

09 | 面向对象：通过词法作用域和调用点理解this绑定

2022-10-08 石川 来自北京



《JavaScript进阶实战课》

[课程介绍 >](#)



讲述：石川

时长 05:49 大小 5.31M



你好，我是石川。

今天，我们来讲讲 JavaScript 中的 **this**。其实讲 **this** 的资料有很多，其中不少已经把这个概念讲的很清楚了。但是为了课程的系统性，我今天也从这个单元咱们讲到的**对象和面向对象的角度**来说一说。

因为现在正好赶上国庆假期，咱们这节课的内容不是很长，所以你学起来也不会很辛苦。但是字少事大，**this** 的概念还是很重要的。所以如果你之前没有具体了解过，还是希望这节课能帮助你更好地理解 **this**。

从直观的印象来看，你可能觉得 **this** 指的是函数本身或它所在的范围，其实这样的理解都是不对。在 JavaScript 中，**this** 是在运行时而不是编写时绑定的。所以要正确地使用它，需要考虑到函数调用时的执行上下文。

默认绑定

我们来看一个简单的例子，在下面的例子中，`a` 是在全局定义的，`aLogger` 的函数是在全局被调用的，所以返回的 `this` 就是全局上下文，所以 `a` 的值自然就是 2。



复制代码

```
1 function aLogger() {
2     console.log( this.a );
3 }
4 var a = 2;
5 aLogger(); // 2
```

这种默认的绑定只在非 `strict mode` 的情况下是可以的。所以如果在 `strict mode` 下，这种默认的绑定是不可以的，则会返回 `TypeError: this is undefined`。

隐式绑定

下面，我们再来看看，如果我们在一个对象 `obj` 里给 `a` 赋值为 3，然后通过调用 `aLogger` 来获取 `a` 的值，这个时候，`aLogger` 被调用时的上下文是在 `obj` 中，所以它的值就是 3。

复制代码

```
1 function aLogger() {
2     console.log( this.a );
3 }
4
5 var obj = {
6     a: 3,
7     logger: aLogger
8 };
9
10 var a = 2;
11
12 obj.logger(); // 3
```

但是隐式绑定也有它的问题，就是当我们把对象里的方法赋值给一个全局变量时，这种绑定就消失了。比如下面的例子中，我们给 `objLogger` 赋值 `obj.logger`，结果 `this` 引用的就是全局中 `a` 的值。

复制代码

```
1 function logger() {
```

```
2     console.log( this.a );
3 }
4
5 var obj = {
6     a: 3,
7     logger: logger
8 };
9
10 var a = 2;
11
12 var objLogger = obj.logger;
13
14 objLogger(); // 2
```



显式绑定

下面，我们再来看看显式绑定。在这种情况下，我们使用的是 `call` 或者 `apply`。通过这种方式，我们可以强行使 `this` 等于 `obj`。

 复制代码

```
1 function logger() {
2     console.log( this.a );
3 }
4
5 var obj = {
6     a: 3
7 };
8
9 logger.call( obj ); // 3
```

这种显式绑定也不能完全解决问题，它也会产生一些副作用，比如在通过 `wrapper` 包装的 `new String`，`new Boolean` 或 `new Number` 的时候，这种绑定就会消失。

硬性绑定

下面，我们再来看看一种硬性绑定的方式。这里，我们使用从 `ES5` 开始支持的 `bind` 来绑定，通过这种方式，无论后续我们怎么调用 `hardBinding` 函数，`logger` 都会把 `obj` 当做 `this` 来获取它的 `a` 属性的值。

 复制代码

```
1 function logger() {
2     console.log( this.a );
```

```
3 }
4
5 var obj = {
6   a: 3
7 };
8
9 var hardBinding = logger.bind( obj );
10
11 setTimeout( hardBinding, 1000 ); // 3
12
13 hardBinding.call( window ); // 3
```



new 绑定

最后，我们再来看看 new 绑定，当我们使用 new 创建一个新的实例的时候，这个新的对象就是 this，所以我们可以看到在新的实例中我们传入的 2，就可以给 loggerA 实例的属性 a 赋值为 a，所以返回的结果是 2。

复制代码

```
1 function logger(a) {
2   this.a = a;
3   console.log( this.a );
4 }
5
6 var loggerA = new logger( 2 ); // 2
```

下面我们来看一个“硬碰硬”的较量，我们来试试用 hard binding 来对决 new binding，看看谁拥有绝对的实力。下面，我们先将 logger 里的 this 硬性绑定到 obj 1 上，这时我们输出的结果是 2。然后，我们用 new 来创建一个新的 logger 实例，在这个实例中，我们可以看到 obj 2 作为新的 logger 实例，它的 this 是可以不受 obj 1 影响的。所以 new 是强于 hard binding 的。

复制代码

```
1 function logger(a) {
2   this.a = a;
3 }
4
5 var obj1 = {};
6
7 var hardBinding = logger.bind( obj1 );
8
9 hardBinding( 2 );
10
```

```
11 console.log( obj1.a ); // 2
12
13 var obj2 = new logger( 3 );
14
15 console.log( obj1.a ); // 2
16 console.log( obj2.a ); // 3
```



之前在评论区也有朋友提到过谋智，也就是开发了火狐浏览器的公司，运营的一个 MDN 网站是一个不错的辅助了解 JavaScript 的平台。通过在 MDN 上的 `bind polyfill` 的代码，我们大概可以看到在 `bind` 中是有一个逻辑判断的，它会看新的实例是不是通过 `new` 来创建的，如果是，那么 `this` 就绑定到新的实例上。

