ConTestを使用したマルチスレッド・ユニットのテスト

並列テストが困難な理由とConTestの活用

Yarden Nir-Buchbinder
Research scientist
IBM

Shmuel Ur
Research scientist
IBM

2006年 4月 04日

並列プログラミングは、バグが発生しやすいことで有名です。さらに悪いことに、並列プログラミングで発生したバグは開発プロセスの終わり近くになって発見されることが多く、この段階でのバグは開発プロセスに深刻な影響を与え、修正することも困難です。従来の単体テスト手法をどれだけ徹底的に実行しても、並列プログラミングにおけるバグをなくすことは困難です。この記事では、並列プログラミングに詳しいShmuel UrとYarden Nir-Buchbinderが、こうしたバグはなぜ見つけるのが難しいのかを説明し、IBM Researchによる新しい解決策を紹介します。

並列プログラミングではバグが発生しやすいということは周知の事実です。こうしたプログラムを書くのは厄介な作業です。プログラムの作成中に忍び込んでくるバグを見つけるのも容易なことではありません。こうしたバグの多くが、システム・テストや機能テストの段階で、あるいは実際にユーザーによって使用された段階で初めて見つかるのです。この段階までくると、こうしたバグの修正は非常に困難なため、バグの修正には（もしも修正することができればの話ですが）多くの費用がかかってしまいます。

この記事ではConTestというツールを紹介します。これは、並列プログラムのテストやデバッグ、カバレッジの測定に使用するツールです。これから説明しますが、ConTestは単体テストに代わるものではなく、並列プログラム上での単体テストの問題点を解決する補助的なテスト手法です。

この記事では、ConTestの使用例をいくつか紹介します。ConTestの基本動作を理解したら、自分自身でこれらの例を試してみてください。

なぜ単体テストだけでは不十分なのか

Java™ の開発者に尋ねれば、単体テストは良いテスト技法だと答えるでしょう。適切な費用をかけて単体テストを行えば、その分の成果は期待できます。単体テストを実行することにより、単

体テストを実行しない場合と比較して、より早い段階でバグを発見し、より簡単にバグを修正することができます。しかし従来の単体テストの手法では、どんなに徹底的にテストを実行したとしても、並列プログラミングで発生するバグを発見するのは容易なことではありません。その結果、こうしたバグの多くがプログラミング作業の終わりの段階で見つかることになります。

なぜ、単体テストでは並列プログラミングで発生するバグを発見することができないのでしょうか。並列プログラム（およびバグ）の問題は、その偏りの無さにあるとよく言われますが、皮肉なことに、並列プログラムの問題は、単体テストの目的に対して偏りがあることなのです。以下の2つの例で、この点を説明します。

同期プロトコルのない名前印刷クラス

最初の例は、名と姓に分かれた名前を印刷するだけという単純なクラスです。説明をわかりやすくするため、このタスクを、名を印刷するスレッド、スペースを印刷するスレッド、姓を印刷して改行するスレッドの3つに分割します。完全な同期プロトコル（ロックの同期化やwait() および notifyAll() の呼び出しなど）により、処理はすべて正しい順序で実行されます。リスト1を見ればわかるとおり、main() は単体テストとして機能し、「Washington Irving」という名前の印刷処理用にこのクラスを呼び出します。

