

## 07 this 基本指向原则解析

更新时间：2020-05-12 10:03:33



“

人生的价值，并不是用时间，而是用深度去衡量的。——列夫·托尔斯泰

”

### this 指向谁？

多数情况下，**this** 指向调用它所在方法的那个对象。

**\*\*说得更通俗点，谁调的函数，this 就归谁。\*\***当调用方法没有明确对象时，**this** 就指向全局对象。在浏览器中，指向 **window**；在 **Node** 中，指向 **Global**。（严格模式下，指向 **undefined**）

**this** 的指向是在调用时决定的，而不是在书写时决定的。这点和闭包恰恰相反。

（注：此处我们谈论的是“多数情况”，也是面试中考察权重较高的一种基本情况。故本节为“基本指向原则解析”。关于 **this** 的花式捆绑，我们在下一节会详细探讨）

区分“声明位置”与“调用位置”

**js** 是词法作用域模型，无论我是一个对象也好，一个方法也好，它的生命周期只和我们声明它的位置有关。我把它写在哪个位置，它就活在哪个位置。

我们来看这样一个例子：

```
// 声明位置
var me = {
  name: 'xiuyan',
  hello: function() {
    console.log(`你好，我是${this.name}`)
  }
}

var you = {
  name: 'xiaoming',
  hello: me.hello
}

// 调用位置
me.hello() // xiuyan
you.hello() // xiaoming
```

各位看到 **hello** 在代码中分别被 **me** 和 **you** 调用了，因此两次调用的 **this** 也就分别指向了 **me** 和 **you**，这没毛病。

我们稍微把这个例子改一下：

```
// 声明位置
var me = {
  name: 'xiuyan',
  hello: function() {
    console.log(`你好，我是${this.name}`)
  }
}

var name = 'BigBear'
var hello = me.hello

// 调用位置
me.hello() // 你好，我是xiuyan
hello() // 你好，我是BigBear
```

这里我们直接调用 **hello** 的时候，输出了全局的 **name** 变量。我们可以理解为是因为 **name** 和 **hello** 都挂在在全局对象 **window** 上，所以 **hello ()** 其实等价于 **window.hello ()**，此时 **hello** 方法内部的 **this** 自然指向 **window**，于是 **this.name** 就等价于 **window.name**。这也没毛病。

我们再改一下：

```
// 声明位置
var me = {
  name: 'xiuyan',
  hello: function() {
    console.log(`你好，我是${this.name}`)
  }
}

var you = {
  name: 'xiaoming',
  hello: function() {
    var targetFunc = me.hello
    targetFunc()
  }
}

var name = 'BigBear'

// 调用位置
you.hello()
```

上面这段代码，大家先给自己 1 分钟的时间，在脑子里面跑一下。

OK，现在我默认你心里已经有了一个自己的答案了（还没有跑完的同学不要急着往下看哈，自觉暂停一下，先有一个自己的结论再来看我们的解析，收获会更大）。

调用位置输出的结果是 **BigBear**—— 竟然不是 **xiaoming?** 的确，我们打眼看过去，直觉上肯定会认为是 **you** 这个对象在调用 **hello** 方法、进而调用 **targetFunc**，所以此时 **this** 肯定指向 **you** 对象啊！为啥会输出一个 **window** 上的 **name** 呢？

我们再复习一下我们开头那句话 ——“**this** 指向调用它所在方法的那个对象”。

回头看我们例题中的 **targetFunc** 这个方法，大家之所以第一直觉会认为它的 **this** 应该指向 **you** 这个对象，其实还是因为把“声明位置”和“调用位置”混淆了。我们看到虽然 **targetFunc** 是在 **you** 对象的 **hello** 方法里声明的，但是在调用它的时候，我们是不是没有给 **targetFunc** 指明任何一个对象作为它前缀？所以 **you** 对象的 **this** 并不会神奇地自动传入 **targetFunc** 里，js 引擎仍然会认为 **targetFunc** 是一个挂载在 **window** 上的方法，进而把 **this** 指向 **window** 对象。

