JavaScript
함수형 프로그래밍의 이해

2024.10.6

두선아

고차 함수

이터레이터

제너레이터

다형성

목 차

함수형 프로그래밍의 개요

함수형 프로그래밍

순수 함수와 데이터를 변환을 중점에 둔 프로그래밍 패러다임 ✅

코드의 가독성, 유지보수성, 재사용성을 높입니다 👍

1
순수 함수
Pure Function

2
일급 함수
First-Class Function

3
고차 함수
Higher Order Function

4
다형성
Polymorphism

순수 함수: 동일한 입력에 대해 항상 동일한 출력을 반환, 사이드 이펙트가 없음
즉, 예측이 가능하고, 부수 효과가 없음
같은 조건에서 항상 같은 결과를 보장

```
add = () => a + b
```

일급 함수

함수가 다른 데이터와 동일하게 취급될 수 있는 특성

자바스크립트의 함수는 일급 객체 first-class citizen 입니다

```
const sayHello = () => console.log("Hello!");
const executeFunction = (fn) => fn();
executeFunction(sayHello); // Hello!
```

- 함수를 변수에 할당 가능
- 함수의 인자로 전달 가능
- 함수의 반환값으로 사용 가능

재사용성과 추상화를 용이하게 합니다

고차 함수

함수를 인자로 받거나 함수를 반환하는 함수

📌 함수를 인자로 받아서 실행하는 함수
Applicative function

```
const num = 10;
const apply = (fn) => fn(num);
// 함수 fn에 num 값을 전달하여 실행
const add10 = (a) => a + 10;
const addSmile = (a) => a + "😀";
```

```
const times = (fn, n) => {
  for (let i = 0; i < n; i++) {
    fn(i);
  }
};
```

내부에서 인자를 적용해 여러 번 실행하는 경우

📌 함수를 만들어 리턴하는 함수
클로저를 생성하여 외부 변수에 접근합니다

```
const addMaker = (a) => (b) => a + b;
// 매개변수 a를 기억하고, b를 더하는 함수 리턴
const add20 = addMaker(20);

console.log(add20); // (b) => 20 + b
console.log(add20(10)); // 30
```

이터러블 이터레이터

iterable은 프로토콜(규약)이다.

= 배열, 문자열, Set, Map, arguments, 사용자 정의 이터러블

Symbol.iterator를 구현한 객체는 이터러블로 취급됩니다

```
const arr = [1, 2, 3];
const iterator = arr[Symbol.iterator]();
console.log(iterator.next());
// { value: 1, done: false }

arr[Symbol.iterator] = null;
console.log(iterator.next()); // 1️⃣
for (const num of arr) console.log(num); // 2️⃣
```

Symbol.iterator가 없는 배열은?

=> 이터러블이 아님

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols

이터레이터의 동작

{ value, done }

이터레이터? next() 메서드를 사용해 순차적으로 값을 생성하는 객체

```
// [Symbol.iterator]() { return this; }
const iteratorSymbol = arr[Symbol.iterator].bind(this); // 1️⃣
const iterator = arr[Symbol.iterator](); // 2️⃣
```

👈 context가 필요!

```
const map = new Map([
  ["a", 1],
  ["b", 2],
  ["c", 3],
]);
const mapIterator = map[Symbol.iterator]();
mapIterator.next(); // { value: ["a", 1], done: false }
for (const a of mapIterator) console.log(a);
// ["b", 2] ["c", 3] (key, value)
```

next() 메서드 실행 👉

```
map.keys(); // MapIterator { "a", "b", "c" }
map.values(); // MapIterator { 1, 2, 3 }
map.entries(); // MapIterator {'a' => 1, 'b' => 2, 'c' => 3}
```

keys, values, entries 메서드도 이터레이터를 반환합니다

🤔 이터레이터도 이터러블이에요

https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Iteration_protocols#returnvalue

제너레이터

iterator를 생성한다

제너레이터는 이터레이터를 쉽게 생성할 수 있는 function* (){}

well-formed iterator의 Symbol.iterator는
다시 자기자신, Symbol.iterator를 반환합니다

```
function* generator() {
  yield 1;
  yield 2;
  yield 3;
  return "🥺";
}
const iterGenerated = generator();
```

function* 구문을 사용하여 정의하며
yield 키워드로 값을 하나씩 반환하고
실행을 일시 중단합니다

```
const customIterable = {
  [Symbol.iterator]() {
    let i = 1;
    return {
      next() {
        return i == 3 ? { done: true } : { value: i++, done: false };
      },
      [Symbol.iterator]() {
        return this;
      },
    };
  },
};
let wellFormIterator = customIterable[Symbol.iterator]();
```

