メモリーよ、ありがとう

AIX での JVM によるネイティブ・メモリーの使用状況を理解する

Andrew Hall
Software Engineer
IBM

2009年 4月 21日

Java™ ヒープの不足だけが、`java.lang.OutOfMemoryError` の原因ではありません。ネイティブ・メモリーが不足すると、通常のデバッグ手法では解決できない `OutOfMemoryError` が発生します。この記事では、ネイティブ・メモリーとは何か、Java ランタイムはネイティブ・メモリーをどのように使用するか、ネイティブ・メモリーが不足するとどのような現象になるのか、そして AIX® ではネイティブ・メモリーの `OutOfMemoryError` をどのようにデバッグすればよいのかを説明します。Linux® システムと Windows® システムの場合についての同じトピックは、関連記事で説明しています。

Java ヒープはすべての Java オブジェクトが割り当てられる場所であり、開発者が Java アプリケーションを作成するときに最も深く関係するメモリー領域です。JVM は開発者をホスト・マシンの特異性から隔離するために設計されたため、メモリーについて考えるときにヒープについて考えるのは当然と言えます。オブジェクトのリークが発生したり、ヒープをすべてのデータを格納できるだけの大きさにしなかったりした場合には、間違いなく Java ヒープの `OutOfMemoryError` が発生します。皆さんはこのようなシナリオをデバッグするいくつかのコツをすでに学んでいることでしょう。しかし Java アプリケーションが処理するデータや並行処理の負荷が増えるにつれ、通常のコツを用いても修正できない `OutOfMemoryError` が発生するようになります。つまり、Java ヒープがフルでなくてもエラーがスローされるという事態です。このような事態が発生したときには、JRE (Java Runtime Environment) の内部で何が起こっているのかを理解する必要があります。

Java アプリケーションは Java ランタイムの仮想環境で稼働しますが、ランタイム自体は、ネイティブ・メモリーをはじめとするネイティブ・リソースを使用する言語 (C など) で作成されたネイティブ・プログラムです。ネイティブ・メモリーとは、ランタイム・プロセスが使用できるメモリーのことで、Java アプリケーションが使用する Java ヒープ・メモリーとは区別されます。仮想化されたすべてのリソース (Java ヒープと Java スレッドを含め) は、仮想マシンの実行中に使用されるデータとともに、ネイティブ・メモリーに格納されなければなりません。つまり、ホスト・マシンのハードウェアおよびオペレーティング・システム (OS) によって課せられたネイティブ・メモリーの制約が、Java アプリケーションで実行可能な内容に影響してくるということです。

この記事は、同じトピックを異なるプラットフォームの場合について説明する 2 つの記事のうちの 1 つです。どちらの記事でも、ネイティブ・メモリーとは何か、Java ランタイムはどのようにネイティブ・メモリーを使用するのか、ネイティブ・メモリーが不足するとどのような現象になるのか、そしてネイティブ・メモリーの `OutOfMemoryError` をどのようにデバッグすればよいかのを説明します。この記事で取り上げているのは AIX の場合で、IBM® Developer Kit for Java に重点を置いて説明しています。もう一方の記事では Windows と Linux の場合を取り上げており、特定の Java ランタイムに重点を置いた説明はしていません。

ネイティブ・メモリーについての概要

まずは、OS およびベースとなるハードウェアによって課せられる、ネイティブ・メモリーの制約について説明します。C などの言語で動的メモリーを管理する方法について十分に理解している方は、次のセクションに進んでもらって構いません。

ハードウェアによる制約

ネイティブ・プロセスに課せられる制約の多くは、OS ではなく、ハードウェアによるものです。あらゆるコンピューターにはプロセッサーと、物理メモリーとして知られるランダム・アクセス・メモリー (RAM) があります。プロセッサーはデータ・ストリームを実行すべき命令と解釈します。データ・ストリームには、整数演算、浮動小数点演算、そしてさらに高度な計算を実行する処理単位が 1 つ以上ありますが、プロセッサー内部には多数のレジスターがあります。レジスターとは、実行される計算の作業ストレージとして使用される非常に高速なメモリー要素で、レジスターのサイズによって、1 回の計算処理で扱うことのできる最大値が決まります。

プロセッサーと物理メモリーを接続するのは、メモリー・バスです。アドレス指定可能なメモリーの量は、物理アドレス (プロセッサーが物理 RAM にインデックスを付けるために使用するアドレス) のサイズによって制限されます。例えば、16-bit の物理アドレスで指定できるのは 0x0000 から 0xFFFF までなので、固有のメモリー・ロケーション数は 2^16 = 65536 となります。それぞれのアドレスがストレージの各バイトを参照する場合、16-bit の物理アドレスでプロセッサーがアドレス指定できるのは、64KB のメモリーという計算になります。

プロセッサーは上記のように、特定のビット数で表されます。ビット数はレジスターのサイズを指すのが通常ですが、例外として、例えば 390 31-bit というように物理アドレスのサイズを指す場合もあります。デスクトップおよびサーバー・プラットフォームでは、この数値は 31、32、または 64 となります。組み込みデバイスやマイクロプロセッサーであれば、ビット数が 4 の場合もあります。物理アドレスのサイズは、レジスター幅と同じにすることができますが、それより大きくすることも、小さくすることもできます。大抵の 64-bit プロセッサーは、適切な OS を実行中であれば、32-bit プログラムを実行することができます。

表 1 に、よく使われるアーキテクチャーと、それぞれのレジスターおよび物理アドレスのサイズを記載します。

表 1. よく使用されるプロセッサー・アーキテクチャーのレジスターおよび物理アドレス・サイズ

アーキテクチャー	レジスター幅 (ビット)	物理アドレス・サイズ (ビット)
(最近の) Intel x86	32	32

		物理アドレス拡張を使用した場合は 36 (Pentium Pro 以上)
x86 64	64	現在は 48-bit (後から拡張可能)
PPC64	64	POWER 5 では 50-bit
390 31-bit	32	31
390 64-bit	64	64

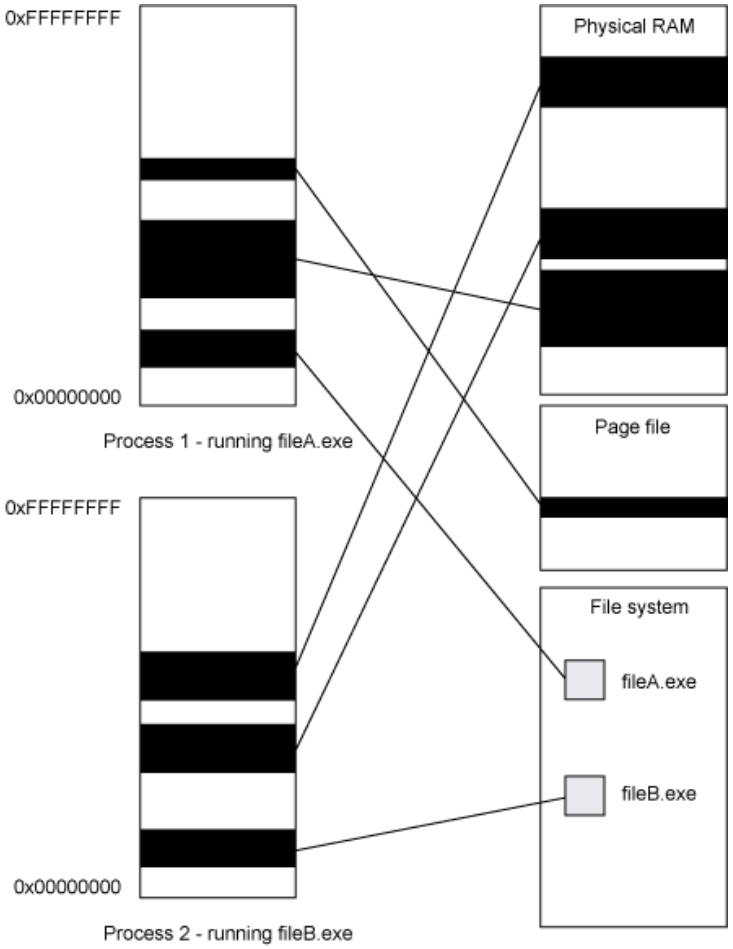
オペレーティング・システムと仮想メモリー

OS を使わずにプロセッサーで直接実行するアプリケーションを作成しているとしたら、プロセッサーがアドレス指定できるメモリーをすべて使用することができます (十分な物理 RAM が接続されているという前提です)。しかし誰もがほとんど、マルチタスクやハードウェア抽象化などの機能を利用するために、何らかの OS を使用してアプリケーションのプログラムを実行しています。

AIX をはじめとするマルチタスク OS では、メモリーを含めたシステム・リソースが複数のプログラムによって使用されます。そのため、それぞれのプログラムには、そのプログラムが動作する物理メモリーの領域を割り当てる必要があります。すべてのプログラムが物理メモリーを直接操作し、与えられたメモリーのみを使用すると想定する OS を設計することは可能です。そのように動作する組み込み OS もなかにはありますが、すべてのプログラムを一緒に使用するテストを行っていない多数のプログラムで構成されている環境では、この動作は実際的でありません。この場合、プログラムのどれもが他のプログラムのメモリーや OS 自体を破壊する可能性があるからです。

仮想メモリーを使用すれば、複数のプロセスが物理メモリーを共有しても、互いのデータを破壊する恐れがありません。仮想メモリーを使用する OS (AIX やその他多くの OS) では、それぞれのプログラムが固有の仮想アドレス空間を持ちます。仮想アドレス空間とはアドレスの論理領域のことで、そのサイズはシステム上のアドレス・サイズによって決まります (つまり、デスクトップおよびサーバー・プラットフォームの場合は 31、32、または 64 ビットとなります)。プロセスが持つ仮想アドレス空間内の領域は、物理メモリー、ファイル、あるいはアドレス指定可能なその他すべてのストレージにマッピングすることができます。OS は物理メモリーに保持されているデータが使用されていないときには、そのデータを物理メモリーからスワップ領域に移し、物理メモリーを最適に使用できるようにします。プログラムが仮想アドレスを使ってメモリーにアクセスしようとすると、OS がオンチップ・ハードウェアと連動してその仮想アドレスを物理ロケーションにマッピングします。そのロケーションは、物理 RAM であることも、ファイルや、スワップ空間であることもあります。メモリー領域がスワップ空間に移された場合には、その領域は使用される前に物理メモリーに再びロードされます。図 1 に、仮想メモリーがプロセスのアドレス空間の領域を共有リソースにマッピングすることによってどのように機能するかを示します。

