

函数式编程进阶：1.杰克船长的黑珍珠号

 赵祥涛
发布到 [技术 / 前端及客户端](#) 发布时间：2019-03-11

本文仅面向以下用户开放，请注意内容保密范围
查看权限：完全公开

函数式编程中常用的范畴论的概念Functor、Monad等在代码中的体现，以及在实际代码中的应用。

函数式编程(Functional Programming)这一理念不论是在前端领域还是后端领域，都逐渐热门起来，现在不大量使用函数式编程技术的大型应用程序已经很罕见了。实际上函数式编程绝不是最近几年才被创造的编程范式，而是在计算机科学的开始，[Alonzo Church](#) 在20世纪30年代发表的lambda 演算,可以说是函数式编程的前世今生。

本篇文章也不是吹捧函数式编程(FP)相对面向对象编程(OOP)的优越，或者建议你使用哪一种编程范式。OOP 和FP 是一种假对立。我所见过的所有优秀的JavaScript应用程序或者类库，都充分利用和广泛混合了FP 和OOP 。

在我们开始之前，我们首先确认你已经熟练掌握了，"[一等公民的函数](#)"，"[高阶函数](#)"，"[纯函数](#)"，"[函数柯里化以及偏应用](#)"，"[函数组合](#)"，"[不可变性](#)"，"[递归](#)"等基本概念(不熟练也没关系,只不过是速度慢一点)。

The Mighty Box

73

浏览

0

点赞

0

评论

👍 赞

⋮

标签

- 原创 函数式编程 范畴论
- Functor

目录

- The Mighty Box
- My First Functor
- 释放Box中值
- Functor的实际应用



我们已经知道如何书写函数式的程序了，即通过管道把数据在一系列纯函数间传递的程序。我们也知道了，这些程序就是声明式的行为规范。但是，控制流(control flow)、异常处理(error handling)、异步操作(asynchronous actions)和状态(state)呢？还有更棘手的副作用(effects)呢？不要着急，我们马上将对上述这些抽象概念赖以建立的基础作一番探究。

首先我们看一段简单的代码，即作为对函数组合等概念的回顾，也作为即将开启的新征程的第一步：

函数式编程进阶：1.杰克船长的黑珍珠号

👍 赞

```
const number = parseInt(trimmed)
const nextNumber = number + 1
return String.fromCharCode(nextNumber)
}

const result = nextChartFromNumberString(' 64')

console.log(result) // => 'A'
```

类似于这段代码的业务代码在我们的日常项目中很常见，不过是不是可以用“中学的函数组合”的概念进行管道操作，并消除这么多的中间变量，保持一种Point-Free 风格的代码：

```
const nextChartFromNumberString = str =>
  String.fromCharCode(parseInt(str.trim()) + 1)

const result = nextChartFromNumberString(' 64')

console.log(result) // => 'A'
```

Are you kidding me? 这是Function Composition？这是Point-Free？好吧！我们换一种更函数式的写法：

```
const compose = (...fns) => x => fns.reduceRight((v, f) => f(v), x)

const trim = str => str.trim()
const toNumber = str => parseInt(str)
const nextNumber = number => number + 1
const createStr = number => String.fromCharCode(number)

const nextChartFromNumberString = compose(createStr, nextNumber, toNumber, trim)

const result = nextChartFromNumberString(' 64')
```

函数式编程进阶：1.杰克船长的黑珍珠号

👍 赞

嗯！终于有了点函数式的味道！这个时候，我们发现传入函数`nextChartFromNumberString`的参数' 64' 像一个工厂的零配件一样在流水线上先后被函数`trim`,`toNumber`,`nextNumber`,`createStr` 所操作。' 64' 像水一样在管道中流通。看到这一幕我们是不是有点眼熟，Array的`map`,`filter`,不就是完全类似的概念吗？所以我们可以用Array把我们输入的参数进行包装：

```
const nextChartFromNumberString = str =>
  [str]
    .map(s => s.trim())
    .map(r => parseInt(r))
    .map(i => i + 1)
    .map(i => String.fromCharCode(i))

const result = nextChartFromNumberString(' 64')

console.log(result) // => ['A']
```

