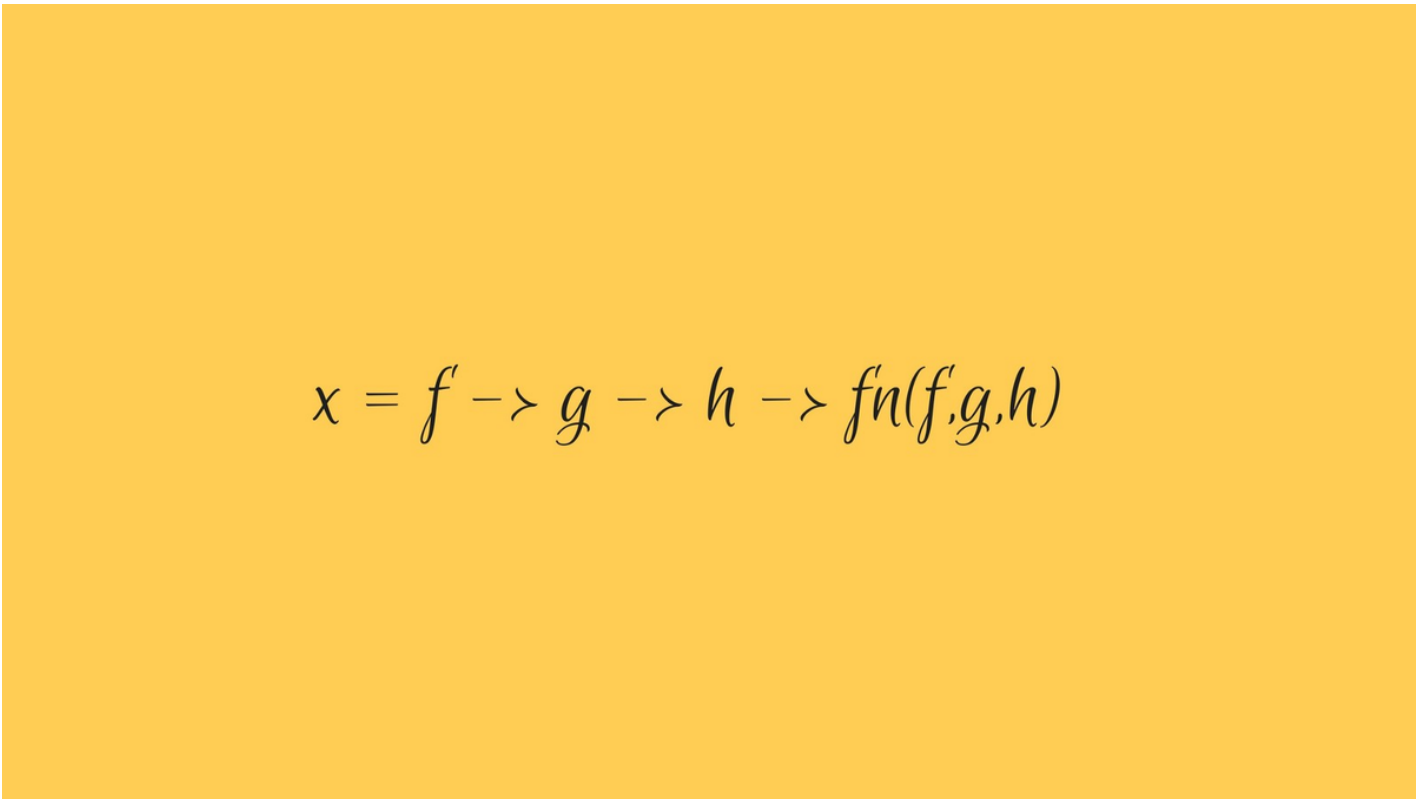
函数式编程简介

> 函数式编程(functional programming)一直是我学习编程以来的兴趣点，本文作为我长时间以来对函数式的一些经验和总结，希望能够给读到的的人提供一些有用的东西。本文期望能在读完后能知道为什么会存在函数式，函数式为什么会强调递归和函数，为什么会有Monad。很多复杂的东西，其实都不是天生的。
> 代码使用JavaScript ES6作为示例。



什么是函数式编程

wiki对于它的定义是:

> "**functional programming** is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data."

