目錄

其它

前言

为解决异步函数的回调陷阱，开发社区不断摸索，终于折腾出 Promise/A+。它的优势非常显著：

1. 不增加新的语法，可以立刻适配几乎所有浏览器
2. 以队列的形式组织代码，易读好改
3. 捕获异常方案也基本可用

这套方案在迭代中逐步完善，最终被吸收进 ES2015。不仅如此，ES2017 中还增加了 Await/Async，可以用顺序的方式书写异步代码，甚至可以正常抛出捕获错误，维护同一个栈。可以说彻底解决了异步回调的问题。

现在大部分浏览器和 Node.js 都已原生支持 Promise，很多类库也开始返回 Promise 对象，更有各种降级适配策略。Node.js 7+ 则实装了 Await/Async。如果您现在还不会使用，那么我建议您尽快学习一下。本文我准备结合近期的开发经验，全面介绍现代化的 JavaScript 异步开发。

关于这本书

这本书最开始的内容源于我在 SegmentFault 做的一场分享： Promise 的 N 种用法。

后来去 GitChat 分享时，选题比较犯难，就跟运营妹子商量要不再做一次同样题目，这样我也省事儿。结果挂出来后乏人问津，只好再跟运营开会讨论，然后决定更名为《JavaScript 异步开发攻略》，并且增加非 Promise 的内容。改名后终于过了下限，然后我就动手写，于是便有了这本小书。

我写这本书时就打算根据环境发展随时更新内容，没想到 GitChat 设计成后期不允许修改文章，要改只能把内容发给运营人工修改。因为我写的时候为了分章节好维护好管理，用的是 Git + Gitbook，索性脆放到 Gitbook 上好了。名字也改成现在这个样子，增加一个“全”字。

目标读者要求

1. 前端水平：初级、中级

2. 了解 JavaScript
3. 最好有异步开发经历，希望写出更好的代码

名词及约定

- ES6 = ES2015
- ES7 = ES2016 + ES2017
- 异步函数 = Async Functions = Await/Async

范例代码会使用 ES6 的语法，也会混用 ES6 Module 和 CommonJS，请大家不要见怪。我会在代码当中加注释，其中会有一些关键内容，请大家不要忽略。

本文中所有代码均以 Node.js 7.0 为基础。

作者介绍

大家好，我叫翟路佳，花名“肉山”，这个名字跟 Dota 没关系，从高中起伴随我到现在。

我热爱编程，喜欢学习，喜欢分享，从业十余年，投入的比较多，学习积累到的也比较多，对前端方方面面都有所了解，希望能与大家分享。

我兴趣爱好比较广泛，尤其喜欢旅游，欢迎大家相互交流。

你可以在这里找到我：

- 博客
- 微博
- GitHub

版权许可

反馈

如果您对于文中的内容有任何疑问，请在评论或 Issue 中告诉我。亦可发邮件给我：meathill[at]gmail.com。谢谢。

异步的问题

之所以会出现这样那样的解决方案，我之所以写这样的文章介绍这些解决方案，肯定是异步本身有问题。

是的，异步就是那样让人难以割舍，又那样让人不易亲近。

异步的起源

故事必须从头说起，在很久很久以前……

为校验表单，JavaScript 诞生了

在那个拨号上网的洪荒年代，浏览器还非常初级，与服务器进行数据交互的唯一方式就是提交表单。用户填写完成之后，交给服务器处理，如果内容合规当然好，如果不合规就麻烦了，必须打回来重填。那会儿网速还是论 Kb 的，比如我刚上网那会儿开始升级到 33.6Kb，主流还是 22.4Kb……

所以很容易想象：当用户填完100+选项，按下提交按钮，等待几十秒甚至几分钟之后，反馈回来的信息却是："您的用户名不能包含大写字母"，他会有多么的崩溃多么的想杀人。为了提升用户体验，网景公司的布兰登·艾克大约用10天时间，开发出 JavaScript 的原型，从此，这门注定改变世界的语言就诞生了。

只是当时大家都还没有认识到这一点，发明它的目的，只是为校验表单。

JavaScript 中存在大量异步计算

同样为了提升用户体验，HTML DOM 也选择了边加载、边生成、边渲染的策略。再加上要等待用户操作，大量交互都以事件来驱动。于是，JavaScript 里很早就存在着大量的异步计算。

这也带来一个好处，作为一门 UI 语言，异步操作帮 JavaScript 避免了页面冻结。

为什么异步操作可以避免界面冻结呢？

同步的利弊

> 假设你去到一家饭店，自己找座坐下了，然后招呼服务员拿菜单来。
>
> 服务员说："对不起，我是'同步'服务员，我要服务完这张桌子才能招呼你。"
>
> 那一桌人明明已经吃上了，你只是想要菜单，这么小的一个动作，服务员却要你等待别人的一个大动作完成。你是不是很想抽ta？

这就是“同步”的问题：顺序交付的工作**1234**，必须按照**1234**的顺序完成。

不过“同步”也有“同步”的好处：逻辑非常简单。你不用担心每步操作会消耗多少时间，反正每一步操作都会在上一步完成之后才进行，只管往后写就是了。

异步的利弊

与之相反，异步，则是将耗时很长的 A 交付的工作交给系统之后，就去继续做 B 交付的工作。等到系统完成之后，再通过回调或者事件，继续做 A 剩下的工作。

从观察者的角度，看起来 AB 工作的完成顺序，和交付他们的时间顺序无关，所以叫“异步”。

那些需要大量计算（比如 Service Worker），或者复杂查询（比如 Ajax）的工作，JS 引擎把它们交给系统之后，就立刻返回继续待机了，于是再进行什么操作，浏览器也能第一时间响应，这让用户的感觉非常好。

有利必有弊，异步的缺点就是：必须通过特殊的语法才能实现，而这些语法就不如同步那样简单、清晰、明了。

异步计算的实现

异步计算有两种常见的实现形式。

事件侦听

这种形式在浏览器里比较常见，比如，我们可以对一个 `<button>` 的用户点击行为增加侦听，在点击事件触发后调用函数进行处理。

```
document.getElementById('#button').addEventListener('click', fun
ction (event) {
  // do something
}, false);
```

也可以使用 DOM 节点的 `onclick` 属性绑定侦听函数：

```
document.getElementById('#button').onclick = function (event) {
  // do something
}
```

回调

到了 Node.js（以及其它 Hybrid 环境），由于要和引擎外部的环境进行交互，大部分操作都变成回调。比如用 `fs.readFile()` 读取文件内容：

```
const fs = require('fs');

fs.readFile('path/to/file.txt', 'utf8', (err, content) => {
  if (err) {
    throw err;
  }
  console.log(content);
});
```

如果你不熟悉 Node.js 也没关系，jQuery 里也有类似的操作，最常见的就是侦听页面加载状态，加载完成后启动回调函数：

```
$(function () {
  // 绑定事件
  // 创建组件
  // 以及其它操作
});
```

异步的问题

回调陷阱

这个问题其实是最直观的问题，也是大家谈的最多的问题。比如下面这段代码：

```
a(function (resultA) {
  b(resultA, function (resultB) {
    c(resultB, function (resultC) {
      d(resultC, function (resultD) {
        e(resultD, function (resultE) {
          f(resultE, function (resultF) {
            // 子子孙孙无穷尽也
            console.log(resultF);
          });
        });
      });
    });
  });
});
```

嵌套层次之深令人发指。这种代码很难维护，有人称之为“回调地狱”，有人称之为“回调陷阱”，还有人称之为“回调金字塔”，其实都无所谓，带来的问题很明显：

1. 难以维护。 上面这段只是为演示写的示范代码，还算好懂；实际开发中，混杂了业务逻辑的代码更多更长，更难判定函数范围，再加上闭包导致的变量使用，那真的难以维护。
2. 难以复用。 回调的顺序确定下来之后，想对其中的某些环节进行复用也很困难，牵一发而动全局，可能只有全靠手写，结果就会越搞越长。

