Orchestrating a brighter world

NEC

JDK 7からJDK 8に移行する際の注意点

2015年6月1日　第1.1版

日本電気株式会社
クラウドプラットフォーム事業部

目次

1. はじめに
2. JDK 7とJDK 8の非互換項目
3. JDK 8のチューニング
4. 参照URL

はじめに

はじめに

背景

2015年4月にOracle Java SE Development Kit 7 (JDK 7)がEOL (End Of Life)を迎えたことを機に、これからはJDK 8を適用したシステムが増加すると過去の経験から想定しています。それに合わせて弊社では、翌5月にJDK 8に対応したWebOTX Application Server V9.3をリリースしました。

有償サポート版のJava SE Advancedは、今後もJDK 7自体のメンテナンスは継続します。しかし、無償で入手できるパブリック・アップデート版はEOL時点のもので強化が終了します。そのため、今後JDKに不具合が検出されたとしても、それを改修した無償版が公開されることはありません。

このことからJDK 7で動作するシステムには速やかに JDK 8に移行することが求められます。とは言えJDK 7と8の間では完全に互換性が保たれてない場合があり、このことが移行を躊躇する要因の1つとなっています。

目的と対象読者

本資料では、新たにJDK 8の使用を試みる利用者に対して、アプリケーションの移行や運用管理における注意点について記載しています。特に、JDK 8をサポートしたWebOTX Application Server V9.3以降を導入される方にとって、JDK 7環境で動作していたアプリケーションをJDK 8に移行させる際に有用です。

前提条件

本資料は作成した2015年6月時点で最新のJDKアップデートバージョン、JDK 8 Update 45を対象としています。その後にリリースされたアップデートバージョンで動作が変わる可能性がありますがご了承ください。

用語定義

パスの定義

本資料では、パスの表記を「¥」としています。UNIX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
${AS_INSTALL}	WebOTXのインストールディレクトリ
${domain}	WebOTXのドメインディレクトリ
${JAVA_HOME}	JDKのインストールディレクトリ

JDK 7とJDK 8の非互換項目

非互換項目

JDK 7とJDK 8の非互換項目は以下のように大きく3パターンに分類できます。

コンパイルできない

- DatagramPacketコンストラクタの仕様変更 (P.8)
- Java言語仕様15.21項の厳格化 (P.9)
- ジェネリクスが指定された型とRAW型の型チェックの強化 (P.11)

実行時に例外が発生

- Proxyクラスの仕様変更 ~ 1 ~ (P.13)
- Proxyクラスの仕様変更 ~ 2 ~ (P.17)
- Collectionインタフェースの仕様厳格化 (P.19)
- ソケットに割り当てるポート番号の範囲に関する仕様変更 (P.20)
- 制限付きパッケージの追加 (P.22)

実行結果が異なる

- MBean、およびMXBeanインタフェースに関する仕様厳格化 (P.24)
- NumberFormat、およびDecimalFormatの丸め動作の修正 (P.26)
- WWW-Authenticateレスポンス・ヘッダの仕様厳格化 (P.27)
- アノテーションに関する動作変更 (P.28)
- BigDecimal#stripTrailingZeros()の戻り値の変更 (P.30)
- LocaleServiceProvider#isSupportedLocale()の追加 (P.32)
- Windowsにおけるuser.homeシステムプロパティ取得方法の変更 (P.34)

DatagramPacket コンストラクタの仕様変更

コンパイルできない

◆ 条件

- アプリケーション内でjava.net.DatagramPacketクラスを使用している場合

◆ 現象

- JDK 7では、コンストラクタの一部でjava.net.SocketException例外がスローされますが、JDK 8ではスローされなくなりました。
- 移行後の再コンパイル時に、この例外をcatchしている箇所でコンパイルに失敗します。

```
try {
        DatagramPacket a = new DatagramPacket(buf, length, address);
} catch (SocketException e) {
}
```

◆ 対処

- java.net.SocketException例外、またはそのスーパークラスのjava.io.IOException例外のcatch節を削除した後にコンパイルしてください。

Java 言語仕様 15.21 項の厳格化(1/2)

コンパイルできない

◆ 条件

- アプリケーション内で型が異なるオブジェクトに対して「==」、または「!=」を使用している場合

  具体的には以下の比較以外を行っている場合。

  - ✓ 数値の型 (Integer と int など)
  - ✓ Boolean と boolean
  - ✓ 参照型と null

◆ 現象

- JDK 7では、下記のような異なるオブジェクトに対して「==」、または「!=」を使用するとfalseが返却されましたが、JDK 8では このような比較を許容しなくなりました。

