Javaの理論と実践: バグを確実につぶす

FindBugsのような検査ツールが一般的なコード間違いに対する第２の防御線を提供

Brian Goetz
Principal Consultant
Quiotix

2004年 6月 29日

プログラミング・スタイルに関する助言の大部分は、高品質で維持管理容易なコードを書くことを目的としています。一番バグを修正しやすいのはバグを作ってしまう前なので、防止策を説くこの助言は当を得たものです。ところが残念なことに、防止策は常に十分とは限りません。高品質のコードを書くための良いツールはいくつかありますが、既存のコードを解析、メンテし、また品質改善するようなツールはほとんどありません。Chris Grandstaffが先に FindBugsの紹介記事を書いていますが、今月はコラムニストのBrian Goetzがその記事を基礎に、このコラムで説明してきた設計原則にコードが合致するかどうかの静的解析にFindBugsを使う方法を説明します。

このシリーズの他の記事を見る

スレッド・セーフなクラスを書くのは困難なものですが、既存クラスがスレッド・セーフかどうかを解析するのはスレッド・セーフであり続けるように改善するのと同じくらい、さらに困難なものです。クラスがどのように動作するのか（または動作することになっているのか）、という知識の大部分、つまり開発者の頭の中では明確であった設計ノートや注記、文書資料の形では書き残されていなかった暗黙の仮定、不変条件、それに想定されるユースケースなどは、クラスを書き終わるのとほぼ同時に消え去ってしまいます。新しいコードを書くよりも、既存コードを扱う方がより困難なものです。

必要なもの：より良いコード監査ツール

当然のことですが、コードを書いている時がコードを高品質なものとする最善の時期と言えます。この時であれば何がどのように動作するのかを最も良く理解しているからです。高品質のコードを書く方法については山ほどの助言があります（ただこのコラムを読めばよいのです！）が、全てを最初から書いたり、書くための時間を好きなだけ十分に取ったりという贅沢はなかなかできません。ではそういう時にはどうすべきなのでしょう。開発者は書き直しを要求しがちなものです。（結局、新しいコードを書いた方が誰か他の人が書いた（あるいは自分自身で書い

たバグだらけの）コードを修正するのよりも楽しいものです。）ただし書き直しもやはり贅沢であり、今日の既知の問題を、明日の未知の問題と交換しているにすぎません。必要なのは既存のコードベースを解析・監査し、コード監査とバグ追跡で開発者を補助するような良い解析ツールなのです。

自動コード検査・監査ツールはFindBugsの導入で最近非常に良くなったと言えると思います。これまでも自動ツールはありましたが、大部分は、プログラムが正しいものだと証明するような非常に困難な問題が対象であったり、またコード・フォーマットや名前付け規則のような表面的な問題に注目したものであったり、あるいは良くてもself-assignment、未使用フィールドのような単純なバグ・パターンを検出するものか、privateだとかprotected宣言される可能性のある未使用メソッド・パラメーターやパブリック・メソッドを検出するようなものでした。ところがFindBugsは違います。FindBugsはバイトコード解析や豊富なバグ・パターン・チェック機能を持ち、コード中にある一般的なバグを検出するのです。また、（意図したであれ意図しないものであれ）良い設計原則から外れている点を検出する手助けもしてくれるのです。（FindBugsの紹介についてはChris Grindstaffによる記事「第1回: コード品質を改善する」と「第2回: カスタムのチェック機能を書く」を見てください。）

設計に関する助言とバグ・パターン

バグのパターンにはそれぞれに対応して、そのバグ・パターンを避けるための設計上の助言要素があるものです。ですからFindBugsがバグ・パターンのチェック機構として、一連の設計原則に合致するかどうかを調べるための監査ツール機能を持っているのも当然なのです。Javaの理論と実践の多くの記事は、設計に関する様々な助言の内、特定の要素（あるいはそれに対応するバグのパターン）に注目したものです。今回の記事では、そうした助言を既存のコードベースに適用する上で、FindBugsがどのように役立つかを説明します。過去の助言を復習しながら、助言を守り損なった時にFindBugsがどのように問題を検出してくれるかを見て行きましょう。

