03 闭包与高阶函数

更新时间：2019-06-24 20:02:44

> 我要扼住命运的咽喉，它妄想使我屈服，这绝对办不到。生活是这样美好，活他一千辈子吧！
>
> ——贝多芬

「JavaScript 中，函数是一等公民」，在各种书籍和文章中我们总能看到这句话。

既然有一等，那么当然也有次等了。

> 如果公民分等级，一等公民什么都可以做，次等公民这不能做那不能做。
> JavaScript的函数也是对象，可以有属性，可以赋值给一个变量，可以放在数组里作为元素，可以作为其他对象的属性，什么都可以做，别的对象能做的它能做，别的对象不能做的它也能做。这不就是一等公民的地位嘛。 — 程墨Morgan

所以它的含义是：函数和其他普通对象一样，其上有属性也有方法，普通对象能做的，函数也可以做。

正因为在 JavaScript 中的极大自由，函数被赋予了卓越的表达力和灵活性，但是也产生了很多让人抓耳挠腮的问题。本文我们就一起讨论一下最常遇见的两个与函数密切相关的概念：闭包和高阶函数。这两个概念在之后设计模式的文章中也会经常碰见。

> **注意：** 本文属于基础篇，如果你已经对本文相关知识点已经很了解了，那么可以跳过本文。如果你不够了解，或者了解的还不完整，那么可以通过本文来复习一下 ~

1. 闭包

1.1 什么是闭包

当函数可以**记住并访问**所在的词法作用域时，就产生了**闭包**，即使函数是在当前词法作用域之外执行。

我们首先来看一个闭包的例子：

```
function foo() {
    var a = 2

    function bar() {
        console.log(a)
    }

    return bar
}

var baz = foo()

baz()           // 输出: 2
```

`foo` 函数传递出了一个函数 `bar`，传递出来的 `bar` 被赋值给 `baz` 并调用，虽然这时 `baz` 是在 `foo` 作用域外执行的，但 `baz` 在调用的时候可以访问到前面的 `bar` 函数所在的 `foo` 的内部作用域。

由于 `bar` 声明在 `foo` 函数内部，`bar` 拥有涵盖 `foo` 内部作用域的闭包，使得 `foo` 的内部作用域一直存活不被回收。一般来说，函数在执行完后其整个内部作用域都会被销毁，因为 JavaScript 的 GC（Garbage Collection）垃圾回收机制会自动回收不再使用的内存空间。但是闭包会阻止某些 GC，比如本例中 `foo()` 执行完，因为返回的 `bar` 函数依然持有其所在作用域的引用，所以其内部作用域不会被回收。

注意： 如果不是必须使用闭包，那么尽量避免创建它，因为闭包在处理速度和内存消耗方面对性能具有负面影响。

1.2 利用闭包实现结果缓存（备忘模式）

备忘模式就是应用闭包的特点的一个典型应用。比如有个函数：

```
function add(a) {
    return a + 1;
}
```

多次运行 `add()` 时，每次得到的结果都是重新计算得到的，如果是开销很大的计算操作的话就比较消耗性能了，这里可以对已经计算过的输入做一个缓存。

所以这里可以利用闭包的特点来实现一个简单的缓存，在函数内部用一个对象存储输入的参数，如果下次再输入相同的参数，那就比较一下对象的属性，如果有缓存，就直接把值从这个对象里面取出来。

```
/* 备忘函数 */
function memorize(fn) {
    var cache = {}
    return function() {
        var args = Array.prototype.slice.call(arguments)
        var key = JSON.stringify(args)
        return cache[key] || (cache[key] = fn.apply(fn, args))
    }
}

/* 复杂计算函数 */
function add(a) {
    return a + 1
}

var adder = memorize(add)

adder(1)            // 输出: 2    当前: cache: { '[1]': 2 }
adder(1)            // 输出: 2    当前: cache: { '[1]': 2 }
adder(2)            // 输出: 3    当前: cache: { '[1]': 2, '[2]': 3 }
```

使用 ES6 的方式会更优雅一些：

```
/* 备忘函数 */
function memorize(fn) {
    const cache = {}
    return function(...args) {
        const key = JSON.stringify(args)
        return cache[key] || (cache[key] = fn.apply(fn, args))
    }
}

/* 复杂计算函数 */
function add(a) {
    return a + 1
}

const adder = memorize(add)

adder(1)            // 输出: 2    当前: cache: { '[1]': 2 }
adder(1)            // 输出: 2    当前: cache: { '[1]': 2 }
adder(2)            // 输出: 3    当前: cache: { '[1]': 2, '[2]': 3 }
```