  そのため、移行後の再コンパイル時に比較箇所でコンパイルに失敗します。

```
boolean bt = true;
Object obj = new Integer(1);
System.out.println(bt == obj);
```

Java 言語仕様 15.21 項の厳格化(2/2)

コンパイルできない

◆ 対処

- 正しい型で比較するよう、アプリケーションを改修してください。
  または、下記のようにキャストにより java.lang.ClassCastException 例外をスローさせ、この例外を捕獲する catch 節の中で false を返すようにアプリケーションを改修してください。

```
boolean bt = true;
Object obj = new Integer(1);
        try {
                System.out.println(bt == (Boolean)obj);
        } catch (ClassCastException cce){
                boolean bf = false;
                System.out.println(bf);
        }
```

ジェネリクスが指定された型とRAW型の型チェックの強化(1/2)

コンパイルできない

◆ 条件

- アプリケーション内で下記のようにジェネリクスを使用しており、JDK 7では警告付きでコンパイルされていた場合

◆ 現象

- JDK 7では警告付きでコンパイルされていましたが、JDK 8ではコンパイルに失敗します。

```
public static void main(String[] args) {
        Baz baz = new Baz();
        bar(baz);
}
static class Baz<T> {
        public static List<Baz<Object>> sampleMethod(Baz<Object> param) {
                List<Baz<Object>> c = new ArrayList<Baz<Object>>();
                c.add(param);
                return c;
        }
}
private static void bar(Baz arg) {
        Baz element = Baz.sampleMethod(arg).get(0);
}
```

⇦

コンパイル結果(例)

エラー: 不適合な型: ObjectをBazに変換できません:
Baz element = Baz.sampleMethod(arg).get(0);

ジェネリクスが指定された型とRAW型の型チェックの強化(2/2)

コンパイルできない

◆ 対処

- 下記のようにRAW型の部分に型を指定してください。

```
public static void main(String[] args) {
        Baz baz = new Baz();
        bar(baz);
}

static class Baz<T> {
        public static List<Baz<Object>> sampleMethod(Baz<Object> param) {
                List<Baz<Object>> c = new ArrayList<Baz<Object>>();
                c.add(param);
                return c;
        }
}

private static void bar(Baz<Object> arg) {
        Baz element = Baz.sampleMethod(arg).get(0);
}
```

Proxy クラスの仕様変更　~1~ (1/4)

実行時に例外が発生

◆ 条件

- java.lang.reflect.Proxy をスーパークラスとする動的プロキシクラスが public でないインタフェースを実装する場合、さらに、そのインタフェースと同じパッケージ内に呼び出しもとが存在しない場合

◆ 現象

- JDK 7では、この動的プロキシクラスの可視性は public でしたが、JDK 8では public ではなくなります。そのため、インタフェースと呼び出し元が同じパッケージ内に配置されていない場合に java.lang.IllegalAccessException 例外が発生します。

```
Exception in thread "main" java.lang.RuntimeException: java.lang.IllegalAccessException: Class sample2.MyProxy can not access a member of class sample1.$Proxy0with modifiers "public"
        at sample2.MyProxy.createProxy(MyProxy.java:36)
        at sample2.MyProxy.<init>(MyProxy.java:13)
        at sample1.MyClass.main(MyClass.java:14)
```

Proxy クラスの仕様変更 ~1~ (2/4)

実行時に例外が発生

```
package sample1;

interface MyInterface {
        String sayHello(String name);
}
```

パッケージ構成


```
package sample1;

class MyClass implements MyInterface{

        public String sayHello( String name ) {
                System.out.println("hello! "+name);
                return "hello!";
        }

        public static void main(String[] args) throws Exception{
                MyProxy p = new MyProxy(MyInterface.class);
                p.getProxyObject();
        }
}
```

Proxy クラスの仕様変更 ~1~ (3/4)

実行時に例外が発生

```
package sample2;

public class MyProxy implements InvocationHandler{
	private Object proxy;
	private Class<?> targetClass = null;

	public MyProxy(Class<?> target) {
		createProxy(target);
	}
	public Object getProxyObject(){
		return proxy;
	}
	private void createProxy(Class<?> target) {
		targetClass = target
		try {
			Class<?> proxyClass = Proxy.getProxyClass(target.getClassLoader(), new Class[] { target });
			Class h = InvocationHandler.class;
			Constructor constructor = proxyClass.getConstructor(h);
			proxy = constructor.newInstance(new Object[] { this });
		  } catch (Exception e) {
		    throw new RuntimeException(e);
	}
}
```

