HITACHI
Inspire the Next

uCosminexus Application Runtime - Cosminexus Developer's Kit for Java 機能解説・リファレンス

3021-3-K02

前書き

■ 対象製品

●適用 OS：Red Hat Enterprise Linux 7（AMD/Intel 64），Red Hat Enterprise Linux 8（AMD/Intel 64）

P-9W43-9M11 uCosminexus Application Runtime with Java for Apache Tomcat 01-00

このプログラムプロダクトのほかにもこのマニュアルをご利用になれる場合があります。詳細は「リリースノート」でご確認ください。

■ 輸出時の注意

本製品を輸出される場合には、外国為替及び外国貿易法の規制並びに米国輸出管理規則など外国の輸出関連法規をご確認の上、必要な手続きをお取りください。

なお、不明な場合は、弊社担当営業にお問い合わせください。

■ 商標類

■ 発行

2022 年 7 月 3021-3-K02

■ 著作権

はじめに

uCosminexus Application Runtime with Java for Apache Tomcat は，日立独自の拡張機能を備えた Java 実行環境と JDK を提供する Cosminexus Developer's Kit for Java を同梱しています。

このマニュアルでは，Cosminexus Developer's Kit for Java について，日立独自の拡張機能の概要，提供機能を利用したメモリチューニングとトラブルシューティングの方法を解説しています（第 1 編　解説編）。また，Cosminexus Developer's Kit for Java が提供する各機能の詳細や使い方を説明しています（第 2 編　リファレンス編）。

なお，このマニュアルでは，Cosminexus Developer's Kit for Java を「日立 JavaVM」と呼んでいます。また，関連マニュアルである「uCosminexus Application Runtime for Apache Tomcat ユーザーズガイド」を「uCosminexus Application Runtime ユーザーズガイド」と表記しています。

■ 対象読者

uCosminexus Application Runtime with Java をご利用のお客様で，次に該当する方を対象としています。

- クラウドまたはオンプレミス環境に構築する業務システム（アプリケーションサーバシステム）を対象に，日立 JavaVM を利用して，信頼性確保に関するアプリケーションの機能（稼働監視，障害検知，保守資料収集）を設計，開発する人。
- 日立 JavaVM を利用して開発したアプリケーションサーバシステムの運用管理，保守を担当する人。

■ このマニュアルで使用する記号

このマニュアルで使用している記号を，次のように定義します。

記号	意　味
|	横に並べられた複数の項目に対する項目間の区切りを示し，「または」を意味します。 （例）　A | B 　A または B を指定することを示します。
{　}	この記号で囲まれている複数の項目のうちから 1 つを選択することを示します。項目が横に並べられ，記号 | で区切られている場合は，そのうちの 1 つを選択します。 （例）　{A | B | C} 　A，B または C のどれかを指定することを示します。
[　]	この記号で囲まれている項目は省略してもよいことを示します。複数の項目が横に並べて記述されている場合には，すべてを省略するか，記号 {} と同じくどれか 1 つを選択します。 （例 1）　[A] 　「何も指定しない」か「A を指定する」ことを示します。

記号	意　味
	(例 2)　[B ｜ C] 「何も指定しない」か「B または C を指定する」ことを示します。
…	記述が省略されていることを示します。 (例)　ABC… ABC の後ろに記述があり，その記述が省略されていることを示します。
＜　＞	この記号で囲まれている項目は，該当する要素やファイルなどを指定したり，該当する要素が表示されたりすることを示します。 (例 1)　＜プロパティ＞ プロパティを記述します。またはプロパティが表示されます。 (例 2)　＜ファイル名＞ ファイル名を指定します。
…	この記号の直前に示す記号を繰り返し，複数個指定できることを示します。 (例)　＜プロパティ＞… プロパティは複数個，繰り返して指定できます。

目次

前書き　2
はじめに　4

第 1 編　解説編

1　概要　12
1.1　日立 JavaVM の提供機能の概要　13
1.1.1　日立 JavaVM の特長　13
1.1.2　拡張機能の概要　13
1.2　システム構築・運用のポイント　15
1.2.1　システム構築のポイント　15
1.2.2　システム運用のポイント　16
1.2.3　システム運用の注意事項　16

2　メモリチューニング　17
2.1　GC と JavaVM のメモリ管理　18
2.1.1　GC と JavaVM のメモリ管理の概要　18
2.1.2　プロセスごとに使用するメモリの見積もり　18
2.2　SerialGC の仕組みと JavaVM のメモリチューニング　21
2.2.1　SerialGC の仕組み　21
2.2.2　Java ヒープのチューニング　31
2.2.3　Java ヒープ内の Tenured 領域のメモリサイズの見積もり　36
2.2.4　Java ヒープ内の New 領域のメモリサイズの見積もり　38
2.2.5　Java ヒープ内に一定期間存在するオブジェクトの扱いの検討　42
2.2.6　Java ヒープの最大サイズ/初期サイズの決定　43
2.2.7　Java ヒープ内の Metaspace 領域のメモリサイズの見積もり　44
2.2.8　拡張 verbosegc 情報を使用した FullGC の要因の分析方法　45
2.3　G1GC の仕組みと JavaVM のメモリチューニング　51
2.3.1　G1GC の仕組み　51
2.3.2　G1GC のチューニング　73

3　トラブルシューティング　87
3.1　JavaVM でのトラブルシューティング　88
3.1.1　資料の取得方法　89
3.1.2　トラブル事象別の原因切り分け方法　94

3.1.3 保守員調査依頼時の提供情報 113
3.2 システムで発生する代表的なトラブル 114

第 2 編 リファレンス編

4 日立 JavaVM の機能詳細 116
4.1 日立 JavaVM の機能の概要 117
4.2 クラス別統計機能 119
4.2.1 クラス別統計機能の概要 119
4.2.2 クラス別統計機能を前提とする機能 120
4.2.3 クラス別統計情報の出力 120
4.2.4 クラス別統計情報出力時の注意事項 122
4.3 インスタンス統計機能 123
4.3.1 インスタンス統計機能の概要 123
4.3.2 インスタンス統計機能で出力するクラス別統計情報 125
4.4 STATIC メンバ統計機能 129
4.4.1 STATIC メンバ統計機能の概要 129
4.4.2 STATIC メンバ統計機能で出力するクラス別統計情報 130
4.5 参照関係情報出力機能 133
4.5.1 参照関係情報出力機能の概要 133
4.5.2 参照関係情報出力機能で出力するクラス別統計情報 135
4.5.3 static フィールドを基点とした参照関係情報出力機能で出力するクラス別統計情報 138
4.5.4 static フィールドを基点とした参照関係情報出力時の注意事項 142
4.6 統計前の GC 選択機能 143
4.6.1 統計前の GC 選択機能の概要 143
4.6.2 GC の選択の指針 144
4.7 Tenured 領域内不要オブジェクト統計機能 145
4.7.1 Tenured 領域内不要オブジェクト統計機能の概要 145
4.7.2 Tenured 領域内不要オブジェクト統計機能で出力するクラス別統計情報 148
4.7.3 Tenured 領域内不要オブジェクト統計機能の実行に関する注意事項 149
4.8 Tenured 増加要因の基点オブジェクトリスト出力機能 154
4.8.1 Tenured 増加要因の基点オブジェクトリスト出力機能の概要 154
4.8.2 Tenured 増加要因の基点オブジェクトリスト出力機能で出力するクラス別統計情報 156
4.9 クラス別統計情報解析機能 157
4.9.1 クラス別統計情報解析機能の概要 157
4.9.2 クラス別統計情報解析機能の出力例 158
4.9.3 クラス別統計情報解析機能の注意事項 160
4.10 Survivor 領域の年齢分布情報出力機能 161
4.10.1 Survivor 領域の年齢分布情報出力機能の概要 161

4.10.2 Survivor 領域の年齢分布情報の出力形式と出力例 161
4.10.3 実行環境での設定 162
4.10.4 Survivor 領域の年齢分布情報出力機能使用時の注意事項 163
4.11 JIT コンパイル時の C ヒープ確保量の上限値設定機能 164
4.12 スレッド数の上限値設定機能 165
4.13 ファイナライズ滞留解消機能 166
4.13.1 ファイナライズ滞留解消機能の概要 166
4.13.2 ファイナライズ滞留解消機能の出力情報 166
4.13.3 ファイナライズ滞留解消機能の実行環境での設定 167
4.13.4 ファイナライズ滞留解消機能の注意事項 167
4.14 ログファイルの非同期出力機能 169
4.14.1 ログファイルの非同期出力機能の概要 169
4.14.2 ログファイルの非同期出力機能の対象ログファイル 169
4.14.3 ログファイルの非同期出力機能のエラーケース 169
4.14.4 ログファイルの非同期出力機能の注意事項 170
4.14.5 ログファイルの非同期出力機能のメモリ所要量 170
4.15 圧縮オブジェクトポインタ機能 171
4.15.1 圧縮オブジェクトポインタ機能の概要 171
4.15.2 圧縮オブジェクトポインタ機能の前提条件 171
4.15.3 圧縮オブジェクトポインタ機能の注意事項 171
4.16 日立 JavaVM の機能使用時の注意事項 173
4.17 Oracle 社の JDK と日立が提供する JDK との非互換性 174
4.17.1 デフォルトで選択されるメモリ管理方式 174
4.17.2 ランタイムイメージ 174
4.17.3 モジュール関連オプション 174

5 日立 JavaVM 起動オプション 176

5.1 日立 JavaVM 拡張オプションの一覧 177
5.2 日立 JavaVM 拡張オプションの詳細 183
-XX:+Hitachi（一覧表示オプション） 184
-XX:[+|-]HitachiThreadDump（拡張スレッドダンプ情報出力オプション） 184
-XX:[+|-]HitachiThreadDumpToStdout（拡張スレッドダンプ標準出力抑止オプション） 193
-XX:[+|-]HitachiThreadDumpWithHashCode（拡張スレッドダンプハッシュコード出力オプション） 194
-XX:[+|-]HitachiThreadDumpWithCpuTime（拡張スレッドダンプ CPU 利用時間出力オプション） 194
-XX:[+|-]HitachiThreadDumpWithBlockCount（拡張スレッドダンプブロック回数出力オプション） 195
-XX:HitachiJavaLog（ログファイル名のプリフィックス指定オプション） 195
-XX:HitachiJavaLogFileSize（最大ログファイルサイズ指定オプション） 196
-XX:[+|-]HitachiJavaLogNoMoreOutput（ログファイル入出力エラー発生時指定オプション） 197

-XX:HitachiJavaLogNumberOfFile（最大ログファイル数指定オプション） 198
-XX:[+|-]JavaLogAsynchronous 199
-XX:[+|-]StandardLogToHitachiJavaLog（標準出力のログファイル出力オプション） 199
-XX:[+|-]HitachiOutputMilliTime（詳細時間出力オプション） 201
-XX:[+|-]HitachiVerboseGC（拡張 verbosegc 情報出力オプション） 202
-XX:[+|-]HitachiCommaVerboseGC（CSV 出力オプション） 210
-XX:HitachiVerboseGCIntervalTime（拡張 verbosegc 情報出力間隔指定オプション） 218
-XX:[+|-]HitachiVerboseGCPrintCause（GC 要因内容出力オプション） 219
-XX:[+|-]HitachiVerboseGCPrintDate（拡張 verbosegc 情報日付出力オプション） 220
-XX:[+|-]HitachiVerboseGCCpuTime（拡張 verbosegc 情報 CPU 利用時間出力オプション） 221
-XX:[+|-]HitachiVerboseGCPrintTenuringDistribution（Survivor 領域の年齢分布出力オプション） 221
-XX:[+|-]HitachiVerboseGCPrintJVMInternalMemory（C ヒープ情報出力オプション） 223
-XX:[+|-]HitachiVerboseGCPrintThreadCount（スレッド数の出力オプション） 224
-XX:[+|-]HitachiVerboseGCPrintDeleteOnExit（java.io.File.deleteOnExit()が使用するヒープサイズの出力オプション） 226
-XX:[+|-]PrintCodeCacheInfo（コードキャッシュ領域情報出力オプション） 228
-XX:CodeCacheInfoPrintRatio（コードキャッシュ領域使用率指定オプション） 230
-XX:[+|-]PrintCodeCacheFullMessage（コードキャッシュ領域枯渇メッセージ出力オプション） 231
-XX:[+|-]HitachiOutOfMemoryCause（例外発生要因種別出力オプション） 233
-XX:[+|-]HitachiOutOfMemoryStackTrace（スタックトレース出力オプション） 234
-XX:HitachiOutOfMemoryStackTraceLineSize（スタックトレース行サイズ指定オプション） 235
-XX:[+|-]HitachiOutOfMemorySize（メモリサイズ出力オプション） 236
-XX:[+|-]HitachiOutOfMemoryAbort（強制終了オプション） 237
-XX:[+|-]HitachiOutOfMemoryAbortThreadDump（スレッドダンプ出力オプション） 238
-XX:[+|-]HitachiOutOfMemoryAbortThreadDumpWithJHeapProf（クラス別統計情報出力オプション） 239
-XX:[+|-]HitachiOutOfMemoryHandling（OutOfMemory ハンドリングオプション） 239
-XX:HitachiOutOfMemoryHandlingMaxThrowCount（最大発生回数の設定オプション） 243
-XX:[+|-]HitachiJavaClassLibTrace（クラスライブラリのスタックトレース出力オプション） 244
-XX:HitachiJavaClassLibTraceLineSize（クラスライブラリのスタックトレース行サイズ指定オプション） 246
-XX:[+|-]HitachiLocalsInThrowable（例外発生時のローカル変数情報収集オプション） 246
-XX:[+|-]HitachiLocalsInStackTrace（スレッドダンプ出力時のローカル変数出力オプション） 250
-XX:[+|-]HitachiLocalsSimpleFormat（ローカル変数情報の出力フォーマット変更オプション） 251
-XX:[+|-]HitachiTrueTypeInLocals（ローカル変数情報の実型名出力オプション） 252
-XX:HitachiCallToString（ローカル変数情報出力オプション） 253
-XX:[+|-]HitachiFullCore（システムリソース解除オプション） 255
-XX:HitachiJITCompileMaxMemorySize（JIT コンパイル時の確保メモリ上限値指定オプション） 256
-XX:[+|-]JITCompilerContinuation（JIT コンパイラ稼働継続機能オプション） 258
-XX:[+|-]UseCompressedOops（圧縮オブジェクトポインタ機能で使用する Java オプション） 259

-XX:HitachiThreadLimit（スレッド数の上限値指定オプション）　260
5.3 日立 JavaVM で使用するプロパティ　262
JP.co.Hitachi.soft.jvm.autofinalizer　262
5.4 日立 JavaVM で指定できる Java HotSpot VM のオプション　263
5.5 日立 JavaVM で使用する環境変数の一覧　269
5.6 日立 JavaVM で使用する環境変数の詳細　270
JAVACOREDIR　270

6 日立 JavaVM で使用するコマンド　271
6.1 コマンドの文法の記述形式　272
6.1.1 記述形式の詳細　272
6.2 コマンドの入力形式　274
6.2.1 入力形式の詳細　274
6.3 日立 JavaVM で使用するコマンドの一覧　276
6.4 日立 JavaVM で使用するコマンドの詳細　277
car_tar_gz（core アーカイブ機能）　277
car_tar_Z（core アーカイブ機能）　279
javacore（core ファイルとスレッドダンプの取得）　281
javagc（GC の強制発生）　284
javatrace（トレース情報の収集）　287
jheapprof（クラス別統計情報付き拡張スレッドダンプの出力）　290
jheapprofanalyzer（クラス別統計情報解析ファイルの CSV 出力）　295

7 トラブルシューティングに使用する資料の詳細　298
7.1 日立 JavaVM のスレッドダンプ　299
7.1.1 スレッドダンプ情報の構成　299
7.2 日立 JavaVM の GC ログ　301
7.3 JavaVM ログ（JavaVM ログファイル）　302
7.3.1 JavaVM ログファイルを出力するオプション　302
7.3.2 拡張 verbosegc 情報の取得　303
7.3.3 コードキャッシュ領域に関するログの内容　306
7.4 日立 JavaVM が出力するメッセージログ（標準出力およびエラーリポートファイル）　308
7.4.1 シグナルが発生した場合　308
7.4.2 C ヒープが不足した場合　318
7.4.3 Internal Error が発生した場合　320
7.4.4 スレッド作成に失敗した場合　321
7.5 JavaVM スタックトレース情報　322
7.5.1 -XX:+HitachiLocalsInThrowable オプションが指定されている場合　323
7.5.2 -XX:+HitachiLocalsInStackTrace オプションが指定されている場合　329

8 アプリケーション実装時の注意事項 331
8.1 アプリケーション実装時の注意事項 332

付録 337
付録 A このマニュアルの参考情報 338
付録 A.1 関連マニュアル 338
付録 A.2 このマニュアルでの表記 338
付録 A.3 英略語 339
付録 A.4 KB（キロバイト）などの単位表記について 340

索引 341

1 概要

Cosminexus Developer's Kit for Java（以降，「日立 JavaVM」と呼びます）は，日立独自の拡張機能を備えた Java 実行環境と JDK を提供しています。

この章では，日立 JavaVM の拡張機能の概要と，日立 JavaVM を利用する場合のシステム構築，運用のポイントについて説明します。

1.1　日立 JavaVM の提供機能の概要

日立 JavaVM は，アプリケーションサーバシステムの安定稼働を支援する拡張機能を備えています。拡張機能の利用によって，システムの処理性能を高めるためのチューニングに使用する情報や，障害発生時の調査・分析といったトラブルシューティングに使用する各種情報を取得できます。

> **ヒント**
>
> uCosminexus Application Runtime with Java が提供する日立 Java VM は，Java SE 11 に準拠しています。対応する Oracle 社製の JDK バージョンは，JDK 11 です。

1.1.1　日立 JavaVM の特長

日立 JavaVM を利用すると，標準の JavaVM と比較して，より多く詳細な粒度でトラブルシューティングに役立つ情報を取得できます。万一の障害発生時でも，取得した各種情報を基に迅速な問題解決につなげることができます。

- 日立 JavaVM では，スレッドダンプ出力時，日立独自に拡張された情報（拡張スレッドダンプ情報）を取得できます。拡張スレッドダンプ情報の活用によって，トラブルの切り分けや原因個所の特定，要因調査を容易に実施できます。
- 日立 JavaVM では，アプリケーション実行時のメモリ使用量を把握するために必要となる情報（拡張 verbosegc 情報※）を JavaVM ログファイルに出力します。この情報を基に，処理性能に影響を及ぼす要因を特定し，適切なチューニングを実施することで，システムの可用性向上を図れます。さらに，日立 JavaVM では，トラブルの要因分析やシステムの状態確認に利用できる情報も，JavaVM ログファイルに出力されるログから取得できます。

  注※

  GC（ガーベージコレクション）実行時の各領域の詳細なメモリサイズが出力されます。

1.1.2　拡張機能の概要

日立 JavaVM が提供する拡張機能の概要を次の表に示します。

各機能の詳細は，「4.　日立 JavaVM の機能詳細」を参照してください。

表 1-1　日立 JavaVM が提供する機能

機能	概要
クラス別統計機能	クラスごとに，参照関係にあるクラスのインスタンスの情報（クラス別統計情報）を拡張スレッドダンプに出力できます。

機能	概要
インスタンス統計機能※	クラスごとに，クラスのインスタンスの数やインスタンスの合計サイズを拡張スレッドダンプに出力できます。
STATIC メンバ統計機能※	クラスごとに，static メンバが持つインスタンスの合計サイズを拡張スレッドダンプに出力できます。
参照関係情報出力機能※	指定したクラスのインスタンスの参照関係を，先頭から順番に拡張スレッドダンプに出力できます。
統計前の GC 選択機能※	クラス別統計情報の出力前に実行する処理を選択できます。
Tenured 領域内不要オブジェクト統計機能※	Tenured 領域内で不要となったオブジェクトを特定できます。
Tenured 増加要因の基点オブジェクトリスト出力機能※	Tenured 領域内不要オブジェクト統計機能を使って特定した，不要なオブジェクトの基点となるオブジェクトの情報を拡張スレッドダンプに出力できます。
クラス別統計情報解析機能	拡張スレッドダンプファイルに出力されたクラス別統計情報を CSV 形式で出力できます。
Survivor 領域の年齢分布情報出力機能	CopyGC 実行時に，Survivor 領域の使用状況が調査できます。この情報は，メモリサイズのチューニングに使用できます。
JIT コンパイル時の C ヒープ確保量の上限値設定機能	JIT コンパイルで使用する C ヒープのサイズに上限値を設定できます。上限値を超えた場合は，JIT コンパイルが中止され，以降のコンパイルはインタプリタ方式で実行されます。これによって，JavaVM の強制終了を防ぎ，システムの停止を抑止できます。
スレッド数の上限値設定機能	使用できるスレッド数に上限値を設定できます。あらかじめ使用するスレッド数の上限を把握し，その数を基に C ヒープに割り当てるメモリサイズを決定することで，C ヒープ不足の発生を防止します。
ファイナライズ滞留解消機能	ファイナライズ処理の滞留を解消し，OS 資源の解放遅れなどの発生を抑止できます。
ログファイルの非同期出力機能	ログのファイル出力を非同期で実行できます。ログのファイル出力処理を専用で実行するスレッドを用意し，ログ出力以外の処理を実行するスレッドと処理のタイミングを合わせることなく，ログファイルを出力できます。
圧縮オブジェクトポインタ機能	Java アプリケーションによって作成される Java オブジェクトのサイズを圧縮することで，Java アプリケーションのメモリ使用効率を向上できます。

注※

クラス別統計情報を出力するクラス別統計機能の一つです。

1.2　システム構築・運用のポイント

ここでは，日立 JavaVM を利用してアプリケーションサーバシステムを構築および運用する際に必要な作業のポイントと注意事項について説明します。

1.2.1　システム構築のポイント

(1)　インストールとセットアップ

日立 JavaVM を利用するために，次の作業を実施します。

1. uCosminexus Application Runtime with Java のインストール

   uCosminexus Application Runtime with Java を，マニュアル「uCosminexus Application Runtime ユーザーズガイド」の手順に従って導入対象の環境へインストールします。uCosminexus Application Runtime with Java をインストールすることで，日立 JavaVM がインストールされます。

2. 日立 JavaVM を使用するための設定

   日立 JavaVM を使用してトラブルシューティングを行うには，JavaVM 起動オプションの設定や，導入対象の環境（アプリケーション基盤）に応じた設定が必要です。マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してデフォルトの設定値を確認してください。

   Java 実行環境として日立 JavaVM を使用するには，アプリケーションサーバシステムを動作させる前に，設定を実施してください。

(2)　メモリチューニングによるシステム処理性能の確保

日立 JavaVM を適用したシステムの処理性能を高めるには，基盤となる JavaVM 自体のチューニングを適切に実施する必要があります。

特に，GC（ガーベージコレクション）の仕組みを考慮した上で，適切なメモリ管理ができるようにチューニングすることで，システムの処理性能が向上します。GC の挙動はメモリ管理方式によって変わるため，システムの要件に合わせて適切なメモリ管理方式を選択してください。日立 JavaVM では，2 つのメモリ管理方式（SerialGC，G1GC）を選択できます。

日立 JavaVM によるメモリ管理の仕組みとメモリチューニングの方法は，「2.　メモリチューニング」を参照してください。

1.2.2　システム運用のポイント

(1)　トラブルシューティング

日立 JavaVM を利用して運用中のアプリケーションサーバシステムに障害が発生した場合，障害からの速やかな回復が最も重要な事項です。さらに障害を繰り返し発生させないためには，原因究明も必要です。トラブルシューティングでは，発生したトラブルの内容に応じて，日立 JavaVM から出力された情報（資料）を活用し，原因の切り分けや発生要因の特定などを行います。日立 JavaVM を利用したトラブルシューティングの詳細は，「3.　トラブルシューティング」を参照してください。

> **ヒント**
>
> 「3.　トラブルシューティング」では，システム運用時に発生する代表的なトラブルに対して確認が必要な資料を示し，原因究明の手順を解説しています。システムの運用を開始する前に，該当の資料や手順の流れを確認しておくことをお勧めします。

(2)　サポートサービスの利用

ご契約のサポートサービスに応じて，日立 JavaVM に関する技術情報や改良版の提供などのサービスを利用できます。詳細については，マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

1.2.3　システム運用の注意事項

日立 JavaVM を利用してシステムを運用する場合の注意事項について説明します。

(1)　一時ファイルに関する注意事項

日立 JavaVM は，ほかのプロセスとの通信のため，次のディレクトリに一時ファイルを作成して使用します。

```
/tmp/hsperfdata_<ユーザ名>/<プロセスID>
```

このため，日立 JavaVM を正常に動作させるには，/tmp に書き込み権限が必要となります。なお，作成された一時ファイルやディレクトリのアクセス権を変更したり，JavaVM 起動中に一時ファイルを削除したりする場合の動作は保証しません。

2 メモリチューニング

この章では，日立 JavaVM によるメモリ管理の仕組みとメモリチューニングの方法について説明します。

2.1　GC と JavaVM のメモリ管理

システムの処理性能を高めるには，基盤となる JavaVM 自体のチューニングを適切に実施する必要があります。

ここでは，GC（ガーベージコレクション）と JavaVM でのメモリ管理の仕組みの概要について説明します。

2.1.1　GC と JavaVM のメモリ管理の概要

JavaVM のチューニングの目的は，システムの処理性能の向上です。特に，GC の仕組みを考慮した上で，適切なメモリ管理ができるようにチューニングすることで，システムの処理性能が向上します。GC の挙動はメモリ管理方式によって変わるため，システムの要件に合わせて適切なメモリ管理方式を選択してください。日立 JavaVM では，次の 2 つのメモリ管理方式を選択できます。

表 2-1　各メモリ管理方式の特長

項番	メモリ管理方式	特長
1	SerialGC	スループットを重視するシステムに適している。 GC には長い時間が掛かる GC（FullGC）と，短い時間で終わる GC（CopyGC）がある。GC の時間を制御できない。 メモリサイズのチューニングをすることで，FullGC の発生回数を減少できる。
2	G1GC	大規模なメモリを使用するシステムやレスポンスを重視するシステムに適している。 項番 1 の方式と比較してスループットが低い。 GC には長い時間が掛かる GC（FullGC）と，短い時間で終わる GC（YoungGC，MixedGC）がある。YoungGC と MixedGC の GC の目標時間を指定できる。 メモリサイズのチューニングに加え，GC を行うスレッド数を増やすことで，FullGC の発生回数を減少できる。

SerialGC については「2.2　SerialGC の仕組みと JavaVM のメモリチューニング」を，G1GC については「2.3　G1GC の仕組みと JavaVM のメモリチューニング」を参照してください。

2.1.2　プロセスごとに使用するメモリの見積もり

アプリケーションサーバの各プロセスで使用するメモリの使用量の見積もりについて説明します。

(1)　アプリケーションサーバが使用する仮想メモリの使用量の見積もり

ここでは，仮想メモリの使用量の見積もり方法について説明します。なお，仮想メモリの使用量の計算式に使用している JavaVM 起動時のオプションの詳細については，「2.2.1(5)　SerialGC 使用時の JavaVM で使用するメモリ空間の構成と JavaVM オプション」を参照してください。

(a) 仮想メモリの使用量の計算式

仮想メモリの使用量（単位：メガバイト）の計算式を次に示します。

アプリケーションサーバの仮想メモリの使用量＝A＋B＋C＋（D＋22＋E）×F＋H

（凡例）

- A：Java ヒープサイズ
  初期値は，JavaVM 起動時の-Xms オプションに指定した値です。この値は，アプリケーションサーバ起動中に，最大で JavaVM 起動時の-Xmx オプションに指定した値まで拡張されます。
- B：Metaspace 領域サイズ
  初期値は，JavaVM が割り当てた値です。この値は，アプリケーションサーバ起動中に，最大で JavaVM 起動時の-XX:MaxMetaspaceSize オプションに指定した値まで拡張されます。
- C：ネイティブプログラム使用領域サイズ
  ネイティブプログラム使用領域サイズの値について，次の表に示します。

表 2-2　ネイティブプログラム使用領域サイズの値

OS 種別	使用領域サイズの値（単位：メガバイト）
Linux	300

- D：アプリケーションサーバのスレッド数
  アプリケーションサーバが使用するスレッド数です。アプリケーションサーバが使用するスレッド数については，マニュアル「uCosminexus Application Runtime ユーザーズガイド」のリソースの見積もりに関する説明を参照してください。
- E：G1GC 使用時のスレッド数
  G1GC を使用する場合に作られるスレッドです。SerialGC を使用する場合は作られません。G1GC を使用する場合は，論理 CPU 数の 3 倍の値にしてください。
- F：スタック領域サイズ
  JavaVM 起動時の-Xss オプションに指定した値です。
- H：コードキャッシュ領域の最大サイズ
  JavaVM 起動時の-XX:ReservedCodeCacheSize オプションに指定した値です。

(b) 仮想メモリの使用量を計算する場合の注意事項

- ネイティブプログラム使用領域サイズの値は変動します。値が変動するのは次のような場合です。
  - ORB のトレースサイズを変更した場合
  - JDBC ドライバの使用領域サイズの値を変更した場合
  - 使用する製品のネイティブライブラリ使用領域サイズの値を変更した場合

このような場合，製品ごとのメモリ使用量をネイティブプログラム使用領域サイズの値に加えてください。製品ごとのメモリ使用量は，使用する製品のドキュメントに従って算出してください。

- アプリケーションサーバの標準構成で使用する仮想メモリの使用量は，使用するソフトウェア製品の影響で，見積もり値よりも大きくなる場合があります。その場合の増加分については，仮想メモリの使用量の値に加えてください。増加分のメモリ使用量は，使用するソフトウェア製品のドキュメントに従い，製品ごとのメモリ使用量を算出してください。
- Linux の場合，メモリのスワップファイルが不足すると，システムがスローダウンします。仮想メモリの上限値を，実メモリサイズ分指定することをお勧めします。

2.2　SerialGC の仕組みと JavaVM のメモリチューニング

ここでは，SerialGC の仕組みと JavaVM のメモリチューニングの方法について説明します。

2.2.1　SerialGC の仕組み

SerialGC の仕組みについて説明します。

(1)　SerialGC の概要

GC は，プログラムが使用し終わったメモリ領域を自動的に回収して，ほかのプログラムが利用できるようにするための技術です。

GC の実行中は，プログラムの処理が停止します。このため，GC を適切に実行できるかどうかが，システムの処理性能に大きく影響します。

プログラムの中で new によって作成された Java オブジェクトは，JavaVM が管理するメモリ領域に格納されます。Java オブジェクトが作成されてから不要になるまでの期間を，**Java オブジェクトの寿命**といいます。

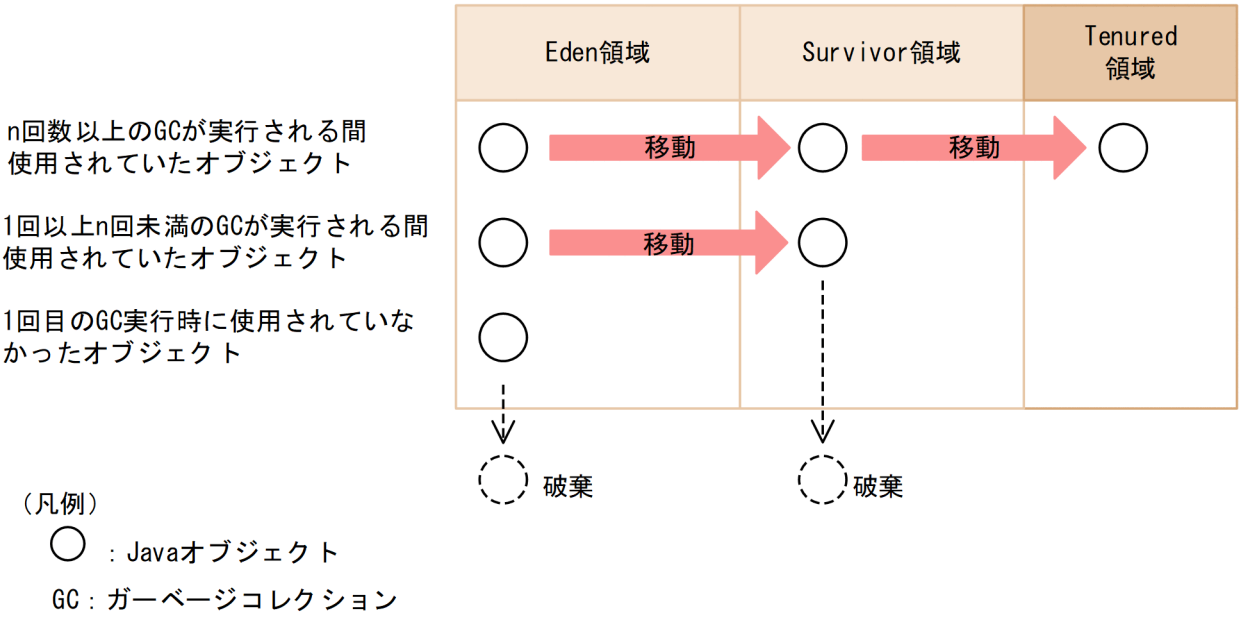
Java オブジェクトには，寿命の短いオブジェクトと寿命の長いオブジェクトがあります。サーバサイドで動作する Java アプリケーションの場合，リクエストやレスポンス，トランザクション管理などで，多くの Java オブジェクトが作成されます。これらの Java オブジェクトは，その処理が終わると不要になる，寿命が短いオブジェクトです。一方，アプリケーションの動作中使用され続ける Java オブジェクトは，寿命が長いオブジェクトです。

効果的な GC を実行するためには，寿命の短いオブジェクトに対して GC を実行して，効率良くメモリ領域を回収することが必要です。また，繰り返し使用される寿命の長いオブジェクトに対する不要な GC を抑止することが，システムの処理性能の低下防止につながります。これを実現するのが，**世代別 GC** です。

世代別 GC では，Java オブジェクトを，寿命が短いオブジェクトが格納される New 領域と，寿命が長いオブジェクトが格納される Tenured 領域に分けて管理します。New 領域はさらに，new によって作成されたばかりのオブジェクトが格納される Eden 領域と，1 回以上の GC の対象になり，回収されなかったオブジェクトが格納される Survivor 領域に分けられます。New 領域内で一定回数以上の GC の対象になった Java オブジェクトは，長期間必要な Java オブジェクトと判断され，Tenured 領域に移動します。

世代別 GC で管理するメモリ空間と Java オブジェクトの概要を次の図に示します。

図 2-1　世代別 GC で管理するメモリ空間と Java オブジェクトの概要



SerialGC の世代別 GC で実行される GC には，次の 2 種類があります。

- CopyGC

  Eden 領域と Survivor 領域を対象にした GC です。Java オブジェクトの作成によって，Eden 領域を使い切ると発生します。

- FullGC

  Tenured 領域も含む，JavaVM 固有領域全体を対象にした GC です。Tenured 領域を一定サイズまで使うと発生します。

一般的に，CopyGC の方が，FullGC よりも短い時間で処理できます。

次に，GC の処理について，ある Java オブジェクト（オブジェクト A）を例にして説明します。

Eden 領域で実行される処理

オブジェクト A の作成後，1 回目の CopyGC が実行された時点で使用されていない場合，オブジェクト A は破棄されます。

1 回目の CopyGC が実行された時点で使用されていた場合，オブジェクト A は Eden 領域から Survivor 領域に移動します。

Survivor 領域で実行される処理

Survivor 領域に移動したオブジェクト A は，そのあと何回か CopyGC が実行されると，Survivor 領域から Tenured 領域に移動します。移動する回数のしきい値は，JavaVM オプションや Java ヒープの利用状況によって異なります。「図 2-1」では，しきい値を n 回としています。

Survivor 領域への移動後，n 回目未満の CopyGC が実行された時点でオブジェクト A が使用されていなかった場合，オブジェクト A はその CopyGC で破棄されます。

Tenured 領域で実行される処理

オブジェクト A が Tenured 領域に移動した場合，そのあとの CopyGC でオブジェクト A が破棄されることはありません。CopyGC は，Eden 領域と Survivor 領域だけを対象としているためです。

このようにしてオブジェクトが移動することで，Tenured 領域の使用サイズは増加します。Tenured 領域を一定サイズまで使うと，FullGC が発生します。

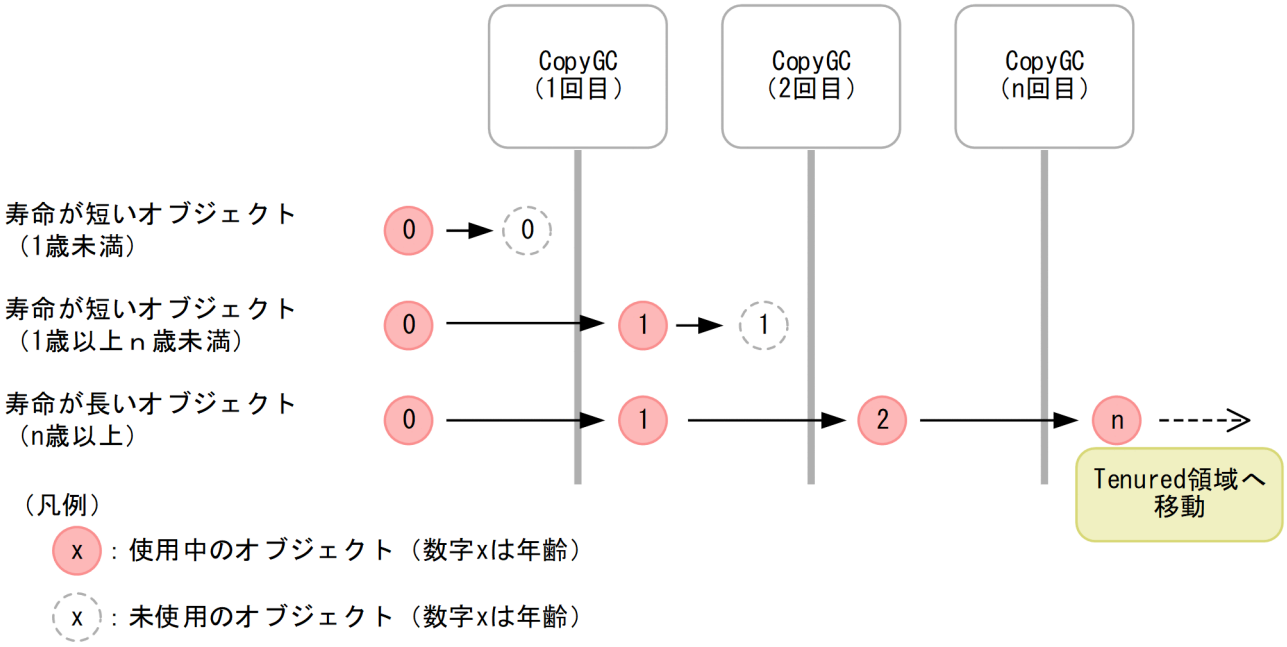
JavaVM のチューニングでは，JavaVM オプションでそれぞれのメモリ空間のサイズや割合を適切に設定することで，不要なオブジェクトが Tenured 領域に移動することを抑止します。これによって，FullGC が頻発することを防ぎます。

(2) オブジェクトの寿命と年齢の関係

オブジェクトが CopyGC の対象になった回数を**オブジェクトの年齢**といいます。

オブジェクトの寿命と年齢の関係を次の図に示します。

図 2-2 オブジェクトの寿命と年齢の関係



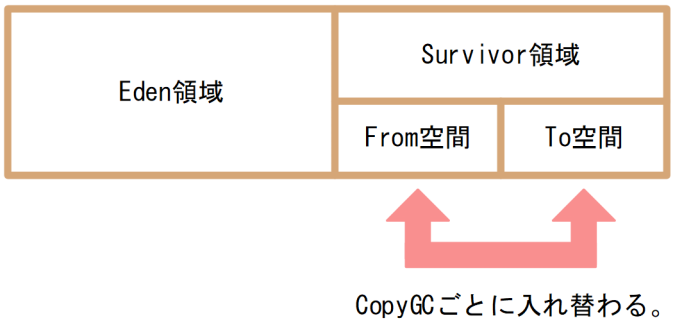
アプリケーションが開始して初期化処理が完了したあとで，何度かの CopyGC が実行されると，長期間必要になる寿命の長いオブジェクトは Tenured 領域に移動します。このため，アプリケーションの開始後しばらくすると，Java ヒープの状態は安定し，作成される Java オブジェクトとしては，寿命が短いオブジェクトが多くなります。特に，New 領域のチューニングが適切にできている場合，Java ヒープが安定したあとの大半のオブジェクトは，1 回目の CopyGC で回収される，寿命が短いオブジェクトになります。

(3) CopyGC の仕組み

JavaVM では，CopyGC の対象になる New 領域のメモリ空間を，Eden 領域，Survivor 領域に分けて管理します。さらに，Survivor 領域は，From 空間と To 空間に分けられます。From 空間と To 空間は，同じメモリサイズです。

New 領域の構成を次の図に示します。

図 2-3　New 領域の構成



Eden 領域は，new によって作成されたオブジェクトが最初に格納される領域です。プログラムで new が実行されると，Eden 領域のメモリが確保されます。

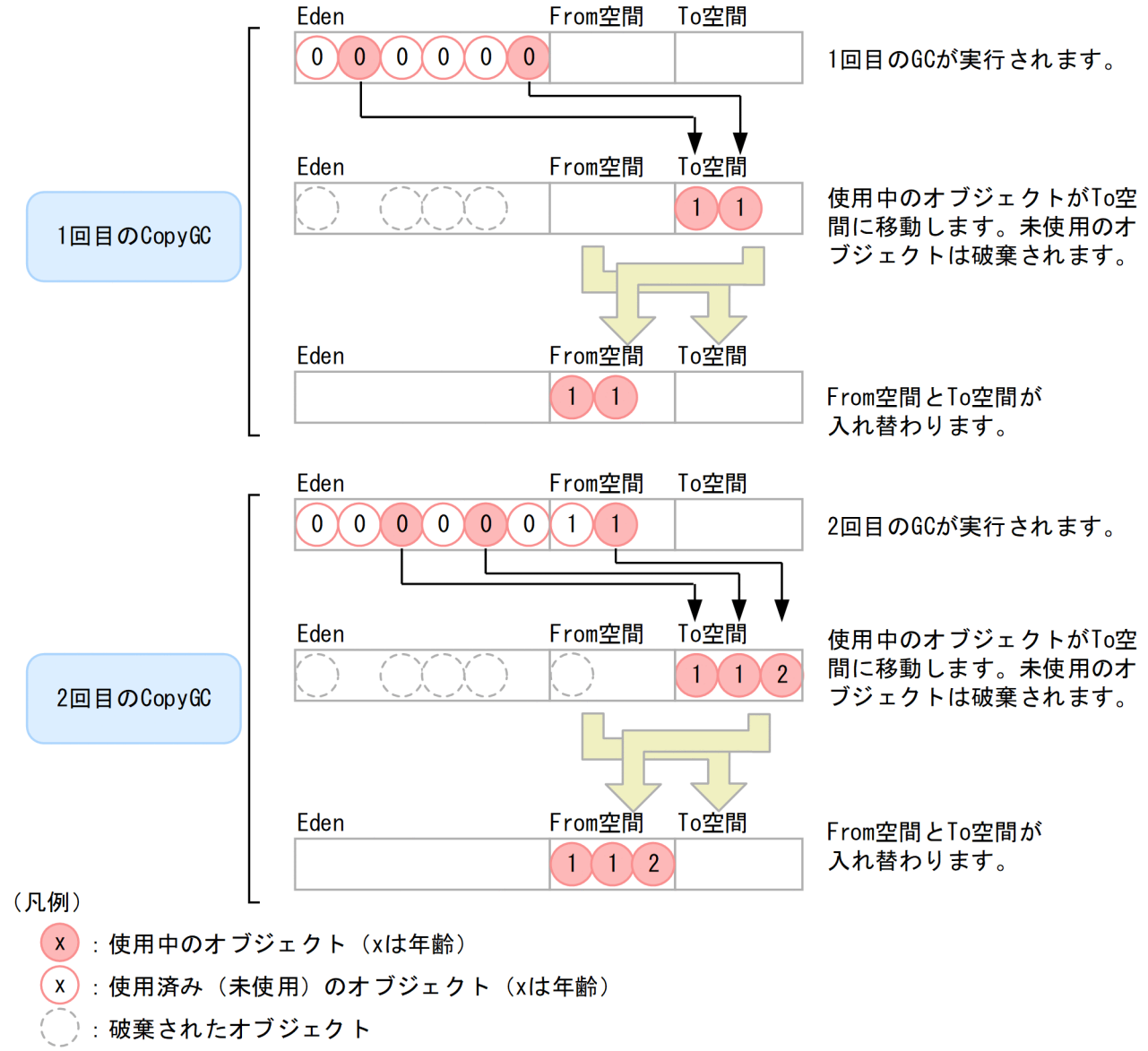
Eden 領域がいっぱいになると，CopyGC が実行されます。CopyGC では，次の処理が実行されます。

1. Eden 領域，および Survivor 領域の From 空間にある Java オブジェクトのうち，使用中の Java オブジェクトが，Survivor 領域の To 空間にコピーされます。使用されていない Java オブジェクトは破棄されます。
2. Survivor 領域の To 空間と From 空間が入れ替わります。

この結果，Eden 領域と To 空間は空になり，使用中のオブジェクトは From 空間に存在することになります。

CopyGC 実行時に発生するオブジェクトの移動を次の図に示します。

図 2-4　CopyGC 実行時に発生するオブジェクトの移動



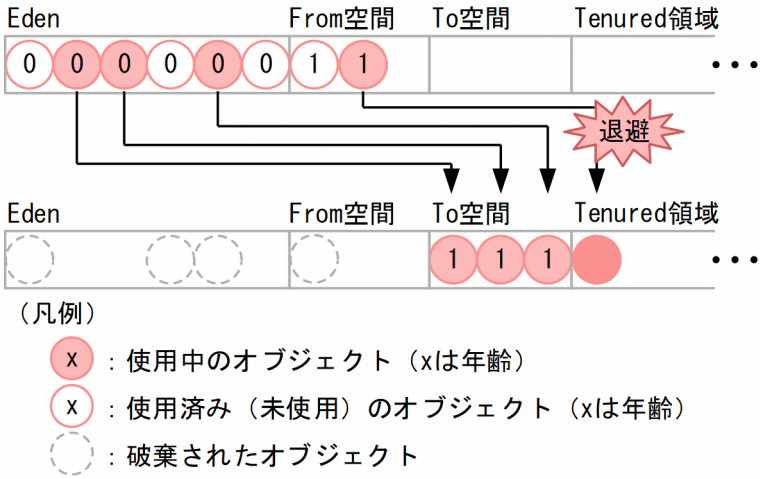
このようにして，使用中のオブジェクトは，CopyGC が発生するたびに，From 空間と To 空間を行ったり来たりします。ただし，寿命の長いオブジェクトを行き来させ続けると，コピー処理の負荷などが問題になります。これを防ぐために，From 空間と To 空間で Java オブジェクトを入れ替える回数にしきい値を設定して，年齢がしきい値に達した Java オブジェクトは Tenured 領域に移動させるようにします。

(4)　オブジェクトの退避

年齢がしきい値に達していない Java オブジェクトが Tenured 領域に移動することを，退避といいます。退避は，CopyGC 実行時に Eden 領域および From 空間で使用中のオブジェクトが多くなり，移動先である To 空間のメモリサイズが不足する場合に発生します。この場合，To 空間に移動できなかったオブジェクトが，Tenured 領域に移動します。

オブジェクトの退避を次の図に示します。

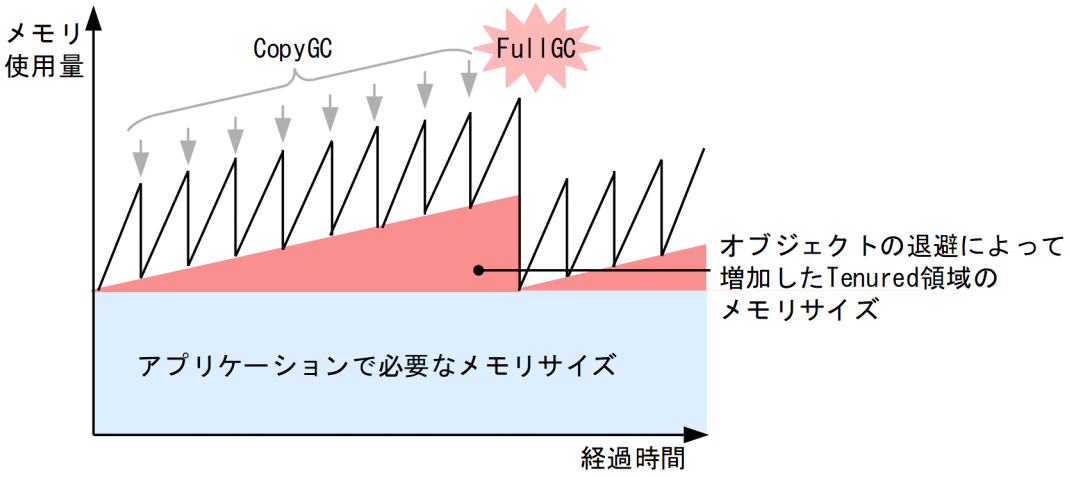
図 2-5　オブジェクトの退避



オブジェクトの退避が発生した場合，Tenured 領域に本来格納されないはずの寿命の短いオブジェクトが格納されます。これが繰り返されると，CopyGC で回収されるはずのオブジェクトがメモリ空間に残っていくため，Java ヒープのメモリ使用量が増加していき，最終的には FullGC が発生します。

オブジェクトの退避が発生した場合のメモリ使用量の変化について，次の図に示します。

図 2-6　オブジェクトの退避が発生した場合のメモリ使用量の変化



FullGC では，システムが数秒から数十秒停止することがあります。

したがって，メモリ空間の構成とメモリサイズを検討するときには，オブジェクトの退避が発生しないように，Eden 領域と Survivor 領域のバランスを検討する必要があります。

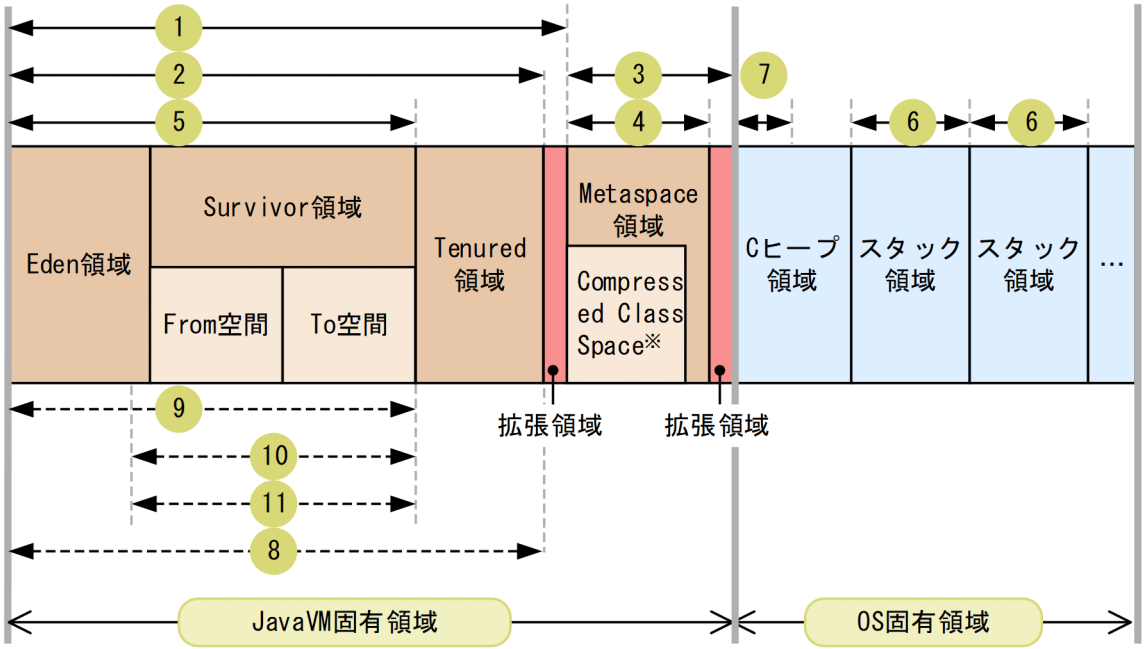
(5)　SerialGC 使用時の JavaVM で使用するメモリ空間の構成と JavaVM オプション

ここでは，SerialGC 使用時の JavaVM で使用するメモリ空間の構成と，JavaVM オプションについて説明します。

JavaVM では，**JavaVM 固有領域**と **OS 固有領域**という，2 種類のメモリ空間を使用します。

JavaVM で使用するメモリ空間の構成を次の図に示します。なお，図中の番号は，「表 2-3　JavaVM メモリ空間のサイズや割合などを指定する JavaVM オプション」の項番と対応しています。

図 2-7　JavaVM で使用するメモリ空間の構成



（凡例）

◀——▶：サイズを指定するオプションの対象になる範囲

◀- - -▶：割合または回数を指定するオプションの対象になる範囲

注※

圧縮オブジェクトポインタ機能が有効の場合だけ割り当てられる領域

それぞれの領域について説明します。なお，Eden 領域，Survivor 領域，および Tenured 領域を合わせた領域を，**Java ヒープ**といいます。

Eden 領域

new によって作成された Java オブジェクトが最初に格納される領域です。

Survivor 領域

New 領域に格納されていた Java オブジェクトのうち，CopyGC 実行時に破棄されなかった Java オブジェクトが格納される領域です。Survivor 領域は，From 空間と To 空間で構成されます。From 空間と To 空間のサイズは同じです。

Tenured 領域

長期間必要であると判断された Java オブジェクトが格納される領域です。Survivor 領域で指定回数を超えて CopyGC の実行対象となり，破棄されなかった Java オブジェクトが，この領域に移動されます。

Metaspace 領域

ロードされた class などの情報が格納される領域です。

ただし，圧縮オブジェクトポインタ機能が有効な場合，Metaspace 領域内に Compressed Class Space という領域が作成されます。この領域に，Java ヒープ内のオブジェクトから参照されるクラス情報が配置されます。そして，それ以外のメソッド情報などが Compressed Class Space 以外の Metaspace 領域に配置されます。圧縮オブジェクトポインタ機能については，「4.15　圧縮オブジェクトポインタ機能」を参照してください。

C ヒープ領域

JavaVM 自身が使用する領域です。JNI で呼び出されたネイティブライブラリでも使用されます。

例えば，次のような領域があります。

コードキャッシュ領域

JIT コンパイルによって生成された JIT コンパイルコードが格納される領域です。

JavaVM は，呼び出し回数やループ回数が多い Java メソッドを JIT コンパイルして実行することで，処理を高速化します。

> **メモ**
>
> **コードキャッシュ領域の最大サイズについて**
>
> コードキャッシュ領域の最大サイズは，ReservedCodeCacheSize オプションに指定します。
>
> ReservedCodeCacheSize オプションには，デフォルト値以上の値を指定してください。Java HotSpot VM のオプションのデフォルト値については，マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。
>
> また，コードキャッシュ領域が枯渇していた場合，または枯渇するおそれがある場合は，コードキャッシュ領域の拡張を検討してください。
>
> コードキャッシュ領域を拡張する場合は，次の点に注意してください。
>
> - JIT コンパイルコードのサイズは計算で見積もることができません。そのため，Java アプリケーション実行環境でコードキャッシュ領域の使用量を実測し，システムが使用するコードキャッシュ領域（64 ビット版で最大 2 メガバイト）の使用量を考慮した上で，コードキャッシュ領域の最大サイズを見積もってください。
> - 仮想メモリの使用量の見積もりについては，「2.1.2　プロセスごとに使用するメモリの見積もり」を参照してください。

スタック領域

Java スレッドのスタック領域です。

それぞれの領域のサイズや割合などを指定する JavaVM オプションを次の表に示します。なお，表の項番は，「図 2-7」と対応しています。

表 2-3　JavaVM メモリ空間のサイズや割合などを指定する JavaVM オプション

項番	オプション名	オプションの意味
1	-Xmx<size>	Java ヒープの最大サイズを設定します。
2	-Xms<size>	Java ヒープの初期サイズを設定します。
3	-XX:MaxMetaspaceSize=<size>	Metaspace 領域の最大サイズを設定します。
4	-XX:MetaspaceSize=<size>	Metaspace 領域の初期サイズを設定します。
5	-Xmn<size>	New 領域の初期値および最大値を設定します。
6	-Xss<size>	1 スレッド当たりのスタック領域の最大サイズを設定します。
7	-XX:ReservedCodeCacheSize=<size>	JavaVM が使用する領域のうち，コードキャッシュ領域の最大サイズを設定します。
8	-XX:NewRatio=<value>	New 領域に対する Tenured 領域の割合を設定します。 <value>が 2 の場合は，New 領域と Tenured 領域の割合が，1:2 になります。
9	-XX:SurvivorRatio=<value>	Survivor 領域の From 空間と To 空間に対する Eden 領域の割合を設定します。 <value>に 8 を設定した場合は，Eden 領域，From 空間，To 空間の割合が，8:1:1 になります。
10	-XX:TargetSurvivorRatio=<value>	CopyGC 実行後の Survivor 領域内で Java オブジェクトが占める割合の目標値を設定します。
11	-XX:MaxTenuringThreshold=<value>	CopyGC 実行時に，From 空間と To 空間で Java オブジェクトを入れ替える回数の最大値を設定します。 設定した回数を超えて入れ替え対象になった Java オブジェクトは，Tenured 領域に移動されます。

注　<size>の単位はバイトです。

重要

Java ヒープ領域，Metaspace 領域，Compressed Class Space，C ヒープ領域のどれかが不足すると OutOfMemory が発生し，メモリ不足が解消されないかぎり正常に稼働できない状態が長く続くことになります。

これに対して，OutOfMemory 発生時のシステムへの影響を小さくするために，次のオプションを使用できます。

-XX:+HitachiOutOfMemoryAbort（OutOfMemory 発生時強制終了機能）

OutOfMemory 発生時強制終了機能は，Java ヒープ不足や Metaspace 領域不足，Compressed Class Space 不足などが原因で OutOfMemory が発生した場合に，アプリケーションを強制終了するための機能です。Java ヒープ領域，Metaspace 領域，

Compressed Class Space，C ヒープ領域のメモリ不足によって OutOfMemory が発生したときに，アプリケーションを強制終了します。

アプリケーションサーバの自動再起動や待機系システムへの自動切り替えなどの仕組みを前提として，アプリケーションサーバプロセスの生存を監視している場合には，このオプションを有効にすることで，正常な状態への回復を促す効果があります。

オプションの詳細については，「-XX:[+|-]HitachiOutOfMemoryAbort（強制終了オプション）」を参照してください。

(6) GC の発生とメモリ空間の関係

GC は，メモリ空間の使用状況に応じて発生します。ここでは，GC が発生するタイミングについて説明します。

重要

RMI を使ってリモートオブジェクトの登録や参照をすると，定期的に GC が発生することがあります。GC が発生するタイミングは，次のどちらかのプロパティにミリ秒単位で指定できます。デフォルトは 3600000 ミリ秒（1 時間）です。

- sun.rmi.dgc.client.gcInterval プロパティ
- sun.rmi.dgc.server.gcInterval プロパティ

GC が発生するタイミングを変更する場合は，どちらかのプロパティにミリ秒単位で任意の値を指定してください。なお，指定できる値の範囲は，1～Long.MAX_VALUE-1 です。詳細は，Web ページ（https://docs.oracle.com/javase/6/docs/technotes/guides/rmi/sunrmiproperties.html）を参照してください。

(a) CopyGC が発生するタイミング

CopyGC は，次のタイミングで発生します。

1. Eden 領域へのアロケーションで空き領域が不足した場合
2. jheapprof コマンドに-copygc オプションを指定して実行した場合

(b) FullGC が発生するタイミング

FullGC は，次のタイミングで発生します。

1. New 領域（Eden 領域と Survivor 領域の合計）で使用しているメモリサイズが Tenured 領域の最大値に対する未使用メモリサイズを上回っている状態の時に，Eden 領域へのアロケーションで空き領域が不足した場合

重要

上記のタイミングで必ず FullGC が発生するわけではありません。この状況になった場合，FullGC を発生させる必要があるかどうかについては，過去の CopyGC での Tenured 領域への移動量の実績値に基づいて，JavaVM が決定します。

2. CopyGC の実施の結果，New 領域（Eden 領域と Survivor 領域の合計）から Tenured 領域へのオブジェクトの移動に失敗した場合
3. New 領域と Tenured 領域のそれぞれの未使用メモリサイズを上回るメモリサイズ（Java オブジェクトのサイズ）のアロケーション要求があった場合
4. CopyGC の実施の結果，次のどちらかの状態になった場合
   - 確保済み Tenured 領域の未使用メモリサイズが 10,000 バイトを下回った場合
   - CopyGC 実施時の Tenured 領域へのオブジェクトの移動によって，確保済み Tenured 領域の拡張が発生した場合
5. java.lang.System.gc()メソッドが実行された場合
6. Metaspace 領域にアロケーションしたいメモリサイズが確保済み Metaspace 領域の未使用メモリサイズを上回る場合
7. javagc コマンドを実行した場合
8. jheapprof コマンドを実行した場合

JavaVM のチューニングでは，主に 1.と 3.の発生を抑えることを検討します。

メモ

FullGC が発生した場合の要因は，拡張 verbosegc 情報を使用して確認できます。FullGC 発生時に要因を確認する方法については，「2.2.8　拡張 verbosegc 情報を使用した FullGC の要因の分析方法」を参照してください。

2.2.2　Java ヒープのチューニング

SerialGC 使用時の Java ヒープのチューニングについて説明します。

一般的に，FullGC の方が，CopyGC よりも長い処理時間を要します。

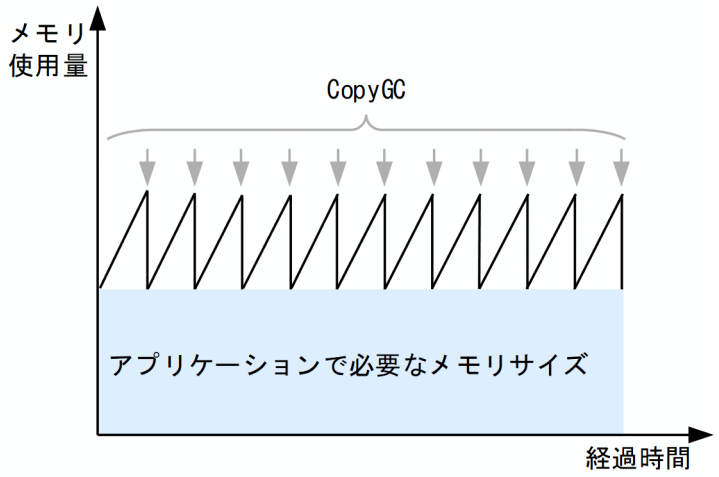
GC の実行中はプログラムの処理が停止するため，FullGC の発生はシステムの処理性能に大きな影響を与えます。このため，New 領域への CopyGC の実施によって適切にメモリを回収して，Java ヒープ全体を対象とする FullGC の発生をできるだけ抑止することが，システムの停止回数の削減や，処理性能向上につながります。これを実現するためには，JavaVM オプションでそれぞれのメモリ空間のサイズや割合を適切に設定することが必要です。

これらを考慮した上で，JavaVM のチューニングは，次の 2 点を目的として実施します。

- FullGC をできるだけ発生させないこと
- FullGC の頻発を抑止し，不要な CopyGC を発生させないこと

理想的なメモリ使用量と経過時間の関係を次の図に示します。

図 2-8　理想的なメモリ使用量と経過時間の関係



この図の場合は，寿命の短いオブジェクトはすべて CopyGC によって回収できていて，オブジェクトの昇格や退避が発生しません。このため，CopyGC 実行後のメモリサイズが一定です。これによって，FullGC が発生しない，安定した状態での運用を実現できます。

JavaVM のチューニングでは，この状態を理想として，JavaVM の使用するメモリ空間の各領域で使用するメモリサイズを見積もってチューニングします。

ヒント

チューニングの目安

「図 2-8」では，FullGC を一度も発生させない理想的な例を示しましたが，現実的には FullGC が 1 回発生する間に CopyGC が 10～20 回程度発生することを目安にして，チューニングしてください。

なお，Java ヒープのチューニングをした段階で Java アプリケーションのテストを実施してください。Java ヒープのメモリを適切に見積もっても FullGC が頻発する場合は，Survivor 領域があふれているなど，チューニングに問題がないか確認してください。

(1)　Java ヒープのメモリサイズの見積もり

JavaVM のチューニングでは，JavaVM 固有領域の各領域のメモリサイズを適切に見積もる必要があります。

見積もりのポイントになるメモリサイズは次のとおりです。

- Java ヒープ全体のメモリサイズ
- Tenured 領域のメモリサイズ
- Survivor 領域のメモリサイズ
- Eden 領域のメモリサイズ

このほか，Metaspace 領域も，必要に応じて見積もります。

CopyGC 後の生存オブジェクトのサイズが Survivor 領域のサイズよりも大きい場合，Survivor 領域があふれ，1 度の CopyGC の実行で Tenured 領域に昇格するオブジェクトが発生します。また，Survivor 領域のサイズが小さい場合に，Survivor 領域の使用率が上がってくると，本来短寿命（CopyGC 間隔以下，または CopyGC 間隔の 1～2 倍程度の寿命）の Java オブジェクトが，数回の CopyGC の実行で Tenured 領域に昇格してしまいます。

ヒント

次に示す問題が確認できた場合は，Survivor 領域の不足が原因で FullGC が発生しています。なお，拡張 verbosegc 情報とは，起動時に-XX:+HitachiVerboseGC オプションを設定しておくと，GC 発生時に JavaVM ログファイルに出力される情報です。

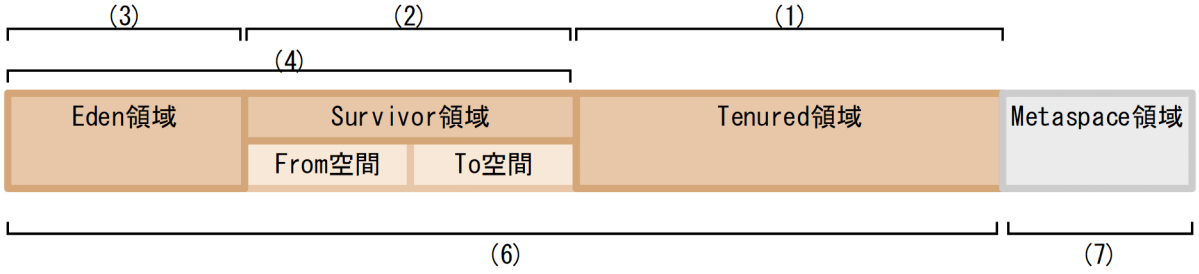
- Tenured 領域の増加要因が短寿命オブジェクトであると特定された場合
- 拡張 verbosegc 情報で，CopyGC 時の Survivor 領域あふれが確認できた場合
- -XX:+HitachiVerboseGCPrintTenuringDistribution 指定時に出力される拡張 verbosegc 情報で，オブジェクトの昇格年齢が常に 1 であることが観測できた場合

これらの問題を回避するためには，-XX:SurvivorRatio オプションの設定値を小さくして，Eden 領域と Survivor 領域の割合を最適化する必要があります。

このことを考慮し，見積もりでは，まず，Tenured 領域のメモリサイズと New 領域のメモリサイズを算出して，それらを基に Java ヒープ全体のメモリサイズを算出します。

メモリサイズを見積もる順序を次の図に示します。図中の番号の順番で見積もりを実施します。

図 2-9　メモリサイズを見積もる順序



見積もり手順を示します。なお，番号は図中の番号と対応しています。

1. Tenured 領域で使用するメモリサイズを見積もります。
   見積もり方法については，「2.2.3　Java ヒープ内の Tenured 領域のメモリサイズの見積もり」を参照してください。

2. Survivor 領域で使用するメモリサイズを見積もります。
   見積もり方法については，「2.2.4(1)　Java ヒープ内の Survivor 領域のメモリサイズの見積もり」を参照してください。

3. Eden 領域で使用するメモリサイズを見積もります。
   見積もり方法については，「2.2.4(2)　Java ヒープ内の Eden 領域のメモリサイズの見積もり」を参照してください。

4. 2.と 3.の合計として，New 領域全体のメモリサイズを算出します。

5. 一定期間存在するオブジェクトの扱いを検討して，必要なメモリサイズを Tenured 領域または New 領域のメモリサイズに追加します。
   検討方法については，「2.2.5　Java ヒープ内に一定期間存在するオブジェクトの扱いの検討」を参照してください。

6. 1.，4.および 5.の合計として，Java ヒープ全体のメモリサイズを算出します。

7. 必要に応じて Metaspace 領域のメモリサイズを見積もります。
   見積もり方法については，「2.2.7　Java ヒープ内の Metaspace 領域のメモリサイズの見積もり」を参照してください。

(2)　Java ヒープのメモリサイズの設定方法

見積もったメモリサイズは，「2.2.1(5)　SerialGC 使用時の JavaVM で使用するメモリ空間の構成と JavaVM オプション」で説明したオプションで指定します。それぞれの領域のメモリサイズの設定方法を次に示します。

Java ヒープ全体のメモリサイズ

-Xmx<size>オプションで最大サイズを指定して，-Xms<size>オプションで初期サイズを指定します。

Tenured 領域のメモリサイズ

-XX:NewRatio=<value>オプションで，Java ヒープ全体に対する，Tenured 領域と New 領域の分割比を指定します。例えば，「-XX:NewRatio=5」とした場合には，-Xmx<size>オプションで指定したメモリサイズが，次のように分割されます。

```
New領域のメモリサイズ：Tenured領域のメモリサイズ ＝ 1：5
```

Survivor 領域のメモリサイズと Eden 領域のメモリサイズ

-XX:SurvivorRatio=<value>オプションで，New 領域全体に対する，Survivor 領域と Eden 領域の分割比を指定します。なお，分割比は，Survivor 領域の From 空間および To 空間に対して Eden 領

域を何倍確保するかの数値で指定します。例えば，「-XX:SurvivorRatio=2」とした場合には，-XX:NewRatio=<value>オプションで決まったNew領域のメモリサイズが，次のように分割されます。

```
Eden領域のメモリサイズ：From空間のメモリサイズ：To空間のメモリサイズ＝2：1：1
```

Metaspace 領域のメモリサイズ

-XX:MaxMetaspaceSize=<size>オプションで最大サイズを指定して，-XX:MetaspaceSize=<size>オプションで初期サイズを指定します。

(3) Javaヒープのメモリサイズの使用状況の確認方法

それぞれのメモリサイズは，アプリケーションを実際に動作させて，メモリ使用量を測定しながらチューニングしていきます。日立JavaVMでは，-XX:+HitachiVerboseGCオプションを指定してアプリケーションを起動することで，GC実行時の各領域の詳細なメモリサイズを**拡張verbosegc情報**として出力できます。この出力内容を基にチューニングを実施してください。

拡張verbosegc情報として出力できる主な内容を次に示します。

- GCの発生日時
- GC種別
- GC情報※1
- GC経過時間
- Eden情報※1
- Survivor情報※1
- Tenured情報※1
- Metaspace領域情報※1
- GC要因内容※2

注※1

各領域について，GC前後のメモリサイズおよび使用メモリサイズが出力されます。

注※2

-XX:+HitachiVerboseGCPrintCauseオプションが指定されている場合に出力されます。

拡張verbosegc情報の出力例とFullGCの要因分析方法については，「2.2.8　拡張verbosegc情報を使用したFullGCの要因の分析方法」を参照してください。また，オプションについては，「-XX:[+|-]HitachiVerboseGC（拡張verbosegc情報出力オプション）」を参照してください。

なお，日立JavaVMでは，javagcコマンドを使用して，任意のタイミングでFullGCを発生させることもできます。この場合，-vオプションを指定することで，拡張verbosegc情報と同じ内容を出力できます。javagcコマンドの詳細については，「javagc（GCの強制発生）」を参照してください。

2.2.3　Java ヒープ内の Tenured 領域のメモリサイズの見積もり

SerialGC 使用時の Tenured 領域のメモリサイズの見積もりについて説明します。

Tenured 領域のメモリサイズは，次のように見積もります。

```
Tenured領域のメモリサイズ
＝アプリケーションで必要なメモリサイズ＋New領域のメモリサイズ
```

ここでは，アプリケーションで必要なメモリサイズの算出方法について説明します。また，見積もったメモリサイズに New 領域のメモリサイズを追加する理由についても説明します。

(1)　アプリケーションで必要なメモリサイズの算出

Tenured 領域のメモリサイズは，アプリケーションが最低限必要とするメモリサイズから見積もります。必要なメモリサイズが確保できない場合，OutOfMemoryError が発生して JavaVM が停止します。

アプリケーションが必要とするメモリサイズは，FullGC 実行時の拡張 verbosegc 情報で，FullGC 実行後に使用しているメモリサイズを確認することで判断できます。これは，FullGC 実行後に Java ヒープ全体から不要なオブジェクトをすべて削除した状態のメモリサイズが，アプリケーションが必要とするメモリサイズに近いと考えられるためです。

FullGC 実行時の拡張 verbosegc 情報の出力例を次に示します。

```
…
[VGC]<Wed May 11 23:12:05 2005>[Full GC 31780K->30780K(32704K), 0.2070500secs][DefNew::Eden:
 3440K->1602K(3456K)][DefNew::Survivor:58K->0K(64K)][Tenured: 28282K->29178K(29184K)][Metasp
ace:3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K
)][cause:ObjAllocFail][User: 0.0156250 secs][Sys: 0.0312500 secs]
…
```

「Full GC」に続いて出力されている情報のうち，GC の実行後の情報「->30780K」を確認します。ここでは，FullGC 実行後に，30,780 キロバイトのメモリサイズを必要としていることが分かります。

何回分かの FullGC の拡張 verbosegc 情報を集め，GC 実行後のメモリサイズがいちばん大きい情報を，アプリケーションが必要とするメモリサイズであると判断してください。

(2)　Java ヒープ内の New 領域のメモリサイズを追加する理由

Tenured 領域のメモリサイズには，アプリケーションが最低限必要とするメモリサイズに，New 領域分のメモリサイズを追加することをお勧めします。これは，Tenured 領域の未使用メモリサイズが New 領域の使用メモリサイズを下回ることによって，FullGC が頻発するのを防ぐためです。

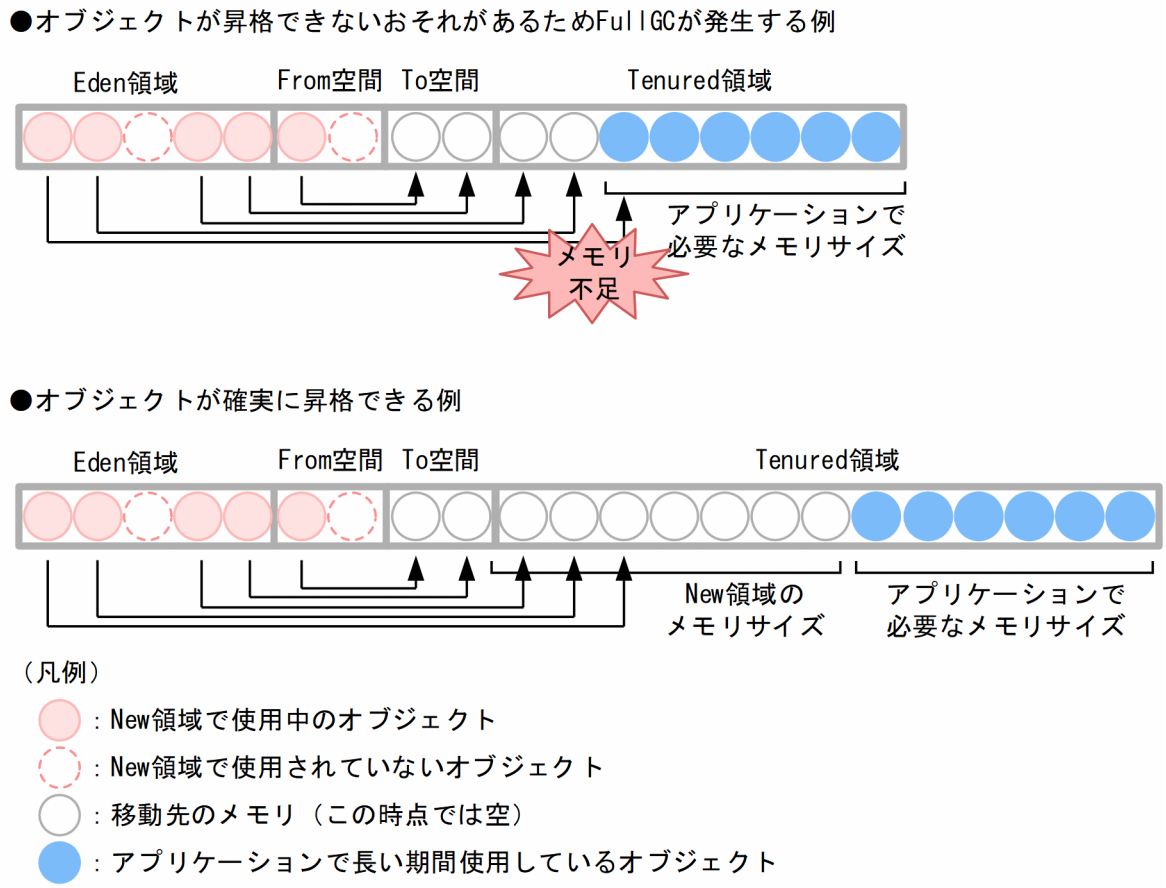
通常，Eden 領域がいっぱいになると，CopyGC が発生します。このとき，Eden 領域と Survivor 領域の From 空間に存在する使用中の Java オブジェクトが，Survivor 領域の To 空間に移動しようとします。このとき，Tenured 領域の未使用領域が Eden 領域と Survivor 領域で使用中のメモリサイズよりも小さいと，New 領域のすべての Java オブジェクトが昇格した場合に，Java オブジェクトを Tenured 領域に移

動できなくなります。そこでJavaVMは，FullGCを発生させ，Tenured領域の未使用メモリサイズを増やそうとします。

これを防ぐために，Tenured領域には，アプリケーションが必要とするメモリサイズに加えて，New領域分のメモリサイズを追加してください。

考え方を次の図に示します。

図 2-10　New領域のメモリサイズを追加する理由の考え方



- オブジェクトが昇格できないおそれがあるためFullGCが発生する例

  Tenured領域のメモリの空き領域（アプリケーションで必要なメモリ領域以外の領域）がNew領域のメモリサイズよりも小さいため，Eden領域およびFrom空間からの移動オブジェクトが多い場合，オブジェクトの昇格に対応できないおそれがあります。この場合，FullGCが発生します。

- オブジェクトが確実に昇格できる例

  Tenured領域のメモリの空き領域（アプリケーションで必要なメモリ領域以外の領域）がNew領域と同じサイズ分確保してあるため，Eden領域およびFrom空間からの移動オブジェクトが多い場合も，オブジェクトの昇格に対応できます。

なお，New領域のメモリサイズの見積もりについては，「2.2.4　Javaヒープ内のNew領域のメモリサイズの見積もり」で説明します。

ヒント

拡張 verbosegc 情報などで確認したときに，CopyGC が発生しないで FullGC が頻発している場合，New 領域からの退避オブジェクトに対して Tenured 領域のメモリサイズが小さいことが考えられます。New 領域のサイズを増やした場合などにこの状態になることがあります。必要に応じて，Tenured 領域のメモリサイズを見直してください。また，New 領域内の Eden 領域と Survivor 領域の関係もあわせて見直してください。

2.2.4 Java ヒープ内の New 領域のメモリサイズの見積もり

SerialGC 使用時の New 領域のメモリサイズの見積もりについて説明します。

New 領域のメモリサイズは，次のように見積もります。

```
New領域のメモリサイズ
＝Survivor領域のメモリサイズ＋Eden領域のメモリサイズ
```

ここでは，Survivor 領域および Eden 領域のメモリサイズを見積もる方法について説明します。

(1) Java ヒープ内の Survivor 領域のメモリサイズの見積もり

Survivor 領域のメモリサイズは，実際にアプリケーションを動作させて，Survivor 領域の使用状況を確認しながらチューニングしていきます。

チューニングの流れを次に示します。

1. アプリケーションでのリクエストおよびレスポンス処理に使用するメモリサイズを見積もり，それを Survivor 領域のメモリサイズに指定して，アプリケーションを実行します。

   このとき，チューニングで使用する情報を出力するために，-XX:+HitachiVerboseGCPrintTenuringDistribution オプションを指定して起動します。

2. Survivor 領域に割り当てられているメモリサイズと，アプリケーション実行時に実際に使用されているメモリ使用量から，メモリ使用率を確認します。

   メモリ使用率が 100%に近い場合，CopyGC 実行時に New 領域および Survivor 領域の From 空間の使用中オブジェクトが To 空間に入り切らなくなり，オブジェクトの退避が発生します。この場合は，Survivor 領域を増やすことを検討してください。

3. Survivor 領域のオブジェクトの年齢分布を確認します。

   Survivor 領域のメモリサイズを増やしたり，昇格するためのしきい値を上げたりすることで，オブジェクトが昇格するのが遅くなります。寿命の長いオブジェクトを Survivor 領域に格納し続けるのは，性能を低下させる要因になります。

逆に，Survivor 領域のメモリサイズを減らしたり，昇格するためのしきい値を下げたりすることで，オブジェクトが昇格するのが早くなります。ただし，寿命の短いオブジェクトが昇格するのは，FullGC の発生頻度を増やす要因になります。

Survivor 領域のメモリサイズと昇格のしきい値は，この 2 つのバランスを取るように検討してください。

それぞれのチューニング作業について説明します。

(a) リクエストおよびレスポンス処理に使用するメモリサイズの見積もり

Survivor 領域は，寿命の短いオブジェクトを格納する領域です。サーバサイドで動作するアプリケーションの場合，リクエストやレスポンスを処理するために使われている，寿命の短いオブジェクトを格納する領域と考えることができます。このため，Survivor 領域のメモリサイズの見積もりでは，ある時点で存在する寿命が短いオブジェクトの最大サイズ，つまり，ある時点でのリクエストやレスポンスの処理に使用するメモリの最大サイズを考えます。例えば，ステートレスなサーブレットで構成されたアプリケーションの場合，Survivor 領域のメモリサイズを，「1 つのリクエスト処理で使用する最大メモリサイズ×リクエストの同時実行数」と考えることができます。

(b) メモリ使用率の確認

「(a)リクエスト/レスポンス処理に使用するメモリサイズの見積もり」で見積もった値を Survivor 領域のメモリサイズとして設定して，アプリケーションを実行します。実行時に使用されるメモリ使用量から，Survivor 領域に割り当てたメモリサイズに対するメモリ使用率を確認します。

> **メモ**
>
> Survivor 領域のメモリサイズは，直接は指定できません。
>
> Survivor 領域のメモリサイズを指定する場合は，まず，-Xmx オプションで Java ヒープの最大サイズを指定して，-XX:NewRatio=<value>によって Java ヒープのメモリサイズを New 領域と Tenured 領域で分ける割合を指定した上で，-XX:SurvivorRatio=<value>オプションによって，Eden 領域との割合を指定する必要があります。

メモリ使用率は，拡張 verbosegc 情報で確認できます。

CopyGC 実行時の拡張 verbosegc 情報の出力例を次に示します。

```
…
[VGC]<Wed May 11 23:12:05 2005>[GC 27340K->27340K(32704K), 0.0432900 secs][DefNew::Eden: 344
0K->0K(3456K)][DefNew::Survivor: 58K->64K(64K)][Tenured: 23841K->27282K(29184K)][Metaspace:
3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][c
ause:ObjAllocFail][User: 0.0156250 secs][Sys: 0.0312500 secs]
…
```

「DefNew::Survivor: 58K->64K(64K)」は，「GC 実行前のメモリサイズ->GC 実行後のメモリサイズ（割り当てられているメモリサイズ)」を意味します。この場合，64 キロバイトの Survivor 領域中 64 キロバイトがすでに使用されていて，使用率は 100%になります。これは，CopyGC で，To 空間のメモリサイ

ズが不足し，Java オブジェクトの退避が行われたことを示しています。退避では，Tenured 領域に本来格納されないはずの寿命の短いオブジェクトが格納され，FullGC の発生頻度を増やす要因になります。このような場合は，Survivor 領域のメモリサイズを増やすことを検討してください。新しい Survivor 領域のメモリサイズは次のように見積もります。

```
新しいSurvivor領域のメモリサイズ
＝現在のSurvivor領域のメモリサイズ＋退避されたJavaオブジェクトの合計サイズ
```

「退避された Java オブジェクトの合計サイズ」は，GC 実行後の Tenured 領域メモリの増加サイズで近似できます。この例では，「Tenured: 23841K->27282K(29184K)」が，Tenured 領域メモリの増加サイズを示していて，27,282 キロバイト－23,841 キロバイト＝ 3,441 キロバイトとなります。

(c) オブジェクトの年齢分布の確認と見積もり

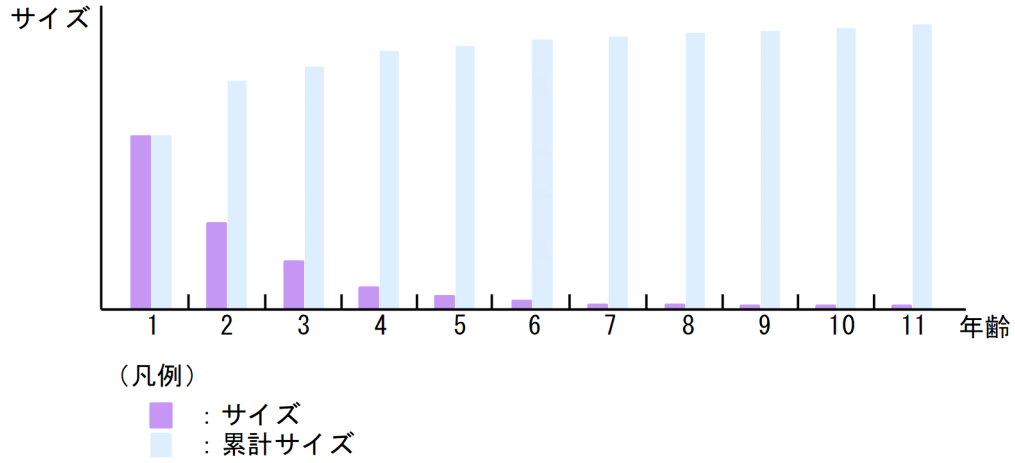
Survivor 領域のオブジェクトの年齢分布を確認し，寿命の長いオブジェクトが存在し続けていないか，または寿命の短いオブジェクトが昇格していないかを確認します。オブジェクトの年齢分布は，-XX:+HitachiVerboseGCPrintTenuringDistribution オプションの出力結果で確認できます。

起動時に-XX:+HitachiVerboseGCPrintTenuringDistribution オプションを指定すると，Survivor 領域の使用状況が CopyGC 発生のタイミングで JavaVM ログファイルに出力されます。出力例を次に示します。

```
[PTD]<Wed May 28 11:45:23 2008>[Desired survivor:5467547 bytes][New  threshold:3][MaxTenurin
gThreshold:31][age1:1357527][age2:1539661]
```

