(注1：文章链接)

你真的懂Promise吗

前言

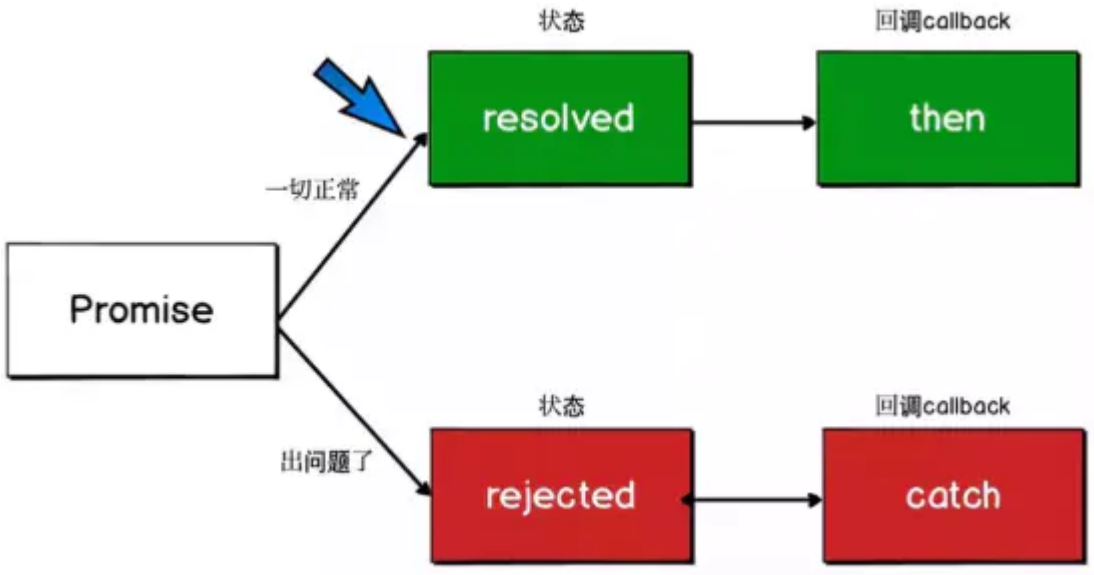
在异步编程中，Promise 扮演了举足轻重的角色，比传统的解决方案（回调函数和事件）更合理和更强大。可能有些小伙伴会有这样的疑问：2020年了，怎么还在谈论Promise？事实上，有些朋友对于这个几乎每天都在打交道的“老朋友”，貌似全懂,但稍加深入就可能疑问百出，本文带大家深入理解这个熟悉的陌生人—— Promise.

基本用法

1.语法

```
new Promise( function(resolve, reject) {...} /* executor */  )
复制代码
```

- 构建 Promise 对象时，需要传入一个 executor 函数，主要业务流程都在 executor 函数中执行。
- Promise构造函数执行时立即调用executor 函数， resolve 和 reject 两个函数作为参数传递给executor，resolve 和 reject 函数被调用时，分别将promise的状态改为fulfilled（完成）或rejected（失败）。**一旦状态改变，就不会再变**，任何时候都可以得到这个结果。
- 在 executor 函数中调用 resolve 函数后，会触发 promise.then 设置的回调函数；而调用 reject 函数后，会触发 promise.catch 设置的回调函数。



值得注意的是，**Promise 是用来管理异步编程的，它本身不是异步的**，new Promise的时候会立即把executor函数执行，只不过我们一般会在executor函数中处理一个异步操作。比如下面代码中，一开始是会先打印出2。

```
let p1 = new Promise(()=>{
    setTimeout(()=>{
      console.log(1)
    },1000)
    console.log(2)
  })
console.log(3) // 2 3 1
复制代码
```

Promise 采用了回调函数延迟绑定技术，在执行 resolve 函数的时候，回调函数还没有绑定，那么只能**推迟回调函数的执行。**这具体是啥意思呢？我们先来看下面的例子：

```
let p1 = new Promise((resolve,reject)=>{
  console.log(1);
  resolve('浪里行舟')
  console.log(2)
})
// then:设置成功或者失败后处理的方法
p1.then(result=>{
 //p1延迟绑定回调函数
  console.log('成功 '+result)
},reason=>{
  console.log('失败 '+reason)
})
console.log(3)
// 1
// 2
// 3
// 成功 浪里行舟
复制代码
```

new Promise的时候先执行executor函数，打印出 1、2，Promise在执行resolve时，触发微任务，还是继续往下执行同步任务， 执行p1.then时，存储起来两个函数（此时这两个函数还没有执行）,然后打印出3，此时同步任务执行完成，最后执行刚刚那个微任务，从而执行.then中成功的方法。

