

## ダンプからデバッグする

Memory Analyzer を使って、メモリー・リークに限らず、さまざまな問題を診断する

[Chris Bailey](#)

Java Support Architect  
IBM

2011年 3月 15日

[Andrew Johnson](#)

Java Tools Developer and Eclipse Memory Analyzer Tool  
Committer  
IBM

[Kevin Grigorenko](#)

Software Engineer, WebSphere Application Server SWAT  
Team  
IBM

Memory Analyzer は、Java™ プロセスのダンプからメモリー・リークやメモリー・フットプリントの問題を診断する強力なツールです。しかも、診断用のコードを組み込まなくても Java コードの実態を詳細に入手できるため、たった 1 つのダンプから難解な問題のデバッグをすることができます。この記事では、ダンプを生成する方法、そして生成したダンプを使ってアプリケーションの状態を調べる方法を学びます。

### Memory Analyzer のバリエーション

IBM Monitoring and Diagnostic Tools for Java - Memory Analyzer は、IBM DTFJ (Diagnostic Tool Framework for Java) を使用して Eclipse MAT (Memory Analyzer Tool) バージョン 1.0 を拡張することによって、MAT に組み込まれている診断機能を IBM Virtual Machines for Java でも使えるようにします。DTFJ は、オペレーティング・システム・レベルのダンプと IBM Portable Heap Dump を使用して Java ヒープの分析を行います。IBM 製品向けの Memory Analyzer は、ISA (IBM Support Assistant) に組み込まれています。スタンドアロンの Eclipse MAT には DTFJ プラグインを使用することができます。ダウンロード・リンクについては、「[参考文献](#)」を参照してください。

デバッグ用のステートメントをコードに追加して、オブジェクトのフィールドや、さらにはデータ・コレクション全体を書き出すという方法は、一般的に用いられる問題解決手法です。多くの場合、問題を理解して解決するために追加の情報が必要であることがわかるたびに、デバッグ用

のステートメントを追加するという作業を繰り返すことになります。このプロセスは効果的かもしれませんが、実を結ばないこともあります。デバッグ用コードのボリュームが増えることで問題が隠れてしまったり、自分が所有していないコードにデバッグ用のコードを追加しなければならなかったりする場合がありますからです。また、デバッグするためにプロセスを最初からやり直さなければならないことや、デバッグによるパフォーマンス全体への影響により、アプリケーションが動作しなくなることもあります。

Memory Analyzer は、メモリー問題の診断に使えるだけでなく、Java アプリケーション全体の状態と振る舞いに関する詳細な情報を得るためにも使用できる、クロスプラットフォームのオープンソース・ツールです。アプリケーションの実行中に Java ランタイムが作成するスナップショット・ダンプを読み込むことによって、Memory Analyzer はデバッグ用のコードでは明らかにできないような難解な問題を診断する手段を提供します。

この記事では、ダンプを生成する方法、そして生成したダンプを使ってアプリケーションの状態を検査および診断する方法について説明します。Memory Analyzer を使用すれば、スレッド、オブジェクト、バンドル、そしてデータ・コレクション全体を検査して、メモリー・リークに限らず、さまざまな Java コードの問題をデバッグできるようになります。

## スナップショット・ダンプのタイプ

現在、Memory Analyzer で扱うことのできるダンプ形式には、以下の 3 つのタイプがあります。

- **IBM PHD (Portable Heap Dump):** この IBM 独自仕様のダンプ形式の出力には、プロセスで使用する各 Java オブジェクトの型とサイズ、およびオブジェクト間の関係のみが含まれます。このダンプ形式の出力は、他の形式に比べると大幅にサイズが小さく、含めることのできる情報量も最も少なくなっています。しかし、メモリー・リークの診断や、アプリケーションのアーキテクチャーおよびメモリー・フットプリントに関する基本的な情報を得るには、通常はこのダンプ出力に含まれるデータで十分です。
- **HPROF バイナリー・ダンプ:** HPROF バイナリー・ダンプ形式の出力には、IBM PHD 形式のダンプ出力に含まれるすべてのデータに加え、Java オブジェクト内に保持されるプリミティブ型データ、およびスレッドの詳細が含まれます。オブジェクト内のフィールドに保持されているデータの値を調べることで、ダンプの取得時に実行されていたメソッドが具体的にわかります。プリミティブ型データが追加で含まれることから、HPROF 形式のダンプ出力は PHD 形式のダンプ出力に比べ、かなりの大きさになります。そのサイズは、使用される Java ヒープのサイズにほぼ匹敵します。
- **IBM システム・ダンプ:** IBM Java ランタイムの使用中には、オペレーティング・システムのネイティブ・ダンプ・ファイル (AIX® または Linux® の場合はコア・ファイル、Windows® の場合はミニダンプ、z/OS® の場合は SVC ダンプ) を Memory Analyzer にロードすることができます。これらのダンプには、実行中のアプリケーションのメモリー・イメージ全体が含まれます。つまり、HPROF 形式のダンプ出力に含まれるすべての情報とデータに加え、ネイティブ・メモリーおよびスレッドに関するすべての情報が含まれます。この形式のダンプ出力は、3 つの形式のなかで最も大きく、最も包括的なものになっています。

上記のうち、IBM PHD と IBM システム・ダンプは、DTFJ (Diagnostic Tool Framework for Java) プラグインがインストールされていなければ使用することができません (「[参考文献](#)」および囲み記事「[Memory Analyzer のバージョン](#)」を参照)。

表 1 に、ダンプ形式による違いを要約します。

表 1. 各ダンプ形式の特徴の要約

ダンプ形式	ディスク上のサイズ (概算)	オブジェクト、クラス、およびクラスローダー	スレッド詳細	フィールド名	フィールドおよび配列参照	プリミティブ型フィールド	プリミティブ型配列の要素	正確なガーベッジ・コレクションのルート	ネイティブ・メモリーおよびネイティブ・スレッド
IBM PHD	Java ヒープ・サイズの 20 パーセント	○	○ (javacore を使用)*	×	○	×	×	×	×
HPROF	Java ヒープ・サイズ	○	○	○	○	○	○	○	×
IBM システム・ダンプ	Java ヒープ・サイズ + 30 パーセント	○	○	○	○	○	○	○	○

\*Memory Analyzer は同時に生成された javacore.txt ファイル (IBM スレッド・ダンプ・ファイル) と heapdump.phd ファイルをロードすることによって、IBM PHD 形式のダンプ出力にスレッドの詳細を含めます。

HPROF 形式と IBM システム・ダンプ形式は、どちらも圧縮率が優れています。オペレーティング・システムのツールを使用して圧縮した場合、通常は元のサイズの 20 パーセント程度になります。

## スナップショット・ダンプの取得

Java ランタイムごとに多種多様なダンプを取得できるように各種のメカニズムが用意されているため、`OutOfMemoryError` に関連するシナリオに限らず、さまざまなシナリオに応じて柔軟にスナップショット・ダンプを生成することができます。使用できるメカニズムは、使用しているベンダーの Java ランタイムによって異なります。

### 前提条件

どの形式のダンプを取得するかに関わらず、ダンプが途中で切り捨てられないように、ダンプ用のディスク・スペースが十分にあることを確認してください。ダンプの保存先となるデフォルト・ロケーションは、JVM プロセスのカレント作業ディレクトリーです。デフォルト・ロケーションを変更するには、IBM JVM の場合、`-Xdump` コマンドライン・オプションを使って保存先を指定し、HotSpot JVM の場合も同様に `-XX:HeapDumpPath` を使って保存先を指定します。これらのオプションの構文については、「[参考文献](#)」のリンクを参照してください。

オペレーティング・システムからのダンプは、IBM JVM と HotSpot JVM の両方で使用することができます。IBM JVM の場合、JDK に付属の `jextract` ツールを使ってダンプを生成し、Memory Analyzer に直接ロードすることができます。HotSpot JVM の場合、`jmap` ツールを使ってコア・ダンプからヒープ・ダンプを抽出します (それぞれの方法については、記事の後のほうで詳しく説明します)。ただし、一部のオペレーティング・システムでは、実行しているプロセスに十分なりソースが割り当てられていることを、`ulimit` を使って確認してからコア・ダンプを生成してくだ

