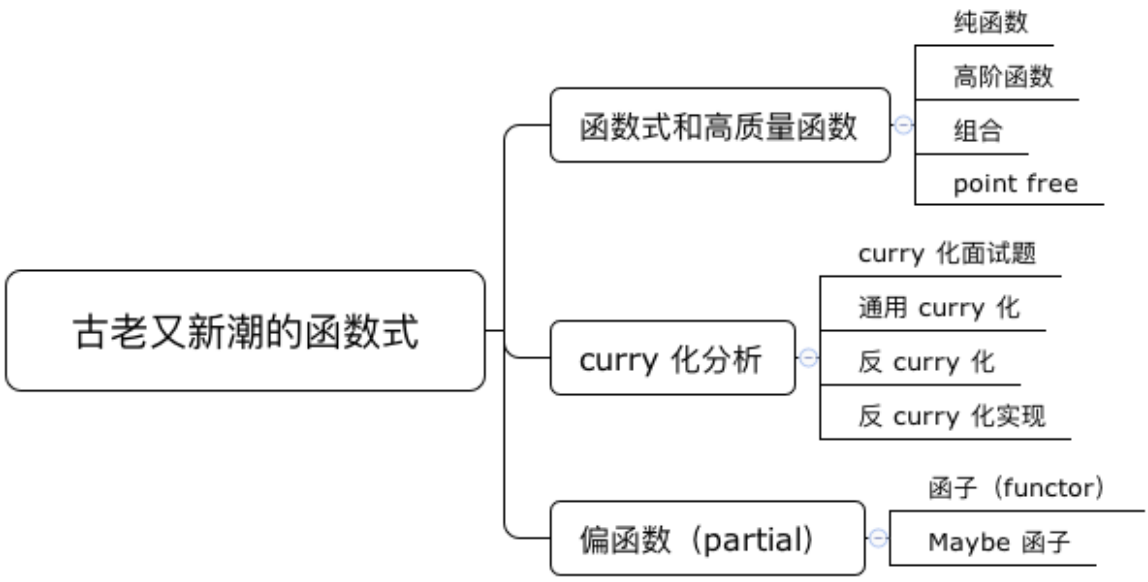
古老又新潮的函数式

函数式这个概念我们在以往的课程中其实已经有所涉及了，比如第 1-2、1-3 课介绍的闭包知识；比如 1-4 课《我们不背诵 API，只实现 API》中剖析的 5 种 compose 方法，以及 reduce 实现 pipe、runPromiseInSequence 等都是典型的函数式概念。

函数式其实很早就出现在了编程领域当中，近些年由于 React 的带动，在前端开发中重新「焕发活力」。

很多读者可能一听到函数式就眉头一皱，毕竟相比于面向对象等其他编程概念，它更加晦涩难懂。对于函数式的学习，也一定不是使用或者模仿 compose 那么简单。这一节课，我们就来梳理几个函数式概念。但是我认为单纯的概念介绍并没有意义，因此也准备了大量实战例题以及库的设计方案，与大家一起分享。



函数式和高质量函数

函数式通常意味着高质量的代码，本小节的主标题之所以是「函数式和高质量函数」，而不是「函数式和高质量代码」，因为在函数式看来，一切都是函数，「函数是第一等公民」。围绕着函数，取代面向过程式的代码，往往能够有以下收益：

