\Orchestrating a brighter world NEC

JDK17からJDK21に移行する際の注意点について

2024年2月　第1.0版

日本電気株式会社
プラットフォームテクノロジーサービス事業部門

\Orchestrating a brighter world

NECは、安全・安心・公平・効率という社会価値を創造し、
誰もが人間性を十分に発揮できる持続可能な社会の実現を目指します。

目次

1. はじめに
2. JDK17 と JDK21 の非互換項目
3. 参考情報

はじめに

本書に関する注意事項
背景と目的
表記ルール

本書に関する注意事項

1. WebOTXは日本電気株式会社の商標登録です。
2. OracleとJavaは、Oracle Corporation及びその子会社、関連会社の米国及びその他の国における商標登録です。
3. 本書の一部または全部を無断に転載することを禁じます。
4. 本書に関しては将来予告なしに変更することがあります。
5. 弊社の許可なく複製、改変することを禁じます。
6. 対処した結果の影響については、弊社は一切責任を負いません。

背景と目的

背景

2023年9月にJDK 21がリリースされました。新しいバージョンのJDKでは新機能やパフォーマンスの向上が行われているため、開発者はより効率的で堅牢なアプリケーションの構築が可能となります。

ただし、最新のJDK 21にはいくつかの非互換が存在するため、移行には潜在的な課題と対処方法を十分に理解した上で、慎重な計画と注意が必要です。

目的と対象読者

本資料では、JDK 21での変更点や非推奨となった機能について解説します。また、移行を円滑に進めるために非互換に対する対処方法を提供します。

本資料の対象読者は、JDK 21の採用を検討するアプリケーション開発者、システムアーキテクト、およびその他のJavaシステムに関わる技術者であり、Javaに関する基本的な知識を有しているものとします。

表記ルール

パスの表記

本資料では、パスの表記を”/”としています。
Windows系の場合は”/”を”¥”に読み替えてください。

インストールディレクトリの表記

プレースホルダ	説明
${JAVA_HOME}	JDKのインストールディレクトリ

JDK17 と JDK21 の非互換項目

JDK17 と JDK21 の非互換項目

1. APIの削除
2. デフォルト文字コードの ”UTF-8” 固定化
3. SHA-1署名済JARファイルの無効化
4. java.security.manager システムプロパティのデフォルト値の挙動変更
5. SunJCEプロバイダのKWPモードに関する問題の修正
6. java.io.File のカノニカルパスとパートパスのキャッシュの削除
7. java.lang.Compiler クラスと java.compiler システムプロパティの廃止
8. java.lang.ThreadGroup 、java.lang.Thread の変更
9. X509Certificate クラスのメソッドの修正
10. Kerberos 5のデフォルトの有効な暗号化タイプからDES, 3DES, RC4の削除
11. 期限が切れたルート証明書の削除
12. java.lang.reflect の内部処理の再実装

1. APIの削除(1/3)

◆ 現象

JDK21で以下のAPIが削除されました。

API	非推奨になった バージョン	削除された バージョン
java.lang.Compiler	9	21
javax.management.remote.rmi.RMIIIOPServerImpl	9	21
java.awt.color.ICC_Profile.finalize()	9	18
java.awt.image.ColorModel.finalize()	9	18
java.awt.image.IndexColorModel.finalize()	9	18
sun.security.ssl.BaseSSLSocketImpl.finalize()	-	19
java.lang.ThreadGroup.allowThreadSuspension(boolean)	1.2	21

1. APIの削除(2/3)

◆ 対処(1/2)

クラス	対応
`java.lang.Compiler`	Javaからネイティブ・コードへのコンパイラをサポートするクラスです。 詳細はP26を参照してください。
`javax.management.remote.rmi.` `RMIIIOPServerImpl`	IIOP経由でエクスポートされるRMIServerImplであり、IIOP経由でエクスポートされたRMIオブジェクトとしてクライアント接続を実装するためのクラスです。 JavaではIIOPプロトコルのサポート廃止が段階的に進められています。サポート廃止の一環でJDK9からこのクラスは非推奨となり、コンストラクタを呼び出すと`java.lang.UnsupportedOperationException`が発生するようになっています。さらにJDK 21では、このクラスは完全に削除され、コンパイルや実行することができなくなりました。 JDK 8以前からの移行でアプリケーションでRMI-IIOPを使用している場合は、`javax.management.remote.rmi.RMIJRMPServerImpl` などのIIOPプロトコルを使用しない別のRMI実装への移行を検討してください。

