IBM WebSphere Real Time for RT Linux
バージョン 3

ユーザー・ガイド

IBM

IBM WebSphere Real Time for RT Linux

バージョン 3

ユーザー・ガイド

IBM

お願い

本書および本書で紹介する製品をご使用になる前に、 163 ページの『特記事項』に記載されている情報をお読みください。

本書は、IBM WebSphere Real Time for RT Linux バージョン 3、および新しい版で明記されていない限り、以降のすべてのリリースおよびモディフィケーションに適用されます。

© Copyright IBM Corporation 2003, 2011.

目次

図 v

表 vii

前書き ix

第 1 章 概要 1
WebSphere Real Time for RT Linux の概要 1
新機能 3
利点 3

第 2 章 IBM WebSphere Real Time for RT Linux の理解 5
Metronome ガーベッジ・コレクターの概要 5
コンパイラー 7
JIT と AOT のコンパイルの比較 8
RTSJ のサポート 9
リアルタイム・スレッドのスケジューリングおよびディスパッチング 9
メモリー管理 14
同期およびリソース共有 18
周期的パラメーターと非周期的パラメーター . . 19
非同期イベント処理 19
必要な文書 20

第 3 章 計画 25
マイグレーション 25
ハードウェアおよびソフトウェアの前提条件 . . . 25
考慮事項 26

第 4 章 WebSphere Real Time for RT Linux のインストール 29
インストール・ファイル 29
Real Time Linux 環境のインストール 29
InstallAnywhere パッケージからのインストール . . 30
手動インストールの実行 31
自動インストールの実行 32
中断されたインストール 34
既知の問題と制約 34
パスの設定 35
クラスパスの設定 36
インストール済み環境のテスト 37
WebSphere Real Time for RT Linux のアンインストール 38

第 5 章 IBM WebSphere Real Time for RT Linux アプリケーションの実行 . 39
スレッドのスケジューリングとディスパッチング . . 39
リアルタイム Java スレッドの優先順位およびポリシー 40
WebSphere Real Time for RT Linux でのコンパイル済みコードの使用 40
AOT コンパイラーの使用. 43
Just-In-Time (JIT) コンパイラー. 58
非ヒープ・リアルタイム・スレッドの使用 61
メモリーおよびスケジューリングの制約 62
クラス・ロードの制約 63
NHRT とともに実行する場合の Java スレッドに関する制約 63
同期 64
非ヒープ・リアルタイム・クラスの安全性 . . . 65
JVM 間でのクラス・データの共用. 70
共有クラス・キャッシュを使用したアプリケーションの実行 71
Metronome ガーベッジ・コレクターの使用 73
プロセッサー使用率の制御 73
Metronome ガーベッジ・コレクターの調整 . . . 73
Metronome ガーベッジ・コレクターの制限 . . . 74

第 6 章 アプリケーションの開発 75
リアルタイムを活用する Java アプリケーションの作成 75
リアルタイム・アプリケーションの作成の概要 . 75
WebSphere Real Time for RT Linux アプリケーションの計画 76
Java アプリケーションの変更 78
リアルタイム・スレッドの作成 79
非同期イベント・ハンドラーの作成 81
NHRT スレッドの作成. 83
RTSJ でのメモリー割り振り 84
高解像度タイマーの使用 86
サンプル・アプリケーション 87
サンプル・アプリケーションのビルド 90
サンプル・アプリケーションの実行 90
サンプル・リアルタイム・ハッシュ・マップ . . . 95
Eclipse を使用した WebSphere Real Time for RT Linux アプリケーションの開発 96
アプリケーションのデバッグ 98
JVM を使用した Eclipse の実行 99

第 7 章 パフォーマンス 101
JVM 間でのクラス・データの共有 (非リアルタイム・モード) 101

第 8 章 セキュリティー 103
共有クラス・キャッシュのセキュリティーの考慮事項 103

第 9 章 トラブルシューティングおよびサポート. 105
一般的な問題判別方法 105

© Copyright IBM Corp. 2003, 2011

Linux の問題判別 105
NLS の問題判別 110
ORB の問題判別 110
OutOfMemory エラーのトラブルシューティング 111
OutOfMemoryError の診断 112
複数のヒープでの問題診断 118
メモリー・リークの回避. 119
メモリー・コンテキスト全体でのリフレクションの使用. 120
スコープ・メモリー域での内部クラスの使用 . . 121
診断ツールの使用 122
ダンプ・エージェントの使用 122
Javadump の使用 126
Heapdump の使用 132
システム・ダンプおよびダンプ・ビューアーの使用 135
Java アプリケーションと JVM のトレース . . 136
JIT および AOT の問題判別 137
Diagnostics Collector 145
ガーベッジ・コレクターの診断 145
共有クラス診断. 152
JVMTI の使用 153
Diagnostic Tool Framework for Java の使用 . . 153
IBM Monitoring and Diagnostic Tools for Java - Health Center の使用 153

第 10 章 参照 155
コマンド行オプション 155
Java オプションとシステム・プロパティーの指定 155
システム・プロパティー. 155
標準オプション. 156
非標準オプション 157
JVM のデフォルト設定 160
WebSphere Real Time for RT Linux クラス・ライブラリー 162
TCK を使用した実行 162

特記事項. 163
商標 164

索引 165

1. WebSphere Real Time for RT Linux の概要 2
2. JIT コンパイラーと AOT コンパイラーの比較。 9
3. ヒープ・オブジェクト参照にアクセスする NHRT の例 65
4. ヒープ・オブジェクト参照にアクセスする NHRT の例 (図 1 の続き) 66
5. RTSJ のフィーチャーと予測可能性の向上との比較 76
6. 月着陸船のダイアグラム 89

© Copyright IBM Corp. 2003, 2011

表

1. リアルタイム・モードで使用される Java コマンド 2
2. ガーベッジ・コレクションとその優先順位の例 6
3. リアルタイム・スレッドおよび非ヒープ・リアルタイム・スレッドによるメモリー・アクセス. 17
4. <*signature*> オプションの例 47
5. java.lang パッケージ内の NHRT セーフではないクラス 69
6. java.lang.reflect パッケージ内の NHRT セーフではないクラス 69
7. java.net パッケージ内の NHRT セーフではないクラス 70
8. java.io パッケージ内の NHRT セーフではないクラス 70
9. java.math パッケージ内の NHRT セーフではないクラス 70
10. リアルタイム・モードでアプリケーションを実行する場合に使用可能なサブオプション . . . 71
11. サンプル・アプリケーションにおける各メモリー域に対する各スレッドの関係 80
12. IBM WebSphere Real Time for RT Linux のスレッド名 130

© Copyright IBM Corp. 2003, 2011

前書き

このユーザーズ・ガイドには、IBM® WebSphere® Real Time for RT Linux に関する一般情報が記載されています。

第 1 章 概要

この情報は IBM WebSphere Real Time for RT Linux に関する説明です。

- 『WebSphere Real Time for RT Linux の概要』
- 3 ページの『新機能』
- 3 ページの『利点』

WebSphere Real Time for RT Linux の概要

WebSphere Real Time for RT Linux は、IBM J9 仮想マシン (JVM) にリアルタイム機能を組み込んだものです。

WebSphere Real Time for RT Linux は、リアルタイム機能によって IBM SDK for Java を拡張する、Software Development Kit を含む Java ランタイム環境です。正確な応答時間に依存するアプリケーションでは、標準的な Java テクノロジーで、WebSphere Real Time for RT Linux に備えられたリアルタイム機能を活用できます。

特徴

リアルタイム・アプリケーションには、絶対的な速度よりも一貫性のある実行時間が必要です。

リアルタイム・モードで JVM を実行する場合、ガーベッジ・コレクションされたヒープのほかに追加メモリー域を使用できます。プログラムでは、任意の数の、再使用可能なスコープ・メモリー域と、再使用不可の永久メモリー域を要求または指定することができます。これらのメモリー域はガーベッジ・コレクションの対象となりません。この機能により、アプリケーションはメモリー使用量をより綿密に制御できます。 また、時間ベースのコレクションを実行するために Metronome ガーベッジ・コレクター も使用されます。 JVM を従来のスループット・モードで実行する場合、スループットを最適化するさまざまな作業ベースのガーベッジ・コレクターを使用することができます。ただし、Metronome ガーベッジ・コレクター を使用した場合よりも個々の遅延が長くなる可能性があります。

従来の JVM を使用してリアルタイム・アプリケーションをデプロイする際の主要な注意点は以下のとおりです。

- ガーベッジ・コレクション (GC) アクティビティーによる遅延が予測できません (長くなる可能性があります)。
- ジャストインタイム (JIT) コンパイルおよび再コンパイルが行われたときにメソッド・ランタイムが遅延し、実行時間が変動します。
- 任意のオペレーティング・システム・スケジューリング。

WebSphere Real Time for RT Linux は、以下の機能を提供してこれらの障害を除去します。

- Metronome ガーベッジ・コレクター (休止時間がきわめて短い、増分的な決定論的ガーベッジ・コレクター)。
- Ahead-of-Time (AOT) コンパイル。

© Copyright IBM Corp. 2003, 2011

• 優先順位ベースの FIFO スケジューリング。

さらに、WebSphere Real Time はリアルタイム・プログラマーのために RTSJ 機能を提供します。 9 ページの『RTSJ のサポート』を参照してください。

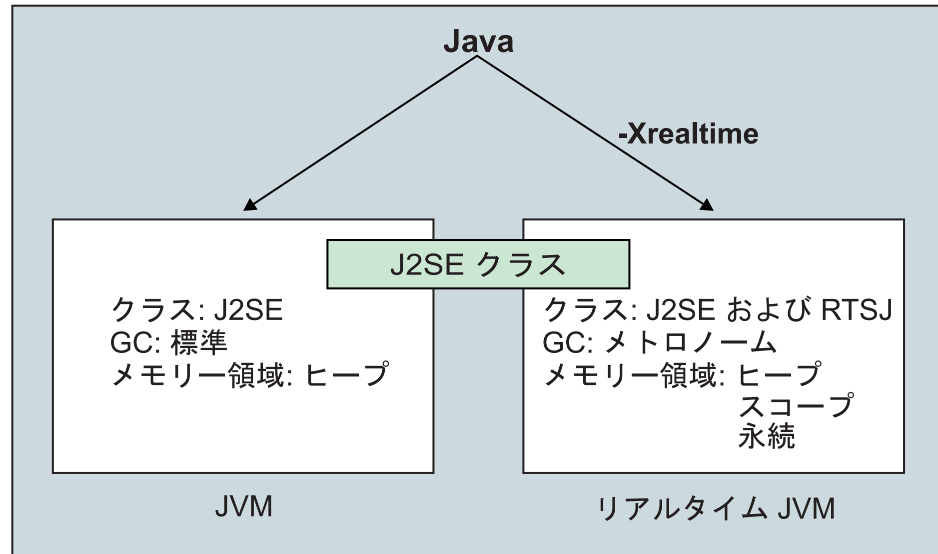


図 1. WebSphere Real Time for RT Linux の概要

リアルタイム機能を有効にするには、JVM または提供された何らかのツールを実行するときに -Xrealtime オプションを使用します。デフォルトでは、JVM および提供されたツールは、リアルタイム機能を有効にしないで実行されます。 図 1 は、WebSphere Real Time for RT Linuxとともに提供される 2 つの JVM の関係を表しています。

以下の Java コマンドは -Xrealtime オプションを認識します。

表 1. リアルタイム・モードで使用される Java コマンド

コマンド	機能
java	デフォルトでは標準モードで実行されますが、-Xrealtime オプションが指定されている場合にはリアルタイム・モードでも実行されます。リアルタイム・モードでは、プログラマーは javax.realtime パッケージ内のクラスにアクセスします。プリコンパイルされた jar ファイルおよび Metronome の決定論的ガーベッジ・コレクション・テクノロジーを使用することができます。
javac、 javah、 javap	デフォルトでは標準モードで実行されますが、-Xrealtime オプションが指定されている場合には、クラスパスに javax.realtime.* クラスを含めます。
admincache	-Xrealtime を指定して実行することも、指定しないで実行することもできますが、**admincache** ツールによる共有キャッシュへのデータの取り込みは、リアルタイム・モードでのみ行うことができます。通常のモードでは、キャッシュ・ユーティリティー (listAllCaches、printStats、など) のみを使用できます。 **jdmpview** と同様に、**admincache** は、リアルタイム JVM のキャッシュにアクセスするためには -Xrealtime を使用して実行する必要があり、通常の JVM のキャッシュにアクセスするためには -Xrealtime を指定しないで実行する必要があります。

表 1. リアルタイム・モードで使用される Java コマンド (続き)

コマンド	機能
jextract	jextract はデフォルトでは標準モードで実行されますが、リアルタイム・モードの JVM で生成されたシステム・ダンプを処理するときには -Xrealtime オプションを指定して実行する必要があります。

新機能

このトピックでは、IBM WebSphere Real Time for RT Linux の変更内容を紹介します。

WebSphere Real Time for RT Linux V3

WebSphere Real Time for RT Linux V3 は、リアルタイム機能を組み込むためにこのリリースから使用可能になったフィーチャーと機能に基づいて、IBM SDK for Java 7 を拡張したものです。以前のバージョンの WebSphere Real Time for RT Linux は、以前のリリースの IBM SDK for Java を基にしていました。

新機能について詳しくは、IBM SDK for Java 7 インフォメーション・センターの新機能を参照してください。

jxeinajar

WebSphere Real Time for RT Linux V3 では、jxeinajar の使用はサポートされなくなりました。jxeinajar に関する以前の情報、特に admincache へのマイグレーションに関する情報を参照する場合は、WebSphere Real Time for RT Linux V2 資料で参照できます。

利点

リアルタイム環境の利点は、Java アプリケーションが標準 JVM の場合よりはるかに予測可能性が高い形で実行され、Java アプリケーションのために一貫性のあるタイミング動作が提供されることです。コンパイルやガーベッジ・コレクションなどのバックグラウンド・アクティビティーは、指定された時刻に行われるため、アプリケーションの実行中にバックグラウンド・アクティビティーが予期せずピークに達することはなくなります。

上記の利点は、以下の機能を使用して JVM を拡張することにより、利用できます。

- Metronome リアルタイム・ガーベッジ・コレクション・テクノロジー
- Ahead-of-Time (AOT) コンパイル
- Real-Time Specification for Java (RTSJ) のサポート

すべての Java アプリケーションは、Metronome ガーベッジ・コレクターと、一定の間隔で行われる、その決定論的なガーベッジ・コレクションを利用することにより、修正せずにリアルタイム環境で実行できます。 WebSphere Real Time for RT Linux から最大の利益を得るために、リアルタイム・スレッド と非ヒープ・リアル

タイム・スレッド の両方を使用して、リアルタイム環境に特化したアプリケーションを作成できます。使用する手法は、アプリケーションのタイミング仕様によって異なります。

多くのリアルタイム Java アプリケーションは、Java ポータビリティーの利点を維持しつつ、目標を達成するために Metronome ガーベッジ・コレクター および AOT の短い休止時間を利用することができます。要件がより厳しいアプリケーションの場合、スコープ・メモリーと永久メモリーで リアルタイム・スレッド および 非ヒープ・リアルタイム・スレッド の RTSJ 機能を使用する必要があります。このようにすると、アプリケーションがリアルタイム環境でのみ実行されるようになり、JSE Java へのポータビリティーという利点が失われます。また、より複雑なプログラミング・モデルを開発することも必要になります。

第 2 章 IBM WebSphere Real Time for RT Linux の理解

このセクションでは、IBM WebSphere Real Time for RT Linux に関する重要なコンポーネントについて説明します。

- 『Metronome ガーベッジ・コレクターの概要』
- 7 ページの『コンパイラー』
  - 8 ページの『JIT と AOT のコンパイルの比較』
- 9 ページの『RTSJ のサポート』
  - 9 ページの『リアルタイム・スレッドのスケジューリングおよびディスパッチング』
  - 14 ページの『メモリー管理』

Metronome ガーベッジ・コレクターの概要

WebSphere Real Time for RT Linuxでは、標準のガーベッジ・コレクターの代わりに、Metronome ガーベッジ・コレクターを使用します。

Metronome ガーベッジ・コレクションと標準的なガーベッジ・コレクションの主な違いは、Metronome ガーベッジ・コレクションが細かく分けられた割り込み可能なステップで少しずつ実行されるのに対して、標準的なガーベッジ・コレクションでは、ガーベッジにマークを付けて収集する間はアプリケーションを停止することです。

例えば、次のようにします。

```
java -Xrealtime -Xgc:targetUtilization=80 yourApplication
```

この例では、60 ミリ秒ごとに 80% の時間がアプリケーションの実行に使用されるように指定しています。残りの 20% の時間は、収集すべきガーベッジが残っている場合、ガーベッジ・コレクションに使用される可能性があります。Metronome ガーベッジ・コレクターに十分なリソースが与えられている場合は、使用率レベルが保証されます。ガーベッジ・コレクションは、ヒープ内のフリー・スペース量が、動的に決定されたしきい値を下回ったときに開始されます。

ガーベッジ・コレクションおよび優先順位

ガーベッジ・コレクション・スレッドは、ヒープ内のガーベッジを生成する優先順位が最も高いスレッドよりも高い優先順位で実行する必要があります。そうしないと、構成された使用率で指定されているように実行されない場合があります。通常の Java スレッドとリアルタイム・スレッドの両方でガーベッジが生成される可能性があるため、ガーベッジ・コレクションは、すべての通常スレッドとリアルタイム・スレッドよりも高い優先順位で実行する必要があります。この優先順位付けは JVM で自動的に行われ、ガーベッジ・コレクションはすべての通常スレッドとリアルタイム・スレッドの最も高い優先順位より 0.5 上の優先順位で実行されます。ただし、非ヒープ・リアルタイム・スレッド (NHRT) がガーベッジ・コレクションに影響を受けないようにすることが重要です。NHRT はすべて、優先順位の最も高い

© Copyright IBM Corp. 2003, 2011

リアルタイム・スレッドよりも高い優先順位で実行してください。これは、NHRT がガーベッジ・コレクションよりも高い優先順位で実行され、遅延が発生しないことを意味します。

表 2 に、定義できる優先順位と、選択内容に応じた関連ガーベッジ・コレクションの優先順位の典型的な例を示します。

Java の優先順位と OS の優先順位との比較については、 12 ページの『優先順位のマッピングおよび継承』を参照してください。

表 2. ガーベッジ・コレクションとその優先順位の例

スレッド	優先順位 (例)
優先順位が最も高いリアルタイム・スレッド	20 (OS の優先順位は 43)
上記の場合のガーベッジ・コレクター	20.5 (OS の優先順位は 44)
ガーベッジ・コレクターとは別に NHRT を実行するには、GC よりも高い優先順位を設定します。	21 (OS の優先順位は 45) 以上。
Metronome アラーム・スレッド	優先順位は 46 (OS の優先順位は 89)

注: このような構成でも、非ヒープ・リアルタイム・スレッドがガーベッジ・コレクションにまったく影響を受けないわけではありません。これは、Metronome アラーム・スレッドが定期的にウェイクアップし、ガーベッジ・コレクションで何らかの動作が必要かどうかを判断できるように、システムで最も高い優先順位で実行されるためです。もちろん、これを行うための作業はささいなものであるため、重要な考慮事項ではありません。

Metronome ガーベッジ・コレクションとクラスのアンロード

Metronome ガーベッジ・コレクターは、IBM WebSphere Real Time ではクラスをアンロードしません。これは、休止時間が異常値になる原因となる非決定論的な量の作業が必要になる可能性があるためです。

Metronome ガーベッジ・コレクターのスレッド

Metronome ガーベッジ・コレクターは、単一のアラーム・スレッドおよび複数のコレクション (GC) スレッドという 2 つのタイプのスレッドで構成されています。デフォルトでは、GC スレッドは 1 つです。 JVM の GC スレッド数は、**-Xgcthreads** オプションを使用して設定することができます。

JVM のアラーム・スレッド数は変更できません。

Metronome ガーベッジ・コレクターは定期的に JVM を検査して、ヒープ・メモリーに十分なフリー・スペースがあるかどうかを確認します。フリー・スペース量が制限値を下回った場合、Metronome ガーベッジ・コレクターは JVM を起動してガーベッジ・コレクションを開始します。

アラーム・スレッド

単一のアラーム・スレッドでは、最小リソースの使用が保証されます。このスレッドは一定の間隔で「ウェイク」し、以下を確認します。

- ヒープ・メモリー内のフリー・スペース量
- ガーベッジ・コレクションが現在行われているかどうか

使用可能なフリー・スペースが不足しており、ガーベッジ・コレクションが行われていない場合は、アラーム・スレッドがコレクション・スレッドを起動してガーベッジ・コレクションを開始します。アラーム・スレッドは、次の JVM の検査のためにスケジュールされた時間になるまでは何も行いません。

コレクション・スレッド

各コレクション・スレッドで、ヒープ・オブジェクトの Java およびリアルタイム・スレッドが検査されます。メモリー域は以下の順序で検査されます。

1. スコープ・メモリー。スコープ・メモリーのオブジェクトによって使用されている、ヒープ内のすべてのライブ・オブジェクトを特定してマークを付けるためのメモリーです。
2. 永久メモリー。永久メモリーのオブジェクトによって使用されている、ヒープ内のすべてのライブ・オブジェクトを特定してマークを付けるためのメモリーです。
3. ヒープ・メモリー。ライブ・オブジェクトを特定してマークを付けるためのメモリーです。

ライブ・オブジェクトにマークが付いている場合は、マークが付いていないオブジェクトをコレクションに使用できます。

ガーベッジ・コレクション・サイクルが完了すると、Metronome ガーベッジ・コレクターはフリー・ヒープ・スペースの容量を確認します。フリー・ヒープ・スペースがまだ不足している場合は、別のガーベッジ・コレクション・サイクルが同じトリガー ID を使用して開始されます。十分なフリー・ヒープ・スペースがある場合は、トリガーが終了し、ガーベッジ・コレクション・スレッドは停止されます。アラーム・スレッドは引き続きフリー・ヒープ・スペースをモニターし、必要に応じて、別のガーベッジ・コレクション・サイクルを起動します。

Metronome ガーベッジ・コレクターの使い方について詳しくは、73 ページの『Metronome ガーベッジ・コレクターの使用』 を参照してください。

コンパイラー

IBM WebSphere Real Time for RT Linux は、さまざまなレベルのコード・パフォーマンスおよび決定論を提供する、いくつかのコード・コンパイル・モデルをサポートしています。

IBM WebSphere Real Time for RT Linux で Java コードのコンパイルに使用できるオプションには、以下のものがあります。

低優先順位の Just-In-Time (JIT) コンパイル

WebSphere Real Time for RT Linux のデフォルトのコンパイル・モデルでは、アプリケーションの実行中に、Just-In-Time コンパイラーを使用して Java アプリケーションの重要なメソッドをコンパイルします。このモードでは、JIT コンパイラーが、非リアルタイム JVM の JIT コンパイラーの動作と同様の方法で機能します。違うのは、WebSphere Real Time for RT Linux の JIT コンパイラーが、どのリアルタイム・スレッドよりも低い優先順位で実行されるという点です。優先順位が低いということは、アプリケーションがリアルタイム・タスクを実行する必要のないときに JIT コンパ

イラーがシステム・リソースを使用するということを意味します。結果として、JIT コンパイラーがリアルタイム・タスクのパフォーマンスに大きく影響することはありません。

Ahead-of-Time (AOT) プリコンパイル済みコード

WebSphere Real Time for RT Linux は、アプリケーションを実行する前のプリコンパイル・ステップで、Java メソッドをネイティブ・コードにコンパイルします。 AOT プリコンパイル済みコードを使用することで、最高レベルの決定論および良好なパフォーマンスが得られます。

混合モード (AOT プリコンパイル済みコードと低優先順位の JIT コンパイルの組み合わせ)

アプリケーションの実行時、AOT と JIT のコンパイル済みコードを一緒に使用することができます。この操作モードでは、非常に良好な決定論および良好なパフォーマンスが得られるほか、実行頻度の高いメソッドについては非常に高いパフォーマンスが得られます。

解釈操作

インタープリターは Java アプリケーションを実行しますが、コードのコンパイルは使用しません。

コンパイル済みコードの使用について詳しくは、 40 ページの『WebSphere Real Time for RT Linux でのコンパイル済みコードの使用』を参照してください。

JIT と AOT のコンパイルの比較

Ahead-of-Time (AOT) コンパイルを使用すると、コードを実行する前に、Java のクラスおよびメソッドをコンパイルすることができます。影響を受けやすいパフォーマンス・パスに対して JIT (Just-In-Time) コンパイラーが与える可能性のある予測不能なタイミング上の影響は、AOT コンパイルでは発生しません。コードが実行前にコンパイルされて、最高レベルの決定論的パフォーマンスが得られるようにするには、AOT コンパイラーを使用してコードをプリコンパイルし、共有クラス・キャッシュ内に入れます。

注: 通常、AOT コンパイルされたコードは、JIT コンパイルされたコードほど高速で実行されません。

Just-In-Time (JIT) コンパイラーは高優先順位の SCHED_OTHER スレッドとして実行されます。これは、標準の Java スレッドの優先順位よりは高く、リアルタイム・スレッドの優先順位よりは低い実行順位です。したがって、JIT コンパイルの場合、リアルタイム・コードでは非決定論的遅延は発生しません。結果として、重要なリアルタイム作業は定刻に実行されます。JIT コンパイラーがその作業より優先されることはないためです。しかし、リアルタイム・コードは、キューに入れられた hot メソッドをコンパイルするだけの十分な時間が JIT にないために、解釈されるコードとして実行されることがあります。 9 ページの図 2 に比較を示します。

通常、アプリケーションにウォームアップ・フェーズがある場合は、JIT を使用して実行し、ウォームアップ・フェーズの完了時に必要に応じて JIT を使用不可にする方が効率がよくなります。この方法を使用すると、JIT コンパイラーでアプリケーションの実行環境用コードを生成できます。

アプリケーションにウォームアップ・フェーズがなく、重要な実行パスが標準のアプリケーション操作でコンパイルされるのかどうか分からない場合、この環境では AOT コンパイルが効果的に機能します。

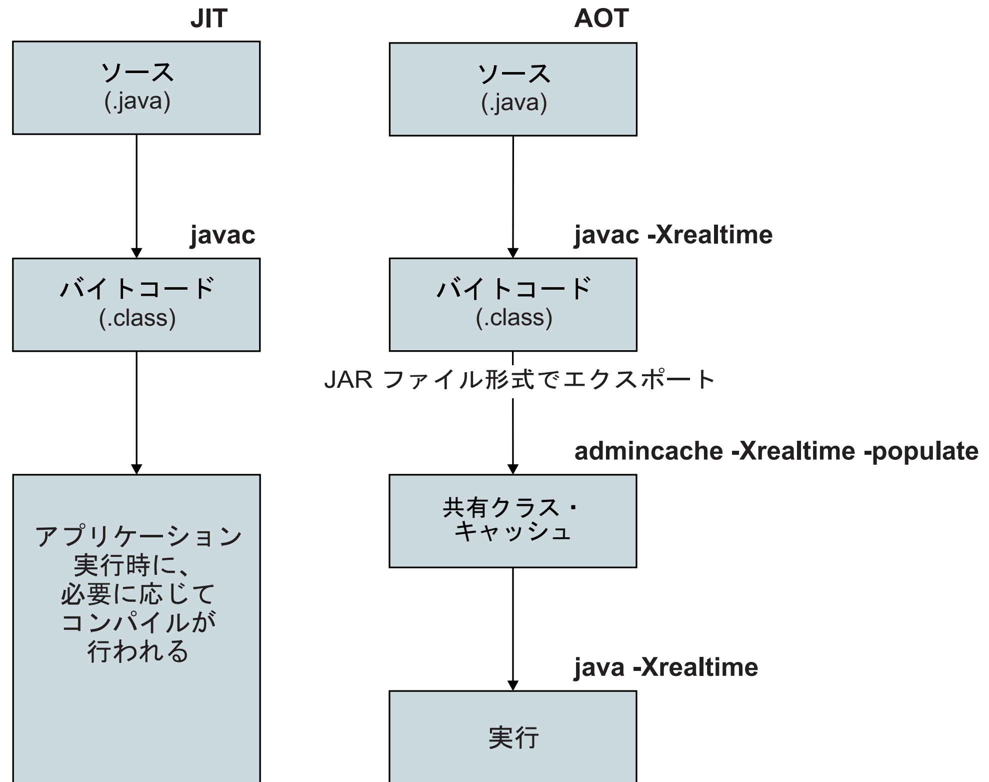


図 2. JIT コンパイラーと AOT コンパイラーの比較。

RTSJ のサポート

WebSphere Real Time for RT Linux は Real-Time Specification for Java (RTSJ) を実装します。

WebSphere Real Time for RT Linux バージョン 3.0 は、RTSJ Technology Compatibility Kit 1.0.2 バージョン J9 3.1.0 FCS に照らして RTSJ に準拠していることが認定されており、また Java Compatibility Kit (JCK) バージョン 7.0 に準拠しています。

リアルタイム・スレッドのスケジューリングおよびディスパッチング

リアルタイム Java スレッドのスレッド・スケジューリングおよびディスパッチングは、Real Time Specification for Java 一部となっています。Linux オペレーティング・システムの優先順位 11 から 89 までを使用してリアルタイム Java スレッドを優先順位付けするためには、スケジューリング・ポリシー SCHED_FIFO が使用されます。

Linux スケジューリング・ポリシーについては、 39 ページの『スレッドのスケジューリングとディスパッチング』を参照してください。

スケジュール可能オブジェクトおよびそのパラメーター

2 つのメイン・タイプのリアルタイム・スケジュール可能オブジェクト (リアルタイム・スレッドおよび非同期イベント・ハンドラー) があります。

これらのスケジュール可能オブジェクトには、以下のパラメーターが関連付けられています。

SchedulingParameters
: **PriorityParameters** は、優先順位に基づいてリアルタイム・スケジュール可能オブジェクトをスケジュールします。

ReleaseParameters
- **PeriodicParameters** は、リアルタイム・スケジュール可能オブジェクトの周期的な解放を記述します。周期的なリアルタイム・スレッドは、定期的間隔で解放されるスレッドです。
- **AperiodicParameters** は、リアルタイム・スケジュール可能オブジェクトの解放を記述します。非周期的なリアルタイム・スレッドは、不定期間隔で解放されます。

MemoryParameters
: リアルタイム・スケジュール可能オブジェクトのメモリー割り振り制約を記述します。

ProcessingGroupParameters
: WebSphere Real Time for RT Linux ではサポートされません。

優先順位スケジューラー

WebSphere Real Time for RT Linux では、スケジューラーは優先順位スケジューラーです。このスケジューラーは、名前が示すとおり、アクティブ優先順位に従って、スケジュール可能オブジェクトの実行を管理します。

スケジューラーは、スケジュール可能オブジェクトのリストを維持して、各オブジェクトが CPU で実行するためにいつ解放可能なのかを判別します。スケジューラーは、各スケジュール可能オブジェクトに関連した各種パラメーターに従う必要があります。メソッド addToFeasibility、isFeasible、および removeFromFeasibility はこのために提供されています。

優先順位およびポリシー

通常の Java スレッド (つまり、`java.lang.Thread` オブジェクトとして割り振られたスレッド) は、スケジューリング・ポリシー SCHED_OTHER、SCHED_RR、または SCHED_FIFO を使用できます。リアルタイム・スレッド (つまり、`java.lang.RealtimeThread` として割り振られたスレッド) および非同期イベント・ハンドラーは、SCHED_FIFO スケジューリング・ポリシーを使用します。

通常の Java スレッドは、ポリシー SCHED_RR または SCHED_FIFO を使用するスレッドによって JVM が開始されない限り、デフォルトのスケジューリング・ポリシー SCHED_OTHER を使用します。ポリシー SCHED_OTHER を使用する通常の Java スレッドのオペレーティング・システム・スレッド優先度は 0 に設定され

ます。ポリシー SCHED_RR または SCHED_FIFO を使用する通常の Java スレッドは、JVM を開始したスレッドの優先順位を継承します。

リアルタイム・スレッド の場合、SCHED_FIFO ポリシーではタイム・スライシングが存在せず、 1 (最低) から 99 (最高) までの 99 個の優先順位がサポートされます。この WebSphere Real Time for RT Linux 実装では、11 以上 38 以下の 28 個のユーザー優先順位がサポートされます。そのため、

```
javax.realtime.PriorityScheduler().getMinPriority()
```

により、11 が返され、

```
javax.realtime.PriorityScheduler().getMaxPriority()
```

により、38 が返されます。

OS 優先順位 81 から 89 は IBM JVM によってワーカー・スレッドをディスパッチするために使用されます。これらのスレッドはすべて、スリープ状態に戻る前に少量の作業を実行するように設計されています。これらのスレッドは以下のとおりです。

- Metronome ガーベッジ・コレクター・アラーム・スレッド。優先順位 89 で実行されます。このスレッドは定期的に実行されて、GC 作業単位をディスパッチします。
- 非同期シグナルを処理する 2 つの非同期シグナル・スレッド。1 つは優先順位が 88 の NHRT (No-Heap Real-Time) スレッドで、もう 1 つは優先順位 87 です。
- タイマー・イベントをディスパッチする 2 つのタイマー・スレッド。1 つはヒープなしタイマー用の優先順位 85 の NHRT スレッドで、もう 1 つは優先順位 83 です。
- 非同期イベント・ハンドラー・スレッド。これは非同期イベント・ハンドラーを実行するためにディスパッチされ、そのハンドラーの優先順位が割り当てられます。システムは、2 つの NHRT ハンドラー・スレッドを優先順位 85 に、そして 8 つのその他のスレッドを優先順位 83 に設定して、始動します。
- 優先順位 88 の非同期シグナルの NHRT スレッド。このスレッドは、ヒープ・ダンプ、コア・ダンプ、および javacore ダンプの要求を処理します。ダンプ・ファイルの作成時には、一時的に優先順位を 89 にランキング調整します。

Metronome GC トレース・スレッドは OS 優先順位 12 で実行され、コンパイル用に Java メソッドをサンプリングする JIT サンプラー・スレッドは OS 優先順位 13 で実行されます。

JIT コンパイル・スレッド (JIT サンプラー・スレッドとは異なる) は、SCHED_OTHER ポリシーを使用して、OS 優先順位 0 で実行されます。

-Xnojit または **-Xint** が指定されている場合は、JIT コンパイル・スレッドおよび JIT サンプラー・スレッドはともに使用不可です。

Metronome ガーベッジ・コレクター および finalizer の優先順位は、優先順位が最高のヒープ割り振りスレッドより高くなるように、(各コレクション・ラウンド前に) 常に変更されます。必ず、ヒープ割り振りスレッドの優先順位が NoHeapRealtimeThreads の優先順位より低くなるようにする必要があります。

ヒープ割り振りスレッドは、スリープ状態でもなく、モニターでブロックされていることもない任意の非 NHRT ユーザー・スレッドです。 JNI インターフェースの外部でネイティブ・コードを実行するユーザー・スレッドは、ヒープ割り振りスレッドとは見なされません。ヒープ割り振りスレッドがウェイクアップするとき、モニターでブロックされなくなっているとき、または JNI を終了するときにガーベッジ・コレクションが進行中の場合、続行する前にガーベッジ・コレクションが終了するまで待機する必要があります。

OS 優先順位 81 は、ヒープから割り振っている内部 JVM スレッド用に予約されています。内部 JVM スレッドが OS 優先順位 81 の場合、ガーベッジ・コレクターは OS 優先順位 82 で実行されます。ヒープ割り振りユーザー・スレッドが リアルタイム・スレッド でない場合、GC の優先順位は OS 優先順位 11 で実行されます。それ以外の場合は、GC は、優先順位が最高のヒープ割り振りユーザー・スレッドよりも 1 つ高い OS 優先順位で実行されます。

GC の優先順位は、コレクション・ラウンドの直前に調整されます。

優先順位のマッピングおよび継承

各 Java 優先順位は関連オペレーティング・システム・ベース優先順位にマッピングされ、各オペレーティング・システム優先順位はスケジューリング・ポリシーに関連付けられます。 WebSphere Real Time for RT Linux Linux オペレーティング・システム・スケジューリング・ポリシーは、 SCHED_OTHER、SCHED_RR、および SCHED_FIFO です。

