マルチスレッド化Javaアプリケーションの作成

コンカレント・プログラミングでよく発生する問題を回避する方法

Alex Roetter

2001年 2月 01日

Java Thread APIにより、プログラマーは、複数のプロセッサーを使用する利点を生かし、ユーザーが要求する会話形式を維持しながらバックグラウンド・タスクを実行するアプリケーションを作成することができます。AlexRoetterは、マルチスレッドの概要をまとめて、Java Thread APIを紹介し、よく発生する問題についてのソリューションを提示しています。

AWTやSwingを使用するグラフィック・プログラムを作成する場合、よほど平凡なプログラムでない限り、マルチスレッド化が必要です。スレッド・プログラミングには、難しい点が多く、経験の浅い開発担当者は、アプリケーションが正しく作動しなかったり、デッドロックが発生したりするような問題に陥りがちです。

この記事では、マルチスレッド化に関連した問題に焦点をあて、よく発生する問題についてのソリューションを提示します。

スレッドとは

プログラムやプロセスには、プログラム・コードに従って命令を実行する複数のスレッドを組み込むことができます。1台のコンピューターで実行できるマルチプロセスのように、マルチスレッドは、作業を並列(parallel)に行っているように見えます。マルチプロセッサー・マシンで実行すると、実際に、並列(parallel)に作業を実行することができます。プロセスと違い、スレッドは、同じアドレス・スペースを共用します。すなわち、スレッドは、同じ変数やデータ構造を読み書きすることができます。

マルチスレッド化プログラムを作成する際には、あるスレッドが他のスレッドの作業を妨げないように細心の注意を払う必要があります。このアプローチを、従業員が、共用のオフィス・リソースを使用したり、お互いに話し合う必要がある場合を除いて、各人が独立して並列的に(parallel)作業を行うオフィスの様子にたとえることができます。ある従業員は、他の従業員が「聞いている」場合だけその従業員に話すことができ、両者は同じ言語で話します。また、従業員は、コピー機が空いていて使用可能な状態(仕掛かり中のコピーや紙詰まりなどが無い状態)になるまで、コピー機を使用できません。この記事を読むと、どのようにすれば、整然と機能する組織の従業員のように、Javaプログラムでうまく調整してスレッドを使用することができるかがお判りになるでしょう。

マルチスレッド化プログラムにおいて、スレッドは、使用可能な実行可能スレッドのプールから獲得され、使用可能なシステムCPUで実行されます。OSでは、スレッドをプロセッサーから作動可能キューまたはブロッキング・キューのどちらにでも移動できます。この状態を、スレッドがプロセッサーを「与えられている」といいます。一方、連携モデルまたはプリエンプティブ・モデルでは、Java仮想計算機(JVM) がスレッドの移動を管理します。スレッドは、作動可能キューからプロセッサーに移動され、プロセッサーでプログラム・コードの実行を開始することができます。

連携スレッド化 により、スレッドは、他の待ち状態のスレッドに、いつ、プロセッサーを空け渡すかを判別することができます。アプリケーション開発担当者は、スレッドが他のスレッドと効率的に作動するようにするために、スレッドがいつ他のスレッドにプロセッサーを渡すかを正確に決定します。欠点は、悪意のあるスレッドや出来の悪いスレッドが、使用可能なCPU時間をすべて消費して、他のスレッドを妨げる可能性があることです。

プリエンプティブ・スレッド・モデルでは、OSがいつでもスレッドに割り込みをかけることができます。これは、通常、一定の間スレッドを稼動させた後に行われます(これをタイム・スライスといいます)。その結果、スレッドが不公平にプロセッサーを占有することはなくなります。しかし、いつでもスレッドに割り込みをかけられるということは、プログラム開発担当者にとって問題が残ります。オフィスの例で考えてみると、ある従業員が他の従業員より先にコピー機を使用していて、途中で中断したとします。次にコピー機を使用する従業員は、前の従業員のオリジナルや出力コピーがコピー機に置かれている状態でコピーを開始するかもしれません。プリエンプティブ・スレッド・モデルでは、スレッドが共用リソースを適切に使用する必要があり、一方、連携スレッド・モデルではスレッドが実行時間を独占しない必要があります。JVM仕様では特定のスレッド・モデルを指定しないので、Java開発担当者は、両方のモデルを考慮してプログラムを作成しなければなりません。スレッドおよびスレッド間の通信について少し説明した後、各モデルにおけるプログラムの設計方法についてお話します。