図 1. 仮想メモリーによるプロセスのアドレス空間と物理リソースとのマッピング

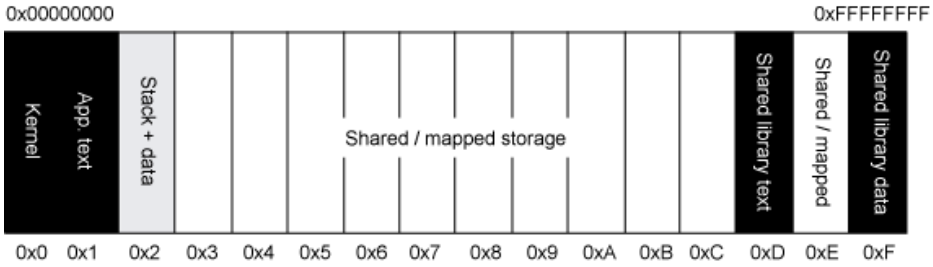


ネイティブ・プログラムのインスタンスは、それぞれ 1 つのプロセスとして実行されます。AIX でのプロセスとは、OS が管理するリソースに関する情報 (ファイルやソケットなどの情報)、仮想アドレス空間、そして 1 つ以上の実行スレッドからなります。

32-bit アドレスは 4GB のデータを参照するものの、4GB のアドレス空間全体がそのプログラム専用に与えられるわけではありません。他の OS (Windows や Linux など) の場合と同じく、アドレス空間は複数のセクションに分割されます。プログラムが使用できるのは、そのうちのいくつかのセクションだけで、後は OS が使用します。Windows や Linux と比べると、AIX のメモリー・モデルはより複雑なので、それだけ正確に調整することが可能です。

AIX の 32-bit メモリー・モデルは、16 X 256MB のセグメントに分割されて管理されます。図 2 に示すのは、デフォルトの32-bit AIX メモリー・モデルのレイアウトです。

図 2. デフォルトの AIX メモリー・モデル



それぞれのセグメントの用途は以下のとおりです。

- セグメント 0: AIX カーネル・データ (ユーザー・プログラムでは直接アクセスできません)
- セグメント 1: アプリケーション・テキスト (実行可能コード)
- セグメント 2: スレッド・スタックおよびネイティブ・ヒープ (malloc/free で管理される領域)
- セグメント 3 から C、および E: メモリーがマッピングされた領域 (ファイルを含む) および共有メモリー
- セグメント D: 共有ライブラリー・テキスト (実行可能コード)
- セグメント F: 共有ライブラリー・データ

ユーザー・プログラムが直接管理できるのは、16 あるセグメントのうち、12 のセグメントだけです。つまり 4GB のうちの 3GB に相当します。ここで最も重要な制約は、ネイティブ・ヒープとすべてのスレッド・スタックは、セグメント 2 に保持されるということです。そこで、AIX ではデータ要件が大きなプログラムに対応するためのラージ・メモリー・モデルを提供しています。

ラージ・メモリー・モデルでは、プログラマーやユーザーが実行可能プログラムをビルドするときにリンカー・オプションを指定するか、あるいはプログラムが開始される前に `LDR_CNTRL` 環境変数を設定することで、ネイティブ・ヒープとして使用する共有/マッピング・セグメントを追加することができます。実行時にラージ・メモリー・モデルを有効にするには、`LDR_CNTRL=MAXDATA=0xN0000000` を設定します。ここで、`N` は `1` から `8` までの値です。値がこの範囲内にない場合には、デフォルトのメモリー・モデルが使用されることになります。ラージ・メモリー・モデルではネイティブ・ヒープがセグメント 3 で始まり、セグメント 2 は最初から存在する (初期) スレッド・スタックのためだけに使用されます。

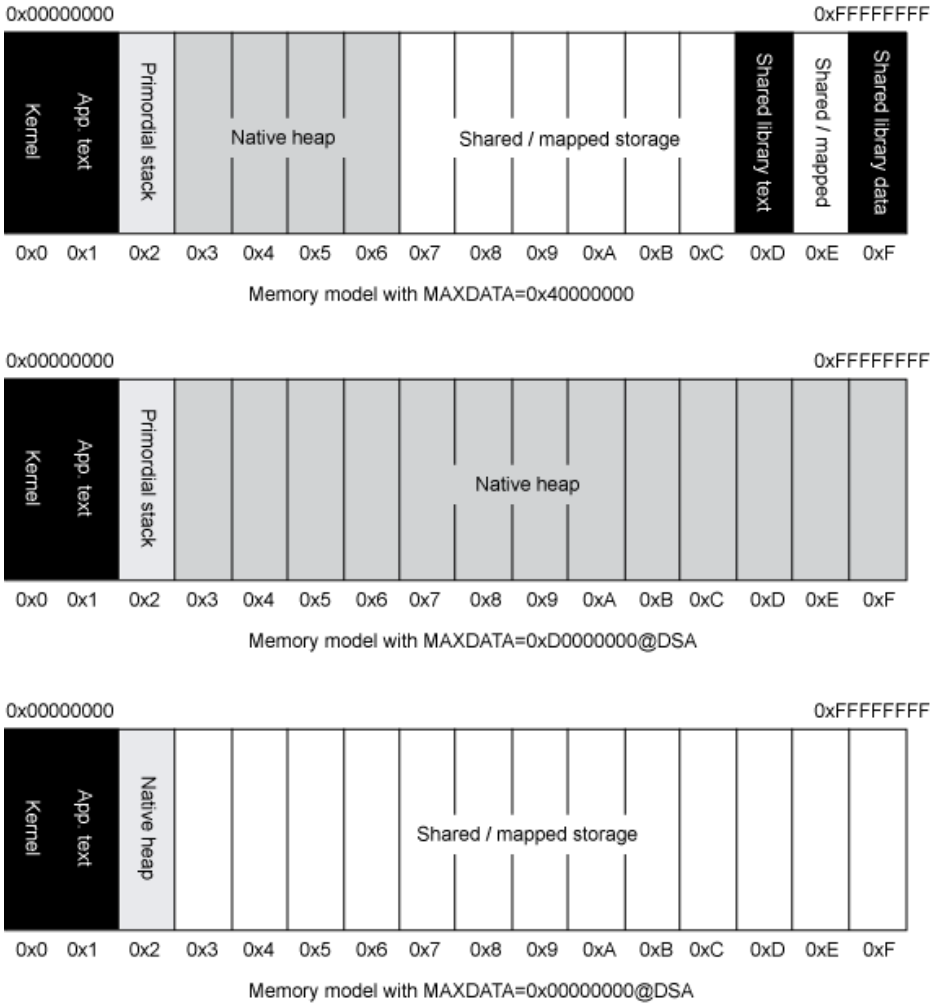
ラージ・メモリー・モデルを使用すると、セグメントは静的に割り当てられることになります。つまり、例えば 4 つのデータ・セグメント (ネイティブ・ヒープの 1GB に相当) を要求した後に、ネイティブ・ヒープのセグメントを 1 つ (256MB) しか割り当てなかったとすると、他の 3 つのデータ・セグメントがメモリー・マッピングに使用できなくなるということです。

2GB を超えるネイティブ・ヒープが必要な場合、AIX 5.1 以降では AIX の超ラージ・メモリー・モデルを使用することができます。超ラージ・メモリー・モデルをコンパイル時に実行可能プログラムに対して有効にするには、ラージ・メモリー・モデルと同じく、リンカー・オプションを指定するか、実行時に `LDR_CNTRL` 環境変数を使用します。実行時に超ラージ・メモリー・モデルを有効にする場合には、`LDR_CNTRL=MAXDATA=0xN0000000@DSA` を指定してください。ここで、`N` は `0` から `D` までの値 (AIX 5.2 以降の場合)、または `1` から `A` までの値 (AIX 5.1 を使用している場合) です。`N` の値が指定するのは、ネイティブ・ヒープに使用可能なセグメント数ですが、ラージ・メモリー・モデルとは異なり、これらのセグメントは必要に応じてメモリー・マッピングにも使用することができます。

IBM Java ランタイムは、`LDR_CNTRL` 環境変数で無効にされていない限り、超ラージ・メモリー・モデルを使用します。

`N` を `1` から `A` までの値に設定すると、皆さんが予想するとおり、セグメント 3 から C までがネイティブ・ストレージに使用されます。AIX 5.2 からは、`N` を `B` またはそれ以降に設定した場合、メモリー・レイアウトが変更され、セグメント D と F は共有ライブラリーには使用されず、ネイティブ・ストレージまたはメモリー・マッピングに使用できるようになります。`N` を `D` に設定すると、ネイティブ・ヒープのセグメント数は最大 13になります (3.25GB)。一方、`N` を `0` に設定すると、セグメント 3 から F までをメモリー・マッピングに使用できます。この場合、ネイティブ・ヒープが保持されるのはセグメント 2 となります。図 3 に、それぞれの AIX メモリー・モデルでのアドレス空間のレイアウトの違いを示します。

図 3. `MAXDATA` の値に応じた AIX メモリー・モデル



ネイティブ・メモリーのリークや過剰なネイティブ・メモリーの使用は、アドレス空間を使い尽くしたのか、あるいは物理メモリーが足りなくなっているのかによって、異なる問題を引き起こします。アドレス空間を使い尽くすことが起こるのは、一般には 32-bit プロセスの場合のみの話です。最大 4GB のアドレス空間をすべて割り当てるのは難しくはないからです。64-bit プロセスには数百、または数千ギガバイトのユーザー空間があり、それを使い切ろうとしても、なかなか

使い切れるものではありません。Java プロセスのアドレス空間を使い切ったとしたら、Java ランタイムには奇妙なシンプトンが現れ始めます (これについては、後で説明します)。実行中のシステムのプロセス・アドレス空間が物理メモリーの大きさを超える場合、メモリー・リークが発生したり、あるいはネイティブ・メモリーを使い過ぎたりすると、OS が余儀なく仮想アドレス空間の一部をスワップアウトすることになります。スワップアウトされたメモリー・アドレスにアクセスするには、そのメモリー・アドレスの中身をハード・ドライブからロードしなければならないため、物理メモリー内に常駐するアドレスから読み出す場合よりも時間がかかります。

RAM がバックアップする大量の仮想メモリーを物理メモリーでデータを保持できないほどに使おうとすると、システムに過剰な負担がかかります。つまり、システムの処理時間がほとんど、スワップ空間との間でのメモリーのコピーに費やされるということです。このような事態においてはコンピューターと個々のアプリケーションのパフォーマンスが劣化するため、ユーザーは必ず問題の存在に気付くことになります。JVM の Java ヒープがスワップアウトされると、ガーベッジ・コレクターのパフォーマンスが著しく劣化し、アプリケーションがハングしたような状態になってしまう可能性があります。複数の Java ランタイムを 1 台のマシンで同時に使用している場合には、すべての Java ヒープを収められるだけの物理メモリーの量が必要です。

Java ランタイムによるネイティブ・メモリーの使用について

Java ランタイムは、前のセクションで概説したハードウェアおよび OS による制約を受ける OS プロセスです。ランタイム環境が提供する機能は何かしらのユーザー・コードによって左右されるので、どのリソースがランタイム環境に必要となるのかは、どんな状況でも予測することができません。管理された Java 環境内で Java アプリケーションが取るあらゆるアクションが、その環境を提供するランタイムのリソース要件に影響する可能性があります。このセクションでは、Java アプリケーションはどのようにネイティブ・メモリーを使用するのか、そしてなぜ使用するのかについて説明します。