1. APIの削除(3/3)

◆ 対処(2/2)

メソッド	対応
`java.awt.color.ICC_Profile.finalize()`	該当のオブジェクトが参照されなくなったとき、そのオブジェクトに関連付けられたシステムリソースを破棄するメソッドです。 JDK17には左記の空実装のメソッドが定義されていましたが、JDK18でそれらは削除されました。 したがって `finalize` メソッドを呼び出す際は `Object` クラスの `finalize` メソッドが呼び出されます。 ただし、JDK17と同様、空実装になっているため、動作の変更や影響はありません。
`java.awt.image.ColorModel.finalize()`	
`java.awt.image.IndexColorModel.finalize()`	
`sun.security.ssl.BaseSSLSocketImpl.finalize()`	従来のJDK実装では、リソースのクローズ漏れを防ぐために `finalize` メソッドで `close` メソッドが呼び出されていましたが、GCの延長で `finalize` メソッドが呼び出されるかはJVMの実装依存であり、全ての環境で呼び出される保証はないため、JDK19で `finalize` メソッドは削除されました。 このため、`SSLSocket` 利用時にリソースのクローズ処理をGCの延長で呼び出される finalize メソッドに任せていた場合は、`try-with-resources` 文や `close` メソッドで適切にリソースをクローズする必要があります。
`java.lang.ThreadGroup` `.allowThreadSuspension(boolean)`	メモリー不足による暗黙の中断を制御するためにVMによって使用されていました。 詳細については、P28を参照してください。

2. デフォルト文字コードの ”UTF-8” 固定化(1/3)

◆ 概要

- JDK 17以前ではjavaソースコードコンパイル時とjavaプログラム実行時のデフォルト文字コードは、システム実行時にオペレーティングシステムとそのロケール情報等から動的に決定されていました。
- 多くの環境では、Linuxでは UTF-8 が、Windowsでは 日本の場合 windows-31j がデフォルト文字コードに使用されていましたが、JDK 18からは一律で UTF-8 が採用されるようになりました。

◆ 条件

下記の条件を満たす場合に影響を受ける可能性があります。

コンパイル時:

- javaソースコードが UTF-8 ではない場合
- `javac` コマンドの実行時に ”`-encoding` <文字コード>” オプションで文字コードを指定していない場合

実行時:

- デフォルト文字コードが UTF-8 以外の場合
- javaコマンドの実行時にシステムプロパティ ”`-Dfile.encoding=`<文字コード>” で文字コードを指定していない場合
- `java.io.InputStreamReader` などの文字のエンコードやデコード機能を持つAPIを、文字コードを指定せずに使用している場合

2. デフォルト文字コードの ”UTF-8” 固定化(2/3)

◆ 現象

コンパイル時:

- javaソースコードは UTF-8 でデコードされます。したがって、ソースコードが UTF-8 以外の場合、文字コードを指定せずにコンパイルするとコンパイルエラーが発生します。

実行時:

- `Charset.defaultCharset()`の戻り値は”UTF-8”になります。
- `Charset.forName(”default”)` を実行すると `UnsupportedCharsetException` がスローされます。
  `java.io.InputStreamReader` などの文字のエンコードやデコード機能を持つAPIを、文字コードを指定せずに使用した場合は、UTF-8 が採用されます。例外として、コンソールIO(`System.out` 、`System.err`)の出力時の挙動は変更されません。

2. デフォルト文字コードの ”UTF-8” 固定化(3/3)

◆ 対処

コンパイル時:

- javaソースコードを UTF-8 に変換してください。
- ソースコードを変換できない場合は ”-encoding <文字コード>” オプションでソースコードの文字コードを指定してください。あるいは ”-J-Dfile.encoding=COMPAT” オプションでJDK 17以前と同じ挙動にすることができます。

