クラス・ローディング問題の神秘を解く 第１回: クラス・ローディングとデバッグ・ツール

クラス・ローディングの動作について、そしてクラス・ローディングの問題処理にJVMが役立つことを学ぶ

Lakshmi Shankar
Java Technology Center Development Team
IBM Hursley Labs

Simon Burns (simon_burns@uk.ibm.com)
Java Technology Center Development Team
IBM Hursley Labs

2005年 9月 29日

クラス・ローディング・コンポーネントは、Java™仮想マシンにとっての基本です。開発者は一般的にクラス・ローディングに関する基本はよく理解していますが、問題が発生した場合に診断を行い、そのソリューションを選択することには苦労しているようです。この4回シリーズの記事では、皆さんがJava開発を行う中で出会いがちなローディングに関する様々な問題を、Lakshmi ShankarとSimon Burnsが解説し、また問題発生の原因と、その解決方法を解説します。彼らの解説は、一般的なJava例外（NoClassDefFoundErrorやClassNotFoundExceptionなど）だけではなく、より困難な問題、つまりクラス・ローダー制約違反やデッドロックなども理解し、解決する上で、大いに役立つでしょう。第1回の今回は、Javaのクラス・ローディングがどのように動作するのか、またJVMで利用可能な、クラス・ローディングの問題を診断する上で役立つツールについて解説します。

クラス・ローダーはJVM（Java Virtual Machine）の中にクラスをロードすることに責任を持ちます。単純なアプリケーションでは、Javaプラットフォームに組み込みのクラス・ローディング機能を利用してクラスをロードします。より複雑なアプリケーションでは、多くの場合、独自のカスタム・クラス・ローダーを定義します。しかし、どんなクラス・ローダーを使うにせよ、クラス・ローディングというプロセスでは多くの問題が発生する可能性があります。そうした問題を避けるためには、クラス・ローディングの基本機構を理解する必要があります。どんな診断機能やデバッグ手法が利用できるかを知っていれば、実際に問題が発生した時にも容易に問題を解決できるはずです。

このシリーズでは、クラス・ローディングの問題を詳細に検討し、また分かりやすい例を使って、そうした問題を説明します。第1回目である今回は、前半のセクションでクラス・ローディン

グの基本を解説します。後半のセクションでは、JVMの持つデバッグ機能の一部を紹介します。今後の3回の記事では、クラス・ローディング例外の解決方法や、皆さんが出会いがちな、より面倒なクラス・ローディング問題を取り上げて行きます。

クラス・ローディングの基本

このセクションでは、クラス・ローディングの中核概念を説明し、ここから先に進むためのナレッジ・ベースを提供することにします。

クラス・ローダー委譲（Class loader delegation）

『クラス・ローダー委譲モデル（class loader delegation model）』は、お互いにローディング・リクエストを渡し合うクラス・ローダー群を表現したグラフです。このグラフのルートには、『ブートストラップ』クラス・ローダーがあります。クラス・ローダーは1つの『委譲の親（delegation parent）』によって作られ、下記の場所にあるクラスを探します。

- キャッシュ
- 親（parent）
- 自分自身（self）

クラス・ローダーは最初に、自分が今まで、この同じクラスをロードするように依頼されたかどうかを判断します。もし依頼されたことがあれば、前回返したものと同じクラス（つまり、キャッシュの中に保存されているクラス）を返します。もし依頼されたことがなければ、そのクラスをロードする機会を自分の親に対して与えます。この2つのステップは再帰的に繰り返されますが、まず深さ方向に行われます。もし親がヌルを返した場合（またはClassNotFoundExceptionを投げた場合）には、そのクラス・ローダーは、そのクラスのソースが自分自身のパスにないかどうかを探します。

