

# Java コードから Java ヒープまで アプリケーションのメモリー使用量を把握し、最適化する

Chris Bailey

Java Service Architect  
IBM

2012年 4月 12日

この記事では Java コードのメモリー使用量を把握できるように、`int` 値を `Integer` オブジェクトに格納する場合のメモリー・オーバーヘッドから、オブジェクト委譲のコスト、各種コレクションの各タイプのメモリー効率まで、さまざまな話題を取り上げます。さらに、アプリケーションの非効率的な部分を判別する方法、そして適切なコレクションを選択してコードを改善する方法についても説明します。

アプリケーション・コードのメモリー使用量の最適化は目新しい話題ではありませんが、一般によく理解されているとも言えません。この記事では、Java プロセスのメモリー使用量について簡単に説明した後、皆さんが作成する Java コードのメモリー使用量を詳しく探ります。そして最後に、`HashMap` や `ArrayList` などの Java コレクションを使用する領域において、アプリケーション・コードのメモリー使用効率を高める方法を紹介します。

## 基礎知識: Java プロセスのメモリー使用量

### ネイティブ・メモリーに関する詳しい説明

Java アプリケーションにおけるプロセスのメモリー使用量について十分に理解するには、Andrew Hall 氏による developerWorks の記事「Thanks for the memory (メモリーよ、ありがとう)」を読んでください。同記事は [Windows](#) および Linux 用と、[AIX](#) 用の 2 本に分かれており、それぞれの OS におけるメモリー・レイアウトと使用可能なユーザー空間について、さらには Java ヒープとネイティブ・ヒープとの間でのやりとりについて解説しています。

Java アプリケーションを実行するために、コマンドラインで「java」と入力するか、Java ベースのミドルウェアを起動すると、Java ランタイムがオペレーティング・システム・プロセスを作成します。それはまるで、C で書かれたプログラムを実行しているかのようです。実際、ほとんどの JVM は主に C あるいは C++ で書かれています。オペレーティング・システム・プロセスとしての Java ランタイムには、メモリーに関して他のあらゆるプロセスと同じ制約が課せられます。その制約とは、アドレス指定可能なメモリー空間はアーキテクチャーによって決まり、ユーザー空間はオペレーティング・システムによって決まるということです。

アーキテクチャーによって決まるアドレス指定可能なメモリー空間は、何ビット・プロセッサを使用しているかによっても変わってきます (例えば、32 ビット・プロセッサであるか 64 ビット・プロセッサであるか、あるいは 31 ビットのプロセッサ (メインフレームの場合) であるかに依存します)。プロセッサがアドレス指定可能なメモリーの範囲は、プロセスで扱えるビット数によって決まります。32 ビットを扱えるとしたら、アドレス指定可能な範囲は  $2^{32}$  です。これは、4,294,967,296 ビット、つまり 4GB に相当します。64 ビット・プロセッサでアドレス指定可能な範囲はこれよりも大幅に増えて、 $2^{64}$  となります。つまり、18,446,744,073,709,551,616、つまり 16 エクサバイトです。

プロセッサ・アーキテクチャーによって決まるアドレス指定可能なメモリー空間は、その一部が OS によって OS 自体のカーネルや C ランタイム (C または C++ で書かれた JVM の場合) のために使用されます。OS と C ランタイムによって使用されるメモリーの量は、どの OS を使用しているかによって変わってきますが、通常は相当な量が使用されます。例えば、Windows がデフォルトで使用する量は 2GB です。残りのアドレス指定可能な空間 (ユーザー空間と呼ばれます) が、実際に実行中のプロセスで使用できるメモリーということになります。

Java アプリケーションで言うと、ユーザー空間は Java プロセスが使用するメモリーであり、実質的には Java ヒープとネイティブ (非 Java) ヒープという 2 つのプールで構成されます。Java ヒープのサイズは JVM の Java ヒープ設定によって制御されます。この Java ヒープ設定には、Java ヒープの最小サイズを設定する `-Xms` と Java ヒープの最大サイズを設定する `-Xmx` があります。ネイティブ・ヒープは、Java ヒープが最大サイズの設定で割り当てられた後の、残りのユーザー空間です。図 1 に一例として、32 ビット Java プロセスの場合のメモリー・レイアウトを示します。

図 1. 32 ビット Java プロセスでのメモリー・レイアウトの例

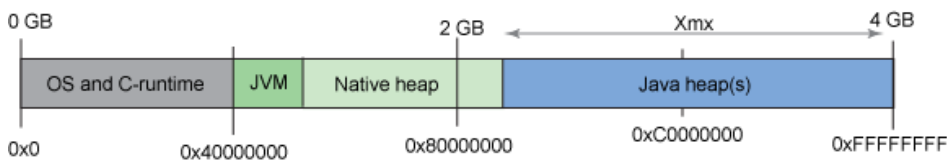


図 1 では、アドレス指定可能な 4GB のメモリー空間のうち、約 1GB を OS と C ランタイムが使用し、約 2GB を Java ヒープが、そしてその残りをネイティブ・ヒープが使用しています。JVM 自体もメモリーを使用すること (OS カーネルと C ランタイムと同じように使用します)、そして JVM が使用するメモリーはネイティブ・ヒープの一部であることに注意してください。

## Java オブジェクトの分析

Java コードで `new` 演算子を使用して Java オブジェクトのインスタンスを作成する場合、そのインスタンスに割り当てられるデータの量は、おそらく皆さんの想像を遥かに超えているでしょう。驚かれるかもしれませんが、例えば `int` 値と `Integer` オブジェクト (`int` 値を保持できる最小のオブジェクト) のサイズの比は、一般に 1:4 にもなります。この増加分のオーバーヘッドは、JVM が Java オブジェクト (この例では `Integer`) を表すために使用するメタデータです。

オブジェクトのメタデータのサイズは、JVM のバージョンやベンダーによって異なりますが、通常は以下のメタデータを合計したサイズとなります。

- ・ **クラス:** オブジェクトの型を表すクラス情報を指すポインター。 `java.lang.Integer` オブジェクトの場合、これは `java.lang.Integer` クラスを指すポインターです。

- **フラグ:** オブジェクトの状態を表すフラグを集めたもの。このなかには、メタデータとしてオブジェクトのハッシュ・コードが含まれる場合にそのことを示すフラグや、オブジェクトの「形式」を示すフラグ (オブジェクトが配列の形式をしているかどうかを示すフラグ) も含まれます。
- **ロック:** オブジェクトの同期情報。つまりオブジェクトが現在同期されているかどうかを示します。

オブジェクトのメタデータの後は、オブジェクト・インスタンスに保管されているフィールドからなるオブジェクト・データの本体が続きます。java.lang.Integer オブジェクトの場合、オブジェクト・データは `int` が 1 つのみです。