Proxy クラスの仕様変更 ~1~ (4/4)

実行時に例外が発生

◆ 対処

- Constructor.setAccessible(true) により accessible フラグを true にしてください。…(a)
  または、Proxy#newProxyInstance() を使用してインスタンスを生成してください。…(b)

```
private void createProxy(Class<?> target) {
        targetClass = target
        try {
            Class<?> proxyClass = Proxy.getProxyClass(target.getClassLoader(), new Class[] { target });
            Class h = InvocationHandler.class;
            Constructor constructor = proxyClass.getConstructor(h);

             constructor.setAccessible(true);     •••(a)

           //  proxy = constructor.newInstance(new Object[] { this });

             proxy = Proxy.newProxyInstance(target.getClassLoader(),new Class[] { target }, this);   •••(b)

          } catch (Exception e) {
            throw new RuntimeException(e);
    }
}
```

Proxy クラスの仕様変更　~2~ (1/2)

実行時に例外が発生

◆ 条件

- アプリケーション内でコンストラクタ java.lang.reflect.Proxy(InvocationHandler h) を呼び出しており、その引数 h が null となる可能性がある場合

◆ 現象

- JDK 7では、コンストラクタ java.lang.reflect.Proxy(InvocationHandler h) 呼び出し時に h が null の場合でも例外は発生しませんでしたが、JDK 8では java.lang.NullPointerException 例外がスローされます。

```
public class ProxyTest2 extends Proxy{

        protected ProxyTest2(InvocationHandler h) {
                super(h);
         }

public static void main(String[] args) {
        InvocationHandler a = null;
        ProxyTest2 b = new ProxyTest2(a);
        }
}
```

Proxy クラスの仕様変更 ~2~ (2/2)

実行時に例外が発生

◆ 対処

- null チェックを行ってください

```
public class ProxyTest2 extends Proxy{

        protected ProxyTest2(InvocationHandler h) {
                super(h);
        }

        public static void main(String[] args) {
                InvocationHandler a = b;
                if ( b != null ) {
                        ProxyTest2 c = new ProxyTest2(b);
           }
        }
}
```

Collection インタフェースの仕様厳格化

実行時に例外が発生

◆ 条件

- アプリケーション内で Collection#removeAll(Collection<?> c) または Collection#retainAll(Collection<?> c) を使用し、引数に null が指定される可能性がある場合

◆ 現象

- JDK 7ではコレクション自体が空の場合は処理が続行されていましたが、JDK 8では java.lang.NullPointerException 例外がスローされます。

```
List<String> list = new ArrayList<String>();
list.removeAll(null);
```

◆ 対処

- null チェックを行ってください

```
List<String> list = new ArrayList<String>();
If (e !=  null) {
    list.removeAll(e);
}
```

ソケットに割り当てるポート番号の範囲に関する仕様変更(1/2)

実行時に例外が発生

◆ 条件

- セキュリティマネージャを利用するアプリケーション内でエフェメラル範囲外のポートをソケットに割り当てている場合。(Windows の場合、エフェメラルポートは規定では 49152 番から 65535 番)

◆ 現象

- JDK 7では処理が続行されていましたが、JDK 8では java.security.AccessControlException 例外が発生します。

```
Exception in thread "main" java.security.AccessControlException: access denied
("java.net.SocketPermission" "localhost:80" "listen,resolve")
        at java.security.AccessControlContext.checkPermission(AccessControlContext.java:457)
        at java.security.AccessController.checkPermission(AccessController.java:884)
        at java.lang.SecurityManager.checkPermission(SecurityManager.java:549)
        at java.lang.SecurityManager.checkListen(SecurityManager.java:1131)
        at java.net.ServerSocket.bind(ServerSocket.java:374)
```

※ JDK 7 Update 51 以降でも同様の問題が発生します。

ソケットに割り当てるポート番号の範囲に関する仕様変更(2/2)

実行時に例外が発生

◆ 対処

- ${AS_INSTALL}¥domains¥WebOTXAdmin¥configのserver.policyファイルに以下を追加

```
permission java.net.SocketPermission “localhost:80”, “listen,resolve”;
```

◆ 参考

- Derby 起動時に この現象が発生する場合は、セキュリティマネージャなしで起動するオプション「-noSecurityManager」を付けることにより回避できます。

制限付きパッケージの追加(1/2)