親クラス・ローダーは、常に最初にクラスをロードする機会が与えられるため、最もルートに近いクラス・ローダーによって、そのクラスがロードされます。これはつまり、全てのコア・ブートストラップ・クラスは、ブートストラップ・ローダーによってロードされることを意味し、これによって適切なバージョンのクラス（例えばjava.lang.Objectなど）が確実にロードされます。またこれによる効果として、クラス・ローダーが見るのは自分自身がロードしたクラス、または自分の親、または祖先（ancestor）がロードしたクラスのみであり、自分の子がロードしたクラスは見えなくなります。

図1. クラス・ローダー委譲モデル

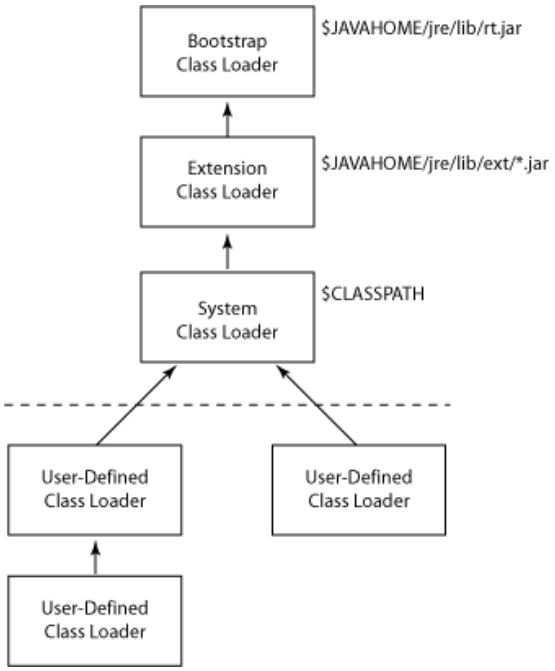


図1は、3つの標準クラス・ローダーを示しています。

ブートストラップ・クラス・ローダー（原始（primordial）クラス・ローダーとも言われます）は、他のクラス・ローダーとは異なり、Javaコードによってインスタンス化することはできません。（なぜなら多くの場合、VM自体の一部としてネイティブ実装されているためです。）このクラス・ローダーはコア・システム・クラスを、ブート・クラスパス（通常はjre/lib ディレクトリーにあるJARファイルです）からロードします。しかしこのクラスパスは、-Xbootclasspathコマンドライン・オプションを使って修正することができます（これについては後ほど説明します）。

『エクステンション』クラス・ローダー（『標準エクステンション』クラス・ローダーとも言われます）は、ブートストラップ・クラス・ローダーの子供です。このクラス・ローダーの主な責任は、エクステンション・ディレクトリー（通常はjre/lib/extディレクトリーにあります）からクラスをロードすることです。このようになっているため、ユーザーのクラスパスを何ら変更することなく、新しいエクステンション（様々なセキュリティー・オプションなど）を単純に追加できます。

『システム』クラス・ローダー（『アプリケーション』クラス・ローダーとも言われます）は、CLASSPATH環境変数で規定されるパスからコードをロードすることに責任を持つクラス・ローダーです。このクラス・ローダーは、デフォルトで、ユーザーが作成する任意のクラス・ローダーの親となっています。またこのクラス・ローダーは、ClassLoader.getSystemClassLoader() メソッドが返すクラス・ローダーでもあります。

