Introduction Functional Programming

関数型プログラミングの考え方を取り入れて予測しやすいコードを書く

ken7253

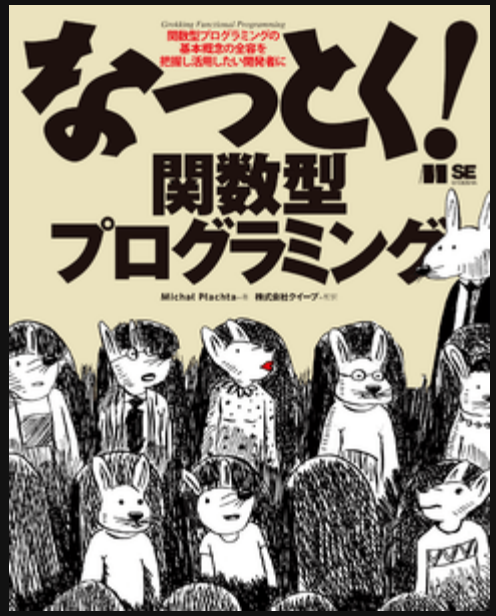
Frontend developer

技術記事を書いたりするのが趣味。

最近はNext.jsを使ったアプリケーションを書いています。

インターフェイス設計やアクセシビリティ・SSG関連の技術に興味があります。

- https://github.com/ken7253
- https://zenn.dev/ken7253
- https://dairoku-studio.com



前日譚

「なっとく！関数型プログラミング」

という本を買って読んでいました。

- 参考になる内容が多かった
- 関数型の考え方を紹介

ガチガチの関数型は難しいけど

少し取り入れてコードの品質を上げる

関数型プログラミングの考え方を部分的に採用し

フロントエンドでも馴染み深いライブラリといえば？

React

The library for web and native user interfaces

Learn React | API Reference

Create user interfaces from components

React lets you build user interfaces out of individual pieces called components.
Create your own React components like `Thumbnail`, `LikeButton`, and `Video`.
Then combine them into entire screens, pages, and apps.

Video.js

話すこと

- 純粋関数について
- シグニチャーを意識して関数を作る

純粋関数

純粋関数とはなにか

関数の分類の一つで特に関数型プログラミングで重要な考え方

下記の条件を**全て満たした関数**のことを純粋関数と呼ぶ

- 引数のみを利用する
- 戻り値は常に一つだけ
- 既存の変数を変更しない

純粋関数の条件

「引数のみを利用する」とは下記のようなこと

```
// 🙆 純粋関数
const pureFunc = (x:number, y:number): number => {
  return x * y;
};

// 🙅 純粋関数ではない
let x = 0;
let y = 0;
const notPureFunc = (): number => {
  return x * y;
};
```

グローバル変数を参照したり、時刻に依存するコードなど。

純粋関数の条件

戻り値は常に１つだけ。

```
// 👮 純粋関数ではない
const notPureFunc = (a: number): number | void => {
  if (a === 0) {
    throw new Error('Error');
  } else if (a >= 100) {
    return a
  }
  // 値を返さない場合もある
}
```

状況によってエラーを発生させたり、そもそも値を返さない場合がある関数など。

純粋関数の条件

既存の変数を変更しない

```
let num = 0;
// 👩 純粋関数
const pureFunc = (n:number): number => {
  return n + 1;
};
pureFunc(num); // num => 0
```

```
let num = 0;
// 👨‍🎨 純粋関数ではない
const notPureFunc = (n:number): number => {
  return n++;
};
notPureFunc(num); // num => 1
```

`Array#push()` や `Array#reverse()` などのメソッド、 `var` / `let` に対する再代入

などを関数の内部で呼び出して、既存の値を変更している場合など

純粋関数だと何が嬉しいか

- テストしやすい
- 意図しない挙動を起こしづらい
- コードの挙動が予測しやすい

副作用を分離することは **「臭いものに蓋をする」** だけに見える。

実際は、処理同士の繋がりが **型によって可視化** されてわかりやすいコードになる。

シグニチャー

シグニチャーとはなにか

シグニチャー（シグネチャ）とは下記の情報をまとめた呼称

- 関数名
- 引数の型
- 返り値の型

シグニチャーの情報が少ない場合

下記のような関数があるとする。

```
export const foo: (a: any, b: any): any => {
  // 外部からは見えない何らかの処理
}
```