表达力更加清晰，因为「一切都是函数」，通过函数的合理命名，函数原子的拆分，我们能够一眼看出来程序在做什么，以及做的过程；

利于复用，因为「一切都是函数」，函数本身具有天然的复用能力；

利于维护，纯函数和幂等性保证同样的输入就有同样的输出，在维护或者调试代码时，能够更加专注，减少因为共享带来的潜在问题。

我们下面来通过概念具体展开。

纯函数

之前我们提到过：

一个函数如果输入参数确定，输出结果是唯一确定的，那么它就是纯函数。

并且纯函数不能修改外部变量，造成副作用，不能调用 Math.radom() 方法以及发送异步请求等，因为这些操作都不具有确定性。

根据定义我们知道纯函数的特点是：

无状态

无副作用

无关时序

幂等（指无论调用多少次，结果相同）

看代码举例：

```
let array = [1,2,3,4]

// array 的 slice 方法属于纯函数方法，它不对数组本身进行操作
// array 的 splice 方法不属于纯函数方法，它对数组本身进行操作

const minusCount = () => {
   window.count--
}

// minusCount 不是纯函数，它依赖并改变外部变量，具有副作用

const setHtml = (node, html) => {
   node.innerHtml = html
}

// setHtml 不是纯函数，同上
```

这样的纯函数不仅易于维护，逻辑清晰，而且具有更好的组合和测试性。之前的课程中我们多次提到，这里不再单独展开。

高阶函数

高阶函数体现了「函数是第一等公民」，它是指这样的一类函数：该函数接受一个函数作为参数，返回另外一个函数。

没错，和高阶组件的概念类似。为什么会有这么一个「怪异」的高阶函数呢？来看一个例子：filterLowerThan10 这个函数接受一个数组作为参数，它会挑选出数组中数值小于 10 的项目，所有符合条件的值都会构成新数组被返回：

```
const filterLowerThan10 = array => {
   let result = []
   for (let i = 0, length = array.length; i < length; i++)
{
        let currentValue = array[i]
        if (currentValue < 10) result.push(currentValue)
```

```
    }
    return result
}
```

另外一个需求，挑选出数组中非数值项目，所有符合条件的值都会构成新数组被返回：

```
const filterNaN = array => {
    let result = []
    for (let i = 0, length = array.length; i < length; i++)
{
        let currentValue = array[i]
        if (isNaN(currentValue)) result.push(currentValue)
    }
    return result
}
```

这都是很基本的面向过程编程的代码。不够优雅的一点是 filterLowerThan10 和 filterNaN 都有遍历的逻辑，都存在了重复的 for 循环。本质上都是遍历一个列表，并用给定的条件过滤列表。我们能否用函数式的思想，将遍历和筛选解耦呢?

好在 JavaScript 对函数式较为友好，我们使用 filter 函数来完成，并进行一定程度的改造：

```
const lowerThan10 = value => value < 10

[12, 3, 4, 89].filter(lowerThan10)

[12, 'sd', null, undefined, {}].filter(isNaN)
```

这非常简单，我们以此来热身进入状态。

另一个高阶函数的典型应用场景是函数缓存：

```
const memorize = fn => {
   let cacheMap = {}
   return function(...args) {
       const cacheKey = args.join('_')
       if (cacheKey in cacheMap) {
           return cache[cacheKey]
       }
       else {
           return cacheMap[cacheKey] = fn.apply(this ||
{}, args)
       }
   }
}
```

高阶函数可以和 decorator 相结合，再来看一个实例，实现有限次数函数调用的装饰器：

```
 class MyClass {
   @callLimit getSum() {}
}
```

实现：

```
function callLimit(limitCallCount = 1, level = 'warn') {
   // 记录调用次数
   let count = 0
   return function(target, name, descriptor) {
       // 记录原始函数
       var fn = descriptor.value
       // 改写新函数
       descriptor.value = function(...args) {
           if (count < limitCallCount) {
               count++
               return fn.apply(this || {}, args)
           }
           if (console[level]) console[level](name, 'call
```

```
limit')
            console.warn(name, 'call limit')
        }
    }
}
```

严格来说，这也不算是一个高阶函数的使用场景，但是体现了类似的思想。读者可以举一反三。

组合

继续延伸我们的场景，如果输入比较复杂，想先过滤出小于 10 的项目需要先保证数组中每一项都是 Number 类型，那么可以：

```
[12, 'sd', null, undefined, {}, 23, 45, 3,
6].filter(value=> !isNaN(value) && value !==
null).filter(lowerThan10)s
```