クラスパス・オプション

表1は、この3つの標準クラス・ローダーのクラスパスを設定するためのコマンドライン・オプションの要約です。

表1. クラスパス・オプション

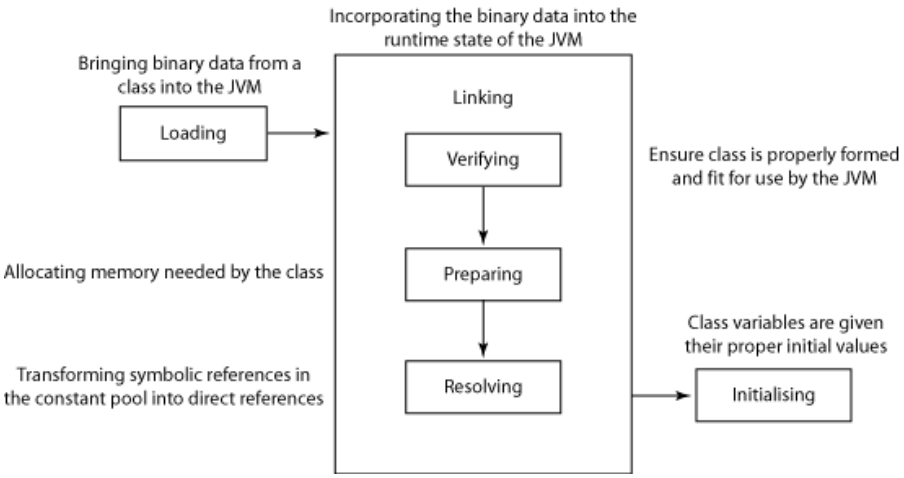
コマンドライン・オプション	説明	関係するクラス・ローダー
-Xbootclasspath:<*directories and zip/JAR files separated by ; or :***>**	ブートストラップ・クラスやリソース用のサーチ・パスを設定します。	ブートストラップ
-Xbootclasspath/a:<*directories and zip/JAR files separated by ; or :***>**	ブート・クラスパスの最後にパスを付加します。	ブートストラップ
-Xbootclasspath/p:<*directories and zip/JAR files separated by ; or :***>**	ブート・クラスパスの前にパスを付加します。	ブートストラップ
-Dibm.jvm.bootclasspath=<*directories and zip/JAR files separated by ; or :***>**	このプロパティーの値は、追加のサーチ・パスとして使われます。このパスは、-Xbootclasspath/p: で定義される任意の値とブート・クラスパスとの間に挿入されます。ブート・クラスパスは、デフォルトか、あるいは-Xbootclasspath: オプションで定義されるパスのいずれかです。	ブートストラップ
-Djava.ext.dirs=<*directories and zip/JAR files separated by ; or :***>**	エクステンション・クラスやリソース用のサーチ・パスを設定します。	エクステンション
-cp or -classpath <*directories and zip/JAR files separated by ; or :***>**	アプリケーション・クラスやリソース用のサーチ・パスを設定します。	システム
-Djava.class.path=<*directories and zip/JAR files separated by ; or :***>**	アプリケーション・クラスやリソース用のサーチ・パスを設定します。	システム

クラス・ローディングのフェーズ

クラスのローディングは、基本的に3つのフェーズ（ロード、リンク、初期化）に分解することができます。

クラス・ローディングに関する大部分の問題を突き詰めると、こうした3つのフェーズのどれかで発生していることが分かります。従って、それぞれのフェーズをよく理解すれば、クラス・ローディングに関する問題の診断も容易になります。図2は、これらのフェーズ示したものです。

図2. クラス・ローディングの各フェーズ



『ロード』フェーズは、必要なクラス・ファイルを見つけること（各クラスパスの検索）と、バイトコードのロード、という2段階から構成されています。JVM内部では、ローディング・プロセスがクラス・オブジェクトに対して非常に基本的なメモリー構造を与えます。このステージで

は、メソッドやフィールドその他、参照クラスは処理されません。そのため、クラスは使用できません。

『リンク』フェーズは、3つのフェーズのうち、最も複雑なフェーズです。このフェーズは次のような3つの主なステージに分解することができます。

- **バイトコード検証。**クラス・ローダーは、そのクラスの形式や振る舞いが適切かどうかを確認するために、そのクラスのバイトコードに対して幾つかのチェックを行います。
- **クラス準備。**このステージでは、フィールドやメソッド、実装インターフェース (各クラスの中で定義されます) を表現するために必要なデータ構造を整えます。
- **解決。**このステージでは、クラス・ローダーは、ある特定なクラスが参照する他の全クラスをロードします。こうしたクラスは、下記のような幾つかの方法で参照されます。
  - スーパークラス
  - インターフェース
  - フィールド
  - メソッド・シグニチャー
  - メソッド内で使用されるローカル変数

