

# 加餐四 | 一篇文章，带你快速理解函数式编程

2020-09-18 月影

跟月影学可视化

进入课程 >



讲述：月影

时长 11:31 大小 10.56M



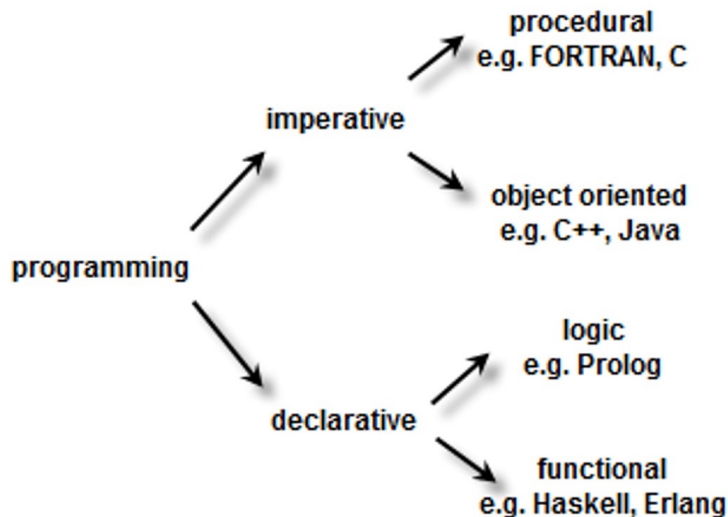
你好，我是月影。今天，我们来讨论函数式编程。

我看到很多同学留言说，课程中给出的代码例子有的地方看不明白。我把同学们看不懂的地方汇总了一下，发现大部分都是我使用函数式编程来写的。比如，我在第 7 讲说过的 parametric 高阶函数，第 12 讲说过的 traverse 的设计，还有第 15 讲中使用的 toPolar/fromPolar 和改进版的 parametric 设计，以及数据篇中的数据处理和 D3.js 的使用。

如果你还不习惯函数式编程思想，并且也觉得这些代码不容易理解，想知道为什么一定要这么设计，那这节课，你一定要好好听，我会和你系统地说过程抽象和函数式编程这个话题。

## 两种编程范式：命令式与声明式

首先，我先来说说什么是编程范式。编程范式有两种，分别是命令式（Imperative）和声明式（Declarative），命令式强调做的步骤也就是怎么做，而声明式强调做什么本身，以及做的结果。因此，编程语言也可以分成命令式和声明式两种类型，如果再细分的话，命令式又可以分成过程式和面向对象，而声明式则可以分成逻辑式和函数式。下面这张图列出了编程语言的分类和每个类型下经典的编程语言。




你注意看，这张图里并没有 JavaScript。实际上像 JavaScript 这样的现代脚本语言，通常具有混合范式，也就是说 JavaScript 同时拥有命令式和声明式的特征。因此开发者可以同时用 JavaScript 写出命令式与声明式风格的代码。举个例子，我们要遍历一个数组，将每一个元素的数值翻倍，我们可以分别用命令式和声明式来实现。

首先，是命令式的实现代码：

```
1 let list = [1, 2, 3, 4];
2
3 let map1 = [];
4 for(let i = 0; i < list.length; i++){
5   map1.push(list[i] * 2);
6 }
```

[复制代码](#)

然后是声明式的实现代码：

 复制代码

```
1 let list = [1, 2, 3, 4];
2 const double = x => x * 2;
3 list.map(double);
```


从上面的代码我们可以看到，虽然两段代码的目的相同，但是具体的实现手段差别很大。其中命令式强调怎么做，使用的是 for 循环来遍历，而声明式强调做什么，用到了 double 算子。

## 函数式与纯函数

既然编程风格有命令式和声明式，为什么我们在一些设计中更多会选择声明式风格的函数式编程，它究竟有什么好处呢？通过和前面的代码对比，我们看到似乎声明式（函数式）代码写起来更加简洁。是的，大部分情况下，函数式编程的代码更加简洁。但除了能减少代码量之外，函数式还有什么具体的好处呢？这个就要从纯函数说起了。

