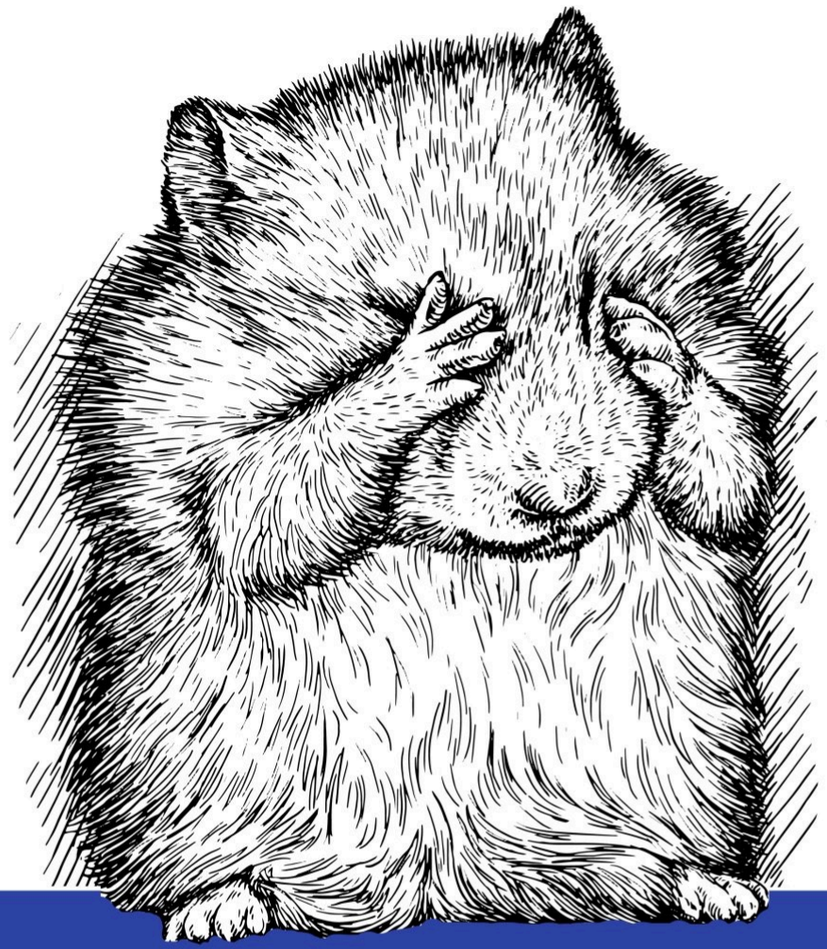
Функциональный подход в программировании

Сергей Колесников

Solutions that might fix the problem without breaking anything



Essential

Hoping This Works

O RLY?

@ThePracticalDev

Зачем применять ФП?

- Упростить и структурировать код
- Улучшить тестируемость

Итог: лучшая поддерживаемость кодовой базы

Что делает функция? Есть ли здесь баг?

```
function doSomething(items) {
    const len = items.length
    for (let i = 0; i < len ; i++) {
        for(let j = 0 ; j < len - i - 1; j++) {
            if (items[j] > items[j + 1]) {
                let temp = items[j]
                items[j] = items[j+1]
                items[j + 1] = temp
            }
        }
    }

    for (let i = 0; i < len ; i++) {
        items[i] = `<div>${items[i]}</div>`
    }

    return items
}
```

Чистая функция

Удовлетворяет двум условиям:

1. является детерминированной;
2. не обладает побочными эффектами (идемпотентна).

Детерминированность

Функция является **детерминированной**, если для одного и того же набора входных значений она возвращает одинаковый результат.

```
const deterministic = mul ⇒ 2 * mul

const nonDeterministic = mul ⇒
    Math.ceil(Math.random() * 100) * mul
```

Идемпотентность — отсутствие побочных эффектов

Выполнение функции не изменяет какое-либо состояние за пределами её области видимости и не оказывает видимого воздействия на внешний мир, кроме возвращения значения (никаких побочных эффектов).

