MJ : A Rational Module System for Java and its Applications

de J. Corwin, D. F. Bacon, D. Grove et C. Murthy
OOPSLA 2003

Introduction

Systèmes grande échelle ⇨ séparation en composants
Spécification des fonctionnalités et hiérarchie

Modularité de Java non adaptée aux systèmes grande échelle
Aspect statique (classes, paquetages) et dynamique (*classloaders*)

Granularité trop fine des classes
Paquetages non hiérarchiques

Stratégie adoptée : contrôle des composants par les *classloaders*
et le `CLASSPATH` ⇨ compliqué et manque de vérification statique

`NoClassDefFoundError`, Erreurs d'exécution ou
boucles infinies entre les *classloaders*

Introduction

Systèmes grande échelle ⇨ séparation en composants
Spécification des fonctionnalités et hiérarchie

Modularité de Java non adaptée aux systèmes grande échelle
Aspect statique (classes, paquetages) et dynamique (*classloaders*)

⮟

Granularité trop fine des classes
Paquetages non hiérarchiques

⮟

Stratégie adoptée : contrôle des composants par les *classloaders*
et le `CLASSPATH` ⇨ compliqué et manque de vérification statique

⮟

`NoClassDefFoundError`, Erreurs d'exécution ou
boucles infinies entre les *classloaders*

Introduction

Systèmes grande échelle ⇨ séparation en composants
Spécification des fonctionnalités et hiérarchie

Modularité de Java non adaptée aux systèmes grande échelle
Aspect statique (classes, paquetages) et dynamique (*classloaders*)

⮟

Granularité trop fine des classes
Paquetages non hiérarchiques

⮟

Stratégie adoptée : contrôle des composants par les *classloaders*
et le `CLASSPATH` ⇨ compliqué et manque de vérification statique

⮟

`NoClassDefFoundError`, Erreurs d'exécution ou
boucles infinies entre les *classloaders*

Plan

Préliminaires

Certaines faiblesses de Java

MJ : un système de gestion de modules

MJ : sa validation

Discussion

Les *classloaders*

Programme Java ≠ un unique programme exécutable
= une composition de fichier *class*

⮟

Ces fichiers *class* sont chargés en mémoire par les *classloaders* en fonction des besoins

Une référence à une classe A dans une classe C est réalisée par un *classloader*

⮟

Problème : le *classloader* ne fait aucune différence entre une référence vers le même module (ou composant) et une référence vers un module différent

Formellement : *linkage* et activation de composants vus de la même façon

Les *classloaders*

Programme Java ≠ un unique programme exécutable
= une composition de fichier *class*

⇓

Ces fichiers *class* sont chargés en mémoire par les *classloaders* en fonction des besoins

Une référence à une classe A dans une classe C est réalisée par un *classloader*

⇓

Problème : le *classloader* ne fait aucune différence entre une référence vers le même module (ou composant) et une référence vers un module différent

Formellement : *linkage* et activation de composants vus de la même façon

Les *classloaders*

Programme Java ≠ un unique programme exécutable
= une composition de fichier *class*

⇓

Ces fichiers *class* sont chargés en mémoire par les *classloaders* en fonction des besoins

Une référence à une classe A dans une classe C est réalisée par un *classloader*

⇓

Problème : le *classloader* ne fait aucune différence entre une référence vers le même module (ou composant) et une référence vers un module différent

Formellement : *linkage* et activation de composants vus de la même façon

Absence de hiérarchie des paquetages
Une unique interface fournie

Visibilité des paquetages

`java.io` et `java.lang` sont totalement indépendants

⇩

`java.io` ne peut récupérer directement les valeurs
des instances de `java.lang.String` ⇨ Perte de performance

Comment faire pour qu'un objet ait une classe d'implémentation A
fournissant une interface abstraite B à un ensemble de ses invocateurs
et une autre interface C plus étendue à un ensemble différent ? La délégation ?

⇩

NON ! La permission est générale

Aucun mécanisme n'existe

Absence de hiérarchie des paquetages
Une unique interface fournie

Visibilité des paquetages
`java.io` et `java.lang` sont totalement indépendants

⮟

`java.io` ne peut récupérer directement les valeurs
des instances de `java.lang.String` ⇨ Perte de performance

Comment faire pour qu'un objet ait une classe d'implémentation A
fournissant une interface abstraite B à un ensemble de ses invocateurs
et une autre interface C plus étendue à un ensemble différent ? La délégation ?

