特定通知パターンを適用してスレッドの実行順序を制御する

通知するスレッドをプログラムによって選択する

Peter Haggar

2001年 12月 01日

クロス・プラットフォームであるという性質は、Javaプラットフォームの主な利点の1つです。ただし、このプラットフォームの振る舞いには、決して望ましいとはいえないものがあります。その1つとして、`notify()` または`notifyAll()` メソッドの呼び出し後に、待機中のスレッドのセットの中のどのスレッドを実行するかを特定できないという問題があります。この問題は、特定通知 --待機中のスレッドのセットから、プログラムによって特定のスレッドを選択し、実行すること-- によって解決することができます。同一または複数のプラットフォーム上での正確かつ一貫性のあるコードの実行を確実なものにするために、この精度が重要になることがよくあります。注: この記事内のすべてのコードは、IBM Java SDK, version 1.3を使用してコンパイルおよび実行しました。

notify() およびnotifyAll() メソッドの使用

Java言語では、イベント通知のために、`wait()`、`notify()`、および`notifyAll()` メソッドが提供されています。`notify()` および`notifyAll()` メソッドは、待機状態にあるスレッドを起こすために使用されます。(`wait()` メソッドを呼び出すと、スレッドは待機状態に入ります。)

`notify()` メソッドでは、現在待機中のスレッドの数にかかわらず、起こされるスレッドは1個のみです。また、待機中のどのスレッドが起こされるかは特定できません。この振る舞いは、1個のスレッドのみが待機中であるとき、または待機中のどのスレッドが起こされてもかまわないときは問題ありません。`notify()` に関しての問題は、複数のスレッドが待機中であり、どのスレッドが起きるかを予測または指定する必要があるときに発生します。JVMによって選択されるスレッドは、ユーザーが制御することはできません。

これに対し、`notifyAll()` メソッドは、待機中のすべてのスレッドを起こします。`notifyAll()` の呼び出し後、待機中の各スレッドは待機状態から抜け出し、ロックを競合します。`notifyAll()` を使用することで、ある条件において待機中のスレッドがある場合に、必ずそれらのスレッドがすべて起こされ、最終的にはすべてが実行されるようにすることができます。ただし、`notify()` と同様に、これらの待機中のスレッドが起こされ、実行される順序を予測または指定することはできません。

notify() またはnotifyAll() の呼び出し時、JVMは、必ずしも優先順位の高いスレッド、または起こすスレッドを選択する時点で最も長い時間待機していたスレッドを選択するとは限りません。また、notify() によって選択されるスレッド、またはnotifyAll() の呼び出し後のスレッドの実行順序は、同じJVM上の同一プログラムの複数回の実行において一定ではありません。このため、これらのメソッドを呼び出した結果スレッドが起こされる順序について仮定することはできません。

これらの問題について説明するために、リスト1 に、5個の異なるロボットを表す5つのスレッドを使用するプログラムを示します。各スレッド (ロボット) は待機状態に入り、ロボットがその仕事を実行するのに必要なすべての情報が含まれるデータの配列を待ちます。このデータの配列が到着すると、notify() が呼び出されます。この時点で、5個のロボット・スレッドのうちの1個が起こされ、その仕事が実行されます。