さい。これは、コア・ダンプが途中で切り捨てられて、分析が制限されることがないようにするために必要な作業です。ulimit で確認した値が適切ではない場合には、それらの値を変更してからプロセスを再起動し、その上でダンプを収集します。AIX、Linux®、z/OS、および Solaris からシステム・ダンプを取得する方法についての詳細は、「[参考文献](#)」に記載されているリンクを参照してください。

## スナップショット・ダンプの取得方法: HotSpot ランタイムの場合

HotSpot ベースの Java ランタイムが生成するダンプは、HPROF 形式のダンプのみです。ダンプを生成する方法は、いくつかの対話型の方法とイベント・ベースの 1 つの方法のなかから選択することができます。

- 対話型の方法:

- Ctrl+Break を使用: 実行中のアプリケーションに対して `-XX:+HeapDumpOnCtrlBreak` コマンドライン・オプションを指定すると、HPROF 形式のダンプと併せ、Ctrl+Break イベントまたは SIGQUIT (通常は `kill -3` によって生成) がコンソールを介して送信されたときにスレッド・ダンプが生成されます。このオプションは、一部のバージョンでは指定することができません。その場合には、以下のコマンドを試してください。

```
-Xrunhprof:format=b,file=heapdump.hprof
```

- jmap ツールを使用: JDK の bin ディレクトリーに提供されている jmap ユーティリティー・ツール ([参考文献](#) を参照) には、実行中のプロセスの HPROF 形式のダンプを要求するオプションがあります。Java 5 では、以下のコマンドを使用します。

```
jmap -dump:format=b pid
```

Java 6 では以下のコマンドを使用します。ここで、live はオプションです。このオプションを指定すると、「ライブ」オブジェクトのみがダンプ・ファイルのプロセス ID (PID) に書き込まれます。

```
jmap -dump[live,]format=b,file=filename pid
```

- オペレーティング・システムを使用: 非破壊的な gcore コマンドを使用するか、あるいは破壊的な `kill -6` または `kill -11` コマンドを使用して、コア・ファイルを生成します。そして生成されたコア・ファイルから、jmap を使用してヒープ・ダンプを抽出します。

```
jmap -dump:format=b,file=heap.hprof path to java executable core
```

- JConsole ツールを使用: JConsole の HotSpotDiagnostic MBean には、dumpHeap 処理が提供されています。この処理によって、HPROF ダンプの生成が要求されます。

- イベント・ベースの方法:

- OutOfMemoryError** イベント発生時: 実行中のアプリケーションに対して `-XX:+HeapDumpOnOutOfMemoryError` コマンドライン・オプションを指定すると、OutOfMemoryError の発生時に HPROF 形式のダンプが生成されます。本番システム用には、この方法を用意しておくのが理想的です。というのも、メモリーの問題を診断する際には必ずといってよいほど、この方法が必要とされますが、この方法の場合、パフォーマンスのオーバーヘッドを継続的に生じることがないからです。HotSpot ベースの Java ランタイムの古いリリースでは、JVM の実行ごとに、この OutOfMemoryError イベントによるヒープ・ダンプをいくつでも生成することができますが、比較的新しいリリースでは、このイベントで生成されるヒープ・ダンプの数は、JVM の 1 回の実行あたり 1 つに限られています。

## スナップショット・ダンプの取得方法: IBM ランタイムの場合

IBM ランタイムでは、さまざまな対話型のシナリオとイベント・ベースのシナリオで PHD 形式のダンプまたはシステム・ダンプを生成することができるダンプ・エンジンとトレース・エンジンを提供しています。さらに、対話型のダンプを生成するには、Health Center ツールを使用することも、Java API を使ったプログラムで行うこともできます。

- 対話型の方法:

- **SIGQUIT** または **Ctrl+Break** を使用: Ctrl+Break または SIGQUIT (通常は `kill -3` によって生成されます) が IBM ランタイムに送信されると、IBM ダンプ・エンジンにユーザー・イベントが生成されます。デフォルトでは、このイベントが生成するのはスレッド・ダンプ・ファイル (`javacore.txt`) のみですが、`-Xdump:heap:events=user` オプションを使用することで、PHD 形式のダンプを生成することができます。あるいは `-Xdump:system:events=user` オプションを使用して Java アプリケーションのシステム・ダンプを生成することもできます。
- **オペレーティング・システムを使用してシステム・ダンプを生成:**
  - AIX の場合: `gencore` (あるいは破壊的な `kill -6` または `kill -11`)
  - Linux/Solaris の場合: `gcore` (あるいは破壊的な `kill -6` または `kill -11`)
  - Windows の場合: `userdump.exe`
  - z/OS の場合: `SVCDUMP` またはコンソール・ダンプ
- **IBM Monitoring and Diagnostics Tools for Java - Health Center** を使用: Health Center ツールには、実行中の Java プロセスから PHD ダンプやシステム・ダンプを要求するメニュー・オプションがあります (「[参考文献](#)」を参照)。
- **イベント・ベースの方法:** IBM ダンプ・エンジンおよびトレース・エンジンには、実行中のメソッドにスローされた例外から発生する多数のイベントをトリガーとして PHD ダンプやシステム・ダンプを生成するための柔軟な機能が揃っています。これらの機能を使用すれば、問題が発生していて診断の対象とするシナリオのほとんどでダンプを生成することができます。
- **IBM ダンプ・エンジンを使用:** ダンプ・エンジンが提供するさまざまなイベントをトリガーとして PHD ダンプやシステム・ダンプを生成することができます。さらに、ダンプ・エンジンでは、イベント・タイプを基準としたフィルタリングによって、ダンプを生成する条件をきめ細かく制御することができます。  
`-Xdump:what` オプションを使用すると、デフォルト・イベントが表示されます。すると、例えば JVM での最初の 4 つの `OutOfMemoryError` 例外によって `heapdump.phd` と `javacore.txt` が生成されることがわかります。  
さらに多くのデータを収集するには、以下の方法で、`OutOfMemoryError` 例外の発生時にヒープ・ダンプではなく、システム・ダンプを生成します。

```
-Xdump:heap:none -Xdump:java+system:events=systhrow,  
filter=java/lang/OutOfMemoryError,range=1..4,request=exclusive+compact+prewalk
```

例外のなかには、さまざまなコードによって、ほとんどのアプリケーションで共通して生成されるものがあります。その一例として `NullPointerException` が挙げられますが、こうした例外では、その例外が発生するさまざまな状況の中で、ある特定の状況の場合においてのみダンプを生成するとなると、そう簡単には行きません。そこで、どの状況でダンプを生成するのかをもう少し具体的に指定するために、「throw」イベントと「catch」イベントに対してフィルタリング・レベルを追加で指定できるようになって



います。このフィルタリング・レベルによって、どのメソッドで「throw」イベント、もしくは「catch」イベントが発生したときに、ダンプを生成するのかを指定することができます。この指定をする場合、filter の値には例外名に続けて # 区切り文字を追加し、その後にダンプの生成を行うトリガーとなるメソッド名を続けるようにします。例えば以下のようにオプションを指定すると、bad() メソッドが `NullPointerException` をスローした場合にシステム・ダンプを生成します。

```
-Xdump:system:events=throw,  
    filter=java/lang/NullPointerException#com/ibm/example/Example.bad
```

以下のようにオプションを指定した場合は、catch() メソッドが `NullPointerException` をキャッチした場合に、システム・ダンプを生成します。

```
-Xdump:system:events=catch,  
    filter=java/lang/NullPointerException#com/ibm/example/Example.catch
```

イベントに対するフィルタリングに加え、ダンプを生成するイベントの範囲を指定することもできます。例えば、以下のオプションは、5 番目に発生した `NullPointerException` のみ、ダンプを生成します。

```
-Xdump:system:events=throw, filter=java/lang/NullPointerException,range=5
```

以下のオプションでは、2 番目、3 番目、および 4 番目に発生した `NullPointerException` のみダンプを生成するように範囲を指定しています。

```
-Xdump:system:events=throw, filter=java/lang/NullPointerException,range=2..4
```