<実行例1>

```
${JAVA_HOME}/bin/javac -encoding <文字コード> <source files>
```

<実行例2>

```
${JAVA_HOME}/bin/javac -J-Dfile.encoding=COMPAT <source files>
```

実行時:

- java.io.InputStreamReader などの文字のエンコードやデコード機能を持つAPIを、文字コードを指定して使用するように書き換えてください。
- ソースコードの更新ができない場合は、”-Dfile.encoding=COMPAT” システムプロパティでJDK 17以前と同じ挙動にすることができます。

<実行例>

```
${JAVA_HOME}/bin/java -Dfile.encoding=COMPAT <source files>
```

3.　SHA-1署名済JARファイルの無効化(1/2)

◆ 条件

以下の全ての条件を満たす場合に影響を受ける可能性があります。

- ✓ SHA-1アルゴリズムで署名された JAR ファイルを利用している場合
- ✓ 2019年1月1日以降にタイムスタンプされた JAR ファイルを利用している場合

◆ 現象

- ■ JDK 21およびJDK 17.0.5、11.0.17、8u351 以上では、SHA-1アルゴリズムで署名された JAR ファイルはデフォルトで制限され、署名されていないものとして扱われます。
  したがって、セキュリティ設定によっては、アプリケーションの実行不可や、リソースへのアクセス制限などが発生する場合があります。

  例外として、2019年1月1日以前にタイムスタンプされた JAR は制限されません。しかし、この例外は将来のJDKリリースで削除される可能性があります。

3.　SHA-1署名済JARファイルの無効化(2/2)

- 対処
  - SHA-1よりも強固なアルゴリズムで JAR ファイルを再署名することを検討してください。
    また、自己責任で java.security ファイルを修正する(以下を参照)ことで、上記の制限を解除することができます。

    ”{JAVA_HOME}/conf/security/java.security” ファイルの以下の部分を削除することで、制限を解除できます。
    - ”SHA1 usage SignedJAR & denyAfter 2019-01-01”
    - ”SHA1 denyAfter 2019-01-01”

4. java.security.manager システムプロパティのデフォルト値の挙動変更(1/3)

◆ 条件

以下のすべての条件を満たす場合に影響を受ける可能性があります。

- ✓ java.lang.System クラスの setSecurityManager() メソッドを利用している場合
- ✓ setSecurityManager() メソッドの引数に、null 以外を指定している場合
- ✓ コマンドラインで java.security.manager システムプロパティを設定していない場合

◆ 現象

- ■ java起動時に、java.security.manager システムプロパティのデフォルト値(null)の場合、System.setSecurityManager(SecurityManager) を null 以外の引数で実行した際の挙動が、java.security.manager システムプロパティに disallow を指定した時と同様になりました。

  したがって、java起動時に java.security.manager システムプロパティの設定をしない(null の)場合、javaの実行時に System.setSecurityManager() メソッドによる、動的なセキュリティマネージャの設定をする際、UnsupportedOperationException をスローします。

4. java.security.manager システムプロパティのデフォルト値の挙動変更(2/3)

- 対処
  - 従来と同様、動的にセキュリティマネージャの設定を行う場合は、java起動時に、コマンドラインで java.security.managerシステムプロパティの値に allow を指定してください。
    ただし、セキュリティマネージャは、将来のバージョンで削除される可能性があるため、セキュリティマネージャを利用している場合は、別の手法を検討してください。
- 実行時の挙動確認

ソースコード例

```
try{
      // Security Manager object
      SecurityManager sm = new SecurityManager();

      System.setSecurityManager(sm);
      System.out.println("success");

} catch (Exception e){
      e.printStackTrace();
      System.out.println("false");
}
```

4. java.security.manager システムプロパティのデフォルト値の挙動変更(3/3)

JDK17:プロパティがnullの場合

```
${JAVA_HOME}/bin/java <your app>
WARNING: A terminally deprecated method in java.lang.System has been called
WARNING: System::setSecurityManager has been called by <your app> (file:/ <your app dir> /)
WARNING: Please consider reporting this to the maintainers of <your app>
WARNING: System::setSecurityManager will be removed in a future release
success
```

