Java Native Interface を使用する上でのベスト・プラクティス

JNI プログラミングで最も一般的な 10 のミスを避けるための手法とツール

Michael Dawson
Advisory Software Developer
IBM

Graeme Johnson
J9 Virtual Machine Development Manager
IBM

Andrew Low
STSM, J9 Virtual Machine
IBM

2009年 7月 07日

JNI (Java™ Native Interface) は、Java コードと他のプログラミング言語で作成されたコードとの統合を可能にする標準 Java API です。例えば、これまでに蓄積した既存のコード資産をサービス指向アーキテクチャー (SOA) やクラウド・ベースのシステムで使用したいと思ったら、JNI はツールキットになくてはならない存在です。その一方、然るべき注意を払って JNI を使用しなければ、瞬く間にパフォーマンスに劣る、安定性のないアプリケーションという結果になってしまいます。そこで、この記事では代表的な 10 の JNI プログラミングの落とし穴を明らかにして、それぞれの落とし穴に陥らないようにするためのベスト・プラクティスを説明するとともに、これらのプラクティスを実践する上で利用できるツールを紹介します。

アプリケーションを開発するには、Java 環境と Java 言語が無難であり、効率的です。そうは言っても、アプリケーションによっては、純粋な Java プログラム内では実行できないタスクを実行しなければならないこともあります。これに該当するタスクには、例えば以下のものがあります。

JNI の進化

JNI が Java プラットフォームに組み込まれるようになったのは JDK 1.1 リリースからです。その後、JNI は JDK 1.2 リリースで拡張されました。JDK 1.0 リリースには初期のネイティブ・メソッド・インターフェースが組み込まれていましたが、これにはネイティブ・コードと Java コードとが明確に分離されていませんでした。このようなインターフェースではネイティブ・コードが JVM 構造に直接アクセスすることになるため、JVM 実装の間でも、プラッ

トフォーム間でも、さらには JDK のバージョン間でも移植することは不可能です。そのため、JDK 1.0 モデルに従った大量のネイティブ・コードを使用するアプリケーションをアップグレードするのは、複数の異なる JVM 実装で実行可能なネイティブ・コードを開発するのと同じくらいにコストがかかるという状況でした。

JDK 1.1 リリースでの JNI 導入によって、以下のことが実現しました。

- バージョン間の独立性
- プラットフォーム間の独立性
- VM 間の独立性
- サード・パーティー・クラス・ライブラリーの開発

興味深い点として、PHP などの新しい言語ではネイティブ・コードのサポートという点で、上記の問題にまだ四苦八苦しています。

- コードを作成し直さなくても済むように、既存のレガシー・コードを統合する。
- 使用可能なクラス・ライブラリーにはない機能を実装する。例えば Java 言語で `ping` を実装している一方、基本クラス・ライブラリーに ICMP (Internet Control Message Protocol) 機能がないというのであれば、この機能が必要になります。
- パフォーマンスやその他の環境固有のシステム特性を最大限利用するために C/C++ で作成されたコードを統合する。
- Java 以外のコードを必要とする特殊な状況に対応する。例えば、コア・クラス・ライブラリーの実装には、パッケージ間での呼び出しや、他の Java セキュリティー・チェックをバイパスする必要が考えられます。

これらのタスクを実現させてくれるのが、JNI です。JNI は Java コードとネイティブ・コード (C/C++ など) の間でのやり取りに明確な API を定義することで、この 2 つの実行をはっきりと切り分けます。ほとんどの場合、ネイティブ・コードが JVM のメモリーを直接参照することはないので、ネイティブ・コードを一度作成すれば、JVM の実装やバージョンが違っても作成したネイティブ・コードは確実に機能します。

JNI では、ネイティブ・コードは自由に Java オブジェクトを操作して、フィールド値を取得および設定することができます。メソッドを呼び出すにも、Java コードで同じ関数に適用される数々の制約なしに行えます。しかしこの自由さは両刃の剣です。つまり、上記に挙げたタスクを実行する能力の代わりに、Java 言語の安全性が犠牲になることを意味します。アプリケーション内で JNI を使用すると、マシン·リソース (メモリー、I/Oなど) へのアクセスを下位レベルで強力に行えるようになるため、通常 Java 開発者に提供されるセーフティー・ネットなしで作業することになります。JNI が持つ柔軟性と能力は、プログラミングのプラクティスにリスクをもたらし、それが原因でパフォーマンス不足、バグ、さらにはプログラムの異常終了といった問題まで起こりかねません。アプリケーションの全体的な完全性を守るためには、アプリケーションに組み込むコードに注意を払い、有効なプラクティスに従う必要があります。

この記事では、JNI を使用してコーディングおよび設計を行う際に犯しがちな 10 の間違いを取り上げます。その目的は、皆さんがこれらの間違いを認識して避けることで、最初から有効に機能する安全で効果的な JNI コードを作成できるようになってもらうことです。さらに、これらの問題を新しいコードや既存のコードで検出するために使用できるツールと手法、そしてその効果的な適用方法を説明します。

JNI プログラミングの落し穴は、以下の 2 つのカテゴリーに分けられます。

- **パフォーマンス**: コードは意図される機能を実行するものの、その実行速度が遅かったり、あるいはプログラム全体の処理速度を低下させたりするという落とし穴です。
- **正確性**: コードは往々にして機能しますが、必要な機能を確実に提供しなかったり、最悪の場合は異常終了やハングアップをしたりするという落とし穴です。

パフォーマンスの落とし穴

JNI を使用するプログラマーが陥りやすい上位 5 位までのパフォーマンスの落し穴は以下のとおりです。

メソッド ID、フィールド ID、クラスをキャッシュしない

ネイティブ・コードから Java オブジェクトのフィールドへアクセスしたり、メソッドを呼び出したりするには、`FindClass()`、`GetFieldID()`、`GetMethodId()`、および `GetStaticMethodID()` を呼び出さなければなりません。`GetFieldID()`、`GetMethodID()`、`GetStaticMethodID()` の場合、指定されたクラスに対して返される ID は、JVM プロセスが存続している限り変更されません。しかしフィールドやメソッドを取得するための呼び出しには、JVM 内でかなりの処理が必要になります。フィールドやメソッドがスーパークラスから継承されている場合、それらのフィールドやメソッドを見つけるために JVM がクラス階層を上へウォークスルーすることになるためです。ID は指定されたクラスでは同じなので、ID をいったん検索したら、後はその ID を再利用してください。同様に、クラス・オブジェクトの検索にもコストがかかるため、クラス・オブジェクトもキャッシュする必要があります。

例えばリスト 1 は、静的メソッドを呼び出すために必要な JNI コードです。

リスト 1. JNI での静的メソッドの呼び出し

```
int val=1;
jmethodID method;
jclass cls;

cls = (*env)->FindClass(env, "com/ibm/example/TestClass");
if ((*env)->ExceptionCheck(env)) {
   return ERR_FIND_CLASS_FAILED;
}
method = (*env)->GetStaticMethodID(env, cls, "setInfo", "(I)V");
if ((*env)->ExceptionCheck(env)) {
   return ERR_GET_STATIC_METHOD_FAILED;
}
(*env)->CallStaticVoidMethod(env, cls, method,val);
if ((*env)->ExceptionCheck(env)) {
   return ERR_CALL_STATIC_METHOD_FAILED;
}
```

