Javaコードの診断: プラットフォーム依存を引き起こす"犯人"

プラットフォーム依存のバグ・パターンにスポットライトを当てる

Eric Allen

2001年 5月 01日

「Write once, run anywhere (一度書けば、どこででも実行できる)」。これがJava言語の掲げる約束ですが、時おりその約束を果たせないことがあります。JVMが前例のないほどのクロス・プラットフォームの相互運用性を実現していることは確かですが、仕様と実装の両方における小さな欠陥により、しばしば、多数のプラットフォームでプログラムが正しく動作しないことがあります。この記事では、Eric Allen氏が、気を付けるべきJavaプログラミングのプラットフォーム依存の側面について説明します。たとえば、末尾再帰呼び出しや、組み込みのベンダー、バージョン、およびオペレーティング・システムへの依存です。さらに、この種の依存性を回避するいくつかの方法も紹介します。

Java言語でプログラミングすることの大きな利点の1つは、驚くほどのプラットフォーム独立性が提供される点です。ターゲット・プラットフォームのそれぞれについて別々のビルドを作成する代わりに、単にバイト・コードをコンパイルして、JVMのある任意のプラットフォームに配布できます。あるいは、少なくとも、そのように話が進むことになっています。

しかし、実際には、話はそれほど単純ではありません。Javaプログラミングは、複数プラットフォームの場合の開発者の時間を計り知れないほど節約できるとはいえ、異なるJVMバージョン間には多くの互換性の障害が存在します。そのような障害の中には、簡単に見つけ出して訂正できるものもあります。たとえば、パス名を構成するときにプラットフォーム固有のセパレーターを使用すること、などです。しかし、回避するのが困難または不可能な障害もあるのです。

したがって、説明できないプログラムの異常の中には、特定のJVMのバグが原因になっているものがあり得ることも頭に置いておくことが重要です。

ベンダー依存のバグ

JVMに存在するプラットフォーム依存の潜在的なバグをいくつかをご覧になりたければ、SunのJavaバグ・データベース (参考文献を参照) を垣間見るだけで十分でしょう。ここに列挙されているバグの多くは、特定のプラットフォーム上のJVMのにみ当てはまる、実装上の バグです。たまたまそのプラットフォーム上で開発しているのでなければ、自分のプログラムがそのバグにつまずくということを知ることさえないでしょう。

しかし、プラットフォームに依存するバグのすべてが、JVMの実装面のバグに起因するとは限りません。プラットフォームに依存する重大なバグが、JVMの仕様のそのものから引き起こされることがあるのです。JVMの詳細が仕様レベルで決定されないままになっていると、各JVMでベンダーに依存する動作を生み出すことになってしまいます。

たとえば、以前の「Javaコードのパフォーマンスを向上させる」(2001年5月) で見たとおり、JVM仕様では、末尾再帰呼び出し の最適化を必須とは定めていません。末尾再帰呼び出しとは、再帰的なメソッド呼び出しで、1つのメソッド内でいちばん最後の操作として発生する呼び出しのことです。もっと一般的には、任意のメソッド呼び出し (再帰的であっても、なくても) のうち、1つのメソッド内で最後に発生する呼び出しのことを、末尾呼び出し といいます。たとえば、次の単純なコードについて考えてみましょう。

リスト1. 末尾再帰による階乗の計算

```
public class Math {
  public int factorial(int n) {
    return _factorial(n, 1);
  }
  private int _factorial(int n, int result) {
    if (n <= 0) {
      return result;
    }
    else {
      return _factorial(n - 1, n * result);
    }
  }
}
```

この例では、publicのfactorial メソッドと、privateのヘルパー メソッド`_factorial` には、どちらも末尾呼び出しが含まれています。factorial には、`_factorial` に対する末尾呼び出しが含まれており、`_factorial` には、それ自身に対する再帰的な末尾呼び出しが含まれています。

これが、階乗 を記述する方法としては複雑すぎると思えたとしても、不思議ではありません。次のような、もっと自然な形で記述しないのは、なぜなのでしょうか。

リスト2. 純粋な再帰による階乗の計算

```
public class Math {
  int factorial(int n) {
    if (n <= 0) {
      return 1;
    }
    else {
      return n * factorial(n-1);
    }
  }
}
```

その答えは、末尾呼び出しでは非常に強力な最適化が許可されているから、ということです。末尾呼び出しでは、呼び出した側の のメソッドのために構築されたスタック・フレームを、呼び出された側の メソッドのスタック・フレームと交換できます。これにより、実行時のスタックの深さが劇的に減少し、スタック・オーバーフローを回避できます (特に、末尾呼び出しが再帰的である場合。リスト2の`_factorial` の場合がそう)。