这样的做法得益于 JavaScript filter 对函数式的友好支持，链式调用也在一定程度上实现了组合性。

更加通用的组合做法是使用 compose 方法，收益非常直观：

单一功能的小函数更好维护

通过组合，将单一功能的小函数串联起来，完成复杂的功能

复用性更好，硬编码更少

point free

point free 是指一种函数式的编程风格，有时候也可以叫做 tacit programming。point 在这里的意思是指形参，那么 point free，自然就是指没有行参了。这样做的目的是什么呢？没有参数，就意味着我们将注意力放在函数本身上。一般参数存在的意义是传递或者携带某个值，函数根据这个值，来得到

另一个值。这样造成的困扰是我们不得不操作数据，同时要给参数命名。如果没有参数，不返回一个数据，那么 point free 的目的就是得到一个函数。

当然业务中不可能永远不存在参数，因此我们允许底层函数非 point free，而 point free 函数更像是一种上层封装，它灵活调度带有参数的的底层函数，通过 point free 和非 point free 的解耦，使得代码更具有声明式特征，更具有美感。

point free 是我们的追求，而非标准，过度使用某种模式往往让代码「变坏」，这里给大家介绍这种概念，「见多识广」和「矫枉过正」往往只有一线之隔。

curry 化分析

curry 化也是一个常见的概念，维基百科对齐解释为：

在计算机科学中，柯里化（currying），又译为卡瑞化或加里化，是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数而且返回结果的新函数的技术。这个技术由克里斯托弗·斯特雷奇以逻辑学家哈斯凯尔·加里命名的。

简单来说，就是在一个函数中预先填充几个参数，这个函数返回另一个函数，这个返回的新函数将其参数和预先填充的参数进行合并，再执行函数逻辑。

那我们上述代码举例：

```
const filterLowerThan10 = array => {
   let result = []
   for (let i = 0, length = array.length; i < length; i++)
{
       let currentValue = array[i]
       if (currentValue < 10) result.push(currentValue)
   }
   return result
}
```

filterLowerThan10 还是硬编码写死了 10 这个阈值，我们用 curry 化的思想将其改造：

```
const filterLowerNumber = number => {
   return array => {
       let result = []
       for (let i = 0, length = array.length; i < length;
i++) {
           let currentValue = array[i]
           if (currentValue < number)
result.push(currentValue)
       }
       return result
   }
}

const filterLowerThan10 = filterLowerNumber(10)


filterLowerThan10([1, 3, 5, 29, 34])
```

curry 化面试题

再通过一到面试题加深理解。

实现 add 方法，要求：

```
add(1)(2) == 3 // true

add(1)(2)(3) == 6 // true
```

分析这道题：add 函数每次执行后一定需要保证返回一个函数，以供后续继续调用，且返回的这个函数还有返回自身，以支持连续调用。同时，为了满足例题条件，需要改写内部返回的函数 toString：

```
const add = arg1 => {
  const fn = arg2 => {
       return fn
   }
   fn.toString = function () {

   }
   return fn

}
```

为了进行「求和」操作，需要在 add 函数内部维护一个闭包变量 args，args 是个数组，记录了每次调用是传进来的参数，toString 方法体中对参数进行求和，fn 方法体中对数组 args 进行添加当前参数的操作：

```
const add = arg1 => {
  let args = [arg1]
  const fn = arg2 => {
      args.push(arg2)
       return fn
   }
   fn.toString = function () {
       return args.reduce((prev, item) => prev + item, 0)
   }
   return fn
}
```

注意这里只支持：

```
add(1)(2)(3)
```

单个参数的调用，如果更加通用化，支持：

```
add(1)(2, 3)(4)
```

需要我们改动为：

```
const add = (...arg1) => {
  let args = [...arg1]
  const fn = (...arg2) => {
      args = [...args, ...arg2]
       return fn
   }
   fn.toString = function () {
       return args.reduce((prev, item) => prev + item, 0)
   }
   return fn
}
```