リアルタイム Java スレッドはポリシー SCHED_FIFO を使用しますが、通常の Java スレッドは JVM を始動したスレッドのポリシーを使用します。通常の Java スレッドのデフォルト・スケジューリング・ポリシーは SCHED_OTHER ですが、**chrt** などのユーティリティーを使用して、ポリシー SCHED_RR または SCHED_FIFO を設定できます。スレッド優先順位およびポリシーについて詳しくは、39 ページの『スレッドのスケジューリングとディスパッチング』を参照してください。

以下の表に、Java 優先順位がネイティブ・オペレーティング・システム優先順位にどのようにマッピングされるのかを示します。一部の Java 優先順位は JVM 用に予約されており、対応する Java 優先順位がない一部のネイティブ優先順位も JVM によって使用されます。

注:

- 優先順位 1 から 10 は通常の Java スレッドによって使用されます。
  - ポリシー SCHED_OTHER の場合、Java 優先順位 1 から 10 は、オペレーティング・システム優先順位 0 にマップされます。
  - ポリシー SCHED_FIFO または SCHED_RR の場合、Java 優先順位 1 から 10 は、JVM を始動したスレッドの優先順位を継承します。
- 優先順位 11 以上はリアルタイム・スレッドおよび非ヒープ・リアルタイム・スレッドによって使用されます。
- スケジュール可能オブジェクトは常に、そのアクティブ優先順位で実行されます。アクティブ優先順位は、最初はスケジュール可能オブジェクトのベース優先

順位です。ただし、アクティブ優先順位は、優先順位の継承によって一時的に上げることができます。スケジュール可能オブジェクトのベース優先順位は実行中に変更できます。

ユーザー・ベース優先順位:

```
Java 優先順位 1 から 10: SCHED_OTHER、OS 優先順位 0

Java 優先順位 11: SCHED_FIFO、OS 優先順位 25
Java 優先順位 12 SCHED_FIFO、OS 優先順位 27
Java 優先順位 13 SCHED_FIFO、OS 優先順位 29
Java 優先順位 14 SCHED_FIFO、OS 優先順位 31
Java 優先順位 15 SCHED_FIFO、OS 優先順位 33
Java 優先順位 16 SCHED_FIFO、OS 優先順位 35
Java 優先順位 17 SCHED_FIFO、OS 優先順位 37
Java 優先順位 18 SCHED_FIFO、OS 優先順位 39
Java 優先順位 19 SCHED_FIFO、OS 優先順位 41
Java 優先順位 20 SCHED_FIFO、OS 優先順位 43
Java 優先順位 21 SCHED_FIFO、OS 優先順位 45
Java 優先順位 22 SCHED_FIFO、OS 優先順位 47
Java 優先順位 23 SCHED_FIFO、OS 優先順位 49
Java 優先順位 24 SCHED_FIFO、OS 優先順位 51
Java 優先順位 25 SCHED_FIFO、OS 優先順位 53
Java 優先順位 26 SCHED_FIFO、OS 優先順位 55
Java 優先順位 27 SCHED_FIFO、OS 優先順位 57
Java 優先順位 28 SCHED_FIFO、OS 優先順位 59
Java 優先順位 29 SCHED_FIFO、OS 優先順位 61
Java 優先順位 30 SCHED_FIFO、OS 優先順位 63
Java 優先順位 31 SCHED_FIFO、OS 優先順位 65
Java 優先順位 32 SCHED_FIFO、OS 優先順位 67
Java 優先順位 33 SCHED_FIFO、OS 優先順位 69
Java 優先順位 34 SCHED_FIFO、OS 優先順位 71
Java 優先順位 35 SCHED_FIFO、OS 優先順位 73
Java 優先順位 36 SCHED_FIFO、OS 優先順位 75
Java 優先順位 37 SCHED_FIFO、OS 優先順位 77
Java 優先順位 38 SCHED_FIFO、OS 優先順位 79
```

内部ベース優先順位:

```
内部 Java 優先順位 39: SCHED_FIFO、OS 優先順位 81
内部 Java 優先順位 40 SCHED_FIFO、OS 優先順位 83
内部 Java 優先順位 41 SCHED_FIFO、OS 優先順位 84
内部 Java 優先順位 42 SCHED_FIFO、OS 優先順位 85
内部 Java 優先順位 43 SCHED_FIFO、OS 優先順位 86
内部 Java 優先順位 44 SCHED_FIFO、OS 優先順位 87
内部 Java 優先順位 45 SCHED_FIFO、OS 優先順位 88
OS 優先順位 11、12、13
偶数の OS 優先順位 26、28、30、...、82
OS 優先順位 89
```

http://www.rtsj.org/specjavadoc/book_index.html のセクション『Synchronization』も参照してください。

優先順位の継承:

スレッドのアクティブ優先順位は、より高い優先順位のスレッドによって要求されるロックを保持するため、一時的にランキング調整されることがあります。このようなロックは、内部 JVM ロック、または同期メソッドまたは同期ブロックに関連したユーザー・レベル・モニターである可能性があります。そのため、通常の Java スレッドの優先順位は、スレッドがロックを解放する時点まで、一時的にリアルタイム優先順位を持つ場合があります。

優先順位の継承の 1 つの結果として、SCHED_OTHER スレッドのスレッド・ポリシーが、一時的に SCHED_FIFO に変更されます。

ベース優先順位およびアクティブ優先順位について詳しくは、 RTSJ 仕様のセクション『Synchronization』を参照してください。

メモリー管理

メモリー・ヒープのガーベッジ・コレクションは、ガーベッジ・コレクションによって発生する予測不能な動作があるために、リアルタイム・プログラミングにおいては常に障害であると考えられています。 IBM WebSphere Real Time for RT Linux の Metronome ガーベッジ・コレクターを使用すると、高い水準の決定論的 GC パフォーマンスを実現できます。同時に、Real-Time Specification for Java (RTSJ) では、ガーベッジ・コレクションが実行されたヒープの外側にあるオブジェクトのメモリー・モデルに対していくつかの拡張機能が用意されているため、Java プログラマーは存続時間の短いオブジェクトおよび存続時間の長いオブジェクトの両方を明示的に管理できます。

メモリー域

RTSJ には、オブジェクトの割り振りに使用できるメモリー域という概念があります。一部のメモリー域はヒープの外側に置かれ、システムおよびガーベッジ・コレクターがオブジェクトに対して実行できる内容に制限が加えられています。例えば、いくつかのメモリー域にあるオブジェクトは決してガーベッジ・コレクションの対象になりませんが、ガーベッジ・コレクターがこれらのメモリー域でヒープ内のオブジェクトに対する参照をスキャンし、ヒープの整合性を保持することはできます。

メモリー管理には、以下の 3 つの基本的なタイプがあります。

- ヒープ・メモリーは従来型の Java ヒープですが、Metronome ガーベッジ・コレクターによって管理されます。
- スコープ・メモリーはアプリケーションから明確に要求する必要があります。また、このメモリーを使用できるのは、非ヒープ・リアルタイム・スレッドおよび非ヒープ非同期イベント・ハンドラーを含むリアルタイム・スレッドだけです。
- 永久メモリーは、スケジュール可能なオブジェクト (具体的には非ヒープ・リアルタイム・スレッドや非ヒープ非同期イベント・ハンドラー) によって参照可能なオブジェクトが入れられるメモリー域を表します。このメモリーは、アプリケーションで使用されない場合でも、クラス・ロードおよび静的な初期化で使用されます。

永久メモリーまたはスコープ・メモリーは、物理メモリーを使用するように指定できます。物理メモリーは、かなり高速なアクセスなどの特性を備えたメモリー領域から構成されています。通常、物理メモリーの使用頻度は高くないため、標準的な JVM ユーザーに影響を与える可能性はあまりありません。

ヒープ・メモリー

最大サイズは **-Xmx** によって制御されますが、初期ヒープ・サイズ (**-Xms**) は設定し*ない* ようにするか、最大ヒープ・サイズ **-Xmx** と同値に設定するようにしてください。リアルタイムでは、ヒープが初期ヒープ・サイズから最大ヒープ・サイズまで大きくなることがないためです。最大ヒープ・サイズに達してフリー・スペースがなくなると、OutOf MemoryError が発生します。通常、リアルタイム JVM は従来型の JVM よりも多くのヒープ・メモリーを消費します。これは、決定論的収集を

サポートするために、オブジェクトを異なる方法で編成する必要があり、結果としてヒープのフラグメント化が進むためです。また、配列は複数のフラグメントに分割され、それぞれのフラグメントにヘッダーが付けられます。大小のオブジェクトの比率および配列の使用量によって変わりますが、多くの場合、アプリケーションでは 20% を超えるヒープ・スペースが必要になります。

Metronome ガーベッジ・コレクターは、アプリケーションの実行中にガーベッジを収集するという点で、主流の JVM にある「ほとんどコンカレントの」コレクターに似ています。理想世界では、アプリケーションがメモリー不足になる前にコレクション・サイクルが完了しますが、非常に高速な割り振りを行う一部のアプリケーションは、Metronome ガーベッジ・コレクターの収集能力を上回る速度で割り振りを行うことがあります。収集速度は各種の細かい制御によって影響を受けますが、最終的に OutOf MemoryError をスローする前に、Metronome を従来からの stop-the-world 型 GC に強制的に戻す制御が 1 つあります。ランタイム・パラメーターは **-Xgc:synchronousGCOnOOM** で、それと対をなすパラメーターは **-Xgc:nosynchronousGCOnOOM** です。デフォルトは **-Xgc:synchronousGCOnOOM** です。

スコープ・メモリー

RTSJ には、スコープ・メモリーという概念があります。これは、明確に定義された存続時間を持つオブジェクトから使用できます。スコープは、明示的に入力することも、オブジェクトの run() メソッドを実行する前に事実上スコープ内に入るスケジュール可能なオブジェクト (リアルタイム・スレッドまたは非同期イベント・ハンドラー) に付加することもできます。各スコープには参照カウントがあり、これがゼロになった時点で、そのスコープにあるオブジェクトを閉じる (ファイナライズする) ことができるようになり、そのスコープに関連付けられているメモリーが解放されます。ファイナライズが完了するまで、そのスコープの再使用はブロックされます。

スコープ・メモリーは、VTMemory と LTMemory という 2 つのタイプに分けることができます。スコープ・メモリーのこれらのタイプは、領域からオブジェクトを割り振るために必要な時間によって変わります。 LTMemory では、メモリー域のメモリー使用量がそのメモリー域の初期サイズを下回る場合に線形時間での割り振りが保証されます。 VTMemory にそのような保証はありません。

スコープはネストすることができます。ネストされたスコープが入力されると、以降のすべての割り振りは、その新規スコープに関連付けられたメモリーから取得されます。ネストされたスコープが完了すると、直前のスコープが元に戻され、以降の割り振りは再度そのスコープから取得されます。

スコープ・オブジェクトの存続時間の関係から、一連の制限付き割り当て規則によって、スコープ・オブジェクトへの参照を制限する必要があります。スコープ・オブジェクトへの参照を、外部スコープの変数に割り当てたり、ヒープ域または永久域のいずれかにあるオブジェクトのフィールドに割り当てたりすることはできません。スコープ・オブジェクトへの参照は、同じスコープ内または内部スコープ内にのみ割り当てることができます。仮想マシンは、誤った割り当てが試行されると、それらを検出し、IllegalAssignmentError 例外をスローします。スコープ・メモリーのタイプを柔軟に選択できることで、アプリケーションは、コード内の構文的に定義された特定の領域に適した特性を持つメモリー域を使用できます。

領域のサイズはその領域の作成時に指定する必要があり、コマンド行パラメーター **-Xgc:scopedMemoryMaximumSize** によってその最大値が制御されます。デフォルトは 8 MB です。大部分の目的ではこれで十分です。

永久メモリー

永久メモリーは、アプリケーション内のすべてのスケジュール可能オブジェクトおよびスレッドの間で共有されるメモリー・リソースです。永久メモリーに割り振られるオブジェクトは、常に非ヒープ・スレッドおよび非同期イベント・ハンドラーから使用することができます。また、ガーベッジ・コレクションによって発生する遅延の影響を受けません。オブジェクトは、プログラムの終了時にシステムによって解放されます。

サイズは **-Xgc:immortalMemorySize** によって制御されます。例えば、**-Xgc:immortalMemorySize=20m** とした場合は 20 MB に設定されます。デフォルトは 16 MB です。大量のクラス・ロードを行わない限り、通常はこれで十分です。ほとんどの OutOfMemoryError 例外の原因は、クラス・ロードであると考えられます。

メモリー所要量の見積もり

十分なメモリーを割り振るために必要となる情報の入手方法です。

妥当な方法としては、予期されるオブジェクトを保持するために必要なメモリーに十分な安全マージンを持たせた量を指定します。アプリケーションの分析は、必要なオブジェクトの数および性質を明らかにする上で役立ちます。ただし、オブジェクトに必要な実際のサイズは、システムによって変わることがあります。SizeEstimator クラスを使用すると、実際のオブジェクト・サイズを考慮に入れたより対応範囲の広い情報が得られます。

SizeEstimator クラス

SizeEstimator クラスは、オブジェクトの格納に必要なメモリーの量に関するガイダンス情報を提供します。この見積もりは、オブジェクト自体のために割り振るべき最小メモリー・スペースを示すもので、オブジェクトが例えばその作成時などに必要とする可能性がある他のリソースのためのメモリー所要量は考慮に入っていません。

このクラスについて詳しくは、http://www.rtsj.org/specjavadoc/book_index.html を参照してください。

メモリーの使用

Java スレッド、リアルタイム・スレッド、および非ヒープ・リアルタイム・スレッドを比較します。

Real-Time Specification for Java (RTSJ) では、リアルタイム・スレッドをサポートするために RealtimeThread クラスおよび NoHeapRealtimeThread クラスという 2 つのクラスが追加されています。

- リアルタイム・スレッドと非ヒープ・リアルタイム・スレッドはどちらもスケジュール可能なオブジェクトです。スケジュール可能なオブジェクトとして、これらのスレッドには、リリース、スケジューリング、メモリー、および処理グループというパラメーターがあります。
- リアルタイム・スレッドからは、スコープ・メモリーと永久メモリー内のオブジェクトだけでなく、ヒープ・メモリー内のオブジェクトにもアクセスできます。
- 非ヒープ・リアルタイム・スレッドからアクセスできるのは、スコープ・メモリー域と永久メモリー域のみです。
- 非ヒープ・リアルタイム・スレッドには、他のリアルタイム・スレッドよりも高い優先順位が必要です。この優先順位が他のリアルタイム・スレッドよりも低い場合、ガーベッジ・コレクターからの干渉を受けることなく実行されるという利点がなくなります。

注: 他のリアルタイム・スレッドよりも優先順位が高い非ヒープ・リアルタイム・スレッドは、ガーベッジ・コレクションによって中断されることはありません。

表 3. リアルタイム・スレッドおよび非ヒープ・リアルタイム・スレッドによるメモリー・アクセス

スレッド	永久メモリー	スコープ・メモリー	ヒープ・メモリー
通常のスレッド	✓	✗	✓
リアルタイム・スレッド	✓	✓	✓
非ヒープ・リアルタイム・スレッド	✓	✓	✗

メモリー域のタイプ

永久メモリー

永久メモリーは、ガーベッジ・コレクションの対象にはなりません。永久メモリーにスペースを割り振った場合、アプリケーションが終了するまで、そのスペースを再利用することはできません。

- 永久メモリーの性質上、メモリーを再使用する方法は複数あります。その 1 つとして、再使用可能オブジェクトのプールを作成することが考えられます。スコープ・メモリーを使用することもできます。
- 永久メモリーのオブジェクトは、スコープ・メモリーのオブジェクトを参照することはできません。永久メモリーにあるオブジェクトのフィールドにスコープ・メモリーのオブジェクトを割り当てると、IllegalAssignmentError 例外がスローされます。

スコープ・メモリー

スコープ・メモリーは、スケジュール可能オブジェクトの初期メモリー域として使用でき、スケジュール可能オブジェクトを使用して入力することもできます。参照されなくなった場合は、メモリー域からすべてのオブジェクトがクリアされます。スコープ・メモリー域で実行されているスケジュール可能オブジェクトは、その領域からすべてのオブジェクト割り振りを実行します。スコープ・メモリー域が使用されていない場合は、そのメモリー域内のオブジェクト

がファイナライズされ、メモリーが再利用されて、スコープを再使用するための準備が行われます。スコープ・メモリー域がスケジュール可能オブジェクトで使用できなくなった場合は、メモリーが他のユーザー用に再利用されます。

ScopedMemory インスタンスで示されるメモリー域は、Java ヒープには存在せず、ガーベッジ・コレクションの対象にはなりません。ScopedMemory オブジェクトを NoHeapRealtimeThread に関連付けられている初期メモリー域として使用するか、あるいは、NoHeapRealtimeThread 内で ScopedMemory.enter メソッドを使用してメモリー域を入力すると安全です。

物理メモリー

物理メモリーは、メモリー自体の特性 (ページング不可や不揮発性など) が重要な場合に使用します。

線形時間割り振りスキーム (LTMemory)

LTMemory は、メモリー域からのメモリー消費量がメモリー域の初期サイズよりも少ない場合に、線形時間を割り振るためにシステムによって保証されるメモリー域を示します。割り振りの実行時間は、メモリー消費量がメモリー域の初期サイズと最大サイズの範囲内にある場合に変更することができます。また、基礎システムでは、初期サイズと最大サイズの範囲内にあるメモリーが常に使用可能であることを保証する必要はありません。

可変時間割り振りスキーム (VTMemory)

VTMemory は、**VTMemory** 域からの割り振りの実行時間が線形時間内である必要はないという点を除けば、LTMemory と同じです。

ヒープ・メモリー

ヒープ・メモリーのオブジェクトは、スコープ・メモリーのオブジェクトを参照することはできません。ヒープ・メモリーにあるオブジェクトのフィールドにスコープ・メモリーのオブジェクトを割り当てると、IllegalAssignmentError 例外がスローされます。

同期およびリソース共有

リアルタイム・システムでは、異なる優先順位で実行される 3 つ以上のスレッドが相互に同期するときに、優先順位逆転と呼ばれる状態になることがあります。この状態では、より高い優先順位のスレッドの実行が、より低い優先順位のスレッドによって長時間ブロックされます。 WebSphere Real Time for RT Linux は、この状態を回避するために優先順位継承というスキームを使用します。

より高い優先順位のタスクの実行がより低い優先順位のタスクによってブロックされる場合、高い優先順位のタスクがブロックされなくなるまでの間、低い優先順位のタスクが一時的にランキング調整されて、高い優先順位と一致するようになります。

周期的パラメーターと非周期的パラメーター

リアルタイム・スレッドでは、スケジュール可能なオブジェクトをどのような頻度でリリースするのかを決定する、いくつかのリリース・パラメーターが使用できます。リリース・パラメーターの例として、周期的パラメーターおよび非周期的パラメーターがあります。

周期的パラメーター

このクラスは、定期的間隔でリリースされるスケジュール可能オブジェクトで使用されます。

AbsoluteTime
: ミリ秒およびナノ秒単位で表示されます。

RelativeTime
: 特定のイベントの時間の長さをミリ秒およびナノ秒単位で表したものです。例えば、イベントが開始したときと終了したときの絶対時刻を測定することができます。その後で、2 つの測定値の差によって相対時間を計算することができます。

非周期的パラメーター

このクラスは、非定期的間隔でリリースされるスケジュール可能オブジェクトで使用されます。最初の非周期的イベントが完了する前に 2 番目のイベントが発生する可能性があるため、未解決要求のキューの長さを定義することができます。

非同期イベント処理

非同期イベント・ハンドラーは、スレッドの外部で発生するイベント (例えば、アプリケーションのインターフェースからの入力) に反応します。リアルタイム・システムでは、これらのイベントには、アプリケーションに設定された期限内に応答する必要があります。

非同期イベントはシステム割り込みおよび POSIX シグナルと関連付けることができ、またタイマーとリンクさせることができます。

リアルタイム・スレッド と同様に、非同期イベント・ハンドラー にはいくつかのパラメーターが関連付けられています。これらのパラメーターのリストは、 10 ページの『スケジュール可能オブジェクトおよびそのパラメーター』に記載されています。

シグナル・ハンドラー

POSIXSignalHandler は、SIGQUIT シグナル、SIGTERM シグナル、および SIGABRT シグナルをサポートします。 SIGQUIT のデフォルト動作では Javadump が生成されます。 CPU 時間およびファイルの読み取りと書き込みを除き、Javadump の生成によって実行中のプログラムの操作が妨げられることはありません。 Javadump が生成されると、Javadump が完了するまでプログラムが中断されます。Javadump が生成されている間は、アプリケーションのパフォーマンスは予測できません。

障害時にすべてのコア・ダンプおよび Javadump の生成を抑止するためには、**-Xdump:none** を使用してください。

SIGQUIT シグナルが出されたときにシステム・ダンプと Javadump の生成のみを抑止するためには、**-Xdump:java:none -Xdump:java:events=gpf+abort** を指定してください。

以下のシグナルは、POSIXSignalHandler メカニズム (/usr/include/bits/signum.h で定義されているシグナル記述) によって 非同期イベント・ハンドラー (AEH) に付加することができます。

```
#define SIGQUIT        3       /* Quit (POSIX).  */
#define SIGABRT        6       /* Abort (ANSI).  */
#define SIGKILL        9       /* Kill, unblockable (POSIX).  */
```

その他のシグナルは、現在サポートされていません。上のリストにあるすべてのシグナルは非同期シグナルであり、外部で生成されたイベントではなくアプリケーションまたは JVM コードの障害を示すものであるため、同期シグナル (SIGILL や SIGSEGV など) への付加はサポートされません。

注: デフォルトでは、JVM によって SIGQUIT が受信されると、Java アプリケーションがダンプ (例えば、Javadump) を生成します。 SIGQUIT は何らかの付加された AEH にも送信されますが、この送信により、紛らわしい動作または望ましくない動作が行われることがあります。これは、Java コマンド行で **-Xdump:none:events=user** オプションを使用することによって無効にできます。

必要な文書

WebSphere Real Time for RT Linux は Real-Time Specification for Java (RTSJ) を実装します。

WebSphere Real Time for RT Linux バージョン 2.0 は、RTSJ Technology Compatibility Kit バージョン 3.0.13 FCS に照らして RTSJ 1.0.2 に準拠していることが認定されており、またバージョン 6.0 の Java Compatibility Kit (JCK) に準拠しています。

サポートされる機能

以下の機能がサポートされています。

- ヒープ割り振りでの割り振り速度の制約。スケジュール可能なオブジェクトがヒープ内でオブジェクトを作成する比率を制限します。

サポートされない機能

以下の機能はサポートされていません。

- 優先順位シーリング・エミュレーション・プロトコル。例えば、PriorityCeilingEmulation をモニター制御ポリシーとして使用することを許可しません。
- アトミック・アクセス・サポート (仕様に適合させるために必要な場合を除きます)。
- 基本優先順位スケジューラー以外のスケジューラーは、アプリケーションでは使用できません。

- コストの制約。

Real-Time Specification for Java のために必要な文書

このセクションでは、Real-Time Specification for Java (RTSJ) の『必要な文書』セクションが引用されています。 RTSJ の標準実装と異なる点については注記が行われています。

1. **実現可能性検査アルゴリズムはデフォルトのアルゴリズムです。**

   「実現可能性検査アルゴリズムがデフォルトではない場合には、実現可能性検査アルゴリズムについて文書で説明してください。」
2. **アプリケーションでは、基本優先順位スケジューラーのみを使用できます。**

   「基本優先順位スケジューラー以外のスケジューラーがアプリケーションで使用できる場合には、スケジューリングの章で詳しく説明されているように、スケジューラーの動作およびスケジューラーの相互作用について文書化してください。また、そのスケジューラーのスケジュール可能オブジェクトを構成するクラスのリストも文書化してください (ただし、このリストが基本スケジューラーのスケジュール可能オブジェクトのリストと同じである場合は、文書化は不要です)。」
3. **より高い優先順位のスケジュール可能オブジェクトによって占有されているスケジュール可能オブジェクトは、その優先順位のキューの先頭に置かれます。**

   「より高い優先順位のスケジュール可能オブジェクトによって占有されているスケジュール可能オブジェクトは、そのアクティブ優先順位のキュー内の、実装環境によって決められた位置に置かれます。占有されているスケジュール可能オブジェクトが該当のキューの先頭に置かれない場合、実装環境で、その配置に使用されるアルゴリズムについて文書で説明する必要があります。この仕様の将来のバージョンでは、キューの先頭への配置が必須になる可能性があります。」
4. **コスト制約はサポートされていません。**

   「実装環境でコスト制約がサポートされている場合には、実装環境では、現行 CPU 使用量が更新される頻度について文書で説明する必要があります。」
5. **単純順次マッピングはサポートされています。**

   「何らかの物理メモリー・タイプのフィルターによって実装されたメモリー・マッピングは、連続バイトの単純順次マッピングでない限り、文書で説明する必要があります。」
6. **WebSphere Real Time for RT Linux では、Metronome ガーベッジ・コレクターのサブクラスが提供されていません。**

   「実装環境で、GarbageCollector のすべてのサブクラスの動作について文書で詳しく説明する必要があります。」
7. **WebSphere Real Time for RT Linux では MonitorControl サブクラスが提供されていません。**

   「この仕様で詳しく説明されていない MonitorControl サブクラスを提供する実装環境では、その影響について (特に、優先順位逆転制御について、また、新

規ポリシーをサポートできないスケジューラーがある場合にはそのスケジューラーについて) 文書で説明する必要があります。」

8. **より高い優先順位のスケジュール可能オブジェクトによって必要とされるモニターを保持しているスケジュール可能オブジェクトの優先順位は、モニターの解放などが行われるときまで、より高い優先順位にランキング調整されます。SUSE Linux Enterprise Real Time 10 SP2 更新カーネル・バージョン 2.6.22.19-0.16 より前のバージョンのカーネル、および Red Hat Enterprise Linux 5.1 MRG 2.6.24.7-73 Errata 1 で実行される場合 (これらのレベル以上のカーネルでは、キューの先頭にスケジュール可能オブジェクトが置かれます)、この時点でスケジュール可能オブジェクトが実行可能でなくなる (つまり、より高い優先順位の作業が行われる) と、そのスケジュール可能オブジェクトは元の (ランキング調整されない) 優先順位のキューの最後に置かれます。**

   「優先順位逆転回避アルゴリズムが使用されるために、「ランキング調整された」優先順位が適用されなくなったときに、スケジュール可能オブジェクトが新規キューの先頭に置かれない場合には、実装環境で、キューイングの動作について文書で説明する必要があります。」

9. **基本スケジューラーは、WebSphere Real Time for RT Linux で提供される唯一のスケジューラーです。**

   「基本スケジューラー以外に使用可能なスケジューラーがある場合に、同期の意味体系がデフォルトの PriorityInheritance インスタンスのために定義された規則と異なっているときには、どのように異なるのかを実装環境で文書で説明する必要があります。同期の章に記載されている基本優先順位スケジューラーでの意味体系と同等の優先順位継承 (および、サポートされる場合には、優先順位シーリング・エミュレーション・プロトコル) が使用される新規スケジューラーの動作について文書で説明する必要があります。」

10. **イベントが起動されてから関連するバインド済みイベント・ハンドラーがスケジューリングされるまでの最悪のケースの時間は、(それ以上の優先順位の競合するスケジュール可能オブジェクトまたはシステム活動がなく、またガーベッジ・コレクションによる介入がない場合には) 平均 40µs であり、100µs を超えることはありません。 fire メソッド、AsyncEvent オブジェクト、またはハンドラーを駆動するスケジュール可能オブジェクトがヒープを参照する場合、ガーベッジ・コレクションによる潜在的な影響は (A) に記載されたとおりです。これは、コードが解釈されること、および (バインド済みの) 単一ハンドラーがそのイベントで構成されていることを前提としています。**

    「バウンドの発生が原因で AsyncEvent が起動されてから関連する AsyncEventHandler がリリースされるまでの (それ以上の優先順位のスケジュール可能オブジェクトが実行可能になっていない場合の) 最悪のケースの応答間隔を、何らかの参照アーキテクチャーに関して文書で説明する必要があります。」

11. **ATC 対応スレッドで AsynchronouslyInterruptedException が起動されてからその例外が最初に送信されるまでの、最悪のケースの間隔は (それ以上の優先順位の競合するスケジュール可能オブジェクトまたはシステム・アクティビティーがなく、またガーベッジ・コレクションによる介入がない場合には) 平均 35µs であり、160µs を超えることはありません。この場合の ATC 対応とは、スレッドが、ATC-deferred でない領域内で AI 対応メソッドで実行されている**

こと、および例外の送信までそれらの状態が維持されることを意味します。ガーベッジ・コレクションによる潜在的な影響は (A) に記載されたとおりです。ターゲット・スレッドがネイティブ・コードに含まれている場合、無限の遅延が発生する可能性があります。これは、コードが解釈されることを前提としています。

「ATC 対応スレッドで AsynchronouslyInterruptedException が起動されてからその例外が最初に送信されるまでの (それ以上の優先順位のスケジュール可能オブジェクトが実行可能になっていない場合の) 間隔を、何らかの参照アーキテクチャーに関して文書で説明する必要があります。」

12. **適用外。応答 4 を参照してください。**

    「コスト制約がサポートされていて、実装環境で、スコープ・メモリー内のオブジェクトのファイナライザーを実行するコストを、スコープを終了させることによってそのコストの参照カウントをゼロに削減したオブジェクト以外のスケジュール可能オブジェクトに割り当てる場合、コストを割り当てるための規則について文書で説明する必要があります。」

13. **RealtimeSecurity の標準実装に対しては、変更が行われていません。**

    「RealtimeSecurity の実装環境が、必要とされる実装環境より多くの制限がある場合、またはランタイム構成オプションを備えている場合には、それらの機能について文書で説明する必要があります。」

14. **スコープ・メモリー域内のオブジェクトのためのファイナライザーは、その領域を参照する最後のスレッドによって、つまり、スレッドの参照カウントが 1 からゼロに減少したときに実行されます。ファイナライザーの実行に関連するすべてのコストは、そのスレッドに割り当てられます。**

    「実装環境では、スコープ・メモリー内のオブジェクトに関するファイナライザーを、そのスコープが再入される前、かつそのスコープに関する何らかの getReferenceCount() 呼び出しから戻る前に実行することができます。」ただし、そのようなファイナライザーを実行する場合には、文書による説明が必要です。

15. **レゾリューションは設定できません。**

    「サポート対象のクロックごとに、レゾリューションが設定可能であるかどうか、また設定可能な場合にはサポートされる値を、文書で指定する必要があります。」

16. **WebSphere Real Time for RT Linux では、リアルタイム・クロック以外のクロックは提供されていません。**

    「実装環境に、必須のリアルタイム・クロック以外のクロックが組み込まれている場合には、どのようなコンテキストでそれらのクロックを使用できるのか、文書で示す必要があります。」

注:

A テスト用の参照アーキテクチャーは、LS20、4–way、1 MB キャッシュを備えた 2 GHz、および 4 GB のメモリーです。

B ガーベッジ・コレクションにより、ヒープに関連したスレッド内の何らかの個所で遅延が発生する可能性があります。ガーベッジ・コレクターは、ヒープ・メモリーを使い尽くした場合の動作を管理する、2 つの基本モードのいずれかで作動させることができます。ガーベッジ・コレクターが、このような状況で直ちに OutOfMemoryError をスローするように設定されている場合、最悪のケースのガーベッジ・コレクション遅延は、一般には 1 ms 未満です。現在では、一部の状況で (例えば、深くネストされたスタックを含む多数のスレッド、または多数の大規模スコープがある場合)、遅延が長くなる可能性があります。ガーベッジ・コレクターが、OutOfMemoryError をスローする前に同期ガーベッジ・コレクションを実行するように設定されている場合、潜在的なコレクション遅延は、ヒープ内の稼働中オブジェクトの数、および他のメモリー域内のオブジェクトの数に関連します。このような状況では、一般的なヒープ・サイズの場合に何秒も遅延する可能性があるため、遅延は無制限であると考えられます。

第 3 章 計画

WebSphere Real Time for RT Linux をインストールする前に、このセクションを読んでください。

- 『マイグレーション』
-
- 『ハードウェアおよびソフトウェアの前提条件』
- 26 ページの『考慮事項』

マイグレーション

WebSphere Real Time for RT Linux は、リアルタイム・アプリケーション用に変更された Linux 環境で稼働します。リアルタイム環境では、標準的な Java アプリケーションを使用できます。あるいは、WebSphere Real Time の機能を活用するために、ご使用のアプリケーションを変更することもできます。

システムのマイグレーション

Linux サポート・チームからの指示に従ってください。

ハードウェアおよびソフトウェアの前提条件

WebSphere Real Time for RT Linux のためにサポートされているハードウェア、オペレーティング・システム、および Java 環境を確認するには、このリストを使用してください。

ハードウェア

WebSphere Real Time for RT Linux の認定ハードウェア構成は、以下のシステムのマルチプロセッサー・バリアントです。

- IBM BladeCenter® LS20 (タイプ 8850-76U、8850-55U、7971、7972)
- IBM eServer™ xSeries® 326m (タイプ 7969-65U、7969-85U、7984-52U、7984-6AU)
- IBM BladeCenter LS21 (タイプ 7971-6AU)
- IBM BladeCenter HS21 XM Dual Quad Core (タイプ 7995)

WebSphere Real Time for RT Linux 用の認定を維持するためには、ハイパースレッディングがサポートされる IBM システムでは、ハイパースレッディングを有効にしないようにする必要があります。

また、WebSphere Real Time for RT Linux は、以下の特性を備えた、サポート対象のオペレーティング・システムを実行するハードウェアでサポートされます。

- 最小で 512 MB の物理メモリー。
- 最低で Intel Pentium 4、AMD Opteron、または Intel Atom Processor。

© Copyright IBM Corp. 2003, 2011

認定されたハードウェア構成になっていないシステムの場合、IBM はパフォーマンスに関して言及しません。認定されたハードウェア構成でのパフォーマンスの考慮事項は、 101 ページの『第 7 章 パフォーマンス』で詳しく説明されています。

ハイパースレッディングがサポートされるシステムでは、WebSphere Real Time for RT Linux を使用する際にパフォーマンスが悪影響を受けないようにするために、ハイパースレッディングを無効にしてください。

オペレーティング・システム

- Red Hat Enterprise Linux 5.3 MRG。 29 ページの『Real Time Linux 環境のインストール』を参照してください。
- SUSE Linux Enterprise Real Time (SLERT) 10。 29 ページの『Real Time Linux 環境のインストール』を参照してください。

考慮事項

WebSphere Real Time for RT Linux の使用時には、いくつかの要因に配慮する必要があります。

- 可能な場合には、複数のリアルタイム JVM を同一のシステムで実行しないようにしてください。複数のリアルタイム JVM を実行すると、ガーベッジ・コレクターが複数実行されるためです。各 JVM は、他の JVM のメモリー域について認識しません。結果として、JVM 間で GC サイクルと休止時間を調整することはできません。つまり、ある JVM が別の JVM の GC パフォーマンスに悪影響を及ぼす可能性があります。 複数の JVM を使用する必要がある場合は、**taskset** コマンドを使用して、各 JVM が特定のプロセッサーのサブセットにバインドされるようにしてください。
- Ahead-of-Time コンパイラーを使用してプリコンパイルされたコードでは、**-Xdebug** オプションと **-Xnojit** オプションを使用することはできません。**-Xdebug** は Ahead-of-Time (AOT) コンパイラーとは異なる方法でコードをコンパイルするため、サポートされないからです。

