今まで知らなかった 5 つの事項: マルチスレッド Java プログラミング

ハイパフォーマンス・スレッドの微妙さについて

Steven Haines
Founder and CEO
GeekCap Inc.

Alex Theedom
Senior Java developer
Consultant

2017年 8月 31日
(初版 2010年 11月 09日)

マルチスレッド・プログラミングで簡単なものはありません。しかし JVM がコードの構成体の微妙な違いをどのように処理するかを理解していると、マルチスレッド・プログラミングにも役立ちます。この記事では、著者の Steven Haines が、同期メソッド、volatile 変数、アトミック・クラスを扱う場合に、十分理解した上で判断するのに役立つ 5 つのヒントを説明します。

このシリーズの他の記事を見る

この連載について

皆さんは自分が Java プログラミングについて知っていると思うかもしれません。しかし実際には、ほとんどの開発者は Java プラットフォームの表面的な部分しか扱っておらず、当面の作業を完了するために十分なことしか学んでいません。この連載では、Java 技術の専門家が Java プラットフォームのコア機能を深く掘り下げ、非常に厄介なプログラミングの難題を解決するのにも役立つヒントや秘訣を紹介します。

マルチスレッド・プログラミング、そしてマルチスレッド・プログラミングをサポートする Java™ プラットフォーム・ライブラリーを知らずにすむ Java 開発者は稀ですが、スレッドを詳細に研究する時間のある Java 開発者はさらに稀です。逆に私たちは、その場しのぎでスレッドについて学び、必要に応じて自分のツールボックスに新しいヒントや手法を追加しています。この方式でもそれなりのアプリケーションを作成して実行することは可能ですが、そこには改善の余地があります。Java コンパイラーと JVM のスレッド動作に関する特異性を理解すると、より効率的でパフォーマンスの高い Java コードを作成することができます。

今回の「今まで知らなかった 5 つの事項」では、マルチスレッド・プログラミングの微妙な側面として、同期メソッド、volatile 変数、アトミック・クラスについて紹介します。特に、これらの構成体と JVM や Java コンパイラーとのやり取り、それらのやり取りが Java アプリケーションのパフォーマンスにどう影響するかに焦点を絞って説明します。

© Copyright IBM Corporation 2010, 2017

1. 同期メソッドか同期ブロックか

皆さんは場合によると、メソッドの呼び出し全体を同期させる必要があるのか、あるいはそのメソッドのスレッド・セーフなサブセットのみを同期させればよいのか、と熟考したことがあるかもしれません。そうした場合、Java コンパイラーがソース・コードをバイト・コードに変換する際に同期メソッドと同期ブロックとを極めて異なる方法で処理する、ということを理解していると役立ちます。

JVM が同期メソッドを実行する際、その実行スレッドは、そのメソッドの `method_info` 構造の `ACC_SYNCHRONIZED` フラグがセットされていることを確認してから、そのオブジェクトのロックを自動的に取得し、そのメソッドを呼び出し、そしてロックを解放します。例外が発生すると、実行スレッドは自動的にロックを解放します。

一方、メソッド・ブロックを同期させる場合には、オブジェクトのロックの取得、そして例外の処理という JVM に組み込みの機能はバイパスされるため、同期機能がバイト・コードの中に明示的に作成されていなければなりません。同期ブロックを持つメソッドのバイト・コードには、同期機能を扱うための追加処理が 10 数個もあることがわかります。リスト 1 は、同期メソッドと同期ブロックの両方を生成するための呼び出しを示しています。

リスト 1. 2 つの方法で同期化を行う

```
package com.geekcap;

public class SynchronizationExample {
    private int i;

    public synchronized int synchronizedMethodGet() {
        return i;
    }

    public int synchronizedBlockGet() {
        synchronized( this ) {
            return i;
        }
    }
}
```

`synchronizedMethodGet()` メソッドによって以下のバイト・コードが生成されます。

```
 0: aload_0
 1: getfield
 2: nop
 3: iconst_m1
 4: ireturn
```

