FUNCTIONAL PROGRAMMING IN REACT

工业聚@Ctrip

A WAY TO IMPROVE REACT SKILL

- Step 1: Learn some JavaScript
- Step 2: Learn some React
- Step 3: Learn some Haskell
- Step 4: Learn some Category Theory
- Step 5: Go to step 1

YOUR IMPRESSION ABOUT FP

- Programming tricks
- Programming design patterns
- Programming paradigm
- Computer Science
- Mathematics
- Philosophy
- Others

FP VS NON-FP

- Non-FP: Discover mathematics from programming
- FP: Apply mathematics to programming
- The best practices in Non-FP follow the principle of FP
- Such like avoid global state, Defensive Programming, comosable and so on
- The FP parts of a codebase are always safest parts.

CALCULATION AND EFFECT

The code is all about Calculation and Effect

- Calculation is an Effect of Code
- Code with not only Calculation Effect means Side Effects
- Code with only Calculation Effect means Purity

PURE FUNCTION

- Works like Math formula
- Always return the same result when got the same arguments
- No other effects

```
const add = (x, y) => x + y
// immutable variable
const person = { firstname: 'Jade', lastname: 'Gu' }
// 'person' is just works like alias
const getFullName = person => `${person.firstname} ${person.lastna
```

IMPURE FUNCTION

Programming in an unsafe way

- Your code change the Environment irreversibly
- The Environment your code depends on become unexpected

```
// depends on the memory of the computer
let count = 0
// different result with the same arguments
let getCount = () => Count++
let setCount = n => count = n // n may not be number
// cause side effects in console
let printCount = () => console.log(id)
// cause side effects in DOM
let renderCount = () => document.getElementById('count').innerHTML
```

WHY PURITY MATTERS?

Remove a non-existent element will make your app crash

- Side Effects are dangerous
- Side Effects are hard to reason about
- Side Effects don't scale
- Side Effects make thing became more complex than it should be

THE HIGHER-ORDER FUNCTION

Our best friends in Front-end Development

- A function returns another function
- A function takes another function as the argument

```
// map is higher-order function
const map = (list, f) => list.map(f)
// filter is also higher-order function
const filter = f => list => list.filter(f)
// ajax is higher-order function
ajax(url, data, successCallback, errorCallback)
// fs.readFile is higher-order function
fs.readFile('/filename.txt', (error, data) => {})
```

A LITTLE JAVASCRIPT PROBLEM

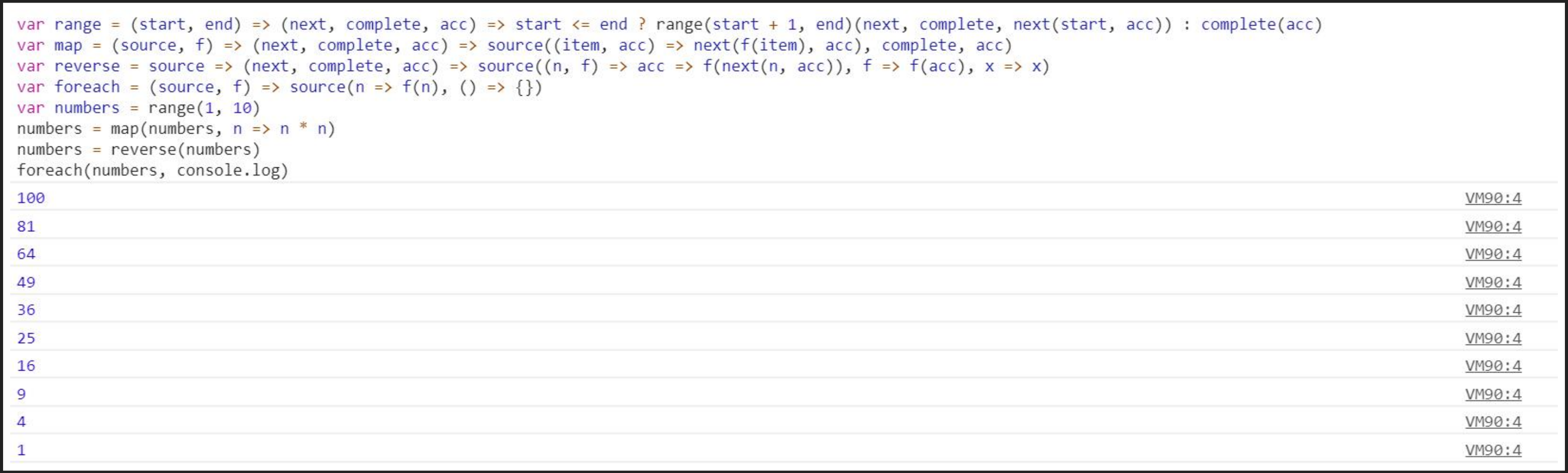
- No arrays, No object, no dot operator, no for-loop, no while-loop
- How to solve the problem without the data structure?

```
var numbers = range(1, 10);
numbers = map(numbers, function (n) { return n * n });
numbers = reverse(numbers);
foreach(numbers, console.log);
/* output:
  100
  81
  64
  49
  36
  25
  16
  9
  4
  1
*/
```

