一、函数式编程

- 命令式编程：详细的命令机器怎么（How）去处理一件事情以达到你想要的结果（What）

想要建立起函数编程的思维，首先要明白什么是命令式编程，所谓命令式编程，其实就是详细的命令机器怎么去处理一件事情，以达到你想要的结果。日常编程中用的最常见的一种编程方式就是命令式编程，例：

有一个需求，需要给数组每个元素加一

```
// 初级程序员
let arr = [1, 2, 3, 4];
let newArr = [];
for (var i = 0; i < arr.length; i++) {
  newArr.push(arr[i] + 1);
}
console.log(newArr); // [2,3,4,5]
```

所有的代码都是死的，不可复用

有经验的开发者为了提高代码复用性，会写一个函数来实现，如：

```
let newArr = (arr) => {
  let res = [];
  for (let i = 0; i < arr.length; i++) {
    res.push(arr[i] + 1);
  }
  return res;
}
console.log(newArr(arr)); // [2,3,4,5]
```

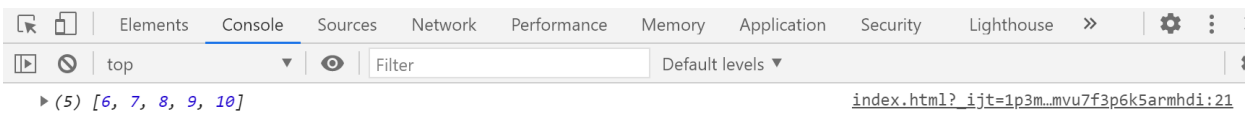
但是这仍然是一段不可复用的代码，如果需求改为将数组中的元素加5呢？或者其他运算呢？重新定义一个循环运算显然是不可取的。

以上两种编程方式都可以理解为命令式编程。

- 函数式编程是一种编程范式，是一种构建计算机程序结构和元素的风格，它把计算看作是对数学函数的评估，避免了状态的变化和数据的可变。

通过函数式编程的方式改写出来的代码是什么样的呢？

```
<script>
  let arr = [1,2,3,4,5];

  let fn = (arr, fn) => {
    let newArr = [];
    for (let i = 0; i < arr.length; i++) {
      newArr.push(fn(arr[i])); // 存入运算任务
    }
    return newArr;
  }

  // 任务
  let add = item => item + 5;

  // 运算结果
  let addResult = fn(arr, add);

  console.log(addResult);
</script>
```

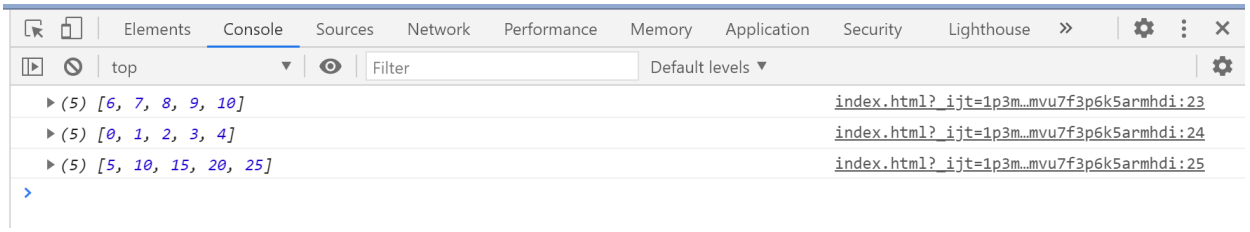


函数式编程主张将复杂的函数组合成简单的函数，将运算过程尽量写成一系列嵌套的函数调用，当我们再需要实现对数组元素的其他操作时，就无需将循环的逻辑再实现一遍，只需将其所要完成的任务传入即可。

例如我们需要再计算数组的每一项减去1或者乘以5

```
<script>
  let arr = [1,2,3,4,5];

  let fn = (arr, fn) => {
    let newArr = [];
    for (let i = 0; i < arr.length; i++) {
      newArr.push(fn(arr[i])); // 存入运算任务
    }
    return newArr;
  }

  // 任务
  let add = item => item + 5;
  let sub = item => item - 1;
  let mul = item => item * 5;

  // 运算结果
  let addResult = fn(arr, add);
```

```
let subResult = fn(arr, sub);
let mulResult = fn(arr, mul);

console.log(addResult);
console.log(subResult);
console.log(mulResult);
</script>
```



其实就是**将程序分解为一些更可重用、更可靠且更易于理解的部分，然后再将他们组合起来，形成一个更容易推理的程序整体，这就是函数式编程的思想。**

将一个任务拆分成多个最小颗粒的函数，然后通过组合的方式来完成我们的任务，这跟组件化的思想非常类似，将整个页面拆分成若干个组件，然后拼装起来完成整个页面，在函数式编程里面，**组合**是一个非常重要的思想，我们只需要完成我们想要完成的功能代码编写即可，无需关心其他的内部实现。

我们需要考虑的是在这些被忽略的实现里，是否会产生一些副作用的操作，由于这个原因，**函数式编程**产生了以下**两个重要的要求**

1. 纯函数
2. 数据不可变

二、纯函数

什么是纯函数?

