# Promise

## 什么是promise



本意是承诺。

Promise 是异步编程的一种解决方案，比传统的解决方案——回调函数和事件——更合理和更强大。它由社区最早提出和实现，ES6将其写进了语言标准，统一了用法，原生提供了promise对象。

所谓promise，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上说，Promise 是一个对象，从它可以获取异步操作的消息。Promise 提供统一的 API，各种异步操作都可以用同样的方法进行处理。

如何理解：

* 没有异步就不需要promise。
* Promise本身不是异步，只是我们去编写异步代码的一种方式

## promise的典型用法

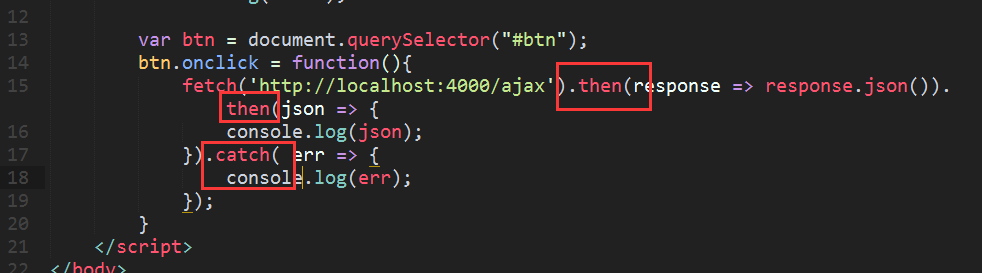
在学习promise的时候，一定要站在两个角度看待promise：

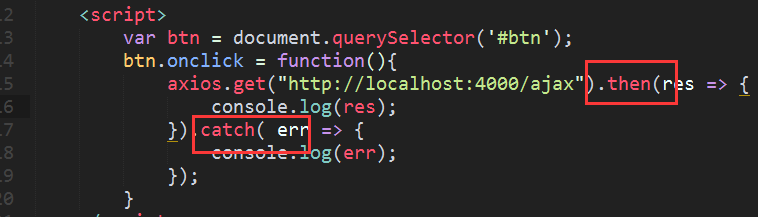
* 站在使用者的角度
* 站在定义者的角度

在如下几个地方，都必须使用promise的写法：

* Fetch
* Axios

典型写法：



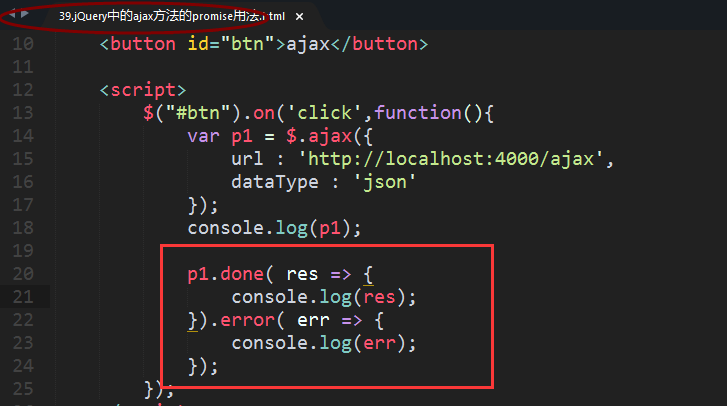


特点是：

* then方法通常是表示异步操作成功时的回调，catch方法通常是表示异步操作失败时的回调
* 在调用的时候then在前后，catch在后
* then方法可以调用多次，前一个then的返回值，会作为后一个then的参数。
* 支持链式调用

其它的一些异步操作，我们也可以使用promise的用法。

比如，jquery中的ajax方法，也支持promise的用法，如下：



注意：和fetch/axios不同的是，此处使用done方法来替代then方法。

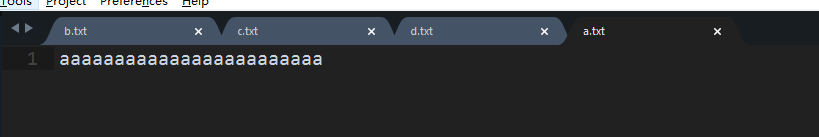
## 为何需要promise

再次声明：

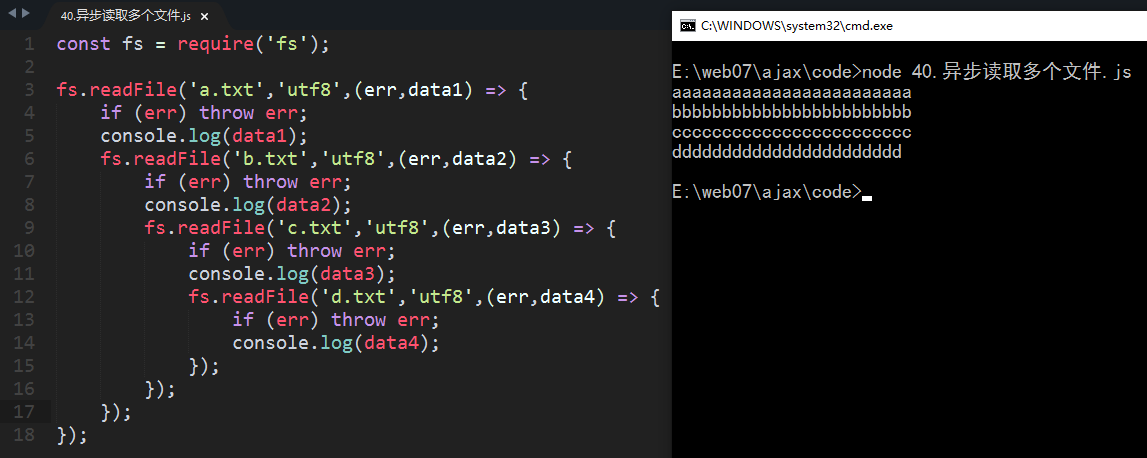
* 没有异步，就不需要promise
* Promise本身不是异步的，只是编写异步代码的一种优雅方式。