仔细观察发现Array只是我们数据的容器，我们只是想利用Array的map方法罢了，其他的方法我们暂时用不到，那么我们何不创建一个Box容器呢？

```
const Box = x => ({
  map: f => Box(f(x)),
  inspect: () => `Box(${x})`
})

const nextChartFromNumberString = str =>
  Box(str)
    .map(s => s.trim())
    .map(r => parseInt(r))
    .map(i => i + 1)
    .map(i => String.fromCharCode(i))

const result = nextChartFromNumberString(' 64')
```

函数式编程进阶：1.杰克船长的黑珍珠号

👍 赞

这里使用函数Box来生产对象，不使用ES6的Class的原因是，尽量避免了"糟糕"的new和this关键字，new让人误以为创建了Class的实例，但其实根本不存在所谓的实例化，只是简单的属性委托机制(对象组合的一种)；而this则引入了执行上下文和词法作用域的问题，而我只是想创建一个简单的对象而已！

inspect方法的目的是为了使用NodeJs中的console.log隐式的调用它，方便我们查看数据的类型；而这一方法在浏览器中不可行，可以用console.log(String(x)) 来代替

My First Functor

Box中这个map 跟数组那个著名的map 一样，除了前者操作的是Box(x) 而后者是[x]。它们的使用方式也几乎一致，把一个值丢进Box,然后不停的map、map、map....：

```
Box.of(2).map(two => two + 2);  
// => Box(4)  
  
Box.of('flamethrowers').map(s => s.toUpperCase());  
// => Box('FLAMETHROWERS')  
  
// append 和 prop 是都已经柯里化的函数，等待接受下一个参数  
Box.of('bombs').map(append(' away')).map(prop('length'));  
// => Box(10)
```

这是讲解函数式编程的第一个容器，我们将它称之为Box,而数据就像杰克船长瓶子中的黑珍珠号一样，我们只能通过map方法去操作其中的值，而Box像是一种虚拟的屏障，也可以说在一定程度上保护Box中的值，不被随意的获取和操作。

为什么要使用一种这样的思路？因为我们能够在不离开Box的情况下操作容器里面的值。Box里的值传递给map 函数之后，就可以任我们操作；操作结束后，为了防止意外再把它放回它所属的Box。这样做的结果是，我们能连续地调用map，运行任何我们想运行的函数。甚至还可以改变值的类型，就像上面最后一个例子中那样。

map是可以使用lambda表达式变换容器内的值的有效且安全的途径。

等等，如果我们能一直调用map，那它不就是个组合(composition)么！这里边是有什么数学魔法在起作用？是Functor。各位，这个数学魔法就是Functor。

函数式编程进阶：1.杰克船长的黑珍珠号

👍 赞

没错，`Functor` 就是一个签了合约的接口。我们本来可以简单地把它称为`Mappable`，`Functor` 是范畴学里的概念，我们会在后面讨论于此相关的数学知识；暂时我们先用这个名字很奇怪的接口做一些不那么理论的、实用性的练习。

把值装进一个容器，而且只能使用`map`来处理它，这么做的理由到底是什么呢？如果我们换种方式来问，答案就很明显了：让容器自己去运用函数能给我们带来什么好处？

答案是：**抽象，对于函数运用的抽象。**

当`map`一个函数的时候，我们请求容器来运行这个函数不夸张地讲，这是一种十分强大的理念。

`map`知道如何在上下文中应该函数值。它首先会打开该容器，然后把值通过函数映射为另外一个值，最后把结果值再次包裹到一个新的同类型的容器中。拥有这种函数的类型被称为**`Functor`**。

`map`的一般定义为：

```
map :: (a -> b) -> Box(a) -> Box(b)
```

(先接收一个`a->b`的函数，然后再接收一个`Box(a)`)作为参数，最后返回一个`Box(b)`

毫无疑问这种链式的连续调用太眼熟了。其实绝大多数的开发人员一直在使用`Functor`却没有意识到而已。比如：

- `Array`的`map`和`filter`
- `Jquery`的`css`和`style`
- `Promise`的`then`和`catch`方法(What? `Promise`也是一种`Functor`? Yes!)
- `Rxjs Observable`的`map`和`filter` (什么？异步函数的组合？Relax!)

