

JavaScript関数型プログラミング入門

JavaScript es6 関数型プログラミング es2015 ECMAScript2015

JavaScript関数型プログラミング 入門

1. 概念編 (*concept*)

1.1. 関数型プログラミングって何だ？

関数だけでコードを組み立てるスタイルのこと。

```
result = fuga(hoge(value));
```

※ 命令型プログラミング (Imperative Programming)

```
let a = 1;  
let b = 2;  
let c = a + b;
```

1.2. 関数って何だ

数学者が使う「関数」という言葉と全く同じ意味である。例：平方計算関数

$$f(x) = x * x;$$

- 戻り値が必ずある
- 渡される引数が同じであれば、常に同じ結果を返す（参照透過性）
- 外部の変数を決して参照しない（const参照可）
- 関数の呼出は、引数の値を変えることはない
- 関数の目的は、引数から値を計算することのみである

- 関数はそれ以外にその世界に影響（画面表示、ログ出力、ファイル書き込み、音を出すなど）を与えない
- 関数は外の世界からの情報を受け取らない。例えば：キーボードやハードディスクを読んだりしない

1.3. *JavaScript*関数型プログラミングって何だ？

JavaScript言語を使って関数型プログラムを書くこと。

- JavaScriptの関数は「ファーストクラスオブジェクト（第一級オブジェクト）」である。

変数に代入したり、配列にセットしたり、他の関数に食わせたりできる。

- JavaScriptいろんなスタイルでプログラミングすることができる
 - 命令型
 - オブジェクト指向(OOP)

○ 関数型

目的に基づいて適するスタイルを選ぶのがポイントである。

1.4. 「綺麗」「汚い」の意味

ソースコードに対する評価の一つ、よく"綺麗"とか、"汚い"とかが耳に入りますが、関数型プログラミングでこれらの言葉は特別な意味を持つ。

- コードの副作用

プログラムコードが外の世界に見えるが、コンピュータを「ピー」と鳴らすなり、スクリーンにダイアログを出したり、といったことをする場合、このコードは副作用を持つと言う。

自身以外の世界を1つでも改変すれば、副作用があると言える。

- 「綺麗」

副作用がなければ、「綺麗」だ！

```
function square( x ) {  
    return x * x;  
}
```

- 「汚い」 副作用のある命令的なコードこそ、「汚い」！

```
let value = 2;  
value = value * value;  
alert(value);
```

- うまく分割することが大事

関数型プログラミングでは、プロジェクト実の状況によって、綺麗な部分と汚い部分をバランスよく分割することが重要である。

例えば：綺麗な関数型の部分を 8 割、汚い命令型の部分を 2 割とするとか。

```
function square( x ) {  
    return x * x;  
}
```

```
const value = 2;  
const result = square( value );  
alert(result);
```

- 副作用を減らす方法

依存する外部変数やコンテキストなどを引数として
抽出する

```
// 外部変数  
let dayCnt = 5;  
  
// 副作用あり  
function getDate() {  
    // 外部変数とカレント時刻コンテキストを参照  
    return moment().add(dayCnt, 'days');  
}  
  
let date = getDate();  
  
  
// 副作用なし  
function getDateAfterDay(date, dayCnt) {  
    // 外部変数とカレント時刻コンテキストを参照  
    return moment(date).add(dayCnt, 'days');  
}
```

```
// 外部変数
const dayCnt = 5;
const date = getDate();
```

1.5. 高階プログラミング

高階関数を使って、プログラミングすること。

高階関数：関数を引数にしたり、あるいは関数を戻り値とするような関数のこと。

```
// アロー関数を使って、配列の各要素の値の2倍で構成される配列を作りたい
const twoMultipleArray = [1, 2, 3].map( (value) => 2 * value )
```

