1 小册食用指南
已学完 学习时长: 1分44秒
2 更新计划
学习时长: 29秒
3 JS 基础知识点及常考面试题（一）
学习时长: 25分49秒
4 JS 基础知识点及常考面试题（二）
学习时长: 26分4秒
5 ES6 知识点及常考面试题
已学完 学习时长: 37分8秒
6 JS 异步编程及常考面试题
已学完 学习时长: 26分57秒
7 手写 Promise
已学完 学习时长: 25分29秒
8 Event Loop
已学完 学习时长: 13分51秒
9 JS 进阶知识点及常考面试题
已学完 学习时长: 19分31秒

JS 异步编程及常考面试题

在上一章节中我们了解了常见 ES6 语法的一些知识点。这一章节我们将会学习异步编程这一块的内容，鉴于异步编程是 JS 中至关重要的内容，所以我们将会用三个章节来学习异步编程涉及到的重点和难点，同时这一块内容也是面试常考范围，希望大家认真学习。

更新日志

- 各小节内容优化

并发（concurrency）和并行（parallelism）区别

这两个名词是很多人都常会混淆的知识点。其实混淆的原因可能只是两个名词在中文上的相似，在英文上来说完全是不同的单词。

并发指的是有任务 A 和任务 B，在一段时间内通过任务间的切换完成了这两个任务，这种情况就可以称之为并发。

并行指的是假设 CPU 中存在两个核心，那么我就可以同时完成任务 A、B。同时完成多个任务的情况就可以称之为并行。

另外这两者直接拿来对比其实也是不妥的。并发指的是场景 / 需求，比如说我们这个业务有高并发的场景，但是并行指的是能力，表明我们目前的功能是可以实现这件事情的。

回调函数（Callback）

回调函数存在两大问题：

1. 信任问题
2. 可读性

信任问题指的是我们将回调交给了第三方去调用，可能会出现意料之外的事情，比如说不能保证调用次数。

可读性多指回调地狱（Callback hell）。假设多个请求存在依赖性，你可能就会写出如下代码：

```js
ajax(url, () => {
    // 处理逻辑
    ajax(url1, () => {
        // 处理逻辑
        ajax(url2, () => {
            // 处理逻辑
        })
    })
})
```

以上代码一眼看去就不利于阅读和维护。当然你可能会想说解决这个问题还不简单，把函数分开来写不就得了。

```js
function firstAjax() {
  ajax(url1, () => {
    // 处理逻辑
    secondAjax()
  })
}
function secondAjax() {
  ajax(url2, () => {
    // 处理逻辑
  })
}
ajax(url, () => {
  // 处理逻辑
  firstAjax()
})
```

以上的代码虽然看上去利于阅读了，但是还是治标不治本。

回调地狱的根本问题是：

1. 嵌套函数存在耦合性，一旦有所改动，就会牵一发而动全身
2. 嵌套函数一多，就很难处理错误

当然，回调函数还存在着别的几个缺点：

- 不能使用 `try catch` 捕获错误
- 不能直接 `return`

在接下来的几小节中，我们将来学习通过别的技术解决这些问题。

常见考点

- 什么是回调函数？
- 回调函数有什么缺点？
- 如何解决回调地狱问题？

Generator

`Generator` 算是 ES6 中难理解的概念之一了。它最大的特点就是可以控制函数的执行。在这一小节中我们不会去讲什么是 `Generator`，而是把重点放在一些容易困惑的地方。

```js
function *foo(x) {
  let y = 2 * (yield (x + 1))
  let z = yield (y / 3)
  return (x + y + z)
}
let it = foo(5)
console.log(it.next())   // => {value: 6, done: false}
console.log(it.next(12)) // => {value: 8, done: false}
console.log(it.next(13)) // => {value: 42, done: true}
```

你也许会疑惑为什么会产生与你预想不同的值，接下来让我为你逐行代码分析原因：

- 首先 `Generator` 函数调用和普通函数不同，它会返回一个迭代器
- 当执行第一次 `next` 时，传参会被忽略，并且函数暂停在 `yield (x + 1)` 处，所以返回 `5 + 1 = 6`
- 当执行第二次 `next` 时，传入的参数等于上一个 `yield` 的返回值，如果你不传参，`yield` 永远返回 `undefined`。此时 `let y = 2 * 12`，所以第二个 `yield` 等于 `2 * 12 / 3 = 8`
- 当执行第三次 `next` 时，传入的参数会传递给 `z`，所以 `z = 13, x = 5, y = 24`，相加等于 `42`

