



CDI 26

JYTHON 34

DOCKER入門 43

WEBSOCKET 50

NOVEMBER/DECEMBER 2015

JavaTM magazine

By and for the Java community 

ライブラリ

正しいライブラリの検索

05 JCOMMANDER:
コマンドラインの解析

11 BYTE BUDDY:
バイトコードの生成

16 JSOUP:
優雅なHTML解析

22 JVMによるライブラリの
検索、ロード


```
//from the editor /
```

Andrew Binstock: Java Magazine編集長。以前はDr. Dobbsの編集者であり、さまざまな技術文書の出版も担当。また、オープンソースPDFライブラリ「iText」の商業部門として、iText Softwareを共同で創業。さまざまなオープンソース・プロジェクトにも貢献している。著書に、『Practical Algorithms』（Addison-Wesley、1995）を含む3冊のプログラミング関連書籍がある。



オープンソース・ライセンスを改革する

複雑化したオープンソース・ライセンスには処方箋が必要です。
その処方箋となるのがクリエイティブ・コモンズのモデルです。

開発者と話したり、さまざまなフォーラムを訪れたりしていつも驚くのですが、広く普及しているオープンソース・ライセンスについて、その内容がまったくと言っていいほど知られていません。コードをフリーなオープンソースとして公開する際は、いまだにパブリック・リポジトリ (GitHub、Bitbucket、SourceForge を代表とする、急速に縮小中のサイト群) が一般的な共有方法として使われますが、どのライセンスを選択すべきかという知識は広まっていないようです。多くの開発者から見た「ライセンス」という分野は、法律家の説明を要する風変わりな用語 (コピーレフト、BSD 2 条項など) に満ちた、広大でぼんやりとした領域です。コードを公開するための細かな調査を好まない開発者を救おうと、Apache、GPL、MIT などの主流ライセンスが推奨されていますが、これらのライセンスに関する実用的な説明はほとんどありません。

知性にあふれ、抽象性の破綻などという問題をとことん議論して喜ぶような人たちが、オープンソース・ソフトウェア (OSS) ライセンスの基本的な仕組みを理解しようとししないのは、奇妙に思えるかもしれません。しかし、私に言わせれば、この見方は時代遅れです。開発者が興味を持つのは仕事に必須の抽象的な構成概念であるにも関わらず、なぜ、自身の創作物をより大きなコミュニティに提供するために、法律家が書いた謎の条項を理解しなければならないのでしょうか。

その理由は、オープンソース・ライセンスが雑然とした条項の集まりで、重複、矛盾のある（さらに複雑であることも多い）歴史上の偶然の産物だからです。ライセンスは、開発者が簡単に理解できるようには作られませんでした。そうなった原因の1つは、オープンソース・ライセンスを監督するグループであるOpen Source Initiative (OSI) の怠慢です。同グループはさまざまな政治的目的のために、オープンソースの創生期に結成され、当時もっとも必要であったことを成し遂げました。それは、OSSライセンスの増殖を止めることです。OSIの結成以前は、コードをリリースする各企業が独自のライセンスを表記していました。これはたしかに、オープンソースの考えに則って行われたことですが、その結果避けられない問題が生まれました。それは、何ページもの法律文書を目を通さなければ、何ができるか、あるいは何ができないのか正確に理解することが非常に難しいということです。OSIは特定のライセンスを「オープンソース」として認定することによって、OSSの増殖を止めるのに貢献しました。そして、承認済みライセンスのサブセットを利用することを促進して、ライセンスのプールをさらに縮小させました（**注**：もっとも、OSIの承認を得なくても、独自のライセンスやコードをオープンソースと呼ぶことはできます。実際、SQLiteやTeXなど、私たちが今日オープンソースとして見ている大規模プロジェクトのいくつかは、OSIライセンスの下で発行されていません）。

CREATE
THE FUTURE

oracle.com/java

20
YEARS
1995-2015

Java™

ORACLE®

//from the editor /

ただし、広く普及しているOSI認定ライセンスのプールは大量の条項であふれかえっており、その多くは、理解するために法律家の見解が必要です。OSSライセンスのアナリストの中には、その条項のデータベースを保持している人もいます。このライセンス・プール内の要件は500件を超えるそうです。積極的に条項を理解しようという開発者の気持ちをくじくような数値です。結局、「どのライセンスを選択するか」は、ほぼ無意味になってしまいました。しかし、それでいい筈はありません。

フリー・ソフトウェアの歴史を思い出さない限り、あるいは使用条件について時間をかけて学ぶことのない限り、これらのライセンスの本来の境界線、つまりフリー・ソフトウェアとオープンソースの明確な違いを認識することはできないでしょう。一般に、フリー・ソフトウェアには「コピーレフト」条項があります。この場合、ライセンス対象のコードを利用するあらゆるソフトウェアを、同じライセンス条項の下で提供する必要があります (Lesser General Public License、いわゆるLGPLは有名な例外にあたります)。一方、OSSにはそのような条件はありません。したがって、GPL (フリー) とApache License (OSS) のどちらを選択するかは重要ですが、BSD (3条項) とBSD (2条項) のどちらを選択するかは重要ではありません。一方のBSDライセンスでコードをリリースした後に、もう一方のライセンスを選択すればよかったと後悔するような人はいないのです。

しかし時間を費やし、Webサイトやフォーラム、同僚との会話の中でこの難解な内容を学ばないかぎり、ほとんどの開発者は知識を持つこともできません。この問題の解決方法は、過去の文書に基づいたライセンスの利用を禁止し、クリエイティブ・コモンズが採用しているモデルに移行することだと私は強く思います。クリエイティブ・コモンズは、アーティストの創作物を対象とした、OSIに相当する組織です。

クリエイティブ・コモンズでは、パブリック・ドメインから始まる段階的なライセンス一式が規定されています。また、定義済みの条件を追加できるようになっています。条件の数は少なく、著作権者の表示、商用利用が可能か否か、二次創作が可能か否か、さらに二次創作は同じライセンス下でのみ許可されるという条件もあります。このシステムがあることで、長い時間をかけて調査、分析することなく、アーティストはどのライセンスを利用すべきかを正しく判断できます。そして、各条項を適宜組み合わせ、適切なライセンスを正しく選択できるのです。

ソフトウェアでも、このシステムを導入する必要があります。複雑で矛盾する過去の遺物を取り払い、とにかくシンプルで良識のあるシステムが求められています。

編集長 **Andrew Binstock**
javamag_us@oracle.com
[@platypusguy](http://platypusguy.com)



ライブラリ：
新たな秘宝を発掘する

この元々のビジョンは、オブジェクトより多少大きな単位で実現されることになりました。それがライブラリです。そして現在、ライブラリの充実こそが、その言語の幅広い利用の証となっています。このことは、C++ や C#、Python、そしてもちろん Java にも当てはまります。実際のところ、Java には選択可能なライ





CÉDRIC BEUST

Cédric Beust:

1996年より
Javaのコーディ
ングの経験
があり、長年、
Java言語およ
びJavaライブ
ラリの開発に積
極的に関与。フ
ランスのニー
ス大学でコン
ピュータ・サイ
エンスの博士
号を取得。

JCommander:もっと手軽に コマンドライン解析

アノテーションを利用して非常に複雑なコマンドラインを解析できる使いやすいライブラリ

筆者は数年前、おもにコマンドラインから使用するアプリケーションを開発する必要に迫られました。複雑なコマンドライン・パラメータの解析が求められる、かなり壮大なプロジェクトでした。自然な流れとして、まずは柔軟性を維持しながらアプリケーションのコマンドライン構文を簡単に指定できるようなライブラリを探すことにしました。その目的でいくつかのライブラリを見つけましたが、いずれもかなり古風で、Javaの登場よりも前から存在するアイデアや手法が使用されていたため、行き詰まってしまいました。また、いずれのライブラリでもJavaの最新機能を利用できませんでした。

筆者は新しいアイデアを色々と試し始めました。気が付くと、既存のライブラリを用いるという最初のアイデアは完全に捨て、[JCommander](#)を作成していました。JCommanderは、コマンドライン引数を容易に解析でき、できるだけ多くの引数構文のスタイルに対応するように設計された、最新式のオープンソース・ライブラリです。引数は文字列、数値、コマンドに限らず、リストや任意のJavaオブジェクト、パスワードなども含めることができます。概要を見てみましょう。

概要

JCommanderを設計する際に最初に実現したことは、結局のところ、コマンドラインのすべてのオプションを解析して、その結果をJavaオブジェクトとして返すことです。オブジェクトと言っても、そのほとんどはPOJO (Plain Old Java Object) と呼ばれる非常に単純なものです。ロジック用のメソッドがなく、フィールドとgetter/setterしか持たない単なるコンテナの場合も少なくありません。

では手始めとして、以下の行を解析する簡単な「hello world」プログラムを記述してみます。

```
tool --name Cedric --verbose
```

解析後の情報は以下のクラスでキャプチャできます。

```
class Args {
    boolean verbose;
    String name;
}
```

その際、アノテーションを使用して、JCommanderに対してこのクラスの初期化方法を指定します。

```
class Args {
    @Parameter(names = "--verbose")
    boolean verbose;

    @Parameter(names = "--name")
    String name;
}
```



必要な操作は、このクラスのインスタンスでJCommanderを初期化し、コマンドラインのパラメータを渡すことです。解析の完了後、適切なフィールドにすべて割り当てられた正しい値がインスタンスに設定されます。

```
public static void main(String[] argv) {
    Args args = new Args();
    new JCommander(args).parse(argv);

    System.out.println("Hello " + args.name
        + ", verbose is:" + args.verbose)
}
```

アノテーションは、次のようなさまざまな理由でこの種のアプローチに非常に適しています。まず、構文が引数クラス内で非常に明確に展開されます。また、ソースを読むだけでプログラムが受け入れる引数が分かります。他の利点についてもこの記事で簡潔に説明します。

アノテーションの力

JCommanderのアプローチの中でもっとも際立った特徴は、アノテーションの利用です。筆者はいつもJavaのアノテーションを強く支持してきました。しかし、筆者はアノテーションを設計した委員会のメンバーであったため、やや偏向的かもしれません。それでも、現在、そして10年後も、アノテーションが極めて表現豊かなJavaのプログラミング・スタイルを支えるという考えは変わりません。

ここで覚えておくべき重要なポイントは、Java要素、たとえばクラス、フィールド、メソッドなどに追加の意味を関連付けようとする場合、アノテーションが非常に大きな力を発揮するという点です。パッケージ情報、ホスト名、ポート名など、Java要素と具体的に結び付かない情報はすべて、外部から指定する必要があります。この単純なルールを念頭に置けば、アノテーションがJCommanderにとって最適な選択肢であることは明白です。

さらに、アノテーションは複数の属性を持つことができるため、Java要素に関連付

**JCommander の
アプローチの中で
もっとも際立った特
徴は、アノテーショ
ンの利用です。ア
ノテーションは極
めて表現豊かな
Java のプログラミ
ング・スタイルを
支えます。**

けるメタデータをより細かく設定できます。先ほどのコードでは属性は1つのみ (`names`) でしたが、以下のように他の属性を追加できます。

```
@Parameter(names = {"--output", "-o"},
    required = true,
    description = "The output file")
String file;
```

`required`属性を指定したため、このパラメータが省略されると、JCommanderが例外をスローします。

Exception in thread "main" com.beust.jcommander.ParameterException:The following option is required:--output

注意点ですが、`names`が複数形になっています。つまり、この属性には複数の名前を指定でき、以下の2つのコマンドラインは同じ意味になります。

- tool --out file
- tool -o file

コマンドライン・オプションの入力はユーザーによって好みのスタイルが変わるという、よくある問題を解決するための機能です。

使用法の説明

前項でdescription属性を紹介したのは、この属性が、JCommanderでは特別な意味を持つからです。パラメータにこの属性がある場合、説明情報が自動的に収集されます。収集された情報は、構文の許容スタイルの説明をまとめて表示するために使用されます。無効な構文が入力された場合など、ユーザーに対してヘルプ・メッセージを表示したければ、JCommanderオブジェクトのusage()を呼び出すだけで実現できます。以下のような内容が表示されます。

Usage:<main class> [options]

Options:

- debug Debug mode (default: false)
- * --groups Group names to be run
- log, -verbose Level of verbosity (default:1)
- long A long number (default:0)

この説明に記載されるのは構文に関する最大限の情報で、JCommanderがオプションのアノテーションに基づいて収集したものです。オプション名、必須かどうか（アスタリスクで示す）、デフォルト値、そして言うまでもなく説明が載っています。

この機能は、別の重要なプログラミング原則である、「Don't Repeat Yourself (重複の排除)」を表現しています。引数クラス内に構文をいったん指定しておけば、ヘルプ・バナーの表示に際して同じ作業を繰り返す必要はありません。JCommanderが自動的に対応します。

型

JCommanderはデフォルトで多数の型を理解しますが、それらすべての型がアリティの概念につながります。アリティは、パラメータが必要とする値の数を定義するものです。以下に例を示します。

- booleanパラメータには値は不要: 値は、パラメータが存在する場合はtrue、省略した場合はfalse
- スカラー (int、long、string) には値が必要。例: `--logLevel 3`
- リストには複数の値が必要

JCommanderは、パラメータの型に基づいてこれらのアリティを自動的に推論します。さらに、デフォルトのアリティでは要件を満たさない場合、独自のアリティを定義できます。そのため、異なる種類の構文が許可されます。たとえば、booleanが1のアリティを持つと指定でき、その場合、以下のような構文がサポートされることになります。

```
tool --verbose true
```

JCommanderでは、可変のアリティ(任意の数の値を指定できるパラメータ)もサポートしています。`--files file1 file2 file3`.

