リアルタイム Java、第 4 回: リアルタイム・ガーベッジ・コレクション

Benjamin Biron
Software Developer
IBM Ottawa Lab

Ryan Sciampacone
Senior Software Developer
IBM Ottawa Lab

2007年 5月 02日

Java の技術がリアルタイム (RT) 開発に適した環境となるのを妨げていたのは従来のガーベッジ・コレクション (GC) の確定性のない一時停止ですが、そんなJava 言語で確定性のある GC 動作を実現するのが、IBM WebSphere Real Time の一部となっている MetronomeGC です。Metronome を他の機能と組み合わせれば、開発者は Java 言語でハード RT アプリケーションを作成することができます。今回の記事では、Metronomeが確定性のある GC のために使用する手法、Metronomeの開発に伴う技術的問題、そしてGC を調整するために用意されているツールと機能について説明します。

リアルタイム・システムとガーベッジ・コレクション

リアルタイム (RT) アプリケーションの開発は、ランタイム動作の局所に時間的制限を課すという点で、汎用アプリケーションとは区別されます。通常そのような制限が設けられるのは、割り込みハンドラーなどといったアプリケーションのセクションです。割り込みハンドラーでは、割り込みに応答するコードが一定時間内にその作業を完了させなければなりません。心臓モニターや防衛システムなどのハードRT システムでは、時間制限を守れなければシステム全体の致命的障害とみなされます。一方ソフト RT システムでは、時間制限に間に合わなかった場合、GUIが監視対象のストリームの結果を一部表示しないなどの悪影響が出ますが、システム障害にはなりません。

Java アプリケーションでは、Java 仮想マシン (JVM) がランタイム動作の最適化、オブジェクト・ヒープの管理、オペレーティング・システムとハードウェアとのインターフェースを取る動作を行います。言語とプラットフォームの間にあるこの管理層はソフトウェア開発を容易にするものの、一定量のオーバーヘッドをプログラムにもたらします。オーバーヘッドの一例は、GCです。GC は一般的に、アプリケーションに確定性のない一時停止を発生させます。GC の一時停止は頻度も期間も予測不可能なことから、Java 言語は従来からRT アプリケーションの開発には不適切だとされてきました。RTSJ (Real-time Specification for Java) に基づく既存のソリューションに

は、開発者がJava 技術の確定性のない側面に関わらなくても済むようにしているものもありますが、それには既存のプログラミング・モデルを変更する必要が生じます。

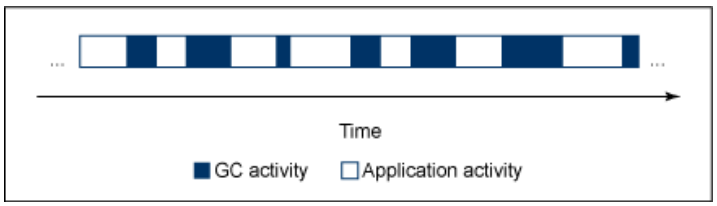
Metronome は、標準 Java アプリケーションでの一時停止時間を短い時間に制限し、指定のアプリケーション使用率を実現する確定性のあるガーベッジ・コレクターです。一時停止時間は、インクリメンタル方式の収集、そしてこのVM に対する抜本的変更をはじめとする慎重な設計上の決定によって、制限かつ短縮されます。使用率とは、特定の期間内にアプリケーションが実行を許可される時間のパーセンテージで、その期間の残りの部分はGC 専用となります。Metronome では、アプリケーションに配分する使用率レベルをユーザーが指定できます。Metronome を RTSJと組み合せて使用することで、開発者は、短い一時停止時間による確定性を備え、さらに期間が極めて短い場合には一時停止が発生しないソフトウェアを作成できます。この記事では、RTアプリケーションに関する従来の GC の制約事項と Metronome の詳細な手法を説明し、Metronome を使ってハード RT アプリケーションを開発する際のツールと手引きを紹介します。

従来の GC

従来の GC 実装では、ヒープ・メモリーの復旧に STW (Stop-The-World) という手法を使用します。この手法では、アプリケーションがヒープの空きメモリーがなくなるまで動作し、空きメモリーがなくなった時点でGC がすべてのアプリケーション・コードを停止してガーベッジ・コレクションを実行します。ガーベッジ・コレクションが済むと、アプリケーションは続行できるようになります。

図 1 に、GC アクティビティーに対する従来の STW 一時停止を示します。この場合の一時停止は通常、頻度も期間も予測不可能です。従来の GCに確定性がない理由は、メモリーの復旧に必要な作業量が、アプリケーションで使用するオブジェクトの合計数とサイズ、オブジェクト間の相互接続、そして以降の割り当てに十分なヒープ・メモリーを解放するために必要な作業量に依存するためです。

図 1. 従来の GC による一時停止



従来の GC に確定性がない理由

GC 時間が無期限で予測不可能な理由は、GC の基本コンポーネントについて考えると理解できます。GC による一時停止は通常、マークとスイープという2 つのフェーズで構成されます。さまざまな実装と手法を駆使して、これらのフェーズの意味を結合または変更することや、別の手段 (圧縮してヒープ内のフラグメント化を減少させるなど)で GC を拡張したり、あるいは特定のフェーズを実行中のアプリケーションと同時に動作させることは可能ですが、この 2 つのコンセプトが従来のGC での技術的な基本となっています。

マーク・フェーズの目的は、アプリケーションに可視のすべてのオブジェクトを追跡し、これらのオブジェクトのストレージが再利用されないようにライブ・オブジェクトとしてマークを付けることです。この追跡は、スレッド・スタック、オブジェクトのグローバル参照などの内部構造

で構成されるルート・セットから開始されます。このルート・セットから(直接または間接的に)到達可能なすべてのオブジェクトがマークされるまで一連の参照がトラバースされます。マーク・フェーズの完了時にマークが付けられていないオブジェクトは、アプリケーションが到達できなかったオブジェクトで、ルート・セットからどの参照を辿っても検出できなかったオブジェクトです(デッド・オブジェクト)。マーク・フェーズの長さが予測不可能なのは、特定時点でのアプリケーションのライブ・オブジェクトの数、そしてすべての参照をトラバースしてシステム内のすべてのライブ・オブジェクトを検出するためのコストが予測できないためです。一貫したシステム動作での以前の時間特性に基づいて所要時間を予測するという方法も考えられますが、予測がどれだけ正確であるかどうかが、もう1 つの不確定性の要因となってしまいます。