JDK20:プロパティがnullの場合

```
${JAVA_HOME}/bin/java <your app>
java.lang.UnsupportedOperationException: The Security Manager is deprecated and will be removed in a future
release
        at java.base/java.lang.System.setSecurityManager(System.java:429)
        at <your app>.main(<your app>.java :◯)
false
```

対処
JDK20:プロパティにallowを指定する場合

```
${JAVA_HOME}/bin/java -Djava.security.manager=allow <your app>
WARNING: A terminally deprecated method in java.lang.System has been called
WARNING: System::setSecurityManager has been called by <your app> (file:/ <your app dir> /)
WARNING: Please consider reporting this to the maintainers of <your app>
WARNING: System::setSecurityManager will be removed in a future release
success
```

5. SunJCEプロバイダのKWPモードに関する問題の修正(1/4)

◆ 条件

以下の全ての条件を満たす場合に影響を受ける可能性があります。

- ✓ `javax.crypto.Cipher` クラスのインスタンスを作成する際に、`getInstance()` メソッドを利用している場合
- ✓ `getInstance()` メソッドの引数に関して下記を満たす場合
  1. プロバイダ: ”SunJCE”
  2. アルゴリズム: ”AESWrapPad” or ”AES/KWP/NoPadding”
- ✓ `Cipher.init()` メソッドでiv(初期化ベクトル)を指定しており、その値が ”0xA65959A6” 以外の場合

◆ 現象

- ■ KWPモードおよび ”NoPadding” でのAES暗号を行う際(”AESWrapPad” 、”AES/KWP/NoPadding”)は、IV(初期化ベクトル)に ”0xA65959A6” (デフォルト値)以外を指定することができなくなりました。
- ■ ”KWP” モードおよび ”NoPadding” でのAES暗号を行う際、IV(初期化ベクトル)に ”0xA65959A6” (デフォルト値)以外を指定すると、`InvalidAlgorithmParameterException` が出力されます。
- ■ 既に別のIVで暗号化しているデータは、復号ができなくなる可能性があります。

5. SunJCEプロバイダのKWPモードに関する問題の修正(2/4)

- 対処
  - ”KWP” モードおよび ”NoPadding” でのAES暗号を行う際は、IV(初期化ベクトル)に ”0xA65959A6” を指定してください。またはIVに値を指定しないでください。
  - 既に別のIVで暗号化を行っているデータは、一度復号して、新たなIVで暗号化してください。

5. SunJCEプロバイダのKWPモードに関する問題の修正(3/4)

◆ 初期化ベクトルごとの確認例

比較するIV群

```
static final byte[] KW_IV_VALID = {   // 0xA65959A6
        (byte) 0xA6, (byte) 0x59, (byte) 0x59, (byte) 0xA6,};
static final byte[] KW_IV_INVALID = { // 0xA65959AA
        (byte) 0xA6, (byte) 0x59, (byte) 0x59, (byte) 0xAA,};
static final byte[] KW_IV_INVALID_LEN = { // 0xA65959A6AA
        (byte) 0xA6, (byte) 0x59, (byte) 0x59, (byte) 0xA6, (byte) 0xAA,};
```

実行部

```
HexFormat hf = HexFormat.of().withUpperCase();
String[] algos = new String[] {"AESWrapPad","AES/KWP/NoPadding"};
for (byte[] iv : new byte[][]{KW_IV_VALID, KW_IV_INVALID, KW_IV_INVALID_LEN}) {
    for (String algo : algos) {
        System.out.println();
        System.out.println("algo = " + algo);
        System.out.println("iv =   " + hf.formatHex(iv));
        try {
          KeyGenerator keyGen = KeyGenerator.getInstance("AES");
          keyGen.init(128);
          SecretKey key = keyGen.generateKey();
          Cipher cipher = Cipher.getInstance(algo, "SunJCE");
          cipher.init(Cipher.WRAP_MODE, key, new IvParameterSpec(iv));
          byte[] wrappedKey = cipher.wrap(key);
          System.out.println("wrappedKey =" + hf.formatHex(wrappedKey));
        } catch(Exception e) {
          System.out.println("Error : " + e.getMessage());
        }
    }}
```