メソッドを呼び出すたびにクラスとメソッド ID を検索するとなると、ネイティブ・コードが 6 つの呼び出しを行うことになりますが、クラスとメソッド ID が初めて必要になったときにキャッシュしておけば、2 つの呼び出しを行うだけで済みます。

キャッシングはアプリケーションの実行時間に多大な影響を与えます。以下に記載する 2 つのバージョンのメソッドを見てください。どちらも結果的に同じことを行いますが、リスト 2 のバージョンでは、キャッシュされたフィールド ID を使用します。

リスト 2. キャッシュされたフィールド ID を使用する場合

```
int sumValues2(JNIEnv* env, jobject obj, jobject allValues){

   jint avalue = (*env)->GetIntField(env, allValues, a);
   jint bvalue = (*env)->GetIntField(env, allValues, b);
   jint cvalue = (*env)->GetIntField(env, allValues, c);
   jint dvalue = (*env)->GetIntField(env, allValues, d);
   jint evalue = (*env)->GetIntField(env, allValues, e);
   jint fvalue = (*env)->GetIntField(env, allValues, f);

   return avalue + bvalue + cvalue + dvalue + evalue + fvalue;
}
```

パフォーマンスのヒント、その 1

よく使用するクラス、フィールド ID、メソッド ID を検索したら、グローバル・キャッシュに入れること。

リスト 3 のバージョンでは、キャッシュされたフィールド ID を使用していません。

リスト 3. フィールド ID をキャッシュしない場合

```
int sumValues2(JNIEnv* env, jobject obj, jobject allValues){
   jclass cls = (*env)->GetObjectClass(env,allValues);
   jfieldID a = (*env)->GetFieldID(env, cls, "a", "I");
   jfieldID b = (*env)->GetFieldID(env, cls, "b", "I");
   jfieldID c = (*env)->GetFieldID(env, cls, "c", "I");
   jfieldID d = (*env)->GetFieldID(env, cls, "d", "I");
   jfieldID e = (*env)->GetFieldID(env, cls, "e", "I");
   jfieldID f = (*env)->GetFieldID(env, cls, "f", "I");
   jint avalue = (*env)->GetIntField(env, allValues, a);
   jint bvalue = (*env)->GetIntField(env, allValues, b);
   jint cvalue = (*env)->GetIntField(env, allValues, c);
   jint dvalue = (*env)->GetIntField(env, allValues, d);
   jint evalue = (*env)->GetIntField(env, allValues, e);
   jint fvalue = (*env)->GetIntField(env, allValues, f);
   return avalue + bvalue + cvalue + dvalue + evalue + fvalue
}
```

リスト 2 のバージョンは、10,000,000 回実行するのに 3,572 ミリ秒かかります。リスト 3 のバージョンの場合は 86,217 ミリ秒で、これはリスト 2 の 24 倍の所要時間です。

配列のコピーを作成する

JNI は Java コードとネイティブ・コードとの間に明確なインターフェースを提供します。この分離を維持するため、配列は不透明なハンドルとして渡されます。そのため、ネイティブ・コードが配列要素を set 呼び出しや get 呼び出しで操作するためには、JVM を呼び出さなければなりません。Java 仕様では、これらの呼び出しが配列への直接アクセスを可能にするか、あるいは配列のコピーを返すかについては、それぞれの JVM 実装に任せています。例えば、JVM による配列の最適化の方法が、配列を隣接して保管するようになっていない場合には、配列のコピーを返すことになります (このような JVM についての説明は、「参考文献」を参照してください)。

これらの呼び出しによって、操作対象の要素がコピーされる場合もあります。例えば 1,000 の要素からなる配列で `GetLongArrayElements()` を呼び出すと、少なくとも 8,000 バイトが割り振られてコピーされることになります (それぞれ 8 バイトの `long` の要素が 1,000 個)。続いてその配列の中身を `ReleaseLongArrayElements()` で更新する場合には、配列を更新するために、さらに 8,000 バイトのコピーが必要になります。新しい `GetPrimitiveArrayCritical()` を使用するとしても、Java 仕様では、JVM が配列全体をコピーすることを許可しています。

パフォーマンスのヒント、その 2

ネイティブ・コードに必要な配列の部分だけを取得して更新すること。配列の一部だけが必要な場合には、配列全体をコピーしないようにする適切な API 呼び出しを使用してください。

`GetTypeArrayRegion()` メソッドと `SetTypeArrayRegion()` メソッドを使えば、配列全体ではなく、配列の一部を取得して更新することができます。大規模な配列にはこの 2 つのメソッドを使用してアクセスすることで、ネイティブ・コードが実際に使用する配列の部分だけをコピーできるようになります。

例として、リスト 4 に記載する同じメソッドの 2 つのバージョンを見てください。

リスト 4. 同じメソッドの 2 つのバージョン

```
jlong getElement(JNIEnv* env, jobject obj, jlongArray arr_j,
                 int element){
   jboolean isCopy;
   jlong result;
   jlong* buffer_j = (*env)->GetLongArrayElements(env, arr_j, &isCopy);
   result = buffer_j[element];
   (*env)->ReleaseLongArrayElements(env, arr_j, buffer_j, 0);
   return result;
}

jlong getElement2(JNIEnv* env, jobject obj, jlongArray arr_j,
                  int element){
     jlong result;
     (*env)->GetLongArrayRegion(env, arr_j, element,1, &result);
     return result;
}
```

最初のバージョンでは配列全体のコピーが 2 つ作成される一方、2 番目のバージョンではコピーはまったく行われません。1,000 バイトの配列で最初のメソッドを 10,000,000 回実行すると 12,055 ミリ秒かかりますが、2 番目のバージョンの場合はわずか 1,421 ミリ秒です。つまり、最初のバージョンでの処理時間は 8.5 倍も長いことになります。

パフォーマンスのヒント、その 3

1 回の API 呼び出しで、配列のできるだけ多くの部分を取得または更新すること。配列の要素を大きなブロックで取得、更新できる場合には、配列の要素を 1 つずつ繰り返し処理することは禁物です。

その一方、結局は配列の要素すべてを取得することになるのであれば、`GetTypeArrayRegion()` によって配列の要素を 1 つずつ取得するという方法では優れたパフォーマンスを望めません。パフォーマンスを最適化するためには、配列の要素を実用に適う最も大きなブロックで取得、更新してください。配列に含まれる要素のすべてを繰り返し処理するとしたら、リスト 4 の

`getElement()` メソッドのバージョンはいずれも適切ではありません。代わりに、1 回の呼び出しで配列から妥当な大きさの要素のブロックを取得し、これらの要素を繰り返し処理するという操作を、配列全体の処理が完了するまで繰り返します。

パラメーターを渡す代わりに JVM にアクセスする

メソッドを呼び出すときには、複数のフィールドを持つ 1 つのオブジェクトを渡すか、それともフィールドを個々に渡すかを選択できる場合がよくあります。オブジェクト指向の設計では、オブジェクトを渡したほうがカプセル化という点で有益です。なぜなら、オブジェクトのフィールドが変更されても、メソッド・シグニチャーを変更する必要がないからです。しかし JNI の場合、必要な個々のフィールドそれぞれの値を取得するためには、ネイティブ・コードが 1 つ以上の JNI 呼び出しを行って JVM にアクセスしなければなりません。ネイティブ・コードから Java コードへの遷移には、通常のメソッド呼び出し以上にコストがかかるため、これらの追加の呼び出しは、余分なオーバーヘッドをもたらします。したがって JNI では、渡されたオブジェクトに数多く含まれる個々のフィールドについてネイティブ・コードが JVM にアクセスするとなると、パフォーマンスが劣化することになります。

