Java 開発者のための Node.js

Web アプリケーションのための軽量のイベント駆動型 I/O

Andrew Glover
CTO
App47

2012年 1月 13日

Node.js は、Java プラットフォームの標準的なマルチスレッド手法を単一スレッドのイベント駆動型 I/O に置き換えることによって並行処理の問題を解決します。この記事では、Andrew Glover が Node.js について紹介し、そのイベント駆動型の並行処理が、筋金入りの Java 開発者たちの興味さえもかきたてている理由を説明します。さらに、Node.js の Express フレームワーク、Mongolian DeadBeef、そして MongoDB を利用して、並行処理の能力、スケーラビリティー、永続性を兼ね備えた Web アプリケーションを構築する方法を説明します。

JavaScript は、Web アプリケーション開発においてこれまで過小評価されてきたヒーローとして、ここ数年で評価を高めてきました。この高く評価されるようになった JavaScript は、それまで「おもちゃの言語」として JavaScript に取りあわずにきた多くのソフトウェア開発者を驚かせてきました。(開発者たちが声を大にしてその言語に対する忠誠を誓っているという意味で) JavaScript よりも定評のある言語は他にもありますが、ブラウザーに依存しない標準スクリプト言語という独特の立場により、JavaScript がその座を奪われることはありませんでした。クライアント・サイドの Web 開発には、おそらく JavaScript が世界で最もよく使用されている言語でしょう。

Java 開発者のための JavaScript

JavaScript は、今日の Java 開発者にとって重要なツールです。この言語を習得するのは難しくありません。JavaScript の変数、型、関数、およびクラスを含め、ファーストクラスの Web アプリケーションを構築するために必要な構文を紹介している Andrew Glover の**記事を読んでください**。

JavaScript は、サーバー・サイドのスクリプティングでも使われていて、その活躍の範囲は広がりつつあります。過去にもサーバー・サイドの JavaScript への試みはありましたが、これまでにないほどの話題を呼んでいるのが、Node.js です。

スケーラブルなネットワーク・プログラムを作成する開発者を支援する目的で設計された Node.js は、サーバー・サイドのプログラミング環境で、まったく新しい世代の開発者向けに JavaScript を作り変えたと言ってもよいほどです。多くの Java 開発者にとって Node.js の最大の魅力となっているのは、ソフトウェアの並行処理に対するその新鮮な手法です。Java プラットフォームは並行処理に対する手法を進化させ続けているものの (Java 7 および Java 8 での大きな進化が期待されて

います)、Node.js は並行処理の必要を満たすというだけでなく、最近の Java 開発者の多くが採用するようになっている、より軽量の方法で並行処理の問題に対処します。JavaScript でのクライアント・サイドのスクリプティングと同じく、Node.js 環境でのサーバー・サイドのスクリプティングには非常に優れた点があります。それは、機能するというだけでなく、多くの Java 開発者が現在問題を抱えている領域で功を奏するからです。

この記事では、サーバー・サイドのスクリプティングを革新する Node.js を紹介します。Node.js を特別なものにしているアーキテクチャーの概要から説明し、その後、データ・パーシスタンスとして MongoDB を活用するスケーラブルな Web アプリケーションをいかに簡単に構築できるかを示します。Node.js の面白さ、そして Node.js を使用すれば実用的な Web アプリケーションを素早くアセンブルできることを実際に目で確かめてください。

Node.js のイベント駆動型の並行処理

Node.js は Google の V8 JavaScript エンジンをベースにした、スケーラブルなイベント駆動型 I/O 環境です。Google V8 は、実際には JavaScript をネイティブ・マシン・コードにコンパイルしてから実行するため、通常、JavaScript は高速なパフォーマンスとは結び付きませんが、Google V8 は極めて高速なランタイム・パフォーマンスを実現します。したがって、Node.js を使用すれば、高い並行処理能力を持ち、電光石火の如く高速な動作をするネットワーク・アプリケーションを迅速に構築できるというわけです。

