富士通 Java ブログ

JDK 11 から JDK 21 へ移行する際の注意点
～主な非互換を伴う機能変更の影響と対処方法～

2026 年 2 月 20 日 初版

田端 大志

OpenJDK では、重要な機能変更は JDK Enhancement Proposal（以降、「JEP」）として提案され、6 か月ごとのメジャーバージョンリリースで導入されます。JEP として機能変更が提案される際には、その目的や動機、リスクをまとめたドキュメントが作成されます。
JEP の中には非互換を生じるものもあり、既存システムに影響を与える場合があるため、JDK をバージョンアップする際には、そのバージョンで導入された JEP の内容を把握しておく必要があります。1 回のメジャーバージョンアップごとに導入される JEP の数は（JDK 9 を除き）概ね 10 件前後ですが、JDK のバージョンアップをまとめて行う場合には、各バージョンで導入されたすべての JEP を確認しなければなりません。
多くの JDK ベンダが長期サポート（LTS）バージョンとして提供していることから、JDK 11 から JDK 21 へ一気に移行するケースがありますが、その場合は、JDK 11 から JDK 21 の間で導入された合計 112 件(注 1)の JEP を確認する必要があります。
本稿では、JDK 21 への移行の勘所として、この 112 件の中から既存システムへ影響を与える可能性が特に高い 5 件を抜粋し、それぞれの「影響を受けるケース」「影響内容」「対処」について解説します。

なお、JDK 8 から移行する場合は、JDK 9 から JDK 11 までの JEP も考慮に入れる必要があります。本稿で解説はしませんが、各バージョンで導入された JEP 一覧のページを参考情報として以下に示します。

JDK 9 で導入された JEP 一覧
JDK 10 で導入された JEP 一覧
JDK 11 で導入された JEP 一覧

注 1) プレビュー版や実験版として導入された JEP も計上しています。

JEP 363: Remove the Concurrent Mark Sweep (CMS) Garbage Collector

JDK 14 以降、Concurrent Mark Sweep GC（以降、「CMS GC」）が削除されました。

影響を受けるケース

以下のいずれかのオプションを指定して、CMS GC を有効化、または、挙動を調整している場合。

- -XX:+UseConcMarkSweepGC
- -XX:+CMSAbortSemantics
- -XX:CMSAbortablePrecleanMinWorkPerIteration
- -XX:CMSBitMapYieldQuantum
- -XX:CMSBootstrapOccupancy
- -XX:+CMSClassUnloadingEnabled
- -XX:CMSClassUnloadingMaxInterval
- -XX:+CMSCleanOnEnter
- -XX:CMSConcMarkMultiple
- -XX:+CMSConcurrentMTEnabled
- -XX:CMSCoordinatorYieldSleepCount
- -XX:+CMSEdenChunksRecordAlways
- -XX:CMSExpAvgFactor

- -XX:+CMSExtrapolateSweep
- -XX:CMSIncrementalSafetyFactor
- -XX:CMSIndexedFreeListReplenish
- -XX:CMSInitiatingOccupancyFraction
- -XX:CMSIsTooFullPercentage
- -XX:CMSLargeCoalSurplusPercent
- -XX:CMSLargeSplitSurplusPercent
- -XX:+CMSLoopWarn
- -XX:CMSMaxAbortablePrecleanLoops
- -XX:CMSMaxAbortablePrecleanTime
- -XX:CMSOldPLABMax
- -XX:CMSOldPLABMin
- -XX:CMSOldPLABNumRefills
- -XX:CMSOldPLABReactivityFactor
- -XX:+CMSOldPLABResizeQuicker
- -XX:CMSOldPLABToleranceFactor
- -XX:+CMSPLABRecordAlways
- -XX:+CMSParallelInitialMarkEnabled
- -XX:+CMSParallelRemarkEnabled
- -XX:+CMSParallelSurvivorRemarkEnabled
- -XX:CMSPrecleanDenominator
- -XX:CMSPrecleanIter
- -XX:CMSPrecleanNumerator
- -XX:+CMSPrecleanRefLists1
- -XX:+CMSPrecleanRefLists2
- -XX:+CMSPrecleanSurvivors1
- -XX:+CMSPrecleanSurvivors2
- -XX:CMSPrecleanThreshold
- -XX:+CMSPrecleaningEnabled
- -XX:+CMSPrintChunksInDump
- -XX:+CMSPrintObjectsInDump
- -XX:CMSRemarkVerifyVariant
- -XX:+CMSReplenishIntermediate
- -XX:CMSRescanMultiple
- -XX:CMSSamplingGrain
- -XX:+CMSScavengeBeforeRemark
- -XX:CMSScheduleRemarkEdenPenetration
- -XX:CMSScheduleRemarkEdenSizeThreshold
- -XX:CMSScheduleRemarkSamplingRatio
- -XX:CMSSmallCoalSurplusPercent
- -XX:CMSSmallSplitSurplusPercent
- -XX:+CMSSplitIndexedFreeListBlocks
- -XX:CMSTriggerRatio
- -XX:CMSWorkQueueDrainThreshold
- -XX:+CMSYield
- -XX:CMSYieldSleepCount
- -XX:CMSYoungGenPerWorker
- -XX:CMS_FLSPadding
- -XX:CMS_FLSWeight
- -XX:CMS_SweepPadding
- -XX:CMS_SweepTimerThresholdMillis
- -XX:CMS_SweepWeight
- -XX:+FLSAlwaysCoalesceLarge
- -XX:FLSCoalescePolicy
- -XX:FLSLargestBlockCoalesceProximity
- -XX:OldPLABWeight
- -XX:ParGCDesiredObjsFromOverflowList

