bind()函数

bind语法格式: func.bind(thisArg[, arg1[, arg2[, ...]]])参数

thisArg: 当绑定函数被调用时，该参数会作为原函数运行时的 this 指向。当使用new 操作符调用绑定函数时，该参数无效。

arg1, arg2, ... : 当绑定函数被调用时，这些参数将置于实参之前传递给被绑定的方法返回值

返回=>有指定的this值和初始化参数改造的原函数拷贝

**几个重要的对比输出:**

function foo() {

this.b = 100;

console.log("this:",this);

console.log("this.a:",this.a);

console.log("this.b:",this.b);

return this.a;

}

var func = foo.bind({a:1});

console.log("func: ",func); //①

console.log("foo: ",foo); //②

new func(); //③

new foo(); //④

func(); //⑤

foo(); //⑥

①结果:

func: ƒ foo() {

this.b = 100;

console.log("this:",this);

console.log("this.a:",this.a);

console.log("this.b:",this.b);

return this.a;

}

②结果:

foo: ƒ foo() {

this.b = 100;

console.log("this:",this);

console.log("this.a:",this.a);

console.log("this.b:",this.b);

return this.a;

}

①②对比总结:

bind返回的函数,跟原函数一模一样

③结果:

this: foo {b: 100}

this.a: undefined

this.b: 100

④结果:

this: foo {b: 100}

this.a: undefined

this.b: 100

③④对比总结:(new的时候的总结)

如果函数通过return返回给一个对象，那么new返回的是个this就指向这个对象

如果函数没有通过return返回给一个对象(或者通过return返回给了一个不是对象的变量),那么通过new返回的是个this，指向的是**构造函数的原型对象**

与传入的参数对象没有任何关系

新对象.\_\_proto\_\_===构造函数.prototype

⑤结果:

this: {a: 1, b: 100}

this.a: 1

this.b: 100

⑥结果:

this: Window {}

this.a: undefined

this.b: 100

⑤⑥对比总结:(执行的时候的总结)

**执行**bind返回的函数,这个函数的this指向的是传入的这个**参数对象**,这个对象保留了自己的属性a,并且带上了原函数的属性b

也就是执行两个函数的时候存在区别!!

const person = {

age: 20,

getAge() {

return this.age

}

}

const getAge = person.getAge

console.log(getAge()) //执行时this指向window对象,所以结果为undefined

const person = {

age: 20,

getAge() {

return this.age

}

}

const getAge = person.getAge.bind(person) //把this指向了person对象

console.log(getAge()) // 可以取到里面的age值,输出20

**用原生js给函数的原型添加一个bind方法:**

Function.prototype.bind=function (context) {

if(typeof this !=='function'){

throw new Error(`${this.name} is not a function`)

}

const srcFun=this// 保存原始函数

const arg=Array.prototype.slice.call(arguments,1)// 把arguments类数组转为真实数组

let noop=function(){}

const fBound= function () {

if(this instanceof noop){

context=this

}

// 合并新旧参数

return srcFun.apply(context,arg.concat(Array.prototype.slice.call(arguments,0)))

}

if(this.prototype){

noop.prototype=this.prototype//维护原型关系，指向原始函数

}

fBound.prototype=new noop()//新函数的prototype的\_\_proto\_\_指向noop.prototype

return fBound

}