Java ヒープとガーベッジ・コレクション

Java ヒープは、オブジェクトが割り当てられるメモリー領域です。IBM Developer Kits for Java Standard Edition には 1 つの物理ヒープがありますが、IBM WebSphere Real Time などの専門化した Java ランタイムには複数のヒープがあります。ヒープは、IBM gencon ポリシーの Nursery 領域や Tenured 領域といったセクションに分割することができます。大抵の Java ヒープは、ネイティブ・メモリーの隣接スラブとして実装されます。

ヒープのサイズは、Java コマンドラインから `-Xmx` オプションと `-Xms` オプションを使って制御します (`mx` はヒープの最大サイズ、`ms` は初期サイズです)。論理ヒープ (アクティブに使用されているメモリーの領域) はヒープ上のオブジェクトの数とガーベッジ・コレクション (GC) の所要時間に応じて大きくなったり小さくなったりするものの、使用されるネイティブ・メモリーの量は常に一定であり、`-Xmx` の値、つまり最大ヒープ・サイズによって指定されます。メモリー・マネージャーはヒープがメモリーの隣接スラブであることに依存するため、ヒープを拡大する必要があっても、ネイティブ・メモリーを追加で割り当てることはできません。したがって、すべてのヒープ・メモリーは前もって予約しておく必要があります。

ネイティブ・メモリーを予約することは、割り当てることとは異なります。予約されたネイティブ・メモリーは、物理メモリーやその他のストレージでバックアップされません。アドレス空間のチャンクを予約しても物理リソースを使用することにはなりませんが、そのチャンクを他の目

的では使用できなくなります。使用されることのないメモリーを予約することによって発生するリークは、割り当てられたメモリーがリークすることと同じくらいに深刻な問題です。

AIX 上の IBM ガーベッジ・コレクターは、ヒープで使用されている領域が縮小するにつれてヒープのセクションをデコミット (そのセクションのバックアップ・ストレージを解放) することによって、物理メモリーの使用量を最小限にします。

ほとんどの Java アプリケーションでは、プロセスのアドレス空間を最も多く使用するのは Java ヒープであるため、Java ランチャーは Java ヒープ・サイズを基にして、アドレス空間をどのように構成するかを決定します。表 2 に、さまざまなヒープ·サイズに対して設定されるデフォルトのメモリー・モデル構成を記載します。デフォルトのメモリー・モデルは、Java ランチャーが起動する前に `LDR_CNTRL` 環境変数を手動で設定することによって変更することができます。Java ランタイムを組み込んでいる場合、あるいは独自のランチャーを作成している場合には、適切なリンカー・フラグを指定するか、またはランチャーを起動する前に `LDR_CNTRL` を設定して、メモリー・モデルを自分で構成することになります。

表 2. ヒープ・サイズごとのデフォルト `LDR_CNTRL` 設定

ヒープの範囲	LDR_CNTRL 設定	最大ネイティブ・ヒープ・サイズ	最大マッピング・スペース (ネイティブ・ヒープを占有しない場合)
-Xmx0M ~ -Xmx2304M	MAXDATA=0xA0000000@DSA	2.5GB	512MB
-Xmx2304M ~ -Xmx3072M	MAXDATA=0xB0000000@DSA	2.75GB	512MB
> -Xmx2304M	MAXDATA=0x0@DSA	256MB	3.25GB

JIT (Just-In-Time) コンパイラー

JIT コンパイラーは実行時に Java バイトコードを最適なネイティブ・バイナリー・コードにコンパイルします。このコンパイルによって Java ランタイムの実行時の処理速度が大幅に改善され、Java アプリケーションをネイティブ・コードに匹敵するほどの速度で実行できるようになります。

バイトコードのコンパイルにはネイティブ・メモリーを使用しますが (これは、`gcc` などの静的コンパイラーは、実行するのにメモリーが必要なのと同様です)、JIT (実行可能コード) からの出力もネイティブ・メモリーに格納しなければなりません。そのため、JIT でコンパイルされたメソッドが多数含まれる Java アプリケーションは、規模の小さなアプリケーションよりも多くのネイティブ・メモリーを使用します。

クラスとクラスローダー

Java アプリケーションは、オブジェクト構造とメソッド・ロジックを定義するクラスで構成されます。また、Java アプリケーションでは、Java ランタイム・クラス・ライブラリー (`java.lang.String` など) のクラスを使用しますが、これにはサード・パーティーのライブラリーが使用される場合もあります。これらのクラスは、使用している間はメモリー内に存在していなければなりません。

Java 5 以降の IBM 実装では、クラスローダーごとにネイティブ・メモリーのスラブを割り当て、そこにクラス・データを格納するようになっています。Java 5 以降の共有クラス技術は共有メモリーの領域を読み取り専用 (つまり共有可能な) クラス・データが格納されるアドレス空間にマッ

ピングします。これにより、複数の JVM が同じマシンで稼働するときには、クラス・データを格納するために必要な物理メモリーの量が削減されます。また、共有クラスによって JVM 起動時間も短縮されます。

共有クラス・システムは、固定サイズの共有メモリー領域をアドレス空間にマッピングします。共有クラス・キャッシュが完全に占有されているわけではなく、キャッシュに (他の JVM によってロードされた) 現在使用されていないクラスが含まれている場合もあるため、共有クラスを使用すると、共有クラスを使用しないで実行した場合よりも多くのアドレス空間が占有されるはずです。注意する点として、共有クラスがクラスローダーのアンロードを阻止することはありませんが、クラス・データのサブセットはクラス・キャッシュ内にそのまま残ります。共有クラスについての詳細は、「参考文献」を参照してください。

ロードするクラスが増えれば増えるほど、多くのネイティブ・メモリーが使用されます。さらに、それぞれのクラスローダーにはネイティブ・メモリーのオーバーヘッドもあります。したがって、それぞれ 1 つのクラスをロードする多数のクラスローダーを使用すると、1 つのクラスローダーで多数のクラスをロードするよりも、ネイティブ・メモリーの使用量は増えることになります。メモリーに入れる必要があるのはアプリケーションのクラスだけではないことを覚えておいてください。フレームワーク、アプリケーション・サーバー、サード・パーティーのライブラリー、そして Java ランタイムにも、オンデマンドでロードされてスペースを占有するクラスが含まれています。

Java ランタイムはクラスをアンロードしてスペースを再利用することができますが、それには厳しい条件が適用されます。Java ランタイムが単一のクラスをアンロードすることはできません。クラスではなく、クラスローダーがアンロードされるため、そのクラスローダーでロードしたすべてのクラスがアンロードされることになります。クラスローダーをアンロードできるのは、以下の条件が当てはまる場合のみです。

- Java ヒープに、そのクラスロードを表す `java.lang.ClassLoader` オブジェクトへの参照が含まれていないこと。
- Java ヒープに、そのクラスローダーがロードしたクラスを表す `java.lang.Class` オブジェクトへの参照が含まれていないこと。
- そのクラスローダーがロードしたクラスのオブジェクトが、Java ヒープで有効 (参照されている状態) になっていないこと。

注目に値する点は、Java ランタイムがすべての Java アプリケーション用に作成する 3 つのデフォルト・クラスローダー (ブートストラップ、拡張、およびアプリケーション) が上記の基準を満たすことは決してないということです。そのため、どのシステム・クラス (`java.lang.String` など) にしても、アプリケーション・クラスローダーによってロードされたアプリケーション・クラスにしても、解放することはできません。

クラスローダーの収集が可能になったとしても、ランタイムがクラスローダーを収集するのは、GC サイクルの一環としてのみです。IBM gencon GC ポリシー (コマンドライン引数 `-Xgcpolicy:gencon` を指定すると有効になります) では、クラスローダーを主要な (Tenured 領域の) コレクションのときにのみアンロードします。アプリケーションが gencon ポリシーを実行中で、多数のクラスローダーを作成および解放している場合には、Tenured 領域のコレクションが実行されてから、次にこのコレクションが実行されるまでの間に、収集可能なクラスローダーが大量

のネイティブ・メモリーを保持した状態になっていることがわかるはずです。さまざまな IBM GC ポリシーについての詳細は、「参考文献」を参照してください。

気付かないうちに、実行時にクラスが生成されているという事態もあり得ます。多くの JEE アプリケーションでは JSP (JavaServer Pages) 技術を使って Web ページを生成します。JSP を使用すると、実行された .jsp ページごとにクラスが生成されますが、このクラスはそれをロードしたクラスローダーの存続期間 (通常は、Web アプリケーションの存続期間) にわたって存続することになります。

Java リフレクションによってクラスが生成されることもよくあります。`java.lang.reflect` API の使用時には、Java ランタイムがオブジェクト (`java.lang.reflect.Field` など) のリフレクションを実行するメソッドをリフレクションの実行対象のオブジェクトまたはクラスに接続しなければなりません。この「アクセサー」では、セットアップがほとんど必要ない代わりに実行速度が遅い JNI (Java Native Interface) を使用することも、実行時にリフレクション先のオブジェクト・タイプごとに動的にクラスを作成することもできます。後者の方式をセットアップするには時間がかかりますが、実行速度には優れています。そのため、特定のクラスに頻繁にリフレクションを実行するようなアプリケーションには最適です。

Java ランタイムは、クラスに対してリフレクションを実行する最初の数回は JNI メソッドを使用しますが、このアクセサーは何度か使われているうちにバイトコード・アクセサーへと拡張されます。バイトコード・アクセサーでは、クラスを作成し、そのクラスを新しいクラスローダーでロードする必要が出てきます。そのためリフレクションを多数実行すると、アクセサー・クラスとクラスローダーが多数作成される結果となります。そしてリフレクションを実行しているオブジェクトへの参照を保持することによって、これらのクラスが有効な状態のままになり、スペースを占有し続けます。バイトコード・アクセサーの作成にはかなりの時間がかかるため、Java ランタイムはこれらのアクセサーを後で使えるようにキャッシュに入れます。一部のアプリケーションとフレームワークはリフレクションの対象オブジェクトもキャッシュに入れるため、ネイティブ・メモリーのフットプリントが増加します。

リフレクション・アクセサーの振る舞いは、システム・プロパティーで制御することができます。IBM Developer Kit for Java 5.0 の場合、デフォルトの拡張しきい値 (バイトコード・アクセサーに拡張するまで JNI アクセサーを使用する回数) は 15 です。この値を変更するには、`sun.reflect.inflationThreshold` システム・プロパティーを設定します。このプロパティーは、Java コマンドラインで `-Dsun.reflect.inflationThreshold=N` を指定すれば、設定することができます。`inflationThreshold` を `0` またはそれより小さい値に設定すると、アクセサーは拡張されません。この設定は、アプリケーションが多数の `sun.reflect.DelegatingClassloader` (バイトコード・アクセサーをロードするために使われるクラスローダー) を作成していることがわかった場合に役立ちます。

リフレクション・アクセサーに影響する (かなり誤解されている) 設定は、もう 1 つあります。`-Dsun.reflect.noInflation=true` を指定すると、リフレクション・アクセサーはまったく拡張されなくなりますが、代わりに、バイトコード・アクセサーが何に対しても使用されます。そのため `-Dsun.reflect.noInflation=true` を使用すると、作成されるリフレクション・クラスローダーが増え、リフレクション・クラスローダーによって使用されるアドレス空間の量も増えることになります。

