Javaの理論と実践: volatile を扱う

volatile 変数を使うためのガイドライン

Brian Goetz
Principal Consultant
Quiotix

2007年 6月 19日

Java™ 言語に元々含まれている同期機構として、synchronized ブロック (およびメソッド) と volatile 変数の 2 つがあります。どちらも、スレッド・セーフにコードを描画するために提供されています。2 つのうち、volatile 変数の方が同期の仕組みとしては強力でない (しかし場合によると、より単純、あるいはコストが安い) のですが、誤った使い方をしやすいものでもあります。今回の Java の理論と実践では、Brian Goetz が volatile 変数を正しく使うためのパターンをいくつか調べ、volatile 変数を適用できる限界について注意を促します。

このシリーズの他の記事を見る

Java 言語での volatile 変数は、「synchronized の軽量版」と考えることができます。volatile 変数を使うために必要なコーディングは synchronized ブロックを使う場合よりも少なく、また多くの場合は volatile 変数の方が実行時のオーバーヘッドが少ないのですが、volatile 変数を使うと、synchronized を使ってできることの一部しかできません。この記事では、volatile 変数を効果的に使うためのパターンをいくつか説明し、volatile 変数を使うべきではない場合について注意を促します。

ロックには、相互排除と可視性という、基本となる 2 つの特徴があります。相互排除は、指定されたロックを保持できるのは 1 度に 1 つのスレッドのみであることを意味します。そしてこの性質を使うと、1 度に 1 つのスレッドしか共有データを使用できないように共有データへのアクセスを調整するプロトコルを実装することができます。可視性はもっと芸が細かく、ロックを解放する前に共有データに加えられた変更が、次にそのロックを取得する別のスレッドから見えるように保証します。同期によって可視性が保証されないと、スレッドは共有変数の値として、古い、あるいは一貫性のない値を見てしまうかもしれません。そうなると、深刻な問題が大量に発生する可能性があります。

volatile 変数

volatile 変数は、可視性という特徴は synchronized と同じですが、synchronized のようなアトミック性を持っていません。これはつまり、スレッドは自動的に volatile 変数の最新の値を見るということ

です。スレッド・セーフを提供するために volatile 変数を使うことはできますが、使える場合は非常に限定されます。つまり、複数の変数の間か、または 1 つの変数の現在の値と将来の値との間に、制約が課されない場合にしか使うことができません。そのため、複数の変数を関連付ける不変式 (例えば “start <=end” など) を持つカウンターや ミューテックス、その他のクラスを実装するためには、volatile のみでは十分に強力とは言えません。

ロックの代わりに volatile 変数を使いたい主な理由としては、単純さ、あるいはスケーラビリティーという 2 つのいずれかでしょう。一部のイディオムは、ロックではなく volatile 変数を使った方がコーディングしやすく、また読みやすくなります。また volatile 変数は (ロックと違い) スレッドをブロックできないため、スケーラビリティーの問題を起こしにくくなります。書き込みよりも読み取りの方が圧倒的に多い場合にも、ロックよりも volatile 変数を使った方がパフォーマンスの点で有利な可能性があります。

volatile を正しく使うための条件

ロックの代わりに volatile 変数を使えるのは、非常に限定された状況に限られます。必要なスレッド・セーフを volatile 変数で提供するためには、下記の条件を両方とも満足する必要があります。

- その変数への書き込みが、その変数の現在の値に依存しない。
- その変数が、他の変数との不変式に使われていない。

この 2 つの条件に記されていることは、基本的に volatile 変数に書き込める有効な値は (その変数の現在の状態を含め) 他のプログラムの状態には依存しないということです。

最初の条件から、volatile 変数をスレッド・セーフなカウンターとして使うことはできません。インクリメント演算 (`x++`) は 1 つの演算のように見えますが、実際には「読み取り、変更、書き込み」という連続した複合演算であり、アトミックに実行する必要があります。しかし volatile 変数には必要なアトミシティがありません。演算が適切であるためには、演算中に `x` の値は不変でなければなりませんが、これを volatile 変数を使って実現することはできません。(ただし、その値が 1 つのスレッドからしか書き込めないようにしてあれば、最初の条件を無視することができます。)

