03 | 如何通过部分应用和柯里化让函数具象化？

2022-09-24 石川 来自北京

《JavaScript进阶实战课》
课程介绍 >

讲述：石川
时长 12:20　大小 11.27M

你好，我是石川。

在前面两节课里，我说过函数式编程的核心就是把数据作为输入，通过算法进行计算，最后输出结果。同时我也提到，在函数式 + 响应式编程中，面对未知、动态和不可控时，可以通过纯函数和不可变等手段减少副作用、增加确定性，及时地适应和调整。

那么现在你来想想，**在输入、计算和输出这个过程中，什么地方是最难控制的呢？**对，就是输入。因为它来自外界，而计算是在相对封闭的环境中，至于输出只是一个结果。

所以今天这节课，我们就来说说输入的控制。

部分应用和柯里化

在前面课程里也讲过，函数的输入来自参数，其中包括了函数定义时的**形参**和实际执行时的**实参**。另外，我们也通过 React.js 中的 props 和 state 以及 JavaScript 中的对象和闭包，具体了解了如何通过不可变，做到对**运行时的未知**状态变化的管理。

那今天，我们就从另外一个角度理解下对编程时“未知”的处理，即如果我们在编写一个函数时，需要传入多个实参，其中一部分实参是先明确的，另一部分是后明确的，那么该如何处理呢？

其实就是**部分应用（partial application）和柯里化（currying）**。下面我们就一起来看看函数式编程中，如何突破在调用点（call-site）传参的限制，做到部分传参和后续执行。

