

@IT eBookシリーズ Vol.23

JavaScriptアプリ開発入門

かわさきしんじ[著]



JavaScriptアプリ開発入門

最近の Web 開発では、どのような技術を使い、どのようにアプリが開発されているのだろう。モダン JS ライブラリをよく知らない .NET 開発者に向けて、その世界をまとめて紹介する。また C# と JavaScript におけるクラス定義を比較し、TypeScript や ECMAScript 2015 で JavaScript プログラミングがどう変わるかを見ていく。

● 今から始める JavaScript アプリ開発

[.NET 開発者のための JavaScript ライブラリカタログ（MVC フレームワーク編）](#)

[.NET 開発者のための JavaScript ライブラリカタログ（デスクトップアプリ編）](#)

[.NET 開発者のための JS ライブラリカタログ（gulp 編）](#)

[.NET 開発者のための JS ライブラリカタログ（Grunt 編）](#)

● C#×JavaScript

[C# 開発者のための最新 JavaScript 事情（クラス定義）](#)

[C# 開発者のための最新 JavaScript 事情（関数編）](#)

[C# 開発者のための最新 JavaScript 事情（Babel 編）](#)

[C# 開発者のための最新 JavaScript 事情（モジュール編）](#)

[C# 開発者のための最新 JavaScript 事情（Promise 編）](#)

[C# 開発者のための最新 JavaScript 事情（Promise 編パート2）](#)

[C# 開発者のための最新 JavaScript 事情（async 関数編）](#)

[C# 開発者のための最新 JavaScript 事情（ジェネレータ関数編）](#)

[ECMAScript の最新情報を得るには](#)

初出：@IT .NET Insider フォーラム

特集：今から始める JavaScript アプリ開発

特集：C# × JavaScript

【注意】

・eBook 化に際して、上記の記事を一部加筆修正しています。

・本書に記載されている社名・商品名は一般に各社の商標または登録商標です。

・その他、免責事項については@IT Web サイトのポリシーに準拠します。

<http://www.atmarkit.co.jp/aboutus/copyright/copyright.html>

●今から始める JavaScript アプリ開発

.NET 開発者のための JavaScript ライブラリカタログ (MVC フレームワーク編)

Insider.NET 編集部 かわさきしんじ (2015 年 11 月 13 日)

最近の Web 開発では、どんな技術を使い、どのようにアプリが開発されているのだろう。モダン JS ライブラリをよく知らない .NET 開発者に向けて、その世界をまとめて紹介する。

ここでは、最新の Web 技術にはあまり取り組めていない .NET 開発者を対象として、Web 標準技術を使ってアプリ開発を行う際によく使われているライブラリやフレームワークなどを紹介していく。モダン Web の世界では、現在、どんなことが行われているかをのぞいてみよう（すでにこれらの技術を活用している方にとっては「何をいまさら」な話ばかりだろうがご容赦願いたい）。本章ではクライアントサイド MVC フレームワークを紹介する。

クライアントサイド MVC フレームワーク

ここ数年、JavaScript の世界ではクライアントサイドで MVC な Web アプリを構築するためのフレームワークが多く登場している。クライアントサイド MVC な Web アプリでは、サーバーは基本的に Web API として機能し、重量級の HTML フラグメントをクライアントに送信したりするのではなく、クライアント側で必要になるデータのみを送信したり、データの永続化を行うのがその主要な機能となる。

クライアントサイドでは[シングルページアプリ \(Single Page Application : SPA\)](#) などの形でアプリを構築し ***1**、更新が必要になったら、必要なデータだけをサーバーから受け取るようにすることで、通信量や更新にかかる時間を低減できるようになってきた。

***1** シングルページアプリ (SPA) とは、単一ページで構成される Web アプリのこと。最初に Web アプリに必要な基本要素が単一のページに全て読み込まれ、その後はアプリの実行に必要なデータが必要に応じてサーバーとの間で動的に送受信される。単一ページの Web アプリではあるが、論理的なページナビゲーションなどの機能も持ち、あたかも複数ページで構成されている Web ページのように動作するものもある。従来の Web アプリよりも高い応答性を実現可能なことから注目が高まっている。最近では、JavaScript などの Web 標準技術で SPA を実装するためのフレームワークが多数登場している（以下で紹介しているフレームワークでも SPA を実装可能だ）。

MVC（あるいは MV*^{*}：MVC、MVVM など、「MV」で始まるパターンの総称）を設計思想に持つ JavaScript フレームワークは、JavaScript を利用した中規模から大規模な Web アプリ開発に秩序をもたらし、きちんとしたアプリ設計が可能になることから、多数のクライアントサイド MVC フレームワーク（以下、MVC フレームワー

クとする) が登場したのだ。

そうした MVC フレームワークの中で、このところ、人気が一番あるのが「[AngularJS](#)」である。この他にも、以前から広く使われている「[Backbone.js](#)」、マイクロソフト謹製の「[Knockout.js](#)」、データバインディングや UI コンポーネントに特化して機能を絞り込んだシンプルな「[Vue.js](#)」など、多くの MVC フレームワークが存在している。

では、これらのうちのいくつかを簡単に紹介しておこう。

AngularJS

[AngularJS](#) はGoogleとコミュニティによって開発がなされているオープンソースなクライアントサイドの MVW フレームワークだ(「MVW」の「W」は「Whatever」=「何でも」を意味する)。次バージョンである「[Angular 2](#)」(後述) もリリースされたが、AngularJS は現在でも広く使われている。

[MongoDB](#) (ドキュメント指向データベース) / [Express](#) (Node.js 上で動作するサーバーサイド MVC フレームワーク) / [AngularJS](#) (クライアントサイド MVC フレームワーク) / [Node.js](#) (サーバーサイドでの Web アプリ実行環境) の頭文字を取った「[MEAN スタック](#)」の構成要素でもある。

AngularJS の特徴としては次のようなことが挙げられる。

- テンプレート
- 双方向データバインディング

AngularJS では、HTML ページは「テンプレート」として扱われる。また、DOM 要素を操作したり、モデル / コントローラーから得たデータをビューに埋め込んだりするために使われるさまざまな「ディレクティブ」が用意されている(開発者が独自のディレクティブを作成することも可能)。そして、テンプレートを AngularJS がコンパイルすることで、ユーザーが実際に目にするビューが作成されるようになっている。

また、.NET 開発者の方であれば、おなじみの単方向 / 双方向データバインディングもサポートされている。これにより、ビューとモデルの間でのデータの同期がとてもシンプルに記述できる。

以下に簡単な例を示す(抜粋)。

```
<div ng-controller="FooController as fooctrl">
  <input type="text" ng-model="fooctrl.test" />
  {{ fooctrl.test }}
  ..... 省略 .....
</div>
```

```
var app = angular.module('myapp', []);

app.controller('FooController', function() {
  this.test = "foobar";
});
```

テンプレート／ディレクティブ／データバインディング

上は AngularJS が解釈する HTML テンプレートであり、強調書体で示しているのが AngularJS のディレクティブだ。下はこのテンプレートにアタッチされるコントローラーの JavaScript コードである。

「ng-controller」ディレクティブで指定された「FooController」コントローラーは、この <div> タグで囲まれた部分にアタッチされる（「as fooctrl」となっているので、この範囲内では「fooctrl」でこのコントローラーを参照できる）。また、このコントローラーは test プロパティを持ち、その初期値は「foobar」となっている。そして、「ng-model」ディレクティブはこれが双方向データバインディングを行うことを意味しており、その対象は「fooctrl.test」だ（「fooctrl」は「FooController」の別名）。そのため、このコードを実行すると、テキストボックスには「foobar」が表示され、その内容を変更すると test プロパティの値が自動的に変更される（双方向データバインディング）。

また「{{ }}」もディレクティブ（補間ディレクティブ）であり、HTML に二重波かっこで囲まれている式の評価結果が埋め込まれる（単方向データバインディング）。上の例では「fooctrl.test」の値が変更されると、その値がリアルタイムで HTML ページに反映される。



上記コードの実行画面

テンプレートのコンパイル過程では、AngularJS は静的な HTML ファイルではなく、DOM を走査していき、AngularJS のディレクティブがあれば、それに見合った処理を行っていき、最終的な DOM を生成する。これに

より、テンプレート内で特定のコントローラーがアタッチされている箇所では、そのコントローラーが所有するスコープと DOM との動的なバインディングが形成され、データの更新が即座に DOM に反映されるアプリを開発できるようになっている。

これらの機構により、AngularJS では非常にシンプルなコードで Web アプリを記述できる。ただし、AngularJS はその哲学として「UI のデザインからロジックの記述、テストに至るまで、アプリ構築の全ての道筋をフレームワークが導くことができたらそれはとても素晴らしいこと」と考えている（「[What Is Angular?](#)」の「The Zen of Angular」を参照）。AngularJS が全ての面倒を見てくれるが、覚えなければならないこともたくさんあり（その分、覚えたら楽できるということでもあるが）、AngularJS にロックインされるということでもある。

また、AngularJS の主たる用途は CRUD 処理を伴うアプリであり、ゲームなど、複雑な DOM 操作が必要になるアプリはどちらかというと苦手だ。

Angular 2

注意

Angular 2 に関する内容は 2015 年 11 月時点のものです。電子書籍化に当たり、動作を確認したところ、Angular 2 の α -51 版までは動作しましたが、 β 版、RC 版、製品版ではこのままのコードでは動作しません。ステップバイステップのチュートリアルについては Angular 2 の公式サイト内の「[QUICKSTART TYPESCRIPT](#)」ページなどをご覧ください。

[Angular 2](#) は上で見た AngularJS の次バージョンだ。2016 年 9 月には RC が外れて正式版がリリースされた。

Angular 2 は当初、グーグルが開発していた「AtScript」という JavaScript ベースのプログラミング言語で開発が行われていたが、後にマイクロソフトの TypeScript で開発が進められることが発表された（AtScript と TypeScript は一本化される）。MVC フレームワークの開発で天敵同士が協力するということで大きく報道されたことを覚えている方もいらっしゃるだろう。

Angular 2 では TypeScript が持つさまざまな機能が活用されている。例えば、クラスにメタデータを付加するためのデコレーターや、あるモジュールから特定のシンボルを導入する `import` 文などがそうだ。以下に例を示す。

```
import {bootstrap, Component, FORM_DIRECTIVES} from 'angular2/angular2';
@Component({
  selector: 'my-app',
```

```
template:`
<div>
  <div><input [(ng-model)]="test"></div>
  {{test}}
</div>
`,
// 「templateUrl: './doublebind.html',」として HTML テンプレートを外出しもできる
directives: [FORM_DIRECTIVES]
})
class AppComponent {
  public test = "foobar";
}

bootstrap(AppComponent);
```

Angular 2 で書いたアプリのコード (TypeScript)

ここでは「@Component」デコレーターにより、AppComponent クラスが「コンポーネント」であるというメタデータを付加している。コンポーネントは Angular 2 で導入されたもので、アプリのビューを構築したり、関連するビジネスロジックを記述したりするために使われる。AngularJS では、(AngularJS のディレクティブを含んだ) テンプレート／コントローラー／コントローラーのスコープにより、Web アプリを構成する特定の要素が実現されていたが、Angular 2 では「コンポーネント」を使用して、これらを実現する。

なお、上記のようにテンプレートが長くなるとコードが見つらくなるので、直書きをするのではなく、別ファイルに HTML コードを記述して、これを「templateUrl」フィールドで参照することも可能だ (コメント参照)。また、コンポーネントの定義自体を別ファイルとして import 文でこれを取り込むようにするのも、コンポーネント化によるコードの可読性と再利用性を高める上では重要になるだろう。TypeScript や ECMAScript 2015 では JavaScript にモジュールの概念が取り入れられているのも、Angular 2 でこうした構造化が可能になった理由の一つだと思われる。

そのため、コンポーネントには「selector」フィールドや「template」フィールドがある。前者は CSS セレクターを指定するもので、この場合は対応する HTML ファイルに <my-app> タグがあれば、そこにこのコンポーネント (AppComponent クラス) のインスタンスを表示する。

「template」フィールドは、その表示に使用されるテンプレートである (ここではバッククォートを使って複数行の文字列をテンプレートとしている)。「[(ng-model)]」はそのつづりからも分かるように、これが双方向デー

タバインディングを行うためのもので Angular 2 ディレクティブであり、「{{test}}」は単方向データバインディングを行う。

というわけで、上記の Angular 2 コードは先ほど見た AngularJS コードとほぼ同様なことを行うものだ。これをホストする HTML ファイルは次のようになる（抜粋）。

```
<!DOCTYPE html>
<html>
<head>
  …… Angular 2に必要な JavaScript ファイルの読み込み ……
  <script>
    System.config({
      transpiler: 'typescript',
      typescriptOptions: { emitDecoratorMetadata: true }
    });
    System.import('./app.ts');
  </script>
</head>
<body>
  <h1>Angular 2 Sample</h1>
  <my-app />
</body>
</html>
```

Angular 2 アプリの index.html ファイル

上で見た「my-app」が <body> タグ内に記述されていることに注目しよう。この他、<head> タグ内では、アプリの構成と TypeScript で書かれたアプリ本体の読み込みが行われている。

このように、Angular 2 はテンプレート／双方向データバインディングといった概念は AngularJS と同じでも、その構造は大きく変化している。上述の通り、現段階ではまだ開発者プレビューの状態だが、気になる人は今から Angular 2 にも注目しておくようにしよう。

Backbone.js

Backbone.js は古参の MVC フレームワークであり、現在でも広く使われている。AngularJS と比べるとシン

ブルで、その分柔軟性も高い。また、MVC フレームワークに分類されてはいるが、ビューとモデルの結合度が高いのも特徴といえる。コントローラーと呼ばれるオブジェクトは Backbone.js には存在せず、これに相当するものとして、Router オブジェクトが提供されている。

また、Backbone.js は [Underscore.js](#) (ユーティリティ関数を多数含んだ JavaScript ライブラリ) に依存している他 (Backbone.js と Underscore.js は作者が同じ)、DOM 操作を行ったり RESTful な API を利用して永続化したりするには [jQuery](#) がほぼ必須となる。そのため、Backbone.js はこれら二つの JavaScript ライブラリと同時に使用するのが一般的だ。

Backbone.js では Model / Collection / View / Router の各オブジェクトが重要な役割を果たしている。

- Backbone.Model : 単一のデータを表現するオブジェクト
- Backbone.Collection : Model のコレクションを表現するオブジェクト
- Backbone.View : Web ページの特定箇所を描画するビュー
- Backbone.Router : 主に URL のルーティング制御を行う

以下に簡単な (という割には長くなってしまったが) 例を示す。

```
<!DOCTYPE html>
<html>
  ..... 省略 .....
<body>
  <div id="msgbody"></div>
  <form id="myform">
    <input type="text" id="msg">
    <input type="submit" val="submit">
  </form>
  <script src="js/app.js"></script>
</body>
</html>
```

```
(function() {
  var Msg = Backbone.Model.extend({
    defaults:{
      body : "world",
```

```
    },
  });
  var MsgView = Backbone.View.extend({
    render: function(){
      $(this.el).html('hello ' + this.model.get('body'));
      return this;
    },
    initialize: function() {
      this.model.on('change', this.render, this);
    },
  });

  var MyFormView = Backbone.View.extend({
    el: '#myform',
    events: {
      'submit': 'submit'
    },
    submit: function(e) {
      e.preventDefault();
      this.model.set({body: $('#msg').val()});
    }
  });

  var msg = new Msg();
  var msgView = new MsgView({model: msg});
  var myFormView = new MyFormView({model: msg});
  $('#msgbody').html(msgView.render().el);
})();
```

Backbone.js で作成した簡単な Web アプリ

ここではコレクションを使用せずに、単一データを表す Backbone.Model オブジェクトを拡張した Msg オブジェクトを作成し、そのインスタンスを二つのビュー（MsgView オブジェクトと MyFormView オブジェクト）に関連付けている。

```
var Msg = Backbone.Model.extend({
    ..... 省略 .....
});

..... 省略 .....

var msg = new Msg();
var msgView = new MsgView({model: msg});
var myFormView = new MyFormView({model: msg});
```

モデルとビューの関連付け (JavaScript)

そして、ビューにはそれぞれ render 関数があり、これが実際の HTML 描画を行う。MsgView オブジェクトの render 関数は「this.model.get(...)」というプロパティゲッターを使用して、body プロパティの値を取得して、文字列 "hello " とそれを連結したものを描画している。一方、MyFormView オブジェクトではテキストボックスに入力された値を利用して、このモデルの値を更新している。

ここで重要なのが、モデルに更新が起こるとイベントが発生することだ。上に抜粋したように、二つのビューは同一の Msg オブジェクトをそのモデルとしているため、ここでは、MyFormView オブジェクトで行われた変更が 'change' イベントを介して、MsgView オブジェクトに伝えられる。そこで、このイベントを MsgView オブジェクトでは監視して、値が変更されたら render 関数を呼び出すことで、表示内容を更新するようにしている。

```
initialize: function() {
    this.model.on('change', this.render, this);
},
```

'change' イベントを監視するように初期化時に設定 (MsgView オブジェクト)

AngularJS と比べるとかなりコード量が多い。また、あくまでも Backbone.js 自体はアプリを抽象化したようなクラス／オブジェクトが用意されているわけでもない。モデルとビュー（とここでは触れなかったがルーター）をうまく切り分けて、アプリを設計できるようにするためのシンプルなフレームワークであり、目指すところはフルスタックな AngularJS とは異なるだろう。

実際には Backbone.js でのアプリ開発をサポートするフレームワークとして [Marionette.js](#) や、双方向データバインディングを実装する [Backbone.stickit](#) などのフレームワーク／ライブラリも存在するので、興味のある方はそれらについても調査してみよう。

クライアントサイドの MVC フレームワーク競争は、現状、AngularJS が大きくリードをしているが、Backbone.js はまだまだ使われているテクノロジーでもあり、これから AngularJS → Angular 2 への移行がうまくいかなければ、枯れたテクノロジーとしてこれからも使い続けられるかもしれない。

●今から始める JavaScript アプリ開発

.NET 開発者のための JavaScript ライブラリカタログ (デスクトップアプリ編)

Insider.NET 編集部 かわさきしんじ (2015 年 12 月 04 日)

本章では Windows、OS X、Linux で動作するクロスプラットフォームなデスクトップアプリを開発するためのフレームワークを取り上げる。

前章では MVC フレームワークを三つ取り上げた。本章では JavaScript でクロスプラットフォームなデスクトップアプリを開発するためのフレームワークを見てみよう。

クロスプラットフォームなデスクトップアプリ開発

ここで見ていくフレームワークは以下の二つだ。

- [Electron](#)
- [NW.js](#)

[Node.js](#) にはサーバーサイドで JavaScript を使用して、高速な Web アプリを開発するために必要な要素が詰まっているが（この辺りの事情については少々古い記事になるが『[サーバサイド JavaScript の本命「node.js」の基礎知識](#)』などを参考にしてほしい。基本は変わっていない）、これをクライアントサイドでも活用してネイティブデスクトップアプリを開発するための技術が Electron と NW.js になる。

また、これらは [Chromium](#)（簡単にいえば、グーグルの Chrome のベースとなるオープンソースな Web ブラウザー）と組み合わせて、Web ページを UI に持つアプリを構築することで、Chromium が動作する Windows、OS X、Linux でクロスプラットフォームにアプリを実行できるようにしている。

クロスプラットフォームと聞くと、Windows / iOS / Android の世界の話のような気もするがそうではなく、これらはデスクトップアプリまでも Web 標準技術でクロスプラットフォームに動かそうというお話である。なお、ここではインストール方法などについては割愛する。個々のページの説明を参照されたい（本稿では npm 経由でインストールしている）。

Electron

[Electron](#) は [GitHub](#) が開発しているオープンソースなフレームワークだ。以前は「Atom Shell」という名前

で知られていた。これを使用したアプリも数多く存在している。本フォーラムでも取り上げている [Visual Studio Code](#) は Electron を使用していることでもよく知られている。

Electron 自体はあくまでも、Web 標準技術でデスクトップアプリを開発するためのフレームワークなので、実際には Node.js 用の各種モジュールと組み合わせて、Web アプリを作る要領でデスクトップアプリを開発していくことになるだろう。

最小限の Electron アプリ

Electron の「[Quick Start](#)」ページにある最小限の Electron アプリ（を改変したもの）を以下に示す。最小限の Electron アプリは、package.json ファイル（アプリの構成を記述）、JavaScript ファイル（アプリのロジックや UI 制御コードを記述）、HTML ファイル（アプリの UI を構成）の三つで構成される。

```
{
  "name"    : "your-app",
  "version" : "0.1.0",
  "main"    : "main.js"
}
```

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Hello World!</title>
  </head>
  <body>
    <h1>Hello World!</h1>
    We are using node <script>document.write(process.versions.node)</script>,
    Chrome <script>document.write(process.versions.chrome)</script>,
    and Electron <script>document.write(process.versions.electron)</script>.
  </body>
</html>
```

```
var electron = require('electron');
```

```
var app = electron.app;
var BrowserWindow = electron.BrowserWindow;

var mainWindow = null;

app.on('window-all-closed', function() {
  app.quit();
});

app.on('ready', function() {
  mainWindow = new BrowserWindow({width: 800, height: 600});

  mainWindow.loadURL('file://' + __dirname + '/index.html');

  mainWindow.on('closed', function() {
    mainWindow = null;
  });
});
```

最小限の Electron アプリ

Electron の「[Quick Start](#)」ページより。上から順に package.json ファイル、index.html ファイル、main.js ファイルとなっている。

package.json ファイルでは、このアプリの名前とバージョン、エントリポイントを記述している。ここではエントリポイントは main.js ファイルとなっている。

HTML ファイル (index.html ファイル) の <script> タグ内部では、「process.versions.node」のように Node.js の提供する API を利用している。それ以外には特に説明の必要はないだろう。

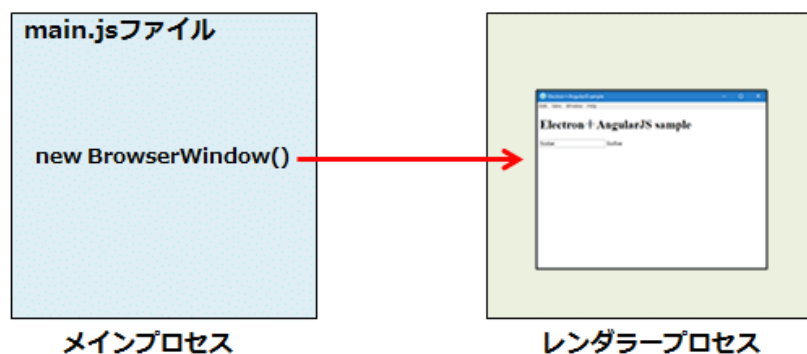
JavaScript ファイル (main.js ファイル) でやっていることをまとめると、Electron を導入して、全てのウィンドウが閉じられたらアプリを終了するように設定し (app.quit メソッド呼び出し)、Electron の初期化が完了したら (「app.on('ready', ……」) ウィンドウを作成して、そこに index.html ファイルの内容を読み込んでいる。

これを実行すると次のような画面が表示される。

メインプロセスとレンダラープロセス

ここで重要になるのが、Electron には**メインプロセス**と**レンダラープロセス**の 2 種類のプロセスがあることだ。

メインプロセスは OS にネイティブな GUI の管理やレンダラープロセスの起動、アプリ全体のライフサイクル管理を行う。レンダラープロセスは、個々の Web ページを表示する。ざっくりとまとめると、メインプロセスとは「アプリの構成を記述する package.json ファイルで main 属性に指定された JavaScript ファイル（上の例では main.js ファイル）を実行するプロセス」のことだ。そして、そこから上のコードサンプルにあるように「new BrowserWindow(...)」を実行することでレンダラープロセスが生成される。



メインプロセスとレンダラープロセス

レンダラープロセスからはネイティブな GUI（とはメニューやダイアログなどのことだ）を自由に操作できないようになっているが、これはセキュリティおよびリソースリークの危険を考慮してのことである。ただし、メインプロセスとレンダラープロセスの間ではプロセス間通信が可能なため、これを利用してレンダラープロセスからメインプロセスに GUI 操作を要求することは可能だ（後述）。

と書いたところで、あまり理解も進まないと思うので、以下では前章にチラリとお見せした AngularJS を利用した Web アプリコードを Electron アプリ化してみよう。

AngularJS を利用した簡単な Electron アプリ

AngularJS アプリを Electron アプリ化というと何やら大変そうだが、元が数行のコードしかないので、AngularJS 側のコードは改変することなく Electron アプリにできた（が、実際にはそこまで簡単ではないだろう）。以下にコードを示す。今度は main.js ファイル以外に、AngularJS コードを含む app.js ファイルが必要になる。

まずは index.html ファイルと対応する AngularJS コードだ。

```
<!DOCTYPE html>
<html ng-app="myapp">
<head>
  ..... 省略 .....
  <script src="js/angular.min.js"></script>
```



```
<script src="js/app.js"></script>
</head>
<body>
  ..... 省略 .....
  <div ng-controller="FooController as fooctrl">
    <p>
      <input type="text" ng-model="fooctrl.test" />
      {{ fooctrl.test }}
    </p>
  </div>
</body>
</html>
```

```
var app = angular.module('myapp', []);

app.controller('FooController', function() {
  this.test = "foobar";
});
```

AngularJS アプリを Electron アプリ化

これは双方向バインドを使用して、テキストボックスに入力された値をリアルタイムでその隣に表示するだけのアプリだ。この状態では Electron の要素は何ら存在していない。

エン트리ポイントとなる main.js ファイルを以下に示す。実は先ほどの main.js ファイルとほとんど変わっていない（変数名が変わったくらいだ）。

```
var electron = require('electron');
var app = electron.app;
var BrowserWindow = electron.BrowserWindow;

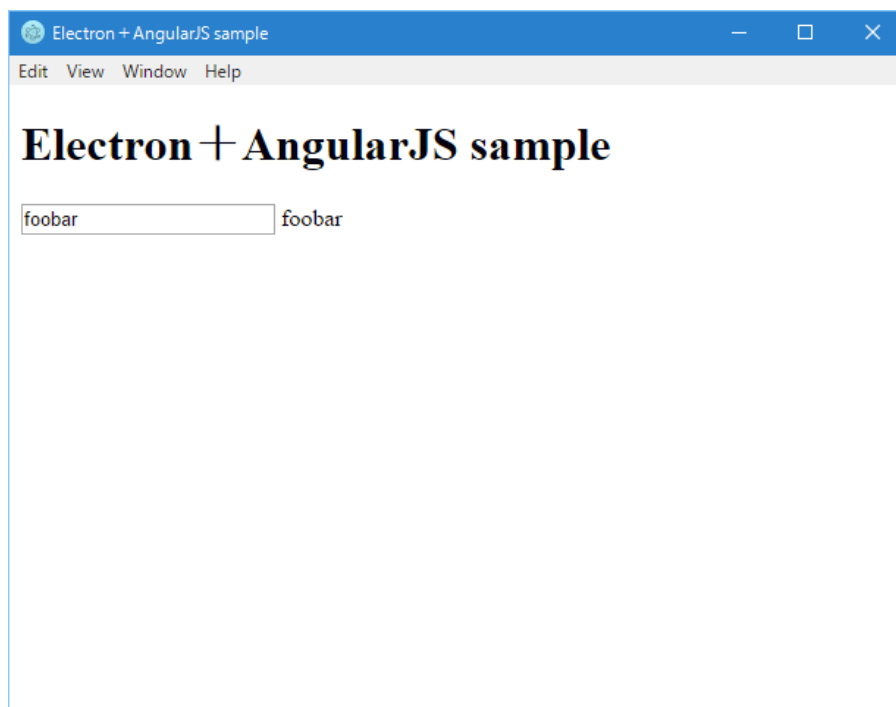
var mainWnd = null;

app.on('window-all-closed', function() {
  app.quit();
});
```

```
});  
  
app.on('ready', function() {  
  mainWnd = new BrowserWindow({width: 800, height: 600});  
  
  mainWnd.loadURL('file://' + __dirname + '/index.html');  
  
  mainWnd.on('closed', function() {  
    mainWnd = null;  
  });  
});
```

Electron アプリのエントリーポイント

これを実行すると次のようになる。



AngularJS アプリを Electron アプリ化

正直、AngularJS を使っただけで、説明することは特に増えていない。が、Electron を使うと（恐らくは）比較的容易に Web アプリのデスクトップアプリ化が実現できることが予想できると思う。しかも、これが Windows でも Mac でも Linux でも動作するというのはなかなか魅力的だ。

プロセス間通信を行ってみる

次に、メインプロセスとレンダラープロセスとの間で、プロセス間通信を行ってみよう。Electron でプロセス間通信を行うには二つの手段がある。一つは ipcMain / ipcRenderer モジュールを使用するもの。もう一つはより簡便にプロセス間通信を行える remote モジュールを使用するものだ。

最初に ipcMain / ipcRenderer モジュールを使って、テキストボックスに入力された値をメインプロセスに送信する例と remote モジュールを使用してレンダラープロセスからダイアログを表示する例を見てみよう。

index.html と app.js ファイルを次のように変更する。ボタンを三つ追加して、FooController クラスのメソッドと結び付けているだけだ。

```
<!DOCTYPE html>
<html ng-app="myapp">
  ..... 省略 .....
<body>
  ..... 省略 .....
  <div ng-controller="FooController as fooctrl">
    <p>
      <input type="text" ng-model="fooctrl.test" />
      {{ fooctrl.test }}
    </p>
    <button ng-click="fooctrl.send()">send async</button>
    <button ng-click="fooctrl.sendsync()">send sync</button>
    <button ng-click="fooctrl.showdialog()">show dialog</button>
  </div>
</body>
</html>
```

```
..... 省略 .....
var ipcRenderer = electron.ipcRenderer;
var dialog = electron.remote.dialog;

app.controller('FooController', function() {
  this.test = "foobar";
  this.send = function() {
```

```
ipcRenderer.send('asynchronous-message', this.test);  
};  
this.sendsync = function() {  
  console.log(ipcRenderer.sendSync('synchronous-message', this.test));  
};  
this.showdialog = function() {  
  dialog.showMessageBox({  
    type: "info",  
    title: "Info",  
    message: "hello from renderer proc",  
    buttons: ["OK"]  
  });  
};  
});
```

変更後の index.html ファイルと app.js ファイル

showMessageBox メソッド呼び出しではボタンは最低でも一つ必要ようだ。

HTML ファイルの説明は不要だろう。以下では、app.js ファイルに注目していく。

レンダラープロセス（メインプロセスで作成された Web ページ側）からプロセス間通信を行うには ipcRenderer モジュールを使用する。あるいは remote モジュールを介して、メインプロセスが提供する機能を使用する。これを行っているのが最初の 2 行だ。

メインプロセスへの送信を非同期に行うには ipcRenderer.send メソッドを使用する。これはメインプロセスへのデータを送ったら、そこでメソッド呼び出しは終了する。

同期的に通信するには ipcRenderer.sendSync メソッドを使用する。ただし、こちらはメインプロセスからの返信が到着するまで Web ページの動作をブロックするので注意が必要だ（メインプロセスから返された値が sendSync メソッドの戻り値となる。従って、上のコードではメインプロセスから返された値が console.log メソッドにより出力される）。

send / sendSync メソッドの第 1 引数はプロセス間通信に使用する「チャンネル」、第 2 引数は送信データである。

ダイアログ表示コードは後で見るとして、指定された二つのチャンネルで待ち受けるメインプロセスのコードは次

のようになる。

```
var electron = require('electron');
var app = electron.app;
var BrowserWindow = electron.BrowserWindow;
var ipcMain = electron.ipcMain;

..... 省略 .....