実行時に例外が発生

◆ 条件

- セキュリティマネージャを利用するアプリケーション内で以下のパッケージを使用している場合
  - ✓ com.sun.media.sound
  - ✓ com.sun.corba.se

◆ 現象

- JDK 7では、セキュリティマネージャを利用するアプリケーション内でこれらのパッケージを使用できていましたが、JDK 8では、これらのパッケージが制限付きパッケージに追加されたため、実行時に java.security.AccessControlException 例外が発生します。
  そのため、JDK 8では java.security.AccessControlException が発生します。

```
Exception in thread "main" java.security.AccessControlException: access denied
("java.lang.RuntimePermission" "accessClassInPackage.com.sun.media.sound")
        at java.security.AccessControlContext.checkPermission(AccessControlContext.java:457)
        at java.security.AccessController.checkPermission(AccessController.java:884)
        at java.lang.SecurityManager.checkPermission(SecurityManager.java:549)
        at java.lang.SecurityManager.checkPackageAccess(SecurityManager.java:1564)
```

制限付きパッケージの追加(2/2)

実行時に例外が発生

◆ 対処

- ${AS_INSTALL}¥domains¥WebOTXAdmin¥configのserver.policyファイルに以下を追加

```
permission “java.lang.RuntimePermission” “accessClassInPackage.<パッケージ名>”;
```

◆ 参考

- 制限付きパッケージとは、java.lang.SecurityManager#checkPackageAccess(String pkg)によりアクセス制御を受けるパッケージで、その一覧は${JAVA_HOME}¥jre¥lib¥security¥java.security で、 package.access というキーに対して列挙されています。
- アプリケーションが、制限パッケージに含まれるAPIを使用しているかどうかは、 JDK 8 以降 **jdeps** というコマンドを利用して調べることができます。

(使用例)

```
>jdeps  sample.jar
    sample.jar -> C:¥Program Files¥Java¥jdk1.8.0_45¥jre¥lib¥rt.jar
    sample (sample.jar)
       -> java.io
       -> java.lang
       -> javax.ejb
```

MBean および MXBean インタフェースに関する仕様厳格化(1/2)

実行結果が異なる

◆ 条件

- アプリケーション内で独自に作成した MBean または MXBean を使用している場合。

◆ 現象

- JDK 7 では、 public ではないインタフェースの公開が許容されていましたが、JDK 8 では仕様に厳密に従い、public でないインタフェースを公開しようとすると javax.management.NotCompliantMBeanException が発生します。

```
private static interface PrivateMBean {
        public int[] getInts();
}

private static interface PrivateMXBean {
        public int[] getInts();
}

public static class Private implements PrivateMXBean, PrivateMBean {
        public int[] getInts() {
            return new int[]{1,2,3};
        }
 }
```

MBean および MXBean インタフェースに関する仕様厳格化(2/2)

実行結果が異なる

◆ 対処

- public ではない MBean を公開しないよう、アプリケーションを改修してください。
- システムプロパティ jdk.jmx.mbeans.allowNonPublic を指定して旧互換動作をさせることも可能です。ただし、本システムプロパティは将来的に廃止されます。

```
public static interface PublicMBean {
        public int[] getInts();
}

public static interface PublicMXBean {
        public int[] getInts();
}

public static class Public implements PublicMXBean, PublicMBean {
        public int[] getInts() {
            return new int[]{1,2,3};
        }
}
```

NumberFormat および DecimalFormat の丸め動作の修正

実行結果が異なる

◆ 条件

- アプリケーション内で java.text.NumberFormat#format() あるいは java.text.DecimalFormat#format() を使用している場合。

◆ 現象

- 0.8055d という値は、以下を実行することにより JDK 7 では 0.806 となりますが、JDK 8 では 0.805 となります。

```
NumberFormat nf = java.text.NumberFormat.getInstance();
System.out.println(nf.format(0.8055d));
```

◆ 対処

- double 型ではなく BigDecimal 型を使用するようアプリケーションを改修してください。

```
BigDecimal bd = new java.math.BigDecimal(“0.8055”);
NumberFormat nf = java.text.NumberFormat.getInstance();
System.out.println(nf.format(bd));
```

