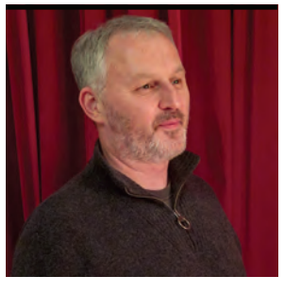

ANDREW DINN、
ANDREW HALEY

# JITコンパイラの実際の動作

## JITによるコードの変換方法

ほとんどのJava開発者は、JVMがJust-In-Time（JIT）コンパイラを利用してJavaプログラムのパフォーマンスを改善していることをご存知でしょう。しかし、JITコンパイラの実際の動作とその利点については、理解されていないことが多いようです。本記事では、JITコンパイラの動作の仕組みと、パフォーマンスが向上する理由について説明します。ここではJDKのオープンソース・リリースであるOpenJDKについて取り上げますが、本記事の内容のすべては、オラクルのリリースに対しても同様に当てはまります。

### インタプリタとコンパイラ

Javaクラスは、JVMで実行できる状態にするために、javacのコンパイルによってバイトコードへと変換する必要があります。Java Magazineの以前の記事（「JVMによるライブラリの検索、ロード、実行」）で説明しているように、バイトコードは、アーキテクチャに依存しない移植可能なファイル形式でクラスの詳細をエンコードしたものです。しかし、そのバイトコードを実行するためには、特定のプロセッサとオペレーティング・システムを対象としてビルドされたJVM（javaプログラム）が必要になります。

OpenJDKのjavaプログラムは、Javaメソッドを実行するための複数の方法を提供します。『Java仮想マシン仕様』の説明にあるとおり、Javaメソッドは最初、スタック・ベースの仮想マシンを実装するインタプリタによって実行されます。このインタプリタにより、Javaの完全な実装が提供されます。しかし、あくまでインタプリタが使用されるため、最適なパフォーマンスにはなりません。

高レベルの視点では、インタプリタは、冗長な計算を排除またはスキップするための最適化（計算順序の変更など）を多数実行することはできません。また、低レベルの視点では、インタプリタは基盤プロセッサの命令セットやキャッシュ、およびメモリ・システムを最大限に活用できません。

一方、JITコンパイラでは、これら両方のレベルでパフォーマンスの向上を見込めます。JITコンパイラは高レベルの戦略を用いて、バイトコードを、大幅に少ない計算量で同じ結果を出す同等の操作に変換します。このように変換された操作は、プロセッサ命令セットを最大限に利用して低レベルのネイティブ・マシン・コードへとエンコードされます。それと同時に、有用なデータができる限り長くプロセッサのレジスタやキャッシュ内に維持されるようにして、メモリ・システムの遅延を最小限に抑えます。また、コードの生成時に、キャッシュ・サイズやヒープ・サイズ、使用可能なプロセッサ数などの実行時定数を考慮することもできます。

インタプリタの処理にどれほど時間がかかるのかについて、フィボナッチ数列の要素を計算するプログラムの例を確認しましょう。

```
x0 = 1
x1 = 1
xn = xn-1 + xn-2 for n > 1
```

この実装は、1つのJavaクラスで表すと次のようになります。

```
class Fib
{
  public static
  void main(String[] args) {
    int i = Integer.valueOf(args[0]);
    System.out.println(fib(i));
  }
```

```
  private static int fib(int i) {
    if (i <= 0) {
      return 0;
    } else if (i == 1) {
      return 1;
    } else {
      return (fib(i - 1) +
              fib(i - 2));
    }
  }
}
```

次のようにjavaコマンドに-Xintフラグを渡すことで、インタプリタのみを用いてこのクラスを実行できます。

```
$ time java -Xint Fib 48
512559680

real  18m7.649s
user  18m8.074s
sys   0m0.088s
```

timeコマンドはLinuxおよびUNIXの固有コマンドである点に注意してください。[編集注：Microsoft Windowsのユーザーは Jem Berkes氏作成のptimeユーティリティを利用できます。このユーティリティは、本書のダウンロード・サイトからも入手できます]

## JITによって生成されるバイトコード・インタプリタ

遅いように見えますが、OpenJDKのインタプリタの実行速度が他のほとんどのインタプリタよりも優れていることは注目に値します。その理由の1つは、OpenJDKのインタプリタが、JITコンパイラによって生成されたマシン・コードを用いて実装されていることです。JITコンパイラはJVMの起動時に、バイトコードで表現できる操作ごとに1つずつ、ネイティブ・マシン・コードの小さなスニペットとしてインタプリタを生成します。このインタプリタは一度に1つのバイトコード操作に対して動作します。その際には、その操作用に生成されたコードにジャンプして、その後次のバイトコードを実行します。

