Les nouveautés de Java 23 : partie 3

Le premier article de cette série a détaillé les fonctionnalités proposées via des JEPS relatives à la syntaxe et aux API dans le JDK 23. Comme pour les précédentes versions de Java, cette version 23 inclut des JEPs, mais aussi, et surtout, des évolutions et des améliorations sur la fiabilité (corrections de nombreux bugs), l'outillage, la performance et la sécurité.

Le second article a détaillé les autres évolutions dans les API ainsi que les fonctionnalités dépréciées et retirées.

Ce troisième article est consacré aux évolutions dans la JVM HotSpot, dans les outils et dans la sécurité.

Les évolutions dans la JVM HotSpot

Une seule JEP concerne des évolutions dans la JVM :

- JEP 474 : ZGC: Generational Mode by Default

Mais plusieurs autres évolutions sont proposées dans la JVM Hotspot.

La JEP 474 : le mode générationnel par défaut dans ZGC

ZGC (Z Garbage Collector) est un ramasse-miettes hautement scalable et à faible latence introduit dans la JVM Hotspot de Java 11 à titre expérimental et garanti pour une utilisation en production à partir de Java 15.

Son objectif principal est de gérer des heaps de très grande taille (jusqu'à 16 téraoctets) avec un impact minimal sur les performances des applications, en se concentrant particulièrement sur un temps de pause inférieur à la milliseconde.

Le mode générationnel pour Z Garbage Collector a été introduit dans le JDK 21 via la JEP 439. Avant cela, ZGC gérait tous les objets dans un même ensemble.

Avec le mode générationnel, ZGC gère deux générations séparées : une pour les objets jeunes et une pour les objets anciens, ce qui implique qu'elle peut récupérer les objets jeunes plus fréquemment. Ainsi, la surcharge du processeur est plus faible et la quantité de mémoire libérée est plus importante et plus rapidement.

La JEP 474 définit le mode générationnel comme utilisé par défaut pour ZGC : la valeur par défaut de l'option `ZGenerational` passe donc de `false` à `true`.

Le mode non générationnel, qui est généralement moins bon que le mode générationnel dans la plupart des cas, est déclaré comme déprécié dans cette version et sera supprimé dans une version

future.

Avec ces modifications, le comportement suivant sera observé en fonction des arguments de ligne de commande fournis :

- `-XX:+UseZGC`
  - Le mode générationnel de ZGC est utilisé
- `-XX:+UseZGC -XX:+ZGenerational`
  - Le mode générationnel de ZGC est utilisé. Un avertissement indiquant que l'option `ZGenerational` est dépréciée et sera supprimée dans une future version est émis

```
C:\java>java -XX:+UseZGC -XX:+ZGenerational HelloWorld
OpenJDK 64-Bit Server VM warning: Option ZGenerational was deprecated in version
23.0 and will likely be removed in a future release.
Hello World
```

- `-XX:+UseZGC -XX:-ZGenerational`
  - Le mode non générationnel de ZGC est utilisé. Un avertissement indiquant que l'option `ZGenerational` est dépréciée et sera supprimée dans une future version est émis. Un avertissement indiquant que le mode non générationnel est déprécié.

```
C:\java>java -XX:+UseZGC -XX:+ZGenerational HelloWorld
OpenJDK 64-Bit Server VM warning: Option ZGenerational was deprecated in version
23.0 and will likely be removed in a future release.
Hello World
```

Cette évolution peut induire des impacts sur les applications existantes utilisant ZGC :

- Les JVM qui utilisaient ZGC et qui basculent vers ZGC générationnel peuvent rencontrer des différences dans la sortie des journaux et dans les données disponibles à partir des API de maintenance et de gestion
- ZGC générationnel se comporte différemment du ZGC non générationnel. Le principal risque que cela présente est que certaines applications qui sont non générationnelles par nature pourraient subir une légère dégradation des performances. Cette situation devrait être rare

Le nouvel événement JFR `jdk.SerializationMisdeclaration` (JDK-8275338)

Un nouvel événement JFR est ajouté dans le JDK 23 : `jdk.SerializationMisdeclaration`.

Cet événement est émis à l'exécution lorsqu'un aspect des champs et méthodes liés à la sérialisation est mal déclaré. En activant l'événement `jdk.SerializationMisdeclaration`, un événement JFR sera émis pour chaque aspect incorrectement déclaré d'une classe `Serializable` lorsqu'elle est chargée dans la JVM.

