Javaの理論と実践: 弱参照でメモリー・リークを塞ぐ

弱参照によってオブジェクト・ライフサイクルの関係表現が容易に

Brian Goetz
Principal Consultant
Quiotix

2005年 11月 22日

Java™ 言語でのプログラムは、理論的に「メモリー・リーク」とは無縁なはずです。ところが、オブジェクトが既にプログラムの論理状態の一部でなくなっているにも関わらず、ガーベジ・コレクションで処理されない場合があります。今回は、清掃技術者であるBrian Goetzが、意図せずにオブジェクトが残ってしまう場合についての一般的な原因を究明し、弱参照（weak reference）によって漏れ（leak）を防ぐ方法を解説します。

このシリーズの他の記事を見る

プログラムがもはや使用していないオブジェクトをGC（ガーベジ・コレクション）によって再利用する場合、オブジェクトの『論理的』ライフタイム（アプリケーションがそのオブジェクトを使用する時間）と、そのオブジェクトに対して存在している参照の『実際の』ライフタイムは、同じである必要があります。ほとんどの場合、これは適切なソフトウェア・エンジニアリング手法によって自動的に行われ、私達がオブジェクトのライフタイムの問題に注意を払う必要はほとんどありません。しかし場合によると私達は、想定以上に長くメモリー中にオブジェクトを保持するような参照を作ってしまうものです。こうした状況は、非意図的オブジェクト保持（unintentional object retention）と呼ばれます。

グローバル・マップによるメモリー・リーク

非意図的オブジェクト保持の原因として最も一般的なものは、メタデータと一時的オブジェクト（transient object）との関連付けにMapを使う場合です。例えば、中間的なライフタイムを持つオブジェクトがあるとしましょう。つまり、そのオブジェクトを割り当てたメソッド・コールよりも長いライフタイムを持ち、アプリケーションよりも短いライフタイムを持つオブジェクト、例えばクライアントからのソケット接続などです。そして、このソケットに何らかのメタデータ、例えば接続を行ったユーザーのIDを関連付けたいとします。Socketが作られた時には、この情報が分からず、また、あなたはSocketクラスやそのインスタンス化を制御していないので、Socketオブジェクトにデータを追加することができません。この場合の典型的な手法としては、そうした情報を、リスト1のSocketManagerクラスに示すようなグローバルMapに保存します。

リスト1.グローバルMapを使ってメタデータとオブジェクトを関連付ける

```
public class SocketManager {
    private Map<Socket,User> m = new HashMap<Socket,User>();

    public void setUser(Socket s, User u) {
        m.put(s, u);
    }
    public User getUser(Socket s) {
        return m.get(s);
    }
    public void removeUser(Socket s) {
        m.remove(s);
    }
}

SocketManager socketManager;
...
socketManager.setUser(socket, user);
```

この手法の問題点は、メタデータのライフタイムとソケットのライフタイムを結びつける必要がある、という点です。しかしあなたが、プログラムがソケットを必要としなくなるのはいつかを厳密に知り、対応するマッピングをMapから削除することを忘れないようにしないと、リクエストが処理されソケットが閉じられた後も、SocketオブジェクトとUserオブジェクトはMapの中に永遠に残ってしまうのです。そうすると、アプリケーションは二度とそれらを使わないにもかかわらず、SocketとUserオブジェクトはガーベジ・コレクションされません。これをチェックせずに放置しておくと、プログラムを長期間実行しているうちにメモリーが足りなくなってしまいます。プログラムがSocketを使わなくなるのはいつかを識別するための手法は、ごく些細な場合を除いて、手動によるメモリー管理に似ており、面倒で間違いを起こしやすいものになります。

メモリー・リークを検出する

通常、プログラムにメモリー・リークがある場合の最初の兆候としては、OutOfMemoryErrorが投げられること、あるいは、頻繁にガーベジ・コレクションが行われるためパフォーマンスが低下し始めることなどがあります。幸いガーベジ・コレクターは、メモリー・リークの検出に使用可能な大量の情報を提供してくれます。例えば -verbose:gcや-Xloggcというオプションを使ってJVMを呼び出すと、GCが実行される度に、コンソール、あるいはログ・ファイルに診断メッセージが出力されるのです。出力される情報としては、どのくらいGCに時間がかかったか、現在のヒープ使用状況、どのくらいの量のメモリーが回復されたか、などがあります。GCの使用状況をロギングしても邪魔にはなりません。ですから実稼働では、メモリー問題の分析やGCの調整が必要な場合に備えて、デフォルトでGCロギングをイネーブルにしておくのは妥当なことです。

