作者：玲.海  
链接：https://zhuanlan.zhihu.com/p/288384170  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

Promise基本上现在不管是大厂还是小厂，promise 已经成为了面试必考知识点；关于 Promise，想必大家都又所了解，可是又掌握了多少，真正面试的时候，又能有多少把握呢？，今天小编特意在此整理promise常考知识点，希望对你的面试保驾护航。

**常见 Promise 面试题**

首先，我们以常见的 Promise 面试题为切入点，我们看看面试官们都爱考什么：

1. Promise 解决了什么问题 & 业界实现？
2. Promise 的基础特征？
3. Promise 常用的 API 有哪些？
4. 能不能手写一个符合 Promise/A+ 规范的 Promise?
5. then 的链式调用&值穿透特性原理
6. Promise 在事件循环中的执行过程是怎样的？
7. Promise 有什么缺陷，可以如何解决？

这几个问题由浅入深，我们一个一个来看：

**1. Promise 出现的原因 & 业界实现**

在 Promise 出现以前，在我们处理多个异步请求嵌套时，代码往往是这样的。。。

let fs = require('fs')

fs.readFile('./name.txt','utf8',function(err,data){

fs.readFile(data, 'utf8',function(err,data){

fs.readFile(data,'utf8',function(err,data){

console.log(data);

})

})

})

为了拿到回调的结果，我们必须一层一层的嵌套，可以说是相当恶心了。而且基本上我们还要对每次请求的结果进行一系列的处理，使得代码变的更加难以阅读和难以维护，这就是传说中臭名昭著的**回调地狱**～产生**回调地狱**的原因归结起来有两点：

1.**嵌套调用**，第一个函数的输出往往是第二个函数的输入；

2.**处理多个异步请求并发**，开发时往往需要同步请求最终的结果。

原因分析出来后，那么问题的解决思路就很清晰了：

1.**消灭嵌套调用**：通过 Promise 的链式调用可以解决(.then())；

2.**合并多个任务的请求结果**：使用 Promise.all 获取合并多个任务的错误处理。

Promise 正是用一种更加友好的代码组织方式，解决了异步嵌套的问题。

我们来看看上面的例子用 Promise 实现是什么样的：

let fs = require('fs')

function read(filename) {

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

fs.readFile(filename, 'utf8', (err, data) => {

if (err) reject(err);

resolve(data);

})

})

}

read('./name.txt').then((data)=>{

return read(data)

}).then((data)=>{

return read(data)

}).then((data)=>{

console.log(data);

},err=>{

console.log(err);

})

臃肿的嵌套变得线性多了有木有？没错，他就是我们的异步神器 Promise！

让我们再次回归刚才的问题，**Promise 为我们解决了什么问题？**在传统的异步编程中，如果异步之间存在依赖关系，就需要通过层层嵌套回调的方式满足这种依赖，如果嵌套层数过多，可读性和可以维护性都会变得很差，产生所谓的“回调地狱”，而 Promise 将嵌套调用改为链式调用，增加了可阅读性和可维护性。也就是说，Promise 解决的是异步编码风格的问题。**那 Promise 的业界实现都有哪些呢？**业界比较著名的实现 Promise 的类库有 bluebird、Q、ES6-Promise。

**2. Promise基本特征**

promise 有三个状态：pending，fulfilled，or rejected；「规范 Promise/A+ 2.1」

1. new promise时， 需要传递一个executor()执行器，执行器立即执行；
2. executor接受两个参数，分别是resolve和reject；
3. promise 的默认状态是 pending；
4. promise 只能从pending到rejected, 或者从pending到fulfilled，状态一旦确认，就不会再改变；
5. promise 必须有一个then方法，then 接收两个参数，分别是 promise 成功的回调 onFulfilled, 和 promise 失败的回调 onRejected；「规范 Promise/A+ 2.2」

