BEN EVANS

Java HotSpot VMコード・キャッシュについて

コード・キャッシュのあふれを検出し、回避する方法を学ぶ

Java HotSpot VMには高度なJust-In-Time（JIT）コンパイラが搭載されています。このコンパイラにより、Java HotSpot VMが稼働するすべてのプラットフォームに対して、高度に最適化されたマシン・コードを生成できます。

本記事では、Java HotSpot VMのJITコンパイラの1つの重要な側面である、コード・キャッシュについて説明します。コード・キャッシュを理解することで、他の方法では追跡が困難なさまざまなパフォーマンス問題を把握できるようになります。

注：Java Magazineの本号の記事「JITコンパイラの実際の動作」と過去の記事「Java HotSpot VMにおけるJITコンパイルの概要」（PDF）では、Java HotSpot VMおよびJITコンパイラの入門レベルのトピックについて取り上げています。

JITコンパイラおよびコード・キャッシュの話を始める前に、まずはJavaメソッドのライフサイクルについて考えます。

Javaメソッドのライフサイクル

Javaプラットフォームで実行中のプログラムに新たにロードおよびリンクされるコードの最小単位はクラスです。そのため、新しいメソッドが読み込まれるときには、そのメソッドを含むクラスのクラス・ローディング・プロセスが必ず実行されます。

このクラス・ローディング・プロセスはピンチ・ポイント、すなわちJavaプラットフォームのセキュリティ・チェックが多数集中する場所として機能します。したがって、Javaメソッドのライフサイクルは、実行中のJVMに新しいクラスを読み込むクラス・ローディング・プロセスから始まります。

クラス・ローディング

クラス・ローディングは、（通常はディスクから読み込まれる）バイト・データのストリームから始まります。このストリームはクラス・ファイル形式である必要があります。バイト・ストリームがクラス・ファイル形式の場合は、クラス・ローダーでリンク処理を実行できます。

このリンク処理のプロセスは複数のフェーズで構成されます。第1のフェーズは検証フェーズで、これはもっとも重要なフェーズでもあります。検証フェーズとは、新しいクラス・ファイルがJavaの堅牢なプログラミング・モデルを侵害しないことをJVMで確認するためのフェーズです。

検証フェーズの間に、いくつかのセキュリティ制約がチェックされます。検証内容の例を以下に挙げます。

- メソッドがアクセス制御キーワードを正しく使用していること
- 正しい静的型を指定してメソッドが呼び出されていること
- 適切に型付けされた値のみが変数に代入されていること
- 変数が使用される前に適切に初期化されていること

メソッドのバイトコードに対しても広範なチェックが実行されます。ここでの重要なポイントは、JVMがスタック・マシンであることです。

この方式は熟考を重ねて選択されました。レジスタベースのマシンと比較して、スタック・マシン上ではセキュリティなどに関するプロパティの正当性を非常に容易に証明できます。したがって、バイトコードに対するチェックの大部分を、クラス・ローディングの時点で静的な解析により経済的に実行できます。その結果、危害をもたらすコードが実行中のJVMに入り込む可能性が非常に低くなります。

たとえば、レジスタの内容を追跡しなくても、メソッドのあらゆ

写真：JOHN BLYTHE

る地点でスタックの状態を推測できます。

注意点として、パフォーマンス上の理由から、JDK関連のクラス（rt.jarに含まれるもの）はチェックされません。JDK関連のクラスは、原始クラス・ローダーによってロードされます。原始クラス・ローダーは包括的なセキュリティ・チェックを実行しません。

このようにクラス・ローディングをバイトコード検証の機会として利用することで、クラス・ローディング・プロセスの速度は低下します。しかし、その見返りとして、実行時の速度は飛躍的に向上します。一度チェックを行っておけば、コードを実行するときにチェックを省略できるからです。

Java HotSpot VMでのクラス・ローディングの実装方法

バイト・ストリームをクラス・オブジェクトに変換するために使用する重要なメソッドは、Javaメソッドの`ClassLoader::defineClass()`です。このメソッドはネイティブ・メソッドの`ClassLoader::defineClass1()`に処理を委譲し、このネイティブ・メソッドでは基本的なチェックと文字列変換を行った後に`JVM_DefineClassWithSource()`というC関数を呼び出します。

