POONAM BAJAJ

HotSpotの秘宝

HotSpot Serviceability Agentは、実行中のJavaプロセスとコア・ファイルのデバッグを行える強力なツールです。

JDKに含まれるHotSpot Serviceability Agentは、ほとんどの人に知られていない、いわば秘宝です。Serviceability Agent（SA）は、実行中のJavaプロセスやコア・ファイル（Microsoft Windowsではクラッシュ・ダンプとも呼ばれます）をデバッグできる、Java APIおよびツールのセットです。

SAは、Javaプロセスやコア・ファイルを調べることができるため、Java HotSpot VMだけでなくJavaプログラムのデバッグにも適しています。SAはスナップショット・デバッガであり、休止したJavaプロセスの状態やコア・ファイルの状態を確認できます。SAをJavaプロセスにアタッチした場合、Javaプロセスはその時点で休止します。これにより、Javaのヒープの調査、当該プロセスにおいて休止時点で実行中だったスレッドの参照、Java HotSpot VMの内部データ構造の検証、ロード済みクラスやメソッドのコンパイル済みコードなどの確認といった作業が可能になります。プロセスは、SAをデタッチした後に再開されます。

SAのバイナリ

SAの機能およびユーティリティの詳細について解説する前に、JDKに含まれるSAのバイナリについて説明します。JDKには、SAのバイナリが2つ付属しています。

- Microsoft Windows版: sa-jdi.jarとjvm.dll
- Oracle Solaris版およびLinux版: sa-jdi.jarとlibsaproc.so

これらのバイナリには、SAのJava APIだけでなく、そのAPIを使用して実装された便利なデバッグ・ツールも含まれています。

SAのバイナリをすべて含むJDKのバージョン

以下のバージョンのJDKには、SAのバイナリ一式がすべて含まれます。

- JDK 7（すべてのプラットフォーム）
- 6u17以降（Oracle SolarisおよびLinux）
- 6u31以降（Microsoft Windows）

上記より前のバージョンの場合、Microsoft Windows版のJDKにはSAが付属しておらず、Oracle Solaris版とLinux版のJDKにはSAの一部のクラスのみが付属しています。上記のJDKの各バージョンでは、記載のプラットフォームでSAのすべてのクラスを利用できます。

SAを利用する理由

dbx、GDB、WinDbgといったネイティブのデバッグ・ツールが数多く存在する中で、SAを利用する理由は何でしょうか。

第1の理由は、SAがプラットフォームに依存しないJavaベースのツールであることです。そのため、SAを利用することで、Javaがサポートされるあらゆるプラットフォームにおいて、Javaのプロセスおよびコア・ファイルをデバッグできます。さらに、ネイティブのデバッガを使用してJavaプロセスやJava HotSpot VMをデバッグする場合は、OSネイティブなプロセスの状態は調査できるものの、そのプロセスにおけるJavaやJava仮想マシン（JVM）の状態の調査はできないという制約があります。

たとえば、Javaのヒープ内にあるオブジェクトを確認する必要がある場合、ネイティブのデバッガでは生の16進数が表示されますが、SAには、その16進数を解釈し、オブジェクトとして表示する機能があります。また、SAはJavaのヒープを理解でき、Javaヒープの境界、Javaヒープ内のオブジェクト、ロード済みのクラス、スレッド・オブジェクト、Java HotSpot VMの内部表現などを認識できます。SAを利用することで、Javaプロセスやコア・ファイルに関するJavaレベルの詳細情報やJVMレベルの詳細情報を簡単に調べることができます。

> **より詳細に**
> SAを利用することで、Javaプロセスやコア・ファイルに関する**Javaレベルの詳細情報やJVMレベルの詳細情報**を簡単に調べることができます。

SAのデバッグ・ツール

SAのデバッグ・ツールには、おもに次の2つがあります。いずれもSAのAPIを使用して実装されます。

- HSDB: GUIツール形式のメイン・デバッガ
- CLHSDB: HSDBのコマンドライン版

HSDB: GUIデバッガ。HSDBを使用して、Javaプロセス、コア・ファイル、さらにはリモートのJavaプロセスを容易に調べることができます。まず、Microsoft WindowsマシンでのHSDBの起動方法と利用方法を説明します。