  コードをデバッグするには、解釈されたコードまたは JIT コンパイルされたコードを使用してください。
- javax.realtime パッケージを使用する Java ソース・コードをコンパイルするために、com.sun.tools.javac.Main インターフェースを使用する場合には、クラスパスに `sdk/jre/lib/i386/realtime/jclSC170/realtime.jar` が含まれていることを確認する必要があります。このタイプのコンパイルの一般的な例として、ant コンパイルがあります。
- オプションの JavaComm パッケージは WebSphere Real Time for RT Linux にインストールして、リアルタイム JVM と非リアルタイム JVM の両方からアクセスすることができます。インストールと構成について詳しくは、『http://publib.boulder.ibm.com/infocenter/java7sdk/v7r0/topic/com.ibm.java.lnx.70.doc/user/jcommchapter.html』を参照してください。WRT におけるリアルタイム JVM は、通常の Java スレッドで使用する JavaComm API をサポートします。ただし、JavaComm を使用して外部デバイスにアクセスする場合、決定論やリアルタイム・パフォーマンスに関する保証は得られません。したがって、非ヒープ・リアルタイム・スレッドおよびリアルタイム・スレッドとともに、あるいはリアルタイム動作が必要な場合には、JavaComm を使用しないでください。

- 以前のリリースの WebSphere Real Time for RT Linux でプリコンパイル済みコードとクラスの保管に使用された共有キャッシュは、このリリースの WebSphere Real Time for RT Linux で使用されるキャッシュとは互換性がありません。以前のキャッシュの内容を再生成する必要があります。
- 共有クラス・キャッシュを使用する際、そのキャッシュの名前は 53 文字以下にする必要があります。
- **ps** コマンドは、Java スレッド名を切り捨てます。

  ps コマンドの長さは 15 文字までに制限されています。15 文字を超える長さのスレッド名を指定した場合、その名前は **ps** コマンドによって切り捨てられます。
- WebSphere Real Time for RT Linux は NTLoginModule (NTLM) 認証をサポートしていません。

  NTLoginModule (NTLM) は、Windows サービスへのアクセス認証を支援するために使用されます。 NTLM を使用した認証は、Windows プラットフォームでのみサポートされています。つまり、WebSphere Real Time for RT Linux は NTLM 認証をサポートしていません。

第 4 章 WebSphere Real Time for RT Linux のインストール

この製品をインストールするには、以下の手順に従ってください。

- 『インストール・ファイル』
- 『Real Time Linux 環境のインストール』
- 30 ページの『InstallAnywhere パッケージからのインストール』
  - 31 ページの『手動インストールの実行』
  - 32 ページの『自動インストールの実行』
  - 34 ページの『既知の問題と制約』
- 35 ページの『パスの設定』
- 36 ページの『クラスパスの設定』
- 37 ページの『インストール済み環境のテスト』
- 38 ページの『WebSphere Real Time for RT Linux のアンインストール』

インストール・ファイル

以下のインストール・ファイルが必要です。

IBM WebSphere Real Time for RT Linux は 2 種類の InstallAnywhere パッケージで提供されます。

インストール可能なパッケージ

インストール可能なパッケージでは、システムが構成されます。例えば、プログラムは環境変数を設定する場合があります。

- `wrt-3.0-0.0-rtlinux-x86_32-sdk.bin`
- `wrt-3.0-0.0-rtlinux-x86_32-jre.bin`

アーカイブ・パッケージ

これらのパッケージでは、ファイルはシステム解凍されますが、構成は実行されません。

- `wrt-3.0-0.0-rtlinux-x86_32-sdk.archive.bin`
- `wrt-3.0-0.0-rtlinux-x86_32-jre.archive.bin`

Real Time Linux 環境のインストール

WebSphere Real Time for RT Linux をインストールするには、まず Real Time Linux をインストールする必要があります。

作業を開始する前に

WebSphere Real Time for RT Linux をインストールするには、まず 64 ビット版の Real Time Linux をインストールする必要があります。

Red Hat Enterprise Linux 5.3 MRG

© Copyright IBM Corp. 2003, 2011

- Red Hat Enterprise Linux 5.3 MRG のリアルタイム・コンポーネントのインストールについて詳しくは、以下の場所にある、RT-Linux RHEL 5.3 MRG 1.1.2 用のインストールの説明を参照してください: https://www.redhat.com/docs/en-US/Red_Hat_Enterprise_MRG/1.1/html/Realtime_Installation_Guide/index.html

SUSE Linux Enterprise Real Time 10

- SUSE Linux Enterprise Real Time 10 のインストールについて詳しくは、http://www.novell.com/products/realtime/eval.html を参照してください。

多数のファイル記述子を使用してさまざまなクラス・インスタンスをロードすると、「`java.util.zip.ZipException: error in opening zip file`」というエラー・メッセージ、またはファイルが開けなかったことを知らせる別の形式の IOException が表示される場合があります。これを解決するには、**`ulimit`** コマンドを使用して、ファイル記述子のプロビジョンを増やします。開けるファイルの現行の制限数を確認するには、以下のコマンドを使用します。

```
ulimit -a
```

開けるファイル数を増やすには、以下のコマンドを使用します。

```
ulimit -n 8196
```

InstallAnywhere パッケージからのインストール

これらのパッケージには、インストール・オプションの設定をガイドする対話式プログラムが含まれています。このプログラムは、グラフィカル・ユーザー・インターフェースとして実行することも、システム・コンソールから実行することもできます。

始める前に

お使いのシステムに、次の両方の共有ライブラリーがインストールされている必要があります。

- GNU C ライブラリー V2.3 (glibc)
- `libstdc++.so.5`

`libstdc++.so.5` 共有ライブラリーがない場合、インストール時に次のエラーを含む Java コア・ダンプが生成される場合があります。

```
JVMJ9VM011W Unable to load j9dmp24: libstdc++.so.5: cannot open shared object file:
No such file or directory
JVMJ9VM011W Unable to load j9gc24: libstdc++.so.5: cannot open shared object file:
No such file or directory
JVMJ9VM011W Unable to load j9vrb24: libstdc++.so.5: cannot open shared object file:
No such file or directory
```

インストール可能パッケージをインストールする場合、システムに rpm-build ツールがインストールされている必要があります。インストールされていない場合は、インストール・プログラムで新しいパッケージを RPM データベースに登録することができません。 rpm-build ツールがインストールされているかどうかを調べるには、次のコマンドを入力します。

```
rpm -q rpm-build
```

このタスクについて

InstallAnywhere のパッケージは、ファイル拡張子が `.bin` になっています。

パッケージには次の 2 つのタイプがあります。

インストール可能
: これらのパッケージをインストールすると、環境変数の設定などの、システムの構成も行われます。

アーカイブ
: これらのパッケージをシステムにインストールすると、ファイルが抽出されますが構成は一切行われません。

手順

- パッケージを対話式にインストールするには、手動インストールを実行します。
- ユーザーとの追加的な対話を行わずにパッケージをインストールするには、自動インストールを実行します。 インストールするシステムの数が多い場合には、こちらのオプションを選ぶと便利な場合があります。
- インストール・プロセスが完了したら、このセクションで説明する構成手順に従い、パスや CLASSPATH 環境変数の設定などを行ってください。

タスクの結果

製品がインストールされます。

注: Ctrl+C を押すなどのインストール・プロセスを中断する操作は行わないようにしてください。プロセスが中断されると、製品の再インストールが必要になる場合があります。詳しくは、 34 ページの『中断されたインストール』を参照してください。

インストール可能パッケージを使用している場合には、問題が発見されたことを示すメッセージが表示される可能性があります。アーカイブ・パッケージのインストール時には、メッセージは生成されません。インストール可能パッケージを使用している場合に表示されるメッセージの例を、次のリストに示します。

`The installer cannot run on your configuration. It will now quit.`
: このエラー・メッセージは、ユーザー ID がインストール・プロセス実行の権限を持っていない場合に表示されます。処理を継続できないため、インストール・プログラムは終了します。この問題を修正するには、root 権限を持つユーザー ID を使用して、再びインストールを開始してください。

`An RPM package is already installed. Uninstall the package before proceeding.`
: このメッセージは、RPM パッケージが既にインストールされていることを示します。処理を継続できないため、インストール・プログラムは終了します。この問題を修正するには、処理を先に進める前に RPM パッケージをアンインストールしてください。

手動インストールの実行

InstallAnywhere パッケージから、製品を対話式にインストールします。

始める前に

インストール・プロセスを始める前に、以下の条件を確認してください。

- 既に RPM パッケージから WebSphere Real Time for RT Linux をインストールしている場合には、手順を進める前にこのパッケージをアンインストールする必要があります。
- root 権限のあるユーザー ID を持っている必要があります。

手順

1. 一時ディレクトリーにインストール・パッケージ・ファイルをダウンロードします。
2. 一時ディレクトリーに移動します。
3. シェル・プロンプトに `./package` と入力して、インストール・プロセスを開始します。ここで、*package* はインストールするパッケージの名前です。
4. インストーラー・ウィンドウに表示されているリストから言語を選択し、「**次へ**」をクリックします。選択可能な言語のリストは、お使いのシステムのロケール設定に基づくものです。
5. 使用許諾契約書を読みます。スクロール・バーを使用して許諾契約書の文章の最後まで読んでください。インストールを先に進めるには、使用許諾契約書に同意する必要があります。条件に同意するには、ラジオ・ボタンを選択し、「**OK**」をクリックします。

   注: 使用許諾契約書の文章を最後まで読むと、使用許諾契約書に同意するラジオ・ボタンを選択することができるようになります。
6. インストールするターゲット・ディレクトリーを選択するよう求められます。デフォルトのディレクトリーにインストールしない場合は、ブラウザー・ウィンドウを使用して、「**選択 (Choose)**」をクリックし、別のディレクトリーを選択します。インストール・ディレクトリーを選択したら、「**次へ**」をクリックして手順を続行します。
7. これまでに選択した内容を再確認するよう求められます。選択内容を変更する場合は、「**前へ**」をクリックします。選択内容が正しければ、「**インストール**」をクリックしてインストールを進めます。
8. インストール・プロセスが完了したら、「**完了**」をクリックして終了します。

自動インストールの実行

インストールするシステムが複数あり、使用するインストール・オプションが決まっている場合には、自動インストール手順を使用することをお勧めします。一度手動インストール手順を使用してインストールし、その結果生成された応答ファイルを使用して、ユーザーとの追加的な対話を行わずにそれ以降のインストールを実行します。

手順

1. 手動インストールを完了させて、応答ファイルを作成します。以下の方法のいずれかを使用します。

- GUI を使用して、インストール・プログラムで応答ファイルを作成するように指定します。応答ファイルは installer.properties という名前で、インストール・ディレクトリーに作成されます。
- コマンド・ラインを使用して、手動インストールのコマンドに -r オプションを追加し、応答ファイルへの絶対パスを指定します。例えば、次のようにします。

  ```
  ./package -r /path/installer.properties
  ```

応答ファイルの内容例:

```
INSTALLER_UI=silent
USER_INSTALL_DIR=/my_directory
```

この例では、 */my_directory* がインストール時に選択したターゲット・インストール・ディレクトリーです。

2. オプション: 必要に応じて、応答ファイルを編集してオプションを変更します。

   注: アーカイブ・パッケージには、次のような既知の問題があります。応答ファイルを使用したインストールは、応答ファイル中のディレクトリーを変更した場合でも、デフォルト・ディレクトリーを使用します。デフォルト・ディレクトリーに以前のインストールが存在している場合、上書きされます。

   それぞれ異なるインストール・オプションの複数の応答ファイルを作成する場合、 *myfile*.properties という形式で、それぞれの応答ファイルに固有の名前を指定してください。

3. オプション: ログ・ファイルを生成します。 サイレント・インストールを実行しているため、インストール・プロセスの終了時に、状況メッセージが表示されることはありません。インストールの状況を記録したログ・ファイルを生成するには、以下の手順を実行します。
   a. 次のコマンドを使用して、必須のシステム・プロパティーを設定します。

      ```
      export _JAVA_OPTIONS="-Dlax.debug.level=3 -Dlax.debug.all=true"
      ```

   b. ログの出力をコンソールに送信するために、次の環境変数を設定します。

      ```
      export LAX_DEBUG=1
      ```

4. パッケージ・インストーラーを実行する際に、**-i** silent オプションを指定し、さらに **-f** オプションで応答ファイルを指定して、自動インストールを開始します。 例えば、次のようにします。

   ```
   ./package -i silent -f /path/installer.properties 1>console.txt 2>&1
   ```

   ```
   ./package -i silent -f /path/myfile.properties 1>console.txt 2>&1
   ```

   プロパティー・ファイルの指定には、完全修飾パスと相対パスのどちらでも使用できます。これらの例では、1>console.txt 2>&1 の文字列で、標準エラー出力ストリームと標準出力ストリームからのインストール・プロセスに関する情報を、現行ディレクトリーの console.txt ログ・ファイルにリダイレクトしています。インストールに問題があると思われる場合には、このログ・ファイルを調べてください。

   注: インストール・ディレクトリーに複数の応答ファイルがある場合、デフォルトの応答ファイルである installer.properties が使用されます。

中断されたインストール

パッケージ・インストーラーがインストール中に予期せず停止した場合 (例えば、ユーザーが Ctrl+C を押した場合)、そのインストールは壊れており、製品のアンインストールや再インストールはできません。アンインストールや再インストールを行おうとすると、「Fatal Application Error」というメッセージが表示される可能性があります。

このタスクについて

この問題を解決するには、以下の手順に従って、ファイルを削除して再インストールを行ってください。

手順

1. レジストリー・ファイル /var/.com.zerog.registry.xml を削除します。
2. インストールが置かれるディレクトリーが作成されている場合には、それを削除します。例: opt/IBM/javawrt3/
3. インストール・プログラムを再び実行します。

既知の問題と制約

InstallAnywhere パッケージには、いくつかの既知の問題と制約があります。

- システムに共有ライブラリー libstdc++.so.5 がない場合、インストールは失敗し、Java コア・ダンプが生成されます。詳しくは、30 ページの『InstallAnywhere パッケージからのインストール』を参照してください。
- インストール・パッケージの GUI は、Orca 画面読み上げプログラムに対応していません。 GUI の代わりとして、自動インストール・モードを使用することができます。
- インストール終了後に、 *./package* と入力して再度プログラムを開始した場合、プログラムで以下のメッセージが表示されます。

  ```
  ENTER THE NUMBER OF THE DESIRED CHOICE, OR PRESS <ENTER> TO ACCEPT THE DEFAULT:
  ```

  Enter を押してデフォルトを選択すると、プログラムは応答しなくなります。数字を入力してから、Enter を押してください。
- パッケージをインストールしてから、異なるモードで再度インストールしようとした場合 (例えば、コンソール・モードやサイレント・モード)、以下のエラー・メッセージが表示される場合があります。

  ```
  Invocation of this Java Application has caused an InvocationTargetException.
  This application will now exit
  ```

  GUI モードを使用してインストールを行った後、コンソール・モードで再びインストール・プログラムを実行した場合は、このメッセージは表示されません。アンインストール・オプションを選択するプログラムを実行していて、このエラーが表示された場合 (インストール可能パッケージのみ) は、代わりに、38 ページの『WebSphere Real Time for RT Linux のアンインストール』の説明に従って **./_uninstall/uninstall** コマンドを使用してください。

インストール可能パッケージのみに関する問題

- 既存のインストール済み環境を、InstallAnywhere パッケージを使用してアップグレードすることはできません。WebSphere Real Time for RT Linux をアップグレードするには、まず以前のバージョンをアンインストールする必要があります。
- 異なるインストール・ディレクトリーを使用する場合でも、同一システム上に WebSphere Real Time for RT Linux の同一バージョンの 2 つの異なるインスタンスを同時にインストールすることはできません。例えば、`/previous` ディレクトリーに WebSphere Real Time for RT Linux V3 を保持しながら、同時に `/current` ディレクトリーに WebSphere Real Time for RT Linux サービスを新たにインストールすることはできません。インストール・プログラムはバージョン番号をチェックします。プログラムが、同じバージョン番号の既存のパッケージを発見すると、ユーザーは既存のパッケージのアンインストールを求められます。
- パッケージがインストールされた後で、GUI を使用して再びパッケージ・インストーラーを実行した場合、パッケージのアンインストールを選択することができます。このアンインストール・オプションは、自動インストール・モードでは使用できません。自動インストール・モードで再度パッケージ・インストーラーを実行した場合、プログラムは一切動作を行いません。

アーカイブ・パッケージのみに関する問題

- 応答ファイルのインストール・ディレクトリーを変更し、その応答ファイルを使用して自動インストールを実行した場合、インストール・プログラムは新しいインストール・ディレクトリーを無視し、代わりにデフォルトのディレクトリーを使用します。デフォルト・ディレクトリーに以前のインストールが存在している場合、上書きされます。

パスの設定

PATH 環境変数を設定してある場合、シェル・プロンプトで名前を入力することにより、アプリケーションまたはプログラムを実行できます。

このタスクについて

注: このセクションの説明に従って **PATH** 環境変数を変更すると、ご使用のパスにある既存の Java 実行可能ファイルがすべてオーバーライドされます。

ツールへのパスは、その都度、ツール名の前にパスを入力して指定します。 例えば、SDK が `opt/IBM/javawrt3/` にインストールされている場合、シェル・プロンプトで以下のように入力することにより、*myfile.java* という名前のファイルをコンパイルすることができます。

```
opt/IBM/javawrt3/bin/javac myfile.java
```

毎回絶対パスを入力しないで済むようにするには、

1. ホーム・ディレクトリーにあるシェル始動ファイル (ご使用のシェルによりますが、通常は、**.bashrc**) を編集し、**PATH** 環境変数に絶対パスを追加してください。例えば、次のようにします。

   ```
   export PATH=opt/IBM/javawrt3/bin:opt/IBM/javawrt3/jre/bin:$PATH
   ```

2. 再びログオンするか、または、更新したシェル・スクリプトを実行して、新しい **PATH** 設定をアクティブにします。
3. ファイルを **javac** ツールでコンパイルします。 例えば、*myfile.java* というファイルをコンパイルするには、シェル・プロンプトで、以下のように入力します。

   ```
   javac -Xrealtime myfile.java
   ```

   PATH 環境変数を指定することにより、Linux は、どの現行ディレクトリーからでも **javac** ツール、**java** ツール、および **javadoc** ツールなどの実行可能ファイルを見つけられるようになります。ご使用のパスの現行値を表示するには、コマンド・プロンプトで次のように入力します。

   ```
   echo $PATH
   ```

次のタスク

CLASSPATH 環境変数を設定する必要があるかどうか判別するためには、『クラスパスの設定』を参照してください。

クラスパスの設定

CLASSPATH 環境変数により、SDK ツール (**java**、**javac**、および **javadoc** ツールなど) に Java クラス・ライブラリーの場所を通知します。

このタスクについて

CLASSPATH 環境変数は、以下のいずれかに該当する場合にのみ明示的に設定してください。

- 例えば、独自に作成したものなど異なるライブラリーまたはクラス・ファイルが必要であり、それが現行ディレクトリーにない場合。
- bin および lib ディレクトリーの位置を変更して、同じ親ディレクトリーが既にない場合。
- 同一システム上に、異なるランタイム環境を使用する複数のアプリケーションを作成し実行する予定の場合。

CLASSPATH の現行値を表示するには、シェル・プロンプトで次のコマンドを入力します。

```
echo $CLASSPATH
```

別にインストール済みの他のバージョンも含めて、異なるランタイム環境を使用する複数のアプリケーションを作成し実行する場合、**CLASSPATH** および **PATH** をアプリケーションごとに明示的に設定する必要があります。複数のアプリケーションを同時に実行し、異なるランタイム環境を使用する場合は、それぞれのアプリケーションを固有のシェルで実行する必要があります。

一度に 1 つのバージョンの Java のみを実行する場合は、シェル・スクリプトを使用して、異なるランタイム環境を切り替えることができます。

次のタスク

インストールが正常に行われたことを検証するには、 37 ページの『インストール済み環境のテスト』を参照してください。

インストール済み環境のテスト

インストールが正常に行われたかどうかを検査するには、**-version** オプションを使用します。

このタスクについて

Java のインストール済み環境は、標準の JVM およびリアルタイム JVM から構成されています。

手順

以下の手順に従って、インストール済み環境をテストします。

1. 標準の JVM のバージョン情報を表示するには、シェル・プロンプトで以下のコマンドを入力します。

   ```
   java -version
   ```

   このコマンドが正常に実行された場合は、以下のようなメッセージが返されます。

   ```
   java version "1.7.0"
   WebSphere Real Time V3 (build pxi3270rt-20110518_02)
   IBM J9 VM (build 2.6, JRE 1.7.0 Linux x86-32 20110516_82445 (JIT enabled,
   AOT enabled)
   J9VM - R26_head_20110515_0456_B82363
   JIT  - r11_20110510_19526
   GC   - R26_head_20110513_1009_B82250
   J9CL - 20110516_82445)
   JCL - 20110516_01 based on Oracle 7b145
   ```

   リアルタイム JVM ではなく標準の JVM を使用する場合は、IBM User Guides for Java v7 on Linux を参照してください。

   注: バージョン情報は正確ですが、日付はこの例に示されているものより後の日付である可能性があります。この日付ストリングの形式は yyyymmdd です。後ろにそのコンポーネントに固有の追加情報が続く場合もあります。

2. リアルタイム JVM のバージョン情報を表示するには、シェル・プロンプトで以下のコマンドを入力します。

   ```
   java -Xrealtime -version
   ```

   このコマンドが正常に実行された場合は、以下のようなメッセージが返されます。

   ```
   java version "1.7.0"
   WebSphere Real Time V3 (build pxi3270rt-20110518_02)
   IBM J9 VM (build 2.6, JRE 1.7.0 real-time Linux x86-32 20110516_82445 (JIT
   enabled, AOT enabled)
   J9VM - R26_head_20110515_0456_B82363
   JIT  - r11_20110510_19526
   GC   - R26_head_20110513_1009_B82250
   J9CL - 20110516_82445)
   JCL - 20110516_01 based on Oracle 7b145
   ```

注: バージョン情報は正確ですが、プラットフォーム・アーキテクチャーと日付はこの例に示されているものとは異なっている可能性があります。この日付ストリングの形式は yyyymmdd です。後ろにそのコンポーネントに固有の追加情報が続く場合もあります。

WebSphere Real Time for RT Linux のアンインストール

WebSphere Real Time for RT Linux を削除するためのプロセスは、使用したインストールのタイプによって異なります。

始める前に

InstallAnywhere インストール可能パッケージの場合は、ルート権限を持つユーザー ID が必要です。

このタスクについて

InstallAnywhere アーカイブ・パッケージにアンインストール・プロセスはありません。アーカイブ・パッケージをシステムから削除するには、パッケージのインストール時に選択したターゲット・ディレクトリーを削除してください。

InstallAnywhere インストール可能パッケージで製品をアンインストールするには、以下のステップで説明するように、コマンドを使用するか、インストール・プログラムを再度実行します。

手順

- オプション: **uninstall** コマンドを使用して手動でアンインストールします。
  1. インストールがあるディレクトリーに移動します。 例えば、次のように入力します。

     ```
     cd /opt/IBM/javawrt3
     ```

  2. コマンド ./_uninstall/uninstall を入力して、アンインストール・プロセスを開始します。
- オプション: アンインストール・プログラムが簡単に見つからない場合は、代わりに手動インストールを実行できます。インストール・プログラムでは、製品がすでにインストールされていることが検出され、その後で前のインストールをアンインストールすることができます。

第 5 章 IBM WebSphere Real Time for RT Linux アプリケーションの実行

リアルタイム・アプリケーションを実行する際に役立つ重要な情報を説明します。

- 40 ページの『WebSphere Real Time for RT Linux でのコンパイル済みコードの使用』
- 61 ページの『非ヒープ・リアルタイム・スレッドの使用』
- 70 ページの『JVM 間でのクラス・データの共用』
- 73 ページの『Metronome ガーベッジ・コレクターの使用』

スレッドのスケジューリングとディスパッチング

Linux オペレーティング・システムでは、さまざまなスケジューリング・ポリシーがサポートされています。デフォルトの一般的タイム・シェアリング・スケジューリング・ポリシーは SCHED_OTHER であり、ほとんどのスレッドで使用されます。SCHED_RR および SCHED_FIFO は、リアルタイム・アプリケーションのスレッドで使用できます。

プロセッサーで次に実行される実行可能スレッドは、カーネルが決定します。カーネルでは、実行可能スレッドのリストが保持されています。このカーネルは優先順位が最も高いスレッドを検索し、そのスレッドを次に実行するスレッドとして選択します。

スレッドの優先順位とポリシーは、以下のコマンドを使用してリストすることができます。

```
ps -emo pid,ppid,policy,tid,comm,rtprio,cputime
```

policy には以下が示されます。

- TS。SCHED_OTHER を表します。
- RR。SCHED_RR を表します。
- FF。SCHED_FIFO を表します。
- -。報告されるポリシーはありません。

出力は以下の例のようになります。

```
  PID  PPID POL   TID COMMAND         RTPRIO     TIME
18314 30285 -       - java                 - 00:01:40
    -     - RR  18314 -                    6 00:00:00
    -     - RR  18315 -                    6 00:01:40
    -     - FF  18318 -                   88 00:00:00
    -     - RR  18323 -                    6 00:00:00
    -     - FF  18324 -                   13 00:00:00
    -     - RR  18325 -                    6 00:00:00
    -     - RR  18326 -                    6 00:00:00
    -     - FF  18327 -                   11 00:00:00
    -     - FF  18328 -                   89 00:00:00
```

© Copyright IBM Corp. 2003, 2011

この出力は、Java プロセス、有効なスケジューリング・ポリシー、優先順位が「-」(その他) のメイン・スレッド、および優先順位が 11 から 89 までのいくつかのリアルタイム・スレッドを示しています。

現行のスケジューリング・ポリシーを照会する場合は、**sched_getscheduler** を使用するか、前述の例に示されている **ps** コマンドを使用してください。

プロセスについて詳しくは、 106 ページの『一般的なデバッグ手法』を参照してください。

リアルタイム Java スレッドの優先順位およびポリシー

リアルタイム・スレッド (つまり、java.realtime.RealtimeThread として割り振られたスレッド) および非同期イベント・ハンドラーは、SCHED_FIFO スケジューリング・ポリシーを使用します。

リアルタイム Java スレッドのスレッド・スケジューリングおよびディスパッチングは、Real Time Specification for Java (RTSJ) の一部です。このトピック (リアルタイム Java スレッドのスケジューリング・ポリシーおよび優先順位の取り扱いを含む) については、セクション 9 ページの『RTSJ のサポート』で説明します。

WebSphere Real Time for RT Linux でのコンパイル済みコードの使用

IBM WebSphere Real Time for RT Linux はいくつかのコード・コンパイル・モデルをサポートしており、さまざまなレベルのコード・パフォーマンスおよび決定論を提供しています。

解釈操作

これが最も単純なコード・コンパイル・モデルです。インタープリターは Java アプリケーションを実行しますが、コードのコンパイルは使用しません。インタープリターは良好な決定論を示しますが、パフォーマンスが非常に低くなるため、実動システムでこの操作モードを使用することは避けてください。

解釈操作を使用するには、Java コマンド行で **-Xint** オプションを指定します。

低優先順位の Just-In-Time (JIT) コンパイル

WebSphere Real Time for RT Linux のデフォルトのコンパイル・モデルでは、アプリケーションの実行中に、Just-In-Time コンパイラーを使用して Java アプリケーションの重要なメソッドをコンパイルします。このモードでは、JIT コンパイラーが、非リアルタイム JVM の JIT コンパイラーの動作と同様の方法で機能します。違うのは、WebSphere Real Time for RT Linux の JIT コンパイラーが、どのリアルタイム・スレッドよりも低い優先順位で実行されるという点です。優先順位が低いということは、アプリケーションがリアルタイム・タスクを実行する必要のないときに JIT コンパイラーがシステム・リソースを使用するということを意味します。結果として、JIT コンパイラーがリアルタイム・タスクのパフォーマンスに大きく影響することはありません。

JIT コンパイラーは、コンパイル関連のアクティビティー用に、コンパイル・スレッドとサンプラー・スレッドという 2 つのスレッドを使用しま

す。これらのスレッドは、リアルタイム・タスクよりも低い優先順位で実行されます。コンパイル・スレッドは、アプリケーションとは非同期で実行されます。これは、どのような場合でも、アプリケーション・スレッドは、コンパイル・スレッドによるメソッド・コンパイルの完了を待たないということを意味します。サンプラー・スレッドは、アプリケーション・スレッドに対して定期的に非同期メッセージを送信し、各スレッドで現在実行されているメソッドを特定します。アプリケーション・スレッドでのメッセージ処理時間はごく短時間です。より優先順位の高いリアルタイム・タスクがあるためにサンプリング・スレッドを実行できない場合、メッセージは送信されません。 JIT コンパイラーを使用すると決定論にわずかな影響が出ますが、多くのユーザーにとって最高のパフォーマンスが得られるのはこのコンパイル・モードです。

優先順位の低い JIT を使用してアプリケーションを実行するには、59 ページの『JIT の使用可能化』を参照してください。

Ahead-of-Time (AOT) プリコンパイル済みコード

WebSphere Real Time for RT Linux は、アプリケーションを実行する前のプリコンパイル・ステップで、Java メソッドをネイティブ・コードにコンパイルします。 WebSphere Real Time for RT Linux V2 より前のプリコンパイル・ステップでは jxeinajar ツールにより Ahead-of-Time コンパイラーを使用したメソッドのコンパイルが行われ、その結果が特殊な Java 実行可能ファイル内に格納されていました。これらのファイルは、バインドされた JAR ファイル内に収集されることがあります。アプリケーションの実行時、JXE からメソッドのクラスがロードされた際に JVM が AOT コードをロードできるように、バインド済みの JAR ファイルがアプリケーションのクラスパスに追加されます。この方法を使用すると、コマンド行で **-Xnojit** オプションを指定することにより、JIT コンパイラーが完全に使用できなくなります。アプリケーションはすでに作成されているどのプリコンパイル済み AOT コードでも使用することができ、他のメソッドについてはインタープリターを使用できます。この操作モードでは JIT コンパイラーが存在しないためにサンプリング・スレッドやコンテキスト・スイッチによるパフォーマンスの低下がなく、高い決定論が得られます。 Java の仕様に従いながら Java コードを事前にコンパイルすることは難しいため、通常、AOT コンパイルされたコードは解釈するよりははるかに高速であるものの、多くの場合、JIT コンパイルされたコードよりは実行速度が低くなります。

WebSphere Real Time for RT Linux V2 および以降のバージョンは、IBM Java 6 の JVM に用意されている共有クラス・テクノロジーを使用して、JXE ファイル内ではなく共有クラス・キャッシュ内に AOT コードを格納します。 admincache ツールでは、キャッシュの内容を照会して、既存のすべてのキャッシュをリストし、1 つのキャッシュにクラスおよび AOT コードを取り込むことができます。 AOT コンパイルされたコードを格納することの利点は、アプリケーションの JAR ファイルが変更されないことと、アプリケーションの実行時にクラスパスの変更が不要なことです。

共有クラス・キャッシュには、使用可能な仮想アドレス・スペースに基づいた実用上のサイズ制限があります。これは、すべての JAR ファイルを AOT コンパイルすることが実際的ではないことを意味します。選択的な AOT コンパイルを実行する必要があります。

共有クラス・キャッシュ内の AOT コードを使用してアプリケーションを実行すると、クラスが JVM にロードされる際に、クラスのメソッドの AOT コードが自動的にロードされます。クラスのメソッドの AOT コードをインストールするためにクラスをロードする際に発生する追加コストがあるため、アプリケーション内のパフォーマンス重視部分が実行される前に、できるだけ多くのクラスをプリロードすることが重要です。

AOT プリコンパイル済みコードを使用することで、最高レベルの決定論および良好なパフォーマンスが得られます。 AOT コードは、**-Xshareclasses** オプションおよび **-Xaot** オプションを指定してアプリケーションを実行する場合に使用できます。 **-Xaot** オプションは、デフォルトでオンになっています。

admincache ツールを使用して AOT コードを共有クラス・キャッシュに格納して使用するには、 44 ページの『admincache ツールの使用』を参照してください。 jxeinajar から admincache へのマイグレーションに関する情報については、WebSphere Real Time for RT Linux V2 資料を参照してください。

AOT コンパイルされたコードを使用してアプリケーションを実行する例については、 93 ページの『AOT を使用したサンプル・アプリケーションの実行』を参照してください。

混合モード (AOT プリコンパイル済みコードと低優先順位の JIT コンパイルの組み合わせ)

アプリケーションの実行時、AOT と JIT のコンパイル済みコードを一緒に使用することができます。この操作モードでは、非常に良好な決定論および良好なパフォーマンスが得られるほか、実行頻度の高いメソッドについては非常に高いパフォーマンスが得られます。このモードの主な利点は、アプリケーションの最も重要な部分が、通常は AOT または JIT のコンパイル済みコードよりもはるかに低速であるインタープリターで実行されることが決してないように、AOT プリコンパイルが使用されることです。 JIT コンパイラーは、実行頻度の高いすべての解釈対象メソッドを、アプリケーションのパフォーマンスを大きく低下させることなく動的に特定できるため、すべてのメソッドをプリコンパイルする必要はありません。コマンド行に **-Xshareclasses** オプションが追加された場合のデフォルト・モードは、混合モードです。

AOT コンパイルと JIT コンパイルを混用してアプリケーションを実行するには、 93 ページの『AOT を使用したサンプル・アプリケーションの実行』を参照してください。

コンパイルの明示的な管理

JIT コンパイラーを有効にしたコンパイル・モードでは、java.lang.Compiler API を使用して、JIT コンパイラーの操作を明示的に制御できます。 JIT コンパイラーは、渡されたクラスのメソッドを、compileClass() メソッドを使用してコンパイルします。 compileClass() の動作は同期的であるため、与えられたメソッドがコンパイルされるまで戻りません。アプリケーションは、アプリケーション実行時のメイン・フェーズで使用されるクラスに対して反復適用することにより、初期化フェーズで compileClass() を使用することがあります。初期化フェーズの終了時に、Compiler.disable() メソッドを呼び出して、コンパイル・スレッドおよびサンプリング・スレッドを完全に無

効にしてください。この手法の主な難点は、特にアプリケーションの開発時に、アプリケーションの初期化フェーズでロードしてコンパイルするクラスのリストを管理するという問題です。

アプリケーションでのコンパイルの管理について詳しくは、IBM Real-Time Class Analysis Tool for Java を参照してください。

コンパイルのコマンド行オプションの概要

アプリケーションは、**-Xjit** オプションを使用して JIT を有効にして実行することも、**-Xnojit** オプションを使用して JIT なしで実行することもできます。 **-Xjit** がデフォルト・モードです。

アプリケーションは、**-Xshareclasses -Xaot** オプションを使用して AOT コードを有効にして実行することができます。 AOT コードを無効にするには、**-Xnoaot** オプションを使用します。 **-Xaot** がデフォルト・オプションですが、**-Xshareclasses** オプションも指定されていない限り、効果はありません。AOT コードを共有クラス・キャッシュ内に格納する必要があるためです。

AOT コンパイラーの使用

Java コードをプリコンパイルするには、以下の手順に従ってください。この手順では、**javac** コマンドでの **-Xrealtime** オプション、**admincache** ツール、および **java** コマンドでの **-Xrealtime** オプションと **-Xnojit** オプションの使用について説明します。

このタスクについて

Ahead-of-Time コンパイラーを使用するということは、コンパイルをアプリケーションの実行時から切り離すということを意味します。また、使用頻度の高いメソッドだけに留まらず、より多くのメソッドを同時にコンパイルできます。以下の手順で示すとおり、アプリケーションにあるすべての内容をコンパイルすることも、個々のクラスだけをコンパイルすることもできます。

注: 共有クラス・キャッシュを使用する場合は、そのキャッシュの名前は 53 文字以下にする必要があります。

手順

1. シェル・プロンプトから、以下のように入力します。

   ```
   javac -Xrealtime source
   ```

   このコマンドにより、リアルタイム環境で使用するための Java バイトコードがソースから作成されます。 9 ページの図 2を参照してください。
2. 生成されたクラス・ファイルをパッケージして JAR ファイルにします。例えば、test.jar を作成するには、以下のようにします。

   ```
   jar cvf test.jar source
   ```
3. シェル・プロンプトから、以下のように入力します。

   ```
   admincache -Xrealtime -populate -aot test.jar -cacheName myCache -cp test.jar
   ```

   このコマンドにより test.jar ファイルがプリコンパイルされ、その出力が出力ディレクトリー `./aot` に書き込まれます。

4. シェル・プロンプトから、以下のように入力します。 共有クラス・キャッシュ内の AOT コードを使用してファイルを実行するには、シェル・プロンプトで以下のように入力します。

