this、apply、call、bind

> 链接：https://juejin.cn/post/6844903496253177863

这又是一个面试经典问题，也是 ES5中众多坑中的一个，在 ES6 中可能会极大避免 this 产生的错误，但是为了一些老代码的维护，最好还是了解一下 this 的指向和 call、apply、bind 三者的区别。

this 的指向

在 ES5 中，其实 this 的指向，始终坚持一个原理：**this 永远指向最后调用它的那个对象**，来，跟着我朗读三遍：**this 永远指向最后调用它的那个对象，this 永远指向最后调用它的那个对象，this 永远指向最后调用它的那个对象。**记住这句话，this 你已经了解一半了。

下面我们来看一个最简单的例子： 例 1：

```
var name = "windowsName";
function a() {
    var name = "Cherry";

    console.log(this.name);          // windowsName

    console.log("inner:" + this);    // inner: Window
}
a();
console.log("outer:" + this)         // outer: Window复制代码
```

这个相信大家都知道为什么 log 的是 windowsName，因为根据刚刚的那句话“**this 永远指向最后调用它的那个对象**”，我们看最后调用 `a` 的地方 `a();`，前面没有调用的对象那么就是全局对象 window，这就相当于是 `window.a()`；注意，这里我们没有使用严格模式，如果使用严格模式的话，全局对象就是 `undefined`，那么就会报错 `Uncaught TypeError: Cannot read property 'name' of undefined`。

再看下这个例子： 例 2：

```
var name = "windowsName";
var a = {
    name: "Cherry",
    fn : function () {
        console.log(this.name);      // Cherry
    }
}
a.fn();复制代码
```

在这个例子中，函数 fn 是对象 a 调用的，所以打印的值就是 a 中的 name 的值。是不是有一点清晰了呢~

我们做一个小小的改动： 例 3：

```
var name = "windowsName";
    var a = {
        name: "Cherry",
        fn : function () {
            console.log(this.name);      // Cherry
        }
    }
    window.a.fn();复制代码
```

这里打印 Cherry 的原因也是因为刚刚那句话“**this 永远指向最后调用它的那个对象**”，最后调用它的对象仍然是对象 a。

我们再来看一下这个例子： 例 4：

```
    var name = "windowsName";
    var a = {
        // name: "Cherry",
        fn : function () {
            console.log(this.name);      // undefined
        }
    }
    window.a.fn();复制代码
```

这里为什么会打印 `undefined` 呢？这是因为正如刚刚所描述的那样，调用 fn 的是 a 对象，也就是说 fn 的内部的 this 是对象 a，而对象 a 中并没有对 name 进行定义，所以 log 的 `this.name` 的值是 `undefined`。

这个例子还是说明了：**this 永远指向最后调用它的那个对象**，因为最后调用 fn 的对象是 a，所以就算 a 中没有 name 这个属性，也不会继续向上一个对象寻找 `this.name`，而是直接输出 `undefined`。

再来看一个比较坑的例子： 例 5：

```
    var name = "windowsName";
    var a = {
        name : null,
        // name: "Cherry",
        fn : function () {
            console.log(this.name);      // windowsName
        }
    }

    var f = a.fn;
    f();复制代码
```

这里你可能会有疑问，为什么不是 `Cherry`，这是因为虽然将 a 对象的 fn 方法赋值给变量 f 了，但是没有调用，再接着跟我念这一句话：“**this 永远指向最后调用它的那个对象**”，由于刚刚的 f 并没有调用，所以 `fn()` 最后仍然是被 window 调用的。所以 this 指向的也就是 window。

由以上五个例子我们可以看出，this 的指向并不是在创建的时候就可以确定的，在 es5 中，永远是**this 永远指向最后调用它的那个对象。**

再来看一个例子： 例 6：

```
var name = "windowsName";

function fn() {
    var name = 'Cherry';
    innerFunction();
    function innerFunction() {
        console.log(this.name);      // windowsName
    }
}

fn()复制代码
```