はじめに、いくつかの環境変数を設定する必要があります。次のように、SA_JAVAに、JDK/binフォルダ内のJava実行ファイルの場所を設定します。

```
set SA_JAVA=
d:\java\jdk1.7.0_03\bin\java
```

Microsoft Windowsでは、PATH環境変数に、デバッグの対象となるプロセスやコア・ファイルで使用されるJVMバイナリの場所と、Debugging Tools for Windowsがインストールされているフォルダの場所を含める必要があります。たとえば、次のように設定します。

```
set PATH= d:\java\jdk1.7.0_03\bin\
server;d:\windbg;%PATH%
```

PATH環境変数を設定した後、次のコマンドでHSDBを起動します。

```
java -Dsun.jvm.hotspot.debugger.
useWindbgDebugger=true -classpath
d:\java\jdk1.7.0_03\lib\sa-jdi.jar
sun.jvm.hotspot.HSDB
```

Oracle SolarisマシンまたはLinuxマシンの場合は、SA_JAVAにJava実行ファイルの場所を設定することのみ必要です。この設定を行った後に、次のコマンドでHSDBを起動します。

```
java -Dsun.jvm.hotspot.debugger.
useProcDebugger=true -classpath
/java/jdk1.7.0/lib/sa-jdi.jar
```

上記の起動コマンドによって、**図1**に示すようにHSDBのGUIツールが起動します。

図2で、このGUIツールで利用できる便利なユーティリティを確認できます。このうちのいくつかを説明します。

図3に示されているオブジェクト･インスペクタ（Inspector）は、Javaオブジェクトを調べるために使用できます。

図4は、Javaプロセス内の特定のアドレスを検索する方法を示しています。

図5は、Object Histogramです。

図6に示すように、ヒープの境界を検出することもできます。

CLHSDB: コマンドライン･デバッガ。 CLHSDBは、HSDBのコマンドライン版です。

CLHSDBに対しても、HSDBと同様に環境変数を設定する必要があります。Microsoft Windowsでこのツールを起動するには、次のコマンドを使用します。

```
java -Dsun.jvm.hotspot.debugger.
useWindbgDebugger=true -classpath
d:\java\jdk1.7.0_03\lib\sa-jdi.jar
sun.jvm.hotspot.CLHSDB
```

CLHSDBでは、HSDBで利用できる機能のほぼすべてを利用できます。たとえば、特定のJavaオブジェクトを調べるには、inspectコマンドを使用します（**リスト1**）。