5. SunJCEプロバイダのKWPモードに関する問題の修正(4/4)

◆ 実行結果

JDK17

```
algo = AESWrapPad
iv =   A65959A6
wrappedKey =75A236C15F992145D25E171B72D3943DC93228B7C2FEB7AF

algo = AES/KWP/NoPadding
iv =   A65959A6
wrappedKey =2836411EC9FCE1CDF1EF3E66077E9220EF580DD825A4543C

algo = AESWrapPad
iv =   A65959AA
wrappedKey =49FAB1E295B46D370A0A36D65D9B30BF18C5A4ED051A441C

algo = AES/KWP/NoPadding
iv =   A65959AA
wrappedKey =F16536280536D47B15ABDEC7751608A1EDC867762AAAEDDA

algo = AESWrapPad
iv =   A65959A6AA
Error : Invalid IV length

algo = AES/KWP/NoPadding
iv =   A65959A6AA
Error : Invalid IV length
```

JDK21

```
algo = AESWrapPad
iv =   A65959A6
wrappedKey =2061A130FE1D9893E21D18EF66F655E15E2C2050CEDA3D0C

algo = AES/KWP/NoPadding
iv =   A65959A6
wrappedKey =BD72EC0503A5B5688EA6F3627999193971A9A41F56B23FD2

algo = AESWrapPad
iv =   A65959AA
Error : Invalid IV, got 0xA65959AA instead of 0xA65959A6

algo = AES/KWP/NoPadding
iv =   A65959AA
Error : Invalid IV, got 0xA65959AA instead of 0xA65959A6

algo = AESWrapPad
iv =   A65959A6AA
Error : Invalid IV, got 0xA65959A6AA instead of 0xA65959A6

algo = AES/KWP/NoPadding
iv =   A65959A6AA
Error : Invalid IV, got 0xA65959A6AA instead of 0xA65959A6
```

6. java.io.File のカノニカルパスとパートパスのキャッシュの削除

- 条件

  以下のすべての条件を満たす場合に影響を受ける可能性があります。

  - ✓ sun.io.useCanonCaches 、sun.io.useCanonPrefixCache システムプロパティを設定している場合
  - ✓ シンボリックリンクを利用している場合

- 現象
  - java.io.File クラスの内部実装においてカノニカルパスとパートパスがキャッシュされなくなりました。
  - 内部キャッシュの挙動を指定するシステムプロパティ sun.io.useCanonCaches と、sun.io.useCanonPrefixCache が廃止されました。

    システムプロパティ sun.io.useCanonCaches および sun.io.useCanonPrefixCache を設定しても、それらは何の効果も発揮しません。
  - パスがキャッシュされなくなったことにより、Fileクラスにシンボリックリンクを指定し、かつ運用中にシンボリックリンクのリンク先が変更された場合の挙動が以前と異なる場合があります。
- 対処
  - 代替方法はありません。
  - キャッシュに保存されたパスに依存した実装を行っている場合は、正しいパスによる実装に変更してください。

7. java.lang.Compiler クラスと java.compiler システムプロパティの廃止(1/2)

◆ 条件

以下のいずれかの条件を満たす場合は、影響を受ける可能性があります。

- ✓ 実行時に java.lang.Compiler クラスを参照している場合
- ✓ java起動時に java.compiler システムプロパティを設定して実行している場合

◆ 現象(1/2)

Javaは実行時のJITコンパイラの動作を制御する仕組みとして以下の2つの機能を提供していましたが、
近年はJITコンパイラの機能の幅が広がりこの仕組みでは対応できなくなったため、機能ごと削除されました。

削除された機能	概要
java.lang.Compilerクラス	JITコンパイラの動作を制御
java.compilerシステムプロパティ	JITコンパイラを切り替える仕組み

- ■ java.lang.Compiler クラスを参照していた場合、コンパイルエラーが発生します。
  また、実行時に java.lang.Compiler クラスを参照すると ClassNotFoundException がスローされます。

7. java.lang.Compiler クラスと java.compiler システムプロパティの廃止(2/2)

◆ 現象(2/2)

コマンドラインで java.compiler システムプロパティを設定して実行すると、下記のシステムプロパティが廃止されたことを示す警告が表示されるようになりました。

```
Java HotSpot(TM) 64-Bit Server VM warning: The java.compiler system property is obsolete and no longer supported, use -Xint
```