お分かりのとおり、このC関数がJVMへのエントリ・ポイントとなり、このC関数を通じてJava HotSpot VMのC++コードにアクセスできます。Java HotSpot VMでは`SystemDictionary`を使用して、`ClassFileParser`の`parseClassFile()`メソッドにより新しいクラスをロードします。

クラス・ローディングが完了すると、メソッドのバイトコードがC++オブジェクト（`methodOop`）内に配置され、バイトコード・インタプリタで使用できる状態になります。

この処理はメソッド・キャッシュと呼ばれることもありますが、実際にはパフォーマンス上の理由から、バイトコードは`methodOop`内にインラインで保持されます。

メソッドのコンパイル方法

Java HotSpot VMのバイトコード・インタプリタ内では、多数のパフォーマンス・カウンタやトレース・カウンタが管理されます。サーバー・コンパイラの場合、メソッドは10,000回実行されるとコンパイルされます。

コンパイラから出力されるコードはマシン・コード（特定のオペレーティング・システムおよびCPUで使用するための専用コード）です。マシン・コードは、中心的な要素である`CodeCache`（C++オブジェクト）内に配置されます。CodeCacheは`CodeBlob`インスタンス（コンパイル後のメソッド・コードの表現）を保持するヒープのような構造体です。

コード・ブロブがコード・キャッシュ内に配置されると、実行中のシステムが、インタプリタ・モードから、新しくコンパイルされたコードを使用するモードに切り替えられます（ポインタの更新を含むこの更新処理はポインタ書き換え（pointer swizzling）と呼ばれることもあります）。

PrintCompilation

JITコンパイル・サブシステムの制御に使用できる非常にシンプルなフラグの1つが`-XX:+PrintCompilation`です。このスイッチを指定することで、JITスレッドによってコンパイル関連のメッセージが標準ログに追記されるようになります。`PrintCompilation`については、前述の2つ目の参照記事で詳細に説明しています。

脱最適化

Java HotSpot VMのサーバー・モードで実行される最適化は、常に妥当であるとは限りません。そのため、サニティ・チェック（ガード条件と呼ばれることも多い）を使用して、最適化が適切に行われるように保護します。このチェックが失敗した場合は、妥当でない推測に基づいて生成されたコードの脱最適化を実行します。

その後、Java HotSpot VMによって、再検討の後にまた別の最適化が試されることもよくあります。そのため、同じメソッドに対して脱最適化と再コンパイルが何度か実行されることがあります。

脱最適化イベントは、`PrintCompilation`ログで「made not entrant」や「made zombie」などの行により示されます。

これらの行は、すでにコンパイルされてコード・ブロブが生成されていた特定のメソッドが、脱最適化されたことを示します。脱最適化は一般的に、新しいクラスがロードされ、Java HotSpot VMによる推測が無効になったことが原因で実行されます（ただし、別の原因の場合もあります）。

プログラムの準備中の処理

Javaプログラムが起動して初期化フェーズが終了した後は、通常は平常運用に移行し、コードのホット・パスが現れ始めます。

PrintCompilationスイッチをオンにしてプログラムを複数回実行し、コンパイルされたメソッドに関するログを収集した場合、以下のようなパターンが浮かび上がります。

- コンパイルは通常、最終的に停止する
- コンパイルされたメソッドの数が安定する
- 同じプラットフォーム上で同じテスト入力に対してコンパイルされた一連のメソッドは、通常はかなり一致する
- コンパイルされたメソッドの細部は、実際に使用するJVM、オペレーティング・システム・プラットフォーム、CPUによって異なる

