Advanced Java

AMT

2 - Advanced Java

Résumé du document

Ce document présente des fonctionnalités avancées de Java couramment utilisées dans les environnements d'applications multi-tiers, comme les chargeurs de classes, la réflexion, les proxys, les annotations au moment de l'exécution et les annotations au moment de la compilation.

Table des matières

1. Chargeurs de classes (Class loaders)

Les chargeurs de classes en Java chargent les classes dynamiquement dans la JVM lors de l'exécution. Les principaux types sont :

- Chargeur de classe Bootstrap : charge les classes du noyau de la JVM.
- Chargeur de classe Platform : gère les classes spécifiques à la plateforme.
- Chargeur de classe système (System) : également appelé chargeur de classe d'application, il charge les classes à partir du classpath.

Les chargeurs de classes fonctionnent en délégant le chargement d'une classe en chaîne, du chargeur d'application jusqu'au chargeur Bootstrap. En cas d'échec dans toute la chaîne, une ClassNotFoundException est levée. On peut créer des chargeurs de classes personnalisés pour des cas spécifiques comme le chargement de classes depuis des sources non standards ou la transformation de classes.

Exemple de création d'un chargeur de classe personnalisé :

```
public class CustomClassLoader extends ClassLoader {
  public CustomClassLoader(ClassLoader parent) {
    super(parent);
  }

  @Override protected Class<?> findClass(String name) throws ClassNotFoundException {
    System.out.println("Chargement avec chargeur personnalisé : " + name);
    return super.findClass(name);
  }
}
```

2. Réflexion (Reflection)

La réflexion est la capacité d'un programme à examiner et à modifier sa structure et son comportement à l'exécution. En Java, elle est souvent utilisée pour :

- Charger dynamiquement des classes
- Créer des objets
- Invoquer des méthodes
- Accéder aux champs

Elle permet également de contourner l'encapsulation, ce qui rend son utilisation puissante mais délicate.

Exemple d'utilisation de la réflexion pour invoquer une méthode :

```
Class<?> clazz = Class.forName("com.example.MyClass");
Object object = clazz.newInstance();
Method method = clazz.getMethod("myMethod");
method.invoke(object);
```

3. Proxys

Un proxy est un objet servant d'interface pour un autre objet. Dans les applications multi-tiers, les proxys sont utilisés pour intercepter des appels de méthode, ajouter des comportements, et implémenter des fonctionnalités comme le chargement différé, la mise en cache, ou les contrôles de sécurité.

Exemple d'implémentation d'un proxy simple :

```
public class MyInvocationHandler implements InvocationHandler {
    private final Object target;
    public MyInvocationHandler(Object target) {
        this.target = target;
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("Avant appel de méthode");
        Object result = method.invoke(target, args);
        System.out.println("Après appel de méthode");
        return result;
    }
}
```

4. Annotations au moment de l'exécution (Runtime annotations)

Les annotations ajoutent des métadonnées aux classes, méthodes et champs Java. Elles sont souvent utilisées dans les applications pour :

- L'injection de dépendances
- La persistance
- La validation
- Les contrôles de sécurité

Les annotations peuvent être lues au moment de l'exécution grâce à la réflexion.

Exemple de traitement d'annotations au moment de l'exécution :

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface MyAnnotation {
    String value();
}

@MyAnnotation("Bonjour le monde!")
class MyClass {}

Class<?> clazz = Class.forName("com.example.MyClass");
MyAnnotation annotation = clazz.getAnnotation(MyAnnotation.class);
System.out.println(annotation.value());
```

5. Annotations au moment de la compilation (Compile-time annotations)

Les annotations au moment de la compilation permettent de générer du code avant l'exécution. Elles peuvent accélérer le démarrage des grandes applications en évitant l'introspection coûteuse de la base de code. Pour les traiter, un processeur d'annotations (annotation processor) est utilisé.

Exemple d'implémentation d'un processeur d'annotations :

```java
@SupportedAnnotationTypes("mypackage.MyAnnotation")
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class MyAnnotationProcessor extends AbstractProcessor {
    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {
        // traitement de l'annotation et génération de code
        return true;
    }
}
```