稍微解释一下：

备忘函数中用 `JSON.stringify` 把传给 `adder` 函数的参数序列化成字符串，把它当做 `cache` 的索引，将 `add` 函数运行的结果当做索引的值传递给 `cache`，这样 `adder` 运行的时候如果传递的参数之前传递过，那么就返回缓存好的计算结果，不用再计算了，如果传递的参数没计算过，则计算并缓存 `fn.apply(fn, args)`，再返回计算的结果。

当然这里的实现如果要实际应用的话，还需要继续改进一下，比如：

1. 缓存不可以永远扩张下去，这样太耗费内存资源，我们可以只缓存最新传入的 n 个；
2. 在浏览器中使用的时候，我们可以借助浏览器的持久化手段，来进行缓存的持久化，比如 cookie、localStorage 等；

这里的复杂计算函数可以是过去的某个状态，比如对某个目标的操作，这样把过去的状态缓存起来，方便地进行状态回退。

复杂计算函数也可以是一个返回时间比较慢的异步操作，这样如果把结果缓存起来，下次就可以直接从本地获取，而不是重新进行异步请求。

注意： cache 不可以是 `Map`，因为 `Map` 的键是使用 `===` 比较的，因此当传入引用类型值作为键时，虽然它们看上去是相等的，但实际并不是，比如 `[1]!==[1]`，所以还是被存为不同的键。

```
//  X 错误示范
function memorize(fn) {
  const cache = new Map()
  return function(...args) {
    return cache.get(args) || cache.set(args, fn.apply(fn, args)).get(args)
  }
}

function add(a) {
  return a + 1
}

const adder = memorize(add)

adder(1)    // 2    cache: { [ 1 ] => 2 }
adder(1)    // 2    cache: { [ 1 ] => 2, [ 1 ] => 2 }
adder(2)    // 3    cache: { [ 1 ] => 2, [ 1 ] => 2, [ 2 ] => 3 }
```

2. 高阶函数

高阶函数就是输入参数里有函数，或者输出是函数的函数。

2.1 函数作为参数

如果你用过 `setTimeout`、`setInterval`、ajax 请求，那么你已经用过高阶函数了，这是我们最常看到的场景：**回调函数**，因为它将函数作为参数传递给另一个函数。

比如 ajax 请求中，我们通常使用回调函数来定义请求成功或者失败时的操作逻辑：

```
$.ajax("/request/url", function(result){
    console.log("请求成功！")
})
```

在 Array、Object、String 等等基本对象的原型上有很多操作方法，可以接受回调函数来方便地进行对象操作。这里举一个很常用的 `Array.prototype.filter()` 方法，这个方法返回一个新创建的数组，包含所有回调函数执行后返回 `true` 或真值的数组元素。

```
var words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];

var result = words.filter(function(word) {
    return word.length > 6
})       // 输出: ["exuberant", "destruction", "present"]
```

回调函数还有一个应用就是钩子，如果你用过 Vue 或者 React 等框架，那么你应该对钩子很熟悉了，它的形式是这样的：

```
function foo(callback) {
    // ... 一些操作
    callback()
}
```

2.2 函数作为返回值

另一个经常看到的高阶函数的场景是在一个函数内部输出另一个函数，比如：

```
function foo() {
    return function bar() {}
}
```

主要是利用闭包来保持着作用域：

```
function add() {
    var num = 0
    return function(a) {
        return num = num + a
    }
}
var adder = add()

adder(1)     // 输出: 1
adder(2)     // 输出: 3
```

1. 柯里化

柯里化（Currying），又称部分求值（Partial　Evaluation），是把接受多个参数的原函数变换成接受一个单一参数（原函数的第一个参数）的函数，并且返回一个新函数，新函数能够接受余下的参数，最后返回同原函数一样的结果。

核心思想是把多参数传入的函数拆成单（或部分）参数函数，内部再返回调用下一个单（或部分）参数函数，依次处理剩余的参数。

柯里化有 3 个常见作用：

1. 参数复用
2. 提前返回
3. 延迟计算/运行

先来看看柯里化的通用实现：

```
// ES5 方式
function currying(fn) {
    var rest1 = Array.prototype.slice.call(arguments)
    rest1.shift()
    return function() {
        var rest2 = Array.prototype.slice.call(arguments)
        return fn.apply(null, rest1.concat(rest2))
    }
}

// ES6 方式
function currying(fn, ...rest1) {
  return function(...rest2) {
    return fn.apply(null, rest1.concat(rest2))
  }
}
```