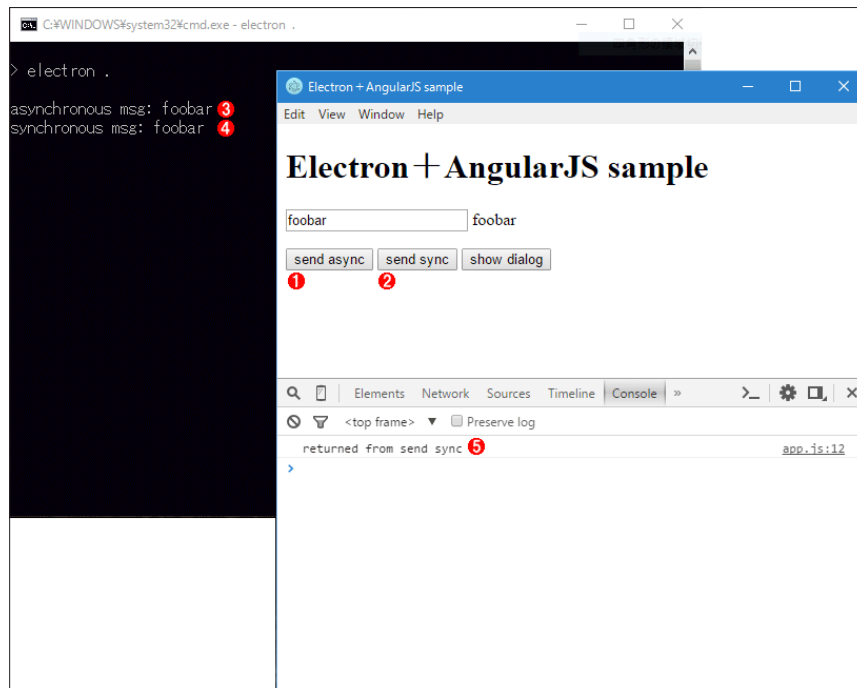
ipcMain.on('asynchronous-message', function(e, arg) {
  console.log('asynchronous msg: ' + arg);
});

ipcMain.on('synchronous-message', function(e, arg) {
  console.log('synchronous msg: ' + arg);
  e.returnValue = 'returned from send sync';
});
```

main.js ファイル (JavaScript)

メイン側では ipcMain.on メソッドを使用して、イベントを待ち受ける。第 1 引数には先ほど見た「チャンネル」を指定する。レンダラープロセスからメッセージが送信されたときには、第 2 引数に指定した無名関数で処理が行われる。この関数にはイベント情報を含むオブジェクトが渡されるが、同期通信では、このオブジェクトの returnValue プロパティに値をセットすることで、ブロックしていた sendSync メソッドが終了する（ここでは「returned from send sync」を設定している）。

アプリを実行すると次のようになる。



メインプロセスとレンダラープロセスのプロセス間通信

- (1) レンダラープロセスが非同期にメッセージを送信。
- (2) 非同期に送信されたメッセージをメインプロセスがコンソールに出力。
- (3) レンダラープロセスが同期的にメッセージを送信。
- (4) 同期的に送信されたメッセージをメインプロセスがコンソールに出力。
- (5) メインプロセスから返された値を、レンダラープロセスがコンソールに出力。

最後にダイアログを表示するコードを再掲しておこう。

..... 省略

```
var dialog = electron.remote.dialog;
```

```
app.controller('FooController', function() {
```

..... 省略

```
this.showdialog = function() {
```

```
  dialog.showMessageBox({
```

```
    type: "info",
```

```
    title: "Info",
```

```
    message: "hello from renderer proc",
```

```
    buttons: ["OK"]
```

```
  });
```

```
};
```

```
});
```

remote モジュールを使って、レンダラープロセスからダイアログを表示

先ほども述べたが、レンダラープロセスからはネイティブな UI を直接操作できない。Electron の remote モジュールを使用すると、上で見たような生なプロセス間通信を行わずに、メインプロセスが所有するオブジェクトを遠隔的に使用できる。ここでは、「remote.dialog」オブジェクトの `showMessageBox` メソッドに情報を適宜指定して呼び出しているだけだ。ただし、複数の OS が手元にある方は実際に試してみると分かるが、OS ごとに異なる UI が表示される。

このように、Electron を使うと、Web 標準技術を使って、OS ごとに固有な GUI を活用したネイティブアプリを容易に開発できる。また、Electron ではメインプロセスとレンダラープロセスの 2 種類のプロセスがあり、それぞれが可能なこと／不可能なことが明確に別れている。明快なフレームワーク設計だが、プロセス間で情報をやり取りするには通信を行う必要がある。

だいが長くなってしまったので、NW.js については簡単に触れるに留めておこう。

NW.js

NW.js もまた Node.js、Chromium、Web 標準技術を使用してクロスプラットフォームなデスクトップアプリを開発するためのオープンソースなフレームワークだ ([Intel Open Source Technology Center](#) で誕生した)。以前は「node-webkit」と呼ばれていた。また、誕生したのは Electron よりも先で、すでに多くのデスクトップアプリが開発されている。

AngularJS を使用した簡単な NW.js アプリ

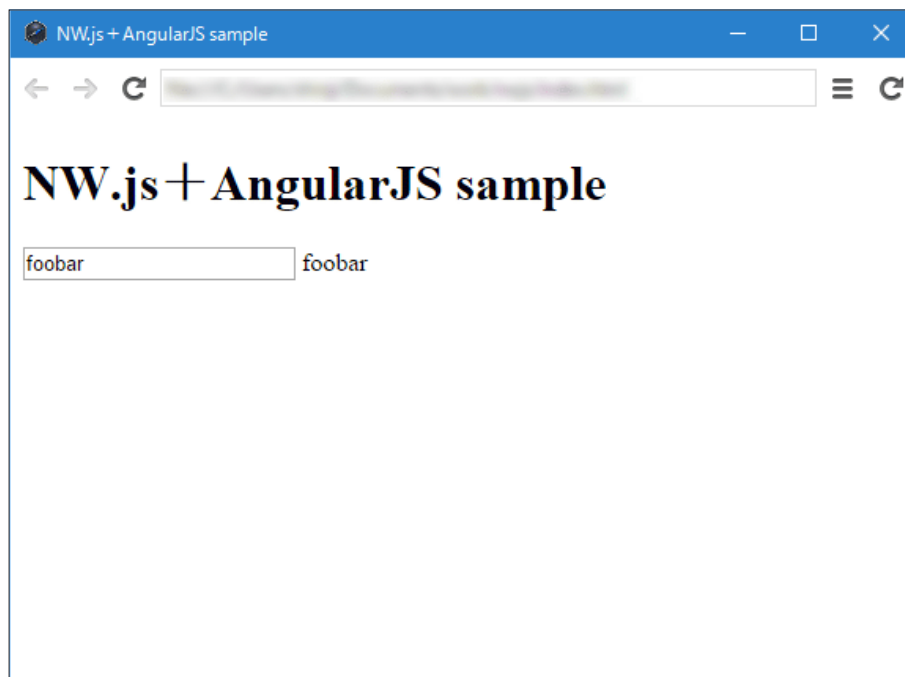
Electron のときと同じく、まずは AngularJS を使用した Web アプリを NW.js アプリ化してみよう。といっても、今回も元のコードにはほとんど手を加えていない。なお、Electron ではアプリのエントリーポイントは JavaScript ファイルとなっていたが、NW.js では HTML ファイルをエントリーポイントとするのが一般的だ (JavaScript ファイルをエントリーポイントとすることも可能なようだ)。

そのため、`package.js` ファイルは以下のようになる。

```
{
  "name": "nwjs-test",
  "main": "index.html"
}
```

NW.js アプリの `package.json` ファイル

HTML ファイルと JavaScript ファイルについては、冒頭に示したものと同様なので省略しよう。これを実行すると次のような画面が表示される。



NW.js アプリの実行画面

ウィンドウの外枠こそ違うが、Electron アプリと同様にさっくりとデスクトップアプリ化できた。

ただし、このままでは NW.js のパワーは全く使っていない（最初の Electron アプリで Electron コードが全く使われていなかったのと同じだ）。そこで、ありがちだが、index.html ファイルの中から NW.js（というか、Node.js）の機能を使用してみよう。

```
<!DOCTYPE html>
<html ng-app="myapp">
  ..... 省略 .....
<body>
  ..... 省略 .....
  <script>
    var os = require('os');
    document.write("platform: " + os.platform());
  </script>
</body>
</html>
```

変更した index.html

このように <script> タグ内部でも require 関数の呼び出しが可能だ。これは、NW.js はウィンドウが作成されるときに Node.js のコンテキストに存在する global オブジェクトのメンバー（require 関数など）を、そのウィンドウの Web のコンテキスト（window オブジェクト）に移設するからだ。このとき、実際には require 関数は Node.js のコンテキストで呼び出され、その結果が Web コンテキストに返送されることになる。

NW.js のコンテキストは便利な面、コンテキストを意識する必要があり、慣れないと困惑を招く場合もある。詳細が気になる人は「[Transfer objects between window and node](#)」ページを参考に実際にコードを書きながら挙動を試してみるのがよい。

また、global オブジェクトは複数のウィンドウ間でも共有される。これを活用することで、ページ間での情報を受け渡しがとても簡便な形で行える。が、グローバルな変数の値はいつどこで誰が書き換えるかは分からない。使いすぎには注意しよう。

と言いながら、NW.js のサンプルの最後として、global オブジェクトの使用例を示す。index.html ファイルに以下のようにボタンを追加して、そのハンドラーを FooController クラスに追加する。

```
<!DOCTYPE html>
<html ng-app="myapp">
  ..... 省略 .....
  <body>
    ..... 省略 .....
    <div ng-controller="FooController as fooctrl">
      <p>
        ..... 省略 .....
      </p>
      <button ng-click="fooctrl.go2ndpage()">go 2nd page</button>
    </div>
  </body>
</html>
```

```
app.controller('FooController', function() {
  this.test = global.test || 'foobar';
  this.go2ndpage = function() {
    global.test = this.test;
    window.location.href = 'secondwin.html';
  };
});
```

```
};  
});
```

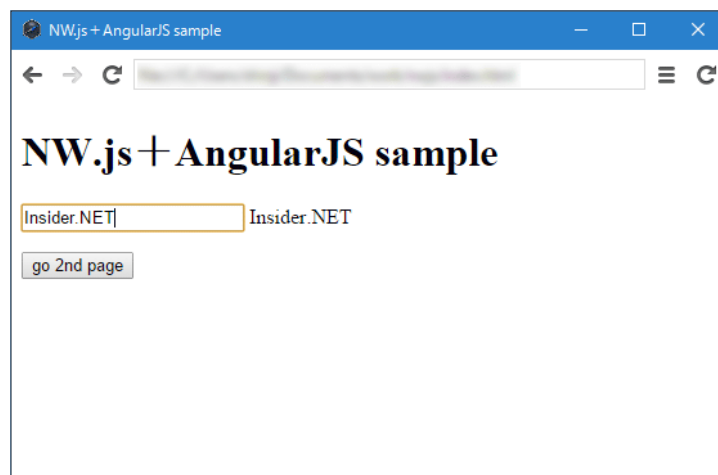
変更した index.html ファイルと app.js ファイル

app.js ファイルはコントローラーの登録部分だけを抜粋している。

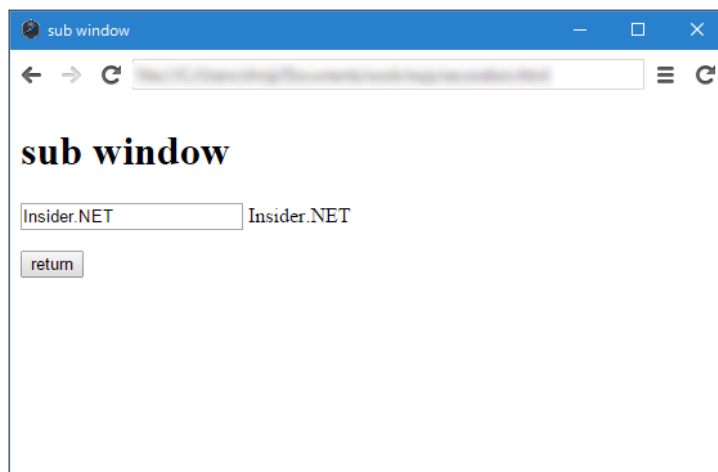
実際には、これと同様な構造の HTML ファイル (secondwin.html ファイル) と JavaScript ファイル (sub.js ファイル) を用意している。

ボタンがクリックされると、そのハンドラーでは「global.test」変数に「this.test」変数の値をセットして、secondwin.html ファイルの内容を読み込む。すると、secondwin.html ファイルが読み込んだ sub.js ファイルでは global.test 変数の値を使ってテキストボックスが初期化されるわけだ (「this.test = global.test || 'foobar';」 行。global.test 変数がなかった場合には "foobar" が初期値として使われるようにしてある)。

これを実行すると、次のようになる。



[go 2nd page] ボタンをクリック



最初のページで入力した値が受け渡される

ページ間の情報の受け渡し

ここでは複数ウィンドウではなく、単一ウィンドウでページ遷移をする形のアプリとなっているが、global オブジェクトを利用して情報を受け渡しているのが分かる。乱用はまずいが、時と場合によっては便利に使えるだろう。

Electron と NW.js

ここまで、Electron と NW.js について簡単に見てきた。どちらもクロスプラットフォームなデスクトップアプリを開発するためのフレームワークであるが、その構成は意外と異なっている。

「[Technical Differences Between Electron and NW.js](#)」ページでは、これら二つのフレームワークの技術的な差異について述べられている。簡単にまとめると次のようになる。

- **エントリポイント**: Electron は JavaScript ファイル。NW.js は HTML ファイルが一般的（ただし、JavaScript ファイルをエントリポイントとすることも可能）
- **コンテキスト**: Electron では Web ページに新たなコンテキストを導入しない。NW.js には Node.js コンテキストと Web コンテキストが存在する
- **Node 統合**: Electron では Chromium に手を加えていない。NW.js は Chromium に手を入れている（その分、コンテキストを便利に使える）
- **情報の受け渡し**: Electron ではプロセス間通信を使用。NW.js では global オブジェクトを使用可能

この他にも、インストーラーや自動アップデートなど、さまざまな面で相違点がある。興味のある方は上記のページや「[NW.js & Electron Compared](#)」ページを参考にしてほしい。

●今から始める JavaScript アプリ開発

.NET 開発者のための JavaScript ライブラリカタログ (gulp 編)

Insider.NET 編集部 かわさきしんじ (2015 年 12 月 18 日)

Web アプリ開発における日々の煩雑な作業を自動化してくれるツールである「gulp」の基本を本章では見ていこう。

前章ではデスクトップアプリを JavaScript で開発するためのライブラリを二つ紹介した。本章と次章は「ビルドシステム」あるいは「タスクランナー」と呼ばれるライブラリを二つ紹介しよう。本章では Visual Studio 2015 でも利用可能な gulp を取り上げる。

ビルドシステムとは

Web アプリ開発時には、その過程でさまざまな作業が必要になる。例えば、JavaScript コードの最小化／難読化や、CSS プリプロセッサによる CSS メタ言語のコンパイル（とその最小化）などがそうだ。こうしたことを自動化してくれるのが、Web アプリ開発におけるビルドシステムやタスクランナーと呼ばれるツールの役割だ。

有名なところでは、本章で取り上げる [gulp](#)^{*1} と次章に取り上げる [Grunt](#) がある。この他にも [Broccoli.js](#)、[Brunch](#) などのツールがある。

^{*1} gulp の [FAQ](#) によれば「gulp is always lowercase. The only exception is in the gulp logo where gulp is capitalized」（gulp は常に小文字表記。例外はロゴで、これは先頭が「G」になっている）とあるので、本稿ではこれに従い「gulp」と表記する。

gulp とは

[gulp](#) は、Node.js 上に実装されている「ストリーミング」ビルドシステムだ。ここでいう「ストリーミング」とは、ある入力を処理した結果（出力）を、次の処理の入力とすることで全体としての処理を進めていく概念だ（UNIX のパイプ処理を使い慣れている方であれば、よくご存じの概念だろう）。

本稿執筆時点（2015 年 12 月）では、gulp は Visual Studio 2015（以下、VS 2015）の ASP.NET Web アプリのプロジェクトテンプレートで標準のタスクランナーとして使用されていた。2016 年 11 月時点では ASP.NET の Web プロジェクトではこれとは別のタスクランナー（バンドルシステム）が使用されていることには注意されたい。以下では本稿執筆時点で使われていた gulp のスクリプトファイルを例に、gulp の特徴を見ていく。

なお、ソリューションエクスプローラーで Web アプリプロジェクトを右クリックして、コンテキストメニューの [Bundler & Minifier] - [Convert To Gulp] を選択することで、gulp のスクリプトファイルを作成できる。作成されるファイルは本稿のものとは異なっているが（「min:html」タスクがある、入力元／出力先の指定方法が変わっているなど）、その内容の基本は上記の方法で生成した gulp のスクリプトファイルにも当てはまるので、参考にしていきたい。

ASP.NET 5 アプリのプロジェクトテンプレートに見る gulp でのタスク定義

以下に ASP.NET 5 の Web アプリプロジェクトを作成した際に、同時に作成されていた gulpfile.js ファイルの内容を示す（上述した通り、現在の ASP.NET Core Web アプリプロジェクトでは異なるタスクランナーがファイルのバンドリング／圧縮に使われているが、その設定を gulp のスクリプトファイルに変換することは可能だ）。

```
"use strict";

var gulp = require("gulp"),
    rimraf = require("rimraf"),
    concat = require("gulp-concat"),
    cssmin = require("gulp-cssmin"),
    uglify = require("gulp-uglify");

..... 省略 .....

gulp.task("min:js", function () {
    return gulp.src([paths.js, "!" + paths.minJs], { base: "." })
        .pipe(concat(paths.concatJsDest))
        .pipe(uglify())
        .pipe(gulp.dest("."));
});

gulp.task("min:css", function () {
    return gulp.src([paths.css, "!" + paths.minCss])
        .pipe(concat(paths.concatCssDest))
        .pipe(cssmin())
        .pipe(gulp.dest("."));
});
```

```
gulp.task("min", ["min:js", "min:css"]);
```

ASP.NET 5 アプリのプロジェクトテンプレートで生成される gulpfile.js ファイル»

省略した部分にはファイル名の設定 (node-glob 形式)、成果物をクリーンするタスクの定義が含まれている。

このように gulp では、JavaScript コードを使って、その処理内容を記述していくのが大きな特徴である (gulp と並んでメジャーなタスクランナーである Grunt では、JSON 形式に構成を記述していくのが一般的だ)。冒頭部分では、gulp 本体と gulp で使用するプラグインをスクリプトに導入している (require メソッド)。プラグインは単一の処理を行うもので、これを pipe メソッドでつないでいくことで、gulp は全体として一つのタスクを処理する。

以下ではこのスクリプトをもう少し詳しく見ていこう。

タスク定義

タスクの定義には gulp.task メソッドを使用する。その基本構文は次のようになる。

```
gulp.task("タスク名", function () {  
    そのタスクで実行する処理  
});
```

タスク定義の構文

「min:js」と「min:css」の二つのタスクはこの形式でタスクを定義している (上述の方法で ASP.NET Core の Web アプリプロジェクトから gulpfile.js ファイルを生成した場合のコードについては後述のコラムを参照されたい)。

```
gulp.task("min:js", function () {  
    return gulp.src([paths.js, "!" + paths.minJs], { base: "." })  
        .pipe(concat(paths.concatJsDest))  
        .pipe(uglify())  
        .pipe(gulp.dest("."));  
});
```

「min:js」タスクの定義

なお第 2 引数には、そのタスクが依存するタスク名を配列として指定できる。上記の gulpfile.js ファイルの最終行はこの形式でタスクを定義している。

```
gulp.task("min", ["min:js", "min:css"]);
```

「min」タスクは「min:js」と「min:css」の二つのタスクに依存する

「min」タスクは「min:js」タスク（JavaScript ファイル）と「min:css」タスク（CSS ファイルの最小化）を行うだけなので、第3引数の関数は省略されている。

次に「min:js」タスクを例に、タスク定義で行う処理をどう記述していくかを見てみる。

タスクの内容

「min:js」タスクでは次のような処理が記述されていた。

```
return gulp.src([paths.js, "!" + paths.minJs], { base: "." })
  .pipe(concat(paths.concatJsDest))
  .pipe(uglify())
  .pipe(gulp.dest("."));
```

「min:js」タスクで行う処理

「gulp.src(...)」以降を「return」している理由は後で説明する。まずは、実際に行う処理のスタート地点となる gulp.src メソッドから見ていこう。

gulp.src メソッドは **node-glob** 形式で記述された処理対象のファイル名とオプションを受け取る（後者は省略可能）。そして、名前がマッチしたファイルを「**vinyl**」（仮想ファイルフォーマット）形式のストリームとして生成する。

ここではファイル名として「paths.js」と「!" + paths.minJs」が配列の形で渡されている（ファイル名を一つだけ指定するのであれば、配列にする必要はない）。なお、paths.js と paths.minJs は上記のリストで省略した部分でその値が次のように定義されている。

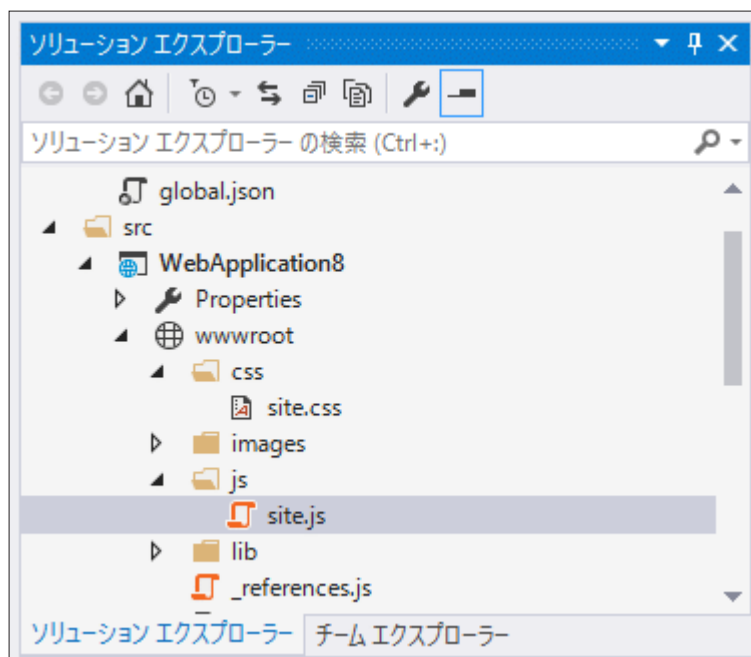
- paths.js : ./wwwroot/js/**/*.js
- paths.minJs : ./wwwroot/js/**/*.min.js

paths.minJs については、「!" + paths.minJs」と、その先頭に「!"」が付加されている。「!"」は否定を意味し、ここでは処理対象のファイルに paths.minJs にマッチするものを含まないように指定していることになる。

node-glob 形式についての詳しい説明は省略するが（リンクを参照されたい）、要するに「min:js」タスクで

は「./wwwroot/js」ディレクトリ以下にある「*.min.js」ファイルを除く全ての「*.js」ファイルを処理対象として、それらのストリームを生成するという意味になる。

そして、生成されたストリームは、pipe メソッドによって concat プラグインへと引き渡される。そこで、次にパイプによるストリーム処理を見ていこう。なお、実際の ASP.NET 5 アプリのデフォルトのディレクトリ構成を見ると、処理対象の JavaScript ファイルは以下の一つだけだ。



デフォルトでは site.js ファイルのみが対象となる

パイプによる処理の結合

生成されたストリームは、pipe メソッドを使って [concat プラグイン](#)へと送られる。「concat」（連結）という名前の通り、これは複数のファイルを一つにまとめてくれるプラグインだ。concat プラグインに渡している引数 `paths.concatJsDest` は結合後のファイル名で「js/site.min.js」となっている（省略部分で定義）。

```
.pipe(concat(paths.concatJsDest))  
.pipe(uglify())  
.pipe(gulp.dest("."));
```

パイプによる処理の結合

連結されたファイルは次に最小化（あるいは難読化）を行う [uglify プラグイン](#)へ送られる。処理が行われたストリームは最後に `gulp.dest` メソッドへと送られる。

`gulp.dest` メソッドは指定されたディレクトリへとファイルを書き込む（そして、その内容をさらにストリームと

して出力するので、ここから処理をさらに進めることも可能だ。これは、複数の JavaScript ファイルを連結したものを一度ファイルとして書き込んでから、今度はそれを uglify プラグインで処理して「.min.js」ファイルとして保存するといった場合に便利に使える。

gulp.dest メソッドには引数として「.」(カレントディレクトリ)を指定している。実際には、指定されたディレクトリに対して、ファイルの相対パスを追加した位置に書き込みが行われる。この場合は「./wwwroot/js」ディレクトリ以下にファイルが作成されることになる。詳細については「[gulp API docs](#)」ページを参照してほしい。

「min:css」タスクも同様な処理を行っている。また、ここでは紹介を省略している「clean:js」「clean:css」「clean」の三つのタスクは npm の [rimraf パッケージ](#) を使用して、結合／最小化されたファイルの削除を行っている（これは gulp のプラグインではない）。

だいたいの説明が終わったので、実際にこれを実行してみよう（ただし、デフォルトでは site.js ファイルの中身はコメントだけで空に等しいので、ファイルが実際に作成されることだけを確認することになる）。

【コラム】 ASP.NET Core の Web アプリプロジェクトの場合

本稿の冒頭でも述べたが 2016 年 11 月の時点では、Visual Studio 2015 の ASP.NET Core Web アプリプロジェクトでは gulp ではなく、Bundler & Minifier というツールを使って、JavaScript ファイルのバンドリング／最小化が行われる。ただし、その構成を gulp のスクリプトに変換することが可能だ。以下に示すのは、変換後のスクリプトファイルから「min:js」タスクの設定内容を抜き出したものである。

```
gulp.task("min:js", function () {
  var tasks = getBundles(regex.js).map(function (bundle) {
    return gulp.src(bundle.inputFiles, { base: "." })
      .pipe(concat(bundle.outputFileName))
      .pipe(uglify())
      .pipe(gulp.dest("."));
  });
  return merge(tasks);
});
```

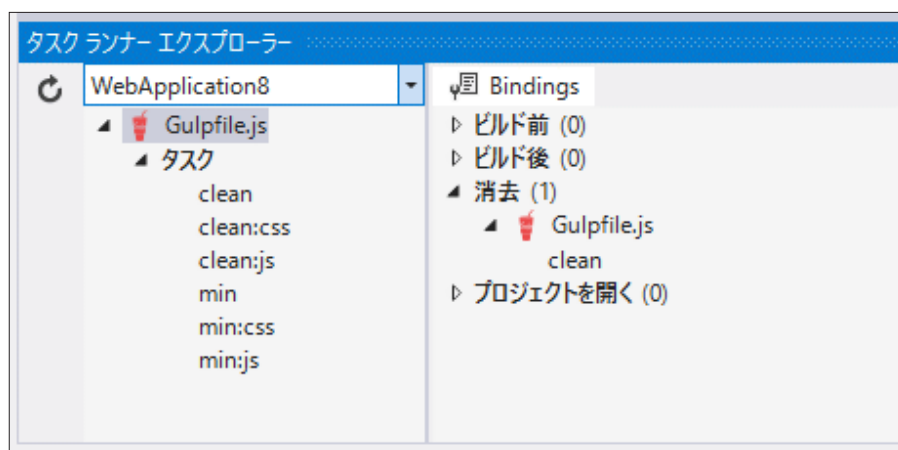
ASP.NET Core Web アプリプロジェクトにおける「min:js」タスクの設定

上で述べた paths.js などが使われておらず、さらに「return gulp.src～」がラップされているのが

大きな違いといえる。こちらのバージョンでは `merge-stream` というプラグインを使用して、複数のストリームを一つのストリームに統合するようになっている。また、入力ファイル、出力先の指定は、`getBundles` 関数を用いて外部ファイル (`bundleconfig.json` ファイル) から入手するようになっている。

gulp タスクの実行

VS 2015 の ASP.NET 5 アプリのプロジェクトテンプレートの話をしているので、実行にもまずは VS 2015 を使ってみよう。これには VS 2015 のメニューバーから [表示] - [その他のウィンドウ] - [タスク ランナー エクスプローラー] を選択して、タスクランナーエクスプローラーを表示する。

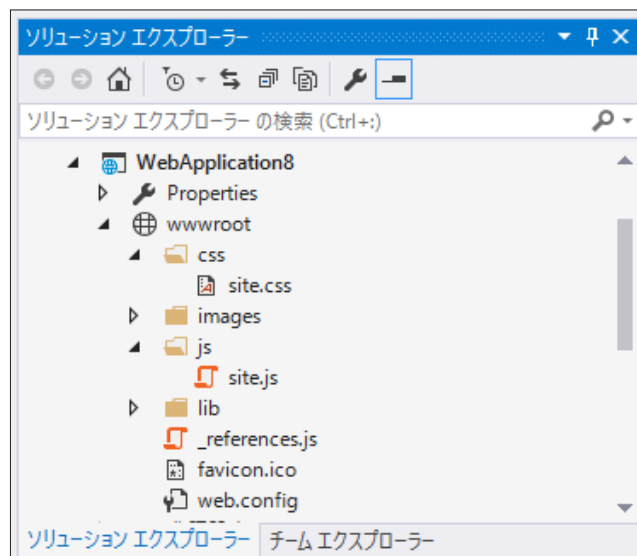


タスクランナーエクスプローラー

左側のペーンには上で紹介した gulp のタスクが表示されている。これらをダブルクリックすれば、そのタスクが実行される。タスクを実行すると、右側にその実行結果が表示される。また、[Bindings] というタブも表示されている。

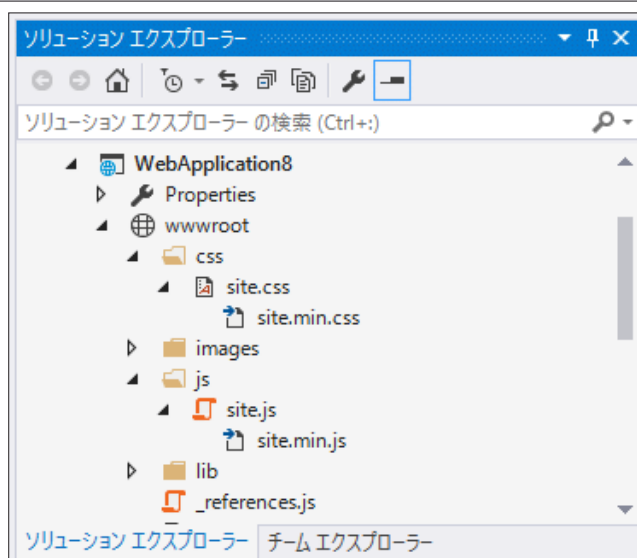
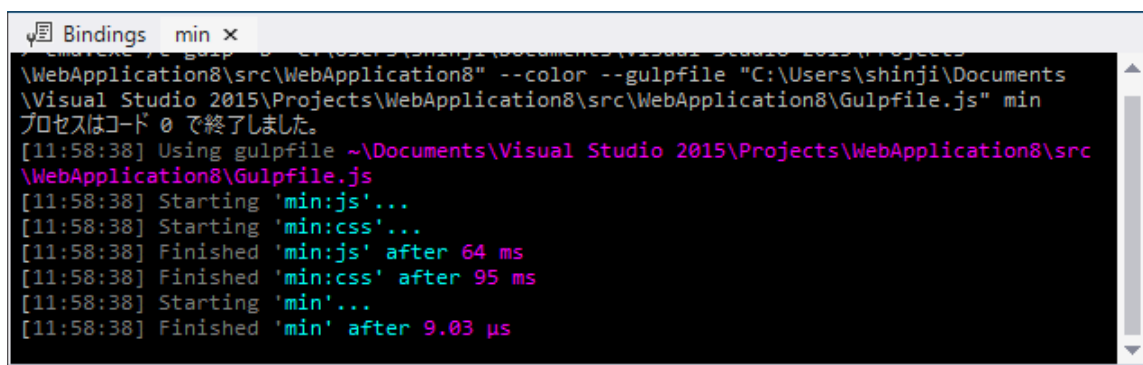
[Bindings] タブは、VS 2015 の IDE で行われる各種のイベントと gulp タスクのバインディングを表す。ここでは [消去] が gulp の「clean」タスクと関連付けられていることが分かる。つまり、メニューバーなどからソリューションをクリーンすると「clean」タスクが自動的に実行されるということだ。

では、実際にタスクを実行してみよう。その前に、`wwwroot` 以下のディレクトリ構成を再掲しておく。`site.css` ファイルと `site.js` ファイルがあることを確認してほしい。



タスク実行前の wwwroot ディレクトリ

この状態で左側のペーンで [min] タスクをダブルクリックすると、タスクランナーエクスプローラーの右側のペーンに次のように表示されるとともに、ソリューションエクスプローラーでもファイルが二つ生成されたことが分かる。



「min」タスクを実行

上の画像はタスクランナーエクスプローラーの右側のペーンに出力された結果。下の画像を見ると、「site.min.css」ファイルと「site.min.js」ファイルが作成されたことが分かる。

なお、gulp はコマンドラインからでも実行できる（筆者の環境では何ごともなく実行できたが、パス設定などが必要であれば適宜設定を行ってほしい）。これには gulp コマンドに実行したいタスクを指定して実行する。

```
> gulp min
[12:32:21] Warning: gulp version mismatch:
[12:32:21] Global gulp is 3.9.0
[12:32:21] Local gulp is 3.8.11
[12:32:22] Using gulpfile ~\Documents\Visual Studio 2015\Projects\.....
[12:32:22] Starting 'min:js'...
[12:32:22] Starting 'min:css'...
[12:32:22] Finished 'min:js' after 100 ms
[12:32:22] Finished 'min:css' after 93 ms
[12:32:22] Starting 'min'...
[12:32:22] Finished 'min' after 53  $\mu$ s
```

コマンドラインからの gulp の実行

グローバルにインストールした gulp とローカルにインストールした gulp のバージョンが違っていると怒られているのはご愛敬だ。環境やバージョンによっては、出力結果が異なる場合がある。

タスクは並列に実行される

ここで興味深いのはタスクランナーエクスプローラーの出力結果だ。最初に説明した通り、「min」タスクは「min:js」タスクと「min:css」タスクの二つに依存する（簡単にはこれらを実行する）タスクなので、これらが実行されるのはよいとして、これら二つが並列に実行されていることに注目してほしい。

```
[11:58:38] Starting 'min:js'...
[11:58:38] Starting 'min:css'...
[11:58:38] Finished 'min:js' after 64 ms
[11:58:38] Finished 'min:css' after 95 ms
[11:58:38] Starting 'min'...
[11:58:38] Finished 'min' after 9.03  $\mu$ s
```

タスクは並列に実行される

「min:js」タスクと「min:css」タスクが並列に実行されているのが分かる（強調書体の部分）

gulp ではタスクは「デフォルトでは、全力で並列に実行する」（意識）とされている。そのため、二つのタスクは並列に実行される。そして、「min」タスクは、これら二つのタスクに依存しているので、それらの実行が完了した時点で実行を開始するのだ（ただし、処理を関数として渡していないので実際には何もしない）。

並列に実行されるタスクの終了を gulp が知るには、何かのヒントが必要になる。このために、ここで使われているのが gulp.src メソッドで作成されるストリームだ。ここでは、ストリームが終了するのを gulp が待機することで非同期処理が行えるようになっている。gulp.task メソッドで、gulp.src メソッドの結果を「return」しているのはそのためだ。

```
gulp.task("min:js", function () {  
  return gulp.src([paths.js, "!" + paths.minJs], { base: "." })  
    .pipe(concat(paths.concatJsDest))  
    .pipe(uglify())  
    .pipe(gulp.dest("."));  
});
```

gulp.src メソッドの結果を「return」することでタスクの並列実行が可能になっている

gulp はストリームの終了を待機して、非同期処理が完了したことを知る。

試しに「min:js」と「min:css」の二つのタスク定義から「return」を削除して、「min」タスクを実行すると、その出力は次のようになる。タスクを並列処理するためのヒントが得られないため、ここではシリアルに実行されている。

```
[12:10:41] Starting 'min:js'...  
[12:10:41] Finished 'min:js' after 12 ms  
[12:10:41] Starting 'min:css'...  
[12:10:41] Finished 'min:css' after 4.66 ms  
[12:10:41] Starting 'min'...  
[12:10:41] Finished 'min' after 5.75 μs
```

タスクがシリアルに実行される

ストリームを返送していない場合には、あるタスクが完了してから次のタスクが実行される。

タスクを並列かつ順序性を持って実行する方法としては、今見たように gulp.task メソッドでストリームを返ししながら、あのタスクが終わったら、このタスクを実行するように gulp に伝える必要がある。コールバックを記述する方法もあるが、ここでは紹介は割愛する。

例をもう一つ見てみよう。ここでは、「min:css」タスクが「min:js」タスクに依存するようにする（サンプルなので、意味論的には間違っていることには注意）。これにはタスク定義を次のように変更する。「min:js」タスクは最初の定義のままだ。

```
gulp.task("min:js", function () {
  return gulp.src([paths.js, "!" + paths.minJs], { base: "." })
    .pipe(concat(paths.concatJsDest))
    .pipe(uglify())
    .pipe(gulp.dest("."));
});

gulp.task("min:css", ["min:js"], function () {
  return gulp.src([paths.css, "!" + paths.minCss])
    .pipe(concat(paths.concatCssDest))
    .pipe(cssmin())
    .pipe(gulp.dest("."));
});
```

「min:css」タスクが「min:js」タスクに依存するように変更

変更したら、「clean」タスクを実行して、「site.min.css」ファイルと「site.min.js」ファイルを削除して、今度は「min:css」タスクを実行する。この実行結果を以下に示す。