`Generator` 函数一般见到的不多，其实也于他有点绕有关系，并且一般会配合 co 库去使用。另外，我们可以通过 `Generator` 函数解决回调地狱的问题，可以把之前的回调地狱例子改写为如下代码：

```js
function *fetch() {
    yield ajax(url, () => {})
    yield ajax(url1, () => {})
    yield ajax(url2, () => {})
}
let it = fetch()
let result1 = it.next()
let result2 = it.next()
let result3 = it.next()
```

常见考点

- 你理解的 Generator 是什么？

Promise

`Promise` 翻译过来就是承诺的意思，这个承诺会在未来有一个确切的答复，并且该承诺有三种状态，分别是：

1. 等待中（pending）
2. 完成了 （resolved）
3. 拒绝了（rejected）

这个承诺一旦从等待状态变成为其他状态就永远不能更改状态了。

```javascript
new Promise((resolve, reject) => {
  resolve('success')
  // 无效
  reject('reject')
})
```

当我们在构造 `Promise` 的时候，构造函数内部的代码是立即执行的：

```js
new Promise((resolve, reject) => {
  console.log('new Promise')
  resolve('success')
})
console.log('finifsh')
// new Promise -> finifsh
```

`Promise` 实现了链式调用，也就是说每次调用 `then` 之后返回的都是一个 `Promise`，并且是一个全新的 `Promise`，原因也是因为状态不可变。如果你在 `then` 中 使用了 `return`，那么 `return` 的值会被 `Promise.resolve()` 包装

```js
Promise.resolve(1)
  .then(res => {
    console.log(res) // => 1
    return 2 // 包装成 Promise.resolve(2)
  })
  .then(res => {
    console.log(res) // => 2
  })
```

当然了，`Promise` 也很好地解决了回调地狱的问题，可以把之前的回调地狱例子改写为如下代码：

```js
ajax(url)
  .then(res => {
      console.log(res)
      return ajax(url1)
  }).then(res => {
      console.log(res)
      return ajax(url2)
  }).then(res => console.log(res))
```

最后，`Promise` 的几个 API 也经常被考到，比如说 `all`、`race`、`allSettled`。API 做题的作用这里就不讲了，大家可以查阅 MDN 文档。

常见考点

主要考手写 Promise，概念类问题不多。

- Promise 的特点是什么，分别有什么优缺点？
- 什么是 Promise 链？
- Promise 构造函数执行和 then 函数执行有什么区别？
- `all`、`race`、`allSettled` 各有什么作用？

async 及 await

一个函数如果加上 `async`，那么该函数就会返回一个 `Promise`。

```js
async function test() {
  return "1"
}
console.log(test()) // -> Promise {<resolved>: "1"}
```

`async` 就是将函数返回值使用 `Promise.resolve()` 包裹了下，和 `then` 中处理返回值一样，并且 `await` 只能配套 `async` 使用。

> 目前 `await` 可以直接脱离 `async` 在顶层调用，但是需要在 ESM 模块中。Chrome 中可以没有模块限制，但是这只是 V8 的一个特性。

```js
async function test() {
  let value = await sleep()
}
```

`async` 和 `await` 可以说是异步终极解决方案了，相比直接使用 `Promise` 来说，优势在于处理 `then` 的调用链，能够更清晰准确的写出代码，毕竟写一大堆 `then` 也很恶心，并且也能优雅地解决回调地狱问题。

当然也存在一些缺点，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。

```js
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(url)
  await fetch(url1)
  await fetch(url2)
}
```

下面再来看一个使用 `await` 的例子：

```js
let a = 0
let b = async () => {
  a = a + await 10
  console.log('2', a) // -> '2' 10
}
b()
a++
console.log('1', a) // -> '1' 1
```