   ```
   java -Xrealtime -Xshareclasses:name=myCache -cp test.jar -Xnojit MyTestClass
   ```

   共有クラス・キャッシュ内の AOT コードを使用してファイルを実行するには、呼び出し頻度の高いメソッドを再コンパイルして、新規の JAR ファイルを作成せずにシェル・プロンプトで以下のように入力します。

   ```
   java -Xrealtime -Xshareclasses:name=myCache -cp test.jar MyTestClass
   ```

   これらのコマンドでは、ステップ 3 でプリコンパイルした同じ JAR ファイルが使用されます。

admincache ツールの使用

admincache ツールを使用して、ワークステーション上の共有クラス・キャッシュを管理します。

IBM WebSphere Real Time for RT Linux 製品では、admincache ツールを使用して、クラスまたはクラスと AOT コンパイル・コードを含む共有クラス・キャッシュを作成できます。キャッシュの作成後、このツールを使用して既存のキャッシュを調べることもできます。

共有クラス・キャッシュを使用すると、複数の JVM のシナリオでメモリー・フットプリントを削減し、アプリケーションの起動時間を短縮できます。

共有クラス・キャッシュは、WebSphere Real Time for RT Linux では、非リアルタイム・モードおよびリアルタイム・モードの両方で使用できます。ただし、キャッシュの形式、作成、およびデータの取り込みの各方法は異なります。リアルタイム・モードのキャッシュは、非リアルタイム・モードのキャッシュと互換性がありません。非リアルタイム・モードのキャッシュは、標準 JVM と同じ方法で作成され、データが取り込まれます。つまり、このキャッシュは、アプリケーションの実行時に、ユーザーには意識されずに JVM によって作成されて、データが取り込まれます。リアルタイム・モード (**-Xrealtime** オプションを指定) で、admincache を使用して共有クラス・キャッシュの作成とデータの事前取り込み (**-populate** オプション指定) を行う必要があります。リアルタイム・モードで実行するアプリケーションは、事前にデータを取り込んであるキャッシュから内容を読み取ることはできますが、内容を変更することはできません。

リアルタイム・モードで作成された共有クラス・キャッシュは、アプリケーションをリアルタイム・モードで実行している場合にのみ使用できます。非リアルタイム・モードで作成された共有クラス・キャッシュは、アプリケーションを非リアルタイム・モードで実行している場合にのみ使用できます。このことは、admincache ツールにも当てはまります。JVM でリアルタイム・モードで作成されたキャッシュを管理するには、**-Xrealtime** オプションを指定して admincache を使用します。JVM で非リアルタイム・モードで作成されたキャッシュを管理するには、**-Xrealtime** オプションを使用しないでください。実行時に共有クラス・キャッシュに接続するには、**-Xshareclasses** オプションをコマンド行に追加します。

ワークステーション上に複数の共有クラス・キャッシュを作成することができます。その場合は、各キャッシュに特定の名前を付け、特定のディレクトリーに格納

します。新しいキャッシュを作成する場合は、**-cacheName** *<name>* オプションでキャッシュの名前を指定できます。キャッシュの名前は 53 文字以下である必要があります。

共有クラス・キャッシュはデフォルトで /tmp/javasharedresources ディレクトリーに作成されますが、**-cacheDir** *<directory>* オプションを指定することで作成場所をオーバーライドできます。共有クラス・キャッシュの内部形式は、そのキャッシュが作成されているワークステーションの特性によって異なります。つまり、共有クラス・キャッシュはネットワーク・ドライブ上には作成できないということです。これは安全対策として重要です。もう 1 つの理由としては、ネットワーク・ファイル・システムから共有クラス・キャッシュにアクセスすることで、パフォーマンスが低下し、予期しない影響が生じる可能性があるからです。

コマンド行でキャッシュ名を指定しなかった場合は、デフォルトで **sharedcc_<user_login>** という名前になります。

非リアルタイム・モードでの共有キャッシュの操作について詳しくは、 101 ページの『JVM 間でのクラス・データの共有 (非リアルタイム・モード)』を参照してください。

注: IBM WebSphere Real Time for RT Linux V2 SR1 以降では、**-classpath** オプションを **-populate** オプションと組み合わせて使用する必要があります。

リアルタイム共有クラス・キャッシュの作成:

admincache ツールを使用して、リアルタイム・モードでアクセス可能な共有クラス・キャッシュを作成します。

注: 共有クラス・キャッシュ・ファイルをデフォルトの設定値で作成する場合は、セキュリティーに関する考慮事項への注意が必要です。共有クラス・キャッシュのセキュリティーに関する考慮事項、およびデフォルトの許可の変更について詳しくは、 103 ページの『共有クラス・キャッシュのセキュリティーの考慮事項』を参照してください。

共有クラス・キャッシュの作成には、admincache ツールの **-populate** オプションを使用します。このオプションは、JAR ファイルの検索対象である JAR ファイルのリスト、ディレクトリー、またはディレクトリー・ツリーと組み合わせて使用します。指定または検出された JAR ファイルごとに、admincache は共有クラス・キャッシュに JAR ファイルの各クラスを保管します。クラス・メソッドも、**-noaot** オプションを指定しない限り、AOT コンパイルされて、共有クラス・キャッシュに保管されます。

-classpath オプションを **-populate** と組み合わせて使用する必要があります。このオプションを使用しないと、以下のエラー・メッセージが表示されます。

```
-populate action requires -classpath <class path> option to be specified
```

admincache の **-help** オプションを指定すると、admincache がキャッシュにデータを取り込む方法を制御する場合に使用できるサブオプションがリストされます。

```
$ admincache -Xrealtime -help
Usage: admincache [option]*
  where [option] can be:
```

```
    -help | -?              Action: show this help
    -Xrealtime              use in real time environment
    -cacheName <name>       specify name of shared cache (Use %%u to substitute username)
    -cacheDir <dir>         set the location of the JVM cache files
    -listAllCaches          Action: list all existing shared class caches
    -printStats             Action: print cache statistics
    -printAllStats          Action: print more verbose cache statistics
    -destroy                Action: destroy the named (or default) cache
    -destroyAll             Action: destroy all caches
    -populate               Action: Create a new cache and populate it
      -searchPath <path>    specify the directory in which to find files if no files specified
                            (default is .)
                            only one -searchPath option can be specified
      -classpath <class path> specify the classpath that will be used at runtime to access this cache
                            the -classpath option is required
      -[no]recurse          [do not] recurse into subdirectories to find files to convert
                            (default do not recurse)
      -[no]grow             if specified cache exists already, [do not] add to it (default no grow)
                            if -grow is not selected, specified cache will be removed if present
      -verbose              print out progress messages for each jar
      -noisy                print out progress messages for each class in each jar
      -quiet                suppress all output
      -[no]aot              also perform AOT compilation on methods after storing classes into cache
      -aotFilter <signature>  only matching methods will be AOT compiled and stored into cache
         e.g. -aotFilter {mypackage/myclass.mymethod(I)I} compiles only mymethod(I)I
         e.g. -aotFilter {mypackage/myclass.mymethod*} compiles any mymethod
         e.g. -aotFilter {mypackage/myclass.*} compiles all methods from myclass
      -aotFilterFile <file>  only methods matching those in file will be AOT compiled and stored into
                            cache (input file must have been created by -Xjit:verbose={precompile},
                            vlog=<file>)
      -printvmargs          print VM arguments needed to access populated cache at runtime
      [jar file]*.[jar][zip]  explicit list of jar files to populate into cache
                            if no files are specified, all files.[jar][zip] in the searchPath
                            will be converted.
Exactly one action option must be specified
```

注: 共有クラス・キャッシュを使用する場合、**-cacheName** オプションで指定する名前は 53 文字以下である必要があります。

JAR ファイルのリストを指定できます。この場合は、それらの JAR ファイルのクラスのみが共有クラス・キャッシュに追加されます。JAR ファイルのリストを指定しない場合は、**-searchPath <path>** オプションを使用して、.jar ファイルまたは .zip ファイルの検索対象とするディレクトリー・ツリーを指定します。**-recurse** オプションがデフォルトです。これは、ディレクトリー・ツリーで .jar ファイルまたは .zip ファイルを再帰的に検索することを意味します。**-norecurse** オプションは、指定されたディレクトリーのみで検索を行うことを意味します。**-classpath <class path>** オプションを指定し、指定された JAR ファイルを処理するために必要なクラスを admincache がすべて検出できるようにします。共有クラス・キャッシュへのデータ取り込みの一環として、クラスが JVM にロードされます。そのため、admincache が JAR ファイルからクラスのロードを試みるときに、参照先のすべてのクラスおよびスーパークラスを検出できることが重要です。

-grow オプションは、キャッシュ・ディレクトリー内に同じ名前の既存の共有クラス・キャッシュがある場合に、新規の JAR ファイルを既存のキャッシュ・コンテンツに追加することを指定します。**-nogrow** オプションは、古いキャッシュ・ディレクトリー内に同じ名前の既存の共有クラス・キャッシュがある場合に、新しい JAR ファイルで古いキャッシュ・コンテンツを置き換えることを指定します。 **-grow** オプションは、変更されたクラスを置き換えるのではなく、共有クラス・キャッシュに存在しない新規の JAR ファイルを追加する場合に使用します。**-grow** オプション

は、キャッシュ内に既に存在するがアプリケーション変更のために変更されたクラスの更新には使用しないください。既存のクラスを更新するには、現在のクラス・コンテンツを含むまったく新しいキャッシュを作成してください。クラスの変更時に共有クラス・キャッシュを更新しなかった場合、アプリケーションは新しいクラス・コンテンツで正常に実行されますが、共有クラス・キャッシュは利用されません。これは、変更されたクラスは、共有クラス・キャッシュからではなく、ディスクからロードされるためです。ディスクからクラスをロードした場合は、そのクラスの AOT コンパイル・コードは使用できないことを意味します。クラスを変更した場合は、共有クラス・キャッシュを再生成してください。

admincache から提供される詳細のレベルを制御するには、**-quiet**、**-verbose**、および **-noisy** の各オプションを使用します。

クラス内のメソッドに対する Ahead-Of-Time (AOT) プリコンパイルを指定して、共有クラス・キャッシュにデータを取り込むには、**-aot** オプションを使用します。AOT プリコンパイルを実行せず、共有クラス・キャッシュにクラスを保管する作業だけを行うには、**-noaot** オプションを使用します。**-aot** オプションがデフォルト設定です。

一部のメソッドを選択してプリコンパイルするには、**-aotFilter** <*signature*> オプションまたは **-aotFilterFile** <*file*> オプションを使用します。<*signature*> は、メソッド・シグニチャーの簡易正規表現であり、中括弧で囲みます。ここで、任意の文字シーケンスの代わりに '*' を使用できます。シェルがメソッド・シグニチャーの文字を解釈しないようにするには、<*signature*> を単一引用符で囲む必要があります。

表 4に、<*signature*> オプションの例をいくつか示します。

表 4. <signature> オプションの例

シグニチャー	意味
`-aotFilter '{java/lang/*}'`	AOT が java/lang パッケージ内のメソッドをコンパイルします。
`-aotFilter '{*.sample*}'`	AOT が "sample" から始まるメソッドをコンパイルします。
`-aotFilter '{mypackage/myclass.mymethod (I)I}'`	AOT はこのシグニチャーを正確に持つメソッドをコンパイルします。

-aotFilterFile <*file*> オプションでは、<*file*> の内容を使用して、AOT コンパイル対象のメソッドを選択します。他のメソッドは AOT コンパイルされません。<*file*> の内容は、**-Xjit:verbose={precompile},vlog=**<*file*> オプションを使用して、アプリケーションの以前の実行時に生成されたものです。<*file*> に保管される詳細出力では、内部形式が使用されます。**-aotFilterFile** オプションでは、この形式が必須です。

注: -vlog=<*file*> オプションは、"file" というファイルを直接生成するわけではありません。詳細出力の生成時に、日付とプロセス ID のストリングが "file" に付加されます。**-Xjit:verbose={precompile},vlog=my_file** オプションを指定すると、生成されるファイル名は `my_file.<date>.<#>.<process id>` というようになります。コマンド行オプションを 1 つの特定の JVM に指定したり、JVM ごとに異なる

-Xjit コマンド行オプションを指定したりすることが難しい場合がある複数の JVM から成るシナリオでは、こうした追加フィールドにより、個々の詳細ログ・ファイルを生成することが簡単になります。単一の JVM のシナリオでは、コマンド行に指定されたファイル名にこれらの数値が付加されます。

生成されたファイルは、編集することなく **-aotFilterFile** オプションで使用できます。**-Xjit:verbose={precompile},vlog=<file>** オプションを使用してアプリケーションを複数回実行することで生成された複数の詳細ログ・ファイルは、**-aotFilterFile** オプションを使用することで、連結して admincache に提供できます。

-printvmargs オプションを使用すると、アプリケーションの実行時に適正な引数がコマンド行に指定されるようにできます。

```
$ admincache -Xrealtime -classpath myapp.jar -cacheDir myCacheDir -cacheName
myCache -populate myapp.jar -printvmargs

admincache 1.02
Converting files
Processing classes in /team/triage/180724/bin/myapp.jar into shared class cache
No errors while processing jar file /team/triage/180724/bin/myapp.jar

Processing complete

VM args needed at runtime: -Xshareclasses:name=myCache,cacheDir=/tmp/peter
-classpath myapp.jar -Xaot
```

この例では、出力の最後の行に、共有クラス・キャッシュに保管されているクラスおよび AOT メソッドを使用できるように、アプリケーションの実行時にコマンド行に追加すべきオプションが示されています。この例に示されているオプションを使用するには、以下のコマンドを入力します。

```
java -Xshareclasses:name=myCache,cacheDir=myCacheDir -classpath myapp.jar -Xaot
myMainClass <application arguments>
```

admincache による共有クラス・キャッシュの管理:

admincache ツールには、システム上の共有クラス・キャッシュを管理するためのユーティリティーがいくつか含まれています。

admincache ツールには、いくつかのアクティビティーに役立つユーティリティーが用意されています。

- キャッシュ内にある共有クラス・キャッシュをリストします。
- 共有クラス・キャッシュの内容に関する詳細を提供します。
- 特定のキャッシュ・ディレクトリー内のキャッシュを、部分的にまたはすべて削除します。

使用可能な共有クラス・キャッシュのリスト:

admincache ツールは、キャッシュ内に存在する共有クラス・キャッシュのリストを提供します。

キャッシュにあるすべての共有クラス・キャッシュのリストを取得するには、**-listAllCaches** オプションを使用し、**-cacheDir** オプションを使用してキャッシュ・ディレクトリーを指定します。

```
$ admincache -Xrealtime -listAllCaches

admincache 1.02

Listing all caches in cacheDir /tmp/javasharedresources/

Cache name              level         persistent  last detach time

Compatible shared caches
sharedcc_username       Java6 32-bit  yes         Thu Oct 16 17:02:39 2008
rtCache                 Java6 32-bit  yes         Thu Oct 16 17:03:12 2008

Incompatible shared caches
nonrtCache              Java6 32-bit  yes         Thu Oct 16 17:17:32 2008
```

この例では、デフォルトのキャッシュ・ディレクトリーに、以下の 2 つの互換性のある共有クラス・キャッシュがあります。

- *username* でログインしているユーザーに対するデフォルトの名前付きキャッシュ
- *rtCache* というもう 1 つのキャッシュ

また、この例では、*nonrtCache* という互換性のないキャッシュも示されています。*nonrtCache* は、JVM が非リアルタイム・モードで実行されているときに作成されたキャッシュです。つまり、このキャッシュには **-Xrealtime** オプションを使用してアクセスすることはできません。

リアルタイム・モードの JVM では、非リアルタイム・モードで作成されたキャッシュを表示できます。非リアルタイム・モードの JVM では、リアルタイム・モードで作成されたキャッシュを表示できません。

```
$ admincache -listAllCaches
J9 Java(TM) admincache 1.0
Licensed Materials - Property of IBM

(c) Copyright IBM Corp. 1991, 2008  All Rights Reserved
IBM is a registered trademark of IBM Corp.
Java and all Java-based marks and logos are trademarks or registered
trademarks of Oracle Corporation


Listing all caches in cacheDir /tmp/javasharedresources/

Cache name              level         persistent  last detach time

Compatible shared caches
nonrtCache              Java6 32-bit  yes         Thu Oct 16 17:17:32 2008
```

この例では、*nonrtCache* がリストされています。これは、**-Xrealtime** が指定されていないため、互換性ありとして表示されています。

共有クラス・キャッシュの内容の検査:

admincache ツールは、共有クラス・キャッシュの内容を記述します。

admincache ツールの **-printStats** オプションを使用すると、共有クラス・キャッシュの主な内容が記述された概要を取得できます。特定のキャッシュ・ディレクトリー内の特定のキャッシュに関する情報を取得するには、**-cacheName** オプションおよび **-cacheDir** オプションを使用します。以下の例では、デフォルトのキャッシュ・ディレクトリー内の *nonrtCache* キャッシュに関する情報が提供されています。

```
$ admincache -cacheName nonrtCache -printStats

admincache 1.02

Current statistics for cache "nonrtCache":


base address       = 0xD5445000
end address        = 0xD6437000
allocation pointer = 0xD5529FA8

cache size         = 16776852
free bytes         = 14070360
ROMClass bytes     = 1166004
AOT bytes          = 1437412
Data bytes         = 57440
Metadata bytes     = 45636
Metadata % used    = 1%

# ROMClasses       = 372
# AOT Methods      = 981
# Classpaths       = 1
# URLs             = 0
# Tokens           = 0
# Stale classes    = 0
% Stale classes    = 0%

Cache is 16% full
```

注: 共有クラス・キャッシュを使用する場合は、そのキャッシュの名前は 53 文字以下にする必要があります。

ここには、このキャッシュに関する役立つさまざまな情報が示されています。

- キャッシュのサイズ。`cache size = 16776852` として示されます。
- キャッシュ内の使用可能なスペース。`free bytes = 14070360` として示されています。キャッシュ使用率が約 16% であることがわかります。
- キャッシュに保管されているクラスの数。`# ROMClasses = 372` として示されます。
- キャッシュに保管されている AOT メソッドの数。`# AOT Methods = 981` として示されます。

admincache ツールの **-printStats** オプションで提供される情報について詳しくは、printStats ユーティリティーを参照してください。

-printAllStats オプションでは、共有クラス・キャッシュの内容の詳細な説明が提供されます。この情報には、キャッシュに保管されているクラスおよび AOT メソッドのリストが含まれます。**-printAllStats** オプションからの出力は詳細です。

キャッシュに含まれるクラスは、以下のような行で示されます。

```
1: 0xD643B788 ROMCLASS: java/lang/ClassLoader at 0xD5469B88.
```

この行は、java/lang/ClassLoader クラスがキャッシュに含まれていることを示しています。このアドレスは、共有クラス・キャッシュの内部アドレスであり、診断目的に使用する以外はほとんど意味を持ちません。

キャッシュに含まれる AOT メソッドは、以下のような行で示されます。

```
1: 0xD643B290 AOT: callerClassLoader
       for ROMClass java/lang/ClassLoader at 0xD5469B88.
```

これらの行は、java/lang/ClassLoader クラスの callerClassLoader メソッドがキャッシュに含まれていることを示しています。表示されるアドレスは、共有キャッシュの内部アドレスです。**-printAllStats** オプションからの出力には、キャッシュ内の各 AOT メソッドのシグニチャーは表示されません (シグニチャーはパラメーターの型と戻りの型で構成されます)。

admincache ツールの **-printAllStats** オプションで提供される情報について詳しくは、printAllStats ユーティリティーを参照してください。

共有クラス・キャッシュの破棄:

admincache ツールには、指定されたキャッシュ・ディレクトリー内の特定のキャッシュまたはすべてのキャッシュを削除するオプションがあります。

特定のキャッシュ・ディレクトリー内の特定のキャッシュを削除するには (ユーザーがそのための許可を持っている場合)、admincache ツールの **-destroy** オプションを使用します。すべてのキャッシュを削除するには (ユーザーがそのための許可を持っている場合)、**-destroyAll** オプションを使用します。例えば、次のようにします。

```
$ admincache -Xrealtime -destroy

admincache 1.02

JVMSHRC256I Persistent shared cache "sharedcc_username" has been destroyed
```

キャッシュを削除すると、デフォルトのキャッシュ・ディレクトリーに含まれる使用可能な共有クラス・キャッシュのリストに、削除したキャッシュが表示されなくなります。

```
$ admincache -Xrealtime -listAllCaches

admincache 1.02

Listing all caches in cacheDir /tmp/javasharedresources/

Cache name           level         persistent  last detach time

Compatible shared caches
rtCache              Java6 32-bit  yes         Thu Oct 16 17:03:12 2008

Incompatible shared caches
nonrtCache           Java6 32-bit  yes         Thu Oct 16 17:17:32 2008
```

-destroyAll オプションを指定すると、現行の JVM との互換性にかかわらず、指定されたキャッシュ・ディレクトリー内のすべてのキャッシュが削除されます。**-destroyAll** オプションを使用する場合は、十分に注意してください。

```
$ admincache -Xrealtime -destroyAll

admincache 1.02

Attempting to destroy all caches in cacheDir /tmp/javasharedresources/

JVMSHRC256I Persistent shared cache "rtCache" has been destroyed
JVMSHRC256I Persistent shared cache "nonrtCache" has been destroyed
```

この結果、マシン上で使用可能な共有クラス・キャッシュはなくなります。

```
$ admincache -Xrealtime -listAllCaches

admincache 1.02

JVMSHRC005I No shared class caches available
```

キャッシュにアクセスする許可を現行ユーザーが持っていない場合は、**-destroy** オプションを指定しても **-destroyAll** オプションを指定しても、キャッシュは破棄されません。

共有クラス・キャッシュの実質的なサイズ:

admincache ツールは、共有クラス・キャッシュのサイズ変更に関する情報を提供します。

小さいアプリケーションでは、作成するキャッシュのサイズを非常に大きくすることなく、共有クラス・キャッシュにすべてのクラスとメソッドを取り込むことができます。大きいアプリケーションでは、作成される共有クラス・キャッシュのサイズが実質的に大きくなり過ぎることがあります。これは、共有クラス・キャッシュの内容全体に対応できる十分な大きさの仮想アドレス・スペースが、JVM プロセスに必要であるためです。共有クラス・キャッシュ・テクノロジーを使用する場合に当てはまる考慮事項がいくつかあります。

共有クラス・キャッシュでは、そのキャッシュに接続するすべての JVM で、キャッシュ全体が事実上アドレス可能である必要があります。これは、700 MB を超える共有クラス・キャッシュの使用を避けることを意味しています。admincache ツールでは、キャッシュのサイズを見積もることができます。このツールによって、キャッシュが 700 MB の限度よりも大きいことが示された場合は、保管するクラスの数を減らすか、またはキャッシュに保管する AOT メソッドを絞り込むようアドバイスするメッセージが表示されます。

```
$ admincache -Xrealtime -populate veryBigJar.jar -cp <my class path>

admincache 1.02

WARNING: predicted cache size (15960MB) exceeds recommended maximum shared class cache size of 700MB
If your jar files contain primarily class files then you may not be able to create a cache of this size
        or you may not be able to connect to the created cache when you run your application.
Alternatively, you may want to more selectively compile AOT methods by using -aotFilterFile
To override this warning message, please directly specify -Xscmx15960M on your command-line
        but beware that the resulting failure may not occur until the very end of the population
        procedure.
```

admincache ツールは、取り込むために指定または検出された JAR ファイルの合計サイズに基づいて、キャッシュ・サイズを控えめに見積もります。つまり、クラス・ファイル以外のファイルが JAR ファイルに数多く含まれている場合、この見積もりは正確でない可能性があります。キャッシュ・サイズのより正確な見積もりを取得するには、クラス・ファイルのみを含む一時的なバージョンの JAR ファイルを作成します。引き続き admincache ツールから警告メッセージが出る場合は、JAR ファイル内の AOT プリコンパイル対象のメソッドを絞り込むことを検討してください。この場合は、**-aotFilter** <*pattern*> オプションまたは **-aotFilterFile** <*file*> オプションを使用します。admincache ツールのメッセージから、これらのオプションで除外された AOT メソッドはサイズの見積もりの計算に含まれないことが分かります。

警告メッセージをオーバーライドして、キャッシュ読み込みステップに進むには、指示された **-Xscmx** オプションを admincache コマンド行に追加します。見積もりサイズが非常に大きいと、要求されたサイズの共有クラス・キャッシュを admincache ツールで作成できない可能性があります。これを解決するには、admincache ツールが続行できるようになるまでキャッシュ・サイズを小さくしてください。

ディスクに書き込まれる最終的なキャッシュのサイズは、指定されたクラスおよび AOT メソッドを保持するのに必要な大きさによってのみ決まります。つまり、初期キャッシュ・サイズを大きめに指定しても問題はありません。

共有クラス・キャッシュでの SDK クラスの保管:

SDK の JAR ファイルをすべて含むキャッシュの作成は、必ずしもすべてのアプリケーションで必要になるとは限りません。

SDK 内の JAR ファイルの数およびサイズによっては、これらの JAR ファイルをすべて含むキャッシュを作成しようとすると、作成されるキャッシュが大きすぎることを知らせる警告メッセージが表示されます。多くのアプリケーションでは、SDK の JAR ファイルの大半は参照されません。

SDK の主な JAR ファイルは `SDK/jre/lib` ディレクトリーに置かれています。ほとんどのアプリケーションにおいて、JAR ファイルの中で最も重要なファイルは `rt.jar` です。これは、Java 6 リリースでの新規ファイルです。Java 6 リリースより前では別の JAR ファイルに格納されていたクラスのコレクションが、`rt.jar` です。共有クラス・キャッシュに `rt.jar` のみを取り込み、AOT コンパイラーでそのメソッドをすべてコンパイルすると、約 300 MB のサイズのキャッシュが作成されます。`rt.jar` クラスのメソッドの大半は、通常のアプリケーションでは参照されません。共有クラス・キャッシュに `rt.jar` を取り込むには、以下の手順を実行します。

1. `rt.jar` のクラスのみを共有クラス・キャッシュに取り込みます。これで、キャッシュのスペースのうち、約 50 MB が消費されます。
2. **-aotFilterFile** <*file*> オプションを使用して、プログラムが使用する可能性があるメソッドのみをコンパイルします。アプリケーションを実行することによって、<*file*> を生成できます。

SDK には、他にもよく使用される重要な JAR ファイルとして、以下のものがあります。

- `sdk/jre/lib/i386/realtime/jclSC160/realtime.jar`
- `sdk/jre/lib/i386/realtime/jclSC160/vm.jar`
- `sdk/jre/lib/java.util.jar`

`realtime.jar` には、Real Time Specification for Java (RTSJ) の IBM 実装が含まれています。アプリケーションが RTSJ の機能のいずれかを使用する場合は、`realtime.jar` ファイルを共有クラス・キャッシュに保管することで、決定論的なパフォーマンスを向上させることができます。`vm.jar` には、すべてのアプリケーションで一般的に使用されている内部 JVM クラスがいくつか含まれています。`java.util.jar` には、複数のコンテナー・クラスが含まれているため、決定論的なパフォーマンスが向上するように、すべてのアプリケーションの共有クラス・キャッシュに保管する必要があります。

sdk/jre/lib ディレクトリーおよび sdk/jre/lib/ext ディレクトリー内のその他の JAR ファイルは、アプリケーションがそれらのクラスを使用する場合に共有クラス・キャッシュに保管できます。アプリケーションがこれらのクラスを使用するかどうかを識別する最も簡単な方法は、プログラムの実行時に **-verbose:dynload** オプションを使用することです。**-verbose:dynload** オプションは、アプリケーションの現在の実行によってロードされたクラスのみを示します。例えば、次のようになります。

```
<Loaded java/io/InputStreamReader from /myjdk/sdk/jre/lib/rt.jar>
<  Class size 2126; ROM size 2280; debug size 0>
<  Read time 54 usec; Load time 47 usec; Translate time 86 usec>
<Loaded java/util/LinkedHashSet from /myjdk/sdk/jre/lib/java.util.jar>
<  Class size 1218; ROM size 1136; debug size 0>
<  Read time 48 usec; Load time 31 usec; Translate time 55 usec>
<Loaded java/util/HashSet from /myjdk/sdk/jre/lib/java.util.jar>
<  Class size 3171; ROM size 2664; debug size 0>
<  Read time 71 usec; Load time 70 usec; Translate time 118 usec>
```

この出力例では、2 つの異なる SDK JAR ファイルからロードされた 3 つのクラスが示されています。java/io/InputStreamReader クラスは rt.jar からロードされました。java/util/LinkedHashSet クラスおよび java/util/HashSet クラスは、java.util.jar からロードされました。

admincache に関するその他の考慮事項:

admincache を使用する場合に役立つ情報を紹介します。

キャッシュへのデータの取り込みと永久メモリーのサイズ変更

admincache ツールで共有クラス・キャッシュにデータをリアルタイム・モードで取り込む場合は、取り込みのプロセスを通じて各クラスを読み込む必要があります。それぞれのクラスは永久メモリーを消費するため、永久メモリーのデフォルトのサイズでは、要求されたすべてのクラスに対応できない可能性があります。**admincache** ツールは、キャッシュに取り込むクラスが多いと、OutOfMemory エラーをスローし、**-Xgc:immortalMemorySize=32M** オプションを使用して、永久メモリーのサイズをデフォルトの 16 MB より大きくすることを試みます。

クラスを変更する場合

ディスク上でクラス・ファイルが変更されると、共有クラス・キャッシュ・テクノロジーは、共有クラス・キャッシュに保管されているそのクラスのキャッシュ・バージョンを使用してはならないことを自動的に検出します。プログラムは正しく動作しますが、共有クラス・キャッシュを完全には利用できなくなり、そのクラスの AOT メソッドは使用されません。アプリケーションでクラスを変更した場合は、共有クラス・キャッシュを再作成してください。**-grow** オプションを使用して、変更されたクラスを含む JAR ファイルのみを取り込み直さないようにしてください。このオプションは、JAR ファイルがキャッシュ内に既に存在するシナリオに対応するように設計されていません。

共有キャッシュの管理

ロードされたファイルがない場合でも、共有キャッシュにはアドレス・スペースが必要です。共有クラス・キャッシュが JVM プロセスでメモリーをどう消費するか

について詳しくは、 114 ページの『IBM JVM によるメモリーの管理方法』を参照してください。

プリコンパイルされた JAR ファイルの共有クラス・キャッシュへの格納

共有クラス・キャッシュには、IBM によって提供された Java クラスのすべてまたは一部を格納したり、組み込んだりすることができます。このプロセスでは、**-Xrealtime** オプションを指定した **javac**、および **admincache** ツールを使用して、共有クラス・キャッシュにクラスを格納します。

始める前に

事前に共有クラス・キャッシュに格納される JAR ファイルは、**-Xrealtime** オプションを使用する場合、および **-Xrealtime** オプションを指定して Java を実行する場合にのみサポートされます。これらの同じ JAR ファイルは実行時の **-Xrealtime** の有無に関わらず使用できますが、キャッシュに格納された JAR ファイルは、**-Xrealtime** が指定されている場合にのみ使用できます。

注: 共有クラス・キャッシュを使用する際、そのキャッシュの名前は 53 文字以下にする必要があります。

このタスクについて

admincache ツールを使用して、JAR ファイルを共有クラス・キャッシュに格納できます。 **admincache** では、3 とおりの中のいずれかの方法でアプリケーションをビルドできます。

注:

- Linux システムでタイムアウトが設定されている場合、大きな JAR ファイルをプリコンパイルする場合には、その設定をオーバーライドする必要があります。オーバーライドしないと、コンパイルがタイムアウトになり、JAR ファイルが作成されません。

アプリケーション内のすべてのクラスおよびメソッドのプリコンパイル:

この手順では、アプリケーション内のすべてのクラスをプリコンパイルします。JAR ファイル・セットを共有クラス・キャッシュに格納します。これらの JAR ファイルにあるすべてのクラスのすべてのメソッドが、キャッシュ内に格納されます。最適化された JAR ファイルでは、すべてのメソッドがコンパイルされています。

このタスクについて

この例の目的から、アプリケーションは環境変数 *$APP_HOME* が指定するディレクトリーに、JAR ファイルはそのサブディレクトリーである *$APP_HOME*/lib にあります。アプリケーションでは、IBM が core.jarrt.jar 内で提供しているクラスの一部を使用することもできます。その場合は、アプリケーション・コード、つまり、main.jar および util.jar のみをプリコンパイルできます。

共有クラス・キャッシュは、デフォルトでは /tmp/javasharedresources にあります。このキャッシュを別のディレクトリーに置くには、**-cacheDir** オプションを使

用します。ネットワーク・ファイル・システム上にキャッシュを作成することはできません。

手順

1. シェル・プロンプトから、以下のように入力します。 `cd $APP_HOME`

   ここで、*$APP_HOME* はアプリケーションのディレクトリーです。
2. シェル・プロンプトから `cd $APP_HOME/lib` と入力します。 `$APP_HOME/lib` は、`main.jar` および `util.jar` が保管されているディレクトリーです。
3. シェル・プロンプトから `admincache -Xrealtime -populate -aot -classpath $APP_HOME/lib -searchPath $APP_HOME/lib -norecurse` と入力します。 この手順により、`$APP_HOME/lib` にある各 JAR ファイルが最適化されます。進行状況情報が画面に表示され、`$APP_HOME/aot` ディレクトリーに新規の JAR ファイルが作成されます。 **-cacheName <name>** を使用してキャッシュ名を指定できますが、指定がない場合には、ユーザーのログインに基づいたデフォルト名が使用されます。

   注: -cacheName オプションで指定する名前は、53 文字以下である必要があります。
4. シェル・プロンプトから `admincache -Xrealtime -listAllCaches` と入力すると、存在するキャッシュが表示されます。

次のタスク

そのほかのオプションについては、`admincache -Xrealtime -help` と指定してください。

使用頻度の高いメソッドのプリコンパイル:

プロファイルを対象とした AOT コンパイルを使用することで、アプリケーションが頻繁に使用するメソッドのみをプリコンパイルできます。 AOT コンパイルでは、**-Xjit:verbose={precompile},vlog=optFile** という特殊なオプションを指定してアプリケーションを実行することにより生成されたオプション・ファイルを使用して、JAR ファイル・セットを共有クラス・キャッシュに格納します。オプション・ファイルにリストされているメソッドのみがプリコンパイルされます。

始める前に

作業を開始する前に、通常は JIT (Just-In-Time) コンパイラーでコンパイルされるメソッドのリストを作成してください。

このタスクについて

-Xjit:verbose={precompile} オプションで生成されたファイルは編集できます。このファイルには、プリコンパイルされるメソッドが明示的に指定されています。これらのメソッドは固有のものです。つまり、これらのメソッドには、コンパイルされる各メソッドに対する完全な署名が含まれています。それにより、`com/acme/sample.myMethod(J)V` はコンパイルされますが、`com/acme/sample.myMethod(I)V` はコンパイルされません。

注: 共有クラス・キャッシュを使用する場合は、そのキャッシュの名前は 53 文字以下にする必要があります。

手順

1. シェル・プロンプトから、以下のように入力します。

   ```
   cd $APP_HOME
   ```

   ここで、*$APP_HOME* はアプリケーションのディレクトリーです。
2. シェル・プロンプトから、以下のように入力します。

```
java -Xjit:verbose={precompile},vlog=$APP_HOME/app.precompileOpts ¥
     -cp $APP_HOME/lib/demo.jar applicationName
```

   ここで、
   - *app.precompileOpts* は、JIT を使用してコンパイルされるメソッドをリストしたログ・ファイルの名前です。
   - *applicationName* はアプリケーションの名前です。

   このコマンドにより、JIT を使用してコンパイルされるメソッドのリストが作成されます。
3. シェル・プロンプトから、以下のように入力します。

   ```
   cd $APP_HOME/lib
   ```

   $APP_HOME/lib は、アプリケーションの JAR ファイルが保管されているディレクトリーです。
4. すべてのサンプル・アプリケーション・メソッドをコンパイルしてキャッシュに入れるには、以下のように入力します。