リスト 5 に記載する 2 つのメソッドを見てください。2 番目のバージョンでは、フィールド ID がキャッシュされていることを前提としています。

リスト 5. メソッドの 2 つのバージョン

```
int sumValues(JNIEnv* env, jobject obj, jint a, jint b,jint c, jint d, jint e, jint f){
   return a + b + c + d + e + f;
}

int sumValues2(JNIEnv* env, jobject obj, jobject allValues){

   jint avalue = (*env)->GetIntField(env, allValues, a);
   jint bvalue = (*env)->GetIntField(env, allValues, b);
   jint cvalue = (*env)->GetIntField(env, allValues, c);
   jint dvalue = (*env)->GetIntField(env, allValues, d);
   jint evalue = (*env)->GetIntField(env, allValues, e);
   jint fvalue = (*env)->GetIntField(env, allValues, f);

   return avalue + bvalue + cvalue + dvalue + evalue + fvalue;
}
```

パフォーマンスのヒント、その 4

可能な場合は個々のパラメーターを JNI ネイティブ・コードに渡し、処理を行うためにデータが必要となったときに、ネイティブ・コードが JVM を呼び出すようにすること。

`sumValues2()` メソッドの場合、JNI の呼び出しが 6 回必要となります。このメソッドを 10,000,000 回実行するには 3,572 ミリ秒かかります。処理時間がわずか 596 ミリ秒の `sumValues()` に比べ、これは 6 倍の処理時間です。`sumValues()` は JNI メソッドに必要なデータを渡すことで、JNI によるかなりのオーバーヘッドを回避しています。

ネイティブ・コードと Java コードの境界を誤って選択する

ネイティブ・コードと Java コードとを分け隔てる境界の定義は開発者に任されていますが、境界の選択は、アプリケーション全体のパフォーマンスを大きく左右します。Java コードからネイ

ティブ・コードを呼び出すコスト、そしてネイティブ・コードから Java コードを呼び出すコストは、通常の Java メソッド呼び出しのコストを遙かに上回るからです。さらに Java コードとネイティブ・コードとの間の遷移は、JVM がコードの実行を最適化する際の障害となる可能性もあります。例えば、Java コードとネイティブ・コードとの間で遷移する回数が多くなれば、Just-In-Time コンパイラーの効率は低下します。測定結果によると、Java コードからネイティブ・コードを呼び出すには、通常のメソッドを呼び出す場合に比べ、5 倍の時間がかかります。同様に、ネイティブ・コードから Java コードを呼び出すにも、かなりの時間を要します。

パフォーマンスのヒント、その 5

Java コードからネイティブ・コードへ遷移する回数と、ネイティブ・コードから Java コードを呼び出す回数が最小限になるように、Java とネイティブとの分離を定義すること。

したがって、Java コードとネイティブ・コードとの分離は、この 2 つのコード間の遷移が最小限になるように設計しなければなりません。遷移するのは必要な場合のみに限り、遷移に係るコストを償却するだけの十分な処理をネイティブ・コードで行う必要があります。遷移を最小限にする上で重要な点は、データを境界の Java 側またはネイティブ側のどちらか正しいほうに保持することです。データが誤った側にあると、もう一方の側がそのデータにアクセスするために、ひっきりなしに遷移が行われることになります。

例えば JNI を使用してシリアル・ポートへのインターフェースを提供する場合には、2 つの異なるインターフェースが考えられます。その 1 つは、リスト 6 に記載するインターフェースです。

リスト 6. シリアル・ポートとのインターフェース: バージョン 1

```
/**
 * Initializes the serial port and returns a java SerialPortConfig objects
 * that contains the hardware address for the serial port, and holds
 * information needed by the serial port such as the next buffer
 * to write data into
 *
 * @param env JNI env that can be used by the method
 * @param comPortName the name of the serial port
 * @returns SerialPortConfig object to be passed ot setSerialPortBit
 *          and getSerialPortBit calls
 */
jobject initializeSerialPort(JNIEnv* env, jobject obj,  jstring comPortName);

/**
 * Sets a single bit in an 8 bit byte to be sent by the serial port
 *
 * @param env JNI env that can be used by the method
 * @param serialPortConfig object returned by initializeSerialPort
 * @param whichBit value from 1-8 indicating which bit to set
 * @param bitValue 0th bit contains bit value to be set
 */
void setSerialPortBit(JNIEnv* env, jobject obj, jobject serialPortConfig,
  jint whichBit,  jint bitValue);

/**
 * Gets a single bit in an 8 bit byte read from the serial port
 *
 * @param env JNI env that can be used by the method
 * @param serialPortConfig object returned by initializeSerialPort
 * @param whichBit value from 1-8 indicating which bit to read
 * @returns the bit read in the 0th bit of the jint
 */
```

```
jint getSerialPortBit(JNIEnv* env, jobject obj, jobject serialPortConfig,
  jint whichBit);

/**
 * Read the next byte from the serial port
 *
 * @param env JNI env that can be used by the method
 */
void readNextByte(JNIEnv* env, jobject obj);

/**
 * Send the next byte
 *
 * @param env JNI env that can be used by the method
 */
void sendNextByte(JNIEnv* env, jobject obj);
```

リスト 6 では、シリアル・ポートの構成データはすべて、`initializeSerialPort()` メソッドによって返される Java オブジェクトに保管されます。つまり、ハードウェアのあらゆる部分の設定は、Java コードが完全に制御するということです。リスト 6 のバージョンにはいくつかの問題があり、そのために、リスト 7 のバージョンよりもパフォーマンスに劣ります。

リスト 7. シリアル・ポートとのインターフェース: バージョン 2

```
/**
 * Initializes the serial port and returns an opaque handle to a native
 * structure that contains the hardware address for the serial port
 * and holds information needed by the serial port such as
 * the next buffer to write data into
 *
 * @param env JNI env that can be used by the method
 * @param comPortName the name of the serial port
 * @returns opaque handle to be passed to setSerialPortByte and
 *          getSerialPortByte calls
 */
jlong initializeSerialPort2(JNIEnv* env, jobject obj, jstring comPortName);

/**
 * sends a byte on the serial port
 *
 * @param env JNI env that can be used by the method
 * @param serialPortConfig opaque handle for the serial port
 * @param byte the byte to be sent
 */
void sendSerialPortByte(JNIEnv* env, jobject obj, jlong serialPortConfig,
    jbyte byte);

/**
 * Reads the next byte from the serial port
 *
 * @param env JNI env that can be used by the method
 * @param serialPortConfig opaque handle for the serial port
 * @returns the byte read from the serial port
 */
jbyte readSerialPortByte(JNIEnv* env, jobject obj,  jlong serialPortConfig);
```

パフォーマンスのヒント、その 6

アプリケーションのデータが Java コードとネイティブ・コードの境界の正しい側に置かれ、そのデータを使用するコードがその境界を何度も行き来することなくアクセスできるように、アプリケーションのデータを構造化すること。

