CoffeeScript と Node.js による関数型の JavaScript

関数型のスクリプトによって Web アプリケーションの複雑さを克服する

Andrew Glover

2012年 3月 22日

CoffeeScript は JavaScript をより扱いやすくした言語として知られていますが、他にも詳しく調べる価値のある優れた点があります。この記事では著者の Andrew Glover が、CoffeeScript の簡潔な構文によって JavaScript ライブラリーの関数型の構成体を容易に活用できること、特に Node.js を使用したサーバー・サイドのプログラミングで有効なことを説明します。そして最後に、JavaScript のユーティリティー・ライブラリーである Underscore.js を使用して CoffeeScript と Node.js のコレクションを処理する方法について、一連の簡単な例を示しながら説明します。

CoffeeScript は比較的新しい言語であり、JavaScript の欠陥にうんざりしている開発者にとっては魅力的な代替手段です。CoffeeScript を使用すると、Ruby と Python とを合わせたような感じの軽量かつ直感的な言語でコーディングすることができます。CoffeeScript で記述したコードをコンパイルすると JavaScript が生成され、その JavaScript はブラウザーに表示される Web アプリケーションに使用することができます。またコンパイルされた JavaScript はサーバー・サイド・アプリケーションのための Node.js でもそのままシームレスに実行することができます。この記事では、CoffeeScript を使用すると JavaScript の関数型の側面を扱えるようになるという、CoffeeScript が持つメリットに焦点を当てます。最近のプログラミング言語の傾向を取り入れた CoffeeScript の簡潔な構文により、JavaScript ライブラリーに隠されている関数型プログラミングの世界が開けてきます。

主な言語での関数型プログラミング

主なプログラミング言語 (Java 言語、C++、C# など) のなかで、明確に関数型プログラミング言語であると言えるものはありませんが、どの言語もアドオン・ライブラリーやフレームワークによって (程度に差はあるものの) 関数型プログラミングをすることができます。さらに重要な点として、関数型プログラミングは複雑なアプリケーションでもバグの発生が少なく、生産性が向上するという理由から、Clojure、F#、Erlang などの言語が主流になりつつあります。

JavaScript と同様に、関数型プログラミングは非常に有用である一方で、これまであまり一般的ではありませんでした。JavaScript はおもちゃの言語であると見なされ、関数型プログラミングはあまりにも複雑であるという評判が立ってしまいました。しかし、並行性の高いアプリケーションの需要が高まるにつれ、状態が変化する命令型プログラミング・スタイルに代わる手段へのニーズも高まってきました。実は関数型プログラミングは、言われているほど複雑なわけではなく、

ある特定のタイプのアプリケーションに特有の複雑な動作をコード化するためのスマートなツールなのです。

この記事では、Underscore.js という JavaScript ライブラリーを利用して CoffeeScript と Node.js で関数型スクリプトを作成する方法について説明します。この 3 つの技術を組み合わせると、関数型プログラミングを活用するサーバー・サイド・アプリケーションやブラウザー・ベースのアプリケーションを JavaScript を使用して開発するための強力なスタックを構成することができます。

この記事は、私が以前に Java 開発者向けに執筆した入門記事「Java 開発 2.0: Java 開発者のための JavaScript」と「Java 開発者のための Node.js」がベースになっています。ここでは、皆さんの開発環境に Node.js が含まれ、皆さんが Node.js による基本的なプログラミングを理解しているという前提で説明します。

CoffeeScript と Node.js のセットアップ

Node.js が既に開発環境に含まれている場合には、Node.js のパッケージ・マネージャー (NPM) を使用して CoffeeScript をインストールすることができます。以下のコマンドは CoffeeScript パッケージをグローバルにインストールするように NPM に指示しています。

```
$> npm install -g coffee-script
```

CoffeeScript を使用する場合の主な作業は、プログラムを作成すること、作成したプログラムを .coffee ファイルに保存すること、そして保存した結果を JavaScript にコンパイルすることです。CoffeeScript の構文は JavaScript の構文に似ているため、ほとんどの開発者にとって理解しやすいはずです。例えば、リスト 1 の CoffeeScript は JavaScript とよく似ていますが、おなじみの丸括弧やセミコロンはあまり使われていません。

リスト 1. 典型的な CoffeeScript

```
$> coffee -bpe "console.log 'hello coffee'"
console.log('hello coffee');
```

`coffee` コマンドは何らかの管理タスクの短縮形です。coffee コマンドは CoffeeScript ファイルを JavaScript にコンパイルし、CoffeeScript ファイルを実行し、さらには対話型の環境、つまり (Ruby の `irb` に似た) REPL としても動作します。

