Java プログラムでのメモリー・リークの処理

メモリー・リークの検出とその予防

Jim Patrick
Advisory Programmer
IBM Pervasive Computing

2001年 2月 01日

Java プログラムでメモリー・リークが発生するのでしょうか？その通りです。一般に信じられていることとは異なり、Java プログラミングでもメモリーの管理を考慮する必要があります。この記事では、メモリー・リークの原因は何か、そしてそれが問題になるのはどんな場合かを説明します。また、プロジェクトでメモリー・リークの問題に取り組む上での実際的な例も含まれています。

Java プログラムでメモリー・リークが発生しているかどうかを見分ける方法

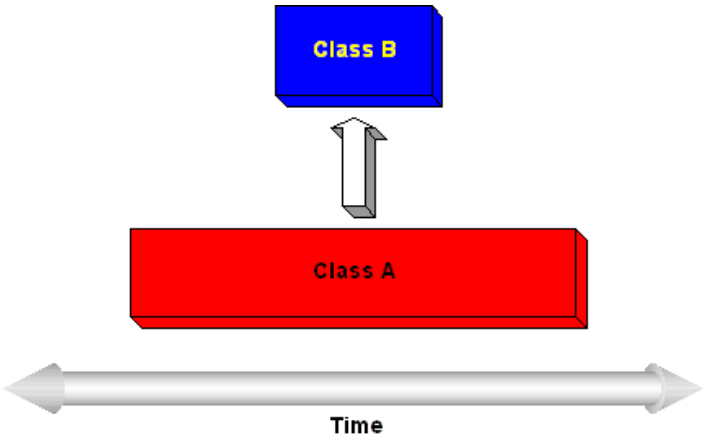
メモリーの割り振りや解放の面倒な処理をしなくてよいということが、Java などのプログラミング言語を使用することの利点の 1つであることは、プログラマーならだれでもよくご存じのことでしょう。プログラマーがオブジェクトを作成すると、アプリケーションでそのオブジェクトが不要になった時点で Java がそれを削除してくれます。このメカニズムは、ガーベッジ・コレクションと呼ばれています。Java では、このようなプロセスのおかげで他のプログラミング言語で悩みの種となってきた問題の 1つであるメモリー・リークが解決されているということになります。しかし、本当に解決されているのでしょうか？

話を進める前に、ガーベッジ・コレクションが実際にはどのような処理なのかをまず復習しておきましょう。ガーベッジ・コレクションで実行するべきことは、アプリケーションで不要になったオブジェクトを検出し、それ以降にそれらのオブジェクトにアクセスしたり参照したりしなくなった時点でそれを削除する、ということです。ガーベッジ・コレクションは、Java アプリケーションの存続期間全体にわたって存在し続けるルート・ノードやルート・クラスから開始し、参照されるあらゆるノードを検索します。ノード全体を網羅するにつれて、どのオブジェクトが実際に参照されているかが記録されます。参照されなくなったクラスがあれば、それがガーベッジ・コレクションの対象に含められます。それらのオブジェクトが削除される時点で、それらによって使用されているメモリー・リソースが Java 仮想マシン (JVM) に戻されます。

そのため、Java コードにおいては、プログラマーがメモリーのクリーンアップを管理する必要がなく、使用されなくなったオブジェクトのガーベッジ・コレクションが自動的に実行される、と

いうのは本当です。しかし、ここで重要なことは、オブジェクトが使用されていないものとして数えられるのは、それがもう参照されなくなった時点である ということです。図 1 をご覧ください。

図 1. 使用されていないが参照されている場合



この図には、ある Java アプリケーションに含まれる 2つのクラスが示されており、それぞれ存続期間が違っています。まずクラスA のインスタンスが生成され、かなり長い期間にわたって存在し続けます。プログラムの存続期間全体にわたるとしてもよいでしょう。その後、クラスB が作成され、クラスA は、新たに作成されたそのクラスへの参照を追加します。ではここで、クラスB がユーザー・インターフェース・ウィジェットであり、表示された後、結局はユーザーによって閉じられるものとしましょう。クラスB は不要になったものの、クラスA に含まれるクラスB への参照がクリアされないとすれば、次にガーベッジ・コレクションが実行されたとしても、クラスB は存在し続け、メモリー・スペースを占有し続けることになります。