结合其它可以参考到的定义，可以总结为：

- 它是一种编程范式，和面向对象、结构式、逻辑式属于同级。
- **它强调避免在编程中进行状态修改而是数据转换。**
- 它能够将计算也就是常规编程语言的函数作为普通数据进行处理。

- 它的理论基础是 λ表达式(lambda calculus), 它是图灵完整的。

对比面向对象编程，面向对象通过抽象数据为对象，通过对象的状态和关系来进行编程，
而函数式强调无状态编程，编程是数据转换，接受输入，给出输出。

学习它能给你什么

- 更好的代码可读性和可维护性
- 易于抽离重用
- 易于测试
- 让你重新思考编程

它不能给你什么

- 你的算法并不能因此而获得直接的提高，函数式只是一种编码风格，它所表达的是如何合理的组织代码，如果你不能在oo中实现最短路径，用fp也不能。

应该学习它吗

如果想要扩展知识面，了解为什么当下许多语言都添加了lambda表达式，都开始添加对函数式的支持，那么你应该学习它。因为如果你有一把锤子，看什么都是钉子。如果只了解面向对象，那么很多

可以以更简单的方式解决的问题，你就没有必要用class重新new一个对象进行处理，尤其是在你的语言已经有可用工具的情况下。多学点东西终归是好的。

本质上来说，函数式只是一种风格，它并不会带来编码水平上的提高，也不是一种能够完美解决一种问题的技术方案，可以说它只是为了提高代码可读性和可维护性而与其它范式并存的一种选择，如果你对当前的编码风格没有什么问题，那么其实你并不需要它。

如何进行函数式编程

无状态编程

如果编程的变量都是不可变，那么很多情况就会发生变化。
变量不可变，就不存在“状态”、”变量“这种说法，编程所能做的，就只能是进行数据转换。
因为没有了可变状态，那么很多问题的定位就很容易，因为不需要关注数据变化，同时多线程的race condition也可以解决，因为不用去修改状态了。

可以通过代码来示例，尝试写一个排序数组的代码。

```
let list = [1, 3, 8, 9, 2, 2, 2];
list.sort();

function sort(list){
    for(let i = 0; i < list.length; i++){
        for(let j = i; j < list.length; j++){
            if (list[i] > list[j]){
                swap(i, j)
            }
        }
    }

    function swap(i, j){
        const puppet = list[i];
        list[i] = list[j];
        list[j] = list[i];
    }
}
```

变量不可变后，我们保留同样的实现逻辑进行改写：

```
let list = [1, 3, 8, 9, 2, 2, 2];
let listSorted = sort(list);

function sort(list){
    if (list.length === 0) return [];
    const minItem = getMin(list);
    const restList = remove(minItem, list);
    return [minItem, ...sort(restList)];

    function getMin(list, result = undefined) {
        if (list.length === 0) return current;
        let [item, ...restList] = list;
        let nextResult = result === undefined ? item :
                         item < result ? item : result;
        return getMin(restList, nextResult);
    }

    function remove(item, list) {
        if (list.length === 0) return [];
        let [head, ...restList] = list;
        if (head === item) {
            return remove(item, restList);
        } else {
            return [item, ...remove(item, restList)];
        }
    }
}
```

> 子函数的代码风格是为了隔离作用域，避免代码混乱，在sicp和重构都有相关提及，感兴趣可以深入了解。

对比可以首先感受到的是，代码变长了。
但是实际上，代码变短了，之所以感觉代码变长是我们没有在真正的编程中实际使用它，上述代码写了很多子函数，实际上 remove 和 getMin 都可以被很多其它地方所**复用,** 并且对于list来说它们是非常

常用的，它们甚至都不需要做改变就可以直接提取出作用域抽象到一个工具库中。

> 实际上，随着你编写越来越多的函数，为了减少重复工作量你会识别函数之间的共性进行抽象，你会不停的扩展自己的工具库，这些函数本质上和语言本身提供的地位没有什么不同，某些层面上，你是在扩展语言

另外可以看出来的一点是，到处都在用递归，而且实际上，代码的逻辑虽然都是冒泡排序，但是部分代码使得效率稍微变低了，T(n) 和 T(2n) 的差别。
由于失去了状态，我们不得不用递归进行循环的数据处理，getMin通过函数参数来实现状态保存，这也就是**尾递归**。

还有一点就是，代码更容易理解了，可读性无疑是比之前高的，而且由于函数都是做数据转换，所以更加容易测试了，并且更加容易维护。可以对比下函数的调用部分。

- 第一种

```
let list = [1, 3, 8, 9, 2, 2, 2];
list.sort();
```

- 第二种

```
let list = [1, 3, 8, 9, 2, 2, 2];
let listSorted = sort(list);
```

比起改变原有数据，通过调用函数进行重新建立变量进行存储，代码之间依赖关系更加清晰了。如果在代码进行多次修改的情况下，中间插入了数行代码，想要某一行获取排序过后的list，你就不得不考虑到中间有可能有其它代码修改了list的情况：

```
let list = [1, 3, 8, 9, 2, 2, 2];
doSomethingA(list);
//... and some other code
list.sort();
doSomethingB(list);
... // so many lines
```

但是如果变量不可变，就不会有这种情况发生，listSorted永远是listSorted。

函数作为可处理数据

函数式的另一个特点是，**函数是一等公民。**

这么说给人一种很难以理解的感觉，简单来说就是：**函数和其它数据一样，可以被赋值给其它变量，也可以像其它变量一样被传递给函数。**
个人认为，这一点决定了一个语言，能否进行函数式编程。

