Chapitre 3
Rappels sur les éléments externes à Spring

1. Codage equals et hashCode

L'égalité entre objets joue un rôle crucial dans Spring, particulièrement dans les architectures qui utilisent le mapping objet-relationnel (ORM), comme Hibernate et JPA. Dans ces systèmes, les notions de proxy sont fréquentes. Lorsqu'on travaille avec deux objets en mémoire, il est important de faire attention à ne pas confondre un proxy avec l'objet qu'il représente.

Pour éviter ce problème, vous pouvez utiliser `instanceof` au lieu de `getClass()` dans votre méthode `equals(). instanceof` vérifie qu'un objet est une instance d'une classe particulière ou de l'une de ses sous-classes, donc il renverra true même si l'un des objets est un proxy Hibernate.

Voici comment vous pourriez implémenter cela :

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof MyEntity)) return false;
    MyEntity myEntity = (MyEntity) o;
    return Objects.equals(businessKey, myEntity.businessKey);
}
```

Dans cet exemple, nous utilisons `instanceof` pour vérifier que l'objet est une instance de `MyEntity` ou de l'une de ses sous-classes. Cela permettra à `equals()` de fonctionner correctement même si l'un des objets est un proxy Hibernate.

Par ailleurs, du point de vue métier, deux objets peuvent être considérés comme identiques même s'ils existent comme instances distinctes en mémoire. Cette distinction est particulièrement importante lorsqu'on travaille avec des collections.

De plus, nous utilisons parfois des clés techniques qui ne sont renseignées qu'au moment de la persistance.

Les méthodes `equals` et `hashCode` en Java ont une définition spécifique qui détermine comment les objets doivent être comparés et identifiés. Dans le contexte de Spring, Hibernate et JPA, où les proxies et les listes sont largement utilisés, il est nécessaire d'adapter ces méthodes pour assurer un comportement correct. Dans ce chapitre, nous utilisons les termes *equals* et *hashcode* pour faire référence respectivement aux méthodes `equals` et `hashCode`.

Nous explorerons une interprétation de la documentation officielle Java sur `equals` et `hashCode`, en adaptant ces concepts pour une utilisation efficace dans des architectures basées sur les proxies et les collections, comme c'est le cas avec Spring, JPA et Hibernate.

Une traduction approximative de la documentation officielle pour `equals` et `hashCode` est présentée ci-dessous.

equals

```
public boolean equals(Object obj)
```

Indique si un objet est égal à celui passé en paramètre.

La méthode `equals` implémente une relation d'équivalence entre deux références d'objets non nulles.

Elle est réflexive : pour toute référence d'une valeur non nulle `x`, `x.equals(x)` doit retourner `true`.

Elle est symétrique : pour toute référence d'une valeur non nulle `x` et `y`, `x.equals(y)` retourne `true` si et seulement si `y.equals(x)` retourne `true`.

Elle est transitive : pour toute référence d'une valeur non nulle `x`, `y` et `z`, si `x.equals(y)` retourne `true` et si `y.equals(z)` retourne `true` alors `x.equals(z)` doit retourner `true`.

Elle est consistante : pour toute référence d'une valeur non nulle `x` et `y`, de multiples invocations de `x.equals(y)` retournent toujours `true` ou de façon consistante toujours `false`, si aucune information fournie utilisée dans les comparaisons d'égalité sur les objets n'est modifiée.

Pour une référence non nulle sur une valeur `x`, `x.equals(null)` doit retourner `false`.

La méthode `equals` pour la classe `Object` implémente la relation d'équivalence la plus discriminante possible sur les objets ; cela consiste à ce que pour toutes les valeurs de référence non nulles `x` et `y`, cette méthode renvoie `true` si et seulement si `x` et `y` se réfèrent au même objet (`x == y` a la valeur `true`).

Remarque

Notez qu'il est généralement nécessaire de redéfinir la méthode hashCode chaque fois que la méthode equals est surchargée, de manière à maintenir le contrat général pour la méthode hashCode, qui stipule que les objets égaux doivent avoir des codes de hachage égaux.

Paramètres

`obj` : la référence de l'objet avec lequel comparer.

Valeur retournée

`true` si cet objet est le même que celui transmis par l'argument `obj` ; `false` sinon.

Voir aussi

`hashCode()`, `HashMap`.

hashCode