WWW-Authenticate レスポンス・ヘッダの仕様厳格化

実行結果が異なる

◆ 条件

- アプリケーション内で Digest 認証を行っている場合。

◆ 現象

- JDK 7 では、下記の algorithm および qop の値が誤って引用符で囲まれていました。JDK 8 では仕様に厳密に従い、引用符で囲まれなくなります。そのため、認証先のサーバの実装が、引用符で囲まれることを想定している場合、 Digest 認証に失敗します。

```
Digest username= " foo " , realm= " ****** " , nonce= "
1359097999996:13ed87b1b78c157232d609a099bcdb6e " , nc=00000001, uri= " /****** " , response=
" b6f80b049b4b39000da79c96442e0740 " , algorithm= " MD5 " , opaque= "
3E8794E4CE80B19E5DF888D615FFBBA5 " , cnonce= "
DGKKOPAFPJKCKKBDLFECINONACKFJIFNDOGKGLIO " , qop= " auth "
```

◆ 対処

- Digest 認証に失敗する場合は、認証先のサーバを変更するか、認証先のサーバ運用者に対処を依頼してください。

アノテーションに関する動作変更(1/2)

実行結果が異なる

◆ 条件

- アプリケーションのコンパイル時にコンパイラによって合成ブリッジ・メソッドが生成される場合。さらに、合成ブリッジ・メソッドの生成対象となるメソッド、またはメソッド内のパラメータにアノテーションを使用している場合。

◆ 現象

- JDK 7 では、自動生成される合成ブリッジ・メソッドにアノテーションは付加されませんが、JDK 8 では付加されます。そのため、付加されるアノテーションによっては、実行結果に差が出る可能性があります。

◆ 参考

- 共変戻り値・・・メソッドをオーバーライドした場合に、子クラスのメソッドの戻り値型に、親クラスで定義された戻り値型のサブクラスを定義すること。
- 合成ブリッジ・メソッド・・・共変戻り値を使った場合に、コンパイラにより自動生成される、親クラスと同じ戻り値型のメソッドのこと。

アノテーションに関する動作変更(2/2)

実行結果が異なる

```
abstract class T<A, B> {
    B m(A a) {
        return null;
    }
}

class AnnotationTest extends T<Integer, Integer> {
    @MethodAnnotation
    Integer m(@ParamAnnotation Integer i) {
        return i;
    }

    public class VisibilityChange extends
AnnotationTest {
    }
}
```

※ jdk1.8.0_45 でコンパイル

クラスファイル(抜粋)

```
java.lang.Object m(java.lang.Object);
  descriptor: (Ljava/lang/Object;)Ljava/lang/Object;
  flags: ACC_BRIDGE, ACC_SYNTHETIC
  Code:
    stack=2, locals=2, args_size=2
       0: aload_0
       1: aload_1
       2: checkcast     #2                  // class java/lang/Integer
       5: invokevirtual #3                  // Method
m:(Ljava/lang/Integer;)Ljava/lang/Integer;
       8: areturn
    LineNumberTable:
      line 22: 0
  RuntimeVisibleAnnotations:
    0: #16()
  RuntimeVisibleParameterAnnotations:
    parameter 0:
      0: #18()
```

#16：MethodAnnotation　に対応

#18：ParamAnnotation　に対応

JDK 8 で
追加された箇所

◆ 対処

- 実行結果に差があり、アプリケーションの動作に影響を及ぼす場合は、アノテーションを使用しないなど、アプリケーションの改修を行ってください。

BigDecimal#stripTrailingZeros() の戻り値の変更

実行結果が異なる

◆ 条件

- アプリケーション内で「0」に相当する値に対してBigDecimal#stripTrailingZeros() を使用している場合。

◆ 現象

- JDK 7では元の値がそのまま返されましたが、JDK 8では BigDecimal.ZERO が返されます。
- 以下のコードを実行すると、JDK 7では「0.0000」が出力されますが、JDK 8では「0」が出力されます。

```
BigDecimal dec = new BigDecimal("0.0000");
System.out.println(dec.stripTrailingZeros());
```

BigDecimal#stripTrailingZeros() の戻り値の変更

実行結果が異なる

◆ 対処

- 結果の違いがアプリケーション全体に影響する場合、以下のような処理を追加してください。

```
BigDecimal dec = new BigDecimal("0.0000");
BigDecimal _dec = dec.stripTrailingZeros();

if (_dec.equals(BigDecimal.ZERO)){
        System.out.println(dec.toPlainString());
} else {
        System.out.println(_dec);
}
```

LocaleServiceProvider#isSupportedLocale() の追加(1/2)

実行結果が異なる

◆ 条件

- アプリケーション内で java.util.spi.LocaleServiceProvider を拡張したクラスを使用し、その中でJDKでサポートされていない独自のロケールを定義している場合。

```
private final static Locale YEN = new Locale("ja", "JP", "YEN");
```

