常见的三个 JS 面试题

原创 前端小智 大迁世界 2019-09-25 19:30

你还要我怎样
薛之谦 - 意外

本文不是讨论最新的 JavaScript 库、常见的开发实践或任何新的 ES6 函数。相反，在讨论 JavaScript 时，面试中通常会提到三件事。我自己也被问到这些问题，我的朋友们告诉我他们也被问到这些问题。

然，这些并不是你在面试之前应该学习的唯一三件事 - 你可以通过多种方式更好地为即将到来的面试做准备 - 但面试官可能会问到下面是三个问题，来判断你对 `JavaScript` 语言的理解和 DOM 的掌握程度。

让我们开始吧！注意，我们将在下面的示例中使用原生的 JavaScript，因为面试官通常希望了解你在没有 jQuery 等库的帮助下对JavaScript 和 DOM 的理解程度。

问题 1: 事件委托代理

在构建应用程序时，有时需要将事件绑定到页面上的按钮、文本或图像，以便在用户与元素交互时执行某些操作。

如果我们以一个简单的待办事项列表为例，面试官可能会告诉你，当用户点击列表中的一个列表项时执行某些操作。他们希望你用 JavaScript 实现这个功能，假设有如下 HTML 代码:

```
<ul id="todo-app">
  <li class="item">Walk the dog</li>
  <li class="item">Pay bills</li>
  <li class="item">Make dinner</li>
  <li class="item">Code for one hour</li>
</ul>
```

你可能想要做如下操作来将事件绑定到元素:

```
document.addEventListener('DOMContentLoaded', function() {
  let app = document.getElementById('todo-app');
  let itimes = app.getElementsByClassName('item');

  for (let item of items) {
    item.addEventListener('click', function(){
      alert('you clicked on item: ' + item.innerHTML);
    })
  }
})
```

虽然这在技术上是可行的，但问题是要将事件分别绑定到每个项。这对于目前 `4` 个元素来说，没什么大问题，但是如果在待办事项列表中添加了 `10,000` 项(他们可能有很多事情要做)怎么办?然后，函数将创建 10,000 个独立的事件侦听器，并将每个事件监听器绑定到 DOM ，这样代码执行的效率非常低下。

在面试中，最好先问面试官用户可以输入的最大元素数量是多少。例如，如果它不超过 `10` ，那么上面的代码就可以很好地工作。但是如果用户可以输入的条目数量没有限制，那么你应该使用一个更高效的解决方案。

如果你的应用程序最终可能有数百个事件侦听器，那么更有效的解决方案是将一个事件侦听器实际绑定到整个容器，然后在单击它时能够访问每个列表项， 这称为 **事件委托**，它比附加单独的事件处理程序更有效。

下面是事件委托的代码:

```
document.addEventListener('DOMContentLoaded', function() {
  let app = document.getElementById('todo-app');

  app.addEventListener('click', function(e) {
    if (e.target && e.target.nodeName === 'LI') {
      let item = e.target;
      alert('you clicked on item: ' + item.innerHTML)
    }
  })
})
```

问题 2：在循环中使用闭包

闭包常常出现在面试中，以便面试官衡量你对 JS 的熟悉程度，以及你是否知道何时使用闭包。

闭包基本上是内部函数可以访问其范围之外的变量。 闭包可用于实现隐私和创建函数工厂， 闭包常见的面试题如下:

> 编写一个函数，该函数将遍历整数列表，并在延迟3秒后打印每个元素的索引。

经常不正确的写法是这样的:

```
const arr = [10, 12, 15, 21];
for (var i = 0; i < arr.length; i++) {
  setTimeout(function() {
    console.log('The index of this number is: ' + i);
  }, 3000);
}
```

如果运行上面代码， `3` 秒延迟后你会看到，实际上每次打印输出是 `4` ，而不是期望的 `0, 1, 2, 3` 。

为了正确理解为什么会发生这种情况，了解为什么会在 JavaScript 中发生这种情况将非常有用，这正是面试官试图测试的内容。

原因是因为 `setTimeout` 函数创建了一个可以访问其外部作用域的函数（闭包），该作用域是包含索引 `i` 的循环。 经过 `3` 秒后，执行该函数并打印出 `i` 的值，该值在循环结束时为 `4` ，因为它循环经过 `0,1,2,3,4` 并且循环最终停止在 `4` 。