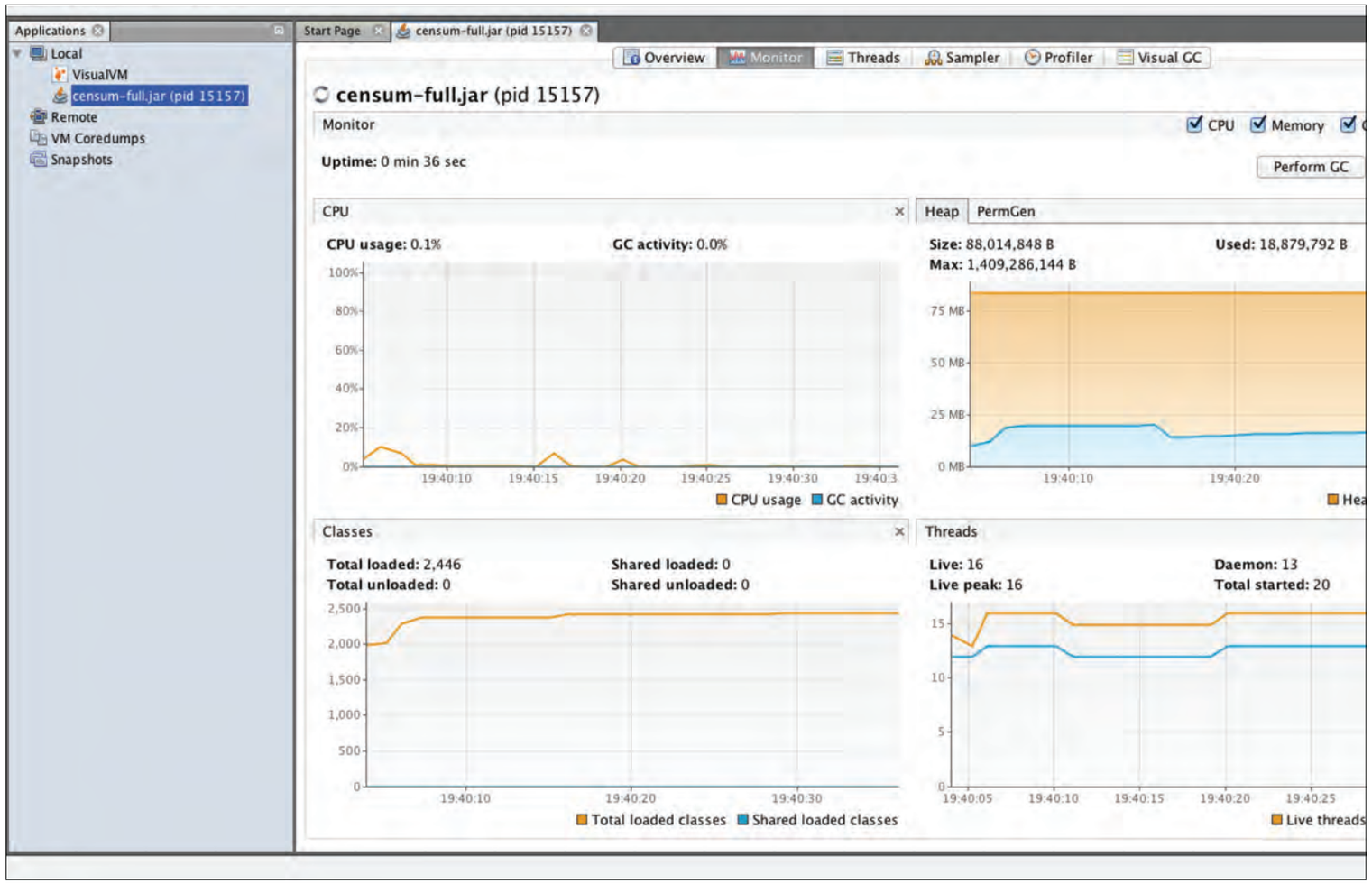

図1: Java VisualVMでのクラス・ローディング・データの表示

注: ある特定のメソッドのコンパイル済みのコードが、すべてのプラットフォームで同程度のサイズになることは保証されていません。

同程度のサイズになるのが一般的なパターンですが、通常とは異なる様相を示す場合もあるため、常にチェックを行う必要があります。便利なチェック方法の1つがJava VisualVMを使用することです。Java VisualVMのClassesセクション（図1の左下のパネル）には、クラス・ローディングの概要を示すグラフが示されます。

コード・キャッシュがあふれた場合の動作

一言で言えば、コンパイルが停止します。停止の原因は、コンパイル後のコード・ブロブをコード・キャッシュから削除する方法は通常、脱最適化しかないためです。

コード・キャッシュ領域は、コード・キャッシュから無効となった「zombie」コード・ブロブをフラッシュすることで解放されます（呼出し元の変更などにより今後は使用されなくなった「not entrant」ブロブは、時間の経過とともにzombieとなります）。

Java 7 Update 4以降のJDKのバージョンでは、投機的フラッシュという新たなコード・キャッシュ・フラッシュの方法が追加されました。このアプローチでは、古いメソッドがフラッシュの対象としてマークされ、このメソッドを保持するmethodOopとのリンクが切られます。このコンパイル済みのメソッドをVMで呼び出すことが必要になった場合、メソッドが再度methodOopにリンクされ、フラッシュの対象から外されます。

