function send(url, callback) {

// ajax请求，请求成功了回调

// callback();

}

1完事了执行2，2完事了执行3，就这样就是回调地狱

send('url1', () => {

send('url2', () => {

send('url3', () => {

send('url4', () => {

// ......

});

});

});

});

上面其实就是一种异步编程解决方案，除了回调之外，还可以采用事件通知机制、发布订阅模式，也可以使用ES6中新增的promise

**Promise**

Promise：是ES6中新增的异步编程解决方案，体现在代码中它是一个对象，可以通过Promise构造函数来实例化。

1、如果直接这样写，就会报错，原因是promise要接收一个函数

var p = new Promise();

console.log(p);

2、这样就可以得到一个promise，它的状态是pending阶段，值是undefined

var p = new Promise(() => {});

console.log(p);

3、这样发现，会先打印‘实例化了’，再打印p。这说明promise里的函数是立刻就会执行的，所以，一般情况下，我们也不会直接这么写，而是用一个函数把promise包起来，点击时调用或者什么事件调用

var p = new Promise(() => {

console.log('实例化了。。。')

});

console.log(p);

4、状态：

Pending(进行中) ===> Resolved(已完成)

Pending(进行中) ===> Rejected(已失败)

现在上面的状态就是pending，如何让它进到成功或者失败的状态呢？

5、成功状态，只要在函数中调用resolve即可

var p = new Promise((resolve, reject) => {

resolve();

});

console.log(p);

这个时间状态就是resolved，值是undefined，因为resolve()没有传参

1. 失败状态，只要调用reject()

var p = new Promise((resolve, reject) => {

reject();

});

console.log(p);

这个时间状态就是rejected，值是undefined

1. 走了成功就不会再走失败

var p = new Promise((resolve, reject) => {

resolve();

reject();

console.log('3333');

});

console.log(p);

此时状态就是成功，不会变成失败了

而且，会先打印3333，这说明，成功或失败并不会阻止后续代码的执行。所以，一般我们会在这里加return，后续代码有不有都没有意义了。

8、成功或失败及参数

var p = new Promise((resolve, reject) => {

// 一般在这里我们会做ajax请求，哪就有请求成功或失败

if (true) {

resolve('成功');

} else {

reject('失败');

}

});

console.log(p);

成功状态就是resolved，value就是 "yes"

失败状态就是rejected，value就是 "no"

哪么这两个参数具体传到哪里去了呢？

10、参数

var p = new Promise((resolve, reject) => {

if (true) {

resolve('yes');

} else {

reject('no');

}

});

p.then(

(data) => console.log(data), // yes

(err) => console.log(err) // no

);

Promise的实例p还有一个then方法，这个方法执行的时候，会接收两个参数，一个是成功的回调，一个是失败的回调，参数分别在这个两个函数里面。

11、promise实例下面有then和catch方法，分别是成功要执行和失败执行。

var p = new Promise((resolve, reject) => {

if (0) {

resolve('yes');

} else {

reject('no');

}

});

p.then((data) => {

console.log(data);

});

p.catch((err) => {

console.log(err);

});

1. then的返回值，then执行后会再返回一个全新的promise实例

如果在then中不手动return，也会返回一个promise实例，而且是成功的状态，value是undefined

var p = new Promise((resolve, reject) => {

if (true) {

resolve('yes');

} else {

reject('no');

}

});

// p的then调用，返回一个全新的promise，在这个里面return出的111，会做为p2成功的参数

// 因为这里return回来的是一个非promise，所以它就会做为p2的成功的参数

var p2 = p.then(() => {

return 111;

});

p2.then((data) => {

console.log(data); // 111

});

返回一个promise，且是成功的状态

var p2 = p.then(() => {

// 返回一个promise，并且是成功的，则p2的状态就是成功的

return new Promise((res, rej) => {

res('p2成功');

});

});

console.log(p2); // 成功状态

返回一个promise，且是失败的状态

var p2 = p.then(() => {

// 返回一个promise，并且是失败的，则p2的状态就是失败的

return new Promise((res, rej) => {

rej('p2失败');

});

});

console.log(p2); // 失败状态

总结：then的返回值共有两种情况：

一是返回非promise实例，则会返回一个全新的promise，且这个新的promise是成功的状态，成功状态的参数就是返回的这个值，如果没有return 则是undefined