大部分のプログラミング状況は、最初の条件あるいは 2 番目の条件に違反するため、スレッド・セーフを実現するための方法としては、volatile 変数は `synchronized` ほど一般的には適用できません。リスト 1 は、スレッド・セーフではない、数字範囲のクラスを示しています。このクラスは不変式を含んでおり、下限は必ず上限以下です。

リスト 1. スレッド・セーフではない数字範囲クラス

```
@NotThreadSafe
public class NumberRange {
   private int lower, upper;

   public int getLower() { return lower; }
   public int getUpper() { return upper; }

   public void setLower(int value) {
      if (value > upper)
         throw new IllegalArgumentException(...);
      lower = value;
   }

   public void setUpper(int value) {
      if (value < lower)
         throw new IllegalArgumentException(...);
      upper = value;
   }
}
```

範囲の状態変数はこのように制約されているため、lower フィールドと upper フィールドを volatile にするだけではクラスをスレッド・セーフにするには不十分であり、やはり同期が必要です。そうでないと、ある運の悪いタイミングでは、setLower と setUpper を実行する (一貫性のない値を持つ) 2 つのスレッドによって、範囲に一貫性がなくなります。例えば最初の状態が ((0, 5)) だったとし、スレッド A が setLower(4) を呼び出すと同時にスレッド B が setUpper(3) を呼び出すとすると、演算は途中で割り込まれた形で不適切に実行され、どちらの演算も、不変式を守るはずのチェックにパスしてしまいます。その結果、範囲は (4, 3) という無効な値を保持することになります。setLower() と setUpper() の演算を、範囲に対する他の演算との関係から見てアトミックにする必要があります。しかしフィールドを volatile にしても、アトミックにはなりません。

パフォーマンスに関する考慮事項

volatile 変数を使う主な動機は、単純さです。場合によると volatile 変数を使った方が、対応するロックを使うよりも遥かに単純になります。volatile 変数を使う 2 番目の動機として、パフォーマンスがあります。場合によると、同期の機構として、ロックよりも volatile 変数の方がパフォーマンスが高いかもしれません。

「X は常に Y よりも高速である」という形で、正確かつ一般的に言うことは、特に JVM に本質的な動作に関しては、非常に困難です。(例えば、ある状況では VM はロックを完全に排除できるかもしれません。その場合には、抽象的に volatile と synchronized の相対コストを語るのは難しくなります。) とは言え、現在の大部分のプロセッサーのアーキテクチャーでは、volatile の読み取りは nonvolatile の読み取りとほとんど同じ程度に低コストです。一方 volatile の書き込みは nonvolatile の書き込みよりも大幅に高コストです。これは可視性を保証するためにメモリー・フェンスの動作が要求されるためですが、それでも volatile 書き込みはロック取得よりも一般的には低コストです。

ロックの場合とは異なり、volatile 演算はブロック動作を行いません。そのため volatile は、安全に使用できる場合には、ロックよりも少しスケーラビリティーが優れています。書き込みよりも読み取りが圧倒的に多い場合には、volatile 変数は大抵、ロックと比較して同期のパフォーマンス・コストを低下させます。

volatile を正しく使うためのパターン

並行処理のエキスパートの多くは、ユーザーがまったく volatile 変数を使わないように指導しがちです。これは、volatile 変数を正しく使おうとすると、ロックよりも volatile の方が難しいためです。しかし、明確なパターンが、いくつか存在しています。注意深くそれらのパターンに従えば、それらのパターンを非常にさまざまな状況で安全に使うことができます。ただし、どのような場合に volatile を使用できるか、その制限についての規則を常に念頭に置く必要があります。つまり、プログラム中の他のいかなるものにもまったく依存しない状態に対してのみ、volatile を使用できるのです。この規則を守れば、これらのパターンを危険な領域にまで拡張しようとしなくなるはずです。