then方法的执行结果也会返回一个Promise对象。因此我们可以进行then的链式执行，这也是解决回调地狱的主要方式。

7. 如果调用 then 时，promise 已经成功，则执行onFulfilled，参数是promise的value

8. 如果调用 then 时，promise 已经失败，那么执行onRejected, 参数是promise的reason

9. 如果 then 中抛出了异常，那么就会把这个异常作为参数，传递给下一个 then 的失败的回调onRejected

下面就通过例子进一步讲解。

//构建Promise

var promise = new Promise(function (resolve, reject) {

if (/\* 异步操作成功 \*/) {

resolve(data);

} else {

/\* 异步操作失败 \*/

reject(error);

}

});

resolve函数的作用：在异步操作成功时调用，并将异步操作的结果，作为参数传递出去；  
reject函数的作用：在异步操作失败时调用，并将异步操作报出的错误，作为参数传递出去。

Promise实例生成以后，可以用then方法指定resolved状态和reject状态的回调函数。

/\* 接例3.1 \*/

promise.then(onFulfilled, onRejected);

promise.then(function(data) {

// do something when success

}, function(error) {

// do something when failure

});

而then方法中指定的回调函数，将**在当前脚本所有同步任务执行完才会执行**。如下例：

var promise = new Promise(function(resolve, reject) {

console.log('before resolved');

resolve();

console.log('after resolved');

});

promise.then(function() {

console.log('resolved');

});

console.log('outer');

-------output-------

before resolved

after resolved

outer

resolved

**3. Promise常用api**

**Promise常用api**

* Promise.resolve()
* Promise.reject()
* Promise.prototype.catch()
* Promise.prototype.finally()
* Promise.all()
* Promise.race(）

下面具体说一下每个方法的实现:

**Promise.resolve**

默认产生一个成功的 promise。

static resolve(data){

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

resolve(data);

})

}

**Promise.reject**

默认产生一个失败的 promise，Promise.reject 是直接将值变成错误结果。

static reject(reason){

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

reject(reason);

})

}

**Promise.prototype.catch**

Promise.prototype.catch 用来捕获 promise 的异常，**就相当于一个没有成功的 then**。

Promise.prototype.catch = function(errCallback){

return this.then(null,errCallback)

}

**Promise.prototype.finally**

finally 表示不是最终的意思，而是无论如何都会执行的意思。 如果返回一个 promise 会等待这个 promise 也执行完毕。如果返回的是成功的 promise，会采用上一次的结果；如果返回的是失败的 promise，会用这个失败的结果，传到 catch 中。

Promise.prototype.finally = function(callback) {

return this.then((value)=>{

return Promise.resolve(callback()).then(()=>value)

},(reason)=>{

return Promise.resolve(callback()).then(()=>{throw reason})

})

}

**Promise.all**

promise.all 是解决并发问题的，多个异步并发获取最终的结果（如果有一个失败则失败）。

Promise.all = function(values) {

if (!Array.isArray(values)) {

const type = typeof values;

return new TypeError(`TypeError: ${type} ${values} is not iterable`)

}

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

let resultArr = [];

let orderIndex = 0;

const processResultByKey = (value, index) => {

resultArr[index] = value;

if (++orderIndex === values.length) {

resolve(resultArr)

}

}

for (let i = 0; i < values.length; i++) {

let value = values[i];

if (value && typeof value.then === 'function') {

value.then((value) => {

processResultByKey(value, i);

}, reject);

} else {

processResultByKey(value, i);

}

}

});

}

测试一下：

let p1 = new Promise((resolve, reject) => {

setTimeout(() => {

resolve('ok1');

}, 1000);

})

let p2 = new Promise((resolve, reject) => {

setTimeout(() => {

reject('ok2');

}, 1000);

})

Promise.all([1,2,3,p1,p2]).then(data => {

console.log('resolve', data);

}, err => {

console.log('reject', err);

})