更严重的问题

面试的时候，问到回调的问题，如果候选人只能答出“回调地狱，难以维护”，在我这里顶多算不功不过，不加分。要想得到满分必须能答出更深层次的问题。

为了说明这些问题，我们先来看一段代码。假设有这样一个需求：

> 遍历目录，找出最大的一个文件。

```
// 这段代码来自于 https://medium.com/@wavded/managing-node-js-call
back-hell-1fe03ba8baf 我加入了一些自己的理解
/**
 * @param dir 目标文件夹
 * @param callback 完成后的回调
 */
function findLargest(dir, callback) {
  fs.readdir(dir, function (err, files) {  // [1]
    if (err) return callback(err); // {1}
    let count = files.length; // {2}
    let errored = false; // {2}
    let stats = []; // {2}
    files.forEach( file => { // [2]
      fs.stat(path.join(dir, file), (err, stat) => { // [3]
        if (errored) return; // {1}
        if (err) {
          errored = true;
          return callback(err);
        }
        stats.push(stat); // [4] {2}

        if (--count === 0) { // [5] {2}
          let largest = stats
            .filter(function (stat) { return stat.isFile(); })
            .reduce(function (prev, next) {
              if (prev.size > next.size) return prev;
              return next;
            });
          callback(null, files[stats.indexOf(largest)]); // [6]
        }
      });
    });
  });
}

findLargest('./path/to/dir', function (err, filename) { // [7]
```

```
  if (err) return console.error(err);
  console.log('largest file was:', filename);
});
```

这里我声明了一个函数 `findLargest()`，用来查找某一个目录下体积最大的文件。它的工作流程如下（参见代码中的标记“[n]”）：

1. 使用 `fs.readdir` 读取一个目录下的所有文件
2. 对其结果 `files` 进行遍历
3. 使用 `fs.readFile` 读取每一个文件的属性
4. 将其属性存入 `stats` 数组
5. 每完成一个文件，就将计数器减一，直至为0，再开始查找体积最大的文件
6. 通过回调传出结果
7. 调用此函数的时候，需传入目标文件夹和回调函数；回调函数遵守 Node.js 风格，第一个参数为可能发生的错误，第二个参数为实际结果

断开的栈与 `try/catch`

我们再来看标记为“{1}”的地方。在 Node.js 中，几乎所有异步方法的回调函数都是这种风格：

```
/**
 * @param err 可能发生的错误
 * @param result 正确的结果
 */
function (err, result) {
  if (err) { // 如果发生错误
    return callback(err);
  }

  // 如果一切正常
  callback(null, result);
}
```

通常来说，错误处理的一般机制是“捕获” -> “处理”，即 `try/catch`，但是这里我们都没有用，而是作为参数调用回调函数，甚至要一层一层的通过回调函数传出去。为什么呢？

无论是事件还是回调，基本原理是一致的：

> 把当前语句执行完；把不确定完成时间的计算交给系统；等待系统唤起回调。

于是栈被破坏了，无法进行常规的 **`try/catch`** 。

我们知道，函数执行是一个“入栈/出栈”的过程。当我们在 A 函数里调用 B 函数的时候，JS 引擎就会先把 A 压到栈里，然后再把 B 压到栈里；B 运行结束后，出栈，然后继续执行 A；A 也运行完毕后，出栈，栈已清空，这次运行结束。

这个时候，我们如果中断代码执行，可以检索完整的堆栈，完整的作用域链（闭包），获取任何我们想获取的信息。

可是异步回调函数（包括事件处理函数，下同）不完全如此，比如上面的代码，无论是 `fs.readdir` 还是 `fs.readFile` ，都不会直接调用回调函数，而是继续执行其它代码，直至完成，出栈。眞正调用回到函数的是引擎，并且是启用一个新栈，压入栈成为第一个函数。所以如果回调报错，一方面，我们无法获取之前启动异步计算时栈里的信息，不容易判定什么导致了错误；另一方面，套在 `fs.readdir` 外面的 `try/catch` ，也根本捕获不到这个错误。

结论：回调函数的栈与启动异步操作的栈断开了，无法正常使用 `try/catch` 。

迫不得已使用外层变量

我们再来看代码中标记为“{2}”的地方。我在这里声明了3个变量， `count` 用来记录待处理文件的数量； `errored` 用来记录有没有发生错误； `stats` 用来记录文件状态。

这3个变量会在 `fs.stat()` 的回调函数中使用。因为我们没法确定这些异步操作的完成顺序，所以只能用这种方式判断是否所有文件都已读取完毕。虽然基于闭包的设计，这样做一定行得通，但是，操作外层作用域的变量，还是存在一些隐患。比如，这些变量同样也可以被其它同一作用域的函数访问并且修改。

我们平时说“关注点集中”，哪里的变量就在哪里声明哪里使用哪里释放，就是为了避免这种情况。

同样的原理，在第二个“{1}”这里，因为遍历已经执行完，触发回调的时候已经无力回天，所以只能根据外层作用域的记录，逐个判断。

结论：同时执行多个异步回调时，因为没法预期它们的完成顺序，所以必须借助外层作用域的变量。

小结

我们回来总结一下，异步回调的传统做法有四个问题：

1. 嵌套层次很深，难以维护
2. 代码难以复用
3. 堆栈被破坏，无法正常检索，也无法正常使用 `try/catch/throw`
4. 多个异步计算同时进行，无法预期完成顺序，必须借助外层作用域的变量，有误操作风险

异步的发展

最初，在浏览器环境下，大家遭遇的异步导致的问题还不是很严重。因为那会儿以事件侦听为主，大部分处理函数不需要嵌套很多层，也就是 Ajax 批量加载资源的时候可能有些头大，平时不怎么能听到这方面的抱怨。（所以大家都跑去做模组解决方案了，并没有在这方面很上心。）

但是当 Node.js 问世之后，对异步的依赖一下子加剧了。

因为那个时候，后端语言无论是 PHP、Java、Python 都已经相当成熟，Node.js 想要在服务器端站稳脚跟，必须有独到之处。于是，异步运算带来的无阻塞高并发就成了 Node.js 的镇店之宝、主打功能。

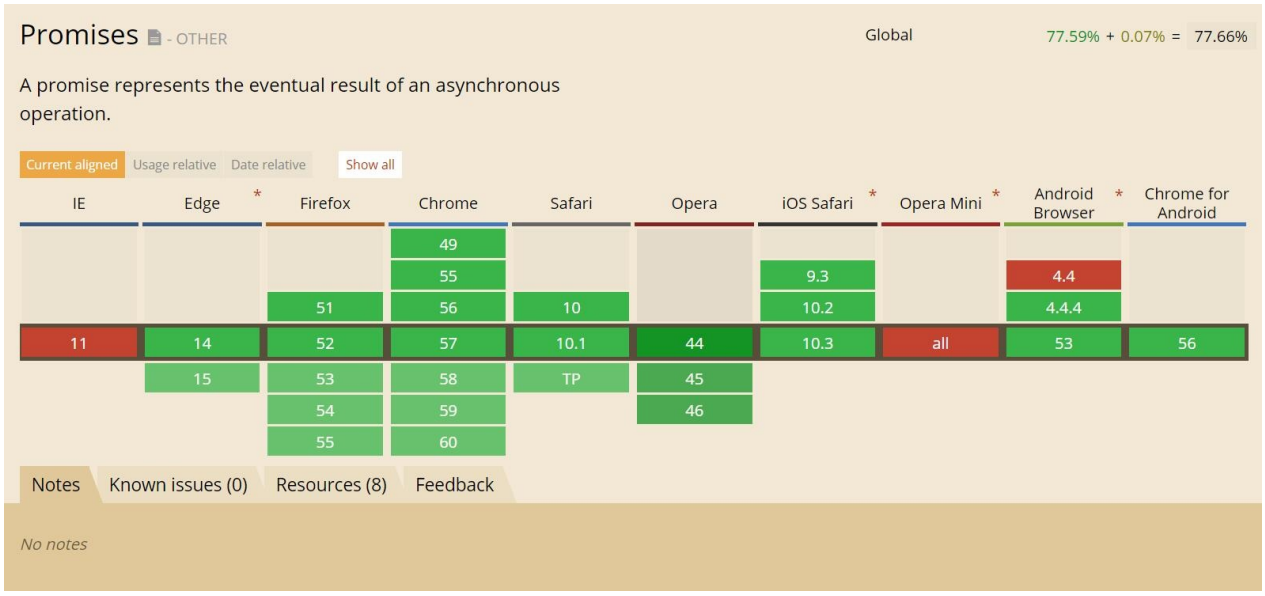
然而写了才知道，虽然能无阻塞高并发，但是无数层嵌套的回调函数也使得代码维护与重构变得异常困难，抱怨之声四起，大家方开始更努力的探索解决方案。

最终，Promise/A+ 被摸索出来。

Promise 方案

> 有没有一种方案，既能保留异步在无阻塞上的优势，又能让我们写代码写的更舒服呢？

社区经过长时间探索，最终总结出 Promise/A+ 方案，并且纳入 ES2015 规范，如今，大部分运行环境都已经原生支持它。



Promise 的支持情况

这套方案有以下好处：

1. 可以很好的解决回调嵌套问题
2. 代码阅读体验很好
3. 不需要新的语言元素

这套方案由众多开发者共同探索得来，不同类库的实现略有不同，但日后都会支持 ES2015 标准。本文也以其为准。

Promise 详解

我们先来看 Promise 的用法。

```
new Promise(
  /* 执行器 executor */
  function (resolve, reject) {
    // 一段可能耗时很长的异步操作
    doAsyncAction(function callback(err, result) {
      if (err) {
        return reject(err);
      }
      resolve(result);
    });
  }
)
  .then(function resolve() {
    // 成功，下一步
  }, function reject() {
    // 失败，做相应处理
  });
```