リスト1. 名前印刷クラス

```
public class NamePrinter {

   private final String firstName;
   private final String surName;

   private final Object lock = new Object();
   private boolean printedFirstName = false;
   private boolean spaceRequested = false;

   public NamePrinter(String firstName, String surName) {
      this.firstName = firstName;
      this.surName = surName;
   }

   public void print() {
      new FirstNamePrinter().start();
      new SpacePrinter().start();
      new SurnamePrinter().start();
   }

   private class FirstNamePrinter extends Thread {
      public void run() {
         try {
            synchronized (lock) {
               while (firstName == null) {
                  lock.wait();
               }
               System.out.print(firstName);
               printedFirstName = true;
               spaceRequested = true;
               lock.notifyAll();
            }
         } catch (InterruptedException e) {
            assert (false);
         }
      }
   }

   private class SpacePrinter extends Thread {
```

```
      public void run() {
         try {
            synchronized (lock) {
               while ( ! spaceRequested) {
                  lock.wait();
               }
               System.out.print(' ');
               spaceRequested = false;
               lock.notifyAll();
            }
         } catch (InterruptedException e) {
            assert (false);
         }
      }
   }

   private class SurnamePrinter extends Thread {
      public void run() {
         try {
            synchronized(lock) {
               while ( ! printedFirstName || spaceRequested || surName == null) {
                  lock.wait();
               }
               System.out.println(surName);
            }
         } catch (InterruptedException e) {
            assert (false);
         }
      }
   }

   public static void main(String[] args) {
      System.out.println();
      new NamePrinter("Washington", "Irving").print();
   }
}
```

必要であれば、このクラスをコンパイルして実行し、名前が正しく印刷されることを確認することができます。次に、リスト2のように、同期プロトコルをすべて取り除いてみましょう。

リスト2. 同期プロトコルのない名前印刷クラス

```
public class NakedNamePrinter {

   private final String firstName;
   private final String surName;

   public NakedNamePrinter(String firstName, String surName) {
      this.firstName = firstName;
      this.surName = surName;
      new FirstNamePrinter().start();
      new SpacePrinter().start();
      new SurnamePrinter().start();
   }

   private class FirstNamePrinter extends Thread {
      public void run() {
         System.out.print(firstName);
      }
   }

   private class SpacePrinter extends Thread {
      public void run() {
         System.out.print(' ');
      }
```

```
   }

   private class SurnamePrinter extends Thread {
      public void run() {
         System.out.println(surName);
      }
   }

   public static void main(String[] args) {
      System.out.println();
      new NakedNamePrinter("Washington", "Irving");
   }
}
```

こうすると処理を正しい順序で実行する指示がなくなってしまうため、このクラスの動作はまったく不正確なものになります。しかし、このクラスをコンパイルして実行したらどうなるでしょうか。実は、「Washington Irving」は正しく印刷され、同期プロトコルを指定した場合とまったく同じ結果になります。

このことから何が言えるでしょうか。同期プロトコルが指定されたこのクラスを、テストで使用する並列クラスに置き換えて考えてください。単体テストを繰り返し実行し、すべて問題なかった場合、当然このクラスには問題がないと考えるでしょう。しかし、今見たとおり、同期プロトコルがまったく指定されていない場合でも、同じように正しい結果が得られるのです。ということは、同期プロトコルを間違って指定した場合でも、正しい結果が得られると考えることができます。つまり、プロトコルをテストしたつもりでも、実際にはテストしていなかったことになるのです。

次に、別の例を見てみましょう。

バグのある作業キュー

以下のクラスは、一般的な並列ユーティリティのモデルである作業キューです。このクラスには、タスクをキューに入れるメソッドと、そのタスクを処理するメソッドがあります。work() メソッドは、タスクをキューから取り出す前にキューが空であるかどうかを確認し、空の場合はそのまま待機します。enqueue() メソッドは、待機中のすべてのスレッド（存在する場合）に通知します。この例をわかりやすくするために、タスクは単純な文字列とし、この文字列を印刷するだけの処理とします。この例でも、main() は単体テストとして機能します。ところで、このクラスにはバグが1つあります。

リスト3. 印刷キュー

```
import java.util.*;

public class PrintQueue {
   private LinkedList<String> queue = new LinkedList<String>();
   private final Object lock = new Object();

   public void enqueue(String str) {
      synchronized (lock) {
         queue.addLast(str);
         lock.notifyAll();
      }
   }

   public void work() {
      String current;
```

```
      synchronized(lock) {
         if (queue.isEmpty()) {
            try {
               lock.wait();
            } catch (InterruptedException e) {
               assert (false);
            }
         }
         current = queue.removeFirst();
      }
      System.out.println(current);
   }

   public static void main(String[] args) {
      final PrintQueue pq = new PrintQueue();

      Thread producer1 = new Thread() {
         public void run() {
            pq.enqueue("anemone");
            pq.enqueue("tulip");
            pq.enqueue("cyclamen");
         }
      };

      Thread producer2 = new Thread() {
         public void run() {
            pq.enqueue("iris");
            pq.enqueue("narcissus");
            pq.enqueue("daffodil");
         }
      };

      Thread consumer1 = new Thread() {
         public void run() {
            pq.work();
            pq.work();
            pq.work();
            pq.work();
         }
      };

      Thread consumer2 = new Thread() {
         public void run() {
            pq.work();
            pq.work();
         }
      };

      producer1.start();
      consumer1.start();
      consumer2.start();
      producer2.start();
   }
}
```