```
[12:24:04] Starting 'min:js'...
[12:24:04] Finished 'min:js' after 62 ms
[12:24:04] Starting 'min:css'...
[12:24:04] Finished 'min:css' after 74 ms
```

実行結果

すると、「min:css」タスクを実行しただけのはずなのに、「min:js」タスクも実行され、「site.min.js」ファイルが作成されることも確認できるはずだ。

gulp のプラグイン

ここまでASP.NET 5用のプロジェクトテンプレートにより生成されるgulpfile.jsファイルで使われているプラグインだけを使ってきた。だが、gulpには便利なプラグインが他にもたくさん用意されている。例えば、CSS周りでSassファイルのコンパイルを行う [gulp-scss](#) プラグイン、TypeScriptコードのコンパイル用の [gulp-typescript](#) プラグイン、ECMAScript 2015コードをBabelを使ってコンパイルするための [gulp-babel](#)、各種リントツール用のプラグインなどがある。

プラグインは gulp の [プラグイン検索ページ](#) で検索できるので、こんなプラグインがあるといいなと思ったらまずはここで検索を試みよう。

本章では gulp を取り上げた。タスクランナーエクスプローラーを使えば、VS 2015 IDE から便利に gulp を使えるので、いろいろと試してみよう。

●今から始める JavaScript アプリ開発

.NET 開発者のための JavaScript ライブラリカタログ (Grunt 編)

Insider.NET 編集部 かわさきしんじ (2015 年 12 月 24 日)

Web アプリ開発における日々の煩雑な作業を自動化してくれるツールである「Grunt」の基本を本章は見ていこう。

本章は、gulp と並んで有名なタスクランナーである「Grunt」を取り上げる。

Grunt とは

Grunt は gulp 同様に、Web アプリ開発で日常的に発生する細々とした作業を自動化してくれるツールだ。できることは同様だが、その設計思想やタスクの構成方法は異なっている。gulp の方が後発であることと、Node.js が提供するストリーム API を使って Node.js 的なタスク構成を行うことに対して、Grunt では JSON 形式でタスクを記述していく。

以下では、Visual Studio 2015 (以下、VS 2015) の ASP.NET Web アプリのプロジェクトで Grunt を使ってみよう。

[コラム] VS 2015 で Grunt を使うには

前章で説明したように、2016 年 11 月の時点では gulp は VS 2015 のデフォルトのタスクランナーではない。そのため、デフォルトの状態では、package.json ファイルなどはプロジェクトには含まれていない。

面倒なことをしたくなければ、ソリューションエクスプローラーでプロジェクトを右クリックして、コンテキストメニューから [Bundler & Minifier] - [Convert To Gulp] を選択し、(本稿では使用しないが) gulp のスクリプトファイルを作成しよう。これにより、package.json ファイルが作成される。その後は、以下で行っているような「npm install ~ --save-dev」コマンドを安心して実行できる。

また、Grunt のコマンドライン、プラグインをインストールして、Gruntfile.js ファイルを作成すれば、VS 2015 のタスクランナーエクスプローラーが自動的にこれを識別してくれるはずだ。

VS 2015 環境への Grunt のインストール

VS 2015 の ASP.NET の Web アプリのプロジェクトでは、タスクランナーエクスプローラーを介して、IDE か

ら Grunt の各種タスクを実行できる。ただし、そこで使用するプラグインやローカルにインストールが必要な Grunt は前もってインストールをしておく必要がある。ローカルに Grunt をインストールするには、プロジェクトディレクトリで以下のようなコマンドを実行する（上述のコラムも参照されたい）。

```
> npm install grunt --save-dev
```

プロジェクト配下のディレクトリに Grunt をインストール

これにより、package.json ファイル（プロジェクトの構成や依存関係を記述したファイル）に「開発時に Grunt を使用する」ことが記録される。プラグインのインストールについては、この後、順次必要なものをインストールしていく。基本的にはコマンドプロンプトから以下のようなコマンドを実行すればよい。これにより、先ほどと同様に、package.json ファイルで「開発時にはこのプラグインを使用する」という情報が記録され、Grunt の構成ファイルからそのプラグインを使用できるようになる。

```
> npm install grunt-contrib-<プラグイン名> --save-dev
```

プラグインをインストールするコマンドライン

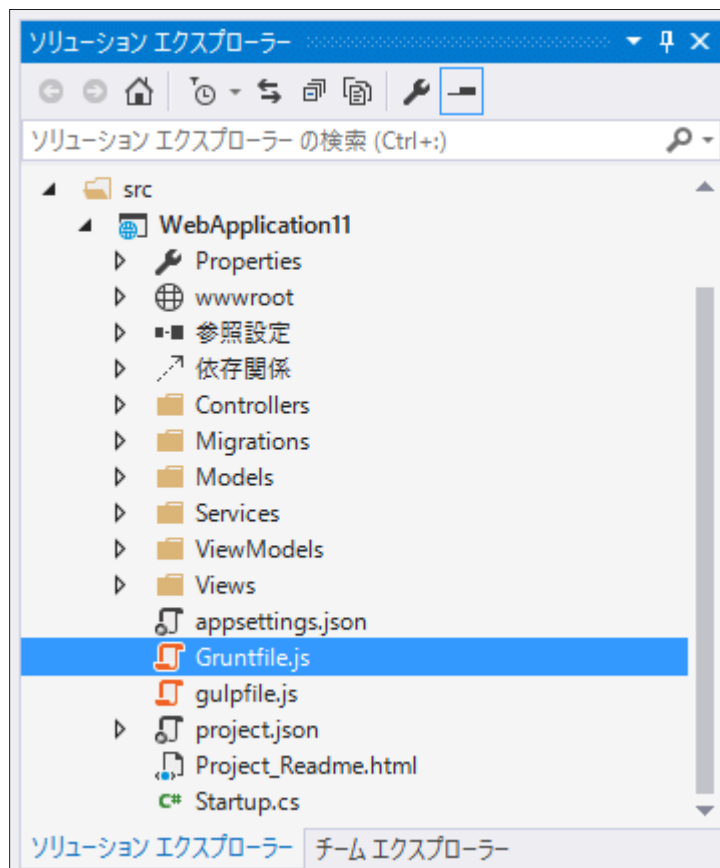
また、コマンドプロンプトから手作業で Grunt を実行するには、Grunt のコマンドラインインターフェースを「グローバル」にインストールしておく必要もある。

```
> npm install grunt-cli -g
```

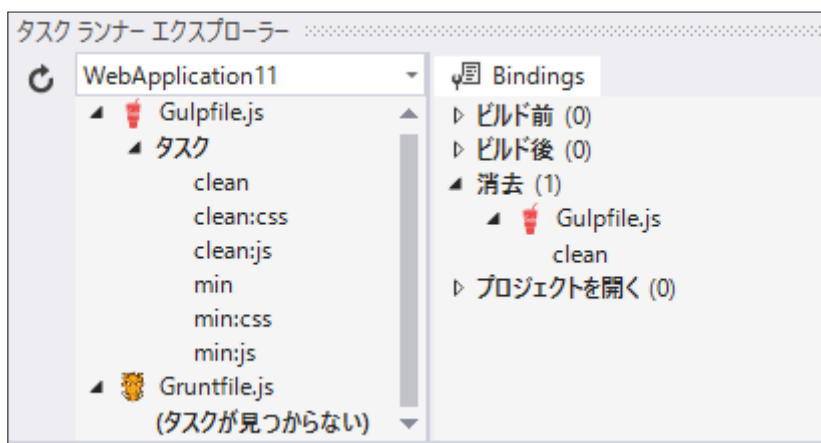
グローバルに Grunt のコマンドラインインターフェースをインストール

これにより、任意のディレクトリから「grunt」コマンドを実行できるようになる（が、上で見たようにプロジェクトで Grunt を使用するには、ローカルに Grunt をインストールしておく必要がある）。

上記の作業を行い、プロジェクトのルートディレクトリに Gruntfile.js ファイルを作成しよう。タスクランナーエクスプローラーを開くと、次のように表示されるはずだ。



プロジェクトのルートに Gruntfile.js ファイルを作成



タスクランナーエクスプローラーに [Gruntfile.js] が表示される

タスクランナーエクスプローラーに Grunt を認識

ただし、まだタスクを何も定義していないので、「(タスクが見つからない)」と表示されている。そこで、まずは簡単なタスクを構成してみよう。

Hello Grunt

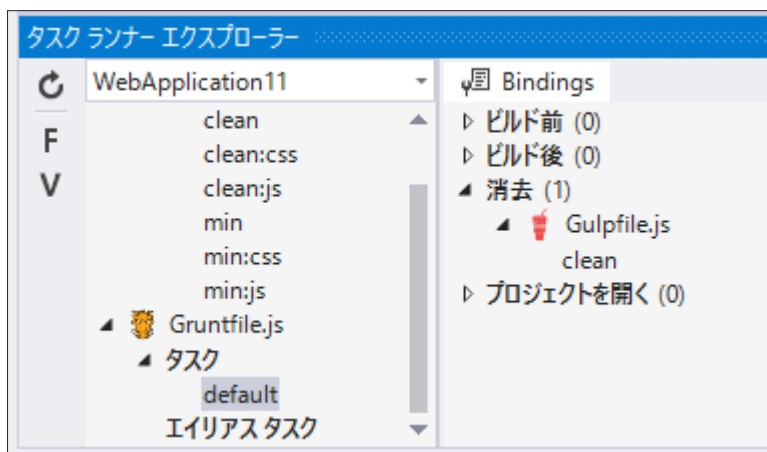
以下は、Grunt で Hello メッセージをコンソールに表示するタスクを構成したものだ。

```
module.exports = function (grunt) {  
  grunt.registerTask('default', function () {  
    console.log('hello from grunt');  
  });  
};
```

Hello メッセージを表示するタスク

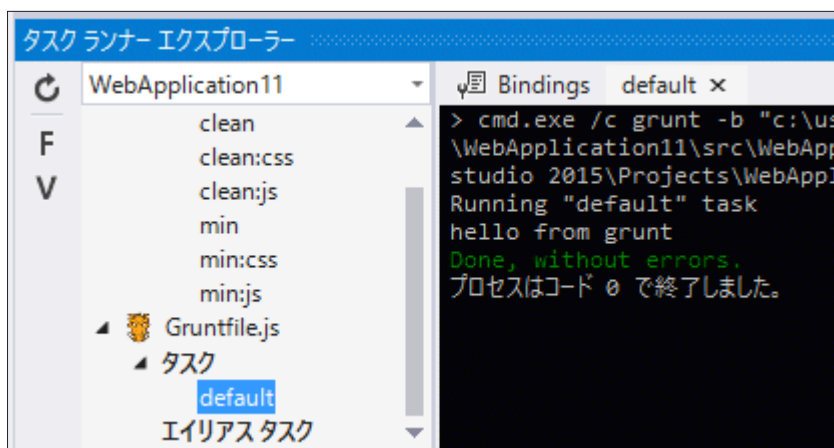
大ざっぱに説明しておく、これは Gruntfile.js ファイル（モジュール）が無名関数を外部に公開していて、その関数では Grunt に「default」という名前のタスクを登録しているということになる（その内容はもちろん、コンソールにメッセージを表示するだけだ）。

これを記述して、Gruntfile.js ファイルを保存すると、タスクランナーエクスプローラーの表示が次のように変化する（変化しないときには、左上にある「更新」ボタンをクリックする）。



タスクランナーエクスプローラーに「default」タスクが表示された

「default」をダブルクリックすると、その実行結果が次のように表示される。



default」タスクの実行結果

コマンドラインからは次のように実行できる。ちなみに「default」タスクは、grunt コマンドに何も指定しなかったときに実行されるタスクなので、以下では単に「grunt」とだけ入力している。

```
> grunt
Running "default" task
hello from grunt

Done, without errors.
```

コマンドラインからの grunt コマンドの実行

では、より前章と同様な、JavaScript ファイル／CSS ファイルを連結して、最小化するようなタスクを定義しながら、Grunt での実際のタスク構成を見ていこう。

Gruntfile.js ファイルの構造とタスク定義

先ほどの Gruntfile.js ファイルでは省略したが、Grunt のタスク構成は `grunt.initConfig` メソッドで行う。先ほどの `grunt.registerTask` メソッドは Grunt への「カスタムタスク」の登録に使用する。

ということで、もう少しちゃんとした Gruntfile.js ファイルの構造は次のようになる。

```
module.exports = function (grunt) {
  grunt.initConfig({
    ..... Grunt で実行するタスクの定義 .....
  });

  ..... grunt.loadNpmTasks メソッドで使用するプラグインを読み込む .....

  ..... grunt.registerTask メソッドなどでカスタムタスクを登録 .....
};
```

Gruntfile.js ファイルの構造

ここでは以下の二つのプラグインを使ってみよう。Grunt では、プラグインとはタスクを実行する本体といってもよいもので、ファイルの連結や最小化、リンティングなどなど、多数のプラグインが提供されている。その名前は「grunt-contrib-＜プラグイン名＞」となる。

- [grunt-contrib-uglify](#) : 複数の JavaScript ファイルの内容をまとめて最小化する
- [grunt-contrib-cssmin](#) : 複数の CSS ファイルの内容をまとめて最小化する

プラグインは Grant の「[Plugins](#)」ページで検索可能だ。

gulp では、複数ファイルを concat プラグインでまとめ、パイプ（ストリーム）を介して、それを uglify プラグインに渡すことで、連結と最小化を行っていたが、Grunt では粒度のより大きなプラグイン（タスク）が提供されているともいえる。

プラグインのインストール

これらのプラグインを使用するには、コマンドプロンプトから以下のコマンドを実行しておく必要がある ***1**。

```
> npm install grunt-contrib-uglify --save-dev
> npm install grunt-contrib-cssmin --save-dev
```

grunt-contrib-uglify / grunt-contrib-cssmin プラグインのインストール

以上でプラグインを Gruntfile.js ファイルから利用できるようになった。

***1** ちなみに VS 2015 の Node.js アプリ用のプロジェクトテンプレートを使って、Node.js アプリを開発する場合には、npm でのモジュール管理機能が IDE に組み込まれている。こうした機能が早いところ、ASP.NET 5 アプリでも使えるようになるとうれしい。

JavaScript ファイルの連結と最小化

連結といっても、ASP.NET 5 の MVC アプリのプロジェクトテンプレートでは、wwwroot ディレクトリ以下には JavaScript ファイルは一つしかないので、実際には特に何もしないことになるが、そこは雰囲気ということでご容赦されたい。

JavaScript ファイルを連結して最小化するための構成は以下のようになる。

```
module.exports = function (grunt) {
  grunt.initConfig({
    uglify: {
      js: {
        src: ['wwwroot/js/**/*.js', 'wwwroot/js/site.min.js'],
        dest: 'wwwroot/js/site.min.js'
      }
    }
  });
};
```

```
    }  
  }  
});  
  
grunt.loadNpmTasks('grunt-contrib-uglify');  
};
```

JavaScript ファイルの連結／最小化を行うタスク

grunt.initConfig メソッド内では上のように JSON 形式の記述方法を使って、タスクの構成を行っていく。「grunt-contrib-＜プラグイン名＞」の「＜プラグイン名＞」の部分をタスクとして記述する（この場合は「grunt-contrib-uglify」の「uglify」）。

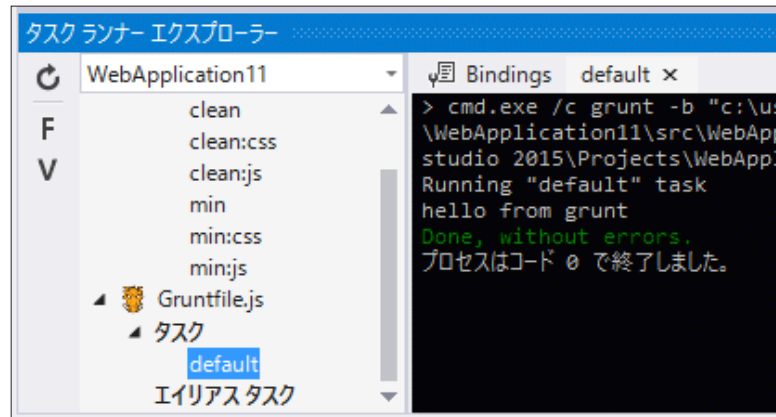
多くのプラグインでは複数のタスクを定義できる。ここでは前章と同様な「js」という名前のタスクのみを定義している（が、grunt-contrib-uglify プラグインが JavaScript ファイルだけを対象とするのでこれは名前としてはよろしくない。実際には、行う処理に応じたタスク名を指定するようにしよう）。

Grunt からは、これらのタスクには「uglify」タスク、そのサブタスク「uglify:js」のようにしてコマンドラインやタスクランナーエクスプローラーからアクセスできる。

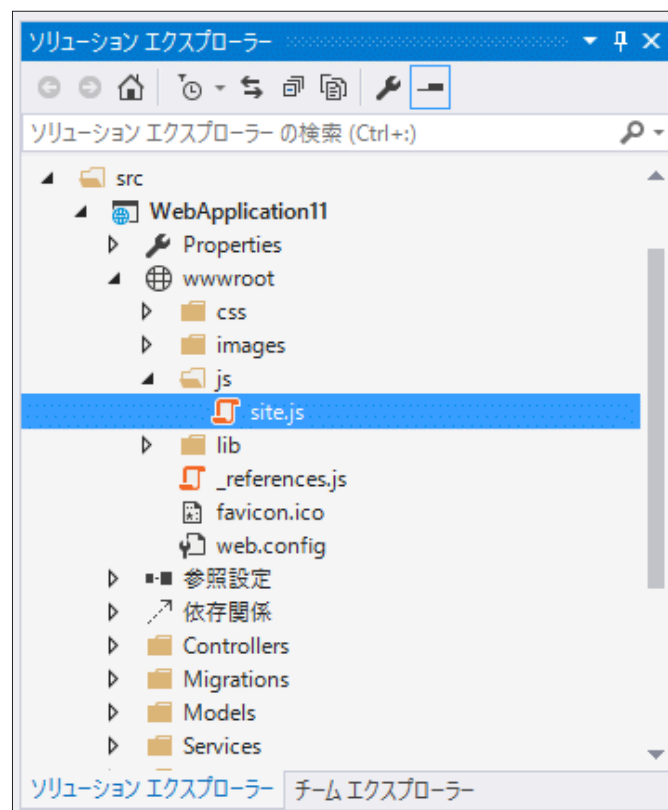
その下の src プロパティと dest プロパティは、処理対象のファイルと処理結果の書き込み先のファイルとなる。ここでは、処理対象は「wwwroot/js」以下の任意の「.js」ファイルとなっている。書き込み先は「wwwroot/js/site.min.js」ファイルだ。ただし、同一のディレクトリからファイルを読み出して、ファイルを書き込んでいるので、処理対象から「site.min.js」ファイルを除外するように src プロパティには「!wwwroot/js/site.min.js」を指定している（dest ディレクトリに処理後のファイルを書き出すような場合には、このような考慮は不要になる）。

最後に、grunt.loadNpmTasks メソッドで「grunt-contrib-uglify」プラグインを読み込んでいる。

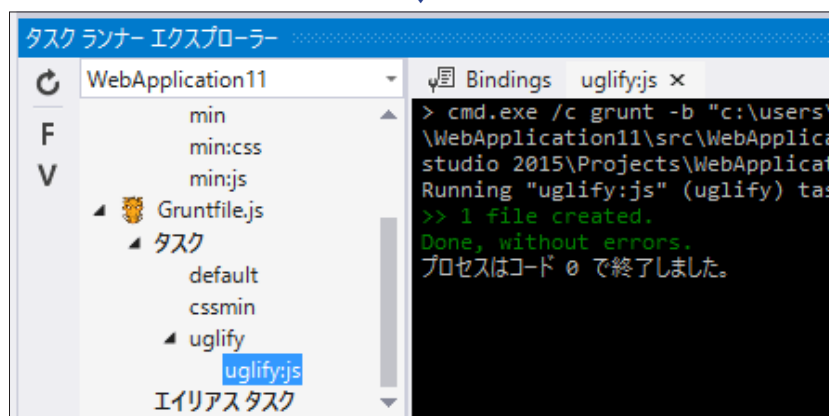
タスクを定義したら、タスクランナーエクスプローラーからこれを実行してみよう。これにはタスクランナーエクスプローラーで「uglify:js」をダブルクリックすればよい（site.min.js ファイルがあったら、ソリューションエクスプローラーから削除しておこう）。



実行前の「wwwroot/js」ディレクトリの内容



[uglify.js] をダブルクリック



「site.min.js」ファイルが作成された

「uglify.js」タスクの実行

これが Grunt でのタスク定義と実行の基本型となる。

CSS ファイルの連結と最小化

CSS ファイルの連結と最小化についても見ておこう。内容はほとんど変わらないので、コードの詳細な説明は省略する。

```
module.exports = function (grunt) {  
  grunt.initConfig({  
    uglify: {  
      js: {  
        src: ['wwwroot/js/**/*.js', '!wwwroot/js/site.min.js'],  
        dest: 'wwwroot/js/site.min.js'  
      }  
    },  
    cssmin: {  
      css: {  
        src: ['wwwroot/css/**/*.css', '!wwwroot/css/site.min.css'],  
        dest: 'wwwroot/css/site.min.css'  
      }  
    }  
  });  
  
  grunt.loadNpmTasks('grunt-contrib-uglify');  
  grunt.loadNpmTasks('grunt-contrib-cssmin');  
};
```

CSS ファイルの連結と最小化

パッケージ内容の読み込みとバナーの追加

Grunt では、そのプロジェクトの構成や依存関係を記述した package.json ファイルの内容を読み込んで、それをタスク内で使用することもできる。これには例えば、次のように記述する(以下はあくまでも例である。uglify タスクは本来、最小化を行うタスクなのでこういうものは入れない方が好ましい)。


```
module.exports = function (grunt) {  
  grunt.initConfig({  
    pkg: grunt.file.readJSON('package.json'),  
    uglify: {  
      options: {  
        banner: '/* <%= pkg.name %>: <%= pkg.version %>'  
      },  
      ..... 省略 .....  
    },  
    ..... 省略 .....  
  });  
  ..... 省略 .....  
};
```

package.json ファイルの読み込みとバナーの追記

package.json ファイルの読み込みには `grunt.file.readJSON` メソッドを使う。これにより、package.json ファイルの内容が読み込まれる。その内容は「<%= %>」で囲むことで利用できる。

この例では、「uglify」タスクの options プロパティ（とは、このタスクのオプションのことだ）で「banner」プロパティを指定して、パッケージ名（`pkg.name` プロパティ）とそのバージョン（`pkg.version` プロパティ）を「site.min.js」ファイルの先頭に追記するようにしている。

このように `initConfig` メソッド内では、タスク名以外のプロパティを追加することで、これを内部で「<%= %>」表記を使い参照できるようになる。もう一つ例を示しておこう。例えば、[grunt-contrib-concat](#) プラグインを使い、ファイルの連結だけを行うタスクがあったとする（「grunt-contrib-concat」プラグインの使い方についてはこれまでと同様なので省略する）。できるのは `all.js` ファイルと `all.css` ファイルの二つだ。

```
module.exports = function (grunt) {  
  grunt.initConfig({  
    concat: {  
      js: {  
        src: [  
          'wwwroot/js/**/*.js',  
          '!wwwroot/js/site.min.js',  
        ],  
      },  
    },  
  });  
};
```

```
        '!wwwroot/js/all.js'
    ],
    dest: 'wwwroot/js/all.js'
  },
  css: {
    src: [
      'wwwroot/css/**/*.css',
      '!wwwroot/css/site.min.css',
      '!wwwroot/css/all.css'
    ],
    dest: 'wwwroot/css/all.css'
  },
  uglify: {
    js: {
      src: [
        'wwwroot/js/**/*.js',
        '!wwwroot/js/site.min.js',
        '!wwwroot/js/all.js'
      ],
      dest: 'wwwroot/js/site.min.js'
    },
    cssmin: {
      css: {
        src: [
          'wwwroot/css/**/*.css',
          '!wwwroot/css/site.min.css',
          '!wwwroot/css/all.css'
        ],
        dest: 'wwwroot/css/site.min.css'
      }
    }
  }
});
```

```
..... 省略 .....  
grunt.loadNpmTasks('grunt-contrib-concat');  
};
```

JavaScript ファイルの連結タスク／最小化タスク、CSS ファイルの連結タスク／最小化タスクで src プロパティが同じ

注目してほしいのは、JavaScript ファイルを連結するタスク（「concat:js」タスク）と最小化するタスク（「uglify:js」タスク）で、src プロパティの値が同じことだ。ここでは、どちらも『「wwwroot/js」ディレクトリ以下にある JavaScript ファイルのうち、これらのタスクが生成したものを除いたもの全て』ということになっている（要するに「all.js」ファイルと「site.min.js」ファイルは除外する）。CSS 関連のタスクについても同様だ。

これらは重複している上に、入力が煩雑だ。というわけで、一つにまとめるのが正しい。これはもっと単純に次のように記述できる。

```
module.exports = function (grunt) {  
  grunt.initConfig({  
    jssrc: [  
      'wwwroot/js/**/*.js',  
      '!wwwroot/js/site.min.js',  
      '!wwwroot/js/all.js'  
    ],  
    csssrc: [  
      'wwwroot/css/**/*.css',  
      '!wwwroot/css/site.min.css',  
      '!wwwroot/css/all.css'  
    ],  
    concat: {  
      js: {  
        src: '<%= jssrc %>',  
        dest: 'wwwroot/js/all.js'  
      },  
      css: {  
        src: '<%= csssrc %>',  
        dest: 'wwwroot/css/all.css'  
      }  
    }  
  })  
};
```

```
    },  
    ..... 省略 .....  
  });  
  
  ..... 省略 .....  
};
```

処理対象のファイルの指定を 1 カ所にまとめる

このように共通要素をプロパティにくくりだし、「<%= %>」記法を使って、それらを参照することで Gruntfile.js ファイルを読みやすく、かつメンテナンスしやすくなる。

本章では成果物をクリーンするタスクは定義しなかったが、興味のある方は [grunt-contrib-clean](#) プラグインを使って、勉強がてらタスクを定義してみよう。

ここまでは VS 2015 環境で Grunt を使用して、タスクを自動化する基礎について見た。gulp と Grunt では、できることは似たようなものだが、Node.js のストリーム API をベースに JavaScript ベースでタスクを定義していく gulp、JSON での記述がメインとなる Grunt とその特徴は大きく異なる。また、gulp では単一機能を提供するプラグインを組み合わせで一つのタスクとするのに対して、Grunt ではプラグインが提供する処理の粒度がもう少し大きくなるようだ。どちらを選択するかは、個々人の好みということもあるだろうが、VS 2015 で gulp が採用されるなど、今後の主流となりそうなのは gulp かもしれない。

Grunt 自体の詳細な解説は「[Grunt で始める Web 開発爆速自動化入門](#)」を参照してほしい。

● C# × JavaScript

C# 開発者のための最新 JavaScript 事情（クラス定義）

Insider.NET 編集部 かわさきしんじ（2015 年 10 月 22 日）

C# と JavaScript におけるクラス定義を比較し、TypeScript や ECMAScript 2015 で JavaScript プログラミングがどう変わるかを見ていく。

クロスプラットフォーム開発の時代の到来

現在はクロスプラットフォーム開発が当たり前の時代だ。Visual Studio 2015 のプロジェクトテンプレートを見ても、Windows 以外の OS 向けのアプリ開発が当然のようにサポートされている。モバイルデバイスでは iOS、Android などの非 Windows 系統 OS が一般的であり、デスクトップ OS に関しても Windows が優位を保ってはいるものの、OS X (Mac) や Linux などが存在感を示している。

このような状況で、クロスプラットフォーム開発を行おうという場合、.NET 開発者には大きく分けて二つの選択肢がある。一つはもちろん、.NET Core / Mono などの OSS 化された .NET Framework をベースにすることである。この場合、これまでに磨きを掛けてきた技術をクロスプラットフォーム開発でも利用できる。

そしてもう一つの選択肢は HTML、JavaScript、CSS といった Web 標準技術を使用することだ。今や JavaScript は「Web アプリ開発におけるアセンブリ言語 *1」どころか、デスクトップアプリ開発におけるアセンブリ言語ともなりつつある。Web ブラウザー、JavaScript の実行エンジンなどがあらゆる OS 上に搭載されたことで、これらはいとも簡単に OS の壁を打ち破ってクロスプラットフォームなアプリを実行するための基盤となった。

工具箱にはツールがいくつあっても困ることはない。そこで、以下では C# のコードを JavaScript 系統の各種言語で書き直しながら、その相違点や類似点を見ていく。TypeScript、ECMAScript 2015 は JavaScript 系統の言語でありながら、C# と同様なクラスベースのオブジェクト指向プログラミングも可能である。.NET 開発者がこれから JavaScript の世界に飛び込むというのであれば、従来の JavaScript 5 ではなく、これらの言語を手にしてみるのがよいかもしれない。

*1 スコット・ハンセルマンによるブログ記事「[JavaScript is Assembly Language for the Web: Semantic Markup is Dead! Clean vs. Machine-coded HTML](#)」「[JavaScript is Web Assembly Language and that's OK.](#)」を参照されたい（英語）。要するに、JavaScript は今や Web アプリやデスクトップアプリの世界における中間言語的な存在となりつつあるということだ。

C# と JavaScript

実際にコードを見る前に、ありがちではあるが、C# と JavaScript の思想の違いと、2015 年時点での JavaScript の周辺事情について簡単にまとめておこう。

クラスベース vs プロトタイプベース

C# はクラスベースのオブジェクト指向言語だ。オブジェクトを生成するためのひな型としてクラスを定義し、それを基に実際のインスタンスを生成し、それらのプロパティやメソッドを活用して、プログラムを組み上げていく。

これに対して、JavaScript ではプロトタイプと呼ばれるオブジェクトを基にプログラムを組み上げていく。何らかのオブジェクトは、そのプロトタイプ（原型）を基に生成され、必要に応じて生成されたオブジェクトにさまざまな属性を付加したり、それら进行操作したりしていくのがプロトタイプベースの言語の特徴だ。そして、JavaScript ではこのプロトタイプを利用することで、クラスベースのオブジェクト指向言語と同様な処理を行える。

簡単な例を以下に示す（JavaScript 5）。

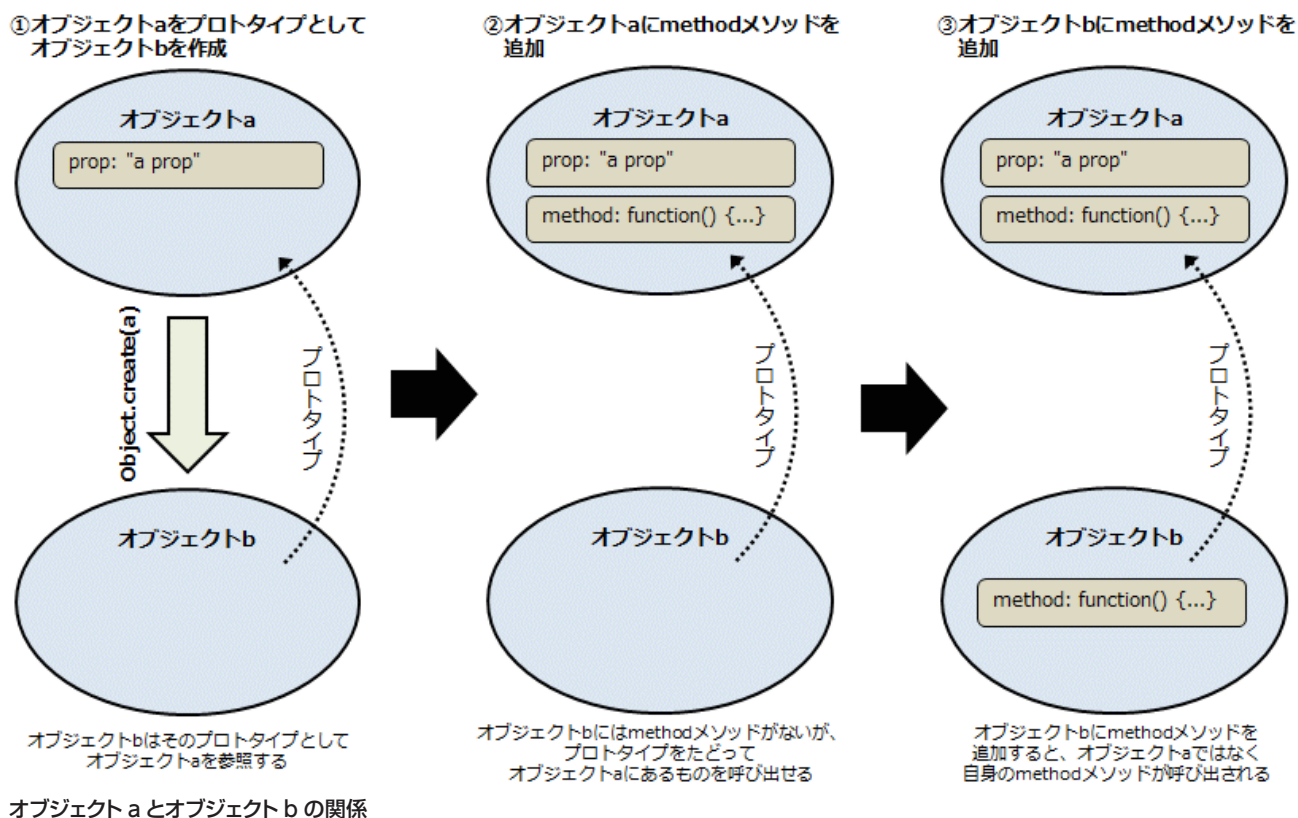
```
var a = {  
  prop: "a prop"  
};  
  
var b = Object.create(a); // (1)  
  