控制台等待 1s 后输出：

"resolve [ 1, 2, 3, 'ok1', 'ok2' ]"

**Promise.race**

Promise.race 用来处理多个请求，采用最快的（谁先完成用谁的）。

**4. 手写一个符合 Promise/A+ 规范的 Promise**

依据promise的基本特征，我们试着勾勒下 Promise 的形状：

// 三个状态：PENDING、FULFILLED、REJECTED

const PENDING = 'PENDING';

const FULFILLED = 'FULFILLED';

const REJECTED = 'REJECTED';

class Promise {

constructor(executor) {

// 默认状态为 PENDING

this.status = PENDING;

// 存放成功状态的值，默认为 undefined

this.value = undefined;

// 存放失败状态的值，默认为 undefined

this.reason = undefined;

// 调用此方法就是成功

let resolve = (value) => {

// 状态为 PENDING 时才可以更新状态，防止 executor 中调用了两次 resovle/reject 方法

if(this.status === PENDING) {

this.status = FULFILLED;

this.value = value;

}

}

// 调用此方法就是失败

let reject = (reason) => {

// 状态为 PENDING 时才可以更新状态，防止 executor 中调用了两次 resovle/reject 方法

if(this.status === PENDING) {

this.status = REJECTED;

this.reason = reason;

}

}

try {

// 立即执行，将 resolve 和 reject 函数传给使用者

executor(resolve,reject)

} catch (error) {

// 发生异常时执行失败逻辑

reject(error)

}

}

// 包含一个 then 方法，并接收两个参数 onFulfilled、onRejected

then(onFulfilled, onRejected) {

if (this.status === FULFILLED) {

onFulfilled(this.value)

}

if (this.status === REJECTED) {

onRejected(this.reason)

}

}

}

写完代码我们可以测试一下：

const promise = new Promise((resolve, reject) => {

resolve('成功');

}).then(

(data) => {

console.log('success', data)

},

(err) => {

console.log('faild', err)

}

)

控制台输出：

"success 成功"

现在我们已经实现了一个基础版的 Promise，但是还不要高兴的太早噢，这里我们只处理了同步操作的 promise。如果在 executor()中传入一个异步操作的话呢，我们试一下：

const promise = new Promise((resolve, reject) => {

// 传入一个异步操作

setTimeout(() => {

resolve('成功');

},1000);

}).then(

(data) => {

console.log('success', data)

},

(err) => {

console.log('faild', err)

}

)

执行测试脚本后发现，promise 没有任何返回。

因为 promise 调用 then 方法时，当前的 promise 并没有成功，一直处于 pending 状态。所以如果当调用 then 方法时，当前状态是 pending，我们需要先将成功和失败的回调分别存放起来，在executor()的异步任务被执行时，触发 resolve 或 reject，依次调用成功或失败的回调。

结合这个思路，我们优化一下代码：

const PENDING = 'PENDING';

const FULFILLED = 'FULFILLED';

const REJECTED = 'REJECTED';

class Promise {

constructor(executor) {

this.status = PENDING;

this.value = undefined;

this.reason = undefined;

// 存放成功的回调

this.onResolvedCallbacks = [];

// 存放失败的回调

this.onRejectedCallbacks= [];

let resolve = (value) => {

if(this.status === PENDING) {

this.status = FULFILLED;

this.value = value;

// 依次将对应的函数执行

this.onResolvedCallbacks.forEach(fn=>fn());

}

}

let reject = (reason) => {

if(this.status === PENDING) {

this.status = REJECTED;

this.reason = reason;

// 依次将对应的函数执行

this.onRejectedCallbacks.forEach(fn=>fn());

}

}

try {

executor(resolve,reject)

} catch (error) {

reject(error)

}

}

then(onFulfilled, onRejected) {

if (this.status === FULFILLED) {

onFulfilled(this.value)

}

if (this.status === REJECTED) {

onRejected(this.reason)

