实现 Function.phototype.bind 函数

先讲 this 是什么，this 为什么会变：
js 执行一个函数会创建一个执行上下文（也就是执行环境）
具体就是当执行到一个函数时。底层会 1.先创建一些额外的东西；2.再执行函数里面的代码
一. 创建 额外的东西（有 3 个）：

1. 创建作用域链中的变量的引用（生成一个指针指过去）：

例如：执行下方 c 函数时会现在内存中创建 b 这个变量的引用指针：

```
var a = function() {
    var b = 5
    var c = function() {
        // 这里能引用到 c 函数外面的变量 b，这就是 js 作用域链的功劳
        log(b)    // 5
    }
    c()
}
a()
```

2. 创建函数中的变量，函数和参数：

例如: 执行 c 函数时，会在内存中创建 d e f 这三个变量，以及给 g 变量分配空间（这一步叫变量提升）

```
var c = function(d, e, f) {
    var g = 10
}
c(1, 2, 3)
```

3. 求 this：this 是 js 提供给函数特有的一个东西。然而 this 值是会变的，故这里用 "求" 这个字眼

二. 开始执行函数里的代码

前面说了 this 是会变的，如何确定 this，就一句话：!!! this 值是由函数调用的方式所决定的!!!!
总结来说，!!!! 就是看调用时候函数 . 前面是什么!!!!

例子：

```
var t = {
    b: function() {
        log(this)
    }
}

t.b() // 此时 this 就 b 函数 . 前面决定的，故 this 为 t，这里 log 出来就 t 这个 object
```

var t2 = t.b
t2() // 这句话等价于 window.t2(), 此时 . 前面是 window，故 log 出来是 window

再来个复杂的：

```
var a = {
    b:{
     c: function() {
            log(this)
      }
    }
}
```

a.b.c() // 此时 this 是 b，因为 c 函数前的 . 是 b

总结判断 this 的方法：就是看函数 . 前面是什么

既然 this 会变，我们写代码时想要 this 就是我们需要的 this，那我们需要绑定 this，js 委员会在 1995 年 es3 标志中提供了一个方法来绑定 this
就是 Function.phototype.apply 这个 api 就是用来绑定 this 的，

使用方式： 某函数.apply(给他指定的 this)
例子：
我们有一个全局变量 global ，给他挂上一 a ，即：global.a = 'global'
还有一个 test 的 object ，我们令 test.a = 'test'
看代码：

```
global.a = 'global'
var test = {
    a: 'test',
    b: function() {
        // 我们期望 this 为 test 这个 object
        // 那 this.a 就是 'test'
        console.log('this.a', this.a)
    }
}
// 此时，我用一个变量接住 test.b()
var that = test.b
// 我们期望调用 that 函数时 log 出来是 test
// 我们给他绑定 this
// 第一个参数是强行指定 this,这里我们就指定 test 这个 object
that.apply(test) // 这样运行结果就为 'test'

// 否则
that() // global
```

```
Run:  topic.js
/usr/local/bin/[illegible]
this.a test
this.a global
```

但是 apply 有个缺点就是，就是你调用 a 函数.apply() 函数后，那 a 函数会被立刻执行，

我们想要有一个功能(我们称为 bind 函数)
他可以这样用： var b 函数 = a 函数.bind(this) ,
此时 a 函数 不会执行，
等代码调用 b 函数 时，才会去执行 绑定了 this 的 a 函数

栗子：

```
global.a = 'global'
var test = {
    a: 'test',
    b: function() {
        // 我们期望 this 为 test 这个 object
        // 那 this.a 就是 'test'
        console.log('this.a', this.a)
    }
}
// 此时，我用一个变量接住 test.b()
var that = test.b

// 提前绑定好 that 函数的 this
var 函数b = that.bind(test)

// 需要调用时
函数b() // test
```

```
Run:  topic.js
/us
this.a test
```

进入正题：如何实现 bind 函数
我把重要的说一下，不重要的就先忽略（比如能给定参数列表这个先忽略）：
功能 1. 返回一个函数
功能 2. 给这个函数绑定 this
功能 3.当一个绑定函数是用来构建一个值的（就是 new XX 时），原来提供的 this 就会被忽略
功能 4. 维护原型关系

对应功能 1. 很简单,return 一个 function
对应功能 2. 我们需在借住 Function.phototype.apply 这个 api
对应功能 3. 我们通过判断是不是不是通过 new 来生成实现（写个 if）
对应功能 4. 我们强行改他的原型关系

