# クラス・ローディング問題の神秘を解く 第2回: クラス・ローディングに関する基本的な例外

## クラス・ローディングに関する単純な問題と難問を詳しく調べる

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

Lakshmi Shankar
Java Technology Center Development Team
IBM Hursley Labs

2005年 12月 06日

この4回シリーズの記事では、アプリケーション開発者が問題を理解し、デバッグできるようになることを目標に、Java™のクラス・ローディングを検証しています。この第2回では、IBM Hursley Labs のLakshmi ShankarとSimon Burnsが、幾つかの例外に関して取り上げます。これらは非常に簡単なものですが、新人の開発者でも経験豊かな開発者でも、しばしば戸惑いがちな問題でもあります。

4回シリーズの2回目である今回は、アプリケーションを実行する際によく投げられる、様々なクラス・ローディング例外について検証します。これらの例外は一般的に見られるものですが、Java開発者は、必ずしもこれらをよく理解していないようです。この記事では、そうした例外を順次取り上げながら、それらの振る舞いをハイライトする詳細な例を挙げ、その原因を説明し、そして考えられる解決手法を示します。最初に取り上げるのは非常に一般的なClassNotFoundExceptionですが、ExceptionInInitializerErrorのような、あまり知られていない例外も取り上げることにします。

この記事を読むためには、クラス・ローダーの委譲モデル(delegation model)や、クラスのリンクに関するフェーズやステージについて慣れている必要があります。まず、このシリーズの最初の記事を読むことから始めるようにお勧めします。

## ClassNotFoundException

ClassNotFoundExceptionは、最も一般的なタイプのクラス・ローディング例外です。これは、ローディング・フェーズで発生します。Java仕様では、ClassNotFoundExceptionを次のように記述しています。

アプリケーションが、

- Classクラスの中でforName() メソッドを使ったストリング名によって
- ClassLoaderクラスの中でfindSystemClass method() メソッドを使ったストリング名によって
- ClassLoaderクラスの中でloadClass() メソッドを使ったストリング名によって

  クラスの中にロードしようとした時、規定された名前を持つクラスに対する定義が見つからない場合に投げられます。

つまりClassNotFoundExceptionは、クラスを明示的にロードしようとする試みが失敗した場合に投げられるのです。リスト1に示すテスト・ケースは、ClassNotFoundExceptionを投げるコードの例です。

## リスト1.ClassNotFoundExceptionTest.java

```
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLClassLoader;

public class ClassNotFoundExceptionTest {

    public static void main(String args[]) {
        try {
            URLClassLoader loader = new URLClassLoader(new URL[] { new URL(
                "file://C:/CL_Article/ClassNotFoundException/")});
            loader.loadClass("DoesNotExist");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
    }
}
```

このテスト・ケースは、クラス・ローダー（ MyClassLoader ）を定義しています。このクラス・ローダーは、存在していないクラス（ DoesNotExist ）をロードするために使われています。これを実行すると、次のような例外が発生します。

```
java.lang.ClassNotFoundException: DoesNotExist
    at java.net.URLClassLoader.findClass(URLClassLoader.java:376)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:572)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:504)
    at ClassNotFoundExceptionTest.main(ClassNotFoundExceptionTest.java:11)
```

このテストはloadClass() への明示的コールを使ってロードを試みるため、ClassNotFoundExceptionが投げられます。

クラス・ローダーはClassNotFoundExceptionを投げることによって、クラス・ローダーが探そうとする場所に、そのクラスを定義するために必要なバイトコードが存在していないことを知らせます。こうした例外は、通常は簡単に修正することができます。使用されているクラスパスが想定通り設定されていることは、IBMの冗長オプションを使ったチェックで確認することができます。（ このオプションの詳細については、第1回目の記事を見てください。 ）もし、クラスパスが正しく設定されているにもかかわらず、やはりエラーが出る場合には、必要なクラスがクラスパス上に無いということです。これを修正するためには、そのクラスを、クラスパスで指定されているディレクトリーまたはJARファイルに移動するか、あるいは、そのクラスが保存されている位置をクラスパスに追加します。

## NoClassDefFoundError

NoClassDefFoundErrorも、ロード・フェーズ中にクラス・ローダーが投げる一般的な例外です。JVM仕様では、NoClassDefFoundErrorを次のように定義しています。

> Java仮想マシンまたはClassLoaderインスタンスが、（通常のメソッド・コールの一部として、または新しい表現を使って新しいインスタンスを作成する一部として）クラスの定義の中にロードしようとして、そのクラスの定義が見つからない時に投げられます。
>
> 検索対象のクラス定義が、現在実行中のクラスをコンパイルした時には存在したにもかかわらず、もはや見つかりません。