错误处理

Promise 对象的错误**具有“冒泡”性质，会一直向后传递**，直到被 onReject 函数处理或 catch 语句捕获为止。具备了这样“冒泡”的特性后，就不需要在每个 Promise 对象中单独捕获异常了。

要遇到一个then，要执行成功或者失败的方法，但如果此方法并没有在当前then中被定义，则顺延到下一个对应的函数

```
function executor (resolve, reject) {
  let rand = Math.random()
  console.log(1)
  console.log(rand)
  if (rand > 0.5) {
    resolve()
  } else {
    reject()
```

```
  }
}
var p0 = new Promise(executor)
var p1 = p0.then((value) => {
  console.log('succeed-1')
  return new Promise(executor)
})
var p2 = p1.then((value) => {
  console.log('succeed-2')
  return new Promise(executor)
})
p2.catch((error) => {
  console.log('error', error)
})
console.log(2)
复制代码
```

这段代码有三个 Promise 对象：p0～p2。无论哪个对象里面抛出异常，都可以通过最后一个对象 p2.catch 来捕获异常，通过这种方式可以将所有 Promise 对象的错误合并到一个函数来处理，这样就解决了每个任务都需要单独处理异常的问题。

通过这种方式，我们就消灭了嵌套调用和频繁的错误处理，这样使得我们写出来的代码更加优雅，更加符合人的线性思维。

Promise链式调用

我们都知道可以把多个Promise连接到一起来表示一系列异步骤。这种方式可以实现的关键在于以下两个Promise固有行为特性：

- 每次你对Promise调用then，它都会创建并返回一个新的Promise，我们可以将其链接起来；
- 不管从then调用的完成回调（第一个参数）返回的值是什么，它都会被自动设置为被链接Promise（第一点中的）的完成。

先通过下面的例子，来解释一下刚刚这段话是什么意思，然后详细介绍下链式调用的执行流程

```
let p1=new Promise((resolve,reject)=>{
    resolve(100) // 决定了下个then中成功方法会被执行
})
// 连接p1
let p2=p1.then(result=>{
    console.log('成功1 '+result)
    return Promise.reject(1)
// 返回一个新的Promise实例，决定了当前实例是失败的，所以决定下一个then中失败方法会被执行
},reason=>{
    console.log('失败1 '+reason)
    return 200
})
// 连接p2
let p3=p2.then(result=>{
    console.log('成功2 '+result)
},reason=>{
    console.log('失败2 '+reason)
})
```

```
// 成功1 100
// 失败2 1
复制代码
```

我们通过返回 Promise.reject(1)，完成了第一个调用then创建并返回的promise p2。p2的then调用在运行时会从return Promise.reject(1) 语句接受完成值。当然，p2.then又创建了另一个新的promise，可以用变量p3存储。

new Promise出来的实例，成功或者失败，取决于executor函数执行的时候，**执行的是resolve还是reject决定的**，或**executor函数执行发生异常错误**，这两种情况都会把实例状态改为失败的。

p2执行then返回的新实例的状态，决定下一个then中哪一个方法会被执行，有以下几种情况：

- 不论是成功的方法执行，还是失败的方法执行（then中的两个方法），凡是执行抛出了异常，则都会把实例的状态改为失败。
- 方法中如果返回一个新的Promise实例（比如上例中的Promise.reject(1)），返回这个实例的结果是成功还是失败，也决定了当前实例是成功还是失败。
- 剩下的情况基本上都是让实例变为成功的状态，上一个then中方法返回的结果会传递到下一个then的方法中。

我们再来看个例子

```
new Promise(resolve=>{
    resolve(a) // 报错
// 这个executor函数执行发生异常错误，决定下个then失败方法会被执行
}).then(result=>{
    console.log(`成功：${result}`)
    return result*10
},reason=>{
    console.log(`失败：${reason}`)
// 执行这句时候，没有发生异常或者返回一个失败的Promise实例，所以下个then成功方法会被执行
// 这里没有return，最后会返回 undefined
}).then(result=>{
    console.log(`成功：${result}`)
},reason=>{
    console.log(`失败：${reason}`)
})
// 失败：ReferenceError: a is not defined
// 成功：undefined
复制代码
```

async & await

从上面一些例子，我们可以看出，虽然使用 Promise 能很好地解决回调地狱的问题，但是这种方式充满了 Promise 的 then() 方法，如果处理流程比较复杂的话，那么整段代码将充斥着 then，语义化不明显，代码不能很好地表示执行流程。

