JS函数式编程与Ramda.js实践

— 邵乾飞

什么是函数式编程?


啥啥啥 写的这是啥

什么是函数式编程

函数式编程 [编辑]

维基百科，自由的百科全书

函数式编程（英语：functional programming）或称**函数程序设计**、**泛函编程**，是一种编程范式，它将计算机运算视为函数运算，并且避免使用程序状态以及易变对象。其中，λ演算（lambda calculus）为该语言最重要的基础。而且，λ演算的函数可以接受函数当作输入（引数）和输出（传出值）。

比起指令式编程，函数式编程更加强调程序执行的结果而非执行的过程，倡导利用若干简单的执行单元让计算结果不断渐进，逐层推导复杂的运算，而不是设计一个复杂的执行过程。

什么是函数式编程

一个小例子：

```
// 我想获取一组用户的名字
var users = [
  { name: "Bob", age: 20 },
  { name: "Tom", age: 25 },
  { name: "Mike", age: 30 }
];

// 命令式：
var names = [];
for (var i = 0; i < users.length;i++) {
  names.push(users[i].name);
}

// 声明式：
var names = users.map(item => item.name);

// 声明式：
var prop = propName => data => data[propName]
var names = users.map(prop("name"));

// 声明式：
var names = users.map(R.prop("name"));

// 声明式：
var names = R.pluck("name")(users);
```

很多时候，我们不需要告诉计算机如何一步一步地去做（实现），

我们只关心想要得到什么。

Ramda.js简介

Ramda.js简介

特点:

- 数据不变性,函数无副作用
- 函数本身均自动柯里化
- data last(要操作的数据通常在最后面)

相关链接:

- Ramda.js官网
- Ramda.js常用方法

Ramda.js简介-对比Lodash

```
const getUserName  = item => item.name

/**我想定义一个方法：接收users数组，返回users的name组成的数组**/

// 方法一：利用lodash的map
const getAllNames = users => {
  return _.map(users, getUserName);
}

// 方法二：利用Ramda的map
const getAllNames = users => {
  return R.map(getUserName, users);
}
// ===> 可演变为：
const getAllNames = R.map(getUserName)
```

```
getAllNames([{name: 'Susan', age: 20}, {name: 'Tom', age: 21}]) // ['Susan', 'Tom']
```