复制代码

```
1 this instanceof fNOP &&
2 oThis ? this : oThis
3
4 // ... and:
5
6 fNOP.prototype = this.prototype;
7 fBound.prototype = new fNOP();
```

那么我们对比 `new` 和 `bind` 各有什么好处呢？用 `new` 的好处是可以帮助我们忽略 `hard binding`，同时可以预设函数的实参。用 `bind` 的好处是任何 `this` 之后的实参，都可以当做是默认的实参。这样就可以用来创建我们之前 [第 3 讲](#)说过的柯理式中的部分应用。比如在下面的例子中，`1` 和 `2` 就作为默认实参，在 `partialFunc` 中我们只要输入 `9`，就可以得到 `3` 个数字相加的结果。

复制代码

```
1 function fullFunc (x, y, z) {
2   return x + y + z;
3 }
4
5 const partialFunc = fullFunc.bind(this, 1, 2);
6 partialFunc(9); // 12
```

除了硬性绑定外，还有一个软性绑定的方式，它可以在 `global` 或 `undefined` 的情况下，将 `this` 绑定到一个默认的 `obj` 上。



```

1  if (!Function.prototype.softBind) {
2      Function.prototype.softBind = function(obj) {
3          var fn = this,
4              curried = [].slice.call( arguments, 1 ),
5              bound = function bound() {
6                  return fn.apply(
7                      (!this ||
8                          (typeof window !== "undefined" &&
9                              this === window) ||
10                             (typeof global !== "undefined" &&
11                                 this === global)
12                      ) ? obj : this,
13                      curried.concat.apply( curried, arguments )
14                  );
15              };
16          bound.prototype = Object.create( fn.prototype );
17          return bound;
18      };
19  }

```

在下面的例子当中，我们可以看到，除隐式、显式和软性绑定外，obj2 在 timeout 全局作用域下，返回的默认绑定结果。

```

1  function logger() {
2      console.log("name: " + this.name);
3  }
4  var obj1 = { name: "obj1" },
5      obj2 = { name: "obj2" },
6      obj3 = { name: "obj3" };
7
8  var logger1 = logger.softBind( obj1 );
9  logger1(); // name: obj1
10
11 obj2.logger = logger.softBind( obj1 );
12 obj2.logger(); // name: obj2
13
14 logger1.call( obj3 ); // name: obj3
15
16 setTimeout( obj2.logger, 1000 ); // name: obj1

```

同样地，这样的软性绑定也支持我们前面说的柯理式中的部分应用。

```

1  function fullFunc (x, y, z) {

```

```
2     return x + y + z;
3 }
4
5 const partialFunc = fullFunc.softBind(this, 1, 2);
6 partialFunc(9); // 12
```



延伸：箭头函数

在 `this` 的绑定中，有一点是需要注意的，那就是当我们使用箭头函数的时候，`this` 是在词法域里面的，而不是根据函数执行时的上下文。比如在下面的例子中，我们看到返回的结果就是 2 而不是 3。

 复制代码

```
1 function logger() {
2     return (a) => {
3         console.log( this.a );
4     };
5 }
6 var obj1 = {
7     a: 2
8 };
9
10 var obj2 = {
11     a: 3
12 };
13
14 var logger1 = logger.call( obj1 );
15
16 logger1.call( obj2 ); // 2
```

通过箭头函数来做 `this` 绑定的一个比较常用的场景就是 `setTimeout`。在这个函数中的 `this` 就会绑定在 `logger` 的函数词法域里。

 复制代码

```
1 function logger() {
2     setTimeout(() => {
3         console.log( this.a );
4     }, 1000);
5 }
6 var obj = {
7     a: 2
8 };
9 logger.call( obj ); // 2
```


如果我们不用箭头函数的话，也可以通过 `self = this` 这样的方式将 `this` 绑定在词法域里。



```
1 function logger() {
2     var self = this;
3     setTimeout( function(){
4         console.log( self.a );
5     }, 1000 );
6 }
7 var obj = {
8     a: 2
9 };
10 logger.call( obj ); // 2
```

但是通常为了代码的可读性和可维护性，在同一个函数中，应该一以贯之，要么尽量使用词法域，干脆不要有 `this`；或者要用 `this`，就通过 `bind` 等来绑定，而不是通过箭头函数或者 `self = this` 这样的“奇技淫巧”来做绑定。

总结


这节课我们学习了 `this` 的绑定，它可以说是和函数式中的 `closure` 有着同等重要性的概念。如果说函数式编程离不开对 `closure` 的理解，那么不理解 `this`，在 `JavaScript` 中用面向对象编程也会一头雾水。这两个概念虽然理解起来比较绕脑，但是一旦理解，你就会发现它们的无处不在。

思考题

我们今天在讲 `this` 的绑定时，用到了 `call` 和 `bind`，我们知道 `JavaScript` 中和 `call` 类似的还有 `apply`，那么你觉得在处理绑定时，它和 `call` 效果一样吗？

欢迎在留言区分享你的答案、交流学习心得或者提出问题，如果觉得有收获，也欢迎你把今天的内容分享给更多的朋友。

分享给需要的人，Ta购买本课程，你将得 18 元

 生成海报并分享



上一篇 08 | 深入理解继承、Delegation和组合

下一篇 10 | JS有哪8种数据类型，你需要注意什么？

精选留言 (1)

💬 写留言



拉莱耶的猫

2022-10-09 来自北京

隐式绑定那一节, aLogger跟代码里不一致

作者回复: 谢谢指出，这里做些修改，之后也会更新到文稿里。

```
function aLogger() {  
  console.log( this.a );  
}  
  
var obj = {  
  a: 3,  
  logger: aLogger  
};  
  
var a = 2;  
obj.logger(); // 3
```