リスト 6 のインターフェースで最も明らかな問題は、設定または取得したビットごとに JNI 呼び出しが必要になるだけでなく、シリアル・ポートに対してバイトの読み取り/書き込みを行うにしても、JNI 呼び出しが必要になることです。そのため、バイトの読み取り/書き込みごとに 9 倍の JNI 呼び出しが行われるという結果になります。2 つ目の問題は、リスト 6 ではシリアル・ポートの構成情報を Java オブジェクトに保管していますが、Java コードとネイティブ・コードとの境界で言うと、構成情報が使用されるのは反対側であるという点です。この構成情報が必要となるのはネイティブ側のみです。Java 側に構成情報を保管すると、ネイティブ・コードがこの構成情報を設定/取得するには、Java コードの呼び出しを多数行わなければなりません。リスト 7 では、構成情報をネイティブ・コードの構造 (例えば、C の `struct`) の中に保管し、不透明なハンドルを Java コードに返すことで、以降の呼び出しでこの構成情報を返せるようにしています。つまり、ネイティブ・コードはその実行中に、Java コードを呼び出さなくても、構造内で直接、シリアル・ポートのハードウェア・アドレスや使用可能な次のバッファーなどの情報にアクセスできるということです。したがって、リスト 7 のインターフェースを使用した実装のほうが、パフォーマンスという点で遙かに優れています。

JVM に通知することなく多数のローカル参照を使用する

JNI 関数によって返されたすべてのオブジェクトには、ローカル参照が作成されます。例えば `GetObjectArrayElement()` を呼び出すと、配列内の該当するオブジェクトへのローカル参照が返されます。ここで、かなりの大きさの配列でリスト 8 のコードを実行した場合、どれだけの数のローカル参照が使用されるかを考えてみてください。

リスト 8. ローカル参照の作成

```
void workOnArray(JNIEnv* env, jobject obj, jarray array){
   jint i;
   jint count = (*env)->GetArrayLength(env, array);
   for (i=0; i < count; i++) {
      jobject element = (*env)->GetObjectArrayElement(env, array, i);
      if((*env)->ExceptionOccurred(env)) {
         break;
      }

      /* do something with array element */
   }
}
```

`GetObjectArrayElement()` が呼び出されるたびに、該当する要素に対してローカル参照が作成されます。このローカル参照は、ネイティブ・コードが完了するまで解放されません。そのため配列のサイズが大きいほど、作成されるローカル参照の数も多くなります。

パフォーマンスのヒント、その 7

ネイティブ・コードによって多数のローカル参照が作成される場合には、それぞれの参照が必要なくなった時点で、その参照を削除すること。

作成されたこれらのローカル参照は、ネイティブ・メソッドが終了すると自動的に解放されます。JNI 仕様で要件としている各ネイティブ・コードが作成可能なローカル参照の数は最低でも 16 です。大抵のメソッドにはこの数で十分ですが、メソッドが存続している間にこれより多くのローカル参照にアクセスしなければならないメソッドもなかにはあります。その場合には、必要なくなった参照を削除するか (JNI の `DeleteLocalRef()` 呼び出しを使用)、または JVM に対し、使用するローカル参照の数が増えることを通知しなければなりません。

リスト 9 では、リスト 8 の例に `DeleteLocalRef()` の呼び出しを追加しています。この呼び出しによって、JVM にローカル参照が不要になったことを通知し、配列のサイズとは関係なく同時に存在するローカル参照の数を妥当な数に制限しています。

リスト 9. `DeleteLocalRef()` の追加

```
void workOnArray(JNIEnv* env, jobject obj, jarray array){
   jint i;
   jint count = (*env)->GetArrayLength(env, array);
   for (i=0; i < count; i++) {
      jobject element = (*env)->GetObjectArrayElement(env, array, i);
      if((*env)->ExceptionOccurred(env)) {
         break;
      }

      /* do something with array element */

      (*env)->DeleteLocalRef(env, element);
   }
}
```

パフォーマンスのヒント、その 8

ネイティブ・コードに同時に存在するローカル参照の数が多い場合には、JNI の `EnsureLocalCapacity()` メソッドを呼び出して JVM に通知し、JVM がローカル参照の処理を最適化できるようにすること。

16 を超えるローカル参照を使用することを JVM に通知するには、JNI の `EnsureLocalCapacity()` メソッドを呼び出します。これによって、JVM はネイティブ・コードのローカル参照の処理を最適化することができます。JVM に通知しなければ、必要なローカル参照を作成できないために `FatalError` が発生したり、JVM が採用するローカル参照管理と使用されるローカル参照の数が一致しないためにパフォーマンスが劣化する可能性があります。

正確性の落とし穴

JNI の正確性に関する上位 5 位までの落とし穴は、以下のとおりです。

- 誤った `JNIEnv` を使用する
- 例外をチェックしない
- 戻り値をチェックしない
- 配列メソッドを誤った方法で使用する
- グローバル参照を誤った方法で使用する

誤った `JNIEnv` を使用する

ネイティブ・コードを実行するスレッドは、`JNIEnv` を使用して JNI メソッド呼び出しを行います。しかし、`JNIEnv` は要求されたメソッドをディスパッチするためだけに使われるわけではありません。JNI 仕様では、それぞれの `JNIEnv` は対象スレッドに専用のものであると規定しています。JVM はこの前提を頼りに、`JNIEnv` 内にスレッド固有の追加情報を保管します。そのため、あるスレッドの `JNIEnv` を別のスレッドが使用することによって、デバッグするのが難しい、微妙なバグやプログラムの異常終了が生じる可能性があります。

正確性のヒント、その 1

JNIEnv は、それが関連付けられた 1 つのスレッドだけで使用すること。

スレッドが JNIEnv を取得するには、JavaVM オブジェクトを介した JNI 呼び出しインターフェースによって GetEnv() を呼び出します。JavaVM オブジェクト自体は、JNIEnv オブジェクトを使用して JNI の GetJavaVM() メソッドを呼び出すことで取得することができます。このオブジェクトは、キャッシュすることも、スレッド間で共有することもできます。JavaVM オブジェクトのコピーをキャッシュすれば、スレッドが JNIEnv にアクセスする必要があるときに、キャッシュされたオブジェクトにアクセスすることによって、その固有の JNIEnv にアクセスすることができます。ただしその JNIEnv を検索するには大量の処理が必要になるので、最適なパフォーマンスを実現するためには、スレッドがメソッドを呼び出すときに、そのスレッドの開始時に受け取った JNIEnv をメソッドに渡すようにしてください。

例外をチェックしない

ネイティブ・コードが呼び出すことのできる JNI メソッドの多くは、実行中のスレッドで例外を発生させる可能性があります。Java コードの実行時に例外が発生すると、実行フローが変更されて例外処理のコード・パスが自動的に呼び出されることになります。ネイティブ・コードが JNI メソッドを呼び出すときには例外が発生する可能性があるものの、例外の有無をチェックして適切なアクションを実行するかどうかは、ネイティブ・コードに任されています。そのため JNI プログラミングでは、JNI メソッドを呼び出し、その呼び出しが完了したら例外をチェックせずにそのまま続行するという落とし穴に陥りがちです。この落とし穴は、バグだらけのコードやプログラムの異常終了といった事態を引き起こしかねません。