都是返回同样类型的`Functor`，因此可以不断的连续调用,其实这些都是`Box`理念的延伸，后面的文章会给出更详细的说明和讨论。

```
[1, 2, 3].map(x => x + 1).filter(x => x > 2)

$("#mybtn").css("width", "100px").css("height", "100px").css("background", "red");

Promise.resolve(1).then(x => x + 1).then(x => x.toString())

Rx.Observable.fromEvent($input, 'keyup')
  .map(e => e.target.value)
```

函数式编程进阶：1.杰克船长的黑珍珠号

👍 赞

这里使用连续dot、dot、dot链式调用而不是使用compose组合的原因是为了更方便的理解，compose甚至摆脱了对原始对象的依赖，更符合函数式的思维

释放Box中值

类似于`Box(2).map(x => x + 2)` 我们已经可以把任何类型的值，包装到Box中，然后随意的map、map、map...

另一个问题，我们怎么取出来我们的值呢？我想要的结果是4 而不是Box(4)！

如果黑珍珠号不能从瓶子中释放出来又有什么用处呢？接下来让杰克斯派洛船长拔出宝剑，释放出来黑珍珠号！

是时候，为我们的这个最为原始的Functor添加别的方法了。

```
const Box = x => ({
  map: f => Box(f(x)),
  fold: f => f(x),
  inspect: () => `Box(${x})`
})

Box(2)
  .map(x => x + 2)
  .fold(x => x)  // => 4
```

嗯，看出来fold 和map 的区别了吗？

map 是把函数执行的结果重新包装到Box中后然后返回一个新的Box类型，而**fold** 则是直接把函数执行的结果return出来，就结束了！

Functor的实际应用

Try-Catch

在许多情况下都会发生JavaScript的错误，特别是在与服务器通讯时，或者时在试图访问一个为null的对象的属性时。我们总是要预先做好最坏的打算。而这种大部分都是通过try-catch 来实现的。

函数式编程进阶：1.杰克船长的黑珍珠号



```
const findColor = name => ({ black: '#00000', white: '#ffffff', red: '#ff3366' })[name]

const result = findColor('white').slice(1).toUpperCase()
console.log(result) // => 'FFFFFF'

const result2 = findColor('green').slice(1).toUpperCase()
console.log(result2) // => TypeError: Cannot read property 'slice' of undefined
```

那么现在代码报错了，try-catch可以一定程度上解决这个问题：

```
try {
  const result2 = findColor('green').slice(1).toUpperCase()
  console.log(result2)
} catch (e) {
  console.log('error', e.message) // => error: Cannot read property 'slice' of undefined
}
```

一旦发生了错误，JS会立即终止程序，并创建导致该问题的函数的调用堆栈跟踪，并保存到Error对象中，catch就像是我们代码的避风港湾一样。但是try-catch能妥善的解决我们的问题吗？try-catch存在以下缺点：

- 难以与其他函数组合或链接,总不能让管道中的下一个函数处理上一个函数抛出的错误吧
- 违反了引用透明原则，因为抛出异常会导致函数调用出现另一个出口，所以不能确保单一的可预测的返回值
- 会引起副作用，因为异常会在函数调用之外对堆栈引发不可预料的影响
- 违反局域性的原则，因为用于恢复异常的代码和原始的函数调用渐行渐远，当发生错误的时候，函数会离开局部栈和环境
- 不能只关心函数的返回值，调用者需要负责声明catch块中的异常匹配类型来管理特定的异常
- 当有多个异常条件的时候会出现嵌套的异常处理块

异常应该由一个地方抛出，而不是随处可见

上面的描述和代码可以看出，try-catch是完全被动的解决方式，若是机能轻松的处理错误甚至包容错误，该有多好？

不错，是Either 登场的时候了，Either 包含了两个分支Left 和Right ,其中Left 指代出现异常的分支，Right 指代正常

函数式编程进阶：1.杰克船长的黑珍珠号

👍 赞

Left & Right 完全类似于Promise中的 Reject & Resolve

上面的描述实在是太笼统了，让我直接看看代码：

```
const Left = x => ({
  map: f => Left(x),
  fold: (f, g) => f(x),
  inspect: () => `Left(${x})`
})

const Right = x => ({
  map: f => Right(f(x)),
  fold: (f, g) => g(x),
  inspect: () => `Right(${x})`
})

const resultLeft = Left(4).map(x => x + 1).map(x => x / 2)
console.log(resultLeft) // => Left(4)

const resultRight = Right(4).map(x => x + 1).map(x => x / 2)
console.log(resultRight) // => Right(2.5)
```

