マルチコア・システムでの Java 並行性バグのパターン

あまりよく知られていない 6 つの Java 並行性に関するバグのパターン

Zhi Da Luo
Software Engineer
IBM China

Yarden Nir-Buchbinder
Research Scientist
IBM

Raja Das
Software Architect
IBM

2010年 12月 21日

並行性に関するバグのパターンを調べることで、並行プログラミングに対する全般的な認識が深まると同時に、機能しない、あるいは機能しない可能性のあるコーディング・イディオムを学ぶことができます。この記事では、Zhi Da Luo、Yarden Nir-Buchbinder、Raja Das の 3 人の著者が、マルチコア・システム上で実行される Java™ アプリケーションのスレッド・セーフティーやパフォーマンスを脅かす、あまり知られていない 6 つの並行性に関するバグを明らかにします。

マルチスレッド・プログラミングの経験が浅いプログラマーにとって、ソフトウェアをマルチコア・システム対応にする際に問題となるのは次の 2 点です。1 つは、並行性によって Java プログラムに、データ・レースやデッドロックなどといった新しいバグのカテゴリーが加わることです。これらのバグは再現させて診断するのが非常に難しいという問題があります。そしてもう 1 つは、多くのプログラマーはマルチスレッド・プログラミングの特定のイディオムの微妙な部分を認識していないという点です。この認識不足が、コードのエラーの原因となります。

並行プログラムにバグを混入させないためには、Java プログラマーはマルチスレッド・コードでバグが発生しやすい重要な部分を認識して、バグのないソフトウェアを作成できるようになる必要があります。この記事は、並行プログラミングを難しくしている部分をこれから理解しようとしている段階、あるいはある程度理解している段階の Java 開発者を対象としています。記事では、ダブルチェック・ロッキング、スピン・ウェイト、wait-not-in-loop などのよく知られた Java 並行性に関するバグのパターンを焦点とするのではなく、あまりよく知られていないけれども、実際の Java アプリケーションに頻繁に現れている 6 つのパターンを紹介します。実のところ、6

つのパターンのうち最初の 2 つの例は、人気の高い 2 つの Web サーバーで見つかった本物のバグです。

1. Jetty で見つかっているアンチパターン

並行性に関するバグで最初に紹介するのは、広く使われているオープンソースの HTTP サーバー、Jetty で検出されたもので、Jetty コミュニティーが認めている実際のバグです (バグ・レポートについては、「参考文献」を参照)。

リスト 1. volatile が指定されていて、ロックが保持されないフィールドでのアトミックでない処理

```
// Jetty 7.1.0,
// org.eclipse.jetty.io.nio,
// SelectorManager.java, line 105

private volatile int _set;
......
public void register(SocketChannel channel, Object att)
{
   int s=_set++;
   ......
}
......
public void addChange(Object point)
{
synchronized (_changes)
   {
      ......
   }
}
```

リスト 1 のエラーは、以下の要素が重なった結果、発生します。

- 最初に、`_set` が `volatile` として宣言されています。これは、複数のスレッドがこのフィールドにアクセスできることを意味します。
- けれども、`_set++` はアトミックではありません。つまり、必ずしも 1 つの不可分の処理として実行されるわけではなく、むしろ 3 つの別個の処理のシーケンス (読み取り–変更–書き込み) の簡略表現と言えます。
- さらに、`_set++` はロックで保護されません。そのため、複数のスレッドが同時に `register` メソッドを呼び出したとするとレース·コンディションが発生し、誤った `_set` 値が得られるという結果になります。

このタイプのエラーは Jetty のコードに見つかったように、皆さんが作成するコードでも簡単に発生する可能性があります。そこで、どのようにしてそのエラーが発生するのかをこれから詳しく見ていきます。

バグ・パターンの要素

コードの論理シーケンスを追っていくと、このバグ・パターンを解明するのに役立ちます。以下に記載するのは、変数 i に関する処理です。

```
i++
--i
i += 1
i -= 1
i *= 2
```