Java 5 以降では、javacore ダンプを取得することによって、クラスおよび JIT コードに使用されているメモリーの量を測定することができます。javacore は、ダンプ取得時の Java ランタイムの内部状態のサマリーを記載するプレーン・テキスト・ファイルです。このサマリーには、割り当てられているネイティブ・メモリー・セグメントに関する情報が含まれています。IBM Developer Kit for Java 5 および 6 の新しいバージョンでは javacore の中にメモリーの使用量が要約されますが、それよりも古いバージョン (Java 5 SR10 および Java 6 SR3 より前のバージョン) の場合には、この記事のサンプル・コード・パッケージに含まれる Perl スクリプトを使って、データを照合して表示することができます (「ダウンロード」を参照)。このスクリプトを実行するには、AIX やその他のプラットフォームに対応した Perl インタープリターが必要です。詳細は「参考文献」を参照してください。

javacore は、`OutOfMemoryError` がスローされると生成されますが (この例外が発生するのは、おそらくアドレス空間が不足しているということです)、`SIGQUIT` を Java プロセスに送信することによって javacore をトリガーすることもできます (`kill -3 <pid>`)。メモリー・セグメントの使用量を要約するには、以下のコードを実行してください。

```
perl get_memory_use.pl javacore.<date>.<time>.<pid>.txt
```

スクリプトの出力は、以下のようになります。

```
perl get_memory_use.pl javacore.20080111.081905.1311.txt
Segment Usage           Reserved Bytes
Class Memory            281767824
Internal Memory         25763872
JIT Code Cache          67108864
JIT Data Cache          33554432
Object Memory           536870912
```

JNI

JNI ではネイティブ・コード (C や C++ などのネイティブ言語で作成されたアプリケーション) で Java メソッドを呼び出すことも、その逆を行うことも可能です。Java ランタイム自体は、ファイルやネットワーク I/O などのクラス・ライブラリー関数を実装する上で、JNI コードに大きく依存します。JNI アプリケーションが Java ランタイムのネイティブ・メモリーのフットプリントを増加させる原因には、以下の 3 つがあります。

- JNI アプリケーションのネイティブ・コードは、プロセスのアドレス空間にロードされる共有ライブラリーまたは実行可能コードにコンパイルされます。大きなネイティブ・アプリケーションは、単にロードするだけで、プロセスのアドレス空間の大きなチャンクを占有します。
- ネイティブ・コードは Java ランタイムとアドレス空間を共有しなければなりません。ネイティブ・メモリーの割り当て、あるいはネイティブ・コードが行うメモリー・マッピングは、Java ランタイムからメモリーを奪うことになります。
- 特定の JNI 関数は、通常の操作の一環としてネイティブ・メモリーを使用することができます。`GetTypeArrayElements` 関数と `GetTypeArrayRegion` 関数は、Java ヒープのデータをネイティブ・メモリー・バッファーにコピーしてネイティブ・コードから操作可能にすることができます。コピーが行われるかどうかは、ランタイム実装に依存します。IBM Developer Kit for Java 5.0 以降では、ネイティブ・コピーを行います。この変更は、オブジェクトをヒープ

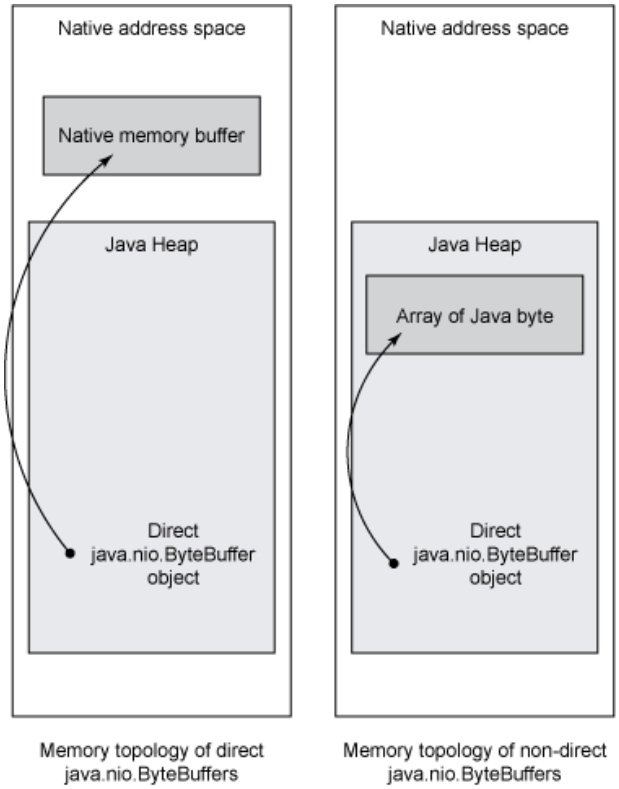
に固定しないようにするために行われました (JVM 外部のコードがオブジェクトを参照していることから、オブジェクトをメモリー内に固定しなければなりませんでした)。これは、Java ヒープをフラグメント化できないことを意味する一方で (1.4.2 では可能でした)、ランタイムのネイティブ・メモリーのフットプリントが増加する結果となりました。コピーを実行することで、大量の Java ヒープ・データにアクセスすると、同じく大量のネイティブ・ヒープを使用する可能性があります。

NIO

Java 1.4 で追加された新規 I/O (NIO) クラスによって、チャネルおよびバッファー・ベースで入出力処理 (I/O) を実行する新しい方法が導入されました。Java ヒープのメモリーでバックアップされる I/O バッファーだけでなく、NIO は Java ヒープではなくネイティブ・メモリーでバックアップされるダイレクト `ByteBuffer` (`java.nio.ByteBuffer.allocateDirect()` メソッドで割り当てられるバッファー) のサポートを追加します。ダイレクト `ByteBuffer` は、I/O を実行するためにネイティブ OS ライブラリー関数に直接渡すことができます。つまり、Java ヒープとネイティブ・ヒープとの間でのコピーが必要なくなるため、シナリオによっては処理速度が大幅に向上します。

ダイレクト `ByteBuffer` が存在する場所については混乱しがちですが、アプリケーションは I/O 操作を調整するために引き続き Java ヒープ上のオブジェクトを使用する一方、データを保持するバッファーはネイティブ・メモリーに保持されます。Java ヒープ上のオブジェクトには、ネイティブ・ヒープ・バッファーへの参照しか含まれません。非ダイレクト `ByteBuffer` は、そのデータを Java ヒープ上の `byte[]` 配列に保持します。図 4 に、ダイレクト `ByteBuffer` オブジェクトと非ダイレクト `ByteBuffer` オブジェクトの違いを示します。

図 4. ダイレクトおよび非ダイレクト `java.nio.ByteBuffer` のメモリー・トポロジー



Memory topology of direct java.nio.ByteBuffers

Memory topology of non-direct java.nio.ByteBuffers

ダイレクト `ByteBuffer` オブジェクトはそのネイティブ・バッファーを自動的にクリーンアップしますが、この自動クリーンアップを実行できるのは Java ヒープ GC の際だけです。つまり、ネイティブ・ヒープの状況に応じて自動的にクリーンアップすることはできません。GC が発生するのは、Java ヒープがヒープ割り当て要求に対応できなくなるまでフルになった場合、あるいは Java アプリケーションが明示的に GC を要求した場合のみです (GC の明示的要求はパフォーマンス上の問題の原因となるため推奨されません)。

ネイティブ・ヒープがフルになり、1 つ以上のダイレクト `ByteBuffer` に GC を実行できるようになる (そしてネイティブ・ヒープ上のスペースを空けるために解放できる) という異常な場合も考えられますが、Java ヒープはほとんどの場合、空なので GC は発生しません。

スレッド

アプリケーション内のすべてのスレッドには、そのスタック (ローカル変数を保持し、関数呼び出しの際に状態を維持するためのメモリー領域) を保持するためのメモリーが必要です。実装によっては、Java スレッドが別個のネイティブ・スタックおよび Java スタックを持つこともあります。スタック空間の他、各スレッドにはスレッドのローカル・ストレージ用、そして内部データ構造用のネイティブ・メモリーも必要です。

スタックのサイズは Java 実装によってもアーキテクチャーによっても異なりますが、一部の実装では Java スレッドのスタック・サイズを指定することができます。標準的な値は、256KB から 756KB の間です。

スレッドごとに使用するメモリーはかなり少ないとは言え、数百のスレッドを使用するアプリケーションとなると、スレッド・スタックの合計メモリー使用量はかなりのものになるはずです。スレッドの実行に使用できるプロセッサーが足りなくなるほど大量のスレッドを持つアプリケーションを実行すると、非効率であるばかりか、パフォーマンスが劣化し、メモリー使用量が増える結果となります。

ネイティブ・メモリー不足を判断する方法

Java ランタイムは、ネイティブ・ヒープの不足とはかなり異なる方法で Java ヒープの不足に対処しますが、ネイティブ・ヒープ不足と Java ヒープ不足は同じようなシンプトンを見せることがあります。Java アプリケーションは何をするにもオブジェクトを割り当てるため、Java ヒープが枯渇した場合、Java アプリケーションが機能するのは極めて困難になります。Java ヒープがフルであることを示す GC パフォーマンスの劣化と `OutOfMemoryError` は、Java ヒープがフルになると同時に発生します。

それとは対照的に、Java ランタイムが起動してアプリケーションが安定した状態になると、ネイティブ・ヒープが完全に枯渇してもアプリケーションは機能し続けます。この場合は、必ずしも奇妙な振る舞いを見せるわけではありません。ネイティブ・メモリーの割り当てが必要なアクションは、Java ヒープの割り当てが必要なアクションに比べると遙かに少ないからです。ネイティブ・メモリーを必要とするアクションは JVM 実装によってさまざまに異なりますが、いくつか一般的な例を挙げると、スレッドの開始、クラスのロード、ある種のネットワークおよびファイル I/O の実行があります。

また、ネイティブ・メモリー不足の振る舞いは、Java ヒープのメモリー不足と比べると一貫性を欠いています。これは、ネイティブ・ヒープの割り当てには単一の制御ポイントがないためです。すべての Java ヒープ割り当ては Java メモリー管理システムによって制御される一方、ネイティブ・コードは、それが JVM 内、Java クラス・ライブラリー内、またはアプリケーション・コード内のいずれにあるとしても、ネイティブ・メモリーの割り当てを試行することができます。そしてネイティブ・メモリーの割り当てに失敗したコードは、その設計者の意図とは無関係に失敗の処理を行います。例えば JNI インターフェースを介して `OutOfMemoryError` をスローする、画面にメッセージを出力する、あるいは何の反応も起こさずにただ失敗して後で再試行するなどです。

振る舞いを予測できないということは、ネイティブ・メモリーの枯渇は一通りの単純な方法では識別できないということです。OS からのデータと Java ランタイムによる診断の裏付けが必要になります。

