函数式编程

课程介绍

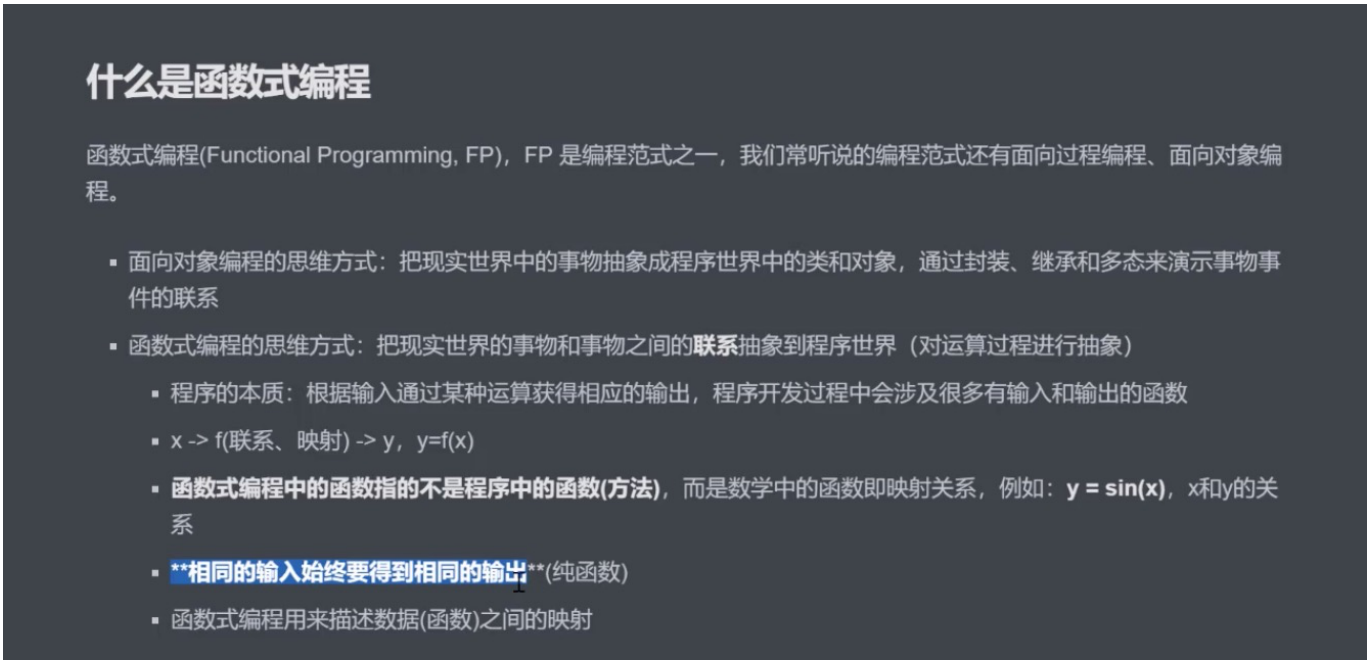
1. 为什么要学习函数编程以及什么是函数式编程
2. 函数式编程的特性（纯函数、柯里化、函数组合等）
3. 函数式编程的应用场景
4. 函数式编程库Lodash

为什么要学函数式编程

1. 函数式编程是虽则React的流行受到越来越多的关注
2. Vue 3也开始拥抱函数式编程
3. 函数式编程可以抛弃this
4. 打包过程中可以更好的利用tree shaking 过滤无用代码
5. 方便测试、方便并行处理
6. 很多库可以帮助我们进行函数式开发：lodash、underscore、ramda

函数式编程概念

将运算过程抽象成函数，进行复用



函数是一等公民

1. 函数可以存储变变量中
2. 函数作为参数
3. 函数作为返回值

在JS中，函数就是一个普通的对象(可以通过 new Function())，我们可以把函数存储到变量/数组中，它还可以作为另一个函数的参数和返回值，甚至我们可以在程序运行的时候通过 new Function(`alert(1)`)来构造一个新的函数。