パターン #1: ステータス・フラグ

volatile 変数の、おそらく最も正統的な使い方は、1 度だけ発生する重要なライフサイクル・イベントの発生 (初期化が完了した、あるいはシャットダウンが要求された、など) を示す、単純なブール値のステータス・フラグとして使う方法です。

多くのアプリケーションには、「シャットダウンの準備ができていないので、さらに作業をしなさい」という形式の制御構成体が含まれています (リスト 2)。

リスト 2. variable 変数をステータス・フラグとして使う

```
volatile boolean shutdownRequested;

...

public void shutdown() { shutdownRequested = true; }

public void doWork() {
   while (!shutdownRequested) {
      // do stuff
   }
}
```

shutdown() メソッドは、ループ外のどこかから (別のスレッドで) 呼び出される可能性があります。従って、shutdownRequested 変数の可視性を適切に保証するためには何らかの形式の同期が必要です。(shutdown() メソッドは、GUI イベント・スレッドのアクション・リスナーである JMX リスナーから、RMI や Web サービスなどを通じて呼び出されるかもしれません。) しかし synchronized ブロックを使ってループをコーディングしようとすると、リスト 2 のように volatile ステータス・フラグを使ってコーディングするよりも、ずっと面倒です。volatile によってコーディングが単純化され、またステータス・フラグはプログラム中の他の状態にはまったく依存しないため、これは volatile の使い方として適切です。

このタイプのステータス・フラグの一般的な特徴の 1 つとして、通常は状態遷移が 1 通りしかありません。shutdownRequested フラグは false から true になり、そしてプログラムがシャットダウンします。セットされたり、リセットされたりする状態フラグにこのパターンを拡張することはできますが、それは遷移サイクル (false から true へ、そして true から false へ) が検出されない場合のみです。それ以外の場合には、何らかのアトミックな状態遷移の仕組み (例えばアトミック変数など) が必要です。

パターン #2: 1 度だけ安全に公開する

同期していない場合には可視性に問題が生じることがありますが、プリミティブ値ではなくオブジェクト参照に書き込む場合には、その問題の原因が一層探りにくくなります。同期していない場合には、オブジェクト参照の最新の値を見たとしても、それは別のスレッドが書き込んだ値であり、そのオブジェクトの状態の古い値を見ている可能性があります。(この危険性は、オブジェクト参照が同期することなく読み取られる、悪名高いダブルチェック・ロック・イディオムの根本原因です。つまり、最新の参照を見ているにもかかわらず、その参照によって、部分的に作成されたオブジェクトを見てしまうという危険性です。)

安全にオブジェクトを公開するための 1 つの方法は、オブジェクト参照を volatile にする方法です。リスト 3 に示す例では、起動中にバックグラウンド・スレッドがデータベースからデータをロードします。他のコードは、このデータを利用できそうな場合には、そのデータを使おうとする前にデータが公開されていたかどうかを確認します。

リスト 3. 安全に 1 度だけ公開するために volatile 変数を使う

```
public class BackgroundFloobleLoader {
   public volatile Flooble theFlooble;

   public void initInBackground() {
      // do lots of stuff
      theFlooble = new Flooble();  // this is the only write to theFlooble
   }
}

public class SomeOtherClass {
   public void doWork() {
      while (true) {
         // do some stuff...
         // use the Flooble, but only if it is ready
         if (floobleLoader.theFlooble != null)
            doSomething(floobleLoader.theFlooble);
      }
   }
}
```

theFlooble 参照が volatile ではないと、doWork() のコードは、theFlooble 参照を参照解除する際に、部分的にしか作成されていない Flooble を見てしまう危険があります。