   ```
   admincache -Xrealtime -populate -cacheName myCache ¥
              -aotFilterFile $APP_HOME/app.precompileOpts ¥
              -cp $APP_HOME/lib/demo.jar
   ```
5. `realtime.jar` および `vm.jar` をコンパイルしてキャッシュに入れるには、以下のように入力します。

```
admincache -Xrealtime -populate -grow -cacheName myCache ¥
           -aotFilterFile $APP_HOME/app.precompileOpts ¥
           -searchPath $JAVA_HOME/jre/bin/realtime/jclSC160
           -cp $APP_HOME/lib/demo.jar
```

6. `rt.jar` をコンパイルしてキャッシュに入れるには、以下のように入力します。

```
admincache -Xrealtime -populate -grow -cacheName myCache ¥
           -aotFilterFile $APP_HOME/app.precompileOpts ¥
           $JAVA_HOME/jre/lib/rt.jar
           -cp $APP_HOME/lib/demo.jar
```

7. このコマンドをテストするには、**-nojit** オプションを指定してアプリケーションを実行します。キャッシュ内のコードが使用されます。シェル・プロンプトから、以下のように入力します。

```
java -Xrealtime -Xshareclasses:name=myCache -Xnojit ¥
     -cp $APPHOME/aot/demo.jar applicationName
```

   ここで、*applicationName* はアプリケーションの名前です。

IBM により提供されたファイルのプリコンパイル:

rt.jar など、IBM によって提供されたファイルをプリコンパイルすることで、パフォーマンスと予測可能性の間の妥協点を見つけることができます。

このタスクについて

このプリコンパイルは、アプリケーションの JAR をプリコンパイルする作業と似ていますが、実行時に追加の要件が適用されるため、JRE 内のファイルではなくこれらのファイルが使用されるように、ブート・クラスパスを正しく指定する必要があります。これは **-Xshareclasses** オプションを使用して行うことができます。このオプションにより、デフォルトのクラスパスの場所より先に、指定されたクラス・キャッシュを最初に参照するように JVM に指示が出されます。

注: 共有クラス・キャッシュを使用する場合は、そのキャッシュの名前は 53 文字以下にする必要があります。

アプリケーションで使用する rt.jar を、以下のようにしてプリコンパイルします。

手順

1. シェル・プロンプトから、以下のように入力します。 cd *$JAVA_HOME*/lib ここで、*$JAVA_HOME* は Java ホーム・ディレクトリーです。
2. **admincache** ツールを実行します。シェル・プロンプトで、次のように入力します。
   ```
   admincache -Xrealtime -populate -cacheName myCache
       -classpath <class path> rt.jar
   ```
   このコマンドにより、IBM 提供の rt.jar というファイルをプリコンパイルした結果が、myCache というキャッシュに取り込まれます。
3. キャッシュ名を指定するための **-Xshareclasses** オプションを指定して、アプリケーションを実行します。アプリケーションを実行するには、以下のように入力します。
   ```
   java -Xrealtime -Xnojit -Xshareclasses:name=myCache
           -classpath:$APP_HOME/main.jar:$APP_HOME/util.jar ...
   ```

Just-In-Time (JIT) コンパイラー

標準 SDK クラス・ライブラリーの一部として提供される java.lang.Compiler クラスを使用して、JIT コンパイラーがいつ、どのように作動するのかを制御できます。IBM は、Compile.compileClass() メソッド、Compiler.enable() メソッド、および Compiler.disable() メソッドを完全にサポートします。

例えば、アプリケーションのウォームアップを行う場合にそのアプリケーション内の主要なメソッドがコンパイル済みであることが分かっているときには、アプリケーションのウォームアップ後に Compiler.disable() メソッドを呼び出して、そのアプリケーションのそれ以降の実行中に JIT コンパイルが行われないようにする必要があります。

メソッドのコンパイルは、以下の 2 とおりの方法で制御できます。

- コンパイル可能な一連のメソッドを指定します。
  ```
  Compiler.command("{<method specification>}(compile)");
  ```

ここで、*<method specification>* は、この時点でロードされていて、コンパイルされる予定になっている、すべてのメソッドのリストです。 *<method specification>* は完全修飾されたメソッド名を表しています。アスタリスクは、ワイルドカード突き合わせを表します。

例えば、既にロードされている、java.lang.String で始まるすべてのメソッドをコンパイルするには、次のように指定します。

```
Compiler.command("{java.lang.String*}(compile)");
```

注: このコマンドを使用すると、java.lang.String クラスに属するメソッドのみでなく、java.lang.StringBuffer クラスに属する、コンパイルを希望しないメソッドもコンパイルされます。 java.lang.String クラスに属するメソッドのみをコンパイルするには、次のように指定します。

```
Compiler.command("{java.lang.String.*}(compile)");
```

- このスレッドが実行されて続行される前にコンパイル・キューにあるすべてのメソッドをコンパイルするように指定します。

  ```
  Compiler.command("waitOnCompilationQueue");
  ```

  コンパイラーを使用不可にする前にコンパイル・キューが空になったことを確認しなければならないことがあります。一連のメソッドおよびクラスをコンパイルするための代表的な手法は、次のとおりです。

```
Compiler.enable();                              // コンパイラーがアクティブであることを確認
Compiler.command("{com.mycompany.*}(compile)"); // コンパイル対象のすべてのメソッドをキューに入れる
Compiler.command("waitOnCompilationQueue");     // 対象メソッドがすべてコンパイルされるまで待機
Compiler.disable();                             // コンパイラーをオフにする
```

JNI 遷移中の決定論

デフォルトでは、JIT は、Java からネイティブ JNI への (J2N) 高性能な遷移が行われるように最適化されたコードを生成します。以下のコード・シーケンスを使用してネイティブ・ライブラリーを再ロードするときに、決定論的な減少が発生する可能性があります。

```
RegisterNatives / UnregisterNatives / RegisterNatives
```

より低速で決定論的なコードに戻すには、コマンド行オプション **-Xjit:disableDirectToJNI** を使用してください。

JIT の使用可能化

JIT は、いくつかの方法で明示的に使用可能にすることができます。どちらのコマンド行オプションも、**JAVA_COMPILER** 環境変数をオーバーライドします。

手順

- Java アプリケーションを実行する前に、**JAVA_COMPILER** 環境変数を「jitc」に設定します。 シェル・プロンプトで、次のように入力します。
  - **Korn シェルの場合:** `export JAVA_COMPILER=jitc`

    注: 本書では、特に明記しない限り Korn シェル・コマンドを使用します。
  - **Bourne シェルの場合:**

    ```
    JAVA_COMPILER=jitc
    export JAVA_COMPILER
    ```

– **C シェルの場合:** `setenv JAVA_COMPILER jitc`

JAVA_COMPILER 環境変数が空ストリングの場合、JIT は使用不可のままになります。環境変数を使用不可に設定するには、シェル・プロンプトで unset **JAVA_COMPILER** と入力します。

- java.compiler プロパティーを「jitc」に設定するために、JVM コマンド行で **-D** オプションを指定します。 シェル・プロンプトで、java -Djava.compiler=jitc <*MyApp*> と入力します。
- JVM コマンド行で **-Xjit** オプションを使用します。 **-Xint** オプションを同時に指定してはなりません。 シェル・プロンプトで、java -Xjit <*MyApp*> と入力します。

JIT の使用不可化

JIT は、いくつかの方法で使用不可にすることができます。どちらのコマンド行オプションも、**JAVA_COMPILER** 環境変数をオーバーライドします。

このタスクについて

手順

- Java アプリケーションを実行する前に、**JAVA_COMPILER** 環境変数を「NONE」または空ストリングに設定します。 シェル・プロンプトで、以下のように入力します。
  - **Korn シェルの場合:** `export JAVA_COMPILER=NONE`

    注: 本書のこれ以降の部分では、Korn シェル・コマンドを使用します。
  - **Bourne シェルの場合:**
    ```
    JAVA_COMPILER=NONE
    export JAVA_COMPILER
    ```
  - **C シェルの場合:** `setenv JAVA_COMPILER NONE`
- JVM コマンド行で **-D** オプションを使用して、 java.compiler プロパティーを「NONE」または空ストリングに設定します。 シェル・プロンプトで、java -Djava.compiler=NONE <*MyApp*> と入力します。
- JVM コマンド行で **-Xint** オプションを使用します。 シェル・プロンプトで、java -Xint <*MyApp*> と入力します。

JIT が使用可能かどうかを判別する

JIT の状況は **-version** オプションを使用して判別できます。

手順

シェル・プロンプトで次のように入力します。

```
java -version
```

JIT が使用できない場合には、以下の内容のメッセージが表示されます。

```
(JIT disabled)
```

JIT が使用できる場合には、以下の内容のメッセージが表示されます。

```
(JIT enabled)
```

非ヒープ・リアルタイム・スレッドの使用

Metronome ガーベッジ・コレクションを使用すると、より一貫性のある応答時間が得られますが、ガーベッジ・コレクションによる中断を完全に回避することが適切な場合もあります。

NoHeapRealtimeThread (NHRT) は RealtimeThread を拡張したものです。これは、ヒープ・メモリーにアクセスできない点で RealtimeThread と異なります。 NHRT は、ガーベッジ・コレクション・サイクル中であっても (制約はいくつかありますが) ヒープにアクセスすることなく実行を続けることができます。つまり、ヒープにアクセスしないため、そのプログラミング・モデルはリアルタイム・スレッドの場合のプログラミング・モデルとは異なるということになります。

NHRT を使用する際の考慮事項

NHRT に関して考慮すべき点:

- NHRT は、主として、ガーベッジ・コレクションが許されないタスクで使用されます。例えば、ご使用のアプリケーションにとって時間がきわめて重要であり、いかなる中断も許されないような場合です。
- 時間が重要であるという理由で NHRT を使用している場合には、Ahead-of-Time (AOT) コンパイラーの使用、すなわち **-Xnojit** オプションの指定も考慮してください。
- **-Xrealtime** オプションを使用すると、自動的に Metronome ガーベッジ・コレクターが使用されます。ガーベッジ・コレクションは細かく分けられた割り込み可能なステップで少しずつ実行されるため、Metronome ガーベッジ・コレクターを使用すると NHRT をコーディングする必要が少なくなる可能性があります。
- NHRT スレッドは、ガーベッジ・コレクターより優先順位が高いため、ガーベッジ・コレクターによる影響を受けることなく実行されます。 Java スレッドの優先順位は 1 から 10 までです。NHRT が存在する場合、Java スレッドの優先順位は、プログラムで設定された優先順位に関わりなく 0 に再設定されます。ガーベッジ・コレクターの優先順位は、最高位のリアルタイム・スレッドより若干高い値に自動的に設定されます。 NHRT の優先順位は、ユーザーにより、最高位のリアルタイム・スレッドより少なくとも 1 高い値に設定されます。これにより、NHRT はガーベッジ・コレクターから独立するようになっています。

  注: NHRT でガーベッジ・コレクションが全く行われないわけではではありません。Metronome アラーム・スレッドのガーベッジ・コレクターが、システム内で最も高い優先順位で実行されるからです。この優先順位設定により、JVM がアクティブ化されて、ガーベッジ・コレクターが何らかの動作を行う必要があるかどうか検査されるようになります。 Metronome アラーム・スレッドを実行するための作業量は小さく、パフォーマンスに大きな影響を与えることはありません。マルチプロセッサー・システムでは、アラーム・スレッドは NHRT スレッドと同時に実行することができるため、ガーベッジ・コレクションによる中断は発生しません。
- NHRT の対象は、スコープ・メモリー域と永久メモリー域に限定されているので、Java メソッドでは、ヒープから NHRT が割り振られていないことを確認するための検査が行われます。開始メソッドは、NHRT がヒープから割り振られて

いないかどうかを検査し、割り振られている場合には例外 (MemoryAccessError) を返します。 NHRT は ImmortalMemory と ScopedMemory にのみアクセスできます。

- ロックのセマンティクスは変更されていないため、ロックが共有されている場合には、通常のスレッドによって NHRT スレッドがブロックされることがあります。
- ヒープを使用するスレッドと同じメソッドを NHRT が使用しようとしたときに、それが同期メソッドの場合には、ヒープを使用するスレッドの優先順位が高くなることがあります。
- NHRT とヒープ・スレッドとの通信には、非ブロッキング・キューを使用してください。非ブロッキング・キューを使用しない場合には、2 つのタイプのスレッドを分けてください。

例外

NHRT を使用するときに、以下の例外が発生することがあります。

- IllegalAssignmentError。例えば、スコープ・メモリーへの参照を永久メモリーで作成しようすると、このエラーが発生することがあります。
- MemoryAccessError。例えば、NHRT がヒープ・メモリーを参照しようとすると、このエラーが発生することがあります。

非同期イベント処理の制約

以下のいくつかの場合に、ガーベッジ・コレクション中に NHRT がブロックされることがあります。

1. NHRT が、ヒープ・メモリーから割り振られたハンドラーと既に関連付けされている AsyncEvent に対して fire()、setHandler()、または addHandler() のいずれかを呼び出したとき
2. NHRT が、ヒープ・メモリーから割り振られたハンドラーと関連付けされている Timer に対して destroy()、start()、または stop() のいずれかを呼び出したとき
3. NHRT がスコープを終了させる最後のスレッドであり、そのスコープから Timers または AsyncEvents をシャットダウンしている場合。ただし、Timers または AsyncEvents が、ヒープ・メモリーから割り振られたハンドラーと関連付けされている場合。

NHRT で上記のような状況を回避するためには、次のようにします。

1. NHRT から起動される可能性がある AsyncEvent または Timer には、ヒープから割り振られたハンドラーを追加しないでください。
2. ハンドラーがヒープ・メモリーから割り振られている AsyncEvents または Timers があるスコープから、NHRT が最後に終了することがないようにしてください。

メモリーおよびスケジューリングの制約

JVM は、非ヒープ・リアルタイム・スレッドが、ヒープ上のオブジェクトへの参照をオペランド・スタックにロードしないようにします。その場合、javax.realtime.MemoryAccessError はスローされます。

また、JVM は、スコープ・メモリー内のオブジェクトへの参照が、ヒープ・メモリーまたは永久メモリーに保管されないようにします。スコープ・メモリーは NHRT により使用されるとは限りませんが、永久メモリーが不適切であり、NHRT コンテキストでメモリーの割り振り解除が必要な場合は、使用される可能性があります。

NHRT の実行中に、オブジェクトへの参照がフィールドに取り込まれた場合は、そのフィールドで、ヒープ上のオブジェクトへの既存の参照を正常に上書きできます。既存の参照は、MemoryAccessError が生成されることなく、NHRT によって正常に上書きされます。

クラス・ロードの制約

クラスは、クラス・ローダーと同じメモリー域にロードされます。クラス・ローダーのデフォルトの領域は永久メモリーです。

応答時間が予期したとおりになるように、アプリケーションを「ウォームアップ」する必要があります。また、アプリケーションは、クラス・ロードによってリアルタイム・スレッドや非同期イベント・ハンドラーが後で中断されないように、クラスを早期にロードする必要があります。

NHRT とともに実行する場合の Java スレッドに関する制約

JVM ではシステム・プロパティーが共有され、スレッドからそれらのシステム・プロパティーへのアクセスが可能であるため、NHRT が実行される JVM で getProperties メソッドおよび setProperties メソッドを使用する際には注意が必要です。システム・プロパティーを NHRT からアクセス可能にするには、それらのプロパティーが永久メモリーになければなりません。

`java.lang.System` クラスには、スレッドとシステム・プロパティーとの相互作用を可能にする、以下のメソッドを含む、複数のメソッドが用意されています。

```
String getProperty(String)
String getProperty(String,String)
Properties getProperties()

String setProperty(String,String)
void setProperties(Properties)
```

リアルタイム JVM は、すべてのシステム・プロパティーを保管するために、リアルタイム JVM オブジェクト専用に作成された com.ibm.realtime.ImmortalProperties クラスのインスタンスを使用します。このインスタンスを使用することで、System.setProperty() メソッドまたは System.getProperties.setProperty() メソッドのすべての呼び出しで、永久メモリーにプロパティーが保管されるようになります。この場合、特別なユーザー・コードは必要ありませんが、プロパティーが設定されるたびに、一部の永久メモリーが消費されるということを理解しておいてください。

共有 Properties オブジェクトがシステム・プロパティーの保管に使用されるため、setProperties() メソッドの呼び出しは少し難しくなります。NHRT が実行されているリアルタイム JVM でアプリケーションを実行する場合は、setProperties メソッドの呼び出しで、永久メモリーに作成された com.ibm.realtime.ImmortalProperties クラス (またはサブクラス) のインスタンスを渡す必要があります。このインスタンスを使用することで、setProperties メソッドを使用して設定されたすべてのプロパティーが確実に永久メモリーに保管されます。

注: setProperties(null) を呼び出すと、新しい ImmortalProperties オブジェクトがデフォルト設定のプロパティーで内部的に作成されます。これにより永久メモリーがさらに消費されます。

getProperties() メソッドの呼び出しでは、設定したオブジェクトまたはデフォルトのプロパティー・オブジェクト (com.ibm.realtime.ImmortalProperties オブジェクト) のいずれかが返されます。getProperties() メソッドを呼び出す既存のコードとの互換性を最大化するために、ImmortalProperties オブジェクトは、標準の JVM 内のオブジェクトを直列化してから直列化解除します。デフォルトでは、ImmortalProperties の直列化の際に、通常の Properties オブジェクトが直列化されます。これは、標準的な JVM には ImmortalProperties オブジェクトがなく、直列化解除に失敗するためです。このデフォルトの動作をオーバーライドするために、ImmortalProperties クラスには enabledReplacement(boolean) メソッドが用意されており、false を指定して呼び出した場合、デフォルトの動作が無効になります。この場合、直列化で ImmortalPropeties オブジェクトを直列化してから、直列化解除することができ、解除後のオブジェクトをリアルタイム JVM での System.setProperties メソッドの呼び出しで使用することができます。

注: 直列化解除は永久メモリーで行われ、現行の限定リソースを消費しすぎる場合があります。

セキュリティー・マネージャー

システムに設定されたセキュリティー・マネージャーは、JVM のすべてのタイプのスレッドで使用されます。そのため、NHRT が実行されるリアルタイム JVM では、セキュリティー・マネージャーを永久メモリーに割り振る必要があります。リアルタイム JVM により、コマンド行オプションで指定された任意のセキュリティー・マネージャーが永久メモリーに確実に割り振られます。このセキュリティー・マネージャーは、System.setSecurityManager(SecurityManager) メソッドの呼び出しを介して設定することもできます。アプリケーションがこの方法でセキュリティー・マネージャーを設定する場合は、NHRT を正しく実行できるように、セキュリティー・マネージャーが永久メモリーから割り振られたものであることを確認する必要があります。

セキュリティー・マネージャーによって返されたすべてのオブジェクトおよびスローされた例外は、(キャッシュした場合は) 永久メモリーに保管するか、あるいは現行の割り振りコンテキストに割り振る必要があります。

同期

MonitorControl クラスおよびそのサブクラスである PriorityInheritance は、同期 (特に優先順位逆転の制御) を管理します。これらのクラスでは、優先順位逆転の制御ポリシーを、デフォルトとして、または特定のオブジェクトに対して設定することができます。

WaitFreeReadQueue、WaitFreeWriteQueue、および WaitFreeDequeue の各クラスでは、スケジュール可能なオブジェクト (特に NoHeapRealtimeThread のインスタンス) と通常の Java スレッドの間の無待機通信が可能です。

WaitFree キュー・クラスにより、NoHeapRealtimeThread のインスタンスと、ガーベッジ・コレクションによる遅延が発生する可能性のあるスケジュール可能なオブジェクトとで共有されるデータへの安全な同時アクセスが提供されます。

非ヒープ・リアルタイム・クラスの安全性

JSE API の一部を必ずしも非ヒープ・コンテキストで使用できるわけではありません。ヒープ・スレッドと非ヒープ・スレッド間で共有されるクラスには制約事項があります。JVM で提供されるクラスが安全に使用できることを認識しておいてください。

オブジェクトの共有

非ヒープ・リアルタイム・スレッドで実行されるメソッドは、ヒープ上のオブジェクトへの参照のロードを試みるたびに、javax.realtime.MemoryAccessError をスローします。

図 3 に、回避する必要があるいくつかのコード例を示します。

```
/**
 * NHRTError1
 *
 * この例は、ヒープ・オブジェクト参照にアクセスする NHRT の簡単な
 * デモンストレーションです。
 *
 * 生成されるエラーは以下のとおりです。
 *
 * Exception in thread "NoHeapRealtimeThread-0" javax.realtime.MemoryAccessError
 *      at NHRT.run(NHRTError1.java:56)
 *      at javax.realtime.RealtimeThread.runImpl(RealtimeThread.java:1754)

 */
import javax.realtime.*;

public class NHRTError1 {
    public static void main(String[] args) {
        NHRTError1 example = new NHRTError1();

        example.run();
    }

    public NHRTError1() {
        message = new String("This on the heap.");
    }

    static public String message; /* NHRT は静的フィールド (常に永久) に直接アクセスできます。*/
    static public NHRT myNHRT = null;

    public void run() {
            ImmortalMemory.instance().executeInArea(new Runnable() {
            public void run() {
                NHRTError1.this.myNHRT = new NHRT();
            }
        });

        myNHRT.start();

        try {
            myNHRT.join();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
```

図 3. ヒープ・オブジェクト参照にアクセスする NHRT の例

```
    /* NHRT クラス */
    class NHRT extends NoHeapRealtimeThread {
        public NHRT() {
            super(null, ImmortalMemory.instance());
        }

        /* 静的参照を介して NHRTError1.message にストリングを表示します */
        public void run() {
            System.out.println("Message: " + NHRTError1.message);
        }
    }

}
```

図 4. ヒープ・オブジェクト参照にアクセスする NHRT の例 (図 1 の続き)

65 ページの図 3 では、以下のような **javax.realtime.MemoryAccessError** が生成されます。

```
Exception in thread "NoHeapRealtimeThread-0" javax.realtime.MemoryAccessError
        at NHRTError1$NHRT.run(NHRTError1.java:56)
        at javax.realtime.RealtimeThread.runImpl(RealtimeThread.java:1754)
```

オブジェクトを非ヒープ・リアルタイム・スレッドと標準的な Java スレッドの両方からアクセス可能にする場合は、そのオブジェクトを永久メモリーに割り振る必要があります。同様に、オブジェクトを非ヒープ・リアルタイム・スレッドとリアルタイム・スレッドからアクセス可能にする場合は、そのオブジェクトをスコープ・メモリー域に保持することもできます。

65 ページの図 3 では、"This on the heap." というストリングへの参照がクラス変数に保持されました。永久メモリーにすべてのクラスが割り振られているため、この変数には NHRT からアクセス可能です。このストリングは NHRT のコンストラクターに渡された可能性もあります。

ほとんどのオブジェクトに他のオブジェクトへの参照が含まれるため、通常のスレッドと NHRT 間でこのようなオブジェクトを共有する際には注意する必要があります。通常のスレッドと NHRT 間で共有されるオブジェクトの典型的な例として、永久メモリーに割り振られている LinkedList があります。十分注意しないと、標準的なスレッドがヒープ上にあるオブジェクトを LinkedList に取り込む可能性があります。また、さらに懸念されるのは、オブジェクトをトラッキングするために LinkedList によって割り振られたデータ構造が、通常のスレッドによってヒープに割り振られると、NHRT で容易に MemoryAccessError が発生することです。

一部のクラスは、それらの個々のインスタンスが割り振られる可能性のある場所に関係なく、NHRT スレッドと他のスレッドで安全に共有することはできません。これらのクラスは、通常はキャッシュ目的で、クラス変数に保管されているオブジェクトに依存します。アドレスをキャッシュに入れる典型的な例が InetAddress です。この InetAddress の特定のメソッドを呼び出す最初のスレッドがヒープ上で実行されている場合、今後、同じメソッドを NHRT から呼び出すのは危険です。

NHRT によるオブジェクトのロック

NHRT は他のスレッドと同期しないようにする必要があります。例えば、以下のシナリオを考えてみます。

- 優先順位の低いリアルタイム・スレッドが同期化ブロックまたは同期化メソッドを入力して、オブジェクトで同期します。
- 優先順位の高い NHRT は、同じオブジェクトでの同期の試行時にブロックされます。
- 優先順位の継承により、リアルタイム・スレッドは一時的に NHRT と同じ優先順位であると見なされます。
- その後、ガーベッジ・コレクションは NHRT より高い優先順位で実行されるため、NHRT に割り込む可能性があります。NHRT を使用するのはガーベッジ・コレクションによる割り込みを回避するためなので、このシナリオでは NHRT を使用する意味がなくなってしまいます。

同じオブジェクトでの NHRT と他のスレッドの同期を回避できない場合もありますが、その可能性を最小限に抑える必要があります。オブジェクト共有時での不必要な同期の回避は慎重に行ってください。

安全なクラスに関する制約事項

一部の考慮事項は、アプリケーションにリアルタイム・スレッドと非ヒープ・リアルタイム・スレッドの両方のオブジェクトが含まれている場合に適用されます。

- 非ヒープ・リアルタイム・スレッドでは、リアルタイム・スレッドとの相互作用による MemoryAccessError が発生する可能性があります。
- 非ヒープ・リアルタイム・スレッドでは、リアルタイム・スレッドに起因するガーベッジ・コレクションによる遅延が偶発的に発生する場合があります。

非ヒープ・リアルタイム・スレッドで発生する MemoryAccessError

2 つのタイプのスレッドの両方が同じクラスのメソッドを呼び出した場合、リアルタイム・スレッドは、ヒープから割り振られたオブジェクトをクラスの静的変数に保管して「汚染する」可能性があります。非ヒープ・リアルタイム・スレッドは、それらのヒープ・オブジェクトへのアクセス試行時に、MemoryAccessError を受け取ります。この汚染はクラスのインスタンスでも発生する可能性があります。残念ながら、これらの問題は両方とも典型的なコーディング・パターンで見られる可能性が高いため、いくつかのケースで試してみることをお勧めします。

クラスで時間のかかる操作が行われているときは、後続操作のパフォーマンスを向上させるために、多くの場合、結果がキャッシュに入れられるように選択されます。このキャッシュは、通常、クラス内の静的変数に固定されている HashMap などのコレクションです。ヒープ・コンテキストで動作するリアルタイム・スレッドは、このコレクションにヒープ・オブジェクトを保管できます。コレクションにはこのオブジェクト自体だけでなく、インフラストラクチャー・オブジェクト (インデックスの一部など) も追加することができます。非ヒープ・リアルタイム・スレッドが後でコレクションへのアクセスを試行するときは、アクセスの対象が他のスレッドによって追加されたオブジェクトでない場合でも、インフラストラクチャー・オブジェクトのロードが試行されるため、MemoryAccessError が表示されます。クラス・ライブラリーが作成され、パフォーマンスに応じて調整されると、これらのキャッシュはより頻繁に使用されるようになります。

また、クラス・インスタンスは、さまざまな方法でヒープ・オブジェクトによって汚染されるようになります。例えば、永久メモリーに作成されたために、両方のタ

イプのスレッドからアクセス可能なインスタンスがあるとします。ヒープ・コンテキスト内のリアルタイム・スレッドが最初にオブジェクトを使用すると、2 次オブジェクトが元のオブジェクトのフィールドに保管される場合があります。2 番目のオブジェクトがヒープ・コンテキスト内にある場合に、非ヒープ・リアルタイム・スレッドがそのオブジェクトを使用すると、MemoryAccessError が再度表示されます。これらの 2 次オブジェクトは、最初の使用時に常に追加されるわけではなく、数回使用された後に追加される場合があります。また、これらのオブジェクトは、使用頻度の高いメソッドのパフォーマンスを向上させるように設計されている場合があります。

ガーベッジ・コレクションによる遅延が発生する NoHeap スレッド

非ヒープ・スレッドには、ガーベッジ・コレクションによる遅延が発生しないように、他のスレッドよりも高い優先順位を割り当てる必要があります。

さらに、クラスに同期メソッドが含まれている場合、そのメソッドを呼び出す非ヒープ・リアルタイム・スレッドで、ガーベッジ・コレクションによる遅延が意図せず発生する可能性があります。このシナリオは、 66 ページの『NHRT によるオブジェクトのロック』に記載されています。

クラスに同期メソッド (静的メソッドまたはインスタンス・メソッドのいずれか) が含まれている場合、そのメソッドを呼び出す非ヒープ・リアルタイム・スレッドで、ガーベッジ・コレクションによる遅延が意図せず発生する可能性があります。この問題は、リアルタイム・スレッドが同期メソッド (静的メソッドまたはインスタンス・メソッド) にアクセスしているときに、リアルタイム・スレッドが完了するまで待機しないようにする別の同期メソッドを非ヒープ・リアルタイム・スレッドが呼び出そうとした場合に発生します。非ヒープ・リアルタイム・スレッドの優先順位がリアルタイム・スレッドの優先順位よりも高い場合、リアルタイム・スレッドの優先順位が上がります。そのスレッドがガーベッジ・コレクションによる割り込みを強制的に待機させられた場合、優先順位が逆転する可能性があります。これは、ガーベッジ・コレクター・スレッドの優先順位が、優先順位の最も高い (同期メソッド入力の待機を現在ブロックされている非ヒープ・リアルタイム・スレッドほど高くはない場合があります) リアルタイム・スレッドよりも高いためです。

このような問題を解決する唯一の方法は、非ヒープ・リアルタイム・スレッドが、他のスレッド・タイプと共有しているクラスまたはインスタンスで同期メソッドを絶対に呼び出さないようにすることです。残念ながら、例えば、同期ブロックが含まれている可能性があることや、同期メソッドが呼び出される可能性があることなど、メソッドが同期化されているかどうかを、メソッド・シグニチャーから常に判断できるわけではありません。

要約

NoHeapRealtimeThread クラスではリアルタイム環境での複雑さが大幅に増し、異なるタイプのスレッドが 1 つの環境で動作すると、非常に多くの問題が発生する可能性があります。アプリケーションの開発時には、クラスをさまざまなスレッド・タイプで共用する領域を慎重に設計する必要があります。これらのスレッドによる SDK でのクラスの使用は特に重要です。分析が複雑であるため、SDK で提供されるすべてのクラスがこのような共用時に安全であると保証することはできません。代わりに、クラスの少量のサブセットが検証されました。最初に、

MemoryAccessError について重点的に検証を行い、クラスが非ヒープ・スレッドと他のタイプのスレッドの両方で使用できるように、クラスの分析、テスト、および、必要に応じて変更を行った結果をリストにしました。

安全なクラス

このセクションでは一連のクラスをリストしています。NoHeapRealtimeThread およびその他のスレッド・タイプでこれらのクラスを安全に使用できるようにすることを目的としています。

特に、MemoryAccessError 発生時の安全面に重点を置いています。以下のリストでは、同じ JVM で 3 つのすべてのスレッド・タイプで使用できるクラスの詳細を示しています。

注: クラスの個々のインスタンスが常に安全に共有されるわけではありません。

すべてのスレッド・タイプでクラスが確実に安全に使用できるようにするには、以下のルールに従います。

- インスタンスは、そのインスタンスにアクセスするスレッドからアクセス可能なメモリー域に作成する必要があります。
- クラスにパブリック静的フィールドがある場合は、それらのフィールドにヒープ・オブジェクトを保管しないようにしてください。
- クラスにパブリック・インスタンス・フィールドがある場合は、それらのフィールドにヒープ・オブジェクトを保管しないようにしてください。

IBM から提供されるクラスがすべて NHRT セーフであるわけではありません。NHRT セーフであるクラスが含まれているのは、以下のパッケージです。

- java.lang パッケージ
- java.lang.reflect パッケージ
- java.lang.ref パッケージ (すべてのクラス)
- java.net パッケージ
- java.io パッケージ
- java.math パッケージ

以下の表で、これらのパッケージ内の NHRT セーフではないクラスを示します。

表 5. java.lang パッケージ内の NHRT セーフではないクラス

クラス	メソッド
java.lang.ProcessBuilder	*
java.lang.Thread	getAllStackTraces()Ljava.util.Map;
java.lang.ThreadGroup	*
java.lang.ThreadLocal	*
java.lang.InheritableThreadLocal	*

表 6. java.lang.reflect パッケージ内の NHRT セーフではないクラス

クラス	メソッド
java.lang.reflect.Proxy.*	*

表 7. *java.net* パッケージ内の *NHRT* セーフではないクラス

クラス	メソッド
java.net.SocketPermission.*	newPermissionCollection()Ljava.net.SocketPermissionCollection;

表 8. *java.io* パッケージ内の *NHRT* セーフではないクラス

クラス	メソッド
java.io.ExpiringCache	*
java.io.SequenceInputStream	*
java.io.FilePermission	newPermissionCollection()Ljava.io.FilePermissionCollection;
java.io.ObjectInputStream	*
java.io.ObjectOutputStream	*
java.io.ObjectStreamClass	*

表 9. *java.math* パッケージ内の *NHRT* セーフではないクラス

クラス	メソッド
java.math.BigInteger	*

パッケージには、安全ではないクラスを含むサブパッケージが含まれていることがあります。例えば、以下のクラスは NHRT セーフではありません。

- java.lang.management.*
- java.lang.annotation.*
- java.lang.instrument.*

クラスが NHRT セーフと見なされる場合でも、クラスを NHRT で使用するのは適切ではない場合があります。アプリケーション開発者は、クラスのリアルタイム要件を、そのクラスが NHRT セーフであるかどうかに関係なく、ケースバイケースで判断する必要があります。

JVM 間でのクラス・データの共用

Java 仮想マシン (JVM) では、ディスクのメモリー・マップ・キャッシュ・ファイルにクラス・データを保管することによって、JVM 間でそのクラス・データを共有できます。

複数の JVM がキャッシュを共有する場合、共有によって全体の仮想ストレージの使用量が削減されます。また、共有によって、キャッシュ作成後の JVM の起動時間も短縮されます。共有クラス・キャッシュは、アクティブな JVM に依存せず、破棄されるまで存続します。共有キャッシュには、以下を含むことができます。

- ブートストラップ・クラス
- アプリケーション・クラス
- クラスを記述するメタデータ
- Ahead-of-time (AOT) コンパイル済みコード

共有クラス・キャッシュは、IBM WebSphere Real Time for RT Linux では、非リアルタイム・モードおよびリアルタイム・モードの両方で使用できます。ただし、キャッシュの形式、作成、およびデータの取り込みの各方法は異なります。リアル

タイム・モードのキャッシュは、非リアルタイム・モードのキャッシュと互換性がありません。非リアルタイム・モードのキャッシュは、標準 JVM と同じ方法で作成され、データが取り込まれます。つまり、このキャッシュは、アプリケーションの実行時に、ユーザーには意識されずに JVM によって作成されて、データが取り込まれます。リアルタイム・モード (**-Xrealtime** オプションを指定) で、**admincache** を使用して共有クラス・キャッシュの作成とデータの事前取り込み (**-populate** オプション指定) を行う必要があります。リアルタイム・モードで実行するアプリケーションは、事前にデータを取り込んであるキャッシュから内容を読み取ることはできますが、内容を変更することはできません。

キャッシュの作成、データの取り込み、および破棄には、**admincache** ツールを使用します。

アプリケーションが共有クラス・キャッシュを使用できるようにするには、コマンド行に **-Xshareclasses** オプションを追加します。リアルタイム・モードのキャッシュは読み取り専用のため、**-Xshareclasses** の非リアルタイム・モード・サブオプションの中にはリアルタイム・モードで使用できないものもあります。

詳しくは、 44 ページの『admincache ツールの使用』、 101 ページの『JVM 間でのクラス・データの共有 (非リアルタイム・モード)』、および 152 ページの『共有クラス診断』を参照してください。

共有クラス・キャッシュを使用したアプリケーションの実行

共有クラス・キャッシュを使用してアプリケーションを実行するには、コマンド行で **-Xshareclasses** オプションを使用します。

表 10 に、**-Xshareclasses** オプションを使用して、リアルタイム・モードでアプリケーションを実行する場合に使用可能なサブオプションを示します。

