Université Nice Sophia Antipolis

Chargement dynamique

F. Mallet

Adaptation du cours de Ph. Collet et M. Buffa

miage.m1@gmail.com

http://deptinfo.unice.fr/~fmallet/

Définition d'un ClassLoader

- La JVM contient un **`ClassLoader`**
- Les **`ClassLoader`**s permettent de charger des classes depuis le système de fichiers, mais aussi depuis de multiples endroits (BD, réseau, etc… sous forme de tableaux d'octets)
- Rôle :
  - convertir un nom de classe en tableau d'octets représentant la classe

```
Class<?> c = loadClass(String nomClasse, boolean resolveIt);
```

Nom qualifié (ex: java.util.Date)

Edition de lien

Edition de lien

- **Vérification** de la représentation binaire
  - Code opérations valides, saut vers le début d'une instruction, signatures des méthodes
    - **`VerifyError`**
- **Préparation** de la classe
  - Création des champs statiques et initiliazition à leur valeur par défaut => pas d'exécution de code à ce stade (=> Initialisation)
- **Résolution** des références symboliques
  - Chargement des classes référencées et résolution de lien (vers attributs, méthodes, constructeurs)
    - **`IllegalAccessError`** (accès à un champs non autorisé)
    - **`InstantiationError`** (instantiation d'une classe abstraite)
    - **`NoSuchFieldError`**, **`NoSuchMethodError`**, **`UnsatisfiedLinkError`** (méthode native non implémentée)

Généralités sur les ClassLoaders

- Toutes les JVMs ont un ClassLoader (il peut charger certaines classes sans vérifications, JDK…)
- Le CL par défaut implémente une méthode **`loadClass()`** qui cherche dans le CLASSPATH, les fichiers .jar et/ou .zip
- On peut créer de nouveaux CL en dérivant de la classe **`ClassLoader`** et en redéfinissant **`loadClass()`** et/ou les méthodes qu'elle utilise (**`findClass`**,…)
- Utilisation d'un cache
  - Une classe ne peut être chargée qu'**une seule fois** par un CL!
- CL parent (SystemClassLoader par défaut)
  - on demande **d'abord** au parent.

Quand les classes sont-elles chargées ?

- Ca dépend !
- En général quand :
  - Un nouvelle instance est créée
  - Référence statique comme :
    - System.out,
    - String.class,
    - Class.forName("widget.AvecFond");
- Bon à savoir pour améliorer le comportement des gros programmes !
  - Pré-charger les classes !

Ecrire son propre ClassLoader?

- Intérêt ?
  - Charger des classes depuis le WWW,
  - Charger des classes depuis une BD,
  - Charger des classes *« différemment »*
- Attention à la sandbox
  - Ne marche pas avec les applets sauf si on la signe et qu'on modifie le gestionnaire de sécurité!
- JDK1.2 : URLClassLoader (java.net)

Java1: Le ClassLoader

- Sous-classer ClassLoader et implémenter la méthode abstraite loadClass
  - 1) Vérifier le nom de la classe (déjà chargée ?)
  - 2) Vérifier s'il s'agit d'une classe « Système »
  - 3) Essayer de la charger
  - 4) Définir la classe pour la VM
  - 5) La résoudre (charger les dépendances)
  - 6) Renvoyer la classe à l 'appelant
- JDK1.3 et plus
  - sous-classer SecureClassLoader (java.security) si on veut respecter les recommandations relatives à la politique de sécurité de Java

Java 2: findClass

- Depuis Java2
  - Il suffit de redéfinir la méthode

    `Class<?> findClass(String s)`
  - Cette méthode dit à quel endroit se trouve le bytecode (sur une URL, une base de données, créé dynamiquement, …)
  - … et c'est tout !

```
protected Class<?> findClass(String name)
                                    throws ClassNotFoundException {
  byte[] b = loadClassData(name);
  return super.defineClass(name, b, 0, b.length);
}
private byte[] loadClassData(String name)
            throws ClassNotFoundException { /* A DEFINIR */ }
```

Utilisation de son ClassLoader

- `Class<?> c = cl.loadClass("Foo");`
- `Object o = c.newInstance();`
- `((Foo) o).f();` // ?!?