一例として、GetFieldID() を呼び出し、要求されたフィールドが見つからない場合には NoSuchFieldError を生成するコードがあるとします。ネイティブ・コードが例外をチェックしないで続行し、返されたと見なすフィールド ID を使用した場合、プログラムが異常終了する可能性があります。例えばリスト10 のコードを見てください。Java クラスが変更されて charField フィールドが存在しなくなっているとしたら、このコードは NoSuchFieldError をスローする代わりに、異常終了することになるかもしれません。

リスト 10. 例外チェックの失敗

```
jclass objectClass;
jfieldID fieldID;
jchar result = 0;

objectClass = (*env)->GetObjectClass(env, obj);
fieldID = (*env)->GetFieldID(env, objectClass, "charField", "C");
result = (*env)->GetCharField(env, obj, fieldID);
```

正確性のヒント、その 2

例外を発生させる可能性のある JNI 呼び出しを行った後は、必ず例外をチェックすること。

プログラムの異常終了が起きてからデバッグするよりも、前もって例外をチェックするコードを組み込んでおくほうが遙かに簡単です。大抵は、例外が発生したかどうかを単純にチェックし、例外が発生した場合には直ちに Java コードに戻って例外をスローするという方法を使えます。この場合、スローされた例外は標準的な Java 例外処理プロセスを使用して処理、または表示されることになります。リスト 11 に、例外をチェックする例を記載します。

リスト 11. 例外のチェック

```
jclass objectClass;
jfieldID fieldID;
jchar result = 0;

objectClass = (*env)->GetObjectClass(env, obj);
fieldID = (*env)->GetFieldID(env, objectClass, "charField", "C");
if((*env)->ExceptionOccurred(env)) {
   return;
}
result = (*env)->GetCharField(env, obj, fieldID);
```

例外をチェックすることもなく、解決することもなければ、予期せぬ振る舞いに至る可能性があります。以下のコードでは、何が問題であるかを特定できますか？

```
fieldID = (*env)->GetFieldID(env, objectClass, "charField", "C");
if (fieldID == NULL){
   fieldID = (*env)->GetFieldID(env, objectClass,"charField", "D");
}
return (*env)->GetIntField(env, obj, fieldID);
```

問題は、このコードは最初の `GetFieldID()` がフィールド ID を返さない場合には対応しますが、この呼び出しによってセットされる例外をクリアしないという点です。そのため、ネイティブ・コードから戻ると即刻、例外がスローされることになります。

戻り値をチェックしない

多くの JNI メソッドには、呼び出しが成功したかどうかを示す戻り値があります。そこでよく陥りがちな落とし穴は、例外をチェックしないという落とし穴と同じく、コードが戻り値をチェックせずに、呼び出しが成功したという前提で続行することです。大抵の JNI メソッドでは、例外のステータスと戻り値の両方が設定されるため、アプリケーションがこのいずれかをチェックすることで、メソッドが正しく実行されたかどうかを把握できるようになっています。

正確性のヒント、その 3

常に JNI メソッドからの戻り値をチェックして、エラーを処理するためのコード・パスを組み込むこと。

例えば、以下のコードでは何が問題であるかわかりますか？

```
clazz = (*env)->FindClass(env, "com/ibm/j9//HelloWorld");
method = (*env)->GetStaticMethodID(env, clazz, "main",
                   "([Ljava/lang/String;)V");
(*env)->CallStaticVoidMethod(env, clazz, method, NULL);
```

問題は、`HelloWorld` クラスが見つからない場合、あるいは `main()` メソッドが存在しない場合には、ネイティブ・コードが異常終了することです。

配列メソッドを誤った方法で使用する

`GetXXXArrayElements()` メソッドと `ReleaseXXXArrayElements()` メソッドでは、配列要素を要求することができます。同様
に、`GetPrimitiveArrayCritical()`、`ReleasePrimitiveArrayCritical()`、`GetStringCritical()`、および `ReleaseStringCritical()` では、配列要素またはストリング・バイトを要求することに

よって、配列あるいはストリングの直接ポインターを取得する可能性を最大にすることができます。しかし、これらのメソッドにはその使い方に関連した、陥りがちな 2 つの落とし穴があります。その 1 つは、`ReleaseXXX()` メソッドの呼び出しで変更をコミットし忘れることです。`Critical` バージョンを使ったとしても、実際に配列またはストリングの直接ポインターを取得するという保証はありません。JVM のなかには常にコピーを返すものがあるからです。これらの JVM では、`ReleaseXXX()` の呼び出しに `JNI_ABORT` を指定した場合や、`ReleaseXXX()` を呼び出し忘れた場合、配列に加えられた変更はコピーに反映されません。

一例として、以下のコードを検討してみてください。

```
void modifyArrayWithoutRelease(JNIEnv* env, jobject obj, jarray arr1) {
   jboolean isCopy;
   jbyte* buffer = (*env)-> (*env)->GetByteArrayElements(env,arr1,&isCopy);
   if ((*env)->ExceptionCheck(env)) return;

   buffer[0] = 1;
}
```

正確性のヒント、その 4

`GetXXX()` を呼び出すごとに、毎回忘れずにモードを `0`(コピーに反映してメモリーを解放)に設定した `ReleaseXXX()` を呼び出すこと。

配列の直接ポインターを提供する JVM では配列は更新されますが、コピーを返す JVM では更新されません。このことが原因で、コードが一部の JVM では機能し、その他の JVM では機能しないという事態に陥る場合があります。リスト 12 に記載するように、必ず ReleaseXXX() を呼び出すようにしてください。

リスト 12. ReleaseXXX() 呼び出しの組み込み

```
void modifyArrayWithRelease(JNIEnv* env, jobject obj, jarray arr1) {
   jboolean isCopy;
   jbyte* buffer = (*env)-> (*env)->GetByteArrayElements(env,arr1,&isCopy);
   if ((*env)->ExceptionCheck(env)) return;

   buffer[0] = 1;

   (*env)->ReleaseByteArrayElements(env, arr1, buffer, JNI_COMMIT);
   if ((*env)->ExceptionCheck(env)) return;
}
```

もう 1 つの落とし穴は、`GetXXXCritical()` を呼び出してから `ReleaseXXXCritical()` を呼び出すまでに実行するコードに関して、仕様に規定されている制約に従わないことです。ネイティブ・コードがこの 2 つのメソッドの間で JNI を呼び出すこと、またはブロックすることは、いかなる理由でも禁止されます。これらの制約に従わないと、アプリケーション内や JVM 全体で断続的なデッドロックが発生する可能性があります。

例えば、以下のコードは問題ないように見えるかもしれません。

```
void workOnPrimitiveArray(JNIEnv* env, jobject obj, jarray arr1) {
   jboolean isCopy;
   jbyte* buffer = (*env)->GetPrimitiveArrayCritical(env, arr1, &isCopy);
   if ((*env)->ExceptionCheck(env)) return;

   processBufferHelper(buffer);

   (*env)->ReleasePrimitiveArrayCritical(env, arr1, buffer, 0);
   if ((*env)->ExceptionCheck(env)) return;
}
```

正確性のヒント、その 5

GetXXXCritical() を呼び出してから ReleaseXXXCritical() を呼び出すまでは、いかなる理由でもコードが JNI を呼び出したり、ブロックしたりしないようにすること。