- 如果函数的调用参数相同，则永远返回相同的结果，它不依赖于程序执行期间函数外部任何状态或数据的变化，必须只依赖其输入参数。

也就是说如果给定相同的参数，一个函数返回的结果也必然相同，它不会引起任何副作用。

非纯函数示例：

```
// 打折计算价格
let discount = 0.8;
const calculatePrice = (price) => price * discount;
let price = calculatePrice(200);
console.log(price);
```

calculatePrice函数使用了一个没有作为参数传递给函数的变量，函数内使用的discount变量，需要从函数外部去获取，这样导致了函数只能保证相同的输入，不能保证相同的输出，所以这不是一个纯函数。

```
let discount = 0.8;
let price = calculatePrice(200);
console.log(price); // 160

discount = 0.9;
price = calculatePrice(200);
console.log(price); // 180
```

当discount变化，比如它是0.8或者是0.9的时候，同样是输入两百作为参数，但是得到的结果却不一样，这样也就导致函数相同的输入不能保证有相同的输出。

纯函数示例：

```
// 纯函数
const calculatePrice = (price,discount) => price * discount;
let price = calculatePrice(200,0.8);
console.log(price);
```

它符合纯函数的定义，不依赖于任何外部数据，也不改变任何外部数据，相同的输入永远会得到相同的输出，并且没有副作用，所以它是纯函数。

三、函数副作用

什么是函数副作用?

函数内部与外部互动，典型的情况就是函数内部修改全局变量的值，产生运算以外的其他结果，函数副作用会给程序设计带来不必要的麻烦，给程序带来十分难以查找的错误，并且降低程序的可读性。

严格的函数式语言要求函数必须无副作用，在函数式编程里，函数没有副作用，变量都是不易改变的，而在面向对象中，可变状态和副作用都是十分常见，甚至是被鼓励的。

JavaScript中的函数可以访问、修改或定义在函数外的变量或者是参数，例如：

```
let a = 5;
let foo = () => a = a * 10;
foo();
console.log(a); // 50
```

foo函数内部修改了全局变量a，当函数执行调用以后，a的值是50，显然foo函数除了运算以外，还修改了函数外部的变量，这就是函数造成的副作用。

除了我们自己定义的函数可能产生副作用，原生的api也会产生副作用，例如：

```
let arr = [1, 2, 3, 4, 5, 6];
arr.slice(1, 3);  // 纯函数，返回[2,3]，原数组不改变
arr.splice(1, 3); // 非顺纯数，返回[2,3,4],原数组改变
arr.pop(); // 非纯函数，返回6，原数组改变
```

编写出没有副作用的程序是不可能的，而且有些副作用是不可避免且至关重要的，比如console，比如输入和输出，函数式编程者并没有消除所有的副作用，我们的**目标是尽可能的减少函数副作用。**

```
// 不纯的函数
const foo = (something) => {
  const dt = new Date().toISOString();
  console.log(`${dt}:${something}`);
  return something;
}
foo('hello');
```

foo函数违背了相同的输入总是得到相同的输出原则，我们通过依赖注入对foo函数进行改进

```
// 依赖注入
const foo = (d, log, something) => {
  const dt = d.toISOString();
  return log(`${dt}: ${something}`);
}

const something = '你好网易';
const d = new Date();
const log = console.log.bind(console);
foo(d,log,something);
```

所谓依赖注入，就是将不纯的部分提取出来作为参数，它有如下几个基本要求：

1. 将不纯的部分提取出来
2. 让不纯的代码远离核心代码
3. 让核心代码变为纯函数

这么做看起来很麻烦，也不是为了消灭掉副作用，主要是为了控制不确定性。其实就是把不确定性移动到了更小的函数中，这样我们就可以将副作用相关代码远离核心代码并且保存起来进行集中管理。

另一种控制副作用的思路是把产生副作用的部分保护起来。正确的使用纯函数可以产生更加高质量的代码，并且也是一种更加干净的编码方式，**无论是依赖注入还是其他的方案，处理副作用的原则都是将其带来的不确定性限制在一定范围内，让其他部分得以保持纯的特性。**

在JavaScript中想要保证函数无副作用这项特性，需要开发人员注意：

1. 函数入口使用参数运算，而不修改它
2. 函数内不修改函数外的变量
3. 运算结果通过函数返回给外部

四、可变性和不可变性

- 可变性是指一个变量在创建以后可以任意修改
- 不可变性指一个变量一旦被创建，就永远不会发生改变，不可变性是函数式编程的的核心概念，没有它程序中的数据流是有损的

可变性示例：

```
let data = { count: 1 };
let foo = (data)=>{
  data.count = 3;
}
console.log(data.count); // 1
// 调用foo函数
foo(data);
console.log(data.count); // 3
```


保证数据不可变示例：

```
let data = { count: 1 };
let foo = (data) => {
  let lily = JSON.parse(JSON.stringify(data));
  lily.count = 3;
}
console.log(data.count); // 1
// 调用foo函数
foo(data);
console.log(data.count); // 1
```