6K-4

Javaにおけるリフレクションの特殊化

Specialization for Reflection in Java

杉本 駿[†] Shun Sugimoto　**古関 聰**[††] Akira Koseki　**小松 秀昭**[††] Hideaki Komatsu　**深澤 良彰**[†] Yoshiaki Fukazawa

1　始めに

Java におけるリフレクションは RMI(Remote Method Invocation) の実装などにも使用される有用な API であり、SOA(Service Oriented Architecture) などの分散システムアーキテクチャにおいて重要なインターフェースである。しかし、リフレクション処理は実行時のコストが非常に大きいことで知られている。そこで本稿では、このリフレクション処理に特殊化を施し、実行時のコストを下げる手法を提案する。本手法では、リフレクション処理に対し実行時にプロファイルをとり、オブジェクトの属するクラスやアクセスされるフィールドやメソッドを把握する。その情報を利用して最適化対象のプログラムのバイトコードを書き換えることで、リフレクション処理を直接参照に変換する。これによりリフレクション処理にかかる大きなオーバーヘッドを削減することができる。

2　研究目的

Java におけるリフレクションを利用すると、JVM(Java Virtual Machine) にロードされたクラスの内部情報をコードからアクセスしたり、ソースコードで選択したクラスではなく実行時に選択したクラスを扱うことができる。そのため、リフレクションは柔軟性の高いアプリケーションを構築するのに非常に有効なプログラム手段である。SOA などのシステムでは、リモートで接続されたマシン同士がお互いにシステムのコンポーネントとなり、通信し合うことで機能を共有している。このマシン同士で、新たに実装の異なるコンポーネントが追加された場合などでも、リフレクションを利用すれば柔軟に対応することができる。

リフレクションの柔軟性により、ローカルに接続された自分のマシン内の自分の JVM 自身へのアクセスにも RMI が利用されやすい。これらは、本来リフレクションによって実装される必要がないが、システムの保守・運用性の面から全てリフレクションによって実装される。これらの処理がシステム全体で大きなボトルネックとなってしまう。従って、リフレクション処理のコスト削減は重要である。

3　本手法の特徴

本稿で提案するのは、リフレクション処理そのものの実行コストを下げることではなく、特殊化によって、リフレクションを使わずに実行コストが低く同じ動作をする処理に置換するものである。ただし、静的な特殊化ではリフレクションのメリットである柔軟性を残しつつ、代替するのは難しい。そこで、動的な特殊化を施すことにより、開発者が意図するリフレクションの柔軟性を損なうことなく、実行コストを下げる手法を提案する。

本手法では、プログラムの中のリフレクション処理に対して、実行時にプロファイルをとり、オブジェクトの属するクラスやアクセスされるフィールドやメソッドを把握しクラス情報として保持する。そのクラス情報を基に、アクセス頻度の高いフィールドやメソッドへのリフレクション処理の部分のバイトコードを JIT(Just In Time) コンパイラによって動的に直接参照に書き換えることで、実行時のオーバヘッドを大きく削減することができる。

4　本手法の流れと適用例

始めに、本手法の流れを提示し、その後に実際に Java ソースレベルでの適用例を示す。

4.1　実行時プロファイル

本手法は現在主流の適応的コンパイル方式の JVM/JIT コンパイラでのプログラムの実行を前提としている。適応的コンパイルでは、起動当初はインタプリタとして実行し、よく呼び出されるメソッドや繰り返し実行されるようなコードの検出 (プロファイリング) を行い、その部分をネイティブコードにコンパイルすることで次回からはインタープリタ方式ではなくネイティブコードを直接実行する。これにより、起動時のオーバヘッドや利用メモリ増大を抑えつつ、効率よく実行速度を向上することができる。

本手法では、プロファイル時にリフレクションによってアクセスされるオブジェクトの属するクラスやアクセスされるフィールドやメソッドを把握すると同時に、そのアクセス頻度も含めてクラス情報として保持する。