一部のJVMはこの最適化を実装していますが、これを実装していないJVMもあります。その結果、同じプログラムが、一部のプラットフォームではスタック・オーバーフローを起こし、別のプラットフォームでは起こさない、ということになるわけです。もしこの最適化を静的に実行できるのであれば、末尾呼び出しを最適化した形にバイトコードをコンパイルするだけで、プラットフォームとは独立してその最適化の恩恵を受けることができます。しかし、残念なことに、上記で参照した記事で説明したとおり、この最適化を静的に実行することはできません。

バージョン依存のバグ

末尾呼び出しに起因するプラットフォーム依存は、JVM仕様そのものが原因です。しかし、より多く見受けれるプラットフォーム依存の原因は、JVMの実装におけるバグにあります。Swingの場合、そのようなバグが広く見られます。

たとえば、JDK 1.4の`JOptionPane`コンポーネントには、それと関連したバグがあります。`JOptionPane`で、ブランク行の直後の行にテキストを追加し、下矢印キーを押しても、何も起こりません。次のようにして、自分で試してみてください。

- 新しい`JOptionPane`を開きます。
- `OptionPane`で、Enterキーを2回押します。
- 「test」と入力します。
- 上矢印キーを押します。
- 下矢印キーを押します。

このような順序で (あるいは、これと似たような順序で) 操作を行うと、`JOptionPane`がおかしな状態になります。このコンポーネントを使ったプログラムのユーザーがこのバグを見つけると、おそらく、必死にキーボードを叩いてその状態から抜け出そうとするに違いありません。(この状態から回復するのは、それほど難しくありません。たとえば、右矢印キーを押せば大丈夫です。) いったんその状態から抜け出してしまうと、ユーザーはその固まった状態を気にしなくなり、バグとして報告することさえしないでしょう。バグを頻発するソフトウェアが多いので、ユーザーの許容範囲が実質的に広くなっているわけです。

しかし、そこに思わぬ落とし穴があります。このバグは、私がテストしたすべてのプラットフォーム (Windows、Solaris、およびLinux) 用のバージョンのSun JDK 1.4に存在します。したがって、これは、オペレーティング・システムには依存しない、SunのJDKのバグのようです。

この例は、プラットフォーム依存が、単にOS依存ではなく、単にベンダー依存でもないことを示しています。これは、バージョン依存であり、そのような依存性は下位方向(backward)にも、上位方向(forward)にも存在します。

開発チームは、通常、下位互換性を保つことの重要性は認識していますが、多くの場合、その互換性の維持は、後のバージョンに先送りされてしまうことが多々あります。理屈の上では、この期待は正しいかもしれませんが、現実問題としては、この期待は裏切られます。というわけで、Sunはバージョン1.4でパフォーマンスを改善するべく懸命に努力したとはいえ、そのバージョンでSwingにバグを持ち込んでしまったとしても、驚くにはあたりません。

ちなみに、Swingのパフォーマンスに満足していないのは、Sunだけではありませんでした。Eclipseプロジェクトという、高度に統合されたツールを開発するための、堅固で、オープ

ン・ソースの、機能を満載した、商用品質のプラットフォームを提供することを目指して立ち上げられたプロジェクトが、Standard Widget Toolkit (SWT) と呼ばれる、まったく新しいウィジェット・ツールキットを実装しています。SWTは非常に軽量です。それは、Swingとは違って、SWTが作動するプラットフォームに固有のウィンドウ操作システムを活用しているからです (参考文献を参照)。APIは、それが実装されているプラットフォームが違っても同一ですが、ルック・アンド・フィールは完全にプラットフォームに依存しています。したがって、このツールキットに関しては、プラットフォーム依存についての新しい問題があるものと予想されます。

OS依存のバグ

Javaプラットフォームで経験することのある陰に潜んでいるプラットフォーム依存の最後の例として、ファイルを開き、それをエディター・ウィンドウに読み込むコードを記述する場合のことを考えてみましょう。最初は、次のようなコードを記述するかもしれません。

```
FileReader reader = new FileReader(file);
_editorKit.read(reader, tempDoc, 0);
```

`_editorKit.read` の呼び出しにより、ファイルの内容を一時文書に読み込み、後でそれを、開いているファイルのコレクションに追加します。しかし、この2行の後、`reader` を再び参照することはありません。

このコードは、ライス大学のフリーのオープン・ソースJava IDE (参考文献を参照) の初期のバージョンから取りました。さて、クリーンアップ・コード散在バグ・パターンに慣れておられる読者であれば、このコードが、まさにそのパターンの実例であることにお気付きでしょう。

ファイルの内容を読み取るために`FileReader` が構築されますが、その`FileReader` は閉じられていません。もちろん、クリーンアップ・コード散在の他のインスタンスと同じく、そのファイルを再びアクセスしようとしない限り、このバグは何の症状も現しません。しかし、プラットフォームによっては、そのような場合でも、何も症状が現れないこともあるのです。