Пример **детерминированной** *(возвращаемое значение зависит только от входного значения)*, но **неидемпотентной** *(меняет внешнюю переменную)* функции:

```
var isFunctionCalled = false

function foo(bar) {
    isFunctionCalled = true
    return bar + '1'
}
```

Композиция

```
const multiply = (a, b) ⇒ a * b
const addOne = x ⇒ x + 1
const square = x ⇒ x * x

const operate = (x, y) ⇒ {
    const product = multiply(x, y)
    const incremented = addOne(product)
    const squared = square(incremented)

    return squared
}

operate(3, 4) // ⇒ ((3 * 4) + 1)^2 ⇒ (12 + 1)^2 ⇒ 13^2 ⇒
169
```

Композиция

```
const multiply = (a, b) ⇒ a * b
const addOne = x ⇒ x + 1
const square = x ⇒ x * x

const operate = (x, y) ⇒ square(addOne(multiply(x, y)))

operate(3, 4)
```

Композиция функций — применение одной функции к результату другой

```
const multiply = (a, b) ⇒ a * b
const addOne = x ⇒ x + 1
const square = x ⇒ x * x

const operate = compose(
    square,
    addOne,
    multiply
)

operate(3, 4)
// даст тот же результат, что и
square(addOne(multiply(3, 4)))

// compose(f, g)(value) эквивалентно f(g(value))
```

Каррирование

- **Функции высших порядков** — это такие функции, которые могут принимать в качестве аргументов и возвращать другие функции.
- **Каррирование** — преобразование функции от многих аргументов в набор функций, каждая из которых является функцией от одного аргумента.

```
const bar = foo(a)

foo(a, b) == foo(a)(b)
bar(b) == foo(a, b)
```

Пример каррированой функции

```
// Фильтрация книг по году
const publishedInYear = curry((year, book) ⇒ book.year === year)

// Показать названия книг, опубликованных в определённый год
const titlesForYear = (books, year) ⇒
  compose(
    map(book ⇒ book.title)
    filter( publishedInYear(year) ),
  )(books)
```

Иммутабельность

Неизменяемым (англ. immutable) называется объект, состояние которого не может быть изменено после создания. Результатом любой модификации такого объекта всегда будет новый объект, при этом старый объект не изменится.

Функция в процедурном стиле

```
function doSomething(items) {
    const len = items.length
    for (let i = 0; i < len ; i++) {
        for(let j = 0 ; j < len - i - 1; j++) {
            if (items[j] > items[j + 1]) {
                let temp = items[j]
                items[j] = items[j+1]
                items[j + 1] = temp
            }
        }
    }

    for (let i = 0; i < len ; i++) {
        items[i] = `<div>${items[i]}</div>`
    }

    return items
}
```

Эта же функция в функциональном стиле

```
const array = [ 2, 1, 3, 5, 4, 0 ]

function renderAsSortedDivList(items) {
    return [  ... items ] // создаём копию массива
        .sort( (a, b) ⇒ a - b ) // сортируем его
        .map(
            item ⇒ '<div>' + item + ‘</div>’
        ) // изменяем элементы массива
}

const renderedArray = renderAsSortedDivList(array)
console.table({
    "Original": array,
    "Rendered": renderedArray
})
```

Функция, написанная в функциональном стиле с помощью библиотеки Ramda

```
import { compose, map, sort } from 'ramda'

const toDiv = item ⇒ `<div>${item}</div>`
const sortAsc = sort((a, b) ⇒ a - b)

const renderAsSortedDivList = compose(map(toDiv), sortAsc)

const renderedArray = renderAsSortedDivList(array)
console.table({
    "Original": array,
    "Rendered": renderedArray
})
```

Что дальше?

- Мышление в стиле Ramda: https://habr.com/ru/post/348868/
- Документация к Ramda: https://ramdajs.com/docs/
- Структура и интерпретация компьютерных программ *Гарольд Абельсон и Джералд Джей Сассман*