それぞれのJVMスレッドは、メソッド呼出しあたり1つのフレームを格納したJVMスタックを所有しています。各フレームには、オペランド・スタックと呼ばれるスタックが含まれています（詳細については、『Java仮想マシン仕様』を参照してください）。多くのバイトコード操作で、1つ以上の値をオペランド・スタックからポップして、結果を計算し、場合によってはその結果をスタックにプッシュして戻すか、またはローカル変数の値として書き込むという処理を実行します。処理ステップでは、2つの数値の加算、入力オブジェクトのフィールドからの値フェッチ、特定のインデックスにある配列要素へのアクセスなどが必要となる場合があります。そのような操作は容易に1つから2つのネイティブ・マシン命令に変換されます。

if、whileなどのフロー制御操作は、バイトコードのポインタを更新し、場合によってはthenやelseの分岐に進むか、あるいはwhileループ条件の判定に戻ります。invokevirtualなどのメソッド操作の呼出しでは、新しいフレームを作成します。その際に、オペランド・スタックの引数を用いて新しいローカル領域に値を入力します。また、return操作はフレームを削除し、場合によっては呼出し元のオペランド・スタックにメソッドの結果をプッシュします。

起動時におけるインタプリタ・コードの生成には、C++などの高レベル言語で記述する場合と比較して重要な利点があります。それは、低レベルのアセンブラによって命令シーケンスがエンコードされることです。そのため、高レベル言語で記述されたインタプリタと比較して、ランタイム・プロセッサの能力をさらに有効に活用できます。たとえば、専用のマシン・レジスタを用いて、現在のJavaスレッドおよびフレームやバイトコード・ポインタなどのよく使用される値を高速に参照できます。また、コマンドライン・オプションを使用して異なる機能を持つインタプリタを生成することもできます。

JIT対応インタプリタに最適化を適用する機会はいくつかあります。たとえば、同時に参照されることの多いバイトコード命令の組み合わせをマージできます。thisフィールドを読み込む際にそのようなマージが行われます。マージャは、aload_0、getfieldという2つのバイトコード命令を、この順に実行する必要があります。aload_0は、ローカル領域にあるthisのオブジェクト参照を読み取ってレジスタに格納します。このオブジェクト参照は常にローカル・スロット0の位置にあります。その後、その値をスタックの最上層にプッシュします。getfieldは、スタックの最上層にあるオブジェクト参照をポップして同じレジスタ

に格納します。さらに、そのオブジェクトの開始アドレスから数オフセットの位置にある値を第2レジスタへと読み込み、読み込んだ値をスタックの最上層にプッシュします。途中で実行されるプッシュとポップは、無駄な動作です。さらに、連続する命令のそれぞれを実行するためには2つの命令バイトを2回読み取り、その命令に関連するコード・テンプレートへ2回ジャンプする必要があります。JVMはこの元の2つの命令の代わりに、疑似命令fast_agetfieldを配置することで、冗長なプッシュ/ポップ操作をなくし、1回のバイトコードの読み取りとジャンプのみで済むようにします。また、よく使用されるライブラリ呼出し（例：java.lang.ref.Reference.get()）の実装は、インタプリタが高速に実行できるように、特別にコーディングされています。

## JITコンパイル

この最適化されたインタプリタよりもさらにパフォーマンスを向上させるためには、JITコンパイラが必要です。しかし、JITコンパイラでも、すべてのメソッドがコンパイルによって最適化されたマシン・コードへと変換されるわけではありません。JVMはもっとも効果的なメソッドに集中して最適化します。

JITコンパイラは、たとえ単純なメソッドであっても、バイトコードを解析して最適化されたマシン・コードを生成するためには多くの処理を実行する必要があります。一方、大量のソースからJavaコードを徹底的にテストすれば、大半のメソッドがほとんど呼び出されず、中にはまったく呼び出されていないメソッドもあることが分かります。そのため、わざわざすべてのメソッドをコンパイルしたとしても、短縮できる時間はわずかであり、意味がありません。それならば、頻繁に呼び出されるコードに対してJITコンパイラの処理時間を集中させる方が効率的です。優れたJITコンパイラは、ホット・メソッドに対してのみ実行時間の短縮を試みます。この理由から、OpenJDK JVMにはHotSpotという名前が付けられました。よく言われることですが、実行時間の90%がコードの10%の実行に費やされています。したがって、その10%に最適化処理を集中させることが理想です。

