ПрАТ «ПВНЗ «ЗАПОРІЗЬКИЙ ІНСТИТУТ ЕКОНОМІКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ»

Використання функціонального програмування у JavaScript та фреймворках

Доповідач
магістрант групи ІПЗ 211-м Іжиков А.І.

Науковий керівник
доцент Жеребцов О.А.

Мета та актуальність теми

У наш час розробка веб-застосунків за допомогою фреймворків є актуальною задачею та має великий попит. Підходи проектування, послуги, патерни програмування, методології моделювання – ніхто та ніщо не стоїть на місці. Тому важливо вміти своєчасно та обґрунтовано обрати необхідний і дієвий підхід із сотень можливих.

У роботі здійснено огляд механізмів та можливостей парадигми функціонального програмування (ФП) при веб-розробці у JavaScript та React.

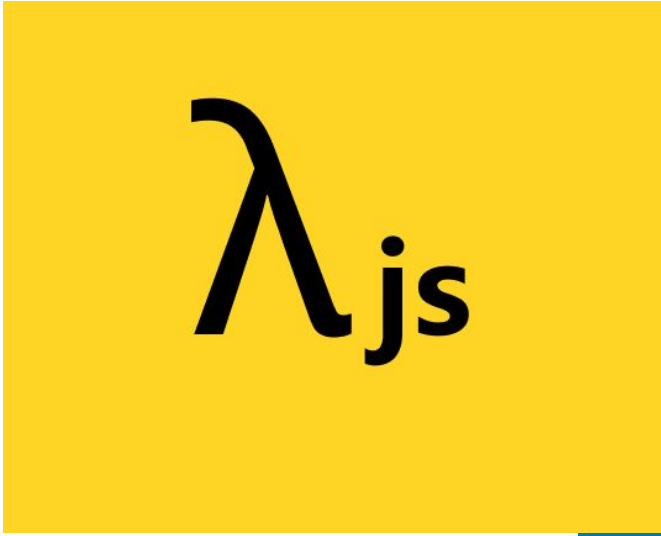
Мультипарадигмова мова програмування JavaScript

JavaScript — мультипарадигмова мова програмування.

Це універсальна мова, що дозволяє програмувати у різних стилях та використовувати декілька парадигм.

Недоліком мультипарадигмового підходу є те, що задля забезпечення ефективної роботи з об'єктами та масивами імперативний і об'єктно-орієнтований стилі припускають мутації.

Об'єкти мають бути мінливими, щоб значення їх властивостей могло бути змінено за допомогою методів.



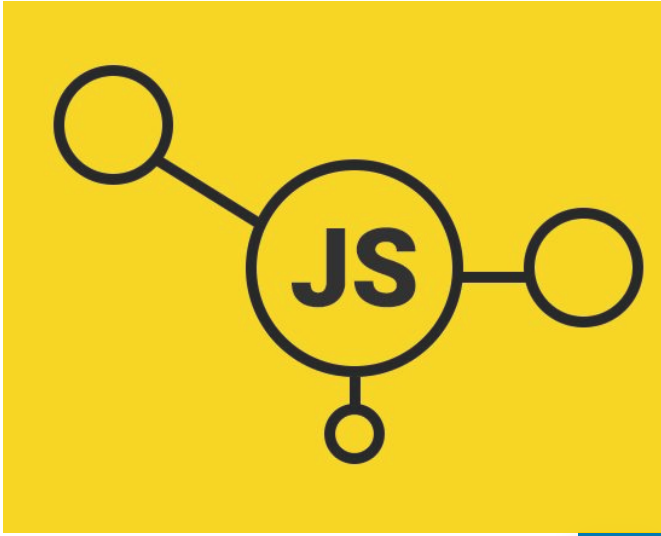
JavaScript

Умовно наявне у JS від ФП

- ✓ Функції першого класу
- ✓ Анонімні функції та лямбда-синтаксис
- ✓ Замикання

Умовно немає, або реалізація вважається небезпечною

- – Чистота
- – Незмінність, або імутабельність



Проблеми чистих функціональних мов програмування

Відомо, що функціональний підхід передбачає обгортання у функції практично всього поспіль. Доводиться писати багато маленьких функцій, що багаторазово використовуються, і викликати їх одну за одною, щоб отримати результат на кшталт **(func1.func2.func3)** або **func1(func2(func3())).**

Як обробляти умову if-else?

Монада Either.

Як обробляти винятки Null?