ヒープの境界を確認するには、universeコマンドを使用します（**リスト2**）。

リスト3と**リスト4**に、このツールで利用できるすべてのコマンドを示します。

その他のツール

SAには、上記で紹介したツールのほかにも便

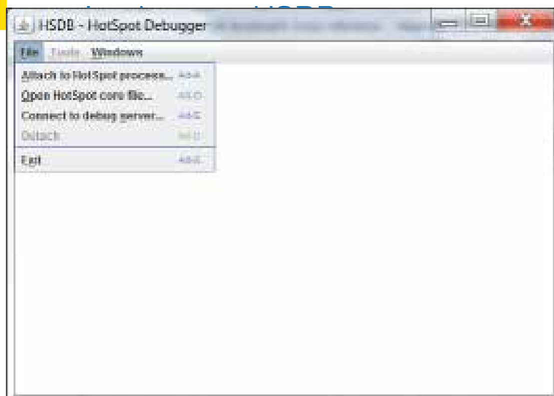


図1

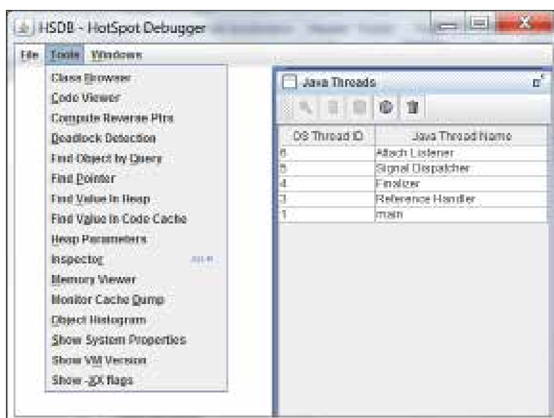


図2

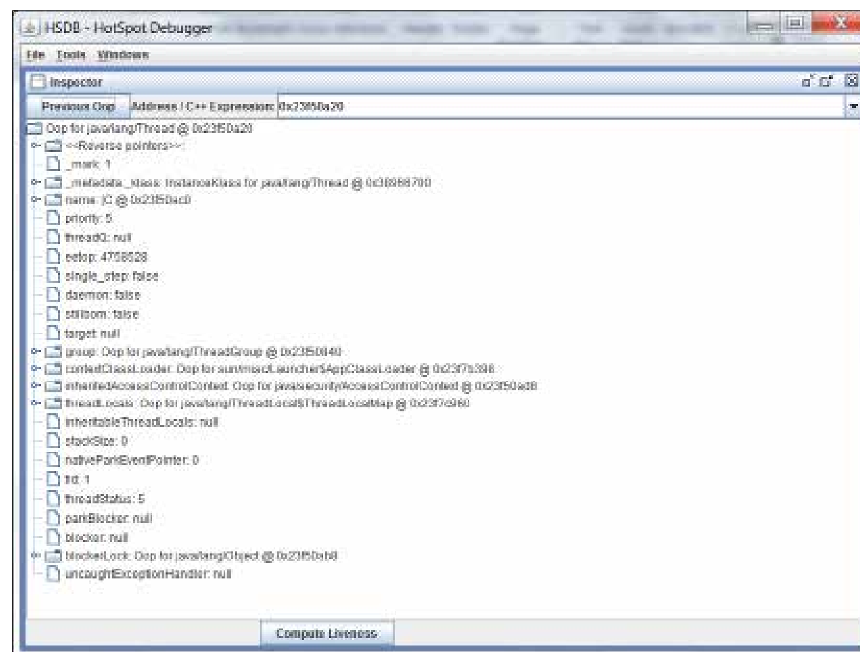


図3

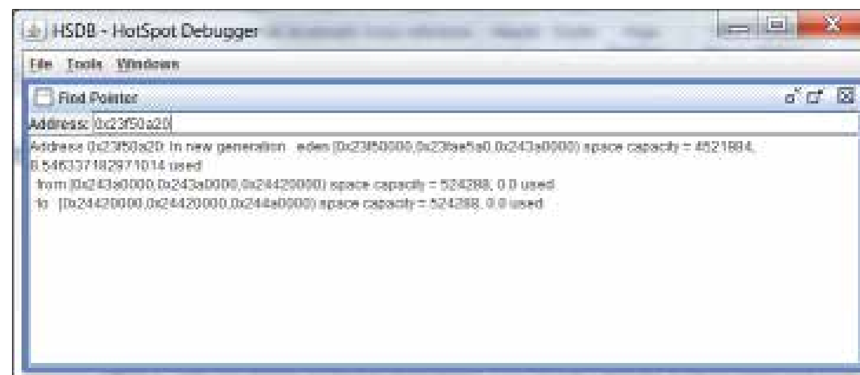


図4

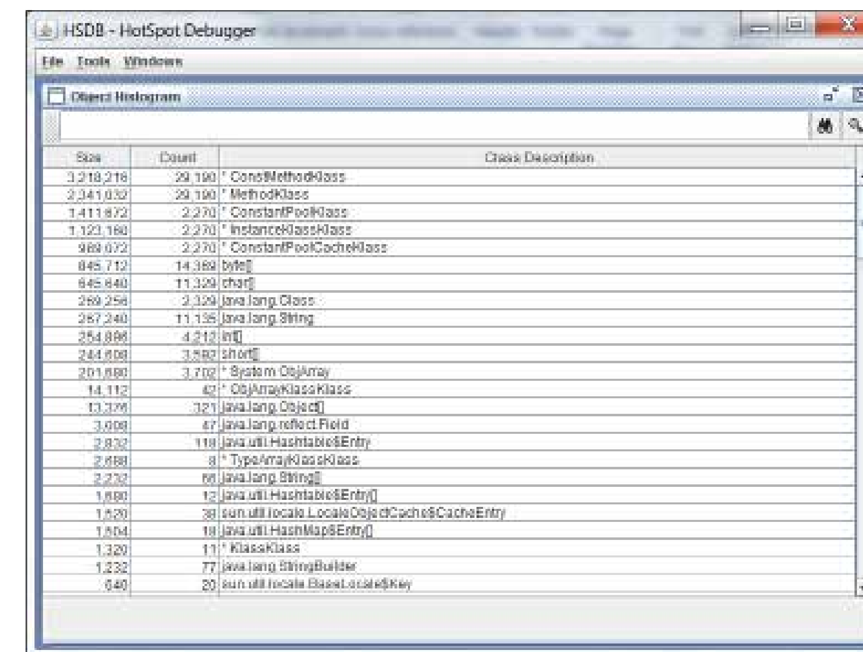


図5

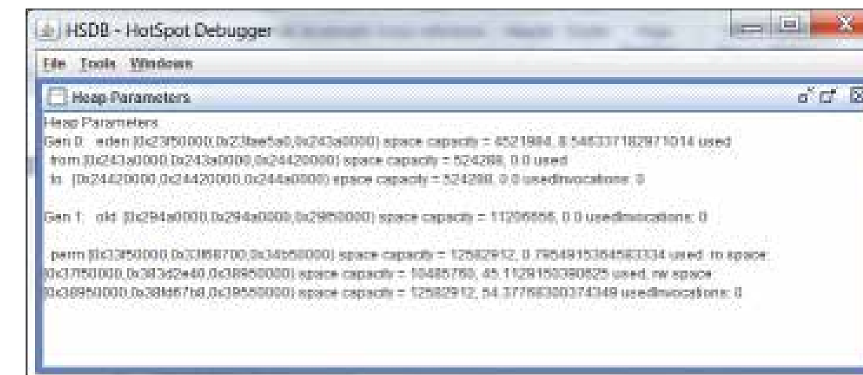
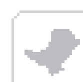


図6

COMMUNITY
JAVA IN ACTION
JAVA TECH
ABOUT US

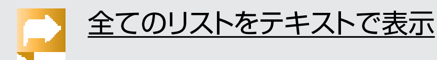


利なユーティリティがバンドルされています。各種ユーティリティの使用方法とそれぞれの出力結果について説明します。

- FinalizerInfoは、ファイナライズ可能なオブジェクトの詳細情報を出力します(**リスト5**参照)。
- HeapDumperは、ヒープをHPROF形式でダンプします(**リスト6**参照)。
- PermStatは、Permanent領域の統計情報を出力します(**リスト7**参照)。
- PMapは、プロセスのプロセス・マップを出力します(**リスト8**参照)。Oracle Solarisのpmapツールに類似しています。
- SQLに類似した言語であるSOQL(Structured Object Query Language)を使用して、Javaのヒープに対する問合せを実行できます(**リスト9**参照)。JHatにもSOQLを使用するためのインタフェースがあります。また、JHatには、SOQLについての詳細なドキュメントもあります。
- JSDB(JavaScript Debugger)は、SAに対するJavaScriptのインタフェースです(**リスト10**参照)。具体的には、MozillaのRhino JavaScriptエンジンをベースとしたコマンドラインのJavaScriptシェルです。JSDBに関する詳細は、Java HotSpot VMのオープンソース・リポジトリにあるhotspot/agent/doc/jsdb.htmlファイルに記載されています。