これらのデフォルトの型では不十分だったり、アプリケーションによっては、より複雑なオプション指定が必要な場合もあります。たとえば、前の例では、文字列の形式で出力ファイルを指定しました。文字列の代わりに、実際の`java.io.File`オブジェクトを供給できれば便利ではないでしょうか。そのような場合に型コンバータが重宝されます。

先ほどの例を、文字列の代わりに実際のJavaファイルを指定するように修正しましょう。

```
@Parameter(names = "-file",
            converter = FileConverter.class)
```

File file;

`converter`属性が追加されています。実装が必要になる属性です。

```
public class FileConverter
    implements IStringConverter<File> {
    @Override
    public File convert(String value) {
        return new File(value);
    }
}
```

型コンバータはいくつでも指定でき、JCommanderはフィールドの型に基づいて型コンバータを自動的に使用します。非常にシンプルです。

構文の柔軟性

できるだけ多くの構文スタイルをサポートするために、JCommanderでは空白以外のセパレータを指定できるようになっています。たとえば、`java Main -log 3`の代わりに、`java Main -log=3`や`java Main -log:3`を使用してもかまいません。すべて、`separators`属性によって以下のように指定できます。

```
@Parameters(separators = "=")
public class SeparatorEqual {
    @Parameter(names = "-level")
    private Integer level = 2;
}
```

JCommander は非常に複雑 なコードベース や構文スタイル に対応しており、 コードを分かり やすく整理する ために、さまざ まな機能が用意 されています。

実際の処理内容を示すために、各コードの最後にいくつかのアサーションを追加しました。

```
JCommander jc = new JCommander();

CommandAdd add = new CommandAdd();
jc.addCommand("add", add);

CommandCommit commit = new CommandCommit();
jc.addCommand("commit", commit);

jc.parse("-v", "commit", "--amend",
        "--author=cbeust", "A.java", "B.java");

Assert.assertTrue(cm.verbose);
Assert.assertEquals(jc.getParsedCommand(),
        "commit");

Assert.assertTrue(commit.amend);
Assert.assertEquals(commit.author, "cbeust");
Assert.assertEquals(commit.files,
        Arrays.asList("A.java", "B.java"));
```

アサーションから分かるとおり、このコードは正しくコマンドラインを解析し、引数を期待される変数内に配置しました。

アーキテクチャ

JCommanderは非常に複雑なコードベースや構文スタイルに対応しており、コードを分かりやすく整理するために、さまざまな機能が用意されています。

複数の引数オブジェクト: 構文が拡大するうちに、1つの巨大な引数クラスを維持することがやや難しくなってきたと気づくことがあります。JCommanderではこのような引数クラスを複数のクラスに分割して、より直観的な形でオプションを整理することができます。

```
CommandRead argRead = new CommandRead();
CommandWrite argWrite = new CommandWrite()
JCommander jc = new JCommander(argRead, argWrite);
jc.parse(argv);
```

```
// argRead and argWrite are now both initialized
```

パラメータ・デリゲート: 複数のプログラムを記述していると、既存の引数クラスを再利用したいと思うことがあります。そのような場合に、パラメータ・デリゲートを使用できます。パラメータ・デリゲートとは、簡単に言えば、他の引数クラスへのポインタです。前の例では、異なる2つの引数クラスを作成し、これらの引数クラスを直接JCommanderで宣言することにしましたが、今回はこれらの引数クラスに委譲することになります。この操作は、`@ParameterDelegate`アノテーションを使用して実行できます。

```
class MainParams {
    @Parameter(names = "-v")
    private boolean verbose;

    @ParametersDelegate
    private ArgRead argRead = new ArgRead();

    @ParametersDelegate
    private ArgWrite argWrite = new ArgWrite();
}
```

この宣言後、`MainParams`という1つの引数パラメータを宣言する必要があります。この引数パラメータには、`ArgRead`および`ArgWrite`がまとめて格納されます。

多言語対応

JVMは複数言語に対応しているので、JCommanderはいずれのJVM言語からでも簡単に使用できます。筆者は現在、JCommanderをKotlinプロジェクトで以下のように使用しています。

```
class Args {
    @Parameter(names = arrayOf("--buildFile"))
    var buildFile:String?= null

    @Parameter(names = arrayOf("--tasks"))
    var tasks:Boolean = false
}
```

```
fun main(argv:Array<String>) {
    val args = Args()
```




```
JCommander(args).parse(*argv)
println("Args:${args}")
}
```

以下にGroovyの例を示します。

```
import com.beust.jcommander.*

class Args {
    @Parameter(names = ["-f", "--file"],
        description = "File to load.")
    List<String> file
}

new Args().with {
    new JCommander(it, args)
    file.each {
        println "file:${new File(it).name}"
    }
}
```

さらに同じ例をScalaで表すと以下のようになります。

```
import java.io.File
import com.beust.jcommander.JCommander
import com.beust.jcommander.Parameter
import collection.JavaConversions._

object Main {
  object Args {
    @Parameter(
      names = Array("-f", "--file"),
      description = "File to load.")
    var file: java.util.List[String] = null
  }

  def main(args:Array[String]):Unit = {
    new JCommander(Args, args.toArray:_*)
    for (filename <- Args.file) {
      val f = new File(filename)
    }
  }
}
```

```
        printf("file:%s\n", f.getName)
    }
}
}
```

まとめ

JCommanderには他にも多数の機能があります。その一部を紹介します。

- 国際化:説明テキストを適切にローカライズ可能
- パラメータの隠ぺい
- オプションの短縮
- 大文字小文字の区別なし(オプション)
- デフォルト値、デフォルト値ファクトリ
- 動的パラメータ(コンパイル時には未知のパラメータを解析する機能)

まとめると、JCommanderは、コマンドライン・パラメータを解析するための柔軟なライブラリであり、解析や解釈の機能を容易に保守し、進化させることができるように開発者を支援します。

LEARN MORE

- [GitHubのJCommander](#)
- [JCommanderのディスカッション・グループ](#)
- [JCommanderサンプル・ファイル](#)

Fabian Lange (@CodingFabian) :
Instana のリード・
エージェント開
発者兼パフォー
マンス・ギーク。
Instana では IT
運用ソリューションの構築に従事。
JavaOne Rock
Star の講演者も
務める。

エージェントを作成し、main()をロードする前にツールを実行し、実行時に動的にクラスを変更する

これらのライブラリはすべて、Javaコードの特定のバイトコード命令を記述、変更する目的で作成されたものです。しかし、これらのライブラリを利用するためには、バイトコードの仕組みを理解している必要があります。Javaソース・コードの理解とは根本的に異なる知識が求められます。加えて、これらのライブラリはJavaコードよりも利用しづらく、テストも困難です。それは、あるメソッド呼出しの引数の順序がそのシグネチャに一致しているか、あるいはJava言語仕様に違反していないかといったことを、Javaコンパイラで検証できないからです。さらに、これらのライブラリはやや古く、アノテーション、ジェネリクス、デフォルト・メソッド、ラムダ式などの新しいJavaの機能のすべてをサポートしていません。

注:本記事のサンプルでは、Byte Buddy 0.6 APIを使用しています。

Hello World、Byte Buddy

以下のHelloWorldサンプル(リスト1)はByte Buddyのドキュメントに記載されているものですが、実行時に新しいクラスを簡潔に作成するための必要手順がすべて示されています。

■ リスト1:

```
Class<? extends Object> clazz = new ByteBuddy()
    .subclass(Object.class)
    .method(ElementMatchers.named("toString"))
    .intercept(FixedValue.value("Hello World!"))
    .make()
    .load(getClass().getClassLoader(),
        ClassLoadingStrategy.Default.WRAPPER)
    .getLoaded();
assertThat(clazz.newInstance().toString(),
    is("Hello World!"));
```

Byte BuddyのすべてのAPIが関数型スタイルに対応しており、ビルダー方式の流れるような記述が可能です。コードでまず最初にやることとして、サブクラスの作成元となるクラスをByte Buddyに指定します。このサンプルでは、単純にObjectのサブクラスを作成していますが、finalではない任意のクラスのサブクラスを作成できます。Byte Buddyでは、ジェネリックな戻り型がClass<? extends SuperClass>となることが保証されます。サブクラスのビルダーを取得できたので、次にtoStringという名前のメソッドの呼出しをインターセプトし、すでにjava.lang.Objectで定義されている同名のメソッドを呼び出す代わりに固定値を返すということをByte Buddyに指示します。

ここでインターセプトという用語について疑問に思う人もいるでしょう。通常、サブクラスを作成する場合に、サブクラス内でスーパークラスのメソッドの実装を変更するときはオーバーライドという用語を使用します。一方、インターセプトとは、アスペクト指向プログラミング (AOP) 関連の用語であり、メソッドが呼び出されたときに「何を実行すべきか」をより強く表した概念です。

サブクラスの振る舞いについて宣言し終わったら、makeを呼び出して、クラスのいわゆるUnloaded表現を取得します。Unloaded表現は.classファイルのような役割を担い、実際、対応するクラス・ファイルを保管する機能もサポートしています。

最後に、リスト1のように、クラス・ローダーを使用してクラスをロードし、ロードされたクラスへの参照を取得します。Byte Buddyの使い始めの頃は、通常、

クラスのロードを行うために使用する`ClassLoaderStrategy`が問題になることはありません。しかし、可視化や、特定のロード順を適用する目的で、特定のクラス・ローダーを用いて新しいクラスをロードする必要性が生じるケースもあります。

Byte Buddyによって生成されるクラスは、通常のクラスと見分けが付かない点に注意してください。他のライブラリやプロキシとは異なり、痕跡が残らないのです。生成されるコードは、そのサブクラスを実装するためにJavaコンパイラが作成するコードと同じです。

ElementMatcherと実装

Byte Buddyを使用してクラスの振る舞いの追加または変更を行う際の一般的なタスクは、フィールド、コンストラクタ、メソッドをルックアップすることです。これらのタスクを簡単に実行できるように、Byte Buddyには使いやすい事前定義のElementMatcherが多数付属しています。たとえば、`hasParameter()`や`isAnnotatedWith()`は、メソッドのシグネチャをチェックします。また、便利なエイリアスもあり、`isEquals()`や`isSetter()`などは、Java共通の命名パターンを使用してメソッド名をマッチングします。これらの事前定義のマッチャーを使用することで、インターセプトするメソッドについて簡潔に記述できます。これらのマッチャーがなければ、非常に冗長な記述が必要になります。さらに、カスタムのElementMatcherを実装して、あらゆる複雑なユースケースに対応することも可能です。

また、`intercept()`内で使用できる、事前定義の代替実装も多数あります。2つの例を紹介しましょう。`MethodCall`は、パラメータを使用して異なるメソッドを呼び出すことができます。また、`Forwarding`は、同じパラメータを使用して別のオブジェクトの同じメソッドを呼び出すことができます。

さらに強力なインターセプト・メカニズムは、[MethodDelegation](#)によって実現されます。あるメソッドに委譲するときに、先にカスタムのコードを実行し、その後その呼出しを元の実装に委譲することができます。また、[@Origin](#)アノテーションを使用して、元のコール・サイトの情報に動的にアクセスすることもできます(リスト2)。次に示すように、他のメソッドに委譲する際にも、元のコール・サイトの情報に動的にアクセスできます。

元のメソッドまたはそのスーパー・メソッドをターゲット・メソッドから呼び出すために、Byte Buddyでは@DefaultCallパラメータと@SuperCallパラメータを使用できます。

実行時には起こりえるが、テスト目的で確実に再現するのが難しいシナリオで単体テストを記述したい場合があります。たとえば、リスト3では、制御フローをテストするために、乱数ジェネレータで特定の結果を生成する必要があります。

```
Lottery lottery = new Lottery(mockRandom);
assertTrue(lottery.win());
```

これまでの例では、`subclass()`を使用して、本来のサブクラスを強化したものを作成しました。Byte Buddyにはこの他に、`rebase`および`redefine`という2つの処理モードがあります。いずれのオプションも特定のクラスの実装を変更するものですが、`rebase`が既存のコードを維持するのに対して、`redefine`は既存のコードを上書きします。ただし、これらの変更方法には制限事項があります。すでにロードされたクラスを変更するためには、Byte BuddyをJavaエージェントとして動作させる必要があります(詳細は後述)。



単体テストやその他の特殊なケースで、Byte Buddyが最初にクラスをロードすることが保証される場合は、ロード中に実装を変更できます。その目的で、Byte BuddyはTypeDescriptionという概念をサポートしています。TypeDescriptionは、アンロード状態のJavaクラスを表します。そのようなクラスのプールを、(まだロードされていない)クラスパスから移入し、クラスを変更してからロードできます。たとえば、リスト3のLotteryクラスはリスト4のように修正できます。

■ リスト4:

```
TypePool pool = TypePool.Default.ofClassPath();
new ByteBuddy()
    .redefine(pool.describe("Lottery").resolve(),
        ClassFileLocator.ForClassLoader.ofClassPath())
    .method(ElementMatchers.named("win"))
    .intercept(FixedValue.value(true))
    // & make and load;
```

```
assertTrue(new Lottery().win());
```

注: `Lottery.class`をこのサンプルでの`describe`の呼出しに使用することはできません。Byte Buddyがクラスを書き換える前にクラスがロードされてしまうからです。Javaクラスがロードされた後は、通常はそのクラスをアンロードすることはできません。

Byte BuddyによるAOPエージェント

以下のサンプルでは、パフォーマンス監視およびロギング用のエージェントを作成します。このサンプルはJAX-WSエンドポイントの呼出しをインターセプトし、その呼出しにかかった時間を出力します。このようなエージェントは、`java.lang.instrument`の[Javadoc](#)で説明されている各種規則に従う必要があります。エージェントは、`-javaagent`コマンドライン引数で起動され、実際の`main`メソッドの前に実行されます（このため、`premain`というメソッド名になっています）。通常、エージェントはそれ自体のフックをインストールします。このフックがトリガーされた後に、通常のプログラムによってクラスがロードされます。この方法を使えば、ロード済みのクラスを変更できないという制約を回避できます。エージェントはスタック可能であり、好きな数だけ使用できます。エージェントのコードをリスト5に示します。

■ リスト5:

```
public class Agent {
    public static void premain(String args,
                                Instrumentation inst) {
        new AgentBuilder.Default()
            .rebase(isAnnotatedWith(Path.class))
            .transform((b, td) ->
                b.method(
                    isAnnotatedWith(GET.class)
                    .or(isAnnotatedWith(POST.class)))
                .intercept(to(Agent.class)))
            .installOn(inst);
    }
}

@RuntimeType
public static Object profile(@Origin Method m,
                             @SuperCall Callable<?> c)
    throws Exception {
    long start = System.nanoTime();
    try {
        return c.call();
    } finally {
        long end = System.nanoTime();
        System.out.println("Call to " + m + " took "
            + (end - start) + " ns");
    }
}
}
```

デフォルトのAgentBuilderを取得した後、rebaseの対象となるクラスを指定します。このサンプルでは、javax.ws.rs.Pathアノテーション付きのクラスのみを変更します。次に、このビルダーに対して、対象クラスの変換の仕方をtransformで指定します。このサンプルでは、エージェントはGETまたはPOSTというアノテーションの付いたメソッドの呼出しをインターセプトして、profileメソッドに委譲します。この処理を機能させるために、installOn()を使用してこのエージェントをInstrumentationにフックする必要があります。

profileメソッド自体は次の3つのアノテーションを使用します。[RuntimeType](#)は、戻り型の[Object](#)を、インターセプトしたメソッドが使用する実際の戻り型へ適合させる必要があることをByte Buddyに指示するものです。[Origin](#)は、イン

ターセプトした実際のメソッドへの参照を取得するためものです。この参照は、メソッド名を出力するために使用します。そして、[SuperCall](#)は、元のメソッド呼出しを実際に実行するためのものです。前のサンプルとは対照的に、superの呼出しを自分で行なう必要があります。これは、メソッド呼出しの前後で独自コードを実行して、時間を計測するためです。

Byte Buddyによるメソッドのインターセプトの実装方法を、デフォルトのJava `InvocationHandler`と比較すれば、Byte Buddyのメソッドの方がはるかに最適化されていることが分かります。Byte Buddyではインターセプトによって必要な引数のみが渡されるのに対して、`InvocationHandler`の場合は以下のインタフェースを満たす必要があるからです。

```
Object invoke(Object proxy,  
              Method method, Object[] args)
```

特に、自動ボクシングが必要になるプリミティブな引数や戻り型を使用する場合に、このメリットが大きくなります。Byte Buddyでは追加の[RuntimeType](#)アノテーションによって、ボクシングが最小限に抑えられます。単純なボクシングについてはJVMによってほぼ最適化されますが、[InvocationHandler](#)のような複雑なインタフェースについては、そうならない場合もあります。