Promise 是一个代理对象，它和原先的异步操作并无关系。它接受一个“执行器 executor”作为参数，我们把原先要执行的异步（并非一定要异步，以后会说明，这里你可以先不深究）操作放进去。

执行器在 Promise 实例创建后立刻开始执行。执行器自带两个参数：`resolve` 和 `reject`，这二位都是函数，执行它们会改变 Promise 实例的状态。

Promise 实例有三个状态：

1. `pending` [待定] 初始状态，新创建的实例处于这个状态
2. `fulfilled` [实现] 操作成功，在执行器里调用 `resolve()` 之后，实例切换到这个状态
3. `rejected` [被否决] 操作失败，在执行器里调用 `reject()` 之后，实例切换到这个状态

Promise 实例状态改变之后，就会触发后面对应的 `.then()` 参数里的函数，继续执行后续步骤。另外，Promise 的实例状态只会改变一次，确定为 `fulfilled` 或 `rejected` 之一后就不会再变。

一个简单的例子

```
new Promise( resolve => {
  setTimeout( () => {
    resolve('hello');
  }, 2000);
})
  .then( value => {
    console.log( value + ' world');
  });

// 输出：
// hello world
```

这是最简单的一种情况。执行器里面是一个定时器，2秒种之后，它会执行 `resolve('hello')` ，将 Promise 实例的状态置为 `fulfilled` ，并传出返回值 `'hello'` 。接下来 `.then()` 里面的 `resolve` 响应函数就会被触发，它接受前面 Promise 返回的值 `'hello'` ，将其与 `' world'` 连接起来，输出“hello world”。

再来一个稍复杂的例子

```
new Promise( resolve => {
  setTimeout( () => {
    resolve('hello');
  }, 2000);
})
  .then( value => {
    return new Promise( resolve => {
      setTimeout( () => {
        resolve('world')
      }, 2000);
    });
  })
  .then( value => {
    console.log( value + ' world');
  });

// 输出：
// world world
```

这个例子与上一个例子的不同之处在于，我在 `.then()` 的后面又跟了一个 `.then()`。并且在第一个 `.then()` 里又返回了一个 Promise 实例。于是，第二个 `.then()` 就又等了2秒才执行，并且接收到的参数是第一个 `.then()` 返回的 Promise 返回的 `'world'` (好拗口)，而不是起始 Promise 返回的 `'hello'`。

这就必须说明 Promise 里 `.then()` 的设计。

.then()

`.then()` 其实接受两个函数作为参数，分别代表 `fulfilled` 状态时的处理函数和 `rejected` 状态时的处理函数。只不过通常情况下，我会建议大家使用 `.catch()` 捕获 `rejected` 状态。这个后面还会说到，暂时按下不表。

`.then()` 会返回一个新的 Promise 实例，所以它可以链式调用，如前面的例子所示。当前面的 Promise 状态改变时，`.then()` 会执行特定的状态响应函数，并将其结果，调用自己的 Promise 的 `resolve()` 返回。

Promise.resolve()

这里必须插入介绍一下 Promise.resolve()。它是 Promise 的静态方法，可以返回一个状态为 `fulfilled` 的 Promise 实例。这个方法非常重要，其它方法都需要用它处理参数。

它可以接受四种不同类型的参数，并且返回不同的值：

1. 参数为空，返回一个 `fulfilled` 实例，响应函数的参数也为空
2. 参数不为空、也不是 Promise 实例，返回 `fulfilled` 实例，只不过响应函数能得到这个参数
3. 参数为 Promise 实例，直接原样返回
4. 参数为 thenable 对象，立刻执行它的 `.then()`

用一段代码作为示范吧，比较简单，就不一一解释了。

```
Promise.resolve()
  .then( () => {
    console.log('Step 1');
    return Promise.resolve('Hello');
  })
  .then( value => {
    console.log(value, 'World');
    let p = new Promise( resolve => {
      setTimeout(() => {
        resolve('Good');
      }, 2000);
    });
    return Promise.resolve(p);
  })
  .then( value => {
    console.log(value, ' evening');
    return Promise.resolve({
      then() {
        console.log(', everyone');
      }
    })
  });

// 输出：
// Step 1
// Hello World
// （2秒之后） Good evening
// , everyone
```

提醒大家一下，标准定义的 Promise 不能被外界改变状态，只能在执行器里执行 `resolve` 或者 `reject` 来改变。

继续 `.then()` 的话题

结合上一小节关于 Promise.resolve() 的讲解，我们应该可以推断出 `.then()` 里的状态响应函数返回不同结果对进程的影响了吧。所以在连续 `.then()` 的例子中，第一个 `.then()` 返回了新的 Promise 实例，第二个就会等待其完成后才会触发。

好的，接下来我们换一个思路，假如一个 Promise 已经完成了，再给它加一个 `.then()` ，会有什么效果呢？我们来试一下。

```
let promise = new Promise(resolve => {
  setTimeout(() => {
    console.log('the promise fulfilled');
    resolve('hello, world');
  }, 1000);
});

setTimeout(() => {
  promise.then( value => {
    console.log(value);
  });
}, 3000);

// 输出
// （1秒后）the promise fulfilled
// （3秒后）hello, world
```

1秒后，Promise 完成；3秒后，给它续上一个 `.then()` ，因为它已经处于 `fulfilled` 状态，所以立刻执行响应函数，输出“hello, world”。

这一点很值得我们关注。Promise 从队列操作中脱胎而成，带有很强的队列属性。异步回调开始执行后，我们无法追加操作，也无法判定结束时间。而使用 Promise 的话，我们不需要关心它什么时候开始什么时候结束，只需要在后面追加操作，即可。另外，Promise 是一个普通对象，我们也可以像对待别的对象那样将它进行传递。这为它带来了独特的价值。

Promise 小测验

好的，我们已经初步了解了 Promise 的使用方式；了解了 `.then()` 怎么处理响应函数；知道 `.then()` 会返回新的 Promise 实例，所以可以链式调用；还知道 `Promise.resolve()` 是怎么转化普通类型到 Promise 类型的。可能有些同学看到这里，自我感觉良好，觉得已经学完了。所以接下来我们来做一个小测验，看看大家的掌握程度。

> 下面这道题出自 We have a problem with promises

问题：下面的四种 **promises** 的区别是什么？

（假定 `doSomething()` 和 `doSomethingElse()` 都会返回 Promise 对象。另外请尽量在足够宽的屏幕上观看，不然换行可能会影响视觉效果。）

```
// #1
doSomething().then(function () {
  return doSomethingElse();
});

// #2
doSomething().then(function () {
  doSomethingElse();
});

// #3
doSomething().then(doSomethingElse());

// #4
doSomething().then(doSomethingElse);
```

仔细看一看，默想一下答案，不要着急往下翻。

好，准备好了么？我们继续了哟。

3...

2..

1.

答案揭晓

第一题

```
doSomething()
  .then(function () {
    return doSomethingElse();
  })
  .then(finalHandler);
```

答案：

```
doSomething
|-----------|
            doSomethingElse(undefined)
            |------------|
                         finalHandler(resultOfDoSomethingElse)
                         |------------|
```

这道题比较简单，几乎和前面的例子一样，我就不多说了。

第二题

```
doSomething()
  .then(function () {
    doSomethingElse();
  })
  .then(finalHandler);
```

答案：

```
doSomething
|-----------------|
                  doSomethingElse(undefined)
                  |------------------|
                  finalHandler(undefined)
                  |------------------|
```

这道题就有一定难度了。虽然 `doSomethingElse` 会返回 Promise 对象，但是因为 `.then()` 的响应函数并没有把它 `return` 出来，所以这里其实相当于 `return null`。我们知道，`Promise.resolve()` 在参数为空的时候会返回一个状态为 `fulfilled` 的 Promise，所以这里两步是几乎一起执行的。

第三题

```
doSomething()
  .then(doSomethingElse())
  .then(finalHandler);
```

答案：

```
doSomething
|-----------------|
doSomethingElse(undefined)
|---------------------------------|
                  finalHandler(resultOfDoSomething)
                  |------------------|
```

