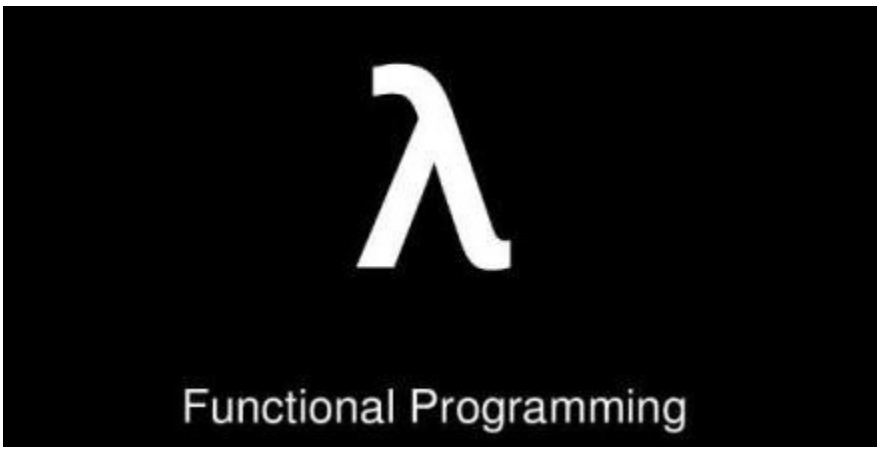
本章介绍：纯函数，闭包，高阶函数，柯里化，编程规范



什么是纯函数?

什么是闭包？为什么会产生闭包？如何产生的?

什么是高阶函数?

什么是函数的柯里化?

什么是函数式编程？声明式编程？命令式编程?

问题1：什么是纯函数?

```
当我们创建一个函数，如果这个函数具备以下两个特点：
1）  这个函数指定了输入与输出。并且当调用参数相同时
     这个函数永远返回相同的结果，并且不依赖于任何外部状态或数据。
```

```
2） 这个函数不会发生任何突变（mutation）或产生任何副作用（effect）。
```

当满足以上两点我们就称这个函数为【纯函数（Pure Function）】。 换言之，如果使用一个函数时候，不使用他的返回值但是确有意义或作用的话，说明这个函数是【非纯函数】。

问题2：什么是闭包？为什么会产生闭包？如何产生的?

闭包就是一个函数包含着对另一个函数的引用

在创建函数的时候，js 会产生相应的执行环境，在执行环境里会生成活动对象、作用域链等。

执行环境下，js 首先会利用作用域进行变量提升，然后会按顺序进行执行，此时会对变量进行赋值等操作，

执行完毕后会把执行环境从执行环境栈中弹出。
但是由于有可能一个函数包含着对另一个活动对象的引用

导致被引用的活动对象一直没有被释放，这就是js闭包产生的原因。

因此由于 js 本质是everything is object，在互相引用的过程中就会产生闭包。

由此可以延展：

```
1    1）ES6 let 声明之所以会产生“暂存死区”（既 let 代码块以上无法使用 let 声明的变量）的现象，
2    也是由于 let 在执行环境中并没有变量提升的过程。
3    2）ES6 const 无法再次被赋值，也是因为它只有声明阶段，没有赋值阶段。所以 const 声明的变量
```

也无法再次被赋值。

因此闭包具有以下特点：

```
1    1）函数嵌套函数
2    2）函数内部可以引用外部的参数和变量
3    3）参数和变量不会被垃圾回收机制回收
```

问题3：什么是高阶函数?

当开发初期，我们使用函数对业务逻辑和运算进行封装，使得函数可以根据我们的入参进行相应逻辑运算与转换。但是如果当一个函数的参数也是一个函数时，那么这个函数的处理业务的复杂度增加，使得其成为一个**高阶函数**。我们可以通过一个例子来观察高阶函数区别于普通函数的特点。

