付録5　ES2016とES2017の動向 英語識別子版

武舎 広幸

ECMAScript 2015（略してES2015あるいはES6）で非常に大きな機能追加がありましたが、今後は毎年少しずつ機能が追加され、ECMAScript 2016、ECMAScript 2017、...のようにバージョンが付けられることが決まっています。

この付録では、既に仕様が固まったECMAScript 2016と、現在議論が行われているECMAScript 2017で追加されそうなものを紹介します。

ECMAScript 2016

ECMAScript 2016（ES2016）では以下の2つの機能が追加されました。

- 累乗を表す演算子（オペレータ）「**」
- Array.prototype.includes

累乗を表す演算子「**」

まずは累乗を表す演算子「**」（アスタリスクの連続）です。ほかの多くの言語で使われていますので、多くのプログラマーにとってはおなじみでしょう。
Math.pow（「第16章「Math」」参照）で同じことができますが、演算子として使えるようになりました。

example/chae/exae-01-1/main.js

```
/* 2016年12月現在、ChromeやEdgeで動作 */
const a = 3**4;
console.log(a); // 81
const b = Math.pow(3,4);
console.log(a === b); // true
```

2016年12月現在、ChromeやEdgeで動作するようになっています。

Array.prototype.includes

Array.prototype.includesは配列の中に指定の要素が含まれているかを検査してくれます。indexOfを使って戻り値が0以上かどうかを判定すれば類似の検査は可能ですが、NaNの処理などに少し違いがあります。

example/chae/exae-01-2/main.js

```
const array2 = [1, 4, [9, 16], 25, 36, NaN];
console.log(array2.includes(1)); // true
console.log(array2.indexOf(1)); // 0
console.log(array2.includes(0)); // false
console.log(array2.indexOf(0)); // -1
```

```
console.log(array2.includes(16)); // false
console.log(array2.indexOf(16)); // -1
console.log(array2.includes(NaN)); // true
console.log(array2.indexOf(NaN)); // -1   (indexOfではNaNは見つからない)
console.log(["山", "川", "やま", "かわ"].includes("山")); // true
console.log(["山", "川", "やま", "かわ"].includes("かわ")); // true
console.log(["山", "川", "やま", "かわ"].includes("革")); // false
```

既にNodeを含めほとんどのJavaScriptエンジンで動作します。

ECMAScript 2017

ECMAScript 2017（ES2017）では次のような機能が追加の検討対象（ステージ4という最終検討段階）になっています。

- 関数構文の末尾カンマの許容
- 文字列用のメソッドpadStartとpadEnd——文字列を希望する幅で揃えて表示
- Object.entriesおよびObject.values——オブジェクトを配列に簡単に変換
- Object.getOwnPropertyDescriptors——プロパティのディスクリプタを一度に取得
- async/await——非同期処理をわかりやすく記述

それぞれの機能について説明していきます。2016年12月現在の段階では、機能によって利用できる環境が異なりますので、実行可能な環境についても説明を加えます。最新の情報は「第1章「JavaScriptの歴史と開発環境」」でも紹介したkangaxが管理しているhttp://kangax.github.io/compat-table/es2016plus/などを参照してください。

関数構文の末尾カンマの許容

「第3章「変数、定数、リテラル、データ型」」で見たように、配列やオブジェクトの最終要素の後ろに（余分な）「,」を書いても構文エラーにはなりません。ES2017では、関数定義や呼び出しの引数リストの最後に（余分な）「,」を書くことも許されるようになります。

example/chae/exae-02-1/main.js

```
/* 2016年12月現在、Safari（最新版）で動作 */
function func1(arg1, arg2,) {
  return arg1 * arg2;
}
console.log(func1(3, 5,));

const result =
        (function( arg1, arg2,) {
          console.log(`足し算の結果=${arg1+arg2}`, );
          return arg1 * arg2;
        }) ( 3, 5, );
console.log(`掛け算の結果=${result}`,);

/* 実行結果
15
足し算の結果=8
掛け算の結果=15
*/
```

この機能が採用された大きな理由は、配列やオブジェクトの場合と同じで、データ

の入れ替えや追加が楽になることですが、もうひとつの理由のひとつとしてバージョン管理システムとの「相性」があるそうです。

たとえば、次のような関数定義（部分）があったとします。

```
function 関数1(
  引数1 // 引数1の説明
)
```

これに次のように引数を加えたとします。

```
function 関数1(
  引数1, // 引数1の説明
  引数2  // 引数2の説明
)
```

すると、2行の差異ができてしまいます。

これに対して、最後の引数の終わりに「,」を置いておくと「引数2,」の行ひとつだけしか違わないことになります。

```
function 関数1(
  引数1, // 引数1の説明
 )

   ↓

function 関数1(
  引数1, // 引数1の説明
  引数2, // 引数2の説明
)
```

