CHRISTINE H. FLOOD

OpenJDKの新しいガベージ・コレクタ

今後リリース予定のG1ガベージ・コレクタとShenandoahガベージ・コレクタについて

OpenJDKのガベージ・コレクション（GC）の世界が熱を帯びてきています。JDK 9で新たにデフォルト・ガベージ・コレクタとなるG1に加えて、Red HatからもShenandoahという新しい代替コレクタがリリースされます。本記事では、現在のパラレル・コレクタとG1の違いを解説し、Shenandoahの機能について検討します。

GCとは

GCは、使用されなくなった再利用可能なメモリを自動的に回収する方法の1つです。オブジェクトを手動で割り当てて破棄する他の言語とは異なり、GCがあれば、プログラマーが各オブジェクトを取得してその必要性を判断するための検査を行う必要がなくなります。すべてを知り尽くした掃除係のようなGCプロセスが裏で音もなく動作して、使い道のなくなったオブジェクトを破棄し、整理します。このような整理整頓によってプログラムが効率化します。

JVMはプログラムのデータをオブジェクトとして構造化します。オブジェクトにはフィールド（データ）が含まれており、そのデータがヒープと呼ばれる管理されたアドレス空間に配置されます。以下のJavaクラスについて考えます。このクラスは、単純なバイナリ・ツリー・ノードを表しています。

```
class TreeNode {
    public TreeNode left, right;
    public int data;
    TreeNode(TreeNode l,  TreeNode r, int d) {
        left = l; right = r; data = d;
    }
    public void setLeft(TreeNode l) { left = l;}
    public void setRight(TreeNode r) {right = r;}
}
```

次に、このクラスに対して以下の操作を実行します。

```
TreeNode left = new TreeNode(null, null, 13);
TreeNode right = new TreeNode(null, null, 19);
TreeNode root = new TreeNode(left, right, 17);
```

このコードでは、位置17をルートとし、位置13に左側のサブノードを、位置19に右側のサブノードを持つバイナリ・ツリーを作成しています（図1）。

この後さらに、右側のサブノードを以下のように置き換えます。この操作により、位置19のサブノードはどのノードにも接続していないガベージ（ゴミ）になります。

```
root.setRight(new TreeNode(null, null, 21));
```

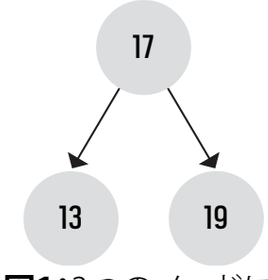

図1：3つのノードによるツリー

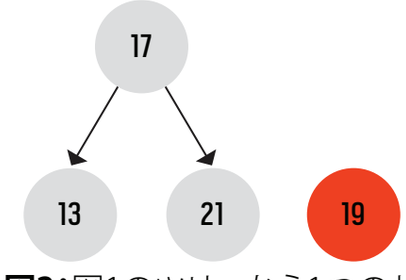

図2：図1のツリーから1つのサブノードを置き換える

この結果、図2のような状況になります。

ご想像のとおり、データ構造の構成と操作の過程で、ヒープは図3のようになっていきます。

データのコンパクションとは、メモリ内のアドレスを変更することを表します。Javaプログラムは、オブジェクトが特定のアドレスにあることを想定しています。ガベージ・コレクタがオブジェクトを移動した場合、Javaプログラムはその新しい位置を把握する必要があります。もっとも簡単な方法は、すべてのJavaスレッドを停止し、すべてのオブジェクトのコンパクションを行い、古いアドレスの参照をすべて新しいアドレスを指すように更新してから、Javaプログラムを再開することです。しかし、この方法ではJavaスレッドが実行されない期間（GC一時停止時間）が長くなります。

アプリケーションが長く稼働しない状態は、Javaプログラマーにとって好ましくありません。GC一時停止時間を減らすための一般的な手法は2つあります。GC関連の文献では、それぞれコンカレント・アルゴリズム（プログラムの実行中にGC処理を行う）およびパラレル・アルゴリズム（Javaスレッドの停止中にGC処理時間を短縮するため、スレッド数を増やす）と呼ばれています。現在のOpenJDKのデフォルト・ガベージ・コレクタはパラレル・アルゴリズムを採用しています（このガベージ・コレクタは、コマンドラインで-XX:+UseParallelGCを追加して手動で指定できます）。パラレル・ガベージ・コレクタは、多数のGCスレッドを利用して並外れたスループットを実現します。

