double-checked lockingとSingletonパターン

この破たんしたプログラミング・イディオムを多角的に検討する

Peter Haggar
Senior Software Engineer
IBM

2002年 5月 01日

すべてのプログラム言語には、それぞれのイディオムがあります。それらの多くは、知っておいて損がなく、使う価値があるもので、プログラマーたちは、それらの作成、学習、および実装に貴重な時間を費やしています。問題は、いくつかのイディオムが、触れ込みとは異なっていたり、説明どおりに機能しなかったりすることが、後になって判明することです。Javaプログラム言語には、いくつかの便利なプログラミング・イディオムが含まれています。一方で、その後の研究により、使用すべきでないことが明らかになったイディオムも、いくつかあります。double-checked lockingは、そうした、決して使用すべきでないイディオムの1つです。Peter Haggar氏はこの記事で、double-checked lockingイディオムのルーツを探り、それがなぜ開発されたのか、また、なぜ機能しないのかを検討します。

Singleton作成パターンは、一般的なプログラミング・イディオムの1つです。複数のスレッドで使用する場合には、なんらかのタイプの同期化を使用する必要があります。Javaプログラマーたちは、より効率の高いコードを作成することを目指して、コードの同期化を減らすためにSingleton作成パターンで使用する、double-checked lockingイディオムを作成しました。しかし、ほとんど知られていないJavaメモリー・モデルの詳細が原因で、このdouble-checked lockingイディオムは、動作が保証されていません。常に障害が起こるわけではなく、散発的に障害が起こるのです。さらに、その障害の理由が分かりにくく、Javaメモリー・モデルのささいな事項に関連しています。こうしたことがあるため、double-checked lockingに起因するコード障害を調べることは、きわめて困難です。この記事の、これ以降の部分では、double-checked lockingイディオムがどこで破たんするのかを理解するために、このイディオムについて詳しく検討します。

Singleton作成イディオム

double-checked lockingイディオムの起源を知るためには、一般的なSingleton作成イディオムを理解する必要があります。リスト1に、このイディオムを示します。

リスト1. Singleton作成イディオム

```
import java.util.*;
class Singleton
{
  private static Singleton instance;
  private Vector v;
  private boolean inUse;
  private Singleton()
  {
    v = new Vector();
    v.addElement(new Object());
    inUse = true;
  }
  public static Singleton getInstance()
  {
    if (instance == null)          //1
      instance = new Singleton();  //2
    return instance;               //3
  }
}
```

このクラスの設計では、`Singleton` オブジェクトが一つだけ作成されるようになっています。このコンストラクターは`private` と宣言されていて、`getInstance()` メソッドは1つのオブジェクトだけを作成します。このような実装は、シングルスレッド・プログラムの場合には問題ありません。しかし、複数のスレッドが導入される場合には、`getInstance()` メソッドを同期化によって保護しなければなりません。`getInstance()` メソッドが保護されていないと、`Singleton` オブジェクトの2つの異なるインスタンスが戻される可能性があります。ここで、2つのスレッドが同時に`getInstance()` メソッドを呼び出し、さらに以下の一連のイベントが発生するものとします。

1. スレッド1が、`getInstance()` メソッドを呼び出し、//1で`instance` が`null` であることを判別します。
2. スレッド1は`if` ブロックに入りますが、//2の行を実行する前に スレッド2に取って代わられます。
3. スレッド2は、`getInstance()` メソッドを呼び出し、//1で`instance` が`null` であることを判別します。
4. スレッド2は`if` ブロックに入り、新しい`Singleton` オブジェクトを作成して、//2でこの新規オブジェクトを変数`instance` に代入します。
5. スレッド2は //3で`Singleton` オブジェクト参照を戻します。
6. スレッド2がスレッド1に取って代わられます。
7. スレッド1は中断個所から再開し、行 //2を実行して、別の`Singleton` オブジェクトが作成されます。
8. スレッド1は //3でこのオブジェクトを戻します。