这一题的语法陷阱也不小。首先，`doSomethingElse` 和 `doSomethingElse()` 的区别在于，前者是变量，指向一个函数；而后者是则是直接执行了函数，并返回其返回值。所以这里 `doSomethingElse` 立刻就开始执行了，和前面 `doSomething` 的启动时间相差无几，可以忽略不计。然后，按照 Promise 的设计，当 `.then()` 的参数不是函数的时候，这一步会被忽略不计，所以 `doSomething` 完成后就跳去执行 `finalHandler` 了。

第四题

```
doSomething()
  .then(doSomethingElse)
  .then(finalHandler);
```

答案：

```
doSomething
|-----------|
            doSomethingElse(resultOfDoSomething)
            |------------|
                         finalHandler(resultOfDoSomethingElse)
                         |------------------|
```

这一题比较简单，就不解释了。

怎么样？都答对了么？还是有点小难度的，对吧？

Promise 进阶

我们继续学习 Promise。

错误处理

Promise 会自动捕获内部异常，并交给 `rejected` 响应函数处理。比如下面这段代码：

```
new Promise( resolve => {
  setTimeout( () => {
    resolve();
  }, 2000);
  throw new Error('bye');
})
  .then( value => {
    console.log( value + ' world');
  })
  .catch( error => {
    console.log( 'Error: ', error.message);
  });

// 立刻输出：
// Error: bye
```

可以看到，原定2s之后 `resolve()` 并没有出现，因为在 Promise 的执行器里抛出了错误，所以立刻跳过了 `.then()`，进入 `.catch()` 处理异常。

这里需要注意，如果把抛出错误的语句放到回调函数里，则是另外一副光景：

```
new Promise( resolve => {
  setTimeout( () => {
    throw new Error('bye');
  }, 2000);
})
  .then( value => {
    console.log( value + ' world');
  })
  .catch( error => {
    console.log( 'It\'s an Error: ', error.message);
  });

// (2s之后)输出：
// ./code/2-3-catch-error.js:3
//     throw new Error('bye');
//     ^

// Error: bye
//     at Timeout.setTimeout [as _onTimeout] (/Users/meathill/Do
cuments/Book/javascript-async-tutorial/code/2-3-catch-error.js:3
:11)
//     at ontimeout (timers.js:488:11)
//     at tryOnTimeout (timers.js:323:5)
//     at Timer.listOnTimeout (timers.js:283:5)
```

正如异步的问题分析的那样，异步回调中，异步函数的栈，和回调函数的栈，不是一个栈。所以 Promise 的执行器只能捕获到异步函数抛出的错误，无法捕获回调函数抛出的错误。

回到上面这段代码，当回调函数抛出错误时，我们没有捕获处理，运行时就出面捕获了，于是报错、回调栈被终结。此时，Promise 对象的状态并未被改变，所以下面的 `.then()` 响应函数和 `.catch()` 响应函数都没有触发，我们看到的，只是默认的错误输出。（2017-07-19 更新）

这也印证了 Promise 的问题：它没有引入新的语法元素，所以无法摆脱栈断裂带来的问题。在错误处理方面，它只是“能用”，并不好用，无法达到之前的开发体验。

reject

正如我们前面所说，`.then(fulfilled, reject)` 其实接收两个参数，分别作为成功与失败的回调。不过在实践中，我更推荐上面的做法，即不传入第二个参数，而是把它放在后面的 `.catch()` 里面。这样有两个好处：

1. 更加清晰，更加好读
2. 可以捕获前面所有 `.then()` 的错误，而不仅是这一步的错误

> 在小程序里需要注意，抛出的错误会被全局捕获，而 `.catch` 反而不执行，所以该用两个参数还是要用。

更复杂的情况

当队列很长的时候，情况又如何呢？我们看一段代码：

```
new Promise(resolve => {
  setTimeout(() => {
    resolve();
  }, 1000);
})
  .then( () => {
    console.log('start');
    throw new Error('test error');
  })
  .catch( err => {
    console.log('I catch: ', err);

    // 下面这一行的注释将引发不同的走向
    // throw new Error('another error');
  })
  .then( () => {
    console.log('arrive here');
  })
  .then( () => {
    console.log('... and here');
  })
  .catch( err => {
    console.log('No, I catch: ', err);
  });

// 输出：
// start
// I catch: test err
// arrive here
// ... and here
```

实际上，`.catch()` 仍然会使用 `Promise.resolve()` 返回其中的响应函数的执行结果，与 `.then()` 并无不同。所以 `.catch()` 之后的 `.then()` 仍然会执行，如果想彻底跳出执行，就必须继续抛出错误，比如把上面代码中的 `another error` 那行注释掉。这也需要大家注意。

所以，我们也可以下结论，Promise 并没有真正请回 `try/catch/throw`，它只是模拟了一个 `.catch()` 出来，可以在一定程度上解决回调错误的问题，但是距离真正还原栈关系，正常使用 `try/catch/throw` 其实还很远。

小结

简单总结一下 Promise 的错误处理。与异步回调相比，它的作用略强，可以抛出和捕获错误，基本可以按照预期的状态执行。然而它仍然不是真正的 `try/catch/throw` 。

`.catch()` 也使用 `Promise.resolve()` 返回 `fulfilled` 状态的 Promise 实例，所以它后面的 `.then()` 会继续执行，在队列很长的时候，也容易出错，请大家务必小心。

另外，所有执行器和响应函数里的错误都不会真正进入全局环境，所以我们有必要在所有队列的最后一步增加一个 `.catch()` ，防止遗漏错误造成意想不到的问题。

```
doSomething()
  .doAnotherThing()
  .doMoreThing()
  .catch( err => {
    console.log(err);
  });
```

在 Node.js v7 之后，没有捕获的内部错误会触发一个 Warning，大家可以用来发现错误。

Promise 进阶

接下来的时间，我会继续把 Promise 的知识补完。

`Promise.reject(reason)`

这个方法比较简单，就返回一个状态为 `rejected` 的 Promise 实例。

它接受一个参数 `reason`，作为状态说明，交由后面的 `.catch()` 捕获。为了与其它异常处理共用一个 `.catch()`，我们可以用 `Error` 实例作为 `reason`。

另外，`Promise.reject()` 也不认 `thenable`。

```
let error = new Error('something wrong');
Promise.reject(error)
  .then( value => {
    console.log('it\'s ok');
    console.log(value);
  })
  .catch( err => {
    console.log('no, it\'s not ok');
    console.log(err);

    return Promise.reject({
      then() {
        console.log('it will be ok');
      },
      catch() {
        console.log('not yet');
      }
    });
  });
```

`Promise.all([p1, p2, p3, ....])`

`Promise.all([p1, p2, p3, ....])` 用于将多个 Promise 实例，包装成一个新的 Promise 实例。

它接受一个数组（其实是 `iterable`，不过我觉得暂时不要引入更多概念了.....）作为参数，数组里可以是 Promise 对象，也可以是别的值，这些值都会交给 `Promise.resolve()` 处理。当所有子 Promise 都完成，该 Promise 完成，返回值是包含全部返回值的数组。有任何一个失败，该 Promise 失败，`.catch()` 得到的是第一个失败的子 Promise 的错误。

```
Promise.all([1, 2, 3])
  .then( all => {
    console.log('1: ', all);
    return Promise.all([ function () {
      console.log('ooxx');
    }, 'xxoo', false]);
  })
  .then( all => {
    console.log('2: ', all);
    let p1 = new Promise( resolve => {
      setTimeout(() => {
        resolve('I\'m P1');
      }, 1500);
    });
    let p2 = new Promise( resolve => {
      setTimeout(() => {
        resolve('I\'m P2');
      }, 1450)
    });
    return Promise.all([p1, p2]);
  })
  .then( all => {
    console.log('3: ', all);
    let p1 = new Promise( resolve => {
      setTimeout(() => {
        resolve('I\'m P1');
      }, 1500);
    });
    let p2 = new Promise( (resolve, reject) => {
      setTimeout(() => {
```

```
        reject('I\'m P2');
      }, 1000);
    });
    let p3 = new Promise( (resolve , reject) => {
      setTimeout(() => {
        reject('I\'m P3');
      }, 3000);
    });
    return Promise.all([p1, p2, p3]);
  })
  .then( all => {
    console.log('all', all);
  })
  .catch( err => {
    console.log('Catch: ', err);
  });

// 输出：
// 1:  [ 1, 2, 3 ]
// 2:  [ [Function], 'xxoo', false ]
// 3:  [ 'I\'m P1', 'I\'m P2' ]
// Catch:  I'm P2
```