```
public int hashCode()
```

Retourne une valeur de code de hachage pour l'objet.

Cette méthode fournit une prise en charge au bénéfice des tables de hachage telles que celles fournies par `HashMap`.

Le contrat général de `hashCode` est : chaque fois qu'il est appelé sur le même objet plus d'une fois lors d'une exécution d'une application Java, la méthode `hashCode` doit toujours retourner le même entier, tant qu'aucune information utilisée dans les comparaisons sur l'objet n'est modifiée.

Cet entier n'a pas besoin de rester cohérent d'une exécution d'une application à une autre exécution de la même application (sauf à le rendre persistant à un moment donné pour le réutiliser).

Si deux objets sont égaux selon la méthode `equals (Object)`, alors l'appel de la méthode `hashCode` sur chacun des deux objets doit produire le même résultat sous forme d'entier.

Dans le cas où deux objets sont inégaux selon la méthode `equals (java.lang.Object)`, la méthode `hashCode` appelée sur chacun de ces deux objets peut produire des résultats distincts. Toutefois, le programmeur doit être conscient que la production de résultats sous la forme d'entiers distincts pour les objets inégaux peut améliorer la performance des tables de hachage.

Autant que possible, la méthode `hashCode` définie par la classe `Object` ne retourne des entiers distincts que pour des objets distincts. Elle est généralement mise en œuvre par la conversion de l'adresse interne de l'objet en un entier, mais cette technique de mise en œuvre n'est pas requise par le langage de programmation Java.

Valeur retournée

Une valeur de hashcode pour cet objet.

Voir aussi

`equals(java.lang.Object)`,

`System.identityHashCode(java.lang.Object)`.

1.1 Description de la problématique

Par défaut, Java utilise les identifiants mémoire pour les opérations de `equals` et `hashCode`. Bien que cela soit suffisant pour des cas simples, la personnalisation de ces méthodes devient complexe dans des environnements utilisant des proxies, tels que Spring, Hibernate et JPA.

Dans ces systèmes, les clés primaires des entités, identifiées par l'annotation `@Id` (ou de manière composite avec `@IdClass` ou `@EmbeddedId`), ne sont assignées qu'après l'enregistrement dans la base de données, sauf si l'ID est prédéfini. Les collections dans ces environnements dépendent des méthodes `equals` et `hashCode` pour déterminer si un élément est déjà présent.

Il est important de noter que les proxies de Spring modifient le comportement de ces méthodes. Ces méthodes doivent être déléguées à l'objet réel derrière le proxy, mais ne peuvent pas être interceptées par l'AOP de Spring puisqu'elles appartiennent déjà à un objet proxy. Pour intercepter `equals` et `hashCode` avec un proxy, il faudrait utiliser AspectJ.

Concernant les proxies impliqués dans le chargement paresseux (*lazy loading*), il est recommandé d'éviter de comparer les références ou les types d'objets. Il faut plutôt se concentrer sur le contenu des objets, ce qui entraîne un chargement forcé dans le contexte de JPA et Hibernate, ou l'instanciation et l'injection pour Spring.

1.2 Mise en œuvre

La solution la plus judicieuse consiste souvent à utiliser une clé métier basée sur des données concrètes, bien que cela puisse être compliqué à gérer. Habituellement, on utilise une clé primaire séquentielle, distincte des clés métier, qui se caractérise principalement par son unicité. Pour effectuer des comparaisons, on compare les valeurs des attributs, et pour le calcul du `hashCode`, on calcule le `hashCode` de chaque variable et on combine ces valeurs pour obtenir un `hashCode` global. Cependant, la situation se complique lorsque des objets liés sont inclus dans la comparaison. Ces considérations de comparaison et de calcul de `hashCode` se retrouvent souvent dans des parties de code très génériques et répétitives, comme la couche domaine, où elles apportent peu de valeur ajoutée.

Une solution a été proposée sur ce forum :
https://discourse.hibernate.org/

```
@Getter
@Setter
public class ModelElabore implements Serializable {

  private volatile VMID vmid;
  private Long id;
  private String nom;

  public boolean equals(Object obj) {
    final boolean returner;
    if (obj instanceof ModelElabore) {
      return getVmid().equals(((ModelElabore) obj).getVmid());
    } else {
      returner = false;
    }
    return returner;
  }

  public int hashCode() {
    return getVmid().hashCode();
  }

  private Object getVmid() {
    if (vmid != null || vmid == null && id == null) {
```