文字列用のメソッドpadStartとpadEnd

メソッドpadstartやpadEndを使うと文字列を揃えたり、前後に指定した文字を指定した数だけ埋めたりすることができます。

たとえばpicturesという名前のフォルダに画像がたくさん入っています。画像には順番にpict000.jpg、pict001.jpg、...と名前が付けられています。この画像を順番に処理しようと思っています。ここでは、たとえば<img>タグを使って画像を順番に出力するとしてみましょう。

このとき「桁上がり」の処理が面倒です。pict009.jpgのあとはpict010.jpgとなるので、出力する0をひとつ減らさなければなりません。また、pict099.jpgのあとはpict100.jpgとなります。

文字列に対して使えるメソッドpadStartを使うと次のようにできます。

example/chae/exae-02-2/main.js

```
/* 2016年12月現在、FirefoxやSafariが対応 */
for (let i=0; i<=300; i++) {
  const numPart = i.toString().padStart(3, 0);
  const filename = `pictures/pict${numPart}.jpg`;
  console.log(`<img src="${filename}" style="width: 4cm;">`);
}
```

```
/* 実行結果
<img src="pictures/pict000.jpg" style="width: 4cm;">
<img src="pictures/pict001.jpg" style="width: 4cm;">
<img src="pictures/pict002.jpg" style="width: 4cm;">
...
<img src="pictures/pict009.jpg" style="width: 4cm;">
<img src="pictures/pict010.jpg" style="width: 4cm;">
<img src="pictures/pict011.jpg" style="width: 4cm;">
<img src="pictures/pict012.jpg" style="width: 4cm;">
...
*/
```

文字列sに対してs.padStart(maxLength, fillString=' ')の形で使われます。sがmaxLengthに満たない場合、左側（Start側）にfillStringで指定された文字（列）を繰り返し埋めてくれます。

もうひとつpadEndはs.padEnd(maxLength, fillString=' ')の形で使われ、sがmaxLengthに満たない場合、右側（End側）にfillStringで指定された文字（列）を繰り返し埋めてくれます。

どちらもfillStringを指定しないと（半角）スペースになります。

もうひとつ例を見てみましょう。

example/chae/exae-02-3/main.js

```
console.log("".padStart(30, "1234567890"));
const l = 0.34e-2.toString();
const m = "3.14";
const n = "3";
console.log(l.padStart(10) + m.padStart(10) + n.padStart(10));
console.log(l.padStart(10, "_") + m.padStart(10, "_") + n.padStart(10, "_"));
console.log(l.padEnd(10) + m.padEnd(10) + n.padEnd(10));

console.log("".padStart(20, "一二三四五六七八九〇"));
let table = "".padStart(20, "＝＊＊＝") + "\n";
table += "| "+"右寄せ".padStart(8,"  ")+" | "+"左寄せ".padEnd(8,"  ")+" |\n";
table += "| "+"左寄せ".padEnd(8,"  ")+" | "+"右寄せ".padStart(8,"  ")+" |\n";
table += "".padStart(20,"＝＊＊＝");
console.log(table);

/* 実行結果
12345678901234567890123456789O
    0.0034      3.14         3
____0.0034______3.14_________3
0.0034    3.14      3
一二三四五六七八九〇一二三四五六七八九〇
＝＊＊＝＝＊＊＝＝＊＊＝＝＊＊＝
|          右寄せ | 左寄せ          |
| 左寄せ          |          右寄せ |
＝＊＊＝＝＊＊＝＝＊＊＝＝＊＊＝
*/
```

Object.entriesおよびObject.values

Object.entriesおよびObject.valuesでオブジェクトのプロパティや値を一度に取得することができます。ただし、シンボルのプロパティは取得されません。

例を見てみましょう。Object.entriesは配列を要素にもつ配列を返します。内側の配列はオブジェクトの「プロパティ」と「値」の2つの要素をもっています。

example/chae/exae-02-4/main.js list1

```
/* 2016年12月現在、FirefoxやChromeが対応 */
const SPECIAL = Symbol();
let betsEach = {
  王冠: 0,
  錨: 5,
  ハート: 2,
  スペード: 0,
  クラブ: 3,
  ダイヤ: 0,
  [SPECIAL]: 13,
};

const array1 = Object.entries(betsEach);
console.log(array1);
/* 実行結果 (node)
[ [ '王冠', 0 ],
  [ '錨', 5 ],
  [ 'ハート', 2 ],
  [ 'スペード', 0 ],
  [ 'クラブ', 3 ],
  [ 'ダイヤ', 0 ] ]
*/

for (let i=0; i<array1.length; i++) {
  console.log(array1[i]);
}
/* 実行結果
[ '王冠', 0 ]
[ '錨', 5 ]
[ 'ハート', 2 ]
[ 'スペード', 0 ]
[ 'クラブ', 3 ]
[ 'ダイヤ', 0 ]
*/
```

