

什么是函数式编程

是相对于面向过程、面向对象编程而言的。

按照我的理解，这是将一个整个任务执行的过程看做一个流水线，单个函数就是流水线上一个结点。

流水线上，每经过一个结点，原材料就向着最终成品更进一步

而在函数式编程中，函数处理的是数据，每经过一个函数的处理，数据就朝着我们想要的形式更进一步

函数式编程的特点

1. 函数是一等公民
- 函数能够存在于任何变量能够去到的地方
- 比如可以作为返回值，可以作为参数
2. 惰性执行
- 只在需要的时候执行，不会产生无意义的中间变量。
3. 无状态
- 对于一个函数，给定相同的输入，一定能够得到相同的输出，不依赖外部状态的变化
- This 指针；IO 操作；全局变量
4. 数据不可变
- 所有数据都是不可变的，如果想要修改一个对象，需要创建一个新的对象用来修改，而不是修改已有的对象。

无副作用

```
const map = (arr, fn) => arr.map(fn);

const arr = [1, 2, 3, 4, 5];
const fn = (x) => x + 1;

const result = map(arr, fn);

// result: [2, 3, 4, 5, 6]
```

一个数据经过 map 处理后，不应该修改原数据

纯函数

1. 便于测试和优化
- 实际项目开发中意义非常重大。
- 可以轻松断言函数的执行结果，可以保证函数的优化不会影响其他代码的执行，十分符合TDD（测试驱动开发）的思想，代码更加健壮
2. 可缓存性
- 没有副作用，依赖明确（函数签名），易于理解和观察，函数还便于添加 JSDoc 注释，辅助开发
3. 自文档化
- 没有副作用，依赖明确（函数签名），易于理解和观察，函数还便于添加 JSDoc 注释，辅助开发
4. 更少 BUG
- 不存在指向不明的引用，不会修改参数。
- 很多共享状态是 BUG 产生的来源

函数执行流水线的构建

柯里化

是流水线上的加工站

将一个多元函数转换成依次调用的单元函数

```
const add = (x) => (y) => x + y;

const result = add(1)(2)(3)(4)(5);
// result: 15
```

函数的返回值只有一个，那么函数的参数也就只能有一个，这样才能完成流水线的组装：让每个加工站的输出刚好流向下个加工站的输入

部分函数应用 vs 柯里化

```
const add = (x) => (y) => x + y;

const result = add(1)(2)(3)(4)(5);
// result: 15
```

部分函数应用：将部分参数固定，返回一个新的函数，这个函数可以接受剩余的参数并返回结果。

```
const add1 = add(1);
const result = add1(2)(3)(4)(5);
// result: 15
```

柯里化：将多元函数转换成依次调用的单元函数。

```
const add = (x) => (y) => x + y;
const result = add(1)(2)(3)(4)(5);
// result: 15
```

高级柯里化

```
const add = (x) => (y) => x + y;

const result = add(1)(2)(3)(4)(5);
// result: 15
```

高级柯里化：将多元函数转换成依次调用的单元函数，并且支持部分函数应用。

```
const add = (x) => (y) => x + y;

const add1 = add(1);
const result = add1(2)(3)(4)(5);
// result: 15
```

柯里化的应用

```
const add = (x) => (y) => x + y;

const result = add(1)(2)(3)(4)(5);
// result: 15
```

柯里化的应用：将多元函数转换成依次调用的单元函数，并且支持部分函数应用。

```
const add = (x) => (y) => x + y;

const add1 = add(1);
const result = add1(2)(3)(4)(5);
// result: 15
```

函数组合实例；理解

```
const add = (x) => (y) => x + y;

const result = add(1)(2)(3)(4)(5);
// result: 15
```

函数组合实例：将多元函数转换成依次调用的单元函数，并且支持部分函数应用。

```
const add = (x) => (y) => x + y;

const add1 = add(1);
const result = add1(2)(3)(4)(5);
// result: 15
```

函数组合的应用

```
const add = (x) => (y) => x + y;

const result = add(1)(2)(3)(4)(5);
// result: 15
```

函数组合的应用：将多元函数转换成依次调用的单元函数，并且支持部分函数应用。

```
const add = (x) => (y) => x + y;

const add1 = add(1);
const result = add1(2)(3)(4)(5);
// result: 15
```

注意：.add() 这样的调用是面向对象才有的，并不属于函数式

实际上是一种 pipe 的概念。和 Linux 中的命令调用思想一致

函数组合的好处

让代码更具语意，更有表现力，可读性更强

编程经验

- 柯里化中把要操作的数据放到最后
- 函数组合中函数要求单输入
- 函数组合的 Debug
- 多参考 Ramda

函数式编程缺陷

- 性能：对函数过渡包装，上下文切换过程中产生额外的性能开销
- 在 JS 这种非函数式语言中，函数式比起直接的语句指令慢
- 资源占用：不允许修改状态，需要创建新的对象，对内存、gc 会造成较大的压力
- 递归陷阱

▼	JavaScript函数式编程
	<ul style="list-style-type: none"><li>什么是函数式编程</li></ul>
▼	函数式编程的特点
▼	纯函数
	<ul style="list-style-type: none"><li>1. 便于测试和优化</li></ul>
	<ul style="list-style-type: none"><li>2. 可缓存性</li></ul>
	<ul style="list-style-type: none"><li>3. 自文档化</li></ul>
	<ul style="list-style-type: none"><li>4. 更少 BUG</li></ul>
	<ul style="list-style-type: none"><li>1. 函数是一等公民</li></ul>
	<ul style="list-style-type: none"><li>2. 惰性执行</li></ul>
	<ul style="list-style-type: none"><li>3. 无状态</li></ul>
▼	4. 数据不可变
	<ul style="list-style-type: none"><li>无副作用</li></ul>
▼	函数执行流水线的构建
▼	柯里化
	<ul style="list-style-type: none"><li>部分函数应用 vs 柯里化</li></ul>
	<ul style="list-style-type: none"><li>高级柯里化</li></ul>
	<ul style="list-style-type: none"><li>柯里化的应用</li></ul>
▼	函数组合
	<ul style="list-style-type: none"><li>函数组合实例；理解</li></ul>
▼	函数组合的应用
	<ul style="list-style-type: none"><li>注意：<code>.add()</code> 这样的调用是面向对象才有的，并不属于函数式</li></ul>
	<ul style="list-style-type: none"><li>实际上是一种 <code>pipe</code> 的概念。和 Linux 中的命令调用思想一致</li></ul>
	<ul style="list-style-type: none"><li>函数组合的好处</li></ul>
▼	编程经验
	<ul style="list-style-type: none"><li>柯里化中把要操作的数据放到最后</li></ul>
	<ul style="list-style-type: none"><li>函数组合中函数要求单输入</li></ul>
	<ul style="list-style-type: none"><li>函数组合的 Debug</li></ul>
	<ul style="list-style-type: none"><li>多参考 Ramda</li></ul>
▼	函数式编程缺陷
	<ul style="list-style-type: none"><li>性能：对函数过渡包装，上下文切换过程中产生额外的性能开销</li></ul>
	<ul style="list-style-type: none"><li>资源占用：不允许修改状态，需要创建新的对象，对内存、gc 会造成较大的压力</li></ul>
	<ul style="list-style-type: none"><li>递归陷阱</li></ul>