一旦函数和其它数据是等同地位，那么代码的灵活性就变的极其大。
比如我们上述的sort排序，目前只能是给数字列表排序，而且都是升序。
我们可以通过增加参数，来让它支持降序，我会省略一些子函数的实现，避免不必要的表达。

```
function sort(list, ascendingOrder){
    if (list.length === 0) return [];
    ~~const minItem = getMin(list);~~
    const minItem = ascendingOrder ? getMin(list) : getMax(list);
    const restList = remove(minItem, list);
    return [minItem, ...sort(restList, ascendingOrder)];
}
```

问题是，以后我们可能需要支持更多option，比如支持给以下数据通过id排序

```
[{userId: 1, name: 'xx'}, {userId: 2, name: 'xxx'}, ...]
```

另外如果都是通过增加参数来处理，代码中会有很多if判断，那会使得代码的可读性和复杂度上升。

解决的方法就是，将函数的部分逻辑暴露出去，我们将获取list中具体用于排序值的逻辑交给调用者实现，就可以实现给各种复杂类型的数据进行排序。

```
function sort(list, getKey = id) {
    if (list.length === 0) return [];
    const minItem = getMinBy(getKey, list);
    const restList = removeBy(getKey, minItem, list);
    return [minItem, ...sort(restList, getKey)];


    function getMinBy(getKey, list, result = undefined) {
        if (list.length === 0) return current;
        let [item, ...restList] = list;
        let nextResult = result === undefined ? item :
                         getKey(item) < getKey(result) ? item : result;
        return getMin(restList, nextResult);
    }
```

```
15
16      function removeBy(getKey, item, list) {
17          if (list.length === 0) return [];
18          let [head, ...restList] = list;
19          if (getKey(head) === getKey(item)) {
20              return remove(item, restList);
21          } else {
22              return [item, ...remove(item, restList)];
23          }
24      }
25  }
26
27  function id(n) {
28      return n;
29  }
```

> id函数接受一个数原封不动的返回它，大多数开源工具库都提供它，它就像number中的0，终归是有用的

相比之前的代码，现在的代码不再直接获取list中每一项，而是暴露了这个过程给外部进行实现，这个时候我们就可以类似这种的数据进行排序。

```
1  let list = [{userId: 1, name: 'xx'}, {userId: 2, name: 'xxx'}, ...]
2  const sortedList = sort(list, obj => obj.id);
```

实际上，我们还可以给它暴露给多的逻辑，比如如何进行大小的对比，JavaScript原生就是这么干的

```
1  list.sort((a, b) => a - b)
```

以上的抽象方式，另一个名字叫**Inverse Control**, 也就是控制反转，上述的编程方式是对它最灵活的体现。

Map

在编程中经常做的一件事，就是要对列表数据进行处理。
比如，将list中每一项都翻倍。

```
1  const list = [1, 2, 3, 4];
2  for(let i = 0; i < list.length; i++) {
```

