javascript是一门单线程的语言，也就是说一次只能完成一件任务，如果有多个任务，就需要排队进行处理。如果一个任务耗时很长，后面的任务也必须排队等待，这样大大的影响了整个程序的执行。为了解决这个问题，javascript语言将任务分为两种模式：

同步：当我们打开网站，网页的页面骨架渲染和页面元素渲染，就是一大推同步任务。

异步：我们在浏览新闻时，加载图片或音乐之类占用资源大且耗时久的任务就是异步任务

在javascript发展过程中，出现过不同的方式去处理异步任务。

**1. Callback函数**

回调函数是最早解决异步编程的方法。无论是常见的setTimeout还是ajax请求，都是采用回调的形式把事情在某一固定的时刻进行执行。回调函数的处理一般将callback函数作为参数传进函数，在合适的时候被调用执行。回调函数的优点就是简单、容易理解和实现，但有个致命的缺点，容易出现回调地狱（Callback hell）,即多个回调函数嵌套使用。造成代码可读性差、可维护性差且只能在回调中处理异常。

例如：

setTimeout(function callback(){

 console.log('aa');

}, 1000); #在一秒后执行回掉函数，打印出’aa’

ajax (url,{},function callback(){

 console.log ("ajax success",res); #在向url执行的地址发出 ajax请求，完成收到回复以后调用callback函数

})

回调函数缺点例子，多个回调函数嵌套使用，可读性差：

ajax (url, {} ， () => {

    //to do something

    ajax (url1, {} ， () => {

        // to do something

        ajax (url2, {} ， () => {

            // to do something

        })

    })

})

**2.消息订阅模式**

我们假定，存在一个"信号中心"，某个任务执行完成，就向信号中心"发布"（publish）一个信号，其他任务可以向信号中心"订阅"（subscribe）这个信号，从而知道什么时候自己可以开始执行。这就叫做 发布/订阅模式（publish-subscribe pattern），又称观察者模式(observer pattern）

例子：

//利用jquery的插件实现

//首先，f2向消息中心订阅success事件

jQuery.subscribe('success', f2);

//对f1进行改写：

function f1(){

    ajax(url,() => {

        //todo

        jQuery.publish('success'); //当f1执行完毕后，向消息中心jQuery发布success事件，f2由于订阅了success时间，所以f2将执行

    })

}

//f2执行完毕后，可以取消订阅

jQuery.unsubscribe('success', f2)

**3. Promise对象机制（ES2015/ES6）**

[Promise](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Promise)是CommonJS工作组提出的一种规范, 可以获取异步操作的消息，也是异步处理中常用的一种解决方案。Promise的出现主要是用来解决回调地狱、支持多个并发的请求，获取并发请求的数据并且解决异步的问题。

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

# 构建promise对象需要传入一个function，function的参数为两个方法的指针，其中resolve为执行成功时调用的方法指针， reject为执行失败时调用的方法指针

 setTimeout(()=>{ //在promise对象中做一些异步操作。此处例子为10秒后进行生成随机数

//setTimeout为异步执行，当执行到setTimeout语句时候，会马上执行下面语句，而在指定delay以后再执行setTimeout中的语句。

  let num = parseInt(Math.random()\*100);

    if (num > 50) {

    resolve ("num > 50"); // 如果数字大于50就调用函数resolve，参数为字符”num > 50”, 并且将promise自身状态变成Resolved

}

else {

     reject ("num <50”) ;// 否则就调用代表失败的函数reject，参数为字符串”num < 50”, 并且将promise自身状态变成Rejected

   }

  },10000)

});

p.then((res) => { // 当promise状态变为Resolved时，执行then语句，then语句中定义一个方法，这个方法的参数为resolve方法返回的的结果。

    console.log(res);

}). catch((err) => { // 当promise状态变为Rejected时，执行catch语句，catch语句中定义一个方法，这个方法的参数为reject方法返回的结果。

 console.log(err);

})

---Promise有三种状态：等待pending、成功Resolved、失败rejected；状态一旦改变，就不会再变化，在Promise对象创建后，会马上执行。当调用resolve函数时，等待状态变为Resolved状态，并将resolve()方法的返回值给相应的状态处理方法（then）作为其回调函数的参数。当调用reject函数时，变为失败状态rejected，并将reject()方法的返回值给相应的状态处理方法（catch）作为其回调函数的参数。

---在promise对象接受一个function作为其参数。该function的参数为两个方法的指针，其中resolve为执行成功时调用的方法指针， reject为执行失败时调用的方法指针。并且调用resolve方法的参数将作为返回值，当作then方法的参数。调用reject方法的参数将作为返回值，当作catch方法的参数.

