# Promise

Promise 对象是一个代理对象（代理一个值），被代理的值在Promise对象创建时可能是未知的。它允许你为异步操作的成功和失败分别绑定相应的处理方法（handlers）。 这让异步方法可以像同步方法那样返回值，但并不是立即返回最终执行结果，而是一个能代表未来出现的结果的promise对象

一个 Promise就3种状态:  
pending: 初始状态，既不是成功，也不是失败状态。  
fulfilled: 意味着操作成功完成。  
rejected: 意味着操作失败。

pending 状态的 Promise 对象可能会变为fulfilled 状态并传递一个值给相应的状态处理方法，也可能变为失败状态（rejected）并传递失败信息。当其中任一种情况出现时，Promise 对象的 then 方法绑定的处理方法（handlers ）就会被调用（then方法包含两个参数：onfulfilled 和 onrejected，它们都是 Function 类型。当Promise状态为fulfilled时，调用 then 的 onfulfilled 方法，当Promise状态为rejected时，调用 then 的 onrejected 方法， 所以在异步操作的完成和绑定处理方法之间不存在竞争）。

因为 Promise.prototype.then 和 Promise.prototype.catch 方法返回promise 对象， 所以它们可以被链式调用。

## 关于promise对象

创建一个promise实例，然后直接打印会报错

<**script**>  
 **var** pro = **new** Promise();  
 console.log(pro);  
</**script**>

原因就是当创建一个promise实例的时候，需要在里面放一个函数：

<**script**>  
 **var *pro*** = **new *Promise***(() => {});  
 ***console***.log(***pro***); *// 打印promise实例*</**script**>

----------------------

如果在函数里面，写一些代码，会发现内部代码立即执行了，所以大多数情况下，不会直接把promise直接暴露在外面，而是把promise用一个函数包一下：

<**script**>  
 **var *pro*** = **new *Promise***(() => {  
 ***console***.log(**"实例化..."**); *// 立即执行* });  
</**script**>

----------------

在promise的函数中传入两个参数：resolve和reject ，表示成功的状态和失败的状态：

<**script**>  
 **var *pro*** = **new *Promise***((resolve, reject) => {  
 ***console***.log(**"实例化..."**);  
 *// resolve() // "fulfilled"* reject() *// "rejected"* });  
 ***console***.log(***pro***);  
</**script**>

------------------

一旦进入到成功或失败的状态，就无法再改变状态了：

<**script**>  
 **var *pro*** = **new *Promise***((resolve, reject) => {  
 ***console***.log(**"实例化..."**);  
 resolve() *// "fulfilled" 进入到成功状态，reject就没有用了* reject() *// 不会执行* });  
 ***console***.log(***pro***);  
</**script**>

但是不会阻止后续代码的执行：

<**script**>  
 **var *pro*** = **new *Promise***((resolve, reject) => {  
 ***console***.log(**"实例化..."**);  
 resolve() *// "fulfilled" 进入到成功状态，reject就没有用了* reject() *// 不会执行* ***console***.log(**"啦啦啦~"**); *// 会执行* });  
 ***console***.log(***pro***);  
</**script**>

所以我们通常使用return阻止后续代码的执行：

<**script**>  
 **var *pro*** = **new *Promise***((resolve, reject) => {  
 ***console***.log(**"实例化..."**);  
 **return** resolve() *// "fulfilled" 进入到成功状态，reject就没有用了* reject() *// 不会执行* ***console***.log(**"啦啦啦~"**); *// 不会执行* });  
 ***console***.log(***pro***);  
</**script**>

----------------------

当异步代码执行成功时，我们应该调用resolve(...), resolve会把实参传给then；   
当异步代码失败时应该调用reject(...) ， reject会把实参传给catch

如果把resolve和reject互换位置，可以发现，即使异步函数成功执行了，但是走的还是reject(); 执行的还是catch方法。所以promise和异步操作的执行状态没有任何关系，它只负责传值，异步操作成功了传什么；失败了传什么。它只是一种书写形式上的改变，改变了回调地狱。你ajax操作的成功和失败和promise没有任何关系：

<**script**>  
 **var *pro*** = **new *Promise***((resolve, reject) => {  
 **var** xhr = **new *XMLHttpRequest***();  
 xhr.**onreadystatechange** = **function** () {  
 **if** (xhr.**readyState** == 4) {  
 **if** (xhr.**status** == 200) {  
 resolve(**'欧耶，成功了:'** + xhr.**responseText**)  
 } **else** {  
 reject(**'哥们，失败了:'** + xhr.**statusText**)  
 }  
 }  
 }  
 xhr.open(**'get'**, **'a.txt'**, **true**);  
 xhr.send(**null**);  
 });  
 ***pro***.then((successMessage) => {  
 *// successMessage的值是上面调用resolve(...)方法传入的实参  
 // successMessage参数不一定非要是字符串类型* ***console***.log(**'then:'** + successMessage);  
 });  
 ***pro***.catch((error) => {  
 ***console***.log(error);  
 })  
</**script**>

你也可以把catch里面的函数写到then里面，但是这种写法易读性不好，不推荐：

<**script**>  
 pro.then((successMessage) => {  
 *// successMessage的值是上面调用resolve(...)方法传入的实参  
 // successMessage参数不一定非要是字符串类型* ***console***.log(**'then:'** + successMessage);  
 }, (error) => {  
 ***console***.log(error);  
 });  
</**script**>