リスト1 / リスト2 / リスト3 / リスト4 / リスト5 / リスト6

```
hsdb> inspect 0x23f50a20
instance of Oop for java/lang/Thread @ 0x23f50a20 @ 0x23f50a20 (size = 104)
_mark: 1
_metadata._klass: InstanceKlass for java/lang/Thread @ 0x38966700 Oop @
0x38966700
name: [C @ 0x23f50ac0 Oop for [C @ 0x23f50ac0
priority: 5
threadQ: null null
eetop: 4758528
single_step: false
daemon: false
stillborn: false
target: null null
group: Oop for java/lang/ThreadGroup @ 0x23f50840 Oop for java/lang/ThreadGroup
@ 0x23f50840
contextClassLoader: Oop for sun/misc/Launcher$AppClassLoader @ 0x23f7b398 Oop
for sun/misc/Launcher$AppClassLoader @ 0x23f7b398
inheritedAccessControlContext: Oop for java/security/AccessControlContext @
0x23f50ad8 Oop for java/security/AccessControlContext @ 0x23f50ad8
threadLocals: Oop for java/lang/ThreadLocal$ThreadLocalMap @ 0x23f7c960 Oop for
java/lang/ThreadLocal$ThreadLocalMap @ 0x23f7c960
inheritableThreadLocals: null null
stackSize: 0
nativeParkEventPointer: 0
tid: 1
threadStatus: 5
parkBlocker: null null
blocker: null null
blockerLock: Oop for java/lang/Object @ 0x23f50ab8 Oop for java/lang/Object @
0x23f50ab8
uncaughtExceptionHandler: null nullCheck heap boundaries
```


全てのリストをテキストで表示

実際の使用

次に、SAツールを使用して、クラッシュしたJavaプログラムを実際にデバッグする例を示します。そのために、Java Native Interface(JNI)コードを使用した単純なプログラムを用意しました。このプログラムでは、バイト配列に対し、サイズの上限を超えた書き込みを行います。その結果、Javaヒープ内の次に続くオブジェクトが上書きされ破損します。そのため、ガベージ・コレクタがヒープをスキャンしようとした際に、プログラムがクラッシュします。**リスト11**を参照してください。

プログラム・カウンタ(PC)0xfe5d2c17のobjArrayKlass::oop_follow_contents(oopDesc*) でクラッシュしていることがわかります。**リスト12**に、hs_errファイルに出力された、クラッシュ発生時のスタック・トレースを示します。

クラッシュによって、コア・ファイルが生成されました。このコア・ファイルをHSDBで開き(**図7**参照)、詳細な情報を確認してクラッシュの原因を調べます。

図8に、クラッシュの発生時にPC 0xfe5d2c17の周辺で実行されたコードを

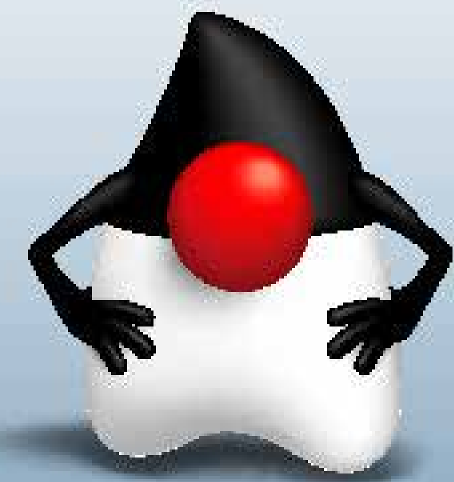
GIVE BACK!
ADOPT A JSR
Find your JSR here