需要来看一段典型的异步代码 --- 异步读取多个文件

首先，创建4个文本文件，如下：



编写代码，使用node的fs模块来读取文件，如下：



功能没有任何问题，但是这个写法比较讨厌--- 这就是**臭名昭著的回调地狱**（callback hell）。

针对这个写法，非常的糟糕，需要有一种机制，让这个写法变得更加优雅一些。

所以，在前端社区中，就有一些牛人，纷纷提供了一些新的写法，以避免回调地狱。

后来，es6就在这些基础之上，将Promise直接纳入的规范。从而成为es6中的标准用法。

## promise A+规范

由于promise本身出自于民间，就有很多不同的实现版本。

Es6将promise纳入自己规范的时候，也遵循了一个相应的标准 -- Promise A+规范。

将其归纳为4321规范。

* 4：4大术语
* 3：3种状态
* 2：2种事件
* 1：1个对象

### 4大术语

一定要结合异步操作来理解。

既然是异步，这个操作需要有个等待的过程，从操作开始，到获取结果，有一个过程的。

**解决（fulfill）**：指一个 promise 成功时进行的一系列操作，如状态的改变、回调的执行。虽然规范中用 fulfill 来表示解决，但在后世的 promise 实现多以 **resolve** 来指代之。

**拒绝（reject）**：指一个 promise 失败时进行的一系列操作。

**终值（eventual value）**：所谓终值，指的是 promise 被解决时传递给解决回调的值，由于 promise 有一次性的特征，因此当这个值被传递时，标志着 promise 等待态的结束，故称之终值，有时也直接简称为值（value）。