实际上有多处方法来正确的解这道题:

```
const arr = [10, 12, 15, 21];

for (var i = 0; i < arr.length; i++) {
  setTimeout(function(i_local){
```

```
    return function () {
      console.log('The index of this number is: ' + i_local);
    }
  }(i), 3000)
}
```

```
const arr = [10, 12, 15, 21];
for (let i = 0; i < arr.length; i++) {
  setTimeout(function() {
    console.log('The index of this number is: ' + i);
  }, 3000);
}
```

问题 3：事件的节流（throttle）与防抖（debounce）

有些浏览器事件可以在短时间内快速触发多次，比如调整窗口大小或向下滚动页面。例如，监听页面窗口滚动事件，并且用户持续快速地向下滚动页面，那么滚动事件可能在 3 秒内触发数千次，这可能会导致一些严重的性能问题。

如果在面试中讨论构建应用程序，出现滚动、窗口大小调整或按下键等事件请务必提及 **防抖(Debouncing)** 和 **函数节流（Throttling）** 来提升页面速度和性能。这两兄弟的本质都是以**闭包**的形式存在。通过对事件对应的回调函数进行包裹、以自由变量的形式缓存时间信息，最后用 setTimeout 来控制事件的触发频率。

Throttle：第一个人说了算

throttle 的主要思想在于：在某段时间内，不管你触发了多少次回调，都只认第一次，并在计时结束时给予响应。

这个故事里，‘裁判’ 就是我们的节流阀， 他控制参赛者吃东西的时机， “参赛者吃东西”就是我们频繁操作事件而不断涌入的回调任务，它受 “裁判” 的控制,而计时器，就是上文提到的以自由变量形式存在的时间信息，它是 “裁判” 决定是否停止比赛的依据，最后，等待比赛结果就对应到回调函数的执行。

总结下来，所谓的“节流”，是通过在一段时间内无视后来产生的回调请求来实现的。只要 裁判宣布比赛开始，裁判就会开启计时器，在这段时间内，参赛者就尽管不断的吃，谁也无法知道最终结果。

对应到实际的交互上是一样一样的：每当用户触发了一次 scroll 事件，我们就为这个触发操作开启计时器。一段时间内，后续所有的 scroll 事件都会被当作“参赛者吃东西——它们无法触发新的 scroll 回调。直到“一段时间”到了，第一次触发的 scroll 事件对应的回调才会执行，而“一段时间内”触发的后续的 scroll 回调都会被节流阀无视掉。

现在一起实现一个 throttle：

```
// fn是我们需要包装的事件回调, interval是时间间隔的阈值
function throttle(fn, interval) {
  // last为上一次触发回调的时间
  let last = 0

  // 将throttle处理结果当作函数返回
  return function () {
      // 保留调用时的this上下文
      let context = this
```

```
        // 保留调用时传入的参数
        let args = arguments
        // 记录本次触发回调的时间
        let now = +new Date()

        // 判断上次触发的时间和本次触发的时间差是否小于时间间隔的阈值
        if (now - last >= interval) {
        // 如果时间间隔大于我们设定的时间间隔阈值，则执行回调
            last = now;
            fn.apply(context, args);
        }
    }
}

// 用throttle来包装scroll的回调
const better_scroll = throttle(() => console.log('触发了滚动事件'), 1000)

document.addEventListener('scroll', better_scroll)
```

Debounce：最后一个参赛者说了算

防抖的主要思想在于：我会等你到底。在某段时间内，不管你触发了多少次回调，我都只认最后一次。

继续大胃王比赛故事，这次换了一种比赛方式，时间不限，参赛者吃到不能吃为止，当每个参赛都吃不下的时候，后面10分钟如果没有人在吃，比赛结束，如果有人在10分钟内还能吃，则比赛继续，直到下一次10分钟内无人在吃时为止。

