25
fév.
2008

De la bonne implémentation du Singleton en Java

Java / JEE › Articles | Tags : design patterns, java Par Olivier Croisier

Le Singleton est sans doute le plus connu des design patterns, et souvent le premier cité lors des entretiens techniques.

Pourtant, son implémentation correcte en Java est plus complexe qu'il n'y paraît.

Rappel : le design pattern Singleton

Le Singleton répond à deux exigences :

- garantir qu'une unique instance d'une classe donné sera créée
- offrir un point d'accès universel à cette instance.

Ce design pattern est tout indiqué pour implémenter des services qui :

- sont fonctionnellement uniques au sein de l'application (ex: système de logging centralisé, gestion de la configuration...)
- doivent pouvoir être appelés par toutes les couches de l'application. Il serait en effet peu pratique de passer une référence au service à toutes les classes devant l'utiliser.

Voyons maintenant comment on l'implémente en Java.

Un singleton basique

Un simple enchaînement de réflexions permet de déduire les caractéristiques d'une classe Singleton :

1. Afin de garantir l'unicité du Singleton, il est nécessaire de contrôler strictement son

processus d'instanciation. Il faut donc interdire à tout code extérieur d'utiliser l'opérateur "new" et de créer des instances supplémentaires. Pour cela, il suffit de déclarer un constructeur de visibilité "privé" (voir le Java Quiz #1 (http://thecodersbreakfast.net/index.php?post/2008/02/20/22-java-quiz-1)).
2. Conséquence : pour obtenir une référence sur une instance du Singleton, le code appelant devra obligatoirement passer par une méthode utilitaire au lieu du constructeur. Cette méthode sera nécessairement statique, car à cet instant, le code appelant ne dispose encore d'aucune référence sur l'instance du singleton, et ne peut donc accéder qu'à ses membres statiques.
3. La méthode utilitaire étant statique, elle ne peut accéder qu'aux propriétés également statiques de la classe. L'instance unique devra donc être statique aussi.

Voici donc la version minimale du Singleton :

```
1 /**
2  * Implémentation simple d'un singleton.
3  * L'instance est créée à l'initialisation.
4  */
5 public class Singleton
6 {
7     /** Constructeur privé */
8     private Singleton()
9     {}
10
11    /** Instance unique pré-initialisée */
12    private static Singleton INSTANCE = new Singleton();
13
14    /** Point d'accès pour l'instance unique du singleton */
15    public static Singleton getInstance()
16    {   return INSTANCE;
17    }
18}
```

Lazy-loading

Dans l'implémentation ci-dessus, l'instance du Singleton est automatiquement créée au chargement de la classe par son classloader.
Bien qu'il s'agisse de la meilleure solution dans la plupart des cas, il peut arriver que l'on souhaite retarder l'initialisation de l'instance jusqu'au premier appel de "getInstance()". Cela se justifie par exemple si le programme n'a pas systématiquement besoin des services du singleton.

Implémentation basique

Voici l'implémentation la plus fréquente :

```
1 public class Singleton
2 {
3     /** Constructeur privé */
4     private Singleton()
5     {}
6
7     /** Instance unique non préinitialisée */
8     private static Singleton INSTANCE = null;
9
10    /** Point d'accès pour l'instance unique du singleton */
11    public static Singleton getInstance()
12    {
13        if (INSTANCE == null)
```

```
14        {   INSTANCE = new Singleton();
15        }
16        return INSTANCE;
17    }
18}
```

Cette implémentation semble correcte à première vue.
Pourtant, elle est extrêmement dangereuse en environnement multithreadé, car deux threads peuvent exécuter le test simultanément et créer ainsi chacun une instance du singleton. Elle doit donc être absolument proscrite.

Synchronisation globale

Afin de résoudre ce problème de concurrence des threads, on peut évidemment synchroniser la méthode "getInstance()" :

```
1 public class Singleton
2 {
3     /** Constructeur privé */
4     private Singleton()
5     {}
6
7     /** Instance unique non préinitialisée */
8     private static Singleton INSTANCE = null;
9
10    /** Point d'accès pour l'instance unique du singleton */
11    public static synchronized Singleton getInstance()
12    {
13        if (INSTANCE == null)
14        {   INSTANCE = new Singleton();
15        }
16        return INSTANCE;
17    }
18}
```

Le problème est ainsi résolu, mais au prix d'une pénalité sur les performances. Si le singleton est accédé souvent (système de log...), le ralentissement de l'application peut être important.