a.method = function() { console.log("hello"); } // (2)  
b.method();  
b.method = function() { console.log("goodbye"); } // (3)  
b.method();
```

プロトタイプを利用した JavaScript プログラミング（ただし、意味はない）

このコードではまず「a」という名前のオブジェクトを生成し、次に (1) で Object.create メソッドを使い、オブジェクト a を「プロトタイプ」としてオブジェクト b を生成している（Object.create メソッドは引数に指定されたオブジェクトをプロトタイプとして、新規にオブジェクトを生成するメソッド）。その後、(2) でオブジェクト a の method プロパティに関数を設定している（関数をその値とするプロパティのことをメソッドと呼ぶ）。そして、オブジェクト b に対してそのメソッドを呼び出している。

オブジェクト a にメソッドを追加したのに、オブジェクト b を利用してそれを呼び出せるのは、「オブジェクト b の

プロトタイプがオブジェクト a であり、オブジェクト b に method メソッドがない場合にはそのプロトタイプをたどり、オブジェクト a の method メソッドを探し当てる」からだ（これはクラスの継承階層をたどって、メソッド呼び出しを解決するのに似ているが、その機構がクラスなのかプロトタイプなのかが大きく異なる点だ）。これを図にすると次のようになる。



最後に、**(3)** でオブジェクト b に同名の method メソッドを追加して、そのメソッドを呼び出している。オブジェクト b が method メソッドを持つようになったので、今度はオブジェクト a の同名メソッドではなく、オブジェクト b 自身のメソッドが呼び出される。

そして、このプロトタイプという機構を使うことで、JavaScript ではクラスベースのオブジェクト指向プログラミングにおけるメンバーの定義、クラスの継承などに相当する処理を行える。

この他にも、JavaScript は動的型付け／プライベートなメンバーを基本的には持てない／インターフェースがない／名前空間がない、C# は基本的に静的型付け／関数がファーストクラスオブジェクトではないなど、C# と JavaScript との間で異なる言語的な特徴は多々あるが、それについては本特集で C# コードを JavaScript コードに書き直していく際に取り上げていくことになるだろう。

JavaScript 5、ECMAScript、TypeScript

現在、広く使われている JavaScript のバージョンは 5.1 である。ここでいう「JavaScript 5」(JavaScript 5.1)

とは [Ecma international](#) が策定するスクリプト言語仕様である [ECMAScript 5.1](#) に準拠して、各ベンダーが実装しているスクリプト言語実装／実行環境の総称だ。

2015 年 6 月には、ECMAScript 5.1 に対する大幅なアップデートとして ECMAScript 2015 が策定された。ECMAScript は、JavaScript 5 に対してクラスベースのオブジェクト指向プログラミングを可能にする構文拡張、アロー関数、2 進／8 進リテラル、デフォルト引数、可変長引数などの機能追加が行われたバージョンである（2016 年 6 月には ECMAScript 2016 も策定されているが、これについては本稿では触れない）。各 Web ブラウザーにおける新機能のサポート状況は「[ECMAScript 6 compatibility table](#)」で確認できる。2016 年 10 月時点ではデスクトップ OS 版の Web ブラウザーではかなりの数の機能がサポートされる状態になった（モバイルデバイスの Web ブラウザーでのサポートはこれからというところだ）。

JavaScript の言語仕様自体の進化の一方で「altJS」と呼ばれる、JavaScript の代替言語を開発しようという活動も活発に行われている。

当初は Web ブラウザー内で動作するスクリプト言語として生まれた JavaScript だが、現在では Web アプリを開発したり、Windows 8.x や Windows 10 のストアアプリ（UWP アプリ）を開発したり、あるいは [Electron](#) のように Windows / OS X / Linux で動作するクロスプラットフォームなデスクトップアプリを開発したりと、その用途は大きく広がっている。そして、これらのアプリ開発を行う上でピュアな JavaScript ではうまくいかない部分も出てきた。これをカバーするために、JavaScript にさまざまな機能を追加しながらも、最終的にはそれを JavaScript のコードにコンパイルして実行可能な新たな言語が登場したのである。

そうした altJS 言語の代表格といえるのがマイクロソフトの手による [TypeScript](#) だ。その言語仕様は JavaScript 5 の言語仕様のスーパーセットとなっており、正しい JavaScript 5 コードは正しい TypeScript コードとなる一方で、JavaScript 5 に対してクラス定義、型注釈、アロー関数などの機能が追加されている。

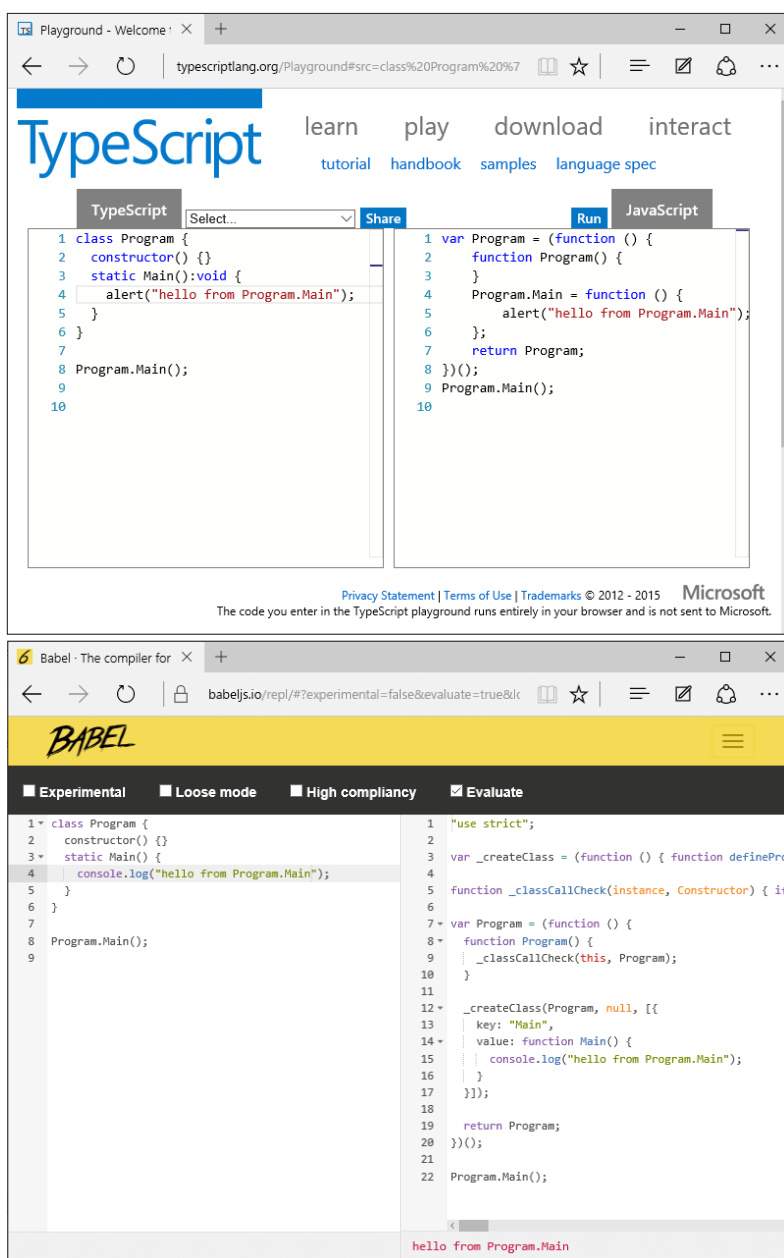
TypeScript は ECMAScript 2015 の仕様を意欲的に取り込んでいるが（クラス定義など）、その一方で独自に拡張している機能もある。その最終的なゴールは ECMAScript 2015 に完全準拠したプログラミング言語となることではなく、よりよい JavaScript として、ECMAScript の最新仕様のスーパーセットであり続けることにあ
るかもしれない。

ECMAScript 2015 / TypeScript を試してみるには

TypeScript に関しては、Visual Studio が標準でサポートしている。が、「[Playground](#)」ページで単純なコードであれば、実際に入力して、それがどんな JavaScript コードにコンパイルされるかを確認し、コードを実行してみることも可能だ。本稿で出てくる TypeScript コードは Playground ページで動作を確認している。

同様に、ECMAScript 2015 も Web ブラウザーを使って「[Babel の Try it out](#)」ページでそのコードを実地に試せるようになっている。[Babel](#) は ECMAScript 2015 コードを JavaScript 5 コードにコンパイルしてくれる、JavaScript コンパイラーであり、altJS と同様に ECMAScript 2015 のコードを JavaScript 5 のコードにコンパイルすることで、ECMAScript 2015 をサポートしていない Web ブラウザーでも実行できるようになる。ただし、本稿執筆に際しては「Try it out」ページでコードを入力／実行してその動作を確認するためだけに使用した。

あるいは [Node.js](#) の最近のバージョンでは node コマンドで ECMAScript 2015 のコードを実行可能だ (Node.js のメジャーバージョンごとに ECMAScript 2015 のサポートの度合いは異なるので注意が必要だ。例えば、筆者が試したところでは、クラス定義の構文を含んだコードは 6.6.0 では実行可能だったが、4.6.1 LTS では実行できなかった)。



TypeScript と ECMAScript 2015 を Web ページ上で試してみる

上は TypeScript の Playground ページ。下は Babel の Try it out ページだ。Babel の方では右下に出力結果が表示されているのが分かる。

なお、自分の PC に Node.js がインストールされているのであれば、「npm install -g typescript」コマンド（TypeScript の場合）や「npm install -g babel -cli」コマンド（Babel の場合）などを実行することで、これらをローカルにインストールできる。（ただし、後者についてはお行儀が悪いとされているので注意しよう。また、babel-cli だけではなく、「npm install -save-dev babel-preset-es2015」コマンドなどで必要なプラグインをインストールしておく必要もある。「babel-preset-es2015」は Babel で ECMAScript 2015 コードをトランスパイルするのに必要なプラグイン一式を集めたプリセットだ）。

能書きはこのくらいにして、実際に C# のコードを書き直してみることにしよう。

Hello World

最初に見るのは「Hello World」だ。コードを以下に示す。

```
using System;

namespace cs
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("hello from Program.Main in Program.cs");
            Console.ReadLine();
        }
    }
}
```

C# による Hello World プログラム

見慣れたコードである。まずはこれと同等なプログラムを JavaScript 5 / TypeScript / ECMAScript 2015 で記述してみよう。もちろん、これらはスクリプト言語なので、実際には「console.log("Hello World");」なり「alert("Hello World");」なりの 1 行で済んでしまうのだが、そこはそれ。プログラムの構造も同様なものにしてみよう。

ただし、ここでは「using」による名前空間の導入、「namespace」による名前空間の指定は保留としておく。つまり、JavaScript 5 / TypeScript / ECMAScript 2015 でクラスをどのように定義するかに着目する。

JavaScript 5 でのクラス定義

では、JavaScript 5 でのクラス定義から見てみよう。厳密には、JavaScript はクラスベースのオブジェクト指向プログラミング言語ではなく、プロトタイプを利用してクラスと同様な機構を実現する。以下で行っているのはクラス定義「的」な処理だ。

```
var Program = (function() {    // 即実行関数の実行結果を変数 Program に代入
    function Program() {}    // コンストラクターの定義
    Program.Main = function() { // 静的メソッドの定義
        console.log("hello from Program.Main");
    };
    return Program;           // コンストラクターが戻り値
})();                         // 即時実行関数を実行

Program.Main();
```

JavaScript 5 でのクラス定義

JavaScript 5 でプログラムを記述している方には説明の必要もないだろうが、ざっくりと解説をしておこう。

大枠ではこのコードは「var Program = (function() { ... })();」となっている。これはかつて囲まれた関数（無名関数）を実行した結果を変数 Program に代入するものだ（このようにかっこで囲まれて定義され、最後にかっこを付けて、呼び出される形式の関数のことを「即時実行関数」と呼ぶ）。そして、無名関数の内部では何もしない関数 Program を定義し、その関数の Main プロパティにコンソールに出力を行う関数をセットして、最後に関数 Program を戻り値としている。よって、変数 Program には戻り値となった関数 Program が代入される。JavaScript 5 では、このような記述をクラス定義の代わりとして使える。

即時実行関数内で定義されている関数 Program は、C# におけるコンストラクターとして機能する。ところで、JavaScript では関数もオブジェクトであり、プロパティを設定できる。これを利用して、「Program」を名前空間のように扱い、そこに静的メソッド（スタティックメソッド／クラスメソッド）を定義しているのが、「Program.Main = ...」という部分になる。

Program.Main 関数は静的メソッドなのでインスタンスがなくても呼び出せる。これが最後の「Program.Main()」呼び出しだ。

このように、JavaScript 5 でクラスベースのオブジェクト指向プログラミングに近いことをやろうとすると、C# に慣れた人にとってはかなり面倒なことになる。そこで、TypeScript や ECMAScript 2015 では、クラスベー

スのオブジェクト指向プログラミングも行えるように構文が拡張されたのである。というわけで、次にこれらの言語でクラス定義がどのようにになるかを見てみよう。

ここで注意しておきたいのは、TypeScript にしても ECMAScript 2015 にしても、従来の JavaScript に対してクラスを利用したプログラミングも可能になるような構文が拡張されたものであり、JavaScript の本質には変わりはないという点だ。つまり、プロトタイプを基にオブジェクトを生成し、個々のオブジェクトを操作するというスタイルには本来的には変わりはない。

TypeScript / ECMAScript 2015 でのクラス定義

まずは TypeScript から見てみよう。冒頭でも述べたように、TypeScript で拡張された大きな機能はクラス定義にまつわる部分と、静的な型指定（型注釈）が可能になった点である。

```
class Program {           // クラス定義
    constructor() {}      // コンストラクターの名前は「constructor」
    static Main():void {  // 静的メソッドの定義。戻り値は void
        console.log("hello from Program.Main");
    }
}

Program.Main();
```

TypeScript でのクラス定義の例

C# プログラマーにもすっきりと分かりやすいはずだ。気になるのは「static Main():void」という部分くらいだろう。「static」は C# の「static」と同じでこれが C# でいうところの静的メソッド（クラスメソッド）であることを意味する。また、「:void」という部分は「型注釈」と呼ばれるもので、この場合は Main メソッドの戻り値の型が「void」であることを示す。つまり、C# で「static void Main」と記述したのと同様ということだ。その他にはコンストラクターの名前が「constructor」となっていることは覚えておこう。

なお、TypeScript では型注釈がサポートされているが、ECMAScript 2015 にはこれは含まれていない。

次に ECMAScript 2015 におけるクラス定義を見てみよう。

```
class Program {    // クラス定義
    constructor() {} // コンストラクターは「constructor」
    static Main() { // 静的メソッドの定義。型注釈はない
        console.log("hello from Program.Main");
    }
}

Program.Main();
```

ECMAScript 2015 でのクラス定義の例

型注釈が ECMAScript 2015 にはない点を除けば TypeScript と同様だ。特に説明を追加する必要はないだろう。

以上で最初に見た C# プログラムが JavaScript 5 / TypeScript / ECMAScript 2015 でどうなるかが分かったはずだ。TypeScript / ECMAScript 2015 では、C# プログラマーにとって直感的に理解しやすいコードが書ける。ただし、このプログラムはあまりにもシンプルだったので、次にインスタンス変数やインスタンスメソッドを持ったクラスをどう記述するかを見てみよう。

インスタンスメンバーを持つクラス

ここでは C# の以下のようなプログラムを JavaScript 5 / TypeScript / ECMAScript 2015 に書き換えてみよう。

```
using System;

namespace cs
{
    public class Foo {
        private string msg;           // インスタンス変数の宣言
        public Foo(string msg = "foo") { // デフォルト引数
            this.msg = msg;           // インスタンス変数の初期化
        }
        public void Hello() {         // インスタンスメソッド
            Console.WriteLine("hello " + msg);
        }
    }
}
```

```
}

public class Program
{
    public static void Main(string[] args)
    {
        var f = new Foo("world");
        f.Hello();
    }
}
}
```

少し複雑になった C# プログラム

ここでは、Program クラスとは別に Foo クラスを定義し、Main メソッドでは Foo クラスのインスタンスを利用するようにした。注目してほしいのはあくまでも Foo クラスの中身だ。Foo クラスではプライベートなインスタンス変数 msg と、パブリックなインスタンスメソッド Hello を宣言している。コンストラクターにはデフォルト引数が指定してあり、引数なしでコンストラクターを呼び出すとインスタンス変数 msg の値は "foo" に設定される。

Program.Main メソッドでは、Foo クラスのインスタンスを生成して、そのインスタンスに対して Hello メソッド呼び出しを行っているだけだ。

JavaScript 5 での記述

JavaScript 5 では、どんなコードになるだろうか。

```
var Foo = (function() {
    function Foo(msg) {
        this.msg = msg === void 0 ? "foo" : msg; // インスタンスプロパティの初期化
    }
    Foo.prototype.Hello = function() { // インスタンスメソッドの定義
        console.log("hello " + this.msg);
    }
    return Foo;
})();

var f = new Foo("world");
```

```
f.Hello();
```

インスタンスプロパティとインスタンスメソッドを持つオブジェクトの定義

最後の 2 行は、C# 版の Main メソッドで行っていた処理を外出しにしたものだ。特に説明の必要はないだろう。では、Foo クラスの定義の内容を見ていこう。

インスタンスプロパティとデフォルト引数

まず注目したいのは「this.msg = msg === void 0 ? "foo" : msg;」という部分だ。

C# 版のコードではインスタンス変数 msg を「string msg;」として宣言して、その後、コンストラクター内部で「this.msg = msg;」として初期値を代入していたが、JavaScript 5 ではこのような形でインスタンスプロパティ（インスタンス変数）を宣言することはない。コンストラクター（Foo 関数）の中で直接「this.msg = ~」のようにして、個々のインスタンスが持つプロパティを初期化していけばよい。

この「this」は、new 演算子とともに呼び出されたコンストラクター内部では新規に作成されたオブジェクトを、他の関数では呼び出しに使用されたオブジェクトを参照する。そのため、ここでは「new Foo(...)」として呼び出され、新規に作成されたオブジェクトに「msg」という名前のプロパティを追加することになる。なお、JavaScript ではオブジェクト自身が持つメンバーを参照する際には「this」が必須である。

以上で「this.msg =」までの説明は終わりだが、まだ「msg === void 0 ? "foo" : msg」という部分が残っている、C# 版のコードでは「Foo(string msg = "foo")」のようにしてデフォルト引数が指定されていた。JavaScript 5 にはデフォルト引数がないので、これをエミュレートしているのがこの部分だ（TypeScript / ECMAScript 2015 にはデフォルト引数が追加されている）。

「new Foo()」と Foo コンストラクターを無引数で呼び出すと msg パラメーターの値は「undefined」となる。そこで、このコードでは「void 0 が常に undefined となる」性質と三項演算子を組み合わせて、デフォルト引数のエミュレートを行っている ***2**。

***2** JavaScript コードに記述する「undefined」はグローバルな変数であり、その値は「プリミティブ値の undefined」である。同時に「undefined」は予約語ではないので、コード中で識別子として利用できる。そのため、「var undefined = ...」などとした場合には、もともとの undefined の値が隠ぺいされてしまう（詳細は「[undefined](#)」ページなどを参照されたい）。一方、「void 0」は常に「undefined」となる（void 演算子は引数を一つ取り、常に undefined を返す）ので、この値を msg パラメーターと比較し、その値が「undefined」かどうかを安全に判定している。そして、引数なしでコンストラクターが呼び出されたら、デフォルト引数の値である "foo" を this.msg に代入しているの

である。

インスタンスメソッドとプロトタイプ

「`Foo.prototype.Hello = function() {...}`」は個々のインスタンスに対して呼び出しを行うインスタンスメソッドを定義している。先ほど、Main メソッドを定義したときには「`Program.Main = function() {...}`」のようにしたのに対して、こちらでは「`prototype`」というキーワードが増えている。

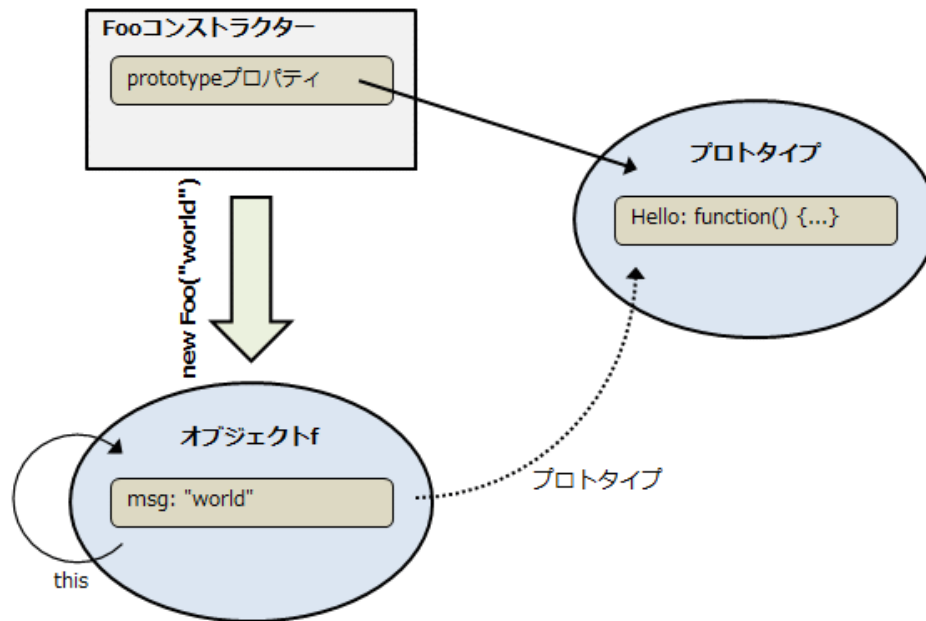
先ほども述べたように、JavaScript では全てのオブジェクトは何らかのプロトタイプを基に生み出される。そして、上のコードに出てきた `prototype` こそが、JavaScript のオブジェクト生成の源であるプロトタイプを参照するものであり、以降で説明するクラスの継承などでも重要な役割を果たす。

JavaScript では、全てのオブジェクトはそのプロトタイプを持つ（`__proto__` プロパティ）。特に何も指定しなければ `Object` 型のオブジェクトが、そのプロトタイプとなる。そして、プロトタイプを同じくするオブジェクトは全て、そのプロトタイプが持つ属性を共有する（「よって、通常のオブジェクトは『`toString`』メソッドなどの `Object` 型のインスタンスメソッドを共通に呼び出せる」と書けば、何となく .NET の世界との類似性が感じられるのではなかろうか）。

一方、コンストラクターは `prototype` プロパティを持つ。そして、そのコンストラクターを利用して作成された全てのオブジェクトは、コンストラクターの `prototype` プロパティが参照するプロトタイプを共有するように作成されるのである。

よって、上のコードの「`Foo.prototype.Hello = function() {...}`」は「`Foo` コンストラクターのプロトタイプの `Hello` プロパティに関数を設定する」ことになる（その値が関数となっているプロパティのことを「メソッド」と呼ぶ）。そして、`Foo` コンストラクターを利用して作成されたオブジェクトを利用して、その `Hello` メソッドを呼び出せるのだ。

と文字で説明しても分かりにくいので、上のコードで行っている二つの処理を図にしてみよう。



Foo コンストラクターと、それによって作成されるオブジェクト、そのプロトタイプ

とまあ、JavaScript 5 のコードはやはり C# プログラマーには難解なところがあるかもしれない。では、TypeScript / ECMAScript 2015 ではどうなるだろう。

TypeScript / ECMAScript 2015 での記述

TypeScript のコードは以下ようになる。

```
class Foo {  
  private msg:string;           // プライベートなプロパティの宣言  
  constructor(msg:string = "foo") { // デフォルト引数  
    this.msg = msg;             // プロパティの初期化  
  }  
  public Hello():void {         // インスタンスメソッド  
    console.log("hello " + this.msg);  
  }  
}  
  
var f = new Foo("world");  
f.Hello();
```

インスタンスプロパティとインスタンスメソッドを持つオブジェクトの定義 (TypeScript)

C# プログラマーには非常に分かりやすい形で記述できる。

まず「constructor(msg:string = "foo")」とデフォルト引数を指定できることが分かる。もう一つ注意しておきたいのは「private msg:string;」行だ。TypeScript ではプライベートなプロパティを定義できるのだ。ただし、これはあくまでも仕様上の話である。TypeScript コードは最終的に JavaScript コードに変換されるが、前述した通り、JavaScript ではプライベートなプロパティは存在しない（クロージャを使うことで、疑似的に隠ぺいすることは可能だ）。上記のコードを JavaScript 5 にコンパイルしたものを見てみよう。

```
var Foo = (function () {  
    function Foo(msg) {  
        if (msg === void 0) { msg = "foo"; }  
        this.msg=msg; // 生成されたコードでは自由に msg プロパティを使用可能  
    }  
    Foo.prototype.Hello = function () {  
        console.log("hello " + this.msg);  
    };  
    return Foo;  
})();  
var f = new Foo("world");  
f.Hello();
```

コンパイル後のコード (JavaScript 5)

先ほど見た JavaScript 5 版のコードとほぼ同じものが出来上がっている（違いは、三項演算子ではなく if 文を使ってデフォルト引数をエミュレートしているところくらいだ）。そして、TypeScript のコードで指定した「private」という型注釈（に関連した情報）は一切ない。型注釈はあくまでも TypeScript でコードを記述する際の支援だと思えるのがよい。JavaScript へのコンパイル時にデータ型に関連したエラーが発生するのであれば、それは静的なコード解析で問題があることを教えてくれている。よって、そこを修正することで、JavaScript に変換された後でもデータ型の取り扱いに関して整合性が保証されるということだ。これは Hello メソッドに付加した「public」についても同様だ。

次に ECMAScript 2015 のコードを見てみよう。TypeScript のコードとほぼ同様なことが期待される（型注釈を除く）。

```
class Foo {  
  constructor(msg = "foo") {  
    this.msg = msg; // プロパティの初期化  
  }  
  
  Hello() { // インスタンスメソッド  
    console.log("hello " + this.msg);  
  }  
}  
  
var f = new Foo("world");  
f.Hello();
```

ECMAScript 2015 の場合のインスタンスプロパティとインスタンスメソッドを持つオブジェクトの定義

TypeScript のコードとほぼ同様だが、型注釈以外に異なる点がある。それは、TypeScript にはあった「private msg:string;」行に相当する行がないことだ。実は ECMAScript 2015 では、このようなプロパティは定義できない。JavaScript 5 のコードと同様に、コンストラクターの中でプロパティを適宜設定していただく。

なお、TypeScript と ECMAScript 2015 でのコーディングの違いとして、TypeScript ではコンストラクターのパラメーター指定で直接プロパティに値を代入できる点がある。これを「パラメータープロパティ宣言」と呼ぶ。以下に例を示す。

```
class Foo {  
  // private msg:string; ← プロパティの宣言を削除  
  constructor(private msg:string = "foo") { // コンストラクターの引数でプロパティを宣言  
    // this.msg = msg; ← コンストラクター内での初期化は不要  
  }  
  .....省略.....  
}
```

パラメータープロパティ宣言

上のコードのようにコンストラクターのパラメーターにアクセス修飾子を付加することで、クラス定義内でのプロパティの定義や、コンストラクター内部でのプロパティへのパラメーター値の代入などを記述することなく、プロパティの宣言と初期化が可能だ。このコードではデフォルト値を指定しているので初期化は必ず行われるが、デフォルト値を指定していない場合に引数を指定せずにコンストラクターを呼び出すと、そのプロパティの値は「undefined」となる（コンパイル時にエラーも発生する）。デフォルト引数と組み合わせて使うのが良策だと思われる。プロパティがたくさんある場合には、パラメータープロパティ宣言を使うことでコーディング量を大幅に削減できるだろう。

では次に、クラスの継承について見てみよう。

クラスの継承

C# 版のコードを以下に示す。コードをシンプルにするために、メソッドの定義、デフォルト引数の指定などは行っていない。

```
using System;

namespace cs
{
    public class BASE
    {
        public string bprop; // BASE クラスのインスタンス変数
        public BASE(string bprop)
        {
            this.bprop = bprop;
        }
    }

    public class DERIVED : BASE
    {
        public string dprop; // DERIVED クラスのインスタンス変数
        public DERIVED(string bprop, string dprop) : base(bprop)
        {
            // ↑ base() 呼び出しで基底クラスのメンバーを初期化
            this.dprop = dprop;
        }
    }
}
```

```
}

public class Program
{
    public static void Main(string[] args)
    {
        var b = new BASE("base1");
        var d = new DERIVED("base2", "derived");
        Console.WriteLine(b.bprop);
        Console.WriteLine(d.bprop);
        Console.WriteLine(d.dprop);
    }
}
}
```

クラス継承を行うコード (C#)

このコードも特に説明の必要はないだろう。BASE クラスが基底クラス、DERIVED クラスが派生クラスで、Main メソッドでは両者のオブジェクトを生成して、それぞれのメソッドを呼び出しているだけだ。

JavaScript 5 での記述

例によって JavaScript 5 での記述例を見てみよう。

```
var BASE = (function() {
    function BASE(bprop) {
        this.bprop = bprop;
    }
    return BASE;
})();

var DERIVED = (function() {
    function DERIVED(bprop, dprop) {
        BASE.call(this, bprop); // 基底クラスのコンストラクター呼び出し
        this.dprop = dprop;
    }
})
```

```
    return DERIVED;  
  })();  
  
  // 継承関係の確立: DERIVED.prototype のプロトタイプを BASE.prototype に設定  
  Object.setPrototypeOf(DERIVED.prototype, BASE.prototype);  
  
  var b = new BASE("base1");  
  var d = new DERIVED("base2", "derived");  
  console.log(b.bprop);  
  console.log(d.bprop);  
  console.log(d.dprop);
```

クラスの継承 (JavaScript 5)

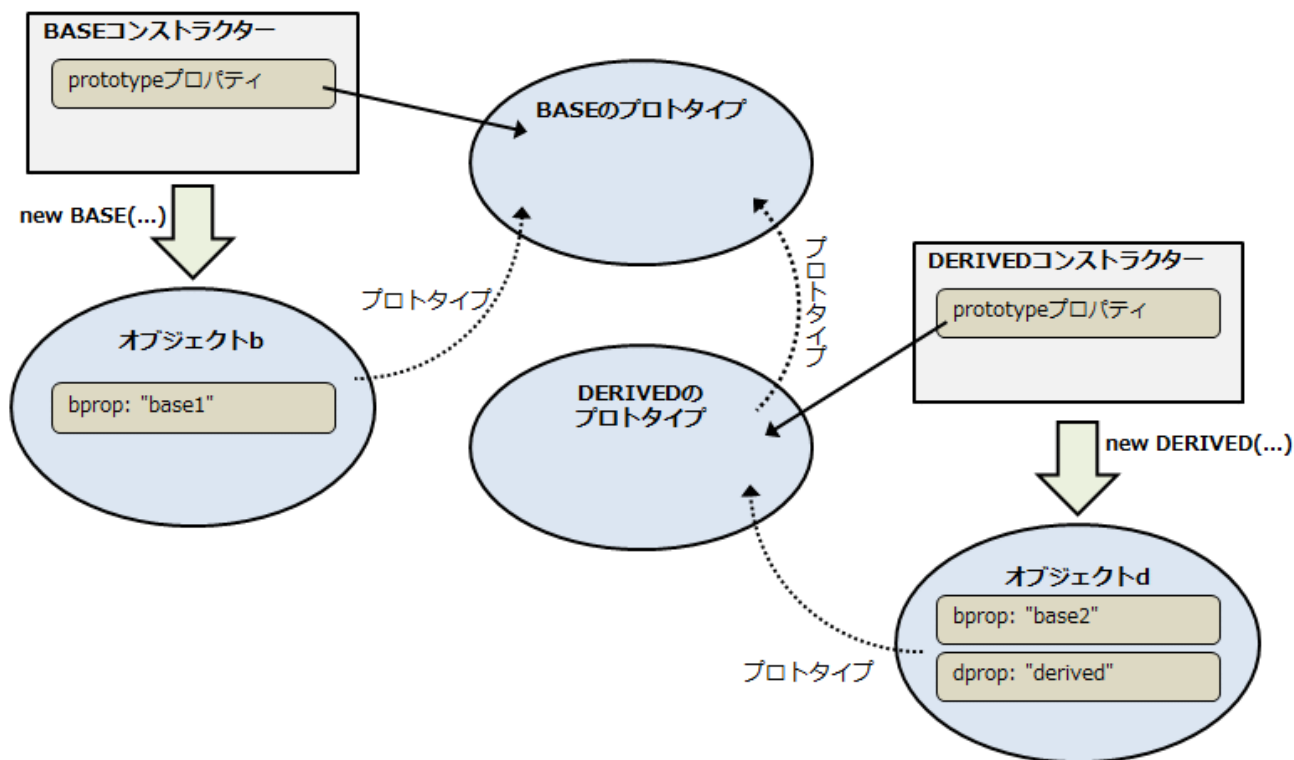
ここではメソッドを定義しておらず「クラス名.prototype.メソッド名 = function() {...}」という行がなくなっているため、即時実行関数で囲む必要はないのだが、ここまでのサンプルと同様な記述をしている。

まず、C# 版のコードでは、派生クラスのコンストラクターを定義する際に、基底クラスのメンバーを初期化するために「public DERIVED(string bprop, string dprop) : base(bprop)」として基底クラスのコンストラクターを呼び出していた。これに対して、JavaScript 5 版のコードではコンストラクター内部で「BASE.call(this, BASE コンストラクターに渡す引数)」(または「BASE.apply(this, BASE コンストラクターに渡す引数)」)という形で呼び出すことにも注意しておこう。

また、C# では上のコードのように明示的にコンストラクターを呼び出さない場合には、最初に基底クラスの無引数のコンストラクターが呼び出されるが、JavaScript 5 で上記のようなコードを書いた場合、常に自分でコンストラクター呼び出しを行う必要がある(Javascript 5 にはクラス機構が用意されていてそうした処理を自動的にしてくれるわけではなく、クラスの継承に相当する処理を自力で書いているだけだからだ)。

次に C# ではクラスを継承する際には「class 派生クラス名 : 既定クラス名」という記述をするが、JavaScript 5 では先ほど出てきたプロトタイプのすげ替えを行うことで、継承に相当する処理を行う。これを行っているのが「Object.setPrototypeOf(DERIVED.prototype, BASE.prototype);」という行だ。これは DERIVED コンストラクターのプロトタイプのプロトタイプ (DERIVED.prototype.__proto__ プロパティ) を、コンストラクター BASE のプロトタイプにしている。

BASE と DERIVED の二つのコンストラクター、それらを利用して作成されるオブジェクト、プロトタイプの間を以下に示す。



プロトタイプをチェーンさせることで継承関係を構築する

図を見ると分かるように、DERIVED.prototype のプロトタイプを BASE.prototype が参照するオブジェクトにすることで、「BASE が親で DERIVED が子」という継承関係が確立される。このように JavaScript ではプロトタイプをチェーンさせることで、クラスの継承階層をさかのぼりながらプロパティやメソッドの検索を行うようになっている（ただし、この例ではインスタンスメソッドなどは定義していないので、図には現れていない）。このこと自体は ECMAScript 2015 でも変わらないし、TypeScript でもそうした操作はもちろん可能だ。

TypeScript / ECMAScript 2015 での記述

では、TypeScript と ECMAScript 2015 でクラス継承がどのように記述できるかを見てみよう。まずは TypeScript からだ。

