02 this、new、bind、call、apply

更新时间：2019-06-25 15:06:16

> 不想当将军的士兵，不是好士兵。
>
> ——拿破仑

虽然标题是 `this`、`new`、`bind`、`call`、`apply`，但实际上这些都离不开 `this`，因此本文将着重讨论 `this`，在此过程中分别讲解其他知识点。

> **注意：** 本文属于基础篇，如果你已经对本文相关知识点已经很了解了，那么可以跳过本文。如果你不够了解，或者了解的还不完整，那么可以通过本文来复习一下 ~

1. this 指向的类型

刚开始学习 JavaScript 的时候，`this` 总是最能让人迷惑，下面我们一起看一下在 JavaScript 中应该如何确定 `this` 的指向。

`this` 是在函数被调用时确定的，它的指向**完全取决于函数调用的地方，而不是它被声明的地方**（除箭头函数外）。当一个函数被调用时，会创建一个执行上下文，它包含函数在哪里被调用（调用栈）、函数的调用方式、传入的参数等信息，`this` 就是这个记录的一个属性，它会在函数执行的过程中被用到。

`this` 在函数的指向有以下几种场景：

1. 作为构造函数被 `new` 调用；
2. 作为对象的方法使用；
3. 作为函数直接调用；
4. 被 `call`、`apply`、`bind` 调用；
5. 箭头函数中的 `this`；

下面我们分别来讨论一下这些场景中 `this` 的指向。

1.1 new 绑定

函数如果作为构造函数使用 `new` 调用时， `this` 绑定的是新创建的构造函数的实例。

```
function Foo() {
    console.log(this)
}

var bar = new Foo()       // 输出: Foo 实例, this 就是 bar
```

实际上使用 `new` 调用构造函数时，会依次执行下面的操作：

1. 创建一个新对象；
2. 构造函数的 `prototype` 被赋值给这个新对象的 `__proto__`；
3. 将新对象赋给当前的 `this`；
4. 执行构造函数；
5. 如果函数没有返回其他对象，那么 `new` 表达式中的函数调用会自动返回这个新对象，如果返回的不是对象将被忽略；

1.2 显式绑定

通过 `call`、`apply`、`bind` 我们可以修改函数绑定的 `this`，使其成为我们指定的对象。通过这些方法的第一个参数我们可以显式地绑定 `this`。

```
function foo(name, price) {
    this.name = name
    this.price = price
}

function Food(category, name, price) {
    foo.call(this, name, price)       // call 方式调用
    // foo.apply(this, [name, price])    // apply 方式调用
    this.category = category
}

new Food('食品', '汉堡', '5块钱')

// 浏览器中输出: {name: "汉堡", price: "5块钱", category: "食品"}
```

`call` 和 `apply` 的区别是 `call` 方法接受的是参数列表，而 `apply` 方法接受的是一个参数数组。

```
func.call(thisArg, arg1, arg2, ...)       // call 用法
func.apply(thisArg, [arg1, arg2, ...])    // apply 用法
```

而 `bind` 方法是设置 `this` 为给定的值，并返回一个新的函数，且在调用新函数时，将给定参数列表作为原函数的参数序列的前若干项。

```
func.bind(thisArg[, arg1[, arg2[, ...]]])    // bind 用法
```

举个例子：

```
var food = {
    name: '汉堡',
    price: '5块钱',
    getPrice: function(place) {
        console.log(place + this.price)
    }
}

food.getPrice('KFC ')   // 浏览器中输出: "KFC 5块钱"

var getPrice1 = food.getPrice.bind({ name: '鸡腿', price: '7块钱' }, '肯打鸡 ')
getPrice1()       // 浏览器中输出: "肯打鸡 7块钱"
```

关于 `bind` 的原理，我们可以使用 `apply` 方法自己实现一个 `bind` 看一下：

```
// ES5 方式
Function.prototype.bind = Function.prototype.bind || function() {
    var self = this
    var rest1 = Array.prototype.slice.call(arguments)
    var context = rest1.shift()
    return function() {
        var rest2 = Array.prototype.slice.call(arguments)
        return self.apply(context, rest1.concat(rest2))
    }
}

// ES6 方式
Function.prototype.bind = Function.prototype.bind || function(...rest1) {
    const self = this
    const context = rest1.shift()
    return function(...rest2) {
        return self.apply(context, [...rest1, ...rest2])
    }
}
```