リスト 1. RobotController and Robot class using arbitrary notification with notify()

```
class Robot extends Thread
{
  private RobotController controller;
  private int robotID;

  public Robot(RobotController cntrl, int id)
  {
    controller = cntrl;
    robotID = id;
  }

  public void run()
  {
    synchronized(controller)
    {
      byte[] data;
      while ((data = controller.getData()) == null)
      {
        try {
          System.out.println("data is null...waiting " + robotID);
          controller.wait();                                    //1
        }
        catch (InterruptedException ie) {}
      }
      //Now we have data to move the robot
      System.out.println("Robot " + robotID + " Working");
    }
  }
}

class RobotController
{
  private byte[] robotData;
  private Robot rbot1;
  private Robot rbot2;
  private Robot rbot3;
  private Robot rbot4;
  private Robot rbot5;

  public static void main(String args[])
  {
    RobotController controller = new RobotController();
    controller.setup();
  }

  public void setup()
```

```
  {
    rbot1 = new Robot(this, 1);
    rbot2 = new Robot(this, 2);
    rbot3 = new Robot(this, 3);
    rbot4 = new Robot(this, 4);
    rbot5 = new Robot(this, 5);
    rbot3.setPriority(Thread.MAX_PRIORITY);                          //2
    rbot1.start();
    rbot2.start();
    rbot3.start();
    rbot4.start();
    rbot5.start();
    begin();
  }

  public void begin()
  {
    for (int i=0;i<5;i++)
    {
      try {
        Thread.sleep(500);
      }
      catch (InterruptedException ie){}
      putData(new byte[10]);
      synchronized(this){
        System.out.println("Calling notify");
        notify();                                                    //3
      }
    }
  }

  public byte[] getData()
  {
    if (robotData != null)
    {
      byte[] d = new byte[robotData.length];
      System.arraycopy(robotData, 0, d, 0, robotData.length);
      robotData = null;
      return d;
    }
    return null;
  }

  public void putData(byte[] d)
  {
    robotData = d;
  }
}
```

このコードには、`Robot` クラスおよび`RobotController` クラスという2つのクラスが含まれます。コードが実行されると、`RobotController` クラスが5個の`Robot` オブジェクトを作成します。各`Robot` オブジェクトは、それぞれ別のスレッド上で実行されます。その後、各スレッドが開始され、//1のところで、動作に必要なデータが提供されるまで、すべて待機状態に入ります。

そしてコードは`begin()` メソッドに入り、`notify()` メソッドを5回呼び出すループに入ります。このプロセスにより、必ず最終的に5個のロボットが起こされます。最初の500ミリ秒はスリープ状態になり、その後ロボットがデータの配列を使用できるようになります。(スリープは、必ずすべてのロボットが最初に待機状態になるようにするために実行されます。)ロボットが動作するためのデータが使用可能になると、//3で`notify()` が呼び出され、1個のロボットがそのデータに対して動作を開始できます。

リスト1のコードを複数回実行すると、さまざまな出力が得られます。リスト2および3は、このプログラムを2度別々に実行した出力結果のサンプルです。

リスト2. リスト1のプログラムの初回の出力結果

```
data is null...waiting 3
data is null...waiting 1
data is null...waiting 2
data is null...waiting 4
data is null...waiting 5
calling notify
Robot 3 Working
calling notify
Robot 1 Working
calling notify
Robot 2 Working
calling notify
Robot 4 Working
calling notify
Robot 5 Working
```

リスト3. リスト1のプログラムの2回目の出力結果

```
data is null...waiting 1
data is null...waiting 2
data is null...waiting 3
data is null...waiting 4
data is null...waiting 5
calling notify
Robot 1 Working
calling notify
Robot 2 Working
calling notify
Robot 3 Working
calling notify
Robot 4 Working
calling notify
Robot 5 Working
```

プログラムの複数回の実行において、待機中のスレッドが実行される順序が異なることに注意してください。`notify()` の各呼び出し後に5個のロボットが起こされる順序を気にしない場合は、このコードで問題ありません。しかし、ある1個のロボットを最初に起こしたい場合、または単に特定の順序でロボットを起こしたい場合は、問題が発生します。提供されるデータに基づいてロボットの実行順序を制御する必要がある状況について考えてみましょう。たとえば、ロボットが、それぞれに異なる重量の材料を処理するよう設計されていると仮定します。さらに、提供されるデータには、配列に含まれるコマンドを実行した場合に、ロボットがどれだけの重量を移動するかについての情報が含まれるとします。この情報に基づき、他のロボットに優先して特定のロボットを選択する必要があるかもしれません。

