Javaの理論と実践: スレッド・セーフの特性について

スレッド・セーフは「全てかゼロか」の命題ではありません

Brian Goetz
Principal Consultant
Quiotix

2003年 9月 23日

並行性のエキスパートであるBrian Goetzは以前の記事で、`Hashtable`や`Vector`クラスを「条件付きスレッド・セーフ」として解説しました。クラスはスレッド・セーフであるべきなのでしょうか? 残念ながらスレッド・セーフは「全てかゼロか」の命題ではなく、意外にも定義が難しいものなのです。しかしBrianが今回のJavaの理論と実践で述べているように、クラスがスレッド・セーフであるかどうかの分類をJavadocの中に記述する手間をかけるのは非常に重要なことなのです。

このシリーズの他の記事を見る

Joshua Blochは素晴らしい著書である『Effective Java Programming Language Guide』(参考文献)の52項「Document Thread Safety」で、クラスが保証するスレッド・セーフがどんなものなのかを厳密に、文章で書いておくように開発者に対して勧めています。これはしばしば繰り返されてきた素晴らしいアドバイスなのですが、あまり実行されてはいません。(Blochが、プログラミングの難問に関する対話の中で「私の兄のようなつもりでコーディングしないでください。」と言っているように)※j-jone3.htmlより

みなさんはクラスに関するJavadocを見て、何度も「このクラスはスレッド・セーフなのだろうか?」と疑問に思ったことがありませんか。明確なドキュメントが無いと、読み手はクラスのスレッド・セーフに関して誤った推測をしてしまうかもしれないのです。つまり、読み手はスレッド・セーフではないのにスレッド・セーフであると推測するかもしれませんし(これはかなりまずいことです!)、あるいはメソッドの1つを呼びだす前に、オブジェクトを同期化することでスレッド・セーフにできると推測してしまうかもしれません (これは正しいかも知れませんし、単に非能率なだけかも知れませんし、最悪の場合はスレッド・セーフだという錯覚を与えてしまうかも知れません) 。いずれにせよ、インスタンスがスレッドをまたがって共有される場合は、クラスがどのように振る舞うのかを文書として明らかにしておく方が良いのです。

この落とし穴の例として、クラス`java.text.SimpleDateFormat`はスレッド・セーフではないのですが、そのことはJDK1.4までJavadocにドキュメント化されていませんでした。おかげでどれほどの開発者が、誤って`SimpleDateFormat`のstaticなインスタンスを作成してしまったり、重負荷時に

プログラムが正しく動作しないことに気付かずマルチ・スレッドからこれを使用したりしたことでしょう。みなさんの顧客や同僚にはそんな目に遭わせないでください！

忘れる前に記録せよ(できないなら会社を辞めなさい)

スレッド・セーフをドキュメント化すべき時期は、当然ながら初めてクラスを書く時です。クラスを書いている時にスレッド・セーフの条件やクラスの振る舞いを評価しておく方が、数か月後にドキュメント化するよりもはるかに簡単なのです。クラスを書いている時以上に実装で何が起きているかを理解している時はないのです。また、クラスを作成する際にスレッド・セーフが保証することをドキュメント化しておけば (メンテナンス者がクラスの仕様の一部としてそのドキュメントを参照することが期待できるので)、将来の変更においても、スレッド・セーフに関して当初意図したことが保持される可能性が高くなるのです。

クラスがスレッド・セーフであるか否かが、クラスのバイナリー属性として簡単に記述できるようになっていれば良いのですが、残念ながらそれほど単純ではないのです。クラスがスレッド・セーフではない場合、そのクラスのオブジェクトへのすべてのアクセスを同期化することでスレッド・セーフにすることができるのでしょうか。他のスレッドとの衝突を容認せず、したがって、基本オペレーションだけでなく複合オペレーションにも同期化を要求するようなオペレーションのシーケンスはあるでしょうか？あるオペレーション群をアトミックに実行する事を要求するメソッド間には、状態の依存性があるのでしょうか？並行アプリケーションでクラスを使用する際に開発者が知るべきなのは、こうした情報なのです。