对比 throttle 来理解 debounce： 在 throttle 的逻辑里， ‘裁判’ 说了算，当比赛时间到时，就执行回调函数。而 debounce 认为最后一个参赛者说了算，只要还能吃的，就重新设定新的定时器。

现在一起实现一个 debounce：

```
// fn是我们需要包装的事件回调，delay是每次推迟执行的等待时间
function debounce(fn, delay) {
  // 定时器
  let timer = null

  // 将debounce处理结果当作函数返回
  return function () {
    // 保留调用时的this上下文
    let context = this
    // 保留调用时传入的参数
    let args = arguments

    // 每次事件被触发时，都去清除之前的旧定时器
    if(timer) {
        clearTimeout(timer)
    }
    // 设立新定时器
    timer = setTimeout(function () {
      fn.apply(context, args)
    }, delay)
  }
}

// 用debounce来包装scroll的回调
const better_scroll = debounce(() => console.log('触发了滚动事件'), 1000)

document.addEventListener('scroll', better_scroll)
```

用 Throttle 来优化 Debounce

debounce 的问题在于它“太有耐心了”。试想，如果用户的操作十分频繁——他每次都不等 debounce 设置的 delay 时间结束就进行下一次操作，于是每次 debounce 都为该用户重新生成定时器，回调函数被延迟了不计其数次。频繁的延迟会导致用户迟迟得不到响应，用户同样会产生“这个页面卡死了”的观感。

为了避免弄巧成拙，我们需要借力 throttle 的思想，打造一个“有底线”的 debounce——等你可以，但我有我的原则：delay 时间内，我可以为你重新生成定时器；但只要delay的时间到了，我必须要给用户一个响应。这个 throttle 与 debounce “合体”思路，已经被很多成熟的前端库应用到了它们的加强版 throttle 函数的实现中：

```
// fn是我们需要包装的事件回调, delay是时间间隔的阈值
function throttle(fn, delay) {
  // last为上一次触发回调的时间, timer是定时器
  let last = 0, timer = null
  // 将throttle处理结果当作函数返回

  return function () {
    // 保留调用时的this上下文
    let context = this
    // 保留调用时传入的参数
    let args = arguments
    // 记录本次触发回调的时间
    let now = +new Date()

    // 判断上次触发的时间和本次触发的时间差是否小于时间间隔的阈值
    if (now - last < delay) {
    // 如果时间间隔小于我们设定的时间间隔阈值，则为本次触发操作设立一个新的定时器
       clearTimeout(timer)
       timer = setTimeout(function () {
          last = now
          fn.apply(context, args)
        }, delay)
    } else {
        // 如果时间间隔超出了我们设定的时间间隔阈值，那就不等了，无论如何要反馈给用户一次响应
        last = now
        fn.apply(context, args)
    }
  }
}

// 用新的throttle包装scroll的回调
const better_scroll = throttle(() => console.log('触发了滚动事件'), 1000)

document.addEventListener('scroll', better_scroll)
```

代码部署后可能存在的BUG没法实时知道，事后为了解决这些BUG，花了大量的时间进行log 调试，这边顺便给大家推荐一个好用的BUG监控工具 Fundebug。

参考:

Throttling and Debouncing in JavaScript

The Difference Between Throttling and Debouncing

Examples of Throttling and Debouncing

Remy Sharp’s blog post on Throttling function calls

前端性能优化原理与实践

交流

我是小智，公众号「大迁世界」作者，**对前端技术保持学习爱好者。我会经常分享自己所学所看的干货**，在进阶的路上，共勉!

关注公众号，后台回复**福利**，即可看到福利，你懂的。

延伸阅读

理清JS中的深拷贝与浅拷贝

掌握JS函数中的几种参数形式（函数基础）

通过实现25个数组方法来理解及高效使用数组方法(长文,建议收藏)

面试 10

面试 · 目录

上一篇
36 个JS 面试题为你助力金九银十(面试必读)

下一篇
35 道咱们必须要清楚的 React 面试题