- -XX:+ParGCTrimOverflow
- -XX:+ParGCUseLocalOverflow
- -XX:+ResizeOldPLAB
- -XX:+UseCMSBestFit
- -XX:+UseCMSInitiatingOccupancyOnly
- -XX:CMSAbortablePrecleanWaitMillis
- -XX:CMSWaitDuration
- -XX:CMSTriggerInterval
- -XX:+FLSVerifyAllHeapReferences
- -XX:+FLSVerifyLists
- -XX:+FLSVerifyIndexTable
- -XX:+BindCMSThreadToCPU
- -XX:CPUForCMSThread
- -XX:ParGCStridesPerThread
- -XX:ParGCCardsPerStrideChunk
- -XX:+BlockOffsetArrayUseUnallocatedBlock

影響内容

CMS GC に関連するオプションが認識されず、Java Virtual Machine（以降、「JVM」）を起動することができません。

（例）OpenJDK 21 で、-XX:+UseConcMarkSweepGC オプションを指定した場合のエラー内容

```
Unrecognized VM option 'UseConcMarkSweepGC'
Error: Could not create the Java Virtual Machine.
Error: A fatal exception has occurred. Program will exit.
```

対処

JDK 14 以降では CMS GC を利用できないため、代替のガーベジコレクターへ移行する必要があります。以下の対処を順番に行ってください。

1. CMS GC に関連するオプションを JVM の起動オプションから削除します。
2. 移行先のガーベジコレクターを有効化するオプションを JVM の起動オプションに指定します。
   特別な要件がない場合、多くのワークロードで良好なパフォーマンスを発揮する G1 GC への移行を推奨します。
   大規模なヒープを利用する場合や、極めて高い応答性能が要求される場合、ZGC や Shenandoah GC への移行も検討してください。ただし、これらのガーベジコレクターは G1 GC と比べてアプリケーションのスループットが低下する傾向があります。

JEP 373: Reimplement the Legacy DatagramSocket API

JDK 15 以降、java.net.DatagramSocket、および、java.net.MulticastSocket の API の処理を行う java.net.DatagramSocketImpl の内部実装が、古くからの JDK で使われてきたレガシーな実装から、NIO を利用した実装に置き換わりました。

影響を受けるケース

以下のいずれかに該当する場合に影響を受けます。

1. java.net.DatagramSocket、または、java.net.MulticastSocket の connect(InetAddress address, int port)を呼び出し、ルーティングできないアドレスや 0 番ポートのような不正な接続先に接続している場合。
2. java.net.DatagramSocket、または、java.net.MulticastSocket の disconnect を呼び出して接続先との関連付けを解除した後、同じソケットで再度 connect を呼び出して接続を行っている場合。
3. java.net.DatagramSocket、または、java.net.MulticastSocket で、connect を使って接続を確立する前にデータグラムを受信している場合。

上記以外にも、API 仕様に定義されていない動作に依存する使い方をしている場合に発生する影響があります。そのようなケースの詳細は、JEP 373 のドキュメントをご参照ください。

