【讲义】this指针/闭包/作用域

#2021#

this指针详解

概念

可以认为this是当前函数/当前模块的运行环境的上下文, 是一个指针型变量, 可以理解为一个动态的对象, 普通函数中的this是在调用时才被绑定确认指向的.

this的出现, 使得复用函数时可以使用不同的上下文, 也就是说通过不同的this调用同一个函数, 可以产出不同的结果。

所以就显而易见出现一个问题, 既然this是一个动态的东西, 我们应该怎么判断它到底绑定的是什么内容呢?

this的绑定规则

1. 默认绑定

指函数独立调用的时候，不带任何修饰的函数引用。

- 非严格模式下 this 指向全局对象(浏览器下指向 Window，Node.js 环境是 Global)
- 严格模式下，this 绑定到 undefined ,严格模式不允许this指向全局对象。

比如如下代码, 如果在浏览器环境下执行, 严格模式和非严格模式的结果是不同的:

- 非严格模式会输出 hello
- 严格模式会报错, Uncaught TypeError: Cannot read properties of undefined (reading 'a')

```
var a = 'hello'

var obj = {
    a: 'lubai',
    foo: function() {
        // 'use strict';
```

```
        console.log(this.a)
    }
}

var bar = obj.foo

bar() // hello
```

Tips: 普通函数做为参数传递的情况, 比如setTimeout, setInterval, 非严格模式下的this指向全局对象

```
var name = 'lubai';
var person = {
    name: 'hahahahahah',
    sayHi: sayHi
}
function sayHi(){
    console.log(this); // { name: hahahahhah, sayHi: Fn }
    setTimeout(function(){
        console.log('Hello,', this.name); // Hello, lubai
    })
}
person.sayHi();
```

2. 隐式绑定

与默认绑定相反, 函数调用的时候有显式的修饰, 比如说某个对象调用的函数。

比如下面这段代码, 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。

```
var a = 'hello'

var obj = {
    a: 'lubai',
    foo: function() {
```

```
        console.log(this.a)
    }
}

obj.foo(); // lubai
```

Tips: 那如果有链式调用的情况呢? this会绑定到哪个对象上?

```
function sayHi(){
    console.log('Hello,', this.name);
}
var person2 = {
    name: 'lubai',
    sayHi: sayHi
}
var person1 = {
    name: 'hahhahaahh',
    friend: person2
}

person1.friend.sayHi(); // Hello, lubai
```

3. 显式绑定

通过函数call apply bind 可以修改函数this的指向(call 与 apply 方法都是挂载在 Function 原型下的方法，所有的函数都能使用)

call 和 apply

call和apply都是改变函数的this指向并且执行, 那么有什么异同呢?

- call和apply的第一个参数会绑定到函数体的this上，如果不传参数，例如fun.call()，非严格模式，this默认还是绑定到全局对象
- call函数接收的是一个参数列表，apply函数接收的是一个参数数组。

```
func.call(this, arg1, arg2, ...)
func.apply(this, [arg1, arg2, ...])
```

```
var person = {
    "name": "lubai"
};

function changeWork(company, work) {
    this.company = company;
    this.work = work;
};

changeWork.call(person, '字节', '前端');
console.log(person.work); // '前端'

changeWork.apply(person, ['腾讯', '产品']);
console.log(person.work); // '产品'
```

Tips: 如果我们调用call和apply时, 传入的是基本类型数字或者字符串, 绑定this的时候会把他们转换成对象

```
function getThisType () {
    console.log('this指向内容',this, typeof this);
}
getThisType.call(1); // this指向内容 Number {1} object
getThisType.apply('lubai');  // this指向内容 String {'lubai'} object
```

bind

bind 方法 会创建一个新函数。当这个新函数被调用时，bind() 的第一个参数将作为它运行时的 this，之后的一序列参数将会在传递的实参前传入作为它的参数.

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

看下面这段代码

```
var publicAccounts = {
    name: '爪哇',
    author: 'lubai',
    subscribe: function(subscriber) {
        console.log(`${subscriber} ${this.name}`)
    }
}

publicAccounts.subscribe('部部')   // 部部 爪哇

var subscribe1 = publicAccounts.subscribe.bind({ name: '测试名称A', author: '测试作者B' }, '测试订阅者C')

subscribe1()      // 测试订阅者C 测试名称A
```

4. new绑定

new的作用咱们上节课已经说过了, 这节课再简单提一下:

1. 创建一个空对象
2. 将空对象的 proto 指向原对象的 prototype
3. 执行构造函数中的代码
4. 返回这个新对象

构造函数中的this指向了新生成的实例studyDay.

```
function study(name){
    this.name = name;
}
var studyDay = new study('lubai');
console.log(studyDay); // {name: 'lubai'}
console.log(studyDay.name); // lubai
```

5. this绑定的优先级

new绑定 > 显式绑定 > 隐式绑定 > 默认绑定

- 看一下这段代码输出什么?

  tips: 显示绑定优先级比隐式绑定更高。

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

var obj1 = {
    a: 2,
    foo: foo
}

var obj2 = {
    a: 3,
    foo: foo
}

obj1.foo(); // 2
obj2.foo(); // 3

obj1.foo.call(obj2); // 3
obj2.foo.call(obj1); // 2
```

- 再看下这段代码输出什么?