我们希望过滤这个数据，找到价格高于 30 元产品，首先我们先使用一些常用的数组操作来完成：

```
const data = [
    { id: 1, food: "手撕面包", price: 34 },
    { id: 2, food: "牛奶", price: 20 },
    { id: 3, food: "拿铁", price: 26 },
    { id: 4, food: "卡布奇诺", price: 28 },
    { id: 5, food: "馥芮白", price: 40 },
    { id: 6, food: "摩卡", price: 32 },
    { id: 7, food: "耶加雪啡", price: 128 },
];

// 不使用函数式编码
// way1
let one = [];
for (let i = 0, len = data.length; i < len; i++) {
    if (data[i].price > 30) {
        one.push(data[i].food);
    }
}
console.log(one);    // [ '手撕面包', '馥芮白', '摩卡', '耶加雪啡' ]

// way2
let two = data.map(item => {
    if (item.price > 30) { return item.food }
}).filter(l => typeof l != 'undefined');
console.log(two);    // [ '手撕面包', '馥芮白', '摩卡', '耶加雪啡' ]

// way3
let three = data.filter(l => l.price > 30).map(l => l.food);
```

```
29 console.log(three);  // [ '手撕面包', '馥芮白',
   '摩卡', '耶加雪啡' ]
30
```

上面的 way2, way3 我们都是在 map 内部直接创建了函数去处理业务。事实上ES6中 map, filter, reduce, some, every 也都是 **高阶函数** ，因为他们也是接受一个函数，根据函数执行返回结果，即

```
1 const map = list => order => list.map(order)。
```

当我们使用上面的三种方式去过滤数据的时候，可以发现，我们关注点在于处理什么数据？按照什么条件处理？这也是其弊端所在，接下来我们对条件和数据进行抽象：

```
1  // way4
2  // order 作为条件传入，等待 data 的输入
3  const select = order => data =>
   data.reduce((prev, next) => {
4     if (order(next)) prev = [...prev, next];
5     return prev;
6  }, []);
7
8  /// 业务部分
9  const condition = data => data.price > 30;
    // 抽象函数条件
10 const getDataByCondition = select(condition);
   // 生成数据获取函数
11 const getConditionResult =
   getDataByCondition(data);
12
```

```
console.log(getConditionResult.map(l =>
l.food));
//  [ '手撕面包', '馥芮白', '摩卡', '耶加雪啡' ]
```

对比上面的三种实现，在 reduce 高阶函数参与之后，我们将条件和数据进行抽象，将过滤函数的关键点都提取了出来，为此我们创造了一个能够接受其他函数进行逻辑处理的**高阶函数**：**select**。它使得我们的关注点，从逻辑的判断，**转换成了函数的编写与合理化的命名**。多种函数互相组合，互相赋能，这也是高阶函数的魅力所在。

但仔细观察上面的实现（way4），似乎也存在问题，reduce 作为一个高阶函数，应该可以进一步抽象，来应对跟多场景，因此我们可以进一步拓展：

```
// way 4 改版

// 抽象 reducer 内部函数
const select = order => data =>
data.reduce(order, []);
// 定义 条件生成函数
const conditionHandler = condition => (prev,
next) => {
    if (condition(next)) prev = [...prev,
next];
    return prev;
}

// 业务部分
const condition = item => item.price > 30;
        // 定义条件
```

```
const filterByPrice = conditionHandler(condition);    // 生成过滤函数
const getDataByCondition = select(filterByPrice);     // 生成数据处理函数

console.log(getConditionResult.map(l => l.food));
//  [ '手撕面包', '馥芮白', '摩卡', '耶加雪啡' ]

const condition = item => item.food === "摩卡";
      // 修改条件
console.log(getConditionResult.map(l => l.food));     // [ '摩卡' ]

```

到这里，我们针对筛选这个业务场景，抽象了两个可复用的高阶函数，select 和 conditionHandler。我在使用这两个函数去处理筛选业务时，我的关注点在于如何合理化命名函数，理解业务并创建对应的条件函数。由此我们发现 **函数式编程** 的基本思想就是这种高度抽象的编程规范。而 **高阶函数** 则是在这种编程思维下所使用的一种编码方式而已。

