関数

関数は一連の処理に名前をつけてまとめたものです。他の人の作ったものを利用するだけでは関数などなくても、必要な処理を必要なだけ列挙すれば期待する結果が得られるコードは理論上は実現可能です。しかし、数万行の一連のコードを扱うのは事実上不可能です。

そこで理解できる大きさにグループ化して、名前をつけたものが関数です。関数呼び出しはネストできるので、難しいロジックに名前をつけて関数を作り、それらのロジックを並べたちょっと複雑なタスクを人間の仕事に近い高水準な関数にできます。関数は決まった処理を単純に実行するだけではなく、引数をとって、柔軟に動作させることもできますし、返り値を返すこともできます。

どの程度の分量が適切かはロジックの複雑さによります。単純な仕事を延々と行っている（Reactのコンポーネントのレンダリングなど）であれば、数画面分のコードでもなんとかなるでしょうし、帰って細かく分け過ぎてしまうと、全体像の把握が難しくなります。一方で複雑なロジックだと20行でも難しいかもしれません。

関数の基本形態は以下の通りです。

- 関数は名前を持ちます。何かに代入したり、引数で渡す場合は省略可能です。
- 引数はカンマ区切りで名前と型を列挙していきます。引数がない場合は省略可能です。引数の型はジェネリクスを使って省略可能にもできますが、基本的に指定が必要です。
- 返り値はreturnで返します。返り値がない場合は`return`を省略可能です。
- もしreturn文の数が一つ、または複数個あっても、毎回同じ型を返しているのであればreturnの型は省略可能です。

```
function 関数名(引数リスト): 返り値の型 {
  return 返り値
}
```

TypeScriptの関数には次のような特徴があります。

- 関数そのものを変数に入れて名前をつけられるし、データ構造にも組み込める
- 他の関数の引数に関数を渡せる
- 関数の返り値として返すことができる

関数を受け取る関数があると、プログラムの柔軟性が飛躍的に高まります。従来のJavaScriptは関数の使い勝手が極めて便利だった一方で、言語の他の機能は少なく、関数を多用した数々のテクニックが生み出されました。一方で、かなり黒魔術な、一見すると魔法のような使われ方も多数ありました。近年ではECMAScriptやTypeScriptのバージョンアップで数多くの機能が増え、トリッキーな使い方はだいぶ減っていますが、重要なことには変わりません。

関数にはいくつかのバリエーションがあります。

- 名前がない関数は無名関数、あるいはアノニマス関数と呼びます。変数に代入したり、他の関数の引数に渡す場所で利用されます。とくに、関数の内部で作られる関数を「クロージャ」と呼びます。
- 何かのオブジェクトに属する関数は「メソッド」と呼びます。
- 時間のかかる処理を行い、それが終わるまで他のタスクを途中で行える関数を非同期関数と呼びます（非同期処理の章で扱います）

名前のない無名関数にはアロー演算子を使った省略記法も追加されました。ふつうの`function`キーワードを使った宣言とほぼ同等で置き換えできることが多いのですが、動作については少し異なる部分もあります。こちらについても順をおって説明します。

```
const f = (引数リスト) => {
  return 返り値
}
```

関数はTypeScriptの中では、プログラムの構成をつくるための重要な部品です。いたるところで使われます。言語のバージョンアップとともに、定義、使い方などもいろいろ追加されました。表現したい機能のために、ややこしい直感的でないコードを書く必要性がかなり減っています。

関数の引数と返り値の型定義

TypeScriptでは関数やクラスのメソッドでは引数や返り値に型を定義できます。 元となるJavaScriptで利用できる、すべての書き方に対応しています。

なお、Javaなどとは異なり、同名のメソッドで、引数違いのバリエーションを定義するオーバーロードは使えません。

関数への型付け

```
// 昔からあるfunctionの引数に型付け。書く引数の後ろに型を書く。
// 返り値は引数リストの () の後に書く。
function checkFlag(flag: boolean): string {
  console.log(flag);
  return "check done";
}

// アロー関数も同様
const normalize = (input: string): string => {
  return input.toLowerCase();
}
```

