function Point(x, y) {

this.x = x;

this.y = y;

}

Point.prototype.toString = function() {

return this.x + ',' + this.y;

};

var p = new Point(1, 2);

p.toString(); // '1,2'

var emptyObj = {};

var YAxisPoint = Point.bind(emptyObj, 0/\*x\*/);

// 以下这行代码在 polyfill 不支持,

// 在原生的bind方法运行没问题:

//(译注：polyfill的bind方法如果加上把bind的第一个参数，即新绑定的this执行Object()来包装为对象，Object(null)则是{}，那么也可以支持)

var YAxisPoint = Point.bind(null, 0/\*x\*/);

//0作为arguments参数传给YAxisPoint，当newYAxisPoint 时 调试进入bind函数，原有对象x = 0，5被argumsnts添加进去了。

var axisPoint = new YAxisPoint(5);

axisPoint.toString(); // '0,5'

如果

var YAxisPoint = Point.bind(null, 0/\*x\*/，3);

var axisPoint = new YAxisPoint(5);

axisPoint.toString(); // '0,3'

那么 Point bind时，参数是怎么对应上的，只传0时，this.y为undefined，那么new YAxisPoint(5)5是什么赋给y的

**涉及到柯里化的问题，参数部分传递，提前绑定好函数里面的某些参数,达到参数复用的效果。**

if (!Function.prototype.binds) {

Function.prototype.binds = function(oThis) {

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

// closest thing possible to the ECMAScript 5

// internal IsCallable function

throw new TypeError('Function.prototype.bind - what is trying to be bound is not callable');

}

var aArgs = Array.prototype.slice.call(arguments, 1),

fToBind = this,

fNOP = function() {},

fBound = function() {

return fToBind.apply(this instanceof fNOP

? this

: oThis,

// 获取调用时(fBound)的传参.bind 返回的函数入参往往是这么传递的

aArgs.concat(Array.prototype.slice.call(arguments)));

////concat连接参数了，初始bind时aArgs从bind中取参数，然后返回一个函数出去，这里形成了闭包，当构建的新函数执行时，仍会调用闭包的变量，且用concat合并了新的参数，再传给调用函数。

};

// 维护原型关系

if (this.prototype) {

// Function.prototype doesn't have a prototype property

fNOP.prototype = this.prototype;

}

fBound.prototype = new fNOP();

console.log(fBound)

return fBound;

};

}

bind 和 call/apply 一样，都是用来改变上下文 this 指向的，不同的是，call/apply 是直接使用在函数上，**而 bind 绑定 this 后返回一个函数（闭包），bind的时候不会立即执行。**

**对构造函数使用bind时**

function Test3(a, b) { this.a = a; this.b = b; }

Test3.prototype.add = function () { return this.a + this.b; }

// 如果不用 bind，正常来说这样处理

var t1 = new Test3(1, 2);

t1.add(); // 3, this 指向 t1

// 使用 bind

var NewTest3 = Test3.bind(null, 3);

var t2 = new NewTest3(4);

t2.add(); // 7, this 指向 t2

将绑定函数当做构造函数使用，bind 提供的 this 指向无效，但是还是可以预设参数。

bind的实现：

1、简单实现参数和方法绑定

// 利用 apply 改变 this 指向

Function.prototype.bind = function(context) {

var that = this;

return function() {

return that.apply(context, arguments); }

}

2、上述实现只能传递一个参数（context），原生的 bind 方法可以传入多个参数，作如下修改

// 将 bind 方法的参数提取出来拼接到返回的闭包函数中

Function.prototype.bind = function(context) {

var that = this;

var args = Array.prototype.slice.call(arguments, 1);

return function() {

// 预设参数一定是 args 在前拼接

return that.apply(context, args.concat(Array.prototype.slice.call(arguments))) ;

}

}

3、修改之后可以传多个参数了，测试一和测试二的场景也满足，但是测试三的场景不满足（不能作为构造函数）

Function.prototype.bind = function(context) {

var that = this;

var args = Array.prototype.slice.call(arguments, 1);

var bound = function() {

if(this instanceof bound) {

// 如果是作为构造函数，那么第一个参数 context 就不需要了

return that.apply(this, args.concat(Array.prototype.slice.call(arguments)));

} else {

return that.apply(context, args.concat(Array.prototype.slice.call(arguments))) ;

}

}

// 维护原型关系

if(this.prototype) {

bound.prototype = this.prototype;

}

return bound;

}