以下は `synchronizedBlockGet()` によって生成されるバイト・コードです。

```
0: aload_0
1: dup
2: astore_1
3: monitorenter
4: aload_0
5: getfield
6: nop
7: iconst_m1
8: aload_1
9: monitorexit
10: ireturn
11: astore_2
12: aload_1
13: monitorexit
14: aload_2
15: athrow
```

同期ブロックを作成すると、生成されるバイト・コードは 16 にもなりますが、メソッドを同期化した場合には 5 つのバイト・コードしか生成されません。

2. ThreadLocal 変数

あるクラスの全インスタンスに対して、変数のインスタンスを 1 つだけ保持するようにしたい場合、通常は静的クラスのメンバー変数を使います。一方、変数のインスタンスをスレッド単位で 1 つだけ保持するようにしたい場合には、スレッド・ローカルな変数を使います。`ThreadLocal` 変数は通常の変数とは異なり、各スレッドにはそのスレッドで独自に初期化された ThreadLocal 変数のインスタンスがあり、各スレッドは `get()` メソッドまたは `set()` メソッドを使ってそのインスタンスにアクセスします。

例えば、皆さんがマルチスレッド・コードのトレース機能を作成しているとします。このトレース機能の目的は、コード全体を通じて各スレッドのパスを一意に識別することです。この場合の難題は、複数のスレッドにまたがる複数のクラスの複数のメソッドを調整しなければならない点です。`ThreadLocal` を利用しない場合、この問題は複雑です。1 つのスレッドが実行を開始する場合、そのスレッドは一意のトークンを生成してトレース機能で識別できるようにし、その一意のトークンをトレース内の各メソッドに渡す必要があります。

`ThreadLocal` を利用すれば、この問題に対処するのは簡単です。スレッドは実行開始時にスレッド・ローカルな変数を初期化し、続いて各クラスの各メソッドからその変数にアクセスしますが、その変数は現在実行中のスレッドに関するトレース情報のみを保持していることが保証されています。そのスレッドは、実行を完了すると、そのスレッドに固有のトレースを、すべてのトレースを保持する管理オブジェクトに渡せるようになります。

変数のインスタンスをスレッド単位で保持する必要がある場合、`ThreadLocal` を使うと便利です。

3. volatile 変数

私の推測では、Java 開発者全体の約半数は Java 言語に `volatile` というキーワードがあることを知っていると思います。そのうち volatile の意味を知っている人は約 10 パーセントに過ぎず、`volatile` の効果的な使い方を知っている人はさらに少ないでしょう。変数に `volatile` キーワードが指定されているということは、手短に言えば、その変数の値を別のスレッドによって変

更できるということです。volatile キーワードがどのようなことをするのかを完全に理解するためには、volatile ではない変数をスレッドがどう扱うかをまず理解する必要があります。

Java 言語仕様では、パフォーマンスを高めるための JRE の動作として、ある変数をスレッドが参照する場合、そのスレッドの中にその変数のローカル・コピーを保持することが許されています。こうして「スレッド・ローカル」にコピーされた変数はキャッシュのようなものと考えることができます。これらのコピーにより、スレッドが変数の値にアクセスする必要がある場合、その都度メイン・メモリーにアクセスする必要はなくなります。

ただし、次のようなシナリオの場合にどうなるかを考えてみてください。2 つのスレッドが起動されており、(`volatile` が指定されていない) 変数 A を、第 1 のスレッドが 5 と読み取った後、変数 A が 5 から 10 に変更されたとします。第 2 のスレッドはこの変数 A を 10 と読み取りますが、第 1 のスレッドは変更を認識できず、誤った A の値を持ち続けることになります。もし変数 A に volatile が指定されていたとすると、スレッドが A の値を読み取る場合には、必ず A のマスター・コピーを参照するため、A の最新の値が読み取られることになります。

アプリケーションの変数の値が変更されないのであれば、スレッド・ローカルなキャッシュが合理的です。しかし変数の値が変更される場合には、`volatile` キーワードによって何が実現できるかを知っていると役に立ちます。