-javaagentなしでエージェントを使用

エージェントを使用した実行時のコード生成と変更は強力な技ですが、その実現のために `-javaagent` 引数を指定しなければならないのが不都合な場合もあります。Byte Buddyには、Java [Attach API](#)を使用する便利な機能があります。このAPIは元々、実行時に診断ツールをロードする目的で作成されたもので、エージェントを現在実行中のJVMにアタッチします。使用するには追加の `byte-buddy-agent.jar` ファイルが必要になります。このJARファイルには、`ByteBuddyAgent` というユーティリティ・クラスが含まれています。このクラスを使用して、`ByteBuddyAgent.installOnOpenJDK()` を呼び出します。このメソッドは、JVMを `-javaagent` 付きで起動する場合と同じことを実行します。このアプローチを使用する場合の他の相違点は、`installOn(inst)` を呼び出す代わりに `installOnByteBuddyAgent()` を呼び出すことです。

まとめ

JDKの動的プロキシや、人気のある3つのサード・パーティのバイトコード操作用ライブラリがすでに存在しますが、Byte Buddyはこれらに潜む重大な問題を解消します。流れるようなAPIでジェネリクスを使用しているため、変更中の実際の型が分からなくなることはありません(他のアプローチではそうなりがちです)。Byte Buddyには豊富なマッチャー、トランスフォーマ、実装も付属しており、ラムダ式を介してこれらを利用できるため、比較的簡潔で読みやすいコードになります。

そのため、バイトコードの読取りや低レベルでの開発に慣れていない開発者にも理解しやすいものになっています。次のByte Buddyバージョン0.7では、ジェネリック型に関するすべてのインフラストラクチャがサポートされる予定です。そのため、実行時でもジェネリック型や型アノテーションを容易に操作できるようになります。バイトコード処理用のコードを大量に記述している開発者として、筆者はこのライブラリを自分で利用し、皆さんにも推薦します。[編集注:Byte Buddyは、2015年のJavaOneカンファレンスでDuke's Choice Awardを受賞しました]

LEARN MORE

- [Java 8のJVM仕様](#)
- [Stack OverflowのByte Buddyページ](#)

Mert Çalis, kan:
Java
Champion。
『PrimeFaces
Cookbook』、
『Beginning
Spring』(Wiley
Publications)
**の共著者で、ト
ルコで一番活
発なJavaユー
ザー・グループ
AnkaraJUGの
創立者。**

HTMLの解析、指定された要素の抽出、構造の検証、コンテンツのサニタイジングを簡単に実行

概要

HTMLの要素自身や属性、テキストといったコンテンツの操作も可能です。廃止されたタグを新しいバージョンのタグに変換できるため、HTML 4.xベースの古いコンテンツをHTML5やXHTMLに更新することもできます。さらに、ホワイトリストによるクリーンアップやHTML出力の整形、足りないタグの自動補完も可能です。こうした機能は、後ほど、実例で紹介します。

本記事のすべてのサンプルには、原稿執筆時点の最新版であるjsoupバージョン1.8.3を利用しており、すべてのソース・コードは、[GitHub](#)からダウンロードできます。

DOMは、HTMLドキュメントを言語に依存せずに表現し、ドキュメントの構造やスタイルを定義します。図1に、jsoupフレームワークのクラス図を示します。各クラスとDOM要素の対応付けについては、後ほど説明します。

jsoupの主要な要素となるのは、[org.jsoup.nodes.Node](#)抽象クラスです。このクラスはDOMツリー内のノードで、ドキュメント自身か、ドキュメント内のテキスト・ノード、コメント、要素(フォーム要素)のいずれかを指します。[Node](#)クラスは親

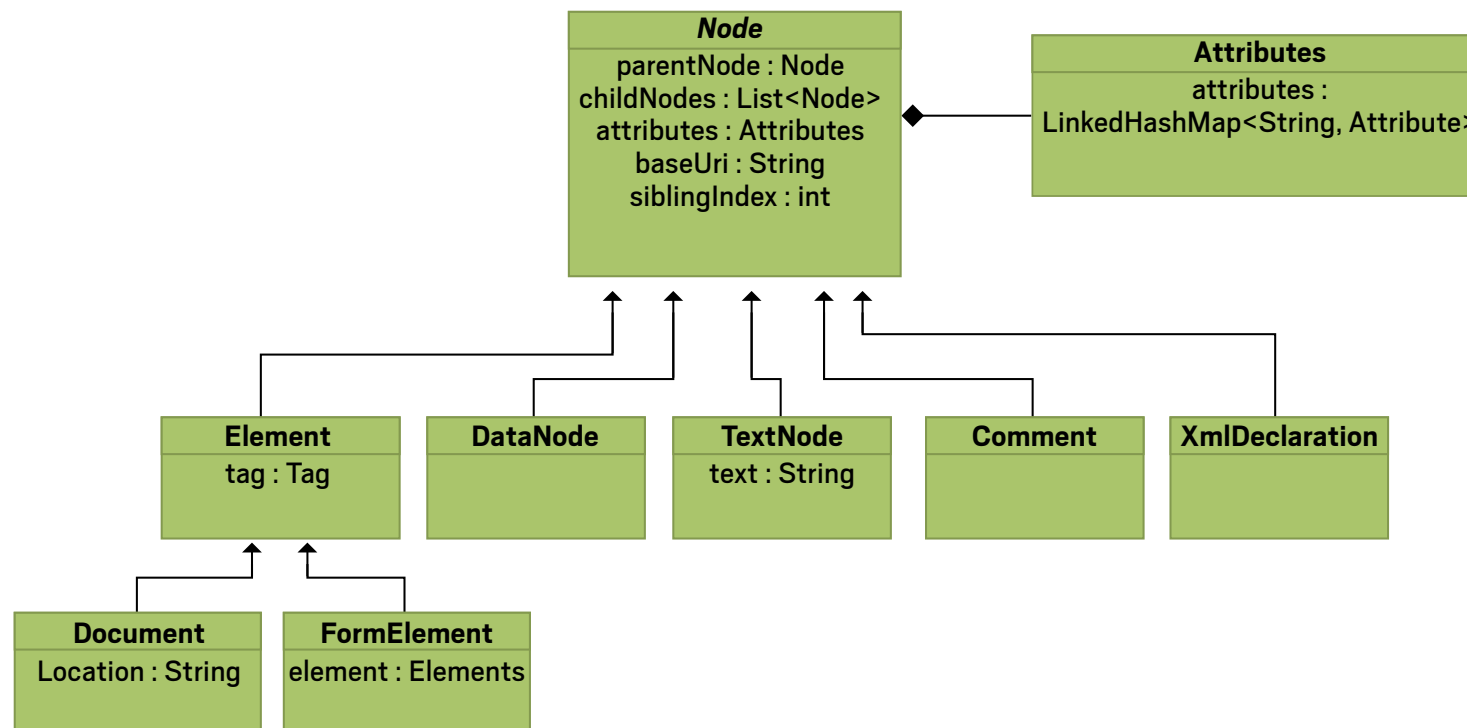


図1: jsoupのクラス図

ノードを参照できるので、親ノードのすべての子ノードを認識できます。

`Element`クラスはHTML要素を指し、タグ名、属性、子ノードで構成されています。`Attributes`クラスはHTML要素の属性を格納するコンテナで、`Node`クラスの内部にあります。

使用の開始

jsoupの最新バージョンは、依存関係を次のように定義し、Mavenのセントラル・リポジトリから入手します。バージョン1.8.3には、Java 5以上が必要です。

```
<dependency>
  <groupId>org.jsoup</groupId>
  <artifactId>jsoup</artifactId>
  <version>1.8.3</version>
</dependency>
```

Gradleユーザーは、以下の記述でアーティファクト（成果物）を取得できます。

`org.jsoup:jsoup:1.8.3`

jsoupの機能を使用するうえで、`org.jsoup.Jsoup`がメインのアクセス・ポイント・クラスとして非常によく使われます。このクラスの基本メソッドがHTMLドキュメントを解析し、対象となるHTMLドキュメントはファイル、入力ストリーム、文字列、URLのいずれかの形式で受け渡されます。Listing 1の例では、HTMLテキストを解析し、最初に要素のノード名、その後にHTML要素のテキストを出力（コードの下段またはコードの後半をご覧ください）しています。

■ リスト1:

```
public class Example1Main {

    static String htmlText = "<!DOCTYPE html>" +
        " <html>" +
        " <head>" +
        "   <title>Java Magazine</title>" +
        " </head>" +
        " <body>" +
        "   <h1>Hello World!</h1>" +
```

```
    " </body>" +
    "</html>";

    public static void main(String... args) {
        Document document = Jsoup.parse(htmlText);
        Elements allElements =
            document.getAllElements();
        for (Element element : allElements) {
            System.out.println(element.nodeName()
                + " " + element.ownText());
        }
    }
}
```

出力結果は次のようになります。

```
#document
html
head
title Java Magazine
body
h1 Hello World!
```

DOM要素の選択方法: 解析したHTMLから目的の要素を検索する場合は反復処理を行います。jsoupには複数の方法が準備されています。具体的には、DOM固有の `getElementBy*` メソッドまたはCSSやjQueryのようなセクタを使用します。両方のアプローチを紹介するため、例として、Webページを解析してHTMLの<a>タグのリンクをすべて抽出してみます。**Listing 2**のコードは、Javaチャンピオンの紹介ページを解析し、「New!」が付いているすべてのJavaチャンピオンのリンク名を抽出するものです（図2を参照）。

「New!」を表示するタグはリンクのすぐ右隣にあります。したがって、それぞれのリンクについて、直後の兄弟要素の内容を確認していきます。

DOM 固有のメソッドよりも、CSS や jQuery のようなセクタの方が強力です。 セクタは、選択範囲を絞り込むために組み合わせで使用できます。

■ リスト2:

```
public class Example2Main {

    public static void main(String... args)
        throws IOException {
        Document document = Jsoup.
connect(
    "https://java.net/website/" +
    "java-champions/bios.html" )
    .timeout(0).get();

    Elements allElements =
        document.
getElementsByTag("a");
    for (Element element : allElements) {
        if ("New!".equals(
            element.nextElementSibling() != null
            ? element.nextElementSibling()
            .ownText()
            : "")) {
            System.out.println(
                element.ownText());
        }
    }
}
}
```



図2: 解析するHTMLページの一部

```

        ("https://java.net" +
        " /website/java-champions/bios.html")
        .timeout(0).get();
    Elements allElements = document.select
        ("a[href*=#]");
    for (Element element : allElements) {
        if ("New!".equals(element
            .nextElementSibling() != null
            ? element.nextElementSibling
            ().ownText() : "")) {
            System.out.println(element
                .ownText());
        }
    }
}
}
}

```

DOM固有のメソッドよりも、セレクトタの方が強力です。セレクトタは、選択範囲を絞り込むために組み合わせで使用できます。先ほどのコード例では、「New!」というテキストを自分でチェックしました。しかし、これではちょっと平凡です。リスト4の例では、hrefが#で始まるリンクの後に存在し、「New!」というテキストを含むタグを選択しています。この例から、セレクトタがいかに強力であるかがわかるでしょう。

■ リスト3:

```
public class Example3Main {  
  
    public static void main(String... args)  
        throws IOException {  
        Document document = Jsoup.connect
```

■ リスト4:

```
public class Example4Main {

    public static void main(String... args)
        throws IOException {
        Document document = Jsoup.connect
            ("https://java.net" +
            ".website/java-champions/bios.html")
            .timeout(0).get();
        Elements allElements = document.select
            ("a[href*=#] ~ font:containsOwn" +
            "(New!)");
        for (Element element : allElements) {
            System.out.println(element
                .previousElementSibling()
                .ownText());
        }
    }
}
```

ここでは、セレクトは要素としてのタグを探しています。そして、`previousElementSibling()`メソッドを呼び出して1つ前の要素に戻り、リンクを表

示しています。この`select()`メソッドは、`Document`、`Element`、`Elements`の各クラスで使用できます。現在のjsoupでは、セレクトタに対するXPath問合せはサポートされていません。セレクトタについての詳しい情報は、[jsoupのサイト](#)で確認できます。

ノードのトラバース:jsoupが提供している[org.jsoup.select.NodeVisitor](#)インタフェースには、[head\(\)](#)と[tail\(\)](#)という2つのメソッドが含まれています。匿名クラスでこのインタフェースを実装し、パラメータとして[document.traverse\(\)](#)メソッドに渡すことによって、ノードの開始時と終了時にコールバックを受けることができます。**Listing 5**のコードでは、この方法を活用して単純なHTMLテキストをトラバースし、すべてのノード情報を出力しています。

jsoup バージョン 1.6.2 以降では、組み込み XML パーサーによる XML ファイルの解析がサポートされています。XML の解析も簡単にできることがよくわかります。

■ リスト5:

```
public class Example5Main {

    static String htmlText = "<!DOCTYPE html>" +
        "<html>" +
        "<head>" +
        "<title>Java Magazine</title>" +
        "</head>" +
        "<body>" +
        "<h1>Hello World!</h1>" +
        "</body>" +
        "</html>";

    public static void main(String... args)
        throws IOException {
        Document document = Jsoup.parse(htmlText);

        document.traverse(new NodeVisitor() {
            public void head(Node node, int depth){
                System.out.println("Node start:"
                    + node.nodeName());
            }

            public void tail(Node node, int depth){
                System.out.println("Node end:" +
                    node.nodeName());
            }
        });
    }
}
```

トラバースの結果、次のように出力されます。

```
Node start:#document
Node start:#doctype
Node end:#doctype
Node start: html
Node start: head
```

```
Node start: title
Node start: #text
Node end: #text
Node end: title
Node end: head
Node start: body
Node start: h1
Node start: #text
Node end: #text
Node end: h1
Node end: body
Node end: html
Node end: #document
```

悪意のある HTML 入力を防止するソリューションとして、WYSIWYG エディタを使用し、jsoup のホワイトリスト・サニタイザによって HTML 出力をフィルタリングする方法があります。

出力は、次のようになります。

```
<?xml version="1.0"encoding="UTF8">
<entries>
  <entry>
    <key>
      xxx
    </key>
    <value>
      yyy
    </value>
  </entry>
  <entry>
    <key>
      xxx
    </key>
    <value>
      zzz
    </value>
  </entry>
</entries>
```

セレクタを使用し、指定したXMLタグから値を選択することも可能です。Listing 7のコード・スニペットでは、`<entry>`タグ内の`<value>`タグを選択しています。

■ リスト6:

```
public class Example6Main {

    static String xml =
        "<?xml version=\"1.0\" +
        \"encoding=\"UTF8\"><entries><entry>\" +
        \"<key>xxx</key>\" +
        \"<value>yyy</value></entry>\" +
        \"<entry><key>xxx</key>\" +
        \"<value>zzz</value>\" +
        \"</entry></entries></xml>\";

    public static void main(String... args) {
        Document doc =
            Jsoup.parse(xml, "", Parser.xmlParser());
        System.out.println(doc.toString());
    }
}
```

■ リスト7:

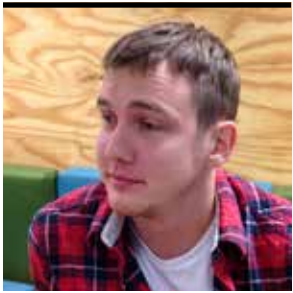
```
Document doc =  
    Jsoup.parse(xml, "", Parser.xmlParser());  
  
Elements elements = doc.select("entry value");  
Iterator<Element> it = elements.iterator();  
while (it.hasNext()) {  
    Element element = it.next();  
    System.out.println(element.nodeName() +  
        " - " + element.ownText());  
}
```