スイープ・フェーズの目的は、マーク・フェーズの完了後にヒープを調べ、デッド・オブジェクトのストレージをヒープ用の空きストアに戻して割り当てに再利用できるようにすることです。マーク・フェーズと同様、デッド・オブジェクトを空きメモリー・プールにスイープして戻す際のコストはまったく予測できません。システム内のライブ・オブジェクトの数とサイズはマーク・フェーズから算定できますが、ヒープ内でのオブジェクトの位置と空きメモリー・プールに対する適合性を分析するために必要となる作業量は予測不可能となるかもしれません。

RT アプリケーションに対する従来の GC の適合性

RT アプリケーションは確定的な時間内で実世界からの影響に対応しなければなりませんが、従来の GC ではこの要件を満たすことができません。GCが使用されていないメモリーを取り戻すためには、アプリケーションが停止しなければならないためです。メモリーを取り戻すのに必要な時間には制限がなく、さまざまに変動します。その上、GCがどの時点でアプリケーションに割り込むかも従来から予測できません。アプリケーションが停止する期間は、GC が空きスペースを取り戻すためにアプリケーションの進行が一時停止される時間であることから、一時停止時間と呼ばれます。一時停止時間は一般的に、アプリケーション応答性の上限時間を表すため、RTアプリケーションの場合は短い一時停止時間が要件となります。

Metronome GC

Metronome の手法は、GC サイクルに費やされる時間をクォンタ (quanta) と呼ばれる時間単位に分割するというものです。この手法に従って各フェーズは個別ステップを重ねて作業全体を完了するように設計されているため、コレクターは以下のように動作できます。

1. 非常に短い確定した期間、アプリケーションをプリエンプトする
2. 収集作業を進める
3. アプリケーションを再開させる

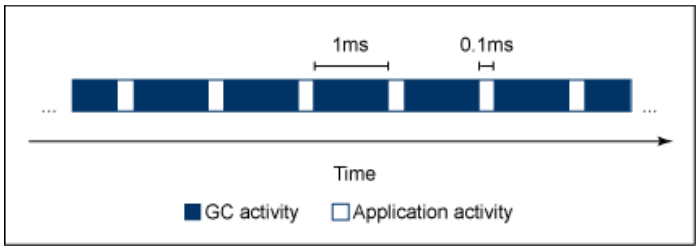
従来のモデルでは、アプリケーションが不意に停止し、GC が時間の制限なく完了するまで行われてから GC が停止し、アプリケーションが再開しますが、上記のシーケンスはそれとは対照的です。

STW GC サイクルを短時間に制約した一時停止に分割すれば GC の影響を軽減できますが、RT アプリケーションにはそれだけでは足りません。RTアプリケーションの時間要件を満たすには、どの期間内でも十分な時間をアプリケーション専用に割り当てなければなりません。そうでないと要件に反することになり、アプリケーションは失敗します。例えば、GC一時停止が 1 ミリ秒に制限されているというシナリオを考えてみてください。1 ミリ秒の GC 一時停止の合間にアプリケー

ションが実行できる時間が0.1 ミリ秒だけだとすると、アプリケーションはほとんど進行しません。これでは、それほど複雑でもない RT システムでも作業を進める時間がないため失敗します。事実上、一時停止の間隔が短すぎると、完全なSTW GC と何ら変わらなくなります。

図 2 に示すのは、GC の実行が時間の大部分を占めるにも関わらず、1 ミリ秒の一時停止時間をそのまま維持する場合のシナリオです。

図 2. 一時停止時間は短いながらも、アプリケーションの時間がほとんどない場合



使用率

一時停止時間を制限する他、アプリケーションと GC の両方に割り当てる時間のパーセンテージに確定性を持たせるための手段も必要です。アプリケーション使用率は、アプリケーションを最後まで実行する間に連続してスライドしていく時間ウィンドウ内でアプリケーションに割り当てられる時間のパーセンテージとして定義されます。Metronomeが保証するのは、処理時間の特定のパーセンテージがアプリケーション専用になるということです。残りの時間をどう使用するかは GC 次第で、アプリケーションに割り当てられることもあれば、GCが使用する場合もあります。一時停止時間を短くすることにより、従来のコレクターよりきめの細かい使用率が保証されます。使用率の測定間隔がゼロに近くなるにつれ、アプリケーションの予想使用率は0% または 100% のいずれかとなります。これは、測定時間は GC クォンタのサイズより小さいためです。使用率は、完全にスライディング・ウィンドウのサイズの厳密な測定に基づいて保証されます。Metronomeが使用するクォンタは 10 ミリ秒のウィンドウで 500 マイクロ秒です。デフォルト使用率ターゲットは 70% に設定されています。

図 3 に、500 マイクロ秒のタイムスライス単位で分割され、10 ミリ秒のウィンドウで 70% の使用率を維持する GC サイクルを示します。

図 3. スライディング・ウィンドウでの使用率

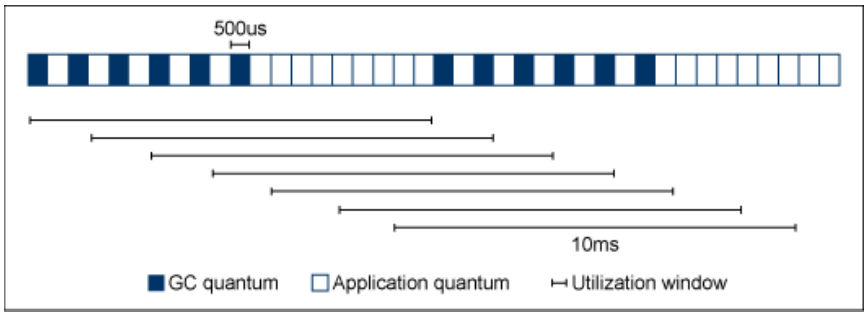


図 3 では、それぞれのタイムスライスが GC またはアプリケーションを実行するクォンタを表し、タイムスライスの下にあるバーがスライディング・ウィンドウを表しています。いずれのスライディング・ウィンドウにも最大6 つの GC クォンタと最小 14 のアプリケーション・クォンタがあります。連続する GC クォンタでターゲット使用率を維持することになったとしても、各

GC クォンタの後には少なくとも 1 つのアプリケーション・クォンタが続きます。これにより、アプリケーションの一時停止時間が 1 クォンタの長さに制限されるというわけです。ただし、ターゲット使用率が50% 未満に設定されると、GC が割り当てに対応できるようにするために GC クォンタが連続することになります。

図 4. と図 5 に、一般的なアプリケーション使用率のシナリオを示します。図 4 では、使用率が 70% に下がっている領域が、GC サイクルが実行されている領域です。GCがアクティブでない場合はアプリケーションの使用率が 100% になっていることに注目していください。