スレッド・セーフを定義する

スレッド・セーフを明白に定義するのは驚くほど困難で、ほとんどの定義はひどく循環的なようです。Googleで少し探してみれば、スレッド・セーフなコードの定義として典型的な、でも役には立たない、むしろ説明のようなものが見つかります。

- ...スレッド間で無用なやりとりがなく、複数のプログラミング・スレッドから呼び出すことができる。
- ...呼び出し側のいかなるアクションも必要とせずに、一度に1つ以上のスレッドに呼び出される可能性がある。

このような定義では、私たちがスレッド・セーフで混乱してしまうのも不思議はありません。これらの定義は「複数のスレッドから安全に呼び出せるならば、クラスはスレッド・セーフである」という以上のことは言っておらず、その通りではあっても、スレッド・セーフなクラスをスレッド・セーフではないクラスと区別する助けにはなりません。では、セーフとはどういう意味なのでしょう？

現実にはスレッド・セーフの定義はどうしてもある程度循環的なものになってしまいます。これはクラスの仕様 (クラスが行う事やその副作用、どの状態が有効/無効か、不変条件、必須条件、事後条件等々、を非公式に散文的に記述したもの) に頼らざるを得ないためです。(仕様によって定められた、オブジェクトの状態制約は、 [内部状態ではなく] 外部から見える状態 [つまりpublicメソッドを呼び出し、publicフィールドにアクセスすることで見ることができる状態] にのみ適用され、これが実際にそのprivateフィールドで見えるものになります。)

スレッド・セーフ

クラスがスレッド・セーフであるためには、まずシングル・スレッド環境で正確に振る舞わなければなりません。クラスが正しく実装されている場合、言い換えればその仕様に従っている場合には、そのクラスのオブジェクトに対するどんなシーケンスのオペレーション(publicフィールドの読み込み書き込み、およびpublicメソッドの呼び出し)であっても、オブジェクトを無効な状態にすることはできず、無効な状態にあるオブジェクトを参照するはずもなく、クラスの不変条件、必須条件、事後条件のいずれにも反することはないはずです。

上で述べたような意味でクラスがスレッド・セーフであるためにはさらに、複数のスレッドからアクセスされた時にも、そうしたスレッドの（実行環境による）実行スケジューリングやインターリーブに関わらず、呼び出し側コードの同期化を要したりもせずに、正しく振る舞い続ける必要があります。その結果、スレッド・セーフなオブジェクトに対するオペレーションはどのスレッドからも一定で、常に一貫性のあるものに見えるのです。

正確さとスレッド・セーフの関係は、ACID(原子性、一貫性、独立性、持続性)トランザクションについての記述における一貫性と独立性の関係に非常に似ています。つまり、あるスレッドにとってみると、別のスレッドがそのオブジェクトに対して行うオペレーションは並行的ではなく、(順序は決められませんが）順次的に見えるのです。

メソッド間の状態依存性

`Vector`の要素を繰り返す以下のコードを考えてみてください。`Vector`のすべてのメソッドは同期化されていますが、追加同期のないマルチスレッド環境でこのコードを使用するのはまだ安全とは言えません。他のスレッドが、ちょうど悪いタイミングで要素を削除してしまうと、`get()`が`ArrayIndexOutOfBoundsException`をスローすることになってしまうからです。

```
Vector v = new Vector();
    // contains race conditions -- may require external synchronization
    for (int i=0; i<v.size(); i++) {
      doSomething(v.get(i));
    }
```

このコードでは、`get(index)`の仕様の一部として必須条件があり、`index`は負でなく`size()`より小さい必要があります。しかし、マルチスレッド環境では、`size()`が最後に読み取られた値が、まだ有効かどうかを知る方法がありません。ですから（最後に`size()`を呼び出す前から`Vector`の排他ロックを保持していない限り）、`i<size()`かどうかを知る事はできないわけです。

