リアルタイム Java: 第 2 回: コンパイル技術を比較する

Java でのネイティブ・コードの静的コンパイルと動的コンパイルにおける課題

Mark Stoodley
Advisory Software Developer
IBM Toronto Lab

Kenneth Ma
Staff Software Developer
IBM Toronto Lab

Marius Lut
Staff Software Developer
IBM Toronto Lab

2007年 4月 17日

リアルタイム Java™ についての 5 回連載の第 2 回では、Java 言語でのネイティブ・コードのコンパイルに伴う問題について取り上げます。動的コンパイル (Just-in-time) も、静的コンパイル (Ahead-of-time) も、単独では Java アプリケーションのすべての要件に適合できません。そこで、著者がこの 2つのコンパイル技術をさまざまな実行環境で比較し、それぞれが互いの長所をどのように生かすかを説明します。

このシリーズの他の記事を見る

Java アプリケーションのパフォーマンスは、時として開発コミュニティーでの熱い討論の火種となってきました。Java 言語が設計された目的は、アプリケーションの移植性という重要な目標をサポートするように解釈されることだったため、初期の Java ランタイムが提供していたパフォーマンス・レベルは、C やC++などのコンパイルされた言語で実現可能なレベルに比べて遥かに劣っていました。ですが、このような高いパフォーマンス・レベルを持つ言語でも、生成されるコードは限られた数のシステムでしか実行できません。そこで、Java ランタイムのベンダーたちはこの 10 年の間に高性能の動的コンパイラーを開発しました。それが、JIT (Just-In-Time) として知られるコンパイラーです。JIT コンパイラーはプログラムの実行中に、最も頻繁に実行されるメソッドを選択的にネイティブ・コードにコンパイルします。C または C++ で作成されたプログラムのようにネイティブ・コードへのコンパイルをプログラムの実行前に行うのではなく、プログラムの実行時まで遅らせることによって、移植性の要件を維持するというわけです。一部の JIT コンパイラーはインタープリターも使わずにすべてのコードをコンパイルしますが、そのようなコンパ

イラーにしても、プログラムの実行中に動作することによって Java アプリケーションの移植性を確保します。

動的コンパイル技術が非常に進歩したおかげで、最近の JIT コンパイラーが実現するアプリケーション・パフォーマンスは、C や C++ で作成して静的にコンパイルされた多種多様なアプリケーションのパフォーマンスに匹敵するようになっています。それでも多くのソフトウェア開発者は、自らの経験や事例証拠から、動的コンパイルによってプログラムの動作が大幅に妨げられる可能性があると考えています。動的コンパイルでは、コンパイラーがアプリケーションと CPU を共有しなければならないためです。一部の開発者は、こうしたパフォーマンスの問題は静的コンパイルが解決するはずだという固い信念を持って、Java コードの静的コンパイルを頑なに提唱しています。アプリケーションや実行環境によっては、静的コンパイルが Java パフォーマンスに極めて有効であったり、実際的な唯一のオプションであることは真実です。ただし、Java アプリケーションを静的にコンパイルする場合、十分なパフォーマンスを達成するためには多くの難題が関わってきます。平均的な Java 開発者は、動的 JIT コンパイラーのメリットを完全に理解していないのかもしれません。

この記事では、Java 言語の静的コンパイルと動的コンパイルのそれぞれに伴う問題を、リアルタイム (RT) システムに対する影響に焦点を絞って検討します。まず Java 言語インタープリターがどのように動作するかを簡単に説明した後、最近の JIT コンパイラーが実行するネイティブ・コードのコンパイルの長所と欠点を取り上げます。続いて紹介するのは、IBM® が WebSphere® Real Time でリリースした AOT コンパイル技術です。この技術についても長所と欠点を説明した後、この 2 つのコンパイル・ストラテジーを比較対照して、AOT コンパイルのほうがおそらく有効な方法となるアプリケーションの分野と実行環境を指摘します。重要なのは、この 2 つのコンパイル技術は互いに排他的ではないという点です。それぞれの技術が持つ長所と欠点によって、その技術が最も効果を発揮するアプリケーションの種類が決まります。

Java プログラムの実行方法

Java プログラムは初めに、Java SDK の javac プログラムによってクラス・ファイルと呼ばれるプラットフォームに依存しない固有のフォーマットにコンパイルされます。このフォーマットは、Java 言語で作成されたプログラムを実行するのに必要なすべての情報を定義することから、Java プラットフォームと見なすことができます。この Java プラットフォームを特定のネイティブ・プラットフォームに対して実装するのが仮想マシンで、Java ランタイム環境 (JRE) としても知られる Java プログラムの実行エンジンに組み込まれています。例えば、Linux® ベースの Intel x86 プラットフォーム、Sun Solaris プラットフォーム、そして AIX® オペレーティング・システム上で稼動する IBM System p™ プラットフォームのそれぞれに JRE があります。これらの JRE 実装が、Java プラットフォーム向けに作成されたプログラムを正しく実行するために必要なすべてのネイティブ・サポートを実装します。