Synchronisation locale

Si l'on est attentif, on s'aperçoit que la synchronisation n'est requise qu'au moment exact de la création de l'instance. Ne pourrait-on donc pas distinguer supprimer la synchronisation globale sur la méthode, et ne l'appliquer que dans le cas où l'instance doit être créée ?

```
1 public class Singleton
2 {
3     /** Constructeur privé */
4     private Singleton()
5     {}
6
7     /** Instance unique non préinitialisée */
8     private static Singleton INSTANCE = null;
9
10    /** Point d'accès pour l'instance unique du singleton */
11    public static Singleton getInstance()
12    {
13        if (INSTANCE == null)
14        {
15            synchronized(Singleton.class)
16            {
17                if (INSTANCE == null)
18                {   INSTANCE = new Singleton();
```

```
19                 }
20             }
21         }
22         return INSTANCE;
23     }
24}
```

Hélas, cette solution, appelée "double-checked locking", ne fonctionne pas non plus (http://thecodersbreakfast.net/index.php?post/2008/02/03/13-pourquoi-le-double-check-locking-ne-fonctionne-pas) .

Technique du Holder

En revanche, une technique fonctionne correctement : la technique dite du "Holder".
Elle repose sur l'utilisation d'une classe interne privée, responsable de l'instanciation de l'instance unique du Singleton.

```
1  public class Singleton
2  {
3      /** Constructeur privé */
4      private Singleton()
5      {}
6
7      /** Holder */
8      private static class SingletonHolder
9      {
10         /** Instance unique non préinitialisée */
11         private final static Singleton instance = new Singleton();
12     }
13
14     /** Point d'accès pour l'instance unique du singleton */
15     public static Singleton getInstance()
16     {
17         return SingletonHolder.instance;
18     }
19}
```

Cette technique joue sur le fait que la classe interne ne sera chargée en mémoire que lorsque l'on y fera référence pour la première fois, c'est-à-dire lors du premier appel de "getInstance()" sur la classe Singleton. Lors de son chargement, le Holder initialisera ses champs statiques et créera donc l'instance unique du Singleton.

Cerise sur le gâteau, elle fonctionne correctement en environnement multithreadé et ne nécessite aucune synchronisation explicite !

Sérialisation des singletons

Pour finir, rappelez-vous qu'il existe une seconde façon d'instancier des objets : par désérialisation.

Si votre Singleton implémente java.io.Serializable, il faut absolument empêcher que sa désérialisation n'entraîne la création de nouvelles instances. Pour cela, la javadoc (http://java.sun.com/javase/6/docs/api/java/io/Serializable.html) indique que la méthode "readResolve()" permet de remplacer tout objet désérialisé par un objet personnalisé. Utilisons cela à notre avantage :

```
1  public class Singleton implements Serializable
2  {
3      /** Constructeur privé */
4      private Singleton()
```

```
5      {}
6
7      /** Instance unique pré-initialisée */
8      private static Singleton INSTANCE = new Singleton();
9
10     /** Point d'accès pour l'instance unique du singleton */
11     public static Singleton getInstance()
12     {   return INSTANCE;
13     }
14
15     /** Sécurité anti-désérialisation */
16     private Object readResolve() {
17         return INSTANCE;
18     }
19}
```

Ainsi, toute instance désérialisée du Singleton sera remplacée par notre instance unique.

Commentaires

1. Le dimanche 2 mars 2008, 20:31 par olive

   Notons que les objets instanciés pas Spring sont par défaut des singletons.

   Par ailleurs, un autre question peut être intéressante à traiter : comment avoir un singleton partagé entre plusieurs serveurs d'application ?

2. Le dimanche 6 avril 2008, 15:39 par HollyDays

   Pour compléter cet article, voici des éléments de réponse à deux questions complémentaires :
   1 - Comment obtenir un singleton lorsqu'on utilise plus d'un ClassLoader ?
   2 - Comment obtenir un singleton lorsqu'on utilise plus d'une machine virtuelle (je généralise ici la question d'olive) ?