メモリー・リークが問題になるのはどういう場合か

作成したプログラムの実行開始後、しばらくして`java.lang.OutOfMemoryError` を受け取るようであれば、メモリー・リークの可能性が高いと言えます。そのような明らかな場合以外に、メモリー・リークが問題になるのはどういう場合でしょうか？ 完ぺき主義のプログラマーなら、ありとあらゆる メモリー・リークを調査し、修正しなければならないと答えることでしょう。しかし、一足飛びにそのような結論に達する前に、プログラムの存続期間やメモリー・リークのサイズなど、考慮するべきいくつかの点があります。

1つの可能性として、アプリケーションの存続期間中にガーベッジ・コレクションがまったく実行されない場合を考えてください。たとえプログラムの中で `System.gc()` を明示的に呼び出すとしても、JVM がガーベッジ・コレクションのルーチンをいつ呼び出すか、あるいはそもそも呼び出すかどうかについての保証は何もありません。一般にガーベッジ・コレクションは、プログラムで現在使用可能なメモリーより多くのメモリーが必要になる時点より前に自動的に実行されることはありません。余分のメモリーが必要になった時点で JVM は、ガーベッジ・コレクションのルーチンを呼び出すことによってさらに多くのメモリーを使用可能にしようとします。その試みによっても十分なリソースが解放されなかったなら、JVM はオペレーティング・システムから余分のメモリーを入手します。可能な最大量に達するまで、この操作が繰り返されます。

たとえば、設定を変更するための簡単なユーザー・インターフェース要素を表示する小さな Java アプリケーションがあり、そのプログラムでメモリー・リークが発生しているとしましょう。そのアプリケーションがクローズされるまで、ガーベッジ・コレクションが起動されることさえない、という可能性があるのです。というのは、JVM にはプログラムで必要となるオブジェクトをすべて作成するのに十分なメモリーがあり、しかも使用可能なメモリーが残っているからです。この場合、そのプログラムの実行中に、もはや不要になったオブジェクトが引き続きメモリーを占有していても、実際上は問題となりません。

開発中の Java コードがサーバー上で 24時間実行されるものであれば、メモリー・リークの問題の重要性は、前述の設定ユーティリティーの場合よりずっと高くなります。コードのどこかでほんのわずかなメモリー・リークが発生しているだけであっても、それを実行し続けるなら、JVM は使用可能なメモリーをすべて使い尽くしてしまう結果になります。

逆のケースもあります。プログラムの存続期間は比較的短いものの、たくさんの一時オブジェクト (または少数でも大量のメモリーを消費するオブジェクト) を割り振って、不要になった後も参照を解除しない Java コードがあれば、それによってメモリーの限界に達する可能性があります。

最後に考慮する点は、メモリー・リークはまったく問題ではないということです。C++ などの他の言語の場合に発生するメモリー・リークでは、メモリーが失われ、それ以降にオペレーティング・システムに戻されることが決してないという状態になってしまいますが、Java のメモリー・リークはそれほど危険なものではありません。Java アプリケーションの場合、不要になったオブジェクトは、オペレーティング・システムから JVM に与えられたメモリー・リソースの中に存在しています。ですから、理論的には、Java アプリケーションとその JVM がクローズされると、割り振られたメモリーはすべてオペレーティング・システムに戻されることになります。

アプリケーションでメモリー・リークが発生しているかどうかを判断する

Windows NT プラットフォームで実行される Java アプリケーションでメモリー・リークが発生しているかどうかを調べる場合、アプリケーション実行中にタスク・マネージャでメモリーの設定を調べればよいと思うかもしれません。しかし、実行中の Java アプリケーションをいくつか調べてみると、ネイティブ・アプリケーションと比較してそれらが使用するメモリーが多いことに気付くことでしょう。私が携わったいくつかの Java プロジェクトの場合は、起動直後に 10 から 20MB のシステム・メモリーが使用されていました。これに対して、オペレーティング・システム付属のネイティブ・プログラム、Windows エクスプローラの場合は、5MB程度です。

Java アプリケーションでのメモリー使用についてもう 1つ注意するべきことは、IBM JDK 1.1.8 JVM で実行される一般的なプログラムの場合、実行時間が長ければ長いほど消費するシステム・メモリーが大きくなるように見えることです。プログラムがメモリーをシステムに戻すことがまったくないまま、やがて大量の物理メモリーがアプリケーションに割り振られることになるように思えます。このような現象は、メモリー・リークが発生しているということを意味しているのでしょうか？