实现：

```
add(1)(2, 3)(4) == 10
```

还有一个细节，如果将 == 改为 ===，将会输出 false，这并不奇怪。因为 add 调用后的返回值类型始终为 Function，我们只是改写了其 toString 方法，利用了隐式转换规则而已。

通用 curry 化

我们回到 filterLowerThan10 函数的案例中，从中感受到 curry 化的优势：

- 提高复用性
- 减少重复传递不必要的参数
- 动态根据上下文创建函数

其中动态根据上下文创建函数，也是一种惰性求值的体现：

```
const addEvent = (function() {
   if (window.addEventListener) {
       return function (type, element, handler, capture) {
           element.addEventListener(type, handler,
```

```
capture)
        }
    }
    else if (window.attachEvent){
        return function (type, element, fn) {
            element.attachEvent('on' + type, fn)
        }
    }
})()
```

这是一个典型兼容 IE9 浏览器事件 API 的例子，根据兼容性的嗅探，充分利用 curry 化思想，完成了需求。

那么我们如何编写一个通用化的 curry 函数呢?

```
const curry = (fn, length) => {
    length = length || fn.length
    return function (...args) {
        if (args.length < length) {
            return curry(fn.bind(this, ...args), length -
args.length)
        }
        else {
            return fn.call(this, ...args)
        }
    }
}
```

这里我们利用 Function.length 获取函数预期需要的参数个数，并利用了 bind 方法绑定参数。

如果不想使用 bind，另一种常规思路是对每次调用时产生的参数进行存储：

```
const curry = fn => {
    return tempFn = (...arg1) => {
        if (arg1.length >= fn.length) {
```

```
            return fn(...arg1)
        }
        else {
            return (...arg2) => tempFn(...arg1, ...arg2)
        }
    }
}
```

简化为：

```
const curry = fn =>
    judge = (...arg1) =>
        arg1.length >= fn.length
            ? fn(...arg1)
            : (...arg2) => judge(...arg1, ...arg2)
```

总之，实现原理就是：先用闭包把传入参数保存起来，当传入参数的数量足够执行函数时，就开始执行函数。抽象成步骤：

先逐步接受参数，并进行存储，以供后续使用

先不进行函数计算，延后执行

在符合条件时，根据存储的参数，统一传给函数进行计算

反 curry 化

反 curry 化与 curry 正好相反。反 curry 化在于扩大函数的适用性，使本来作为特定对象所拥有的功能函数可以被任意对象所使用。

说到「特定对象所拥有的功能函数可以被任意对象所使用」，有经验的读者可能会想到用于类型判断的 Object.prototype.toString.call(target)：

```
const foo = () => ({})
const bar = ''
```

```
Object.prototype.toString.call(foo) === '[object
Function]'
// true

Object.prototype.toString.call(bar) === '[object String]'
// true
```

通过反 curry 化，我们将这个 Object 原型上的属性函数变得更加通用：

```
const toString = Object.prototype.toString.unCurry
```

或者有一个 UI 组件 Toast：

```
function Toast (options) {
   this.message = ''
}

Toast.prototype = {
   showMessage: function () {
       console.log(this.message)
   }
}
```

这样的代码，使得 Toast 实例均可使用 showMessage 方法：

```
new Toast({}).showMessage()
```

如果有一个变量对象：

```
const obj = {
   message: 'uncurry test'
}
```

如果想使用 Toast 原型上的 showMessage 方法：

```
const unCurryShowMessaage =
unCurry(Toast.prototype.showMessage)

unCurryShowMessaage(obj)
```

就是使用反 curry 化的另一个场景了。

反 curry 化实现

那么上述的 unCurry 方法应该如何实现呢?