『初期化』フェーズでは、クラス内に含まれているすべての静的イニシャライザーが実行されます。このフェーズの最後では、静的フィールドが、そのデフォルト値に初期化されます。

こうした3つのフェーズの最後では、クラスは完全にロードされ、いつでも使用できるようになっています。ここで、クラス・ローディングは遅延方式(lazy manner)で行うこともできることに注意してください。つまりクラス・ローディング・プロセスの一部は、クラスをロードする時ではなく、最初にそのクラスを使う時に行うこともできるのです。

明示的ローディングと暗黙的ローディング

クラスをロードするには2つの方法があります。つまり明示的ローディングと暗黙的ローディングですが、両者の違いは僅かです。

『明示的』クラス・ローディングは、下記のメソッド・コールのいずれかを使ってクラスをロードする場合に発生します。

- cl.loadClass() (clはjava.lang.ClassLoaderのインスタンスです)
- Class.forName() (起動するクラス・ローダーが、カレント・クラスを定義するクラス・ローダーです)

これらのメソッドの1つが呼び出されると、そのクラスがクラス・ローダーによってロードされます (クラス名が引数です)。もしそのクラスが既にロードされている場合には、単純に参照が返されます。ロードされていない場合には、ローダーは委譲モデルを介して、そのクラスをロードします。

『暗黙的』クラス・ローディングは、参照やインスタンス化、継承など (明示的メソッド・コールによらないもの) の結果としてクラスがロードされる場合に発生します。これらの場合には、ローディングは裏で開始され、JVMが必要な参照を解決し、クラスをロードします。明示的クラス・ローディングの場合と同様、もしそのクラスが既にロードされている場合には、単純に参

照が返されます。ロードされていない場合には、ローダーは委譲モデルを介して、そのクラスをロードします。

多くの場合、クラスは、明示的クラス・ローディングと暗黙的クラス・ローディングの組み合わせによってロードされます。例えばクラス・ローダーは、あるクラスをまず明示的にロードし、その後、それが参照しているクラスを暗黙的にロードする、ということがあり得ます。

JVMのデバッグ機能

前のセクションでは、クラス・ローディングに関する基本的な原理を紹介しました。このセクションでは、IBMのJVMに組み込まれている様々なデバッグ補助機能について説明します。他のJVMの場合も、似たようなデバッグ機能を持っているかも知れません。詳しくは、そのJVMに関するドキュメンテーションを参照してください。

冗長出力

IBM JVMの冗長出力機能は、-verboseというコマンドライン・オプションを使ってオンします。冗長出力は、あるイベントが発生した場合（例えばクラスがロードされた、など）に、コンソール上に情報を表示します。クラス・ローディングに関する追加の情報を表示するためには、冗長クラス出力を使います。これは、-verbose:classオプションを使ってアクティブにすることができます。

冗長出力を解釈する

冗長出力には、開かれたJARファイルが全てリストアップされており、また、こうしたJARファイルへのフル・パスも含まれています。下記はその一例です。

```
...
[Opened D:\jre\lib\core.jar in 10 ms]
[Opened D:\jre\lib\graphics.jar in 10 ms]
...
```

ロードされた全クラスが、ロード元のJARファイルまたはディレクトリーと共にリストアップされます。下記はその一例です。

```
...
[Loaded java.lang.NoClassDefFoundError from D:\jre\lib\core.jar]
[Loaded java.lang.Class from D:\jre\lib\core.jar]
[Loaded java.lang.Object from D:\jre\lib\core.jar]
...
```

冗長クラス出力は、その他の情報（例えば、スーパークラスがロードされるのはいつか、静的イニシャライザーが実行されるのはいつか）なども表示します。下記は、そうした出力の例です。

```
...
[Loaded HelloWorld from file:/C:/myclasses/]
[Loading superclass and interfaces of HelloWorld]
[Loaded HelloWorldInterface from file:/C:/myclasses/]
[Loading superclass and interfaces of HelloWorldInterface]
[Preparing HelloWorldInterface]
[Preparing HelloWorld]
[Initializing HelloWorld]
[Running static initializer for HelloWorld]
...
```