◆ 現象

- JDK 8 では isSupportedLocale(Locale locale) が追加されました。このメソッドの引数に独自のロケールを指定した場合、サポートされているにもかかわらず false が返されます。

LocaleServiceProvider#isSupportedLocale() の追加(2/2)

実行結果が異なる

◆ 対処

- isSupportedLocale(Locale locale) を下記のようにオーバーライドすることで、独自ロケールが引数に指定された場合に true を返すようにしてください。

```
public boolean isSupportedLocale(Locale locale) {

        boolean sb = super.isSupportedLocale(locale);

        if ( !sb && locale.equals(YEN)){
                return true;
        } else {
                return false;
        }
}
```

Windows における user.home 取得方法の変更

実行結果が異なる

◆ 条件

- Windows Server 2008 以降において、アプリケーション内で System.getProperty(“user.home”) を使用している。さらに、以下の値が異なる場合。
  - ✓ 環境変数 %USERPROFILE% の値　　・・・(a)
  - ✓ レジストリの以下の値　　・・・(b)
    (HKEY_CURRENT_USER¥Software¥Microsoft¥Windows¥CurrentVersion¥Explorer¥Shell Folders¥Desktop)に設定されている値の一つ上のディレクトリ

◆ 現象

- JDK 7 では、 (b) の値を取得していましたが、JDK 8 では (a) の値が取得されます。

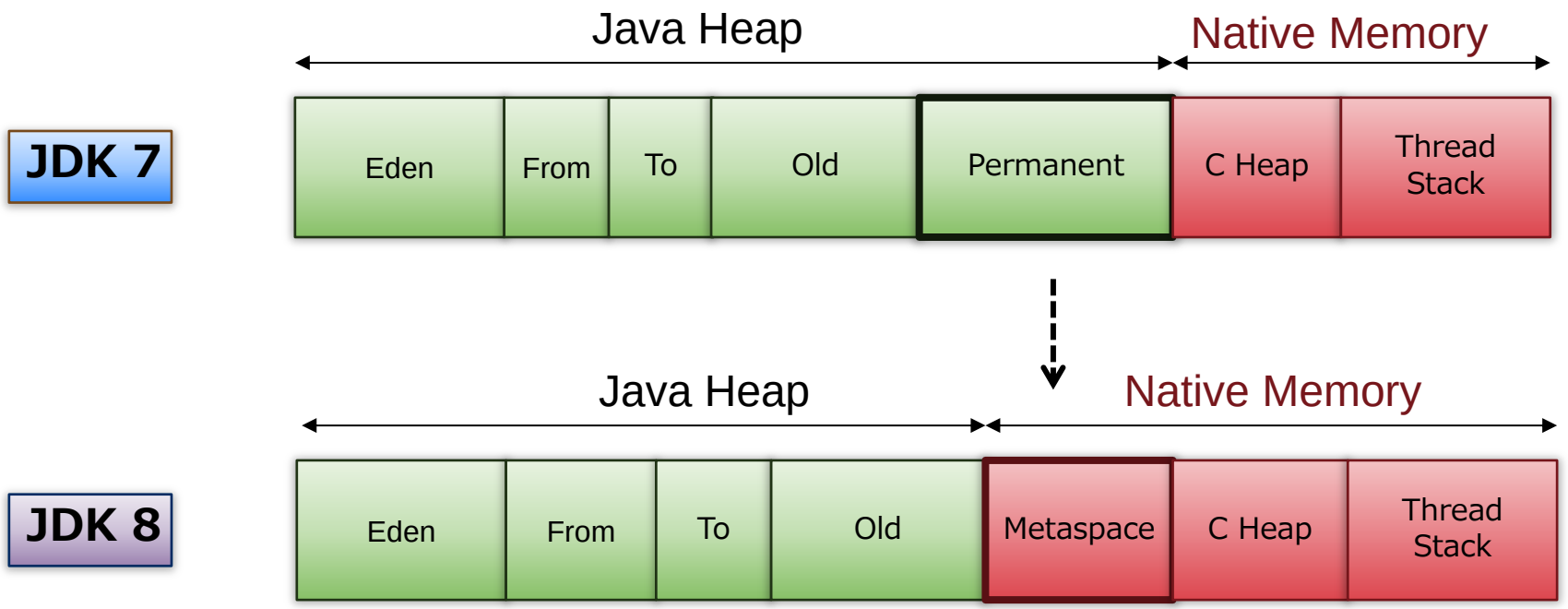
◆ 対処

- (b) の値を取得するには、下記のようにシステムプロパティに (b) の値を指定してください。
  -Duser.home=<*pathname*>

JDK 8のチューニング

Permanent 領域から Metaspace 領域へ

- **JDK 7 でJava Heap にあった Permanent 領域は、 JDK 8 で廃止されました。JDK 7 で Permanent 領域に格納されていたデータのうち、クラスに関するメタデータは JDK 8 では、新たに設けられた Metaspace と呼ばれる領域に格納されます。**