我们来分析：unCurry 的参数是一个「希望被其他对象所调用的方法」，暂且称为 fn，unCurry 执行后返回一个新的函数，该函数的第一个参数是预期要执行方法的对象（obj），后面的参数是执行这个方法时需要传递的参数。

```
function unCurry(fn) {
   return function () {
       var obj = [].shift.call(arguments)
       return fn.apply(obj, arguments)
   }
}
```

改成 ES6 的写法：

```
const unCurry = fn => (...args) => fn.call(...args)
```

以上是正常函数实现 uncurry 的实现。我们也可以将 uncurry 挂载在函数原型上：

```
Function.prototype.unCurry = !Function.prototype.unCurry
|| function () {
   const self = this
   return function () {
       return Function.prototype.call.apply(self,
arguments)
```

```
    }
}
```

这里不太好理解的点在于：Function.prototype.call.apply(self, arguments)，其实这个问题本课程的读者群里也有人问过，我们就一起来讨论下，拆开看就会非常清晰了。

> 第一步：Function.prototype.call.apply(self, arguments) 可以看成 Fn.apply(self, arguments)，Fn 函数执行时，this 指向了 self。而根据代码，self 是调用 unCurry 的函数，执行结果就是 Fn(arguments)，只不过 this 被绑定在 self 上，用 callFn(arguments) 来表示。
>
> 第二步：callFn(arguments) 解析，callFn 指的是：Function.prototype.call，call 方法第一个参数是用来指定 this 的，因此 callFn(arguments) 相当于 callFn(arguments[0], arguments[n - 1])。

因此，最终执行就相当于：callFn(arguments[0], arguments[n - 1])，也就是说反 curry 化后得到的函数，第一个参数是用来决定 this 指向的，也就是需要应用的目标对象，剩下的参数是函数执行所需要的参数。

当然，我们可以借助 bind 实现：

```
Function.prototype.unCurry = function() {
 return this.call.bind(this)
}
```

借助 bind，call/apply 实现过程相对抽象，读者可以根据示例尝试理解。这里允许我再赘述一下：

call 中的 this 指的是调用它的函数，call 的内部实现中：第一个参数替换了这个函数中的 this，其余作为形参执行了函数。而我们的代码：Function.prototype.call.apply，使用 apply 之后，apply 的第一个参数更换了 call 中的 this。因此执行时，实际执行计算的的函数为 self。

这里再补充一个例子，供大家理解：

```
const push = Array.prototype.push.unCurry()

const test = { foo: 'lucas' }
push(test, 'messi', 'ronaldo', 'neymar')
console.log(test)

// {0: "messi", 1: "ronaldo", 2: "neymar", foo: "lucas",
length: 3}
```

我们借助了数组的 push 方法，应用在对象上，test 对象多了类似数组的属性，键为数组索引。

偏函数（partial）

如果了解了 curry 化，那么偏函数（partial application）就很容易理解了。如果说 curry 化是将一个多参数函数转换成多个单参数函数，也就是 curry 化将 n 原函数转换 n 个一元函数，那么偏函数就是固定一个函数的一个或者多个参数，即将一个 n 元函数转换成一个 （n － k） 元函数：

curry 化：n = n * 1

partial：n = n/k * k

响应偏函数实现：

```
const partial = (fn, ...rest) => (...args) => fn(...rest,
...args)
```

使用 bind 版本实现：

```
const partial = (fn, ...args) => fn.bind(null, ...args)
```

函子（functor）

说到函子，大部分没有深入过函数式编程的读者可能有点陌生，而函子确实一个很重要的函数式编程思想。目前社区上介绍的并不算多，我们这里来进行一下了解。

我想先从链式调用说起，看以下代码：

```
const addHelloPrefix = str => `Hello : ${str}`
const addByeSuffix = str => `${str}, bye!`
```

addHelloPrefix 和 addByeSuffix 分别给所接收到的字符串添加固定的字符串前缀和后缀，我们可以这样使用：

```
addByeSuffix(addHelloPrefix('lucas'))
```

得到返回结果："Hello : lucas, bye!"

