80% 应聘者都不及格的 JS 面试题

2017-04-10 前端大全

（点击上方公众号，可快速关注）

> 作者：王仕军
> zhuanlan.zhihu.com/p/25855075
> 如有好文章投稿，请点击 → 这里了解详情

> 写在前面，笔者在做面试官这 2 年多的时间内，面试了数百个前端工程师，惊讶的发现，超过 80% 的候选人对下面这道题的回答情况连及格都达不到。这究竟是怎样神奇的一道面试题？他考察了候选人的哪些能力？对正在读本文的你有什么启示？且听我慢慢道来

不起眼的开始

招聘前端工程师，尤其是中高级前端工程师，扎实的 JS 基础绝对是必要条件，基础不扎实的工程师在面对前端开发中的各种问题时大概率会束手无策。在考察候选人 JS 基础的时候，我经常会提供下面这段代码，然后让候选人分析它实际运行的结果：

```
for (var i = 0; i < 5; i++) {
  setTimeout(function() {
    console.log(new Date, i);
  }, 1000);
}

console.log(new Date, i);
```

这段代码很短，只有 7 行，我想，能读到这里的同学应该不需要我逐行解释这段代码在做什么吧。候选人面对这段代码时给出的结果也不尽相同，以下是典型的答案：

- A. 20% 的人会快速扫描代码，然后给出结果：0,1,2,3,4,5；
- B. 30% 的人会拿着代码逐行看，然后给出结果：5,0,1,2,3,4；
- C. 50% 的人会拿着代码仔细琢磨，然后给出结果：5,5,5,5,5,5；

只要你对 JS 中同步和异步代码的区别、变量作用域、闭包等概念有正确的理解，就知道正确答案是 C，代码的实际输出是：

```
2017-03-18T00:43:45.873Z 5
2017-03-18T00:43:46.866Z 5
2017-03-18T00:43:46.868Z 5
```

> 2017-03-18T00:43:46.868Z 5
> 2017-03-18T00:43:46.868Z 5
> 2017-03-18T00:43:46.868Z 5

接下来我会追问：如果我们约定，用箭头表示其前后的两次输出之间有 1 秒的时间间隔，而逗号表示其前后的两次输出之间的时间间隔可以忽略，代码实际运行的结果该如何描述？会有下面两种答案：

- A. 60% 的人会描述为：5 -> 5 -> 5 -> 5 -> 5，即每个 5 之间都有 1 秒的时间间隔；
- B. 40% 的人会描述为：5 -> 5,5,5,5,5，即第 1 个 5 直接输出，1 秒之后，输出 5 个 5；

这就要求候选人对 JS 中的定时器工作机制非常熟悉，循环执行过程中，几乎同时设置了 5 个定时器，一般情况下，这些定时器都会在 1 秒之后触发，而循环完的输出是立即执行的，显而易见，正确的描述是 B。

如果到这里算是及格的话，100 个人参加面试只有 20 人能及格，读到这里的同学可以仔细思考，你及格了么？

追问 1：闭包

如果这道题仅仅是考察候选人对 JS 异步代码、变量作用域的理解，局限性未免太大，接下来我会追问，如果期望代码的输出变成：5 -> 0,1,2,3,4，该怎么改造代码？熟悉闭包的同学很快能给出下面的解决办法：

```
for (var i = 0; i < 5; i++) {
  (function(j) {  //j = i
    setTimeout(function() {
      console.log(new Date, j);
    }, 1000);
  })(i);
}

console.log(new Date, i);
```

巧妙的利用 IIFE（Immediately Invoked Function Expression：声明即执行的函数表达式）来解决闭包造成的问题，确实是不错的思路，但是初学者可能并不觉得这样的代码很好懂，至少笔者初入门的时候这里琢磨了一会儿才真正理解。

有没有更符合直觉的做法？答案是有，我们只需要对循环体稍做手脚，让负责输出的那段代码能拿到每次循环的 i 值即可。该怎么做呢？利用 JS 中基本类型（Primitive Type）的参数传递是按值传递（Pass by Value）的特征，不难改造出下面的代码：

```
var output = function (i) {
    setTimeout(function() {
        console.log(new Date, i);
    }, 1000);
};

for (var i = 0; i < 5; i++) {
    output(i); // 这里传过去的 i 值被复制了
}

console.log(new Date, i);
```

能给出上述 2 种解决方案的候选人可以认为对 JS 基础的理解和运用是不错的，可以各加 10 分。当然实际面试中还有候选人给出如下的代码：

```
for (let i = 0; i < 5; i++) {
    setTimeout(function() {
        console.log(new Date, i);
    }, 1000);
}

console.log(new Date, i);
```

细心的同学会发现，这里只有个非常细微的变动，即使用 ES6 块级作用域（Block Scope）中的 let 替代了 var，但是代码在实际运行时会报错，因为最后那个输出使用的 i 在其所在的作用域中并不存在，i 只存在于循环内部。

