深入call,apply,bind到手动封装

jCodeLife
0.194 2020.08.05 21:29:23 字数 2,438 阅读 53

编辑文章

`call、apply、bind的作用是改变函数运行时的this指向。`

我们先来聊聊this
你最开始的时候是在哪里听到this的呢？现在提起它第一印象是什么呢？
记得我最开始接触this时，是在构造函数构造出对象的时，如下：

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.sayInfo = function(){
        console.log("我叫" + this.name + ",我今年" + this.ag + "岁了");
    };
}
var alice = new Person("Alice",20);
```

那时候知道this代表的就是当前对象，this很灵活
但随着学习的深入，发现this被使用地方很多。当逻辑变得复杂时，this指向也变得混乱，以至于一时间难以想明白哪个指向哪个。原来this里面有大学问，所以笔试面试也经常问到。比如下面代码输出什么：

```
var obj = {
  foo: function(){
    console.log(this)
  }
}
var bar = obj.foo
obj.foo()
bar()
```

答案是：obj、window
不知道答对了没有，对了就恭喜你哈！错了也别伤心

我们先来梳理梳理，看看this指向的几种情况吧：

1. 构造函数通过new构造对象时 this指向该对象
   构造函数通过new产生对象时，里面this指代就是这个要生成的新对象；这个比较容易理解，因为new的内部原理：

- 隐式生成this对象
- 执行this.xxx = xxx
- 返回this对象

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    console.log(this);
}
var alice = new Person("Alice",20);
```

jCodeLife
总资产13 (约1.20元)

推荐阅读

```
    a: "obj",
    fn: function () {
        console.log(this.a)
    }
}
obj.fn()//obj
```

4. 普通函数普通执行时，this指向window； 普通执行，就是指非通过其他人调用

```
//1. 普通的函数执行
function fn(){
  console.log(this)//window
}
fn()

//2. 函数嵌套的执行，非别人调用
function fn1() {
    function fn2() {
        console.log(this)//window
    }
    fn2()
}
fn1()

//函数赋值之后再调用
var a = "window";
var obj = {
    a: "obj",
    fn: function () {
        console.log(this.a)
    }
}
var fn1 = obj.fn
fn1()//window
```

5. 数组里面的函数，按数组索引取出运行时，this指向该数组

```
function fn1(){
    console.log(this);
}
function fn2(){}
var arr = [fn1,fn2]
arr[0]();//arr
```

6. 箭头函数内的this值继承自外围作用域

   运行时会首先到父作用域找，如果父作用域还是箭头函数，那么接着向上找，直到找到我们要的this指向。即箭头函数中的 this继承父级的this(父级非箭头函数)。call或者apply都无法改变箭头函数运行时的this指向。

7. call,apply,bind可以改变函数运行时的this指向

   当然是非箭头函数

   这里我们分开来讲并实现封装

- `call`

  call方法第一个参数是要绑定的this指向，后面传入的是函数执行的实参列表。换句话说，this就是你 call 一个函数时，传入的第一个参数。

```
var obj = {}
function fn(){
    console.log(this);
}
fn.call(obj);//obj
```

```
fn()相当于fn.call(null)
```

```
fn1(fn2)相当于fn1.call(null,fn2)
```

```
obj.fn()相当于obj.fn.call(obj)
```

在仔细想想，视乎fn.call(obj)相当于obj对象里添加一个一样的fn函数并执行fn()，执行完后删除该属性。（记住这点，理解这点有助于接下来手写实现call函数）

当call函数传入第一个参数this为null或者undefined时，默认指向window，严格模式下指向undefined

```
var English = 60;
var qulity =60;
var alice = {
    name: "alice",
    age: 10,
    English: 100,
    qulity: 90
}
function sum( {
    console.log(this.English + this.qulity);
}
sum.call(alice);//100+90
sum.call(null);//60+60
```

另外，fn.call(undefined) 或者fn.call(null) 可以简写为 fn.call()

了解了call的基本用法，接下来手写call函数
首先，因为它是每个方法身上都有calll方法，所以call应该是定义在Function原型上的，并且参数个数不定，那就先不写，到时候我们用arguments来操作参数

```
Function.prototype._call = function(){
}
```

再来想想，我们通过_call方法要实现：

1. 改变函数运行时的this指向，让它指向我们传递的第一个参数，即arguments[0]
2. 让函数执行

其实就这两点，关键是怎么实现呢？
上面有一点让大家记住的，就是fn.call(obj)相当于obj对象里添加一个一样的fn函数，并执行fn()，执行完后删除该属性。
先来得到我们传递的第一个参数（this指向），用个变量保存起来，方便到时调用函数。但是当没有传入或者传入null、undefined时默认window：

```
var _obj = arguments[0] || window;
```

接着，在_obj对象中添加一个属性fn，值为要执行call的函数。因为在函数调用call的时候this就是指代该函数，所以：

```
obj.fn = this;
```

递过去显然办不到。这里我们使用一个函数eval()，这个函数可以将传递的字符串当js代码来执行，返回执行结果。
所以我们先将参数都处理成字符串格式就好：

```
var _args = [];
for (var i = 1; i < arguments.length; i++) {
    _args.push("arguments[" + i + "]");
}
var _str = _args.join(",");
```