図 4. 全体的な使用率

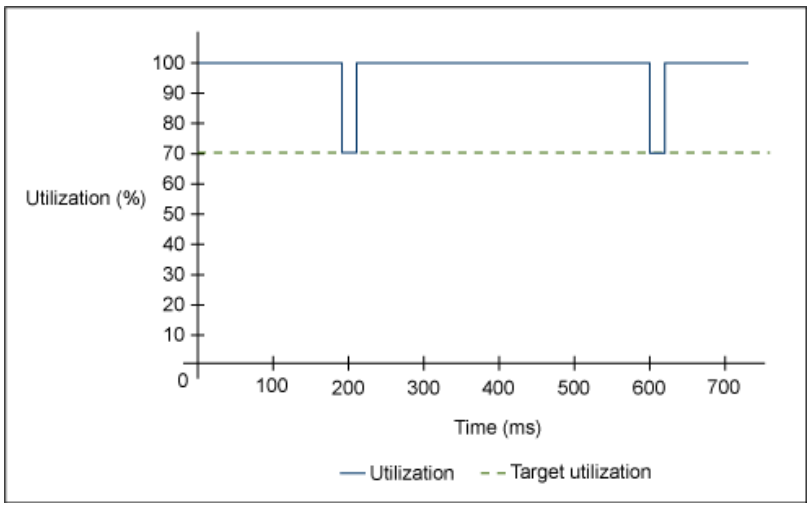


図 5 に、図 4 の GC サイクルの部分を抜粋します。

図 5. GC サイクル使用率

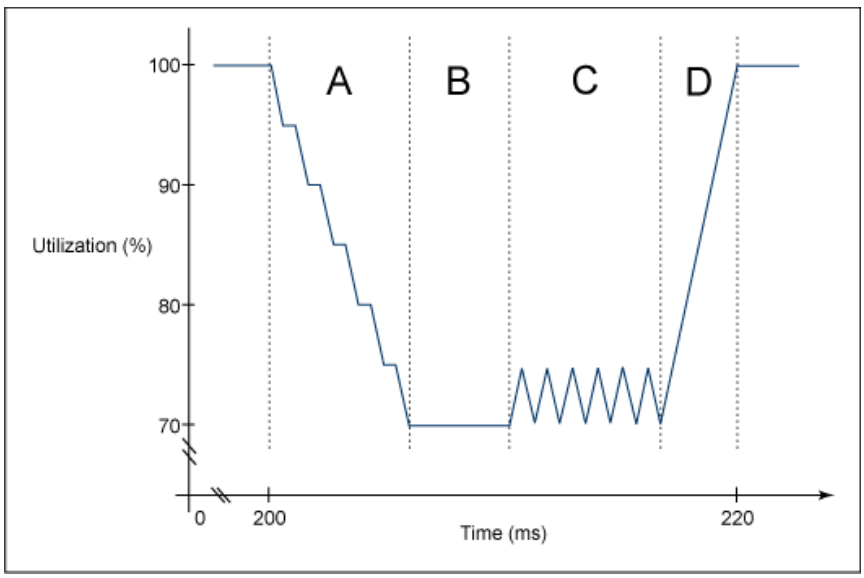


図 5 の A の部分は階段状のグラフになっています。下降している部分は GC クォンタに相当し、平らな部分はアプリケーション・クォンタに相当します。この階段が示しているのは、GCはアプリケーションとインターリーブすることによって短時間の一時停止に従い、その結果ターゲット

使用率までの下降が階段状になっているということです。Bの部分に含まれるのはアプリケーション・アクティビティーのみで、すべてのスライディング・ウィンドウで使用率ターゲットを維持しています。使用率パターンの先頭のみにGC アクティビティーが示されることは珍しくありません。このようなパターンになるのは、GC は実行が許可される場合は常に (一時停止時間と使用率を維持して)実行されるためです。つまり、GC は割り当てられた時間をパターンの最初に使い果たし、時間ウィンドウの残りの部分でアプリケーションが復旧できるようにします。Cの部分は、使用率がターゲット使用率に近づいたときの GC アクティビティーを示しています。上昇している部分はアプリケーション・クォンタを、下降している部分はGC クォンタを表します。この部分がギザギザの状態になっているのも、一時停止を短時間に維持するために GC とアプリケーションがインターリーブするためです。Dの部分は GC サイクルが完了する前の部分で、ここでの上昇は、GC が実行状態でなくなったため、アプリケーションが再び 100% の使用率になることを示しています。

Metronome では、ターゲット使用率をユーザーが指定できます。詳細は、この記事の「Metronome の調整」セクションで説明します。

Metronome を使用したアプリケーションの実行

Metronome は、既存のアプリケーションに RT 動作を提供するように設計されています。ユーザー・コードの変更は必要ありませんが、ターゲット使用率が所望のアプリケーション・スループットを維持すると同時に、GCが割り当てに対応できるようにするためには、理想的なヒープ・サイズとターゲット使用率をアプリケーションに合わせて調整しなければなりません。RT特性が保たれ、アプリケーション・スループットが十分であることを確実にするには、ユーザーが維持したいと思う最大の負荷でアプリケーションを実行してください。スループットまたは使用率が十分でない場合に講じることができる措置については、この記事の「Metronome の調整」 セクションで説明します。特定の状況においては、Metronome によって短時間の一時停止が保証されるとしても、アプリケーションの RT 特性に対応しきれません。そのような場合は、RTSJを使用して、GC による一時停止時間を回避できます。

RTSJ (Real-time Specification for Java)

RTSJ とは、「Java プログラムをリアルタイム・アプリケーションとして使用できるようにするための Java プラットフォームに対する補足仕様」です。Metronomeは、特に `RealtimeThread` (RT スレッド)、`NoHeapRealtimeThread` (NHRT)、そして永続メモリーをはじめとする RTSJ の特定の特徴を認識する必要があります。RT スレッドは Java スレッドで、そのとりわけ重要な特性として、正規Java スレッドより高い優先順位で実行されます。NHRT は RT スレッドですが、このスレッドにはヒープ・オブジェクトの参照を含めることができません。言い換えれば、NHRTにアクセス可能なオブジェクトは GC に影響されるオブジェクトを参照できないということです。その代わり、GC は GC サイクルの間でさえも、NHRTのスケジューリングを妨げません。つまり、NHRT は一時停止時間の影響を受けません。永続メモリーは、GC に影響されないメモリー空間を提供し、NHRTが永続オブジェクトを参照できるようにします。これらの特徴は RTSJ の一部でしかありません。完全な仕様へのリンクは、「参考文献」に記載してあります。

確定性のある GC に伴う技術的問題

Metronome は J9 仮想マシン内でいくつかの重要な手法を用いて、確定性のある一時停止時間を実現するとともに、GC の安全性を保証します。これらの手法には、arraylet、時間に基づくガーベッジ・コレクターのスケジューリング、ライブ・オブジェクトの追跡を目的としたルート構造

