Javaの理論と実践: Javaメモリ・モデルを修正する　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？

Brian Goetz
Principal Consultant
Quiotix

2004年 2月 24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。元のJMMには深刻な問題がいくつか見つかり、`volatile`や`final`、それに`synchronized`などといった、簡単なはずの概念に対して驚くほど困難な意味体系になってしまったのです。Javaの理論と実戦の今回の記事ではBrian Goetzが、JMMを修正するために`volatile`や`final`の意味体系をどのように強化するかを説明します。こうした変更の一部は既にJDK 1.4に統合されており、残りはJDK 1.5に統合される事になっています。

このシリーズの他の記事を見る

Javaプラットフォームは、Java以前の大部分のプログラミング言語よりもスレッドやマルチ・プロセスの考え方を大幅に採り入れています。Java言語の持つ、プラットフォームに依存しない並行性やマルチスレッドのサポートは意欲的で画期的なものですが、まさにそれが故にメモリ・モデルの問題が、Javaを設計した人たちが元々考えていたよりもちょっと難しいのも驚くには当たらないかも知れません。同期化やスレッド・セーフにまつわる混乱の根元の大部分は、あまり直感的ではない、Javaメモリ・モデル（JMM）の微妙さにあるのです。JMMは元々、Java言語仕様（Java Language Specification）の17章で規定され、JSR 133で再定義されたものです。

例えば、全てのマルチプロセッサー・システムにキャッシュ一貫性があるわけではありません。一つのプロセッサーがキャッシュに変数の更新値を持っていて、メイン・メモリにはまだ吐き出していない場合には、他のプロセッサーにはその更新値が見えないかも知れません。キャッシュ一貫性が無いと、異なる2つのプロセッサーはメモリ上の同じ位置で、異なった2つの値を見るかも知れないのです。そう聞くと恐ろしいと思うかも知れませんが、設計上そうなっているのです。これは、より高いパフォーマンスとスケーラビリティを得るための手段なのですが、こうした問題に対応するコードを生成する開発者やコンパイラーには重荷になります。

メモリ・モデルとは何か、なぜ必要なのか

メモリ・モデルはプログラム中の変数（インスタンス・フィールドやスタティック・フィールド、配列要素など）と、これらの変数を現実のコンピューター・システムのメモリに保存して取

り出すための、低レベルの詳細との関係を記述しています。オブジェクトは最終的にはメモリに保存されるのですが、コンパイラーやランタイム、プロセッサー、キャッシュなどが変数に割り当てられたメモリ上の場所に変数を動かしたり、そこから取り出したりするタイミングには多少の自由度があるのです。例えば、コンパイラーはループ・インデックス変数をレジスターに保存する事で最適化しようとするかもしれませんし、キャッシュは変数の新しい値をメインメモリに吐き出すタイミングをもう少し都合の良い時間まで遅らせるかも知れません。こうした最適化はすべてパフォーマンスを高めるためであり、一般的にはユーザーが気にする必要はないのですが、マルチプロセッサー・システムでは、こうした複雑な機構が時々見えてしまうのです。

このシリーズ他の記事も忘れずに

第2回「How will the JMM change under JSR 133」（英語）

JMMは、データをプロセッサー固有のキャッシュ（またはレジスター）とメイン・メモリの間で動かす順序に関しては、プログラマーが`synchronized`や`volatile`を使って明示的に一定の可視性（visibility）保証を要求しない限り、コンパイラーやキャッシュに対して大幅な自由を許しています。これはつまり、同期化がない場合のメモリ操作は、異なるスレッドから見た場合には、異なった順序で起こる可能性があるということになります。

