《2w字大章 38道面试题》彻底理清JS中this指向问题

JS每日一题 2021-12-14 00:31

前言

当一个函数调用时，会创建一个执行上下文，这个上下文包括函数调用的一些信息(调用栈，传入参数，调用方式)，`this`就指向这个执行上下文。

> this不是静态的，也并不是在编写的时候绑定的，而是在**运行时绑定**的。它的绑定和函数声明的位置没有关系，只取决于函数调用的方式。

本篇文章有点长，涉及到很多道面试题，有难有简单，如果能耐心的通读一编，我相信以后this都不成问题。

学习this之前，建议先学习以下知识:

- **JavaScript之预编译**[1]
- **JavaScript之手撕new**[2]
- **JavaScript之手撕call/apply**[3]
- **JavaScript之静态作用域与动态作用域**[4]
- **JavaScript之手撕数组高阶函数**[5]

在文章的最开始，陈列一下本篇文章涉及的内容，保证让大家不虚此行。

- 默认绑定
- 隐式绑定
- 隐式绑定丢失
- 显式绑定
- 显式绑定应用
- new绑定
- 箭头函数绑定
- 综合题
- 总结

this指向哪里

在`JavaScript`中，要想完全理解`this`，首先要理解`this`的绑定规则，`this`的绑定规则一共有5种：

1. 默认绑定
2. 隐式绑定
3. 显式(硬)绑定
4. `new`绑定
5. `ES6`新增箭头函数绑定

下面来一一介绍以下 `this` 的绑定规则。

1.默认绑定

默认绑定通常是指函数独立调用，不涉及其他绑定规则。**非严格模式下，`this` 指向 `window` ，严格模式下，`this` 指向 `undefined`** 。

题目1.1：非严格模式

```
var foo = 123;
function print(){
 this.foo = 234;
    console.log(this); // window
 console.log(foo); // 234
}
print();
```

非严格模式，`print()` 为默认绑定，`this` 指向 `window` ，所以打印 `window` 和 `234` 。

这个 `foo` 值可以说道两句：如果学习过预编译的知识，在预编译过程中， `foo` 和 `print` 函数会存放在全局 `GO` 中(即 `window` 对象上)，所以上述代码就类似下面这样：

```
window.foo = 123
function print() {
    this.foo = 234;
    console.log(this);
 console.log(window.foo);
}
window.print()
```

题目1.2：严格模式

把`题目1.1`稍作修改，看看严格模式下的执行结果。

> `"use strict"`可以开启严格模式

```
"use strict";
var foo = 123;
function print(){
    console.log('print this is ', this);
    console.log(window.foo)
    console.log(this.foo);
}
console.log('global this is ', this);
print();
```

注意事项：开启严格模式后，函数内部 `this` 指向 `undefined`，但全局对象 `window` 不会受影响

答案

```
global this is Window{...}
print this is undefined
123
Uncaught TypeError: Cannot read property 'foo' of undefined
```

题目1.3：let/const

```
let a = 1;
const b = 2;
var c = 3;
function print() {
    console.log(this.a);
    console.log(this.b);
    console.log(this.c);
}
print();
console.log(this.a);
```

`let/const` 定义的变量存在暂时性死区，而且不会挂载到 `window` 对象上，因此 `print` 中是无法获取到 `a和b` 的。

答案

```
undefined
undefined
3
undefined
```

题目1.4：对象内执行

```
a = 1;
function foo() {
    console.log(this.a);
}
const obj = {
    a: 10,
    bar() {
        foo(); // 1
    }
}
obj.bar();
```

`foo` 虽然在 `obj` 的 `bar` 函数中，但 `foo` 函数仍然是独立运行的， `foo` 中的 `this` 依旧指向 `window` 对象。

题目1.5：函数内执行

```
var a = 1
function outer () {
  var a = 2
  function inner () {
    console.log(this.a) // 1
  }
  inner()
}
outer()
```

这个题与 `题目1.4` 类似，但要注意，不要把它看成闭包问题

题目1.6：自执行函数

```
a = 1;
(function(){
    console.log(this);
    console.log(this.a)
}())
function bar() {
    b = 2;
    (function(){
        console.log(this);
```

```
        console.log(this.b)
    }())
}
bar();
```

> 默认情况下，自执行函数的`this`指向`window`

自执行函数只要执行到就会运行，并且只会运行一次，`this`指向`window`。

答案

```
Window{...}
1
Window{...}
2 // b是imply global，会挂载到window上
```

2.隐式绑定

函数的调用是在某个对象上触发的，即调用位置存在上下文对象，通俗点说就是**XXX.func()**这种调用模式。

此时 `func` 的 `this` 指向 `XXX`，但如果存在链式调用，例如 `XXX.YYY.ZZZ.func`，记住一个原则：**this永远指向最后调用它的那个对象**。

题目2.1：隐式绑定

```
var a = 1;
function foo() {
    console.log(this.a);
}
// 对象简写，等同于 {a:2, foo: foo}
var obj = {a: 2, foo}
foo();
obj.foo();
```

- `foo()`：默认绑定，打印 `1`
- `obj.foo()`：隐式绑定，打印 `2`

答案

```
1
2
```

`obj`是通过`var`定义的，`obj`会挂载到`window`之上的，`obj.foo()`就相当于`window.obj.foo()`，这也印证了**this永远指向最后调用它的那个对象**规则。

题目2.2：对象链式调用

感觉上面总是空谈链式调用的情况，下面直接来看一个例题：

```
var obj1 = {
    a: 1,
    obj2: {
        a: 2,
        foo(){
            console.log(this.a)
        }
    }
}
obj1.obj2.foo() // 2
```

3.隐式绑定的丢失

隐式绑定可是个调皮的东西，一不小心它就会发生绑定的丢失。一般会有两种常见的丢失：

- 使用另一个变量作为函数别名，之后使用别名执行函数

- 将函数作为参数传递时会被隐式赋值

隐式绑定丢失之后，`this`的指向会启用默认绑定。

具体来看题目：

题目3.1：取函数别名

```
a = 1
var obj = {
    a: 2,
    foo() {
        console.log(this.a)
    }
}
var foo = obj.foo;
obj.foo();
foo();
```