ツールを利用すると、GCログ出力をグラフィカルに表示することができます。そうしたツールの一つが、無料のJTune（参考文献）です。GC後のヒープ・サイズのグラフを見ることによって、そのプログラムのメモリー使用状況の傾向が分かります。ほとんどのプログラムでは、メモリー使用を2つのコンポーネント、つまり『ベースライン』使用と『現在の負荷』による使用というコンポーネントに分割することができます。サーバー・アプリケーションの場合、ベースライン使用というのは、そのアプリケーションがどの負荷にも割り当てられておらず、リクエストを受け付け可能という状態で使用しているメモリーを言います。現在の負荷による使用というのは、リクエストの処理プロセスで使用され、そのリクエスト処理が完了すると解放されるメモリーを言います。負荷がほぼ一定である限り、アプリケーションのメモリー使用状況は、かなり速く定常に達するのが普通です。アプリケーションが初期化を完了し、負荷が増加していないにもかかわら

ずメモリー使用が上昇傾向を続けるのであれば、恐らくプログラムは、以前のリクエストを処理する中で生成されたオブジェクトを保持しているのです。

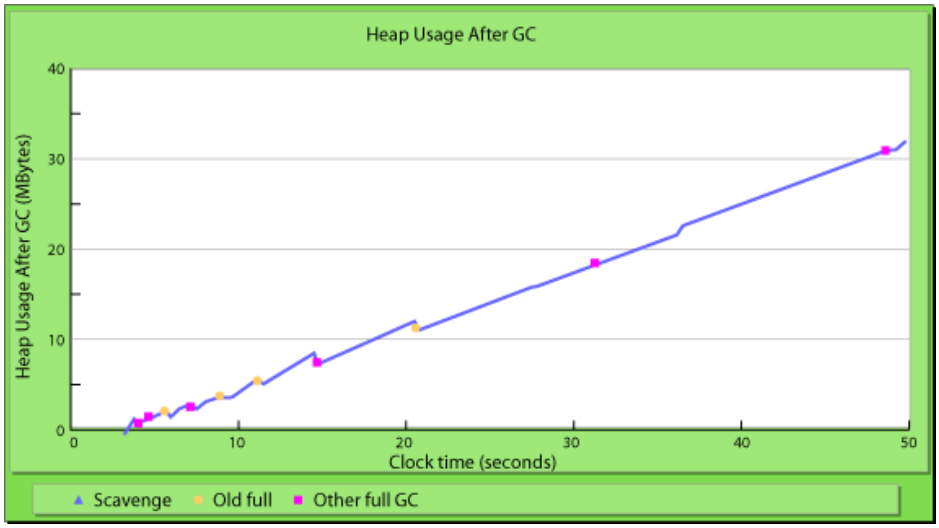
リスト2は、メモリー・リークがあるプログラムを示しています。MapLeakerは、スレッド・プールにあるタスクを処理し、各タスクの状態をMapの中に記録します。残念なことに、MapLeakerはタスクを終了してもエントリーを削除しません。そのためステータス・エントリーとタスク・オブジェクトは、（その内部状態と共に）永遠に累積し続けます。

リスト2.Mapベースのメモリー・リークがあるプログラム

```
public class MapLeaker {
    public ExecutorService exec = Executors.newFixedThreadPool(5);
    public Map<Task, TaskStatus> taskStatus
        = Collections.synchronizedMap(new HashMap<Task, TaskStatus>());
    private Random random = new Random();

    private enum TaskStatus { NOT_STARTED, STARTED, FINISHED };

    private class Task implements Runnable {
        private int[] numbers = new int[random.nextInt(200)];

        public void run() {
            int[] temp = new int[random.nextInt(10000)];
            taskStatus.put(this, TaskStatus.STARTED);
            doSomeWork();
            taskStatus.put(this, TaskStatus.FINISHED);
        }
    }

    public Task newTask() {
        Task t = new Task();
        taskStatus.put(t, TaskStatus.NOT_STARTED);
        exec.execute(t);
        return t;
    }
}
```

図1は、MapLeaker でGCが行われた後の、アプリケーションのヒープ・サイズのグラフを示しています。この上昇曲線は、正にメモリー・リークの証と言うことができます。（実際のアプリケーションでは、これほど劇的な傾斜になることはありません。しかし充分長期間GCデータを集めれば、明らかに分かるものです。）