例外に関する議論

「例外をめぐる議論」でチェック済み例外に関して挙げられた問題点の一つが、例外があまりにも簡単に「取り落とされて」しまいがちである、ということです。つまりリスト1に挙げるように、例外を捉えながらその例外を修正するようなアクションも取らず、別の例外も投げないということが起きるのです。この取り落としがよく起きるのはプロトタイプ化の際で、単純にプログラムをコンパイルするために、（後でそこに何らかのエラー処理を入れ込むつもりで）空の`catch`ブロックをコード化してしまう時です。一部の人はこうした場合が頻繁に発生することをとりあげ、Javaの言語設計で採り入れた例外処理手法がうまく動作しない証拠だと見なしていますが、私は単に正しいツールを使わなかったためだとしか思いません。FindBugsを使えば、こうした空の`catch`ブロックを簡単に検出してフラグを立ててくれるのです。もしあなたが意図的に例外を無視しようとするのであれば、単に処理を忘れたのではなく意図的に無視していることが読んでいる人にも分かるように、記述的なコメントを追加しておいた方が無難です。

リスト1. 例外を取り落とす

```
try {
  mumbleFoo();
}
catch (MumbleFooException e) {
}
```

ハッシュの徹底

私は「ハッシュの徹底」の中で、`Object.equals()`と`Object.hashCode()`を正しくオーバーライドする基本ルールの概要、特に平等なオブジェクトは（`equals()`に従って）平等な`hashCode()`値を持つ必要があることを説明しました。この規則は知ってさえいれば守るのは簡単です（また一部のIDEには一貫した方法で両者を定義するウィザードがあります）。ただし一方のメソッドをオーバーライドし、もう一方をオーバーライドし忘れてしまうと、（そこにあるコードに問題があるのではなく、存在していないコードに問題があるので）このバグを検査で検出するのは非常に困難なものになります。

FindBugsにはこの問題が起きる場合の多くをチェックする機能があり、例えば`equals()`をオーバーライドして`hashCode()`はオーバーライドしていないとか、`hashCode()`をオーバーライドして`equals()`はオーバーライドしていない、などをチェックすることができます。こうしたチェックはFindBugsのチェック機能の中では最も単純なもので、クラスにある一連のメソッド・シグニチャーを調べ、`equals()`と`hashCode()`の両方がオーバーライドされているかどうかを判定するだけです。　また、間違って`Object`以外の引数タイプで`equals()`を定義してしまう可能性もあります。この構成は有効ではありますが、期待するような動作をしません。Covariant Equalsチェック機能はそうした疑わしいオーバーライドとして、次のようなものを検出するのです。

```
public void boolean equals(Foo other) { ... }
```

このチェック機能に関連したものがConfusing Method Namesチェック機能です。この機能は`hashcode()`や`tostring()`のような名前を持つメソッドや、同じ名前でどこかが大文字になっているかだけが異なるメソッドを持つクラス、それにスーパークラス・コンストラクターと同じように名前がついたメソッドを持つクラスなどでトリガーされます。Java言語の仕様ではこうしたメソッド名は全て有効ですが、こうしたメソッド名は意図した通りのものではない可能性があります。同様にSerializationチェック機能はフィールド`serialVersionUID`が`final`ではない場合、または`long`ではない場合、`static`ではない場合にトリガーされます。

ファイナライザーは厄介者

私は「ガーベジコレクションとパフォーマンス」の中で、ファイナライザーは極力使わないようにと皆さんを説得したつもりです。ファイナライザーにはパフォーマンス・コストの点で大きな問題があり、しかも予測可能な期間で実行するという保証がないのです（あるいは全く実行しないかも知れません）。とは言ってもファイナライザーを使わざるを得ない時もあり、そういう時にはいくつかの間違いをおかしがちです。ファイナライザーを使う必要がある時には普通、ファイナライザーはリスト2に示すような構造になっているはずです。

リスト2. ファイナライザーの適切な定義

```
protected void finalize() { try {
      doStuff();
    }
    finally { super.finalize();
    }
  }
```