如果我们想链式调用：

```
'lucas'.addHelloPrefix().addByeSuffix()

// VM176:1 Uncaught TypeError: "lucas".addHelloPrefix is
not a function
```

得到报错信息，是因为字符串并不存在 addHelloPrefix 方法，因此调用失败。如果 'lucas' 这样的字符串是一个复杂类型，或者是一个类，也许问题就能解决：

```
class Person {
   constructor(value) {
       this.value = value
   }
   addHelloPrefix() {
       return `Hello : ${this.value}`
   }
   addByeSuffix() {
       return `${this.value}, bye`
```

```
    }
}
```

这样的 Person 声明并不足以完成链式调用，链式调用的关键是 addHelloPrefix 和 addByeSuffix 方法仍然返回该类实例，而不是字符串。我们改动如下：

```
class Person {
    constructor(value) {
        this.value = value
    }
    addHelloPrefix() {
        return new Person(`Hello : ${this.value}`)
    }
    addByeSuffix() {
        return new Person(`${this.value}, bye`)
    }
}
```

执行代码：

```
new Person('lucas').addHelloPrefix().addByeSuffix()
```

输出：

```
{value: "Hello : lucas, bye"}
```

这样一来，似乎举例目标更近了些。我们试图将上述操作变得完全通用，定义一个 Functor 类：

```
class Functor {
    constructor(value) {
        this.value = value
    }
    static of(value) {
        return new Functor(value)
    }
```

```
    apply(fn) {
        return Functor.of(fn(this.value))
    }
}
```

Functor 可以理解为函子雏形，我们看它做了什么：Functor 的 constructor 按照惯例接收数据；同时定义 Functor 一个静态方法 of，这个方法专门用来返回一个 Functor 实例对象；apply 方法接受一个 fn，使用 fn 对当前实例的 value 进行计算，得到新的 value 之后交给静态 of 方法，最终得到还有新 value 的实例。

这样一来，可以：

```
Functor.of('lucas').apply(addHelloPrefix).apply(addByeSuffix)
```

仍然得到结果：

```
{value: "Hello : lucas, bye!"}
```

我们总结一下：

Functor 可以理解为一个容器，这个容器中含有值 this.value.；

Functor 具有 apply 方法，该方法将容器里面的每一个值，应用到到另一个容器；

上述所说的「应用到另一个容器」，是根据 of 方法，得到新的实例；

所有运算，都是通过函子 Functor 来完成，这样一来，所有运算不直接针对值，而是针对这个值的容器 —— Functor；

这样也保证了值的不可变性；

函数式编程有个约定，函子 Functor 需要拥有一个 of 方法，用来生成新的容器。

Maybe 函子

在函字 Functor 的基础上，为了安全性，我们将函子里的空值过滤掉：

```
class Maybe {
   constructor(value) {
       this.value = value
   }
   static of(value) {
       return new Maybe(value)
   }
   apply(fn) {
       return this.value ? Maybe.of(fn(this.value)) : new
Maybe(null)
   }
}
```

注意在 apply 方法中，对当前值 this.value 进行判断，如果非空，返回 Maybe.of(fn(this.value)) 调用，否则直接返回 Maybe(null)。

这就是简单的 Maybe 函子，为此我们总结一下：**不同类型的函子，可以完成不同的功能。他们的共同点是：每个函子并没有直接去操作需要处理的数据（我们没有看到的 this.value 的直接写操作）而是通过 apply 接口应用 fn，并最终返回一个新的函子。**

总结

函数式编程博大精深，这一节课只是给大家介绍了 JavaScript 中结合函数式编程常用的概念，已经足够日常开发了。掌握这些概念的 JavaScript 实现是进阶所必需的要求。

此外读者还可以更多考虑函数式编程的性能负担以及框架类库函数式实现的话题，函数式结合 ES Next decorator 也可以玩出很多花样；Ramda.js 是一个典型的函数式类库，感兴趣的读者也可以深入研究。同时给大家推荐一本函数式编程的书《JS 函数式编程指南》，它特有免费的中文版：mostly-adequate-guide。

总之，函数式的话题我们就到此结束，你可以研究得更深、更多，但理解本节课内容足够让读者有的放矢，在工程中合理使用。

点击查看下一节

那些年常考的前端算法（1）