この結果、`getInstance()` メソッドによって作成される`Singleton` オブジェクトは1つだけであると想定されているにもかかわらず、このオブジェクトが2つ作成されることになります。この問題は、リスト2に示すように、`getInstance()` メソッドを同期化して、一度に1つのスレッドだけがこのコードを実行できるようにすることにより、解決できます。

リスト2. スレッド・セーフなgetInstance() メソッド

```
public static synchronized Singleton getInstance()
{
  if (instance == null)          //1
    instance = new Singleton();  //2
  return instance;               //3
}
```

リスト2のコードは、`getInstance()` メソッドに対するマルチスレッド・アクセスでは、問題なく機能します。しかし、これを分析すると、同期化が必要なのは、メソッドの最初の呼び出しのときだけであることが分かります。それ以降の呼び出しでは同期化は必要ありません。これは、//2のコードを実行する呼び出しは最初の呼び出しだけであり、同期化を必要とするのはこの行だけであるためです。それ以外のすべての呼び出しは、`instance` が非`null` であることを判別し、それを戻します。最初の呼び出し以外は、どの呼び出しでも、複数のスレッドが同時に実行されても問題ありません。ただし、このメソッドは`synchronized` であるため、最初の呼び出しのとき以外は必要がないにもかかわらず、メソッドを呼び出すたびに同期化するコストを払うことになります。

このメソッドの効率を高めることを目指して、double-checked lockingというイディオムが作成されました。この考えは、メソッドの最初の呼び出しを除くすべての呼び出しで高コストな同期を避けるというものです。同期化のコストはJVMごとに異なります。最初のころは、このコストは非常に高くなることがありました。より進んだJVMが登場するようになると、同期化のコストは低下しましたが、それでも、`synchronized` されたメソッドまたはブロックに入ったりそこから出たりすることで、パフォーマンスが悪影響を受けます。JVMテクノロジーが進歩したとはいえ、プログラマーは、処理時間を不必要に費やすことを望んではいないのです。

同期化が必要なのはリスト2の行 //2だけですので、リスト3に示すように、同期化されたブロック内でラップすることができます。

リスト3. getInstance() メソッド

```
public static Singleton getInstance()
{
  if (instance == null)
  {
    synchronized(Singleton.class) {
      instance = new Singleton();
    }
  }
  return instance;
}
```

リスト3のコードには、複数のスレッドおよびリスト1で示されたものと同じ問題があります。`instance` が`null` である場合に、2つのスレッドが同時に`if` ステートメントに入る可能性があります。そして、1つのスレッドが`synchronized` ブロックに入って`instance` を初期化し、もう一方のスレッドはブロックされます。最初のスレッドが`synchronized` ブロックから出ると、待機していたスレッドがこのブロックに入り、もう1つの`Singleton` oオブジェクトを作成します。2番目のスレッドが`synchronized` ブロックに入るときには、`instance` が非`null` であるかどうかの検査が行われないことに注意してください。

double-checked locking

リスト3の問題を修正するためには、`instance` をもう一度検査する必要があります。ここから、「double-checked locking」という名前が付けられました。リスト3にdouble-checked lockingイディオムを適用したものがリスト4です。

リスト4. double-checked lockingの例

```
public static Singleton getInstance()
{
  if (instance == null)
  {
    synchronized(Singleton.class) {  //1
      if (instance == null)          //2
        instance = new Singleton();  //3
    }
  }
  return instance;
}
```

double-checked lockingの背景となる理屈は、//2で行われる2番目の検査により、リスト3の場合のように2つの異なる`Singleton` オブジェクトが作成される可能性がなくなるということです。ここで、次の一連のイベントが起こるものとします。