Par exemple, si la méthode `writeObject()` d'une classe `Serializable` a une signature correcte, mais est déclarée publique par inadvertance, elle n'est pas utilisée par le mécanisme de sérialisation. Cela pourrait surprendre le développeur de la classe. Pour aider à diagnostiquer de tels problèmes, l'événement `jdk.SerializationMisdeclaration` peut être activé. Exemple :

le fichier User.java

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class User implements Serializable {

  private static final long serialVersionUID = -8980564301845020399L;

  private String nom;
  private String prenom;

  public User(String nom, String prenom) {
    this.nom = nom;
    this.prenom = prenom;
  }

  public User() {
  }

  private void readObject(ObjectInputStream aInputStream) throws
ClassNotFoundException, IOException {
    nom = aInputStream.readUTF();
    prenom = aInputStream.readUTF();
  }

  public void writeObject(ObjectOutputStream aOutputStream) throws IOException {
    aOutputStream.writeUTF(nom);
    aOutputStream.writeUTF(prenom);
  }

  public static void main(String[] args) {
    User user = new User("Dupond", "Martin");

    try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(
"User.ser"))) {
      oos.writeObject(user);
    } catch (IOException i) {
      i.printStackTrace();
    }
  }
}
```

Les événements `jdk.SerializationMisdeclaration` doivent être activés. Le profil par défaut `default.jfc` n'active pas ces événements, contrairement au profil `profile.jfc`.

Le code ci-dessus peut être compilé et exécuté avec une activation de JFR avec le profil `profile` enregistré dans un fichier `user.jfr`. Ce fichier contient un événement de type `jdk.SerializationMisdeclaration`

```
C:\java>javac User.java

C:\java>java
-XX:StartFlightRecording=settings=profile,disk=false,dumponexit=true,duration=60s,file
name=user.jfr User
[0.444s][info][jfr,startup] Started recording 1. The result will be written to:
[0.444s][info][jfr,startup]
[0.444s][info][jfr,startup] C:\java\user.jfr

