Javaの理論と実践: Javaメモリ・モデルを修正する　第2回

JSR 133の下でJMMはどのように変わるのか？

Brian Goetz
Principal Consultant
Quiotix

2004年 3月 30日

JSR 133は3年近く活動していますが、最近Javaメモリ・モデル（JMM）をどう扱うべきかについて公開勧告を出しました。このシリーズの第1回ではコラムニストのBrian Goetzが、元々のJMMで見つかった深刻な問題のいくつかに焦点を当てました。こうした問題によって、簡単なはずの概念が驚くほど困難な意味体系になってしまったのです。今回は新しいJMMの下でvolatileやfinalの意味体系がどのように変わるか、またそうした変更によって、意味体系が大部分の開発者の直感に合うようになることを明らかにします。こうした変更の一部は既にJDK 1.4に統合されていますが、残りはJDK 1.5まで待つ必要があります。

このシリーズの他の記事を見る

並行コードを書くのは元来難しいものです。言語でそれをさらに困難にすべきではありません。Javaプラットフォームでは最初からスレッド化をサポートしており、これには（適切に同期化されたプログラムに対して「Write Once, Run Anywhere（一度書けばどこでも実行できる）」を保証するはずの）プラットフォームによらず動作するメモリ・モデルも含まれていたのですが、元々のメモリ・モデルには少し穴があったのです。多くのJavaプラットフォームではJMMが要求するよりも高度な保証を提供しているのですが、JMMの穴は、どのプラットフォームでも動作する並行Javaプログラムを簡単に書く上での障害だったのです。そこで2001年の5月、Javaメモリ・モデルを修正すべくJSR 133が組織されたのです。前回は、こうした穴のいくつかについてお話ししました。今月はその穴がどのようにしてふさげられたかを説明します。

可視性の問題、再訪

JMMを理解するために必要となる重要な概念の一つが可視性の概念です。スレッドAが`someVariable = 3`を実行すると他のスレッドは、値3がスレッドAによって書かれたことを認識する、とどうして言えるのでしょう？ ある別のスレッドが`someVariable`に対する値3を即座には見ないかも知れない、という理由がいくつかあるのです。コンパイラーがもっと効率良く実行するために命令をリオーダー（reorder）したかも知れませんし、`someVariable`がレジスターにキャッシュされたかも知れませんし、値は書き込みプロセッサーのキャッシュに書き込まれたのですが、メイン・メモリにはまだ吐き出されていないかも知れませんし、古い値が読み込みプロセッサーのキャッシュにまだ残っているかも知れません。あるスレッドが、他のスレッドが書き

込んだ変数をいつの時点で確実に「見る」のかを決定するのはメモリ・モデルなのです。特にメモリ・モデルは、スレッドにまたがるメモリ操作の可視性を保証するvolatileやsynchronizedそれにfinalの意味体系を定義しているのです。

このシリーズ他の記事も忘れずに

第1回「Javaメモリ・モデルとは何か、そもそもなぜうまく行かなかったのか？」（2004年2月）

関連モニター解放の一部としてスレッドが同期化ブロックから出る時には、JMMはローカル・プロセッサーのキャッシュをメイン・メモリに吐き出すように要求します。（実際にはメモリ・モデルはキャッシュのことを言うわけではなく、キャッシュやレジスターその他のハードウェアとコンパイラー最適化を包含する、ローカル・メモリという抽象的概念のことを言うのです。）同じように、同期化ブロックに入る際のモニター取得の一部として、（次に行われる読み込みがローカル・キャッシュではなく直接メイン・メモリに行くように）ローカル・キャッシュは無効化されます。あるスレッドが、対象のモニターに保護された同期化ブロック期間中に変数を書き込み、同じモニターに保護された同期化ブロック期間中に別のスレッドがその変数を読み出す時には、その変数への書き込みは読み取り側のスレッドから見えることが、この過程によって保証されるのです。同期化がない場合には、JMMはこの保証をしません。複数のスレッドが同じ変数にアクセスする時には必ず同期化（またはその弟分のvolatile）を使う必要があるのはこの理由からです。