```
function foo(something) {
    this.a = something
}
var obj1 = {
    foo: foo
}
var obj2 = {}

obj1.foo(2);
console.log(obj1.a); // 2
```

```
obj1.foo.call(obj2, 3);
console.log(obj2.a); // 3

var bar = new obj1.foo(4);
console.log(obj1.a); // 2
console.log(bar.a); //4
```

- 再看下这段代码输出什么?

```
function foo(something) {
    this.a = something
}
var obj1 = {
}

var bar = foo.bind(obj1);
bar(2);
console.log(obj1.a); //2

var baz = new bar(3);
console.log(obj1.a); // 2
console.log(baz.a); // 3
```

箭头函数

箭头函数比较特殊, 咱们单独拎出来看.

- 箭头函数中没有 arguments

普通函数可以通过arguments拿到所有参数, 而箭头函数不可以, 如果你说你经常碰到也有在箭头函数中用arguments的, 那么真正拿的其实是外层的function。

```
function constant() {
    return () => arguments[0]
```

```
}

let result = constant(1);
console.log(result()); // 1
```

如果要拿到所有箭头函数的参数, 我们可以直接用参数的解构

```
let nums = (...nums) => nums;
```

- 箭头函数没有构造函数

箭头函数与正常的函数不同，箭头函数没有构造函数 constructor，所以也不能使用 new 来调用，如果我们直接使用 new 调用箭头函数，会报错。

```
let fun = ()=>{}
let funNew = new fun();
// 报错内容 TypeError: fun is not a constructor
```

- 箭头函数没有原型对象

```
let fun = ()=>{}
console.log(fun.prototype); // undefined
```

- 箭头函数中没有自己的this

箭头函数中如果用到了this, 那么this的指向由定义箭头函数的位置决定, 而不像普通函数是在调用时才绑定的.

咱们把上面讲到的默认绑定Tips的例子稍微改一下, 改成箭头函数看一下输出.

```
var name = 'lubai';
var person = {
```

```
    name: 'hahahahahah',
    sayHi: sayHi
}
function sayHi(){
    console.log(this); // { name: hahahahhah, sayHi: Fn }
    setTimeout(() => {
        console.log('Hello,', this.name); // Hello, hahahahhah
    })
}
person.sayHi();
```

练习

1. 看代码输出

```
var name = '123';

var obj = {
  name: '456',
  print: function() {
      function a() {
          console.log(this.name);
      }
      a();
  }
}

obj.print(); // 123
```

2. 看代码输出

```
function Foo(){
    Foo.a = function(){
        console.log(1);
```

```
    }
    this.a = function(){
        console.log(2)
    }
}

Foo.prototype.a = function(){
    console.log(3);
}

Foo.a = function(){
    console.log(4);
}

Foo.a(); // 4
let obj = new Foo();
obj.a(); // 2
Foo.a(); // 1
```

3. 看代码输出

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

var obj = {
  length: 5,
  method: function(fn) {
    fn(); // 10
    arguments[0](); // 2,  arguments: { 0: fn, 1: 1, length: 2 }
  }
};

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

4. 手写实现call apply bind

闭包的概念及应用场景

定义

闭包是指那些能够访问自由变量的函数。
自由变量是指在函数中使用的，但既不是函数参数也不是函数局部变量的变量。

1. 从理论角度：所有的函数都是闭包。因为它们都在创建的时候就将上层上下文的数据保存起来了。哪怕是简单的全局变量也是如此，因为函数中访问全局变量就相当于是在访问自由变量，这个时候使用最外层的作用域。
2. 从实践角度：以下函数才算是闭包：
   - 即使创建它的上下文已经销毁，它仍然存在（比如，内部函数从父函数中返回）
   - 在代码中引用了自由变量

应用场景

1. 柯里化函数
柯里化的目的在于：避免频繁调用具有相同参数函数，同时又能够轻松的复用。
其实就是封装一个高阶函数。