**据因（reason）**：也就是拒绝原因，指在 promise 被拒绝时传递给拒绝回调的值。

### 3种状态

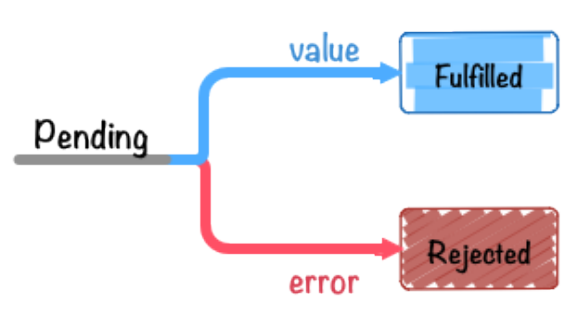
在异步操作中，当操作发出时，需要处于**等待状态**。

当操作完成时，就有相应的结果，结果有两种：

* 成功了
* 失败了

一共是3种状态，如下：

* 等待态（Pending）
* 执行态（Fulfilled）
* 拒绝态（Rejected）



针对每一种状态，有一些规范：

**等待态（Pending）**

处于等待态时，promise 需满足以下条件：

* 可以迁移至执行态或拒绝态

**执行态（Fulfilled）**

处于执行态时，promise 需满足以下条件：

* 不能迁移至其他任何状态
* 必须拥有一个不可变的终值

**拒绝态（Rejected）**

处于拒绝态时，promise 需满足以下条件：

* 不能迁移至其他任何状态
* 必须拥有一个不可变的据因

### 2种事件

针对3种状态，只有如下两种转换方向：

* pending --> fulfilled
* pendeing --> rejected

在状态转换的时候，就会触发事件。

如果是pending --> fulfiied，就会触发onFulFilled事件

如果是pendeing --> rejected，就会触发onRejected事件

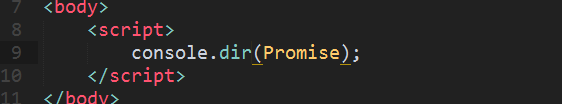
### 1个对象

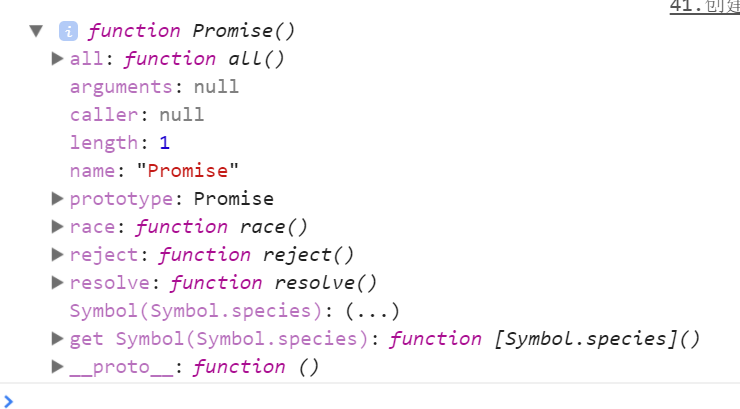
就是指promise对象

## promise的基本用法

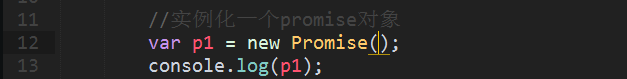
### 实例化Promise对象

在es6中，本身提供了一个Promise的构造器，如下：

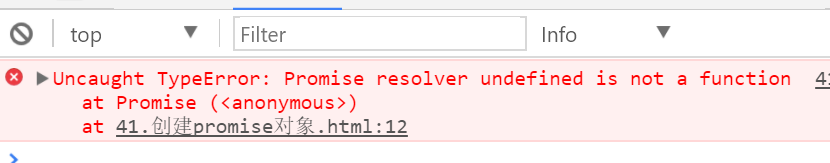




可以使用new 直接创建一个，如下：



执行，如下：

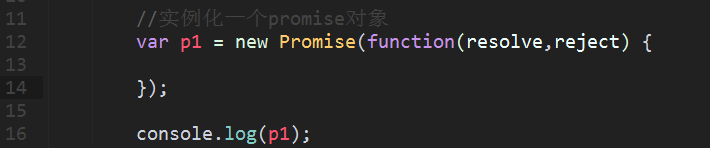


之所以报错，是因为这个Promise构造器，必须要传递一个参数。

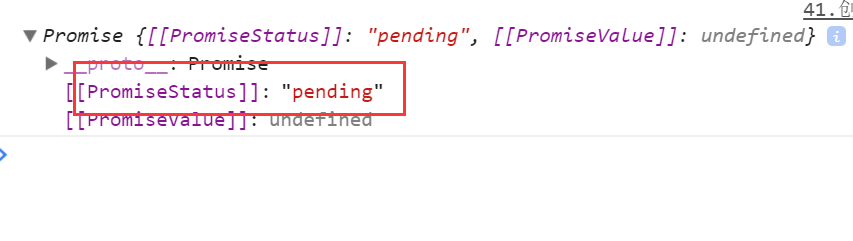
构造器的参数，是一个回调函数，包含两个参数：

* resolve
* reject

编写如下：



再次查看，如下：



说明，我们已经创建好了Promise对象，当前这个对象处于 pending 状态。

回调函数中的两个参数，其作用就是用于转换状态：

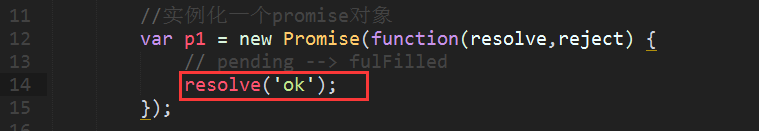
* resolve，将状态从pending --> fullFilled
* reject，将状态从pending --> rejected

直接使用函数调用的方式来进行转换，在转换的时候必须要传递相应参数

Resolve函数的参数，就是指 终值（value）

Reject函数的参数，就是指 据因（reason）

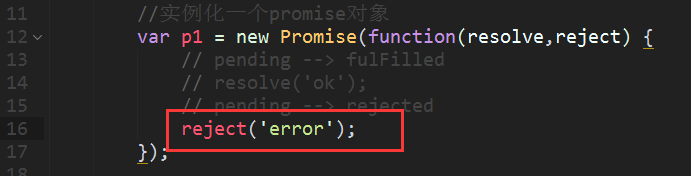
编写代码如下：



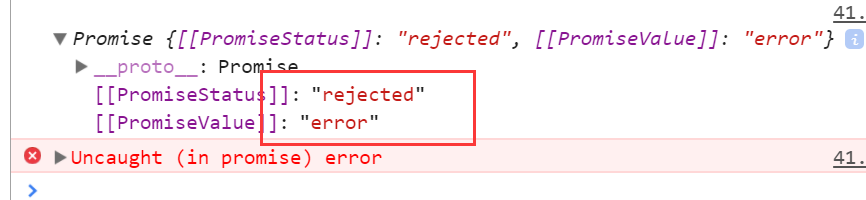
查看如下：



其中，resolved 就是 fulFilled



查看，如下：

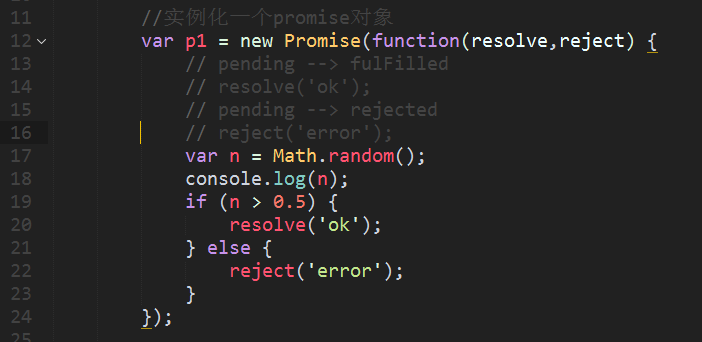


关于Promise的创建工作，已经完成了。

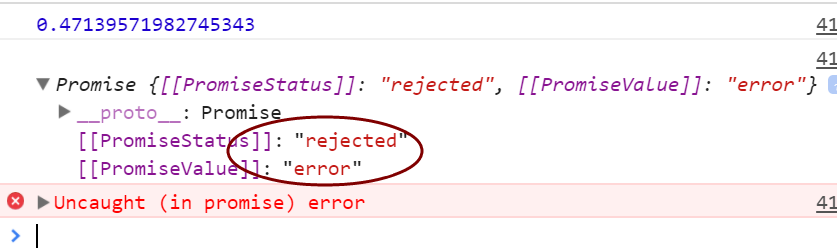
在创建promise对象，只需要根据需求，转换状态即可。无非就是调用两个函数：

* resolve，传递value
* reject，传递reason

比如，可以编写如下代码：



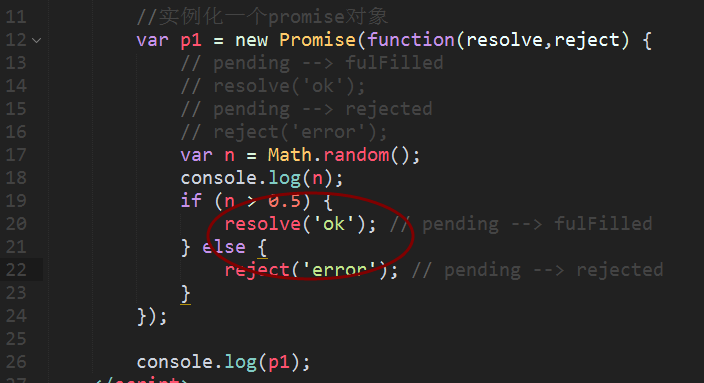
执行，如下：





### then方法

思考，针对刚才的代码：



如何获取这里的终值 ok 或者 据因 error呢？

此时，就需要 利用 事件机制。

针对3种状态，只有如下两种转换方向：

* pending --> fulfilled
* pendeing --> rejected

**在状态转换的时候，就会触发事件。**

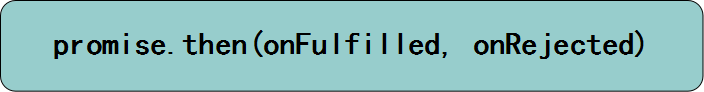
如果是pending --> fulfiied，就会触发onFulFilled事件