> 実際には、オペランド・スタックのサイズには実用上の制限がありますが、プログラマーがその制限を超えるメソッドを作成することはめったにありません。JVM には、そのようなメソッドをどうにか作成してしまったプログラマーに通知をしてくれる安全性チェックの機能があります。

Java プラットフォームのプログラム表現で重要な部分は、Java クラスの各メソッドが実行する演算を記述する一連のバイトコードです。バイトコードは、理論上は無限大のオペランド・スタッ

クを使って演算を記述します。このスタック・ベースのプログラム表現によってプラットフォームの中立性が実現しますが、それが可能なのは、どのネイティブ・プラットフォームの CPU の場合でもこのプログラム表現は、使用できるレジスター数には依存しないためです。オペランド・スタックで実行可能な演算はすべて、ネイティブ・プロセッサーの命令セットとは別に定義されますが、これらのバイトコードの実行を定義するのが Java 仮想マシン (JVM) 仕様 (「参考文献」を参照) です。どのネイティブ・プラットフォームの JRE であっても、Java プログラムを実行するときには、この JVM 仕様が設定するルールに従わなければなりません。

スタックをベースとしたネイティブ・プラットフォームはほとんどないため (Intel X87 浮動小数点コプロセッサーは注目に値する例外です)、ほとんどのネイティブ・プラットフォームは Java バイトコードを直接実行することができません。この問題に対処するため、初期の JRE ではバイトコードを解釈して Java プログラムを実行していました。つまり、JVM は以下の操作を繰り返すループで動作します。

1. 実行する次のバイトコードをフェッチする。
2. バイトコードをデコードする。
3. オペランド・スタックから必要なオペランドをフェッチする。
4. JVM 仕様に従って演算を実行する。
5. 結果をスタックに書き込む。

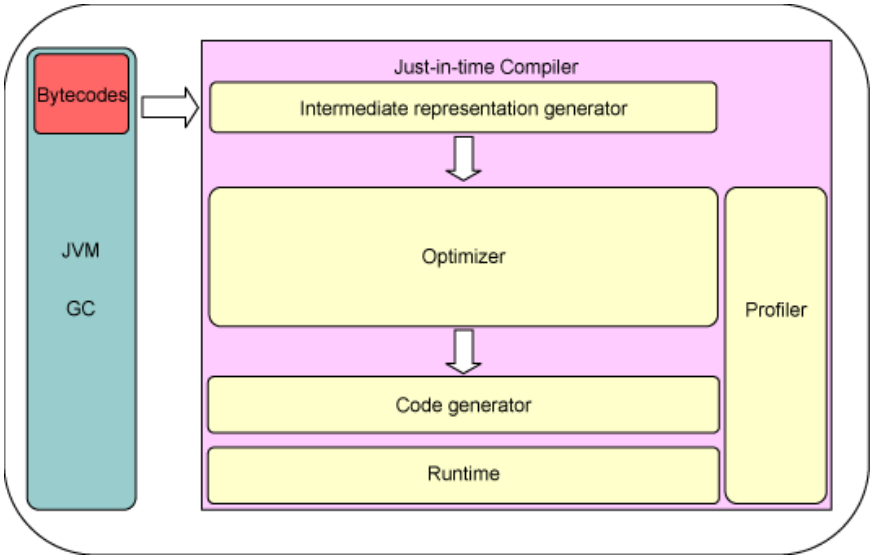
この方法の長所は単純なところで、JRE 開発者が作成しなければならないのは、それぞれのタイプのバイトコードを処理するコードだけです。しかも、演算を記述するのに使用できるバイトコードは 255 足らずなので、実装コストも抑えられます。ですが、当然パフォーマンスは欠点となります。そのため、他に多くの長所があるにも関わらず、初期のJava プラットフォームは多くの人々に非難されていました。

C や C++ とのパフォーマンスのギャップを埋めるということは、すなわち、Java プラットフォームで移植性が犠牲にならないようなネイティブ・コードのコンパイルを開発するということを意味していました。

Java コードのコンパイル

Java プログラミングの「一度作成すればどこででも実行できる」というスローガンはすべてのケースに当てはまるわけではないという事例証拠はありますが、Java プログラミングは実際に、多種多様なアプリケーションで有効です。その一方、ネイティブ・コンパイルはまさにその性質上、プラットフォームに固有なものです。それでは、Java プラットフォームがプラットフォームの中立性を犠牲にせずにネイティブ・コンパイルのパフォーマンスを達成するにはどうしたらいいのでしょう。その答は、この 10 年間そうであったように、JIT コンパイラーという形式で動的コンパイルを行うことです (図 1 を参照)。

図 1. JIT コンパイラー



JIT コンパイラーではパフォーマンスを向上させるため、実行中の Java プログラムのメソッドが 1 度に 1 つずつネイティブ・プロセッサーの命令にコンパイルされます。このプロセスにはメソッドの内部表現の生成が伴います。この内部表現はバイトコードとは異なりますが、ターゲット・プロセッサーのネイティブ命令よりも高級な表現になっています (IBM JIT コンパイラーは、一連の式ツリーを使用してメソッドの演算を表現します)。JIT コンパイラーは一連の最適化を行って品質と効率性を向上させ、最後にコード生成ステップを実行して最適化された内部表現をターゲット・プロセッサーに固有の命令に変換します。生成されたコードはランタイム環境を利用して、型キャストが正しいことを確認したり、コード自体で直接実行するには実用的でない特定の型のオブジェクトを割り当てたりするなどのアクティビティーを実行します。JIT コンパイラーはアプリケーション・スレッドとは別のコンパイル・スレッドで動作するので、コンパイルが発生までアプリケーションが待機する必要はありません。