イベント駆動型 I/O という言葉は、Java 開発者には聞き慣れないかもしれませんが、これはまったく新しい概念ではありません。Node.js では、開発者が Java プラットフォームで使い慣れているマルチスレッド・プログラミング・モデルを使用するのではなく、単一スレッドの並行処理手法を使用し、イベント・ループで強化します。この Node.js 構成体が、ノンブロッキング I/O または非同期 I/O を可能にします。通常の呼び出しは、例えばデータベースに対するクエリーの実行結果が返されるのを待機するなどして他の処理をブロックしますが、Node.js での呼び出しはブロックしません。Node.js アプリケーションは、処理コストの高い I/O アクティビティーが完了するまで待機する代わりにコールバックを発行します。リソースが返されると、アタッチされたこのコールバックが非同期で呼び出されます。

Node.js を選ぶ理由とは

エンタープライズ開発の主役としての Java プラットフォームの地位を確立する助けとなったのは、その並行処理手法です。Java プラットフォームがエンタープライズ開発の主役であることは、今後も変わりはないでしょう。Netty (および Gretty。「参考文献」を参照) などのフレームワーク、そして NIO や `java.util.concurrent` などのコア・ライブラリーが理由となり、並行処理に対処するための第一の選択肢として JVM が選ばれてきました。その一方、Node.js の何が特別なのかと言うと、Node.js は並行処理プログラミングの問題を解決するように特別に設計された最新の開発環境であることです。Node.js のイベント駆動型プログラミング・パラダイムでは、ライブラリーを追加しなくても並行処理が機能します。これは、マルチコア・ハードウェアに注目する開発者にとって嬉しい知らせです。

Node.js プログラムでは、他の助けがなくても並行処理が機能します。もし Java プラットフォームで前述のシナリオを実行しようと思ったら、従来のスレッドや Java NIO の新しいライブラリー、さらには改善内容を盛り込んで更新された `java.util.concurrent` パッケージなど、複雑で時間のかかる手法を選ぶことになります。Java による並行処理は強力ではありますが、簡単には理解することができません。つまり、コードが難解になるということです。これに対し、Node.js のコールバック・メカニズムは言語に組み込まれるため、`synchronized` のような特別な構成体がなくて

も並行処理が機能します。Node.js の並行処理モデルは極めて単純なので、より多くの開発者が利用することができます。

Node.js の JavaScript 構文も、入力の手間を大幅に省いてくれます。ほんのわずかなコードだけで、無数の同時接続に対応できる、高速かつスケーラブルな Web アプリケーションを構築することができます。Java プラットフォームでもこのような Web アプリケーションを構築することはできますが、それにはさらに多くのコードと、数々の追加ライブラリーおよび構成体が必要です。Node.js を使用するための新しいプログラミング環境を操作する心配をしているとしたら、その必要はありません。ある程度の JavaScript の知識があれば、Node.js は簡単に習得することができます。これは、私が断言します。

Node.js の導入

前述のとおり、Node.js を使ってみるのは簡単です。インターネットには数々の優れたチュートリアルも揃っています。この記事では (「Java 開発 2.0: Java 開発者のための JavaScript」と同じく)、Java 開発者が Node.js を最大限利用できるように手助けすることに重点を置きます。そこで、いつもの「Hello, world」Web サーバー・アプリケーションを構築する手順を説明する代わりに、早速、実際に役立つ有意義なアプリケーションに取り掛かります。このアプリケーションは、Node.js をベースに構築する foursquare のようなアプリケーションだと思ってください。

ビデオ・デモ: 「Getting started with Node.js」

この便利なフレームワークを別の方法で使い始めたいと思ったら、同じく Andrew Glover によるこのビデオ・デモを見てください。Node.js の概要と、開発者に役立つ機能をさらに学ぶことができます (ビデオ・デモの説明の内容を読むこともできます)。

Node.js をインストールするには、お使いのプラットフォームに対応した手順に従う必要があります。OSX などの UNIX ライクなプラットフォームを使用している場合には、NVM (Node Version Manager) を使用することをお勧めします。NVM が、Node.js の適切なバージョンをインストールする際の詳細を処理してくれます。いずれにしても、今すぐ Node.js をダウンロードしてインストールしてください。