```
// 假设我们有一个求长方形面积的函数
function getArea(width, height) {
    return width * height
}
// 如果我们碰到的长方形的宽老是10
const area1 = getArea(10, 20)
const area2 = getArea(10, 30)
const area3 = getArea(10, 40)

// 我们可以使用闭包柯里化这个计算面积的函数
function getArea(width) {
    return height => {
        return width * height
    }
```

```
}

const getTenWidthArea = getArea(10)
// 之后碰到宽度为10的长方形就可以这样计算面积
const area1 = getTenWidthArea(20)

// 而且如果遇到宽度偶尔变化也可以轻松复用
const getTwentyWidthArea = getArea(20)
```

2. 使用闭包实现私有方法/变量
   其实就是模块的方式, 现代化的打包最终其实就是每个模块的代码都是相互独立的。

```
function funOne(i){
    function funTwo(){
        console.log('数字: ' + i);
    }
    return funTwo;
};
var fa = funOne(110);
var fb = funOne(111);
var fc = funOne(112);
fa();       // 输出: 数字: 110
fb();       // 输出: 数字: 111
fc();       // 输出: 数字: 112
```

3. 匿名自执行函数

```
var funOne = (function(){
  var num = 0;
  return function(){
      num++;
      return num;
  }
})();
```

```
console.log(funOne());       // 输出: 1
console.log(funOne());       // 输出: 2
console.log(funOne());       // 输出: 3
```

4. 缓存一些结果
   比如在外部函数创建一个数组, 闭包函数内可以更改/获取这个数组的值，其实还是延长变量的生命周期，但是不通过全局变量来实现。

```
function funParent(){
    let memo = [];
    function funTwo(i){
        memo.push(i);
        console.log(memo.join(','))
    }
    return funTwo;
};

const fn = funParent();

fn(1);
fn(2);
```

总结

- 创建私有变量
- 延长变量的生命周期
  一般函数的词法环境在函数返回后就被销毁，但是闭包会保存对创建时所在词法环境的引用，即便创建时所在的执行上下文被销毁，但创建时所在词法环境依然存在，以达到延长变量的生命周期的目的

代码题

1. 实现compose函数, 得到如下输出

```
// 实现一个compose函数，用法如下：
function fn1(x) {
    return x + 1;
}

function fn2(x) {
    return x + 2;
}

function fn3(x) {
    return x + 3;
}

function fn4(x) {
    return x + 4;
}


const a = compose(fn1, fn2, fn3, fn4);
console.log(a(1)); // 1+4+3+2+1=11
```

2. 实现一个柯里化函数

```
function currying() {

}

const add = (a, b, c) => a + b + c;
const a1 = currying(add, 1);
const a2 = a1(2);
console.log(a2(3)) // 6
```

作用域

作用域是在运行时代码中的某些特定部分中变量，函数和对象的可访问性。

换句话说，作用域决定了代码区块中变量和其他资源的可见性。

作用域就是一个独立的地盘，让变量不会外泄、暴露出去。也就是说作用域最大的用处就是隔离变量，不同作用域下同名变量不会有冲突。

ES6 之前 JavaScript 没有块级作用域,只有全局作用域和函数作用域。ES6的到来，为我们提供了**块级作用域**,可通过新增命令let和const来体现。

全局作用域

在代码中任何地方都能访问到的对象拥有全局作用域。

- 最外层函数 和在最外层函数外面定义的变量拥有全局作用域

```
var outVariable = "我是最外层变量"; //最外层变量
function outFun() { //最外层函数
    var inVariable = "内层变量";
    function innerFun() { //内层函数
        console.log(inVariable);
    }
    innerFun();
}
console.log(outVariable); //我是最外层变量
outFun(); //内层变量
console.log(inVariable); //inVariable is not defined
innerFun(); //innerFun is not defined
```

- 所有未定义直接赋值的变量自动声明为拥有全局作用域

```
function outFun2() {
    variable = "未定义直接赋值的变量";
    var inVariable2 = "内层变量2";
```

```
}
outFun2();
console.log(variable); //未定义直接赋值的变量
console.log(inVariable2); //inVariable2 is not defined
```

- 所有window对象的属性拥有全局作用域

  window.location

- 弊端

  如果我们写了很多行 JS 代码，变量定义都没有用函数包括，那么它们就全部都在全局作用域中。这样就会 污染全局命名空间, 容易引起命名冲突。

函数作用域

函数作用域,是指声明在函数内部的变量，和全局作用域相反，局部作用域一般只在固定的代码片段内可访问到，最常见的例如函数内部。

