C-024

起動時検証処理の省略による Java™ 起動高速化方式の検討

Study of Java™ Boot Acceleration Method by Verification Omission During Start-Up

福井大輔
Daisuke Fukui

(株) 日立製作所　横浜研究所
Hitachi, Ltd., Yokohama Research Laboratory

1. はじめに

カーナビやホームゲートウェイ、プリンタといった様々な機器に Java 仮想マシンを搭載し、携帯電話と同様に Java アプリケーションをインストール可能にすることで、機器の利便性向上と他社差別化を計る動きが出ている。また、様々な機器上で動作する Java アプリケーションを統一的な手法で管理するための基盤として、OSGi と呼ばれる技術に注目が集まっている。

OSGi では Java アプリケーションを「バンドル」と呼ばれる単位で管理し、バンドルのインストールや削除、起動、停止といったライフサイクル制御を可能にする。OSGi を搭載した機器では、ユーザが様々なサービスを利用することで非常に多くの（例えば 100 以上の）バンドルがインストールされる可能性がある。こうした環境では、機器の起動時に多くのバンドルが一斉に処理を開始するため、起動時間が長くなる。この問題を解決するには、Java プログラムの起動時間を短縮する仕組みが求められる。

そこで本研究では、組み込み機器上で動作する Java プログラムを対象に、起動処理の分析を行った。その結果、クラスファイルの検証処理にかかる時間が Java プログラムの起動時間に大きな影響を与えることが分かった。しかし、検証処理は Java プログラムの安全性に関わる重要な処理であるため、検証処理時間を短縮するには安全性の維持が最も大きな課題となる。本研究では安全性を維持しつつ検証処理を最適化することで Java プログラムの起動時間を短縮する方法について検討する。

2. 起動処理の分析

gcc のプロファイラである gprof を利用し、Java プログラムの起動処理を分析した。結果を図 1 に示す。評価結果が示す通り、インタプリタの処理は起動時間の 44%となっている。インタプリタは Java のバイトコード命令を読み込んで解釈実行する部分であり、プログラムロジック本体の処理時間を表すものである。起動時はプログラムロジック以外の処理に 56%が費やされていることが分かる。

次いで処理負荷が高いのは参照解決である。Java 仮想マシンはシンボル参照によるクラスの動的リンクを前提として設計されている。Java クラスファイル内のコンスタントプールと呼ばれる領域にシンボル参照情報が格納されており、この情報を基に Java 仮想マシンはクラスを動的にロードする。この処理の多くは起動時に発生することから、起動時間の 15%を占めるという結果が示されている。

その他、VM 処理、同期処理、検証処理などの負荷が高くなっているが、参照解決を含め、これらの多くはクラスロードに関する処理である。その処理負荷は GC の 2%を除くと最大で起動時間の 54%に達することから、本研究ではまずクラスロード処理に着目し、処理内容の分析と課題抽出を行った。

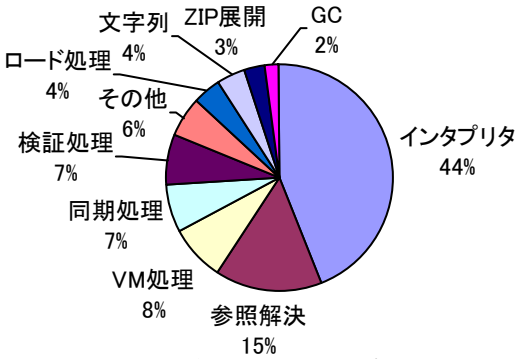

図.1　起動処理の分析結果

3. クラスロード処理の課題

Java 仮想マシンは未ロードのクラスを検出すると、クラスローダと呼ばれるモジュールを利用してクラスロード処理を行う。処理内容は実行順に大きく分けると下記の通りである。

① クラスファイル検索、ZIP 展開、パス 1 検証
② クラスフォーマット検証（パス 2 検証）
③ 親クラス・インタフェース・this クラスロード
④ バイトコード検証（パス 3 検証）
⑤ 実行時検証（パス 4 検証）
⑥ 参照解決
⑦ 初期化

まず最初に、クラスローダはロード対象クラスのクラス名を基にクラスファイルを検索する。検索場所についてはクラスローダの実装に依存して決まる。検索対象が JAR ファイルの場合、ロード対象クラスが見つかると、JAR ファイルの該当エントリを ZIP 展開してクラスファイルデータをメモリ上に配置する。この際、クラスファイルのマジックナンバー等をチェックするパス 1 検証が行われる。

Java 仮想マシン仕様[1]によると、検証処理はパス 1 からパス 4 の 4 段階に分かれる。パス 2 はクラスファイルに含まれる情報の内、バイトコード以外の情報に関するフォーマットをチェックする。例えばクラス参照やメソッド参照のインデックスがコンスタントプールの有効範囲内であるか、などがチェックされる。

パス 3 のバイトコード検証は、検証処理の中で最も負荷の高い処理を行う。メソッド毎のバイトコードをチェック

し、オペランドスタックのオーバーフロー・アンダーフローのチェックやローカル変数・コンスタントプールの不正インデックスのチェックなどを行う。

パス 4 の実行時検証は、メソッド参照先のクラスが存在するか、またメソッドディスクリプタに一致するメソッドが存在するかなどを実際に他クラスのロード処理を行うことでチェックする。