通过上面的例子，我已经能理解 **高阶函数** 其实就是接受其他函数作为入参的一种函数而已。

问题4：知道了闭包和高阶函数，那么什么是函数的柯里化呢？

之前对柯里化有一定了解的朋友一定知道柯里化函数的特点或者作用:

```
1) 参数可复用
   2) 提前确认
   3) 延迟运行
```

我们通过一个简单的例子来理解这三个特征： 如何实现一个加法函数，使其可以接受任意个参数和组合形式进行加法运算，即

- add(1)(2)(3) // 6
- add(1, 2, 3) // 6
- add(1, 2)(3) // 6

代码实现：

```
function curry(fn, scope, ...args) {
    let
        len = fn.length,
// 拿到 函数 的 参数长度
        prex = args,
// 保存上一次的 prex
        context = scope;
// 保存作用域
    let newFn = (...rest) => {
        let last = prex.slice(0).concat(rest);
// 合并入参
        if (last.length < len) {
            return curry.call(this, fn,
context, ...last);  // 继续柯里化
        } else {
            return fn.call(context, ...last);
            // 获执行函数
        }
```

```
    }
    return newFn;
}

function add(a, b, c) {
    return a + b + c;
}

let curryAdd = curry(add, add);
console.log(curryAdd(1)(2)(3)); // 6
console.log(curryAdd(1, 2, 3)); // 6
console.log(curryAdd(1, 2)(3)); // 6
```

上面的代码我加了注释，可以看到，通过curry **高阶函数**，当传入参数不足的时候，我们利用闭包的特点，保存之前入参，并且返回一个新的函数来继续等待接收参数，以达到**参数复用** 和 **延时执行** 的目的。可以看到函数经过柯里化之后，对简单的 a + b + c 这个过程进行了抽象，为这个简单的 + 法操作进行赋能，让你可以控制每一个变量并根据不同的情况进行函数的简单函数的复杂抽象来应对更多的情况，从而达到 **提前确认** 的特点。

> 由此对于函数的柯里化，我们已经有所领悟。即 **柯里化(Currying)就是将需要多个参数的函数转换为一个函数的过程，当提供较少的参数时，返回一个等待剩余参数的新函数。**这就是函数的柯里化。

上面的加法还可以继续拓展，如果想支持无限个参数进行加法，应该怎么做呢?

- add(1,2)(3)(1)(2) // 9
- add(1)(1)(1)(1)(1)...(n) // n * 1

我们需要对上面的 curry 和 add 函数进行改造，来应对这种需求

```
function curry(fn, scope, ...args) {
    let
        prex = args,              // 保存上一次的参数
        context = scope;          // 保存作用域
    let newFn = (...rest) => {
        let last = [...prex, ...rest];  // 合并入参
        return (                  // 当没有入参时，执行函数，否则继续柯里化函数
            rest.length > 0 ?
                curry(fn, context, ...last) :
                fn.call(context, ...last)
        )
    }
    return newFn;
}

function add(...args) {
    return args.reduce((last, next) => last + next, 0);
}

var curryAdd = curry(add);
console.log(curryAdd(1, 2, 3)());           // 6
console.log(curryAdd(1, 2)(3)());           // 6
console.log(curryAdd(1)(1)(1)(1)(1)());     // 5
```

```
console.log(curryAdd(1, 1)(1, 1)());          //
4
```

这样就实现了多参版本的 add 方法。需要关注的是，在实现多参版本时，我们对 add 函数进行改造，我们使用了 **reduce** 这个**高阶函数**,**高阶函数**和**柯里化**的结合，使得我们不必关注函数接受的每个参数，而专注于为函数进行赋能。这也是函数式编程的核心所在（专注于IO);

当我们了解柯里化之后，我们很容易理解 **bind** 函数(绑定指针，返回一个等待执行的函数) 的实现与原理了：

```
// 使用：  fn.bind(this, ...args); |
Function.prototype.bind = fn....
Function.prototype.bindFn = function (context)
{
    let bindedFn = this;  // 拿到 bind 的函数
    let args =
Array.prototype.slice.call(arguments, 1); // 去
除构造函数，拿到入参
    let pendingFn = function (...rest) {
        let last = [...args, rest];
        return bindedFn.apply(context,
...last);
    }
    return pendingFn;   // 返回一个待执行的函数，
接受新的参数
}
```

到目前为止，我们在回顾一下到底什么是函数的柯里化？相信我们可以更好的理解柯里化(Currying)就是将需要多个参数的函数转换为一个函数的过程，当提供较少的参数时，返回一个等待剩余参数的新函数。