ネイティブ・メモリー不足の例

ネイティブ・メモリーの枯渇が Java ランタイムにどのように影響するかを確かめられるよう、この記事のサンプル・コード (「ダウンロード」を参照) にはネイティブ・ヒープを異なる方法で枯渇させる Java プログラムがいくつか含まれています。これらの例では、C で作成されたネイティブ・ライブラリーを使用してネイティブ・プロセス空間を使い切り、その上でネイティブ・メモリーを使用するアクションの実行を試みます。サンプル・コードはすでにビルドされた状態で提供されていますが、サンプル・パッケージの最上位ディレクトリーにある README.html ファイルでコードのコンパイル手順を説明しています。

`com.ibm.jtc.demos.NativeMemoryGlutton` クラスが提供する `gobbleMemory()` メソッドは、ほとんどすべてのメモリーが使い果たされるまで、`malloc` をループで呼び出します。このメソッドはその役割を果たすと、以下のような標準的なエラー・フォーマットで割り当て済みのバイト数を出力します。

```
Allocated 1953546736 bytes of native memory before running out
```

それぞれのデモの出力は、32-bit AIX で実行中の IBM Java ランタイムについて取得されたものです。サンプル・プログラムのバイナリーは、サンプル・パック (「ダウンロード」を参照) の中に提供されています。

使用した IBM Java ランタイムのバージョンは以下のとおりです。

```
java version "1.5.0"
Java(TM) 2 Runtime Environment, Standard Edition (build pap32devifx-20080811c (SR8a))
IBM J9 VM (build 2.3, J2RE 1.5.0 IBM J9 2.3 AIX ppc-32
j9vmap3223ifx-20080811 (JIT enabled)
J9VM - 20080809_21892_bHdSMr
JIT  - 20080620_1845_r8
GC   - 200806_19)
JCL  - 20080811b
```

ネイティブ・メモリーの不足時にスレッドを開始しようとした場合

`com.ibm.jtc.demos.StartingAThreadUnderNativeStarvation` クラスは、プロセス・アドレス空間が枯渇した時点でスレッドを開始しようとします。多数のアプリケーションがそれぞれの存続期間全体にわたってスレッドを開始することから、Java プロセスのメモリー不足を発見するには、これが一般的な方法です。

`StartingAThreadUnderNativeStarvation` による出力は以下のとおりです。

```
$ ./run_thread_demo_linux_aix_32.sh
Allocated 2652372992 bytes of native memory before running out
JVMDUMP006I Processing Dump Event "systhrow", detail "java/lang/OutOfMemoryError
" - Please Wait.
JVMDUMP007I JVM Requesting Snap Dump using '/u2/andhall/aix_samples_pack/OutOfNa
tiveBehaviour/Snap.20081207.105130.487430.0001.trc'
JVMDUMP010I Snap Dump written to /u2/andhall/aix_samples_pack/OutOfNativeBehavio
ur/Snap.20081207.105130.487430.0001.trc
JVMDUMP007I JVM Requesting Heap Dump using '/u2/andhall/aix_samples_pack/OutOfNa
tiveBehaviour/heapdump.20081207.105130.487430.0002.phd'
JVMDUMP010I Heap Dump written to /u2/andhall/aix_samples_pack/OutOfNativeBehavio
ur/heapdump.20081207.105130.487430.0002.phd
JVMDUMP007I JVM Requesting Java Dump using '/u2/andhall/aix_samples_pack/OutOfNa
tiveBehaviour/javacore.20081207.105130.487430.0003.txt'
JVMDUMP010I Java Dump written to /u2/andhall/aix_samples_pack/OutOfNativeBehavio
ur/javacore.20081207.105130.487430.0003.txt
JVMDUMP013I Processed Dump Event "systhrow", detail "java/lang/OutOfMemoryError"
.
java.lang.OutOfMemoryError: ZIP006:OutOfMemoryError, ENOMEM error in ZipFile.open
        at java.util.zip.ZipFile.open(Native Method)
        at java.util.zip.ZipFile.<init>(ZipFile.java:238)
        at java.util.jar.JarFile.<init>(JarFile.java:169)
        at java.util.jar.JarFile.<init>(JarFile.java:107)
        at com.ibm.oti.vm.AbstractClassLoader.fillCache(AbstractClassLoader.java
:69)
        at com.ibm.oti.vm.AbstractClassLoader.getResourceAsStream(AbstractClassL
oader.java:113)
```

```
        at java.util.ResourceBundle$1.run(ResourceBundle.java:1111)
        at java.security.AccessController.doPrivileged(AccessController.java:197
)
        at java.util.ResourceBundle.loadBundle(ResourceBundle.java:1107)
        at java.util.ResourceBundle.findBundle(ResourceBundle.java:952)
        at java.util.ResourceBundle.getBundleImpl(ResourceBundle.java:789)
        at java.util.ResourceBundle.getBundle(ResourceBundle.java:726)
        at com.ibm.oti.vm.MsgHelp.setLocale(MsgHelp.java:103)
        at com.ibm.oti.util.Msg$1.run(Msg.java:44)
        at java.security.AccessController.doPrivileged(AccessController.java:197
)
        at com.ibm.oti.util.Msg.<clinit>(Msg.java:41)
        at java.lang.J9VMInternals.initializeImpl(Native Method)
        at java.lang.J9VMInternals.initialize(J9VMInternals.java:194)
        at java.lang.ThreadGroup.uncaughtException(ThreadGroup.java:764)
        at java.lang.ThreadGroup.uncaughtException(ThreadGroup.java:758)
        at java.lang.Thread.uncaughtException(Thread.java:1315)
K0319java.lang.OutOfMemoryError: Failed to create a thread: retVal -1073741830,
errno 11
        at java.lang.Thread.startImpl(Native Method)
        at java.lang.Thread.start(Thread.java:979)
        at com.ibm.jtc.demos.StartingAThreadUnderNativeStarvation.main(StartingA
ThreadUnderNativeStarvation.java:33)
```

`java.lang.Thread.start()` の呼び出しによって、新しい OS スレッドへのメモリー割り当てが試行されます。この試行は失敗し、`OutOfMemoryError` がスローされる結果となります。`JVMDUMP` の行はユーザーに対し、Java ランタイムの標準 `OutOfMemoryError` デバッグ・データが生成されたことを通知しています。

最初の `OutOfMemoryError` を処理しようとしたことによって、2 番目の `:OutOfMemoryError, ENOMEM error in ZipFile.open` が発生しました。このように、ネイティブ・プロセス・メモリーが使い果たされると複数の `OutOfMemoryError` が発生するのは一般的なことです。なぜなら、デフォルトの `OutOfMemoryError` 処理ルーチンのいくつかが、ネイティブ・メモリーを割り当てなければならない場合があるからです。参考になる情報には聞こえないかもしれませんが、Java アプリケーションがスローするほとんどの `OutOfMemoryError` は Java ヒープ・メモリーの不足が原因であり、Java ヒープ・メモリーが不足しても、ランタイムがネイティブ・ストレージを割り当てる妨げにはなりません。このシナリオでスローされる `OutOfMemoryError` を Java ヒープの枯渇によってスローされる `OutOfMemoryError` と区別する唯一の手掛かりはメッセージです。

ネイティブ・メモリーの不足時にダイレクト `ByteBuffer` を割り当てようとした場合

`com.ibm.jtc.demos.DirectByteBufferUnderNativeStarvation` クラスは、アドレス空間が枯渇したときにダイレクト (つまり、ネイティブに操作される) `java.nio.ByteBuffer` オブジェクトの割り当てを試行します。その結果生成される出力は以下のとおりです。

```
$ ./run_directbytebuffer_demo_aix_32.sh
Allocated 2652372992 bytes of native memory before running out
JVMDUMP006I Processing Dump Event "systhrow", detail "java/lang/OutOfMemoryError
" - Please Wait.
JVMDUMP007I JVM Requesting Snap Dump using '/u2/andhall/aix_samples_pack/OutOfNa
tiveBehaviour/Snap.20081207.105307.610498.0001.trc'
JVMDUMP010I Snap Dump written to /u2/andhall/aix_samples_pack/OutOfNativeBehavio
ur/Snap.20081207.105307.610498.0001.trc
JVMDUMP007I JVM Requesting Heap Dump using '/u2/andhall/aix_samples_pack/OutOfNa
tiveBehaviour/heapdump.20081207.105307.610498.0002.phd'
JVMDUMP010I Heap Dump written to /u2/andhall/aix_samples_pack/OutOfNativeBehavio
```

```
ur/heapdump.20081207.105307.610498.0002.phd
JVMDUMP007I JVM Requesting Java Dump using '/u2/andhall/aix_samples_pack/OutOfNa
tiveBehaviour/javacore.20081207.105307.610498.0003.txt'
JVMDUMP010I Java Dump written to /u2/andhall/aix_samples_pack/OutOfNativeBehavio
ur/javacore.20081207.105307.610498.0003.txt
JVMDUMP013I Processed Dump Event "systhrow", detail "java/lang/OutOfMemoryError"
.
JVMDUMP006I Processing Dump Event "systhrow", detail "java/lang/OutOfMemoryError
" - Please Wait.
JVMDUMP007I JVM Requesting Snap Dump using '/u2/andhall/aix_samples_pack/OutOfNa
tiveBehaviour/Snap.20081207.105308.610498.0004.trc'
JVMDUMP010I Snap Dump written to /u2/andhall/aix_samples_pack/OutOfNativeBehavio
ur/Snap.20081207.105308.610498.0004.trc
JVMDUMP007I JVM Requesting Heap Dump using '/u2/andhall/aix_samples_pack/OutOfNa
tiveBehaviour/heapdump.20081207.105308.610498.0005.phd'
JVMDUMP010I Heap Dump written to /u2/andhall/aix_samples_pack/OutOfNativeBehavio
ur/heapdump.20081207.105308.610498.0005.phd
JVMDUMP007I JVM Requesting Java Dump using '/u2/andhall/aix_samples_pack/OutOfNa
tiveBehaviour/javacore.20081207.105308.610498.0006.txt'
UTE430: can't allocate buffer
UTE437: Unable to load formatStrings for j9mm
UTE430: can't allocate buffer
UTE437: Unable to load formatStrings for j9mm
UTE430: can't allocate buffer
UTE437: Unable to load formatStrings for j9mm
UTE430: can't allocate buffer
UTE437: Unable to load formatStrings for j9mm
UTE430: can't allocate buffer
UTE437: Unable to load formatStrings for j9mm
JVMDUMP013I Processed Dump Event "systhrow", detail "java/lang/OutOfMemoryError"
.
java.lang.OutOfMemoryError: ZIP006:OutOfMemoryError, ENOMEM error in ZipFile.ope
n
        at java.util.zip.ZipFile.open(Native Method)
        at java.util.zip.ZipFile.<init>(ZipFile.java:238)
        at java.util.jar.JarFile.<init>(JarFile.java:169)
        at java.util.jar.JarFile.<init>(JarFile.java:107)
        at com.ibm.oti.vm.AbstractClassLoader.fillCache(AbstractClassLoader.java
:69)
        at com.ibm.oti.vm.AbstractClassLoader.getResourceAsStream(AbstractClassL
oader.java:113)
        at java.util.ResourceBundle$1.run(ResourceBundle.java:1111)
        at java.security.AccessController.doPrivileged(AccessController.java:197
)
        at java.util.ResourceBundle.loadBundle(ResourceBundle.java:1107)
        at java.util.ResourceBundle.findBundle(ResourceBundle.java:952)
        at java.util.ResourceBundle.getBundleImpl(ResourceBundle.java:789)
        at java.util.ResourceBundle.getBundle(ResourceBundle.java:726)
        at com.ibm.oti.vm.MsgHelp.setLocale(MsgHelp.java:103)
        at com.ibm.oti.util.Msg$1.run(Msg.java:44)
        at java.security.AccessController.doPrivileged(AccessController.java:197)
        at com.ibm.oti.util.Msg.<clinit>(Msg.java:41)
        at java.lang.J9VMInternals.initializeImpl(Native Method)
        at java.lang.J9VMInternals.initialize(J9VMInternals.java:194)
        at java.lang.ThreadGroup.uncaughtException(ThreadGroup.java:764)
        at java.lang.ThreadGroup.uncaughtException(ThreadGroup.java:758)
        at java.lang.Thread.uncaughtException(Thread.java:1315)
K0319java.lang.OutOfMemoryError: Unable to allocate 1048576 bytes of
direct memory after 5 retries
        at java.nio.DirectByteBuffer.<init>(DirectByteBuffer.java:197)
        at java.nio.ByteBuffer.allocateDirect(ByteBuffer.java:303)
        at com.ibm.jtc.demos.DirectByteBufferUnderNativeStarvation.main(
DirectByteBufferUnderNativeStarvation.java:27)
Caused by: java.lang.OutOfMemoryError
        at sun.misc.Unsafe.allocateMemory(Native Method)
        at java.nio.DirectByteBuffer.<init>(DirectByteBuffer.java:184)
        ... 2 more
```