   1 - Le singleton présenté ci-dessus est thread-safe, mais pas ClassLoader-safe. Or la machine virtuelle peut utiliser plusieurs ClassLoaders en même temps, et une classe en mémoire n'est unique en mémoire que par rapport à son ClassLoader (autrement dit, une classe peut être chargée plus d'une fois en mémoire, si elle l'est par un ClassLoader différent à chaque fois). Les serveurs d'application JavaEE, par exemple, utilisent un ClassLoader pour chaque application Web. Ainsi, deux applications Web pourront utiliser la même bibliothèque (éventuellement dans une version différente !) sans risquer de conflit.

   Si l'on veut que le Singleton soit commun à toutes les classes chargées quel que soit leur ClassLoader, il faut s'assurer que l'objet soit créé par le ClassLoader système et non le ClassLoader courant. Vous pouvez vous référer à l'article surguy.net/articles/commu... (http://surguy.net/articles/communication-across-classloaders.xml) qui explique comment faire (même si sa technique pour récupérer le ClassLoader système est un peu compliquée et pas très portable : appeler juste ClassLoader.getSystemClassLoader() aurait été plus simple...)

   2 - Un singleton partagé entre JVM est, d'abord et avant tout, un objet auquel on peut accéder à travers le réseau. Or le protocole standard de communication inter-JVM s'appelle RMI. Autrement dit, un singleton classique accessible via RMI peut tout à fait faire office de singleton multi-JVM. Et cette solution fonctionne également avec les

serveurs d'application : il suffit que l'opération distante de création renvoie toujours le même objet.

3. Le lundi 15 juillet 2013, 18:27 par Alex

   La version basique du singleton bien que nécessitant une instanciation immédiate est thread-safe ou je me trompe ?

4. Le lundi 15 juillet 2013, 22:15 par Olivier Croisier (http://thecodersbreakfast.net)

   Effectivement, si tout l'état est initialisé lors du chargement de la classe et n'est plus modifiable ensuite, le singleton est thread-safe.

5. Le mercredi 17 juillet 2013, 14:20 par HollyDays

   En relisant ce billet, il me semble qu'il comporte une petite erreur lorsqu'il dit : « l'instance du Singleton est automatiquement créée au démarrage de l'application.»

   A ma connaissance, ce n'est pas le cas. L'instance du Singleton est automatiquement créée au chargement de sa classe en mémoire. Si la classe est chargée au démarrage de l'application, alors le singleton sera effectivement instancié au démarrage de l'application. Mais contrairement à ce qui se passe avec d'autres langages de programmation (C + + typiquement), Java essaie justement de charger le minimum de classes au démarrage de l'application pour accélérer ce démarrage. Et c'est précisément cette propriété (initialisation du singleton uniquement lorsque sa classe est chargée en mémoire) qui fait que la solution du Holder fonctionne et permet, dans tous les cas, une initialisation retardée au premier usage du singleton.

   (Soit dit en passant, c'est justement parce que, en C/C + +, tous les static sont systématiquement initialisés au démarrage de l'application que les développeurs C/C + + ont inventé le pattern «double-check locking»... qui a marché, mais ne marche plus sur les processeurs modernes et avec les compilateurs modernes !)

   Pour conclure, j'ajoute que depuis Java 5, il existe encore une autre écriture du Singleton, qui à la fois est thread-safe, gère correctement la désérialisation, et dont le code source est beaucoup plus compact, puisque il tient en une seule ligne :

   enum Singleton { INSTANCE }

   Et voilà !

6. Le lundi 11 juillet 2016, 13:57 par Seb

   Hello,
   Une petite correction sur le lien vers '"double-checked locking", ne fonctionne pas non plus.' qui devrait pointer vers http://thecodersbreakfast.net/index... (http://thecodersbreakfast.net/index.php?post/2008/02/03/13-pourquoi-le-double-check-locking-ne-fonctionne-pas) , le changement de domaine n'a pas été reporté.

7. Le lundi 11 juillet 2016, 14:12 par Olivier Croisier (http://thecodersbreakfast.net)

   Merci, c'est rectifié !

