FUJITSU

FUJITSU Software
Enterprise Application Platform V1.0.0

OpenJDKユーザーズガイド

Windows(64) / Linux(64)

B1FW-6080-01Z0(00)
2020年10月

まえがき

本書の目的

本書は、OpenJDKの利用について説明しています。

本書の読者

本書は、以下の読者を対象に書かれています。

- OpenJDKを利用する方

本書の構成

本書は次の構成になっています。

第1章 OpenJDKの概要

第2章 拡張機能

第3章 異常発生時の対応

第4章 オープンソフトウェアについて

本書の表記について

本書では、説明するうえで、次に示す名称、略称および記号を使用しています。

- マニュアル中で、自身を参照する場合は「本書」と記述します。
  本製品のマニュアル名称を記述する場合、マニュアル名の先頭の「FUJITSU Software Enterprise Application Platform V1.0.0」を省略します。

商標について

- Linux(R)は米国およびその他の国におけるLinus Torvaldsの登録商標です。
- Microsoft、Windows、およびWindows Serverは、米国 Microsoft Corporation の、米国およびその他の国における登録商標または商標です。
- OpenJDKは、Oracle America, Inc の商標です。
- OracleおよびJavaは、オラクルおよびその関連会社の登録商標です。
- そのほか、本マニュアルに記載されている会社名および製品名は、それぞれ各社の商標または登録商標です。

輸出管理規制について

本ドキュメントを輸出または第三者へ提供する場合は、お客様が居住する国および米国輸出管理関連法規等の規制をご確認のうえ、必要な手続きをおとりください。

出版年月および版数

2020年10月 初版

著作権表示

目　次

第1章 OpenJDKの概要

OpenJDK (Open Java Development Kit) は、プログラミング言語Javaの標準仕様JavaSEの参照実装です。

FUJITSU Software Enterprise Application Platformでは、OpenJDKに障害修正や拡張機能を提供しています。

1.1 概要

本製品で提供しているOpenJDKについて説明します。

OpenJDKのリビジョン

以下のリビジョンをベースに提供しています。

jdk-11.0.8-ga

OpenJDK のインストール先

OpenJDKは、以下のディレクトリーにインストールされます。

Windows64

- {本製品のインストールディレクトリー}¥openjdk¥jdk11

Linux64

- /opt/FJSVeapf/openjdk/jdk11

1.2 仕様

1.2.1 Java SEの仕様

Java SEの仕様については、以下を参照してください。

参照

JDK 11 APIドキュメント

https://docs.oracle.com/javase/jp/11/docs/api/index.html

拡張機能については、“第2章 拡張機能”を参照してください。

1.2.2 JDK ツール

JDKツールの仕様については、以下を参照してください。

参照

JDK ツールの仕様

https://docs.oracle.com/javase/jp/11/tools/

各JDKツールを使用する場合、以下に注意してください。

jcmd

- JavaプログラムがWindowsサービスとして動作している場合、jcmdのオプションとしてメインクラス名は使用できません。 Windows64
- 実行ユーザーと違うセッションのJavaプログラムに対して、正しく動作しない場合があります。対象Javaプログラムをツールと同じセッションで動作させてください。 Windows64

jconsole

- SSLを使用せずにJavaプログラムに対してjconsoleを接続した場合、以下のようなメッセージのダイアログが表示されます。

  セキュアな接続が失敗しました。非セキュアで再試行しますか。
  SSLを使用して*接続対象名*に接続できません。
  (ユーザ名およびパスワードはプレーン・テキストで送信されます。)

  接続対象名

  接続対象のJavaプログラムのプロセスID、ホスト名とポート番号など

  ローカル監視を使用して接続する場合、または 外部のネットワークと分離された安全な環境で使用する場合、[Insecure connection]ボタンを押下して、Javaプログラムに接続してください。

  安全が保証されていない環境で使用する場合、パスワードやSSLの設定を推奨します。

- 実行ユーザーと違うセッションのJavaプログラム、またはWindowsサービスとして動作するJavaプログラムに対して使用する場合、リモート監視を使用してJavaプログラムに接続してください。 Windows64
- 実行ユーザーと違うセッションのJavaプログラムに対して、正しく動作しない場合があります。対象Javaプログラムをツールと同じセッションで動作させてください。 Windows64
- コマンドを実行すると、以下のメッセージが出力される場合があります。

  Gtk-Message: xx:xx:xx.xxx: Failed to load module "canberra-gtk-module"

  本メッセージが出力された場合、以下のライブラリーをインストールしてください。

  パッケージ

  libcanberra-gtk2

  アーキテクチャー

  x86_64

- jconsoleの"ヘルプ"メニューから"オンライン・ユーザ・ガイド"を起動した場合、「存在しないページ」となります。また、"JConsoleについて"の起動後のダイアログに記載されるURLを表示した場合も、「存在しないページ」となります。

jhsdb

- 以下のモードは、サポート対象外です。
  - clhsdb
  - hsdb
  - jsnap
  - debugd
- jstackモードの"--mixedオプション"は使用できません。

jps

- オプションで、「hostname」にIPアドレスを指定しかつ、ポートを指定した場合、「protocol」または「//」の指定が必須になります。
- JavaプログラムがWindowsサービスとして動作している場合、使用できません。 Windows64