`JavaScript`对于引用类型，其地址指针存放在栈内存中，真正的本体是存放在堆内存中的。

上面将`obj.foo`赋值给`foo`，就是将`foo`也指向了`obj.foo`所指向的堆内存，此后再执行`foo`，相当于直接执行的堆内存的函数，与`obj`无关，`foo`为默认绑定。笼统的记，**只要fn前面什么都没有，肯定不是隐式绑定**。

答案

```
2
1
```

不要把这里理解成 `window.foo` 执行，如果 `foo` 为 `let/const` 定义， `foo` 不会挂载到 `window` 上，但不会影响最后的打印结果

题目3.2：取函数别名

如果取函数别名没有发生在全局，而是发生在对象之中，又会是怎样的结果呢？

```
var obj = {
    a: 1,
    foo() {
        console.log(this.a)
    }
};
var a = 2;
var foo = obj.foo;
var obj2 = { a: 3, foo: obj.foo }

obj.foo();
foo();
obj2.foo();
```

`obj2.foo` 指向了 `obj.foo` 的堆内存，此后执行与 `obj` 无关(除非使用 `call/apply` 改变 `this` 指向)

答案

```
1
2
3
```

题目3.3：函数作为参数传递

```
function foo() {
  console.log(this.a)
}
function doFoo(fn) {
  console.log(this)
  fn()
}
var obj = { a: 1, foo }
var a = 2
doFoo(obj.foo)
```

用函数预编译的知识来解答这个问题：函数预编译四部曲前两步分别是：

1. 找形参和变量声明，值赋予 `undefined`
2. 将形参与实参相统一，也就是将实参的值赋予形参。

`obj.foo` 作为实参，在预编译时将其值赋值给形参 `fn`，是将 `obj.foo` 指向的地址赋给了 `fn`，此后 `fn` 执行不会与 `obj` 产生任何关系。 `fn` 为默认绑定。

答案

```
Window {…}
2
```

题目3.4：函数作为参数传递

将上面的题略作修改， `doFoo` 不在 `window` 上执行，改为在 `obj2` 中执行

```
function foo() {
  console.log(this.a)
}
function doFoo(fn) {
  console.log(this)
  fn()
}
var obj = { a: 1, foo }
```

```
var a = 2
var obj2 = { a: 3, doFoo }
obj2.doFoo(obj.foo)
```

- `console.log(this)`：`obj2.doFoo` 符合 `xxx.fn` 格式，`doFoo` 的为隐式绑定，`this` 为 `obj2`，打印 `{a: 3, doFoo: f}`
- `fn()`：没有于 `obj2` 产生联系，默认绑定，打印2

答案

```
{a: 3, doFoo: f}
2
```

题目3.5：回调函数

下面这个题目我们写代码时会经常遇到：

```
var name='zcxiaobao';
function introduce(){
    console.log('Hello,My name is ', this.name);
}
const Tom = {
    name: 'TOM',
    introduce: function(){
```

```
        setTimeout(function(){
            console.log(this)
            console.log('Hello, My name is ',this.name);
        })
    }
}
const Mary = {
    name: 'Mary',
    introduce
}
const Lisa = {
    name: 'Lisa',
    introduce
}

Tom.introduce();
setTimeout(Mary.introduce, 100);
setTimeout(function(){
    Lisa.introduce();
},200);
```

> `setTimeout`是异步调用的，只有当满足条件并且同步代码执行完毕后，才会执行它的回调函数。

- `Tom.introduce()执行`：`console`位于`setTimeout`的回调函数中，回调函数的`this`指向`window`
- `Mary.introduce`直接作为`setTimeout`的函数参数(类似题目`题目3.3`)，会发生隐式绑定丢失，`this`为默认绑定
- `Lisa.introduce`执行虽然位于`setTimeout`的回调函数中，但保持`xxx.fn`模式，`this`为隐式绑定。

答案

```
Window {…}
Hello, My name is  zcxiaobao
Hello,My name is  zcxiaobao
Hello,My name is  Lisa
```

所以如果我们想在`setTimeout`或`setInterval`中使用外界的`this`，需要提前存储一下，避免`this`的丢失。

```
const Tom = {
    name: 'TOM',
    introduce: function(){
        _self = this
        setTimeout(function(){
            console.log('Hello, My name is ',_self.name);
        })
    }
}
Tom.introduce()
```

题目3.6：隐式绑定丢失综合题

```
name = 'javascript' ;
let obj = {
```

```
    name: 'obj',
    A (){
        this.name += 'this';
        console.log(this.name)
    },
    B(f){
        this.name += 'this';
        f();
    },
    C(){
      setTimeout(function(){
          console.log(this.name);
      },1000);
    }
}
let a = obj.A;
a();
obj.B(function(){
    console.log(this.name);
});
obj.C();
console.log(name);
```

本题目不做解析，具体可以参照上面的题目。

答案

```
javascriptthis
javascriptthis
javascriptthis
undefined
```

4.显式绑定

显式绑定比较好理解，就是通过`call()、apply()、bind()`等方法，强行改变`this`指向。

上面的方法虽然都可以改变`this`指向，但使用起来略有差别：

- `call()和apply()`函数会立即执行
- `bind()`函数会返回新函数，不会立即执行函数
- `call()和apply()`的区别在于`call`接受若干个参数，`apply`接受数组。

题目4.1：比较三种调用方式

```
function foo () {
  console.log(this.a)
}
var obj = { a: 1 }
var a = 2

foo()
foo.call(obj)
```

```
foo.apply(obj)
foo.bind(obj)
```

- `foo()`：默认绑定。
- `foo.call(obj)`：显示绑定，`foo` 的 `this` 指向 `obj`
- `foo.apply(obj)`：显式绑定
- `foo.bind(obj)`：显式绑定，但不会立即执行函数，没有返回值

答案

```
2
1
1
```

题目4.2：隐式绑定丢失

`题目3.4` 发生隐式绑定的丢失，如下代码：我们可不可以通过显式绑定来修正这个问题。

```
function foo() {
  console.log(this.a)
}
function doFoo(fn) {
  console.log(this)
```

```
  fn()
}
var obj = { a: 1, foo }
var a = 2
doFoo(obj.foo)
```

