12 异步编程模型与异步解决方案

更新时间：2020-03-31 10:34:26

> 知识犹如人体的血液一样宝贵。——高士其

JS 异步解决方案，始终是面试中的热点与重点。

不少同学对异步知识存在一些认知误区，甚至压根没有意识到这个知识体系的存在。一些本身技术水平不错的同学，在面试前做了大量的 Promise、async/await 相关的面试题，对单个知识点的特性和命题思路了如指掌，却在面对。“谈谈你所了解的 JS 异步方案”、或者稍微隐晦一点的 “Promise 到底解决了什么痛点？这样的痛点还可以如何解决？” 这样的问题时手足无措，实在让人惋惜。

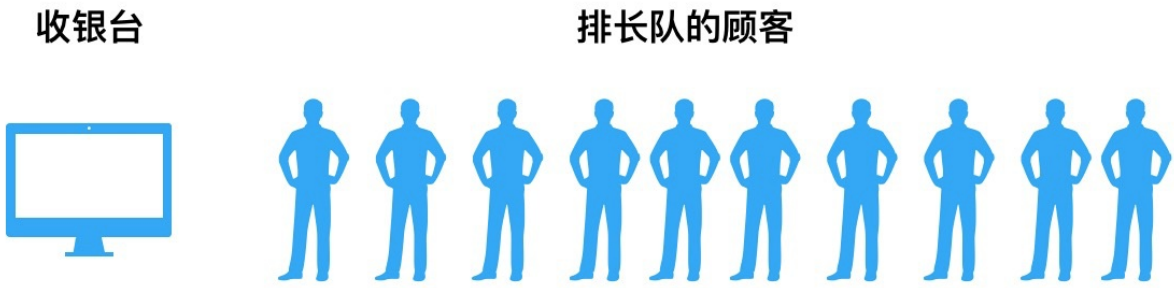
事实上，大家在各种面经里喜闻乐见的 Promise、Generator、async/await 之流，之所以如此顺利地成为面试官们的心头好，无非就是因为它们可以帮助我们优雅地解决异步。面试官固然也会想要通过考察孤立的知识点来考察你基本功的扎实度，但本质上，他真正想要的往往都是那个 “更上一层楼” 的答案 —— 想知道你对 JS 异步 以及异步解决方案，到底理解到什么程度。

异步到底是啥？为啥这么重要？为了解决异步，我们有哪些可取的手段？手段本身有哪些利弊、手段与手段之间又存在着怎样的进化关系？本节及下节，我们就通过回答这一系列的问题，来帮大家建立起一个坚不可摧的 JS 异步知识体系。

