Sérialisation

Jonathan Lejeune

Sorbonne Université/LIP6-INRIA

SRCS – Master 1 SAR 2019/2020

sources :

Développons en Java, Jean-Michel Doudoux

Introduction

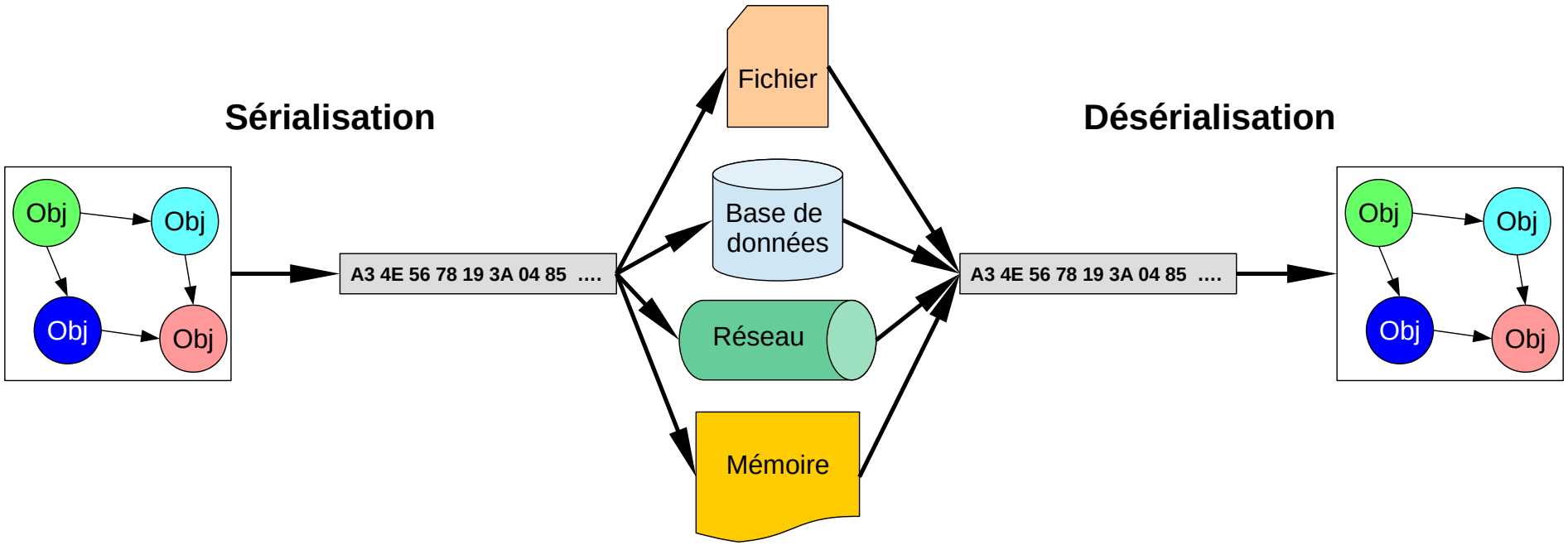
Définitions

- **Sérialisation** : Transformer la valeur d'un objet en un flux d'octets

  synonymes : marshalling, linéariser

- **Désérialisation** : Transformer un flux d'octets pour construire un objet en mémoire

  synonymes : unmarshalling, délinéariser



Introduction

A quoi ça sert ?

- rendre persistants des objets
- stocker des objets momentanément inutilisés
- sauvegarder une configuration
- échanger des objets
- créer une copie intégrale d'un objet (clonage en profondeur)

Comment faire ?

- Manuellement
- Automatiquement via l'environnement d'exécution
  ex : sérialisation JVM
- Via un outil ou une API
  ex : Protobuf, Avro, Parquet, XDR

Formats de sérialisation

Binaire

```
10001101010101101010110101101010101001110001101 0
```

- ✓ Moins volumineux
- ✓ Plus rapide à encoder/décoder
- ✗ Modification directe difficile

Textuel

```
<?xml version="1.0" encoding="UTF-8"?>
  <parent>
    ....
```

- ✓ Portabilité ⇒ utilisation d'une structuration standard (XML, JSON)
- ✓ Modification directe facile
- ✗ Plus volumineux
- ✗ Coûteux en ressources pour être traiter

Problématiques de la sérialisation

- Comment assurer le fait qu'un objet sérialisé soit toujours compatible à sa désérialisation ?
- Comment assurer un format portable ?
- Comment gérer et maintenir les références ?
- Comment gérer les cycles ?
- Comment spécifier les parties du graphe que l'on ne souhaite pas sérialiser ?