1. スレッド1が`getInstance()` メソッドに入ります。
2. `instance` が`null` であるため、スレッド1は //1で`synchronized` ブロックに入ります。
3. スレッド1がスレッド2に取って代わられます。
4. スレッド2が`getInstance()` メソッドに入ります。
5. `instance` はまだ`null` であるため、スレッド2は //1でロックを獲得しようとします。しかし、スレッド1がロックを保持しているため、スレッド2は //1でブロックされます。
6. スレッド2がスレッド1に取って代わられます。
7. スレッド1が実行されます。//2でインスタンスがまだ`null` になっているため、スレッド1は`Singleton` オブジェクトを作成し、その参照を`instance` に代入します。
8. スレッド1が`synchronized` ブロックを抜け出し、`getInstance()` メソッドからインスタンスを戻します。
9. スレッド1がスレッド2に取って代わられます。
10. スレッド2が //1でロックを獲得し、`instance` が`null` になっているかどうかを調べます。
11. `instance` は非`null` になっているため、2番目の`Singleton` オブジェクトは作成されず、スレッド1で作成されたオブジェクトが戻されます。

double-checked lockingの背景となっている理屈は申し分ありません。残念ながら、現実はまったく異なります。double-checked lockingの問題は、シングル・プロセッサー・マシンまたはマルチプロセッサー・マシンで機能することが保証されないことです。

double-checked lockingの障害の問題点は、JVMの実装のバグのせいではなく、現在のJavaプラットフォームのメモリー・モデルが原因となっています。このメモリー・モデルは、いわゆる「アウトオブオーダー書き込み」を許し、それがこのイディオムの障害の主要な原因となっています。

アウトオブオーダー書き込み

この問題を理解するためには、上記のリスト4の行 //3を再検討する必要があります。この行のコードは`Singleton` オブジェクトを作成し、このオブジェクトを参照するために変数`instance` を

初期化します。この行のコードの問題は、`Singleton` コンストラクターの本体が実行される前に、変数`instance` が非`null` になる可能性があることです。

え？ そんなはずはないだろうって？しかし、事実そのとおりなのです。なぜこのようなことが起こるのかを説明するに先立ち、double-checked lockingイディオムが破たんする理由を調べるしばらくの間、この事実を受け入れてください。ここで、リスト4のコードについて、次の一連のイベントが起こるものと考えてみてください。

1. スレッド1が`getInstance()` メソッドに入ります。
2. `instance` が`null` であるため、スレッド1は //1で`synchronized` ブロックに入ります。
3. スレッド1は //3に進んでインスタンスを非`null` にしますが、このときにはまだ コンストラクターが実行されていません。
4. スレッド1がスレッド2に取って代わられます。
5. スレッド2は、インスタンスが`null` になっているかどうかを調べます。インスタンスが`null` になっていないため、スレッド2は、完全に構成され、しかし部分的にしか初期化されていない`Singleton` オブジェクトを、`instance` 参照に対して戻します。
6. スレッド2がスレッド1に取って代わられます。
7. スレッド1は、コンストラクターを実行してそれに対する参照を戻すことにより、`Singleton` オブジェクトの初期化を完了します。

この一連のイベントの結果、ある期間のあいだ、スレッド2が、コンストラクターが実行されていないオブジェクトを戻すことになります。

これがどのようにして生じるのかを調べるために、`instance =new Singleton();` の行を次の疑似コードで置き換えてください。

```
mem = allocate();             //Allocate memory for Singleton object.
instance = mem;               //Note that instance is now non-null, but
                              //has not been initialized.
ctorSingleton(instance);      //Invoke constructor for Singleton passing
                              //instance.
```

この疑似コードは、単に可能性があるというだけでなく、いくつかのJITコンパイラーで実際に行われていることなのです。実行の順番が、順不同になってしまっていることが分かりますが、現在のメモリー・モデルではこれが許されてしまうのです。JITコンパイラーがこのようなことを行う以上、double-checked lockingの問題は、単なる教室での演習として片付けるわけにはいきません。