後ほど、ユーザーがこのファイルを削除しようとしたとしましょう。UNIXでは、開かれているファイルでも削除できるため、閉じられていない`FileReader` の痕跡があっても、何も問題が起きません。しかし、Windows上のユーザーの場合は、開かれているファイルは削除できないため、例外がスローされます。上記のコードのバグは、単体テストの1つが、UNIX上では合格しても、Windows上では失敗したことから、発見されました。問題を診断できた後は、これを修正するのは難しくありません。

```
      FileReader reader = new FileReader(file);
      _editorKit.read(reader, tempDoc, 0);
      reader.close(); // win32 needs readers closed explicitly!
```

クロス・プラットフォームのコストはゼロではない

この記事で取り上げた例が示しているとおり、Java言語は、プラットフォーム依存のバグに対して免疫があるわけではありません。これらのバグは実に多種多様ですが、いろいろな場合にそのいくつかに遭遇することが予想されます。

クロス・プラットフォームのコードを記述するコストは、他の多くの言語に比べてJavaの場合にずっと低くなりますが、決してゼロではありません。最善の提案は、できる限り多くのプラットフォームと、できる限る多くのバージョンのJVMで、単体テストを実行することです。そして、もちろん、バグの起きやすいコードを記述することを避けます。バグの起きやすいコードと、プラットフォーム依存は、最悪の組み合わせです。今月の記事で取り上げた内容を要約すると、次のようになります。

- パターン: ベンダー依存のバグ。
- 症状 : エラーが一部のJVMで起きるが、他のJVMでは起きない。
- 原因 : JVM仕様にある、いくつかの未指定の部分 (たとえば、末尾再帰呼び出しの最適化が必須とはされていない、など)。このタイプの原因は、バージョン依存の バグより発生頻度が低い。
- 治療法と予防策 : 遭遇する問題によって異なる。
- パターン: バージョン依存のバグ。
- 症状: エラーが一部のバージョンのJVMで起きるが、他のバージョンでは起きない。
- 原因: 特定のJVM実装 (たとえば、Swing) におけるバグ。これは、ベンダー依存の バグよりも発生頻度が高い。
- 治療法と予防策: 遭遇する問題によって異なる。
- パターン: OS依存のバグ。
- 症状: エラーが一部のオペレーティング・システムで発生するが、その他のオペレーティング・システムでは発生しない。
- 原因: システムの動作の規則が、オペレーティング・システムごとに異なる (たとえば、UNIXでは、開かれているファイルを削除できるが、Windowsでは削除できない)。
- 治療法と予防策: 遭遇する問題によって異なる。

この記事で説明した後半の2つのバグを識別する手助けをしていただいた、DrJava開発者のBrian Stoler氏とJohn Garvin氏に感謝いたします。

関連トピック

- DrJava は、ライス大学のオープン・ソースのフリーJava IDEで、read-eval-printループを備えています。
- Standard Widget Toolkitの詳細については、Eclipse のWebサイトをお調べください。
- XPの背後にある考え方の要約は、Extreme Programming のWebサイトにアクセスしてください。
- XPについてさらに学ぶことに関心をお持ちの場合は、Roy Miller氏とChris Collins氏による「XPの真髄」(developerWorks、2001年3月) という記事が、この機動的なソフトウェア開発の方法論についての簡潔な概要を提供しています。
- Javaコードおよびライブラリーとシームレスに相互運用できるPythonの実装、Jython をダウンロードできます。
- Eric Allenの 「Javaコードの診断」 の全記事をお読みください。バグ・パターンに関する完全装備の記事になっています。
  - バグ・パターン： Javaプログラムで頻発しがちなバグの分析と修正
  - 「宙ぶらりん複合型」バグ・パターン： ヌル・ポインター例外の最もよくある原因を鎮圧する
  - 「ヌル・フラグ」バグ・パターン： 例外状況を表すフラグとしてヌル・ポインターを使うことを避ける
  - Javaコードのパフォーマンスを向上させる： 末尾再帰変換はアプリケーションの速度を向上させる可能性はあるが、すべてのJVMで可能な操作ではない
  - 虚偽の実装というバグ・パターン: 第2回： 表明とユニット・テスト - バグを除去するための実行可能なドキュメンテーション -
  - 「みなし子スレッド」バグ・パターン： マスター・スレッドが自滅し、その他のスレッドが生き残っていると、どうなるか?
  - 拡張可能アプリケーションの設計 第1回： ブラック・ボックス、オープン・ボックス、またはガラス・ボックス: どんな場合にどれがふさわしいか?
  - 深さ優先visitorと、破綻したディスパッチ： このVisitorパターンの変形を使えば、コードをより簡潔にできます
  - replによる対話式評価： ソフトウェアを効率的かつ対話式に診断するためのテクニックとツール
  - 「付け足し初期化コード」バグ・パターン： 引数の足りないコンストラクターを避ければ、このバグを撃退できる
- Javaに関するその他の参考文献は、developerWorks のJavaテクノロジー・ゾーンでご覧いただけます。