テストを実行したところ、何も問題がないようです。このクラスを開発した担当者も、この結果には満足でしょう。2つのプロデューサーと2つのコンシューマーがあり、それぞれの順序が複雑なために待機が発生するこのクラスのテストは単純なものではありませんでしたが、正常に機能しました。

しかし、このクラスには上で書いたバグが潜んでいるのです。それがどんなバグかわかりましたか? わからなかった場合は、もう少しこの記事を読み進めてください。すぐに答えが見つかります。

並列プログラミングにおける偏り

この2つの単体テストの例では、なぜ並列プログラミングのバグが見つからなかったのでしょうか。原則として、スレッド・スケジューラーは途中でスレッドを切り替え、順序を変えて実行できることになっているのですが、そうならない場合も多くあります。通常の単体テスト内の並列タスクは規模が小さく数も少ないため、たとえスレッドをwait() メソッドなどで強制的に切り替えたとしても、スケジューラーがスレッドを切り替える前にタスクが終了してしまう場合が多いのです。スケジューラーによってスレッドの切り替えが実行された場合でも、常にプログラム内の同じ場所で切り替えが行われてしまうことになりがちです。

これまでに説明したように、問題は並列プログラムの偏りにあります。つまり、いくつもの考えられるパターンのうち、たった1つのインターリービング（異なるスレッド内にあるコマンドの相対順序）をテストするだけで終わってしまうということです。では、もっと多くのインターリービングが実行されるのはどんな場合でしょうか。それは、並列クラスとプロトコルの間にもっと多くの並列タスクと複雑な相互処理が追加された場合です。すなわち、システム・テストと機能テストを実行した場合、またはすべての機能をエンド・ユーザーのサイトで実行した場合です。この時点で、すべてのバグが表面に現れてきます。

ConTestを使用した単体テスト

JVMの偏りを少なくするためには、単体テストを実行する際の「あいまいさ」が必要になります。ここでConTestの出番となります。ConTestを使用してリスト2のNakedNamePrinterを何回か実行すると、リスト4のようにさまざまな実行結果を得ることができます。

リスト4. 同期プロトコルのない名前印刷クラスをConTestを使用して実行した場合の実行結果

```
>Washington Irving(the expected result)

> WashingtonIrving(the space was printed first)

>Irving
 Washington(surname + new-line printed first)

> Irving
Washington(space, surname, first name)
```

実行結果が必ずしもこのとおりの順序になるとは限りません。同じ実行結果が繰り返し表示される場合もあります。最初の2つの結果が何回か繰り返し表示され、その次に最後の2つの結果が表示される場合もあります。しかし、何回か繰り返すうちに必ずすべての結果が表示されます。ConTestを使用することにより、すべての種類のインターリービングを実行することができます。インターリービングはランダムに選択されるため、同じテストを何回も実行すれば、そのたびに違ったインターリービングが実行されることになるためです。それとは対照的に、ConTestを使用してリスト1のNamePrinterを実行すると、常に正しい結果になります。この場合、同期プロトコルによって強制的に正しい順序で実行されるため、ConTestを使用しても、生成されるのは正しいインターリービングのみになります。

ConTestを使用してPrintQueueを実行した場合、異なった順序で花の名前が印刷されます。この結果を見て単体テストは成功だと考えるかもしれませんが、何回か繰り返して実行してくださ

