はじめに

JavaScript: Everything From ES2016 to ES2019のを読み、本稿はES2018で導入された機能についてわかったことを自分なりにまとめたものになります。

レスト構文とスプレッド構文(Rest/Spread)をオブジェクトにも使用可能

ES2015では、配列にスプレッド構文を使用することができました。

```
const name = ["taro", "ichiro", "hanako"];
const spread_name = [...name, "jiro"];

console.log(spread_name); //["taro", "ichiro", "hanako", "jiro"]
```

ES2018では、オブジェクトにもスプレッド演算子を使用できるようになりました。

```
let sampleObj = {
    a: "1",
    b: "2",
    c: "3",
    d: "4",
}

let {
    a,
    b,
    ...z
} = sampleObj;
console.log(a); //1
console.log(b); //2
console.log(z); //{c: "3", d: "4"}
```

スプレッド演算子によって複製されたオブジェクトについて

スプレッド演算子によって複製されたオブジェクトについて、
元のオブジェクトが変更された場合、複製されたオブジェクトには変更がされません。

```
let sampleObj = {
    a: "1",
    b: "2",
    c: "3",
    d: "4",
}
let copy = {
    ...sampleObj
};
console.log(copy); //{a: "1", b: "2", c: "3", d: "4"}

sampleObj.e = 5;
console.log(sampleObj); //{a: "1", b: "2", c: "3", d: "4"}

//複製されたオブジェクトには変更されない
console.log(copy); //{a: "1", b: "2", c: "3", d: "4"}
```

非同期イテレータ

`for-await-of` 構文を使用して、非同期に反復可能なオブジェクトを繰り返すループを作ることができます。

`for-of` 構文では、反復可能なオブジェクト(`[Symbol.iterator]()`)が利用できました。
`for-await-of` 構文では、非同期反復可能なオブジェクト(`[Sysmbol.iterator]()`)が利用できます。
以下のサンプルコードは、 `async function*` で宣言した非同期ジェネレータを繰り返して処理するコードです。

```
let sleep = (ms) => new Promise((func) => setTimeout(func, ms));

async function* asyncGenerator() {
    yield 1;
```

```
    await sleep(1000);
    yield 2;
    await sleep(1000);
    yield 3;
    await sleep(1000);
}

(async function() {
    for await (item of asyncGenerator()) {
        console.log(item);
    }
})();
```

Promise.prototype.finally()

MDNによると、次のように示されています。

> finally() メソッドは、Promise を返します。成功・失敗にかかわらず、promise が確立したら指定したコールバック関数が実行されます。これにより、promise が成功裏に実行されたか否かに関わりなく、Promise が処理された後に実行されなければならないコードを提供できます。

```
let myPromise = new Promise((resolve, reject) => {
    resolve();
})

myPromise
    .then(() => {
        console.log('still working');
    })
    .catch(() => {
        console.log('there was an error');
    })
    .finally(() => {
        console.log('Done!');
    })
// still working
// Done!
```

正規表現に関する新しい機能

正規表現に関連する新しい機能について、ここでは以下3つを紹介します。

- `s` フラグ (s(dotAll) flag for regular)
- 名前付きキャプチャグループ (RegExp named capture groups)
- 後読みアサーション(RegExp Lookbehind Assertions)

s(dotAll) flag for regular

正規表現に `s` オプションが導入されました。
https://github.com/tc39/proposal-regexp-dotall-flag

`/./s` 、 `new RegExp('.', 's')` のように使用して、
改行コードにマッチするための正規表現を記述できます。

```
//sオプションを使用しない例
/foo[^]bar/.test('foo\nbar');
//true

//sオプションを使用する例
/foo.bar.test/s.test('foo\nbar\ntest');
//true
```

名前付きキャプチャグループについて

名前付きキャプチャグループとは、 `(?<name>...)` で表現されたキャプチャグループのことをいいます。
例文どおりですが、名前付きキャプチャグループを使用すれば、
`exec` メソッドで返されたオブジェクトの `groups.name` プロパティを参照することで、
正規表現にマッチした値を操作することができます。

```
//名前付きなしキャプチャグループ
let re = /(\d{4})-(\d{2})-(\d{2})/
let result = re.exec('2020-01-01')
console.log(result)
//["2020-01-01", "2020", "01", "01", index: 0, input: "2020-01-01", groups: undefined]
//index: 0
//input: "2020-01-01"
//groups: undefined

//名前付きキャプチャグループ
let re = /(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/u;
let result = re.exec('2020-01-01');
console.log(result)
//["2020-01-01", "2020", "01", "01", index: 0, input: "2020-01-01", groups: {…}]
//index: 0
//input: "2020-01-01"
//groups:
// year: "2020"
// month: "01"
// day: "01"
```

後読みアサーション

肯定後読みアサーションは（?<=...）で表現されます。
直前に ... がある場合にマッチします。

否定後読みアサーションは（?<!...）で表現され、
直前に ... がない場合にマッチします。

```
let alphabet = 'abcdef';
let result = alphabet.match(/(?<=abc)def/);
console.log(result)
//["def", index: 3, input: "abcdef", groups: undefined]

let result = alphabet.match(/(?<!abc)def/);
console.log(result)
//null
let result = alphabet.match(/(?<!xyz)def/);
console.log(result)
//["def", index: 3, input: "abcdef", groups: undefined]
```