影響内容

"影響を受けるケース"で示した番号に応じて、以下の影響が発生します。

1. connect で不正な接続先への接続に失敗したとき、java.io.UncheckedIOException がスローされるようになりました。
   なお、JDK 14 以前は不正な接続先に connect で接続しても例外はスローされませんでしたが、接続後にデータグラムの送受信を行ったとき、例外がスローされる可能性がありました。
2. 接続先との関連付けの解除に失敗したとき、java.io.UncheckedIOException がスローされるようになりました。
   なお、JDK 14 以前は接続先との関連付けの解除に失敗しても例外はスローされませんでしたが、同じソケットで再度 connect を呼び出して接続し、データグラムの送受信を行ったとき、例外がスローされる可能性がありました。
3. connect を使って接続を確立する前に受信していたデータグラムは、connect を使って接続を確立したときに破棄されます。

対処

"影響を受けるケース"で示した番号に応じて、以下の方法で回避・解決してください。

1. 接続先アドレス、および、ポート番号が適切となるようにコードを変更してください。
2. disconnect が java.io.UncheckedIOException をスローしたときは、ソケットの状態が不定になることがあるため、ソケットを閉じるようにコードを変更してください。
3. 原則として、connect で接続を確立してから通信を行うようにコードを変更してください。 どうしても接続前に受信したデータグラムを取り扱う必要がある場合は、connect を利用する代わりに、受信したパケットが目的の相手からのものであるかをフィルタリングする処理をアプリケーションで実装してください。

JEP 374: Deprecate and Disable Biased Locking

JDK 15 以降、パフォーマンス最適化の一つであったバイアスロックがデフォルトで無効化され、関連するすべてのオプションが非推奨となりました。その後、JDK 18 でバイアスロックは完全に削除されました。

影響を受けるケース

以下のいずれかに該当する場合に影響を受けます。

1. 以下のいずれかの JVM オプションを指定して、バイアスロックの挙動を調整している場合。
   - -XX:+UseBiasedLocking
   - -XX:BiasedLockingStartupDelay
   - -XX:BiasedLockingBulkRebiasThreshold
   - -XX:BiasedLockingBulkRevokeThreshold
   - -XX:BiasedLockingDecayTime
   - -XX:+UseOptoBiasInlining
   - -XX:+PrintBiasedLockingStatistics
   - -XX:+PrintPreciseBiasedLockingStatistics
2. アプリケーションがバイアスロックによる性能向上に依存している場合。なお、JDK 14 以前はバイアスロックはデフォルトで有効であるため、以下のような競合がない同期処理を大量に行うアプリケーションでは、バイアスロックによる性能向上に意図せず依存している可能性があります。
   - シングルスレッドで、synchronized メソッドを大量に呼び出しているアプリケーション
   - シングルスレッドで、レガシーなコレクション API である java.util.Vector や java.util.Hashtable のメソッドを大量に呼び出しているアプリケーション

影響内容

"影響を受けるケース"で示した番号に応じて、以下の影響が発生します。

1. バイアスロックに関連するオプションが認識されず、JVM を起動することができません。

（例）OpenJDK 21 で、-XX:+UseBiasedLocking オプションを指定した場合のエラー内容


```
Unrecognized VM option 'UseBiasedLocking'
Error: Could not create the Java Virtual Machine.
Error: A fatal exception has occurred. Program will exit.
```

2. バイアスロックが無効になることで、バイアスロックによる性能向上に依存していたアプリケーションでは、性能劣化が見られる可能性があります。

対処

"影響を受けるケース"で示した番号に応じて、以下の方法で回避・解決してください。

1. バイアスロックに関連するオプションを JVM の起動オプションから削除します。
2. バイアスロックによる性能向上に依存しない API を使用することを推奨します。
    - シングルスレッド環境では、メソッド呼び出し時に内部的な同期を行う java.util.Vector や java.util.Hashtable の代わりに、メソッド呼び出し時に内部的な同期を行わない java.util.ArrayList や java.util.HashMap を使用してください。
    - マルチスレッド環境では、java.util.concurrent パッケージの高性能な並行処理用クラスを使用してください。

JEP 390: Warnings for Value-Based Classes

JDK 16 以降、値ベースクラスに関連する一部の操作に、新たな警告が導入されました。 値ベースクラスについては「値ベースのクラス(Java SE 21 & JDK 21)」をご参照ください。

影響を受けるケース

以下のいずれかに該当する場合に影響を受けます。

1. プリミティブ型ラッパークラスのコンストラクタを直接呼び出している場合。
2. 値ベースクラスのインスタンスをロックオブジェクトとしている場合。

影響内容

"影響を受けるケース"で示した番号に応じて、以下の影響が発生します。

1. javac コマンドでのコンパイル時に、「削除予定」であることを示す警告が表示されます。

（例）OpenJDK 21 で、該当のコードを含むプログラムをコンパイルした場合の警告

```
Test.java:3: 警告: [removal] Integer の Integer(int)は推奨されておらず、削除用にマークされています
        Integer i = new Integer(1);
                    ^
警告 1 個
```