}

if (this.status === PENDING) {

// 如果promise的状态是 pending，需要将 onFulfilled 和 onRejected 函数存放起来，等待状态确定后，再依次将对应的函数执行

this.onResolvedCallbacks.push(() => {

onFulfilled(this.value)

});

// 如果promise的状态是 pending，需要将 onFulfilled 和 onRejected 函数存放起来，等待状态确定后，再依次将对应的函数执行

this.onRejectedCallbacks.push(()=> {

onRejected(this.reason);

})

}

}

}

测试一下：

const promise = new Promise((resolve, reject) => {

setTimeout(() => {

resolve('成功');

},1000);

}).then(

(data) => {

console.log('success', data)

},

(err) => {

console.log('faild', err)

}

)

控制台等待 1s 后输出：

"success 成功"

ok！大功告成，异步问题已经解决了！

熟悉设计模式的同学，应该意识到了这其实是一个**发布订阅模式**，这种收集依赖 -> 触发通知 -> 取出依赖执行的方式，被广泛运用于发布订阅模式的实现。

**5. then 的链式调用&值穿透特性**

我们都知道，promise 的优势在于可以链式调用。在我们使用 Promise 的时候，当 then 函数中 return 了一个值，不管是什么值，我们都能在下一个 then 中获取到，这就是所谓的**then 的链式调用**。而且，当我们不在 then 中放入参数，例：promise.then().then()，那么其后面的 then 依旧可以得到之前 then 返回的值，这就是所谓的**值的穿透**。那具体如何实现呢？简单思考一下，如果每次调用 then 的时候，我们都重新创建一个 promise 对象，并把上一个 then 的返回结果传给这个新的 promise 的 then 方法，不就可以一直 then 下去了么？那我们来试着实现一下。这也是手写 Promise 源码的重中之重，所以，打起精神来，重头戏来咯！

有了上面的想法，我们再结合 [Promise/A+](https://link.zhihu.com/?target=https%3A//promisesaplus.com/) 规范梳理一下思路：

1. then 的参数 onFulfilled 和 onRejected 可以缺省，如果 onFulfilled 或者 onRejected不是函数，将其忽略，且依旧可以在下面的 then 中获取到之前返回的值；「规范 Promise/A+ 2.2.1、2.2.1.1、2.2.1.2」
2. promise 可以 then 多次，每次执行完 promise.then 方法后返回的都是一个“新的promise"；「规范 Promise/A+ 2.2.7」
3. 如果 then 的返回值 x 是一个普通值，那么就会把这个结果作为参数，传递给下一个 then 的成功的回调中；
4. 如果 then 中抛出了异常，那么就会把这个异常作为参数，传递给下一个 then 的失败的回调中；「规范 Promise/A+ 2.2.7.2」
5. 如果 then 的返回值 x 是一个 promise，那么会等这个 promise 执行完，promise 如果成功，就走下一个 then 的成功；如果失败，就走下一个 then 的失败；如果抛出异常，就走下一个 then 的失败；「规范 Promise/A+ 2.2.7.3、2.2.7.4」
6. 如果 then 的返回值 x 和 promise 是同一个引用对象，造成循环引用，则抛出异常，把异常传递给下一个 then 的失败的回调中；「规范 Promise/A+ 2.3.1」
7. 如果 then 的返回值 x 是一个 promise，且 x 同时调用 resolve 函数和 reject 函数，则第一次调用优先，其他所有调用被忽略；「规范 Promise/A+ 2.3.3.3.3」

我们将代码补充完整：

const PENDING = 'PENDING';

const FULFILLED = 'FULFILLED';

const REJECTED = 'REJECTED';

const resolvePromise = (promise2, x, resolve, reject) => {

// 自己等待自己完成是错误的实现，用一个类型错误，结束掉 promise Promise/A+ 2.3.1

if (promise2 === x) {

return reject(new TypeError('Chaining cycle detected for promise #<Promise>'))