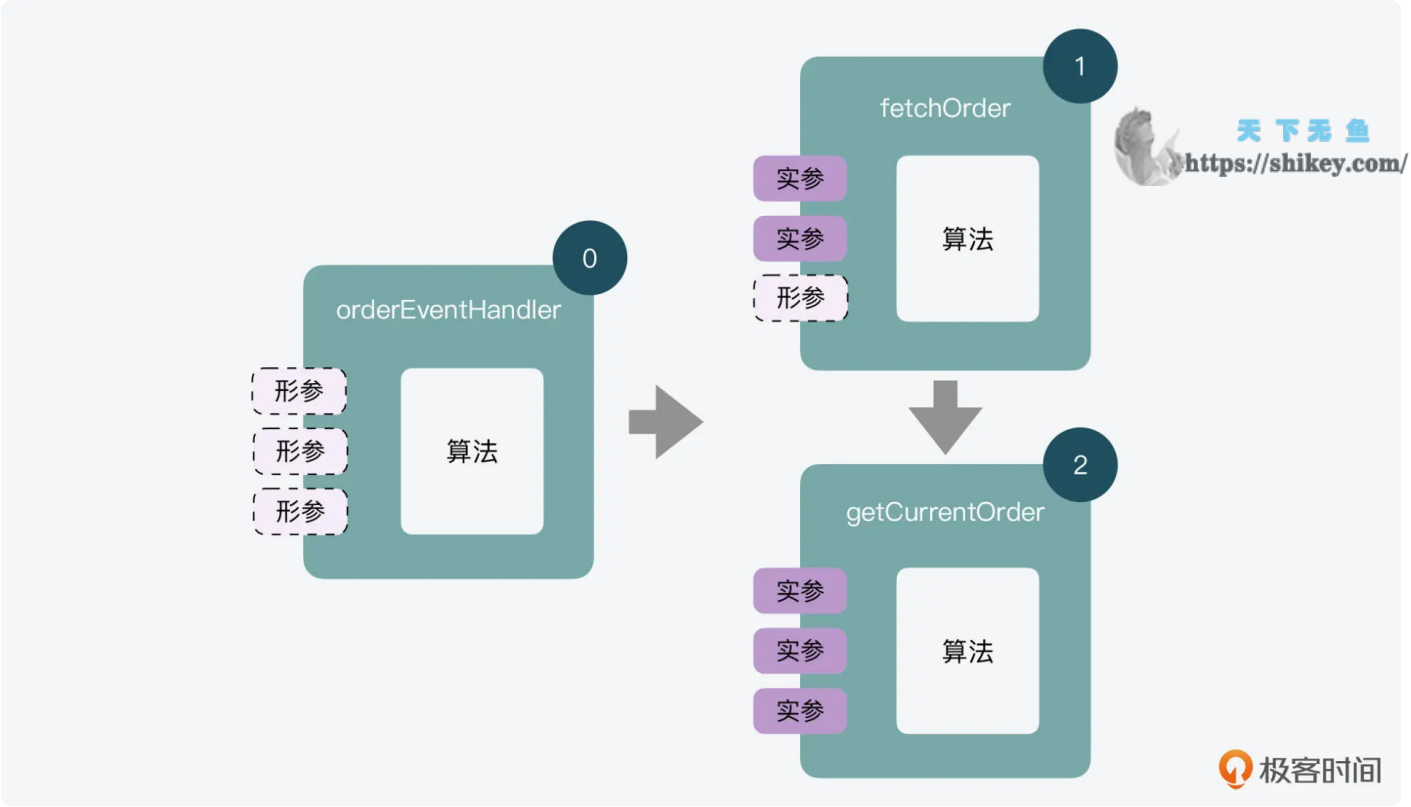
通过部分应用延迟实参传入

我们知道，函数式编程重在声明式和可读性，而且强调每个函数尽量解决一个单一问题。假设有一个 orderEventHandler 函数，它比较抽象，因此缺少可读性；又或者假设下面这个函数需要 url、data 和 callback 三个参数的输入，才能执行，我们预先知道它的 url，却不知道它的 data 和 callback。这时该怎么办呢？

复制代码

```
function orderEventHandler(url,data,callback) {
    // ..
}
```

要解决这些问题，我们就可以通过部分应用。下面是它的一个执行流程图。



也就是说，我们可以通过 orderEventHandler 函数，具象出一个专门的 fetchOrder 函数。通过这种方式，我们就提前预置了已知参数 url，减少了后面需要传入的参数数量，同时也增加了代码的可读性。

复制代码

```
function fetchOrder(data,cb) {
    orderEventHandler( "http://some.api/order", data, cb );
}
```

可是如果我们想进一步具象化，预制一些参数怎么办？比如下面的 getCurrentOrder，如果我们想把前面 fetchOrder 里的 data，也内置成 order: CURRENT_ORDER_ID，这样会大量增加代码结构的复杂性。

复制代码

```
function getCurrentOrder(cb) {
    getCurrentOrder( { order: CURRENT_ORDER_ID }, cb );
}
```

所以在函数式编程中，我们通常会使用部分应用。它所做的就是**抽象一个 partial 工具**，在先预制部分参数的情况下，后续再传入剩余的参数值。如以下代码所示：

```
var fetchOrder = partial( orderEventHandler, "http://some.api/order" );
var getCurrentOrder = partial( fetchOrder, { order: CURRENT_ORDER_ID } );
```


partial 工具可以借助我们在上节课提到过的**闭包**，以及 ES6 中引入的**...延展操作符**（spread operator）这两个函数式编程中的利器来实现。

我先来说一下延展操作符，它的强大之处就是可以在函数调用或数组构造时，将数组表达式或者 string 在语法层面展开。在这里，我们可以用它来处理预置的和后置的实参。而闭包在这里再次发挥了记忆的功能，它会记住前置的参数，并在下一次收到后置的参数时，可以和前面记住的前置参数一起执行。

```
var partial =
    (fn,...presetArgs) =>
        (...laterArgs) =>
            fn( ...presetArgs, ...laterArgs );
```

除此之外，我们在上一讲里提到的 bind 也可以起到类似的作用。但 **bind 通常是在面向对象中用来绑定 this 的**，用作部分应用的话会相对比较奇怪，因为这里我们不绑定 this，所以第一个参数我们会设置为 null。

当然，这么用确实不会有什么问题，但是一般来说，为了不混淆 bind 的使用场景，我们最好还是用自己定义的 partial 工具。

```
var fetchOrder = httpEvntHandler.bind( null, "http://some.api/order" );
```