スレッドとJava言語

Java言語を使用してスレッドを作成する場合には、`Thread` タイプのオブジェクト(またはサブクラス) をインスタンス化して、それに `start()` メッセージを送信します。(プログラムは、`Runnable` インターフェースをインプリメントしているオブジェクトに`start()` メッセージを送信することができます。)各スレッドの性質の定義は、`run()` メソッドに含まれています。runメソッドは、従来のプログラムの `main()` に相当するものです。スレッドは、`run()` が制御を戻すまで継続して実行され、戻った時点で消滅します。

ロック

ほとんどのアプリケーションでは、お互いに動作を通信しあい同期化するためにスレッドを必要とします。Javaプログラムでこの操作を行う最も簡単な方法が、ロックを使用する方法です。多重アクセスを防ぐため、スレッドは、リソースを使用する前に、ロックを獲得および解除することができます。一度に1人の従業員しかキーを占有できないコピー機のロックを想像してみてください。キーが無いと、マシンを使用できません。共用変数に関するロックにより、Javaスレッドは、迅速かつ容易に通信および同期化を行えます。オブジェクトに関してロックを保有しているスレッドは、そのオブジェクトにアクセスするスレッドが他に無いことがわかっています。ロックのあるスレッドが優先される場合でも、元のスレッドが起動し、作業を完了して、ロックを解除するまでは、他のスレッドはそのロックを獲得できません。使用中のロックを獲得しようとす

るスレッドは、ロックを保有しているスレッドがロックを解除するまで、スリープ状態に入ります。ロックが解放されると、スリープ状態のスレッドは、作動可能キューに移動します。

Javaプログラミングでは、各オブジェクトに1つのロックがあります。スレッドは、`synchronized` キーワードを使用して、オブジェクトに対するロックを獲得することができます。メソッドまたは同期化されたコード・ブロックは、コードが実行される前にオブジェクトのロックを獲得する必要があるので、指定されたクラスの特定のインスタンスに対しては、一度に1つのスレッドによってしか実行できません。コピー機のたとえを用いると、コピー機が競合しないようにするためには、一度に1人の従業員しかコピー機を使用できないようにして、コピー機の使用を同期化すればよいのです。以下のコードのサンプルは、このことを示しています。コピー機の状態を変更するメソッド(`Copier` オブジェクト内)を同期化メソッドとして宣言することにより、これを行います。`Copier` オブジェクトを使用する必要がある従業員は、`Copier` オブジェクトにつき1つのスレッドしか同期化されたコードを実行できないので、列を作って待たなければなりません。

```
class CopyMachine {

   public synchronized void makeCopies(Document d, int nCopies) {
      //only one thread executes this at a time
   }

   public void loadPaper() {
      //multiple threads could access this at once!

      synchronized(this) {
         //only one thread accesses this at a time
         //feel free to use shared resources, overwrite members, etc.
      }
   }
}
```

きめの細かい (Fine-grain) ロック

オブジェクト・レベルでロックを使用するのは、大まか過ぎる場合があります。共用リソースに対するほんのわずかなアクセスにも、他の同期化されたメソッドへのアクセスを許可せずに、オブジェクト全体をロックする必要があるのでしょうか。オブジェクトに複数のリソースがある場合、1つのスレッドがリソースの一部だけを使用できるようにするために、すべてのスレッドに対し、オブジェクト全体をロックする必要はありません。オブジェクトにあるロックは1つなので、以下に示すように、単なるロックとしてのダミー・オブジェクトを使用することができます。

```
class FineGrainLock {
   MyMemberClass x, y;
   Object xlock = new Object(), ylock = new Object();
   public void foo() {
      synchronized(xlock) {
         //access x here
      }
      //do something here - but don't use shared resources
      synchronized(ylock) {
         //access y here
      }
   }
   public void bar() {
      synchronized(xlock) {
         synchronized(ylock) {
            //access both x and y here
         }
      }
```

```
      //do something here - but don't use shared resources
   }
}
```

これらのメソッドは、`synchronized` キーワードを使用してメソッド全体を宣言して、メソッド・レベルで同期化を行う必要はありません。同期化されたメソッドが獲得するオブジェクト・レベルのロックではなく、メンバー・ロックを使用します。