このようすを理解するために、リスト5のコードについて考えてみてください。これには、`getInstance()` メソッドから余分な部分を取り除いたものが含まれています。作成されるアセンブリー・コード (リスト6) を検討しやすくするために、「double-check」の要素を除去してあります。ここでの関心事は、`instance=new Singleton();` の行がJITコンパイラーでどのようにコンパイルされるのかということだけです。また、コンストラクターがアセンブリー・コードでどのように実行されるのかを理解しやすくするために、簡単なコンストラクターを用意しました。

リスト5. アウトオブオーダー書き込みを示すためのSingletonクラス

```
class Singleton
{
  private static Singleton instance;
  private boolean inUse;
  private int val;

  private Singleton()
  {
    inUse = true;
    val = 5;
  }
  public static Singleton getInstance()
  {
    if (instance == null)
      instance = new Singleton();
    return instance;
  }
}
```

リスト6は、リスト5の`getInstance()` メソッドの本体からSun JDK 1.2.1 JITコンパイラーによって生成された、アセンブリー・コードを含んでいます。

リスト6. リスト5のコードから生成されたアセンブリー・コード

```
;asm code generated for getInstance
054D20B0   mov         eax,[049388C8]      ;load instance ref
054D20B5   test        eax,eax             ;test for null
054D20B7   jne         054D20D7
054D20B9   mov         eax,14C0988h
054D20BE   call        503EF8F0            ;allocate memory
054D20C3   mov         [049388C8],eax      ;store pointer in
                                           ;instance ref. instance
                                           ;non-null and ctor
                                           ;has not run
054D20C8   mov         ecx,dword ptr [eax]
054D20CA   mov         dword ptr [ecx],1   ;inline ctor - inUse=true;
054D20D0   mov         dword ptr [ecx+4],5 ;inline ctor - val=5;
054D20D7   mov         ebx,dword ptr ds:[49388C8h]
054D20DD   jmp         054D20B0
```

注: 以下の説明では、命令アドレスの値がすべて`054D20` で始まっていますので、命令アドレスの最後の2つの値を使用してアセンブリー・コードの行を参照しています。例えば、`B5` は`test eax,eax` を表しています。

このアセンブリー・コードは、無限ループ内で`getInstance()` メソッドを呼び出すテスト・プログラムを実行して生成されたものです。このプログラムの実行中に、Microsoft Visual C++ デバッガーを実行し、テスト・プログラムに対応するJavaプロセスに接続してください。そして、実行を中断して、無限ループを表すアセンブリー・コードを見付けてください。

アセンブリー・コードの最初の2行の`B0` および`B5` では、メモリー位置`049388C8` から`eax` に`instance` 参照をロードし、`null` になっているかどうかを調べます。これは、リスト5の`getInstance()` メソッドの最初の行に対応しています。このメソッドが初めて呼び出されたときには、`instance` は`null` になっていて、コードは`B9` に進みます。`BE` のコードは、ヒープ内のメモリーを`Singleton` オブジェクト用に割り振り、そのメモリーへのポインターを`eax` に保管します次の行`C3` では、`eax` 内のポインターを受け取り、それを、メモリー位置`049388C8` にあるイン

スタンス参照に戻して保管します。これにより、`instance` が非`null` になり、有効な`Singleton` オブジェクトを表すようになります。ただし、このオブジェクトのコンストラクターはまだ実行されていません。このような状態では、double-checked lockingは破たんします。そして、行`C8` で`instance` ポインターが参照読み出し(dereference)され、`ecx` に保管されます。行`CA` および`D0` は、値`true` および`5` を`Singleton` オブジェクトに保管する、インライン・コンストラクターを表しています。行`C3` を実行してからコンストラクターを完了するまでの間にこのコードが割り込みを受けると、double-checked lockingは失敗します。