- 実行ユーザーと違うセッションのJavaプログラムに対して、正しく動作しない場合があります。対象Javaプログラムをツールと同じセッションで動作させてください。 Windows64

jstat

- JavaプログラムがWindowsサービスとして動作している場合、使用できません。 Windows64
- 実行ユーザーと違うセッションのJavaプログラムに対して、正しく動作しない場合があります。対象Javaプログラムをツールと同じセッションで動作させてください。 Windows64

1.3 ポート番号

OpenJDKのコマンドが使用するポート番号を以下に示します。以下に示すポート番号がすでに使用されている場合は、別の番号に再設定してください。

rmidコマンド

- ポート番号:1098/tcp
- 省略時:固定
- 変更可能
- ポート番号の設定箇所:オプション "-port"

rmiregistryコマンド

- ポート番号:1099/tcp
- 省略時:固定
- 変更可能
- ポート番号の設定箇所:引数 "port"

jstatdコマンド

- ポート番号:1099/tcp
- 省略時:固定
- 変更可能
- ポート番号の設定箇所:オプション "-p"

1.4 既知の問題

既知の問題はありません。

1.5 制限事項

制限事項はありません。

1.6 注意事項

JDK8からJDK11への移行

JDK11への移行は、以下の「移行ガイド」を参考にしてください。

https://docs.oracle.com/javase/jp/11/migrate/index.html

JDKツール

各ツールの注意事項は、"1.2.2 JDK ツール"を参照してください。

OSの仮想メモリー管理 Linux64

OSの仮想メモリー管理は、malloc()関数による仮想メモリー獲得要求を処理する際、プロセス内の各スレッドごとに、64メガバイトの大きさを1単位とした「malloc()用領域」を確保(リザーブ)します。

そのため、malloc()関数を実行(他の関数の延長でmalloc()関数が間接的に実行された場合も含む)するスレッドの数が多いプロセスを実行した場合、OSの仮想メモリー管理によるスレッド数に依存した「malloc()用領域」の確保により、同一のプログラムを実行するプロセスの場合でも、当該プロセスに対して確保される仮想メモリーが大きくなる場合があります。

- 確保された「malloc()用領域」のうち、実際に使用される仮想メモリー量(システム資源量)は、各スレッドのmalloc()関数で指定された要求量と、OSによる仮想メモリー管理用制御域の大きさの総和です。
- psコマンドを使用し、仮想メモリー量(VSZ)の値を参照することで確認できます。

ハードウェア資源の動的再構成機能

アプリケーションを実行するハードウェア環境によっては、「DR(Dynamic Reconfiguration)機能」のハードウェア資源の動的再構成機能が利用できる場合があります。

ハードウェア資源の動的再構成機能を利用する場合、以下の注意が必要です。

- Javaプロセスの再起動

  OSから見たプロセッサー数が、シングルプロセッサーからマルチプロセッサー、またはマルチプロセッサーからシングルプロセッサーへとCPU数が変化する場合は、Javaプロセスの停止後に構成変更をしてください。

  マルチプロセッサーからマルチプロセッサーへの変化の場合は、Javaプロセスの停止は不要です。

- 実行するJavaアプリケーションへの考慮

  システム運用中にCPU数が変化するため、以下のJava APIを使用するJavaアプリケーションは、アプリケーション実行中にCPU数が動的に変化する可能性があることを考慮して各処理を実装する必要があります。

  利用可能なCPU数に影響を受けるJavaアプリケーションは、ポーリング処理などにより処理を調整する必要があります。

  ー Java Platform API仕様:

    - java.lang.Runtime#availableProcessors()
    - java.lang.management.OperatingSystemMXBean#getAvailableProcessors()

  ー JVMTI(Java Virtual Machine Tool Interface):

    - GetAvailableProcessors()

Exception発生時に「<<no stack trace available>>」となる場合

Javaアプリケーション実行でExceptionが発生した場合に、スタックトレースが出力されず代わりに「<<no stack trace available>>」と出力される場合があります。

これはJava VM内部でExceptionを生成する際、「スタックトレース出力の情報」を格納するためのJavaヒープが不足していることになります。

このような場合、Java VMでは、java.lang.OutOfMemoryErrorを発行しないため、Javaのヒープサイズ(メモリー割り当てプールの大きさ)を大きくするなど対策をとり、Javaヒープの空きを十分に取った上でスタックトレースを確認してください。

スタックトレースについては、"3.1 スタックトレース"を参照してください。

SwingやAWTなどを使用したGUIアプリケーションについて Linux64

- GUIアプリケーションを実行すると、以下のメッセージが出力される場合があります。

```
Gtk-Message: xx:xx:xx.xxx: Failed to load module "canberra-gtk-module"
```