- ne marche pas car seul le nouveau cl connaît **Foo** !
- Deux solutions:
  - Tout programmer dynamiquement
  - Utiliser une interface connue du ClassLoader

Interface Plugin

- Seul le nouveau CL connaît **`Foo`**, si on veut transtyper, il faut définir une interface en commun, qui elle, est connue *aussi* du CL par défaut !
- Par exemple **`Plugin`** connue par le CL par défaut, qui est implémentée par les classes chargées
- Les navigateurs utilisent ce mécanisme pour charger les Applets !

Ecrire des plugins

- Truc simple : lire des plugins dans un répertoire
- Définir une interface d'utilisation pour les plugins.
- Chaque plugin devra implémenter cette interface
- Chaque classe *sera donc un Plugin*
- Il suffit ensuite de
  - 1) Lire le contenu du répertoire
  - 2) Pour chaque nom de classe présent dans le répertoire "plugins"
    - `Class c = Class.forName(nomClasse);`
    - `Plugin p = (Plugin) c.newInstance();`
    - `p.dessine(); // déclarée dans Plugin.java`

Plugins suite...

- Difficultés lorsque
  - Pas de constructeur par défaut
  - Les plugins sont dans un .jar ou une BD
    - Chercher des ressources ? Images, sons, etc…
- Les plugins, il y en a partout
  - Photoshop, Winamp, Firefox, Eclipse, …

Université Nice Sophia Antipolis

Plugins suite...

- Class.forName(nomCLasse)
  - est l'équivalent du loadClass(nomCLasse) étudié dans le cas d'un ClassLoader custom !
- Depuis JDK 1.2, il existe une classe URLClassLoader dans java.net

Plugins : modèle idéal

- Modèle idéal = un répertoire plugins,
- Des fichiers .jar dans le répertoire,
- L'application « découvre » les .jar et « avale » les plugins qu'ils contiennent.
- Chaque .jar contient
  - la classe qui implémente l'interface Plugin.java,
  - les autres classes dont elle a besoin,
  - Les images, icônes, sons, docs, etc… dont le plugin a besoin…

Plugin : modèle idéal

- Mais problème lorsqu'il y a de trop nombreuses classes dans le plugin… comment tester qu'une classe *est réellement* un plugin sans essayer de la charger, ce qui prend du temps…
  - Class.forName() est long…
  - Class.newInstance() aussi…
- Eclipse par exemple, contient des centaines de plugins chargés au démarrage…
- Certains plugins étendent d'autres plugins et donc en dépendent,
- L'ordre de chargement est important…

Comment gérer ses plugins ?

- La solution passe souvent par un descripteur, une « carte » des plugins
  - Eclipse suit une norme pour décrire les plugins, leurs dépendances, etc… Il utilise un descripteur XML
- Le problème No 1 avec les plugins est le temps de chargement…
- Mais les avantages sont nombreux…

Développer un programme extensible par plugins

- Il faut fournir
  - un SDK pour le développeur de plugins
    - Pour qu'il puisse compiler et tester ses plugins sans avoir le code de l'application principale,
  - Classes et Interfaces nécessaires,
  - Un ou plusieurs plugins d'exemples, avec le fichier ant correspondant,
  - De la doc, un tutorial, etc.

Changer les plugins sans relancer le programme principal

- Principe des serveurs de Servlets/Jsps : on ajoute des classes mais on ne relance pas le programme…
- Exemple
  - Serveurs d'application en Java
  - On ne l'arrête jamais…
  - On dépose des plugins et ils sont «découverts» à chaud…

Rechargement à chaud

- Principe : changer le class Loader à chaque rechargement de plugin…
  - Chaque class loader gère un « cache » des classes, si on re-instancie le class loader, on peut charger/recharger des plugins à chaud…
  - Utilisation d'un URLClassLoader
  - Il suffit de re-scanner toutes les 5 secondes par exemple le répertoire qui contient les plugins… ou bien à la demande (clic sur un bouton)….