必ずしもすべてのJITコンパイラーが上記のようなコードを生成するわけではありません。コンストラクターが実行されてからでなければ`instance` を非`null` にしないようなコードを生成するものもあります。IBM SDK for JavaテクノロジーV1.3とSun JDK 1.3は、ともにこのようなコードを生成します。しかし、だからといって、このようなインスタンスでdouble-checked lockingを使用すべきであるというわけではありません。障害の原因となりうるものが、ほかにもあるのです。また、自分のコードがどのJVMで実行されるのかは、必ずしも常に分かるとは限りませんし、JITコンパイラーが、このイディオムを破たんさせるコードを生成するように変更される可能性は、常にあるのです。

double-checked locking: その2

現行のdouble-checked lockingコードが機能しないので、リスト7に示すような、このコードの別バージョンを利用して、上に示したアウトオブオーダー書き込みの問題を避けようとしてみました。

リスト7. アウトオブオーダー書き込みの問題を解決する試み

```
public static Singleton getInstance()
{
  if (instance == null)
  {
    synchronized(Singleton.class) {      //1
      Singleton inst = instance;         //2
      if (inst == null)
      {
        synchronized(Singleton.class) {  //3
          inst = new Singleton();        //4
        }
        instance = inst;                 //5
      }
    }
  }
  return instance;
}
```

リスト7のコードを見ると、ややおかしな方向に進んでいることに気が付くはずです。そもそも、double-checked lockingは、単純な3行の`getInstance()` メソッドの同期化を回避する手段として作り出されたものでした。リスト7のコードは、手に余るほど肥大化しています。さらに、このコードでは問題が修正されないのです。注意深く検討してみると、その理由が明らかになります。

このコードはアウトオブオーダー書き込みの問題を避けようとしています。そのために、ローカル変数`inst` と2番目の`synchronized` ブロックを導入しています。この理屈は、次のようになっています。

1. スレッド1が`getInstance()` メソッドに入ります。
2. `instance` が`null` であるため、スレッド1は //1で最初の`synchronized` ブロックに入ります。
3. ローカル変数`inst` が`instance` の値を入手します。この値は、//2では`null` になっています。
4. `inst` が`null` であるため、スレッド1は //3で2番目の`synchronized` ブロックに入ります。
5. そしてスレッド1が //4でコードの実行を開始し、`inst` を非`null` にしますが、この時点ではまだ`Singleton` のコンストラクターが実行されていません。(これはまさに、上で見たアウトオブオーダー書き込みの問題です。)
6. スレッド1がスレッド2に取って代わられます。
7. スレッド2が`getInstance()` メソッドに入ります。
8. `instance` が`null` であるため、スレッド2は //1で最初の`synchronized` ブロックに入ろうとします。現在スレッド1によってこのロックが保持されているため、スレッド2はブロックされます。
9. そして、スレッド1が //4の実行を完了します。
10. スレッド1はその後、完全に構成された`Singleton` オブジェクトを //5で変数`instance` に代入し、両方の`synchronized` ブロックを終了します。
11. スレッド1が`instance` を戻します。
12. その後で、スレッド2が実行され、//2で`instance` を`inst` に代入します。
13. スレッド2は、`instance` が非`null` であることを確認し、それを戻します。

ここで鍵となる行は //5です。この行は、`instance` が必ず`null` になることを保証するか、あるいは完全に構成された`Singleton` オブジェクトを参照するものと想定されています。理論と現実のくい違いが起こると、問題が発生します。

現行のメモリー・モデルの定義により、リスト7のコードは機能しません。Java言語仕様 (JLS) では、`synchronized` ブロック内のコードが`synchronized` ブロックの外側に移動しないことが求められています。しかし、`synchronized` ブロックに含まれていないコードを`synchronized` ブロックの中に 移動することはできない、とは言っていません。

JITコンパイラーは、この部分を最適化の好機として捕えるはずです。この最適化により、//4のコードと //5のコードが除去され、それらが結合され、リスト8に示すコードが生成されることが考えられます。

リスト8. リスト7のコードを最適化したもの

```
public static Singleton getInstance()
{
  if (instance == null)
  {
    synchronized(Singleton.class) {      //1
      Singleton inst = instance;         //2
      if (inst == null)
      {
        synchronized(Singleton.class) {  //3
          //inst = new Singleton();      //4
          instance = new Singleton();
        }
        //instance = inst;               //5
      }
    }
  }
  return instance;
}
```

この最適化が行われると、前に述べたものと同じアウトオブオーダー書き込みの問題が生じます。

volatileを使ってみますか?

