III. Class Loading

Christoph Denzler / Zoltán Majó

Classloading bezeichnet den Prozess mit dem die JVM den Bytecode einer Klasse sucht, findet und lädt. Klassen werden als Objekte der Klasse `Class` im Hauptspeicher verwaltet. Doch zuvor müssen sie von irgendwo her in die JVM geladen werden. Üblicherweise werden Klassen vom Filesystem her oder von einer URL (z.B. bei Applets) geladen. Mittels selbstgeschriebenen Klassenladern ist es auch möglich verschlüsselte .class-Files zu laden und/oder Klassen von einer Datenbank zu laden.

1 Classloader Hierarchien

Für das Laden von Klassen sind Classloaders zuständig, genauer Instanzen der abstrakten Klasse `java.lang.ClassLoader`. Jede geladene Klasse bleibt mit ihrem Classloader assoziiert. Dieser Classloader kann mit der Methode `getClassLoader()` beim Klassenobjekt angefordert werden. Beispiel: `this.class.getClassLoader()` liefert den Classloader der Klasse des eigenen Objektes zurück.

Zudem ist jeder Thread mit einem Classloader assoziierbar. Dieser Classloader wird als Context-Classloader bezeichnet. Er lässt sich auf jedem Thread mit `getContextClassLoader` bzw. `setContextClassLoader` abfragen und setzen.

Klassenlader sind hierarchisch organisiert. In JavaSE bestimmen drei Classloader die Spitze dieser Hierarchie:

Bootstrap-Classloader: lädt die Klassen der Java Runtime aus `rt.jar`. Der Bootstrap-Classloader ist eine Ausnahme. Er ist keine Instanz von `ClassLoader` sondern integraler Bestandteil der JVM und die Wurzel der Classloader-Hierarchie.

Extension-Classloader: lädt alle Extension-Klassen in den JARs in <JRE>/lib/ext. Dieses Verzeichnis kann eingestellt werden mit dem System Property `java.ext.dirs`. Es ist möglich eigene Klassen in dieses Verzeichnis abzulegen (immer in Form von jar-Files).

System-Classloader: lädt alle durch `-cp`, `-classpath` oder die `java.class.path` System Property adressierten Klassen. Dieser Classloader erhält man mittels `ClassLoader.getSystemClassLoader()`; Der System-Classloader ist der übergeordnete Classloader aller selbstdefinierten Classloader.

Jeder Classloader verwaltet einen Cache mit den Klassen die er schon geladen hat. Ein Classloader darf unter keinen Umständen eine Klasse mehrfach laden.

2 Sichtbarkeit

Jeder Classloader hat einen übergeordneten (parent) Classloader. Klassen die von einem übergeordneten Classloader geladen wurden sind sichtbar für Klassen eines untergeordneten Classloaders – nicht umgekehrt.

Mit Classloadern können auch Namensräume erstellt werden. Abgesehen von der gerade erwähnten Sichtbarkeitsregel, haben Klassen, die von unterschiedlichen Classloadern geladen wurden, keinen Zugriff aufeinander. Insbesondere gilt: wird derselbe Bytecode von zwei verschiedenen Classloadern geladen, so sind dies für die JVM zwei verschiedene Klassen.

2.1 Drei Prinzipien des Classloadings

1. Delegation

2. Sichtbarkeit

3. Einmaligkeit

3 Standard-Suchalgorithmus

Die oben beschrieben Hierarchie von Classloaders bildet die Grundlage für die Suche nach Klassen.

1. Finde die Klasse im Cache des Classloaders

2. Falls nicht gefunden: finde die Klasse mit Parent-Classloader (rekursiv). Ist dieser nicht gesetzt, benutze direkt den Boostrap-Classloader.

3. Falls nicht gefunden: (d.h. alle Parent-Classloaders konnten die Klasse nicht laden, wir müssen die Klasse nun selber finden) Suche die Klasse mit der Methode ClassLoader.findClass(). Werden eigene Classloaders definiert, so muss findClass() überschrieben werden und selbst einen Suchalgorithmus implementieren.

4. Falls nicht gefunden: throw new ClassNotFoundException

Jetzt muss nur noch festgelegt werden, mit welchem Classloader die Suche angefangen wird. Dieser Classloader wird als *initialer Classloader* bezeichnet. In den meisten Fällen wird der initiale Classloader *implizit* von der JVM ausgewählt:

1. In Klasse A wird new-Operator der Klasse B aufgerufen. Die JVM wird den Classloader von A als initialen Classloader verwenden.

2. Klasse B ist abhängig von A (B ist z.B. Superklasse von A). Die JVM wird für das Laden von A den Classloader von B als initialen Classloader verwenden.

3. `Class.forName()` oder `ObjectInputStream.resolveClass()` wird ohne Angabe eines Classloaders aufgerufen. Die JVM sucht im Callstack das erste Objekt, das nicht mit dem Bootstrap-Classloader geladen wurde. Dessen Classloader wird zum initialen Classloader.