セマフォー

沢山のスレッドが少しのリソースに対してアクセスする必要が生じることがよくあります。たとえば、多数のスレッドが、クライアント要求に応じるWebサーバーで実行されているとします。これらのスレッドは、データベースに接続する必要がありますが、データベースに接続できる数は限られています。多数のスレッドに対して、効率的にデータベースへの接続を割り当てるにはどうすればよいのでしょう。リソースのプールに対するアクセスを制御する(単なる1つのスレッド・ロックを使用するのではなく)1つの方法) として、セマフォーをカウントする方法があります。セマフォーのカウントは、使用可能なリソースのプールの管理をカプセル化します。単純なロックの最初にインプリメントすると、セマフォーは、使用できるリソース数に初期化されるスレッド・セーフ・カウンターになります。たとえば、セマフォーを使用可能なデータベース接続数に初期化します。各スレッドがセマフォーを獲得するたびに、使用可能な接続数は1つずつ減っていきます。リソースを使いつくすと、セマフォーはリリースされ、カウンターが増やされます。セマフォーによって管理されているすべてのリソースが使用中の場合には、セマフォーを獲得しようとしたスレッドは、リソースが解放されるまで単にブロックされます。

セマフォーは、「消費者 - 生産者問題」の解決に一般的に使用されます。この問題は、あるスレッドが、別のスレッドがこれから使用しようする作業を完了しようとしている際に起こります。消費スレッドは、生産スレッドによるデータの生成完了後にしか、さらに多くのデータを獲得することができません。この方法でセマフォーを使用するためには、初期値がゼロのセマフォーを作成して、消費スレッドをセマフォー上でブロック化します。それぞれの作業単位が完了するたびに、生産スレッドはセマフォーにシグナルを送ります(セマフォーを解放します)。消費スレッドは、データ単位を消費して別のデータ単位が必要になるたびに、再度セマフォーを獲得しようとし、その結果、セマフォーの値は、常に、消費可能な完了作業単位数になります。このアプローチは、消費スレッドを起動し、完了作業をチェックして、使用可能なものが何もなければスリープさせる方法よりも効率的です。

セマフォーは、Java言語では直接サポートされていませんが、オブジェクト・ロックの最初に容易にインプリメントされます。簡単なインプリメンテーションを、以下に示します。

```
class Semaphore {
   private int count;
   public Semaphore(int n) {
      this.count = n;
   }
   public synchronized void acquire() {
      while(count == 0) {
         try {
            wait();
         } catch (InterruptedException e) {
            //keep trying
         }
      }
```

```
      count--;
   }

   public synchronized void release() {
      count++;
      notify(); //alert a thread that's blocking on this semaphore
   }
}
```

よく発生するロック問題

残念ながら、ロックの使用には多くの問題がつきものです。よく発生する問題とそのソリューションをあげてみます。

デッドロック。

デッドロックは、さまざまなスレッドが決してリリースされることがないロックを待っているために、すべての作業が完了しないという、マルチスレッド化における典型的な問題です。2つのスレッドを、フォークとナイフを共用して食べる順番を待たなければならない空腹な2人にたとえてみましょう。彼らは、お互いに2つのロックを獲得する必要があります。1つは、共用しているフォークというリソース、もう1つは共用しているナイフというリソースです。スレッドAがナイフを獲得し、スレッドBがフォークを獲得したとします。スレッドAは、今度は、フォークに対する待ちでブロックされ、スレッドBはスレッドAが持っているナイフに対する待ちでブロックされます。これは説明用に作成した例ですが、この例を通して、このような状況は見つけにくいにも関わらず、よく起こることがおわかりでしょう。すべての場合において、デッドロックを検出して完全になくしてしまうことは難しいのですが、次のいくつかの規則に従えば、システムの設計からデッドロック発生の可能性を取り除くことができます。