「New threshold:」に続けて出力されているのが，次の CopyGC で昇格する Java オブジェクトの最低の年齢です。例の場合は，次回の CopyGC で，年齢が 3 歳以上の Java オブジェクトが昇格します。「age<数値>:」に続けて出力されているのが，Survivor 領域で 1 歳からその年齢までの Java オブジェクトが使用しているメモリサイズの累計です。例の場合は，1 歳の Java オブジェクトのメモリサイズが 1,357,527 バイト，1 歳と 2 歳の Java オブジェクトのメモリサイズの累計が 1,539,661 バイトであることを示しています。また，累計から逆算することで，2 歳の Java オブジェクトのメモリサイズが 182,134(1,539,661-1,357,527)バイトであることが分かります。一般的に，Survivor 領域のオブジェクトの年齢分布は次の図に示すグラフのようになります。

図 2-11　Survivor 領域のオブジェクトの年齢分布



グラフの「サイズ」は，ある「年齢」の Java オブジェクトの合計のサイズです。また，「累計サイズ」は，ある「年齢」までの Java オブジェクトの合計のサイズです。

このグラフでは，Survivor 領域の Java オブジェクトが占めるサイズが，年齢が上がるに連れて減少しています。また，1 歳年齢が上がったときに減少するサイズは年齢が若いほど大きくなります。このことから，次のことが分かります。

- 若い年齢の Java オブジェクトほど，Survivor 領域中に占めるサイズが大きい
- 若い年齢の Java オブジェクトほど，GC で回収されやすい

この例では，6 歳以上の Java オブジェクトはほとんど CopyGC で回収されていません。そのため，Java オブジェクトの昇格のしきい値が 7 歳以上の場合，回収される可能性が低い Java オブジェクトに対して CopyGC を行うことになり，性能を低下させる要因になります。逆に，Java オブジェクトの昇格のしきい値が 2 歳以下の場合，CopyGC で回収される可能性の高いオブジェクトが昇格することになり，FullGC の発生が増す要因になります。この例では，昇格のしきい値を 5～6 歳程度とするのが，バランスの取れたしきい値となります。

グラフの傾きはシステムによって異なるため，Survivor 領域のオブジェクトの年齢分布を確認し，システムごとの最適な昇格年齢を決定することが重要です。

メモ

Java オブジェクトの昇格のしきい値は，CopyGC ごとに動的に変更され，-XX:MaxTenuringThreshold=<value>オプションと，Survivor 領域のメモリサイズおよび-XX:TargetSurvivorRatio=<value>オプションに設定した値を基に決まります。-XX:MaxTenuringThreshold=<value>は，昇格のしきい値の最大の年齢です。Java オブジェクトの年齢がこの値を超えると，必ず昇格します。-XX:TargetSurvivorRatio=<value>は，Survivor 領域のメモリ使用率の目標値です。JavaVM は，Survivor 領域のメモリ使用率が，できるだけこの値に近くなるように，昇格のしきい値を決定します。具体的には，CopyGC が終了したときの 1 歳から n 歳までの Java オブジェクトの累計サイズが，目標の使用率となる

n を探し，次の CopyGC の昇格のしきい値を n にします。-XX:TargetSurvivorRatio=<value>のデフォルト値は 50%です。Survivor 領域のメモリ使用率が大きいほど Survivor 領域を有効に利用できますが，Survivor 領域の空きに余裕がなくなるため，オブジェクトの退避が発生しやすくなります。

(2) Java ヒープ内の Eden 領域のメモリサイズの見積もり

Eden 領域のメモリサイズは，CopyGC を発生させる間隔に影響します。Eden 領域のメモリサイズを大きく取ると，CopyGC の発生間隔が長くなります。なお，CopyGC に掛かる時間は，使用中のオブジェクトの個数に影響され，Eden 領域のメモリサイズにはあまり影響されません。このため，CopyGC が頻発しないように，Eden 領域のメモリサイズを十分に確保することが，性能向上には効果的です。

2.2.5 Java ヒープ内に一定期間存在するオブジェクトの扱いの検討

これまでの説明は，オブジェクトの寿命に応じて，それぞれの領域に次のように格納することを前提としてきました。

- アプリケーションの動作に必要な寿命の長いオブジェクトは，Tenured 領域に格納する。
- リクエスト処理やレスポンス処理などの寿命の短いオブジェクトは，New 領域に格納する。

しかし，寿命が中間的な一定期間使用されるオブジェクトがあります。このようなオブジェクトは，寿命は長くありませんが，何回かの CopyGC の対象になります。

メモリサイズの見積もりでは，これらのオブジェクトを，New 領域，Tenured 領域のどちらかに格納することを前提として，見積もりをする必要があります。

ここでは，それぞれの特徴を示します。アプリケーションの種類や目的に応じて，どちらかのメモリサイズを増加させるように見積もりをしてください。

(1) Java ヒープ内の New 領域に格納する方法

一定期間存在するオブジェクトを New 領域で管理する方法です。New 領域のメモリサイズに，これらの一定期間存在するオブジェクト分のメモリサイズを追加して見積もります。

New 領域サイズを大きくしてオブジェクトの Tenured 領域への移動を抑止することによって，FullGC の発生も抑止できます。ただし，CopyGC 実行時に New 領域内にある使用中オブジェクトの数が増えるため，New 領域内でのコピー処理に時間が掛かり，1 回当たりの CopyGC 実行時間は長くなります。CopyGC の実行時間が FullGC 実行時間よりも長くなるような場合は，メモリサイズの再見積もりが必要です。また，メモリ空間のサイズ設定によっては，本来 CopyGC で回収されるはずの寿命の短いオブジェクトが使用する領域が不足して，Tenured 領域への退避が発生するおそれがあります。この場合は，最終的には FullGC が発生してしまいます。

なお，一定期間存在するオブジェクトを New 領域で管理できているかどうかは，拡張 verbosegc 情報で確認できます。実際にアプリケーションを動作させて，出力された拡張 verbosegc 情報で，CopyGC 発生後の Tenured 領域のメモリサイズが大幅に増えていないことを確認してください。

New 領域での管理に失敗している場合，システムの処理性能が大幅に低下していることがあります。また，New 領域で管理できるオブジェクトの最大の年齢には限界があります（限界はプラットフォームやバージョンによって異なります。詳細は，マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照し，Java HotSpot VM の-XX:MaxTenuringThreshold=<value>オプションのデフォルト値に関する説明を確認してください）。New 領域で管理できていないことが分かった場合は，一定期間存在するオブジェクトは，Tenured 領域に格納して管理することを検討してください。Tenured 領域で管理する方法については，「2.2.5(2)　Java ヒープ内の Tenured 領域に格納する方法」を参照してください。

(2)　Java ヒープ内の Tenured 領域に格納する方法

New 領域で管理するオブジェクトの最大の年齢は，-XX:MaxTenuringThreshold=<value>オプションで指定できます。例えば，「-XX:MaxTenuringThreshold=2」を指定しておけば，3 回目の CopyGC の対象になったオブジェクトは，すべて Tenured 領域に移動します。

この方法を使用すると，CopyGC の対象になるオブジェクトが少なくなり，実行時間が短縮できます。ただし，多くのオブジェクトが Tenured 領域に移動するため，Tenured 領域がいっぱいになった段階で FullGC が発生します。システムを安定して動作させるためには，システムに掛かる負荷が低いときなどに，FullGC を強制的に発生させてください。FullGC を強制的に発生させるには，次の方法があります。

- プログラム内で System.gc()メソッドを呼び出す
- javagc コマンドを実行する
  javagc コマンドの詳細については，「javagc（GC の強制発生）」を参照してください。

2.2.6　Java ヒープの最大サイズ/初期サイズの決定

Tenured 領域および New 領域の見積もりができたら，それらを基に Java ヒープの最大サイズと初期サイズを決定します。

Java ヒープの最大サイズは，次のように決定します。

```
Javaヒープの最大サイズ
＝Tenured領域のメモリサイズ＋New領域のメモリサイズ
```

Java ヒープのメモリサイズを設定する場合，まず，-Xmx オプションに，拡張領域のサイズも含む，Java ヒープ領域の最大サイズを指定します。次に，-Xms オプションに，Java ヒープ領域の初期サイズを指定します。なお，-Xms オプションに指定するサイズは，-Xmx オプションに指定したサイズよりも小さくする必要があります。

JavaVM は，起動時に，-Xms オプションで指定された分のメモリ領域を Java ヒープ領域として確保します。そのあと，アプリケーション実行中に-Xms オプションで指定した以上のメモリ領域が必要になった場合に，-Xmx オプションで指定したサイズになるまで，ヒープ領域を追加して割り当てていきます。逆に，アプリケーションの中で不要なメモリ領域が発生した場合は，-Xms オプションで指定したサイズまで，Java ヒープ領域として確保している領域を減らしていきます。

なお，システムの安定稼働のためには，-Xmx オプションと-Xms オプションには同じ値を指定することをお勧めします。

2.2.7　Java ヒープ内の Metaspace 領域のメモリサイズの見積もり

SerialGC 使用時の Metaspace 領域のメモリサイズの見積もりについて説明します。Metaspace 領域は，ロードされた class などが格納される領域です。

Metaspace 領域のメモリサイズは，「2.2.1(5)　SerialGC 使用時の JavaVM で使用するメモリ空間の構成と JavaVM オプション」で示したとおり，-Xmx オプションで指定したメモリサイズ（Java ヒープ）とは別に確保されます。

Java HotSpot VM のオプションのデフォルト値については，マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

次に，アプリケーションサーバを例に Metaspace 領域の使用量の見積もり方法を示します。

Metaspace 領域の使用量の見積もり方法

Metaspace 領域でのメモリ使用量は，大体，ロードされるクラスファイルの合計サイズになります。

Metaspace 領域のメモリサイズは，-XX:MaxMetaspaceSize=<size>オプション，および-XX:MetaspaceSize=<size>オプションで指定します。これらのオプションのデフォルト値については，マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。なお，アプリケーションのインポート時に一時的に Metaspace 領域の使用量が増加する場合があります。

Metaspace 領域は，JavaVM 上で動作するプログラムのすべてのクラスファイルのサイズの総和から見積もることができます。例えば，Tomcat の場合，アプリケーションのクラスファイル，Tomcat が提供するクラスファイル，JDK が提供するクラスファイルのサイズの総和です。なお，JDK が提供するクラスファイルのサイズは約 130 メガバイトです。

> **メモ**
>
> **開発環境でのソフト参照による Metaspace 領域の圧迫の解消方法**
>
> アプリケーションサーバでは，ソフト参照の影響によって，アプリケーションをアンデプロイしたときに Metaspace 領域の解放が遅延することがあります。このため，開発環境などでアプリケーションのデプロイおよびアンデプロイを繰り返した場合，Metaspace 領域の解放遅延が原因で，Metaspace 領域が圧迫されることがあります。ソフト参照による Metaspace 領域の圧迫は，次のオプションを指定すると解消できます。

- -XX:SoftRefLRUPolicyMSPerMB=0

-XX:SoftRefLRUPolicyMSPerMB オプションに 0 を指定すると，すべてのソフト参照が無効になります。ソフト参照は，性能向上のためのキャッシュとして使用されることが多いため，このオプションの指定によってアプリケーションの性能が劣化するおそれがあります。このため，このオプションは，開発環境で Metaspace 領域が圧迫された場合にだけ指定してください。

メモ

Metaspace 領域の特殊なチューニングの考え方

Metaspace 領域はオプションに設定された値のメモリを必ず使用するわけではありません。実際には，実行に必要なサイズのメモリだけ使用します。

- Metaspace 領域の OutOfMemoryError の発生リスクを下げる
  見積もり時に想定していなかった Metaspace 領域の使用量の増加によって，OutOfMemoryError が発生するリスクを下げるためには，見積もり値にバッファを持たせた値を-XX:MaxMetaspaceSize と-XX:CompressedClassSpaceSize に設定してください。Metaspace 領域の使用量がこの値を超えるまでは Metaspace 領域の OutOfMemoryError は発生しません。
- Metaspace 領域に起因する FullGC の発生リスクを下げる
  見積もり時に想定していなかった Metaspace 領域の使用量の増加によって，Metaspace 領域に起因する FullGC が発生するリスクを下げるためには，見積もり値にバッファを持たせた値を-XX:MetaspaceSize，-XX:MaxMetaspaceSize，-XX:CompressedClassSpaceSize に設定してください。Metaspace 領域の使用量がこの値を超えるまでは Metaspace 領域に起因する FullGC は発生しません。

2.2.8 拡張 verbosegc 情報を使用した FullGC の要因の分析方法

SerialGC 使用時の拡張 verbosegc 情報を使用した FullGC の要因の分析方法について説明します。拡張 verbosegc 情報は，JavaVM オプションである-XX:+HitachiVerboseGC オプションを指定することによって出力できる日立 JavaVM のログ情報です。チューニングに役立つ情報のほか，障害要因の分析にも役立つ情報が出力されます。

チューニング実行時に拡張 verbosegc 情報を参照することで，次の情報を確認できます。

- GC 実行前と実行後の各領域の使用メモリサイズ
- GC が発生した要因

また，-XX:+HitachiVerboseGC とほかの JavaVM オプションを組み合わせることによって，さらに詳細な情報が出力できます。-XX:+HitachiVerboseGC オプション，およびそのほかの JavaVM 拡張オプションの詳細については，「5.2　日立 JavaVM 拡張オプションの詳細」を参照してください。

(1)　拡張 verbosegc 情報の出力形式の概要

拡張 verbosegc 情報は，CopyGC が発生した場合と，FullGC が発生した場合に出力されます。

CopyGC が発生すると，GC の種類として「GC」と出力されます。また，FullGC が発生すると，GC の種類として「Full GC」と出力されます。種類に続いて，それぞれの領域の「＜GC 前のメモリサイズ＞->＜GC 後のメモリサイズ＞（＜確保済み領域サイズ＞）」が出力されます。

以降に，FullGC が発生した場合の拡張 verbosegc 情報の出力例を示します。拡張 verbosegc 情報には，ほかにも GC の発生要因や GC スレッドの CPU 時間も出力されます。

拡張 verbosegc 情報の出力形式の詳細，およびそれぞれのオプションの詳細については，「5.　日立 JavaVM 起動オプション」を参照してください。

(2)　FullGC 発生時の拡張 verbosegc 情報の出力例

ここでは，FullGC の発生時の拡張 verbosegc 情報の出力例を示します。

(a)　New 領域メモリサイズに対して Tenured 領域が不足した場合

New 領域（Eden 領域と Survivor 領域の合計）で使用しているメモリサイズが Tenured 領域の最大値に対する未使用メモリサイズを上回った場合の，拡張 verbosegc 情報の出力例を次に示します。**太字**の部分が FullGC 発生の要因を示す個所です。

```
…
[VGC]<Wed May 11 23:12:05 2005>[GC 27340K->27340K(32704K), 0.0432900 secs][DefNew::Eden: 344
0K->0K(3456K)][DefNew::Survivor: 58K->58K(64K)][Tenured: 23841K->27282K(29184K)][Metaspace:
3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][c
ause:ObjAllocFail][User: 0.0156250 secs][Sys: 0.0312500 secs]
[VGC]<Wed May 11 23:12:05 2005>[Full GC 30780K->30780K(32704K), 0.2070500 secs][DefNew::Eden
: 3440K->1602K(3456K)][DefNew::Survivor: 58K->0K(64K)][Tenured: 27282K->29178K(29184K)][Meta
space: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 3
88K)][cause:ObjAllocFail][User: 0.0156250 secs][Sys: 0.0312500 secs]
…
```

この出力例からは，次のことが分かります。

- New 領域で使用しているメモリサイズ（3440K ＋ 58K ＝ 3498K）が，Tenured 領域の最大値に対する未使用メモリサイズ（29184K－27282K ＝ 1902K）を上回りました。
- FullGC の要因は，オブジェクトアロケーションの失敗です。

(b) CopyGC 時に Tenured 領域へのオブジェクトの移動に失敗した場合

CopyGC の実施の結果，New 領域（Eden 領域と Survivor 領域の合計）から Tenured 領域へのオブジェクトの移動に失敗した場合の，拡張 verbosegc 情報の出力例を次に示します。**太字**の部分が FullGC 発生の要因を示す個所です。

```
…
[VGC]<Thu Oct 20 11:04:42 2011>[GC 26418K->26418K (29696K), 0.0000000 secs][DefNew::Eden:818
8K->8188K(8192K)][DefNew::Survivor: 1021K->1021K(1024K)][Tenured:17208K->17208K (20480K)][Me
taspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K,
 388K)][cause:ObjAllocFail][User: 0.0000000 secs][Sys: 0.0000000 secs][IM: 877K, 1104K, 0K][
TC: 9][DOE: 0K, 0]
[VGC]<Thu Oct 20 11:04:42 2011>[Full GC 26418K->6450K(29696K), 0.0156250 secs][DefNew::Eden:
8188K->0K(8192K)][DefNew::Survivor:1021K->0K(1024K)][Tenured:17208K->6450K(20480K)][Metaspac
e: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)
][cause:PromotionFail][User: 0.0156250 secs][Sys: 0.0000000 secs][IM: 925K, 1104K, 0K][TC: 9
][DOE: 0K, 0]
```

この出力例からは，次のことが分かります。

- FullGC の要因は，CopyGC による，New 領域から Tenured 領域へのオブジェクトの移動の失敗です。

(c) オブジェクトアロケーション時に Tenured 領域が不足した場合

アロケーションしたいメモリサイズ（new で作成する Java オブジェクトのサイズ）が Tenured 領域の未使用メモリサイズを上回る場合の，拡張 verbosegc 情報の出力例を次に示します。**太字**の部分が FullGC 発生の要因を示す個所です。

```
…
[VGC]<Wed May 11 23:53:18 2005>[GC 28499K->28490K(32704K), 0.0540590 secs][DefNew::Eden: 808
K->0K(3456K)][DefNew::Survivor: 64K->62K(64K)][Tenured: 27626K->28428K(29184K)][Metaspace: 3
634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][ca
use:ObjAllocFail][User: 0.0156250 secs][Sys: 0.0312500 secs]
[VGC]<Wed May 11 23:53:18 2005>[Full GC 28490K->8959K(32704K), 0.1510380 secs][DefNew::Eden:
 0K->0K(3456K)][DefNew::Survivor: 62K->0K(64K)][Tenured: 28428K->8959K(29184K)][Metaspace: 3
634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][ca
use:ObjAllocFail][User: 0.0156250 secs][Sys: 0.0312500 secs]
…
```

この出力例からは，次のことが分かります。

- Tenured 領域の未使用メモリサイズ（29184K－28428K ＝ 756K）を上回るメモリサイズの Java オブジェクトを，new で作成しようとしました。
- FullGC の要因は，オブジェクトアロケーションの失敗です。

(d) Tenured 領域の未使用メモリサイズが 10,000 バイトを下回った場合

CopyGC 実施の結果，確保済み Tenured 領域の未使用メモリサイズが 10,000 バイトを下回った場合の，拡張 verbosegc 情報の出力例を次に示します。**太字**の部分が FullGC 発生の要因を示す個所です。

```
…
[VGC]<Fri May 25 15:21:33 2007>[GC 15436K->15416K(19840K), 0.0111825 secs][DefNew::Eden: 441
3K->0K(4416K)][DefNew::Survivor: 512K->509K(512K)][Tenured: 10511K->14906K(14912K)][Metaspac
e: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)
][cause:ObjAllocFail][User: 0.0000000 secs][Sys: 0.0000000 secs]
[VGC]<Fri May 25 15:21:33 2007>[Full GC 15416K->8622K(19840K), 0.0284614 secs][DefNew::Eden:
 0K->0K(4416K)][DefNew::Survivor: 509K->0K(512K)][Tenured: 14906K->8622K(14912K)][Metaspace:
 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][
cause:ObjAllocFail][User: 0.0312500 secs][Sys: 0.0000000 secs]
…
```

この出力例からは，次のことが分かります。

- 1 行目の CopyGC で New 領域から Tenured 領域にオブジェクトが移動したことによって，Tenured 領域の使用済みメモリサイズが 10,511 キロバイトから 14,906 キロバイトに増加しました。これによって，確保済み Tenured 領域の未使用メモリサイズが 14,912 キロバイト－14,906 キロバイト＝ 6 キロバイトとなり，10,000 バイト（約 10 キロバイト）を下回りました。
- 1 行目の CopyGC の原因は，オブジェクトアロケーションの失敗です。1 行目の CopyGC と 2 行目の FullGC は，Java プログラムに制御が戻る前に連続して発生します。

(e) CopyGC 実施時に Tenured 領域の拡張が発生した場合

CopyGC 実施時の Tenured 領域へのオブジェクトの移動によって，確保済み Tenured 領域の拡張が発生した場合の，拡張 verbosegc 情報の出力例を次に示します。**太字**の部分が FullGC 発生の要因を示す個所です。

```
…
[VGC]<Fri May 25 15:42:00 2007>[GC 12745K->10151K(15872K), 0.0048346 secs][DefNew::Eden: 441
6K->0K(4416K)][DefNew::Survivor: 137K->512K(512K)][Tenured: 8192K->9639K(10944K)][Metaspace:
 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][
cause:ObjAllocFail][User: 0.0156250 secs][Sys: 0.0000000 secs]
[VGC]<Fri May 25 15:42:00 2007>[GC 14563K->14536K(19072K), 0.0104957 secs][DefNew::Eden: 441
2K->0K(4416K)][DefNew::Survivor: 512K->510K(512K)][Tenured: 9639K->14026K(14144K)][Metaspace
: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)]
[cause:ObjAllocFail][User: 0.0156250 secs][Sys: 0.0000000 secs]
[VGC]<Fri May 25 15:42:00 2007>[Full GC 14536K->8610K(19072K), 0.0287254 secs][DefNew::Eden:
 0K->0K(4416K)][DefNew::Survivor: 510K->0K(512K)][Tenured: 14026K->8610K(14144K)][Metaspace:
 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][
cause:ObjAllocFail][User: 0.0312500 secs][Sys: 0.0000000 secs]
…
```

この出力例からは，次のことが分かります。

- 2 行目の CopyGC で New 領域から Tenured 領域へのオブジェクトが移動したことによって，Tenured 領域が最低でも 14,026 キロバイト以上必要になりました。このため，確保済み Tenured 領域サイズが 10,944 キロバイトから 14,144 キロバイトに拡張されました。
- 2 行目の CopyGC の原因は，オブジェクトアロケーションの失敗です。2 行目の CopyGC と 3 行目の FullGC は，Java プログラムに制御が戻る前に連続して発生します。

(f)　java.lang.System.gc()メソッドが実行された場合

アプリケーション内で java.lang.System.gc()メソッドが実行された場合の，拡張 verbosegc 情報の出力例を次に示します。**太字**の部分が FullGC 発生の要因を示す個所です。

```
…
[VGC]<Mon Apr 18 20:36:29 2005>[Full GC 330K->150K(3520K), 0.0387690 secs][DefNew::Eden: 330
K->0K(2048K)][DefNew::Survivor: 0K->0K(64K)][Tenured: 0K->150K(1408K)][Metaspace: 3634K(4492
K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][cause:System.gc][User: 0.0156250 secs][Sys: 0.0312500 secs]
…
```

この出力例からは，次のことが分かります。

- FullGC の要因は，アプリケーション内での java.lang.System.gc()メソッド呼び出しです。

(g)　Metaspace 領域へのオブジェクトアロケーションに失敗した場合

Metaspace 領域にアロケーションしたいメモリサイズが確保済み Metaspace 領域の未使用メモリサイズを上回る場合の，拡張 verbosegc 情報の出力例を次に示します。**太字**の部分が FullGC 発生の要因を示す個所です。

```
…
[VGC]<Wed Jan 07 01:56:13 2015>[Full GC 11273K->6037K(15872K), 0. 0060004 secs][DefNew::Eden
: 442K->0K(4416K)][DefNew::Survivor: 512K->0K(512K)][Tenured: 10319K->6037K(10944K)][Metaspa
ce: 22811K(24520K, 24576K)->22811K(24520K, 24576K)][class space: 10758K(10988K, 11008K)->107
58K(10988K, 11008K)][cause:MetaspaceAllocFail][User: 0.0312002 secs][Sys: 0.0000000 secs]
…
```

この出力例からは，次のことが分かります。

- Metaspace 領域にアロケーションしようとしたメモリサイズが，確保済み Metaspace 領域の未使用メモリサイズ（24576 キロバイト－22811 キロバイト＝ 1765 キロバイト）を上回りました。
- FullGC の要因は，Metaspace 領域のアロケーションの失敗です。

(h)　javagc コマンド実行による FullGC が発生した場合

javagc コマンドを実行した場合の，拡張 verbosegc 情報の出力例を次に示します。**太字**の部分が FullGC 発生の要因を示す個所です。

```
…
[VGC]<Mon Apr 18 21:46:50 2005>[Full GC 369K->189K(3520K), 0.0403010 secs][DefNew::Eden: 369
K->0K(2048K)][DefNew::Survivor: 0K->0K(64K)][Tenured: 0K->189K(1408K)][Metaspace: 3634K(4492
K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][cause:JavaGC Command][User: 0.0156250 secs][Sys: 0.0312500 secs]
…
```

この出力例からは，次のことが分かります。

- FullGC の要因は，javagc コマンド実行です。

(i)　jheapprof コマンド実行による FullGC が発生した場合

jheapprof コマンドを実行した場合の，拡張 verbosegc 情報の出力例を次に示します。**太字**の部分が FullGC 発生の要因を示す個所です。

```
…
[VGC]<Mon Apr 18 21:46:50 2005>[Full GC 369K->189K(3520K), 0.0403010 secs][DefNew::Eden: 369
K->0K(2048K)][DefNew::Survivor: 0K->0K(64K)][Tenured: 0K->189K(1408K)][Metaspace: 3634K(4492
K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][cause:JHeap
Prof Command][User: 0.0156250 secs][Sys: 0.0312500 secs]
…
```

この出力例からは，次のことが分かります。

- FullGC の要因は，jheapprof コマンド実行です。

2.3　G1GC の仕組みと JavaVM のメモリチューニング

ここでは，G1GC の仕組みと JavaVM のメモリチューニングの方法について説明します。

2.3.1　G1GC の仕組み

G1GC の仕組みを説明します。

(1)　G1GC の概要

GC は，プログラムが使用し終わったメモリ領域を自動的に回収して，ほかのプログラムが利用できるようにするための技術です。

GC の実行中は，プログラムの処理が停止します。このため，GC を適切に実行できるかどうかが，システムの処理性能に大きく影響します。

プログラムの中で new によって作成された Java オブジェクトは，JavaVM が管理するメモリ領域に格納されます。Java オブジェクトが作成されてから不要になるまでの期間を，**Java オブジェクトの寿命**といいます。

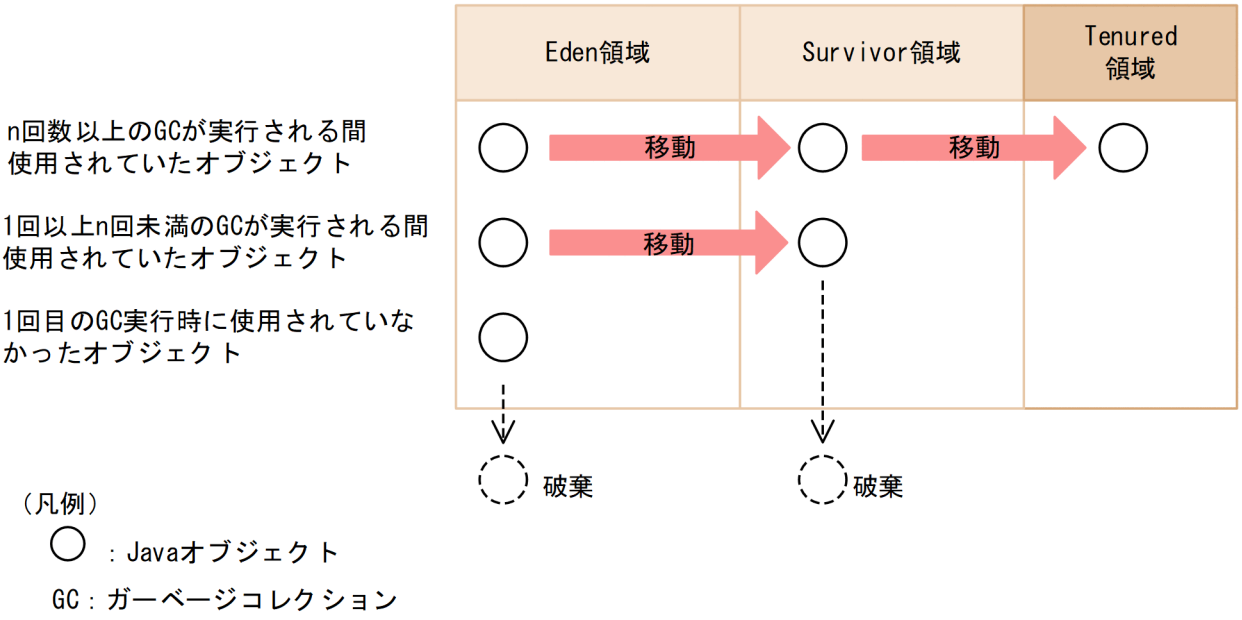
Java オブジェクトには，寿命の短いオブジェクトと寿命の長いオブジェクトがあります。サーバサイドで動作する Java アプリケーションの場合，リクエストやレスポンス，トランザクション管理などで，多くの Java オブジェクトが作成されます。これらの Java オブジェクトは，その処理が終わると不要になる，寿命が短いオブジェクトです。一方，アプリケーションの動作中使用され続ける Java オブジェクトは，寿命が長いオブジェクトです。

効果的な GC を実行するためには，寿命の短いオブジェクトに対して GC を実行して，効率良くメモリ領域を回収することが必要です。また，繰り返し使用される寿命の長いオブジェクトに対する不要な GC を抑止することが，システムの処理性能の低下防止につながります。これを実現するのが，**世代別 GC** です。

世代別 GC では，Java オブジェクトを，寿命が短いオブジェクトが格納される New 領域と，寿命が長いオブジェクトが格納される Tenured 領域に分けて管理します。New 領域はさらに，new によって作成されたばかりのオブジェクトが格納される Eden 領域と，1 回以上の GC の対象になり，回収されなかったオブジェクトが格納される Survivor 領域に分けられます。New 領域内で一定回数以上の GC の対象になった Java オブジェクトは，長期間必要な Java オブジェクトと判断され，Tenured 領域に移動します。

世代別 GC で管理するメモリ空間と Java オブジェクトの概要を次の図に示します。

図 2-12　世代別 GC で管理するメモリ空間と Java オブジェクトの概要



G1GC の世代別 GC で実行される GC には，次の 3 種類があります。

- YoungGC

  Eden 領域と Survivor 領域を対象にした GC です。Java オブジェクトの作成によって，Eden 領域を使い切ると発生します。

- MixedGC

  Eden 領域と Survivor 領域と Tenured 領域の一部を対象にした GC です。Concurrent Marking と呼ばれるオブジェクトの解析処理に基づき発生します。Java オブジェクトの作成によって，Eden 領域を使い切ると発生します。

- FullGC

  Tenured 領域も含む，JavaVM 固有領域全体を対象にした GC です。Tenured 領域や Metaspace 領域，Humongous 領域を新たに確保できないと発生します。

一般的に，YoungGC と MixedGC の方が，FullGC よりも短い時間で処理できます。

次に，GC の処理について，ある Java オブジェクト（オブジェクト A）を例にして説明します。

Eden 領域で実行される処理

オブジェクト A の作成後，1 回目の YoungGC または MixedGC が実行された時点で使用されていない場合，オブジェクト A は破棄されます。

1 回目の YoungGC または MixedGC が実行された時点で使用されていた場合，オブジェクト A は Eden 領域から Survivor 領域に移動します。

Survivor 領域で実行される処理

Survivor 領域に移動したオブジェクト A は，そのあと何回か YoungGC または MixedGC が実行されると，Survivor 領域から Tenured 領域に移動します。移動する回数のしきい値は，JavaVM オプションや Java ヒープの利用状況によって異なります。「図 2-12」では，しきい値を n 回としています。

Survivor 領域への移動後，n 回目未満の YoungGC または MixedGC が実行された時点でオブジェクト A が使用されていなかった場合，オブジェクト A はその YoungGC または MixedGC で破棄されます。

Tenured 領域で実行される処理

オブジェクト A が Tenured 領域に移動した場合，そのあとの YoungGC でオブジェクト A が破棄されることはありません。YoungGC は，Eden 領域と Survivor 領域だけを対象としているためです。MixedGC が発生した場合は，Tenured 領域内の一部のオブジェクトが破棄されます。

MixedGC で破棄されるオブジェクトの数よりも，Tenured 領域に移動するオブジェクトの数が多い場合，Tenured 領域の使用サイズは増加します。新たに Tenured 領域を確保できなくなると，FullGC が発生します。

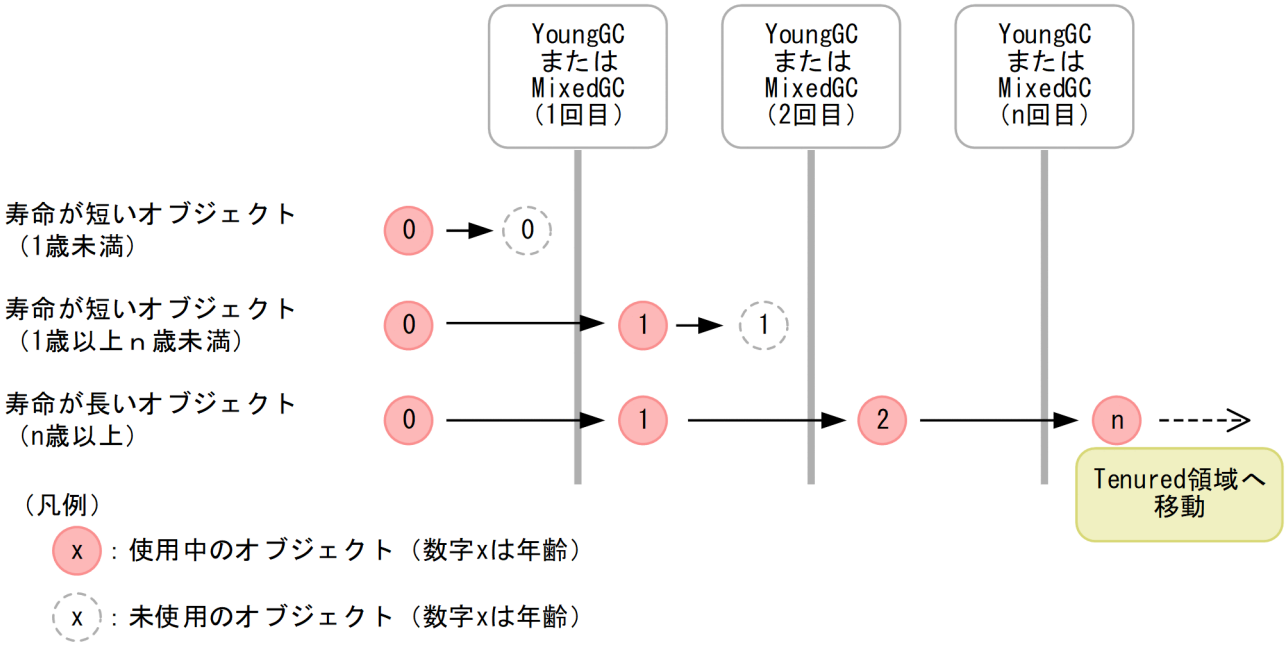
JavaVM のチューニングでは，JavaVM オプションでそれぞれのメモリ空間のサイズや割合を適切に設定することで，不要なオブジェクトが Tenured 領域に移動することを抑止します。これによって，FullGC が頻発することを防ぎます。

(2)　オブジェクトの寿命と年齢の関係

オブジェクトが YoungGC または MixedGC の対象になった回数を**オブジェクトの年齢**といいます。

オブジェクトの寿命と年齢の関係を次の図に示します。

図 2-13　オブジェクトの寿命と年齢の関係



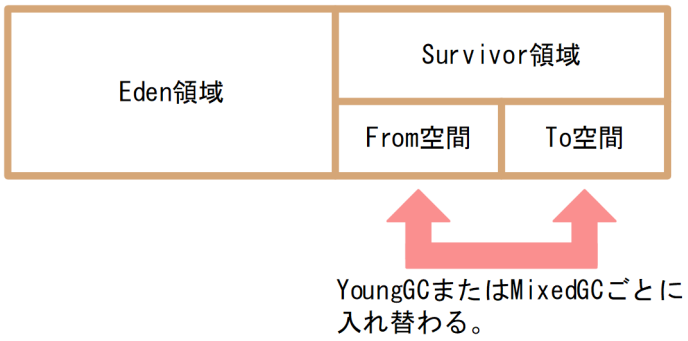
アプリケーションが開始して初期化処理が完了したあとで，何度かの YoungGC または MixedGC が実行されると，長期間必要になる寿命の長いオブジェクトは Tenured 領域に移動します。このため，アプリケーションの開始後しばらくすると，Java ヒープの状態は安定し，作成される Java オブジェクトとしては，寿命が短いオブジェクトが多くなります。特に，New 領域のチューニングが適切にできている場合，Java ヒープが安定したあとの大半のオブジェクトは，1 回目の YoungGC または MixedGC で回収される，寿命が短いオブジェクトになります。

(3) New 領域を対象とした GC の仕組み

JavaVM では，YoungGC または MixedGC の対象になる New 領域のメモリ空間を，Eden 領域，Survivor 領域に分けて管理します。さらに，Survivor 領域は，From 空間と To 空間に分けられます。From 空間と To 空間は，同じメモリサイズです。

New 領域の構成を次の図に示します。

図 2-14 New 領域の構成



Eden 領域は，new によって作成されたオブジェクトが最初に格納される領域です。プログラムで new が実行されると，Eden 領域のメモリが確保されます。

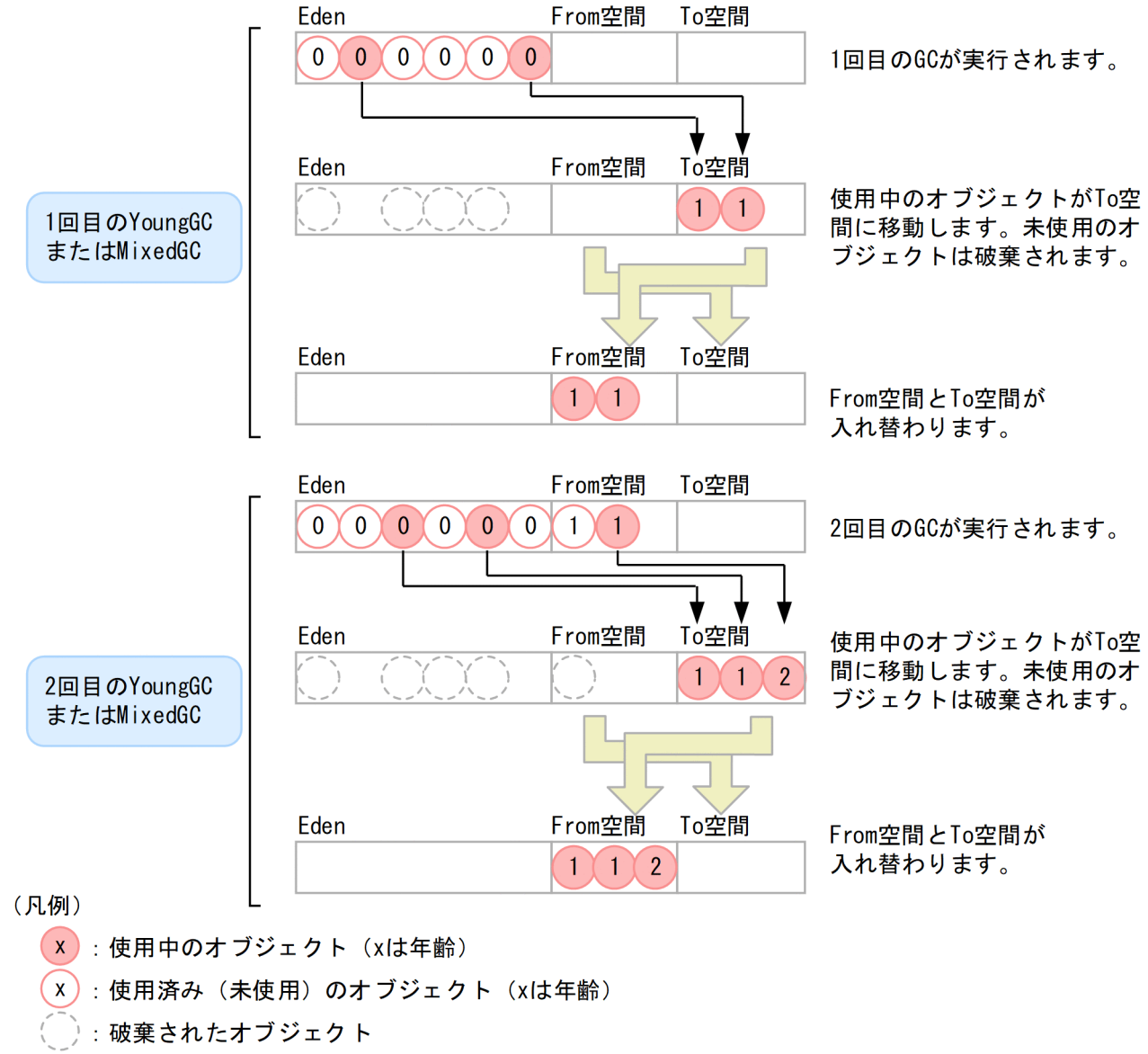
Eden 領域がいっぱいになると，YoungGC または MixedGC が実行され，次の処理が実行されます。

1. Eden 領域および Survivor 領域の From 空間にある Java オブジェクトのうち，使用中の Java オブジェクトが，Survivor 領域の To 空間にコピーされます。使用されていない Java オブジェクトは破棄されます。
2. Survivor 領域の To 空間と From 空間が入れ替わります。

この結果，Eden 領域と To 空間は空になり，使用中のオブジェクトは From 空間に存在することになります。

YoungGC または MixedGC 実行時に発生するオブジェクトの移動を次の図に示します。

図 2-15　CopyGC 実行時に発生するオブジェクトの移動

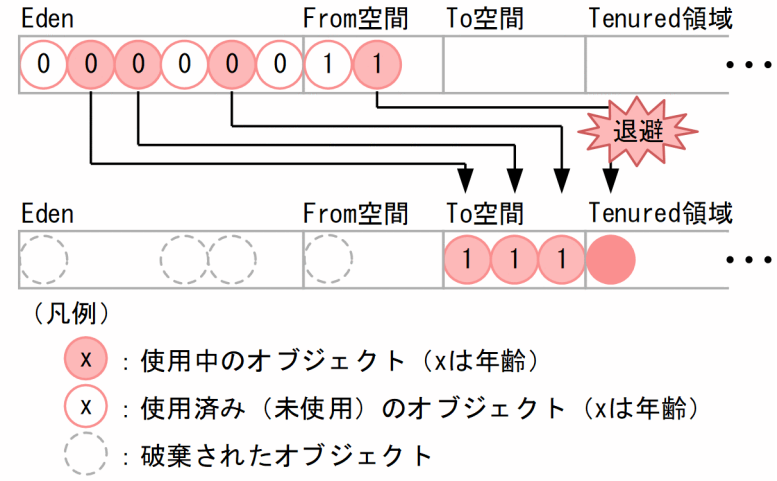


このようにして，使用中のオブジェクトは，YoungGC または MixedGC が発生するたびに，From 空間と To 空間を行ったり来たりします。ただし，寿命の長いオブジェクトを行き来させ続けると，移動処理の負荷などが問題になります。これを防ぐために，From 空間と To 空間で Java オブジェクトを入れ替える回数にしきい値を設定して，年齢がしきい値に達した Java オブジェクトは Tenured 領域に移動させるようにします。

(4)　オブジェクトの退避

年齢がしきい値に達していない Java オブジェクトが Tenured 領域に移動することを，退避といいます。退避は，YoungGC または MixedGC 実行時に Eden 領域および From 空間で使用中のオブジェクトが多くなり，移動先である To 空間のメモリサイズが不足する場合に発生します。この場合，To 空間に移動できなかったオブジェクトが，Tenured 領域に移動します。

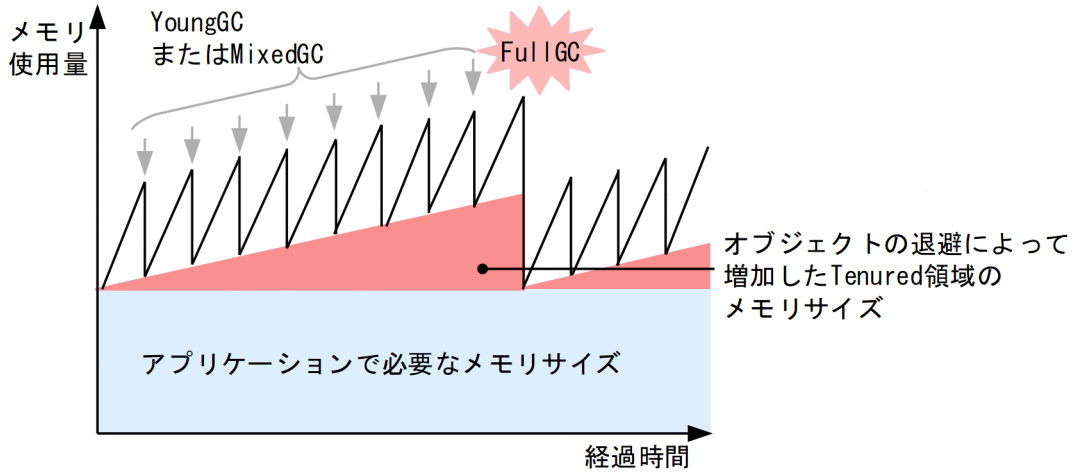
図 2-16　オブジェクトの退避



オブジェクトの退避が発生した場合，Tenured 領域に本来格納されないはずの寿命の短いオブジェクトが格納されます。これが繰り返されると，YoungGC または MixedGC で回収されるはずのオブジェクトがメモリ空間に残っていくため，Java ヒープのメモリ使用量が増加していき，最終的には FullGC が発生します。

オブジェクトの退避が発生した場合のメモリ使用量の変化について，次の図に示します。

図 2-17　オブジェクトの退避が発生した場合のメモリ使用量の変化



FullGC では，システムが数秒から数十秒停止することがあります。

したがって，メモリ空間の構成とメモリサイズを検討するときには，オブジェクトの退避が発生しないように，Eden 領域と Survivor 領域のバランスを検討する必要があります。

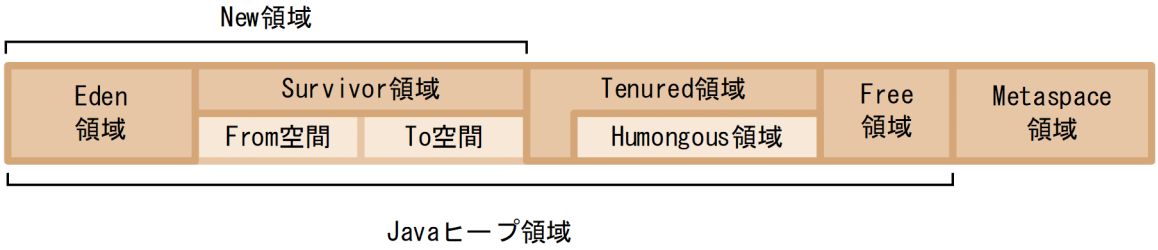
(5)　メモリ空間とリージョンの関係

G1GC で管理する Java ヒープ領域は，オブジェクトの寿命が短いオブジェクトが格納される New 領域と，寿命が長いオブジェクトが格納される Tenured 領域に分けて管理します。このうち New 領域はさらに 2 つの領域に分けて管理されており，作成されたばかりのオブジェクトが格納される Eden 領域と，1

回以上の GC の対象になり，回収されなかったオブジェクトが格納される Survivor 領域に分けられます。New 領域内で一定回数以上の GC の対象になったオブジェクトは，寿命が長いオブジェクトと判断され，Tenured 領域に移動します。

アプリケーション実行中の G1GC で管理するメモリ空間の構成を次の図に示します。なお，Eden 領域，Survivor 領域，Tenured 領域，Free 領域を合わせた領域を Java ヒープ領域といいます。

図 2-18　アプリケーション実行中の G1GC で管理するメモリ空間の構成



G1GC では，Tenured 領域があらかじめ割り当てられていません。Tenured 領域にオブジェクトを移動させる場合，Free 領域のリージョンを Tenured 領域に割り当ててオブジェクトを移動します。また，G1GC では GC 後に New 領域に対して拡張や縮小をします（以降，リサイズと表記します）。リサイズで New 領域を拡張する場合は，Free 領域のリージョンに割り当て，縮小する場合は New 領域のリージョンを回収し，Free 領域に割り当てます。サイズの大きなオブジェクトを作成する場合は，Humongous 領域にオブジェクトが格納されます。

各領域の用途を次に示します。

- New 領域
  寿命が短いオブジェクトが格納される領域です。Eden 領域と Survivor 領域から構成されます。GC 後に次の GC に向けてリサイズされます。詳細については，「(8)　YoungGC」を参照してください。
- Eden 領域
  作成されたオブジェクトが最初に格納される領域です。
- Survivor 領域
  New 領域に格納されていたオブジェクトのうち，GC 実行時に回収されなかったオブジェクトが格納される領域です。Survivor 領域は From 空間と To 空間で構成されます。From 空間と To 空間は同じサイズです。
- Tenured 領域
  寿命が長いオブジェクトが格納される領域です。Survivor 領域で指定回数を超えて GC 実行対象になり，回収されなかったオブジェクトが移動します。
- Humongous 領域
  Tenured 領域の一部でサイズの大きなオブジェクトを格納する領域です。Humongous 領域のオブジェクトは，FullGC または Concurrent Cleanup で回収されます。Concurrent Cleanup の詳細については，「(9)　Concurrent Marking（CM）」を参照してください。
- Free 領域

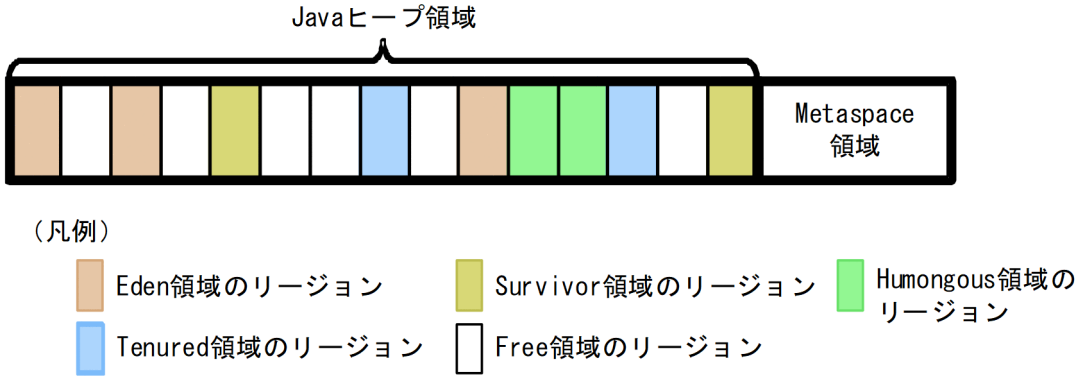
New 領域や Tenured 領域に割り当てられていない領域です。Tenured 領域のリージョンが足りなくなった場合や GC 後の New 領域をリサイズする場合に，New 領域や Tenured 領域に割り当てられます。

- Metaspace 領域

  ロードされた class などの情報を格納する領域です。なお，Metaspace 領域はリージョン単位で管理されていません。Metaspace 領域のオブジェクトは FullGC でだけ回収されます。

「図 2-18」は Java ヒープ領域の各領域を連続領域として表現していますが，実際には各領域は連続領域として確保されていません。G1GC では Java ヒープ領域をリージョンと呼ばれるブロック単位で管理しており，次の図のように領域が分散して確保されています。

図 2-19　実際のメモリ空間の構造



ここではそれぞれの領域のリージョンを領域名＋リージョンと表記します。例えば，Eden 領域のリージョンを Eden リージョン，New 領域のリージョンは New リージョンと表記します。

(6)　リージョンの使われ方

オブジェクトが作成されると，最初に Eden リージョンに格納されます。1 つのオブジェクトのサイズが，リージョンサイズの半分を超えている大きいオブジェクト※の場合は，1 つのオブジェクトに対し，連続した複数のリージョンが割り当てられ格納されます。連続した複数のリージョンは Free 領域から確保され，Humongous 領域に割り当てられます。Humongous 領域は FullGC または Concurrent Cleanup でだけ回収の対象となるため，サイズの大きなオブジェクトが多数作成されるような場合には，G1GC は不向きとなります。詳細については，「2.3.2(3)(a)　FullGC の発生を抑止する考え方」を参照してください。

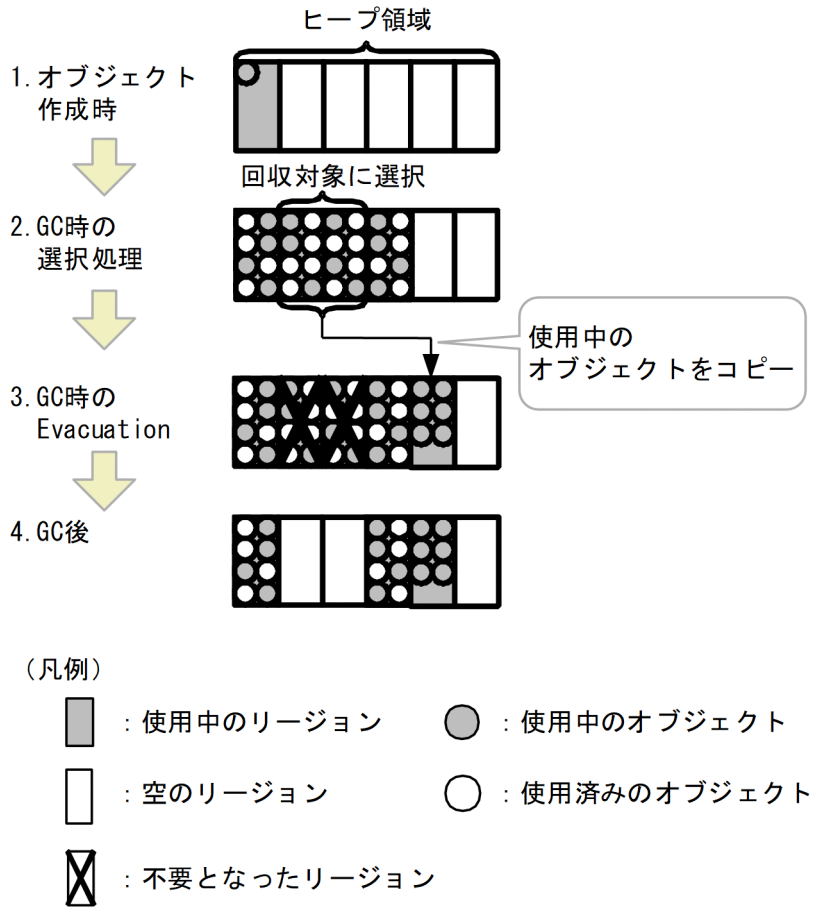
また，1 つのリージョンのサイズは，起動時に Java ヒープ領域の初期サイズの比率から求められ，起動中は固定です。1 つのリージョンの最小サイズは 1MB であり，最大サイズは 32MB です。

注※

ここでいうサイズの大きな 1 つのオブジェクトとは，多数のインスタンスフィールドを持つオブジェクトや長大な配列を持つオブジェクトのことを指します。リージョンのサイズが 1MB の場合，512KB を超えるようなオブジェクトを指し，例えば約 13 万個の int 型のインスタンスフィールドを持つようなオブジェクトや約 50 万個の要素数のバイト配列を持つオブジェクトなどが当てはまります。

リージョンの使用から回収までの流れを次の図に示します。

図 2-20　リージョンの使われ方



この図はリージョンの使われ方を示した図です。図の 1.～4.の処理の詳細を次に示します。

1. オブジェクト作成時

   オブジェクトが作成される場合，Eden リージョンにオブジェクトを格納します。サイズの大きなオブジェクトを作成する場合は，Free リージョンを Humongous リージョンに割り当てオブジェクトを格納します。Eden リージョンに空き領域がなくなると，新しい Eden リージョンにオブジェクトを格納し，Humongous リージョンに空き領域がなくなると，新たに Free リージョンから Humongous リージョンを割り当て，オブジェクトを格納します。

2. GC 時の選択処理

   GC の要件を満たすと，実行する GC 方式の基準に従って，対象とするリージョンを選択します。

3. GC 時の Evacuation

   回収の対象となったリージョン内で使用中のオブジェクトは，ほかのリージョンにコピーされます。この処理を Evacuation といいます。Evacuation 後のリージョンは，使用済みのオブジェクトとコピー済みである不要となったオブジェクトだけ格納されているため，リージョン単位で回収されます。

4. GC 後

   回収されたリージョンは Free リージョンとなり，再利用されます。

(7)　G1GC で実行される GC

G1GC の世代別 GC で実行される GC には，次の 3 種類があります。

1. YoungGC

   New 領域を対象とする GC です。YoungGC には YoungGC(normal)と YoungGC 中にマーキングをする YoungGC(initial-mark)があります。詳細については，「(8)　YoungGC」を参照してください。なお，単に YoungGC と表記した場合，YoungGC(normal)と YoungGC(initial-mark)の両方に当てはまる事項となります。また，YoungGC(normal)と YoungGC(initial-mark)を区別して表す場合は，“(normal)”と“(initial-mark)”を明記します。ただし，ログファイルの GC 種別には YoungGC(normal)は YoungGC という種別で出力します。YoungGC は Java オブジェクトの作成によって，Eden 領域を使い切ると発生します。

2. MixedGC

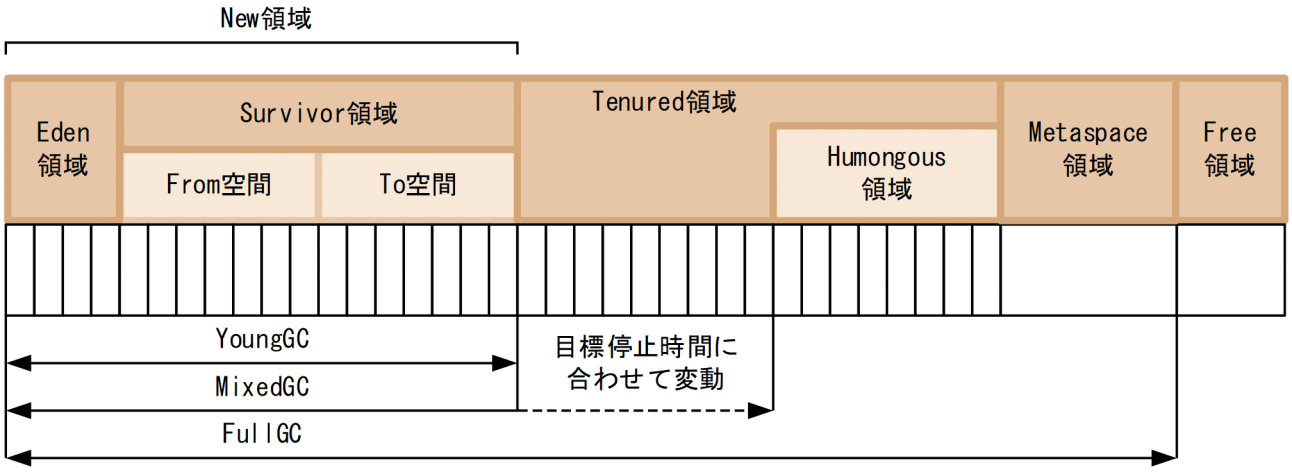
   New 領域と Tenured 領域を対象とする GC です。Tenured リージョンは目標停止時間に合わせて部分的な範囲を対象とします。詳細については，「(10)　MixedGC」を参照してください。MixedGC は YoungGC 同様，Java オブジェクトの作成によって，Eden 領域を使い切ると発生しますが，Concurrent Marking と呼ばれるオブジェクトが使用中かどうかの解析処理の結果に基づいて発生します。詳細については，「2.3.1(9)　Concurrent Marking（CM)」を参照してください。そのため，解析が十分にされていない場合や解析の結果 MixedGC の効果が低いと予測される場合は，YoungGC(normal)が発生します。

3. FullGC

   Tenured 領域や Metaspace 領域，Humongous 領域を含む，JavaVM 固有領域全体を対象にした GC です。詳細については，「(11)　FullGC」を参照してください。Tenured 領域や Metaspace 領域，Humongous 領域を確保できなかった場合に発生します。

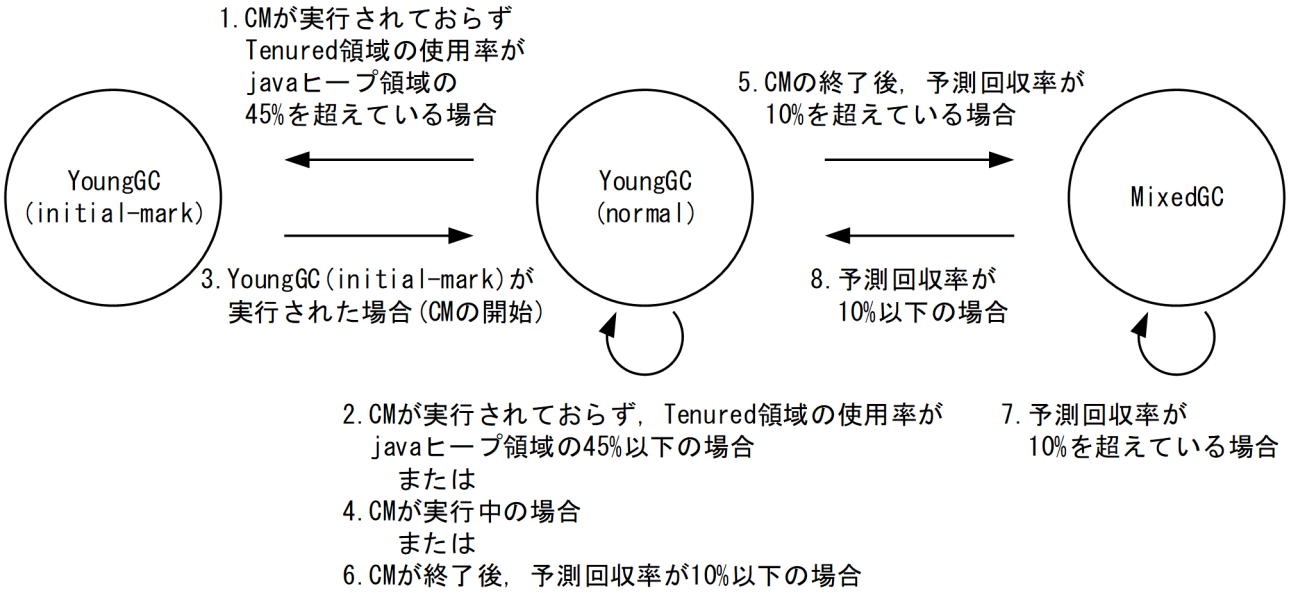
それぞれの GC が対象とする領域をまとめると次の図となります。

図 2-21　各 GC の対象範囲



また，各 GC の状態遷移図を次の図に示します。

図 2-22　各 GC の状態遷移図



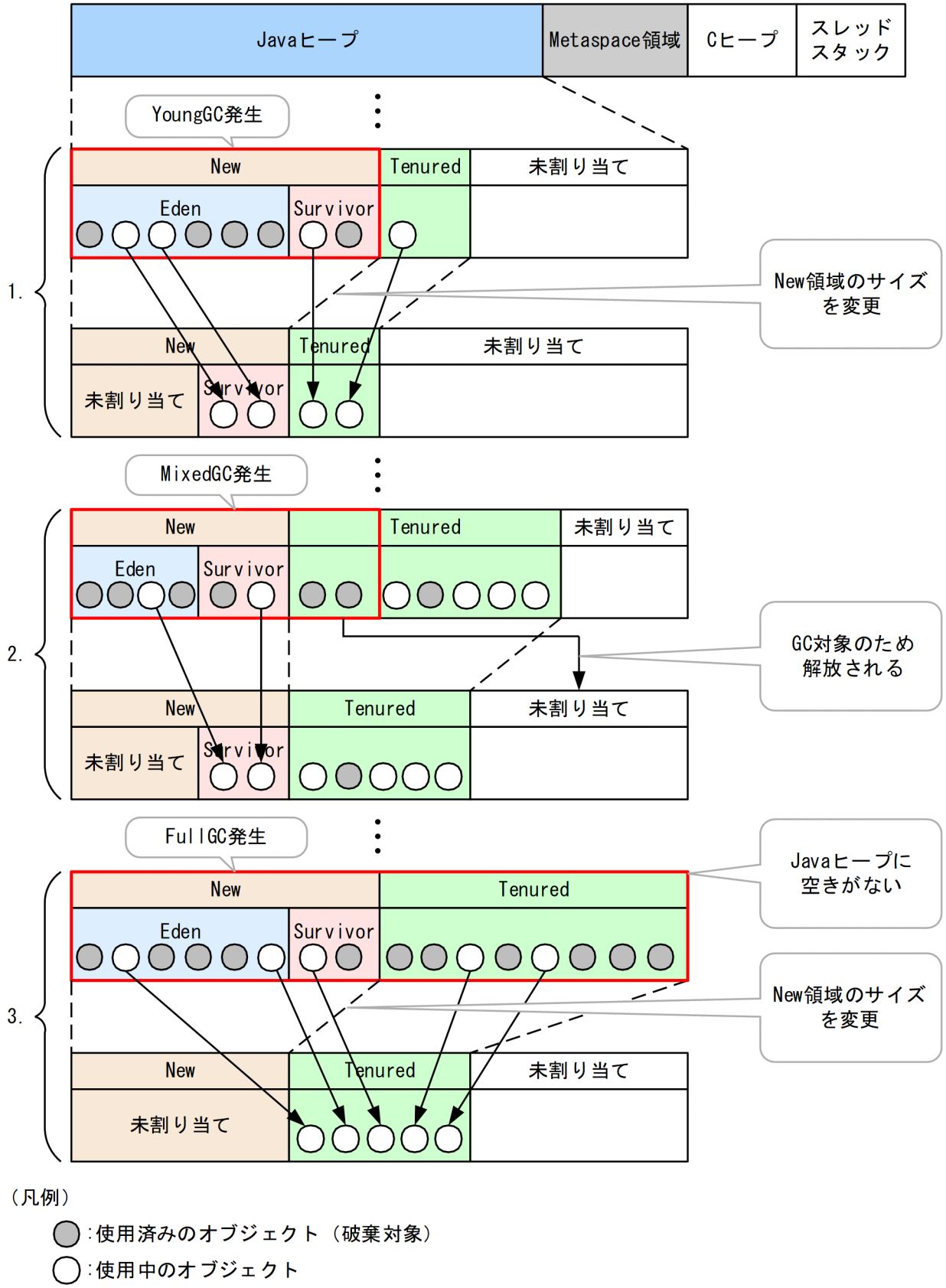
1. YoungGC(normal)実行後，Concurrent Marking（CM）が実行されておらず，Tenured 領域の使用率が Java ヒープ領域の 45%を超えた場合は，次の GC では YoungGC(initial-mark)を実行する状態に遷移します。
2. YoungGC(normal)実行後，CM が実行されておらず，Tenured 領域の使用率が Java ヒープ領域の 45%以下の場合は，次の GC も YoungGC(normal)を実行する状態のままです。
3. YoungGC(initial-mark)実行後，次の GC では YoungGC(normal)と CM を並行して実行する状態に遷移します。
4. CM が実行中の場合は，YoungGC(normal)を実行する状態のままです。
5. CM 終了直後の YoungGC(normal)で予測回収率が 10%を超えている場合，次の GC では MixedGC を実行する状態に遷移します。
6. 予測回収率が 10%以下の場合，次の GC も YoungGC(normal)を実行する状態のままです。
7. MixedGC 実行後，予測回収率が 10%を超えている場合，次の GC では MixedGC を実行する状態のままです。
8. MixedGC 実行後，予測回収率が 10%以下の場合，次の GC では YoungGC(normal)を実行する状態に遷移します。

ただし，次の場合 GC の状態に関係なく，条件を満たすと対応する GC を実行します。

- YoungGC または，MixedGC を実行しても空き領域を確保できなかった場合，FullGC を実行します。FullGC 実行後は YoungGC(normal)を実行する状態に遷移します。
- サイズの大きなオブジェクトを確保時，CM が実行されておらず，Tenured 領域の使用サイズと確保するオブジェクトのサイズの合計が Java ヒープ領域の 45%を超えている場合，YoungGC(initial-mark)を実行します。

次に，G1GC の処理の流れについて，Java オブジェクトを例にして説明します。

図 2-23　G1GC の流れ



1. YoungGC

この図のように New 領域に割り当てたリージョンに空きがなくなると YoungGC が発生します。YoungGC では使用中のオブジェクトは Survivor 領域に割り当てたリージョンに移動し，使用済みのオブジェクトはリージョンごと解放します。また，SerialGC の CopyGC と同様，YoungGC 発生時に使用中のオブジェクトは Survivor 領域に割り当てたリージョン間を移動し続け，ある一定の回数を移動すると Tenured 領域に割り当てたリージョンに移動します。YoungGC 後，上の図のように GC に掛かった時間から，次の GC に掛かる時間を予測し，New 領域のサイズを変更します。この図の場合は，予測した時間より GC に掛かる時間が長いため，New 領域を縮小した場合の例です。

2. MixedGC

Tenured 領域の使用率が増加すると，MixedGC が発生します。MixedGC では New 領域に割り当てたリージョンに加えて，目標停止時間内に収まる範囲で，一部の Tenured 領域に割り当てたリージョンを GC の対象とします。この一部の Tenured 領域に割り当てたリージョンは，アプリケーションと並行して実行しているオブジェクトが使用中かどうかの解析情報に基づき，解放されるサイズが大きいと予測されるリージョンから優先して GC の対象となります。そのため，オブジェクトの情報解析が十分にされていない場合や解析の結果 MixedGC の効果が低い場合は，MixedGC は発生しません。

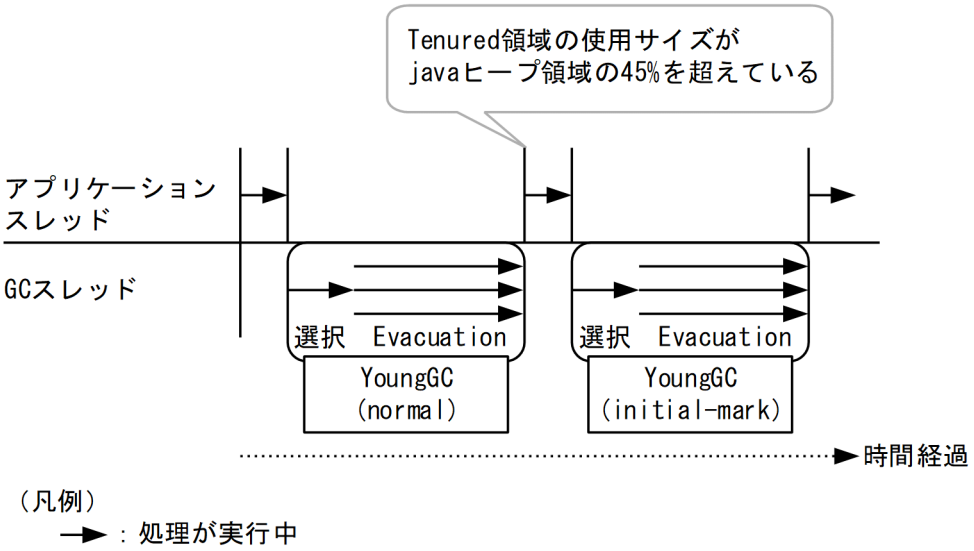
3. FullGC

Java ヒープ内のリージョンに空きがなくなり，MixedGC が発生しない場合，Java ヒープ全体を対象として FullGC が発生します。

(8) YoungGC

YoungGC の流れを次の図に示します。

図 2-24 YoungGC の流れ



(a) 実行契機

1. YoungGC(normal)

Eden 領域にオブジェクトを確保できなかった場合に発生します。また，サイズの大きなオブジェクトを Humongous 領域に確保できなかった場合にも発生します。

2. YoungGC(initial-mark)

   直前に実行された YoungGC(normal)終了時に，Tenured 領域の使用サイズが Java ヒープ領域の 45%を超えていた場合，1.の実行契機を満たすと YoungGC(initial-mark)が実行されます。また，サイズの大きなオブジェクトを確保時，Tenured 領域の使用サイズとオブジェクトの確保サイズの合計サイズが Java ヒープ領域の 45%を超えた場合，YoungGC(initial-mark)が実行されます。

(b) 対象範囲

New 領域

(c) 処理内容

- YoungGC が発生するとシングルスレッドでリージョンの選択処理をし，そのあとマルチスレッドで Evacuation をします。
- YoungGC の Evacuation では Eden 領域と From 空間内の使用中オブジェクトを To 空間または Tenured 領域に移動し，Eden 領域と From 空間を回収します。移動や回収の仕組みは CopyGC の仕組みと同じです。CopyGC の詳細については，「2.2.1(3) CopyGC の仕組み」を参照してください。
- YoungGC では，これまでに発生した GC の GC 停止時間の統計から予測をし，次回の GC の予測停止時間が目標停止時間内に収まるように New 領域のサイズを変更します。
- New 領域は最小サイズと最大サイズが存在し，その範囲内でリサイズをします。New 領域は全 GC で対象となるため，GC 停止時間は New 領域が最小サイズの場合に掛かる GC 停止時間より短くすることはできません。
- CM が終了後の YoungGC では，予測回収サイズが Java ヒープ領域の 10%を超えていた場合，次回の GC を MixedGC にするか判定します。次回の GC に MixedGC が選択された場合，Tenured リージョンを多く対象とするように予測して New 領域のサイズを変更します。
- YoungGC(initial-mark)の Evacuation では Evacuation 中にローカル変数や使用中のオブジェクトから直接参照されているオブジェクトにマーク付けをします。このマーキングの結果は，Concurrent Marking に利用されます。Concurrent Marking の処理の詳細については，「(9) Concurrent Marking（CM)」を参照してください。

(d) 処理結果

Eden 領域：オブジェクトが回収され，空になります。GC 後リサイズされます。

Survivor 領域：From 空間のオブジェクトが回収され，空になります。GC 後リサイズされます。

Tenured 領域：長期間必要と判断されたオブジェクトが Tenured 領域に移動します。

Humongous 領域：変化はありません。

Metaspace 領域：変化はありません。

Free 領域：GC 後のリサイズによって，増減します。

(e)　アプリケーションの停止の有無

停止します。

(f)　ほかの GC との関係

CM：YoungGC 中に実行されません。

MixedGC：YoungGC 中に実行されません。

FullGC：YoungGC 中に実行要件を満たすと，YoungGC を中止して実行されます。

(g)　補足

- 関連オプション

  Evacuation をするスレッド数は-XX:ParallelGCThreads オプションで変更できます。スレッド数を増やすと YoungGC に掛かる時間が小さくなります。また，オプションを指定しない場合，スレッド数は-XX:ParallelGCThreads オプションのデフォルト値が用いられます。-XX:ParallelGCThreads オプションについては，「5.4　日立 JavaVM で指定できる Java HotSpot VM のオプション」の「表 5-9　指定できる Java HotSpot VM のオプション」にある「-XX:ParallelGCThreads」を参照してください。

- 確認方法

  YoungGC の確認はログ上の GC の種別が“YoungGC”または“YoungGC(initial-mark)”であることから確認できます。また，New 領域のリサイズは Eden 領域のサイズ変化と Survivor 領域のサイズ変化から確認できます。

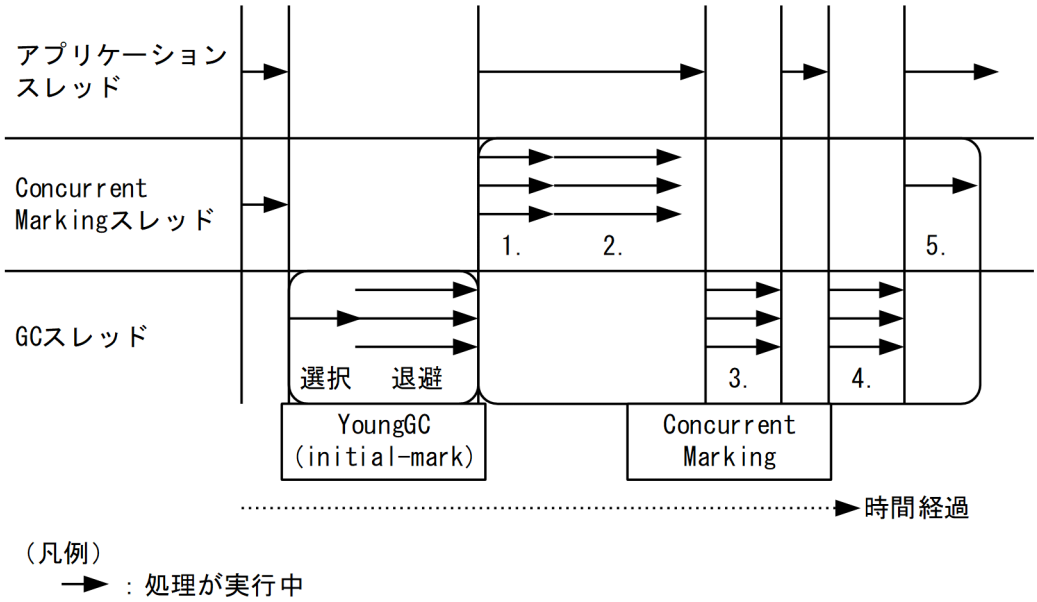
```
[VG1]<Wed Jun 12 11:21:10 2013>[Young GC 899070K/899072K(1048576K)->501755K/501760K(1048576K), 0.0931560 secs][Status:-][G1GC::Eden: 389120K(389120K)->0K(397312K)][G1GC::Survivor: 41984K->41984K][G1GC::Tenured: 459776K->459776K][G1GC::Humongous: 2048K->2048K][G1GC::Free: 609536K->607232K][Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][cause:G1EvacuationPause][RegionSize: 1024K][Target: 0.2000000 secs][Predicted: 0.2495800 secs][TargetTenured: 0K][Reclaimable: 0K(0.00%)][User: 0.0156250 secs][Sys: 0.0312500 secs][IM: 729K, 928K, 0K][TC: 509][DOE: 16K, 171][CCI: 2301K, 49152K, 2304K]
```

  このログの場合 GC 前の New 領域のサイズは 389120K ＋ 41984K ＝ 431104K であり，GC 後の New 領域のサイズは 397312K ＋ 41984K ＝ 439296K であるため，New 領域が拡張されたことが分かります。ログの記述内容や詳細に関しては，「-XX:[+|-]HitachiVerboseGC（拡張 verbosegc 情報出力オプション）」を参照してください。

(9)　Concurrent Marking（CM）

Concurrent Marking の流れを次の図に示します。

図 2-25　Concurrent Marking の流れ



（凡例）
→ ：処理が実行中

(a)　実行契機

CM は大きく次の 5 つの処理から構成されます。それぞれの実行契機について示します。なお，1.～5.の表記はこの図の 1.～5.に対応しています。

1. Concurrent Root Region Scan：
   YoungGC(initial-mark)終了後，実行されます。
2. Concurrent Mark：
   1.の終了後，1.に続けて実行されます。
3. Remark：
   2.の終了後，アプリケーションを停止できるタイミングで実行されます。
4. Cleanup：
   3.の終了後，アプリケーションを停止できるタイミングで実行されます。
5. Concurrent Cleanup：
   4.の終了後，4.に続けて実行されます。

(b)　対象範囲

New 領域，Tenured 領域および Humongous 領域

(c)　処理内容

CM は大きく次の 5 つの処理から構成されます。それぞれの処理の詳細を示します。

1. Concurrent Root Region Scan：

ローカル変数や使用中のオブジェクトから直接参照されている Survivor 領域内のオブジェクトにマークを付ける処理です。マルチスレッドで実行されます。

2. Concurrent Mark：

   YoungGC(initial-mark)のマーキングと 1.のマーキングでマークを付けたオブジェクトが参照しているオブジェクトにマークを付ける処理です。マルチスレッドで実行されます。

3. Remark：

   2.のマーキング中に参照関係が変化したオブジェクトのマークを付け直す処理です。マルチスレッドで実行されます。

4. Cleanup：

   リージョンごとに使用中オブジェクトの合計サイズを求める処理です。また，次の CM に備え，マークの初期化もします。マルチスレッドで実行されます。

5. Concurrent Cleanup：

   使用中のオブジェクトが 1 つも存在しないリージョンを回収する処理です。シングルスレッドで実行されます。

(d) 処理結果

Eden 領域：

使用中のオブジェクトに対応した領域にマークが付きます。

5.の処理で Eden リージョンが回収された場合，領域サイズが減少します。

Survivor 領域：

使用中のオブジェクトに対応した領域にマークが付きます。

5.の処理で Survivor リージョンが回収された場合，領域サイズが減少します。

Tenured 領域：

使用中のオブジェクトに対応した領域にマークが付きます。

5.の処理で Tenured リージョンが回収された場合，領域サイズが減少します。

Humongous 領域：

使用中のオブジェクトに対応した領域にマークが付きます。

5.の処理で Humongous リージョンが回収された場合，領域サイズが減少します。

Metaspace 領域：

変化はありません。

Free 領域：

使用中のオブジェクトが 1 つも存在しないリージョンを回収した場合，領域サイズが増加します。

(e) アプリケーションの停止の有無

1. Concurrent Root Region Scan：

   CM スレッドで，アプリケーションを停止しないで実行されます。

2. Concurrent Mark：
   CM スレッドで，アプリケーションを停止しないで実行されます。
3. Remark：
   VM スレッドで，アプリケーションを停止して実行されます。
4. Cleanup：
   VM スレッドで，アプリケーションを停止して実行されます。
5. Concurrent Cleanup：
   CM スレッドで，アプリケーションを停止しないで実行されます。

アプリケーションは停止しませんが，4.の処理の終了から 5.の処理が終了するまでの間に FullGC や新しいリージョンの確保がされた場合，5.の処理が終了するまで，YoungGC(normal)や FullGC や新しいリージョンの確保の処理を待たせます。

(f) ほかの GC との関係

注　CM の処理中は YoungGC(initial-mark)は発生しません。

1. Concurrent Root Region Scan：
   - YoungGC(normal)：実行されません。
   - MixedGC：実行されません。
   - FullGC：実行されません。
2. Concurrent Mark：
   - YoungGC(normal)：2.の処理中に実行された場合，2.の処理を中断します。YoungGC(normal)が終了すると，再開されます。
   - MixedGC：実行されません。
   - FullGC：2.の処理中に実行された場合，2.の処理を中止します。途中結果は破棄されます。
3. Remark：
   - YoungGC(normal)：実行されません。
   - MixedGC：実行されません。
   - FullGC：実行されません。
4. Cleanup：
   - YoungGC(normal)：実行されません。
   - MixedGC：実行されません。
   - FullGC：実行されません。
5. Concurrent Cleanup：

- YoungGC(normal)：5.の処理中に実行された場合，5.の処理が終了するまで YoungGC(normal) を待機させます。
- MixedGC：実行されません。
- FullGC：5.の処理中に実行された場合，5.の処理が終了するまで FullGC を待機させます。

(g) 補足

- 関連オプション

  CM をするスレッド数は-XX:ConcGCThreads オプションで指定できます。CM をするスレッド数を増やすとスループットが低下します。-XX:ConcGCThreads オプションを指定しなかった場合，スレッド数は-XX:ConcGCThreads オプションのデフォルト値となります。詳細については，「5.4 日立 JavaVM で指定できる Java HotSpot VM のオプション」の「表 5-9 指定できる Java HotSpot VM のオプション」にある「-XX:ConcGCThreads」を参照してください。

- 確認方法

  Concurrent Root Region Scan，Concurrent Mark，Concurrent Cleanup の確認はログの先頭に［VCM］の識別子が付いていることから確認できます。Concurrent Mark のログのイメージを次に示します。

```
[VCM]<Wed Jul 31 11:45:23 2013>[Concurrent Mark Start][User: 0.0000000 secs][Sys: 0.0000000
secs]
[VCM]<Wed Jul 31 11:45:31 2013>[Concurrent Mark End][User: 0.0321549 secs][Sys: 0.0129454 se
cs]
```

VCM のログが複数行にわたるため，CM ログは開始時に Start，終了時に End が出力されます。複数行に渡る場合，CPU 使用時間は End のログに Start から End までの処理時間が出力され，Start のログは 0 が出力されます。

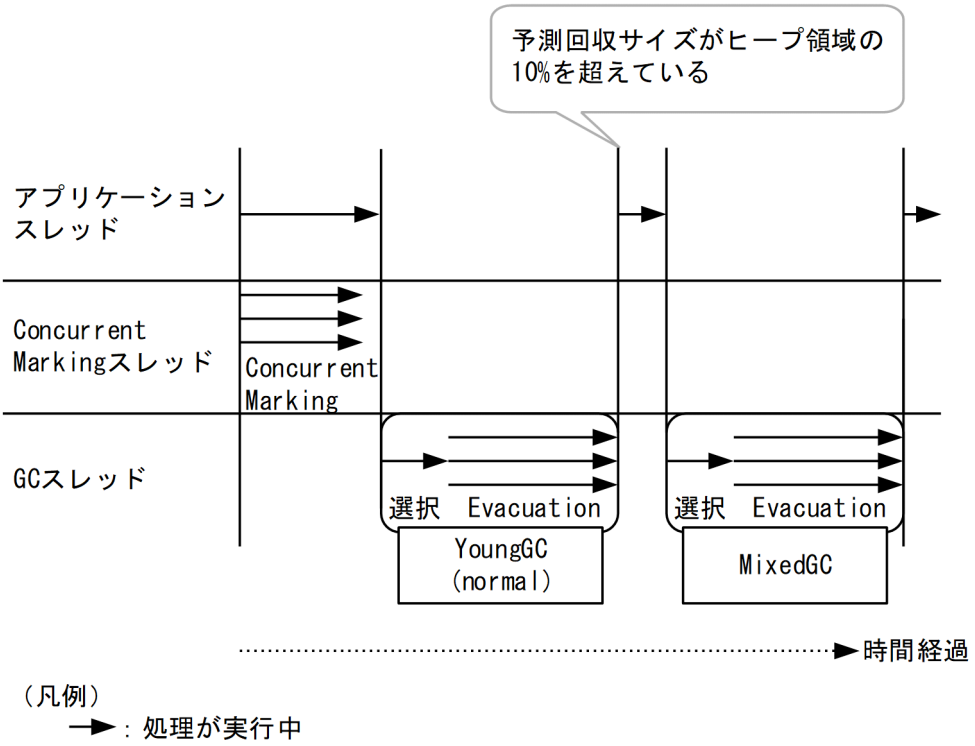
ログの記述内容や詳細に関しては，「-XX:[+|-]HitachiVerboseGC（拡張 verbosegc 情報出力オプション）」を参照してください。

また Remark，Cleanup の処理は処理中にアプリケーションを停止するため，VG1 ログに出力されます。Remark，Cleanup の処理は GC ログの GC 種別が“CM Remark”，“CM Cleanup”という出力から判断できます。

(10) MixedGC

MixedGC の処理の流れを次の図に示します。

図 2-26　MixedGC の処理の流れ



(a)　実行契機

予測回収サイズが Java ヒープ領域の 10%を超えている場合，次の GC に MixedGC が予約されます。その判定は CM 終了直後の YoungGC(normal)終了時，または MixedGC の終了時にされます。