◆ 対処

- java.lang.Compiler クラスを用いた実装をしないでください。
- java.lang.Compiler クラスをユーザアプリから参照している場合は、参照しない実装にしてください。
- ”-Djava.compiler=none”を指定してJITコンパイラを使用しないインタプリタモードを有効にしていた場合は、”-Xint”を指定してください。
- ”-Djava.compiler=<コンパイラ名>”を指定してJavaVM固有のJITコンパイラを有効化していた場合は、そのJavaVMベンダのドキュメントを参照して代替オプションを指定してください。

8. java.lang.ThreadGroup java.lang.Thread の変更(1/4)

◆ 現象(1/2)

下記のメソッドの動作が変更、削除されました。

java.lang.Thread	
suspend()	resume()
stop()	

java.lang.ThreadGroup	
suspend()	resume()
stop()	destroy()
isDestroyed()	setDaemon()
isDaemon()	allowThreadSuspension()

8. java.lang.ThreadGroup java.lang.Thread の変更(2/4)

◆ 現象(2/2)

下記のメソッドの動作が変更、削除されました。

クラス	メソッド	変更前の動作	変更後の動作
• java.lang.ThreadGroup • java.lang.Thread	suspend()	スレッドの中断をします。	UnsupportedOperationException をスローします。
	resume()	中断されたスレッドを再開します。	
	stop()	スレッドを強制的に停止させます。	
• java.lang.ThreadGroup	destroy()	スレッド・グループとそのサブグループの全てを破棄します。	何もしません。
	isDestroyed()	スレッド・グループが破棄されたかどうかを判定します。	false を返します。
	setDaemon()	スレッド・グループのデーモンの状態を変更します。	JDK17以前と同様、ThreadGroup の daemonステータスを設定しますが、そのステータスは何も効果を示しません。
	isDaemon()	スレッド・グループがデーモン・スレッド・グループであるかどうかを判定します。	JDK17以前と同様、ThreadGroup の daemonステータスを取得しますが、そのステータスは何も効果を示しません。
	allowThreadSuspension()	メモリー不足による暗黙の中断を制御するためにVMによって使用されます。	削除されました。このメソッドを参照している場合、コンパイルエラーが発生します。

8. java.lang.ThreadGroup java.lang.Thread の変更(3/4)

◆ 対処(1/2)

クラス	メソッド	対処方法
• java.lang.ThreadGroup • java.lang.Thread	• suspend() • resume()	例えば次のような方法で、suspend()/resume()のメカニズムを独自に実装してください。 ① スレッドの期待する状態を示す変数を用意し、各スレッドからその変数をポーリングさせます。 ② スレッドを一時停止させる場合は、変数に中止を示す値を設定し、各スレッドを変数が中止になっていたら Object.wait() を使用して待機させます。 ③ スレッドを再開させる場合は、変数に再開を示す値を設定し、Object.notify() / Object.notifyAll() を使用してスレッドの待機を解除します。

8. java.lang.ThreadGroup java.lang.Thread の変更(4/4)

◆ 対処(2/2)

クラス	メソッド	対処方法
• java.lang.ThreadGroup • java.lang.Thread	• stop()	例えば次のような方法で、stop() のメカニズムを独自に実装してください。 ①ターゲットスレッドの実行停止を示す何らかの変数を用意し、その変数に基づいてスレッドの停止を行います。 ② interrupt() メソッドを使用します。
• java.lang.ThreadGroup	• destroy() • isDestroyed() • setDaemon() • isDaemon() • allowThreadSuspension()	ThreadGroup クラスは非推奨のため、類似クラスである ThreadPoolExecutor クラスを使用した実装に変更してください。

9. X509Certificate クラスのメソッドの修正

◆ 現象

- java.security.cert.X509Certificate クラスの下記メソッドにおいて、エンコード不良や無効な値が含まれている場合、CertificateParsingException をスローする仕様でしたが、正常に動作せず、”null” を返していました。今回の修正で ”null” を返すのではなく、正しく CertificateParsingException をスローするようになりました。
  - getSubjectAlternativeNames()
  - getIssuerAlternativeNames()
  - getExtendedKeyUsage()