パラレル・ガベージ・コレクタ

パラレル・ガベージ・コレクタは、オブジェクトが「生き残って」きたGCサイクル数に応じて、オブジェクトを2つの領域（若い世代の領域と古い世代の領域）に振り分けます。若いオブジェクトはまず若い世代の領域に割り当てられます。その後、コンパクションの段階で、若い世代にコレクションされてきた回数が確認され、一定回数以下であればそのまま若い世代の領域に留まります。この回数を超えて生き残っているオブジェクトは、古い世代へと移動されます。ヒープ全体のコレクションには時間がかかりすぎるため、そのために一時停止するよりも、ヒープの中で生存期間の短いオブジェクトが含まれる可能性の高い部分のみをコレクションすればよい、という論理です。最終的には、古いオブジェクトもコレクションが必要になります。

ガベージ・コレクタが若いオブジェクトのみをコレクションするためには、古い世代のどのオブジェクトが若い世代のオブジェクトを参照しているかを把握する必要があります。さらに、新しいオブジェクトの新しい位置を参照するように古いオブジェクトを更新する必要があります。そのために、JVMはカード・テーブルと呼ばれる概要データ構造を管理します。古い世代のオブジェクトに参照が書き込まれるときは常に、カード・テーブルに印が付けられます。そうすることで、次に若い世代へのGCサイクルが発生したときに、JVMがこのカードをスキャンして、古い世代から若い世代への参照を探すことができます。パラレル・ガベージ・コレクタはこのような参照を把握しているため、取り除くべきオブジェクトや更新すべき参照を特定できます。パラレル・ガベージ・コレクタは複数のGCスレッドを用いて、プログラムの一時停止中のGC処理時間を短縮します。

> Shenandoahはデータのコンパクションを同時実行的に処理します。そのため、Shenandoahでは、アプリケーションの一時停止時間を最小化するために、収集する領域の数を制限する必要はありません。

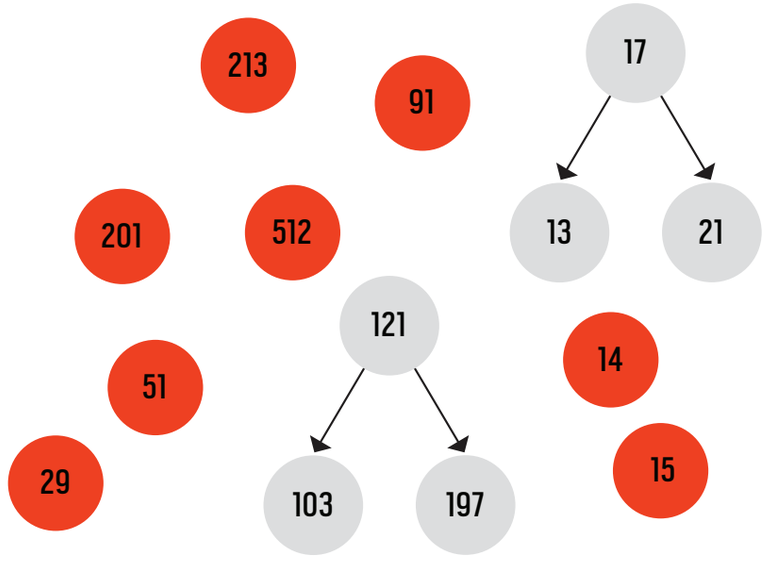

図3：多くの未使用のデータ項目が含まれるヒープ

ガベージ・ファースト・ガベージ・コレクタ

G1（ガベージ・ファースト）と名付けられた新しいJDKガベージ・コレクタは、パラレル・スレッドとコンカレント・スレッドの両方を利用します。G1はコンカレント・スレッドを用いて、生きているオブジェクトをJavaプログラムの実行中にスキャンします。また、パラレル・スレッドを用いてオブジェクトをすばやくコピーし、一時停止時間を短縮します。

G1はヒープを多数の領域に分割します。プログラム実行中の任意の時点において、各領域は、古い世代と若い世代のいずれかの領域になります。若い世代の領域はCG一時停止時間が発生するごとにコレクションする必要がありますが、古い世代の領域については、ユーザーが指定した一時停止時間目標以内にコレクションできると予測した数だけ、柔軟にコレクションします。G1はこの柔軟性によって、ガベージのもっとも多いヒープ領域に集中して、古いオブジェクトのGC処理を実行できます。また、G1はユーザーが指定した一時停止時間に基づいて、コレクションによる一時停止時間を調整することもできます。