一見問題なさそうに見えますが、`processBufferHelper()` が呼び出されたときに実行される可能性のあるコードのすべてが、上述の制約に違反していないことを確認しなければなりません。これらの制約は、コードがネイティブ・コード自体の一部であろうとなかろうと、`GetXXXCritical()` 呼び出しから `ReleaseXXXCritical()` 呼び出しまでに実行されるすべてのコードに適用されます。

グローバル参照を誤った方法で使用する

ネイティブ・コードでは、オブジェクトが不要になるまでガーベッジ・コレクションの対象にならないようにグローバル参照を作成することができます。そこでよくある落とし穴が、作成されたグローバル参照を削除し忘れたり、グローバル参照の現状が完全にわからなくなってしまったりすることです。以下のコードはグローバル参照を作成する一方、そのグローバル参照を削除することも、どこかに保管することもしません。

```
lostGlobalRef(JNIEnv* env, jobject obj, jobject keepObj) {
   jobject gref = (*env)->NewGlobalRef(env, keepObj);
}
```

正確性のヒント、その 6

常にグローバル参照を追跡し、オブジェクトが必要なくなったときには確実にそのグローバル参照が削除されるようにすること。

グローバル参照が作成されると、JVM はガーベッジ・コレクション対象からの除外リストにそのグローバル参照を追加します。ネイティブ·コードから Java コードへ戻ってしまうと、グローバル参照は解放されないだけでなく、アプリケーションがグローバル参照を後で解放するために取得する手段もなくなってしまいます。つまり、オブジェクトは永遠に存続されるということです。グローバル参照を解放しなければ、グローバル参照がオブジェクト自体を存続させておくだけでなく、そのオブジェクトからアクセスできるすべてのオブジェクトも存続させることになってしまいます。場合によっては、重大なメモリー・リークという結果になることも考えられます。

よくある落とし穴を避ける方法

たった今、新しい JNI コード、または他から継承してきた JNI コードを作成し終えたとします。よくある落とし穴に陥っていないことを確実にする方法、あるいは継承したコードに潜んでいる落とし穴を見つける方法を考えてみてください。表 1 には、よくある落とし穴に陥るのを確実に防ぐために使用できる手法を示しています。

表 1. JNI プログラミングの落し穴を特定するためのチェックリスト

	キャッシュしない	配列のコピーを作成する	誤った境界	JVM に頻繁にアクセスする	過剰な数のローカル参照を使用する	誤った JNIEnv を使用する	例外をチェックしない	戻り値をチェックしない	配列を誤って使用する	グローバル参照を誤って使用する
仕様との照らし合わせ						X	X		X	
メソッド・トレース	X	X	X	X			X		X	X
ダンプ										X
`-verbose:jni`					X					
コード・レビュー	X	X	X	X	X	X	X	X	X	X

一般的な落とし穴の多くは、開発サイクルの初期段階で以下の作業を行うことによって見つけることができます。

- 新しいコードを仕様に照らし合わせて検証する
- メソッド・トレースを分析する
- `-verbose:jni` オプションを使用する
- ダンプを生成する
- コードをレビューする

新しいコードを JNI 仕様に照らし合わせて検証する

仕様で規定されている制約事項のリストを保持し、ネイティブ・コードがそのリストに準拠しているかどうかを手作業または自動コード分析によって確認するのは、効果的なプラクティスです。制約に従っていない場合に発生する可能性のある、捕えにくい断続的な障害をデバッグするよりも、仕様との準拠を確実にすることで、必要な作業は大幅に減ります。新しく開発したコード (または馴染みのないコード) で行う仕様適合性チェックでは、少なくとも以下の項目を確認してください。

- `JNIEnv` が、それと関連付けられたスレッドでのみ使用されていること。
- `GetXXXCritical()` の `ReleaseXXXCritical()` セクションで JNI メソッドが呼び出されないこと。
- クリティカル・セクションに入るメソッドの場合、クリティカル・セクションから抜けてクリティカル・セクションを解放した上で戻ること。
- 例外を発生させる可能性のあるすべての JNI 呼び出しの後で、例外の有無がチェックされること。
- 各 JNI メソッド内のすべての `Get` 呼び出しに、対応する `Release` 呼び出しがあること。

IBM の JVM 実装には、自動 JNI チェックを有効にするオプションが含まれています。実行速度が遅くなるという犠牲は伴いますが、コードに対する優れたユニット・テストと組み合わせることで、このオプションは強力なツールとなります。適合性チェックの際、またはネイティブ・コードが原因だと考えられるバグが発生した際に、この JNI チェックを有効にしてアプリケーションを実行したり、ユニット・テストを行ったりすることができます。これによって、上記に挙げた仕様適合性チェックの他、以下の点が確認されます。

- JNI メソッドに渡されるパラメーターが正しい型であること。
- JNI コードが配列の終わり以降を読み取らないこと。
- 有効なポインターだけが JNI メソッドに渡されること。

JNI チェックがレポートする検出結果のすべてがすべて、コード内のエラーというわけではありません。検出結果のなかには、意図されている役目を果たすことを確実にするために、コードの詳細なレビューを提案するという目的のものもあります。

JNI チェック・オプションを有効にするには、以下のコマンド行を使用します。

```
Usage: -Xcheck:jni:[option[,option[,...]]]

        all             check application and system classes
        verbose         trace certain JNI functions and activities
        trace           trace all JNI functions
        nobounds        do not perform bounds checking on strings and arrays
        nonfatal        do not exit when errors are detected
        nowarn          do not display warnings
        noadvice        do not display advice
        novalist        do not check for va_list reuse
        valist          check for va_list reuse
        pedantic        perform more thorough, but slower checks
        help            print this screen
```

標準開発プロセスの一環として IBM JVM の `-Xcheck:jni` オプションを使用すれば、コーディングのエラーを一層簡単に見つけられるようになります。特に、`JNIEnv` を誤ったスレッドで使用するという落とし穴、そしてクリティカル領域を誤って使用するという落とし穴を一掃する上で効果を発揮します。

最近の Sun JVM にも、`-Xcheck:jni` という同じ名前のオプションが用意されています。このオプションによる動作は IBM のバージョンとは異なり、提供される情報も異なりますが、その目的は同じです。つまり、仕様に従っていないコードがあると警告を出し、一般的な JNI の落とし穴の事例を見つけられるようにするとい目的です。

メソッド・トレースを分析する

呼び出されたネイティブ・コードだけでなく、これらのネイティブ・コードが行う JNI の呼び出しも含めたトレースを生成すると、数多くのよくある落とし穴を一掃するのに役立ちます。トレースでは、例えば以下の問題があるかどうかを調べます。

- 多数の `GetFieldID()` および `GetMethodID()` 呼び出し。特に、複数の呼び出しが同じフィールドとメソッドを対象にしている場合、フィールドおよびメソッドがキャッシュされていないことを意味します。
- `GetTypeArrayRegion()` ではなく、`GetTypeArrayElements()` に対する呼び出しのインスタンス。不必要なコピーが行われている可能性があることを意味します。
- Java コードとネイティブ・コードとの頻繁な切り替え。タイムスタンプによって頻繁に切り替えられていることが示されている場合、Java コードとネイティブ・コードとの間の誤った境界が、パフォーマンスの劣化をもたらす可能性があることを示します。
- ネイティブ関数を呼び出すたびに、多数の `GetFieldID()` 呼び出しが行われるというパターン。これは、必要なパラメーターをネイティブ・コードに渡す代わりに、ネイティブ・コー