(b)　対象範囲

New 領域と Tenured 領域の一部

(c)　処理内容

- MixedGC が発生すると，シングルスレッドでリージョンの選択処理を，マルチスレッドで Evacuation を実行します。
- MixedGC では New 領域を GC 対象とし，予測停止時間が目標停止時間に収まる範囲で Tenured 領域を部分的に GC 対象に追加します。
- MixedGC の Evacuation では，New 領域に対しては YoungGC の Evacuation と同じ処理をします。詳細については，「(8)　YoungGC」を参照してください。
- GC 対象に追加した Tenured 領域に対しては，Tenured リージョン内の使用中のオブジェクトを別の Tenured リージョンに詰め直します。
- MixedGC 後も予測回収サイズが Java ヒープ領域の 10%を超えている場合，継続して MixedGC が選択され，要件を満たしていない場合，通常の YoungGC が実行されます。

(d) 処理結果

Eden 領域：

オブジェクトが回収され，空になります。GC 後リサイズされます。

Survivor 領域：

From 空間のオブジェクトが回収され，空になります。GC 後リサイズされます。

Tenured 領域：

長期間必要と判断されたオブジェクトが Tenured 領域に移動します。GC 対象に追加された領域のオブジェクトが回収されます。

Humongous 領域：

変化はありません。

Metaspace 領域：

変化はありません。

Free 領域：

GC 後のリサイズによって，増減します。

(e) アプリケーションの停止の有無

停止します。

(f) ほかの GC との関係

CM：MixedGC 中に実行されません。

YoungGC：MixedGC 中に実行されません。

FullGC：MixedGC 中に実行要件を満たすと，MixedGC を中止して実行されます。

(g) 補足

- 関連オプション

  Evacuation をするスレッド数は-XX:ParallelGCThreads オプションで変更できます。スレッド数を増やすと MixedGC に掛かる時間が小さくなります。また，オプションを指定しない場合，スレッド数は OS が認識している CPU 数が用いられます。-XX:ParallelGCThreads オプションについては，「5.4　日立 JavaVM で指定できる Java HotSpot VM のオプション」の「表 5-9　指定できる Java HotSpot VM のオプション」にある「-XX:ParallelGCThreads」を参照してください。

- 確認方法

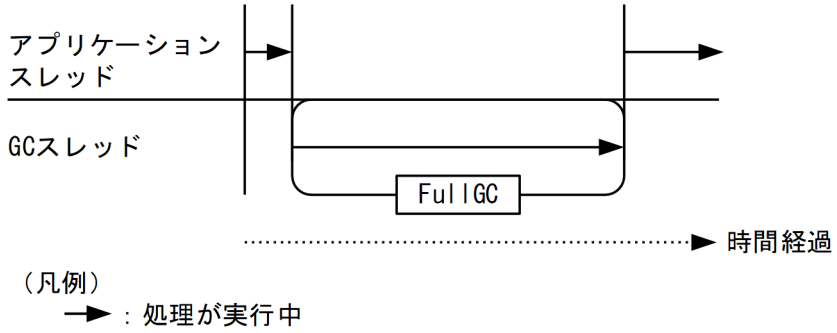
  MixedGC の確認はログの GC の種別の“Mixed GC”から確認できます。また，TargetTenured の項目から MixedGC で選択された Tenured 領域のサイズを確認できます。オブジェクトの予測回収サイズに関しては Reclaimable の項目から確認できます。ログの記述内容や詳細に関しては，「-XX:[+|-]HitachiVerboseGC（拡張 verbosegc 情報出力オプション）」を参照してください。

```
[VG1]<Wed Jun 12 11:21:10 2013>[Mixed GC 899070K/899072K(1048576K)->501742K/501760K(10485
76K), 0.0931560 secs][Status:-][G1GC::Eden: 389120K(389120K)->0K(397312K)][G1GC::Survivor
: 41984K->41984K][G1GC::Tenured: 459776K->459776K][G1GC::Humongous: 2048K->2048K][G1GC::F
ree: 609536K->607232K][Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space:
356K(388K, 388K)->356K(388K, 388K)][cause:G1EvacuationPause][RegionSize: 1024K][Target: 0
.2000000 secs][Predicted: 0.2495800 secs][TargetTenured: 2048K][Reclaimable: 17703K(1.69%
)][User: 0.0156250 secs][Sys: 0.0312500 secs][IM: 729K, 928K, 0K][TC: 509][DOE: 16K, 171]
[CCI: 2301K, 49152K, 2304K]
```

(11) FullGC

FullGC の流れを次の図に示します。

図 2-27 FullGC の流れ



(a) 実行契機

Free 領域がなくなり，Evacuation のコピー先が確保できなかった場合や Humongous 領域が確保できなかった場合に実行されます。また，Metaspace 領域の空き領域がなくなった場合や System.gc()が呼ばれた場合も実行されます。

(b) 対象範囲

New 領域，Tenured 領域，Humongous 領域および Metaspace 領域

(c) 処理内容

- FullGC はシングルスレッドでアプリケーションスレッドを停止して実行されます。
- FullGC が発生すると目標停止時間に関係なく，GC が終了するまでアプリケーションを停止します。
- FullGC の処理は SerialGC の FullGC と同じ処理です。FullGC を実行しても領域が確保できなかった場合は，OutOfMemory が発生します。詳細については，「2.2.1 SerialGC の仕組み」を参照してください。
- 一般的に，YoungGC や MixedGC の方が FullGC に比べて短い時間で処理できます。

(d) 処理結果

Eden 領域：オブジェクトが回収され，空になります。

Survivor 領域：使用中のオブジェクトが Tenured 領域に移動し，空になります。

Tenured 領域：使用中のオブジェクトが詰め直されます。

Humongous 領域：使用中のオブジェクトが詰め直されます。

Metaspace 領域：使用中のオブジェクトが詰め直されます。

Free 領域：回収されたリージョンによって増加します。

(e)　アプリケーションの停止の有無

停止します。

(f)　ほかの GC との関係

CM：FullGC 中は実行されません。

YoungGC：FullGC 中は実行されません。

MixedGC：FullGC 中は実行されません。

(g)　補足

- 確認方法

  FullGC の確認はログ上の GC の種別が“FullGC”であることから確認できます。また，FullGC ではこれから実行する GC に対して予測をしないため，予測停止時間や予測回収サイズは 0 で表示されます。

```
[VG1]<Wed Jan 15 12:51:32 2014>[Full GC 130443K/131072K(131072K)->55462K/56320K(131072K),
 2.3265610 secs][Status:-][G1GC::Eden: 0K(43008K)->0K(43008K)][G1GC::Survivor: 0K->0K][G1
GC::Tenured: 131072K->56320K][G1GC::Humongous: 0K->0K][G1GC::Free: 0K->74752K][Metaspace:
 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K
)][cause:ObjAllocFail][RegionSize: 1024K][Target: 0.2000000 secs][Predicted: 0.0000000 se
cs][TargetTenured: 0K][Reclaimable: 0K(0.00%)][User: 2.1700000 secs][Sys: 0.0000000 secs]
[IM: 277185K, 261856K, 44544K][TC: 1168][DOE: 0K, 0][CCI: 5808K, 49152K, 5952K]
```

2.3.2　G1GC のチューニング

G1GC のチューニング方法について説明します。システム構築時にチューニングを実施し，システム要件を満たしていることを確認してから本番環境に適用してください。

Java ヒープ領域のサイズに関係するオプションの一覧を次の表に示します。なお，表の項番 1～8 は「図 2-28」の番号と対応しています。

表 2-4　Java ヒープ領域内の各領域と Metaspace 領域のサイズを指定するオプション

項番	オプション名	オプションの意味
1	-Xms<size>	Java ヒープの初期サイズを設定します。※1

項番	オプション名	オプションの意味
2	-Xmx<size>	Java ヒープの最大サイズを設定します。※1
3	-XX:NewSize=<value>	New 領域の最小サイズを設定します。※2
4	-XX:MaxNewSize=<value>	New 領域の最大サイズを設定します。※2
5	-XX:NewRatio=<value>	Java ヒープ領域に対する New 領域の割合を設定します。※2 <value>が 2 の場合は，New 領域と Tenured 領域の割合が，1:2 になります。
6	-XX:SurvivorRatio=<value>	New 領域に対する Survivor 領域で最大に取ることができる領域サイズの割合を設定します。 Survivor 領域の最大で取ることができる領域サイズは次の式で求められます。 (Survivor 領域のサイズ)※3=(New 領域のサイズ)/SurvivorRatio
7	-XX:MetaspaceSize=<size>	Metaspace 領域の初期サイズを設定します。※4
8	-XX:MaxMetaspaceSize=<size>	Metaspace 領域の最大サイズを設定します。※4

注※1

-Xms と-Xmx の値は同じ値を指定することを推奨します。

注※2

項番 3～5 のオプションを指定した場合，New 領域のリサイズが制限され，目標停止時間内に GC 停止時間を抑えようとする G1GC のメリットを損なうため，G1GC では指定しないことを推奨します。また，項番 3～5 のオプションを指定しなかった場合，New 領域の初期サイズは Java ヒープの初期サイズ×0.2 のサイズが確保され，Eden 領域の初期サイズは SurvivorRatio のデフォルト値である 8 に従って割り当てられます。

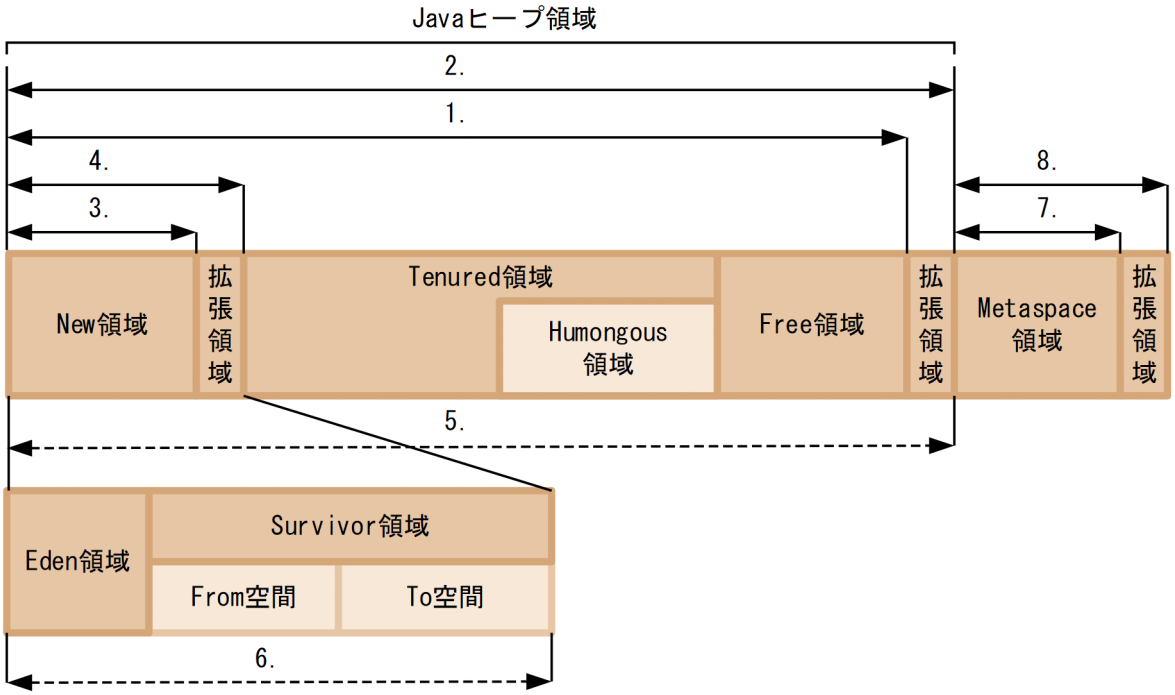
注※3

Survivor 領域のサイズとは To 空間または From 空間のサイズを指します。To 空間と From 空間の合計サイズではないことに注意してください。

注※4

-XX:MetaspaceSize と-XX:MaxMetaspaceSize の値は同じ値を指定することを推奨します。

図 2-28　Java ヒープ領域内の各領域と Metaspace 領域のサイズとオプションの対象範囲



（凡例）

X. ：表中の項番と対応

◀——▶ ：サイズを指定するオプションの対象範囲

◀- - -▶ ：割合を指定するオプションの対象範囲

G1GC のチューニングに利用するオプションを次の表に示します。各オプションの詳細については，「5.4 日立 JavaVM で指定できる Java HotSpot VM のオプション」の各オプションの説明を参照してください。

表 2-5　G1GC のチューニングオプション

項番	オプション名	デフォルト値	オプションの意味
1	-XX:MaxGCPauseMillis=<value>	200	目標停止時間[ms]
2	-XX:ParallelGCThreads=<value>	CPU 数※ (OS が認識している CPU 数)	Evacuation の処理をするスレッド数
3	-XX:ConcGCThreads=<value>	(ParallelGCThreads + 2) / 4	CM 処理をするスレッド数

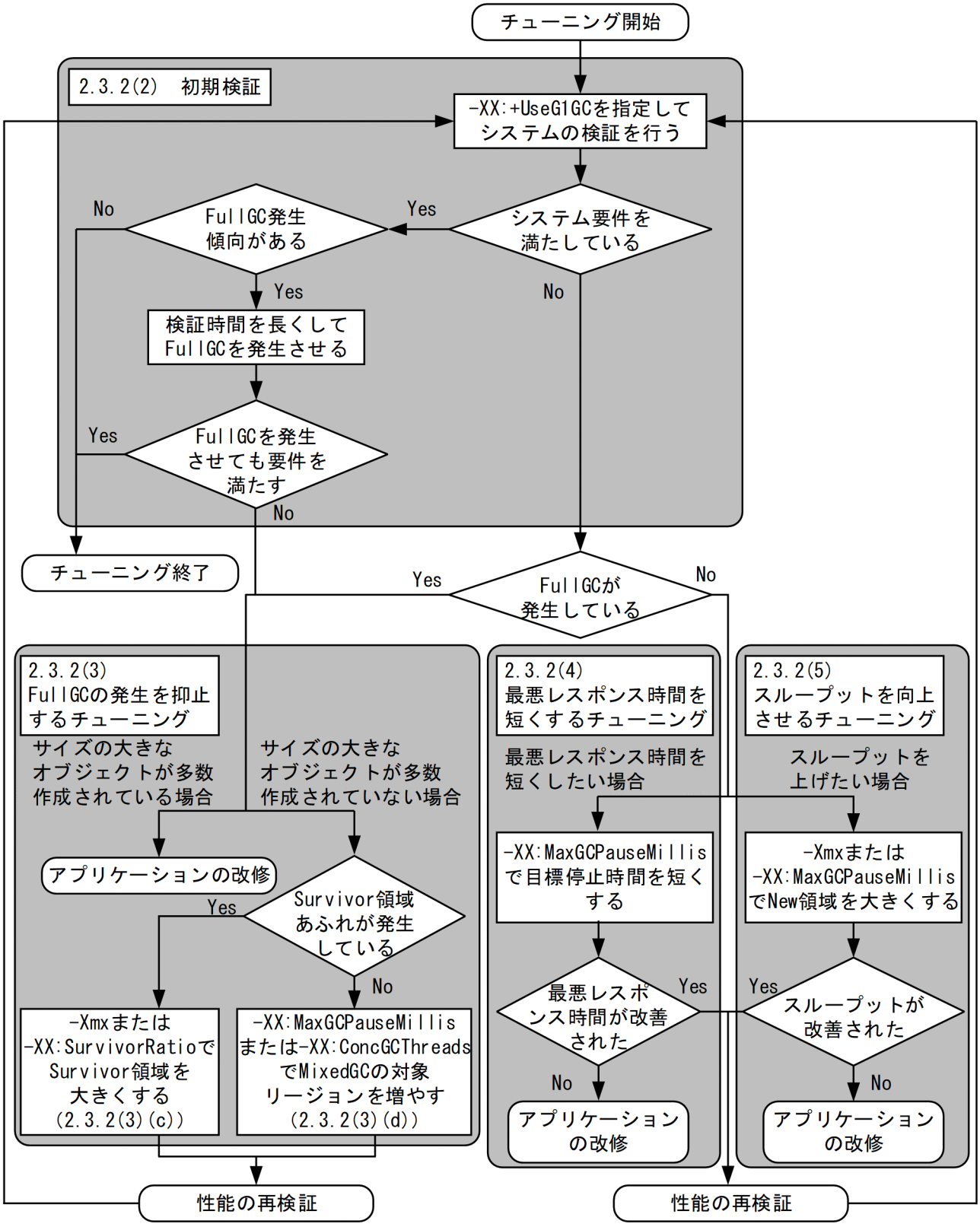
注※

実行環境の CPU 数によって計算式が異なります。

(1)　チューニングの流れ

チューニングの全体の流れを次の図に示します。

図 2-29　チューニングの全体の流れ

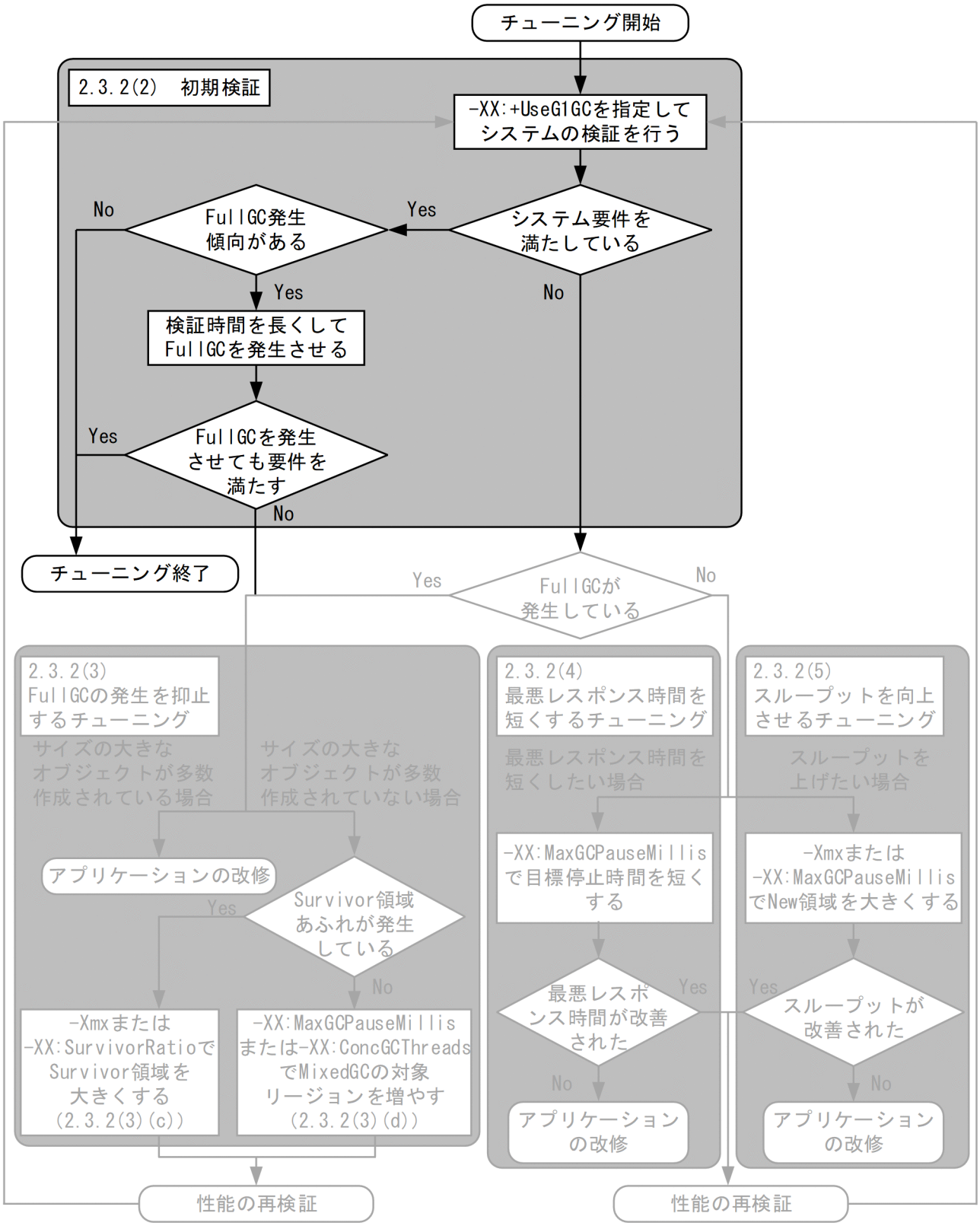


次から，それぞれのチューニングの詳細を説明します。

(2)　初期検証

初期検証の流れを次の図に示します。

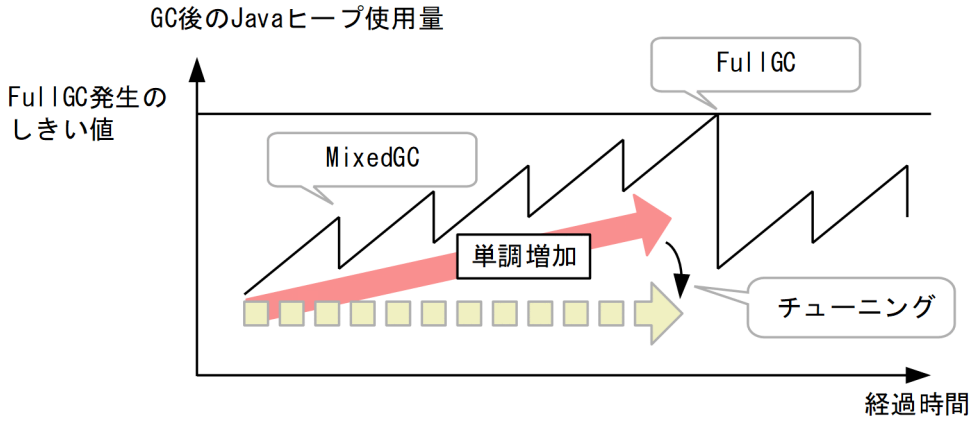
図 2-30　初期検証の流れ



1. G1GC を有効にしてシステムの検証をします。検証の結果，システム要件を満たしている場合でも，FullGC の発生傾向がないかどうかを確認します。システム要件を満たしていない場合は，「(3)　FullGC の発生を抑止するチューニング」へ進んでください。
2. 検証中に FullGC が発生していない場合でも，システムを長期間動かすことで，将来 FullGC が発生する場合があります。そのため，FullGC の発生傾向がないかどうかを確認します。

   FullGC の発生傾向のイメージを次の図に示します。

   図 2-31　FullGC の発生傾向のイメージ



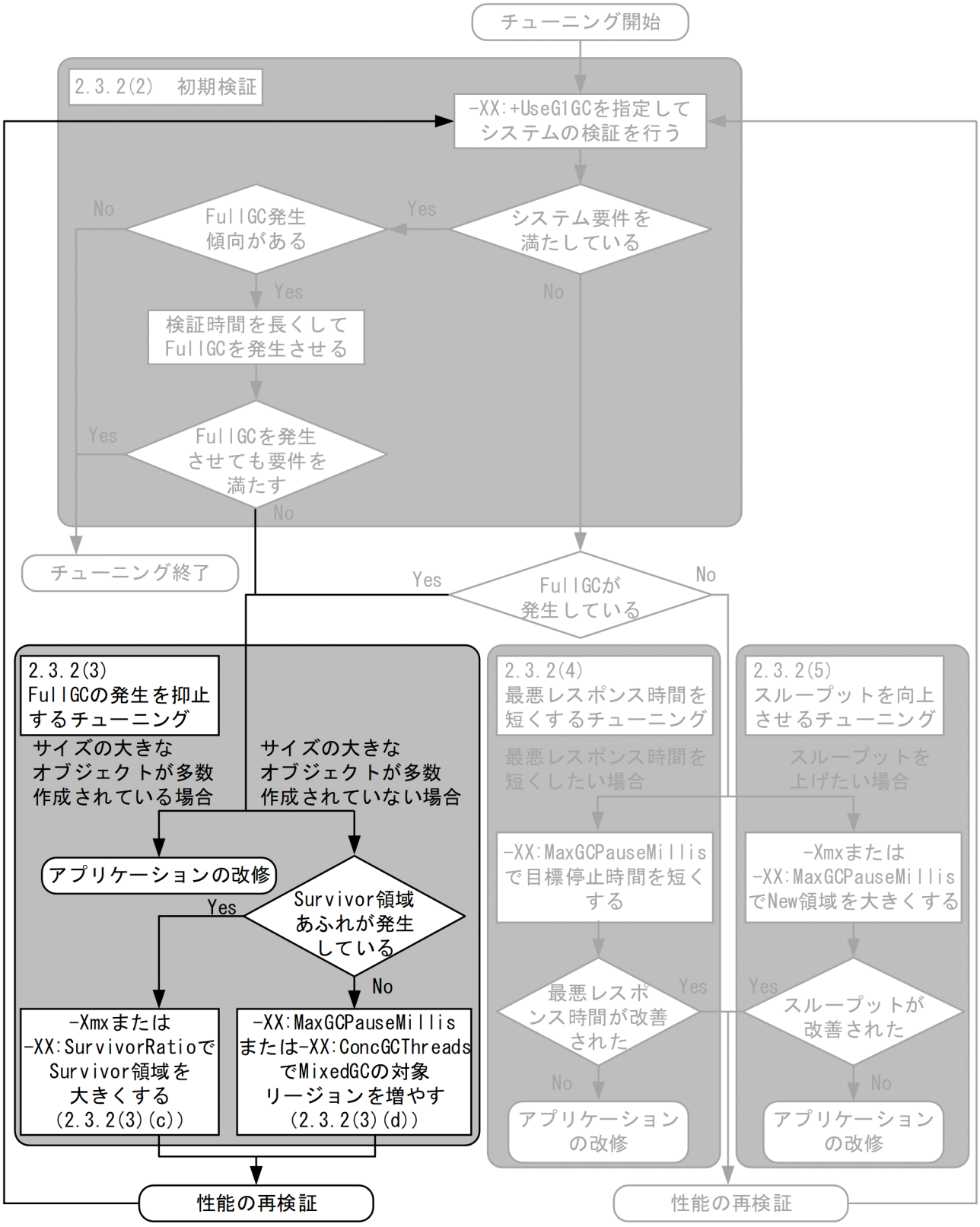
   この図は検証中の GC 後の使用量をグラフにしたイメージです。この図のように MixedGC が発生していても，Java ヒープ使用量が長期的には単調増加している場合，FullGC の発生傾向があります。
3. システム要件を満たしている場合でも，FullGC の発生傾向があった場合，検証時間を延ばし FullGC を発生させ，FullGC が発生してもシステム要件が満たしているかどうかを確認します。FullGC を発生させた結果，システム要件を満たさなくなった場合は，FullGC の発生を抑止するチューニングをします。また，検証時間を延ばすことができない場合は，FullGC が発生していなくても，FullGC の発生を抑止するチューニングをし，FullGC の発生傾向をなくしてください。FullGC の発生を抑止するチューニングについては，「(3)　FullGC の発生を抑止するチューニング」を参照してください。

(3)　FullGC の発生を抑止するチューニング

(a)　FullGC の発生を抑止する考え方

FullGC の発生を抑止するチューニングの流れを次の図に示します。

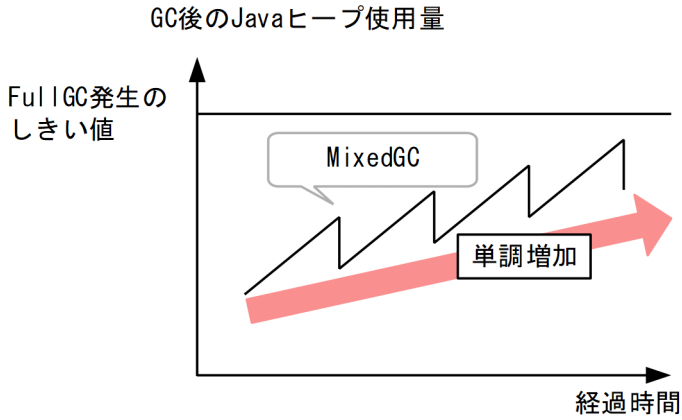
図 2-32　FullGC の発生を抑止するチューニングの流れ



FullGC は Tenured 領域の使用サイズが増加し続け，Free リージョンが確保できなくなった場合実行されます。

FullGC の発生を抑止するチューニングのイメージを次の図に示します。

図 2-33　FullGC の発生を抑止するチューニングのイメージ



FullGC の発生を抑止するには，Tenured 領域の使用サイズの増加を防ぐ必要があります。そのため，FullGC の発生を抑止するには次の 2 つの方法が考えられます。

1. Tenured 領域へ移動するオブジェクト数を減らす
2. Tenured 領域内で不要なオブジェクトを回収する

Survivor 領域あふれが発生している場合は 1 の方法で，発生していない場合は 2 の方法でチューニングしてください。1 の方法の詳細については，「(c)　Survivor 領域のサイズチューニング」を，2 の方法の詳細については，「(d)　MixedGC で選択されるリージョンを増やすチューニング」を参照してください。チューニングをした場合，再度システムの検証をし，要件を満たしているか確認してください。

(b)　サイズの大きなオブジェクトが多数作成されている場合

1 つのオブジェクトのサイズが大きいオブジェクトが多数作成される場合，連続した領域が確保できないで，FullGC が実行される場合があります。G1GC ではメモリをリージョンで管理しているため，オブジェクトのサイズが大きいオブジェクトが多数作成されるシステムには不向きです。オブジェクトのサイズが大きいオブジェクトが多数作成されることによって，FullGC が発生している場合は，アプリケーションの改修をしてください。

サイズの大きなオブジェクトはログの「Humongous」の項目から確認できます。

```
[VG1]<Wed Jan 15 12:51:32 2014>[Full GC 130443K/131072K(131072K)->55462K/56320K(131072K), 2.
3265610 secs][Status:-][G1GC::Eden: 0K(43008K)->0K(43008K)][G1GC::Survivor: 0K->0K][G1GC::Te
nured: 131072K->56320K][G1GC::Humongous: 1024K->0K][G1GC::Free: 0K->74752K][Metaspace: 3634
K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][cause
:ObjAllocFail][RegionSize: 1024K][Target: 0.2000000 secs][Predicted: 0.0000000 secs][TargetT
enured: 0K][Reclaimable: 0K(0.00%)][User: 2.1700000 secs][Sys: 0.0000000 secs][IM: 277185K,
261856K, 44544K][TC: 1168][DOE: 0K, 0][CCI: 5808K, 49152K, 5952K]
```

(c)　Survivor 領域のサイズチューニング

Tenured 領域へ移動するオブジェクト数を減らす方法は，Survivor 領域にオブジェクトが存在する間に，オブジェクトを回収する方法です。Survivor 領域の空き領域がなくなると，本来寿命の短いオブジェクト

が Survivor 領域あふれによる退避で Tenured 領域に昇格します。そのため，Survivor 領域を大きくすることで，寿命の短いオブジェクトを Survivor 領域で回収し，Tenured 領域への移動量を減らします。

ログに次のように「to exhausted」が出力されている場合，Survivor 領域あふれが発生しています。この場合 Survivor 領域を大きくするために，-Xmx オプションまたは-XX:SurvivorRatio オプションでチューニングをしてください。

```
[VG1]<Wed Jun 12 11:21:10 2013>[Young GC 899070K/899072K(1048576K)->501755K/501760K(1048576K
), 0.0931560 secs][Status:to exhausted][G1GC::Eden: 389120K(389120K)->0K(397312K)][G1GC::Sur
vivor: 41984K->41984K][G1GC::Tenured: 459776K->459776K][G1GC::Humongous: 2048K->2048K][G1GC:
:Free: 609536K->607232K][Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 3
56K(388K, 388K)->356K(388K, 388K)][cause:G1EvacuationPause][RegionSize: 1024K][Target: 0.200
0000 secs][Predicted: 0.2495800 secs][TargetTenured: 0K][Reclaimable: 0K(0.00%)][User: 0.015
6250 secs][Sys: 0.0312500 secs][IM: 729K, 928K, 0K][TC: 509][DOE: 16K, 171][CCI: 2301K, 4915
2K, 2304K]
```

チューニングに用いるオプション名とチューニング方法を次に示します。

- -Xmx オプション

  指定する値を大きくすることで，Java ヒープ領域のサイズが大きくなります。Java ヒープ領域全体のサイズが大きくなることで，Survivor 領域のサイズも大きくなります。ただし，Java ヒープ領域の最大サイズを大きくすると，New 領域がリサイズされる範囲の最小値と最大値が大きくなります。これによって最小停止時間が大きくなるため，チューニングをしたあとに再度システム要件を満たしているか検証してください。

- -XX:SurvivorRatio オプション

  指定する値を小さくすることで，予約した New 領域のうち，Survivor 領域に割り当てる割合を大きくします。ただし，この方法は Eden 領域がチューニング前に比べて小さくなるため，YoungGC や MixedGC が発生しやすくなる副作用があります。そのため，チューニングをしたあとに再度システム要件を満たしているか検証してください。

  重要

  Survivor 領域を大きくする方法として-XX:NewRatio や-XX:NewSize，-XX:MaxNewSize オプションを使って New 領域のサイズを大きくする方法もあります。しかし，この場合 GC 後の New 領域のリサイズが制限されるため，New 領域のサイズを変更するオプションの指定は推奨しません。

(d) MixedGC で選択されるリージョンを増やすチューニング

MixedGC で対象とするリージョンを増やすには，1 度の MixedGC の対象リージョンを増やす方法と，MixedGC の発生頻度を上げて対象リージョン延べ数を増やす方法があります。1 度の MixedGC の対象リージョン数を増やす方法はレスポンスが低下し，対象リージョンの延べ数を増やす方法はスループットが低下します。システム要件にあわせてチューニング方法を選択してください。

- -XX:MaxGCPauseMillis オプション

指定する値を大きくすることで，目標停止時間が大きくなります。MixedGC では New 領域と Tenured 領域の一部を対象として GC をします。Tenured 領域は New 領域を選択しても予測停止時間に対して目標停止時間に余裕がある場合に選択されます。そこで，目標停止時間を大きくすることで，1 度の MixedGC で選択される Tenured リージョンの数が増えます。ただし，この方法は GC 停止時間が長くなるため，レスポンスが低下します。

- -XX:ConcGCThreads オプション

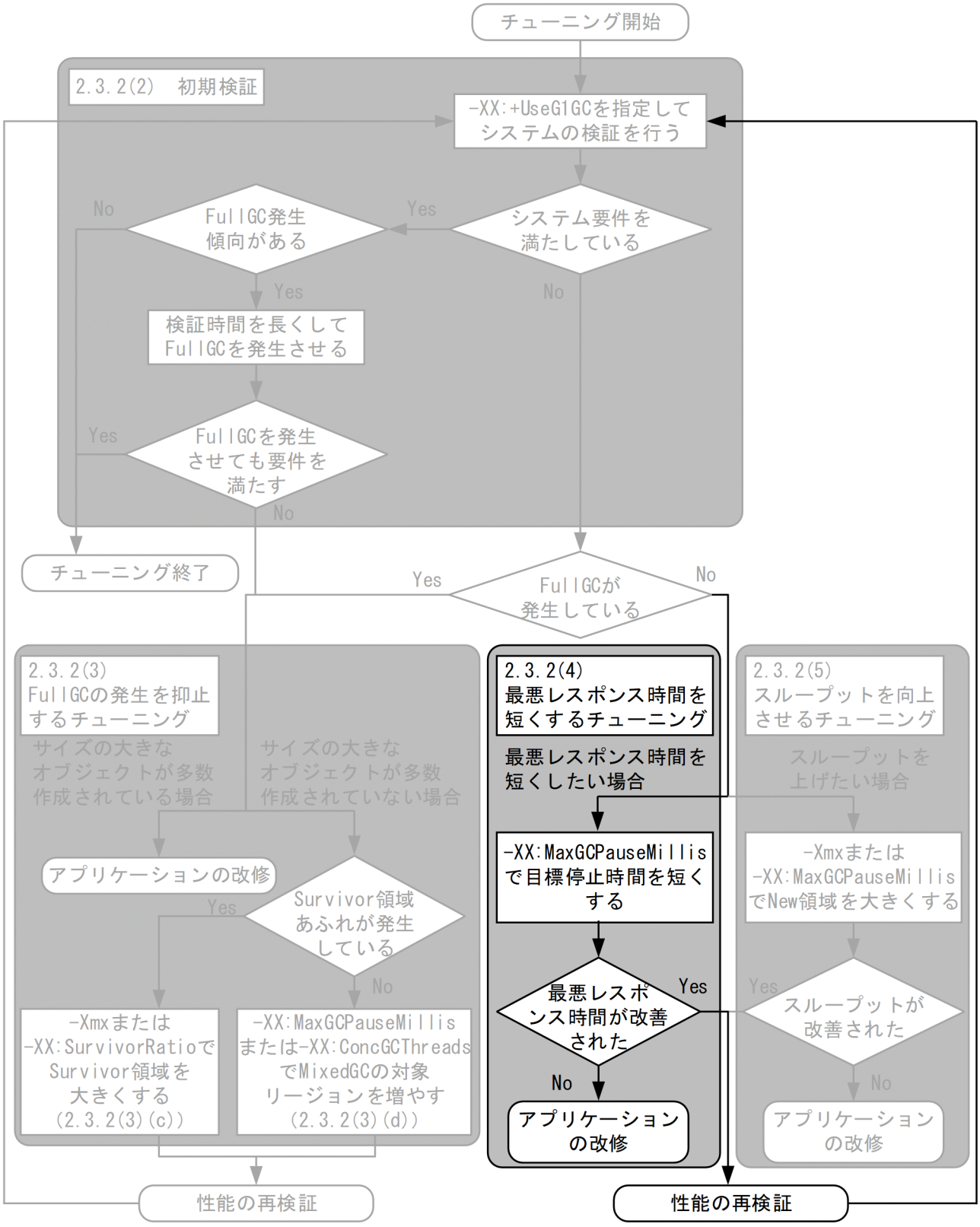
  指定する値を大きくすることで，CM をするスレッド数が増えます。MixedGC が実行されるためには，CM が終了している必要があります。そのため，CM をするスレッド数を増やし，CM の終了間隔を短くします。これによって MixedGC の発生回数が増え，MixedGC で選択されるリージョンの延べ数が増えます。ただし，この方法では，アプリケーションスレッドと並行して処理をするスレッド数が増えるため，スループットが低下します。

  また，CM スレッド数は Evacuation をするスレッド数より多い値を指定できません。そのため，-XX:ConcGCThreads オプションとあわせて，-XX:ParallelGCThreads オプションの値も大きくする必要があります。

(4) 最悪レスポンス時間を短くするチューニング

最悪レスポンス時間を短くするチューニングの流れを次の図に示します。

図 2-34　最悪レスポンス時間を短くするチューニングの流れ



このチューニングは，GC によるアプリケーションの停止時間を短くすることで，最悪レスポンス時間を短くする方法です。そのため，レスポンス時間のうち，GC によるアプリケーションの停止時間が占める割合が大きい場合に有用です。GC によるアプリケーションの停止時間は，VG1 ログの gc_time の項目から取得できます。

VG1 ログの詳細については，「-XX:[+|-]HitachiVerboseGC（拡張 verbosegc 情報出力オプション）」を参照してください。

最悪レスポンス時間を短くするには，GC によるアプリケーションの停止時間を短くする必要があります。G1GC では停止時間を指定することができるため，指定する値を小さくすることで最悪レスポンス時間を短くします。

- -XX:MaxGCPauseMillis オプション

  指定する値を小さくすることで，目標停止時間が短くなり，最悪レスポンス時間が短くなります。ただし，この方法では GC 回数が多くなりスループットが低下します。そのため，最悪レスポンス時間を短くするチューニングをした場合，再度システム要件を満たしているか検証してください。また，G1GC ではどの GC でも New 領域を GC の対象とするため，New 領域の回収に掛かる時間以下には GC 時間を抑えられません。それ以上 GC 停止時間を短くできない GC 停止時間を最小停止時間と呼びます。目標停止時間を短くしても最悪レスポンス時間が短くならない場合は，ログの GC 停止時間を確認してください。

  GC 停止時間は次のログの**太字**の部分から確認できます。

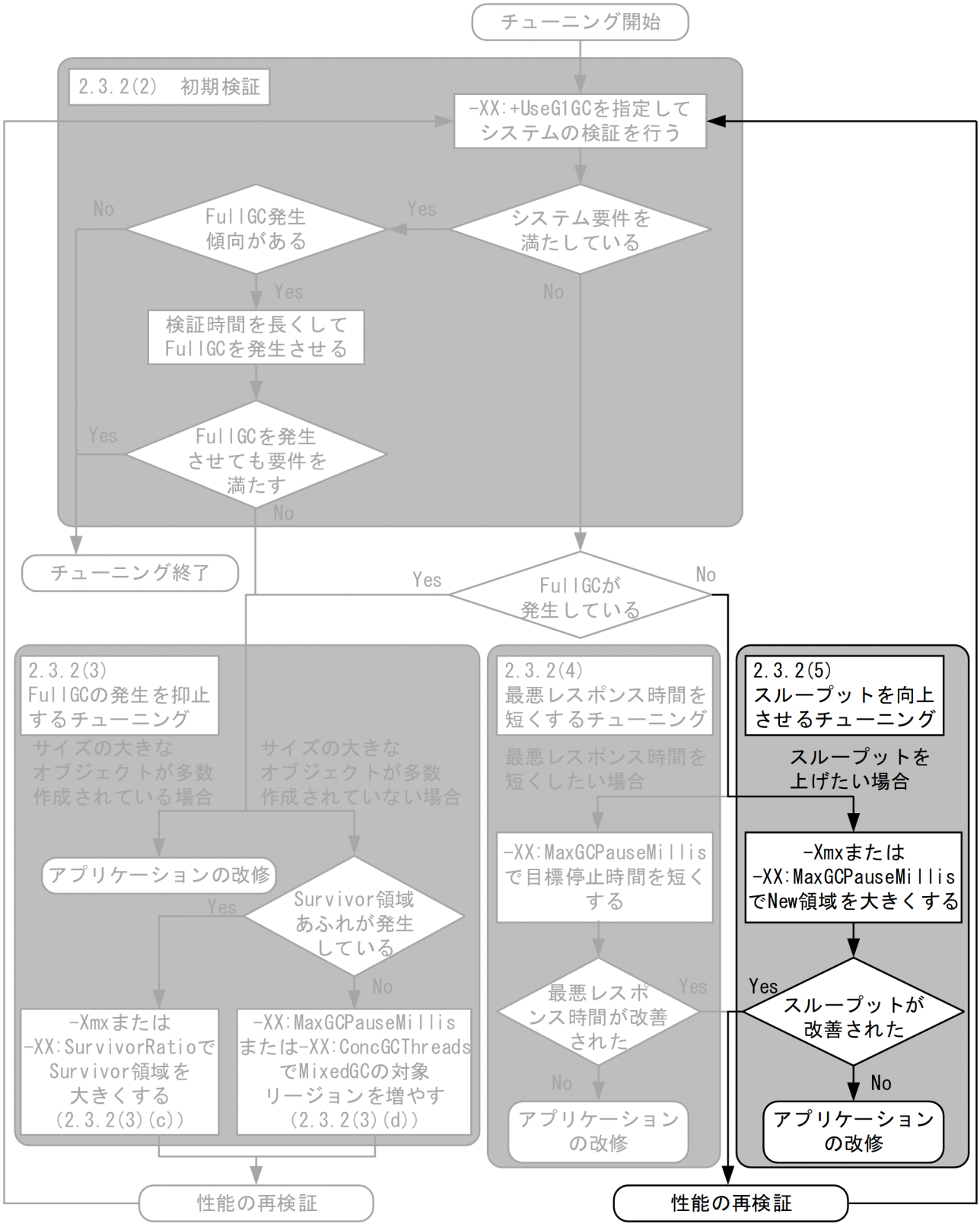
```
[VG1]<Wed Jun 12 11:21:10 2013>[Young GC 899070K/899072K(1048576K)->501755K/501760K(1048576K
), 0.0931560 secs][Status:-][G1GC::Eden: 389120K(389120K)->0K(397312K)][G1GC::Survivor: 4198
4K->41984K][G1GC::Tenured: 459776K->459776K][G1GC::Humongous: 2048K->2048K][G1GC::Free: 6095
36K->607232K][Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 3
88K)->356K(388K, 388K)][cause:G1EvacuationPause][RegionSize: 1024K][Target: 0.2000000 secs][
Predicted: 0.2495800 secs][TargetTenured: 0K][Reclaimable: 0K(0.00%)][User: 0.0156250 secs][
Sys: 0.0312500 secs][IM: 729K, 928K, 0K][TC: 509][DOE: 16K, 171][CCI: 2301K, 49152K, 2304K]
```

目標時間を短くしても GC の停止時間が短くなっていない場合は，最小停止時間に達しています。また，目標時間を最小停止時間に近づけ過ぎると，目標停止時間を超えたときの超え幅が大きくなり，最悪レスポンス時間が長くなる場合があります。目標停止時間を短くしてもシステム要件を満たせない場合は，チューニングではシステム要件を満たせません。その場合は，アプリケーションを改修してください。

(5) スループットを向上させるチューニング

スループットを向上させるチューニングの流れを次の図に示します。

図 2-35　スループットを向上させるチューニングの流れ



スループットを向上させるためには，GC の発生回数を減らす必要があります。YoungGC や MixedGC は割り当てられた Eden 領域を使い切ると発生します。そこで，New 領域を大きくすることで Eden 領域を大きくし，GC の発生回数を減らします。New 領域を大きくするためには，-Xmx オプションまたは-XX:GCMaxPauseMillis オプションを用いてチューニングをします。

- -Xmx オプション

  指定する値を大きくすることで，Java ヒープ領域の最大サイズを大きくします。Java ヒープ領域の最大サイズが大きくなることで，New 領域も大きくなります。

- -XX:MaxGCPauseMillis オプション

  指定する値を大きくすることで，目標停止時間が長くなります。YoungGC では目標停止時間内に収まる範囲で，できるだけ大きく New 領域を取るため，目標停止時間を長くすることで，New 領域も大きくなります。

スループットを向上させるチューニングをしてもスループットが向上しない場合は，New 領域のサイズがリサイズされる範囲の最大サイズに達していることが考えられます。ログの Eden 領域＋ Survivor 領域の値の変化からリサイズがされているか確認してください。ログの確認方法の詳細については，「2.3.1(8) YoungGC」を参照してください。なお，スループットを向上させるチューニングをするとレスポンスが低下する場合があるため，チューニング後は性能の再検証をする必要があります。

3 トラブルシューティング

この章では，発生する代表的なトラブルについて，日立 JavaVM が生成した資料の確認が必要となるケースと，原因究明の手順について説明します。さらにアプリケーションサーバシステムで発生する代表的なトラブルについても説明します。

3.1　JavaVM でのトラブルシューティング

ここでは，JavaVM のトラブルの原因の切り分けに使用する資料と，代表的なトラブル事象の原因の切り分けの方法について説明します。

JavaVM を実行しているプロセスで想定されるトラブルには，次に示す種別があります。

- プロセスダウン
  JavaVM を実行しているプロセスが異常終了します。
- プロセスハングアップ（無応答）
  JavaVM を実行しているプロセスの応答が返らなくなります。
- スローダウン
  JavaVM を実行しているプロセスの応答は返りますが，応答時間が長くなります。
- OutOfMemory（メモリ不足）
  メモリ不足によって，プロセスが終了します。

トラブル種別ごとに取得する資料を，次の表に示します。

表 3-1　トラブル種別ごとの取得資料一覧

項番	資料の種類	トラブル種別			
		プロセスダウン	プロセスハングアップ（無応答）	スローダウン	OutOfMemory（メモリ不足）
1	エラーリポートファイル	○	－	－	○※
2	core ダンプ／スタックトレースファイル	○	○※	○	○※
3	JavaVM のスレッドダンプファイル	－	○	○	○
4	JavaVM ログファイル	○	○	○	○
5	標準出力と標準エラー出力	○	○	○	○

（凡例）
○：取得します。
－：取得しません。

注※
発生個所によっては資料が取得できない場合があります。

資料の説明と取得方法を次の表に示します。

表 3-2　資料の説明と取得方法

項番	資料の種類	説明	取得方法の参照先
1	エラーリポートファイル	JavaVM がダウンした場合に出力されます。	3.1.1(1)
2	core ダンプ／スタックトレースファイル	プロセスのメモリイメージがダンプ出力されます。 また，ダンプからスレッドのスタック状態が出力されます。	3.1.1(2)
3	JavaVM のスレッドダンプファイル	JavaVM の稼働情報やスレッドのスタック状態が出力されます。	3.1.1(3)
4	JavaVM ログファイル	JavaVM の GC 活動情報を含むトラブルシュート情報が JavaVM ログファイルに出力されます。	3.1.1(4)
5	標準出力と標準エラー出力	JavaVM のメッセージやエラーメッセージが出力されます。	※

注※

お客さまが使用するシステムの取得方法に従ってください。

取得した資料を使用して，トラブルの原因を判別します。資料の詳細内容については，それぞれ次の表に示す個所を参照してください。

表 3-3　調査方法の参照先

項番	資料の種類	説明
1	エラーリポートファイル	「7.4　日立 JavaVM が出力するメッセージログ（標準出力およびエラーリポートファイル）」を参照してください。
2	core ダンプ／ スタックトレースファイル	※
3	JavaVM のスレッドダンプファイル	「7.1　日立 JavaVM のスレッドダンプ」を参照してください。
4	JavaVM ログファイル	「7.3　JavaVM ログ（JavaVM ログファイル）」を参照してください。
5	標準出力と標準エラー出力	※

注※

保守員が使用する資料ですが，トラブルの原因を判別できることがあります。「3.1.2　トラブル事象別の原因切り分け方法」参照してください。

3.1.1　資料の取得方法

トラブルの原因の切り分けに使用する資料の取得方法について説明します。

(1)　エラーリポートファイル

JavaVM が出力するエラーリポートファイルの取得について説明します。

JavaVM でクラッシュが発生した場合，JavaVM はデバッグ情報を標準出力とエラーリポートファイルに出力します。エラーリポートファイルが出力されるタイミングを次に示します。

- JNI の中でシグナルが発生したとき
- JavaVM で C ヒープ不足が発生したとき
- JavaVM で予期しないシグナルが発生したとき
- JavaVM で Internal Error（内部論理エラー）が発生したとき

ただし，エラーリポートファイルの作成時にシグナルやメモリ不足が発生した場合，このファイルは作成されません。

エラーリポートファイルの出力先および出力ファイル名は，次のとおりです。

```
<カレントディレクトリ>/hs_err_pid<JavaプロセスのプロセスID>.log
```

メモ

C ヒープが不足した場合，次の順序でメッセージ出力および core ダンプの生成が実行されます。必要な情報を取得してください。

1. C ヒープ不足を示すメッセージログが，エラーリポートファイルおよび標準出力に出力されます。
2. 1.の実行中にメモリ不足が発生した場合，メッセージが標準出力に出力されます。
3. 簡易メッセージの出力処理中にさらにメモリ不足が発生した場合，メッセージおよびエラーログファイルの出力処理を中止して，core ダンプを生成します。

重要

クラッシュが発生した場合や C ヒープ不足時に標準出力に出力される JavaVM の情報は，-XX:+StandardLogToHitachiJavaLog オプションを指定しても日立 JavaVM ログファイルに出力されません。コンソールにだけ出力されます。

-XX:+StandardLogToHitachiJavaLog オプションについては，「-XX:[+|-]StandardLogToHitachiJavaLog（標準出力のログファイル出力オプション）」を参照してください。

(2) core ダンプ／スタックトレースファイル

core ダンプの取得について，説明します。

プロセスがダウンした場合，OS によって出力された core ダンプを取得します。

core ダンプを取得したあとに，core ダンプからスタックトレースファイルを取得する場合は，javatrace コマンドを実行してください。スタックトレース情報は，JavaVM の異常終了の原因を究明するために必要な情報です。

コマンド詳細については，「javatrace（トレース情報の収集）」を参照してください。

javatrace コマンドの実行形式を次に示します。

```
javatrace 〈coreダンプのファイル名称〉〈coreダンプを生成した実行ファイルの名称〉
```

JavaVM が異常終了して，「core.<プロセス ID>」というファイル名称で core ダンプが作成されているときにこのコマンドを実行すると，実行結果として，カレントディレクトリに「javatrace.log」というファイルが出力されます。スタックトレースファイルの出力先は，<core ダンプを生成した実行ファイルの名称>にパス名を指定することで，出力先を変更できます。

JavaVM が異常終了して core ダンプが作成されたときに出力されるメッセージの例を，次に示します。実行する javatrace コマンド文字列がメッセージ内に表示されます。文字列内の core は，core.<プロセス ID>とします。

```
:
# You can get further information from javatrace.log file generated
# by using javatrace command.
# usage: javatrace core-file-name loadmodule-name [out-file-name] [-l(library-name)...]
# Please use javatrace command as follows and submit a bug report
# to Hitachi with javatrace.log file:
# [/opt/Cosminexus/jdk/bin/javatrace core /opt/Cosminexus/jdk/bin/java]
```

また，実行中の Java プロセスで core ダンプとスレッドダンプを同時に取得できます。実行中の Java プロセスで core ダンプとスレッドダンプを同時に取得する場合は，javacore コマンドを実行してください。コマンド詳細については，「javacore（core ファイルとスレッドダンプの取得）」を参照してください。

```
javacore -p 〈Javaプロセスのプロセス ID〉
```

プロセス ID の取得方法は，「(3)　JavaVM のスレッドダンプファイル」の「参考」を参照してください。

上記の形式でコマンドを実行すると，次のメッセージが出力されます。

```
send SIGQUIT to 8662: ? (y/n)
```

「y」を入力すると，実行中の Java プログラムのカレントディレクトリに，"javacore<プロセス ID>.<出力日時>.core"（core ダンプ），および"javacore<プロセス ID>.<出力日時>.txt"（スレッドダンプ）が出力されます。「n」を入力すると，core ダンプおよびスレッドダンプを取得しないでコマンドの実行を終了します。

core ダンプおよびスレッドダンプを取得すると，実行中の Java プログラムには，次のメッセージが出力されます。

```
Now generating core file (javacore8662.030806215140.core)...
done
:
Writing Java core to javacore8662.030806215140.txt... OK
```

重要

次のシェルコマンドを実行して，core ダンプファイルサイズの上限値を無制限にしてください。

csh（C シェル）の場合

```
limit coredumpsize  unlimit
```

sh（標準シェル）および ksh の場合

```
ulimit -c unlimited
```

また，これらの設定に加え，シェルコマンドを実行して，ファイルサイズの上限値を無制限にしておくことをお勧めします。

csh（C シェル）の場合

```
limit filesize unlimit
```

sh（標準シェル）および ksh の場合

```
ulimit -f unlimited
```

(3) JavaVM のスレッドダンプファイル

スレッドダンプファイルを取得する方法について説明します。

スレッドダンプファイルを取得するには，kill コマンドを実行します。

```
kill -3 <JavaプロセスのプロセスID>
```

時間の経過に応じた各スレッドの状態遷移を確認するときには，目安として 3 秒おきに 10 回程度実行します。

スレッドダンプファイルの出力先および出力ファイル名は，次のとおりです。

```
<カレントディレクトリ>/javacore<プロセスID>.<出力日時>※.txt
```

注※

出力日時の形式は YYMMDDhhmmss です。

YY：年（西暦で下2桁），MM：月（2桁），DD：日（2桁）

hh：時間（24時間表記），mm：分（2桁），ss：秒（2桁）

スレッドダンプファイルは，出力先を変更できます。

出力先を変更する場合は，JAVACOREDIR 環境変数に出力先を指定してください。JAVACOREDIR 環境変数の詳細については，「5.6　日立 JavaVM で使用する環境変数の詳細」を参照してください。

メモ

コマンドに指定するプロセス ID は，次の方法で取得できます。

```
ps -ef | grep <識別名>
```

スレッドダンプを取得するプロセスの識別ができるように，プロセス起動時のオプションに識別可能な任意の名称（-D“<識別名>”）を指定してください。

任意の識別名（JavaServer01）でスレッドダンプを取得する例を示します。

```
$ java -DJavaServer01 A-Server &
$ ps -ef | grep JavaServer01
root     24249 24165 99 14:45 pts/1    00:17:57 java -DJavaServer01 A-Server
 :
$ kill -3 24249
```

なお，スレッドダンプには，コマンドラインが出力されています。コマンドライン中の-D“<識別名>”で，どのサーバのスレッドダンプであるかを判別できます。

```
Fri Jul 19 13:27:07 2019

Full thread dump Java HotSpot(TM) 64-Bit Server VM (11+28-CDK0987-20190712 mixed m
ode)
/opt/Cosminexus/jdk/bin/java -D[Server:server-one] -Xms64m -Xmx512m -server …

System Properties
 :
```

(4)　JavaVM ログファイル

JavaVM ログファイルを取得するための設定について説明します。

JavaVM ログとは，製品が標準の JavaVM に追加した拡張オプションを使用して取得できるログです。標準の JavaVM よりも，多くのトラブルシュート情報が取得できます。このファイルには，JavaVM の GC のログも出力されます。

JavaVM ログファイルの出力先および出力ファイル名は，次のとおりです。カレントディレクトリに「javalogxx.log」という名称で出力されます。xx は 01 から始まる 2桁の通番です。

```
<カレントディレクトリ>/javalogxx.log
```

JavaVM ログファイルは，出力先およびファイル名を変更できます。

出力先を変更する場合は，-XX:HitachiJavaLog:＜ディレクトリおよびファイル名称＞オプションを指定してください。指定した出力先に指定したファイル名称で出力されます。

-XX:HitachiJavaLog オプションの詳細については，「5.2　日立 JavaVM 拡張オプションの詳細」を参照してください。

(5)　OS 稼働情報

次に示すコマンドを実行して CPU 利用率およびメモリ使用量を取得します。60 秒間隔で取得することを推奨します。取得間隔を長くすると，情報の取得による性能劣化を少なくできますが，情報の精度が低下することがあります。

```
vmstat -n <実行間隔(秒)> <繰り返し回数>
top -d <実行間隔(秒)> -n <繰り返し回数>
sar -A <実行間隔(秒)> <繰り返し回数>
```

3.1.2　トラブル事象別の原因切り分け方法

事象別に原因の切り分け方法を説明します。

- プロセスダウン
- プロセスハングアップ（無応答）
- プロセススローダウン
- OutOfMemoryError 障害

(1)　プロセスダウン

JavaVM を実行しているプロセスがダウンをしたときの原因を切り分ける方法を説明します。

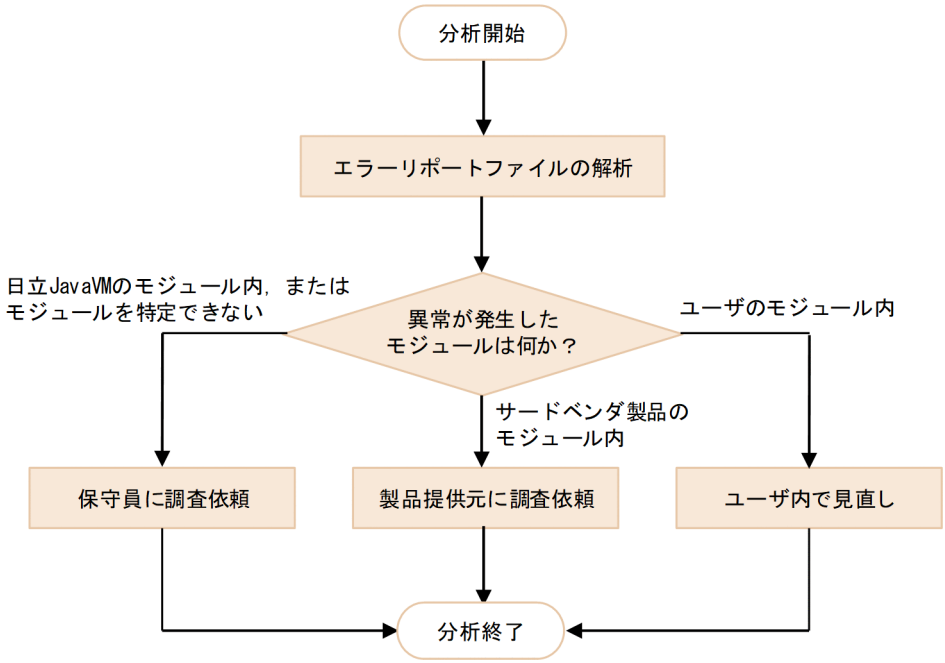
ここでの説明で使用する資料について，取得方法の参照先は次のとおりです。

取得情報	参照先
エラーリポートファイル	「3.1.1(1)　エラーリポートファイル」
core ダンプ／ スタックトレースファイル	「3.1.1(2)　core ダンプ／スタックトレースファイル」

(a)　調査の流れ

プロセスがダウンをしたときに，原因を切り分けるための調査の流れを次の図に示します。

図 3-1　プロセスダウン時の調査の流れ



【調査ポイント】

ダウンした個所を特定し，該当モジュールの提供元に調査を依頼します。

【有効な資料】

- エラーリポートファイル
- core ダンプ／スタックトレースファイル

1. エラーリポートファイルを解析し，ダウンした個所を特定します。
   ユーザのモジュール内と特定できた場合は，ダウン原因がないかどうかを見直します。
   サードベンダ製品と特定できた場合は，製品提供元に調査を依頼します。
   特定できない場合は，保守員に調査を依頼します。
2. core ファイルが出力されている場合は，スタックトレースの情報を取得します。
   保守員に調査を依頼するときの資料です。

(b)　エラーリポートファイルの解析

エラーリポートファイルを解析します。

```
#
# A fatal error has been detected by the Java Runtime Environment:
#
# SIGSEGV (0xb) at pc=0x00007fd4f526960d, pid=23725, tid=23726
#                    ポイント3
# JRE version: Java(TM) SE Runtime Environment (11.0+28) (build 11+28-CDK0987-20190712)
# Java VM: Java HotSpot(TM) 64-Bit Server VM (11+28-CDK0987-20190712, mixed mode, serial gc, linux-amd64)
# Problematic frame:
# C [/home/jdk/APS/libUser.so+0x60d] Index_check+0x91    ポイント2
#   ポイント1
# Core dump will be written. Default location: Core dumps may be processed with "/usr/libexec/abrt-
hookccpp %s %c %p %u %g %t e" (or dumping to /home/jdk/APS/core.23725)
#

 :

---------- T H R E A D ----------

Current thread (0x00007fd4f9360000): JavaThread "main" [_thread_in_native, id=23726,
stack(0x00007fd4f7a0d000,0x00007fd4f7b0e000)]

Java frames: (J=compiled Java code, j=interpreted, Vv=VM code)
j User.UserNameCheck()V+0 at User.UserNameCheck(Native Method)

j User.name()V+5 at User.name(User.java:16)
                                    ポイント4
j User_select.main([Ljava/lang/String;)V+12 at User_select.main(User_select.java:20)

v ~StubRoutines::call_stub
```

【解析のポイント】

- ポイント 1

  JavaVM 以外で異常終了（ダウンしたフレームの種別）

- ポイント 2

  ダウンしたライブラリとライブラリ内の位置

- ポイント 3

  ダウン原因の例外コード

- ポイント 4

  ネイティブメソッドが呼び出された Java プログラムのファイル名と行番号

この例では，ダウンしたライブラリは/home/jdk/APS/User.so で，ダウンしたライブラリ内の位置は「0x91」です。

/home/jdk/APS/User.so の提供元に調査を依頼します。

- エラーリポートファイルでの確認点

  JavaVM 以外の異常終了かどうか確認する場合について説明します。

  【解析のポイント】のポイント 1 に示すフレームの種別が“C”の場合は，「JavaVM 以外の異常終了」を示します。“C”ではない場合（“V”や“j”などがある）は，JavaVM のフレームです。フレームの種別を次に示します。

表 3-4　フレームの種別

フレームの種別	フレーム
C	ネイティブの C フレーム

フレームの種別	フレーム
J	C フレーム以外のフレーム（コンパイルされた Java のフレームも含む）
j	インタプリタで実行される Java のフレーム
V	JavaVM のフレーム
v	JavaVM が生成したスタブのフレーム

- スタックトレースの情報の解析

  JavaVM が異常終了して core ダンプが出力された場合，異常終了した原因の究明に必要なネイティブ関数の情報（スタックトレース情報）を，javatrace コマンドで取得できます。全スレッドのスタックトレース情報が出力されます。ダウンしたライブラリの提供元が解析する資料は，ダウンしたモジュールを特定するための参考情報となることがあります。

```
 :

* CORE DUMP INFORMATION(00): signal=11 si-code=0 errno=0
 Command: /home/Java/SI/09-87JDK/opt/Cosminexus/jdk/bin/java
  rax = 00000000.00000000, rbx = 00007fd4.f6aefca0
  rcx = 00007fd4.f5d348d0, rdx = 00000000.00000002
  rsi = 00007fd4.f7b0c8d0, rdi = 00007fd4.f93603c0
  rbp = 00007fd4.f7b0c820, rsp = 00007fd4.f7b0c820
  r8 = 00007fd4.f6aefca0, r9 = 00000000.00000001
  r10 = 00007fd4.f7b0c5b0, r11 = 00007fd4.f526957c
  r12 = 00007fd4.f7b0cc20, r13 = 00007fd4.f6aefca0
  r14 = 00007fd4.f7b0c8d8, r15 = 00007fd4.f9360000
  cs = 00000000.00000033, ds = 00000000.00000000
  ss = 00000000.0000002b, es = 00000000.00000000
  fs = 00000000.00000000, gs = 00000000.00000000
  eflags = 00000000.00010202, rip = 00007fd4.f526960d
BACKTRACE:
  sp=00007fd4.f7b0c820(-010) rp=00007fd4.f526960d :libUser.so:Index_check()+91   ポイント1
  sp=00007fd4.f7b0c830(-020) rp=00007fd4.f5269657 :libUser.so:get_name()+42      ポイント2
  sp=00007fd4.f7b0c850(-020) rp=00007fd4.f5269673 :libUser.so:Java_User_UserNameCheck()+1a

 :
```

【解析のポイント】

- ポイント 1

  ダウンしたライブラリとダウンした関数と個所

- ポイント 2

  ダウンした関数の呼び出し元

この例では「Java_User_UserNameCheck 関数」，「get_name 関数」，「Index_check 関数」の順で呼び出され「Index_check 関数」で異常が発生しています。

(2) プロセスハングアップ（無応答）

JavaVM を実行しているプロセスがハングアップしたときの原因を切り分ける方法を説明します。

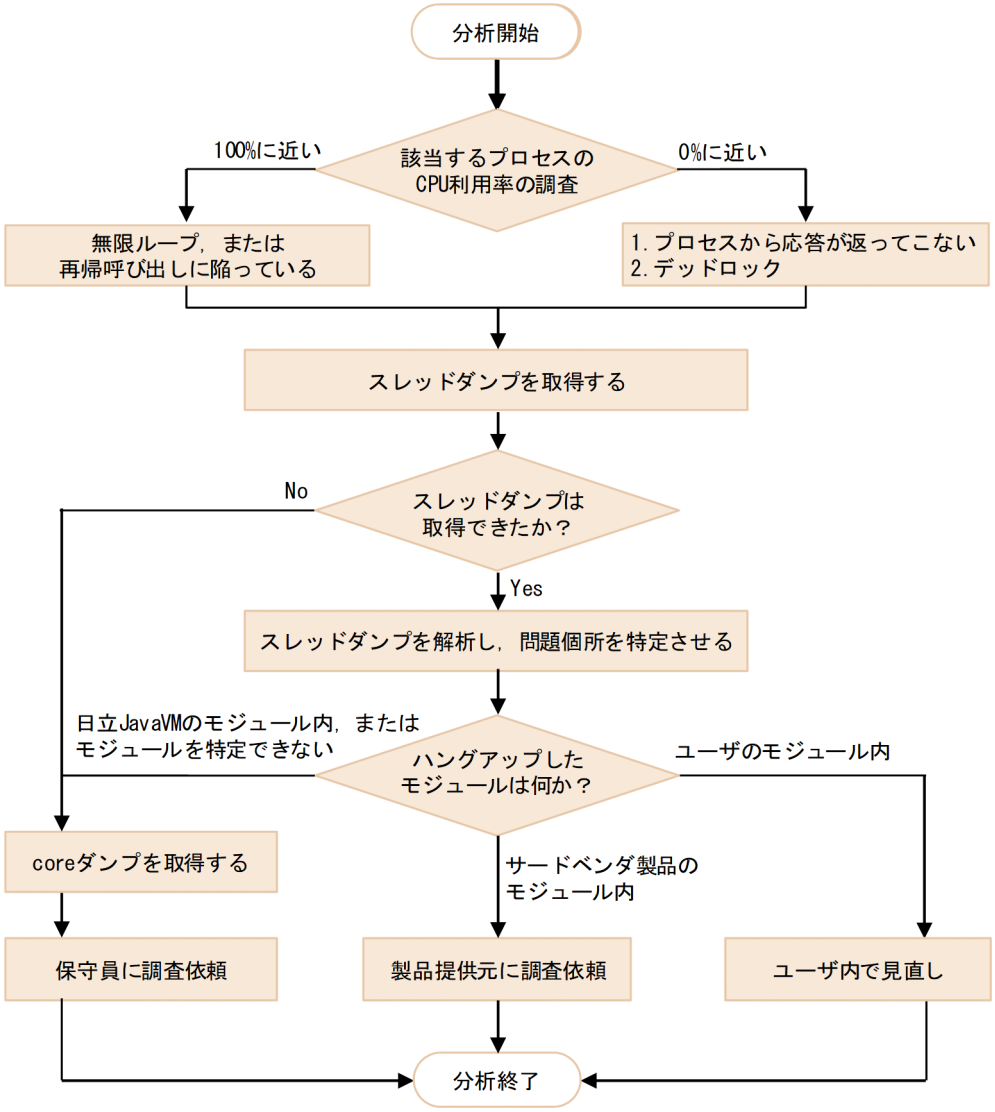
ここでの説明で使用する資料について，取得方法の参照先は次のとおりです。

取得情報	参照先
CPU 利用率	「3.1.1(5)　OS 稼働情報」
スレッドダンプ	「3.1.1(3)　JavaVM のスレッドダンプファイル」
core ダンプ	「3.1.1(2)　core ダンプ／スタックトレースファイル」

(a)　調査の流れ

プロセスがハングアップをしたときに，原因を切り分けるための調査の流れを次の図に示します。

図 3-2　プロセスハングアップ時の調査の流れ



【調査ポイント】

該当するプロセスの CPU 使用率と突き合わせて，原因を推定し，スレッドダンプで特定します。

【有効な資料】

- OS 稼働情報
- スレッドダンプ

1. CPU 利用率の調査をします。

   該当するプロセスの CPU 利用率の調査をします。

   100%に近い場合は，無限ループや再帰呼び出しに陥っている可能性が高いです。

   0%に近い場合は，次の 2 点が考えられます。

   - プロセスから応答が返ってこない
   - デッドロック

   この場合「プロセスから応答が返ってこない」の方が多くみられます。

2. スレッドダンプを取得します。
3. JavaVM 自身がハングアップしているとスレッドダンプが取得できない場合があります。
4. スレッドダンプが取得できた場合は，スレッドダンプを解析しハングアップしたモジュールを特定し，モジュールの提供元に調査依頼をします。
5. スレッドダンプが取得できない場合は，javacore コマンドで core ダンプを取得して，保守員に調査依頼をします。

   core ダンプ取得時，ヒープ領域が大きいプロセスでは，プロセスの動作を一時的に停滞させることがあるので注意してください。

(b) スレッドダンプ（プロセス無応答）の解析例

ネイティブメソッドの実行応答が返ってこないため，ハングアップが発生している例を次の図に示します。

スレッドダンプファイル1

```
"main" #1 prio=5 os_prio=0 jid=<N/A> tid=0x00002aaaabded000 nid=0x5df0 runnable
[0x00000000400ff000..0x00000000400ffe40]                    ポイント2          ポイント1
   java.lang.Thread.State: RUNNABLE
  stack=[0x0000000040101000..0x0000000040003000..0x0000000040001000..0x0000000040000000]
  [user cpu time=23470ms, kernel cpu time=30ms] [blocked count=0, waited count=0]
                at User.UserNameCheck(Native Method)          ポイント3
                at User.name(User.java:16)
                at User_select.main(User_select.java:20)
        :
```

スレッドダンプファイル2

```
"main" #1 prio=5 os_prio=0 jid=<N/A> tid=0x00002aaaabded000 nid=0x5df0 runnable
[0x00000000400ff000..0x00000000400ffe40]                    ポイント2          ポイント1
   java.lang.Thread.State: RUNNABLE
  stack=[0x0000000040101000..0x0000000040003000..0x0000000040001000..0x0000000040000000]
  [user cpu time=28370ms, kernel cpu time=30ms] [blocked count=0, waited count=0]
                at User.UserNameCheck(Native Method)          ポイント3
                at User.name(User.java:16)
                at User_select.main(User_select.java:20)
        :
```

スレッドダンプファイル3

```
"main" #1 prio=5 os_prio=0 jid=<N/A> tid=0x00002aaaabded000 nid=0x5df0 runnable
[0x00000000400ff000..0x00000000400ffe40]                    ポイント2          ポイント1
   java.lang.Thread.State: RUNNABLE
  stack=[0x0000000040101000..0x0000000040003000..0x0000000040001000..0x0000000040000000]
  [user cpu time=31740ms, kernel cpu time=30ms] [blocked count=0, waited count=0]
                at User.UserNameCheck(Native Method)          ポイント3
                at User.name(User.java:16)
                at User_select.main(User_select.java:20)
        :
```

【解析のポイント】

スレッドダンプを複数回取得して時系列で観察し，それぞれのスレッドダンプで tid（スレッド ID）が同じスレッドのスタックトレースを比較調査します。

- ポイント 1

  スレッド属性が runnable の場合，このスレッドは実行可能状態にあります。

  このスレッドが CPU 利用率の増加に関与しています（wait for monitor entry である場合は CPU 利用率を増加させません）。

- ポイント 2

  複数のスレッドダンプファイルで，同じ tid のスレッド属性がすべて runnable です。長時間にわたって実行中となっている可能性があります。

- ポイント 3

  複数のスレッドダンプファイルで，同じ tid のスレッドがすべて同じ場所を実行中です。

この例では，Native Method 実行の応答が返ってこないためハングアップが発生している可能性が高いです。

(c) スレッドダンプ（デッドロック）の解析例

デッドロックが発生しているスレッドダンプの例を次の図に示します。

```
"Thread-2" #11 prio=5 os_prio=0 jid=<N/A> tid=0x00002aaab92e1800 nid=0x51b9 runnable
[0x0000000040d0b000..0x0000000040d0b9b0]                                    ポイント1
   java.lang.Thread.State: RUNNABLE
  stack=[0x0000000040d0d000..0x0000000040c0f000..0x0000000040c0d000..0x0000000040c0c000]
  [user cpu time=2060ms, kernel cpu time=0ms] [blocked count=0, waited count=0]
                at Thread3.methodC(UserName.java:75)
                at Thread3.run(UserName.java:71)

"Thread-1" #10 prio=5 os_prio=0 jid=<N/A> tid=0x00002aaab92f2000 nid=0x51b8 waiting for monitor entry
[0x0000000040c0a000..0x0000000040c0aa30]                                    ポイント2
   java.lang.Thread.State: BLOCKED (on object monitor)
  stack=[0x0000000040c0c000..0x0000000040b0e000..0x0000000040b0c000..0x0000000040b0b000]
  [user cpu time=0ms, kernel cpu time=0ms] [blocked count=3, waited count=0]
                at Thread2.methodA(UserName.java:53)
                - waiting to lock <0x00002aaaaf886628> (a [B)
                - locked <0x00002aaaaf886648> (a [B)        ポイント3
                at Thread2.run(UserName.java:46)

"Thread-0" #9 prio=5 os_prio=0 jid=<N/A> tid=0x00002aaab9304800 nid=0x51b7 waiting for monitor entry
[0x0000000040b09000..0x0000000040b09ab0]
   java.lang.Thread.State: BLOCKED (on object monitor)
  stack=[0x0000000040b0b000..0x0000000040a0d000..0x0000000040a0b000..0x0000000040a0a000]
  [user cpu time=0ms, kernel cpu time=0ms] [blocked count=1, waited count=0]
                at Thread1.methodB(UserName.java:26)
                - waiting to lock <0x00002aaaaf886648> (a [B)
                - locked <0x00002aaaaf886628> (a [B)
                at Thread1.run(UserName.java:19)
```

【解析のポイント】

- ポイント 1

  スレッド属性が runnable の場合，このスレッドは実行可能状態にあります。

  このスレッドはデッドロックとは無関係です。

- ポイント 2

  スレッド属性が wait for monitor entry の場合，このスレッドはロックの取得待ちであることを示しています。

  デッドロックを起こしているスレッドである可能性があります。

- ポイント 3

  スレッドがロックを取得していて，かつポイント 2 でスレッドのロック取得待ちである場合，デッドロックを起こしているスレッドである可能性が高いです。

ポイント 2，3 に当てはまるスレッドに対し，ロックしているオブジェクトのアドレスを突き合わせながらデッドロックを検出します。

この例の場合，Thread-1 は，＜ 0x00002aaaaf886648 ＞のロックを取得して，＜ 0x00002aaaaf886628 ＞の取得待ちを示します。一方，Thread-0 は，＜ 0x00002aaaaf886628 ＞のロックを取得していて，かつ＜ 0x00002aaaaf886648 ＞の取得待ちとなっています。Thread-1 と Thread-0 がデッドロックしていることが分かります。

(d) スレッドダンプ（無限ループ）の解析例

無限ループが発生しているスレッドダンプの例を次の図に示します。

スレッドダンプファイル1

```
"main" #1 prio=5 os_prio=0 jid=<N/A> tid=0x00002aaaabded000 nid=0x537e runnable
[0x00000000400ff000..0x00000000400ffe40]          ポイント2          ポイント1
   java.lang.Thread.State: RUNNABLE
  stack=[0x0000000040101000..0x0000000040003000..0x0000000040001000..0x0000000040000000]
  [user cpu time=6790ms, kernel cpu time=140ms] [blocked count=0, waited count=0]
                at java.lang.String.<init>(String.java:644)
                at UserSerach.methodMain(UserSerach.java:17)
                at UserSerach.LoadObject(UserSerach.java:10)          ポイント3
                at UserSerach.main(UserSerach.java:6)
```

スレッドダンプファイル2

```
"main" #1 prio=5 os_prio=0 jid=<N/A> tid=0x00002aaaabded000 nid=0x537e runnable
[0x00000000400ff000..0x00000000400ffe40]          ポイント2          ポイント1
   java.lang.Thread.State: RUNNABLE
  stack=[0x0000000040101000..0x0000000040003000..0x0000000040001000..0x0000000040000000]
  [user cpu time=1280ms, kernel cpu time=50ms] [blocked count=0, waited count=0]
                at java.util.Arrays.copyOf(Arrays.java:3332)
                at
java.lang.AbstractStringBuilder.ensureCapacityInternal(AbstractStringBuilder.java:125)
                at java.lang.AbstractStringBuilder.append(AbstractStringBuilder.java:449)
                at java.lang.StringBuilder.append(StringBuilder.java:136)
                at UserSerach.methodMain(UserSerach.java:16)
                at UserSerach.LoadObject(UserSerach.java:10)          ポイント3
                at UserSerach.main(UserSerach.java:6)
```

スレッドダンプファイル3

```
"main" #1 prio=5 os_prio=0 jid=<N/A> tid=0x00002aaaabded000 nid=0x5370 runnable
[0x00000000400ff000..0x00000000400ffe40]          ポイント2          ポイント1
   java.lang.Thread.State: RUNNABLE
  stack=[0x0000000040101000..0x0000000040003000..0x0000000040001000..0x0000000040000000]
  [user cpu time=1660ms, kernel cpu time=70ms] [blocked count=0, waited count=0]
                at PrintControl.methodA(UserSerach.java:26)
                at UserSerach.methodMain(UserSerach.java:19)
                at UserSerach.LoadObject(UserSerach.java:10)          ポイント3
                at UserSerach.main(UserSerach.java:6)
```

【解析のポイント】

スレッドダンプを複数回取得して時系列で観察し，それぞれのスレッドダンプで tid（スレッド ID）が同じスレッドのスタックトレースを比較調査します。

- ポイント 1

  スレッド属性が runnable の場合，このスレッドは実行可能状態にあります。このスレッドが CPU 利用率の増加に関与しています（wait for monitor entry である場合は CPU 利用率を増加させません）。

- ポイント 2

  複数のスレッドダンプファイルで，同じ tid のスレッド属性がすべて runnable です。長時間にわたって実行中となっている可能性があります。

- ポイント 3

  同一メソッド内の特定の行が複数回繰り返し実行されているような場合，無限ループの可能性があります。

(e) スレッドダンプ（無限ループ）のローカル変数情報による解析例

プログラムがローカル変数に作成したカウンタの使い方を誤ったために，無限ループが発生しているスレッドダンプの例を次の図に示します。

プログラム

```
■ローカル変数countの値でループ処理
10:     int count = sharedData.getCount();
11:     while (count < sharedData.getLength()) {      ポイント1
12:         int skip = sharedData.doTask(); // 処理量を返す
13:         sharedData.setCount(count + skip); // 処理した分はskip
14:         count = sharedData.getCount();
15:     }
          :
```

スレッドダンプファイル1

```
(正常)
at com.hitachi.software.DataProcessor(DataProcessor.java:13)
    (com.hitachi.software.SharedData) sharedData [arg1] = <0x00002aaaaf882a18>
    (int) count = 24      ポイント2
    (int) skip = 8
        :
```

スレッドダンプファイル2

```
(正常)
at com.hitachi.software.DataProcessor(DataProcessor.java:13)
    (com.hitachi.software.SharedData) sharedData [arg1] = <0x00002aaaaf882a18>
    (int) count = 64      ポイント2
    (int) skip = 16
        :
```

スレッドダンプファイル3

```
(異常)
at com.hitachi.software.DataProcessor(DataProcessor.java:15)
    (com.hitachi.software.SharedData) sharedData [arg1] = <0x00002aaaaf882a18>
    (int) count = 16      ポイント2
    (int) skip = 8
        :
```

【解析のポイント】

スレッドダンプを複数回取得して時系列で観察し，それぞれのスレッドダンプで tid（スレッド ID）が同じスレッドでループカウンタなどのループ処理に影響するローカル変数の値を確認します。

- ポイント 1

  プログラムで，ループの動作に関与するローカル変数を確認します。

- ポイント 2

  スレッドダンプに出力されている，該当するローカル変数の値を確認します。

この例では，ローカル変数 count の値が増えてループ処理が抜けることを期待したプログラムにもかかわらず，count の値が減少しています。

(3)　プロセススローダウン

JavaVM を実行しているプロセスがスローダウンをしたときの原因を切り分ける方法を説明します。

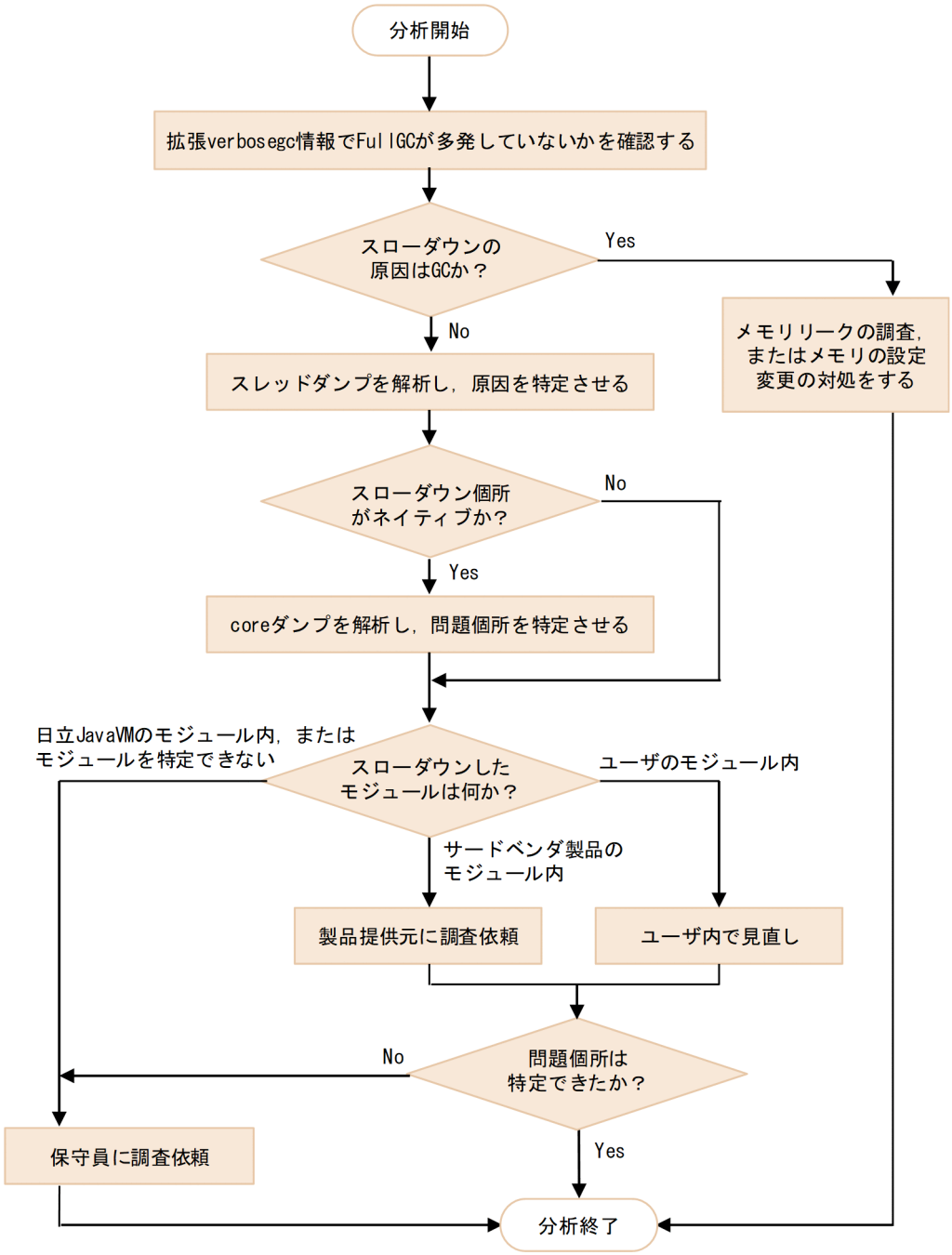
ここでの説明で使用する資料取得方法の参照先は次のとおりです。

取得情報	参照先
JavaVM ログファイル	「3.1.1(4)　JavaVM ログファイル」
スレッドダンプ	「3.1.1(3)　JavaVM のスレッドダンプファイル」
core ダンプ	「3.1.1(2)　core ダンプ／スタックトレースファイル」
CPU 利用率	「3.1.1(5)　OS 稼働情報」

(a)　調査の流れ

プロセスがスローダウンをしたときに，原因を切り分けるための調査の流れを次の図に示します。

図 3-3　プロセススローダウン時の調査の流れ



【調査ポイント】

拡張 verbosegc 情報とスレッドダンプで特定します。

【有効な資料】

- JavaVM ログファイル（拡張 verbosegc 情報）
- スレッドダンプ
- core ダンプ
- OS 稼働情報

1. 拡張 verbosegc 情報で，FullGC が多発していないかどうかを確認します。
   拡張 verbosegc 情報は JavaVM ログファイルに出力されます。
2. GC が原因の場合は，メモリリークを調査します。
   メモリリークがない場合はメモリの設定変更の対処をします。
3. GC が原因ではない場合は，スレッドダンプを解析し，問題個所を特定します。
4. スレッドダンプの解析の結果，スローダウンをしている個所がネイティブメソッドの場合，さらに core ダンプの解析をします。
5. core ダンプの解析の結果，スローダウンをしたモジュールを特定し，各モジュールの開発元に調査を依頼します。
6. サードベンダ製品を含めて問題個所を特定できない場合は，保守員に調査依頼をします。