しかし、このメソッドがある一定時間再び呼び出されることがない場合は、methodOopはインタプリタ・モードに戻り、コード・ブロブがフラッシュの対象になります。

起動時の動作

アプリケーションの起動時間がコード・キャッシュにとって問題になりうる原因を理解するために、想像上のSpringアプリケーションを取り上げます。

Springアプリケーションは、Bootstrapクラスを使用して起動します。このクラスは、作成と組

み立てを要するインスタンスの詳細を記載したXMLファイルを探します（このファイルにはロード対象のクラスが定義されます）。

そのため、Springアプリケーションのクラス・ローディングには2つのフェーズがあります。第1フェーズではブートストラップの開始に必要なクラスをロードし、第2フェーズではアプリケーション・クラスを通常どおりロードします。

JITコンパイルの観点からは、このように2つのフェーズがある点が重要です。Springフレームワークでは、ロードとインスタンス化の対象となるクラスを検出するために、リフレクションやその他のさまざまな技術を駆使していることがその理由です。このようなフレームワーク関連のメソッドはアプリケーション起動時に頻繁に呼び出されますが、その後はまったく使用されません。

あるSpringフレームワークのメソッドがコンパイル対象となるほどの回数実行されたとしても、アプリケーションからはそのメソッドはほとんど利用されません。その結果、アプリケーションの起動時間はわずかに向上しますが、その代償として、稀少なリソースが使い果たされてしまいます。フレームワークのメソッドが大量にコンパイルされた場合、コード・キャッシュ全体が使い果たされ、本当にコンパイルする必要があるアプリケーションのメソッドのために使用する領域がなくなる可能性があります。

このような問題を解決するために、JVMでは時間の経過とともにカウンタ値を減らす手法を使用します。もっとも単純な方法では、30秒ごとに50%ずつ、メソッド呼出しのカウンタ値を減らします。

つまり、メソッドが起動時のみに使用される場合、メソッド呼出しのカウンタ値は数分以内に、事実上ゼロまで減算されることになります。その結果、使用頻度の低いSpringフレームワークのメソッドが貴重なコード・キャッシュ領域を消費する事態を回避できます。

コンパイルとコード・キャッシュを制御するスイッチ

コンパイルとコード・キャッシュを制御するスイッチは以下のとおりです。

- -XX:+PrintCompilation：コンパイル・イベントと脱最適化イベントに関するログ・エントリを表示
- -XX:CompileThreshold=n：メソッドがコンパイルされるための条件となるメソッド呼出し回数を変更
- -XX:ReservedCodeCacheSize=YYm：使用するコード・キャッシュの総サイズを設定
- -XX:+UseCodeCacheFlushing：利用頻度の低いコード・ブロブのフラッシュをJVMに許可（Java 7 Update 4以降ではデフォルトでオン）

コード・キャッシュがあふれているアプリケーションの修復方法

コード・キャッシュのあふれを検出してその問題を解決するためには、まずキャッシュが限界状態にあることを確認する必要があります。「compilation halted」警告が発行されるときは常に、キャッシュが限界状態にあります。キャッシュのサイズが小さすぎるか否かを以下の手順で確認し、問題を修復することができます。

1. -XX:+PrintCompilationを使用して、実際にコンパイルされているメソッドを出力します。
2. この出力が安定状態になるまで待機します。
3. 数回の実行を繰り返します。結果セットが安定していることを確認します。.
4. -XX:ReservedCodeCacheSizeを使用して、コード・キャッシュのサイズを増やしてみます（まずは2倍にすると良いでしょう）。コンパイルされたメソッドの数が増加した場合は、元のコード・キャッシュが小さすぎたと見なすことができます。
5. 全体的なパフォーマンスの再テストを実行し、コード・キャッシュのサイズを増やすことでアプリケーション・パフォーマンスの他の側面に悪影響が出ないことを確認します。

以上のような最適化には次のような重要な経験則があります。すなわち、一度変更を加えたら、その結果を慎重に測定しなければなりません。本記事で説明したように、どのような変更を試すべきかを知るためには、まず、JVMが実際にどのように動作しているかを理解することが肝心です。

Ben Evans：London Java Communityの運営を助けるとともに、ユーザー・コミュニティの代表としてJCP Executive Committeeに参加。

learn more

オラクルのJVM仕様（Java SE 8）