⮟

NON ! La permission est générale

Aucun mécanisme n'existe

Absence de hiérarchie des paquetages
Une unique interface fournie

Visibilité des paquetages
`java.io` et `java.lang` sont totalement indépendants

⮟

`java.io` ne peut récupérer directement les valeurs
des instances de `java.lang.String` ⇨ Perte de performance

Comment faire pour qu'un objet ait une classe d'implémentation A
fournissant une interface abstraite B à un ensemble de ses invocateurs
et une autre interface C plus étendue à un ensemble différent ? La délégation ?

⮟

NON ! La permission est générale

Aucun mécanisme n'existe

Absence de hiérarchie des paquetages
Une unique interface fournie

Visibilité des paquetages
`java.io` et `java.lang` sont totalement indépendants

⮟

`java.io` ne peut récupérer directement les valeurs
des instances de `java.lang.String` ⇨ Perte de performance

Comment faire pour qu'un objet ait une classe d'implémentation A
fournissant une interface abstraite B à un ensemble de ses invocateurs
et une autre interface C plus étendue à un ensemble différent ? La délégation ?

⮟

NON ! La permission est générale

Aucun mécanisme n'existe

Versions multiples des classes

Les applications grande échelle nécessitent souvent différentes versions d'une même classe ou encore d'un même paquetage

Exemple : CORBA et les ORBs
Appels de deux serveurs à partir d'un serveur d'application : 2 ORBs sur le même espace d'adressage de la machine virtuelle

➢

Utilisation de copies de la classe org.omg.CORBA.ORG

Prolifération des *classpaths* et des *classloaders* "personnels" sur les serveurs d'application ⇨ confusion entre les développeurs

➢

Nécessité d'avoir un mécanisme plus simple

Versions multiples des classes

Les applications grande échelle nécessitent souvent différentes versions d'une même classe ou encore d'un même paquetage

Exemple : CORBA et les ORBs
Appels de deux serveurs à partir d'un serveur d'application : 2 ORBs sur le même espace d'adressage de la machine virtuelle

➘

Utilisation de copies de la classe org.omg.CORBA.ORG

Prolifération des *classpaths* et des *classloaders* "personnels" sur les serveurs d'application ⇨ confusion entre les développeurs

➘

Nécessité d'avoir un mécanisme plus simple

Versions multiples des classes

Les applications grande échelle nécessitent souvent différentes versions d'une même classe ou encore d'un même paquetage

Exemple : CORBA et les ORBs
Appels de deux serveurs à partir d'un serveur d'application : 2 ORBs sur le même espace d'adressage de la machine virtuelle

⇩

Utilisation de copies de la classe org.omg.CORBA.ORG

Prolifération des *classpaths* et des *classloaders* “personnels” sur les serveurs d'application ⇨ confusion entre les développeurs

⇩

Nécessité d'avoir un mécanisme plus simple

Objectifs

Création d'un nouveau système de gestion des modules de haut niveau plus adapté en évitant la complexité de la gestion personnelle (développement, déploiement, compréhension, maintenance) du `CLASSPATH` et des *classloaders*

Très peu de changements du code existant
Rester très proche du langage Java

Vérification statique des références

`ClassNotFoundException` et `NoClassDefFoundException`

Objectifs

Création d'un nouveau système de gestion des modules de haut niveau plus adapté en évitant la complexité de la gestion personnelle (développement, déploiement, compréhension, maintenance) du `CLASSPATH` et des *classloaders*

Très peu de changements du code existant
Rester très proche du langage Java

Vérification statique des références

~~`ClassNotFoundException`~~ et ~~`NoClassDefFoundException`~~

Principe général

Création d'un registre de composants contenant :
la description de tous les modules et
les archives des classes fournies par les modules

Données sur les modules

⇨ Emplacement et nature des classes fournies par le module ⇦
`export`, `hide`, `dynamic export` et `dynamic hide`

⇨ Dépendances entre les modules et les classes relatives ⇦
`provides` et `import`

⇨ Visiblité des classes fournies ⇦
`seal`, `unseal`, `forbid` et `unforbid`

⇨ Code d'initialisation ⇦

Principe général

Création d'un registre de composants contenant :
la description de tous les modules et
les archives des classes fournies par les modules

Données sur les modules

⇨ Emplacement et nature des classes fournies par le module ⇦
`export`, `hide`, `dynamic export` et `dynamic hide`

⇨ Dépendances entre les modules et les classes relatives ⇦
`provides` et `import`

⇨ Visiblité des classes fournies ⇦
`seal`, `unseal`, `forbid` et `unforbid`

⇨ Code d'initialisation ⇦

Principe général

Création d'un registre de composants contenant :
la description de tous les modules et
les archives des classes fournies par les modules