读到现在了应该能够理解这是为什么了吧(o゜▽゜)o。

怎么改变 this 的指向

改变 this 的指向我总结有以下几种方法：

- 使用 ES6 的箭头函数
- 在函数内部使用 `_this = this`
- 使用 `apply`、`call`、`bind`
- new 实例化一个对象

例 7：

```
var name = "windowsName";

var a = {
    name : "Cherry",

    func1: function () {
        console.log(this.name)
    },

    func2: function () {
        setTimeout(  function () {
            this.func1()
        },100);
    }

};

a.func2()     // this.func1 is not a function复制代码
```

在不使用箭头函数的情况下，是会报错的，因为最后调用 `setTimeout` 的对象是 window，但是在 window 中并没有 func1 函数。

我们在改变 this 指向这一节将把这个例子作为 demo 进行改造。

箭头函数

众所周知，ES6 的箭头函数是可以避免 ES5 中使用 this 的坑的。**箭头函数的 this 始终指向函数定义时的 this，而非执行时。**，箭头函数需要记着这句话："箭头函数中没有 this 绑定，必须通过查找作用域链来决定其值，如果箭头函数被非箭头函数包含，则 this 绑定的是最近一层非箭头函数的 this，否则，this 为 undefined"。

例 8：

```
var name = "windowsName";

var a = {
    name : "Cherry",

    func1: function () {
        console.log(this.name)
    },

    func2: function () {
        setTimeout( () => {
            this.func1()
        },100);
    }

};

a.func2()     // Cherry复制代码
```

在函数内部使用 `_this = this`

如果不使用 ES6，那么这种方式应该是最简单的不会出错的方式了，我们是先将调用这个函数的对象保存在变量 `_this` 中，然后在函数中都使用这个 `_this`，这样 `_this` 就不会改变了。 例 9：

```
var name = "windowsName";

var a = {

    name : "Cherry",

    func1: function () {
        console.log(this.name)
    },

    func2: function () {
        var _this = this;
        setTimeout( function() {
            _this.func1()
        },100);
    }

};

a.func2()     // Cherry复制代码
```

这个例子中，在 func2 中，首先设置 `var _this = this;`，这里的 `this` 是调用 `func2` 的对象 a，为了防止在 `func2` 中的 setTimeout 被 window 调用而导致的在 setTimeout 中的 this 为 window。我们将 `this(指向变量 a)` 赋值给一个变量 `_this`，这样，在 `func2` 中我们使用 `_this` 就是指向对象 a 了。

使用 apply、call、bind

使用 apply、call、bind 函数也是可以改变 this 的指向的，原理稍后再讲，我们先来看一下是怎么实现的：

使用 apply

例 10：

```
    var a = {
        name : "Cherry",

        func1: function () {
            console.log(this.name)
        },

        func2: function () {
            setTimeout(  function () {
                this.func1()
            }.apply(a),100);
        }

    };

    a.func2()             // Cherry复制代码
```

使用 call

例 11：

```
    var a = {
        name : "Cherry",

        func1: function () {
            console.log(this.name)
        },

        func2: function () {
            setTimeout(  function () {
                this.func1()
            }.call(a),100);
        }

    };

    a.func2()             // Cherry复制代码
```

使用 bind

例 12：

```
    var a = {
        name : "Cherry",

        func1: function () {
            console.log(this.name)
        },

        func2: function () {
            setTimeout(  function () {
                this.func1()
            }.bind(a)(),100);
        }

    };

    a.func2()             // Cherry复制代码
```

apply、call、bind 区别

刚刚我们已经介绍了 apply、call、bind 都是可以改变 this 的指向的，但是这三个函数稍有不同。

在 MDN 中定义 apply 如下；

> apply() 方法调用一个函数, 其具有一个指定的this值，以及作为一个数组（或类似数组的对象）提供的参数

语法：

> fun.apply(thisArg, [argsArray])