表 2 に、最もよく使われるイベントとフィルターについて要約します。

表 2. 使用可能なダンプ・イベント

イベント	説明	使用可能なフィルタリング条件	例
gpf	一般保護違反 (クラッシュ)		-Xdump:system:events=gpf
user	ユーザー生成シグナル (SIGQUIT または Ctrl+Break)		-Xdump:system:events=user
vmstop	System.exit() の呼び出しを含む、VM シャットダウン	終了コード	-Xdump:system:events=vmstop,filter=#0..#10 VM シャットダウンの終了コードが 0 から 10 までの場合に、システム・ダンプを生成します。
load	クラスのロード	クラス名	-Xdump:system:events=load,filter=com/ibm/example/Example com.ibm.example.Example クラスがロードされた場合に、システム・ダンプを生成します。
unload	クラスのアンロード	クラス名	-Xdump:system:events=unload,filter=com/ibm/example/Example com.ibm.example.Example クラスがアンロードされた場合に、システム・ダンプを生成します。
throw	スローされた例外	例外クラス名	-Xdump:system:events=throw,filter=java/net/ConnectException ConnectException が生成された場合に、システム・ダンプを生成します。
catch	キャッチされた例外	例外クラス名	-Xdump:system:events=catch,filter=java/net/ConnectException ConnectException がキャッチされた場合に、システム・ダンプを生成します。
systhrow	JVM による Java 例外のスロー (このイベントは JVM 内部でエラー条件が検出された場合にのみトリガーされるため、throw イベントとは異なります。)	例外クラス名	-Xdump:system:events=systhrow,filter=java/lang/OutOfMemoryError OutOfMemoryError が生成された場合に、システム・ダンプを生成します。
allocation	Java オブジェクトの割り当て	割り当てられるオブジェクトのサイズ	-Xdump:system:events=allocate,filter=#5m 5MB を超えるサイズのオブジェクトが割り当てられた場合に、システム・ダンプを生成します。

- IBM トレース・エンジンを使用: トレース・エンジンでは、アプリケーションで実行中の任意の Java メソッドの入り口または出口で PHD ダンプやシステム・ダンプをトリガーすることができます。それには、IBM トレース・エンジンを制御する `-xtrace` コマンドライン・オプションに `trigger` キーワードを追加します。トリガー・オプションの構文は以下のとおりです。

```
method{methods[,entryAction[,exitAction[,delayCount[,matchcount]]]]}
```

以下のコマンドライン・オプションをアプリケーションに追加すると、`Example.trigger()` メソッドが呼び出されたときにシステム・ダンプが生成されます。

```
-Xtrace:maximal=mt,trigger=method{com/ibm/example/Example.trigger,sysdump}
```

以下のコマンドライン・オプションは、`Example.trigger()` メソッドが呼び出されると、PHD ダンプを生成します。

```
-Xtrace:maximal=mt,trigger=method{com/ibm/example/Example.trigger,heapdump}
```

一方、メソッドが呼び出されるたびにダンプが生成されないようにするために、ダンプ生成の条件となる範囲を設定することをお勧めします。以下の例は、`Example.trigger()` の最初の 5 回の呼び出しを無視してからダンプを 1 回トリガーします。

```
-Xtrace:maximal=mt,trigger=method{com/ibm/example/Example.trigger,sysdump,,5,1}
```

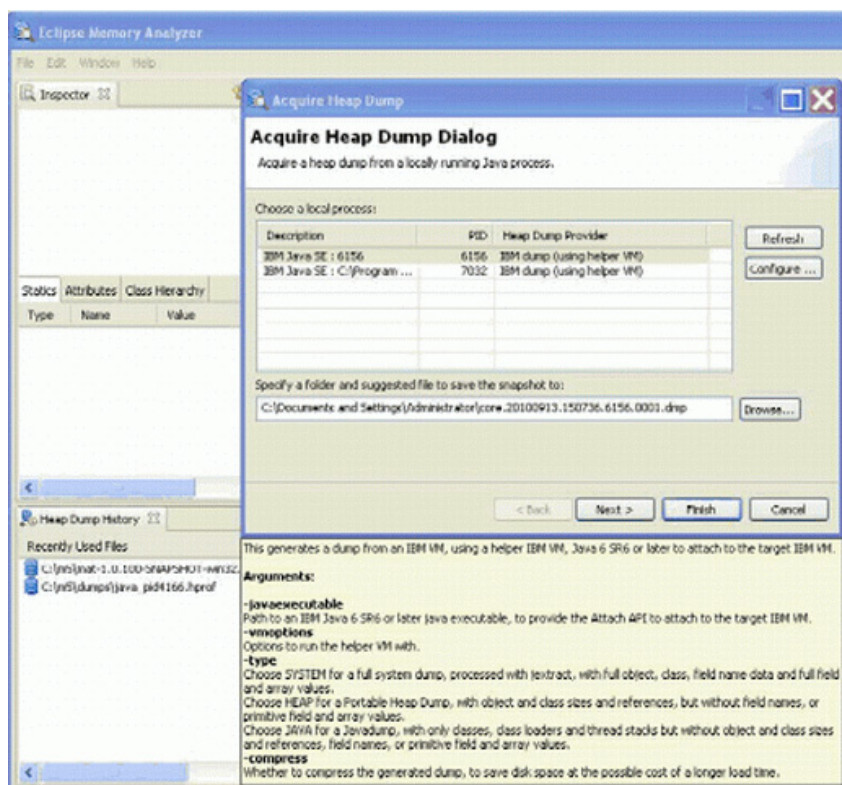
上記の例では、`exitAction` に対する指定が空になっていることに注意してください。これは、メソッドの入り口でのみダンプをトリガーしているためです。

- ・ **プログラムによる手法:** IBM ランタイムでは、`javaDump()`、`heapDump()`、および `systemDump()` メソッドを持つ `com.ibm.jvm.Dump` クラスも提供しています。これらのメソッドは、それぞれスレッド・ダンプ、PHD ダンプ、システム・ダンプを生成します。

## Memory Analyzer を使用してダンプを取得する

ランタイム自体が提供するダンプを取得するためのメソッドに加え、Memory Analyzer でも「Acquire Heap Dump (ヒープ・ダンプの取得)」オプション (図 1 を参照) を使用して、Memory Analyzer と同じマシンで実行されている Java プロセスのスナップショット・ダンプをトリガーしてロードすることができます。

### 図 1. Memory Analyzer のヒープ・ダンプ取得機能を使用する



HotSpot ベースのランタイムでは、Memory Analyzer は `jmap` を使用してダンプを生成します。IBM ランタイムの場合、ダンプを生成する手段には Java 遅延接続機能とプログラムから呼び出す API が使用されます。Java 遅延接続機能を使用するには、Java 6 SR6 が必要です。これより前のリリースには、この機能が含まれていません。



## 事後処理に関する要件

IBM システム・ダンプについては、JDK に付属の `jextract` ツールを使用して事後処理する必要があります。

`jextract core`

できれば、java プロセスを実行するために使用していたライブラリーに対する読み取りアクセス権を設定し、ダンプを生成した JDK インストールに提供されている `jextract` を、ダンプを生成した物理マシンで実行してください。`jextract` がダンプの処理に大量の CPU サイクルを使用するとしたら、この条件を許容できない本番システムもあります。その場合には、例えば本番前のテスト・システムなど、本番システムとできるだけよく似たシステムで実行してください。ただし、Java ランタイムの SR (Service Refresh) バージョンと FP (Fix Pack) バージョンは同じでなければなりません。

`jextract` が生成する ZIP ファイルには、元のコア・ダンプ、処理後のダンプ表現、Java 実行可能ファイル、および java プロセスで使ったライブラリーが含まれます。`jextract` を実行した後は、元の (圧縮されていない) コア・ダンプを削除しても構いません。Memory Analyzer にロードしなければならないのは、生成された ZIP ファイルのほうです。

`jextracted` システム・ダンプから PHD ダンプを抽出するには、ZIP ファイルを `jdmview` にロードして、`heapdump` コマンドを実行します (「[参考文献](#)」を参照)。

## Memory Analyzer を使って問題を分析する

Memory Analyzer では、アプリケーションでメモリー・リークが発生している領域、または使用可能なメモリーに対してアプリケーションに必要なメモリー・フットプリントが大きすぎる領域を探すことによって、`OutOfMemoryError` を診断することができます。Memory Analyzer は自動リーク検出を行い、「Leak Suspects (リーク診断)」レポートを生成します (「[参考文献](#)」を参照)。