図4に示すとおり、G1はオブジェクトを自由にコンパクションして新しい領域に配置できます。

G1は、各領域で生きているデータの量と、その生きているデータのコピーにかかるおおよその時間を把握しています。ユーザーが一時停止時間を最短に抑えたい場合、G1はごく少数の領域のみをコレクションすることができます。ユーザーが一時停止時間を気にしていない場合や、一時停止時間目標をかなり長く設定している場合には、G1はコレクションする領域を増やす可能性もあります。

G1は、若い世代の領域のみをコレクションできるように、カード・テーブルのデータ構造を管理する必要があります。また、他の古い世代の領域から参照されている古い世代の領域ごとのレコードも管理しなければなりません。このデータ構造は、内部記憶集合（into remembered set）と呼ばれます。

短い一時停止時間を指定することの欠点は、G1の処理がプログラムの割当て速度に追いつかない場合があることです。その場合、最終的には処理を取りやめ、代わりに全体を停止する「Full GC」モードを開始します。Full GCモードでは、スキャンとコピーの両方の処理がJavaスレッドの停止中に行われます。部分的なコレクションで一時停止時間目標を達成できない場合には、Full GCが実行され、割り当てられた時間を確実に超過することに注意が必要です。

総じて、G1はスループットの制約と一時停止時間の制約のバランスを図った、優秀な総合的コレクタであると言えます。

Shenandoahによるコンカレント・コピーで特に難しいのが、コピー処理を実行するGCスレッドとヒープにアクセスするJavaスレッドが、オブジェクトのアドレスについて一致した情報を扱う必要があることです。

Shenandoahガベージ・コレクタ

Shenandoahガベージ・コレクタは、まだOpenJDKディストリビューションには含まれていないOpenJDKプロジェクトです。このガベージ・コレクタは、G1と同じ領域ベースのヒープ・レイアウトを利用し、同じコンカレント・スキャン・スレッドを用いて各領域で生きているデータの量を計算します。異なる点は、コンパクション段階での処理方法です。

Shenandoahはデータのコンパクションを同時実行的に処理します（鋭い読者は、アプリケーションがオブジェクトの読取りやオブジェクトへ

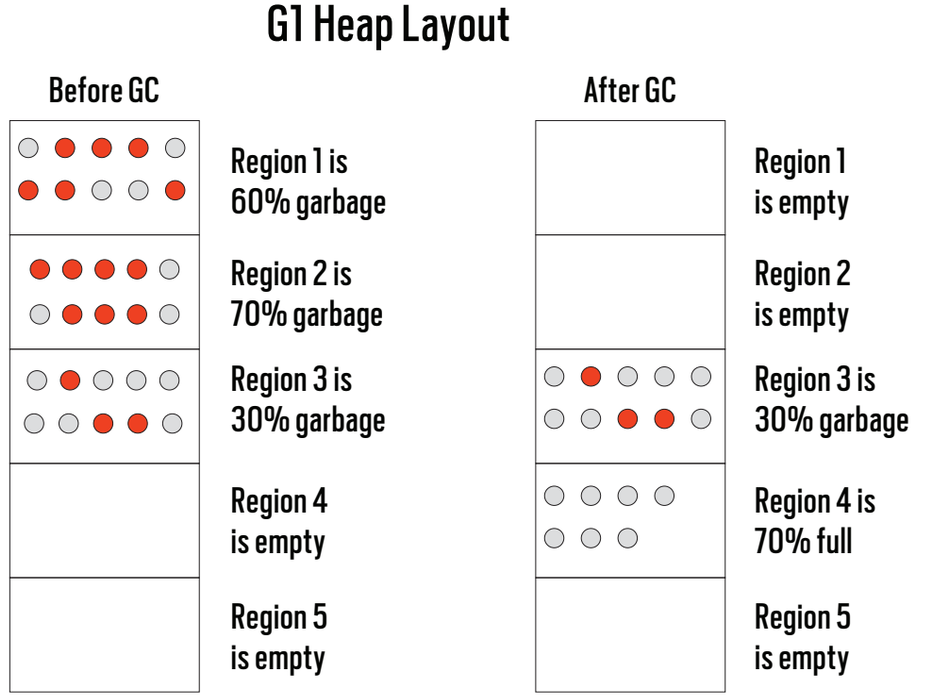

図4：G1実行前と実行後。領域1と領域2は領域4へとコンパクションされます。新しいオブジェクトは領域4に割り当てられます。領域3は、スペースの回収率が少なすぎる（30%）のに対してコピー処理が多すぎる（70%）ため、処理されません。