JDK 16 以降の javac コマンドでコンパイルしたときに警告が表示されることで、アプリケーションで該当する API を使用しているかどうかを確認できます。JDK の移行に伴いアプリケーションの再コンパイルを行わない場合、または、コンパイル時の警告を抑制している場合は、既存のアプリケーションのバイナリに対して JDK 16 以降の jdeprscan コマンドを利用することで確認できます。
jdeprscan コマンドの使い方については「jdeprscan コマンド」をご参照ください。

2. javac コマンドでのコンパイル時に、値ベースクラスをロックオブジェクトとして使用した同期が不適切であることを示す警告が表示されます。

（例）OpenJDK 21 で、該当のコードを含むプログラムをコンパイルした場合の警告

```
Test.java:4: 警告: [synchronization] 値ベース・クラスのインスタンスで同期しようとしました
        synchronized (i) {
        ^
警告 1 個
```


JDK 16 以降の javac コマンドでコンパイルしたときに警告が表示されることで、アプリケーションが該当する方法で同期を行っているかどうかを確認できます。JDK の移行に伴いアプリケーションの再コンパイルを行わない場合、または、コンパイル時の警告を抑制している場合は、JVM の起動オプションに -XX:+UnlockDiagnosticVMOptions -XX:DiagnoseSyncOnValueBasedClasses=[value]を指定し、アプリケーションを実行することで確認できます。

- value が 1 の場合：値ベースクラスをロックオブジェクトとして使用した同期が行われると、致命的なエラーとして扱われ、JVM が停止するようになります。
- value が 2 の場合：値ベースクラスをロックオブジェクトとして使用した同期が行われると、以下のようなログが出力されるようになります。

```
[0.054s][info][valuebasedclasses] Synchronizing on object 0x0000000620848df8 of klass java.lang.Integer
[0.055s][info][valuebasedclasses]       at Test.main(Test.java:4)
[0.055s][info][valuebasedclasses]       - locked <0x0000000620848df8> (a java.lang.Integer)
```

対処

"影響を受けるケース"で示した番号に応じて、以下の方法で回避・解決してください。

1. プリミティブ型ラッパークラスのインスタンス生成は、valueOf、または、オートボクシングを利用するようにコードを変更してください。以下に例を示します。

```
Integer i = new Integer(1);     // 警告発生 プリミティブ型ラッパークラスのコンストラクタ

Integer j = Integer.valueOf(1); // valueOf
Integer k = 1;                  // オートボクシング
```

2. 値ベースクラスのインスタンスをロックオブジェクトとして使用することをやめて、別途専用のロックオブジェクトを使用するようにコードを変更してください。以下に例を示します。

```
// 警告発生 値ベースクラスのインスタンスをロックオブジェクトとして使用
Integer lock = 0;
synchronized (lock) {
    // ...
}

// 専用のロックオブジェクトを使用
Integer i = 0;
Object lock = new Object();
synchronized (lock) {
    // ...
}
```

JEP 400: UTF-8 by Default

JDK 18 以降、Java 標準 API が使用するデフォルトの文字コードが、実行環境に依存しないで、原則として **UTF-8** に統一されました。

影響を受けるケース

以下のいずれかに該当する場合に影響を受けます。

1. システムプロパティ file.encoding の値が UTF-8 以外の実行環境（例：日本語 Windows のデフォルトは MS932）で、文字コードの指定の有無が異なる複数のコンストラクタを持つクラスについて、そのうち文字コードを指定しないコンストラクタで生成したインスタンスを使い、非 ASCII 文字を操作している場合。
   例えば、java.io.FileReader が持つ以下 5 つのコンストラクタのうち、引数に文字コードを指定していない、i、ii、iv のコンストラクタが該当します。
   i. FileReader(File file)
   ii. FileReader(FileDescriptor fd)
   iii. FileReader(File file, Charset charset)
   iv. FileReader(String fileName)

v. FileReader(String fileName, Charset charset)