要は、NoClassDefFoundErrorは暗黙的クラス・ロードが不成功に終わった結果として投げられる、ということです。

リスト2からリスト4のテスト・ケースは、クラスBの暗黙的ロードが失敗するため、NoClassDefFoundErrorを起こします。

### リスト2.NoClassDefFoundErrorTest.java

```
public class NoClassDefFoundErrorTest {
 public static void main(String[] args) {
  A a = new A();
 }
}
```

### リスト3.A.java

```
public class A extends B {
}
```

### リスト4.B.java

```
public class B {
}
```

これらのリストにあるコードをコンパイルした後、Bに対するクラスファイルを削除します。そうすると、コードを実行した時に、次のようなエラーが起こります。

```
Exception in thread "main" java.lang.NoClassDefFoundError: B
    at java.lang.ClassLoader.defineClass0(Native Method)
    at java.lang.ClassLoader.defineClass(ClassLoader.java:810)
    at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:147)
    at java.net.URLClassLoader.defineClass(URLClassLoader.java:475)
    at java.net.URLClassLoader.access$500(URLClassLoader.java:109)
    at java.net.URLClassLoader$ClassFinder.run(URLClassLoader.java:848)
    at java.security.AccessController.doPrivileged1(Native Method)
    at java.security.AccessController.doPrivileged(AccessController.java:389)
    at java.net.URLClassLoader.findClass(URLClassLoader.java:371)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:572)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:442)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:504)
    at NoClassDefFoundErrorTest.main(NoClassDefFoundErrorTest.java:3)
```

クラスAはクラスBを継承しています。そのためクラスAがロードされると、クラス・ローダーは暗黙的にクラスBをロードします。クラスBは存在しないため、NoClassDefFoundErrorが投げられます。もしクラス・ローダーが、（例えばloadClass("B") コールによって）クラスBをロードするように明示的に指示された場合は、代わりにClassNotFoundExceptionが投げられたはずです。

明らかなことですが、この例固有の問題を修正するには、適切なクラス・ローダーのクラスパス上にクラスBが存在する必要があります。この例は非常に単純で非現実的に思えるかも知れませんが、無数のクラスが存在する複雑な現実のシステムでも、パッケージングやデプロイの間にクラスが見落とされたような場合に、この例のような状況は発生するのです。

この例では、AはBを継承しています。しかし、もしAがBを他の方法で（例えばメソッド・パラメーターとして、あるいはインスタンス・フィールドとして）参照している場合には、やはり同じエラーが発生します。もし2つのクラスの間での関係が、継承ではなく参照の関係である場合は、Aのロード中ではなく、最初にAをアクティブに使用しようとする時にエラーが投げられます。

## ClassCastException

クラス・ローダーによって投げられる、もう一つの例外は、ClassCastExceptionです。これは、タイプ比較の中に互換性のないタイプが見つかった結果として投げられます。JVM仕様では、ClassCastExceptionについて、次のように言っています。

> コードが、サブクラスに対して（そのサブクラスのインスタンスではない）オブジェクトをキャストしようとしたことを示すために投げられます。

リスト5は、ClassCastExceptionを挙げるコードの例を説明しています。

### リスト5.ClassCastException.java

```
public class ClassCastExceptionTest {
    public ClassCastExceptionTest() {
    }

    private static void storeItem(Integer[] a, int i, Object item) {
        a[i] = (Integer) item;
    }

    public static void main(String args[]) {
        Integer[] a = new Integer[3];
        try {
            storeItem(a, 2, new String("abc"));
        } catch (ClassCastException e) {
            e.printStackTrace();
        }
    }
}
```

リスト5では、storeItem() メソッドが呼ばれ、Integer配列とint、そしてストリングが渡されます。しかし内部的には、このメソッドは次の2つを行っています。

- （パラメーター・リストに対して）Stringオブジェクト・タイプをObjectタイプに暗黙的にキャストする

- このObjectタイプをIntegerタイプに対して（メソッド定義の中で）明示的にキャストする

このプログラムが実行されると、次の例外が発生します。

```
java.lang.ClassCastException: java.lang.String
    at ClassCastExceptionTest.storeItem(ClassCastExceptionTest.java:6)
    at ClassCastExceptionTest.main(ClassCastExceptionTest.java:12)
```