これとは対照的に、CやC++といった言語には明示的なメモリ・モデルはありません。その代わりCプログラムはプログラムを実行しているプロセッサーのメモリ・モデルを継承します。（実は与えられたアーキテクチャに対するコンパイラーは、下にあるプロセッサーのメモリ・モデルと、コンパイラーにも一定の適合性に従うべき責任があることをおそらく知っているのですが。）これはつまり並行なCプログラムは、あるプロセッサー・アーキテクチャでは正しく実行できても、別のプロセッサー・アーキテクチャでは実行できないかも知れない、ということです。JMMには最初は混乱させられるかも知れませんが、重要な利点があるのです。つまりJMMに従って正しく同期化されたプログラムは、Javaが動作するプラットフォームであればどんなものでも正しく実行できるのです。

オリジナルのJMMにおける欠陥

Java言語仕様（Java Language Specification）の17章で規定されているJMMは、一貫性を持った、どんなプラットフォームでも動作するメモリ・モデルを定義しようとする意欲的な試みでしたが、微妙で、しかも重大な欠陥があったのです。`synchronized`や`volatile`の意味体系は非常に混乱を招きやすく、古いメモリ・モデルの下で同期化したコードを適切に書くのはあまりにも難しいので、知識ある開発者でさえ、時には規則を無視してしまったのです。

古いJMMでは驚くべきことや紛らわしいことができてしまうのです。例えばコンストラクターで設定される値を持っていないように見える final フィールド（これで不変であるはずのオブジェクトが不変でなくなってしまう）や、メモリ操作のリオーダ（reorder）で予期せぬ結果になったりすることなどです。本来は効果的であるはずのコンパイラー最適化が一部動作しないこともありました。double-checked locking の問題に関する記事を読んだ事があれば（参考文献）、メモリ操作リオーダがどれほどややこしいものか、適切に同期化を行わないと（または積極的に同期化を回避しないと）微妙な、しかも重大な問題がコードに入り込む可能性があるかを思い出して頂けるでしょう。もっとまずいのは、不適切に同期化されたプログラムであっても、負荷が軽い場合や単一プロセッサーの場合、そしてJMMよりも強力なメモリ・モデルを要求するプロセッサーの場合など、ある状況では正しく動作するように見えてしまう事です。

リオーダという用語は、実際の、明らかなメモリ操作のリオーダを行ういくつかのクラスを記述するために使っています。

- コンパイラーは、プログラムの意味体系を変えない場合には最適化として、ある命令群を自由にリオーダすることができる。
- プロセッサーは、ある条件下では所定の手続きを踏まずに操作を実行する事ができる。
- キャッシュは一般的に、プログラムが書き込んだのとは異なる順序で、変数群をメイン・メモリに書き戻す事ができる。

こうした条件のどれもが、（ある別のスレッドから見た場合に）操作の順序がプログラムで規定したのとは異なって見える原因になります。

JSR 133の目標

JSR 133はJMMを修正するのが責務ですが、いくつかの目標を持っています。

- タイプの安全性（type-safety）を含め、既存の安全性保証を維持する。
- どこから来たのか、に関する安全性（out-of-thin-air safety）を提供する。これは変数の値は「どこからともなく」作られたわけではない、と言う意味です。ですからスレッドが、変数に値Xがある、と認識するということは、過去にその変数に対して値Xを実際に書いたスレッドがあるはずです。
- 「正しく同期化された」プログラムの意味体系をできるだけ単純、直感的にする。そうすることで「正しい同期化」を正式かつ直感的に定義できるはずです（そうすれば２つの定義が一貫したものになるのです！）。
- プログラマー達が、信頼性が高く正しいマルチスレッドのプログラムを、自信を持って作成できるようにする。もちろん、並行アプリケーションが簡単に書けるような魔法はありませんが、目標としては、アプリケーション作成者がメモリ・モデルの微妙さを全て理解しなくてもすむようにする事です。
- 広範囲にわたる、ごく一般的なハードウェア構成で高機能なJMM実装ができるようにする。最近のプロセッサーはメモリ・モデルに関して大きく異なっています。JMMは、構成が現実的なものであれば、その性能を犠牲にすることなく、可能な限り広範囲な構成を許すべきです。
- 同期化表現法として、オブジェクトを公開できるもの、同期化無しにそのオブジェクトが見えるようにできるものを提供する。これは安全性の保証として新しいもので、初期化安全（initialization safety）と呼ばれます。
- 既存のコードへの影響を最小限にする。