HotSpot VMの実際の動作は次のコマンドで確認できます。

```
$ time java -XX:+PrintCompilation Fib 48
  66    1  3 java.lang.String::indexOf (70 bytes)
  67    3  3 java.lang.String::hashCode (55 bytes)
  ...
  73   10  3 java.lang.String::equals (81 bytes)
  73   11  3 Fib::fib (29 bytes)
  74   12  4 Fib::fib (29 bytes)
  75   11  3 Fib::fib (29 bytes)   made not entrant
512559680

real   0m26.866s
user   0m26.861s
sys    0m0.021s
```

見てのとおり、このコンパイル済みのコードは、インタプリタの50倍の速さで実行されます（このログの3列目は最適化レベルを表します。最適化レベルについては後で説明します）。

プログラムでString.indexOfメソッドを呼び出している箇所がないにもかかわらず、このメソッドが最初のコンパイル対象メソッドとなっている点に注意してください。このメソッドがコンパイルされるのは、Javaの起動時にランタイム環境をセットアップするためのコードが実行され、そのコードの大部分が文字列処理で構成されているためです。したがって、最初のホット・メソッドはStringクラス内にあります。

また、fibがコンパイルされた後に、さらに高い最適化レベルで再コンパイルされています。元のコンパイル済みコードは、新しいバージョンのコードが生成された後に破棄されます。このことから、メソッドのコンパイル方法は複数あるように推察されます。実際、OpenJDKには複数のJITコンパイラが存在します。

## JITコンパイラの選択

コンパイルによるパフォーマンスの向上は、バイトコードをインタプリタで処理する代わりにマシン・コードを実行することで得られます。しかし、それ自体はかならずしも大きな差につながりません。最大の効果は、マシン・コードの生成時にコンパイラが実行する高レベルおよび低レベルの最適化によって得られます。

OpenJDKには2種類のJITコンパイラが含まれており、これらは一般にクライアント・コンパイラおよびサーバー・コンパイラと呼ばれます。当初はjavaコマンドラインで、-clientオプションまたは-serverオプションを用いて、使用するコンパイラを指定する必要がありました。しかし、JDKの最近のリリースでは、デフォルトの構成で両方のコンパイラが

使用されます。いわゆる階層型コンパイル・モードです。階層型コンパイルの実際の動作は、先ほどのトレースで確認できます。fibメソッドはまずレベル3でコンパイルされています。レベル3ではクライアント・コンパイラを使用しますが、さらにメソッド呼出し回数と分岐で選択されたパスをカウントするコードも含まれます。その後、fibがレベル4で再コンパイルされています。レベル4はサーバー・コンパイラを使用します。

JITクライアント・コンパイラは、通常は短い時間のみ（おそらく数秒から数分）実行されるデスクトップ・アプリケーションに対応するように設計されています。これに対して、JITサーバー・コンパイラは、数時間、数日間、あるいは数か月間実行されるアプリケーションを想定して作られています。いずれのコンパイラも最適化を行いますが、そのトレードオフは大きく異なります。基本的に、JITクライアント・コンパイラが行う最適化は、JITサーバー・コンパイラより少なくなります。そのためコンパイル済みコードの実行時間は長くなりますが、その生成時間は短くなります。

このトレードオフは簡単に確認できます。たとえば、OpenJDKリポジトリのOpenJDKソース・コードにおけるJavaライブラリ・ソースは、サブディレクトリのjdk/src/share/classes内にあります。本記事では例として、javadocを次のようにクライアント・コンパイラのみで実行しました。その際にjdkfilesに、src/share/classes以下にあるすべてのJavaソース・ファイルが記載された入力ファイルを指定しました。

```
$ time javadoc \
-quiet -J-XX:TieredStopAtLevel=2 @jdkfiles
real   4m41.775s
user   5m31.390s
sys    0m1.862s
```

コマンドライン・オプションの-Jを用いて、基盤のJVMに引数を渡しています。-XX:TieredStopAtLevel=2オプションは、JVMに対してレベル1（インタプリタ済み）またはレベル2（クライアント・コンパイラ、プロファイリングなし）のみを実行するように指定するものです。

サーバー・コンパイラのみを用いてJVMを実行する場合は、-XX:-TieredCompilationオプションを渡します。このオプションでは、階層型コンパイルがオフになります。その場合、次のような実行時間が記録されます。

```
$ time javadoc \
-quiet -J-XX-TieredCompilation @jdkfiles
real   3m30.083s
user   4m50.410s
sys    0m1.880s
```