高阶函数

高阶函数（Higher-order function）

1. 可以把函数作为参数传递给另一个函数
2. 可以把函数作为另一个函数的返回结果

```
function forEach (array, fn) {
    for (let i = 0; i < array.length; i++) {
        fn(array[i]);
    }
}

const arr = [1, 2, 3, 4, 5, 6];

forEach(arr, function (item) {
    console.log(item);
})
```

```
function once (fn) {
    let done = false;

    return function () {
        if (done) return;

        done = true;
        return fn.apply(this, arguments);
    }
}

const pay = once(function (money) {
    console.log(`支付了 ${money} 元`);
```

```
14 })
15
16 pay(5);
17 pay(5);
18 pay(5);
19 pay(5);
```

高阶函数的意义

1. 抽象可以帮我们屏蔽细节，只需要关注与我们的目标
2. 高阶函数式用来抽象通用的问题

练习方法：模拟Array的方法every、map、some、filter、forEach

闭包

闭包（Closure）：函数和其周围的状态（词法环境）的引用捆绑在一起形成闭包。

1. 可以在另一个作用域中调用一个函数的内部函数并访问到该函数的作用域中的成员
2. 闭包的本质：函数在执行的时候会放到一个执行栈上，当函数执行完毕之后会从执行栈上移除，但是堆上的作用域成员因为被外部引用不能释放，因此内部函数依然可以访问外部函数的成员。

好处：延长的变量的作用范围和时间

缺点：会持续占用内存，甚至导致内存泄漏

```
1 function makeFn() {
2     const msg = 'Hello World';
3   return function () {
4     console.log(msg);
5   }
6 }
```

从浏览器的运行代码的过程来看，当makePower返回的函数被执行时，闭包出现了，是power变量

```
1 function makePower (power) { // power 形成了闭包
2   return function (number) {
3     return Math.pow(number, power);
4   }
5 }
6
7 let power2 = makePower(2);
8 let power3 = makePower(3);
```

```
9
10 console.log(power2(4));
```

纯函数概念

1. 纯函数：相同的输入永远会得到相同的输出，而且没有任何可观察的副作用
    a. 纯函数就类似数学中的函数（用来描述输入和输出之间的关系），y = f(x)
2. lodash 是一个纯函数的功能库，提供了对数组、数字、对象、字符串、函数等操作的一些方法
3. 数组slice和splice分别是纯函数和不纯的函数
    a. slice 返回数组中的指定部分，不会改变原数组
    b. splice 对数组进行操作返回该数组，会改变原数组（看下面的结果，给splice传入相同参数，三次执行的结果不一样）
4. 函数式编程不会保留计算中间的结果，所以变量是不可变的（无状态的）
5. 我们可以把一个函数的执行结果交给另一个函数去处理

```
1 let array = [1, 2, 3, 4, 5, 6];
2
3 console.log(array.slice(0, 3)); // [1, 2, 3]
4 console.log(array.slice(0, 3)); // [1, 2, 3]
5 console.log(array.slice(0, 3)); // [1, 2, 3]
6 console.log("======================");
7 console.log(array.splice(0, 3)); // [1, 2, 3]
8 console.log(array.splice(0, 3)); // [4, 5, 6]
9 console.log(array.splice(0, 3)); // []
```

Lodash

是一个一致性、模块化、高性能的 JavaScript 实用工具库。https://www.lodashjs.com/

纯函数的好处

1. 可缓存。因为纯函数对相同的输入始终有相同的结果，所以可以把纯函数的结果缓存起来。
2. 可测试。纯函数让测试更方便，有输入有输出
3. 并行处理。
    a. 在多线程环境下并行操作共享的内存数据很可能会出现意外情况
    b. 纯函数不需要访问共享的内存数据，所以在并行环境下可以任意运行纯函数（Web Worker）

```
1 const _ = require('lodash');
2
3 function getArea (r) {
```

```
  console.log(r);
  return Math.PI * r * r;
}

let getAreaWithMemory =_.memoize(getArea);
console.log(getAreaWithMemory(4)); // 会打印r
console.log(getAreaWithMemory(4)); // 不会打印r

// 模拟 memoize 方法的实现
function memoize (fn) {
  let cache = {};
  return function () {
    let key = JSON.stringify(arguments);
    cache[key] = cache[key] || fn.apply(fn, arguments);
    return cache[key];
  }
}

let getAreaWithMemory2 = memoize(getArea);
console.log(getAreaWithMemory2(4)); // 会打印r
console.log(getAreaWithMemory2(4)); // 不会打印r
```