HPROF ダンプおよび IBM システム・ダンプで使用可能な追加データ (特にフィールド名とフィールド値) も、「Inspector (インスペクター)」ビューと OQL (Object Query Language) の機能を使うことによって、「何がメモリーを使い果たしているのか」という問題に限らず、広範な種類の問題を診断することを可能にします。例えば、コレクションの占有率と負荷係数を突き止めて、コレクションが効率的なサイズであるかどうかを調べたり、`ConnectException` に関連付けられたホスト名とポート番号を調べて、アプリケーションが試行していた接続を確認したりすることができます。

## インスペクターを使ってオブジェクトに含まれるフィールドを調べる

Memory Analyzer で任意のオブジェクトを選択すると、「Inspector (インスペクター)」ビューに、オブジェクトのクラス階層、属性、静的フィールドを含め、そのオブジェクトに関して入手可能な情報が表示されます。このビューの「Attributes (属性)」パネルには、オブジェクトと関連付けられたインスタンスのフィールドと値が表示され、「Statics (静的フィールド)」パネルには、クラスと関連付けられた静的フィールドとそれらの値が表示されます。

図 2 に示す、単純な `java.net.URL` オブジェクトの「Inspector (インスペクター)」ビューを見ると、この URL のプロトコルの種類や宛先など、オブジェクトに関する詳細がわかります。

図 2. 「インスペクター」ビューの「静的フィールド」パネル、「属性」パネル、および「クラス属性」パネル

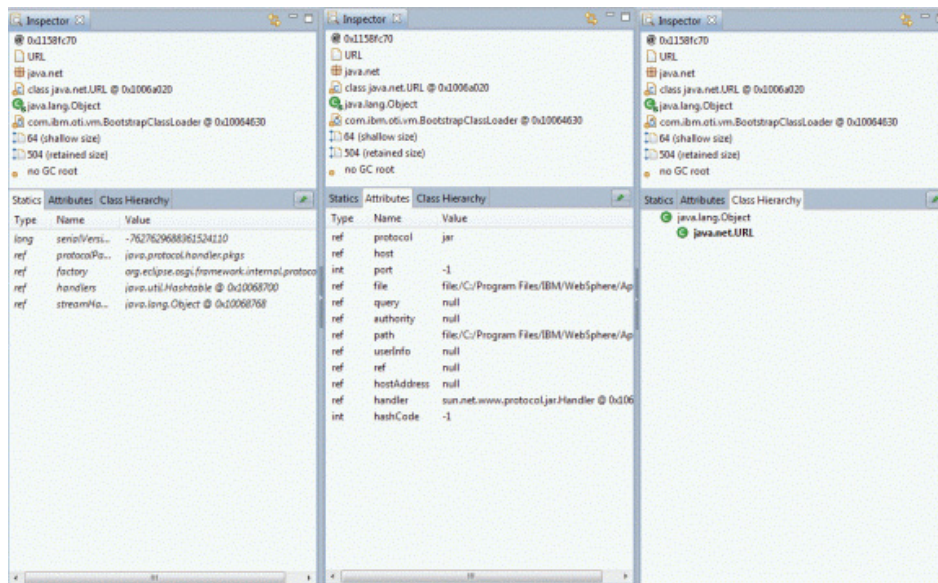


図 2 の「Attributes (属性)」パネルを見ると、URL オブジェクトは (path および file フィールドに指定された場所にある) ローカル・ファイルシステムに置かれた JAR ファイル (protocol フィールド) を参照していることがわかります。

## OQL を使用してオブジェクトに対するクエリーを実行する

OQL を使用して、SQL のようなカスタム・クエリーでダンプに対してクエリーを実行することができます。このトピックは、それ自体で 1 本の記事が書けるほどなので、ここでは例をいくつか紹介するだけにとどめます。詳細については、Memory Analyzer に組み込まれた OQL に関するヘルプの内容を参照してください。

OQL が特に役立つのは、一連のオブジェクトの外部参照とフィールドのパスを辿って、特定のフィールドを突き止める場合です。例えば、クラス A に B という型の `foo` というフィールドがあり、クラス B に `String` 型の `bar` というフィールドがあるとします。その場合、以下の単純なクエリーで、この `String` 型の `bar` をすべて抽出することができます。

```
SELECT aliasA.foo.bar.toString()
FROM A aliasA
```

上記では、クラス A に `aliasA` という別名を指定して、`SELECT` 節でこの別名を参照しています。このクエリーは、クラス A のインスタンスからしか抽出しません。クラス A のすべてのインスタンスからだけでなく、すべてのサブクラスからも抽出するとしたら、以下のクエリーを使用します。

```
SELECT aliasA.foo.bar.toString()
FROM INSTANCEOF A aliasA
```

以下は、さらに複雑な `DirectByteBuffer` を使用した例です。

```
SELECT k, k.capacity
FROM java.nio.DirectByteBuffer k
WHERE ((k.viewedBuffer=null)and(inbounds(k).length>1))
```

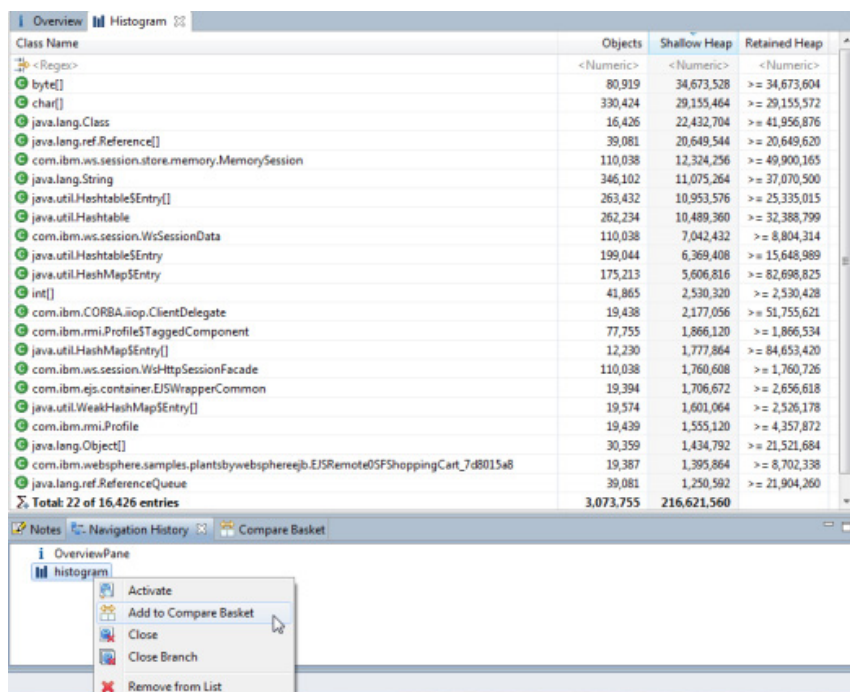
この例で取得しようとしているのは、`DirectByteBuffer` の `capacity` フィールドです。このフィールドによって、このオブジェクトが保持するネイティブ・メモリーがわかります。この例ではさらに、値が `null` の `viewedBuffer` フィールド (他の `DirectByteBuffer` のビューに過ぎないフィールド) と複数のインバウンド参照を持つ `DirectByteBuffer` をフィルタリングで除外します。これにより、まだクリーンアップが完了していないオブジェクトをファントム参照によって調べてしまうことがなくなります (つまり、この例では「ライブ」`DirectByteBuffer` だけを必要としているのです)。

## ビューまたはダンプの比較を実行する

Memory Analyzer では、クエリーによって生成されたテーブルを比較することができます。比較するテーブルとしては、あるビューの `String` オブジェクトが別のビューのコレクション・オブジェクト内に存在するかどうかを調べるために 1 つのダンプから生成されたテーブルを選択することも、例えばオブジェクトのコレクション数の増加などといったデータの変更を調べるために複数のダンプから生成されたテーブルを選択することもできます。

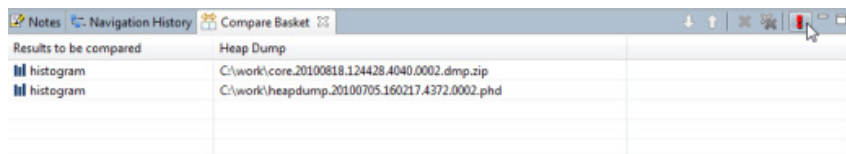
比較を実行するには、該当するテーブルを「Compare Basket (比較バスケット)」に追加してから、バスケット内の比較対象とするエントリーを要求します。それにはまず、「Navigation History (ナビゲーション履歴)」でテーブルのエントリーを見つけて選択し、コンテキスト・メニューから「Add to Compare Basket (比較バスケットに追加)」を選択します (図 3 を参照)。