通过柯里化每次传一个参数

我们接着来看看柯里化。

可以看到，在下面的例子中，我们把之前的 httpEventHandler 做了柯里化处理之后，就不需要一次输入 3 个参数了，而是每次只传入一个参数。第一次，我们传入了 url 来获取订单；之

后，我们传入了当前订单的 id；最后，我们获得了当前订单后，传入一个订单修改的参数来做相关修改。

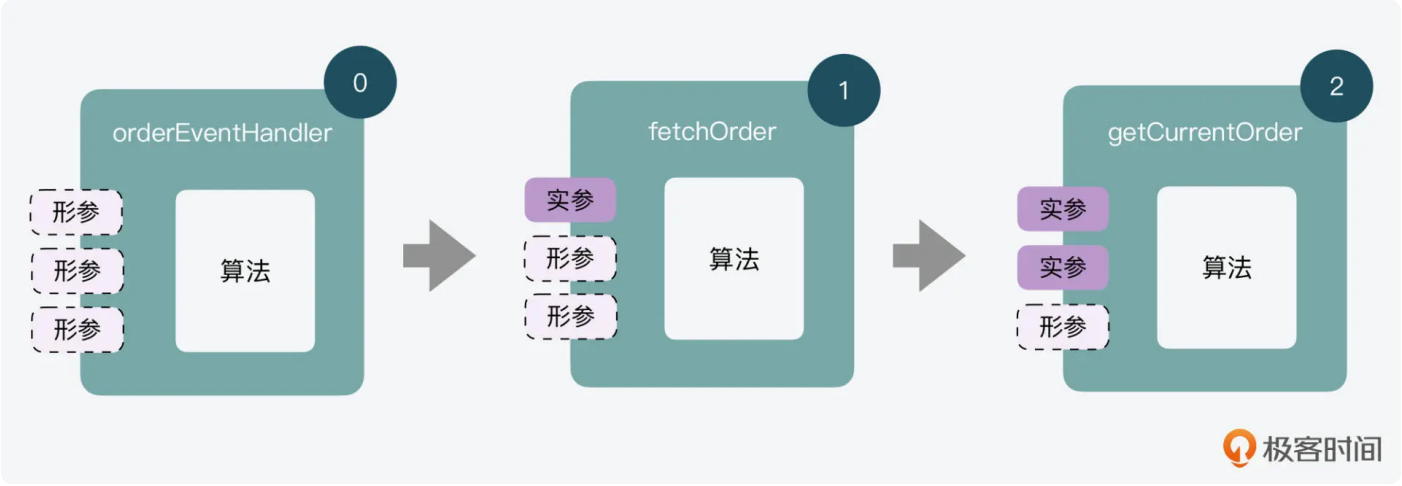

```
var curriedOrderEvntHandler = curry( orderEventHandler );

var fetchOrder = curriedHttpEvntHandler( "http://some.api/order" );

var getCurrentOrder = fetchOrder( { order: CURRENT_ORDER_ID } );

getCurrentOrder( function editOrder(order){ /* .. */ } );
```

你同样可以来看一下它的一个执行流程图，看看柯里化是如何实现的。



实际上，和部分应用类似，这里我们**也用到了闭包和...延展操作符**。

在柯里化中，延展操作符可以在函数调用链中起到承上启下的作用。当然，市面上实现部分应用和柯里化的方式有很多，这里我选了一个“可读性”比较高的。因为和部分应用一样，它有效说明了参数前后的关系。

```
function curry(fn,arity = fn.length) {
    return (function nextCurried(prevArgs){
        return function curried(nextArg){
            var args = [ ...prevArgs, nextArg ];
            if (args.length >= arity) {
                return fn( ...args );
            }
            else {
                return nextCurried( args );
```

```
            }
        };
    })( [] );
}
```


好了，通过部分应用和柯里化的例子，我们能够发现**函数式编程处理未知**的能力。但这里我还想强调一点，这个未知，跟我们说的应用在运行时的未知是不同的。这里的未知指的是编程时的未知，比如有些参数是我们提前知道的，而有一些是后面加入的。