したがって、32 ビット JVM を実行している場合に java.lang.Integer オブジェクトのインスタンスを作成すると、オブジェクトのレイアウトは図 2 のようになります。

図 2. 32 ビット Java プロセスでの `java.lang.Integer` オブジェクトのレイアウト例

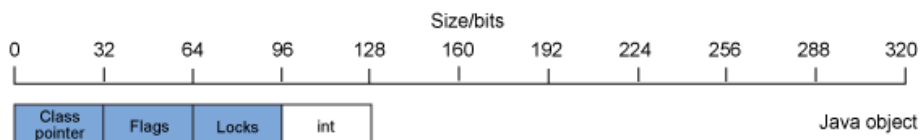


図 2 に示されているように、`int` 値の 32 ビットのデータを格納するために、128 ビットのデータが使用されます。128 ビットのうち、32 ビット以外はすべてオブジェクトのメタデータが使用しています。

## Java 配列オブジェクトの分析

`int` 値の配列をはじめとする配列オブジェクトは、標準的な Java オブジェクトと似たような形式と構造を持ちますが、最も違う点は、配列オブジェクトには配列のサイズを示すメタデータが追加されることです。配列オブジェクトは以下のメタデータで構成されます。

- **クラス:** オブジェクトの型を表すクラス情報を指すポインター。`int` フィールドの配列の場合、これは `int[]` クラスを指すポインターです。
- **フラグ:** オブジェクトの状態を表すフラグを集めたもの。このなかには、メタデータとしてオブジェクトのハッシュ・コードが含まれる場合にそのことを示すフラグや、オブジェクトの形式を示すフラグ (オブジェクトが配列の形式をしているかどうかを示すフラグ) も含まれます。
- **ロック:** オブジェクトの同期情報。つまりオブジェクトが現在同期されているかどうかを示します。
- **サイズ:** 配列のサイズ。

図 3 に一例として、`int` 配列オブジェクトのレイアウトを示します。

図 3. 32 ビット Java プロセスでの `int` 配列オブジェクトのレイアウト例

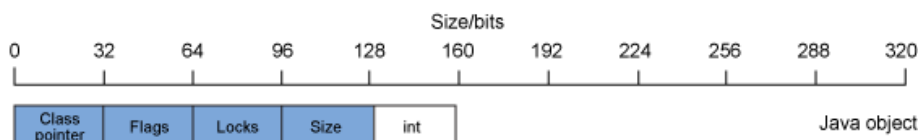


図 3 では、`int` 値の 32 ビットのデータを格納するために、160 ビットのデータが使用されています。160 ビットのうち、32 ビット以外はすべて配列のメタデータが使用しています。byte、int、long などのプリミティブ型の場合、エントリーが 1 つのみの配列は、これらに対応する単一フィールドのラッパー・オブジェクト (`Byte`、`Integer`、または `Long`) よりも、メモリーの点でコストがかかります。

## さらに複雑なデータ構造の分析

優れたオブジェクト指向の設計およびプログラミングでは、カプセル化 (データへのアクセスを制御するインターフェース・クラスを提供すること) と委譲 (ヘルパー・オブジェクトを使用してタスクを実行すること) を用いることを推奨しています。カプセル化と委譲によって、ほとんどのデータ構造の表現には複数のオブジェクトが関与することになります。その単純な一例は、`java.lang.String` オブジェクトです。`java.lang.String` オブジェクトには文字型の配列がデータとして格納され、この文字型配列に対する読み書きと管理を行う `java.lang.String` オブジェクトによってカプセル化されます。`java.lang.String` オブジェクトのレイアウトは、32 ビット Java プロセスでは図 4 のようになります。

図 4. 32 ビット Java プロセスでの `java.lang.String` オブジェクトのレイアウト例

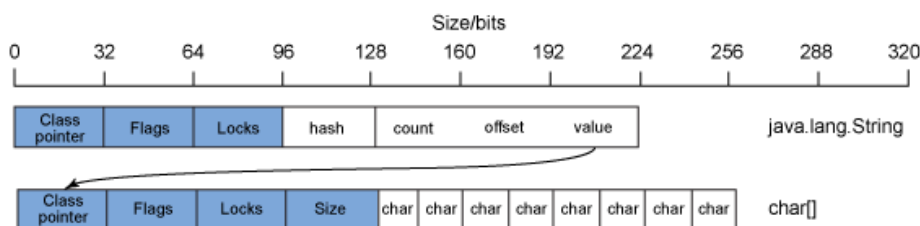


図 4 に示されているように、`java.lang.String` オブジェクトにはオブジェクトの標準的なメタデータに加え、文字列データを管理するためのフィールドが含まれています。これらのフィールドは通常、ハッシュ値、文字列のサイズのカウンタ、文字列データへのオフセット、文字の配列自体へのオブジェクト参照です。

つまり、8 文字の文字列 (char データによる 128 ビット) の場合、文字配列に 256 ビットのデータが使用され、その配列を管理する `java.lang.String` オブジェクトに 224 ビットのデータが使用されるため、128 ビット (16 バイト) のデータを表現するには合計 480 ビット (60 バイト) が必要になります。このオーバーヘッド比は、3.75:1 です。

一般に、データ構造が複雑になるにつれ、オーバーヘッドは大きくなっていきます。このことに関しては、次のセクションでさらに詳しく検討します。

## 32 ビットおよび 64 ビットの Java オブジェクト

これまでの例で取り上げたオブジェクトでのサイズとオーバーヘッドは、32 ビット Java プロセスに適用されるものです。「[基礎知識: Java プロセスのメモリー使用量](#)」セクションで説明したように、64 ビット・プロセッサの場合、32 ビット・プロセッサよりもアドレス指定可能なメモリー空間は遥かに大きくなります。64 ビット・プロセスでは、Java オブジェクトに含まれる一部のデータ・フィールドのサイズ (具体的には、オブジェクトのメタデータと、別のオブジェクトを

参照するフィールド) も 64 ビットまで増やす必要があります。それ以外のデータ・フィールドの型 (int、byte、long など) については、サイズは変わりません。図 5 に、64 ビットの Integer オブジェクトおよび int 配列のレイアウトを示します。