副作用

如果函数依赖于外部状态就无法保证输出相同，就会带来副作用。 所有的外部交互都有可能代理副作用，副作用也使得方法通用性下降不适合扩展和可重用性，同时副作用会给程序中带来安全隐患给程序带来不确定性，但是副作用不可能完全禁止，尽可能控制它们咋可控范围内发生。

副作用来源：

a. 外部变量
b. 配置文件
c. 数据库
d. 获取用户的输入
e. ……

柯里化(Haskell Brooks Curry)系列

定义：

1. 当一个函数有多个参数的时候先传递一部分参数调用它（这部分参数以后永远不变）
2. 然后返回一个新的函数接收剩余的参数，返回结果

总结为：

- 柯里化可以让我们给一个函数传递较少的参数得到一个已经记住了某些固定参数的新函数（使用闭包来记住）
- 这是一种对函数参数的‘缓存’
- 让函数变的更灵活，让函数的粒度更小
- 可以把多元函数转换成一元函数，可以组合使用函数产生强大的功能

使用柯里化解决硬编码的问题

```
// 柯里化
function checkAge (min) {
  return function (age) {
    return age >= min;
  }
}
// ES6写法
var checkAge = min => (age => age >= min);

let checkAge18 = checkAge(18);
let checkAge20 = checkAge(20);

console.log(checkAge18(20));
console.log(checkAge20(19));
```

Lodash中的柯里化函数

_.curry(func)

- 功能：创建一个函数，该函数接收一个或多个func的参数，如果func所需要的参数都被提供则执行func并返回执行的结果。否则继续返回该函数并等待接收剩余的参数
- 参数：需要柯里化的函数
- 返回值：柯里化后的函数

```
const _ = require('lodash')

function checkAge(min, age) {
  return age >= min;
}

const curried = _.curry(checkAge);

```

```
let checkAge18 = curried(18);
let checkAge20 = curried(20);

console.log(checkAge18(20)); // true
console.log(checkAge20(22)); // true
```

```
const _ = require('lodash');
const match = _.curry(function match (reg, str) {
  return str.match(reg);
});
const haveSpace = match(/\s+/g);
const haveNumber = match(/\d+/g);
const filter = _.curry(function (func, array) {
  return array.filter(func);
});
const findSpace = filter(haveSpace);

console.log(haveSpace('helloworld')); // null
console.log(haveNumber('123abc')); // ['123]
console.log(filter(haveSpace, ['John Connor', 'John_Donne'])); // [
 'John Connor' ]
console.log(findSpace(['John Connor', 'John_Donne'])); // [ 'John Co
nnor' ]
```

柯里化原理模拟

产生一个新函数，引用了已经传入的参数，等待剩余参数传入并执行

```

function curry (func) {
  return function curriedFn(...args) {
    // 判断实参和形参的个数
    if (args.length < func.length) {
      return function () {
        return curriedFn(...args.concat(Array.from(arguments)));
      }
    } else {
      return func(...args);
```

```
    }
  }
}

const curried = curry(function (a, b, c) {
  return a + b + c;
})

console.log(curried(1, 2, 3)); // 6
console.log(curried(1, 2)(3)); // 6
console.log(curried(1)(2, 3)); // 6
```

函数组合概念

- 纯函数和柯里化很容易写出洋葱代码，比如h(g(f(x)))
  - 获取数组的最后一个元素再转换成大写字母，`_.toUpper(_.first(_.reverse(array)))`
  - 函数组合可以让我们把细粒度的函数重新组合生成一个新的函数
  - 管道：
    - a -> fn -> b, 优化为：
    - a -> (f3 -> m -> f2 -> n -> f1) -> b

函数组合

- 函数组合(compose): 如果一个函数要经过多个函数处理才能得到最终值，这个时候可以把中间过程的函数合并成一个函数
  - 函数就像是数据管道，函数组合就是把这些管道连接起来，让数据穿过多个管道形成最终结果
  - 函数组合默认是从右到左执行

```
function compose (f, g) {
  return function (value) {
    return f(g(value));
  }
}

function reverse (array) {
  return array.reverse();
}

function first (array) {
  return array[0];
```

```
}

const last = compose(first, reverse);

console.log(last([1, 2, 3, 4])); // 4
```