実際に発生していることを理解するには、JVM がヒープ用のシステム・メモリーをどのように使用するかをよく理解する必要があります。`java.exe` を実行するとき、ガーベッジ・コレクションによって回収されるヒープ領域の初期サイズと最大サイズを制御するオプションがあります (それ

ぞれ -ms および -mx)。Sun JDK 1.1.8 では、デフォルトとして 1MB の初期設定値、および 16MB の最大値が使用されています。IBM JDK 1.1.8 では、デフォルトとして、マシンの物理メモリーの合計サイズの半分が使用されています。それらのメモリー設定値は、JVM がメモリー不足になった場合にどうなるかに関して直接的な影響があります。JVM は、ガーベッジ・コレクション・サイクルが完了するまで待つことなくヒープ領域を大きくしている可能性があります。

それで、メモリー・リークを検出して解消するためには、タスク・モニター・ユーティリティーのようなプログラムではなく、それ以上の機能を持つツールが必要になります。メモリー・リークを検出するには、メモリー・デバッグ用のプログラムを使うのが便利です (参考文献を参照)。一般にそのようなプログラムを使えば、ヒープ中のオブジェクトの数、各オブジェクトのインスタンスの数、そしてそれらのオブジェクトで使用されているメモリーに関する情報が得られます。さらに、各オブジェクトの参照や参照元に関する有用な情報を表示する機能が含まれていることもあり、それによってメモリー・リークの発生元を追跡できます。

この後、Sitraka Software 社の JProbe デバッガーを使ってメモリー・リークを検出し解消する方法をご紹介します。これは、デバッガー・ツールを使ってメモリー・リークを解消するための基本的な概念を理解する助けとなるでしょう。

メモリー・リークの例

ここで示す例は、私の属する部門で商用リリースのために開発した Java JDK 1.1.8 アプリケーションです。テスト担当者が何時間も作業した後、問題が明らかになりました。この Java アプリケーションの元になったコードとパッケージは、ある程度の期間にわたって複数のプログラマー・グループによって開発されたものです。このアプリケーションのメモリー・リークは、私の推測では、どこか別のところで開発されたコードをプログラマーがよく理解していなかったことが原因です。

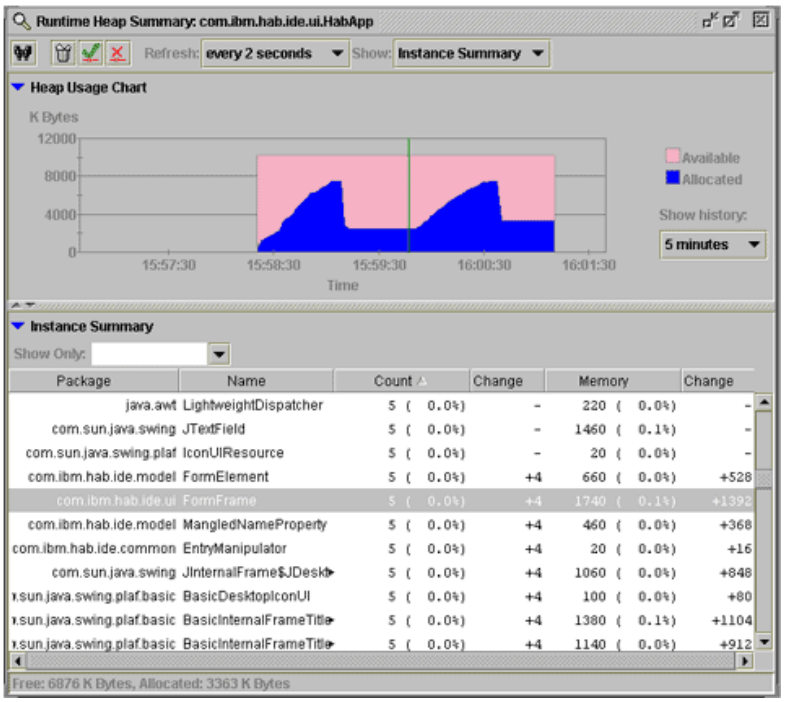
問題の Java コードは、Palm OS のネイティブ・コードを書くことなく、Palm PDA のためのアプリケーションを作成するためのものです。ユーザーは、グラフィカル・インターフェースを使用することによってフォームを作成し、そこにコントロールを配置し、それらのコントロールのイベントを接続することによって、Palm アプリケーションを作成できます。テスト担当者は、この Java アプリケーションを使ってフォームやコントロールを作成しては削除することを繰り返すうちに、ついにはメモリー不足になることを発見しました。開発者のマシンの方が物理メモリーの量が多かったため、開発者はこの問題を検出できませんでした。