メモリリークの調査方法は，OutOfMemory 障害の時と同じ方法を用います。

スレッドダンプの調査方法は，プロセスハングアップの時と同じ方法を用います。

(4) OutOfMemoryError 障害

OutOfMemoryError 障害が発生したときの原因を切り分ける方法を説明します。

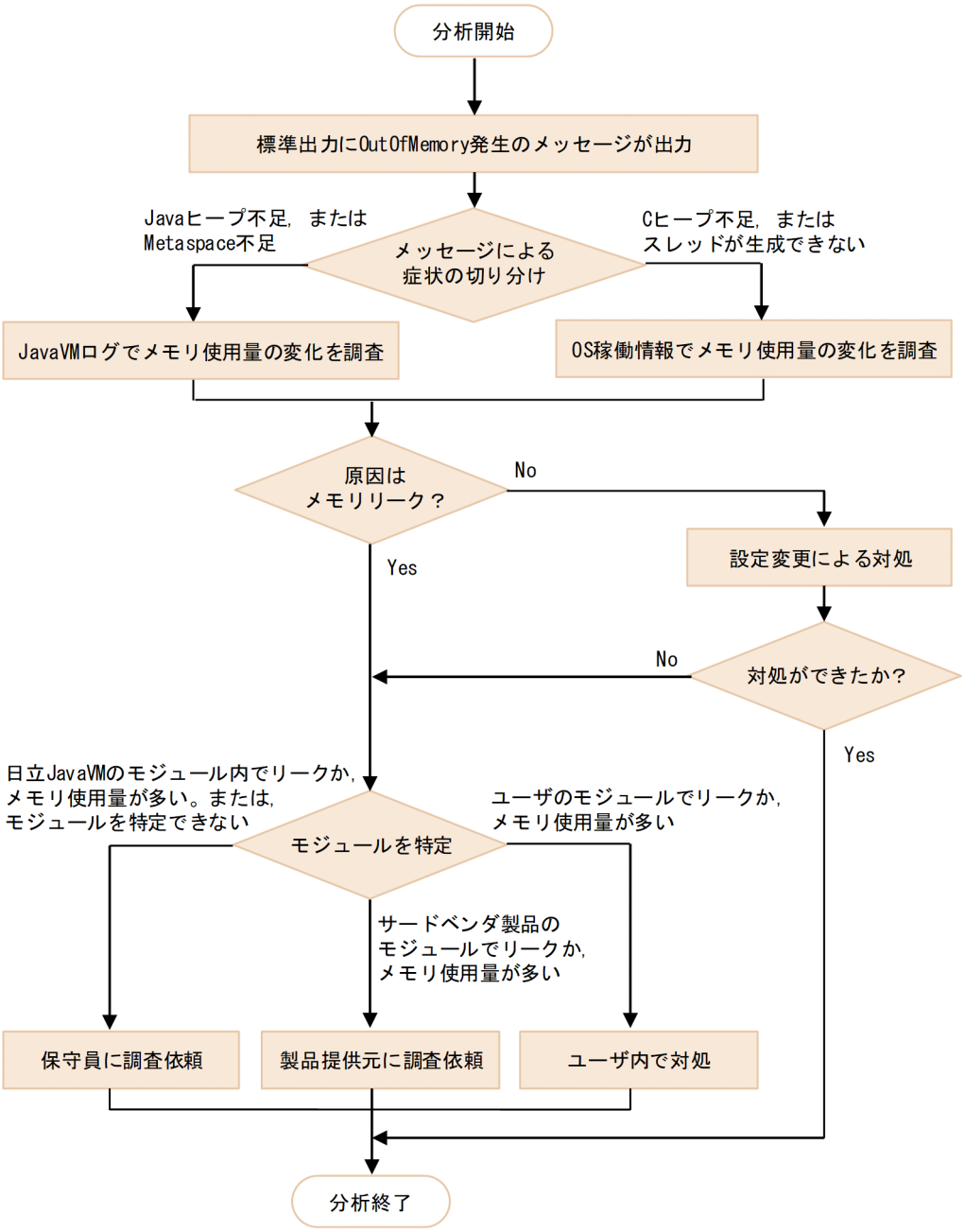
資料取得方法の参照先は，次のとおりです。

取得情報	参照先
JavaVM ログファイル	「3.1.1(4) JavaVM ログファイル」
メモリ使用量	「3.1.1(5) OS 稼働情報」
スレッドダンプ	「3.1.1(3) JavaVM のスレッドダンプファイル」
core ダンプ	「3.1.1(2) core ダンプ／スタックトレースファイル」

(a) 調査の流れ

OutOfMemoryError 障害が発生したときに，原因を切り分けるための調査の流れを次の図に示します。

図 3-4　OutOfMemory 障害の調査の流れ



【調査ポイント】

該当する症状に応じて切り分けたあと，メモリ量の変化を調査します。

【有効な資料】

- JavaVM ログファイル
- OS 稼働情報

1. 症状の切り分けをします。

   メモリ不足が発生すると標準出力に OutOfMemoryError が出現します。標準出力に出力されるメッセージに詳細が記述される場合があるため，そちらを参照し症状の切り分けをします。

2. メモリ使用量の変化を調査します。

   症状が「C ヒープ不足」または「スレッドが生成できない」場合には，OS 稼働情報を使ってメモリ使用量の変化を調査します。症状が「Java ヒープ不足」および「Metaspace 不足」の場合には，JavaVM ログの拡張 verbosegc 情報を使ってメモリ使用量の変化を調整します。

3. メモリ使用量の変化を観察した結果，メモリリークではない場合は，設定変更による対処をします。

   設定変更による対処ができない場合は，メモリ使用量の多いモジュールを特定し，モジュールの提供元に調査依頼をします。

4. メモリリークの場合は，メモリリークしているモジュールを特定します。

   サードベンダ製品を含めてリークが見当たらない場合は，保守員に調査依頼をします。

(b)　メッセージによる症状の切り分け

Java ヒープ，Metaspace が確保できなかった場合

各メモリ領域が不足した場合のメッセージの例を示します。

● Javaヒープが確保できなかった場合

```
【java.lang.Throwable.printStackTrace()の出力】
java.lang.OutOfMemoryError: Java heap space

【日立ログ出力(-XX:+HitachiOutOfMemoryStackTrace指定時)】
[OOM][Thread: 0x00007f8eef082000]<Fri Jul 19 18:13:32 2019>[java.lang.OutOfMemoryError : requested
32000024 bytes. (Java Heap) : 10 threads exist]
                                                    “Java Heap”と出力される
[OOM][Thread: 0x00007f8eef082000] at S150100.fn(S150100.java:13)
[OOM][Thread: 0x00007f8eef082000] at S150100.main(S150100.java:3)
```

● Metaspaceが確保できなかった場合

```
【java.lang.Throwable.printStackTrace()の出力】
Exception in thread "Thread-15" java.lang.OutOfMemoryError: Metaspace

【日立ログ出力(-XX:+HitachiOutOfMemoryStackTrace指定時)】
[OOM][Thread: 0x00007fe22eb35000]<Fri Jul 19 18:15:58 2019>[java.lang.OutOfMemoryError : requested 32
bytes. (Meta Space) : 10 threads exist]
                                                    “Meta Space”と出力される
[OOM][Thread: 0x00007fe22eb35000] at java.lang.ClassLoader.defineClass1(Native Method)
[OOM][Thread: 0x00007fe22eb35000] at java.lang.ClassLoader.defineClass(ClassLoader.java:1126)
[OOM][Thread: 0x00007fe22eb35000] at
java.security.SecureClassLoader.defineClass(SecureClassLoader.java:180)
[OOM][Thread: 0x00007fe22eb35000] at
jdk.internal.loader.BuiltinClassLoader.defineClass(BuiltinClassLoader.java:801)
[OOM][Thread: 0x00007fe22eb35000] at
jdk.internal.loader.BuiltinClassLoader.findClassOnClassPathOrNull(BuiltinClassLoader.java:699)
[OOM][Thread: 0x00007fe22eb35000] at
jdk.internal.loader.BuiltinClassLoader.loadClassOrNull(BuiltinClassLoader.java:622)
[OOM][Thread: 0x00007fe22eb35000] at
jdk.internal.loader.BuiltinClassLoader.loadClass(BuiltinClassLoader.java:580)
[OOM][Thread: 0x00007fe22eb35000] at
jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(ClassLoaders.java:181)
[OOM][Thread: 0x00007fe22eb35000] at java.lang.ClassLoader.loadClass(ClassLoader.java:631)
[OOM][Thread: 0x00007fe22eb35000] at java.lang.Class.forName0(Native Method)
[OOM][Thread: 0x00007fe22eb35000] at java.lang.Class.forName(Class.java:403)
[OOM][Thread: 0x00007fe22eb35000] at sun.launcher.LauncherHelper.loadMainClass(LauncherHelper.java:760)
[OOM][Thread: 0x00007fe22eb35000] at
sun.launcher.LauncherHelper.checkAndLoadMain(LauncherHelper.java:655)
```

- Java ヒープが確保できなかった場合

  JavaVM ログファイルの詳細情報に“Java Heap”と出力されます。

- Metaspace が確保できなかった場合

JavaVM ログファイルの詳細情報に“Meta Space”と出力されます。

C ヒープが確保できなかった場合

C ヒープが確保できなかった場合のメッセージの例を示します。

● Cヒープが確保できなかった場合

ネイティブライブラリ（JNI）内でC Heapを確保できなかった場合

```
【java.lang.Throwable.printStackTrace()の出力】
java.lang.OutOfMemoryError: unable to create new native thread. 1020 threads exist.
        at java.base/java.lang.Thread.start0(Native Method)
        at java.base/java.lang.Thread.start(Thread.java:804)
        at S150100_kai.main(S150100_kai.java:11)

【日立ログ出力（-XX:+HitachiOutOfMemoryStackTrace指定時）】
[OOM][Thread: 0x00007f6c761b9000]<Fri Jul 19 18:08:21 2019>[java.lang.OutOfMemoryError : requested
1048576 bytes. (C Heap) : unable to create thread : 1020 threads exist]
                                                                    “C Heap”と出力される
[OOM][Thread: 0x00007f6c761b9000] at java.lang.Thread.start0(Native Method)
[OOM][Thread: 0x00007f6c761b9000] at java.lang.Thread.start(Thread.java:804)
[OOM][Thread: 0x00007f6c761b9000] at S150100_kai.main(S150100_kai.java:11)
```

ネイティブライブラリ（java.awt.font）内でC Heapを確保できなかった場合

```
【標準出力】
java.lang.OutOfMemoryError

【日立ログ出力（-XX:+HitachiOutOfMemoryStackTrace指定時）】
出力はなし。

【その他】
エラーリポートファイルとメモリダンプが出力される。
```

JavaVM自身がC Heapを確保できなかった場合

```
【標準出力】
Exception in thread "VMThread" java.lang.OutOfMemoryError: requested 64000 bytes

【JavaVMログの出力（-XX:+HitachiOutOfMemoryStackTrace指定時）】
出力はなし。
                                “requested サイズ bytes”と出力される
```

- JavaVM のネイティブライブラリ（JNI）内で C ヒープが確保できなかった場合

  JavaVM ログファイルの詳細情報に“C Heap”と出力されます。

- ネイティブライブラリ（java.awt.font）内で C ヒープが確保できなかった場合

  標準出力で java.lang.OutOfMemoryError が出力されます。エラーリポートファイルと core ダンプが出力されます。エラーリポートファイルの内容は，標準出力にも出力されます。

- JavaVM 自身が C ヒープの確保ができなかった場合

  OutOfMemoryError 例外の詳細情報に“requested *xxx* bytes”と出力されます。

ユーザプログラム内で OutOfMemoryError を発生させた場合

ユーザプログラム内で OutOfMemoryError を発生させた場合のメッセージの例を示します。

● ユーザプログラム内でOutOfMemoryErrorを発生させた場合

```
【標準出力】
Exception in thread "main" java.lang.OutOfMemoryError: oom
        at MultiThread.main(MultiThread.java:312)

【日立ログ出力（-XX:+HitachiOutOfMemoryStackTrace指定時）】
[OOM][Thread: 0x00007f1324db5800]<Fri Jul 19 18:25:58 2019>[java.lang.OutOfMemoryError :(Unknown) : 10
threads exist]
[OOM][Thread: 0x00002aaaabded800] at MultiThread.main(MultiThread.java:312)
                                                          “Unknown”と出力される
```

- ユーザプログラム内で明示的に OutOfMemoryError を発生させた場合，またはユーザのネイティブライブラリ（JNI）内で C ヒープが確保できなかった場合など，JavaVM が OutOfMemoryError 発生要因を特定できないとき，JavaVM ログファイルの詳細情報に“Unknown”と出力されます。

(c) 各使用量の分析

Java ヒープ使用量の分析

Java ヒープ使用量を分析した例を示します。

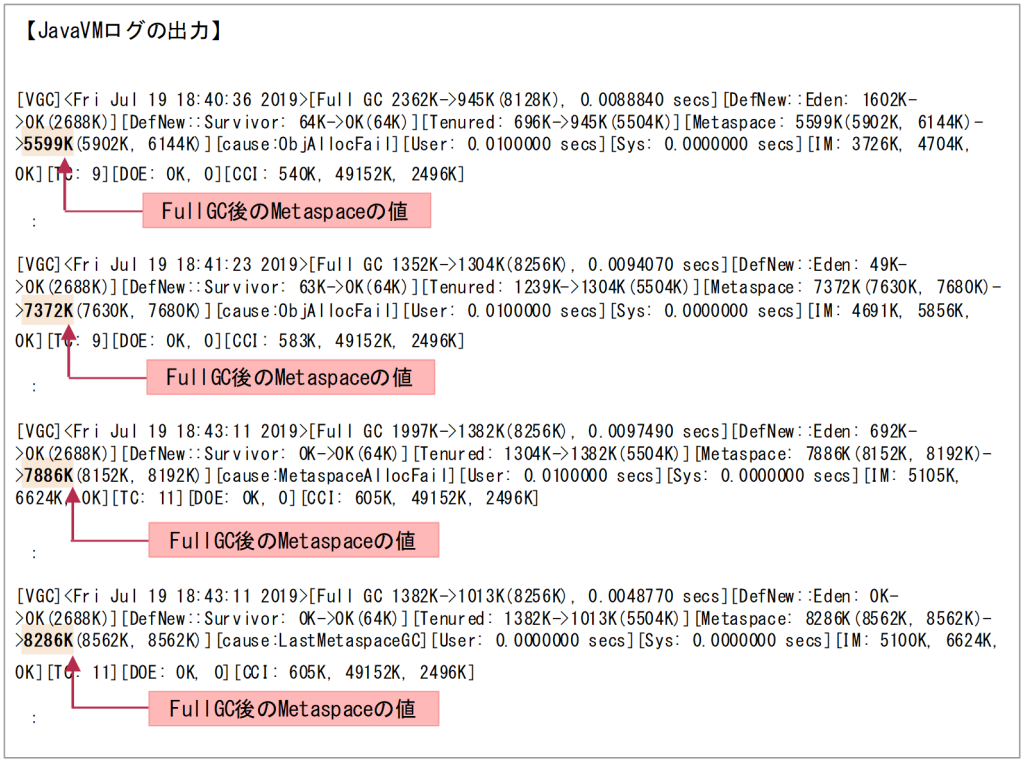
```
【JavaVMログ（拡張verbosegc情報）の出力】

[VGC]<Fri Jul 19 17:53:33 2019>[Full GC 15780K->15780K(20684K), 0.0285540 secs][DefNew::Eden: 0K-
>0K(24832K)][DefNew::Survivor: 0K->0K(768K)][Tenured: 15780K->15780K(52644K)][Metaspace: 2828K(4100K,
4352K)->2828K(4100K, 4352K)][cause:ObjAllocFail][User: 0.0300000 secs][Sys: 0.0000000 secs][IM: 2413K,
2848K, 0K][TC: 9][DOE: 0K, 0][CCI: 473K, 49152K, 2496K]                    FullGC後のTenuredの値

 :

[VGC]<Fri Jul 19 17:54:48 2019>[Full GC 31585K->31566K(43980K), 0.0323550 secs][DefNew::Eden: 0K-
>0K(27008K)][DefNew::Survivor: 0K->0K(832K)][Tenured: 31585K->31566K(57344K)][Metaspace: 2828K(4100K,
4352K)->2828K(4100K, 4352K)][cause:ObjAllocFail][User: 0.0300000 secs][Sys: 0.0000000 secs][IM: 2413K,
2848K, 0K][TC: 9][DOE: 0K, 0][CCI: 473K, 49152K, 2496K]                    FullGC後のTenuredの値

 :

[VGC]<Fri Jul 19 17:57:23 2019>[Full GC 47030K->47026K(56376K), 0.0323550 secs][DefNew::Eden: 0K-
>0K(27008K)][DefNew::Survivor: 0K->0K(832K)][Tenured: 47030K->47026K(57344K)][Metaspace: 2828K(4100K,
4352K)->2828K(4100K, 4352K)][cause:ObjAllocFail][User: 0.0300000 secs][Sys: 0.0000000 secs][IM: 2413K,
2848K, 0K][TC: 9][DOE: 0K, 0][CCI: 473K, 49152K, 2496K]                    FullGC後のTenuredの値

 :

[VGC]<Fri Jul 19 17:58:12 2019>[Full GC 50585K->50566K(82944K), 0.0323550 secs][DefNew::Eden: 0K-
>0K(27008K)][DefNew::Survivor: 0K->0K(832K)][Tenured: 50566K->50566K(57344K)][Metaspace: 2828K(4100K,
4352K)->2828K(4100K, 4352K)][cause:ObjAllocFail][User: 0.0300000 secs][Sys: 0.0000000 secs][IM: 2413K,
2848K, 0K][TC: 9][DOE: 0K, 0][CCI: 473K, 49152K, 2496K]                    FullGC後のTenuredの値
```

FullGC 時に，Java ヒープ使用量（DefNew と Tenured）を確認するときの例です。

この例では，“Tenured”の FullGC 後の値が 15780K→31566K→47026K→50566K と増えていることが確認できます。

Metaspace 使用量の分析

Metaspace 使用量を分析した例を示します。



FullGC 時に，Metaspace 使用量（Metaspace の値）を確認するときの例です。

この例では，“Metaspace”の FullGC 後の値が 5599K→7372K→7886K→8286K と増えていることが確認できます。

C ヒープ使用量の分析

JavaVM を実行しているプロセスのメモリ使用量を分析します。

OS の稼働情報からメモリ使用量の増加状況を確認します。

(d)　スレッドダンプによるローカル変数情報の解析例

プログラムがローカル変数に設定する値を誤ったために，OutOfMemory が発生しているスレッドダンプの例を次の図に示します。

プログラム

```
■領域を確保する処理
93:     int newcount = count + len;
94:     if (newcount > buf.length) {
95:         byte newbuf[] = new byte[Math.max(buf.length << 1, newcount)];
96:         System.arraycopy(buf, 0, newbuf, 0, count);
97:         buf = newbuf;
98:     }
          :
```

ポイント1

スレッドダンプファイル

```
at java.io.ByteArrayOutputStream.write(ByteArrayOutputStream.java:95)
  locals:
    (java.io.ByteArrayOutputStream) this = <0x00002aaaabdf4000>
    (byte[]) b [arg1] = <0x00002aaaaf882a18>
    (int) off [arg2] = 0
    (int) len [arg3] = 8192
    (int) newcount = 268443242
        :
```

ポイント2

【解析のポイント】

スレッドダンプを取得して，領域を確保する処理に関与するローカル変数の値を確認します。

- ポイント 1

  プログラムで，領域を確保する処理に関与するローカル変数を確認します。

- ポイント 2

  スレッドダンプに出力されている，該当するローカル変数の値を確認します。

この例では，ローカル変数 newcount の値である 268443242（約 250MB）または，それ以上の値で byte 領域を確保しています。

(e)　設定変更による対処

設定を変更することで対処できる場合は，症状とシステムの状況に応じて対処をします。

表 3-5　症状と対処

症状	システムの状況	対処
Java ヒープ不足	• FullGC 直前の Java ヒープ使用量が Java ヒープの最大値に到達した。 • FullGC 後も Old 領域に空きがない。	• Java ヒープの最大値（-Xmx）を大きくする。
Metaspace 不足	• FullGC 後も Metaspace に空きがない。 • FullGC が頻発する。	• Metaspace の最大値（-XX:MaxMetaspaceSize）を大きくする。
C ヒープ不足	• C ヒープ使用量が使用可能なメモリ使用量に到達した。 • C ヒープ使用量がカーネルパラメタの上限値に達した。	• OS のシステムメモリを大きくする。 • 不要なプロセスを停止する。
	• 生成されたスレッド数が多く，スレッドスタックで使用するメモリ量（C ヒープ使用量）が上限値に達した。	• OS のシステムメモリを大きくする。

症状	システムの状況	対処
		• スレッドスタックサイズ（-Xss）を小さくする。

3.1.3　保守員調査依頼時の提供情報

保守員に調査依頼をする場合に，保守員に提供する情報を示します。

次の内容をできる限り正確に提供してください。

1. トラブルの現象とトラブルが発生した時期
2. トラブルが発生する直前に発生した操作

また，次の情報も提供してください。

1. 使用しているシステムの情報
   - OS とそのバージョン，メモリ，CPU（プロセッサ数）
   - 使用している JavaVM のバージョン・リビジョン（パッチ）
2. トラブル種別ごとの取得資料

表 3-6　トラブル種別ごとの取得資料一覧

項番	資料の種類	トラブル種別			
		プロセスダウン	プロセスハングアップ（無応答）	スローダウン	OutOfMemory（メモリ不足）
1	エラーリポートファイル	○	－	－	○※
2	core ダンプ／スタックトレースファイル	○	○※	○	○※
3	JavaVM のスレッドダンプファイル	－	○	○	○
4	JavaVM ログファイル	○	○	○	○
5	標準出力と標準エラー出力	○	○	○	○

（凡例）

○：取得します。

－：取得しません。

注※

発生個所によっては資料が取得できない場合があります。

3. アプリケーションサーバをお使いの場合，アプリケーションサーバが出力するトラブルシューティングの資料

3.2　システムで発生する代表的なトラブル

アプリケーションサーバシステムの運用では，ログインできない，応答が返らない，通常 1～2 秒で終了する処理が 10 秒以上掛かるなど，業務に支障をきたすような問題（トラブル）が発生することがあります。これらのトラブルを繰り返し発生させないようにするためには原因究明が必要となります。この原因究明の過程で，JavaVM が生成した資料の確認が必要となるケースがあります。ここでは，アプリケーションサーバで発生する代表的なトラブルを示し，そのトラブルに対して，JavaVM が生成した資料の確認が必要となるケースを説明します。なお，JavaVM が生成した資料とその確認方法の詳細については，「3.1　JavaVM でのトラブルシューティング」を参照してください。

アプリケーションサーバで発生する代表的なトラブルの定義を次の表に示します。

表 3-7　アプリケーションサーバで発生する代表的なトラブルの定義

項番	トラブルの種類	定義
1	OutOfMemory	（アプリケーションサーバの）プロセスのメモリが不足し，処理が継続できない現象。
2	プロセスダウン	アプリケーションサーバのプロセスが消滅してしまう現象。
3	トランザクションタイムアウト	アプリケーションサーバ上で管理されるトランザクションが開始されてから終了するまでの想定時間内に処理が終了しない現象。
4	プロセスハングアップ	アプリケーションサーバのプロセスは存在しているが，アプリケーションサーバの動作が停止しているように見える現象。
5	スローダウン	応答時間が通常より長くなっている現象。
6	リクエストタイムアウト	Web サーバが，アプリケーションサーバにリクエストを送信してから，応答を受け取るまでの想定時間内に処理が終了しない現象。

アプリケーションサーバで発生する代表的な各トラブルについて，JavaVM が生成した資料の確認が必要となるケースを，次の表に示します。資料の確認方法は，「3.1.2　トラブル事象別の原因切り分け方法」で説明しています。

表 3-8　JavaVM が生成した資料の確認が必要となるケース

項番	トラブルの種類	調査の進め方	JavaVM 生成資料の確認方法参照先
1	OutOfMemory	アプリケーションサーバのサーバ稼働ログで，「java.lang.OutOfMemoryError」のメッセージを確認した場合，JavaVM が生成した資料を取得して調査を進めます。	「3.1.2(4) OutOfMemoryError 障害」
2	プロセスダウン	アプリケーションサーバのプロセスダウンが発生した場合，JavaVM が生成した資料を取得して調査を進めます。	「3.1.2(1)　プロセスダウン」
3	トランザクションタイムアウト	トランザクションタイムアウトが発生したことは，アプリケーションサーバのサーバ稼働ログのメッセージから確認できます。その上で，アプリケーションサーバのサーバ稼	「3.1.2(3)　プロセススローダウン」

項番	トラブルの種類	調査の進め方	JavaVM 生成資料の確認方法参照先
		働ログをさらに詳細に確認して，タイムアウトの原因の可能性が高い個所を特定することで原因究明を進めていきます。この過程で，アプリケーションのサーバ稼働ログの出力不足などによって，どこの処理で時間が掛かっているのか特定できない場合は，現象を再現させた上で，JavaVM が生成した資料を取得することで問題個所を特定できる可能性があります。	
4	プロセスハングアップ	アプリケーションサーバのプロセスは存在しているが，アプリケーションサーバのサーバ稼働ログでアプリケーションが動作している様子が見られない場合，JavaVM が生成した資料を取得することで問題個所を特定できる可能性があります。	「3.1.2(2)　プロセスハングアップ（無応答）」
5	スローダウン	FullGC 実行時間の影響によって現象が発生している可能性があります。JavaVM が生成した資料を取得して確認します。	「3.1.2(3)　プロセススローダウン」
6	リクエストタイムアウト	トランザクションタイムアウトが発生しておらず，リクエストタイムアウトが発生する場合，パラメタ設定に問題がないか確認します。具体的には，Web サーバ側に設定するリクエストタイムアウトの設定値は，アプリケーションサーバ側に設定するトランザクションタイムアウトの設定値より十分大きな値を設定する必要があります。それでも問題が解決しない場合，インフラ設計を見直し，受け付けたリクエストがリクエストタイムアウト時間以内に処理できるように流量設計を行い，パラメタのチューニングを検討します。	－

（凡例）

－：該当なし

ここで説明した代表的な障害に当てはまらない場合，出力されたメッセージや収集したトラブルシューティングのための情報を確認して調査します。

4 日立 JavaVM の機能詳細

日立 JavaVM を使用すると，障害時の資料やチューニングで使用する情報を取得できます。この章では，日立 JavaVM が提供する機能について説明します。

4.1 日立 JavaVM の機能の概要

アプリケーションサーバのプロセスは，JavaVM 上で実行されます。

日立 JavaVM で提供している機能を次に示します。

- クラス別統計機能
  - インスタンス統計機能※1
  - STATIC メンバ統計機能※1
  - 参照関係情報出力機能※1
  - 統計前の GC 選択機能※1
  - Tenured 領域内不要オブジェクト統計機能※1
  - Tenured 増加要因の基点オブジェクトリスト出力機能※1
- クラス別統計情報解析機能※2
- Survivor 領域の年齢分布情報出力機能
- JIT コンパイル時の C ヒープ確保量の上限値設定機能
- スレッド数の上限値設定機能
- ファイナライズ滞留解消機能
- ログファイルの非同期出力機能
- 圧縮オブジェクトポインタ機能

注※1

クラス別統計情報を出力するクラス別統計機能の一つです。

注※2

クラス別統計情報解析機能を使うと，拡張スレッドダンプファイルに出力されたクラス別統計情報を CSV 形式で出力できます。

ヒント

クラス別統計機能の Tenured 領域内不要オブジェクト統計機能を実行した場合，次に示す機能は無効となります。

- インスタンス統計機能
- STATIC メンバ統計機能
- 統計前の GC 選択機能

日立 JavaVM では，障害発生時の要因分析やシステムの状態確認に利用できるよう，ログの出力内容が拡張されています。このログは，日立 JavaVM の JavaVM ログファイルに出力され，標準の JavaVM より

も，多くのトラブルシュート情報が取得できます。さらに，このログ（拡張 verbosegc 情報）を利用して適切なチューニングを実施することで，システムの可用性向上が図れます。日立 JavaVM の JavaVM ログファイルについては，「7.3　JavaVM ログ（JavaVM ログファイル）」を参照してください。日立 JavaVM のチューニングについては，「2.　メモリチューニング」を参照してください。

次節以降で，日立 JavaVM の各機能について説明します。

日立 JavaVM の機能を使用する際の注意事項については，「4.16　日立 JavaVM の機能使用時の注意事項」を参照してください。また，Oracle 社の JDK との非互換性については，「4.17　Oracle 社の JDK と日立が提供する JDK との非互換性」を参照してください。

4.2　クラス別統計機能

ここでは，クラス別統計機能について説明します。

クラス別統計機能を使用すると，クラスごとに，参照関係にあるクラスのインスタンスの情報を拡張スレッドダンプに出力できます。

4.2.1　クラス別統計機能の概要

各クラスのインスタンスが持つメンバの配下にあるすべてのインスタンスのサイズを，クラスごとに統計情報として拡張スレッドダンプに出力できます。この統計情報を**クラス別統計情報**といいます。クラス別統計情報を複数回出力すると，GC による Java オブジェクトの変化や，寿命が短い Java オブジェクトの状態，クラスごとのサイズの変化，Java オブジェクトの親子関係などを調査できます。これらの情報は，1 トランザクション当たりのメモリ使用量の測定や，メモリリークの検出などに利用できます。

クラス別統計情報は，次に示す機能によって出力されます。

- インスタンス統計機能
- STATIC メンバ統計機能
- 参照関係情報出力機能
- 統計前の GC 選択機能
- Tenured 領域内不要オブジェクト統計機能
- Tenured 増加要因の基点オブジェクトリスト出力機能

各機能については，「4.2.2　クラス別統計機能を前提とする機能」を参照してください。

また，クラス別統計機能では，Java ヒープ（Eden 領域，Survivor 領域および Tenured 領域を合わせたもの）にあるインスタンスを統計対象とします。クラス別統計情報の出力方法については，「4.2.3　クラス別統計情報の出力」を参照してください。

> **メモ**
>
> 拡張スレッドダンプは，デフォルトで出力されるように設定されています。拡張スレッドダンプを取得するための方法については，「3.1.1(3)　JavaVM のスレッドダンプファイル」を参照してください。出力情報については，「7.1　日立 JavaVM のスレッドダンプ」を参照してください。
>
> また，クラス別統計情報は，CSV 形式に出力することもできます。クラス別統計情報を CSV 形式で出力する方法については，「4.9　クラス別統計情報解析機能」を参照してください。

4.2.2　クラス別統計機能を前提とする機能

クラス別統計機能を前提とする機能の種類と概要を次の表に示します。

表 4-1　クラス別統計機能を前提とする機能の種類と概要

種類	概要	参照先
インスタンス統計機能	クラスごとに，クラスのインスタンスがメンバとして持つインスタンスの合計サイズを出力します。	4.3
STATIC メンバ統計機能	各クラスの static メンバが持つインスタンスの合計サイズを出力します。	4.4
参照関係情報出力機能	指定したクラス（インスタンス）をメンバに持つクラスの参照関係を出力します。	4.5
統計前の GC 選択機能	クラス別統計情報を出力する前に，GC を実行するかどうかを選択できます。この機能は，jheapprof コマンド実行時にオプションで指定します。デフォルトの設定では，クラス別統計情報を出力する前に FullGC が実行されます。	4.6
Tenured 領域内不要オブジェクト統計機能	Tenured 領域内で不要となったオブジェクトを特定します。	4.7
Tenured 増加要因の基点オブジェクトリスト出力機能	Tenured 領域内不要オブジェクト統計機能を使って特定した，不要オブジェクトの基点となるオブジェクトの情報を出力します。	4.8

これらの機能のうち，インスタンス統計機能，STATIC メンバ統計機能，参照関係情報出力機能，および Tenured 増加要因の基点オブジェクトリスト出力機能は，クラス別統計機能の実行時に有効になります。

統計前の GC 選択機能は，クラス別統計機能を実行する際に設定できます。クラス別統計情報の調査目的に合わせて選択してください。詳細は，「4.6.2　GC の選択の指針」を参照してください。

4.2.3　クラス別統計情報の出力

ここでは，クラス別統計情報を出力する方法について説明します。

クラス別統計情報を拡張スレッドダンプに出力するには，jheapprof コマンドを利用します。クラス別統計情報を出力したい Java プロセスや，参照関係情報を出力したいクラスを指定して，jheapprof コマンドを実行します。

jheapprof コマンド実行時には，次の指定ができます。

- クラス別統計情報を取得する前に GC を実行するかどうかの指定

次に，jheapprof コマンドの実行形式と実行例，および各指定方法について説明します。

(1) jheapprof コマンドの実行形式と実行例

jheapprof コマンドの実行形式と実行例を次に示します。jheapprof コマンドの詳細については，「jheapprof（クラス別統計情報付き拡張スレッドダンプの出力）」を参照してください。

実行形式

```
jheapprof [-f|-i] [-explicit|-noexplicit] [-class <クラス名>] [-fullgc|-copygc|-nogc] [-garbage|-nogarbage] [-rootobjectinfo|-norootobjectinfo] [-rootobjectinfost <サイズ>] [-force] -p <プロセスID>
```

注　-explicit, -noexplicit は使用できません。

実行例

ここでは，プロセス ID が 2463 の Java プロセスのクラス別統計情報を出力します。

1. -p オプションに，クラス別統計情報を出力したい Java プロセスのプロセス ID を指定して，jheapprof コマンドを実行します。

```
% jheapprof -p 2463
```

jheapprof コマンドで-f オプションを省略している場合，次の確認メッセージが表示されます。

プロセス ID を確認するメッセージが次の形式で表示されます。

```
send SIGQUIT to 2463: ? (y/n)
```

2. y を入力します。

クラス別統計情報付き拡張スレッドダンプが出力されます。実行中の Java プログラムでは次のメッセージが出力されます。

```
Writing Java core to javacore2463.030806215140.txt... OK
```

実行中の Java プログラムは，カレントディレクトリにクラス別統計情報付き拡張スレッドダンプ（javacore<プロセス ID>.<日時>.txt）を作成し，プログラムを継続します。

(2) GC の実行有無を指定する場合

クラス別統計情報を出力する前に，GC を実行するかどうかを選択できます。この機能を統計前の GC 選択機能といいます。クラス別統計情報を出力する前に，GC を実行するかどうかは，jheapprof コマンドに次のどれかのオプションを指定して実行します。

- -fullgc
  FullGC を実行してから，クラス別統計情報を出力します。
- -copygc
  CopyGC を実行してから，クラス別統計情報を出力します。
- -nogc
  GC を実行しないで，クラス別統計情報を出力します。

統計前の GC 選択機能については，「4.6　統計前の GC 選択機能」を参照してください。なお，Tenured 領域内不要オブジェクト統計機能を実行する場合，統計前の GC 選択機能は実行できません。

4.2.4　クラス別統計情報出力時の注意事項

クラス別統計情報出力時の注意事項について説明します。

- 起動中の Java プロセスに対して，-copygc オプションを設定した jheapprof コマンドを実行すると，統計前に CopyGC を実行しようとします。この場合に，Tenured 領域の空き容量が足りないと，CopyGC が実行できないときがあります。

  CopyGC が実行できない場合，JavaVM 起動オプションに-XX:+HitachiVerboseGC を指定していても，GC が発生したときの拡張 verbosegc 情報は出力されません。なお，クラス別統計情報を含む拡張スレッドダンプは，GC 実行時と同様に出力されます。
- JavaVM 起動時に，-XX:+PrintGCDetails が指定された Java プロセスに対して，-copygc オプションを設定した jheapprof コマンドを実行すると，統計前に CopyGC を実行します。この場合，-XX:+PrintGCDetails の指定によって出力される GC 情報には，「Full GC」と出力されます。

4.3　インスタンス統計機能

ここでは，インスタンス統計機能について説明します。

インスタンス統計機能は，クラス別統計情報を出力する機能の一つです。クラスごとに，クラスのインスタンスの数やインスタンスの合計サイズが出力できます。

インスタンス統計機能は jheapprof コマンドを使用して出力します。jheapprof コマンドの実行形式と実行例については，「4.2.3　クラス別統計情報の出力」を参照してください。

4.3.1　インスタンス統計機能の概要

インスタンス統計機能は，アプリケーションのメモリリークの調査で使用します。

インスタンス統計機能では，classA のインスタンス→classA のメンバ変数（クラスは classB）→classB のインスタンス→...とインスタンスの参照関係を調べ，ほかのインスタンスへの参照を持たないインスタンスのサイズをそのインスタンスをメンバに持つクラスに再帰的に加算します。つまり，インスタンス統計機能では，各クラスのインスタンスが参照しているインスタンスの合計サイズが出力されます。

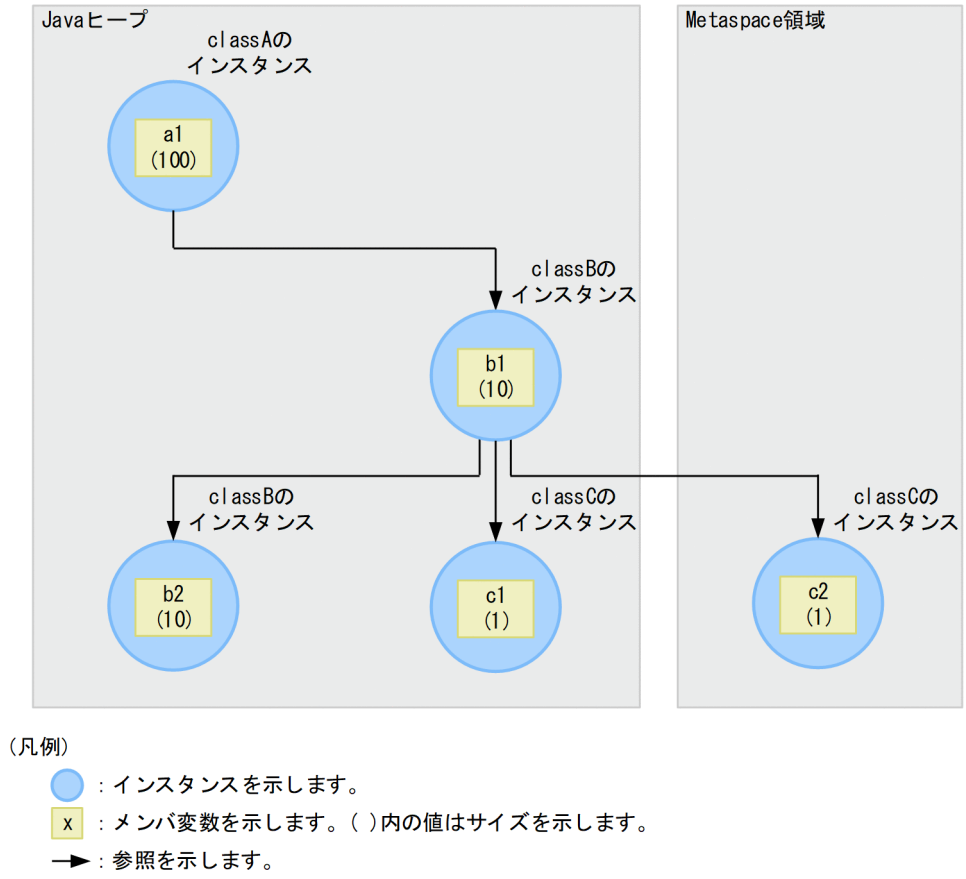
メモリリークの原因を調査する場合は，次のように，メモリリークを調査したいアプリケーションの処理の前後でインスタンス統計機能を実行し，1.と 3.のインスタンス数やインスタンスの合計サイズの差分を取り，その増加量を見てメモリリークの原因となっているクラスを特定します。

1. インスタンス統計機能を実行します。
2. メモリリークを調査したいアプリケーションの処理を実行します。
3. インスタンス統計機能を実行します。

なお，インスタンス統計機能は，各クラスのインスタンスが参照しているすべてのインスタンスのサイズを再帰的に加算するため，各クラスのインスタンスだけのサイズ（参照しているインスタンスのサイズを含まないサイズ）を調べることはできません。

統計対象ごとの出力結果について，次の図に示すインスタンス構造を例に説明します。

図 4-1　インスタンス構造の例



インスタンス a1，b1，b2，および c1 がクラス別統計情報の対象となります。

統計対象ごとのインスタンス数とインスタンスの合計サイズを次の表に示します。

表 4-2　統計対象ごとのインスタンス数とインスタンスの合計サイズ

統計対象	クラス A		クラス B		クラス C	
	インスタンス数	合計サイズ※	インスタンス数	合計サイズ※	インスタンス数※	合計サイズ※
Java ヒープ	1	121	2	21	1	1

注※　合計サイズは，インスタンスの合計サイズを示します。単位はバイトです。

各クラスのインスタンスの合計サイズの算出式を次に示します。

- クラス A の場合：a1+b1+b2+c1
- クラス B の場合：b1+b2+c1
- クラス C の場合：c1

インスタンス統計機能では，基点となるオブジェクトから次の順序に従って参照されるオブジェクトの参照関係を調べます。基点となるオブジェクトは，ほかの参照関係で調べられていないオブジェクトが該当します。1.，2.は調査の優先順を示します。

1. Java ヒープ内のアドレスの低い順
2. Metaspace 領域内のアドレスの低い順

参照先のオブジェクトが調査済みの場合は，分岐点まで戻って参照関係を調べます。

また，参照先のオブジェクトがほかの参照関係の基点となるオブジェクトである場合は，参照先オブジェクトとして扱います。すべての基点となるオブジェクトがなくなるまで参照関係を調べます。

インスタンス統計機能の場合，インスタンス数には各クラスのインスタンス数が出力されます。インスタンスの合計サイズには，次の内容が出力されます。

- 基点となるオブジェクトのサイズは，該当するクラスに加算されます。参照先のオブジェクトのサイズは，該当するクラスに加算され，さらに基点となるオブジェクト，および該当するクラスまでの参照関係にあるすべてのオブジェクトの該当するクラスにも加算されます。

4.3.2　インスタンス統計機能で出力するクラス別統計情報

インスタンス統計機能で出力するクラス別統計情報の出力形式，出力項目および出力例について説明します。

(1)　出力形式と出力項目

インスタンス統計機能で出力するクラス別統計情報の出力形式を次に示します。

- 出力形式

```
Java Heap Profile
-----------------------
_________________Size__Instances__Class_________________
   <total_size>       <Instance_count>   <class_name>
   <total_size>       <Instance_count>   <class_name>
...
```

- 出力項目

  出力形式で示した各項目について説明します。

  表 4-3　出力項目（インスタンス統計機能）

出力項目	意味
<total_size>	インスタンスの合計サイズがバイト単位で出力されます。
<Instance_count>	インスタンスの数が出力されます。
<class_name>	クラス名が出力されます。

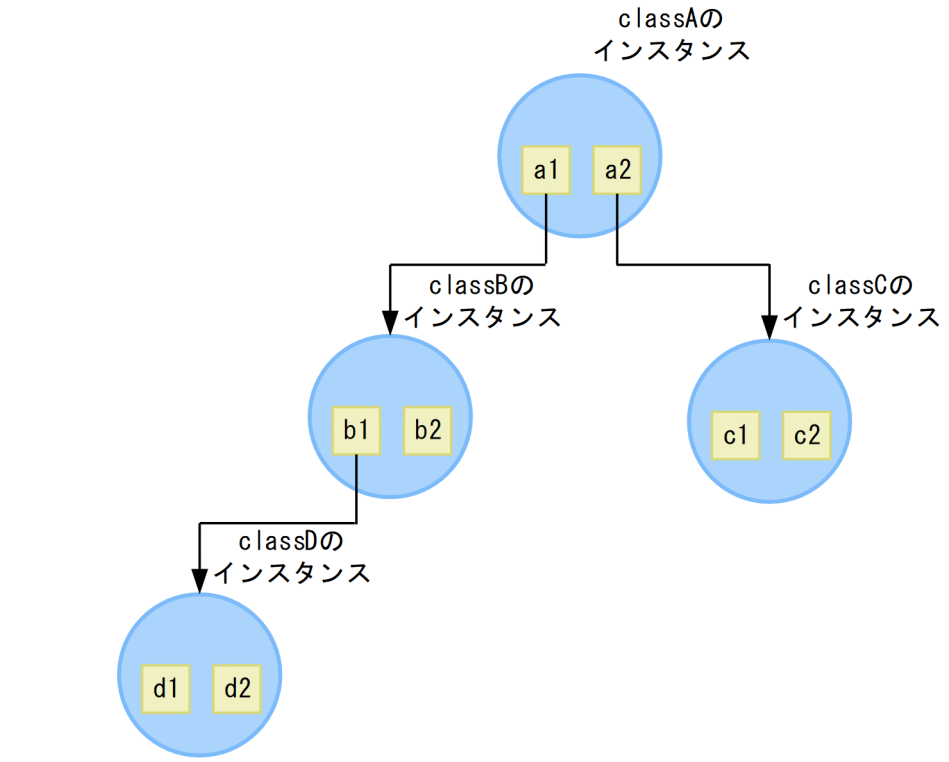
(2)　出力例

インスタンス統計機能で出力するクラス別統計情報の出力例を，次のソースを例にして説明します。

```
public class instance {
  public static void main(String args[]) {
    classA cls_a = new classA();
    try {
      Thread.sleep(20000);
    } catch (Exception e) {}
  }
}
class classA {
  classB a1;
  classC a2;
  classA() {
    a1 = new classB();
    a2 = new classC();
  }
}
class classB {
  classD b1;
  String b2;
  classB() {
    b1 = new classD();
    b2 = null;
  }
}
class classC {
  String c1, c2;
  classC() {
    c1 = null;
    c2 = null;
  }
}
class classD {
  String d1, d2;
  classD() {
    d1 = null;
    d2 = null;
  }
}
```

このソースの場合の，インスタンス構造を次の図に示します。

図 4-2　インスタンス構造（インスタンス統計機能）



（凡例）

：インスタンスを示します。

x ：メンバ変数を示します。

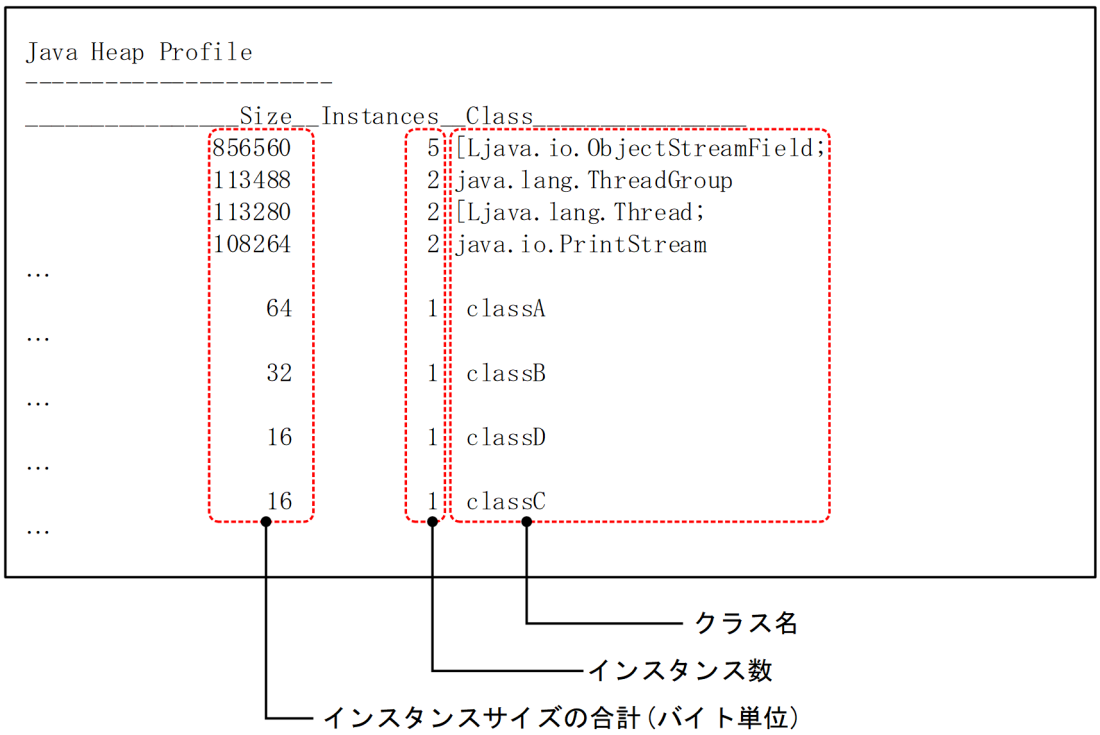
→：参照を示します。

このようなインスタンス構造の場合，インスタンス統計機能では，次のように各クラスのサイズを加算します。

- classA のサイズ：a1+a2+ b1+b2+ c1+c2+ d1+d2
- classB のサイズ：b1+b2+ d1+d2
- classC のサイズ：c1+c2
- classD のサイズ：d1+d2

インスタンス統計機能の出力結果を次の図に示します。

図 4-3　出力結果（インスタンス統計機能）



4.4　STATIC メンバ統計機能

ここでは，STATIC メンバ統計機能について説明します。

STATIC メンバ統計機能は，クラス別統計情報を出力する機能の一つです。クラスごとに，static メンバが持つインスタンスの合計サイズが出力できます。

STATIC メンバ統計機能は jheapprof コマンドを使用して出力します。jheapprof コマンドの実行形式と実行例については，「4.2.3　クラス別統計情報の出力」を参照してください。

4.4.1　STATIC メンバ統計機能の概要

STATIC メンバ統計機能は，インスタンス統計機能と同様に，アプリケーションのメモリリークの調査で使用します。

インスタンス統計機能と異なる点は，インスタンス統計機能が最初に取り出したインスタンスの非 static フィールドから参照しているインスタンスのサイズを再帰的に加算するのに対し，STATIC メンバ統計機能は最初に取り出したインスタンスの static フィールド（クラスの static フィールド）から参照しているインスタンスのサイズを再帰的に加算するという点です。これによって，各クラスの static メンバが持つインスタンスの合計サイズを得ることができます。ただし，最初に取り出すインスタンス以外では，インスタンス統計機能および STATIC メンバ統計機能共に，インスタンスの非 static メンバを基に参照関係を調べます。

インスタンス統計機能と STATIC メンバ統計機能の違いについては，「4.4.2(2)　出力例」を参照してください。

メモリリークの原因を調査する場合は，次のように，メモリリークを調査したいアプリケーションの処理の前後で STATIC メンバ統計機能を実行し，1.と 3.のインスタンス数やインスタンスの合計サイズの差分を取り，その増加量を見てメモリリークの原因となっているクラスを特定します。

1. STATIC メンバ統計機能を実行します。
2. メモリリークを調査したいアプリケーションの処理を実行します。
3. STATIC メンバ統計機能を実行します。

なお，STATIC メンバ統計機能は，各クラスの static フィールドが参照しているインスタンスのサイズを再帰的に加算するため，各クラスのインスタンスだけのサイズ（参照しているインスタンスのサイズを含まないサイズ）の合計を調べることはできません。

STATIC メンバ統計機能では，基点となるオブジェクトから参照関係を調べます。基点となるオブジェクトは，JavaVM が持つすべてのクラスの STATIC メンバが参照し，ほかの参照関係で調べられていないオブジェクトが該当します。

また，参照先のオブジェクトが調査済みの場合は，分岐点まで戻って参照関係を調べます。すべての基点となるオブジェクトがなくなるまで参照関係を調べます。

STATIC メンバ統計機能の場合，インスタンス数とインスタンスの合計サイズには，次の内容が出力されます。

- 参照関係にあるすべてのオブジェクトのサイズとオブジェクト数を基点となるオブジェクトに合計します。この値を基点となるオブジェクトが参照している STATIC メンバを持つクラスに合計して統計値とします。

4.4.2 STATIC メンバ統計機能で出力するクラス別統計情報

STATIC メンバ統計機能で出力するクラス別統計情報の出力形式，出力項目および出力例について説明します。

(1) 出力形式と出力項目

STATIC メンバ統計機能で出力するクラス別統計情報の出力形式を次に示します。

- 出力形式

```
Java Heap Dump Static Profile
----------------------
_________________Size__Instances__Class_________________
   <total_size>       <Instance_count>   <class_name>
   <total_size>       <Instance_count>   <class_name>
...
```

- 出力項目

  出力形式で示した各項目について説明します。

表 4-4 出力項目（STATIC メンバ統計機能）

出力項目	意味
<total_size>	インスタンスの合計サイズがバイト単位で出力されます。
<Instance_count>	インスタンスの数が出力されます。
<class_name>	クラス名が出力されます。

(2) 出力例

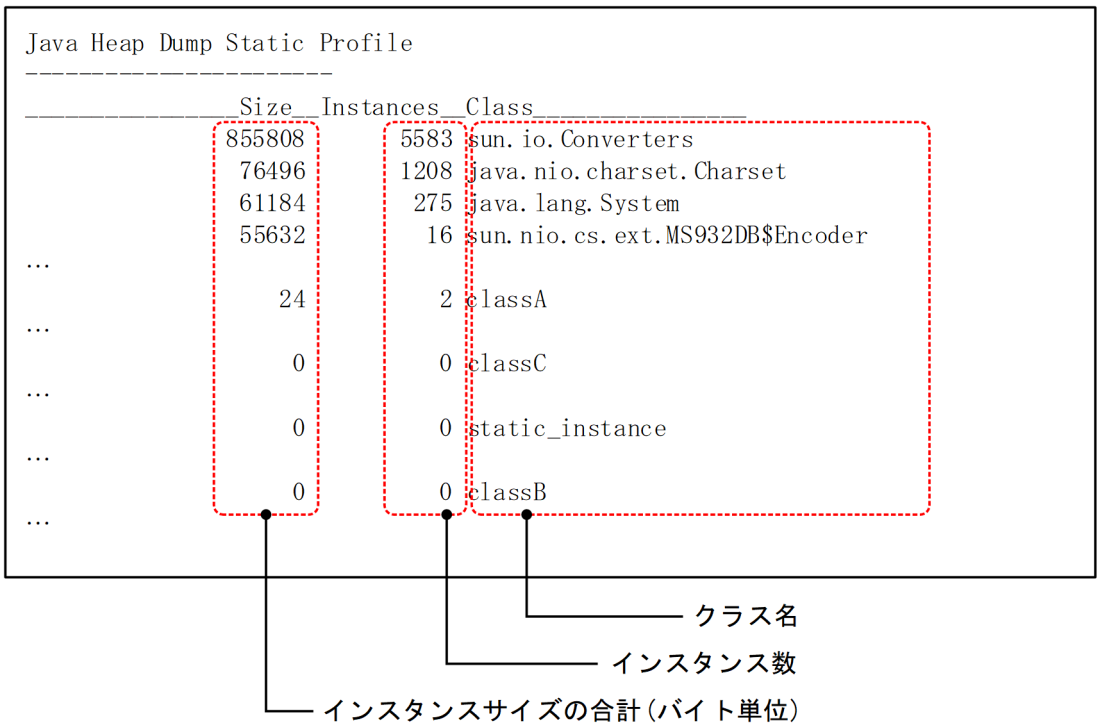
STATIC メンバ統計機能で出力するクラス別統計情報の出力例を，次のソースを例にして説明します。

```
public class static_instance {
  public static void main(String args[]) {
    classA cls_a;
    classB cls_b;
```

```
    classC cls_c;

    cls_a = new classA();
    cls_b = new classB();
    cls_c = new classC();
    cls_b.cls_c = cls_c;
    cls_a.cls_b = cls_b;

    try {
      Thread.sleep(20000);
    } catch (Exception e) {}
  }
}
class classA {
  static classB cls_b;
}
class classB {
  classC cls_c;
}
class classC {
}
```

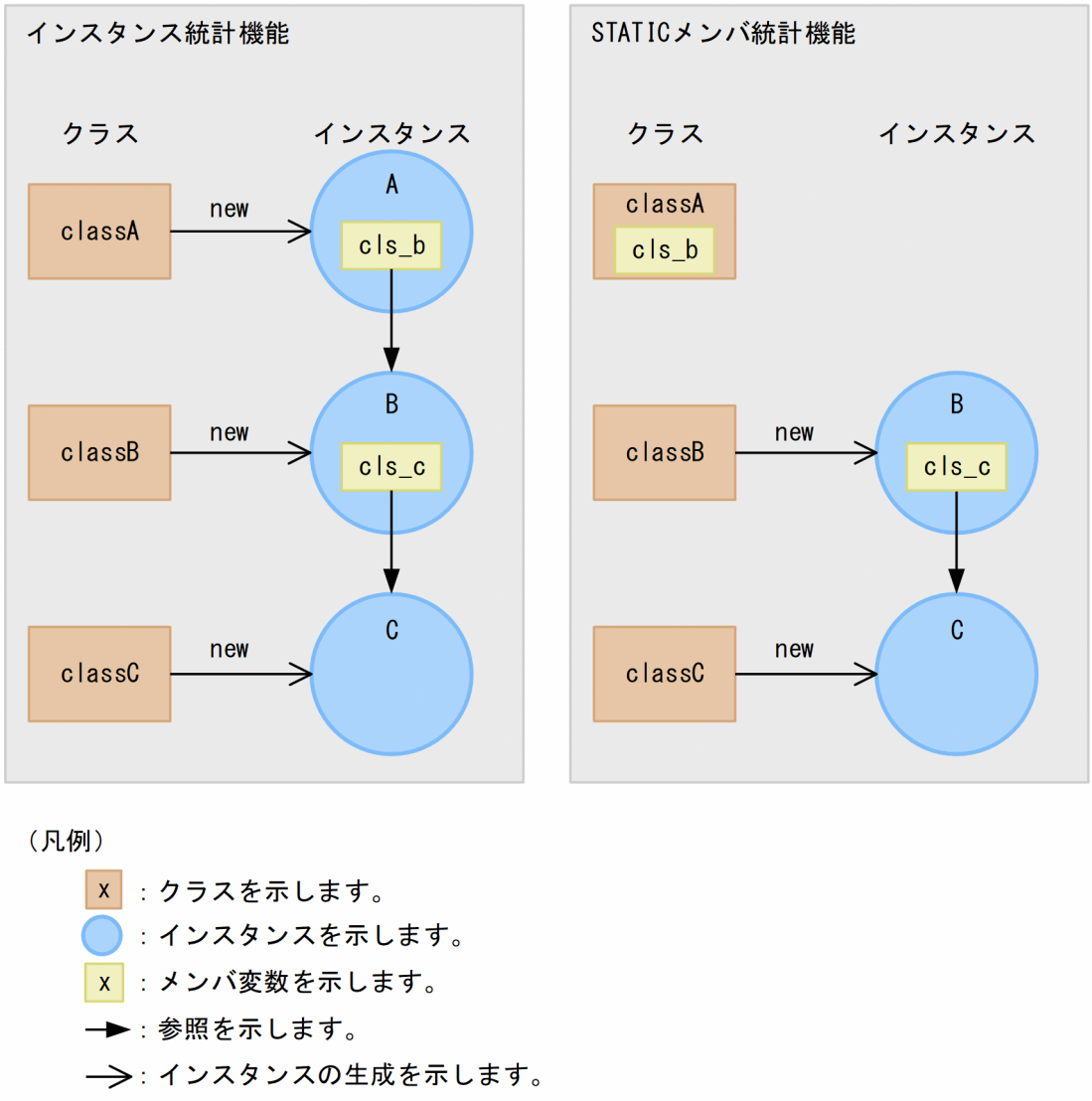
STATIC メンバ統計機能の出力結果を次の図に示します。

図 4-4　出力結果（STATIC メンバ統計機能）



また，上記のソースの場合，インスタンス統計機能と STATIC メンバ統計機能では，参照関係に違いがあります。インスタンス統計機能と STATIC メンバ統計機能の参照関係の相違を次の図に示します。

図 4-5　インスタンス統計機能と STATIC メンバ統計機能の参照関係の相違



それぞれ機能の参照関係は次のようになっています。

- インスタンス統計機能の参照関係

  インスタンス A のインスタンス変数 cls_b → インスタンス B のインスタンス変数 cls_c → インスタンス C
- STATIC メンバ統計機能の参照関係

  クラス A のクラス変数 cls_b → インスタンス B のインスタンス変数 cls_c → インスタンス C

4.5　参照関係情報出力機能

ここでは，参照関係情報出力機能について説明します。

参照関係情報出力機能は，指定したクラスのインスタンスの参照関係が先頭から順番に出力できます。

4.5.1　参照関係情報出力機能の概要

jheapprof コマンドの-class オプションに指定したクラスのインスタンスがどのクラスから参照されているかを，インスタンスの参照関係の先頭から順番に出力します。

指定したクラスのインスタンスが複数ある場合は，該当するすべてのインスタンスが出力されます。同じ名前のインスタンスが複数ある場合でも，インスタンス名のあとに次の情報が出力されるため，別のインスタンスかどうか識別できます。

- インスタンスのアドレス
- インスタンスが所属している領域名称

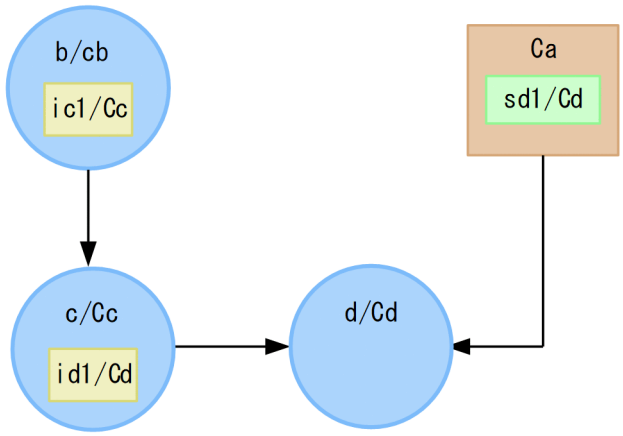
参照関係情報出力機能では，基点となるオブジェクトから次の順序に従って参照されるオブジェクトの参照関係を調べます。基点となるオブジェクトは，ほかの参照関係で調べられていないオブジェクトが該当します。

1. Java ヒープ内のアドレスの低い順

参照先のオブジェクトが-class オプションで指定したクラスの場合，参照関係情報には，基点となるオブジェクトから-class オプションで指定したクラスのオブジェクトまでの参照関係が出力されます。参照先のオブジェクトが調査済みの場合は，分岐点まで戻って参照関係を調べます。参照先のオブジェクトがほかの参照関係の基点となるオブジェクトである場合は，参照先オブジェクトとして扱います。すべての基点となるオブジェクトがなくなるまで参照関係を調べます。

また，jheapprof コマンドで-staticroot オプションを指定すると，static フィールドを基点とした参照関係情報出力機能が有効になります。この機能は，参照関係情報出力機能を前提としています。static フィールドを基点とした参照関係情報出力機能では，参照関係情報出力機能が出力する参照関係情報に，static フィールドを基点とした参照関係情報を追加して出力します。この出力情報は，static フィールドを基点とした参照関係によるメモリリークの原因を究明するために使用します。

メモリリーク状態の参照関係の例を次に示します。



（凡例）

：クラスを示します。

x ：<staticフィールド名>/<クラス名>を示します。

x ：インスタンス（<インスタンス名/<クラス名>）を示します。

x ：<インスタンスフィールド名>/<クラス名>を示します。

→：参照を示します。

図の参照関係のインスタンス「d/Cd」に static フィールドを基点とした参照関係情報出力機能を実行すると，次の参照関係情報が出力されます。

```
Reference of class Cd
---------------------
Cb(0x088065f0)[Tenured]
  Cc(0x08806668)[Tenured]
    Cd(0x088066e0)[Tenured]
------

Reference of class Cd from static field
---------------------------------------
static field Ca.sd1
  Cd(0x088066e0)[Tenured]
------
```

参照関係情報出力機能で出力する参照関係情報

staticフィールドを基点とした参照関係情報出力機能で出力する参照関係情報

この情報を基に，リーク対策として，次のフィールドの参照をクリアします。このことで，「d/Cd」は GC で回収されるため，メモリリークを解消することができます。

- インスタンス「c/Cc」のインスタンスフィールド「id1」
- クラス「Ca」の static フィールド「sd1」

参照関係情報出力機能の参照関係情報については，「4.5.2　参照関係情報出力機能で出力するクラス別統計情報」を，static フィールドを基点とした参照関係情報については，「4.5.3　static フィールドを基点とした参照関係情報出力機能で出力するクラス別統計情報」を参照してください。

4.5.2　参照関係情報出力機能で出力するクラス別統計情報

参照関係情報出力機能で出力するクラス別統計情報の出力形式，出力項目および出力例について説明します。

(1)　出力形式と出力項目

参照関係情報出力機能で出力するクラス別統計情報の出力形式を次に示します。

- 出力形式

```
Reference of class <オプション指定クラス名>
-----------------------------------------※
<クラス名>(<アドレス>)[<領域名称>]
  <クラス名>(<アドレス>)[<領域名称>]
    <オプション指定クラス名>(<アドレス>)[<領域名称>]
------
<クラス名>(<アドレス>)[<領域名称>]
  java.lang.ref.Finalizer<<繰り返し数> times>
    <クラス名>(<アドレス>)[<領域名称>]
      <クラス名>(<アドレス>)[<領域名称>]
        <オプション指定クラス名>(<アドレス>)[<領域名称>]
------
...
```

注※

<オプション指定クラス名>の文字数に 19 を加算した数の「-（ハイフン）」が出力されます。

- 出力項目

出力形式で示した各項目について説明します。

表 4-5　出力項目（参照関係情報出力機能）

出力項目	意味
<クラス名>	jheapprof コマンドの-class オプションに指定したクラスのインスタンスが参照するクラスの名称が出力されます。
<アドレス>	インスタンスのアドレスが出力されます。
<領域名称>	インスタンスが所属する領域が出力されます。 • Eden：Eden 領域を示します。 • Survivor：Survivor 領域を示します。 • Tenured：Tenured 領域を示します。
<オプション指定クラス名>	jheapprof コマンドの-class オプションに指定したクラス名が出力されます。
java.lang.ref.Finalizer	finalize()メソッドを持つクラスで生成する java.lang.ref.Finalizer のオブジェクトを 1 つにまとめて出力することを示します。
<繰り返し数>	Finalizer インスタンスの参照が連続した回数が出力されます。

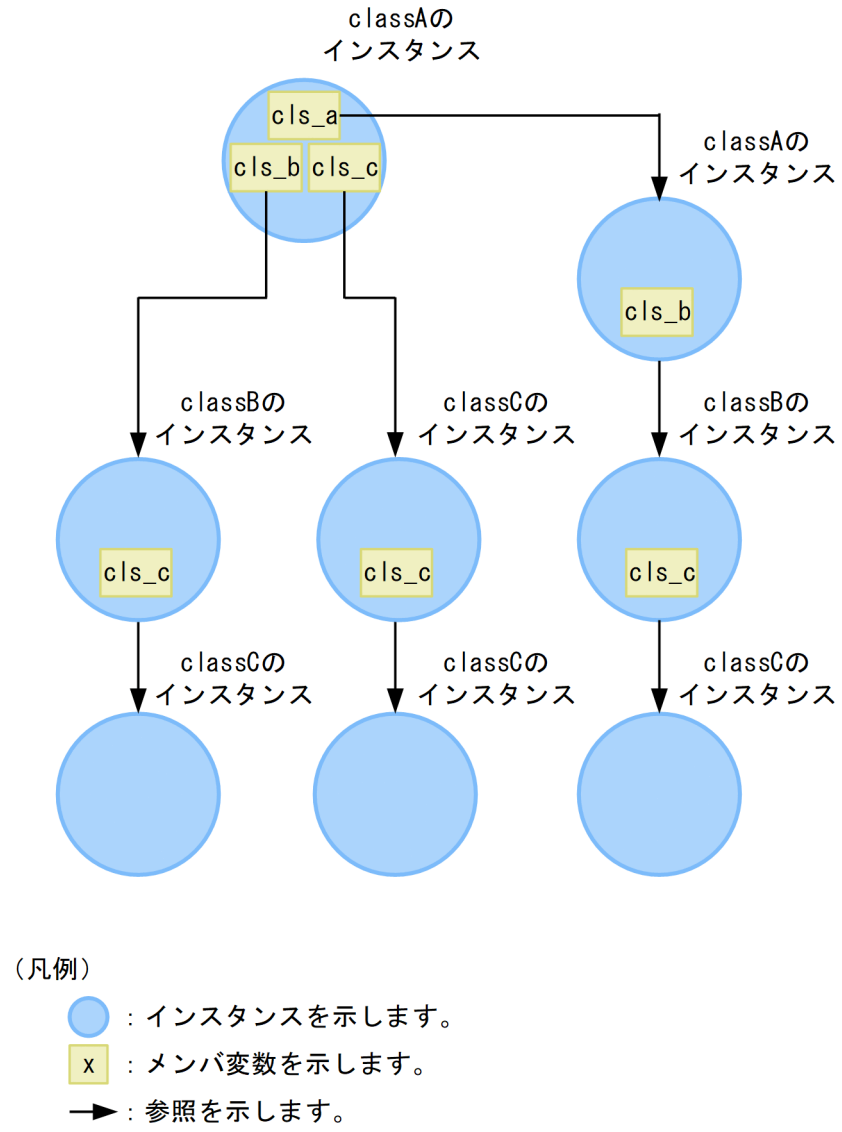
(2) 出力例

参照関係情報出力機能で出力するクラス別統計情報の出力例を，次のソースを例にして説明します。

```
public class instance2 {
  public static void main(String args[]) {
    classA cls_a1 = new classA();    classA cls_a2 = new classA();
    classB cls_b1 = new classB();    classB cls_b2 = new classB();
    classC cls_c1 = new classC();    classC cls_c2 = new classC();
    classC cls_c3 = new classC();    classC cls_c4 = new classC();
    cls_a1.cls_a = cls_a2;    cls_a1.cls_b = cls_b1;
    cls_a1.cls_c = cls_c1;    cls_a2.cls_b = cls_b2;
    cls_b1.cls_c = cls_c2;    cls_b2.cls_c = cls_c3;
    cls_c1.cls_c = cls_c4;
    try {
      Thread.sleep(20000);
    } catch (Exception e) {}
  }
}
class classA {
  classA cls_a;
  classB cls_b;
  classC cls_c;

  classA() {
    classB cls_b;
  }
}
class classB {
  classC cls_c;
}
class classC {
  classC cls_c;
}
```

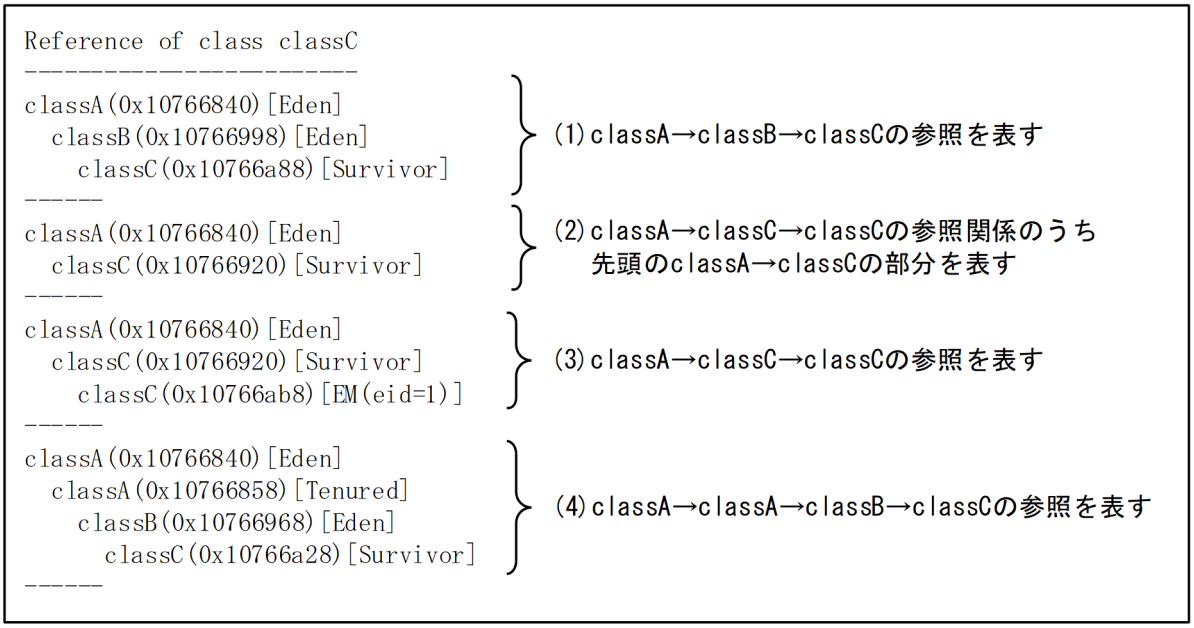
インスタンス構造を次の図に示します。

図 4-6　インスタンス構造（参照関係情報出力機能）



参照関係情報出力機能の出力結果を次の図に示します。この場合，jheapprof コマンドに引数「-class <クラス名>」を指定して実行します。

図 4-7　出力結果（参照関係情報出力機能）



注1　(1)～(4)の出力順はインスタンスの参照関係を調べるときの状態で処理順が変化する場合があります。
注2　(xxxxxxxxxx)のxxxxxxxxxxは，インスタンスのメモリ上のアドレスを示します。
また，[yy...yy]のyy...yyは，インスタンスが所属する領域を示します。

classA のアドレスは，すべて同じアドレス（0x10766840）になっています。したがって，classA はすべて同じインスタンスであることが分かります。一方，(1)と(4)の classB は，アドレスが異なっているので，別のインスタンスであることが分かります。

なお，GC の発生によってインスタンスのメモリ上の配置が変化します。そのため，アドレスおよび領域名称は，出力するたびに変化する場合があります。

4.5.3　static フィールドを基点とした参照関係情報出力機能で出力するクラス別統計情報

static フィールドを基点とした参照関係情報出力機能で出力するクラス別統計情報の出力形式，出力項目および出力例について説明します。

(1)　出力形式と出力項目

- 出力形式

  static フィールドを基点とした参照関係情報出力機能で出力するクラス別統計情報の出力形式を次に示します。

```
Reference of class <オプション指定クラス名> from static field
------------------------------------------------------------※1
static field <staticフィールド宣言クラス名>※2.<staticフィールド名>※2
<クラス名>(<アドレス>)[<領域名称>]
```

```
  <クラス名>(<アドレス>)[<領域名称>]
    <オプション指定クラス名>(<アドレス>)[<領域名称>]
------
...
```

注※1

<オプション指定クラス名>の文字数に 37 を加算した数の「-（ハイフン）」が出力されます。

注※2

参照関係の基点を示します。

- 出力項目

出力形式で示した各項目について説明します。

表 4-6　出力項目（static フィールドを基点とした参照関係情報出力機能）

出力項目	意味
<static フィールド宣言クラス名>	基点となる static フィールドを宣言しているクラス名が出力されます。
<static フィールド名>	基点となる static フィールドの名称が出力されます。
<クラス名>	jheapprof コマンドの-class オプションに指定したクラスのインスタンスを参照する，インスタンスのクラスの名称が出力されます。
<アドレス>	インスタンスのアドレスが出力されます。
<領域名称>	インスタンスが所属する領域が出力されます。 • Eden：Eden 領域を示します。 • Survivor：Survivor 領域を示します。 • Tenured：Tenured 領域を示します。
<オプション指定クラス名>	jheapprof コマンドの-class オプションに指定したクラス名が出力されます。

(2)　出力例

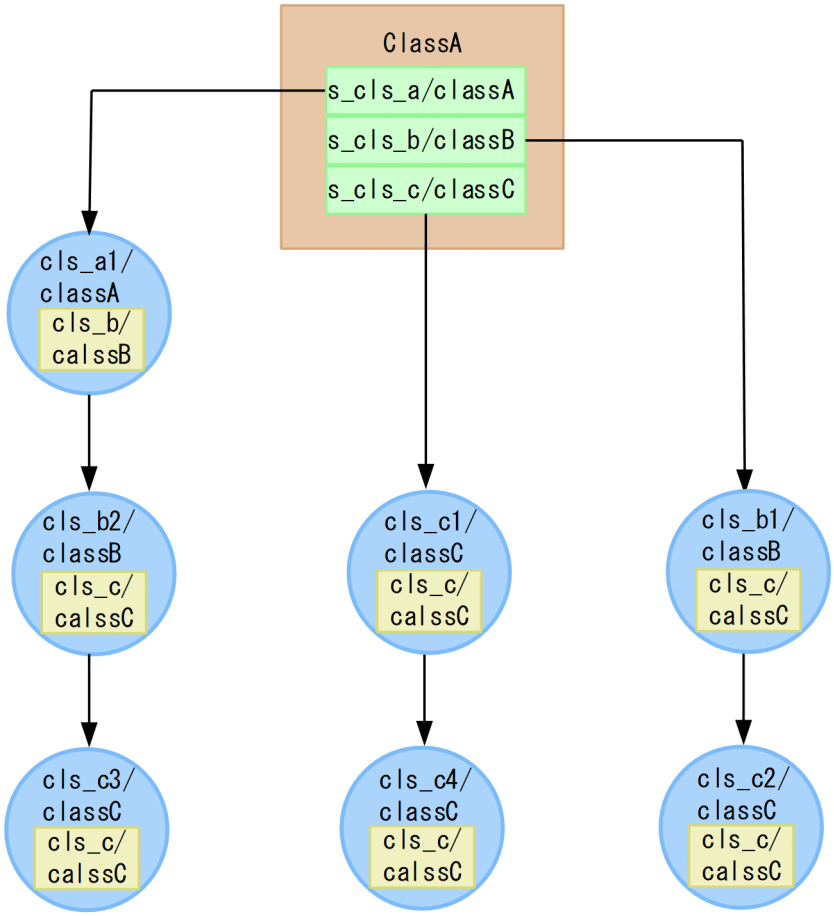
static フィールドを基点とした参照関係情報出力機能で出力するクラス別統計情報の出力例を，次のソースを例にして説明します。

```
import JP.co.Hitachi.soft.jvm.MemoryArea.*;
public class static_reference {
  public static void main(String args[]) {
   try {
      classA cls_a1 = new classA();
      classB cls_b1 = new classB();
      classB cls_b2 = new classB();
      classC cls_c1 = new classC();
      classC cls_c2 = new classC();
      classC cls_c3 = new classC();
      BasicExplicitMemory emem = new BasicExplicitMemory();
      classC cls_c4 = (classC)emem.newInstance(classC.class);
      cls_a1.s_cls_a = cls_a1;
      cls_a1.s_cls_b = cls_b1;
      cls_a1.s_cls_c = cls_c1;
```

```
      cls_a1.cls_b = cls_b2;
      cls_b1.cls_c = cls_c2;
      cls_b2.cls_c = cls_c3;
      cls_c1.cls_c = cls_c4;
      Thread.sleep(20000);
    } catch (Exception e) {e.printStackTrace();}
  }
}
class classA {
  static classA s_cls_a;
  static classB s_cls_b;
  static classC s_cls_c;
  classB cls_b;
}
class classB {
  classC cls_c;
}
class classC {
  classC cls_c;
  public classC(){
  }
}
```

インスタンス構造を次の図に示します。

図 4-8　インスタンス構造（static フィールドを基点とした参照関係情報出力機能）



（凡例）

：クラスを示します。

x ：<staticフィールド名>/<クラス名>を示します。

x ：インスタンス（<インスタンス名/<クラス名>）を示します。

x ：<インスタンスフィールド名>/<クラス名>を示します。

→：参照を示します。

static フィールドを基点とした参照関係情報出力機能の出力結果を次の図に示します。この場合，jheapprof コマンドに引数「-class <クラス名> -staticroot」を指定して実行します。

図 4-9　出力結果（static フィールドを基点とした参照関係情報出力機能）

```
Reference of class classC from static field
-------------------------------------
static field classA.s_cls_a
  classA(0x10511d08)[Tenured]
    classB(0x10511d78)[Eden]
      classC(0x10511df0)[Survivor]
------
static field classA.s_cls_b
  classB(0x10511d68)[Tenured]
    classC(0x10511de0)[Tenured]
------
static field classA.s_cls_c
  classC(0x10511dd0)[Tenured]
    classC(0x03bc0000)[EM(eid=1)]
------
static field classA.s_cls_c
  classC(0x10511dd0)[Tenured]
------
```

(1)classAのstaticフィールドs_cls_a
→classA→classB→classCの参照を表す

(2)classAのstaticフィールドs_cls_b
→classB→classCの参照を表す

(3)classAのstaticフィールドs_cls_c
→classC→classCの参照を表す

(4)classAのstaticフィールドs_cls_c
→classC→classCの参照関係のうち，先頭のclassAのstaticフィールドs_cls_c→classCの部分を表す

注1　(1)～(4)の出力順はインスタンスの参照関係を調べるときの状態で処理順が変化する場合があります。

注2　(xxxxxxxxxx)のxxxxxxxxxxは，インスタンスのメモリ上のアドレスを示します。
また，[yy...yy]のyy...yyは，インスタンスが所属する領域を示します。

4.5.4　static フィールドを基点とした参照関係情報出力時の注意事項

jheapprof コマンドの実行時間は，static フィールドを基点とした参照関係情報出力機能を有効にすると，static フィールドを基点とした参照関係情報出力機能が無効な場合に比べて，参照関係情報出力機能の実行時間分長くなります。

4.6　統計前の GC 選択機能

ここでは，統計前の GC 選択機能について説明します。

統計前の GC 選択機能を使用すると，クラス別統計情報の出力前に実行する処理を選択できます。

4.6.1　統計前の GC 選択機能の概要

クラス別統計機能を実行すると，拡張スレッドダンプへクラス別統計情報が出力できます。統計前の GC 選択機能では，クラス別統計情報を出力する前に実施する処理を選択できます。調査目的に合わせて，実施する処理を選択することで，Java オブジェクトのさまざまな変化の様子をクラス別統計情報に取得できます。

統計前の GC 選択機能を使用する場合，jheapprof コマンドの引数で実行する処理を指定します。クラス別統計機能実行前に実施できる処理と jheapprof コマンドの引数を次の表に示します。

表 4–7　クラス別統計機能実行前に実施できる処理と jheapprof コマンドの引数

処理の種類	処理内容	jheapprof コマンドの引数
FullGC の実行	Tenured 領域も含む，JavaVM 固有領域全体を対象に，使用済みのオブジェクトを回収します。	-fullgc
CopyGC の実行	Eden 領域および Survivor 領域だけを対象に，使用済みのオブジェクトを回収します。	-copygc
GC を実行しない	使用済みのオブジェクトがあっても回収しません。	-nogc

なお，JavaVM 起動オプションで-XX:+HitachiVerboseGC および-XX:+HitachiVerboseGCPrintCause を指定している場合にクラス別統計機能を実行すると，拡張 verbosegc 情報に次の情報が出力されます。

- GC 種別
- 拡張スレッドダンプに GC が発生した要因

これらの情報は，jheapprof コマンドに指定する引数によって，出力される情報が異なります。jheapprof コマンドの引数と出力情報の関係を次の表に示します。

表 4–8　jheapprof コマンドの引数と出力情報の関係

jheapprof コマンドの引数	GC 種別	GC が発生した要因
-fullgc	FullGC	JHeapProf Command
-copygc	GC	JHeapProf Command

4.6.2　GC の選択の指針

統計前の GC 選択機能で，どの処理を指定するかは，出力されるクラス別統計情報の調査目的によって異なります。

ここでは，調査目的に合わせて，処理を選択する指針について説明します。

統計前の GC 選択機能で選択できる処理は，jheapprof コマンドの引数で指定します。処理は，どのオブジェクトを調査対象とするか，クラス別統計情報をどのような調査で使用するかによって選択します。

処理の選択の指針を次の表に示します。

表 4–9　処理の選択の指針

処理 （jheapprof コマンドの引数）	調査対象	クラス別統計情報の調査方法の例
FullGC の実行 （-fullgc）	FullGC によるオブジェクトの変化	FullGC によって，使用済みのオブジェクトが回収されるため，Java ヒープのメモリリークの原因となるオブジェクトを特定できます。
CopyGC の実行 （-copygc）	CopyGC によるオブジェクトの変化	CopyGC によって，Tenured 領域へ移動する寿命の長いオブジェクトを特定できます。この情報から，FullGC の発生頻度が増大する原因となるオブジェクトを特定できます。
GC は実行しない （-nogc）※	GC の実行で回収されてしまう短寿命オブジェクト	使用済みのオブジェクトを含むすべてのオブジェクトの情報が出力されます。CopyGC が多発している場合などに，その原因となるメモリ占有率の高いオブジェクト（超大オブジェクト）を特定できます。

注※

jheapprof コマンドの引数に-nogc を指定した場合，すべての寿命の短いオブジェクトの情報が出力されるため，ログの出力量が増加します。

4.7　Tenured 領域内不要オブジェクト統計機能

ここでは，Tenured 領域内不要オブジェクト統計機能について説明します。

Tenured 領域内不要オブジェクト統計機能は，クラス別統計情報を出力する機能の一つです。Tenured 領域内不要オブジェクト統計機能を使用すると，Tenured 領域内で不要となったオブジェクトを特定できます。

Tenured 領域内不要オブジェクト統計機能を使用する場合は，jheapprof コマンドの引数に-garbage を指定します。jheapprof コマンドの詳細については，「jheapprof（クラス別統計情報付き拡張スレッドダンプの出力)」を参照してください。

4.7.1　Tenured 領域内不要オブジェクト統計機能の概要

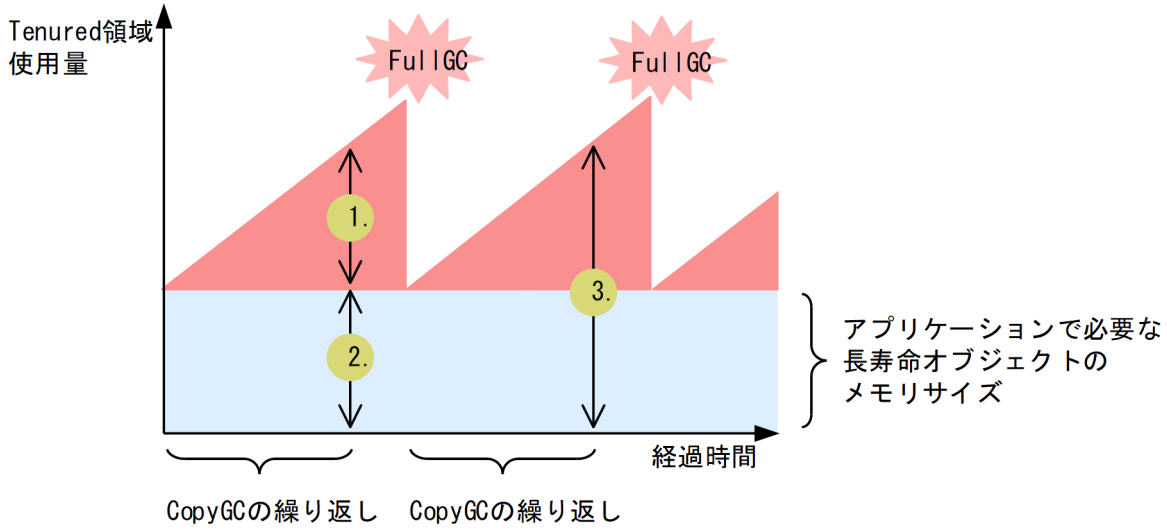
Tenured 領域内不要オブジェクト統計機能では，Tenured 領域内に蓄積された不要となったオブジェクトだけを特定して，スレッドダンプファイルに出力します。Tenured 領域内不要オブジェクト統計機能の仕組みについて説明します。