また冗長出力は、（もし例外が発生した場合に）内部的に投げられる例外の幾つかを、スタック・トレースを含めて表示します。

-verboseを使って問題を解決する

冗長出力は、クラスパスの問題を解決するために役立ちます（例えば、開かれていないJARファイル（従ってクラスパス上にありません）や、誤った場所からロードされたクラスの問題など）。

IBM Verbose Class Loading

クラス・ローダーはどこでクラスを探すのか、ある特定なクラスをロードするのはどのクラス・ローダーか、などが分かると便利な場合がよくあります。こうした情報は、IBM Verbose Class Loadingのコマンドライン・オプション、-Dibm.cl.verbose=<class name>を使って取得することができます。正規表現を使ってクラス名を宣言できるのです。例えばHello*は、名前がHelloで始まるすべてのクラスを追跡します。

このオプションは、ユーザー定義のクラス・ローダーに対しても、（それらが直接、間接にjava.net.URLClassLoaderを継承している限り）動作します。

IBM Verbose Class Loadingの出力を解釈する

IBM Verbose Class Loadingの出力は、指定されたクラスをロードしようとするクラス・ローダーと、そのクラス・ローダーが探そうとしている場所を表示します。例えば、次のようなコマンドラインを使ったとしましょう。

```
java -Dibm.cl.verbose=ClassToTrace MainClass
```

ここでMainClassは、そのmainメソッドの中でClassToTraceを参照しています。このコマンドで、ここに示すような出力が得られます。

MainClassが、そのmainメソッドの中でClassToTraceを参照する

```
ExtClass loader attempting to find ClassToTrace
ExtClass loader using classpath D:\jre\lib\ext\gskikm.jar;D:\jre\lib\ext\ibmjceprovider.jar;
 D:\jre\lib\ext\indicim.jar;
 D:\jre\lib\ext\jaccess.jar;D:\jre\lib\ext\ldapsec.jar;
 D:\jre\lib\ext\oldcertpath.jar
ExtClass loader could not find ClassToTrace.class in D:\jre\lib\ext\gskikm.jar
ExtClass loader could not find ClassToTrace.class in D:\jre\lib\ext\ibmjceprovider.jar
ExtClass loader could not find ClassToTrace.class in D:\jre\lib\ext\indicim.jar
ExtClass loader could not find ClassToTrace.class in D:\jre\lib\ext\jaccess.jar
ExtClass loader could not find ClassToTrace.class in D:\jre\lib\ext\ldapsec.jar
ExtClass loader could not find ClassToTrace.class in D:\jre\lib\ext\oldcertpath.jar
ExtClass loader could not find ClassToTrace
AppClass loader attempting to find ClassToTrace
AppClass loader using classpath C:\tests;D:\lib\tools.jar
AppClass loader could not find ClassToTrace.class in C:\tests
AppClass loader could not find ClassToTrace.class in D:\lib\tools.jar
AppClass loader could not find ClassToTrace
Exception in thread "main" java.lang.NoClassDefFoundError: ClassToTrace at MainClass(MainClass.java:6)
```

クラス・ローダーは、親が子の前に来る形でリストアップされています。これは、標準の委譲モデルがこうなっているからです。つまり、親が先なのです。

ブートストラップ・クラス・ローダーに対する出力が何もないことに注意してください。出力は、java.net.URLClassLoaderを継承するクラス・ローダーに対してのみ生成されます。また、クラス・ローダーは、そのクラス名に従ってリストアップされていることにも注意してください。つまり、あるクラス・ローダーに2つのインスタンスがある場合は、両者を区別できないこともあり得ます。

IBM Verbose Class Loadingを使って問題を解決する

IBM Verbose Class Loadingオプションは、すべてのクラス・ローダーのクラスパスがどのように設定されているかをチェックする場合に非常に便利です。またこのオプションは、対象とするクラスをロードするのがどのクラス・ローダーか、そのクラスをどこからロードしているのかも示してくれます。このため、適切なバージョンのクラスがロードされているかどうかを容易に調べることができます。