1. 首先先修正 `doFoo()` 函数的 `this` 指向。

```
doFoo.call(obj, obj.foo)
```

2. 然后修正 `fn` 的 `this` 。

```
function foo() {
  console.log(this.a)
}
function doFoo(fn) {
  console.log(this)
  fn.call(this)
}
var obj = { a: 1, foo }
var a = 2
doFoo(obj.foo)
```

大功告成。

题目4.3：回调函数与call

接着上一个题目的风格，稍微变点花样：

```
var obj1 = {
    a: 1
}
var obj2 = {
    a: 2,
    bar: function () {
        console.log(this.a)
    },
    foo: function () {
        setTimeout(function () {
            console.log(this)
            console.log(this.a)
        }.call(obj1), 0)
    }
}
var a = 3
obj2.bar()
obj2.foo()
```

乍一看上去，这个题看起来有些莫名其妙，`setTimeout`那是传了个什么东西？

做题之前，先了解一下`setTimeout`的内部机制：(关于异步的执行顺序，可以参考**JavaScript之EventLoop**[6])

```
```

```
setTimeout(fn) {
    if (回调条件满足) (
        fn
    )
}
```

这样一看，本题就清楚多了，类似`题目4.2`，修正了回调函数内`fn`的`this`指向。

答案

```
2
{a: 1}
1
```

题目4.4：注意call位置

```
function foo () {
    console.log(this.a)
}
var obj = { a: 1 }
var a = 2

foo()
foo.call(obj)
foo().call(obj)
```

- `foo()`：默认绑定
- `foo.call(obj)`：显式绑定
- `foo().call(obj)`：对 `foo()` 执行的返回值执行 `call`， `foo` 返回值为 `undefined`，执行 `call()` 会报错

答案

```
2
1
2
Uncaught TypeError: Cannot read property 'call' of undefined
```

题目4.5：注意call位置(2)

上面由于 `foo` 没有返回函数，无法执行 `call` 函数报错，因此修改一下 `foo` 函数，让它返回一个函数。

```
function foo () {
    console.log(this.a)
    return function() {
        console.log(this.a)
    }
}
var obj = { a: 1 }
var a = 2
```

```
foo()
foo.call(obj)
foo().call(obj)
```

- `foo()`：默认绑定
- `foo.call(obj)`：显式绑定
- `foo().call(obj)`：`foo()`执行，打印`2`，返回匿名函数通过`call`将`this`指向`obj`，打印`1`。

这里千万注意：最后一个`foo().call(obj)`有两个函数执行，会打印**2个值**。

答案

```
2
1
2
1
```

题目4.6：bind

将上面的`call`全部换做`bind`函数，又会怎样那？

> call是会立即执行函数，bind会返回一个新函数，但不会执行函数

```
function foo () {
    console.log(this.a)
    return function() {
        console.log(this.a)
    }
}
var obj = { a: 1 }
var a = 2

foo()
foo.bind(obj)
foo().bind(obj)
```

首先我们要先确定，最后会输出几个值？`bind` 不会执行函数，因此只有两个 `foo()` 会打印 `a` 。

- `foo()` : 默认绑定，打印 `2`
- `foo.bind(obj)` : 返回新函数，不会执行函数，无输出
- `foo().bind(obj)` : 第一层 `foo()` ，默认绑定，打印 `2` ，后 `bind` 将 `foo()` 返回的匿名函数 `this` 指向 `obj` ，不执行

答案

```
2
2
```

题目4.7：外层this与内层this

做到这里，不由产生了一些疑问：如果使用`call、bind`等修改了外层函数的`this`，那内层函数的`this`会受影响吗？(注意区别箭头函数)

```
function foo () {
    console.log(this.a)
    return function() {
        console.log(this.a)
    }
}
var obj = { a: 1 }
var a = 2
foo.call(obj)()
```

`foo.call(obj)`：第一层函数`foo`通过`call`将`this`指向`obj`，打印`1`；第二层函数为匿名函数，默认绑定，打印`2`。

答案

```
1
2
```

题目4.8：对象中的call

把上面的代码移植到对象中，看看会发生怎样的变化？

```
var obj = {
    a: 'obj',
    foo: function () {
        console.log('foo:', this.a)
        return function () {
            console.log('inner:', this.a)
        }
    }
}
var a = 'window'
var obj2 = { a: 'obj2' }

obj.foo()()
obj.foo.call(obj2)()
obj.foo().call(obj2)
```

看着这么多括号，是不是感觉有几分头大。没事，咱们来一层一层分析：

- `obj.foo()()`：第一层 `obj.foo()` 执行为隐式绑定，打印出 `foo:obj` ；第二层匿名函数为默认绑定，打印 `inner:window`
- `obj.foo.call(obj2)()`：类似 `题目4.7` ，第一层 `obj.foo.call(obj2)` 使用 `call` 将 `obj.foo` 的 `this` 指向 `obj2` ，打印 `foo: obj2` ；第二层匿名函数默认绑定，打印 `inner:window`
- `obj.foo().call(obj2)`：类似 `题目4.5` ，第一层隐式绑定，打印： `foo: obj` ，第二层匿名函数使用 `call` 将 `this` 指向 `obj2` ，打印 `inner: obj2`

题目4.9：带参数的call

显式绑定一开始讲的时候，就谈过`call/apply`存在传参差异，那咱们就来传一下参数，看看传完参数的this会是怎样的美妙。

```
var obj = {
  a: 1,
  foo: function (b) {
    b = b || this.a
    return function (c) {
      console.log(this.a + b + c)
    }
  }
}
var a = 2
var obj2 = { a: 3 }

obj.foo(a).call(obj2, 1)
obj.foo.call(obj2)(1)
```

要注意`call`执行的位置：

- `obj.foo(a).call(obj2, 1)`:
  - `obj.foo(a)`: foo的AO中b值为传入的a(形参与实参相统一)，值为2，返回匿名函数fn
  - 匿名函数`fn.call(obj2, 1)`: fn的this指向为obj2，c值为1
  - `this.a + b + c = obj2.a + FooAO.b + c = 3 + 2 + 1 = 6`