ドが Java コードから (処理を完了するために) 必要なデータを読み取るようにしていることを意味します。
- 例外をスローする可能性のある JNI メソッドの呼び出しの後に、`ExceptionOccurred()` または `ExceptionCheck()` が呼び出されていないこと。ネイティブ・コードが例外を適切にチェックしていないことを意味します。
- `GetXXX()` メソッドと `ReleaseXXX()` メソッドの呼び出し回数の不一致。解放し忘れていることを意味します。
- `GetXXXCritical()` 呼び出しと `ReleaseXXXCritical()` 呼び出しの間での JNI メソッドの呼び出し。仕様で規定されている制約に従っていないことを意味します。
- `GetXXXCritical()` 呼び出しから `ReleaseXXXCritical()` 呼び出しまでの時間が長いこと。仕様で規定されている、呼び出しをブロックしないという制約が守られていない可能性があることを意味します。
- `NewGlobalRef()` の呼び出し回数と `DeleteGlobalRef()` の呼び出し回数の大幅な違い。グローバル参照が、不要になった時点で解放されていないことを意味します。

一部の JVM 実装では、メソッド・トレースを生成するためのメカニズムを提供しています。また、プロファイラーやコード・カバレッジ・ツールなどの外部ツールを使用してトレースを生成することもできます。

IBM JVM 実装では、トレース情報をさまざまな方法で生成できるようになっています。その 1 つは、`-Xcheck:jni:trace` オプションを使用する方法です。このオプションでは、呼び出されたネイティブ・メソッドだけでなく、これらのネイティブ・メソッドが行う JNI の呼び出しも含めたトレースを生成します。リスト 13 に、トレースの一部を抜粋して示します (一部の行は読みやすいように改行を加えてあります)。

リスト 13. IBM JVM 実装で生成されたメソッド・トレース

```
Call JNI: java/lang/System.getPropertyList()[Ljava/lang/String; {
00177E00   Arguments: void
00177E00   FindClass("java/lang/String")
00177E00   FindClass("com/ibm/oti/util/Util")
00177E00   Call JNI: com/ibm/oti/vm/VM.useNativesImpl()Z {
00177E00     Arguments: void
00177E00     Return: (jboolean)false
00177E00   }
00177E00   Call JNI: java/security/AccessController.initializeInternal()V {
00177E00     Arguments: void
00177E00     FindClass("java/security/AccessController")
00177E00     GetStaticMethodID(java/security/AccessController, "doPrivileged",
             "(Ljava/security/PrivilegedAction;)Ljava/lang/Object;")
00177E00     GetStaticMethodID(java/security/AccessController, "doPrivileged",
             "(Ljava/security/PrivilegedExceptionAction;)Ljava/lang/Object;")
00177E00     GetStaticMethodID(java/security/AccessController, "doPrivileged",
             "(Ljava/security/PrivilegedAction;Ljava/security/AccessControlContext;)
             Ljava/lang/Object;")
00177E00     GetStaticMethodID(java/security/AccessController, "doPrivileged",
             "(Ljava/security/PrivilegedExceptionAction;
             Ljava/security/AccessControlContext;)Ljava/lang/Object;")
00177E00     Return: void
00177E00   }
00177E00   GetStaticMethodID(com/ibm/oti/util/Util, "toString",
             "([BII)Ljava/lang/String;")
00177E00   NewByteArray((jsize)256)
00177E00   NewObjectArray((jsize)118, java/lang/String, (jobject)NULL)
00177E00   SetByteArrayRegion([B@0018F7D0, (jsize)0, (jsize)30, (void*)7FF2E1D4)
00177E00   CallStaticObjectMethod/CallStaticObjectMethodV(com/ibm/oti/util/Util,
```

```
            toString([BII)Ljava/lang/String;, (va_list)0007D758) {
00177E00     Arguments: (jobject)0x0018F7D0, (jint)0, (jint)30
00177E00     Return: (jobject)0x0018F7C8
00177E00   }
00177E00   ExceptionCheck()
```

リスト 13 に記載するトレースの抜粋には、呼び出されたネイティブ・コード (例えば、`AccessController.initializeInternal()V`) に続いて、そのネイティブ・コードが行った JNI の呼び出しが示されています。

`-verbose:jni` オプションを使用する

Sun と IBM の JVM ではどちらも、`-verbose:jni` オプションを用意しています。IBM JVM の場合、このオプションを有効にすると、実行されている JNI の呼び出しに関する情報が提供されます。リスト 14 はその一例です。

リスト 14. IBM JVM の `-verbose:jni` による JNI の呼び出しのリスト

```
<JNI GetStringCritical: buffer=0x100BD010>
<JNI ReleaseStringCritical: buffer=100BD010>
<JNI GetStringChars: buffer=0x03019C88>
<JNI ReleaseStringChars: buffer=03019C88>
<JNI FindClass: java/lang/String>
<JNI FindClass: java/io/WinNTFileSystem>
<JNI GetMethodID: java/io/WinNTFileSystem.<init> ()V>
<JNI GetStaticMethodID: com/ibm/j9/offload/tests/HelloWorld.main ([Ljava/lang/String;)V>
<JNI GetMethodID: java/lang/reflect/Method.getModifiers ()I>
<JNI FindClass: java/lang/String>
```

Sun JVM の `-verbose:jni` オプションを有効にすると、実行されている呼び出しに関する情報は提供されませんが、使用されているネイティブ・コードに関する追加情報が提供されます。リスト 15 はその一例です。

リスト 15. Sun JVM の `-verbose:jni` の使用

```
[Dynamic-linking native method java.util.zip.ZipFile.getMethod ... JNI]
[Dynamic-linking native method java.util.zip.Inflater.initIDs ... JNI]
[Dynamic-linking native method java.util.zip.Inflater.init ... JNI]
[Dynamic-linking native method java.util.zip.Inflater.inflateBytes ... JNI]
[Dynamic-linking native method java.util.zip.ZipFile.read ... JNI]
[Dynamic-linking native method java.lang.Package.getSystemPackage0 ... JNI]
[Dynamic-linking native method java.util.zip.Inflater.reset ... JNI]
```

このオプションを有効にすると、JVM に通知せずに過剰なローカル参照が使用されている場合、JVM が警告を出すことにもなります。例えば、IBM JVM では以下のようなメッセージを生成します。

```
JVMJNCK065W JNI warning in FindClass: Automatically grew local reference frame capacity
from 16 to 48. 17 references are in use.
Use EnsureLocalCapacity or PushLocalFrame to explicitly grow the frame.
```

`-verbose:jni` と `-Xcheck:jni:trace` オプションによって必要な情報は取得しやすくなるものの、取得した情報を手作業で調べるには、かなりの作業を要します。そこで、JVM によって生成されるトレース・ファイルを処理して、注意すべき兆候を見つけることのできるスクリプトやユーティリティーを作成することが賢明な策となります。

ダンプを生成する

実行中の Java プロセスから生成されるダンプには、JVM の状態に関する豊富な情報が記載されます。多くの JVM では、ダンプには、グローバル参照に関する情報が含まれています。例えば最近の Sun JVM には、ダンプ情報に以下の行が含まれるようになっています。

```
JNI global references: 73
```

これを利用すると、解放されるべき時点で解放されないグローバル参照が作成されていないかどうかを評価するために、当該箇所を実行する前後でダンプを生成することができます。