ここで、このスクリプトをファイルにします。

```
console.log "hello coffee"
```

次に、このファイルを JavaScript にコンパイル (つまり変換) します。

```
$> coffee -c hello.coffee
```

その結果、hello.js というファイルが生成されます。生成された JavaScript は Node.js で使用可能なため、この JavaScript を即座に Node.js 環境で実行することができます。

リスト 2. Node.js で JavaScript を実行する

```
$> node hello.js
hello coffee!
```

あるいは、リスト 3 のように `coffee` コマンドを使用して元の .coffee ファイルを実行することもできます。

リスト 3. Node.js で CoffeeScript を実行する

```
$> coffee hello.coffee
hello coffee!
```

watchr ユーティリティー

オープンソース・コミュニティーにより、テストの実行やコードのコンパイル等々の作業をするための便利なファイル監視ユーティリティーがいくつも作成されています。これらのユーティリティーは通常、コマンドラインで実行され、非常に軽量です。ここでは、開発環境内のすべての .coffee ファイルの監視を行い、それらのファイルが保存されるとコンパイルが実行され、.js ファイルが生成されるように、監視ツールを構成します。

そのためのユーティリティーとして私が気に入っているものは、Ruby ライブラリーの `watchr` です。`watchr` を使用するには、開発環境に Ruby と RubyGems をインストールしておく必要があります。これらがインストールできたら、以下のコマンドを実行し、グローバルな Ruby ライブラリーとして (対応するユーティリティーも含めて) `watchr` をインストールします。

```
$> gem install watchr
```

`watchr` では正規表現を用いて、監視対象のファイルと、それらのファイルにどのような処理を行うかを定義します。以下のコマンドは `src` ディレクトリーにあるすべての .coffee ファイルをコンパイルするように `watchr` を構成します。

```
watch('src\/.*\.coffee') {|match| system "coffee --compile --output js/ src/"}
```

この場合の `coffee` コマンドは、コンパイルによって生成される .js ファイルが `js` ディレクトリーに保存されるように指定していることに注意してください。

watchr を起動するには、ターミナル・ウィンドウで以下のように入力します。

```
$> watchr project.watchr
```

これで、`src` ディレクトリーのどの .coffee ファイルを変更した場合にも、`watchr` によって新しい `.js` ファイルが必ず生成され、`js` ディレクトリーに配置されるようになります。

CoffeeScript の概要

CoffeeScript には数多くの有用な特徴があり、JavaScript よりも扱うのがはるかに容易です。主なところでは、波括弧、セミコロン、`var` キーワード、`function` キーワードは CoffeeScript には必

要ありません。実際、CoffeeScript の数ある特徴の中で私が気に入っているのは、リスト 4 のように関数を定義できることです。

リスト 4. CoffeeScript では関数の定義が容易です

```
capitalize = (word) ->
  word.charAt(0).toUpperCase() + word.slice 1

console.log capitalize "andy" //prints Andy
```

ここでは、CoffeeScript で単語の先頭の文字を大文字にする単純な関数を定義しています。CoffeeScript で関数を定義するための構文では、矢印に続いて関数の内容を定義します。また関数の本体は空白で区切られるため、CoffeeScript には波括弧がありません。また、多くの言語で丸括弧が使われているところに丸括弧がないことにも注意してください。上記 CoffeeScript の `word.slice 1` はコンパイルされると、単に JavaScript の `word.slice(1)` になります。そして先ほどと同じく、関数の本体が空白で区切られることに注意してください。関数を定義する行の下にインデントされているすべてのコードが関数本体です。その行の下にある、インデントされていない `console.log` は、関数の定義が完了したことを意味します。(この、関数を定義する構文に波括弧がなく、関数本体がインデントされているという 2 つの特徴は、それぞれ Ruby と Python の特徴が CoffeeScript にも取り入れられたものです。)

よく理解できない人のために示すと、リスト 4 に対応する JavaScript 関数はリスト 5 のようなものになります。

リスト 5. たった 1 行の関数でも JavaScript で記述するのは面倒です

```
var capitalize = function(word) {
  return word.charAt(0).toUpperCase() + word.slice(1);
};

console.log(capitalize("andy"));
```

変数

どのような変数が定義された場合も、CoffeeScript ではコンパイル時にその変数の前に JavaScript のための `var` を自動的に追加します。そのため、CoffeeScript でコーディングする際は `var` を付けるように注意する必要がありません。(JavaScript では `var` キーワードはオプションです。`var` を付けないと変数はグローバルになり、グローバル変数は、ほとんど必ず望ましくない結果を招きます。)

また CoffeeScript ではパラメーターのデフォルト値を定義することもできます (リスト 6)。

リスト 6. パラメーターのデフォルト値

```
greeting = (recipient = "world") ->
  "Hello #{recipient}"

console.log greeting "Andy" //prints Hello Andy
console.log greeting()      //prints Hello world
```