전개 연산자, 구조분해할당, 나머지 연산자

```
function* infinity(i = 0) {
  while (true) yield i++;
}

function* limit(l, iter) {
  for (const value of iter) {
    yield value;
    if (value === l) return;
  }
}

function* odds(l) {
  for (const value of limit(l, infinity(1))) {
    if (value % 2) yield value;
  }
}

for (const value of odds(10)) {
  console.log(value); // 1 3 5 7 9
}
```

odds: 3개의 Generator로 조합된 Generator 함수

객체가 이터러블 프로토콜을 따른다면
for...of, 전개 연산자, 구조분해할당, 나머지 연산자 등
여러 곳에 사용할 수 있습니다

```
// Spread operator
console.log(...odds(5)); // 1 3 5
console.log([...odds(1), ...odds(10)]);
// [1, 1, 3, 5, 7, 9]

// Destructuring
const [head, ...tail] = odds(10);
console.log(head); // 1
console.log(tail); // [3, 5, 7, 9]

// Rest parameter
const [a, b, ...rest] = odds(10);
console.log(a); // 1
console.log(b); // 3
console.log(rest); // [5, 7, 9]
```

다양한 상황에서
사용할 수 있는 성질을
다형성이라 합니다

다형성과 함수형 프로그래밍

여러 데이터 구조에서 동일한 인터페이스로 작업할 수 있는 능력

데이터 변환을 위한
map의 동작을 구현한 함수 👉

```
const map = (f, iter) => {
  let res = []; // 순수함수
  for (const value of iter) {
    res.push(f(value));
    // 어떤 값을 수집할지 콜백함수에게 완전히 위임
  }
  return res;
};
```

다양한 데이터 구조에 대해
동일한 방식으로 작업할 수 있습니다

```
console.log(map((a) => a + "⭐", odds(5)));
// ["1⭐", "3⭐", "5⭐"]

const m = new Map();
m.set("점심", "🥩");
m.set("저녁", "🍗");
console.log(map(([k, v]) => [k, v + " 조림"], m));
// [["점심", "🥩 조림"], ["저녁", "🍗 조림"]]
```

다형성은
함수형 프로그래밍의
유연성을 극대화합니다

함수형 프로그래밍의 패턴들

조건 필터링, 축약을 위한
filter, reduce의 동작을 구현한 함수

```
const filter = (f, iter) => {
  let res = [];
  for (const value of iter) {
    if (f(value)) res.push(value);
  }
  return res;
};

const reduce = (f, acc, iter) => {
  if (!iter) {
    iter = acc[Symbol.iterator]();
    // 3번째 인자가 없으면 (initialValue가 없으면)
    // iter에 acc의 이터레이터를 할당
    acc = iter.next().value; // 1번째 값
  }
  for (const value of iter) {
    acc = f(acc, value);
    // callback 함수는 acc와 value를 받아서
    // acc에 누적된 값을 반환한다.
  }
  return acc;
};
```

```
const menu = [
  { name: "랍스터 마라 크림 짬뽕", price: 42000 },
  { name: "캐비아 모둠전", price: 26000 },
  { name: "마라 크림 새우 딤섬", price: 24000 },
  { name: "대통령 명장 텐동", price: 22000 },
];
console.log(
  "가격만 뽑아낸 배열",
  map((p) => p.price, menu)
); // [42000, 26000, 24000, 22000]
console.log(
  "가격이 25000 이상인 제품의 가격을 모두 더한 값",
  reduce(
    add,
    filter(
      (p) => p > 25000, // [42000, 26000, 24000, 22000]
      newMap((p) => p.price, menu) // [42000, 26000]
    )
  )
); // 68000
```

- map과 filter를 조합해 특정 조건에 맞는 데이터를 추출
- reduce로 축약

이러한 중첩 사용은 코드를 간결하고 직관적으로 만듭니다.

여러 개의 함수를 조합하여
복잡한 로직을 구현할 수 있는 점이
함수형 프로그래밍의 큰 장점입니다.

결론

함수형 프로그래밍의 장점

코드의 예측 가능성 증가

부수효과 최소화

테스트 코드 작성 용이

다형성을 활용해 다양한 데이터 구조에 대해 동일한 작업 수행

어떻게 잘 쓸 수 있을지
고민해봐요✨

thank you!

reference & link

MDN : https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols

인프런 - 함수형 프로그래밍과 JavaScript ES6+ : https://www.inflearn.com/course/functional-es6

월간 CS - 함수형 프로그래밍 스터디: https://github.com/monthly-cs/2024-10-functional-es6