这里很容易懂，就不一一解释了。

常见用法

`Promise.all()` 最常见就是和 `.map()` 连用。

我们改造一下前面的例子。

```
// FileSystem.js
const fs = require('fs');

module.exports = {
  readDir: function (path, options) {
    return new Promise( resolve => {
      fs.readdir(path, options, (err, files) => {
        if (err) {
```

```
          throw err;
        }
        resolve(files);
      });
    });
  },
  stat: function (path, options) {
    return new Promise( resolve => {
      fs.stat(path, options, (err, stat) => {
        if (err) {
          throw err;
        }
        resolve(stat);
      });
    });
  }
};

// main.js
const fs = require('./FileSystem');

function findLargest(dir) {
  return fs.readDir(dir, 'utf-8')
    .then( files => {
      return Promise.all( files.map( file => fs.stat(file) ));
    })
    .then( stats => {
      let biggest = stats.reduce( (memo, stat) => {
        if (stat.isDirectory()) {
          return memo;
        }
        if (memo.size < stat.size) {
          return stat;
        }
        return memo;
      });
      return biggest.file;
    })
    .catch(console.log.bind(console));
}
```

```
findLargest('some/path/')
  .then( file => {
    console.log(file);
  })
  .catch( err => {
    console.log(err);
  });
```

在这个例子当中，我使用 Promise 将 `fs.stat` 和 `fs.readdir` 进行了封装，让其返回 Promise 对象。然后使用 `Promise.all()` + `Array.prototype.map()` 方法，就可以进行遍历，还可以避免使用外层作用域的变量。

Promise.race([p1, p2, p3, ...])

`Promise.race()` 的功能和用法与 `Promise.all()` 十分类似，也接受一个数组作为参数，然后把数组里的值都用 `Promise.resolve()` 处理成 Promise 对象，然后再返回一个新的 Promise 实例。只不过这些子 Promise 有任意一个完成，`Promise.race()` 返回的 Promise 实例就算完成，并且返回完成的子实例的返回值。

它最常见的用法，是作超时检查。我们可以把异步操作和定时器放在一个 `Promise.race()` 里，如果定时器触发时异步操作还没返回，就可以认为超时了，然后就可以给用户一些提示。

```
let p1 = new Promise(resolve => {
  // 这是一个长时间的调用，我们假装它就是正常要跑的异步操作
  setTimeout(() => {
    resolve('I\'m P1');
  }, 10000);
});
let p2 = new Promise(resolve => {
  // 这是个稍短的调用，假装是一个定时器
  setTimeout(() => {
    resolve(false);
  }, 2000)
});
Promise.race([p1, p2])
  .then(value => {
    if (value) {
      console.log(value);
    } else {
      console.log('Timeout, Yellow flower is cold');
    }
  });

// 输出：
// Timeout, Yellow flower is cold
```

注意，这里 `p1` 也就是原本就要执行的异步操作并没有被中止，它只是没有在预期的时间内返回而已。所以一方面可以继续等待它的返回值，另一方面也要考虑服务器端是否需要做回滚处理。

Promise 嵌套

这种情况在初涉 Promise 的同学的代码中很常见，大概是这么个意思：

```
new Promise( resolve => {
  console.log('Step 1');
  setTimeout(() => {
    resolve(100);
  }, 1000);
})
  .then( value => {
    return new Promise(resolve => {
      console.log('Step 1-1');
      setTimeout(() => {
        resolve(110);
      }, 1000);
    })
      .then( value => {
        console.log('Step 1-2');
        return value;
      })
      .then( value => {
        console.log('Step 1-3');
        return value;
      });
  })
  .then(value => {
    console.log(value);
    console.log('Step 2');
  });
```

因为 `.then()` 返回的也是 Promise 实例，所以外层的 Promise 会等里面的 `.then()` 执行完再继续执行，所以这里的执行顺序稳定为从上之下，左右无关，“1 > 1-1 > 1-2 > 1-3 > 2”。但是从阅读体验和维护效率的角度来看，最好把它展开：

```
new Promise( resolve => {
  console.log('Step 1');
  setTimeout(() => {
    resolve(100);
  }, 1000);
})
  .then( value => {
    return new Promise(resolve => {
      console.log('Step 1-1');
      setTimeout(() => {
        resolve(110);
      }, 1000);
    });
  })
  .then( value => {
    console.log('Step 1-2');
    return value;
  })
  .then( value => {
    console.log('Step 1-3');
    return value;
  })
  .then(value => {
    console.log(value);
    console.log('Step 2');
  });
```

二者是完全等价的，后者更容易阅读。

队列

有时候我们不希望所有动作一起发生，而是按照一定顺序，逐个进行。这样的形式，就是队列。在我看来，队列是 Promise 的核心价值，即使是异步函数在大部分浏览器和 Node.js 里实装的今天，队列也仍有其独特的价值。

用 Promise 实现队列的方式很多，这里兹举两例：

```
// 使用 Array.prototype.forEach
function queue(things) {
  let promise = Promise.resolve();
  things.forEach( thing => {
    // 这里很容易出错，如果不把 `.then()` 返回的新实例赋给 `promise` 的话，就不是队列，而是批量执行
    promise = promise.then( () => {
      return new Promise( resolve => {
        doThing(thing, () => {
          resolve();
        });
      });
    });
  });
  return promise;
}

queue(['lots', 'of', 'things', ....]);

// 使用 Array.prototype.reduce
function queue(things) {
  return things.reduce( (promise, thing) => {
    return promise.then( () => {
      return new Promise( resolve => {
        doThing(thing, () => {
          resolve();
        });
      });
    });
  }, Promise.resolve());
}

queue(['lots', 'of', 'things', ....]);
```

这个例子如此直接我就不再详细解释了。下面我们看一个相对复杂的例子，假设需求：

> 开发一个爬虫，抓取某网站。

```
const spider = require('spider');

function fetchAll(urls) {
  return urls.reduce((promise, url) => {
    return promise.then( () => {
      return fetch(url);
    });
  }, Promise.resolve());
}
function fetch(url) {
  return spider.fetch(url)
    .then( content => {
      return saveOrOther(content);
    })
    .then( content => {
      let links = spider.findLinks(content);
      return fetchAll(links);
    });
}
let url = ['http://blog.meathill.com/'];
fetchAll(url);
```

这段代码，我假设有一个蜘蛛工具（spider）包含基本的抓取和分析功能，然后循环使用 `fetch` 和 `fetchAll` 方法，不断分析抓取的页面，然后把页面当中所有的链接都加入到抓取队列当中。通过递归循环的方式，完成网站抓取。

Generator

如果你了解 Generator，你应该知道 Generator 可以在执行时中断，并等待被唤醒。如果能把它们连到一起使用应该不错。

```
let generator = function* (urls) {
  let loaded = [];
  while (urls.length > 0) {
    let url = urls.unshift();
    yield spider.fetch(url)
      .then( content => {
        loaded.push(url);
        return saveOrOther(content);
      })
      .then( content => {
        let links = spider.findLinks(content);
        links = _.without(links, loaded);
        urls = urls.concat(links);
      });
  }
  return 'over';
};

function fetch(urls) {
  let iterator = generator();

  function next() {
    let result = iterator.next();
    if (result.done) {
      return result.value;
    }
    let promise = iterator.next().value;
    promise.then(next);
  }

  next();
}

let urls = ['http://blog.meathill.com'];
fetch(urls);
```

Generator 可以把所有待抓取的 URL 都放到一个数组里，然后慢慢加载。从整体来看，暴露给外界的 `fetch` 函数其实变简单了很多。但是实现 Generator 本身有点费工夫，其中的利弊大家自己权衡吧。

小结

关于 Promise 的内容到此告一段落。这里我介绍了大部分的功能、函数和常见用法，有一些特殊情况会在后面继续说明。

Async Function

Async Function，即异步函数，为异步编程带来了非常大的提升，瞬间把开发效率和维护效率提升了一个数量级。

它的用法非常简单，只要用 `async` 关键字声明一个函数为“异步函数”，这个函数的内部就可以使用 await 关键字，让其中的语句等待异步执行的结果，然后再继续执行。我们还是用代码来说话吧：

```
function resolveAfter2Seconds(x) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(x);
    }, 2000);
  });
}

async function f1() {
  var x = await resolveAfter2Seconds(10);
  console.log(x);
}
f1();

// 输出：
// （2秒后）10
```

这段代码来自 MDN。里面先声明了 `resolveAfter2Seconds` 函数，执行它会返回一个 Promise 对象，等待2秒钟之后完成。然后声明了异步函数 `f1`，里面只有两行代码，第一行用 `await` 表示要等后面的 Promise 完成，再进行下一步；于是2秒之后，输出了“10”。