Volatileに対する新たな保証

元々のvolatileの意味体系では、volatileフィールドの読み書きはレジスターやローカル・プロセッサーのキャッシュに対してではなく直接メイン・メモリに対して行われるという事と、（スレッドに代わって行われる）volatile変数に対する操作はスレッドが要求した順で行われる事だけを保証していました。これは言い換えれば、古いメモリ・モデルでは変数が読み書きされていることが見える事だけを約束していて、他の変数への書き込みが見えるとは約束していない、ということを意味するのです。これは効率的に実装するのは簡単なのですが、当初考えられていたほど有用ではないことが分かったのです。

Volatile変数の読み書きは他のvolatile変数の読み書きでリオーダーされる事はないのですが、非volatile変数の読み書きではやはりリオーダーされるのです。第1回の記事で、configOptionsに対する正しい値や、（Mapの要素のような）configOptionsで間接的に到達できる全ての変数がスレッドBから見えるように保証するには、リスト1のコードでは（古いメモリ・モデルで見ると）なぜ不十分なのかを学びました。その理由としてはconfigOptionsの初期化がvolatile変数initializedの初期化でリオーダーされたかもしれない、というものでした。

リスト1. volatile変数を「見張り」として使う

```
Map configOptions;
char[] configText;
volatile boolean initialized = false;
// In Thread A
configOptions = new HashMap();
configText = readConfigFile(fileName);
processConfigOptions(configText, configOptions);
initialized = true;
// In Thread B
while (!initialized) sleep();
// use configOptions
```

残念ながら、この状況・・共有変数一式が初期化された事を示すためにvolatileフィールドを「見張り」として使う・・になるのはvolatileの使い方として一般的なのです。JSR 133専門家グループ（Expert Group）は正にこの状況や、これに似た状況に対処するために、「volatile読み書きは他のメモリ操作でリオーダーされない」とする方がより実際的だ、と結論づけたのです。新しいメモリ・モデルでは、スレッドAがvolatile変数Vに書き込み、スレッドBがVから読み出す時には、Vに書き込む時にAから見えるどんな変数値も、Bから見えることが保証されるようになったのです。その結果、volatileフィールドへのアクセスに対するコストは少し高くなりましたが、`volatile`の意味体系がより有用なものになったのです。

何の前に何が起きるのか？

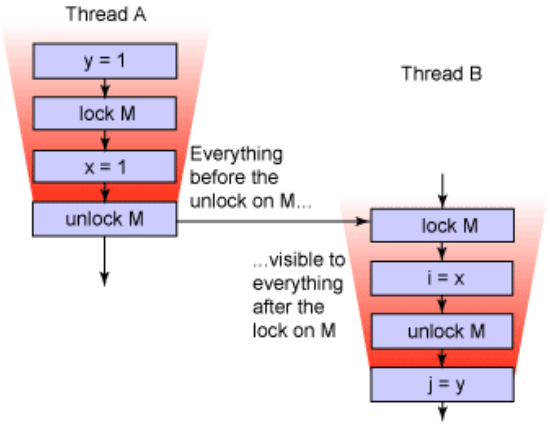
変数の読み書きのような操作は、いわゆる「プログラム配列（program order）」・・プログラムの意味体系が規定する読み書きの配列・・に従って、スレッド内で配列されます。（実際にはコンパイラーはas-if-serial意味体系が確保される限り、スレッド内のプログラム配列に対して多少の自由が許されています。）異なるスレッドで行われる操作は、お互いの関係において必ずしも配列されるとは限りません。つまり2つのスレッドが開始され、それぞれが共通のモニターで同期されることもなく、共通のvolatile変数に触れることもなく実行される場合には、（3番目のスレッドから見える）一方のスレッドで行われる操作と、他方のスレッドで行われる操作との相対的な順序については全く完全に、何も予測できないのです。

スレッドが開始される時や、あるスレッドが別のスレッドとつなげられる時、スレッドがモニターを取得・解放する（同期化ブロックに入る、またはそこから出る）時、スレッドがvolatile変数にアクセスする時には追加的な配列保証（ordering guarantees）が生成されます。JMMは、プログラムが複数スレッドでの操作を調整するために同期化やvolatile変数を使用する時に行われる配列保証について説明しています。新しいJMMは、非公式に、happens-before（事前発生）と呼ばれる配列を定義していますが、これは次に示すようにプログラム内で行われる全操作の部分配列です。