用它将一个 `sayHello` 函数柯里化试试：

```
// 接上面
function sayHello(name, age, fruit) {
  console.log(console.log(`我叫 ${name},我 ${age} 岁了, 我喜欢吃 ${fruit}`))
}

var curryingShowMsg1 = currying(sayHello, '小明')
curryingShowMsg1(22, '苹果')           // 输出: 我叫 小明,我 22 岁了, 我喜欢吃 苹果

var curryingShowMsg2 = currying(sayHello, '小衰', 20)
curryingShowMsg2('西瓜')               // 输出: 我叫 小衰,我 20 岁了, 我喜欢吃 西瓜
```

更高阶的用法参见：JavaScript 函数式编程技巧 - 柯里化

2. 反柯里化

柯里化是固定部分参数，返回一个接受剩余参数的函数，也称为部分计算函数，目的是为了缩小适用范围，创建一个针对性更强的函数。核心思想是把多参数传入的函数拆成单参数（或部分）函数，内部再返回调用下一个单参数（或部分）函数，依次处理剩余的参数。

而**反柯里化**，从字面讲，意义和用法跟函数柯里化相比正好相反，扩大适用范围，创建一个应用范围更广的函数。使本来只有特定对象才适用的方法，扩展到更多的对象。

先来看看反柯里化的通用实现吧~

```
// ES5 方式
Function.prototype.unCurrying = function() {
  var self = this
  return function() {
    var rest = Array.prototype.slice.call(arguments)
    return Function.prototype.call.apply(self, rest)
  }
}

// ES6 方式
Function.prototype.unCurrying = function() {
  const self = this
  return function(...rest) {
    return Function.prototype.call.apply(self, rest)
  }
}
```


如果你觉得把函数放在 Function 的原型上不太好，也可以这样：

```
// ES5 方式
function unCurrying(fn) {
  return function (tar) {
    var rest = Array.prototype.slice.call(arguments)
    rest.shift()
    return fn.apply(tar, rest)
  }
}

// ES6 方式
function unCurrying(fn) {
  return function(tar, ...argu) {
    return fn.apply(tar, argu)
  }
}
```

下面简单试用一下反柯里化通用实现，我们将 Array 上的 `push` 方法借出来给 arguments 这样的类数组增加一个元素：

```
// 接上面
var push = unCurrying(Array.prototype.push)

function execPush() {
  push(arguments, 4)
  console.log(arguments)
}

execPush(1, 2, 3)    // 输出: [1, 2, 3, 4]
```

简单说，函数柯里化就是对高阶函数的降阶处理，缩小适用范围，创建一个针对性更强的函数。

```
function(arg1, arg2)             // => function(arg1)(arg2)
function(arg1, arg2, arg3)       // => function(arg1)(arg2)(arg3)
function(arg1, arg2, arg3, arg4) // => function(arg1)(arg2)(arg3)(arg4)
function(arg1, arg2, ..., argn)  // => function(arg1)(arg2)…(argn)
```

而反柯里化就是反过来，增加适用范围，让方法使用场景更大。使用反柯里化， 可以把原生方法借出来，让任何对象拥有原生对象的方法。

```
obj.func(arg1, arg2)        // => func(obj, arg1, arg2)
```

可以这样理解柯里化和反柯里化的区别：

1. 柯里化是在运算前提前传参，可以传递多个参数；
2. 反柯里化是延迟传参，在运算时把原来已经固定的参数或者 this 上下文等当作参数延迟到未来传递。

更高阶的用法参见：JavaScript 函数式编程技巧 - 反柯里化

3. 偏函数

偏函数是创建一个调用另外一个部分（参数或变量已预制的函数）的函数，函数可以根据传入的参数来生成一个真正执行的函数。其本身不包括我们真正需要的逻辑代码，只是根据传入的参数返回其他的函数，返回的函数中才有真正的处理逻辑比如：

```
var isType = function(type) {
  return function(obj) {
    return Object.prototype.toString.call(obj) === `[object ${type}]`
  }
}

var isString = isType('String')
var isFunction = isType('Function')
```

这样就用偏函数快速创建了一组判断对象类型的方法~

偏函数和柯里化的区别：

1. **柯里化**是把一个接受 n 个参数的函数，由原本的一次性传递所有参数并执行变成了可以分多次接受参数再执行，例如：`add = (x, y, z) => x + y + z`→`curryAdd = x => y => z => x + y + z`；
2. **偏函数**固定了函数的某个部分，通过传入的参数或者方法返回一个新的函数来接受剩余的参数，数量可能是一个也可能是多个；

当一个柯里化函数只接受两次参数时，比如 `curry()()`，这时的柯里化函数和偏函数概念类似，可以认为偏函数是柯里化函数的退化版。