THE POWER OF FUNCTION

Lambda calculus

- We can just use Function do all possiable calculus in computer
- All programming language features can be desugered into Function



```
var range = (start, end) => (next, complete, acc) => start <= end ? range(start + 1, end)(next, complete, next(start, acc)) : complete(acc)
var map = (source, f) => (next, complete, acc) => source((item, acc) => next(f(item), acc), complete, acc)
var reverse = source => (next, complete, acc) => source((n, f) => acc => f(next(n, acc)), f => f(acc), x => x)
var foreach = (source, f) => source(n => f(n), () => {})
var numbers = range(1, 10)
numbers = map(numbers, n => n * n)
numbers = reverse(numbers)
foreach(numbers, console.log)
100                                                                   VM90:4
81                                                                    VM90:4
64                                                                    VM90:4
49                                                                    VM90:4
36                                                                    VM90:4
25                                                                    VM90:4
16                                                                    VM90:4
9                                                                     VM90:4
4                                                                     VM90:4
1                                                                     VM90:4
```

WHAT IS THE CONSISTENCY OF THE FOLLOWING?

- String
- Array
- Object
- Tree
- Promise

MAPABLE AND FUNCTOR

- map(string, char => char)
- map(array, item => item)
- map(object, value => value)
- map(tree, node => node)
- map(promise, asyncValue => asyncValue)

```
fmap id = id
fmap (g . f) = fmap g . fmap f
```

WHAT IS THE CONSISTENCY OF THE FOLLOWING?

- Number
- String
- Array
- Function

ADDITION OF NUMBERS

- 0 is identity value
- + is associative operation

```
0 + n = n
n + 0 = n
(0 + 1) + 2 = 0 + (1 + 2)
```

MULTIPLICATION OF NUMBERS

- **1** is identity value
- ***** is associative operation

```
1 * n = n
n * 1 = n
(1 * 2) * 3 = 1 * (2 * 3)
```

CONCATENATION OF ARRAY

- [] is identity value
- ++ is associative operation

```
[] ++ [1,2,3] = [1,2,3]
[1,2,3] ++ [] = [1,2,3]
[1,2] ++ [3] ++ [4] = [1,2] ++ ([3] ++
```

CONCATENATION OF STRING

- "" is identity value
- + is associative operation

```
"abc" + "" = "abc"
"" + "abc" = "abc"
"abc" + "xyz" + "123" = "abc" + ("xyz" + "1
```

COMPOSITION OF FUNCTIONS

- id is identity value
- compose is associative operation

```
id = \x -> x
f . id = f
id . f = f
(f . g) . h = f . (g . h)
```

MONOID

a monoid is an algebraic structure with a single associative binary operation and an identity element.

- All of the identity value named **mempty**
- All of the associative operation named **mappend**

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
```

SOURCE OF POWER OF REACT

Why React works better than jQuery in complex web app?

- jQuery is lack in side effects management
- The methods in $.fn mostly perform Side Effects
- React splits UI Dev into two parts: JSX && Renderer
- JSX can be Pure Function without Side Effects and are easy to compose.
- Renderer(ReactDOM/ReactNative) performs Side Effects later

FUNCTIONAL SETSTATE

The common question: Why is setState async?

- Some people want setState performs side-effects immediately
- setState is optimized by performing batches updates if possible
- setState(updater[, callback]) is higher-order function

```
// imperative style
this.setState(newState)
this.setState(anotherState)
console.log(this.state)
// functional style
this.setState(state => update(state), () => console.log(this.state
this.setState(state => update(state), () => console.log(this.state
```

HIGHER-ORDER COMPONENT

HOC is the subset of the higher-order function

- a higher-order component is a function that takes a component and returns a new component.