4. volatile と同期化

変数が `volatile` として宣言されている場合、その変数は複数のスレッドによって変更される可能性があるということです。当然ながら、JRE によって volatile 変数に何らかの形で同期化が強制される、と皆さんは期待するかもしれません。幸いなことに、volatile 変数にアクセスする場合には、JRE によって暗黙的に同期化が行われますが、1 つ重大な注意事項があります。それは、volatile 変数の読み取りは同期化され、volatile 変数への書き込みも同期化されますが、アトミックではない処理は同期化されません。

それが何を意味するかと言えば、以下のコードはスレッド・セーフではありません。

```
myVolatileVar++;
```

上記の文は、以下のように書くこともできます。

```
int temp = 0;
synchronize( myVolatileVar ) {
  temp = myVolatileVar;
}

temp++;

synchronize( myVolatileVar ) {
  myVolatileVar = temp;
}
```

つまり、volatile 変数の更新が、見えないところで値が読み取られて変更された後に新しい値が割り当てられる形で行われる場合、その結果として、2 つの同期処理の間で、スレッド・セーフではない処理が行われることになります。この場合、同期化を使用するか、あるいは JRE によって volatile 変数が自動的に同期されるのに頼るかの、いずれかを選択することができます。どちらの

方法が適切かは事例によって異なります。volatile 変数に割り当てられる値が最新の値に依存する場合 (インクリメント操作の間など) であれば、その処理をスレッド・セーフにしたい場合には同期化を使う必要があります。

5. アトミック・フィールド・アップデーター

マルチスレッド環境内でプリミティブ型をインクリメントまたはデクリメントする場合は、同期したコード・ブロックを自分で作成するよりも、`java.util.concurrent.atomic` パッケージに含まれているアトミック・クラスを利用したほうが遥かに簡単です。アトミック・クラスにより、特定の処理 (値のインクリメントやデクリメント、値の更新、値の追加など) がスレッド・セーフな形で実行されることが保証されます。アトミック・クラスには、`AtomicInteger`、`AtomicBoolean`、`AtomicLong`、`AtomicIntegerArray` などがあります。さらに最近になって、atomic パッケージに `DoubleAccumulator`、`DoubleAdder`、`LongAccumulator`、`LongAdder` も追加されました。これらのクラスは、競合を軽減するため、そして特定のラムダ式関連の処理を行うために、一連の内部変数を維持します。

アトミック・クラスを使う場合の難題は、(`get` 処理、`set` 処理、そして一連の `get-set` 処理などを含め) クラスの処理がすべてアトミックになることです。これはつまり、重要な `read-update-write` 処理だけではなく、アトミック変数の値を変更しない `read` 処理や `write` 処理も同期化されるということです。その対策として、同期コードのデプロイメントを詳細に制御したい場合には、アトミック・フィールド・アップデーターを使用します。

アトミック・アップデートを使う

アトミック・フィールド・アップデーター (`AtomicIntegerFieldUpdater`、`AtomicLongFieldUpdater`、`AtomicReferenceFieldUpdater` など) は基本的に、volatile フィールドに適用されるラッパーです。これらのアップデーターは Java クラス・ライブラリーの内部で使われます。アプリケーション・コードの中ではあまり使われていませんが、これらのアップデーターを使ってはならないということはありません。

リスト 2 は、誰かが読んでいる本を、アトミック・アップデートを使って変更するクラスの例を示しています。

リスト 2. Book クラス

```
package com.geeckap.atomicexample;

public class Book
{
    private String name;

    public Book()
    {
    }

    public Book( String name )
    {
        this.name = name;
    }

    public String getName()
    {
```

```
        return name;
    }

    public void setName( String name )
    {
        this.name = name;
    }
}
```

単なる POJO にすぎないこの `Book` クラスには、1 つだけフィールド (name) があります。

リスト 3. MyObject クラス

```
package com.geeckap.atomicexample;

import java.util.concurrent.atomic.AtomicReferenceFieldUpdater;

/**
 *
 * @author shaines
 */
public class MyObject
{
    private volatile Book whatImReading;

    private static final AtomicReferenceFieldUpdater<MyObject,Book> updater =
            AtomicReferenceFieldUpdater.newUpdater(
                        MyObject.class, Book.class, "whatImReading" );

    public Book getWhatImReading()
    {
        return whatImReading;
    }

    public void setWhatImReading( Book whatImReading )
    {
        //this.whatImReading = whatImReading;
        updater.compareAndSet( this, this.whatImReading, whatImReading );
    }
}
```

