# Promise

## 概念

ES6开始支持。
Promise对象用于一个异步操作的最终完成（包括成功和失败）及结果值的表示。
简单说，就是处理异步请求的。
之所以叫做Promise，就是我承诺做这件事，如果成功则怎么处理，失败则怎么处理。

```
// 语法
new Promise(
    /* 下面定义的函数是executor */
    function(resolve, reject) {...}
);
```

### executor

- executor 是一个带有 resolve 和 reject **两个参数的函数** 。
- executor 函数在Promise构造函数执行时立即执行，被传入resolve和reject函数作为参数（executor 函数在Promise构造函数返回新建对象前被调用）。
- executor 内部通常会执行一些异步操作，一旦完成，可以调用resolve函数来将promise状态改成fulfilled即完成，或者在发生错误时将它的状态改为rejected即失败。
- 如果在executor函数中抛出一个错误，那么该promise 状态为rejected。executor函数的返回值被忽略
- executor中，resolve或reject只能执行其中一个函数

### Promise的状态

- pending: 初始状态，不是成功或失败状态。
- fulfilled: 意味着操作成功完成。
- rejected: 意味着操作失败。

```
setInterval(func[, delay]); // 间隔多少毫秒就执行函数一次，循环执行
setTimeout(func[, delay]);  // 等待多少毫秒就执行函数一次，结束
delay      // 延时，缺省0，立即执行
func  // 延时到的时候执行的函数
```

```
var myPromise = new Promise(function(resolve, reject){
    console.log('do sth.')
    setTimeout(()=>{
        console.log('~~~~~')
        resolve('ok');
        //reject('error');
    }, 3000);
    console.log('++++++++++')
})

```

```
console.log(myPromise);

setInterval(() => {
    console.log(myPromise, '++++')
}, 1000); // 每隔1秒执行一次
```

**Promise.then(onFulfilled, onRejected)**
then方法返回一个**新的Promise**对象。参数是2个函数，根据当前Promise对象的状态来调用不同的函数，fulfilled走onFulfilled函数，rejected走onRejected函数。

**Promise.catch(onRejected)**

为当前Promise对象添加一个拒绝回调onRejected函数，返回一个**新的Promise对象。**

Promise 提供2个方法：

- Promise.resolve(value) 返回 状态为fulfilled的Promise对象
- Promise.reject(reason)返回 状态为rejected状态的Promise对象

# 原理实验

## 1、都不处理

```
var A = new Promise(function(resolve, reject){
    console.log('do sth.')
    setTimeout(()=>{
        console.log('~~~~~')
        resolve('ok');
        reject('error');
        console.log('+++++')
    }, 3000); // 延时3秒执行一次结束
})

var B = A.then();

setInterval(()=>{
    console.log(A);
    console.log(B);
    console.log('~~~~~~~~~~~~~~~~~~~~')
}, 1000);
```

A执行完后，不管成功与否，其状态无函数处理，那么A的状态就是B的状态。
A fulfilled则B也fulfilled，A rejected则B也rejected。
可以简单总结为，A没函数管，B也被感染，得向后传递这种状态，**透传。**

**注意**：失败的打印的时候会有`<rejected>`字样，`Promise { <rejected> 'error' }`，而成功的不会显示有这个尖括号的。

## 2、处理

```
var A = new Promise(function(resolve, reject){
    console.log('do sth.')
    setTimeout(()=>{
        console.log('~~~~~')
        resolve('ok');
        //reject('error');
        console.log('+++++')
    }, 3000); // 延时3秒执行一次结束
})

var B = A.then(
    value => { // 只处理成功
        console.log(value);
        return 'A.then.OK' // 返回的value
    }
)

// var B = A.then(undefined,
//     reason=>{ // 只处理失败
//         console.log(reason);
// })

setInterval(()=>{
    console.log(A);
    console.log(B);
    console.log('~~~~~~~~~~~~~~~~~~~~')
}, 1000);
```