1.6. 関数型プログラミングはなぜクレイジーか

- **副作用**を起こせないだからこそ、関数型プログラミングは、現実には何もできない。

清浄な部分と不浄な部分を分けることで解決できた。

- プログラムがもの凄く**非効率**になり得る。
 - 常に新しいを作り出す制約で、大量のメモリ確保とコピーが必要とする場合がある。
 - ループの代わりに再帰を多用するより、スタックに沢山の情報を積まねばならず。

```
const list = [  
  {  
    name : 'foo',  
    age  : 21  
  },  
  {  
    name : 'bar',  
    age  : 10  
  },  
  {  
    name : 'hoge',  
    age  : 23  
  }  
];  
  
const list2 = list.map(({ name, age }) => {
```



```
    return {  
      name,  
      age : age + 1  
    };  
  });
```

```
console.log(list);  
console.log(list2);
```

```
// 0 1 1 2 3 5 8 13 ...
```

```
function fibonacci(n) {  
  return n < 2 ? n : (fibonacci(n - 1) + fibonacci(n - 2));  
}
```

```
// 0 1 1 2 3 5 8 13 ...
```

```
function fibonacci2(n) {  
  if (n < 2) {  
    return n;  
  }  
}
```

```
function fib(prev, curr, leftCnt) {  
  return leftCnt === 0 ? curr : fib(curr, prev + curr, leftCnt - 1);  
}  
  
return fib(1, 1, n - 2);  
}
```

```
for (let i = 1; i < 10; i++) {  
  console.log(`${i}:${fibonacci(i)}`);  
  console.log(`${i}:${fibonacci2(i)}`);  
}
```

だが幸いなことに、ハードウェアとソフトウェアの進化やさまざまな最適化技

1.7. 関数型プログラミングはなぜ素晴らしいか

- 関数型プログラミングは**バグを減らす**

引数だけに依存するので、プログラムの振る舞いも、バグの再現も簡単になる

- 関数型プログラミングは**簡潔だ**

高階関数を使うことで、命令型プログラミングに出てくる大量の一時的な変数

- 関数型プログラミングは**エレガント**だ

最大の利点は、プログラミングを数学の領域に戻してくれることだ。

将来は、関数型プログラミングで作られたソースコードの正しさを証明できる

- **並列処理**のプログラミングモデルが簡単だ

JavaScriptがシングルスレッドのため享受できないが、一般論では、関数型：

2. 実用編 (*practical use*)

2.1. 配列のイテレーション

- **forEach()**

与えられた関数を、配列の各要素に対して一度ずつ実行する。

```
// 配列の内容を出力する

[2, 5, 9].forEach((val, idx) => {
  console.log(`a[${idx}] = ${val}`);
});

// logs:
// a[0] = 2
// a[1] = 5
// a[2] = 9
```

- **entries()**

配列内の各要素に対する key/value ペアを含む新しい Array Iterator オブジェクト を取得する。

```
// for文で配列の内容を出力する
var a = ['a', 'b', 'c'];
var iterator = a.entries();

for (let [idx, val] of iterator) {
  console.log(idx, val);
}

// 0, 'a'
// 1, 'b'
// 2, 'c'
```

2.2. 配列の変換

- **map()**

与えられた関数を配列のすべての要素に対して呼び出し、その結果からなる新しい配列を生成して返す。


```
// 配列の要素の2倍の値で構成する配列の生成
const numbers = [1, 4, 9];
const doubles = numbers.map((num) => {
  return num * 2;
});

// doubles is now [2, 8, 18]
// numbers is still [1, 4, 9]
```

- **filter()**

引数として与えられたテスト関数を各配列要素に対して実行し、それに合格したすべての配列要素からなる新しい配列を生成する。

```
// 小さい値をすべて取り除く
var filtered = [12, 5, 8, 130, 44].filter((value) => {
  return (element >= 10);
});
// filtered は [12, 130, 44] となる（10未満の配列要素が取り除かれて
```