在面试命题过程中，**this** 指向问题如果想往难了出，就会像楼上这样把声明位置和调用位置故意揉在一起，考验你对两者的区分能力。但只要各位能记住，“不管方法被书写在哪个位置，它的 **this** 只会跟着它的调用方走”这个核心原则，就一定不会出错。

## “秒杀”技巧 —— 特殊情境下的 **this** 指向

好消息！在三种特殊情境下，**this** 会 100% 指向 **window**：

- 立即执行函数（IIFE）
- **setTimeout** 中传入的函数
- **setInterval** 中传入的函数

也就是说大家在做 **this** 指向题的时候，第一步其实倒不该是老老实实去看 **this** 所在的函数属于哪个对象，而是应该先定位 **this** 是否出现在了以上三种类型的函数里面。如果是，那么想也不想，直接去对应 **window** 就好了～

老实做题没毛病，但太慢了，显得你不够老练。我们要的，是秒杀～

我们一个一个来看这三种情景对应的面试题一般会怎么出：

### 立即执行函数

所谓立即执行函数，就是定义后立刻调用的匿名函数（参见下面这道例题里 **hello** 方法的函数体里这种写法）。

```
var name = 'BigBear'

var me = {
  name: 'xiuyan',
  // 声明位置
  sayHello: function() {
    console.log(`你好，我是${this.name}`)
  },
  hello: function() {
    (function(cb) {
      // 调用位置
      cb()
    })(this.sayHello)
  }
}

me.hello() // 大家再猜下输出啥了？
```

经过我们楼上的提点，相信这里大家可以想都不想就说出 `me.hello` 的执行结果，没错，就是 `BigBear`，是 `window.name` 的值。

但其实，即便不考虑立即执行的匿名函数这种所谓的“特殊情况”，大家按照我们上面的指向原则来分析，结果也是一样的。立即执行函数作为一个匿名函数，在被调用的时候，我们往往就是直接调用，而不会（也无法）通过属性访问器（即 `xx.xxx`）这样的形式来给它指定一个所在对象，所以它的 `this` 是非常确定的，就是默认的全局对象 `window`。

## setTimeout 和 setInterval 中传入的函数

考虑到 `setTimeout` 和 `setInterval` 中函数的 `this` 指向机制其实是一样的，咱们这里拿 `setTimeout` 来开刀就够了：

```
var name = 'BigBear'

var me = {
  name: 'xiuyan',
  hello: function() {
    setTimeout(function() {
      console.log(`你好，我是${this.name}`)
    })
  }
}

me.hello() // 你好，我是BigBear
```

是不是觉得好神奇？我们的 `this.name` 明明看起来是在 `me.hello()` 里被调用的，结果却输出了 `window.name`。`setTimeout` 到底对函数做了什么？

其实，我们所看到的延时效果（`setTimeout`）和定时效果（`setInterval`），都是在全局作用域下实现的。无论是 `setTimeout` 还是 `setInterval` 里传入的函数，都会首先被交付到全局对象手上。因此，函数中 `this` 的值，会被自动指向 `window`。

## “危险”的严格模式

相信大家在进行我们专栏学习前，一定有不少同学听说过“严格模式下 `window` 会变成 `undefined`”这样的说法。

在有的书籍中，也会强调 `setTimeout` 传入函数中 `this` 的值在非严格模式下指向 `window` 对象，在严格模式下是 `undefined`。好像只要加了严格模式，那么 `undefined` 就是一种必然——确实如此吗？

事实上，严格模式确实在一些情况下会导致 **this** 指向 **undefined**， 但并非总是如此 —— 没错，事情并不简单，我们需要分情况来看：

## 普通函数中的 **this** 在严格模式下的表现

所谓“普通函数”，这里我们是相对于箭头函数来说的。在非严格模式下，直接调用普通函数时，正如我们开篇所说，函数中的 **this** 默认指向全局变量（**window** 或 **global**）：

```
function showThis() {  
  console.log(this)  
}  
  
showThis() // 输出 Window 对象
```