ES7中新增的异步编程方法，async/await的实现是基于 Promise的，简单而言就是async 函数就是返回Promise对象，是generator的语法糖。很多人认为async/await是异步操作的终极解决方案：

- 语法简洁，更像是同步代码，也更符合普通的阅读习惯；
- 改进JS中异步操作串行执行的代码组织方式，减少callback的嵌套；
- Promise中不能自定义使用try/catch进行错误捕获，但是在Async/await中可以像处理同步代码处理错误。

不过也存在一些缺点，因为 await 将异步代码改造成了同步代码，如果多个异步代码没有依赖性却使用了 await 会导致性能上的降低。

```
async function test() {
  // 以下代码没有依赖性的话，完全可以使用 Promise.all 的方式
  // 如果有依赖性的话，其实就是解决回调地狱的例子了
  await fetch(url1)
  await fetch(url2)
  await fetch(url3)
}
复制代码
```

观察下面这段代码，你能判断出打印出来的内容是什么吗?

```
let p1 = Promise.resolve(1)
let p2 = new Promise(resolve => {
  setTimeout(() => {
    resolve(2)
  }, 1000)
})
async function fn() {
  console.log(1)
// 当代码执行到此行（先把此行），构建一个异步的微任务
// 等待promise返回结果，并且await下面的代码也都被列到任务队列中
  let result1 = await p2
  console.log(3)
  let result2 = await p1
  console.log(4)
}
fn()
console.log(2)
// 1 2 3 4
复制代码
```

如果 await 右侧表达逻辑是个 promise，await会等待这个promise的返回结果，**只有返回的状态是resolved情况**，才会把结果返回,如果promise是失败状态，则await不会接收其返回结果，await下面的代码也不会在继续执行。

```
let p1 = Promise.reject(100)
async function fn1() {
  let result = await p1
  console.log(1) //这行代码不会执行
}
复制代码
```

我们再来看道比较复杂的题目：

```
console.log(1)
setTimeout(()=>{console.log(2)},1000)
async function fn(){
    console.log(3)
```

```
    setTimeout(()=>{console.log(4)},20)
    return Promise.reject()
}
async function run(){
    console.log(5)
    await fn()
    console.log(6)
}
run()
//需要执行150ms左右
for(let i=0;i<90000000;i++){}
setTimeout(()=>{
    console.log(7)
    new Promise(resolve=>{
        console.log(8)
        resolve()
    }).then(()=>{console.log(9)})
},0)
console.log(10)
// 1 5 3 10 4 7 8 9 2
复制代码
```

做这道题之前，读者需明白：

- 基于微任务的技术有 MutationObserver、Promise 以及以 Promise 为基础开发出来的很多其他的技术，本题中resolve()、await fn()都是微任务。
- 不管宏任务是否到达时间，以及放置的先后顺序，每次主线程执行栈为空的时候，引擎会优先处理微任务队列，**处理完微任务队列里的所有任务**，再去处理宏任务。

接下来，我们一步一步分析：

- 首先执行同步代码，输出 1，遇见第一个setTimeout，将其回调放入任务队列（宏任务）当中，继续往下执行
- 运行run(),打印出 5，并往下执行，遇见 await fn()，将其放入任务队列（微任务）
- await fn() 当前这一行代码执行时，fn函数会立即执行的,打印出3，遇见第二个setTimeout，将其回调放入任务队列（宏任务），await fn() 下面的代码需要等待返回Promise成功状态才会执行，所以6是不会被打印的。
- 继续往下执行，遇到for循环同步代码，需要等150ms,虽然第二个setTimeout已经到达时间，但不会执行，遇见第三个setTimeout，将其回调放入任务队列（宏任务），然后打印出10。值得注意的是，这个定时器 推迟时间0毫秒实际上达不到的。根据HTML5标准，setTimeOut推迟执行的时间，最少是4毫秒。
- 同步代码执行完毕，此时没有微任务，就去执行宏任务，上面提到已经到点的setTimeout先执行，打印出4
- 然后执行下一个setTimeout的宏任务，所以先打印出7，new Promise的时候会立即把executor函数执行，打印出8，然后在执行resolve时，触发微任务，于是打印出9
- 最后执行第一个setTimeout的宏任务，打印出2