図 5. 64 ビット Java プロセスでの java.lang.Integer オブジェクトおよび int 配列オブジェクトのレイアウト例

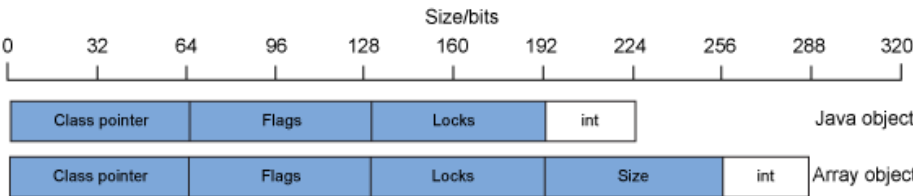


図 5 に示されている 64 ビットの Integer オブジェクトの場合、int フィールド用の 32 ビットを格納するために使用されるデータは 224 ビットです (オーバーヘッド比 7:1)。64 ビットの単一要素からなる int 配列では、32 ビットの int エントリを格納するために 288 ビットのデータが使用されます (オーバーヘッド比 9:1)。この影響は、実際のアプリケーションでは、今まで 32 ビット Java ランタイムで実行されていたアプリケーションを 64 ビット Java ランタイムに移した場合、アプリケーションの Java ヒープ・メモリー使用量が劇的に増加するという形で現れます。増加率は一般に、元のヒープ・サイズの約 70 パーセントです。例えば、32 ビット Java ランタイムで 1GB の Java ヒープを使用する Java アプリケーションは、64 ビット Java ランタイムでは 1.7GB の Java ヒープを使用することになります。

このメモリー使用量の増加は、Java ヒープに限られた話ではありません。ネイティブ・ヒープ・メモリー領域の使用量も同じく増加します。その増加率は、場合によっては 90 パーセントにも及びます。

表 1 に、アプリケーションが 32 ビット・モードで実行される場合と、64 ビット・モードで実行される場合の、それぞれのオブジェクトと配列のフィールド・サイズを記載します。

表 1. 32 ビット Java ランタイムと 64 ビット Java ランタイムでのオブジェクトのフィールド・サイズ

フィールドの型	フィールド・サイズ (ビット)			
	オブジェクト		配列	
	32 ビット	64 ビット	32 ビット	64 ビット
boolean	32	32	8	8
byte	32	32	8	8
char	32	32	16	16
short	32	32	16	16
int	32	32	32	32
float	32	32	32	32
long	64	64	64	64
double	64	64	64	64
オブジェクト・フィールド	32	64 (32*)	32	64 (32*)



オブジェクト・メタデータ	32	64 (32*)	32	64 (32*)
--------------	----	----------	----	----------

\*オブジェクト・フィールドのサイズ、およびオブジェクト・メタデータの各エントリーに使用されるデータのサイズは、[参照圧縮または圧縮 OOP](#) という技術を使用することで、32 ビットに縮小することができます。

## 参照圧縮および圧縮 OOP (Ordinary Object Pointer)

IBM JVM と Oracle JVM は、それぞれ参照圧縮 (-Xcompressedrefs) オプション、圧縮 OOP (-XX:+UseCompressedOops) オプションという形でオブジェクト参照圧縮機能を提供しています。これらのオプションを使用することで、オブジェクト・フィールドとオブジェクト・メタデータ値を 64 ビットではなく 32 ビットにして格納することができます。これによる効果は、アプリケーションが 32 ビット Java ランタイムから 64 ビット Java ランタイムに移されることによって生じる 70 パーセントの Java ヒープ・メモリーの増加が帳消しになることです。ただし、これらのオプションは、ネイティブ・ヒープのメモリー使用量には影響しません。64 ビット Java ランタイムでのネイティブ・ヒープのメモリー使用量は、やはり 32 ビット Java ランタイムよりも大きくなります。

## Java コレクションのメモリー使用量

大抵のアプリケーションでは、大量のデータを保管して管理するために、コア Java API の一部として提供されている標準 Java コレクション・クラスが使用されます。アプリケーションにとってメモリー・フットプリントの最適化が重要な場合には、それぞれのコレクションが提供する関数、そしてコレクションに伴うメモリー・オーバーヘッドについて理解しておくが大いに役に立ちます。一般に、コレクションが持つ関数の機能レベルが高いほど、そのメモリー・オーバーヘッドは大きくなります。したがって、必要以上の関数を提供するコレクションのタイプを使用すると、不要なメモリー・オーバーヘッドが追加されることになります。

よく使用されているコレクションには以下のものがあります。

- `HashSet`
- `HashMap`
- `Hashtable`
- `LinkedList`
- `ArrayList`

`HashSet` を除き、上記では関数とメモリー・オーバーヘッドが多いものから順にコレクションを記載してあります (`HashSet` は `HashMap` オブジェクトのラッパーであるため、このコレクションが提供する関数は `HashMap` よりも少ないとは言え、実質的にはメモリー・サイズは少し大きくなります)。

## Java コレクション: `HashSet`

`HashSet` は、`Set` インターフェースの実装です。Java Platform SE 6 API のドキュメントでは、`HashSet` を以下のように説明しています。

重複要素が含まれないコレクション。より正式には、`e1.equals(e2)` のように `e1` と `e2` の要素ペアを持たず、`null` 要素は最大でも 1 つしかないセットです。その名前が示すように、このインターフェースは、数学で言う集合の抽象化をモデル化します。

格納できる null 要素は最大 1 つであり、重複する要素は許容されないという点で、HashSet が持つ機能の数は HashMap よりも限られます。この実装は HashMap のラッパーであり、HashMap オブジェクトに格納できる要素は、HashSet オブジェクトによって管理されます。HashMap の機能を制限する関数が追加されているということは、HashSet の方が HashMap よりもメモリー・オーバーヘッドが多少大きくなることを意味します。

図 6 に、32 ビット Java ランタイムでの HashSet のレイアウトとメモリー使用量を示します。

図 6. 32 ビット Java ランタイムでの HashSet のメモリー使用量とレイアウト

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.HashSet @ 0x10a6d908	16	144
java.util.HashMap @ 0x10a6d918	48	128

図 6 には、java.util.HashSet オブジェクトの Shallow Heap (個々のオブジェクトのメモリー使用量) と Retained Heap (個々のオブジェクトとその子オブジェクトのメモリー使用量) が、いずれもバイト単位で示されています。Shallow Heap のサイズは 16 バイトで、Retained Heap のサイズは 144 バイトとなっています。HashSet を作成する際のデフォルト容量 (集合に格納できるエントリーの数) は 16 エントリーになります。このデフォルト容量で作成され、エントリーが格納されていない HashSet は、144 バイトを占有します。これは、HashMap のメモリー使用量よりも 16 バイト多いこととなります。表 2 に、HashSet の特性を記載します。

表 2. HashSet の特性

項目	内容
デフォルト容量	16 エントリー
空の場合のサイズ	144 バイト
オーバーヘッド	16 バイト + HashMap のオーバーヘッド
10K のコレクションの場合のオーバーヘッド	16 バイト + HashMap のオーバーヘッド
検索/挿入/削除のパフォーマンス	O(1) — 要素の数とは関係なく、(ハッシュ衝突がないことを前提として) 所要時間は一定です。