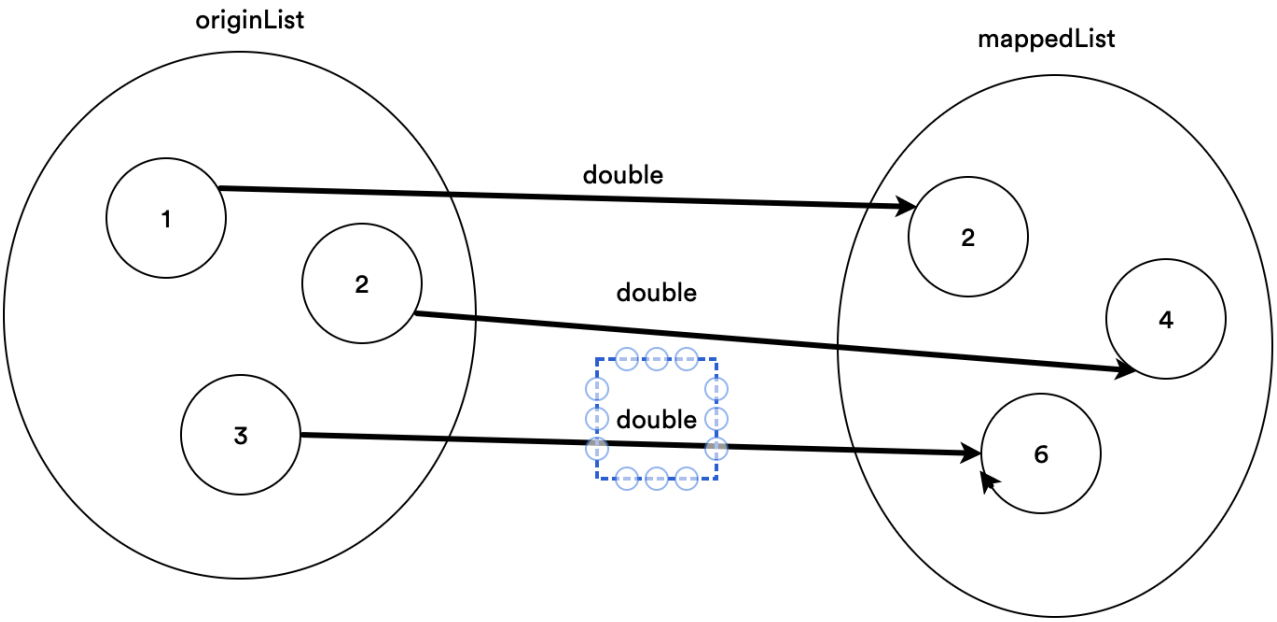
```
    list[i] = 2 * list[i];
}
```

更多时候，我们做的可能不只是数据的double，比如，通过userId去获取userName：

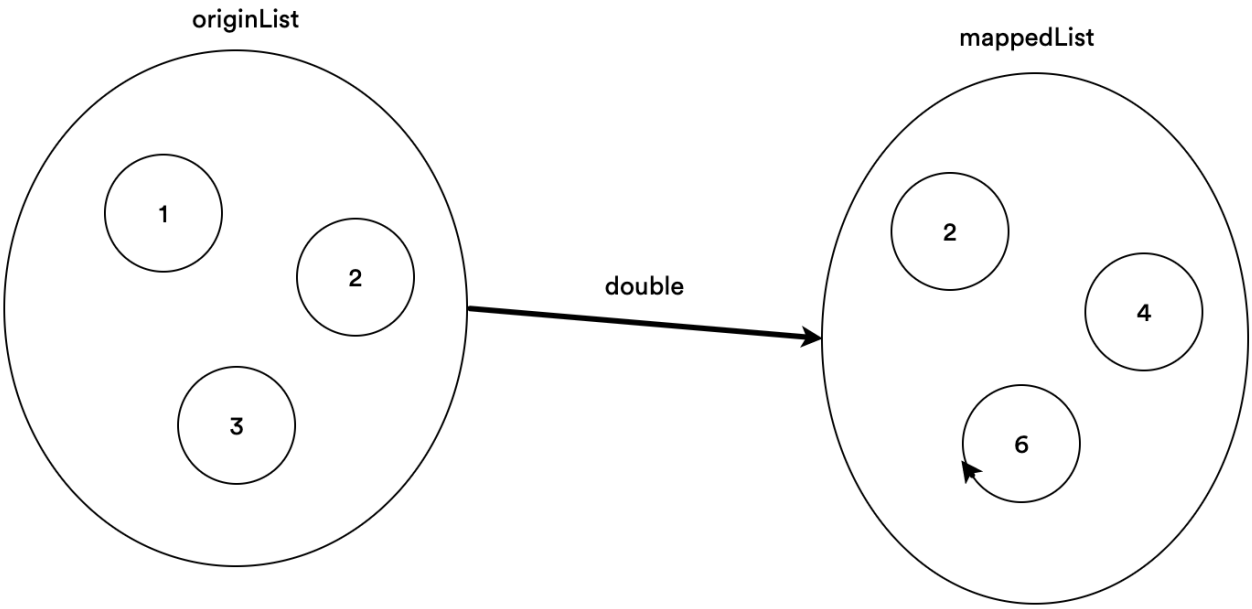
```
const users = {
    '1': 'hello',
    '98': 'world',
    ...
}

let userIdList = [1, 2, 99];
for(let i = 0; i < userIdList; i++) {
    userIdList[i] = users[userIdList[i]];
}
```

上述操作都有一个共性，都是在对数据进行映射，只是内部的元素发生改变，从一个list转换为另一个list：



对list的每个元素都进行double操作，我们可以简化改图，相当于对list做double操作：



那么double就可以抽象出来，做一个逻辑抽离：

```
const list = [1, 2, 3, 4];
for(let i = 0; i < list.length; i++) {
    list[i] = double(list[i]);
}

function double(x) {
    return x * 2;
}
```

同理，可以用于其它例子中：

```
const users = {
    '1': 'hello',
    '98': 'world',
    ...
}

let userIdList = [1, 2, 99];
for(let i = 0; i < userIdList; i++) {
    userIdList[i] = findUserNameByUserId(userIdList[i], users);
```

```
}

function findUserNameByUserId(userId, userMap) {
    return userMap[userId];
}
```

通过抽离子函数，代码的可读性更高了，并且做到了指责分离，一个函数做一件事。
上面的代码中double和findUserNameByUserId都是很简单的操作，这么短的代码为什么要分离，可以给出合理的理由：

