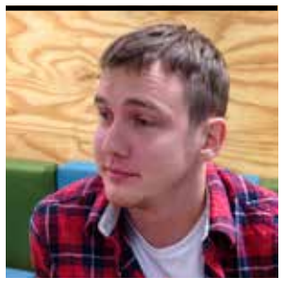

OLEG ŠELAJEV

JVMによるライブラリの検索、ロード、実行

JVMのプログラム実行方法を理解する鍵となるクラス・ローダー

Oleg Šelajev（@shelajev）：ZeroTurnaroundのエンジニア、著者、講演者、講師、開発提唱者。Clojure、Git、MacVimの研究を楽しみつつ、タルトゥ大学でソフトウェアの動的更新とコード進化に関する博士号の取得を目指す。

クラスはJavaの型システムを構成する要素ですが、クラスが果たす基本的な役割はそれだけではありません。クラスはコンパイル・ユニット、つまり個別にロードしてJVMプロセスを実行できる最小のコードです。クラス・ローディングのメカニズムが定められたのは、Javaの創生期であるJDK 1.0の頃にまで遡ります。そしてこのメカニズムが、Javaのクロス・プラットフォームとしての高い人気に大きな影響を及ぼしました。コンパイル後のJavaコード（クラス・ファイルおよびパッケージ化されたJARファイルの形式）は、多数あるサポート対象オペレーティング・システムのいずれであっても、そこで実行中のJVMプロセスにロードできます。だからこそ、コンパイル済みのバイナリ形式ライブラリを開発者が容易に配布できるのです。ソース・コードやプラットフォーム依存のバイナリよりもJARファイルの方がはるかに配布しやすいことがJavaの普及を支えました。特にオープンソース・プロジェクトにおいてその傾向は顕著です。

本記事では、Javaのクラス・ローディングのメカニズムの詳細とその仕組みについて説明します。また、クラスのクラスパス内での検索方法や、メモリへのロード方法、クラスを使用するための初期化方法についても説明します。

JVMへのクラス・ローディングのメカニズム

以下のような単純なJavaプログラムについて考えます。

```
public class A {
 public static void main(String[] args) {
 B b = new B();
 int i = b.inc(0);
 System.out.println(i);
 }
}
```

この一片のコードをコンパイルして実行すると、JVMによってプログラムへのエントリ・ポイントが正しく認識され、クラスAのmainメソッドの実行が開始されます。しかし、JVMは、インポートされているクラス、あるいは参照されているクラスでも、かならずしも先行的に（すぐに）ロードするわけではありません。つまり具体的に言えば、JVMはnew B()文に対応するバイトコード命令に当たったときにはじめてBクラスの検索とロードを試みます。クラスのロード・プロセスを開始する方法には、クラスのコンストラクタ呼び出しの他に、クラスの静的メンバーへのアクセスや、Reflection API経由でのクラスへのアクセスなどもあります。

実際にクラスをロードするために、JVMはクラス・ローダー・オブジェクトを使用します。すでにロードされたすべてのクラスは自身のクラス・ローダーへの参照を持っており、JVMはそのクラス・ローダーを使用して、このクラスから参照されるすべてのクラスをロードします。先ほどの例では、クラスBのロードは、大まかに言って以下のJava文と同等です。A.class.getClassLoader().loadClass("B").

ここには次のようなパラドックスがあります。それは、すべてのクラス・ローダーは、それ自体java.lang.Classloader型のオブジェクトであり、開発者はこのオブジェクトを使用して、クラス名によってクラスを検索してロードできることです。この「卵が先か鶏が先か」の問題に困惑し、すべてのJDKクラス（例：java.lang.String）をロードする最初のクラス・ローダーがどのように作成されるのだろうと疑問に思うかもしれません。もちろん、それはまっとうな考え方です。実は、JVMのコア部分には、プラットフォーム依存のネイティブ・コードで書かれた原始クラス・ローダーが存在しています。このクラス・ローダーはブートストラップ・クラス・ローダーと呼ばれており、java.lang package内のクラスやJavaプリミティブ用のクラスなど、JVM自体が必要とするクラスをロードします。アプリケーション・クラスをロードするのは、Javaで書かれた通常のユーザー定義クラス・ローダーです。つまり、必要であれば、開発者はユーザー定義クラス・ローダーの処理方法を変更できます。

クラス・ローダー階層

JVMのクラス・ローダーはツリー階層内にまとめられており、階層内のすべてのクラス・ローダーに1つの親がいます。クラス・ローダーはクラスの検索とロードを試みる前に、必要なクラスを階層上の親がロードできる（またはすでにロードしている）かどうかチェックします。このチェックを行うことで、クラスを繰り返しロードするという重複を回避できます。一般に、親クラス・ローダーのクラスは子から見えますが、その他のクラス・ローダーからは見えません。クラスの委譲関係と可視性を基本としたこの構造により、階層内のクラス・ローダーの責務を分離し、クラス・ローダーがクラスをロードする起点を特定の場所に限定することが可能です。