Lodash中的组合函数

- _.flow(), 从左到右执行
- _.flowRight(), 从右到左执行， 这个用的更多

```
const _ = require('lodash');

const reverse = arr => arr.reverse();
const first = arr => arr[0];
const toUpper = s => s.toUpperCase();

const f = _.flowRight(toUpper, first, reverse);
const f2 = _.flow(reverse, first, toUpper);

console.log(f(['a', 'b', 'c'])); // C
console.log(f2(['aa', 'bb', 'cc'])) // CC
```

组合函数原理模拟

```
function compose (...funcs) {
  return function (value) {
    return funcs.reverse().reduce(function (acc, fn) {
      return fn(acc);
    }, value);
  }
}

const reverse = arr => arr.reverse();
const first = arr => arr[0];
const toUpper = s => s.toUpperCase();
const f = compose(toUpper, first, reverse);

```

```
14 console.log(f(['a', 'b', 'c'])); // C
```

函数组合-结合律

- 函数的组合要满足**结合律**（associativity）：
  - 我们既可以把_.toUpper和_.first组合，还可以把_.first和_.reverse组合

```
1  // 函数组合要满足结合律
2  const _ = require('lodash');
3
4  const f = _.flowRight(_.toUpper, _.first, _.reverse);
5  const f2 = _.flowRight(_.flowRight(_.toUpper, _.first), _.reverse);
6  const f3 = _.flowRight(_.toUpper, _.flowRight(_.first, _.reverse));
7
8  console.log(f(['one', 'two', 'three'])); // THREE
9  console.log(f2(['one', 'two', 'three'])); // THREE
10 console.log(f3(['one', 'two', 'three'])); // THREE
```

函数组合-调试

想以下代码中，写一个trace函数，来调试运行结果

```
1  // 函数组合的调试
2  // NERVER SAY DIE  --> nerver-say-die
3  const _ = require('lodash');
4
5  const split = _.curry((seq, str) => _.split(str, seq));
6  const join = _.curry((seq, array) => _.join(array, seq));
7  const map = _.curry((fn, array) => _.map(array ,fn));
8  const trace = _.curry((tag, v) => {
9    console.log(tag, v);
10   return v;
11 })
12 const f = _.flowRight(join('-'), trace('map'), map(_.toLower), trace
   ('split'), split(' '));
13
14 console.log(f('NERVER SAY DIE'));
```

Lodash-fp模块

内部的模块，是函数优先，参数置后。普通的函数式参数优先，函数置后。

```
// NERVER SAY DIE  --> nerver-say-die
const fp = require('lodash/fp');
const f = fp.flowRight(fp.join('-'),fp.map(fp.toLower), fp.split(' ')
);

console.log(f('NERVER SAY DIE'));
```

Lodash-map方法的小问题

这是因为_.map中，parseInt接受3个参数，第二参数书进制类别，但是实际传入的是数组下标，所以会有问题。

fp.map中的parseInt只会接受1个参数，所以默认就会使十进制

```
console.log(_.map(['23', '8', '10'], parseInt)); // [23, NaN ,2]
console.log(fp.map(parseInt, ['23', '8', '10'])); // [23, 8 ,10]
```

Pointfree

Point Free: 我们可以把数据处理的过程定义成与数据无关的合成运算，不需要用到代表数据的那个参数，只要把简单的运算步骤合成到一起，在使用这种模式之前我们需要定义一些辅助的基本运算函数。

- 不需要指明处理的数据
- 只需要合成运算过程
- 需要定义一些辅助的基本运算函数

其实就是函数组合

```
// 非Point Free模式
function f (word) {
  return word.toLowerCase().replace(/\s+/g, '_');
}

console.log(f('Hello    World')); // hello_world

// Point Free模式
```

```
const fp = require('lodash/fp');

const f2 = fp.flowRight(fp.replace(/\s+/g, '_'), fp.toLower);

console.log(f2('Hello    World')); // hello_world
```

```
const fp = require('lodash/fp');

const f = fp.flowRight(fp.join('.'), fp.map(fp.flowRight(fp.toUpper, fp.first)), fp.split(' '));

console.log(f('world wild web')); // W.W.W
```