の書込みを試みるうちに、オブジェクトをあちらこちらへ移動する必要があるのではないかと思うでしょうが、心配はいりません。この点については後ほど説明します）。そのため、Shenandoahでは、アプリケーションの一時停止時間を最小化するために、コレクションする領域の数を制限する必要はありません。代わりに、成果の大きい領域、つまり生きているオブジェクトがほとんど含まれていない領域（逆に言えば「死んだ」スペースが多くある領域）をすべて選びます。一時停止の状態になるのは、スキャンの前後に実行される特定のブックキーピング・タスクに関連するステップだけです。

Shenandoahによるコンカレント・コピーで特に難しいのが、コピー処理を実行するGCスレッドとヒープにアクセスするJavaスレッドが、オブジェクトのアドレスについて一致した情報を扱う必要があることです。このアドレスは場合によって複数の場所に保存されます。また、アドレスの更新が同時に起きているように見える必要があります。コンピュータ・サイエンスにおける他の厄介な問題と同様に、間接参照を一段階追加することでこの問題を解決できます。

オブジェクトは、間接ポインタ用の追加スペースと一緒に割り当てられます。Javaスレッドがオブジェクトにアクセスするときに、まず間接ポインタを読み取って、オブジェクトが移動されたかどうかを確認します。ガベージ・コレクタは、オブジェクトを移動する場合、新しい位置を指すように間接ポインタを更新します。新しいオブジェクトは、自身を指す間接ポインタと一緒に割り当てられます。オブジェクトがGCの実行中にコピーされる場合に限り、間接ポインタが別の位置を指すようになります。

この間接ポインタには、コストがあります。ポインタを読み取ってオブジェクトの現在の位置を把握するために、スペースと時間の両方が消費されます。しかし、これらのコストは思っているほど高くはありません。スペースに関して言えば、Shenandoahでは、部分的コレクションに対応するためのヒープ外のデータ構造（カード・テーブルや内部記憶集合といったもの）が不要です。時間に関して言えば、読取りの問題を解消する各種戦略があります。最適化JITコンパイラは、プログラムが配列サイズなどの不変フィールドにアクセスしていることを認識できます。そのような場合に間接参照の読取りが不要になるように、オブジェクトの古いコピーと新しいコピーのいずれかを読み取るべきです。さらに、Javaプログラムが同じオブジェクトから複数のフィールドを読み取る場合、JITによってそのことを認識して、後の読取りではForwardingポインタの読取りをなくすことができるでしょう。

Shenandoahがコピーしている最中のオブジェクトにJavaプログラムが書き込む場合、競合状態になります。この競合は、JavaスレッドがGCスレッドに協力することで解決します。Javaスレッドがコピー対象となっているオブジェクトに書き込む直前に、まずそのオブジェクトを独自の割当て領域にコピーして、自身が先にオブジェクトをコピーしたことを確認してから、実際の書込みを行うようにします。GCスレッドが先にオブジェクトをコピーしている場合は、Javaスレッドは自分で割り当てたコピーを捨てて、GCのコピーを使用できます。

このように、Shenandoahでは、生きているオブジェクトのコピー中に一時停止する必要がないため、一時停止時間が大幅に短縮されます。

まとめ

エンド・ツー・エンドのスループットを最大限まで高めたい場合は、おそらく現在JDKに付属しているパラレル・ガベージ・コレクタを使用すべきです。一時停止時間とスループットの適度なバランスを求める場合は、新しいG1ガベージ・コレクタが適しています。Shenandoahは、大規模な（20GBを超える）ヒープを伴って実行される、応答時間が重視されるアプリケーション（金融取引、Eコマースなど、ユーザーがGCによる明らかな遅延を不快に思うような対話型アプリケーション）にとって1つの有効な選択肢になります。**</article>**

Christine H. Flood：Red HatでのJavaプラットフォームのプリンシパル・ソフトウェア・エンジニア。Red HatではShenandoah開発を担当。

learn more

- 著者によるShenandoahに関する2014年のプレゼンテーション
- JVMのガベージ・コレクションに関するオラクルのチュートリアル