```
function doSomething(){
    var blogName="浪里行舟";
    function innerSay(){
        alert(blogName);
    }
    innerSay();
}
alert(blogName); // 报错
innerSay(); // 报错
```

作用域是分层的，内层作用域可以访问外层作用域的变量，反之则不行

块级作用域

块级作用域可通过新增命令let和const声明，所声明的变量在指定块的作用域外无法被访问。块级作用域在如下情况被创建：

- 在一个函数内部
- 在一个代码块（由一对花括号包裹）内部

let 声明的语法与 var 的语法一致。你基本上可以用 let 来代替 var 进行变量声明，但会将变量的作用域限制在当前代码块中。

块级作用域有以下几个特点：

- 声明变量不会提升到代码块顶部
- 禁止重复声明
- 变量只在当前块内有效

一道比较经典的面试题, 涉及作用域以及事件循环:

```
for(var i = 0; i < 10; i++) {
  setTimeout(function(){
      console.log(i)
  })
}
// 输出 10 10 10 10 10 10 10 10 10 10

for(let i = 0; i < 10; i++) {
  setTimeout(function(){
      console.log(i)
  })
}
// 输出 0 1 2 3 4 5 6 7 8 9
```

第一个变量i是用var声明的，在全局范围内有效，所以全局中只有一个变量i，每次循环时，setTimeOut定时器里指的是全局变量i，而循环里的十个setTimeOut是在循环结束后才执行，所以输出十个10。

第二个变量i是用let声明的，当前的i
只在本轮循环中有效，每次循环的i其实都是一个新的变量，所以setTImeOut定时器的里面的i其实不是同一变量，所以输出0123456789

作用域链

有点类似于原型链, 在原型中我们找一个属性的时候, 如果当前实例找不到, 就会去父级原型去找.

作用域链也是类似的原理, 找一个变量的时候, 如果当前作用域找不到, 那就会逐级往上去查找, 直到找到全局作用域还是没找到，就真找不到了.

Tips: 那最先在哪个作用域里寻找呢? 在执行函数的那个作用域? 还是在创建函数的作用域?

记住！！ 要到创建这个函数的那个域”。 作用域中取值,这里强调的是“创建”，而不是“调用”

```
var a = 10
function fn() {
  var b = 20
  function bar() {
    console.log(a + b) //30
  }
  return bar
}
var x = fn(),
  b = 200
x() //bar()
```

Coding

1. 看一下输出

```
var b = 10;
(function b(){
  b = 20;
    // 内部作用域，会先去查找是有已有变量b的声明，有就直接赋值20，确实有了呀。发现了具名函数function b(){}，拿此b做赋值；
    // IIFE的函数无法进行赋值（内部机制，类似const定义的常量），所以无效。
  console.log(b); // fn b
```

```
    console.log(window.b); // 10
})();
```

- 函数表达式与函数声明不同，函数名只在该函数内部有效，并且此绑定是常量绑定。
- 对于一个常量进行赋值，在 strict 模式下会报错，非 strict 模式下静默失败。
- IIFE中的函数是函数表达式，而不是函数声明。

2. 看一下输出

```
var a = 3;

function c() {
    alert(a);
}
(function () {
    var a = 4;
    c(); // 3
})();
```

3. 看一下输出

```
function v() {
    var a = 6;
    function a() {

    }
    console.log(a);
}

v(); // 6
```

```
function v() {
    var a;
```

```
    function a() {

    }
    console.log(a);
}

v(); // fn a
```

js会把所有变量都集中提升到作用域顶部事先声明好，但是它赋值的时机是依赖于代码的位置，那么js解析运行到那一行之后才会进行赋值，还没有运行到的就不会事先赋值。也就是变量会事先声明，但是变量不会事先赋值。

碰到这种问题可以先想一下变量提升和函数声明提升的规则, 原则上是变量被提升到最顶部, 函数声明被提升到最顶部变量的下方.

尝试着把这两段代码在大脑中编译一下:

- 第一段代码

```
function v() {
    var a;
    function a() {

    }
    a=6;
    console.log(a);
}

v(); // 6
```

- 第二段代码

```
function v() {
    var a;
    function a() {
```

```
    }
    console.log(a);
}

v(); // fn a
```

4. 看一下输出

```
function v() {
    console.log(a); // fn a

    var a = 1;

    console.log(a); // 1

    function a() {

    }

    console.log(a); // 1

    console.log(b); // fn b

    var b = 2;

    console.log(b); // 2

    function b() {

    }

    console.log(b); // 2
}
v();
```

按照刚才的思路转换一下：

```
function v() {
    var a;
    var b;
    function a() {}
    function b() {}

    console.log(a); // fn a
    a=1;
    console.log(a); // 1
    console.log(a); // 1

    console.log(b); // fn b
    b=2;
    console.log(b); // 2
    console.log(b); // 2
}
v();
```