図1.メモリー使用が上昇を続けている



メモリー・リークがあるに違いない、と分かったら、次のステップは、どんなタイプのオブジェクトが問題を引き起こしているのかを見つけることです。メモリー・プロファイラーであれば、オブジェクト・クラス毎に整理されたヒープのスナップショットを生成することができます。ヒープ・プロファイル用の商用ツールには幾つか素晴らしいものがありますが、メモリー・リークの検出に費用をかける必要はありません。既に組み込まれているhprofツールでも用が足りるのです。hprofを使ってメモリー使用を追跡するには、-Xrunhprof:heap=sitesオプションを使ってJVMを呼び出します。

リスト3は、アプリケーションのメモリー使用を整理したhprof出力のうちの、関係のある部分を示しています。（hprofツールは、アプリケーションが終了した後、あるいはアプリケーションにkill -3という信号を送った場合、あるいはWindowsでCtrl+Breakを押した場合での使用状況を分類整理します）。2つのスナップショットの間で、Map. EntryオブジェクトとTaskオブジェクト、そしてint[]オブジェクトによる使用が大きく伸びていることに注意してください。

リスト3.Map.EntryオブジェクトとTaskオブジェクトの成長を示す HPROF出力

```
SITES BEGIN (ordered by live bytes) Fri Oct 28 16:30:48 2005
          percent          live          alloc'ed  stack class
 rank   self  accum     bytes objs     bytes  objs trace name
    1 70.13% 70.13%   5694888 13909   5694888 13909 300305 int[]
    2 18.27% 88.40%   1483976    68 278273632 13908 300321 int[]
    3  4.11% 92.51%    333816 13909    333816 13909 300310 java.util.HashMap$Entry
    4  2.74% 95.25%    222544 13909    222544 13909 300304 com.quiotix.dummy.MapLeaker$Task
    5  2.42% 97.67%    196640     2    262192     11 300325 java.util.HashMap$Entry[]
    6  0.66% 98.33%     53680 3355    222464 13904 300324 java.util.concurrent.LinkedBlockingQueue$Node
SITES END

SITES BEGIN (ordered by live bytes) Fri Oct 28 16:31:32 2005
          percent          live          alloc'ed  stack class
 rank   self  accum     bytes objs     bytes  objs trace name
    1 77.07% 77.07%  41176024 100020  41176024 100020 301069 int[]
    2 12.98% 90.05%   6933768  359 2001885688 100020 301093 int[]
    3  4.49% 94.55%   2400480 100020   2400480 100020 301082 java.util.HashMap$Entry
    4  3.00% 97.54%   1600320 100020   1600320 100020 301068 com.quiotix.dummy.MapLeaker$Task
    5  1.96% 99.50%   1048592     1   2097248     14 301104 java.util.HashMap$Entry[]
    6  0.05% 99.55%     25936 1621   1600240 100015 301101 java.util.concurrent.LinkedBlockingQueue$Node
SITES END
```

リスト4はhprof出力のうち、また別の部分であり、Map. Entryオブジェクトへの割り当て場所に関するコール・スタック情報を示しています。この出力は、どのコール・チェーンがMap.Entryオブジェクトを生成しているかを示しています。通常は、ちょっとしたプログラム解析を行えば、メモリー・リークの原因をピンポイントで突き止めるのは容易なものです。

リスト4.Map.Entryオブジェクトへの割り当て場所を示すHPROF出力

```
TRACE 300446:
 java.util.HashMap$Entry.<init>(<Unknown Source>:Unknown line)
 java.util.HashMap.addEntry(<Unknown Source>:Unknown line)
 java.util.HashMap.put(<Unknown Source>:Unknown line)
 java.util.Collections$SynchronizedMap.put(<Unknown Source>:Unknown line)
 com.quiotix.dummy.MapLeaker.newTask(MapLeaker.java:48)
 com.quiotix.dummy.MapLeaker.main(MapLeaker.java:64)
```

弱参照（weak reference）が救いに

SocketManagerの問題は、Socket-Userマッピングのライフタイムと Socketのライフタイムとを一致させる必要があることですが、言語には、そのルールを強制するための単純な方法がありません。そのためにプログラムは、手動によるメモリー管理に似た手法に頼らざるを得ません。幸いJDK 1.2の時点でのガーベジ・コレクターには、こうした、オブジェクトのライフサイクル依存関係を宣言するための方法が用意されています。ですから、この種のメモリー・リークをガーベジ・コレクターで防ぐことができるのです。そのためには、『弱参照』を利用します。

弱参照は、オブジェクト参照（referentと呼ばれます）に対するホルダーです。弱参照を利用すると、referentがガーベジ・コレクションされるのを妨げることなく、referentに対する参照を維持することができます。ガーベジ・コレクターがヒープをトレースする場合、あるオブジェクトに対して存在している参照が弱参照のみである場合には、存在している参照が全く無いかのように、referentがGCの候補となります。そして、それまで存在している弱参照は、全てクリアーされます（弱参照のみによって参照されているオブジェクトは、弱い到達可能（weakly reachable）と言われます）。