このシナリオから、OutOfMemoryError がスローされることによって多数の JVMDUMP 情報メッセージが生成されることがわかるはずです。Java トレース・エンジンが生成するいくつかの UTE エラー・メッセージは、ネイティブ・バッファーを割り当てられないことをレポートします。トレース・エンジンはデフォルトで有効に設定されてアクティブになっていることから、これらの UTE エラー・メッセージは、ネイティブ・メモリーが不足している場合の一般的なシンプトンです。最後に 2 つの OutOfMemoryError が出力されていますが、これは zip ライブラリーでの派生的な割り当ての失敗と java.nio.DirectByteBuffer による独自のエラーです。

デバッグ手順とその手法

java.lang.OutOfMemoryError や、メモリー不足に関するエラー・メッセージが発生したときに最初に行うべきことは、どの類のメモリーが枯渇しているかを判断することです。その最も簡単な方法として、まずは Java ヒープがフルであるかどうかを確認してください。Java ヒープが OutOfMemory 発生の原因でなかったとしたら、今度はネイティブ・ヒープの使用量を分析します。

IBM 製品のデバッグ

この記事でのガイドラインは一般的なデバッグの原則であり、ネイティブ・メモリー不足のシナリオを理解し、それぞれに固有なネイティブ・メモリーの問題をデバッグする際の参考となることを目的としています。IBM 製品で発生した問題を報告する場合には、必ずその製品に対応する MustGather 資料に従って、サポート・チームに必要なデータを所定のフォーマットで集めてください (「参考文献」を参照)。ISA (IBM Support Assistant) ワークベンチに統合されたツール、IBM Guided Activity Assistant には、Developer Kit for Java を含め、IBM 製品での問題をデバッグするための最新ワークフローが用意されています。

Java ヒープを確認する

Java ヒープの使用状況を確認するには、OutOfMemoryError がスローされたときに生成された javacore ファイルを調べるという方法、または詳細 GC データを使用するという方法の 2 つがあります。javacore ファイルは通常、Java プロセスの作業ディレクトリー内に生成され、javacore.<date>.<time>.<pid>.txt という形式の名前が付けられています。このファイルをテキスト・エディターで開くと、以下のようなセクションが見つかるはずです。

```
0SECTION       MEMINFO subcomponent dump routine
NULL           ================================
1STHEAPFREE    Bytes of Heap Space Free: 416760
1STHEAPALLOC   Bytes of Heap Space Allocated: 1344800
```

このセクションが示しているのは、javacore が生成されたときに解放された Java ヒープの量です。値は 16 進形式であることに注意してください。ヒープの割り当てが失敗したことが原因で OutOfMemoryError がスローされた場合には、GC トレース・セクションの内容は以下のようになります。

```
1STGCHTYPE     GC History
3STHSTTYPE     09:59:01:632262775 GMT j9mm.80 -   J9AllocateObject() returning NULL!
32 bytes requested for object of class 00147F80
```

J9AllocateObject() returning NULL! は、オブジェクト割り当てルーチンが正常に完了しなかったために、OutOfMemoryError がスローされるという意味です。

ガーベッジ・コレクターがあまりにも頻繁に実行されていることが原因で `OutOfMemoryError` がスローされる場合もあります (これは、ヒープがフルであるため、Java アプリケーションの実行がほとんど、あるいはまったく進んでいない証拠です)。この場合には、Heap Space Free が非常に小さな値になっているはずです。また、GC の履歴には以下のメッセージのいずれかが示されます。

```
1STGCHTYPE     GC History
3STHSTTYPE     09:59:01:632262775 GMT j9mm.83 -     Forcing J9AllocateObject()
to fail due to excessive GC
```

```
1STGCHTYPE     GC History
3STHSTTYPE     09:59:01:632262775 GMT j9mm.84 -     Forcing
J9AllocateIndexableObject() to fail due to excessive GC
```

`-verbose:gc` コマンドライン・オプションは、ヒープ占有率をはじめとする GC 統計が含まれるトレース・データを生成します。この情報を IBM Monitoring and Diagnostic Tools for Java – GCMV (Garbage Collection and Memory Visualizer) ツールでグラフに表示すると、Java ヒープが大きくなっているかどうかがわかります。`verbose:gc` によるデータを収集してグラフ化する方法について説明している記事へのリンクは、「参考文献」を参照してください。

ネイティブ・ヒープの使用量を測定する

メモリー不足の原因が Java ヒープの枯渇ではないと判断した場合、次のステップとなるのはネイティブ・メモリー使用量のプロファイルを作成することです。

AIX プロセスの調整について十分理解している場合には、お望みのツールチェーンを使用してネイティブ・プロセス・サイズをモニターしてください。1 つの選択肢として、IBM Monitoring and Diagnostic Tools for Java – GCMV (Garbage Collection and Memory Visualizer) ツールを使用することができます。

GCMV は元々、詳細 GC ログをグラフ化し、ユーザーがガーベッジ・コレクターを調整するときに Java ヒープ使用量とパフォーマンスの変化を確認できるようにするために作成されたツールです。その後、GCMV は拡張され、Linux および AIX のネイティブ・メモリーのログを始めとする他のデータ・ソースもグラフ化できるようになりました。GCMV は ISA にプラグインとして同梱されています。「参考文献」に、ISA および GCMV をダウンロードしてインストールする方法、および GCMV を使って GC のパフォーマンス問題をデバッグする方法について説明している記事へのリンクが記載されています。

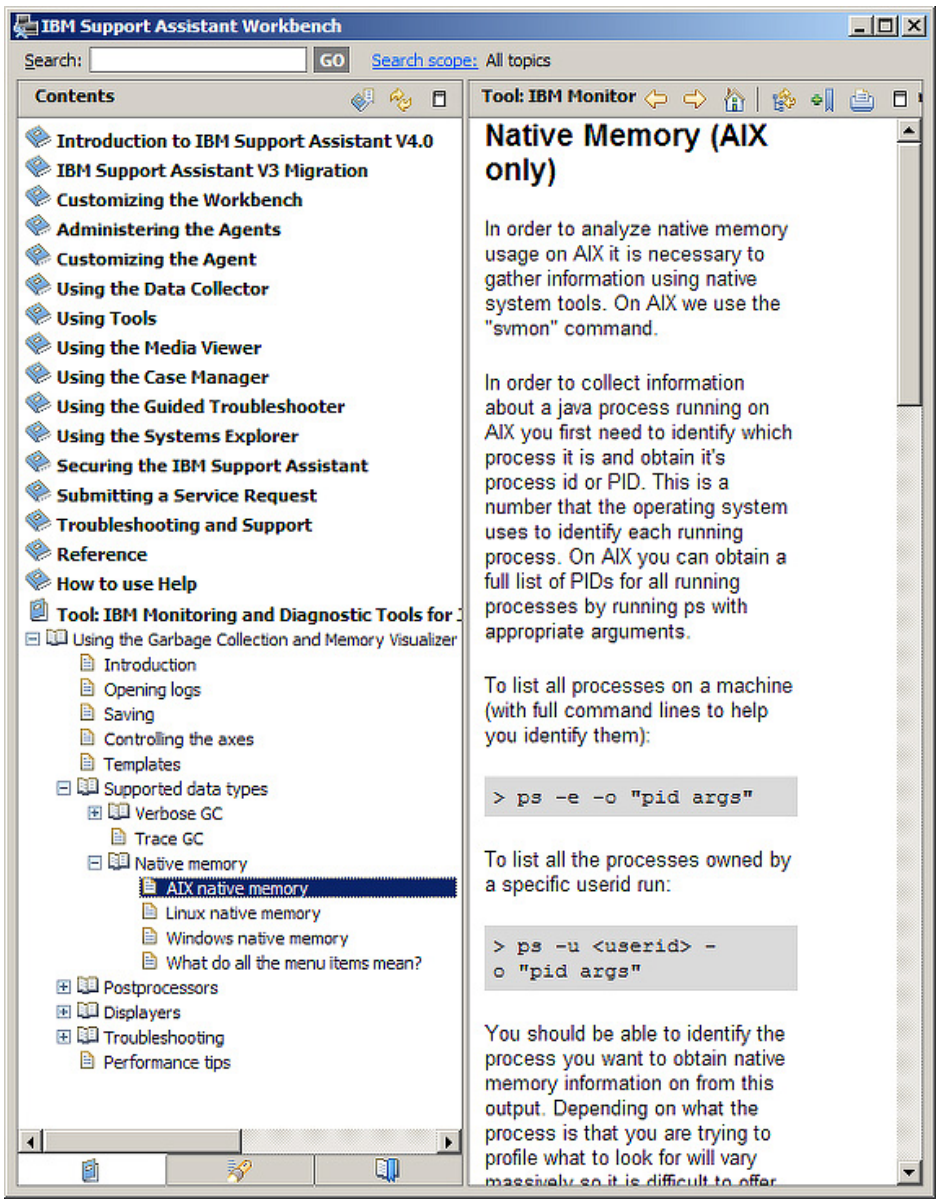
GCMV で AIX ネイティブ・メモリーのプロファイルをグラフ化するには、まず始めにスクリプトを使ってネイティブ・メモリーのデータを収集する必要があります。GCMV の AIX ネイティブ・メモリー・パーサーは AIX `svmon` コマンドからの出力を読み取ります。正しいフォームでデータを収集するスクリプトは、GCMV ヘルプ資料に提供されています。このスクリプトを見つける方法は以下のとおりです。