注意して欲しいのですが、新しいメモリ・モデルの下でも double-checked locking のようにうまく行かない手法はやはりうまく行かないのであって、それを「修正」するのは新しいメモリ・モデルの目標ではありません。（ただし、`volatile`の新しい意味体系では、double-checked lockingの代替としてよく提案される手法の一つが（使わないように推奨されていますが）正しく動くようにしています。）

JSR 133の作業が行われるようになってから、こうした問題は指摘されたよりもずっと微妙な事が分かってきました。開拓者の仕事というのはそういうものです！ 最終的な、正式の意味体系は当初期待したものよりも複雑で、実際のところ当初想定されたものとは大きく異なる形式になった

のですが、非公式な意味体系は明確、直感的です。これについてはこの第2回の記事で概略を説明する予定です。

同期化と可視性

たいていのプログラマーは`synchronized`と言うキーワードが、あるモニターが保護するブロックに一度に複数のスレッドが入り込むのを防ぐ相互排除（ mutex: mutual exclusion ）を強制することを知っています。ところが同期化には別の面もあり、（ JMMが規定する ）メモリの可視性に関する一定の規則も強制するのです。同期化は、同期化ブロックから出るときにはキャッシュがクリアされ、同期化ブロックに入るときにはキャッシュが無効化されることを保証するのです。こうすることで、あるモニターに保護された同期化ブロック期間中に一つのスレッドが書く値は、同じモニターに保護された同期化ブロックを実行している他のどのスレッドからも見えるのです。同期化はまた、コンパイラーが命令を同期化ブロック内から外へ移動することはない、ということも保証します（ 同期化ブロック外から内部への命令の移動は、場合によってはできますが ）。JMMは、同期化が無い時にはこの保証はしません。同じ変数に対して複数のスレッドがアクセスする時には必ず同期化（ またはその弟分、volatile ）を使う必要があるのは、この理由からです。

問題その1: 不変オブジェクトが実は不変ではない

JMMの驚くべき欠陥の一つに、不変オブジェクトの値が`final`キーワードの使用で不変性が保証されているはずにもかかわらず、変わるように見えてしまう事があります。（ お知らせ：オブジェクトの全フィールドを`final`にしたからといってオブジェクトが不変になるわけではありません。全てのフィールドは同時に基本型であるか、不変オブジェクトへの参照でもある必要があるのです。 ）不変オブジェクトは`String`と同様、同期化は必要無いはずです。ところがメモリへの書き込みでの変化をスレッドからスレッドへ伝達する際に遅れが生じる可能性があり、あるスレッドが最初に不変オブジェクトの値として見たものが、少し後で見ると違った値に見えてしまうという、レース状態が発生する可能性があるのです。

なぜこんなことが起きるのでしょう？ Sun 1.4 JDKにある`String`の実装を考えてみてください。ここには基本的に3つの重要な最終フィールドがあります。文字配列への参照と長さ、そして表現されている文字列の始まりを記述する、文字配列へのオフセットです。`String`は文字配列のみを持つのではなく、この方法で実装されます。これによって`String`が生成される度にテキストを新しい配列にコピーすることなく、文字配列が複数の`String`オブジェクトや`StringBuffer`オブジェクトの間で共有できるようになります。例えば`String.substring()`は、元々の`String`と同じ文字配列を共有し、長さとオフセット・フィールドのみ異なる新しいストリングを生成します。

次のコードを実行する事を考えてみてください。

```
String s1 = "/usr/tmp";
String s2 = s1.substring(4);   // contains "/tmp"
```