Metaspace 領域のチューニング・パラメータ

◆ **この変更に伴い、メモリ領域のチューニング・パラメータも変更になります。主要なチューニング・パラメータは下記のとおりです。**

- ✓ -XX:MaxMetaspaceSize=<NNN> (default:無制限)
- ✓ -XX:MetaspaceSize=<NNN> (default:12MB ～ 22MB程度)

◆**これらのパラメータは、通常、指定する必要はありません。**

ただし、アプリケーション内で、メモリ・リークが発生した場合、(例えば、ThreadLocal　にクラスを格納していた場合、アプリケーションが配備解除されても、スレッドは破棄されず、クラスの情報が GC 対象にならない) Metaspace は際限なく消費され、サーバのメモリ不足を引き起こす可能性があります。このような状況を未然に防ぐためには、 MaxMetaspaceSize を指定しておくことが有効です。

階層型コンパイルの有効化

◆ **JDK 7では、-XX:+TieredCompilation を付けることで有効化(既定では無効)が可能な階層型コンパイルが、JDK 8 では既定で有効になりました。**

階層型コンパイルにより、サーバの起動性能が大幅に向上。

- 階層型コンパイルとは

✓ JIT コンパイラによって行われるコンパイルにおいて、サーバコンパイラとクライアントコンパイラを組み合わせて最適化を図ったコンパイル手法のことです。

- JIT コンパイラ(Just-In-Time compiler) ・・・ バイトコードをネイティブコードに書き換えるコンパイラ。
- クライアントコンパイラ(C1) ・・・ JVMの起動後、早期にコンパイルを開始。コンパイル時間が短く、コンパイル時のメモリ消費量もそれほど多くない。
- サーバコンパイラ(C2) ・・・コンパイル時間がC1より長く、メモリ消費量も多い。しかし、C1より高速に動作するコードを生成する。

階層型コンパイルに関連したチューニング

- **JIT コンパイラにより生成されたネイティブコードは、コード・キャッシュというメモリ領域に配置されます。そのため、階層型コンパイルを行う場合、コード・キャッシュサイズの既定値では、すぐにキャッシュサイズがいっぱいになる可能性があります。いっぱいになると、JIT コンパイルが行われないため、性能が低下します。**
  - コード・キャッシュサイズがいっぱいになると、${AS_INSTALL}¥domains¥${domain}¥logs¥server.log に以下のような警告が出力されます。