Listing 8の例で定義しているのは、テキストの単純なホワイトリストに従ってHTMLテキストをクリーンアップするテスト・メソッドです。このリストでは、テキストの単純な整形を行うb、em、i、strong、uの各HTMLタグのみを許可しています。

Listing 9は、`basic()`を使用した例ですが、a、b、blockquote、br、cite、code、dd、dl、dt、em、i、li、ol、p、pre、q、small、span、strike、strong、sub、sup、u、ulの各HTMLタグが許可されます。

本記事で紹介したjsoupの機能はほんの一端に過ぎません。他にも、HTMLの整形、HTMLタグの属性やテキストの操作、CSSセレクタの構造や疑似構造の結合などの便利な機能が提供されています。つまり、HTMLの処理が必要なら、いかなる場合もjsoupの使用が検討に値します。

- [Stack Overflowでのjsoupに関する話題](#)
- [jsoupレシピ](#)
- [HTMLパーサーの比較](#)



OLEG ŠELAJEV

Oleg Šelajev
(@shelajev) :
ZeroTurnaround
のエンジニア、
著者、講演者、
講師、開発提
唱者。Clojure、
Git、MacVim の
研究を楽しみつ
つ、タルトゥ大学
でソフトウェアの
動的更新とコー
ド進化に関する
博士号の取得を
目指す。

JVMによるライブラリの 検索、ロード、実行

JVMのプログラム実行方法を理解する鍵となるクラス・ローダー

クラスはJavaの型システムを構成する要素ですが、クラスが果たす基本的な役割はそれだけではありません。クラスはコンパイル・ユニット、つまり個別にロードしてJVMプロセスを実行できる最小のコードです。クラス・ローディングのメカニズムが定められたのは、Javaの創生期であるJDK 1.0の頃にまで遡ります。そしてこのメカニズムが、Javaのクロス・プラットフォームとしての高い人気に大きな影響を及ぼしました。コンパイル後のJavaコード(クラス・ファイルおよびパッケージ化されたJARファイルの形式)は、多数あるサポート対象オペレーティング・システムのいずれであっても、そこで実行中のJVMプロセスにロードできます。だからこそ、コンパイル済みのバイナリ形式ライブラリを開発者が容易に配布できるのです。ソース・コードやプラットフォーム依存のバイナリよりもJARファイルの方がはるかに配布しやすいことがJavaの普及を支えました。特にオープンソース・プロジェクトにおいてその傾向は顕著です。

本記事では、Javaのクラス・ローディングのメカニズムの詳細とその仕組みについて説明します。また、クラスのクラスパス内での検索方法や、メモリへのロード方法、クラスを使用するための初期化方法についても説明します。

JVMへのクラス・ローディングのメカニズム

以下のような単純なJavaプログラムについて考えます。

```
public class A {  
    public static void main(String[] args) {  
        B b = new B();  
        int i = b.inc(0);  
        System.out.println(i);  
    }  
}
```

この一片のコードをコンパイルして実行すると、JVMによってプログラムへのエントリ・ポイントが正しく認識され、クラスAのmainメソッドの実行が開始されます。しかし、JVMは、インポートされているクラス、あるいは参照されているクラスでも、かならずしも先行的に(すぐに)ロードするわけではありません。つまり具体的に言えば、JVMはnew B()文に対応するバイトコード命令に当たったときにはじめてBクラスの検索とロードを試みます。クラスのロード・プロセスを開始する方法には、クラスのコンストラクタ呼び出しの他に、クラスの静的メンバーへのアクセスや、Reflection API経由でのクラスへのアクセスなどもあります。

実際にクラスをロードするために、JVMはクラス・ローダー・オブジェクトを使用します。すでにロードされたすべてのクラスは自身のクラス・ローダーへの参照を持っており、JVMはそのクラス・ローダーを使用して、このクラスから参照されるすべてのクラスをロードします。先ほどの例では、クラスBのロードは、大まかに言って以下のJava文と同等です。`A.class.getClassLoader().loadClass("B")`.

23

クラス・ローディングの問題に直面していなくても、また日常的にプラグイン・アーキテクチャを構築していなくても、クラス・ローディングを理解することは、アプリケーションの内部動作の理解につながります。また、さまざまなJavaツールについて、その動き方の洞察もできるようになります。そして、クラスパスをクリーンで最新な状態に維持することがいかに良いことかが明白になります。

</article>

また、多くのセキュリティ機能は、権限チェックを行うためにクラス・ローダー階層を利用しています。たとえば、有名な[sun.misc.Unsafe.getUnsafe\(\)](#)メソッドは、ブートストラップ・クラス・ローダーによってロードされたクラスから呼び出された場合に、正常動作としてUnsafeクラスのインスタンスを返します。ブートストラップ・クラス・ローダーからはシステム・クラスのみが返されるため、Unsafe APIを使用するすべてのライブラリはReflection APIを利用してプライベート・フィールドから参照を読み取る必要があります。

一般に、ライブラリやフレームワークの開発時は、クラス・ローディングに関する問題について一切憂慮する必要はありません。クラス・ローディングは実行時に発生する動的なプロセスであり、開発者がそこに変更を加えることはほとんどありません。また、典型的なJavaライブラリの場合、クラス・ローディング・スキームを変更する利点もほぼありません。

Oleg Šelajev (@shelajev) : ZeroTurnaround のエンジニア、著者、講演者、講師、開発提唱者。Clojure、Git、MacVim の研究を楽しみつつ、タルトゥ大学でソフトウェアの動的更新とコード進化に関する博士号の取得を目指す。

LEARN MORE

- ・ クラス・ローダーの管理に関する情報
- ・ クラス・ローダーのJVM仕様

Antonio Goncalves :

Java/Java EE
を専門とする
上級開発
者。Java EE 5
や Java EE 6、
Java EE 7 に関
する書籍を執
筆。Paris JUG
および Devovx
France を設
立した Java
Champion。
各種 JSR の
独立系 JCP
メンバーを
務める他、
PluralSight で
は Java EE のト
レーニングを
開講。

Contexts and Dependency Injection: 新しいJava EEツールボックス

本

シリーズでは、Contexts and Dependency Injection (CDI) の神秘を解くことに挑戦してきました。前回までの記事では、依存性の注入における強い型付けの意味、CDIによるサード・パーティ製フレームワークの統合方法、そして、インターセプタ、デコレータ、イベントによる疎結合の作成方法について説明しました。最終回となる今回の記事では、Java EEとCDIの統合について説明します。

Java EEは、Javaランタイムの拡張機能です。Java EEが管理する環境のもと、コンテナがコンポーネントにさまざまなサービスを提供します。サービスの中には、ライフサイクル管理、セキュリティ、検証、永続化などがあり、もちろんインジェクションも含まれています。多くの場合、永続化とトランザクションはひとまとまりとして提供され、アプリケーションのバックエンドの開発に利用されます。Web層では、Java EEはサーブレットやWebSocket [編集注:[こちらの記事](#)もご覧ください]、JavaServer Faces (JSF) との組合せで使用されます。こういった機能

Web層とサービス層の結合

Java EEにはさまざまな技術が含まれており、Webアプリケーション、RESTインタフェース、バッチ処理、非同期メッセージング、永続化など、あらゆる種類のアーキテクチャを作成できます。図1に示すように、こういったアプリケーションは、プレゼンテーション、ビジネス・ロジック、ビジネス・モデル、外部サービスとの相互運用などのいくつかの層に分類でき、ニーズに応じて、ステートレスからステートフル、フラット・レイヤーから多階層まで、あらゆる種類のアーキテクチャを実現できます。しかし、ここで1つ問題となるのは、Web層とサービス層にはそれぞれのパラダイムがあり、それぞれの言語があることです。そのため、この2つの層

を結合する際にCDIが重要な役割を果たすことになります。

サービス層で使われるJava:HTMLを使用するWebクライアントとデータベース定義言語を使用するデータベースを除き、Java EEの大半でJavaが主言語として利用されます。そのため、ほとんどのアプリケーション層にJavaが登場することになります。たとえば、ビジネス・モデル層ではJava Persistence APIエンティティとして、ビジネス・ロジック層では単純なBeanとしてJavaが使われています。プレゼンテーション層の一部でもJavaが使用されており、JSFバックエンドBeanはJavaで書かれています。

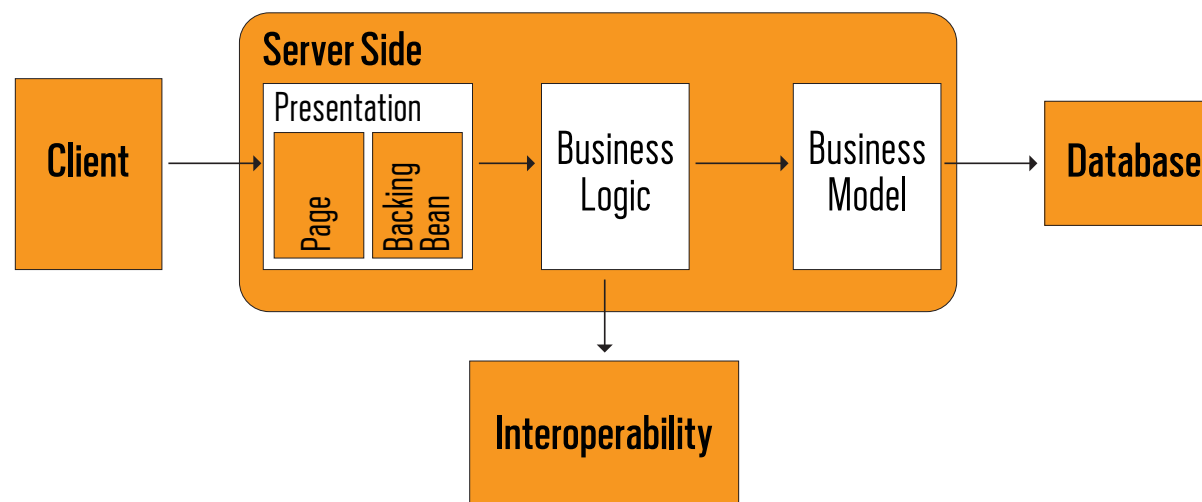


図1.アプリケーションの標準的な層

■ リスト1:

```
// Value Expressions
#{purchaseOrderBean.subtotal}
#{purchaseOrderBean.customer.name}

// Array Expressions
#{purchaseOrderBean.orders[2]}

// Method Expressions
#{purchaseOrderBean.compute}

// Parameterized Method Calls
#{purchaseOrderBean.compute('5')}
```

値式は、データの読取りや書込みに利用できるため、もっともよく使われます。ここでは、ページからPurchaseOrderBeanのsubtotal属性や顧客のname属性にアクセスしています。この構文を使うと、角括弧でインデックスを指定して配列やリストの項目にアクセスすることも可能です。例にあげた式では、Beanのインデックスの値が2である発注を返しています。ELの機能のもう1つの便利な点は、メソッド式がサポートされていることです。メソッド式は、Beanのパブリック・メソッドを呼び出すために使用します。対象メソッドに戻り値があっても構いません。ここでは、式でPurchaseOrderBeanのcomputeメソッドを呼び出しています。パラメータのあるメソッド呼出しには、パラメータを使用できます。ここでは、計算に使う値として数値の5を渡しています。

JSFページ:ここで、もう一度プレゼンテーション層に戻ります。ELは、さまざまな形でJSFページに存在しています。たとえば、リスト2では、発注の小計や付加価値税 (VAT) の率を表示するために値式が使われています。このバインディングは双方向です。つまり、ページがサーバーに送信されると、式によってこれらの属性の値が変更されます。発注額を計算するなど、ボタンがクリックされたときにアクションを実行する必要がある場合、メソッド式を使用すると便利です。ここでは、Computeボタンをクリックした際にPurchaseOrderBeanの計算メソッドを呼び出しています。

■ リスト2:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
<h:body>
  <h:form>
    <h:outputLabel value="Subtotal:"/>
    <h:inputText
      value="#{purchaseOrderBean.subtotal}"/>
    <h:outputLabel value="VAT rate:"/>
    <h:inputText
      value="#{purchaseOrderBean.vatRate}"/>
    <h:commandLink value="Compute"
      action="#{purchaseOrderBean.compute}"/>
  </h:form>
</h:body>
</html>
```

CDI Bean: リスト3のPurchaseOrderBeanにはsubtotalとvatRateという属性があり、それぞれgetterとsetterを持っています。また、指定されたVAT率を使って発注の合計金額を計算するメソッドもあります。このクラスは、@Namedアノテーション以外に特に目立つ点はありません。仮にこのアノテーションがなければ、このBeanはEL名を持たないため、ページにバインドすることができなくなります。



31

■ リスト8:

```
@Named
@RequestScoped
public class BookService {

    public Book persist(Book book) {
        // ...
    }

    public List<String> findAllImages() {
        // ...
    }

    public List<Book> findByCategory(
        long categoryId) {
        // ...
    }
}
```

会話スコープ:最後の組込みスコープは、会話スコープです。会話スコープは、ユーザーに関連付けられた状態を保持し、サーバーへの複数のリクエストにまたがって存在するという点でセッション・スコープと似ています。ただし、セッション・スコープとは違い、会話スコープはアプリケーションによって明示的に存続期間が定められます。たとえば、顧客がプロフィールを作成するためのウィザードが複数のWebページで構成されるとします(リスト9を参照)。会話のライフサイクルを制御するために、CDIではConversation APIが提供されています。これは、インジェクションによって取得できます。ユーザーがプロフィールの作成を始める際は、beginメソッドを呼び出して会話を開始します。ユーザーは、会話が終わるまで次のページや前のページなど、自由にページを移動できます。ここから分かるように、会話スコープは明示的に存続期間を定める必要がある唯一のスコープです。他のスコープを持つBeanはすべてCDIコンテナによってクリーンアップされます。会話の場合は、明示的に開始と終了を行う必要があります。それを怠ると、タイムアウトが発生します。

■ リスト9:

```
@Named
@ConversationScoped
public class CustomerWizard implements
    Serializable {
```

```
@Inject
private Conversation conversation;

private Customer customer =
    new Customer();

public void initProfile () {
    conversation.begin();
    // ...
}

public void endProfile () {
    // ...
    conversation.end();
}
}
```

依存スコープ:ここまで見てきたスコープは、すべてコンテキストが明確なスコープです。つまり、コンテナがスコープのライフサイクルを管理しており、インジェクションされるBeanへの参照もコンテキストが明確です。CDIコンテナは、これらのオブジェクトに指定されたスコープどおり、正しいタイミングでオブジェクトを作成し、インジェクションすることを保証しています。依存スコープは、コンテキストが不明確なスコープで、実際には疑似スコープと呼ばれています。デフォルトのCDI Beanスコープは依存スコープです。Beanが特定のスコープを宣言していない場合、依存スコープBeanとしてインジェクションされます。これは、インジェクションされるBeanと同じスコープを持つという意味になります。たとえば、リスト10の例では、リクエスト・スコープを持つサービス (**BookService**) が依存スコープの**IsbnGenerator**をインジェクションする場合、インジェクションされた**IsbnGenerator**もリクエスト・スコープになります。依存Beanのインスタンスは、もう一方のオブジェクトに厳密に依存します。**つまり、IsbnGeneratorは、BookServiceが作成されたときにインスタンス化され、BookServiceが破棄されたときに破棄されます。**@**Dependent**アノテーションはいつでも使用できますが、これはデフォルトのスコープなので、明示的に記述しなくても構いません。