La sérialisation binaire Java

Caractéristiques

- Fonctionnalité offerte par la JVM
- Enrichissement de l'API des I/O
- Un format portable pour sauvegarder/charger un objet indépendamment de l'OS

Outils offert au programmeur

- Mot-clé `transient`
- Interfaces : `Serializable`, `Externalizable`
- Classes : `ObjectInputStream`, `ObjectOutputStream`

La sérialisation binaire Java

L'interface `Serializable`

- Interface marqueur du package `java.io`
  $\Rightarrow$ n'offre pas de méthodes
- Mécanisme de sérialisation binaire par défaut
- Interface qui doit être obligatoirement héritée par :
  - soit la classe de l'objet que l'on veut sérialiser
  - soit une de ses classes mères
  - soit une de ses interfaces mères

  sinon `java.io.NotSerializableException`

Le mot clé `transient`

Permet de marquer un attribut pour ne pas le prendre en compte dans le processus de sérialisation.

Sérialiser un graphe d'objet

Principe

Pour pouvoir sérialiser un graphe d'objets, toute référence doit désigner un objet qui est de type `Serializable`

Que faire si la classe d'un attribut n'est pas `Serializable` ?

- Si on est éditeur de la classe
  $\Rightarrow$ la faire étendre de `Serializable`
- Si on n'est pas éditeur de la classe et classe non final
  $\Rightarrow$ créer une classe fille `Serializable` et utiliser cette sous-classe
- Si on souhaite ignorer l'attribut
  $\Rightarrow$ ajouter le mot clé `transient`

Quelques classes sérialisable et non sérialisable de l'API java

Classes non sérialisables

- `Object`
- Classes singletons : `Math`, `System`
- Classes de contexte d'exécution : `Runtime`, `Thread`, `Process`, `ClassLoader`
- Classes de communication : `InputStream`, `OutputStream`, `Socket`

Classe sérialisables

- Classe type de base : `Integer`, `Long`, `Double`, `String`, `Boolean`, ...
- La classes `Throwable` : exceptions, error
- la classe `Class`
- Toute classe d'implémentation concrète de collection
  $\Rightarrow$ les éléments de la collection doivent être aussi Serializable

La classe `ObjectOutputStream`

Caractéristiques

- Décorateur de `OutputStream`
- Offre toute les méthodes de `DataOutputStream`
- Sérialise un objet ou un graphe d'objets :
  - Parcours des objets un à un
  - Tenir compte des références d'objets déjà sérialisés

La méthode `void writeObject(Object o)`

- sérialise le graphe d'objets dont l'objet racine est fourni en paramètre.
- Par défaut le mécanisme de sérialisation écrit pour chaque objet :
  - la classe de l'objet
  - les valeurs de champs
- Par défaut tous les champs d'un objet sont sérialisés sauf :
  - les attributs `static`
  - les attributs `transient`

La classe `ObjectInputStream`

Caractéristiques

- Décorateur de `InputStream`
- Offre toute les méthodes de `DataInputStream`
- Désérialise un objet ou un graphe d'objet à partir d'un flux

La méthode `Object readObject()`

- Renvoie une instance à caster vers le type présumé (faire éventuellement un `instanceof`)
- **La classe présumée doit pouvoir être chargée**
  $\Rightarrow$ `ClassNotFoundException` sinon
- les champs `transient` sont initialisés avec la valeur `null`.
  $\Rightarrow$ cet état doit être géré par l'objet pour éviter les `NullPointerException`
- **Attention : le constructeur de la classe n'est pas invoqué**

La gestion des versions

Objectif

Détecter les différences de versions entre la classe des données déjà sérialisés et la classe de l'instance à sérialiser

Comment faire ?