  本メッセージが出力された場合、以下のライブラリーをインストールしてください。

パッケージ

libcanberra-gtk2

・ GUIアプリケーションで日本語を表示する場合、以下のパッケージをインストールしてください。

パッケージ

vlgothic-fonts

第2章 拡張機能

独自で実装している機能一覧です。

- メモリー領域が確保できずにJavaVMが終了した場合のメッセージ出力強化
- JavaVM終了時のメッセージ出力機能
- hs_err_pidログのコンソール出力機能
- java.lang.System.gc()実行時におけるスタックトレース出力機能
- Windows64 不正なFileNotFoundExceptionの発生抑止
- Linux64 SctpChannel使用時のIOExceptionの発生抑止
- GC不可時のクラスヒストグラム出力抑止
- 統合ロギングフレームワークのスレッド情報の強化
- javadocコマンドが生成するURLの不具合修正

2.1 メモリー領域が確保できずにJavaVMが終了した場合のメッセージ出力強化

多量のスレッドを生成すると、ユーザー空間に多量のスタックが割り当てられます。これにより、Javaスレッド内でのユーザー空間不足が原因、または、仮想メモリー不足が原因で、メモリー領域が確保できずにJavaVMが終了した場合に、以下のメッセージを出力します。

出力されるメッセージ

Fatal Error occurred in thread *threadname*.

threadname

OutOfMemoryErrorを検出したスレッド名

2.2 JavaVM終了時のメッセージ出力機能

JavaVMが終了したときに、JavaVMが終了したことを通知するメッセージを標準出力へ出力します。

Javaアプリケーションがその制御論理として終了したこと、およびJavaVMを終了させたメソッドの実行状態を確認できるようになります。

オプション

-XX:+PrintVMExitMessage

このオプションを指定した場合に、JavaVM終了時のメッセージ出力機能が有効となります。

出力されるメッセージ

以下のJavaVMを終了させるメソッドによってJavaVMが終了していた場合、当該メソッドを実行したJavaスレッドのスタックトレース情報を標準出力へ出力します。

- java.lang.System.exit()
- java.lang.Runtime.exit()
- java.lang.Runtime.halt()

```
Thread dump at JVM_Halt(status code=1):

"main" #1 prio=5 os_prio=0 cpu=97.61ms elapsed=0.10s tid=0x00007fdd7c012800 nid=0x6b20 runnable
[0x00007fdd86433000]

   java.lang.Thread.State: RUNNABLE
```

```
        at java.lang.Shutdown.halt0(java.base@11.0.5/Native Method)

        at java.lang.Shutdown.halt(java.base@11.0.5/Shutdown.java:152)

        - locked <0x000000008ec04d10> (a java.lang.Shutdown$Lock)

        at java.lang.Shutdown.exit(java.base@11.0.5/Shutdown.java:175)

        - locked <0x000000008ec04b90> (a java.lang.Class for java.lang.Shutdown)

        at java.lang.Runtime.exit(java.base@11.0.5/Runtime.java:115)

        at java.lang.System.exit(java.base@11.0.5/System.java:1749)

            at Sample.main(Sample.java:13)


#### JavaVM terminated: OpenJDK 64-Bit Server VM (11.0.5+10-2019-11-29-Fujitsu-B01 mixed mode),
[pid=27423] TimeMillis=1575420481414 Time=Wed Dec  4 00:48:01 2019
```

対象のメソッドを実行しない場合は、JavaVM終了時にJavaVMが終了したことを通知する、以下のようなメッセージを出力します。

```
#### JavaVM terminated: OpenJDK 64-Bit Server VM (11.0.5+10-2019-11-29-Fujitsu-B01 mixed mode), [pid=28309]
TimeMillis=1575421344895 Time=Wed Dec 4 01:02:24 2019
```

2.3 hs_err_pidログのコンソール出力機能

Java VMが異常を検知したときは、hs_err_pidファイルとコンソールにサマリー部分が出力されます。hs_err_pidファイルと同様のエラー内容すべてをコンソールに出力する機能を提供します。

オプション

```
-XX:+VerboseConsoleOnFatalError
```

このオプションを指定した場合に、hs_err_pidログのコンソール出力機能が有効となります。

2.4 java.lang.System.gc()実行時におけるスタックトレース出力機能

Javaアプリケーション実行時にSystem.gc()メソッドの実行状態の確認ができるように、当該メソッドを実行したJavaスレッドのスタックトレース情報を標準出力へ出力します。

オプション

```
-XX:+PrintJavaStackAtSystemGC
```

このオプションを指定した場合に、スタックトレース出力機能が有効となります。

出力例

```
"main" #1 prio=5 os_prio=0 cpu=91.70ms elapsed=0.10s tid=0x00007f6c1c010000 nid=0x55bf runnable [0x00007f6c25523000]
java.lang.Thread.State: RUNNABLE
at java.lang.Runtime.gc(java.base@11.0.4/Native Method)
at java.lang.System.gc(java.base@11.0.4/System.java:1768)
at SystemGC.main(SystemGC.java:4)
```