JVM のクラス・ローダーはツリー階層内にまとめられており、階層内のすべてのクラス・ローダーに 1 つの親がいます。クラス・ローダーはクラスの検索とロードの前に、必要なクラスを親がロードできる（またはすでにロードしている）かどうかチェックします。

では、Javaアプリケーションのクラス・ローダー階層を確認し、クラス・ローダーが通常ロードするのはどのようなクラスかを見ていきましょう。階層のルートにあるJavaがブートストラップ・クラス・ローダーです。このローダーは、JVM自体の実行に必要となるシステム・クラスをロードします。JDKディストリビューションによって提供されているすべてのクラスがこのクラス・ローダーによってロードされると思ってかまいません（開発者は、ブートストラップ・クラス・ローダーがロードできるクラス・セットの範囲を、-Xbootclasspath JVMオプションを使用して拡大できます）。

ライブラリがブート・クラスパス上に置かれている場合でも、自動的にロードされ初期化されるわけではありません。クラスはあくまで要求に応じてJVMにロードされます。したがって、ブートストラップ・クラス・ローダーからアクセス可能なクラスでも、実際のロードを開始するためには、アプリケーションによってアクセスされる必要があります（ロード・プロセスの興味深い点は、JARファイルをブート・クラスパスに追加しておけば、JDKクラスをオーバーライドできることです。あまり良くない考えではあるものの、より強力なツールを作成できる可能性があります）。

ブートストラップ・クラス・ローダーには、拡張クラス・ローダーと呼ばれる特殊な子がいます。拡張クラス・ローダーは、拡張ディレクトリ（後述）からクラスをロードします。ロードされたクラスは、マシン固有の構成（ロケール、セキュリティ・プロバイダなど）を指定するために使用できます。拡張ディレクトリの場所は、java.ext.dirsシステム・プロパティで指定します。筆者のマシンでは、このプロパティは以下のように設定されています。

```
/Users/shelajev/Library/Java/Extensions:/Library/Java/
JavaVirtualMachines/jdk1.8.0_40.jdk/Contents/Home/jre/
lib/ext:/Library/Java/Extensions:/Network/Library/Java/
Extensions:/System/Library/Java/
Extensions:/usr/lib/java
```

このプロパティの値を変更することで、JVMプロセスに追加でロードするライブラリを変更できます。

この次がシステム・クラス・ローダーです。このローダーは、アプリケーション・クラスおよびクラスパス上にあるクラスをロードします。ユーザーは-cpプロパティでクラスパスを指定できます。

拡張クラス・ローダーとシステム・クラス・ローダーは共にURLClassloader型であり、同じように振る舞います。つまり、最初に親に委譲し、その後必要なら、そこではじめて対象のクラスを検索し、解決します。

Webアプリケーションのクラス・ローダー階層はもう少し複雑です。1つのアプリケーション・サーバーに複数のアプリケーションを同時にデプロイできるため、アプリケーションが自身のクラスを他のアプリケーションのクラスから区別できる仕組みが必要になります。そのため、すべてのWebアプリケーションは独自のクラス・ローダーを使用し、そのクラス・ローダーが、そのWebアプリケーションで使用するライブラリをロードします。このような独立性を確保することで、1つのサーバーに複数のWebアプリケーションがデプロイされても、競合なしに同じライブラリの異なるバージョンを取り扱えるようになっています。アプリケーション・サーバーは必然的に、すべてのWebアプリケーションに対してそれぞれのクラス・ローダーを提供することになります。そして各クラス・ローダーがそのアプリケーションのライブラリをロードします。このような構造が機能するのは、Webアプリケーションのクラス・ローダーが、先に親クラス・ローダーに検索を委譲せず、最初からアプリケーションのWARファイル内にパッケージ化されたクラスの検索を試みるからです。

多くのセキュリティ機能は、権限チェックを行うためにクラス・ローダー階層を利用しています。

正しいクラスの検索

一般に、完全修飾名が同一の複数のクラスにJVMがアクセスできる場合、その競合解消の方法は単純で簡潔です。それは、最初に見つかった該当クラスを優先するというものです。ほとんどのクラス・ローダーが拡張しているURLClassloaderは、クラスパスで指定された順でディレクトリを探索し、該当の名前を持つクラスの中で最初に見つけたものをロードします。

同じことが、名前が同一のJARファイルにも言えます。JARファイルは、名前順ではなく、クラスパス内に出現する順でスキャンされます。必要なクラスのエントリが1つ目のJARファイルで見つかれば、そのクラスがロードされます。見つからなければクラスパスのスキャンが続行し、2つ目のJARファイルに到達します。当然ながら、クラスパス上のいずれの場所にもクラスが見つからない場合は、ClassNotFoundの例外がスローされます。

一般的に、クラスパス内のディレクトリの順序に依存すると不安定になりがちです。そこで代替措置として、-Xbootclasspathにクラスを追加することで、それらのクラスを最初にロードするように指定できます。このアプローチに特に間違った点はありませんが、ブート・クラスパスに変更が加えられると、プロジェクトの維持は大変です。クラスがどこからロードされるかという直観的な認識が崩れ、誰もが混乱することになるでしょう。むしろ、なぜクラスパス上に名前が同一のクラスが複数存在するのかを調べ、曖昧さを根本的に解消する方が得策です。おそらく、何らかの依存先ファイルのバージョンをアップグレードする、キャッシュをクリアする、クリーン・ビルドを実行するなどの対策によって、重複を十分解消できるでしょう。

