第三课 高级函数

学习目录

- **惰性函数**
- **高阶函数**
- **函数柯里化**

一．惰性函数

在开发功能时，有时候会有多个比较庞大的 if 条件语句组成的庞大第三方函数，比如在封装不同浏览器的事件处理函数时，在封装不同浏览器的上传接口时，在封装 ajax 请求时等这种只需要判断 1 遍就可以知道该应用哪种条件分支处理方式的情况，就适合惰性载入函数。

使用惰性载入函数，可以让条件语句知道哪个分支适合当前环境之后，在下次进入这个第三方函数时就不在进行判断，而是直接使用对应的条件语句分支即可。

惰性载入意思是函数执行的分支只会在函数第一次掉用的时候执行，在第一次调用过程中，当前这个函数会被覆盖为另一个按照合适方式执行的函数，从而保证对原函数的调用就不用经过整体执行的条件语句分支了。

1.普通事件绑定

```
function addEventFn(eventType, element, fn) {
    if (element.addEventListener) {
        element.addEventListener(eventType, fn, false);
```

```
    }
    else if(element.attachEvent){
        element.attachEvent('on' + eventType, fn);
    }
    else{
        element['on' + eventType] = fn;
    }
}
```

2.惰性载入函数事件绑定方式一

```
function addEventFn(eventType, element, fn) {
    if (element.addEventListener) {
        addEventFn = function (eventType, element, fn) {
            element.addEventListener(eventType, fn, false);
        }
    }
    else if(element.attachEvent){
        addEventFn = function (eventType, element, fn) {
            element.attachEvent('on' + eventType, fn);
        }
    }
    else{
```

```
        addEventFn = function (eventType, element, fn) {
            element['on' + eventType] = fn;
        }
    }
    return addEventFn(eventType, element, fn);
}
```

3.惰性载入函数事件绑定方式二

```
var addEvent = (function () {
    if (document.addEventListener) {
        return function (eventType, element, fn) {
            element.addEventListener(eventType, fn, false);
        }
    }
    else if (document.attachEvent) {
        return function (eventType, element, fn) {
            element.attachEvent('on' + eventType, fn);
        }
    }
    else {
        return function (eventType, element, fn) {
            element['on' + eventType] = fn;
```

```
        }
    }
})();
```

第三种方式与第二种方式相比是在声明函数时先指定一个函数，从而减少函数在第一次执行时带来的性能损耗。

二．高阶函数

当一个函数可以接收另外一个函数作为参数的时候，这个函数被称为高阶函数。

比较好的体现是 map、filter 等数组方法。

当一个函数在内部返回一个函数的时候，这个函数被称为高阶函数。

比较好的体现是 bind 方法。

高阶函数的好处是可以抽离出业务逻辑代码中不容易变化和容易变化的代码，把容易变化的代码抽离出来放在作为参数的函数中。

1.map 可以理解主要用来修改遍历数据，方法的第一个参数接受一个回到函数，该回调函数又接受三个参数（正在遍历的当前元素，该元素的下标，数组），第二个参数是用来绑定回调函数中的 this，返回值为新数组。

```
var mapArr = arr.map((value,index,arr) => {
    return value + '666666';
},this);
console.log(mapArr);//["1666666", "2666666", "3666666", "4666666", "5666666", "6666666"]
```

2.filter 可以理解主要用来过滤删除数据，根据设定的条件来过滤掉不要的元素，参数形式和 map 一样，返回值为新数组。

```
var filterArr = arr.filter((value,index,arr) => {
    if(value > 2){
        return true;
    }
},this);
console.log(filterArr);//[3, 4, 5, 6]
```

3.bind 方法

bind 方法用来构建一个新的函数，调用 bind()方法不会执行原函数，也不会改变原函数本身，只会返回一个已经修改了 this 指向的新函数，bind 实参在其传入参数的基础上往后获取参数执行。

```
function fn(a,b,c){
    console.log(a,b,c);
}
var fn1 = fn.bind({abc : 123},600);
fn(100,200,300)  //这里会输出--> 100,200,300
fn1(100,200,300)  //这里会输出--> 600,100,200
```

fn 调用时函数内 this 指向 window，fn1 调用时函数内 this 指向{abc : 123}。

bind 方法的内部实现简单介绍，bind 方法核心本质是使用函数作为返回值来修改 this

指向操作。

```
Function.prototype.testBind = function(context) {
  var fn = this;//保留原函数引用
  var args = Array.prototype.slice.call(arguments);//保留原函数参数
  return function newFn() {
    var newArgs = args.concat(Array.prototype.slice.call(arguments));//合并原函数参数与新函数参数
    return fn.apply(context, newArgs);//返回一个新函数，并在调用新函数时通过 apply 绑定目标 this
  }
}
```

三．函数柯里化

函数柯里化是高阶函数的进一步应用，简单点说就是先给原始函数传入几个参数，让它先生成一个新的函数，然后让新的函数去处理接下来的其他参数。

```
function sum(a, b) {
    return a + b
}
function firstSum(a) {
    return function (b) {
        return a + b
    }
```

```
}

sum(100, 500)//600

var newFn1 = firstSum(100);

newFn1(500);//600
```

这么做的好处主要有动态新建函数、延迟执行、参数复用等等。

1.动态新建函数

其实有点类似于第一个知识点提到的惰性载入函数方式二的内容，当发现条件语句分支的某种条件符合要求时返回一个新的函数即可。

2.延迟执行

其实 bind 方法的实现就是利用了函数柯里化来做到延迟执行新函数。

3.参数复用

当多次调用一个函数，并且传递的参数绝大多数情况下是相同的，那么该函数可以考虑使用函数柯里化来封装。

```
function curry(fn,args) {

    var args = args || [];//保存递归调用动态参数集合

    return function () {

        var _args = args.concat(Array.prototype.slice.apply(arguments));//
合并递归调用柯里化目标函数得到的新的函数中的参数集合
```

```
            if(_args.length < fn.length) {
                return curry.call(this, fn, _args);//当传入的参数集合小于目标函数的参数数量时，继续递归调用，把参数传递给下一个柯里化函数，继续得到新的函数
            }else{
                return fn.apply(this, _args);//当传入的参数集合大于目标函数的参数数量时，调用新的函数
            }
        }
    }
    function addNum(a,b,c){//目标函数
        return a * b * c;
    }
    var curryAddFn = curry(addNum);//柯里化目标函数得到新的函数
    var newAddNum = curryAddFn(10)(1);//复用前两个参数得到新的函数
    console.log(newAddNum(3));//在复用参数的基础上传递第三个参数调用新函数
    console.log(newAddNum(6));//在复用参数的基础上传递第三个参数调用新函数
    console.log(newAddNum(9));//在复用参数的基础上传递第三个参数调用新函数
```

谢谢观看！

我是星星课堂老师：周小周