リスト 7 は、このパラメーターのデフォルト値が、リスト 6 に対応する JavaScript でどう処理されるかを示しています。

リスト 7. 記述が簡潔でない JavaScript

```
var greeting;

greeting = function(recipient) {
 if (recipient == null) recipient = "world";
 return "Hello " + recipient;
};
```

条件分岐

CoffeeScript では、リスト 8 のように `and`、`or`、`not` などのキーワードを導入することによって条件分岐を処理します。

リスト 8. CoffeeScript の条件分岐

```
capitalize = (word) ->
 if word? and typeof(word) is 'string'
  word.charAt(0).toUpperCase() + word.slice 1
 else
  word

console.log capitalize "andy"   //prints Andy
console.log capitalize null     //prints null
console.log capitalize 2        //prints 2
console.log capitalize "betty"  //prints Betty
```

リスト 8 では `?` 演算子を使用して存在を確認しています。単語の先頭文字を大文字にしようとする前に、このスクリプトでは `word` というパラメーターが `null` ではないこと、そしてそのパラメーターが `string` 型であることを確認しています。CoffeeScript では `==` の代わりに `is` が使えるので非常に便利です。

関数型プログラミングのためのクラス定義

JavaScript はクラスを直接サポートしていません。つまり、JavaScript はクラス・ベースの言語ではなく、プロトタイプ・ベースの言語です。オブジェクト指向プログラミングが染みついている人にとっては、このプロトタイプ・ベースであることによって混乱を招きがちです。私達にはクラスが必要なのです。私達を満足させるために、CoffeeScript には `class` 構文が用意されています。この class 構文により、標準的な JavaScript にコンパイルされる際に、関数の中に一連の関数が定義されます。

リスト 9 では、`class` キーワードを使用して `Message` というクラスを定義しています。

リスト 9. もちろん、CoffeeScript にはクラスがあります

```
class Message
 constructor: (@to, @from, @message) ->

 asJSON:  ->
  JSON.stringify({to: @to, from: @from, message: @message})

mess = new Message "Andy", "Joe", "Go to the party!"
console.log mess.asJSON()
```

リスト 9 では `constructor` キーワードを使用してコンストラクターを定義しています。そして名前を入力した後に関数を入力することで、メソッド (`asJSON`) を定義しています。

CoffeeScript と Node.js

CoffeeScript をコンパイルすると JavaScript になるため、CoffeeScript は Node.js によるプログラミングに自然な形でフィットし、既に簡潔な Node.js のコードをさらに洗練されたコードにするために、特に役立ちます。単純にコードを比較するとわかるように、CoffeeScript は Node.js で多用されるコールバックを整理することに極めて長けています。リスト 10 は、純粋な JavaScript を使用して Node.js による単純な Web アプリケーションを定義したものです。

リスト 10. Node.js による Web アプリケーションを JavaScript で定義する

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

var app = express.createServer(express.logger());

app.put('/', function(req, res) {
  res.send(JSON.stringify({ status: "success" }));
});

var port = process.env.PORT || 3000;

app.listen(port, function() {
  console.log("Listening on " + port);
});
```

上記と同じ Web アプリケーションを CoffeeScript で作成し直すと、Node.js のコールバックによる余分な構文要素がなくなります (リスト 11)。

リスト 11. CoffeeScript によって Node.js を単純化する

```
express = require 'express'

app = express.createServer express.logger()

app.put '/', (req, res) ->
  res.send JSON.stringify { status: "success" }

port = process.env.PORT or 3000

app.listen port, ->
  console.log "Listening on " + port
```

リスト 11 では、JavaScript の `||` の代わりに `or` 演算子を追加しています。また私は、`app.listen` の匿名関数を示すには、矢印を使用する方が `function()` を入力するよりも容易なことにも気付きました。

CoffeeScript の表現は理解が容易

ここまで読むと、CoffeeScript では抽象的な記号ではなく通常の英語の表記が使用されることに気付いたと思います。CoffeeScript では `!==` と入力する代わりに、もっと直感的な `isnt` を使用します。同様に、`===` は `is` になります。

このファイルに対して `coffee -c` を実行すると、リスト 10 で定義される JavaScript が、ほぼそのまま CoffeeScript によって生成されることがわかります。CoffeeScript が生成する JavaScript は 100% 有効であり、どのような JavaScript ライブラリーとでも動作します。

関数型のコレクションと Underscore.js

JavaScript プログラミングのための関数型ユーティリティー集とも言われる Underscore.js は、JavaScript による開発を容易にするための関数ライブラリーです。Underscore.js は何よりも、それぞれ特定のタスクに最適なコレクション指向の関数を大量に提供しています。

例えば、ある数の集合から奇数をすべて見つける必要があるとします。一例として、0 から 9 までの数の集合について考えてみましょう。この場合、CoffeeScript と Underscore.js を組み合わせて使用すると、コマンドの入力を多少減らすことができ、おそらくバグもいくらか減らすことができます。リスト 12 は基本的なアルゴリズムを示していますが、Underscore.js が集約関数 (この場合は `filter`) を提供しています。

リスト 12. Underscore.js による filter 関数