表 10. リアルタイム・モードでアプリケーションを実行する場合に使用可能なサブオプション

オプション	意味
cacheDir=<*directory*>	共有クラス・キャッシュ・データの読み取りと書き込みを行うディレクトリーを設定します。デフォルトでは、<*directory*> は /tmp/javasharedresources です。ディレクトリー名は、キャッシュを作成するために admincache コマンドで使用する **-cacheDir** オプションで指定した名前と一致している必要があります。
name=<*name*>	使用する共有クラス・キャッシュの名前。この名前は、キャッシュを作成するために admincache コマンドで使用する **-cacheName** オプションで指定した名前と一致している必要があります。名前は 53 文字以下である必要があります。

表 10. リアルタイム・モードでアプリケーションを実行する場合に使用可能なサブオプション (続き)

オプション	意味
none	クラスの共有を明示的に使用不可にします。コマンド行の最後に追加して、クラス・データ共有を使用不可にできます。このサブオプションにより、コマンド行の前のほうにあるクラス共有引数は無効になります。
nonfatal	エラーまたは警告にかかわらず、常に JVM を開始します。クラス・データ共有が失敗していても、JVM の開始を許可します。JVM の通常の動作では、クラス・データ共有が失敗した場合、JVM の開始は拒否されます。**nonfatal** が選択されていて、共有クラス・キャッシュの初期化が失敗した場合、JVM では読み取り専用モードでのキャッシュへの接続が試行されます。この試行が失敗すると、JVM はクラス・データ共有を使用せずに開始します。
silent	すべての出力メッセージを抑止します。すべての共有クラス・メッセージ (エラー・メッセージを含む) をオフにします。JVM で初期化が行われないようにリカバリー不能エラー・メッセージが表示されます。
verbose	詳細出力を使用可能にし、共有クラス・キャッシュの全体的な状況と、詳細なエラー・メッセージを提供します。
verboseAOT	コンパイル済み AOT コードがキャッシュで検出された場合 (AOT メソッドのロード要求時など)、詳細出力を使用可能にします。
verboseHelper	Java ヘルパー API の詳細出力を使用可能にします。この出力には、クラス・ローダーによるヘルパー API の使用方法が示されます。
verboseIO	クラス・ロード要求の詳細出力を使用可能にします。このオプションでは、キャッシュ入出力アクティビティーに関する詳細出力が提供され、検出されたクラスに関する情報がリストされます。

これらのオプションが適正かどうかを確認するには、admincache と一緒に **-printvmargs** オプションを使用します (詳しくは、**-printvmargs** を参照してください)。**nonfatal** オプションは、共有クラス・キャッシュに関する警告とエラーを強制的に無視するよう JVM に指示するため、一般的な用途には適していません。**none** オプションは、クラス共有を明示的に使用不可にします。コマンド行で **-Xshareclasses** オプションを省略した場合も、同じ働きになります。

-Xshareclasses サブオプションについて詳しくは、クラス・データ共有のコマンド行オプションを参照してください。

Metronome ガーベッジ・コレクターの使用

WebSphere Real Time for RT Linux では、標準的なガーベッジ・コレクターの代わりに、Metronome ガーベッジ・コレクターを使用します。

プロセッサー使用率の制御

Metronome ガーベッジ・コレクターが使用可能な処理能力の量を制限することができます。

Metronome ガーベッジ・コレクターによるガーベッジ・コレクションを **-Xgc:targetUtilization=N** オプションを使用して制御することで、ガーベッジ・コレクターで使用される CPU 量を制限できます。

例えば、次のようにします。

```
java -Xrealtime -Xgc:targetUtilization=80 yourApplication
```

この例では、60 ミリ秒ごとに 80% の時間がアプリケーションの実行に使用されるように指定しています。残りの 20% の時間はガーベッジ・コレクションに使用されます。Metronome ガーベッジ・コレクターに十分なリソースが与えられている場合は、使用率レベルが保証されます。ガーベッジ・コレクションは、ヒープ内のフリー・スペース量が、動的に決定されたしきい値を下回ったときに開始されます。

Metronome ガーベッジ・コレクターの調整

アプリケーションで使用するメモリー量を制御することで、リアルタイム環境を調整できます。例えば、**-Xmx**、**-Xgc:immortalMemorySize=size**、**-Xgc:scopedMemoryMaximumSize=size**、および **-Xgc:targetUtilization=N** の各オプションを使用します。

- **-Xmx** オプションは、ヒープのサイズを制限する場合に使用します。

  選択値はヒープ・サイズの上限として使用されるため、長期間での可能性の高い使用量を反映します。低すぎる値を選択した場合には、メモリーの占有スペースは低下しますが、ガーベッジ・コレクションの頻度が高くなり、全体的なスループットが低下します。適切なリアルタイム・パフォーマンスのために、ページングを回避してください。通常、マシン上のすべての実行中のプロセスの占有スペースが物理メモリー・サイズを超えないようにします。

- **-Xgc:immortalMemorySize=size** オプションは、永久メモリー域のサイズを制御する場合に使用します。

  永久メモリーの使用は注意深く分析する必要があります。「理想的な」アプリケーションでは、永久メモリーは開始時に使用しますが、その後は使用を停止します。永久オブジェクトの割り振りが続行すると、アプリケーションは永久メモリーが枯渇するまで実行し続けることができます。現在の使用量は、コードに以下を追加することで取得できます。

  ```
  long used = ImmortalMemory.instance().memoryConsumed();
  ```

- **-Xgc:scopedMemoryMaximumSize=size** オプションは、アプリケーションが過剰な量のスコープ・メモリーを要求しないようにするために使用します。このオプションは、調整ではなく診断用に使用してください。

- **-Xgc:targetUtilization=N** オプションは、最悪の状態 (ヒープ・オブジェクトの最大割り振りレート) で、ガーベッジ・コレクターが、アプリケーションでガーベッジが生成されるよりも高速にガーベッジを収集できるようにするために設定します。

  通常はデフォルト値で十分ですが、アプリケーションでガーベッジが生成される可能性がある速度よりも若干速くコレクターがガーベッジを収集できる程度に使用率を上げることで、アプリケーションのパフォーマンスが改善される場合があります。
- **-Xgcthreads <n>** オプションは、追加スレッドを作成してガーベッジ・コレクションを並行で実行する場合に使用します。

  デフォルトでは単一のスレッドを使用します。ワークロードでガーベッジの生成が高速で、CPU サイクルが使用可能な対称型マルチプロセッサーで実行している場合は、このパラメーターを >1 に設定することで、パフォーマンス上のメリットが得られる可能性があります。

  注: このパラメーターにあまりにも大きい値を設定すると、スループットに悪影響が出る可能性があります。

Metronome ガーベッジ・コレクターの制限

まれなケースですが、ガーベッジ・コレクション時の休止が予想よりも長くなる場合があります。

ガーベッジ・コレクション時は、ルート・スキャン・プロセスが使用されます。ガーベッジ・コレクターは、ヒープ内の既知のライブ参照から調べます。参照には以下が含まれます。
- アクティブ・スレッド呼び出しスタックのライブ参照変数。
- 静的参照。
- 永久メモリーおよびスコープ・メモリー内のすべてのオブジェクト参照。

アプリケーション・スレッド・スタックのすべてのライブ・オブジェクト参照を検索するために、ガーベッジ・コレクターはそのスレッドの呼び出しスタックにあるすべてのスタック・フレームをスキャンします。各アクティブ・スレッド・スタックは一度にスキャンされます。このため、スキャンは単一の GC 休止内で実行される必要があります。

その結果、スタックが非常に深いスレッドが複数ある場合は、コレクション・サイクル開始時のガーベッジ・コレクション休止時間が長くなるため、システム・パフォーマンスが予想よりも悪くなる可能性があります。

永久メモリーは段階的に処理されます。他のスコープ・メモリー域はすべて一度に処理されます。このため、スコープ・メモリー域を頻繁かつ大量に使用すると、ルート・スキャンでスコープ・メモリーを処理する場合にガーベッジ・コレクション休止時間が長くなるため、システム・パフォーマンスが予想よりも悪くなる可能性があります。

第 6 章 アプリケーションの開発

リアルタイム・アプリケーションの作成に関する重要な情報について、コードの例を示しながら説明します。

- 『リアルタイムを活用する Java アプリケーションの作成』
- 87 ページの『サンプル・アプリケーション』
- 95 ページの『サンプル・リアルタイム・ハッシュ・マップ』
- 96 ページの『Eclipse を使用した WebSphere Real Time for RT Linux アプリケーションの開発』

リアルタイムを活用する Java アプリケーションの作成

以下の各例では、リアルタイム環境を活用する方法について説明しています。これらの例には、コードに対する変更は全くなしで Java アプリケーションをリアルタイムで実行する、最も簡単な例から、非ヒープ・リアルタイム・スレッドを計画して記述する、複雑な処理までさまざまな例があります。お客様のアプリケーションに最適な手法を決定するために役立つ根拠も提供されています。

リアルタイム・アプリケーションの作成の概要

リアルタイム・テクノロジーのフィーチャーを活用するために、複雑な NHRT (No-Heap Real-Time) アプリケーションを作成する必要はありません。既存のコードをほとんど変更しなくても、その利点の一部が得られます。

アプリケーション・プログラマーは、WebSphere Real Time for RT Linux を利用するために以下のステップを実行できます。

1. 標準 Java アプリケーションをリアルタイム JVM で実行して、Metronome ガーベッジ・コレクションの利点を得て、アプリケーションのランタイムの予測可能性を大幅に向上させる。
2. コードのプリコンパイル後に **-Xnojit** オプションを追加して、AOT (Ahead-Of-Time) コンパイラーを使用する。 55 ページの『プリコンパイルされた JAR ファイルの共有クラス・キャッシュへの格納』を参照してください。
3. ご使用のアプリケーションで、java.lang.Thread を javax.realtime.RealtimeThread で置き換える。 AOT オプションと比較して若干の向上が見られる場合があります。

   リアルタイム・スレッドを使用する主な利点は、各スレッドに付与する優先順位を制御できる点にあります。リアルタイム・スレッドは周期的にすることもできます。これらの利点を活用するには、アプリケーション自体に変更を行う準備をする必要があります。
4. タイマーまたは外部イベントを処理するためにリアルタイム・スレッドおよび非同期イベント・ハンドラーを使用する特定のアプリケーションを計画および作成する。以下の 3 つの要因を検討してください。
   - リアルタイム・スレッドに割り当てる優先順位の計画
   - オブジェクトを保持するために使用するメモリー域の決定

© Copyright IBM Corp. 2003, 2011

- イベント・ハンドラーとの通信

5. 非ヒープ・リアルタイム・スレッドを使用する特定のアプリケーションを計画および作成する。 NHRT スレッドはリアルタイム・スレッドの拡張であり、割り当てる優先順位およびメモリー域を検討する必要があります。一般的に、このステップは、アプリケーションが GC 休止時間 (ミリ秒未満) に匹敵する時間でイベントを処理する必要がある場合にのみ実行します。非ヒープ・リアルタイム・スレッドでの開発の複雑さを過小評価しないでください。

図 5 に、前述のステップを示します。

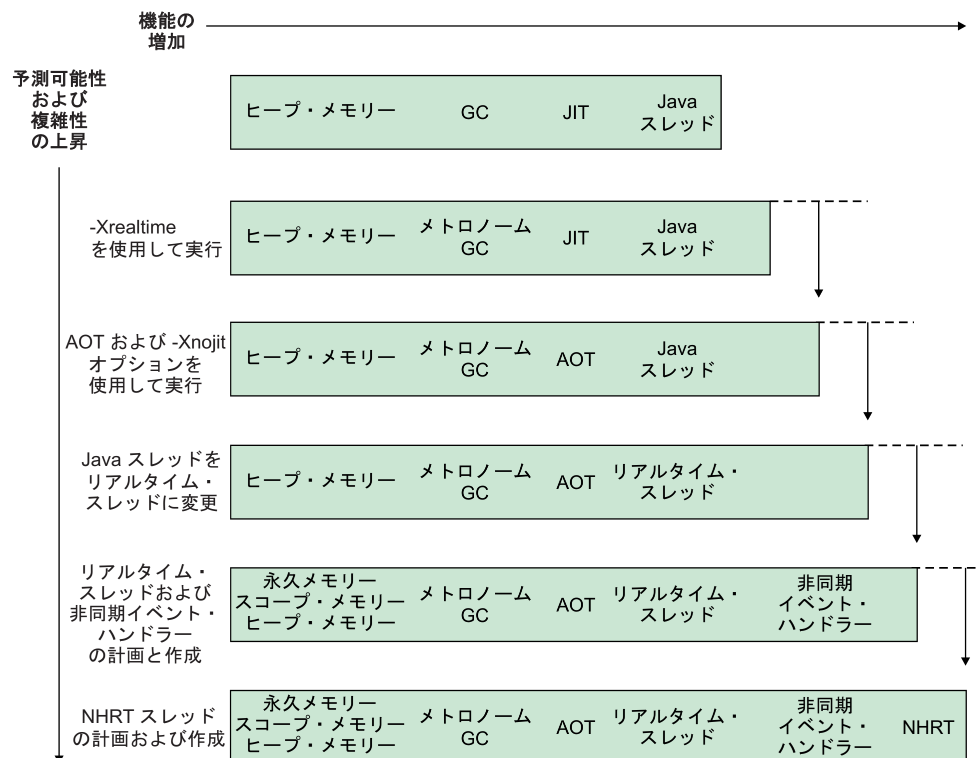


図 5. RTSJ のフィーチャーと予測可能性の向上との比較

WebSphere Real Time for RT Linux アプリケーションの計画

リアルタイム Java アプリケーションを作成する準備をする際には、Java スレッド、リアルタイム・スレッド、または非ヒープ・リアルタイム・スレッドのうちどれを使用するかを検討する必要があります。また、スレッドが使用するメモリー域を決定することもできます。

このタスクについて

アプリケーションを計画する際に行う必要がある決定について以下のステップで記載します。

手順

1. タスクを特定する。
2. 以下のように、タイミング期間を決定する。
   - 10 ミリ秒より長い応答。Java スレッドを選択して、Metronome ガーベッジ・コレクターのみを活用します。

     このタイプのスレッドは、保管用にヒープ・メモリーのみを使用します。欠点はガーベッジ・コレクションがアプリケーションを中断する点ですが、Metronome ガーベッジ・コレクターによって制御されるため、中断の長さとタイミングは予測可能です。
   - 10 ミリ秒未満の応答。リアルタイム・スレッドを選択します。

     リアルタイム・スレッドは、ヒープ・メモリー、スコープ・メモリー、または永久メモリーに配置できます。リアルタイム・スレッドを使用する利点は、以下のとおりです。
     - 標準の Java スレッドよりも高い優先順位で実行できる。
     - ガーベッジ・コレクションが、Metronome ガーベッジ・コレクターによって制御される。ただし、ガーベッジ・コレクターは、リアルタイム・スレッドの最高の優先順位よりも高い優先順位で実行されるため、プログラムの実行が中断されます。
   - 1 ミリ秒未満の応答。非ヒープ・リアルタイム・スレッドを選択します。

     非ヒープ・リアルタイム・スレッドの優先順位はガーベッジ・コレクションよりも高く設定できるため、Metronome によって大幅に中断されることはありません。 Metronome アラーム・スレッドのみが、最上位の優先順位で実行され、CPU の使用量が非常に低く抑えられます。
3. アプリケーションで非同期イベント・ハンドラーが必要かどうかを決定する。この要件は、プログラムの構造に依存します。
   - 10 ミリ秒未満の時間応答。リアルタイム・スレッドを選択します。
   - 1 ミリ秒未満の時間応答。非ヒープ・リアルタイム・スレッドを選択します。
4. スレッド優先順位を決定する。 一般的に、期間が短ければ短いほど、優先順位は高くなります。
5. メモリー特性を決定する。
   - タスクの割り振りレートが可変または高く、GC が過負荷になる可能性がある場合は、レート制限を課す (MemoryParameters を使用) か、スコープ・メモリー域に割り振ることを検討します。
   - タスクが計算時に大量の一時データを生成する場合は、スコープ・メモリー域の使用を検討します。
   - タスクが JVM の存続期間に必要なデータを始動時に生成する場合は、永久メモリーの使用を検討します。 JVM の存続期間にわたってオブジェクトが作成し続けられる場合は、永久メモリーは使用しないようにしてください。
   - タスクで通信する必要がある場合、特に NHRT スレッドで実行されている場合は、通信用にスコープ・メモリー域を使用することを検討します。
   - タスクが NHRT スレッドで実行されている場合は、スコープ・メモリー域 (例えば、LTMemory) を作成して、非ヒープ・スレッド、ランタイム・パラメーター、および場合によっては、タスクとの通信に使用される待機なしキュー

を入れることを検討します。 LTMemory オブジェクトは、永久スコープまたは別のスコープのいずれかに作成して、非ヒープ・スレッドがこのオブジェクトを参照したときにエラーが出ないようにする必要があります。

6. アプリケーションの構造およびコンテンツが決定したら、ランタイム・オプションを変更して、アプリケーションのパフォーマンスを改善する。次のステップに、これを行う方法を記載します。
   a. アプリケーションの初期テスト中に、**-Xmx**、**-Xgc:immortalMemorySize=size**、および **-Xgc:scopedMemoryMaximumSize=size** の各オプションを使用して、ヒープ・メモリー、スコープ・メモリー、および永久メモリーに十分なスペース容量を設定する。

      注: Metronome GC の場合、Metronome GC はヒープのサイズを増やさないため、初期ヒープ・サイズと最大ヒープ・サイズは同じである必要があります。ヒープの拡張は、決定性のない操作です。
   b. **-verbose:gc** オプションを使用して、使用メモリー量を決定する。
   c. **-Xgc:targetUtilization** オプションを変更して、ガーベッジ・コレクションの実行に十分な時間を用意する。 デフォルトは 70% であり、ほとんどのアプリケーションで通常、このパーセンテージで十分です。ガーベッジ・コレクション・レートが割り振りレートより若干頻繁になるようにしてください。
   d. **-Xmx** オプションを使用して、ヒープ・メモリーの現実的なサイズを設定する。

Java アプリケーションの変更

リアルタイム Java フィーチャーを利用するコードを作成するには、スレッド用に `java.lang.Thread` の代わりに `javax.realtime.RealtimeThread` を使用します。

始める前に

この例は、`demo/realtime/sample_application.zip` ファイル内の JavaRadar.java クラスに基づいています。

このタスクについて

リアルタイム・スレッドのプログラミング・モデルは、標準 Java アプリケーション用のものと似ています。ただし、リアルタイム・スレッドをプログラムに追加するこのかなり粗い方法は、WebSphere Real Time for RT Linux のフィーチャーをフル活用するわけではありません。そうするには、優先順位が関連付けられるようにスレッドを変更し、さらに使用するメモリー域を検討する必要があります。

スレッドのクラスを変更するだけでは、リアルタイム・スレッドのデフォルトの優先順位が標準 Java スレッドの優先順位よりも高いため、アプリケーションでわずかな利点しか得られません。

JavaRadar を RealtimeThread に変更するには、拡張するクラスを `Thread` から `RealtimeThread` に変更します。

java.lang.Thread を javax.realtime.RealtimeThread に置換

サンプル・アプリケーション内の JavaRadar クラスは、 java.lang.Thread を拡張しています。例えば、次のようにします。

```
public class JavaRadar extends Thread implements Radar
```

この Java スレッドをリアルタイム・スレッドにするには、このクラスの定義を以下のように再定義します。

```
public class RTJavaRadar extends RealtimeThread implements Radar
```

リアルタイム・スレッドの作成

これまで、1 つのアプリケーションの変更のみを行ってきました。今度は、コードをいくらか記述してみましょう。リアルタイムの優先順位とメモリー域を利用するために、リアルタイム・スレッドを使用するアプリケーションを作成することができます。

始める前に

この例は、demo/realtime/sample_application.zip ファイル内にある JavaRadar.java、 RTJavaRadar.java、および RTJavaControlLauncher.java の各クラスに基づいています。

この例では、 78 ページの『Java アプリケーションの変更』で説明されているサンプルと同じサンプルによって永久メモリーを使用する方法を示しています。

このタスクについて

リアルタイム・スレッド用のプログラミング・モデルは、標準の Java アプリケーション用のプログラミング・モデルと類似しています。

リアルタイム・スレッドを使用する利点は、以下のとおりです。

- リアルタイム・スレッドに対する OS レベルのスレッド優先順位の完全サポート。
- スコープ・メモリー域または永久メモリー域の使用。
  - スコープ・メモリーを使用すれば、ガーベッジ・コレクションに影響を与えずに、メモリーの割り振り解除を明示的に制御できます。
  - 非ヒープ・リアルタイム・スレッドを使用すれば、永久メモリーを使ってガーベッジ・コレクションによる一時停止を回避できます。
  - ヒープ内のオブジェクトを参照するリアルタイム・スレッドは、ヒープ・メモリー内に格納されているリアルタイム・スレッドと同様にガーベッジ・コレクションを受けることがあります。
  - 無ヒープ・リアルタイム・スレッドは、ヒープ・メモリー内のオブジェクトを参照できず、その結果ガーベッジ・コレクションから影響を受けません。

80 ページの表 11 では、各優先度は、SimulationThread が最高の優先度を持つということに基づいて割り当てられています。これは、このスレッドが外部イベントを表し、プログラム内のいずれによっても優先使用されることが許可されてはならないためです。 RadarThread は、コントローラーからの ping に素早く応答する必要があります。この応答が速いほど、月着陸船の高度の測定がより正確になります。

また、ListenThread もコントローラーからのコマンドに素早く応答しなければなりませんが、RadarThread に比べるとその次の優先度になります。

これらの 3 つのスレッドは、シミュレーションがサーバーとして実行されるためスコープ・メモリー内にあります。このサーバーは、1 つのシミュレーションの実行を完了すると、スコープ・メモリー域を抜け、その後シミュレーションの別の実行を待機するために、このスコープ・メモリー域に再度入ることができます。このサーバーでは、それ自体をリセットできるように、スコープ・メモリーが使用されています。

RTJavaRadarthread には、各コントローラー・スレッドの中で最高の優先度があります。この理由は、このスレッドが高さを得るために時間を使用しているため、タイミングにより敏感に反応するようにするためです。このスレッドのメモリーは永久メモリーです。その理由は、このスレッドが NHRT として実行されていて、コントローラーは一度だけ実行され、メモリーは JVM の終了時に解放されるためです。

RTJavaControlThread と RTJavaEventThread については、時間制約がそれほどクリティカルではないため、ヒープ・メモリーの使用を許容できます。

RTLoadThread では月着陸船に役立つ機能は実行されません。ただし、RTLoadThread では、メモリーの割り振りと割り振り解除を他のスレッドよりも低い優先度で、より高い優先度のスレッドのパフォーマンスに影響を与えることなく実行できる有意性が示されます。

表 11. サンプル・アプリケーションにおける各メモリー域に対する各スレッドの関係

メモリー	スレッド	優先度
スコープ	demo.sim.SimulationThread	38
	demo.sim.RadarThread	37
	demo.sim.SimulationThread.ListenThread	36
永久	demo.controller.RTJavaRadarThread	15
ヒープ	demo.controller.RTJavaControlThread	14
	demo.controller.RTJavaEventThread	13
スコープおよびヒープ	demo.controller.RTLoadThread	12

例

次の demo.sim.SimulationThread からのコードは、38 の優先度が設定されている箇所を示しています。 **1** このコードの行では、JVM 内で使用できる最大の優先度を取得します。

```
super(null, area);

// 「this」を使用しているように、優先度を個々に設定します。
// PriorityScheduler.MAX_PRIORITY は、
// 推奨されなくなったことに注意してください。
this.setSchedulingParameters(new PriorityParameters(PriorityScheduler
      .getMaxPriority(this)));    1
```

次の demo.sim.SimLauncher からのコードは、スコープ・メモリーが定義されている箇所を示しています。 **2** は、LTMemory の割り振りを示しています。このメモリーは、メモリーを線形時間で割り振るスコープ・メモリー域です。

```
final IndirectRef<MemoryArea> myMemRef = new IndirectRef<MemoryArea>();

/*
 * LTMemory オブジェクトを NHRT がアクセス可能なメモリー域に作成する
 * 必要があります。
 */
ImmortalMemory.instance().enter(new Runnable() {
    public void run() {
        myMemRef.ref = new LTMemory(10000000); 2
    }
});

final MemoryArea simMemArea = myMemRef.ref;
```

simMemArea によって参照される ScopedMemoryArea オブジェクトは、永久メモリー内に割り振られています。これは、NHRT が ScopedMemoryArea を表すオブジェクトを参照できる必要があるためです。このオブジェクトをヒープに割り振ると、IllegalArgumentException をスローする NHRT コンストラクターが生成されます。これは、そのメモリー域の引数がヒープ上にあったためです。

```
simMemArea.enter(new Runnable() {
    public void run() {
        try {
            CommsControl commsControl = new CommsControl();
```

次の demo.controller.RTJavaControlLauncher からのコードは、永久メモリーが定義され、RTJavaRadar によって使用される箇所を示しています。 RTJavaRadar は、コントローラー JVM の全存続期間の間に一度実行されるため、開始時にのみメモリーを割り振るように設計されています。つまり、これは永久メモリーで安全に実行できます。Controller が最初にスコープ・メモリー域に入る必要なしに RTJavaRadar メソッドにアクセスできるため、アプリケーションの設計にメリットがもたらされます。 Controller がリアルタイムの Java と同様に通常の Java でも実行するように記述されているため、スコープ・メモリー域に入るのは困難です。

```
final RadarPort radarPort = commsControl.getRadarPort();
EventPort eventPort = commsControl.getEventPort();

final IndirectRef<RTJavaRadar> radarRef = new IndirectRef<RTJavaRadar>();

// Immortal (永久メモリー域) 内に RTJavaRadar を作成します。
// このスレッドは NHRT です。
// このスレッドがスコープ・メモリー域内にあった場合、他のスレッドとの対話は
// より複雑になります。
ImmortalMemory.instance().enter(new Runnable() {
    public void run() {
        // Radar のリアルタイム・バージョン。
        radarRef.ref = new RTJavaRadar(radarPort, ImmortalMemory
                .instance());
    }
});

RTJavaRadar radarJava = radarRef.ref;
```

非同期イベント・ハンドラーの作成

非同期イベント・ハンドラーは、タイマー・イベント、またはスレッド以外で発生したイベント (例えば、アプリケーションのインターフェースからの入力) に反応し

ます。リアルタイム・システムでは、これらのイベントは、アプリケーションに設定した期限内に応答する必要があります。

始める前に

この例は、demo/realtime/sample_application.zip ファイル内にある RTJavaEventThread.java と RTJavaControlLauncher.java の各クラスに基づいています。

このタスクについて

このサンプル・アプリケーションでは、イベント・スレッドが衝突または着陸をシグナル通知する、シミュレーションからのイベントを待機します。このスレッドのリアルタイム・バージョンでは、AsyncEvent メカニズムが使用されています。これらのイベントは、適切な状況メッセージを出力したり、コントローラーを終了させたりするために使用します。

RTJavaEventThread には、2 つの非同期イベントが定義されています。これらのイベントには、ともにパラメーターはありません。

```
public class RTJavaEventThread extends RealtimeThread {

    private AsyncEvent landEvent = new AsyncEvent(),   Land
            crashEvent = new AsyncEvent();  Crash
```

これらのイベントでは、次のように 2 つの非同期イベント・ハンドラーを作成して登録します。

```
    /**
     * 着陸イベントが発生すると起動される実行可能オブジェクトを受け渡します。
     *
     * @param 着陸イベントが起動されると実行される実行可能コード。
     */
    public void addLandHandler(Runnable runnable) {
        AsyncEventHandler handler = new AsyncEventHandler(runnable);
        this.landEvent.addHandler(handler);
    }

    /**
     * 衝突イベントが発生すると実行される実行可能オブジェクトを受け渡します。
     *
     * @param 衝突イベントが起動されると実行される実行可能コード。
     */
    public void addCrashHandler(Runnable runnable) {
        AsyncEventHandler handler = new AsyncEventHandler(runnable);
        this.crashEvent.addHandler(handler);
    }
```

衝突または着陸のメッセージが受信されると、対応する非同期イベント・ハンドラーが起動され、Runnable オブジェクトがリリースされます。

```
                tag = this.eventPort.receiveTag();

                switch (tag) {
                case EventPort.E_CRSH:
                    // 衝突
                    this.crashEvent.fire();
                    this.running = false;
                    break;
                case EventPort.E_LAND:
                    // 着陸
```

```
            this.landEvent.fire();
            this.running = false;
            break;
        }
```

タスクの結果

RTJavaControlLauncher.java には、addLandHandler と addCrashHandler のメソッド呼び出しが含まれています。受け渡される Runnable オブジェクトによりメッセージがコンソールに出力され、制御スレッドは関連付けられた非同期イベント・ハンドラーが起動されると停止されます。これらが起動される時点については、RTJavaEventThread.java を参照してください。

```
// 着陸ハンドラー用に実行可能な AEH。
javaEventThread.addLandHandler(new Runnable() {
    public void run() {
        System.out.println("LAND!");
    }
});

// 衝突ハンドラー用に実行可能な AEH。
javaEventThread.addCrashHandler(new Runnable() {
    public void run() {
        System.out.println("CRASH!");
    }
});
```

NHRT スレッドの作成

非ヒープ・リアルタイム・スレッド (NHRT) を Java アプリケーションに追加するには、このチュートリアルを使用して独自にプログラムを作成するか、または変更します。

始める前に

この例は、demo/realtime/sample_application.zip ファイル内にある SimulationThread.java と SimLauncher.java の各クラスに基づいています。

このタスクについて

demo.sim.SimulationThread クラスは、デモ・アプリケーション内のシミュレーションの一部です。これは、実世界の代替物となることを目的としているため、システムのその他の部分からの中断なしで実行される可能性が高いです。このスレッドは、使用できる最高の優先度を持つ NoHeapRealtimeThread として作成され、このスレッドがガーベッジ・コレクションまたはシステム上の他のスレッドによって中断されないようにします。

SimulationThread では、次のコンストラクターで、スーパー・コンストラクター「NoHeapRealtimeThread(SchedulingParameters scheduling, MemoryArea area)」を呼び出した後で、その SchedulingParameters と ReleaseParameters を個別に設定します。

```
public SimulationThread(MemoryArea area, ControlPort controlPort,
          EventPort eventPort, RadarThread radarThread) {

        super(null, area);

        // 「this」を使用しているように、優先度を個々に設定します。
        // PriorityScheduler.MAX_PRIORITY は、
        // 推奨されなくなったことに注意してください。
```

```
        this.setSchedulingParameters(new PriorityParameters(PriorityScheduler
                .getMaxPriority(this)));

        ReleaseParameters releaseParms = new PeriodicParameters(null,
                new RelativeTime(period, 0)); // 20ms サイクル (50Hz)
        this.setReleaseParameters(releaseParms);

        // それぞれのスレッドを特定することがよい手法です。
        this.setName("SimulationThread");

        this.controlPort = controlPort;
        this.eventPort = eventPort;
        this.radarThread = radarThread;
    }
```

また、このシミュレーション内のその他のアクティブ・スレッドも非ヒープ・リアルタイム・スレッド (NHRT) として作成されますが、優先度は若干低めです。優先度の調整については、 79 ページの『リアルタイム・スレッドの作成』を参照してください。

このシミュレーションには、あるシミュレーションの完了後に再開するように、無期限に実行するオプションがあります。このシミュレーションは NHRT から構成されているため、ScopedMemory または ImmortalMemory を選択できます。サンプル・アプリケーションでは、シミュレーションに ScopedMemory が使用されています。これは、シミュレーションの完了時に割り振られた ScopeMemoryArea を抜けてから、ScopedMemory に再度入って次の実行を待機することが適切なためです。この場合、状態はある実行から次の実行へ持ち越されません。

ほとんどのクラスが NHRT セーフですが、ほとんどのクラスは NHRT セーフでないように実行できます。例えば、DatagramSockets が永久メモリー域または外部スコープ・メモリー域に保持さていた場合、それらはメモリー域をまたがるように設計されていないため、問題が発生する場合があります。サンプル・アプリケーションでは、このような問題を防止するために 1 つの ScopedMemory 域のみが使用されています。

RTSJ でのメモリー割り振り

RTSJ では、複数の方法でオブジェクトを特定のメモリー域に割り振ることができ、特定の時点でどの方法を選択すればよいのかが常に明らかであるとは限りません。

それぞれの方法に特性があり、その特性は RTSJ の実装によって異なり、パフォーマンスまたは最終的なメモリー占有スペースのいずれかに影響を与えます。このセクションでは、使用可能なオプションについての概要を説明し、各オプションがオブジェクトの割り振りで最適な選択肢となる可能性がある場合を示します。

静的イニシャライザー

オブジェクトを永久メモリー域に割り振る最も単純な方法は、静的イニシャライザーへの割り振りです。利点として、メモリー・コンテキストを変更する問題に対処する必要がありませんが、このパターンが適切な状況は非常に限られています。この方法は、消費永久メモリー量が、オブジェクト自体に必要な量に限定される点で効率的です。

MemoryArea.newInstance(Class c)

スレッドがメモリー・コンテキストにあり、オブジェクトを別の領域 (スレッドのスコープ・スタックに既に存在する必要がある) に割り振る場合に、この方法は単純明快です。利点として、インスタンス化するクラスにアクセスするだけで済む点が挙げられますが、newInstance メソッドは適切なコンストラクターを作成する必要があります。このパターンは、特定のクラスのオブジェクトをたまに割り振る必要がある場合に最適ですが、それ以外の場合には、メモリー使用量が多くなる傾向があります。

MemoryArea.newInstance(Constructor c, Object[] args)

これも、スレッドがメモリー・コンテキストにあり、オブジェクトを別のコンテキスト (スレッドのスコープ・スタックに既に存在する必要がある) に割り振る場合に、単純な方法になります。この場合、コンストラクターおよびいくつかの引数を渡す必要があり、コンストラクターが現在のメモリー・コンテキストで有効であることをユーザーが保証する必要があります。 newInstance メソッドはコンストラクターを作成する必要がないため、メモリー使用量は newInstance(Class c) よりも少なくなります。そのため、このパターンは、オブジェクトを割り振る頻度がより高い場合により適しており、コンストラクターを前もって割り振って、ImmortalMemory などに保管する犠牲を払う価値があると思われます。

MemoryArea.enter(Runnable r) に続けて new <class>()

この方法は、特定の MemoryArea を割り振り用の新規デフォルトにして、リフレクションおよび付随するコンストラクター・オブジェクトが必要なくなるようにします。そのため、オブジェクト自体以外に追加のメモリー使用が発生しないため、多くのオブジェクトを作成する場合に最適な方法となります。この方法は、どのスレッドのスコープ・スタックでも目的の領域がアクティブでない場合にのみうまくいきます。一般的に、Runnable または静的/インスタンス・フィールドのどちらかでパラメーターを渡す必要があるため、 Runnable メモリー域を作成する要件により、newInstance を使用するよりもこの方法は複雑になります。

MemoryArea.executeInArea(Runnable r) に続けて new <class>()

この方法も、特定の MemoryArea を割り振りようの新規デフォルトにして、リフレクションおよび付随するコンストラクター・オブジェクトが必要なくなるようにします。そのため、オブジェクト自体以外に追加のメモリー使用が発生しないため、多くのオブジェクトを作成する場合に最適な方法となります。この方法は、現在のスレッドのスコープ・スタックに目的の領域が既に存在しており、それゆえ MemoryArea.enter よりフレキシブルな場合にのみ使用できます。一般的に、Runnable または静的/インスタンス・フィールドのどちらかでパラメーターを渡す必要があるため、 Runnable を作成する要件により、newInstance を使用するよりもこの方法は複雑になります。

Class.newInstance()

この方法は、新規インスタンスを現在のメモリー域に作成するため、MemoryArea.enter または executeInArea のどちらかと組み合わせて使用する必要があります。オブジェクト自体以外に追加のメモリー使用は発生しません。

高解像度タイマーの使用

リアルタイム・クロックは、標準 JVM に関連したクロックよりも高い精度を備えています。

始める前に

この例は、demo/realtime/sample_application.zip ファイル内の RTJavaRadar.java クラスに基づいています。

このタスクについて

通常の Java では、クロックおよびタイマーを処理する機能が制限されています。 Real-Time Specification for Java では、ナノ秒精度を備え、実時間に十分な大きさの絶対時刻を実現しています。 javax.realtime.HighResolutionTime およびそのサブクラスを使用して、2 つのコンポーネント (ミリ秒およびナノ秒) を使用して時刻を表します。

