

特殊なバイトコードでメソッドの呼出しを大幅に効率化。その仕組みを理解することで、JVMがコードを実行する方法を解き明かす

本

まずは、簡単なJavaコードを見てみます。

Javaコンパイラがこのコードから生成したバイトコードを確認するためには、`-c`スイッチを付けて`javap`ツールを使用します。実行すると、逆コンパイルされたコードが表示されます。

```
0: invokestatic #2 // Method
      java/lang/System.currentTimeMillis:()J
3: lstore_1
4: new #3 // class java/util/HashMap
7: dup
```

写真: JOHN BLYTHE

```
8: invokespecial #4 // Method java/util/HashMap."<init>":()V
11: astore_3
12: aload_3
13: ldc #5 // String now
15: ldc #6 // String bar
17: invokevirtual #7 // Method java/util/HashMap.put:
    (Ljava/lang/Object; Ljava/lang/Object;)
    Ljava/lang/Object;
20: pop
21: aload_3
22: astore 4
24: aload 4
26: ldc #8 // String foo
28: ldc #9 // String baz
30: invokeinterface #10, 3 // InterfaceMethod java/util/Map.put:
    (Ljava/lang/Object;Ljava/lang/Object;)
    Ljava/lang/Object;
35: pop
```

[編集注：インデントされた行は、前の行の続きです。] 実際のJavaのメソッド呼出しは、何種類かある、`invoke*`という形式のバイトコードの1つに変換されます。JVMレベルのコードを見るのは初めてだというJavaプログラマーの方は、このことを知って驚かれるかもしれません。

それでは、逆コンパイルされたコードの最初の部分を詳しく見てみます。

```
0: invokestatic #2 // Method java/lang/System.currentTimeMillis():J
3: lstore_1
```

`System.currentTimeMillis()` メソッドの静的呼出しは、`invokestatic` 命令コードに変換され、バイトコードの0の位置に格納されています。このメソッドにはパラメータがないため、呼出し前に評価スタックにロードするものではありません。



次に、バイト・ストリームに2つのバイト00 02が登場します。この2つは組み合わされて16ビット数値（この場合は#2）となり、クラス・ファイル内の表（この表を定数プールと呼びます）に対するオフセットとして使用されます。定数プールのインデックスはすべて16ビットであるため、命令コードがプール内のエントリを参照する場合は常に、エントリのオフセットをエンコードした連続する2バイトが必要です。

便利なことに、逆コンパイラはオフセット#2に対応するメソッドが何かを示すコメントを追加します。今回の例では、予想どおり`System.currentTimeMillis()`メソッドとなっています。javapでは、逆コンパイルされた出力の中に、呼び出されたメソッドの名前、メソッドが受け取るパラメータの型（括弧内）、メソッドの戻り型が表示されます。

呼出し先から戻る際には、呼出しの結果がスタックに配置されます。オフセット3の位置には、引数のない単独の命令コード`lstore_1`があります。この処理は、戻り値をlong型のローカル変数に保存するものです。

もちろん、人間が読めば、この値は二度と使われないことを理解できます。しかし、Javaコンパイラにおける設計上の目標の1つは、論理的に意味をなすかどうかにかかわらず、Javaソース・コードの内容をできる限り忠実に表現する点にあります。そのため、`System.currentTimeMillis()`の戻り値は、この後のプログラムで使われることなくても保存されます。

それでは、逆コンパイルされたコードの次の部分を見てみます。

```
4: new #3 // class java/util/HashMap
7: dup
8: invokespecial #4 // Method java/util/HashMap."<init>":()V
11: astore_3
12: aload_3
13: ldc #5 // String now
15: ldc #6 // String bar
17: invokevirtual #7 // Method java/util/HashMap.put:
    (Ljava/lang/Object;Ljava/lang/Object;)
    Ljava/lang/Object;
20: pop
```





ドはinvokeinterfaceですが、実際に呼び出されるメソッドは同じです。ここでも、命令35のpopでput()の戻り値は破棄されています。

ここまでの部分で、Javaコンパイラは呼出しの状況に応じてinvokestatic、invokevirtual、invokeinterfaceを生成し分けていることがわかりいただけたかと思います。