もう1つのアイデアとして、変数`inst` および`instance` に`volatile` というキーワードを使用する方法があります。JLS (参考文献を参照) によると、`volatile` として宣言された変数は、順序が一貫していて、従って順序変更されないものと想定されています。しかし、`volatile` を使用してdouble-checked lockingの問題を修正しようとすると、2つの問題が生じます。

- この場合の問題は、順序の一貫性に関するものではありません。コードが、順序変更されるのではなく、移動されるのです。
- 多くのJVMでは、順序の一貫性について、`volatile` が正しく実装されません。

2番目の問題は、詳しく採り上げる価値があります。ここで、リスト9に示すコードについて考えてみてください。

リスト9. volatileに関する順序の一貫性

```
class test
{
  private volatile boolean stop = false;
  private volatile int num = 0;

  public void foo()
  {
    num = 100;    //This can happen second
    stop = true;  //This can happen first
    //...
  }

  public void bar()
  {
    if (stop)
      num += num;  //num can == 0!
  }
  //...
}
```

JLSによると、`stop` および`num` は`volatile` として宣言されるため、順序が一貫している必要があります。これは、`stop` が`true` の場合、`num` を`100` に設定していなければならないということを意味します。ただし、多くのJVMは`volatile` の順序一貫性機能を実装していないため、この振る舞いは当てになりません。したがって、スレッド1が`foo` を呼び出し、スレッド2が同時に`bar` を呼び出す場合には、スレッド1では、`num` が`100` に設定される前に`stop` が`true` に設定される可能性があります。これが原因で、スレッド2が、`stop` が`true` であるにもかかわらず、`num` がまだ`0` に設定されていると認識することがあります。ほかにも`volatile` と64ビット変数のアトミシティに関する問題がありますが、これは、この記事の範囲を超えています。このトピックの詳細については、参考文献を参照してください。

解決策

要するに、double-checked lockingは (いかなる形式のものであっても)、どのJVM実装でも動作するとは保証できないため、使用すべきではありません。JSR-133では、メモリー・モデルに関する

問題に対処していますが、double-checked lockingはこの新規メモリー・モデルではサポートされません。したがって、ユーザーがとりうる選択肢としては、次の2つがあります。

- リスト2に示したような、`getInstance()` メソッドの同期化を受け入れる。
- 同期化を行わず、`static` フィールドを使用する。

2番目の選択肢をリスト10に示します。

リスト10. 静的フィールドを使用したSingleton実装

```
class Singleton
{
  private Vector v;
  private boolean inUse;
  private static Singleton instance = new Singleton();

  private Singleton()
  {
    v = new Vector();
    inUse = true;
    //...
  }

  public static Singleton getInstance()
  {
    return instance;
  }
}
```

リスト10のコードは、同期化を使用せず、また、`static getInstance()` メソッドへの呼び出しが行われるまでは`Singleton` オブジェクトを作成しないようになっています。これは、同期化を行わないことを目標とする場合には、優れた選択肢です。

ストリングは不変ではない

アウトオブオーダー書き込みの問題と、コンストラクターが実行される前に参照が非`null` になることを考えると、読者は`String` クラスに興味を抱くことと思います。次のコードについて考えてください。

```
private String str;
//...
str = new String("hello");
```