Données sur les modules

⇨ Emplacement et nature des classes fournies par le module ⇦
`export`, `hide`, `dynamic export` et `dynamic hide`

⇨ Dépendances entre les modules et les classes relatives ⇦
`provides` et `import`

⇨ Visiblité des classes fournies ⇦
`seal`, `unseal`, `forbid` et `unforbid`

⇨ Code d'initialisation ⇦

Principe général

Création d'un registre de composants contenant :
la description de tous les modules et
les archives des classes fournies par les modules

Données sur les modules

⇨ Emplacement et nature des classes fournies par le module ⇦
`export`, `hide`, `dynamic export` et `dynamic hide`

⇨ Dépendances entre les modules et les classes relatives ⇦
`provides` et `import`

⇨ Visiblité des classes fournies ⇦
`seal`, `unseal`, `forbid` et `unforbid`

⇨ Code d'initialisation ⇦

Principe général

Création d'un registre de composants contenant :
la description de tous les modules et
les archives des classes fournies par les modules

Données sur les modules

⇨ Emplacement et nature des classes fournies par le module ⇦
`export`, `hide`, `dynamic export` et `dynamic hide`

⇨ Dépendances entre les modules et les classes relatives ⇦
`provides` et `import`

⇨ Visiblité des classes fournies ⇦
`seal`, `unseal`, `forbid` et `unforbid`

⇨ Code d'initialisation ⇦

Exemple

```
provides “catalina.jar”;

import * from xerces;
import * from bootstrap;
import com.sun.tools.* from tools;

hide * in *;
export org.apache.catalina.* to webapp;
export org.apache.catalina.servlets to servlets;

forbid org.apache.catalina.* in *;

module catalina
{
   public static void load() {...}
   public static void main(String []args) {...}
}
```

Exemple

```
provides “catalina.jar”;

import * from xerces;
import * from bootstrap;
import com.sun.tools.* from tools;

hide * in *;
export org.apache.catalina.* to webapp;
export org.apache.catalina.servlets to servlets;

forbid org.apache.catalina.* in *;

module catalina
{
   public static void load() {...}
   public static void main(String []args) {...}
}
```

Exemple

```
provides “catalina.jar”;

import * from xerces;
import * from bootstrap;
import com.sun.tools.* from tools;

hide * in *;
export org.apache.catalina.* to webapp;
export org.apache.catalina.servlets to servlets;

forbid org.apache.catalina.* in *;

module catalina
{
   public static void load() {...}
   public static void main(String []args) {...}
}
```

Exemple

```
provides “catalina.jar”;

import * from xerces;
import * from bootstrap;
import com.sun.tools.* from tools;

hide * in *;
export org.apache.catalina.* to webapp;
export org.apache.catalina.servlets to servlets;

forbid org.apache.catalina.* in *;

module catalina
{
   public static void load() {...}
   public static void main(String []args) {...}
}
```

Exemple

```
provides “catalina.jar”;

import * from xerces;
import * from bootstrap;
import com.sun.tools.* from tools;

hide * in *;
export org.apache.catalina.* to webapp;
export org.apache.catalina.servlets to servlets;

forbid org.apache.catalina.* in *;

module catalina
{
   public static void load() {...}
   public static void main(String []args) {...}
}
```

Particularités

JVM : classe nommée par son *classname* et son *classloader*
(objet connu à l'exécution)
MJ : les *classloaders* sont nommés par leur *classname* et leur *modulename*
(existence statique)

En réalité, les modules jouent le rôle des *classloaders* de manière statique

Modules statiques Vs. modules dynamiques
Distinction précise des résolutions dynamiques par le `dynamic export`

Utilitaires de compilation : `modjavac` et `javamodc`

Particularités

JVM : classe nommée par son *classname* et son *classloader*
(objet connu à l'exécution)
MJ : les *classloaders* sont nommés par leur *classname* et leur *modulename*
(existence statique)

En réalité, les modules jouent le rôle des *classloaders* de manière statique

Modules statiques Vs. modules dynamiques
Distinction précise des résolutions dynamiques par le `dynamic export`

Utilitaires de compilation : `modjavac` et `javamodc`

Particularités

JVM : classe nommée par son *classname* et son *classloader*
(objet connu à l'exécution)
MJ : les *classloaders* sont nommés par leur *classname* et leur *modulename*
(existence statique)

En réalité, les modules jouent le rôle des *classloaders* de manière statique

Modules statiques Vs. modules dynamiques
Distinction précise des résolutions dynamiques par le `dynamic export`