- 可读性于和可扩展性，大多数实际编码中，对于list中数据的处理不会只有一行double这么简单，全部写在for循环中会代码结构变得混乱。
- 抽离出的子函数，总是可以发现它是可重用的。

可以更进一步，抽离出这一list映射的模式，也就是map:

```
function map(fn, list) {
    if (list.length === 0) return [];
    let [x, ...xs] = list;
    return [fn(x), ...map(fn, xs)];
}
```

然后用map去改写原有的代码：

```
const list = [1, 2, 3, 4];
for(let i = 0; i < list.length; i++) {
    list[i] = double(list[i]);
}
const doubledList = map(double, list);
```

我并不会说，代码越短越好，但是大多数情况下，简短的代码都更易读。
现在，我们只需要给map传不同的处理函数fn和不同的list，就可以少写很多代码。

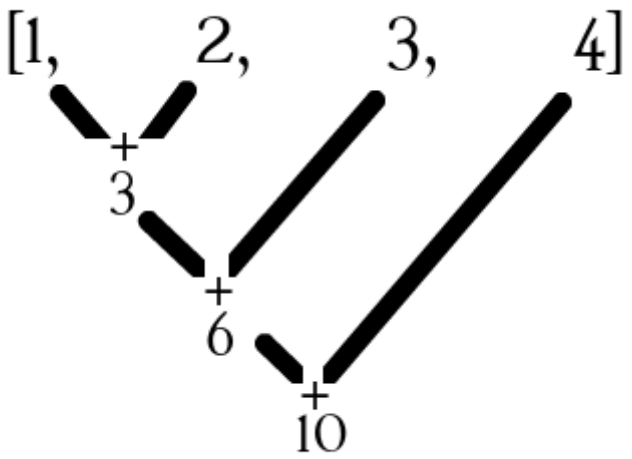
Reduce

代码中经常用到的另一种模式，就是不同结构之间的转换。
比如，求list的和:

```
const list = [1, 2, 3, 4];
let sum = 0;
for(const item of list){
```

```
    sum += item;
}
```

这样的情景，map是不适用的，但是我们可以通过其它逻辑进行抽象。
一种理解是，这是一种数据的折叠操作。



相当于，我们确定一个值作为初始值，后续不断用list的中每个值来修改它，直到结束。
比如：求和的初始值是0

第一次：扫描的元素是1，执行1+0，初始值更新为1
第二次：扫描的元素为2，执行1+2，初始值更新为3
第三次：扫描的元素为3，执行3+3，初始值更新为6
第四次：扫描的元素为4，执行6+4，初始值更新为10
结束

初始值可以换个名字，积累值，也就是accumulate
可以看出我们需要一个函数，参数为上一次的积累值 和 当前处理的元素，返回新的积累值

```
function add(acc, current) {
    return acc + current;
}
```

之后，便可以这样修改原代码

```
const list = [1, 2, 3, 4];
let sum = 0;
for(const item of list){
   sum = add(sum, item);
}
```

```
6
```

同理，这里的add也和之前的double一样，都是可以根据需求灵活改变的
为了便于其它地方适用，这儿也和map一样抽象出一个函数，也就是reduce，或者叫fold

```
function reduce(fn, acc, list) {
    if (list.length === 0) return acc;
    const [x, ...xs] = list;
    return reduce(fn, fn(acc, x), xs);
}
```

来改写下之前的代码：

```
const list = [1, 2, 3, 4];
const sum = reduce(add, 0, list);
```

reduce的灵活性远高于map，比如将list转换为hash table:

```
const userList = [{userId: 1, userName: 'hello'}, {userId: 2, userName:
'world'},...];
const userIdToNameMap = reduce(deal, {}, userList);

function deal(acc, user) {
    return {
        ...acc,
        [user.id]: user.name,
    }
}
```