## Java コレクション: HashMap

HashMap は、Map インターフェースの実装です。Java Platform SE 6 API のドキュメントでは、HashMap を以下のように説明しています。

キーを値にマッピングするオブジェクト。マップには重複するキーを含めることはできません。各キーがマッピングできる値は最大 1 つです。

HashMap は、ハッシュ関数を使用してキーをキー/値のペアが格納されるコレクションのインデックスへと変換することで、キー/値のペアを保管する手段を提供します。これにより、データ・ロケーションへの高速アクセスが可能になります。null エントリーおよび重複するエントリーを格納できることから、HashMap は HashSet を単純化したものであると言えます。

HashMap の実装は、HashMap\$Entry オブジェクトの配列です。図 7 に、32 ビット Java ランタイムでの HashMap のメモリー使用量とレイアウトを示します。

## 図 7.32 ビット Java ランタイムでの **HashMap** のメモリ使用量とレイアウト

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.HashMap @ 0x10a6d918	48	128
java.util.HashMap\$Entry[16] @ 0x10a6d948	80	80

図 7 に示されているように、HashMap を作成すると、HashMap オブジェクトと併せて、デフォルト容量の 16 エントリーで HashMap\$Entry オブジェクトの配列が作成されます。したがって、完全に空の HashMap のサイズは 128 バイトになります。HashMap に格納されるキー/値のペアは HashMap\$Entry オブジェクトによってラップされますが、このオブジェクト自体にもオーバーヘッドがあります。

ほとんどの HashMap\$Entry オブジェクトの実装には、以下のフィールドが含まれます。

- int KeyHash
- Object next
- Object key
- Object value

32 バイトの HashMap\$Entry オブジェクトは、コレクションに格納されたデータのキー/値のペアを管理します。つまり、HashMap のオーバーヘッドの合計には、HashMap オブジェクト、HashMap\$Entry 配列エントリー、そしてエントリーごとの HashMap\$Entry オブジェクトが含まれるということです。これは、以下の式で表現することができます。

HashMap オブジェクト + 配列オブジェクトのオーバーヘッド + (エントリー数 \*  
(HashMap\$Entry 配列エントリー + HashMap\$Entry オブジェクト))

10,000 のエントリーを持つ HashMap の場合、HashMap、HashMap\$Entry 配列、および HashMap\$Entry オブジェクトだけでオーバーヘッドは約 360K になります。これは、保管されるキーと値のサイズが計上される前の数字です。

表 3 に HashMap の特性を記載します。

**表 3. HashMap の特性**

項目	内容
デフォルト容量	16 エントリー
空の場合のサイズ	128 バイト
オーバーヘッド	64 バイト + 36 バイト/エントリー
10K のコレクションの場合のオーバーヘッド	~360K
検索/挿入/削除のパフォーマンス	O(1) — 要素の数とは関係なく、(ハッシュ衝突がないことを前提として) 所要時間は一定です。

## Java コレクション: **Hashtable**

Hashtable は HashMap と同じく、Map インターフェースの実装です。Java Platform SE 6 API のドキュメントでは、Hashtable を以下のように説明しています



このクラスは、キーを値にマッピングするハッシュ・テーブルを実装します。null 以外の任意のオブジェクトを、キーまたは値として使用することができます。

Hashtable は HashMap と非常によく似ていますが、2 つの制約事項があります。1 つはキーのエントリーとしても、値のエントリーとしても null 値を使用できないこと、そしてもう 1 つは、同期されたコレクションであることです。それとは対照的に、HashMap は null 値を受け入れ、同期をとりません。ただし、`Collections.synchronizedMap()` メソッドを使用して同期することは可能です。

Hashtable の実装にしても、HashMap の実装と同じくエントリー・オブジェクトの配列です (この場合は `Hashtable$Entry` オブジェクトの配列)。図 8 に、32 ビット Java ランタイムでの Hashtable のメモリー使用量とレイアウトを示します。

図 8. 32 ビット Java ランタイムでの Hashtable のメモリー使用量とレイアウト

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.Hashtable @ 0x1bae9290	40	104
java.util.Hashtable\$Entry[11] @ 0x1bae92b8	64	64

図 8 には、Hashtable を作成すると、40 バイトのメモリーを使用する Hashtable オブジェクトと、11 エントリーのデフォルト容量が設定された `Hashtable$Entry`s の配列が作成されることが示されています。空の Hashtable の場合、その合計サイズは 104 バイトです。

`Hashtable$Entry` は、事実上 `HashMap` と同じ以下のデータを保管します。

- `int` `keyHash`
- `Object` `next`
- `Object` `key`
- `Object` `value`

したがって、`Hashtable$Entry` オブジェクトでも、Hashtable に含まれるキー/値のエントリーは 32 バイトなので、10K のエントリー・コレクション (約 360K) の Hashtable のオーバーヘッドとサイズは、HashMap の計算と同様になります。

表 4 に、Hashtable の特性を記載します。

表 4. Hashtable の特性

項目	内容
デフォルト容量	11 エントリー
空の場合のサイズ	104 バイト
オーバーヘッド	56 バイト + 36 バイト/エントリー
10K のコレクションの場合のオーバーヘッド	~360K
検索/挿入/削除のパフォーマンス	O(1) — 要素の数とは関係なく、(ハッシュ衝突がないことを前提として) 所要時間は一定です。

ご覧のとおり、Hashtable のデフォルト容量は HashMap よりも少し小さくなっています (HashMap の 16 エントリーに対して Hashtable では 11)。それ以外の主な違いは、Hashtable は null のキーと値を使用できないこと、そしてデフォルトで同期されることです。同期は必要ない場合があります、同期によってコレクションのパフォーマンスが低下します。

## Java コレクション: LinkedList

LinkedList は、List インターフェースのリンク・リスト実装です。Java Platform SE 6 API のドキュメントでは、LinkedList を以下のように説明しています。

順序付きコレクション (シーケンスとしても知られています)。このインターフェースを使用すると、要素を挿入するリスト内の位置を正確に制御することができます。要素にアクセスするには、その要素の整数インデックス (リスト内の位置) を使用します。また、リスト内の要素を検索することもできます。セットとは異なり、リストでは一般に重複要素が許可されます。

このコレクションの実装は、LinkedList\$Entry オブジェクトのリンク・リストです。図 9 に、32 ビット Java ランタイムでの LinkedList のメモリー使用量とレイアウトを示します。

図 9. 32 ビット Java ランタイムでの LinkedList のメモリー使用量とレイアウト

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.LinkedList @ 0x11624d50 Thread	24	48
java.util.LinkedList\$Link @ 0x11624d68	24	24