JDK 21 の標準 API のうち、該当するコンストラクタを持つクラスの一覧は以下です。

- java.lang パッケージ
  - String（引数に byte[]を持つコンストラクタのみ影響あり）
- java.io パッケージ
  - FileReader
  - FileWriter
  - InputStreamReader
  - OutputStreamWriter
  - PrintStream
  - PrintWriter
- java.util パッケージ
  - Formatter
  - Scanner
- java.util.zip パッケージ
  - ZipFile
  - ZipOutputStream
  - ZipInputStream
- com.sun.net.httpserver パッケージ
  - BasicAuthenticator

なお、java.util.zip パッケージの ZipFile、ZipOutputStream、ZipInputStream は該当するコンストラクタを持ちますが、従来から文字コードを指定しない場合は UTF-8 が自動で選択されていたため、本修正の影響を受けません。

2. 引数に文字コードを指定しない非推奨メソッドの java.net.URLEncoder::encode(String s)や java.net.URLDecoder::decode(String s)を使用し、URLEncoder に定義されている「英数字文字」、「特殊文字」、「空白文字」以外をエンコードまたはデコードしている場合。
3. javac コマンドで UTF-8 以外の文字コードで保存されている、かつ非 ASCII 文字を含む Java ソースファイルをコンパイルしている場合。
4. java.nio.charset.Charset.forName("default")を使用している場合。

影響内容

"影響を受けるケース"で示した番号に応じて、以下の影響が発生します。

1. 文字のエンコード、または、デコードに失敗し、期待する処理が行われなくなります。
   例えば、java.io.FileReader(File file)を利用し、MS932 で保存されている、かつ、非 ASCII 文字を含むファイルを読み込む場合、JDK 18 以降では文字化けが発生するようになります。
2. エンコードまたはデコードの結果が変化する可能性があります。
3. UTF-8 以外の文字コードで保存されている、かつ非 ASCII 文字を含む Java ソースファイルを、javac コマンドでコンパイルするとコンパイルエラーが発生します。以下に例を示します。

   UTF-8 以外の文字コード（MS932）で保存されている、かつ非 ASCII 文字を含む Java ソースファイル

```
public class Test {
    public static void main(String[] args) {
        System.out.println(”こんにちは”);
    }
}
```


OpenJDK 21 で、上記の Java ソースファイルをコンパイルした場合のエラー内容

```
Test.java:3: エラー: この文字(0x82)は、エンコーディング UTF-8 にマップできません
    System.out.println(”????????”);
                       ^
...（中略）...
エラー7 個
```

4. java.nio.charset.Charset.forName("default")は、以下の実行時例外をスローします。

```
java.nio.charset.UnsupportedCharsetException: default
        at java.base/java.nio.charset.Charset.forName(Charset.java:527)
        at Test.main(Test.java:7)
```

対処

"影響を受けるケース"で示した番号に応じて、以下の方法で回避・解決してください。

1. 該当するコンストラクタを持つ API を利用する際は、文字コードを明示的に指定するようにコードを変更してください。
   JDK 17 以前と同様に、実行環境のデフォルトの文字コードで API を利用する場合は、以下のように文字コードを設定することを推奨します。

   ```
   String encoding = System.getProperty(”native.encoding”); // 実行環境のデフォルトの文字コードを取得
   Charset cs = (encoding != null) ? Charset.forName(encoding) : Charset.defaultCharset();
   var reader = new FileReader(file, cs);
   ```

   コードの変更が困難な場合は、JVM の起動オプションに-Dfile.encoding=COMPAT を指定することで、JDK 17 以前と同様に実行環境のデフォルトの文字コードで API を利用することができます。

2. 文字コードを指定しない非推奨メソッドの代わりに、文字コードを指定するメソッドを使用するようにコードを変更してください。指定する文字コードは UTF-8 を推奨します。

3. Java ソースファイルの文字コードを UTF-8 に変更してください。 Java ソースファイルの文字コードの変更が困難な場合は、コンパイル時に-encoding [文字コード]オプションでソースファイルの文字コードを指定してください。
   JDK 17 以前と同様に、実行環境のデフォルトの文字コードで Java ソースファイルをコンパイルする場合は、システムプロパティ native.encoding の値をコンパイル時の文字コードに指定してください。native.encoding の値は以下の方法で取得できます。

   ```
   java -XshowSettings:properties --version 2>&1 | grep 'native.encoding'
   ```

4. 代わりに java.nio.charset.Charset.forName("US-ASCII")を使用してください。

おわりに

本稿では、JDK 11 から JDK 21 への移行時に既存システムへ影響を与える可能性が高い 5 件の JEP（363/373/374/390/400）を解説しました。まずは既存システムに影響があるか否かを判断し、本稿を参考に対処を行ってください。