こうした検証処理は Java プログラムの安全性を保障する上で重要な機能であるが、一度検証したクラスファイルであっても起動時に毎回検証処理を行うため、起動時間を延ばす要因の一つになっている。よって、安全性を保障しつつ検証処理を省略できれば、起動時間の短縮に大きな効果があると考えられる。

4. 起動時における検証処理の省略

検証処理を省略する方法として、Java 仮想マシンの起動オプションを利用する方法がある。検証処理設定用起動オプションには、ALL/REMOTE/NONE のいずれかの値を設定できる。ALL を設定した場合は全ての検証処理が実行される。REMOTE を設定した場合はパス 2 検証の一部を省略できる。NONE を設定した場合は一切の検証処理を省略できる。

検証処理オプションを NONE に設定することにより起動時間は最短となるが、セキュリティ上のリスクを抱えることになる。全ての検証処理を省略することにより、悪意のあるプログラムによってシステムが破壊される恐れがある。そこで本研究では安全性が確認されたクラスのみを検証処理の対象から外す方式を採用した。

本研究ではまず、プログラムの起動時にロードされた検証済のクラス（以下、起動時ロードクラス）を専用のプロファイラで検出する。次に、プロファイル結果を基に、起動時ロードクラスを纏めて JAR ファイルの先頭部分に格納する。最後に、起動時ロードクラスの数を MANIFEST ファイルに記述する。このようにして作成された JAR ファイルを用いて起動時に MANIFEST ファイルをチェックし、記載されているクラス数だけ JAR ファイルの先頭から起動時ロードクラス群を一括ロードする。さらに起動時ロードクラスの検証処理を省略することで起動時間を短縮する(図 2)。

特定の JAR ファイルに格納された起動時ロードクラスのみを対象として検証処理を省略することで、安全性を維持したまま起動時間を短縮できる。起動時に悪意のあるクラスを別の JAR ファイルから動的ロードした場合は通常通り検証処理が実施される。また、起動後は通常通りの検証処理を行うため、起動後の検証要否判定は不要になり、本システム導入による処理負荷を最小限に抑えることができる。

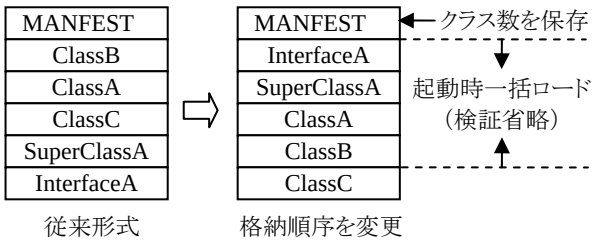

図.2　クラスファイルの格納順序を最適化

5. 評価結果と考察

本研究で開発した高速化技術の効果について評価を行った。評価環境を表 1 に示す。

表 1. 評価環境

項目	評価環境
ハードウェア	ルネサス販売製 Voyager RTS7751R2D
CPU	SH4/7751R 240MHz
キャッシュ	命令 16Kb データ 30Kb
バスクロック	120MHz
メモリ	64Mbyte
二次記憶	HDD
OS	Linux2.4.19
Java 実行環境	CDC HI 1.0.1
プロファイル	Foundation Profile

プログラムの「起動時」の定義は、プログラムの main メソッドが呼ばれた直後から、main メソッドが返る直前までとしている。これは多くの Java プログラムが main メソッドの実行スレッドで起動処理を行うと推測されるためである。評価対象プログラムには、筆者が過去に実装した一般的な Java プログラムを採用した。評価結果を図 3 に示す。

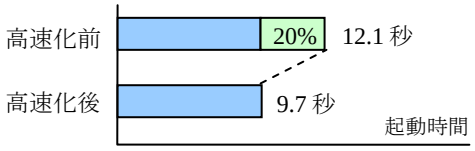

図 3. 起動高速化方式の評価結果

本研究で開発した高速化技術により、当初 12.1 秒であった起動時間を 9.7 秒に短縮した。短縮率は 20%である。これは図 1 で示した検証処理の割合である 7%を大きく上回る。従来の JAR ファイル形式では、クラスファイルの格納順序は特に規定されておらず、ZIP 展開時に不要な検索処理や関数呼び出し、およびディスク I/O が発生していた。本研究方式ではこれらの処理負荷も軽減されるため、単に検証処理を省略した場合よりも起動時間の短縮効果が高まったと考えられる。

結果が示すように、検証処理およびその他のクラスロード処理が Java プログラムの起動時間に与える影響は大きい。本研究では Java の動的ロードの特性を維持したまま起動時間を短縮する方法について述べたが、Java 仮想マシンに静的リンクすることでさらに高速化する方法も検討を進めている。しかしこの方法では動的ロードの利点が失われるため、今後さらなる検討が必要である。

参考文献

[1] Tim Lindholm・Frank Yellin, The Java Virtual Machine Specification, Addison-Wesley, Inc., 1996