常用的方法

1、Promise.resolve()

Promise.resolve(value)方法返回一个以给定值解析后的Promise 对象。 Promise.resolve()等价于下面的写法:

```
Promise.resolve('foo')
// 等价于
new Promise(resolve => resolve('foo'))
复制代码
```

Promise.resolve方法的参数分成四种情况。

（1）参数是一个 Promise 实例

如果参数是 Promise 实例，那么Promise.resolve将**不做任何修改、原封不动地**返回这个实例。

```
const p1 = new Promise(function (resolve, reject) {
  setTimeout(() => reject(new Error('fail')), 3000)
})
const p2 = new Promise(function (resolve, reject) {
  setTimeout(() => resolve(p1), 1000)
})
p2
  .then(result => console.log(result))
  .catch(error => console.log(error))
// Error: fail
复制代码
```

上面代码中，p1是一个 Promise，3 秒之后变为rejected。p2的状态在 1 秒之后改变，resolve方法返回的是p1。由于p2返回的是另一个 Promise，导致p2自己的状态无效了，由p1的状态决定p2的状态。所以，后面的then语句都变成针对后者（p1）。又过了 2 秒，p1变为rejected，导致触发catch方法指定的回调函数。

（2）参数不是具有then方法的对象，或根本就不是对象

```
Promise.resolve("Success").then(function(value) {
 // Promise.resolve方法的参数，会同时传给回调函数。
  console.log(value); // "Success"
}, function(value) {
  // 不会被调用
});
复制代码
```

（3）不带有任何参数

Promise.resolve()方法允许调用时不带参数，直接返回一个resolved状态的 Promise 对象。如果希望得到一个 Promise 对象，比较方便的方法就是直接调用Promise.resolve()方法。

```
Promise.resolve().then(function () {
  console.log('two');
});
console.log('one');
// one two
复制代码
```

（4）参数是一个thenable对象

thenable对象指的是具有then方法的对象,Promise.resolve方法会将这个对象转为 Promise 对象，然后就立即执行thenable对象的then方法。

```
let thenable = {
  then: function(resolve, reject) {
    resolve(42);
  }
};
let p1 = Promise.resolve(thenable);
p1.then(function(value) {
  console.log(value);  // 42
});
复制代码
```

2、Promise.reject()

Promise.reject()方法返回一个带有拒绝原因的Promise对象。

```
new Promise((resolve,reject) => {
    reject(new Error("出错了"));
});
// 等价于
 Promise.reject(new Error("出错了"));

// 使用方法
Promise.reject(new Error("BOOM!")).catch(error => {
    console.error(error);
});
复制代码
```

值得注意的是，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**。所以，最好在它们前面加上return语句，这样就不会有意外。

```
new Promise((resolve, reject) => {
  return reject(1);
  // 后面的语句不会执行
  console.log(2);
})
复制代码
```

3、Promise.all()

```
let p1 = Promise.resolve(1)
let p2 = new Promise(resolve => {
  setTimeout(() => {
    resolve(2)
  }, 1000)
})
let p3 = Promise.resolve(3)
Promise.all([p3, p2, p1])
```

```
  .then(result => {
 // 返回的结果是按照Array中编写实例的顺序来
    console.log(result) // [ 3, 2, 1 ]
  })
  .catch(reason => {
    console.log("失败:reason")
  })
复制代码
```

Promise.all 生成并返回一个新的 Promise 对象，所以它可以使用 Promise 实例的所有方法。参数传递promise数组中**所有的 Promise 对象都变为resolve的时候**，该方法才会返回， 新创建的 Promise 则会使用这些 promise 的值。

如果参数中的**任何一个promise为reject的话**，则整个Promise.all调用会**立即终止**，并返回一个reject的新的 Promise 对象。

4、Promise.allSettled()

有时候，我们不关心异步操作的结果，只关心这些操作有没有结束。这时，ES2020 引入Promise.allSettled()方法就很有用。如果没有这个方法，想要确保所有操作都结束，就很麻烦。Promise.all()方法无法做到这一点。