図 1 にはプロファイル作成用フレームワークも記載されています。このフレームワークは、定期的にスレッドをサンプリングして実行中のプログラムの動作を監視し、頻繁に実行されるメソッドを検出します。また、特殊化されたプロファイルを持つメソッドが、今回プログラムを実行している間に変更される可能性のない動的な値を保管するための機能も提供します。

この JIT コンパイル手順はプログラムの実行中に行われるため、プラットフォームの中立性は保たれます。つまり、中立の Java プラットフォーム・コードは配布形式をそのまま維持するということです。C や C++ などの言語には、このようなメリットはありません。これらの言語のネイティブ・コンパイルのステップはプログラムの実行前に行われるため、ネイティブ・コードはつまり、(ネイティブ・プラットフォーム) 実行環境に配布されたものということになります。

課題

JIT コンパイルを使用すればプラットフォームの中立性は保たれますが、それには犠牲も伴います。コンパイルはプログラムの実行と同時に行われるため、コードのコンパイルにかかる時間がプログラムの実行時間に加算されるのです。重要な C または C++ プログラムをビルドしたことがある人なら誰でもわかるように、コンパイルは通常、短時間のプロセスにはなりません。

この欠点に対処するために、最近の JIT コンパイラーでは 2 つの方法のいずれかを採用しています (場合によっては両方)。そのうちの 1 つは、すべてのコードをコンパイルする一方、時間のかかる分析や変換は一切行わないという方法です。こうするとコードを短時間で生成できるため、コンパイルによるオーバーヘッドはあるとしても、ネイティブ・コードを繰り返し実行することによって実現されるパフォーマンス向上で簡単に相殺できます。もう 1 つの方法は、コンパイルのリソースを、ホット・メソッドとも呼ばれる頻繁に実行される少数のメソッド専用にすることです。この方法ではコンパイルのオーバーヘッドが低く抑えられるため、ホット・コードを繰り返し実行することによってもたらされるパフォーマンス上のメリットでさらに簡単に相殺できます。大抵のアプリケーションは少数のホット・メソッドの実行だけに時間を費やすので、コンパイルのパフォーマンス・コストを最小限に抑えるには、後者の方法が効果的です。

動的コンパイラーが根本的に複雑なのは、あるメソッドの実行がプログラム全体のパフォーマンスに影響する度合いをどの程度把握する必要があるのかということと、そのコードをコンパイルすることによってその後のプログラムの実行にどれだけのメリットを期待できるのかということ、とのバランスを取ることが難しいところにあります。極端な例として、プログラムを実行した後、その特定の実行において最も寄与したメソッドを完全に把握しているにも関わらず、プログラムはすでに完了しているため、それらのメソッドをコンパイルしても何の価値もないという場合があります。それとは逆に、プログラムを開始する前には、どのメソッドが重要であるかまったくわからないにも関わらず、どのメソッドにも最大のメリットが潜在するとしている場合もあります。大抵の動的コンパイラーは、重要なメソッドについての知識の必要性と、その知識から期待できるメリットのバランスを取ることにより、この極端な 2 つの例の中間あたりで動作します。

Java 言語ではクラスを動的にロードしなければならないという事実は、Java コンパイラーの設計を大きく左右します。まだロードされていない別のクラスを参照するコードがコンパイルされた場合を考えてみてください。例えば、まだロードされていないクラスの静的フィールドの値を読み取るメソッドです。Java 言語では、クラスの参照を最初に実行したときに、そのクラスを現行の JVM にロードして解決するという要件があります。つまり参照は最初の実行まで解決されないため、その静的フィールドのロード元となるアドレスがないということになります。コンパイラーはこのような事態に対処するため、クラスがまだロードされていない場合には、そのクラスをロードして解決するコードを生成します。こうしてクラスが解決されるとアドレスが既知になるため、元のコードの位置がスレッド・セーフな方法で変更されて、静的フィールドのアドレスに直接アクセスできるようになります。

IBM JIT コンパイラーでは、安全でしかも効率的なコード・パッチの手法を使用できるように、相当な努力が払われました。その結果、クラスが解決された後に実行されるネイティブ・コードは、フィールドが解決されたのがコンパイル時であるかのように、単にフィールドの値をロードするようになっています。これに代わる手段は、フィールドの位置を検出して値をロードする前に、フィールドが解決されているかどうかを常にチェックするコードを生成することですが、未解決のフィールドの中でも、解決されると頻繁にアクセスされるようになるフィールドについては、この単純な方法が大きなパフォーマンス上の問題になる可能性があります。