2.5 不正なFileNotFoundException発生の不具合修正 Windows64

別プロセスが原因でzip形式ファイルやモジュールファイル(jarファイル等含む)にアクセスできずに不正なFileNotFoundExceptionが発生していた問題を修正し、アクセスできるようにしました。

2.6 SctpChannel使用時のIOExceptionの発生抑止 Linux64

マルチスレッドでSctpChannelを使用している時に、以下のIOExceptionが発生することを抑止します。

```
Caused by: java.io.IOException: ファイルを開きすぎです
at sun.nio.ch.ServerSocketChannelImpl.accept0(Native Method)
at sun.nio.ch.ServerSocketChannelImpl.accept(ServerSocketChannelImpl.java:XX)
```

注意

必要なパッケージ

jdk.sctpモジュール (com.sun.nio.sctpパッケージ) を使用する場合に以下のパッケージが必要です。

```
lksctp-tools-x.x.xx-x.xxx.x86_64.rpm
```

x.x.xx-x.xxx

パッケージのバージョンです。OSのバージョンによって異なります。

2.7 GC不可時のクラスヒストグラム出力抑止

クラスヒストグラムの出力時に、GCが実行できなかった場合に、警告メッセージを出力し、クラスヒストグラムの出力を抑止します。

出力されるメッセージ

```
The PrintClassHistogram operation was canceled because GC could not be run.
```

2.8 統合ロギングフレームワークのスレッド情報の強化

統合ロギングフレームワークを使用して「thread」を含むスレッド生成情報のログメッセージを出力する場合、スレッド種別を付加します。

オプション

```
-Xlog:tagx,os+thread,tagy
-Xlog:thread*
```

どちらかのオプションを指定した場合に、ログメッセージにスレッド種別を追加します。

tagxまたはtagy

-Xlogで使用可能なログ・タグです。

例

- スレッドログオプション指定例

```
-Xlog:os+thread
```

- 出力例

```
[0.085s][info][os,thread] Thread started (tid: 1332, type: type, attributes: stacksize: default, flags: CREATE_SUSPENDED
STACK_SIZE_PARAM_IS)
```

type:スレッドの種別が表示されます。例えば，以下の種別が表示されます。

- pgc
- cgc
- vm
- java
- compiler
- os

2.9 javadocコマンドが生成するURLの不具合修正

javadocコマンドでHTMLドキュメントを生成する場合、URLが誤って相対パスに変換される場合がある問題を修正し、正常なURLが生成されるようにします。

以下の「ソースコード」をjavadocコマンドで生成した場合を例として説明します。

```
/**
*本メソッドの詳細は、<a href="ftp://www.example.com/">FTP Site</a>を参照してください。
*@param p1 引数1
*/
```

問題発生時のURL（例）

```
<div>本メソッドの詳細は、<a href="../../a/ftp://www.example.com"> FTP Site</a>を参照してください。</div>
```

正常なURL（例）

```
<div>本メソッドの詳細は、<a href="ftp://www.example.com/"> FTP Site</a>を参照してください。</div>
```

第3章 異常発生時の対応

3.1 スタックトレース

Javaアプリケーションで例外(java.lang.Throwableのインスタンス)がスローされた場合などに出力されるスタックトレースは、エラーが発生するまでの経緯(メソッドの呼び出し順番)が示されています。このスタックトレースを解析すると、エラーが発生した箇所と原因を確認できます。

スタックトレースの出力先

スタックトレースの出力先は、標準エラーです。通常のJavaアプリケーションの場合は、コンソールに出力されますが、Servlet/JSP/EJBアプリケーションの場合は、ログファイル(標準出力、標準エラー出力またはJava VMの出力を格納するファイルなど)に出力されます。

スタックトレースの出力方法

Javaでスローされた例外をcatch節でキャッチし、例外のprintStackTraceメソッドを実行することで、スタックトレースを出力できます。

スローされた例外をtry-catch構文で処理するメソッドがスレッドにない場合、そのスレッドは停止され、Java VMによってスタックトレースが出力されます。

ポイント

java.lang.Throwable.printStackTrace()メソッドでスタックトレースを出力する方法

```
try {
    SampleBMPSessionRemote bmpSessionRemote = bmpSessionHome.create();
} catch(Exception e) {
    e.printStackTrace();
}
```

スタックトレースの出力形式

```
例外クラス名: エラーメッセージ
  at クラス名.メソッド名1(ソース名:行番号)     呼び出し先
  at クラス名.メソッド名2(ソース名:行番号)
            :
            :
  at クラス名.メソッド名N(ソース名:行番号)     呼び出し元
```