C:\java>jfr print --events jdk.SerializationMisdeclaration user.jfr
jdk.SerializationMisdeclaration {
  startTime = 20:17:15.319 (2024-09-30)
  misdeclaredClass = User (classLoader = app)
  message = "method public void User.writeObject(java.io.ObjectOutputStream) throws
java.io.IOException must be private"
}
```

Le nouvel algorithme lors de Full GC pour Parallel GC (JDK-8329203)

L'algorithme précédent consistait en trois passages par chaque objet actif dans le heap de la JVM :

1. Marquage d'objets utilisés
2. Calcul de nouveaux emplacements pour chaque objet vivant
3. Déplacement d'objets vers de nouveaux emplacements et mise à jour des champs de chaque objet

Les emplacements des objets calculés à l'étape 2 sont stockés à l'aide de mémoire off heap pour éviter un quatrième passage pour parcourir les objets utilisés dans le heap.

Le problème est que ce schéma ne s'adapte pas bien à certaines typologies d'application. De plus, cette structure de données utilise 1,5 % du heap Java dans la mémoire off heap.

L'algorithme Full GC du ramasse-miettes Parallel GC a été réimplémenté pour utiliser un algorithme parallèle plus classique Mark-Sweep-Compact : il utilise désormais le même algorithme de full GC que celui des ramasse-miettes Serial et G1.

Le nouvel algorithme fonctionne nettement mieux pour les typologies d'applications problématiques et n'a pas besoin de mémoire supplémentaire, ce qui réduit l'empreinte mémoire. Il permet ainsi d'optimiser les performances dans certains cas spécifiques, en réduisant l'utilisation du heap de 1,5 % lors de l'utilisation de Parallel GC.

Le Parallel GC lève une `OutOfMemoryException` avant que le heap ne soit complètement étendu (JDK-8328744)

Un bug existant peut avoir empêché l'utilisation complète du heap de la JVM alloué avec l'option en ligne de commande -Xmx. Ce bug a été corrigé.

En tant qu'effet de bord de ce correctif, les JVM peuvent rencontrer une utilisation accrue du heap lors de l'utilisation de Parallel GC. Les utilisateurs doivent, si nécessaire, ajuster la taille maximale du heap.

Le changement de la valeur de `LockingMode` de `LM_LEGACY` à `LM_LIGHTWEIGHT` (JDK-8319251)

Un nouveau mécanisme de verrouillage léger pour le verrouillage du moniteur d'objets a été introduit dans le JDK 21 (JDK-8291555).

L'option `LockingMode` de la JVM a été introduite pour permettre la sélection de ce nouveau mécanisme (`LM_LIGHTWEIGHT`, valeur `2`) à la place du mécanisme par défaut (`LM_LEGACY`, valeur `1`).

Dans le JDK 23, la valeur par défaut de `LockingMode` a été remplacée par `LM_LIGHTWEIGHT`. Cela ne devrait pas modifier le comportement sémantique du verrouillage du moniteur Java. Il devrait être neutre en termes de performances pour presque toutes les applications.

Il est possible de revenir au mécanisme historique, en utilisant l'option en ligne de commande `-XX:LockingMode=1`, mais il faut tenir compte que le mode historique devrait être supprimé dans une version future.

L'assouplissement de l'alignement des tableaux (JDK-8139457)

Les bases des éléments de tableau ne sont plus alignées inconditionnellement sur huit octets. Au lieu de cela, ils sont désormais alignés sur la taille de leur type d'élément. Cela améliore l'empreinte mémoire dans certains modes de la JVM. Comme l'alignement des éléments de tableau Java n'est pas exposé aux utilisateurs, il n'y a aucun impact sur le code Java standard qui accède aux éléments individuels.

Par contre, ce changement a des implications sur les méthodes d'accès de bas niveau. Les accès non sécurisés aux tableaux pourraient désormais être non-alignés. Par exemple, il n'est pas garanti que le fonctionnement de `Unsafe.getLong(byteArray, BYTE_ARRAY_BASE_OFFSET + 0)` soit effectué sur des plates-formes qui n'autorisent pas les accès non-alignés.

Une solution de contournement est l'utilisation des méthodes `Unsafe.[get|put]Unaligned*`. Les API `ByteBuffer` et `VarHandle` qui autorisent les vues de `byte[]` sont mises à jour pour refléter cette modification (JDK-8318966). Les tableaux obtenus via `GetPrimitiveArrayCritical()` ne doivent pas être exploités sous l'hypothèse d'un alignement de base de tableau particulier.

L'option `TrimNativeHeapInterval` de la JVM Hotspot n'est plus expérimentale (JDK-8325496)

L'option `TrimNativeHeapInterval` permet à la JVM de réduire le heap natif à intervalles réguliers.

La syntaxe de cette option est de la forme : `-XX:TrimNativeHeapInterval=millis`

La valeur, précisée en ms, indique l'intervalle auquel la JVM réduira le heap natif. Des valeurs plus faibles récupéreront de la mémoire plus rapidement au prix d'une surcharge plus élevée. La valeur `0` (par défaut) désactive la réduction du heap natif. La réduction du heap natif est effectuée dans un thread dédié.

Cette option, qui était une option expérimentale, a été officiellement promue en tant qu'option produit.

Cette option n'est disponible que sur Linux avec GNU C Library (glibc).