動的コンパイルのメリット

Java プログラムを動的にコンパイルするという方法には、静的にコンパイルされた言語で通常生成が可能なコードよりも優れたコードを生成できるという重要なメリットがあります。最近の JIT

コンパイラーは多くの場合、生成されたコードにフックを挿入してプログラムの動作方法に関する情報を収集するため、再コンパイルするメソッドが選択された場合、その動的な動作はさらに最適化されます。

この方法をわかりやすく説明する例は、特定の arraycopy 演算の長さを収集する場合です。arraycopy 演算を実行するたびに、その長さがほとんど一定であることが判明した場合、その最も頻繁に使用される arraycopy の長さに合わせて特殊化されたコードを生成するか、あるいはその長さに合わせて調整した一連のコードを呼び出すことができます。メモリー・システムの特質と命令セットの設計が原因となって、最も汎用的なメモリー・コピーのルーチンが、特定の長さをコピーするために作成されたコードほど高速になることはめったにありません。例えば、アラインされた 8 バイトのデータをコピーするには 1 命令または 2 命令で済みますが、それに対して、どんなアライメントのどんなバイト数でも処理できる汎用のコピー・ループを使った場合は、同じ 8 バイトをコピーするのにおそらく 10 の命令が必要となります。このように特定の 1 つの長さに対して特殊化したコードを生成したとしても、生成されるコードは別の長さに対しても正しいコピー操作を行うはずです。コードは共通して観測された長さに対する処理時間を短縮化するためだけに生成されるので、パフォーマンスは平均して改善されるというわけです。このタイプの最適化は大抵の場合、静的にコンパイルされる言語には実用的ではありません。なぜなら、考えられるすべての実行で長さが一定であることは、特定の 1 回のプログラムの実行で長さが一定であることに比べると可能性が低いからです。

この類の最適化としては、クラス階層ベースの最適化も重要な例です。例えば仮想メソッド呼び出しでは、呼び出しの受信側オブジェクトのクラスを調べて、その受信側オブジェクトに対する仮想メソッドを実際に実装するターゲットを見つけなければなりません。調査によると、ほとんどの仮想呼び出しのターゲットは、すべての受信側オブジェクトに対して 1 つのみであることがわかっているため、JIT コンパイラーでは仮想呼び出しよりも効率的な直接呼び出しのためのコードを生成します。つまり、JIT コンパイラーは時間のかかる仮想呼び出しを実行するのではなく、コードをコンパイルする際にクラス階層の状態を分析することによって、仮想呼び出しの単一のターゲット・メソッドを検出し、そのターゲット・メソッドを直接呼び出すコードを生成するというわけです。当然のことながら、クラス階層が変更されて 2 番目のターゲットのメソッドが可能になった場合、JIT コンパイラーは最初に生成されたコードを修正して仮想呼び出しが実行されるようにできますが、実際には修正が必要になることはめったにありません。このような修正が必要な場合も、この最適化を静的に行うのはとても厄介です。

動的コンパイラーでは通常、少数のホット・メソッドのコンパイルのみに絞ってコンパイル作業を行うので、より集約的な分析を実行して一層有効なコードを生成すれば、コンパイルによる見返りはさらに大きくなります。実際、最近の JIT コンパイラーのほとんどが、非常に頻繁に使用されることがわかったメソッドの再コンパイルもサポートします。これらのホット・メソッドを (コンパイル時間にはそれほど重点を置いていない) 静的コンパイラーに通常見られるような極めて集約的な最適化を使用して分析および変換をすれば、コードの品質とパフォーマンスをさらに改善できます。

以上の改善点や同様の改善点を組み合わせた結果、多くの Java アプリケーションで、動的コンパイルが C や C++ 言語での場合の静的なネイティブ・コンパイルとのパフォーマンス・ギャップを埋め、場合によってはそれ以上に優れたパフォーマンスをもたらすという効果が出ています。

欠点

動的コンパイルにはそれでもなお、場合によってはこれを理想的なソリューションとは言えなくしている欠点があります。例えば、動的コンパイルは頻繁に実行されるメソッドを識別する時間、そしてこれらのメソッドをコンパイルする時間が必要なためアプリケーションはウォームアップ状態になりますが、この期間、パフォーマンスはまだそのピークに達しません。このウォームアップ期間が、さまざまな理由によりパフォーマンス問題となり得ます。まず第一に、多数の初期コンパイルがアプリケーションの開始時間に直接影響します。これらの初期コンパイルがアプリケーションを開始状態に至らせるまでの時間を遅らせるだけでなく (Web サーバーは、初期化段階を経てから有効な作業を行えるような状態になることを考えてください)、このウォームアップ段階で頻繁に実行されるメソッドも、アプリケーションの安定状態のパフォーマンスに大きく貢献するとは考えられません。JIT コンパイルを実行すると、開始時間を遅らせることになり、長期的なアプリケーションのパフォーマンスも大幅に改善されるというわけではなく、ことさら無駄になります。最近のすべての JVM はシステム調整を行ってこの開始時間の不利を軽減するものの、この問題をあらゆるケースで完全に解消することはできません。