- 最初の1行目は、スローされた例外のクラス名とエラーメッセージです。
  エラーメッセージがない場合もあります。
- 2行目以降は、メソッドの呼び出し元(クラス名.メソッド名N)から呼び出し先(クラス名.メソッド名1)に向かって下から上に出力されます。
  2行目(クラス名.メソッド名1)が、例外をスローしたメソッドの情報です。
- "メソッド名"が"<init>"の場合、コンストラクターを示します。
- "メソッド名"が"<clinit>"の場合、static initializerを示します。
- "(ソース名:行番号)"が"(Native Method)"の場合、Javaのネイティブメソッド(.soや.dllファイル)を示します。
- クラスのコンパイル時にデバッグ情報を削除した場合、"(ソース名:行番号)"には、ソース名しか表示されなかったり、"Unknown Source"と表示されたりする場合があります。

3.2 スレッドダンプ

スレッドダンプには、Javaプロセスの各スレッドの情報(スタックトレース形式)が含まれているため、ハングアップやデッドロックなどの動作具合を調査できます。

スレッドダンプが出力される契機および出力先

Jakarta EEアプリケーション

[出力契機]

一定の条件を満たした場合、コンテナの機能により自動的に採取される場合と利用者の任意のタイミングで手動による採取があります。

- 自動採取:アプリケーションがタイムアウトまたは無応答になった場合
- 手動採取:

  Linux64 kill -QUIT [プロセスID]でJava VMに対してQUITシグナルを送り採取します。

  Windows64 jcmdコマンドで採取します。

[出力先]

標準出力、JavaVMの出力をロギングしているファイル

その他のJavaプログラム

[出力契機]

利用者の任意のタイミングで手動で採取できます。

Linux64

- ターミナルからJavaプログラムを起動した場合

  以下、どちらかの方法で採取できます。

  - [Ctrl]+[¥]キー(英語キーボードの場合バックスラッシュキー)押下
  - kill -QUIT [プロセスID]

- ターミナル以外からJavaプログラムを起動した場合

  kill -QUIT [プロセスID]で採取します。

Windows64

- コマンドプロンプトからJavaプログラムを起動した場合

  以下、どちらかの方法で採取できます。

  - [Ctrl]+[Break]キー押下
  - jcmdコマンド

- コマンドプロンプト以外からJavaプログラムを起動した場合

  jcmdコマンドで採取します。

[出力先]

コンソール(標準出力)

"-Xrs"オプションが指定されたJavaプロセスの場合、当該プロセスへ送られた[Ctrl]+[Break]キー押下またはQUITシグナルに対する動作は、OSのデフォルト動作になります。

そのため、"-Xrs"オプションを指定したJavaプロセスに対して[Ctrl]+[Break]キー押下またはQUITシグナルが送られると、当該Javaプロセスは強制終了または異常終了します。

スレッドダンプを出力する可能性があるJavaプロセスに対して、" -Xrs"オプションは指定しないでください。

ただし、Windowsでサービスとして登録されるJavaプロセスの場合は、" -Xrs"オプションを指定しない場合、ログオフ時に強制終了してしまいます。これが不都合な場合は、" -Xrs"オプションを指定してください。

3.3 クラッシュダンプ Windows64

異常を調査する場合に採取する、クラッシュダンプの採取方法を説明します。

クラッシュダンプの採取には、Windowsエラー報告「Windows Error Reporting（WER）」の機能を使用します。

次の例を参考にして、WERを設定してください。

例

WERの設定例

1. コマンドプロンプトなどで"regedit"コマンドを投入し、レジストリーエディターを起動します。
2. 「HKEY_LOCAL_MACHINE¥SOFTWARE¥Microsoft¥Windows¥Windows Error Reporting¥LocalDumps」キーを作成します。
3. LocalDumpsキーにREG_DWORD型でDumpTypeという値を作成し、「2」を設定します。

3.4 コアダンプ Linux64

コアダンプ採取のための注意事項を説明します。

コアダンプが出力されない場合の確認

- コアダンプが出力されない場合の原因として、システムリソース等の問題がまず考えられます。カレントディレクトリーの書込み権、ディスク容量、limit(1)コマンド結果を確認してください。
- ハード／オペレーティングシステムの出荷時、またはオペレーティングシステムのUpdate適用により、デフォルトではコアダンプの出力が設定されていない場合があります。"コアダンプ出力の設定方法"を参照して、コアダンプが出力されるように設定してください。

コアダンプ出力の設定方法

- コマンドで、GlassFishまたはLauncherを起動する場合

  "ulimit -c unlimited"コマンド実行後、GlassFishまたはLauncherを起動します。

- OSの自動起動でGlassFishを起動する場合

  unitファイルに以下の設定を追加してください。unitファイルでの定義方法については、「GlassFishユーザーズガイド」の「メッセージブローカのサービス化」を参照してください。

  - 記載するセクション：[Service]
  - 設定項目：LimitCORE
  - 設定値：infinity

3.5 ログ情報

hs_err_pidログ

Javaプロセスが異常終了した場合に出力されます。

デフォルト出力

あり（異常時）

デフォルトファイル名

hs_err_pid*processid*.log

processid

プロセスID

デフォルトパス

javaコマンド実行時のカレントディレクトリー

世代数

デフォルト:1、最大値:1(固定)