- `obj.foo.call(obj2)(1)`:
  - `obj.foo.call(obj2)`: obj.foo的this指向obj2，未传入参数，b = this.a = obj2.a = 3；返回匿名函数fn
  - 匿名函数`fn(1)`: c = 1，默认绑定，this指向window
  - `this.a + b + c = window.a + obj2.a + c = 2 + 3 + 1 = 6`

答案

```
6
6
```

麻了吗，兄弟们。进度已经快过半了，休息一会，争取把`this`一次性吃透。

5.显式绑定扩展

上面提了很多`call/apply`可以改变`this`指向，但都没有太多实用性。下面来一起学几个常用的`call与apply`使用。

题目5.1：apply求数组最值

JavaScript中没有给数组提供类似max和min函数，只提供了`Math.max/min`，用于求多个数的最值，所以可以借助apply方法，直接传递数组给`Math.max/min`

```
const arr = [1,10,11,33,4,52,17]
Math.max.apply(Math, arr)
Math.min.apply(Math, arr)
```

题目5.2：类数组转为数组

`ES6`未发布之前，没有`Array.from`方法可以将类数组转为数组，采用`Array.prototype.slice.call(arguments)`或`[].slice.call(arguments)`将类数组转化为数组。

题目5.3：数组高阶函数

日常编码中，我们会经常用到 `forEach、map` 等，但这些数组高阶方法，它们还有第二个参数 `thisArg` ，每一个回调函数都是显式绑定在 `thisArg` 上的。

例如下面这个例子

```
const obj = {a: 10}
const arr = [1, 2, 3, 4]
arr.forEach(function (val, key){
    console.log(`${key}: ${val} --- ${this.a}`)
}, obj)
```

答案

```
0: 1 --- 10
1: 2 --- 10
2: 3 --- 10
3: 4 --- 10
```

关于数组高阶函数的知识可以参考: JavaScript之手撕高阶数组函数

6.new绑定

使用 `new` 来构建函数，会执行如下四部操作：

1. 创建一个空的简单 `JavaScript` 对象（即 `{}`）；
2. 为步骤1新创建的对象添加属性 `__proto__`，将该属性链接至构造函数的原型对象；
3. 将步骤1新创建的对象作为 `this` 的上下文；
4. 如果该函数没有返回对象，则返回 `this`。

关于new更详细的知识，可以参考：**JavaScript之手撕new**[7]

通过new来调用构造函数，会生成一个新对象，并且把这个新对象绑定为调用函数的this。

题目6.1：new绑定

```
function User(name, age) {
    this.name = name;
    this.age = age;
}
var name = 'Tom';
var age = 18;
```

```
var zc = new User('zc', 24);
console.log(zc.name)
```

答案

```
zc
```

题目6.2：属性加方法

```
function User (name, age) {
  this.name = name;
  this.age = age;
  this.introduce = function () {
    console.log(this.name)
  }
  this.howOld = function () {
    return function () {
      console.log(this.age)
    }
  }
}
var name = 'Tom';
var age = 18;
var zc = new User('zc', 24)
```

```
zc.introduce()
zc.howOld()()
```

这个题很难不让人想到如下代码，都是函数嵌套，具体解法是类似的，可以对比来看一下啊。

```
const User = {
  name: 'zc';
  age: 18;
  introduce = function () {
    console.log(this.name)
  }
  howOld = function () {
    return function () {
      console.log(this.age)
    }
  }
}
var name = 'Tom';
var age = 18;
User.introduce()
User.howOld()()
```

- `zc.introduce()`: zc是new创建的实例，this指向zc，打印`zc`
- `zc.howOld()()`: zc.howOld()返回一个匿名函数，匿名函数为默认绑定，因此打印18(阿包永远18)

答案

```
zc
18
```

题目6.3：new界的天王山

`new`界的天王山，每次看懂后，没过多久就会忘掉，但这次要从根本上弄清楚该题。

接下来一起来品味品味：

```
function Foo(){
    getName = function(){ console.log(1); };
    return this;
}
Foo.getName = function(){ console.log(2); };
Foo.prototype.getName = function(){ console.log(3); };
var getName = function(){ console.log(4); };
function getName(){ console.log(5) };

Foo.getName();
getName();
Foo().getName();
getName();
new Foo.getName();
new Foo().getName();
new new Foo().getName();
```

1. 预编译

```
GO = {
    Foo: fn(Foo),
    getName: function getName(){ console.log(5) };
}
```

2. 分析后续执行

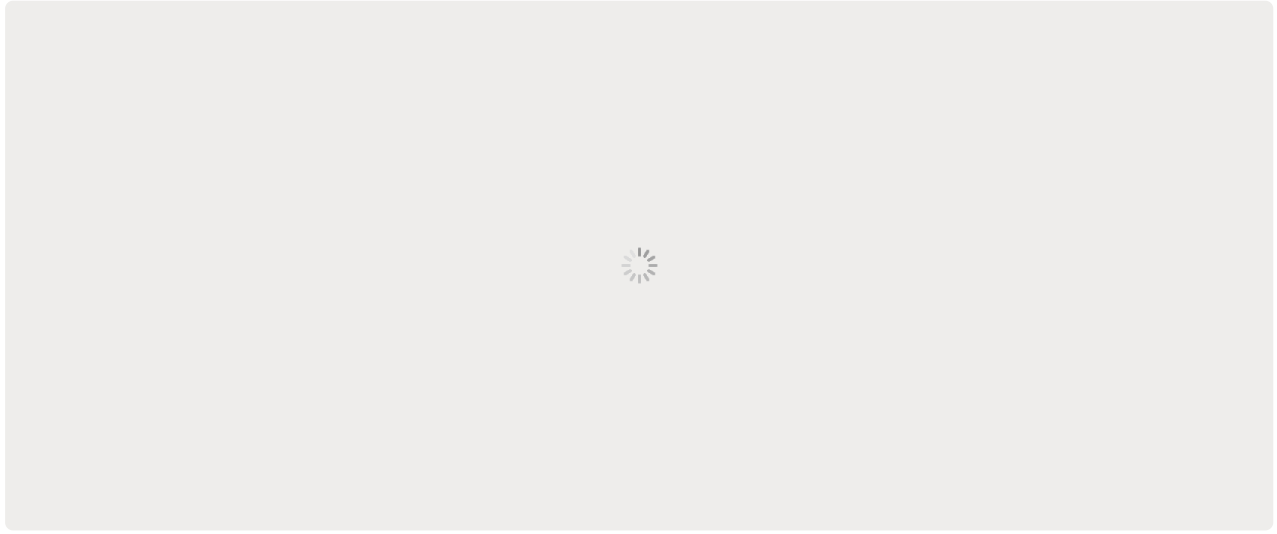
- `Foo.getName()`: 执行Foo上的getName方法，打印 `2`
- `getName()`: 执行GO中的getName方法，打印 `4`
- `Foo().getName()`

```
// 修改全局GO的getName为function(){ console.log(1); }
getName = function(){ console.log(1) }
// Foo为默认绑定，this -> window
// return window
return this
复制代码
```

