パフォーマンスの目: 例外の例外

本当のコストを理解する

Jack Shirazi
Director
JavaPerformanceTuning.com

Kirk Pepperdine
CTO
JavaPerformanceTuning.com

2004年 2月 10日

Javaパフォーマンス狂、Jack ShiraziとKirk Pepperdine（JavaPerformanceTuning.com のそれぞれディレクターとCTO）は開発者を悩ませるパフォーマンス問題は何なのかと、インターネット中の議論を追跡しています。今回はJavaRanchに立ち寄り、例外に関して起きている井戸端会議での話題に対して、裏の裏を解説しながら反論を試みます。

このシリーズ最初の記事で、例外をスローするコスト（代償）について解説しました。今月は別の角度からこの問題を再度とりあげます。つまり、JVMはスローされた例外をどのように扱うのか。最適化例外コーディングは最適化と言うには未熟なのか、逆にベスト・プラクティスなのかという問題です。

コーディングの分かれ道：こんな風、それともあんな風？

パフォーマンスを話題にしているディスカッション・グループを訪れると、いつもこんな質問で一杯になっています。「他のみんなと同じ流儀で、こんな風にコーディングすべきだろうか、それともももっとパフォーマンスを上げるために、あんな風にコーディングすべきだろうか」。伝統的な知恵の教えるところでは、やたらに最適化するよりも、パフォーマンスを計測した結果、最適化が必要だと分かるまではベスト・プラクティスを適用すべきだということになります。ところが現実には私たちは、１行のコードを書く度に、パフォーマンスに影響するような決断をしているのです。

JavaRanchでの議論の一つは、型の安全性を確実にするための方法として、相反する2つの方法を検証しています。一つは例外をスローするもの、もう一つは`instanceof`を使う方法です。そして質問を投げかけています。「どちらの手法がより良いのか？」リスト1とリスト2はこの2つの方法を示しています。

リスト1. 分岐にinstanceofを使う

```
Listing 1: using instanceof to branch
      for (int i = 0; i < max; i++)
      {
         Object obj = myVector.elementAt(i);
         if (obj instanceof MySpecialClass)
         {
            // do this
         }
      }
```

リスト2. 分岐に例外をスローする

```
      for (int i = 0; i < max; i++) {
        try {
          MySpecialClass myClass = (MySpecialClass)myVector.elementAt(i);
          // do this
        } catch (ClassCastException cce) {
          continue; // for loop
        }
      }
```

こういう質問をする際に陥りがちな問題は、質問が簡単な例に要約されてしまっているために細かなニュアンスが失われてしまうことです。十分な内容説明をせずに質問をしてしまうと、延々と続く混乱した議論を引き起こすことになります。これは質問を読んで回答する人それぞれが、問題について独自のニュアンスを持ち込んでしまうからです。持ち込まれるニュアンスで内容が深まることもありますが、元々提示された問題から議論がそれてしまいがちなものです。それを念頭に置いた上で、このスレッドで見つけたメッセージの痕跡から一定の真実をふるい分けることができるかどうか、見ることにしましょう。

例外の特性

開発者が例外について最初に言うことは、それが高くつく、ということでしょう。なぜ高くつくのかをたどっていくと、一番普通の答えとしては、例外スタックの現在の状態をキャプチャーしなければならないから、と言うことになるでしょう。確かにこれは例外を使うコストとしての大きな部分を占めていますが、例外の特徴をいくつか挙げてみると、これが問題のごく一部を言っているにすぎないことが分かります。下に挙げた項目は例外の特徴の一部です。

- スローできる
- キャッチできる
- プログラム的に生成できる
- JVMによって生成できる
- ファーストクラス・オブジェクトとして表される
- 3から始まる継承の深さを持つ
- `String`（と1.4からの`StackTraceElements`）からなる
- ネイティブ・メソッド`fillInStackTrace()`に依存する

例外とその他あらゆるオブジェクトとの違いは、例外はスローとキャッチができると言うことです。例外がスローされることでトリガされる、一連のイベントを調査してみることにしましょう。

例外を扱うコスト

例外をスローするために、JVMは`athrow`バイトコード命令を発行します。`athrow`命令によってJVMは例外オブジェクトを実行スタックから無くします。次に`athrow`命令は現在の実行スタック・フレームを検索して、そのクラス、またはそのスーパークラス中にあって例外を処理する最初の`catch`文を探し出します。もし現在のスタック・フレームに`catch block`が見つからない場合は、現在のスタックフレームは解放され、次のスタック・フレームのコンテキストの中で例外が再度スローされます。条件に合致する`catch`文を持つスタック・フレームが見つかるまで、または実行スタックの底に達するまで、これが繰り返されます。最終的には、適切な`catch`ブロックが見つからない場合はすべてのスタックフレームは解放され、`ThreadGroup`オブジェクトに例外を処理する機会が与えられた後にスレッドは終了します（`ThreadGroup.uncaughtException`を見てください）。適切な`catch`ブロックが見つかると、プログラム・カウンターはそのブロックの1行目のコードにリセットされます。