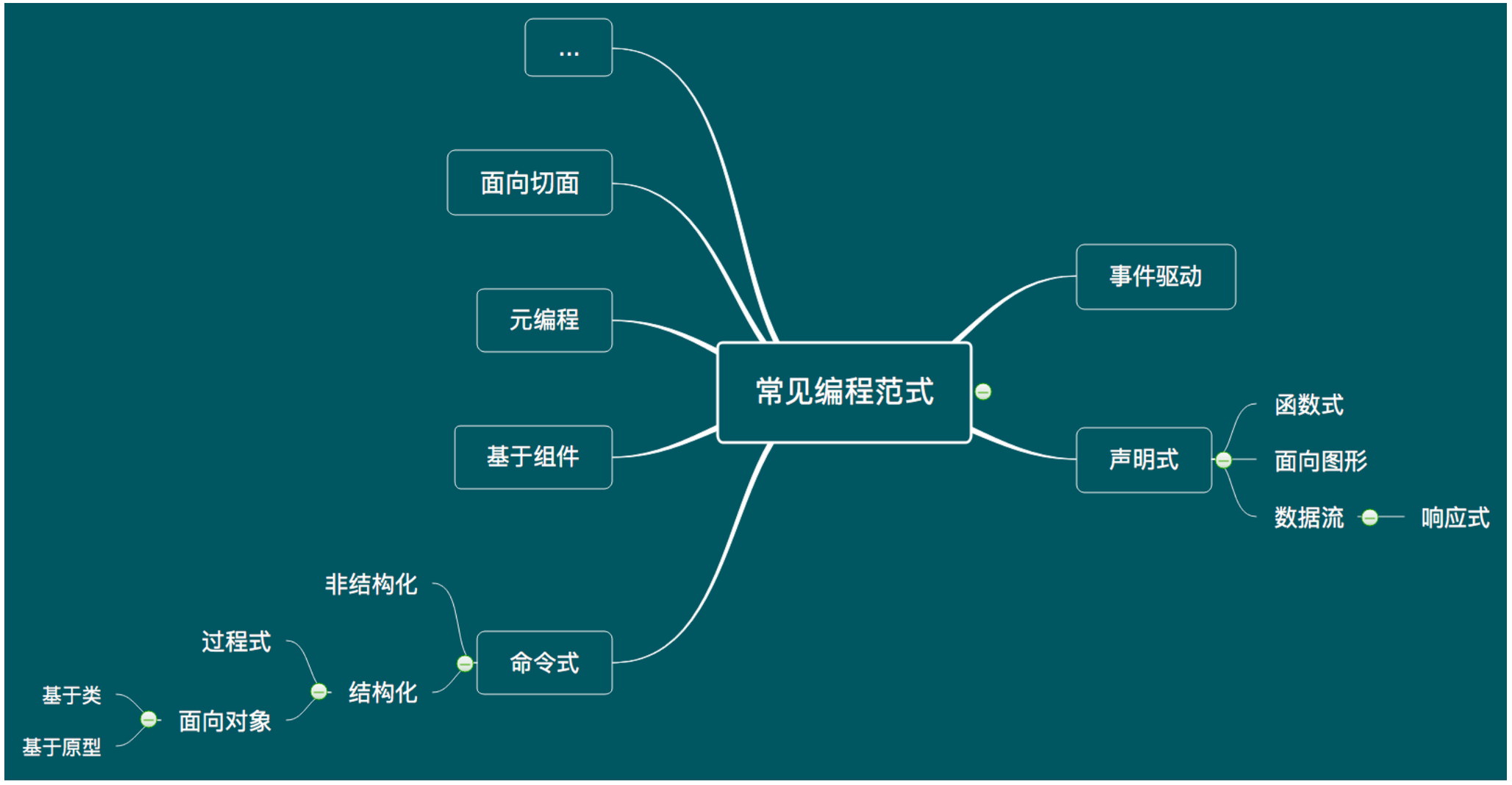
JS函数式编程的特点

JS函数式编程的特点

特点:

- 声明式编程
- 函数是一等公民
- 纯函数
- 柯里化
- 推崇函数组合(compose)
- Pointfree

常见的编程范式


常见编程范式
...
面向切面
元编程
基于组件
事件驱动
声明式
函数式
面向图形
数据流
响应式
命令式
非结构化
结构化
过程式
面向对象
基于类
基于原型

声明式编程

SQL语句

```sql
SELECT name, population FROM world WHERE name IN ('India', 'China');
```

JS

```js
// 获取一组用户的年龄
const users = [
  { name: "Bob", age: 20 },
  { name: "Mike", age: 30 }
];
const ages = R.pluck("age")(users);
```

- 屏蔽实现的细节
- 让计算机明白目标，而非流程

函数是一等公民

特点:

- 函数可以赋值给其他变量
- 函数可以作为函数的参数
- 函数可以作为函数返回值

函数是一等公民-作为变量

```
const sayHello = userName => `hi ${userName}`
// 函数后面跟括号
const greeting = sayHello('Bob') // "hi Bob"
```

```
// 函数后面无括号
const print = console.log // ƒ log() { [native code] }
print('hello world')
```

函数是一等公民-作为参数

```
const users = [
  { name: "Bob", age: 20 },
  { name: "Tom", age: 25 },
  { name: "Mike", age: 30 }
];
const printUserName = user => console.log(user.name)

users.forEach(printUserName);
```

函数是一等公民-作为返回值

```
// prop方法运行之后会返回一个函数
var prop = propName => data => data[propName]

var names = users.map(prop("name"));
var ages = users.map(prop("age"));
```

纯函数

特点:

- 相同的输入永远有相同的输出(引用透明)
- 没有副作用(修改UI、打印日志、修改外部变量...)
- 函数是什么

纯函数

```
let MaxPassengerCount = 20
// 非纯函数，  依赖了外部状态
const isOverload = passengerCount => {
  return passengerCount > MaxPassengerCount
}

// 纯函数，不依赖外部状态
const isOverload = passengerCount => {
  const MaxPassengerCount = 20
  return passengerCount > MaxPassengerCount
}
```

柯里化

```
f: (X,Y) => Z
curry(f): X => Y => Z
```

特点:

- 接受多个参数的函数变换成接受一个单一参数的函数，并且返回接受余下的参数而且返回结果的新函数
- 为纪念 Haskell Brooks Curry
- JS中通过闭包将参数隐藏在柯里化函数的环境中