このテスト・ケースはStringタイプの何かをIntegerに変換しようとしているため、明示的キャストによって例外が投げられるのです。

テスト対象のオブジェクト（例えばリスト5のitemなど）と、キャストの対象となるターゲット・クラス（Integer）が与えられると、クラス・ローダーは下記の規則をチェックします。

- **通常のオブジェクト（配列ではないもの）の場合**:オブジェクトは、ターゲット・クラスのインスタンスまたはターゲット・クラスのサブクラスである必要がある。もしターゲット・クラスがインターフェースであるならば、そのクラスは（そのインターフェースを実装する場合には）サブクラスと考えられる。
- **配列タイプの場合**:ターゲット・クラスは配列タイプであるか、あるいはjava.lang.Object, java.lang.Cloneableまたはjava.io.Serializableでなければならない。

もし、上記の規則のいずれかに違反している場合には、クラス・ローダーによってClassCastExceptionが投げられます。こうした例外を修正するために最も容易な方法は、オブジェクトのキャスト先のタイプが上記の規則に従っているかどうかを注意深くチェックすることです。場合によっては、クラスのキャストを行う前にinstanceofチェックを使用した方が賢明かも知れません。

## UnsatisfiedLinkError

クラス・ローダーは、対応する適切なネイティブ定義にネイティブ・コールをリンクする上で、重要な役割を果たします。UnsatisfiedLinkErrorは、リンク・フェーズの解決ステージの期間中、存在しない、あるいはどこかに置き忘れられているネイティブ・ライブラリーをプログラムがロードしようとした場合に発生します。JVM仕様では、次のような場合にUnsatisfiedLinkErrorが投げられると規定しています。

> Java仮想マシンが、nativeと宣言されたメソッドに対する適切なネイティブ言語定義を見つけられなかった場合

ネイティブ・メソッドが呼び出されると、クラス・ローダーは、そのメソッドを定義するネイティブ・ライブラリーをロードしようとします。もし、このライブラリーが見つからないと、このエラーが投げられます。

リスト6は、UnsatisfiedLinkErrorを投げるテスト・ケースを示しています。

## リスト6.UnsatisfiedLinkError.java

```
public class UnsatisfiedLinkErrorTest {

    public native void call_A_Native_Method();

    static {
        System.loadLibrary("myNativeLibrary");
    }

    public static void main(String[] args) {
        new UnsatisfiedLinkErrorTest().call_A_Native_Method();
    }
}
```

このコードは、ネイティブ・メソッド、call_A_Native_Method() を呼び出します ( このメソッドはmyNativeLibraryというネイティブ・ライブラリーの中で定義されています )。このライブラリーは存在しないため、プログラムを実行すると次のようなエラーが発生します。

```
The java class could not be loaded. java.lang.UnsatisfiedLinkError:
  Can't find library myNativeLibrary  (myNativeLibrary.dll)
  in sun.boot.library.path or java.library.path
sun.boot.library.path=D:\sdk\jre\bin
java.library.path= D:\sdk\jre\bin

at java.lang.ClassLoader$NativeLibrary.load(Native Method)
    at java.lang.ClassLoader.loadLibrary0(ClassLoader.java:2147)
    at java.lang.ClassLoader.loadLibrary(ClassLoader.java:2006)
    at java.lang.Runtime.loadLibrary0(Runtime.java:824)
    at java.lang.System.loadLibrary(System.java:908)
    at UnsatisfiedLinkErrorTest.<clinit>(UnsatisfiedLinkErrorTest.java:6)
```

ネイティブ・ライブラリーのロードは、System.loadLibrary() を呼ぶクラスのクラス・ローダー ( リスト6では、UnsatisfiedLinkErrorTestのクラス・ローダー ) によって開始されます。これがどのようなクラス・ローダーであるかによって、どこを検索するかは異なります。

- ブートストラップ・クラス・ローダーがロードするクラスの場合は、sun.boot.library.pathを検索します。
- エクステンション・クラス・ローダーがロードするクラスの場合は、まずjava.ext.dirsを検索し、次にsun.boot.library.path、その次にjava.library.pathを検索します。
- システム・クラス・ローダーがロードするクラスの場合は、まずsun.boot.library.pathを検索し、次にjava.library.pathを検索します。

リスト6では、システム・クラス・ローダーによってUnsatisfiedLinkErrorTestクラスがロードされています。このクラス・ローダーは、参照対象のネイティブ・ライブラリーをロードするためにsun. boot.library.pathを検索し、次にjava.library.pathを検索します。このライブラリーは、どちらの場所にも存在しないため、クラス・ローダーはUnsatisfiedLinkageErrorを投げます。