要知道，一个普通的函数通常是在调用点执行时传入参数的，而通过部分应用和柯里化，我们做到了可以先传入部分已知参数，再在之后的某个时间传入部分参数，这样从时间和空间上，就将一个函数分开了。

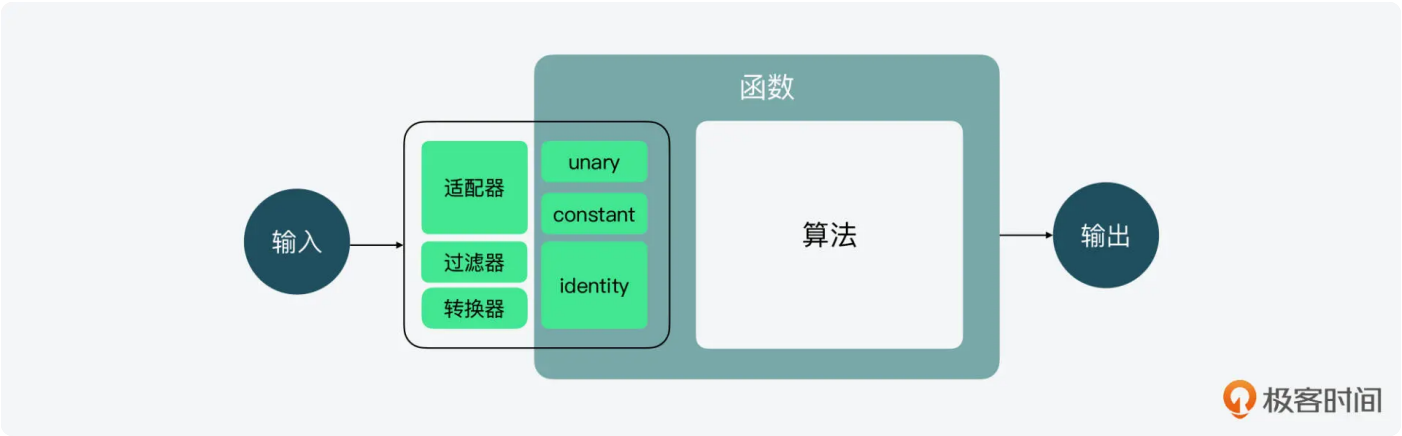
而这样做除了一些实际的好处，比如处理未知，让函数从抽象变具体、让具体的函数每次只专心做好一件事、减少参数数量之外，还有一个更抽象的好处，就是**体现了函数式底层的声明式思想**。

在这里，我们让代码变得更可读。

还有哪些常用的参数处理工具？

在函数式编程中，我们把参数的数量叫做 **arity**。从上面的例子中，我们可以看到，部分应用可以减少每次函数调用时需要传入的参数，而柯里化更是把函数调用时需要传入的参数数量，降到了 1。它们实际上都起到了**控制参数数量**的作用。

而在函数式编程中，其实还有很多可以帮助我们处理参数输入的工具。下面，我们就通过 unary、constant 和 identity 这几个简单的例子来一起看看。



改造接口 unary

我们先来看看改造函数的工具。其中，最简单的工具就是一元参数（unary）了，它的作用是把一个接收多个参数的函数，变成一个只接收一个参数的函数。其实现也很简单：


```
function unary(fn) {
    return function oneArg(arg){
        return fn( arg );
    };
}
```

你可能会问它有什么用？我来举个例子。

当你想通过 parseInt，把一组字符串通过 map 来映射成整数，但是 parseInt 会接收两个参数，而如果你直接输入 parseInt 的话，那么“2”就会成为它的第二个参数，这肯定不是你期待的结果吧。

所以这时候，unary 就派上用场了，它可以让 parseInt 只接收一个参数，从而就可以正确地打出你想要的结果。

```
["1","2","3","4","5"].map( unary( parseInt ) ); // [1,2,3,4,5]
```

