apply和call的用法

JS面向对象编程 学习笔记

call 和 apply

EC3给Function的原型定义了两个方法，它们是 Function.prototype.call 和 Function.prototype.apply。在实际的开发中，特别是函数式编程风格的代码中，call和apply尤为重要。能熟练的使用这两个方法模式我们真正成为一名JavaScript程序员的重要一步。

call 和 apply 的区别

它们的作用其实是一模一样的，区别仅仅在于传入的参数形式不同。

> - apply 接受两个参数，第一个参数用来制定函数体内this的指向，第二个参数为一个带下标的集合，这个集合可以为数组，也可以为类数组，apply方法把这个集合中的元素作为参数传递给被调用的函数。

```
var fn = function (a,b,c){
    alert([a,b,c,]);  // [1,2,3]
};
fn.apply(null,[1,2,3])
```

> - call 传入的参数数量不固定，第一个参用来制定函数体内的this指向，从第二个参数开始，每个参数被依次传入函数体内。

```
var fn = function (a,b,c){
    alert([1,2,3])
}
```

- 当使用 call 或者 apply 时，如果我们传入的第一个参数为null，函数体内的this会默认指向宿主对象，在浏览器中，如果使用严格模式，则还为null。

```
var fn = function (){
    alert(this === window) //true
}
fn.call(null)

var fn2 = function (){
    "use strict"
    alert(this === null) //true
}
fn2.call(null)
```

---

call 和 apply 的用途

1.改变this指向，直接看代码

```
var obj1 = {
    name:"fq"
};
var obj2 = {
    name:"mm"
}

window.name = 'window';

var getName = function (){
    alert(this.name)
}

getName() // window
getName.call(obj1)  //fq
getName.call(obj2)  //mm
```

- 在实际开发中，经常会遇到this指向被不经意改变的场景，比如有一个div节点，div节点的onclick事件中的this本来是指向这个div的。

```
document.getElementById('div').onclick = function (){
    alert(this.id)   //div
}
```

- 假设该事件函数中有一个内部的函数fn，在事件内部调用fn函数时，fn函数体内的this就指向了window，而不是我们预期的div，这个时候我们就可以用call 和 apply去改变this指向了。

```
document.getElementById('div').onclick = function (){
    alert(this.id)   //div
    var fn = function (){
        alert(this.id)  //undefined
    };
    fn();
};

//之前都是保存一下this，更优雅的做法可以这样
document.getElementById('div').onclick = function (){
    alert(this.id)   //div
    var fn = function (){
        alert(this.id)  //undefined
    };
    fn.call(this);
};
```

- 案例:内部丢失的this
  或许你某天会觉得 document.getElementById函数有点太长了,也去你会这么做:

```
var getId = document.getElementById;
getId('div');  //但是会报错...
```

这是因为document.getElementById内部的this实际上在调用的时候 是需要指向document的,所以我们需要手动修正this

```
document.getElementById = (function (fn){
```

```
    return function (){
        return fn.apply(document,arguments);
    }
})(document.getElementById)
```

对于上面的代码，等式右边的函数自执行的结果为内部的匿名函数，但是执行的时候相当于先把之前的 document.getElementById 保存到fn中了，如下：

```

var fn = document.getElementById;
document.getElementById = function (){
    return fn.apply(document,arguments) //传进来的实参在arguments中
}
```

然后当用变量再次存储document.getElementById的时候这时候实际运行的是上面第二个等式后面的函数，然后返回的之前存储的fn运行的结果，但是在函数执行的时候，通过apply修正了this指向document。

2.Function.prototype.bind

大部分高级浏览器都实现了内置的Function.prototype.bind方法，用来指定内部的this指向，它返回一个修改this之后的函数，但是并不会想apply和
call那样直接执行函数，来看下面的代码:

```
var obj = {
    fn(){
        console.log(this);
    }
}
setTimeout(obj.fn, 1000);  //window
setTimeout(obj.fn.bind(obj), 1000); //obj
```

那么咱们看看bind的实现原理是什么

```
Function.prototype.bind = function(context){
    var _this = this;
    return function(){
        return _this.apply(context,arguments);
    }
}
```

也就是先把 之前的函数的引用保存起来，然后返回一个新的函数，只不过这个函数在执行的时候 返回的是保存的引用改变this之后的执行结果。

3.借用其它对象的方法

> 我们都知道，杜鹃既不会筑巢，也不会孵雏，而是把自己的蛋寄托给云雀等其他鸟类，让他们代为孵化和养育。同样，在JavaScript中也存在类似的借用现象。

借用方法的第一种场景是“借用构造函数”，通过这种技术，可以实现一些类似继承的效果：

```
var A = function (name){
    console.log(name)
};
var B = function (){
    A.apply(this,agruments);
};

B.prototype.getName = function (){
    console.log(this.name)
}
var b = new B('momo');
b.getName();  // momo
```

借用方法的第二种场景跟我们更加密切。
函数的参数列表arguments是一个类数组的对象，虽然它也有“小标”,但它并非正在的数组，所以不能像数组一样进行排序操作或者往集合里面添加一个新元素。这种情况下，我们常常会借用Array.prototype对象上的方法。比如想往arguments中添加一个新元素，通常会借用Array.prototype.push;

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

在操作arguments的时候我们经常频繁的去找Array.prototype对象借用方法。
想把arguments转换成真正的数组的时候，可以借用Array.prototype.slice方法，想截取arguments列表中第一个元素的时候，由可以借用Array.prototype.shift方法。这些借用其实

很常见，没什么好说的，那么他们内部实现的机制原理是什么呢？ 不妨咱们翻开v8引擎的源码来看看吧！

```
function ArrayPush(){
    var n = TO_UINT32(this.length); //被push对象的length
    var m = %_ArgumentsLength(); //push的参数个数
    for(var i=0; i<m; i++){
        this[i+n] = %_Arguments[i]; //赋值元素
    }
    this.length = m + n;
    return this.length;
}
```

通过上面这段代码可以看到，Array.prototype.push实际上是一个属性赋值的过过程，把参数按照下标依次添加到被push的对象上面，顺便修改了这个对象的length属性。至于被修改的对象是谁，到底是个数组还是个对象，这个并不重要。

那么改写成 JavaScript 的代码 push 应该是这样的

```
var Utils = {
    push(){
        var n = arguments[0].length || 0,
            m = arguments.length - 1;

        for(var i=0; i < m; i++){
            arguments[0][i+n] = arguments[i + 1]
        }

        arguments[0].length = m + n;

        return arguments[0].length;
    }
}

var o = {};
Utils.push(o,1,2,3); // 3
console.log(o); //Object {0: 1, 1: 2, 2: 3, length: 3}
```

由此可以推断我们可以把“任意”的对象传入Array.prototype.push。为什么要把“任意”这两个字加引号呢？ 因为这个对象其实还要满足2各条件：

> - 对象本身可以存储属性
> - 对象的length属性可读可写

对于第一个条件，对象本身存取属性并没有问题，但是如果借用Array.prototype.push方法的不是一个Object类型数据，而是一个number类型的数据呢？我们无法在number身上存取其他数据，那么从下面的测试代码可以发现，一个number类型的数据不可能借用到这个方法：

```
var a = 1;
Array.prototype.push.call(a,'first');
alert(a.length)  // undefined
alert(a[0]) //undefined
```

对于第二个条件，函数的length属性就是只读的，表示形参的个数，我们尝试把一个函数当做this传入Array.prototype.push：

```
var fn = function (){};
Array.prototype.push.call(fn,'first'); //报错
alert(fn.length);
```