第二の問題は、一部のアプリケーションでは動的コンパイルに関連する遅延をただ単に許容できないという点です。GUI などの対話型アプリケーションがその一例で、この場合、コンパイル・アクティビティーがアプリケーションのパフォーマンスを大幅に改善することはないため、ユーザーのエクスペリエンスに悪影響を及ぼす可能性があります。

最後に、厳しいタスクの期限を持つリアルタイム環境で機能するように設計されたアプリケーションは、コンパイルの不確定なパフォーマンスの影響や動的コンパイラー自体のメモリー・オーバーヘッドを許容できないという問題もあります。

このように、JIT コンパイル技術は静的な言語パフォーマンスのレベルあるいはそれ以上のレベルのパフォーマンスを実現するまでに開発されてはいますが、動的コンパイルが単純にふさわしいないというアプリケーションも一部にあります。このようなシナリオでは、Java コードの AOT (Ahead-Of-Time) コンパイルが適切なソリューションになる可能性があります。

AOT Java コンパイル

原則的に、Java 言語のネイティブ·コンパイルは、C++ や Fortran などの従来の言語用に開発されたコンパイル技術による、単純なアプリケーションであるべきなのですが、Java 言語自体の動的な特性がさらなる複雑さを招き、静的にコンパイルされた Java プログラムのコード品質に影響を及ぼすことがあります。ただし、基本的な考え方は変わりません。つまり、プログラムを実行する前に Java メソッドのネイティブ・コードを生成し、プログラムの実行を開始したらネイティブ・コードを直接使用できるようにするという考えです。そこには、JIT コンパイラーのランタイム・パフォーマンスやメモリーを犠牲にしない、あるいはインタープリターの初期パフォーマンスのオーバーヘッドをなくすという目標があります。

課題

動的なクラス・ロードは動的 JIT コンパイラーにとっての課題ですが、AOTコンパイルでは一段と大きな問題になります。実行中のコードがクラスを参照するまで、そのクラスをロードできないからです。AOTコンパイルはプログラムの実行前に行われるため、コンパイラーはどのクラスがロードされたかを推測できません。つまりコンパイラーには、静的フィールドのアドレスもオブ

ジェクトのインスタンス・フィールドのオフセットもわからず、さらには直接の (つまり仮想以外の) 呼び出しであっても呼び出しの実際のターゲットが不明だということです。これらの情報のいずれかを推測した場合、コードの実行時にその推測が外れていたとすると、コードは不正となり、Java への準拠が台無しにされることになります。

コードはどんな環境でも実行できるため、クラス・ファイルはコードがコンパイルされた時とは同じではない場合があります。一例として、ある JVM インスタンスがディスクの特定の場所からクラスをロードし、その後のインスタンスが別の場所あるいはネットワークでさえからもそのクラスをロードするという可能性も考えられます。さらに、バグ修正が行われている開発環境を考えてみてください。プログラムを実行するたびにクラス・ファイルの内容が変わっているだけでなく、プログラムが実行されるまで Java コードが存在しないという可能性さえあります。これは、例えば Java リフレクション・サービスがプログラムのアクティビティーをサポートするために実行時に新しいクラスを生成することも珍しくないためです。

統計、フィールド、クラス、そしてメソッドに関する知識の欠如は、Java コンパイラーに含まれる最適化フレームワークの大部分にとって厳しい障害になるということを意味します。インライン化は、静的または動的コンパイラーによって適用されるおそらく最も重要な最適化ですが、呼び出しのターゲット・メソッドに関する情報がコンパイラーになければ、これを適用することもできません。

インライン化

インライン化は、関数の呼び出しコードを呼び出し側の関数に挿入することによって、実行時のプロローグまたはエピローグのオーバーヘッドを生じさせないコードを生成することを目的とした手法です。しかし、おそらくインライン化の最も大きなメリットは、最適化プログラムに可視となるコードの範囲が広がり、より優れた品質のコード生成が可能になるという点です。以下は、インライン化する前のコードの一例です。

```
int foo() { int x=2, y=3; return bar(x,y); }
final int bar(int a, int b) { return a+b; }
```

コンパイラーが、上記の bar が foo() 内で呼び出されるものだとわかれば、bar からのコードを foo() 内での bar() 呼び出しに置き換えることができます。上記の例では bar() メソッドが final なので、これが foo() で呼び出されることになります。実際上の例でも、動的 JIT コンパイラーは推測的にターゲット・メソッドのコードをインライン化し、大多数の場合はインライン化されたコードが正解となります。上記の場合、JIT コンパイラーは以下のコードを生成します。

```
int foo() { int x=2, y=3; return x+y; }
```

この例では、値の伝播と呼ばれる最適化に続いて簡易化を行うことにより、単純に 5 を返すコードを生成できます。インライン化を使用しなければこのような最適化は行えないため、パフォーマンスが大幅に下がる結果となります。静的コンパイルの場合でのように bar() メソッドが解決されない場合、この最適化は不可能なため、コードは仮想呼び出しを実行しなければなりません。その場合、実行時には 2 つの数値を加算するのではなく乗算する別の bar が呼び出しの実際のターゲットとなる可能性があります。このような理由から、インライン化は Java プログラムの静的コンパイル中にはネイティブに実行できません。