WebSphere Real Time for RT Linux は、基盤オペレーティング・システムのサポートを使用して、高解像度時刻を提供します。現在の Linux カーネルは、最大で 4 ミリ秒が保証された精度のクロックを提供しています。 WebSphere Real Time for RT Linux で提供される Linux パッチは、 1 マイクロ秒に近い精度のクロックを提供します。

RTJavaRadar クラスは、以下のように高解像度タイマーの使用データを示します。

- **1** はリアルタイム・クロックを取得します。
- **2** は現在の絶対時刻を取得します。
- **3** は時刻のナノ秒コンポーネントを取得します。リアルタイム・クロックの正確性とは、ナノ秒の使用が妥当であるという意味です。
- **4** は ping 前後の時刻を取得します。
- **5** は着陸船のディセント速度を返します。
- **6** は、スレッドを 5 ミリ秒待機させてから別の反復を実行します。

```
public void run() {
       // 以下のオブジェクトは前もって作成されており、各反復で
       // 再使用されます。
       Clock rtClock = Clock.getRealtimeClock();          1
       AbsoluteTime time = rtClock.getTime();             2

       try {
           double height = 0.0, lastheight;
           long millis = time.getMilliseconds(), lastmillis;
           long nanos = time.getNanoseconds(), lastnanos;     3

           while (this.running) {

               lastmillis = millis;
               lastnanos = nanos;
               lastheight = height;

               // time = rtClock.getTime() 形式を使用するのではなく、
               // このメソッドは、既存の AbsoluteTime オブジェクトの
               // 値を置き換えます。
               rtClock.getTime(time);                 4
               millis = time.getMilliseconds();
               nanos = time.getNanoseconds();
```

```
            // ping を送信して pong を受け取るのにかかる時間を
            // 計測します。
            this.radarPort.ping();

            rtClock.getTime(time);                    4

            height = (time.getMilliseconds() - millis)
                    / demo.sim.RadarThread.timeScale;
            height += ((time.getNanoseconds() - nanos) / 1.0e6)      5
                    / demo.sim.RadarThread.timeScale;

            double difference = ((double) (millis - lastmillis)) / 1.0e3
                    + ((double) (nanos - lastnanos)) / 1.0e9;
            double speed = (height - lastheight) / difference;

            this.myHeight = height;
            this.mySpeed = speed;

            try {
                sleep(5);                         6
            } catch (InterruptedException e) {
                // これは重要ではありません。
            }
        }
```

前のコードは、JavaRadar クラスの以下の標準 JVM コードと比較できます。

```
public void run() {
        try {
            double height = 0.0, lastheight;

            long nanos = System.nanoTime(), lastnanos;
            while (this.running) {
                /* x ミリ秒ごとに高さを設定 */
                Thread.sleep(5);
                lastnanos = nanos;
                lastheight = height;

                nanos = System.nanoTime();

                this.radarPort.ping();

                // 時間目盛りは、ミリ秒当たりの高さの単位
                height = ((System.nanoTime() - nanos) / 1.0e6)
                        / demo.sim.RadarThread.timeScale;

                double speed = (height - lastheight)
                        / (((double) (nanos - lastnanos)) / 1.0e9);

                this.myHeight = height;
                this.mySpeed = speed;
            }
```

サンプル・アプリケーション

サンプル・アプリケーションで一連の例を使用して、Java プログラムのリアルタイム特性を改善するために使用可能な WebSphere Real Time for RT Linux のフィーチャーを例示します。

サンプル・アプリケーションのソース・ファイルは、demo/realtime/sample_application.zip ファイルに含まれています。

サンプルは以下の 2 つの主コンポーネントから成ります。

- 月着陸船の単純な例である**シミュレーション**。位置は、地上の高さおよび着陸エリアからの距離で定義されます。 89 ページの図 6を参照してください。

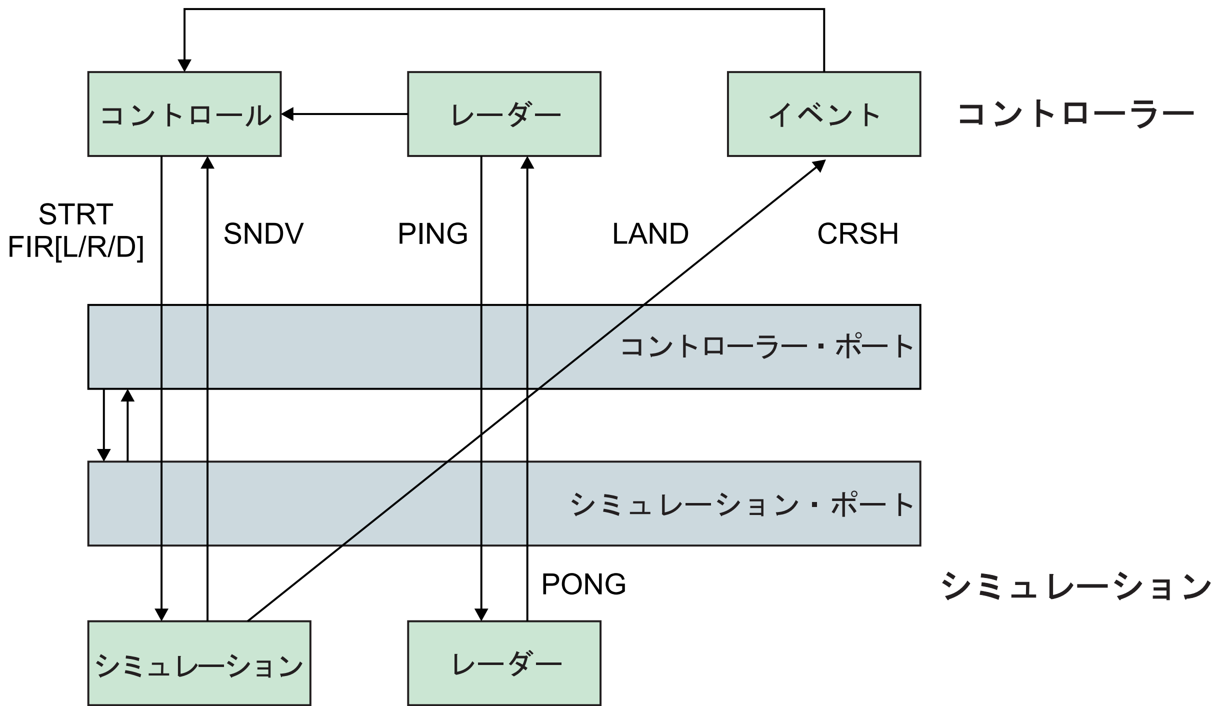
  シミュレーション・クラスは 非ヒープ・リアルタイム・スレッド (NHRT) を使用して作成され、この資料ではそれ以上変更しません。
- コマンドをシミュレーションに送信する**コントローラー**。これはレーダー ping を送信して、着陸船の高さを判別し、この情報に基づいて着陸船のディセント・レートを制御します。また、コントローラーは、情報ストリームを着陸船から受け取ります (例えば、着陸船の着陸エリアからの距離)。

  コントローラーは、標準 Java でまず作成されます。 78 ページの『Java アプリケーションの変更』でこれを発展させて、リアルタイム Java プログラムにします。

着陸の結果によって、コントローラーには 2 つのメッセージ (衝突または着陸) のいずれかが送信されます。

サンプル・アプリケーションを使用して、以下の操作を実行できます。
- シミュレーションとコントローラーの両方を一緒に実行して、一緒に実行されるリアルタイムおよび標準の Java クラスの組み合わせを例示する。詳しくは、 90 ページの『サンプル・アプリケーションのビルド』および 90 ページの『サンプル・アプリケーションの実行』を参照してください (サンプル・アプリケーションからの期待される出力についても確認できます)。

  注: LaunchBoth クラスを使用して、シミュレーションとコントローラーを同時に開始できます。
- Metronome ガーベッジ・コレクター と標準ガーベッジ・コレクターを使用した場合の差異を比較する。詳しくは、 90 ページの『Real Time なしでのサンプル・アプリケーションの実行』および 92 ページの『Metronome ガーベッジ・コレクターを使用したサンプル・アプリケーションの実行』を参照してください。
- Ahead-Of-Time (AOT) コンパイラー。詳しくは、 93 ページの『AOT を使用したサンプル・アプリケーションの実行』を参照してください。



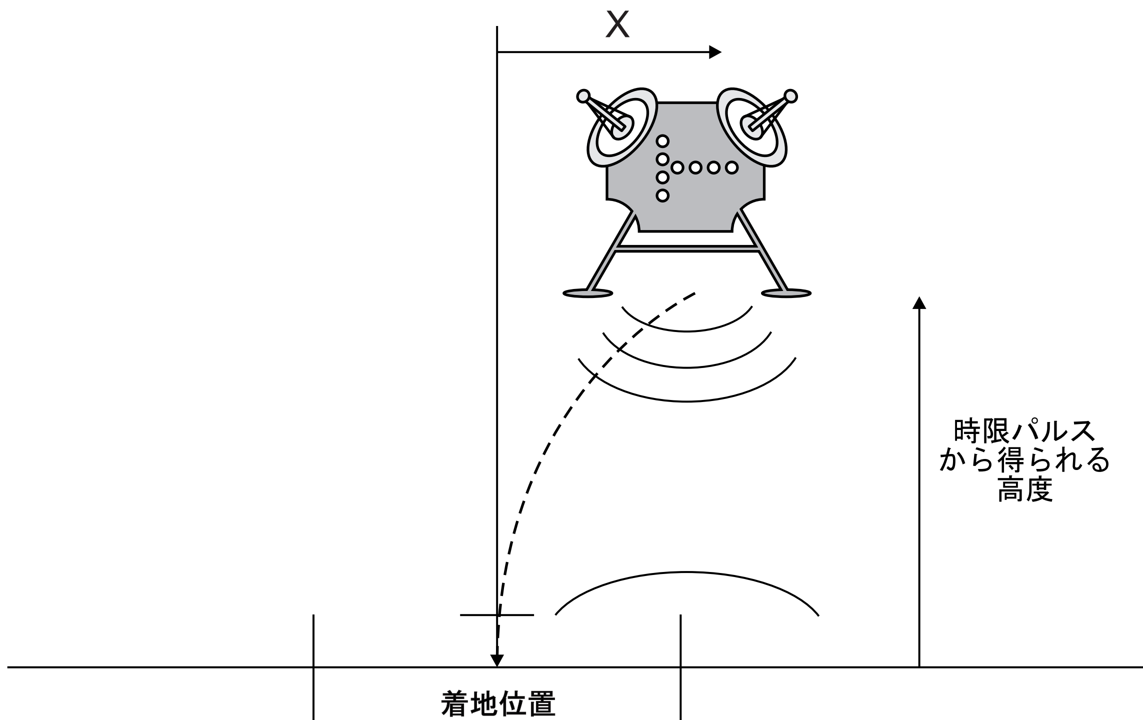


図 6. 月着陸船のダイアグラム

このダイアグラムは、サンプル提供のモジュールの関係を示します。ダイアグラムの上部は、コントローラーとシミュレーターを示しています。コントローラーには

3 つのスレッド (制御、レーダー、およびイベントのスレッド) があります。シミュレーターには 2 つのスレッド (シミュレーターとレーダーのスレッド) があります。ダイアグラムの下部は月着陸船を示しており、左右 2 つの制御機能、および着陸船の高さを判別するパルスが示されています。

サンプル・アプリケーションのビルド

サンプル・アプリケーションのソース・コードはガイダンスとして提供されています。実行できるようにするための準備として、Java ソース・コードを解凍してコンパイルする必要があります。

手順

1. 作業ディレクトリーを作成します。
2. 以下のように、サンプル・アプリケーションを作業ディレクトリーに解凍します。
   ```
   unzip sample_application.zip
   ```
3. 以下のように、出力用の新規ディレクトリーを作成します。
   ```
   mkdir classes
   ```
4. ソースをコンパイルします。
   a. 以下のように、ファイルのリストを生成します。
   ```
   find -name "*.java" > source
   ```
   b. 以下のように、ソースをコンパイルします。
   ```
   javac -Xrealtime -Xlint:deprecated -g -d classes @source
   ```
   c. 以下のように、クラス・ファイルの jar ファイルを作成します。
   ```
   jar cf demo.jar -C classes/ .
   ```

次のタスク

これで、サンプル・アプリケーションを実行できます。

サンプル・アプリケーションの実行

WebSphere Real Time には、標準 JVM に加えてリアルタイム JVM (**-Xrealtime** コマンド行引数で開始) が用意されています。

サンプル・アプリケーションには、別々の JVM で実行されるように設計されている以下の 2 つのコンポーネントが含まれています。

- リアルタイム Java でのみ実行されるシミュレーション。
- 非リアルタイムとリアルタイムのどちらの Java でも実行可能なコントローラー。

コントローラー・コードを各種モードで実行して、 IBM Real-Time Java テクノロジーの長所を例示します。

Real Time なしでのサンプル・アプリケーションの実行

この手順では、IBM WebSphere Real Time を利用せずにサンプル・アプリケーションを実行します。

始める前に

サンプル・アプリケーションを実行するには、まずサンプル・ソース・コードをビルドする必要があります。詳しくは、 90 ページの『サンプル・アプリケーションのビルド』を参照してください。

手順

1. 以下のように、シミュレーションを開始します。

   ```
   java -Xrealtime -classpath ./demo.jar -Xgc:scopedMemoryMaximumSize=11m
   demo.sim.SimLauncher <port>
   ```

   このコマンドで、*<port>* はワークステーションの割り振り解除ポートです。
2. 以下のように、コントローラーを開始します。

   ```
   java -classpath ./demo.jar -mx300m demo.controller.JavaControlLauncher <host>
   <port>
   ```

   このコマンドで、 *<host>* はシミュレーションを実行しているワークステーションのホスト名、 *<port>* は前のステップで指定されたポートです。

タスクの結果

このアプリケーションは、シミュレーションおよびコントローラーが開始したことを示すメッセージを生成します。

```
SimLauncher: Waiting for connections...
Starting control thread...
```

コントローラー内の値の一部のポイント・サンプルが、以下のようにコンソールに出力されます。

```
x=99.50, radar=199.11, y=198.34, vx=-0.71, vy=-0.43, timeSinceLast=0.19, targetVx=-6.01, targetVy=-9.00
x=95.50, radar=194.59, y=192.70, vx=-2.70, vy=-2.43, timeSinceLast=0.20, targetVx=-5.94, targetVy=-9.00
x=87.50, radar=186.57, y=183.06, vx=-4.70, vy=-4.40, timeSinceLast=0.20, targetVx=-5.77, targetVy=-9.00
x=76.46, radar=172.84, y=169.42, vx=-5.42, vy=-6.75, timeSinceLast=0.20, targetVx=-5.60, targetVy=-9.00
x=65.36, radar=155.58, y=151.84, vx=-5.50, vy=-9.19, timeSinceLast=0.20, targetVx=-5.57, targetVy=-9.00
x=54.36, radar=138.06, y=135.24, vx=-5.44, vy=-7.63, timeSinceLast=0.20, targetVx=-5.56, targetVy=-9.00
x=43.26, radar=120.57, y=117.22, vx=-5.67, vy=-9.62, timeSinceLast=0.20, targetVx=-5.52, targetVy=-9.00
x=32.36, radar=103.60, y=100.72, vx=-5.47, vy=-9.06, timeSinceLast=0.20, targetVx=-5.43, targetVy=-9.00
x=21.52, radar=84.60, y=82.86, vx=-5.32, vy=-9.09, timeSinceLast=0.20, targetVx=-5.60, targetVy=-9.00
x=10.72, radar=67.07, y=65.56, vx=-5.30, vy=-10.54, timeSinceLast=0.20, targetVx=-5.65, targetVy=-9.00
x=0.76, radar=51.08, y=49.78, vx=-4.30, vy=-7.52, timeSinceLast=0.20, targetVx=-0.50, targetVy=-9.00
x=-5.24, radar=37.07, y=35.94, vx=-2.30, vy=-8.26, timeSinceLast=0.20, targetVx=0.50, targetVy=-9.00
x=-7.24, radar=20.05, y=19.90, vx=-0.30, vy=-6.15, timeSinceLast=0.20, targetVx=0.50, targetVy=-9.00
x=-6.36, radar=2.68, y=2.80, vx=0.27, vy=-10.08, timeSinceLast=0.20, targetVx=0.50, targetVy=-9.00
```

シミュレーションが停止する直前に、イベント要約メッセージが発行されます。

```
Fire down transitions 141, fire horizontally transitions 141
LAND!
```

ポイント・サンプルとイベント要約メッセージに加えて、コントローラーは、同じディレクトリー内に graph.svg というグラフを生成します。このグラフにはポイント・サンプルのプロットが含まれています。 このグラフは、標準の非リアルタイム JVM を使用してアプリケーションを実行している場合の、JavaRadar スレッドに対するガーベッジ・コレクション休止の効果を示しています。レーダーの高さを表すデータにはスパイクがあります。スパイクは、コントローラー・アプリケーション

に影響する標準のガーベッジ・コレクションの休止によって引き起こされます。一部の実行では、ガーベッジ・コレクションの休止が長すぎて障害を引き起こし、以下のメッセージが表示されます。

```
CRASH!
```

ガーベッジ・コレクションによって生じた休止時間を表示するには、コントローラーの起動コマンドに **-verbose:gc** オプションを追加します。

```
java -classpath ./demo.jar -verbose:gc -mx300m demo.controller.JavaControlLauncher
<host> <port>
```

Metronome ガーベッジ・コレクターを使用したサンプル・アプリケーションの実行

-Xrealtime オプションを追加することで、コードを書き換えずに、標準の Java アプリケーションをリアルタイム環境で実行できます。このオプションは、リアルタイム Java 言語フィーチャーと Metronome ガーベッジ・コレクターをともに使用可能にします。

始める前に

サンプル・アプリケーションを実行するには、まずサンプル・ソース・コードをビルドする必要があります。詳しくは、 90 ページの『サンプル・アプリケーションのビルド』を参照してください。

手順

1. 以下のように、シミュレーションを開始します。

   ```
   java -Xrealtime -classpath ./demo.jar -Xgc:scopedMemoryMaximumSize=11m
   demo.sim.SimLauncher <port>
   ```

   このコマンドで、*<port>* はワークステーションの割り振り解除ポートです。
2. 以下のように、コントローラーを開始します。

   ```
   java -Xrealtime -classpath ./demo.jar -mx300m
   demo.controller.JavaControlLauncher <host> <port>
   ```

   このコマンドで、 *<host>* はシミュレーションを実行しているワークステーションのホスト名、 *<port>* は前のステップで指定されたポートです。同じワークステーションで両方の JVM を実行すると、振る舞いの決定性が低下する可能性があります。詳しくは、 26 ページの『考慮事項』を参照してください。

タスクの結果

アプリケーションが実行され、次のような出力が生成されます。

1. シミュレーションおよびコントローラーが開始したことを示すメッセージ。
2. コントローラー値のポイント・サンプル。
3. ポイント・サンプルのプロットを含む、同じディレクトリー内の `graph.svg` というグラフ。
4. イベント要約メッセージ。

Metronome ガーベッジ・コレクションを使用してアプリケーションを実行した場合、ポイント・サンプルと対応するグラフは以下の状況を示す傾向があります。

- レーダーの高さのデータにスパイクがない。
- 実際の高さのデータを正確に辿っている。

これは、ガーベッジ・コレクションによる休止が短時間の状態で、コントローラーのコードが実行されているためです。

Metronome ガーベッジ・コレクションの休止は頻繁にありますが、休止期間は通常 1 ミリ秒未満です。非リアルタイム・ガーベッジ・コレクションの休止は少なめですが、通常は数十から数百ミリ秒続きます。これらの休止の差異は、**-verbose:gc** オプションをコントローラーの実行コマンドに追加することで確認できます。

詳細ガーベッジ・コレクション出力について詳しくは、 145 ページの『verbose:gc 情報の使用』を参照してください。

AOT を使用したサンプル・アプリケーションの実行

この手順では、AOT (Ahead-Of-Time) コンパイラーを使用して、コードを書き換えることなく、標準 Java アプリケーションをリアルタイム環境で実行します。このサンプルを使用して、JIT (Just-In-Time) コンパイラーを使用した場合の同一アプリケーションの実行と比較します。

Ahead-Of-Time コンパイルについて詳しくは、 40 ページの『WebSphere Real Time for RT Linux でのコンパイル済みコードの使用』を参照してください。

始める前に

サンプル・アプリケーションを実行するには、まずサンプル・ソース・コードをビルドする必要があります。詳しくは、 90 ページの『サンプル・アプリケーションのビルド』を参照してください。

このタスクについて

AOT (Ahead-Of-Time) コンパイラーは、Java アプリケーションを実行前にネイティブ・コードにコンパイルします。 Just-In-Time (JIT) コンパイルによって生じる中断がないので、アプリケーションがどのように実行されるかをより正確に予測できます。

手順

1. アプリケーション・バイトコードをネイティブ・コードに変換します。
   a. この変換は、最初にサンプルを通常の JIT (Just-In-Time) コンパイラーで実行することによって行われます。

   ```
   java -Xrealtime -Xjit:verbose={precompile},vlog=./sim.aotOpts ¥
        -classpath ./demo.jar -Xgc:scopedMemoryMaximumSize=11m
        demo.sim.SimLauncher <port>
   ```

   このコマンドで、*<port>* はワークステーションの割り振り解除ポートです。

   b. 別のウィンドウで、アプリケーションを実行します。

   ```
   java -Xrealtime -Xjit:verbose={precompile},vlog=./control.aotOpts ¥
        -classpath ./demo.jar -Xmx300m demo.controller.JavaControlLauncher
        localhost <port>
   ```

   このコマンドで、*<port>* は前のステップで指定したポートです。 この結果、アプリケーションの出力は以下のメッセージのようになります。

```
Fire down transitions 141, fire horizontally transitions 141
```

および

```
Land!
```

c. 前のステップで作成した AOT オプション・ファイルを結合します。

```
cat sim.aotOpts.20081014.234958.13205 control.aotOpts.20081014.234958.13205
> sample.aotOpts
```

前のステップで作成されたログ・ファイルで使用される名前には、ファイル名に日付とプロセス ID が付加されています。ファイル名の形式は、**vlog=** オプションで指定されます。例えば、**vlog=sim.aotOpts** の場合、**sim.aotOpts.20081014.234958.13205** のようなファイル名が生成されます。

d. realtime.jar、vm.jar、rt.jar、およびアプリケーション demo.jar に含まれている sample.aotOpts ファイル内のファイルをコンパイルします。共有クラス・キャッシュを使用する場合は、そのキャッシュの名前は 53 文字以下にする必要があります。

```
admincache -Xrealtime -populate -cacheName "sample" -aotFilterFile
sample.aotOpts -classpath ./demo.jar ¥
$JAVA_HOME/jre/lib/i386/realtime/jclSC160/vm.jar ¥
$JAVA_HOME/jre/lib/i386/realtime/jclSC160/realtime.jar ¥
$JAVA_HOME/jre/lib/rt.jar ¥
./demo.jar
```

以下のようなコンパイル結果が報告されます。

```
J9 Java(TM) admincache 1.0
Licensed Materials - Property of IBM

(c) Copyright IBM Corp. 1991, 2008 All Rights Reserved
IBM is a registered trademark of IBM Corp.
Java and all Java-based marks and logos are trademarks or registered
trademarks of Oracle Corporation

JVMSHRC256I Persistent shared cache "sample" has been destroyed
Converting files
Converting /team/mstoodle/demo/sdk/jre/lib/i386/realtime/jclSC160/vm.jar into shared class cache
Succeeded to convert jar file /team/mstoodle/demo/sdk/jre/lib/i386/realtime/jclSC160/vm.jar
Converting /team/mstoodle/demo/sdk/jre/lib/i386/realtime/jclSC160/realtime.jar into shared class cache
Succeeded to convert jar file /team/mstoodle/demo/sdk/jre/lib/i386/realtime/jclSC160/realtime.jar
Converting /team/mstoodle/demo/sdk/jre/lib/rt.jar into shared class cache
Succeeded to convert jar file /team/mstoodle/demo/sdk/jre/lib/rt.jar
Converting /team/mstoodle/demo/demo.jar into shared class cache
Succeeded to convert jar file /team/mstoodle/demo/demo.jar

Processing complete
```

注: 行「

```
JVMSHRC256I Persistent shared cache "sample" has been destroyed
```

」は、このコマンドによって「sample」という既存のすべてのキャッシュが破棄されて、指定したキャッシュが作成されることを意味します。

e. 以下のように、データが入力されているキャッシュのコンテンツを表示します。

```
admincache -Xrealtime -cacheName "sample" -printStats
```

2. 以下のように、シミュレーションを開始します。

```
java -Xrealtime -Xnojit -Xmx300m -Xshareclasses:name="sample" ¥
     -classpath ./demo.jar -Xgc:scopedMemoryMaximumSize=11m ¥
     demo.sim.SimLauncher <port>
```

このコマンドで、*<port>* はこのワークステーションの割り振り解除ポートです。

3. 以下のように、コントローラーを開始します。

```
java -Xrealtime -Xnojit -Xmx300m -Xshareclasses:name="sample" ¥
     -classpath ./demo.jar ¥
     demo.controller.JavaControlLauncher <host> <port>
```

このコマンドで、 *<host>* はシミュレーションを実行しているワークステーションのホスト名、 *<port>* は前のステップで指定されたポートです。同じワークステーションで両方の JVM を実行すると、振る舞いの決定性が低下する可能性があります。詳しくは、 26 ページの『考慮事項』を参照してください。

タスクの結果

アプリケーションが実行され、次のような出力が生成されます。

1. シミュレーションおよびコントローラーが開始したことを示すメッセージ。
2. コントローラー値のポイント・サンプル。
3. ポイント・サンプルのプロットを含む、同じディレクトリー内の `graph.svg` というグラフ。
4. イベント要約メッセージ。

Ahead-of-Time コンパイルを使用してアプリケーションを実行した場合、ポイント・サンプルと対応するグラフは以下の状況を示す傾向があります。

- レーダーの高さのデータにスパイクがない。
- 実際の高さのデータを正確に辿っている。

これは、コントローラーのコードは、ガーベッジ・コレクションによる休止が短時間で、JIT コンパイルによって中断されずに実行されているためです。

共有クラス・キャッシュを使用してこのアプリケーションを実行する利点は、コントローラー JVM およびシミュレーション JVM が、両方の JVM でロードされるクラスが使用するメモリーの一部を共有する点にあります。

サンプル・リアルタイム・ハッシュ・マップ

WebSphere Real Time for RT Linux には、IBM SDK for Java 7 の標準の HashMap よりも、put メソッドにおいてより一貫したパフォーマンスを提供する、HashMap と HashSet の実装が含まれています。

IBM 提供の標準 java.util.HashMap は、高スループット・アプリケーションに適しています。また、拡張する必要があるハッシュ・マップの最大サイズが分かっているアプリケーションにも役立ちます。可変サイズに拡張可能なハッシュ・マップが必要なアプリケーションの場合、使用に応じて、標準ハッシュ・マップではパフォーマンス上の問題が生じる可能性があります。標準ハッシュ・マップは、put メソッドを使用して新規項目をハッシュ・マップに追加する際の応答時間が優れています。ただし、ハッシュ・マップが満杯になると、より大きな補助ストレージを割り振る

必要があります。このため、現在の補助ストレージ内の項目をマイグレーションする必要が生じます。ハッシュ・マップが大きい場合、put の実行時間も長くなる可能性があります。例えば、操作に数ミリ秒かかることがあります。

WebSphere Real Time for RT Linux には、サンプル・リアルタイム・ハッシュ・マップが含まれています。これは、標準 java.util.HashMap と同じ機能のインターフェースを提供しますが、はるかに一貫性のある put メソッドのパフォーマンスを可能にします。補助ストレージを作成して、ハッシュ・マップが満杯になったときにすべての項目をマイグレーションする代わりに、サンプル・ハッシュ・マップは追加の補助ストレージを作成します。この新しい補助ストレージは、ハッシュ・マップ内のその他の補助ストレージにチェーニングされます。チェーニングにより、初期段階では、空の補助ストレージが割り振られてその他の補助ストレージにチェーニングされる間、若干のパフォーマンスの低下が生じます。バッキング (補助ストレージへの保管) ハッシュ・マップが更新されると、すべての項目をマイグレーションする必要がある場合よりも高速になります。リアルタイム・ハッシュ・マップの欠点として、get、put、および remove の各操作が若干遅くなります。操作が遅くなるのは、ルックアップごとに、一回だけではなく、バッキング・ハッシュ・マップのセット全体を探索する必要があるためです。

リアルタイム・ハッシュ・マップを試行するには、`RTHashMap.jar` ファイルを、ブート・クラスパスの先頭に追加します。 WebSphere Real Time for RT Linux を `$WRT_ROOT` ディレクトリーにインストールしている場合は、以下のオプションを追加して、標準ハッシュ・マップの代わりにリアルタイム・ハッシュ・マップをアプリケーションで使用します。

```
-Xbootclasspath/p:$WRT_ROOT/demo/realtime/RTHashMap.jar
```

リアルタイム・ハッシュ・マップ実装のソース・ファイルおよびクラス・ファイルは、 `demo/realtime/RTHashMap.jar` ファイル内に含まれています。また、リアルタイム java.util.LinkedHashMap および java.util.HashSet 実装も提供されています。

Eclipse を使用した WebSphere Real Time for RT Linux アプリケーションの開発

Eclipse を使用すると、リアルタイム・アプリケーションの開発時に豊富な機能が用意された IDE が提供されます。

始める前に

Eclipse のアプリケーション開発環境を使用してリアルタイム・アプリケーションを今回初めて開発する場合、以下の手順を使用して環境を構成してください。

WebSphere Real Time for RT Linux には、Oracle の標準の **javac** コンパイラーが用意されています。使用するコンパイラーに制限はありませんが、そのコンパイラーで有効な Java 5.0 のクラス・ファイルを生成する必要があります。ただし、javax.realtime.* Java クラスがビルド・パスに存在している必要があります。

このタスクについて

Eclipse でアプリケーションを開発するには、以下の手順に従ってください。

手順

1. http://www.eclipse.org/downloads/ から Eclipse をダウンロードします。 Java 5.0 の正しいコンパイルのためには、Eclipse 3.1.2 を使用することをお勧めします。
2. Eclipse を実行するために、IBM SDK and Runtime Environment for Linux platforms, Java 2 Technology Edition, Version 5.0 準拠の JVM をダウンロードします。
3. WebSphere Real Time for RT Linux パッケージから `opt/IBM/javawrt3/jre/lib/i386/realtime/jclSC160/realtime.jar` ファイルを解凍します。
4. Eclipse を開き、プロジェクトを作成します。**「ファイル」** > **「新規」**とクリックします。 「**新規プロジェクト**」パネルから「**Java プロジェクト**」を選択します。
5. 「**次へ**」をクリックして、「**新規 Java プロジェクト**」パネルを表示します。
   a. プロジェクト名を入力します (例えば、RTSJ-Tests)。
   b. JDK コンパイラーが 5.0 に設定されていることを確認します。
6. 「**終了**」をクリックします。
7. 作業ディレクトリーを作成して、`opt/IBM/javawrt3/jre/lib/i386/realtime/jclSC160/realtime.jar` ファイルをインポートします。
8. **「ファイル」** > **「新規」** > **「フォルダー」**とクリックして、「**新規フォルダー**」パネルを開きます。新規フォルダー名を入力します (例えば、*deplib*)。
9. 「**終了**」をクリックします。
10. `realtime.jar` をインポートするには、**「ファイル」** > **「インポート」**とクリックして「**インポート**」パネルを開きます。
11. 「**ファイル・システム**」をクリックしてから「**次へ**」をクリックします。
12. JVM が解凍されたファイル・システムで `opt/IBM/javawrt3/jre/lib/i386/realtime/jclSC160/` ディレクトリーを開きます。
13. realtime.jar ファイルの横にあるチェック・ボックスを選択し、インポートするフォルダー (`RTSJ-Tests/deplib` など) を指定して、「**選択したフォルダーのみを作成 (Create selected folders only)**」オプションが選択されていることを確認します。
14. 「**終了**」をクリックします。
15. jar ファイルをライブラリー・パスに追加します。プロジェクトを右クリックし、「**プロパティー**」をクリックして「**プロパティー**」パネルを開きます。
16. 「**Java のビルド・パス**」と「**ライブラリー**」タブをクリックします。「**JAR の追加**」をクリックします。
17. プロジェクト・ディレクトリーの下の **realtime.jar** をクリックします。「**OK**」をクリックします。

タスクの結果

この手順が正常に実行されると、「**ライブラリー**」タブの .jar ファイルのリストに realtime.jar ファイルが表示されます。

例

Eclipse では、realtime.src.jar を使用して RTSJ クラスに追加情報を表示することができます。これを行うには、インポートされた realtime.jar ファイルの「プロパティー」ウィンドウを開き、「**Java ソースの添付**」をクリックし、realtime.src.jar ファイルの場所を「**ロケーション・パス:**」に入力します。

次のタスク

Eclipse で Apache Ant を使用してアプリケーションをビルドする場合は、Ant ビルド・スクリプトのクラス・パスに realtime.jar ファイルを追加します。例えば、次のようにします。

```
<property name="rtsj.src" location="." />
<property name="rtsj.deplib" location="deplib" />
<property name="rtsj.jar.dir" location="build/rtsj-jar.dir" />

<!-- このパッケージ用の .class ファイルを生成する -->
<target name="compile" depends="init">
<javac destdir="${rtsj.jar.dir}"
srcdir="${rtsj.src}"
target="1.5"
classpath="${rtsj.deplib}/realtime.jar:${rtsj.src}"
debug="true"/>
</target>
```

これは、Ant ビルド・スクリプトのほんの一部です。

アプリケーションのデバッグ

Eclipse Application Developer を使用して、ローカルまたはリモート側でアプリケーションをデバッグできます。

このタスクについて

リアルタイム・アプリケーションをリモート側でデバッグするには、デバッグ対象の JVM に以下のオプションが必要になります。

```
-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=10100
```

手順

1. アプリケーションが実行されている Linux 環境で、以下を入力します。

   ```
   java -Xrealtime -agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=10100
   ```

   ここで、

   - server=y は、JVM がデバッガーからの接続を受け入れることを指示します。
   - suspend=y により、JVM はデバッガーが接続するまで待機してから実行します。
   - address=10100 は、デバッガーが JVM に接続するポート番号です。この番号は通常 1024 を超えた数値です。

   JVM は以下のメッセージを表示します。

   ```
   Listening for transport dt_socket at address: 10100
   ```

2. Eclipse でアプリケーションを開いて、「**デバッグ**」を選択します。

3. リモート・アプリケーションをデバッグするための新規構成を作成する必要があります。同じプロジェクト内のアプリケーションが実行され、それぞれの実行で同じポートを listen する場合は、構成を 1 つ作成するだけで構いません。
4. 構成の作成後は、構成の名前、デバッグするアプリケーションが入っているプロジェクトの名前、アプリケーションが実行されているワークステーションの *hostname*、および **-agentlib** オプションで渡したポート番号を入力します。
5. 「**デバッグ**」をクリックして、デバッグ・セッションを開始します。「**デバッグ**」パースペクティブが開き、リモート側でデバッグされている JVM の状態が表示されます。

JVM を使用した Eclipse の実行

このセクションでは、WebSphere Real Time for RT Linux JVM を使用して Eclipse を実行する方法について説明します。

JVM を使用して Eclipse を実行するには、eclipse コマンドに以下を指定する必要があります。

- 使用する WebSphere Real Time for RT Linux JVM の java 実行可能プログラムへの完全修飾ディレクトリー
- **-Xrealtime** JVM オプション
- Eclipse で使用する永久メモリーのサイズ。これは少なくとも 128M にする必要があります。

JVM を使用して Eclipse を実行する例:

```
eclipse -vm $JAVA_HOME/jre/bin/java -vmargs -Xrealtime -Xgc:immortalMemorySize=128M
```