の処理、すべてのライブ・オブジェクトを確実に検出するためのJ9 仮想マシンと GC との連携、そして GC クォンタに対して J9 仮想マシンを中断するためのメカニズムが含まれます。

arraylet

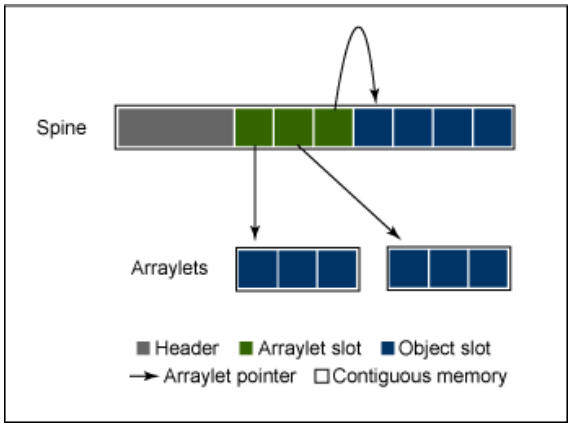
Metronome は収集プロセスをインクリメンタル方式の作業単位に分割して確定性のある一時停止時間を実現しますが、状況によっては、割り当てがGC に一時的な中断を引き起こすことがあります。その一例は、大きなオブジェクトの割り当てです。ほとんどのコレクター実装では、割り当てサブシステムが空きヒープ・メモリーのプールを保持します。このプールのメモリーは、アプリケーションがオブジェクトの割り当てで消費し、コレクターがスイープによって補充します。最初の収集が行われた後の空きヒープ・メモリーには、以前はライブ・オブジェクトで、現在はデッド・オブジェクトとなっているオブジェクトが含まれることになります。これらのオブジェクトがいつ、どのようにしてデッド・オブジェクトになるかのパターンは予測できないため、隣り合うデッド・オブジェクトが合体されたとしても、ヒープ上の空きメモリーはさまざまなサイズのフラグメント化されたチャンクの集まりとなります。さらに、収集サイクルごとにパターンが異なるフリー・チャンクが返される可能性もあります。その結果、メモリーの空きチャンクが要求を満たすだけの大きさでない場合、かなり大きなオブジェクトの割り当ては失敗することになります。そのような大きなオブジェクトの一般的な例は配列です。標準オブジェクトが数十のフィールド以上に大きくなることはめったにないので、たいていのJVM では通常、オブジェクト・サイズが 2K に達することはありません。

フラグメント化の問題を解決するため、一部のコレクターでは収集サイクルに圧縮 (デフラグ) フェーズを実装しています。スイープの完了後に割り当て要求に対応できなければ、システムは既存のライブ・オブジェクトをヒープ内で移動し、複数のフリー・チャンクを1 つの大きなチャンクに合体させようとします。このフェーズは、オンデマンド機能として実装されたり、コレクターのファブリック (セミスペース・コレクターなど)に組み込まれたり、あるいはインクリメンタル方式になることがあります。いずれのシステムにしてもそれぞれ妥協点がありますが、一般的に言うと、圧縮フェーズは時間の点でも作業の点でもコストがかかる方法です。

WebSphere Real Time の現行の Metronome バージョンでは、圧縮システムを実装していません。フラグメント化が問題にならないようにするため、Metronomeでは、arraylet を使用しています。arraylet とは、標準的な線形表現を複数の別個の部分に分割し、それぞれ単独で割り当てられるようにする手法です。

図 6 では、配列オブジェクトをスパインとして表しています (ヒープ上の他のオブジェクトが唯一参照できる集中オブジェクト)。実際の配列の内容が含まれるのは、一連のarrayletリーフです。

図 6. arraylet



arraylet リーフは他のヒープ・オブジェクトからは参照されないため、任意の位置と順序でヒープ全体に分散できます。これらのリーフのサイズは固定され、付加的な間接参照である要素の位置を簡単に計算できるようになっています。図6 に示されているように、リーフの後続データをスパインに組み込むことにより、スパイン内部でのフラグメント化によるメモリー使用のオーバーヘッドは最適化されます。

このフォーマットでは、アレイ・スパインのサイズが無制限に大きくなる可能性がありますが、既存のシステムでは今のところまだ問題になっていません。

GC クォンタのスケジューリング

GC の確定性のある一時停止をスケジューリングするため、Metronome では以下の 2 種類のスレッドを使用して、一貫したスケジューリング、そして短時間で割り込みのない一時停止の両方を実現しています。

- **アラーム・スレッド。**Metronome では GC クォンタのスケジューリングに確定性を持たせるため、アラーム・スレッドをハートビート・メカニズム専用にしています。アラーム・スレッドの優先度は非常に高く(システム内の他のどの JVM スレッドよりも高い優先度)、GC クォンタ (Metronome では 500 マイクロ秒) と同じ周期でウェイクアップしてGC クォンタのスケジューリングを行うべきかどうかを判断します。スケジューリングの必要がある場合、アラーム・スレッドは実行中の JVM を中断してGC スレッドを稼動状態にします。アラーム・スレッドがアクティブになる期間は極めて短いため (通常は 10 マイクロ秒未満)、アプリケーションが認識することはありません。
- **GC スレッド。**GC スレッドは、GC クォンタの間、実際の作業を行うスレッドです。GC スレッドはまず、アラーム・スレッドによる JMV の中断状態を終了させます。それからクォンタの残りの部分でGC 作業を実行し、スレッド自体のスケジューリングにより、クォンタの終了時点が近くなるとスリープ状態に戻って JVM を再開します。GC スレッドは、クォンタが終了するまでに次の作業項目を完了できない場合には、プリエンプトされてスリープ状態になることもあります。RTSJに関して言えば、GC スレッドの優先順位は NHRT 以外のすべての RT スレッドを上回ります。

連携中断メカニズム

Metronome では一連の小さなインクリメンタル方式の一時停止を使用して GC サイクルを完了しますが、それぞれのクォンタに対してはSTW方式で JVM を中断しなければならないことには変わ

りありません。これらの STW の一時停止ごとに、Metronome はJ9 仮想マシンで連携中断メカニズムを使用します。このメカニズムは、特別なネイティブ・スレッド機能に依存してスレッドを中断することはしません。代わりに使用するのは、非同期スタイルのメッセージング・システムです。このシステムを使用してJava スレッドに対し、ヒープをはじめとする内部 JVM 構造へのアクセスを解放し、処理の再開が指示されるまでスリープするよう通知します。J9仮想マシン内の Java スレッドは、中断要求が発行されたかどうかを定期的にチェックし、発行されている場合は以下のように動作します。

1. 保持されている内部 JVM 構造をすべて解放します。
2. 保持されているオブジェクト参照を明示的ロケーションに保管します。
3. 中央 JVM 中断メカニズムに、安全な状態になったことを通知します。
4. スリープ状態になって再開の指示を待機します。