このパターンの鍵となる要件は、公開されるオブジェクトが、スレッド・セーフであるか、あるいは実質的に不変であることです (実質的に不変という意味は、そのオブジェクトの状態が、そのオブジェクトが公開された後は変更されない、ということです)。volatile 参照は、公開された形式でのオブジェクトの可視性を保証するかもしれませんが、もしそのオブジェクトの状態が公開後に変化するのであれば、さらに同期する必要があります。

パターン #3: 独立した観測結果

volatile を安全に使える、もう 1 つの単純なパターンは、プログラム内で使用するために観測結果が定期的に「公開される」場合です。例えば、現在の温度を感知する環境センサーがある場合です。バックグラウンド・スレッドは、このセンサーを数秒ごとに読み取り、現在の温度を含む volatile 変数を更新するかもしれません。そうすると他のスレッドは、常に最新の値が見えていることを知った上で、この変数を読み取ることができます。

このパターンの、もう 1 つの応用として、プログラムに関する統計を収集する場合があります。リスト 4 は、最後にログオンしたユーザーの名前を認証メカニズムが記憶する方法を示しています。lastUser 参照は、この値を公開してプログラムの他の部分で使用できるように、繰り返し使われます。

リスト 4. 独立した観測結果を複数公開するために volatile 変数を使う

```
public class UserManager {
   public volatile String lastUser;

   public boolean authenticate(String user, String password) {
      boolean valid = passwordIsValid(user, password);
      if (valid) {
         User u = new User();
         activeUsers.add(u);
         lastUser = user;
      }
      return valid;
   }
}
```

このパターンは、この前のパターンの拡張です。つまり値はプログラム内の他の場所で使うために公開されますが、公開は 1 度のみのイベントではなく、一連の独立したイベントです。このパターンでは、公開される値が実質的に不変である (つまり値の状態が公開後変化しない) ことが必要です。この値を利用するコードは、この値が随時変化する可能性があることを認識する必要があります。

パターン #4: 「volatile bean」のパターン

volatile beanのパターンは、JavaBeans を「見せかけの構造体 (glorified struct) (※訳注: メンバーが public であり、構造体と変わらないような単純な構成をしたクラスのことを指すようです)」として使用するフレームワークに適用できます。volatile beanのパターンでは、JavaBean はゲッターやセッターを持つ独立したプロパティー・グループのコンテナーとして使われます。volatile beanのパターンを使う理由は、多くのフレームワークが可変データ・ホルダーのコンテナー (例えば HttpSession など) を提供する一方、こうしたコンテナーに置かれるオブジェクトはスレッド・セーフでなければならないためです。

volatile beanのパターンでは、JavaBean のすべてのデータ・メンバーは volatile であり、そしてゲッターとセッターは単純なものである必要があります (ゲッターとセッターには、適当なプロパティーを取得したり設定したりするためのロジック以外は何も含んではいけません)。さらに、オブジェクト参照であるデータ・メンバーにとって、参照されるオブジェクトは実質的に不変でなければなりません。(このため、配列の値を持つプロパティーを持つことができません。これは、配列参照が volatile と宣言されると、要素そのものではなく参照のみしか volatile として認識されないためです。) volatile 変数の場合と同じく、JavaBean のプロパティーが関係する不変式や制約はないかもしれません。volatile beanのパターンに従う JavaBean の例をリスト 5 に示します。