データにより材料の重量がロボット3に適していることが示されているため、`notify()` が呼び出されたときに最初にロボット3が実行されるようにしたいとします。これをプログラムに基づいて実行するための明白な方法はないため、スレッド3の優先順位を最高に設定しようとするかもしれません (リスト1の //2を参照)。この場合、IBM Java SDK, version 1.3使用時のほとんどの場合において、ロボット3が最初に実行されます。ただし、リスト3の出力結果を見るとわかるように、必ずそうであるという保証はありません。

JVMにより優先順位が最も高いスレッドが常に起こされているらしいことがわかった場合でも、Java言語仕様 (JLS) において、待機中のスレッドが起こされる順序は不定であることが明確に述べられているため、どの場合も必ずこうなると確信することはできません。また、このコードが実行される可能性のある他のJVMについて、やはり何の保証もありません。

`notifyAll()` メソッドの振る舞いに関しても同様の問題が発生します。リスト1のコードを修正して`notifyAll()` を使用するには、以下の3点を変更します。

1. `begin()` メソッドから`for` ループを削除します。`notifyAll()` を呼び出すことによりすべてのスレッドが解放されるため、ループは不要です。
2. //3の`notify()` の呼び出しを、`notifyAll()` の呼び出しに置き換えます。
3. `getData()` メソッドから、`robotData = null;` ステートメントを削除します。ここでは、1個のロボットだけではなく、すべてのロボットについてのコマンドがデータに含まれると仮定します。

`notifyAll()` の呼び出しにより、すべてのロボットが起こされ、それぞれの仕事が実行されます。ここでも、すべてのロボットを起こし、その実行順序について気にしない場合は、これで問題ありません。しかし、これらのスレッドが起こされる順序を制御する必要がある場合は、`notify()` のときと同じ問題が発生します。これらのスレッドが起こされる順序は不定です。

リスト1のコードを修正して`notifyAll()` を使用し、プログラムを数回実行しました。スレッド3の優先順位は、このスレッドを最初に実行させるために最高に設定されたままになっていることに注意してください。ほとんどの実行において、スレッド3が最初に実行され、リスト4に示すような出力が得られます。ただし、同じプログラムをさらに実行した結果、リスト5に示すような出力が得られました。

リスト4. notifyAll() を使用したリスト1のプログラムの初回の出力結果

```
data is null...waiting 3
data is null...waiting 1
data is null...waiting 2
data is null...waiting 4
data is null...waiting 5
Calling notifyAll
Robot 3 Working
Robot 1 Working
Robot 2 Working
Robot 4 Working
Robot 5 Working
```

リスト5. notifyAll() を使用したリスト1のプログラムの2度目の出力結果

```
data is null...waiting 1
data is null...waiting 3
data is null...waiting 2
data is null...waiting 4
data is null...waiting 5
Calling notifyAll
Robot 1 Working
Robot 3 Working
Robot 2 Working
Robot 4 Working
Robot 5 Working
```

リスト4および5は、プログラムの複数回の実行において、待機中のスレッドの実行順序が異なることを示しています。さらに、`notifyAll()` が呼び出されるたびに、すべての待機中のスレッドが起こされ、ロックを競合することも示しています。

`notify()` または`notifyAll()` を呼び出した結果、どのスレッドが実行されるかをプログラムによって制御するための方法が必要です。この判断をJVMの気まぐれに任せるのではなく、我々の手にゆだねられるようにすることは、我々にはより大きな制御力が得られ、ユーザーにとってもより確実かつフレキシブルな実装が提供されることを意味します。