---------------------

then方法执行完之后有一个返回值，返回的是一个新的Promise对象，所以可以使用链式方法继续调用then方法：

<**script**>  
 **var** pro = **new** Promise((resolve, reject) => {  
 **return** resolve()  
 });  
 **var** pro2 = pro.then(() => {  
 console.log(**'then1'**);  
 })  
 pro2.then(() => {  
 console.log(**'then2'**);  
 })  
 console.log(pro2); *// Promise* console.log(pro2 **instanceof** Promise); *// true* console.log(pro == pro2); *// false*</**script**>

--------------------

在pro2 的then里面拿到的值是 pro1的then里面的返回值，你返回一个111， pro2的then里面拿到的就是111， 你返回一个undefined ,pro2的then里面拿到的就是undefined;   
你返回一个 Promise对象，pro2的then里面拿到的就是 返回的那个Promise对象的resolve的值，  
pro2的catch里面拿到的就是 返回的那个Promise对象的reject的值，

<**script**>  
 **var *pro1*** = **new *Promise***((resolve, reject) => {  
 resolve(**'成功1'**)  
 });  
 **var *pro2*** = ***pro1***.then((value) => {  
 ***console***.log(value); *// 成功1  
 // return 111* **return new *Promise***((resolve, reject) => {  
 *// resolve('成功2')* reject(**"失败2"**)  
 });  
 });  
 ***pro2***.then((value) => {  
 ***console***.log(**'then2：'** + value); *// then2：成功2* });  
 ***pro2***.catch((error) => {  
 ***console***.log(error);  
 });  
 ***console***.log(***pro2***);  
  
</**script**>

--------------------

catch 除了可以接收reject信息，还可以拦截错误信息，捕获前面链式调用中的错误信息:

<**script**>  
 **var *pro1*** = **new *Promise***((resolve, reject) => {  
 resolve(**'成功1'**)  
 });  
 **var *pro2*** = ***pro1***.then((value) => {  
 ***console***.log(1);  
 })  
 .then((value) => {  
 ***console***.log(2);  
 a *// 这里写个a就报错了* })  
 .then((value) => {  
 ***console***.log(3);  
 })  
 .catch((err) => {  
 ***console***.log(err);  
 })  
 .finally(() => {  
 *// 不过是then还是catch， 我都会执行* ***console***.log(**'执行完了！'**);  
 })  
  
 ***console***.log(**'啦啦啦~'**);  
</**script**>

finally方法不管前面是走的then还是catch 都会执行。

## Promise静态方法

### resolve()

Promise有个静态方法：resolve()

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

resolve(1)

});

上面的代码可以写成：

.then(() => { return Promise.resolve(2) })

<**script**>  
 **var *pro1*** = **new *Promise***((resolve, reject) => {  
 resolve(**'成功1'**)  
 })  
 *// .then((value) => {  
 // return new Promise((resolve, reject) => {  
 // resolve(1)  
 // });  
 // })* .then(() => {  
 **return *Promise***.resolve(2)  
 })  
 .then((value) => {  
 ***console***.log(value);  
 })  
  
</**script**>

### reject()

Promise还有个静态方法：reject()

<**script**>  
 **var *pro1*** = **new *Promise***((resolve, reject) => {  
 resolve(**'成功1'**)  
 })  
  
 .then(() => {  
 **return *Promise***.reject(2)  
 })  
  
 .catch((value) => {  
 ***console***.log(value);  
 })  
  
</**script**>

### all()

这个方法返回一个新的promise对象，该promise对象在iterable参数对象里所有的promise对象都成功的时候才会触发成功，一旦有任何一个iterable里面的promise对象失败则立即触发该promise对象的失败。这个新的promise对象在触发成功状态以后，会把一个包含iterable里所有promise返回值的数组作为成功回调的返回值，顺序跟iterable的顺序保持一致；如果这个新的promise对象触发了失败状态，它会把iterable里第一个触发失败的promise对象的错误信息作为它的失败错误信息。Promise.all方法常被用于处理多个promise对象的状态集合。

<**script**>  
 **var *pro1*** = **new *Promise***((resolve, reject) => {  
 *setTimeout*(() => {  
 resolve(**'成功1'**)  
 ***console***.log(**'a'**);  
 }, 1000);  
 });  
 **var *pro2*** = **new *Promise***((resolve, reject) => {  
 *setTimeout*(() => {  
 resolve(**'成功2'**)  
 ***console***.log(**'b'**);  
 }, 1500);  
 });  
 **var *pro3*** = **new *Promise***((resolve, reject) => {  
 *setTimeout*(() => {  
 resolve(**'成功3'**)  
 ***console***.log(**'c'**);  
 }, 500);  
 });  
  
 **var *p*** = ***Promise***.all([***pro1***,***pro2***,***pro3***])  
 *// console.log(p);* ***p***.then((data) => {  
 ***console***.log(data);  
 })  
</**script**>

### race()

当iterable参数里的任意一个子promise被成功或失败后，父promise马上也会用子promise的成功返回值或失败详情作为参数调用父promise绑定的相应句柄，并返回该promise对象。

哪个快就先返回哪个

<**script**>  
 **var *pro1*** = **new *Promise***((resolve, reject) => {  
 *setTimeout*(() => {  
 resolve(**'成功1'**)  
 }, 1000);  
 });  
 **var *pro2*** = **new *Promise***((resolve, reject) => {  
 *setTimeout*(() => {  
 resolve(**'成功2'**)  
 }, 1500);  
 });  
 **var *pro3*** = **new *Promise***((resolve, reject) => {  
 *setTimeout*(() => {  
 resolve(**'成功3'**)  
 }, 500);  
 });  
  
 **var *p*** = ***Promise***.race([***pro1***,***pro2***,***pro3***])  
 *// console.log(p);* ***p***.then((data) => {  
 ***console***.log(data);  
 })  
  
</**script**>