2.3. 配列の検索

- `find()`

配列内の要素が指定されたテスト関数を満たす場合、配列内の 値 を返す。そうでない場合は `undefined` を返す。

```
// 素数を判定
function isPrime(value) {
  let start = 2;
  while (start <= Math.sqrt(value)) {
    if (value % start++ < 1) {
      return false;
    }
  }
  return value > 1;
}
```

```
console.log([4, 6, 8, 12].find(isPrime)); // undefined, not found
console.log([4, 5, 8, 12].find(isPrime)); // 5
```

- **findIndex()**

配列内の要素が指定されたテスト関数を満たす場合、配列内の インデックス を返す。そうでない場合は -1 を返す。

// 素数を判定

```
function isPrime(value) {
  let start = 2;
  while (start <= Math.sqrt(value)) {
    if (value % start++ < 1) {
      return false;
    }
  }
  return value > 1;
}
```

```
console.log([4, 6, 8, 12].findIndex(isPrime)); // -1, not found
console.log([4, 6, 7, 12].findIndex(isPrime)); // 2
```

2.4. 配列の判定

- **some()**

与えられた関数によって実行されるテストに合格する要素が配列の中にあるかどうかをテストします。

見つかったら、すぐtrueを返す。

```
// 10以上の数字あるかをチェック
```

```
function isBigEnough(element, index, array) {  
    return (element >= 10);  
}
```

```
let passed = [2, 5, 8, 1, 4].some(isBigEnough);  
// passed は false  
passed = [12, 5, 8, 1, 4].some(isBigEnough);  
// passed は true
```

- **every()**

与えられた関数によって実行されるテストに配列のすべての要素が合格するかどうかをテストする。


```
// 全ての要素が10以上の数字あるかをチェック
```

```
function isBigEnough(element, index, array) {  
  return (element >= 10);  
}
```

```
let passed = [12, 5, 8, 130, 44].every(isBigEnough);  
// passed is false  
passed = [12, 54, 18, 130, 44].every(isBigEnough);  
// passed is true
```

2.5. 配列の畳み込み

- **reduce()**

隣り合う 2 つの配列要素に対して（左から右へ）同時に関数を適用し、単一の値にする。

```
// 配列数字の合計値を計算
```

```
// 2回呼び出し  
([1, 4, 9].reduce((sum, val) => {  
  return sum + val;  
}
```

```
    }));  
    // 14
```

```
    // 3回呼び出し  
    ([1, 4, 9].reduce((sum, val) => {  
        return sum + val;  
    }, 0));  
    // 14
```

```
    // 1回呼び出し  
    ([1].reduce((sum, val) => {  
        return sum + val;  
    }));  
    // 1
```

```
    // 1回呼び出し  
    ([1].reduce((sum, val) => {  
        return sum + val;  
    }, 0));  
    // 1
```

```
    // 空配列、初期値設定あり  
    ([].reduce((sum, val) => {  
        return sum + val;  
    }, 0));  
    // 0
```

```
// 空配列、初期値設定なし
console.error(`${[]}.reduce((sum, val) => {
  return sum + val;
})}`);
// TypeError: Reduce of empty array with no initial value
```

2.6. クローザー

関数に閉じ込めた変数定義によって、シングルトン Instanceを簡単に作れる。

```
// シーケンス番号Generator
```

```
const Sequence = (() => {
  let count = 0;

  return {
    next: () => {
      return count++;
    }
  };
})();
```

```
Sequence.next(); // 0
```

```
Sequence.next(); // 1
```

```
// httpRequestMiddleware.js
```

```
// axiosのインスタンスを取得
```

```
const getAxiosInstance = (() => {
```

```
  let instance = null;
```

```
  return () => {
```

```
    // インスタンスが既に生成された場合はそのインスタンスを返す
```

```
    if (instance) {
```

```
      return instance;
```

```
    }
```

```
    // axiosインスタンス化
```

```
    instance = axios.create({
```

```
      // HTTP通信時の共通デフォルト設定はここです
```

```
      responseType : 'json',
```

```
      paramsSerializer(params) {
```

```
        return qs.stringify(params, { arrayFormat : 'repeat' } )
```

```
      }
```

```
    });
```

```
    return instance;
```

```
  };
```

```
})();
```

```
// getAxiosInstance().request(.....
```