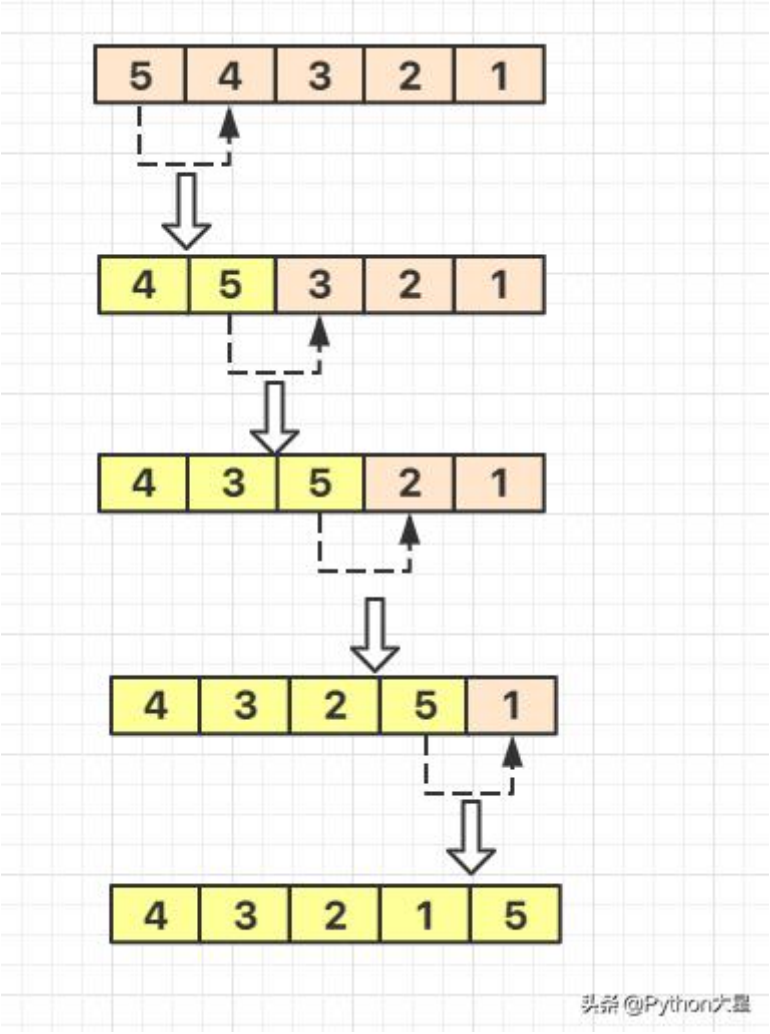
Declarative programming

函数式的代码，都有一个特点，那就是声明式。
由于不能修改状态，只能通过抽象函数来进行数据转换，那么就不得不写出这样的代码。

```
const list = [1, 2, 3, 4];
const doubledList = map(double, list);
```

上述代码直接就可以读出，将list进行double的转换，它并没有提及细节，比如：double是什么？double是如何执行的？
这就是声明式，只强调要什么，不强调具体如何实现。
事实上，由于无状态，函数式的代码大多如此（但是如果你非要在lisp中写c的风格，实际上也没有人能阻止你），可以说，函数式是自己选择了声明式，而不是声明式选择了函数式。

冒泡排序的逻辑是什么，我们可以简单写一下。
对于一个list，找出其中最小/最大的元素，可以确认其在返回数组中的位置。
从list中除去这个元素，对剩下的列表重复此过程，找到剩下的位置的元素。



对比代码可以发现，代码几乎就是上述的直接翻译。

```
function sort(list){
    if (list.length === 0) return [];
    const minItem = getMin(list);
    const restList = remove(minItem, list);
    return [minItem, ...sort(restList)];
}
```

这就是声明式！！

point-free

我们继续接着以上的代码风格往下走...
我们有如下数据:

```
const userList = [{userName: 'xxx', id: 3}, {userName: 'ding', id: 5}, ...];
const firendsRelations = [{
        userId: 3,
        firendsIdList: [1, 2, 3]
    },
    {
        userId: 2,
        firendsIdList: [4, 5, 6]
    },
];
```

需求是：给我们一个getCurrentUserName来获取当前用户的userName，要求获取他的firendList。
常规的指令式，可以这样写：

```
let currentUserName = getCurrentUserName();
let currentUserInfo = null;
for (const userInfo of userList) {
    if(userInfo.name === currentUserName) {
        currentUserInfo = userInfo;
        break;
    }
}
let currentUserId = currentUserInfo.userId;

let currentUserFriendList = null;
for (const relation of firendsRelations) {
    if (relation.userId = currentUserId) {
        currentUserFriendList = relation. firendsIdList
    }
}
```

但是我们是函数式，无状态下，这样写：

```
const currentUserName = getCurrentUserName();
const currentUserInfo = getCurrentUserNameUserInfo(currentUserName, userList);
const currentUserId = currentUserInfo.id;
const currentUserRelation = getCurrentUserRelation(currentUserId,
firendsRelations);
const currentUserFriendList = currentUserRelation.friendList;

function getCurrentUserNameUserInfo(userName, userList) {
    return find((userInfo) => userInfo.userName === userName, userName);
}

function getCurrentUserRelation(userId, relations) {
    return find((relation) => relation.userId === userId, relations);
}

function find(preFn, list) {
    return reduce((acc, item) => preFn(item) ? item : acc, null, list);
}
```

问题是，我们多了很多无必要的中间变量，currentUserInfo, cuurentuserId等等，实际上都是只用了一次，完全没有必要声明它。
所以，函数可以这样写:

```
const currentUserFriendList = getCurrentUserRelation (
getCurrentUserNameUserInfo(
getCurrentUserName(), userList).id, firendsRelations).friendList;
```