}

// Promise/A+ 2.3.3.3.3 只能调用一次

let called;

// 后续的条件要严格判断 保证代码能和别的库一起使用

if ((typeof x === 'object' && x != null) || typeof x === 'function') {

try {

// 为了判断 resolve 过的就不用再 reject 了（比如 reject 和 resolve 同时调用的时候） Promise/A+ 2.3.3.1

let then = x.then;

if (typeof then === 'function') {

// 不要写成 x.then，直接 then.call 就可以了 因为 x.then 会再次取值，Object.defineProperty Promise/A+ 2.3.3.3

then.call(x, y => { // 根据 promise 的状态决定是成功还是失败

if (called) return;

called = true;

// 递归解析的过程（因为可能 promise 中还有 promise） Promise/A+ 2.3.3.3.1

resolvePromise(promise2, y, resolve, reject);

}, r => {

// 只要失败就失败 Promise/A+ 2.3.3.3.2

if (called) return;

called = true;

reject(r);

});

} else {

// 如果 x.then 是个普通值就直接返回 resolve 作为结果 Promise/A+ 2.3.3.4

resolve(x);

}

} catch (e) {

// Promise/A+ 2.3.3.2

if (called) return;

called = true;

reject(e)

}

} else {

// 如果 x 是个普通值就直接返回 resolve 作为结果 Promise/A+ 2.3.4

resolve(x)

}

}

class Promise {

constructor(executor) {

this.status = PENDING;

this.value = undefined;

this.reason = undefined;

this.onResolvedCallbacks = [];

this.onRejectedCallbacks= [];

let resolve = (value) => {

if(this.status === PENDING) {

this.status = FULFILLED;

this.value = value;

this.onResolvedCallbacks.forEach(fn=>fn());

}

}

let reject = (reason) => {

if(this.status === PENDING) {

this.status = REJECTED;

this.reason = reason;

this.onRejectedCallbacks.forEach(fn=>fn());

}

}

try {

executor(resolve,reject)

} catch (error) {

reject(error)

}

}

then(onFulfilled, onRejected) {

//解决 onFufilled，onRejected 没有传值的问题

//Promise/A+ 2.2.1 / Promise/A+ 2.2.5 / Promise/A+ 2.2.7.3 / Promise/A+ 2.2.7.4

onFulfilled = typeof onFulfilled === 'function' ? onFulfilled : v => v;

//因为错误的值要让后面访问到，所以这里也要跑出个错误，不然会在之后 then 的 resolve 中捕获

onRejected = typeof onRejected === 'function' ? onRejected : err => { throw err };

// 每次调用 then 都返回一个新的 promise Promise/A+ 2.2.7

let promise2 = new Promise((resolve, reject) => {

if (this.status === FULFILLED) {

//Promise/A+ 2.2.2

//Promise/A+ 2.2.4 --- setTimeout

setTimeout(() => {

try {

//Promise/A+ 2.2.7.1

let x = onFulfilled(this.value);

// x可能是一个proimise

resolvePromise(promise2, x, resolve, reject);

} catch (e) {

//Promise/A+ 2.2.7.2

reject(e)

}

}, 0);

}

if (this.status === REJECTED) {

//Promise/A+ 2.2.3

setTimeout(() => {

try {

let x = onRejected(this.reason);

resolvePromise(promise2, x, resolve, reject);

} catch (e) {

reject(e)

}

}, 0);

}

if (this.status === PENDING) {

this.onResolvedCallbacks.push(() => {

setTimeout(() => {

try {

let x = onFulfilled(this.value);

resolvePromise(promise2, x, resolve, reject);

} catch (e) {

reject(e)

}

}, 0);

});

this.onRejectedCallbacks.push(()=> {

setTimeout(() => {

try {

let x = onRejected(this.reason);

resolvePromise(promise2, x, resolve, reject)