二是返回一个全新的promise实例，这时就依据这个新的promise实例的状态来决定是成功还是失败。

总之：then会返回一个新的promise实例，至于是成功还是失败，看返回的是什么

思考：以下打印的是？

var p = new Promise((res, rej) => {

res('成功');

});

P

.then(() => {

console.log(1);

})

.then(() => {

console.log(2);

})

.then(() => {

console.log(3);

});

打印1 2 3，为什么2 3也会打印出来呢，因为打印1了之后，其实是默认返回了一个promise，且是成功的状态，值是undefined。哪么因为是成功的状态，所以下面的哪个then又会执行，执行后又会返回一个新的promise，就这样一直进行下去。

1. catch

var p = new Promise((res, rej) => {

res('成功');

});

P

.then(() => {

console.log(1);

})

.then(() => {

console.log(2);

console.log(a); // 访问a，会报错

})

.then(() => {

console.log(3);

})

.catch(err => {

console.log(err); // 这样catch来捕获上面a的报错，不会导致代码终断

});

总结，当像上面这样链式调用时，只要在最后写一个catch，就可以捕获任何一个then的报错

15、finally 不管成功或失败，都会执行这句话

var p = new Promise((res, rej) => {

res('成功');

});

p

.then(() => {

console.log(1);

})

.then(() => {

console.log(2);

// 访问a，会报错，这样用下面的catch来捕获

console.log(a);

})

.then(() => {

console.log(3);

})

.catch(err => {

console.log(err); // 捕获上面的失败

})

.finally(() => {

// finally的作用是：不论成功失败，都会执行到这里来

console.log('执行完了');

});

16、两个静态方法：

Promise.resolve(成功的参数)

Promise.reject(失败的参数)

把参数包装成成功或者失败的promise实例

var p = new Promise((res, rej) => {

res(); // 成功

});

p

.then(() => {

console.log(1);

// 这里默认返回了一个promise，成功且值是undefined

})

.then(() => {

// return new Promise((res, rej) => {

// res(200);

// });

return Promise.resolve(200); // 默认返回一个成功的promise，可以替代上面返回

})

.then((data) => {

console.log(data); // 打印成功的

})

.then(() => {

// return new Promise((res, rej) => {

// rej('失败');

// });

return Promise.reject('失败'); // 返回一个失败的promise，可以替代上面哪句话

})

.catch((err) => {

console.log(err); // 捕获失败

});

总结：这两个方法是promise下面的静态方法，都是返回一个promise实例，一个是成功，一个是失败。只是写法简洁点，可以替代new Promise().别的用处不大

1. 两个静态方法：

Promise.all(); 一起完成

Promise.race(); 赛跑，看谁最先状态改变，就以谁的状态为准

var p1 = new Promise((res, rej) => {

setTimeout(() => res('a'), 1000);

});

var p2 = new Promise((res, rej) => {

setTimeout(() => res('b'), 1500);

});

var p3 = new Promise((res, rej) => {

setTimeout(() => res('c'), 500);

});

// Promise.all(); 一起完成

// var p = Promise.all([p1, p2, p3]);

// p.then(data => {

// console.log(data); // ["a", "b", "c"] 当三个promise都执行完了后，才打印出这个数组

// });

// Promise.race(); 赛跑，看谁最先状态改变，就以谁为准

var p = Promise.race([p1, p2, p3]);

p.then(data => {

console.log(data); // c 赛跑，因为第三个最快

});

**实例化：**

// 实例化Promise，它接收一个函数做为参数，这个函数再接收两个参数，分别是resolve和reject，分别代表异步成功要执行的回调函数和异步失败要执行的回调函数。（这两个名字是随意的）

let p = new Promise(function(resolve, reject) {

if ( /\* 异步操作成功 \*/ ) {

resolve(data)

} else {

reject(err)

}

});

图片，加载图片是一个异步，我们用加载图片来演示，如果成功，则会执行onload，失败则会执行onerror

let imgs = [

'https://timgsa.baidu.com/timg?image&quality=80&size=b9999\_10000&sec=1527252256191&di=7b25587cff2e776e89552261e850e286&imgtype=0&src=http%3A%2F%2Fimg4.duitang.com%2Fuploads%2Fitem%2F201504%2F12%2F20150412H1301\_ZH4Mw.jpeg',

'https://timgsa.baidu.com/timg?image&quality=80&size=b9999\_10000&sec=1527252256190&di=b1746bacd6c437c00df60858c341a083&imgtype=0&src=http%3A%2F%2Fimg5q.duitang.com%2Fuploads%2Fitem%2F201504%2F12%2F20150412H5248\_yTPMx.jpeg',