对于以上代码你可能会有疑惑，让我来解释下原因：

这道题目正确答案是 10。因为加法运算法，先算左边再算右边，所以会把 0 固定下来。如果我们把题目改成 `await 10 + a` 的话，答案就是 11 了。

常见考点

- async 及 await 的特点
- 它们的优点和缺点分别是什么？

常用定时器函数

异步编程当然少不了定时器了，常见的定时器函数有 `setTimeout`、`setInterval`、`requestAnimationFrame`。

我们先来讲讲最常用的 `setTimeout`，很多人认为 `setTimeout` 是延时多久，那就应该是多久后执行。

其实这个观点是错误的，因为 JS 是单线程执行的，如果前面的代码影响了性能，就会导致 `setTimeout` 不会按期执行。当然了，我们可以通过代码去修正 `setTimeout`，从而使定时器**相对准确**。

```js
let period = 60 * 1000 * 60 * 2
let startTime = new Date().getTime()
let count = 0
let end = new Date().getTime() + period
let interval = 1000
let currentInterval = interval

function loop() {
  count++
  // 代码执行所消耗的时间
  let offset = new Date().getTime() - (startTime + count * interval);
  let diff = end - new Date().getTime()
  let h = Math.floor(diff / (60 * 1000 * 60))
  let hdiff = diff % (60 * 1000 * 60)
  let m = Math.floor(hdiff / (60 * 1000))
  let mdiff = hdiff % (60 * 1000)
  let s = mdiff / (1000)
  let sCeil = Math.ceil(s)
  let sFloor = Math.floor(s)
  // 得到下一次循环所消耗的时间
  currentInterval = interval - offset
  console.log('时: '+h, '分: '+m, '毫秒: '+s, '秒向上取整: '+sCeil, '代码执行时间: '+offset, '下次

  setTimeout(loop, currentInterval)
}

setTimeout(loop, currentInterval)
```

接下来我们来看 `setInterval`，其实这个函数作用和 `setTimeout` 基本一致，只是该函数是每隔一段时间执行一次回调函数。

通常来说不建议使用 `setInterval`。第一，它和 `setTimeout` 一样，不能保证在预期的时间执行任务。第二，回调函数执行时间不确定，可能会出现意外情况。

这里我解释下第二个缺点。比如说定时器设定每 100 毫秒执行一次，回调函数中需要执行一些耗时操作，需要 300 毫秒。

第一次回调函数正常被执行，100 毫秒之后第二个回调函数进入定时器队列等待执行，再 100 毫秒之后第三个回调函数也需要进入队列，但是此时队列中已经有相同的函数在排队了，所以此次函数不会被推入队列中等待执行，也就导致了意外情况的发生。

如果你有循环定时器的需求，其实完全可以通过 `requestAnimationFrame` 来实现

```js
function setInterval(callback, interval) {
  let timer
  const now = Date.now
  let startTime = now()
  let endTime = startTime
  const loop = () => {
    timer = window.requestAnimationFrame(loop)
    endTime = now()
    if (endTime - startTime >= interval) {
      startTime = endTime = now()
      callback(timer)
    }
  }
  timer = window.requestAnimationFrame(loop)
  return timer
}

let a = 0
setInterval(timer => {
  console.log(1)
  a++
  if (a === 3) cancelAnimationFrame(timer)
}, 1000)
```

首先 `requestAnimationFrame` 自带函数节流功能，基本可以保证在 16.6 毫秒内只执行一次（不掉帧的情况下），并且该函数的延时效果是精确的，没有其他定时器时间不准的问题，当然你也可以通过该函数来实现 `setTimeout`。

常见考点

- setTimeout、setInterval、requestAnimationFrame 区别是什么？

小结

异步编程是 JS 中较难掌握的内容，同时也是很重要的知识点。以上提到的每个知识点其实都可以作为一道面试题，希望大家可以好好掌握以上内容如果大家对于这个章节的内容存在疑问，欢迎在评论区与我互动。

留言

输入评论（Enter换行，⌘ + Enter发送）

发表评论

全部评论（109）

缘 缘 | 2月前
promise 成功完成状态不应该是fulfilled?
2 2

> **用户79804...** | 1月前
> 对，他写错了
> 点赞 回复
>
> **拉莫英俊** | 6天前
> 他的意思应该是说成功就调用 resolve
> 点赞 回复

魏log | 4月前
"但是此时队列中已经有相同的函数在排队了，所以此次函数不会被推入队列中等待执行"

为啥不会被推入队列？同一个函数不能排队执行多次吗？
1 回复

那个少年爱学习 | 9月前
回调函数指，函数B作为函数A的一个实参，并在函数A内部进行了调用，函数B就是回调函数
fun B（）{}
fun A（参数1）{}
A（B）；
1 回复