这样的代码，...确实是短了，但是并没有可读性，如何优化呢?
首先，将取user.id, user.firendList这样的操作写成函数

```
function getIdProp(obj) {
    return obj.id;
}

function getFriendListProp(obj) {
    return obj.firendList;
```

```
}
```

然后改写它

```
const currentUserFriendList = getFriendListProp(
getCurrentUserRelation (
getIdProp(
getCurrentUserNameUserInfo(
getCurrentUserName(), userList), firendsRelations)));
```

> 很多函数库比如ramda和lodash都提供来add、prop等类似工具，如果你疑惑为什么明明可以写成1 + 1和user.name缺非要用add(1, 1) 和 prop("name", user), 这就是原因，不打断数据处理流。

然后就是对于userList和firendsRelations这种参数的处理，我们可以使用高阶函数来处理
改写getCurrentUserNameUserInfo和getCurrentUserRelation，为了形式简洁和统一，将原有function声明改为使用lambda箭头：

```
const getCurrentUserNameUserInfo = (userList) => (userName) => {
    return find((userInfo) => userInfo.userName === userName, userName);
}

const getCurrentUserRelation = (relations) => (userId) => {
    return find((relation) => relation.userId === userId, relations);
}
```

我们的目的是为了让函数能够通过只接受部分参数来生成更多通用函数，事实上，这就是**柯里化**。

在用它来改写目标代码：

```
const currentUserFriendList = getFriendListProp(
getCurrentUserRelationWithUserId (
getIdProp(
getCurrentUserNameUserInfoWithUserName(
getCurrentUserName())));

const getCurrentUserNameUserInfoWithUserName =
getCurrentUserNameUserInfo(userList);
const getCurrentUserRelationWithUserId = getCurrentUserRelation(firendsRelations);
```

如果在多来几个处理函数，可能就会变成（）））））））了，简直是在写lisp :|。
为了减少（），我们在抽象一个compose

```
function compose(...fns) {
    return (args) => reduceRight((args, fn) => fn(args), args, fns)
    // reduceRight和之前reduce实现基本一致，仅仅是扫描方向不同，为从后向前处理ilst
}
```

用它来改写目标代码：

```
const getCurrentUserFriendList = compose(
    getFriendListProp,
    getCurrentUserRelationWithUserId
    getIdProp
    getCurrentUserNameUserInfoWithUserName
);
const currentUserName = getCurrentUserName();
const currentUserFriendList = getCurrentUserFriendList(currentUserName);
```

可以看出，这样的写法使得数据的处理流程非常的清晰。
这种组合函数，函数之间参数直接传递无需中间参数的风格，就是point-free！！

Monad

关于函数式，我们已经了解了很多内容，但是还有极其关键的一点被遗漏了。
副作用和异常处理

我们可以用compose来组合数据处理，但是我们并没有考虑参数报错的情况。
针对每一个阶段数据流处理的出错，我们都可能会有不同的处理，在指令式中多写几个if就可以，但是函数式中该如何处理?

处理的思路是：

- 将数据分为一类，一类是常规数据，一类是错误数据。
- 函数对数据进行识别，如果是常规数据，就正常执行，如果是错误数据，就不继续进行处理。
- 有专门的错误处理函数来应对错误数据的处理。

但是，如何让所有的函数都支持判断常规和错误数据呢。
你不可能写所有的函数在开始对他们进行一个同样的if判断，那违背了编程原则。
答案是函数，用函数处理函数，同时也用函数处理数据。

```
const Box = (left, right) => {
    return {
```

```
        map: (fn) => {
            if (err) return Box(left, right);
            try {
                return Box(fn(left));
            } catch (err) {
                return Box(left, err);
            }
        },
        catch: (fn) => {
            fn(right);
            return Box(left, right);
        },
        get: () => left,
    }
}
```

如下，我们写了一个Box，可以理解为oop当中的constructor，某种方面上他们完全等同，只是fp拥有来操作运算过程的能力。
Box接受两个参数，left代表常规数据，right代表Error。
使用时不需要传递right，只需要传递left就可以。

可以直接用代码来说明如何使用：

```
const a = someData;
const value = Box(a)
      .map(processA)
      .map(processB)
      .map(processC)
      .map(processD)
      .catch((err) => dealError(err));

