Javaの理論と実践: ソフト参照でメモリー・リークを塞ぐ

ソフト参照で安直なキャッシュが可能に

Brian Goetz
Principal Consultant
Quiotix

2006年 1月 24日

『Javaの理論と実践』の前回の記事では、Java™の清掃技術者であるBrian Goetzが、『弱参照（weak reference）』を検証しました。弱参照を利用すると、ガーベジ・コレクターに対して、「あるオブジェクトがガーベジ・コレクションされるのは妨害しないが、そのオブジェクトへの参照は維持したい」という意図を伝えることができます。今回の記事では、Referenceオブジェクトの、もう一つの形式、『ソフト参照（soft reference）』を検証します。ソフト参照も、ガーベジ・コレクターの助けを借りてメモリーの使用管理を行い、また潜在的なメモリー・リークを防止するのです。

このシリーズの他の記事を見る

ガーベジ・コレクションによって、Javaプログラムはメモリー・リークを回避できる可能性があります。少なくとも、「メモリー・リーク」の定義を非常に限定的なものにすれば、そう言うことができるでしょう。しかし、だからと言って、Javaプログラムにおけるオブジェクト・ライフタイムの問題は完全に無視できる、ということにはなりません。通常、Javaプログラムにおけるメモリー・リークは、オブジェクト・ライフタイムに充分注意しなかった場合、あるいはオブジェクト・ライフサイクル管理のための標準機構を無視したような場合に起こります。例えば前回の記事では、オブジェクトのライフサイクルを区切ることを忘れると、メタデータを一時的オブジェクト（transient object）と関連付けようとする際に、非意図的オブジェクト保持（unintentional object retention）を引き起こす可能性があることを示しました。その他にも、それらと同じようにオブジェクト・ライフサイクルの管理を無視したり、壊したりする可能性のあるイディオムがあり、それらによってメモリー・リークが起こり得るのです。

オブジェクト・ロイタリング（loitering）

リスト1のLeakyChecksumクラスは、ある種のメモリー・リークを説明しています。このリークは、オブジェクト・ロイタリング（loitering、「うろつく」の意）と呼ばれることがあります。このクラスは、あるファイルの内容のチェックサムを計算するための、getFileChecksum() メソッドを提供しています。getFileChecksum() メソッドは、そのファイルの内容を読み取ってバッファーにいれ、チェックサムを計算します。もっと直接的な実装であれば、単純にバッファーを、getFileChecksum() の中のローカル変数としてアロケートするところですが、このバージョン

はもっと「賢く」、インスタンス・フィールドにバッファーをキャッシュすることによってメモリー・チャーン（memory churn）を減少させています。しかし多くの場合、この「最適化」では、期待したような効果が得られません。つまりオブジェクト・アロケーションの方が、多くの人が考えているよりも安くすむのです。（またバッファーをローカル変数からインスタンス変数に格上げすると、同期化を追加しない限り、そのクラスはスレッド・セーフでなくなることにも注意してください。単純な実装では、getFileChecksum()をsynchronized として宣言する必要がなく、並行呼び出しの場合のスケーラビリティーが、より高くなります。）

リスト1.「オブジェクト・ロイタリング」を示すクラス

```
// BAD CODE - DO NOT EMULATE
public class LeakyChecksum {
    private byte[] byteArray;

    public synchronized int getFileChecksum(String fileName) {
        int len = getFileSize(fileName);
        if (byteArray == null || byteArray.length < len)
            byteArray = new byte[len];
        readFileContents(fileName, byteArray);
        // calculate checksum and return it
    }
}
```

このクラスには多くの問題がありますが、ここではメモリー・リークの問題に焦点を絞りましょう。バッファーをキャッシュしようと決めた理由は、恐らく「プログラム内から何度も呼ばれるに違いない。従って、再アロケーションするよりもバッファーを再利用した方が効率的」という想定からでしょう。しかし結果として、バッファーは常にプログラムから到達可能なため、（LeakyChecksumオブジェクトがガーベジ・コレクションされない限り）解放されなくなってしまいます。さらに悪いことに、このバッファーは大きくなる可能性はありますが、小さくなる可能性はありません。従ってLeakyChecksumはバッファーの大きさを、処理されたファイルの最大サイズのまま保持してしまうのです。少なくとも、これによってガーベジ・コレクターに圧力が加わり、より頻繁にガーベジ・コレクションが要求されることになります。つまり、将来のチェックサムを計算するために大きなバッファーを用意しておくことは、利用可能なメモリーを効率的に利用する上では不適切かも知れないのです。