看到这里，聪明的你可能会问：除了一元，会不会有二元、三元？答案是有的。二元就是 binary，或是函数式中的“黑话”dyadic；三元就是 tenary。顾名思义，它们分别代表的就是把一个函数的参数数量控制在 2 个和 3 个。

改造参数 constant

如果你用过 JavaScript promise 的话，应该对 then 不陌生。从函数签名的角度来看，它只接收函数，而不接收其他值类型作为参数。比如下面例子中，34 这个值就是不能被接收的。

```
promise1.then( action1 ).then( 34 ).then( action3 );
```

这里你可能会问，**什么是函数签名？**函数签名一般包含了参数及其类型返回值，还有类型可能引发或传回的异常，以及相关的方法在面向对象中的可用性信息（如关键字 public、static 或 prototype）。你可以看到在 C 或 C++ 中，会有类似这样的签名，如下所示：


```
// C
int main (int arga, char *argb[]) {}

// C++
int main (int argc, char **argv) {/** ... **/ }
```

而在 JavaScript 当中，基于它“放荡不羁”的特性，就没有那么多条条框框了，甚至连命名函数本身都不是必须的，就更不用说签名了。那么遇到 then 这种情况怎么办呢？

在这种情况下，我们其实可以编写一个只返回值的 constant 函数，这样就解决了接收的问题。由此也能看出，JavaScript 在面对各种条条框框的时候，总是上有政策下有对策。

```
function constant(v) {
    return function value(){
        return v;
    };
}
```

然后，我们就可以把值包装在 constant 函数里，通过这样的方式，就可以把值作为函数参数传入了。

```
promise1.then( action1 ).then( constant( 34 ) ).then( action3 );
```

不做改造 identity

还有一个函数式编程中常用的工具，也就是 identity，它既不改变函数，也不改变参数。它的功能就是输入一个值，返回一个同样的值。你可能会觉着，这有啥用？

```
```

```
function identity(v) {
    return v;
}
```


其实它的作用也很多。比如在下面的例子中，它可以作为**断言**（predicate）， 来过滤掉空值。在函数式编程中，断言是一个可以用来做判断条件的函数，在这个例子里，identity 就作为判断一个值是否为空的断言。

复制代码

```
var words = "   hello world  ".split( /\s|\b/ );
words; // ['', '', '', 'hello', 'world', '', '']

words.filter( identity ); // ['hello', 'world']
```

当然，identity 的功能也不止于此，它也可以用来**做默认的转化工具**。比如以下例子中，我们创建了一个 transLogger 函数，可以传入一个实际的数据和相关的 lower 功能函数，来将文字转化成小写。

复制代码

```
function transLogger (msg,formatFn = identity) {
    msg = formatFn( msg );
    console.log( msg );
}

function lower(txt) {
    return txt.toLowerCase();
}

transLogger( "Hello World" );             // Hello World
transLogger( "Hello World", lower );      // hello world
```

除了以上这些工具以外，还有更复杂一些的工具来解决参数问题。比如在讲部分应用和柯里化的时候，提到它在给我们带来一些灵活性的同时，也仍然会有一些限制，即**参数的顺序问题**，我们必须按照一个顺序来执行。而有些三方库提供的一些工具，就可以将参数倒排或重新排序。

重新排序的方式有很多，可以通过解构（destructure），从数组和对象参数中提取值，对变量进行赋值时重新排序；或通过延展操作符把一个对象中的一组值，“延展”成单独的参数来处

理；又或者通过 .toString() 和正则表达式解析函数中的参数做处理。

但是，有时我们**在灵活变通中也要适度**，不能让它一不小心变成了“奇技淫巧”，所以对于类似“重新排序”这样的技巧，在课程中我就不展开了，感兴趣的话你可以在延伸阅读部分去深入了解。

总结

通过今天这节课，我们能看到在面对未知、动态和不可控时，函数式编程很重要的一点就是**控制好输入**。

在课程里，我们一起重点了解了函数的输入中的参数，知道部分应用和柯里化，可以让代码更好地处理编程中的未知，让函数从抽象变具体，让具体的函数每次只专心做好一件事，以及可以在减少参数的数量之外，还能够增加可读性。