```
class BASE {
  bprop:string; // インスタンスのプロパティの宣言
  constructor(bprop:string) {
    this.bprop = bprop;
  }
}

class DERIVED extends BASE { // 継承を行うには extends キーワードを使用する
  dprop:string; // インスタンスのプロパティの宣言
```

```
constructor(bprop:string, dprop:string) {  
    super(bprop);           // 基底クラスのコンストラクター呼び出し  
    this.dprop = dprop;  
}  
}  
  
var b = new BASE("base1");  
var d = new DERIVED("base2", "derived");  
console.log(b.bprop);  
console.log(d.bprop);  
console.log(d.dprop);
```

クラスの継承 (TypeScript)

C# で派生クラスを定義するには「class 派生クラス : 基底クラス」と派生クラスと基底クラスで「:」をサンドイッチするような形で継承関係を記述していたが、TypeScript では extends キーワードを使用する。

また、コンストラクターで基底クラスのコンストラクターを呼び出す方法も C# とは異なっている。TypeScript ではコンストラクター内部で「super」メソッドを呼び出して、基底クラスのインスタンスの初期化に必要な情報を引き渡す (TypeScript でも ECMAScript 2015 でも super メソッド呼び出しは必須である)。

JavaScript 5 版のコードとは異なり、Object.setPrototypeOf メソッドを明示的に呼び出す必要はないことに注意しよう。継承にまつわる処理を隠ぺいしてくれるのが TypeScript のよいところだ (ただし、コンパイル後のコードを見ると、予想以上に複雑な処理をしている。興味のある方は調べてみてほしい)。

ECMAScript 2015 のコードは、例によって、TypeScript バージョンとほぼ同様な。

```
class BASE {  
    constructor(bprop) {  
        this.bprop = bprop;  
    }  
}  
  
class DERIVED extends BASE {  
    constructor(bprop, dprop) {  
        super(bprop);           // 基底クラスのコンストラクター呼び出し  
    }  
}
```



```
        this.dprop = dprop;           // プロパティの初期化（宣言がないことに注意）
    }
}

var b = new BASE("base1");
var d = new DERIVED("base2", "derived");
console.log(b.bprop);
console.log(d.bprop);
console.log(d.dprop);
```

クラスの継承 (ECMAScript 2015)

ここまでに見てきたように、クラスベースのオブジェクト指向プログラミングに慣れている開発者にとっては、TypeScript や ECMAScript 2015 は極めてとっつきやすい言語だといえる。

ここでは C# / JavaScript 5 / TypeScript / ECMAScript 2015 でクラス定義がどんなものになるかを比較した。とはいえ、本来の趣旨ではなさそうな JavaScript コードの説明が増えたのは、TypeScript や ECMAScript 2015 では表面的にはクラスを使うことですっきりと分かりやすいコードが記述できるが、内部的にはそういうことをしているのだということを知ることが JavaScript、プロトタイプベースのオブジェクト指向プログラミングを理解する上でも重要だからだ。

● C# × JavaScript

C# 開発者のための最新 JavaScript 事情（関数編）

Insider.NET 編集部 かわさきしんじ（2015 年 11 月 06 日）

コードを書きながら、C# と JavaScript における関数の違いを比較し、C# プログラマーが注意すべき点などを見ていこう。

ここまでは C# におけるクラス定義が、JavaScript 5 ***1** / TypeScript / ECMAScript 2015 ***2** のそれぞれでどのように変わるかを見た。本章では C# における関数と、最新の JavaScript 系言語とを比較しながら、その差について見てみよう。なお、以下では単に「JavaScript」と表記した場合には JavaScript 系の三つの言語を意味する。JavaScript 5.1 を対象とする場合には「JavaScript 5」と書くことにする。

***1** 現在、広く使われている JavaScript のバージョンは 5.1 である。ここでいう「JavaScript 5」とは Ecma international が策定するスクリプト言語仕様である ECMAScript 5.1 に準拠して、各ベンダーが実装しているスクリプト言語実装／実行環境の総称を指す。

***2** 2015 年 6 月に久しぶりのメジャーバージョンアップとして ECMAScript 2015 が策定された（なお、このバージョンの ECMAScript は「ECMAScript 6」あるいは略称として「ES6」などと表記されることもよくある）。ECMAScript 2015 は、JavaScript 5 に対してクラスベースのオブジェクト指向プログラミングを可能にする構文拡張、アロー関数、2 進／8 進リテラル、デフォルト引数、可変長引数などの機能追加が行われている。また、2016 年 6 月には次バージョンとなる ECMAScript 2016 も策定されている。

関数とは何か

もちろん、関数とは「何らかの入力を受け取り、何らかの出力を返す」ものだ。そして、プログラム中の複数カ所に登場して同じ処理をするコードを、一つにまとめることで保守性を高めたり、意味のあるひとまとまりのコードに名前を付けることでコードの可読性を高めたりするために関数は使われる。

ただし、C# にしても、JavaScript にしても「関数」的な振る舞いをするコードにはさまざまな種類がある。前章でも出てきた C# のコンストラクターやクラスメソッド、インスタンスメソッドなどはその代表だ。それらの全てを取り上げて、C# と JavaScript の比較をするわけにもいかないなので、ここではいくつかを取り上げて、C# と JavaScript それぞれの特徴を考えてみよう。

実際のコーディングで気になることはあまりないかもしれないが、一つ覚えておきたいのは「JavaScript にお

ける関数とメソッド」だ。「プロパティの値となっている関数のこと」をメソッドと呼ぶ（そして、メソッドの呼び出しに使われたオブジェクトが「this」としてメソッドに渡される）。

以下では、メソッドと関数を区別する必要がなければ、これらを総称して「関数」と表記する。

一方、C# では関数は多くの場合、何らかのクラスに所属するメソッドとして記述する。こちらもメソッドと関数を区別する必要がなければ「関数」と表記する。

JavaScript における関数定義の概要

JavaScript ではいくつかの方法で関数を定義できる。以下では、C# で似た処理を行うコードと並記していく。なお、ここではコードがシンプルになるように、C# のコードは Program クラスの静的メソッドとして記述することにする。

関数宣言

まず関数宣言を使う方法だ。これは JavaScript 5 / TypeScript / ECMAScript 2015 のいずれでも可能だ。

```
function 関数名 ( パラメーターリスト ) {  
    関数本体  
}
```

関数宣言

これにより、指定した関数名でその関数を呼び出せるようになる。以下に例を示す（TypeScript では型注釈を付加できるが、ここでは TypeScript のコードは割愛する）。

```
function Hello(s) {  
    console.log("hello " + s);  
}  
  
Hello("JavaScript");
```

関数宣言の例

これに近いのが C# でのオーソドックスな静的メソッド定義だろう（以降のコードでは using による名前空間の導入は省略している）。

```
namespace cs
{
    class Program
    {
        static void Hello(string s)
        {
            Console.WriteLine("Hello " + s);
        }

        static void Main(string[] args)
        {
            Hello("C#");

            Console.ReadKey();
        }
    }
}
```

C# での静的メソッド（クラスメソッド）定義

クラスのインスタンスを必要とせずに、関数名とパラメーターを指定するだけで呼び出し可能だ。JavaScript 5 / ECMAScript 2015 では型注釈が使えないので、関数の戻り値やパラメーターに対する型指定がないが、それ以外はほぼ同様な。これについてはあまり述べることはない。

関数式

次に関数式を使う方法を見てみよう。以下は関数式の構文である。

```
function (パラメーターリスト) {
    関数本体
}
```

関数式

関数式では関数名を付けることも可能だが、ここでは省略している。

関数式による関数定義では名前のない関数（無名関数／匿名関数）が作られるので、これを実際に利用するには、変数やプロパティに代入するか、即時実行する必要がある。

```
var Hello = function(s) {    // 関数式で定義した無名関数を変数 Hello に代入
    console.log("hello " + s);
};

Hello("JavaScript");          // 変数を介して関数を呼び出し

(function (s) {              // 無名関数を即時実行
    console.log("hello " + s);
})("function expression");
```

関数式を使った関数定義の例

構文的に似ているものとしては、C# にも匿名メソッドと呼ばれる機構がある。これとデリゲートを使うことで、変数にメソッドを代入できる。ただし、現在、C# の世界では匿名メソッドはあまり使われず、次に説明するラムダ式を使うのが一般的だ。対して、JavaScript の世界では、プロパティへの関数の代入などで、関数式／無名関数を使うのはよくある。というわけで、構文的に似ているからといって比較するのもどうかと思うのだが、一応、コード例を示しておこう。

```
namespace cs
{
    class Program
    {
        static void Main(string[] args)
        {
            Action<string> Hello = delegate (string s)
            {
                Console.WriteLine("hello " + s);
            };

            Hello("C#");

            Console.ReadKey();
        }
    }
}
```

C# は静的型付けを持つので、匿名メソッドのパラメーター／戻り値の型を指定するデリゲート `Action<T>` を使って、その型の変数に匿名メソッドを代入し、その変数を介してメソッドを呼び出している（`Action<T>` は引数を持つ、戻り値を返さない関数を表すデリゲート）。

アロー関数

TypeScript と ECMAScript 2015 では、アロー関数と呼ばれる関数も定義できる。上でも述べているように、これは C# におけるラムダ式に相当する。かっこの中には関数が受け取るパラメーターのリストを、「`=>`」の後ろに関数の本体を記述する。

```
( パラメーターリスト ) => { 関数本体 }
```

アロー関数の構文

パラメーターが一つの場合はかっこを、関数本体が単文の場合は波かっこを省略できる。パラメーターがないときにはかっこは省略できない（「`() => ...`」のようにかっこだけを記述する）。

以下の例は ECMAScript 2015 のものだ。

```
var Hello = (s) => { console.log("Hello " + s); };

Hello("arrow function ");
```

アロー関数を使った関数定義の例その 1 (ECMAScript 2015)

パラメーターが一つだけならかっこは省略でき、関数本体が単文なら波かっこを省略できる（関数本体が `return` 文だけなら「`return`」キーワードの記述も省略できる）。よって、これは以下のようにも記述できる。なお、パラメーターがない場合にはかっこは省略できず「`() => ...`」のように記述する必要がある。

```
var Hello = s => console.log("Hello " + s);

Hello("arrow function");
```

アロー関数を使った関数定義の例その 2 (ECMAScript 2015)

TypeScript では型注釈を付加できるので、例えば、以下のような定義になる。ECMAScript 2015 と同様、関数本体が 1 行だけであれば波かっこは省略可能だ。ただし、型注釈を付加した場合には、パラメーターが一つだけであってもかっこは省略できない。

```
var Hello = (s: string) => alert("Hello " + s);  
  
Hello("TypeScript");
```

アロー関数を使った関数定義の例 (TypeScript)

C# でラムダ式を使ったコードは次のようになる。

```
namespace cs  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Action<string> Hello = s => Console.WriteLine("hello " + s);  
  
            Hello("C#");  
  
            Console.ReadKey();  
        }  
    }  
}
```

ラムダ式を使用したメソッド定義

ラムダ式を使うと型推論機構が働いてくれるため、前述の匿名メソッドのときとは異なり、パラメーターの型を明記する必要がなくなっている。また、TypeScript / ECMAScript 同様、関数本体が単文なのでかっこも必要ない。C# と TypeScript / ECMAScript での記述がそっくりで、(慣れると) とてもシンプルに記述できるのが分かる。

最後に TypeScript では関数のオーバーロードが可能だが、JavaScript / ECMAScript 2015 ではオーバーロードできない。本稿では割愛するが、興味のある方は「[TypeScript で学ぶ JavaScript 入門:第 11 回 関数に関するいくつかのトピック](#)」などを参照してほしい。

さて、ここまで基本中の基本ともいえる関数定義の方法を見てきた。次にもう少し高度なトピックとして JavaScript におけるクラスメンバーとしてのメソッドを見てみよう。

前章のクラス定義の際に取り扱ったのであまり説明することはないのだが、いくつか補足的な事項もある。

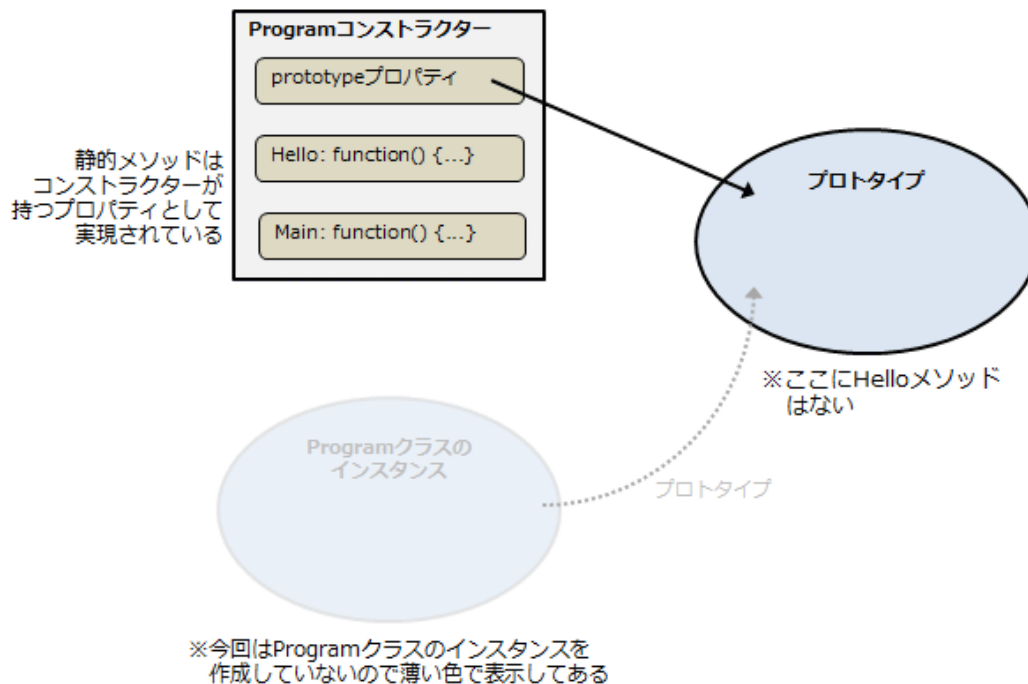
クラスの静的メソッド

まずは最初に見た C# コードを、それと同様な構造を持つように書き直しておこう。まずは JavaScript 5 から。無名関数の即時実行により、変数 Program にコンストラクター関数 Program が設定されているのが分かる（ただし、ここでは Program クラスのオブジェクトは生成していない）。また、二つの静的メソッドも無名関数を使って、Program オブジェクトのプロパティとして設定されている（このような場合に関数宣言を使って、静的メソッドを定義はできない）。

```
var Program = (function() {  
    function Program() {};  
    Program.Hello = function(s) {  
        console.log("hello " + s);  
    };  
    Program.Main = function() {  
        Program.Hello("JavaScript");  
    };  
    return Program;  
})();  
  
Program.Main();
```

JavaScript 5 で C# 版のコードを書き直したもの

ここで重要なのは、静的メソッド Hello の呼び出し方だ。C# のコードでは単に「Hello(...)」としていたが、JavaScript では「クラス名. 静的メソッド名 (パラメーター)」の形で呼び出す必要がある。これは JavaScript でクラスベース（的な）オブジェクト指向プログラミングを行う場合、クラスの静的メソッドはコンストラクター関数（ここでは Program オブジェクト）のプロパティとして実装されるからだ。



クラスの静的メソッド

C# であれば、あるクラスのメソッド（静的メソッド／インスタンスメソッド）から同じクラスの静的メソッドを呼び出すのにクラス名を付加する必要はなかった。しかし、JavaScript では、上の図にあるようにインスタンスメソッドの検索経路（プロトタイプチェーン）上には静的メソッドは存在しない。また、あるクラスの静的メソッドから同じクラスの静的メソッドを呼び出す場合にも、そのメソッドは見えないのでクラス名で修飾してやる必要があるのだ。

このことは、ECMAScript 2015 でも TypeScript でも変わらない。

```
class Program {  
  constructor() {}  
  static Hello(s) {  
    console.log("hello " + s);  
  }  
  static Main() {  
    Program.Hello("ECMAScript 2015");  
  }  
}  
  
Program.Main();
```

```
class Program {  
    constructor() {}  
    static Hello(s: string): void {  
        alert("hello " + s);  
    }  
    static Main():void {  
        Program.Hello("TypeScript");  
    }  
}  
  
Program.Main();
```

ECMAScript 2015 / TypeScript で書き換えたもの

上（前ページ）は [ECMAScript 2015 バージョン](#)。

下は [TypeScript バージョン](#)。

どちらのコードでもクラス名を静的メソッドの前に指定する必要がある。

また、JavaScript 5 バージョンのコードでは無名関数を使っていたが、ECMAScript 2015 / TypeScript ではメソッド宣言を関数宣言と同じ形式で行っているのが分かる。ただし、正式にはこれは、ECMAScript 2015 ではメソッド定義、TypeScript ではメンバー関数宣言と呼ばれる。静的メソッドを定義する構文は次のようになっている（概要）。

```
class クラス名 {  
    static プロパティ名 ( パラメーターリスト ) { 関数本体 }  
    .....  
}
```

```
class クラス名 {  
    アクセス修飾子 static プロパティ名 <型パラメーター> ( パラメーターリスト ) : 戻り値型 {  
        関数本体 }  
    .....  
}
```

ECMAScript 2015 / TypeScript で静的メソッドを定義する構文

TypeScript ではアクセス修飾子 / 型パラメーター / 戻り値型は省略可能。上のコードはアクセス修飾子と型パラメーターを省略している。

なお、前章の「クラス定義」でも述べたが、ECMAScript 2015 ではクラス定義内にフィールド（メンバー変数）の宣言は記述できない。コンストラクターの中で直接、必要なメンバーの初期値を設定していく。これに対して、TypeScript ではこれらを宣言したり、コンストラクターのパラメーターリストを使用したりして、それらを初期化できる。

インスタンスメソッド

インスタンスメソッドは、ご存じの通り、あるクラスのインスタンスを介して呼び出し可能なメソッドだ。

JavaScript 5 だと、C# 開発者にはちょっと奇妙なインスタンスメソッドが定義できる。ということは、ECMAScript 2015 / TypeScript でも同じことができるということだ。なお、前章でも述べたが、JavaScript では自身のオブジェクトのプロパティ（メンバー）を参照するのに「this」が必須である。

```
var Foo = (function() {  
    function Foo(lang) {  
        this.lang = lang;  
        this.SeeYou = function() {  
            console.log("see you " + this.lang);  
        };  
    };  
    Foo.prototype.Hello = function() {  
        console.log("hello " + this.lang);  
    }  
    return Foo;  
})();  
  
var f = new Foo("JavaScript");  
f.Hello();  
f.SeeYou();
```

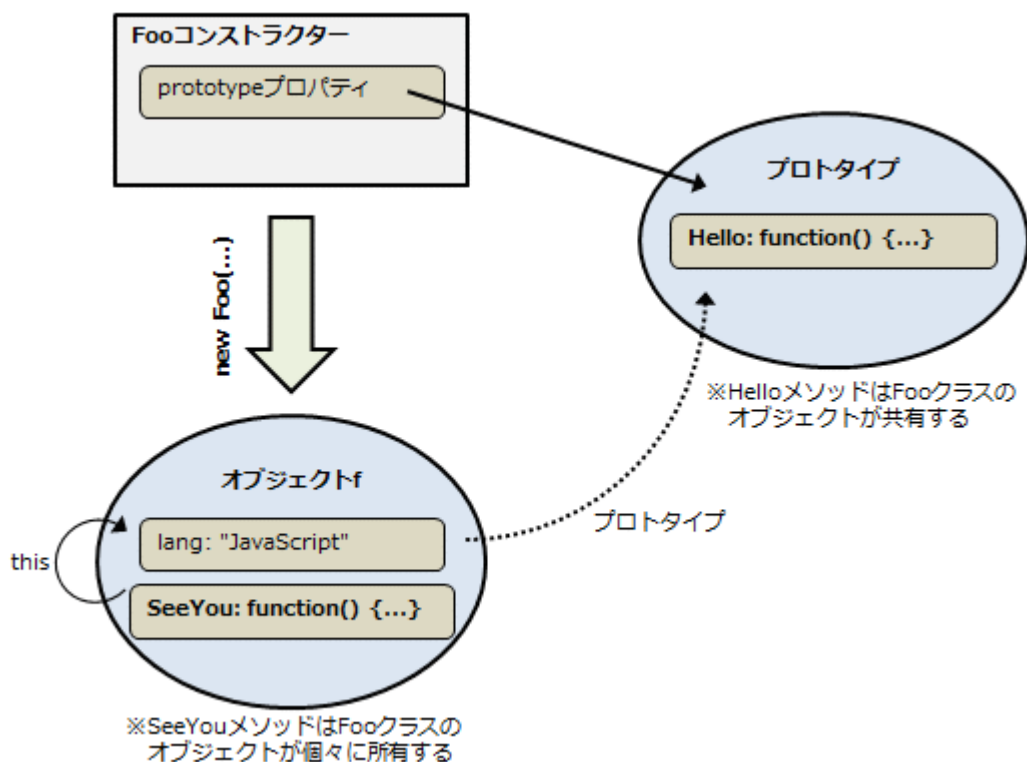
2種類のインスタンスメソッド

強調書体で表示されている部分に注目されたい。

通常、インスタンスメソッドとして考えられているのは、関数 Foo の prototype プロパティに追加されている Hello 関数だ。これは、「new Foo()」として生成されたオブジェクト全てが共有し、プロトタイプチェーンをたどることで呼び出される。また、「new」と共にコンストラクター関数を呼び出すと、インスタンスが新たに生成さ

れ、それがコンストラクターに渡される。よって、この場合は Foo 関数内では this が新たに生成されたインスタンスを参照し、これにより「this. ~」が新規に生成されたオブジェクトのプロパティを参照することになる（この場合はプロパティとメソッドが追加される）。

コンストラクター関数 Foo の中ではプロパティ（フィールド／メンバー変数）に加えて「this.SeeYou = function() { ...}」として「個々のインスタンスが所有する（＝プロパティとして持つ）関数」（インスタンスメソッド）が定義されている。全てのインスタンスが別個にこの関数を所有するためにメモリ使用量の面で不利になるので、通常はこのようなことはしないだろう。ただし、コンストラクター内部でこのようなインスタンスごとにメソッドを持たせるのではなく、何らかの理由から後付けで特定のインスタンスにメソッドを追加できることは覚えておいてもよいかもしれない。



2 種類のインスタンスメソッド

同様なコードをあえて C# で書くと、次のようになるだろう（する必要があるかどうかは別にして）。ここでは、Action デリゲートとラムダ式を組み合わせている。

```
namespace cs
{
    class Foo
    {
        private string lang;
        public Action SeeYou;
```

```
public Foo(string lang)
{
    this.lang = lang;
    this.SeeYou = () => Console.WriteLine("See You " + this.lang);
}

public void Hello()
{
    Console.WriteLine("hello " + this.lang);
}
}

class Program
{
    public static void Main()
    {
        var f = new Foo("C#");
        f.Hello();
        f.SeeYou();

        Console.ReadKey();
    }
}
```

インスタンスごとにメソッドを持つ C# コード

ECMAScript 2015 / TypeScript のコードも示しておこう。最初の JavaScript 5 コードと同様にコンストラクター内部で無名関数を利用して、個々のインスタンスが所有するメソッドを定義できている。

```
class Foo {
    constructor(lang) {
        this.lang = lang;
        this.SeeYou = function() {
            console.log("See You " + this.lang);
        }
    }
}
```

```
    }  
    Hello() {  
        console.log("hello " + this.lang);  
    }  
}
```

```
var f = new Foo("ECMAScript 2015");  
f.Hello();  
f.SeeYou();
```

```
class Foo {  
    private lang: string;  
    public SeeYou: Function;  
    constructor(lang: string) {  
        this.lang = lang;  
        this.SeeYou = function(): void {  
            alert("See You " + this.lang);  
        }  
    }  
    Hello(): void {  
        alert("hello " + this.lang);  
    }  
}
```

```
var f = new Foo("TypeScript");  
f.Hello();  
f.SeeYou();
```

同じことをする ECMAScript 2015 / TypeScript のコード

上は ECMAScript 2015 バージョン。

下は TypeScript バージョン。

次にクローージャの記述を見てみよう。

クローージャ

クローージャとは、自身の関数定義を取り囲む構文スコープに存在する変数などの値を利用できる関数のことだ。

JavaScript では、これを利用して、情報隠ぺいが実現されている。以下に例を示す。

```
function getInc(init) {  
    var start = init;  
    return function() { // 「return () => start++;」 と等価  
        return start++;  
    }  
}  
  
var inc = getInc(0);  
console.log(inc());  
console.log(inc());
```

クロージャの例 (JavaScript)

この例では、getInc 関数は関数を返すが、return 文の前でローカル変数 start を定義している。そして、return 文で返送している関数はこのローカル変数 start の値にアクセスできる。getInc 関数が終了しても、このローカル変数は破棄されずに生存を続けるため、その後で関数が呼び出されてもこの値を利用できる。変数 start の初期値はインクリメントを行う際の初期値となり、返送された関数が呼び出されるたびにその値が 1 ずつ増えていく。

また、変数 start には、getInc 関数とその戻り値である無名関数以外からはアクセスする手段もない。このため、外部からアクセスしてほしくない情報をこのような形で保持することで情報も隠ぺいできる。

このように関数外部の構文スコープで定義されている情報を補足（キャプチャー）して、利用するような関数のことをクロージャと呼ぶ。JavaScript ではこのようなクロージャが活用されている。上記のコード自体は ECMAScript 2015 でも TypeScript でも変わらないので、これらのコードは割愛しよう。

そしてもちろん、C# でもクロージャは実現可能だ。以下に同じことを C# で行う例を示す。

```
namespace cs  
{  
    class Program  
    {  
        static Func<int> getInc(int init)  
        {  
            var start = init;
```

```
        return () => start++;  
    }  
  
    public static void Main()  
    {  
        var inc = getInc(0);  
        Console.WriteLine(inc());  
        Console.WriteLine(inc());  
  
        Console.ReadKey();  
    }  
}  
}
```

C# で記述したクロージャ

上の JavaScript コードでは関数式を使用していたが (JavaScript コードのコメントにもある通り、「return () => start++;」でもよい。このような場合は「return start++;」から「return」の記述を省略して単に「start++;」だけを書けばよいのである)、C# バージョンではラムダ式を使っている。また、getInc 関数の戻り値型である「Func<int>」は引数を取らずに、int 型の戻り値を返すことを意味するデリゲートである。両者を見ると、戻り値型の指定以外はほぼ同様に記述できることが分かる。

だが、両者の間には大きな違いもある。

this の挙動

その大きな違いを検証するために、上のコードを以下のように書き換えてみた (今度はクラス Bar を定義し、そのインスタンスごとにインクリメントの初期値を共有する。あるインスタンスから得たインクリメント関数は、実行するたびに this.start (インスタンスのメンバー「start」) の値を 1 増加させる。かなり作弄的なコードだ)。

```
namespace cs  
{  
    class Bar  
    {  
        private int start;  
        public Bar(int init)
```



```
{
    start = init;
}

public Func<int> getInc()
{
    return () => this.start++;
}
}

class Program
{
    public static void Main()
    {
        var b = new Bar(0);
        var inc = b.getInc();
        Console.WriteLine(inc()); // 0
        Console.WriteLine(inc()); // 1
        var inc2 = b.getInc();    // 新たなインクリメント関数を取得
        Console.WriteLine(inc2()); // 2: カウントは以前の値を引き継ぐ
        Console.WriteLine(inc2()); // 3

        Console.ReadKey();
    }
}
}
```

変更後の C# コード

インクリメント関数を取得するたびに、初期値からカウントが始まってほしいと思うのが普通だが、これはサンプルなのでそういうものだと思ってほしい。

これを JavaScript と ECMAScript 2015 で書き直したものが以下だ。ここではアロー関数を使わずに無名関数を使っている。

```
var Bar = (function() {  
  function Bar(init) {  
    this.start = init;  
  }  
  Bar.prototype.getInc = function() {  
    return function() {  
      return this.start++;  
    }  
  }  
  return Bar;  
})();  
  
class Baz {  
  constructor(init) {  
    this.start = init;  
  }  
  getInc() {  
    return function() {  
      return this.start++;  
    }  
  }  
}  
  
var b1 = new Bar(0);  
var b2 = new Baz(0);  
var inc1 = b1.getInc();  
var inc2 = b2.getInc();  
console.log(inc1());  
console.log(inc2());
```

インスタンスごとに初期値を持つようにしたコードを JavaScript / ECMAScript 2015 で書き直したもの

上にあるのが JavaScript 5 なコード、下のクラス定義が ECMAScript 2015 なコードになる。

これを実行してみると、「Cannot read property 'start' of undefined」と怒られてしまう。JavaScript では、メソッドの呼び出しに使われたオブジェクトが「this」としてそのメソッドに渡される。そのため、取得したインクリメント関数（inc1 関数／ inc2 関数）を単独で実行しても何らかのオブジェクトが this としてインクリメント関

数に渡されることはない（興味ある方は「[Lambdas and using 'this'](#)」ページ（英語）などを参照されたい）。
要するに、上記の書き方では Bar クラスのインスタンスをクローージャの中で利用できないのである。そして、アロー関数ではこれが可能になっている。

```
class Baz {
  constructor(init) {
    this.start = init;
  }
  getInc() {
    return () => this.start++;
  }
}

var b = new Baz(0);
var inc1 = b.getInc();
console.log(inc1()); // 0
console.log(inc1()); // 1
var inc2 = b.getInc();
console.log(inc2()); // 2
console.log(inc2()); // 3
```

アロー関数では「`this`」変数でクローージャ作成時のオブジェクトを参照できる

このように、無名関数とアロー関数では `this` に関連する挙動が異なるので、注意が必要だ。最後に TypeScript 版のコードも示しておこう。メンバー変数の宣言や型注釈以外は上のコードと同様なので説明は省略する。

```
class Baz {
  private start: number;
  constructor(init: number) {
    this.start = init;
  }
  getInc(): Function {
    number () => this.start++;
  }
}
```

```
var b = new Baz(0);
var inc1 = b.getInc();
alert(inc1());
alert(inc1());
var inc2 = b.getInc();
alert(inc2());
alert(inc2());
```

上のコードを TypeScript で書き直したもの

なお、無名関数でも上記と同様な処理を行いたい場合には、一度、this 変数の値をローカル変数にコピーするとよい（これにより、その値が構文スコープに存在するようになるため、クローージャから利用できるようになる）。

```
class Baz {
  constructor(init) {
    this.start = init;
  }
  getInc() {
    var _this = this;
    return function() {
      return _this.start++;
    }
  }
}
```

this 変数の値をローカル変数にコピーするように書き直したコード（ECMAScript 2015）

ここまで関数定義に関連するコードを C# / JavaScript 5 / ECMAScript 2015 / TypeScript を用いて書きながら、C# と JavaScript の差異を見た。ラムダ式／アロー関数による関数記述が C# と ECMAScript 2015 / TypeScript ではほぼ同一になるなど、C# プログラマーには便利な面もある一方で、JavaScript ならではのクセ、this の挙動が C# プログラマーの直感とは異なるなど、注意すべき点もある。

● C# × JavaScript

C# 開発者のための最新 JavaScript 事情（Babel 編）

Insider.NET 編集部 かわさきしんじ（2015 年 12 月 11 日）

C# から少し離れて、ECMAScript 2015 コードを JavaScript 5 コードに変換するツールである Babel の使い方を見ていこう。

C# と JavaScript の話題から少し離れて、Babel を使用して ECMAScript 2015（以下、ES2015）のコードを JavaScript 5.x（以下、JS5）のコードにトランスパイル（コンパイル）する方法について見てみよう。

ECMAScript 2015 のトランスパイル（コンパイル）とは

ES2015 は 2015 年 6 月に策定された。そして、そのもともとの主戦場である Web ブラウザーでもかなりの数の機能が[サポート状況](#)されるようになってきた。その一方で、現在では ES は 1 年ごとに新バージョンの策定が行われるようになっている。そこで策定された新機能があらゆる Web ブラウザーに実装されるまでにはそれなりの時間が必要だ。策定された／策定中の新機能を先取りして使いたい／評価したいときにはどうすればよいか。その方法の 1 つとして考えられたのが、新バージョンのコードを（全ての Web ブラウザーでの動作が見込める）JS5 のコードに変換することだ。

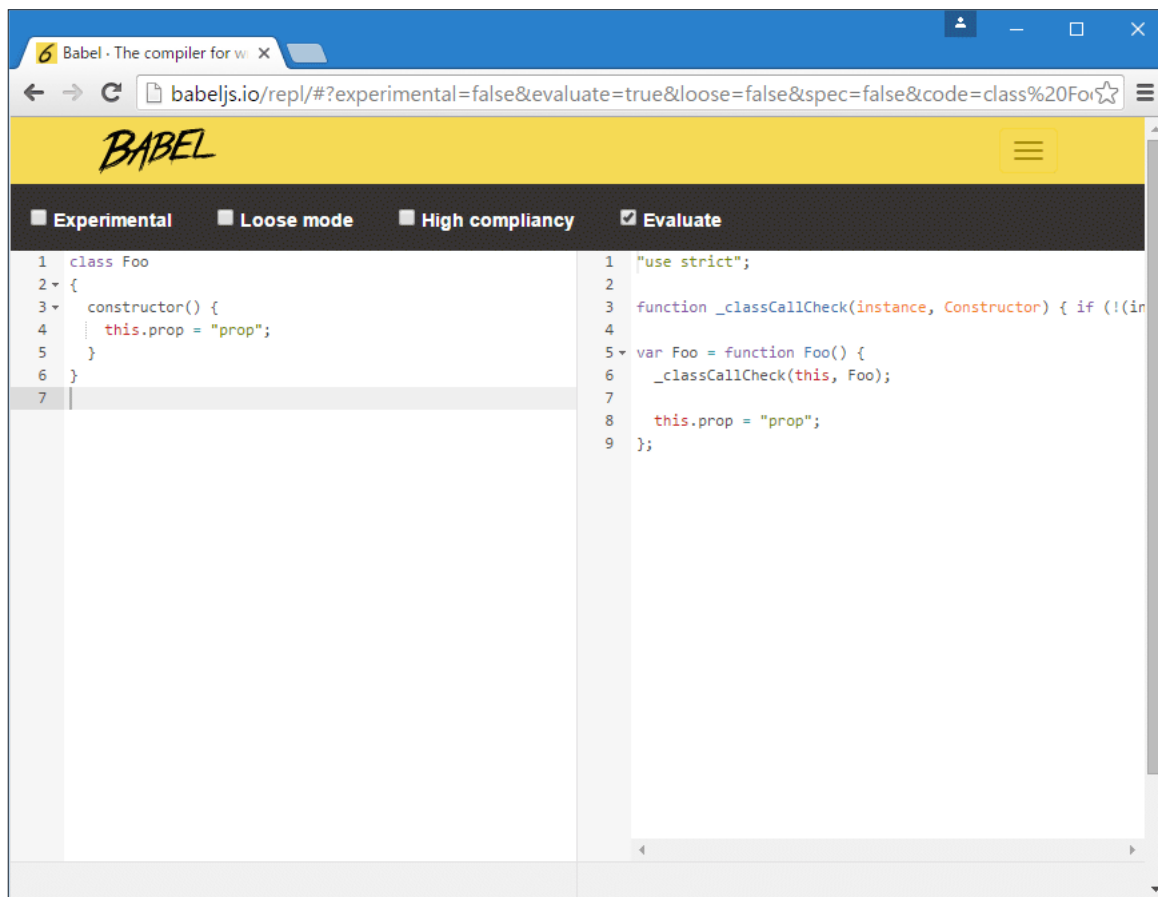
このような特定のプログラミング言語（この場合は ES2015）のソースコードを別のプログラミング言語（この場合は JS5）のソースコードに変換するツールのことを「トランスパイラー」と呼ぶ。ES2015 → JS5 のトランスパイラーとして有名なのが本稿で取り上げる [Babel](#) だ。ES2015 / JS5 トランスパイラーとしては、この他にもグーグルによる [traceur](#) があるが、ここでは取り上げない。

Babel は ES2015 / ES2016 / ES2017（ES2017 は現在策定中の仕様だ）のコードを JS5 にトランスパイルすることで、これらを（完全には）サポートしていない Web ブラウザーでもその機能を使えるようにするものだが、このような技術にはもう一つ「ポリフィル」（polyfill）と呼ばれるものもある。Web アプリ開発の文脈では、ポリフィルは Web ブラウザーがネイティブにはサポートしていない機能を付加（して、例えば ES2015 の機能を利用できるように）するためのライブラリのようなものと考えられる（Babel は下方に向けて互換性を提供するもので、ポリフィルは上方に向けて互換性を追加するものと考えられることができる）。

Babel とは

今も述べたように、Babel は ES → JS5 トランスパイラーであり（ES2015 以外にも上で述べたように ES2016 や ES2017、さらに [React](#) で使われる JSX コードのトランスパイルなども[サポート](#)している）、これを利用するこ

とで、次世代の ES の新機能を実際にコーディングしながら（コマンドラインや Web 上で利用可能な [REPL](#) 環境で*1）試してみたり、ビルド工程に組み込むことで Web アプリ開発に新機能を使用したりできるようになる。なお、本稿では Babel と ES2015 の組み合わせに焦点を当てる。



Babel の「Try it out」ページ

***1** REPL とは「Read（入力）－ Eval（評価）－ Print（出力） Loop（のループ）」の略。Babel の REPL 環境である「Try it out」ページでは、左側のテキストボックスに ES2015 コードを入力すると、それがトランスパイルされて、右側のペーンに表示される（サポートされていない ES2015 の機能もあるので、全てを試せるわけではない）。

手元の PC で Babel を利用するには、npm コマンドを使って、これをインストールする必要がある。ここでは Windows PC を例にインストール方法を説明する。

Babel のインストール

Babel を使用するにはコマンドラインインターフェースと、トランスパイルを行うためのプリセットのインストールが必要だ。

まず、Babel のコマンドラインインターフェースをグローバルにインストールする。ここでは Babel をグロー

バル (-i) にインストールしているが、[Babel のインストールドキュメント](#)では「プロジェクトごとに Babel のどのバージョンに依存するかはまちまちなのでグローバルにインストールすることはおすすめしない」とあるので注意すること（だが、少なくとも筆者の環境だと、上記ページで推奨されている「npm install -save-dev babel-cli」コマンドでは、babel コマンドを実行できるような形でインストールされなかったので、ここでは敢えてグローバルにインストールをしている）。