- 同じスレッド内で、プログラム配列の前の方に出てくる操作は、後に出てくる操作よりもhappens-beforeである
- 同じモニター上で、モニターのアンロックはロックよりもhappens-beforeである
- volatileフィールドへの書き込みは、それに続く、同じvolatileの読み込みよりもhappens-beforeである
- あるスレッドの`Thread.start()`へのコールは、開始されたそのスレッドでのどの操作よりもhappens-beforeである
- スレッドでの全操作は、そのスレッドの`Thread.join()`から成功（success）して戻る他のどのスレッドよりもhappens-beforeである

上に挙げた規則の中で新規であり、リスト1に挙げた例の問題を解決するのは、volatile変数の読み書きを管理する3番目の規則です。volatile変数`initialized`への書き込みは`configOptions`の初期化の後に起こるので、`configOptions`の使用は`initialized`の読み込みの後であり、`initialized`の読み込みは`initialized`への書き込みの後に起こるので、スレッドAによる`configOptions`の初期化は、スレッドBによる`configOptions`使用の前に起こると結論づける事ができます。ですから、`configOptions`とそれから到達できる変数はスレッドBから見えるのです。

図1. 同期化を使って、スレッドにまたがるメモリ書き込みの可視性を保証する



データ・レース（Data races）

ある変数を複数のスレッドが読み、少なくとも一つのスレッドが書き込み、なおかつ書き込みと読み出しがhappens-before関係によって配列されていない時には、プログラムにはデータ・レース（data race）があり、従って「適切に同期化」されていないプログラムであると言われます。

これはdouble-checked locking問題を解決するのか？

double-checked locking問題に対して提案されている解決方法の一つは遅延初期化されたインスタンス（lazily initialized instance）を保持するフィールドをvolatileフィールドにするというものです。（double-checked locking問題と、その解決方法として提案されたアルゴリズム的な方法ではなぜうまく行かないかの説明については参考文献を見てください。）古いメモリ・モデルの下では、これではdouble-checked lockingをスレッド・セーフにしませんでした。その理由はvolatileフィールドへの書き込みは、他の非volatileフィールド（例えば新しくコンストラクトされたオブジェクトのフィールドなど）への書き込みで、やはりリオーダーでき、そのためvolatileインスタンス参照は、不完全にコンストラクトされたオブジェクトへの参照をやはり保持できたためです。

新しいメモリ・モデルの下では、double-checked lockingに対するこの「解決方法」で表現法（idiom）がスレッド・セーフになるのです。それにもかかわらず、まだこの表現法を使うべきではないのです！ Double-checked lockingの要点は、ごく初期のJDKでは同期化が比較的高価だったという大きな理由から、共通コード・パスの同期化を不要にするために考えられた、パフォーマンス最適化のはずだった、ということなのです。その後、非競合同期化（uncontended synchronization）はずっと安価になったのですが、`volatile`の意味体系に加えられた新しい変更によって、一部のプラットフォームでは古い意味体系よりも比較的高価になってしまったのです。（実質的には、volatileフィールドへの各読み書きはちょうど、「半」同期化のようなものです。つまりvolatileの読み込みはモニターが取得するのと同じメモリ意味体系を持ち、volatileへの書き込みはモニターが解放するのと同じ意味体系を持っているのです。）ですから、double-checked lockingの目標が、より単純な、同期化による手法よりも改善されたパフォーマンスを得る事だとすると、この「修正版」解決方法もあまり役には立たちません。

Double-checked lockingの代わりに、遅延初期化（lazy initialization）ができるInitialize-on-demand Holder Class表現法を使えば、スレッド・セーフな上に高速であり、かつdouble-checked lockingよりも混乱が少なくなります。

リスト2. Initialize-On-Demand Holder Class表現法

```
private static class LazySomethingHolder {
  public static Something something = new Something();
}
...
public static Something getInstance() {
  return LazySomethingHolder.something;
}
```