得到的_str的值为"arguments[1],arguments[2],arguments[3],arguments[4],arguments[5]...."
接着就可以通过eval执行函数了

```
eval('_obj.fn(' + _str + ')');
```

函数执行完，将我们在对象身上添加的fn删掉即可

```
delete _obj.fn;
```

完整代码：

```
Function.prototype._call = function () {
    var _obj = arguments[0] || window;
    _obj.fn = this;//将当前函数赋值给对象的一个属性
    var _args = [];
    for (var i = 1; i < arguments.length; i++) {
        _args.push("arguments[" + i + "]");
    }
    var _str = _args.join(",");
    var result = eval('_obj.fn(' + _str + ')');
    delete _obj.fn;
    return result;
}

var obj = {
    name: 'obj'
}
function fn() {
    console.log(this);
    console.log(arguments);
}
fn._call(obj, 1, 2, 3, 4);

```

修改成ES6的写法：

```
Function.prototype._call = function () {
    let params = Array.from(arguments);//得到所以实参数组
    let _obj = params.splice(0, 1)[0];//获取第一位作为对象，即this指向
    _obj.fn = this
    var result = _obj.fn(...params);//splice截取了第一位，params包含剩下的参数
    delete _obj.fn
    return result;
}
```

- `apply`

  apply跟call非常相似，只是传参形式不同。apply接受两个参数，第一个参数也是要绑定给this的值，第二个参数是一个数组。

跟call一样，当第一个参数为null、undefined的时候，默认指向window。

```
Function.prototype._apply = function (obj, args) {
    var _obj = obj || window;
    _obj.fn = this;
    // 执行函数_obj.fn()前,将参数处理成字符串，最后删除属性即可
    var result;
    if (args) {
        var _args = [];
        for(var i = 0;i<args.length;i++){
            _args.push('args['+i+']');
        }
        var str = _args.join(",");
        result = eval("_obj.fn(" + str + ")");
    } else {
        result = _obj.fn();
    }
    delete _obj.fn;
    return result;
}
```

用ES6的写法简化如下：

```
Function.prototype._apply = function (_obj, args) {
    _obj.fn = this;
    var result = args ? _obj.fn(...args) : _obj.fn();
    delete _obj.fn;
    return result;
}
```

是不是发现apply 和 call 的用法几乎相同？是的！唯一的差别在于：当函数需要传递多个变量时, apply 可以接受一个数组作为参数输入, call 则是接受一系列的单独变量。

利用call和apply可改变函数this指向的特性，可以借用别的函数实现自己的功能，如下：

```
function Person(name, age, sex) {
    this.name = name;
    this.age = age;
    this.sex = sex;
}
function Student(name, age, sex, grade, tel, address) {
    this.name = name;
    this.age = age;
    this.sex = sex;
    this.grade = grede;
    this.tel = tel;
    this.address = address;
}
var alice = new Student("alice", 20, 'famale',88,"134****4559","海天二路33号")
```

我们发现在构建Student对象时，Person和Student两个类存在很大的耦合，代码优化中也说尽量低耦合。那这种情况我们可以使用call和apply

```
function Person(name, age, sex) {
    this.name = name;
    this.age = age;
    this.sex = sex;
}
function Student(name, age, sex, grade, tel, address) {
    Person.call(this,name, age, sex);
    this.grade = grade;
    this.tel = tel;
    this.address = address;
```

同样利用call和apply来借用别的函数实现自己的功能还有很多，再举几个例子开发一下思路：

- 将类数组转化为数组
  如，将函数arguments类数组转成数组返回

```
function fn(){
    return Array.prototype.slice.call(arguments);
}
console.log(fn(1,2,3,4));//[1,2,3,4]
```

- 数组追加

```
var arr1 = [1,2,3];
var arr2 = [4,5,6];
var total = [].push.apply(arr1, arr2);//6
// arr1 [1, 2, 3, 4, 5, 6]
```

- 判断变量类型是不是数组

```
function isArray(obj){
    return Object.prototype.toString.call(obj) == '[object Array]';
}
isArray([]) // true
isArray('a') // false
```

- 简化比较长的代码执行语句
  比如console.log()每次要写那么多个字母，写个log()不好吗

```
function log(){
  console.log.apply(console, arguments);
}
```

当然也有更方便的 var log = console.log()

讲完call和apply，最后再来看看bind

- `bind`
  和call很相似，第一个参数是this的指向，从第二个参数开始是接收的参数列表。区别在于bind非立即执行，而是返回函数等待执行。

我们先看个例子，再来详细小结一下bind：

```
var n = 1;
var obj = {
    n:2
}
function fn(){
    console.log(this.n);
}
var temp = fn.bind(obj);//temp-->fn(){}
temp();//2
```

再来看：

```
function fn1() {
```

```
    z = 3;
var fn2 = fn1.bind(o,x,y);
fn2("c");//o, [1, 2, "c"]
```