- thisArg：在 fun 函数运行时指定的 this 值。需要注意的是，指定的 this 值并不一定是该函数执行时真正的 this 值，如果这个函数处于非严格模式下，则指定为 null 或 undefined 时会自动指向全局对象（浏览器中就是window对象），同时值为原始值（数字，字符串，布尔值）的 this 会指向该原始值的自动包装对象。
- argsArray：一个数组或者类数组对象，其中的数组元素将作为单独的参数传给 fun 函数。如果该参数的值为 null 或 undefined，则表示不需要传入任何参数。从ECMAScript 5 开始可以使用类数组对象。浏览器兼容性请参阅本文底部内容。

apply 和 call 的区别

其实 apply 和 call 基本类似，他们的区别只是传入的参数不同。

call 的语法为：

```
  fun.call(thisArg[, arg1[, arg2[, ...]]])复制代码
```

所以 apply 和 call 的区别是 call 方法接受的是若干个参数列表，而 apply 接收的是一个包含多个参数的数组。

例 13：

```
var a ={
    name : "Cherry",
    fn : function (a,b) {
        console.log( a + b)
    }
}

var b = a.fn;
b.apply(a,[1,2])     // 3复制代码
```

例 14：

```
var a ={
    name : "Cherry",
    fn : function (a,b) {
        console.log( a + b)
    }
}

var b = a.fn;
b.call(a,1,2)       // 3复制代码
```

bind 和 apply、call 区别

我们先来将刚刚的例子使用 bind 试一下

```
var a ={
    name : "Cherry",
    fn : function (a,b) {
        console.log( a + b)
    }
}

var b = a.fn;
b.bind(a,1,2)复制代码
```

我们会发现并没有输出，这是为什么呢，我们来看一下 MDN 上的文档说明：

> bind()方法创建一个新的函数, 当被调用时，将其this关键字设置为提供的值，在调用新函数时，在任何提供之前提供一个给定的参数序列。

所以我们可以看出，bind 是创建一个新的函数，我们必须要手动去调用：

```
var a ={
    name : "Cherry",
    fn : function (a,b) {
        console.log( a + b)
    }
}

var b = a.fn;
b.bind(a,1,2)()           // 3复制代码
```

========== 更新==========

JS 中的函数调用

看到留言说，很多童靴不理解为什么 例 6 的 innerFunction 和 例 7 的 this 是指向 window 的，所以我就来补充一下 JS 中的函数调用。 例 6:

```
var name = "windowsName";

function fn() {
    var name = 'Cherry';
    innerFunction();
    function innerFunction() {
        console.log(this.name);      // windowsName
    }
}

fn()复制代码
```

例 7:

```
var name = "windowsName";

var a = {
    name : "Cherry",

    func1: function () {
        console.log(this.name)
    },

    func2: function () {
        setTimeout(  function () {
            this.func1()
        },100);
    }

};

a.func2()     // this.func1 is not a function复制代码
```

函数调用的方法一共有 4 种

1. 作为一个函数调用
2. 函数作为方法调用
3. 使用构造函数调用函数
4. 作为函数方法调用函数（call、apply）

作为一个函数调用

比如上面的 例 1： 例 1：

```
    var name = "windowsName";
    function a() {
        var name = "Cherry";

        console.log(this.name);          // windowsName

        console.log("inner:" + this);    // inner: Window
    }
    a();
    console.log("outer:" + this)         // outer: Window复制代码
```

这样一个最简单的函数，不属于任何一个对象，就是一个函数，这样的情况在 JavaScript 的在浏览器中的非严格模式默认是属于全局对象 window 的，在严格模式，就是 undefined。