```
public class IsbnGenerator {
    public String generateNumber() {
        return "13-84356-" +
            Math.abs(new Random().nextInt());
    }
}
```

```
public class BookService {
```

```
@Inject
private IsbnGenerator generator;
// ...
}
```

本記事では、@Namedアノテーションによるバインディングとスコープによる状態管理を活用し、Web層とサービス層を結合する方法について見てきました。CDIを使用すると、プレゼンテーション層のコンポーネントとビジネス・ロジック層のコンポーネントの区別がなくなります。どちらのコンポーネントでも、スコープを定義でき、インジェクションでき、ELから参照できます。アプリケーション・ロジックを強制的に技術的な階層に当てはめるのではなく、それがどんなものであれ、必要なアーキテクチャに応じてアプリケーションを階層化できるようになります。アーキテクチャ階層がフラットなものなら、CDIを使用して同じような階層を持つアーキテクチャを作成するのもよいでしょう。あらゆるものがCDI BeanであるJava EEアプリケーションを書くことも可能です。

- [CDI仕様](#)
- [Beginning Java EE 7](#)
- [PluralSightのCDI 1.1講座](#)
- [Weld CDIリファレンス実装](#)

ダイジェストとレビュー

年に1度のJava開発者のお祭りであるJavaOneが、10月下旬にカリフォルニア州サンフランシスコで開催されました。今年は、9,000名を超える開発者が500近くのセッションに参加しました。1人当たりの参加セッション数は、平均で14セッションにのぼりました。

今年のカンファレンスの根幹をなすテーマは、Java の 20 周年記念と、クラウドやモノのインターネット (IoT) の領域で高まる Java の存在です。

今回のイベントで発表された新しいクラウド・サービスの中で特筆すべきなのが、[Java SE Cloud Service](#) と呼ばれるオラクルのサービスです。このサービスには、Java や、Git、Maven、Hudson などの一連の開発ツールが組み込まれています。これらはすべて、プログラミングをクラウドに移行することを目的としています。

また、IoTトラックで注目を集めたのが、小型デバイスにスケールダウンできる Java の能力です。このテーマは、ベンダー展示エリアの Java Lounge でも注目されました。そこでは、参加者がはんだごてなどの道具と、ここ数年でたいへんな人気を集めている愛好家向け小型技術である Raspberry Pi を使って、デバイスの構築を体験しました。イベント前の土曜日には、子供向けの技術コンベンションである JavaOne4Kids も開催され、450 名の参加者が Java でロボットをプログラミングする方法を学びました。

JavaOne で毎年恒例になっているのが、特に功績のあった Java プロジェクトやコミュニティ・メンバーを表彰する [Duke's Choice Awards](#) です。今年の受賞者は、AsciiDocFX（ドキュメント生成ツール）、Byte Buddy（19 ページで詳しく解説しているバイトコードの生成および操作ライブラリ）、OmniFaces（Web アプリケーション用ライブラリ）、KumuluzEE（マイクロサービス有効化技術）でした。

オラクルはほとんどのセッションを録画し、オンラインで無償公開しています。見逃したセッションのビデオを見るには、綿密に分類、整理された[一覧](#)を参考にするといいでしょう。

次回の JavaOne カンファレンスは、2016 年の 9 月 18 日から 22 日にかけて、サンフランシスコで開催される予定です。その他のカンファレンスやイベントの一覧については、今号のイベントのセクションをご覧ください。



Jim Baker: Jython 開発の中核を担う。『The Definitive Guide to Jython』(Apress) の共著者で、Rackspace の上級ソフトウェア開発者と Python Software Foundation のフェローを務める。

Josh Juneau: アプリケーション開発者、システム・アナリスト、DBA。『The Definitive Guide to Jython』の共著者で、その他にも Java プログラミングに関する多くの著作を発表している。

PythonとJavaのライブラリを使用してプロジェクトを簡単に作成できる言語

本記事では、Apache POIを使用してスプレッドシートを操作するという簡単な例をもとに、Jython 2.7の機能を詳しく解説します。本記事でJythonの機能を理解すれば、最新リリースをダウンロードしてプロジェクトを開始することも十分可能でしょう。

Jythonは、ひとことで言うとJavaプラットフォーム用のPythonですが、Python言

Jythonプログラムを引数なしで実行すると、コンソールが起動します。リスト1に示すように、本記事では執筆時点の最新リリース・バージョン(2.7.0)を使用します(**注:**本記事では、UNIX系システムの使用を想定し、bashやその他のシェルのコマンドライン・プロンプトとして「\$」を使用しています。Jython 2.7は、Windowsでも問題なく動作します)。

```
$ jython
Jython 2.7.0 (default:..., Apr 29 2015, 02:25:11)
[Java HotSpot(TM) 64-Bit Server VM (Oracle Corp.)]
Type "help", "copyright", "credits" or "license"
for more information.
>>>
```

コンソールには、入力の先頭行を示すプロンプト「>>>」が表示されます。まずは、dict型を使用するところから始めましょう。これは、ディクショナリ(辞書)、すなわちキーと値を持つ変更可能なマップです。キーと値の両方に任意のオブジェクトを使用できます。後ほど詳しく説明しますが、PythonオブジェクトでもJavaオブジェクトでも構いません。このような万能性のため、ディクショナリはほとんどのPythonプログラムで頻繁に利用されています。リスト2は、ディクショナリの使用例です。

■ リスト2:

```
>>> d1 = {'one':1, 'two':2, 'three':3}
>>> d1['three']
3
>>> # Equivalent construction by using keywords
>>> # Note that '#' introduces a comment, including
    in the console
>>> d2 = dict(one=1, two=2, three=3)
>>> d2['one']
1
>>> d1 == d2
True
>>> len(d1) # length of d1
3
>>> # Note that there is no construction equivalent
    using keywords,
>>> # because keywords are limited to strings that
    would also be
>>> # valid Python identifiers.
>>> inverted = {'one':1, 'two':2, 'three':3}
```

Python 2.7では、ディクショナリの内包表記のサポートも追加されています。内包表記は特殊な構文のひとつで、生成式をもとに特定の型のコレクションを構築するために使います。元となるディクショナリのすべての項目について、値をキーにマッピングする(キーと値を逆転させる)内包表記は、次のように記述できます。

```
>>> inverted_d1 =
...{ v: k for k, v in d1.iteritems() }
```

(ドット3つは、すべてのテキストを1行で入力することを意味します)このコンパクトな内包表記の構文にはいくつかのバリエーションがあり、リストやセットの作成にも使用できます。

マッピングを逆転させるこの便利な機能は、後々の再利用の可能性があります。そこで、この処理を関数として定義しておきましょう。関数の定義は、コンソールでPythonによって記述することもできます。しかし、ここでは別の方法でコンソールを使用する方法を説明しましょう。

`basics.py`という名前のファイルを作成し、ファイルの中身に次のテキストを入力します。

```
def inverted(d):
    ... return { v: k for k,v in d.iteritems() }
```

(ドット3つは、テキストを1行で入力することを意味します)このコード・フラグメントを詳しく見てゆきます。まず、関数(この例ではinverted)はdefキーワードで定義することが分かります。Pythonでは、中括弧などの構文ではなく、空白文字によってプログラムの階層構造を定義します(ここで使用している中括弧は、ディクショナリの内包表記です。中括弧は、dictまたはsetコレクションを作成することを意味します。通常、Pythonコードのインデント・レベルは4つの空白文字で表しますが、この記事では、スペースの制約の関係上、2つの空白文字を使用しています)。Pythonの哲学はシンプルで、コードが構造に対応するようにインデントされているなら、中括弧でくるような構文は冗長だと考えます。ただし、さまざまな細かい表記ルールと同じように、Pythonのアプローチに慣れるまでには少し時間がかかるかもしれません。

それでは、もう一度Jythonコンソールを起動します。ただし今回は、`jython27 -i basics.py`と実行して先ほど作成したファイルを読み込みます。コンソールが起動すると、標準のプロンプト「>>>」が表示されます。ここで、何が読み込まれているかを確認するために、`dir`関数を呼び出します。引数なしで`dir`関数を呼び出すと、現在のモジュールが対象となります。

```
$ jython27 -i basics.py
>>> dir()
['__builtins__', '__doc__',
 '__file__',
 '__name__', '__package__',
 'inverted']
>>> inverted({'1': 'one', '2': 'two'})
```


36

です。ところで、このコードには、Javaで指定したり、Scalaで推論されるような静的な型が存在していません。静的なプログラムの解析（字句解析）を行うと、プログラムのテキストを実行せずに調べるだけで、コンパイラやIDEなどのツールがプログラムの特性を判断できます。変数のスコープはどこまでか。変数の型は何なのか。型は一貫性を持って使用されているか、つまり、コードで型チェックが行われているか。到達不能であるため削除できるコード（デッド・コード）はあるか。定数の畳み込みやインライン化が可能か。そういったことを判断できるのです。ここで述べたような項目のうち、Jythonがサポートしているのは、変数のスコープを静的に判断することのみです（CPythonは、ある程度の定数畳み込みやデッド・コードの削除を行うことができます。Python 3.5には、漸進的型付けのサポートの一環として標準の静的型アノテーションが導入される予定です。漸進的型付けとは、動的な型アプローチと静的な型アプローチを組み合わせた型システムです）。

このコードでは、`callback`が定義されていない場合に関数を定義しており、その名前は`callback`です。こうしたスタイルに最初は少し戸惑うかもしれません。この関数は、`process_workbook`関数のスコープの内部で定義されています。実際は`callback`関数はクロージャです。スコープが字句解析によって決定されるだけでなく、条件を満たす場合のみ定義されています。この動作は、Javaの動作とはまったく異なります。ここからも、Python言語の動的な性質がわかります。どれほど`process_workbook`を静的に解析しても、`callback`が目の前の関数を指すのか、あるいは指さないのかを判断することはできません。しかしよく考えると、ソース・コードはJythonによってすでにJavaバイトコードにコンパイルされています。そうするとここでの問題は、コンパイル済みの関数の本体に、`callback`という名前が割り当てられるかどうかということになります。結局のところ、この条件付き定義のオーバーヘッドは、変数割り当て程度のものでしかないというのが実際のところです。このことから、Jythonを使うとJavaの手法とPythonの手法を自在に使い分けられることがわかります。

さて、次は各ワークブックのスプレッドシート、各スプレッドシートの行、そして各行のセルに対して反復処理を行ってみましょう。ワークブック、スプレッドシート、行といったオブジェクトは、すべてJythonが反復処理できる `java.lang.Iterable` を実装しています。驚くまでもないかもしれませんが、Jython で実現されている統合によって、Javaコードがfor-eachループでPythonの `iterable` (と `iterator`) の反復処理を行うことができるようになっています。

`callback(cell)`で`cell`に対して`callback`が呼ばれると、Jythonランタイムは`callback`オブジェクトが呼び出し可能 (callableオブジェクト) であるかという動的な型チェックを行います。Pythonの単純なルールとして、

すべてのcallableオブジェクトは、特別なメソッド `__call__` を実装しています。関数はすべてこの特別なメソッドを実装していますが、任意のクラスもこれを実装できます。Pythonでは、この型付けアプローチはダック・タイピングと呼ばれています。この名前は、アヒルのように見え、アヒルのように泳ぎ、アヒルのように鳴くものは、おそらくアヒルであるということに由来しています。Pythonは、開発者が自分で何をしているのかわかっているという前提に立つため、呼び出しは自由です。

ただし、プログラムが実行されたときに、`__call__`という特別なメソッドが対象のオブジェクトで利用できなければ、Pythonの`TypeError`例外が発生します。もちろん、`__call__`自身で例外が発生する可能性もあります。

それでは、コールバックを定義し、スプレッドシートにハードコーディングされた式をExcelの式と同様に検査してみましょう。セルに式が存在する場合、その式の文字列は`getCellFormula()`メソッドで取得できます。POIの式は、Excelの式とは違って「=」記号が頭に付いていないので注意してください。

Pythonはメソッドだけでなくプロパティもサポートしています。そのため、JythonではJavaオブジェクトの高度な操作が可能で、getterやsetterは[get](#)や[set](#)を省略してプロパティのように扱うことができます。検査用のコールバックは、次のように記述できます。

```
def print_if_hardcoded(cell):
    try:
        float(cell.cellFormula)
        ref = CellReference(cell)
        print ref.formatAsString(), cell
    except:
        pass
```

このコードには、Pythonをはじめとする動的言語でよく目にするパターンが含まれています。まず何かを実行してみて、例外をキャッチするという方法です(このパターンは俗に、「許可を求めるより謝罪するほうが簡単」と呼ばれています)。ここでは、式の文字列の取得(取得できない場合は、POIが `IllegalStateException` 例外を発生させます)と文字列から浮動小数値を生成(失敗した場合は、Pythonの `ValueError` 例外が発生します)するという2つのアクセッ

**Jythonを使う
と、Javaの手法
とPythonの手
法を自在に使
い分けることが
できます。**

40

答えは、`any(hardcoded_cells(get_cells(spreadsheet, "A1:G8")))`で得られます。

ここで行ったことは、ワークブックを操作する高レベルPython APIの第一歩を定義するものです。こうしたAPIは、スプレッドシートの中で使用する関数にある程度似ています。そして一方、JavaのPOIライブラリの低レベルな機能も使うことができます。

最後のトピックは、この点に関することです。こういったスプレッドシートは、コンプライアンス・テストに合格できるでしょうか。少し複雑ではありますが、Jenkinsなどの継続的インテグレーション・サービスをセットアップし、GitHubのプル・リクエストの一環としてスプレッドシートのテストを実行することを考えてみましょう。その場合、このテストはどのように定義して実行すればよいでしょうか。Pythonのエコシステムには、いくつかの優れたテスト・フレームワークがあります。たとえば、標準ライブラリや、xUnit形式のテストを実装した[unittest](#)などです。しかしそれ以外にも選択肢があり、[unittest](#)上に構築されるNoseテスト・フレームワークは、たいへん使いやすいことから広く普及しています。

たとえば、あるクロス集計が正しいことを確認したいとします。数値の精度の問題が考慮されていれば、クロス集計の行の小計の合計は列の小計の合計と一致します。リスト9をご覧ください。

■ リスト9:

```
from nose.tools import assert_almost_equals
def assert_crosstab(spreadsheet, range1, range2):
    assert_almost_equals(
        sum(get_nums(spreadsheet, range1)),
        sum(get_nums(spreadsheet, range2)))
```

この関数を定義すると、リスト10のような簡単なテスト・スクリプトを書くことができます。

■ リスト10:

```
finance_wb = XSSFWorkbook("financials.xlsx")
main_sheet = finance_wb.sheetAt(0)
```