(1)　不要なオブジェクトのサイズの出力

CopyGC の繰り返しによって，長寿命オブジェクトが Tenured 領域に蓄積します。蓄積した長寿命オブジェクトのうち，時間が経過して用途を失ったオブジェクトは，不要なオブジェクトとなって Tenured 領域内に残ります。その後，メモリがいっぱいになったタイミングで FullGC が発生します。CopyGC の発生から FullGC の発生までの Tenured 領域の使用量は，Tenured 領域内不要オブジェクト統計機能，およびインスタンス統計機能で確認できます。

Tenured 領域内不要オブジェクト統計機能，およびインスタンス統計機能を使って特定できる内容を次の図に示します。

図 4-10　Tenured 領域内不要オブジェクト統計機能，およびインスタンス統計機能を使って特定できる内容



統計前 GC を実施しないでインスタンス統計機能を実行した場合，この図の 3.のサイズが出力されます。このサイズは，この図の 1.に該当する Tenured 領域内で不要となったオブジェクトのサイズ，およびこの図の 2.に該当する Tenured 領域内で使用中のオブジェクトを含んだ Tenured 領域内のメモリ使用状況になります。

一方，Tenured 領域内不要オブジェクト統計機能を実行した場合は，この図の 2.の使用中のオブジェクトを除いた Tenured 領域内のメモリ使用状況（この図の 1.に該当）を出力できます。Tenured 領域内不要オブジェクト統計機能を使うことで，Tenured 領域の増加要因となる不要となったオブジェクトを特定できるため，FullGC を抑止できます。

(2)　不要なオブジェクトの参照関係の確認

Tenured 領域内不要オブジェクト統計機能では，基点となるオブジェクトは Tenured 領域内のアドレスの低い順に検索されます。検索されたオブジェクトの中でも，ほかの参照関係で調べられていないオブジェクトが基点となるオブジェクトになります。

参照先のオブジェクトが調査済みの場合は，分岐点まで戻って参照関係を調べます。また，参照先のオブジェクトがほかの参照関係の基点となるオブジェクトである場合は，参照先オブジェクトとして扱います。すべての基点となるオブジェクトがなくなるまで参照関係を調べます。

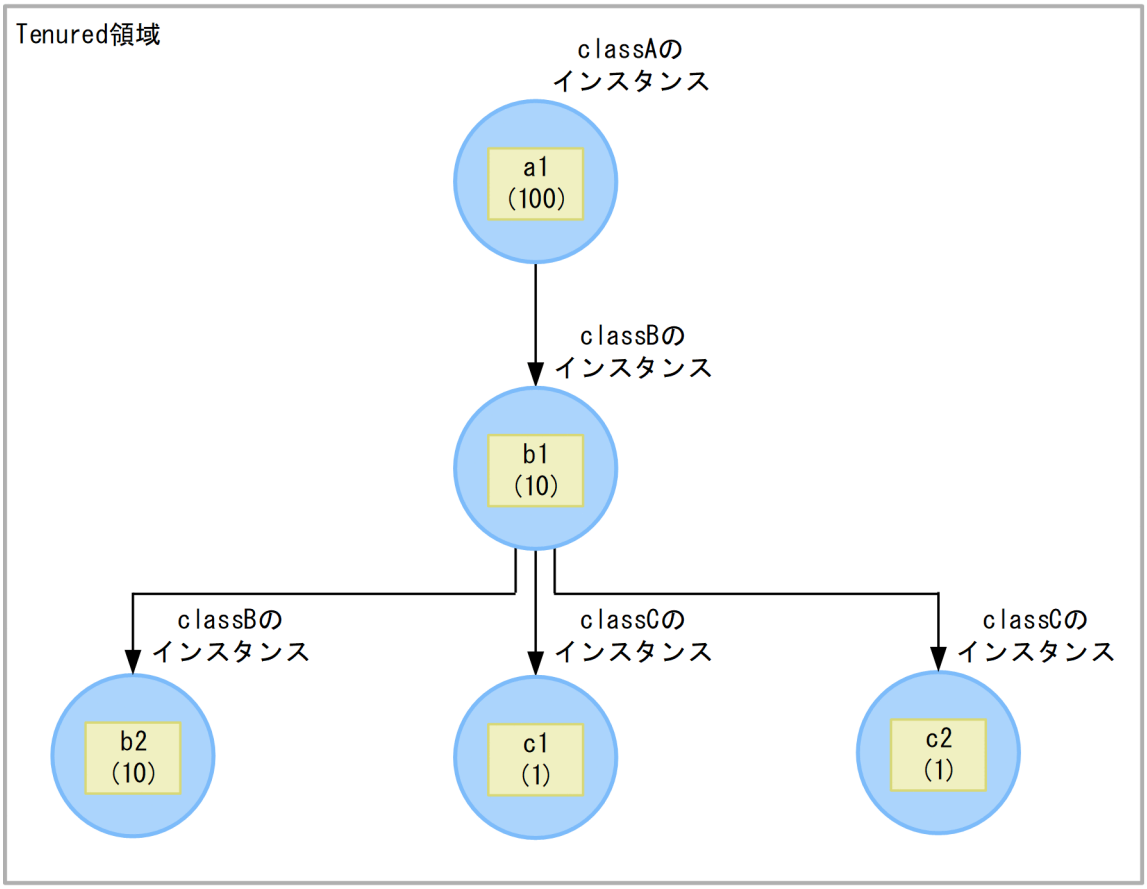
Tenured 領域内不要オブジェクト統計機能の場合，インスタンス数，およびインスタンスサイズの合計が出力されます。インスタンス数は該当するクラスを加算します。インスタンスの合計サイズには次の内容が出力されます。

- 基点となるオブジェクトのサイズは，該当するクラスに加算されます。参照先のオブジェクトのサイズは，該当するクラスに加算され，さらに基点となるオブジェクト，および該当するクラスまでの参照関係にあるすべてのオブジェクトの該当するクラスにも加算されます。

なお，Tenured 領域内不要オブジェクト統計機能を実行した場合，インスタンス統計機能，STATIC メンバ統計機能，および統計前の GC 選択機能は無効となります。

Tenured 領域内の不要オブジェクトによる参照関係の例を示します。

図 4-11　Tenured 領域内の不要なオブジェクトによる参照関係の例



（凡例）

：インスタンスを示します。

x ：メンバ変数を示します。( )内の値はサイズを示します。

→：参照を示します。

この図の参照関係について説明します。

インスタンス数

- classA：a1 が存在するため 1
- classB：b1，b2 の 2 つが存在するため 2
- classC：c1，c2 の 2 つが存在するため 2

インスタンスサイズの合計

- classA：classA のインスタンス（a1）と，その参照先のインスタンス（b1，b2，c1，c2）のサイズを合計した 122

- classB：classB のインスタンス（b1，b2）と，その参照先のインスタンス（c1，c2）のサイズを合計した 22
- classC：classC のインスタンス（c1，c2）を合計した 2

Tenured 領域内不要オブジェクト統計機能を実行して，「図 4-11」の参照関係の情報を出力した場合の出力例を次に示します。

```
Garbage Profile
---------------
_________________Size__Instances__Class_________________
                  122           1 A
                   22           2 B
                    2           2 C
```

4.7.2 Tenured 領域内不要オブジェクト統計機能で出力するクラス別統計情報

Tenured 領域内不要オブジェクト統計機能で出力するクラス別統計情報の出力形式，出力項目および出力例について説明します。

(1) 出力形式と出力項目

Tenured 領域内不要オブジェクト統計機能で出力するクラス別統計情報の出力形式を次に示します。

- 出力形式

```
Garbage Profile
-----------------------
_________________Size__Instances__Class_________________
            <サイズ>       <個数>          <クラス名>
            <サイズ>       <個数>          <クラス名>...
```

- 出力項目

  出力形式で示した各項目について説明します。

表 4-10 出力項目（Tenured 領域内不要オブジェクト統計機能）

出力項目	意味
<サイズ>	インスタンスの合計サイズがバイト単位で出力されます。
<個数>	インスタンスの数が出力されます。
<クラス名>	クラス名が出力されます。

(2) 出力例

Tenured 領域内不要オブジェクト統計機能で出力するクラス別統計情報の出力例を示します。

```
Garbage Profile
---------------
_____________Size__Instances__Class_____________
          35234568      10648 java.util.HashMap
           5678900      10668 [Ljava.util.HashMap$Entry;
           4456788       7436 java.util.HashMap$Entry
           4321000        200 java.util.WeakHashMap
           1234568        190 [Ljava.util.WeakHashMap$Entry
            454400          4 java.util.WeakHashMap$Entry
                 0          0 java.lang.Class
...
```

4.7.3 Tenured 領域内不要オブジェクト統計機能の実行に関する注意事項

Tenured 領域内不要オブジェクト統計機能の注意事項を次に示します。

(1) Tenured 領域内不要オブジェクト統計機能を FullGC 直後に実行した場合の注意事項

Tenured 領域内不要オブジェクト統計機能を FullGC 直後に実行すると，統計対象の Tenured 領域内の不要なオブジェクトが回収されている状態で統計処理を実行します。そのため，クラス別統計情報中のインスタンスサイズの合計，およびインスタンス数が小さくなり，不要なオブジェクトが効果的に特定されません。不要なオブジェクトを効果的に特定するためには，次の説明に沿って Tenured 領域内不要オブジェクト統計機能を実行してください。なお，Tenured 領域内不要オブジェクト統計機能の実行について，FullGC の発生のタイミングが判明している場合と，判明していない場合に分けて説明します。

(a) FullGC が発生するタイミングが判明している場合

Tenured 領域内不要オブジェクト統計機能を FullGC の直前に実行すると，統計対象である Tenured 領域内の不要オブジェクトの数が多い状態で統計処理が実行されます。そのため，クラス別統計情報中のインスタンスサイズの合計，およびインスタンス数が大きくなり，不要なオブジェクトを効果的に特定できます。

(b) FullGC が発生するタイミングが判明していない場合

クラス別統計情報中のインスタンスサイズの合計，およびインスタンス数をできるだけ大きくし，不要なオブジェクトを効果的に特定する方法を示します。

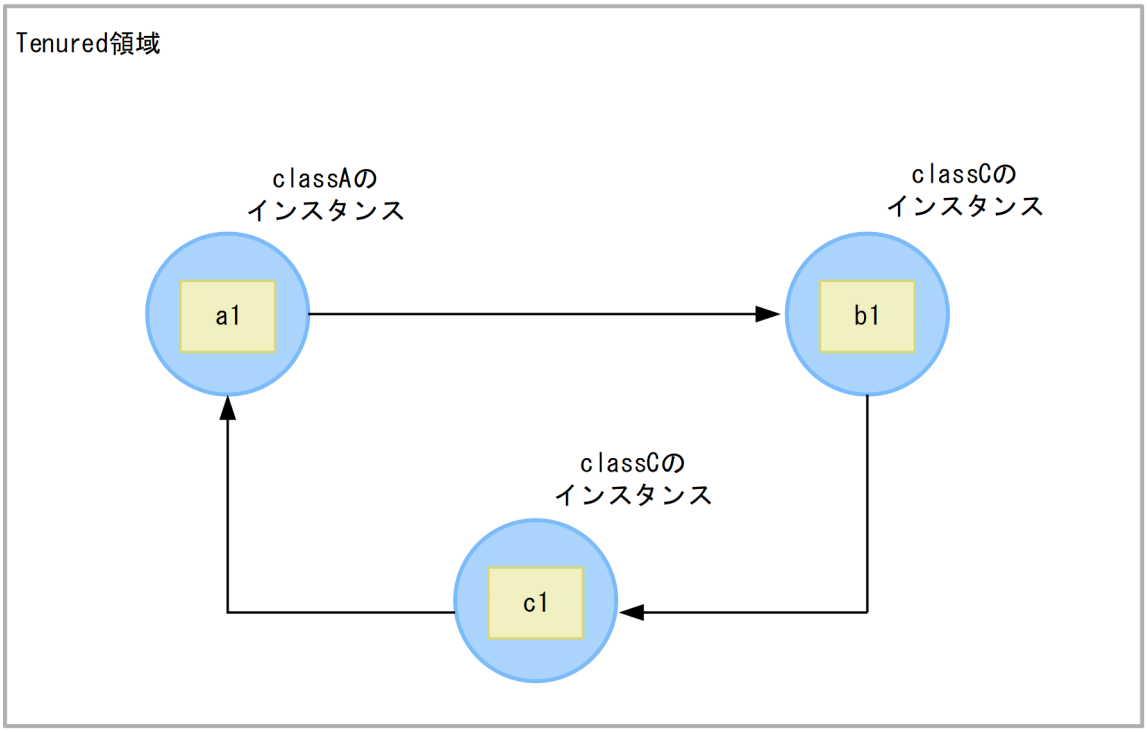
1. FullGC の発生のタイミングを知るために JavaVM 起動時に拡張 verbosegc 情報を出力するオプション-XX:+HitachiVerboseGC を設定します。オプションを指定することで GC の情報を取得できます。
2. JavaVM に対して Tenured 領域内不要オブジェクト統計機能を一定の間隔で実行します。それによって，GC の情報と，複数のクラス別統計情報を取得できます。

3. 拡張 verbosegc 情報から FullGC の日時を取得できるため，FullGC により近いクラス別統計情報を選択します。FullGC に近いクラス別統計情報は，統計対象である Tenured 領域内の不要なオブジェクトの数が多い状態で統計処理された情報です。

(2) 統計結果に関する注意事項

次の図のような参照関係の場合を例に統計結果に関する注意事項を説明します。

図 4-12 参照関係の例（統計結果に関する注意事項）



（凡例）

：インスタンスを示します。

x ：メンバ変数を示します。( )内の値はサイズを示します。

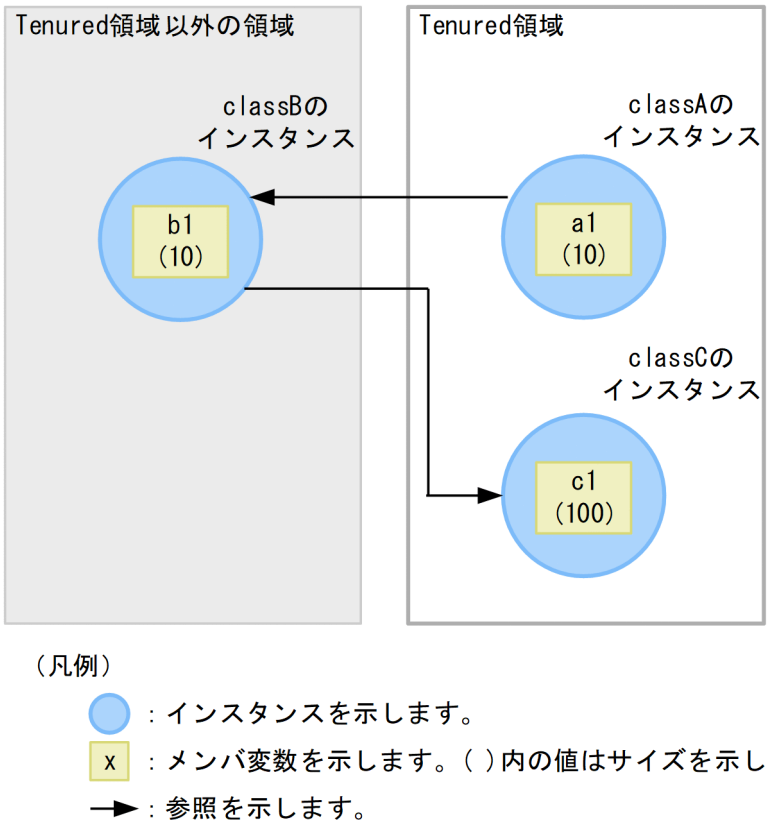
→：参照を示します。

この図で，a1 が最も下位のアドレスとした場合，a1 を基点となるオブジェクトとして，a1→b1→c1 という参照関係で統計処理を実行します。このとき，b1 または c1 を基点となるオブジェクトとしている場合，Tenured 領域内不要オブジェクト統計機能，および Tenured 増加要因の基点オブジェクトリスト機能の統計結果には，意図したとおりの結果が出力できません。

(3) 統計対象とならないオブジェクトに関する注意事項

次の図のように Tenured 領域のオブジェクト（a1 および c1）が，Tenured 領域以外のオブジェクト（b1）と参照関係がある場合を例に，統計対象とならないオブジェクトに関する注意事項を説明します。

図 4-13　参照関係の例（統計対象とならないオブジェクト）



（凡例）

：インスタンスを示します。

x　：メンバ変数を示します。( )内の値はサイズを示します。

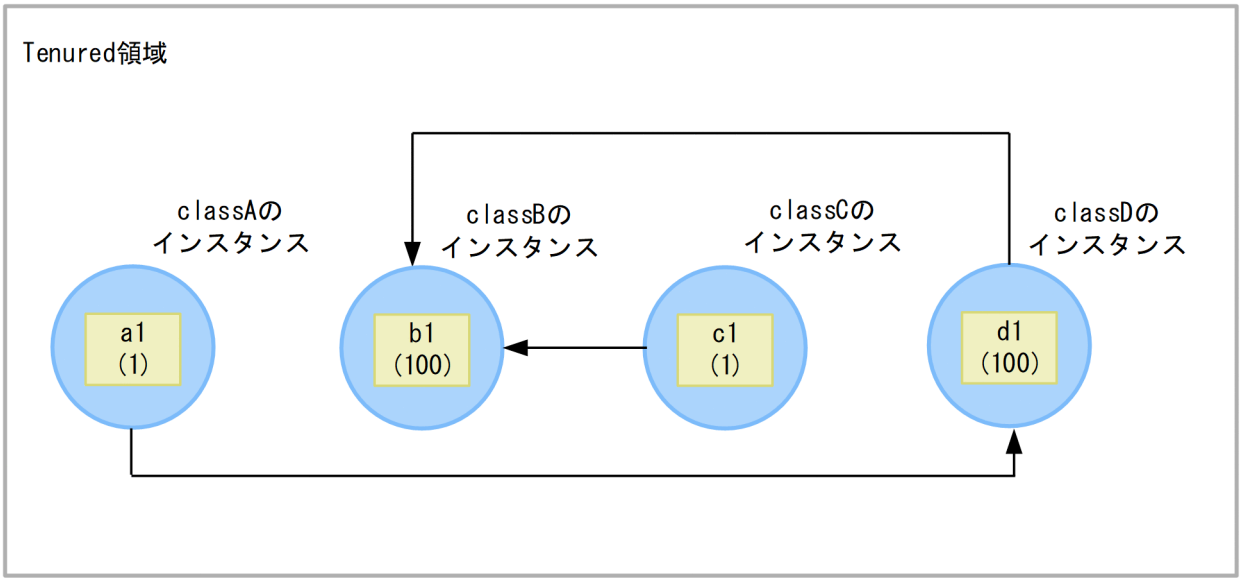
：参照を示します。

この図の場合，Tenured 領域以外のオブジェクト（b1）は統計対象とはなりません。しかし，クラス B のインスタンスサイズの合計には，参照先の Tenured 領域のオブジェクト（c1）のサイズが加算されます。

(4)　複数のオブジェクトから参照関係がある場合の注意事項

次の図のように複数の参照元（c1 および d1）から 1 つのオブジェクト（b1）を参照している参照関係がある場合を例に，複数のオブジェクトから参照関係がある場合の注意事項を説明します。

図 4-14　参照関係の例（複数のオブジェクトから参照関係がある場合）



（凡例）

：インスタンスを示します。

x ：メンバ変数を示します。( )内の値はサイズを示します。

→：参照を示します。

この図の場合，Tenured 領域内不要オブジェクト統計機能を実行すると，その参照元が属している参照関係の基点となるオブジェクト（c1 および a1）のうち，アドレスが最も低位である基点となるオブジェクトから統計処理が実行されます。そのため，アドレスが最も低位のオブジェクトが a1 で，基点となるオブジェクトが c1 である場合，Tenured 領域内不要オブジェクト統計機能，および Tenured 増加要因の基点オブジェクトリスト機能の統計結果には，意図したとおりの結果が出力できません。

(5)　統計値の増加に関する注意事項

オプション-XX:+HitachiAutoExplicitMemory を指定した Java プロセスに対して，Tenured 領域内不要オブジェクト統計機能，および Tenured 増加要因の基点オブジェクトリスト出力機能を実行すると，次に示すような現象が発生する場合があります。

- Tenured 領域内不要オブジェクト統計機能で出力されるクラス別統計情報のうち，float 型の配列型（クラス名に[F と出力される情報）のインスタンスサイズの合計，およびインスタンス数の統計値が，本来の統計値よりも大きくなります。クラス別統計情報の出力例を次に示します。

```
Garbage Profile
---------------
_____________Size__Instances__Class_____________
          43861400     473859 [F
                 0          0 java.util.Collections$EmptyMap
                 0          0 sun.security.util.Debug
                 0          0 java.nio.ByteOrder
```

- Tenured 増加要因の基点オブジェクトリスト出力機能で出力される Tenured 増加要因の基点オブジェクトリストに，float 型の配列型（クラス名に[F と出力される情報）が出力されることでインスタンスサイズの合計の統計値が，本来の統計値よりも大きくなります。Tenured 増加要因の基点オブジェクトリストの出力例を次に示します。

```
Garbage Profile Root Object Information
---------------------------------------
*, [F # 43861400
```

4.8　Tenured 増加要因の基点オブジェクトリスト出力機能

ここでは，Tenured 増加要因の基点オブジェクトリスト出力機能について説明します。

Tenured 増加要因の基点オブジェクトリスト出力機能は，クラス別統計情報を出力する機能の一つです。Tenured 増加要因の基点オブジェクトリスト出力機能は，Tenured 領域内不要オブジェクト統計機能を使って特定した，不要なオブジェクトの基点となるオブジェクトの情報を出力できます。

Tenured 増加要因の基点オブジェクトリスト出力機能を使用する場合は，jheapprof コマンドの引数に-rootobjectinfo を指定します。jheapprof コマンドの詳細については，「jheapprof（クラス別統計情報付き拡張スレッドダンプの出力）」を参照してください。

4.8.1　Tenured 増加要因の基点オブジェクトリスト出力機能の概要

Tenured 増加要因の基点オブジェクトリスト出力機能では，Tenured 領域内不要オブジェクト統計機能を使って特定した，不要なオブジェクトの基点となるオブジェクトの情報をリストにしてスレッドダンプファイルに出力します。

また，Tenured 増加要因の基点オブジェクトリスト出力機能は，Tenured 領域内不要オブジェクト統計機能が前提になります。

Tenured 増加要因の基点オブジェクトリスト出力機能の仕組みについて説明します。

(1)　基点となるオブジェクトのリスト出力

Tenured 増加要因の基点オブジェクトリスト出力機能は，Tenured 領域内不要オブジェクト統計機能の処理のあとに続けて実行されます。Tenured 領域内不要オブジェクト統計機能で実行する処理については，「4.7.1　Tenured 領域内不要オブジェクト統計機能の概要」を参照してください。

Tenured 増加要因の基点オブジェクトリスト出力機能は，Tenured 領域内のアドレスの低い順にオブジェクトを検索します。検索したオブジェクトの中で，すでに参照関係で調べられているオブジェクト，および不要なオブジェクトのクラス情報を取得して，出力候補リストに保存します。

出力候補リストに保存した情報をインスタンスサイズの合計でソートし，合計サイズが jheapprof コマンドの-rootobjectinfost オプションで指定した値以上のクラス情報だけが出力されます。

(2)　不要なオブジェクトの参照関係の確認

Tenured 増加要因の基点オブジェクトリスト出力機能を実行して基点となるオブジェクトを取得する際の，Tenured 領域内の不要オブジェクトによる参照関係の例を示します。

インスタンス数

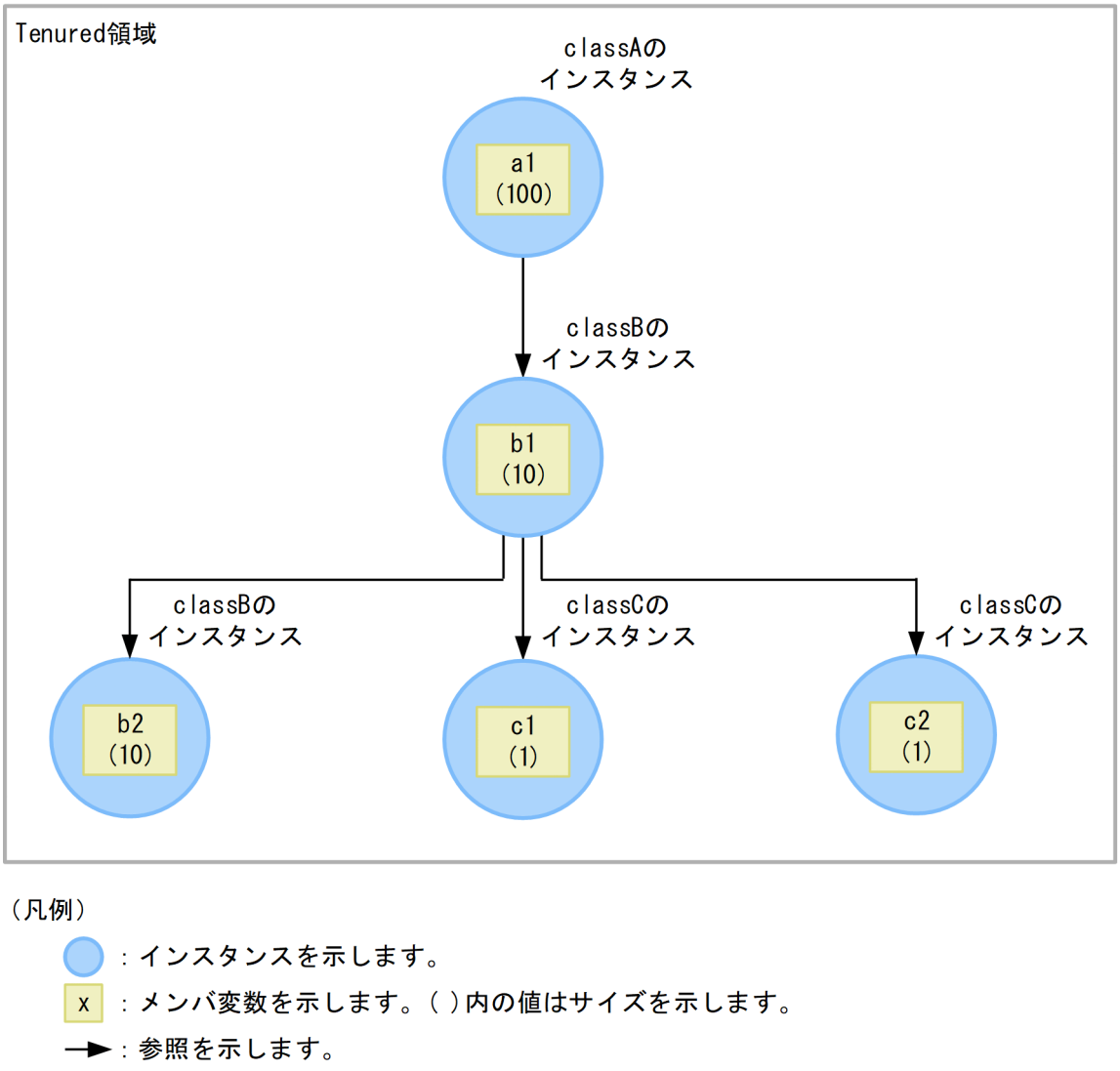
- classA：a1 が存在するため 1

- classB：b1，b2 の 2 つが存在するため 2
- classC：c1，c2 の 2 つが存在するため 2

インスタンスサイズの合計

- classA：classA のインスタンス（a1）と，その参照先のインスタンス（b1，b2，c1，c2）のサイズを合計した 122
- classB：classB のインスタンス（b1，b2）と，その参照先のインスタンス（c1，c2）のサイズを合計した 22
- classC：classC のインスタンス（c1，c2）を合計した 2

図 4-15　Tenured 領域内の不要なオブジェクトによる参照関係の例



Tenured 領域内不要オブジェクト統計機能を実行してこの図のような参照関係の情報を取得した場合に，Tenured 増加要因の基点オブジェクトリスト出力機能で取得する情報を次に示します。

- 基点オブジェクト：a1
- 基点オブジェクトのクラス：A

- 基点オブジェクトのクラス A のインスタンスサイズの合計：122

「図 4-15」の参照関係の情報を Tenured 増加要因の基点オブジェクトリスト出力機能を実行して出力した場合の出力例を次に示します。

```
Garbage Profile Root Object Information
---------------------------------------
*, A # 122
```

4.8.2 Tenured 増加要因の基点オブジェクトリスト出力機能で出力するクラス別統計情報

Tenured 増加要因の基点オブジェクトリスト出力機能で出力するクラス別統計情報の出力形式，および出力項目について説明します。

- 出力形式

```
Garbage Profile Root Object Information
---------------------------------------
*, <クラス名> # <サイズ>
…
```

- 出力項目

  出力形式で示した各項目について説明します。

表 4-11　出力項目（Tenured 領域内不要オブジェクト統計機能）

出力項目	意味
<クラス名>	Tenured 増加要因となった基点となるオブジェクトのクラス名が出力されます。
<サイズ>	Tenured 領域内不要オブジェクト統計機能のクラス別統計情報から取得したインスタンスサイズの合計がバイト単位で出力されます。

4.9　クラス別統計情報解析機能

ここでは，クラス別統計情報解析機能について説明します。

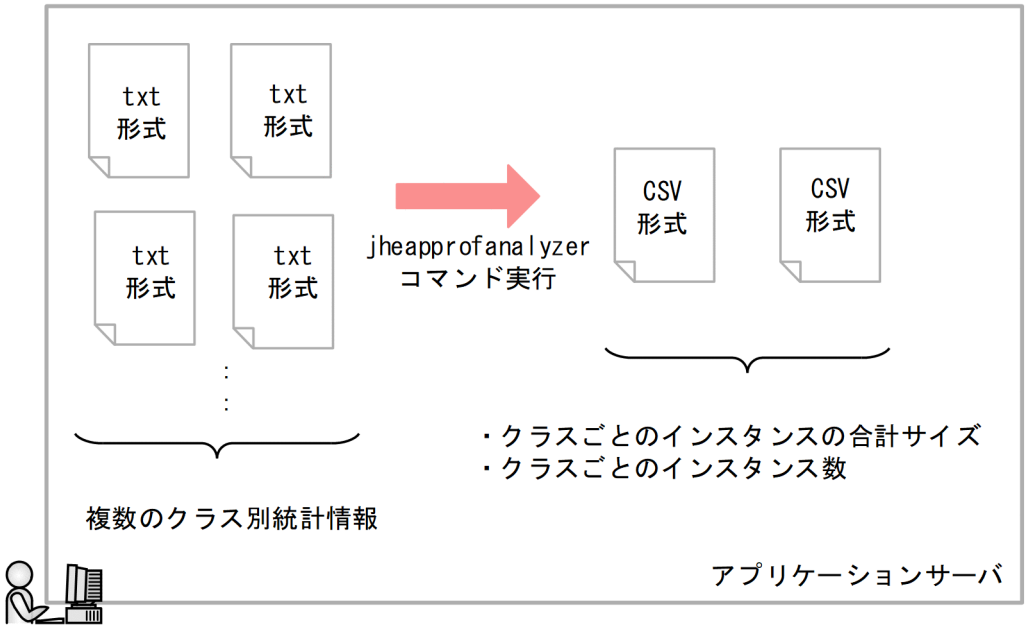
クラス別統計情報解析機能を使用すると，クラス別統計情報で取得した情報を CSV 形式で出力できます。

4.9.1　クラス別統計情報解析機能の概要

jheapprofanalyzer コマンド（クラス別統計情報解析機能）を実行することで，クラス別統計情報の付いた複数の拡張スレッドダンプファイルを入力ファイルとして，クラスごとのインスタンスの合計サイズ，およびクラスごとのインスタンス数を時系列に出力します。出力されるファイルは，CSV 形式で出力されます。

クラス別統計情報として取得した情報を基に，クラス別統計情報解析機能を実行して CSV 形式に出力する場合の流れを次に示します。

図 4-16　クラス別統計情報解析機能を実行して CSV 形式に出力する場合の流れ



クラス別統計情報解析機能では，インスタンスの合計サイズが大きいインスタンスの情報を出力して，そのインスタンスのメモリ使用量だけを確認することもできます。インスタンスの合計サイズが大きいものだけを出力する場合には-DJP.co.Hitachi.soft.jvm.tools.jheapprofanalyzer.threshold にしきい値を指定し，jheapprofanalyzer コマンドに指定して実行します。jheapprofanalyzer コマンドの詳細については，「jheapprofanalyzer（クラス別統計情報解析ファイルの CSV 出力）」を参照してください。

4.9.2　クラス別統計情報解析機能の出力例

クラス別統計情報解析機能の入力ファイル，出力ファイル，および出力形式について説明します。

(1)　入力ファイル

クラス別統計情報解析機能で使用する入力ファイルは，クラス別統計情報を出力した拡張スレッドダンプファイルです。

(2)　出力ファイル

クラス別統計情報解析機能で出力するファイルは，クラスごとのインスタンスの合計サイズを出力したファイル，およびクラスごとのインスタンス数を出力したファイルの 2 種類です。出力ファイルはカレントディレクトリに次のファイル名で作成されます。

表 4-12　出力ファイルのファイル名

出力ファイルの種類	出力ファイルのファイル名の例
インスタンス合計サイズファイル	JheapprofAnalyzer_size_*nnn*.csv
インスタンス数ファイル	JheapprofAnalyzer_num_*nnn*.csv

（凡例）

nnn：ファイルの分割番号が出力されます。分割番号の範囲は 001～999 です。

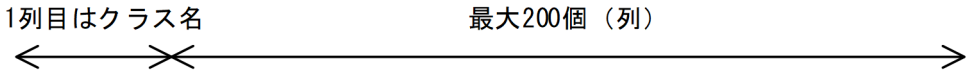
列が 201 列を超える場合は出力ファイルが分割されます。また，999 ファイルを超えた場合は，001 に戻りファイルは書き換えられます。

分割する列数は，201 列（クラス名 1 列＋値 200 列）を超えた場合とし，出力形式は分割したファイルも同じになります。

(3)　出力形式

クラス別統計情報解析機能で出力されるファイルの出力形式を次の図に示します。なお，インスタンス合計サイズ，およびインスタンス数が出力された CSV ファイルの出力形式は同じです。

図 4-17　クラス別統計情報解析機能で出力されるファイルの出力形式



class name,	入力ファイル名,	入力ファイル名,	・・・	入力ファイル名
クラス名,	値-1-1,	値-1-2,	・・・	値-1-xxx
:	:	:	・・・	:
クラス名,	値-y-1,	値-y-2,	・・・	値-y-xxx

（凡例）
入力ファイル名：処理対象に指定したクラス別統計情報
クラス名：入力ファイルに出力されていたクラス名
値：インスタンスの合計サイズ，またはインスタンス数

クラス名と値，および値と値の間はコンマで区切ります。また，行の最後は値（空白も含む）で終了します。

クラス名の出力順はランダムです。値は入力ファイルの先頭行にある日付を基に，日付の古いものから横に並びます。同じ日付の入力ファイルがある場合はランダムに連続して横に並びます。

メモ

クラス別統計情報解析機能を複数回実行すると，処理途中のクラスが消滅したり追加されたりする場合があります。また，該当するクラスがない場合の値には 0 が出力されます。次の図のクラス情報の例を使用して説明します。

図 4-18　クラス情報の例

1回目のクラス別統計情報（A.txt）

```
ClassA      100
ClassB      100
```

2回目のクラス別統計情報（B.txt）

```
ClassA      100
ClassB       30
ClassC       50
ClassB        0
```

3回目のクラス別統計情報（C.txt）

```
ClassA      100

ClassC       50
```

上記のようなクラス情報の場合で，-DJP.co.Hitachi.soft.jvm.tools.jheapprofanalyzer.threshold のしきい値を 0 にしたときの出力結果は次の図のようになります。

図 4-19　クラス別統計情報解析機能の出力例

```
class name,A.txt,B.txt,C.txt
ClassA,100,100,100
ClassB,100,30,0
ClassC,0,50,50
ClassD,0,0,0
```

インスタンス合計サイズの最大値は 0～2^{63}-1，インスタンス数の最大値は 0～2^{31}-1 です。1 つの入力ファイルに同じクラス名がある場合は，インスタンスサイズの合計が加算されます。また，インスタンス数も加算されます。加算されたことによって，それぞれの最大値を超えた場合は，指定した最大値が出力されます。なお，1 つのクラスについて，すべての入力ファイルで該当するクラスの情報がない，またはしきい値未満の場合は，そのクラスの情報は出力されません。

4.9.3　クラス別統計情報解析機能の注意事項

クラス別統計情報解析機能を使用する場合，jheapprofanalyzer コマンド実行中に入力ファイルの更新，および削除の操作はしないでください。クラス別統計情報解析機能は，日付を取得するとき，およびデータを読み込むときの 2 回ファイルを開きます。そのため，コマンド実行中に入力ファイルの更新，および削除の操作をした場合の結果は保証されません。

4.10　Survivor 領域の年齢分布情報出力機能

ここでは，Survivor 領域の年齢分布情報出力機能について説明します。

Survivor 領域の年齢分布情報出力機能を使用すると，CopyGC 実行時に，Survivor 領域の使用状況が調査できます。この情報は，メモリサイズのチューニングに使用できます。

4.10.1　Survivor 領域の年齢分布情報出力機能の概要

日立 JavaVM では，多くのトラブルシュート情報が取得できるように，ログの出力内容が標準の JavaVM よりも拡張されています。この JavaVM ログは，日立 JavaVM の JavaVM ログファイルに出力されます。Survivor 領域の年齢分布情報出力機能は，CopyGC 実行時，日立 JavaVM の JavaVM ログファイルに，Survivor 領域の Java オブジェクトの年齢分布情報を出力できる機能です。この情報を使用すると，Survivor 領域のオブジェクトの使用状況が調査でき，Survivor 領域のメモリサイズのチューニングができます。Survivor 領域のメモリサイズのチューニングについては，「2.2.4(1)　Java ヒープ内の Survivor 領域のメモリサイズの見積もり」を参照してください。

Survivor 領域の年齢分布情報出力機能は，Survivor 領域の年齢分布情報に加えて日時も出力できます。また，出力先が JavaVM ログファイルなので，ほかのログとの同期が取れます。

日立 JavaVM の JavaVM ログファイルについては，「7.3　JavaVM ログ（JavaVM ログファイル）」を参照してください。また，日立 JavaVM のチューニングについては，「2.　メモリチューニング」を参照してください。

4.10.2　Survivor 領域の年齢分布情報の出力形式と出力例

Survivor 領域の年齢分布情報は，CopyGC 発生時に，CopyGC のログのあとに続けて出力されます。Survivor 領域の年齢分布情報の出力形式と出力例を次に示します。

出力形式

```
[PTD]<date>[Desired survivor:size bytes][New threshold:value][MaxTenuringThreshold: max_value][age1:total_age1][age2:total_age2]...[agen:total_agen]
```

説明

- PTD：Survivor 領域の年齢情報であることを示す識別子
- date：GC が発生した日時（-XX:+HitachiVerboseGCPrintDate（拡張 verbosegc 情報日付出力オプション）を指定した場合だけ出力）※1
- size：Survivor 領域のサイズ（単位：バイト）
- value：次回の GC 発生時に Tenured 領域へ移動するオブジェクトの年齢のしきい値
- max_value：-XX:MaxTenuringThreshold の指定値※2

- total_age1：1 歳のオブジェクトが使用しているメモリサイズの合計（単位：バイト）
- total_age2：1 歳から 2 歳までのオブジェクトが使用しているメモリサイズの合計（単位：バイト）
- total_agen：1 歳から n 歳までのオブジェクトが使用しているメモリサイズの合計（単位：バイト）※3

注

-XX:+HitachiCommaVerboseGC を指定している場合は，次の形式で出力されます。

```
PTD,date,size,value,max_value,total_age1,total_age2,...,total_agen
```

注※1

対応する CopyGC のログと同じ時刻が表示されます。

注※2

-XX:MaxTenuringThreshold の指定値には，CopyGC 実行時に，From 空間と To 空間で Java オブジェクトを入れ替える回数のしきい値を設定します。

注※3

存在するオブジェクトを最小年齢から最大年齢まで順番に表示します。表示されるオブジェクトの最大年齢が max_value の値に近いと，寿命の長いオブジェクトが存在することを示します。

出力例

```
[VGC]<Wed May 28 11:45:23 2008>[GC 648K->136K(1984K), 0.0013020 secs][DefNew::Eden: 512K-
>0K(512K)][DefNew::Survivor: 0K->0K(64K)][Tenured: 136K->136K(1408K)][Metaspace: 3634K(44
92K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][cause:
ObjAllocFail][User: 0.0000000 secs][Sys: 0.0000000 secs]
[PTD]<Wed May 28 11:45:23 2008>[Desired survivor:5467547 bytes][New  threshold:30][MaxTen
uringThreshold:31][age1:1357527][age2:1539661]
```

この出力例では，次の内容が確認できます。

- 出力契機は，2008 年 5 月 28 日(水)11 時 45 分 23 秒に発生した CopyGC です。
- Survivor 領域のメモリサイズは 5,467,547 バイトです。Survivor 領域のオブジェクトは 2 歳までです。1 歳のオブジェクトが使用しているメモリサイズは 1,357,527 バイト，1 歳から 2 歳までのオブジェクトが使用しているメモリサイズは 1,539,661 バイトです。

4.10.3　実行環境での設定

Survivor 領域の年齢分布情報出力機能を使用する場合，次の設定が必要です。

指定するオプションを次に示します。

- -XX:+HitachiVerboseGCPrintTenuringDistribution
  Survivor 領域の年齢分布情報を日立 JavaVM の JavaVM ログファイルへ出力します。デフォルト値は，マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

4.10.4　Survivor 領域の年齢分布情報出力機能使用時の注意事項

Survivor 領域の年齢分布情報出力機能を使用すると，CopyGC 実行時のログが使用しない場合に比べて 倍以上に増えます。この機能は，Survivor 領域のチューニング時だけに使用することをお勧めします。

4.11　JIT コンパイル時の C ヒープ確保量の上限値設定機能

JavaVM がサポートしている Just In Time 方式でのコンパイル（**JIT コンパイル**）では，コンパイル実行中に C ヒープを使用します。処理数が多いメソッドや大量のメソッドに対して JIT コンパイルを実行した場合，コンパイル処理のために確保される C ヒープのサイズも多大になり，C ヒープ不足が発生することがあります。この場合，JavaVM の強制終了やアプリケーションサーバの異常終了などのトラブルが発生して，システムが全面停止してしまうおそれがあります。

このような問題の発生を防止するために，JIT コンパイルで使用する C ヒープのサイズに上限値を設定できます。上限値を超えた場合は，JIT コンパイルが中止され，以降のコンパイルはインタプリタ方式で実行されます。これによって，JavaVM の強制終了を防ぎ，システムの停止を抑止できます。

JIT コンパイル時の C ヒープ確保量の上限は，-XX:HitachiJITCompileMaxMemorySize オプションで指定します。-XX:HitachiJITCompileMaxMemorySize オプションの詳細は，「-XX:HitachiJITCompileMaxMemorySize（JIT コンパイル時の確保メモリ上限値指定オプション）」を参照してください。

4.12　スレッド数の上限値設定機能

アプリケーションで使用するスレッド数が多くなると，C ヒープで使用するメモリ使用量が増加します。メモリ使用量の増加に伴い C ヒープ不足が発生した場合，JavaVM の強制終了，アプリケーションサーバの異常終了などのトラブルが発生して，システムが全面停止してしまうおそれがあります。

このような問題の発生を防止するために，使用できるスレッド数に上限値を設定できます。あらかじめ使用するスレッド数の上限を把握し，その数を基に C ヒープに割り当てるメモリサイズを決定することで，C ヒープ不足の発生を防止します。なお，設定した上限値を超えたスレッドが生成された場合は，例外がスローされます。アプリケーションでこの例外をキャッチして，適切な対処をすることで，システムの停止を抑止できます。

スレッド数の上限は，-XX:HitachiThreadLimit オプションで指定します。-XX:HitachiThreadLimit オプションの詳細は，「-XX:HitachiThreadLimit（スレッド数の上限値指定オプション）」を参照してください。

4.13　ファイナライズ滞留解消機能

ここでは，ファイナライズ滞留解消機能について説明します。ファイナライズ滞留解消機能を使用すると，ファイナライズ処理の滞留を解消でき，OS 資源の解放遅れなどの発生を抑止できます。

4.13.1　ファイナライズ滞留解消機能の概要

finalize()メソッドで OS 資源の解放処理を定義しているアプリケーションでは，ファイナライズ処理が滞留して，OS 資源の解放遅れが発生することがあります。ファイナライズ滞留解消機能を使用すると，JavaVM 内でファイナライズ処理が滞留している状態を検知して，滞留しているファイナライズ処理を解消します。

ファイナライズ滞留解消機能では，ファイナライズ滞留を検知するために，Java 起動時に FinalizerThread を監視するファイナライズ処理監視スレッドを生成します。FinalizerThread は，Java 実行時に常に 1 つあり，オブジェクトの finalize()メソッドを 1 つずつ処理するスレッドです。

ファイナライズ処理監視スレッドでは，FinalizerThread で処理しているオブジェクトを定期的に監視します。次の条件を満たす場合，ファイナライズ処理が滞留していると見なして，新たにファイナライズスレッドを生成します。

- FinalizerThread でオブジェクトのファイナライズ処理が進んでいない。
- ファイナライズキューにオブジェクトがある。

FinalizerThread とファイナライズスレッドの両方で，ファイナライズ処理を実行して，滞留しているファイナライズ処理の解消を促します。なお，ファイナライズスレッドは，ファイナライズキューが空になると終了します。

4.13.2　ファイナライズ滞留解消機能の出力情報

ファイナライズ滞留解消機能では，次に示すタイミングでメッセージを標準出力に出力します。

- ファイナライズ処理の滞留を検知して，ファイナライズスレッドを新たに生成する場合
- 生成したファイナライズスレッドが終了した場合

それぞれのタイミングで出力するメッセージの形式と出力例を次に示します。

- ファイナライズ処理の滞留を検知して，ファイナライズスレッドを新たに生成する場合
  ファイナライズ処理の滞留を検知して，ファイナライズスレッドを新たに生成する場合に出力されるメッセージの形式を次に示します。

```
# FinalizerWatcherThread: Create: create secondary finalizer thread. [queue length = queue] <date>
```

説明

queue：ファイナライズキューの要素数

date：ファイナライズスレッドを新たに生成した日時

出力例を次に示します。

```
# FinalizerWatcherThread: Create: create secondary finalizer thread. [queue length = 128] <M
on May 26 18:00:36 JST 2008>
```

- 生成したファイナライズスレッドが終了した場合

  生成したファイナライズスレッドが終了した場合に出力されるメッセージの形式を次に示します。

  ```
  # FinalizerWatcherThread: Finish: secondary finalizer thread is finished. <date>
  ```

  説明

  date：生成したファイナライズスレッドが終了した日時

出力例を次に示します。

```
# FinalizerWatcherThread: Finish: secondary finalizer thread is finished.  <Mon May 26 20:12
:26 JST 2008>
```

4.13.3　ファイナライズ滞留解消機能の実行環境での設定

ファイナライズ滞留解消機能を使用する場合，次の設定が必要です。

JavaVM のプロパティに指定するオプションを次に示します。

- -DJP.co.Hitachi.soft.jvm.autofinalizer=true

  ファイナライズ滞留解消機能を有効にします。この機能を有効にすると，Java 起動時に監視スレッドを生成します。デフォルト値は true のため，無効にする場合は false を指定してください。

  プロパティのオプションの詳細は，「5.3　日立 JavaVM で使用するプロパティ」を参照してください。

4.13.4　ファイナライズ滞留解消機能の注意事項

ファイナライズ滞留解消機能を使用する場合の注意事項を次に示します。

- JavaVM のプロパティの値は，Java 起動時に参照されます。このため，JavaVM 実行中に java.lang.System.setProperty()メソッドなどで，JavaVM のプロパティの値を変更しても，ファイナライズ滞留解消機能は有効になりません。
- JavaVM のプロパティに同一のオプションが複数指定された場合，最後に指定されたオプションの値が有効になります。

- ファイナライズ滞留解消機能では，ファイナライズの滞留を監視するための常駐スレッドを 1 つ生成します。また，ファイナライズの滞留を検知すると，滞留を解消するためのスレッドを 1 つ生成します。ファイナライズの滞留が発生してもシステムに影響しない場合は，「-DJP.co.Hitachi.soft.jvm.autofinalizer=false」を指定して，ファイナライズ滞留解消機能を無効にすることで，ファイナライズの滞留を監視するスレッドとファイナライズの滞留を解消するスレッドの生成を抑止できます。

4.14　ログファイルの非同期出力機能

ここではログファイルの非同期出力機能について説明します。

4.14.1　ログファイルの非同期出力機能の概要

この機能は，ログのファイル出力を非同期でできるように変更する機能です。ここに記載している「非同期」とは，ログのファイル出力処理を専用でするスレッドを用意することで，ログのファイル出力以外の処理を実行するスレッドとは処理のタイミングを合わせることなく，ログをファイルに出力していくことを指します。ここでは，例として，GC 発生時に出力されるログファイルについて説明します。

GC の実行中は，プログラムの処理が停止します。日立 JavaVM ログファイル機能では，GC を実行しているスレッドが，GC の情報のログファイル出力処理もしているため，GC の情報をログファイルに出力する処理が完了するまで次の処理へ進むことができません。

そこで，この機能を有効にすることによって，GC 情報のログファイル出力処理は専用のスレッドが実行するようになります。そのため，それ以外の処理を実行するスレッドが GC 情報のログファイル出力処理の完了を待つことによる処理の遅延を削減することができます。

しかし，この機能を使用することでログの一部が欠けるおそれがあります。この機能は，一時的にログをバッファにためておき，その後ログファイルへ出力するという処理をしています。ログの出力量が非常に多い場合や，ファイル入出力処理が極端に遅い場合に，一部のログが出力されないおそれがあります。そのため，GC によるプログラムの処理の停止時間として 100msec 以下を達成するようなチューニングをしている場合は，この機能の使用を推奨します。一方で，100msec 以下を達成するようなチューニングをしていない場合には，この機能の使用は推奨しません。

4.14.2　ログファイルの非同期出力機能の対象ログファイル

この機能の対象とするファイルを次に示します。

- 日立 JavaVM の JavaVM ログファイル

4.14.3　ログファイルの非同期出力機能のエラーケース

ファイルへの I/O 処理速度より，バッファへの書き込み処理速度が速い場合，バッファがラウンドしてログが出力されない場合があります。

バッファがラウンドしてログが出力されなかった場合の出力様式

```
[ASY]<skip_count> log is skipped.

skip_count：バッファがラウンドして出力できなかった場合のスキップした行数
```

この機能で使用するログに出力する専用のスレッドの生成に失敗した場合は，次のメッセージを標準出力に出力し，JavaVM を起動しません。

JavaVM ログファイル用のスレッドの生成に失敗

```
Error occurred during initialization of VM
Could not create thread for VM: Asynchronous javalog thread creation failed. thread_count th
reads exist.

thread_count: 存在するスレッド数
```

4.14.4　ログファイルの非同期出力機能の注意事項

ログファイルの非同期出力機能の使用時，-XX:HitachiOutOfMemoryStackTraceLineSize または-XX:HitachiJavaClassLibTraceLineSize オプションに 4096 より大きな値を指定した場合は，オプションの値として 4096 が指定されたものとして動作します。その場合の動作の詳細については，「5.2　日立 JavaVM 拡張オプションの詳細」を参照してください。

4.14.5　ログファイルの非同期出力機能のメモリ所要量

ログファイルの非同期出力機能は，新たにバッファとスレッドを生成します。この機能が有効の場合，新たに生成されたバッファとスレッドは JavaVM 終了時まで生存します。したがって，ログファイルの非同期出力機能を使用することでバッファとスレッド用にメモリを消費します。メモリ消費量を次に説明します。

まず，バッファのメモリ消費量を次に示します。

```
（ログ1行分のサイズ = 4096バイト）×（バッファにためこめる行数 = 1024行）= 4メガバイト
```

また，スレッドのメモリ消費量は-Xss オプションに指定された値です。

表 4-13　プラットフォームごとの-Xss のデフォルト値

プラットフォーム	-Xss のデフォルト値
Linux	1 メガバイト

4.15　圧縮オブジェクトポインタ機能

ここでは，圧縮オブジェクトポインタ機能について説明します。

4.15.1　圧縮オブジェクトポインタ機能の概要

圧縮オブジェクトポインタ機能は，Java アプリケーションによって作成される Java オブジェクトのサイズを圧縮することで，Java アプリケーションのメモリ使用効率を向上させる機能です。この機能を有効にすることによって，アプリケーションサーバ上で動作している Java アプリケーション実行中での，Java ヒープ領域の使用量を少なくできます。

圧縮オブジェクトポインタ機能は，-XX:+UseCompressedOops を指定することで有効にできます。UseCompressedOops オプションの詳細については，「5.2　日立 JavaVM 拡張オプションの詳細」を参照してください。

この機能を有効にすると，実行時のメモリ使用量と JavaVM の実行性能が変化する場合があります。また，JavaVM 実行時の OS 環境の違いによっても，この機能有効時の JavaVM の実行性能が変動します。

4.15.2　圧縮オブジェクトポインタ機能の前提条件

Java ヒープ領域の設定サイズの総和が 32GB 未満のときに有効です。設定サイズが 32GB 以上の場合は，-XX:+UseCompressedOops を指定しても，この機能は無効となります。

圧縮オブジェクトポインタ機能が有効となるヒープ領域サイズ指定値
（-Xmxオプションの指定値）＜ 32GB

4.15.3　圧縮オブジェクトポインタ機能の注意事項

1. この機能を有効にすることによって，Java アプリケーション実行時の Java ヒープ領域の使用量が減少し，その変化量は Java アプリケーションに依存します。もし，この機能を有効にしたことによるメモリ使用量の変化を問題視する場合は，この機能を OFF にしてください。
2. この機能を有効にすることによって，Java アプリケーション実行時のスループットなどの実行性能に影響を与える場合があるのでご注意ください。この機能が実行性能に影響する点は，次の 2 点です。

   (a)JavaVM が扱うデータ（Java オブジェクト）サイズが減少し，より効率的なメモリアクセスができる（性能向上要因）

   (b)JavaVM が Java オブジェクトのサイズを圧縮するための計算が必要となる（性能低下要因）

   (a)(b)の 2 点の要因から，この機能有効時，JavaVM 実行時の性能が変化することになります。また，(b)については，JavaVM 起動時の OS 環境の違いによって，計算方法（詳細は次の（補足））も変化し

実行性能に影響します。もし，この機能を有効にしたことによる性能の変化を問題視する場合は，この機能を OFF にしてください。

（補足）Java オブジェクトのサイズの圧縮について

圧縮オブジェクトポインタ機能有効時，JavaVM は Java オブジェクトのサイズを圧縮して管理しますが，その圧縮方法（以降，モードと呼びます）は複数存在します。JavaVM は起動時に，その実行環境状況によってどのモードを選択するかを決定します。以降，JavaVM は起動時に選択したモードを適用して，Java オブジェクトのサイズを圧縮します。

どのモードを選択するかの条件は，次の 2 つとなります。

1. Java ヒープ領域の指定値

2. 1.のヒープ領域が仮想アドレス空間のどこに割り当てられるか

そのため，JavaVM 起動時のヒープ領域サイズの指定値を変更した場合，または，同一 OS 上で動作しているほかのプロセスやライブラリのロード状況によっては，JavaVM 起動ごとに異なるモードが適用されます。

どのモードが選択されるかは，実際にヒープ領域が仮想メモリ空間上のどこに割り当てられるかに依存するため，特定の圧縮モードが常に適用されるようにこの機能を有効とすることは困難です。この機能を無効・有効とした場合だけ性能の差異が発生するだけでなく，この機能を有効とした場合も，JavaVM の起動ごとに性能の差異が発生する場合もあることにご注意ください。

参考として圧縮オブジェクトポインタ機能有効時に JavaVM が選択できる各モードについて，選択条件と圧縮方法の概要を次の表※に示します。

表 4-14　圧縮オブジェクトポインタ機能のモード一覧

モード	選択条件	概要説明
モード 1	Java ヒープ領域の総サイズが 4GB 未満であり，仮想アドレス空間上，Java ヒープ領域が 4GB よりも低位側に存在する場合	最も簡単な計算方法で，Java オブジェクトのサイズを圧縮します。
モード 2	モード 1 の適用ができないで，仮想アドレス空間上，そのヒープ領域が 32GB よりも低位側に存在する場合	モード 1 の次に簡単な計算方法によって Java オブジェクトのサイズを圧縮します。
モード 3	モード 1 とモード 2 が適用できなかった場合	これらモードのうち，最も複雑な方法で Java オブジェクトの圧縮をします。

注※

表に示した内容は JavaVM の内部処理を参考として示したものです。以降の JDK のアップデートや修正によって予告なく変更される場合があります。

4.16　日立 JavaVM の機能使用時の注意事項

日立 JavaVM の機能を使用する際の注意事項について説明します。

- SIGXFSZ シグナル受信時の動作について

  JavaVM が SIGXFSZ シグナルを受信した場合，次のメッセージを標準出力に出力して処理を継続します。

  ```
  Java HotSpot(TM) 64-Bit Server VM warning: File size limit exceeded.
  ```

  ただし，ファイルサイズの上限を超える場合は，ファイルへ書き込みません。

- SIGXCPU シグナル受信時の動作について

  JavaVM が SIGXCPU シグナルを受信した場合，次のメッセージを標準出力に出力して処理を継続します。

  ```
  Java HotSpot(TM) 64-Bit Server VM warning: CPU time limit exceeded.
  ```

- フォントに関するメッセージについて

  X サーバのようにコンソール以外のリモート端末上で，GUI のアプリケーションを起動する際，次のようなメッセージが出力される場合があります。

  ```
  Warning: Cannot convert string
  ”-monotype-arial-regular-r-normal--*-140-*-*-p-*-iso8859-1” to type FontStruct
  ```

  この場合，xfs コマンドでフォントサーバを起動し，X サーバ側で xset コマンドを使用してフォントパスにフォントサーバを指定すると回避できます。その際，フォントサーバの設定ファイルに不足しているフォントパスが指定されていることを確認してください。

4.17　Oracle 社の JDK と日立が提供する JDK との非互換性

ここでは，Oracle 社の JDK と日立が提供する JDK との非互換性について説明します。

4.17.1　デフォルトで選択されるメモリ管理方式

Oracle 社の JDK 9 以降では，デフォルトで選択されるメモリ管理方式はガベージファースト・コレクタ（G1GC）に変更されました。しかし，日立が提供するメモリ管理方式は，シリアルガベージコレクタ（SerialGC）であることに注意してください。

4.17.2　ランタイムイメージ

ランタイムイメージの作成および再構成はサポートしていません。また，ランタイムイメージを作成する jlink コマンドは同梱していません。

4.17.3　モジュール関連オプション

モジュールに関連したオプションで，サポートしていないオプションまたは制限事項のあるオプションがあります。

(1)　サポートしていないオプション

次に示すオプションはサポートしていません。

- --limit-modules
- --module（または-m）
- --patch-module
- --upgrade-module-path

各オプションの詳細は，次の Web ページを参照してください。

https://docs.oracle.com/en/java/javase/11/tools/java.html

(2)　制限事項のあるオプション

次に示すオプションには制限事項があります。これらのオプションには，更新対象モジュールに JDK 内モジュール※を指定しないでください。

- --add-exports

- --add-opens
- --add-reads

注※

JDK 内モジュールとは，次の Web ページに記載されているすべてのモジュールです。
https://docs.oracle.com/javase/jp/11/docs/api/index.html

各オプションの詳細は，次の Web ページを参照してください。

https://docs.oracle.com/en/java/javase/11/tools/java.html

制限事項の対象となる使用例を次に示します。

(使用例 1)

--add-exports=java.base/jdk.internal.jimage=my.module
更新対象に java.base モジュールを指定しないでください。

(使用例 2)

--add-opens=java.base/jdk.internal.jimage=my.module
更新対象に java.base モジュールを指定しないでください。

(使用例 3)

--add-reads=java.logging=my.module
更新対象に java.logging モジュールを指定しないでください。

5 日立 JavaVM 起動オプション

この章では，日立 JavaVM 起動オプションについて説明します。

日立 JavaVM では JavaVM 起動オプションとして，Java HotSpot VM のオプションのほか，日立固有の拡張オプションを指定できます。ここでは，拡張オプションの詳細，および日立 JavaVM で指定できる Java HotSpot VM のオプションについて説明します。

5.1　日立 JavaVM 拡張オプションの一覧

日立 JavaVM は，日立固有の拡張オプションを使用できます。

日立 JavaVM 拡張オプションの一覧を，次の表に示します。

重要

指定できるオプションについて

日立 JavaVM で指定できるオプションは，java コマンドに対してオプション指定なしで起動した場合に表示されるオプションと，java コマンドに対して-X か-XX か-XX:+Hitachi オプションを指定したときに表示されるオプションだけです。それ以外のオプションを指定した場合，動作は保証しません。

次の表の「関連情報」とは，指定したキーに関する情報の参照先です。

表 5-1　日立 JavaVM 拡張オプションの一覧

分類	オプション名称	概要	関連情報
一覧表示オプション	-XX:+Hitachi	日立 JavaVM 拡張オプションの一覧を表示します。	
拡張スレッドダンプ機能オプション	-XX:[+|-]HitachiThreadDump	拡張スレッドダンプ情報を出力するかどうかを指定します。	
	-XX:[+|-]HitachiThreadDumpToStdout	標準出力にスレッドダンプを出力するかどうかを指定します。	
	-XX:[+|-]HitachiThreadDumpWithHashCode	スレッド情報にハッシュコードを出力するかどうかを指定します。	
	-XX:[+|-]HitachiThreadDumpWithCpuTime	スレッド情報にユーザ CPU 時間とカーネル CPU 時間を出力するかどうかを指定します。	
	-XX:[+|-]HitachiThreadDumpWithBlockCount	スレッド情報に処理をブロックした回数と待ち状態になった回数を出力するかどうかを指定します。	
JavaVM ログファイルオプション	-XX:HitachiJavaLog※1	ログファイル名のプリフィックスを指定します。	
	-XX:HitachiJavaLogFileSize※1	1 ファイルの最大ファイルサイズを指定します。	

分類	オプション名称	概要	関連情報
	-XX:[+|-]HitachiJavaLogNoMoreOutput※1	ログファイル作成時に，入出力エラーが発生した場合の動作について指定します。	
	-XX:HitachiJavaLogNumberOfFile※1	作成するログファイルの最大ファイル数を指定します。	
	-XX:[+|-]JavaLogAsynchronous	ログファイルの非同期出力機能を有効にします。	
	-XX:[+|-]StandardLogToHitachiJavaLog	日立 JavaVM の標準出力と標準エラー出力の内容を，JavaVM ログファイルに出力するかどうかを指定します。	
詳細時間出力オプション	-XX:[+|-]HitachiOutputMilliTime	ミリ秒までの時間を出力するかどうかを指定します。	
拡張 verbosegc 機能オプション	-XX:[+|-]HitachiVerboseGC※2	GC が発生したときの拡張 verbosegc 情報を出力するかどうかを指定します。	
	-XX:[+|-]HitachiCommaVerboseGC	拡張 verbosegc 情報を CSV 形式で出力するかどうかを指定します。	
	-XX:HitachiVerboseGCIntervalTime	拡張 verbosegc 情報を出力する時間の間隔を指定します。	
	-XX:[+|-]HitachiVerboseGCPrintCause	GC の要因内容を出力するかどうかを指定します。	
	-XX:[+|-]HitachiVerboseGCPrintDate	拡張 verbosegc 情報に日付を出力するかどうかを指定します。	
	-XX:[+|-]HitachiVerboseGCCpuTime	GC のプロセッサ時間を出力するかどうかを指定します。	
	-XX:[+|-]HitachiVerboseGCPrintTenuringDistribution	GC 発生時に JavaVM ログファイルへ Survivor 領域のオブジェクトの年齢分布を出力します。	
	-XX:[+|-]HitachiVerboseGCPrintJVMInternalMemory	日立 JavaVM 内部で管理しているヒープ情報を JavaVM ログファイルへ	

分類	オプション名称	概要	関連情報
		出力するかどうかを指定します。	
	-XX:[+|-]HitachiVerboseGCPrintThreadCount	Java スレッドの数を監視するために，Java スレッドの数を JavaVM ログファイルに出力するかどうかを指定します。	
	-XX:[+|-]HitachiVerboseGCPrintDeleteOnExit	java.io.File.deleteOnExit()を呼び出したことによって日立 JavaVM が確保した累積のヒープサイズとメソッドの呼び出し回数を JavaVM ログファイルに出力するかどうかを指定します。	
コードキャッシュ領域情報出力機能オプション	-XX:[+|-]PrintCodeCacheInfo	コードキャッシュ領域の使用量を出力するかどうか，また，使用量がしきい値に達したことを知らせるメッセージを出力するかどうかを指定します。	「7.3.3　コードキャッシュ領域に関するログの内容」
	-XX:CodeCacheInfoPrintRatio	コードキャッシュ領域の使用量がしきい値に達したことを知らせるメッセージを出力する契機となる，コードキャッシュ領域の使用率を指定します。	
	-XX:[+|-]PrintCodeCacheFullMessage	Java メソッドが JIT コンパイルの対象になった場合，コードキャッシュ領域が枯渇していたときにメッセージを出力するかどうかを指定します。	「7.3.3　コードキャッシュ領域に関するログの内容」
OutOfMemoryError 発生時の拡張機能オプション	-XX:[+|-]HitachiOutOfMemoryCause※2	OutOfMemoryError 発生時の発生要因種別を出力するかどうかを指定します。	
	-XX:[+|-]HitachiOutOfMemoryStackTrace※2	OutOfMemoryError 発生時のスタックトレースを出力するかどうかを指定します。	
	-XX:HitachiOutOfMemoryStackTraceLineSize	OutOfMemoryError 発生時に出力するスタック	

分類	オプション名称	概要	関連情報
		トレースの 1 行の文字数を指定します。	
	-XX:[+|-]HitachiOutOfMemorySize※2	OutOfMemoryError 発生時に要求したメモリのサイズを出力します。	
	-XX:[+|-]HitachiOutOfMemoryAbort	OutOfMemoryError 発生時，メッセージとメモリダンプを出力して強制終了するかどうかを指定します。	
	-XX:[+|-]HitachiOutOfMemoryAbortThreadDump	OutOfMemoryError 発生時にスレッドダンプを出力かどうかを指定します。	
	-XX:[+|-]HitachiOutOfMemoryAbortThreadDumpWithJHeapProf	-XX:+HitachiOutOfMemoryAbortThreadDump で出力するスレッドダンプログファイルにクラス別統計情報を出力します。	
	-XX:[+|-]HitachiOutOfMemoryHandling	OutOfMemory ハンドリング機能を有効にするかどうかを指定します。	
	-XX:HitachiOutOfMemoryHandlingMaxThrowCount	OutOfMemory ハンドリング機能を有効にした場合の，Java ヒープ不足または Metaspace，Compressed Class Space 不足が原因の OutOfMemory 発生回数合計値の 1 時間当たりの上限値を指定します。	
クラスライブラリトレース機能オプション	-XX:[+|-]HitachiJavaClassLibTrace※2	クラスライブラリのスタックトレースを出力するかどうかを指定します。	
	-XX:HitachiJavaClassLibTraceLineSize	クラスライブラリのスタックトレースの 1 行の文字数を指定します。	
ローカル変数情報出力機能オプション	-XX:[+|-]HitachiLocalsInThrowable	例外発生時のスタックトレースに，ローカル変数情報を出力するかどうかを指定します。	

分類	オプション名称	概要	関連情報
	-XX:[+|-]HitachiLocalsInStackTrace	スレッドダンプ出力時のスタックトレースに，ローカル変数情報を出力するかどうかを指定します。	「7.5 JavaVM スタックトレース情報」
	-XX:[+|-]HitachiLocalsSimpleFormat	ローカル変数情報出力を，簡易フォーマットにするかどうかを指定します。	「7.5 JavaVM スタックトレース情報」
	-XX:[+|-]HitachiTrueTypeInLocals	ローカル変数情報出力時に，ローカル変数オブジェクトの実際の型名を文字列として出力するかどうかを指定します。	「7.5 JavaVM スタックトレース情報」
	-XX:HitachiCallToString	ローカル変数情報出力時に，ローカル変数オブジェクトの変数値を文字列として出力するかどうかを指定します。	「7.5 JavaVM スタックトレース情報」
システムリソース解除オプション	-XX:[+|-]HitachiFullCore	システムリソース RLIMIT_CORE の設定を変更するかどうかを指定します。	
リソースの上限値指定オプション	-XX:HitachiJITCompileMaxMemorySize	JIT コンパイル時に確保するメモリの上限値を指定します。	
	-XX:HitachiThreadLimit	スレッド数の上限値を指定します。	
JIT コンパイラ稼働継続機能オプション	-XX:[+|-]JITCompilerContinuation	JIT コンパイラ稼働継続機能を有効にするかどうかを指定します。	
圧縮オブジェクトポインタ機能オプション	-XX:[+|-]UseCompressedOops	圧縮オブジェクトポインタ機能を有効にするかどうかを設定します。	「4.15　圧縮オブジェクトポインタ機能」

（凡例）

空欄：関連情報はありません。

注※1

JavaVM ログファイルについての設定です。

注※2

次のオプションを指定した場合，JavaVM ログファイルが出力されます。

```
-XX:+HitachiOutOfMemoryStackTrace
-XX:+HitachiOutOfMemoryCause
```

```
-XX:+HitachiOutOfMemorySize
-XX:+HitachiVerboseGC
-XX:+HitachiJavaClassLibTrace
```

5.2 日立 JavaVM 拡張オプションの詳細

日立 JavaVM 拡張オプションの詳細について説明します。

なお，本文中では，次の Java プログラムを例として使用しています。

Java プログラム例 1

```
class Example1 {
    public static void main(String[] args) {
        Example1 e1 = new Example1();
        Object obj = new Object();
        e1.method(1, 'Q', obj);  // 5行目
    }

    void method(int l1, char l2, Object l3) {
        float l4 = 4.0f;
        boolean l5 = true;
        double l6 = Double.MAX_VALUE; // double型の最大値
        Object[] l7 = new Object[10];

        try {
          <例外発生!>  // 15行目
        } catch (Exception e) {
          e.printStackTrace();
        }
    }
}
```

Java プログラム例 2

```
class Example2 {
    public static void main(String[] args) {
        Example2 e2 = new Example2();
        e2.method();  // 4行目
    }

    synchronized void method() {
        int l1 = 1;
        float l2 = 2.0f;
        String l3 = "local 3";
        Character l4 = new Character('X');
        Object l5 = new Thread();
        Object[] l6 = new Thread[10];

        <ここでスレッドダンプ出力!>  // 15行目
    }
}
```

Java プログラム例 3

```
class Example3 {
    public static void main(String[] args) {
        Example3 e3 = new Example3();
        e3.method();  // 4行目
    }
```

```
    void method() {
        String l1 = "local 1";
        StringBuffer l2 = new StringBuffer(l1);
        l2.append(" + local 2");
        Boolean l3 = new Boolean(false);
        Character l4 = new Character('X');
        Long l5 = new Long(Long.MIN_VALUE);  // long型の最小値
        Object l6 = new Thread();
        Object[] l7 = new Thread[10];

        try {
          <例外発生!>  // 18行目
        } catch (Exception e) {
          e.printStackTrace();
        }
    }

    public String toString() {
        return "I am an Example3 instance.";
    }
}
```

-XX:+Hitachi（一覧表示オプション）

形式

```
-XX:+Hitachi
```

説明

日立 JavaVM 拡張オプションを一覧表示します。

このオプションは，Java プログラムを実行しません。また，アプリケーションサーバの起動オプションに指定した場合，アプリケーションサーバは起動されません。

-XX:[+|-]HitachiThreadDump（拡張スレッドダンプ情報出力オプション）

形式

```
-XX:+HitachiThreadDump
```

-Xrs オプションが指定されていない場合，スレッドダンプ出力時に拡張スレッドダンプ情報を出力します。

```
-XX:-HitachiThreadDump
```

スレッドダンプ出力時に標準のスレッドダンプ情報を出力します。

説明

拡張スレッドダンプ情報を出力するかどうかを指定します。

スレッドダンプは，標準出力，および次に示すファイルに出力されます。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

スレッドダンプ出力先

デフォルトの出力先は，JavaVM 実行時のカレントディレクトリです。環境変数 JAVACOREDIR を指定することで，出力先を変更することができます。

スレッドダンプ出力ファイル名

```
javacore<プロセス番号>.<YYMMDDhhmmss>※.txt
```

注※

YY：年（西暦で下2桁），MM：月（2桁），DD：日（2桁）

hh：時間（24時間表記），mm：分（2桁），ss：秒（2桁）

スレッドダンプ情報の構成を，次に示します。

表 5-2　スレッドダンプ情報の構成

出力情報	内容
ヘッダ	スレッドダンプを開始した日付および時刻，JavaVM バージョン情報，起動コマンドラインを出力します。
システム設定	次の情報を出力します。 • JDK の実行環境のインストールディレクトリ • JDK を構成するライブラリのインストールディレクトリ • システムクラスパス • Java コマンドオプション
動作環境	次の情報を出力します。 • ホスト名 • OS バージョン • CPU 情報 • リソース情報
Java ヒープ情報	Java ヒープの各世代のメモリ使用状況を出力します。
JavaVM 内部メモリマップ情報	JavaVM 自身の確保しているメモリの領域情報を出力します。
JavaVM 内部メモリサイズ情報	JavaVM 自身の確保しているメモリのサイズ情報を出力します。
アプリケーション情報	次の情報を出力します。 • シグナルハンドラ • 環境変数

出力情報	内容
	• カレントディレクトリ情報
ライブラリ情報	ローディングされているライブラリの情報を出力します。
スレッド情報 <スレッド 1> : <スレッド n>	スレッドごとにスレッド情報を出力します。現存する全スレッドのスタックトレースを出力します。
Java モニタダンプ※	Java モニタオブジェクトの一覧を表示します。スレッド間の排他待ちの状況を確認できます。
JNI グローバル参照情報	JavaVM が保持している JNI のグローバル参照の数を出力します。 JNI グローバル参照は，次の場合に作成されます。 • JavaVM が自身の起動や実行のために必要な場合。 • JNI がサポートする NewGlobalRef 関数を発行した場合。
フッタ	スレッドダンプを終了した日付および時刻を出力します。

注※　notify 待ちの一覧が表示されない場合があります。

出力形式

ヘッダ

```
EEE MMM dd hh:mm:ss yyyy※

Full thread dump Java HotSpot(TM) <VM type>※ (25.20-b23-CDK0970-<ビルド日> mixed mode)
<起動コマンドライン>
…
```

注※

EEE は曜日，MMM は月，dd は日を表します。また，hh は時間，mm は分，ss は秒，yyyy は年（西暦）を表します。

<VM type>は Client VM，Server VM または 64-Bit Server VM のどれかを表します。

システム設定

```
System Properties
-----------------
Java Home Dir   : <JDK実行環境インストールディレクトリ>
Java DLL Dir    : <JDKを構成するライブラリのインストールディレクトリ>
Sys Classpath   : <システムクラスパス>
User Args       :
<Javaコマンドオプション1>
<Javaコマンドオプション2>
…
```

動作環境

```
Operating Environment
---------------------
```

```
Host    : <ホスト名>:<IPアドレス>
OS      : <OSバージョン>
CPU     : <CPU種別>, <利用可能CPU数/システム全体のCPU数>

Resource Limits -
         RLIMIT_CPU     :<プロセスで使用可能な秒数>
         RLIMIT_FSIZE   :<最大ファイルサイズ(単位：バイト)>
         RLIMIT_DATA    :<malloc可能なサイズ(単位：バイト)>
         RLIMIT_STACK   :<スタックの最大サイズ(単位：バイト)>
         RLIMIT_CORE    :<coreの最大サイズ(単位：バイト)>
         RLIMIT_RSS     :<プロセスの常駐サイズ(単位：バイト)>
         RLIMIT_AS      :<プロセストータルの利用可能メモリ(単位：バイト)>
         RLIMIT_NOFILE  :<最大のファイルディスクリプタ値>
```

Java ヒープ情報（SerialGC を使用している場合）

```
Heap Status
-----------
def new generation   max <最大の容量>, total <現在の容量>, used <使用中メモリ> (<最大の容量に対する使用率>% used/max, <現在の容量に対する使用率>% used/total)
                     [<領域先頭アドレス>, <コミット済み領域の末尾アドレス>, <予約済み領域の末尾アドレス>)
  eden space <現在の容量>,  <使用率>% used [<領域先頭アドレス>, <使用中領域の先頭アドレス>, <予約済み領域の末尾アドレス>)
  from space <現在の容量>,   <使用率>% used [<領域先頭アドレス>, <使用中領域の先頭アドレス>, <予約済み領域の末尾アドレス>)
  to   space <現在の容量>,   <使用率>% used [<領域先頭アドレス>, <使用中領域の先頭アドレス>, <予約済み領域の末尾アドレス>)
tenured generation   max <最大の容量>, total <現在の容量>, used <使用中メモリ> (<最大の容量に対する使用率>% used/max, <現在の容量に対する使用率>% used/total)
                     [<領域先頭アドレス>, <コミット済み領域の末尾アドレス>, <予約済み領域の末尾アドレス>)
   the space <現在の容量>,  <使用率>% used [<領域先頭アドレス>, <使用中領域の先頭アドレス>, <次の空きブロックの先頭アドレス>, <予約済み領域の末尾アドレス>)
Metaspace       max<最大の容量>, capacity <コミット済みの領域からフリー領域を除いたメモリサイズ>, committed <コミット済みのメモリサイズ>, reserved <予約済みのメモリサイズ>, used <使用中メモリ> (<最大の容量に対する使用率>% used/max, <現在の容量に対する使用率>% used/total)
  class space    max <最大の容量>, capacity <コミット済みの領域からフリー領域を除いたメモリサイズ>, committed <コミット済みのメモリサイズ>, reserved <予約済みのメモリサイズ>, used <使用中メモリ> (<最大の容量に対する使用率>% used/max, <現在の容量に対する使用率>% used/committed)
                     [<領域先頭アドレス>, <使用中領域の先頭アドレス>, <コミット済み領域の末尾アドレス>, <予約済み領域の末尾アドレス>)
```

注

容量およびメモリサイズの単位はキロバイトです。

Java ヒープ情報（G1GC を使用している場合）

```
Heap Status
-----------
garbage-first heap   total <現在の容量>, used <使用中のメモリ> [<領域先頭アドレス>, <コミット済みの末尾アドレス>, <予約済み領域のアドレス>)
  region size <1リージョンのサイズ>, <New領域の使用中リージョン数> young (<New領域の使用中メモリ>), <Survivor領域の使用中リージョン数> survivors (<Survivor領域の使用中メモリ>)
```

```
 Metaspace       max <最大の容量>, capacity <コミット済みの領域からフリー領域を除いたメモリサイズ>, committed <コミット済みのメモリサイズ>, reserved <予約済みのメモリサイズ>, used <使用中メモリ> (<最大の容量に対する使用率>% used/max, <現在の容量に対する使用率>% used/committed)
  class space    max <最大の容量>, capacity <コミット済みの領域からフリー領域を除いたメモリサイズ>, committed <コミット済みのメモリサイズ>, reserved <予約済みのメモリサイズ>, used <使用中メモリ> (<最大の容量に対する使用率>% used/max, <現在の容量に対する使用率>% used/committed)
                  [<領域先頭アドレス>, <使用中領域の先頭アドレス>, <コミット済み領域の末尾アドレス>, <予約済み領域の末尾アドレス>)
```

JavaVM 内部メモリマップ情報

```
JVM Internal Memory Map
-----------------------
<メモリ確保関数>:address = <開始アドレス> - <終了アドレス> (size:<サイズ>)
```

注

<メモリ確保関数>：mmap()かmalloc()のどちらかが出力されます。

<開始アドレス>：メモリ領域の開始アドレスが 16 進で出力されます。

<終了アドレス>：メモリ領域の終了アドレスが 16 進で出力されます。

<サイズ>：確保しているメモリ領域のサイズが出力されます（単位：バイト）。

JavaVM 内部メモリサイズ情報

```
JVM Internal Memory Status
--------------------------
  Heap Size      :<確保しているメモリサイズ>※
  Alloc Size     :<使用中のメモリサイズ>※
  Free Size      :<未使用のメモリサイズ>※
```

注※

単位：バイト

アプリケーション環境

```
Application Environment
-----------------------
Signal Handlers -※1
  SIGHUP         :<シグナルハンドラ情報>
  SIGINT         :<シグナルハンドラ情報>
…
  SIGSOUND       :<シグナルハンドラ情報>
  SIGSAK         :<シグナルハンドラ情報>

Signal Handlers -※2
シグナル種別： [シグナルハンドラアドレス], sa_mask[0]=シグナルマスク, sa_flags=特殊フラグ
…

Environment Variables -
```

```
<環境変数>=<値>
…
Current Directory -
  /opt/Cosminexus/CC/server/…
```

注※1

次の情報が表示されます。

- シグナルハンドラがインストールされている場合は，そのアドレス。
- SIG_DFL である場合は，default。
- SIG_IGN である場合は，ignored。

注※2

次の情報が表示されます。

- シグナル種別には，/usr/include/sys/signal.h に定義されているシグナル名。
- シグナルハンドラアドレスには，シグナルハンドラのアドレスが 16 進数で出力されます。ライブラリ名＋オフセットという形式で表示されることもあります。
- シグナルマスクには，sigaction()で取り出せる構造の sa_mask フィールド値が 16 進数で出力されます。
- 特殊フラグには，sigaction()で取り出せる構造の sa_flags フィールド値が 16 進数で出力されます。

ライブラリ情報

```
Loaded Libraries
----------------
Dynamic libraries :
<開始アドレス>-<終了アドレス>   <コマンド>
<開始アドレス>-<終了アドレス>   <ライブラリ>
…
```

スレッド情報

```
-----------
"<スレッド名>" <daemon> prio=<優先度> jid=<ハッシュ値> tid=<スレッドID> nid=<nativeID> <s
tatus> [<開始アドレス>...<終了アドレス>]
    java.lang.Thread.State: <スレッドの現在のステータス>
    stack=[<スタック開始アドレス>..<YellowPageアドレス>..<RedPageアドレス>..<スタック終了
アドレス>]
    [user cpu time=<ユーザー時間>ms, kernel cpu time=<カーネル時間>ms] [blocked count=<ブ
ロック回数>, waited count=<待機回数>]
         at <クラス名>.<メソッド名>(<メソッド情報>)
…
```

出力内容を説明します。

<スレッド名>

Thread クラスのコンストラクタに指定されたスレッド名称が出力されます。

- <daemon>：デーモンスレッドである場合に，"daemon"と出力されます。
- <優先度>：Thread#setPriority で設定された優先度が出力されます。

- <ハッシュ値>：System.identityHashCode()を呼び出して得られる値と同一の値が 8桁の 16 進数で出力されます。
- <スレッド ID>：スレッドオブジェクトのメモリ上のアドレス。
- <nativeID>：OS レベルのスレッド ID。
- <status>：スレッドの状態。

  runnable：実行中または実行可能なスレッド

  in Object.wait(), waiting for monitor entry またはwaiting on condition：モニタロック待ちのスレッド

  sleeping：中断状態のスレッド
- <開始アドレス>：Java フレームの最高位スタックアドレスが 16 進数で出力されます。
- <終了アドレス>：JavaLock のある最高位スタックアドレスが 16 進数で出力されます。
- <ユーザー時間>：スレッド開始からのユーザー時間がミリ秒単位で出力されます。
- <カーネル時間>：スレッド開始からのカーネル時間がミリ秒単位で出力されます。
- <ブロック回数>：スレッド開始から，処理がブロックされた回数が出力されます。
- <待機回数>：スレッド開始から，処理が待ち状態になった回数が出力されます。

<スレッドの現在のステータス>

スレッドの現在のステータスを表すメッセージが出力されます。メッセージの内容は java.lang.Thread.State 列挙型に対応します。

<スタック開始アドレス>

スタック開始アドレスが 16 進数で出力されます。

<YellowPage アドレス>

スタック Yellow ガードページ先頭アドレスが 16 進数で出力されます。

<RedPage アドレス>

スタック Red ガードページ先頭アドレスが 16 進数で出力されます。

<スタック終了アドレス>

スタック終了アドレスが 16 進数で出力されます。

<クラス名>

クラス名が出力されます。

<メソッド名>

メソッド名が出力されます。

<メソッド情報>

次のメソッド情報が出力されます。

- Native Method

  ネイティブメソッドの場合に出力されます。

- ファイル名：行番号
  Java メソッドで行番号付きでコンパイルされている場合に出力されます。
- Unknown Source
  Java メソッドで行番号なしでコンパイルされている場合に出力されます。

Java モニタダンプ

```
Java monitor
------------
<ロックオブジェクト>@<ハッシュコード> <オーナー情報>
   <待機状態>:<待機スレッド数>
      <待機スレッド情報>
```

出力内容を説明します。

<ロックオブジェクト>

ロック対象オブジェクトのクラス名が出力されます。

<ハッシュコード>

Object.hashCode で得られるハッシュコードが出力されます。

<オーナー情報>

- owner "<スレッド名>"<スレッド ID>
  オーナーがある場合に出力されます。
- no owner
  オーナーがない場合に出力されます。

<待機状態>

- ... waiting to enter
  メソッド実行待ちの場合に出力されます。
- ... waiting to be notified
  通知待ちの場合に出力されます。

<待機スレッド数>

スレッド数が出力されます。

<待機スレッド情報>

「"<スレッド名>" <スレッド ID>」の形式で出力されます。

JNI グローバル参照数の情報

```
JNI Information
---------------
JNI global references: <JNIグローバル参照数>
```

出力内容を説明します。

<JNI グローバル参照数>

JavaVM が保持しているグローバル参照の数が出力されます。

注

JNI グローバル参照は JavaVM の内部でも再利用されるため，JNI がサポートする DeleteGlobalRef 関数を発行して JNI グローバル参照を削除しても，数値は減少しません。また，NewGlobalRef 関数を発行して JNI グローバル参照を新規作成しても，JavaVM が再利用した JNI グローバル参照を割り当てた場合は数値は増加しません。

拡張スレッドダンプ情報との比較を，次に示します。

表 5-3　標準スレッドダンプと拡張スレッドダンプの出力情報の比較

出力情報	標準スレッドダンプ	拡張スレッドダンプ
ヘッダ	×	○
システム設定	×	○
動作環境	×	○
Java ヒープ情報	×	○
JavaVM 内部メモリマップ情報	×	○
JavaVM 内部メモリサイズ情報	×	○
アプリケーション環境	×	○
ライブラリ情報	×	○
スレッド情報	○	○※1
Java モニタダンプ	×	○
JNI グローバル参照数の情報	○	○
フッタ	×	○
スレッドダンプ出力先	標準出力	標準出力※2 JavaVM ログファイル

（凡例）

○：出力されます。

×：出力されません。

注※1

スタックの開始および終了のアドレス情報などが出力されます。

注※2

-XX:+HitachiThreadDumpToStdout オプションが指定された場合に出力されます。

注意事項

- 環境変数 JAVACOREDIR で指定したディレクトリへの出力に失敗した場合，カレントディレクトリに出力されます。
- カレントディレクトリへの出力に失敗した場合，標準エラー出力に出力されます。なお，この場合，スレッドダンプは標準出力に出力されません。