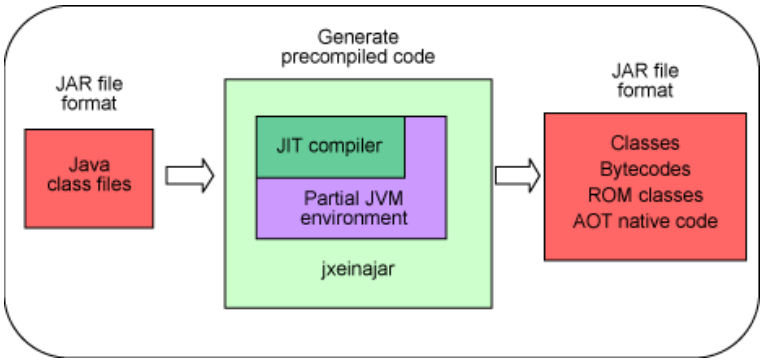
したがって、AOT コードはすべての静的参照、フィールド参照、クラス参照、そしてメソッド参照が未解決のまま生成され、実行時にこれらの参照の 1 つひとつが現行のランタイム環境に合った正しい値で更新される必要があります。このプロセスは、初めて実行する際のパフォーマンスに直接影響します。すべての参照は最初の実行時に解決されるためです。もちろん、それ以降の

実行ではコードをパッチした結果が有効になり、インスタンスまたは静的フィールド、つまりメソッドのターゲットはより直接的に参照されることになります。

それに加え、Java メソッドに対して生成されたネイティブ・コードに必要となる値は通常、単一の JVM インスタンスでしか使用できません。例えば、コードは JVM ランタイムの特定のラインタイム・ルーチンを呼び出して未解決のメソッドの検索やメモリーの割り当てなどの特定のアクションを実行しますが、これらのランタイム・ルーチンのアドレスは、JVM がメモリーにロードされるたびに変更される可能性があります。そのため、AOT でコンパイルしたコードは JVM の現行の実行環境に結合してからでないと実行できません。これ以外の例としては、ストリングのアドレスや、定数プール・エントリーの内部ロケーションなどがあります。

WebSphere Real Time では、AOT のネイティブ・コード・コンパイルは jxeinajar というツールで実行されます (図 2 を参照)。このツールは、ネイティブ・コードのコンパイルを JAR ファイルに含まれるすべてのクラスのすべてのメソッドに適用するか、あるいは対象のメソッドに選択的に適用します。コンパイルした結果は JXE (Java eXEcutable) として知られる内部フォーマットに保管されますが、この内部フォーマットへの保管は永続コンテナーへの保管と同じくらいに簡単です。

図 2. jxeinajar



すべてのコードを静的にコンパイルすれば、実行時に最大数のネイティブ・コードが実行されることになるため最善の方法だと思うかもしれませんが、それにはいくつかの代償が伴います。まず、コンパイルするメソッドが多くなればなるほど、コードが占有するメモリーも増えるという点です。コンパイルされたネイティブ・メソッドの大きさは、バイトコードの約 10 倍になります。ネイティブ・コード自体の密度がバイトコードよりも低い上、コードを JVM に結合して例外の発生時やスタック・トレースが要求されたときに正しく実行されるようにするために、コードに関するメタデータを追加で含めなければならないからです。平均的な Java アプリケーションを構成する JAR ファイルには通常、ほとんど実行されない多くのメソッドが含まれます。これらのメソッドがコンパイルされることで、メリットがほとんど期待できない上にメモリーが犠牲になるというわけです。このサイズ上の犠牲にはディスクへのコードの保管、ディスクから JVM へのコードの移動、そして JVM へのコードの結合という関連コストが伴います。コードが複数回実行されるのでなければ、これらのコストが、解釈と比較した場合のネイティブ・コードのパフォーマンス上のメリットで相殺されるとは考えられません。

このサイズ問題に対する対策は、コンパイルされたメソッドと解釈されたメソッドとの間の呼び出し (つまり、コンパイルされたメソッドが解釈されたメソッドを呼び出す場合、あるいはその逆の場合) が、解釈されたメソッド間でのメソッドの呼び出しやコンパイルされたメソッド間での呼

び出しよりもコストが高くなる可能性があるという事実を物語ります。動的コンパイラーは、最終的には JIT でコンパイルされたコードで頻繁に呼び出されるすべての解釈済みメソッドをコンパイルすることによってこのコストを軽減しますが、動的コンパイラーを使用しなければ、このコストを相殺することはできません。したがって、メソッドが選択的にコンパイルされる場合は、コンパイル済みメソッドから、コンパイルされないメソッドへの遷移を最小限に抑えるように注意する必要があります。ただし、正しい一連の正しいメソッドを選択して、考えられるすべての実行でこの問題を回避するのは容易なことではありません。

利点

AOT でコンパイルしたコードには、上記で概説した欠点や課題がありますが、Java プログラムを事前にコンパイルすることでパフォーマンス上のメリットがもたらされることは確かです。このメリットは、動的コンパイラーが常に効果的なソリューションとなるわけではない環境では顕著になります。