特定通知により問題を解決する

`notify()` または`notifyAll()` の呼び出しによって待機中のスレッドが起こされる順序を制御するには、異なるオブジェクトにおける待機中のスレッドの同期化が必要です。リスト1のコード内のすべてのスレッドは、`RobotController` オブジェクトについて同期化されています。`notify()` または`notifyAll()` が呼び出されると、`RobotController` オブジェクトの`synchronized` ブロック内で同期化が実行されます。このため、5個のスレッドのすべてが待機中で、`RobotController` の`synchronized` ブロック内で`notify()` が呼び出されると、5個のスレッドから1個が選択されます。`notifyAll()` が呼び出されると、待機中の5個のスレッドすべてが最終的には実行されますが、その実行順序を制御することはできません。

notify() を使用した特定通知

待機中のスレッドの実行順序を制御するには、個別に通知する各スレッドまたは一緒に通知するスレッドのセットについて別々のロック・オブジェクトを作成する必要があります。このため、リスト1のロボットの例において、提供されるデータに基づき、毎回特定の順序でスレッドに通知する必要があると仮定します。これを実行するには、それぞれのオブジェクトに各スレッドをロックし、それらのオブジェクトに対して、その対応するスレッドを実行させる順序で通知する必要があります。リスト1のコードを特定通知の手法を加えて修正したバージョンであるリスト6のコードについて考えてみましょう。

リスト 6. Using specific notification with notify()

```
import java.util.*;                                          //1
class Robot extends Thread
{
  private RobotController controller;
  private int robotID;
  private byte[] lock;                                       //2

  public Robot(RobotController cntrl, int id)
  {
    controller = cntrl;
    robotID = id;
  }

  public byte[] getLock()                                    //3
  {                                                          //4
    return lock;                                             //5
  }                                                          //6

  public void run()
  {
    lock = new byte[0];                                      //7
    synchronized(lock)                                       //8
    {
      byte[] data;
```

```
      while ((data = controller.getData()) == null)
      {
        try {
          System.out.println("Thread " + robotID + " waiting");
          lock.wait();                                          //9
        }
        catch (InterruptedException ie) {}
      }
      //Now we have data to move the robot
      System.out.println("Robot " + robotID + " Working");
    }
  }
}

class RobotController
{
  private byte[] robotData;
  private Vector threadList = new Vector();                     //10
  private Robot rbot1;
  private Robot rbot2;
  private Robot rbot3;
  private Robot rbot4;
  private Robot rbot5;

  public static void main(String args[])
  {
    RobotController controller = new RobotController();
    controller.setup();
  }

  public void setup()
  {
    rbot1 = new Robot(this, 1);
    rbot2 = new Robot(this, 2);
    rbot3 = new Robot(this, 3);
    rbot4 = new Robot(this, 4);
    rbot5 = new Robot(this, 5);
    threadList.addElement(rbot1);                               //11
    threadList.addElement(rbot2);                               //12
    threadList.addElement(rbot3);                               //13
    threadList.addElement(rbot4);                               //14
    threadList.addElement(rbot5);                               //15
    rbot3.setPriority(Thread.MAX_PRIORITY);
    rbot1.start();
    rbot2.start();
    rbot3.start();
    rbot4.start();
    rbot5.start();
    begin();
  }

  public void begin()
  {
    for (int i=4;i>=0;i--)                                  //16
    {
      try {
        Thread.sleep(500);
      }
      catch (InterruptedException ie){}
      putData(new byte[10]);

      Robot rbot = (Robot)threadList.elementAt(i);              //17
      byte[] robotLock = rbot.getLock();                        //18
      synchronized(robotLock) {                                 //19
        System.out.println("Calling notify");
        robotLock.notify();                                     //20
      }
```

```
    }
  }

  public synchronized byte[] getData()                              //21
  {
    if (robotData != null)
    {
      byte[] d = new byte[robotData.length];
      System.arraycopy(robotData, 0, d, 0, robotData.length);
      robotData = null;
      return d;
    }
    return null;
  }

  public void putData(byte[] d)
  {
    robotData = d;
  }
}
```