```
const defaultProps = defaults => InputComponent => {
  const OutputComponent = props => <InputComponent {...defaults} {...pr
  return OutputComponent
}
const ComponentWithDefaultProps = defaultProps({ name: 'Jade' })(MyComp
```

RENDER PROPS

Render props is the subset of the higher-order function

- A component with a render prop takes a function that returns a React element and calls it instead of implementing its own render logic

```
<Mouse>
  {mouse => (
    <p>The mouse position is {mouse.x}, {mouse.y}</p>
  )}
</Mouse>
```

REACT SUSPENSE

Based on the idea of FP: algebraic-effects

- fetch data without async/await

```
const commentInfoFetcher = createFetcher(
fetchCommentInfo // function that returns a fetch() pror
)
function CommentsInfo({ commentId }) {
const commentInfo = commentInfoFetcher.read(commentId)
return <div>
  Title: {commentInfo.title}
  Description: {commentInfo.description}
</div>
}
```

ALGEBRAIC-EFFECTS

Rise and Resume

- never resume -> Exception effect
- resume once -> Promise/Generator/State
- resume multiple times

```
with(handler) {
  const forename = prompt('What is your forena
  const surname = prompt ('What is your surnam
  return([forename, surname].join(' '))
}
```

ALGEBRAIC-EFFECTS IN JAVASCRIPT

Sebastian proposed two years ago

```
function otherFunction() {
    console.log(1);
    let a = perform { x: 1, y: 2 };
    console.log(a);
    return a;
  }

  do try {
    let b = otherFunction();
    b + 1;
  } catch effect -> [{ x, y }, continuation] {
    console.log(2);
    let c = continuation(x + y);
    console.log(c);
    c + 1;
  }
```

ALGEBRAIC-EFFECTS IN JAVASCRIPT

(throw + catch ≈ rise) && (re-run ≈ resume)

- throw promise -> catch promise -> promise.then(reRun)
- throw observable -> catch observable -> observable.subscribe(reRun)

```
const main = operators => {
  let { fetchJSON, interval } = operators;
  let data = fetchJSON("https://api.github.com/repos/zeit/next.js"
  let count = interval(1000 / 60);
  let angle = count % 360;
  let root = document.getElementById("root");
  root.style.transform = `rotate(${angle}deg)`;
  root.innerHTML = data.stargazers_count;
};

withHandler({ interval, fetchJSON })(main);
```

TAIL CALL OPTIMIZATION FOR JS

A special case of algebraic-effects

- Optimizing tail-resumptions to a regular function call
- Tail call optimization demo

```
const tco = f => {
  let isCalling = false
  return (...args) => {
      if (isCalling) throw args
      while (true) {
          try {
              isCalling = true
              return f(...args)
          } catch(params) {
              if (params instanceof Error) throw params
              args = params
          } finally {
              isCalling = false
          }
      }
  }
```

UNSAFE RESUMPTION

why did React deprecated componentWill*?

- Many React Dev perform side-effects in componentWill*
- Re-render component with the same props and state should be safe
- Solution: performs side-effects in componentDid*

REACTIVE REACT

Put react in rxjs

- toReactComponent :: observable -> react-component

```
const App = from("hello rxjs-react!").pipe(
  concatMap(char => of(char).pipe(delay(300))),
  scan((str, char) => str + char, ""),
  toReactComponent(text => {
      return (
      <div>
          <h1>{text}</h1>
      </div>
      );
  })
);
```

REACTIVE REACT

Put rxjs in react

- reactive :: component -> reactive-component

```
const App = reactive(() => {
  const hello$ = from('hello rxjs-react!').pipe(
      concatMap(char => of(char).pipe(delay(300))),
      scan((str, char) => str + char, ''),
      map(text => <h1>{text}</h1>)
  )
  return <div>{hello$}</div>
})
```

CONCLUSION

- FP is awesome in web dev when using it in a right way(Such like React)
- JS is not friendly enough for FP (e.g algebraic-effects)
- Learning FP can improve our understanding of programming

Q & A

关注我的公众号，了解更多前端玩法