另外，我们也学习了更多“个子小，功能大”的工具，我们不仅可以通过这些工具，比如 unary 和 constant 来改造函数和参数，从而解决适配问题；同时，哪怕是看上去似乎只是在“透传”值的 identity，实际上都可以用于断言和转化。而这样做的好处，就是可以尽量提高接口的适应性和适配性，增加过滤和转化的能力，以及增加代码的可读性。

思考题

今天我们主要学习了柯里化，而与它相反的就是反柯里化（uncurry），那么你知道反柯里化的用途和实现吗？

欢迎在留言区分享你的答案，也欢迎你把今天的内容分享给更多的朋友。

延伸阅读

- JS 函数签名
- C/C++ 函数签名与名字修饰（符号修饰）
- JS 中的结构
- Functional Light JS
- JavaScript Patterns - Chapter 4 Functions

上一篇　02 | 如何通过闭包对象管理程序中状态的变化？

下一篇　04 | 如何通过组合、管道和reducer让函数抽象化？

精选留言 (8)

写留言

Guit
2022-09-29 来自北京

```
const currying = (fn) => {
  const l = fn.length
  return function curried(...prevArgs) {
    if (l === prevArgs.length) {
      return fn(...prevArgs)
    }
    return (...nextArg) => curried(...nextArg, ...prevArgs)
  }
}
```


Sunny
2022-09-29 来自北京

看下这个单词 tenary 是否少了一个 "r"，正确的是 ternary?

作者回复: 谢谢指正，我查了一下，你的拼写是正确的


L

2022-09-28 来自北京

var curriedOrderEvntHandler = curry(orderEventHandler);

var fetchOrder = curriedHttpEvntHandler("http://some.api/order");

var getCurrentOrder = getOrder({ order: CURRENT_ORDER_ID });

getCurrentOrder(function editOrder(order){ /* .. */ });
这里是否也有问题 为什么前后的函数对不上呢

> 作者回复: 是的，谢谢你的细心，这里有个勘误，getOrder应该都统一成fetchOrder。

1

L

2022-09-28 来自北京

function getCurrentOrder(cb) {
  getCurrentOrder({ order: CURRENT_ORDER_ID }, cb);
  (fetchOrder)
}
可是如果我们想进一步具象化，预制一些参数怎么办？比如下面的 getCurrentOrder，如果我们想把前面 getOrder（fetchOrder） 里的 data，也内置成 order: CURRENT_ORDER_ID，这样会大量增加代码结构的复杂性。
这两个地方是不是错了 应该都是fetchOrder

> 作者回复: 是的，谢谢你的细心，这里有个勘误，getOrder应该都统一成fetchOrder。

1

L

2022-09-28 来自陕西

可是如果我们想进一步具象化，预制一些参数怎么办？比如下面的 getCurrentOrder，如果我们想把前面 getOrder 里的 data，也内置成 order: CURRENT_ORDER_ID，这样会大量增加代码结构的复杂性。

Saul

2022-09-27 来自北京

```
var curriedOrderEvntHandler = curry( orderEventHandler );

var fetchOrder = curriedHttpEvntHandler( "http://some.api/order" );

var getCurrentOrder = getOrder( { order: CURRENT_ORDER_ID } );

getCurrentOrder( function editOrder(order){ /* .. */ } );
```

第二行我怎么看不懂

> 作者回复: 原先的orderEventHandler函数有3个参数，url, data 和 callback。
> function orderEventHandler(url,data,callback) {}
>
> 第一行把orderEventHandler做了柯里化，之后可以每次只传一个参数。
> 第二行是在柯里化后的curriedHttpEvntHandler中传入了第一个url实参。

1

向上
2022-09-27 来自浙江

function getCurrentOrder(cb) { getCurrentOrder({ order: CURRENT_ORDER_ID }, cb);}
这里内部函数是否应该用fetchOrder

灯火阑珊
2022-09-26 来自湖北

equational reasoning