ES6 方式用了一些 ES6 的知识比如 `rest` 参数、数组解构，感兴趣的话可以看看后面的文章 <基础篇：ES6 中可能遇到的知识点> 中的详细介绍。

注意： 如果你把 `null` 或 `undefined` 作为 `this` 的绑定对象传入 `call`、`apply`、`bind`，这些值在调用时会被忽略，实际应用的是默认绑定规则。

```
var a = 'hello'

function foo() {
    console.log(this.a)
}

foo.call(null)          // 浏览器中输出: "hello"
```

1.3 隐式绑定

函数是否在某个上下文对象中调用，如果是的话 `this` 绑定的是那个上下文对象。

```
var a = 'hello'

var obj = {
    a: 'world',
    foo: function() {
        console.log(this.a)
    }
}

obj.foo()       // 浏览器中输出: "world"
```

上面代码中，`foo` 方法是作为对象的属性调用的，那么此时 `foo` 方法执行时，`this` 指向 `obj` 对象。也就是说，此时 `this` 指向调用这个方法的对象，如果嵌套了多个对象，那么指向**最后一个**调用这个方法的对象：

```
var a = 'hello'

var obj = {
    a: 'world',
    b:{
        a:'China',
        foo: function() {
            console.log(this.a)
        }
    }
}

obj.b.foo()      // 浏览器中输出: "China"
```

最后一个对象是 `obj` 上的 `b`，那么此时 `foo` 方法执行时，其中的 `this` 指向的就是 `b` 对象。

1.4 默认绑定

函数独立调用，直接使用不带任何修饰的函数引用进行调用，也是上面几种绑定途径之外的方式。非严格模式下 `this` 绑定到全局对象（浏览器下是 `winodw`，node 环境是 `global`），严格模式下 `this` 绑定到 `undefined` （因为严格模式不允许 `this` 指向全局对象）。

```
var a = 'hello'

function foo() {
    var a = 'world'
    console.log(this.a)
    console.log(this)
}

foo()             // 相当于执行 window.foo()

// 浏览器中输出: "hello"
// 浏览器中输出: Window 对象
```

上面代码中，变量 `a` 被声明在全局作用域，成为全局对象 `window` 的一个同名属性。函数 `foo` 被执行时，`this` 此时指向的是全局对象，因此打印出来的 `a` 是全局对象的属性。

注意有一种情况：

```
var a = 'hello'

var obj = {
    a: 'world',
    foo: function() {
        console.log(this.a)
    }
}

var bar = obj.foo

bar()             // 浏览器中输出: "hello"
```

此时 `bar` 函数，也就是 `obj` 上的 `foo` 方法为什么又指向了全局对象呢，是因为 `bar` 方法此时是作为函数独立调用的，所以此时的场景属于默认绑定，而不是隐式绑定。这种情况和把方法作为回调函数的场景类似：

```
var a = 'hello'

var obj = {
    a: 'world',
    foo: function() {
        console.log(this.a)
    }
}

function func(fn) {
    fn()
}

func(obj.foo)              // 浏览器中输出: "hello"
```

参数传递实际上也是一种隐式的赋值，只不过这里 `obj.foo` 方法是被隐式赋值给了函数 `func` 的形参 `fn`，而之前的情景是自己赋值，两种情景实际上类似。这种场景我们遇到的比较多的是 `setTimeout` 和 `setInterval`，如果回调函数不是箭头函数，那么其中的 `this` 指向的就是全局对象.

其实我们可以把默认绑定当作是隐式绑定的特殊情况，比如上面的 `bar()`，我们可以当作是使用 `window.bar()` 的方式调用的，此时 `bar` 中的 `this` 根据隐式绑定的情景指向的就是 `window`。

2. this 绑定的优先级

`this` 存在多个使用场景，那么多个场景同时出现的时候，`this` 到底应该如何指向呢。这里存在一个优先级的概念，`this` 根据优先级来确定指向。优先级：**new 绑定 > 显示绑定 > 隐式绑定 > 默认绑定**