図 3. 「ナビゲーション履歴」ビューから「比較バスケット」へのテーブルの追加



「Compare Basket (比較バスケット)」に2つのエントリーを追加した後は、パネルの右上隅にある「Compare the results (結果の比較)」ボタン (赤い感嘆符) をクリックすれば比較を実行することができます (図 4 を参照)。

図 4. 比較バスケットに入れられたエントリーの結果の比較



## メモリー・フットプリントとメモリーの効率性

Memory Analyzer の重要な用途の1つとして、メモリー・リークが発生していないとしても、ヒープの大部分を使用しているコンポーネントを見つけることが挙げられます。メモリー使用量を減少させることができるとすれば、システムの容量やパフォーマンスを改善して、セッション数を増やしたり、ガーベッジ・コレクションに費やす時間を短縮したりすることができます。

最初のステップとなるのは、「Top Components (上位コンポーネント)」レポートの作成です。このレポートはシステムを構成するコンポーネント別にメモリー使用量を分割し、各コンポーネントでの使用量を分析して、メモリーを無駄に使用している処理を探します。別のオブジェクトによって支配 (保持) されるオブジェクトは、支配側のオブジェクト (ドミネーター) に所有されていると言えます。「Top Components (上位コンポーネント)」レポートには、別のオブジェクトには所有されていないオブジェクトがすべてリストアップされます。これらのオブジェクトが、ヒープを最も支配している上位ドミネーターです。上位ドミネーターは、オブジェクトのクラスを使用するクラスローダーによってさらに分割され、これらの上位ドミネーターとそれぞれが所有するオブジェクトのすべてが、対応するクラスローダーに割り当てられます。さらに詳しく分析するには、レポート内の該当するクラスローダーを選択して、そのクラスローダー固有の新規コンポーネント・レポートを開きます。

各コンポーネントに対しては、コレクション・オブジェクトが分析されます。`java.util.*` で示されるコレクション・クラスは、十分にテストされたリスト、セット、およびマップの実装を提供することから、プログラマーにとって大幅な時間の節約となります。平均的なアプリケーションには、数百万のコレクションがあることもあります。したがって、コレクション内に無駄なスペースがあると、かなりのスペースが浪費されることになります。

空のコレクションは、メモリー浪費の原因としてよくありがちな例の1つです。`ArrayList`、`Vector`、`HashMap`、および `HashSet` はデフォルトで、すぐにエントリーを格納できるように、おそらくエントリー数 10 のデフォルト・サイズを持つバッキング配列として作成されます。アプリケーションで驚くほどよく見かけるのは、コレクションが作成されるにも関わらず、そこにはオブジェクトが格納されないことです。これでは瞬く間に大量のメモリーが使われてしまいます。例えば、空のコレクション数が 100,000 あるとすると、バッキング配列だけでも  $100,000 * (24 + 10 * 4)$  バイト、つまり 6MB を使用することになります。

「Empty Collection (空のコレクション)」レポートは、標準コレクション・クラスとその継承クラスを調べ、コレクションのサイズを基準にコレクションを分析します。そして、サイズによってソートしたコレクションごとに、最も多いサイズを先頭に配置したテーブルを生成します。ある



コレクション型のインスタンスの大半が空の場合、レポートはそのコレクションに、メモリー浪費の可能性があるとフラグを立てます。

1つの解決方法は、コレクションにエントリーを格納する必要が出てくるまで、コレクションの割り当てを遅らせることです。あるいは、ランタイムをある程度犠牲にして、デフォルト・サイズが0または1のコレクションを割り当て、必要に応じてサイズを拡大させるという方法もあります。さらに3つ目の方法として、初期化フェーズが完了した後にコレクションのサイズを調整することも可能です。

これに関連するのは、エントリーが少数しかなく、多くのスペースが無駄になっているコレクションです。「Collection Fill Ratio (コレクション充填率)」セクションは、コレクション型ごとに、ある充填率の基準に当てはまるインスタンスの数を示します。この情報から、大部分が空のスペースとなっているコレクションが明らかになります。

## 重複ストリング

典型的なビジネス・アプリケーションでは、ストリングと文字配列がかなりのスペースを占めます。そのため、ストリングと文字配列にしても、分析する価値のある領域です。これに該当するコンポーネント・レポートのセクションは、共通のコンテンツについてストリングを分析します。ストリングは不変です。同じインスタンスを使用するVMを指定すれば、ストリング定数が同じ値を持つことは保証されますが、動的に作成されるストリングにはそのような保証はありません。例えば、データベースやディスクから同じ値を持つデータを読み取って作成された2つのStringは、別々のインスタンスと別々のバッキング文字配列を持つことになります。これらのストリングが保持されるとしたら、相当な占有スペースになりかねません。

この問題は、String.intern()を使用するか、ユーザー・ハッシュ・セットまたはハッシュ・マップを保持することで解決することができます。

## 無駄な char 配列

String.substring()は、元の文字配列を共有する新規Stringを作成することによって、Java言語に実装されるため、元のストリングがまだ必要な場合には効率的ですが、小さなサブストリングだけが必要な場合には、文字配列全体が保持されるため、ある程度のスペースが無駄に使用されることになります。「Waste In Char Arrays (文字配列内の浪費)」クエリーは、文字配列内でストリングによって参照されるだけの無駄なスペースの量を明らかにします。

## Eclipse バンドルとクラスローダー階層

最近のアプリケーションは、アプリケーションの構成部分のある程度分離するために、クラスローダーを基準に複数のコンポーネントに分割することが通常となっています。コンポーネントを更新するには、コンポーネントのクラスローダーの使用を停止し、新しいクラスローダーを使用して新規バージョンのコンポーネントをロードします。やがて、アプリケーションにクラスやオブジェクトへの外部参照またはクラスローダーが含まれていないという前提で、古いバージョンはガーベッジ・コレクションによって解放されます。

「Class Loader Explorer (クラスローダー・エクスプローラー)」クエリーは、システム内のすべてのクラスローダーを表示することから、すべてのアプリケーションに役立ちます。このクエリーはクラス・ロードの問題を理解できるように、クラスローダーがロードしたクラスだけでなく、

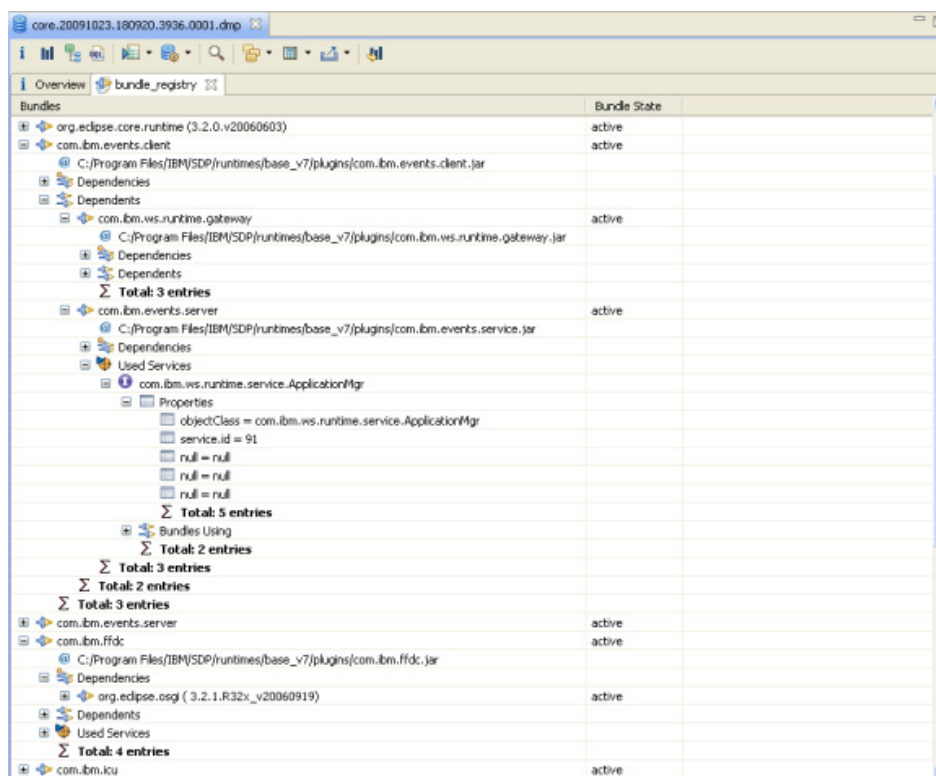


そのクラスローダーの親クラスローダー・チェーンも表示します。表示された情報を調べることで、クラスローダーのコピーが複数あるかどうかを確認することができます。クラスローダーによってクラスのインスタンスがほとんど定義されていない場合、そのクラスローダーはアイドル状態である可能性があります。