図7

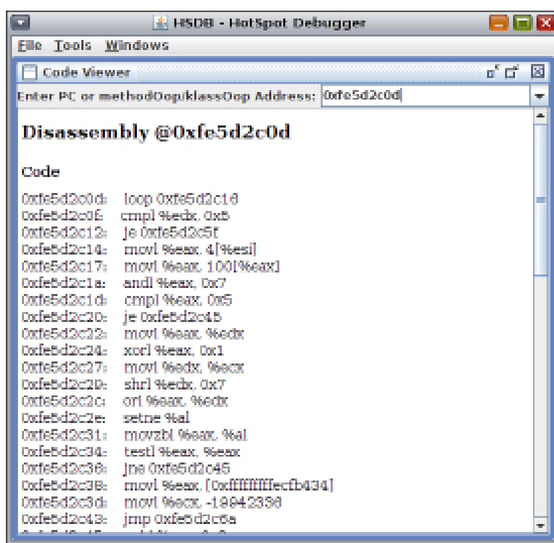


図8

逆アセンブルした結果を示します。

図8に示されている一連の命令は、プロセスがアドレスeax+100にある値にアクセスしようとしたときにクラッシュしたことを示しています。また、hs_errファイルから、次のような各レジスタの内容とEAXレジスタの値を確認できます。

```
EAX=0x6e4f6176, EBX=0xc50a083c,
ECX=0x614a2e2e, EDX=0x00000006
ESP=0xfbc7e360, EBP=0xfbc7e398,
ESI=0xc5036ef0, EDI=0x00000000
EIP=0xfe5d2c17, EFLAGS=0x00010202
```

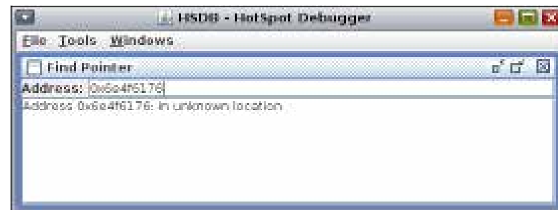


図9

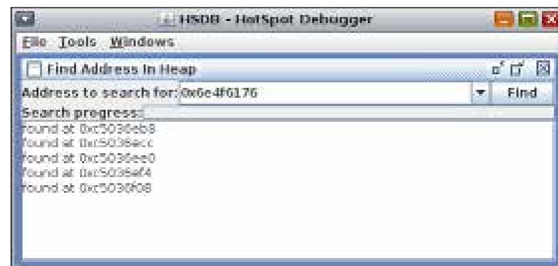


図10

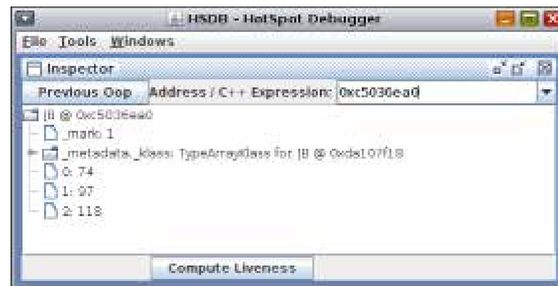


図11

0x6e4f6176には何があったのでしょうか。また、このアドレスにある値を読み取る際にクラッシュが発生したのはなぜでしょうか。**図9**に示すように、これらの情報はHSDBを使用して得られます。

アドレス0x6e4f6176は、Javaヒープ内にはありません。アドレス0x6e4f6176を参照しているJavaヒープ内の場所は、「Find Address in Heap」オプションを使用して検索できます（**図10**参照）。

検出された場所（アドレス）をオブジェクト・インスペクタで調査し、それぞれが何らかのオブジェクトの一部であるかどうかを調べます（**図11**参照）。

オブジェクト・インスペクタでは、検出されたアドレスのすべてについて、0xc5036ea0にあるバイト配列オブジェクトが示されます。

リスト7 / リスト8 / リスト9 / リスト10 / リスト11 / リスト12

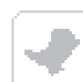
```
java -Dsun.jvm.hotspot.debugger.useWindbgDebugger=true -classpath
d:\java\jdk1.7.0_03\lib\sa-jdi.jar sun.jvm.hotspot.tools.PermStat 5684
Attaching to process ID 5684, please wait...
Debugger attached successfully.
Client compiler detected.
JVM version is 22.1-b02
10713 intern Strings occupying 802608 bytes.
finding class loader instances ..
done.
computing per loader stat ..done.
please wait.. computing liveness.........................................done.
class_loader    classes bytes   parent_loader   alive?  type

<bootstrap>     342     1539808   null          live    <internal>
0x23f7b398      3       28016   0x23f762e0      live    sun/misc/Laun
cher$AppClassLoader@0x38a0e9c0
0x23f762e0      0       0         null          live    sun/misc/Launcher$Ext
ClassLoader@0x389eb420

total = 3       345     1567824     N/A         alive=3, dead=0     N/A
```