スレッドは再開時にオブジェクト・ポインターを再び読み取り、保持していた JVM 関連の構造を再取得します。JVM 構造を解放するという行為により、GCスレッドは安全な方法で構造を処理できるようになります。部分的に更新した構造の読み取り/書き込みを行うと、予期しない動作によってクラッシュする可能性があるためです。オブジェクト・ポインターを保管してから再ロードすることで、スレッドはGC クォンタの期間内に GC がオブジェクト・ポインターを更新できるようにしています。オブジェクト・ポインターの更新は、圧縮などの操作の一環としてオブジェクトが移動された場合に必要になります。

中断メカニズムは Java スレッドと連携するため、各スレッドでの定期的チェックの間隔をできるだけ短くすることが重要になりますが、これは JVMと JIT (Just-In-Time) コンパイラー両方の役目です。中断要求のチェックはオーバーヘッドをもたらすとはいえ、GC のニーズという点では、このチェックによってスタックなどの構造が明確に定義されるため、GCはスタック内の値がオブジェクトへのポインターであるかどうかを正確に判断できます。

この中断メカニズムは、JVM 関連のアクティビティーに現在関わっているスレッドだけに適用されます。Java 以外のスレッド、あるいは JNI(Java Native Interface) コードに含まれず、JNI API を使用しないJava スレッドが中断されることはありません。中断メカニズムが適用されるスレッドが何らかのJVM アクティビティーに関与している場合 (JVM へのアタッチや JNI API の呼び出しなど)、そのスレッドは GC クォンタが完了するまで中断します。これは、Javaプロセスに関連するスレッドをそのままスケジューリングし続けられるという点で重要です。また、これらのスレッド内で少しでも目立ったシステムの混乱があると、スレッドの優先度は尊重されるにしても、GC の確定性に影響する可能性があります。

書き込みバリア

完全な STW コレクターには、アプリケーションでオブジェクト・グラフ内のリンクを混乱させることなく、オブジェクト参照と JVM 内部構造を追跡できるという利点があります。Metronome は GC サイクルを一連の小さな STW フェーズに分割し、GC サイクルの実行をアプリケーションの実行とインターリーブさせることから、システム内のライブ・オブジェクトを追跡し続ける上で潜在的な問題がもたらされます。それは、予期しない動作またはクラッシュが発生する可能性です。このような事態は、アプリケーションがオブジェクトを処理した後、そのオブジェクトの参照を変更して未処理のオブジェクトをコレクターから隠してしまった場合に発生します。図7 に、この隠しオブジェクトの問題を説明します。

図 7. 隠しオブジェクトの問題

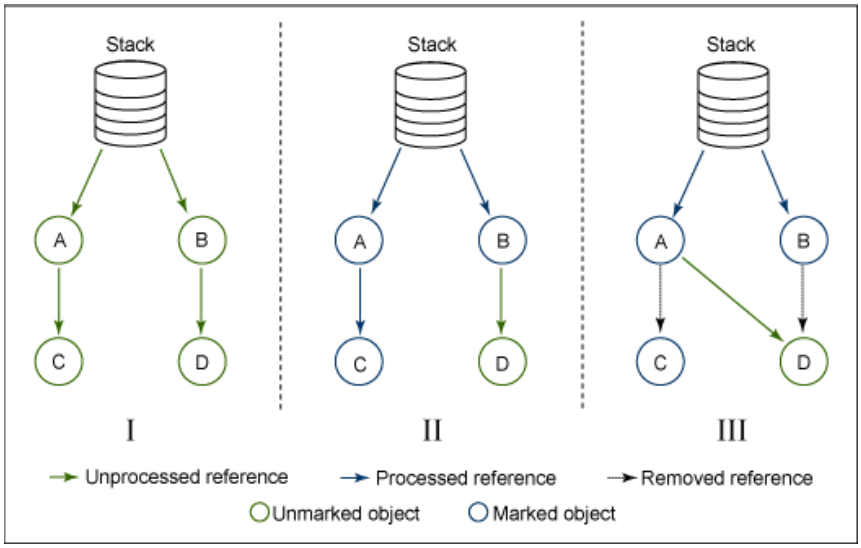


図 7 のセクション I のように、オブジェクト・グラフがヒープ内にあり、Metronome コレクターがアクティブで、このクォンタ内で追跡作業を実行するようにスケジューリングされているとします。コレクターは時間を使い果たしてJVM をセクション II に戻ってスケジューリングしなければならなくなる前に、割り当てられた時間内でルート・オブジェクトとそれによって参照されるオブジェクトをどうにか追跡しました。しかし、アプリケーションの実行中にオブジェクト間の参照が変更され、オブジェクトA は現在、未処理のオブジェクトを指しています。そのため、このオブジェクトはセクション III のどの場所からも参照されなくなっています。GCが別のクォンタに対してスケジューリングし直されて処理を続けると、この隠しオブジェクト・ポインターは見逃されてしまいます。その結果、マークされていないオブジェクトを空きリストに戻すGC のスイープ・フェーズでライブ・オブジェクトが再利用され、ダングリング・ポインターによってJVM または GC で誤った動作、さらにはクラッシュが発生することになります。

このようなエラーを防ぐには、JVM と Metronome とが連携してヒープおよび JVM 構造に対する変更を追跡し、GC がすべての関連オブジェクトを有効に保つようにしなければなりません。その手段となるのが、書き込みバリアです。書き込みバリアはオブジェクトに対するすべての書き込みを追跡し、オブジェクト間の参照の作成および破棄を記録するので、コレクターは隠されている可能性があるライブ・オブジェクトを追跡できます。Metronomeが使用するバリアのタイプは、SATB (Snapshot At The Beginning) バリアと呼ばれます。SATB バリアのコンセプトは、収集サイクルの開始時にヒープの状態を記録し、その時点でのすべてのライブ・オブジェクトならびに現行のサイクルで割り当てられたライブ・オブジェクトを維持するというものです。具体的なソリューションにはYuasa (湯淺) タイプのバリア (「参考文献」を参照) が必要になります。このバリアでは、フィールド・ストアで上書きされた値が記録され、ルート参照が関連付けられているかのように扱われます。上書きする前にスロットの元の値を維持することで、ライブ・オブジェクトのセットを維持および処理できるようにしています。

このようなバリア処理は、JNI グローバル参照リストをはじめとする内部 JVM 構造にも必要です。JNI グローバル参照リストでは、アプリケーションがオブジェクトを追加または除去できます。そのため、除去されたオブジェクトを追跡して(フィールドの上書きと同様に) 隠しオブジェ