FindBugsは下記に挙げるような、ファイナライザーの構成として疑わしいものをいくつか検出します。

- 空のファイナライザー（スーパークラス・ファイナライザーの効果をなくしてしまうもの）
- 何もしないファイナライザー（`super.finalize()`を呼ぶだけで、ある種のランタイム最適化を抑制する可能性のあるもの）
- 明示的なファイナライザー起動（ユーザー・コードから`finalize()`を呼ぶ）
- パブリック・ファイナライザー（ファイナライザーは`protected`として宣言すべきものです）
- `super.finalize()`を呼ばないファイナライザー

こうしたバグ・パターンの例をリスト3に示します。

リスト3. ファイナライザーの間違いとして一般的なもの

```
  // negates effect of superclass finalizer
protected void finalize() { }
// fails to call superclass finalize method
protected void finalize() { doSomething(); }
// useless (or worse) finalizer
protected void finalize() { super.finalize(); }
// public finalizer
public void finalize() { try { doSomething(); } finally { super.finalize() } }
```

同じ「ガーベジコレクションとパフォーマンス」の中で私は、ガーベジ・コレクションにある、もう一つの危険性、つまり`System.gc()`への明示的なコールについても説明しました。そうした明示的なコールはほとんど常に、ガーベジ・コレクターを「補助」したり、「騙したり」しようとする誤った試みであり、多くの場合、パフォーマンスを向上させるよりも低下させる結果となっています。FindBugsは`System.gc()`への明示的なコールを検出し、フラグを立てます（SunのJVMでは、`-XX:+DisableExplicitGC`起動オプションでも、明示的なガーベジ・コレクションを使用不可にすることができます）。

安全な構築のテクニック

私は「安全な構築のテクニック」の中で、オブジェクトの参照がそのコンストラクターを漏洩するのを許すと、深刻な問題をいくつか引き起こすことを示しました。それ以来、`this`参照がコンストラクターを漏らすのを許すことによる危険性はさらに深刻なものになっています。新しいJavaメモリ・モデルでは（JDK 1.5に実装されているJSR 133で規定されている通り）、オブジェクトの参照がコンストラクターの漏洩を許している場合には、初期化安全性の保証を否定しているのです。

オブジェクトの参照は直接的、間接的ないくつかの方法で、コンストラクターを漏洩することができてしまいます。`this`参照を静的変数やデータ構造に保存するのは明らかにまずいのですが、参照がコンストラクターを漏洩できてしまう、もっと微妙な方法もあるのです。例えば参照を非静的な内部クラスに公開するとか、コンストラクター内部からスレッドを開始する（これはほとんど必然的に、参照を新しいスレッドに公開することを意味します）などです。FindBugsにはスレッドがコンストラクターから開始しているインスタンスを検出するチェック機能があります。現在はこうした危険性の全てを検出できるわけではありませんが、将来のバージョンでは、こうした他の初期化安全パターンのいくつかをチェックする機能が含まれる可能性があります。

メモリ・モデルを尊重する

「Javaメモリ・モデルを修正する 第1回」の中では、私は同期化の基本的なルールを説明しました。ある変数を読む時に、別スレッドが最後にその変数に書き込んだ可能性がある時には、また

はある変数に書き込む時に、別スレッドがその変数を読み込む可能性がある時には、同期を行う必要があります。このルールは、特に読み込む時には「忘れて」しまいがちですが、忘れてしまうことでそのプログラムのスレッド・セーフに多くの危険性をはらむことになります。この種類のバグがよく発生するのはクラスをメンテする際です。元々は適切に同期化されていたにもかかわらず、メンテする人がスレッド・セーフのための要求事項を完全に理解しせずに作業するために、バグが発生してしまうのです。

幸いFindBugsにはいくつかのチェック機能があり、同期化が適切になされていないクラスの特定を手助けしてくれるのです。Inconsistent Synchronizationチェック機能はFindBugsが使っているチェック機能の中で、おそらく最も複雑なものと言えるでしょう。これは個々のメソッドだけではなくプログラム全体を解析する必要があり、あるロックがいつ保持されるのかをデータフロー解析を使って判定し、あるクラスがスレッドセーフの保証を与えようとしているのだと試行錯誤法（heuristics）を使って推論するのです。基本的にInconsistent Synchronizationチェック機能は各フィールドに対して、そのフィールドへのアクセスパターンを観察します。もし大部分のアクセスが同期化を伴って行われている場合には、同期化無しのアクセスに対して、潜在的なエラーとしてフラグを立てるのです。同様に、プロパティに対するセッターが同期化されていてゲッターが同期化されていない時にも警告を発します。