そのライブラリーのロードに関係するクラス・ローダーを理解できさえすれば、そのライブラリーを適切な場所に置くことによって、この種の問題は解決できるものです。

## ClassCircularityError

JVM仕様では、下記の場合にClassCircularityErrorが投げられると言っています。

> クラスまたはインターフェースが、自分自身のスーパークラスまたはスーパーインターフェースであるためにロードできなかった場合

このエラーは、リンク・フェーズの解決ステージで投げられます。Javaコンパイラーは、そのような循環状況が発生することを許していないため、これは少しおかしなエラーと言えます。しかし、別々にクラスをコンパイルしてから一緒にしたような場合には、このエラーが起こる場合があります。次のようなシナリオを考えてみてください。まず、リスト7と8のクラスをコンパイルします。

### リスト7.A.java

```
public class A extends B {
}
```

### リスト8.B.java

```
public class B {
}
```

次に、上記とは別に、リスト9と10のクラスをコンパイルします。

### リスト9.A.java

```
public class A {
}
```

### リスト10.B.java

```
public class B extends A {
}
```

最後に、リスト7のクラスAと、リスト10のクラスBとを持ってきて、AまたはBをロードしようとするアプリケーションを実行します。これは、あり得ない状況と思えるかも知れませんが、数多くの別々な部分を一緒に組み合わせるような複雑なシステムでは、非常に起こりがちなことなのです。

当然ですが、この問題を修正するためには、クラスの階層構造が循環的にならないようにすることです。

## ClassFormatError

JVM仕様では、下記の場合にClassFormatErrorが投げられると言っています。

> 要求されたコンパイル済みクラスまたはインターフェースを規定するはずの、バイナリー・データの形式がおかしい場合

この例外は、クラス・ローディングでのリンク・フェーズの検証ステージで投げられます。もしバイトコードが変更されてしまうと（例えば、重要な、あるいは些細な数字が変更された、など）、バイナリー・データの形式がおかしくなる可能性があります。このエラーが発生するのは、例えばバイトコードが故意にハッキングされた場合や、ネットワーク経由でクラスファイルを転送するような場合です。

この問題を修正するには、（場合によっては再コンパイルを行って）正しく修正されたバイトコードのコピーを入手する他ありません。

## ExceptionInInitializerError

JVM仕様によると、ExceptionInInitializerが投げられるのは次のような場合です。

- イニシャライザーが、ある例外、Eを投げることによって突然終了し、EのクラスがErrorではない、またはErrorのサブクラスではない場合であって、ExceptionInInitializerErrorというクラスの新しいインスタンスが（Eを引数として）作られ、Eの代わりに使われた場合
- JVMがExceptionInInitializerErrorというクラスの新しいインスタンスを作ろうとしたものの、Out-Of-Memory-Errorが発生したため、それができず、代わりにOutOfMemoryErrorオブジェクトが投げられた場合

リスト8のコードはExceptionInInitializerErrorを投げます。

### リスト8.ExceptionInInitializerErrorTest.java

```
public class ExceptionInInitializerErrorTest {
    public static void main(String[] args) {
        A a = new A();
    }
}

class A {
    // If the SecurityManager is not turned on, a
    // java.lang.ExceptionInInitializerError will be thrown
    static {
        if(System.getSecurityManager() == null)
            throw new SecurityException();
    }
}
```

静的コード・ブロックの中で例外が発生すると、このエラーは自動的に捕捉され、ExceptionInInitializerErrorでラップされます。これは、次のような出力を見ると分かるでしょう。

```
Exception in thread "main" java.lang.ExceptionInInitializerError
   at ExceptionInInitializerErrorTest.main(ExceptionInInitializerErrorTest.java:3)
Caused by: java.lang.SecurityException
   at A.<clinit>(ExceptionInInitializerErrorTest.java:12)
   ... 1 more
```

このエラーは、クラス・ローディングの初期化フェーズで投げられます。これを修正するための方法としては、ExceptionInInitializerError（Caused by: の下のスタック・トレースに示されます）を引き起こした例外をよく調べ、この例外が投げられないようにする方法を見つけることです。

## 次回は

今回の記事では、クラス・ローディングに関わる例外について、非常に基本的なものから少し分かりにくいものまで、様々なものを学びました。次回では、クラス・ローディングに関するその他の問題として、より複雑なアプリケーションを実行する際に発生しがちな問題を取り上げる予定です。

# 著者について

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。

---

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。