Javadump

『Javadump』（『Javacore』とも言われます）は、また別のIBM診断ツールです。これも便利なツールです。このツールについて詳しく知るためには、IBM Diagnostics Guide（参考文献にリンクがあります）を見てください。Javadumpは、下記のいずれかのイベントが発生した場合に、JVMによって生成されます。

- 致命的なネイティブ例外が発生した
- JVMにヒープ・スペースが足りなくなった
- JVMにシグナルが送られた（例えば、WindowsでControl-Breakが押された、またはLinuxでControl-\が押された、など）
- com.ibm.jvm.Dump.JavaDump() メソッドが呼ばれた

Javadumpがトリガーされると、その瞬間に、日付スタンプ付きのテキスト・ファイルに詳細な情報が記録され、カレントの作業ディレクトリーに保存されます。この情報には、スレッドやロック、スタックなどのデータの他、システム中のクラス・ローダーに関する豊富な情報が含まれています。

Javadumpのクラス・ローダー部分を解釈する

Javadumpファイルの中に提供されている具体的な情報は、JVMが実行しているプラットフォームに依存します。クラス・ローダーに関する部分には、次のような情報が含まれています。

- 定義されたクラス・ローダーと、それらの間の関係
- 各クラス・ローダーがロードしたクラスのリスト

下記は、Javadumpから取ったクラス・ローダー情報のスナップショットです。

```
CL subcomponent dump routine
============================
 Classpath Z(D:\jre\lib\core.jar),...
 Oldjava mode false
 Bootstrapping false
 Verbose class dependencies false
 Class verification VERIFY_REMOTE
 Namespace to classloader 0x00000000
```

```
Start of cache entry pool 0x44D85430
Start of free cache entries 0x44D86204
Location of method table 0x44C23AA0
Global namespace anchor 0x00266894
System classloader shadow 0x00376068
Classloader shadows 0x44D7BA60
Extension loader 0x00ADB830
System classloader 0x00ADB7B0
Classloader summaries
        12345678: 1=primordial,2=extension,3=shareable,4=middleware,
                  5=system,6=trusted,7=application,8=delegating
        -----ta- Loader sun/misc/Launcher$AppClassLoader(0x44D7BA60),
                 Shadow 0x00ADB7B0,
                 Parent sun/misc/Launcher$ExtClassLoader(0x00ADB830)
                Number of loaded classes 1
                Number of cached classes 260
                Allocation used for loaded classes 1
                Package owner 0x00ADB7B0
        -xh-st-- Loader sun/misc/Launcher$ExtClassLoader(0x44D71288),
                 Shadow 0x00ADB830,
                 Parent *none*(0x00000000)
                Number of loaded classes 0
                Number of cached classes 0
                Allocation used for loaded classes 3
                Package owner 0x00ADB830
        p-h-st-- Loader *System*(0x00376068), Shadow 0x00000000
                Number of loaded classes 304
                Number of cached classes 304
                Allocation used for loaded classes 3
                Package owner 0x00000000
ClassLoader loaded classes
        Loader sun/misc/Launcher$AppClassLoader(0x44D7BA60)
                HelloWorld(0x00ACF0E0)
        Loader sun/misc/Launcher$ExtClassLoader(0x44D71288)
        Loader *System*(0x00376068)
                java/io/WinNTFileSystem(0x002CD118)
                java/lang/Throwable(0x002C03A8)
                java/lang/IndexOutOfBoundsException(0x44D45208)
                java/lang/UnsatisfiedLinkError(0x44D42D38)
.....................classes left out to save space.......................
                [Ljava/lang/Class;(0x002CA9E8)
                java/io/InputStream(0x002C9818)
                java/lang/Integer$1(0x002C83E8)
                java/util/Dictionary(0x002C4298)
```

