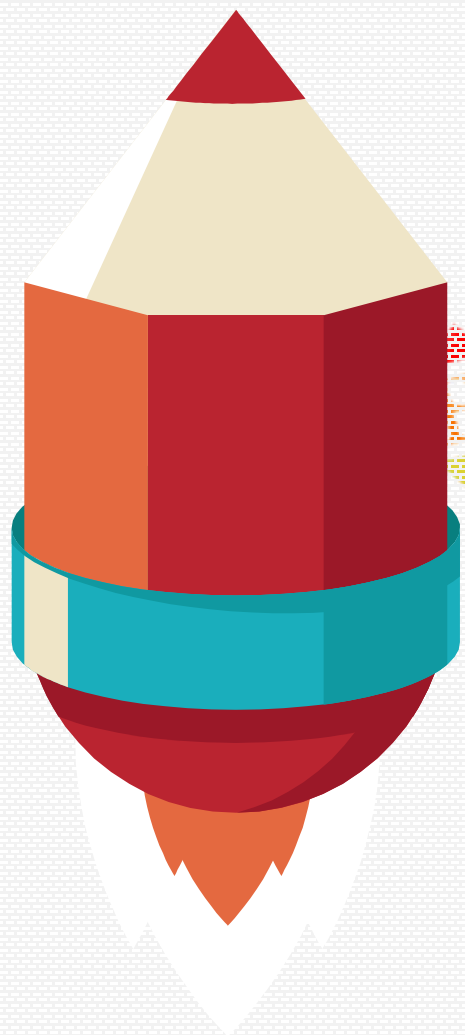




第40课: promise对象的方法

■ 主讲老师: 万章



四知

.then方法

.catch方法

.finally方法

.all/.race方法

.resolve/.reject方法



.then方法

.then方法

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。

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

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。

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。

.then方法的组合状态使用

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

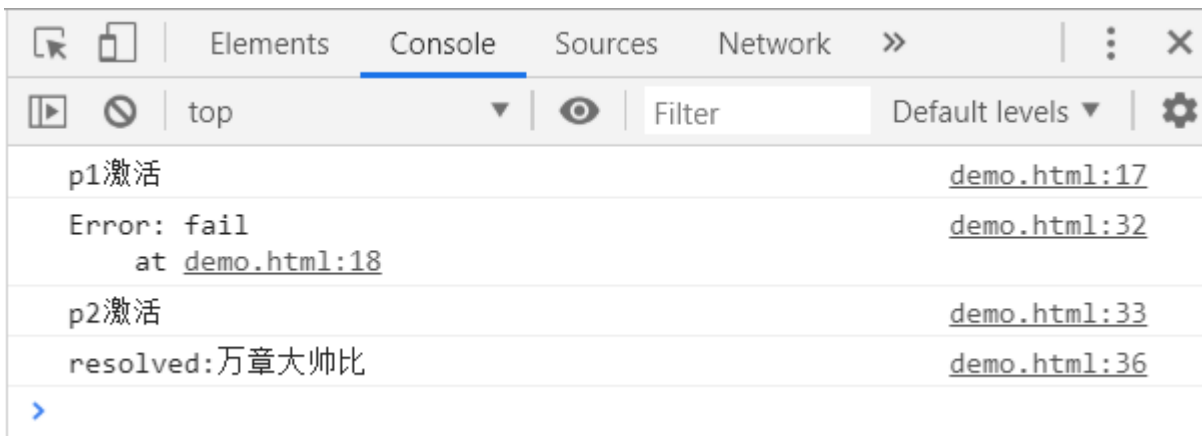
上面代码中，第一个then方法指定的回调函数，返回的是另一个Promise对象。这时，第二个then方法指定的回调函数，就会等待这个新的Promise对象状态发生变化。如果变为resolved，就调用funcA，如果状态变为rejected，就调用funcB。

.then方法的返回值

```
const p1 = new Promise(function (resolve, reject) {
  setTimeout(() => {
    console.log("p1激活");
    reject(new Error('fail'));
  }, 3000)
})

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

p2
  .then(result => {
    console.log(result);
  }, err => {
    console.log(err);
    console.log("p2激活");
    return ("万章大帅比");
  }).then(
  msg => console.log("resolved:" + msg),
  msg => console.log("rejected:" + msg)
)
```



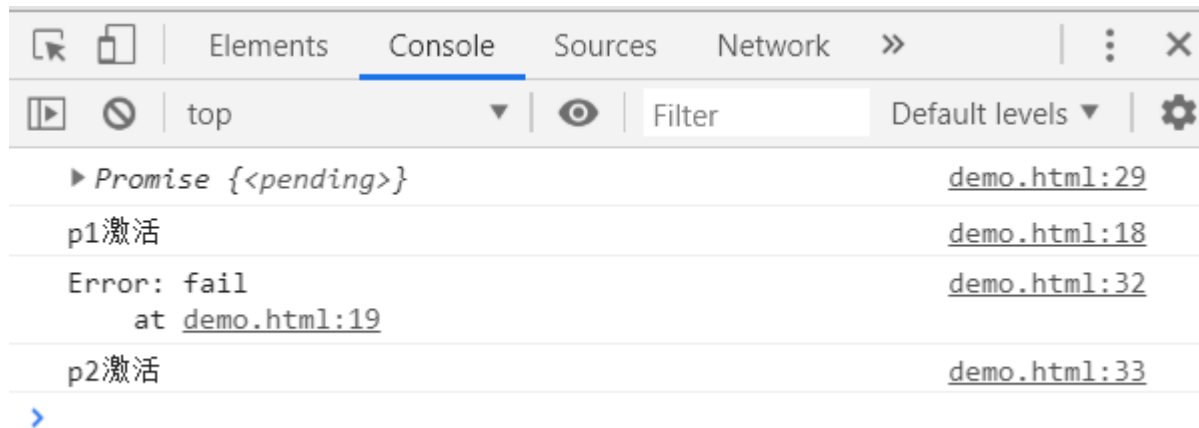
p1激活	demo.html:17
Error: fail at demo.html:18	demo.html:32
p2激活	demo.html:33
resolved:万章大帅比	demo.html:36