问题5：函数式编程？声明式编程？命令式编程？

到这里我们已经了解了 **纯函数** ， **闭包**，**高阶函数** 以及函数的 **柯里化** 的定义、本质，以及他们之间的关系；当我们看透了本质，针对某种业务场景进行抽象，或是希望编写出可复用、易测试、易维护的代码时，我们可能考虑高阶组件，高阶函数的使用。那么就可能需要在编程风格和架构设计上做出改变。

声明式编程与命令式编程的区别

```
蔬菜(类) => 做成菜(方法)，接受入参(各种菜)

蔬菜.做成菜(牛油果，各种蔬菜)；  // 沙拉
蔬菜.做成菜(胡萝卜，青菜，油)；  // 炒蔬菜
```

声明式编程关注点在于 **“我们需要得到什么”** ，这种编程方式只在乎做什么、要得到什么，抽象了实现细节，而 **命令式编程** 实现这种过程则更像是

```
洗干净(蔬菜)
```

```
2 混合(蔬菜,沙拉)
3 放入盘中(混合物)
```

可以发现 **命令式编程** 关注点在于 **“我们如何去做”**，更加在乎计算机执行的步骤，一步一步告诉计算机先干什么，再干什么。把细节按照人类的思想以代码的形式表现出来。这也是为什么命令式编程将直接导致代码难以维护、难以测试、难以复用的原因（部分业务场景）。

函数式编程与声明式编程的关系

在 javascirpt 中，我们将 函数 作为参数进行传递，创造复杂度更高，功能更加强大的函数。我们进行函数式编程，把函数作为 **“一等公民”**。通过纯函数, 递归, 内聚, 惰性计算, 映射等进行组合，使得代码在实现 **“要得到什么”** 这个过程中，获取更强大的抽象与计算能力。因此 **函数式编程** 是 **声明式编程** 的一部分。

程序语言的设计：

程序语言的设计核心本质是方法论，是设计一些好的抽象供人们使用，既然是设计就有很多trade off, 没有客观意义上的好坏之分。和程序语言打交道的有两类人：一类是设计和实现，另一类是程序员用之作为工具，这里只谈谈函数式语言对于程序员可以借鉴的部分。

函数式语言的理论模型是lambda演算，规则非常简单，只有三条：

<表达式>::= <标识符>

<表达式>::= (λ<标识符>.<表达式>)

<表达式>::= (<表达式> <表达式>)

他们为什么叫函数式语言呢？我们可以看下前面链接里的三条规则，它是没有赋值的，这和另外一个理论模型图灵机有着本质的不同。没有赋值，意味着数据不可变(immutable)，给定同样的输入有且只有唯一的输出。这就是我们数学里面对函数的定义，所以叫functional style。

数据不可变是一种很值得提倡的风格，因为数据不可变，这使得调试变得相当简单了，程序也更少出错。而且不可变的数据结构在多线程环境里也相对更容易实现一点。

但是这种狭义意义上“函数式”的风格带来了一些性能上的问题，因为不可变通常意义上意味着要做很多拷贝（内存分配）。怎么设计高效的不可变的数据结构有一个很有意思的问题， 可以参考Chris Okasaki 的博士论文 。这种风格的数据结构也是非常实用的东西，强烈推荐大家好好读下Okasaki的书（亚马逊上也有书卖，但博士论文写得更翔实一点)。

其实这种函数式风格是无关语言的，完全可以用java写出非常函数式风格的代码。但是函数式风格往往意味着非常多的内存分配，如果GC（垃圾回收）不是为这个量身定制的话，意味着非常糟糕的性能。早期的F#就有这个问题，相对OCaml慢了很多。（不知道后来有没有改进)。

前面有人问起函数式语言对GC的要求。函数式风格对GC的要求是很高的，如果GC不是为函数式风格的代码量身定制的，性能会有严重问题的。

函数式风格的代码一般会迅速产生大量的零时垃圾，需要GC能支持fast allocation, 而且一般GC 需要在读写的速度方面上做取舍，函数式风格的代码希望GC读更快，可以接受写更慢点，这些都需要一开始GC在设计方面做取舍。好的量身定制的支持函数式风格的GC会导致分配内存比原地修改还快，因为写有write barrier。