Utilitaires de compilation : `modjavac` et `javamodc`

Particularités

JVM : classe nommée par son *classname* et son *classloader*
(objet connu à l'exécution)
MJ : les *classloaders* sont nommés par leur *classname* et leur *modulename*
(existence statique)

En réalité, les modules jouent le rôle des *classloaders* de manière statique

Modules statiques Vs. modules dynamiques
Distinction précise des résolutions dynamiques par le `dynamic export`

Utilitaires de compilation : `modjavac` et `javamodc`

Apache Tomcat 4.1.18

Objectif : remplacer les mécanismes inhérents aux *classloaders* en utilisant le système de gestion des modules MJ

Méthode : diviser Tomcat et les librairies dont il dépend en modules (1 module par fichier .jar et par librairie ⇨ 30 modules)

Détermination des bonnes relations `import/export`
Suppression des utilisations des *classloaders*
Remplacement de ces utilisations par les bons appels à MJ

Résultat : obtention d'un Tomcat identique en changeant uniquement 400 lignes de codes sur environ 167000. A présent, on peut contrôler la visibilité des classes et par exemple améliorer les performances d'entrée-sortie de 30 %.

Apache Tomcat 4.1.18

Objectif : remplacer les mécanismes inhérents aux *classloaders* en utilisant le système de gestion des modules MJ

Méthode : diviser Tomcat et les librairies dont il dépend en modules (1 module par fichier .jar et par librairie ⇨ 30 modules)

Détermination des bonnes relations `import/export`
Suppression des utilisations des *classloaders*
Remplacement de ces utilisations par les bons appels à MJ

Résultat : obtention d'un Tomcat identique en changeant uniquement 400 lignes de codes sur environ 167000. A présent, on peut contrôler la visibilité des classes et par exemple améliorer les performances d'entrée-sortie de 30 %.

Apache Tomcat 4.1.18

Objectif : remplacer les mécanismes inhérents aux *classloaders* en utilisant le système de gestion des modules MJ

Méthode : diviser Tomcat et les librairies dont il dépend en modules (1 module par fichier .jar et par librairie ⇨ 30 modules)

Détermination des bonnes relations `import/export`
Suppression des utilisations des *classloaders*
Remplacement de ces utilisations par les bons appels à MJ

Résultat : obtention d'un Tomcat identique en changeant uniquement 400 lignes de codes sur environ 167000. A présent, on peut contrôler la visibilité des classes et par exemple améliorer les performances d'entrée-sortie de 30 %.

Apache Tomcat 4.1.18

Objectif : remplacer les mécanismes inhérents aux *classloaders* en utilisant le système de gestion des modules MJ

Méthode : diviser Tomcat et les librairies dont il dépend en modules (1 module par fichier .jar et par librairie ⇨ 30 modules)

Détermination des bonnes relations `import/export`
Suppression des utilisations des *classloaders*
Remplacement de ces utilisations par les bons appels à MJ

Résultat : obtention d'un Tomcat identique en changeant uniquement 400 lignes de codes sur environ 167000. A présent, on peut contrôler la visibilité des classes et par exemple améliorer les performances d'entrée-sortie de 30 %.

Discussion — Conclusion

Article bien organisé mais assez mal écrit

Une solution qui apporte des résultats intéressants, des améliorations de performances significatives

Prévisions de tester MJ sur un serveur d'applications commercial : *WebSphere*

Avantages
Réduction de la complexité, vérification statique, robustesse

---

Des questions ?

Discussion — Conclusion

Article bien organisé mais assez mal écrit

Une solution qui apporte des résultats intéressants, des améliorations de performances significatives

Prévisions de tester MJ sur un serveur d'applications commercial : *WebSphere*

Avantages
Réduction de la complexité, vérification statique, robustesse

Des questions ?

Discussion — Conclusion

Article bien organisé mais assez mal écrit

Une solution qui apporte des résultats intéressants, des améliorations de performances significatives

Prévisions de tester MJ sur un serveur d'applications commercial : *WebSphere*

Avantages
Réduction de la complexité, vérification statique, robustesse

Des questions ?

Discussion — Conclusion

Article bien organisé mais assez mal écrit

Une solution qui apporte des résultats intéressants, des améliorations de performances significatives

Prévisions de tester MJ sur un serveur d'applications commercial : *WebSphere*

Avantages
Réduction de la complexité, vérification statique, robustesse

Des questions ?

Discussion — Conclusion

Article bien organisé mais assez mal écrit

Une solution qui apporte des résultats intéressants, des améliorations de performances significatives

Prévisions de tester MJ sur un serveur d'applications commercial : *WebSphere*

Avantages
Réduction de la complexité, vérification statique, robustesse

Des questions ?