.then方法的返回值注意点

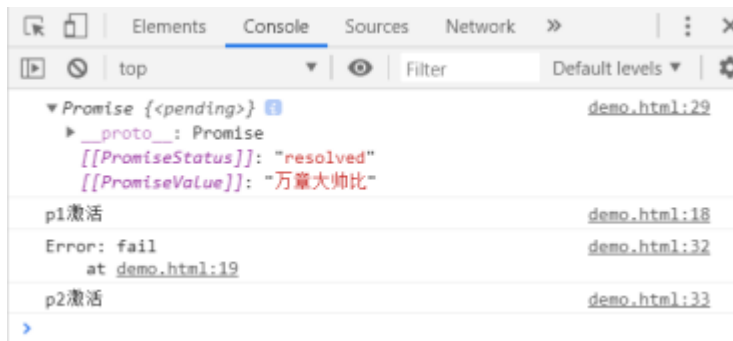
```
const p1 = new Promise(function (resolve, reject) {
  setTimeout(() => {
    console.log("p1激活");
    reject(new Error('fail'));
  }, 3000)
})

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

console.log(p2.then(result => {
  console.log(result);
}, err => {
  console.log(err);
  console.log("p2激活");
  return ("万章大帅比");
})))
```



注意区分同步和异步，同步就是，浏览器从上往下先依次创造p1,p2,并输出了p2.then之后的状态，但记住，此时p2并没有被激活，所以console.log出来的Promise对象就是pending状态



当p1和p2内的回调函数执行后,再点击之前输出的Promise对象，里面的状态就是resolved了,这是因为当我们点开对象时,相当于重新访问了Promise对象当前的数据



.catch方法

.catch方法

Promise.prototype.catch方法是.then(null, rejected)或.then(undefined, rejected)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));
```

等价



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

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

等价



```
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。

.catch方法的触发问题

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

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。

```
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
```

所以当 Promise 状态已经变成resolved，再抛出错误是无效的。

.catch方法的触发问题

```
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  
});
```

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。

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  
  });
```

promise的内部错误

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

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。（即，promise内部的报错不会终止整个js程序的运行，而其他普通函数内一旦报错，程序运行就会终止）

promise的内部错误传递

```
38  const someAsyncThing = function () {
39      return new Promise(function (resolve, reject) {
40          // 下面一行会报错, 因为x没有声明
41          resolve(x + 2);
42      });
43  };
44
45  someAsyncThing().then(function () {
46      return someOtherAsyncThing();
47  }).catch(function (error) {
48      console.log('oh no', error); // oh no [ReferenceError: x is not defined]
49      // 下面一行会报错, 因为 y 没有声明
50      y + 2; //demo.html:48 Uncaught (in promise) ReferenceError: y is not defined
51  }).then(function () {
52      console.log('carry on');
53  });
```

```
oh no ReferenceError: x is not defined      demo.html:48
    at demo.html:41
    at new Promise (<anonymous>)
    at someAsyncThing (demo.html:39)
    at demo.html:45
>
Uncaught (in promise) ReferenceError: y is not defined      demo.html:50
    at demo.html:50
```

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，直接就在函数内报错了，想要解决的话也很简单，在.catch方法的后面再跟上一个catch就可以了



.finally方法

finally方法

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(和try catch finally里面的finally类似)。该方法是 ES2018 引入标准的。

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

右侧代码中，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(() => {})
```



.all/.race方法

.all方法

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

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

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。

.all方法

```
// 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){
  // ...
});
```

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。

其中 resolved函数的参数posts是每个promise的返回结果的json数据组成的数组，每个数组项目都是一个json

.all方法

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]
```

右侧代码中，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。

.race方法

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

```
const 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。

右侧是一个例子，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(console.log)
  .catch(console.error);
```



`.resolve/.reject`方法

.resolve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。

Promise.resolve等价右面的写法。

```
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。如右侧, 并把运行后resolve的参数传递给下一次的then

```
let thenable = {  
  then: function(resolve, reject) {  
    resolve(42);  
  }  
};  
  
let p1 = Promise.resolve(thenable);  
p1.then(function(value) {  
  console.log(value); // 42  
});
```

.resolve方法

(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。

.resolve方法

(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

```
const p = Promise.resolve();  
  
p.then(function () {  
  // ...  
});
```

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

注意右侧代码的输出顺序

setTimeout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。

```
setTimeout(function () {  
  console.log('three');  
}, 0);  
  
Promise.resolve().then(function () {  
  console.log('two');  
});  
  
console.log('one');
```

// one
// two
// three



.resolve方法

注意：上面讲到的all或是race方法的参数如果有不是promise对象的，那么就会用resolve的方法自动转化

.reject方法

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。

```
const thenable = {  
  then(resolve, reject) {  
    reject('出错了');  
  }  
};  
  
Promise.reject(thenable)  
  .catch(e => {  
    console.log(e === thenable)  
  })  
// true
```

注意，Promise.reject()方法的参数，会**原封不动**地作为reject的输出，变成后续方法的参数。这一点与Promise.resolve方法不一致。

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。



基本应用示例

图片加载

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) {  
    const image = new Image();  
    image.onload = resolve;  
    image.onerror = reject;  
    image.src = path;  
  });  
};
```