サイズ

実行するJavaアプリケーションにより可変(通常は数100KB～数MB程度)

レコード長

可変

ローテーション条件

なし

設定箇所

標準出力(*)へも並行して出力する場合は"-XX:+VerboseConsoleOnFatalError"を指定

*)GlassFish使用時:console.log

ガーベジコレクションログ

デフォルト出力

なし

デフォルト出力先

ー 標準出力

ー GlassFish使用時:console.log

デフォルトパス

なし

世代数

デフォルト:1、最大値: オプションで設定

サイズ

上限なし

レコード長

可変

ローテーション条件

オプションで設定

設定箇所(Java VMオプション)

-verbose:gc

ログの出力有無

-Xlog:gc*:file=*file*::filecount=*count*,filesize=*filesize*

file:出力先
count:ローテーション条件
filesize:サイズ

3.6 プロセスが消滅(異常終了)した場合

何の痕跡も残さずに突然プロセスが消滅した場合に、考えられる原因を説明します。

想定される原因:JNI処理の異常

JNI経由でJava以外の言語で開発したネイティブモジュールと連携する際、JNIの使用方法を誤ると、プロセス消滅の原因となります。

このようなときは、"-Xcheck:jni"オプションを指定して、JNI処理でメッセージが出力されないかどうかを確認してください。

JNI処理に誤りがなくても、ネイティブモジュールで異常終了またハングアップが発生すると、Javaアプリケーションのプロセスが消滅する場合があります。例えば、スレッドアンセーフな関数を使用している場合は、注意が必要です。

想定される原因:プログラムによる終了

Javaプロセスが、特別なメッセージ出力などがないまま、予想外の状態で終了した場合、原因の1つとして、次のどれかが考えられます。

- java.lang.Runtime.exit()を予想外の箇所で実行した
- java.lang.Runtime.halt()を予想外の箇所で実行した
- java.lang.System.exit()を予想外の箇所で実行した

"2.2 JavaVM終了時のメッセージ出力機能"を参照して、対処してください。

想定される原因:スタックオーバーフロー Windows64

通常、スタックオーバーフローが発生した場合、java.lang.StackOverflowErrorがスローされ、Windowsエラー報告「Windows Error Reporting(WER)」が検知してユーザーダンプを出力します。

しかし、OSが高負荷状態になったり、スタックオーバーフロー発生時のスタック残量が少なかったりすると、Windowsエラー報告にも制御が渡らないまま、痕跡を残さずにプロセスが消滅することがあります。

したがって、プロセスが消滅した原因が不明な場合は、スタックのサイズを拡張して、現象が改善できるかどうかを確認してください。スタックのサイズを拡張しても改善できない場合は、別の原因を調査してください。

Windowsエラー報告の説明は、"3.3 クラッシュダンプ"を参照してください。

スタックオーバーフローが発生したことを確認できた場合、該当するスタックのサイズをチューニングしてください。

想定される原因:その他

- Java VM以外のモジュールで、シグナルハンドラーを登録した場合、Javaアプリケーションが正常に動作せずに、異常終了することがあります。

3.7 ハングアップ(フリーズ)した場合

Javaプロセスが残っていてもプログラムが無応答になった場合、考えられる原因を説明します。

想定される原因:デッドロック

デッドロックが発生した場合、そのスレッドが停止されます。

ハングアップしたときに、スレッドダンプを採取して、デッドロックがないかどうかを確認してください。

スレッドダンプの詳細は、"3.2 スレッドダンプ"を参照してください。

想定される原因:ガーベジコレクション

ガーベジコレクションが発生すると、ガーベジコレクションが終了するまでの間、Javaアプリケーションのすべてのスレッドが停止されます。

これにより、Javaアプリケーションがハングアップしたかのように見える場合があります。

ガーベジコレクションのログを採取して、ガーベジコレクションが動作したタイミングを照合してください。ガーベジコレクションが原因で無応答のような現象になる場合は、Javaヒープをチューニングして、ガーベジコレクションの動作具合を改善してください。

想定される原因:JNI処理の異常

JNI経由でJava以外の言語で開発したネイティブモジュールと連携する際、JNIの使用方法を誤ると、ハングアップの原因となります。

このようなときは、"-Xcheck:jni"オプションを指定して、JNI処理でメッセージが出力されないかどうかを確認してください。

JNI処理に誤りがなくても、JNIモジュールで異常終了またハングアップが発生すると、Javaアプリケーションがハングアップする場合があります。例えば、スレッドアンセーフな関数を使用している場合は、注意が必要です。

3.8 スローダウンが発生した場合

Javaアプリケーションの動きが遅くなる現象(スローダウン)が発生した場合に、考えられる原因を説明します。

想定される原因:ガーベジコレクション

ガーベジコレクションが発生すると、ガーベジコレクションが終了するまでの間、Javaアプリケーションのすべてのスレッドが停止されます。このため、Javaアプリケーションのレスポンス(応答)が遅くなる場合があります。