我们知道，函数是对过程的封装，但函数的实现本身可能依赖外部环境，或者有副作用（Side-effect）。所谓函数的副作用，是指函数执行本身对外部环境的改变。我们把不依赖外部环境和没有副作用的函数叫做纯函数，依赖外部环境或有副作用的函数叫做非纯函数。

这里，我们先来看一组例子：

 复制代码

```
1 function add(x, y) {
2   return x + y;
3 }
4
5 function getEl(id) {
6   return document.getElementById(id);
7 }
8
9 function join(arr1, arr2) {
10  arr1.push(...arr2);
11  return arr1;
12 }
```

在上面的代码中，add 是一个纯函数，它的返回结果只依赖于输入的参数，与调用的次数、次序、时机等等均无关。而 getEl 是一个非纯函数，它的返回值除了依赖于参数 id，

还和外部环境（文档的 DOM 结构）有关。另外，`join` 也是一个非纯函数，它的副作用是会改变输入参数对象本身的内容，所以它的调用次数、次序和时机不同，我们得到的结果也不同。

## 纯函数的优点

现在我们知道了纯函数与非纯函数的区别，但我们又为什么要人为地把函数划分为纯函数和非纯函数呢？这是因为纯函数与非纯函数相比，有三个非常大的优点，分别是易于测试（上下文无关）、可并行计算（时序无关）、有良好的 Bug 自限性。下面，我——来解释一下。

首先纯函数易于测试，在用单元测试框架的时候，因为纯函数不需要依赖外部环境，所以我们直接写一个简单的测试 case 就可以了。

[复制代码](#)

```
1 //test with pure functions
2 test(t => {
3   dosth...
4
5   done!
6 });
```

而非纯函数因为比较依赖外部环境，在测试的时候我们还需要构建外部环境。

[复制代码](#)

```
1 //test with impure functions
2
3 //always need hooks
4 test.before(t => {
5   //setup environments
6 });
7
8 test.after('cleanup', t => {
9   //clean
10 });
11
12 test(t => {
13   dosth...
14
15   done!
16 });
```

其次，纯函数可以并行计算。在浏览器中，我们可以利用 Worker 来并行执行多个纯函数，在 Node.js 中，我们也可以用 Cluster 来实现同样的并行执行，而使用 WebGL 的时候，纯函数有时候还可以转换为 Shader 代码，利用 GPU 的特性来进行计算。


最后，纯函数有良好的 Bug 自限性。这是什么意思呢？因为纯函数不会依赖和改变外部环境，所以它产生的 Bug 不会扩散到系统的其他部分。而非纯函数，尤其是有副作用的非纯函数，在产生 Bug 后，因为 Bug 可能意外改变了外部环境，所以问题会扩散到系统其他部分。这样在调试的时候，就算发现了 Bug，你可能也找不到真正导致 Bug 的原因，这就给系统的维护和 Bug 追踪带来困难。

总而言之，我们设计系统的时候，要尽可能多设计纯函数，少设计非纯函数，这样能够有效提升系统的可测试性、性能优化空间以及系统的可维护性。

## 函数式编程范式与纯函数

那么问题来了，我们该如何让系统的纯函数尽可能多，非纯函数尽可能少呢？答案是用函数式编程范式。我们还是通过一个例子来理解。

我们要实现一个模块，用它来操作 DOM 中列表元素，改变元素的文字颜色，具体的实现代码如下：

 复制代码

```
1 function setColor(el, color){
2   el.style.color = color;
3 }
4
5 function setColors(els, color){
6   els.forEach(el => setColor(el, color));
7 }
```

这个模块中有两个方法，其中 setColor 是操作一个 DOM 元素，改变它的文字颜色，而 setColors 则是批量操作若干个 DOM 元素，改变所有元素的颜色。