```
def test_financials():
    assert crosstab(main_sheet, "A5:G5", "H1:H4")
```

Noseを実行すると、テストの検出と実行が行われます。Noseは「設定より規約」というアプローチに従っています。規約に従わない部分のみ記述すればよい

ので、簡単に使い始めることができます。結果は次のようになります。

```
$jython -m nose
.
-----
Ran 1 test in 0.033s
```

OK

2行目の各ドットは、Noseが収集したテスト・ファイルの各テストに対応しています。テストは簡単に追加できます。

下位互換性

当然ではありますが、時間が経つにつれて、テクノロジーや言語の機能は進化してゆきます。そのため、Jython 2.7で廃止されたいくつかの重要な機能があります。その中で最も注目すべきなのは、Jython 2.7にはJava 7以上が必要となることでしょう。また、もう1つの重要な点は、インストーラがJythonランチャーを生成する際に、代替JREの使用をサポートしなくなった点です。そのため、[JAVA_HOME](#)を利用する必要があります。

Jython 3.5

Python言語では、リファレンス実装を含む開発が継続的かつ活発に行われています。この記事が掲載される頃には、CPython 3.5がリリースされていることでしょう。今後、CPython 3.5リリースに対応するJython 3.5のリリースも計画されています。そして覚えておいて損はないと思いますが、Jython 2.7では、基本的にPython 3.2と同じ内部ランタイムや標準ライブラリがサポートされています。しかし、まだ多くの作業がJython 3.5のリリース前に残されています。Python 3.5で切望される機能の1つが、省略可能な静的型付けです。実現すれば、さらに密接なJavaとの統合がJythonで可能になります。

しかし、開発のペースはさほど早くはありません。しばらくの間は、Jython 2.7.xが使われることになるでしょう。Python 2.7が広く使用されているかぎり、Jythonチームは2.7.xの作業を継続する予定です。Python 3の採用や移行は、かなりスロー・ペースです。バージョン2.7と3.0の間で、相当大きな変更が行われていることもその一因でしょう。Python 2.7は今でも広く使用されているため、JythonチームはJython 2.7.x系のリリースを定期的に行う予定です。今後のJython 2.7.xのリリースは、パフォーマンスや統合などに関する機能が中心となるはずです。動的言語向けの最適化が進むと考えられるJava 9がリリースされ

ば、パフォーマンスは向上するでしょう。

Jython 3.5の開発は急がれてはいませんが、計画は進行中です。実際、初期段階ではあるものの、Jython 3.5の開発用のブランチはすでに存在しています。現在のところ、Jython 3.5のリリースは今後2年で行われる予定です。

まとめ

Jython 2.7は豊富なツールを提供しています。Jython 2.7を使用することで、開発者はPythonとJavaという最も人気のある2つのエコシステムを同じコードベースで組み合わせることができるようになります。本記事では、Jython 2.7のいくつかの主要な新機能を見てきました。しかし、注目すべき機能は他にもたくさんあります。ぜひjython.orgからJython 2.7をダウンロードし、6カ月ごとに行われるアップデートにもご期待ください。

[編集注:本記事は、JVM言語を紹介するシリーズの一部です。前号では、Kotlinについて解説しました。次号では、業界のフロントエンド・システムとバックエンド・システムの両方に利用されているJVM言語Gosuを取り上げます]



Java Webアプリケーションのパッケージ化とデプロイメントに利用できる人気急上昇中の軽量仮想化コンテナ

(@arungupta) : Couchbase の開発者支援部門のバイス・プレジデント。Java Champion であり、JUG のリーダーも務める。かつての Java EE チーム設立時のメンバー。

全2回シリーズの第1回である本記事では、Dockerの基本概念と動作の仕組みについて説明します。また、Toolboxを使用したDockerの導入方法を紹介し、インストール後の動作を確認する方法をシンプルな「Hello World」アプリケーションを例に説明し、Dockerイメージの概念とその作成方法についても解説します。実際にDockerイメージとしてJavaアプリケーションをパッケージ化し、コンテナとして動作させるので、Dockerの基本を理解できるはずです。イメージやコンテナを調査する基本的なコマンドについても説明し、最後に、コンテナを使用してJava EE (WildFly) アプリケーションをデプロイする方法を紹介します。第2回の記事では、クラスタなどの複数のDockerコンテナを必要とするアプリケーションの作成方法と、Dockerと他のツールの統合について取り上げる予定です。

一般的に、アプリケーションが動作するためには、特定のバージョンのオペレーティング・システム、JDK、アプリケーション・サーバー、データベース・サーバー、その他のインフラストラクチャ・コンポーネントが必要です。最適なエクスペリエンスを実現するためには、特定のポートや特定の量のメモリをバインドし、さらにコンポーネントによって異なる設定を行う必要もあるかもしれません。それらすべて、つまりアプリケーション、インフラストラクチャ・コンポーネント、設定を、

Dockerのコンテナを使用すると、コンポーネントの分離、サンドボックス化、再現性、リソース制限、スナップショットの利用といったメリットを得ることができます。コンテナはハイパーバイザがなくても実行できます。また、軽量であるため、標準のVMよりもかなり高い密度で実行できます。

- **イメージ**: Dockerのビルド・コンポーネントで、アプリケーション運用システムの読み取り専用のテンプレートで構成されています。イメージの例として、WildFlyとJava EEアプリケーションがインストールされたFedoraオペレーティング・システムがあげられます。新しいイメージの作成や、既存のイメージの更新は簡単に行うことができます。
- **コンテナ**: イメージから作成されるランタイムです。これは、Dockerの実行コンポーネントであり、実行、開始、停止、移動、削除といった操作が可能です。各コンテナは、独立して安全に実行できるアプリケーション・プラットフォームです。

Dockerを導入する**もっとも簡単な方法**が、**Docker Toolbox**です。

- Docker Compose ([docker-compose](#)バイナリ) : 多くの場合、アプリケーションはWildFly、MySQL、Apache Webサーバーなどの複数のコンテナで構成されます。Docker Composeを使用すると、マルチコンテナ・アプリケーションの定義や実行が単一の構成ファイルで実現できます。
- Kitematic: コンテナを管理するための強力なGUIで、コマンドライン・インタフェースとGUI画面の間でシームレスな操作が行えるようになっています。また、Docker Hubとも統合されています。
- Docker Quickstart Terminal: ターミナル・アプリケーションです。デフォルトのDocker Machineを作成し、Toolboxのインストールによって作成されたデフォルトのDockerホストと通信するようにDockerクライアントを設定します。
- Oracle VM VirtualBox 5.0.0: ローカル・マシン上にDockerホストを作成する仮想化プロバイダです。

以上のコンポーネントは個別にダウンロードできますが、Docker Toolboxとしてパッケージ化されているので一括ダウンロードが可能です。Docker導入のもっとも簡単な方法は、このDocker Toolboxを使うことです。

まずDocker Toolboxをダウンロードし、マシンにインストールします。Docker Quickstart Terminalを実行すると、デフォルトのDockerホストが作成され、そのDockerホストと通信するDockerクライアントが設定されます。画面には次の内容が出力されます。

```
Creating Machine default...
Creating VirtualBox VM...
Creating SSH key...
Starting VirtualBox VM...
Starting VM...
To see how to connect Docker to this machine,
run: docker-machine env default
Starting machine default...
Setting environment variables for machine default..
```

Docker イメージは、テキスト・ファイルに記述された指示に従ってビルドされます。通常、このファイルは Dockerfile という名前で、イメージのビルドに必要なすべての情報が含まれています。

この出力から、VirtualBox VMにDockerホストが作成され、SSHキーが生成され、VMが起動して、Dockerホストと通信できるようにDockerクライアントが設定されたことが分かります。クライアントは、DOCKER_HOSTやDOCKER_CERT_PATHなどの環境変数によって設定されています。この環境変数の設定は、先ほどの出力からも分かるように、docker-machine env defaultコマンドで行われています。マシン名は、defaultとなっています。

eval \$(docker-machine env default)コマンドを使用すると、このホストと通信するシェルを設定できます。最後に、次のような内容が出力されます。

```
docker is configured to use the default machine
with IP 192.168.99.100
```

docker-machine ip defaultコマンドを実行すると、このホストに割り当てられたIPアドレスが表示されます。名前のマッピングのために、/etc/hostsファイルや使用しているオペレーティング・システムでそれに相当するファイルにIPアドレスを設定することを推奨します。たとえば、次の行を追加します。

```
192.168.99.100 dockerhost
```

Dockerホストのマッピングが正しいことを確認するために、ping dockerhostを実行します。以上で、DockerクライアントがDockerホストと通信する準備が整いました。

Docker Hello World

実際にDockerでHello Worldサンプルを実行する前に、いくつかの基本的なコマンドを確認します。

docker imagesは、ホスト上で利用できるイメージの一覧を表示します。

docker psは、実行中のコンテナの一覧を表示します。現時点では、どのコンテナも開始していないので、コンテナの一覧は空になります。すでに終了したコンテナも含めて表示したい場合は、-aオプションを指定します。このコマンドの出力結果は、1行128文字の幅で表示するときれいに見えます。

なお、docker --helpを実行すると、すべてのコマンドの一覧が表示されます。同様に、docker ps --helpを実行すると、このコマンドで利用できるすべてのオプションが表示されます。各コマンドの動作が簡単に調べられるので、ニーズに応じた使い分けが可能です。



次に、ビルド済みの状態でDocker Hubに格納されている「Hello World」Dockerイメージを実行します。次のコマンドで実行できます。

```
docker run hello-world
```

このコマンドで、次の内容が出力されます（テキストの前後には、別のメッセージも表示されます）。

```
Hello from Docker.
```

出力内容から、次のことが確認できます。

- DockerクライアントとDockerデーモンが正常にインストールされた。
- Dockerデーモンではhello-worldイメージが利用できないが、Docker Hubからイメージをダウンロードできた。
- ダウンロードしたイメージを使ってコンテナが起動され、結果がDockerクライアントにストリーム出力された。

このように、初めてであっても面倒な手順を踏まずにコンテナを起動できます。

Javaを使った最初のDockerイメージをビルドする

DockerイメージはDockerコンテナの起動に使用される読み取り専用のテンプレートで、各イメージは一連のレイヤーで構成されています。Dockerは、複数のレイヤーを1つのイメージにまとめるために、[Unionファイル・システム](#)を使用しています。Unionファイル・システムは、別々のファイル・システムのファイルやディレクトリを透過的に重ね合わせ、一貫性のある1つのファイル・システムを形成するものです。

Dockerが非常に軽量なのは、このレイヤーのおかげです。アプリケーションが新しいバージョンにアップデートされたりJDBCドライバが変更されたりするとDockerイメージの変更が発生しますが、影響するレイヤーを再ビルドするだけで対応できます。そのため、VMと違って、イメージ全体の置換や再ビルドの必要がなく、1つのレイヤーの追加や更新だけで済みます。その結果、配布も高速かつシンプルになります。

すべてのイメージは、ベースとなるオペレーティング・システムのイメージから始まります。たとえば、fedoraはベースとなるFedoraのイメージです。そこに複数のレイヤーを追加します。たとえば、図2に示すように、jboss/wildflyは複数のイメージを使用してビルドされています。

Dockerイメージは、テキスト・ファイルに記述された指示に従ってビルドされ

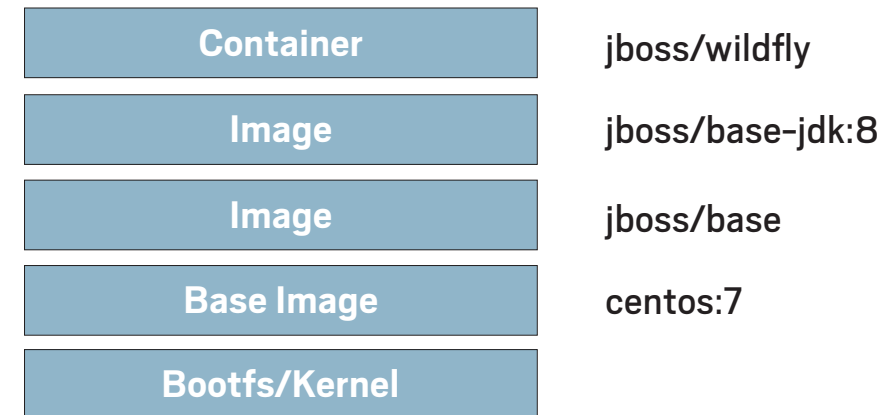


図2.Java EE用のDockerイメージのビルド

ます。通常、このファイルはDockerfileという名前で、イメージのビルドに必要なすべてのコマンドが含まれています。たとえば、ベースとなるオペレーティング・システム、JDK、アプリケーション・サーバー、その他の依存関係などが指定されています。通常、イメージにJDKやアプリケーション・サーバーをダウンロードしてインストールするには、GET、COPY、RUNといったシェルのようなコマンドを使用します。COPY命令は、ローカル・ファイル・システムからコンテナにファイルをコピーする場合にも使用します。オプションとして、JDKやWildFlyがすでに含まれるベース・イメージを使用し、その上にビルドを行うこともできます。Docker Hubにはさまざまなベース・イメージがあるため、ニーズに一致するものを見つけることができます。

Dockerfileには、CMD命令を1つ含めることができます。これは、コンテナ起動時に使用する実行コマンドを指定するものです。複数のCMD命令が指定されている場合、最後のCMD以外は無視されます。

構文の完全なリファレンスは[オンラインで参照可能](#)です。同様に、[ベスト・プラクティス](#)も公開されています。



JDKのバージョンを表示するだけのシンプルなDockerfileは、次のようになります。

```
FROM java:8
CMD ["java", "-version"]
```

これをDockerfileという名前のファイルにコピーし、イメージをビルドします。

```
docker build java-version .
```

このbuildコマンドが、java-versionという名前のDockerイメージをビルドしています。最後の「.」は、イメージをビルドするための命令ファイルがカレント・ディレクトリにあることを示しています。

イメージをビルドすると、参照できるようになります (次の表示例は一部のみで、出力されるすべてのフィールドを表示したものではありません)。

```
> docker images
REPOSITORY TAG IMAGE ID VIRTUAL SIZE
java-sample latest 53bd2cdf4aa2 425.4 MB
```

`docker run java-sample`コマンドでこのコンテナを起動すると、次の出力を確認できます。

```
openjdk version "1.8.0_66-internal"
```

docker psを単独で使用しても、出力にコンテナは表示されません。これは、コンテナが起動中ではないためです。docker ps -aコマンドを使用すると、終了したコンテナを確認できます。

このコンテナの一部としてJARファイルを実行する場合は、COPYを使用してローカル・ファイル・システムからファイルをコピーするか、GETを使用してJARファイルをダウンロードします。その後、CMDコマンドラインにJARファイルを含めます。JVMの構成の設定は、同じ方法ですべて適用できます。

Dockerを使用してJava EEアプリケーションをデプロイする

ここまでは、ごく基本的な例を実行してきました。続いて、WildFlyコンテナにJava EEアプリケーションをデプロイする方法を見てゆきましょう。ここで使用するDockerfileは次のとおりです。