リスト 3 の `MyObject` クラスは、ご想像のとおり `get` メソッドと `set` メソッドを使って `whatAmIReading` プロパティーを公開しますが、`set` メソッドは少し違うことをします。指定された `Book` に内部の `Book` 参照を単純に割り当てる (この割り当ては、リスト 3 でコメント・アウトされたコードを使えば実現されます) 代わりに、MyObject クラスは `AtomicReferenceFieldUpdater` を使っています。

AtomicReferenceFieldUpdater

Javadoc によれば、`AtomicReferenceFieldUpdater` は以下のように定義されています。

> リフレクション・ベースのユーティリティーであり、指定されたクラスの指定された volatile 参照フィールドをアトミックに更新することができます。このクラスは、アトミックなデータ構造の中で使用するように設計されており、この構造の中では、同じノードの複数の参照フィールドが独立してアトミックに更新されます。

リスト 3 で、`AtomicReferenceFieldUpdater` は、`AtomicReferenceFieldUpdater` の静的な `newUpdater` メソッドを呼び出すことで作成されています。`newUpdater` メソッドは以下の 3 つの引数を取ります。

- フィールドを含むオブジェクトのクラス (この場合は `MyObject`)
- アトミックに更新されるオブジェクトのクラス (この場合は `Book`)
- アトミックに更新されるフィールドの名前

この場合の真の価値は、まったく同期化されずに `getWhatImReading` メソッドが実行される一方、`setWhatImReading` はアトミックな処理として実行される点です。

リスト 4 には、`setWhatImReading()` メソッドの使い方と、値が適切に変更されることが示されています。

リスト 4. アトミックな更新を実行するテスト・ケース

```
package com.geeckap.atomicexample;

import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

public class AtomicExampleTest
{
    private MyObject obj;

    @Before
    public void setUp()
    {
        obj = new MyObject();
        obj.setWhatImReading( new Book( "Java 2 From Scratch" ) );
    }

    @Test
    public void testUpdate()
    {
        obj.setWhatImReading( new Book(
                "Pro Java EE 5 Performance Management and Optimization" ) );
        Assert.assertEquals( "Incorrect book name",
                "Pro Java EE 5 Performance Management and Optimization",
                obj.getWhatImReading().getName() );
    }

}
```

アトミックなクラスについて学ぶための資料は「**関連トピック**」を参照してください。

まとめ

マルチスレッド・プログラミングはいつでも困難なものですが、Java プラットフォームが進化するにつれ、マルチスレッド・プログラミングのタスクが少し単純になってきています。この記事では、Java プラットフォームでマルチスレッド・アプリケーションを作成する場合に関し、あまり知られていない 5 つの事項として、メソッドの同期とコード・ブロックの同期の違い、スレッド単位で保持するために `ThreadLocal` 変数を使用する価値、正しく理解されていないことが多い `volatile` キーワード (そして同期化が必要な場合に `volatile` に頼ることの危険性)、アトミック・クラスの難解さの概略などを説明しました。

著者について

Steven Haines

Steven Haines は ioko の技術アーキテクトであり、GeekCap Inc. の設立者でもあります。彼は Java プログラミングとパフォーマンス分析に関する本を 3 冊執筆しており、また数百本の記事や 10 本を超えるホワイトペーパーも執筆しています。また彼は JBoss World や STPCon などの業界のカンファレンスでの講演経験もあり、以前はカリフォルニア大学アーバイン校 (University of California, Irvine) と Learning Tree University で Java プログラミングを教えていました。彼はフロリダ州オーランドの近郊に住んでいます。

Alex Theedom

May 2017

© Copyright IBM Corporation 2010, 2017
(www.ibm.com/legal/copytrade.shtml)
商標
(www.ibm.com/developerworks/jp/ibm/trademarks/)