変数の宣言のときと同じように、型が明確な場合には省略が可能です。

関数への型付け

```
// 文字列のtoLowerCase()メソッドの返り値は文字列なので
// 省略してもstringが設定されたと見なされる
const normalize = (input: string) => {
  return input.toLowerCase();
}

// 文字配列の降順ソート
// ソートに渡される比較関数の型は、配列の型から明らかなので省略してもOK
// 文字列のtoLowerCase()メソッドも、エディタ上で補完が効く
const list: string[] = ["小学生", "小心者", "小判鮫"];
list.sort((a, b) => {
  if (a.toLowerCase() < b.toLowerCase()) {
    return 1;
  } else if (a.toLowerCase() > b.toLowerCase()) {
    return -1;
  }
  return 0;
});
```

関数が何も返さない場合は、`: void` をつけることで明示的に表現できます。 実装したコードで何も返していなければ、自動で `: void` がついているとみなされますが、これから先で紹介するインタフェースや抽象クラスなどで、関数の形だけ定義して実装を書かないケースでは、どのように判断すればいいのか材料がありません。 `compilerOptions.noImplicitAny` オプションが `true` の場合には、このようなケースで `: void` を書かないとエラーになりますので、忘れずに書くようにしましょう。

何も返さない時は*void*

```
function hello(): void {
  console.log("ごきげんよう");
}

interface Greeter {
  // noImplicitAny: trueだとエラー
  // error TS7010: 'hello', which lacks return-type annotation,
  //    implicitly has an 'any' return type.
  hello();
}
```

要注意なのは、レスポンスの型が一定しない関数です。 次の関数は、2019が指定された時だけ文字列を返します。 この場合、TypeScriptが気を利かせて `number | '今年'` という返り値の型を暗黙でつけてくれます。 しかしこの場合、単純な `number` ではないため、 `number` 型の変数に代入しようとするとエラーになります。

ただ、このように返り値の型がバラバラな関数を書くことは基本的にないでしょう。 バグを生み出しやすくなるため、返り値の型は特定の型1つに限定すべきです。 バリエーションがあるとしても、 `| null` をつけるぐらいにしておきます。

内部関数で明らかな場合は省略しても問題ありませんが、公開関数の場合はなるべく省略をやめた方が良いでしょう。

```
// 返り値の型がたくさんある、行儀の悪い関数
function yearLabel(year: number) {
  if (year === 2019) {
    return '今年';
  }
  return year;
}

const label: number = yearLabel(2018);
//  error TS2322: Type 'number | "今年"' is not assignable to type 'number'.
//    Type '"今年"' is not assignable to type 'number'.
```

関数を扱う変数の型定義

関数に型をつけることはできるようになりました。 次は、その関数を代入できる変数の型を定義して見ましょう。

例えば、文字列と数値を受け取り、booleanを返す関数を扱いたいとします。その関数は `check` という変数に入れます。 その場合は次のような宣言になります。 引数はアロー関数のままですが、返り値だけは `=>` の右につけ、 `{ }` は外します。 型定義ではなく、実際のアロー関数の定義の返り値は `=>` の左につきます。 ここが逆転する点に注意してください。

```
let check: (arg1: string, arg2: number) => boolean;
```

`arg2` がもし関数であったら、関数の引数の中に関数が出てくるということで、入れ子の宣言になります。 多少わかりにくいのですが、内側から順番に剥がして理解していくのがコツです。ネストが深くなり、理解が難しい場合はtype宣言で型定義を切り出して分解していく方が良いでしょう。

```
let check: (arg1: string, arg2: (arg3: string) => number) => boolean;
```

サンプルとしてカスタマイズ可能なソート関数を作りました。 通常のソートだと、すべてのソートを行うためになんども比較関数が呼ばれます。 大文字小文字区別なく、A-Z順でソートしたいとなると、その変換関数が大量に呼ばれます。 本来は1要素につき1回ソートすれば十分なはずです。それを実装したのが次のコードです。

まず、変換関数を通しながら、`[オリジナル，比較用に変換した文字列]`という配列を作ります。 その後、後半の変換済みの文字列を使ってソートを行います。 最後に、そのソートされた配列を使い、オリジナルの配列に含まれていた要素だけの配列を再び作成しています。