LeakyChecksumでの問題の原因は、バッファーは論理的にgetFileChecksum() オペレーションにローカルであるにもかかわらず、インスタンス・フィールドに格上げされたためにライフサイクルが人為的に伸ばされてしまったことにあります。その結果、バッファーのライフサイクル管理はJVMに任されず、このクラス自体が行う羽目になるのです。

ソフト参照（soft reference）

前回の記事では、アプリケーションがオブジェクトに到達するための、もう一つの方法としての弱参照を見ました。つまりアプリケーションは弱参照を使用することによって、そのオブジェクトがプログラムで使用中であっても、オブジェクトのライフタイムを引き伸ばすことなく、そのオブジェクトに到達できるのです。Referenceの、もう一つのサブクラスであるソフト参照は、弱参照とは異なりますが、似た用途に使うことができます。弱参照を使うと、アプリケーションはガーベジ・コレクションと干渉しない参照を作成できますが、ソフト参照を使うと、アプリケーションは一部のオブジェクトを「消耗可（expendable）」と指定することによって、ガーベジ・コレクターの助けを借りられるのです。ガーベジ・コレクターは、アプリケーションがどのメモ

リーを使用し、どのメモリーを使用していないかを適切に判断できますが、利用可能なメモリーをどのように効率的に利用するかという判断は、アプリケーションに任されています。どのオブジェクトを保持するかに関して、アプリケーションが貧弱な判断しかできない場合には、アプリケーション用のメモリーが不足しないようにガーベジ・コレクターが懸命に動作するため、パフォーマンスは低下します。

パフォーマンスの最適化手段として、キャッシュは一般的なものです。つまりキャッシュによって、アプリケーションは以前行われた計算結果を再利用でき、再計算の必要が無くなります。キャッシュはCPU利用効率とメモリー利用とのトレードオフであり、利用可能なメモリー量によって、このトレードオフの具合が変わります。キャッシュ動作が少なすぎると、望むようなパフォーマンスは得られません。多すぎると、キャッシュ動作のためにメモリーが浪費され、他の目的に利用できるメモリー量が不足するため、パフォーマンスは低下します。一般的に言って、ガーベジ・コレクターはアプリケーションよりも、メモリー要求を的確に判断できます。従って、こうした判断を下すに際して、ガーベジ・コレクターに助けを求めることには意味があります。ソフト参照は、正にそのためにあるのです。

あるオブジェクトに対して残っている参照が、弱参照あるいはソフト参照のみである場合、そのオブジェクトは『ソフト到達可能（softly reachable）』と言われます。ガーベジ・コレクターは、弱到達可能（weakly reachable）オブジェクトを収集するほど積極的には、ソフト到達可能オブジェクトを収集しません。ガーベジ・コレクターは、本当にメモリーを「必要とする」場合のみ、ソフト到達可能オブジェクトを収集するのです。ソフト参照は、ガーベジ・コレクターに対して、「メモリーが非常に不足しているのでない限り、私はこのオブジェクトを保持しておきたい。しかしメモリーが非常に不足しているのであれば、どうぞ収集してください。私はそれに応じて何とかします」と言うための方法なのです。ガーベジ・コレクターは、OutOfMemoryErrorエラーを投げる前には、すべてのソフト参照をクリアーする必要があります。

LeakyChecksumの問題は、キャッシュされたバッファーの管理にソフト参照を使うことによって修正することができます。これをリスト2に示します。こうすることによって、メモリーがどうしても必要にならない限りバッファーは保持されますが、必要な場合には、ガーベジ・コレクターによって再利用することができます。