```
FROM jboss/wildfly
CMD ["opt/jboss/wildfly/bin/standalone.sh", "-c", "standalone-full.xml", "-b",
"0.0.0.0"]
RUN curl -L https://github.com/javaee-samples/
javaee7-hol/raw/master/solution/movieplex7-
1.0-SNAPSHOT.war -o opt/jboss/wildfly/standalone/
deployments/movieplex7-1.0-SNAPSHOT.war
```

[編集注:最後の2つのコマンドは、折り返されていることに注意してください]

このファイルでは、jboss/wildflyベース・イメージを使用しています。WildFlyは /opt/jboss/wildflyディレクトリにプレインストールされており、WildFlyコンテナの起動時にこのディレクトリが使用されます。-bオプションによって、ネットワーク・インタフェースはパブリックに利用できるすべてのIPアドレスにバインドされます。また、リポジトリからWARファイルをダウンロードしていますが、そのコピー先はWildFlyがデプロイメント対象を監視するディレクトリです。

このイメージを、次のコマンドでビルドします。

```
docker build -t javaee-sample .
```

次に、イメージを実行します。

デフォルトではDockerコンテナがフォアグラウンドで起動されますが、ターミナルとの対話は許可されていません。`-i`オプションを指定すると、標準入力との対話が可能になり、`-t`オプションを指定するとプロセスにTTY(コンソール)をアタッチできます。オプションは組み合わせられるので、`-it`として`-i`と`-t`を一緒に指定します。

次のコマンドで、コンテナを起動します。

```
docker run -it -p 8080:8080 javaee-sample
```

8080はWildFlyイメージが外部と通信するポートです。-pオプションを使用してホスト上のポートと明示的にマッピングする必要があります。この場合、最初の「8080」はホストにマッピングされたポートで、2番目の「8080」はコンテナ内のポートです。

コンテナの起動が完了すると、WildFlyにデプロイされたJava EEアプリケーションは、ローカル・マシンからdockerhost:8080/movieplex7でアクセスできるようになります。前述のDockerのホストとIPアドレスのマッピングに注意してください。図3は、このURLを開いた結果です。

対話モードでコンテナを実行した場合、[Ctrl-C]キーで停止できます。コンテナが停止したことは、次の出力で確認できます。

```
docker ps -a
```

```
CONTAINER ID  IMAGE          COMMAND
1efa5d6f618d  jboss/wildfly  "/opt/jboss/wildfly/b"

CREATED      STATUS      PORTS
About a minute ago  Exited (130) About a minute ago

NAMES
compassionate_mestorf
```

[編集注：十分な幅がある画面では、この出力は2行で表示されます]出力される各カラムには、コンテナについての有益な情報が含まれています。

- 各コンテナに割り当てられた一意のID (1列目)
- コンテナの起動に使用したイメージ名
- コンテナの起動に使用したコマンド
- コンテナが作成された時間
- コンテナが外部との通信に使用するポート (この例では、コンテナがすでに終了しているため、空欄となっています)
- 現在のステータス
- コンテナに割り当てられたランダムな名前 (--nameオプションで名前を指定した場合は、それが表示されます)

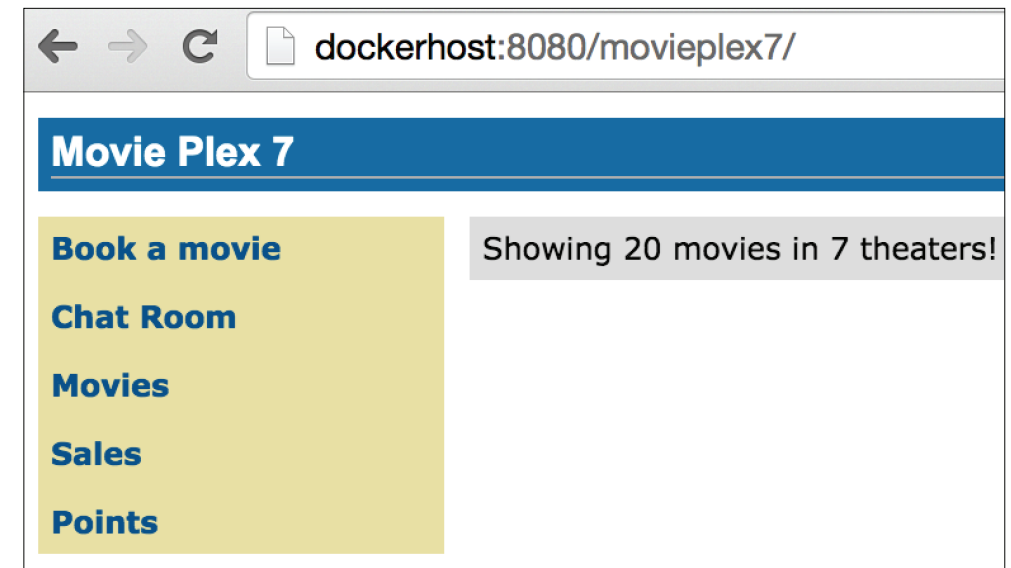


図3. ポート8080で稼働するサンプル・アプリケーション

Linux/UNIXベースのシステムでは、次のコマンドでコンテナのIDを取得できます。

```
docker ps | grep jboss/wildfly | awk '{ print $1 }'
```

コンテナの停止は `docker stop <CONTAINER_ID>` で、削除は `docker rm <CONTAINER_ID>` で行います。停止と削除を一度に行う場合は `docker rm -f <CONTAINER_ID>` を使用します。`docker restart <CONTAINER_ID>` で、コンテナの再起動も可能です。

コンテナを実行する別の方法として、デタッチ (バックグラウンド) モードも使用できます。その場合、-itを-dに変更します。

もう1つの重要なコマンドが、`docker inspect`です。このコマンドは、実行中のコンテナの詳細な情報を表示します。たとえば、次のコマンドを実行すると、コンテナ内のネットワーク・ポートを一覧表示できます。

```
docker inspect --format '{{ .Config.ExposedPorts }}' <CONTAINER_ID>
```



さらに、-Pオプションを指定すると、コンテナのポートをローカル・ホストのハイ・ポート（一般的には23768から61000の範囲のポート）にマッピングできます。コンテナのポートとホストのポートのマッピングは、次のようにして確認できます。

```
docker port <CONTAINER_ID>  
8080/tcp -> 0.0.0.0:32768
```

この場合、dockerhost:32768/movieplex7でアプリケーションにアクセスできます。

まとめ

本記事では、Dockerの基本概念とDockerでJavaアプリケーションをパッケージ化する方法を説明しました。Dockerは、Package Once Deploy Anywhere（一度パッケージ化すればどこでもデプロイできる）パラダイムを実現するもので、アプリケーションのビルド、デプロイメント、拡張方法を変革し、開発環境、テスト環境、本番環境の間の不整合を減らします。

準備できているかどうかにかかわらず、Dockerはすでに目の前にあり、今後長い間私たちの身近に存在し続ける軽量コンテナ技術となるはずです。本シリーズの次の記事では、マルチコンテナ・アプリケーションと、クラスタ内でのコンテナの実行について説明します。

LEARN MORE

- [Dockerを使ってみる](#)
- [コンテナの概要](#)
- [Kubernetes: Dockerオーケストレーション・ツール](#)

ブネー-JAVAユーザー・グループ



300万人以上が住むプネーはインドで9番目に人口が多い都市で、ソフトウェアの輸出が2番目に多い都市でもあります。プネーは数世紀にわたって学術や研究の中心として栄えており、一流の教育機関が複数存在しています。

ブネーJavaユーザー・グループは、インドでもっとも歴史のあるユーザー・グループの1つです。1990

年代後半に設立され、それ以来積極的に活動しています。このJavaユーザー・グループ(JUG)のリーダー陣は何度か代わってきましたが、新しいJavaテクノロジーの学習とJavaの愛好家同士の議論を推奨するという方向性は一貫しています。

通常、議論の中心となるのは、Java言語やJava EEに関するのですが、Javaプラットフォーム上で動作する他の言語についての議論も歓迎しています。たとえば、最近のJUGでは、Java ChampionのAndres Almiray氏によるGroovy言語とGradleビルド・ツールの話が注目を集めました。

このJUGのミーティングは、プネーの教育機関やソフトウェア会社で行われています。グループのメンバーは1,400名以上であり、Twitter ([@JavaPune](#)) や [Google Groups](#)などのさまざまなソーシャル・メディア・チャンネルで活動しています。Javaの新バージョンの発表イベントには、多くの愛好家に参加しています。

プネーのエコシステムは、生まれて間もないにもかかわらず活況を呈しており、このJUGも初心者向けの交流の場を支えるプラットフォームとして活用されています。このJUGは、さまざまなニッチな技術だけでなく、メインストリームとなる技術に関する点でも、プネーの技術コミュニティを支えています。

パート1

WebSocketを使用するアプリケーションの構築

長期存続型コネクションのための使いやすいAPI

DANNY COWARD

Danny Coward :
Liquid Robotics の
プリンシパル・
ソフトウェア・
エンジニア。
以前はオラクル（およびその
前の Sun Microsystems）
で Java 開
発チームに
属し、特に
WebSocket に
関する業務に
従事した。

Java WebSocketが他のJava EE Webコンポーネントと異なる点は、Webクライアントがデータを問い合わせなくても、Java EE WebコンポーネントからWebクライアントにデータをプッシュできることです。本記事では、WebSocketプロトコルの概要、WebSocketの仕組み、および単純なプロジェクトでのWebSocketの使用方法を説明します。Webアプリケーションの概要およびJava EEでのその動作方法に関する基本的な知識さえあれば、本記事を読み進めることができます。

Java WebSocketはHTTPベースの相互作用モデルから発展した考え方で、ブラウザ・クライアントやブラウザ以外のクライアントをJava EEアプリケーションのほうから非同期的に更新できるようにします。Webサイトの相互作用モデルとしては、長らくHTTPリクエスト/レスポンスという相互作用モデルが利用されてきました。この相互作用モデルは機能豊富で、多数の高度なブラウザベース・アプリケーションに適用できます。しかし、それぞれの相互作用はかならずブラウザから、ユーザー側の操作（ページの読み込み、ページの更新、ボタンのクリック、リンクのクリックなど）によって開始されます。

多くの種類のWebアプリケーションによって、ユーザーを画面の前に常に座らせておくのは望ましいことではありません。リアルタイムの市場データを用いる金融機関のアプリケーションから、世界中のユーザーが商品に入札するオークション・アプリケーション、簡単なチャットや在席確認アプリケーションまで、Webアプリケーションでは長らく、サーバー・サイドからクライアントにデータをプッシュできる手段が求められてきました。こうした

Java WebSocket API
には Java API クラス
および Java アノテー
ションが用意されて
いるので、Java EE
Web コンテナ内に常
駐する WebSocket エ
ンドポイントを比較
的簡単に作成するこ
とができます。

ニーズから、長期存続型のHTTPコネクションを保持し続ける、あるいは何らかのクライアント・ポーリング方式を取るといった、場当たり的な手法の数々が生み出されてきました。しかし、いずれも問題の完全解決には至りませんでした。アプローチの刷新が求められ、その結果、WebSocketプロトコルの開発につながったのです。

WebSocketプロトコルの概要

WebSocketプロトコルは、単一のコネクション上で全二重通信チャネルを確立するTCPベースのプロトコルです。端的に言えば、WebSocketプロトコルはHTTPと同じ基盤ネットワーク・プロトコルを使用し、単一のWebSocketコネクション上で、コネクションの両側が同時に相手にメッセージを送信できます。WebSocketプロトコルは、コネクションの単純なライフサイクルとデータ・フレーミング機構を定義しており、バイナリ・メッセージおよびテキストベースのメッセージをサポートしています。HTTPとは異なり、コネクションは長期存続型です。つまり、非対称のHTTPプロトコルとは異なり、メッセージ送信のたびにコネクションを確立し直す必要がありません。そのため、WebSocketプロトコルでは、全データ・メッセージにコネクションに関するすべてのメタ情報を持たせる必要がありません（HTTPではその必要があります）。言い換えれば、コネクションの確立後のメッセージ送信は、HTTPプロトコルと比較してはるかに軽量になります。

しかし、サーバーが情報をプッシュするというタスクにおいて、HTTPプロトコルの上の層にあるポーリング・フレームワークよりもWebSocketの方がふさわしいと言えるもっと大きな理由があります。それは、各クライアントに対して専用のTCPコネクションが作成されることです。データは必要なときだけ送信されるため、WebSocketによってサーバーからクライアントをより効率的に更新できるようになります。


```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection:Upgrade
Sec-WebSocket-Accept:HSmrc0sM1YUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: chat
Sec-WebSocket-Extensions: compress, mux
```

このレスポンスによって、クライアントから送られたTCPコネクション・リクエストをサーバーが受諾することが確認されます。また、コネクションの使用方法に関する制約が課されることがあります。クライアントがそのレスポンスの処理を完了し、課された制約を問題なく受け入れると、TCPコネクションが作成され（図1）、コネクションのそれぞれの側が相手側にメッセージを送信できるようになります。

コネクションの確立後は、以下のようにさまざまな応答が発生します。

- コネクションの一方の側が相手側にメッセージを送信する。このメッセージ送信は、コネクションがオープンしている間、いつでも発生する可能性がある。WebSocketプロトコルのメッセージには、テキストとバイナリの2種類がある。
- コネクション上でエラーが発生する。この場合、エラーによってコネクションが壊れなければ、コネクションの両側に通知される。コネクションが終了しないこのようなエラーは、通信の一方の側が送信したメッセージに形式上の誤りがあるような場合に発生する。
- コネクションが自発的にクローズされる。これは、コネクションの一方の側が、通信を終了し、コネクションをクローズすると決めたことを意味する。コネクションがクローズされる前に、コネクションの相手側に通知される。

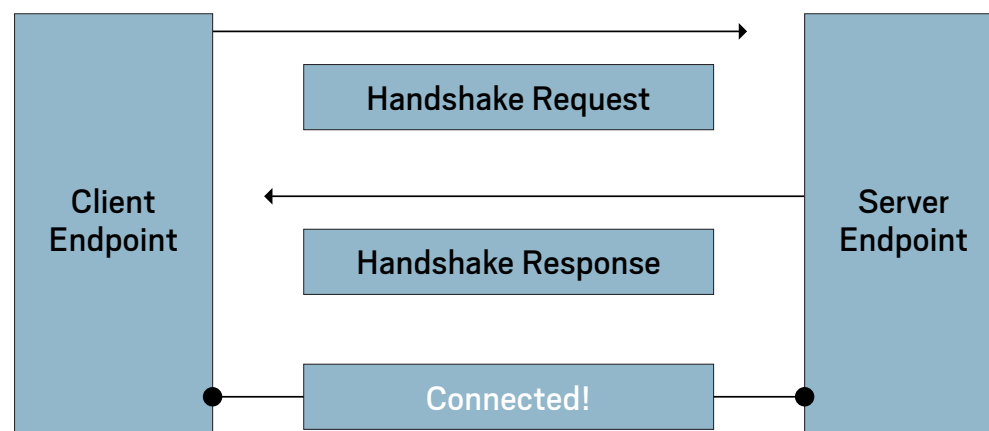


図1.WebSocketコネクションの確立

Java WebSocket APIの概要

Java WebSocket APIにはJava APIクラスおよびJavaアノテーションが用意されているので、Java EE Webコンテナ内に常駐するWebSocketエンドポイントを比較的簡単に作成することができます。一般的なやり方として、サーバー・エンドポイントのロジックを実装するJavaクラスを決め、そこに特殊なJava WebSocket APIアノテーションである@ServerEndpointというクラス・レベルのアノテーションを付加します。次に、そのJavaクラスのメソッドに、いずれかのライフサイクル関連アノテーションを付加します。たとえば、@OnMessageを付加した場合、そのメソッドは、WebSocketクライアントがそのエンドポイントにメッセージを送信するたびに呼び出されるという特殊能力を持ちます。その後、そのクラスをWARファイルのWEB-INF/classesディレクトリ内にパッケージ化します。リスト1にこの例を示します。

■ リスト1:EchoServerサンプル