但这是一个全局的函数，很容易产生命名冲突，所以不建议这样使用。

函数作为方法调用

所以说更多的情况是将函数作为对象的方法使用。比如例 2： 例 2：

```
    var name = "windowsName";
    var a = {
        name: "Cherry",
        fn : function () {
            console.log(this.name);      // Cherry
        }
    }
    a.fn();复制代码
```

这里定义一个对象 `a`，对象 `a` 有一个属性（`name`）和一个方法（`fn`）。

然后对象 `a` 通过 `.` 方法调用了其中的 fn 方法。

然后我们一直记住的那句话“**this 永远指向最后调用它的那个对象**”，所以在 fn 中的 this 就是指向 a 的。

使用构造函数调用函数

> 如果函数调用前使用了 new 关键字, 则是调用了构造函数。 这看起来就像创建了新的函数，但实际上 JavaScript 函数是重新创建的对象：

```
// 构造函数:
function myFunction(arg1, arg2) {
    this.firstName = arg1;
    this.lastName  = arg2;
}

// This    creates a new object
var a = new myFunction("Li","Cherry");
a.lastName;                              // 返回 "Cherry"复制代码
```

这就有要说另一个面试经典问题：new 的过程了，(ฅ_ฅ) 这里就简单的来看一下 new 的过程吧： 伪代码表示：

```
var a = new myFunction("Li","Cherry");

new myFunction{
    var obj = {};
    obj.__proto__ = myFunction.prototype;
    var result = myFunction.call(obj,"Li","Cherry");
    return typeof result === 'obj'? result : obj;
}复制代码
```

1. 创建一个空对象 obj;
2. 将新创建的空对象的隐式原型指向其构造函数的显示原型。
3. 使用 call 改变 this 的指向
4. 如果无返回值或者返回一个非对象值，则将 obj 返回作为新对象；如果返回值是一个新对象的话那么直接直接返回该对象。

所以我们可以看到，在 new 的过程中，我们是使用 call 改变了 this 的指向。

作为函数方法调用函数

> 在 JavaScript 中, 函数是对象。
>
> JavaScript 函数有它的属性和方法。 call() 和 apply() 是预定义的函数方法。 两个方法可用于调用函数，两个方法的第一个参数必须是对象本身
>
> 在 JavaScript 严格模式(strict mode)下, 在调用函数时第一个参数会成为 this 的值， 即使该参数不是一个对象。 在 JavaScript 非严格模式(non-strict mode)下, 如果第一个参数的值是 null 或 undefined, 它将使用全局对象替代。

这个时候我们再来看例 6： 例 6：

```
var name = "windowsName";

function fn() {
    var name = 'Cherry';
    innerFunction();
    function innerFunction() {
        console.log(this.name);      // windowsName
    }
}

fn()复制代码
```

这里的 innerFunction() 的调用是不是属于第一种调用方式：作为一个函数调用（它就是作为一个函数调用的，没有挂载在任何对象上，所以对于没有挂载在任何对象上的函数，在非严格模式下 this 就是指向 window 的）

然后再看一下 例 7： 例 7：

```
var name = "windowsName";

var a = {
    name : "Cherry",

    func1: function () {
        console.log(this.name)
    },

    func2: function () {
        setTimeout(  function () {
            this.func1()
        },100 );
    }

};

a.func2()     // this.func1 is not a function复制代码
```

这个简单一点的理解可以理解为“**匿名函数的 this 永远指向 window**”，你可以这样想，还是那句话**this 永远指向最后调用它的那个对象**，那么我们就来找最后调用匿名函数的对象，这就很尴尬了，因为匿名函数名字啊，笑哭，所以我们是没有办法被其他对象调用匿名函数的。所以说 匿名函数的 this 永远指向 window。

如果这个时候你要问，那匿名函数都是怎么定义的，首先，我们通常写的匿名函数都是自执行的，就是在匿名函数后面加 `()` 让其自执行。其次就是虽然匿名函数不能被其他对象调用，但是可以被其他函数调用啊，比如例 7 中的 setTimeout。