**Promise全攻略（四.Promise的方法使用）**

**Promise是异步任务同步化的解决方案？本文讲通过以下几方面来介绍Promise：**

1. [Promise的出现为了解决什么问题](https://juejin.cn/post/7000572692201996296)
2. [Promise内部对任务的处理](https://juejin.cn/post/7001471518022565902)
3. [Promise的标准规范](https://juejin.cn/post/7002527759658385444)
4. [**Promise的方法使用**](https://juejin.cn/post/7003232754536022029)
5. [Promise在eventLoop中执行顺序](https://juejin.cn/post/7004769081156960269)
6. [源码地址](https://juejin.cn/post/7005003950923448328)

Promise知识分五个章节来阐述，这篇我们讨论：**Promise的方法使用**。  
前面我们说到Promise出现解决的问题，包括Promise内部的多种状态和标准规范，当这些内部原理功能了解了，就来谈谈**Promise给开发者提供了哪些API和方法**，这也是我们开发最常用到的，也是最接地气的一节。  
先梳理下Promise所有的方法吧：

 实例方法：then()，catch()，finally()。  
静态（属性）方法：all()，race()，resolve()，reject()，ES2021新特性中的any()。

**实例方法**

实例方法是通过实例化后才能调用的方法，就是是说new Promise()后才能执行，实例方法挂载在Promise构造函数的prototype上，new的过程有一步就是将构造函数的prototype指向新创建对象的\_\_proto\_\_上。所以返回的新对象可以沿着**作用域链**找到并调用该方法。

**then()**

then()  方法返回一个 Promise。它最多需要有两个参数：Promise 的成功和失败情况的回调函数。

js

复制代码

**const** p = **new** **Promise**((resolve, reject) => {

**resolve**('Success!');

});

promise1.**then**(value => {

console.**log**(value);

},reason =>{

console.**log**(reason);

});

*// expected output: "Success!"*

**参数**  
1、p.then(onFulfilled,onRejected);``.then接收两个非必传函数参数，onFulfilled成功状态，onRejected失败状态,分别对应Promise中resolve()和reject()。  
2、函数中的传参为Promise传出的值。需要注意的是当.then的参数不为函数是，内部会放弃对该状态的处理，但是并不会产生错误。  
例如：当p为Fulfilled是p.then('\_',season=>{})。  
这时then还会返回一个Promise对象，并将上一次Promise的状态及值放进去。

js

复制代码

**let** p = **new** **Promise**((resolve, reject) => {

**resolve**(2)

})

**let** a = p.**then**('\_', '\_')

a.**then**((value) => {

console.**log**(value);

})

*// expected output: "2"*

4、当然then的返回结果是由then中的回调函数决定的。  
例如：当p为Fulfilled是p.then(value=>{ return XXX })。  
这里有三种情况：

* 回调中抛出异常，.then返回的Promise对象也是rejected的。

js

复制代码

**let** p = **new** **Promise**((resolve, reject) => {

**resolve**(2)

})

**let** result = p.**then**((value) => {

**throw** '有问题了'

}, err => { })

console.**log**(result)

*/\* expected output:*

Promise {<pending>}

\_\_proto\_\_: Promise

[[PromiseStatus]]: "rejected"

[[PromiseValue]]: "有问题了"

\*/

* 回调返回结果是非Promise类型对象，.then返回成功状态的Promise且结果为返回值。

js

复制代码

**let** p = **new** **Promise**((resolve, reject) => {

**resolve**(2)

})

**let** result = p.**then**((value) => {

**return** '123'

}, err => { })

console.**log**(result)

*/\* expected output:*

Promise {<pending>}

\_\_proto\_\_: Promise

[[PromiseStatus]]: "resolved"

[[PromiseValue]]: "123"

\*/

* 回调返回结果是Promise对象，.then返回的Promise是回调返回Promise的状态及结果。

js

复制代码

**let** p = **new** **Promise**((resolve, reject) => {

**resolve**(2)

})

**let** result = p.**then**((value) => {

**return** **new** **Promise**((resolve, reject) => {

**reject**('失败')

})

}, err => { })

console.**log**(result)

*/\* expected output:*

Promise {<pending>}

\_\_proto\_\_: Promise

[[PromiseStatus]]: "rejected"

[[PromiseValue]]: "失败"

\*/

**catch()**

catch()方法返回一个Promise，一个**函数参数**，是**失败**的状态回调。它其实与then()中第二个参数是同一处理方法，内部是调用了then()对onRejected的回调处理。

js

复制代码

*// 第一种情况：*

在异步函数中抛出的错误不会被**catch**或**then**()第二个参数捕获到

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

setTimeout(**function**() {

**throw** 'Uncaught Exception!';

}, 1000);

});

p1.**catch**(**function**(e) {

console.**log**(e); *// 不会执行*

});

*// 第二种情况：*

状态变为成功后，虽然会继续向后执行，但无法捕获错误，这也符合状态一旦确定就无法改变的原则

**var** p2 = **new** **Promise**(**function**(resolve, reject) {

**resolve**()

**throw** 'Uncaught Exception!';

});

p2.**catch**(**function**(e) {

console.**log**(e); *// 不会执行*

});

js

复制代码

**var** p = **new** **Promise**(**function** (resolve, reject) {

**reject**(2)

});

p2.**catch**(() => { console.**log**(1); })