A执行完后，成功fulfilled

- 其状态无函数处理，那么A的状态就是B的状态
- 其状态有函数处理，那么A的状态和B的状态，都是成功

A执行完后，失败rejected

- 其状态无函数处理，那么A的状态就是B的状态，都是失败
- 其状态有函数处理，那么A的状态是rejected，而B状态为fulfilled

## 3、都处理

```
var A = new Promise(function (resolve, reject) {
    console.log('do sth.')
    setTimeout(() => {
        console.log('~~~~~')
        resolve('ok');
        reject('error');
        console.log('+++++')
    }, 3000); // 延时3秒执行一次结束
})

var B = A.then(
    value => console.log(value, 1111),    // 处理成功
    reason => console.log(reason, 2222)) // 处理失败
```

```
setInterval(() => {
    console.log(A);
    console.log(B);
    console.log('~~~~~~~~~~~~~~~~~~~~')
}, 1000);
```

A执行完后，不管成功与否，其状态都有函数处理，这些函数的返回值，都会被resolve(return_value)，相当于都是fulfilled。所以，不管A是否成功执行，B的状态总是fulfilled。

## 4、返回Promise对象

```
var A = new Promise(function (resolve, reject) {
    console.log('do sth.')
    setTimeout(() => {
        console.log('~~~~~')
        //resolve('ok');
        reject('error');
        console.log('++++++')
    }, 3000); // 延时3秒执行一次结束
})

var B = A.then(
    value => {
        console.log(value, 1111);
        return Promise.reject(3333);
    },
    reason => {
        console.log(reason, 2222);
        return Promise.reject(4444);
    }
)

setInterval(() => {
    console.log(A);
    console.log(B);
    console.log();
}, 1000);
```

A执行完后，不管成功与否，其状态都有函数处理。
如果处理函数的返回值为Promise.reject(3333)或者Promise.resolve(4444)，将分别对应rejected或fulfilled状态的一个新的Promise对象给B

看看下面输出什么?

```
var myPromise = new Promise(function(resolve, reject){
    console.log('do sth.')
    setTimeout(()=>{
        console.log('~~~~~')
        resolve('ok');
        //reject('error');
    }, 3000); // 延时3秒执行一次结束
})
```

```
let pro1 = myPromise.then(
    value => {/*如果成功则显示结果*/
        console.log(1, 'successful');
        return 1111;
    },
    reason => {/*如果失败则显示原因*/
        console.log(2, 'failed');
        return 2222;
    }
)

let pro2 = myPromise.catch(reason=>{
    console.log(3, reason)
})

// 开始链式调用
pro2.then(
    value => console.log(4, value), // value是什么?
    reason => console.log(5, reason) // reason是什么?
).then(
    value => {
        console.log(6, value) // 已经不是pro2对象了，value是什么
        return Promise.reject('pro2 => new Promise object rejected');
    }
).catch(
    reason => {
        console.log(7, reason);
        return Promise.resolve(reason + ' *')
    }
).then(
    value => console.log(8, value), // value是什么?
    reason => console.log(9, reason) // reason是什么?
) // 返回的是什么?
```

# async

await 操作符用于等待一个Promise对象，因为Promise对象何时执行返回不知道，返回值是等待的Promise的处理结果。

await操作符只能用在async function中，它会暂停当前async function的执，等待Promise处理完

- 等待的Promise正常处理（fulfilled），则返回resolve的参数
- 等待的Promise处理异常（rejected），await会抛出Promise的异常原因

```
var A = new Promise(function(resolve, reject){
    console.log('do sth.')
    setTimeout(()=>{
        console.log('##~~~~~~~~~~~~~~~~~~~~')
        resolve('成功了');
        reject('error');
        console.log('$$~~~~~~~~~~~~~~~~~~~~')
    }, 3000); // 延时3秒执行一次结束
})

```

```
11
12  async function test () {
13      console.log('%%%%%%%%%%%%%%%%%%%%')
14      try {
15          let a = await A;
16          console.log('^^^^^^^^^^^^^^^^^^^', a)
17      } catch (e) { // reject将抛出异常
18          console.log(e);
19      }
20  }
21
22  test() // 并没有阻塞
23  console.log('======================')
```

使用async function和await可以简化Promise的处理。