ここに挙げた類の処理はアトミックではありません (つまり、「読み取り–変更–書き込み」からなる処理です)。Java 言語での `volatile` というキーワードが保証するのは変数の可視性だけで、アトミック性は保証しないことを知っていれば、このようなコードは作成しないはずです。volatile が指定され、ロックで保護されないフィールドに対してアトミックでない処理を行えば、レース・コンディションが発生するのは目に見えています。ただしレース·コンディションが発生するのは、アトミックでない処理に複数のスレッドが同時にアクセスした場合だけです。

スレッド・セーフなプログラムで変数を変更できるのは、1 つの書き込みスレッドだけです。他のスレッドから最新の値を読み取れるようにするには、変数を `volatile` として宣言します。

従って、コードからバグが発生するかどうかは、どれだけの数のスレッドが同時に 1 つの処理を実行できるかによって決まってきます。start-join 関係や外部ロックによって、1 つのスレッドだけがアトミックでない処理を呼び出すようにすれば、そのコード・イディオムはスレッド・セーフになります。

Java コードでは、キーワード `volatile` が保証するのは変数の可視性だけで、アトミック性は保証しないことを肝に銘じてください。変数の処理がアトミックではなく、複数のスレッドがその処理にアクセスできるとしたら、volatile の同期機能に依存するのは禁物です。代わりに、`java.util.concurrent` パッケージの synchronized ブロック、ロック・クラス、アトミック・クラスを使用してください。これらの機能は、プログラムを確実にスレッド・セーフにするように設計されています。

2. ミュータブルなフィールドに対する同期

Java 言語では、synchronized ブロックを使用して相互排他ロックを獲得することにより、マルチスレッド・システムでの共有リソースへのアクセスを保護します。けれどもミュータブルなフィールドに対する同期を行う場合には、相互排他を無効にできてしまう抜け道が存在します。この問題を解決するには、同期対象のフィールドを常に `private final` として宣言することです。なぜそうする必要があるのか理解できるように、ミュータブルなフィールドに対する同期の問題について少し詳しく見ていきます。

更新されるフィールドに対する同時ロック

synchronized ブロックで保護されるのは、同期対象のフィールドそのものではなく、そのフィールドから参照されるオブジェクトです。同期対象のフィールドがミュータブル (初期化以外にも、プログラムのどこででもフィールドを割り当てることができるという意味) なフィールドである場合、同期はほとんど有効な意味を持たなくなります。というのも、異なるスレッドがそれぞれに異なるオブジェクトに対して同期することが可能だからです。

この問題は、オープンソースの Web アプリケーション・サーバーである Tomcat から抜粋したコード・スニペット (リスト 2) に見られます。

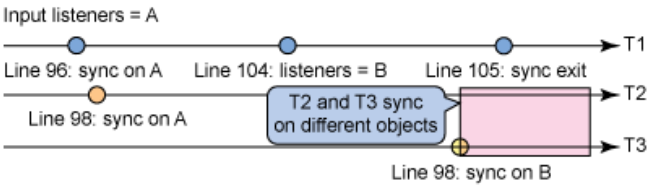
リスト 2. 誤りのある Tomcat

```
96: public void addInstanceListener(InstanceListener listener) {
97:
98:    synchronized (listeners) {
99:       InstanceListener results[] =
100:        new InstanceListener[listeners.length + 1];
101:      for (int i = 0; i < listeners.length; i++)
102:         results[i] = listeners[i];
103:      results[listeners.length] = listener;
104:      listeners = results;
105:   }
106:
107:}
```

例えば、`listeners` が配列 A を参照し、スレッド T1 が配列 A に対するロックを獲得したとします。T1 はその後、配列 B の作成でビジー状態になります。その間にスレッド T2 が現れて、配列 A に対するロックを獲得するためにブロック状態になります。T1 が配列 B に `listeners` を設定し終わってブロックを解除すると、T2 が配列 A に対するロックを獲得し、配列 B のコピーを開始します。その後、スレッド T3 が現れて配列 B に対するロックを獲得した場合を考えてみてください。これらのスレッドはそれぞれに異なるロックを獲得しているので、T2 と T3 は同時に同じ配列 B のコピーを作成していることになります。

図 1 に、このシーケンスを表します。

図 1. ミュータブルなフィールドに対する同期が原因の相互排他の欠如



このような状況では、望ましくない数々の振る舞いが発生する恐れがあります。少なくとも、新しい listeners のうちの 1 つが消失するか、あるいはスレッドのいずれかが `ArrayIndexOutOfBoundsException` を受け取ることは確実です (`listeners` 参照とその長さが、メソッドの任意の時点で変わるためです)。