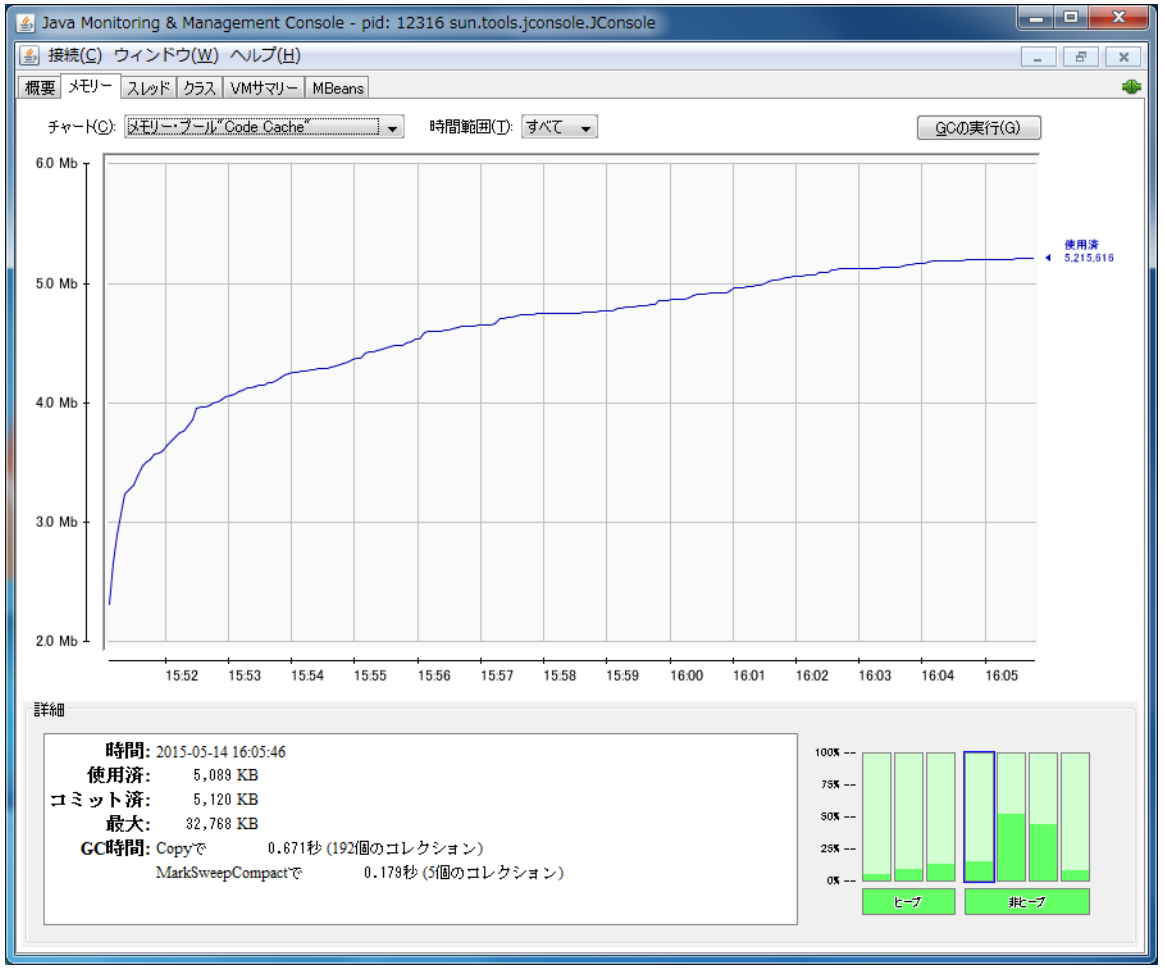
```
Java HotSpot(TM) Server VM warning: CodeCache is full. Compiler has been disabled.
Java HotSpot(TM) Server VM warning: Try increasing the code cache size using -XX:ReservedCodeCacheSize=
```

  - チューニングの指針

1. -XX:+PrintCompilation を使用して、実際にコンパイルされているメソッドを出力。
2. この出力が安定状態になるまで待つ。
3. -XX:ReservedCodeCacheSize を使用して、コード・キャッシュサイズを、試しに 2 倍に増やす。
4. コンパイルされたメソッド数が増加した場合、元のコード・キャッシュサイズが小さすぎたと見なせる。
5. 全体的なパフォーマンスの再テストを行い、コード・キャッシュサイズの変更が他の側面に悪影響を及ぼしていないかどうかを確認する。

コード・キャッシュサイズの監視

- **jconsole を使用してコード・キャッシュサイズを監視することができます。jconsole を起動し、「メモリー」パネルで「メモリー・プール ” Code Cache ”」を選択してください。**



参考URL

JDK 8の互換性ガイド

- http://www.oracle.com/technetwork/jp/java/javase/overview/8-compatibility-guide-2156366-ja.html

JDK 8リリース・ノート

- http://www.oracle.com/technetwork/jp/java/javase/overview/8train-relnotes-latest-2153846-ja.html

Java HotSpot VMコード・キャッシュについて

- http://www.oracle.com/webfolder/technetwork/jp/javamagazine/Java-JA13-Architect-evans.pdf

Java Day Tokyo 2015講演資料 「Java EEアプリケーションサーバの開発現場で見たJava SEの実際」

- http://www.oracle.co.jp/jdt2015/pdf/3-2.pdf

Orchestrating a brighter world

世界の想いを、未来へつなげる。

未来に向かい、人が生きる、豊かに生きるために欠かせないもの。
それは「安全」「安心」「効率」「公平」という価値が実現された社会です。

NECは、ネットワーク技術とコンピューティング技術をあわせ持つ
類のないインテグレーターとしてリーダーシップを発揮し、
卓越した技術とさまざまな知見やアイデアを融合することで、
世界の国々や地域の人々と協奏しながら、
明るく希望に満ちた暮らしと社会を実現し、未来につなげていきます。

Empowered by Innovation