生活中的同步与异步

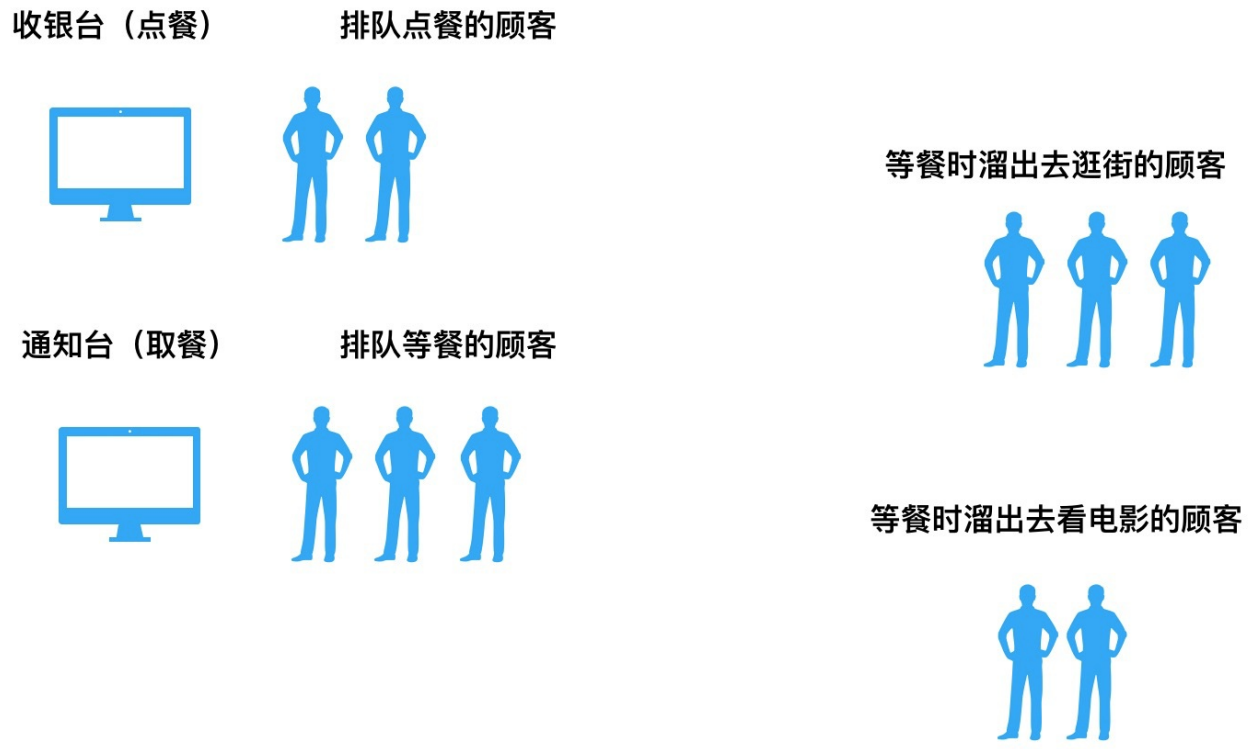
计算机领域中的同步和异步，其实和我们生活当中排队买东西很相似。

比如说咱现在去 KFC 买炸鸡。你点餐、付款可能只需要一分钟，但是等炸鸡做好需要 10 分钟。如果这时候店员跟你说，按照咱们店的规定，客人必须一直站在这儿等着，直到餐品出完为止你才能走、换下一位顾客。这种情况下，你这 10min 除了站在收银台前面和店员大眼瞪小眼外、啥也不能干；排在你后面的那些顾客，更是难上加难。



这种严格按顺序执行任务、做完一件才肯做另一件的行为方式，就是同步编程的特征。

不过要真这么玩，KFC 估计也撑不到现在。实际上我们点餐的过程中，点餐、付款的任务完成后，你大可不必原地等出餐，而是会领到一张取餐纸条。这中间你可以坐在旁边的椅子上玩手机、可以出去转转商场、可以顺手去隔壁电影院买张票 —— 你干啥都行。等出餐成功时，你取餐纸条上的号码会被公布在 KFC 大堂的屏幕上，此时凭纸条去取餐就好了。在这个过程中，点餐和出餐的过程分离到两条任务线里，点餐 1 分钟 1 分钟地点，出餐慢慢来出，只要出完之后通知到取餐人就行了。这就是异步的智慧。



这不只是 KFC 的智慧，也是 JS 的智慧。

JS 中的同步与异步

JS 语言的任务执行模式就分为同步和异步。

大家基于买炸鸡这个故事来理解 JS 中的同步和异步：同步，就是说后一个任务必须严格等待前一个任务执行完再执行，任务的执行顺序和排列顺序是高度一致的（上一个人取到炸鸡之前，下一个人不许点餐）；异步，则恰恰相反，任务的执行顺序不必遵循排列顺序。比如说前一个任务就算没执行完（炸鸡还没出餐），也没关系，先执行下一个任务就好（让下一个人先点餐），等前一个任务的执行结果啥时候出来了（炸鸡炸好了），我再把它临时穿插进来执行下（电脑屏幕上通知到取餐人）。

这其中，异步模式至关重要。大家知道，对我们前端来说，和 KFC 这样的服务行业一样，用户体验就是命。炸鸡店让客人苦等半天吃个炸鸡，你这个店要挨骂；我们页面让用户苦等 2 分钟等一个表单提交的返回结果，同样是极不友好的一种交互体验。假如我们的主线程里，充斥着用户事件、ajax 任务等高耗时的操作，这种情况下还不采用异步方案，页面的卡顿甚至卡死将是不可避免的。

异步进化史

异步在实现上，依赖一些特殊的语法规则。从整体上来说，异步方案经历了如下的四个进化阶段：

回调函数 —> Promise —> Generator —> async/await。

其中 Promise、Generator 和 async/await 都是在 ES2015 之后，慢慢发展起来的、具有一定颠覆性的新异步方案。相较于 “回调函数 “时期的刀耕火种而言，具有划时代的意义。

“回调函数” 时期

所谓 “回调函数” 时期，这里严格来说指代的其实是 Promise 出现前的这么一个相对早期的阶段。在这个阶段里，回调是异步最常见、最基本的实现手段，却不是唯一的招数 —— 像事件监听、发布订阅这样的方式，也经常为我们所用。