重複クラスを検出するためのクエリーによって表示されるのは、複数のクラスローダーによってロードされたクラス名です。これは、クラスローダーのメモリー・リークを示している可能性があります。クラスローダーのリークは、例えばレジストリー内など、システムのどこかで保持されているオブジェクトへの参照が1つでもあれば発生します。オブジェクトはそのオブジェクトのクラスへの参照を保持し、クラスはクラスローダーへの参照を保持し、クラスローダーは定義されたすべてのクラスへの参照を保持するからです。

一般的なクラス・ロードのフレームワークは OSGi フレームワークです。OSGi フレームワークの実装の1つである Eclipse Equinox は、Eclipseベースのアプリケーションでプラグインを分離するために使用されているとともに、WebSphere® Application Server 6.1 以降のバージョンでも使用されています。アプリケーションの状態を理解しようとするときには、すべてのバンドルの状態が把握できると役立ちます。それがまさに、「Eclipse Equinox Bundle Explorer (Eclipse Equinox バンドル・エクスプローラー)」クエリー (図 5 を参照) が表示する内容です。

図 5. Eclipse バンドル・エクスプローラー



システム・ダンプまたは HPROF ダンプには、すべてのオブジェクトとフィールドが含まれます。バンドル・エクスプローラーは、システム内のすべてのバンドルを、それぞれの状態、依存関係、従属オブジェクト、およびサービスと併せて表示します。これにより、予想外にアクティブ状態になっているためにリソースを追加で使用しているバンドルが明らかになります。

## スレッド・データの使用

表 1 に記載されているように、ダンプには、ダンプの取得時に何が行われていたかについて詳細な洞察を得るために、スレッドの詳細情報を含めることができます。詳細情報には、すべてのアクティブなスレッド・スタック、各スレッドのすべてのフレーム、そして最も重要な点として、これらのフレーム上にあるアクティブな Java ローカル変数が含まれます。

### 「スレッドの概要」ビュー

「Thread Overview (スレッドの概要)」ビュー (図 6 を参照) には、JVM 内のすべてのスレッドとそれぞれのスレッドの各種属性 (スレッドが保有するヒープ・サイズ、コンテキスト・クラスローダー、優先度、状態、ネイティブ ID など) が表示されます。

図 6. 「スレッドの概要」ビュー

thread_overview ⓘ			
Name	State	Retained Heap	Context Class Loader
<Regex>	<Regex>	<Numeric>	<Regex>
Attachment 63962	[alive, runnable]	4,728	sun.misc.Launcher\$A
file lock watchdog	[alive, in object wait, waiting,...]	2,760	sun.misc.Launcher\$A
main	[alive, in object wait, waiting,...]	1,832	sun.misc.Launcher\$A
Attach handler	[alive, runnable]	1,752	sun.misc.Launcher\$A
Signal Dispatcher	[alive, runnable]	576	sun.misc.Launcher\$A
JIT Compilation Thread	[alive, in object wait, waiting,...]	344	sun.misc.Launcher\$A
Thread-3		328	sun.misc.Launcher\$A
Gc Slave Thread	[alive, in object wait, waiting,...]	328	sun.misc.Launcher\$A
Gc Slave Thread	[alive, in object wait, waiting,...]	328	sun.misc.Launcher\$A
Gc Slave Thread	[alive, in object wait, waiting,...]	328	sun.misc.Launcher\$A
Gc Slave Thread	[alive, in object wait, waiting,...]	328	sun.misc.Launcher\$A
Gc Slave Thread	[alive, in object wait, waiting,...]	328	sun.misc.Launcher\$A
Gc Slave Thread	[alive, in object wait, waiting,...]	328	sun.misc.Launcher\$A
Gc Slave Thread	[alive, in object wait, waiting,...]	328	sun.misc.Launcher\$A
Thread-1		328	sun.misc.Launcher\$A
Σ Total: 15 entries		14,944	

保有ヒープ・サイズが特に重要な情報となるのは、`OutOfMemoryError` の発生時に Java ヒープの問題自体が発生しているのではなく、複数のスレッドが保有するヒープの合計が「多すぎる」場合です。この場合、原因としては JVM のサイズが小さすぎることに、スレッド・プールのサイズが大きすぎることに、あるいはスレッドの Java ヒープの平均負荷または最大負荷が高すぎることを考えられます。

### 「スレッド・スタック」ビュー

「Thread Stacks (スレッド・スタック)」ビュー (図 7 を参照) には、すべてのスレッドと、スレッドのスタック、スタック・フレーム、およびスタック・フレーム上の Java ローカル変数が表示されます。

## 図 7. 「スレッド・スタック」ビュー

thread_stacks ⓘ		
Object / Stack Frame	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
com.ibm.tools.attach.javaSE.Attachment @ 0x7ff80baf0l	216	4,728
java.util.Timer\$TimerImpl @ 0x7ff80bafb80 file lock wat	176	2,760
java.lang.Thread @ 0x7ff80a52330 main	160	1,832
at java.lang.Object.wait(J)V (Native Method)		
at java.lang.Object.wait()V (Object.java:167)		
at Play.method2(Ljava/lang/String;Ljava/lang/String;)V (		
▶ <local> java.lang.String @ 0x7ff80baa04 user1	48	88
at Play.method1(Ljava/lang/String;Ljava/lang/String;)V (		
at Play.main([Ljava/lang/String;)V (Play.java:6)		
Σ Total: 5 entries		
com.ibm.tools.attach.javaSE.AttachHandler @ 0x7ff80afi	192	1,752
com.ibm.misc.SignalDispatcher @ 0x7ff80ae1db8 Signa	160	576

### スレッドの詳細ビュー

「Thread Overview (スレッドの概要)」ビューと「Thread Stacks (スレッド・スタック)」ビューでは、スレッドを右クリックして、メニューの一番上にある「Thread Details (スレッドの詳細)」を選択するか、「Java Basics (Java 基本情報)」>「Thread Details (スレッドの詳細)」の順に選択すると、ネイティブ・スタックなど(使用可能な場合)、スレッドに関するさらに詳細な情報を表示することができます。

図 7 の例では、`java.lang.Thread` 型の `main` というスレッド(単純なコマンドライン・プログラムでのメイン・スレッド)が展開されていて、このスレッドの各スタック・フレームが表示されています。Java ローカル変数を表示できるスタック・フレームは展開できるようになっています。上記の例では、`String` が `Play.method1` から引数として `Play.method2` に渡されたことがわかります。この文字列の内容は、赤い丸で囲まれている `user1` です。このように、各スレッドのスタック・フレームで実行されていた内容と、実行されていたオブジェクトに基づいて、ダンプの取得時に実行されていた内容を再現またはリバース・エンジニアリングできるとなれば、それがいかに強力な機能になるかは想像できるはずです。

ランタイム最適化のため、メソッド・パラメーターやオブジェクト・インスタンスなどの関連オブジェクトは、(ダンプ内にはあるものの)そのすべてが表示されるわけではありませんが、頻繁に作業対象にされるオブジェクトについては、通常は表示されることに注意してください。

## 例外分析

アプリケーションで例外が発生すると、さらに複雑さが増すことで、例外の原因を分析するのが一層困難になる可能性があります。以下は、この難しさを説明する 2 つの例です。

- ロギング・メカニズムを使用するということは、例外が失われるか、例外メッセージが削除されることを意味します。
- 例外が生成するメッセージには、十分な情報が含まれていません。

最初の例では、例外メッセージまたは例外そのものが完全に失われるため、問題の存在を知ること、問題に関する基本的な情報を入手することも困難です。2 番目の例の場合には、例外はログに記録され、例外メッセージとスタック・トレースも使用できますが、そこには例外の原因を解決できるだけの情報がありません。