図 9 には、LinkedList を作成した結果、24 バイトのメモリーを使用する LinkedList オブジェクトと、単一の LinkedList\$Entry オブジェクトが作成されることが示されています。空の LinkedList の場合、メモリーの合計は 48 バイトとなります。

リンク・リストの利点の 1 つは、正確にサイズが指定されるので、サイズを変更する必要がないことです。デフォルト容量は事実上、1 つのエントリーだけです。エントリーが追加または削除されると、容量が動的に増減されます。ただし、LinkedList\$Entry オブジェクトごとのオーバーヘッドがあることには変わりありません。このオブジェクトが持つデータ・フィールドは以下のとおりです。

- Object previous
- Object next
- Object value

リンク・リストにはキー/値のペアではなく、1 つのエントリーしか格納されないことから、オーバーヘッドは HashMap や Hashtable ほどではありません。また、配列ベースの参照は使用されないため、ハッシュ値を保管する必要もありません。欠点は、リンク・リストの参照には相当な時間がかかる可能性があることです。対象のエントリーを見つけるためには、リンク・リストをトラバースしなければならないためです。したがって、リンク・リストが大きい場合には、参照するのに時間がかかる場合があります。

表 5 に、LinkedList の特性を記載します。

表 5. **LinkedList** の特性

項目	内容
デフォルト容量	1 エントリー
空の場合のサイズ	48 バイト
オーバーヘッド	24 バイト + 24 バイト/エントリー
10K のコレクションの場合のオーバーヘッド	~240K
検索/挿入/削除のパフォーマンス	O(n) — 所要時間は、要素の数が増えるのに比例して増大します。

Java コレクション: **ArrayList**

**ArrayList** は **List** インターフェースの配列実装であり、配列のサイズは変更可能です。Java Platform SE 6 API のドキュメントでは、**ArrayList** を以下のように説明しています。

順序付きコレクション (シーケンスとしても知られています)。このインターフェースを使用すると、要素を挿入するリスト内の位置を正確に制御することができます。要素にアクセスするには、その要素の整数インデックス (リスト内の位置) を使用します。また、リスト内の要素を検索することもできます。セットとは異なり、リストでは一般に重複要素が許可されます。

**LinkedList** とは異なり、**ArrayList** は **Object** の配列を使用して実装されます。図 10 に、32 ビット Java ランタイムでの **ArrayList** のメモリー使用量とレイアウトを示します。

図 10. 32 ビット Java ランタイムでの **ArrayList** のメモリー使用量とレイアウト

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.ArrayList @ 0x1fc279e0	32	88
java.lang.Object[10] @ 0x1fc27a00	56	56

図 10 には、**ArrayList** を作成すると、32 バイトのメモリーを使用する **ArrayList** オブジェクトとデフォルト・サイズ 10 の **Object** 配列が作成され、空の **ArrayList** の場合には合計 88 バイトのメモリーが使用されることが示されています。これは、**ArrayList** では正確にサイズが指定されないの、デフォルト容量が使用されることを意味します。ここでのデフォルト容量は、10 エントリーとなっています。

表 6 に、**ArrayList** の特性を記載します。

表 6. **ArrayList** の特性

項目	内容
デフォルト容量	10
空の場合のサイズ	88 バイト
オーバーヘッド	48 バイト + 4 バイト/エントリー
10K のコレクションの場合のオーバーヘッド	~40K
検索/挿入/削除のパフォーマンス	O(n) — 所要時間は、要素の数が増えるのに比例して増大します。

## その他の「コレクション」のタイプ

標準的なコレクションの他にも、コレクションと見なすことができるクラスがあります。StringBuffer は文字データを管理する点、そして構造および機能が他のコレクションと似ているという点から、コレクションと見なすことができます。Java Platform SE 6 API のドキュメントでは、StringBuffer について以下のように説明しています。

スレッド・セーフな可変の文字列・・・すべての文字列バッファーには容量があります。文字列バッファーに格納されている文字列の長さがその容量を超えない限り、新しい内部バッファー配列を割り当てる必要はありません。内部バッファーがオーバーフローすると、自動的にその容量が増加されます。

StringBuffer は char の配列として実装されます。図 11 に、32 ビット Java ランタイムでの StringBuffer のメモリー使用量とレイアウトを示します。

図 11. 32 ビット Java ランタイムでの **StringBuffer** のメモリー使用量とレイアウト

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.lang.StringBuffer @ 0x2898eb0 buffer text	24	72
char[16] @ 0x2898ec8 buffer text\u0000\u0000\u0000\u0000\u0000	48	48

図 11 には、StringBuffer を作成すると、24 バイトのメモリーを使用する StringBuffer オブジェクトと、16 エントリーのデフォルト・サイズが指定された文字配列が作成されることが示されています。空の StringBuffer の場合、合計 72 バイトのデータとなります。

コレクションと同様に、StringBuffer にも、デフォルト容量とそのサイズを変更するためのメカニズムがあります。表 7 に、StringBuffer の特性を記載します。

表 7. **StringBuffer** の特性

項目	内容
デフォルト容量	16
空の場合のサイズ	72 バイト
オーバーヘッド	24 バイト
10K のコレクションの場合のオーバーヘッド	24 バイト
検索/挿入/削除のパフォーマンス	適用外

## コレクション内の空の空間

オブジェクトの数が指定された各種コレクションのオーバーヘッドだけが、メモリー・オーバーヘッドのすべてではありません。これまでの例で計算したメモリー使用量は、コレクションのサイズが正確に指定されることを前提としていますが、ほとんどのコレクションには、この前提は当てはまりません。大抵のコレクションは、特定の初期容量で作成されてから、データが挿入されます。つまり、コレクションの容量は、そこに格納されるデータよりも大きいのが通常です。このことが、追加のオーバーヘッドをもたらします。

StringBuffer の例で考えると、そのデフォルト容量は 16 文字のエントリーで、サイズは 72 バイトです。コレクションが作成される時点では、この 72 バイトにデータは格納されません。

文字配列に、例えば `"MY STRING"` という文字を挿入する場合、これは 16 文字の配列に 9 文字を格納していることになります。図 12 に、32 ビット Java ランタイムで `"MY STRING"` を格納する `StringBuffer` のメモリー使用量を示します。