この説明から、スローされた例外の処理は非常に高くつく代物だということが分かります。上に挙げた、例外の特徴をもう一度見てください。注意して欲しいのは、JVMが「自発的に」例外を生成することができるという事実を別にすると、その他のコストについては、ファーストクラス・オブジェクトのライフサイクル期間中に発生するコストと変わりはないということです。

ファーストクラス・オブジェクトとしての例外のコスト

リスト2 をもう一度見てください。キャストが失敗した場合にのみ例外がスローされています。JVMはどのようにこれを処理するのでしょうか。型のキャストを実行するためにアプリケーションが要求されるときには、必ず`checkcast`操作が発行されます。この操作は、スタックの最上部にある引数タイプが期待されているものと同じであることを確認する以外、何もしません。同じでない場合には、この操作が`ClassCastException`をスローします。

それほど手荒でない方法でタイプ・チェックを行うには、リスト1 に示すように`instanceof`演算子を使います。`checkcast`と`instanceof`との違いとして、`instanceof`の方はスタックの最上部に、失敗か成功かを示すものとして0か1を残すことです。

`instanceof`演算子は、成功か失敗かを判断するための非常に厳格な規則に従います。この規則は変数リファレンスが`null`か、配列か、インターフェースか、それとも単なるクラスなのかを判定するためのものです。一旦変数のタイプが判明すれば、一致が見つかるか階層構造が終わりになるまで、反対側にあって適合するオペランドの階層構造を検索する必要があります。配列の場合には、下にある要素の型はこれと同じ検索を受ける必要があります。

このコストに加え、一旦`instanceof`を適用すると、通常はそのオブジェクトを次に続くコードにキャストすることになります。次に続くキャストで`checkcast`バイトコードが実行されるようになります。ですから、キャストが動作するかどうか判定するために使った論理を繰り返す必要があります（JVMがこの余分のチェックを最適化していないという前提ですが）。とは言っても`instanceof`演算子を使うからといって新しいオブジェクトを生成する必要は無いので、メモリの面からも、実行リソースの面からも例外を生成して処理するよりはずっとコストのかからない操作だと言えます。ですから、元々の質問に対する答えは明白のようです。でも、本当にそうでしょうか。

隠れた危険性

`ClassCastException`をキャッチするのにも隠れた危険性があります。つまり、「これをしなさい」というコメントを置き換えるコードがスローする`ClassCastException`をキャプチャーしてしまうかもしれないということです。このコードが何かの操作の最中に`ClassCastException`をスローするのであれば、それをキャッチし、キャストから`MySpecialClass`にスローされたものであるとみなし、黙って無視します。ただ、その結果アプリケーションを矛盾した状態に置いてしまうことになるかもしれません。

動的チューニング

ここまでは、2つのコーディング・スタイルで一度だけ実行する場合のコストについて見積もっただけでした。次にはどちらのコーディング・スタイルを使うべきかを決められるように、コードが実行される条件を理解する必要があるのです。

実際にかかるコストが感覚的に分かるように、集合に対して繰り返すケースを考えてみてください。それが似たようなオブジェクトのセットを含む集合であり、ポーリングされたオブジェクトすべてに対して`instanceof`を実行するとすると、この手順では実行時間全体としては不必要なコストをかけていることになります。一方、その集合が異なったオブジェクトのセットを含むものであれば、例外の嵐を呼んでしまうコストをかけるよりは`instanceof`操作を使ったほうが得策と言えるでしょう。

このシナリオがベスト・プラクティスに向けての最後の手がかりとなります。つまり、例外は例外的な状況にのみ限定すべきだということです。例外的な状況では例外を使うことで理想的なパフォーマンスを得ることができます。例外的でない状況では、チェックを使い、例外をスローしないようにすることで理想的なパフォーマンスが得られるのです。

最後に

以上から、ベスト・プラクティス（ここでは、例外は例外的な状況に限定する、ということ）を守れば、より高いパフォーマンスが得られることが分かりました。一体何がベスト・プラクティスなのか、そのベスト・プラクティスのパフォーマンスとして、どんなことを考えるべきなのか、またその代替手段は何か？そうした判断をするために、私たちは時々、この記事で説明したような徹底的な検討をする必要があります。ここでは、必要もないのにコードを最適化していないか心配するのではなく、最も良いコーディング習慣（best coding practice）にたどり着くようにしました。それがパフォーマンスを別にしても適切なものであり、しかも結果的にパフォーマンスも最適なもの・・・つまり両方ともに良い結果が得られたというわけです。

著者について

Jack Shirazi

Jack ShiraziはJavaPerformanceTuning.comのディレクターであり、Java Performance Tuning, 2nd Edition（O'Reilly刊）の著者でもあります。

Kirk Pepperdine

Kirk PepperdineはJava Performance Tuning.comのCTO（Chief Technical Officer）であり、過去15年間、オブジェクト技術やパフォーマンス調整に注力してきました。Ant Developer's Handbook（MacMillan刊）の共著者でもあります。