このアプリケーションを構築するには、いくつかのサード・パーティー・ライブラリーも使用するので、Node.js のパッケージ・マネージャーである NPM (Node Package Manager) をインストールしておくとよいでしょう。NPM でプロジェクトの依存関係をバージョンと一緒に指定すると、該当する依存関係がダウンロードされてビルド・パスに組み込まれます。NPM は多くの点で、Java プラットフォームでの Maven、あるいは Ruby の Bundler と似ています。

Node.js の Express フレームワーク

Node.js は、並行処理の問題に対処できること、そして Web 開発を念頭に作成されていることから、Web 開発者に利用されています。最もよく使われているサード・パーティー製 Node.js ツールの 1 つに、Express という軽量の Web 開発フレームワークがあります。この記事では、Express を使ってアプリケーションを開発します (Express についての詳細は、「参考文献」を参照してください)。

Express には、高度なルーティング、動的テンプレート・ビュー (はやりの Node.js フレームワークである Jade を参照)、コンテンツ・ネゴシエーションをはじめ、多数の機能が満載されています。

また、Express はかなり軽量になっていて、フレームワークを重くする ORM などのお荷物は組み込まれていません。そのため、Rails や Grails、あるいは他のフルスタックの Web フレームワークとは比べものにならないほど軽量です。

Express をインストールして利用する簡単な方法は、NPM の `package.json` ファイル内で、Express を依存関係として宣言することです (リスト 1 を参照)。このファイルは Maven の `pom.xml` や Bundler の `Gemfile` と同様のものですが、フォーマットは JSON です。

リスト 1. NPM の package.json ファイル

```
{
  "name": "magnus-server",
  "version": "0.0.1",
  "dependencies": {
    "express": "2.4.6"
  }
}
```

リスト 1 では、この Node.js プロジェクトの名前 (magnus-server) とバージョン (0.0.1) を指定し、Express のバージョン 2.4.6 を依存関係として宣言しています。NPM の優れた点は、Express の過渡的な依存関係をすべて取得し、Express に必要な他のあらゆるサード・パーティー製 Node.js ライブラリーを速やかにロードするところにあります。

`package.json` でプロジェクトの依存関係を定義した後は、コマンドラインから `npm install` を実行することで、必要なパッケージをインストールすることができます。これにより、NPM が Express と併せて connect、mime などの依存関係がインストールされるはずです。

ネットワーク・アプリケーションを作成する

Magnus Server というサンプル・アプリケーションを作成するために、まず、JavaScript ファイルを作成するところから始めます。私はこのファイルに web.js という名前を付けましたが、他の名前にしても構いません。ファイルを作成したら、お気に入りのエディターまたは IDE でそのファイルを開きます。例えば、Eclipse JavaScript プラグインの JSDT を使用することができます (「参考文献」を参照)

ファイルに、リスト 2 のコードを追加します。

リスト 2. Magnus Server: 出だしのコード

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

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

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

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

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

このコードは短いながらも、かなり大きな役割を果たすので、コードの先頭から説明していきます。まず、Node.js でサード・パーティー製ライブラリーを使用する場合には、`require` 文を使用する必要があります。リスト 2 では、この require 文で Express フレームワークを指定し、`express` 変数でそのハンドルを取得しています。次に、`createServer` 呼び出しによってアプリケーション・インスタンスを作成します。これによって、HTTP サーバーが作成されます。

続いて、`app.put` を使用してエンドポイントを定義します。この例では、アプリケーションのルート (`/`) でリッスンする必須 HTTP メソッドとして、HTTP `PUT` を定義しています。`put` 呼び出しには 2 つのパラメーターがあります。1 つはルートで、もう 1 つはそのルートが呼び出されると実行されるコールバックです。2 つめのパラメーターであるコールバックは、実行時にエンドポイント `/` へのアクセスがあると呼び出される関数です。重要な点として、このコールバックが、Node.js で言うイベント駆動型 I/O、またはイベント化 I/O であることを覚えておいてください。このコールバックは非同期で呼び出されることになります。手動でスレッドを作成しなくても、ここで定義されたエンドポイントは、無数の同時リクエストを処理することができます。

エンドポイント定義の一部として、`/` に対する `PUT` を処理するためのロジックを作成します。ここでは簡単のため、とりあえずレスポンス・タイプを JSON に設定してから、単純な JSON 文書 (`{"status":"success"}`) を返します。巧みな `stringify` メソッドがハッシュを取り、それを JSON に変換することに注目してください。