```

我们将数据a封装到Box中，如果数据流正常处理，那么符合预期。
如果出现异常，catch将会统一处理。

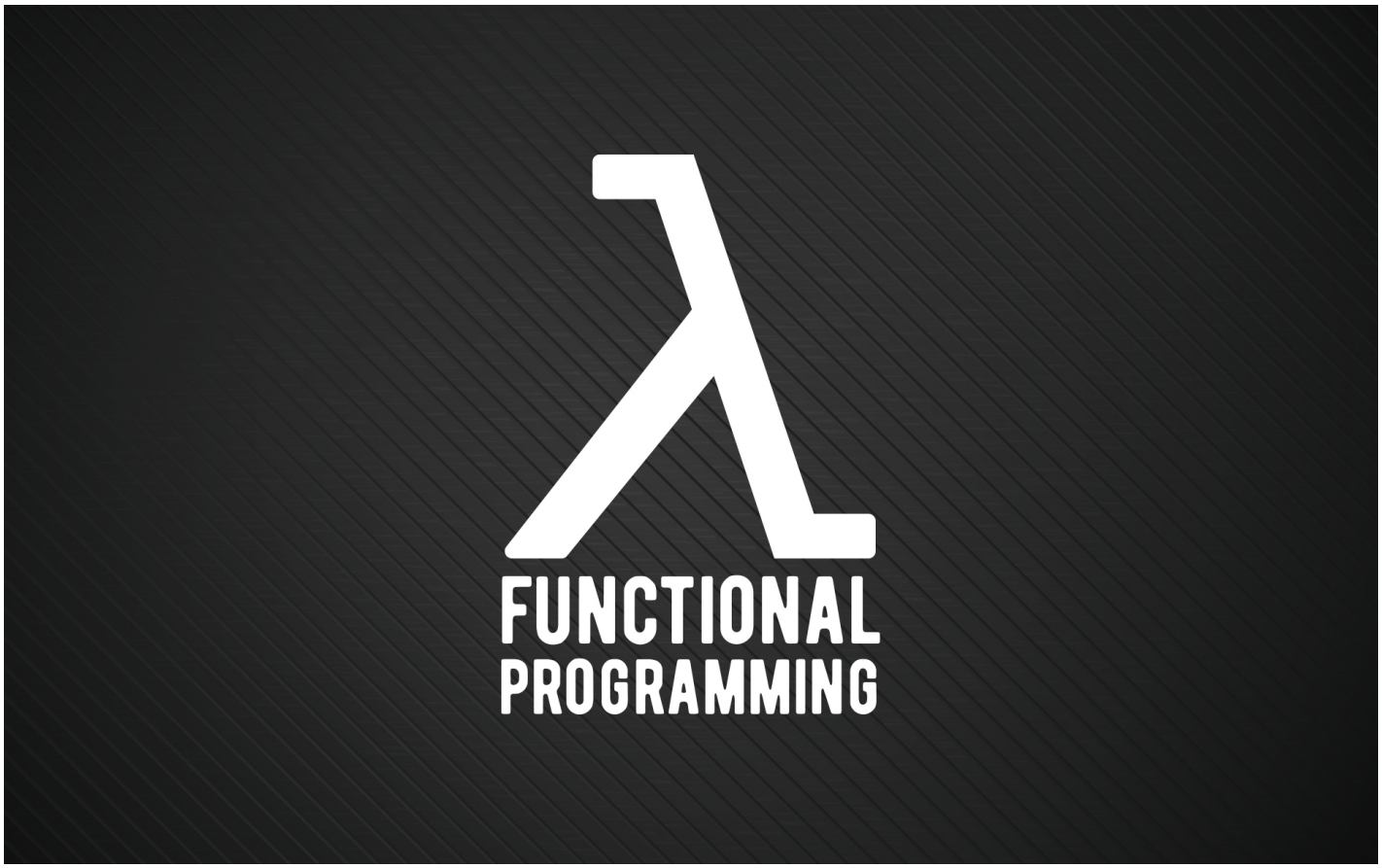
所有关于异常的处理，都被放到catch中，使得代码的副作用可以在一处进行统一处理。
这是函数式的另一种思想：严格对待副作用。

> 副作用是指，编程中那些会影响外界的部分，比如IO，比如Error，比如doucement.write就是一种副作用生成函数。

完全无副作用是不现实的，尤其是程序的本质目的就是为了产生副作用，但是通过将副作用和正常数据处理进行隔离，可以让代码变得更加清爽。

这样处理副作用的数据结构，叫做Monad，上述的Box是用来处理Error，还有各种Monad用来处理IO、State等等。
Monad的另一个作用，是用来在不同的数据之间进行结构转换，链接函数，Box只实现了Map，但是大多数情况下还会有其它和Map类似的转换处理函数，如果我们除了Box还有Box2，就要考虑如何转换到Box2。

> Monad是建立在范畴论基础上的，如果你感兴趣，可以Category theory tutorial



总结

不知不觉，就写了这么多，从无状态编程、lisp的函数作为一等公民、声明式、组合编程、Tail Recursion、point-free，在到最后的haskell中的Monad，我们已经对函数式有了一个比较全面的了解，在常规语言去应用它，已经完全足够了。
后续有机会的话，我还会在介绍下 y combinator 以及 Continuation-passing style。

Welcome to functoinal land!