  - `Foo().getName()`: 执行window.getName()，打印 `1`
  - `Foo()` 执行

- `getName()`：执行GO中的getName，打印`1`

3. 分析后面三个打印结果之前，先补充一些运算符优先级方面的知识(图源：**MDN**[8])



从上图可以看到，部分优先级如下：**new(带参数列表) = 成员访问 = 函数调用 > new(不带参数列表)**

4. `new Foo.getName()`

首先从左往右看：`new Foo`属于不带参数列表的new(优先级**19**)，`Foo.getName`属于成员访问(优先级**20**)，`getName()`属于函数调用(优先级**20**)，同样优先级遵循从左往右执行。

- `Foo.getName`执行，获取到Foo上的`getName`属性

- 此时原表达式变为 `new (Foo.getName)()` ， `new (Foo.getName)()` 为带参数列表(优先级**20**)， `(Foo.getName)()` 属于函数调用(优先级**20**)，从左往右执行
- `new (Foo.getName)()` 执行，打印 `2` ，并返回一个以 `Foo.getName()` 为构造函数的实例

> 这里有一个误区：很多人认为这里的 `new` 是没做任何操作的的，执行的是函数调用。那么如果执行的是 `Foo.getName()` ，调用返回值为 `undefined` ， `new undefined` 会发生报错，并且我们可以验证一下该表达式的返回结果。

```
console.log(new Foo.getName())
// 2
// Foo.getName {}
```

可见在成员访问之后，执行的是**带参数列表格式的new**操作。

5. `new Foo().getName()`
    - 同 `步骤4` 一样分析，先执行 `new Foo()` ，返回一个以 `Foo` 为构造函数的实例
    - `Foo` 的实例对象上没有 `getName` 方法，沿原型链查找到 `Foo.prototype.getName` 方法，打印 `3`
6. `new new Foo().getName()`

从左往右分析: 第一个new不带参数列表(优先级**19**)， `new Foo()` 带参数列表(优先级**20**)，剩下的成员访问和函数调用优先级都是**20**

- `new Foo()` 执行，返回一个以 `Foo` 为构造函数的实例

- 在执行成员访问，`Foo`实例对象在`Foo.prototype`查找到`getName`属性
- 执行`new (new Foo().getName())()`，返回一个以`Foo.prototype.getName()`为构造函数的实例，打印`3`

7. **`new Foo.getName()`与`new new Foo().getName()`区别：**

- `new Foo.getName()`的构造函数是`Foo.getName`
- `new new Foo().getName()`的构造函数为`Foo.prototype.getName`

测试结果如下:

```
foo1 = new Foo.getName()
foo2 = new new Foo().getName()
console.log(foo1.constructor)
console.log(foo2.constructor)
```

输出结果：

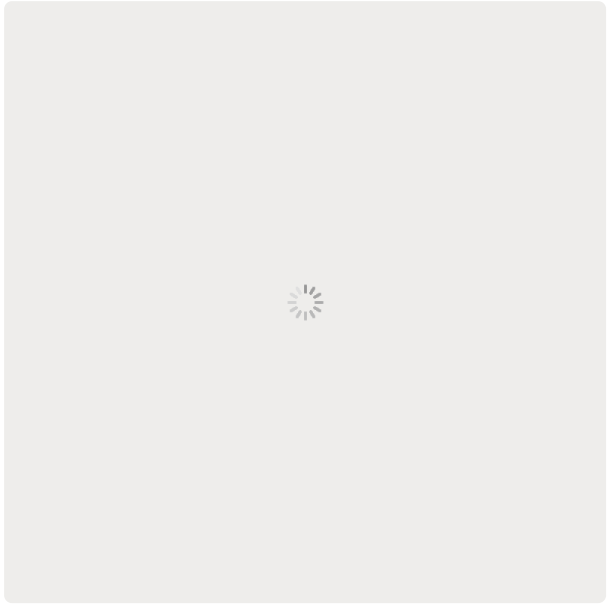
```
2
3
ƒ (){ console.log(2); }
ƒ (){ console.log(3); }
```

通过这一步比较应该能更好的理解上面的执行顺序。

答案

```
2
4
1
1
2
3
3
```

兄弟们，革命快要成功了，再努力一把，以后this都小问题啦。



7.箭头函数

箭头函数没有自己的`this`，它的`this`指向外层作用域的`this`，且指向函数定义时的`this`而非执行时。

1. `this指向外层作用域的this`：箭头函数没有`this`绑定，但它可以通过作用域链查到外层作用域的`this`
2. `指向函数定义时的this而非执行时`：`JavaScript`是静态作用域，就是函数定义之后，作用域就定死了，跟它执行时的地方无关。更详细的介绍见**JavaScript之静态作用域与动态作用域**[9]。

题目7.1：对象方法使用箭头函数

```
name = 'tom'
const obj = {
    name: 'zc',
    intro: () => {
        console.log('My name is ' + this.name)
    }
}
obj.intro()
```

上文说到，箭头函数的`this`通过作用域链查到，`intro`函数的上层作用域为`window`。

答案

```
My name is tom
```

题目7.2：箭头函数与普通函数比较

```
name = 'tom'
const obj = {
    name: 'zc',
    intro:function ()  {
        return () => {
            console.log('My name is ' + this.name)
        }
    },
    intro2:function ()  {
        return function() {
            console.log('My name is ' + this.name)
        }
    }
}
obj.intro2()()
obj.intro()()
```