```
C:\java>java -XX:TrimNativeHeapInterval=1000 Main
[0.032s][warning][trimnative] Native heap trim is not supported on this platform
```

Les évolutions dans l'outillage

Plusieurs améliorations sont apportées à des outils fournis dans le JDK.

La modification de la politique de traitement des annotations par défaut de javac (JDK-8321314)

Les annotations peuvent être traitées lors de la compilation, où `javac` analyse les fichiers sources à compiler à la recherche d'annotations, puis le classpath pour trouver les processeurs d'annotations correspondants, afin qu'ils puissent générer du code source.

Jusqu'au JDK 22, cette fonctionnalité était activée par défaut, ce qui était peut-être raisonnable lorsqu'elle a été introduite dans le JDK 6 vers 2006, mais d'un point de vue actuel, dans le but de rendre la sortie de construction plus robuste contre les processeurs d'annotations placés dans le classpath par inadvertance, c'est beaucoup moins raisonnable.

Jusqu'au JDK 22 inclus, le code qui nécessite un traitement des annotations avant la compilation pouvaient s'appuyer sur le comportement par défaut de `javac` pour traiter les annotations, mais ce n'est plus le cas.

À partir du JDK 23, l'option `-proc:<policy>` a été enrichie : `<policy>` peut maintenant avoir les valeurs suivantes :

- `none` : compilation sans traitement d'annotation. Cette politique existe depuis le JDK 6
- `only` : traitement des annotations sans compilation. Cette politique existe depuis le JDK 6

- `full` : traitement des annotations suivi de la compilation. Cette politique est la valeur par défaut dans les JDK ≤ 22, mais la valeur elle-même est nouvelle

Par conséquent, à partir du JDK 23, `javac` nécessite une option de ligne de commande supplémentaire pour activer le traitement des annotations. Le traitement des annotations n'est exécuté qu'avec une configuration explicite de ce traitement ou avec une demande explicite d'exécution du traitement des annotations sur la ligne de commande `javac`. Si ni `-processor`, ni `--processor-path` ni `--processor-module-path` n'est utilisé, `-proc:only` ou `-proc:full` doit être utilisé. En d'autres termes, en l'absence d'autres options de ligne de commande, `-proc:none` est la valeur par défaut dans le JDK 23.

Il s'agit d'un changement de comportement par rapport à la valeur par défaut existante qui consiste à chercher un traitement d'annotations en recherchant des processeurs dans le classpath sans qu'aucune option explicite liée au traitement d'annotation ne doive être présente.

Ainsi les utilisations de `javac` qui doivent traiter des annotations sans aucune configuration explicite de ces traitements devront être mis à jour pour continuer à exécuter les processeurs d'annotations. Dans les JDK 21 et 22, `javac` affiche une note identifiant ces appels.

Pour conserver l'ancien comportement, l'option `-proc:full` peut être passé à `javac`. La prise en charge de `-proc:full` a été rétroportée dans plusieurs mises à jour de versions LTS antérieures.

Avec Maven, il est possible d'utiliser la propriété `maven.compiler.proc` pour préciser le comportement : `-Dmaven.compiler.proc=full` dans l'option en ligne de commande.

Plusieurs mesures ont été prises pour aider les projets à se préparer au passage à `-proc:full`.

- Depuis les mises à jour de sécurité du JDK d'avril 2024, la prise en charge de `-proc:full` a été rétroportée dans les versions 17.0.11 et 11.0.23 pour les distributions JDK et OpenJDK d'Oracle
- De plus, la version 8u411 d'Oracle prend également en charge `-proc:full`. Avec `-proc:full` rétroporté, il est possible de configurer une compilation qui fonctionnera de la même manière avant et après le changement de la politique par défaut de `javac`
- À partir du JDK 21, `javac` affiche un message d'information si l'utilisation implicite du traitement des annotations dans le cadre de la politique par défaut est détectée

Le nouvel avertissement dangling-doc-comments de -Xlint (JDK-8303689)

Un nouvel avertissement nommé `dangling-doc-comments` est fourni pour l'option `-Xlint` de `javac`, afin de détecter les problèmes liés au placement des commentaires de documentation dans le code source.

Cet avertissement peut être spécifié explicitement (par exemple, `-Xlint:dangling-doc-comments`) ou implicitement en activant toutes sous-options (par exemple, `-Xlint` ou `-Xlint:all`).

Lorsque l'avertissement est activé, `javac` signale tout commentaire de documentation inattendu ou mal placé à proximité d'une déclaration, comme dans les situations suivantes :

- un commentaire de documentation pour une classe de niveau supérieur avant la déclaration du package ou d'importations
- un commentaire de documentation pour une déclaration qui apparaît après le premier élément de cette déclaration, par exemple après des annotations ou des modificateurs de la déclaration
- tout commentaire supplémentaire de documentation avant une déclaration, que `javac` ignorerait autrement

le fichier MaClasse.java

```
/**
 * Un commentaire de documentation mal placé
 */

import java.util.*;

public class MaClasse {

  public static void main(String[] args) {
    System.out.println("Bonjour");
  }

  @Override
  /**
   * Un second commentaire de documentation mal placé
   */
  public String toString() {
    return "MaClasse";
  }
}
```

La compilation de cette classe avec tous les avertissements activés affiche deux avertissements de type `dangling-doc-comments`.