- 次に示すオプションの[+|-]指定が「-」の場合，スレッド情報の一部が出力されないのでご注意ください。

オプション名称	出力されない情報
-XX:[+|-]HitachiThreadDumpWithHashCode	<ハッシュ値>
-XX:[+|-]HitachiThreadDumpWithCpuTime	<ユーザー時間>，<カーネル時間>
-XX:[+|-]HitachiThreadDumpWithBlockCount	<ブロック回数>，<待機回数>

-XX:[+|-]HitachiThreadDumpToStdout（拡張スレッドダンプ標準出力抑止オプション）

形式

`-XX:+HitachiThreadDumpToStdout`

拡張スレッドダンプを標準出力およびスレッドダンプ出力ファイルに出力します。

`-XX:-HitachiThreadDumpToStdout`

拡張スレッドダンプを標準出力に出力しません。スレッドダンプ出力ファイルだけに出力します。

説明

拡張スレッドダンプを標準出力へ出力するかどうかを指定します。

このオプションの指定に関係なく，次のメッセージは出力されます。また，拡張スレッドダンプは JavaVM ログファイルへ出力されます。

```
Writing Java core to <ファイル名（フルパス）>... OK
```

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- `-XX:+HitachiThreadDump`

-XX:[+|-]HitachiThreadDumpWithHashCode（拡張スレッドダンプハッシュコード出力オプション）

形式

-XX:+HitachiThreadDumpWithHashCode

拡張スレッドダンプのスレッド情報にハッシュコードを出力します。

-XX:-HitachiThreadDumpWithHashCode

拡張スレッドダンプのスレッド情報にハッシュコードを出力しません。

説明

拡張スレッドダンプのスレッド情報にハッシュコードを出力するかどうかを指定します。

なお，ハッシュコードは，Java プログラムを実行しているスレッドに対して出力されます。JavaVM の内部動作用スレッドに対しては出力されません。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- -XX:+HitachiThreadDump

-XX:[+|-]HitachiThreadDumpWithCpuTime（拡張スレッドダンプ CPU 利用時間出力オプション）

形式

-XX:+HitachiThreadDumpWithCpuTime

拡張スレッドダンプのスレッド情報に，スレッド開始からのユーザー CPU 時間とカーネル CPU 時間を出力します。

-XX:-HitachiThreadDumpWithCpuTime

拡張スレッドダンプのスレッド情報に，スレッド開始からのユーザー CPU 時間とカーネル CPU 時間を出力しません。

説明

拡張スレッドダンプのスレッド情報に，ユーザー CPU 時間とカーネル CPU 時間を出力するかどうかを指定します。

なお，ユーザー CPU 時間とカーネル CPU 時間は，Java プログラムを実行しているスレッドに対して出力されます。JavaVM の内部動作用スレッドに対しては出力されません。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- -XX:+HitachiThreadDump

-XX:[+|-]HitachiThreadDumpWithBlockCount（拡張スレッドダンプブロック回数出力オプション）

形式

-XX:+HitachiThreadDumpWithBlockCount

拡張スレッドダンプのスレッド情報に，スレッドが処理をブロックした回数と待ち状態になった回数を出力します。

-XX:-HitachiThreadDumpWithBlockCount

拡張スレッドダンプのスレッド情報に，スレッドが処理をブロックした回数と待ち状態になった回数を出力しません。

説明

拡張スレッドダンプのスレッド情報に，スレッドが処理をブロックした回数と待ち状態になった回数を出力するかどうかを指定します。

なお，ハッシュコードは，Java プログラムを実行しているスレッドに対して出力されます。JavaVM の内部動作用スレッドに対しては出力されません。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- -XX:+HitachiThreadDump

-XX:HitachiJavaLog（ログファイル名のプリフィックス指定オプション）

形式

```
-XX:HitachiJavaLog:<文字列>
```

説明

JavaVM ログファイルのプリフィックスおよびログファイルの出力先ディレクトリを指定します。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

次のどれかを指定します。

- -XX:+HitachiVerboseGC
- -XX:+HitachiOutOfMemoryStackTrace
- -XX:+HitachiOutOfMemoryCause
- -XX:+HitachiOutOfMemorySize
- -XX:+HitachiJavaClassLibTrace
- -XX:+JITCompilerContinuation

引数

<文字列>

プリフィックスおよびパスを指定します。次の 3 種類の指定ができます。

プリフィックスを指定する場合

ログファイル名は，<文字列>??.log（??は 01～99 の通し番号）で生成されます。例えば，<文字列>に"Samp"を指定すると，ログファイル名は Samp01.log になります。このオプションを指定しない場合，<文字列>には，"javalog"が設定されます。また，ログファイルはカレントディレクトリに出力されます。

パスを指定する場合

<文字列>にディレクトリを指定した場合，そのディレクトリにファイルが作成されます。ログファイル名は，<文字列>javalog??.log（??は 01～99 の通し番号）で生成されます。

パスとプリフィックスを同時に指定する場合

<文字列>にディレクトリとプリフィックスを指定した場合，そのディレクトリにファイルが作成されます。ログファイル名は，<文字列>??.log（??は 01～99 の通し番号）で生成されます。例えば，<文字列>に"d:¥temp¥Samp"を指定すると，d:¥temp ディレクトリに，Samp01.log が生成されます。

-XX:HitachiJavaLogFileSize（最大ログファイルサイズ指定オプション）

形式

```
-XX:HitachiJavaLogFileSize=<整数値>
```

説明

ログファイルの単純増加を防ぐため，1 ファイルの最大ファイルサイズを指定します。最大ファイルサイズを超えた場合は，そのファイルへの出力は行いません。オプションの指定がない場合は，256 キロバイトが設定されます。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

次のどれかを指定します。

- -XX:+HitachiVerboseGC
- -XX:+HitachiOutOfMemoryStackTrace
- -XX:+HitachiOutOfMemoryCause
- -XX:+HitachiOutOfMemorySize
- -XX:+HitachiJavaClassLibTrace
- -XX:+JITCompilerContinuation

引数

<整数値>

1024～2147483647 の範囲で整数値（単位：バイト）を指定します。範囲外の値が指定された場合は 1024 が設定されます。負の値を指定した場合はエラーとなります。

-XX:[+|-]HitachiJavaLogNoMoreOutput（ログファイル入出力エラー発生時指定オプション）

形式

-XX:+HitachiJavaLogNoMoreOutput

ログファイル出力時にファイル入出力エラーが発生した場合，次のメッセージを標準エラー出力に出力し，ログ情報の出力を停止します。

```
Java logfile output failed.(errno=<エラーとなった入出力関数名>:<エラー番号>)
```

-XX:-HitachiJavaLogNoMoreOutput

ログファイル出力時にファイル入出力エラーが発生した場合，次のメッセージを標準エラー出力に出力し，ログ情報の出力先を標準エラー出力に変更して出力を継続します。

```
Java logfile output failed.(errno=<エラーとなった入出力関数名>:<エラー番号>) Changing output to stderr
```

説明

ログファイル作成時に入出力エラーが発生した場合に，ログ情報の出力方法について指定します。なお，どちらを指定した場合も，JavaVM の処理は継続されます。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

次のどれかを指定します。

- -XX:+HitachiVerboseGC
- -XX:+HitachiOutOfMemoryStackTrace
- -XX:+HitachiOutOfMemoryCause
- -XX:+HitachiOutOfMemorySize
- -XX:+HitachiJavaClassLibTrace
- -XX:+JITCompilerContinuation

-XX:HitachiJavaLogNumberOfFile（最大ログファイル数指定オプション）

形式

```
-XX:HitachiJavaLogNumberOfFile=<整数値>
```

説明

ログファイルの単純増加を防ぐため，作成する最大ファイル数を指定します。最大ファイル数を超えた場合は，再度最初に作成したファイルへ出力を開始します。オプションの指定がない場合は，４が設定されます。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

次のどれかを指定します。

- -XX:+HitachiVerboseGC
- -XX:+HitachiOutOfMemoryStackTrace
- -XX:+HitachiOutOfMemoryCause
- -XX:+HitachiOutOfMemorySize
- -XX:+HitachiJavaClassLibTrace

引数

<整数値>

1～99 の範囲で指定します。100 以上の値が指定された場合は 99 が，0 以下の値が指定された場合は 1 が設定されます。負の値を指定した場合はエラーとなります。

-XX:[+|-]JavaLogAsynchronous

形式

-XX:+JavaLogAsynchronous

ログファイルの非同期出力機能を有効にします。

-XX:-JavaLogAsynchronous

ログファイルの非同期出力機能を無効にします。

説明

ログファイルの非同期出力機能の有効，無効を指定します。

ログファイルの非同期出力機能使用時，-XX:HitachiOutOfMemoryStackTraceLineSize オプション，または-XX:HitachiJavaClassLibTraceLineSize オプションに，4096 より大きな値を指定した場合は，出力するスタックトレース 1 行の文字数に 4096 バイトが指定されたものとして動作します。

指定したバイト数が確保できない場合は警告メッセージが出力され，スタックトレースは出力されません。また，1 行の文字数が指定した文字数を超えた場合，「at」以降の文字列の前半部分を削除して，指定された文字数分出力します。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

-XX:[+|-]StandardLogToHitachiJavaLog（標準出力のログファイル出力オプション）

形式

-XX:+StandardLogToHitachiJavaLog

JavaVM の標準出力と標準エラー出力の内容を JavaVM ログファイルに出力します。

-XX:-StandardLogToHitachiJavaLog

JavaVM の標準出力と標準エラー出力の内容を JavaVM ログファイルに出力しません。日立 JavaVM を起動したコンソールだけに出力します。

説明

この機能を有効にした場合，JavaVM の標準出力と標準エラー出力の内容を JavaVM ログファイルにもあわせて出力します。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

出力形式

```
[id] <date> stdout
```

出力内容を次に説明します。

id

識別子。

標準出力の場合は[STO]が出力され，標準エラー出力の場合は[STE]が出力されます。

date

ログ出力開始時間。

-XX:+HitachiOutputMilliTime オプションが指定されている場合は，ミリ秒単位まで出力されます。

stdout

標準出力または標準エラー出力のメッセージ内容です。

出力されたメッセージの内容が次の場合，上記の出力形式とは異なる形式で出力されます。

- 出力されたメッセージ文字列が 2,000 バイトを超えていて，かつ文字列の最後に改行（¥n）がある場合，次の行に空行が出力されます。
- 出力対象のメッセージ文字列の途中に改行がある場合，改行したあとのメッセージには識別子やログ出力開始時間は出力されません。

出力例

```
[STO]<Fri Jul 19 11:35:33 2019>VM option '+StandardLogToHitachiJavaLog'
[STO]<Fri Jul 19 11:35:33 2019>VM option '+PrintVMOptions'
[STE]<Fri Jul 19 11:35:33 2019>Invalid initial young generation size: -Xmn0k
```

注意事項

JavaVM ログファイル出力時にエラーが発生した場合，標準エラー出力に出力されるエラーメッセージは，JavaVM ログファイルには出力されません。

-XX:[+|-]HitachiOutputMilliTime（詳細時間出力オプション）

形式

-XX:+HitachiOutputMilliTime

JavaVM ログファイルに出力する日時に，ミリ秒まで出力します。

-XX:-HitachiOutputMilliTime

JavaVM ログファイルに出力する日時に，秒まで出力します。

説明

ミリ秒までの時間を出力するかどうかを指定します。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

次のどれかを指定します。

- -XX:+HitachiVerboseGC
- -XX:+HitachiOutOfMemoryStackTrace
- -XX:+HitachiOutOfMemoryCause
- -XX:+HitachiOutOfMemorySize
- -XX:+HitachiJavaClassLibTrace
- -XX:+JITCompilerContinuation

出力例

- 拡張 verbosegc 情報の出力

```
[VGC]<Wed Mar 17 00:45:55.068 2004>(Skip Full:0,Copy:0)[Full GC 149K->149K(1984K), 0.0786
038 secs][DefNew::Eden: 264K->0K(512K)][DefNew::Survivor: 0K->63K(64K)][Tenured: 85K->149
K(1408K)][Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 38
8K)->356K(388K, 388K)]
```

- OutOfMemoryError 発生時の出力

```
[OOM][Thread: 0x00957820]<Wed Mar 17 00:47:00.662 2004>[java.lang.OutOfMemory Error :(C H
eap) :340]
```

- クラスライブラリトレースの出力

```
[CLT][Thread: 0x00286348]<Wed Mar 17 00:47:00.662 2004>
[CLT][Thread: 0x00286348] at java.lang.Shutdown.halt0(Native Method)
[CLT][Thread: 0x00286348] at java.lang.Shutdown.halt(Shutdown.java:145)
```

- JIT コンパイラ稼働継続機能（JIT コンパイル失敗情報）

```
[JCC][Thread: 0x05432c00]<Thu Nov 15 17:10:40.347 2012>[Method: chosa_cmp.func(Ljava/lang
/String;)V][Fail: 3][JITCT: 1]
```

-XX:[+|-]HitachiVerboseGC（拡張 verbosegc 情報出力オプション）

形式

-XX:+HitachiVerboseGC

GC が発生した場合，拡張 verbosegc 情報を JavaVM ログファイルに出力します。

GC の内部領域である Eden，Survivor，Tenured，Metaspace 種別の情報を拡張 verbosegc 情報として出力します。

-XX:-HitachiVerboseGC

GC が発生した場合，拡張 verbosegc 情報を JavaVM ログファイルに出力しません。

説明

GC が発生したときの拡張 verbosegc 情報を出力するかどうかを指定します。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

出力形式（SerialGC を使用している場合）

```
[id] <date> (Skip Full:full_count, Copy:copy_count) [gc_kind gc_info, gc_time secs][Eden: ed
en_info][Survivor: survivor_info][Tenured: tenured_info][Metaspace: metaspace_info][class sp
ace: class_space_info][cause:cause_info] [User: user_cpu secs] [Sys: system_cpu secs][IM: jv
m_alloc_size, mmap_total_size, malloc_total_size][TC: thread_count][DOE: doe_alloc_size, cal
led_count][CCI: cc_used_sizeK, cc_max_sizeK, cc_infoK]
```

出力内容を次に説明します。

id

VGC（JavaVM ログファイル識別子）。

date

GC 開始日時。

-XX:-HitachiVerboseGCPrintDate オプションが指定された場合，出力されません。

full_count

full GC 情報出力をスキップした回数。

-XX:HitachiVerboseGCIntervalTime オプションが指定された場合に出力されます。

copy_count

copy GC 情報出力をスキップした回数。

-XX:HitachiVerboseGCIntervalTime オプションが指定された場合に出力されます。

gc_kind

GC 種別。”FullGC”，”GC”が出力されます。

gc_info

GC 情報。次の形式で出力されます。

```
<GC前の領域長> -> <GC後の領域長>（<領域サイズ>）
```

gc_time

GC 経過時間。

Eden

エデンの種別。”DefNew::Eden”または”ParNew::Eden”が出力されます。

eden_info

Eden 情報。次の形式で出力されます。

```
<GC前の領域長> -> <GC後の領域長>（<領域サイズ>）
```

Survivor

Survivor の種別。”DefNew:: Survivor”または”ParNew::Survivor”が出力されます。

survivor_info

Survivor 情報。次の形式で出力されます。

```
<GC前の領域長> -> <GC後の領域長>（<領域サイズ>）
```

Tenured

Tenured の種別。”Tenured”が出力されます。

tenured_info

Tenured 情報。次の形式で出力されます。

```
<GC前の領域長> -> <GC後の領域長>（<領域サイズ>）
```

metaspace_info

Metaspace 領域の情報。次の形式で出力されます。

```
<GC前の使用サイズ>(<GC前のcapacityサイズ>, <GC前のcommitサイズ>) -> <GC後の使用サイズ>(<GC後のcapacityサイズ>, <GC後のcommitサイズ>)
```

class_space_info

Compressed Class Space の情報。次の形式で出力されます。

```
<GC前の使用サイズ>(<GC前のcapacityサイズ>, <GC前のcommitサイズ>) -> <GC後の使用サイズ>(<GC後のcapacityサイズ>, <GC後のcommitサイズ>)
```

-XX:-UseCompressedOops オプションが指定された場合，出力されません。

cause_info

GC 要因内容。

-XX:-HitachiVerboseGCPrintCause オプションが指定された場合，出力されません。

user_cpu

GC スレッドがユーザーモードで費やした CPU 時間。単位は秒です。

CPU 時間取得に失敗した場合，[User: unknown]のように，”unknown”と表示されます。

-XX:-HitachiVerboseGCCpuTime オプションが指定された場合，出力されません。

system_cpu

GC スレッドがカーネルモードで費やした CPU 時間。単位は秒です。

CPU 時間取得に失敗した場合，[Sys: unknown]のように，”unknown”と表示されます。

-XX:-HitachiVerboseGCCpuTime オプションが指定された場合，出力されません。

jvm_alloc_size

JavaVM 内部で管理している領域のうち，現在使用中の領域のサイズ（mmap_total_size と malloc_total_size の合計サイズのうち，現在使用中の領域のサイズ）。

-XX:-HitachiVerboseGCPrintJVMInternalMemory オプションが指定された場合，出力されません。

mmap_total_size

JavaVM 内部で管理している領域のうち，mmap で割り当てた C ヒープの総サイズ。

-XX:-HitachiVerboseGCPrintJVMInternalMemory オプションが指定された場合，出力されません。

malloc_total_size

JavaVM 内部で管理している領域のうち，malloc で割り当てた C ヒープの総サイズ。

-XX:-HitachiVerboseGCPrintJVMInternalMemory オプションが指定された場合，出力されません。

thread_count

Java スレッドの数。

-XX:-HitachiVerboseGCPrintThreadCount オプションが指定された場合，出力されません。

doe_alloc_size

java.io.File.deleteOnExit()を呼び出して確保した累積のヒープサイズ。

-XX:-HitachiVerboseGCPrintDeleteOnExit オプションが指定された場合，出力されません。

called_count

java.io.File.deleteOnExit()の呼び出し回数。

-XX:-HitachiVerboseGCPrintDeleteOnExit オプションが指定された場合，出力されません。

cc_used_size

GC 発生時のコードキャッシュ領域の使用サイズ。単位はキロバイトです。

-XX:-PrintCodeCacheInfo オプションが指定された場合，出力されません。

cc_max_size

コードキャッシュ領域の最大サイズ。単位はキロバイトです。

-XX:-PrintCodeCacheInfo オプションが指定された場合，出力されません。

cc_info

保守情報。

-XX:-PrintCodeCacheInfo オプションが指定された場合，出力されません。

出力形式（G1GC を使用している場合）

- VG1 ログ

```
[id]<date>[gc_kind gc_info, gc_time secs][Status: gc_status][G1GC:Eden: eden_info][G1GC:S
urvivor: survivor_info][G1GC:Tenured: tenured_info][G1GC:Humongous: humongous_info][G1GC:
Free: free_info][Metaspace: metaspace_info][class space: class_space_info][cause:cause_in
fo][RegionSize: region_size][Target: target_time secs][Predicted: predicted_time secs][Ta
rgetTenured: target_size][Reclaimable: reclaimable_info][User: user_cpu secs] [Sys: syste
m_cpu secs][IM: jvm_alloc_size, mmap_total_size, malloc_total_size][TC: thread_count][DOE
: doe_alloc_size, called_count][CCI: cc_used_sizeK, cc_max_sizeK, cc_infoK]
```

出力内容を次に説明します。

id

VG1（JavaVM ログファイル識別子）。

date

GC 開始日時。

-XX:-HitachiVerboseGCPrintDate オプションが指定された場合，出力されません。

gc_kind

GC 種別。“Full GC”，“Mixed GC”，“Young GC”，“Young GC(initial-mark)”，“CM Remark”，“CM Cleanup”のどれかが出力されます。

gc_info

GC 情報。次の形式で出力されます。

```
<GC前の領域サイズ>/<GC前の領域サイズ(リージョン換算)>(<GC前の領域サイズ>) -> <GC後の領域
サイズ>/<GC前の領域サイズ(リージョン換算)>(<GC後の領域サイズ>)
```

リージョン換算とは領域サイズを 1 リージョンのサイズで切り上げ，1 リージョンのサイズの倍数で表した値です。

gc_time

GC 経過時間。

gc_status

GC の状態。“-”，“to exhausted”が出力されます。

eden_info

Eden 情報。次の形式で出力されます。

```
<GC前の領域サイズ(リージョン換算)>(<GC前の最大領域サイズ(リージョン換算)>) -> <GC後の領域
サイズ(リージョン換算)>(<GC後の最大領域サイズ(リージョン換算)>)
```

リージョン換算とは領域サイズを 1 リージョンのサイズで切り上げ，1 リージョンのサイズの倍数で表した値です。

survivor_info

Survivor 情報。次の形式で出力されます。

```
<GC前の領域サイズ(リージョン換算)> -> <GC後の領域サイズ(リージョン換算)>
```

リージョン換算とは領域サイズを 1 リージョンのサイズで切り上げ，1 リージョンのサイズの倍数で表した値です。

tenured_info

Tenured 情報。次の形式で出力されます。

```
<GC前の領域サイズ(リージョン換算)> -> <GC後の領域サイズ(リージョン換算)>
```

リージョン換算とは領域サイズを 1 リージョンのサイズで切り上げ，1 リージョンのサイズの倍数で表した値です。

humongous_info

Humongous 情報。次の形式で出力されます。

```
<GC前の領域サイズ(リージョン換算)> -> <GC後の領域サイズ(リージョン換算)>
```

リージョン換算とは領域サイズを 1 リージョンのサイズで切り上げ，1 リージョンのサイズの倍数で表した値です。

free_info

Free 情報。次の形式で出力されます。

```
<GC前の領域サイズ(リージョン換算)> -> <GC後の領域サイズ(リージョン換算)>
```

リージョン換算とは領域サイズを 1 リージョンのサイズで切り上げ，1 リージョンのサイズの倍数で表した値です。

metaspace_info

Metaspace 領域の情報。次の形式で出力されます。

```
<GC前の使用サイズ>(<GC前のcapacityサイズ>, <GC前のcommitサイズ>) -> <GC後の使用サイズ>(<GC後のcapacityサイズ>, <GC後のcommitサイズ>)
```

class_space_info

Compressed Class Space の情報。次の形式で出力されます。

```
<GC前の使用サイズ>(<GC前のcapacityサイズ>, <GC前のcommitサイズ>) -> <GC後の使用サイズ>(<GC後のcapacityサイズ>, <GC後のcommitサイズ>)
```

-XX:-UseCompressedOops オプションが指定された場合，出力されません。

cause_info

GC 要因内容。

-XX:-HitachiVerboseGCPrintCause オプションが指定された場合，出力されません。

region_size

1 リージョンのサイズです。

単位はキロバイトです。

target_time

GC によるアプリケーション停止時間の目標時間です。

単位は秒です。

predicted_time

JavaVM が予測した GC によるアプリケーション停止時間です。

単位は秒です。

なお，GC 種別が“Full GC”，“CM Remark”，“CM Cleanup”のときは予測をしないため，0 が出力されます。

target_size

Mixed GC で GC 対象となった Tenured 領域のサイズです。

単位はキロバイト。

なお，GC 種別が”Mixed GC”以外のときは，0 が出力されます。

reclaimable_info

Mixed GC で回収される Tenured 領域の予測回収サイズ情報。次の形式で出力されます。

〈予測回収サイズ〉（〈予測回収率〉）

なお，予測回収サイズ情報は CM 終了直後の Young GC または Mixed GC だけ出力されます。それ以外の場合，予測を行わないため，0 が出力されます。

user_cpu

GC スレッドがユーザーモードで費やした CPU 時間。単位は秒です。

CPU 時間取得に失敗した場合，[User: unknown]のように，"unknown"と表示されます。

-XX:-HitachiVerboseGCCpuTime オプションが指定された場合，出力されません。

system_cpu

GC スレッドがカーネルモードで費やした CPU 時間。単位は秒です。

CPU 時間取得に失敗した場合，[Sys: unknown]のように，"unknown"と表示されます。

-XX:-HitachiVerboseGCCpuTime オプションが指定された場合，出力されません。

jvm_alloc_size

JavaVM 内部で管理している領域のうち，現在使用中の領域のサイズ（mmap_total_size と malloc_total_size の合計サイズのうち，現在使用中の領域のサイズ）。

-XX:-HitachiVerboseGCPrintJVMInternalMemory オプションが指定された場合，出力されません。

mmap_total_size

JavaVM 内部で管理している領域のうち，mmap で割り当てた C ヒープの総サイズ。

-XX:-HitachiVerboseGCPrintJVMInternalMemory オプションが指定された場合，出力されません。

malloc_total_size

JavaVM 内部で管理している領域のうち，malloc で割り当てた C ヒープの総サイズ。

-XX:-HitachiVerboseGCPrintJVMInternalMemory オプションが指定された場合，出力されません。

thread_count

Java スレッドの数。

-XX:-HitachiVerboseGCPrintThreadCount オプションが指定された場合，出力されません。

doe_alloc_size

java.io.File.deleteOnExit()を呼び出して確保した累積のヒープサイズ。

-XX:-HitachiVerboseGCPrintDeleteOnExit オプションが指定された場合，出力されません。

called_count

java.io.File.deleteOnExit()の呼び出し回数。

-XX:-HitachiVerboseGCPrintDeleteOnExit オプションが指定された場合，出力されません。

cc_used_size

GC 発生時のコードキャッシュ領域の使用サイズ。単位はキロバイトです。

-XX:-PrintCodeCacheInfo オプションが指定された場合，出力されません。

cc_max_size

コードキャッシュ領域の最大サイズ。単位はキロバイトです。

-XX:-PrintCodeCacheInfo オプションが指定された場合，出力されません。

cc_info

保守情報。

-XX:-PrintCodeCacheInfo オプションが指定された場合，出力されません。

- VCM

```
[id]<date>[cm_event][User: user_cpu secs][Sys: sys_cpu secs]
```

id

VCM（JavaVM ログファイル識別子）。

date

CM 開始日時。

-XX:-HitachiVerboseGCPrintDate オプションが指定された場合，出力されません。

cm_event

CM 種別。”Concurrent Root Region Scan Start”，”Concurrent Root Region Scan End”，”Concurrent Mark Start”，”Concurrent Mark End”，”Concurrent Mark Stop”，”Concurrent Cleanup Start”，”Concurrent Cleanup End”のどれかが出力されます。

user_cpu

全 CM スレッドがユーザーモードで費やした CPU 時間。単位は秒です。

-XX:-HitachiVerboseGCCpuTime 指定時は出力されません。

CPU 時間取得に失敗した場合，[User: unknown]のように，"unknown"と表示されます。

CM の状態が Start の場合，0 が出力されます。

sys_cpu

全 CM スレッドがカーネルモードで費やした CPU 時間。単位は秒です。

-XX:-HitachiVerboseGCCpuTime 指定時は出力されません。

CPU 時間取得に失敗した場合，[Sys: unknown]のように，"unknown"と表示されます。

CM の状態が Start の場合，0 が出力されます。

出力例

SerialGC を使用している場合

- -XX:HitachiVerboseGCIntervalTime オプションが指定されている場合

```
[VGC]<Wed Mar 17 00:42:30 2004>(Skip Full:0,Copy:0)[Full GC 149K->149K(1984K), 0.0786038
secs][DefNew::Eden: 264K->0K(512K)][DefNew::Survivor: 0K->63K(64K)][Tenured: 85K->149K(14
08K)][Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)-
>356K(388K, 388K)][cause:System.gc][User: 0.0156250 secs][Sys: 0.0312500 secs][IM: 729K,
928K, 0K][TC: 509][DOE: 16K, 170][CCI: 2301K, 49152K, 2304K]
```

G1GC を使用している場合

- VG1 ログ

```
[VG1]<Thu Oct 02 10:38:56.193 2014>[Full GC 753K/2048K(8192K)->678K/1024K(8192K), 0.00979
01 secs][Status:-][G1GC::Eden: 1024K(2048K)->0K(2048K)][G1GC::Survivor: 0K->0K][G1GC::Ten
ured: 1024K->1024K][G1GC::Humongous: 0K->0K][G1GC::Free: 6144K->7168K] [Metaspace: 3634K(
4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)] [cau
se:System.gc][RegionSize: 1024K][Target: 0.2000000 secs][Predicted: 0.0000000 secs][Targe
tTenured: 0K][Reclaimable: 0K(0.00%)][User: 0.0000000 secs][Sys: 0.0000000 secs][IM: 2045
9K, 21920K, 0K][TC: 35][DOE: 0K, 0][CCI: 1172K, 245760K, 2496K]
```

- VCM ログ

```
[VCM]<Wed Jul 24 11:45:20 2013>[Concurrent Root Region Scan Start][User: 0.0000000 secs][
Sys: 0.0000000 secs]
[VCM]<Wed Jul 24 11:45:20 2013>[Concurrent Root Region Scan End][User: 0.0126134 secs][Sy
s: 0.0146961 secs]
[VCM]<Wed Jul 24 11:45:20 2013>[Concurrent Mark Start][User: 0.0000000 secs][Sys: 0.00000
00 secs]
[VCM]<Wed Jul 24 11:45:34 2013>[Concurrent Mark End][User: 0.0156250 secs][Sys: 0.2495800
 secs]
```

-XX:[+|-]HitachiCommaVerboseGC（CSV 出力オプション）

形式

-XX:+HitachiCommaVerboseGC

拡張 verbosegc 情報の出力を，CSV ファイルで取得できるようにコンマ形式で出力します。

拡張 verbosegc 情報に出力される括弧（丸括弧()，角括弧[]，山括弧<>）およびコロン（:）をすべて削除し，コンマ（,）で区切った数値または文字列を出力します。

-XX:-HitachiCommaVerboseGC

拡張 verbosegc 情報を通常形式で出力します。

説明

拡張 verbosegc 情報を CSV 形式で出力するかどうかを指定します。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- -XX:+HitachiVerboseGC

出力形式（SerialGC を使用している場合）

-XX:-HitachiVerboseGCIntervalTime オプションが指定されている場合の出力内容を次に説明します。

```
id, date, full_count, copy_count, inc_count, gc_kind, gc_info, gc_time, eden_info, survivor_
info,tenured_info, metaspace_info, classspace_info, cause_info, user_cpu, system_cpu, jvm_al
loc_size, mmap_total_size, malloc_total_size, thread_count, doe_alloc_size, called_count, cc
_used_size, cc_max_size, cc_info
```

id

JavaVM ログファイル識別子。

date

GC 開始日時。-XX:-HitachiVerboseGCPrintDate オプションが指定された場合，出力されません。

full_count

full GC 情報出力をスキップした回数。

-XX:HitachiVerboseGCIntervalTime オプションが指定された場合に出力されます。

copy_count

copy GC 情報出力をスキップした回数。

-XX:HitachiVerboseGCIntervalTime オプションが指定された場合に出力されます。

inc_count

0 を表示。

-XX:HitachiVerboseGCIntervalTime オプションが指定された場合に出力されます。

gc_kind

GC 種別。"FullGC"または"GC"が出力されます。

gc_info

GC 情報。次の形式で出力されます。単位はキロバイトです。

```
<GC前の領域長>,<GC後の領域長>,<領域サイズ>
```

gc_time

GC 経過時間。単位は秒です。

eden_info

Eden 情報。次の形式で出力されます。単位はキロバイトです。

```
<GC前の領域長>,<GC後の領域長>,<領域サイズ>
```

survivor_info

Survivor 情報。次の形式で出力されます。単位はキロバイトです。

```
<GC前の領域長>,<GC後の領域長>,<領域サイズ>
```

tenured_info

Tenured 情報。次の形式で出力されます。単位はキロバイトです。

```
<GC前の領域長>,<GC後の領域長>,<領域サイズ>
```

metaspace_info

Metaspace 領域の情報。次の形式で出力されます。単位はキロバイトです。

```
<GC前の使用サイズ>,<GC前のcapacityサイズ>,<GC前のcommitサイズ>,<GC後の使用サイズ>,<GC後のcapacityサイズ>,<GC後のcommitサイズ>
```

classspace_info

Compressed Class Space 情報。次の形式で出力されます。単位はキロバイトです。

```
<GC前の使用サイズ>,<GC前のcapacityサイズ>,<GC前のcommitサイズ>,<GC後の使用サイズ>,<GC後のcapacityサイズ>,<GC後のcommitサイズ>
```

cause_info

GC 要因番号。

-XX:-HitachiVerboseGCPrintCause オプションが指定された場合，出力されません。

なお，GC 要因番号については，「-XX:[+|-]HitachiVerboseGCPrintCause（GC 要因内容出力オプション）」を参照してください。

user_cpu

GC スレッドがユーザーモードで費やした CPU 時間。単位は秒です。

CPU 時間の取得に失敗した場合は"unknown"と表示されます。

-XX:-HitachiVerboseGCCpuTime オプションが指定された場合，出力されません。

system_cpu

GC スレッドがカーネルモードで費やした CPU 時間。単位は秒です。

CPU 時間の取得に失敗した場合は"unknown"と表示されます。

-XX:-HitachiVerboseGCCpuTime オプションが指定された場合，出力されません。

jvm_alloc_size

JavaVM 内部で管理している領域のうち，現在使用中の領域のサイズ（mmap_total_size と malloc_total_size の合計サイズのうち，現在使用中の領域のサイズ)。単位はキロバイトです。

-XX:-HitachiVerboseGCPrintJVMInternalMemory オプションが指定された場合，出力されません。

mmap_total_size

JavaVM 内部で管理している領域のうち，mmap で割り当てた C ヒープの総サイズ。単位はキロバイトです。

-XX:-HitachiVerboseGCPrintJVMInternalMemory オプションが指定された場合，出力されません。

malloc_total_size

JavaVM 内部で管理している領域のうち，malloc で割り当てた C ヒープ総サイズ。単位はキロバイトです。

-XX:-HitachiVerboseGCPrintJVMInternalMemory オプションが指定された場合，出力されません。

thread_count

Java スレッドの数。

-XX:-HitachiVerboseGCPrintThreadCount オプションが指定された場合，出力されません。

doe_alloc_size

java.io.File.deleteOnExit()を呼び出して確保した累積のヒープサイズ。単位はキロバイトです。

-XX:-HitachiVerboseGCPrintDeleteOnExit オプションが指定された場合，出力されません。

called_count

java.io.File.deleteOnExit()の呼び出し回数。

-XX:-HitachiVerboseGCPrintDeleteOnExit が指定された場合，出力されません。

cc_used_size

GC 発生時のコードキャッシュ領域の使用サイズ。単位はキロバイトです。

-XX:-PrintCodeCacheInfo オプションが指定された場合，出力されません。

cc_max_size

コードキャッシュ領域の最大サイズ。単位はキロバイトです。

-XX:-PrintCodeCacheInfo オプションが指定された場合，出力されません。

cc_info

保守情報。

-XX:-PrintCodeCacheInfo オプションが指定された場合，出力されません。

-XX:+HitachiVerboseGCPrintTenuringDistribution オプションが指定されている場合の出力内容を次に説明します。

```
id, date, size, value, max_value, total_age1, total_age2, total_agen
```

出力内容については，「-XX:[+|-]HitachiVerboseGCPrintTenuringDistribution（Survivor 領域の年齢分布出力オプション）」を参照してください。

出力形式（G1GC を使用している場合）

- VG1 ログ

```
id, date, gc_kind, gc_info, gc_time, gc_status, eden_info, survivor_info,tenured_info, humongous_info, free_info, metaspace_info, classspace_info, cause_info, region_size, target_time, predicted_time, target_size, reclaimable_info, user_cpu, system_cpu, jvm_alloc_size, mmap_total_size, malloc_total_size, thread_count, doe_alloc_size, called_count, cc_used_size, cc_max_size, cc_info
```

id

JavaVM ログファイル識別子。

date

GC 開始日時。-XX:-HitachiVerboseGCPrintDate オプションが指定された場合，出力されません。

gc_kind

GC 種別。“Full GC”，“Mixed GC”，“Young GC”，“Young GC(initial-mark)”，“CM Remark”，“CM Cleanup”のどれかが出力されます。

gc_info

GC 情報。次の形式で出力されます。単位はキロバイトです。

```
<GC前の領域サイズ>,<GC前の領域サイズ(リージョン換算)>,<GC前の領域サイズ>,<GC後の領域サイズ>,<GC前の領域サイズ(リージョン換算)>,<GC後の領域サイズ>
```

リージョン換算とは領域サイズを 1 リージョンのサイズで切り上げ，1 リージョンのサイズの倍数で表した値です。

gc_time

GC 経過時間。

gc_status

GC の状態。“-”，“to exhausted”が出力されます。

eden_info

Eden 情報。次の形式で出力されます。

```
<GC前の領域サイズ(リージョン換算)>,<GC前の最大領域サイズ(リージョン換算)>,<GC後の領域サイズ(リージョン換算)>,<GC後の最大領域サイズ(リージョン換算)>
```

リージョン換算とは領域サイズを 1 リージョンのサイズで切り上げ，1 リージョンのサイズの倍数で表した値です。

survivor_info

Survivor 情報。次の形式で出力されます。

```
<GC前の領域サイズ(リージョン換算)>,<GC後の領域サイズ(リージョン換算)>
```

リージョン換算とは領域サイズを 1 リージョンのサイズで切り上げ，1 リージョンのサイズの倍数で表した値です。

tenured_info

Tenured 情報。次の形式で出力されます。

```
<GC前の領域サイズ(リージョン換算)>,<GC後の領域サイズ(リージョン換算)>
```

リージョン換算とは領域サイズを 1 リージョンのサイズで切り上げ，1 リージョンのサイズの倍数で表した値です。

humongous_info

Humongous 情報。次の形式で出力されます。

```
<GC前の領域サイズ(リージョン換算)>,<GC後の領域サイズ(リージョン換算)>
```

リージョン換算とは領域サイズを 1 リージョンのサイズで切り上げ，1 リージョンのサイズの倍数で表した値です。

free_info

Free 情報。次の形式で出力されます。

```
<GC前の領域サイズ(リージョン換算)>,<GC後の領域サイズ(リージョン換算)>
```

リージョン換算とは領域サイズを 1 リージョンのサイズで切り上げ，1 リージョンのサイズの倍数で表した値です。

metaspace_info

Metaspace 領域の情報。次の形式で出力されます。単位はキロバイトです。

```
<GC前の使用サイズ>,<GC前のcapacityサイズ>,<GC前のcommitサイズ><GC後の使用サイズ>,<GC後のcapacityサイズ>,<GC後のcommitサイズ>
```

classspace_info

Compressed Class Space 情報。次の形式で出力されます。単位はキロバイトです。

```
<GC前の使用サイズ>,<GC前のcapacityサイズ>,<GC前のcommitサイズ><GC後の使用サイズ>,<GC後のcapacityサイズ>,<GC後のcommitサイズ>
```

cause_info

GC 要因番号。

-XX:-HitachiVerboseGCPrintCause オプションが指定された場合，出力されません。

なお，GC 要因番号については，「-XX:[+|-]HitachiVerboseGCPrintCause（GC 要因内容出力オプション）」を参照してください。

region_size

1 リージョンのサイズです。

単位はキロバイトです。

target_time

GC によるアプリケーション停止時間の目標時間です。

単位は秒です。

predicted_time

JavaVM が予測した GC によるアプリケーション停止時間です。

単位は秒です。

なお，GC 種別が“Full GC”，“CM Remark”，“CM Cleanup”のときは予測をしないため，0 が出力されます。

target_size

Mixed GC で GC 対象となった Tenured 領域のサイズです。

単位はキロバイト。

なお，GC 種別が“Mixed GC”以外のときは，0 が出力されます。

reclaimable_info

Mixed GC で回収される Tenured 領域の予測回収サイズ情報。次の形式で出力されます。

<予測回収サイズ>（<予測回収率>）

なお，予測回収サイズ情報は CM 終了直後の Young GC または Mixed GC だけ出力されます。それ以外の場合，予測を行わないため，0 が出力されます。

user_cpu

GC スレッドがユーザーモードで費やした CPU 時間。単位は秒です。

CPU 時間の取得に失敗した場合は"unknown"と表示されます。

-XX:-HitachiVerboseGCCpuTime オプションが指定された場合，出力されません。

system_cpu

GC スレッドがカーネルモードで費やした CPU 時間。単位は秒です。

CPU 時間の取得に失敗した場合は"unknown"と表示されます。

-XX:-HitachiVerboseGCCpuTime オプションが指定された場合，出力されません。

jvm_alloc_size

JavaVM 内部で管理している領域のうち，現在使用中の領域のサイズ（mmap_total_size と malloc_total_size の合計サイズのうち，現在使用中の領域のサイズ）。単位はキロバイトです。

-XX:-HitachiVerboseGCPrintJVMInternalMemory オプションが指定された場合，出力されません。

mmap_total_size

JavaVM 内部で管理している領域のうち，mmap で割り当てた C ヒープの総サイズ。単位はキロバイトです。

-XX:-HitachiVerboseGCPrintJVMInternalMemory オプションが指定された場合，出力されません。

malloc_total_size

JavaVM 内部で管理している領域のうち，malloc で割り当てた C ヒープ総サイズ。単位はキロバイトです。

-XX:-HitachiVerboseGCPrintJVMInternalMemory オプションが指定された場合，出力されません。

thread_count

Java スレッドの数。

-XX:-HitachiVerboseGCPrintThreadCount オプションが指定された場合，出力されません。

doe_alloc_size

java.io.File.deleteOnExit()を呼び出して確保した累積のヒープサイズ。単位はキロバイトです。

-XX:-HitachiVerboseGCPrintDeleteOnExit オプションが指定された場合，出力されません。

called_count

java.io.File.deleteOnExit()の呼び出し回数。

-XX:-HitachiVerboseGCPrintDeleteOnExit が指定された場合，出力されません。

cc_used_size

GC 発生時のコードキャッシュ領域の使用サイズ。単位はキロバイトです。

-XX:-PrintCodeCacheInfo オプションが指定された場合，出力されません。

cc_max_size

コードキャッシュ領域の最大サイズ。単位はキロバイトです。

-XX:-PrintCodeCacheInfo オプションが指定された場合，出力されません。

cc_info

保守情報。

-XX:-PrintCodeCacheInfo オプションが指定された場合，出力されません。

-XX:+HitachiVerboseGCPrintTenuringDistribution オプションが指定されている場合の出力内容を次に説明します。

```
id, date, size, value, max_value, total_age1, total_age2, total_agen
```

出力内容については，「-XX:[+|-]HitachiVerboseGCPrintTenuringDistribution（Survivor 領域の年齢分布出力オプション）」を参照してください。

- VCM ログ

```
id,date,cm_event,user_cpu,sys_spu
```

id

VCM（JavaVM ログファイル識別子）。

date

CM 開始日時。

-XX:-HitachiVerboseGCPrintDate オプションが指定された場合，出力されません。

cm_event

CM 種別。”Concurrent Root Region Scan Start”，”Concurrent Root Region Scan End”，”Concurrent Mark Start”，”Concurrent Mark End”，”Concurrent Mark Stop”，”Concurrent Cleanup Start”，”Concurrent Cleanup End”のどれかが出力されます。

user_cpu

全 CM スレッドがユーザーモードで費やした CPU 時間。単位は秒です。

-XX:-HitachiVerboseGCCpuTime 指定時は出力されません。

CPU 時間取得に失敗した場合，"unknown"と表示されます。

CM の状態が Start の場合，0 が出力されます。

sys_cpu

全 CM スレッドがカーネルモードで費やした CPU 時間。単位は秒です。

-XX:-HitachiVerboseGCCpuTime 指定時は出力されません。

CPU 時間取得に失敗した場合，"unknown"と表示されます。

CM の状態が Start の場合，0 が出力されます。

出力例

SerialGC を使用している場合

- -XX:HitachiVerboseGCIntervalTime オプションが指定されている場合

```
VGC,Fri Jan 23 21:37:50 2004,11,41,0,GC,16886,16886,65088,0.0559806,
4094,0,4096,447,447,448,12345,16439,60544,1116,1116,4096,0,0.0312500, 0.0156250,729,928,0
,509,2167,2054,2301,49152,2304
VGC,Fri Jan 23 21:37:55 2004,6,24,0,Full GC,65082,65082,65088,0.4294532,
4094,4094,4096,447,447,448,60541,60541,60544,1116,1116,4096,0,0.0156250, 0.0312500,729,92
8,0,509,16,170,2301,49152,2304
…
```

- -XX:+HitachiVerboseGCPrintTenuringDistribution オプションが指定されている場合

```
PTD,Wed May 28 11:45:23 2008,5467547,30,31,1357527,1539661
```

G1GC を使用している場合

- VG1 ログ

```
VG1,Thu Oct 02 10:38:54.920 2014,Full GC,753,2048,8192,678,1024,8192,0.0064767,-,1024,204
8,0,2048,0,0,1024,1024,0,0,6144,7168,3634,3634,4492,3634,3634,4492,356,356,388,356,356,38
```

```
8,1,1024,0.2000000,0.0000000,0,0,0.00,0.0000000,0.0000000,20459,21920,0,35,0,0,1171,24576
0,2496
```

- VCM ログ

```
VCM,Fri Jul 26 21:35:50 2013,Concurrent Mark Start,0.0000000,0.0000000
VCM,Fri Jul 26 21:35:50 2013,Concurrent Mark End,0.0124532,0.0245698
```

-XX:HitachiVerboseGCIntervalTime（拡張 verbosegc 情報出力間隔指定オプション）

形式

```
-XX:HitachiVerboseGCIntervalTime=<整数値>
```

説明

拡張 verbosegc 情報を出力する時間（秒）の間隔を指定します。

オプションを指定した場合

拡張 verbosegc 情報を GC ごとに出力するのではなく，指定された時間を超えた次の GC を出力します。このとき，前回の出力から今回の出力までの間に発生した，次に示す GC の回数も出力します。これらは必ず出力されます。

表 5-4　回数を出力する GC の一覧

文字列	意味
Full	FullGC をスキップした回数
Copy	CopyGC をスキップした回数

オプションを指定しない場合

0 秒が設定され，GC 発生ごとに拡張 verbosegc 情報を出力します。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- -XX:+HitachiVerboseGC

引数

<整数値>

0～2147483647 の範囲の整数値（単位：秒）を指定します。範囲外の値が指定された場合は 0 が設定されます。負の値を指定した場合はエラーとなります。

-XX:[+|-]HitachiVerboseGCPrintCause（GC 要因内容出力オプション）

形式

-XX:+HitachiVerboseGCPrintCause

GC の要因内容を，拡張 verbosegc 情報の行末に出力します。

-XX:-HitachiVerboseGCPrintCause

拡張 verbosegc 情報を通常形式で出力します。

説明

GC の要因内容を出力するかどうかを指定します。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- -XX:+HitachiVerboseGC

-XX:+HitachiCommaVerboseGC オプションが指定されている場合は，次に示す要因番号が出力されます。

表 5-5　GC 要因一覧

要因番号	要因内容	説明	SerialGC 使用時の要因	G1GC 使用時の要因
0	ObjAllocFail	G1GC 未使用時，オブジェクトの割り付け失敗によって GC が発生しました。G1GC 使用時，Evacuation を実施してもオブジェクト割り付け領域が確保できなかったので GC が発生しました。	○	○
1	System.gc	java.lang.System.gc メソッド呼び出しによって GC が発生しました。	○	○
3	DelayedGC	JNI や JVMTI によって保留されていた GC が起動されました。	○	○
4	JavaGC Command	javagc コマンドによって GC が発生しました。	○	○
6	JHeapProf Command	jheapprof コマンドによって GC が発生しました。	○	－
10	JVMTIForceGC	JVMTI 関数 ForceGarbageCollection()によって GC が発生しました。	○	○
11	PromotionFail	CopyGC の昇格失敗によって GC が発生しました。	○	－

要因番号	要因内容	説明	SerialGC 使用時の要因	G1GC 使用時の要因
14	G1HumAllocFail	Humongous 領域へのオブジェクト割り当て失敗によって GC が発生しました。	−	○
15	G1EvacuationPause	オブジェクト割り当て失敗によって GC が発生しました。	−	○
16	Concurrent Marking	アプリケーションを停止して実行する CM 処理が発生しました。	−	○
17	EvacuationFail	Evacuation 失敗によって GC が発生しました。	−	○
18	MetaspaceAllocFail	Metaspace の領域確保失敗によって GC が発生しました。	○	○
19	LastMetaspaceGC	Metaspace の OutOfMemory を出す前に行う最後の GC が発生しました。	○	○

（凡例）

○：該当します。

−：該当しません。

出力例

```
[VGC]<Wed Mar 17 00:42:30 2004>(Skip Full:0,Copy:0)[Full GC 149K->149K(1984K), 0.0786038 secs][DefNew::Eden: 264K->0K(512K)][DefNew::Survivor: 0K->63K(64K)][Tenured: 85K->149K(1408K)][Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][cause:System.gc]
```

-XX:[+|-]HitachiVerboseGCPrintDate（拡張 verbosegc 情報日付出力オプション）

形式

`-XX:+HitachiVerboseGCPrintDate`

拡張 verbosegc 情報の各出力行の先頭に GC 開始日時を出力します。

`-XX:-HitachiVerboseGCPrintDate`

拡張 verbosegc 情報の各出力行の先頭に GC 開始日時を出力しません。

説明

拡張 verbosegc 情報に GC 開始日時を出力するかどうかを指定します。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- -XX:+HitachiVerboseGC

-XX:[+|-]HitachiVerboseGCCpuTime（拡張 verbosegc 情報 CPU 利用時間出力オプション）

形式

-XX:+HitachiVerboseGCCpuTime

拡張 verbosegc 情報に，GC の開始から終了までで，GC 実行スレッドのユーザーモードおよびカーネルモードに費やされたプロセッサ時間を出力します。

-XX:-HitachiVerboseGCCpuTime

拡張 verbosegc 情報に，GC の開始から終了までで，GC 実行スレッドのユーザーモードおよびカーネルモードに費やされたプロセッサ時間を出力しません。

説明

拡張 verbosegc 情報に CPU 利用時間を出力するかどうかを指定します。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- -XX:+HitachiVerboseGC

出力例

```
[VGC]<Thu Oct 02 10:38:53.658 2014>(Skip Full:1,Copy:0)[Full GC 770K->682K(8064K), 0.0050003
 secs][DefNew::Eden: 88K->0K(2304K)][DefNew::Survivor: 0K->0K(256K)][Tenured: 681K->682K(550
4K)] [Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->35
6K(388K, 388K)] [cause:System.gc][User: 0.0000000 secs][Sys: 0.0000000 secs]
```

-XX:[+|-]HitachiVerboseGCPrintTenuringDistribution（Survivor 領域の年齢分布出力オプション）

形式

-XX:+HitachiVerboseGCPrintTenuringDistribution

Survivor 領域の年齢分布を JavaVM ログファイルへ出力します。

-XX:-HitachiVerboseGCPrintTenuringDistribution

Survivor 領域の年齢分布を JavaVM ログファイルへ出力しません。

説明

Survivor 領域の年齢分布を JavaVM ログファイルへ出力するかどうかを指定します。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- -XX:+HitachiVerboseGC

関連オプション

- -XX:+HitachiVerboseGCPrintDate
- -XX:+HitachiCommaVerboseGC

出力形式

```
[id]<date>[Desired survivor:size bytes][New threshold:value][MaxTenuringThreshold: max_value
][age1:total_age1][age2:total_age2]…[agen:total_agen]
```

出力内容を次に説明します。

id

PTD（JavaVM ログファイル識別子）。

date

GC 開始日時。

size

GC 後の Survivor 領域内オブジェクト目標サイズ。

value

次の CopyGC で Tenured 領域に昇格する Java オブジェクトの年齢のしきい値。

この値は，-XX:MaxTenuringThreshold=<value>オプションと，Survivor 領域のメモリサイズ，および，-XX:TargetSurvivorRatio=<value>オプションに設定した値を基に，CopyGC ごとに動的に設定されます。

value 値以上の年齢の Java オブジェクトが，次の CopyGC で Tenured 領域に昇格します。

max_value

CopyGC で Tenured 領域に昇格する Java オブジェクトの年齢のしきい値（value 値）の最大値（MaxTenuringThreshold オプションの指定値）。

value 値は，CopyGC ごとに動的に設定されますが，max_value 値を超えることはありません。

また，年齢が max_value 値以上の Java オブジェクトは，次の CopyGC で，必ず，Tenured 領域に昇格します。

total_age1

1 歳のオブジェクトのバイト数の合計。

total_age2

1 歳から 2 歳までのオブジェクトのバイト数の合計。

total_agen

1 歳から n 歳までのオブジェクトのバイト数の合計。

n が max_value に近ければ寿命の長いオブジェクトが存在するということになります。

出力例

```
[PTD]<Wed Jan 28 17:47:10 2009>[Desired survivor:32768 bytes][New threshold:30][MaxTenuringT
hreshold:30][age1:6872][age2:9632][age3:25632]
```

-XX:[+|-]HitachiVerboseGCPrintJVMInternalMemory（C ヒープ情報出力オプション）

形式

-XX:+HitachiVerboseGCPrintJVMInternalMemory

JavaVM 内部で管理しているヒープ情報を JavaVM ログファイルへ出力します。

-XX:-HitachiVerboseGCPrintJVMInternalMemory

JavaVM 内部で管理しているヒープ情報を JavaVM ログファイルへ出力しません。

説明

JavaVM 内部で管理しているヒープ情報を JavaVM ログファイルに出力するかどうかを指定します。

C ヒープ領域のうち，次の 2 種類の方法で取得したヒープ領域は，JavaVM 内部で管理する方式で管理されています。

- mmap で取得した C ヒープ領域
- malloc で取得した C ヒープ領域

-XX:+HitachiVerboseGCPrintJVMInternalMemory オプションを有効にした場合，mmap で取得した C ヒープの総サイズ（mmap_total_size）と，malloc で取得した C ヒープの総サイズ（malloc_total_size）を出力できます。また，これらの割り当て済みの領域のうち，使用中の領域のサイズの合計値（jvm_alloc_size）も出力できます。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- -XX:+HitachiVerboseGC

出力形式

```
[id] <date> (Skip Full:full_count, Copy:copy_count, Inc:inc_count) [gc_kind gc_info, gc_time
 secs][Eden: eden_info][Survivor: survivor_info][Tenured: tenured_info] [Metaspace: metaspac
e_info][class space: class_space_info][cause:cause_info] [User: user_cpu secs][Sys: system_c
pu secs][IM: jvm_alloc_size, mmap_total_size, malloc_total_size][TC: thread_count][DOE: doe_
alloc_size, called_count]
```

出力内容を次に示します。なお，ここでは，-XX:+HitachiVerboseGCPrintJVMInternalMemory オプションによって出力される項目について説明します。ここで説明している以外の項目については，「-XX:[+|-]HitachiVerboseGC（拡張 verbosegc 情報出力オプション）」の出力形式の説明を参照してください。

jvm_alloc_size

JavaVM 内部で管理している領域のうち，現在使用中の領域のサイズ（mmap_total_size と malloc_total_size の合計サイズのうち，現在使用中の領域のサイズ）。

mmap_total_size

JavaVM 内部で管理している領域のうち，mmap で割り当てた C ヒープの総サイズ。

malloc_total_size

JavaVM 内部で管理している領域のうち，malloc で割り当てた C ヒープの総サイズ。

出力例

```
[VGC]<Wed Jan 27 13:03:36 2010>(Skip Full:0,Copy:0)[GC 489K->152K(3520K), 0.0156080 secs][De
fNew::Eden: 489K->0K(2048K)][DefNew::Survivor: 0K->63K(64K)][Tenured: 0K->88K(1408K)][Metasp
ace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388
K)] [cause:ObjAllocFail][IM: 729K, 928K, 0K][TC: 509][DOE: 16K, 170]
```

-XX:[+|-]HitachiVerboseGCPrintThreadCount（スレッド数の出力オプション）

形式

-XX:+HitachiVerboseGCPrintThreadCount

Java スレッドの数を出力します。

-XX:-HitachiVerboseGCPrintThreadCount

Java スレッドの数を出力しません。

説明

Java スレッドの数を監視するために，Java スレッドの数を JavaVM ログファイルに出力するかどうかを指定します。

スレッドは，スタックに使用するためのメモリを C ヒープとして個々で確保します。このため，スレッド数が多くなると，C ヒープの確保量もスレッド数に比例して多くなります。-XX:+HitachiVerboseGCPrintThreadCount オプションを指定することで，Java スレッドの数を監視できるようになり，C ヒープの確保量の把握ができるようになります。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- -XX:+HitachiVerboseGC

出力形式

```
[id] <date> (Skip Full:full_count, Copy:copy_count, Inc:inc_count) [gc_kind gc_info, gc_time
 secs][Eden: eden_info][Survivor: survivor_info][Tenured: tenured_info][Metaspace: metaspace
_info][class space: class_space_info][cause:cause_info] [User: user_cpu secs][Sys: system_cp
u secs][IM: jvm_alloc_size, mmap_total_size, malloc_total_size][TC: thread_count][DOE: doe_a
lloc_size, called_count]
```

出力内容を次に示します。なお，ここでは，-XX:+HitachiVerboseGCPrintThreadCount オプションによって出力される項目について説明します。ここで説明している以外の項目については，「-XX:[+|-]HitachiVerboseGC（拡張 verbosegc 情報出力オプション）」の出力形式の説明を参照してください。

thread_count

Java スレッドの数。

出力例

```
[VGC]<Wed Jan 27 13:03:36 2010>(Skip Full:0,Copy:0)[GC 489K->152K(3520K), 0.0156080 secs][De
fNew::Eden: 489K->0K(2048K)][DefNew::Survivor: 0K->63K(64K)][Tenured: 0K->88K(1408K)][Metasp
ace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388
K)][cause:ObjAllocFail][IM: 729K, 928K, 0K][TC: 509]
```

-XX:[+|-]HitachiVerboseGCPrintDeleteOnExit（java.io.File.deleteOnExit()が使用するヒープサイズの出力オプション）

形式

-XX:+HitachiVerboseGCPrintDeleteOnExit

java.io.File.deleteOnExit()を呼び出して確保した累積のヒープサイズと，メソッドの呼び出し回数を出力します。

-XX:-HitachiVerboseGCPrintDeleteOnExit

java.io.File.deleteOnExit()を呼び出して確保した累積のヒープサイズと，メソッドの呼び出し回数を出力しません。

説明

java.io.File.deleteOnExit()を呼び出したことによってJavaVMが確保した累積のヒープサイズとメソッドの呼び出し回数を，JavaVMログファイルに出力するかどうかを指定します。

java.io.File.deleteOnExit()は，呼び出すたびに指定されたファイルのパス情報をヒープに確保しますが，確保した領域はプロセスの終了まで解放しないため，メモリの圧迫につながるおそれがあります。-XX:+HitachiVerboseGCPrintDeleteOnExitオプションを指定すると，JavaVMがjava.io.File.deleteOnExit()を呼び出して確保したヒープサイズをログに出力して監視できるようになります。また，java.io.File.deleteOnExit()の呼び出し状況を把握するための補助的な情報として，メソッドの呼び出し回数も同時に出力できます。

出力した情報は，障害発生時，java.io.File.deleteOnExit()の呼び出しによって確保されたヒープサイズを把握してメモリ不足の原因を調査するために役立てられます。また，運用開始前の開発やテスト段階で，java.io.File.deleteOnExit()の呼び出しによって確保したヒープサイズの増加の推移を確認して，運用時にメモリを圧迫する予兆がないかを事前確認するためにも利用できます。

なお，エラーが発生した場合は，エラーメッセージが出力されます。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- -XX:+HitachiVerboseGC

出力形式

JavaVMログファイルの出力形式を次に示します。

```
[id] <date> (Skip Full:full_count, Copy:copy_count, Inc:inc_count) [gc_kind gc_info, gc_time
 secs][Eden: eden_info][Survivor: survivor_info][Tenured: tenured_info] [Metaspace: metaspac
e_info][class space: class_space_info][cause:cause_info] [User: user_cpu secs][Sys: system_c
```

```
pu secs][IM: jvm_alloc_size, mmap_total_size, malloc_total_size][TC: thread_count][DOE: doe_
alloc_size, called_count]
```

出力内容を次に示します。なお，ここでは，-XX:+HitachiVerboseGCPrintDeleteOnExit オプションによって出力される項目について説明します。ここで説明している以外の項目については，「-XX:[+|-]HitachiVerboseGC（拡張 verbosegc 情報出力オプション）」の出力形式の説明を参照してください。

doe_alloc_size

java.io.File.deleteOnExit()を呼び出して確保した累積のヒープサイズ。

called_count

java.io.File.deleteOnExit()の呼び出し回数。

エラーが発生した場合のエラーメッセージの出力形式を次に示します。

```
[DOE]<date>Error occurred during processing of java.io.File.deleteOnExit's heap size output
function. (<保守情報>)
[DOE]java.io.File.deleteOnExit's heap size output function stopped.
```

エラーメッセージの出力内容を次に示します。

DOE

java.io.File.deleteOnExit()のヒープサイズ出力機能でエラーが発生したことを示す識別子。

date

エラーが発生した日時。

出力例

- JavaVM ログファイルの出力例を次に示します。

```
[VGC]<Wed Jan 27 13:03:36 2010>(Skip Full:0,Copy:0)[GC 489K->152K(3520K), 0.0156080 secs]
[DefNew::Eden: 489K->0K(2048K)][DefNew::Survivor: 0K->63K(64K)][Tenured: 0K->88K(1408K)][
Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(
388K, 388K)][cause:ObjAllocFail][IM: 729K, 928K, 0K] [TC: 509][DOE: 16K, 170]
```

- エラーメッセージの出力例を次に示します。

```
[DOE]<Wed Jan 27 13:03:36 2010> Error occurred during processing of java.io.File.deleteOn
Exit's heap size output function. (FindClass:java.lang.String)
[DOE]java.io.File.deleteOnExit's heap size output function stopped.
```

注意事項

- 次の場合は，java.io.File.deleteOnExit()を呼び出しても累積のヒープサイズ，メソッドの呼び出し回数がカウントされません。
  - java.io.File.deleteOnExit()を呼び出した場合に SecurityException 例外が発生したとき（この例外はセキュリティマネージャの SecurityManager.checkDelete()がファイルへの削除アクセスを許可しない場合に発生します。この場合，メソッドの入り口で例外が挙がり，ヒープは確保されません）。

- アプリケーションサーバのバッチアプリケーション実行基盤で作成されたアプリケーションから呼び出した場合。
- 同一のパス名文字列で作成した File インスタンスを使用して java.io.File.deleteOnExit()を呼び出したとき。

- この機能が出力するヒープサイズを確認する際には，次の点に注意してください。
  - java.io.File.deleteOnExit()が確保するヒープの種類は Java ヒープです。
  - ヒープサイズはキロバイト単位で出力され，1 キロバイト未満は切り捨てられます。java.io.File.deleteOnExit()の 1 回の呼び出しで確保するヒープサイズは，ファイルパスの長さに応じて数十バイトから 100 バイト程度であるため，呼び出しごとにヒープサイズの出力結果が増加しないことがあります。この場合，メソッドの呼び出し回数からメソッドの実行を確認できます。

-XX:[+|-]PrintCodeCacheInfo（コードキャッシュ領域情報出力オプション）

形式

`-XX:+PrintCodeCacheInfo`

コードキャッシュ領域の使用量を出力します。

また，コードキャッシュ領域の使用量がしきい値に達したことを知らせるメッセージを出力します。

`-XX:-PrintCodeCacheInfo`

コードキャッシュ領域の使用量を出力しません。

また，コードキャッシュ領域の使用量がしきい値に達したことを知らせるメッセージを出力しません。

説明

コードキャッシュ領域の使用量を出力するかどうか，また，使用量がしきい値に達したことを知らせるメッセージを JavaVM ログファイルに出力するかどうかを指定します。

コードキャッシュ領域については，「2.2.1(5)　SerialGC 使用時の JavaVM で使用するメモリ空間の構成と JavaVM オプション」を参照してください。

このオプションを有効にすると，GC 発生時，コードキャッシュ領域の使用量が拡張 verbosegc 情報に出力されます。また，コードキャッシュ領域の使用量がしきい値に達したとき，メッセージが出力されます。

しきい値は「コードキャッシュ領域の最大サイズ×-XX:CodeCacheInfoPrintRatio オプションの値÷100」です。

-XX:-HitachiVerboseGC オプションを指定している場合でも，javagc コマンドの-v オプションや-s オプションによって出力する拡張 verbosegc 情報には，コードキャッシュ領域の使用量が出力されます。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- -XX:+HitachiVerboseGC

関連オプション

- -XX:+HitachiCommaVerboseGC
- -XX:CodeCacheInfoPrintRatio

出力形式

コードキャッシュ領域の使用量の出力形式を次に示します。

```
[id] <date> (Skip Full:full_count, Copy:copy_count, Inc:inc_count) [gc_kind gc_info, gc_time
 secs][Eden: eden_info][Survivor: survivor_info][Tenured: tenured_info] [Metaspace: metaspac
e_info][class space: class_space_info][cause:cause_info] [User: user_cpu secs][Sys: system_c
pu secs][IM: jvm_alloc_size, mmap_total_size, malloc_total_size][TC: thread_count][DOE: doe_
alloc_size, called_count][CCI: cc_used_sizeK, cc_max_sizeK, cc_infoK]
```

出力内容を次に示します。なお，ここでは，-XX:+PrintCodeCacheInfo オプションによって出力される項目について説明します。ここで説明している以外の項目については，「-XX:[+|-]HitachiVerboseGC (拡張 verbosegc 情報出力オプション)」の出力形式の説明を参照してください。

cc_used_size

GC 発生時のコードキャッシュ領域の使用サイズ。単位はキロバイトです。

cc_max_size

コードキャッシュ領域の最大サイズ。単位はキロバイトです。

cc_info

保守情報。

コードキャッシュ領域の使用量がしきい値に達したことを知らせるメッセージの出力形式を次に示します。

```
[cc_id]<cc_date>CodeCache usage has exceeded the threshold.[cc_used_sizeK, cc_max_sizeK, cc_
infoK]
```

メッセージの出力内容を次に示します。

cc_id

CCI（JavaVM ログファイル識別子）。

cc_date

JIT コンパイルを実行した日時。

cc_used_size

JIT コンパイル後のコードキャッシュ領域の使用サイズ。単位はキロバイトです。

cc_max_size

コードキャッシュ領域の最大サイズ。単位はキロバイトです。

cc_info

保守情報。

出力例

- コードキャッシュ領域の使用量の出力例を次に示します。

```
[VGC]<Wed Mar 17 00:42:30 2004>(Skip Full:0,Copy:0)[Full GC 149K->149K(1984K), 0.0786038
secs][DefNew::Eden: 264K->0K(512K)][DefNew::Survivor: 0K->63K(64K)][Tenured: 85K->149K(14
08K)][Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)-
>356K(388K, 388K)][cause:System.gc][User: 0.0156250 secs][Sys: 0.0312500 secs][IM: 729K,
928K, 0K][TC: 509][DOE: 16K, 170][CCI: 2301K, 49152K, 2304K]
```

- コードキャッシュ領域の使用量がしきい値に達したことを知らせるメッセージの出力例を次に示します。

```
[CCI]<Wed Dec 26 14:27:53 2012>CodeCache usage has exceeded the threshold.[39358K, 49152K
, 39360K]
```

注意事項

- コードキャッシュ領域の使用量がしきい値に達した状態で推移している場合，Java メソッドが JIT コンパイルされてもメッセージは出力されません。

  一方，コードキャッシュ領域の使用量がしきい値未満まで下がったあと，Java メソッドが JIT コンパイルされたことによって再びコードキャッシュ領域の使用量がしきい値に達した場合には，メッセージが出力されます。

- システムが使用するコードキャッシュ領域は最大 2 メガバイトです。そのため，システムのコードキャッシュ領域の使用量によっては，コードキャッシュ領域を最大サイズまで使用していない場合でも，コードキャッシュ領域が枯渇することがあります。

  また，コードキャッシュ領域の使用量のしきい値に大きな値を指定している場合，メッセージを出力する前にコードキャッシュ領域が枯渇する場合があります。コードキャッシュ領域が枯渇する前にメッセージを出力したいときは，「コードキャッシュ領域の最大サイズ－しきい値」の値が 4 メガバイト以上になるように，「-XX:CodeCacheInfoPrintRatio（コードキャッシュ領域使用率指定オプション）」の値を指定してください。

-XX:CodeCacheInfoPrintRatio（コードキャッシュ領域使用率指定オプション）

形式

```
-XX:CodeCacheInfoPrintRatio=<整数値>
```

説明

コードキャッシュ領域の使用量がしきい値に達したことを知らせるメッセージを出力する契機となる，コードキャッシュ領域の使用率を指定します。

ここで指定した使用率を基に，次の計算式でしきい値が計算されます。

コードキャッシュ領域の最大サイズ×-XX:CodeCacheInfoPrintRatioオプションの値÷100

コードキャッシュ領域については，「2.2.1(5)　SerialGC 使用時の JavaVM で使用するメモリ空間の構成と JavaVM オプション」を参照してください。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- -XX:+PrintCodeCacheInfo

引数

<整数値>

0～100 の範囲で整数値（単位：%）を指定します。範囲外の値が指定された場合は 80 が設定されます。

-XX:[+|-]PrintCodeCacheFullMessage（コードキャッシュ領域枯渇メッセージ出力オプション）

形式

-XX:+PrintCodeCacheFullMessage

Java メソッドが JIT コンパイルの対象になった場合，コードキャッシュ領域が枯渇していたときにメッセージを出力します。メッセージは一度だけ出力されます。

-XX:-PrintCodeCacheFullMessage

Java メソッドが JIT コンパイルの対象になった場合，コードキャッシュ領域が枯渇してもメッセージを出力しません。

説明

Java メソッドが JIT コンパイルの対象になった場合，コードキャッシュ領域が枯渇していたときに，メッセージを JavaVM ログファイルに出力するかどうかを指定します。

コードキャッシュ領域については，「2.2.1(5)　SerialGC 使用時の JavaVM で使用するメモリ空間の構成と JavaVM オプション」を参照してください。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- -XX:+HitachiVerboseGC

出力形式

メッセージの出力形式を次に示します。

```
[cc_id]<cc_date>CodeCache is full. Compiler has been disabled.[cc_used_sizeK, cc_max_sizeK,
cc_infoK]
```

メッセージの出力内容を次に示します。

cc_id
: CCI（JavaVM ログファイル識別子）。

cc_date
: Java メソッドが JIT コンパイルの対象になった日時。

cc_used_size
: Java メソッドが JIT コンパイルの対象になったときのコードキャッシュ領域の使用サイズ。単位はキロバイトです。

cc_max_size
: コードキャッシュ領域の最大サイズ。単位はキロバイトです。

cc_info
: 保守情報。

出力例

メッセージの出力例を次に示します。

```
[CCI]<Wed Dec 26 14:38:29 2012>CodeCache is full. Compiler has been disabled.[49151K, 49152K
, 49152K]
```

注意事項

システムが使用するコードキャッシュ領域は最大 2 メガバイトです。そのため，システムのコードキャッシュ領域の使用量によっては，コードキャッシュ領域を最大サイズまで使用していない場合でも，コードキャッシュ領域が枯渇することがあります。

-XX:[+|-]HitachiOutOfMemoryCause（例外発生要因種別出力オプション）

形式

-XX:+HitachiOutOfMemoryCause

OutOfMemoryError 発生時に，例外発生要因種別を JavaVM ログファイルに出力します。-XX:+HitachiOutOfMemoryStackTrace オプションが指定されている場合は，このオプションも設定されます。

出力される要因種別を次に示します。

表 5-6　例外発生要因種別一覧

項番	要因メッセージ	説明
1	C Heap	C ヒープ確保時の例外
2	Java Heap	Java ヒープ確保時の例外
3	Meta Space	Metaspace 領域確保時の例外
4	Compressed Class Space	Compressed Class Space 確保時の例外
5	Unknown	上記例外を特定できない場合
6	Thread Limit	スレッド数の上限値設定機能（-XX:HitachiThreadLimit オプション）を指定した場合に，作成したスレッド数が指定した上限値を超えたときの例外

-XX:-HitachiOutOfMemoryCause

OutOfMemoryError 発生時に，例外発生要因種別を JavaVM ログファイルに出力しません。

説明

OutOfMemoryError 発生時の発生要因種別を出力するかどうかを指定します。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

出力例

スレッド数の上限値設定機能（HitachiThreadLimit オプション）を指定していて，作成したスレッド数が指定した上限値を超えた結果，OutOfMemoryError をスローする場合の例を示します。

```
[OOM][Thread: 0x00062fd0]<Tue Dec 2 16:42:39 2003>[java.lang.OutOfMemoryError :(C Heap) : unable to create thread : 340 threads exist]
```

-XX:[+|-]HitachiOutOfMemoryStackTrace（スタックトレース出力オプション）

形式

-XX:+HitachiOutOfMemoryStackTrace

OutOfMemoryError 発生時に，例外情報とスタックトレースを JavaVM ログファイルに出力します。

スタックトレースは 1 スタックごとにバッファに格納し，コード変換したあとに出力します。スタックトレースの出力は，OutOfMemoryError がスローされるたびに行われるため，OutOfMemoryError をキャッチして再スローした場合には複数回出力されます。なお，スレッド作成時に OutOfMemoryError となった場合は，スタックトレースは出力されません。

-XX:-HitachiOutOfMemoryStackTrace

OutOfMemoryError 発生時に，スタックトレースを JavaVM ログファイルに出力しません。

説明

OutOfMemoryError 発生時に，例外情報とスタックトレースを JavaVM ログファイルに出力するかどうかを指定します。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

出力形式

```
[id] [Thread:thread_id]<date>[java.lang.OutOfMemoryError : requested size bytes (cause) : re
ason : thread_count threads exist]
[id] [Thread:thread_id] stack_trace
```

出力内容を次に説明します。

id

OOM（JavaVM ログファイル識別子）。

thread_id

スレッド ID（スレッドダンプに出力されている tid）。

date

OutOfMemory 例外発生日時。

size

確保しようとしたメモリのサイズ（単位：バイト）が出力されます。-XX:-HitachiOutOfMemorySize オプションが指定されている場合，出力されません。

なお，次の場合は要求したメモリサイズが取り出せません。これらの場合は，サイズとして「unknown」が出力されます。

- Java のメモリ確保機能を使用しないで，明示的に OutOfMemoryError をスローした場合
  標準クラスライブラリによってスローされるものも含みます。例えば，「throw new OutOfMemoryError();」などの処理によってスローされた場合，メモリサイズは出力できません。
- クラスロード時の verifier によって OutOfMemoryError が発生した場合

cause

例外発生要因種別。ただし，-XX:-HitachiOutOfMemoryCause オプションが指定されている場合は出力されません。例外発生要因種別については，「-XX:[+|-]HitachiOutOfMemoryCause（例外発生要因種別出力オプション）」を参照してください。

reason

例外発生理由。スレッドの作成に失敗した場合に出力されます。

thread_count

OutOfMemoryError 発生時のスレッド数。作成に失敗したスレッド数も含みます。

stack_trace

スタックトレース。

出力例

```
[OOM][Thread: 0x00062fd0] <Wed Mar 17 00:41:17 2004>[java.lang.OutOfMemoryError :requested 4
00000 bytes.(C Heap): unable to create thread : 1500 threads exist]
[OOM][Thread: 0x00062fd0] at java.lang.Thread.start(Native Method)
[OOM][Thread: 0x00062fd0] at sub1.<init>(Thread0012.java:22)
[OOM][Thread: 0x00062fd0] at Thread0012.test01(Thread0012.java:73)
[OOM][Thread: 0x00062fd0] at Thread0012.main(Thread0012.java:57)
```

注意事項

JavaVM 自身で作成するスレッドが，メモリ不足によってスレッドの作成に失敗した場合は，例外情報だけが出力されます。スタックトレースは出力されません。

-XX:HitachiOutOfMemoryStackTraceLineSize（スタックトレース行サイズ指定オプション）

形式

```
-XX:HitachiOutOfMemoryStackTraceLineSize=<整数値>
```

説明

OutOfMemoryError 発生時に，出力するスタックトレース 1 行の文字数をバイト数で指定します。オプションの指定がない場合は，1024 バイトが設定されます。指定したバイト数が確保できない場合は警告

メッセージが出力され，スタックトレースは出力されません。また，1 行の文字数が指定した文字数を超えた場合，「at」以降の文字列の前半部分を削除して，指定された文字数分出力します。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- -XX:+HitachiOutOfMemoryStackTrace

引数

<整数値>

1024～2147483647 の範囲で整数値（単位：バイト）を指定します。範囲外の値が指定された場合は 1024 が設定されます。負の値を指定した場合はエラーとなります。

注意事項

非同期ログ出力（-XX:+JavaLogAsynchronous）が有効な場合，-XX:HitachiOutOfMemoryStackTraceLineSize に 4096 より大きな値を指定しても，4096 が設定されます。

-XX:[+|-]HitachiOutOfMemorySize（メモリサイズ出力オプション）

形式

-XX:+HitachiOutOfMemorySize

OutOfMemoryError 発生時に，要求したメモリサイズをバイト単位で出力します。

-XX:+HitachiOutOfMemoryStackTrace オプションが指定されている場合は，このオプションも設定されます。

-XX:-HitachiOutOfMemorySize

OutOfMemoryError 発生時に，要求したメモリサイズを出力しません。

説明

OutOfMemoryError 発生時に要求したメモリのサイズを出力します。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

出力例

```
[OOM][Thread: 0x00062fd0]<Tue Dec 2 16:42:39 2003>[java.lang.OutOfMemoryError :   requested
1024 bytes. (Java Heap) : 20 threads exist]
```

注意事項

次の場合，要求したメモリサイズが取り出せません。

- Java のメモリ確保機能を使用してなくて，明示的に OutOfMemoryError をスローした（Java SE クラスライブラリがスローするものを含みます）場合。

```
例:throw new OutOfMemoryError();
```

- クラスロード時の verifier が OutOfMemoryError を発生させた場合。

```
[OOM][Thread: 0x00062fd0]<Tue Dec 2 16:42:39 2003>[java.lang.OutOfMemoryError :   request
ed size unknown. (Unknown) : 10 threads exist]
```

-XX:[+|-]HitachiOutOfMemoryAbort（強制終了オプション）

形式

`-XX:+HitachiOutOfMemoryAbort`

OutOfMemoryError 発生時にメモリダンプを出力して，強制終了します。

`-XX:-HitachiOutOfMemoryAbort`

OutOfMemoryError 発生時に強制終了しません。

説明

次の原因で OutOfMemoryError が発生した場合，標準出力にメッセージを，カレントディレクトリにメモリダンプまたは core ダンプを出力して強制終了します。

- Java ヒープ不足の場合
- Metaspace 領域不足の場合
- Compressed Class Space 不足の場合
- Java SE クラスライブラリ中での C ヒープ不足の場合

なお，JavaVM 処理中に C ヒープ不足となった場合には，このオプションの指定に関係なく，強制停止します。

強制終了時の終了コード

強制終了した場合の JavaVM の終了コードは 6 です。

なお，UNIX のシェル（sh や csh など）上で実行したとき，終了コードは 0x80 が加算されて 0x86 となります。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

出力例

```
java.lang.OutOfMemoryError occurred.
JavaVM aborted because of specified -XX:+HitachiOutOfMemoryAbort options.
```

注意事項

- -XX:+HitachiOutOfMemoryStackTrace オプションが指定されている場合は，JavaVM ログファイルにスタックトレースを出力したあとに終了します。
- このオプションが指定されている場合，java.io.File.deleteOnExit メソッドや java.lang.Runtime.addShutdownHook メソッドで登録している，JavaVM 終了時の処理は実行されないで強制終了します。

-XX:[+|-]HitachiOutOfMemoryAbortThreadDump（スレッドダンプ出力オプション）

形式

-XX:+HitachiOutOfMemoryAbortThreadDump

OutOfMemoryError 発生時にスレッドダンプを出力します。

-XX:+HitachiOutOfMemoryAbort オプションが指定されている場合に，このオプションは指定できます。

-XX:-HitachiOutOfMemoryAbortThreadDump

OutOfMemoryError 発生時にスレッドダンプを出力しません。

説明

OutOfMemoryError 発生時にスレッドダンプを出力します。ただし，Java SE クラスライブラリで C ヒープ不足の場合は，スレッドダンプの出力による再度の C ヒープ不足発生を避けるため，スレッドダンプは出力しません。

スレッドダンプの出力先は環境変数 JAVACOREDIR，または-XX:+HitachiThreadDumpToStdout オプションで指定します。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- -XX:+HitachiOutOfMemoryAbort
- -XX:+HitachiThreadDump

-XX:[+|-]HitachiOutOfMemoryAbortThreadDumpWithJHeapProf（クラス別統計情報出力オプション）

形式

`-XX:+HitachiOutOfMemoryAbortThreadDumpWithJHeapProf`

-XX:+HitachiOutOfMemoryAbortThreadDump で出力するスレッドダンプログファイルにクラス別統計情報を出力します。

`-XX:-HitachiOutOfMemoryAbortThreadDumpWithJHeapProf`

-XX:+HitachiOutOfMemoryAbortThreadDump で出力するスレッドダンプログファイルにクラス別統計情報を出力しません。

説明

-XX:+HitachiOutOfMemoryAbortThreadDump で出力するスレッドダンプログファイルにクラス別統計情報を出力するかどうかを指定します。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- `-XX:+HitachiOutOfMemoryAbort`
- `-XX:+HitachiOutOfMemoryAbortThreadDump`
- `-XX:+HitachiThreadDump`

注意事項

-XX:+HitachiOutOfMemoryAbortThreadDumpWithJHeapProf オプションは OutOfMemory 発生時に出力されるスレッドダンプに，クラス別統計情報を出力するためのオプションです。G1GC を使用した場合，クラス別統計機能が使用できないため，このオプションも使用できません。G1GC 使用時にこのオプションを指定した場合，スレッドダンプは出力されますが，クラス別統計情報は出力されません。

-XX:[+|-]HitachiOutOfMemoryHandling（OutOfMemory ハンドリングオプション）

形式

`-XX:+HitachiOutOfMemoryHandling`

OutOfMemory ハンドリング機能を有効にします。

-XX:-HitachiOutOfMemoryHandling

OutOfMemory ハンドリング機能を無効にします。

説明

OutOfMemory ハンドリング機能を有効にするかどうかを指定します。

OutOfMemory ハンドリング機能は，OutOfMemory 発生時強制終了機能（-XX:+HitachiOutOfMemoryAbort）と組み合わせて使用します。OutOfMemory 発生時強制終了機能が無効（-XX:-HitachiOutOfMemoryAbort）になっている場合，OutOfMemory ハンドリング機能は無効になります。

OutOfMemory ハンドリング機能を有効にした場合，OutOfMemory 発生時に OutOfMemoryError スロー条件が判定されます。具体的には，次に示す処理を実行中に Java ヒープ不足や Metaspace 領域，Compressed Class Space 不足が原因の OutOfMemory が発生した場合に，アプリケーションサーバの実行を継続するかどうかが判定されます。

- Web コンテナ上の Web アプリケーション（Servlet／JSP）が実行中のリクエスト処理
- EJB クライアントアプリケーションから呼び出された Enterprise Bean が実行中の処理
- Message-driven Bean が実行中の処理
- Timer Service から呼び出された Enterprise Bean が実行中の処理

判定の結果，アプリケーションサーバの実行が継続される場合は，java.lang.OutOfMemoryError がスローされて，OutOfMemory が発生したリクエスト処理だけが中止されます。

JavaVM は，判定結果によって次のように動作します。ただし，Web アプリケーションで java.lang.OutOfMemoryError をキャッチしている場合は，その処理に従います。

- **OutOfMemoryError スロー条件を満たしている場合**

  java.lang.OutOfMemoryError をスローして，OutOfMemory が発生したリクエスト処理だけを中止します。
- **OutOfMemoryError スロー条件を満たしていない場合**

  OutOfMemory 発生時強制終了機能によって，JavaVM を強制終了します。

OutOfMemoryError スロー条件を次に示します。なお，OutOfMemoryError スロー条件に合致しない場合，OutOfMemory 発生時強制終了機能によって，JavaVM は強制終了します。

OutOfMemoryError スロー条件

OutOfMemory が発生した場合に，次に示す条件すべてに該当するとき，JavaVM は OutOfMemoryError スロー条件を満たしているものと判定して，異常終了しないで java.lang.OutOfMemoryError をスローします。

- Java ヒープ不足，または Metaspace 領域，Compressed Class Space 不足が原因の OutOfMemory である。

- Web コンテナ上の Web アプリケーション（Servlet／JSP）が実行中のリクエスト処理，EJB クライアントアプリケーションから呼び出された Enterprise Bean が実行中の処理，Message-driven Bean が実行中の処理，または Timer Service から呼び出された Enterprise Bean が実行中の処理で発生した OutOfMemory である。
- OutOfMemoryError スロー除外条件に該当しない。

OutOfMemoryError スロー除外条件

今回の OutOfMemory が発生した時刻から過去 1 時間以内の Java ヒープ不足が原因の OutOfMemory の発生回数と Metaspace 領域，Compressed Class Space 不足が原因の OutOfMemory の発生回数の合計値（今回の OutOfMemory を含む）が，-XX:HitachiOutOfMemoryHandlingMaxThrowCount オプション値に指定した値よりも大きい。

OutOfMemory 発生時強制終了機能と OutOfMemory ハンドリング機能の OutOfMemory 発生要因ごとの動作を次の表に示します。OutOfMemory ハンドリング機能が有効かどうかは，OutOfMemory の発生原因が Java ヒープ不足または Metaspace 領域，Compressed Class Space 不足の場合の動作に影響します。

表 5-7　OutOfMemory 発生時強制終了機能と OutOfMemory ハンドリング機能の OutOfMemory 発生要因ごとの動作

OutOfMemory 発生要因	OutOfMemory 発生時強制終了機能の対象となるかどうか (OutOfMemory 発生時の動作※1)	OutOfMemory ハンドリング機能の対象となるかどうか (OutOfMemory 発生時の動作)
Java ヒープ不足	対象になります (JavaVM は強制終了します)。	対象になります (判定結果に従って動作します※2)。
Metaspace 領域不足	対象になります (JavaVM は強制終了します)。	対象になります (判定結果に従って動作します※2)。
Compressed Class Space 不足	対象になります (JavaVM は強制終了します)。	対象になります (判定結果に従って動作します※2)。
C ヒープ不足	対象になります (JavaVM は強制終了します)。	対象になりません (JavaVM は強制終了します※3)。
Unknown	対象になりません (java.lang.OutOfMemoryError がスローされます)。	対象になりません (java.lang.OutOfMemoryError がスローされます)。
Thread Limit	対象になりません (java.lang.OutOfMemoryError がスローされます)。	対象になりません (java.lang.OutOfMemoryError がスローされます)。

注※1　OutOfMemory ハンドリング機能は無効（-XX:-HitachiOutOfMemoryHandling）の場合の動作です。

注※2　OutOfMemoryError スロー条件を満たしている場合，java.lang.OutOfMemoryError がスローされます。OutOfMemoryError スロー条件を満たしていない場合，OutOfMemory 発生時強制終了機能の処理に移行して，JavaVM は強制終了します。

注※3　OutOfMemory 発生時強制終了機能によって，JavaVM は強制終了します。

また，OutOfMemory ハンドリング機能が有効な場合，Java ヒープ不足，および Metaspace 領域，Compressed Class Space 不足が原因の OutOfMemory 発生時に OutOfMemory の発生頻度に関する情報が JavaVM ログファイルに出力されます。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- -XX:+HitachiOutOfMemoryAbort

出力形式

```
[id][Thread: thread_id]<date>[Handling: oom_count(max_oom_count)]
```

出力内容を次に説明します。

id

OMH（JavaVM ログファイル識別子）。

thread_id

スレッド ID（スレッドダンプに出力されている tid）。

date

OutOfMemory をハンドリングした日時。

oom_count

今回の OutOfMemory が発生した時刻から過去 1 時間以内の，Java ヒープ不足が原因の OutOfMemory の発生回数と Metaspace 領域，Compressed Class Space 不足が原因の OutOfMemory の発生回数の合計値（今回の OutOfMemory を含む）。