UNIX® 環境でダンプを要求するには、`java` プロセスでコマンド `kill -3` または `kill -QUIT` を実行します。Windows® では、Ctrl+Break を使用します。

IBM JVM の場合、グローバル参照に関する情報を取得するには以下のステップを実行します。

1. コマンド行に `-Xdump:system:events=user` を追加します。これによって、UNIX 系オペレーティング・システムの場合は `kill -3` を、Windows の場合は Ctrl+Break を実行すると、JVM がシステム・ダンプを生成することになります。
2. プログラムの実行中に、後続のダンプを生成します。
3. `jextract -nozip core.XXX output.xml` を実行し、ダンプ情報を人間が読むことのできるフォーマットで output.xml に抽出します。
4. output.xml 内で `JNIGlobalReference` エントリーを探します。このエントリーによって、現在のグローバル参照に関する情報がわかります (リスト 16)。

リスト 16. output.xml 内の `JNIGlobalReference` エントリー

```
<rootobject type="Thread" id="0x10089990" reachability="strong" />
<rootobject type="Thread" id="0x10089fd0" reachability="strong" />
<rootobject type="JNIGlobalReference" id="0x100100c0" reachability="strong" />
<rootobject type="JNIGlobalReference" id="0x10011250" reachability="strong" />
<rootobject type="JNIGlobalReference" id="0x10011840" reachability="strong" />
<rootobject type="JNIGlobalReference" id="0x10011880" reachability="strong" />
<rootobject type="JNIGlobalReference" id="0x10010af8" reachability="strong" />
<rootobject type="JNIGlobalReference" id="0x10010360" reachability="strong" />
<rootobject type="JNIGlobalReference" id="0x10081f48" reachability="strong" />
<rootobject type="StringTable" id="0x10010be0" reachability="weak" />
<rootobject type="StringTable" id="0x10010c70" reachability="weak" />
<rootobject type="StringTable" id="0x10010d00" reachability="weak" />
<rootobject type="StringTable" id="0x10011018" reachability="weak" />
```

後続の Java ダンプでレポートされているエントリー数を調べることで、グローバル参照がリークしているかどうかを判断することができます。

IBM JVM でダンプ・ファイルおよび `jextract` を使用する方法についての詳細は、「参考文献」を参照してください。

コードをレビューする

多くの場合、よくある落とし穴を見つけるのに有効なコード・レビューは、さまざまなレベルで実施することができます。新たにコードを引き継いだ時点で簡単なスキャンを行うことによっ

て、後でデバッグするとなると大幅に時間がかかるような問題を明らかにすることができます。また場合によっては、コードのレビューという手段でしか、コードが戻り値をチェックしないなどの落とし穴を見つけることができません。例えば、以下のコードの問題はコードをレビューすることで簡単に特定できるはずですが、デバッグによって見つけるのは至難の業です。

```
int calledALot(JNIEnv* env, jobject obj, jobject allValues){
   jclass cls = (*env)->GetObjectClass(env,allValues);
   jfieldID a = (*env)->GetFieldID(env, cls, "a", "I");
   jfieldID b = (*env)->GetFieldID(env, cls, "b", "I");
   jfieldID c = (*env)->GetFieldID(env, cls, "c", "I");
   jfieldID d = (*env)->GetFieldID(env, cls, "d", "I");
   jfieldID e = (*env)->GetFieldID(env, cls, "e", "I");
   jfieldID f = (*env)->GetFieldID(env, cls, "f", "I");

}

jclass getObjectClassHelper(jobject object){
   /* use globally cached JNIEnv */
   return cls = (*globalEnvStatic)->GetObjectClass(globalEnvStatic,allValues);
}
```

コードのレビューによって、上記では同じフィールド ID が何度も使用されているにも関わらず、最初のメソッドが適切にフィールド ID をキャッシュしていないこと、そして 2 番目のメソッドが `JNIEnv` を使用すべきスレッド以外のスレッドで `JNIEnv` を使用していることを見分けられるはずです。

まとめ

以上の説明で、JNI プログラミングの落とし穴の上位 10 位までを認識し、既存のコードや新しいコードの中でこれらの落とし穴を見つける上で有効なプラクティスを学習できたはずです。ここで学んだプラクティスを積極的に適用して、作成する JNI コードが適切である可能性と、必要とされるパフォーマンス・レベルをアプリケーションが達成できる可能性を高めてください。

既存のコード資産を効率的に統合する能力は、現在勢いを得ている 2 つの技術、サービス指向アーキテクチャー (SOA) とクラウド・ベースのコンピューティングで成功を収めるには欠かせません。JNI は、SOA またはクラウド・ベースのシステムのビルディング・ブロックとして使用されている Java ベースのプラットフォームに、Java 以外のレガシー・コードとコンポーネントを統合するための重要な技術です。JNI を正しく使用することが、これらのコンポーネントをサービスに対応させるプロセスを加速し、これまでに行った投資から最大限のものを引き出すことを可能にします。

著者について

Michael Dawson

Michael Dawson は、1989年、University of Waterloo でコンピューター・エンジニアリングの学士号を取得した後、Queens University で暗号法を専攻し、1991年に同大学を電気工学の修士号を取得して卒業しました。卒業後は、新興企業から IBM に至るまでのさまざまな企業でセキュリティー・コンサルティングおよび製品の開発に携わった彼は、EDI 通信サービス、クレジット・カード処理、オンライン・オークション、電子請求書、および JVM を含んだ e-commerce アプリケーションの開発、そしてサービスとしての実現を行うチームで指導的役割を果たしています。経験のある技術は C/C++ からJava および Java EE プラットフォーム、そして広範なオペレーティング・システムのコンポーネントに至るまで多岐に渡ります。彼が IBM に入社したのは 2006年です。現在は、J9 JVM チームで Java クラス・ライブラリーおよびコア JVM コンポーネントの実装に取り組んでいます。

Graeme Johnson

Graeme Johnson は、IBM J9 Virtual Machine チームの開発マネージャー兼技術リーダーです。1994年に IBM (以前の Object Technology International) に入社してからは、仮想マシンとデバッガーの開発に携わり、VisualAge for Java および IBM/OTI Smalltalk ランタイムの両方に取り組みました。最近では、Apache Harmony プロジェクト、そして IBM Project Zero の Java/PHP ランタイム・サポートを専門にしています。彼は定期的にカンファレンスで講演を行っています。JavaOne 2006 では Apache Harmony について、EclipseCon 2007 ではマルチプラットフォームの C 言語開発について、International PHP 2006 では PHP ランタイムの調査について講演しました。

Andrew Low

Andrew Low は、大学在学中に OTI (Object Technology International Inc.) でいくつかのチームに協力し、卒業後の 1994年、同社に入社しました。1996年に IBM による OTI 吸収合併の後もそのまま残り、携帯電話や PDA などのごく小規模なシステムから大規模なシステム (IBM zSeries メインフレーム・システム) に至るまでの VM 技術に携わってきました。現在は、IBM Ottawa Lab の J9 VM チームで、技術リーダーとして IBM の Java ランタイムを支えるコア技術の形成に協力しています。J9 VM の開発で重要な役割を果たした彼は、組み込みシステムおよび Java ME 市場のエキスパートとして知られています。最近では、Java ランタイム技術を Web2.0 の世界に導入するための複数の戦略的取り組みに関わっています。