

カスタムHooksと単体 テストの共通点について

@CTOA若手エンジニアコミュニティ勉強会 #5

ken7253

Frontend developer

技術記事を書いたりするのが趣味。

最近はReactを使ったアプリケーションを書いています。

ユーザーインターフェイスやブラウザが好き。

 <https://github.com/ken7253>

 <https://zenn.dev/ken7253>

 <https://dairoku-studio.com>



Hooks（カスタムHooks）とは

- 組み込みのHooks(`useState` / `useEffect` など)を組み合わせて作る関数
- ルールは組み込みHooksと一緒に
 - `use***` という命名規則を持つ
 - コンポーネントもしくはHooksのトップレベルでのみ実行できる

サードパーティで有名なHooks

- `useQuery`(Tanstack Query)
- `useRouter`(next/router)
- `useAtom`(jotai)

特殊な制約を持った関数ぐらいの認識でもOK

<https://ja.react.dev/reference/rules/rules-of-hooks>

関数の単体テストについて簡単に確認

単体テストはどのように書くか

例として、引数として与えられた配列を全て足し合わせる `sum` 関数を考える。

```
export const sum = (array: number[]): number => {
  if (array.some(v => v === Infinity || v === -Infinity)) {
    return Infinity;
  }

  return array.reduce(
    (a, c) => a + (Number.isNaN(c) ? c : 0),
    0
  );
}
```

- 基本的には配列の加算
- `Nan`があった場合 `0` として扱う（無視する）
- `Infinity` が含まれていた場合は常に `Infinity` を返す

この関数に対してのテストを書く想定

単体テストはどのように書くか

よくあるのは関数に引数を渡して、返り値を検査するパターン。

```
import { describe, test, expect } from "vitest";
import { sum } from "./index.ts";

describe('引数として与えられた配列を全て足し合わせるsum関数', () => {
  describe('引数が全て有効な数値の場合', () => {
    test('配列を足し合わせた数値が返却される', () => {
      const array = [1, 2, 3, 4, 5];
      const sumResult = sum(array);

      expect(sumResult).toBe(15);
    })
  });
}

describe('計算不能な数値型が含まれている場合', () => {
  test('NaNが含まれていた場合0として扱う', () => { /* 略 */ });
  test('Infinityが含まれていた場合常にInfinityを返却する', () => { /* 略 */ });
});
});
```

単体テストと純粹関数

このとき関数自体が純粹関数ではない場合テストが書きづらい。

下記のコードは `Math.random()` は実行毎に値が変わってしまうのでテストしづらい。

```
export const sum = (array: number[]) => {
  if (array.some(v => v === Infinity || v === -Infinity)) {
    return Infinity;
  }

  const sumAll = array.reduce(
    (a, c) => a + (Number.isNaN(c) ? c : 0), 0
  );

  return sumAll * Math.random();
}
```

単体テストと純粹関数

このとき関数自体が純粹関数ではない場合テストが書きづらい。

下記のコードは `Math.random()` は実行毎に値が変わってしまうのでテストしづらい。

```
export const sum = (array: number[], randomize: number) => {
  if (array.some(v => v === Infinity || v === -Infinity)) {
    return Infinity;
  }

  const sumAll = array.reduce(
    (a, c) => a + (Number.isNaN(c) ? c : 0), 0
  );

  return sumAll * randomize;
}
```

副作用を除去してテストしやすい関数を作る

副作用は外部から渡して純粋関数にする。

テストをするときは `randomize` に固定値を入れれば保証したいロジックを検査できる。

```
export const sum = (array: number[], randomize: number) => {
  if (array.some(v => v === Infinity || v === -Infinity)) {
    return Infinity;
  }

  const sumAll = array.reduce(
    (a, c) => a + (Number.isNaN(c) ? c : 0), 0
  );

  return sumAll * randomize;
}
```

単体テストの考え方をHooksにも適用する

単体テストの考え方をHooksにも適用する

- テストしづらい副作用は外部から受け取る
- テストコードが簡潔になるようにI/Fを設計する

単体テストの考え方をHooksにも適用する

```
import { useAtom } from "jotai";
import { useRouter } from "next/router";
import { someAtom } from "@/store/someAtom";

export const useFizzBuzz = () => {
  const { pathname } = useRouter();
  const [fizzBuzz, setFizzBuzz] = useAtom(someAtom);
  const result =
    parseInt(pathname, 10) % 15 === 0
      ? "FizzBuzz"
      : parseInt(pathname, 10) % 5 === 0
      ? "Fizz"
      : parseInt(pathname, 10) % 3 === 0
      ? "Buzz"
      : parseInt(pathname, 10);

  setFizzBuzz([...fizzBuzz, result]);
  return fizzBuzz;
};
```

pathname を参照元にFizzBuzzをしてその履歴をstoreに格納するHooks

依存を整理する

このHooksをテストしやすいように修正してみる。

```
import { useAtom, type Atom } from "jotai";

export const useFizzBuzz = <T>(pathname: string, atom: Atom<T>) => { // atomも引数として渡す
  const [fizzBuzz, setFizzBuzz] = useAtom(atom);
  const result =
    parseInt(pathname, 10) % 15 === 0
      ? "FizzBuzz"
      : parseInt(pathname, 10) % 5 === 0
      ? "Fizz"
      : parseInt(pathname, 10) % 3 === 0
      ? "Buzz"
      : parseInt(pathname, 10);

  setFizzBuzz([...fizzBuzz, result]);
  return fizzBuzz;
};
```

関数とHooksのテストの共通項

Hooks

```
import { useAtom, type Atom } from "jotai";

export const useFizzBuzz = <T>(
  pathname: string, atom: Atom<T>
) => {
  const [fizzBuzz, setFizzBuzz] = useAtom(atom);
  const result =
    parseInt(pathname, 10) % 15 === 0
      ? "FizzBuzz"
      : parseInt(pathname, 10) % 5 === 0
      ? "Fizz"
      : parseInt(pathname, 10) % 3 === 0
      ? "Buzz"
      : parseInt(pathname, 10);

  setFizzBuzz([...fizzBuzz, result]);
  return fizzBuzz;
};
```

Function

```
export const sum = (array: number[], randomize: number) => {
  if (array.some(v => v === Infinity || v === -Infinity)) {
    return Infinity;
  }

  const sumAll = array.reduce(
    (a, c) => a + (Number.isNaN(c) ? c : 0), 0
  );

  return sumAll * randomize;
}
```

- どちらも副作用は外部から受け取る
- シグニチャーの情報が増え分かりやすい

モジュールが持つ **責務**が引数の数に現れる
のでそれを基準にコード分割のタイミングを探る

まとめ

- Hooksのテストであっても考え方は単体テストと変わらない
- シグニチャーの情報量を増やして意外性のないHooksになる

シグニチャー：関数名、関数の引数の型、返り値の型などの情報のこと