---在promise中调用resolve或reject并不会终结 Promise 的参数函数的执行。一般来说，调用resolve或reject以后，Promise 的使命就完成了，后继操作应该放到then方法里面，而不应该直接写在resolve或reject的后面。

--- then方法返回的是一个新的Promise实例（注意，不是原来那个Promise实例）。因此可以采用链式写法，即then方法后面再调用另一个then方法。这时候后面的回调函数then，就会等待前一个返回的Promise对象的状态发生变化，才会被调用。

例如：

getJSON("/post/1.json").then ( # getJSON("/post/1.json")返回一个promise对象

post => getJSON(post.commentURL) # 然后then语句返回一个新的promise对象，下一个then语句等待该新 promise对象状态改变才调用下一个then语句

).then (

comments => console.log ("resolved: ", comments),);

--- finally方法用于指定不管 Promise 对象最后状态如何，都会执行的操作。该方法是 ES2018 引入标准的。finally方法的回调函数不接受任何参数，这意味着没有办法知道，前面的 Promise 状态到底是fulfilled还是rejected。这表明，finally方法里面的操作，应该是与状态无关的，不依赖于 Promise 的执行结果。

Promise实例对象

.then(result => {···})

.catch(error => {···})

.finally(() => {···});

--- 我们有两种方法从Promise对象中获取结果。第一种是使用Promise自身，使用then方法的参数response获取Promise返回的结果。第二种是使用await关键字，获取其返回的结果.

--- 当我们知道一个变量为Promise对象时，或者某些关键字将返回Promise对象时候，我们需要明确该Promise对象的两个方面。基于构建Promise对象的本质，第一需要明确此promise对象resolve方法执行的是什么，reject方法执行的是什么。第二是明确resolve方法的返回值是什么，这个返回值将用于then中作为参数。

例如：

Axios模块。

Axios.request({

method: method,

<url:url>

}).then(response => console.log(response.status)

Axios.request() 方法将返回一个Promise对象。由于经过内部处理，我们无法直接看出resolve方法执行的具体逻辑，但是通过查阅Axios文档，我们知道该Promised对象Resolved以后的返回值response是一个对象，并且response.status代表请求的响应码，response.data为请求的返回内容等。所以可以接then方法，该方法的参数为该promise对象resolve方法的返回值，也就是response. 然后根据Axios文档，打印出响应状态码为response.status

**4. async和await关键字 (ES2016/ES7）**

async/await在ES7提出，是目前在javascript异步处理的终极解决方案。

async 其本质是 Generator 函数的语法糖。

async表示修饰函数为异步函数。被async 修饰后的函数返回值是Promise对象。如果async函数中return一个直接量，async 也会把这个直接量封装成 Promise 对象。Promise 对象可以直接使用 then() 方法进行链式调用。

await表示后面的表达式需要等待，await命令后面可以是Promise 或者 原始类型的值。

---- **async定义异步函数, 自动将函数的返回值转换为promise对象**。

例如：

async function f(){

    return "hello async"; #返回为一般变量，但async关键字将其变为promise对象

}

f(). then((res) => {//通过then来指定回调函数且promise内部返回的res作为该回调的参数

    console.log(res); // hello async

})

--- **await关键字只能出现在async函数中**，用于等待后面的promise对象所对应的异步过程执行完成。如果await关键字用于普通函数当中会出错。

--- 如果await后面是一个 Promise 对象，await会阻塞后面的代码，等着该Promise对象resolve。并且await关键字相当于一个提取器，获取后面promise对象的 resolve返回值，作为 await 表达式的运算结果。