1. ISA Version 4 (またはそれ以降) をダウンロードしてインストールし、GCMV ツール・プラグインをインストールします (詳細については、「参考文献」を参照)。
2. ISA を起動します。
3. メニュー・バーで Help >> Help Contents の順にクリックして ISA ヘルプ・メニューを表示します。

4. 左ペインで、Tool: IBM Monitoring and Diagnostic Tools for Java - Garbage Collection and Memory Visualizer >> Using the Garbage Collection and Memory Visualizer >> Supported Data Types >> Native memory >> AIX native memory の順に選択して、AIX ネイティブ・メモリーに関する説明を表示します。

図 5 に、GCMV ヘルプ・ツリーの中で、モニター・スクリプトが記載されているセクションを示します。ヘルプ・ファイルの Supported Data Types タブの下に Native memory のセクションがない場合には、最新の GCMV パッケージにアップグレードする必要があります。

図 5. ISA ヘルプ・ダイアログの GCMV AIX メモリー・モニター・スクリプトの位置



このスクリプトを使用するには、AIX マシンにスクリプトをコピーし、モニター対象の Java プロセスを開始します。ps を実行して Java プロセスのプロセス識別子 (PID) を取得してから、モニター・スクリプトを開始してください (以下のコードで、pid はモニター対象プロセスの

ID、output_file はメモリー・ログを保存するファイル、すなわち GCMV がグラフ化するファイルです)。

```
sh aix_memory_monitor.sh pid > output_file
```

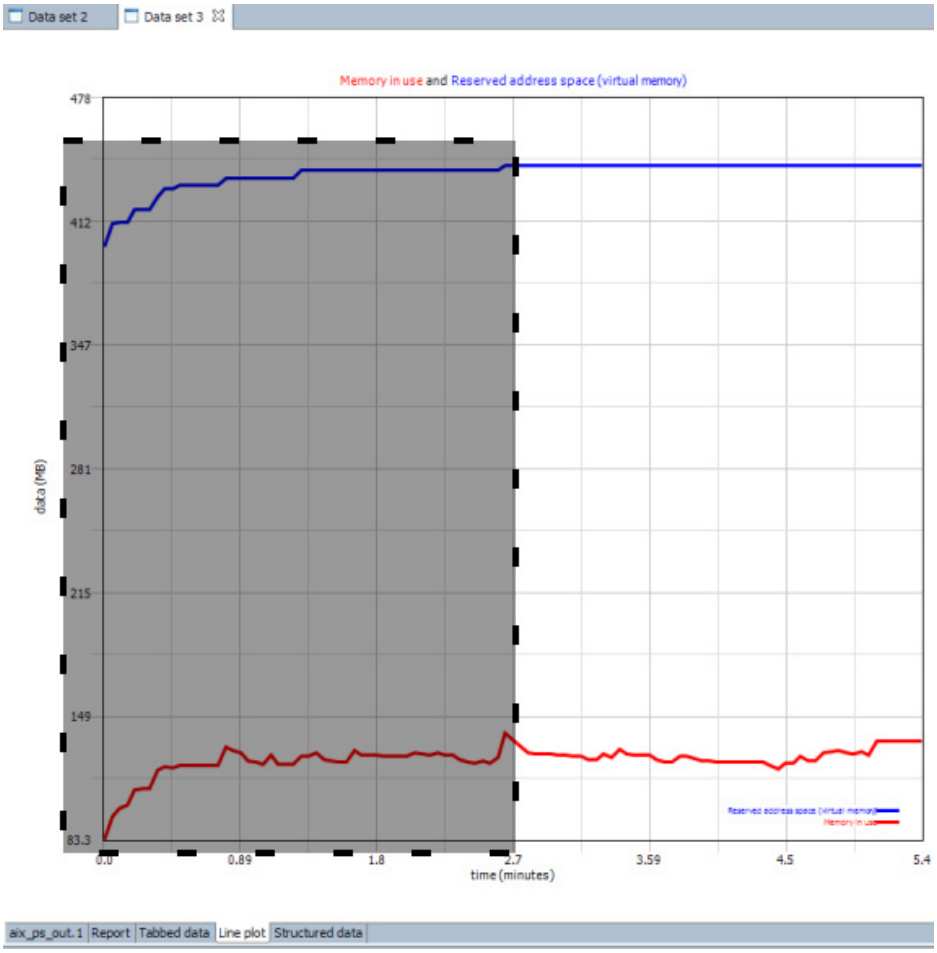
メモリー・ログをグラフ化するには以下の手順を実行します。

1. ISA の Launch Activity ドロップダウン・メニューから、Analyze Problem を選択します。
2. Analyze Problem パネルの上の方にある Tools タブを選択します。
3. IBM Monitoring and Diagnostic Tools for Java - Garbage Collection and Memory Visualizer を選択します。
4. Tools パネルの下の方にある Launch ボタンをクリックします。
5. Browse ボタンをクリックし、ログ・ファイルの位置を指定します。OK をクリックして GCMV を起動します。

ある程度の期間のネイティブ・メモリー使用量のプロファイルが取得できたら、ネイティブ・メモリーのリークが発生しているのか、あるいは使用可能なスペースであまりにも多くのことをしようとしているだけなのかを判断する必要があります。正常に動作している Java アプリケーションでも、ネイティブ・メモリーのフットプリントは起動時から一定ではありません。Java ランタイム・システムのいくつか (特に、JIT コンパイラーおよびクラスローダー) は、時間が経つと初期化し、それによってネイティブ・メモリーが使用されます。初期化の後、しばらくするとメモリーの使用量は横ばいになりますが、初期のネイティブ・メモリーのフットプリントがアドレス空間のほとんどを占めているというシナリオでは、このウォームアップ・フェーズだけでもネイティブ・メモリー不足の原因になり得ます。

図 6 に、Java 負荷テストでのネイティブ・メモリーのフットプリントのグラフを示します。グレーで強調表示した部分がウォームアップ・フェーズです。このフェーズではネイティブ・メモリーのフットプリントが増加し、その後、プロセスが安定状態に達すると一定になります。

図 6. ウォームアップ・フェーズを示す AIX ネイティブ・メモリーのグラフ



ネイティブ・メモリーのフットプリントが作業負荷と相関している場合もあります。アプリケーションが新しい作業負荷に対処するためにスレッドを追加で作成する場合、またはネイティブ・メモリーでバックアップされるストレージ (ダイレクト `ByteBuffer` など) をシステムにかかる負荷の量に比例して割り当てる場合には、負荷が高くなるとネイティブ・メモリーが不足する可能性があります。

JVM ウォームアップ・フェーズでのネイティブ・メモリーの使用量の増加、および負荷に比例したネイティブ・メモリーの使用量の増加が原因のネイティブ・メモリー不足は、使用可能なスペースであまりにも多くのことをしようとしている例です。こうしたシナリオで取り得る方法としては、以下の 3 つがあります。

- **ネイティブ・メモリーの使用量を減らす。**まずは Java ヒープ・サイズを減らすことが効果的な手段です。
- **ネイティブ・メモリーの使用量を制限する。**負荷によってネイティブ・メモリーの増加量が変わる場合には、負荷または負荷のために割り当てるリソースに上限を設ける方法を探してください。
- **使用可能なアドレス空間の量を増やす。**別のメモリー・モデル構成を指定するように `LDR_CNTRL` 環境変数を設定するか、64-bit への移行を検討してください。

正真正銘のネイティブ・メモリー・リークは、ネイティブ・ヒープが絶えず増加し、負荷が取り除かれたり、ガーベッジ・コレクションが実行されたりしても増加が止まらないことによって明らかになります。メモリー・リークの速さは負荷によって変わりますが、リークしたメモリーの合計量が下がることはありません。リークしたメモリーが参照される見込みはないので、スワップアウトして、そのままの状態にしておけます。

リークが発生した場合に取り得る方法は限られています。`LDR_CNTRL` 環境変数で (リーク先のスペースが増えるように) アドレス空間の量を増やすという手段もありますが、それでは最終的にメモリーが不足するまでの時間稼ぎにしかなりません。十分な物理メモリーとアドレス空間がある場合には、プロセスのアドレス空間が枯渇する前にアプリケーションを再起動するという条件で、リークをそのまま許容しておくことも可能です。

何がネイティブ・メモリーを使用しているのかを理解する

ネイティブ・メモリーが不足していると判断した場合、当然次に考えなければならないのは、何がメモリーを使用しているかです。AIX は、どのコード・パスが特定のメモリー・チャンクを割り当てたのかについての情報をデフォルトでは保存しないため、この情報は簡単には入手できません。

ネイティブ・メモリーがどこに消えたのかを理解するために行う最初のステップは、Java 設定に基づいて使用されるはずのネイティブ・メモリーの量を大まかに算出することです。ネイティブ・メモリーの最小使用量は、以下のガイドラインに基づいて概算することができます。

- Java ヒープのサイズは `-Xmx` の値に相当します。
- それぞれの Java スレッドにはネイティブ・スタックと Java スタックがあります。AIX では、この両スタックによってスレッドごとに最低 256KB が使用されます。
- ダイレクト `ByteBuffer` の最小使用量は、`allocate()` ルーチンに指定された値です。

以上の合計が最大ユーザー空間より大幅に少ないとしても、必ずしも安全とは言えません。Java ランタイムにはその他にも、問題の原因となるだけのメモリー量を割り当てるコンポーネントが多数あるからです。一方、この最初の計算が最大ユーザー空間に近いことを示唆しているとしたら、ネイティブ・メモリーの問題を抱えている可能性は大です。ネイティブ・メモリーのリークが疑われる場合、またはメモリーがどこで使用されているかを正確に理解したい場合には、利用できるツールがいくつかあります。

AIX で使用できる多くのメモリー・デバッガーは、一般に以下のカテゴリーのいずれかに分類されます。

- **プリプロセッサー・レベル**。このカテゴリーのデバッガーでは、テスト対象のソースでヘッダーをコンパイルする必要があります。これらのツールのいずれかを使って独自の JNI ライブラリーを再コンパイルすると、コード内のネイティブ・メモリー・リークをトレースすることができます。この種類のツールの一例は、dmalloc です (「参考文献」を参照)。
- **リンカー・レベル**。この場合には、テスト対象のバイナリーをテスト対象のライブラリーと再びリンクする必要があります。個々の JNI ライブラリーを再リンクすることは可能ですが、Java ランタイム全体を再リンクすることはお勧めしません。変更後のバイナリーでの実行はサポートされない可能性があるためです。この種類のツールの一例は、ccmalloc です (「参考文献」を参照)。

- **ランタイム・リンカー・レベル。** `LD_PRELOAD` 環境変数を使用してライブラリーをプリロードして、標準メモリー・ルーチンを、インスツルメンテーションを追加したバージョンに置き換えます。ソース・コードのコンパイルやリンクを再度実行する必要はありませんが、これらのツールの多くは、Java ランタイムではそれほど有効に機能しません (例えば、Linux などの他のオペレーティング・システムで使用可能な NJAMD などのツールは、AIX を十分にサポートしません)。
- **OS レベル。** AIX には、ネイティブ・メモリー・リークをデバッグするための `MALLOCDEBUG` ツールが用意されています。