一貫性のない同期化の他にも、一般的なスレッド関連の間違いをチェックする機能もFindBugsには含まれています。例えば、２つのロックを保持したモニターでの待ち（これは必ずしもバグではありませんが、デッドロックを引き起こす可能性があります）や二重チェックのロック・イディオム（double-checked locking idiom）の使用、非揮発性フィールドに対する不正な遅延初期化、スレッドで`run()`を開始する代わりに`run()`を呼び出してしまう間違い、コンストラクターから`Thread.start()`を呼び出す間違い、ループの中で`wait()`をラップせずに`wait()`をコールする、などといった間違いの検出です。

可変性か、不変性か?

「可変性か、不変性か?」で、またその他の記事でも、私は不変性の美徳を説いてきましたが、不変オブジェクトは一貫性に欠ける状態になり得ないのです。不変オブジェクトは（キーワード`final`を使うことで不変性が保証されているとすれば）元々スレッド・セーフであり、不変オブジェクトへの参照を、（コピーしたり複製したりする必要もなく）自由に共有し、キャッシュすることができるのです。

キーワード`final`は開発者が不変クラスを作るのを補助する目的でJava言語に含まれました。またキーワード`final`によって、コンパイラーやランタイム環境が、宣言された不変性に基づいて最適化できるようになります。ところがフィールドはfinalであることができても、配列要素はfinalではありえないのです。finalフィールドとprivateフィールドを適切に使用することでオブジェクトを不変にすることはできますが、もしそのオブジェクトの状態が配列を含む場合には、こうした内部配列への参照がクラスのメソッドを漏らさないようにすることが重要です。リスト4に示すクラスは不変であろうとしていますが、呼び出し側が`getStates()`を呼んだ後で状態配列を変更できてしまうので、不変ではありません。（これに関連して、可変クラスが可変配列への参照を返す時にもバグが起こり得ます。呼び出し側がこの配列を使う時までには配列の内容が変わっているかも知れないのです。）このイディオムの問題は一般的には「悪意のコード」に対する脆弱性と見なされていますが、（また多くの開発者は自分のシステムが「信頼できない」クラスはどんなものクラスもロードしないので「悪意のコード」を気にしませんが）、悪意のコー

ドが無い場合でもやはり深刻な問題を引き起こす可能性があるのです。変更できない`List`を返すか、配列を返す場合には、返す前に配列を複製した方がより良いと言えます。FindBugsは（リスト4に示す）`getStates()`にあるような間違いを検出することができます。FindBugs自体は必ずしも`States`クラスが不変である必要があるかどうかは分かりませんが、このゲッターが可変のプライベート配列に対してハンドルを返していることは知っており、それに応じたフラグを立てるのです。

リスト4. 可変配列に誤って参照を返す

```
public class States {
  private final String[] states = { "AL", "AR", "AZ", ... };
  public boolean isState(String stateCandidate) { ... }
  public String[] getStates() { return states; }
}
```

無視して構わないバグはない

FindBugsは真に画期的なツールです。本当のバグを、ほとんど常に検出してくるのです。皆さんはFindBugsが見つけてくるバグのパターンの一部は、（例えば「variable self-assignment」のように）探すまでもないような些細なものだと思われるかも知れません。その考えは間違っています。FindBugsの持つチェック機能はどれをとっても、テスト済みで実稼働中の、専門的な手法で開発されたコードにあるバグを検出してきたのです。あなたのコードの中におかしなバグがいるのであれば、FindBugsのコピーを入手して試してみてください。そうすれば納得、あるいは震撼させられることでしょう。

著者について

Brian Goetz

Brian Goetz は18 年間以上に渡って、専門的ソフトウェア開発者として働いています。彼はカリフォルニア州ロスアルトスにあるソフトウェア開発コンサルティング会社、Quiotixの主席コンサルタントであり、またいくつかのJCP Expert Groupの一員でもあります。2005年の末にはAddison-Wesleyから、Brianによる著、Java Concurrency In Practiceが出版される予定です。Brian著による有力業界紙に掲載済みおよび掲載予定の記事のリストを参照してください。