もっと具体的に言うと、`get()`の必須条件が`size()`の結果によって決まるという事から問題が生じているのです。こうした（あるメソッドの結果を別のメソッドの入力条件とする）パターンがあるときには常に状態依存性があり、状態のその要素が、少なくとも２つのメソッド呼び出しの間は変わらないように保証する必要があります。一般的に言って、このための唯一の方法としては、最初のメソッドを呼び出す前から最後のメソッドを呼んだ後まで、オブジェクトの排他ロックを保持する事です。上の例では`Vector`の要素を繰り返すイテレーションの間、`Vector`オブジェクトを同期化します。

スレッド・セーフの程度

上記の例が示すように、スレッド・セーフは全てかゼロかではありません。`Vector`のメソッドはすべて同期化されており、`Vector`は確かにマルチスレッド環境で機能するように設計されています。しかし、そのスレッド・セーフには限界があります。つまり、あるメソッドの組み合わせには状態依存性があるのです。(同様に、イテレーションの間に別スレッドが`Vector`を変更すると、`Vector.iterator()`が返すイテレータは`ConcurrentModificationException`をスローします。)

Javaクラスで通常起こり得る様々なレベルのスレッド・セーフに適用できるような条件として、広く受け入れられているものはありません。とは言え、クラスのコードを書いている間に、スレッド・セーフに関する振る舞いを文書化しようと努力するのは重要な事です。

Bloch は、スレッド・セーフを5つのカテゴリ(不変、スレッド・セーフ、条件付きスレッド・セーフ、スレッド互換、反スレッド)に分類して概説しています。スレッド・セーフ特性を明確にドキュメント化できるのであれば、この体系を使用するかどうかは重要ではありません。この体系には制限 (各カテゴリの境界が100%明確ではなく、どのカテゴリにも含まれないケースがあります)もありますが、手始めとしては良くできたものと言えるでしょう。この分類体系の中心となっているのは、呼び出し側が外部同期でオペレーション(あるいはオペレーションのシーケンス)を囲むことができるか、囲まなければならないかどうか、という点です。スレッド・セーフに関するこれらの5つのカテゴリを以下のセクションで説明します。

不変

このコラムの愛読者であれば、私が不変の持つ価値を称賛しても驚かないでしょう。不変オブジェクトはスレッド・セーフであることが保証されていて、追加同期を要求することはありません。不変オブジェクトは正しく作成されている限り、外部から見える状態は変わらないので、矛盾した状態であると認識されることはありません。`Integer`、`String`、`BigInteger`のようなJavaクラス・ライブラリの基底クラスのほとんどは不変です。

スレッド・セーフ

スレッド・セーフなオブジェクトには、上記の「スレッド・セーフ」セクションで説明したプロパティがあります。つまりオブジェクトが複数スレッドにアクセスされる際は、ランタイム環境でどのようにスレッドがスケジュールされているかによらず、追加同期無しに、クラスの仕様に決められた制約が継続して保持されるのです。このスレッド・セーフの保証は強力なもので、`Hashtable` や`Vector`のような多くのクラスはこの厳しい定義に対応できません。

条件付きスレッド・セーフ

7月の「並行コレクション・クラス」の記事で、条件付きスレッド・セーフについて説明しました。条件付きスレッド・セーフ・クラスとは、個々のオペレーションはスレッド・セーフであっても、一定のシーケンスによるオペレーションでは外部同期を要求するものです。条件付きスレッド・セーフの最も一般的な例は、`Hashtable`や`Vector`が返すイテレータのトラバースです。これらのクラスが返すfail-fastイテレータは、イテレータのトラバース実行中は、下にあるcollectionが変化することはないと想定しています。トラバース期間中に他のスレッドがcollectionを変化させないよう確実を期すため、繰り返しのスレッドは、全トラバースでcollectionに排他アクセスで

きる必要があります。通常、排他アクセスは、ロックでの同期化によって保証されます。ですからクラスのドキュメントには、それがどのロックなのか (通常はそのオブジェクトに備わったモニタ) を指定しておくべきです。