リスト2. ソフト参照でLeakyChecksumを修正する

```
public class CachingChecksum {
    private SoftReference<byte[]> bufferRef;

    public synchronized int getFileChecksum(String fileName) {
        int len = getFileSize(fileName);
        byte[] byteArray = bufferRef.get();
        if (byteArray == null || byteArray.length < len) {
            byteArray = new byte[len];
            bufferRef.set(byteArray);
        }
        readFileContents(fileName, byteArray);
        // calculate checksum and return it
    }
}
```

安直に利用できるキャッシュ

CachingChecksumは、1つのオブジェクトをキャッシュするためにソフト参照を使っており、そのオブジェクトをキャッシュからクリアーする詳細に関しては、JVMに任せています。これと同じように、GUIアプリケーションでビットマップ・グラフィックスをキャッシュする場合には、ソフト参照もよく使われます。ソフト参照が使えるかどうかは、キャッシュされたデータの喪失からアプリケーションが回復できるかどうか、に大きく依存します。

1つ以上のオブジェクトをキャッシュする必要がある場合にはMapが使えますが、ソフト参照を使う、という方法もあります。つまり、キャッシュをMap<K, SoftReference<V>>として管理する、あるいはSoftReference<Map<K,V>>として管理する、という選択肢があるのです。通常は、後者の選択肢が適切です。なぜなら、ガーベジ・コレクターの作業が少なくて済み、メモリー要求が厳しくなった時には、より少ない努力でキャッシュ全体を再利用できるためです。キャッシュ構築のために、ソフト参照ではなく弱参照が誤って使われてしまう場合がありますが、そうするとキャッシュのパフォーマンスが低下します。現実的には、そのオブジェクトが弱到達可能になると、マイナーなガーベジ・コレクションが頻繁に実行されるため、弱参照は非常に高速に（通常はキャッシュされたオブジェクトが再度必要になる前に）クリアーされます。

パフォーマンスの目的からキャッシュ動作に大きく依存するアプリケーションでは、ソフト参照では道具として弱すぎるかも知れません。少なくとも、柔軟な期限処理(flexible expiration)や複製、トランザクショナル・キャッシュなどを備えた高度なキャッシュ・フレームワークの代用になり得るものではありません。しかし、「安直な」キャッシュ機構としては、非常に魅力的なコスト・パフォーマンスを誇っています。

弱参照と同様、ソフト参照は関連の参照キューを使って作成することができ、その参照は、ガーベジ・コレクターによってクリアーされると、待ち行列に入れられます。参照キューは、ソフト参照の場合は弱参照の場合ほど便利ではありませんが、アプリケーションのメモリーが不足し始めている、という管理警報（management alert）を出すために使うことができます。

ガーベジ・コレクターはReferenceをどう処理するか

弱参照もソフト参照も、抽象Referenceクラスを拡張します（また『幽霊参照（phantom reference）』も同じです。これについては今後の記事で取り上げる予定です）。参照オブジェクトは、ガーベジ・コレクターによって特別に扱われます。ガーベジ・コレクターがヒープのトレース中にReferenceに突き当たると、参照対象オブジェクトをマーキングしたりトレースしたりせず、Referenceを、既知のライブReferenceオブジェクトから成るキューに置きます。ガーベジ・コレクターはトレースが終わると、強参照（strong reference）の対象ではなく弱参照の対象となっているような、ソフト到達可能オブジェクトを特定します。そして、現在のガーベジ・コレクションによって取り戻されるメモリー量と、その他のポリシー上の考慮に基づいて、ソフト参照を今回クリアーすべきかどうかを判断します。クリアーされるべきソフト参照が、対応する参照キューを持っている場合には、そのソフト参照は待ち行列に入れられます。そうすると、残りのソフト到達可能オブジェクト（クリアーされないオブジェクト）は、ルート・セットとして扱われ、こうした新しいルートを使ってヒープ・トレースが継続されます。そして、ライブ・ソフト参照によって到達可能なオブジェクトはマーキングされます。

ソフト参照が処理されると、一連の弱到達可能オブジェクト（強参照の対象にもソフト参照の対象にもなっていないもの）が識別されます。これらはクリアーされ、待ち行列に入れられます。