JavaScript と JSON

JSON と JavaScript は実質的に兄弟のような関係にあり、その親和性が Node.js につながっています。Node.js アプリケーション内で JSON を構文解析するには、特殊なライブラリーも構成体も必要ありません。代わりに、オブジェクト・グラフと同じような論理呼び出しを使用します。要するに、Node.js は JSON を家族のように扱うため、JSON ベースの Web アプリケーションを極めて簡単にプログラミングすることができます。

次に作成するのは、アプリケーションにリッスンさせるポートを表す変数です。ポートを指定するには、`PORT` 環境変数を取得するか、あるいは明示的にポートを 3000 に設定します。そして最後に、`listen` メソッドを呼び出してアプリケーションを起動します。ここでもコールバックを渡し、アプリケーションが起動すると、このコールバックが呼び出されて、コンソール (この例では、標準出力) にメッセージを出力するようにしています。

テストしてみましょう！

この巧みなアプリケーションはあらゆる `PUT` に応答するので、コマンドラインで「`node web.js`」と入力するだけでアプリケーションを実行することができます。このアプリケーションをさらに詳しくテストする場合には、WizTools.org の RESTClient をダウンロードすることをお勧めします。RESTClient では、http://localhost:3000 に対してHTTP `PUT` を実行するだけで、Magnus Server が機能しているかどうかを簡単に調べることができます。何も問題がなければ、成功を通知する JSON レスポンスが表示されます (RESTClient のインストールおよび使用方法についての詳細は、「参考文献」を参照してください)。

Express による JSON の処理

JavaScript と JSON は密接に関係しているため、Express での JSON の管理はこの上なく簡単です。このセクションでは、受信した JSON 文書を取得して標準出力に出力できるように、リスト 2

のスケルトン・アプリケーションにもう少しコードを追加します。その後、データをまるごと MongoDB インスタンスに永続化します。

受信した文書は、リスト 3 のような内容になっています (簡単のため、場所の情報は省略されていることに注意してください)。

リスト 3. 受信した文書の内容

```
{
 "deal_description":"free food at Freddie Fingers",
 "all_tags":"free,burgers,fries"
}
```

リスト 4 で、受信した文書を構文解析する機能を追加します。

リスト 4. Express による JSON の構文解析

```
app.use(express.bodyParser());

app.put('/', function(req, res) {
  var deal = req.body.deal_description;
  var tags = req.body.all_tags;

  console.log("deal is : "  + deal + " and tags are " + tags);

  res.contentType('json');
  res.send(JSON.stringify({ status: "success" }));
});
```

リスト 4 には、Express に `bodyParser` を使用するように指示している 1 行が含まれていることに注目してください。これにより、受信した JSON 文書から簡単に (本当に簡単に) 属性を取得できるようになります。

`put` コールバックの中に含まれているのは、受信した文書の `deal_description` 属性と `all_tags` 属性の値を取得するためのコードです。このように実に簡単に、要求した文書から個々の要素を取得することができます。この例の場合、`req.body.deal_description` が `deal_description` の値を取得します。

テストしてください！

この実装はテストすることもできます。それには、`magnus-server` インスタンスをいったん停止してから再起動し、HTTP `PUT` を使って JSON 文書を Express アプリケーションに送信します。このテストでは、始めに成功のレスポンスが表示され、次に、送信した値が Express によって標準出力に出力されます。そして、上記の Freddie Fingers 文書の出力が取得されます。

```
deal is : free food at Freddie Fingers and tags are free, burgers, fries.
```

Node.js での永続化

ここまでの作業で、アプリケーションは JSON 文書を受信して構文解析し、レスポンスを返すようになりました。次に必要なのは、永続化ロジックを追加することです。私は MongoDB をお気に入りなので (「参考文献」を参照)、MongoDB インスタンスによってデータを永続化するこ

とにします。また、受信した JSON の値を保存するためにサード・パーティー製ライブラリーの Mongolian DeadBeef を使用することで、作業をさらに簡単にします。

Mongolian DeadBeef は、Node.js に使用できる数ある MongoDB ライブラリーの 1 つです。このライブラリーを選んだのは、名前が面白いというだけでなく、ネイティブ MongoDB ドライバーをミラーリングしていることから、MongoDB を使っている感覚で使用できるためです。