い。24行目の処理で突然LinkedList.removeFirst() によってNoSuchElementExceptionがスローされるはずです。このバグは以下のシナリオが実行された場合に発生します。

1. 2つのコンシューマー・スレッドが開始されるが、キューが空のためwait() を処理する。
2. プロデューサーがタスクをキューに入れ、両方のコンシューマーに通知する。
3. 一方のコンシューマーがキューをロックし、タスクを処理してキューを空にする。その後ロックを解除する。
4. もう一方のコンシューマーがキューをロックして (このコンシューマーもプロデューサーから通知を受信しているため、キューを処理することができる) タスクを処理しようとするが、すでにキューは空になっている。

単体テストとしては一般的なインターリービングではありませんが、こうしたシナリオはクラスをもっと複雑な処理内で使用した場合には十分起こりうるものです。ConTestを使用すれば、単体テスト内でこのシナリオを実行することができます (ところで、このバグを修正するにはどうすればいいでしょうか。修正の際は注意してください。notifyAll() をnotify() に置き換えると、このシナリオ内のバグは解決できますが、他のシナリオでは正常に機能しなくなります)。

ConTestの仕組み

ConTestの基本的な原理はきわめて単純です。計測段階でクラス・ファイルを変換し、ConTestのランタイム機能への呼び出しを選択された場所に挿入します。ConTestが実行されると、こうした場所でコンテキスト・スイッチがときどき発生します。この選択された場所とは、同期ブロックの出入口や共有変数へのアクセスなど、スレッド間の相対順序が実行結果に影響を与える可能性のある場所のことです。yield() やsleep() などのメソッドを呼び出すことにより、コンテキスト・スイッチが実行されます。こうした処理はランダムに実行されるため、実行のたびに異なったインターリービングが発生することになります。典型的なバグを明らかにする際には、ヒューリスティックスが使用されます。

実際にバグが明らかになったかどうかについて、ConTestは関知しないことに注意してください。ConTestはプログラムの動作を予測することはできません。テストを実行し、どの結果が正しくてどの結果がバグであるかを判断するのは、テストを行うユーザーの仕事です。ConTestは、バグを明らかにする手助けをするだけです。しかし、間違った情報を表示することはありません。ConTestを使用して発生させたインターリービングは、JVMのルールに関する限りすべて正しいものです。

これまで見てきたように、同じテストを繰り返し実行することによってテストの質が向上します。実際のところ、一晩中でも同じテストを繰り返し実行することをお勧めします。そうすることにより、考えられるインターリービングのパターンはすべて試したという確証を持つことができます。

ConTestの機能

基本的な手法の他にも、ConTestには並列プログラミングで発生するバグを明らかにするための重要な機能がいくつかあります。

- **同期カバレッジ機能**:単体テストの場合であれば、コード・カバレッジの測定は非常に有効ですが、並列プログラムの場合は間違いの原因になる場合があります。これまでに見てきた2

つの例（同期プロトコルのない名前印刷クラスとバグのある印刷キュー）では、単体テストはすべてのステートメント（InterruptedException処理を除く）を表示しますが、バグを明らかにすることはできません。同期カバレッジ機能はこのギャップを埋めます。この機能により、同期ブロック内で競合がどれだけ発生しているかがわかります。つまり、同期ブロック内で「注意が必要な」処理が行われていないか、すなわち「注意が必要な」インターリービングが行われていないかを知ることができます。詳細については、「参考文献」を参照してください。