for...ofループ（「第4章「制御フロー」」参照）と分割代入（「第5章「式と演算子」」参照）を使うと次の例のように、すべてのプロパティのキーとその値を簡単に取得することができます。

example/chae/exae-02-4/main.js list2

```
console.log("    マーク | 賭け金");
console.log("----------------------");
for (let [mark, bet] of Object.entries(betsEach)) {
  console.log(mark.padStart(5, "  ") + " | " + bet);
}
console.log(`スペシャル | ${betsEach[SPECIAL]}`);

/* 実行結果
    マーク | 賭け金
----------------------
      王冠 | 0
        錨 | 5
    ハート | 2
  スペード | 0
    クラブ | 3
    ダイヤ | 0
スペシャル | 13
*/
```

Object.valuesは値だけからなる配列を返します。

example/chae/exae-02-4/main.js list3

```
console.log(Object.values(betsEach)); // [ 0, 5, 2, 0, 3, 0 ]
```

Object.getOwnPropertyDescriptors

Object.getOwnPropertyDescriptorsは「第21章「オブジェクトのプロパティに関する詳細」」で説明したプロパティのディスクリプタに関連するものです。その章で説明したObject.getOwnPropertyDescriptorは、オブジェクトとプロパティを指定してそのディスクリプタを表示するものでした（Descriptorと単数形になっている点に注意）。

Object.getOwnPropertyDescriptors(obj)はオブジェクトobjのすべての独自のプロパティのディスクリプタを一度に返します。次の例を見てください。

example/chae/exae-02-5/main.js

```
/* 2016年11月現在、Chromeが対応 */
const SPECIAL = Symbol();
let betsEach = {
  王冠: 0,
  錨: 5,
  [SPECIAL]: 13,
};

let x = Object.getOwnPropertyDescriptors(betsEach);

console.log(x.王冠);
/* 実行結果 node */
// { value: 0, writable: true, enumerable: true, configurable: true }
console.log(x.錨);
// { value: 5, writable: true, enumerable: true, configurable: true }
console.log(x[SPECIAL]);
/* { value: 13,
  writable: true,
  enumerable: true,
  configurable: true }
 */

for (let [property, descriptor] of Object.entries(x)) {
  console.log(property + " : ");
  for (let [p, v] of Object.entries(descriptor)) {
    console.log("  " + p + " : " + v);
  }
}
/* 実行結果　（for文）
王冠：
  value：0
  writable：true
  enumerable：true
  configurable：true
錨：
  value：5
  writable：true
  enumerable：true
  configurable：true
*/
```

async/await

async関数と演算子awaitを使うと、プロミス（「第14章「非同期プログラミング」」参照）を使ったコードがすっきりと書けるようになります。

async関数の実行

2016年12月現在、async関数はNodeや一般のブラウザのリリースバージョンではそのままの形で試すことができないようですが、Traceurを使えばそのままの形で試すことができます。

Traceurを使うにはまずnpmを使ってインストールが必要です（Windowsの場合は「sudo」は取ってください）。

```
$ sudo npm install -g traceur
```

async関数を使ったJavaScriptファイル（xxx.js）を実行するには、次のようにオプション--async-functions trueを付けて実行してください。

```
$ traceur --async-functions true xxx.js
```

普通の関数のキーワードfunctionの前にasyncを付けるとasync関数になります。演算子awaitはasync関数の中でのみ使えるもので、対象のPromiseが成功するまで待ってくれます。

次の例は「第14章「非同期プログラミング」」で見た例と同じ処理をするもので、3つのファイル（a.txt、b.txt、c.txt）を読み込んで、それを別のファイル（d.txt）に書き込むものです。

example/chae/exae-02-6/main.js

```
/*  2018年4月6日現在 node （v8.9.4）も対応したようです
Node 未対応の場合 →  traceur --async-functions true main.js
*/
const fs = require('fs');

function readFile(filename) {
  return new Promise( (funcSuccess, funcFail) => {
    fs.readFile(filename, "utf-8", (err, dataReadIn) => {
      err ? funcFail(err) : funcSuccess(dataReadIn);
    }); });
}

function writeToFile(filename, dataWriting) {
  return new Promise( (funcSuccess, funcFail) => {
    fs.writeFile(filename, dataWriting, err => {
      err ? funcFail(err) : funcSuccess('OK');
    }); });
}

async function fileReadAndWrite() {
  try {
    let dataWriting = await readFile("a.txt");
    dataWriting += await readFile("b.txt");
    dataWriting += await readFile("c.txt");
    await writeToFile("d.txt", dataWriting);
  } catch (err) {
    console.error("エラーが起こりました:" + err);
  }
}
```

```
fileReadAndWrite();
```