尽管这两个方法都非常简单，但它们都改变了外部环境（DOM）所以它们是两个非纯函数。因此，我们在做系统测试的时候，两个方法都需要构建外部环境来实现测试。



如果能让系统测试更简单，我们是不是可以采用函数式编程思想，把非纯函数的个数减少一个呢？当然可以，我们可以实现一个 batch 函数来优化。batch 函数接受的参数是一个函数 f，就会返回一个新的函数。在这个过程中，我们要遵循的调用规则是，如果这个参数有 length 属性，我们就以数组来遍历这个参数，用每一个元素迭代 f，否则直接用当前调用参数来调用 f 就可以了。

具体的实现代码如下：

[复制代码](#)

```
1 function batch(fn){
2   return function(target, ...args){
3     if(target.length >= 0){
4       return Array.from(target).map(item => fn.apply(this, [item, ...args]));
5     }else{
6       return fn.apply(this, [target, ...args]);
7     }
8   }
9 }
```

因为 batch 函数的参数和返回值都是函数，所以它有一个专属的名字，**高阶函数 (High Order Function)**。高阶函数虽然看上去复杂，但它实际上就是一个纯函数。它的执行结果只依赖于参数（传入的函数），与外部环境无关。

我们可以测试一下这个 batch 函数的正确性，方法十分简单只要用下面这个 Case 就行了。

[复制代码](#)

```
1 test(t => {
2   let add = (x, y) => x + y;
3   let listAdd = batch(add);
4
5   t.deepEqual(listAdd([1,2,3], 1), [2,3,4]);
6 });
```

有了 batch 函数之后，我们的模块就可以减少为一个非纯函数。

[复制代码](#)

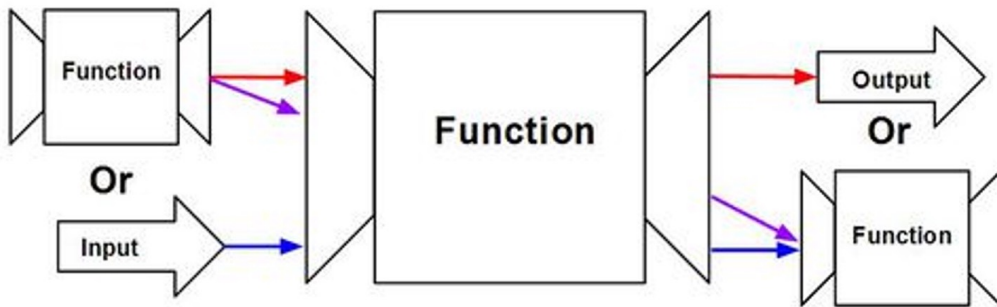
```
1 function setColor(el, color){
```

```
2   el.style.color = color;  
3 }  
4  
5 let setColors = batch(setColor);
```

这里我们用 `batch` 来实现 `setColors`，只要 `batch` 实现正确，`setColors` 的行为就可以保证是正确的。

## 高阶函数与函数装饰器

刚才我说，`batch` 是一个高阶函数。所谓高阶函数，是指输入参数是函数，或者返回值是函数的函数。




如果输入参数和返回值都是函数，这样的高阶函数又叫做**函数装饰器 (Function Decorators)**。当一个高阶函数是用来修饰函数本身的，它就是函数装饰器。也就是说，它是在原始函数上增加了某些带有辅助功能的函数。

这么说你可能不太理解，我们再来看一个例子。

假设，我们的代码库要进行大版本升级，在未来最新的版本中我们想要废弃掉某些 API，由于很多业务中使用了老版本的库，不可能一次升级完，因此我们需要做一个平缓过渡。具体来说就是在当前这个版本中，先不取消这些旧的 API，而是给它们增加一个提示信息，告诉调用它们的用户，这些 API 将会在下一次升级中被废弃。


如果我们手工修改要废弃的 API 代码，这会是一件非常繁琐的事情。而且，我们很容易遗漏或者弄错些什么，从而产生不可预料的 Bug。

所以，一个比较聪明的办法是，我们实现一个通用的函数装饰器。

 复制代码

```
1 function deprecate(fn, oldApi, newApi) {
2   const message = `The ${oldApi} is deprecated.
3   Please use the ${newApi} instead.`;
4
5   return function(...args) {
6     console.warn(message);
7     return fn.apply(this, args);
8   }
9 }
```