一度だけ変換するソート

```
function sort(a: string[], conv: (value: string) => string) {
  const entries = a.map((value) => [value, conv(value)])
  entries.sort((a, b) => {
    if (a[1] > b[1]) {
      return 1;
    } else if (a[1] < b[1]) {
      return -1;
    }
    return 0;
  });
  return entries.map(entry => entry[0]);
}

const a: string[] = ["a", "B", "D", "c"];
console.log(sort(a, s => s.toLowerCase()))
// ["a", "B", "c", "D"]
```

関数を扱う変数に、デフォルトで何もしない関数を設定する

コールバック関数を登録しておく変数に対し、何も代入されないときに呼び出し元が存在チェックをサボっていると、`undefined`に対して関数呼び出しをしたとエラーが発生します。その場合は、とりあえず何もしない関数を代入してエラーを回避したいと思うでしょう。

JavaScriptの世界では型がないため、とりあえず引数を持たず、本体が空の無名関数を入れてしまうと回避はできます。

```
// 何もしない無名関数を入れておく
var callback = function() {};
```

TypeScriptでは、例え引数を利用しなかったとしても、また実際に実行されないのでreturn文を省略した場合でも、変数の関数の型と合わせる必要があります。わかりやすさのために、変数宣言と代入を分けたコードを提示します。

```
// 変数宣言（代入はなし）
let callback: (name: string) => void;

// ダミー関数を設定
callback = (name: string): void => {};
```

もちろん、1行にまとめることもできます。JavaScript的にはどれも違いのない「関数」ですが、引数と返り値が違う関数はTypeScriptの世界では「別の型」として扱われますし、何もしない無名関数は引数も返り値もない関数の型を持っている、という判断が行われます。実際のロジックが空でも定義が必要な点は要注意です。

```
// 変数宣言（代入で推論で型を設定）
let callback = (name: string): void => {};
```

デフォルト引数

TypeScriptは、他の言語と同じように関数宣言のところに引数のデフォルト値を簡単に書くことができます。 また、TypeScriptは型定義通りに呼び出さないとエラーになるため、引数不足や引数が過剰になる、というエラーチェックも不要です。

```
// 新しいデフォルト引数
function f(name="小動物", favorite="小豆餅") {
  console.log(`${name}は${favorite}が好きです`);
}
f(); // 省略して呼べる
```

オブジェクトの分割代入を利用すると、デフォルト値つきの柔軟なパラメータも簡単に実現できます。 以前は、オプショナルな引数は`opts`という名前のオブジェクトを渡すこともよくありました。 今時であれば、完全省略時にはデフォルト値が設定され、部分的な設定も可能な引数が次のように書けます。

```
// 分割代入を使って配列やオブジェクトを変数に展開＆デフォルト値も設定
// 最後の={}がないとエラーになるので注意
function f({name="小動物", favorite="小豆餅"}={}) {
    :
}
```

JavaScriptは同じ動的言語のPythonとかよりもはるかにゆるく、引数不足でも呼び出すこともでき、その場合には変数に`undefined`が設定されました。 `undefined`の場合は省略されたとみなして、デフォルト値を設定するコードが書かれたりしました。 どの引数が省略可能で、省略したら引数を代入しなおしたり・・・とか面倒ですし、同じ型の引数があったら判別できなかったりもありますし、関数の先頭行付近が引数の処理で1画面分埋まる、ということも多くありました。 また、可変長引数があってもコールバック関数がある場合は必ず末尾にあるというスタイルが一般的でしたが、この後に説明する`Promise`を返す手法が一般的になったので、こちらも取扱いが簡単になりました。

```
// デフォルト引数の古いコード
function f(name, favorite) {
    if (favorite === undefined) {
        favorite = "小豆餅";
    }
}

// 古くてやっかいな、コールバック関数の扱い
function f(name, favorite, cb) {
    if (typeof favorite === "function") {
        cb = favorite;
        favorite = undefined;
    }
    :
}
```