AOT でコンパイルしたコードを慎重に使うことで、アプリケーションの開始を迅速に行うことができます。AOT でコンパイルしたコードは JIT でコンパイルしたコードよりも通常は時間がかかりますが、解釈するよりは何倍も時間を短縮できるからです。その上、一般的に AOT でコンパイルしたコードをロードして結合する時間は、重要なメソッドを検出して動的にコンパイルする時間よりも短いため、プログラムの実行時にそのパフォーマンスをより早い段階で達成することができます。対話型アプリケーションでも同様に、動的コンパイルで応答性を下げることなくネイティブ・コードのパフォーマンスが短時間で有効になります。

RT アプリケーションでも、AOT でコードをコンパイルすることにより重要なメリットがもたらされます。そのメリットは、解釈されたパフォーマンスを超える一層確定性のあるパフォーマンスです。WebSphere Real Time が使用する動的 JIT コンパイラーは、特に RT システムでの用途を対象にしているため、コンパイル・スレッドは RT タスクよりも低いレベルで動作し、不確定性の高いパフォーマンス効果を持つコードが生成されないように調整されています。ただし RT 環境によっては、JIT コンパイラーが存在することさえ許容されません。そのような環境では通常、最も厳しく制御された期限管理が必要となります。その場合、AOT でコンパイルされたコードは、達成される確定性の程度に影響を与えることなく、解釈されたコードよりも優れた、コードそのままのパフォーマンスを提供できます。JIT コンパイル・スレッドを排除することで、優先順位の高い RT タスクを開始しなければならないときでも、そのスレッドをプリエンプトすることによるパフォーマンスへの影響がなくなります。

成績表

動的 (JIT) コンパイラーはアプリケーション実行の動的動作、そしてロードされたクラスとその階層に関する知識を活用することで、プラットフォームの中立性をサポートし、高品質のコードを生成します。ただし JIT コンパイラーには限られたコンパイル時間しか許されないため、プログラムのランタイム・パフォーマンスに影響する場合があります。一方、静的 (AOT) コンパイラーはプログラムの動的動作を活用できないこと、あるいはロードされたクラスやクラス階層に関する知識がないことから、プラットフォームの中立性とコード品質が犠牲になりますが、AOT コンパイルには事実上無制限のコンパイル時間が与えられています。AOT のコンパイル時間はランタイム・パフォーマンスに影響しないことがその理由ですが、実際に開発者が静的コンパイルのステップを永遠に待つことはないでしょう。

表 1 に、この記事で説明した Java 言語に関する動的コンパイラーと静的コンパイラーの特性を抜粋して要約します。

表 1. コンパイル手法の比較

	動的 (JIT)	静的 (AOT)
プラットフォーム中立性	あり	なし
コード品質	優良	良
動的動作の活用	あり	なし
クラスおよび階層の知識	あり	なし
コンパイル時間の制限	制限あり。ランタイム・コストあり	非常に低い制限。ランタイム・コストなし
ランタイム・パフォーマンスの影響	あり	なし
コンパイル対象	注意が必要。JIT が処理	注意が必要。開発者が処理

どちらの技術にしても、コンパイルするメソッドを慎重に選択して最大限のパフォーマンスを実現するようにしなければなりません。動的コンパイラーの場合、コンパイラー自体がこの選択を行いますが、静的コンパイラーでは開発者に選択が任されます。JIT コンパイラーにコンパイル対象のメソッドを選択させるのは、コンパイラーの経験則による作業がその特定の状況でどれだけ有効であるかによって、メリットになる場合もならない場合もありますが、大多数の場合はメリットになるはずです。

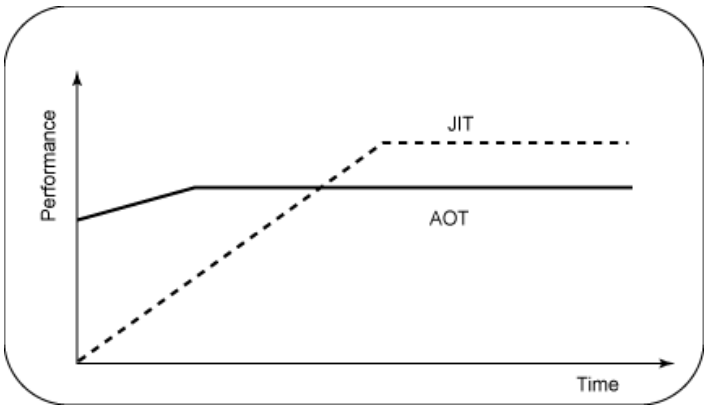
JIT コンパイラーは実行中のプログラムを極めて適切に最適化できるため、多数の実稼動 Java システムにとって最も重要な安定状態のパフォーマンスを実現するには静的コンパイルよりも適しています。一方、対話型パフォーマンスとなると、ユーザーの応答時間の期待を妨げる実行時のコンパイル・アクティビティーが一切ない静的コンパイルの方が威力を発揮します。開始時のパフォーマンスと確定性のあるパフォーマンスは、動的コンパイラーを調整することである程度対処できますが、静的コンパイルであれば必要に応じて開始時間を最大限に短縮化し、最高レベルの確定性を実現できます。表 2 では、この 2 つのコンパイル技術を 4 種類の実行環境で比較しています。