この表現法は、（例えば静的初期化子（static initializers）など）クラス初期化の一部としての操作は、そのクラスを使う全てのスレッドから見える事が保証されているという事実からスレッド・セーフを確保しており、あるスレッドがそのフィールドやメソッドの一つを参照するまで内部クラスはロードされないという事実から遅延初期化（lazy initialization）を確保しています。

初期化安全（initialization safety）

新しいJMMでは初期化安全（initialization safety）に関する新しい保証機構も提供しようとしています。つまり、オブジェクトが適切にコンストラクトされている（つまりそのオブジェクトへの参照はコンストラクターが完了するまで公開されない）限り、あるスレッドから別のスレッドへ参照を渡すために同期が使われているか否かによらず、すべてのスレッドが、（そのオブジェクトのコンストラクターの中で設定された）オブジェクトのfinal field の値を見るのです。さらに、適切にコンストラクトされたオブジェクトのfinal field を通して到達できる変数（例えばfinal field が参照する、オブジェクトのフィールド）はどんな変数でも、やはり他のスレッドから見える事が保証されているのです。これはつまりfinal field が、（他のスレッドから見える、参照の正しい値に加えて）例えば`LinkedList`への参照を含んでいる場合には、（コンストラクト時の）その`LinkedList`の内容も、同期化無しで他のスレッドから見えることを意味します。その結果として、`final`の意味が大幅に強化されたのです。つまり同期化無しでもfinal field には安全にアクセスする事ができ、コンパイラーはfinal field が変わらないものと想定する事ができ、従って繰り返しのフェッチを避けて最適化する事ができるのです。

Finalは最終を表す

final field が値を変えるように見えてしまうという、古いメモリ・モデルでの機構は第1回で概要を説明しました。つまり同期化がないと、他のフィールドから見るfinal field の値は、最初はデフォルト値、その後正しい値、となるのです。

新しいメモリ・モデルの下では、コンストラクター中のfinal field への書き込みと、別のスレッドにある、そのオブジェクトへの共有参照の最初のロードとの間にはhappens-beforeに似た関係があります。コンストラクターが完了すると、final field（と、final field から間接的に到達できる変数）への全書き込みは「凍結」され、（凍結後の）そのオブジェクトへの参照を取得するどのスレッドも、凍結された全てのフィールドの、凍結された値を見るように保証されているのです。final field を初期化する書き込みは、コンストラクターに付随する凍結の後に続く操作によってリオーダーされることはありません。

まとめ

JSR 133は`volatile`の意味体系を大幅に強化しました。その結果、別のスレッドによってプログラムの状態が変更されたことを示すフラグとして、volatileフラグを安心して使えるようになりまし

た。Volatileをより「重量級（heavyweight）」にすることで、volatileを使う事によるパフォーマンスのコストが、場合によっては同期化によるパフォーマンスのコストと近くなってしまいますが、それでもまだ大部分のプラットフォームではきわめて安価です。JSR 133はまた`final`の意味体系も大幅に強化しました。コンストラクト期間中にオブジェクトによる参照が漏洩を許されていない場合には、一旦コンストラクターが完了して、あるオブジェクトへの参照をスレッドが公開すると、そのオブジェクトのfinal field は同期化が無くても見える事や正しい事、また他のスレッドに対して一定である事が保証されるのです。

こうした変更によって、並行プログラムでの不変オブジェクト（immutable objects）の実用性が大幅に強化されました。不変オブジェクトがついに（本来意図された通り）真の意味でスレッド・セーフになり、たとえ不変オブジェクトへの参照をスレッド間で渡すためにデータ・レースが使われたとしてもスレッド・セーフのままなのです。

初期化安全として一つ注意しなければならないのは、オブジェクトによる参照はコンストラクターを漏洩してはならない、つまりコンストラクターは直接・間接によらずコンストラクトされつつあるオブジェクトへの参照を公開してはならない、という点です。これには非静的内部クラス（nonstatic inner classes）への参照の公開も含まれており、一般的にはコンストラクター内部からスレッドを開始することを禁止しています。安全なコンストラクトについてのより詳細な説明については参考文献を見てください。

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。