サーバー・コンパイラに切り替えたことで、経過時間が4分40秒から3分30秒まで短縮されました（25%の短縮）。ユーザー時間（user）の変化は注目に値します。ユーザー時間は、すべてのコアでのCPU時間です。1回目のクライアント・コンパイラによる実行では、CPU時間がreal（経過）時間よりも50秒多く使用されました。2回目のサーバー・コンパイラによる実行では、CPU時間が1分20秒増えています（60%増）。つまり、60%のJITコンパイラ時間の増加によって、総実行時間の25%を削減したことになります。

JITコンパイルは通常、CPUがアイドル状態となるバックグラウンド・スレッドで実行されます。そのため、コンパイルによりCPU時間が消費され、アプリケーションの実行が遅延することはありません。JITコンパイラの有効性は、生成するコードの実行速度とそのコードの生成時間の2つで決まります。生成時間が重要になるのは、以下の条件が満たされるまで実行対象がコンパイル済みコードに切り替わらないためです。

- メソッドが呼び出された回数が、コンパイルのキューに入るほど十分な回数に達する
- コンパイラによるそのメソッドのデキューと、コンパイル済みコードの生成が完了する

サーバー・コンパイラは実行速度に優れたコードを生成しますが、そのコードの実行時に事前の遅れを取り戻す必要が生じます。サーバー・コンパイルされた各メソッドは、クライアント・コンパイルされた同等のメソッドよりも遅く生成されるため、コンパイルの予約時にホットな状態であったメソッドが、コンパイル済みのコードの生成までにコールド状態になる場合があるということです。サーバー・コンパイラの処理が完了する前にプログラムが終了することもあります。

以上のことから、実行時間の長いサーバー・アプリケーションに対してJITコンパイラを選択する必要がある場合は、ほぼ確実にクライアント・コンパイラよりもサーバー・コンパイラを選択すべきです。数週間実行されるコードが可能な限り最適化されると保証されるならば、サーバー起動時に実行速度が上がるまで少し時間がかかっても、その犠牲を払

う価値はあります。

しかし、OpenJDKでは、ほぼすべてのアプリケーションで、その両者の長所を活かしたさらに良いオプションが提供されています。それが、前述のデフォルトの階層型コンパイル構成です。階層型の意味は次のとおりです。まず、ホット・メソッドを、JITクライアント・コンパイラを用いてコンパイルします。これらのクライアント・コンパイルされたメソッドでは実行頻度が計測され、ホット状態が継続する場合は、JITサーバー・コンパイラを用いて再コンパイルされます。階層型コンパイルにより、起動時に良好な実行速度を確保しながら、ホット状態が継続するコードが最大限まで最適化されます。また、プログラムの起動時に大量に呼び出されるが、すぐにコールド状態になるようなコードを過度に最適化する無駄な時間も発生しません。

上記のjavadocの例を階層型コンパイルによって実行した場合、サーバー・コンパイラの数値と比較して大きな違いはありません。

```
$ time javadoc \
-quiet -J-XX+TieredCompilation @jdkfiles
real   3m31.275s
user   5m23.726s
sys    0m2.093s
```

コンパイル時にかかるユーザー時間がかなり増えますが（1分50秒増）、経過時間はそれほど改善していません。これは、クライアント・コンパイルされるjavadocアプリケーションのほぼすべてのコードが、最終的にサーバー・コンパイルされるほど高い頻度で実行されるためです。つまり、早期にJIT対応コードを生成する利点が、実行速度の遅いコードをより長く実行するオーバーヘッドによって相殺されてしまっています。

実際のサーバー・アプリケーションでは、階層型コンパイルのコストが大きくなるケースはほとんどなく、サーバー・コンパイルのみの場合よりも高い効果が得られることが多くなっています。これは、アプリケーションの起動時間が大幅に改善するためです。階層型コンパイルは対話型アプリケーションにも有効です。その理由も、すぐに起動できること、そしてサーバー・コンパイラの処理が進むにつれて実行速度が改善されることにあります。NetBeansなどのアプリケーションを実行する際に、その効果に気付くことでしょう。

## JITだけが実現できる最適化

実行時にメソッドをコンパイルするオーバーヘッドが加わるのに、JITコンパイラを使用するメリットがあるのか疑問に思うかもしれません。C++などのネイティブ言語と同じように、最適なパフォーマンスを得るために事前にコード全体をコンパイルするだけで良いのではないでしょうか（この方式は、事前コンパイルまたはAOTと呼ばれています）。この方式は、コンパイル時にコード全体にアクセスできる場合は可能です。しかし、言うまでもなくJavaは動的言語であり、クラスパスから、あるいはネットワーク経由でもコードをロードできます。JITコンパイラが存在しなければ、動的にロードされたコードをインタプリタによって処理する必要があり、とても満足できる結果にはなりません。

