**Promise 状态**

**状态的特点**

Promise 异步操作有三种状态：pending（进行中）、fulfilled（已成功）和 rejected（已失败）。除了异步操作的结果，任何其他操作都无法改变这个状态。

Promise 对象只有：从 pending 变为 fulfilled 和从 pending 变为 rejected 的状态改变。只要处于 fulfilled 和 rejected ，状态就不会再变了即 resolved（已定型）。

### 状态的缺点

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）。

## then 方法

then 方法接收两个函数作为参数，第一个参数是 Promise 执行成功时的回调，第二个参数是 Promise 执行失败时的回调，两个函数只会有一个被调用。

### then 方法的特点

在 JavaScript 事件队列的当前运行完成之前，回调函数永远不会被调用。

const p = new Promise(function(resolve,reject)

{ resolve('success'); }); p.then(function(value){ console.log(value); });

console.log('first');

// first

// success

通过 **.then** 形式添加的回调函数，不论什么时候，都会被调用。

then 方法将返回一个 resolved 或 rejected 状态的 Promise 对象用于链式调用，且 Promise 对象的值就是这个返回值。

**then 方法注意点**

简便的 Promise 链式编程最好保持扁平化，不要嵌套 Promise。

注意总是返回或终止 Promise 链。

const p1 = new Promise(function(resolve,reject){ resolve(1); }).then(function(result) {   p2(result).then(newResult => p3(newResult)); }).then(() => p4());

创建新 Promise 但忘记返回它时，对应链条被打破，导致 p4 会与 p2 和 p3 同时进行。

大多数浏览器中不能终止的 Promise 链里的 rejection，建议后面都跟上 **.catch(error => console.log(error));**

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。

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

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);

});

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，

**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 });

## 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。

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。

## 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'));

### Generator 函数组成

Generator 有两个区分于普通函数的部分：

* 一是在 function 后面，函数名之前有个 \* ；
* 函数内部有 yield 表达式。

### 执行机制

调用 Generator 函数和调用普通函数一样，在函数名后面加上()即可，但是 Generator 函数不会像普通函数一样立即执行，而是返回一个指向内部状态对象的指针，所以要调用遍历器对象Iterator 的 next 方法，指针就会从函数头部或者上一次停下来的地方开始执行。

**return 方法**

return 方法返回给定值，并结束遍历 Generator 函数。

return 方法提供参数时，返回该参数；不提供参数时，返回 undefined 。

**yield\* 表达式**

yield\* 表达式表示 yield 返回一个遍历器对象，用于在 Generator 函数内部，调用另一个 Generator 函数。

### 语法async

async 是 ES7 才有的与异步操作有关的关键字，和 Promise ， Generator 有很大关联的

async function name([param[, param[, ... param]]]) { statements }

* name: 函数名称。
* param: 要传递给函数的参数的名称。
* statements: 函数体语句。

### 返回值

async 函数返回一个 Promise 对象，可以使用 then 方法添加回调函数。

async function helloAsync(){ return "helloAsync"; }

console.log(helloAsync()) // Promise {<resolved>: "helloAsync"}

helloAsync().then(v=>{ console.log(v); // helloAsync })

async 函数中可能会有 await 表达式，async 函数执行时，如果遇到 await 就会先暂停执行 ，等到触发的异步操作完成后，恢复 async 函数的执行并返回解析值。

await 关键字仅在 async function 中有效。如果在 async function 函数体外使用 await 你只会得到一个语法错误。

function testAwait(){

return new Promise((resolve) => {

setTimeout(function(){

console.log("testAwait");

resolve(); }, 1000); });

}

async function helloAsync(){

await testAwait();

console.log("helloAsync"); }

helloAsync(); // testAwait // helloAsync

## await

await 操作符用于等待一个 Promise 对象, 它只能在异步函数 async function 内部使用。

### 语法

[return\_value] = await expression;

* expression: 一个 Promise 对象或者任何要等待的值。

### 返回值

返回 Promise 对象的处理结果。如果等待的不是 Promise 对象，则返回该值本身。

如果一个 Promise 被传递给一个 await 操作符，await 将等待 Promise 正常处理完成并返回其处理结果。

function testAwait (x) {

return new Promise(resolve => { setTimeout(() => { resolve(x); }, 2000); });

}

async function helloAsync() {

var x = await testAwait ("hello world");

console.log(x); }

helloAsync ();

正常情况下，await 命令后面是一个 Promise 对象，它也可以跟其他值，如字符串，布尔值，数值以及普通函数。

await针对所跟不同表达式的处理方式：

* Promise 对象：await 会暂停执行，等待 Promise 对象 resolve，然后恢复 async 函数的执行并返回解析值。
* 非 Promise 对象：直接返回对应的值。