ただし，今回の OutOfMemory も含めて，1 時間以内の発生回数の合計値が 3601 を超えた場合でも，出力項目の最大値は 3601 になります。

max_oom_count

-XX:HitachiOutOfMemoryHandlingMaxThrowCount オプションに指定した値。

出力例

```
[OMH][Thread: 0x00927f48]<Tue Aug 24 19:02:19 2010>[Handling: 1(60)]
```

注意事項

- OutOfMemory ハンドリング機能は，OutOfMemory の根本原因を解決したり，OutOfMemory 発生によるアプリケーションサーバの終了を確実に回避したりする機能ではありません。また，OutOfMemory 発生後にアプリケーションサーバの実行を継続できることを確実に保証する機能でもありません。この機能は，ユーザープログラム処理の問題による突発的な OutOfMemory が発生した場合に，アプリケーションサーバの終了を一時的に防ぐための機能です。

java.lang.OutOfMemoryError は，OutOfMemory ハンドリング機能の設定に関係なく，ヒープが枯渇している場合に発生します。java.lang.OutOfMemoryError が発生した場合には，アプリケーションサーバをできるだけ速やかに再起動して回復すること，そのあとで OutOfMemory の根本原因を解決することを推奨します。

なお，java.lang.OutOfMemoryError をスローすることによって，リソースのリークやロックなどが発生し，アプリケーションサーバに予期しない動作が発生するおそれがあります。このため，この機能を有効にしている場合に OutOfMemory が発生したときには，適切なタイミングでアプリケーションサーバを再起動する運用にしてください。また，アプリケーションサーバに予期しない動作が発生することを避けたい場合は，この機能を無効にしてください。この場合，突発的な OutOfMemory が発生したときには，アプリケーションサーバは終了します。

この機能を有効にした場合に，予期しない動作になってしまったときは，アプリケーションサーバを再起動してください。また，その後の運用ではこの機能を無効にしてください。

- OutOfMemory ハンドリング機能が有効な場合，OutOfMemory 発生時強制終了機能が有効（-XX:+HitachiOutOfMemoryAbort）でも，jsp や Servlet での処理中に OutOfMemory が発生したときは，強制終了しないで Java SE の仕様どおりに java.lang.OutOfMemoryError がスローされます。

  そのため，例えば，finally 節を使用して適切にリソースを解放していないようなときには，リソースの解放漏れなどが発生することがあります。

  このような問題を避けて，従来どおり強制終了させたい場合は，OutOfMemory ハンドリング機能を無効にしてください。

-XX:HitachiOutOfMemoryHandlingMaxThrowCount（最大発生回数の設定オプション）

形式

```
-XX:HitachiOutOfMemoryHandlingMaxThrowCount=<整数値>
```

説明

OutOfMemory ハンドリング機能を有効にした場合の，Java ヒープ不足または Metaspace 領域，Compressed Class Space 不足が原因の OutOfMemory 発生回数合計値の 1 時間当たりの上限値を指定します。

このオプションで指定した値は，OutOfMemoryError スロー除外条件の判定で使用されます。

Java ヒープ不足または Metaspace 領域，Compressed Class Space 不足が原因の OutOfMemory が発生した場合，次のように処理が実行されます。

- 今回の OutOfMemory が発生した時刻から過去 1 時間以内の Java ヒープ不足が原因の OutOfMemory の発生回数と Metaspace 領域，Compressed Class Space 不足が原因の OutOfMemory の発生回数の合計値（今回の OutOfMemory を含む）が，このオプションに指定した上限値よりも大きい場合，

OutOfMemoryError はスローされません。OutOfMemory 発生時強制終了機能（-XX:+HitachiOutOfMemoryAbort）の処理に移行して，JavaVM は強制終了します。

- このオプションの値として 0 を指定した場合は，OutOfMemory が発生したときに必ず OutOfMemory 発生時強制終了機能の処理に移行して，JavaVM は強制終了します。
- このオプションの値として 3600 を指定した場合は，OutOfMemoryError スロー除外条件の判定が実行されません。この場合，OutOfMemory が発生したときに OutOfMemoryError スロー除外条件以外の OutOfMemoryError スロー条件を満たしている場合，必ず java.lang.OutOfMemoryError がスローされます。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- -XX:+HitachiOutOfMemoryHandling

引数

〈整数値〉

Java ヒープ不足または Metaspace 領域，Compressed Class Space 不足が原因の OutOfMemory 発生回数合計値の 1 時間当たりの上限値を指定します。

0～3600（3601～2^{63}-1 の場合は 3600 として扱います）。

注意事項

自然数以外の値が指定された場合は，定義していないオプションを指定した場合と同様の動作になります。

-XX:[+|-]HitachiJavaClassLibTrace（クラスライブラリのスタックトレース出力オプション）

形式

-XX:+HitachiJavaClassLibTrace

クラスライブラリのスタックトレースを出力します。

-XX:-HitachiJavaClassLibTrace

クラスライブラリのスタックトレースを出力しません。

説明

次に示すシステム全体に影響を与えるメソッドが使用された場合，これらの API のスタックトレースを，JavaVM ログファイルへ出力します。

- java.lang.System.gc

- java.lang.System.exit
- java.lang.System.runFinalizersOnExit
- java.lang.Runtime.exit
- java.lang.Runtime.halt
- java.lang.Runtime.runFinalizersOnExit

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

出力形式

```
[id] [Thread:thread_id]<date>
[id] [Thread:thread_id] stack_trace
```

出力内容を次に説明します。

id：

CLT（JavaVM ログファイル識別子）。

thread_id：

スレッド ID（スレッドダンプに出力されている tid）。

date：

クラスライブラリ使用日時。

stack_trace：

スタックトレース。

出力例 1

```
[CLT][Thread: 0x00062fd0]<Mon Sep 27 12:10:03 2004>
[CLT][Thread: 0x00062fd0] at at java.lang.Shutdown.halt0(Native Method)
[CLT][Thread: 0x00062fd0] at java.lang.Shutdown.halt(Shutdown.java:145)
[CLT][Thread: 0x00062fd0] - locked <0x100101a0> (a java.lang.Shutdown$Lock)
[CLT][Thread: 0x00062fd0] at java.lang.Shutdown.exit(Shutdown.java:222)
[CLT][Thread: 0x00062fd0] - locked <0x1413c0a0> (a java.lang.Class)
[CLT][Thread: 0x00062fd0] at java.lang.Terminator$1.handle(Terminator.java:35)
[CLT][Thread: 0x00062fd0] at sun.misc.Signal$1.run(Signal.java:195)
[CLT][Thread: 0x00062fd0] at java.lang.Thread.run(Thread.java:534)
```

出力例 2

```
[CLT][Thread: 0x009c4000]<Tue Oct 09 15:36:18 2012>
[CLT][Thread: 0x009c4000] at java.lang.Runtime.outputJavaClassLibTrace(Native Method)
[CLT][Thread: 0x009c4000] at java.lang.Runtime.runFinalizersOnExit(Runtime.java:378)
[CLT][Thread: 0x009c4000] at java.lang.System.runFinalizersOnExit(System.java:978)
[CLT][Thread: 0x009c4000] at Program.main(Program.java:8)
```

-XX:HitachiJavaClassLibTraceLineSize（クラスライブラリのスタックトレース行サイズ指定オプション）

形式

```
-XX:HitachiJavaClassLibTraceLineSize=<整数値>
```

説明

-XX:+HitachiJavaClassLibTrace オプションを指定した場合に出力される，クラスライブラリのスタックトレースの 1 行の文字数をバイト数で指定します。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- -XX:+HitachiJavaClassLibTrace

引数

<整数値>

1024～2147483647 の範囲で整数値（単位：バイト）を指定します。範囲外の値が指定された場合は 1024 が設定されます。

出力例

```
[CLT][Thread: 0x00286c58]<Thu Oct 21 14:56:24 2004>
[CLT][Thread: 0x00286c58] at java.lang.Runtime.gc(Native Method)
[CLT][Thread: 0x00286c58] at java.lang.System.gc(System.java:737)
[CLT][Thread: 0x00286c58] at mple.func_012345678～省略～xyz(Sample.java:9)
[CLT][Thread: 0x00286c58] at Sample.main(Sample.java:5)
```

1 行の文字数が指定したバイト数を超える場合，「at」以降の文字列の前半部分が削除されます。出力例の場合，4 行目の Sample の前半が削除されて mple となります。

-XX:[+|-]HitachiLocalsInThrowable（例外発生時のローカル変数情報収集オプション）

形式

-XX:+HitachiLocalsInThrowable

スタックトレース中のメソッドのローカル変数情報を出力します。

ただし，java.lang.StackOverflowError 発生時には，このオプションは無視されます。

-XX:-HitachiLocalsInThrowable

スタックトレース中のメソッドのローカル変数情報を出力しません。

説明

java.lang.Throwable.fillInStackTrace メソッド実行時に，スタックトレース中のメソッドのローカル変数情報を収集します。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

出力形式

```
    locals:
      name: <name>
      type: <type>
      value: <value>
...
```

出力内容を次に示します。

- 1 行目に「locals:」という見出しが出力されます。
- 2 行目以降は，収集可能であった各ローカル変数について，次の情報が 1 行ずつ出力されます。
  1. 変数名
  2. 型名（基本型名，クラス名または配列型名）
  3. 変数値を表現する文字列

なお，各ローカル変数の出力内容は，空行で区切られています。

<name>：

ローカル変数名。

メソッドに渡される引数の場合は，変数名に続いて[arg***]（***は引数番号）が表示されます。

<type>：

ローカル変数の型名（基本型名，クラス名または配列型名）。

<value>：

ローカル変数の値を表現する文字列。

- 基本型：
  値をそのまま文字列化したもの
- クラスまたは配列型：
  変数値が null の場合：(null)
  それ以外の場合：<オブジェクトの存在するアドレス>

値表現の最大文字列長は 64 です。これを超える場合は 64 文字目までを出力したあと，「...」という文字列が出力されます。クラスまたは配列型の場合，次の追加オプションを指定することでより詳細な表現が追加できます。

- -XX:+HitachiLocalsSimpleFormat
- -XX:+HitachiTrueTypeInLocals
- -XX:HitachiCallToString

出力例

Java プログラム例 1 を使用した出力例を，次に示します。

すべてのローカル変数情報が出力される場合

```
  at Example1.method(Example1.java:15)
    locals:
      name: this
      type: Example1
      value: <0x922f42d0>

      name: l1 [arg1]
      type: int
      value: 1

      name: l2 [arg2]
      type; char
      value: 'Q'

      name: l3 [arg3]
      type: java.lang.Object
      value: <0xaf112f08>

      name: l4
      type: float
      value: 4.000000

      name: l5
      type: boolean
      value: true

      name: l6
      type: double
      value: 1.79769E+308

      name: l7
      type: java.lang.Object[]
      value: <0x922f42d8>

  at Example1.main(Example1.java:5)
    locals:
...
```

ローカル変数情報が存在しない場合

- -g オプションまたは-g:vars オプションを付加しないで class ファイルを生成した場合
- -g オプションまたは-g:vars オプションを付加して生成した class ファイルの，native メソッドの場合

```
  at Example1.method(Example1.java:15)
    locals:
      name: this
      type: Example1
      value: <0x922f42d0>

      name: [arg1]
      type: int
      value: 1

      name: [arg2]
      type; char
      value: 'Q'

      name: [arg3]
      type: java.lang.Object
      value: <0xaf112f08>

  at Example1.main(Example1.java:5)
    locals:
...
```

注意事項

- ローカル変数情報を完全に収集するためには，javac で class ファイルを生成する際に，-g オプションまたは-g:vars オプションを付加して class ファイル内にローカル変数情報を埋め込んでおく必要があります。-g オプションまたは-g:vars オプションを付加しないで作成された class ファイルについては，収集可能な範囲でローカル変数情報が出力されます。
- -g オプションまたは-g:vars オプションを付加して生成された class ファイルでも，native メソッドの場合はローカル変数情報が存在しません。
- JIT コンパイラがメソッドを JIT コンパイルする際，最適化の一環として，不要と判断したローカル変数を除去することがあります。

  （例）int not_used = 12345 といった宣言および初期化以降未使用のローカル変数

  この場合，例外発生時点のローカル変数情報には，次の値が出力されます。

型名	出力情報
boolean 型	false
char 型	'¥0'
byte 型 short 型 int 型 long 型 float 型 double 型	0

型名	出力情報
クラス型 配列型	(null)

- 制御構造が複雑で行数が多いメソッドのローカル変数情報を出力する場合，解析に時間が掛かるため，例外発生時の例外オブジェクト生成処理に時間が掛かることがあります。
- java.lang.Thread クラスの getStackTrace メソッドを使用して取得した，カレントスレッドのスタックトレースにローカル変数情報を出力するためには，例外発生時のスタックトレースにローカル変数情報を出力する，例外発生時のローカル変数情報収集オプション（-XX:+HitachiLocalsInThrowable）が必要になります。

-XX:[+|-]HitachiLocalsInStackTrace（スレッドダンプ出力時のローカル変数出力オプション）

形式

`-XX:+HitachiLocalsInStackTrace`

スレッドダンプ出力時のスタックトレースに，ローカル変数情報を出力します。

`-XX:-HitachiLocalsInStackTrace`

スレッドダンプ出力時のスタックトレースに，ローカル変数情報を出力しません。

説明

スレッドダンプ出力時のスタックトレースに，各メソッドのローカル変数情報を追加して出力します。ローカル変数情報の出力内容については，「-XX:[+|-]HitachiLocalsInThrowable（例外発生時のローカル変数情報収集オプション）」を参照してください。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

出力例

Java プログラム例 2 を使用した出力例を，次に示します。

- -XX:+HitachiLocalsSimpleFormat オプションおよび-XX:+HitachiTrueTypeInLocals オプションが指定されている場合

```
"main" prio=1 tid=0xb6e88d20 nid=0xb7492080 runnable [bfffb000..bfffb474]
        at Example2.method(Example2.java:15)
        - locked <0xab040550> (a Example2)
          locals:
            (Example2) this = <0xab040550> (Example2)
            (int) l1 = 1
            (float) l2 = 2.000000
```

```
        (java.lang.String) l3 = <0xaf112cc0> (java.lang.String)
        (java.lang.Character) l4 = <0xab040698> (java.lang.Character)
        (java.lang.Object) l5 = <0xab0407c8> (java.lang.Thread)
        (java.lang.Object[]) l6 = <0xab0408b8> (java.lang.Thread[])
    at Example2.main(Example2.java:4)
      locals:
        (java.lang.String[]) args [arg1] = <0xab040540> (java.lang.String[])
        (Example2) e2 = <0xab040550> (Example2)
```

注意事項

- ローカル変数情報を完全に収集するためには，javac で class ファイルを生成する際に，-g オプションまたは-g:vars オプションを付加して class ファイル内にローカル変数情報を埋め込んでおく必要があります。-g オプションまたは-g:vars オプションを付加しないで作成された class ファイルについては，収集可能な範囲でローカル変数情報が出力されます。
- 一般にスタックトレース情報の収集を行うスレッドと収集対象のスレッドは一致しません。このため，情報収集を行うためには，対象スレッドを停止させる必要があり，toString メソッドを呼び出すことはできません。このため，-XX:HitachiCallToString オプションの指定は無効になります。
- 制御構造が複雑で行数が多いメソッドのローカル変数情報を出力する場合，解析に時間が掛かるため，拡張スレッドダンプの出力やスレッドスタックトレースの取得に時間が掛かることがあります。

-XX:[+|-]HitachiLocalsSimpleFormat（ローカル変数情報の出力フォーマット変更オプション）

形式

`-XX:+HitachiLocalsSimpleFormat`

ローカル変数情報出力を，簡易フォーマットで出力します。

`-XX:-HitachiLocalsSimpleFormat`

ローカル変数情報出力を，通常フォーマットで出力します。

説明

ローカル変数情報の出力フォーマットを，1 変数 1 行で出力する簡易出力フォーマットに変更します。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- `-XX:+HitachiLocalsInThrowable`
- `-XX:+HitachiLocalsInStackTrace`

出力形式

```
    locals:
      (type) name = value
      (type) name = value
...
```

type，name および value の出力内容については，「-XX:[+|-]HitachiLocalsInThrowable（例外発生時のローカル変数情報収集オプション）」を参照してください。

出力例

Java プログラム例 1 を使用した出力例を，次に示します。

```
  at Example1.method(Example1.java:15)
    locals:
      (Example1) this = <0x922f42d0>
      (int) l1 [arg1] = 1
      (char) l2 [arg2] = 'Q'
      (java.lang.Object) l3 [arg3] = <0xaf112f08>
      (float) l4 = 4.000000
      (boolean) l5 = true
      (double) l6 = 1.79769E+308
      (java.lang.Object[]) l7 = <0x922f42d8>
  at Example1.main(Example1.java:5)
    locals:
...
```

-XX:[+|-]HitachiTrueTypeInLocals（ローカル変数情報の実型名出力オプション）

形式

`-XX:+HitachiTrueTypeInLocals`

ローカル変数情報に，実際のオブジェクト型名を出力します。

`-XX:-HitachiTrueTypeInLocals`

ローカル変数情報に，実際のオブジェクト型名を出力しません。

説明

ローカル変数情報収集時に，クラスまたは配列型のローカル変数について，実際に代入されているオブジェクトの型名を出力します。変数値を表現する文字列の後ろの丸括弧内に表示されます。

なお，ローカル変数に格納されているクラスまたは配列型のオブジェクトが JavaVM 内部のオブジェクトの場合は，"internal type"と出力されます。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- -XX:+HitachiLocalsInThrowable
- -XX:+HitachiLocalsInStackTrace

出力例

Java プログラム例 3 を使用した出力例を，次に示します。

- -XX:+HitachiLocalsSimpleFormat オプションおよび-XX:HitachiCallToString=full が指定されている場合

```
  at Example3.method(Example3.java:18)
    locals:
      (Example3) this = <0xaa07db58> "I am an Example3 instance." (Example3)
      (java.lang.String) l1 = <0xae173a28> "local 1" (java.lang.String)
      (java.lang.StringBuffer) l2 = <0xaa07dca0> "local 1 + local 2" (java.lang.StringBuf
fer)
      (java.lang.Boolean) l3 = <0xaa07de18> "false" (java.lang.Boolean)
      (java.lang.Character) l4 = <0xaa07df68> "X" (java.lang.Character)
      (java.lang.Long) l5 = <0xaa07e078> "-9223372036854775808" (java.lang.Long)
      (java.lang.Object) l6 = <0xaa07e1a8> "Thread[Thread-0,5,main]" (java.lang.Thread)
      (java.lang.Object[]) l7 = <0xaa07e298> "[Ljava.lang.Thread;@26e431" (java.lang.Thre
ad[])
  at Example3.main(Example3.java:4)
    locals:
...
```

-XX:HitachiCallToString（ローカル変数情報出力オプション）

形式

-XX:HitachiCallToString=<適用範囲>

説明

<適用範囲>に該当するクラスのローカル変数オブジェクトに対して取得した String オブジェクトの文字列を，変数値を表現する文字列として出力します。

なお，ローカル変数に格納されているオブジェクトがない場合，または JavaVM 内部のオブジェクトの場合は，出力されません。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- -XX:+HitachiLocalsInThrowable

引数

〈適用範囲〉

minimal または full を指定します。

minimal：

java.lang パッケージ内の次に示すクラスが対象になります。

- String
- StringBuffer
- Boolean
- Byte
- Character
- Short
- Integer
- Long
- Float
- Double

なお，空文字（""）を指定した場合も，minimal と同じです。

full：

すべてのクラスおよび配列型が対象になります。

出力例

Java プログラム例 3 を使用した出力例（簡易出力フォーマット）を，次に示します。

-XX:HitachiCallToString=minimalの場合

```
  at Example3.method(Example3.java:18)
    locals:
      (Example3) this = <0xaa07db58>
      (java.lang.String) l1 = <0xae173a28> "local 1"
      (java.lang.StringBuffer) l2 = <0xaa07dca0> "local 1 + local 2"
      (java.lang.Boolean) l3 = <0xaa07de18> "false"
      (java.lang.Character) l4 = <0xaa07df68> "X"
      (java.lang.Long) l5 = <0xaa07e078> "-9223372036854775808"
      (java.lang.Object) l6 = <0xaa07e1a8>
      (java.lang.Object[]) l7 = <0xaa07e298>
  at Example3.main(Example3.java:4)
    locals:
...
```

-XX:HitachiCallToString=fullの場合

```
  at Example3.method(Example3.java:18)
    locals:
      (Example3) this = <0xaa07db58> "I am an Example3 instance."
      (java.lang.String) l1 = <0xae173a28> "local 1"
      (java.lang.StringBuffer) l2 = <0xaa07dca0> "local 1 + local 2"
      (java.lang.Boolean) l3 = <0xaa07de18> "false"
      (java.lang.Character) l4 = <0xaa07df68> "X"
      (java.lang.Long) l5 = <0xaa07e078> "-9223372036854775808"
      (java.lang.Object) l6 = <0xaa07e1a8> "Thread[Thread-0,5,main]"
      (java.lang.Object[]) l7 = <0xaa07e298> "[Ljava.lang.Thread;@26e431"
  at Example3.main(Example3.java:4)
    locals:
...
```

注意事項

- ローカル変数情報を完全に収集するためには，javac で class ファイルを生成する際に，-g オプションまたは-g:vars オプションを付加して class ファイル内にローカル変数情報を埋め込んでおく必要があります。-g オプションまたは-g:vars オプションを付加しないで作成された class ファイルについては，収集可能な範囲でローカル変数情報が出力されます。
- このオプションを指定して，AWT または Swing を利用した Java のプログラムを実行する場合，<適用範囲>には「minimal」を指定してください。<適用範囲>に「full」を指定すると，プログラムが正しく動作しません。
- ユーザプログラムでの例外発生時の原因調査時に限り，<適用範囲>に「full」を指定してください。この機能は，例外オブジェクト生成時，ローカル変数オブジェクトの toString()メソッドを呼び出す仕様です。toString()メソッドの呼び出しによって，原因調査に有用な情報が得られる反面，本来とは異なるタイミングで toString()メソッドを呼び出すことで，製品やユーザプログラムが正しく動作しない場合があるためです。

-XX:[+|-]HitachiFullCore（システムリソース解除オプション）

形式

-XX:+HitachiFullCore

システムリソース RLIMIT_CORE を強制的に最大値に引き上げて，障害発生時にユーザーリミットを無視して core ファイルを作成します。

-XX:-HitachiFullCore

システムリソース RLIMIT_CORE は変更しません。

説明

システムリソース RLIMIT_CORE の設定を変更するかどうかを指定します。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

-XX:HitachiJITCompileMaxMemorySize（JIT コンパイル時の確保メモリ上限値指定オプション）

形式

```
-XX:HitachiJITCompileMaxMemorySize=<整数値>
```

説明

JIT コンパイル時に確保する C ヒープのメモリサイズの上限値を指定します。JIT コンパイルによって確保する C ヒープのメモリが指定値を超えた場合，JavaVM ログファイルにログを出力するとともに，以降の JIT コンパイルを抑制します。JIT コンパイル処理の対象となっていた Java メソッドは，以降，インタプリタでだけ実行されるようになります。なお，JIT コンパイルが抑制された場合でも，JavaVM は強制終了しないで，処理を続行します。

0 を指定した場合，JIT コンパイル時に確保するメモリは上限値で制限されません。

なお，JIT コンパイルは JavaVM の内部スレッド（JIT コンパイラスレッド）で実行されます。JIT コンパイラスレッドは 2 つあるため，このオプションで指定した上限値を 2 で割った値が，1 つの JIT コンパイラスレッドに対する上限値となります。

また，このオプションで上限値を設定した場合も，実際に使用できる C ヒープがそれよりも少ないときには，上限値に達する前に C ヒープ不足が発生することがあります。この場合，JavaVM は強制終了します。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- -server

引数

<整数値>

JIT コンパイル時に確保するメモリの上限値を指定します。単位はバイトです。単位文字として，「k」（キロ），「m」（メガ），「g」（ギガ）も指定できます。指定できる範囲は次のとおりです。範囲外の値を指定した場合は，0 が指定されます。

- 0～2^{64}-1（18446744073709551615）

0 を指定した場合は，JIT コンパイル時に確保するメモリは上限値で制限されません。JIT コンパイル中に C ヒープ不足が発生した場合，JavaVM は強制終了します。

出力形式

```
[id][Thread: thread_id]<date>["thread_name" exceeded max memory size.]
[current_sizeK->new_sizeK/limit_size_per_threadK/limit_sizeK]
[compile_target][byte_code_size]
```

出力内容を次に説明します。

id

JMS（JavaVM ログファイル識別子）。

thread_id

JIT コンパイルを抑制した JIT コンパイラスレッドのスレッド ID。

date

JIT コンパイルを抑制した日時。

-XX:+HitachiOutputMilliTime オプションが指定された場合，ミリ秒単位で出力されます。

thread_name

JIT コンパイルを抑制した JIT コンパイラスレッドのスレッド名。

current_size

JIT コンパイルを抑制した JIT コンパイラスレッドの，現在のメモリ確保サイズ（単位：キロバイト）。

new_size

JIT コンパイルを抑制した JIT コンパイラスレッドの，現在のメモリ確保サイズと追加で確保しようとしたサイズの合計値（単位：キロバイト）。

limit_size_per_thread

1 つの JIT コンパイラスレッドの上限値（単位：キロバイト）。

limit_size

JIT コンパイラスレッド全体の上限値（単位：キロバイト）。

compile_target

JIT コンパイル処理の対象となっていた Java メソッド。

byte_code_size

JIT コンパイル処理の対象となっていた Java メソッドのバイトコードのサイズ（単位：バイト）。

出力例

オプションとして「-XX:HitachiJITCompileMaxMemorySize=536870912」を指定して，1 つの JIT コンパイラスレッドに対して 262145 キロバイトの C ヒープを確保した場合に，上限値に達したときの出力例を示します。

```
[JMS][Thread: 0x03bf1150]<Wed Feb 24 14:33:58 2010>["CompilerThread0" exceeded  max memory s
ize.][262143K->262145K/262144K/524288K][test1.func][213]
```

注意事項

- JIT コンパイル時に確保するメモリが，このオプションで指定した上限値を超えて，JIT コンパイルが抑制されると，アプリケーションのスループットが低下します。
- 自然数以外の値が指定された場合は，定義していないオプションを指定した場合と同様の動作になります。

-XX:[+|-]JITCompilerContinuation（JIT コンパイラ稼働継続機能オプション）

形式

`-XX:+JITCompilerContinuation`

JIT コンパイラ稼働継続機能を有効にします。

`-XX:-JITCompilerContinuation`

JIT コンパイラ稼働継続機能を無効にします。

説明

JIT コンパイラ稼働継続機能を有効にするかどうかを指定します。

JIT コンパイルがアプリケーションを構成するメソッドの論理矛盾で失敗しても，アプリケーションを正常に継続できるようにするためには，この機能を有効にすることをお勧めします。

JIT コンパイラ稼働継続機能を有効にした場合，JIT コンパイルがアプリケーションを構成するメソッドの論理矛盾で失敗しても，JavaVM は JavaVM ログファイルにこの機能のログを出力して，処理を続行します。この場合，JIT コンパイルに失敗したメソッドでは，以降のコンパイルはインタプリタ方式で実行されるようになります。アプリケーションを構成しているメソッドのうち，JIT コンパイルに失敗したメソッド以外のメソッドは JIT コンパイルで実行されるため，アプリケーションは正常に継続されます。ただし，JIT コンパイルが 6 回以上失敗すると，JavaVM はエラーリポートファイル，およびメモリダンプまたは core ダンプを出力して，強制終了します。

JIT コンパイラ稼働継続機能を無効にした場合，JIT コンパイルがアプリケーションを構成するメソッドの論理矛盾で失敗すると，JavaVM はエラーリポートファイル，およびメモリダンプまたは core ダンプを出力して，強制終了します。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

前提オプション

- `-server`
- -XX:+HitachiVerboseGC

出力形式

```
[id]
```

注

[id]以降には JIT コンパイラ稼働継続機能のログが出力されます。

出力内容を次に説明します。

id：

JCC（JavaVM ログファイル識別子）。

出力例

```
[JCC][Thread: 0x05432c00]<Thu Nov 15 17:10:40 2012>[Method: jit_sample.func()V][Fail: 3][JIT
CT: 1]
[JCC][Thread: 0x05432c00][PC: 0x083aff9a][Lib: D:¥work¥jdk¥bin¥server¥jvm.dll+0x3aff9a][VM:
Java HotSpot(TM) Server VM (20.8-b03-CDK0950-20121115 mixed mode windows-x86 )]
[JCC][Thread: 0x05432c00][EAX=0x00000000, EBX=0x00618128, ECX=0x00000000, EDX=0x05485340]
[JCC][Thread: 0x05432c00][ESP=0x0566d3c0, EBP=0x0566d3c4, ESI=0x00618278, EDI=0x00000000]
[JCC][Thread: 0x05432c00][EIP=0x083aff9a, EFLAGS=0x00010202]
[JCC][Thread: 0x05432c00][siginfo: read 0x00000000]
[JCC][Thread: 0x05432c00][Unlock: MethodCompileQueue_lock]
[JCC][Thread: 0x05432c00][NewJITCT: 0x05438800][JITCT: 2]
[JCC][Thread: 0x05432c00][Free: "ResourceArea" 524288 bytes.]
[JCC][Thread: 0x05432c00][stop]
[JCC][Fail: 1][date: Thu Nov 15 10:10:40 2012][Method: jit_sample.func1(Ljava/lang/String;)V
][PC: 0x083ff00a][Lib: D:¥work¥jdk¥bin¥server¥jvm.dll+0x3ff00a]
[JCC][Fail: 2][date: Thu Nov 15 11:11:16 2012][Method: jit_sample.func2()V][PC: 0x083afe3a][
Lib: D:¥work¥jdk¥bin¥server¥jvm.dll+0x3afe3a]
```

-XX:[+|-]UseCompressedOops（圧縮オブジェクトポインタ機能で使用する Java オプション）

形式

-XX:+UseCompressedOops

圧縮オブジェクトポインタ機能を有効にします。

-XX:-UseCompressedOops

圧縮オブジェクトポインタ機能を無効にします。

説明

圧縮オブジェクトポインタ機能の有効，無効を指定します。圧縮オブジェクトポインタ機能は，Java オブジェクトのサイズを圧縮して管理することで，JavaVM 実行時の Java ヒープ領域の使用サイズを削減します。

この機能を有効にするには，次の条件を満たしていることが前提となります。

- Java ヒープ領域，Metaspace 領域の指定サイズの合計値が 32 ギガバイト未満であること

JavaVM 起動時，Java ヒープ領域，Metaspace 領域の指定サイズの合計値が，32 ギガバイト以上の場合は，JavaVM は次のメッセージを標準出力に出力し，圧縮オブジェクトポインタ機能を無効とします。

```
Java HotSpot(TM) 64-Bit Server VM warning: Max heap size too large for Compressed Oops
```

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

-XX:HitachiThreadLimit（スレッド数の上限値指定オプション）

形式

```
-XX:HitachiThreadLimit=<整数値>
```

説明

スレッド数の上限値を指定します。スレッド数の上限値が指定値を超えた場合に，OutOfMemoryError 例外がスローされます。ただし，JavaVM が起動する前に上限値を超えた場合，例外はスローされません。また，スレッド数には，アプリケーションが JNI などを使用して JavaVM の管理外の範囲で作成したスレッド数は含みません。0 を指定した場合，上限値は設定されません。

なお，-XX:+HitachiOutOfMemoryAbort オプションと同時に指定した場合，このオプションの機能によって OutOfMemoryError 例外がスローされた場合，JavaVM の強制終了は実行されません。

また，-XX:+HitachiOutOfMemoryStackTrace オプションと同時に指定した場合，例外メッセージが JavaVM ログファイルに出力されます。

デフォルト値

マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

引数

<整数値>

スレッド数の上限値を 0～2147483647 の範囲で整数値（単位：スレッド数）を指定します。範囲外の値が指定された場合は 0 が設定されます。負の値を指定した場合はエラーとなります。

出力形式

```
Could not create ”name” thread. Threadlimit Exceeded. num threads exist.
```

このメッセージは，java.lang.Throwable.getMessage()で取得できます。また，メッセージ全体が半角128 文字以上になった場合は，メッセージの末尾が省略されます。

name

作成に失敗したスレッド名。

num

現時点でのスレッド数。

-XX:+HitachiOutOfMemoryStackTrace オプションが指定されている場合，JavaVM ログファイルに例外情報とスタックトレースを出力できます。例外発生要因種別は，"Thread Limit"となります。例外発生要因種別については，「-XX:[+|-]HitachiOutOfMemoryCause（例外発生要因種別出力オプション）」を参照してください。

出力例

```
Could not create ”Thread-1” thread. Threadlimit Exceeded. 9 threads exist.
```

この例は，main スレッドによって java.lang.Thread.start が呼び出され，この機能によって例外をスローした場合に，java.lang.Throwable.getMessage()でメッセージを取り出したメッセージの例です。

注意事項

上限値に小さな値を設定すると，起動前に例外をスローします。アプリケーションサーバが使用するスレッド数より小さな値を設定しないでください。

アプリケーションサーバが使用するスレッド数については，マニュアル「uCosminexus Application Runtime ユーザーズガイド」のリソースの見積もりに関する説明を参照してください。

5.3　日立 JavaVM で使用するプロパティ

日立 JavaVM で使用するプロパティの一覧を，次の表に示します。

表 5-8　日立 JavaVM で使用するプロパティの一覧

分類	オプション名称	概要	関連情報
日立 JavaVM で使用するプロパティ	JP.co.Hitachi.soft.jvm.autofinalizer	ファイナライズ滞留解消機能の有効または無効を設定します。	「4.13　ファイナライズ滞留解消機能」

JP.co.Hitachi.soft.jvm.autofinalizer

形式

```
JP.co.Hitachi.soft.jvm.autofinalizer={true|false}
```

説明

ファイナライズ滞留解消機能の有効，無効を指定します。

true，false 以外の値が設定された場合，デフォルト値になります。

デフォルト値

```
JP.co.Hitachi.soft.jvm.autofinalizer=true
```

出力例

- ファイナライズ処理の滞留を検知して，ファイナライズ処理監視スレッドを新たに生成する場合

```
# FinalizerWatcherThread: Create: create secondary finalizer thread.
[queue length = 128] <Mon May 26 18:00:36 JST 2008>
```

- 生成したファイナライズ処理監視スレッドが終了した場合

```
# FinalizerWatcherThread: Finish: secondary finalizer thread is finished.
<Mon May 26 20:12:26 JST 2008>
```

5.4 日立 JavaVM で指定できる Java HotSpot VM のオプション

日立 JavaVM の利用時に指定できる Java HotSpot VM のオプション，およびオプションを指定する際の注意事項について説明します。

指定できる Java HotSpot VM のオプションを次の表に示します。

各オプションのデフォルト値は，マニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。

表 5-9 指定できる Java HotSpot VM のオプション

オプション名	内容	指定可能値
-D<property>	JavaVM のシステムプロパティを指定します。	入力は制限されません。
-agentlib:<libname>[=<options>]	ネイティブエージェントライブラリ<libname>をロードします。	入力は制限されません。
-verbose:<情報種別>	<情報種別>に指定した情報を出力します。<情報種別>に指定できる値を示します。 class： クラスがロードされるたびにクラスに関する情報を出力します。 gc： GC イベントが発生するたびに報告します。 jni： ネイティブメソッドの使用およびそのほかの Java Native Interface（JNI）アクティビティに関する情報を報告します。	指定できる文字列を次に示します。 • class • gc • jni
-Xloggc:<ファイル>	-verbose:gc と同様に GC イベントが発生するたびに報告しますが，そのデータを<ファイル>に記録します。-verbose:gc を指定したときに報告される情報のほかに，報告される各イベントの先頭に，最初の GC イベントからの経過時間（秒単位）が付け加えられます。	入力は制限されません。
-Xms<size>	Java ヒープの初期サイズを設定します。	自然数の値を次に示す単位を使って指定します。 • キロ「k」 • メガ「m」 • ギガ「g」 • テラ「t」 なお，大文字・小文字は区別されません。
-Xmx<size>	Java ヒープの最大サイズを設定します。	自然数の値を次に示す単位を使って指定します。 • キロ「k」 • メガ「m」

オプション名	内容	指定可能値
		• ギガ「g」 • テラ「t」 なお，大文字・小文字は区別されません。
-Xmn<size>	New 領域の初期値および最大値を設定します。	自然数の値を次に示す単位を使って指定します。 • キロ「k」 • メガ「m」 • ギガ「g」 • テラ「t」 なお，大文字・小文字は区別されません。
-Xss<size>	1 スタック領域の最大サイズを設定します。	自然数の値を次に示す単位を使って指定します。 • キロ「k」 • メガ「m」 • ギガ「g」 • テラ「t」 なお，大文字・小文字は区別されません。
-Xprof	このオプションを指定した場合，実行中のプログラムのプロファイルを生成し，プロファイリングデータを標準出力に出力します。このオプションは，プログラム開発用のユーティリティとして提供されています。本番稼働システムでの使用を目的としたものではありません。	−
-Xrunhprof[:<suboption>=<value>,...]	CPU，ヒープ，またはモニタのプロファイリングを有効にします。-Xrunhprof の後ろにコロン「:」を指定して「<suboption>=<value>」を記述します。「<suboption>=<value>」はコンマ「,」で区切って複数指定できます。 サブオプションとそのデフォルト値のリストを取得するには，コマンド java-Xrunhprof:help を実行します。	任意の文字列を指定します。 <suboption>には「=」および「,」は指定できません。 また，<value>には「,」は指定できません。
-XX:NewRatio=<value>	New 領域に対する Tenured 領域の割合を指定します。 <value>が 2 の場合は，New 領域と Tenured 領域の割合が，1:2 になります。 「New 領域の使用中サイズ≧Tenured 領域の空き領域サイズ」になると FullGC が発生します。このオプションに 1 を設定すると，FullGC が多発するので注意してください。	自然数の値を指定します。
-XX:MetaspaceSize=<size>	Metaspace 領域の初期サイズを指定します。	自然数の値を次に示す単位を使って指定します。 • キロ「k」

オプション名	内容	指定可能値
		• メガ「m」 • ギガ「g」 • テラ「t」 なお，大文字・小文字は区別されません。
-XX:MaxMetaspaceSize=<size>	Metaspace 領域の最大サイズを指定します。	自然数の値を次に示す単位を使って指定します。 • キロ「k」 • メガ「m」 • ギガ「g」 • テラ「t」 なお，大文字・小文字は区別されません。
-XX:CompressedClassSpaceSize	Compressed Class Space の最大サイズを指定します。	自然数の値を次に示す単位を使って指定します。 • キロ「k」 • メガ「m」 • ギガ「g」 なお，大文字・小文字は区別されません。
-XX:SurvivorRatio=<value>	New::Survivor 領域の From 空間と To 空間に対する New::Eden 領域の割合を指定します。 <value>に 8 を設定した場合は，New::Eden 領域，From 空間，To 空間の割合が，8:1:1 になります。	自然数の値を指定します。
-XX:TargetSurvivorRatio=<value>	GC 実行後の New::Survivor 領域内で Java オブジェクトが占める割合の目標値（0～100（単位：%））を指定します。	自然数の値を指定します。
-XX:MaxTenuringThreshold=<value>	CopyGC 実行時に，From 空間と To 空間で Java オブジェクトを入れ替える回数のしきい値を指定します。指定した回数を超えて入れ替え対象になった Java オブジェクトは，Tenured 領域に移動されます。 このオプションの有効範囲は，0～デフォルト値です。範囲外の値を指定した場合，しきい値を超えた場合に Tenured 領域へ移動する機能は無効になります。	自然数の値を指定します。
-XX:[+|-]UseSerialGC※	-XX:+UseSerialGC SerialGC を実行します。 -XX:-UseSerialGC SerialGC を実行しません。	指定できる文字を次に示します。 • プラス「+」 マイナス「-」
-XX:[+|-]UseG1GC	-XX:+UseG1GC G1GC を実行します。	指定できる文字を次に示します。 • プラス「+」

オプション名	内容	指定可能値
	-XX:-UseG1GC G1GC を実行しません。	マイナス「-」
-XX:ParallelGCThreads	G1GC の Evacuation を並列実行するスレッドの数を指定します。-XX:+UseG1GC を指定したときに有効になります。なお，このオプションは，デフォルト値で使用することを推奨します。変更する場合は，実際に JavaVM を動作させて測定した値を基に，最適な値を算出してから変更してください。 JavaVM 起動時に，このオプションに指定した数の Evacuation 処理用のスレッドが作成され，処理が実行されます。 このオプションの指定を省略した場合のデフォルト値は次のとおりです。 • 実行環境の論理 CPU 数が 8 以下の場合は，CPU 数になります。 • 実行環境の論理 CPU 数が 9 以上の場合は，「8 ＋ (CPU 数－8)×(5÷8)」（小数点以下は切り捨て）の値になります。 例えば，4CPU の場合は，「ParallelGCThreads ＝ 4」でデフォルト値は 4 になります。また，72CPU の場合は，「ParallelGCThreads ＝ 8 ＋(72－8)×(5÷8)＝ 48」で 48 になります。 このオプションに 0 を指定した場合は，デフォルト値が使用されます。 指定時には，次の点に注意してください。 • このオプションの値を大きくすると，Evacuation 用のスレッド数が増え，Evacuation に割り当てられるリソースが増加します。そのため，値を大きくするとスループットが低下する可能性があります。このオプションの値を変更した場合，性能の要件を満たしているか確認をしてください。 実行環境で作成できるスレッド数を超える値を指定した場合は，JavaVM 起動時に Evacuation 処理用のスレッドの作成に失敗するため，JavaVM を起動できません。	自然数の値を指定します。
-XX:ConcGCThreads	G1GC の Concurrent Marking の処理を並列実行するスレッドの数を指定します。 -XX:+UseG1GC を指定したときに有効になります。なお，このオプションは，デフォルト値で使用することを推奨します。変更する場合は，実際に JavaVM を動作させて測定した値を基に，最適な値を算出してから変更してください。 JavaVM 起動時に，このオプションに指定した数の Concurrent Marking 処理用のスレッドが作成され，処理が実行されます。	自然数の値を指定します。

オプション名	内容	指定可能値
	このオプションの指定を省略した場合のデフォルト値は次のとおりです。 • Max((ParallelGCThreads + 2)/4 , 1) Max(A,B)は A と B のうち，大きい値を取ることを意味します。また，ParallelGCThreads は-XX:ParallelGCThreads の値を意味します。 このオプションに 0 を指定した場合はデフォルト値が使用されます。 指定時には次の点に注意してください。 • 実行環境で作成できるスレッド数を超える値を指定した場合は，JavaVM 起動時に Concurrent Marking 処理用のスレッドの作成に失敗するため，JavaVM を起動できません。 Concurrent Marking 処理用スレッドのスレッド数は Evacuation 処理用のスレッド数を超えて作成することはできません。Evacuation 処理用のスレッド数を超えて指定した場合，メッセージが標準出力に出力され JavaVM の起動に失敗します。	
-XX:MaxGCPauseMillis	G1GC の GC によるアプリケーション停止時間の目標時間を[ms]の単位で指定できます。 -XX:+UseG1GC を指定したときに有効になります。 このオプションに 0 を指定した場合は，メッセージが標準出力に出力され JavaVM の起動に失敗します。 • このオプションに 100 以下の値を指定する場合，ログファイルの非同期出力機能（-XX:+JavaLogAsynchronous）を使用することを推奨します。	自然数の値を指定します。
-XX:ReservedCodeCacheSize	コードキャッシュ領域の最大サイズを指定します。	自然数の値を次に示す単位を使って指定します。 • キロ「k」 • メガ「m」 • ギガ「g」 なお，大文字・小文字は区別されません。

（凡例）

－：該当なし

注※

-XX:[+|-]UseSerialGC オプションと-XX:[+|-]UseG1GC オプションの指定の組み合わせごとに実行される処理を次に示します。

表 5–10　-XX:[+|-]UseSerialGC オプションと-XX:[+|-]UseG1GC オプションの指定の組み合わせごとの処理

-XX:[+|-]UseSerialGC オプションの指定	-XX:[+|-]UseG1GC オプションの指定	実行される処理
指定なし	指定なし	SerialGC を実行
-XX:+UseSerialGC	-XX:+UseG1GC	プロセスの起動に失敗
-XX:+UseSerialGC	-XX:-UseG1GC	SerialGC を実行（デフォルトと同様）
-XX:-UseSerialGC	-XX:-UseG1GC	
-XX:-UseSerialGC	-XX:+UseG1GC	G1GC を実行

重要

- デバッグまたはプロファイリング用オプションについて

  JavaVM オプションの-Xprof や-Xdebug，JVMTI エージェントの hprof や jdwp（-agentlib:<libname>などで指定）は，プログラムの開発用ユーティリティとして提供されているものです。システムの運用では指定しないようにしてください。

- GC の指定について

  アプリケーションサーバの GC は，SerialGC（UseSerialGC）と G1GC（UseG1GC）を選択できます。この 2 つの GC を同時に指定しないようにしてください。同時に指定した場合は，Java プロセスが起動できません。

5.5　日立 JavaVM で使用する環境変数の一覧

日立 JavaVM で使用する環境変数の一覧を，次の表に示します。

表 5-11　日立 JavaVM で使用する環境変数の一覧

分類	環境変数	説明
日立 JavaVM で使用する環境変数	JAVACOREDIR	スレッドダンプファイルの出力先ディレクトリを指定します。

5.6　日立 JavaVM で使用する環境変数の詳細

日立 JavaVM で使用する環境変数の詳細について説明します。

JAVACOREDIR

形式

setenv JAVACOREDIR <スレッドダンプファイルの出力先ディレクトリ>

説明

スレッドダンプファイルを出力する場合の，出力先ディレクトリを指定します。

デフォルト値

デフォルトでは JAVACOREDIR は未設定です。

スレッドダンプ出力先

デフォルトでは，スレッドダンプファイルはカレントディレクトリに出力されます。

前提オプション

-XX:+HitachiThreadDump

指定例

スレッドダンプを/home/user/threaddump に出力します。

```
setenv JAVACOREDIR /home/user/threaddump
```

注意事項

- JAVACOREDIR で指定したディレクトリへのスレッドダンプファイルの出力に失敗した場合は，カレントディレクトリにスレッドダンプファイルを出力します。
- カレントディレクトリへの出力も失敗した場合は，標準エラー出力だけにスレッドダンプを出力します。この場合は，標準出力にはスレッドダンプは出力されません。

6 日立 JavaVM で使用するコマンド

この章では，日立 JavaVM で使用するコマンドの入力形式，機能などについて説明します。

なお，日立 JavaVM は，Java SE 11 に準拠しています。JDK 11 で使用できるコマンドについては，Oracle 社が提供している JDK 11 のドキュメントを参照してください。

6.1　コマンドの文法の記述形式

6.1.1　記述形式の詳細

コマンドの文法の記述形式と使用する記号について説明します。

(1)　記述形式

コマンドの文法について次の形式で説明します。なお，各コマンドは，アルファベットの順に説明します。

形式
　コマンドの入力形式を示します。

機能
　コマンドの機能について説明します。

引数
　コマンドの引数およびオプションについて説明します。

出力形式
　コマンドの出力形式を示します。

入力例・出力例
　コマンドの入力例および出力例を示します。

戻り値
　コマンドの戻り値について説明します。

注意事項
　コマンドを実行する上での注意事項について説明します。

> **メモ**
>
> 各コマンドの説明では，上記の項目のうち必要な項目についてだけ説明しています。また，上記の項目以外に，各コマンドの固有情報を記載している場合があります。

(2)　使用する記号

コマンドの文法は次の表に示す記号および構文要素を使用して記述します。

表 6-1　文法で使用している記号

記号	意　味
|	横に並べられた複数の項目に対する項目間の区切りを示し，「または」を意味します。 （例）　A | B

記号	意　味
	A または B を指定することを示します。
{　}	この記号で囲まれている複数の項目のうちから 1 つを選択することを示します。項目が横に並べられ，記号｜で区切られている場合は，そのうちの 1 つを選択します。 （例）｛A ｜ B ｜ C｝ A，B または C のどれかを指定することを示します。
[　]	この記号で囲まれている項目は省略してもよいことを示します。複数の項目が横に並べて記述されている場合には，すべてを省略するか，記号｛｝と同じくどれか 1 つを選択します。 （例 1）［A］ 「何も指定しない」か「A を指定する」ことを示します。 （例 2）［B ｜ C］ 「何も指定しない」か「B または C を指定する」ことを示します。
…	記述が省略されていることを示します。 （例） ABC… ABC の後ろに記述があり，その記述が省略されていることを示します。
＜　＞	この記号で囲まれている項目は，該当する要素やファイルなどを指定したり，該当する要素が表示されたりすることを示します。 （例 1） ＜プロパティ＞ プロパティを記述します。またはプロパティが表示されます。 （例 2） ＜ファイル名＞ ファイル名を指定します。
…	この記号の直前に示す記号を繰り返し，複数個指定できることを示します。 （例） ＜プロパティ＞… プロパティは複数個，繰り返して指定できます。

表 6-2　文法で使用している構文要素

構文要素	定義
英字	A～Z　a～z
英小文字	a～z
英大文字	A～Z
数字	0～9
英数字	A～Z　a～z　0～9
記号	！　"　#　$　%　&　'　（　）　＋　，　_　・　．　／　：　；　＜　＝　＞　@　［　］　^　－　{　}　￣　タブ　空白

注　すべて半角文字を使用してください。

6.2.1　入力形式の詳細

コマンドの入力形式を次に示します。

コマンド名称 [オプション…]

各項目について説明します。なお，コマンドプロンプトを「$」，コマンド名称を「cmd」と表記します。

(1)　コマンド名称

実行するコマンドのファイル名を指定します。

空白を含むパスを指定してコマンドを実行する場合，パス全体を""で囲む必要があります。

- 誤った指定例：$ /opt/program path/bin/command
- 正しい指定例：$ ”/opt/program path/bin/command”

(2)　引数

引数には，オプションも含まれます。オプションの入力形式および指定規則を次に示します。

(a)　オプションの入力形式

オプションは，「-」（ハイフン）で始まる文字列です。オプションの入力形式には，オプション引数を指定しない形式と，1 個のオプション引数を指定する形式があります。

オプション引数を指定しない形式

$ cmd -オプションフラグ

1 個のオプション引数を指定する形式

$ cmd -オプションフラグ<スペースまたはタブ>オプション引数

（凡例）

- オプションフラグ
  1 文字の半角英数字です。大文字と小文字が区別されます。
- オプション引数
  オプションフラグに対する引数です。

(b)　オプションの指定規則

- オプションフラグは，1 つの「-」（ハイフン）にまとめて指定できません。
  誤った指定例：$ cmd -abc

正しい指定例：$ cmd -a -b -c

- オプション引数を必要とするオプションフラグのオプション引数は，省略できません。
- オプションフラグとオプション引数の間には，スペースまたはタブが必要です。
  誤った指定例：$ cmd -afile
  正しい指定例：$ cmd -a file
- 同じオプションフラグは，複数指定できません。
  誤った指定例：$ cmd -a 1 -a 2
- オプション引数に空白を含む場合，オプション引数全体を""で囲む必要があります。
  誤った指定例：$ cmd -a file 1
  正しい指定例：$ cmd -a ”file 1”

6.3　日立 JavaVM で使用するコマンドの一覧

日立 JavaVM で使用するコマンドの一覧を，次の表に示します。

表 6-3　日立 JavaVM で使用するコマンドの一覧

コマンド名称	分類	概要
car_tar_gz	core アーカイブ機能	core ファイルと関連するライブラリなどを gzip コマンドでまとめて，1 つのアーカイブファイルに出力します。
car_tar_Z	core アーカイブ機能	core ファイルと関連するライブラリなどを compress コマンドでまとめて，1 つのアーカイブファイルに出力します。
javacore	• core ファイルの取得 • スレッドダンプの取得	core ファイルとスレッドダンプを同時に取得します。
javagc	FullGC の強制発生	メモリリーク，システム障害およびアプリケーションのデバッグのために，任意のタイミングで FullGC を発生させます。
javatrace	トレース情報の収集	core ファイルからスタックトレース情報を収集します。
jheapprof	クラス別統計情報付き拡張スレッドダンプの出力	クラス別統計情報をスレッドダンプ中に出力します。
jheapprofanalyzer	クラス別統計情報解析ファイルの CSV 出力	クラス別統計情報解析ファイルを CSV 形式で出力します。

6.4　日立 JavaVM で使用するコマンドの詳細

日立 JavaVM で使用するコマンドの入力形式，機能などを次に示します。

コマンドの格納先

日立 JavaVM で使用するコマンドは，次のディレクトリに格納されています。

- <JDK のインストールディレクトリ>/bin/

car_tar_gz（core アーカイブ機能）

形式

```
car_tar_gz [-i <実行ファイル名>] [-f] [-s|-S] [<coreファイル名>] > <出力ファイル名>
```

機能

core ファイルと関連するライブラリなどをまとめて，1 つのアーカイブファイルに出力します。car_tar_gz コマンドは gzip コマンドを使用して圧縮します。

引数

-i <実行ファイル名>

実行ファイル名を指定します。

-f

パス名付きでライブラリなどのファイルを取得します。

-s

出力するアーカイブファイルのサイズ（単位：バイト）を報告します。

この場合，アーカイブファイルは出力されません。

-S

アーカイブファイルを作成するための，シェルスクリプトを出力します。

このオプションを指定する場合，<出力ファイル名>には，シェルスクリプト名を指定してください。

出力されたシェルスクリプトを実行すると，アーカイブファイルが出力されます。

<core ファイル名>

core ファイル名を指定します。省略した場合，"core"が指定されます。

<出力ファイル名>

アーカイブファイル，またはシェルスクリプトのファイル名を指定します。

入力例

アーカイブファイルを作成する場合

```
car_tar_gz -f core.8326 > corefile.tar.gz
```

アーカイブファイルのサイズを調べる場合

```
car_tar_gz -s core.8326
```

シェルスクリプトからアーカイブファイルを作成する場合

1. シェルスクリプトを作成します。

   ```
   car_tar_gz -S core.8326 > collect_cores.csh
   ```

2. シェルスクリプトに実行権を付与します。

   ```
   chmod +x ./collect_cores.csh
   ```

3. シェルスクリプトを実行して，アーカイブファイルを作成します。

   ```
   ./collect_cores.csh > corefile.tar.gz
   ```

戻り値

0：

正常終了しました。

1：

異常終了しました。

出力メッセージ

次のメッセージを出力した場合，正常なアーカイブファイルは出力されません。

表 6-4　car_tar_gz コマンドで出力されるエラーメッセージ

項番	エラーメッセージ	説明
1	usage: car_tar_gz [-i executable-file] [-f] [-s | -S] [core-file]	コマンドの引数が不正です。
2	car_tar_gz: inner error!	car_tar_gz コマンドの内部エラーです。
3	car_tar_gz: cannot create temporary name	一時的に使用するファイルが作成できません。 カレントディレクトリにある，ファイル car_exec??（?? は 00～99）を削除するか，移動してください。
4	car_tar_gz: ～ : file not found	ファイルが見つかりません。 core ファイルおよび-i オプションで指定した実行ファイルを確認してください。
5	car_tar_gz: ～ : not supported platform	サポートしていない OS です。

注意事項

- カレントディレクトリに書き込み権限が必要です。
- 同じディレクトリ下で，同じ core ファイルまたは作成時間が同じ core ファイルに対して，car_tar_Z コマンドおよび car_tar_gz コマンドは同時に実行できません。
- システムに tar コマンドおよび gzip コマンドがインストールされている必要があります。
- car_tar_gz コマンド実行中に強制終了した場合，カレントディレクトリに car_tar_gz コマンドが作成した一時的なファイルやディレクトリが残ります。

  一時的に作成されるファイル：car_exec??（??は，00～99）

  一時的に作成されるディレクトリ：carYYMMDDhhmm/（YYMMDDhhmm は，core ファイルの作成年月日時分）

car_tar_Z（core アーカイブ機能）

形式

```
car_tar_Z [-i <実行ファイル名>] [-f] [-s|-S] [<coreファイル名>] > <出力ファイル名>
```

機能

core ファイルと関連するライブラリなどをまとめて，1 つのアーカイブファイルに出力します。car_tar_Z コマンドは compress コマンドを使用して圧縮します。

引数

-i <実行ファイル名>

実行ファイル名を指定します。

-f

パス名付きでライブラリなどのファイルを取得します。

-s

出力するアーカイブファイルのサイズ（単位：バイト）を報告します。

この場合，アーカイブファイルは出力されません。

-S

アーカイブファイルを作成するための，シェルスクリプトを出力します。

このオプションを指定する場合，<出力ファイル名>には，シェルスクリプト名を指定してください。

出力されたシェルスクリプトを実行すると，アーカイブファイルが出力されます。

<core ファイル名>

core ファイル名を指定します。省略した場合，"core"が指定されます。

<出力ファイル名>

アーカイブファイル，またはシェルスクリプトのファイル名を指定します。

入力例

アーカイブファイルを作成する場合

```
car_tar_Z -f core.8326 > corefile.tar.Z
```

アーカイブファイルのサイズを調べる場合

```
car_tar_Z -s core.8326
```

シェルスクリプトからアーカイブファイルを作成する場合

1. シェルスクリプトを作成します。

   ```
   car_tar_Z -S core.8326 > collect_cores.csh
   ```

2. シェルスクリプトに実行権を付与します。

   ```
   chmod +x ./collect_cores.csh
   ```

3. シェルスクリプトを実行して，アーカイブファイルを作成します。

   ```
   ./collect_cores.csh > corefile.tar.Z
   ```

戻り値

0：

正常終了しました。

1：

異常終了しました。

出力メッセージ

次のメッセージを出力した場合，正常なアーカイブファイルは出力されません。

表 6-5　car_tar_Z コマンドで出力されるエラーメッセージ

項番	エラーメッセージ	説明
1	usage: car_tar_Z [-i executable-file] [-f] [-s | -S] [core-file]	コマンドの引数が不正です。
2	car_tar_Z: inner error!	car_tar_Z コマンドの内部エラーです。
3	car_tar_Z: cannot create temporary name	一時的に使用するファイルが作成できません。 カレントディレクトリにある，ファイル car_exec??（?? は 00～99）を削除するか，移動してください。
4	car_tar_Z: ～ : file not found	ファイルが見つかりません。

項番	エラーメッセージ	説明
		core ファイルおよび-i オプションで指定した実行ファイルを確認してください。
5	car_tar_Z: ～ : not supported platform	サポートしていない OS です。

注意事項

- カレントディレクトリに書き込み権限が必要です。
- 同じディレクトリ下で，同じ core ファイルまたは作成時間が同じ core ファイルに対して，car_tar_Z コマンドおよび car_tar_gz コマンドは同時に実行できません。
- システムに tar コマンドおよび compress コマンドがインストールされている必要があります。
- car_tar_Z コマンド実行中に強制終了した場合，カレントディレクトリに car_tar_Z コマンドが作成した一時的なファイルやディレクトリが残ります。

  一時的に作成されるファイル：car_exec??（??は，00～99）

  一時的に作成されるディレクトリ：carYYMMDDhhmm/（YYMMDDhhmm は，core ファイルの作成年月日時分）

javacore（core ファイルとスレッドダンプの取得）

形式

```
javacore [-i|-f] [-force] -p <プロセスID>
```

機能

コマンド実行時の core ファイルとスレッドダンプを同時に取得します。

引数

-i

core ファイルおよびスレッドダンプの出力処理の実行を確認するメッセージが表示されます。表示されたメッセージに対して y または n を入力します。このとき，y を入力するとスレッドダンプが出力されます。n を入力すると，何も出力しないで処理を終了します。省略した場合，-f オプションが指定されないかぎり，このオプションは有効です。

-f

-i オプションを無効にします。省略した場合，-i オプションが有効になります。

-force

Java プロセスが作成する/tmp/hsperfdata_<ユーザ名>/<プロセス ID>ファイルの有無の確認をしないで，-p オプションで指定したプロセス ID の Java プロセスに対して core を出力させます。

-p <プロセス ID>

<プロセス ID>には，core ファイルとスレッドダンプを取得する Java プログラムのプロセス ID を指定します。

戻り値

0：

正常終了しました。

1：

異常終了しました。

2：

一定時間内に core 生成処理終了の応答がありませんでした。

入力例

1. -f オプションを省略して javacore コマンドを実行します。

```
javacore -p 8326
```

2. core ファイルおよびスレッドダンプの出力処理の実行を確認するメッセージが表示されます。

```
send SIGQUIT to 8326:?(y/n)
```

3. core ファイルおよびスレッドダンプを取得する場合は y を，取得しない場合は n を入力します。

```
send SIGQUIT to 8326:?(y/n)y
```

4. core ファイルおよびスレッドダンプを取得すると，実行中の Java プログラムでは次のメッセージが出力されます。

```
Now generating core file (javacore8662.030806215140.core)...
done

(スレッドダンプを出力)

Writing Java core to javacore8662.030806215140.txt... OK
```

5. 実行中の Java プログラムは，カレントディレクトリに次のファイルを作成し，プログラムを継続します。

core ファイル

javacore<プロセス ID>.<日時>.core

スレッドダンプ

javacore<プロセス ID>.<日時>.txt

出力メッセージ

次のエラーメッセージまたは警告メッセージを出力した場合，core ファイルやスレッドダンプは取得されません。

表 6-6　javacore コマンドで出力されるエラーメッセージ

項番	エラーメッセージ	説明
1	usage: javacore [-f|-i] [-force] -p process-id	コマンドの引数が不正です。
2	javacore: can't create work file at /tmp, this request canceled	/tmp に参照および書き込み権限がありません。
3	javacore: illegal option --<オプション>	コマンドの引数に指定した<オプション>が不正です。
4	javacore: unexpected error occurred:<エラー原因>	コマンド実行中に予期しないエラーが発生しました。
5	javacore: please delete <削除できなかったファイル名> in <削除できなかったファイルのフルパス>	コマンド終了時に javacore コマンドの内部処理で作成したファイルを削除できませんでした。<削除できなかったファイルのフルパス>にある，削除できなかったファイルを削除してください。
6	<プロセス ID>: No such process	javacore コマンドの引数に指定した<プロセス ID>に該当するプロセスがありません。または，javacore コマンドで指定した<プロセス ID>に該当するプロセスが Java プロセス以外でした。
7	<プロセス ID>: Not owner	実行ユーザは，コマンドの引数に指定した<プロセス ID>に該当するプロセスのオーナーではありません。
8	<プロセス ID>: Now processing previous request, this request canceled	コマンドの引数に指定した<プロセス ID>のプロセスが現在 core を生成中です。
9	<プロセス ID>: Timeout occurred. Java process not responding.	コマンドの引数に指定した<プロセス ID>に該当するプロセスから，一定時間内に core 出力処理終了の応答がありませんでした。

注意事項

- javacore コマンドは，SIGQUIT シグナルを指定されたプロセスに送信します。誤って Java プログラム以外を指定すると，ほかのプログラムが停止することがあります。
- 同じ Java プロセスに対して，同時に javacore コマンドは実行できません。前回の javacore コマンドによる core 出力処理が終了したあとに実行してください。
- JavaVM プロセスが，スレッドダンプ取得要求に反応しないで無応答状態になっている場合には，javacore コマンドによる core ファイルも取得できません。この場合は，kill -6 コマンドの実行によって，JavaVM プロセスを強制停止させて core ファイルを取得してください。
- Linux の場合，gdb の gcore コマンドで core を生成します。gdb がインストールされていない場合には，次のエラーメッセージが標準出力に出力されます。

  ```
  Error occurred in generating core file, gdb not found.
  ```

また，インストールされている gdb のバージョンが古い場合には，次のエラーメッセージが標準出力に出力されます。

```
Error occurred in generating core file, gdb version 5.2 or later needed.
```

- javacore コマンドは，実行時に，/tmp/hsperfdata_<ユーザ名>/<プロセス ID>ファイルを使用します。該当するファイルが存在しない場合は，javacore コマンドによる core ファイルの出力はできません。ただし，-force オプションを指定した場合，/tmp/hsperfdata_<ユーザ名>/<プロセス ID>ファイルの有無の確認をしないで，-p オプションで指定したプロセス ID の Java プロセスに対して core を出力させます。

javagc（GC の強制発生）

形式

```
javagc [-i|-f] [-v] [-s] [-force] [-ehgc] -p <プロセスID>
```

機能

メモリリーク，システム障害およびアプリケーションのデバッグのために，プロセス ID が<プロセス ID>の Java プロセスに対して，任意のタイミングで FullGC を発生させます。

Java プロセスとの通信には SIGQUIT シグナルを使用します。コマンドを実行すると，コマンドの処理内容をユーザに確認します。n（発生させない，実行しない，または送信しない）と回答した場合には，コマンドの処理は実行されません（戻り値が 1 となります）。この確認動作は，-f オプションを指定することで省略できます。

<プロセス ID>の Java プロセスが通常の要因で発生する CopyGC や FullGC を実行中の場合は，その終了を待ってからコマンドを実行します。

引数

-i

次に示すコマンドの処理内容をユーザに確認します。

- GC を発生させるために SIGQUIT シグナルを送信させるかどうか

このオプションより前に指定した，-f オプションを無効にします。

-f

次に示すコマンドの処理内容をユーザに確認しません。

- GC を発生させるために SIGQUIT シグナルを送信させるかどうか

このオプションより前に指定した，-i オプションを無効にします。

-v

-XX:+HitachiVerboseGC オプションの指定がなくても，JavaVM ログファイル作成の規則に従って JavaVM ログファイルを作成して，拡張 verbosegc 情報を出力します。

その際，以下のオプション値も反映した内容の拡張 verbosegc 情報を出力します。

- -XX:+HitachiVerboseGCPrintDate
- -XX:+HitachiVerboseGCPrintCause
- -XX:+HitachiVerboseGCCpuTime
- -XX:+HitachiCommaVerboseGC

-s

標準出力に拡張 verbosegc 情報を出力します。

その際，以下のオプション値も反映した内容の拡張 verbosegc 情報を出力します。

- -XX:+HitachiVerboseGCPrintDate
- -XX:+HitachiVerboseGCPrintCause
- -XX:+HitachiVerboseGCCpuTime
- -XX:+HitachiCommaVerboseGC

-force

Java プロセスが作成する/tmp/hsperfdata_<ユーザ名>/<プロセス ID>ファイルの有無の確認をしないで，-p オプションで指定したプロセス ID の Java プロセスに対して GC を実行します。

-ehgc

このオプションは使用できません。

-p <プロセス ID>

FullGC を実行したいプロセス ID を指定します。

日立 JavaVM 拡張オプションについては，次の個所を参照してください。

- 「5.1　日立 JavaVM 拡張オプションの一覧」
- 「5.2　日立 JavaVM 拡張オプションの詳細」

戻り値

0：

正常終了しました。

1：

異常終了しました。

2：

一定時間内に GC 処理終了の応答がありませんでした。

入力例

1. -i オプションを指定して javagc コマンドを実行します。

```
javagc -i -v -p 8326
```

2. プロセス ID の確認メッセージが表示されます。

```
send SIGQUIT to 8326:?(y/n)
```

3. SIGQUIT シグナルを送信させる場合は y を，送信させない場合は n を入力します。

```
send SIGQUIT to 8326:?(y/n)y
```

出力例

```
[VGC]<Wed Mar 17 00:42:30 2004>(Skip Full:0,Copy:0)[Full GC 149K->149K(1984K), 0.0786038 sec
s][DefNew::Eden: 264K->0K(512K)][DefNew::Survivor: 0K->63K(64K)][Tenured: 85K->149K(1408K)][
Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388
K, 388K)][cause:JavaGC Command]
```

出力メッセージ

表 6-7　javagc コマンドで出力されるメッセージ

項番	メッセージ	説明
1	javagc [-f|-i][-v][-s] [-force] [-ehgc] -p process-id	javagc コマンドへの引数の指定が間違っています。
2	javagc: illegal option--<オプション>	javagc コマンドに指定した<オプション>が不正です。
3	<プロセス ID>: No such process	javagc コマンドの引数に指定した<プロセス ID>に該当するプロセスがありません。または，javagc コマンドで指定した<プロセス ID>に該当するプロセスが Java プロセス以外でした。
4	<プロセス ID>: Not owner	実行ユーザは，javagc コマンドの引数に指定した<プロセス ID>に該当するプロセスのオーナーではありません。
5	<プロセス ID>: Now processing previous request, this request canceled	javagc コマンドの引数に指定した<プロセス ID>に該当するプロセスは，前回の javagc コマンドによる GC を実行中です。javagc コマンドによる GC 実行要求はキャンセルされます。
6	javagc: can't create work file at /tmp, this request canceled	/tmp に参照および書き込み権限がないため，GC 要求ファイルが作成できません。javagc コマンドによる GC 実行要求はキャンセルされます。
7	javagc: unexpected error occurred:<エラー原因>	javagc コマンド実行中に予期しないエラーが発生しました。 <エラー原因>には，例えば下記のような表示がされます。 • 作業用メモリ確保に失敗した場合 malloc systemcall fail (errno=Y)

項番	メッセージ	説明
		• オブジェクトのクローズに失敗した場合 close systemcall fail (errno=Y)
8	<プロセス ID>: Timeout occurred. Java process not responding.	javagc コマンドの引数に指定した<プロセス ID>に該当するプロセスから，一定時間内に GC 処理終了の応答がありませんでした。
9	javagc : please delete <削除できなかったファイル名> in <削除できなかったファイルのフルパス>	javagc コマンドを終了したときに，内部ファイルを削除できませんでした。削除できなかったファイルのフルパスにある，削除できなかったファイルを削除してください。
10	<プロセス ID>: Failed to retry GC. Java process is GC locked.	javagc コマンドに引数に指定した<プロセス ID>に該当するプロセスが，GC 実行が抑止された状態で，GC 処理が実行できませんでした。

注意事項

- 同じ Java プロセスに対して，同時に javagc コマンドは実行できません。前回の javagc コマンドによる GC 処理が終了してから実行してください。前回の GC 処理が終了している場合は，JavaVM ログファイルに出力される拡張 verbosegc 機能の GC の要因に"JavaGC Command"が出力されます。
- javagc コマンドは SIGQUIT シグナルを指定されたプロセス ID に送信します。誤って Java プロセス以外のプロセス ID を指定すると，ほかのプログラムが停止することがあります。
- javagc コマンド実行時に，/tmp/hsperfdata_<ユーザ名>/<プロセス ID>ファイルを使用します。該当するファイルが存在しない場合は，javagc コマンドによる GC の要求はできません。ただし，-force オプションを指定した場合，/tmp/hsperfdata_<ユーザ名>/<プロセス ID>ファイルの有無の確認をしないで，-p オプションで指定したプロセス ID の Java プロセスに対して GC を発生させます。

javatrace（トレース情報の収集）

形式

```
javatrace 〈coreファイル名〉〈実行ファイル名〉[〈出力ファイル名〉][-l 〈ライブラリファイル名〉...]
```

機能

core ファイルからスタックトレース情報を取得します。

このコマンドは，JavaVM が異常終了して core ファイルを生成した場合，その異常終了の原因究明に必要な情報を取得するために実行するコマンドです。プロセスダウン時の詳細要因などが調査できます。

引数

<core ファイル名>

core ファイル名を指定します。

<実行ファイル名>

core ファイルを生成した実行ファイル名を指定します。

<出力ファイル名>

出力ファイル名を指定します。

省略した場合は，カレントディレクトリの"javatrace.log"に出力されます。

-l <ライブラリファイル名>

使用したライブラリファイルを指定します。

共用ライブラリが絶対パスで実行ファイルに取り込まれている場合は，自動的に読み込まれるため，指定する必要はありません。

戻り値

0：

正常終了しました。

1：

異常終了しました。

入力例

JavaVM が異常終了して core ファイルを生成した場合，次のメッセージが表示されます。このメッセージ内の javatrace コマンドの文字列を実行します。

なお，このメッセージは，異常終了時に生成されるエラーリポートファイル（hs_err_pid<プロセス ID>.log）にも出力されます。

```
  :
# You can get further information from javatrace.log file generated
# by using javatrace command.
# usage: javatrace core-file-name loadmodule-name [out-file-name] [-l(library-name)...]
# Please use javatrace command as follows and submit a bug report
# to Hitachi with javatrace.log file:
# [/opt/Cosminexus/jdk/bin/javatrace core /opt/Cosminexus/CC/server/bin/cjstartsv]
#
```

core ファイル名が core の場合

```
<JDKインストールディレクトリ>/bin/javatrace core <JDKインストールディレクトリ>/bin/java
```

OS によっては実際に出力される core ファイル名が core.<プロセス ID>になる場合があります。その場合は，実際に出力された core ファイル名を javatrace の引数に指定してください。

core ファイル名が core.<プロセス ID>の場合

```
<JDKインストールディレクトリ>/bin/javatrace core.8326 <JDKインストールディレクトリ>/bin/java
```

出力メッセージ

次のエラーメッセージまたは警告メッセージを出力した場合，スタックトレース情報は出力されません。

表 6-8　javatrace コマンドで出力されるエラーメッセージ

項番	エラーメッセージ	説明
1	usage : javatrace core-file-name loadmodule-name [out-filename] [-llibrary-name...]	javatrace コマンドへの引数の指定が間違っています。
2	javatrace: Cannot open file-name :No such file or directory	<出力ファイル>で指定したファイル以外のファイル（file-name）が見つかりません。
3	javatrace: Cannot open file-name : Permission denied	ファイル（file-name）の読み込みが許可されていません。
4	javatrace : Cannot create file-name : Already exist	<出力ファイル>に指定したファイル（file-name）がすでに存在しています。
5	* unknown core type(XXXXXXX) ignored.	<core ファイル名>で指定したファイルは，core ファイルではありません。
6	* ERROR : file-name, unknown magic(0xXXXX)	<実行ファイル名>で指定したファイルは，実行ファイルではありません。
7	WARNING : core file may not match loadmodule (core file from 'loadmodule-name')	<実行ファイル名>で指定した実行ファイルと，<core ファイル名>で指定した core ファイルから取り出した実行ファイル（loadmodule-name）が一致していません。
8	javatrace : Bad argument : argument-name is directory	指定したファイル（file-name）はディレクトリです。
9	* ERROR : library-name open : No such file or directory	実行ファイルが取り込んだ共用ライブラリ（library-name）が見つかりません。共用ライブラリが相対パスで取り込まれている場合には，-l オプションで共用ライブラリを明示的に指定する必要があります。
10	javatrace : illegal option -- x	誤ったオプション文字（x）が指定されています。
11	javatrace : Cannot create file-name : Permission denied	<出力ファイル>で指定したファイル（file-name）の出力先に書き込み権限がありません。

注意事項

- javatrace コマンドは core ファイルが生成されたマシンで実行してください。
- JavaVM が異常終了した場合に標準出力とエラーリポートファイルに出力されるメッセージで，javatrace コマンドの第 2 引数が，次のように絶対パスのロードモジュール名になっていないことがあります。

```
#
# You can get further information from javatrace.log file generated
# by using javatrace command.
# usage: javatrace core-file-name loadmodule-name [out-file-name] [-l(library-name)...]
# Please use javatrace command as follows and submit a bug report
# to Hitachi with javatrace.log file:
# [/opt/Cosminexus/jdk/bin/javatrace /users/Java/core ../java]
#
```

その場合は，javatrace コマンドの第 2 引数に指定するロードモジュール名を，絶対パスに変換してから実行してください。

jheapprof（クラス別統計情報付き拡張スレッドダンプの出力）

形式

```
jheapprof [-i|-f] [-class <クラス名>] [-staticroot|-nostaticroot]
          [-explicit|-noexplicit] [-fullgc|-copygc|-nogc]
          [-garbage|-nogarbage] [-rootobjectinfo|-norootobjectinfo]
          [-rootobjectinfost <値> ] [-force] -p <プロセスID>
```

機能

引数に指定したプロセス ID の Java プロセスについて，クラス別統計情報を含んだ拡張スレッドダンプを出力します。

引数

-i

クラス別統計情報付き拡張スレッドダンプの出力処理の実行を確認するメッセージが表示されます。表示されたメッセージに対して y または n を入力します。このとき，y を入力すると，クラス別統計情報を含んだ拡張スレッドダンプが出力されます。n を入力すると，何も出力しないで処理を終了します。省略した場合，-f オプションが指定されないかぎり，このオプションは有効です。

-f

-i オプションを無効にします。省略した場合，-i オプションが有効になります。

-class <クラス名>

<クラス名>に指定したクラス（インスタンス）をメンバに持つクラスの構造を一覧にしてスレッドダンプ中に出力します。

-staticroot

static フィールドを基点とした参照関係情報出力機能を有効にし，static フィールドを基点とした参照関係情報を出力します。省略した場合，-nostaticroot オプションが指定されない限り，このオプションは有効です。

このオプションの前提は，-class オプションです。-class オプションの指定がない場合，このオプションは無効になります。

なお，このオプションと nostaticroot オプションを同時に指定している場合，最後に指定しているオプションが有効になります。

-nostaticroot

static フィールドを基点とした参照関係情報出力機能を無効にします。省略した場合，-staticroot オプションが有効になります。

なお，このオプションと-staticroot オプションを同時に指定している場合，最後に指定しているオプションが有効になります。

-explicit

このオプションは使用できません。

-noexplicit

このオプションは使用できません。

-fullgc

統計する前に実行する GC に FullGC を設定します。省略した場合，-copygc オプションや-nogc オプションが指定されないかぎり，このオプションは有効です。なお，このオプションと-copygc オプションまたは-nogc オプションを同時に指定している場合，最後に指定しているオプションが有効になります。

-copygc

統計する前に実行する GC に CopyGC を設定します。このオプションと-nogc オプションを省略した場合，-fullgc オプションが有効になります。

なお，このオプションと-fullgc オプションまたは-nogc オプションを同時に指定している場合，最後に指定しているオプションが有効になります。

-nogc

統計する前に GC を実行しません。このオプションと-copygc オプションを省略した場合，-fullgc オプションが有効になります。なお，このオプションと-fullgc オプションまたは-copygc オプションを同時に指定している場合，最後に指定しているオプションが有効になります。

-garbage

Tenured 領域内不要オブジェクト統計機能が有効になり，Tenured 領域内の不要なオブジェクトを統計対象としたクラス別統計情報を出力します。また，インスタンス統計機能と STATIC メンバ統計機能は無効になります。省略した場合，-nogarbage オプションが有効になります。統計前 GC 選択機能については，-fullgc オプション，-copygc オプションが無効になり，-nogc オプションは有効になります。このため，統計処理前に GC を実行しません。なお，このオプションと-nogarbage オプションを同時に指定している場合，最後に指定しているオプションが有効になります。

-nogarbage

Tenured 領域内不要オブジェクト統計機能が無効になります。そのため，Tenured 領域内の不要なオブジェクトを統計対象としたクラス別統計情報は出力しません。省略した場合，-garbage オプション

が指定されないかぎり，このオプションは有効です。なお，このオプションと-garbage オプションを同時に指定している場合，最後に指定しているオプションが有効になります。

-rootobjectinfo

Tenured 増加要因の基点オブジェクトリスト出力機能が有効になり，Tenured 増加要因の基点オブジェクトリストを出力します。

このオプションは，-garbage オプションが有効であることが前提です。そのため，-nogarbage オプションを有効にすると，このオプションは無効になります。省略した場合，-norootobjectinfo オプションが指定されないかぎり，このオプションは有効です。なお，このオプションと-norootobjectinfo オプションを同時に指定している場合，最後に指定しているオプションが有効になります。

-norootobjectinfo

Tenured 増加要因の基点オブジェクトリスト出力機能が無効になります。そのため，Tenured 増加要因の基点オブジェクトリストは出力しません。省略した場合，-rootobjectinfo オプションが有効になります。なお，このオプションと-rootobjectinfo オプションを同時に指定している場合，最後に指定しているオプションが有効になります。

-rootobjectinfost <値>

Tenured 増加要因の基点オブジェクトリストの情報量を調節します。インスタンスサイズの合計が，指定した<値>以上のクラス情報だけが，Tenured 増加要因の基点オブジェクトリストに出力されます。省略した場合，0 が設定されます。

このオプションは，-rootobjectinfo オプションが有効であることが前提です。そのため，-norootobjectinfo オプションを有効にすると，このオプションは無効になります。<値>には 0 以上の自然数を指定できます。自然数以外や，文字列を指定した場合は，引数の指定に誤りがあるという内容のエラーメッセージが出力されて終了します。

-force

Java プロセスが作成する/tmp/hsperfdata_<ユーザ名>/<プロセス ID>ファイルの有無の確認をしないで，-p オプションで指定したプロセス ID の Java プロセスに対して拡張スレッドダンプの出力を要求します。

-p <プロセス ID>

<プロセス ID>には，クラス別統計情報を出力する Java プログラムのプロセス ID を指定します。

戻り値

0：

正常終了しました。

1：

異常終了しました。

2：

一定時間内にクラス別統計情報出力処理終了の応答がありませんでした。

入力例

1. -f オプションを省略して jheapprof コマンドを実行します。

```
% jheapprof -p 2463
```

2. プロセス ID の確認メッセージが表示されます。

```
send SIGQUIT to 2463: ? (y/n)
```

3. SIGQUIT シグナルを送信させる場合は y を，送信させない場合は n を入力します。

```
send SIGQUIT to 2463: ? (y/n)y
```

4. クラス別統計情報付き拡張スレッドダンプを出力すると，実行中の Java プログラムでは次のメッセージが出力されます。

```
Writing Java core to javacore2463.030806215140.txt... OK
```

5. 実行中の Java プログラムは，カレントディレクトリにクラス別統計情報付き拡張スレッドダンプ（javacore<プロセス ID>.<日時>.txt）を作成し，プログラムを継続します。

出力形式

クラス別統計情報の出力形式については，「4.9　クラス別統計情報解析機能」を参照してください。

出力メッセージ

次のエラーメッセージまたは警告メッセージが出力された場合，クラス別統計情報付き拡張スレッドダンプは出力されません。

表 6-9　jheapprof コマンドで出力されるメッセージ

項番	エラーメッセージ	説明
1	usage: jheapprof [-f|-i] [-class classname] [-staticroot|-nostaticroot] [-explicit|-noexplicit] [-fullgc|-copygc|-nogc] [-garbage|-nogarbage] [-rootobjectinfo|-norootobjectinfo] [-rootobjectinfostsize] [-force] -p process-id	jheapprof コマンドへの引数の指定が間違っています。
2	jheapprof: illegal option -- <オプション>	jheapprof コマンドに指定した<オプション>が不正です。
3	<プロセス ID>: Now processing previous request, this request canceled	jheapprof コマンドの引数に指定した<プロセス ID>に該当するプロセスが現在クラス別統計情報の出力中です。
4	<プロセス ID>: No such process	jheapprof コマンドの引数に指定した<プロセス ID>に該当するプロセスがありません。または，jheapprof コマンドで指定した<プロセス ID>に該当するプロセスが Java プロセス以外でした。

項番	エラーメッセージ	説明
5	<プロセス ID>: Not owner	jheapprof コマンドの引数に指定した<プロセス ID>のプロセスのオーナーではありません。
6	jheapprof: can't create work file at /tmp , this request canceled	一時ファイル用ディレクトリに参照・書き込み権限がない場合，クラス別統計情報付き拡張スレッドダンプを出力できません。クラス別統計情報付き拡張スレッドダンプの出力要求はキャンセルされます。
7	jheapprof: please delete <削除できなかったファイル名> in <削除できなかったファイルのフルパス>	jheapprof コマンドを終了したときに，内部ファイルを削除できませんでした。削除できなかったファイルのフルパスにある，削除できなかったファイルを削除してください。
8	jheapprof: unexpected error occurred: <エラー原因>	jheapprof コマンド実行中に予期しないエラーが発生しました。 <エラー原因>には，例えば下記のような表示がされます。 • 作業用メモリ確保に失敗した場合 malloc systemcall fail (errno=Y) • オブジェクトのクローズに失敗した場合 close systemcall fail (errno=Y)
9	<プロセス ID>: Timeout occurred. Java process not responding	jheapprof コマンドの引数に指定した<プロセス ID>に該当するプロセスから，一定時間内にクラス別統計情報出力処理終了の応答がありませんでした。

注意事項

- jheapprof コマンドはプログラムの開発用ユーティリティとして提供されているものです。システムの運用では使用しないでください。
- 同じ Java プロセスに対して，同時に jheapprof コマンドは実行できません。前回の jheapprof コマンドによるクラス別統計情報が拡張スレッドダンプに出力されたあとに実行してください。
- 引数に指定したプロセス ID の Java プロセスオーナーではないユーザがこのコマンドを実行すると，メッセージが出力されて，処理は終了します。ただし，ユーザが root である場合は，処理を継続します。
- jheapprof コマンドは，SIGQUIT シグナルを指定されたプロセスに送信します。誤って Java プログラム以外を指定すると，ほかのプログラムが停止することがあります。
- jheapprof コマンド実行時に，/tmp/hsperfdata_<ユーザ名>/<プロセス ID>ファイルを使用します。該当するファイルが存在しない場合，jheapprof コマンドによるクラス別統計情報出力はできません。
- G1GC を使用中に jheapprof コマンドを使用した場合，次のエラーメッセージを出力してスレッドダンプを出力しません。

  ```
  jheapprof: can't use jheapprof and g1gc at the same time.
  ```

- クラス別統計情報を取得するため，jheapprof コマンド実行中の Java プロセスは，ほかの処理をすべて停止して処理を実行します。

jheapprofanalyzer（クラス別統計情報解析ファイルの CSV 出力）

形式

```
jheapprofanalyzer [-J <オプション名>] [<ファイル名> ...]
```

機能

クラス別統計情報解析ファイルを CSV 形式で出力します。

クラス別統計情報解析ファイルで使用する入力ファイルについては，「4.9　クラス別統計情報解析機能」を参照してください。

引数

-J <オプション名>

<オプション名>には，次のオプションを指定できます。また，次のオプション以外を指定した場合は，動作保証の対象外となります。

- -Xms
  メモリ割り当てプールの初期サイズをバイト数で指定します。
- -Xmx
  メモリ割り当てプールの最大サイズをバイト数で指定します。
- -DJP.co.Hitachi.soft.jvm.tools.jheapprofanalyzer.threshold=num
  num: インスタンス合計サイズのしきい値を設定します。範囲は 0～2^{63}-1 （Long.MAX_VALUE）です。インスタンス合計サイズが num 以上のクラスだけ出力します。デフォルト値は，1024 です。

<ファイル名>

クラス別統計情報付き拡張スレッドダンプファイルを指定できます。ファイル名称の規定は，特にありません。また，ファイルの指定は，順不同であり，数に制限はありません。

戻り値

0：
　正常終了しました。

1 以上：
　異常終了しました。

入力例

```
jheapprofanalyzer -J-Xms1024m -J-Xmx1024m -J-DJP.co.Hitachi.soft.jvm.tools.jheapprofanayzer.
threshold=5000
javacore22356.080523161703.txt  javacore22356.080523161711.txt
```

出力形式

クラス別統計情報解析ファイルの出力形式については，「4.9　クラス別統計情報解析機能」を参照してください。

出力メッセージ

次のエラーメッセージが出力された場合，クラス別統計情報解析ファイルは出力されません。また，次のエラーメッセージ以外が出力された場合は，デフォルトの例外処理となります。