`String` クラスは、不変なものと想定されています。しかし、上に述べたアウトオブオーダー書き込みの問題を考えると、それが原因で、ここでも問題が生じる可能性があるのではないのでしょうか？その恐れは確かにあります。2つのスレッドが`String str` にアクセスする場合について考えてください。一方のスレッドで、`str` 参照が、まだコンストラクターが実行されていない`String` オブジェクトを参照している可能性があります。実際、リスト11のコードは、このような事態が生じていることを示しています。断っておきますが、このコードが破たんするのは、私がテストした古いJVMの場合だけです。IBM 1.3 JVMでもSun 1.3 JVMでも、期待どおりに不変の`String` が生成されます。

リスト11. 変化しやすいStringの例

```
class StringCreator extends Thread
{
```

```
  MutableString ms;
  public StringCreator(MutableString muts)
  {
    ms = muts;
  }
  public void run()
  {
    while(true)
      ms.str = new String("hello");          //1
  }
}
class StringReader extends Thread
{
  MutableString ms;
  public StringReader(MutableString muts)
  {
    ms = muts;
  }
  public void run()
  {
    while(true)
    {
      if (!(ms.str.equals("hello")))         //2
      {
        System.out.println("String is not immutable!");
        break;
      }
    }
  }
}
class MutableString
{
  public String str;                         //3
  public static void main(String args[])
  {
    MutableString ms = new MutableString();  //4
    new StringCreator(ms).start();           //5
    new StringReader(ms).start();            //6
  }
}
```

このコードの //4では、//3での2つのスレッドによって共用されている`String` 参照を含む、`MutableString` クラスを作成しています。//5と //6では、2つの別個のスレッドで`StringCreator` と`StringReader` の2つのオブジェクトを作成し、そこでは`MutableString` オブジェクトへの参照を渡しています。`StringCreator` クラスは無限ループに入り、//1で「hello」という値の`String` オブジェクトを作成します。`StringReader` も無限ループに入り、//2で現行の`String` オブジェクトの値が「hello」になっているかどうかを調べます。この値が「hello」になっていない場合には、`StringReader` スレッドはメッセージを出力して停止します。`String` クラスが不変であれば、このプログラムからは何も出力されないはずです。`StringReader` が`str` 参照を、「hello」という値の`String` オブジェクトとして判定しないのは、アウトオブオーダーの問題が生じた場合だけです。

このコードをSun JDK 1.2.1などの古いJVMで実行すると、アウトオブオーダーの問題が発生し、したがって、`String` は不変ではなくなります。

要約

プログラマーたちは、Singletonで高コストな同期化を回避するために、工夫を凝らして、double-checked lockingイディオムを考案しました。残念ながら、このイディオムが、現行のメモリー・

モデルが原因で安全なプログラミング構成概念ではないことが明らかになったのは、これがかなり広く使われるようになった後のことでした。メモリー・モデルの、ぜい弱な部分を改良するための努力が行われています。しかし、新しく提案されたメモリー・モデルでも、double-checked lockingは通用しません。この問題に対する最良の解決策は、同期化を受け入れるか、あるいは`static field`を使用することです。

著者について

Peter Haggar

Peter Haggarは、アメリカのノースカロライナ州にあるResearch Triangle Parkに勤務するIBMシニア・ソフトウェア・エンジニアであり、「Practical Java Programming Language Guide」(Addison-Wesley 社発行) の著者です。彼はまた、Javaプログラミングに関する多数の記事を発表しています。彼は、開発ツール、クラス・ライブラリー、およびオペレーティング・システムなど、幅広いプログラミング経験を持っています。IBMでは、未来のインターネット・テクノロジーに携わり、現在は高性能なWebサービスを中心に研究を行っています。Peterは、多くの業界の会議で頻繁に技術的なスピーチを行っています。彼はIBMに14年以上にわたって勤務しており、また、Clarkson Universityよりコンピューター・サイエンスのB.S. を取得しています。