- `obj.intro2()()`：不做赘述，打印 `My name is tom`
- `obj.intro()()`：`obj.intro()` 返回箭头函数，箭头函数的 `this` 取决于它的外层作用域，因此箭头函数的 `this` 指向 `obj`，打印 `My name is zc`

题目7.3：箭头函数与普通函数的嵌套

```
name = 'window'
const obj1 = {
    name: 'obj1',
    intro:function ()  {
        console.log(this.name)
        return () => {
            console.log(this.name)
        }
    }
}
const obj2 = {
    name: 'obj2',
    intro: ()=>  {
        console.log(this.name)
        return function() {
            console.log(this.name)
        }
    }
}
const obj3 = {
    name: 'obj3',
    intro: ()=> {
        console.log(this.name)
        return () => {
            console.log(this.name)
        }
    }
}
```

```
obj1.intro()()
obj2.intro()()
obj3.intro()()
```

- `obj1.intro()()`：类似`题目7.2`，打印`obj1，obj1`
- `obj2.intro()()`：`obj2.intro()`为箭头函数，`this`为外层作用域`this`，指向`window`。返回匿名函数为默认绑定。打印`window，window`
- `obj3.intro()()`：`obj3.intro()`与`obj2.intro()`相同，返回值为箭头函数，外层作用域`intro`的`this`指向`window`，打印`window，window`

答案

```
obj1
obj1
window
window
window
window
```

题目7.4：new碰上箭头函数

```
function User(name, age) {
    this.name = name;
    this.age = age;
```

```
    this.intro = function(){
        console.log('My name is ' + this.name)
    },
    this.howOld = () => {
        console.log('My age is ' + this.age)
    }
}

var name = 'Tom', age = 18;
var zc = new User('zc', 24);
zc.intro();
zc.howOld();
```

- `zc` 是 `new User` 实例，因此构造函数 `User` 的 `this` 指向 `zc`
- `zc.intro()`：打印 `My name is zc`
- `zc.howOld()`：`howOld` 为箭头函数，箭头函数**this由外层作用域决定，且指向函数定义时的this**，外层作用域为 `User`，`this` 指向 `zc`，打印 `My age is 24`

题目7.5：call碰上箭头函数

箭头函数由于没有 `this`，不能通过 `call\apply\bind` 来修改 `this` 指向，但可以通过修改外层作用域的 `this` 来达成间接修改

```
var name = 'window'
var obj1 = {
  name: 'obj1',
```

```
  intro: function () {
    console.log(this.name)
    return () => {
      console.log(this.name)
    }
  },
  intro2: () => {
    console.log(this.name)
    return function () {
      console.log(this.name)
    }
  }
}
var obj2 = {
  name: 'obj2'
}
obj1.intro.call(obj2)()
obj1.intro().call(obj2)
obj1.intro2.call(obj2)()
obj1.intro2().call(obj2)
```

- `obj1.intro.call(obj2)()`：第一层函数为普通函数，通过 `call` 修改 `this` 为 `obj2`，打印 `obj2`。第二层函数为箭头函数，它的 `this` 与外层 `this` 相同，同样打印 `obj2`。
- `obj1.intro().call(obj2)`：第一层函数打印 `obj1`，第二次函数为箭头函数，`call` 无效，它的 `this` 与外层 `this` 相同，打印 `obj1`
- `obj1.intro2.call(obj2)()`：第一层为箭头函数，`call` 无效，外层作用域为 `window`，打印 `window`；第二次为普通匿名函数，默认绑定，打印 `window`
- `obj1.intro2().call(obj2)`：与上同，打印 `window`；第二层为匿名函数，`call` 修改 `this` 为 `obj2`，打印 `obj2`

答案

```
obj2
obj2
obj1
obj1
window
window
window
obj2
```

8.箭头函数扩展

总结

- 箭头函数没有 `this`，它的 `this` 是通过作用域链查到外层作用域的 `this`，且指向函数定义时的 `this` 而非执行时。
- 不可以用作构造函数，不能使用 `new` 命令，否则会报错
- 箭头函数没有 `arguments` 对象，如果要用，使用 `rest` 参数代替
- 不可以使用 `yield` 命令，因此箭头函数不能用作 `Generator` 函数。
- 不能用 `call/apply/bind` 修改 `this` 指向，但可以通过修改外层作用域的 `this` 来间接修改。
- 箭头函数没有 `prototype` 属性。

避免使用场景

1. 箭头函数定义对象方法

```
const zc = {
    name: 'zc',
    intro: () => {
        // this -> window
        console.log(this.name)
    }
}
zc.intro() // undefined
```

2. 箭头函数不能作为构造函数

```
const User = (name, age) => {
    this.name = name;
    this.age = age;
}
// Uncaught TypeError: User is not a constructor
zc = new User('zc', 24);
```

3. 事件的回调函数

DOM中事件的回调函数中this已经封装指向了调用元素，如果使用构造函数，其this会指向window对象

```
```

```
document.getElementById('btn')
        .addEventListener('click', ()=> {
            console.log(this === window); // true
        })
```

9.综合题

学完上面的知识，是不是感觉自己已经趋于化境了，现在就一起来华山之巅一决高下吧。

题目9.1: 对象综合体

```
var name = 'window'
var user1 = {
    name: 'user1',
    foo1: function () {
        console.log(this.name)
    },
    foo2: () => console.log(this.name),
    foo3: function () {
        return function () {
            console.log(this.name)
        }
    },
    foo4: function () {
```

```
        return () => {
            console.log(this.name)
        }
    }
}
var user2 = { name: 'user2' }

user1.foo1()
user1.foo1.call(user2)

user1.foo2()
user1.foo2.call(user2)

user1.foo3()()
user1.foo3.call(user2)()
user1.foo3().call(user2)

user1.foo4()()
user1.foo4.call(user2)()
user1.foo4().call(user2)
```