関数を含むオブジェクトの定義方法

ES2015以降、関数や定義の方法が増えました。 JavaScriptではクラスを作るまでもない場合は、オブジェクトを作って関数をメンバーとして入れることがありますが、それが簡単にできるようになりました。 setter/getterの宣言も簡単に行えるようになりました。

関数を含むオブジェクトの定義方法

```
// 旧: オブジェクトの関数
var smallAnimal = {
   getName: function() {
     return "小動物";
   }
};
// 旧: setter/getter追加
Object.defineProperty(smallAnimal, "favorite", {
  get: function() {
    return this._favorite;
  },
  set: function(favorite) {
    this._favorite = favorite;
  }
});

// 新: オブジェクトの関数
//     functionを省略
//     setter/getterも簡単に
const smallAnimal = {
  getName() {
    return "小動物"
  },
  _favorite: "小笠原",
  get favorite() {
    return this._favorite;
  },
  set favorite(favorite) {
    this._favorite = favorite;
  }
};
```

クロージャと`this`とアロー関数

関数の中で関数を定義したときに、関数は自分の定義の外にある変数を参照できます。

```
function a() {
  const b = 10;
  function c() {
    console.log({b}); // bが表示される
  }
  c();
}
```

実行時の親子関係ではなく、ソースコードという定義時の親子関係を元にしてスコープが決定されます。これをレキシカルスコープと呼びます。また、このように自分が定義された場所の外の変数を束縛した関数を「クロージャ」と呼びます。TypeScriptでは関数を駆使してロジックを組み上げていきますので、この機能はとても重要です。

以前はJavaのようなオブジェクトを実装するために、関数内部の変数をプライベートメンバー変数のように扱うテクニックがかつてありました。クラスの機能が公式のサポートされたので、今では重要度は低くなっているし、そもそも隠す必要性もあまりなので使うことはありませんが、頭の体操にはなるので、興味がある方は調べてみてください。

レキシカルスコープは今では多くの言語が持っている機能なので、わざわざ名前を呼ぶこともありませんが、TypeScriptでは、知らないと落とし穴に落ちる可能性のあるやや重要な機能となります。前項でオブジェクトの中の関数定義を紹介しました。ここでは、予め定義された変数のように`this`を使っています。しかしこれは変数ではなく、特別な識別子です。レキシカルスコープで束縛できません。クロージャかつ、`this`への束縛ができる新文法としてアロー関数が追加されました。

アロー関数

JavaScriptでは、やっかいなのが`this`です。無名関数をコールバック関数に渡そうとすると、`this`がわからなくなってしまう問題があります。 アロー関数を使うと、その関数が定義された場所の`this`の保持までセットで行いますので、無名関数の`this`由来の問題をかなり軽減できます。 表記も短いため、コードの幅も短くなり、コールバックを多用するところで`function`という長いキーワードが頻出するのを減らすことができます。

アロー関数

```
// アロー関数ならその外のthisが維持される。
this.button.addEventListener("click", () => {
  this.smallAnimal.walkTo("タコ公園");
});
```

アロー関数にはいくつかの記法があります。 引数が1つの場合は引数のカッコを、式の結果をそのまま `return` する場合は式のカッコを省略できます。 ただし、引数の場所に型をつけたい場合は省略するとエラーになります。

アロー関数の表記方法のバリエーション

```
// 基本形
(arg1, arg2) => { /* 式 */ };

// 引数が1つの場合は引数のカッコを省略できる
// ただし型を書くとエラーになる
arg1 => { /* 式 */ };

// 引数が0の場合はカッコが必要
() => { /* 式 */ };

// 式の { } を省略すると、式の結果が return される
arg => arg * 2;

// { } をつける場合は、値を返すときは return を書かなければならない
arg => {
  return arg * 2;
};
```

以前は、 `this` がなくなってしまうため、 `bind()` を使って束縛したり、別の名前（ここでは `self` ）に退避する必要がありました。 そのため、 `var self = this;` と他の変数に退避するコードがバッドノウハウとして有名でした。