Der initiale Classloader kann aber auch explizit ausgewählt werden:

- `ClassLoader.getSystemClassLoader()`

- `Thread.currentThread().getContextClassLoader()`

- `this.getClass().getClassLoader()`

liefern alle einen Classloader auf dem dann loadClass() aufgerufen werden kann.

4 Initialisieren und Finalisieren von Klassen

Ein Classloader ist zuständig für das suchen, finden und laden der Klasse. Er ist *nicht zuständig* für das nachfolgende Linken des Codes und auch *nicht* für die Ausführung der Initialisierung (also Initialisierung von statischen Variablen oder Ausführung von static-Intitializern). Dies sind Aufgaben der JVM. Genau betrachtet ist es für eine JVM zulässig Klassen viel früher zu laden als zu initialisieren, ja es ist auch erlaubt Klassen zu laden und nie zu initialisieren. Dies kann als Optimierung beim Laden von JAR-Files geschehen.

Wer genau wissen will wie Klassen initialisiert werden, oder wer in einem Multithreading -Umfeld Klassen initialisieren will, dem sei das Kapitel 12.4.2 (*Detailed Initialization Procedure*) der Java Language Specification zur Lektüre empfohlen.

Eine Finalisierung einer Klasse ist nicht vorgesehen. Die Sprachdefinition erlaubt es einer JVM-Implementation eine Klasse zu entladen falls *keine* Instanzen davon mehr existieren. Dieses Feature wird aber explizit als eine Optimierung (Speichereffizienz) beschrieben und muss nicht implementiert sein.

Das Entladen von Klassen ist aber an bestimmte Voraussetzungen gebunden, siehe Kapitel 12.7 (*Unloading of Classes and Interfaces*) der JLS. Deshalb erübrigt sich eine Finalisierung von Klassen, die entladen werden können. Das Entladen von Klassen ist nicht explizit spezifiziert noch ist es ein Feature das eine JVM aufweisen muss. Es wird aber als Optimierung gerade in Serverumgebungen standardmässig eingesetzt.

In JEE Umgebungen wird das Unloading aber sehr oft benutzt, nicht zuletzt soll ja der JVM nicht der Speicher ausgehen, wenn mit Hot-Deployment ständig neue Versionen von Applikationen deployed werden.

5 Classloader in JEE Umgebungen

Application Server machen intensiv Gebrauch von Classloadern:

- Trennung von Applikationen (Namespaces)
- Hot Deployment (Nachladen und Entladen von Klassen)
- Aktivierung und Passivierung von Enterprise Java Beans

5.1 JEE Classloader Hierarchie

Die JEE Spezifikation besagt, dass Applikationen durch Classloaders voneinander getrennt werden sollen, sagt aber nicht wie. D.h. jeder Hersteller von Application Server kann seine eigene Strategie wählen und eigene Optimierungen anbringen.

Das Sichtbarkeitsprinzip besagt, dass Child-Classloaders die geladenen Klassen der Parent-Classloaders sehen. Zudem muss weiterhin das Einmaligkeitsprinzip herrschen, denn sonst könnte z.B. eine Singleton-Klasse mehrfach initialisiert werden. Das Delegationsprinzip wird für die WAR-Archive allerdings etwas lockerer interpretiert werden. WAR-, EJB-, RAR- etc. –Applikationen sind als Module zu betrachten, die sich gegenseitig keinen Zugriff gewähren.

JEE Classloader-Hierarchie

Beispiel-Applikation:

Enterprise Applikationen werden in EAR-Files deployed. Jedes EAR-File erhält für sich einen Classloader, innerhalb eines EAR-Files gelten aber etwas spezielle Regeln: EJB-Jar-Files teilen sich alle denselben Classloader, sie „sehen“ sich also gegenseitig. Jedes WAR-File hingegen kriegt seinen eigenen Classloader (gemäss Servlet-Spezifikation). Wie schon angedeutet, wird bei den WAR-Files ein modifizierter Suchalgorithmus angewendet: es wird nämlich zuerst im WAR-File nach Klassen gesucht, bevor an den Parent-Classloader delegiert wird. Dies deshalb, weil die Servlet-Spezifikation eine Isolation der WAR-Files vorschreibt. D.h. alles was im lib-Verzeichnis des WAR-Files liegt muss zwingend auch in der Webapplikation verwendet werden.
Dann sind da noch die sogenannten Dependency-Libraries. Dies sind Jar-Files die sowohl von EJB- als auch von Web-Modulen zugegriffen werden. Z.B. kann eine solches Jar-File Exception Klassen enthalten. Solche Exceptions werden im EJB geworfen und müssen im Servlet abgefangen werden. Oder es sind Klassen die als Argumente übergeben werden.