弱参照のreferentは構成時に設定され、その値がクリアーされていない場合には、get() を使って値を取得することができます。もし弱参照がクリアーされてしまっている場合（referentが既にガーベジ・コレクションされてしまった、あるいは、誰かがWeakReference.clear() を呼んだ場合）には、get() はnullを返します。従ってget() の結果を使う前には、返された値がnullではないことを必ずチェックする必要があります（referentは、やがてガーベジ・コレクションされるはずだからです）。

通常の参照（強参照、strong reference）を使ってオブジェクト参照をコピーする場合には、referentのライフタイムは、最低限でもコピーされる参照のライフタイムと同じだけの長さを持つようにします。よく注意しないと、これはプログラムのライフタイムと同じになってしまう可能性があります（例えばオブジェクトをグローバルコレクションの中に置くような場合）。一方、あるオブジェクトに対して弱参照を作成する場合には、referentのライフタイムは全く拡張されません。『オブジェクトがまだ生きている間』に、そのオブジェクトに到達するための別の方法を、単純に維持するだけです。

弱参照が最も便利なのは、弱いコレクション（weak collection）を構成する場合、例えば、アプリケーションの他の部分がオブジェクトを使っている間だけ、こうしたオブジェクトに関す

るメタデータを保存するような、弱いコレクションを構成する場合です。これは実は、正にSocketManagerクラスが行うべきことなのです。この使い方は、弱参照の使い方として非常に一般的なため、鍵に対して弱参照を使う（値に対しては使いません）WeakHashMapも、JDK 1.2のクラス・ライブラリーに追加されています。通常のHashMapでオブジェクトを鍵として使う場合には、Mapからマッピングが削除されるまで、そのオブジェクトはコレクションされません。WeakHashMapを利用すると、オブジェクトがガーベジ・コレクションされるのを妨げることなく、そのオブジェクトをMapキーとして使えるのです。リスト5は弱参照の使い方の例として、WeakHashMapからのget() メソッドの実装例を示しています。

リスト5.WeakReference.get() の実装例

```
public class WeakHashMap<K,V> implements Map<K,V> {

    private static class Entry<K,V> extends WeakReference<K>
      implements Map.Entry<K,V> {
        private V value;
        private final int hash;
        private Entry<K,V> next;
        ...
    }

    public V get(Object key) {
        int hash = getHash(key);
        Entry<K,V> e = getChain(hash);
        while (e != null) {
            K eKey= e.get();
            if (e.hash == hash && (key == eKey || key.equals(eKey)))
                return e.value;
            e = e.next;
        }
        return null;
    }
```

WeakReference.get() は呼ばれると、（もしreferentがまだ生きていれば）referentに対する強参照を返します。つまり強参照が、マッピングがガーベジ・コレクションされないように防いでくれるので、whileループのボディーの中でマッピングが消えてしまうことを心配する必要はありません。WeakHashMapの実装は、弱参照を持つ一般的なイディオム、つまり、ある内部オブジェクトがWeakReferenceを拡張するというイディオムを説明しています。この理由は、次のセクションで参照キューについて解説する時に明らかになります。

WeakHashMapにマッピングを追加する際には、（鍵はガーベジ・コレクションされるため）後でマッピングが「無くなってしまう」ことを忘れないでください。その結果get() はnullを返します。ですから、get() が返す値がnullかどうかをチェックすることが、通常以上に重要になってきます。

WeakHashMapでリークを塞ぐ

SocketManagerでのリークを修正するのは簡単です。リスト6に示すように、単純にHashMapをWeakHashMapで置き換えるだけです。（もしSocketManagerがスレッド・セーフでなければならない場合には、WeakHashMapをCollections.synchronizedMap() でラップします。）マッピングのライフタイムと鍵のライフタイムを結びつける必要がある場合には、いつでもこの手法を使うことができます。ただし、この手法を使いすぎないように注意する必要があります。ほとんどの場合、使うべき適切なMap実装は、通常のHashMapなのです。

リスト6.SocketManagerをWeakHashMapで修正する

```
public class SocketManager {
    private Map<Socket,User> m = new WeakHashMap<Socket,User>();

    public void setUser(Socket s, User u) {
        m.put(s, u);
    }
    public User getUser(Socket s) {
        return m.get(s);
    }
}
```