図 12. 32 ビット Java ランタイムで `"MY STRING"` を格納する `StringBuffer` のメモリー使用量

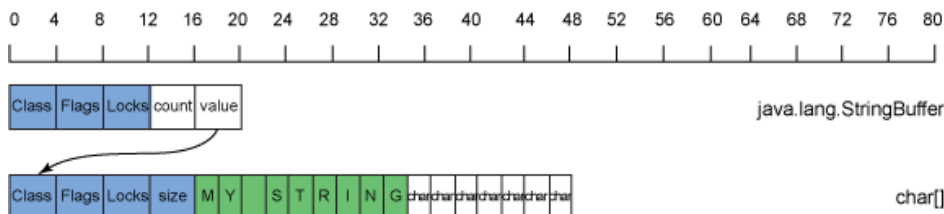


図 12 に示されているように、配列内で他に使用できる 7 文字のエントリーは使用されていませんが、これらのエントリーにもメモリーは使用されます。この例の場合、それは 112 バイトの追加オーバーヘッドに相当します。このコレクションの例では、16 エントリーの容量のうち 9 エントリーを使用しているので、充填率は 0.56 です。コレクションの充填率が低ければ低いほど、空き容量によるオーバーヘッドが増加します。

## コレクションの拡張とサイズ変更

設定された容量に達したコレクションに対して、さらにエントリーを追加するための要求が行われると、新しいエントリーを格納できるように、そのコレクションのサイズが拡張されます。これによって容量は増えますが、その一方で充填率が下がり、メモリー・オーバーヘッドが増加することになります。

使用される拡張アルゴリズムはコレクションによって異なりますが、よく使われる手法は、コレクションの容量を倍にするというものです。この手法は、`StringBuffer` でも使用されています。例として取り上げた前述の `StringBuffer` の場合、バッファに `" OF TEXT"` を追加して `"MY STRING OF TEXT"` という文字列にするとしたら、コレクションを拡張しなければなりません。現在の容量が 16 エントリーであるのに対し、新しい文字のコレクションには 17 のエントリーがあるためです。拡張後のメモリー使用量は、図 13 のようになります。

図 13. 32 ビット Java ランタイムで `"MY STRING OF TEXT"` を格納する `StringBuffer` のメモリー使用量

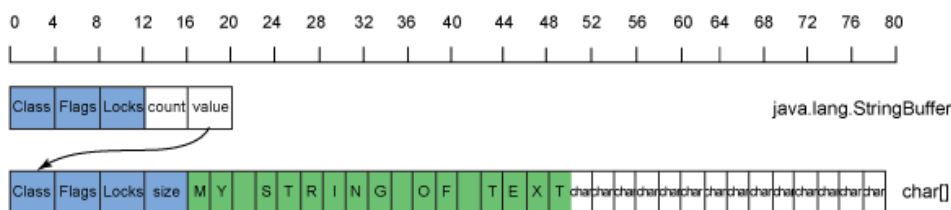


図 13 に示されているとおり、拡張後は 32 エントリーの文字配列となり、そのうち 17 のエントリーが使用されます。そのため、充填率は 0.53 に減少します。これは大幅な減少ではありませんが、空き容量のためにオーバーヘッドは 240 バイトに増えています。

サイズの小さい文字列とコレクションの場合、低い充填率と空き容量に対するオーバーヘッドは、それほど大きな問題に見えないかもしれませんが、サイズが大きくなれば、このオーバー



ヘッドはより顕著になり、コストも高くなってきます。例えば、16MB のデータだけを格納する `StringBuffer` を作成したとすると、この `StringBuffer` は (デフォルトで) 最大 32MB のデータを格納するようにサイズが設定された文字配列を使用します。したがって、空き容量という形で 16MB の追加オーバーヘッドが生じることになります。

## Java コレクション: まとめ

表 8 に、コレクションの特性を要約します。

表 8. コレクションの特性の要約

コレクション	パフォーマンス	デフォルト容量	空の場合のサイズ	10K のエントリーの場合のオーバーヘッド	正確なサイズ指定	拡張アルゴリズム
<code>HashSet</code>	$O(1)$	16	144	360K	なし	x2
<code>HashMap</code>	$O(1)$	16	128	360K	なし	x2
<code>Hashtable</code>	$O(1)$	11	104	360K	なし	x2+1
<code>LinkedList</code>	$O(n)$	1	48	240K	あり	+1
<code>ArrayList</code>	$O(n)$	10	88	40K	なし	x1.5
<code>StringBuffer</code>	$O(1)$	16	72	24	なし	x2

Hash コレクションのパフォーマンスは、どの `List` よりも遥かに優れている一方、エントリーあたりのコストは大幅に高くなります。アクセス・パフォーマンスを考えると、サイズの大きいコレクション (例えばキャッシュを実装するためのコレクションなど) を作成する場合には、追加のオーバーヘッドとは関係なく、Hash ベースのコレクションを使用するのが望ましいです。

サイズの小さいコレクションでは、アクセス・パフォーマンスは比較的小さな問題となるため、そのようなコレクションには `List` を使用するという選択肢があります。`ArrayList` コレクションと `LinkedList` コレクションのパフォーマンスはほとんど同じですが、メモリー・フットプリントは異なります。`ArrayList` は、エントリーあたりのサイズに関しては `LinkedList` を大幅に下回りますが、正確なサイズは指定されません。`List` の実装として `ArrayList` と `LinkedList` のどちらが適切であるかは、`List` の長さがどれくらいになるかをどの程度予測できるかによって左右されます。長さが不明な場合には、`LinkedList` が適切な選択肢になるでしょう。このコレクションでは、空の空間が少なくなるためです。一方、サイズが既知であるとしたら、`ArrayList` を使用した方が、メモリー・オーバーヘッドが遥かに少なくなります。

適切なコレクションのタイプを選択することで、コレクションのパフォーマンスとメモリー・フットプリントとのバランスを適切なものにすることができます。さらに、充填率が最大限になり、未使用の空間が最小限になるようにコレクションのサイズを適切に設定することにより、メモリー・フットプリントを最小限にすることができます。

## 使用されるコレクション: PlantsByWebSphere と WebSphere Application Server バージョン 7

表 8 には、10,000 のエントリーからなる Hash ベースのコレクションを作成する場合のオーバーヘッドは 360K になると示されています。複合 Java アプリケーションがギガバイト・サイズの Java ヒープで実行されることは珍しくないことを考えると、このオーバーヘッドはそれほど大き

いようには思えませんが、大量のコレクションが使用されているとなると、もちろん話は変わってきます。

表 9 に、WebSphere Application Server バージョン 7 に付属の PlantsByWebSphere サンプル・アプリケーションを 5 ユーザーの負荷テストで実行した場合、206MB の Java ヒープ使用量のうち、コレクション・オブジェクトが使用する量を記載します。

表 9. WebSphere Application Server バージョン 7 上での PlantsByWebSphere によるコレクションの使用状況

コレクションのタイプ	インスタンス数	コレクションの合計オーバーヘッド (MB)
Hashtable	262,234	26.5
WeakHashMap	19,562	12.6
HashMap	10,600	2.3
ArrayList	9,530	0.3
HashSet	1,551	1.0
Vector	1,271	0.04
LinkedList	1,148	0.1
TreeMap	299	0.03
合計	306,195	42.9