有効なプラクティスは、同期対象のフィールドは常に `private final` として宣言することです。こうすれば、ロック・オブジェクトはそのままの状態を維持し、`mutex` が保証されます。

3. java.util.concurrent ロックの解除洩れ

`java.util.concurrent.locks.Lock` インターフェースを実装するロックは、複数のスレッドがどのように共有リソースにアクセスするかを制御します。このようなロックにはブロック構造が不要なため、synchronized メソッドまたは synchronized 文よりも柔軟性がありますが、その一方、この柔軟性がコーディング・エラーの原因となることもあります。それは、ブロックを使用しないロックは決して自動的に解除されないからです。`Lock.lock()` が呼び出される場合、同じインスタンスで対応する `unlock()` が呼び出されなければ、結果的にロックが解除されないままとなります。

このような `java.util.concurrent` のロックの解除洩れのバグは、重要なコードでのメソッドの振る舞いを見落とすだけで簡単に発生してしまいます。その一例は、スローされる可能性のある例外を見落とした場合です。リスト 3 でこの例が示されているのは、共有リソースへのアクセス中に `accessResource` メソッドが`InterruptedException` をスローする部分です。この例外がスローされているため、`unlock()` は呼び出されません。

リスト 3. ロックの解除洩れが発生する仕組み

```
private final Lock lock = new ReentrantLock();

public void lockLeak() {
   lock.lock();
   try {
      // access the shared resource
      accessResource();
      lock.unlock();
   } catch (Exception e) {}

public void accessResource() throws InterruptedException {...}
```

ロックが必ず解除されるようにするには、すべての `lock` メソッドを `unlock` メソッドと対にして、`try-finally` ブロックに配置すればよいのです。リスト 4 に、この対処方法を記載します。

リスト 4. unlock 呼び出しを常に finally ブロックに配置すること

```
private final Lock lock = new ReentrantLock();

public void lockLeak() {
   lock.lock();
   try {
      // access the shared resource
      accessResource();
   } catch (Exception e) {}
   finally {
      lock.unlock();
   }

public void accessResource() throws InterruptedException {...}
```

4. synchronized ブロックのパフォーマンスの調整

並行性に関するバグのなかには、コードを壊すことはなくても、アプリケーションのパフォーマンスを低下させるものがあります。一例として、リスト 5 の `synchronized` ブロックを見てください。

リスト 5. synchronized ブロックの不変コード

```
public class Operator {
   private int generation = 0; //shared variable
   private float totalAmount = 0; //shared variable
   private final Object lock = new Object();

   public void workOn(List<Operand> operands) {
      synchronized (lock) {
         int curGeneration = generation; //requires synch
         float amountForThisWork = 0;
         for (Operand o : operands) {
            o.setGeneration(curGeneration);
            amountForThisWork += o.amount;
         }
         totalAmount += amountForThisWork; //requires synch
         generation++; //requires synch
      }
   }
}
```

リスト 5 の 2 つの共有変数へのアクセスは適切に同期されますが、このリストをよく見てみると、`synchronized` ブロックに必要以上の計算処理が必要になっていることに気付くはずです。この問題は、リスト 6 のように行の順序を変更することで解決することができます。

リスト 6. 不変コードが含まれない synchronized ブロック

```
public void workOn(List<Operand> operands) {
   int curGeneration;
   float amountForThisWork = 0;
   synchronized (lock) {
      int curGeneration = generation++;
   }
   for (Operand o : operands) {
      o.setGeneration(curGeneration);
      amountForThisWork += o.amount;
   }
   synchronized (lock)
      totalAmount += amountForThisWork;
   }
}
```

マルチコア・マシン上で実行すると、2 番目のバージョンのパフォーマンスは遥かに改善されることになります。その理由は、リスト 5 では synchronized ブロックが並列実行の妨げとなっているためです。このメソッドは、ループでの計算処理に時間がかかる可能性があります。リスト 6 ではこのループを synchronized ブロックの外に出しているため、複数のスレッドが並行して実行できるようになるというわけです。一般に、synchronized ブロックをできるだけ簡潔にするように心掛けて、スレッド・セーフティーを損ねないようにしてください。

