**this 他喵的到底是什么：理解 JavaScript 中的 this、call、apply 和 bind**

CoolRice [技术漫谈](javascript:void(0);) 7月9日

**this 他喵的到底是什么？很多人都有这个疑问**

JavaScript 中最容易被误解的一点就是 this 关键字。在这篇文章中，你将会了解四种规则，弄清楚 this 关键字指的是什么。隐式绑定、显式绑定、new 绑定和 window 绑定。在介绍这些技术时，你还将学习一些 JavaScript 其他令人困惑的部分，例如 .call、.apply、.bind和 new 关键字。

**视频**

* YouTube 视频链接：https://youtu.be/zE9iro4r918

**正文**

在深入了解 JavaScript 中的 this 关键字之前，有必要先退一步，看一下为什么 this 关键字很重要。this 允许复用函数时使用不同的上下文。换句话说，**“this” 关键字允许在调用函数或方法时决定哪个对象应该是焦点。** 之后讨论的所有东西都是基于这个理念。我们希望能够在不同的上下文或在不同的对象中复用函数或方法。

我们要关注的第一件事是如何判断 this 关键字的引用。当你试图回答这个问题时，你需要问自己的第一个也是最重要的问题是“**这个函数在哪里被调用？**”。判断 this 引用什么的**唯一** 方法就是看使用 this 关键字的这个方法在哪里被调用的。

用一个你已经十分熟悉的例子来展示这一点，比如我们有一个 greet 方法，它接受一个名字参数并显示有欢迎消息的警告框。

function greet (name) {  
  alert(`Hello, my name is ${name}`)  
}

如果我问你 greet 会具体警告什么内容，你会怎样回答？只给出函数定义是不可能知道答案的。为了知道 name 是什么，你必须看看 greet 函数的调用过程。

greet('Tyler')

判断 this 关键字引用什么也是同样的道理，你甚至可以把 this 当成一个普通的函数参数对待 — 它会随着函数调用方式的变化而变化。

现在我们知道为了判断 this 的引用必须先看函数的定义，在实际地查看函数定义时，我们设立了四条规则来查找引用，它们是

1. 隐式绑定
2. 显式绑定
3. new 绑定
4. window 绑定

**隐式绑定**

请记住，这里的目标是查看使用 this 关键字的函数定义，并判断 this 的指向。执行绑定的第一个也是最常见的规则称为 隐式绑定。80% 的情况下它会告诉你 this 关键字引用的是什么。

假如我们有一个这样的对象

const user = {  
  name: 'Tyler',  
  age: 27,  
  greet() {  
    alert(`Hello, my name is ${this.name}`)  
  }  
}

现在，如果你要调用 user 对象上的 greet 方法，你会用到点号。

user.greet()

这就把我们带到隐式绑定规则的主要关键点。为了判断 this 关键字的引用，**函数被调用时先看一看点号左侧**。如果有“点”就查看点左侧的对象，这个对象就是 this 的引用。

在上面的例子中，user 在“点号左侧”意味着 this 引用了 user 对象。所以就**好像** 在 greet 方法的内部 JavaScript 解释器把 this 变成了 user。

greet() {  
  // alert(`Hello, my name is ${this.name}`)  
  alert(`Hello, my name is ${user.name}`) // Tyler  
}

我们来看一个类似但稍微高级点的例子。现在，我们的对象不仅要拥有 name、age 和 greet 属性，还要被添加一个 mother 属性，并且此属性也拥有 name 和 greet 属性。

const user = {  
  name: 'Tyler',  
  age: 27,  
  greet() {  
    alert(`Hello, my name is ${this.name}`)  
  },  
  mother: {  
    name: 'Stacey',  
    greet() {  
      alert(`Hello, my name is ${this.name}`)  
    }  
  }  
}

现在问题变成下面的每个函数调用会警告什么？

user.greet()  
user.mother.greet()

每当判断 this 的引用时，我们都需要查看调用过程，并确认“点的左侧”是什么。第一个调用，user 在点左侧意味着 this 将引用 user。第二次调用中，mother 在点的左侧意味着 this 引用 mother。

user.greet() // Tyler  
user.mother.greet() // Stacey

如前所述，大约有 80% 的情况下在“点的左侧”都会有一个对象。这就是为什么在判断this 指向时“查看点的左侧”是你要做的第一件事。但是，如果没有点呢？这就为我们引出了下一条规则 —

**显式绑定**

如果 greet 函数不是 user 对象的函数，只是一个独立的函数。

function greet () {  
  alert(`Hello, my name is ${this.name}`)  
}  
  
const user = {  
  name: 'Tyler',  
  age: 27,  
}

我们知道为了判断 this 的引用我们首先必须查看这个函数的调用位置。现在就引出了一个问题，我们怎样能让 greet 方法调用的时候将 this 指向 user 对象？。我们不能再像之前那样简单的使用 user.greet()，因为 user 并没有 greet 方法。在 JavaScript 中，每个函数都包含了一个能让你恰好解决这个问题的方法，这个方法的名字叫做 call。

“call” 是每个函数都有的一个方法，它允许你在调用函数时为函数指定上下文。

考虑到这一点，用下面的代码可以在调用 greet 时用 user 做上下文。

greet.call(user)