表 9 から、300,000 を超える各種のコレクションが使用されていることがわかります。さらに、206MB の Java ヒープ使用量のうち、データが格納されている部分を除いてコレクションが使用している領域は、42.9MB (21 パーセント) に上ります。これが意味するのは、コレクションのタイプを変更するか、あるいはコレクションのサイズをより正確に指定することによって、かなりのメモリーを節約できる可能性があるということです。

## Memory Analyzer による低充填率の検出

Java コレクションのメモリー使用量は、IBM Support Assistant のなかで使用可能な IBM Monitoring and Diagnostic Tools for Java - Memory Analyzer ツール (以降、Memory Analyzer とします) を使用して分析することができます (「[参考文献](#)」を参照)。Memory Analyzer には、充填率およびコレクションのサイズを分析する機能もあります。Memory Analyzer の分析を使用すれば、最適化の候補となるコレクションを特定することができます。

Memory Analyzer でコレクション分析機能を利用するには、「Open Query Browser (クエリー・ブラウザーを開く)」 -> 「Java Collections (Java コレクション)」の順にメニューを選択します (図 14 を参照)。

## 図 14. Memory Analyzer での Java コレクションの充填率分析

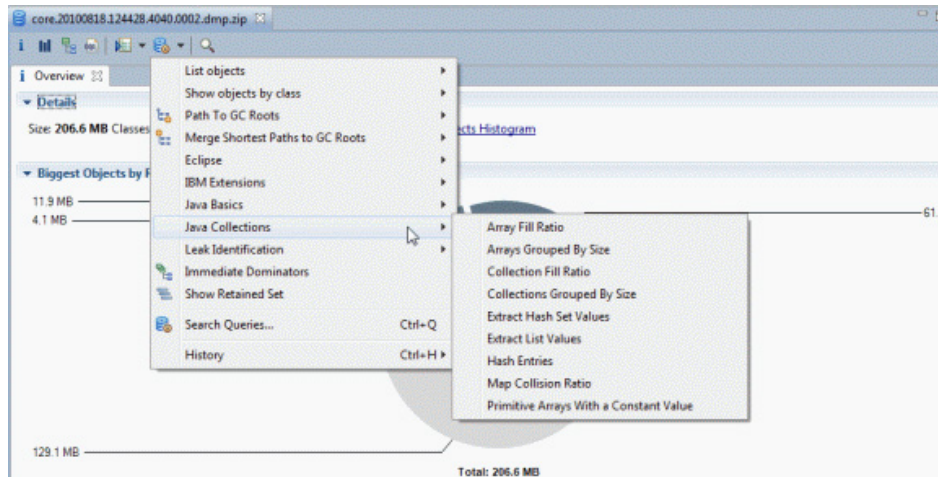


図 14 で選択されている「Collection Fill Ratio (コレクション充填率)」は、現在必要となっているサイズを大幅に上回るコレクションを突き止めるのに最も役立つクエリーです。このクエリーには、以下をはじめとするさまざまなオプションを指定することができます。

- オブジェクト: 対象とするオブジェクト (コレクション) のタイプ
- セグメント: オブジェクトの検出基準とする充填率の範囲

図 15 に、オブジェクト・オプションを「java.util.Hashtable」に設定し、セグメント・オプションを「10」に設定したクエリーを実行した場合の出力を示します。

## 図 15. Memory Analyzer での **Hashtable** の充填率分析

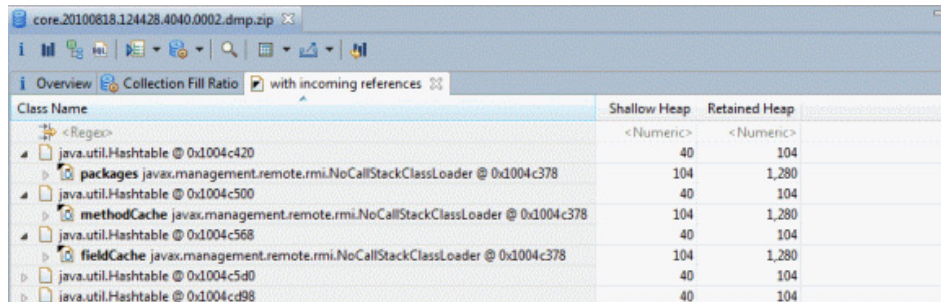
 The screenshot shows the Memory Analyzer application window with the 'Collection Fill Ratio' query results displayed. The table has four columns: 'Fill Ratio', '# Objects', 'Shallow Heap', and 'Retained Heap'. The data is grouped by fill ratio ranges from 0.00 to 0.80. The total number of entries is 262,234, with a total shallow heap of 10,489,360 and a total retained heap of 10,489,360.
 

Fill Ratio	# Objects	Shallow Heap	Retained Heap
<= 0.00	127,016	5,080,640	>= 9,903,168
<= 0.10	19,773	790,920	>= 4,342,728
<= 0.20	75,967	3,038,680	>= 9,869,153
<= 0.30	100	4,000	>= 87,648
<= 0.40	39,076	1,563,040	>= 10,935,440
<= 0.50	96	3,840	>= 316,986
<= 0.60	94	3,760	>= 562,104
<= 0.70	95	3,800	>= 565,888
<= 0.80	17	680	>= 209,816
<b>Σ Total: 9 entries</b>	<b>262,234</b>	<b>10,489,360</b>	

図 15 には、java.util.Hashtable の 262,234 のインスタンスのうち、127,016 (48.4 パーセント) が完全に空であること、そしてほぼすべてのインスタンスのエントリー数がわずかであることが示されています。

これらのコレクションが具体的にどのコレクションに当たるのかを特定するには、結果表の行を選択して右クリックし、コレクションを所有するオブジェクトを表示する場合は「list objects (オブジェクトのリスト)」->「with incoming references (参照されるオブジェクト)」の順に選択し、コレクションの内容を表示する場合は「list objects (オブジェクトのリスト)」->「with outgoing reference (参照するオブジェクト)」の順に選択します。図 16 に、空の Hashtable が参照されているオブジェクトを調べた結果を、いくつかのエントリーが展開された状態で示します。