特定通知の手法を加えるためにリスト1から変更されたコード行は、リスト6の //1から //21までのコメント行に示されています。//2でRobot クラスによりロック・インスタンス変数が追加され、//3から //6ではgetLock() メソッドによりこの値が返されます。run() メソッド内では、//7で、ロック変数が0要素のバイト配列に初期化されます。この配列はロック・オブジェクトとして動作します。続いて、//8でsynchronized ブロックがロック・オブジェクトにロックされ、//9では同じオブジェクト上で待機します。ロックはrun() メソッド内で割り振られるため、各ロボット (スレッド) が確実に別々のオブジェクトにロックされます。ロック・インスタンス変数は、スレッドごとに異なるオブジェクトに初期化されるため、ロックはすべて一意です。

RobotController クラスにおいて、作成された各スレッドを保存するために、//10でVector を作成します。//11から //15で、Vector にスレッドが追加されます。//16で、begin() メソッドには、作成時と逆の順序でスレッドに通知するfor ループが含まれます。//17ではfor ループにおいて、Vector からスレッドが取り出されます。そして //18で、getLock() メソッドを使って、そのスレッドが使用している特定のロック・オブジェクトの問い合わせが実行されます。その後、コードは //19でロック・オブジェクトのsynchronized ブロックに入り、//20でロック・オブジェクトのnotify() メソッドを呼び出します。この特定のロック・オブジェクト上で同期化されるスレッドは1個のみであるため (各スレッドには専用のロックがある)、notify() の呼び出しにより待機中の唯一のスレッドが起こされ、プログラムによって起こされるスレッドを選択することが可能になります。ループを変更することにより、希望する任意の順序でスレッドに通知することができます。

もう1つの変更点として、//21のgetData() メソッドをsynchronized として宣言し、ロボット・スレッドによりこのメソッドが並行して実行されないようにする必要があります。リスト1のコードについては、getData() メソッドが属するRobotController オブジェクトで各スレッドが同期化されたため、この問題はありませんでした。

リスト6のコードを実行すると、ほとんどの場合、リスト7に示すような出力が得られます。通知の順序は常に同じです。ただし、各スレッドが待機状態に入る順序は異なることがあります。5個のスレッドの中でスレッド3の優先順位が最も高いため、ほとんどの場合、スレッド3が最初に待機状態に入りますが、リスト8に示すように、必ずしもそうではありません。重要なのは、スレッ

ドが常にコードで指定した順序で起こされることであり、これは、スレッドへの通知は、プログラムによって制御可能であることを意味します。

リスト7. リスト6のプログラムの初回の出力結果

```
Thread 3 waiting
Thread 1 waiting
Thread 2 waiting
Thread 4 waiting
Thread 5 waiting
Calling notify
Robot 5 Working
Calling notify
Robot 4 Working
Calling notify
Robot 3 Working
Calling notify
Robot 2 Working
Calling notify
Robot 1 Working
```

リスト8. リスト6のプログラムの2度目の出力結果

```
Thread 1 waiting
Thread 2 waiting
Thread 3 waiting
Thread 4 waiting
Thread 5 waiting
Calling notify
Robot 5 Working
Calling notify
Robot 4 Working
Calling notify
Robot 3 Working
Calling notify
Robot 2 Working
Calling notify
Robot 1 Working
```

notifyAll() を使用した特定通知

リスト9 は、リスト6のコードを`notifyAll()` 使用時に動作するように修正したものです。特定通知とともに`notifyAll()` を使用するためにリスト9で変更されたコード行は、//1から //16までのコメント行によって示されています。