而在严格模式下，**this** 将保持它被指定的那个对象的值，所以，如果没有指定对象，**this** 就是 **undefined**：

```
'use strict'  
  
function showThis() {  
  console.log(this)  
}  
  
showThis() // undefined
```

## 全局代码中的 **this** 在严格模式下的表现

所谓全局代码中的 **this**，就是在全局作用域下执行的函数 / 代码段里的 **this**，它可以是这样的：

```
'use strict'  
  
console.log(this) // 直接在全局代码里尝试去拿 this
```

也可以是这样的：

```
'use strict'  
  
var name = 'BigBear'  
  
var me = {  
  name: 'xiuyan',  
  hello: function() {  
    // 全局作用域下实现的延时函数  
    setTimeout(function() {  
      console.log(`你好，我是${this.name}`)  
    })  
  }  
}  
  
me.hello() // 你好，我是BigBear
```

像这样处于全局代码中的 **this**，不管它是否处于严格模式下，它的 **this** 都指向 **Window**（这点要特别注意，区分度非常高，很多同学面试的时候会误以为这里也是 **undefined**）。

所以说，严格模式“危险”，它危险在哪？**undefined** 固然可怕，它往往使我们代码报错的元凶。但站在辅助大家面试的角度，见到 **'use strict'** 就立刻认为 **this** 会是 **undefined**，无疑是件更危险的事情。因此，严格模式下 **this** 的不同表现，还望大家牢记～

# 箭头函数

箭头函数中的 **this** 比较特别，它和严格模式、非严格模式啥的都没关系。它和闭包很相似，都是认“死理”——认“词法作用域”的家伙。所以说箭头函数中的 **this**，和你如何调用它无关，由你书写它的位置决定（和咱们普通函数的 **this** 规则恰恰相反～），例如：

```
var name = 'BigBear'
var me = {
  name: 'xiuyan',
  // 声明位置
  hello: () => {
    console.log(this.name)
  }
}

// 调用位置
me.hello() // BigBear
```

在这个例子里，因为 **this** 在书写的时候，它所在的作用域是全局作用域，于是这个 **this** 就和全局对象绑在了一起。

理解箭头函数的 **this** 指向规则，和理解普通函数的指向规则，需要的是两套完全相反的脑回路。当两个概念像这样剪不断理还乱、非常容易被混淆的时候，面试官就兴奋了。所以有时候箭头函数的 **this** 和普通函数的 **this** 放在一起考，那就是一道新面试题了（你看这帮出题的老头多无聊 / 摊手）：

问：以下两次函数调用的输出结果是什么？为什么？

```
var a = 1
var obj = {
  a: 2,
  func2: () => {
    console.log(this.a)
  },
  func3: function() {
    console.log(this.a)
  }
}

// func1
var func1 = () => {
  console.log(this.a)
}

// func2
var func2 = obj.func2
// func3
var func3 = obj.func3

func1()
func2()
func3()
obj.func2()
obj.func3()
```

答案是 1、1、1、1、2。

各位知道为什么吗？

（必须得知道！因为这道题考察的点，都是我们已经讲透的东西。如果你没有做对，先不要急着往下看，把这一节从头到尾再嚼一遍吧～）

## 问题来了

理解了 **this** 指向的原理之后，问题就来了 —— 这个 **this**，它太蠢太被动了，只知道跟着调用自己的对象走（箭头函数里只认词法作用域），堪称 JS 版的“有奶便是娘”。

可很多时候，我们不想让 **this** 这么没节操，我们希望它能够按照我们的想法指向我们想让它指向的那个对象。于是“如何改变 **this** 指向”这个问题，作为一道经典面试题，永远地留在了各大团队的面试题库里。在下一节，我们就会针对这个问题展开探讨。

}



06 JS 内存管理机制解析

08 改变 this 指向、深入理解  
call/apply/bind 的原理