- **デッドロック防止機能**:ConTestは、矛盾した順序でロックがネストされていないかどうかを分析することができます。こうした状態は、デッドロックを引き起こす危険性があります。テストを実行した後に、オフラインで分析されます。
- **デバッグ補助機能**:ConTestは、並列プログラムのデバッグに役立つランタイム・レポートを生成することができます。レポートには、ロック状態のレポート（どのスレッドが何をロックしているかや、どのスレッドが待機中かなど）、スレッドの現在の場所についてのレポート、最後に変数に割り当てられた値と最後に変数から読み込まれた値についてのレポートがあります。こうしたクエリーはリモートからも実行することができます。たとえば、別のマシンからConTestを実行しているサーバーの状態をクエリーすることができます。デバッグの際に便利なもう1つの機能に、疑似実行機能があります。これは、特定の処理のインターリービングを繰り返し実行する機能です（必ずそのインターリービングが実行されるわけではありませんが、かなり高い確率で実行されます）。
- **UDPネットワーク混乱機能**:ConTestでは、UDP（データグラム）ソケットによるネットワーク通信の領域に並列混乱という考え方が取り入れられています。UDPプログラムでは、ネットワークの信頼性に依存することはできません。パケットが途中で失われたりパケットの順序が変わったりする場合も考えられますが、こうした状況はアプリケーション上で対応する必要があります。マルチスレッディングと同様に、こうした状況もテストを困難なものにします。通常の環境であればパケットは正しい順序で到着するため、混乱が生じた場合の処理機能を実際にテストすることはできませんが、ConTestは混乱したネットワーク状態をシミュレートしてこの機能を実行することができるため、バグの発見に役立ちます。

課題とこれからの方向性

ConTestはJavaプラットフォーム用に開発されました。C/C++ 用（pthreadライブラリー用）のバージョンは、現在IBM内で使用されていますが、Javaバージョンの全機能が搭載されているわけではありません。同期化はJava言語の一機能であること、バイトコードの実装は非常に容易であること、という2つの理由により、Javaコードのほうが C/C++ よりも簡単にConTestを操作することができます。現在は、MPIなどのライブラリーに対応するConTestの開発作業を進めています。C/C++ 用のConTestを使用してみたい場合は、Shmuel UrまたはYarden Nir-Buchbinderまでお問い合わせください。遅延を追加することでツールが機能するため、ハード・リアルタイム・ソフトウェアもConTestを実行する際に問題となります。ConTestを使用する場合のハード・リアルタイム・ソフトウェアの監視と同じような方法を模索していますが、現在のところ、この問題を解決する方法は見つかっていません。

これからの方向性についてですが、現在はリスナーを基にしたツールのConTestへの適用を可能にするリスナー・アーキテクチャーの発表に取り組んでいます。リスナー・アーキテクチャーを使用することにより、原子性チェック機能やデッドロック検出機能などの分析機能の作成が可能になり、関連するシステムの基盤を記述することなく新しい遅延の仕組みを試すことができるようになります。

まとめ

ConTestは、並列プログラムのテストやデバッグ、カバレッジの測定に使用するツールです。イスラエルのハイファにあるIBM Research Laboratoryで開発されたこのツールは、alphaWorksから使用期間限定の体験版を入手できます。ConTestについて質問がある場合は、Shmuel UrまたはYarden Nir-Buchbinderまでお問い合わせください。

ダウンロード

内容	ファイル名	サイズ
Presentation	j-contestsynch-pres.zip	96KB
Sample code	j-contestexamples.zip	8KB

著者について

Yarden Nir-Buchbinder

Yarden Nir-Buchbindeは、テクニオン工科大学でコンピューター・サイエンスの理学士の学位を取得し、ハイファ大学で哲学修士の学位を取得しています。2000年からイスラエルのハイファにあるIBM Research Labに勤務し、並列テストの技法とツールの開発に取り組んでいます。

Shmuel Ur

IBMのMaster InventorであるShmuel Ur博士は、イスラエルのハイファにあるIBM Research Labに研究者として勤務しています。ソフトウェアのテスト分野に従事し、マルチスレッド・プログラムのカバレッジとテストに取り組んでいます。博士は、テクニオン工科大学とハイファ大学でソフトウェアのテストについての講義を行っています。1994年にカーネギー・メロン大学で、アルゴリズムの最適化と組み合わせ論の博士号を取得しています。また、テクニオン工科大学で理学士と理学修士の学位も取得しています。ハードウェアのテスト、人工知能、アルゴリズム、ソフトウェアのテスト、マルチスレッド・プログラムのテストの各分野で書籍を出版しています。博士は、マルチスレッド・アプリケーション・テストの研究会であるPADTADの議長を務めています。また、IBM Verification Conferenceの議長も努めています。