下面说说函数式语言另外一个有争议的特性:

- Curried calling convention (函数的柯里化)

Curry是一个数学家的名字，还有一个比较有名的概念(curry-howard polymorphism), 这里因为篇幅关系 就不提了。

我们可以看到前面的lambda 演算的三条规则里的第二条：所有的函数只有一个参数。这个设计很简洁, 而且表达能力更强，比如 map 函数我们写成

map (callback,list) = { ... 函数内部 }

它在编译器眼中展开实际上等价语义是这样的:

map = function (callback) { function (list) { ... 函数内部} }

用户可以自己决定一次提供一个或者多个参数, 而且它的中间结果再一定程度上也是有语义的，确实表达能力很强, 我们可以写这种风格的代码：

let f = map(callback)
f (list1), f(list2)

传统的语言一般只能写成这种

map(callback,list1), map(callback,list2)

但是它有两个个问题

- 每次 partial application（部分提供参数）产生的closure都会导致内存分配
- 需要运行时额外的bookkeeping 记录函数的arity

像有的函数式语言的编译器（如purescript) 就是这么实现的，结果当然是很慢的性能。
工业级的函数式编译器如ReScript会做静态分析，在编译的时候还是按照传统的多参数情况编译, 如

map = function(callback, list) { .. 函数内部}

然后根据map的具体使用情况来决定是否需要分配内存。静态分析可以做的很复杂，基本上对一阶函数可以消

除掉绝大部分overhead。但是对于高阶函数是很难的，比如:

f = function (callback,value) { callback(value)}

因为编译器此处不知道 callback 的任何信息，只能依赖其它优化内联才有可能做静态分析。

在非纯函数式语言里，这种风格还有一个问题就是语义光看类型不是很清楚的。

问题：函数式编程的特点：

1、函数式编程的自由度很高，可以写出很接近自然语言的代码。

前文曾经将表达式(1 + 2) * 3 - 4，写成函数式语言：

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

对它进行变形，不难得到另一种写法：

```
add(1,2).multiply(3).subtract(4)
```

这基本就是自然语言的表达了。再看下面的代码，大家应该一眼就能明白它的意思吧：

```
merge([1,2],[3,4]).sort().search("2")
```

因此，函数式编程的代码更容易理解。

2. 更方便的代码管理

函数式编程不依赖、也不会改变外界的状态，只要给定输入参数，返回的结果必定相同。因此，每一个函数都

可以被看做独立单元，很有利于进行单元测试（unit testing）和除错（debugging），以及模块化组合。

3. 易于"并发编程"

函数式编程不需要考虑"死锁"（deadlock），因为它不修改变量，所以根本不存在"锁"线程的问题。不必担心一个线程的数据，被另一个线程修改，所以可以很放心地把工作分摊到多个线程，部署"并发编程"（concurrency）。

请看下面的代码：

```
var s1 = Op1();
var s2 = Op2();
var s3 = concat(s1, s2);
```

由于s1和s2互不干扰，不会修改变量，谁先执行是无所谓的，所以可以放心地增加线程，把它们分配在两个线程上完成。其他类型的语言就做不到这一点，因为s1可能会修改系统状态，而s2可能会用到这些状态，所以必须保证s2在s1之后运行，自然也就不能部署到其他线程上了。

多核CPU是将来的潮流，所以函数式编程的这个特性非常重要。

总结

什么是纯函数？当一个函数指定输入后，输出永远相同，并且没有任何突变及副作用，那么这个函数就是一个纯函数

什么是闭包？闭包就是一个函数包含着对另一个函数的引用

什么是高阶函数？接受其他函数作为入参的一种高级函数

什么是函数的柯里化？将需要多个参数的函数转换为一个函数的过程，当提供较少的参数时，返回一个等待剩余参数的新函数

什么是函数式编程？声明式编程？命令式编程？命令式编程关注实现细节，声明式编程关注实现结果，弱化并抽象细节。函数式编程是声明式编程的一部分。

参考：
https://www.zhihu.com/question/471098472/answer/2029186480