全てのReferenceタイプは、待ち行列に入れられる前にクリアーされるため、事後クリーンアップを処理するスレッドは参照対象オブジェクトにアクセスできず、アクセスできるのはReferenceオブジェクトのみです。この理由から、Referenceを参照キューと共に使う場合には、適当な参照タイプをサブクラス化することが一般的です。そしてそれを、（Map.EntryがWeakReferenceを拡張する、WeakHashMapの場合と同じように）直接設計の中で使用するか、あるいは、クリーンアップを必要とする実体への参照を保存するために使用します。

参照処理によるパフォーマンス・コスト

参照オブジェクトによって、ガーベジ・コレクションのプロセスに幾らか余分なコストがかかることになります。つまりガーベジ・コレクションの度毎に、ライブReferenceオブジェクトのリストを構築する必要があり、しかも各参照を適切に処理しなければなりません。これはつまり、ガーベジ・コレクションの都度、Reference毎に（その参照対象がガーベジ・コレクションされるか否かにかかわらず）幾らかのオーバーヘッドが追加される、ということを意味します。Referenceオブジェクトそのものはガーベジ・コレクションの対象であり、参照対象よりも前にガーベジ・コレクションされるかも知れませんが、その場合は待ち行列には入れられません。

配列ベースの集合

オブジェクト・ロイタリングが発生する、もう一つの状況は、スタックや循環バッファーのようなデータ構造を実装するために配列を使用する場合です。リスト3のLeakyStackクラスは、配列に基づくスタックの実装を示しています。pop() メソッドの中でelementsは、先頭ポインターが減少しても、スタックからポップされたオブジェクトに対する参照を維持し続けます。これはつまり、そのオブジェクトに対する参照は、プログラムがその参照を二度と使わないにもかかわらず、相変わらずプログラムから到達可能だということです。そのため、その場所が今後のpush()で再度使われない限り、そのオブジェクトはガーベジ・コレクションされずに残るのです。

リスト3.　配列ベースの集合におけるオブジェクト・ロイタリング

```
class LeakyStack {
    private Object[] elements = new Object[MAX_ELEMENTS];
    private int size = 0;

    public void push(Object o) { elements[size++] = o; }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        else {
            Object result = elements[--size];
            // elements[size+1] = null;
            return result;
        }
    }
}
```

この場合のオブジェクト・ロイタリングに対する治療法は、その参照をスタックからポップした後、その参照をヌル化してしまうことです。これは、リスト3でコメントアウトされたコード行として示されています。しかしこの状況（つまりクラスが自分自身のメモリーを管理している状況）は、「もはや必要なくなったオブジェクトを明示的にヌル化することが適切である」という、ごく稀な例外的な状況にすぎません。ほとんどの場合は、使われていないと思われる参照を

懸命にヌル化しても、パフォーマンスやメモリー利用の点で、全く得られるものはありません。そんなことをしても、パフォーマンスが悪化するか、あるいはNullPointerExceptionに終わるだけです。このアルゴリズムをリンク実装すれば、この問題は起きないかも知れません。つまりリンク実装では、リンク・ノード（つまり保存されているオブジェクトへの参照）のライフタイムは、そのオブジェクトが集合の中に保存されている期間と自動的にリンクされます。また、この問題を解決するために、弱参照を使うこともできます（つまり、強参照ではなく弱参照の配列を維持するのです）。しかし現実には、LeakyStackは自分自身のメモリーを管理しており、従って、もはや必要なくなったオブジェクトへの参照が確実にクリアーされることに対して責任を持ちます。スタックやバッファーの実装に配列を使えば、アロケーションを減少させる最適化にはなりますが、配列に保存されている参照のライフタイム管理に注意しなければならず、実装者にとっては、より大きな負担となります。

まとめ

ソフト参照は、弱参照と同様、アプリケーションのオブジェクト・ロイタリングを防止する上で役立ちます。つまりキャッシュをクリアーするための判断を下す際に、ガーベジ・コレクターの助けを借りてオブジェクト・ロイタリングを防止するのです。ただしソフト参照が適切なのは、アプリケーションが、ソフト参照されたオブジェクトを失うことを許容できる場合のみです。

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。