しかし、JITコンパイラを使用すべき理由はそれだけではありません。実行時コンパイルのもっとも重要な利点は、AOTコンパイラでは不可能な最適化を実行できることです。あまりよく知られていませんが、OpenJDK JITコンパイラは、あらゆる種類のプログラムのコードを、たとえばC++コンパイラで同等のプログラムのコードを生成するよりも速く生成できます。

## まとめ

自分のアプリケーション・コードを用いてJVMの処理内容を確認したい場合は、次のフラグを有効にして実行を試してください。

```
java -XX:+PrintCompilation \
   -XX:+UnlockDiagnosticVMOptions \
   -XX:+PrintInlining MyMainClass arg1 … argN
```

これらのオプションによって、メソッドのコンパイル時にインライン化の決定がなされている様子を観察できます。さまざまなデータセットを用いて十分に長い時間をかけて検証すれば、アプリケーションのフェーズの変化によってコードを前もってコールド・パスに置くときに、メソッドの脱最適化と再コンパイルの様子を確認できるでしょう。実行開始時点からの呼出し回数が蓄積されて、コンパイルに送られるインライン・ツリーが巨大化する様子も確認できるかもしれません。

さらに大胆に調査したい場合は、OpenJDKのソース・コードをダウンロードして逆アセンブラ・ライブラリをビルドしてください（hotspotソース・ツリーのhsdisサブディレクトリを探します）。-XX:+PrintAssemblyオプションを指定して、生成されたネイティブ・マシン・コードを出力できます。OpenJDKのデバッグ版をビルドすることもできます。これにより、その他のさまざまなPrintオプションやTraceオプションを用いて、JITコンパイラの処理内容に関する多くの詳細情報を得ることができます。

もっとも成功しているテクノロジーは目に見えないということがよくあります。あまりにも有効に機能していて、その存在を知る由もないのです。Java HotSpot VMもあらゆる場面でそう言えます。インタプリタ、JITコンパイラ、ガベージ・コレクタ、ランタイム・システムがスムーズかつ迅速に連携しているため、開発者がその存在に気付くことはありません。しかし、ときには実際の動きを知ることで、その利点がよりはっきりと分かるものです。**</article>**

---

**Andrew Dinn**：Red HatのOpenJDKチーム・メンバー。JBossプロジェクトBytemanのリーダーでもある。

**Andrew Haley**：Red HatのJavaチームの技術リーダー。職業上のプログラミング経験は30年を超え、Javaもその登場時から利用している。

## learn more

『Java HotSpot VMにおけるJITコンパイルの概要』



# ブルガリアのJAVAユーザー・グループ

ブルガリアは今、新興企業やアウトソーシングのための人気の拠点になっています。Bulgarian Java User Group（JUG）は2007年9月に設立され、その際にJava関連の問題について議論するメーリング・リストが始まりました。2013年に5人のメンバーがリーダー委員会を立ち上げ、月に1～2回、定例会議とセッションを開催するようになりました。同グループはAdopt OpenJDKの活動に参加し、その後まもなくJava Community Processプログラムにも関与するようになりました。同グループは会員としてOpenJDKにいくつかの変更を反映し、ハッカソンの開催も行っています。

2015年、同グループはjPrimeというコミュニティ・カンファレンスの主催を始めました。このカンファレンスには、400人の参加者と20社のスポンサー企業が集いました。このカンファレンスは、同地域の大規模Javaイベントの1つとして位置付けられています。

同グループは、定期開催される無料の1日ミニカンファレンス「jProfessionals」も立ち上げました。1回目のjProfessionalsミーティングは2015年11月に開催され、Jenkins CIの生みの親である川口耕介氏がおもなスピーカーを務めました。

Bulgarian JUGは、国内外の講演者を招く定期イベントを開催しています。たとえば、Java ChampionのDavid Blevins氏は、TomEEアプリケーション・サーバーに関する講演を行いました。月1回のイベントにはハンズオン・ラボも含まれます。

集まる機会のない夏の間、メンバーは週末のコード・リトリートを始めました。2015年のテーマは、Spring Boot用のJBoss Forgeアドオンの開発でした。

2016年は、さらに多くのイベントを開催する予定です。