リスト 9. Using specific notification with notifyAll()

```
import java.util.*;
class Robot extends Thread
{
  private RobotController controller;
  private int robotID;
  private byte[] lock;

  public Robot(RobotController cntrl, int id, byte[] lockObj)    //1
  {
    controller = cntrl;
    robotID = id;
    if (lockObj == null)                                         //2
      lock = new byte[0];                                        //3
    else                                                         //4
      lock = lockObj;                                            //5
  }
```

```
  public byte[] getLock()
  {
    return lock;
  }

  public void run()
  {
    synchronized(lock)
    {
      byte[] data;
      while ((data = controller.getData()) == null)
      {
        try {
          System.out.println("Thread " + robotID + " waiting");
          lock.wait();
        }
        catch (InterruptedException ie) {}
      }
      //Now we have data to move the robot
      System.out.println("Robot " + robotID + " Working");
    }
  }
}

class RobotController
{
  private byte[] robotData;
  private Vector threadList = new Vector();
  private Robot rbot1;
  private Robot rbot2;
  private Robot rbot3;
  private Robot rbot4;
  private Robot rbot5;
  private byte[] lock1;                                    //6
  private byte[] lock2;                                    //7

  public static void main(String args[])
  {
    RobotController controller = new RobotController();
    controller.setup();
  }

  public void setup()
  {
    lock1 = new byte[0];                                   //8
    lock2 = new byte[0];                                   //9
    rbot1 = new Robot(this, 1, lock1);                     //10
    rbot2 = new Robot(this, 2, lock2);                     //11
    rbot3 = new Robot(this, 3, null);                      //12
    rbot4 = new Robot(this, 4, lock1);                     //13
    rbot5 = new Robot(this, 5, lock2);                     //14
    threadList.addElement(rbot1);
    threadList.addElement(rbot2);
    threadList.addElement(rbot3);
    threadList.addElement(rbot4);
    threadList.addElement(rbot5);
    rbot3.setPriority(Thread.MAX_PRIORITY);
    rbot1.start();
    rbot2.start();
    rbot3.start();
    rbot4.start();
    rbot5.start();
    begin();
  }

  public void begin()
  {
```

```
    for (int i=0;i<3;i++)                                          //15
    {
      try {
        Thread.sleep(500);
      }
      catch (InterruptedException ie){}
      putData(new byte[10]);

      Robot rbot = (Robot)threadList.elementAt(i);
      byte[] robotLock = rbot.getLock();
      synchronized(robotLock) {
        System.out.println("Calling notifyAll");
        robotLock.notifyAll();                                     //16
      }
    }
    System.exit(0);
  }

  public synchronized byte[] getData()
  {
    if (robotData != null)
    {
      byte[] d = new byte[robotData.length];
      System.arraycopy(robotData, 0, d, 0, robotData.length);
      return d;
    }
    return null;
  }

  public void putData(byte[] d)
  {
    robotData = d;
  }
}
```

//1から //5で変更されているRobot クラス・コンストラクターにより、ロボット (スレッド) ごとに固有のロック・オブジェクトを指定すること、または複数のスレッドに同じロック・オブジェクトを持たせることが可能になっています。同じロック・オブジェクトでスレッドをグループ化する場合、notifyAll() の呼び出しにより、このグループ内のスレッドが任意に起こされます。

//6から //9では、2個のロック・オブジェクトを作成、保存するために、RobotController クラスが変更されています。//10から //14では、5個のロボット・スレッドが作成されます。ロボット1と4は、ロック・オブジェクトlock1 を共有します。ロボット2と5は、ロック・オブジェクトlock2 を共有します。ロボット3はロック・オブジェクトを指定しません。このため、ロボット3は、//3において、Robot コンストラクター内でユニークなロック・オブジェクトを取得します。このコードにおけるスレッド間でのロックの共有の理屈を説明すると、以下のようになります。すなわち、スレッドが起こされる順序は制御したいが、他のスレッドとの関係においてスレッド1と4の実行を制御できれば、スレッド1と4の実行順序が前後しても構わないということです。このことは、スレッド2とスレッド5にも当てはまります。スレッド3には専用のロックがあり、このことは、他のすべてのスレッドとの関係において、このスレッドが起こされるときを制御したいということを示しています。

今、3個のロック・オブジェクトと5個のスレッドがあります。2個のスレッドが最初のロック・オブジェクトを、2個のスレッドが2番目のロック・オブジェクトをそれぞれ共有し、1個のスレッドについては共有ではなく専用のロック・オブジェクトがあります。これでこれらのスレッドに対して正しい順序で通知するようにプログラム化することが可能になります。//15のbegin() メソッ