- Par convention, tout objet `Serializable` contient un attribut constant et statique de numéro de version

```
private static final long serialVersionUID = 1L;
```

- Si pas spécifié ⇒ Warning à la compilation et attribution automatique par la JVM du numéro de version (peut être source de bug)
- Lors de la désérialisation, si le numéro de version des données ne correspond pas au numéro de la classe ⇒ `InvalidClassException`

Compatibilité automatique des versions (1/2)

Principe

La sérialisation par défaut peut gérer automatiquement plusieurs évolutions d'une classe sans empêcher la désérialisation de données d'une version précédente

Précondition : le `serialVersionUID` doit rester identique

Cas d'incompatibilités

- Suppression de champs
- Échanger la hiérarchie de deux classe dans la hiérarchie
- Ajout d'un modificateur `static` ou `transient` à un champs
- Changer le type primitif d'un attribut
- Supprimer ou remplacer `Serializable` par `Externizable` et vice-versa
- Transformer une classe en une énumération et vice-versa

Compatibilité automatique des versions (2/2)

Cas de compatibilité

- Ajouter des champs : un champs non lu est initialisé par défaut
- Changer le modificateur d'accès d'un champ
- Retirer le modificateur `static` ou `transient` d'un champ
- Implanter l'interface `Serializable`
- Ajouter un type dans la hiérarchie des classes mères
- Supprimer un type dans la hiérarchie des classes mères

Sérialisation et héritage

Cas où il existe une classe/interface mère `Serializable`

- Si on veut une classe fille Serializable $\Rightarrow \emptyset$
- Si on ne veut pas que la classe fille soit Serializable
  personnaliser la sérialisation pour cette classe en définissant les méthodes writeObject() et readObject()

Cas où il n'existe pas de classe/interface mère `Serializable`

- Si on veut une classe fille Serializable
  $\Rightarrow$ implanter l'interface `Serializable`
- l'état des classes mère n'est pas pris en compte dans la sérialisation/désérialisation même si les champs sont accessibles dans la classe fille
- les classes mères doivent avoir un constructeur par défaut
- Les champs de la classe mère doivent être gérés manuellement

La sérialisation personnalisée

Quand est-ce utile ?

Le mécanisme par défaut ne peut pas répondre à tous les besoins :

- contrôler plus finement les instances utilisées
- tenir compte des données sensibles
- gérer la sérialisation de différentes versions de la classe
- améliorer les performances notamment avec des versions anciennes de Java

Mécanismes proposés par Java

- définir explicitement la liste des champs à inclure dans la sérialisation
- définir les méthodes `writeObject()` et `readObject()`
- définir les méthodes `writeReplace()` et `readResolve()`
- implémenter l'interface `Externalizable`

Les exeptions liées à la sérialisation

`ObjectStreamException`	Classe mère
`InvalidClassException`	– Incompatibilité dans les versions – Type d'un champs primitif ne correspond pas à la données sérialisées – Implantation de `Externalizable` sans constructeur par défaut – Implantation de `Serializable` mais classe mère non sérialisable n'a pas de constructeur par défaut
`NotSerializableException`	La classe n'est pas sérialisable
`StreamCorruptedException`	Flux corrompu ou invalide
`WriteAbortedException`	Une erreur est survenue durant l'écriture du flux
`ClassNotFoundException`	Impossible de charger la classe lors de la désérialisation

Exemple de Sérialisation

```
package srcs;

import java.io.Serializable;

class Mere implements Serializable{
  private int j=-1;
}

class Fille extends Mere {
  private int i=4;
  private String s="ok";
  private Voisin v = new Voisin(this);
}

class Voisin implements Serializable{
  private Fille voisin = null;
  private float x = 3.2f;
  public Voisin(Fille fille) {  this.voisin=fille;  }
}
```

Exemple de Sérialisation

```
FileOutputStream fos = new FileOutputStream("/tmp/f");
ObjectOutputStream oos = new ObjectOutputStream(fos)
oos.writeObject(new Fille());
```

Contenu du fichier

```
ac ed 00 05 73 72 00 0a 73 72 63 73 2e 46 69 6c 6c 65 8e ec
c4 d3 42 4c ff 91 02 00 03 49 00 01 69 4c 00 01 73 74 00 12
4c 6a 61 76 61 2f 6c 61 6e 67 2f 53 74 72 69 6e 67 3b 4c 00
01 76 74 00 0d 4c 73 72 63 73 2f 56 6f 69 73 69 6e 3b 78 72
00 09 73 72 63 73 2e 4d 65 72 65 2e 7e 31 76 ab 7a 6d 79 02
00 01 49 00 01 6a 78 70 ff ff ff ff 00 00 00 04 74 00 02 6f
6b 73 72 00 0b 73 72 63 73 2e 56 6f 69 73 69 6e ec d3 7d b4
49 cd f5 54 02 00 02 46 00 01 78 4c 00 06 76 6f 69 73 69 6e
74 00 0c 4c 73 72 63 73 2f 46 69 6c 6c 65 3b 78 70 40 4c cc
cd 71 00 7e 00 04
```