Монада Maybe.

Як переконатися, що дані, які надано у функції, не змінені і можуть використовуватися і в інших місцях?

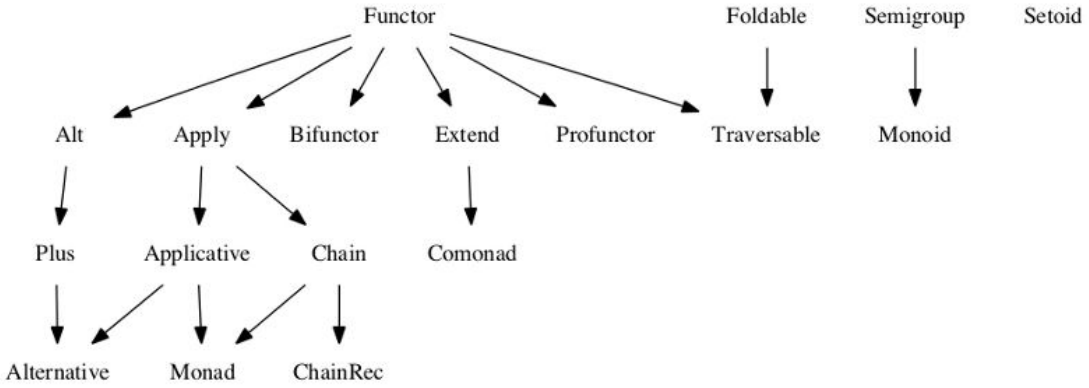
Pure functions, незмінність.

Як зробити функцію, яка приймає кілька значень, частиною ланцюжка (chaining), коли він приймає лише одне?

Карірування та функції вищого порядку.

Монада в безлічі X та специфікація Fantasy-Land

Частіше за все, академічні визначення функціонального програмування зрозуміти складніше, ніж сприйняти ідею. Академічною ознакою монади є ось це: Монада в безлічі X — це моноід з категорії ендофункторів X, в якому морфізм, званий "твор", замінений композицією ендофункторів, а морфізм, званий "одиниця", замінений ендофунктором "тотожність". Через те, що «монада» — це дуже абстрактне поняття, яке не має прямої аналогії з об'єктами реального світу, наукова ознака не дуже легка для розуміння.



```
// map, filter, reduce и т.д.
[0, NaN, Infinity].filter(Boolean)

// обіцянки (проміси)
new Promise((res) => setTimeout(res, 300))

// обробники подій
document.addEventListener('keydown', ({code, key}) => {
    console.log(code, key)
})
```

Приклади функцій вищого порядку

```
const memo = (fn, cache = new Map) => param => {
  if (!cache.has(param)) {
      cache.set(param, fn(param))
  }
  return cache.get(param)
}

const f = memo((x) => x * Math.sin(1 / x))
f(0.314) // обчислити
f(0.314) // взяти з кешу
```

Приклад мемоізації JS

Прийоми ФП у JS

Але і без монад чи аплікаторів JS має чимало прийомів функціонального програмування: композиція, часткове застосування, карування, чисті функції, та багато інших підходів. Одні можливості становляться базою для інших, більш складних.

Завдяки функціям першого класу стають можливими функції вищого порядку, які, в свою чергу, уможливлюють замикання. А завдяки замиканням з'являється мемоізація.

Використання функції як даних, тобто функції першого класу, уможливлює існування функцій вищого порядку, що, у свою чергу, призводить до появи часткового застосування, карування та композиції. Функції вищого порядку – це функції, які приймають чи повертають інші функції.

```
function impure (min, max) {
    return Math.floor(Math.random() * (max - min + 1) + min)
}
impure(1, 10) // 4
impure(1, 10) // 2

function pure (min, max, random = Math.random()) {
    return Math.floor(random * (max - min + 1) + min)
}
pure (1, 10, 0.42) // 5
pure (1, 10, 0.42) // 5
```

Приклади чистої функції у роботі з випадковими значеннями

```
const refTransparency = () => Math.pow(2, 5) + Math.sqrt(100)
refTransparency() // виклик функції
// або
Math.pow(2, 5) + Math.sqrt(100) // можна розкрити і зрозуміти результат
32 + 10 // 42
```

Прозорість посилань у чистих функціях

Чисті функції

Чисті функції завжди повертають той самий результат для тих самих параметрів.

Крім того, чисті функції мають прозорість посилань. Це ефект, який дозволяє замість виклику функції підставити результат роботи.

