JSのスプレッド構文を理解する

JavaScript TypeScript

JavaScript/TypeScriptのスプレッド構文(Spread Syntax)のまとめです。なおタイトルではJSといっていますが以下のコードは全てTypeScriptで書いて確認しています。基本構文は変わらないはずなのでご了承ください。

スプレッド構文(Spread Syntax)

`...foo` の形で記述され、配列やオブジェクトの要素を文字通り展開する構文。流石にそれだけだとピンとこないので配列とオブジェクトそれぞれの場合で例を見ていきます。

配列の場合

例

```
const foo = [1, 2];

// 配列のクローン
const bar = [...foo]; // => [1, 2]
```

```
// 要素を追加して新しい配列を生成
const baz = [...foo, 3, 4]; // => [1, 2, 3, 4]

// 配列をマージ
const hoge = [...foo, ...bar]; // => [1, 2, 1, 2]
```

Array.prototype.concat() 相当のことが簡潔にかけます。

配列のスプレッド構文はECMAScript2015で標準になっています。

オブジェクトの場合

例

```
const foo = { a: 1, b: 2 };

// オブジェクトのクローン
const bar = { ...foo }; // => { a: 1, b: 2 }

// プロパティを追加した新しいオブジェクトの生成
const baz = { ...foo, c: 3 }; // => { a: 1, b: 2, c: 3 }
```

```
// オブジェクトのマージ
const hoge = { ...foo, ...{ c: 3, d: 4 } }; // => { a: 1, b: 2

// 元のオブジェクトに同名プロパティがある場合は置き換わる
const fuga = { ...foo, b: 3 }; // => { a: 1, b: 3 }
const piyo = { ...foo, ...{ a: 3, b: 4 } }; // => { a: 3, b: 4
```

Object.assign() で書いてたようなものは置き換えできそうですね。

オブジェクトのスプレッド構文は、TypeScript >= 2.1で使えます。ECMAScriptにおいては~~ProposalがStage3なので将来的にはそっちにも取り込まれるんじゃないでしょうか。（Stage3からやっぱなしでってあるんですかね？標準化プロセスに詳しい人教えてください）~~ (2018-01-25 追記： Stage4になりました)

注意したい点

配列にしろオブジェクトにしろスプレッド構文はshallow copyなので、ネストしている場合は注意が必要です。

```
const foo = {
  a: {
    b: 1
  }
};

const bar = { ...foo };
foo.a.b = 2;
console.log(bar); // => { a: { b: 2 } }
```

子のオブジェクトは同じ参照なことがわかります。
よってネストしているオブジェクトを扱う場合次のように個別に分割する必要があります。

```
const foo = {
  a: {
    b: 1,
    c: 2
  },
  d: 3
};

const bar = {
  ...foo,
  a: {
    ...foo.a,
```

```
    c: 4
  }
};
console.log(bar); // => { a: {b: 1, c: 4}, d: 3 }
```

分割代入(Destructuring assignment)

分割代入は配列やオブジェクトの要素を取り出して個別の変数に代入するのを簡単に行えるもの。

```
const array = [1, 2, 3];
const [x, y, z] = array;
console.log(x); // => 1
console.log(y); // => 2
console.log(z); // => 3

const obj = { foo: 1, bar: 2, baz: 3 };
const { foo } = obj;
console.log(foo); // => 1
```

この構文とスプレッドの組み合わせで、要素を取り出しつつ残りを変数に代入することが可能です。

```
const array = [1, 2, 3];
const [p, ...q] = array;
console.log(p); // => 1
console.log(q); // => [2, 3]

const obj = { foo: 1, bar: 2, baz: 3 };
const { foo, ...rest } = obj;
console.log(foo); // => 1
console.log(rest); // => { bar: 2, baz: 3 }
```

ちなみにネストしたオブジェクトだとこうなります。

```
const nestedObj = {
  x: { a: 1, b: 2, c: 3 },
  y: [4, 5, 6]
};

const { x: { a, ...restX }, y: [y0, ...restY] } = nestedObj;
console.log(a); // => 1
console.log(restX); // => { b: 2, c: 3 }
console.log(y0); // => 4
console.log(restY); // => [5, 6]
```

余談: Redux Reducerでの使用例

スプレッド構文はオブジェクトをイミュータブルに扱いたい場面で活躍します。そのような場面のひとつとしてReduxのState/Reducerがあるので簡単な例を載せます。

```
interface Todo {
  id: number;
  text: string;
  completed: boolean;
}

interface TodoState {
  todos: { [id: number]: Todo };
}

const todoReducer = (state: TodoState = { todos: {} }, action:
  switch (action.type) {
    case TypeKeys.ADD_TODO:
      const nextId = Object.keys(state.todos).length + 1;
      return {
        todos: {
          ...state.todos,
          [nextId]: {
            id: nextId,
            text: action.payload.text,
            completed: false
          }
        }
```

```
        };
      case TypeKeys.COMPLETE_TODO:
        return {
          todos: {
            ...state.todos,
            [action.payload.id]: {
              ...state.todos[action.payload.id],
              completed: true
            }
          }
        };
      default:
        return state;
    }
};
```