如果是pendeing --> rejected，就会触发onRejected事件

在调用resolve方法或者reject方法的时候，就一定会触发事件。

需要注册onFulFilled事件 和 onRejected事件

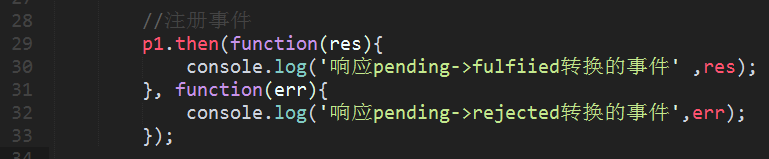
针对事件的注册，Promise对象提供了then方法，如下：



针对 onFulFilled，会自动提供一个参数，作为终值（value）

针对 onRejected，会自动提供一个参数，作为据因（reason）

注册事件，如下：



再次调用：



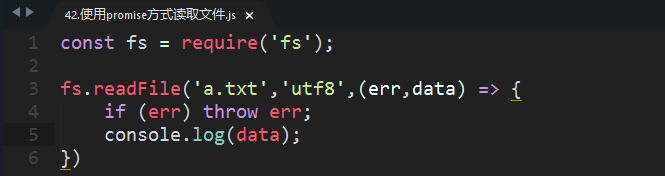


### 关于promise的典型定义

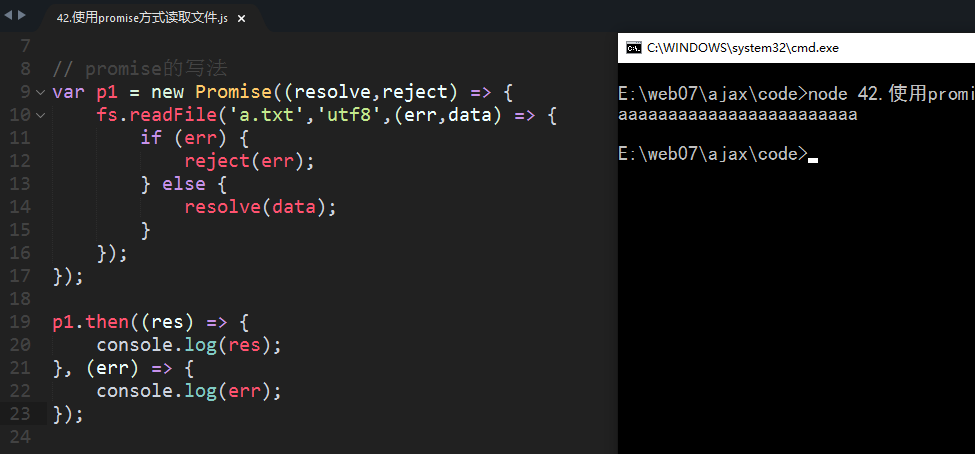
Promise主要是解决异步代码的编写方式。

**案例1：读取文件操作**

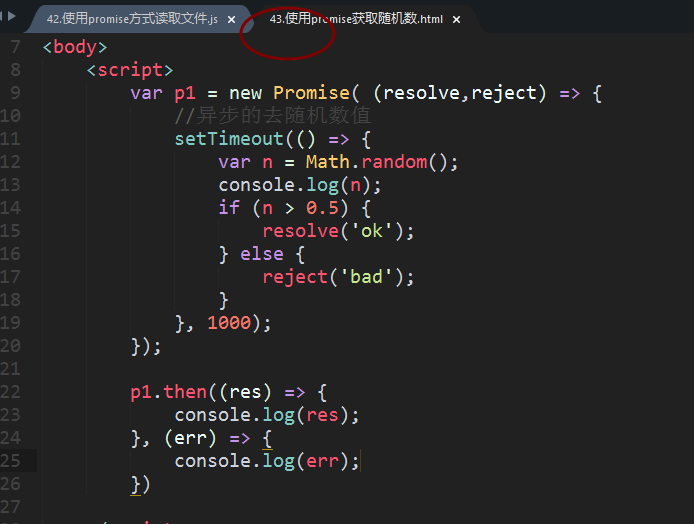
原来的写法，如下：



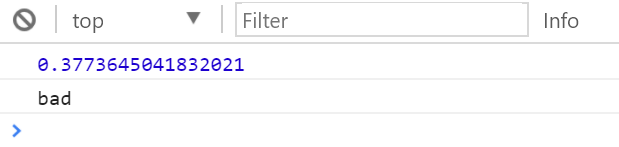
使用promise的写法：

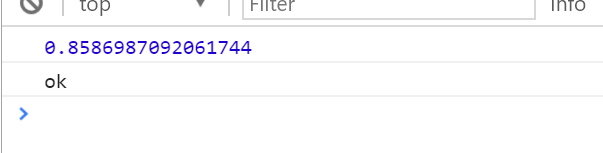


案例2：根据随机数返回结果



执行，如下：

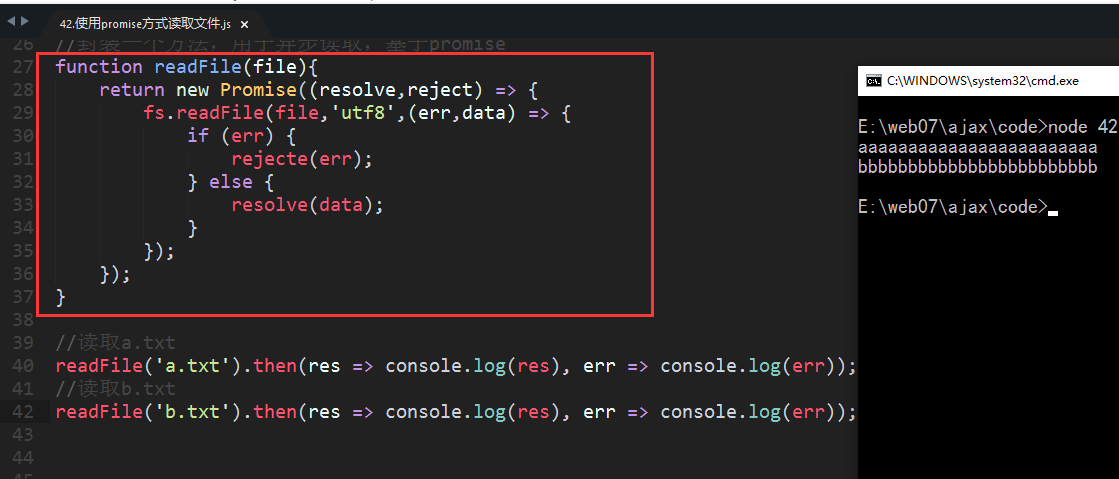