クトの問題を回避し、追加されたオブジェクトを追跡して構造を再スキャンする必要をなくすためにバリアが適用されます。

ルートのスキャンと処理

ガーベッジ・コレクターは、ルートから取得した一連の初期オブジェクトからライブ・オブジェクトの追跡を開始します。ルートは JVM 内の構造で、アプリケーションが明示的に作成したオブジェクト(JNI グローバル参照など) あるいは暗黙的に作成したオブジェクト (スタックなど) へのハード参照を表します。ルート構造のスキャンは、コレクターでのマーク・フェーズの初期機能の一環として行われます。

ほとんどのルートは、オブジェクト参照という点に関して実行中にある程度影響されやすくなっています。そのため、「書き込みバリア」で説明したように、一連の参照に対する変更を追跡する必要があります。ただし、スタックなどの特定の構造では、パフォーマンス上の顕著なペナルティーなしでプッシュやポップを追跡することはできません。このような理由から、Metronome ではスタックのスキャンに湯浅式のバリアに対応した特定の制約と変更を加えています。

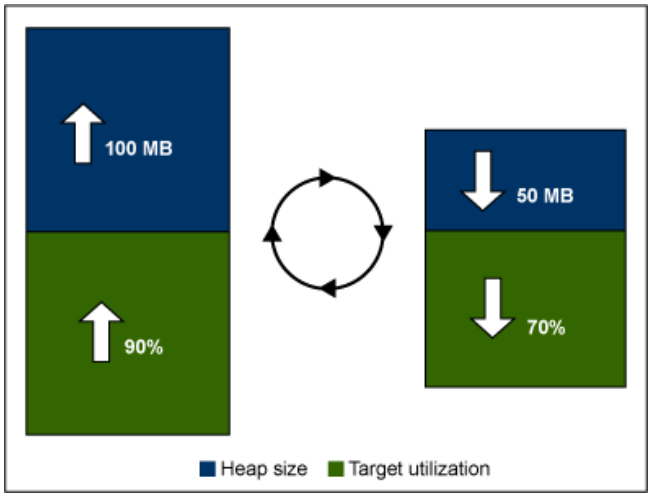
- **スタックのアトミック・スキャン。**個別のスレッド・スタックをアトミック (最小単位) でスキャンするか、あるいは単一のクォンタでスキャンしなければなりません。その理由は、スレッドが実行中に、任意の数の参照(実行中に別の場所に保管された可能性のある参照) をスタックからポップすることがあるためです。スタックのスキャン途中で一時停止すると、ストアを追跡できなくなったり、部分的スキャンの合間でストアを見逃してしまう恐れがあり、その結果、ヒープ内にダングリング·ポインターが作成されてしまいます。アプリケーション開発者は、スタックがアトミックでスキャンされることを認識し、RTアプリケーションで非常に深さのあるスタックを使用しないようにする必要があります。
- **ファジー・バリア。**スタックはアトミックでスキャンする必要がありますが、すべてのスタックを単一のクォンタ内でスキャンするとなると、確定性を保持するのが困難になります。Javaスタックのスキャン中には、GC と JVM が実行をインターリーブできるようになっているため、一連のロードおよびストア操作によって、スレッド間でオブジェクトが移動される可能性があるためです。オブジェクトへの参照を失わないようにするため、GC中にまだスキャンされていないスレッドは、バリアに上書きされた値と保管中の値の両方を追跡させます。書き込みバリアを使用して、保管されたオブジェクトを追跡し、処理済みのオブジェクトに保管されてスタックからポップされたかどうかを判断することが、到達可能性を保持することになります。

Metronome の調整

ヒープ・サイズとアプリケーション使用率の相関関係を理解することは重要です。アプリケーションの最適なスループットには高いターゲット使用率が理想的ですが、GCがアプリケーションの割り当てレートに追いつくことが可能でなければなりません。ターゲット使用率と割り当てレートの両方が高いと、ほとんどの場合はアプリケーションがメモリーを使い果たしてGC をひっきりなしに実行させることになるため、使用率が 0% に下がります。この劣化がもたらすのは、たいていの RT アプリケーションでは許容できないほどの長期間の一時停止です。このようなシナリオが発生した場合、ターゲット使用率を下げてGC の許容時間を増やすか、ヒープ・サイズを大きくして割り当てを増やすか、あるいはその両方の措置を講じるかを選択しなければなりません。場合によっては、一定の使用率ターゲットを維持するために必要なメモリーがないため、パフォーマンスを犠牲にしてターゲット使用率を下げる以外、選択肢がないこともあります。

図 8 に、ヒープ・サイズとアプリケーション使用率の一般的なトレードオフを示します。使用率が高くなると、低い使用率での場合ほど GC を実行できなくなるため、大きなヒープ・サイズが必要になってきます。

図 8. ヒープ・サイズと使用率



使用率とヒープ・サイズとの関係は大幅にアプリケーションに依存するため、適切なバランスを見つけるにはアプリケーションと VM のパラメーターを何度も試してみなければなりません。

Verbose GC

Verbose GC は、GC アクティビティーをログに記録してファイルや画面に出力するツールです。このツールを使用して、パラメーター (ヒープ・サイズ、ターゲット使用率、ウィンドウ・サイズ、およびクォンタ時間)が実行中のアプリケーションをサポートするかどうかを判断できます。リスト 1 は、Verbose GC の出力の一例です。

リスト 1. Verbose GC の出力例

```
<?xml version="1.0" ?>

<verbosegc version="200702_15-Metronome">

<gc type="synchgc" id="1" timestamp="Tue Mar 13 15:17:18 2007" intervalms="0.000">
  <details reason="system garbage collect" />
  <duration timems="30.023" />
  <heap freebytesbefore="535265280" />
  <heap freebytesafter="535838720" />
  <immortal freebytesbefore="15591288" />
  <immortal freebytesafter="15591288" />
  <synchronousgcpriority value="11" />
</gc>

<gc type="trigger start" id="1" timestamp="Tue Mar 13 15:17:45 2007" intervalms="0.000" />

<gc type="heartbeat" id="1" timestamp="Tue Mar 13 15:17:46 2007" intervalms="1003.413">
  <summary quantumcount="477">
    <quantum minms="0.078" meanms="0.503" maxms="1.909" />
    <heap minfree="262144000" meanfree="265312260" maxfree="268386304" />
    <immortal minfree="14570208" meanfree="14570208" maxfree="14570208" />
    <gcthreadpriority max="11" min="11" />
  </summary>
</gc>
```

```
<gc type="heartbeat" id="2" timestamp="Tue Mar 13 15:17:47 2007" intervalms="677.316">
  <summary quantumcount="363">
    <quantum minms="0.024" meanms="0.474" maxms="1.473" />
    <heap minfree="261767168" meanfree="325154155" maxfree="433242112" />
    <immortal minfree="14570208" meanfree="14530069" maxfree="14570208" />
    <gcthreadpriority max="11" min="11" />
  </summary>
</gc>

<gc type="trigger end" id="1" timestamp="Tue Mar 13 15:17:47 2007" intervalms="1682.816"/>

</verbosegc>
```