Переваги чистих функцій:

- ✓ простіше розібратися, як влаштована функція;
- ✓ їх можна запросто кешувати;
- ✓ легко тестувати;
- ✓ легко розпаралелювати.

Найчистіша функція: **(() => {})()**

Найважливіші підходи ФП, які представлено в React

- Функції можуть бути присвоєні іншим змінним;
- декларативне оголошення функції;
- чиста функція без побічних ефектів;
- генерація функції;
- функція вищого порядку;
- комбінації функцій (compose) та керування (curry).

Створення обробника подій в Redux: **const middleware = store => next => action => {...}**

Суть процесу рендерінгу сторінки React – це вкладений процес виклику функцій. Компонент передається батьківським компонентом крізь props.

Це дає контроль, але субкомпоненти не можуть бути змінені безпосередньо всередині props, тобто це повинен бути єдиний потік даних, який у чомусь схожий на характеристики чистої функції та не змінює зовнішній стан.

Використання функціональної парадигми надає React таких корисних характеристик

- Швидкість розробки висока;
- функції, що часто використовуються, можуть постійно повторно використовувати логіку;
- можна легко зрозуміти корисність функції через ім'я, не знаючи внутрішньої реалізації;
- код більш зрозумілий;
- метод є чистою функцією і не впливає на зовнішні змінні;
- послідовність обробки може бути організована довільно.

React-hooks

Використання хуків дозволяє виділити логіку, яка керує побічним ефектом. Раніше цю логіку доводилося розбивати на різні методи життєвого циклу у компоненті. І оскільки з'явилися хуки мінімізації, з'явився React.memo, тепер функціональні компоненти піддаються мемоізації, тобто компонент не буде перетворено або оновлено, якщо його пропси не змінилися.

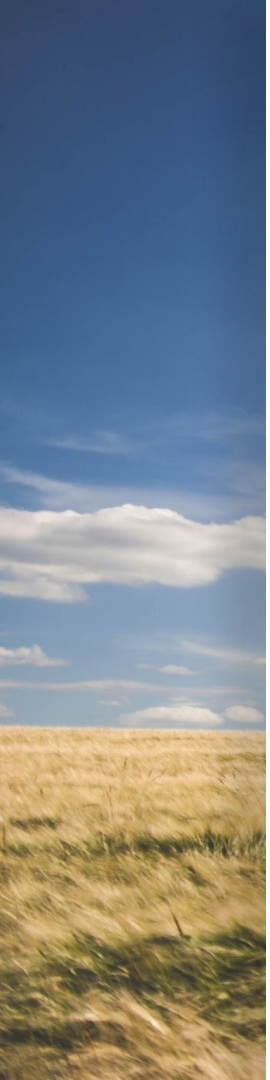
Хуки це просто JavaScript функції, і вони вимагають всього двох правил:

1. Виконувати хуки слід у верхній частині ієрархії функції. Не можна викликати всередині умов, циклів, вкладених функцій тощо. Якщо потрібно, можна визначити кастомний хук і викликати з нього. Інакше реакт не зможе гарантувати порядок виконання хуків.
2. Не можна використовувати хуки у компонентах на класах, тільки в React-функціях чи функціональних компонентах, або викликати хуки з інших кастомних хуків.

Переваги React-hooks

- Легше ділитися логікою стану (кастомні хуки);
- рятування від HOC і wrapper-hell;
- ефективніша мінімізація;
- не потрібно зберігати імена методів;
- не потрібно працювати з прототипами класів;
- не потрібно пам'ятати про this;
- спрощення логіки, пов'язаної із життєвим циклом;
- гнучкіша можливість оптимізації за рахунок мемоізації.

> Краще ментальне правило, яке застосовується до хуків, полягає в тому, що програмувати треба так, ніби будь-яке значення може змінитися в будь-який час.



Висновки

Функціональний підхід та хуки надають досить великі можливості. Починаючи з версії 7.1.0, React-Redux підтримує API хуків та надає такі хуки як useDispatch та useSelector, а починаючи з версії 5.1, з'явилася підтримка хуків і в React-Router. Екосистеми Java Script та React мають усі можливості для використання функціональної парадигми при створенні веб-застосунків.

Розробники React офіційно обіцяють і далі підтримувати класи, але відверто кажуть, що саме хуки стануть головним способом написання React-компонентів.

Дякую за увагу!