リスト 5. volatile beanのパターンに従う Person オブジェクト

```
@ThreadSafe
public class Person {
   private volatile String firstName;
   private volatile String lastName;
   private volatile int age;
```

```
    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public int getAge() { return age; }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

volatile の高度なパターン

先のセクションで説明したパターンは、volatile を使うことが妥当で直接的である、基本的なケースの大部分を網羅しています。このセクションでは、より高度なパターンとして、volatile を使うことでパフォーマンスやスケーラビリティーを高められる場合を調べます。

volatile を使った、より高度なパターンは、非常に気をつけて扱わなければならないかもしれません。ごくわずかな変更でもコードを実行できなくなる可能性があるため、前提条件を注意深く文書化し、パターンを強力にカプセル化しておくことがとても重要になります。また、高度な方法で volatile を使う主な理由がパフォーマンスであることを考えると、volatile を適用する前に、そのようにパフォーマンスを向上させる必要性を明確に示す必要があります。これらのパターンでは、パフォーマンス向上の可能性と引き換えに、読みやすさや維持管理のしやすさが犠牲になります。もしパフォーマンスの向上が必要なければ (あるいは、パフォーマンスの向上が必要なことを厳密な測定プログラムで証明できなければ)、価値あるものを失う代わりに価値のないものを得ることになるため、この選択はおそらく適切ではありません。

パターン #5: 安価な読み書きロックの秘訣

ここまで来ると、volatile はカウンターを実装するのに十分なほど強力ではないことがよく理解できると思います。`++x` は実際には 3 つの演算 (読み取りと追加と保存) の簡略表現なので、もし複数のスレッドが volatile カウンターを同時にインクリメントしようとすると、タイミングが悪いと更新に失敗する可能性があります。

しかし、変更よりも読み取りが圧倒的に多い場合には、本来のロックと `volatile` 変数を組み合わせることで、一般的なコード・パスのコストを削減することができます。リスト 6 はスレッド・セーフなカウンターを示しています。このカウンターは、`synchronized` を使うことでインクリメント演算がアトミックなことを保証し、また volatile を使うことで現在の結果の可視性を保証しています。更新が頻繁ではない場合には、読み取りパスのオーバーヘッドが volatile の読み取り (競合のないロック取得よりも一般的にはコストが低い) のみであるため、この方法のパフォーマンスの方が高いかもしれません。

リスト 6. volatile と **synchronized** を組み合わせ、「安価な読み書きロック」を構成する

```
@ThreadSafe
public class CheesyCounter {
   // Employs the cheap read-write lock trick
   // All mutative operations MUST be done with the 'this' lock held
   @GuardedBy("this") private volatile int value;

   public int getValue() { return value; }

   public synchronized int increment() {
      return value++;
   }
}
```

この方法が「安価な読み書きロック」と呼ばれる理由は、読み取りと書き込みに異なる同期機構が使われているためです。この場合の書き込みは、volatile を使うための最初の条件に違反しているため、volatile を使って安全にカウンターを実装することはできず、ロックを使う必要があります。しかし volatile を使うことで、読み取りの際に現在の値の可視性を保証することができます。そのため、すべての変更される可能性がある操作にはロックを使い、読み取りのみの操作には volatile を使うのです。ロックでは、一度にある値にアクセスできるのは 1 つのスレッドのみですが、volatile の読み取りでは複数のスレッドからのアクセスが許可されます。そのため、volatile を使って読み取りのコード・パスを保護する際には、すべてのコード・パスにロックを使った場合よりも共有の度合いを高めることができます (ちょうど読み書きロックを使った場合と同じです)。ただし、このパターンの扱いには注意をする必要があることを忘れないでください。2 つの競合する同期機構があるため、このパターンの最も基本的な適用の仕方以上のことをしようとすると、この方法は非常に面倒になります。

まとめ

volatile 変数は同期の形式として、ロックよりも単純ですが、強力ではありません。しかし、場合によると、本来のロックよりも優れたパフォーマンスやスケーラビリティーを発揮することがあります。volatile を安全に使うための条件に従えば (つまり変数が、その変数自体の以前の値にも他の変数にもまったく依存しないのであれば)、`synchronized` の代わりに volatile を使うことでコードを単純化できることがあります。しかし多くの場合、`volatile` を使うコードはロックを使うコードよりも注意をして扱う必要があります。ここで説明したパターンは、`synchronized` よりも `volatile`の方が実用にかなっている、最も一般的な場合を示しています。これらのパターンに従うことで、そしてその限界を越えないように注意することで、`volatile` 変数の方が有効な場合の大部分に安全に対応できるはずです。

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。