事件监听

这种形式相信每位前端同学都不陌生，给目标 DOM 绑定一个监听函数，我们用的最多的是 addEventListener：

```
document.getElementByld('#myDiv').addEventListener('click', function (e) {
  console.log('我被点击了')
}, false);
```

通过给 id 为 myDiv 的一个元素绑定了点击事件的监听函数，我们把任务的执行时机推迟到了点击这个动作发生时。此时，**任务的执行顺序与代码的编写顺序无关，只与点击事件有没有被触发有关**。

通过给 id 为 myDiv 的一个元素绑定了点击事件的监听函数，我们把任务的执行时机推迟到了点击这个动作发生时。此时，任务的执行顺序与代码的编写顺序无关，只与点击事件有没有被触发有关。

发布订阅

发布订阅，是一种相当经典的设计模式。大家可以跳转阅读设计模式 - 发布订阅小节。
这里我们直接用 jQuery 中封装过的发布订阅做讲解，会更容易理解一些。
比如说我们想在名为 trigger 的信号被触发后，做点事情，我们可以订阅 trigger 信号：

```
function consoleTrigger() {
   console.log('trigger事件被触发')
}
jQuery.subscribe('trigger',consoleTrigger);
```

这样当 trigger 被触发时，上面对应的回调任务就会执行了：

```
function publishTrigger() {
    jQuery.publish('trigger');
}

// 2s后，publishTrigger方法执行，trigger信号发布，consoleTrigger就会执行了
setTimeout(publishTrigger, 2000)
```

大家会发现这种模式和事件监听下的异步处理非常相似，它们都把任务执行的时机和某一事件的发生紧密关联了起来。

回调函数

回调函数用的最多的地方其实是在 Node 环境下，我们难免需要和引擎外部的环境有一些交流：比如说我要利用网络模块发起请求、或者要对外部文件进行读写等等。这些任务都是异步的，我们通过回调的形式来实现它们。

```
// -- 异步读取文件
fs.readFile(filePath,'utf8',function(err,data){
    if(err) {
      throw err;
    }
    console.log(data);// 输出文件内容
});
```

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

// 发起网络请求
https.get('目标接口', (res) => {
  console.log(data)

}).on("error", (err) => {
  console.log("Error: " + err.message);
});
```

“回调地狱”

当回调只有一层的时候，看起来感觉没什么问题。但是一旦回调函数嵌套的层级变多了之后，代码的可读性和可维护性将面临严峻的挑战。比如当我们想发起连环网络请求时：

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


https.get('目标接口1', (res) => {
  console.log(data)
  https.get('目标接口2', (res) => {
    https.get('目标接口3'),  (res) => {
       console.log(data)
       https.get('目标接口4', (res) => {
         https.get('目标接口5', (res) => {
           console.log(data)
           .....
           // 无尽的回调
         }
       }
    }
  }
})
```

这种情形一点也不夸张。而且其实不只是在 http、在 ajax 这样的网络请求场景里有这种谜之代码，在 “Promise 前” 的那个上古时期，我们经常被这种深不见底的回调困扰：

```
func1(function (resultA) {
  func2(resultA, function (resultB) {
    func3(resultB, function (resultC) {
      func4(resultC, function (resultD) {
        func5(resultD, function (resultE) {
          func6(resultE, function (resultF) {
            console.log(resultF);
            ...
            // 无尽的回调
          });
        });
      });
    });
  });
});
```

这样写代码非常糟糕，它会带来很多问题，最直接的就是：**可读性和可维护性被破坏**。

首先，你的代码会变得非常难以理解。我们这里为了大家理解方便，把每个回调的内部逻辑都写得极为简单。但是实际开发中，回调逻辑往往是有一定分量的。到时候就不是” 一行叠一行 “这么简单了，而是 “一坨叠一坨”。一眼望去，你很难看出这些回调之间到底是谁套谁。想改 A 处的代码，结果却不小心定位到了 B 处，这都是常有的事。

这时候如果你往里面再添油加醋，比如说加上 this、加上箭头函数、加上自由变量啥的，这段代码再过一个星期回来，你自己都很难看懂，更不要说后来的维护者了。

好在早期的前端世界，我们的展示层业务逻辑并没有十分复杂、Node 也还没有问世。那时，前端人普遍觉得用用事件监听、偶尔嵌套那么一两层的回调，小日子也能过得不错。但是随着逻辑的增长和复杂化、随着 Node 对大量异步操作的诉求日益强烈和明显，人们终于坐不住了，要对回调地狱这只小恶魔下手了。在这样的时代背景下，Promise 出现了。

}

11 原型与面向对象真题解析

13 全面掌握现代异步解决方案