保密 | 前端开发 | 12月前
```
function *foo(x) {
  let y = 2 * (yield (x + 1))
  let z = yield (y / 3)
  return (x + y + z)
}...
展开
1 回复

**尤雨硒** | 抖腿工程师 @ 自己跳动 | 1年前
rnm退钱，概念都写的有问题
7 回复

**爱敲代码** | 2年前
无法取消 Promise 能给个具体案例吗?
1 回复

**hopekayo14108** | 2年前
Generator 👍
点赞 回复

**PercyX** | web前端开发 | 2年前
Generator 👍
点赞 回复

**你若像风** | 前端开发工程师 | 2年前
requestAnimationFrame这个api日常工作中用得多吗？秒杀活动之类的的定时器一般用什么？
点赞 2

> **xiaoxiaoo** | 2年前
> 提高动画流畅性会用它，它的执行频率在动画上看不出来，秒杀活动的倒计时定时器，一般用的是settimeinterval
> 点赞 回复
>
> **Rabbitzzc** | 2年前
> 动画中会经常用到的，不然动画很不稳定
> 1 回复

**走进科学爱学习** | 2年前
callback的信任问题比回调地域更严重吧
4 回复

**JJBoom** | 2年前
测试发现，setInterval 并没有累计的问题，是浏览器版本的问题吗？
1 回复

**蘑菇菠菜啦啦啦** | 前端 | 3年前
作为查漏补缺的小册子很好
6 1

> **你若像风** | 2年前
> 我也是这么觉得的，查缺补漏，作者有些地方讲得稍微浅的需要自己查资料，查资料也是一个加深印象的过程呀。况且有些知识点作者总结得还是很好的。我觉得这个价格买很划算了。
> 点赞 回复

**画饼师** | 3年前
"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。" 可是await本来就可以await Promise.all()啊 这怎么能算的上缺点的，太牵强了。
1 1

> **koromon** | 3年前
> Promise.all里面都是异步执行，多个 await 要等前面的执行完才下一个，还是不一样的把
> 1 回复

**小迷** | 前端 | 3年前
requestAnimationFrame这部分有点儿问题，代码我运行了一遍是错的啊
点赞 回复

**爱码士不想说话** | 3年前
requestAnimationFrame 这个不就更不准了吗...，每次都有 16ms 以内的误差；用setTimeout链式确实可以解决累加问题，但你修正了一下，累加反而变得更严重了啊，你的修正一旦超过间隔时间咋办，间隔一秒，有一次卡了三秒，用次数控制会立刻执行三次，setInterval反而只会立刻执行两次......
13 回复

**sophie旭** | 3年前
定时器文章：palmer.arkstack.cn
4 回复

**萱萱** | 软件开发工程师 | 4年前
操作系统没好好学？同一时间间隔发生的是并发，同时发生的是并行，都并行了还怎么单核？作者说的没错啊
6 3

> **屁儿快跑** | 4年前
> 1.你的'同一时间间隔发生的是并发' ,你的意思是同时开始,同时结束么? 在java中多线程编程中,线程并发可不是同时开始,同时结束,实际上是一个线程还未结束,就开始另外一个线程.所以说如果你说的是 同一时间间隔是不恰当的,同一时刻有多个才合适.
> 2.关于你的'都并行了还怎么单核? ' ,照你的意思,在单核计算机时代没有并行这个概念了? 所以说那看你怎么理解并行? 我理解的并行就是在用户看来是一起执行的就是,可以在cpu时间分片或者多核都可以实现.如果你一定要强调同时发生,那么我要承认只有多核才能实现真正...
> 展开
> 点赞 回复
>
> **dendoink** | 3年前
> 哪有那么多可以杠的，我觉得萱萱和作者说得没毛病啊。
> 并发就是我从起床到上学之前刷牙，洗脸，吃了早餐，这是并发。
> 并行就是我一边吃早餐，一边刷手机。
> 就是个概念性的东西，同时开始同时结束的描述也只是为了强调这个时间段的性质，就像是刷牙洗脸吃早餐都是发生在从起床到上学之前这段时间。
> 质疑之前也先看看自己理解是否到位吧。
> 点赞 回复
>
> 查看更多回复

**_tinyant** | 前端工程师 @ tencent | 4年前
16.6 毫秒 = 1000 / 60
点赞 回复

**在胖子的路上越走...** | 4年前
关于你说await把异步代码改造成了同步代码造成的性能降低，通过和朋友交流有不同的意见，他实际上应该还是异步，只不过看上去是同步，js是单线程，改成同步代码会阻塞 js 引擎执行，async await是以写同步代码的方式去写异步代码，以前 JQuery 的Ajax 就可以设置请求为同步请求，就是请求的时候什么都干不了，除了页面上的css动画在动以外，所有js逻辑都等着请求结束才执行，这个才是真正的同步
2 7

> **yck**（作者） | 4年前
> 现在代码都执行完了，只需要请求3个无依赖接口。是通过 Promise.all 的方式好还是三个 await 的方式好？
> 点赞 回复
>
> **在胖子的路...** 回复 yck | 4年前
> 无依赖就用 Promise.all
> "现在代码都执行完了，只需要请求3个无依赖接口。是通过 Promise.all 的方式好还..."
> 点赞 回复
>
> 查看更多回复

**砖用冰西瓜** | 4年前
对于"回调地狱的根本问题"我有保留意见，通过看YDKJS，如果我没有理解错的话，回调主要有两个问题，一个是可读性的问题，另一个是信任问题。这是一个关于回调地狱的总结：juejin.im
4 回复

查看全部 109 条回复
```