関数readFileとwriteFileの定義は本質的には「第14章「非同期プログラミング」」のものと同じですが、async関数「fileReadAndWrite」でawaitが使われています。コールバックを経由して（メソッドresolveから）値を受け取る代わりに、単純にawaitから返される値を使うことができますし、try...catchを使った普通の例外処理でエラーをキャッチすることもできます。

「第14章「非同期プログラミング」」のthenメソッドを使ったもののうち、「fileReadAndWrite」に対応する部分を再掲しますので比較してみてください。

example/chae/exae-02-6-2/main.js list1

```
let dataWriting = "";
readFile("a.txt")
.then(function(dataFromFile) {
  dataWriting += dataFromFile;
  return readFile("b.txt");})
.then(function(dataFromFile) {
  dataWriting += dataFromFile;
  return readFile("c.txt");})
.then(function(dataFromFile) {
  dataWriting += dataFromFile;
  return writeToFile("d.txt", dataWriting);})
.then(function(message) {
  console.log("ファイルの合体に成功しました。");})
.catch(err => {
  console.error("エラーが起こりました:" + err);});
```

次は「第14章「非同期プログラミング」」でジェネレータ（とジェネレータランナー）を使った例です。

async/awaitのコードとよく似ていますが、こちらはジェネレータランナーを用意する必要がありますので少し複雑になります。

example/chae/exae-02-6-3/main.js list1

```
function* fileReadAndWrite() {
  try {
    let dataWriting = yield readFile('a.txt');
    dataWriting += yield readFile('b.txt');
    dataWriting += yield readFile('c.txt');
    yield writeToFile('d.txt', dataWriting);
  } catch (err) {
    console.error("エラーが起こりました:" + err);
  }
}

generatorRunner(fileReadAndWrite);
```

ちなみに、async/awaitでPromise.allも同じように使うことができます。

example/chae/exae-02-7/main.js list1

```
async function fileReadAndWrite() {
  try {
    const results
            = await Promise.all([readFile("a.txt"),
                                 readFile("b.txt"),
                                 readFile("c.txt")]);
    await writeToFile("d.txt", results[0]+results[1]+results[2]);
    console.log("ファイルの合体に成功しました。");
```

```
  } catch (err) {
    console.error("エラーが起こりました:" + err);
  }
}

fileReadAndWrite();
```

上の例ではPromise.allをawaitで待っています。それぞれの値が配列「**結果の配列**」の要素として記憶され、それが合体されてd.txtに書き込まれています。

最後に「第14章「非同期プログラミング」」のロケット打ち上げの例もasync/awaitで書いてみましょう。

example/chae/exae-02-8/main.js

```
/* Node 未対応の場合 →  traceur --async-functions true main.js */
'use strict';
function countDown(second) {
  return new Promise(function(funcSuccess, funcFail) {
    const timeOutIDs = [];
    for (let i=second; i>=0; i--) {
      const timeOutID = setTimeout( () => {
        if(i===13) {
          timeOutIDs.forEach(clearTimeout); /* すべてクリア */
    return funcFail(new Error(`${i}という数は不吉過ぎます`));
  }
        i>0 ? console.log(i + '...') : funcSuccess(console.log("GO!"));},
           (second-i)*1000)
      timeOutIDs.push(timeOutID);
    }
  });
}

function launchRocket() {
  return new Promise(function(funcSuccess, funcFail) {
    if(Math.random() < 0.5) return funcFail(new Error("打ち上げ失敗"));
    console.log("発射!");
    setTimeout(function() {
      funcSuccess("周回軌道に乗った！");
    }, 2*1000); /* 超速のロケット */
  });
}

async function countDownAndLaunchRocket(secondsCountingDown) {
  try {
    await countDown(secondsCountingDown);
    console.log(await launchRocket());
  } catch (err) {
    console.error("管制塔、管制塔。トラブル発生... " + err.message);
  }
}

countDownAndLaunchRocket(15);
```

以上、ECMAScript 2016で追加が決まったものと、ECMAScript 2017で追加されそうなものについて紹介しました。なお、この付録については、おもに Axel Rauschmayer著『Exploring ES2016 and ES2017』（https://leanpub.com/exploring-es2016-es2017/）を参考にしました。