Left 和Right 的区别在于Left会自动跳过map 方法传递的函数，而Right则类似于最基本的Box，会执行函数并把返回值重新包装到Right容器里面。上面的代码说明了Left和Right的基本用法，现在把我们的Left & Right 应用到findColor 函数上吧！

```
const findColor = name => {
  const found = ({ black: '#00000', white: '#ffffff', red: '#ff3366' })[name]
  return found ? Right(found) : Left(null)
}

const result = findColor('green')
```

函数式编程进阶：1.杰克船长的黑珍珠号

👍 赞

```
console.log(result) // => no color
```

不可相信！我们现在竟然线性的处理的错误，并且甚至能够给出一个'no color'的提醒了(通过给fold提供)，但是再仔细思考一下，是不是我们原始的findColor函数，有可能会返回undefined 或者一个正常的值,是不是可以直接包装一下这个函数的返回值呢？

```
const fromNullable = x =>
  x != null ? Right(x) : Left(null)

const findColor = name =>
  fromNullable({ black: '#00000', white: '#ffffff', red: '#ff3366' }[name])

const result = findColor('green')
  .map(c => c.slice(1))
  .fold(() => 'no color', c => c.toUpperCase())

console.log(result) // => no color
```

现在我们已经成功处理了可能出现null或者undefined的情况，那么try-catch呢？是否也可以被Either包装一下呢？

```
const tryCatch = (f, ...arg) => {
  try {
    return Right(f(...arg))
  } catch (e) {
    return Left(e)
  }
}

const jsonFormat = str => JSON.parse(str)

const result = tryCatch(jsonFormat, '{"path": "some path"}')
```

函数式编程进阶：1.杰克船长的黑珍珠号

👍 赞

```
console.log(result) // => 'some path'

const result2 = tryCatch(jsonFormat, 'the way to death')
  .map(x => x.path)
  .fold(() => 'default path', x => x)

console.log(result2) // => 'default path'
```

现在我们的tryCatch即使报错了，也不会打断我们的函数组合了；并且错误得到了合理的控制，不会随意的throw出来一个Error对象了；现在我们的错误像一个安静的孩子一样，再也不会到处大吵大闹了，导致我们崩溃了。

但是我们现在还没有解决tryCatch的多个错误的嵌套问题，不要着急，接下来的文章我们会先继续介绍几个常用的Functor (薛定谔的Maybe,背锅侠的IO等)以及Functor的数学相关的理论，再然后我们将直奔Monad而去,下面给出一句经典名言作为引言：

“A monad is just a monoid in the category of endofunctors. What’s the problem?”

参考资料：

[1]: [What is a functor?](#)

[2]: [So You Want to be a Functional Programmer](#)

[3]: [Two Years of Functional Programming in JavaScript: Lessons Learned](#)

[4]:[Master the JavaScript Interview: What is Functional Programming?](#)

[5]: 《JavaScript函数式编程》

[6]: [写给程序员的范畴论](#)

*本内容仅代表个人观点，不代表网易，仅供内部分享传播，不允许以任何形式外泄，否则追究法律责任。

快来成为第一个打赏的人吧~

函数式编程进阶：1.杰克船长的黑珍珠号

👍 赞

☐ 匿名评论

评论

全部评论 0



暂无评论

相关推荐

gzip在web中的应用探索

黄梦玲 2个月前



严选 Android 组件化实践

张云龙 20 天前



函数式编程进阶：1.杰克船长的黑珍珠号

的应用。

👍 赞

赵祥涛 1 天前



常用链接

- 易协作
- 会议预定
- 游戏部IT资源
- 无线助手
- 关于我们
- 平台用户协议
- OA
- 文具预定
- 网易POPO
- 工作报告
- 帮助中心



POPO服务号



KM APP下载