特に物理メモリー(RAM)が少ないマシンの場合、ガーベジコレクションの実行に伴い、ディスクのスワッピングが発生し、スローダウンすることがあります。

ガーベジコレクションのログを採取して、ガーベジコレクションが動作したタイミングとスローダウンが発生したタイミングを照合してください。ガーベジコレクションが原因でスローダウンになる場合は、Javaヒープをチューニングして、ガーベジコレクションの動作具合を改善してください。

3.9 javaコマンドまたはJava VMのオプション

注意が必要なオプションについて以下に説明します。

-Xrs

本オプションが指定されたJavaプロセスの場合、当該プロセスへ送られた[Ctrl]+[Break]キー押下またはQUITシグナルに対する動作は、OSのデフォルト動作になります。

そのため、本オプションを指定したJavaプロセスに対して[Ctrl]+[Break]キー押下またはQUITシグナルが送られると、当該Javaプロセスは強制終了または異常終了します。

スレッドダンプを出力する可能性があるJavaプロセスに対して、本オプションは指定しないでください。

ただし、Windows(R)の場合、サービスとして登録されるJavaプロセスでは、「-Xrs」オプションを指定しない場合、ログオフ時に強制終了してしまいます。これが不都合な場合は、本オプションを指定してください。

-Xnoclassgc

クラスとメソッドの格納領域(メタスペース)のガーベジコレクションが抑止され、不要なクラスとメソッドが回収されなくなります。

これにより、java.lang.OutOfMemoryErrorが発生しやすくなるため、本オプションは指定しないでください。

3.10 動的コンパイル

Java VMは、Javaアプリケーションとして実行されるJavaメソッドに対して、必要に応じて自動的にコンパイル処理(動的コンパイル)をします。

以下のオプションを指定することで、動的コンパイルが発生するたびにその発生状況ログが標準出力へ出力されます。

```
-XX:+PrintCompilation
```

動的コンパイル発生状況のログ出力機能

どのJavaメソッドが、いつコンパイルされたかの情報を出力します。
Javaメソッドのコンパイルが、短い間に連続して発生している場合、Javaアプリケーションの実行性能に動的コンパイルが影響を与えている可能性があります。

動的コンパイル結果情報の出力形式

```
$1 $2 $3 $4 $5 ($6 bytes) $7
```

$1: Javaメソッドのコンパイル要求発生時間(ログ出力時の時間)

Javaメソッドのコンパイル要求が発生した時間(ログ出力時の時間)を、Java VMが起動されてからの経過時間(ミリ秒)で示します。

$2: 通し番号

コンパイル要求の数(コンパイル要求が発生したJavaメソッドの数)を通し番号で示します。

$3: メソッドのコンパイル種別

メソッドのコンパイル種別を表す5桁の文字列"%s!bn"を示します。5桁の文字は、それぞれ、動的コンパイラーが管理するメソッドの種別を表し、該当しない場合は空白が表示されます。

5桁の文字列は、動的コンパイラーの内部処理を解析するための、保守サポート向けの情報(動的コンパイラー処理ロジックを理解した高度なスキルを必要とする情報)です。

$4:メソッドのコンパイルレベル

動的コンパイラーが管理するメソッドのコンパイルレベルを表す数値を示します。

$5: Javaメソッド名

コンパイル要求が発生したJavaメソッドの名前を示します。

Javaメソッドを部分的にコンパイルする要求の場合($2に「%」の表示が含まれる場合)、Javaメソッド名の後に、Javaメソッド(バイトコード)のどの部分からコンパイルの対象になっているかを示す情報「(@ 数字)」が付加されます。

$6: Javaメソッドのバイト数

コンパイル対象となったJavaメソッドの大きさ(バイトコード・サイズ)をバイト数で示します。

ネイティブメソッドの場合は、(native)が表示されます。

$7: 空白または(static)または made not entrant

コンパイル対象となったJavaメソッドが、static修飾子を持つネイティブメソッドの場合は、"(static)"と出力されます。
コンパイル対象となったJavaメソッドが、ネイティブメソッドでない場合、またはstatic修飾子を持たないネイティブメソッドの場合、この位置には何も出力されません。
"made not entrant"は、これまでにコンパイラーによってつくられた機械命令のコードを無効にしたことを示します。

動的コンパイル発生状況の調査

Javaアプリケーションの実行は、Java VM起動直後はインタープリタ実行だけで行われ、次第にインタープリタ実行と動的コンパイル結果による機械命令実行との混合動作になります。また動的コンパイルにより翻訳された機械命令の内容も、Javaアプリケーションの実行が進むにつれて、次第にJavaアプリケーションの実行状況に合った翻訳結果に最適化されます。

Javaアプリケーションの実行には、以下の動的コンパイルによるオーバーヘッドや最適化状態の推移があるため、Javaアプリケーションとして安定した実行性能になるまで、プロセス起動時からある程度の時間を必要とする場合があります。