それぞれの Verbose GC イベントは `<gc></gc>` タグ内に含まれます。さまざまなイベント・タイプを使用できますが、リスト 1 には最も一般的なイベントが記載されています。同期 GC を表す`synchgc` タイプは、最初から最後まで割り込まれずに実行された GC サイクルです。つまり、アプリケーションとのインターリーブは発生していません。インターリーブの発生理由としては、以下の2 つがあります。

- `System.gc()` がアプリケーションによって呼び出された場合
- ヒープがいっぱいになり、アプリケーションがメモリーを割り当てられなかった場合

`<details>` タグに含まれる同期 GC の理由は 最初のケースでは `system garbage collect`、2 番目のケースでは `out of memory` となっています。最初のケースでは、指定されたパラメーターはアプリケーションの持続性には関係しません。ただし、ユーザー・アプリケーションから`System.gc()` を呼び出すと、たいていはアプリケーション使用率が 0% に落ち込んで長期間の一時停止が発生するため、このケースは避けなければなりません。一方、同期GC が 2 番目のケースの理由で発生した場合 (メモリー不足のエラー)、GC がアプリケーション割り当てに追いつけなかったことを意味します。この場合は、ヒープを増やすか、アプリケーション使用率ターゲットを下げるかして、同期GC の発生を回避することを検討しなければなりません。

`trigger` GC イベント・タイプは、GC サイクルの開始点および終了点に対応します。このイベントは、`heartbeat` GC イベントのバッチを区切るのに役立ちます。`heartbeat` GC イベント・タイプは、複数の GC クォンタの情報をまとめて 1 つの Verbose GC のイベントに要約します。このイベントは、アラーム・スレッド・ハートビートとは関係ありません。`quantumcount` 属性は、`heartbeat` GC 内の GC クォンタの数に相当します。`<quantum>` タグは、`heartbeat` GC 内の GC クォンタに関するタイミング情報を表します。`<heap>` および `<immortal>` タグには `heartbeat` GC 内のクォンタが終了した時点での空きメモリーに関する情報、`<gcthreadpriority>` タグにはクォンタが開始した時点での GC スレッドの優先順位情報が含まれます。

クォンタの時間値は、アプリケーションから見た一時停止時間に相当します。平均クォンタ時間は 500 マイクロ秒に近い値でなければならず、最大クォンタ時間を監視してRT アプリケーションで許容できる一時停止時間の範囲内であることを確実にしなければなりません。GC がシステム内の別のプロセスによってプリエンプトされたためにGC がそのクォンタを完了してアプリケーションを再開することができなった場合、あるいはシステム内の特定のルート構造が乱用されて管理不能なサイズに増大した場合(「Metronome を使用する際に考慮すべき問題」を参照) には、一時停止時間が長くなる可能性があります。

永続メモリーは GC の影響を受けない RTSJ に必要なリソースなので、詳細 GC ログの空き永続メモリーが下がったまま復旧していなくても異常なことではありません。このメモリーはストリン

グ定数やクラスなどのオブジェクトに使用されるため、皆さんはプログラムの動作を認識し、永続メモリーのサイズを必要に応じて調整してください。

通常の傾向が安定していることを確認するには、ヒープ使用量を監視します。ヒープの空きスペースが減少傾向にある場合、アプリケーションによるリークが存在することが考えられます。リークが発生するのは、ハッシュ・テーブルが拡大し続けている場合、大規模なリソース・オブジェクトが無期限で保持されている場合、グローバルJNI 参照がクリーンアップされていない場合など、多数の原因があります。

図 9 と図 10 に、ヒープの空きメモリーが安定している場合と、減少傾向にある場合を示します。局所的に上下するのは当然のこととして想定されています。これは、空きスペースが増加するのはGC サイクルの期間中だけで、アプリケーションがアクティブ状態で割り当てを行っている間は減少するからです。

図 9. 安定したヒープの空きスペース

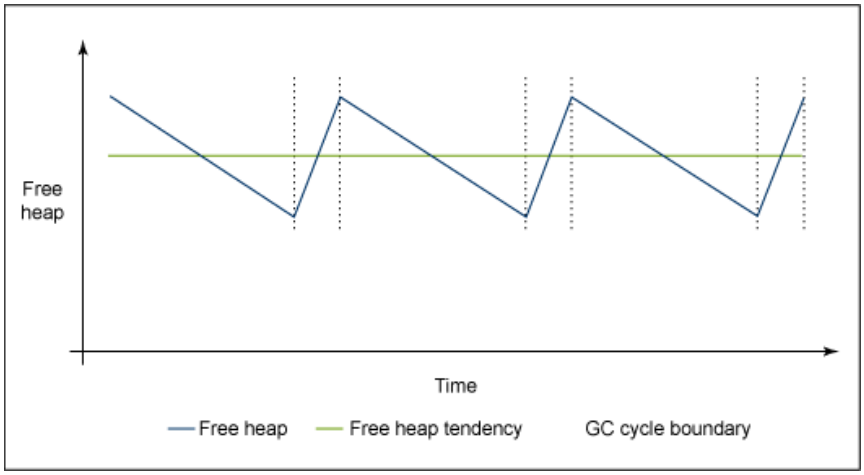
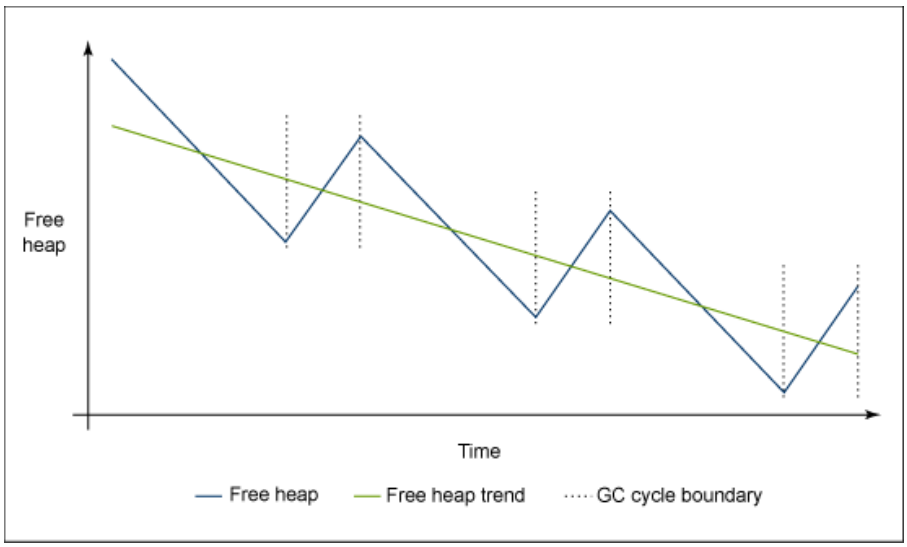