参照キュー（ reference queue ）

WeakHashMapは、マップ・キーの保持に弱参照を使います。これによって、アプリケーションが鍵オブジェクトを使わなくなれば、鍵オブジェクトはガーベジ・コレクションされ、get() 実装はWeakReference.get() がnullを返すかどうかを判断して、生きているマッピングと死んだマッピングを区別することができます。しかし、Mapによるメモリー消費がアプリケーションのライフタイム中ずっと増加し続けるのを防ぐためには、これは半分でしかありません。鍵オブジェクトがコレクションされた後、死んだエントリーをMapから除去するためには、何かをする必要があります。そうしないと、Mapは単純に、死んだ鍵に対応するエントリーで一杯になってしまいます。これはアプリケーションからは見えませんが、鍵はガーベジ・コレクションされてもMap.Entryと値オブジェクトはコレクションされないので、やはりアプリケーションのメモリー不足を引き起こすのです。

死んだマッピングを無くすには、周期的にMapをスキャンし、各弱参照に対してget() を呼び、もしget() がnullを返したらマッピングを削除します。しかし、この方法は、生きているエントリーがMapの中に数多くある場合には、効率が良くありません。もし、弱参照のreferentがガーベジ・コレクションされた時に通知してくれるような方法があれば、素晴らしいことです。参照キューは、正にそのためにあるのです。

ガーベジ・コレクターが、オブジェクトのライフサイクルに関する情報をアプリケーションにフィードバックする上で、参照キューは基本的な手段です。弱参照には2つのコンストラクターがあります。1つはreferentだけを引数とし、もう1つは参照キューも引数に取ります。関連の参照キューを持った弱参照が作られ、referentがGCの候補になると、（ referentではなく ）参照オブジェクトは参照がクリアーされた後、参照キューの待ち行列に入れられます。そうするとアプリケーションは参照キューから参照を取り出すことができ、 referentがコレクションされたことを知るので、関連のクリーンアップ活動（ 弱いコレクションから外れてしまったオブジェクトに対するエントリーを消し去るなど ）を行うことができます。（ 参照キューは、BlockingQueueと同じデキュー・モード（ dequeuing mode ）、つまりポーリングや時刻指定ブロッキング、時間指定無しブロッキングなどのモードを提供しています。 ）

WeakHashMapには、expungeStaleEntries() と呼ばれるプライベート・メソッドがあります。これはほとんどのMapオペレーション中に呼ばれ、有効期限の切れた参照に対する参照キューをポーリングし、それに関連したマッピングを削除します。expungeStaleEntries() の実装例をリスト7に示します。鍵と値のマッピングの保存に使用されているEntryタイプは、WeakReferenceを拡張します。ですからexpungeStaleEntries() が、次の、期限の切れた弱参照を要求すると、Entryを返します。Mapのクリーンアップには、周期的にMapの内容を調べる代わりに参照キューを使った方が

効率的です。これは、参照キューによるクリーンアップ過程では生きているエントリーに触れることはなく、参照キューは実際に待ち行列に入った参照がある場合しか動作しないためです。

リスト7.WeakHashMap.expungeStaleEntries() の実装の例

```
private void expungeStaleEntries() {
 Entry<K,V> e;
        while ( (e = (Entry<K,V>) queue.poll()) != null) {
            int hash = e.hash;

            Entry<K,V> prev = getChain(hash);
            Entry<K,V> cur = prev;
            while (cur != null) {
                Entry<K,V> next = cur.next;
                if (cur == e) {
                    if (prev == e)
                        setChain(hash, next);
                    else
                        prev.next = next;
                    break;
                }
                prev = cur;
                cur = next;
            }
        }
    }
```

まとめ

弱参照と弱いコレクションは、ヒープ管理のための強力なツールであり、これらによってアプリケーションは、通常の参照（強参照）による「完全到達可能、あるいは全く到達不能」という到達機能ではなく、より洗練された概念による到達機能を持てるようになります。次回は、弱参照と関連した、ソフト参照（soft reference）を調べます。そして、弱参照やソフト参照がある場合にガーベジ・コレクターがどのように振る舞うかを検証して行きます。

著者について

Brian Goetz

Brian Goetz は18 年間以上に渡って、専門的ソフトウェア開発者として働いています。彼はカリフォルニア州ロスアルトスにあるソフトウェア開発コンサルティング会社、Quiotixの主席コンサルタントであり、またいくつかのJCP Expert Groupの一員でもあります。2005年の末にはAddison-Wesleyから、Brianによる著、Java Concurrency In Practiceが出版される予定です。Brian著による有力業界紙に掲載済みおよび掲載予定の記事のリストを参照してください。