別の方法

リスト 5 とリスト 6 で使用している 2 つの共有変数に `AtomicInteger` と `AtomicFloat` を使用して、同期をまるごと排除するほうが有効な方法ではないかと思っている方もいるでしょう。この方法が可能かどうかは、他のメソッドがこれらの変数をどのように使用するか、そしてこの 2 つの変数の間に依存関係があるかどうかによって決まります。

5. 多段式アクセス

例えば、2 つのテーブルを保持するアプリケーションに取り組んでいるとします。アプリケーションの一方のテーブルは従業員の名前を通し番号にマッピングし、もう一方のテーブルは通し

番号を給与にマッピングします。このデータは、同時にアクセスして更新できるようにしなければなりません。それには、スレッド・セーフな `ConcurrentHashMap` を使用するという手段があります (リスト 7 を参照)。

リスト 7. 2 段階のアクセス

```
public class Employees {
   private final ConcurrentHashMap<String,Integer> nameToNumber;
   private final ConcurrentHashMap<Integer,Salary> numberToSalary;

   ... various methods for adding, removing, getting, etc...

   public int geBonusFor(String name) {
      Integer serialNum = nameToNumber.get(name);
      Salary salary = numberToSalary.get(serialNum);
      return salary.getBonus();
   }
}
```

このソリューションはスレッド・セーフであるように見えますが、実際にはそうではありません。問題は、`getBonusFor` メソッドがスレッド・セーフになっていないことです。つまり、通し番号を取得してから、その番号を使って給与を取得するまでの間に、別のスレッドが両方のテーブルから従業員を削除する可能性があります。そうなった場合、2 番目のマップにアクセスすると `null` が返され、例外がスローされることになります。

各マップ自体をスレッド・セーフにするだけでは不十分です。マップの間には依存関係があるため、両方のマップにアクセスする処理には、アトミックなアクセスが必要になります。この例では、スレッド・セーフではないコンテナー (`java.util.HashMap` など) を使用した後に、明示的な同期を使用して各アクセスを保護するという方法を使っていたら、スレッド・セーフティーを実現できたはずです。この方法を使えば、必要に応じて synchronized ブロックに両方のアクセスを含めることもできます。

6. 対称ロックのデッドロック

スレッド・セーフなコンテナー・クラスを考えてみてください。つまり、クライアントに対してスレッド・セーフであることを保証するデータ構造です (`java.util` のほとんどのコンテナーは、クライアントがコンテナーの使用を基準に同期を行わなければならないため、スレッド・セーフではありません)。リスト 8 では、可変のメンバーがデータを保管すると、ロック・オブジェクトがそのデータへのすべてのアクセスを保護します。

リスト 8. スレッド・セーフなコンテナー

```
public <E> class ConcurrentHeap {
   private E[] elements;
   private final Object lock = new Object(); //protects elements

   public void add (E newElement) {
      synchronized(lock) {
         ... //manipulate elements
      }
   }

   public E removeTop() {
      synchronized(lock) {
         E top = elements[0];
         ... //manipulate elements
         return top;
      }
   }
}
```

ここで、1 つのメソッドを追加します。このメソッドは、別のインスタンスを引数に取り、そのインスタンスのすべての要素を現在のインスタンスに追加するというメソッドです。このメソッドは、両方のインスタンスの `elements` メンバーにアクセスしなければならないため、両方のインスタンスでロックを取得します (リスト 9 を参照)。

リスト 9. このメソッドを追加することにより、デッドロックに至ります

```
public void addAll(ConcurrentHeap other) {
   synchronized(other.lock) {
      synchronized(this.lock) {
         ... //manipulate other.elements and this.elements
      }
   }
}
```

コンテナーだけではありません

対称ロックのデッドロックは Java 1.4 のリリースで発生したことから、このシナリオはある程度知れ渡っています。Java 1.4 リリースでは、`Collections.synchronized` メソッドによって返される同期コンテナーの一部でデッドロックが発生しました。けれども、対称ロックのデッドロックが発生しやすいのはコンテナーだけではありません。クラスに、同じクラスの別のインスタンスをその引数として取るメソッドがあり、そのメソッドが 2 つのインスタンスのメンバーに対してアトミックな処理を行わなければならないとしたら、常に対称ロックのデッドロックが発生する可能性があります。その典型的な例は、`compareTo` メソッドと `equals` メソッドの 2 つです。