### メソッド呼出しに使われるJVMバイトコード

それでは、メソッド呼出しに使われる可能性がある5つのJVMバイトコードすべてについて見ていきます（表1参照）。いずれのケースでも、バイトコードb0とb1が組み合わされて定数プールのオフセットc1となります。

| 命令コード名          | 引数          | 説明                                           |
|-----------------|-------------|----------------------------------------------|
| invokevirtual   | b0 b1       | CP#C1に見つかったメソッドを仮想ディスパッチで呼び出す                |
| invokespecial   | b0 b1       | CP#C1に見つかったメソッドを「特殊」な（正確な）ディスパッチで呼び出す        |
| invokeinterface | b0 b1 x0 00 | CP#C1に見つかったインタフェース・メソッドをインタフェース・オフセット検索で呼び出す |
| invokestatic    | b0 b1       | CP#C1に見つかった静的メソッドを呼び出す                       |
| invokedynamic   | b0 b1 00 00 | 呼び出すメソッドを動的に検索して呼び出す                         |

表1:メソッド呼出しに使われるJVMバイトコード

実際にJavaコードを書いて生成されたJavaクラスをjavapで逆アセンブルし、それぞれの形式のバイトコードがどのような状況で生成されるのかを確認してみると、よい演習になるでしょう。

もっとも一般的なメソッド呼出しは、仮想ディスパッチを示すinvokevirtualです。仮想ディスパッチという用語は、呼び出されるメソッドが実行時に完全に決定されることを意味しています。これを理解するためには、実行中のアプリケーション内に存在する各クラスに対して、その型に対応するメタデータを格納するメモリ領域がJVM内に確保されていることを知る必要があります。この領域は、（少なくとも、Java HotSpot VMでは）klassと呼ばれており、JVMでの型情報の表現と考えることができます。

Java 7以前では、klassメタデータは、permgenと呼ばれるJavaヒープに格納されていました。Javaヒープ内の





Java 7では、これらのクラスのvtableは図2のようなレイアウトになります。なお、この図はJava 7のpermgen内のレイアウトを示しているため、参照先はklassOopとなっています。そのため、2ワードのオブジェクト・ヘッダー（図では、mおよびkkで示されています）が存在します。前述のとおり、Java 8とJava 9ではこのエントリは存在しませんが、その点を除けば、同じ図が適用されます。