表 2. それぞれの技術が最適な分野

	動的 (JIT)	静的 (AOT)
開始時のパフォーマンス	調整可能。ただしそれほど優れてはいない	最適
安定状態のパフォーマンス	最適	良
対話型パフォーマンス	それほど優れていない	良
確定的パフォーマンス	調整可能。ただし最適ではない	最適

図 3 に、開始時間のパフォーマンスと安定状態のパフォーマンスでの一般的傾向を示します。

図 3. AOT と JIT のパフォーマンス比較



JIT コンパイラーでのパフォーマンスは、メソッドが最初に解釈されるために最初は非常に低いレベルです。コンパイル済みのメソッドが増え、JIT がコンパイルに費やす時間が少なくなるにつれてパフォーマンス曲線は伸び、最終的にはピーク・パフォーマンスに達します。一方、AOT でコンパイルされたコードは初め、解釈されたコードより遥かに優れたパフォーマンスを示しますが、JIT コンパイラーで達成されるパフォーマンスには届きそうにもありません。静的コードを JVM インスタンスに結合するにはある程度のコストがかかるため、パフォーマンスは出だしで安定状態のパフォーマンスよりも低い値になります。ただし、安定状態のレベルに届くまでの時間は、JIT コンパイラーに比べると断然短くなります。

すべての Java 実行環境に適しているネイティブ・コードのコンパイル技術は 1 つもありません。それぞれの技術に、他の技術が得意としない分野での長所があるため、Java アプリケーション開発者の要求を満たすには両方のコンパイル技術が必要となるのです。実際、静的コンパイルと動的コンパイルを併せて使用すれば、パフォーマンスを向上させる可能性が最大限に広がるはずです。ただしこの組み合わせは、Java 言語の一番の売りであるプラットフォーム中立性が問題とならない場合に限られます。

まとめ

この記事では、JIT コンパイラーという形での動的コンパイルと静的 AOT コンパイルのどちらが優れているかということを本題として、Java 言語のネイティブ・コードのコンパイルに関する問題を検討しました。

動的コンパイラーはこの 10 年間で飛躍的に成熟し、多種多様な Java アプリケーションが、C++ や Fortran などの静的にコンパイルされた言語での実装によって実現可能なパフォーマンス、あるいはそれ以上のパフォーマンスを実現できるまでに至っていますが、動的コンパイルは一部のアプリケーションや実行環境にはまだそれほど適していません。AOT コンパイルは、この動的コンパイルの欠点に対する解決策としてもてはやされているものの、Java 言語自体の動的な性質により、ネイティブ・コンパイルの完全な可能性を実現するには課題が残されています。

いずれの技術にしても、Java 実行環境でのネイティブ・コードのコンパイルに関するすべての要件を解決することはできませんが、それぞれが得意とする分野でのツールにはなります。つまり、この 2 つの技術は互いを補完するものなのです。ランタイム・システムで両方のコンパイル・モデルを適切に使用すれば、非常に広範囲のアプリケーション環境で開発者とユーザーにメリットをもたらすことになるでしょう。

著者について

Mark Stoodley

Mark Stoodley は、2001 年に University of Toronto でコンピューター・エンジニアリングの博士号を取得し、その翌年に、IBM Toronto Lab で開発された Java JIT コンパイル技術に携わるため同研究所に入社しました。2005 年初期からは既存の JIT コンパイラーをリアルタイム環境で動作するように適応させることによる、IBM WebSphere Real Time 向け JIT 技術に取り組んでいます。現在は Java コンパイル制御チームのリーダーとして、ネイティブ・コード・コンパイルの有効性をその実行環境で改善するという作業を行っています。余暇には、自宅の改装を楽しんでいます。

Kenneth Ma

Kenneth Ma は、電気工学を学んだ University of Waterloo を 2003年、応用科学の学士として卒業し、その後まもなくして IBM に入社しました。IBM での共同作業期間中に iSeries ツールの開発者として IBM WebSphere プラットフォームに携わり、現在は IBM Testarossa JIT Compiler チームのメンバーとして活躍しています。過去 2 年間は IBM J9 Java Virtual Machine を対象とした AOT コンパイル技術の実装と改善に取り組み、最近ではこの技術を Java SE 環境に移行させています。

Marius Lut

Marius Lut はルーマニアの Polytechnic University of Timisoara で、オートメーションおよびコンピューター・エンジニアリングを学びました。IBM Toronto Lab に入社したのは 1998年のことです。以来、z/OS対応の静的 IBM HPJ コンパイラー、IBM Sovereign JIT コンパイラー、そして過去 3 年は IBM Testarossa JIT 技術に取り組んでいます。その他にも、J2SE、J2ME コンパイラー、組み込みシステムのソフトウェア開発と設計、RT 環境、メインフレーム、そして Intel プロセッサー・ベースのシステムなど、豊富な経験を持ちます。