## 図 16. Memory Analyzer での空の **Hashtable** が参照されているオブジェクトの分析



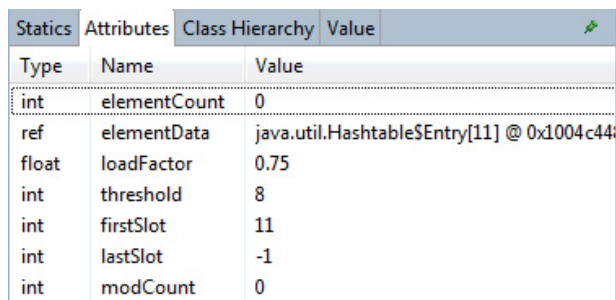
Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.Hashtable @ 0x1004c420	40	104
packages javax.management.remote.rmi.NoCallStackClassLoader @ 0x1004c378	104	1,280
java.util.Hashtable @ 0x1004c500	40	104
methodCache javax.management.remote.rmi.NoCallStackClassLoader @ 0x1004c378	104	1,280
java.util.Hashtable @ 0x1004c568	40	104
fieldCache javax.management.remote.rmi.NoCallStackClassLoader @ 0x1004c378	104	1,280
java.util.Hashtable @ 0x1004c5d0	40	104
java.util.Hashtable @ 0x1004cd98	40	104

図 16 から、空の **Hashtable** のいくつか

は、`javax.management.remote.rmi.NoCallStackClassLoader` コードが所有していることがわかります。

Memory Analyzer の左側のパネルの「Attributes (属性)」ビューを調べると、**Hashtable** についての具体的な詳細がわかります (図 17 を参照)。

## 図 17. Memory Analyzer での空の **Hashtable** の検査



Type	Name	Value
int	elementCount	0
ref	elementData	java.util.Hashtable\$Entry[11] @ 0x1004c44
float	loadFactor	0.75
int	threshold	8
int	firstSlot	11
int	lastSlot	-1
int	modCount	0

図 17 には、**Hashtable** のサイズは 11 (デフォルト・サイズ) となっているのに、このコレクションには何も含まれていないことが示されています。

`javax.management.remote.rmi.NoCallStackClassLoader` コードの場合、コレクションの使用は以下の方法によって最適化できる可能性があります。

- **Hashtable 割り当ての遅延:** **Hashtable** が空になることがよくある場合は、そこに格納するデータがあるときにだけ **Hashtable** を割り当てるのが理に適っています。
- **正確なサイズに従った Hashtable の割り当て:** デフォルト・サイズが使用されていたため、より正確な初期サイズを使用できる可能性があります。

上記の最適化のいずれか、あるいは両方を適用できるかどうかは、このコードが一般的にどのように使用されているか、そして通常どのデータが保管されるかによって決まります。

## PlantsByWebSphere サンプルでの空のコレクション

表 10 に、空のコレクションを識別するために PlantsByWebSphere サンプルでのコレクションを分析した結果を示します。



表 10. WebSphere Application Server バージョン 7 上の PlantsByWebSphere での空のコレクションの使用状況

コレクションのタイプ	インスタンス数	空のインスタンス	空の割合
Hashtable	262,234	127,016	48.4
WeakHashMap	19,562	19,465	99.5
HashMap	10,600	7,599	71.7
ArrayList	9,530	4,588	48.1
HashSet	1,551	866	55.8
Vector	1,271	622	48.9
合計	304,748	160,156	52.6

表 10 には、平均で 50 パーセントを上回るコレクションが空になっていることが示されています。これは、コレクションの使用を最適化することによって、かなりのメモリー・フットプリントを節約できる可能性があることを意味します。最適化を適用するアプリケーションのレベルとしては、PlantsByWebSphere サンプル・コード、WebSphere Application Server、そして Java コレクション・クラス自体が考えられます。

WebSphere Application Server バージョン 7 からバージョン 8 にアップグレードする際に、Java コレクションおよびミドルウェア層でメモリー効率を高めるための取り組みが行われました。例えば、`java.util.WeakHashMap` のインスタンスに伴うオーバーヘッドのうち、かなりの部分は弱参照を扱う `java.lang.ref.ReferenceQueue` のインスタンスが含まれていることが原因となっています。図 18 に、32 ビット Java ランタイムでの `WeakHashMap` のメモリー・レイアウトを示します。

図 18. 32 ビット Java ランタイムでの `WeakHashMap` のメモリー・レイアウト

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.WeakHashMap @ 0x1fcf6d30	48	688
<class> class java.util.WeakHashMap @ 0x1007f748 System Class	8,863	8,863
elementData java.util.WeakHashMap\$Entry[16] @ 0x1fcf6d60	80	80
referenceQueue java.lang.ref.ReferenceQueue @ 0x1fcf6db0	32	560
Σ Total: 3 entries		

図 18 には、`WeakHashMap` が空で、`ReferenceQueue` が必要ないとしても、`ReferenceQueue` オブジェクトに 560 バイト相当のデータを保持する役割があることが示されています。19,465 の空の `WeakHashMap` を持つ PlantsByWebSphere サンプルの場合、`ReferenceQueue` オブジェクトは必要のない 10.9MB のデータを追加していることとなります。WebSphere Application Server バージョン 8 と IBM Java ランタイムの Java 7 リリースでは、`WeakHashMap` の最適化が行われ、この `WeakHashMap` に含まれる `ReferenceQueue` に `Reference` オブジェクトの配列が格納されるようになっていきます。その配列は、遅延して割り当てられるように変更されました。つまり、オブジェクトが `ReferenceQueue` に追加されるときにだけ割り当てられます。

## まとめ

どのアプリケーションにも、大きなサイズのコレクションが、しかも驚くほどの数で存在します。それは、複合アプリケーションともなれば尚更のことです。大量のコレクションが使用され



ている場合、そこに、大幅なメモリー・フットプリントの削減を達成する余地があることは珍しくありません。その方法は、正しいコレクションを選択して適切なサイズに設定し、場合によっては遅延して割り当てることです。このような最適化を行うかどうかは、設計と開発の段階で決定するのが最善ですが、Memory Analyzer ツールを使用して既存のアプリケーションを分析し、潜在的なメモリー・フットプリントの最適化を調べることもできます。

---

## 著者について

Chris Bailey



Chris Bailey は、英国の Hursley Park Development Lab を拠点とする IBM JTC (Java Technology Center) チームの一員です。IBM Java サービスおよびサポート組織でテクニカル・アーキテクトを務める彼は、IBM SDK のユーザーをアプリケーション・デプロイメントの成功に導くことを任務としています。また、新しい要件の収集と評価、新しいデバッグ機能とツールの実現、資料の改善、そして IBM SDK for Java の全体的な品質改善にも携わっています。

© Copyright IBM Corporation 2012

([www.ibm.com/legal/copytrade.shtml](http://www.ibm.com/legal/copytrade.shtml))

商標

([www.ibm.com/developerworks/jp/ibm/trademarks/](http://www.ibm.com/developerworks/jp/ibm/trademarks/))