8. Le samedi 16 décembre 2017, 14:29 par fxrobin

   Bonjour, je reviens aussi sur ce billet et sur l'un des commentaires qui n'a pas été pris en compte à mon avis.

   Dans le billet, juste après l'exemple du Singleton "basique" :
   "*Dans l'implémentation ci-dessus, l'instance du Singleton est automatiquement créée au démarrage de l'application*."

   Dans le commentaire de "HollyDays", le JVM (et son classloader synchonized) charge la classe uniquement quand il en a besoin et donc seulement au premier appel.

Voici la preuve :

```
  public class LazySingleton {

        private static final LazySingleton instance = new LazySingleton();

        private LazySingleton() {
                System.out.println("Construction du Singleton au premier app
        }

        public static final LazySingleton getInstance() {
                return instance;
        }

        @Override
        public String toString() {
           return String.format("Je suis le LazySingleton : %s", super.toStr
        }

  }
```

et le test

```
public class MainProg {

        public static void main(String[] args) {
                System.out.println("Démarrage du programme");
                System.out.println("Mon singleton n'est toujours pas chargé
                System.out.println("Bon allez, je me décide à l'appeler ..."
                LazySingleton singleton = LazySingleton.getInstance();
                System.out.println("Et maintenant je l'affiche ...");
                System.out.println(singleton);
        }

}
```

ce qui donne dans la console :

```
Démarrage du programme
Mon singleton n'est toujours pas chargé ...
Bon allez, je me décide à l'appeler ...
Construction du Singleton au premier appel
Et maintenant je l'affiche ...
Je suis le LazySingleton : demo.LazySingleton@7852e922
```

Par ailleurs, un Singleton qui se fonde sur une enum, oui très bien dans 90% des cas, mais on ne peut pas en hériter, car une enum ne s'hérite pas. Ce qui peut être pénalisant dans les 10% de cas.

Pour terminer, c'est dommage qu'un si bon article reste en l'état et ne soit pas corrigé, car de nombreuses se fondent dessus et, de fait, sont induites en erreur.

9. Le dimanche 17 décembre 2017, 19:14 par store

Avec l'introspection, le singleton tel que défini par les différentes méthodes ci-dessus devient obsolète, pour preuve le code suivant :

public static void main(String args) throws Exception

```
  {
     Singleton singleton1 = Singleton.getInstance();
```

```
        Constructor<?> constructor = singleton1.getClass().getDeclaredConstruct
        constructor.setAccessible( true );
        Singleton singleton2 = (Singleton) constructor.newInstance();
        if( singleton1 == singleton2 )
        {
           System.out.println( "Two objects are same" );
        }
        else
        {
           System.out.println( "Two objects are not same" );
        }
        singleton1.setValue( 1 );
        singleton2.setValue( 2 );
        System.out.println( singleton1.getValue() );
        System.out.println( singleton2.getValue() );
    }
```

Donne le résultat suivant :

Two objects are not same
1
2

10. Le lundi 18 décembre 2017, 11:04 par Olivier Croisier (http://olivier.croisier.free.fr)
    L'article visait à exposer les différentes options possibles, avec leurs avantages et inconvénients, dans le cadre d'une utilisation "normale" de Java. De manière générale, à partir du moment où vous utilisez la réflexion, vous pouvez oublier à peu près toutes les garanties du langage...
11. Le dimanche 24 décembre 2017, 10:00 par store
    Imaginez une seconde la taille du framework Spring ou Hibernate sans utiliser l'introspection, imaginez tous les bugs potentiels avec l'injection Spring sans Reflection, la bande des quatre ont défini le concept Singleton comme pattern, dans les faits surtout en Java, c'est un concept difficile à mettre en œuvre dans des environnements type J2EE.

Rechercher

ok

- Accueil
- Archives

Catégories

- Java / JEE

    - Articles
    - Java Quiz
    - Communauté
    - Liens externes
- Webdev
- Ergonomie & Webdesign
- Autres langages
- Divers
- Vie du blog

Tags (top 10)

- java
- communauté
- spring
- google
- internet
- javascript
- méthodologie
- entreprise
- wicket
- ergonomie

Tous les tags

S'abonner

- Fil des billets
- Fil des commentaires