然后，在模块导出 API 的时候，对需要废弃的方法统一应用这个装饰器。

 复制代码

```
1 // deprecation.js
2 // 引入要废弃的 API
3 import {foo, bar} from './foo';
4 ...
5 // 用高阶函数修饰
6 const _foo = deprecate(foo, 'foo', 'newFoo');
7 const _bar = deprecate(bar, 'bar', 'newBar');
8
9
10 // 重新导出修饰过的API
11 export {
12   foo: _foo,
13   bar: _bar,
14   ...
15 }
```

这样，我们就利用函数装饰器，无侵入地修改了模块的 API，将要废弃的模块用 deprecate 包装之后再输出，就实现了我们想要的效果。这里，我们实现的 deprecate 就是一个纯函数，它的维护和使用都非常简单。

## 过程抽象

理解了前面的例子之后，咱们再回过头来，说说课程中的函数式编程。我们直接来看第 7 节课里 parametric 函数的实现。



```
1 function parametric(xFunc, yFunc) {
2   return function (start, end, seg = 100, ...args) {
3     const points = [];
4     for(let i = 0; i <= seg; i++) {
5       const p = i / seg;
6       const t = start * (1 - p) + end * p;
7       const x = xFunc(t, ...args); // 计算参数方程组的x
8       const y = yFunc(t, ...args); // 计算参数方程组的y
9       points.push([x, y]);
10    }
11    return {
12      draw: draw.bind(null, points),
13      points,
14    };
15  };
16 }
```

如上面代码所示，`parametric` 是一个高阶函数，它比上面的函数装饰器更加复杂一点的是，它的输入是两个函数 `xFunc` 和 `yFunc`，输出也是一个函数，返回的这个函数实际上是一个**过程**，这个过程是对 `x`、`y` 的参数方程根据变量 `t` 的值进行采样。

所以，实际上 `parametric` 函数封装的是一个过程，这种封装过程的思路，叫做**过程抽象**。前面的函数装饰器，还有 `batch` 方法，实际上也是过程抽象。对应的一般程序设计中我们不是封装过程，而是封装数据，所以叫做数据抽象。

过程抽象是函数式编程的基础，函数式编程对待函数就像对待数据一样，都会进行封装和抽象，这样能够设计出非常通用的功能模块。

## 要点总结

函数式编程的内容非常多，这一节课，我只是借助了这些基础的概念和代码，把你带进了函数式编程的大门。

首先，我们了解了两种不同的编程范式，分别是命令式和声明式。其中，函数式属于声明式，而过程式和面向对象则属于命令式。JavaScript 语言是同时具有命令式和声明式特征的编程语言。

然后，我们知道函数式有一个非常大的优点，就是能够减少非纯函数的数量，这也是我们设计系统时要遵循的原则。因为相比于非纯函数，纯函数具有更好的可测试性、执行效率

和可维护性。

最后，我们还学会了使用高阶函数和函数装饰器来设计纯函数，实现通用的功能。这种思路是对过程封装，所以叫做过程抽象，它是函数式编程的基础。

## 小试牛刀

1. 如果你了解 react，你会发现 react-hooks 其实上就是纯函数设计。你可以思考一下，如果引入了它，能给你的系统带来什么好处？
2. 我们在前端业务中，也会用到一些常用的函数装饰器，比如，节流 throttle 和防抖 debounce，你能说说它们的使用场景吗？如果让你实现这两个函数装饰器，你又会怎么做呢？

函数式编程的思想你都理解了吗？那不妨也把这节课分享给你的朋友吧。今天的内容就到这里了，我们下节课见！

提建议

## 更多课程推荐

# 数据结构与算法之美

为工程师量身打造的数据结构与算法私教课

王争

前 Google 工程师



立省¥40

破90000订阅特惠，到手价¥89

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 加餐三 | 轻松一刻：我想和你聊聊前端的未来

下一篇 加餐五 | 周爱民：我想和你分享些学习的道理

## 精选留言 (1)

写留言



ZZJ

2020-09-18

关于迁移API，老师提供了很好的思路，太感谢了。以往都是侵入式的修改。