Functor函子

为什么要学习函子

到目前为止我们已经学习了函数式编程的一些基础，但是我们还没有演示在函数式编程中如何把副作用控制在可控的范围内、异常处理、异步操作。

什么是Functor

- 容器：包含值和值的变形关系（这个变形关系就是函数）
- 函子：是一个特殊的容器，通过一个普通的对象来实现，该对象具有map方法，map方法可以运行一个函数，它对值进行处理（变形关系）

```
class Container {
  constructor(value) {
    this._value = value;
  }

  map (fn) {
    return new Container(fn(this._value));
  }
}

const c = new Container(4)
  .map(v => v + 2)
  .map(v => v * v);

```

```
15 console.log(c); // Container { _value: 36 }
```

总结为：

- 函数式编程的运算不直接操作值，而是有函子完成
- 函子就是一个实现了map契约的对象
- 我们可以把函子想象成一个盒子，这个盒子里封装了一个值
- 想要处理盒子中的值，我们需要给盒子的map方法传递一个处理值的函数（纯函数），由这个函数对值进行处理
- 最终map方法返回一个包含新值的盒子（函子）

MayBe函子

- 我们在编程的过程中可能会遇到很多错误，需要对这些错误做相应的处理
- MayBe函子的作用就是可以对外部的空值情况做处理（控制副作用在允许的范围）

在容器内，添加isNothing方法，调用map方法时，先进行空值判断

```
1  // MayBe 函子
2  class MayBe {
3    static of (value) {
4      return new MayBe(value);
5    }
6
7    constructor (value) {
8      this._value = value;
9    }
10
11   map (fn) {
12     return this.isNothing() ? MayBe.of(null) : MayBe.of(fn(this._val
   ue));
13   }
14
15   isNothing () {
16     return this._value === null || this._value === undefined;
17   }
18 }
19
20 let m = MayBe.of('Hello World')
21           .map(x => x.toUpperCase())
22           .map(x => null)
23           .map(x => x.split(' '))
```

```
24
25 console.log(m);
```

Either函子

- Either两者中的任何一个，类似于If...else...的处理
- 异常会让函数变的不纯，Either函子可以用来做异常处理

```
1  // Either 函子
2  class Left {
3    static of (value) {
4      return new Left(value);
5    }
6
7    constructor (value) {
8      this._value = value;
9    }
10
11   map (fn) {
12     return this;
13   }
14 }
15
16 class Right {
17   static of (value) {
18     return new Right(value);
19   }
20
21   constructor (value) {
22     this._value = value;
23   }
24
25   map (fn) {
26     return Right.of(fn(this._value));
27   }
28 }
29
30 function parseJSON (str) {
31   try {
```

```
    return Right.of(JSON.parse(str));
  } catch (e) {
    return Left.of({ error: e.message });
  }
}

const t1 = parseJSON('{ name: zs }');
console.log(t1); // Left { _value: { error: 'Unexpected token n in JSON at position 2' } }

const t2 = parseJSON('{ "name": "zs" }');
console.log(t2); // Right { _value: { name: 'zs' } }
```

IO函子

- IO函子中的_value 是一个函数，这里是把函数作为值来处理
- IO函子可以把不纯的动作存储到_value中，延迟执行这个不纯的操作(惰性执行)，包装当前的操作纯
- 把不纯的操作交给调用者来处理

```
const fp = require('lodash/fp');

class IO {
  static of (value) {
    return new IO(function () {
      return value;
    })
  }

  constructor (fn) {
    this._value = fn;
  }

  map (fn) {
    // 把当前的fn 和 this._value 组合
    return new IO(fp.flowRight(fn, this._value));
  }
}

const r = IO.of(process).map(p => p.execPath);
```

```
21
22 console.log(r._value());
```

Folktale

Task异步执行

- 异步任务的实现过于复杂，我们使用folktale中的Task来演示
- folktale 一个标准的函数式编程库
  - 和lodash、ramda不同的是，他没有提供很多功能函数
  - 只提供了一些函数式处理的操作，例如：compose、curry等，一些函子Task、Either、Maybe等

```
1 const { compose, curry } = require('folktale/core/lambda');
2 const { toUpper, first } = require('lodash/fp');
3
4 let f = curry(2, (x, y) => x + y);
5
6 console.log(f(1, 2));
7 console.log(f(1)(4));
8
9 let f2 = compose(toUpper, first);
10 console.log(f2(['one', 'two']));
```