図 10. 減少しているヒープの空きスペース



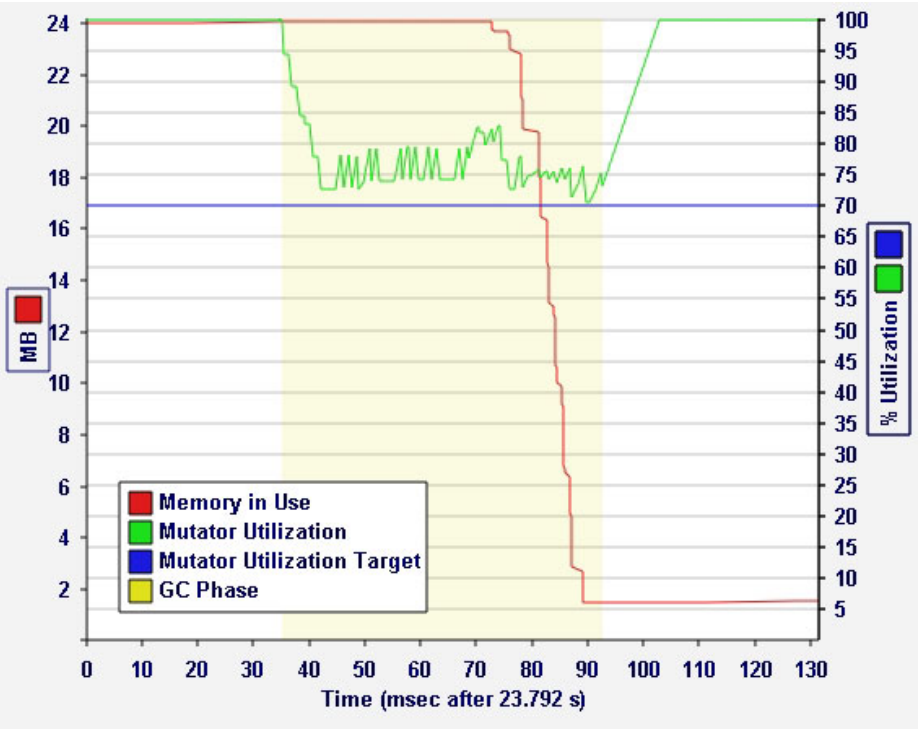
`<gc>` タグの `interval` 属性は、同じタイプの Verbose GC イベントが前回出力されてからの経過時間に相当します。`heartbeat` イベント・タイプの場合、現行 GC サイクルの最初のハートビートであっても、この属性が `trigger start` イベントからの経過時間を示します。

Tuning Fork

Tuning Fork は、ユーザー・アプリケーションに合わせて Metronome を調整するための独立したツールです。Tuning Forkでは、GC の実行後にはトーレス・ログで、GC の実行中にはソケットによってユーザーが GC アクティビティーのさまざまな詳細を調べられるようにしています。TuningFork を考慮してビルドされた Metronome は、Tuning Fork アプリケーション内から調べられる多数のイベントをログに記録します。例えば、長期的なアプリケーション使用率を表示し、それぞれのGC フェーズに費やされた時間を調べることができます。

図 11 は、Tuning Fork で生成された GC パフォーマンス概要のグラフです。このグラフには、ターゲット使用率、ヒープ・メモリー使用量、そしてアプリケーション使用量が示されています。

図 11. Tuning Fork によるパフォーマンス概要



Metronome を使用する際に考慮すべき問題

Metronome は GC の短時間かつ確定性のある一時停止を実現することを目指していますが、この Metronome が実現する結果に悪影響を与え、場合によっては一時停止時間の異常値につながる状況がアプリケーション・コードと基礎となるプラットフォームの両方に発生することがあります。また、標準的なJDK コレクターで当初期待されていた GC の動作が変更される可能性もあります。

RTSJ では、GC が永続メモリーを処理しないことを規定しています。クラスは GC には影響されない永続メモリー内に保持されるため、GC ではアンロードできません。そのため、多数のクラスを

使用するアプリケーションでは適切に永続メモリーを調整し、クラスのアンロードが必要なアプリケーションではWebSphere Real Time 内でプログラミング・モデルを調整する必要があります。

Metronome での GC 作業は時間をベースとしているため、ハードウェア・クロックを変更すると診断が困難な問題が発生する可能性があります。その一例は、システム時刻をNTP (Network Time Protocol) サーバーに同期させてから、ハードウェア・クロックをシステム時刻に同期させるといった変更です。このような変更はGC にとっては急激な時間増加となるため、使用率ターゲットを管理できなくなったり、あるいはメモリー不足のエラーが発生する原因となります。

単一のマシンで複数の JVM を実行すると、JVM 間での干渉が発生して使用率の数字がゆがめられることがあります。高い優先度を持つ RT スレッドであるアラーム・スレッドは他の低い優先度を持つスレッドをプリエンプトし、GCスレッドも同じく RT 優先度で実行されます。十分な GC スレッドとアラーム・スレッドがいつでもアクティブになっている場合、アクティブな GCサイクルを持たない JVM では、その VM の GC がアクティブでないせいで、アプリケーションに時間が割り当てられている期間であっても、そのアプリケーション・スレッドが別のJVM の GC スレッドとアラーム・スレッドによってプリエンプトされる可能性があります。

著者について

Benjamin Biron

Ben Biron は、Carleton University でコンピューター・サイエンスの工学士号を取得した 2006年5月から、J9 Virtual Machine チームに加わっています。彼が専門に取り組んでいるのは、Metronome およびリアルタイム・ガーベッジ・コレクションです。仕事以外の時間は、バレーボール、ゴルフ、そしてオープン・ソース・ソフトウェアの開発を楽しんでいます。

Ryan Sciampacone

Ryan Sciampacone は 1997 年に Carleton University でコンピューター・サイエンスの学士号を取得して以来、コア VM 実装、JNI API 層、AOT コンパイルをはじめとする仮想マシン開発のすべての面に携わってきました。2002 年からは、J9 JVM 向けガーベッジ・コレクションの技術リーダー兼主任アーキテクトとして、JSE 実装で使用可能なスケーラブル・コレクター・スイートならびに Metronome コレクターと ME 構成コレクターに取り組んでいます。仕事以外では、ホッケー、ヨガ、そしてサイクリングを楽しんでいます。