这个题目并不难，就是把上面很多题做了个整合，如果上面都学会了，此题问题不大。

- `user1.foo1()、user1.foo1.call(user2)`：隐式绑定与显式绑定
- `user1.foo2()、user1.foo2.call(user2)`：箭头函数与call
- `user1.foo3()()、user1.foo3.call(user2)()、user1.foo3().call(user2)`：见题目4.8
- `user1.foo4()()、user1.foo4.call(user2)()、user1.foo4().call(user2)`：见题目7.5

答案:

```
var name = 'window'
var user1 = {
    name: 'user1',
    foo1: function () {
        console.log(this.name)
    },
    foo2: () => console.log(this.name),
    foo3: function () {
        return function () {
            console.log(this.name)
        }
    },
    foo4: function () {
        return () => {
            console.log(this.name)
        }
    }
}
var user2 = { name: 'user2' }

user1.foo1()  // user1
user1.foo1.call(user2) // user2

user1.foo2() // window
user1.foo2.call(user2) // window

user1.foo3()() // window
user1.foo3.call(user2)() // window
user1.foo3().call(user2) // user2
```

```
user1.foo4()() // user1
user1.foo4.call(user2)() // user2
user1.foo4().call(user2) // user1
```

题目9.2：隐式绑定丢失

```
var x = 10;
var foo = {
   x : 20,
   bar : function(){
       var x = 30;
       console.log(this.x)

   }
};
foo.bar();
(foo.bar)();
(foo.bar = foo.bar)();
(foo.bar, foo.bar)();
```

突然出现了一个代码很少的题目，还乍有些不习惯。

- `foo.bar()`：隐式绑定，打印`20`
- `(foo.bar)()`：上面提到过运算符优先级的知识，成员访问与函数调用优先级相同，默认从左到右，因此括号可有可无，隐式绑定，打印`20`
- `(foo.bar = foo.bar)()`：隐式绑定丢失，给`foo.bar`起别名，虽然名字没变，但是`foo.bar`上已经跟`foo`无关了，默认绑定，打印`10`

- `(foo.bar, foo.bar)()`：隐式绑定丢失，起函数别名，将逗号表达式的值(第二个foo.bar)赋值给新变量，之后执行新变量所指向的函数，默认绑定，打印`10`。

> 上面那说法有可能有几分难理解，隐式绑定有个定性条件，就是要满足`XXX.fn()`格式，如果破坏了这种格式，一般隐式绑定都会丢失。

题目9.3：arguments(推荐看)

```
var length = 10;
function fn() {
    console.log(this.length);
}

var obj = {
  length: 5,
  method: function(fn) {
    fn();
    arguments[0]();
  }
};

obj.method(fn, 1);
```

这个题要注意一下，有坑。

- `fn()`：默认绑定，打印10

- `arguments[0]()`：这种执行方式看起来就怪怪的，咱们把它展开来看看：

```
arguments: {
    0: fn,
    1: 1,
    length: 2
}
复制代码
```

```
arguments: {
    fn: fn,
    1: 1,
    length: 2
}
复制代码
```

3. 到这里大家应该就懂了，隐式绑定，`fn`函数`this`指向`arguments`，打印2

1. `arguments[0]`：这是访问对象的属性0？0不好理解，咱们把它稍微一换，方便一下理解：

1. `arguments`是一个类数组，`arguments`展开，应该是下面这样：

题目9.4：压轴题(推荐看)

```
var number = 5;
var obj = {
    number: 3,
    fn: (function () {
        var number;
        this.number *= 2;
        number = number * 2;
        number = 3;
        return function () {
            var num = this.number;
            this.number *= 2;
            console.log(num);
            number *= 3;
            console.log(number);
        }
    })()
}
var myFun = obj.fn;
myFun.call(null);
obj.fn();
console.log(window.number);
```

`fn.call(null)` 或者 `fn.call(undefined)` 都相当于 `fn()`

1. `obj.fn` 为立即执行函数: 默认绑定， `this` 指向 `window`

   我们来一句一句的分析：

此时的obj可以类似的看成以下代码(注意存在闭包):

```
obj = {
   number: 3,
   fn: function () {
        var num = this.number;
        this.number *= 2;
        console.log(num);
        number *= 3;
        console.log(number);
    }
}
复制代码
```

- `var number`：立即执行函数的 `AO` 中添加 `number` 属性，值为 `undefined`
- `this.number *= 2`：`window.number = 10`
- `number = number * 2`：立即执行函数 `AO` 中 `number` 值为 `undefined`，赋值后为 `NaN`
- `number = 3`：`AO` 中 `number` 值由 `NaN` 修改为 `3`
- 返回匿名函数，形成闭包

2. `myFun.call(null)`：相当于 `myFun()`，隐式绑定丢失，`myFun` 的 `this` 指向 `window`。

依旧一句一句的分析：

- `var num = this.number`：`this` 指向 `window`，`num = window.num = 10`
- `this.number *= 2`：`window.number = 20`
- `console.log(num)`：打印**10**
- `number *= 3`：当前 `AO` 中没有 `number` 属性，沿作用域链可在立即执行函数的 `AO` 中查到 `number` 属性，修改其值为 `9`
- `console.log(number)`：打印立即执行函数 `AO` 中的 `number`，打印**9**

3. `obj.fn()`：隐式绑定，`fn` 的 `this` 指向 `obj`

   继续一步一步的分析：

   - `var num = this.number`：`this->obj`，`num = obj.num = 3`
   - `this.number *= 2`：`obj.number *= 2 = 6`
   - `console.log(num)`：打印 `num` 值，打印**3**
   - `number *= 3`：当前 `AO` 中不存在 `number`，继续修改立即执行函数 `AO` 中的 `number`，`number *= 3 = 27`
   - `console.log(number)`：打印**27**

4. `console.log(window.number)`：打印**20**

> 这里解释一下，为什么 `myFun.call(null)` 执行时，找不到 `number` 变量，是去找立即执行函数 `AO` 中的 `number`，而不是找 `window.number`：JavaScript采用的静态作用域，当定义函数后，作用域链就已经定死。(更详细的解释文章最开始的推荐中有)