```
_ = require 'underscore'

numbers = _.range(10)

odds = _(numbers).filter (x) ->
 x % 2 isnt 0

console.log odds
```

まず、`_` (つまりアンダーバー) は有効な変数名なので、Underscore.js ライブラリーを参照するように _ を設定しています。次に、奇数かどうかをテストする匿名関数を `filter` 関数に追加しています。ここでは JavaScript の `!==` ではなく CoffeeScript の `isnt` キーワードを使用していることに注意してください。次に、0 から 9 までの数をソートしたいということを `range` 関数を使用して指定しています。あるいは、この範囲に対するステップ・カウントを提供し (つまり 2 つずつカウントし)、任意の数から開始する方法もあります。

`filter` 関数は、渡されたもの (この場合は `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`) をフィルタリングした結果の配列を返します。つまりリスト 12 のコードを実行すると、`[1, 3, 5, 7, 9]` が得られます。

`map` 関数も、JavaScript のコレクションに適用される関数として私が好きなものの 1 つです (リスト 13)。

リスト 13. Underscore.js の map 関数

```
oneUp = _(numbers).map (x) ->
 x + 1

console.log oneUp
```

この場合、出力は `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]` のようになります。基本的に、Underscore.js の map 関数は、引数 (この場合は `numbers`) として渡されたコレクションの各値を 1 つずつ順番に処理してくれるため、各整数に対して繰り返し処理するコードを別途作成する必要がありません。

コレクションのさまざまな側面をテストする必要がある場合、Underscore.js を使用するとテストが非常に簡単になります。例えば単純にリスト 14 のような関数を作成すると、各値が偶数であるかどうかをテストすることができます。

リスト 14. Underscore.js の even 関数

```
even = (x) ->
 x % 2 is 0

console.log _(numbers).all(even)
console.log _(numbers).any(even)
```

上記のように定義した even 関数は、all や any といった Underscore.js の関数の引数として簡単に指定することができます。この場合、all は numbers で渡されたコレクションの各値に even 関数を適用します。次に all は、すべての値が偶数であったかどうかを示すブール値 (この例では false) を返します。同様に、any 関数は、いずれかの値が偶数である場合にブール値 (この例では true) を返します。

さらに Underscore.js を活用する

この記事では Underscore.js が持つ機能の表面的な部分のみを紹介しています。Underscore.js が持つ他の機能として、関数バインディング機能、JavaScript テンプレート機能、オブジェクトの等価性判定機能などがあります (「参考文献」を参照)。

これらの関数をコレクションに適用する必要はないが、何か別のことをコレクションに対して行う必要がある場合はどうすればよいのでしょう？その場合でも、問題ありません。Underscore.js の each 関数を活用すればよいのです。each は手軽なイテレーターとして動作します (つまり裏でループ・ロジックを処理し、繰り返しのたびに、指定された関数を渡します)。Ruby や Groovy を扱ったことがある人であれば、each 関数はおなじみのはずです。

リスト 15. Underscore.js の each 関数

```
_.each numbers, (x) ->
 console.log(x)
```

リスト 15 で、each 関数は引数として、コレクション (numbers 範囲) と、繰り返し処理対象の配列の各値に適用される関数を取ります。この場合は each を使用して現在の繰り返し処理の値をコンソールに出力しています。これと同じように、データベースにデータを永続化したり、ユーザーに結果を返したり、等々の処理も容易に行うことができます。

まとめ

CoffeeScript により、JavaScript プログラミングは新たなもの、そして容易なものになります。Ruby や Python に慣れた人達にとっては特に、CoffeeScript が使い慣れたもののように思えるはずです。この記事では、これらの各言語の特徴を取り入れた CoffeeScript を使用すると、JavaScript スタイルのコードを読みやすく、はるかに短時間で作成できることを説明しました。この記事で示したように CoffeeScript、Node.js、Underscore.js を組み合わせると、基本的な関数型プログラミング・シナリオのコード・スタックが、信じられないほど軽量な上に楽しく作成できるものになります。時間をかけて練習し、この記事で学んだことを拡張することで、Web やモバイルによって動的なやりとりが行われる、もっと複雑なビジネス・アプリケーションも容易に作成できるようになります。