*this*消失を避ける古い書き方

```
// 旧: 無名関数のイベントハンドラではその関数が宣言されたところのthisにアクセスできない
var self=this;
this.button.addEventListener("click", function() {
  self.smallAnimal.walkTo("タコ公園");
});

// 旧: bind()で現在のthisに強制束縛
this.button.addEventListener("click", (function() {
  this.smallAnimal.walkTo("タコ公園");
}).bind(this));
```

`this` を操作するコードは書かない（1）

読者のみなさんはJavaScriptの `this` が何種類あるか説明できるでしょうか？ `apply()` や `call()` で実行時に外部から差し込み、何も設定しない（グローバル）、 `bind()` で固定、メソッドのピリオドの右辺が実行時に設定、といったバリエーションがあります。 これらの `this` の違いを知り、使

いこなせるのがかつてのJavaScript上級者でしたが、このようなコードはなるべく使わないように済ませたいものです。

無名関数で `this` がグローバル変数になってはずれてしまうのはアロー関数で解決できます。

`apply()` は、関数に引数セットを配列で引き渡したいときに使っていました。 配列展開の文法のスプレッド構文 `...` を使うと、もっと簡単にできます。

```
function f(a, b, c) {
    console.log(a, b, c);
}
const params = [1, 2, 3];

// 旧: a=1, b=2, c=3として実行される
f.apply(null, params);

// 新: スプレッド構文を使うと同じことが簡単に行える
f(...params);
```

`call()` は配列の `push()` メソッドのように、引数を可変長にしたいときに使っていました。 関数の中で引数全体は `arguments` という名前のちょっと配列っぽいオブジェクトで参照されます。 そのままではちょっと使いにくいので一旦本物の配列に代入したいという時、 `call()` を使って配列のメソッドを `arguments` に適用するハックがよく利用されていました。 これも引数リスト側に残余（Rest）構文を使うことで本体にロジックを書かずに実現できます。

```
// 旧: 可変長配列の古いコード
function f(a, b) {
    // この2は固定引数をスキップするためのもの
    var list = Array.prototype.slice.call(arguments, 2);
    console.log(a, b, list);
}
f(1, 2, 3, 4, 5, 6);
// 1, 2, [3, 4, 5, 6];

// 新: スプレッド構文。固定属性との共存もラクラク
const f = (a, b, ...c) => {
    console.log(a, b, c);
};
f(1, 2, 3, 4, 5, 6);
// 1, 2, [3, 4, 5, 6];
```

ただし、jQueryなどのライブラリでは、 `this` がカレントのオブジェクトを指すのではなく、選択されているカレントノードを表すという別解釈を行います。 使っているフレームワークが特定の流儀を期待している場合はそれに従う必要があります。

`bind()` の排除はクラスの中で紹介します。

即時実行関数はもう使わない

関数を作ってその場で実行することで、スコープ外に非公開にしたい変数などが見えないようにするテクニックがかつてありました。即時実行関数と呼びます。`function(){}`をかっこでくくって、その末尾に関数呼び出しのための`()`がさらに付いています。これで、エクスポートしたい特定の変数だけを`return`で返して公開をしていました。 今時であれば、公開したい要素に明示的に`export`をつけると、webpackなどのツールがそれ以外の変数をファイル単位のスコープで隠してくれます。

古いテクニックである即時実行関数

```
var lib = (function() {
  var libBody = {};
  var localVariable;

  libBody.method = function() {
    console.log(localVariable);
  }
  return libBody;
})();
```

まとめ

関数についてさまざまなことを紹介してきました。

- 関数の引数と返り値の型定義
- 関数を扱う変数の型定義
- デフォルト引数
- 関数を含むオブジェクトの定義方法
- クロージャと`this`とアロー関数
- thisを操作するコードは書かない（1）
- 即時実行関数はもう使わない

省略、デフォルト引数など、JavaScriptでは実現しにくかった機能も簡単に実装できるようになりました。 関数は、TypeScriptのビルディングブロックのうち、大きな割合をしめています。 近年では、関数型言語の設計を一部取り入れ、堅牢性の高いコードを書こうというムーブメントが起きています。 ここで紹介した型定義をしっかり行うと、その関数型スタイルのコードであっても正しく型情報のフィードバックされますので、ぜひ怖がらずに型情報をつけていってください。