ドにおいて、コードは3個のロック・オブジェクトについてループ処理を実行し、これらに対して順番に通知します。

- まず、//16において、`notifyAll()` の呼び出しにより`lock1` に通知されます。この手順の結果、スレッド1と4が起こされます。これらの2個のスレッドのどちらを先に実行するかについては制御できませんが、他の3個のどのスレッドよりも先にスレッド1またはスレッド4のどちらかを実行することを指定できました。より細かく制御する必要がある場合は、スレッド3についての処理と同様に、スレッド1および4の両方について別々のロック・オブジェクトを作成し、そのオブジェクトに別々に通知します。
- 次に、`lock2` 上で`notifyAll()` が呼び出され、スレッド2およびスレッド5が起こされます。
- 最後に、`notifyAll()` の最後の呼び出しにより、スレッド3が起こされます。リスト9のコードの出力結果のサンプルをリスト10に示します。

リスト10. リスト9のコードの出力結果

```
Thread 3 waiting
Thread 1 waiting
Thread 2 waiting
Thread 4 waiting
Thread 5 waiting
Calling notifyAll
Robot 1 Working
Robot 4 Working
Calling notifyAll
Robot 2 Working
Robot 5 Working
Calling notifyAll
Robot 3 Working
```

この出力結果は、特定通知と`notifyAll()` を使用することにより、待機中のスレッドのセットの実行順序を制御できることを示しています。5個すべてのスレッドをあらかじめ決めた順序で実行させる必要がある場合、リスト6で使用した手法で、`notify()` を使用します。通知の目的によってスレッドをグループ化する必要がある場合にのみ`notifyAll()` を使用します。

結論

待機中のスレッドの実行順序を指定する必要があるときがあります。Javaプラットフォームでは、本質的にこの必要性がサポートされません。この記事で説明した特定通知のパターンを実装することにより、待機中のスレッドの実行順序を制御するためのプログラムの機構を作成することができます。この手法は必然的に、通知する必要がある各スレッドまたはスレッドのセットに対する個別のロック・オブジェクトの作成を伴います。1個のロック・オブジェクトに対して複数のスレッドがあるときは、`notifyAll()` を使用してそれらのスレッドを起こします。各スレッドについて専用のロック・オブジェクトがある場合は、`notify()` を使用します。

特定通知のサポートを追加することにより、待機中のスレッドの実行を直接制御することが可能になります。しかも、複雑さや実行速度といった点に関して、プログラムへの影響はほとんどありません。

関連トピック

- Peter Haggarは、自著 「Practical Java Programming Language Guide」(Addison-Wesley、2000年) で、マルチスレッド化の問題やプログラミングの手法についての章を含む、さまざまなJavaプログラミングのトピックについて説明しています。
- Brian Goetzは、3回連載の 「システム負荷を軽減したスレッド化」で、Java言語を使用した並行プログラミングという難題の解明に取り組んでいます。
  - 第1回 "同期化を敵視することはありません" では、競合のない同期化のパフォーマンスへの影響が、広く信じられているほど大きくないことについて説明されています。
  - 第2回 "競合を低減させる"では、競合のある同期化がプログラムのパフォーマンスに与える影響を軽減するテクニックについて説明されています。
  - 第3回 "常に共用が最善とは限らない" では、スレッドをスレッドごとのデータに関連付けるプロセスを大いに助ける、パワフルだが相応の評価が得られていないクラスThreadLocalについて説明されています。
- Alex Roetterは、彼の記事 "マルチスレッド化Javaアプリケーションの作成" (developerWorks、2001年2月) の中で、Java Thread APIを紹介し、マルチスレッド化に含まれる問題について概説し、一般的な問題の解決策を提示しています。
- developerWorks Javaテクノロジー・ゾーン に、その他のJava参考文献があります。