答案

```
10
9
3
27
20
```

总结

- 默认绑定: 非严格模式下 `this` 指向全局对象，严格模式下 `this` 会绑定到 `undefined`
- 隐式绑定: 满足 `XXX.fn()` 格式， `fn` 的 `this` 指向 `XXX` 。如果存在链式调用，**this永远指向最后调用它的那个对象**
- 隐式绑定丢失：起函数别名，通过别名运行；函数作为参数会造成隐式绑定丢失。
- 显示绑定: 通过 `call/apply/bind` 修改 `this` 指向
- `new` 绑定: 通过 `new` 来调用构造函数，会生成一个新对象，并且把这个新对象绑定为调用函数的 `this` 。
- 箭头函数绑定: 箭头函数没有 `this` ，它的 `this` 是通过作用域链查到外层作用域的 `this` ，且指向函数定义时的 `this` 而非执行时

后语

this到这里基本接近尾声了，松了一口气。这篇文章写了好久，找资源，修改博文，各种乱七八糟的杂事，导致迟迟写不出满意的博文。有可能天生理科男的缘故吧，怎么写感觉文章都很生硬，但好在还是顺利写完了。

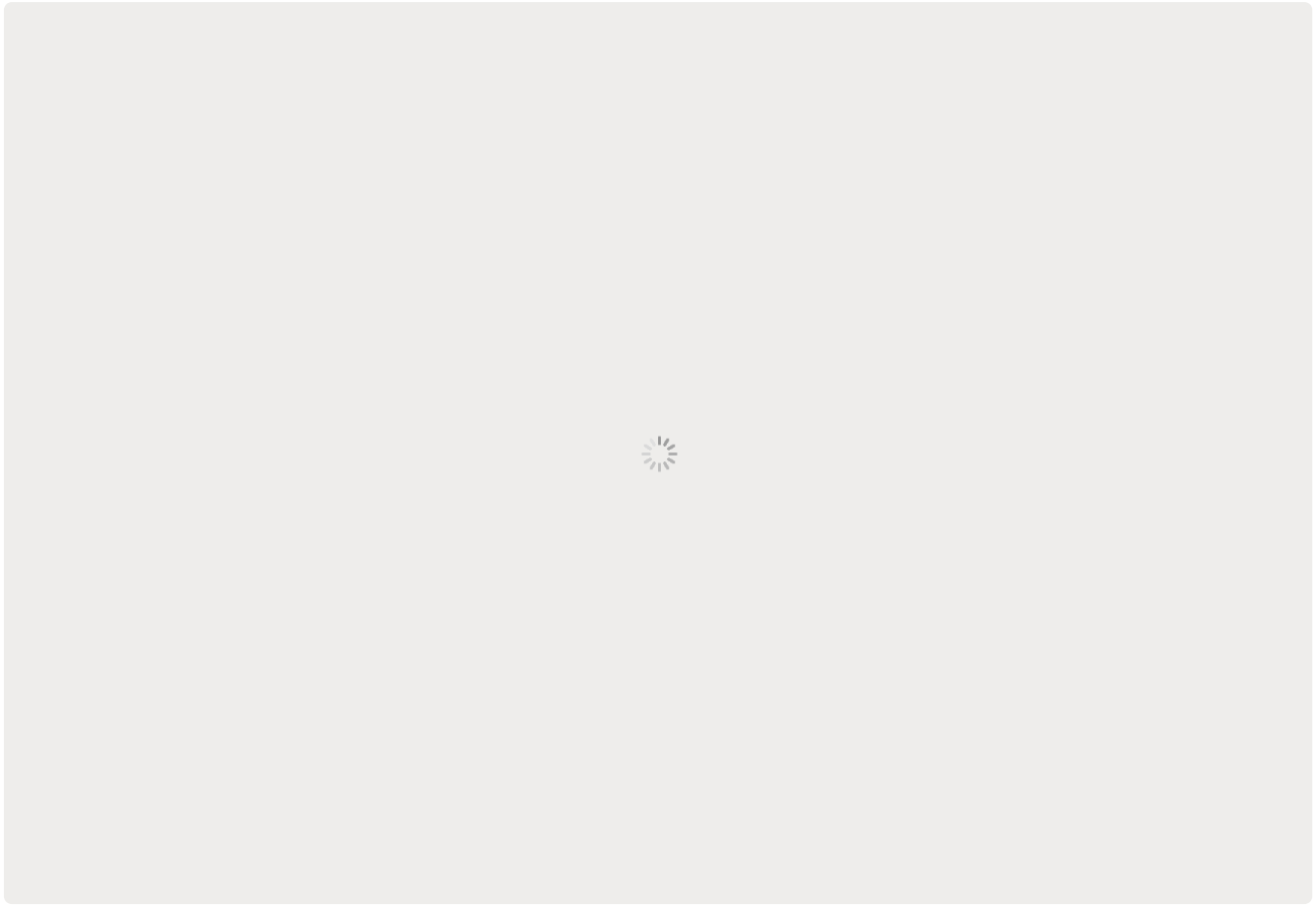
在文章的最后，感谢一下参考的博客和题目的来源

- **霖呆呆大佬**[10]
- **小夕大佬：嗨，你真的懂this吗？**[11]
- 渡一教育的题源

最后按照阿包惯例，附赠一道面试题：

```
var num = 10
var obj = {num: 20}
obj.fn = (function (num) {
  this.num = num * 3
  num++
  return function (n) {
    this.num += n
    num++
    console.log(num)
  }
})(obj.num)
var fn = obj.fn
fn(5)
obj.fn(10)
console.log(num, obj.num)
```

最后祝大家都能学好前端，步步登神，成为大佬。



> 作者：战场小包
> https://juejin.cn/post/7019470820057546766

参考资料

[1] https://juejin.cn/post/7019108835197452301

[2] https://blog.csdn.net/qq_32036091/article/details/120608863

[3] https://blog.csdn.net/qq_32036091/article/details/120589645

[4] https://blog.csdn.net/qq_32036091/article/details/120297142

[5] https://blog.csdn.net/qq_32036091/article/details/120518027

[6] https://blog.csdn.net/qq_32036091/article/details/120618424

[8] https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Operator_Precedence

[9] https://blog.csdn.net/qq_32036091/article/details/120297142

[10] https://juejin.cn/post/6844904083707396109#heading-14

[11] https://juejin.cn/post/6844903805587619854