この例では、3つの標準クラス・ローダーしかありません。

- システム・クラス・ローダー（ sun/misc/Launcher$AppClass loader ）
- エクステンション・クラス・ローダー（ sun/misc/Launcher$ExtClass loader ）
- ブートストラップ・クラス・ローダー（ *System* ）

Classloader summaries（ 要約 ）セクションは、クラス・ローダーのタイプを含めて、システム中にある各クラス・ローダーの詳細を示しています。このシリーズ記事で注目するタイプとしては、『primordial』、『extension』、『system』、『application』、『delegation』です（ リフレクションに使用されます ）。他のタイプ（ 『shareable』、『middleware』、『trusted』 ）は、Persistent Reusable JVMで使われますが、これはこのシリーズの範囲外です。（ 詳しくは、Persistent Reusable JVM User Guideを見てください。参考文献のセクションにリンクがあります。 ）このsummariesセクションは、親クラス・ローダーも示しています。システム・クラス・ローダーの親は、sun/misc/Launcher$ExtClass loader(0x00ADB830) です。この親アドレスは、親クラス・ローダー（ 『shadow』と言われます ）のネイティブ・データ構造に対応します。

ClassLoader loaded classes（ClassLoaderがロードしたクラス）のセクションは、各クラス・ローダーがロードしたクラスをリストアップしています。この例では、システム・クラス・ローダーは1つのクラス（HelloWorld、アドレス0x00ACF0E0）しかロードしていません。

Javadumpsを使って問題を解決する

Javadumpが提供する情報を使うと、どのクラス・ローダーがシステム内に存在しているかを確かめることができます。この中には、ユーザー定義のクラス・ローダーも全て含まれます。ロードされたクラスのリストを見れば、ある特定なクラスをロードしたのはどのクラス・ローダーか、を知ることができます。もしそのクラスが見つからなければ、システム中に存在するどのクラス・ローダーも、そのクラスをロードしなかったことになります（これは通常の場合、ClassNotFoundExceptionとなります）。

Javadumpを使って診断できる問題タイプには、他にも次のようなものがあります。

- クラス・ローダー名前空間の問題。クラス・ローダー名前空間は、クラス・ローダーと、そのクラス・ローダーがロードした全クラスの組み合わせです。例えば、ある特定なクラスが、誤ったクラス・ローダーによってロードされた場合（NoClassDefFoundErrorが起きる場合があります）には、名前空間が不正です。つまり、そのクラスが誤ったクラスパス上にあるのです。こうした問題を正すには、そのクラスを別な場所（例えば通常のJavaクラスパスなど）に置いてみます。そしてそのクラスが、システム・クラス・ローダーによって正常にロードされることを確認します。
- クラス・ローダー制約の問題。このタイプの問題の例については、このシリーズの最終回で解説することにします。

Javaメソッドのトレース

IBM JVMは、組み込みのメソッド・トレース機能を持っています。この機能を利用すると、任意のJavaコード（コア・システム・クラスを含みます）の中にあるメソッドを、コードに何ら修正を加えることなくトレースすることができます。この機能から得られるデータは膨大なので、自分に必要な情報に絞り込むにはトレース・レベルをコントロールします

トレースをイネーブルにするためのオプションは、JVMのリリースによって異なります。これらのオプションの詳細に関しては、IBM Diagnostics Guidesを参照してさい（参考文献にリンクがあります）。

下記にコマンドラインの例を幾つか示します。

IBM Java 1.4.2でHelloWorldを実行中に、すべてのjava.lang.ClassLoaderメソッドをトレースする場合:

```
java -Dibm.dg.trc.methods=java/lang/ClassLoader.*() -Dibm.dg.trc.print=mt HelloWorld
```

同じくIBM Java 1.4.2で、ClassLoaderのloadClass() メソッドとHelloWorldのメソッドをトレースする場合:

```
java -Dibm.dg.trc.methods=java/lang/ClassLoader.loadClass(),HelloWorld.*()
  -Dibm.dg.trc.print=mt HelloWorld
```