```
C:\java>javac -Xlint:all MaClasse.java
MaClasse.java:1: warning: [dangling-doc-comments] documentation comment is not
attached to any declaration
  /**
  ^
MaClasse.java:14: warning: [dangling-doc-comments] documentation comment is not
attached to any declaration
  /**
  ^
2 warnings
```

Comme pour toute sous-option pour `-Xlint`, les avertissements peuvent être supprimés localement, en utilisant l'annotation `@SuppressWarnings` sur une déclaration englobante, en spécifiant les noms des sous-options pour les avertissements à supprimer.

le fichier MaClasse.java

```
/**
 * Un commentaire de documentation mal placé
 */

import java.util.*;
@SuppressWarnings("dangling-doc-comments")
public class MaClasse {

  public static void main(String[] args) {
    System.out.println("Bonjour");
  }

  @Override
  /**
   * Un second commentaire de documentation mal placé
   */
  public String toString() {
    return "MaClasse";
  }
}
```

La compilation de cette classe avec tous les avertissements activés n'affiche aucun avertissement puisque les avertissements de type `dangling-doc-comments` ont été supprimés au niveau de la classe.

```
C:\java>javac -Xlint:all MaClasse.java

C:\java>
```

Remarque : il est possible que lorsque l'avertissement est activé, `javac` signale des « faux positifs » s'il y a des commentaires décoratifs commençant par `/**` et peuvent donc ressembler à un commentaire de documentation. Par exemple, les commentaires qui utilisent une ligne d'astérisques avant et après le reste du texte du commentaire, pour aider à faire ressortir le commentaire. La solution dans de tels cas est de changer le commentaire de sorte qu'il ne commence pas par `/**`, par exemple en changeant au moins un des deux astérisques en un autre caractère.

Le support des modules JavaScript par l'outil `javadoc` (JDK-8317621)

L'option `--add-script` de l'outil `javadoc` prend désormais en charge les modules JavaScript en plus des fichiers de script conventionnels. Les modules sont détectés automatiquement en inspectant l'extension ou le contenu du fichier passé en argument d'option.

L'amélioration de la navigation structurelle dans la documentation `javadoc` (JDK-8320458)

La documentation de l'API générée par le doclet standard est désormais dotée de fonctionnalités de navigation améliorées, notamment une barre latérale contenant une table des matières pour la page actuelle et un fil d'Ariane pour l'élément d'API actuel dans l'en-tête de la page.

[barre latérale javadoc]

Dans la documentation des classes et des interfaces, les entrées de la table des matières peuvent être filtrées à l'aide d'un champ de saisie de texte en haut de la barre latérale.

[barre latérale javadoc]

Un bouton situé en bas de la barre latérale permet de réduire ou de développer la table des matières pour la page en cours.

[barre latérale javadoc]

La vérification des classes par `javap` (JDK-8182774)

La nouvelle option `-verify` de l'outil `javap` affiche des informations supplémentaires sur la vérification du bytecode de classe.

Les évolutions relatives à la sécurité

Il y a plusieurs mises à jour des certificats racines de deux fournisseurs dans le truststore cacerts.

Certaines fonctionnalités renforcent la sécurité sur des points précis.

Les mises à jour de certificats racines dans le truststore cacerts (JDK-8316138 et JDK-8321408)

Quatre certificats racines ont été ajoutés dans le truststore cacerts :

- GlobalSign R46 et E46
- Certainly R1 et E1

La méthode `Subject.getSubject()` requiert la propriété `java.security.manager` avec la valeur 'allow' (JDK-8296244)

En prévision de la suppression du Security Manager dans une version ultérieure, la méthode `Subject.getSubject(AccessControlContext)`, dépréciée pour suppression, a été modifiée pour lever une exception de type `UnsupportedOperationException` si elle est appelée lorsqu'un Security Manager

n'est pas permis.

Lorsque le Security Manager sera supprimé dans une version ultérieure, la méthode `Subject.getSubject(AccessControlContext)` sera encore modifiée pour lever une exception de type `UnsupportedOperationException` de manière inconditionnelle.

Pour cette version 23 du JDK, le comportement est différent selon qu'un Security Manager est autorisé ou non :

- Si un gestionnaire de sécurité est autorisé, c'est-à-dire que la propriété système `java.security.manager` est définie sur la ligne de commande avec une chaîne vide, un nom de classe ou la valeur `allow`, il n'y a pas de changement de comportement par rapport aux versions précédentes.
- Si un gestionnaire de sécurité n'est pas autorisé, si la propriété système `java.security.manager` n'est pas définie sur la ligne de commande ou a été définie sur la ligne de commande avec la valeur `disallow`, les méthodes `doAs()` ou `callAs()` appellent une action avec un `Subject` comme `Subject` actuel pour l'exécution de l'action sur une période limitée. Le `Subject` peut être obtenu à l'aide de la méthode `Subject.current()` lorsqu'il est appelé par le code exécuté par l'action. La méthode `Subject.getSubject()` ne peut pas obtenir le `Subject`, car cette méthode lève une exception de type `UnsupportedOperationException`. L'objet n'est pas hérité automatiquement lors de la création ou du démarrage de nouveaux threads avec l'API Thread. L'objet est hérité par les threads enfants lors de l'utilisation de la concurrence structurée.

La solution de contournement temporaire dans cette version pour maintenir le code plus ancien fonctionnel consiste à utiliser l'option `-Djava.security.manager=allow` pour permettre la définition d'un Security Manager. La méthode `Subject.getSubject()` ne définit pas de Security Manager, mais exige que la fonctionnalité soit autorisée en raison du paramètre `AccessControlContext`.

La migration du code utilisant `Subject.doAs()` et `Subject.getSubject()` est fortement encouragée vers les API de remplacement, `Subject.callAs()` et `Subject.current()`, dès que possible. L'outil `jdeprscan` peut être utilisé pour analyser le classpath à la recherche d'utilisations d'API obsolètes et peut être utile pour trouver l'utilisation de ces deux méthodes.

Le code qui stocke un `Subject` dans un `AccessControlContext` et appelle `AccessController.doPrivileged()` avec ce contexte doit également être migré dès que possible, car ce code cessera de fonctionner lorsque le Security Manager sera supprimé.

Les mainteneurs de code qui utilise l'API `Subject` doivent également auditer leur code pour tous les cas où il peut dépendre de l'héritage du `Subject` actuel dans les threads nouvellement créés. Ce code doit être modifié pour transmettre le sujet au thread nouvellement créé ou modifié pour utiliser la concurrence structurée.

Le changement de comportement des méthodes `RandomGeneratorFactory.create(long)` et `create(byte[])` (JDK-8332476)

Dans les versions précédentes du JDK, `RandomGeneratorFactory.create(long)` utilise comme solution

de fall back l'invocation de la méthode `create()` sans argument si l'algorithme sous-jacent ne prend pas en charge une graine (seed) longue. La méthode `create(byte[])` fonctionne de la même manière.

À partir de cette version, ces méthodes lèvent désormais une exception `UnsupportedOperationException` plutôt que d'utiliser silencieusement `create()`.

Le support pour le Keystore KeychainStore-ROOT (JDK-8320362)

L'utilisation de la classe `HttpsURLConnection` pour se connecter à l'url https://github.com échoue avec l'utilisation du trustStore KeychainStore de MacOS :