**真正的promise定义方式**

回到读取文件案例，需要再读取b.txt，如何实现？

需要将读取文件的操作，封装为一个函数，如下：



在开发的时候，我们定义promise对象，必须是要作为函数的返回值。

任务：

* 分页的各种形式
* 可以使用iscroll插件、minirefresh 实现移动端的上拉加载，下拉刷新
* Promise需要好好消化一下

### catch方法

Promise对象，有一个then，如下：

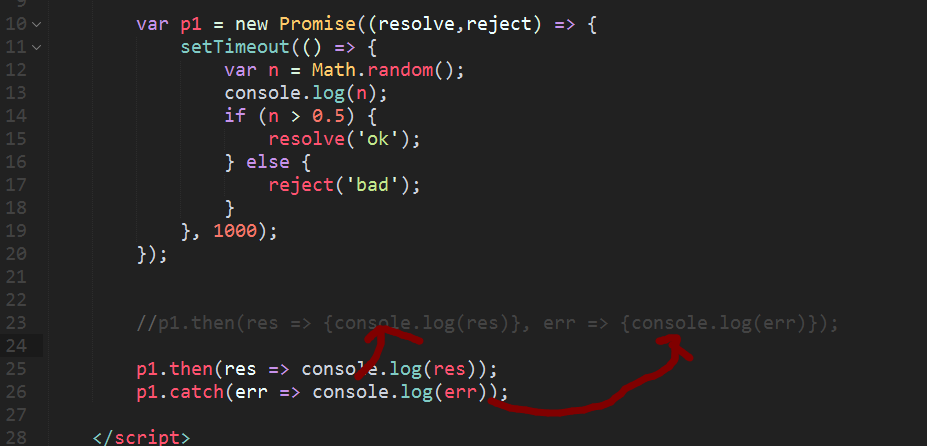
promise.then(onFulfilled, onRejected)

需要写两个回调。其中第二个回调表示 从pending ---> rejected 时的回调。

由于这种写法，辨识度不高。

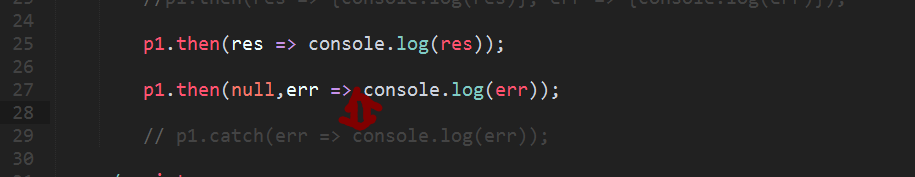
Promise就提供了一个catch方法，用于注册 onRejected回调。

使用如下：

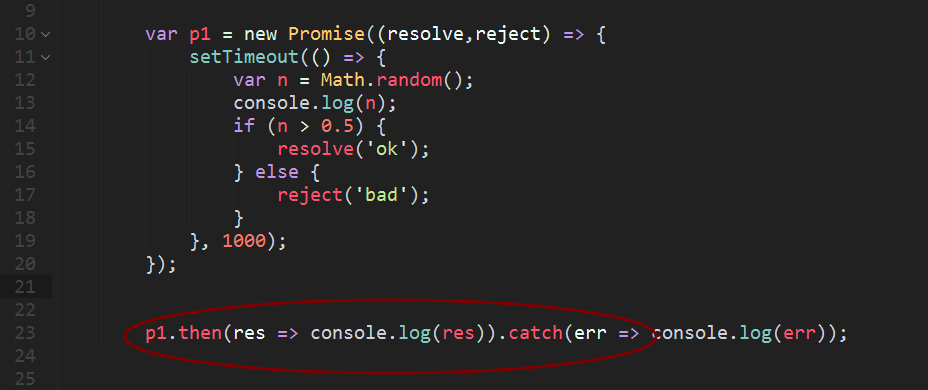


在这里要明白两件事情：

* catch其实是then的简写，then(null，callback)，如下：



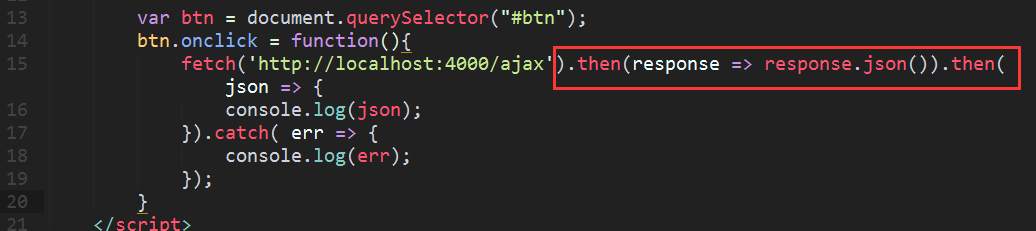
* then方法调用之后，仍然返回的是promise对象，所以可以链式调用，如下：