能想到 ES6 特性的同学虽然没有答对，但是展示了自己对 ES6 的了解，可以加 5 分，继续进行下面的追问。

追问 2：ES6

有经验的前端同学读到这里可能有些不耐烦了，扯了这么多，都是他知道的内容，先别着急，挑战的难度会继续增加。

接着上文继续追问：如果期望代码的输出变成 0 -> 1 -> 2 -> 3 -> 4 -> 5，并且要求原有的代码块中的循环和两处 console.log 不变，该怎么改造代码？新的需求可以精确的描述为：代码执行时，立即输出 0，之后每隔 1 秒依次输出 1,2,3,4，循环结束后在大概第 5 秒的时候输出 5（这里使用大概，是为了避免钻牛角尖的同学陷进去，因为 JS 中的定时器触发时机有可能是不确定的，具体可参见 How Javascript Timers Work）。

看到这里，部分同学会给出下面的可行解：

```
for (var i = 0; i < 5; i++) {
  (function(j) {
    setTimeout(function() {
      console.log(new Date, j);
    }, 1000 * j)); // 这里修改 0~4 的定时器时间
  })(i);
}

setTimeout(function() { // 这里增加定时器，超时设置为 5 秒
  console.log(new Date, i);
}, 1000 * i);
```

不得不承认，这种做法虽粗暴有效，但是不算是能额外加分的方案。如果把这次的需求抽象为：在系列异步操作完成（每次循环都产生了 1 个异步操作）之后，再做其他的事情，代码该怎么组织？聪明的你是不是想起了什么？对，就是 Promise。

可能有的同学会问，不就是在控制台输出几个数字么？至于这样杀鸡用牛刀？你要知道，面试官真正想考察的是候选人是否具备某种能力和素质，因为在现代的前端开发中，处理异步的代码随处可见，熟悉和掌握异步操作的流程控制是成为合格开发者的基本功。

顺着下来，不难给出基于 Promise 的解决方案（既然 Promise 是 ES6 中的新特性，我们的新代码使用 ES6 编写是不是会更好？如果你这么写了，大概率会让面试官心生好感）：

```
const tasks = [];
for (var i = 0; i < 5; i++) {  // 这里 i 的声明不能改成 let，如果要改该怎么做?
  ((j) => {
    tasks.push(new Promise((resolve) => {
      setTimeout(() => {
        console.log(new Date, j);
        resolve(); // 这里一定要 resolve，否则代码不会按预期 work
      }, 1000 * j);  // 定时器的超时时间逐步增加
    }));
  })(i);
}

Promise.all(tasks).then(() => {
  setTimeout(() => {
    console.log(new Date, i);
```

```
    }, 1000);  // 注意这里只需要把超时设置为 1 秒
});
```

相比而言，笔者更倾向于下面这样看起来更简洁的代码，要知道编程风格也是很多面试官重点考察的点，代码阅读时的颗粒度更小，模块化更好，无疑会是加分点。

```
const tasks = []; // 这里存放异步操作的 Promise
const output = (i) => new Promise((resolve) => {
    setTimeout(() => {
        console.log(new Date, i);
        resolve();
    }, 1000 * i);
});

// 生成全部的异步操作
for (var i = 0; i < 5; i++) {
    tasks.push(output(i));
}

// 异步操作完成之后，输出最后的 i
Promise.all(tasks).then(() => {
    setTimeout(() => {
        console.log(new Date, i);
    }, 1000);
});
```

读到这里的同学，恭喜你，你下次面试遇到类似的问题，至少能拿到 80 分。

我们都知道使用 Promise 处理异步代码比回调机制让代码可读性更高，但是使用 Promise 的问题也很明显，即如果没有处理 Promise 的 reject，会导致错误被丢进黑洞，好在新版的 Chrome 和 Node 7.x 能对未处理的异常给出 Unhandled Rejection Warning，而排查这些错误还需要一些特别的技巧（浏览器、Node.js）。

追问 3：ES7

既然你都看到这里了，那就再坚持 2 分钟，接下来的内容会让你明白你的坚持是值得的。

多数面试官在决定聘用某个候选人之前还需要考察另外一项重要能力，即技术自驱力，直白的说就是候选人像有内部的马达在驱动他，用漂亮的方式解决工程领域的问题，不断的跟随业务和技术变得越来越牛逼，究竟什么是牛逼？建议阅读程序人生的这篇剖析。

回到正题，既然 Promise 已经被拿下，如何使用 ES7 中的 async await 特性来让这段代码变的更简洁？你是否能够根据自己目前掌握的知识给出答案？请在这里暂停 1 分钟，思考下。

下面是笔者给出的参考代码：

```
// 模拟其他语言中的 sleep，实际上可以是任何异步操作
const sleep = (timeountMS) => new Promise((resolve) => {
    setTimeout(resolve, timeountMS);
});

(async () => {  // 声明即执行的 async 函数表达式
    for (var i = 0; i < 5; i++) {
        await sleep(1000);
        console.log(new Date, i);
    }

    await sleep(1000);
    console.log(new Date, i);
})();
```

总结

感谢你花时间读到这里，相信你收获的不仅仅是用 JS 精确控制代码输出的各种技巧，更是对于前端工程师的成长期许：扎实的语言基础、与时俱进的能力、强大技术自驱力。