

JAVA SE – ClassLoaders

Les ClassLoaders ou Pourquoi il faut au moins savoir que ça existe

Où une application Java trouve-t-elle les codes des classes qu'elle utilise ?

En première approximation dans des fichiers .class sur le disque. En fait plutôt dans des archives jar sur le disque (ces archives contiennent les ".class").

Ce sont des codes Java nommés ClassLoader qui ont pour responsabilité de rechercher et de "charger" ces codes de classe.

Quand on lance une application simple il y a trois ClassLoaders qui entrent en action :

- au sommet de la hiérarchie le ClassLoader primordial charge les classes standard de la JVM (il sait où les trouver et comment les charger).
- ensuite vient le ClassLoader des extensions : il charge les bibliothèques déployées dans le répertoire jre/lib/ext. Il s'agit de bibliothèques considérées comme des "extensions standard": c'est à dire des codes qui sont considérés comme suffisamment fiables pour être utilisés par différentes applications.
- enfin le ClassLoader applicatif est en charge des codes et bibliothèques (non standard) utilisés par l'application. C'est ce dernier qui est obtenu par l'appel de `ClassLoader.getSystemClassLoader()`.

Les ClassLoaders ont une organisation hiérarchique. Tous les ClassLoaders (sauf celui qui est au sommet) ont nécessairement un "parent". Quand on demande à un ClassLoader de charger une classe celui-ci va automatiquement le soumettre d'abord à son parent. Il est donc impossible d'écrire une classe qui s'appellerait `java.io.FileInputStream` et d'essayer de la faire charger par une application Java. Le ClassLoader racine va automatiquement recevoir la demande et chargera la classe de la JVM au lieu de votre cheval de Troie !

En réalité il est tout à fait possible d'utiliser d'autres ClassLoaders comme `java.net.URLClassLoader` ou d'écrire et d'utiliser ses propres sous-classes de `ClassLoader`. A quoi peuvent servir ces codes ?

- A rechercher des classes à des endroits particuliers comme une recherche sur le réseau (`URLClassLoader`) ou dans une base de données (code à écrire).
- A permettre de changer de version de codes "à chaud". Un ClassLoader nouveau prend dynamiquement en charge les nouveaux codes. Les anciens peuvent être abandonnés ou conservés dans une partie de l'application qui fonctionne "en parallèle". L'application peut être aussi enrichie avec de nouvelles fonctionnalités sans être arrêtée.
- A lire des pseudo-codes dans un format particulier (crypté par exemple), ou à modifier les pseudo-codes de manière à intercaler des codes qui interceptent ou transforment les appels (programmation orientée aspect).
- Etc.

Ces opérations sont courantes dans divers *frameworks* que ce soient des "serveurs d'application" ou des organisations très dynamiques en réseau (comme Jini/Apache-River).

Chaque `ClassLoader` charge donc **ses** classes et les installe dans un environnement spécifique qui lui est propre.

Du coup des classes chargées par des ClassLoaders différents peuvent interagir selon des modalités complexes que le programmeur a du mal à comprendre. Cela conduit à des bugs "inexplicables" car très difficile à détecter et à comprendre.

Prenons un exemple :

- Soit un `ClassLoader` "parent" (nommé *tronc*) et deux `ClassLoaders` rattachés (nommés *brancheA* et *brancheB*).

- Les deux ClassLoaders en parallèle chargent le même code de classe:

```
public class UneIdentite {  
    public static final UUID U_ID = UUID.randomUUID();  
    // autre codes  
}
```
- Si, dans chaque ClassLoader on a un code qui imprime la constante partagée U_ID on peut voir apparaître **deux** valeurs différentes !

Par exemple :

```
2748f24a-a9d9-4b6b-ad82-19563f1bbf71  
cbbd8d86-0952-4b27-b225-ed16327eaabd
```

On a donc une donnée qui nous semble unique et qui ne l'est pas: une donnée statique n'est unique que dans le contexte de son ClassLoader.

Cette propriété peut être tout à fait souhaitable : si, par exemple, les codes static concernent le chargement d'une ressource, il est normal que chaque ClassLoader opère dans son contexte. Par contre si l'on recherche une unicité au niveau global de l'application il faudra que la donnée soit située dans une classe chargée dans un ClassLoader situé le plus haut possible dans la hiérarchie (et que cette classe ne soit pas chargée par ailleurs) !

Quand une même classe est chargée par différents ClassLoaders (ou quand différentes versions de cette même classe sont gérées dans des ClassLoaders différents) on va, nous l'avons dit, au devant de difficultés épineuses.

Supposons que dans l'exemple ci-dessus le ClassLoader *tronc* charge une classe qui constitue un "dictionnaire de référence" (implanté comme un HashMap).

Les ClassLoaders *brancheA* et *brancheB* connaissent ce code (et accèdent à une variable commune de ce type). Chacun enregistre dans ce dictionnaire des entrées dont les clefs sont des instances de la classe UneIdentite (rappelons que chacun a chargé sa version de cette classe). Le dictionnaire ne va pas fonctionner correctement car il va être incapable de détecter des identités !

Deux instances censées représenter la même chose ne seront pas "égales" au sens de la méthode equals définie dans UneIdentite : des opérations comme instanceof, getClass() ou des transtypages (*cast*) n'opèrent pas correctement si les arguments n'ont pas été chargés par le même ClassLoader !

Dans ce cas il faudra être capable d'écrire une méthode equals capable de traverser les frontières de ClassLoader. On est donc face à un dilemme classique en programmation : faut-il écrire un code lourdement blindé pour résister à toutes les situations possibles ou un code plus léger au risque d'oublier des limites d'utilisation ?

Formations JAVA sur ce sujet

Une démonstration de chargement "à chaud" d'un code par un ClassLoader spécial est effectuée dans nos cours Java de base.

- [USL275 - Java : Programmation pour Développeur Confirmé](#)
- [UJD110 - Java : Programmation pour Développeur Débutant](#)
- La manipulation de ClassLoaders est un thème possible dans nos cours "java avancé" adaptés à la demande.

[Toutes les formations Développement](#)

Bernard AMADE

© Demos