所以，我们在使用promise对象时，一般这么描述，异步操作成功的时候，走then，失败的时候就走catch。

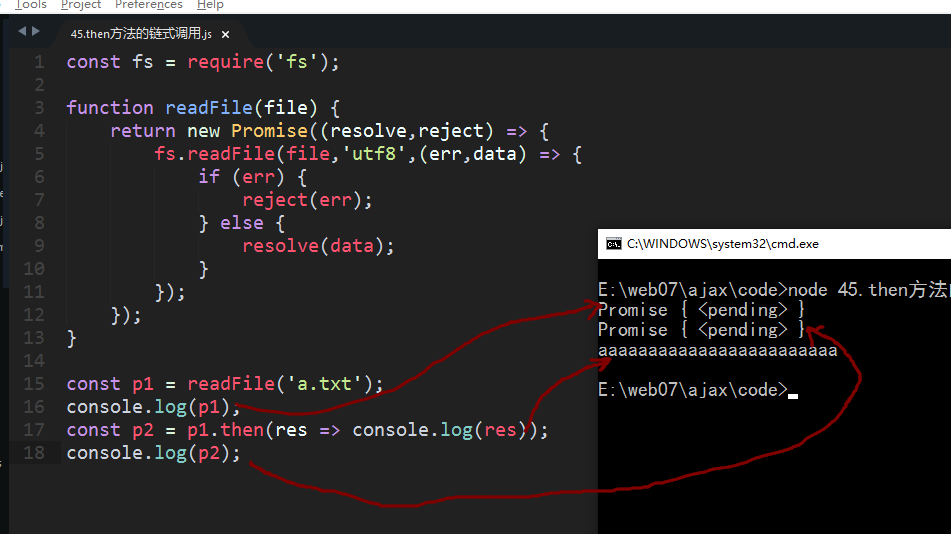
### then方法的链式调用

在fetch的代码中，有如下写法：

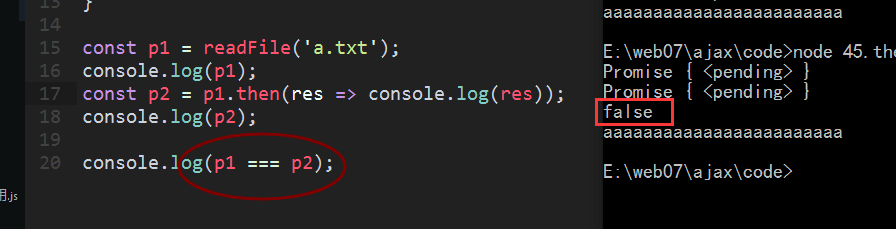


也就说，then方法后面继续的调用then方法，这就是then的链式调用。

编写代码如下：



**重点是p2，then方法返回的是一个promise对象，但它是一个新的promise对象，不再是原来的promise对象**，如下：



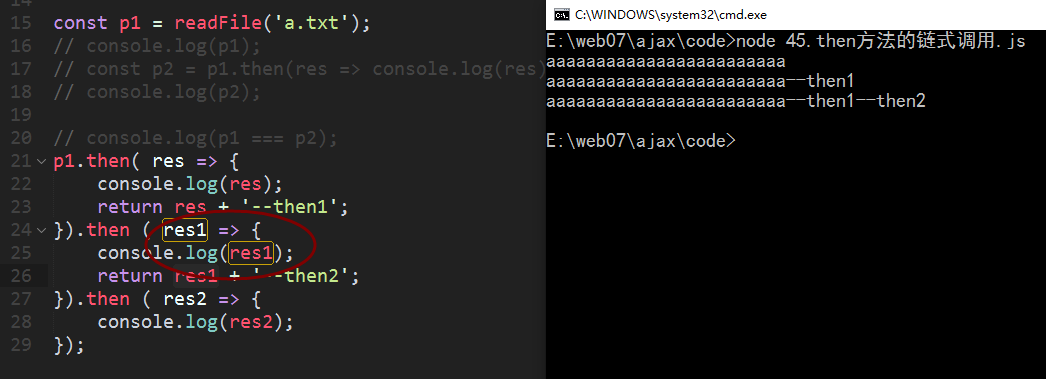
所以，then方法可以实现链式调用。

在then方法的链式调用中，有一个非常重要的特点：

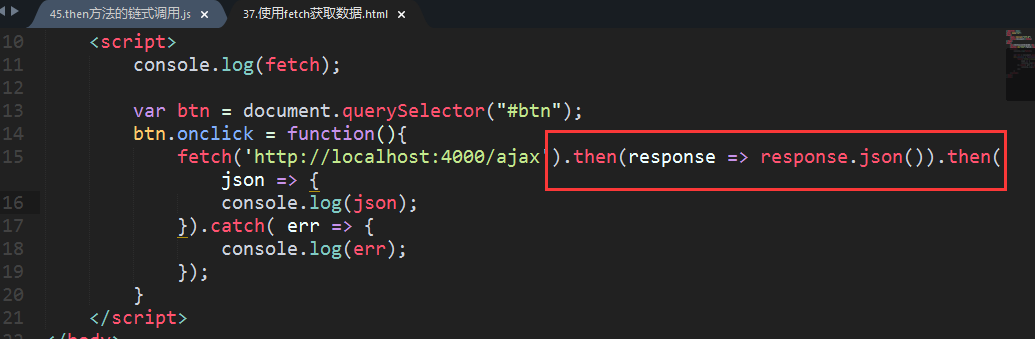
**前一个then方法的返回值，或作为下一个then方法的参数（普通参数，promise对象例外）**