例如：

async SendRequest(method, url){

let response = await Axios.request({

method: method,

url: url

})

console.log(response.status) //控制台输出响应的状态。

}

Axios.request() 方法返回的是一个Promise对象，我们通过await关键字提取该Promise对象的返回值response.从而输出response. status. 如果我们不使用await关键字，只是let response = Axios.request(…),是不对的。这样输出的时候相当于Promise对象.status是错误的。

如果await后面不是一个Promise 对象，那 await 表达式的运算结果就是该对象的值。

--- await只关心异步过程成功的消息resolve(data)，拿到相应的数据data。至于失败消息reject(error)，不关心，不处理。async函数返回的Promise对象的catch函数就负责统一抓取内部所有异步过程的错误。  
async函数内部只要有一个异步过程发生错误，整个执行过程就中断，这个返回的Promise对象的catch就能抓到这个错误。

--- 总体结构为：

在一个函数A上定义关键字async, 这个函数由于被async定义，所以它将返回promise对象。由于是promise对象所以可以对函数A指定then以及catch统一的获取他resolved时的回传值和异常。并且程序执行到async修饰的A函数时候，会异步执行，继续马上执行接下来的代码。

在函数A里面，也就是在关键字async所修饰的函数里面，采用await关键字，await关键字后面修饰一个函数B,该函数B将在内部使用promise对象，在promise对象中包括所有的异步过程逻辑。 await关键字能够处理堵塞处理后面的promise对象获取其成功值。

例子：

function sleep(ms) {

return new Promise((resolve) => {//构建promise对象，在ms以后调用resolve将Promise状态变为成功，传回数据sleep for … ms

setTimeout (() => {

resolve ('sleep for ' + ms + ' ms');

}, ms);

});

}

async function asyncFunction() { //async关键字包裹整个函数返回promise对象

console.time('asyncFunction total executing:');

const sleep1 = await sleep (2000); // await后接sleep函数，在sleep函数中使用promise对象。await将阻塞sleep(2000)提取其返回结果。当sleep函数完成后在执行接下来的代码。

console.log ('sleep1: ' + sleep1);

const [sleep2, sleep3, sleep4]= await Promise.all([sleep(2000), sleep(1000), sleep(1500)]); // Promise.all可以将多个Promise实例包装成一个新的Promise实例。同时，成功和失败的返回值是不同的，成功的时候返回的是一个结果数组，而失败的时候则返回最先被reject失败状态的值。

console.log ('sleep2: ' + sleep2);

console.log ('sleep3: ' + sleep3);

console.log ('sleep4: ' + sleep4);

const sleepRace = await Promise.race([sleep (3000), sleep(1000), sleep(1000)]); // Promse.race就是赛跑的意思，意思就是说，Promise.race([p1, p2, p3])里面哪个结果获得的快，就返回那个结果，不管结果本身是成功状态还是失败状态。

console.log ('sleep race: ' + sleepRace);

console.timeEnd('asyncFunction total executing:');

return 'asyncFunction done.' // async自动将此返回包裹为promise对象成功返回的信息

}

asyncFunction().then(data => {

console.log(data); //asyncFunction返回promise对象，统一抓取成功resolved传回的信息

}).catch(error => {

console.log(error); // asyncFunction返回promise对象，统一抓取错误rejected传回的信息

});

console.log(“GGG”)

输出结果为：

GGG # 首先输出GGG,因为asyncFunction函数被async修饰，所以程序异步非阻塞执行

sleep1: sleep for 2000 ms # 由于await后接sleep函数，await阻塞运行。当获得sleep(2000)结果以后再执行接下来的代码

sleep2: sleep for 2000 ms # await Promise.all([sleep(2000), sleep(1000), sleep(1500)])，await阻塞运行，等待后面Promise.all完成。又因为Promise.all需要后面所有promise都完成才算完成，所以相当于等待后面所有的promise都完成

sleep3: sleep for 1000 ms

sleep4: sleep for 1500 ms

sleep race: sleep for 1000 ms # await Promise.race([sleep (3000), sleep(1000), sleep(1000)]), await阻塞运行，等待后面Promise.race完成。又因为Promise.race需要后面promise任意一个promise完成都算完成，所以肯定是1000ms的 sleep先完成。

asyncFunction total executing:: 6135.050048828125 ms

asyncFunction done.