柯里化-实践

```
// 我们希望为很多金额添加货币前缀
const currency = (prefix,amount) => `${prefix}${amount}`

// 如果没有柯里化：
const phoneSpent = currency('$', '530.50') // "$530.50"
const bookSpent = currency('$', '20.98') // "$20.98"

// 柯里化：
const curriedCurrency = R.curry(currency)
// 生成一个专用于处理美元符号的函数
const dollarCurrency = curriedCurrency('$')
const phoneSpent = dollarCurrency('530.50') // "$530.50"
const bookSpent = dollarCurrency('20.98') // "$20.98"
```

柯里化-实践

```
handleFieldsChange = (key, e) => {
  const { value } = e.target;
  this.setState({ [key]: value });
};

render() {
  const { name, company } = this.state;
  return (
    <div className={styles.searchForm}>
      <Input
        value={name}
        onChange={e => {
          this.handleFieldsChange("name", e);
        }}
      />
      <Input
        value={company}
        onChange={e => {
          this.handleFieldsChange("company", e);
        }}
      />
```

柯里化-实践

```
handleFieldsChange = key => e => {
  const { value } = e.target;
  this.setState({ [key]: value });
};

render() {
  const { name, company } = this.state;
  return (
    <div className={styles.searchForm}>
      <Input value={name} onChange={this.handleFieldsChange("name")} />
      <Input value={company} onChange={this.handleFieldsChange("company")} />
```

柯里化-实践

```
// curry 方法，返回一个函数
export const curry = fn =>
  (innerFunc = (...args) =>
    args.length >= fn.length
      ? //如果参数个数达到原来的函数个数，则直接执行返回
        fn(...args)
      : // 否则返回一个函数：此函数固定了前面的参数
        (...backArgs) => innerFunc(...args, ...backArgs));
```

函数组合(compose)

```
g(h(x))
===> compose(g,h)(x)
```

- JS管道操作符
- KISS原则(**K**eep **I**t **S**imple, **S**tupid)

函数组合(compose)

```
const fruits = ["APPLE", "BANANA", "PEACH"];

// 我希望取得数组中第一个水果 并且转为小写
const head = function(data) {
  return data[0];
};
const toLowerCase = function(data) {
  return data.toLowerCase();
};

const getFirstLowerCaseFruit = function(data) {
  return toLowerCase(head(data));
};
getFirstLowerCaseFruit(fruits); // 'apple'
// ==> 转变为:
const getFirstLowerCaseFruit = R.compose(toLowerCase,head);
getFirstLowerCaseFruit(fruits); // 'apple'

// 得出公式:   g(h(x)) === compose(g,h)(x)
```

函数组合(compose)-React HOC

```
1   class MyComponent extends React.Component {
2     // ...
3   }
4
5   const mapStateToProps = state => ({
6     contract: state.contracts.contract
7   });
8
9   export default auth(connect(mapStateToProps)(withRouter(MyComponent)));
10
```

函数组合(compose)-React HOC

```
class MyComponent extends React.Component {
  // ...
}

const mapStateToProps = state => ({
  contract: state.contracts.contract
});

export default R.compose(
  auth,
  connect(mapStateToProps),
  withRouter
)(MyComponent);
```

函数组合(compose)-React HOC

```
1   const mapStateToProps = state => ({
2     contract: state.contracts.contract
3   });
4
5   @auth
6   @connect(mapStateToProps)
7   @withRouter
8   class MyComponent extends React.Component {
9     // ...
10  }
11
12  export default MyComponent;
13
```

Pointfree

特点:

- 无参数风格编程
- 上层运算不要直接操作数据,只合成运算过程

Pointfree

```
// 判断一个学生是否已经毕业并且性别为男性
const isMale = student => student.gender === 'male'
const wasGraduated = student => student.graduated

const isMaleAndGraduated = student =>
  R.both(isMale, wasGraduated)(student)

// value => f(value) 等价于 f
// Pointfree Style
const isMaleAndGraduated = R.both(isMale, wasGraduated)
```

Pointfree

```
// isMale方法的演变
const isMale = student => student.gender === 'male'

// ==>
const isMale = student => R.equals('male', student.gender)
// ==>
const isMale = student => R.equals('male', R.prop('gender')(student))
// ==>
const isMale = R.pipe(R.prop('gender'), R.equals('male'))
// ==> 或者使用更简洁的方法
const isMale = R.propEq('gender', 'male')
```

Pointfree

```
// wasGraduated方法的演变
const wasGraduated = student => student.graduated
// ==>
const wasGraduated = R.prop('graduated')
```