'https://timgsa.baidu.com/timg?image&quality=80&size=b9999\_10000&sec=1527252256190&di=c5d4e0b24bc28aff9972391241f60459&imgtype=0&src=http%3A%2F%2Fpic.qqtn.com%2Fup%2F2017-12%2F2017120610233753434.jpg'

];

**三个原型方法：**

Promise.prototype.then();

Promise.prototype.catch();

Promise.prototype.finally();

分别用来处理promise结束结果的函数。

then可以处理执行成功的回调

catch可以处理执行失败的异常

finally则是不论成功或失败，都会执行

这是加载一张图片，如果要加载多张图片呢，都加载成功了，一起展示出来怎么办？（注意：promise内部封装的是异步操作，而promise本身它是同步操作。所以，我们一般把它写成一个函数的形式，让它返回一个promise对象。）

function loadImg(url) {

let p = new Promise(function(resolve, reject) {

let oImg = new Image();

oImg.src = url;

oImg.onload = function() {

resolve(this);

};

oImg.onerror = function() {

reject(new Error('图片加载失败'));

}

});

return p;

}

**常用的静态方法**

Promise.all();

可以将多个promise实例包装成一个新的promise实例，它在使用的时候，接收一个数组，这个数组里面就是多个promise实例组成的。

它分两种情况：

1、当所有promise实例的状态都变成resolved，promise.all的状态才会变成resolve，此时返回值组成一个数组，传递给then中的resolve函数

2、只要其中有一个被rejected，promise.all的状态就变成rejected，此时第一个被rejectr 的实例的返回值，会传递给reject的回调函数。

// 调用Promise.all方法

let imgAll = Promise.all([

loadImg(imgs[0]),

loadImg(imgs[3]),

loadImg(imgs[2])

]);

// 成功或失败

imgAll.then(function(arr) {

arr.forEach(item => document.body.appendChild(item))

}).catch(function(err) {

console.log(err);

});

Promise.race()

与all方法类似，也可以将多个Promise实例包装成一个新的Promise实例

不同的是，all时大Promise的状态由多个小Promise共同决定，而race时由第一个转变状态的小Promise的状态决定，第一个是成功态，则转成功态，第一个失败态，则转失败态

Promise.resolve()

用来将一个对象转换为promise对象的。它的使用方式有三种

1. 接收一个promise实例，将不做任何修改，原封不动地返回这个实例
2. 接收一个对象。
3. 接收一个基本数据类型（或者不传参），返回一个新的promise对象，并且这个promise的状态是一个成功时候的状态。参数就会成为执行成功的回调函数的参数。

**then的链式用**

这里返回基本类型的值，会被包装成一个promise的实例，并且是成功的状态，成功状态的value值是返回的参数

let p = new Promise(function(resolve, reject) {

setTimeout(function() {

resolve(1);

}, 1000);

});

p

.then(function(data) {

console.log(data); // 1

return '123';

})

.then(function(data) {

console.log(data); // 123

return '456'

})

.then(function(data) {

console.log(data); // 456

});

这里返回的是promise的情况，但是每个promise都是成功的状态（p这个函数里面写的成功），所以下面的then都会执行，如果失败，就要用catch来捕获错误。如下，就可以做依次发送的ajax了

function p(num) {

return new Promise(function(resolve, reject) {

setTimeout(function() {

resolve(num);

}, 1000);

});

};

p(1)

.then(function(data) {

console.log(data); // 1

return p(2);

})

.then(function(data) {

console.log(data); // 2

return p(3);

})

.then(function(data) {

console.log(data); // 3

});

**面试题：**以下代码打印的结果是

console.log(1);

setTimeout(() => {

console.log(3);

});

setTimeout(() => {

console.log(4);

});

var p = new Promise((res, rej) => {

console.log(5);

res();

});

p.then(() => {

console.log(6);

});

console.log(2);

js里的任务分为宏任务和微任务，具体分析上面：

第一轮，从上到下执行，定时器会把任务推到下一轮去。本轮的宏任务是1 5 2。当本轮宏任务完成之后，会再去看一下微任务，而p.then是微任务，所以6就执行了

第二轮，执行定时器推过来的宏任务

参考：https://juejin.im/post/59e85eebf265da430d571f89