再强调一遍，call 是每个函数都有的一个属性，并且传递给它的第一个参数会作为函数被调用时的上下文。换句话说，this 将会指向传递给 call 的第一个参数。

这就是第 2 条规则的基础（显示绑定），因为我们明确地（使用 .call）指定了 this 的引用。

现在让我们对 greet 方法做一点小小的改动。假如我们想传一些参数呢？不仅提示他们的名字，还要提示他们知道的语言。就像下面这样

function greet (lang1, lang2, lang3) {  
  alert(`Hello, my name is ${this.name} and I know ${lang1}, ${lang2}, and ${lang3}`)  
}

现在为了将这些参数传递给使用 .call 调用的函数，你需要在指定上下文（第一个参数）后一个一个地传入。

function greet (lang1, lang2, lang3) {  
  alert(`Hello, my name is ${this.name} and I know ${lang1}, ${lang2}, and ${lang3}`)  
}  
  
const user = {  
  name: 'Tyler',  
  age: 27,  
}  
  
const languages = ['JavaScript', 'Ruby', 'Python']  
  
greet.call(user, languages[0], languages[1], languages[2])

方法奏效，它显示了如何将参数传递给使用 .call 调用的函数。不过你可能注意到，必须一个一个传递 languages 数组的元素，这样有些恼人。如果我们可以把整个数组作为第二个参数并让 JavaScript 为我们自动展开就好了。有个好消息，这就是 .apply 干的事情。.apply 和 .call 本质相同，但不是一个一个传递参数，你可以用数组传参而且.apply 会在函数中为你自动展开。

那么现在用 .apply，我们的代码可以改为下面这个，其他一切都保持不变。

const languages = ['JavaScript', 'Ruby', 'Python']  
  
// greet.call(user, languages[0], languages[1], languages[2])  
greet.apply(user, languages)

到目前为止，我们学习了关于 .call 和 .apply 的“显式绑定”规则，用此规则调用的方法可以让你指定 this 在方法内的指向。关于这个规则的最后一个部分是 .bind。.bind 和.call 完全相同，除了不会立刻调用函数，而是返回一个能以后调用的新函数。因此，如果我们看看之前所写的代码，换用 .bind，它看起来就像这样

function greet (lang1, lang2, lang3) {  
  alert(`Hello, my name is ${this.name} and I know ${lang1}, ${lang2}, and ${lang3}`)  
}  
  
const user = {  
  name: 'Tyler',  
  age: 27,  
}  
  
const languages = ['JavaScript', 'Ruby', 'Python']  
  
const newFn = greet.bind(user, languages[0], languages[1], languages[2])  
newFn() // alerts "Hello, my name is Tyler and I know JavaScript, Ruby, and Python"

**new 绑定**

第三条判断 this 引用的规则是 new 绑定。若你不熟悉 JavaScript 中的 new 关键字，其实每当用 new 调用函数时，JavaScript 解释器都会在底层创建一个全新的对象并把这个对象当做 this。如果用 new 调用一个函数，this 会自然地引用解释器创建的新对象。

function User (name, age) {  
  /\*  
    JavaScript 会在底层创建一个新对象 `this`，它会代理不在 User 原型链上的属性。  
    如果一个函数用 new 关键字调用，this 就会指向解释器创建的新对象。  
  \*/  
  this.name = name  
  this.age = age  
}  
const me = new User('Tyler', 27)

**window 绑定**

假如我们有下面这段代码

function sayAge () {  
  console.log(`My age is ${this.age}`)  
}  
  
const user = {  
  name: 'Tyler',  
  age: 27  
}

如前所述，如果你想用 user 做上下文调用 sayAge，你可以使用 .call、.apply 或 .bind。但如果我们没有用这些方法，而是直接和平时一样直接调用 sayAge 会发生什么呢？

sayAge() // My age is undefined

不出意外，你会得到 My name is undefined，因为 this.age 是 undefined。事情开始变得神奇了。实际上这是因为点的左侧没有任何东西，我们也没有用 .call、.apply、.bind或者 new 关键字，JavaScript 会默认 this 指向 window 对象。这意味着如果我们向 window 对象添加 age 属性并再次调用 sayAge 方法，this.age 将不再是 undefined 并且变成 window 对象的 age 属性值。不相信？让我们运行这段代码

window.age = 27  
function sayAge () {  
  console.log(`My age is ${this.age}`)  
}

非常神奇，不是吗？这就是第 4 条规则为什么是 window 绑定 的原因。如果其它规则都没满足，JavaScript就会默认 this 指向 window 对象。

在 ES5 添加的 严格模式 中，JavaScript 不会默认 this 指向 window 对象，而会正确地把this 保持为 undefined。

'use strict'  
window.age = 27  
function sayAge () {  
  console.log(`My age is ${this.age}`)  
}  
sayAge() // TypeError: Cannot read property 'age' of undefined

因此，将所有规则付诸实践，每当我在函数内部看到 this 关键字时，这些就是我为了判断它的引用而采取的步骤。

1. 查看函数在哪被调用。
2. 点左侧有没有对象？如果有，它就是 “this” 的引用。如果没有，继续第 3 步。
3. 该函数是不是用 “call”、“apply” 或者 “bind” 调用的？如果是，它会显式地指明 “this” 的引用。如果不是，继续第 4 步。
4. 该函数是不是用 “new” 调用的？如果是，“this” 指向的就是 JavaScript 解释器新创建的对象。如果不是，继续第 5 步。
5. 是否在“严格模式”下？如果是，“this” 就是 undefined，如果不是，继续第 6 步。
6. JavaScript 很奇怪，“this” 会指向 “window” 对象。

* 原文地址：https://tylermcginnis.com/this-keyword-call-apply-bind-javascript
* 原文作者：Tyler McGinnis
* 本文译者：CoolRice
* 本文校对：周家未