Pointfree-代码的坏味道

```
// 错误示范 多了一层包裹函数，而且不符合pointfree风格
fetchList().then().catch(e => dealError(e))
// 正确示范
fetchList().then().catch(dealError)
```

```
// 错误示范
const userNames = users.map(user => getUserName(user))
// 正确示范
const userNames = users.map(getUserName)
```

改善现有代码

改善现有代码

```
// 错误示范
let isValid
const VALID_CODE = 200
// 直接获取code属性，不够健壮
const code = res.code
// 多余的逻辑判断
if (code === VALID_CODE) {
  isValid = true
} else {
  isValid = false
}

// 正确示范
const isValid = R.equals(VALID_CODE, R.path(['code'], res))
// ===>
const isValid = R.compose(R.equals(VALID_CODE), R.path(['code']))(res)
```

改善现有代码

```
// http://git.highso.com.cn:81/main_station/fe-antd-mainsite-order/merge_requests/1
// 违背DRY原则
saveQueryBankTypelist(state, { payload }) {
  return {
    ...state,
    querybanktypelist: payload,
  };
},
saveQuerySubbranchlist(state, { payload }) {
  return {
    ...state,
    querysubbranchlist: payload,
  };
// 改为:
saveState(state, { payload }) {
  const {data, prop} = payload
  return {
    ...state,
    [prop]: data
  };
}
```

改善现有代码

```
// http://git.highso.com.cn:81/fe/fe-h5-order/merge_requests/67
// 显示密码
browsePwd = () => {
  const { showPwd } = this.state;
  this.setState({
    showPwd: !showPwd
  })
}

// 改为更具通用性的代码：
togglePwd = () => this.setState(prevState => ({showPwd: !prevState.showPwd}))
```

改善现有代码

```
// 判断是否微信环境
export function isWeiXin () {
  var ua = window.navigator.userAgent.toLowerCase();
  var Weixin = ua.indexOf('micromessenger') !== -1;
  if (Weixin) {
    localStorage.setItem('goWindows', false);
    return true;
  } else {
    localStorage.removeItem('goWindows');
    $('.shareBtn').hide()
    return false;
  }
}
```

```
// 改为:
export const isWeixin => /micromessenger/i.test(window.navigator.userAgent)
```

单元测试和debug

单元测试

```
Debug
// NOTE: 纯函数接收参数，返回固定值
test('test toNumber', () => {
  expect(toNumber({})).toBe(0)
  expect(toNumber(2)).toBe(2)
  expect(toNumber('2...3re3')).toBe(2.33)
  expect(toNumber('2.33')).toBe(2.33)
  expect(toNumber('-2.33')).toBe(-2.33)
})
Debug
test('test milliFormat', () => {
  expect(milliFormat(1234)).toBe('1,234.00')
  expect(milliFormat('1234')).toBe('1,234.00')
  expect(milliFormat(-1234)).toBe('-1,234.00')
  expect(milliFormat('-1234')).toBe('-1,234.00')
  expect(milliFormat('-123-4')).toBe('0.00')
})
Debug
test('test money', () => {
  expect(money(1234)).toBe('¥1,234.00')
  expect(money(-1234)).toBe('¥-1,234.00')
  expect(money('-1234')).toBe('¥-1,234.00')
  expect(money(1234, 2, false)).toBe('1,234.00')
  expect(money(-1234, 2, false)).toBe('-1,234.00')
  expect(money('-1234', 2, false)).toBe('-1,234.00')
  expect(money('-1234', {})).toBe('¥-1,234.00')
  expect(money('-1234', null)).toBe('¥-1,234.00')
})
```

debug

```
const calculate = R.compose(R.negate,R.inc,R.add)

calculate(2,3) // -6

calculate(2) // NaN  我知道出了一点问题，但是看起来难以下手


// 调试方法
const debug = R.curry(function(tag, value){
  console.log(tag, value);
  return value;
});

const calculate = R.compose(R.negate,R.inc,debug('value here is:'),R.add)
calculate(2) // value here is: ƒ f1(a) {...}
calculate(2,3) // value here is: 5
```

为什么要用函数式编程

为什么要用函数式编程

优点:

- 无副作用，会减少大量的错误来源
- 代码量更少，开发迅速
- Pointfree风格编程无需定义多余变量
- 拆分与组合使函数的功能单一、代码更健壮、可重用,更易于模块化
- 声明式编程，易读、易于理解
- 单元测试和debug更方便
- …

Q&A