假如有这样的场景：一个页面有三个区域，分别对应三个独立的接口数据，使用 Promise.all 来并发请求三个接口，如果其中任意一个接口出现异常，状态是reject,这会导致页面中该三个区域数据全都无法出来，显然这种状况我们是无法接受，Promise.allSettled的出现就可以解决这个痛点：

```
Promise.allSettled([
  Promise.reject({ code: 500, msg: '服务异常' }),
  Promise.resolve({ code: 200, list: [] }),
  Promise.resolve({ code: 200, list: [] })
]).then(res => {
  console.log(res)
  /*
    0: {status: "rejected", reason: {…}}
    1: {status: "fulfilled", value: {…}}
    2: {status: "fulfilled", value: {…}}
  */
  // 过滤掉 rejected 状态，尽可能多的保证页面区域数据渲染
  RenderContent(
    res.filter(el => {
      return el.status !== 'rejected'
    })
  )
})
复制代码
```

Promise.allSettled跟Promise.all类似, 其参数接受一个Promise的数组, 返回一个新的Promise, **唯一的不同在于, 它不会进行短路**, 也就是说当Promise全部处理完成后,我们可以拿到每个Promise的状态, 而不管是否处理成功。

5、Promise.race()

Promise.all()方法的效果是"谁跑的慢，以谁为准执行回调"，那么相对的就有另一个方法"谁跑的快，以谁为准执行回调"，这就是Promise.race()方法，这个词本来就是赛跑的意思。race的用法与all一样，接收一个promise对象数组为参数。

Promise.all在接收到的所有的对象promise都变为FulFilled或者Rejected状态之后才会继续进行后面的处理，与之相对的是Promise.race**只要有一个promise对象进入FulFilled或者Rejected状态的话**，就会继续进行后面的处理。

```
// `delay`毫秒后执行resolve
function timerPromisefy(delay) {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve(delay);
        }, delay);
    });
}
// 任何一个promise变为resolve或reject的话程序就停止运行
Promise.race([
    timerPromisefy(1),
    timerPromisefy(32),
    timerPromisefy(64)
]).then(function (value) {
    console.log(value);    // => 1
});
复制代码
```

上面的代码创建了3个promise对象，这些promise对象会分别在1ms、32ms 和 64ms后变为确定状态，即FulFilled，并且在第一个变为确定状态的1ms后，.then注册的回调函数就会被调用。

6、Promise.prototype.finally()

ES9 新增 finally() 方法返回一个Promise。在promise结束时，无论结果是fulfilled或者是rejected，都会执行指定的回调函数。**这为在Promise是否成功完成后都需要执行的代码提供了一种方式。**这避免了同样的语句需要在then()和catch()中各写一次的情况。

比如我们发送请求之前会出现一个loading，当我们请求发送完成之后，不管请求有没有出错，我们都希望关掉这个loading。

```
this.loading = true
request()
  .then((res) => {
    // do something
  })
  .catch(() => {
    // log err
  })
  .finally(() => {
    this.loading = false
  })
复制代码
```

finally方法的回调函数不接受任何参数，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。

实际应用

假设有这样一个需求：红灯 3s 亮一次，绿灯 1s 亮一次，黄灯 2s 亮一次；如何让三个灯不断交替重复亮灯？ 三个亮灯函数已经存在：

```
function red() {
    console.log('red');
}
function green() {
    console.log('green');
}
function yellow() {
    console.log('yellow');
}
复制代码
```

这道题复杂的地方在于**需要“交替重复”亮灯**，而不是亮完一遍就结束的一锤子买卖，我们可以通过递归来实现：

```
// 用 promise 实现
let task = (timer, light) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (light === 'red') {
        red()
      }
      if (light === 'green') {
        green()
      }
      if (light === 'yellow') {
        yellow()
      }
      resolve()
    }, timer);
  })
}
let step = () => {
  task(3000, 'red')
    .then(() => task(1000, 'green'))
    .then(() => task(2000, 'yellow'))
    .then(step)
}
step()
复制代码
```

同样也可以通过async/await 的实现：

```
//  async/await 实现
let step = async () => {
  await task(3000, 'red')
  await task(1000, 'green')
  await task(2000, 'yellow')
  step()
}
step()
复制代码
```

使用 async/await 可以实现用同步代码的风格来编写异步代码,毫无疑问，还是 async/await 的方案更加直观，不过深入理解Promise 是掌握async/await的基础。

参考资料

- MDN 文档
- 你了解Promise吗?
- 浏览器工作原理与实践
- Web前端开发高级工程师
- 前端开发核心知识进阶
- 再学JavaScript ES(6-10)全版本语法大全
- 《ECMAScript 6 入门教程》
- 《你不知道的javascript(中卷)》