```
      if (vmid == null) { //Avoid the performance impact
                          //of synchronized if we can
        synchronized (this) {
          if (vmid == null)
            vmid = new VMID();
        }
      }
      return vmid;
    }
    return id;
  }
}
```

En utilisant la classe `VMID` de Java, nous exploitons une API Java conçue pour générer un identifiant unique utilisable par toutes les machines virtuelles Java. Cette API est généralement employée pour la gestion des ramasse-miettes (*garbage collectors*) distribués, afin d'identifier les différentes machines virtuelles clientes.

L'utilisation de `VMID` garantit l'obtention d'un identifiant unique. Une alternative serait d'utiliser l'API org.apache.commons.id.uuid de la librairie Apache Commons. Cependant, si l'API Java standard est disponible, elle est souvent préférée pour cette fonction.

Bien que cette solution puisse sembler excessive, elle s'avère efficace dans la majorité des cas et assure la création d'identifiants uniques fiables.

2. Log4j, SLF4J et Logback

Dans le domaine de la journalisation en Java, Log4j a longtemps été le système standard pour enregistrer des messages d'information, de débogage et d'erreur. Cependant, la bibliothèque Logback (http://logback.qos.ch/) est apparue comme une alternative plus performante que Log4j.

Il est important de noter qu'avec Log4j, des préoccupations de sécurité ont été identifiées, notamment dans les versions antérieures à la 2.17. Ces problèmes de sécurité ont été adressés dans les versions plus récentes de Log4j.

Historiquement, *Jakarta Commons Logging* (JCL) était une des premières API de journalisation, mais elle était limitée dans ses fonctionnalités. Vers 2006, des projets majeurs comme Hibernate ont commencé à utiliser l'API SLF4J (http://www.slf4j.org/), contribuant à sa popularité. SLF4J a apporté des améliorations significatives, mais n'a pas complètement supplanté Log4j.

L'un des avantages de SLF4J est sa capacité à centraliser dans un seul ensemble de fichiers de logs les appels effectués par différentes API de journalisation, telles que JCL, Log4j et JUL (Java Util Logging - http://docs.oracle.com/javase/8/docs/api/java/util/logging/package-summary.html). Cela permet de regrouper les logs de diverses sources et de choisir l'implémentation de journalisation souhaitée.

Contrairement à JCL, qui charge les librairies de journalisation au moment de l'exécution via le classloader, SLF4J opère une sélection de l'implémentation de journalisation à compiler en utilisant des *bridges* ou ponts logiciels pour connecter les différents types d'API. Ce mécanisme permet une intégration plus souple et plus contrôlée des bibliothèques de journalisation.

L'API Logback est aujourd'hui une des meilleures API de journalisation. Elle offre en effet beaucoup de souplesse pour générer des logs. Il est possible également d'associer Logback à SLF4J et à Lombok :

```
package fr.enieditions.ep5jasp.chap02.leslogs;

import lombok.extern.slf4j.Slf4j;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@Slf4j
public class LesLogs {
  private static final Logger slf4jLogger = LoggerFactory
    .getLogger(LesLogs.class);

  public static void main(String[] args) {
    slf4jLogger.trace("Bonjour à vous!");
    String nom = "John";
    slf4jLogger.debug("Hi, {}", nom);
    slf4jLogger.info("Log de type info.");
    slf4jLogger.warn("Log de type warn.");
    slf4jLogger.error("Log de type error.");
```

```
        log.info("Log de type info via lombok, {}", nom);
    }
}
```

Cet ouvrage utilise, dans la mesure du possible, les logs Logback.

Bien que nous ayons dû limiter la portée de ce chapitre, il est important de noter que Logback s'intègre efficacement dans une stack ELK (*Elasticsearch*, *Logstash*, *Kibana*) et dans ses concurrents comme Splunk. Lorsque l'on crée des logs, il est essentiel de garder à l'esprit qu'ils peuvent être d'une grande aide pour les équipes responsables de l'exploitation et de la production des applications dans l'environnement cible. Par conséquent, il est recommandé de journaliser un maximum de données utiles.

Spring Boot fournit une configuration par défaut pour la journalisation (*logging*) qui utilise Logback et SLF4J. Par conséquent, si vous utilisez Spring Boot, vous n'avez généralement pas besoin d'inclure explicitement des dépendances de journalisation dans votre fichier de configuration Maven (pom.xml), sauf si vous souhaitez utiliser une bibliothèque de journalisation différente.

Pour utiliser la bibliothèque Log4j2 plutôt que Logback par défaut, nous devons exclure Logback de nos dépendances de départ :

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions>
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-logging</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

Dans cet extrait, nous modifions la dépendance spring-boot-starter-web pour exclure spécifiquement spring-boot-starter-logging, qui est la dépendance par défaut pour Logback. Cela signifie que Logback ne sera pas inclus dans notre projet, nous permettant d'utiliser Log4j2 à la place.