- 実行対象プログラムのクラスファイルを実行時に読み込むため、Javaアプリケーションとしての起動直後には、クラスファイルのロードおよび内容検査によるオーバーヘッドが発生します。
- 実行時に機械命令への翻訳処理をするため、そのオーバーヘッドが発生します。
- 機械命令への翻訳・最適化処理で必要とする情報を、Javaアプリケーション実行と同時に収集するため、Javaアプリケーション開始直後は最適化状態の推移が少なく、結果として実行性能が遅い機械命令となっている場合があります。

なお、Javaアプリケーションとして安定した実行性能になるまでに掛かる時間は、各アプリケーションによって異なります。

Javaアプリケーション実行時における動的コンパイルから受ける影響の有無は、Javaアプリケーションによる業務が開始された際に、動的コンパイルの発生頻度が高いかどうかを判断することで行います。

"動的コンパイル発生状況のログ出力機能"を使用して動的コンパイル結果情報を出力し、その結果から「動的コンパイル結果情報で、Javaメソッドのコンパイルが、短い間に連続して発生している」傾向が見て取れ、かつJavaアプリケーションとして安定した実行性能になるまでに掛かる時間との関連性が見て取れるかどうかを元に判断します。

注意

Javaアプリケーション起動時や、業務を開始してJavaアプリケーションへの入力が始まる時に、動的コンパイル処理が発生する頻度が高くなる傾向は、Javaアプリケーション実行時の一般的な傾向です。そのため、動的コンパイル発生状況の調査の結果として、「動的コンパイルの発生頻度が高い」などの傾向が見て取れる場合であっても、インタープリタ実行と機械命令による実行がバランスよく行われており、対応が不要な場合が多々あります。

Javaアプリケーションとして安定した実行性能になるまでに掛かる時間が実運用上の問題となった場合に、動的コンパイル発生状況の調査をしてください。

注意

Javaアプリケーション起動直後の性能に影響を与える処理は、動的コンパイル処理だけではありません。Javaアプリケーション自体の初期化処理や関連するアプリケーションの起動待ちなど、Javaアプリケーションの起動直後でだけ動作する動的コンパイル以外の要因についても考慮する必要があります。

注意

JavaアプリケーションがJSPで作成されている場合、アプリケーションを配備する際にJSPのプリコンパイル（javacコマンドによりJavaソースをクラスファイルへ変換する処理）が実施されていないと、Javaアプリケーション起動時にjavacコマンド実行によるオーバーヘッドが発生する場合があります。

JSPのプリコンパイル運用ができる場合は、JSPのプリコンパイルを実施することで、Javaアプリケーションとして安定した実行性能になるまでに掛かる時間が小さくなるかどうかを確認してください。

暖気運転

Javaアプリケーションを起動した後、業務としての運用を開始する時点で安定した実行性能を得る状態にするためには、暖機運転という運用をします。

暖機運転とは、Javaアプリケーションを起動した後、実際の業務を開始する前に、業務と同様のダミーデータを使用して疑似実行させることで、事前に主なJavaメソッドを動作させ、クラスファイルのロードやJavaメソッドの動的コンパイルを完了させておくことを言います。

暖機運転により、動的コンパイルなどのオーバーヘッドを減らすことができ、また機械命令への翻訳・最適化処理もその時点で行われるようになるため、業務としての運用開始時点から安定した実行性能を得ることができるようになります。

なお暖機運転をどの程度の期間行ったら良いのかについては、Javaアプリケーションの実行性能が運用要件を満たすところまで安定しているかのかどうかが、判断の基準になります。

ただし、Javaアプリケーションを起動してから業務開始までの時間は無限にあるわけではないので、"動的コンパイル発生状況のログ出力機能"の動的コンパイル結果情報の出力結果を参考に、暖気運用に掛ける時間をある程度のところで区切る判断をする必要があります。

注意

通常、Javaアプリケーションの実行には、実行結果の表示、データの更新などの結果を伴います。暖機運転による実行結果が、Javaアプリケーションの運用自体に影響を与えないように注意する必要があります。

例えば、暖機運転用のダミーデータをそのままデータベースに登録してしまい、運用時に誤ったデータを返却するなどの問題が発生しないように注意してください。

注意

Javaアプリケーション自体の初期化処理や関連するアプリケーションの起動待ちなど、動的コンパイル以外の要因でJavaアプリケーションとして安定した実行性能になるまでに時間が掛かっている場合は、暖機運転では対処できません。

第4章 オープンソフトウェアについて

OpenJDKは、クラスパス例外付きGNU General Public License v2（GPLv2+CPE）に基づいたソフトウェアです。

4.1 ソースコードの配布について

弊社がOSSのライセンス条件によりソースコードの提供義務を負うプログラムについては、ソースコードを提供する用意があります。提供期間は、本製品の受領日から3年間または本製品の保守サポートに関する契約を締結している間のどちらか長い方の期間になります。ソースコードが必要な場合は、弊社技術員へご連絡願います。