} catch (e) {

reject(e)

}

}, 0);

});

}

});

return promise2;

}

}

测试一下：

const promise = new Promise((resolve, reject) => {

reject('失败');

}).then().then().then(data=>{

console.log(data);

},err=>{

console.log('err',err);

})

控制台输出：

"失败 err"

至此，我们已经完成了 promise 最关键的部分：then 的链式调用和值的穿透。搞清楚了 then 的链式调用和值的穿透，你也就搞清楚了 Promise

**6. Promise 在事件循环中的执行过程是怎样的？**

首先我们先了解一下什么是事件循环

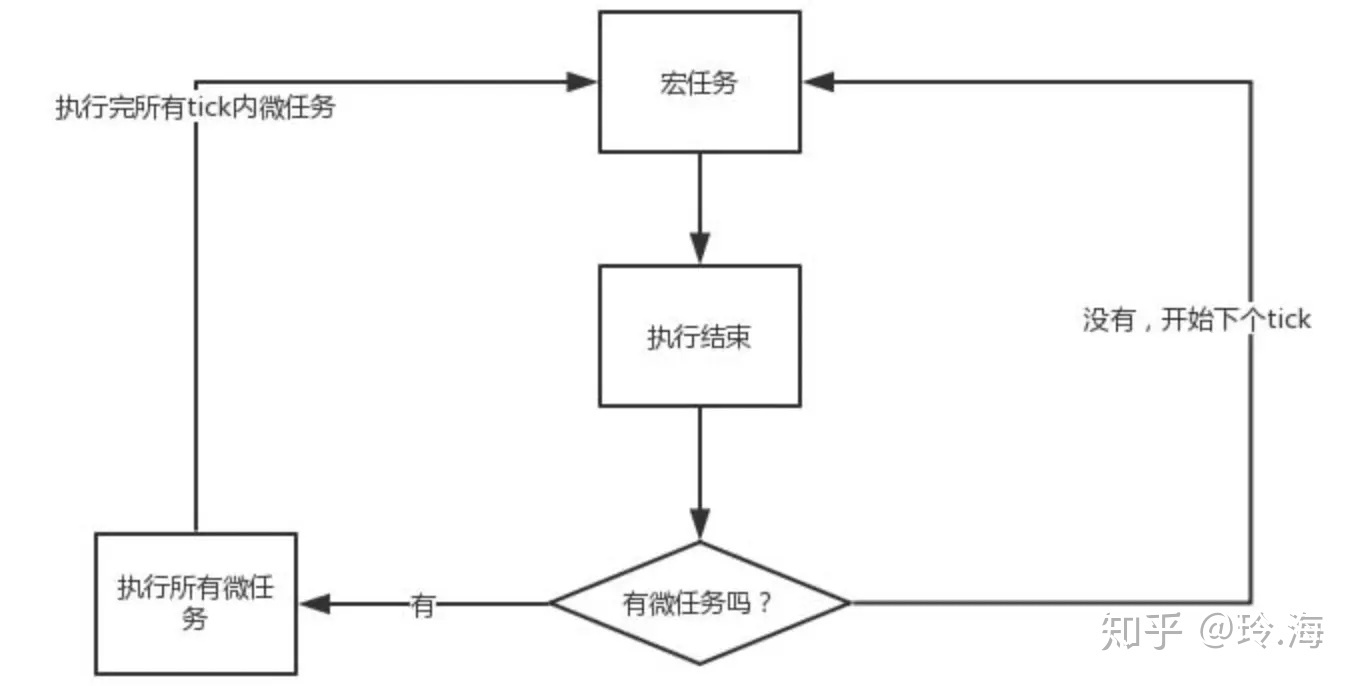
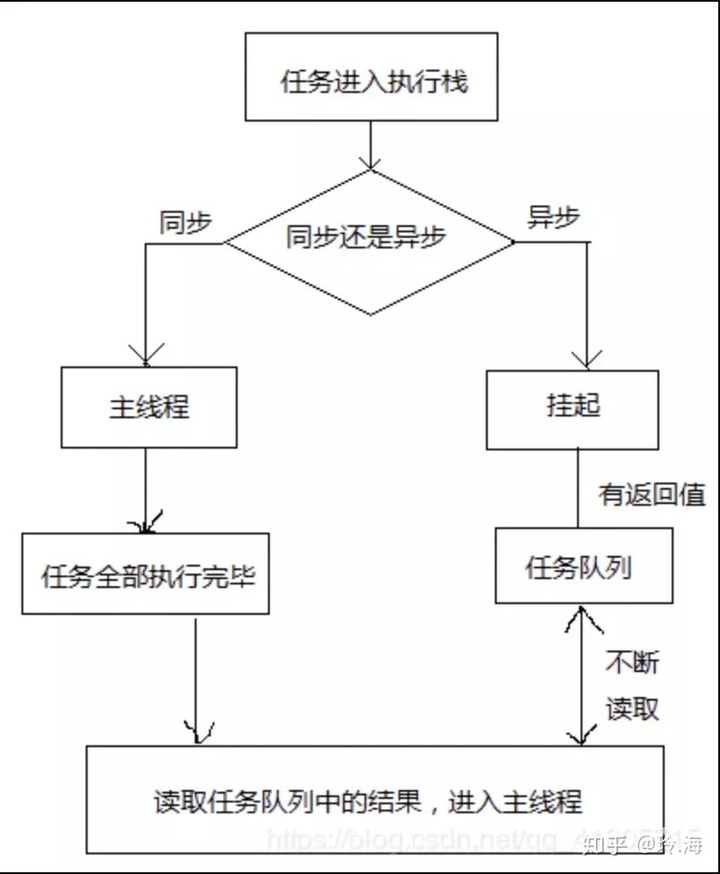
众所周知，JavaScript 是一门单线程语言，虽然在 html5 中提出了 Web-Worker ，