一方、Memory Analyzer はオブジェクト内のフィールドにアクセスできるため、例外オブジェクトから例外メッセージを見つけることができます。場合によっては、元の例外にはない追加データを抽出することも可能です。

## スナップショット・ダンプで例外を見つける

スナップショット・ダンプに存在する例外を見つける 1 つの方法は、Memory Analyzer の OQL 機能を利用して、ダンプで対象のオブジェクトを検索することです。例えば、以下のクエリはすべての例外オブジェクトを検出します。

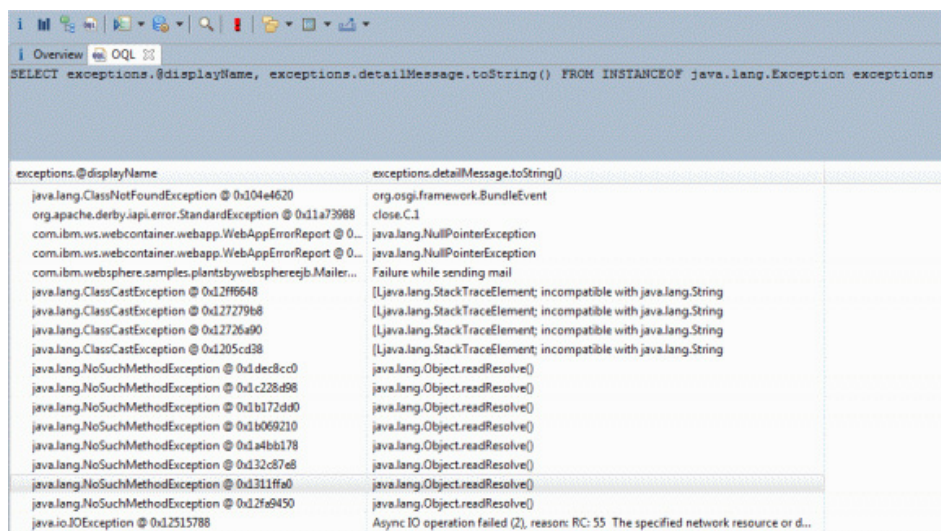
```
SELECT *
FROM INSTANCEOF java.lang.Exception exceptions
```

以下のクエリは、例外をすべて含んだリストを生成します。このリストの各例外に含まれるフィールドは、「Inspector (インスペクター)」ビューを使って調べることができます。例外メッセージが含まれるフィールドは detailMessage フィールドであることはわかっているため、クエリを変更して例外メッセージを直接抽出し、結果テーブルの一部として直に表示することもできます。

```
SELECT exceptions.@displayName, exceptions.detailMessage.toString()
FROM INSTANCEOF java.lang.Exception exceptions
```

図 8 に、上記のクエリによって生成された出力を示します。

## 図 8. 例外を対象とした OQL クエリによる、例外メッセージを含む出力



exceptions.@displayName	exceptions.detailMessage.toString()
java.lang.ClassNotFoundException @ 0x104e4620	org.osgi.framework.BundleEvent
org.apache.derby.jdbc.StandardException @ 0xd1a73988	close.C.1
com.ibm.ws.webcontainer.webapp.WebAppErrorReport @ 0...	java.lang.NullPointerException
com.ibm.ws.webcontainer.webapp.WebAppErrorReport @ 0...	java.lang.NullPointerException
com.ibm.wsphere.samples.plantsbywebsphereejb.Mailer...	Failure while sending mail
java.lang.ClassCastException @ 0xd2ff6648	[Ljava.lang.StackTraceElement; incompatible with java.lang.String
java.lang.ClassCastException @ 0xd27279b8	[Ljava.lang.StackTraceElement; incompatible with java.lang.String
java.lang.ClassCastException @ 0xd2726a90	[Ljava.lang.StackTraceElement; incompatible with java.lang.String
java.lang.ClassCastException @ 0xd205cd38	[Ljava.lang.StackTraceElement; incompatible with java.lang.String
java.lang.NoSuchMethodException @ 0xd1dec8c0	java.lang.Object.readResolve()
java.lang.NoSuchMethodException @ 0xd1c228d98	java.lang.Object.readResolve()
java.lang.NoSuchMethodException @ 0xd1b172dd0	java.lang.Object.readResolve()
java.lang.NoSuchMethodException @ 0xd1b069210	java.lang.Object.readResolve()
java.lang.NoSuchMethodException @ 0xd1a4bb178	java.lang.Object.readResolve()
java.lang.NoSuchMethodException @ 0xd132c87e8	java.lang.Object.readResolve()
java.lang.NoSuchMethodException @ 0xd1311ffa0	java.lang.Object.readResolve()
java.lang.NoSuchMethodException @ 0xd12fa9450	java.lang.Object.readResolve()
java.io.IOException @ 0xd12515788	Async IO operation failed (2), reason: RC: 55 The specified network resource or d...

図 8 には、アプリケーションに存在しているすべての例外と、それぞれの例外がスローされたときに表示されるメッセージが示されています。

## 例外に関する追加情報を抽出する

ダンプから例外オブジェクトを見つけることによって例外メッセージを再現することはできますが、例外メッセージがあまりにも一般的または曖昧すぎて、問題の原因を理解するのが困難な場合もあります。その好例は、java.net.ConnectException です。接続を受け付けられないホストに対してソケット接続を試行した場合、以下のメッセージを受け取ります。



```
java.net.ConnectException: Connection refused: connect
  at java.net.PlainSocketImpl.socketConnect(Native Method)
  at java.net.PlainSocketImpl.doConnect(PlainSocketImpl.java:352)
  at java.net.PlainSocketImpl.connectToAddress(PlainSocketImpl.java:214)
  at java.net.PlainSocketImpl.connect(PlainSocketImpl.java:201)
  at java.net.SocksSocketImpl.connect(SocksSocketImpl.java:377)

  at java.net.Socket.connect(Socket.java:530)
  at java.net.Socket.connect(Socket.java:480)
  at java.net.Socket.(Socket.java:377)
  at java.net.Socket.(Socket.java:220)
```

ソケットを作成しているコードにアクセスできる場合には、そのコードを見れば使用されているホスト名とポート番号がわかるので、上記のメッセージで十分事足ります。しかし、ホスト名とポート番号を外部ソース（ユーザーが入力した値や、データベースなど）から取得する複雑なコードの場合には、ホスト名とポート番号は固定されているわけではないため、このメッセージでは接続が拒否されている理由を知ることができません。

スタック・トレースには、有用なデータが含まれるソケット・オブジェクトが含まれているはずです。そのソケット・オブジェクトを、Memory Analyzer を使ってスナップショット・ダンプで見つけることができれば、接続を拒否されたホスト名とポート番号を特定することができます。

そのためには、例外がスローされたときにダンプを生成するのが最も簡単な方法となります。IBM ランタイムで以下の -Xdump オプションを使用すれば、例外のスローに応じてダンプを生成することができます。

```
-Xdump:system:events=throw,range=1..1,
  filter=java/net/ConnectException#java/net/PlainSocketImpl.socketConnect
```

このオプションは、PlainSocketImpl.socketConnect() メソッドによって ConnectException が初めて発生した時点で IBM システム・ダンプを生成します。

生成されたスナップショット・ダンプを Memory Analyzer にロードした後は、「Open Query Browser (クエリー・ブラウザーを開く)」 > 「Java Basics (Java 基本情報)」 > 「Thread Stacks (スレッド・スタック)」の順に選択することで、スレッドのスタック・トレースに含まれるスレッドと各メソッドに関連付けられたオブジェクトを表示することができます。

カレント・スレッド、そしてスレッドのメソッド・フレームを展開すれば、メソッドに関連付けられたオブジェクトを調べることができます。java.net.ConnectException の場合、最も興味深いメソッドは java.net.Socket.connect() です。このメソッド・フレームを展開すると、メモリー内の java.net.Socket オブジェクトへの参照が示されます。これが、確立しようとしていたソケット接続です。