デッドロックの可能性が潜んでいることがわかりますか？例えば、プログラムが `heap1` と `heap2` という 2 つのインスタンスを保持しているとします。ここでもし、あるスレッドが `heap1.addAll(heap2)` を呼び出し、それと同時に別のスレッドが `heap2.addAll(heap1)` を呼び出したとしたら、この 2 つのスレッドはデッドロックという結果になってしまいます。別の言葉に置き換えると、最初のスレッドは `heap2` のロックを取得しましたが、その前に、2 番目のスレッドがすでにメソッドの実行を開始していて、`heap1` のロックを保持しています。そのため、それぞれのスレッドはもう一方のスレッドが保持するロックを待機し続けることになります。

対称ロックのデッドロックを回避する方法は、2 つのインスタンスのロックを同時に取得しなければならない場合には、その順序を動的に計算して、どのロックを先に取得すればよいかを決定

できるようなインスタンスの順序を判断することです。この回避方法については、Brian Goetz が彼の著書『Java Concurrency in Practice』(「参考文献」を参照) のなかで詳しく説明しています。

まとめ

多くの Java 開発者は、マルチコア環境対応の並行プログラムを作成する方法をまだ学び始めたばかりです。この学習プロセスのなかで、私たちはすでに習得したシングル・スレッド・プログラミング・イディオムをマルチスレッド・プログラミング・イディオムに置き換えています。本質的に、マルチスレッド・プログラミング・イディオムはシングル・スレッド・プログラミング・イディオムよりも複雑です。マルチスレッド・プログラミングの落とし穴を見つけるには、並行性に関するバグのパターンを調べることが有効な方法であり、それによってマルチスレッド・プログラミング・イディオムの微妙な部分を習得できるようにもなります。

バグのパターンをバグ要素の集まりとして認識できるようになれば、コードの作成中、あるいはコード・レビューの最中に、特定のシグナルが警告の役割を果たすようになります。そのために静的分析ツールを使用することもできます。例えばオープンソースの静的分析ツールである FindBugs は、コード内で推定されるバグのパターンを調べます。実際、FindBugs を使用すれば、この記事で説明した 2 番目と 3 番目のバグ・パターンを検出することができます。

静的分析ツールの既知の欠点は、誤った警報を生成することです。そのため、バグのないコード・パターンをチェックするなどして、思ったよりも作業に長い時間がかかることもあります。最近では、並行プログラムをテストするという特定の目的により適した、動的分析ツールも新たに登場してきています。そのようなツールの例として挙げられるのが、IBM® Multicore Software Development Kit (MSDK) と ConcurrentTesting (ConTest) の 2 つです。どちらも alphaWorks から無料で入手することができます。

著者について

Zhi Da Luo

Zhi Da Luo は、IBM China Development Lab の Emerging Technology Institute に勤務するソフトウェア・エンジニアです。彼は 2008年に IBM に入社しました。プログラム分析、バイトコード・インスツルメンテーション、Java 並行プログラムで経験を積んだ彼は、現在、Java 並列ソフトウェアの静的/ランタイム分析ツールに取り組んでいます。彼は、中国・北京の Peking University でソフトウェア・エンジニアリングの修士号を取得しました。

Yarden Nir-Buchbinder

Yarden Nir-Buchbinder は、イスラエルの Technion でコンピューター・サイエンスの理学士号を取得し、Haifa University で哲学の修士号を取得しました。2000年から、IBM Haifa Research Lab で並行性およびテスト・カバレッジを専門に研究を続けています。彼が執筆および共同執筆した出版物は複数あり、特許取得者でもあります。

Raja Das

Raja Das は、IBM Software Group のソフトウェア・アーキテクトです。現在は、マルチコア/複数コア・システムを対象としたライブラリーおよびフレームワークを専門としています。以前は WebSphere Partner Gateway の製品アーキテクトでした。Dr. Das は、プログラミング言語、並列ソフトウェアおよびシステムなどに興味を持っています。