Task函子

```
1 // Task 处理异步任务
2 const fs = require('fs');
3 const { task } = require('folktale/concurrency/task');
4 const { split, find } = require('lodash/fp');
5
6 function readFile (filename) {
7   return task(resolver => {
8     fs.readFile(filename, 'utf-8', (err, data) => {
9       if (err) resolver.reject(err);
10
11       resolver.resolve(data);
12     })
13   })
14 }
```

```
15
16 readFile('../../package.json')
17   .map(split('\n'))
18   .map(find(x => x.includes('version')))
19   .run()
20   .listen({
21     onRejected: err => {
22       console.log(err);
23     },
24     onResolved: value => {
25       console.log(value);
26     }
27   })
```

Pointed函子

IO函子问题

由于IO封装了function，我们需要手动去调用_value函数，会导致_value的执行过多，原因是IO的嵌套过多

Monad函子(单子)

- Monad 函子是可以变扁的Pointed函子，IO(IO(x))
- 一个函子如果具有join和of两个方法并遵守一些定律就是一个Monad

```
const fp = require('lodash/fp');
const fs = require('fs');

class IO {
  static of (value) {
    return new IO(function () {
      return value;
    });
  }

  constructor (fn) {
    this._value = fn;
  }

  map (fn) {
    return new IO(fp.flowRight(fn, this._value));
  }

  join () {
    return this._value();
  }

  flatMap (fn) {
    return this.map(fn).join();
  }
}

let readFile = function (filename) {
  return new IO(function () {
    return fs.readFileSync(filename, 'utf-8');
  })
}

let print = function (x) {
  return new IO(function () {
    console.log(x);
```

```
37      return x;
38    })
39  }
40
41  let r = readFile('../../package.json')
42            .map(fp.toUpper)
43            .flatMap(print)
44            .join();
```

总结

学习顺序：

认识函数式编程 -> 函数相关复习 -> 函数式编程基础 -> 函子

随堂测验

01. 下面关于函数式编程的说法错误的是： [分值：5]

您的回答：C. 函数式编程可以很大程度上提高程序的性能 ┇ D. 函数式编程中的函数是的是程序中的函数或者方法 ✓ （得分：5）

答案解析：

解析：

C 选项函数式编程不能提高程序的性能，因为大量使用闭包在某种程度上会降低性能(占用内存)

D 函数式编程中的函数不是程序中的函数或者方法，而是数学中的函数

---

02. 函数是一等公民包含： [分值：5]

您的回答：A. 函数可以存储在变量中 ┇ B. 函数可以作为参数 ┇ C. 函数可以作为返回值 ✓ （得分：5）

答案解析：

MDN 中关于头等函数(函数是一等公民)的解释只包含前三点

---

03. 下面关于纯函数的说法错误的是： [分值：5]

您的回答：D. 可以通过某种方式避免副作用的发生 ✓ （得分：5）

答案解析：

副作用会让一个函数变的不纯，副作用不可能避免，因为代码难免会依赖外部的配置文件、数据库等，只能最大程度上控制副作用在可控的范围内发生

---

04. 关于柯里化的描述正确的是： [分值：5]

您的回答：A. 柯里化函数 curry，也是高阶函数 ┇ B. 柯里化函数 curry 内部用到了闭包，对函数的参数做了“缓存” ┇ C. 柯里化可以把多个参数的函数转换成只有一个参数的函数，通过组合产生功能更强大的函数 ┇ D. 柯里化让函数变的更灵活，让函数的粒度更小 ✓ （得分：5）

---

05. 关于函数组合说法正确的是： [分值：5]

您的回答：A. 函数可以看做一个处理数据的管道，管道中输入参数 x，在管道中对数据处理后得到结果 y ┇ B. 通过函数组合可以把多个一元函数组合成一个功能更强大的函数 ┇ C. 函数组合需要满足结合律 ┇ D. 函数组合默认的执行顺序是从右到左 ✓ （得分：5）

---

06. 关于函子的说法正确的是： [分值：5]

您的回答：A. 函子是一个特殊的容器(对象)，这个容器内部封装一个值，通过 map 传递一个函数对值进行处理 ┇ B. MayBe 函子的作用是处理外部的空值情况，防止空值的异常 ┇ C. IO 函子内部封装的值是一个函数，把不纯的操作封装到这个函数，不纯的操作交给调用者处理 ┇ D.Monad 函子内部封装的值是一个函数(这个函数返回函子)，目的是通过 join 方法避免函子嵌套 ✓ （得分：5）