如果返回的是promise对象，就作为下一次调用then方法的promise对象。

编写代码如下：



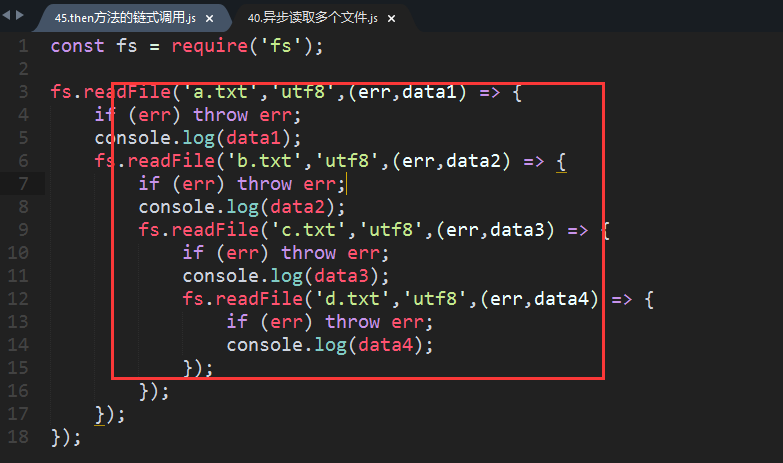
此时，就应该联想到在使用fetch的时候，如下代码：



注意：**这个then方法中回调函数的return 返回值，和then方法本身的返回值是两码事。**

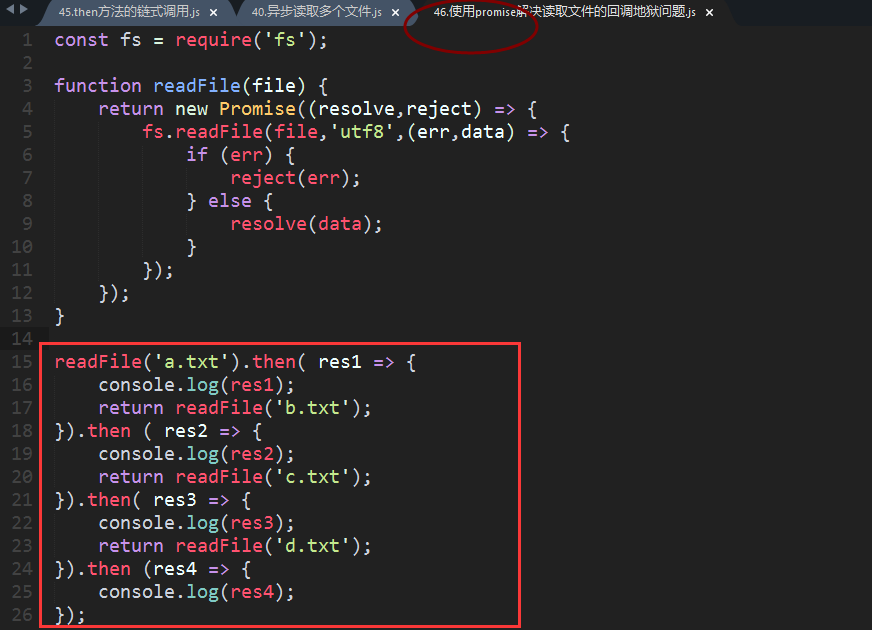
到这儿，我们就可以解决回调地狱的糟糕写法问题。

原来的写法，如下：

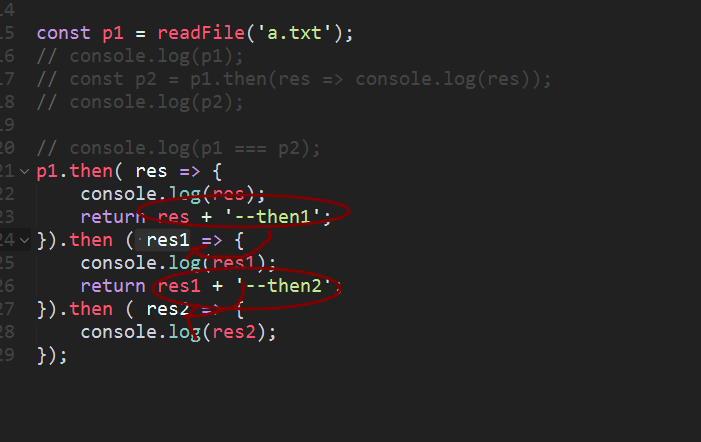


都是通过回调来实现的，层层嵌套，不优雅。

现在，我们就可以通过promise改写，使其使用**同步的一种方式来**编写代码。



如何理解：



这种写法，上一个then的返回值，就直接等于下一个then 的参数。



在这种写法中，返回的是一个promise对象，这个对象就作为下一次调用then方法promise对象。

最后，当有多个then方法调用的时候，需要将catch放到最后，从而捕获任何一个then方法的错误。如下：



### all和race方法

all：所有

race：竞赛

all和race都是Promise构造器对象的静态方法。直接使用Promise调用，如下：

* Promise.all（）
* Promise.reace（）

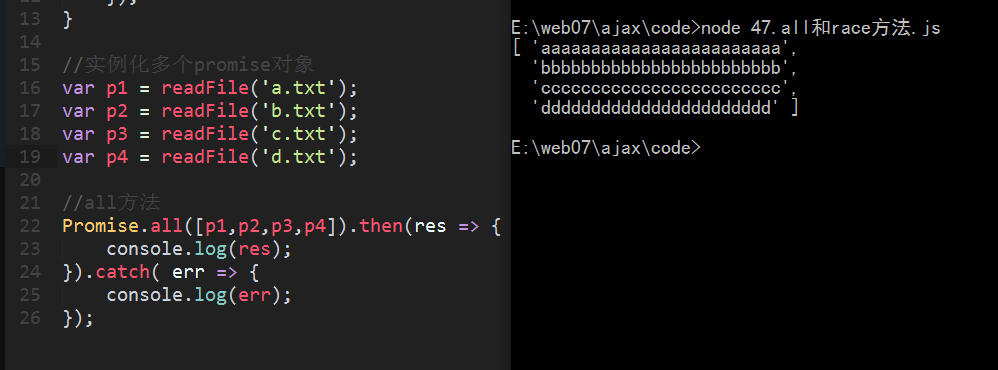
返回值都是promise对象。

当有多个异步操作的时候，经常会有如下两种需求：

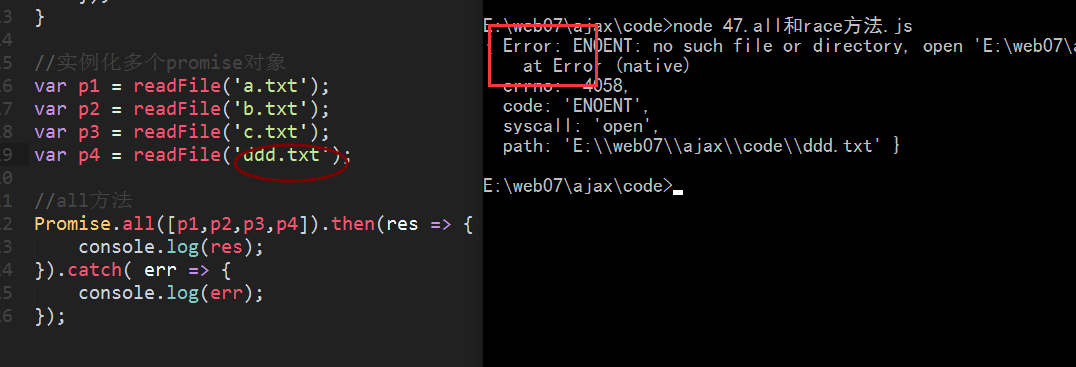