.**then**(() => { console.**log**(2) }, () => { console.**log**(3); });

*//expected output: 1 2*

**finally()**

同样finally()也返回一个Promise，有一个函数参数，但在这个回调中没有Promise的值，无论**成功与否都会调用**，只是操作结束后在这个回调中对一些事物做同一处理。  
例如：在请求前 我们需要一个loading，请求结束后，无论失败与否，都把loading关掉，那关掉的动作就在finally()中执行。  
再来看一个值得注意的点   
由于then方法中的回调没有再return值，所以他的PromiseValue为undefined。  
而finally方法会默认返回这个Promise最终执行结果。

**静态（属性）方法**

**resolve()，reject()**

其实在第三章探讨**状态判断方法**时对[resolve()](https://juejin.cn/post/7002527759658385444)有详细解释，四种传值状态返回不同状态的Promise对象。  
resolve()，reject()也算是语法糖

js

复制代码

**new** **Promise**((resolve, reject) => **resolve**('success')).**then**(() => { }, () => { })

等价与

**Promise**.**resolve**('success').**then**(() => { }, () => { })

**new** **Promise**((resolve, reject) => **reject**('fail')).**then**(() => { }, () => { })

等价与

**Promise**.**reject**('fail').**then**(() => { }, () => { })

**all()**

Promise.all()，返回一个Promise实例，它的参数是接收一个iterable类型（Array,Map,Set）,如果状态是fulfilled，then()的成功回调的参数为一个数组，里面是iterable结果的resolve值。

js

复制代码

**const** p1 = **Promise**.**resolve**(3);

**const** p2 = 42;

**const** p3 = **new** **Promise**((resolve, reject) => {

setTimeout(resolve, 2000, 'foo');

});

**Promise**.**all**([p1, p2, p3]).**then**(values =>console.**log**(values));

*// expected output: 两秒后输出 [3, 42, "foo"]*

1.all()方法会等待所有Promise对象状态都变为resolve时，再触发then()方法。

js

复制代码

**const** p1 = **Promise**.**reject**(3);

**const** p2 = 42;

**const** p3 = **new** **Promise**((resolve, reject) => {

setTimeout(reject, 2000, 'foo');

});

**Promise**.**all**([p1, p2, p3]).**then**(values => console.**log**(values),(err)=>console.**log**('err', err));

*// expected output: 立即输出 err 3*

**Promise**.**all**([p3, p2, p1]).**then**(values => console.**log**(values),(err)=>console.**log**('err', err));

*// expected output: 立即输出 err 2*

2.all()方法中传递的Promise只要有一个状态变为rejected,会立刻调用then()中的onRejcted函数，对错误进行反馈。即便有多个reject也只提示出最早变为rejected状态的值。  
这里引发了一个思考，传进去的Promise是同步调用的还是异步调用的？

js

复制代码

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

setTimeout(resolve, 4000, 'foo');

});

**const** p2 = 42;

**const** p3 = **new** **Promise**((resolve, reject) => {

setTimeout(resolve, 2000, 'foo');

});

**Promise**.**all**([p1, p2, p3]).**then**(values =>console.**log**(values));

*//是异步调用 4秒后输出 ["foo", 13, "foo"]*

让我们看看他的状态的几种情况：

js

复制代码

**var** resolvedPromisesArray = [**Promise**.**resolve**(33), **Promise**.**resolve**(44)];

**var** p = **Promise**.**all**(resolvedPromisesArray);

console.**log**(p);

setTimeout(**function**(){

console.**log**('next tick');

console.**log**(p);

});

*// Promise { <state>: "pending" }*

*// next tick*

*// Promise { <state>: "fulfilled", <value>: Array[2] }*

1，如果传入了Promise的集合，状态执行是**异步**的。在当前循环中p的状态不会立即变为fulfilled。reject也一样

js

复制代码

**var** p = **Promise**.**all**([]);

**var** p2 = **Promise**.**all**([123, "hello"]);

console.**log**(p);

console.**log**(p2)

setTimeout(**function**(){

console.**log**('next tick');

console.**log**(p2);

});

*// Promise { <state>: "fulfilled", <value>: Array[0] }*

*// Promise { <state>: "pending" }*

*// next tick*

*// Promise { <state>: "fulfilled", <value>: Array[2] }*

2.当传入了一个空的可迭代的iterable时，状态的改变是**同步**的，会立即改变的。

**race()**

race()方法与all()方法的区别在于，then()的onResolve回调的参数**不是数组形式**的，传了多个Promise，但最终返回的是最先执行完毕的Promise。then()中的回调也与这个Promise的状态对应。

js

复制代码

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

setTimeout(resolve, 500, 'p1 success');

});

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

setTimeout(resolve, 100, 'p2 success');

});

**Promise**.**race**([p1, p2]).**then**((value) => {

console.**log**(value);

});

*// expected output: "p2 success"*

**any()**

这个方法还属于实验阶段的（2021年9月），没被所有浏览器支持。  
返回一个Promise对象，参数也是传递一个iterable合集。  
1.只要其中的一个 promise 成功，就返回那个已经成功的 promise，并且终止。  
2.如果所有Promise都失败了，那就调用then()的onFulfilled回调。