但这并未改变 JavaScript 是单线程这一核心,可是浏览器又能很好的处理异步请求，

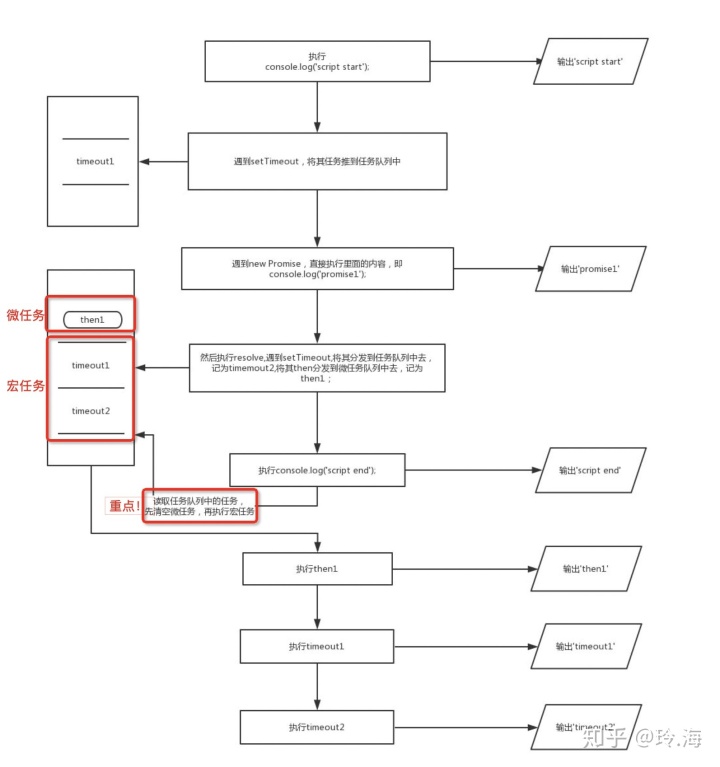
那么到底是为什么呢？

**浏览器执行线程**  
浏览器是多进程的，其中有浏览器渲染进程，其包含的线程有：GUI 渲染线程（负责渲染页面，解析 HTML，CSS 构成 DOM 树）、JS 引擎线程、事件触发线程、定时器触发线程、http 请求线程等主要线程  
  
主线程之外，事件触发线程管理着一个任务队列，异步任务会被主线程挂起，不会进入主线程，而是进入任务队列。只要异步任务有了运行结果，就会在队列任务中放置一个事件；  
一旦执行栈中所有的同步任务执行完毕后，系统就会读取任务队列，将可运行的异步任务添加到可执行栈中，开始执行。

事件循环是指主线程重复从事件队列中取消息、执行的过程。指整个执行流程。  
  
事件队列是一个存储着待执行任务的序列，其中的任务严格按照时间先后顺序执行，排在队头的任务会率先执行，而排在队尾的任务会最后执行。



* 宏任务主要包含：script( 整体代码)、setTimeout、setInterval、I/O、UI 交互事件、setImmediate(Node.js 环境)
* 微任务主要包含：Promise、MutaionObserver、process.nextTick(Node.js 环境)



**7. Promise 有什么缺陷，可以如何解决？**

Promise虽然跳出了异步嵌套的怪圈，用链式表达更加清晰，但是我们也发现如果有大量的异步请求的时候，流程复杂的情况下，会发现充满了屏幕的then，看起来非常吃力，而ES7的Async/Await的出现就是为了解决这种复杂的情况。

Async/Await的增加，可以让接口按顺序异步获取数据，用更可读，可维护的方式处理回调。

**async声明的函数的返回本质上是一个Promise**。

**await是在等待一个Promise的异步返回**

**async函数完全可以看作多个异步操作，包装成的一个 Promise 对象，而await命令就是内部then命令的语法糖**。

现在我们看一下实战：

const setDelay = (millisecond) => {

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

if (typeof millisecond != 'number') reject(new Error('参数必须是number类型'));

setTimeout(()=> {

resolve(`我延迟了${millisecond}毫秒后输出的`)

}, millisecond)

})

}

const setDelaySecond = (seconds) => {

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

if (typeof seconds != 'number' || seconds > 10) reject(new Error('参数必须是number类型，并且小于等于10'));

setTimeout(()=> {

resolve(`我延迟了${seconds}秒后输出的，注意单位是秒`)

}, seconds \* 1000)

})

}

比如上面两个延时函数（写在上面），比如我想先延时1秒，在延迟2秒，再延时1秒，最后输出“完成”，这个过程，如果用then的写法，大概是这样（嵌套地狱写法出门右拐不送）：

setDelay(1000)

.then(result=>{

console.log(result);

return setDelaySecond(2)

})

.then(result=>{

console.log(result);

return setDelay(1000)

})

.then(result=>{

console.log(result);

console.log('完成')

})

.catch(err=>{

console.log(err);

})

咋一看是不是挺繁琐的？如果逻辑多了估计看得更累，现在我们来试一下async/await

(async ()=>{

const result = await setDelay(1000);

console.log(result);

console.log(await setDelaySecond(2));

console.log(await setDelay(1000));

console.log('完成了');

})()

看！是不是没有冗余的长长的链式代码，语义化也非常清楚，非常舒服，那么你看到这里，一定还发现了，上面的catch我们是不是没有在async中实现？接下去我们就分析一下async/await如何处理错误？

**可以使用try...catch语句：**

(async ()=>{

try{

const result = await setDelay(1000);

console.log(result);

console.log(await setDelaySecond(2));

console.log(await setDelay(1000));

console.log('完成了');

} catch (e) {

console.log(e); // 这里捕获错误

}

})()