もうおわかりのことと思いますが、Mongolian DeadBeef を使用する最初のステップは、`package.json` ファイルを更新することです (リスト 5 を参照)。

リスト 5. JSON 構文解析の追加

```
{
  "name": "magnus-server",
  "version": "0.0.1",
  "dependencies": {
    "express": "2.4.6",
 "mongolian": "0.1.12"
  }
}
```

私たちが接続するのは MongoDB データストアなので、`npm install` を実行してプロジェクトが必要とする依存関係も更新しなければなりません。Mongolian DeadBeef の MongoDB ドライバーのパフォーマンスを向上させるためには、ネイティブ C++ bson パーサーをインストールするという方法もあります。その場合、NPM がインストールを手伝ってくれます。

Mongolian DeadBeef を使い始めるには、現行の実装に `require` を追加してから、目的の MongoDB インスタンスに接続します (リスト 6 を参照)。この例では、MongoDB のクラウド・プロバイダーである MongoHQ でホストされているインスタンスに接続します。

リスト 6. magnus-server への Mongolian DeadBeef の追加

```
var mongolian = require("mongolian");
var db = new mongolian("mongo://a_username:a_password@flume.mongohq.com:23034/magnus");
```

`PUT` コールバックの中で、受信した JSON 文書からの値を永続化します (リスト 7 を参照)。

リスト 7. Mongolian の挿入ロジックの追加

```
app.put('/', function(req, res) {
  var deal = req.body.deal_description;
  var tags = req.body.all_tags;

  db.collection("deals").insert({
     deal: deal,
     deal_tags: tags.split(",")
  })

  res.contentType('json');
  res.send(JSON.stringify({ status: "success" }));
});
```

上記のリストを詳しく見てみると、この `insert` 文は、MongoDB シェル内での挿入とまったく同じであることがわかります。これは偶然ではありません。MongoDB のシェルは JavaScript を使用

するからです。そのため、`deal` と `deal_tags` という 2 つのフィールドを持つ文書を簡単に永続化できるようになっています。`tags` ストリングでは `split` メソッドを使って `deal_tags` を配列に設定していることに注意してください。

テストできるでしょうか？(もちろんです！)

このアプリケーションをテストしたいと思ったら (思わないわけがありません)、インスタンスを再起動して別の JSON 文書を送信し、MongoDB 内部の `deals` コレクションを調べてください。送信した文書とほとんど同じ JSON 文書を確認できるはずです。

リスト 8. Mongolian の挿入ロジックの追加

```
{
  deal:"free food at Freddie Fingers",
  deal_tags: ["free", "burgers", "fries"],
  _id: "4e73ff3a41258b7423000001"
}
```

まとめ — 本当にこれだけです！

Node.js の紹介をこれだけの短い記事で終わらせようとしていることに、私が手を抜いていると思うかもしれませんが、本当に、紹介する内容はこれで終わりです！たった 20 行のコードを作成しただけですが、これで本格的な永続化アプリケーションが完成します。これが、Node.js の素晴らしいところです。Node.js のコードは作成するのも、理解するのも驚くほど簡単で、しかも非同期コールバックによって極めて強力なアプリケーションになります。最大限のスケーラビリティーを実現しようと思ったら、完成したアプリケーションをいくつもの PaaS プロバイダーにデプロイすることもできます。

Node.js や MongoDB などの、この記事で取り上げた技術や、Google App Engine、Amazon Elastic Beanstalk、Heroku といった PaaS の選択肢について詳しく学ぶには、「参考文献」セクションを参照してください。

著者について

Andrew Glover

Andrew Glover は、ビヘイビア駆動開発、継続的インテグレーション、アジャイル・ソフトウェア開発に情熱を持つ開発者であるとともに、著者、講演者、起業家でもあります。また、easyb BDD (Behavior-Driven Development) フレームワークの創始者、そして『継続的インテグレーション入門 開発プロセスを自動化する47の作法』、『Groovy in Action』、『Java Testing Patterns』の 3 冊の本の共著者でもあります。詳細は彼のブログにアクセスするか、Twitter で彼をフォローしてください。