```
> npm install -g babel-cli
```

Babel のコマンドラインインターフェースのインストール

これにより、「babel」コマンドが使えるようになる。ただし、この状態でトランスパイルを実行すると、例えば、以下のように「えええっ」とビックリすることになる。

```
> type foo.es6 ..... トランスパイルしたいファイルの内容
class F00
{
  constructor() {
    this.prop1 = "prop1";
  }
}

> babel foo.es6 ..... トランスパイルしたつもりができていない
class F00 {
  constructor() {
    this.prop1 = "prop1";
  }
}
```

Babel のインストールだけではダメ

トランスパイルがきちんと行われるようにするには、トランスパイル対象の項目（トランスフォーメーション）用の「プラグイン」あるいは「プリセット」をプロジェクトごとにローカルにインストールする必要がある。

ES2015 を JS5 にトランスパイルするためのプリセットは「es2015」となっている（プリセットの詳細については Babel の「[Plugins](#)」ページを、es2015 プリセットの内容については「[ES2015 preset](#)」を参照のこと）。

プロジェクトがない状態でちょっと試してみたいのであれば、適当にディレクトリを作成し、そこを作業ディレクトリとしてから、以下のコマンドを実行する。

```
> npm install babel-preset-es2015
```

プリセットのインストール

これにより、カレントディレクトリ以下に node_modules ディレクトリが作成され、そこに各種プラグインがインストールされる（特定のプロジェクトにプリセットをインストールするには「npm install --save-dev babel-preset-es2015」コマンドを実行する。これにより、依存関係が package.json ファイルに記録される）。

トランスパイルを実際に行うには「--presets es2015」オプションを babel コマンドのコマンドラインに追加すればよい。

```
> babel --presets es2015 foo.es6
"use strict";

function _classCallCheck(instance, Constructor) {
  if (!(instance instanceof Constructor)) {
    throw new TypeError("Cannot call a class as a function");
  }
}

var F00 = function F00() {
  _classCallCheck(this, F00);

  this.prop1 = "prop1";
};
```

プリセットに「es2015」を指定して、トランスパイルを実行

長い行についてはこちらで改行を付加しているので、実際の出力とは異なる。

プリセットの指定は .babelrc ファイルに以下のように記述してもよい。


```
{  
  "presets": ["es2015"]  
}
```

プリセットの指定 (JSON)

このようにすれば、「babel source.js」 コマンドで es2015 をプリセットとしてトランスパイルできるようになる。自分でいろいろとコードを書いて試してみるとよい。

次に Web アプリの作成に ES2015 を使ってみよう。

Web アプリを ES2015 で書いてみる

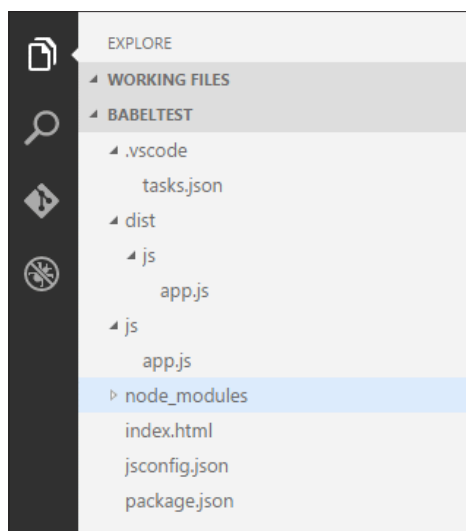
最後に、Web アプリの開発に ES2015 を使い、これを Babel でコンパイルして、JS5 コードにトランスパイルする例を見てみよう。本来は Gulp や Grunt などのビルド関連ツールを利用するところまで見たいところだが、ここでは簡便に Visual Studio Code (以下、VS Code) のタスク機能を使うことにする。

サンプルは「[.NET 開発者のための JavaScript ライブラリカタログ \(MVC フレームワーク編\)](#)」で作成した AngularJS アプリを Superstatic という簡易的な Web サーバーを使い、ローカルホスト上で動作させている(しょうもないコードなので、詳細は割愛する)。

Babel とは無関係な準備がいろいろとあるが、そこは駆け足で見ていこう。

プロジェクトの構成

以下にプロジェクト全体の構成を示す。



プロジェクト全体の構成

dist ディレクトリには Babel で変換した JS5 コードが格納される。js ディレクトリには ES2015 で書かれたソースを格納する。app.js ファイルでは AngularJS のコントローラーを定義して、index.html ファイルから変換後の dist/js/app.js ファイルを読み込む形だ。

jsconfig.json ファイル

VS Code で JS を利用したアプリを開発するには、jsconfig.json ファイルを作成して、以下の記述を行っておくとよい (jsconfig.json ファイルがあると、そのディレクトリが JS プロジェクトのルートであると VS Code が認識し、IntelliSense などの支援が行われるようになる。特に複数の JS ファイルでプロジェクトが構成される際の支援が強力になる)。

```
{
  "compilerOptions": {
    "target": "ES6"
  }
}
```

jsconfig.json ファイルでの target 指定

Superstatic

Superstatic は静的ファイルをホストする簡易的な Web サーバーで、npm から以下のようにしてインストールできる (ここではグローバルにインストールしている)。

```
> npm install -g superstatic
```

Superstatic のインストール

package.json ファイル

package.json ファイルでも、Babel のプリセット指定が可能だ (他の構成要素があれば、それらも記述できるはずだ)。

```
{
  "name": "babeltest",
  ..... 省略 .....
  "scripts": {
    "start": "superstatic --port 3000"
```

```
    },  
    "babel":{  
      "presets": "es2015"  
    }  
  }  
}
```

Babel のプリセット指定

その上の scripts 属性では「npm start」コマンドを実行すると、Superstatic を使って、このアプリをホストするように指定している（ローカルホストのポート 3000）。

準備はここまでだ。後は app.js ファイルを ES2015 で書き直して、これを Babel でトランスパイルできるようにする。

コントローラクラスを作成する

この AngularJS アプリのコントローラーはもともと JS5 で以下のように記述していた。

```
var app = angular.module('myapp', []);  
  
app.controller('FooController', function() {  
  this.test = "foobar";  
});
```

JS5 でのコントローラーの実装

コントローラーの本体である、無名関数部分をクラスとして定義し、これを app.controller メソッドに渡すようにすると以下ようになる。

```
/* global angular */  
var app = angular.module('myapp', []);  
class FooController  
{  
  constructor() {  
    this.test = "foobar";  
  }  
}
```

```
app.controller('FooController', FooController);
```

コントローラーをクラスとして作成

クラスを作成して（その実体は関数だ）、それを app.controller メソッドに渡すようにしているだけだ。なお、先頭のコメント行は、VS Code が識別子「angular」が解決できないので、これはグローバルなオブジェクトだと教えるものである（Angular モジュールのコードは index.html ファイルで読み込んでいるため）。

タスクの構成

最後に VS Code のタスクを構成して、コマンドパレットから ES2015 コードをコンパイルできるようにする。これには次のような tasks.json ファイルを記述する。

```
{
  "version": "0.1.0",
  "command": "babel",
  "isShellCommand": true,
  "showOutput": "always",
  "echoCommand": true,
  "args": ["js/*.js", "--out-dir", "dist"],
  "problemMatcher": "$tsc"
}
```

タスクの構成

エラーメッセージの解釈などで使われるプロブレムマッチャー（problemMatcher）は JS 用のものがなかったので、デフォルトの「\$tsc」のままとなっていることに注意されたい。

ここで重要なのは args 属性で、上の構成では js ディレクトリの下にある全ての「.js」ファイルをトランスパイルして、その結果を「--out-dir」オプションを指定して dist ディレクトリに生成するように指定している。ただし、この指定では dist ディレクトリの下に js ディレクトリが作成されてしまうので、index.html ファイルでは「dist/js/app.js」ファイルを読み込むように指定をしている。

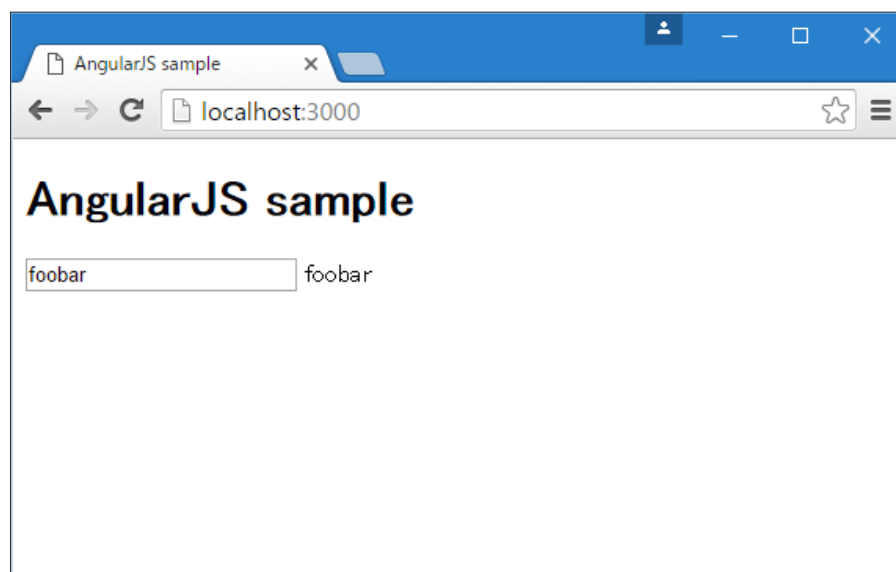
```
<!DOCTYPE html>
<html ng-app="myapp">
<head>
  ..... 省略 .....
```

```
<script src="node_modules/angular/angular.min.js"></script>
<script src="dist/js/app.js"></script>
</head>
<body>
  ..... 省略 .....
</body>
</html>
```

index.html ファイル

構成ができれば、[Ctrl] + [Shift] + [P] キー（macOS では [Command] + [Shift] + [P] キー）を押して、コマンドパレットで「Tasks: Run Task」（「タスク：タスクの実行」）を選択すると、Babel によるトランスパイルが実行されるようになる。

実行結果は次のようになる。



実行結果

非常に簡単なサンプルだが、このようにして Babel を使って、ES2015 なコードでの Web アプリ開発を始めることが分かったと思う。

● C# × JavaScript

C# 開発者のための最新 JavaScript 事情（モジュール編）

Insider.NET 編集部 かわさきしんじ（2015 年 12 月 15 日）

ECMAScript 2015 で導入されたモジュール機構、エクスポート／インポートの方法を Babel で試しながら調べていこう。

前章では Babel という JavaScript トランスパイラーについて話をした。本章では、コマンドラインからこれを使いながら、ECMAScript 2015（以下、ES2015）のモジュール機構について見ていこう。C# コードと 1 対 1 ほどには対応しないので、ES2015 の話題がほとんどとなるがご容赦願いたい。

モジュールとは

説明するまでもないが、「モジュール」とは何らかの関連性を持ったコード群をひとまとまりの固まりとして集めたものだ。モジュールは何らかの機能を外部に提供（エクスポート）し、そのモジュールを利用する側はそれらの機能をインポートして使用する。また、モジュール内部でのみ使用する要素については外部に公開しないようにする。

C# では名前空間やアセンブリ、クラス、アクセス修飾子などを使用することで、論理的／物理的にモジュールを構成できる。

これに対して、JavaScript 5.x（以下、JS5）までの JS には、モジュール的なものを実現することを目的とした言語構造は存在していなかった。JS はもともと Web ブラウザー内部でちょっとした処理を行うための言語であり、大規模なアプリ開発を想定していなかったことを考えると、言語仕様としてモジュールがなかったのはある程度は納得できることだ。

だが、JS の利用領域が広くなり、大規模なアプリ開発にも用いられるようになったことで、機能をモジュールに切り分けて、それらを必要に応じて取り込んで使用できるようにする必要性が高まってきた。

そこで、ES2015 では、言語仕様としてモジュールが導入された。

ECMAScript 2015 におけるモジュール

ES2015 では、一つのファイルが一つのモジュールを構成する。モジュールは、クラスや変数（定数）、関数などを export 宣言により外部にエクスポートできる。エクスポートされた機能を利用する側では import 宣言に

よって、それらを取り込む。簡単な例を以下に示す。

```
// module1.es6
export function getAns2UltimateQ() {
  return 42;
}
```

```
// main.es6
import { getAns2UltimateQ } from './module1';

console.log(getAns2UltimateQ());
```

モジュール利用の例

module1.es6 ファイルで関数をエクスポートし、main.es6 ファイルでそれをインポートして使用している。

このようにエクスポートする側では「export」キーワードに続けて、エクスポートしたいものを記述していくのが基本となる。インポートする側では「import」キーワードに続けてインポートするものを指定して、その後に「from」キーワードとインポート元のモジュールを指定する。

上のコードを Babel でコンパイル／実行すると次のようになる（「npm babel-preset-es2015」コマンドを実行して es2015 プリセットインストールしている）。

```
> babel --presets es2015 module1.es6 -o module1.js
> babel --presets es2015 main.es6 -o main.js
> node main.js
42
```

実行結果

ちなみに前章に紹介した bebel-cli をインストールすると、「babel-node」コマンドも利用できるようになる。これを使用すると、コマンドラインに指定した JS ファイルの実行に必要なファイルのトランスパイルを行った上で、指定した JS ファイルを node コマンドで実行してくれる。そのため、上のコードは次のように実行できる。

```
> del *.js ..... トランスパイル後の .js ファイルを削除
> dir /b ..... ファイルを確認
main.es6
```

```
module1.es6
node_modules
> babel-node --presets es2015 main.es6
42
```

babel-node コマンドによる ES2015 コードの実行

エクスポート／インポートの方法にはいくつかの種類がある。これについては以降で詳しく見ていこう。以下では babel-node コマンドラインの実行結果は頻繁には登場しないが、ここでは Windows 10 と babel-node コマンドで動作を確認している。なお、2016 年 10 月現在、Node.js ではモジュールのインポート／エクスポートはサポートされていない（ので、babel-node コマンドでトランスパイルしたものを node コマンドで実行するのが、モジュールの動作を確認するには一番手っ取り早いだろう）。

エクスポート／インポートの種類と方法

エクスポートには**デフォルトエクスポート**と**名前付きエクスポート**の 2 種類がある。前者はそのモジュールで一番重要なエクスポートであることを意味し、モジュールごとに一つだけ指定可能だ。モジュール内で複数のオブジェクトを公開したいのであれば後者を利用する（数学ライブラリのように多くの関数群を公開するのであれば、こちらを利用することが考えられる）。ちなみに上で示したエクスポートは実際には名前付きエクスポートとなる。

デフォルトエクスポート

デフォルトエクスポートを指定するには「`export default`」宣言を行う。構文は次のようになる。上の例では「`export function() {...}`」と「`default`」がない点に注意しよう。「`default`」の有無が重要だ。

`export default` 関数宣言／クラス宣言／代入文など

デフォルトエクスポートの構文

以下に例を示す。

```
// 例 1
export default function getAns2UltimateQ() { ... };
```



```
// 例 2
export default function() { ... };

// 例 3
export default class Foo {
  ..... 省略 .....
};

// 例 4
export default 3.14;
```

デフォルトエクスポート

実際にはデフォルトエクスポートは 1 モジュールにつき一つだけなので、このようにデフォルトエクスポートをいくつも書いてはいけません。

例 1 と例 2 は関数をデフォルトエクスポートとする例だ。「export default」では無名関数も記述可能である。ただし、例 2 のように無名関数を指定すると、モジュール内でこれを参照する方法がなくなるので注意しよう。

例 3 はクラスをデフォルトエクスポートとしている。あるモジュールが全体として何らかのクラスを実装しているという場合、これをデフォルトエクスポートとするのがよいだろう。上ではクラス名を指定しているが、これを省略することもできる。

例 4 は定数をデフォルトエクスポートとしている。

これらをインポートするには以下のような構文を指定する。

```
import オブジェクト名 from 'モジュールファイル名';
```

デフォルトエクスポートをインポートする

「オブジェクト名」というのは、インポートしたオブジェクトを参照する名前のことだ。これはモジュールで指定した名前とは異なっても構わない。ちなみに、エクスポートは「エクスポート名」を持ち、デフォルトエクスポートではこれは「default」に設定される。

例えば、上の例 1 ～ 4 のデフォルトエクスポートをインポートするには次のようにする。

```
// 例 1 & 例 2
import getAns2UltimateQ from './module1';

console.log(getAns2UltimateQ());

// 例 3
import Foo from './module1';

var f = new Foo();
console.log(f.someprop);

// 例 4
import PI from './module1';

console.log(PI * 1 * 1);
```

デフォルトエクスポートのインポート

これらは全て上で述べた「default」というエクスポート名を持つオブジェクトをインポートして、それらを「getAns2UltimateQ」「Foo」「PI」などの名前で参照することを意味している。

なお、最初に見た「名前付きエクスポート」のインポートとの見た目での違いは「{ }」でオブジェクト名が囲まれているかどうかだ。

名前付きエクスポート／インポート

名前付きエクスポートは、モジュール内で使用している各種オブジェクトを個別にエクスポートするときに使用する。最初の例でも見たように、その基本型は次のようになる。

```
export 変数宣言／関数宣言／クラス宣言など ;
```

名前付きエクスポートの構文（その 1）

以下に例を示す。

```
export function hello() {  
  console.log("hello from module1");  
}  
  
export class Foo {  
  constructor() {  
    this.someprop = "Insider.NET";  
  }  
}  
  
export const PI = 3.14;  // const は ES2015 で導入された機能で定数を表す  
  
// name には export キーワードが付加されていない  
const name = "Insider.NET";
```

名前付きエクスポートの例（その1）

上の例では「const name = "Insider.NET";」行には「export」がない。この場合には、文字列 name はエクスポートされない。

個々のオブジェクトにいちいち「export」を付加するのが面倒なときには、以下のようにしてもよい。

```
export { エクスポートするもののリスト };
```

名前付きエクスポートの構文（その2）

これは外部に公開したいものをエクスポート節（{ }）で指定するというものだ。上の例なら次のように書き換えられる。

```
function hello() {  
  console.log("hello from module1");  
}  
  
class Foo {  
  constructor() {  
    this.someprop = "Insider.NET";  
  }  
}
```

```
    }  
  }  
  
  const PI = 3.14;  
  
  const name = "Insider.NET";  
  
  export { hello, Foo, PI };
```

名前付きエクスポートの例（その 2）

こちらの構文では、コードを記述する際に何をエクスポートするかどうかを気にせずに、どこかの時点で選択的にどれをエクスポートするかを指定できる。

これらをインポートする最も基本的な構文は次のようになる。

```
import { インポートするもののリスト } from 'モジュール';
```

インポートの構文（名前付きインポート）

以下に例を示す。

```
import { hello, Foo } from './module1';  
  
hello();  
  
var f = new Foo();  
console.log(f.someprop);
```

module1 モジュールから hello と Foo をインポート

インポートするものには別名を付けられる。これには次のように as キーワードを使用する。

```
import { hello, Foo as Bar } from './module1';  
  
var f = new Bar();
```

Foo クラスを Bar クラスとして使用する

この場合には、Foo クラスに「Bar」という別名を付けている。このとき、「Foo」という識別子ではこのクラスにはアクセスできないことには注意が必要かもしれない。別名は、アプリの規模が大きくなり、名前が衝突する場合にこれを回避するのに有用になるだろう。

as キーワードは、モジュールを「名前空間」的に使用するためにも使用できる。これを「名前空間インポート」と呼び、「import * as 名前空間名 from ...」のように指定する。例えば、Math モジュールがあったとすると、次のように使用する。

```
// Math.es6  
function square(num) { ... };  
function sqrt(num) { ... };  
function fact(num) { ... };  
..... 省略 .....  
  
export { square, sqrt, fact, ... };
```

```
// main.es6  
import * as Math from './Math';  
  
Math.fact(10);
```

名前空間インポート

これは Math モジュールから全てをインポートし、それらが Math 名前空間に含まれるものとして利用するというものだ。このときには次のような名前付きインポートは行えない。

```
import { fact, square } as Math from './Math'; // エラー!
```

名前空間インポートでは個別にインポートするものは選べない

また、エクスポート側のモジュールで「export」キーワードを付加していないものにはアクセスできない。名前付きエクスポートの最初の例では、以下のように文字列 name には「export」キーワードを付加していなかった。上の名前空間インポートを使って、module1 モジュールからインポートを試みるとどうなるだろう。

```
// module1.es6
export function hello() {
  console.log("hello from module1");
}
```

..... 省略

```
const name = "Insider.NET";
```

```
// main.es6
import * as Mod1 from './module1';

console.log(Mod1.name);
```

```
> babel-node --presets es2015 main.es6
undefined ..... エクスポートされていないものはインポートされない
```

エクスポートされていないものはインポートされない

デフォルトエクスポートと名前付きエクスポートの混在

デフォルトエクスポートと名前付きエクスポート（とそれらのインポート）は混在可能だ。以下に例を示す。ここでは直前に見た module1.es6 ファイルに以下のようなデフォルトエクスポートが追加されているものとしよう。

```
// module1.es6
export default function getAns2UltimateQ() {
  return 42;
}

function hello() {
  console.log("hello from module1");
}
```

..... 省略

```
// main.es6
import getAns, * as Mod1 from './module1';

console.log(getAns());
Mod1.hello();
```

デフォルトエクスポートと名前付きエクスポートをインポート

なお、「名前空間インポート」ではデフォルトエクスポートはインポートできないことには注意しよう。そのため、上のコードではデフォルトエクスポートのインポートと名前空間インポートを別個に指定している。

と書いたが、デフォルトエクスポートのエクスポート名「default」を使うと、名前付きインポートに指定できる。

```
import { default as getAns, hello } from './module1';

console.log(getAns());
```

エクスポート名「default」を使用した名前付きインポート

あるいは、名前付きインポートの特定の項目をデフォルトエクスポートにすることも可能だ。これにも as キーワードを使用する。default だけではなく、別名も指定エクスポートできる（default は特別扱いされる別名だと考えてもよいだろう）。

```
// module1.es6
function getAns2UltimateQ() {
  return 42;
}

function hello() {
  console.log("hello from module1");
}

..... 省略 .....
```

```
export { getAns2UltimateQ as default, hello as sayhello, ...};
```

```
// main.es6  
import getAns , { sayhello } from './module1';  
  
console.log(getAns());  
sayhello();
```

名前付きエクスポートでデフォルトエクスポートを指定

その他のエクスポート

この他にも、インポートした要素をエクスポートするための「export ... from ...」構文、副作用を期待してインポートを行う「import 'モジュール名」構文などがあるが、ここでは紹介は割愛する。

デフォルトエクスポート vs 名前付きエクスポート

デフォルトエクスポートは単一のオブジェクトだけをエクスポートできる。これに対して、名前付きエクスポートは公開したいものを個別に選択できる。「export { ... }」構文を使えば、これを簡便に行える。「名前空間インポート」を使えば、モジュールを名前空間的に使用することも可能だ。

その違いは、デフォルトエクスポートはモジュールごとに一つだけであり、これがそのモジュールで最重要であることを意味することと、それ故にインポートが名前付きインポートよりも簡単に行えることだ。

```
import obj from '...';
```

デフォルトエクスポートのインポート

これに対して、名前付きエクスポートをインポートするには、どれをインポートしたいのかを、明示的に指定する必要がある（名前空間インポートを除く）。

```
import { ... } from '...';
```

名前付きインポート

構文だけを見ると、デフォルトエクスポートと名前付きエクスポートのインポートにはそれほどの労力差はなさそうにも見える。が、実際にアプリを開発する際に、名前付きインポートで必要なものを全て指定しなければな

らない状況になると、この差はここで見るよりも大きなものになることだろう（それを避けるための名前空間インポートなのだろう）。

名前付きエクスポートでは、名前空間インポートを使うことでモジュールを名前空間的に使える。これは魅力的に思えるが、デフォルトエクスポートでも同様なことは可能だ。例えば、デフォルトエクスポートを使うように上で見た Math モジュールを書き換えると次のようになる。

```
// Math.es6
export default {
  square: function square(num) { ... },
  sqrt: function sqrt(num) { ... },
  fact: function fact(num) { ... }
}

// main.es6
import Math from './Math';

Math.fact(10);
```

デフォルトエクスポートを名前空間的に使う

デフォルトエクスポートと名前付きエクスポートは、そのモジュールにおいて個々のオブジェクトが持つ意味合いや、インポートする側での使い勝手を考えて使い分けることになるだろう。

● C# × JavaScript

C# 開発者のための最新 JavaScript 事情（Promise 編）

Insider.NET 編集部 かわさきしんじ（2015 年 12 月 25 日）

ECMAScript 2015 では非同期処理を行うために Promise オブジェクトが導入された。ここではこの基本的な使い方を見ていこう。

前章では [Babel](#) のコマンドラインインターフェースを使用して、ECMAScript 2015（以下、ES2015）のモジュールについて調べた。ここでは ES2015 で採用された Promise オブジェクトによる非同期処理について見ていこう。

Promise とは

ES2015 ではより簡潔な形で非同期処理を行えるように Promise オブジェクトが導入されている。以下では、これまでの非同期処理を簡単に見た後、Promise オブジェクトの使い方の基本を見る。

これまでの非同期処理

JavaScript 5.x（以下、JS5）はシングルスレッドな言語であり、その実行は基本的に線形に行われる。そのため、非同期処理を行うにはイベント機構や `setTimeout` 関数、コールバック関数を組み合わせるなどして行われていた。

```
setTimeout(function() {  
  console.log('hello from 1000ms after');  
}, 1000);  
console.log('hello from outer setTime func');
```

setTimeout 関数による処理の遅延

この例では、`setTimeout` 関数に引数として渡された関数は、1000 ミリ秒の遅延の後に（非同期的に）実行される。このため、外側の `console.log` メソッドの出力が先に表示される。

こうした手法を使えば、JS5 でも何らかの処理を非同期に実行できる。ただし、「この処理が完了したら、このコールバック関数を呼び出してもらって、それが完了したら、今度はあのコールバック関数を呼び出してもらって……」というコードを書くことになるので、一つの関数の中でいくつも無名関数がネストするなど、コードはあまり読みやすいものにはならない。

```
var fs = require('fs');
var http = require('http');

function readFileA(filename, func) {
  fs.readFile(filename, 'utf8', function(err, data) {
    ..... 読み出しが完了したらコールバック「func」を呼び出す .....
  });
}

function getContentsA(url, func) {
  var contents = '';
  var req = http.request({ host: url, method: 'GET' }, function(res) {
    ..... 読み出しが完了したらコールバック「func」を呼び出す .....
  });
  ..... 省略 .....
}

readFileA('urls.txt', function(data) {
  var urls = data.trim().split(/(?:\r\n|[\r\n])/);
  urls.forEach(function(url) {
    getContentsA(url, function(contents) {
      console.log(contents);
    });
  });
});
```

readFileA 関数を呼び出して、それが完了したら、個々の URL に関して getContentsA 関数を呼び出して、それが完了したら……

これに対して、ES2015 で導入された Promise オブジェクトを使うと、よりすっきりとした形で非同期処理を記述できるようになる。

Promise オブジェクト

Promise オブジェクトを使用した場合の典型的な非同期処理の記述は次のようになる。

```
var promise = new Promise(function(resolve, reject) { ... });  
promise.then(function(value) { ... }, function(reason) { ... });
```

Promise オブジェクトを使用した場合の非同期処理の典型

プロミス（Promise オブジェクト）は「何らかの非同期処理の最終結果」を表現する。このオブジェクトの作成時には、そのプロミスで実行する処理を関数として渡す。この関数が非同期処理を行う実体となるが、通常、この関数は二つのコールバック関数をパラメーターに取る。

一つは非同期処理の完了時（成功時）に呼び出す関数、もう一つは非同期処理の失敗時に呼び出す関数だ。前者を「resolve」、後者を「reject」と表記することがよくある。「resolve」と「reject」の二つの関数の実体はプロミスの「then」メソッドに引数として渡す。上で典型としたコードではこれを行っている。

それぞれのコールバック関数のパラメーター「value」と「reason」はそれぞれ、非同期処理の完了によって得られた「値」と非同期処理が完了しなかった「理由」を示すオブジェクトだ（これらはプロミスとして実行する非同期処理内で resolve / reject コールバック関数を呼び出す際に引数としてセットする）。

Promise オブジェクトの使用例

といっても分かりにくいので、簡単なコード例を示しておこう。以下はカレントディレクトリにある「package.json」ファイルを読み出して、それを JSON 形式にパースしたものをコンソールに出力するコードだ。

```
var promise = new Promise(function(resolve, reject) {  
  fs.readFile('package.json', function(err, data) {  
    if (!err) {  
      resolve(data);  
    } else {  
      reject(err);  
    }  
  });  
});  
  
promise.then(function(value) { // resolve コールバック関数  
  var pkginfo = JSON.parse(value);  
  console.log(pkginfo);  
}, function(reason) { // reject コールバック関数
```

```
console.log(reason.message);  
});
```

Promise オブジェクトの使用例

このコードは Node.js に依存していることには注意されたい。

ここで作成した Promise オブジェクトは、Node.js のファイルシステムアクセス API を使用して、ローカルなファイルを読み出している。このときに呼び出している `readFile` メソッドは非同期に実行される。そして、読み出しが完了すると、コールバック関数「`function(err, data)`」が呼び出される。その内部では、エラーがなければコールバック関数「`resolve`」を、エラーが発生時にはコールバック関数「`reject`」を呼び出している。それぞれのコールバック関数では渡された引数の値を使用できる。

このプロミスの `then` メソッドには二つのコールバック関数を渡して、成功時には得られた値を `JSON.parse` メソッドでパースしてコンソールに出力し、失敗時にはその原因を示すメッセージをコンソールに出力している。

このように、非同期に行う処理を表す Promise オブジェクトを作成して、それが成功／失敗したら（すなわち「`then`」）行う処理をさらに記述していくのが、Promise オブジェクトの使い方の基本型だ。

【コラム】 本稿のコードの実行方法

本稿のサンプルコードは Babel のコマンドラインインターフェース (`babel-cli`) と ES2015 用のプリセット (`babel-preset-es2015`) をインストールした環境で実行できる。これには、これまでに見てきたように `babel` コマンドで ES2015 コードを一度 JS コードにトランスパイルして、トランスパイル後のファイルを `node` コマンドに指定するか、`node-babel` コマンドに ES2015 コードを指定する。あるいは Node.js のバージョンによっては、そのまま `node` コマンドで実行できる（筆者は Node.js のバージョン 6.8.1 で動作を確認している）。ただし、Node.js は 2016 年 10 月現在、ES2015 の `import` / `export` をサポートしていないので、モジュールをインポートするには「`var fs = require('fs');`」など従来の形式で記述を行う必要がある。

ファイルの読み込み

以下に示すのは、冒頭に示したファイル読み込み処理を関数にまとめたものだ。コードは見ての通りのものなので説明は割愛する。ファイル読み込みを行う Promise オブジェクトを関数の戻り値としているので、関数呼び出しに `then` メソッドをそのまま追記していける。