请再来看看，哈哈：

```
function Fn1() {
    console.log(this,arguments)
}
var obj = {};
var Fn2 = Fn1.bind(obj);
console.log(new Fn2().constructor);//Fn1
```

惊不惊喜意不意外，new Fn2().constructor居然是Fn1！而且new Fn2()里面的this是对象本身，因为new的关系

我们一起来总结一下吧

小结：

1. 函数调用bind方法时，需要传递函数执行时的this指向，选择传递任意多个实参(x,y,z....)；

2. 返回新的函数等待执行；

3. 返回的新函数在执行时，功能跟旧函数一致，但this指向变成了bind的第一个参数；

4. 同样在新函数执行时，传递的参数会拼接到函数调用bind方法时传递的实参后面，两部分参数拼接后，一并在内部传递给函数作为参数执行；

5. bind返回的函数通过new构造的对象的构造函数constructor依旧是旧的函数(如上例子new Fn2().constructor是Fn1)；而且bind传递的this指向，不会影响通过bind返回的函数通过new构造的对象其里面的this；

所以有了这些总结，我们来开始模拟实现我们的bind

为了不乱，我们先实现基本功能吧：

```
Function.prototype._bind = function (target) {
    //target:改变返回函数执行时的this指向
    var obj = target || window;
    var args = [].slice.call(arguments,1);//获取bind时传入的绑定实参
    var self = this;//要bind的函数
    var _fn= function(){
        var _args = [].slice.call(arguments,0);//新函数执行时传递的实际参数
        return self.apply(obj,args.concat(_args));
    }
    return _fn
}
```

接着，让new新函数生成对象的constructor是旧函数

通过中间函数实现继承

```
Function.prototype._bind = function (target) {
    //target:改变返回函数执行时的this指向
    var obj = target || window;
    var args = [].slice.call(arguments,1);//获取bind时传入的绑定实参
    var self = this;//要bind的函数
    var temp = function(){};//作为中间函数,用于实现继承
    var _fn= function(){
        var _args = [].slice.call(arguments,0);//新函数执行时传递的实际参数
        return self.apply(obj,args.concat(_args));
    }
    //让中间函数的原型指向，要bind函数的原型
    temp.prototype = self.protoype;
    //让新函数的原型指向中间temp的对象，然后找到要bind函数的原型
    _fn.prototype = new temp();//这样新函数生成的对象的constructor就能找到旧的函数
    return _fn
}
```

那怎么来判断是否以new的方式来执行新的这个函数呢？

通过instanceof来判断(这里会比较难理解)
instanceof的用法是判断左边对象是不是右边函数构造出来的
最终的代码如下：

```
//bind的模拟实现
Function.prototype._bind = function (target) {
    //target:改变返回函数执行时的this指向
    var temp = function () { };//作为中间函数,用于实现继承
    //target不存在this默认window，当new调用时无需修改this指向
    var obj = this instanceof temp ? this : (target || window);
    var args = [].slice.call(arguments, 1);//获取bind时传入的绑定实参
    var self = this;//要bind的函数
    var _fn = function () {
        var _args = [].slice.call(arguments, 0);//新函数执行时传递的实际参数
        return self.apply(obj, args.concat(_args));
    }
    //让中间函数的原型指向，要bind函数的原型
    temp.prototype = self.protoype;
    //让新函数的原型指向中间temp的对象，然后找到要bind函数的原型
    _fn.prototype = new temp();//这样新函数生成的对象的constructor就能找到旧的函数
    return _fn
}

//下面为测试代码
var a = 1;
var o = {
    a:2
}
function A(){
    console.log(this.a);
    return arguments;
}
var fn1 =  A._bind(o,1,2,3);
var fn2 = A.bind(o,4,5,6);
console.log(fn1(111),fn2(222))
```

最后总结一下call，apply，bind及其区别

总结

相同点：

- call、apply、bind的作用都是改变函数运行时的this指向。
- 第一个参数都是this指向

区别在于：

- call和apply比较，传参形式不一样；call需要把实参按照形参的个数一个一个传入，apply的第二个参数只需要传入一个数组
- bind和call比较，传参形式跟call一样，但是call和apply是绑定this指向直接执行函数，bind是绑定好this返回函数待执行。

参考资料
原型，原型链，call/apply（下）
一次性讲清楚apply/call/bind
call、apply和bind方法的用法以及区别
你不知道的JS-call,apply手写实现
this 的值到底是什么？一次说清楚
你不知道的JS-bind模拟实现

"小礼物走一走，来简书关注我"

赞赏支持

还没有人赞赏，支持一下

jCodeLife 书山有路勤为径，学海无涯苦作舟

总资产13 (约1.20元) 共写了11.5W字 获得214个赞 共21个粉丝

写下你的评论...

全部评论 2 只看作者 关闭评论 按时间倒序 按时间正序

jCodeLife 作者

2楼 08.06 14:16

bind源码这块还有点问题，回头再跟新

赞 回复

jCodeLife 作者

08.06 22:04

已跟新手动实现bind函数

回复

添加新评论

被以下专题收入，发现更多相似内容 投稿管理

+收入我的专题 领跑计划笔记