  実行結果に CertificateParsingException のスローを想定せず、返り値に ”null” を期待した実装の場合は、正常に動作しない可能性があります。

◆ 対処

- CertificateParsingException を期待した実装に変更してください。

10. Kerberos 5のデフォルトの有効な暗号化タイプから DES, 3DES, RC4の削除

- 条件
  Kerberos 5のJava Generic Security Services (Java GSS)のセキュリティ機能を使用しており、
  DES, 3DES, RC4いずれかの暗号化タイプを使用している場合に影響を受ける可能性があります。

- 現象
  Kerberos 5 GSS-APIで有効化する暗号化タイプを制御するには、krb5.confの`[libdefaults]`セクションの次の2つの設定を使用します。
  - `permitted_enctypes`：有効な暗号化タイプの一覧
  - `allow_weak_crypto` ：permitted_enctypesからDESやRC4のような弱い暗号をフィルタリングするための bool型のフラグ

  JDK 17以前では、`permitted_enctypes`のデフォルト値には強力な暗号化タイプと脆弱な暗号化タイプの両方が含まれており、`allow_weak_crypto`のデフォルト値はfalseでした。このため、従来は例えば脆弱な暗号化タイプからDESだけを有効化するには`allow_weak_crypto`に`true`を指定するだけでも動作しましたが、この設定はDES以外の脆弱な暗号化タイプも有効化するため推奨されていませんでした。
  このような不適切な設定による運用を回避するために、JDK 21では、`permitted_enctypes`のデフォルト値から脆弱な暗号化タイプであるDES, 3DES, RC4が除外されました。

- 対処
  DES, 3DES, RC4などの脆弱な暗号化タイプを使用する場合は、`permitted_enctypes`に使用を許可する暗号化タイプのリストを設定した上で、`allow_weak_crypto`に`true`を設定してください。

11. 期限が切れたルート証明書の削除

◆ 現象

以下の認証局(表記:エイリアス名)が cacerts キーストアから削除されました。
この影響で削除された証明書を使用する場合は、予め証明書を用意する必要があります。

JDK18で削除されたルート証明書	JDK21で削除されたルート証明書
identrustdstx3	secomscrootca1
globalsignr2ca	

◆ 対処

信頼する認証局から証明書を取得し、トラストストアに格納してください。

12. java.lang.reflect の内部処理の再実装(1/2)

◆ 条件

以下の条件を満たす場合、影響を受ける可能性があります。

1. 既存の実装に高度に依存し、文書化がなされていない側面に依存する実装の場合
2. スタックメモリが少ない場合

◆ 現象

1. java.lang.reflect が再実装されたため、既存の reflect 内部の実装が一部変化しました。
   したがって、APIドキュメントで文書化されていないAPIに依存した実装を行っていた場合、正常に動作しない等の影響を受ける可能性があります。

2. JDK18における再実装では、一部の利用パターンにおいて以前の java.lang.reflect の実装より多くのリソースを消費する可能性があり、より多くのスタック領域が必要となります。そのため、JDK17以前では問題ない場合でも、JDK18以降の場合 StackOverflowError が発生する可能性があります。また、クラスの初期化中に StackOverflowError がスローされた場合は、NoClassDefFoundError が発生する可能性があります。

12. java.lang.reflect の内部処理の再実装(2/2)

- 対処
  - ”-Djdk.reflect.useDirectMethodHandle=false”を使用することで、JDK17以前の java.lang.reflect の実装を有効にすることができます。
  - 既存の実装に高度に依存し、文書化がなされていない側面に依存する実装は避けてください。
  - StackOverflowError が発生する場合は、スタックメモリを増加することで解消する可能性があります。

参考情報

参考情報
改版履歴

参考情報

- 「Java Platform Standard Edition ～Oracle JDK 移行ガイド リリース21～」
  - https://docs.oracle.com/javase/jp/21/migrate/index.html#Java-Platform-Standard-Edition

改版履歴

版数	発行日	改版内容
1.0	2024/2/29	新規作成


Orchestrating a brighter world
NEC