FABIAN LANGE

Byte Buddyによる実行時のコード生成

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

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。

見過ごされがちですが、Javaプラットフォームには、JVMのインタプリタやJust-In-Time（JIT）コンパイラによってプログラムが実行される前にプログラムのバイトコードを変更する機能があります。この機能は、プロファイラやオブジェクト・リレーショナル・マッピングを実行するライブラリなどの各種ツールによって使用されますが、アプリケーション開発者が使用することはほとんどありません。しかし、そこには、未開拓の可能性があります。実行時のコード生成によって、ロギングやセキュリティなどの横断的関心事の実装、サード・パーティ・ライブラリの振る舞いの変更（ときにはモックの形で）、パフォーマンス関連データの収集エージェントの開発などが簡単になるからです。

残念ながら、ごく最近まで、実行時のバイトコード生成は難しいことでした。現在のところ、バイトコード生成用のおもなライブラリには以下の3つがあります。

- ASM
- cglib
- Javassist

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、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。

以下の例はASMライブラリを使用して実装したメソッドで、単一の文字列パラメータを持つ別の静的メソッドを呼び出しています。

```
methodVisitor.visitVarInsn(Opcodes.ALOAD, 0);
methodVisitor.visitMethodInsn(
  Opcodes.INVOKESTATIC
  "com/instana/agent/Agent"
  "record"
  "(Ljava/lang/String;)V"
)
```

cglibとJavassistも同じような実装になります。すべてのライブラリでバイトコードおよびシグネチャのString表現の使用が求められるため、見てのとおり、Javaというよりもむしろアセンブリ言語のような印象です。

Byte Buddyは、この問題を異なるアプローチによって解決する新しいライブラリです。Byte Buddyの使命は、Javaの命令に関する知識を持たない開発者でも、実行時のコード生成機能を利用できるようにすることです。このライブラリは、Javaの全機能をサポートすることも目指しています。JDK組込みのプロキシ・ユーティリティで採用されているような、インタフェースの動的な実装を生成するだけのものではありません。Byte Buddy APIはすべてのバイトコード演算子を、ごく普通のJavaメソッド呼出しとして抽象化します。ただし、ASMライブラリへのバックドアも保持しています（Byte BuddyはASMライブラリの上層に実装されています）。

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えたら、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ClassLoadingStrategy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。

リスト2：

```
public static class Agent {
  public static String record(@Origin Method m) {
    System.out.println(m + " called");
  }
}

Class<?> clazz = new ByteBuddy()
  .subclass(Object.class)
  .method(ElementMatchers.isConstructor())
  .intercept(MethodDelegation
    .to(Agent.class)
    .andThen(SuperMethodCall.INSTANCE))
  // & make instance;
```

MethodDelegationは、複数のインターセプト・ターゲットがある場合に、メソッドのシグネチャにもっともよく一致するものを自動的にルックアップします。このルックアップ機能は強力でカスタマイズも可能ですが、ルックアップ部分は単純で理解しやすい状態のまま維持することを推奨します。メソッドを呼び出した後、andThen(SuperMethodCall.INSTANCE)の指定によって、元の呼出しが再開します。

ターゲット・メソッドには、いくつかのアノテーション付きパラメータを指定できます。元のメソッドの引数にアクセスするために、@Argument(position)または@AllParameterを使用できます。元のメソッド自体に関する情報を取得する場合は、@Originを使用できます。そのパラメータの型はjava.lang.reflect.Methodやjava.lang.Class、あるいは、（Java 7以降を使用している場合）java.lang.invoke.MethodHandleでもかまいません。これらの引数から、メソッドの呼出し元に関する情報を取得すればデバッグに使用できます。また、同じメソッドが複数メソッドからのインターセプト・ターゲットになる場合は、異なるコード・パスに関する情報も取得できます。

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。

モック

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。

リスト3：

```
public class Lottery {
  public boolean win() {
    return random.nextInt(100) == 0;
  }
}

Random mockRandom = new ByteBuddy()
  .subclass(Random.class)
  .method(named("nextInt"))
  .intercept(value(0))
  // & make instance;

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

Byte Buddyにはさまざまな種類のインターセプタがあり、モックやスパイを簡単に記述できます。ただし、モックの数が多い場合は、専用のモック・ライブラリに切り替えることを推奨します。実際、人気のあるモック・ライブラリMockitoのバージョン2は、現在Byte Buddyをベースとして再開発されているところです。

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()を呼び出すことだけです。

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]**</article>**

LEARN MORE

- Java 8のJVM仕様
- Stack OverflowのByte Buddyページ