这段代码比较短，不太能体现异步函数的价值，我们改写一下查找最大文件的代码试试看：

```
const fs = require('./FileSystem');

async function findLargest(dir) {
  let files = await fs.readdir(dir);
  let stats = [];
  for (let i = 0, len = files.length; i < len; i++) {
    stats.push(await fs.stat(files[i]));
  }
  return stats
    .filter( stat => stat.isFile() )
    .reduce( (memo, stat) => {
      if (memo.size > stat.size) return memo;
      return stat;
    });
}

findLargest('path/to/dir')
  .then(file => {
    console.log(file);
  });
```

怎么样，是不是异常清晰明了，连代码量都减少了很多。而且，因为每次 `await` 都会等待后面的异步函数完成，所以我们还无意间写出了生成队列的代码。当然你可能不希望要队列，毕竟都异步了，队列的吸引力实在不大，那么我们只需要把 `for` 循环改成下面这个样子就行了：

```
let stats = await Promise.all(files.map( file => fs.stat(file) )
);
```

酷不酷！想不想学？

设计

见识过异步函数的强大之后，我们来捋一捋一下它的具体设计。

异步函数主要由两部分组成：

async

声明一个函数是异步函数。执行异步函数会返回一个 Promise 实例。异步函数的返回值会继续向后传递。

`async` 的语法比较简单，就在普通函数前面加个 `async` 而已。它也支持箭头函数，也可以用作立刻执行函数。

await

`await` 操作符只能用在异步函数的内部。表示等待一个 Promise 完成，然后返回其返回值。语法是这样的：

```
[return_value] = await expression;
```

- **return_value** 返回值：Promise 执行器的返回值
- **expression** 表达式：Promise 对象，其它类型会使用 `Promise.resolve()` 转换

执行到 `await` 之后，异步函数会暂停执行，等待表达式里的 Promise 完成。`resolve` 之后，返回 Promise 执行器的返回值，然后继续执行。如果 Promise 被 `rejected`，就抛出异常。

捕获错误

异步函数最大的改进其实就在捕获错误这里。它可以直接使用 `try/catch/throw` 就像我们之前那样。

比如，一款前后端分离的应用，论坛，需要用户登录。所以在初始化的时候就会向服务器请求用户信息，如果返回了，就继续加载用户关注列表等其它数据；如果返回 401 错误，就让用户登录。用 Promise 的话，我们可能会这样做：

```
function getMyProfile() {
  return fetch('api/me')
    .then(response => {
      if (response.ok) {
        return response.json();
      }
    })
    .then(profile => {
      global.profile = profile
    })
    .catch( err => {
      if (err.statusCode === 401) {
        location.href = '#/login';
      } else {
        // 处理其它错误
      }
    });
}
```

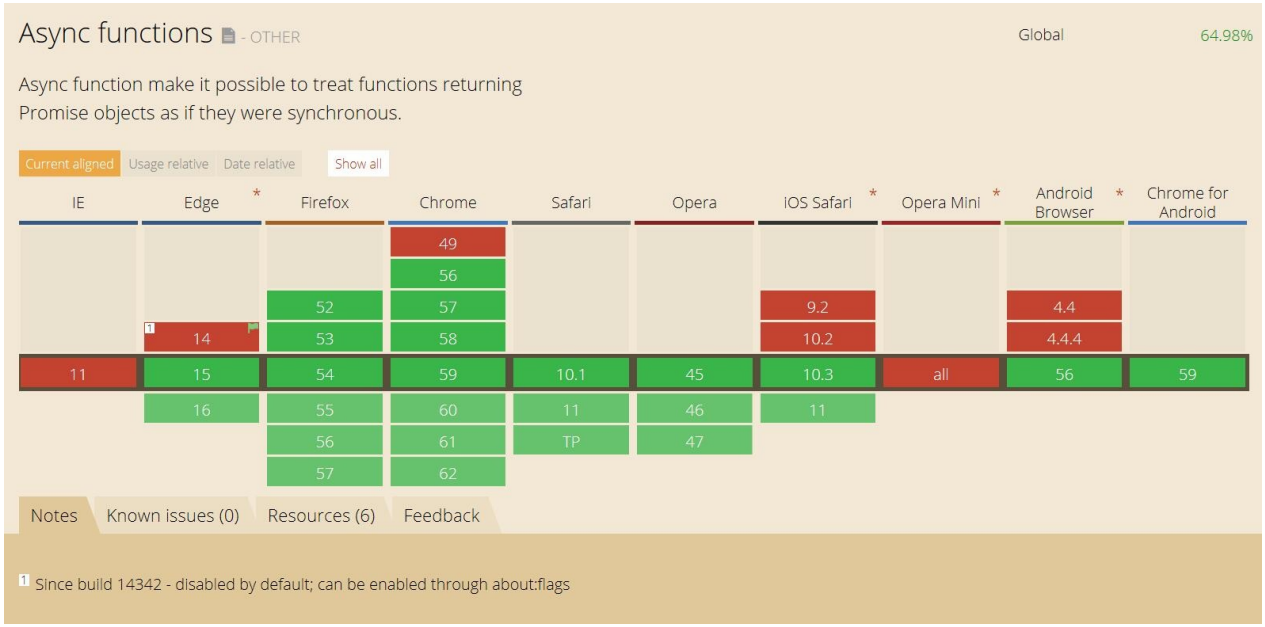
这样做也不是不可以，但是如果业务逻辑越来越复杂，队列也会越来越长，也许每一步都需要处理错误，那么队列就会变得难以维护。

如果用异步函数就会简单很多：

```
async function getMyProfile() {
  let profile;
  try {
    global.profile = await fetch('api/me');
  } catch (e) {
    if (e.statusCode === 401) {
      location.href = '#/login';
    } else {
      // 处理其它错误
    }
  }
}
```

另外，如果你真的捕获到错误，你会发现，它的堆栈信息是连续的，甚至可以回溯到调用异步函数的语句，继而可以审查所有堆栈里的变量。这对调试程序带来的帮助非常巨大。（需要运行时支持，转译的不行。）

普及率



截图于：2017-06-23

可以看出，大部分主流浏览器都已经支持异步函数，值得关注的，只有 iOS 10.2 和 Android 4。当然在国内的话，IE 始终是老大难问题……

Async Function VS Promise

异步函数的优势

异步函数对异步编程来说，有非常很大的提升。尤其在 `try/catch/throw` 方面，异步函数可以延续以前的标准写法，还能正常检索堆栈信息，优势非常大。

Promise 的价值

不过就目前来看，Promise（暂时）并不会被完全取代。

异步函数依赖 Promise

异步函数返回的是一个 Promise 对象；`await` 关键字只能等待 Promise 对象完成操作。

所以 Promise 的知识都还有效，大家也必须学习 Promise，才能更好的使用异步函数。

Promise 能更好的提供队列操作，并且在对象之间传递

Promise 本身是一个对象，所以可以在代码中任意传递。这意味着我们可以在程序的任何位置使用和维护队列，非常方便。

比如我们做一些办公产品，用户可能先做了 A，然后又做了 B。A 和 B 之间存在很强的先后顺序，不能乱。这个时候，队列的价值就能充分提现出来。

异步函数的支持率还不够

从上一节最后的截图可以看到，虽然异步函数的支持率已经很高了，但是在 iOS 10.2、Android 4 等关键平台上，还没有原生实现。这就需要我们进行降级兼容。

然而异步函数的降级代码很难写，Babel 转译后至少要增加3000行，对于一些轻量级应用来说代价不小。如果应用本身不大，或者异步操作并不复杂，用 Promise 可能是更好的选择。

小结

Promise 和异步函数之间不存在替代关系，根据需求和场景选择合适的技术，永远不会错。

实战环节

学会新的异步代码书写方式之后，我们自然希望改造之前的异步回调代码。下面我就带领大家来试一试。

将回调包装成 **Promise**

这是最常见的应用。它有三个显而易见的好处：

1. 可读性更好
2. 返回的结果可以加入任何 Promise 队列
3. 可以使用 Async Functions

我们就拿读取文件来举个例子。

```
// FileSystem.js
const fs = require('fs');

module.exports = {
  readFile: function (path, options) {
    return new Promise( resolve => {
      fs.readFile(path, options, (err, content) => {
        if (err) {
          throw err;
        }
        resolve(content);
      });
    });
  }
};

// promise.js
const fs = require('./FileSystem');

fs.readFile('../README.md', 'utf-8')
  .then(content => {
    console.log(content);
  });

// async.js
const fs = require('.FileSystem');

async function read(path) {
  let content = await fs.readFile(path);
  console.log(content);
}
read();
```