2.7. 束縛関数および部分適用 (*bind*)

`bind()` メソッドは、呼び出された時に新しい関数を生成する。

最初の引数 `thisArg` は新しい関数の `this` キーワードにセットされる。

2 個目以降の引数は、新しい関数より前に、ターゲット関数の引数として与えられる。 (**部分適用**)

```
// 束縛関数
```

```
export default class OfferDetail extends React.Component {  
  
  constructor(props) {  
    super(props);  
    // イベントハンドラーをバインド  
    this.handleStarredClick = this.handleStarredClick.bind(thi  
  }
```

```
handleStarredClick() {
  const starred = this.props.offer ? this.props.offer.isStar
  // 検討中状態を設定したら、リストから削除
  this.props.updateStarredAndRemoveFromList(
    this.props.offerId,
    !starred
  );
}

renderActionBtnNode() {
  return (
    <div>
      <button
        onClick={this.handleStarredClick}
      >
        {starredBtnText}
      </button>
    </div>
  );
}

}
```

// 部分適用

```
function list(...elms) {  
  return [...elms];  
}  
  
const list1 = list(1, 2, 3); // [1, 2, 3]  
console.log(list1);  
  
// 先頭の引数がプリセットされた関数をつくる  
const leadingThirtysevenList = list.bind(undefined, 37);  
  
const list2 = leadingThirtysevenList(); // [37]  
console.log(list2);  
  
const list3 = leadingThirtysevenList(1, 2, 3); // [37, 1, 2, 3]  
console.log(list3);
```

2.8. カリー化 (*Curry*)

複数の引数を取る関数を、1つの引数のみを取る関数のチェーンに変換する。

カリー化されている関数に対して、異なる引数を渡すだけで別々の関数を**定義**できる。

(Haskell Brooks Curry : 数学者)

```
// 数字の判断条件
```

```
let over = (condition) => {  
  return function(num) {  
    return num > condition;  
  };  
};
```

```
// 10より大きいかを判断する関数を定義
```

```
let overTen = over(10);
```

```
console.log(overTen(8)); // => false  
console.log(overTen(20)); // => true
```

```
let over = (condition) => {  
  return function(num) {  
    return num > condition;  
  };  
};
```

```
let ary = [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

```
let newArray1 = ary.filter(over(10)); // ary.filter((x) => { return  
console.log(newArray1); // => [ 11, 12, 13, 14, 15 ]
```

```
let newArray2 = ary.filter(over(8));  
console.log(newArray2); // => [ 9, 10, 11, 12, 13, 14, 15 ]
```


2.9. ジェネレーター (*Generator*)

ジェネレーター関数は、イテレーターのファクトリーとして働く、特別な種類の関数である。この関数は実行すると新しいジェネレーターを返す。

function* 構文を使用している場合に、関数はジェネレーター関数となります。

```
// 数字の判断条件
```

```
function* g1() {  
  yield 2;  
  yield 3;  
  yield 4;  
}
```

```
function* g2() {  
  yield 1;  
  yield* g1();  
  yield 5;  
}
```

```
const iterator = g2();

console.log(iterator.next()); // { value: 1, done: false }
console.log(iterator.next()); // { value: 2, done: false }
console.log(iterator.next()); // { value: 3, done: false }
console.log(iterator.next()); // { value: 4, done: false }
console.log(iterator.next()); // { value: 5, done: false }
console.log(iterator.next()); // { value: undefined, done: true }

// for文でイテレーション
const iterator2 = g2();
for (const val of iterator2) {
  console.log(val);
}

// 1
// 2
// 3
// 4
// 5
```

参考情報

MDN 関数と関数スコープ

MDN 配列

bindメソッド

JavaScriptのカリー化

JavaScriptのジェネレーター