接着我们先把 功能 1、功能 2 做了：

```
const bind = function(thisArg) {
    // 因为 bind 调用是这种方式: 函数a.bind()
    // 那 this 就是 函数a
    let targetFunction = this
    return function() {
        return targetFunction.apply(thisArg)
    }
}
```

然后我们要把 bind 函数挂到 函数类型上，也就是 Function
那么就这样写：Function.prototype.bind = bind

接着实现功能 3 ：当一个绑定函数是用来构建一个值的（就是 new XX() 时），原来提供的 this 就会被忽略（这个是 es5 标准中 Function.prototype.bind 函数的规定）
那我们在上面的代码进一步编写：

```
const bind = function(thisArg) {
    let targetFunction = this
    let out = function() {
        // js 中通过 new 生成的实例就是 this
        // 我们通过判断 this 这个实例是不是构造自 out 这个函数，就可以判断不是调用 new
        // js 中是通过 instanceof 来判断 (使用方式: 实例 instanceof 构造函数 => true/false)
        if (this instanceof out) {
            // 是通过 new 生成的，把 this 改成当前 this，不去动他
            return targetFunction.apply(this)
        } else {
            // 不是通过 new 生成的，那么跟之前代码一样写
            return targetFunction.apply(thisArg)
        }
    }
    return out
}
```

重点就是 217 行 - 222 行

最后实现功能 4： 维护原型关系
这个简单理解就是：如果 函数 a 是一个类（2015 年前的 js 没有 class 语法，要实现类，得借助 function 这个语法），我们得保证 bind 生成的函数（函数 a 通过 bind 后生成的函数） 能用到 原本函数（函数 a） 这个类的所有实例方法

我们上代码：

```
const bind = function(thisArg) {
    let targetFunction = this
    let out = function() {
        // js 中通过 new 生成的实例就是 this
        // 我们通过判断 this 这个实例是不是构造自 out 这个函数，就可以判断不是调用 new
        // js 中是通过 instanceof 来判断（使用方式：实例 instanceof 构造函数 => true/false)
        if (this instanceof out) {
            // 是通过 new 生成的，把 this 改成当前 this，不去动他
            return targetFunction.apply(this)
        } else {
            // 不是通过 new 生成的，那么跟之前代码一样写
            return targetFunction.apply(thisArg)
        }
    }

    // 实现目的 3
    const nop = function() {}
    nop.prototype = targetFunction.prototype
    out.prototype = new nop()

    return out
}
```

那么我们在之前的代码再加上 红框中的代码 就行了
这个不理解就背就行（细枝末节的东西）

到此，bind 函数的主要功能实现我们就讲完了，还有一些简单的功能补充下

1. 能给定参数列表: 就是当函数被调用时，可以预先给函数的传参数
   使用方式 bind(this, 参数 1, 参数 2，参数 3...)
2. bind 后生产的函数也能够传参数

代码：

```
const bind = function(thisArg, ...argsA) {
    let targetFunction = this
    let out = function(...argsB) {
        // js 中通过 new 生成的实例就是 this
        // 我们通过判断 this 这个实例是不是构造自 out 这个函数，就可以判断不是调用 new
        // js 中是通过 instanceof 来判断（使用方式：实例 instanceof 构造函数 => true/false)
        if (this instanceof out) {
            // 是通过 new 生成的，把 this 改成当前 this，不去动他
            return targetFunction.apply(this, argsA.concat(argsB))
        } else {
            // 不是通过 new 生成的，那么跟之前代码一样写
            return targetFunction.apply(thisArg, argsA.concat(argsB))
        }
    }

    // 实现目的 3
    const nop = function() {}
    nop.prototype = targetFunction.prototype
    out.prototype = new nop()

    return out
}
```

加上面红线的代码

完整文字代码：

```
const bind = function(thisArg, ...argsA) {
    let targetFunction = this
    let out = function(...argsB) {
        // js 中通过 new 生成的实例就是 this
        // 我们通过判断 this 这个实例是不是构造自 out 这个函数，就可以判断不是调用 new
        // js 中是通过 instanceof 来判断（使用方式：实例 instanceof 构造函数 => true/false)
        if (this instanceof out) {
            // 是通过 new 生成的，把 this 改成当前 this，不去动他
            return targetFunction.apply(this, argsA.concat(argsB))
```

```
        } else {
            // 不是通过 new 生成的，那么跟之前代码一样写
            return targetFunction.apply(thisArg, argsA.concat(argsB))
        }
    }

    // 实现功能 4
    const nop = function() {}
    nop.prototype = targetFunction.prototype
    out.prototype = new nop()

    return out
}

Function.prototype.bind = bind
```