この状態で何をしている関数であるか理解できる人は少ない。

シグニチャーの情報が少ない場合

シグニチャーをより正確にしてみる

```
export const sum = (first: number, second: number): number | TypeError => {
  // 外部からは見えない何らかの処理
}
```

これにテストを追加してみる。

```
import { sum } from "./sum"

describe('与えられた引数を足し算して返却するsum関数', () => {
  test('自然数同士の足し算が正しく実行されること', () => {/* 省略 */})
  test('引数のどちらかにNaNが渡された場合TypeErrorを返却すること', () => {/* 省略 */})
  test('引数のどちらかにInfinityが渡された場合TypeErrorを返却すること', () => {/* 省略 */})
})
```

こうすることで（テストが無くても）ある程度挙動が推測できる。

情報の多い関数は適切に使用できる

例としてこの関数を実際に使ってみる。

```
import { sum } from "./sum";
import { sendError } from "./sendError";

const [x,y] = [10, 20];
const sumResult = sum(x,y); // number | TypeError

// そのままだと型が合わないので型ガードを利用する。
if (sumResult instanceof TypeError) {
  // sendError = (error: Error) => void;
  sendError(sumResult); // sendError(); はエラー情報をサーバーに送る処理として考える
} else {
  console.log(sumResult);
}
```

このようにシグニチャーから関数の使い方が理解できる。

具体的なコードで見てみる

複数の要素の中から一番高さを持つ要素を探してコンソールに出力する処理

```
// 入出力の情報がないので説明のための関数名が長くなりがち
const displayHighestElementByElementList = () => {
  // 要素の取得
  const elements = document.querySelectorAll('.some-class');
  const elementList = Array.from(elements);
  // 比較とソート
  const sortedFromClientHeight = [...elementList].sort((prev, next) => {
    return next.clientHeight - prev.clientHeight;
  });
  // コンソールへの出力
  console.log(sortedFromClientHeight[0]);
};

displayHighestElementByElementList();
```

一つの関数にいろいろやらせている例

このようなコードがあった場合、次のように変えてみる。

具体的なコードで見てみる

複数の要素の中から一番高さを持つ要素を探してコンソールに出力する処理

```
// 与えられた要素の配列から一番高さを持つ要素を返す関数
const getHighestElement = (elementList: Element[]): Element => {
  const sorted = [...elementList].sort((prev, next) => {
    return next.clientHeight - prev.clientHeight
  });

  return sorted[0];
};
// 要素の取得
const elementList = Array.from(document.querySelectorAll('.some-class'));
// コンソールへの出力
console.log(getHighestElement(elementList));
```

- 要素の取得 -> 比較関数 -> 出力 という値の流れが掴みやすい。
- 入出力の型情報があることで挙動が推測しやすい関数になる。

具体的なコードで見てみる

複数の要素の中から一番高さを持つ要素を探してコンソールに出力する処理

```
// 与えられた要素の配列から一番高さを持つ要素を返す関数
const getHighestElement = (elementList: Element[]): Element =>
  elementList.reduce((acc, current) =>
    acc.clientHeight >= current.clientHeight ? acc : current
  );
// 要素の取得
const elementList = Array.from(document.querySelectorAll(".some-class"));
// コンソールへの出力
console.log(getHighestElement(elementList));
```

もっと関数型っぽい書き方だとこう。

配列のメソッドをうまく使って無駄なく宣言的に記述する。

まとめ

関数を設計するときにいくつか持つとよい視点がある。

- 純粋関数という視点を持つ
- 関数の設計を行う場合はシグニチャーに情報を持たせる