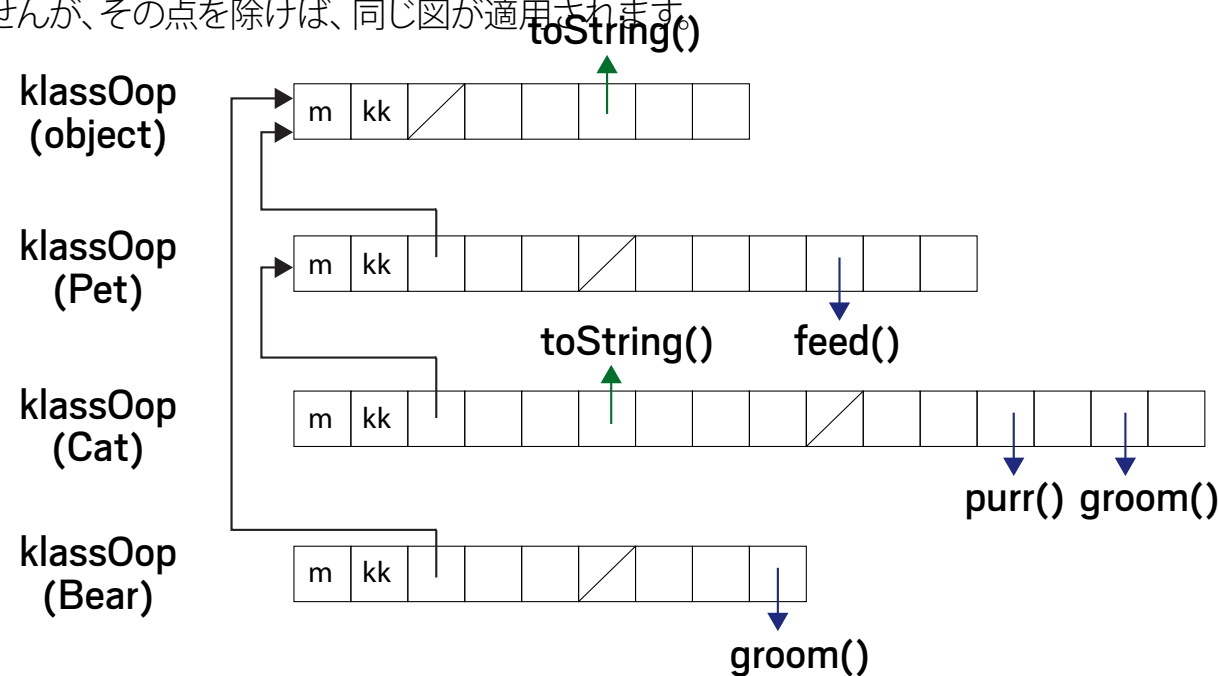


図2: 図1に示したクラスのvtableの構造

Cat::feedを呼び出す場合を考えてみます。JVMは、Catクラスでオーバーライドされたメソッドを見つけられないため、Petのklassへのポインタをたどります。このklassにはfeed()の実装が存在するため、そのコードが呼び出されることになります。このvtable構造がうまく動作するのは、Javaがクラスの単一継承しか実装していないためです。つまり、どの型にも、直接的なスーパークラスは1つしかありません（スーパークラスのないオブジェクトは例外です）。

invokeinterfaceが使われる場合は、もう少し複雑になります。たとえば、すべてのFurryの実装でgroom()メソッドが必ずしもvtableの同じ場所にあるとは限りません。Cat::groomとBear::groomでオフセットが異なるのは、それぞれのクラスの継承階層が異なるためです。そのため、コンパイル時にインタフェース型しかわからないオブジェクトのメソッドを呼び出す場合、追加でルックアップを行う必要があります。

インタフェースの呼出しの際に、追加のルックアップ作業が若干必要にはなりますが、インタフェースの使用を



避けて微少な最適化を試みるべきではないという点に注意してください。JVMにはJITコンパイラが搭載されているため、この2つのケースに実質的なパフォーマンスの差はないことを覚えておいてください。

## メソッド呼出しの例

別の例を見えます。次のコードについて考えてみます。

```
Cat tom = new Cat();  
Bear pooh = new Bear();  
Furry f;
```

```
tom.groom();  
pooh.groom();
```

```
f = tom;  
f.groom();
```

```
f = pooh;  
f.groom();
```

このコードから、次のバイトコードが生成されます。

```
Code:  
0: new      #2      // class scratch/Cat  
3: dup  
4: invokespecial #3      // Method scratch/Cat."<init>":()V  
7: astore_1  
8: new      #4      // class scratch/Bear  
11: dup  
12: invokespecial #5      // Method scratch/Bear."<init>":()V  
15: astore_2  
16: aload_1
```





呼び出されることになります。これは、Javaのオブジェクト指向のルールに違反します（厳密に言えば、リスコフの置換原則に違反することになります）。

そのため、finalメソッドの呼出しはinvokevirtual命令にコンパイルされる必要があります。実際のJavaのHotSpot VMは、finalメソッドを検出して非常に効率よく実行できるように最適化されています。

## まとめ

ここまでで、JVMがサポートする5つの呼出し命令のうち4つを説明してきました。残るinvokedynamicは、これだけで1本の記事が必要になるほど奥深く興味深いテーマです。2回シリーズの2回目となる次の記事では、invokedynamicとそれに関連する題材を扱いたいと思います。</article>

---

**Ben Evans** (@kittylst) : Java ChampionおよびjClarityの創業者/技術フェローであり、London Java Community (LJC) 主催者。Java SE/EE Executive Committeeのメンバーでもある。近刊の『Optimizing Java』(O'Reilly) を含む4冊の本を執筆している。

learn more

[javap](#)  
[JavaのJVM命令セット](#)