4.2　バイトコード書き換えによる特殊化

プロファイルによって得たクラス情報を基に、特殊化を施す。

リフレクションによってアクセスされるフィールドやメソッドが何度も呼び出される場合、その度にクラス情報から文字列による検索が行われてしまうため、非常にオーバヘッドが大きくなってしまう。そこで、アクセス頻度の高いフィールドやメソッドはリフレクション処理に入る前に条件分岐式によって直接参照処理に流れるように特殊化を施す。また、その際にアクセス頻度の高い順番に条件分岐を施すことにより、条件分岐の判定式のオーバヘッドを少しでも削減する。

[†] 早稲田大学大学院理工学研究科
[††] 日本 IBM(株) 東京基礎研究所

4.3 本手法の適用例

Java のソースコードレベルで本手法の適用例を示す。

ここでは、リフレクションによるメソッド呼び出しを例に説明する。このメソッドは Integer クラスのオブジェクトを一つ引数として受け取る。”o”は呼び出すメソッドが属するクラスのオブジェクトで、”method_ name”が呼び出すメソッド名である。

以下、例のソースコードを示す。

<リフレクション処理によるメソッド呼び出し>

```
Object o = new Class_name();
…
Integer i = new Integer(0);
Class c = o.getClass();
Class[] m_arg = {Integer.class};
Method m = c.getMethod("method_name",m_arg);
Object[] o_arg = {i};
m.invoke(arg);
```

このプログラムをプロファイルした結果、オブジェクト”o”が属するクラスが、クラス A(40%)、クラス B(20%)、クラス C(30%)、その他 (10%) であったとする。すると、特殊化は以下のように施される。

<リフレクション処理に特殊化を加えたメソッド呼び出し>

```
Object o = new Class_name();
…
Integer i = new Integer(0);
if(o instanceof A){
   //直接参照によるメソッド呼び出し
   ((A)o).method_name(i);
} else if(o instanceof C){
   ((C)o).method_name(i);
} else if(o instanceof B){
   ((B)o).method_name(i);
} else {
   //リフレクション処理
   …(略)…
}
```

これにより、クラス A,B,C の合計 90%はリフレクションによる実行ではなく、直接参照によって実行されることになる。さらに、アクセス頻度の高い順、A,C,B の順番で分岐させることにより、条件分岐の判定式の実行回数を抑えている。これらの特殊化をプログラムの実行時に動的に JIT コンパイラによって施すことによって、リフレクション処理にかかるオーバヘッドを削減している。

5 評価

本手法を適用した際に、オーバヘッドがどの程度削減できるかをテストし、その結果を以下に示す。

ベンチマークに使用したプログラムは、簡単なメソッドをリフレクションにより 500,000 回呼び出すものである。今回は、このプログラムを一度コンパイルした後、生成されたバイトコードと、それを BCEL[1] の CCK(Class Construction Kit)[2] を使って書き換えて特殊化を施したもう一つのバイトコードとで、実行時間を比較することで行う。また、引数の種類 (クラス)、数をいくつか変えて実験を行った。

なお、実行環境は Intel(R) Pentium(R) M processor 1.60GHz、512MB RAM、jre1.5.0_6 である。ここで、表 1 に実験結果の一部を示す。

実行時間 (msec)		
引数	特殊化前	特殊化後
引数無し	1102	20
Integer(1)	1102	50
Integer(2)	1132	50
Integer(3)	1201	70
Boolean(1),Double(3)	1262	90
Short(4)	1503	290
Integer(2),String(3)	1522	300
Boolean(1),Integer(2),String(2)	1502	270
Long(6)	1332	131
Integer(7)	1382	140

表 1: 本手法を適用したプログラムの実験結果

ここで、左側の引数欄のクラス名の右の括弧内の数字は、そのクラスのオブジェクトを何個引数として受け取るかを示している。表を見ると、引数の数が多いほど実行時間が増えているのが分かる。これは引数の数の分、クラス情報からの文字列による検索回数が増えるからである。特殊化後でもその傾向は変わらないが、大きく実行時間を減らすことに成功した。

6 終わりに

本稿ではプログラムの実行時にプロファイルをとり、その情報を基にリフレクション処理を直接参照による処理に動的に変換する手法を提案した。これにより、リフレクションの柔軟性を保持しつつ、実行コストを削減することができる。

参考文献

[1] BCEL, http://jakarta.apache.org/bcel/

[2] Class Construction Kit, http://bcel.sourceforge.net/cck.html