* 确保所有的异步操作完成之后，才进行某个操作，只要有一个失败，就不进行
* 只要有一个异步操作文章，就里面执行某个操作。

有点类似于运算符中的 逻辑与 和 逻辑或。

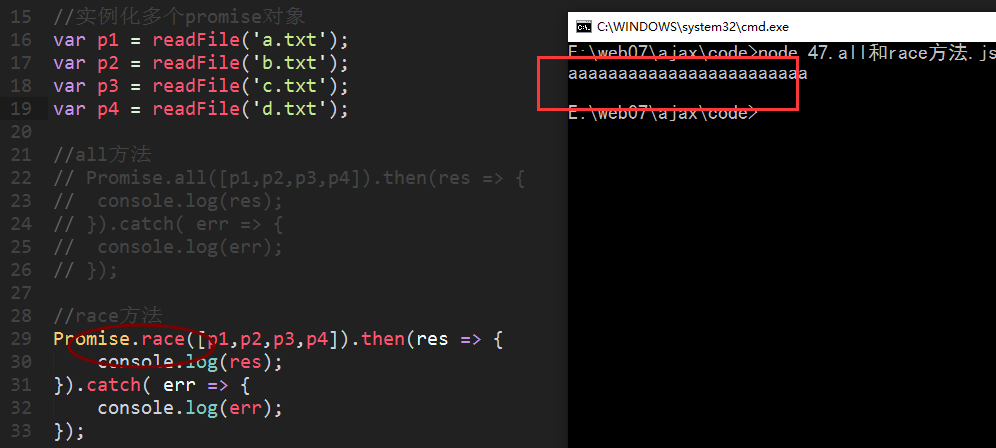
all使用如下：



如果有错误，如下：



**race的用法**



比如，在一个页面中，需要发多个ajax请求，必须保证所有的数据都返回，才显示页面，可以使用all。

又比如，异步获取一个数据，这个数据有多个接口，只要有一个成功即可，此时就可以race。

## 使用第三方的Promise库

对开发中使用promise进行小结：

没有异步，就不需要promise。

不使用promise，其实也是可以解决异步编程的问题。使用promise，会使异步的编码变得更加优雅，功能会更强。

在进行promise编程的使用，有如下两个场景：

* 直接使用别人封装好的promise对象，比如fetch、axios
* 需要自己封装promise对象

注意：axios和fetch必须使用promise方式，如：



针对自己封装promise对象，又可以有如下两种方式：

* 自己封装
* 可以使用第三方的promise库

比如，针对第三方的promise库，有两个知名的库：

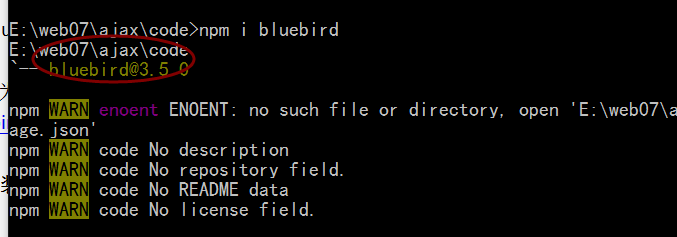
* bluebird
* q.js

可以利用bluebird 和 q.js 快速的生成promise对象。

以bluebird为例，在服务端演示其用法。

<http://bluebirdjs.com/docs/getting-started.html>

**第一步，安装**



**第二步，使用**