Socket オブジェクトを選択すると、「Inspector (インスペクター)」ビューにこのオブジェクトのフィールドが表示されます (図 9 を参照)。



## 図 9. Socket オブジェクトの「インスペクター」ビュー

Statics Attributes Class Hierarchy		
Type	Name	Value
boolean	created	true
boolean	bound	false
boolean	connected	false
boolean	closed	false
ref	closeLock	java.lang.Object @ 0xd01148
boolean	shutIn	false
boolean	shutOut	false
ref	impl	java.net.SocksSocketImpl @ 0xd01148
boolean	oldImpl	false

図 9 に示されている情報は、それほど役に立ちません。その理由は、Socket の実際の実装は、impl フィールドにあるためです。impl オブジェクトの内容を調べるには、Socket オブジェクトを展開してメイン・パネルで「impl java.net.SocksSocketImpl」の行を選択するか、「Inspector (インスペクター)」ビューの「impl」フィールドを右クリックして「Go Into (移動)」を選択してください。すると、SocksSocketImpl のフィールドが「Inspector (インスペクター)」ビューに表示されます (図 10 を参照)。

## 図 10. SocksSocketImpl オブジェクトの「インスペクター」ビュー

Statics Attributes Class Hierarchy		
Type	Name	Value
ref	server	null
int	port	1080
ref	external_a...	null
boolean	useV4	false
ref	cmdsock	null
ref	cmdIn	null
ref	cmdOut	null
int	timeout	0
int	trafficClass	0
boolean	shut_rd	false
boolean	shut_wr	false
ref	socketInput...	null
int	fdUseCount	1
ref	fdLock	java.lang.Object @ 0xd03748
boolean	closePending	false
int	CONNECTI...	0
int	CONNECTI...	1
int	CONNECTI...	2
int	resetState	0
ref	resetLock	java.lang.Object @ 0xd03758
ref	fd1	null
ref	anyLocalBo...	null
int	lastfd	-1
int	backlogQ	50
int	sport	0
ref	socket	java.net.Socket @ 0xd00fd0
ref	serverSocket	null
ref	fd	java.io.FileDescriptor @ 0xd00fd0
ref	address	java.net.Inet4Address @ 0xc0000000
int	port	100
int	localport	0

図 10 に示したビューから、address フィールドと port フィールドにアクセスすることができます。この例では、ポート番号は 100 であることがわかりますが、address フィールドは java.net.Inet4Address オブジェクトを指しています。同じプロセスを辿って Inet4Address オブジェクトのフィールドを調べると、図 11 の結果が表示されます。

## 図 11. Inet4Address オブジェクトの「インスペクター」ビュー

Statics		
Attributes		
Class Hierarchy		
Type	Name	Value
ref	hostName	baileyt60p
int	address	162002664
int	family	1
ref	canonicalH...	null

これで、hostName が baileyt60p に設定されていることがわかります。

## ヒントとコツ

以下に、参考にできるヒントとコツをいくつか紹介します。

- Memory Analyzer 自体もメモリー不足になる可能性があることを忘れないでください。メモリー不足が発生する場合、Eclipse MAT では MemoryAnalyzer.ini ファイルの `-Xmx` を編集します。ISA バージョンの場合は、ISA Install/rcp/eclipse/plugins/com.ibm.rcp.j2se.../jvm.properties ファイルを編集します。
- 32 ビット版の Memory Analyzer で、上記の編集を行ってもメモリー不足になる場合には、64 ビット版の Eclipse MAT を使用するか、ヘッドレス・モード（[参考文献](#)）を参照）を試してみてください（現在、ISA ツールでは 64 ビットをサポートしていません）。
- Memory Analyzer は、スワップ・ファイルをダンプのディレクトリーに書き込むことによって、ダンプのリロード時間を短縮します。これらのファイルを圧縮して別のマシンに送信し、ダンプと同じディレクトリーに配置すれば、ダンプをまるごとリロードする必要がなくなります。
- ダンプのサイズがダンプ生成時のガーベッジ・コレクターと関連していない場合は、「Overview (概要)」タブの「Unreachable Objects Histogram (到達不能オブジェクトのヒストグラム)」リンクを参照してください。Java ヒープに、Memory Analyzer によって削除された大量のガーベッジが含まれていた可能性があります（例えば、終身世代コレクションがしばらく実行されていなかった場合など）。
- オブジェクト A とオブジェクト B は、互いを直接参照していない一方で、この 2 つのオブジェクトの両方にオブジェクト・セット C への外部参照がある場合、オブジェクト・セット C が保有するヒープは、A または B が保有するヒープのいずれにも含まれませんが、A と B 両方のドミネーターが保有するセットに含まれることになります。場合によっては B が一時的に、実際には A の子孫であるオブジェクト・セット C を参照していることもあります。この場合、A を右クリックして「Java Basics (Java 基本情報)」 > 「Customized Retained Set (カスタマイズされた保有セット)」の順に選択し、B のアドレスを除外 (-x) パラメーターに指定することができます。
- 一度に複数のダンプをロードして比較することができます。それには、新しいほうのダンプのヒストグラムを開き、上部にある「Compare (比較)」ボタンをクリックして、基準とするダンプを選択します。
- 参照ツリーを探索する際には、参照は直接または間接的に「親」リファレンスを逆参照できるため、探索ループまたは探索サイクルに入ってしまう可能性があることに注意してください（リンク・リストなどにおける場合）。オブジェクト・アドレスにも注意してください。さらに注意する点として、オブジェクトのクラス名が `class` で始まる場合は、そのクラスの静的インスタンスを探索していることになります。

- 大部分のビューでは、表示される String 値は 1,024 文字に制限されます。完全な String が必要な場合には、オブジェクトを右クリックして「Copy (コピー)」 > 「Save value to file (値をファイルに保存)」の順に選択してください。
- ほとんどのビューにはエクスポート・オプションがあり、HTML による結果のほとんどはファイルシステムに作成されます。そのため、データをエクスポートして共有したり、さらに変換したりすることができます。これと関連して、グリッド内の行を選択して Ctrl+C を押すことで、選択した行のテキストをクリップボードにコピーすることができます。

## まとめ

Eclipse.org での説明によると、Memory Analyzer は当初、「メモリー・リークを検出し、メモリー使用量を低減するために利用できる、高速で充実した機能を備えた Java ヒープ・アナライザー」として開発されました。しかし、Memory Analyzer にはその説明を遥かに超えた能力があることは明らかです。通常のメモリー問題を診断する上での役割に加え、スナップショット・ダンプはトレースやパッチなどの他のタイプの問題判別手法の代わりとして、あるいはその補足として使用することができます。特に HPROF ダンプと IBM システム・ダンプでは、Memory Analyzer がメモリーの内容 (元のソース・コードで使われているプリミティブ型やフィールド名など) を明らかにします。この記事で取り上げたさまざまなビューを使用すれば、全体的なメモリー・フットプリントとメモリーの効率性、Eclipse バンドルとクラスローダーの関係、スレッド・データ使用量とスタック・フレーム上にあるローカル変数、例外などをはじめ、目の前の問題を詳しく調べたり、リバース・エンジニアリングを行ったりすることができます。さらに OQL と Memory Analyzer プラグイン・モデルも、一般的な分析の自動化に役立つ問い合わせ言語とプログラムによるメソッドを使用することで、ダンプの調査を容易にします。

---

## 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。

---

Andrew Johnson



Andrew Johnson は、英国 Hursley の IBM Java Technology Center に勤務する公認技術者兼顧問ソフトウェア・エンジニアです。ケンブリッジ大学で電子工学の学士号を取得した後、1988年に IBM に入社しました。1996年からは、Java 課仮想マシン、JIT コンパイラー、そして Java の問題診断用ツールに取り組んでいます。Eclipse Memory Analyzer が IBM VM からダンプを読み込むためのアダプターを作成した彼は、現在、このプロジェクトのコミッターとなっています。

---

Kevin Grigorenko



Kevin Grigorenko は、WebSphere Application Server SWAT チームのソフトウェア・エンジニアです。このチームでは、特に重要なカスタマー・サポートのケースで、世界的規模でオンサイトおよびリモートでの製品欠陥に対する追加サポートを行っています。彼は現在、WebSphere Application Server ならびに JVM や各種オペレーティング・システムなどの関連スタック製品での問題判別に取り組んでいます。Java Enterprise Edition、C、C++、Perl、PHP、Python、Ruby、および .NET を含め、開発においても豊富な経験を積んでいます。

© Copyright IBM Corporation 2011

([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/))