条件付きスレッド・セーフ・クラスをドキュメント化する場合には、それが条件付きスレッド・セーフである事だけでなく、どのシーケンスのオペレーションを同時アクセスから保護する必要があるかもドキュメント化しておくべきです。こうしておけばユーザーはかなりの確率で、他のシーケンスのオペレーションでは追加同期を要求しないのだと判断してくれるでしょう。

スレッド互換

スレッド互換クラスはスレッド・セーフでありませんが、同期化を適切に使用することで、並行環境でも安全に使用することができます。これはすべてのメソッド呼び出しを`synchronized`ブロックで囲む事かも知れませんし、(`Collections.synchronizedList()`のように)すべてのメソッドを同期化するラッパー・オブジェクトを作成する事かも知れません。あるいは、あるシーケンスのオペレーションを`synchronized`ブロックで囲むことかも知れません。スレッド互換クラスの使い易さを最大限に生かすためには、呼び出し側に対して特定のロックで同期化するように要求すべきではなく、すべての呼び出しで同じロックを使用するようにすべきです。そうすることによって、スレッド互換オブジェクトを他のスレッド・セーフ・オブジェクトのインスタンス変数として保持し、所有オブジェクトの同期化にならうことができるのです。

多くの共通クラス、`collection` クラスの`ArrayList`や`HashMap`、`java.text.SimpleDateFormat`、JDBCクラスの`Connection` や`ResultSet`などはスレッド互換です。

反スレッド

反スレッド・クラスは、呼び出される外部同期にかかわらず、同時に使用するのは安全ではありません。反スレッドは稀であり、通常はクラスが (他のスレッドで実行され、他のクラスの振る舞いに影響する) staticなデータを変更する際に発生します。反スレッドのクラスの例としては、`System.setOut()`を呼び出すクラスがあります。

スレッド・セーフのドキュメントで他に考慮すべきこと

スレッド・セーフなクラス(および、軽度のスレッド・セーフを備えたクラス)は、呼び出し側が排他アクセスのためにオブジェクトをロックするのを許す場合もあり、許さない場合もあります。`Hashtable`クラスは全ての同期化に対して、オブジェクトに内在しているモニタを使用しますが、`ConcurrentHashMap`クラスは使用しません。また実際、排他アクセス用に`ConcurrentHashMap`オブジェクトをロックする方法はありません。クラスがどの程度のスレッド・セーフなのかをドキュメント化するのに加えて、 (オブジェクトに内在しているロックのように) 特定のロックがクラスの振る舞いに特別な意味を持っているかどうかもドキュメント化しておくべきです。

クラスがスレッド・セーフであることをドキュメント化することによって(実際スレッド・セーフであると言う前提ですが)、2つの貴重なサービスを提供できます。クラスを維持管理する人に対して、スレッド・セーフに影響を与えるような変更や拡張はするべきでないと伝えたり、クラスのユーザーに対して、外部同期無しで使用することができると知らせたりする事ができるの

です。クラスはスレッド互換である、あるいは条件付きスレッド・セーフである、とドキュメント化しておけば、同期化を適切に使用することで、複数スレッドでも安全に使用できる、とユーザーに伝えることができます。クラスが反スレッドであることをドキュメント化しておけば、たとえ外部同期を使っても複数スレッドからは安全に使用することができない、とユーザーに伝えることができます。どの場合でも、（発見も修正も困難な）潜在的で重大なバグを、発生前に防止できるのです。

結論

スレッド・セーフに関するクラスの振る舞いは仕様の一部であり、きちんとドキュメント化すべきです。これに関する明確な記述方法は(まだ)存在しないので、文章として記述するしかありません。Blochによる５段階の分類体系は、あらゆるケースを網羅してはいませんが、出発点としては非常に良いものです。全てのクラスがJavadocの中に、（Blochの分類くらいの)明確なスレッド振る舞いの記述を含んでいてくれれば、大変素晴らしい事なのですが。

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。