メソッド・トレース出力を解釈する

メソッド・トレース出力の例はここにあります（前のセクションの2番目のコマンドラインを使っています）。

メソッド・トレース出力の例

```
 ...
 > java/lang/ClassLoader.loadClass Bytecode method, This = 0x00D2B7B0, Signature: (Ljava/lang/String;)Ljava/
lang/Class;
  > java/lang/ClassLoader.loadClass Bytecode method, This = 0x00D2B7B0, Signature: (Ljava/lang/
String;Z)Ljava/lang/Class;
   > java/lang/ClassLoader.loadClass Bytecode method, This = 0x00D2B830, Signature: (Ljava/lang/
String;Z)Ljava/lang/Class;
   < * java/lang/ClassLoader.loadClass Bytecode method, looking for matching catch block for
 java.lang.ClassNotFoundException
  < java/lang/ClassLoader.loadClass Bytecode method
 < java/lang/ClassLoader.loadClass Bytecode method
 > HelloWorld.main Bytecode static method, Signature: ([Ljava/lang/String;)V
 < HelloWorld.main Bytecode static method
```

トレースの各ラインからは、上に示したものよりも、さらに多くの情報が得られます。上記のうちの1ラインを、完全に見てみましょう。

```
12:57:41.277 0x002A23C0 04000D   > java/lang/ClassLoader.loadClass Bytecode method,
       This = 0x00D2B830, Signature: (Ljava/lang/String;Z)Ljava
/lang/Class;
```

このトレースには、下記が含まれています。

- 12:57:41.277: メソッドへの入り口と出口でのタイムスタンプ
- 0x002A23C0: スレッドID
- 04000D: 一部の高度な診断で使用される、内部JVMトレース・ポイント
- その他の情報は、そのメソッドに入っているか（>）あるいはそのメソッドから出たか（<）を示し、またその後に、そのメソッドについての詳細を示しています。

メソッド・トレースで問題を解決する

メソッド・トレースを使用すると、次のような様々なタイプの問題を解決することができます。

- パフォーマンス・ホットスポット: タイムスタンプを使用すると、実行に大幅な時間を要しているメソッドを見つけることができます。
- ハングアップ: 最後にそのメソッドに入った、ということから、そこでアプリケーションがハングアップしたと分かる場合が多いものです。
- 不正なオブジェクト: アドレスを利用すると、そのオブジェクトに対するコンストラクター・コールのアドレスと一致比較を行い、適切なオブジェクトに対してメソッドが呼び出されているかどうかをチェックすることができます。
- 予期しないコード・パス: 入力ポイントと出力ポイントを追うことによって、予期せぬコード・パスをプログラムが通っていないかを調べることができます。
- その他の失敗: 最後にそのメソッドに入った、ということから、そこで失敗が発生したと分かる場合が多いものです。

次回は

今回の記事では、JVMでのクラス・ローディングの基本と、IBMのJVMに用意されているデバッグ機能について学びました。次回の記事では、この知識を応用して、Javaアプリケーションを実行する際に出会いがちなクラス・ローディングに関する様々な問題を理解し、解決して行きます。

著者について

Lakshmi Shankar

Lakshmi Shankarは、イギリスのIBM Hursley Labsのソフトウェア技術者です。IBMで3年以上働いており、Javaパフォーマンスやテスト、開発など、Hursley Labsで幅広い経験を積んでいます。最近まで、IBM Java技術のクラス・ローディング・コンポーネントの所有者でした。現在は、情報管理チームの開発者の一員です。

Simon Burns

Simon Burnsは、IBM Hursley LabsのJava Technology Centreにおける、Persistant Reusable JVM のコンポーネント所有者であり、チームリーダーでした。JVM開発に3年以上従事しており、Persistant Reusable JVM技術とz/OSプラットフォームを専門にしています。またCICSとも緊密に作業しながら、この技術の活用のために協力してきました。オープンソースのEclipse Equinoxプロジェクト（現在はEclipse 3.1に統合されています）の一員として、OSGiフレームワークに従事した経験もあります。現在は、コンポーネント化に取り組んでいます。