```
var fs = require('fs');

function readFileAsync(filename) {
  return new Promise(function(resolve, reject) {
    fs.readFile(filename, 'utf8', function(err, data) {
      if (!err) {
        resolve(data);
      } else {
        reject(err);
      }
    });
  });
}

// readFileAsync 関数の使用例
readFileAsync('package.json').then(function(data) {
  console.log(JSON.parse(data));
}).catch(function(reason) {
  console.log(err);
});

readFileAsync('urls.txt').then(function(data) {
  var urls = data.trim().split(/(?:\r\n|[\r\n])/);
  console.log(urls);
});
```

非同期にファイル読み込みを行う readFileAsync 関数とその使用例»

ここで注目してほしいのは二つ。一つは「catch」メソッドだ。もう一つは二つ目の使用例では reject コールバック関数を呼び出していないことだ。

先ほど、then メソッドには二つの関数を引数として渡すと述べたが、reject コールバック関数に関しては catch メソッドに渡してもよい。実際には resolve コールバック関数も reject コールバック関数もオプションとなっている（ただし、どちらかを渡さないと何も起きない）。catch メソッドは実質的には「then(undefined, function() {...})」と同様に考えてよい。

ただし、then メソッドに reject コールバック関数を渡した場合には、そのプロミスでエラーが発生したときには reject 関数が呼び出されるが、それ以降にエラーが発生した場合にはそのエラーを処理することはできない。then / catch メソッドのチェーンについては後で紹介するが、エラーをまとめてトラップするのであれば、専用のハンドラーを用意して次のように記述できる。

```
var promise = new Promise(...);
function err_handler() { ... }

promise
  .then(...)
  .then(...)
  .catch(err_handler);
```

then / catch メソッドのチェーン

Web ページの読み出し

もう一つ例を見てみよう。これは Node.js の http モジュールを使用して、Web ページの内容を取得する getWebPageAsync 関数だ。

```
var http = require('http');

function getWebPageAsync(url) {
  return new Promise(function(resolve, reject) {
    var contents = '';
    var req = http.request({ host: url, method: 'GET' }, function(res) {
      res.on('data', function(chunk) {
        contents += chunk;
      });
      res.on('end', function() {
        resolve(contents);
      });
    });
    req.on('error', function(reason) { reject(reason.message); });
    req.end();
  });
}
```

```
}
```

```
// 使用例
```

```
getWebPageAsync('www.buildinsider.net')  
  .then(contents => console.log(contents));
```

Web ページの内容を取得する `getWebPageAsync` 関数とその使用例

引数に渡されたホスト名を使用して、Web ページの内容を取得している。取得が終わったら、`resolve` コールバック関数を呼び出し、何らかのエラーが発生したら `reject` コールバック関数を呼び出すようになっている。

使用例では、アロー関数を使用して、記述を簡潔にした。実際には、以下のようにもっと簡潔に記述もできる。

```
getWebPageAsync('www.buildinsider.net')  
  .then(console.log);
```

使用例をさらに簡潔な記述に

正直、どうでもいいような内容の二つの関数を紹介したのは、これらを使って二つのことを試してみたいからだ。一つは、複数のプロミスの非同期実行。もう一つは、プロミスチェーンだ。

Promise.all メソッド

同じような処理をプロミスを使って行う場合、それらをまとめて処理することもできる。これには `Promise.all` メソッドを使用する。実際には、配列の `map` メソッドを使用して、配列の要素にプロミスを生成する関数を適用し、それらを `Promise.all` メソッドで処理するのが一般的な手順になる。以下に例を示す。

```
var urls = [  
  'www.google.com',  
  'www.microsoft.com'  
];  
  
var promiselist = urls.map(getWebPageAsync)  
Promise.all(promiselist)  
  .then(contents => console.log(contents));
```

プロミスをまとめて処理

ここでは URL となる文字列を配列に格納して、map メソッドでその要素に getWebPageAsync 関数を適用している。getWebPageAsync 関数は「プロミスを返す」ので、変数 promiselist には「プロミスの配列」が格納される。

Promise.all メソッドはこれらの配列の状態を表すプロミスを返す。そして、全てのプロミスが終了すると then メソッドが呼び出される。このときに渡される値（上の例では contents 引数）は、それぞれの処理結果を要素とする配列となる（contents[0]、contents[1] などとしてアクセス可能。上の例ではまとめてコンソールに出力している）。

プロミスチェーン

ここまでが準備だ。最後に、プロミスチェーン（then / catch メソッドの連結実行）を見てみよう。ここでは urls.txt ファイルから URL の一覧を取得して、それぞれのページの内容を読み込んで、コンソールに出力する。

```
var fs = require('fs');
var http = require('http');

function readFileAsync(filename) {
  return new Promise(function(resolve, reject) {
    ..... 省略 .....
  });
}

function getWebPageAsync(url) {
  return new Promise(function(resolve, reject) {
    ..... 省略 .....
  });
}

readFileAsync('urls.txt')
  .then(data => {
    var urls = data.trim().split(/(?:\r\n|\n)/);
    return Promise.all(urls.map(getWebPageAsync));
  })
  .then(contents => console.log(contents))
  .catch(reason => console.log(reason));
```

then / catch メソッドはチェーンできる。これは、これらのメソッドが「プロミスを送り、Promise は then / catch メソッドを持つ」からだ（ここでは紹介しなかったが、Promise オブジェクトには「thenable」という概念もある。これについては次章で紹介するが、簡単には「Promise オブジェクトでなくとも then メソッド呼び出しと同様な挙動を示すオブジェクトはプロミスチェーンの中から呼び出せる」という概念だ）。

ここでは、readFileAsync 関数が返すプロミス进行处理する resolve コールバック関数（アロー関数を使って記述）内で、読み込んだ文字列（改行で区切られた URL 一覧）から前後の空白文字（改行文字）を削除したものを、その途中に含まれている改行文字をデリミターとして分解して、配列に格納している。URL 一覧が配列になったので、後は上で見たように、これをプロミスの配列にして（urls.map メソッド呼び出し）、それを Promise.all メソッドに食わせたものを戻り値としている。ここまでは、最初の then メソッドで行っていることだ。

その後は、全ての処理が完了した時点で、そのページの内容をコンソールに出力し、エラー発生時にはそのことをコンソールに出力するようにしているだけだ。

このように、Promise オブジェクトを使用すると、then / catch メソッドをチェーンさせながら極めて簡潔な形で非同期処理を記述できるようになる。アロー関数を使用すると「function() { ... }」という記述もなくなるので、さらに簡潔になる。

● C# × JavaScript

C# 開発者のための最新 JavaScript 事情(Promise 編パート 2)

Insider.NET 編集部 かわさきしんじ (2016 年 01 月 08 日)

Promise オブジェクトの状態と、Promise オブジェクトが提供するメソッド、thenable オブジェクトなど、少し高度な話題について見てみよう。

前章では Promise オブジェクトを使用した非同期処理の基本について見た。本章では Promise オブジェクトについてさらに詳しく見ていこう。

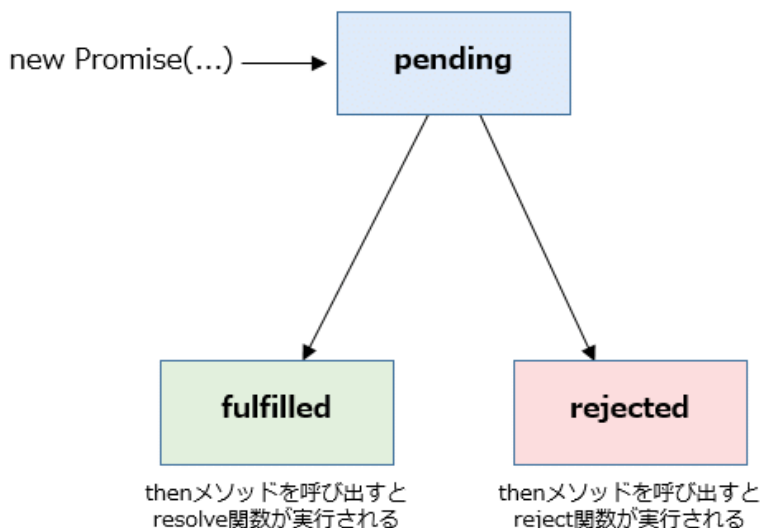
Promise オブジェクトの状態

Promise オブジェクトは次の三つの状態を持つ。

- fulfilled
- rejected
- pending

fulfilled と rejected は、Promise オブジェクトが表す非同期処理が完了しており、fulfilled はそれが成功したことを、rejected は失敗したことを表す。pending は非同期処理がまだ完了していないことを表す。これらは相互に排他的であり、かつ一度 fulfilled 状態あるいは rejected 状態になった Promise オブジェクトの状態がそれ以外に遷移することはない。

これを図に示すと次のようになる。



Promise オブジェクトの状態

また、pending 以外の状態 (fulfilled あるいは rejected のいずれかの状態) のことを「settled」 (= fulfilled か rejected のいずれかに落ち着いて変化することがない状態) と呼ぶことがある。

fulfilled 状態の Promise オブジェクトに対して then メソッドを呼び出すと、第 1 引数に指定した関数 (= 成功時に実行される関数) が呼び出される。rejected 状態の場合は、第 2 引数に指定した関数 (= 失敗時に実行される関数) が呼び出される。pending 状態の場合は、他の二つの状態のいずれかに状態が変化したときに実行される関数のセットアップが行われる。

```
// new Promise() 呼び出しで作成した直後のプロミスは pending 状態
var p = new Promise((resolve, reject) => { ... });

..... 省略 .....

p.then(
  function(value) { ... }, // p が fulfilled 状態の場合に呼び出される
  function(reason) { ... } // p が rejected 状態の場合に呼び出される
);
```

Promise オブジェクトの状態

最後に Promise オブジェクトが fulfilled 状態か rejected 状態のいずれかになるか、プロミスチェーンによって別の Promise オブジェクトの状態と関連付けられることを「resolved」(解決された) と呼ぶことがある。

コンストラクタ呼び出しによって作成された直後の Promise オブジェクトは pending 状態になるが、最初から fulfilled / rejected 状態の Promise オブジェクトを作成するメソッドもある。それが次に見る Promise.resolve メソッドと Promise.reject メソッドだ。

Promise.resolve メソッドと Promise.reject メソッド

Promise.resolve メソッドは、引数に渡した値を完了値として成功した状態の Promise オブジェクトを返す (引数に Promise オブジェクトを渡した場合は、その Promise オブジェクトが戻り値となる)。これはつまり、以下の Promise コンストラクタ呼び出しと同じことを意味する。

```
// 以下の二つのコンストラクタ／メソッド呼び出しは同じ意味
var p1 = new Promise((resolve, reject) => resolve("always resolved"));
var p2 = Promise.resolve("always resolved");

// p1 と p2 は成功しているので常に第 1 引数 (= resolve 関数) が呼び出される
p1.then(
  function(value) { console.log("success: " + value); },
  function(reason) { console.log("fail: " + reason); }
);

p2.then(
  value => console.log("success: " + value),
  reason => console.log("fail: " + reason)
);
```

Promise.resolve メソッドは成功した Promise オブジェクトを返す

出力結果は共に「success: always resolved」になる。なお、then メソッドに引数に渡す関数を p1 オブジェクトについては関数式を、p2 オブジェクトについてはアロー関数を使用しているが、これらも同じ意味になる。

これは何かのオブジェクトを Promise オブジェクト化して、それを使って何らかの処理を行いたいときに活用できる（後述）。

Promise.reject メソッドは、引数に渡した値を非同期処理が失敗した理由とする Promise オブジェクトを返す。Promise.resolve メソッドと対称的に、こちらは以下のコンストラクタ呼び出しと同じことを意味する。

```
// 以下の二つのコンストラクタ／メソッド呼び出しは同じ意味
var p1 = new Promise((resolve, reject) => reject("always rejected"));
var p2 = Promise.reject("always rejected");

// p1 と p2 は失敗しているので常に第 2 引数 (= reject 関数) が呼び出される
p1.then(
  function(value) { console.log("success: " + value); },
  function(reason) { console.log("fail: " + reason); }
);

p2.then(
```

```
value => console.log("success: " + value),  
reason => console.log("fail: " + reason)  
);
```

Promise.reject メソッドは失敗した Promise オブジェクトを返す

出力結果は共に「fail: always rejected」になる。

常に失敗した（rejected 状態の）Promise オブジェクトが戻り値となるので、通常時にはあまり使い道はないだろうが、テスト／デバッグ用途で使えると思われる。

次に前章で説明した Promise.all メソッドと対になる Promise.race メソッドについて簡単に見ておこう。

Promise.race メソッド

前章で見た Promise.all オブジェクトは、「配列などに格納した複数の Promise オブジェクトの全てが完了すること」を表す Promise オブジェクトを返す。全てのプロミスが完了するのを待機し、それぞれの完了結果を配列の要素として取り扱える。

Promise.race メソッドも配列などに格納された複数の Promise オブジェクトを受け取り、新たな Promise オブジェクトを返すが、Promise.all メソッドとは異なり、複数の Promise オブジェクトのいずれかが完了した時点で、新たな Promise オブジェクトも完了する。

以下に例を示す。

```
var promiselist = [  
  new Promise(  
    (resolve, reject) => setTimeout(() => resolve("promise0"), 500)  
  ),  
  new Promise(  
    (resolve, reject) => setTimeout(() => resolve("promise1"), 100)  
  ),  
  new Promise(  
    (resolve, reject) => setTimeout(() => resolve("promise2"), 1000)  
  )  
];
```

```
Promise.race(promiselist).then(  
  value => console.log("resolve: " + value),  
  reason => console.log("reject: " + reason)  
);
```

Promise.race メソッド

ここでは三つの Promise オブジェクトを要素とする配列を作成して、それを Promise.race メソッドに渡している。要素となる Promise オブジェクトでは setTimeout 関数を使って、それぞれのプロミスが完了するタイミングに差を付けている。

このコードを実行すると次のようになる。

```
>node promisetest.js  
resolve: promise1
```

いずれかの Promise オブジェクトが完了した時点で、Promise.race メソッドが返す Promise オブジェクトも完了する

ここでは node コマンドで上記の ES2015 コードを実行しているが、Babel のコマンドラインツール／プリセットのインストール／設定をしているのであれば、babel-node コマンドを使ってもよい。なお、Node.js のバージョンによっては node コマンドに「--harmony」コマンドラインオプションが必要な場合がある（以下同様）。

なお、上の実行結果では分からないのだが、プロンプトが表示されて、コンソールへの入力が可能になるまでには少し時間がかかる。これは Promise.race メソッドが返す Promise オブジェクトは完了しても、もともとの配列に格納されていた Promise オブジェクトの実行は続けられるためだ。上のコードを少し変更して試してみると、このことが分かる。

```
var promiselist = [  
  new Promise(  
    (resolve, reject) => setTimeout(() => {  
      console.log("promise0");  
      resolve("promise0");  
    }, 500)  
  ),  
  ..... 省略 .....  
];  
  
Promise.race(promiselist).then(  
  value => console.log("resolve: " + value),  
  reason => console.log("reject: " + reason)  
);
```

```
value => console.log("resolve: " + value),  
reason => console.log("reject: " + reason)  
);
```

Promise.race メソッドが返す Promise オブジェクトが完了しても、配列の各要素の実行は続けられる

ここでは最初の要素が完了したときに、コンソールに出力するように変更している。このコードの実行結果は次のようになる。

```
>node promisetest.js  
resolve: promise1  
promise0
```

Promise.race メソッドが返す Promise オブジェクトが完了しても、配列の各要素の実行は続けられる

このように、最初に実行が完了しなかった Promise オブジェクトについても、その完了時にコンソールにメッセージを表示している。

余談ではあるが、Promise.all メソッドに渡した複数の Promise オブジェクトのいずれかが失敗したときにはどうなるだろう。上のコードを再度改変して、少し試してみよう。

```
var promiselist = [  
  new Promise(  
    (resolve, reject) => setTimeout(() => reject("promise0"), 500)  
  ),  
  ..... 省略 .....  
];  
  
Promise.all(promiselist).then(  
  value => console.log("resolve: " + value),  
  reason => console.log("reject: " + reason)  
);
```

Promise.all メソッドが待機する全ての Promise オブジェクトの中で reject されるものが出た場合

ここでは最初の要素でタイムアウト発生後に reject 関数を呼び出している（失敗）。この場合、実行結果は次のようになる。比較のため、全ての Promise オブジェクトが成功して完了した場合の出力結果も示しておこう。

※失敗した場合

```
>node promisetest.js  
reject: promise0
```

※全て成功した場合

```
>node promisetest.js  
resolve: promise0,promise1,promise2
```

Promise.all メソッドが待機する全ての Promise オブジェクトの中で reject されるものが出た場合

この結果から分かる通り、Promise.all メソッドが返す Promise オブジェクトは、それが待機する Promise オブジェクトのいずれかが失敗して完了した場合には、その時点で完了する（そのため、失敗時には「promise0」のみが表示されている。これは成功時の「promise0,promise1,promise2」とは異なる結果だ）。

では、最後に「thenable」（then 可能な）オブジェクトについて見てみよう。

thenable オブジェクト

Promise オブジェクトは then メソッドと catch メソッドのチェーンにより、処理を連続させることで、コールバック関数を連鎖させることなく非同期処理を簡潔に記述できるのが大きなメリットである。

このチェーンの中に「Promise オブジェクト以外のオブジェクトを組み込めない」というとそうでもない。そうした場面で使えるのが「thenable」オブジェクトだ。これは「then メソッドを呼び出し可能」つまり Promise オブジェクト的に扱えるオブジェクトのことだ。

では、then メソッドとはどんなものだっただろう。それはもちろん、resolve 関数と reject 関数を引数に受け取り、それを使って、Promise オブジェクトが完了したときに対応する処理を行うメソッドだ。そういうわけで、これと同様な処理を行う関数あるいはオブジェクトであれば、チェーンの中でこれを使用できる。

以下に簡単な例を示す。

```
var thenableobj = {
  then: function(resolve, reject) {
    setTimeout(() => resolve("always resolved"), 1000);
  }
};

console.log("begin");
var p = Promise.resolve(thenableobj);
p.then(
  value => console.log(value),
  reason => console.log(reason)
)
console.log("end");
```

シンプルな thenable オブジェクト

最初に thenableobj オブジェクトを作成している。このオブジェクトは、then メソッドを持っている。これは resolve 関数と reject 関数を受け取る。ここでは、常に resolve 関数を呼び出すだけだが、これによりこのオブジェクトは「Promise オブジェクトの作法に従った then メソッド呼び出し」が可能になっている。

次に、これを先ほど紹介した Promise.resolve メソッドに渡すことで、Promise オブジェクト化している。これにより、その下で行っているように then メソッドを呼び出せる。

今度は thenable な関数の例を見てみよう。この関数は then メソッドを持つオブジェクトを返す。そして、then メソッドの中では引数 condition の値で、resolve 関数／reject 関数のいずれかを呼び出すようになっている。

```
function thenablefunc(condition) {
  return {
    then: function(resolve, reject) {
      if (condition) {
        resolve("resolved");
      } else {
        reject("rejected");
      }
    }
  }
}
```

```
};  
}  
  
console.log("begin");  
var p = Promise.resolve(thenablefunc(false));  
p.then(  
  value => console.log(value),  
  reason => console.log(reason)  
);  
console.log("end");
```

thenable 関数の例（その1）

これまでの二つの例を見て「どちらも then メソッドがあるんだから、Promise オブジェクトにしなくていいんじゃない?」と思った人もいるかもしれない（というか、筆者がそうだ）。つまり、以下のようなコードを書いても問題ないのではないか、という話だ。

```
function thenablefunc(condition) {  
  return {  
    then: function(resolve, reject) {  
      ..... 省略 .....  
    }  
  };  
}  
  
console.log("begin");  
// var p = Promise.resolve(thenablefunc(false));  
thenablefunc(false).then(  
  value => console.log(value),  
  reason => console.log(reason)  
);  
console.log("end");
```

thenable 関数の例（その2）

実際に試してみると分かるが、この場合、thenablefunc 関数の呼び出しは同期的に行われ、その戻り値に対する then メソッド呼び出しも同期的に行われている。そのため、上の二つのコードの実行結果は異なるものになる。

※ Promise.resolve(thenablefunc(false)) に対して then メソッドを呼び出した場合

```
>node promisetest.js  
begin  
end  
rejected
```

※ thenablefunc(false) の戻り値に対して then メソッドを呼び出した場合

```
>node promisetest.js  
begin  
rejected  
end
```

thenable オブジェクトの動作の違い

もちろん、then メソッドを持ち、Promise オブジェクトライクな動作をすれば、それは thenable オブジェクトなので、後者の使い方が悪いというわけではないと思われる。が、実行が同期的か非同期的かの違いはあるので、注意はしておこう。

だが、上で試した thenable な関数呼び出しに対して then メソッドを呼び出す場合 (thenablefunc(false).then(...))、これは Promise オブジェクトを返すわけではないので、これ以上のチェーンはできない。ということは、やはり Promise.resolve メソッドによる Promise オブジェクト化は必須と考えるべきだ。

thenable オブジェクト内部でコールバックによる非同期処理を行っていれば、それは非同期に実行される。以下は前章で作成した getWebPageAsync 関数を thenable オブジェクト化したものだ。then メソッドを持つオブジェクトを返すようにしただけだ。

```
var http = require('http');  
  
function getWebPageThenable(url) {  
  return {  
    then: function(resolve, reject) {  
      var contents = '';
```

```
var req = http.request({ host: url, method: 'GET' }, function(res)
{
    res.on('data', function(chunk) {
        contents += chunk;
    });
    res.on('end', function() {
        resolve(contents);
    });
});
req.on('error', function(reason) { reject(reason.message); });
req.end();
}
};
}

console.log("begin");
var p = Promise.resolve(getWebPageThenable("www.microsoft.com"));
p.then(
    value => console.log(value),
    reason => console.log(reason)
);
console.log("end");
```

getWebPageThenable 関数

興味のある方は、この実行結果についても検討してみてください。

● C# × JavaScript

C# 開発者のための最新 JavaScript 事情 (async 関数編)

Insider.NET 編集部 かわさきしんじ (2016 年 01 月 22 日)

Promise オブジェクトをベースに、非同期処理をよりスッキリと記述できるようになる async 関数の基本について見ていこう。

前章では Promise オブジェクトについて少し詳しく見た。ここでは Promise オブジェクトをベースとした async 関数 (async / await キーワード) による非同期処理の基本について見ていこう。なお、async 関数は 2016 年 7 月に仕様が確定したところだ。恐らくは 2017 年に発行予定の ECMAScript 2017 の言語仕様に含まれるものと思われる。

async 関数

C# 開発者の方であれば、async / await キーワードによる非同期処理の記述は既にお手のものであろう。上述したように 2017 年に発行予定の ECMAScript 2017 (以下、ES2017) では、C# と似た構文で async / await キーワードを使って非同期処理を記述できるようになる。C# では Task (Task<T>) 型のオブジェクトと async / await キーワードを組み合わせていたが、ES2015 では Promise と async / await キーワードを組み合わせで使用 (両者のパターンが似ているのが意図的なものかどうかは不明だ)。

つまり、非同期処理を待機する箇所で「await」し、「await」を含んだ関数は「async」関数として宣言をするということだ。これにより、await した箇所で処理は中断し、制御は async 関数の呼び出し側へと返され、非同期処理が完了した時点で async 関数の残りのコードが実行されるようになる。これを ECMAScript の仕様策定を行っている「[Ecma TC39](#)」では「[async 関数](#)」と呼んでいる。

以下に簡単な使用例を示す。C# の async / await キーワードの使い方と同様だ。なお、doLongTimeTask 関数は Promise オブジェクトを返す関数だ (つまり、何らかの非同期処理を行う)。

```
async function doLongTimeTaskAsync(n, msg) {  
  var result = await doLongTimeTask(n, msg);  
  console.log(result);  
}
```

ECMAScript 2017 で導入される予定の async 関数

await により、doLongTimeTask 関数が完了するのを待機しながら、制御は呼び出し元へ返される。そして、doLongTimeTask 関数が成功して完了すると、以降の行（console.log メソッド呼び出し）が実行される。doLongTimeTask 関数が失敗した場合には例外が発生する（そのため、本来は例外処理まで記述しておくのが正しいだろう。後述）。

では、Promise オブジェクトと then メソッドを使った場合のコードと、async 関数を使った場合のコードとを比較してみよう。なお、本稿のコードは基本的に [Babel の「Try it out」ページ](#)、Windows 10 / Mac (OS X) 上の Babel コマンドラインツールで動作を確認している（プリセットには「babel-preset-latest」を使用している）。

Promise オブジェクトと then メソッドを使った場合

doLongTimeTask 関数のコードが以下のようなものだったとする。

```
function doLongTimeTask(n, msg) {
  return new Promise(function(resolve, reject){
    setTimeout(function() {
      resolve('resolved: ' + msg);
    }, n);
  });
}

// アロー関数を使用した場合
function doLongTimeTask(n, msg) {
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve('resolved: ' + msg), n);
  });
}
```

doLongTimeTask 関数

この関数は、新規に作成した Promise オブジェクトを返し、その Promise オブジェクトが行う非同期処理は「引数 n に指定した時間（ミリ秒単位）が経過した後に resolve 関数を呼び出す」ことだ。

これまでのおさらいになるが、ES2015 では then メソッドを使い、次のようなコードに非同期処理をスッキリと記述できる。

```
function doLongTimeTaskPromise(n, msg) {  
  var p = doLongTimeTask(n, msg);  
  p.then(function(value) { // value => console.log(value)  
    console.log(value);  
  }, function(reason) {   // reason => console.log(reason)  
    console.log(reason)  
  });  
}  
  
console.log('begin');  
doLongTimeTaskPromise(1000, 'promise');  
console.log('end');
```

Promise オブジェクトと then メソッドの組み合わせ

コメントアウトしているのは、アロー関数を使った場合の resolve 関数 / reject 関数だ。

変数 `p` が参照する Promise オブジェクトは非同期に処理を行い、それが完了すると、then メソッドに渡した resolve 関数（あるいは reject 関数）が呼び出される。処理が完了するのを待機している間、コードの実行は進む。よって、このコードの実行結果は次のようになる。

```
>babel-node --presets latest asyncctest.es6  
begin  
end  
resolved: promise
```

Promise オブジェクトを素で使用する

コンソールには「begin」と「end」の出力が先に行われ、最後に resolve 関数の出力「resolved: promise」が行われる（上述したようにプリセットとしては「babel-preset-latest」をインストールしてある）。

async 関数を使った場合

async 関数を使うと、同期処理を行うコードと似た形で非同期処理を記述できる。先ほども紹介したが、async 関数を使った場合のコードを以下に示す。なお、トップレベルで直接「await doLongTimeTask(...)」のような記述はできないので注意しよう。


```
function doLongTimeTask(n, msg) {  
  ..... 省略 .....  
}  
  
async function doLongTimeTaskAsync(n, msg) {  
  var result = await doLongTimeTask(n, msg);  
  console.log(result);  
}  
  
console.log('begin');  
doLongTimeTaskAsync(1000, 'async');  
console.log('end');
```

Promise オブジェクト / then メソッドを使用したコードと async 関数を使用したコード

Promise オブジェクトを使った場合でもコードはシンプルだが、then / catch メソッドを使った Promise オブジェクトに固有な記述の仕方になる。一方、async 関数を使えば、関数宣言の先頭に「async」が、非同期処理を待機する箇所に「await」がある以外は、同期処理を書くのと同様に記述できる。

なお、Babel のコマンドラインツールで上記のコードをトランスパイルするには、「babel-preset-latest」プリセットを事前にインストールしておく必要がある（本稿執筆時点＝2016 年 11 月 07 日現在）。

```
>babel-node --presets latest asyncTest.es6  
begin  
end  
resolved: async
```

async 関数を使った場合の実行結果

「--presets」オプションを毎回指定するのが面倒であれば、「.babelrc」ファイルに以下の記述をしておくとい。

```
{  
  "presets": ["latest"]  
}
```

.babelrc ファイルで使用するプリセットを指定

Promise オブジェクトと async 関数

ここで Promise オブジェクトを返す関数と、async 関数とに注目しよう。

```
function doLongTimeTask(n, msg) {  
  return new Promise((resolve, reject) => {  
    setTimeout(() => resolve('resolved: ' + msg), n);  
  });  
}  
  
async function doLongTimeTaskAsync(n, msg) {  
  var result = await doLongTimeTask(n, msg);  
  console.log(result);  
}  
  
doLongTimeTaskAsync(1000, 'async');
```

Promise オブジェクトで行う非同期処理と、それを待機する async 関数

Promise オブジェクトが実行する処理の中で resolve 関数呼び出しが行われると、その引数の値が待機した関数（というか、Promise オブジェクト）の戻り値となる。そのため、上のコードを実行すると「resolved: async」という文字列が変数 result には代入される。async 関数の呼び出し側が resolve 関数を与える必要はない（恐らく、組み込みの resolve / reject 関数が提供され、これらの処理を行ってくれる。詳細は [async 関数の仕様のドラフト](#)などを参照されたい）。

要するに、Promise オブジェクトと then メソッドを使った場合に resolve 関数に記述していた処理は、await で待機したコード以降にそのまま記述できるようになっているということだ。

また、先ほども書いたように、reject 関数を呼び出すと例外が発生する。そのため、catch メソッドを Promise オブジェクトにチェーンさせるのではなく、try ~ catch 文で例外を補足するのが async 関数のやり方になる。以下に例を示す（あくまでもサンプルなので、setTimeout 関数で指定された時間が経過したら、有無をいわず、reject 関数を呼び出している）。

```
function doLongTimeTask(n, msg) {  
  return new Promise((resolve, reject) => {
```

```
        setTimeout(() => reject('rejected: ' + msg), n);
    });
}

async function doLongTimeTaskAsync(n, msg) {
    var result;
    try {
        result = await doLongTimeTask(n, msg);
        console.log(result);
    } catch(e) {
        console.log(e);
    }
}

doLongTimeTaskAsync(1000, 'async');
```

reject 関数呼び出しは例外を発生させる

また、上記のコード例では「Promise オブジェクトを返す関数呼び出しを待機している」が以下のような記述も可能だ。

```
async function doLongTimeTaskAsync(n, msg) {
    var promise = doLongTimeTask(n, msg);
    var result = await promise;
    console.log(result);
}
```

Promise オブジェクトを await した結果を変数に代入 Promise オブジェクトを await した結果を変数に代入

名前から分かる通り、変数 promise は doLongTimeTask 関数の戻り値である Promise オブジェクトを参照し、変数 result にはそれを待機した結果（この場合は文字列）が代入される。

非同期処理に順序性を持たせる

Promise オブジェクトと then メソッドを使った場合は、以下のように then メソッド／catch メソッドをチェーンさせて、非同期処理の実行に順序性を持たせることができる。

```
var promise = ...
promise.then(function(value) {
  ...
  return value2;
}).then(function(value2) {
  ...
  return value3;
}).then(function(value3) {
  ...
}).catch(function(reason) {
  ...
});
```

プロミスチェーン

一方、`async` 関数を使った場合には、非同期処理を行う箇所で `await` を使って待機するだけだ。例えば、以下のコードでは 1 秒ごとにコンソールにメッセージが表示される。

```
async function doLongTimeTaskAsync() {
  var result;
  result = await doLongTimeTask(1000, 'task1');
  console.log(result);
  result = await doLongTimeTask(1000, 'task2');
  console.log(result);
  result = await doLongTimeTask(1000, 'task3');
  console.log(result);
}

doLongTimeTaskAsync();
```

`await` を使って、非同期処理を順序よく実行する

Promise オブジェクトをそのまま使用している上のコードも十分に分かりやすいものだが、比べてみると、`async` 関数の方がさらに分かりやすいコードとなっている。

もう少しまともな例

ここまで、setTimeout 関数を使ってコンソールに出力するだけのコードを見てきた。そこで以前も使用した Web ページの内容を取得する関数を async 関数から使用する例も見ておこう。といっても、基本は同じだ。

```
var http = require('http');

function getWebPage(url) {
  return new Promise(
    function(resolve, reject) {
      var contents = '';
      var req = http.request({ host: url, method: 'GET' }, function(res)
      {
        res.on('data', function(chunk) {
          contents += chunk;
        });
        res.on('end', function() {
          resolve(contents);
        });
      });
      req.on('error', function(reason) { reject("fail :" + reason.
message); });
      req.end();
    });
}

async function getWebPageAsync(url) {
  try {
    var contents = await getWebPage(url);
    console.log(contents);
  } catch(e) {
    console.log(e);
  }
}
```

```
getWebPageAsync('www.microsoft.com');
```

Web ページを取得する async 関数

非同期処理を行い Promise オブジェクトを返す getWebPage 関数と、それを使用する getWebPageAsync 関数がある。後者では try ~ catch 文によりエラーが発生した場合には、その原因が分かるようになっている。興味のある方は、getWebPageAsync 関数に存在しない URL を与えて実行してみよう。なお、このサンプルについては Babel の「[Try it out](#)」ページでは動作しないので、手元の環境で試す必要がある。

それでは、本章の最後に TypeScript での async 関数の仕様についても見ておこう。

TypeScript で async 関数を使う

TypeScript 1.7 以降では async 関数がサポートされているので、上記のコードもコンパイル可能だ。ただし、コンパイラターゲットを ES2015 (ES6) に設定する必要がある。コンパイル後のコードが ES2015 になる。

例として、先ほど見た 1 秒ごとにコンソールに出力するコードを TypeScript でコンパイルしてみよう。

```
>type asyncctest.ts
function doLongTimeTask(n, msg) {
  ..... 省略 .....
}

async function doLongTimeTaskAsync() {
  var result;
  result = await doLongTimeTask(1000, 'task1');
  console.log(result);
  ..... 省略 .....
}

doLongTimeTaskAsync();

>tsc --version
Version 2.0.7
```

```
>tsc --target es6 asyncctest.ts
```

```
>node asyncctest.js
```

```
resolved: task1
```

```
resolved: task2
```

```
resolved: task3
```

TypeScript で async 関数を含むコードをコンパイル／実行

TypeScript でのコンパイル時には「--target」オプション（または「-t」オプション）に「es6」を指定する。これにより、拡張子を「.js」とするファイルが生成される。後は node コマンドを使用して実行できる。

● C# × JavaScript

C# 開発者のための最新 JavaScript 事情 (ジェネレータ関数編)

Insider.NET 編集部 かわさきしんじ (2016 年 02 月 26 日)

ECMAScript 2015 のジェネレータ関数と yield 式を使うと、C# の反復子ブロックと yield return 文と似た形ですっきりと反復処理を記述できる。

ここまでは ECMAScript 2015 (以下、ES2015) の Promise オブジェクトと、次期 ECMAScript で導入予定の async / await キーワードを使った非同期処理の記述方法を見た。ここからは ES2015 で導入されたジェネレータ関数とこれに関する幾つかの構文要素を紹介する。ジェネレータ関数を使うと、C# でいうところの反復子 (イテレータ) と同様な記述が行えるようになる。

C# の反復子

C# の反復子 (イテレータ) とは、「yield return」文および「yield break」文を本体に含んだメソッドやプロパティ (getter) のことで、コレクションを反復的に処理するために使用する (というのは、本フォーラムの読者の方ならよくご存じのことだろう)。

以下に簡単な例を示す。C# では反復子 (iterator) という呼び方が一般的だが、ここでは ES2015 のジェネレータ関数に合わせて sampleGenerator というメソッドを作成している。