この問題を調査するため、私は JProbe を使うことによって、エラーの発生箇所を調べました。JProbe には非常に便利なツールやメモリー・スナップショットの機能がありますが、それにもかかわらずこの作業は、まず特定のメモリー・リークの原因を突き止めてからコードを変更し、結果を確認するという、うんざりするような反復作業になりました。

JProbe には、デバッグ・セッション中に実際に記録する情報を選択するためのオプションがいくつか用意されています。実際に作業をしていくうちに、必要な情報を得るためにはパフォーマンス・データの収集機能をオフにし、ヒープ・データの追跡に集中すると効率が最高になることがわかってきました。JProbe には、Runtime Heap Summary という機能があります。これは、Java アプリケーションの実行時のヒープ・メモリー使用量の時間変化を表示する機能です。また、JVM に対してガーベッジ・コレクション実行を強制するためのツールバー・ボタンも用意されていま

す。この機能は、Java アプリケーションの中で、あるクラスの特定のインスタンスが不要になった時点で、そのインスタンスをガーベッジ・コレクションにより回収するとどうなるかを調べる上で非常に便利でした。ヒープ・ストレージの量の時間変化を図 2 に示します。

図 2. Runtime Heap Summary



「Heap Usage Chart」の中で、青い部分はヒープ・スペースの割り振り量を示しています。Java プログラムの起動後、安定した時点で、ガーベッジ・コレクションを強制実行しました。そのことは、緑の線より左側で青色領域が突然落ち込んでいることによって示されています (緑の線はチェックポイントの挿入を示しています)。次に、フォームを 4個追加してから削除し、その後、再度ガーベッジ・コレクションを実行しました。プログラムは、フォームが 1個だけの初期状態に戻っているにもかかわらず、チェックポイントの後の青色領域のレベルは、チェックポイントの前の青色領域のレベルよりも全体として高くなっています。このことは、メモリー・リークの発生を示唆しています。そこで、「Instance Summary」を見てみると、確かにメモリー・リークが発生していました。チェックポイント後には、`FormFrame` クラス (フォームのメイン UI クラス) のカウントが 4 増加しています。

原因を突き止める

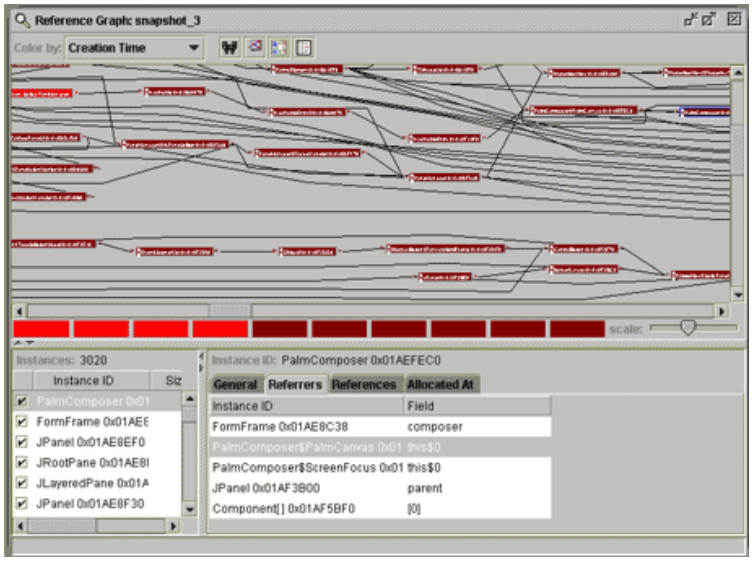
テスト担当者から報告された問題の発生箇所を突き止めるためにまずしたことは、簡単で再現可能なテスト・ケースを作ることでした。この例の場合、1個のフォームを追加した後、そのフォームを削除し、それからガーベッジ・コレクションを強制実行すると、削除されたフォームに伴うたくさんのクラス・インスタンスが存在し続けるということがわかりました。この問題は、JProbe Instance Summary ビューを見ると明らかです。そこでは、各 Java クラスのヒープに存在するインスタンスの数が表示されます。

ガーベッジ・コレクターの回収を妨げている参照を特定するため、JProbe の「Reference Graph」を使用して、削除された`FormFrame` クラスをまだ参照しているのはどのクラスかを調べました (図