`MALLOCDEBUG` を使用してメモリー・リークを診断する方法については、記事「Isolate and resolve memory leaks using MALLOCDEBUG on AIX Version 5.3」を参照してください。ここではリークが発生している Java アプリケーションの出力に焦点を絞り、ネイティブ・メモリーがリークしている JNI アプリケーションを `MALLOCDEBUG` でデバッグする例を説明します。

この記事のサンプル・パック (「ダウンロード」を参照) に含まれている `LeakyJNIApp` という Java アプリケーションは、ループで動作し、ネイティブ・メモリーでリークが起きている JNI メソッドを呼び出します。デフォルトでは、ネイティブ・メモリーが枯渇するまで実行を続けます。アプリケーションを終了させるには、秒単位の実行時間をコマンドライン引数として渡してください。

`malloc` デバッグ用の環境を構成するには、`MALLOCDEBUG` および `MALLOCTYPE ####`を以下のように設定します。

```
export MALLOCTYPE=debug
export MALLOCDEBUG=report_allocations,stack_depth:3
```

`stack_depth:3` パラメーターを追加しているのは、`malloc` を呼び出したときに収集するスタック・トレースを制限するためです。JVM には独自のスレッド・スタック構造があり、これがスタック・ウォーキング・アプリケーションを混乱させてクラッシュを引き起こす場合があります。そのため、スタックの深さを 3 レベルに制限することで、予期せぬ振る舞いが起こるのを防ぎます。

環境の構成が完了したら、`LeakyJNIApp` を 10 秒間実行して、`malloc` ログが含まれる `stderr` 出力を取得します。

```
./run_leaky_jni_app_aix_32.sh 10 2>memory_log.txt
```

これにより、memory_log.txt ファイルにはリークが発生したメモリー・ブロックに関する詳細が含まれます。

```
    Allocation #1175: 0x328B0C00
        Allocation size: 0x400
        Allocation traceback:
        0xD046F4D8  malloc
        0x32834258  Java_com_ibm_jtc_demos_LeakyJNIApp_nativeMethod
        0x30CEE7B0  ??
```

メモリー・ログ・ファイルを検査することによって、問題の原因を突き止められる場合があります。あるいは、「Isolate and resolve memory leaks using MALLOCDEBUG on AIX Version 5.3」に付属の format_mallocdebug_op.sh スクリプトを使用してメモリー・ログを要約することもできます。

サマリー・スクリプトを memory_log.txt ファイルで実行すると、以下の出力が生成されます。

```
$ ./format_mallocdebug_op.sh memory_log.txt
Parsing output from debug malloc ...
Analysed 50 stacks ...
Analysed 100 stacks ...
Analysed 150 stacks ...
Analysed 200 stacks ...
Analysed 250 stacks ...
Analysed 300 stacks ...
Analysed 350 stacks ...
Analysed 400 stacks ...
Analysed 450 stacks ...
Analysed 500 stacks ...
Analysed 550 stacks ...
Analysed 600 stacks ...
Analysed 650 stacks ...
Analysed 700 stacks ...
Analysed 750 stacks ...
Analysed 800 stacks ...
Analysed 850 stacks ...
Analysed 900 stacks ...
Analysed 950 stacks ...
Analysed 1000 stacks ...
Analysed 1050 stacks ...
Analysed 1100 stacks ...
Analysed 1150 stacks ...

??
Java_com_ibm_jtc_demos_LeakyJNIApp_nativeMethod
malloc
################################
98304 bytes leaked in 96 Blocks
################################
.
.
.
```

この出力から、リークの原因が `LeakyJNIApp.nativeMethod()` であることがわかります。

専用のデバッグ・アプリケーションにも、同様の機能を提供しているものがいくつかあります。オープンソースのツールと専用のツールを含め、新しいツールが常に次々と開発されているので、最先端の技術を調べてみる価値はあります。

OS およびサード・パーティーのツールによってデバッグは容易になりますが、それでも確実なデバッグ手法が必要なことには変わりありません。以下に、提案されるステップをいくつか記載します。

- **テスト・ケースを抽出すること。**ネイティブ・リークを再現するために使用できるスタンドアロンの環境を作成します。これによって、デバッグが大幅に単純化されます。
- **テスト・ケースをできる限り絞り込むこと。**関数をスタブ化して、ネイティブ・リークを引き起こしているコード・パスを特定するようにしてください。独自の JNI ライブラリーがあ

る場合には、ライブラリー全体を 1 つずつスタブ化して、リークの原因となっているライブラリーを判別します。
- Java ヒープ・サイズを減らすこと。プロセスで仮想アドレス空間を最も多く使用すると考えられるのは、Java ヒープです。Java ヒープを減らすことによって、ネイティブ・メモリーを使用する他のものが、より多くのスペースを使えるようになります。ネイティブ・メモリーがリークしている場合には、プログラムの実行時間を延ばすための時間稼ぎとなります。
- ネイティブ・プロセス・サイズを相関させること。ネイティブ・メモリー使用量の時間推移をグラフ化したら、そのグラフをアプリケーションの作業負荷と GC データと比較できるようになります。リークの速さが負荷のレベルに比例する場合、それは、それぞれのトランザクションまたは操作のパス上にある何かがリークの原因となっていることを示唆します。GC の発生時にネイティブ・プロセスのサイズが著しく小さくなるとしたら、原因はリークではなく、ネイティブ・メモリーでバックアップされるオブジェクト (ダイレクト `ByteBuffer` など) が多いことです。このようなオブジェクトによって保持されるメモリー量を減らすには、Java ヒープ・サイズを減らすか (それによって、コレクションをより頻繁に発生させます)、またはガーベッジ・コレクターによるクリーンアップに依存する代わりにオブジェクト・キャッシュ内でそれらのオブジェクトを自ら管理してください。

Java ランタイム自体が原因と考えられるメモリー・リークまたはメモリー使用量の増加が確認された場合には、ランタイム・ベンダーに詳細なデバッグを依頼しなければならないこともあります。

限界を取り除く: 64-bit への変更

32-bit Java ランタイムの場合、アドレス空間が比較的小さいことから、簡単にネイティブ・メモリー不足が発生します。32-bit OS が提供する 2GB から 4GB のユーザー空間が、システムに搭載された物理メモリーの量に満たないことは珍しくありません。最近のデータ量の多いアプリケーションとなると、使用可能なスペースを簡単に使いきってしまう場合があります。

アプリケーションが 32-bit アドレス空間に収まりきらない場合、64-bit Java ランタイムに移行することで大幅にユーザー空間を増やすことができます。64-bit Java ランタイムを AIX で実行できれば、448 ペタバイトのアドレス空間のおかげで、膨大な Java ヒープ、そしてアドレス空間に関する頭痛の種が少ない世界への道が開けます。

ただし、64-bit への移行はネイティブ・メモリーに関するすべての問題に共通のソリューションではなく、すべてのデータを保持するのに十分な物理メモリーが必要であることには変わりありません。Java ランタイムが物理メモリーに収まらないとしたら、パフォーマンスは耐えきれないほどに劣化します。OS がスワップ空間との間で頻繁に Java ランタイムのデータをやりとりしなければならなくなるためです。これと同じ理由から、64-bit はメモリー・リークに対する恒久的なソリューションにもなりません。リーク先のスペースを増やすことで、次の強制再起動までの時間を稼いでいるだけです。

32-bit のネイティブ・コードを 64-bit ランタイムで使用することは不可能なので、ネイティブ・コード (JNI ライブラリー、JVMTI (JVM Tool Interface)、JVMPI (JVM Profiling Interface)、および JVMDI (JVM Debug Interface) エージェント) はすべて、64-bit 用にコンパイルしなおす必要があります。また、64-bit ランタイムの動作速度は、同じハードウェア上の 32-bit ランタイムよりも遅くなる可能性があります。64-bit ランタイムは 64-bit ポインター (ネイティブ・アドレス参照) を

使用するため、同じデータが含まれる同じオブジェクトだとしても、64-bit では 32-bit での場合よりも大きなスペースを占めます。オブジェクトが大きくなると、同じデータ量を保持するにも大きなヒープでなければ同様の GC パフォーマンスを維持できません。このことから、OS とハードウェア・メモリー・システムの動作が遅くなります。意外なことに、Java ヒープのサイズが大きくなっても、GC 停止時間が長くなることはありません。停止時間は、ヒープ上の有効なデータの量 (増えてはいないはずです) に大きく左右されること、そして一部の GC アルゴリズムはヒープが大きいほど効率性が上がることがその理由です。

これまで 64-bit ランタイムのパフォーマンスは対応する 32-bit ランタイムに劣っていましたが、この状況は IBM Developer Kit for Java 6.0 で大幅に改善されています。圧縮参照技術 (`-Xcompressedrefs` コマンドライン引数によって有効になります) が追加されたことにより、大きな Java ヒープ (Service Refresh 2 では 20GB から 30GB まで) で 32-bit オブジェクトのアドレッシングを使用することが可能になりました。そのため、以前の 64-bit ランタイムにおける速度低下の大きな原因であった「オブジェクトの膨張」がなくなります。

Java ランタイム・パフォーマンスの比較研究は、この記事では行いませんが、64-bit への移行を検討しているとしたら、64-bit でのアプリケーションのテストを早い段階で、そして可能であれば IBM Developer Kit for Java 6 の圧縮参照を利用して行うだけの価値はあります。

まとめ

大規模な Java アプリケーションを設計および実行する際には、ネイティブ・メモリーを理解することが不可欠であるとは言え、この部分の作業については無視されがちです。ネイティブ・メモリーは複雑なマシンや OS の詳細に関連しますが、Java はそのような詳細から開発者を救うために設計されているからです。しかし、JRE はネイティブ・プロセスであり、これらの複雑な詳細によって定義された環境のなかで機能しなければなりません。そのため、Java アプリケーションから最高のパフォーマンスを引き出すためには、Java ランタイムによるネイティブ・メモリーの使用にアプリケーションがどのように影響するかを理解する必要があります。

ネイティブ・メモリーの不足は Java ヒープの不足と同じように見えるものの、これをデバッグして解決するにはJava ヒープ不足の場合とは異なるツール・セットが必要になります。ネイティブ・メモリーの問題を解決する鍵となるのは、Java アプリケーションを実行するハードウェアおよび OS によって課せられる制約を理解し、理解したその内容と、ネイティブ・メモリーの使用量をモニターする OS ツールの知識とを組み合わせることです。この手法に従うことで、Java アプリケーションで発生する可能性のある最も困難な問題のいくつかを解決する態勢が整います。

ダウンロード

内容	ファイル名	サイズ
Native memory example code	j-nativememory-aix.zip	33KB
Javacore memory analysis script	j-nativememory-aix2.zip	3KB

著者について

Andrew Hall

Andrew Hall joined IBM's Java Technology Centre in 2004, starting in the system test team where he stayed for two years. He spent 18 months in the Java service team, where he debugged dozens of native-memory problems on many platforms. He is currently a developer in the Java Reliability, Availability and Serviceability team. In his spare time he enjoys reading, photography, and juggling.