- 複数のスレッドが同じ順序でロック・グループを獲得するようにする。このアプローチを使用すると、Xの所有者が、Xを待っているYの所有者を待っているというような問題が無くなります。
- 複数のロックを1つのロックにグループ化する。先程の例では、フォークかナイフを獲得する前に獲得しなければならない銀製品のロックを作成します。
- リソースに、ブロッキングせずに読むことができる変数のラベルを付ける。銀製品ロックの獲得後、スレッドは、銀製品セットがすべて使用できるかどうかを調べるために、変数を検査します。使用できることが判明したら、スレッドは関係のあるロックを獲得できます。使用できないことが判明したら、マスターの銀製品ロックを解放して、後で再度試行します。
- 最も重要なことは、コードを作成する前に、システム全体を設計することです。マルチスレッド化は難しいので、コードの作成を開始する前に設計を完全にしておくことが、検出が困難なロック問題を回避することにつながります。

揮発性変数。

`volatile` キーワードは、コンパイラーを最適化する手段として言語に取り入れられました。例として、次のコードを見てください。

```
class VolatileTest {
   boolean flag;
   public void foo() {
      flag = false;
      if(flag) {
         //this could happen
      }
   }
}
```

最適化コンパイラーは、`if` ステートメントの本体が決して実行されないと判断し、コードのコンパイルを行わないかもしれません。このクラスがマルチスレッドによってアクセスされる場合であっても、前のコードで設定された後で、`if` ステートメントでテストされる前に、別のスレッドによって`flag` が設定される可能性があります。`volatile` キーワードを使用して変数を宣言すると、コンパイル時に変数の値を予測することにより、コードのセクションを最適化しないようにコンパイラーに通知します。

アクセス不能スレッド。
スレッドは、オブジェクト・ロック以外の状態でブロックする必要がある場合があります。入出力は、Javaプログラミングにおけるこの問題の最もよい例です。スレッドがオブジェクト内の入出力呼び出しでブロックされる時に、そのオブジェクトは、他のスレッドに対してまだアクセス可能でなければなりません。そのオブジェクトが原因で、入出力操作のブロッキングを取り消すことがよくあります。同期化メソッドでブロッキング呼び出しを行うスレッドは、このようなタスクを不可能にしてしまうことがあります。オブジェクトの他のメソッドも同期化されている場合、そのオブジェクトは、スレッドがブロックされている間本質的にフリーズされます。他のスレッドは、オブジェクト・ロックを獲得できないので、オブジェクトにメッセージを送信できなくなります(たとえば、入出力操作の取り消しなど)。ブロッキング・コールを行うコードは同期化しないようにしてください。または、同期化されたブロッキング・コードがあるオブジェクトに非同期化メソッドがあるようにしてください。この技法は、結果のコードがまだスレッド保護であるようにするために若干の注意が必要ですが、ロックを保留しているスレッドがブロックされている場合でも、オブジェクトが他のスレッドに対応することができます。

スレッド・モデルに応じた設計

スレッド・モデルがプリエンプティブ・モデルであるか連携モデルであるかは、仮想計算機のインプリメンテーションによって決まり、インプリメンテーションごとに異なります。そのため、Java開発担当者は、両方のモデルで作動するプログラムを作成する必要があります。

プリエンプティブ・モデルでは、前述のように、アトミック・ブロック・コードを除き、コードのどのセクションの途中でもスレッドに割り込みをかけることができます。アトミック・セクションとは、開始されると、スワップ・アウトされる前に現行のスレッドによって終了させられるコード・セグメントです。Javaプログラミングでは、32ビットより小さい変数への割り当てはアトミック・オペレーションで行われ、`double` および`long` (両方とも64ビット)タイプの変数は除外されます。そのため、アトミック・オペレーションは同期化を必要としません。共用リソースに対するアクセスを適切に同期化するためにロックを使用することによって、マルチスレッド化プログラムがプリエンプティブ仮想計算機で正しく作動することが十分保証されます。

連携スレッドの場合、スレッドが他のスレッドの実行時間を奪わないようにするために、スレッドが日常処理として確実にプロセッサーを解放するようにするのは、プログラマーの仕事です。

これを行う方法の1つに、`yield()` コールがあります。これは、現行のスレッドをプロセッサーから作動可能キューに移動させます。もう1つの方法は、`sleep()` コールです。これは、スレッドがプロセッサーを渡すようにし、sleepへの引数に指定された時間が経つまでスレッドを実行できないようにします。