解決、リンク、検証

クラスを検索し、その初期のメモリ内表現をJVMプロセス内に作成した後は、そのクラスの検証、準備、解決、初期化が行われます。

- **検証**：検証では、クラスが破損していないこと、および構造的に正しいことが確認されます。構造的に正しいとは、実行時コンスタント・プールが妥当であること、変数の型が正しいこと、変数がアクセスされる前に初期化されていることを意味します。-noverifyオプションを指定すれば、検証を無効化できます。JVMプロセスが悪意のあるコードを実行する可能性がない場合は、厳密な検証はおそらく不要です。検証を無効化すると、JVMの起動が速くなります。そして利点がもう1つあります。一部のクラス（特に各種ツールによって実行中に動的に作成されるクラス）がJVMとしては妥当で安全でも、厳密な検証プロセスを通過できないケースです。そうしたツールを使用する場合に、この検証を無効化します。一般に開発環境では、このような対処は許容されます。
- **準備**：クラスの準備には、静的フィールドをそれぞれの型に合うデフォルト値に初期化する処理が含まれます（たとえば、準備後はint型のフィールドには0が格納され、参照はnullになります）。
- **解決**：クラスの解決は、実行時コンスタント・プール内のシンボリック参照が実際に必要とされる型の妥当なクラスを指し示していることをチェックします。シンボリック参照の解決が契機となり、参照先のクラスのロードが行われます。JVM仕様によると、この解決プロセスは遅延方式で実行される場合があり、その場合はクラスの使用時までロードが行われません。
- **初期化**：クラスが準備済み、検証済みであることを前提として、クラスの初期化子（イニシャライザ）を実行します。初期化中、静的フィールドがコード内で指定された値に初期化されます。すべての静的初期化ブロックのコードを結合した静的初期化子（スタティック・イニシャライザ）メソッドも実行されます。初期化プロセスは、ロードされたクラスごとに1回のみ実行すべきものであり、同期的なプロセスです。これは特に、クラスの初期化によって他のクラスの初期化が起動される恐れがあるため、デッドロックに注意して初期化を行う必要があるからです。

JVMによるクラスのロード、リンク、初期化の実行方法について、詳細はJava仮想マシン仕様の第5章を参照してください。

クラス・ローダーに関するその他の考慮事項

このクラス・ローディングのモデルは、Javaプラットフォームが持つ動的な動作の根幹部分です。このモデルがあるからこそ、実行時にクラスを動的に検索してリンクすることが可能になるばかりか、各種ツールがアプリケーションに接続するためのインタフェースも提供されます。

また、多くのセキュリティ機能は、権限チェックを行うためにクラス・ローダー階層を利用しています。たとえば、有名なsun.misc.Unsafe.getUnsafe()メソッドは、ブートストラップ・クラス・ローダーによってロードされたクラスから呼び出された場合に、正常動作としてUnsafeクラスのインスタンスを返します。ブートストラップ・クラス・ローダーからはシステム・クラスのみが返されるため、Unsafe APIを使用するすべてのライブラリはReflection APIを利用してプライベート・フィールドから参照を読み取る必要があります。

まとめ

一般に、ライブラリやフレームワークの開発時は、クラス・ローディングに関する問題について一切憂慮する必要はありません。クラス・ローディングは実行時に発生する動的なプロセスであり、開発者がそこに変更を加えることはほとんどありません。また、典型的なJavaライブラリの場合、クラス・ローディング・スキームを変更する利点もほぼありません。

しかし、互いに分離することを意図したモジュール・システムやプラグイン・システムを作成する場合は、クラス・ローディング・スキームの強化が功を奏することもあるでしょう。カスタムのクラス・ローダーは、基本的にすべてのクラスに影響を及ぼすため、文字どおりアプリケーションのあらゆる箇所に、特定困難なバグを引き込む可能性があることを覚えておいてください。独自のクラス・ローディング機能を設計する場合は、特別な注意が必要です。

本記事では、JVMによるランタイムへのクラス・ローディングの方法、Javaで使用されるクラス・ローダーの階層モデル、および典型的なJavaアプリケーションの階層モデルについて確認しました。

クラス・ローディングの問題に直面していなくても、また日常的にプラグイン・アーキテクチャを構築していなくても、クラス・ローディングを理解することは、アプリケーションの内部動作の理解につながります。また、さまざまなJavaツールについて、その動き方の洞察もできるようになります。そして、クラスパスをクリーンで最新な状態に維持することがいかに良いことかが明白になります。

</article>

Oleg Šelajev（@shelajev）：ZeroTurnaroundのエンジニア、著者、講演者、講師、開発提唱者。Clojure、Git、MacVimの研究を楽しみつつ、タルトゥ大学でソフトウェアの動的更新とコード進化に関する博士号の取得を目指す。

LEARN MORE

- クラス・ローダーの管理に関する情報
- クラス・ローダーのJVM仕様