```
java -Djavax.net.ssl.trustStoreType=KeychainStore DemoHttpsURLConnection
https://github.com

SSLHandshakeException: PKIX path building failed:
sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid
certification path to requested target
```

Cela se produit parce que le KeychainStore ne contient pas les certificats d'autorité de certification intermédiaire requis et que le fournisseur d'Apple ne renvoie pas de certificats racines.

Le « KeychainStore » du fournisseur Apple prend désormais en charge deux types de keystores :

- « KeychainStore » : contient les clés privées et les certificats pour le trousseau actuel de l'utilisateur
- « KeychainStore-ROOT » : contient les certificats du trousseau de certificats racine du système

Les options `thread` et `timestamp` de la propriété `java.security.debug` (JDK-8051959)

La propriété système `java.security.debug` accepte désormais des arguments qui ajoutent l'ID du thread, le nom du thread, les informations de l'appelant et les informations d'horodatage aux instructions de débogage pour tous les composants ou un composant spécifique.

`+timestamp` peut être ajouté aux options de débogage pour afficher un horodatage pour cette option de débogage

`+thread` peut être ajouté aux options de débogage pour afficher les informations sur le thread et l'appelant pour cette option de débogage

Exemples :

- `-Djava.security.debug=all+timestamp+thread` ajoute des informations d'horodatage et de thread à chaque instruction debug générée.

- `-Djava.security.debug=properties+timestamp` ajoute des informations d'horodatage à chaque instruction de débogage générée pour le composant properties.

Il est possible d'utiliser l'option `-Djava.security.debug=help` pour afficher une liste complète des composants et arguments pris en charge.