```
import javax.websocket.OnMessage;
import javax.websocket.server.ServerEndpoint;
```

```
@ServerEndpoint{"/echo"}
public class EchoServer {
```

```
@OnMessage
public String echo (String incomingMessage){
    return "I got this (" +
        incomingMessage + ")" +
        " so I am sending it back !";
}
```

このWebSocketエンドポイントは、WebアプリケーションのURI領域の/echoにマッピングされます。任意のWebSocketクライアントがこのURIに対してメッセージを送信するたびに、受け取ったメッセージを基にメッセージを作成し、そのメッセージをレスポンスとして即座に返します。

Java WebSocket APIには、すべてのWebSocketライフサイクル関連イベントをインターセプトするための手段や、同期モードおよび非同期モードでメッセージを送信するための手段が備わっています。デコーダ・クラスとエンコーダ・クラスを使用すれば、WebSocketメッセージを任意のJavaクラスに、または任意の

WebSocketエンドポイントのライフサイクル関連イベントをインターセプトできるようにすることです。まず、クラス・レベルのアノテーションを確認します。

@ServerEndpoint: このAPIの中心的なアノテーションであり、多数のWebSocketエンドポイントを作成する場合に頻繁に使用することになります。このクラス・レベルのアノテーションの唯一の必須属性として、value属性があります(表1)。value属性には、WebアプリケーションのURI領域における、このエンドポイントの登録先となるURIパスを指定します。

@ClientEndpoint: [@ClientEndpoint](#) アノテーションは、クライアント・エンドポイントとなるJavaクラスに対してクラス・レベルで使用します。このクライアント・エンドポイントが、サーバー・エンドポイントへのコネクションを開始します。このアノテーションは、Java EE Webコンテナに接続するリッチ・クライアント・アプリケーション内でよく利用されます。必須属性はありません。

次に、ライフサイクル関連のアノテーションを説明します。

@OnOpen: このメソッド・レベルのアノテーションは、WebSocketエンドポイントに新たな接続がなされる場合はかならずそのエンドポイント上のこのアノテーション付きメソッドをJava EE Webコンテナが呼び出す必要があると宣言するものです。このメソッドは引数なしか、またはオプションの`Session`パラメータ、`EndpointConfig`パラメータの一方または両方、およびオプションのWebSocketパス・パラメータを引数にできます。ここで、`javax.websocket.Session`は、今オープンされたWebSocketコネクションを表すAPIオブジェクトで、`javax.websocket.EndpointConfig`はこのエンドポイントの構成情報を表すAPIオブジェクトです。WebSocketパス・パラメータについては後で説明します。

@OnMessage: このメソッド・レベルのアノテーションは、コネクション上に新しいメッセージが到達したときにかかわらず、Java EE Webコンテナがその対象のメソッドを呼び出す必要があると宣言するものです。メソッドには、特定の種類のパラメータ・リストが必要ですが、

表1:@ServerEndpointの属性

表2:クラス・レベルのアノテーションの属性

実際に構築する:WebSocket時計

Java WebSocket APIをひとつと確認し、はじめてのWebSocketアプリケーションを確認するために必要な知識は十分に得られました。以下に紹介するClockアプリケーションは単純なWebアプリケーションです。このアプリケーションを実行すると、図2のようなindex.html Webページが表示されます。

「**Start**」ボタンをクリックすると、図3のように時計が起動し、現在の時刻が表示されます。この時刻は1秒ごとに更新されます。

「**Stop**」ボタンをクリックすると、図4のように時計が停止し、再び起動するまでそのまま停止し続けます。

このアプリケーションは、単一のWebページ(index.html)と単一のJava WebSocketエンドポイント(ClockServer)により構成されます。「**Start**」が押されると、index.htmlはJavaScriptコードを使用して、ClockServerエンドポイントとのWebSocketコネクションを確立します。ClockServerエンドポイントは、更新時刻メッセージを1秒ごとにブラウザ・クライアントに送り返します。JavaScriptコードは送られたメッセージを処理し、ページ上に表示します。「**Stop**」をクリックすると、index.htmlページ内のJavaScriptコードがClockServerに対してstopメッセー

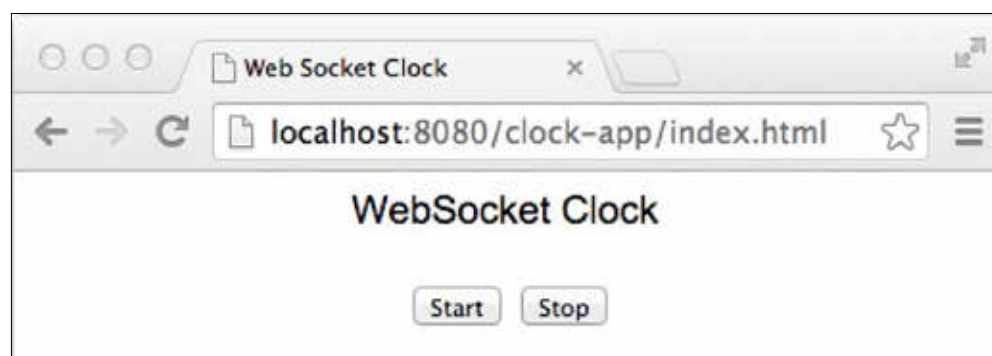


図2. WebSocket Clockオフ

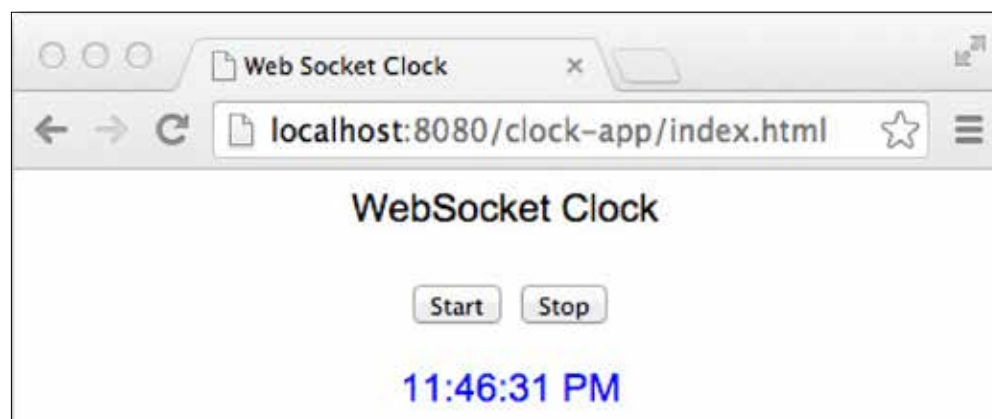


図3. WebSocket Clockオン

ジを送信します。その結果、更新時刻の送信が停止します。このアーキテクチャを図5に示します。

次にコードを見てみましょう。まずはクライアントからです。[コード・リスト全体は、本記事の[ダウンロード・ページ](#)からダウンロードできます(編集部より)]
リスト2のWebSocketクライアント・コードを以下に抜粋します。

■ リスト2: WebSocketクライアント・コード (JavaScript)

```
...
function start_clock() {
    var wsUri =
        "ws://localhost:8080/clock-app/clock";
    websocket = new WebSocket(wsUri);
    websocket.onmessage = function (evt) {
        last_time = evt.data;
        writeToScreen(
            "<span style='color:blue;'>" +
                last_time + "</span>");
    }
}
```

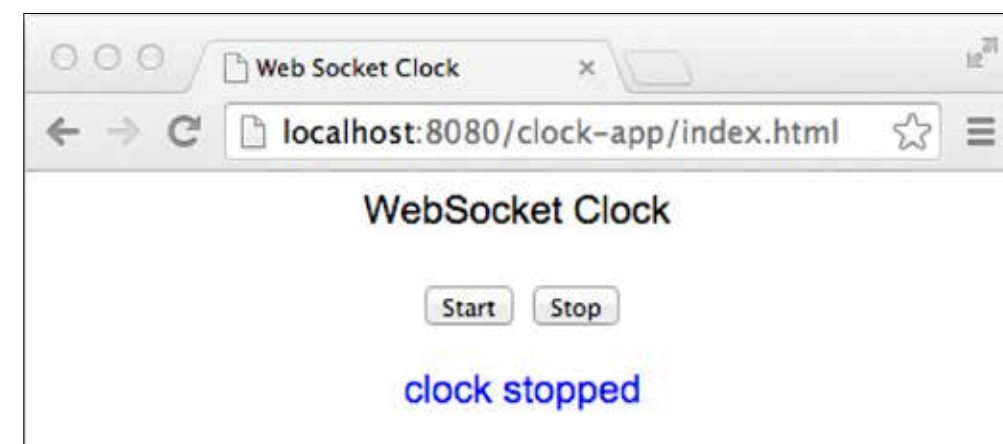


图4.WebSocket Clock停止

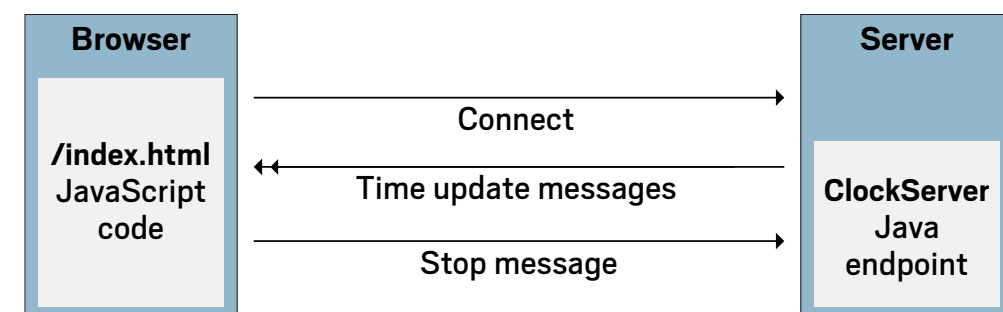


図5.Clockのアーキテクチャ

WebSocketのインスタンス数について

まとめ

</article>

LEARN MORE

- ・ [OracleのJava WebSocketチュートリアル](#)

実力を試す

賢人にとっては易しい問題が難しく、難しい問題が易しい。しかしクイズに英知はあるのか？
確かめよう。

このクイズ・コーナーでは、認定テスト 1Z0-808:Oracle Certified Associate, Java SE 8 Programmer, [Oracle Certified Java Programmer](#) から出題します。本号の問題には、用心深いコーディング技術者でも陥りがちな、前号を上回る予想外の罠が仕掛けられています。だいたいは、想像通りのコーディングで Java 言語のど真ん中を進めば、罠を回避できます。しかし、ここで確認するように、Stream(ストリーム)に関連する問題は、その固有の罠が仕掛けられており、思い通りにうまくいかないかも知れません。(解答は、設問の後の「解答」に記載)。

設問 1 次のコードについて：

```
class ProductNotFoundException extends Exception{
```

```
class SalesPerson {
    String name;
    List<String> products = new ArrayList<>();
    public List<String> getProducts() throws
        ProductNotFoundException {
        products.add("SoundCard");
        return products;
    }
}
```

および

```
class SalesApp {
    public static void main(String[] args) {
        SalesPerson sp = new SalesPerson();
    }
}
```

```
List<String> products = sp.getProducts();
System.out.println(products.get(0));
}
}
```

どのような結果が出力されますか。

- a. SoundCard
- b. 実行時に `ProductNotFoundException` がスローされる
- c. 0
- d. コンパイル・エラーが発生する

設問 2 次のコードについて：

```
String wishMsg = "Happy day!";  
wishMsg.concat(" Tom");  
String msg = (wishMsg.length() > 10) ?  
    "Too long" : "Sent";  
System.out.println(msg+" "+wishMsg);
```

どのような結果が出力されますか。

- a. Sent:Happy day!
- b. Too long:Happy day!
- c. Too long:Happy day!Tom
- d. コンパイル・エラーが発生する

設問 3 AClass.java、BClass.java、IFace.java ファイルの内容：

```
public abstract class AClass {
    public void aMethod() {
        System.out.println("Method");
    }
}
```

```
//fix this /
```

```
public abstract void bMethod();  
}  
  
public interface IFace {  
    public void cMethod();  
}  
  
public abstract class BClass  
    extends AClass implements IFace {
```

説明文として正しいものを選択してください。

- a. AClass.java ファイルのみコンパイルが通る
- b. IFace.java ファイルのみコンパイルが通る
- c. BClass.java ファイルのみコンパイルが通る
- d. すべてのファイルのコンパイルが通る

設問 4 ストリームを見てみましょう。
次のコードについて（行番号付き）：

```
10.Stream<Integer> stm =  
    Stream.of(10, 30, 20, 40);  
11.int n1 = stm.findFirst().get();  
12.boolean divByTen =  
    stm.allMatch(n -> n%10 == 0);  
13.System.out.println(n1 + ":" + divByTen);
```

どのような結果が出力されますか。

- a. 10:true
- b. 0:false
- c. コンパイル・エラーが発生する
- d. 実行時に例外がスローされる

設問 1 正解はDです。このプログラムはコンパイル・エラーになります。`getProducts()` メソッドが、`ProductNotFoundException` をスローすることを宣言しています。`getProducts()` は `main()` メソッド内で呼び出されます。そのため、`main()` メソッドが `ProductNotFoundException` を処理するか、`ProductNotFoundException` をスローすることを宣言しなければなりません。

設問 2 正解は A です。String は不変です。wishMsg は変更されず、Happy day! という値を持ち続けます。選択肢 B と C は誤りです。連結の結果は維持されません。選択肢 D も誤りです。このコードのコンパイルは通ります。

設問 3 正解は D です。選択肢 A は誤りです。AClass のコンパイルは通ります。定義のないメソッドを含むクラスは、abstract として宣言する必要があります。abstract クラスには具象メソッドを含めることができます。選択肢 B も誤りです。インタフェース内で宣言されているメソッドは、デフォルトで abstract です。選択肢 C も誤りです。BClass は abstract メソッドである cMethod() と bMethod() をオーバーライドしていないことから、このクラスは abstract として宣言されています。

設問 4 正解は D です。ストリームの実装は、ストリームが再利用されていることを検出した場合に、`java.lang.IllegalStateException` をスローすることがあります。11 行目で、`stm` ストリームを使って 1 つ目の要素を取得しており、このストリームは自動的にクローズされます。しかし、12 行目で、このクローズされた `stm` ストリームの要素にアクセスして、各要素が 10 で割り切れるかをチェックしようとしています。そのため、`java.lang.IllegalStateException` がスローされます。同じデータソースを再度探索する必要がある場合は、新しいストリームを作成しなければなりません。選択肢 A と B は誤りです。実行時に `java.lang.IllegalStateException` がスローされます。選択肢 C も誤りです。このコードのコンパイルは通ります。