全てのリストをテキストで表示

これは、0xc5036ea0にあるオブジェクトが、これらの検出されたアドレスの直前にある、もっとも近接した有効なオブジェクトであることを意味しています。さらに詳しく調べた場合、検出されたアドレスは、バイト配列オブジェクトの境界を過ぎていることがわかります。本来、このバイト配列オブジェクトはアドレス0xc5036eb0で終了すること、さらにア



リスト13 リスト14

```
(dbx) x 0xc5036ea0/100c
0xc5036ea0:   '\001' '\0' '\0' '\0' '\030' '\0177' '\020' '£' '\003' '\0'
'\0' '\0'
'H' 'e' 'l' 'l'
0xc5036eb0:   'o' ' ' 'J' 'a' 'v' 'a' '.' 'H' 'e' 'l' 'l' 'o' ' ' 'J' 'a' 'v'
0xc5036ec0:   'a' '.' 'H' 'e' 'l' 'l' 'o' ' ' 'J' 'a' 'v' 'a' '.' 'H' 'e' 'l'
0xc5036ed0:   '\003' '\0' '\0' '\0' 'a' 'v' 'a' '.' 'H' 'e' 'l' 'l' 'o' ' ' 'J' 'a'
0xc5036ee0:   'v' 'a' '.' 'H' 'e' 'l' 'l' 'o' ' ' 'J' 'a' 'v' 'a' '.' 'H' 'e'
0xc5036ef0:   'l' 'l' 'o' ' ' 'J' 'a' 'v' 'a' '.' 'H' 'e' 'l' 'l' 'o' ' ' 'J'
0xc5036f00:   'a' 'v' 'a' '.'
```

全てのリストをテキストで表示

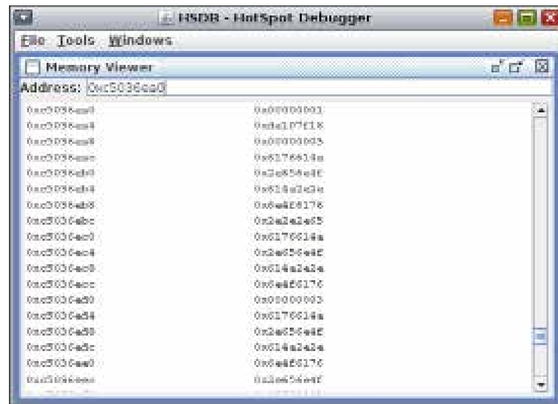


図12

ドレス0xc5036eb0からは次のオブジェクトが開始されていることが想定されていました。メモリのアドレス0xc5036ea0にある生データを**図12**に示します。

この生データは、dbxデバッガを使用して、文字列として表示できます。この文字列を**リスト13**に示します。0xc5036ea0にあるオブジェクトにはバイト・ストリームが含まれており、そのバイト・ストリームが3要素というサイズの上限を超えて、0xc5036eb0から始まるオブジェクトを上書きしていることが明確にわかります。

大きな手がかりが得られました。この後は、コード内で「Hello Java.Hello Java. . .」というバイト列が記述されている場所を検索することで、バイト配列のオーバーフローを引き起こしている問題のコードを容易に特定できます。**リスト14**に、この例のJNIコードでエラーの原因となっていた行を示します。このように、簡単にデバッグできました。

まとめ

オラクルのJVM Sustaining Engineering部門では、上記の例のように、Java HotSpot VMで発生するクラッシュやハングなどの問題をServiceability Agentを利用して毎日のようにデバッグしています。SAは便利で強力なデバッグ・ツールであり、Java HotSpot VMの内部について学ぶためにも役立ちます。本記事で、SAを理解していただけたことを願っています。SAを利用して、デバッグを楽しんでください。</article>

LEARN MORE

- SA-Plugin for VisualVM

Java
YOUR LOCAL JAVA USER GROUP NEEDS YOU
Find your JUG here