文字列`s2`はオフセット4、長さ4を持ちますが、同じ`"/usr/tmp"`を含む文字配列を`s1`と共有します。`String`コンストラクターが実行される前に、`Object`のコンストラクターは長さとオフセットの最終フィールドを含めて、全てのフィールドをデフォルト値で初期化します。`String`コンストラクターが実行されると、今度は長さとオフセットがそれぞれ必要な値に設定されます。ところが古いメモリ・モデルでは、同期化が無い時には別のスレッドから見た場合にオフセット・

フィールドが一時的にデフォルト値の0を持ち、その後で正しい値である4を持つように見えるのです。その結果として`s2`の値が`"/usr"`から`"/tmp"`に変わります。これは意図した事ではなく、すべてのJVM、すべてのプラットフォームで起きるわけではありませんが、古いメモリ・モデルの仕様では許されていたのです。

問題その2: volatile、非volatile保存のリオーダ

既存のJMMが非常に混乱した結果を引き起こす領域のもう一つが`volatile`フィールドに対するメモリ操作リオーダです。既存のJMMではvolatile読み込みと書き込みに関して、レジスターに値をキャッシュしたりプロセッサー固有のキャッシュをバイパスしたりすることを禁じ、直接メイン・メモリに行くように命令します。これによって複数のスレッドが常に、与えられた変数の最新の値を見られるようになります。ところが、`volatile`に関するこの定義は当初意図されたほど有用ではなく、実際の`volatile`の意味に関して大きな混乱を引き起こしたのです。

同期化が無い状態のパフォーマンスを上げるために、コンパイラーやランタイム、キャッシュは一般的に、現在実行中のスレッドに差が分からない限り、通常のメモリ操作をリオーダしても良い事になっています。（これは、within-thread as-if-serial semanticsと呼ばれます。）ところが、volatile読み込みと書き込みはスレッド全体に渡って完全に配列が決まっているのです。コンパイラーやキャッシュはお互いにvolatile読み込みと書き込みをリオーダしてはならないのです。残念ながらJMMは通常における変数の読み込みと書き込みに関して、volatile読み込みと書き込みのリオーダを許してしまうのです。つまり、どんな操作が完了したのかの表示としてvolatileフラグを使う事ができないのです。次のコードを考えてみてください。ここではvolatileフィールド`initialized`で初期化が完了したことを示そうとしています。

リスト1. volatileフィールドを「見張り」変数として使う

```
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
```

ここではvolatile変数`initialized`に、他の操作一式が完了した事を示す見張りとして動作させようとしたのです。これは良い考えなのですが、古いJMMの下ではうまく行きません。古いJMMでは、非volatile書き込み（例えば`configOptions`フィールドへの書き込みや、`configOptions`が参照する`Map`のフィールドへの書き込み）は、volatile書き込みでリオーダできてしまうのです。ですから別のスレッドが`initialized`を真（true）と見たとしても、`configOptions`フィールドの一定した現在のビューや、`configOptions`フィールドが参照するオブジェクトはまだ得られないかも知れないのです。`volatile`の古い意味体系では、読み書きされている変数の可視性のみを約束し、他の変数に関しては何の約束もしません。この手法は効率的に実装するのは簡単なのですが、当初想定されていたほど有用ではない事が分かったのです。

まとめ

Java言語仕様（Java Language Specification）の17章で規定されている通り、JMMにはいくつか深刻な問題があり、問題なく見えるようなプログラムにも、直感的でない、望ましくないことが起きてしまう可能性があります。並行クラスを適切に書くのがあまりにも難しいと、まず確実に、多くの並行クラスは予期したようには動作しませんし、確実にプラットフォームの問題と言える事になります。幸いメモリ・モデルとして、大部分の開発者が直感的に思うものにより近く、かつ古いメモリ・モデルの下で適切に同期化されたコードを壊さないようなものを生成する事は可能であり、JSR 133の過程で正にそれが成し遂げられたのです。次回は新しいメモリ・モデル（大部分は既に1.4 JDKに組み込まれています）の詳細を見ていきます。

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。