```
static IEnumerable<int> sampleGenerator(int from, int to)
{
    while(from <= to)
    {
        yield return from++;
    }
}

static void Main(string[] args)
{
    foreach (var i in sampleGenerator(1, 5))
    {
        Console.WriteLine(i);
    }
}
```



```
Console.ReadKey();  
}
```

C# の反復子

sampleGenerator メソッドを呼び出しても、その本体がすぐに実行されるのではなく、その内容をカプセル化したオブジェクトが返される（メソッドの戻り値の型によって、これは IEnumerator / IEnumerator<T> を実装する「列挙子オブジェクト」か、IEnumerable / IEnumerable<T> を実装する「列挙可能なオブジェクト」のいずれかとなる）。その後、列挙が行われるたびに、カプセル化されたメソッド内の「yield return」文まで実行され、その値と共に制御が呼び出し側にいったん戻される（「yield break」文が実行されると、そこで反復子の実行は終了する）。

上のコード例であれば、foreach 文でまず「列挙可能なオブジェクト」が作成され、その後、列挙が行われるたびに列挙可能なオブジェクトによってカプセル化された sampleGenerator メソッドの「yield return from++;」行までが実行される。この行によって返された値が Main メソッドの変数 i に代入され、それがコンソールに出力されると、再度、列挙が行われ……といった処理を反復子が終了するまで繰り返される。

ES2015 では、ジェネレータ関数により、上記と同様なコードを記述できる。

ES2015 のジェネレータ関数

ジェネレータ関数を定義するには「function」キーワードに続けて「*」を付加する。以下に「function*」宣言と「function*」式の大ざっぱな構文を示す。

```
// function* 宣言によるジェネレータ関数の定義  
function* 関数名 ( パラメーターリスト ) {  
    実行する処理 (yield を含む)  
}  
  
// function* 式によるジェネレータ関数の定義  
var 変数名 = function* ( パラメーターリスト ) {  
    実行する処理 (yield を含む)  
}
```

ジェネレータ関数の構文

以下に上記の C# コードと同様な処理を行う ES2015 コードを示す。

```
function* sampleGenerator(from, to) {  
  while(from <= to)  
    yield from++;  
}  
  
for (var i of sampleGenerator(1,5)) {  
  console.log(i);  
}
```

ES2015 のジェネレータ関数

C# とは異なり「yield return」ではなく「yield」に続けて、呼び出し側に戻す値を記述している。また、for ループでは「in」ではなく「of」が使われている点にも注意してほしい（for-of 文。後述）。

sampleGenerator 関数を呼び出しても、その本体がすぐに実行されるのではなく、その内容をカプセル化したイテレータオブジェクトが返される。その後、列挙が行われるたびにカプセル化されたメソッド内の「yield」式まで実行され、その値と共に制御が呼び出し側にいったん戻される。

上のコード例であれば、for-of 文でまずイテレータオブジェクトが作成され、その後、列挙が行われるたびにカプセル化された sampleGenerator 関数の「yield from++;」行までが実行される。この行によって返された値が変数 i に代入され、それがコンソールに出力されると、再度、列挙が行われ……といった処理を反復が終了するまで繰り返される（この辺のロジックは C# コードの説明で既に話した通りだ）。

for-of ループを使えば、イテレータオブジェクトの処理を隠蔽し、反復処理をスマートに記述できるが、次ページではこのオブジェクトについて少し詳しく見てみよう。

イテレータオブジェクト

ジェネレータ関数呼び出しにより返されたイテレータオブジェクトには next メソッドがある。これを呼び出すことで列挙を一つ進めることになる。例を以下に示す。

```
function* sampleGenerator(from, to) {  
  while(from <= to) {
```

```
        yield from++;
    }
}

var gen = sampleGenerator(1, 5);
var result = gen.next();
console.log("value: " + result.value);
console.log("done: " + result.done);

result = gen.next();
console.log("value: " + result.value);
console.log("done: " + result.done);
```

..... 省略

// 出力結果

```
value: 1
done: false
```

..... 省略

```
value: 5
done: false
value: undefined
done: true
```

イテレータオブジェクトの next メソッド

next メソッドの戻り値は done プロパティと value プロパティを持つオブジェクト（IteratorResult オブジェクト）であり、done プロパティは反復処理が終了したかどうかを、value プロパティは列挙された値を示す。上の出力結果を見ると、数値「5」までの列挙が終わると、次の next メソッド呼び出しでは done プロパティが true に、value プロパティが undefined になり、反復処理が完了したことが分かる。

next メソッドを使うと、while ループを以下のように書くこともできる。ただし、ループの継続条件を next メソッドの戻り値の done プロパティでチェックする上に、その value プロパティを後から使おうとすると、あまりきれいな書き方にはならない（素直に for-of 文を使うのがよいだろう）。

```
var gen = sampleGenerator(1, 5);
var result;
while (!(result = gen.next()).done) {
  console.log(result.value);
}
```

next メソッドを利用したループ

反復処理の実際

一つ目の ES2015 コードでも内部的には next メソッド呼び出しによって反復処理を行っている。例えば以下は npm で babel-cli / babel-latest / babel-polyfill パッケージを導入した環境で、一つ目のコードを JavaScript 5.1 にトランスパイルした結果だ (for-of ループのみ抜粋。改行は筆者が適宜挿入。また、コードの先頭に「import "babel-polyfill";」 行が必要になる)。

```
for (var _iterator = sampleGenerator(1, 5)[Symbol.iterator](), _step;
    !(_iteratorNormalCompletion = (_step = _iterator.next()).done);
    _iteratorNormalCompletion = true) {
  var i = _step.value;

  console.log(i);
}
```

for-of ループをトランスパイルした結果

深くは踏み込まないが、「sampleGenerator(1, 5)[Symbol.iterator]()」の「sampleGenerator(1,5)」は実際には上で見た sampleGenerator 関数ではなく、これをラップする関数だ。参考までに Babel によるジェネレータ関数のトランスパイル結果を以下に示す。大本のジェネレータ関数のコードをトランスパイルした結果は sampleGenerator\$ 関数に含まれている (後述)。

```
function sampleGenerator(from, to) {
  return regeneratorRuntime.wrap(function sampleGenerator$(_context) {
    while (1) {
      switch (_context.prev = _context.next) {
        case 0:
          from;
```

```
    case 1:
      if (!(from < to)) {
        _context.next = 7;
        break;
      }

      _context.next = 4;
      return from;

    case 4:
      from++;
      _context.next = 1;
      break;

    case 7:
    case "end":
      return _context.stop();
  }
}
}, _marked[0], this);
}
```

ジェネレータ関数の内容をラップした sampleGenerator 関数

sampleGenerator 関数の実行時には大本のジェネレータ関数 (sampleGenerator\$ 関数) を利用するための設定が行われる。

ラップした側の sampleGenerator 関数は「iterable」オブジェクト（イテレート可能なオブジェクト）と呼ばれるオブジェクトを返す。このオブジェクトには Symbol.iterator プロパティがあり、これが「イテレータオブジェクト」を返す関数」を参照している。このイテレータオブジェクトの実体は、IteratorResult オブジェクトを戻り値とする next メソッドを持つようにラップされたジェネレータ関数だ。よって「sampleGenerator(1, 5) [Symbol.iterator]()」は全体としてはラップ後のジェネレータ関数を手に入れる処理をしていることになる。

ジェネレータ関数のトランスパイル結果についても簡単に触れておこう（2016 年 11 月時点では生成されるコードは上記のコードとは異なるが、やっていることは大筋としてこれと同じだ）。この中では while による無限

ループを実行しながら、反復処理を行っている。_context の prev プロパティと next プロパティの値は最初は 0 となっているので、「case 0:」節と「case 1:」節をそのまま実行する。

「case 1:」節では、最初に変数 from の値は変数 to よりも小さいので、next プロパティの値を 4 にしてから変数 from の値を返送する。この時点で制御はいったん呼び出し側に戻される（そして、console.log メソッドにより出力される）。

次の繰り返しでは「case 4:」節が実行され、変数 from の値をインクリメントして、next プロパティの値を 1 に設定してループの先頭へと戻る。次のループの「case 1:」節では変数 from の値が変数 to よりも小さいので……といった具合に処理が進められる（結果としては、ジェネレータ関数に記述した内容が実行されるのが分かるはずだ）。

難しいところはさておき、イテレート可能なオブジェクトを IEnumerable / IEnumerable<T> インタフェース（列挙可能なオブジェクト）に相当するもの、イテレータオブジェクトが IEnumerator / IEnumerator<T> インタフェース（列挙子オブジェクト）に相当するものとイメージすると理解が早いかもしれない。ちなみにジェネレータ関数が返すオブジェクトはイテレート可能なオブジェクトでもあり、イテレータオブジェクトでもある。

ジェネレータ関数自体についてはこのくらいにして、次ページではこれと一緒に使われることが多い for-of 文について見ていこう。

for-of 文

for-of 文も ES2015 で導入された構文だ。これはイテレート可能なオブジェクト（コレクション）を対象として、反復処理を行うために使用する。イテレート可能なオブジェクトとは、先ほども出てきた Symbol.iterator プロパティを持つオブジェクトのことであり（Symbol も ES2015 で導入された機能だが、ここでは詳細は割愛する）、ジェネレータ関数以外にも配列や文字列などがこれに該当する。

以下に例を示す。

```
var array1 = [ "foo", "bar", "baz"];
array1.hoge = "hoge hoge";
var obj = { name: "foo", tel: "xxxx" };

// hoge プロパティまで列挙される。また、要素にはインデックス指定でアクセスする
for (var i in array1) {
```

```
    console.log(i + ": " + array1[i]);
  }

  for (var i in obj) {
    console.log(i + ": " + obj[i]);
  }

  // 配列の要素が列挙される
  for (var i of array1) {
    console.log(i);
  }

  // イテレート可能ではないオブジェクトでは for-of 文はエラーとなる
  for (var i of obj) { // エラー!
    console.log(i);
  }
```

for-of 文と for-in 文

Babel の「[Try it out](#)」 ページで試してみると分かるが、最後の for-of 文はエラーとなる。

for-in 文はオブジェクトが持つ列挙可能なプロパティ（の名前）を取り出す。このため、例えば配列にこれを使うとうとすると、配列の要素以外のプロパティまで取り出してしまうなど、使い勝手はよくなかった（もちろん、そのために配列には forEach メソッドがあるわけだが）。配列以外の要素がないにしても、配列に対して for-in で取り出せるのはプロパティ（＝インデックス）なので、実際に要素を扱うには個々の要素をインデックス指定することになるので少々手間がかかる。

これに対して、for-of 文はそのプロパティの値を（配列であれば要素を直接）反復処理できる。ジェネレータ関数と組み合わせれば、コレクションの反復処理がきれいにできるのは上で見た通りだ（興味のある方は for-in 文とジェネレータ関数を組み合わせて実行してみよう）。

最後に文字列の例を見てみよう。文字列も Symbol.iterator プロパティを持つイテレート可能なオブジェクトだ。

```
var str = new String("string");
str[Symbol.iterator] = function() {
  var s = this;
  var l = s.length;
```

```
return {
  next: function() {
    if (l > 0) {
      l--
      return { value: s[l], done: false }
    } else {
      return { done: true }
    }
  }
}

var s = "";
for (var item of str) {
  s += item;
}
console.log(s)
```

文字列を逆順に列挙

Symbol.iterator プロパティを定義する場合、String 型のオブジェクトとして文字列を生成しておく必要がある（基本データ型としての文字列にプロパティを設定しても、アクセス終了後に回収されてしまうため）。

ここでは、String 型のオブジェクトを作成し、その Symbol.iterator プロパティを設定している。つまり、イテレータオブジェクトを変更している。そのため、この関数は next メソッドを持つオブジェクトを返すようになっていて、next メソッドでは文字列の末尾から順に 1 文字ずつ value プロパティの値として返送し（このときは done プロパティの値は false）、最後に done プロパティを true にしたオブジェクトを返送する。これにより、str オブジェクトを反復処理すると、文字列の個々の要素を逆順に取り出せるようになる（出力結果は「gnirts」となる）。

最後に、yield 式について触れておく。

yield 式

すでに見たように、ジェネレータ関数の内部では yield 式によって、呼び出し側に値と制御をいったん戻す。基本はこれなのだが、yield* 式によって別のジェネレータ関数の反復処理を行ったり、yield 式で中断した処理から再開する際に呼び出し側から何らかの値を与えたりできる。

最初に yield* 式のサンプルを見てみよう。

```
function* generator1(from, to) {  
  yield from;  
  yield* generator2(from+1, to-1);  
  yield to;  
}  
  
function* generator2(from, to) {  
  var array = [ "", "i", "ii", "iii", "iv", "v",  
               "vi", "vii", "viii", "ix", "x" ]  
  
  while(from <= to) {  
    yield array[from++];  
  }  
}  
  
for (var i of generator1(1,5)) {  
  console.log(i);  
}
```

yield* 式の使用例

yield* 式は、この式に与えたイテレート可能なオブジェクトを反復処理し、イテレート可能なオブジェクトから返された値を一つずつ呼び出し側に返送するものだ。

このサンプルでは、ジェネレータ関数が二つある。generator1 ジェネレータ関数は指定した範囲の最初と最後の値は自身で yield しているが、その間の値については「yield* generator2(from+1, to-1)」として generator2 ジェネレータ関数を利用している。generator2 ジェネレータ関数では、与えられた範囲のローマ数字を返すようにしている（範囲外の数値が与えられた場合のエラー処理はここでは割愛）。

これを実行すると、「1」「ii」「iii」「iv」「5」が順に表示される。

次に、yield 式に呼び出し側が値を渡す例を見てみよう。

```
function* generator(from, to) {  
  var msg = "";  
  while (from <= to) {  
    if (msg == "terminate") {  
      return;  
    } else if (msg == "skip") {  
      from++; // 変数 from をインクリメントすることでスキップ  
    }  
    msg = yield from++;  
  }  
}  
  
var gen = generator(1,3);  
console.log(gen.next().value);  
console.log(gen.next("terminate").value);  
console.log(gen.next().value)
```

next メソッドに引数を渡すと、それが yield 式の値となる

ここでは next メソッドに文字列「terminate」を渡すと反復処理を終了し、文字列「skip」を渡すと変数 from の値をインクリメントするようにしている。呼び出し側からの値を受け取るには「変数 = yield ……」のような書き方をする。C# では反復子ブロックには return 文を書けないが、ES2015 では return 文に到達するとジェネレータ関数が終了し、return 文に与えた値を value プロパティの値に、done プロパティの値を true にしたオブジェクトが呼び出し側に返送される。

反復処理の継続／終了などを、呼び出し側の要因で決定したいなどの場合には、この形式の yield 式が役に立つだろう。

ここまでは、ジェネレータ関数とこれに関連して for-of 文、yield 式について見た。C# 開発者の方であれば、その概念についても容易に理解できるはずなので、ぜひとも覚えておこう。なお、詳細については [ES2015 の仕様](#)を見てもよいが、[MDN \(Mozilla Developer Network\)](#) の該当ページが分かりやすい。

- [「Iterators and generators」](#) ページ：イテレータオブジェクトとジェネレータ関数に関する概要
- [「function*」](#) ページ：ジェネレータ関数の概要
- [「Iteration protocols」](#) ページ：イテレート可能なオブジェクト、イテレータオブジェクトなどについて

- [「for...of」](#) ページ: for-of 文の概要
- [「yield」](#) ページおよび [「yield*」](#) ページ: yield 式と yield* 式について

● C# × JavaScript

ECMAScript の最新情報を得るには

Insider.NET 編集部 かわさきしんじ (2016 年 03 月 11 日)

ECMAScript の仕様策定の過程と、最新情報を追いかけるのに役立つ Web サイトをいくつか紹介していこう。

最後に、ECMAScript の仕様策定の過程と、最新情報を追いかけるのに役立つ Web サイトをいくつか紹介していこう。

ECMAScript の仕様策定の過程

そもそも ECMAScript (以下、ES) とは、一般に「JavaScript」と呼ばれ、各ベンダーにより実装されているスクリプト言語に共通する仕様を定めたものであり、その名の通り、[Ecma International](#) の [TC39](#) (Technical Committee 39) によって策定が進められている (ECMA-262)。TC39 とは ES とこれに関連する各種の仕様を策定する委員会のことだ。

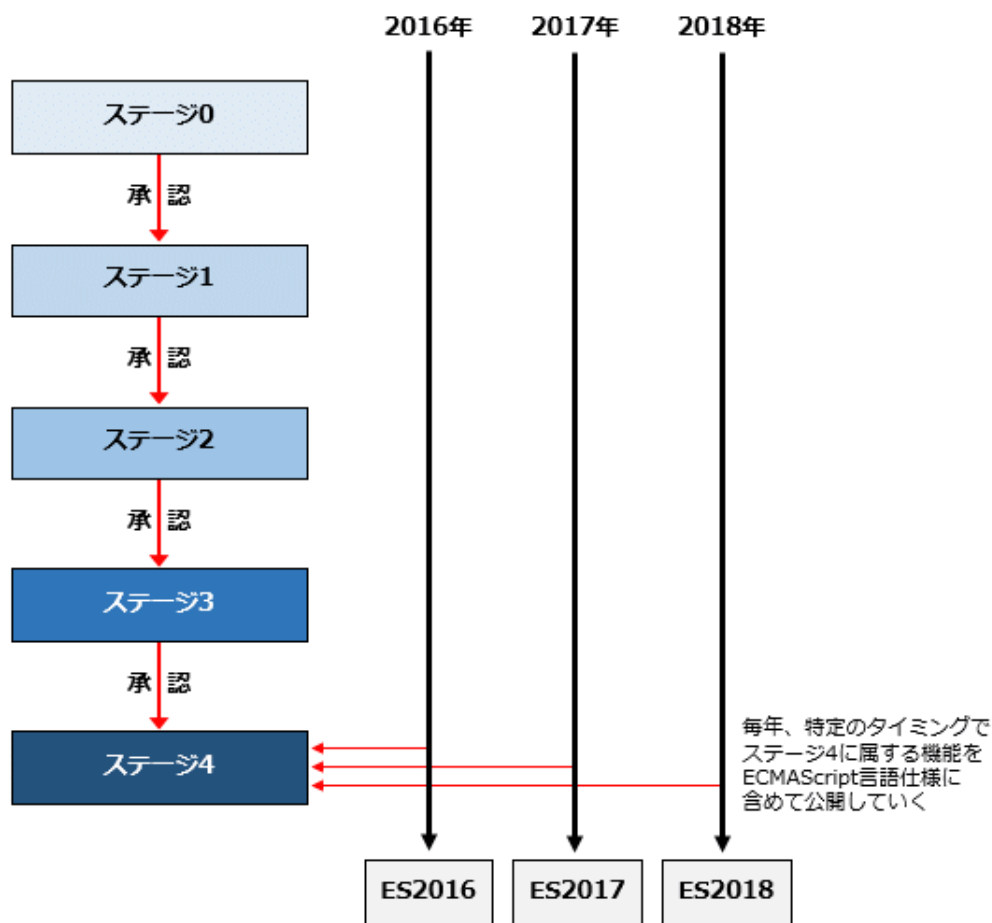
現在では、ES の言語仕様は 1 年に一度のペースでリリースされるようになっている (2016 年 6 月に公開された ECMAScript 2016 がその第 1 弾となる)。

実際の言語仕様の策定では、それに追加する (あるいは変更を加える) 個々の機能ごとに、以下の 5 つのステージに分けて議論が進められていく。次のステージに進むには TC39 による承認が必要になる。

- ステージ 0: Strawman (わら人形)
- ステージ 1: Proposal (提案)
- ステージ 2: Draft (ドラフト)
- ステージ 3: Candidate (候補)
- ステージ 4: Finished (完了)

各ステージで行うこと

各ステージで何が行われるかを簡単に説明しておこう。詳細については「[The TC39 Process](#)」ページや azu 氏のブログ記事「[ECMAScript の仕様策定に関するカンニングペーパー](#)」などを参照されたい。



仕様策定の過程

非公式な形式で提案された機能やアイデアはステージ0となる。ここから、その機能に関する議論を進める責任者の決定、どんな問題をどのように解決するのかや、おおまかなAPIとその使い方の提示、実装上の問題点の洗い出しなどを行い、「[The TC39 Process](#)」ページにあるステージ1の「Entrance Criteria」を満たすとそれはステージ1に進む。

ステージ1ではステージ0で示された問題、その解決策、実装上の問題点などについて議論が行われる。この際にはポリフィル機能やデモを使って議論が行われる。その後、新機能の仕様の最初期ドラフトが作成され、より詳細な精査が可能になるとステージ2になる。ドラフト段階では主要な機能についてはある程度の仕様が固められているが、それ以外の部分はTODOなどの状態になっているもよい。

ステージ2では、TC39によりレビュー者が指名される（レビュー者は仕様の作成者であってではなく、また専門的知識があることが求められる）。レビュー者からのフィードバックを基にドラフトは修正されていく。仕様の全て（構文、セマンティクス、API）が完成して（Complete状態になり）、レビュー者とESの編集者がこれを承認するとステージ3になる。

ステージ3では完成した仕様を基に実装が行われて、実装者／ユーザーの両者からのフィードバックによりさら

なる微修正が行われる(修正は重大なものだけに限られる)。ステージ 4に進むには TC39 が策定している [Test262](#) 受け入れテストの作成、そのテストにパスした実装が **2 つ**、ES の編集者による仕様の承認が必要になる。

ステージ 4 になったものは ES の言語仕様に含まれる準備ができたものとして、年に一度のタイミングで言語仕様に含まれてリリースされる。ES2016 の場合、2016 年 3 月 1 日の時点でドラフトのスナップショットが作成され、6 月の公開に向けて最終的な作業が開始されている。

これを踏まえた上で、「[tc39/ecma262](#)」ページを見てみると、本特集で取り上げた [async 関数](#) は 2016 年 7 月にステージ 4 (Finished) となり、ES2017 仕様へ取り込まれるのを待っている段階となっている ***1**。

このような過程を経て、ES には機能追加あるいは機能変更が行われるので、これから先にどのような機能が追加されるかを知りたいと考えているのなら、どんな機能がどのステージにあるかを注目しておくといよい(関連する URL は次節で紹介しよう)。

ちなみに ES2016 に含まれる新機能は [Array.prototype.includes](#) メソッドと [べき乗演算子](#) の 2 つだけである([このページ](#)を見ると ES2016 に取り込まれたのがこの 2 つであることが分かる)。

***1** azu 氏のブログ「[Web Scratch](#)」の記事「[ECMAScript の仕様策定に関するカンニングペーパー](#)」によれば、実際には仕様が提出される半年くらい前にはそこに含まれる機能はおおよそフリーズされるとのことだ。

ES2016 に取り込まれた 2 つの機能

ES2016 では [Array.prototype.includes](#) メソッドが追加される。これは、次の構文で配列に検索対象 (searchElement パラメーターで指定された値) が含まれているかを調べるものだ。fromIndex パラメーターには検索を開始するインデックスを指定する(省略した場合のデフォルト値は 0。つまり配列の先頭から検索を行う)。

```
Array.prototype.includes (searchElement [, fromIndex])
```

[Array.prototype.includes](#) の構文

以下に使用例を示す。

```
console.log([1, 2, 3].includes(1));           // true
console.log('Insider.NET'.includes('Insider')); // true
console.log('Insider.NET'.includes('Insider', 1)); // false
```

includes メソッドの使用例

最後の例では検索位置に「1」を指定しているので、結果は false になる。

このように includes メソッドは文字列でも使用可能だ。includes メソッドがないと配列や文字列に特定の値（や文字列）が含まれているかを調べるには [indexOf メソッドを使う](#) 必要があった。これには以下のようなコードを記述することになり、一目で何をしているかが分かりづらかった。

```
// indexOf メソッド
console.log([1,2,3].indexOf(1) !== -1); // true

// includes メソッド
console.log([1,2,3].includes(1)); // true
```

indexOf メソッドと includes メソッド

includes メソッドが追加されることで、配列や文字列に特定の要素が含まれているかを判断するコードをスッキリと書けるようになるはずだ。Node.js (V8 JavaScript エンジン) では既にこれを実装している。

[べき乗演算子](#)は「**」で「a ** b」で「a^b」の演算を行う。使用例を以下に示す。特に説明は必要ないだろう。以下は Babel の「[Try it out](#)」ページで試せる。

```
console.log(2 ** 10); // 1024
```

べき乗演算子の使用例

ここまで ES の仕様策定の過程について見てきた。次ページでは ES に関する最新情報をウォッチするのに役立つであろう URL を幾つか紹介していこう。

最新のドラフトと追加機能を把握するには

現在では ES の仕様に関する情報は GitHub 上に集約されている。以下に主な URL を示す。

GitHub および Ecma で公開されている情報

- [GitHub 上の ECMA-262 リポジトリ](#): ステージ 1 からステージ 4 の状態にある提案はこのページで確認できる。
- [ES の最新ドラフト](#): リンクをクリックすると分かるが、これは既に 2017 年にリリース予定の ES2017 のドラフトとなっている。また、ドラフトの変更点を継続的に追いかけるには[このページ](#)をチェックするのがよい
- [ES2016 のドラフト](#): こちらは ES2016 のドラフト（ただし、URL から想像してアドレスバーに「~/ecma262/2015/」と入力しても ES2015 のドラフトは表示されないので注意）
- [仕様策定過程](#): 仕様策定の過程がどうなっているかの詳細はこのページで説明されている
- [ステージ 0 の提案](#): ステージ 0 の状態の提案はこのページで一覧できる
- [TC39 のミーティングのログ](#): TC39 のミーティングでどんな話し合いが行われたか、各提案のステージがどうなったかを調べるにはこのページから各ミーティングのログを参照しよう
- [ES2015 の言語仕様](#): 策定された言語仕様については Ecma の Web サイトで公開されている

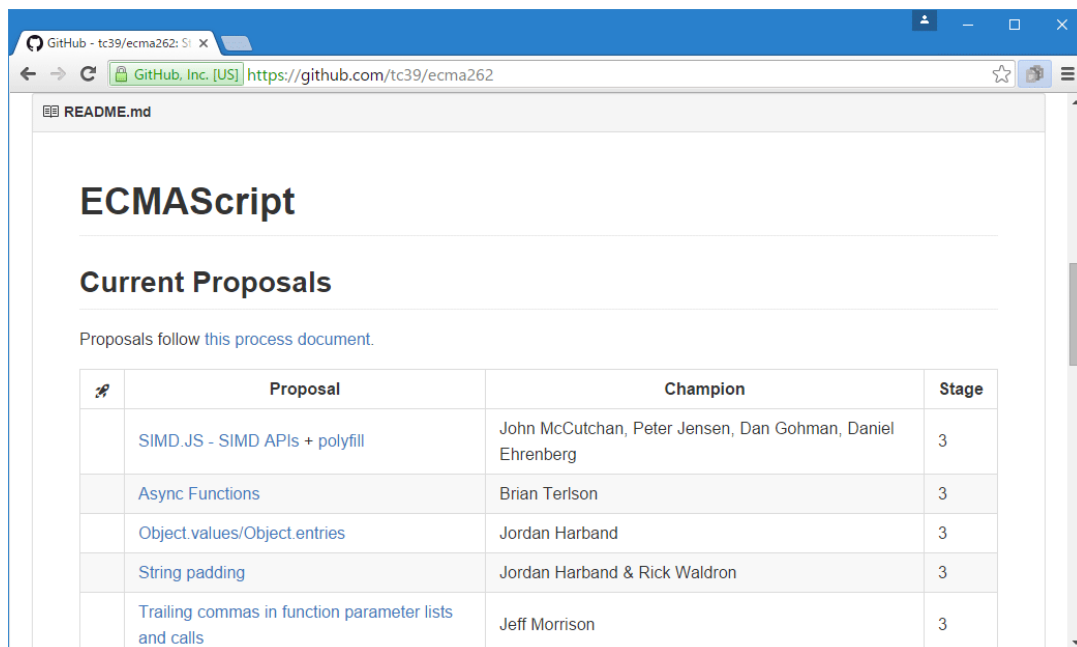
重要なのは、ES では言語仕様が継続的に管理されているという点だ。冒頭にも述べたが、ES の言語仕様は 1 年に一度のペースでリリースされる。そして、ステージ 4 になった機能は直近のリリースに含まれる。TC39 では常にその最新仕様（に追加する機能や変更）についての議論が行われ、年に一度のタイミングで新機能を含んだバージョンが正式な言語仕様として公開されると考えよう。



ES2017 のドラフト

「もう ES2017 ?」 と思うかもしれないが、最新のドラフトは ES2017 のものになるということだろう。

ステージ 0 ～ステージ 4 の各状態にある提案は、上記のページを見れば分かる。その詳細については、各提案のリンクをクリックすればよい。近い将来にどんな機能が言語仕様に追加されるかを予習したければ、ステージ 3 の提案を中心に調べてみよう。



ECMAScript

Current Proposals

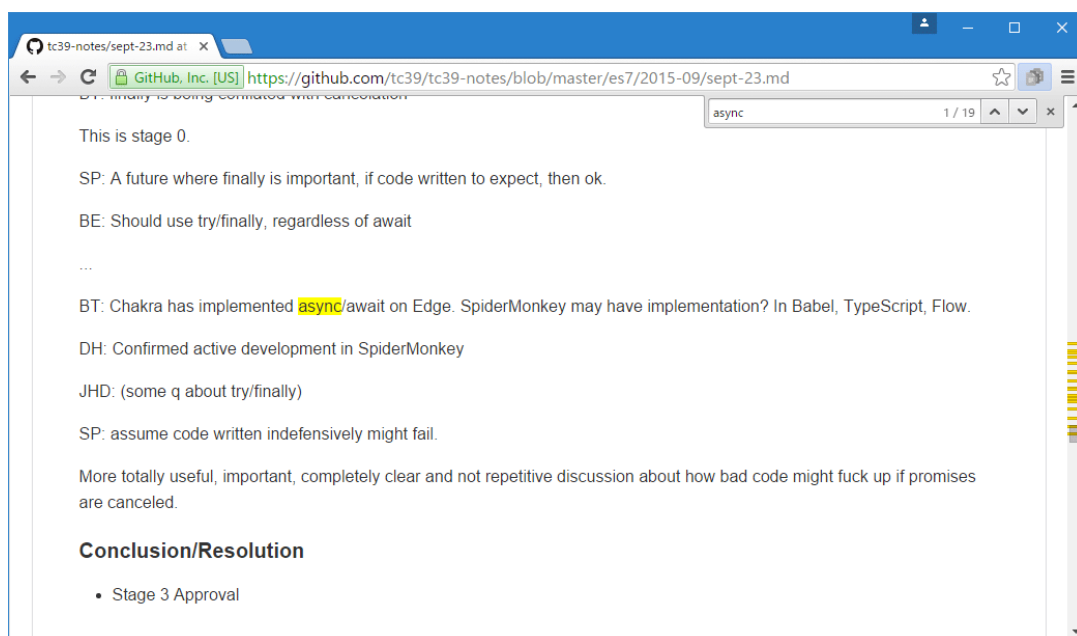
Proposals follow [this process document](#).

	Proposal	Champion	Stage
	SIMD.JS - SIMD APIs + polyfill	John McCutchan, Peter Jensen, Dan Gohman, Daniel Ehrenberg	3
	Async Functions	Brian Terlson	3
	Object.values/Object.entries	Jordan Harband	3
	String padding	Jordan Harband & Rick Waldron	3
	Trailing commas in function parameter lists and calls	Jeff Morrison	3

ステージ 1 ～ステージ 3 にある提案の一覧

本特集でも紹介した「Async Functions」はまだステージ 3 だ。

ある機能が次のステージに進んだかどうかは、上に示した [TC39 のミーティングログ](#) を見ていく。例えば、[2015 年 9 月 23 日のログ](#) を参照していくと、`async` 関数がステージ 3 に進んでもよいと認証されたことが分かるはずだ（同じく、[2015 年 11 月 17 日のログ](#) ではステージ 4 へは認証されなかったことも分かる）。



tc39-notes/sept-23.md at master · tc39/tc39-notes · GitHub

1 / 19

This is stage 0.

SP: A future where finally is important, if code written to expect, then ok.

BE: Should use try/finally, regardless of await

...

BT: Chakra has implemented `async/await` on Edge. SpiderMonkey may have implementation? In Babel, TypeScript, Flow.

DH: Confirmed active development in SpiderMonkey

JHD: (some q about try/finally)

SP: assume code written indensively might fail.

More totally useful, important, completely clear and not repetitive discussion about how bad code might fuck up if promises are canceled.

Conclusion/Resolution

- Stage 3 Approval

`async` 関数がステージ 3 に進むことが認証されたログ

最下行に「Stage 3 Approval」とあるのが分かる。

なお、「[Ecma TC39](#)」ページを見ると分かるが、TC39 では ES 本体の仕様策定以外にも、ES 仕様に準拠しているかを調べるテストスイート ([test262](#))、ES の国際化 API ([ECMA-402](#)) など手掛けている。興味のある方はこれらについてもチェックしておこう。

その他のリソース

上では Ecma が公式に開いているページを紹介した。ここではそれ以外に見ておいた方がよいページを紹介しよう。

その他のリソース

- [Web Scratch](#): 先ほども紹介した azu 氏によるブログ (本稿を書く際にも確認のために参考にさせてもらった)。azu 氏は以下で紹介する「[JSer.info](#)」の運営／管理も行っている
- [JSer.info](#): 週に一度のペースで JavaScript の最新情報を紹介してくれている。[リアルタイム版](#)もある。最新情報を知りたい方は定期的にチェックをしておこう
- [Stack Overflow](#): 困ったときにはのぞいてみるとよいかもしれない。[日本語版もある](#)が、情報の集積度は本家の方がさすがに多い。英語を苦にしない人であれば本家を参照するのがオススメだ
- [HTML Experts.jp](#): HTML5 / ES / CSS でアプリ開発を行うのに役立つ情報が豊富に掲載されている
- [POSTD](#): 海外で話題のブログ記事やニュースを翻訳している。ES に限らず、開発者向けの濃い話が掲載されるので、興味ある記事はチェックするようにしよう