将任何异步操作包装成 Promise

不止回调，其实我们可以把任意异步操作都包装城 Promise。假设需求：

> 1. 弹出确认窗口，用户点击确认再继续，点击取消就中断
> 2. 由于样式有要求，不能使用 `window.confirm()`

之前我们的处理方式通常是：

```
something.on('done', function () { // 先做一些处理
  popup = openConfirmPopup('确定么'); // 弹出确认窗口
  popup.on('confirm', function goOn () { // 用户确认后继续处理
    // 继续处理
    next();
  });
});
```

如今，借助 Promise 的力量，我们可以把弹窗封装成 Promise，然后就可以将其融入队列，或者简单的使用 Async 等待操作完成。

```
function openConfirmPopup(msg) {
  let popup = createPopup(msg);
  return new Promise( (resolve, reject) => {
    popup.confirmButton.onclick = resolve;
    popup.cancelButton.onclick = reject;
  });
}

// pure promise
doSomething()
  .then(() => {
    return openConfirmPopup('确定么')
  })
  .then( () => {
    // 继续处理
    next();
  });

// async functions
await doSomething();
if (await openConfirmPopup('确定么')) {
  // 继续处理
  next();
}
```

Node.js 8 的新方法

Node.js 8 于今年5月底正式发布，带来了很多新特性。其中，`util.promisify()`，尤其值得我们注意。

为保证向下兼容，Node.js 里海量的异步回调函数自然都会保留，如果我们把每个函数都封装一遍，那真是齁麻烦齁麻烦，比齁还麻烦。

所以 Node.js 8 就提供了 `util.promisify()` ——“Promise 化”方法，方便我们把原来的异步回调方法瞬间改造成支持 Promise 的方法。接下来，想继续 `.then().then().then()` 搞队列，还是 Await 就看实际需要了。

> Bluebird 也提供了类似的方法，不妨看下它的文档。

我们看下官方范例。已知读取目录文件状态的 `fs.stat`，想得到支持 Promise 的版本，只需要这样做：

```
const util = require('util');
const fs = require('fs');

const stat = util.promisify(fs.stat);
stat('.')
  .then((stats) => {
    // Do something with `stats`
  })
  .catch((error) => {
    // Handle the error.
  });
```

怎么样，很简单吧？按照文档的说法，只要符合 Node.js 的回调风格，所有函数都可以这样转换。也就是说，只要满足下面两个条件，无论是不是原生方法，都可以：

1. 最后一个参数是回调函数
2. 回调函数的参数为 `(err, result)`，前面是可能的错误，后面是正常的结果

结合 Async Functions 使用

同样是上面的例子，如果想要结合 Async Functions，可以这样使用：

```
const util = require('util');
const fs = require('fs');

const stat = util.promisify(fs.stat);
async function readStats(dir) {
  try {
    let stats = await stat(dir);
    // Do something with `stats`
  } catch (err) { // Handle the error.
    console.log(err);
  }
}
readStats('.');
```

自定义 Promise 化处理函数

那如果现有的使用回调的函数不符合这个风格，还能用 `util.promisify()` 么？答案也是肯定的。我们只要给函数增加一个属性 `util.promisify.custom` ，指定一个函数作为 Promise 化处理函数，即可。请看下面的代码：

```
const util = require('util');

// 这就是要处理的使用回调的函数
function doSomething(foo, callback) {
  // ...
}

// 给它增加一个方法，用来在 Promise 化时调用
doSomething[util.promisify.custom] = function(foo) {
  // 自定义生成 Promise 的逻辑
  return getPromiseSomehow();
};

const promisified = util.promisify(doSomething);
console.log(promisified === doSomething[util.promisify.custom]);
// prints 'true'
```

如此一来，任何时候我们对目标函数 doSomething 进行 Promise 化处理，都会得到之前定义的函数。运行它，就会按照我们设计的特定逻辑返回 Promise 对象。

有了 `util.promisify`，升级异步回到函数，使用 Promise 或者异步函数真的方便了很多。

降级

IE

遗憾的是，除了 Edge 版本，IE 都不支持原生 Promise（更不要提异步函数了）。不过好在 Promise 不需要新的语言元素，所以我们完全可以用独立类库来补足。这里推荐 Q 和 Bluebird，它们都完全兼容 ES 规范，也就是说，前面介绍的用法都能奏效。

当然如果你不是非要 `new Promise()` 不可，用 jQuery 也完全可以。

jQuery

jQuery 早在 1.5 版本就开始这方面的探索，不过它起的名字不太一样，叫 Deferred 对象。实现也不太一样，有一些特殊的 API，比如 .done()。不过升级到 3.0 之后，jQuery 就完全兼容 ES2015 的 Promise 规范，并且通过了相关测试。所以如果使用新版本，我们大可以按照前面的教程来操作，只是 jQuery 需要使用工厂方法来创建 Promise 实例，与标准做法略有区别：

```
$.deferred(function (resolve) {
  // 执行异步操作吧
})
  .then( response => {
    // 继续下一步
  });
```

至于那些独有 API，我们当它们不存在就好了。另外，jQuery 的 jqXHR 也是 Promise 对象，所以完全可以用 `.then()` 方法操作：

```
$.ajax(url, {
  dataType: 'json'
})
  .then(json => {
    // 做后续操作
  });
```

异步函数

异步函数因为引入了新的语法元素，要想在比较古老的浏览器里使用，必须用 Babel 进行转译。

转译的时候我们需要考虑目标平台。异步函数的转译通常分为两步：

1. 转化为 generator
2. 兼容实现 generator

转译为 generator

所以如果目标平台支持 generator，那么只需要用 transform-async-to-generator 插件就好：

```
npm i --save-dev transform-async-to-generator
```

然后在 .babelrc 里启用

```
{
  "plugins": ["transform-async-to-generator"]
}
```

转译为 ES5

不过考虑到现实因素，支持 generator 的浏览器多半可以自动升级，很可能已经支持异步函数，转译的需求可能并不大。国内最大的转译原因还是根深蒂固的 Windows + IE，于是我们需要彻底转译成 ES5。

这时就需要同时多个插件共同工作了。包括 Syntax async functions、Regenerator transform、Async to generator transform、Runtime transform。这些插件会帮你把异步函数转译成 generator，然后转译成 regenerator，然后再引入 regenerator 库，最终实现在低版本浏览器里运行异步函数的效果。

```
npm i --save-dev babel-plugin-transform-regenerator babel-plugin
-syntax-async-functions babel-plugin-transform-runtime babel-plu
gin-transform-async-to-generator
```

然后在 .babelrc 里启用它们：

```
{
  "plugins": [
    "babel-plugin-transform-regenerator",
    "babel-plugin-syntax-async-functions",
    "babel-plugin-transform-async-to-generator",
    "babel-plugin-transform-runtime"
  ]
}
```

> 需要注意的是，这样至少会引入3000多行代码（据说），对轻量应用影响很大。另外，编译之后的代码，也无法享受到栈和错误捕获方面的优势。

还有一个偷懒的做法，就是索性把 ES2015、ES2017 通通转译成 ES5，这样代码量会增加很多，但配置相对简单：

```
npm i babel-preset-es2015 babel-preset-es2017 babel-plugin-trans
form-runtime --save-dev
```

```
{
  "presets": ["es2015", "es2017"],
  "plugins": ["transform-runtime]
}
```

还不会用 **Babel**？

Babel 转译，以及 Webpack 打包其实也包含不少的内容，这里我就不详细介绍了，尚不了解的同学请自行寻找资料学习。如果英文阅读能力尚可的话，直接看官网基本就够了。

- Babel 官网
- Webpack 官网
- Babel + Webpack 配置

在小程序中使用 Promise

国内开发者尤其是前端，肯定不能避开“小程序”这个话题。

从小程序 API 文档可以看出，大部分交互都要藉由异步回调来完成（我猜测这多半是跟原生应用交互导致的）。所以自然而然的，我也想用更好的方式去操作。

因为客户端的 WebView 中不支持原生 Promise，所以“微信 Web 开发工具”中也移除了对 Promise 的支持，需要我们自己处理。

好在正如之前所说，Promise 不需要引入新的语言元素，兼容性上佳，所以我们只要引用成熟的 Promise 类库就好。这里我选择 Bluebird。

安装

```
cd /path/to/my-xcx
npm install bluebird --save-dev
```

这样会把 Bluebird 安装在小程序目录的 `node_modules` 里。因为小程序并不支持从中加载，所以我们需要手工把 `node_modules/bluebird/browser/bluebird.js` 复制出来，放到小程序的 `utils` 目录内。

使用

使用时只需要 `import Promise from './utils/bluebird';`，就可以在开发环境中自由使用 `new Promise()` 了。这里拿 `wx.login` 作为例子：

```
import Promise from './utils/bluebird';

return new Promise(resolve => {
  wx.login({
    success(result) {
      resolve(result);
    }
  });
})
  .then(result => {
    console.log(result);
  });
```

