JavaScript 柯里化

1. （掌握）柯里化的定义

1. 先看一下维基百科对柯里化的解释：

    1. 在计算机科学中，柯里化(英语：Curring)，又译为卡瑞化或加里化
    2. 是把接收多个参数的函数，变成接收一个单一参数(最初函数的第一个参数)的函数，并且返回接受余下的数，而且返回的结果的新函数的技术
    3. 柯里化声称，如果你固定某些参数，你将得到接受余下参数的一个函数

2. 总结：只传递给函数一部分参数来调用它，让它返回一个函数去处理剩余的参数，这个过程就称之为柯里化

```
// 我们有一个函数，可以接收四个参数
function foo(m, n, x, y){
     return m + n + x + y
}
// 调用的时候一次性传入四个参数
foo(1, 2, 3, 4)

// 现在将这个函数做一些改变
function bar(m){
    // 先只接收第一个参数，然后返回一个函数
    return function(n){
        // 依次类推
        return function(x){
            return function(y){
                return m + n + x + y
            }
        }
    }
}
// 这里也先只传入一个参数，因为返回的是一个函数，所以可以继续调用传参
bar(1)(2)(3)(4)

// 将第一种函数变成第二种函数的过程就是柯里化
```

2. （理解）柯里化的过程和结构

1. 柯里化的过程：

```
```

```
// 正常函数的书写
function add(x, y, z) {
  return x + y + z
}

var result = add(1, 2, 3)
console.log(result)

// 现在我们将上述的函数转为柯里化
function sum(x) {
  return function (y) {
    return function (z) {
      return x + y + z
    }
  }
}
var result = sum(1)(2)(3)
console.log(result)

// 简化柯里化函数
var sum2 = x => y => z => x + y + z
console.log(sum2(1)(2)(3))
/*
  上述简化过程复杂一点就是
  var sum2 = x =>{
    return y =>{
      return z =>{
        return x + y + z
      }
    }
  }
  // 箭头函数只有一句执行代码的时候，可以省略花括号，如果有 return 也可以省略
*/
```

2. 注意，不一定是要每一个返回的函数都只传入一个参数，只要是将原来的函数拆分成功多函数的方法，这个 转换的过程 叫做柯里化的过程，而不是结果

3. （掌握）为什么需要柯里化

1. 为什么需要柯里化
    1. 在函数式编程中，我们其实往往希望 一个函数处理的问题尽可能的单一 ，而 不是将一大堆的处理过程交给一个函数来处理
    2. 那么 我们是否就可以将每次传入的参数在单一的函数中进行处理 。处理完后在 下一个函数中在使用处理后的结果

2. 柯里化-执行单一逻辑的原则：

```
function add(x, y, z) {
  return x + y + z
}
// 现在对这个案例做出一个修改：传入的函数需要分别被进行如下处理
/*
  第一个参数 + 2、第二个参数 * 2、第三个参数 ** 2
*/
// 现在通过柯里化来实现一下
var foo = x => {
  x = x + 2
  return y => {
    y = y * 2
    return z => {
      z = z * z
      return x + y + z
    }
  }
}
console.log(foo(1)(2)(3))
```

3. 柯里化-逻辑的复用：

```
// 下面我们有一段这样的需求，需要每次都传入一个参数和 5 相加
function add(num1, num2) {
  return num1 + num2
}
// 那我们在调用的时候，就是如下的情况
// console.log(add(5, 10))
// console.log(add(5, 100))
// console.log(add(5, 1000))

// 用柯里化转换一下看看
function foo(num1) {
  return function (num2) {
    return num1 + num2

  }
}

// console.log(foo(5)(10))
// console.log(foo(5)(100))
// console.log(foo(5)(1000))
// 这样看好像也陷入了一样的问题，这个重复的传入的 5 还是没有解决
// 所以我们可以换上一种方式输出
var res = foo(5)
console.log(res(10))
console.log(res(100))
console.log(res(1000))
```

```
```

```
function log(data, type, message) {
  console.log(`[${data.getHours()}:${data.getMinutes()}][${type}][${message}]`)
}

// log(new Date(), 'debug', '轮播图的bug')
// log(new Date(), 'debug', '查询菜单的bug')
// log(new Date(), 'debug', '查询数据的bug')

// 使用柯里化复用逻辑
var log1 = data => type => message => {
  console.log(`[${data.getHours()}:${data.getMinutes()}][${type}][${message}]`)
}

var res = log1(new Date)
res('debug')('轮播图的bug')
res('feture')('查询菜单的bug')

var res = log1(new Date)('debug')
res('轮播图的bug')
res('查询菜单的bug')
res('查询数据的bug')
```

4. （掌握）柯里化的实现

```
function log(data, type, message) {
  return `[${data.getHours()}:${data.getMinutes()}][${type}][${message}]`
}

function add(num1, num2) {
  return num1 + num2
}

function add1(x, y, z) {
  console.log(x + y + z)
  return x + y + z
}

function foo(x, y, z) {
  x = x + 2
  y = y * 2
  z = z * z
  return x + y + z
}

// 之前我们将上述函数使用柯里化都是手动实现
// 所以我们现在希望编写一个函数
// 这个函数在传入一个函数之后,就会返回一个函数,这个返回的函数就是生产后柯里化后的函数
function jcCurring(fn) {
  // 判断当前已经接收的参数的个数，和参数本身需要的接收的参数是否一致了
```

```
  // 检测 ...args 里面的参数和 fn传进来的参数是否一致
  // ...args 参数个数获取长度即可，fu怎么获取呢，直接函数名.length即可
  function curried1(...args) {
    if (args.length >= fn.length) {
      // fn(...args)
      // 直接调用的话会产生 this 问题，比如返回之后 外层调用的时候是
res.call('aaa',10,20)
      // 所以为了保证this不发生改变，在内部指定this
      // 本来是在全局调用，如果外部使用了其他方式改变，那就是不是原来的this了
      return fn.apply(this, args)
    } else {
      // 没有达到个数时,需要返回一个新的函数,继续来接收新的参数
      function curried2(...args2) {
        // 接收到参数后，需要递归调用 curried1来检测个数是否达到
        // 这里需要的传入this和上一层判断是一样的
        return curried1.apply(this, args.concat(args2))
      }
      return curried2
    }
  }
  return curried1
}

// 传入需要执行的函数
var res = jcCurring(add1)
console.log(res(1, 2)(3))

var res = jcCurring(foo)
console.log(res(1)(2)(3))

var res = jcCurring(log)
console.log(res(new Date)('debug')('查询数据错误'))
```