

JS 异步编程及常考面试题

并发（concurrency）和并行（parallelism）区别

涉及面试题：并发与并行的区别？

异步和这小节的知识点其实并不是一个概念，但是这两个名词确实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。

并发是宏观概念，我分别有任务 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。

并行是微观概念，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。

回调函数（Callback）

涉及面试题：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？

回调函数应该是大家经常使用到的，以下代码就是一个回调函数的例子：

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

但是回调函数有一个致命的弱点，就是容易写出回调地狱（Callback hell）。假设多个请求存在依赖性，你可能就会写出如下代码：

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

```
    })  
  })  
}
```

以上代码看起来不利于阅读和维护，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

```
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，但是还是没有解决根本问题。

回调地狱的根本问题就是：

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，比如不能使用 `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。

Generator

涉及面试题：你理解的 Generator 是什么？

`Generator` 算是 ES6 中难理解的概念之一了，`Generator` 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 `Generator` 的一些容易困惑的地方。

```
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，接下来就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。当然，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：

```
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()
```

Promise

涉及面试题：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？

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，也就是说一旦状态变为 resolved 后，就不能再次改变

```
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

```
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

```
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：

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

前面都是在讲述 `Promise` 的一些优点和特点，其实它也是存在一些缺点的，比如无法取消 `Promise`，错误需要通过回调函数捕获。

async 及 await

涉及面试题：async 及 await 的特点，它们的优点和缺点分别是什么？await 原理是什么？

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

```
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

```
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。

```
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)  
}
```

下面来看一个使用 `await` 的例子：

```
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

- 首先函数 `b` 先执行，在执行到 `await 10` 之前变量 `a` 还是 0，因为 `await` 内部实现了 `generator`，`generator` 会保留堆栈中东西，所以这时候 `a = 0` 被保存了下来
- 因为 `await` 是异步操作，后来的表达式不返回 `Promise` 的话，就会包装成 `Promise.resolve(返回值)`，然后会去执行函数外的同步代码
- 同步代码执行完毕后开始执行异步代码，将保存下来的值拿出来使用，这时候 `a = 0 + 10`

上述解释中提到了 `await` 内部实现了 `generator`，其实 `await` 就是 `generator` 加上 `Promise` 的语法糖，且内部实现了自动执行 `generator`。如果你熟悉 `co` 的话，其实自己就可以实现这样的语法糖。

常用定时器函数

涉及面试题：`setTimeout`、`setInterval`、`requestAnimationFrame` 各有什么特点？

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

```
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, '下次执行时间: '+nextTime)
```

```
    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，它存在执行累积的问题，请看以下伪代码

```
function demo() {
  setInterval(function(){
    console.log(2)
  },1000)
  sleep(2000)
}
demo()
```

以上代码在浏览器环境中，如果定时器执行过程中出现了耗时操作，多个回调函数会在耗时操作结束以后同时执行，这样可能会带来性能上的问题。

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

```
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++
}, 1000)
```

```
    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`。