```
C:\java>java -Djava.security.debug=help Main

all             turn on all debugging
access          print all checkPermission results
certpath        PKIX CertPathBuilder and
                CertPathValidator debugging
combiner        SubjectDomainCombiner debugging
gssloginconfig
                GSS LoginConfigImpl debugging
configfile      JAAS ConfigFile loading
configparser    JAAS ConfigFile parsing
jar             jar verification
logincontext    login context results
jca             JCA engine class debugging
keystore        KeyStore debugging
pcsc            Smartcard library debugging
policy          loading and granting
provider        security provider debugging
pkcs11          PKCS11 session manager debugging
pkcs11keystore
                PKCS11 KeyStore debugging
pkcs12          PKCS12 KeyStore debugging
properties      Security property and configuration file debugging
sunpkcs11       SunPKCS11 provider debugging
scl             permissions SecureClassLoader assigns
securerandom    SecureRandom
ts              timestamping
x509            X.509 certificate debugging

+timestamp can be appended to any of above options to print
                a timestamp for that debug option
+thread can be appended to any of above options to print
                thread and caller information for that debug option

The following can be used with access:

stack           include stack trace
domain          dump all domains in context
failure         before throwing exception, dump stack
                and domain that didn't have permission

The following can be used with stack and domain:

permission=<classname>
```

```
              only dump output if specified permission
              is being checked
codebase=<URL>
              only dump output if specified codebase
              is being checked

The following can be used with provider:

engine=<engines>
              only dump output for the specified list
              of JCA engines. Supported values:
              Cipher, KeyAgreement, KeyGenerator,
              KeyPairGenerator, KeyStore, Mac,
              MessageDigest, SecureRandom, Signature.

The following can be used with certpath:

ocsp          dump the OCSP protocol exchanges
verbose       verbose debugging

The following can be used with x509:

ava           embed non-printable/non-escaped characters in AVA components as hex
strings

Note: Separate multiple options with a comma
```

La possiblité d'une vérification sensible à la casse dans ccache et keytab lors de la recherche d'entrée Kerberos (JDK-8331975)

Lors de la recherche d'une entrée keytab ou d'un cache d'informations d'identification (ccache) pour un Principal dans Kerberos, le nom du Principal est comparé au nom de l'entrée d'une manière non sensible à la casse. Toutefois, de nombreuses implémentations Kerberos traitent les noms de Principal comme sensibles à la casse. Par conséquent, si deux Principal ont des noms qui ne diffèrent que par la casse, il existe un risque de sélectionner le mauvais keytab ou l'entrée ccache.

Une nouvelle propriété de sécurité nommée `jdk.security.krb5.name.case.sensitive` est introduite pour contrôler la comparaison des noms. Si cette propriété vaut `true`, alors la comparaison des noms de Principal lors de la recherche d'entrée keytab et ccache sera sensible à la casse. La valeur par défaut est `false` pour garantir la compatibilité descendante.

De plus, si une propriété système portant le même nom est spécifiée, elle remplacera la valeur de la propriété de sécurité définie dans le fichier `java.security`.

Conclusion

Java poursuit son évolution avec le JDK 23 qui propose beaucoup de nouveautés et d'améliorations qui vont permettre à Java de rester pertinent aujourd'hui et demain.

Ce troisième article de cette série est consacré aux évolutions dans JVM Hotspot, dans les outils, les API et dans la sécurité.

Il y a notamment un gros changement sur le comportement par défaut du ramasse-miettes Z (ZGC) et que les développeurs doivent se préparer à la suppression des méthodes d'accès à la mémoire de la classe `sun.misc.Unsafe`.

Toutes les évolutions proposées dans le JDK 23 sont détaillées dans les releases notes.

N'hésitez donc pas à télécharger et tester une distribution du JDK 23 auprès d'un fournisseur pour anticiper la release de la prochaine version LTS de Java.