3)。このプロセスは、この問題をデバッグする上で最も面倒な処理の 1つでした。というのは、使用されていないオブジェクトをたくさんの異なるオブジェクトが引き続き参照していたからです。それらの参照元のうちのどれが問題の原因であるかを試行錯誤によって突き止めることは、非常に時間のかかる作業でした。

この場合、ルート・クラス (左上の赤い部分) が問題の発生元になっています。オリジナルの`FormFrame` クラスから線をたどると、右の方の青色で示したクラスになります。

図 3. 参照グラフでメモリー・リークをトレースする



この例の場合、主な犯人は、静的ハッシュ・テーブルを含むフォント・マネージャー・クラスでした。参照元のリストをトレースすることにより、ルート・ノードは、それぞれのフォームで使用されるフォントを格納するための静的ハッシュ・テーブルであることがわかりました。さまざまなフォームを独立してズームインまたはズームアウトできるようにするため、そのハッシュ・テーブルには、特定のフォームのためのすべてのフォントから構成されるベクトルが含まれるようにしていました。フォームのズーム・ビューが変更されると、フォント・ベクトルが取り出され、それぞれのフォント・サイズにズーム係数を乗算することになっていました。

このフォント・マネージャー・クラスの問題点は、フォーム作成時にフォント・ベクトルがハッシュ・テーブルに入れられるものの、フォームが削除される時点でそのベクトルを削除することが考慮されていない、ということでした。したがって、この静的ハッシュ・テーブルは、実際にはアプリケーション自体の存続期間にわたって存在し、各フォームを参照するキーを削除することが決してありませんでした。その結果、フォームとそれに対応するクラスのすべてがメモリー内に存在し続けたのです。

修正の適用

この問題を解決する簡単な方法は、フォント・マネージャー・クラスにメソッドを 1つ追加し、ユーザーがフォームを削除した時点で、そのメソッドから、ハッシュ・テーブルの `remove()` メソッドを、該当するキーを指定して呼び出すようにすることです。その`removeKeyFromHashtables()` メソッドは、下記のとおりです。

```
public void removeKeyFromHashtables(GraphCanvas graph) {
  if (graph != null) {
    viewFontTable.remove(graph);     // remove key from hashtable
                                     // to prevent memory leak
  }
}
```

次に、このメソッドの呼び出しを`FormFrame`クラスに追加します。`FormFrame`では、実際にはSwingの内部フレームを使用することによってフォームUIが実装されています。それで、内部フレームが完全にクローズされた時点で実行されるメソッドの中に、フォント・マネージャーに追加したメソッドの呼び出しを追加しました。

```
/**
* Invoked when a FormFrame is disposed. Clean out references to prevent
* memory leaks.
*/
public void internalFrameClosed(InternalFrameEvent e) {
  FontManager.get().removeKeyFromHashtables(canvas);
  canvas = null;
  setDesktopIcon(null);
}
```

以上の変更をコードに加えた後、デバッガーを使って同じテスト・ケースを実行することにより、削除されたフォームに対応するオブジェクト・カウントが下がったことを確認しました。

メモリー・リークの予防

メモリー・リークは、いくつかの一般的な問題に注意することにより防ぐことができます。しばしばメモリー・リークの原因となるのは、ハッシュ・テーブルやベクトルなどのコレクション・クラスです。そのクラスが`static`と宣言されていて、アプリケーションの存続期間全体にわたって存在する場合、その可能性が特に大きくなります。

よくある別の問題は、あるクラスをイベント・リスナーとして登録しておきながら、そのクラスがもう使用されなくなった時点で登録削除するのを忘れている場合です。また、クラスのメンバー変数が別のクラスを指す場合、適当なタイミングでその変数をヌルにセットする必要があります。

結論

メモリー・リークの原因を検出するには特別なデバッグ・ツールが必要であり、その作業は、そのようなツールを使用したとしてもうんざりさせられるようなプロセスになることがあります。しかし、そのツールを使いこなせるようになり、オブジェクト参照をトレースする場合にどんなパターンを探せばいいかがわかってくると、メモリー・リークを追跡できるようになることでしょう。さらに、プログラミング・プロジェクトを救うかもしれない貴重なスキルを身に付けることができます。それだけでなく、将来のプロジェクトでメモリー・リークを防ぐためにどんなコーディング・スタイルが求められるかに関する洞察も得ることができます。

著者について

Jim Patrick

Jim Patrick は、IBM のパーベイシブ・コンピューティング部の顧問プログラマーです。1996年以来、Java のプログラミングに携わっています。