所以 `this` 的判断顺序：

1. **new 绑定：** 函数是否在 `new` 中调用？如果是的话 `this` 绑定的是新创建的对象；
2. **显式绑定：** 函数是否是通过 `bind`、`call`、`apply` 调用？如果是的话，`this` 绑定的是指定的对象；
3. **隐式绑定：** 函数是否在某个上下文对象中调用？如果是的话，`this` 绑定的是那个上下文对象；
4. 如果都不是的话，使用默认绑定。如果在严格模式下，就绑定到 `undefined`，否则绑定到全局对象；

3. 箭头函数中的 this

箭头函数 是根据其声明的地方来决定 `this` 的，它是 ES6 中出现的知识点，在后文 <基础篇：ES6 中可能遇到的知识点> 中会有更详细讲解。

箭头函数的 `this` 绑定是无法通过 `call`、`apply`、`bind` 被修改的，且因为箭头函数没有构造函数 `constructor`，所以也不可以使用 `new` 调用，即不能作为构造函数，否则会报错。

```
var a = 'hello'

var obj = {
    a: 'world',
    foo: () => {
        console.log(this.a)
    }
}

obj.foo()              // 浏览器中输出: "hello"
```

我们可以看看 ECMAScript 标准中对箭头函数的描述：

> 原文： An ArrowFunction does not define local bindings for arguments, super, this, or new.target. Any reference to arguments, super, this, or new.target within an ArrowFunction must resolve to a binding in a lexically enclosing environment. Typically this will be the Function Environment of an immediately enclosing function.
>
> **翻译：** 箭头函数不为 `arguments`、`super`、`this` 或 `new.target` 定义本地绑定。箭头函数中对 `arguments`、`super`、`this` 或 `new.target` 的任何引用都解析为当前所在词法作用域中的绑定。通常，这是箭头函数所在的函数作用域。
>
> — ECMAScript Language Specification - Arrow Function | ECMA 标准 - 箭头函数

4. 一个 this 的小练习

用一个小练习来实战一下：

```
var a = 20

var obj = {
    a: 40,
    foo:() => {
        console.log(this.a)

        function func() {
            this.a = 60
            console.log(this.a)
        }

        func.prototype.a = 50
        return func
    }
}

var bar = obj.foo()        // 浏览器中输出: 20
bar()                      // 浏览器中输出: 60
new bar()                  // 浏览器中输出: 60
```


稍微解释一下：

1. `var a = 20` 这句在全局变量 window 上创建了个属性 `a` 并赋值为 20；
2. 首先执行的是 `obj.foo()`，这是一个箭头函数，箭头函数不创建新的函数作用域直接沿用语句外部的作用域，因此 `obj.foo()` 执行时箭头函数中 `this` 是全局 window，首先打印出 window 上的属性 `a` 的值 20，箭头函数返回了一个原型上有个值为 50 的属性 `a` 的函数对象 `func` 给 `bar`；
3. 继续执行的是 `bar()`，这里执行的是刚刚箭头函数返回的闭包 `func`，其内部的 `this` 指向 window，因此 `this.a` 修改了 `window.a` 的值为 60 并打印出来；
4. 然后执行的是 `new bar()`，根据之前的表述，new 操作符会在 `func` 函数中创建一个继承了 `func` 原型的实例对象并用 `this` 指向它，随后 `this.a = 60` 又在实例对象上创建了一个属性 `a`，在之后的打印中已经在实例上找到了属性 `a`，因此就不继续往对象原型上查找了，所以打印出第三个 60；

如果把上面例子的箭头函数换成普通函数呢，结果会是什么样？

```
var a = 20

var obj = {
    a: 40,
    foo: function() {
        console.log(this.a)

        function func() {
            this.a = 60
            console.log(this.a)
        }

        func.prototype.a = 50
        return func
    }
}

var bar = obj.foo()        // 浏览器中输出: 40
bar()                      // 浏览器中输出: 60
new bar()                  // 浏览器中输出: 60
```

这个例子就不详细讲解了。

如果把上面两个例子弄懂原理，基本上 `this` 的指向就掌握的差不多啦~

推介阅读：

1. Function.prototype.bind() - JavaScript | MDN
2. Function.prototype.call() - JavaScript | MDN
3. Function.prototype.apply() - JavaScript | MDN
4. this - JavaScript | MDN

01 开篇 | 为什么学习本专栏？

03 闭包与高阶函数