注意事项

经过我的实际测试，在小程序里抛出错误并不会被 `.catch()` 捕获，而是直接进入全局错误处理。所以在小程序开发中，我们需要放弃前面说的，尽量抛出错误的做法，转用 `reject()` 。

```
// 不要这样做！
new Promise( resolve => {
  wx.checkSession({
    success() {
      resolve();
    },
    fail() {
      throw new Error('Weixin session expired');
    }
  });
});

// 推荐这样做
new Promise( (resolve, reject) => {
  wx.checkSession({
    success() {
      resolve();
    },
    fail() {
      reject('Weixin session expired');
    }
  });
});
```

Async Functions

“微信 Web 开发者工具”里面集成了 Babel 转译工具，可以将 ES6 编译成 ES5，不过 Async Functions 就不支持了。此时我们可以选择自行编译，或者只使用 Promise。

自行编译时，请注意，小程序页面没有 `<script>`，只能引用同名 `.js`，所以要留神输出的文件名。这里建议把 JS 写在另一个文件夹，然后用 Babel 转译，把最终文件写过来。

```
babel /path/to/my-xcx-src -d /path/to/my-xcx --source-map --watc
h
```

其它场景

现在越来越多的库与框架都开始返回 Promise 对象，这里暂列一二。

Fetch API

Fetch API 是 XMLHttpRequest 的现代化替代方案，它更强大，也更友好。它直接返回一个 Promise 实例，并分两步返回结果。

第一次返回的 `response` 包含了请求的状态，接下来可以调用 `response.json()` 生成新的 Promise 对象，它会在加载完成后返回结果。

```
fetch('some.json')
  .then( response => {
    if (response.ok) {
      return response.json();
    }
    throw new Error(response.statusText);
  })
  .then( json => {
    // do something with the json
  })
  .catch( err => {
    console.log(err);
  });
```

除了 `response.json()` 之外，它还有 `.blob()`，`.text()` 等方法，具体可以参考 MDN 上的文档。

它的浏览器覆盖率请看这里，值得注意的是，iOS 10.2 还不支持，所以现在必须提供兼容方案。

Gulp

我们知道，gulp 为了提速，默认采用并发异步的方式执行任务。如果一定要顺序执行，有三个方式：

1. 任务完成后调用参数 callback
2. 返回一个 stream 或者 vinyl file
3. 返回一个 Promise 对象

所以这个时候，应用异步函数再合适不过

```
gulp.task('task', async () => {
  let readFile = util.promisify(fs.readFile);
  let content = await readFile('sfbj.txt');
  content = content.replace('old', 'new');
  return content;
});
```

H5

H5 项目中，动画的比重很大，有些动画有顺序要求，这个时候，用 Promise 来处理就非常合适。

这里提供一个我写的函数供大家参考：

```
function isTransition(dom) {
  let style = getComputedStyle(dom).getPropertyValue('transition'
);
  return style !== 'all 0s ease 0s';
}

export function next(dom) {
  let eventName = isTransition(dom) ? 'transitionend' : 'animati
onend';
  return new Promise(resolve => {
    dom.addEventListener(eventName, function handler(event) {
      dom.removeEventListener(eventName, handler);
      resolve(event);
    }, false);
  });
}
```

这个函数会根据元素的样式判定是使用 `transition` 动画还是 `animation` 动画，然后侦听相应事件，在事件结束后， `resovle` 。

使用的时候，可以先把写好的动画样式绑上去，然后侦听，比如下面：

```
this.actions = next(loading)
  .then(() => {
    el.classList.remove('enter');
  })
  .then(() => {
    wukong.classList.add('in');
    wukong.style.transform = 'translate3d(0,0,0)';
    return next(wukong);
  })
  .then(() => {
    let bufu = this.queue.getResult('bufu');
    bufu.className = 'bufu fadeInUp animated';
    el.appendChild(bufu);
    return next(bufu);
  })
  .then(() => {
    let faxing = this.queue.getResult('faxing');
    faxing.className = 'faxing fadeInUp animated';
    el.appendChild(faxing);
    return next(faxing);
  })
  .then(() => {
    let bg = this.queue.getResult('homepage');
    bg.className = 'bg fadeIn animated';
    el.insertBefore(bg, el.firstChild);
    return next(bg);
  });
```

在这种场景，使用 Promise 会比使用异步函数更方便维护。

Vuex

Vue 从去年开始大热，越来越多同学借助其全家桶进行开发。其中，Vuex 负责状态管理，换言之，在复杂应用中，Vuex 通常作为全局的数据中心。

> 如果您还不熟悉 Vuex，可以阅读它的官方文档。

Vuex 要求开发者使用显式的方式修改数据。对立刻生效的修改，调用 `stat.commit(type, payload)` 提交；对需要异步数据交互之后才能生效的修改，通过 `stat.dispatch(type, payload)` 提交。

比如说，一个电商网站，有限量促销商品，库存很少，于是很容易发生用户下单时才发现被抢空的情况。这个时候，系统就需要帮助用户重新加载促销商品。同时，还要给出相应提示。换言之，我们不仅需要提交异步修改，还要知道异步修改是什么时候完成的。

一方面，可以通过监测特定属性，也就是借助 `vm.$watch` 来进行。不过这种方式很难区分前置条件，比如我们可以 `.$watch` 商品列表，但是商品列表有好几种原因会刷新，如果都写在一起，逻辑就很分裂。另一种方法，利用 `stat.dispatch` 会返回对应 `action` 函数的返回值的特性，可以直接返回代理异步操作的 Promise，这样我们就可以给出适当的提示了。

```
// buy.js
api.checkProduct(productId) // 检查商品是否还在
  .then( response => {
    if (response.code === 0) {
      return api.checkout(); // 正常，结账
    }
    throw new Error(response.code);
  })
  .catch( err => { // 没了
    if (err.message === NO_MORE_PRODUCT) {
      let popup = PopupManager.alert('您要购买的商品已售空，正为您查找其它的促销商品....');
      this.$store.dispatch(ActionTypes.REFETCH_SALES)
        .then(response => {
          popup.msg = '已重新获取促销商品，请尽快选购';
          popup.state = PopupManager.READY;
        });
    }
  });

// action.js
[ActionTypes.REFETCH_SALES] ({commit, state}) {
  state.isFetching = true;
  return api.fetch() // 要点：这里的 Promise 会返回给 dispatch 的地方

    .then(json => {
      commit(MutationTypes.RESET_PRODUCT_LIST, json);
    });
}
```

回顾

Pormise

相较于传统的回调模式，Promise 有着巨大的进步，值得我们学习和使用。

优势

1. 可以很好地解决异步回调不好写、不好读的问题
2. 可以使用队列，并且在对象之间传递
3. 不引入新语言元素，大部分浏览器已经原生支持，可以放心使用；个别不支持的，也有完善的解决方案

不足

1. 引入了不少新概念、新写法，学习成本不低
2. 也会有嵌套，可能看起来还很复杂
3. 没有真正解决 `return/try/catch` 的问题

Async Functions

异步函数是 ES2017 里非常有价值的新特性，可以极大的改善异步开发的环境。

除了覆盖率，几乎找不出什么黑点。

目前大部分主流浏览器都已经实现对它的支持，除了 IE 和 iOS 10.2。我们可以针对这两个系列的浏览器里进行降级，不过请注意，因为 Async Functions 引入了新的语法元素，所以必须用 Babel 转译的方式来处理。

一些小 Tips

这是我犯过的一些错误，希望成为大家前车之鉴。

- `.resolve()` `.reject()` 不会自动 `return`。

- Promise 里必须 `.resolve()` `.reject()` `throw err` 才会改变状态，`.then()` 不需要。
- `resolve()` 只会返回一个值，返回多个值请用数组或对象。

参考阅读

- MDN Promise
- MDN Promise 中文
- 阮一峰：ECMAScript 6 入门 - Promise 对象
- [翻译] We have a problem with promises
- MDN Generator
- Async Function
- await
- 让微信小程序支持 ES6 的 Promise 特性
- util.promisify() in Node.js v8
- util.promisify 官方文档

更新记录

方便大家日后跟进阅读。

2017-07-16

- 增加在 Vuex 的 action 中返回 Promise 对象，实现 `dispatch` 时处理异步的内容。

待增补内容

这里主要放日常看到，但暂时写不进去的内容。

Vue 2.4

其更新日志中写到：

- Full SSR + async component support in core: SSR now supports rendering async components used anywhere and the client also supports async components during the hydration phase. This means async components / code-splitting now just works during SSR and is no longer limited at the route level.

看看是不是合适加进去。