お考えのように、コードの任意の場所に単にこれらのコールを置くだけでは必ずしも作動しません。スレッドがロック状態を保持している場合(スレッドは同期化メソッドまたはコード・ブロックにあるので)、スレッドは、`yield()` をコールしても、ロックを解除しません。これは、実行中のスレッドが同じロックを待っている他のスレッドにロックを渡しても、待ち状態になっているスレッドが実行されないことを意味します。この問題を緩和するためには、スレッドが同期化メソッドにない時には`yield()` をコールします。コードが非同期化メソッド内の同期化ブロック内で同期化されるようにして、そのブロックの外側で`yield()` をコールします。

もう1つのソリューションは、`wait()` のコールです。これにより、プロセッサーは現在オブジェクト内にあってそのオブジェクトに属しているロックを解放します。このアプローチは、オブジェクトがその1つのロックしか使用していないので、オブジェクトがメソッド・レベルで同期化されている場合には、うまく作動します。Fine-grainロックを使用する場合には、`wait()` を実行してもロックは解除されません。また、`wait()` に対するコールでブロックされているスレッドは、待ち状態のスレッドを作動可能キューに移動する`notify()` を別のスレッドがコールするまで起動しません。`wait()` コールでブロックされているすべてのスレッドを起動するには、1つのスレッドで`notifyAll()` をコールします。

スレッドとAWT/Swing

SwingまたはAWTあるいはその両方を使用するGUIを使ったJavaプログラムでは、AWTイベント・ハンドラーは独自のスレッドで実行されます。開発担当者は、このGUIスレッドが時間がかかる作業を行うのを妨げないように注意しなければなりません。このスレッドは、ユーザー・イベントの処理およびGUIの再ドローを行う必要があるからです。言い換えると、GUIスレッドがビジーである場合には、プログラムはフリーズしているように見えます。MouseListenersやActionListenersのようなSwingコールバックはすべて、Swingのスレッドにより(適切なメソッドをコールして)通知されます。このアプローチでは、リスナーのコールバック・メソッドに作業を行わせるための別のスレッドを作成させて、リスナーが行うすべての実質的な作業を実行しなければなりません。目的は、リスナーのコールバックが迅速に戻るようにすることです。こうすることによって、Swingのスレッドが他のイベントに対応できるようになります。

Swingのスレッドを非同期的に実行したり、イベントへの対応や画面の再ドローを行ったりする場合、他のスレッドはどのようにしてSwingの状態を安全に変更することができるのでしょうか。前述のように、SwingコールバックはSwingのスレッド内で実行されます。そのため、Swingコールバックは、安全にSwingデータを変更し、画面にドローすることができます。

しかし、Swingコールバックの結果として起こる変更以外の他の変更はどうでしょう。Swingのスレッド以外のスレッドにSwingデータを変更させると、スレッドは保護されません。Swingでは、この問題を解決するために、`invokeLater()` と`invokeAndWait()` の2つのメソッドが用意されています。Swingの状態を変更するには、これらのメソッドのいずれかをコールして、正しく作動する`Runnable`オブジェクトを渡すだけで十分です。`Runnable` オブジェクトは、通常、独自のスレッドなので、このオブジェクトは実行されるスレッドとして作成されるとお考えになるでしょう。

実際には、そうではありません。このオブジェクトもスレッドを保護しないのです。Swingは、このオブジェクトをキューに入れて、その後任意の時点で、その実行メソッドを実行します。こうすることによって、Swingの状態スレッドへの変更を保護します。

要約

Java言語の設計には、単純なアプレット以外では、マルチスレッド化が不可欠です。特に、入出力およびGUIプログラミングでは、ユーザーの使い勝手のよさを確保するためにマルチスレッド化が必要です。この記事で説明した簡単なルールに従って、プログラミングに着手する前に、共用リソースへのアクセスも含め、システムを完全に設計することにより、発生しやすく見付けにくいスレッドの問題を回避できます。

関連トピック

- Java 2プラットフォーム、標準版、バージョン1.3のAPI仕様は、Java 2 API Documentation を参照してください。
- JVMによるスレッドおよびロックの処理について詳しくは、The Java Virtual Machine Specification を参照してください。
- Allen HolubのTaming Java Threads (APress、2000年6月) は、優れた解説書です。
- Allenの記事もしも自分が王様だったら: Javaプログラム言語のスレッド化不具合への解決策提案 (developerWorks、2000年10月) では、氏が素晴らしい言語の弱点と指摘している点について記述されています。