表 6-10　jheapprofanalyzer コマンドで出力されるエラーメッセージ

項番	エラーメッセージ	説明	出力後の動作
1	usage: jheapprofanalyzer [options] file... where options include: -J-Xms<size> set initial Java heap size -J-Xmx<size> set maximum Java heap size -J-DJP.co.Hitachi.soft.jvm.tools.jheapprofanalyzer.threshold=<num> set instance total size threshold	JheapprofAnalyzer クラスへの引数の指定が間違っています。	(a)
2	JheapprofAnalyzer: Illegal property value<num>. Default is assumed.	JP.co.Hitachi.soft.jvm.tools.jheapprofanalyzer.threshold の<num>に数字以外を指定しました。または<num>が範囲外となっています。	(b)
3	JheapprofAnalyzer: can't open input file<ファイル名>	ディレクトリに<ファイル名>がありません。または，別の原因でファイルを開けません。	(c)
4	JheapprofAnalyzer: can't read input file<ファイル名>	<ファイル名>の読み込みに失敗しました。	(c)
5	JheapprofAnalyzer: Illegal input file format<ファイル名>	<ファイル名>は，クラス別統計情報付き拡張スレッドダンプファイルではありません。	(c)
6	JheapprofAnalyzer: can't open output file<ファイル名>	出力ファイルが開けません。 エラーの原因として，次の状態が考えられます。 • 出力ファイルがディレクトリとなっています。 • 出力ファイルがありません。 • 別の原因で出力ファイルを開けません。	(a)
7	JheapprofAnalyzer: can't write output file<ファイル名>	<ファイル名>の書き込みに失敗しました。	(a)

（凡例）

(a)：エラーとなり処理を終了します。

(b)：デフォルトを仮定して処理を続行します。

(c)：処理を続行し，指定したすべての入力ファイルのエラーチェックをします。

注意事項

クラス別統計情報解析機能では，日付を取得するときと，データを読み込むときにファイルを開きます。そのため，コマンド実行中に入力ファイルの更新および削除の操作をした場合の結果は保証されません。

7 トラブルシューティングに使用する資料の詳細

この章では，トラブルシューティングで使用するログやトレース情報について説明します。

7.1　日立 JavaVM のスレッドダンプ

日立 JavaVM のスレッドダンプを調査すると，システムのデッドロックなどの Java プログラムレベルでのトラブル要因の調査が容易になります。

出力される情報の種類は，アプリケーションサーバの起動時に指定しているオプションによって異なります。起動オプションについては「5.　日立 JavaVM 起動オプション」を，起動オプションのデフォルト値についてはマニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。また，日立 JavaVM の資料取得の方法については，「3.　トラブルシューティング」を参照してください。

7.1.1　スレッドダンプ情報の構成

日立 JavaVM のスレッドダンプ情報の構成を次の表に示します。

表 7-1　スレッドダンプ情報の構成

出力情報	内容
ヘッダ	日付，JavaVM バージョン情報，起動コマンドラインを出力します。
システム設定	次の情報を出力します。 • JDK の実行環境のインストール場所を表す Java ホームパス • JDK を構成するライブラリのインストールディレクトリを表す Java DLL パス • システムクラスパス • Java コマンドオプション
動作環境	次の情報を出力します。 • ホスト名 • OS バージョン • CPU 情報 • リソース情報
Java ヒープ情報	Java ヒープの各世代のメモリ使用状況を出力します。
JavaVM 内部メモリマップ情報	JavaVM 自身の確保しているメモリの領域情報を出力します。
JavaVM 内部メモリサイズ情報	JavaVM 自身の確保しているメモリのサイズ情報を出力します。
アプリケーション環境	次の情報を出力します。 • シグナルハンドラ • 環境変数
ライブラリ情報	ローディングされているライブラリの情報を出力します。
スレッド情報 <スレッド 1> : <スレッド n>	スレッドごとにスレッド情報を出力します。

出力情報	内容
Java モニタダンプ※	Java モニタオブジェクトの一覧を表示します。
JNI グローバル参照情報	JavaVM が保持している JNI のグローバル参照の数を出力します。
クラス別統計情報	jheapprof コマンドで指定した Java プロセスのクラスごとに次の情報を出力します。 • インスタンスがメンバとして持つインスタンスの合計サイズ，およびインスタンスの参照関係 • static メンバが持つインスタンスの合計サイズ • Tenured 領域の増加原因となるオブジェクトのクラス，およびインスタンスの合計サイズ
フッタ	スレッドダンプが終了した時刻を表示します。

注※　notify 待ちの一覧が表示されないことがあります。

日立 JavaVM のスレッドダンプ情報の詳細については，「-XX:[+|-]HitachiThreadDump（拡張スレッドダンプ情報出力オプション）」を参照してください。なお，クラス別統計情報については，「4.2　クラス別統計機能」を参照してください。

7.2　日立 JavaVM の GC ログ

GC のログは，JavaVM ログファイルに出力されます。

詳細については，「7.3　JavaVM ログ（JavaVM ログファイル）」を参照してください。

7.3　JavaVM ログ（JavaVM ログファイル）

JavaVM ログファイルは，日立 JavaVM が標準の JavaVM に追加した拡張オプションを使用して出力されます。このログを取得するためには，対象のアプリケーションサーバを起動するときに必要なオプションを指定しておく必要があります。

7.3.1　JavaVM ログファイルを出力するオプション

JavaVM ログファイルを出力するオプションを次に示します。

- -XX:+HitachiOutOfMemoryStackTrace

  OutOfMemoryError 発生時にスタックトレースを出力するオプションです。なお，このオプションを指定した場合に同時に指定される，-XX:+HitachiOutOfMemorySize および-XX:+HitachiOutOfMemoryCause が指定された場合も，JavaVM ログファイルが出力されます。

- -XX:+HitachiVerboseGC

  GC が発生したときの拡張 verbosegc 情報を出力するオプションです。拡張 verbosegc 情報の取得については，「7.3.2　拡張 verbosegc 情報の取得」を参照してください。

- -XX:+HitachiJavaClassLibTrace

  System.gc()，System.exit()，Runtime.exit()，または Runtime.halt()のどれかの API を実行したときに，これらの API の呼び出しトレースを出力するオプションです。

  なお，-XX:HitachiJavaClassLibTraceLineSize オプションを指定している場合，出力されるトレースは，指定した文字数（バイト数）以内で出力されます。1 行の文字数が指定した値を超える場合は，「at」以降の文字列の前半部分が削除されて，指定した文字数分出力されます。

- -XX:+JITCompilerContinuation

  JIT コンパイラ稼働継続機能を有効にするオプションです。JIT コンパイルがアプリケーションを構成するメソッドの論理矛盾で失敗すると，JIT コンパイラ稼働継続機能のログが JavaVM ログファイルに出力されます。

それぞれのオプションを指定した場合の出力内容の詳細については，次の個所を参照してください。

- 「-XX:[+|-]HitachiOutOfMemoryStackTrace（スタックトレース出力オプション）」
- 「-XX:[+|-]HitachiVerboseGC（拡張 verbosegc 情報出力オプション）」
- 「-XX:[+|-]HitachiJavaClassLibTrace（クラスライブラリのスタックトレース出力オプション）」
- 「-XX:[+|-]JITCompilerContinuation（JIT コンパイラ稼働継続機能オプション）」

7.3.2　拡張 verbosegc 情報の取得

次の表に示すオプションを指定すると，拡張 verbosegc 情報を取得できます。拡張 verbosegc 情報からは，そのサーバで必要とする Java ヒープ領域サイズ，Metaspace 領域サイズなどを見積もるための情報が取得できます。なお，Java ログには，OutOfMemoryError 発生時のスタックトレースも出力されます。

表 7-2　拡張 verbosegc 情報の取得を指定するオプション

オプション	意味
-XX:+HitachiVerboseGC	拡張 verbosegc 情報を出力するかどうかを指定します。GC の内部領域である Eden，Survivor，Tenured，Metaspace 領域種別ごとに情報を出力します。デフォルトでは出力されません。-XX:+HitachiVerboseGC と指定すると拡張 verbosegc 情報が出力され，-XX:-HitachiVerboseGC と指定すると拡張 verbosegc 情報は出力されません。
-XX:+HitachiVerboseGCPrintDate	拡張 verbosegc 情報を出力するログの各行の先頭に，ログを出力した日付を表示するかどうかを指定します。
-XX:+HitachiVerboseGCCpuTime	GC の開始から終了までの間で，GC の実行スレッドのユーザモードおよびカーネルモードに費やされた時間だけを表示するか，GC の開始から終了までの実時間を表示するかを指定します。
-XX:HitachiVerboseGCIntervalTime=<時間間隔>	-XX:+HitachiVerboseGC に対する出力時間の間隔を数値（単位：秒）で指定します。時間間隔のデフォルト値は 0（GC 発生のたびに出力）です。なお，時間間隔を指定すると，その時間間隔の間に発生した GC 回数も表示されます。
-XX:+HitachiVerboseGCPrintCause	拡張 verbosegc 情報を出力するログに，GC が発生した原因を表示するかどうかを指定します。
-XX:+HitachiCommaVerboseGC	拡張 verbosegc 情報を出力するログを CSV 形式で出力するかどうかを指定します。CSV 形式で出力する場合，拡張 verbosegc 情報の括弧「( )」「[ ]」「< >」や，区切り「:」はすべて省略され，数値または文字列が「,」で区切られて出力されます。
-XX:+HitachiVerboseGCPrintTenuringDistribution	Survivor 領域の年齢分布情報を出力するかどうかを指定します。デフォルトでは出力されません。出力形式や出力情報については，「4.10　Survivor 領域の年齢分布情報出力機能」を参照してください。
-XX:+HitachiVerboseGCPrintJVMInternalMemory	JavaVM 内部で管理しているヒープ情報を JavaVM ログファイルに出力するかどうかを指定します。
-XX:+HitachiVerboseGCPrintThreadCount	Java スレッドの数を監視するために，Java スレッドの数を JavaVM ログファイルに出力するかどうかを指定します。
-XX:+HitachiVerboseGCPrintDeleteOnExit	java.io.File.deleteOnExit()を呼び出したことによって JavaVM が確保した累積のヒープサイズとメソッドの呼び出し回数を，JavaVM ログファイルに出力するかどうかを指定します。

オプション	意味
-XX:+PrintCodeCacheInfo	コードキャッシュ領域の使用量を出力するかどうか，また，使用量がしきい値に達したことを知らせるメッセージを出力するかどうかを指定します。

ログファイルの出力形式と出力例を次に示します。

出力形式

```
[id] <date> (Skip Full:full_count, Copy:copy_count) [gc_kind gc_info, gc_time secs][Eden:
 eden_info][Survivor: survivor_info][Tenured: tenured_info][Metaspace: metaspace_info][cl
ass space: class_space_info]  [cause:cause_info] [User: user_cpu secs] [Sys: system_cpu s
ecs][IM: jvm_alloc_size, mmap_total_size, malloc_total_size][TC: thread_count][DOE: doe_a
lloc_size, called_count][CCI: cc_used_sizeK, cc_max_sizeK, cc_infoK]
```

説明

- id：JavaVM ログファイル識別子

  JavaVM ログファイル識別子を次の表に示します。この識別子は，ログをログの内容（機能）ごとにフィルタリングして調査する場合に利用できます。

  表 7-3　JavaVM ログファイル識別子

識別子	ログの内容
CCI	コードキャッシュ領域情報
CLT	クラスライブラリのスタックトレース
JCC	JIT コンパイル失敗情報
JMS	JIT コンパイルを抑制した JIT コンパイラスレッドに関する情報
OMH	OutOfMemory の発生頻度に関する情報
OOM	OutOfMemoryError 発生時の例外情報とスタックトレース
PTD	Survivor 領域の年齢分布情報
VGC	拡張 verbosegc 情報

- date：日時
- full_count：FullGC をスキップした回数（-XX:HitachiVerboseGCIntervalTime を指定した場合だけ）
- copy_count ：CopyGC をスキップした回数（-XX:HitachiVerboseGCIntervalTime を指定した場合だけ）
- gc_kind：GC 種別（Full GC または GC）
- gc_info：GC 情報（GC 前の領域長 -> GC 後の領域長(領域サイズ)）

  (例)　264K->0K(512K)
- gc_time：GC 経過時間（単位：秒）

- eden_info：Eden 情報
- survivor_info：Survivor 情報
- tenured_info：Tenured 情報
- metaspace_info：Metaspace 領域情報
- class_space_info：CompressedClassSpace 情報
- cause_info：GC の原因
- user_cpu secs：GC スレッドがユーザモードに費やした CPU 時間（単位：秒）
- system_cpu secs：GC スレッドがカーネルモードに費やした CPU 時間（単位：秒）
- jvm_alloc_size：JavaVM 内部で管理している領域のうち，現在使用中の領域のサイズ（mmap_total_size と malloc_total_size の合計サイズのうち，現在使用中の領域のサイズ）（-XX:+HitachiVerboseGCPrintJVMInternalMemory を指定した場合だけ）
- mmap_total_size：JavaVM 内部で管理している領域のうち，mmap で割り当てた C ヒープの総サイズ（-XX:+HitachiVerboseGCPrintJVMInternalMemory を指定した場合だけ）
- malloc_total_size：JavaVM 内部で管理している領域のうち，malloc で割り当てた C ヒープの総サイズ（-XX:+HitachiVerboseGCPrintJVMInternalMemory を指定した場合だけ）
- thread_count：Java スレッドの数（-XX:+HitachiVerboseGCPrintThreadCount を指定した場合だけ）
- doe_alloc_size：java.io.File.deleteOnExit()を呼び出して確保した累積のヒープサイズ（-XX:+HitachiVerboseGCPrintDeleteOnExit を指定した場合だけ）
- called_count：java.io.File.deleteOnExit()の呼び出し回数（-XX:+HitachiVerboseGCPrintDeleteOnExit を指定した場合だけ）
- cc_used_size：GC 時のコードキャッシュ領域の使用サイズ（単位：キロバイト）（-XX:+PrintCodeCacheInfo を指定した場合だけ）
- cc_max_size：コードキャッシュ領域の最大サイズ（単位：キロバイト）（-XX:+PrintCodeCacheInfo を指定した場合だけ）
- cc_info：保守情報（-XX:+PrintCodeCacheInfo を指定した場合だけ）

出力例

-XX:+HitachiCommaVerboseGC オプションを指定した場合の出力例を次に示します。

```
VGC,Fri Jan 23 21:37:50 2004,11,41,0,GC,16886,16886,65088,0.0559806,
4094,0,4096,447,447,448,12345,16439,60544,1116,1116,4096,0,0.0312500, 0.0156250,729,928,0
,509,2167,2054,2301,49152,2304
VGC,Fri Jan 23 21:37:55 2004,6,24,0,Full GC,65082,65082,65088,0.4294532,
4094,4094,4096,447,447,448,60541,60541,60544,1116,1116,4096,0,0.0156250, 0.0312500,729,92
8,0,509,16,170,2301,49152,2304
```

7.3.3　コードキャッシュ領域に関するログの内容

JavaVM は，呼び出し回数やループ回数が多い Java メソッドを JIT コンパイルして実行することで，処理の高速化を行います。JIT コンパイルによって生成された JIT コンパイルコードは，コードキャッシュ領域に配置されます。

通常，コードキャッシュ領域のサイズはデフォルト値で問題ありません。しかし，実行環境や Java アプリケーションの規模によっては，コードキャッシュ領域のサイズがデフォルト値では枯渇する場合があります。

コードキャッシュ領域が枯渇すると，JavaVM は JIT コンパイルを実行できなくなり，Java アプリケーションの実行で十分な性能が得られないおそれがあります。この場合は，「-XX:[+|-]PrintCodeCacheInfo (コードキャッシュ領域情報出力オプション)」や「-XX:[+|-]PrintCodeCacheFullMessage（コードキャッシュ領域枯渇メッセージ出力オプション)」を有効にして，コードキャッシュ領域の使用量や出力されるメッセージを監視するようにしてください。

これらのオプションについては，「-XX:[+|-]PrintCodeCacheInfo（コードキャッシュ領域情報出力オプション)」および「-XX:[+|-]PrintCodeCacheFullMessage（コードキャッシュ領域枯渇メッセージ出力オプション)」を参照してください。

(1)　コードキャッシュ領域の使用量がしきい値に達したことを知らせるメッセージの出力内容

コードキャッシュ領域の使用量がしきい値に達したことを知らせるメッセージの出力形式を次に示します。

```
[cc_id]<cc_date>CodeCache usage has exceeded the threshold.[cc_used_sizeK, cc_max_sizeK, cc_infoK]
```

出力項目について次に示します。

出力項目	説明
cc_id	CCI（JavaVM ログファイル識別子）が出力されます。
cc_date	JIT コンパイルを実行した日時が出力されます。
cc_used_size	JIT コンパイル後のコードキャッシュ領域の使用サイズが出力されます（単位：キロバイト)。
cc_max_size	コードキャッシュ領域の最大サイズが出力されます（単位：キロバイト)。
cc_info	保守情報が出力されます。

(2)　コードキャッシュ領域が枯渇したことを知らせるメッセージの出力内容

コードキャッシュ領域が枯渇したことを知らせるメッセージの出力形式を次に示します。

```
[cc_id]<cc_date>CodeCache is full. Compiler has been disabled.[cc_used_sizeK, cc_max_sizeK, cc_infoK]
```

出力項目について次に示します。

出力項目	説明
cc_id	CCI（JavaVM ログファイル識別子）が出力されます。
cc_date	Java メソッドが JIT コンパイルの対象になった日時が出力されます。
cc_used_size	Java メソッドが JIT コンパイルの対象になったときのコードキャッシュ領域の使用サイズが出力されます（単位：キロバイト）。
cc_max_size	コードキャッシュ領域の最大サイズが出力されます（単位：キロバイト）。
cc_info	保守情報が出力されます。

7.4　日立 JavaVM が出力するメッセージログ（標準出力およびエラーリポートファイル）

日立 JavaVM でクラッシュが発生した場合，日立 JavaVM によってデバッグ情報が標準出力とエラーリポートファイルに出力されます。

エラーリポートファイルに出力されるのは，次の場合です。

- JNI の中でのシグナルが発生したとき
- JavaVM で C ヒープ不足が発生したとき
- JavaVM で予期しないシグナルが発生したとき
- JavaVM で Internal Error（内部論理エラー）が発生したとき

ここでは，次の場合のメッセージログの出力内容について説明します。

表 7-4　日立 JavaVM が出力するメッセージログ

メッセージの種類	出力先
JNI の中でのシグナルが発生した場合，または JavaVM でのメッセージ※	標準出力 エラーリポートファイル
C ヒープが不足した場合のメッセージ※	標準出力 エラーリポートファイル
Internal Error が発生した場合のメッセージ※	標準出力 エラーリポートファイル
スレッド作成に失敗した場合のメッセージ※	標準出力

注※　日立 JavaVM 独自の出力先または出力内容があります。

なお，スレッド作成に失敗した場合のメッセージは，標準出力だけに出力されます。

7.4.1　シグナルが発生した場合

シグナルが発生した場合，次に示す項目がログ出力されます。出力内容には，日立 JavaVM として拡張された内容が含まれます。

- 異常終了位置とシグナル種別※
- 時間情報※
- カレントスレッド情報
- スタックトレース
- シグナル情報の格納先アドレス※

- シグナル情報
- siginfo 情報※
- レジスタ情報
- スタックの先頭から格納されている情報
- スレッド情報
- VM の状態
- メモリ情報※
- Java ヒープの使用状況※
- Card table マップアドレスの表示
- Polling page アドレスの表示
- Large pages 確保失敗情報
- CodeCache 情報
- Event 情報
- ライブラリ
- コマンドおよび VM パラメタ※
- 環境変数
- 登録済みシグナルハンドラ
- マシン情報※
- システム名，CPU，実メモリ，および VM 情報
- javatrace 起動コマンドのコマンドライン※

注※　製品で拡張された出力内容です。

それぞれの出力内容について説明します。

(1)　異常終了位置とシグナル種別

異常終了時の状態に応じて，次のどちらかの内容が出力されます。この内容は，日立 JavaVM で拡張された出力内容です。

(a)　シグナルを検出した場合

次のメッセージが出力されます。

```
#
# A fatal error has been detected by the Java Runtime Environment:
```

次の内容が出力されます。

シグナルを検出した場合の出力内容

```
#
# A fatal error has been detected by the Java Runtime Environment:
#
#  〈発生したシグナル名〉 (〈シグナル番号〉) at pc=〈PCアドレス〉, pid=〈プロセスID〉, tid=〈スレッドID〉
#
# JRE version: 〈jreバージョン情報〉
# Java VM: Java HotSpot(TM) 〈VM種別〉 (〈Oracleバージョン情報〉-〈日立バージョン情報〉-〈ビルド年月日〉 mixed mode, 〈オプション情報〉, 〈GC種別〉, 〈OS名〉-〈CPU種別〉)
# Problematic frame:
# 〈種別コード〉 [〈シグナルが発生したライブラリ名〉+〈オフセット〉]
#
```

注

シグナルが発生した関数名が取り出せた場合，「<シグナルが発生したライブラリ名>＋<オフセット>」に続いて，その関数名とオフセットが表示されることがあります。

(b) 内部論理エラーが発生した場合

次の内容が出力されます。

内部論理エラーが発生した場合の出力内容

```
#
#  Internal Error (〈ファイル名〉:〈行数〉 または, Internal Errorのコード), pid=〈プロセスID〉, tid=〈スレッドID〉
#  〈内部論理エラー種別〉: 内部論理エラーメッセージ
#
#  JRE version: 〈jreバージョン情報〉
#  Java VM: Java HotSpot(TM) 〈VM種別〉 (〈Oracleバージョン情報〉-〈日立バージョン情報〉-〈ビルド年月日〉 mixed mode, 〈オプション情報〉, 〈GC種別〉, 〈OS名〉-〈CPU種別〉)
#
#  〈coreファイル情報〉
```

注

Internal Error には，ファイル名および行数の組み合わせ，または Internal Error のコードが出力されます。<内部論理エラー種別>には，発生した内部論理エラーの種類によって，"fatal error"，"guarantee(<論理式>) failed"，または"Error"のどれかが出力されます。

(2) 時間情報

次の内容が出力されます。この内容は，日立 JavaVM で拡張された出力内容です。

```
Time: 〈実行日付〉 elapsed time: 〈実行時間〉 seconds 〈(実行時間 形式出力)〉
```

注

実行日付の例を次に示します。

例：「Wed Aug 25 14:55:04 2004」

秒単位だけでは実行時間が把握しづらいため，<(実行時間 形式出力)>ではユーザが読みやすいように，(DAYS HOURS MINS SECS)フォーマットで出力します。

(例) elapsed time: 900 seconds (0d 0h 15m 0s)

(3) カレントスレッド情報

スレッドの種類に応じて次の 3 種類の情報が出力されます。

```
Current thread (<アドレス>): <スレッド名> "<スレッド名称>" [_<状態>, id=<スレッドID>, stac
k(<開始アドレス>,<終了アドレス>)]

または
Current thread (<アドレス>): <スレッド名> [_id=<スレッドID>, stack(<開始アドレス>,<終了アド
レス>)]

または
Current thread is native thread
```

(4) スタックトレース

次の内容が出力されます。ただし，Current thread が JavaThread 以外の場合は出力されません。

```
Java frames: (J=compiled Java code, j=interpreted, Vv=VM code)
 <スタックトレース>
 :
```

(5) シグナル情報の格納先アドレス

次の内容が出力されます。この内容は，日立 JavaVM で拡張された出力内容です。

```
siginfo address: <アドレス>, context address: <アドレス>
```

(6) シグナル情報

次の内容が出力されます。

```
siginfo: si_signo=<発生したシグナル番号> (<発生したシグナル名>), si_code: <番号> (<シグナル
理由種別>)[, si_errno: <番号>] (, si_pid: <プロセスID> [(current process) | (invalid)] , si
_uid: <ユーザID> [, si_status: <終了値またはシグナル番号>]) | (, si_addr: <アドレス>) | (, si
_band: <SIGPOLLのバンド番号>)
```

(7) siginfo 情報

次の内容が出力されます。この内容は，日立 JavaVM で拡張された出力内容です。

```
siginfo structure dump (location: <siginfoのアドレス>)
<siginfoのアドレス> <siginfoのアドレス> <siginfoのアドレス> <siginfoのアドレス>
 :
<siginfoのアドレス> <siginfoのアドレス> <siginfoのアドレス> <siginfoのアドレス>
```

注

<siginfo のアドレス>は 16 進数で出力されます。

(8) レジスタ情報

次の内容が出力されます。ただし，内部論理エラーの場合は出力されません。

```
Registers:<レジスタ情報>
 :
```

注

BSP レジスタの値がデバッガ（gdb）の値と異なって出力されます。これは，デバッガでは BSP の指すバッキングストア領域の内容を出力しており，BSP の指す位置を修正しているためです。

(9) スタックの先頭から格納されている情報

次の内容が出力されます。ただし，内部論理エラーの場合は出力されません。

```
Top of Stack: (sp=<スタックポインタのアドレス>)
<アドレス>:   <格納されている内容>
 :
```

注

<格納されている内容>は 16 進数で出力されます。

(10) スレッド情報

次の内容が出力されます。

```
Threads class SMR info:
_java_thread_list=<スレッドリストアドレス>, length=<スレッドリスト長>, elements={
<スレッドアドレス1>,<スレッドアドレス2>,<スレッドアドレス3>,<スレッドアドレス4>
 :
}
_to_delete_list=<スレッドリストアドレス>, length=<スレッドリスト長>, elements={
<スレッドアドレス1>,<スレッドアドレス2>,<スレッドアドレス3>,<スレッドアドレス4>,
 :
}
next-> <スレッドリストアドレス>, length=<スレッドリスト長>, elements={
```

```
<スレッドアドレス1>,<スレッドアドレス2>,<スレッドアドレス3>,<スレッドアドレス4>,
 :
}

Java Threads: ( => current thread )
  <アドレス> JavaThread “スレッド名称“ [<状態>, id=<スレッドID>, stack(<開始アドレス>,<終了アドレス>)]
 :
=><アドレス> JavaThread “スレッド名称“ [<状態>, id=<スレッドID>, stack(<開始アドレス>,<終了アドレス>)]

Other  Threads:
  <アドレス> <スレッド名> [stack(<開始アドレス>,<終了アドレス>)] [id=<スレッドID>]
 :
```

(11) VMの状態

次の内容が出力されます。

```
VM state:<現在の状態>

VM Mutex/Monitor currently owned by a thread: <mutexs/moniter>
```

注

この情報に続いて，ロック情報が出力される場合があります。

(12) メモリ情報

次の内容が出力されます。この内容は，日立JavaVMで拡張された出力内容です。

```
Memory :
<メモリ確保関数>:address<開始アドレス> - <終了アドレス>(size:<サイズ>)
         :

        Heap Size:<確保しているメモリサイズ>
        Alloc Size:<使用中のメモリサイズ>
        Free Size:<未使用のメモリサイズ>
```

<メモリ確保関数>は，mmap()またはmalloc()のどちらかです。アドレスは16進数で表示されます。

各種メモリサイズの単位はバイトです。

(13) Javaヒープの使用状況

次の内容が出力されます。この内容は，日立JavaVMで拡張された出力内容です。

```
Heap※
<Javaヒープ情報>
```

注※

見出し部分は拡張スレッドダンプとエラーリポートファイルで次のように異なります。

拡張スレッドダンプの場合	Heap Status ----------
エラーリポートファイルの場合	Heap

(14) Card table マップアドレスの表示

次の内容が出力されます。

```
Card table byte_map: [<アドレス>,<アドレス>] _byte_map_base: <アドレス>
```

(15) Polling page アドレスの表示

次の内容が出力されます。

```
Polling page: <アドレス>
```

(16) Large pages 確保失敗情報

mmap()関数でのメモリ確保に失敗した場合に，失敗回数が出力されます。

```
Large page allocation failures have occurred  <回数> times
```

(17) CodeCache 情報

次の内容が出力されます。

SegmentedCodeCache オプションが ON の場合

```
CodeHeap 'non-nmethods': size=<全体サイズ> used=<使用済みサイズ> max_used=<最大サイズ> free=
<空きサイズ>
 bounds [<bottom>, <commit addr>, <reserve addr>]
CodeHeap 'profiled nmethods': size=<全体サイズ> used=<使用済みサイズ> max_used=<最大サイズ>
free=<空きサイズ>
 bounds [<bottom>, <commit addr>, <reserve addr>]
CodeHeap 'non-profiled nmethods': size=<全体サイズ> used=<使用済みサイズ> max_used=<最大サイ
ズ> free=<空きサイズ>
 bounds [<bottom>, <commit addr>, <reserve addr>]
 total_blobs=<CodeBlobの総数> nmethods=<nmethodsの総数> adapters=<adapterの総数>
 compilation: <enabled/disabled>
          stopped_count=<コンパイラ停止回数>, restarted_count=<コンパイラ再開回数>
 full_count=<コードキャッシュ領域が枯渇した回数>
```

SegmentedCodeCache オプションが OFF の場合

```
CodeCache: size=<全体サイズ> used=<使用済みサイズ> max_used=<最大サイズ> free=<空きサイズ>
 bounds [<bottom>, <commit addr>, <reserve addr>]
 total_blobs=<CodeBlobの総数> nmethods=<nmethodsの総数> adapters=<adapterの総数>
 compilation: <enabled/disabled>
          stopped_count=<コンパイラ停止回数>, restarted_count=<コンパイラ再開回数>
 full_count=<コードキャッシュ領域が枯渇した回数>
```

(18) Event 情報

イベントバッファの内容がすべて出力されます。

```
<イベント種別名>(<イベント数> events):
<イベントレコード>
    :
```

イベント情報はリングバッファで管理していて，イベント種別ごとに保持するイベント数の最大値は 10 です。イベント数が 0 の場合は<イベントレコード>には"No events"が出力されます。

出力できる<イベント種別>の出力例を次に示します。

- Compilation events

  JIT コンパイル情報

  ```
  Compilation events (10 events):
  Event: 0.923 Thread 0x00002aaab2f01800  389    b        java.io.FileOutputStream::write (
  12 bytes)
  Event: 0.923 Thread 0x00002aaab2f01800 nmethod 389 0x00002aaaac3ea490 code [0x00002aaaac3
  ea5e0, 0x00002aaaac3ea668]
      :
  ```

- GC Heap History

  before GC，after GC 情報

  ```
  GC Heap History (4 events):
  Event: 23.719 GC heap before
  {Heap before GC invocations=0 (full 0):
   def new generation   max 154880K, total 9664K, used 345K (0.2% used/max, 3.6% used/total
  )
      :
  ```

- Deoptimization events

  Deopt 情報

  ```
  Deoptimization events (10 events):
  Event: 0.818 Thread 0x00002aaaaba7e000 Uncommon trap 24 fr.pc 0x00002aaaac3d1eec
  Event: 0.818 Thread 0x00002aaaaba7e000 Uncommon trap 54 fr.pc 0x00002aaaac3d0dd8
      :
  ```

- Classes redefined

  クラス再定義情報

```
Classes redefined (3 events):
Event: 0.297 Thread 0x000000517b7b2800 redefined class name=foo.Foo, count=1
Event: 0.303 Thread 0x000000517b7b2800 redefined class name=foo.Foo, count=2
Event: 0.310 Thread 0x000000517b7b2800 redefined class name=foo.Foo, count=3
     :
```

- Internal exceptions

  internal exception 情報

```
Internal exceptions (2 events):
Event: 0.025 Thread 0x00002aaaaba7e000 Threw 0x00000000db606140 at /hotspot/src/share/vm/
prims/jni.cpp:4008
Event: 0.061 Thread 0x00002aaaaba7e000 Threw 0x00000000db649980 at /hotspot/src/share/vm/
prims/jvm.cpp:1167
     :
```

- Events

  クラスローダ情報など

```
Events (10 events):
Event: 0.080 loading class 0x00002aaab302e990
Event: 0.080 loading class 0x00002aaab302e990 done
Event: 4.286 Executing VM operation: EnableBiasedLocking
Event: 4.286 Executing VM operation: EnableBiasedLocking done
     :
```

(19) ライブラリ

次の内容に続いて，ローディングされているライブラリの一覧が出力されます。

```
Dynamic libraries:
<ライブラリ>
 :
```

(20) コマンドおよびVMパラメタ

次の内容が出力されます。この内容は，日立JavaVMで拡張された出力内容です。

```
Command : <コマンドライン>

Java Home Dir   : <JDK実行環境インストールディレクトリ>
Java DLL Dir    : <JDKのライブラリインストールディレクトリ>
Sys Classpath   : <システムクラスパス>
User Args       :
<コマンドオプション1>
<コマンドオプション2>
 :
```

(21) 環境変数

次の内容が出力されます。

```
Environment Variables:
<環境変数=値>
 :
```

(22) 登録済みシグナルハンドラ

次の内容が出力されます。

```
Signal Handlers:
<シグナル種別>:
 [<シグナルハンドラアドレス>], sa_mask[0]=<シグナルマスク>, sa_flags=<特殊フラグ>
 :
Changed Signal Handlers -
<シグナル種別>: [<シグナルハンドラアドレス>], sa_mask[0]=<シグナルマスク>, sa_flags=<特殊フラグ>
 :
```

出力内容の意味は次のとおりです。

- <シグナル種別>：/usr/include/sys/signal.h に定義されているシグナル名です。
- <シグナルハンドラアドレス>：シグナルハンドラのアドレスが 16 進で出力された値です。「ライブラリ名＋オフセット」という形式で表示されることもあります。
- <シグナルマスク>：sigaction()で取り出せる構造の sa_mask フィールド値が 16 進で出力された値です。
- <特殊フラグ>：sigaction()で取り出せる構造の sa_flags フィールド値が 16 進で出力された値です。

(23) マシン情報

次の内容が出力されます。この内容は，日立 JavaVM で拡張された出力内容です。

```
Host: <ホスト名>:<IPアドレス>
```

注

<IP アドレス>には複数の IP アドレスが表示されることがあります。

(24) システム名，CPU，実メモリ，および VM 情報

次の内容が出力されます。

```
OS: <OSバージョン>

[uname:<uname出力>]
[libc:<libcのバージョン番号>]
[rlimit:<リミット値>]
[load average:<ロードアベレージ>]
[/proc/meminfo:</proc/meminfoの内容>]
```

```
CPU: <利用可能CPU数>[,<CPU種別>]

Memory: <実メモリ情報>

vm_info: <VM情報>
```

(25) javatrace 起動コマンドのコマンドライン

次の内容が出力されます。この内容は，日立 JavaVM で拡張された出力内容です。

```
# You can get further information from javatrace.log file generated
# by using javatrace command.
# usage: javatrace core-file-name loadmodule-name [out-file-name] [-l(library-name)...]
# Please use javatrace command as follows and submit a bug report
# to Hitachi with javatrace.log file:
#[<インストールディレクトリ>/bin/javatrace <coreファイル> <ロードモジュール>]
```

7.4.2 C ヒープが不足した場合

C ヒープが不足した場合，次の順序でメッセージ出力およびダンプ出力，または core ダンプの生成が実行されます。

1. C ヒープ不足を示すメッセージログが，エラーリポートファイルおよび標準出力に出力されます。
2. 1.の実行中にメモリ不足が発生した場合，簡易メッセージが標準出力に出力されます。
3. 簡易メッセージの出力処理中にさらにメモリ不足が発生したときに，メッセージおよびエラーログファイルの出力処理を中止して，core ダンプを生成します。

それぞれの出力形式を次に示します。

(1) C ヒープ不足を示すメッセージログの出力内容

C ヒープ不足を示すメッセージログの出力形式を次に示します。この形式は，エラーリポートファイルと標準出力で共通です。

```
Exception in thread <ThreadName> java.lang.OutOfMemoryError: requested <n> bytes [for <messa
ge>].

Memory Status
-------------
  maximum size of data segment
      soft(current) limit :getrlimit(RLIMIT_DATA)で取得したソフトリミット値 kbytes (16進数)
      hard limit          :getrlimit(RLIMIT_DATA)で取得したハードリミット値 kbytes (16進数)
  current end of the heap :sbrk(0)によって取得した値
  JVM allocation size by malloc :JavaVMが割り当てたメモリサイズ kbytes (16進数)
  malloc information
      total space in arena            :mallinfo.arenaの値
      number of ordinary blocks       :mallinfo.ordblksの値
```

```
      number of small blocks          :mallinfo.smblksの値
      number of holding blocks        :mallinfo.hblksの値
      space in holding block headers  :mallinfo.hblkhdの値
      space in small blocks in use    :mallinfo.usmblksの値
      space in free small blocks      :mallinfo.fsmblksの値
      space in ordinary blocks in use :mallinfo.uordblksの値
      space in free ordinary blocks   :mallinfo.fordblksの値
      cost of enabling keep option    :mallinfo.keepcostの値

Heap Status
-----------
〈Javaヒープ情報〉
-----------
Stack Trace
-----------
〈スタックトレース〉
-----------

JVM Internal Memory Status
--------------------------
〈独自メモリ管理機能で管理している領域情報〉
```

このようなメッセージが出力された場合は，C ヒープを減らすなど，適切な対策をしてください。

出力項目について次に示します。

表 7-5　C ヒープが不足した場合のメッセージログの出力項目

出力項目	説明
ThreadName	Thread#getName()メソッドで取り出せるスレッド名称が出力されます。
n	メモリ確保要求サイズが出力されます。
message	保守員による調査に必要な内部メッセージが出力されます。出力されない場合もあります。
Java ヒープ情報	Java ヒープの使用状況が出力されます。
スタックトレース	メモリ不足の発生したスレッドが Java コードを実行しているスレッドである場合に，スタックトレースが出力されます。 コンパイル処理のような JavaVM の内部処理を実行するスレッドでメモリ不足が発生した場合には出力されません。

(2)　メモリ不足を示すメッセージの出力内容

C ヒープ不足を示すメッセージログが出力されている間にさらにメモリ不足が発生した場合は，処理の続行ができません。この場合は，次の形式の簡易メッセージが標準出力に出力されます。

```
java.lang.OutOfMemoryError:requested 〈n〉 bytes for 〈message〉
```

出力項目について次に示します。

表 7-6　メモリ不足が発生した場合の簡易メッセージの出力項目

出力項目	説明
n	メモリ確保要求サイズが出力されます。
message	保守員による調査に必要な内部メッセージが出力されます。

(3)　core ダンプの生成を示すメッセージの出力内容

簡易メッセージの出力処理中にさらにメモリ不足が発生した場合，メッセージおよびエラーログファイルの出力処理を中止して，core ダンプを生成します。core ダンプが生成されると，次の形式のメッセージが標準出力に出力されます。

```
Can't create logs because of memory shortage.
Insufficient memory for malloc. JVM generates core file
```

7.4.3　Internal Error が発生した場合

JavaVM 内部の論理エラーである Internal Error が発生した場合，次の情報が出力されます。

- 異常終了位置とシグナル種別
- 時間情報※
- カレントスレッド情報
- シグナル情報の格納先アドレス※
- スレッド情報
- VM の状態
- メモリ情報※
- ヒープ情報※
- Card table マップアドレス
- Polling page アドレス
- Large pages 確保失敗情報
- CodeCache 情報
- Event 情報
- ライブラリ
- コマンド・VM パラメタ※
- 環境変数
- 登録済みシグナルハンドラ

- マシン情報※
- システム名，CPU，実メモリ，および VM 情報
- javatrace 起動コマンドのコマンドライン※

注※　製品で拡張された出力内容です。

それぞれの情報の出力形式については，「7.4.1　シグナルが発生した場合」を参照してください。

7.4.4　スレッド作成に失敗した場合

メモリ不足（OutOfMemoryError）などが発生して新しくスレッドを作成できなかった場合，その時点でのスレッド数が，標準出力に出力されます。ここには，作成に失敗したスレッド数も含まれます。

メモリ不足などによってスレッドの作成に失敗した場合の出力例を次に示します。

```
java.lang.OutOfMemoryError:unable to create new native thread.1200 threads exist.
…
```

JavaVM 起動時にスレッド作成に失敗した場合の出力例を次に示します。

```
Error occurred during initialization of VM
Could not create thread for VM:VM Thread.5 threads exist.
```

7.5　JavaVM スタックトレース情報

ここでは，スタックトレースに出力される情報のうち，日立 JavaVM で拡張された内容について説明します。

サーバやアプリケーションでトラブルが発生した場合，トラブル発生までのスタックトレースの内容を確認することで，トラブル発生の要因を調査できます。

スタックトレースは，次のどれかのタイミングで出力されます。

- アプリケーションサーバまたはアプリケーションで例外が発生した場合
- バッチサーバまたはバッチアプリケーションで例外が発生した場合
- JavaVM のスレッドダンプが出力された場合

日立 JavaVM では，サーバを起動するときのオプションの指定によって，スタックトレースに Java メソッドのローカル変数についての情報を出力できます。例外発生時に Java メソッドに定義されていたローカル変数の情報は，トラブルの要因を分析するために有効です。

なお，ここで対象にする**ローカル変数**とは，メソッドに渡される引数およびインスタンスメソッドでの呼び出しの対象になるオブジェクト（this）のことです。**ローカル変数情報**には，これらのローカル変数の変数名，型名，およびローカル変数の値が出力されます。なお，型名は，基本形名，クラス名（インタフェース名を含みます）または配列形名のことです。

スタックトレースにローカル変数情報を出力するためのオプションを，次の表に示します。起動オプションについては「5.　日立 JavaVM 起動オプション」を，起動オプションのデフォルト値についてはマニュアル「uCosminexus Application Runtime ユーザーズガイド」を参照してください。また，日立 JavaVM の資料取得の方法については，「3.　トラブルシューティング」を参照してください。

表 7-7　スタックトレースにローカル変数情報を出力するためのオプション

起動オプション	スタックトレースを出力するタイミング	同時に指定できるオプション
-XX:+HitachiLocalsInThrowable	サーバまたはアプリケーション内で例外が発生したとき※	• -XX:+HitachiLocalsSimpleFormat • -XX:+HitachiTrueTypeInLocals • -XX:HitachiCallToString=<適用範囲指定>
-XX:+HitachiLocalsInStackTrace	JavaVM のスレッドダンプを出力したとき	• -XX:+HitachiLocalsSimpleFormat • -XX:+HitachiTrueTypeInLocals

注※　ただし，発生した例外が「java.lang.StackOverflowError」または「java.lang.OutOfMemoryError」の場合，スタックトレースにローカル変数は出力されません。

以降に，それぞれのオプションを指定した場合に出力される内容について，例を基に説明します。それぞれのオプションを指定したときに出力される項目の詳細については，次の個所を参照してください。

- 「-XX:[+|-]HitachiLocalsInThrowable（例外発生時のローカル変数情報収集オプション）」
- 「-XX:[+|-]HitachiLocalsInStackTrace（スレッドダンプ出力時のローカル変数出力オプション）」

7.5.1　-XX:+HitachiLocalsInThrowable オプションが指定されている場合

java.lang.Throwable.printStackTrace メソッドで出力されるスタックトレース情報の 1 スタックフレーム情報ごとに，そのスタックフレームに対応するメソッド内のローカル変数情報が挿入されて出力されます。

(1)　標準の形式および簡易出力フォーマットでの出力例

ローカル変数を出力するための機能として，-XX:+HitachiLocalsInThrowable オプションだけを指定している場合の出力例です。

Java プログラムの例と，それに対するスタックトレース内のローカル変数情報の出力例を示します。

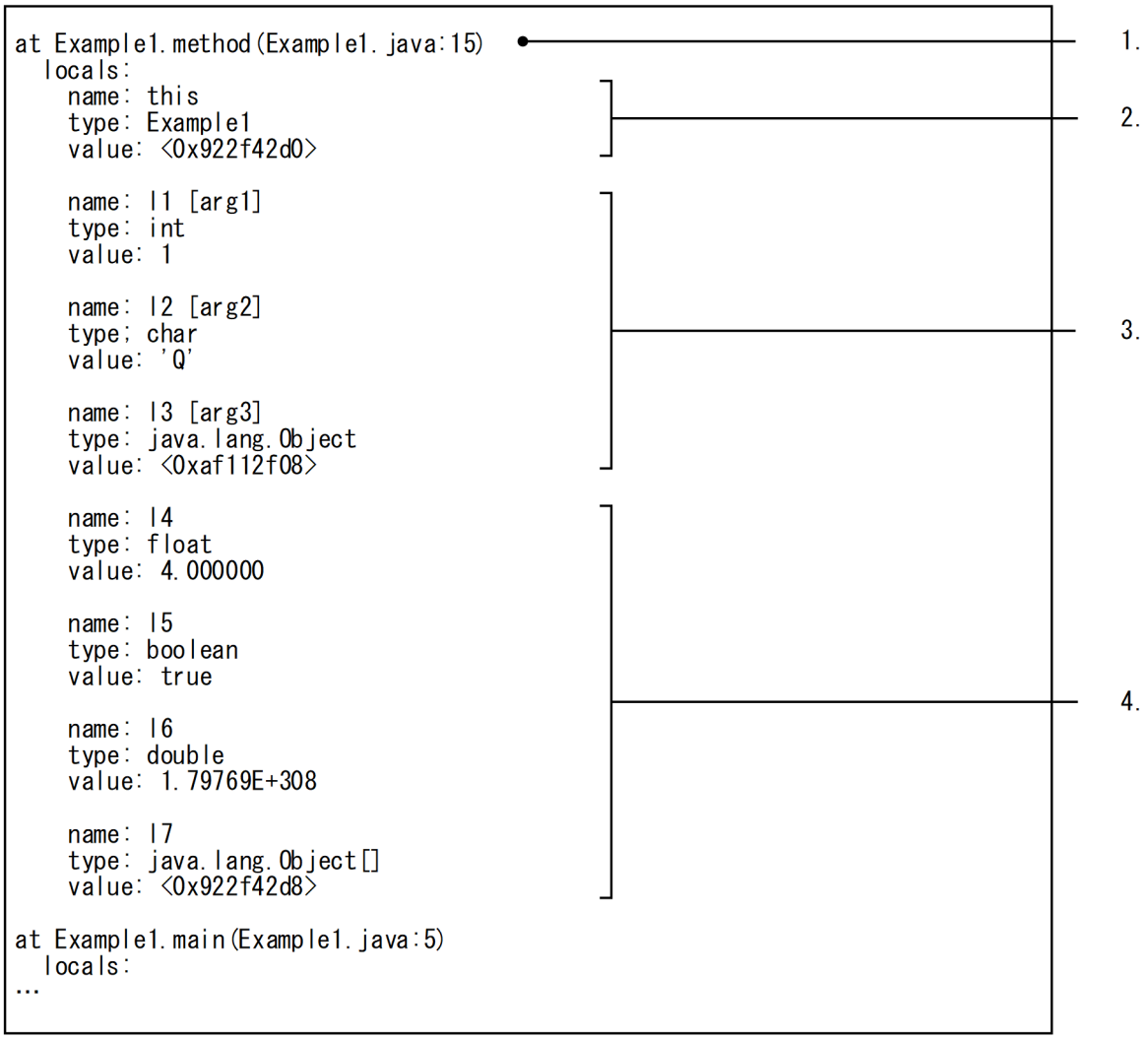
Java プログラムの例 1

```
class Example1 {
    public static void main(String[] args) {
        Example1 e1 = new Example1();
        Object obj = new Object();
        e1.method(1, 'Q', obj); // e1.methodメソッドを実行します（5行目）。
    }

    void method(int l1, char l2, Object l3) {
        float l4 = 4.0f;
        boolean l5 = true;
        double l6 = Double.MAX_VALUE;
        Object[] l7 = new Object[10];

        try {
          <例外発生!> // methodメソッドの処理内で例外が発生した場合の処理です（15行目）。
        } catch (Exception e) {
          e.printStackTrace(); // スタックトレース情報を出力します（17行目）。
        }
    }
}
```

出力例を次に示します。

この例は，Java プログラムの例 1 の 5 行目で実行した e1.method メソッドの処理内で例外が発生したときに，17 行目の e.printStackTrace メソッドによって出力されるスタックトレース情報です。

図 7-1　Java プログラムの例 1 に対するローカル変数情報の出力例（-g オプションまたは-g:vars オプションを指定して作成された class ファイルの場合）



出力内容について説明します。

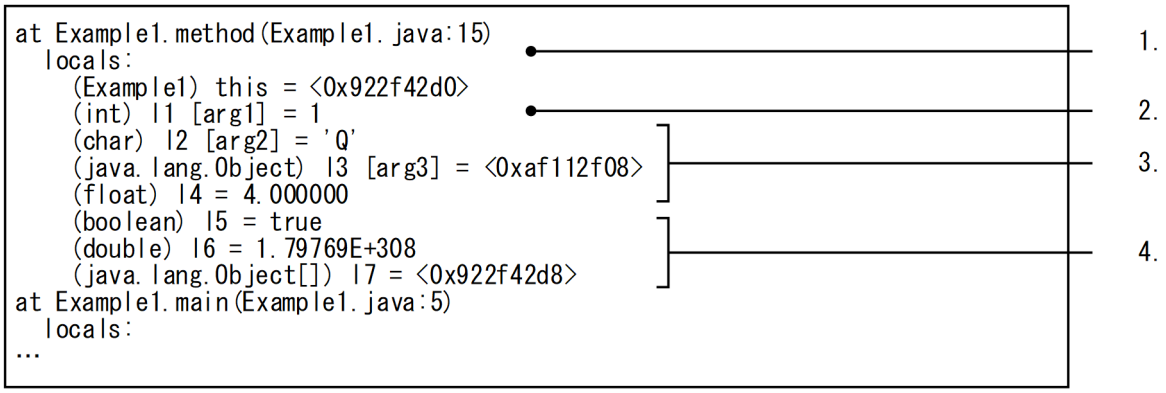
1. スタックトレースを出力する処理を実行したメソッドの情報が出力されます。この例では，Java プログラム 1 の 15 行目で例外が発生したときのスタックトレース情報が出力されることを示しています。
2. ローカル変数情報として，インスタンスメソッドの呼び出しの対象になるオブジェクトについての情報が出力されます。この例の場合は，Java プログラムの例 1 の 3 行目で作成した Example1 クラスのオブジェクトのクラス名とアドレスが出力されます。
3. ローカル変数情報として，method メソッドの引数として指定されたローカル変数の値についての情報が出力されます。変数名の後ろの[arg*]は，何番目のメソッド引数かを示す情報です。Java プログラムの例 1 の 5 行目で e1.method メソッドを実行したときに指定された値が出力されます。

   なお，l1 および l2 は基本型（int 型および char 型）の変数なので実際の値が出力されます。l3 は java.lang.Object クラス型の変数なので，アドレスで出力されます。
4. ローカル変数情報として，method メソッド内のローカル変数のうち，メソッド引数として指定した以外のローカル変数の値についての情報が出力されます。

なお，l4～l6 は基本型（float 型，boolean 型および double 型）の変数なので実際の値が出力されます。l7 は java.lang.Object クラス型の変数なので，アドレスで出力されます。

次に，-XX:+HitachiLocalsSimpleFormat オプションを指定した場合の，簡易出力フォーマットでの出力例を示します。なお，出力内容の説明は，標準の形式と同じです。

図 7-2　-XX:+HitachiLocalsSimpleFormat オプションを指定した場合の出力例

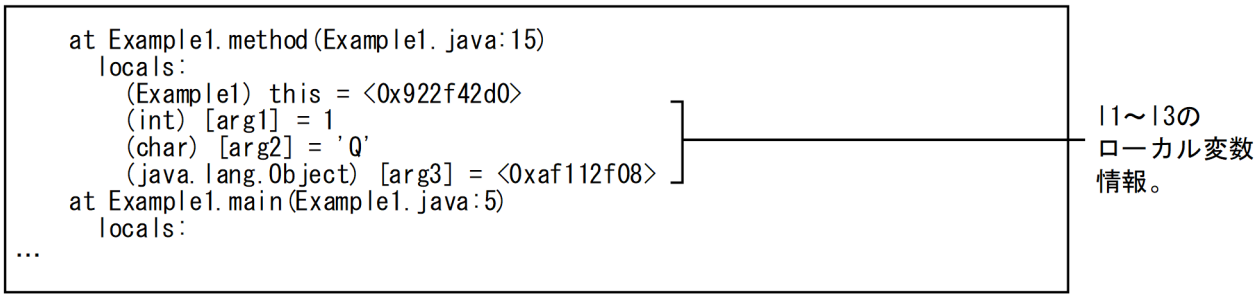


また，javac コマンド実行時に，-g オプションまたは-g:vars オプションを指定しなかった場合，ローカル変数情報がないため，出力内容が次のように制限されます。これらは，native メソッドを実行した場合も同様です。

- 出力できるローカル変数が，メソッドに渡される引数とインスタンスメソッド呼び出し対象オブジェクト（this）だけになります。
- メソッドに渡される引数は，変数名が出力されません。引数番号だけが出力されます。
- native メソッドの場合，ローカル変数の値として native メソッドを呼び出した時点での値が出力されます。native メソッドの実行結果を反映した出力結果にはなりません。

ローカル変数情報がない場合の，Java プログラムの例 1 に対する出力例を次に示します。ここでは，簡易出力フォーマットでの出力例を示します。

図 7-3　ローカル変数情報がない場合の出力例（簡易出力フォーマット）



「図 7-2」に比べて，次の違いがあります。

- メソッドに渡される引数（l1～l3）の変数名が出力されません。

- メソッド呼び出し時点の値が出力されるため，メソッド内で設定されたローカル変数についての情報（l4～l7）が出力されません。

(2) クラスまたは配列型の変数を文字列として出力する場合の出力例

出力するローカル変数がクラスまたは配列型の場合，アドレスだけの値表現ではトラブルシューティングに必要な情報が取得できない場合があります。このとき，-XX:HitachiCallToString オプションの指定をしておくと，クラスまたは配列型の変数の値を文字列で取得できます。オプションには，適用範囲として，minimal または full が指定できます。

-XX:HitachiCallToString=minimal オプションが指定されている場合は，java.lang パッケージ内のクラスのうち，String，StringBuffer，Boolean，Byte，Character，Short，Integer，Long，Float，または Double が対象になります。-XX:HitachiCallToString=full オプションが指定されている場合は，すべてのクラスが対象になります。

次に，Java プログラムの例と，-XX:HitachiCallToString オプションが指定されている場合の出力例を示します。なお，ここでは簡易出力フォーマットで示します。

Java プログラムの例 2

```
class Example2 {
    public static void main(String[] args) {
        Example2 e2 = new Example2();
        e2.method();// e2.methodメソッドを実行します（4行目）。
    }

    void method() {
        String l1 = ”local 1”;
        StringBuffer l2 = new StringBuffer(l1);
        l2.append(” + local 2”);
        Boolean l3 = new Boolean(false);
        Character l4 = new Character(’X’);
        Long l5 = new Long(Long.MIN_VALUE);
        Object l6 = new Thread();
        Object[] l7 = new Thread[10];

        try {
          〈例外発生!〉 // methodメソッドの処理内で例外が発生した場合の処理です（18行目）。
        } catch (Exception e) {
          e.printStackTrace(); // スタックトレース情報を出力します（20行目）。
        }
    }

    public String toString() {
        return ”I am an Example2 instance.”;
    }
}
```

-XX:HitachiCallToString=minimal オプションを指定した場合の出力例を次に示します。

この例は，Java プログラムの例 1 の 4 行目で実行した e2.method メソッドの処理内で例外が発生したときに，20 行目の e.printStackTrace メソッドによって出力されるスタックトレース情報です。

図 7-4　-XX:HitachiCallToString=minimal オプションを指定した場合の出力例（簡易出力フォーマット）

```
at Example2.method(Example2.java:18)
  locals:
    (Example2) this = <0xaa07db58>
    (java.lang.String) l1 = <0xae173a28> "local 1"
    (java.lang.StringBuffer) l2 = <0xaa07dca0> "local 1 + local 2"
    (java.lang.Boolean) l3 = <0xaa07de18> "false"
    (java.lang.Character) l4 = <0xaa07df68> "X"
    (java.lang.Long) l5 = <0xaa07e078> "-9223372036854775808"
    (java.lang.Object) l6 = <0xaa07e1a8>
    (java.lang.Object[]) l7 = <0xaa07e298>
at Example2.main(Example2.java:4)
  locals:
...
```

1.
2.

出力内容について説明します。

1. クラス型のローカル変数のうち，文字列を出力する適用対象のクラスの型のローカル変数情報です。アドレスに続いて，文字列に変換した値が出力されます。
2. クラス型のローカル変数のうち，文字列を出力する適用対象外のクラスの型のローカル変数情報です。アドレスだけが出力されます。

次に，-XX:HitachiCallToString=full オプションを指定した場合の出力例を示します。

図 7-5　-XX:HitachiCallToString=full オプションを指定した場合の出力例（簡易出力フォーマット）

```
at Example2.method(Example2.java:18)
  locals:
    (Example2) this = <0xaa07db58> "I am an Example2 instance."
    (java.lang.String) l1 = <0xae173a28> "local 1"
    (java.lang.StringBuffer) l2 = <0xaa07dca0> "local 1 + local 2"
    (java.lang.Boolean) l3 = <0xaa07de18> "false"
    (java.lang.Character) l4 = <0xaa07df68> "X"
    (java.lang.Long) l5 = <0xaa07e078> "-9223372036854775808"
    (java.lang.Object) l6 = <0xaa07e1a8> "Thread[Thread-0,5,main]"
    (java.lang.Object[]) l7 = <0xaa07e298> "[Ljava.lang.Thread;@26e431"
at Example2.main(Example2.java:4)
  locals:
...
```

minimalオプション
指定時には
出力されなかった
文字列。

「図 7-4」に比べて，次の違いがあります。

- minimal を指定した場合に適用対象外のクラスだった java.lang.Object クラスおよび java.lang.Object クラスの配列のローカル変数（l6 および l7）に対しても，文字列が出力されます。

ただし，次の場合は，文字列は出力されないで，アドレスだけが出力されます。

- ローカル変数の値が null だった場合
- 文字列で出力するときに再度例外が発生して，正常に値が得られなかった場合

> **重要**
>
> ローカル変数情報で出力するオブジェクトは，ほかのスレッドで並行して操作していることがあります。このため，このオプションを指定して出力した文字列は，実際に例外が発生したときのオブジェクトに対応する情報とは異なっているおそれがあります。

(3) クラスまたは配列型の変数の実際の型名を出力する場合の出力例

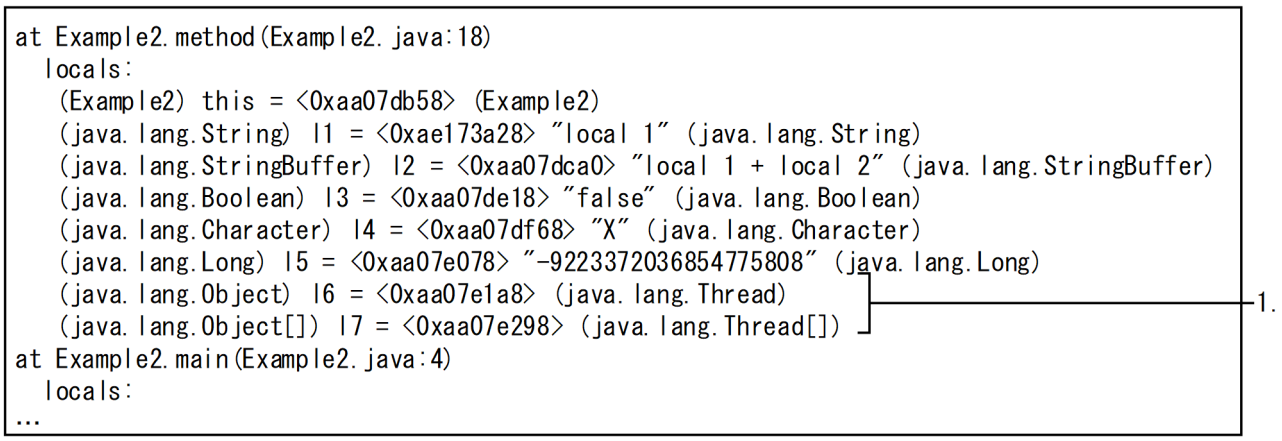
出力するローカル変数がクラスまたは配列型の場合，変数の型名と実際に代入されている値の型名が異なる場合があります。例えば，クラスの継承関係やインタフェースの実装関係によっては，異なる型名の値が変数に代入されます。

この場合に，-XX:+HitachiTrueTypeInLocals オプションの指定によって，クラスまたは配列型の変数に実際に代入されている値の型名を追記して取得できます。

取得した文字列は，値表現の末尾に半角スペース 1 文字分が追記され，( )で囲まれて出力されます。このとき，出力文字列長に制限はありません。

次に，-XX:+HitachiTrueTypeInLocals オプションが指定されている場合の出力例を示します。ここでは簡易出力フォーマットで示します。実行するプログラムは，Java プログラムの例 2 です。また，この例は，-XX:HitachiCallToString=minimal オプションが指定されている場合の例です。

図 7-6 -XX:+HitachiTrueTypeInLocals オプションを指定した場合の出力例（簡易出力フォーマット）



出力内容について説明します。

1. 変数の型名は java.lang.Object クラスまたは java.lang.Object クラスの配列ですが，実際に代入されている値の型名は java.lang.Thread クラスおよび java.lang.Thread クラスの配列であることが出力されています。

なお，-XX:+HitachiTrueTypeInLocals オプションは，-XX:HitachiCallToString=full オプションとも同時に指定できます。

7.5.2 -XX:+HitachiLocalsInStackTrace オプションが指定されている場合

スレッドダンプ内のスタックトレース情報の 1 スタックフレーム情報ごとに，そのスタックフレームに対応するメソッド内のローカル変数情報が挿入されて，出力されます。

出力形式および出力内容は，-XX:+HitachiLocalsInThrowable が指定されている場合に標準出力に出力される内容と同じです。

ただし，-XX:+HitachiLocalsInStackTrace オプションでは，次のオプションの指定は無効になります。

- -XX:HitachiCallToString オプション

Java プログラムの例と，それに対するスタックトレース内のローカル変数情報の出力例を示します。

Java プログラムの例 3

```
class Example3 {
    public static void main(String[] args) {
        Example3 e3 = new Example3();
        e3.method();
    }

    synchronized void method() {
        int l1 = 1;
        float l2 = 2.0f;
        String l3 = "local 3";
        Character l4 = new Character('X');
        Object l5 = new Thread();
        Object[] l6 = new Thread[10];

        <ここでスレッドダンプ出力!>
    }
}
```

出力例を次に示します。この例は，次の場合の例です。

- -g オプションまたは-g:vars オプションを指定して作成された class ファイルである
- -XX:+HitachiLocalSimpleFormat オプションを指定している
- -XX:+HitachiTrueTypeInLocals オプションを指定している

図 7-7　Java プログラムの例 3 に対するローカル変数情報の出力例

```
...
"main" prio=1 tid=0xb6e88d20 nid=0xb7492080 runnable [bfffb000..bfffb474]
        at Example3.method(Example3.java:15)
        - locked <0xab040550> (a Example3)
          locals:
            (Example3) this = <0xab040550> (Example3)
            (int) l1 = 1
            (float) l2 = 2.000000
            (java.lang.String) l3 = <0xaf112cc0> (java.lang.String)
            (java.lang.Character) l4 = <0xab040698> (java.lang.Character)
            (java.lang.Object) l5 = <0xab0407c8> (java.lang.Thread)
            (java.lang.Object[]) l6 = <0xab0408b8> (java.lang.Thread[])
        at Example3.main(Example3.java:4)
          locals:
            (java.lang.String[]) args [arg1] = <0xab040540> (java.lang.String[])
            (Example3) e3 = <0xab040550> (Example3)
...
```

8 アプリケーション実装時の注意事項

この章では，日立 JavaVM を利用してアプリケーションを実装する際の JDK に関する注意事項について説明します。

8.1　アプリケーション実装時の注意事項

ここでは，アプリケーション実装時の JDK に関する注意事項について説明します。

- URLClassLoader でのクラスのローディングについて

  URLClassLoader 経由で jar ファイルからクラスをローディングする場合，ローディング処理中に JavaVM の内部で作成されるオブジェクトが，JavaVM 終了時まで削除されないことがあります。適宜，Java アプリケーションを再起動してください。

- メソッド長の上限について

  JavaVM の仕様によって，1 メソッドのバイトコードは，64 キロバイト以内にする必要があります。

  64 キロバイトを超えると，クラスファイル生成時にエラーとなるか，またはクラスのロード時に java.lang.LinkageError 例外が発生します。

  また，64 キロバイト以内であっても，非常に複雑で行数が多い場合は，次のような弊害が発生することがあります。

  - GC 処理の実行に非常に時間が掛かる。
  - JIT コンパイルに非常に時間が掛かる。
  - JIT コンパイルに非常に多くのメモリを消費する。

  さらに，ローカル変数情報出力機能が有効な場合は，次の弊害も発生することがあります。

  - 拡張スレッドダンプの出力に時間が掛かる。
  - スレッドスタックトレースの取得に時間が掛かる。
  - 例外発生時の例外オブジェクト生成処理に時間が掛かる。

  そのため，Java ソース上の 1 メソッドの行数は，コメントや空行を除いて，およそ 500 行以内にすることをお勧めします。

- ファイルディスクリプタのクローズについて

  java.lang.Runtime.exec()，および java.lang.ProcessBuilder.start()で起動した子プロセスは，Process.getInputStream，getErrorStream，または getOutputStream のそれぞれのメソッドで取り出したストリームを通じてプロセス間通信をします。

  親プロセスでは，これらのメソッドを使わない場合でも 3 つのファイルディスクリプタを消費することに注意してください。

  ファイルディスクリプタをクローズするのは次の場合です。

  - Process オブジェクトのファイナライズが完了した場合
  - Process.destroy()を呼び出した場合
  - これらストリームに対して明示的に close()メソッドを呼び出した場合

- システムプロパティ java.library.path について

  システムプロパティ java.library.path には，ユーザのネイティブライブラリの検索パスを指定します。デフォルト値を次に示します。

JDK 下のネイティブライブラリのディレクトリ，環境変数 LD_LIBRARY_PATH の設定値，"/usr/lib64:/lib64:/lib:/usr/lib"

- java.net.Socket.connect()のタイムアウトについて

  java.net.Socket.connect()でのソケットの接続が，OS に設定されている TCP 通信のタイムアウト値によってタイムアウトすることがあります。TCP 通信のタイムアウト値になると，java.net.Socket.connect()に指定したタイムアウト値よりも前でも，タイムアウト値を指定していないときでも，接続がタイムアウトします。

  TCP 通信のタイムアウト値でタイムアウトした場合，次の詳細メッセージを含む java.net.ConnectException 例外がスローされます。

  メッセージの内容

  ```
  Connection timed out: connect [errno=10060, syscall=select]
  ```

  TCP 通信のタイムアウトについては，OS のドキュメントを参照してください。

- クラスロードのタイミングについて

  Java でのクラスファイルのメモリへの読み込み（クラスロード処理）は，プログラムの実行中に，そのクラスが初めて必要となったタイミングで実行されます。そのため，ある処理の初回実行でクラスロードの回数が多くなると，2 回目以降と比較して，処理時間が長くなることがあります。この場合，処理の実行に必要なクラスを事前にロードすることで，処理時間が改善します。

- 異なる OS 間で発生する文字化けについて

  文字コードの Unicode へのマッピングは，OS によって異なることがあります。このため，次の場合に文字化けが発生することがあります。文字化けが発生する代表的な文字には，—，～，‖，−，¢，£，¬があります。

  - UTF-8 などの Unicode のエンコーディングを使用して，異なる OS 間で上記文字データを受け渡す場合
  - 上記文字を含む文字列リテラルがあるソースプログラムから作成したクラスファイルを，異なる OS 上でリコンパイルしないでそのまま実行する場合

- Java API で使用するファイルについて

  Java API で使用するポリシーファイルや，ログイン構成ファイルなどのコンフィグレーションファイルは，UTF-8 エンコーディング方式でエンコーディングする必要があります。

- JNI 関数で文字列操作をする場合の UTF-8 エンコーディングについて

  JNI の場合，次の文字列操作をする関数で使用するエンコーディングは，標準 UTF-8 ではなく，Modified UTF-8 です。

  - NewStringUTF
  - GetStringUTFLength
  - GetStringUTFChars
  - ReleaseStringUTFChars
  - GetStringUTFRegion

- if 文の判定でジャンプできる長さの上限について

  JavaVM の仕様によって，if 文の判定でジャンプできる長さは 32 キロバイトまでです。ジャンプ先が 32 キロバイトを超える場合は，java.lang.LinkageError となります。

- java.nio.channels.FileChannel クラスの map，transferFrom，および transferTo メソッドについて

  map メソッドでは，2 ギガバイトを超えるファイルはサポートしていません。

  また，transferFrom および transferTo メソッドでの count 指定では，2 ギガバイト以上の値はサポートしていません。

- スレッドのスタックサイズについて

  JNI 関数 AttachCurrentThread()などを使用して，ネイティブスレッドを JavaVM に接続する場合は，-Xss オプションの指定値よりも大きなスタックサイズのスレッドで接続してください。

- Java プロセスが SSL/TLS 通信のクライアントとして動作する場合の注意点について

  SSL/TLS を使った通信相手のプロトコルを変更した場合，通信に失敗するおそれがあります。

  通信相手（サーバ）側で使用できる SSL/TLS プロトコルバージョンを変更したあと，クライアント側の Java プロセスを再起動しないで SSL/TLS を使った通信をしようとした場合，クライアント側の Java プロセスで例外が発生して通信に失敗するおそれがあります。

  これは，既知の接続先に対して以前接続に成功したプロトコルの情報がクライアント側に残っていると，そのプロトコルバージョンで SSL/TLS 通信を試みる，という挙動によるものです。

  通信相手が使用できる SSL/TLS プロトコルバージョンを変更する場合は，クライアント側の Java プロセスを再起動してください。

  また，通信相手（サーバ）側で使用できる SSL/TLS プロトコルバージョンを変更したあと，クライアント側の Java プロセスを再起動しないで通信が失敗した場合は，対象の Java プロセスを再起動してください。

- fork システムコールについて

  JNI や JVMTI で呼び出されるネイティブメソッド，またはネイティブコードで，fork()システムコール発行だけで現行プロセスのコピーの子プロセスを生成，または実行した場合，その親子プロセスの動作は保証できません。fork()システムコール発行による子プロセス生成後は，必ず exec()システムコールで新規プログラムをローディングしてから起動してください。また，Java 環境下で子プロセスを生成する場合には，Java クラスライブラリの java.lang.Runtime.exec()メソッドを使用することをお勧めします。

- シグナルについて

  JNI や JVMTI で呼び出されるネイティブメソッド，またはネイティブコードで，次のシグナルに対してシグナルハンドラを登録した場合，動作は保証しません。

  SIGHUP，SIGINT，SIGQUIT，SIGILL，SIGFPE，SIGBUS，SIGSEGV，SIGPIPE，SIGTERM，SIGUSR2，SIGCHLD，SIGXCPU，SIGXFSZ，（_SIGRTMAX-2）番のシグナル

- システムライブラリ関数やシステムコール呼び出し中のシグナル受信について

  システムライブラリ関数やシステムコール呼び出し中に，次のシグナルを受信することがあります。

  SIGHUP，SIGINT，SIGQUIT，SIGILL，SIGFPE，SIGBUS，SIGSEGV，SIGPIPE，SIGTERM，SIGUSR2，SIGCHLD，SIGXCPU，SIGXFSZ，（_SIGRTMAX-2）番のシグナル

システムライブラリ関数やシステムコールを呼び出す場合，該当する関数の処理が，これらのシグナル受信によって中断されて，エラーリターン（errno 値に EINTR が設定されるなど）することがあります。この場合は，適切な処置（例えば，再実行など）を実施してください。

- JNI プログラム中で使用できないライブラリについて

  JavaVM 内で管理している情報が更新されて不正な情報となる場合があるため，JNI プログラム中で次のライブラリは使用できません。

  - setjmp()
  - longjmp()
  - _setjmp()
  - _longjmp()
  - sigsetjmp()
  - siglongjmp()

- Windows との文字データの受け渡しについて

  Windows との文字データの受け渡しをする場合には，エンコーディングに MS932，windows-31j，cswindows31j のどれかを指定してください。

  Shift_JIS や SJIS を指定すると，次の文字が不正な文字コードに変換されます。

  ```
  ・NEC拡張文字　①②...⑳，ⅠⅡ...Ⅹ，㈱，㍾㍽㍼，㍉㌔㌢㍍など
  ・NEC選定IBM拡張文字　ⅰⅱ...ⅹなど
  ・IBM拡張文字　ⅠⅡ...Ⅹ，ⅰⅱ...ⅹ，№，℡など
  ```

- 生成できるスレッド数の上限値について

  システム全体で生成できるスレッド数の上限は，/proc/sys/kernel/threads-max です。また，1 ユーザ当たりの生成できるスレッド数の上限は，/etc/security/limits.conf の nproc の値（ulimit -u と同じ）となります。システムに応じて，これらの値を調整してください。

- 通知待ちのモニタ数について

  JVMTI インタフェースを使用して取得した，通知待ちのモニタ情報が不正となる場合があります。

- 半角カタカナについて

  JIS X 201 で制定されている文字のうち，半角カタカナは JavaSE の API を使用した GUI の実装では利用できません。

- java.security.SecureRandom クラスについて

  /dev/random ファイルがあるプラットフォームでは，SecureRandom クラスの一部の API で/dev/random ファイルから OS が生成した乱数を取得します。/dev/random ファイルから乱数を取得するのは，次の条件がすべて重なる場合です。

  - java.security.egd プロパティまたは java.security ファイル中の securerandom.source に file:/dev/random/を指定する
  - SecureRandom クラスの generateSeed()メソッドまたは getSeed()メソッドを呼び出す

OS の乱数生成速度には限度があります。このため，短い間隔で乱数の取得処理を実行すると，OS による乱数生成完了まで処理が完了しないので注意してください。

- java.io.tmpdir プロパティについて

  java.io.tmpdir プロパティには，書き込み権限があり，かつ存在するディレクトリを指定してください。java.io.tmpdir プロパティの初期値は/tmp です。

  また，Java RMI の動的クラスローディング機能や Java API の java.io.File.createTempFile()では，java.io.tmpdir プロパティで指定されたディレクトリに一時ファイルを作成します。これらの機能を正常に動作させるため，JavaVM プロセス起動中は一時ファイルの作成先を削除しないでください。

- デフォルトエンコーディングについて

  アプリケーションサーバのデフォルトエンコーディングを次に示します。

  LANG が C または POSIX のとき：US-ASCII（別名 ASCII）

  LANG が jp_JP，jp_JP.eucJP，ja_JP.ujis，japanese，japanese.euc のどれかのとき：x-euc-jp-linux（別名 EUC_JP_LINUX）

  LANG が ja_JP.utf8 のとき：UTF-8（別名 UTF8）

付録

付録 A　このマニュアルの参考情報

このマニュアルを読むに当たっての参考情報を示します。

付録 A.1　関連マニュアル

関連マニュアルを次に示します。必要に応じてお読みください。

表記	マニュアル名	資料番号
uCosminexus Application Runtime ユーザーズガイド	uCosminexus Application Runtime for Apache Tomcat ユーザーズガイド	3021-3-K01

付録 A.2　このマニュアルでの表記

サービス名，製品名，機能名などの名称を，次のように表記しています。

表記		サービス名，製品名，機能名など
GC		ガーベージコレクションの略。プログラムが動作中に確保したメモリ領域のうち，不要になった領域を自動的に解放する機能です。
アプリケーションサーバ		アプリケーションの実行環境です。
Linux	Red Hat Enterprise Linux 7（AMD/Intel 64）	Red Hat Enterprise Linux 7.1 （AMD/Intel 64）以降
	Red Hat Enterprise Linux 8（AMD/Intel 64）	Red Hat Enterprise Linux 8.1 （AMD/Intel 64）以降
Tomcat		Apache Tomcat
uCosminexus Application Runtime		uCosminexus Application Runtime with Java for Apache Tomcat
uCosminexus Application Runtime サポートサービス		OSS Support Service with uCosminexus Application Runtime with Java for Apache Tomcat
Windows		Microsoft Windows
クラス別統計		日立クラス別統計
日立 JavaVM		Cosminexus Developer's Kit for Java
保守員		uCosminexus Application Runtime サポートサービスの契約に基づくお問い合わせ窓口のことです。

付録 A.3　英略語

このマニュアルで使用している英略語を次に示します。

英略語	英字での表記
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
AWT	Abstract Window Toolkit
CPU	Central Processing Unit
CSV	Comma Separated Value
DLL	Dynamic Link Library
EUC	Extended UNIX Code
GC	Garbage Collection
CM	Concurrent Marking
GUI	Graphical User Interface
I/O	Input/Output
ID	Identifier
IP	Internet Protocol
JAR	Java Archive
Java SE	Java Platform, Standard Edition
JavaVM または JVM	Java Virtual Machine
JDBC	Java Database Connectivity
	JDBC
JDK	Java Development Kit
	JDK
JIS	Japanese Industrial Standards
JMS	Java Message Service
JNI	Java Native Interface
JSP	JavaServer Pages
	JSP
JST	Japan Standard Time
JVMTI	Java Virtual Machine Tool Interface
ORB	Object Request Broker

英略語	英字での表記
OS	Operating System
RMI	Remote Method Invocation
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UTF-8	8-bit UCS Transformation Format
VM	Virtual Machine

付録 A.4　KB（キロバイト）などの単位表記について

1KB（キロバイト），1MB（メガバイト），1GB（ギガバイト），1TB（テラバイト）はそれぞれ 1,024 バイト，$1{,}024^2$ バイト，$1{,}024^3$ バイト，$1{,}024^4$ バイトです。

索引

記号

-XX:[+¦-]HitachiCommaVerboseGC 210
-XX:[+¦-]HitachiFullCore 255
-XX:[+¦-]HitachiJavaClassLibTrace 244
-XX:[+¦-]HitachiJavaLogNoMoreOutput 197
-XX:[+¦-]HitachiLocalsInStackTrace 250
-XX:[+¦-]HitachiLocalsInThrowable 246
-XX:[+¦-]HitachiLocalsSimpleFormat 251
-XX:[+¦-]HitachiOutOfMemoryAbort 237
-XX:[+¦-]HitachiOutOfMemoryAbortThreadDump 238
-XX:[+¦-]HitachiOutOfMemoryAbortThreadDumpWithJHeapProf 239
-XX:[+¦-]HitachiOutOfMemoryCause 233
-XX:[+¦-]HitachiOutOfMemoryHandling 239
-XX:[+¦-]HitachiOutOfMemorySize 236
-XX:[+¦-]HitachiOutOfMemoryStackTrace 234
-XX:[+¦-]HitachiOutputMilliTime 201
-XX:[+¦-]HitachiThreadDump 184
-XX:[+¦-]HitachiThreadDumpToStdout 193
-XX:[+¦-]HitachiThreadDumpWithBlockCount 195
-XX:[+¦-]HitachiThreadDumpWithCpuTime 194
-XX:[+¦-]HitachiThreadDumpWithHashCode 194
-XX:[+¦-]HitachiTrueTypeInLocals 252
-XX:[+¦-]HitachiVerboseGC 202
-XX:[+¦-]HitachiVerboseGCCpuTime 221
-XX:[+¦-]HitachiVerboseGCPrintCause 219
-XX:[+¦-]HitachiVerboseGCPrintDate 220
-XX:[+¦-]HitachiVerboseGCPrintDeleteOnExit 226
-XX:[+¦-]HitachiVerboseGCPrintJVMInternalMemory 223
-XX:[+¦-]HitachiVerboseGCPrintTenuringDistribution 221
-XX:[+¦-]HitachiVerboseGCPrintThreadCount 224
-XX:[+¦-]JavaLogAsynchronous 199
-XX:[+¦-]JITCompilerContinuation 258
-XX:[+¦-]PrintCodeCacheFullMessage 231
-XX:[+¦-]PrintCodeCacheInfo 228
-XX:[+¦-]StandardLogToHitachiJavaLog 199
-XX:[+¦-]UseCompressedOops 259
-XX:+Hitachi 184
-XX:+HitachiJavaClassLibTrace 302
-XX:+HitachiLocalsInStackTrace オプションが指定されている場合 329
-XX:+HitachiLocalsInThrowable オプションが指定されている場合 323
-XX:+HitachiOutOfMemoryAbort (OutOfMemory 発生時強制終了機能) 29
-XX:+HitachiOutOfMemoryStackTrace 302
-XX:+HitachiVerboseGC 302
-XX:+JITCompilerContinuation 302
-XX:CodeCacheInfoPrintRatio 230
-XX:HitachiCallToString 253
-XX:HitachiJavaClassLibTraceLineSize 246
-XX:HitachiJavaLog 195
-XX:HitachiJavaLogFileSize 196
-XX:HitachiJavaLogNumberOfFile 198
-XX:HitachiJITCompileMaxMemorySize 256
-XX:HitachiOutOfMemoryHandlingMaxThrowCount 243
-XX:HitachiOutOfMemoryStackTraceLineSize 235
-XX:HitachiThreadLimit 260
-XX:HitachiVerboseGCIntervalTime 218

C

car_tar_gz 277
car_tar_Z 279
Card table マップアドレス 314
CodeCache 情報 314
Concurrent Marking（CM） 65
CopyGC 22
core アーカイブ機能 277, 279
core ダンプ／スタックトレースファイル 89
core ダンプ／スタックトレースファイル
取得方法 90
core ダンプの生成を示すメッセージの出力内容 320
core ファイルとスレッドダンプの取得 281
C ヒープが不足した場合 318
C ヒープが不足した場合のメッセージログの出力項目 319
C ヒープ不足を示すメッセージログの出力内容 318
C ヒープ領域 28

E

Eden 領域 27, 57
Event 情報 315

F

Free 領域 57
FullGC 22, 52, 72

G

G1GC 18
G1GC の仕組み 51
G1GC のチューニング 73
GC の強制発生 284
GC の選択の指針 144

H

Humongous 領域 57

I

Internal Error が発生した場合 320

J

javacore 281
JAVACOREDIR 270
javagc 284
javatrace 287
javatrace 起動コマンドのコマンドライン 318
JavaVM 固有領域 27
JavaVM スタックトレース情報 322
JavaVM のスレッドダンプファイル 89
取得方法 92
JavaVM ログ（JavaVM ログファイル） 302
JavaVM ログファイル 89
取得方法 93
JavaVM ログファイルを出力するオプション 302
Java オブジェクトの寿命 21, 51
Java ヒープ 27
Java ヒープの使用状況 313
Java ヒープのチューニング 31
jheapprof 290
jheapprofanalyzer 295
JIT コンパイル 164
JP.co.Hitachi.soft.jvm.autofinalizer 262

L

Large pages 確保失敗情報 314

M

Metaspace 領域 28, 58
MixedGC 52, 69

N

New 領域 57

O

OS 稼働情報
取得方法 94
OS 固有領域 27
OutOfMemory（アプリケーションサーバ） 114
OutOfMemory（メモリ不足） 88

OutOfMemoryError 障害
各使用量の分析　110
スレッドダンプによるローカル変数情報の解析例　111
設定変更による対処　112
調査の流れ　106
メッセージによる症状の切り分け　108
OutOfMemoryError 障害（JavaVM）　106

P

Polling page アドレス　314

S

SerialGC　18
SerialGC の仕組み　21
siginfo 情報　312
STATIC メンバ統計機能　129
STATIC メンバ統計機能で出力するクラス別統計情報　130
Survivor 領域　27, 57
Survivor 領域の年齢分布情報出力機能　161
Survivor 領域の年齢分布情報出力機能使用時の注意事項　163
Survivor 領域の年齢分布情報出力機能の概要　161
Survivor 領域の年齢分布情報の出力形式と出力例 161

T

Tenured 増加要因の基点オブジェクトリスト出力機能　154
Tenured 領域　27, 57
Tenured 領域内不要オブジェクト統計機能　145
Tenured 領域内不要オブジェクト統計機能で出力するクラス別統計情報　148
Tenured 領域内不要オブジェクト統計機能の概要 145
Tenured 領域内不要オブジェクト統計機能の実行に関する注意事項　149

V

VM の状態　313

Y

YoungGC　52, 63

あ

圧縮オブジェクトポインタ機能　171

い

異常終了位置とシグナル種別　309
インスタンス統計機能　123
インスタンス統計機能で出力するクラス別統計情報　125

え

エラーリポートファイル　89
取得方法　89

お

オブジェクトの年齢　23, 53

か

拡張 verbosegc 情報　35
拡張 verbosegc 情報の取得　303
拡張 verbosegc 情報の取得を指定するオプション303
カレントスレッド情報　311
環境変数　316

く

クラス別統計機能　119
クラス別統計機能を前提とする機能　120
クラス別統計情報　119
クラス別統計情報解析機能　157
クラス別統計情報解析ファイルの CSV 出力　295
クラス別統計情報付き拡張スレッドダンプの出力 290
クラス別統計情報の出力　120
クラスまたは配列型の変数の実際の型名を出力する場合の出力例　328
クラスまたは配列型の変数を文字列として出力する場合の出力例　326

こ

コードキャッシュ領域　28
コードキャッシュ領域に関するログの内容　306
コマンドおよび VM パラメタ　316

さ

参照関係情報出力機能　133
参照関係情報出力機能で出力するクラス別統計情報　135

し

時間情報　310
シグナル情報　311
シグナル情報の格納先アドレス　311
システム名，CPU，実メモリ，および VM 情報　317

す

スタックトレース　311
スタックトレースにローカル変数情報を出力するためのオプション　322
スタックの先頭から格納されている情報　312
スタック領域　28
スレッド作成に失敗した場合　321
スレッド情報　312
スレッドダンプ情報の構成　299
スローダウン　88
スローダウン（アプリケーションサーバ）　114

せ

世代別 GC　21, 51

と

統計前の GC 選択機能　143
統計前の GC 選択機能の概要　143
登録済みシグナルハンドラ　317
トラブルシューティング
　JavaVM　88
　アプリケーションサーバ　114
トラブルシューティングに使用する資料の詳細　298
トランザクションタイムアウト（アプリケーションサーバ）　114
トレース情報の収集　287

ひ

日立 JavaVM が出力するメッセージログ　308
日立 JavaVM が出力するメッセージログ（標準出力およびエラーリポートファイル）　308
日立 JavaVM で使用するコマンド　271
日立 JavaVM で使用するコマンドの一覧　276
日立 JavaVM で使用するコマンドの詳細　277
日立 JavaVM の GC ログ　301
日立 JavaVM の機能詳細　116
日立 JavaVM の機能の概要　117
日立 JavaVM のスレッドダンプ　299
標準出力と標準エラー出力　89
標準の形式および簡易出力フォーマットでの出力例　323

ふ

ファイナライズ滞留解消機能　166
プロセススローダウン
　調査の流れ　104
プロセススローダウン（JavaVM）　104
プロセスダウン　88
　エラーリポートファイルの解析　95
　調査の流れ　94
プロセスダウン（JavaVM）　94
プロセスダウン（アプリケーションサーバ）　114
プロセスハングアップ（アプリケーションサーバ）　114
プロセスハングアップ（無応答）　88
　スレッドダンプ（デッドロック）の解析例　101
　スレッドダンプ（プロセス無応答）の解析例　99
　スレッドダンプ（無限ループ）の解析例　101
　スレッドダンプ（無限ループ）のローカル変数情報による解析例　103
　調査の流れ　98
プロセスハングアップ（無応答）（JavaVM）　97

ほ

保守員調査依頼時の提供情報　113

ま

マシン情報　317

め

メモリ情報　313

メモリ不足を示すメッセージの出力内容　319

ら

ライブラリ　316

り

リクエストタイムアウト（アプリケーションサーバ）　114

れ

レジスタ情報　312

ろ

ローカル変数　322

ローカル変数情報　322

ログファイルの非同期出力機能　169

株式会社 日立製作所

〒 100-8280　東京都千代田区丸の内一丁目 6 番 6 号