注: Eclipse SDK は、WebSphere Real Time for RT Linux アプリケーションで使用可能な各種リアルタイム・メモリー・オプションを使用しません。この結果、特に何時間あるいは何日も再始動せずに Eclipse を使用した場合、永久メモリーが枯渇します。 **OutOfMemory** エラーが発生した場合は、**-Xgc:immortalMemorySize** オプションに指定する値を大きくして、 Eclipse で使用する永久メモリーの量を増加させることができます。

第 7 章 パフォーマンス

WebSphere Real Time for RT Linux は、最高のスループット・パフォーマンスまたは最小のメモリー・フットプリントを実現させるためでなく、GC による休止を一貫して短くするために最適化されています。

ハイパースレッディングがサポートされるシステムでは、ハイパースレッディングを有効にしないようにする必要があります。これは、WebSphere Real Time for RT Linux を使用する際のパフォーマンスへの悪影響を避けるためです。

タイミングの変動性を軽減するため、および Real-Time Specification for Java (RTSJ) をサポートするために、いくつかの標準 IBM Java ランタイム最適化を無効にする必要があります。これにより、**-Xrealtime** パラメーターを指定して標準 Java アプリケーションを実行するときに、全体的なパフォーマンスの低下が見られる可能性があります。

認定されたハードウェア構成でのパフォーマンス

認定されたシステムは、WebSphere Real Time for RT Linux のパフォーマンス・ゴールを達成するために十分なクロック細分性とプロセッサー速度を備えています。例えば、過負荷状態でなくヒープ・サイズが十分なシステムで実行される、適切に作成されたアプリケーションでは通常、GC による休止が 1 ミリ秒よりはるかに短い時間 (通常は約 500 マイクロ秒) になります。GC サイクル中、デフォルト環境設定が適用されるアプリケーションは、10 ミリ秒時間枠の移動中に経過する時間の 30% を超えて休止することはありません。どの 10 ミリ秒の期間内でも、GC による休止時間は、通常は合計で 3 ミリ秒未満になります。

タイミングの変動性の軽減

標準 JVM での変動性の 2 つの主要な原因は、WebSphere Real Time for RT Linux で次のように処理されます。

- Java コードの準備: ロードおよびジャストインタイム (JIT) コンパイルは、Ahead-Of-Time (AOT) コンパイルによって処理されます。 43 ページの『AOT コンパイラーの使用』を参照してください。
- ガーベッジ・コレクションによる休止: 標準ガーベッジ・コレクター・モードで生じる可能性のある長時間の休止は、Metronome ガーベッジ・コレクター を使用することによって回避されます。 73 ページの『Metronome ガーベッジ・コレクターの使用』を参照してください。

JVM 間でのクラス・データの共有 (非リアルタイム・モード)

クラスの共有は非リアルタイム・モードでもサポートされますが、その動作はリアルタイム・モードの場合とは異なります。

Java 仮想マシン (JVM) 間でのクラス・データの共有は、ディスク上のメモリー・マップ・キャッシュ・ファイルにデータを保管することによって行うことができます。複数の JVM がキャッシュを共有する場合、共有によって全体の仮想ストレー

© Copyright IBM Corp. 2003, 2011

ジの使用量が削減されます。また、共有によって、キャッシュ作成後の JVM の起動時間も短縮されます。共有クラス・キャッシュは、実行中のどの JVM からも独立しており、削除されるまで存続します。

共有キャッシュには、以下を含むことができます。
- ブートストラップ・クラス
- アプリケーション・クラス
- クラスを記述するメタデータ
- Ahead-of-time (AOT) コンパイル済みコード

第 8 章 セキュリティー

このセクションには、セキュリティーに関する重要な情報が含まれています。

共有クラス・キャッシュのセキュリティーの考慮事項

共有クラス・キャッシュは、キャッシュを管理しやすくし、ユーザビリティーを向上するように設計されていますが、デフォルトのセキュリティー・ポリシーは適切でない場合があります。

共有クラス・キャッシュを使用するときには、アクセスを制限してセキュリティーを強化できるようにするために、新規ファイルのデフォルト許可に配慮する必要があります。

ファイル	デフォルトの許可
新規共有キャッシュ	グループおよびその他の読み取り許可
`javasharedresources` ディレクトリー	全ユーザーに対する読み取り、書き込み、および実行許可

キャッシュを破棄または拡張するには、キャッシュ・ファイルとキャッシュ・ディレクトリーの両方に対する書き込み許可が必要です。

キャッシュ・ファイルに対するファイル許可の変更

共有クラス・キャッシュへのアクセスを制限するには、**chmod** コマンドを使用します。

必要な変更	コマンド
アクセスをユーザーおよびグループに制限	chmod 770 /tmp/javasharedresources
アクセスをユーザーに制限	chmod 700 /tmp/javasharedresources
ユーザーを特定のキャッシュのみの読み取りおよび書き込みアクセスに制限	chmod 600 /tmp/javasharedresources/<*file for shared cache*>
ユーザーおよびグループを特定のキャッシュのみの読み取りおよび書き込みアクセスに制限	chmod 660 /tmp/javasharedresources/<*file for shared cache*>

共有クラス・キャッシュの作成について詳しくは、 45 ページの『リアルタイム共有クラス・キャッシュの作成』を参照してください。

アクセス許可がないキャッシュへの接続

適切なアクセス許可がないキャッシュに接続しようとすると、以下のエラー・メッセージが表示されます。

```
JVMSHRC226E Error opening shared class cache file
JVMSHRC220E Port layer error code = -302
JVMSHRC221E Platform error message: Permission denied
```

© Copyright IBM Corp. 2003, 2011

```
JVMJ9VM015W Initialization error for library j9shr25(11): JVMJ9VM009E J9VMDllMain
failed
Could not create the Java virtual machine.
```

第 9 章 トラブルシューティングおよびサポート

WebSphere Real Time for RT Linux のトラブルシューティングおよびサポート

- 『一般的な問題判別方法』
- 111 ページの『OutOfMemory エラーのトラブルシューティング』
- 122 ページの『診断ツールの使用』

一般的な問題判別方法

問題判別は、発生した障害の種類や、適切な処置方針を理解する上で役に立ちます。

発生した問題の種類が分かれば、以下の 1 つ以上の作業を実行することができます。

- 問題を修正する。
- 適切な回避策を見つける。
- IBM に送るバグ・レポートを生成するために必要なデータを収集する。

Linux の問題判別

このセクションでは、Linux での問題判別について説明します。

IBM SDK for Java 7 ユーザー・ガイドには、Linux での問題の診断についての次のような有用なガイダンスが記載されています。

- Linux 環境のセットアップと確認
- 一般的なデバッグ手法
- クラッシュの診断
- ハングのデバッグ
- メモリー・リークのデバッグ
- パフォーマンス上の問題のデバッグ

この情報は、IBM SDK for Java 7 - Linux の問題判別で参照できます。

以下の情報は、IBM WebSphere Real Time for RT Linux についての補足情報です。

Linux 環境のセットアップと確認

IBM WebSphere Real Time for RT Linux 上で、JVM がシステム・ダンプを生成するように正しく構成されていることを確認します。

Linux システム・ダンプ (コア・ファイル)

クラッシュが発生した場合に取得すべき最も重要な診断データは、Linux システム・ダンプ (コア・ファイル) です。このファイルが生成されていることを確認する

© Copyright IBM Corp. 2003, 2011

には、IBM SDK for Java 7 ユーザー・ガイドの説明に従って、オペレーティング・システムの設定と使用可能なディスク・スペースを確認する必要があります。

Java 仮想マシンの設定

JVM は、クラッシュの発生時にコア・ファイルを生成するように設定されている必要があります。コマンド行で `java -Xrealtime -Xdump:what` を実行します。このオプションの出力は以下のとおりです。

```
-Xdump:system:
    events=gpf+abort+traceassert+corruptcache,
    label=/mysdk/sdk/jre/bin/core.%Y%m%d.%H%M%S.%pid.dmp,
    range=1..0,
    priority=999,
    request=serial
```

示されている値はデフォルトの設定値です。クラッシュの発生時にコア・ファイルを生成するには、少なくとも `events=gpf` を設定する必要があります。オプションは、コマンド行オプション **-Xdump:system[:name1=value1,name2=value2 ...]** で変更および設定することができます。

一般的なデバッグ手法

Java スレッド名はオペレーティング・システムで表示できるため、ps コマンドを使用するとデバッグの際に便利です。トレース・ツールを使用する際は、IBM WebSphere Real Time for RT Linux の正しいコマンドを使用する必要があります。

プロセス情報の分析

IBM WebSphere Real Time for RT Linux で ps コマンドを実行すると、以下のような出力が表示されます。

```
ps -eLo pid,tid,rtprio,comm,cmd
29286 29286      - java            jre/bin/java -Xrealtime -jar example.jar
29286 29287      - main            jre/bin/java -Xrealtime -jar example.jar
29286 29290     88 Signal Reporter jre/bin/java -Xrealtime -jar example.jar
29286 29295      - JIT Compilation jre/bin/java -Xrealtime -jar example.jar
29286 29296     13 JIT Sampler     jre/bin/java -Xrealtime -jar example.jar
29286 29297      - Signal Dispatch jre/bin/java -Xrealtime -jar example.jar
29286 29298      - Finalizer maste jre/bin/java -Xrealtime -jar example.jar
29286 29299     11 Gc Slave Thread jre/bin/java -Xrealtime -jar example.jar
29286 29300     89 Metronome GC Al jre/bin/java -Xrealtime -jar example.jar
29286 29301      - Thread-2        jre/bin/java -Xrealtime -jar example.jar
29286 29302     43 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29303     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29304     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29305     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29306     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29307     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29311     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29312     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29313     85 Realtime AEH No jre/bin/java -Xrealtime -jar example.jar
29286 29314     85 Realtime AEH No jre/bin/java -Xrealtime -jar example.jar
29286 29315     87 Realtime Schedu jre/bin/java -Xrealtime -jar example.jar
29286 29316     79 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29317     85 Realtime Non-he jre/bin/java -Xrealtime -jar example.jar
29286 29318     83 Realtime Heap T jre/bin/java -Xrealtime -jar example.jar
29286 29319     83 Realtime Heap T jre/bin/java -Xrealtime -jar example.jar
29286 29321     45 RealtimeThread- jre/bin/java -Xrealtime -jar example.jar
29286 29343     43 RealtimeThread- jre/bin/java -Xrealtime -jar example.jar
29286 29345      - stdout reader j jre/bin/java -Xrealtime -jar example.jar
29286 29346      - stderr reader j jre/bin/java -Xrealtime -jar example.jar
```

e すべてのプロセスを選択します。

L スレッドを表示します。

o 表示列の事前定義された形式を指定します。指定される列は、プロセス ID、スレッド ID、スケジューリング・ポリシー、リアルタイム・スレッド優先順位、およびプロセスに関連付けられたコマンドです。この情報は、ある時点において、アプリケーションおよび仮想マシン内のどのスレッドが実行されているのかを理解する上で役立ちます。

トレース・ツール

Linux には、**strace**、**ltrace**、および **mtrace** の 3 つのトレース・ツールがあります。コマンド `man strace` を実行すると、使用可能なすべてのオプションが表示されます。

strace

strace ツールは、システム呼び出しをトレースします。このツールは、既に使用可能なプロセスに対して使用するか、新規のプロセスで開始することができます。 strace は、プログラムが実行したシステム呼び出しおよびプロセスが受け取ったシグナルを記録します。システム呼び出しごとに、名前、引数、および戻り値が使用されます。 strace では、ソースを必要とせずにプログラムをトレースできます (再コンパイルは不要です)。 strace に **-f** オプションを指定すると、fork したシステム呼び出しの結果作成された子プロセスがトレースされます。 strace を使用して、プラグインの問題を調べたり、プログラムが正しく開始されない理由の解明を試みたりすることができます。

Java アプリケーションで strace を使用するには、`strace java -Xrealtime <class-name>` と入力します。

strace ツールのトレース出力は、`-o` オプションを使用してファイルに送ることができます。

ltrace

ltrace ツールは、ディストリビューションに依存します。これは strace に非常によく似ています。このツールは、実行中のプロセスによって呼び出された動的ライブラリー呼び出しをインターセプトして記録します。 strace は、実行中のプロセスが受け取ったシグナルに対しても同様の動作をします。

Java アプリケーションで ltrace を使用するには、`ltrace java -Xrealtime <class-name>` と入力します。

mtrace

mtrace は GNU ツール・セットに含まれています。これは、malloc、realloc、および free に対する特殊なハンドラーをインストールして、これらの関数の使用をすべてトレースしてファイルに記録できるようにします。このトレース処理はプログラムの効率を下げるため、通常使用時には使用可能にしないでください。 mtrace を使用するには、**IBM_MALLOCTRACE** に 1 を設定し、**MALLOC_TRACE** にトレース情報の保管先となる有効なファイルを設定します。ユーザーに、このファイルに対する書き込み権限が必要です。

Java アプリケーションで mtrace を使用するには、以下のように入力します。

```
export IBM_MALLOCTRACE=1
export MALLOC_TRACE=/tmp/file
java -Xrealtime <class-name>
mtrace /tmp/file
```

クラッシュの診断

クラッシュする前の実行プロセスと Java 環境に関する情報を収集する場合は、以下のガイドラインに従ってください。

プロセス情報の収集

クラッシュが発生するまでに何が起きていたのかを調査する場合は、コア・ファイルを分析するのではなく、**gdb** コマンドと **bt** コマンドを使用して、障害が発生したスレッドのスタック・トレースを表示してください。

Java 環境の理解

Javadump を使用して、各スレッドが実行していた内容、およびどの Java メソッドが実行されていたのかを判別します。関数のアドレスとライブラリーのアドレスを突き合わせて、各時点で実行されていたコードのソースを判別します。

-verbose:gc オプションを使用して Java ヒープ・メモリー域、永久メモリー域、およびスコープ・メモリー域の状態を確認します。以下について検討します。

- いずれかのメモリー域でメモリーが不足していて、これがクラッシュの原因になったかどうか。
- ガーベッジ・コレクション中にクラッシュが発生したかどうか (ガーベッジ・コレクションで障害が発生した可能性があります)。
- ガーベッジ・コレクション後にクラッシュが発生したかどうか (メモリーが破損した可能性があります)。

パフォーマンス上の問題のデバッグ

パフォーマンス上の問題をデバッグする際は、IBM SDK for Java 7 ユーザー・ガイドのトピックに加え、IBM WebSphere Real Time for RT Linux についての以下の特定の項目についても考慮してください。

メモリー域のサイズ変更

ヒープ・メモリー、永久メモリー、およびスコープ・メモリーのサイズを変更することによって、JVM を調整できます。 適切なサイズを選択して、パフォーマンスを最適化してください。適切なサイズを使用することによって、ガーベッジ・コレクターがより簡単に必要な使用効率を達成できるようになります。

メモリー域のサイズ変更について詳しくは、 145 ページの『Metronome ガーベッジ・コレクターのトラブルシューティング』を参照してください。

JIT コンパイルおよびパフォーマンス

JIT を使用する場合は、リアルタイム動作に対する影響を考慮する必要があります。

予測可能な動作が必要であるが、良好なパフォーマンスも必要であるという場合には、Ahead-Of-Time (AOT) コンパイルを使用することを検討する必要があります。詳しくは、 40 ページの『WebSphere Real Time for RT Linux でのコンパイル済みコードの使用』を参照してください。

Linux における既知の制限

Linux は急速に発展しているため、JVM とオペレーティング・システムの相互作用、特にスレッドの分野に関するさまざまな問題が持ち上がっています。

Linux システムに影響を与える可能性がある以下の制限に注意してください。

プロセスとしてのスレッド

Java スレッドの数がプロセスの許容最大数を超えた場合、プログラムで以下のことが発生する可能性があります。

- エラー・メッセージが表示される
- **SIGSEGV** エラーが発生する
- 停止する

詳しくは、http://www.volano.com/report/index.html にある「*The Volano Report*」を参照してください。

フローティング・スタックの制限

フローティング・スタックなしで実行している場合は、**-Xss** の設定内容とは無関係に、スレッドごとに 256 KB の最小ネイティブ・スタック・サイズが指定されます。

フローティング・スタックの Linux システムでは、**-Xss** の値が使用されます。フローティング・スタックでない Linux システムからマイグレーションする場合は、すべての **-Xss** 値を十分な大きさにし、256 KB という最小値に依存しないようにしてください。

glibc の制限

シンボルが見つからないために libjava.so ライブラリーをロードできなかったということを示すメッセージ (__bzero など) を受け取った場合は、GNU C ランタイム・ライブラリー (glibc) の旧バージョンがインストールされている可能性があります。 SDK for Linux でのスレッド実装には、glibc バージョン 2.3.2 以上が必要です。

フォントの制限

Red Hat システムにインストールする場合、フォント・サーバーが Java TrueType フォントを検出できるようにするには、以下を実行してください (例えば、Linux IA32 で)。

```
/usr/sbin/chkfontpath --add opt/IBM/javawrt3/jre/lib/fonts
```

これはインストール時に実行する必要があります。また、このコマンドを実行するには、「root」としてログオンしている必要があります。フォントの問題について詳しくは、「*Linux SDK and Runtime Environment User Guide*」を参照してください。

Linux Red Hat MRG カーネルでのパフォーマンスの問題

詳細ガーベッジ・コレクションを使用可能にして WebSphere Real Time を開始した場合、Red Hat MRG カーネルの構成に関する問題が原因で、アプリケーション・スレッドが予期せずに一時停止することがあります。これらの一時停止は詳細 GC の出力には報告されませんが、ネットワーク構成によっては数ミリ秒間停止することがあります。リモート側で定義された LDAP ユーザーから開始された JVM が最も大きな影響を受けます。ネーム・サービス・キャッシュ・デーモン (nscd) が開始されず、ネットワーク遅延が発生するためです。この問題を解決するには、nscd を開始します。以下の手順に従って nscd サービスの状況を確認し、問題を修正してください。

1. 以下のコマンドを入力して、nscd デーモンが実行されているか確認します。
```
/sbin/service nscd status
```
デーモンが実行されていない場合、以下のメッセージが表示されます。
```
nscd is stopped
```
2. root ユーザーとして以下のコマンドを実行し、nscd サービスを開始します。
```
/sbin/service nscd start
```
3. root ユーザーとして以下のコマンドを実行し、nscd サービスの開始情報を変更します。
```
/sbin/chkconfig nscd on
```

これで nscd プロセスは実行状態になりました。リブート後に自動的に開始されます。

NLS の問題判別

JVM には、さまざまなロケールに対応するサポートが組み込まれています。

IBM SDK for Java 7 ユーザー・ガイドには、NLS に関する問題の診断についての次のような有用なガイダンスが記載されています。

- フォントの概要
- フォント・ユーティリティー
- NLS に関する一般的な問題と、考えられる原因

この情報は、IBM SDK for Java 7 - NLS の問題判別で参照できます。

ORB の問題判別

ORB の問題をデバッグする際にまず実行すべきことは、問題が分散アプリケーションのサーバー・サイドにあるのか、クライアント・サイドにあるのかを判断することです。ここでは、標準的な RMI-IIOP セッションを、オブジェクトへのアクセスを要求するクライアントと、アクセスを提供するサーバーの間の簡単な同期通信と考えます。

IBM SDK for Java 7 ユーザー・ガイドには、ORB に関する問題の診断についての次のような有用なガイダンスが記載されています。

- ORB 問題の特定
- スタック・トレースの解釈

- ORB トレースの解釈
- 共通問題
- IBM ORB サービス: データ収集

この情報は、IBM SDK for Java 7 - ORB の問題判別で参照できます。

以下の情報は、IBM WebSphere Real Time for RT Linux についての補足情報です。

IBM ORB サービス: データ収集

サービス用の Java バージョンの出力を収集するには、次のコマンドを実行します。

```
java -Xrealtime -version
```

予備テスト

問題が発生すると、ORB は次のような org.omg.CORBA.* 例外を生成する場合があります。

- 原因を示すテキスト
- マイナー・コード
- 完了状況

ORB を問題の原因と考える前に、以下の項目を確認してください。

- 同じような構成でシナリオを再現できる。
- JIT が使用不可である。
- AOT コンパイル・コードが使用されていない。

その他の処置として以下のことがあります。

- その他のプロセッサーの電源を切ります。
- 可能な場合は、同時マルチスレッド (SMT) をオフにします。
- クライアントまたはサーバーとのメモリー依存関係を除去します。物理メモリー不足は、パフォーマンスの低下、明らかなハング、またはクラッシュの原因になる可能性があります。これらの問題を除去するために、メモリーの十分なヘッドルームを確保するようにしてください。
- 物理ネットワークの問題 (ファイアウォール、通信リンク、ルーター、DNS ネーム・サーバーなど) を確認します。これらは、CORBA COMM_FAILURE 例外の主な原因です。試験的に、ご使用のワークステーション名を ping してみてください。
- アプリケーションで DB2® などのデータベースを使用している場合は、最も信頼性のあるドライバーに切り替えます。例えば、DB2 AppDriver を切り離す場合は、低速で複数のソケットを使用するが、より信頼性のある Net Driver に切り替えます。

OutOfMemory エラーのトラブルシューティング

OutOfMemoryError 例外、メモリー・リーク、および非表示メモリー割り振りへの対処

Metronome ガーベッジ・コレクターの一般的なトラブルシューティング情報については、 145 ページの『Metronome ガーベッジ・コレクターのトラブルシューティング』を参照してください。

OutOfMemoryError の診断

Metronome ガーベッジ・コレクターでの OutOfMemoryError 例外の診断は、このガーベッジ・コレクターが定期的な性質を持っているために、標準の JVM の場合よりも複雑になることがあります。

各タイプのヒープの特性については、 14 ページの『メモリー管理』で説明されています。通常、RTSJ アプリケーションは、標準の Java アプリケーションよりも約 20% 多いヒープ・スペースを必要とします。

キャッチされていない OutOfMemoryError が発生した場合、JVM はデフォルトでは以下の診断出力を生成します。

- スナップ・ダンプ。 122 ページの『ダンプ・エージェントの使用』を参照してください。
- Heapdump。 132 ページの『Heapdump の使用』を参照してください。
- Javadump。 126 ページの『Javadump の使用』を参照してください。
- システム・ダンプ。 135 ページの『システム・ダンプおよびダンプ・ビューアーの使用』を参照してください。

以下のように、ダンプ・ファイル名はコンソール出力に示されます。

```
JVMDUMP006I Processing dump event "systhrow", detail "java/lang/OutOfMemoryError" - please wait.
JVMDUMP007I JVM Requesting Snap dump using 'Snap.20081017.104217.13161.0001.trc'
JVMDUMP010I Snap dump written to Snap.20081017.104217.13161.0001.trc
JVMDUMP007I JVM Requesting Heap dump using 'heapdump.20081017.104217.13161.0002.phd'
JVMDUMP010I Heap dump written to heapdump.20081017.104217.13161.0002.phd
JVMDUMP007I JVM Requesting Java dump using 'javacore.20081017.104217.13161.0003.txt'
JVMDUMP010I Java dump written to javacore.20081017.104217.13161.0003.txt
JVMDUMP013I Processed dump event "systhrow", detail "java/lang/OutOfMemoryError".
```

コンソール出力に示され、Javadump にも記録される Java バックトレースは、Java アプリケーションのどこで OutOfMemoryError が発生したのかを示します。次のステップは、どの RTSJ メモリー域がいっぱいになっているのかを判別することです。 JVM メモリー管理コンポーネントは、障害が発生した割り振りのサイズ、クラス・ブロック・アドレス、およびメモリー・スペース名を示すトレース・ポイントを発行します。以下のように、このトレース・ポイントはスナップ・ダンプで見つかります。

```
<< 行省略... >>
09:42:17.563258000 *0xf2888e00       j9mm.101  Event       J9AllocateIndexableObject() returning NULL! 80
bytes requested for object of class 0xf1632d80 from memory space 'Metronome' id=0xf288b584
```

割り振られているオブジェクトのタイプによって、トレース・ポイント ID およびデータ・フィールドがここに示されているものと異なる場合があります。この例のトレース・ポイントは、アプリケーションが、Metronome ヒープのメモリー・セグメント id=0x809c5f0 で、class 0x81312d8 タイプの 33.6 MB のオブジェクトを割り振ろうとした際に障害が発生したことを示しています。

Javadump のメモリー管理情報を確認することによって、どの RTSJ メモリー域が影響を受けるかを判別できます。

```
NULL           ---------------------------------------------------------------
0SECTION       MEMINFO subcomponent dump routine
NULL           =================================
NULL
1STMEMTYPE     Object Memory
NULL           region     start      end        size       name
1STHEAP        0xF288B584 0xF2A1C000 0xF6A1C000 0x04000000 Default
NULL
1STMEMUSAGE    Total memory available: 67108864 (0x04000000)
1STMEMUSAGE    Total memory in use:    66676824 (0x03F96858)
1STMEMUSAGE    Total memory free:      00432040 (0x000697A8)
NULL
NULL           region     start      end        size       name
1STHEAP        0xF288B5A4 0xF17FF008 0xF27FF008 0x01000000 Immortal
NULL
1STMEMUSAGE    Total memory available: 16777216 (0x01000000)
1STMEMUSAGE    Total memory in use:    00450816 (0x0006E100)
1STMEMUSAGE    Total memory free:      16326400 (0x00F91F00)
NULL
1STSEGTYPE     Internal Memory
NULL           segment    start      alloc      end        type       size
1STSEGMENT     0x0808DA48 0x0814A0A8 0x0814A0A8 0x0815A0A8 0x01000040 0x00010000
1STSEGMENT     0x0808DB50 0x08131EB8 0x08131EB8 0x08141EB8 0x01000040 0x00010000
<< lines removed for clarity >>
```

Javadump のクラス・セクションを確認することによって、割り振られているオブジェクトのタイプを判別できます。

```
NULL           ---------------------------------------------------------------
0SECTION       CLASSES subcomponent dump routine
NULL           =================================
<< 行省略... >>
1CLTEXTCLLOD    ClassLoader loaded classes
2CLTEXTCLLOAD    Loader *System*(0xF182BB80)
<< 行省略... >>
3CLTEXTCLASS      [C(0xF1632D80)
```

Javadump の情報により、割り振りが通常のヒープ (ID=0xF288B584) 内で文字配列に対して試行されたことや、ヒープの合計割り振りサイズが、該当する 1STHEAP 行が示すとおり 67108864 10 進バイトまたは 0x04000000 16 進バイト (つまり 64 MB) であることが分かります。

この例では、障害が発生した割り振りが、合計ヒープ・サイズとの関係で大きくなっています。アプリケーションが 33 MB の複数のオブジェクトを作成すると予想される場合、次のステップは、**-Xmx** オプションを使用してヒープのサイズを大きくすることです。

障害が発生したアプリケーションは、合計ヒープ・サイズとの関係で小さくなる方が一般的です。これは、以前の割り振りによってヒープが埋まっているためです。その場合、次のステップは、Heapdump を使用して、既存のオブジェクトに割り振られているメモリーの量を調べることです。

Heapdump は、すべてのオブジェクトのリストと、それらのオブジェクト・クラス、サイズ、および参照が含まれている圧縮バイナリー・ファイルです。Heapdump は、IBM Support Assistant (ISA) からダウンロードできる Java 用メモリー・ダンプ診断ツール (MDD4J) を使用して分析します。

MDD4J を使用すると、Heapdump をロードして、ヒープ・スペースを大量に消費している疑いのあるオブジェクトのツリー構造を特定することができます。このツールには、ヒープ上のオブジェクトに関するさまざまなビューが用意されています。例えば、MDD4J は、リークが疑われるものの詳細を示すビューを表示し、ヒープ・サイズの消費量が多い上位 5 つのオブジェクトおよびパッケージを示すこともできます。ツリー・ビューを選択すると、リークが発生しているコンテナー・オブジェクトの性質に関するさらに詳しい情報が表示されます。

デフォルトでは、すべての RTSJ メモリー・スペース内のすべてのオブジェクトが含まれている単一の Heapdump ファイルが生成されます。メモリー・スペースごとに別々の Heapdump を要求するには、コマンド行オプション **-Xdump:heap:request=multiple** を使用します。複数のダンプを使用すると、特定のメモリー域に割り振られたオブジェクトのセットだけを調べることができます。Heapdump の特定は、コンソール出力に示されたファイル名によって行います。

```
JVMDUMP006I Processing Dump Event "uncaught", detail "java/lang/OutOfMemoryError" - Please Wait.
<< 行省略... >>
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Default0809DCD8-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Default0809DCD8-0002.phd
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Immortal0809DCF4-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Immortal0809DCF4-0002.phd
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Scope0809DD10-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Scope0809DD10-0002.phd
<< 行省略... >>
JVMDUMP013I Processed Dump Event "uncaught", detail "java/lang/OutOfMemoryError".
Exception in thread "RTJ Memory Consumer (thread_type=Realtime)" java.lang.OutOfMemoryError
 at tests.com.ibm.jtc.ras.runnable.DepleteMemory.depleteMemory(DepleteMemory.java:57)
<< 行省略... >>
```

IBM JVM によるメモリーの管理方法

IBM JVM は、クラス、コンパイル済みコード、Java オブジェクト、Java スタック、JNI スタック用のメモリー領域など、各種コンポーネント用のメモリーを必要とします。これらのメモリー領域のいくつかは、連続したメモリー内にある必要があります。その他のメモリー領域は、複数の小さなメモリー領域にセグメント化して、相互にリンクすることができます。

動的にロードされるクラスおよびコンパイル済みコードは、動的にロードされるクラス用のセグメント化メモリー領域に格納されます。クラスはさらに細かく分割され、書き込み可能メモリー領域 (RAM クラス) および読み取り専用メモリー領域 (ROM クラス) に格納されます。実行時、アプリケーションを始動する際にクラス・キャッシュは連続したメモリー領域にメモリー・マップされますが、必ずしもロードされるとは限りません。 クラス・キャッシュ内のクラスおよびコンパイル済みコードがストレージにマップされるのは、クラスがアプリケーションによって参照されるためです。クラスの ROM コンポーネントは、このクラスを参照する複数のプロセス間で共有されます。クラスの RAM コンポーネントは、そのクラスが最初に JVM によって参照された際に動的にロードされるクラス用のセグメント化メモリー領域に作成されます。クラス・キャッシュにある、クラスのメソッドの AOT コンパイル済みコードは、実行可能な動的コード・メモリー領域にコピーされます。このコードはプロセス間で共有されないためです。クラス・キャッシュからロードされないクラスは、キャッシュに入れられるクラスに似ていますが、ROM クラスの情報が、動的にロードされるクラス用のセグメント化メモリー領域に作成され

る点が異なります。動的に生成されるコードは、キャッシュに入れられるクラスの AOT コードを保持する場合と同じ動的コード・メモリー領域に格納されます。

-Xrealtime オプションを指定せずに JVM を実行した場合、すべての Java オブジェクトは、標準のヒープ・メモリーに格納されます。 **-Xrealtime** オプションを使用した場合、オブジェクトは、永久メモリーおよびスコープ・メモリーという 2 つの追加メモリー領域から割り振ることもできます。

各 Java スレッドのスタックは、セグメント化メモリー領域をまたぐことがあります。各スレッドの JNI スタックは、連続したメモリー領域を使用します。

JVM をどのように構成すべきか判別するには、**-verbose:sizes** オプションを指定して実行します。このオプションにより、サイズ管理可能なメモリー領域に関する情報が出力されます。連続していないメモリー領域の場合は、その領域を大きくする必要があるたびに獲得されるメモリーの量を示す増分が出力されます。

-Xrealtime -verbose:sizes オプションを使用した出力の例を以下に示します。

```
-Xmca32K                        RAM class segment increment
-Xmco128K                       ROM class segment increment
-Xms64M                         initial memory size
-Xgc:immortalMemorySize=16M     immortal memory space size
-Xgc:scopedMemoryMaximumSize=8M scoped memory space maximum size
-Xmx64M                         memory maximum
-Xmso256K                       operating system thread stack size
-Xiss2K                         java thread stack initial size
-Xssi16K                        java thread stack increment
-Xss256K                        java thread stack maximum size
```

この例では、RAM クラス・セグメントは最初は 0 ですが、必要に応じて 32 KB のブロック単位で大きくなることが示されています。 ROM クラス・セグメントは最初は 0 で、必要に応じて 128 KB のブロック単位で大きくなります。これらのサイズは、**-Xmca** オプションおよび **-Xmco** オプションを使用して制御できます。 RAM クラスおよび ROM クラスのセグメントは必要に応じて大きくなるため、通常、これらのオプションを変更する必要はありません。

永久メモリーは連続した領域であるため、比較的大きなスペースを事前割り振りする必要がある場合があります。この例の場合、永久メモリー領域には 16 MB が事前割り振りされています。この永久メモリー領域に 16 MB を超えるオブジェクトを書き込もうとすると、OutOfMemory 例外を受け取ります。定義上、このメモリー領域はガーベッジ・コレクションの対象にならないためです。

スコープ・メモリー領域は連続しており、この例では 8 MB が事前割り振りされています。プログラムの実行時に多くのスコープ・メモリー領域をアクティブにする場合は、より大きなスコープ・メモリー領域を指定する必要がある場合があります。

クラス・キャッシュを使用する場合にメモリー・マップ領域がどの程度の大きさになるかを判別するには、admincache ユーティリティーを使用します。コマンド `admincache -Xrealtime -printStats -nologo` の出力例を以下に示します。

```
J9 Java(TM) admincache 1.0

Current statistics for cache "sharedcc_localuser":
```

```
base address       = 0xA52B4000
end address        = 0xA59B7000
allocation pointer = 0xA59B4000

cache size         = 7356040
free bytes         = 330604
ROMClass bytes     = 3798460
AOT bytes          = 3101560
Data bytes         = 3812
Metadata bytes     = 121604
Metadata % used    = 1%

# ROMClasses       = 1044
# AOT Methods      = 1652
# Classpaths       = 2
# URLs             = 1
# Tokens           = 0
# Stale classes    = 0
% Stale classes    = 0%

Cache is 95% full
```

キャッシュ・サイズは、メモリー・マップ領域のスペースが 7 MB をわずかに超えていることを示しています。 ROM クラスおよび AOT のバイトがそれぞれ 3 MB をわずかに超え、このスペースの大部分を占めています。

永久メモリー・スペースでの OutOfMemoryError の例

この例では、永久メモリー・スペースでの OutOfMemoryError を特定する方法を示し、問題の発生を防ぐための手順について説明します。

以下のスナップ・ダンプは、永久メモリー域 id=0x809dd1c で 2 つの小さな割り振り要求が失敗したことを示しています。

```
16:08:04.876087000 083d4000       j9mm.100  Event       J9AllocateObject() returning NULL!
  16 bytes requested for object of class 0x8110e60 from memory space 'Immortal' id=0x809dd1c
16:08:04.876171000 083d4000       j9mm.100  Event       J9AllocateObject() returning NULL!
  32 bytes requested for object of class 0x81180f0 from memory space 'Immortal' id=0x809dd1c
```

以下の Javadump は、永久メモリー・スペースがいっぱいであることを示しています。

```
NULL           --------------------------------------------------------------
0SECTION       MEMINFO subcomponent dump routine
NULL           =================================
1STHEAPFREE    Bytes of Heap Space Free: 3f0c000
1STHEAPALLOC   Bytes of Heap Space Allocated: 4000000
1STHEAPFREE    Bytes of Immortal Space Free: 0
1STHEAPALLOC   Bytes of Immortal Space Allocated: 1000000
<< 行省略... >>
1STSEGTYPE     Object Memory
NULL           segment  start    alloc    end       type     bytes
1STSEGSUBTYPE  Immortal Segment  ID=0809DD1C
1STSEGMENT     0809D510 B279D008 B379D008 B379D008  00001008 1000000
```

MDD4J 分析は、使用可能メモリーの大部分を消費する非常に大きな LinkedList が割り振られていることを示しています。

永久メモリー域内のオブジェクトはガーベッジ・コレクションの対象にならないため、永久メモリー域に割り振るオブジェクトの数は最小化することが推奨されます。永久メモリーの最も一般的な用途は、主として JVM およびアプリケーション

の初期化時に実行される限定的なアクティビティーであるクラス・ロードです。多数のロード・クラス (または他の永久メモリーの用途) を持つアプリケーションでは、**-Xgc:immortalMemorySize=<size>** オプションを使用して、永久メモリー域のサイズを大きくできます。永久メモリー域のデフォルト・サイズは、16 MB です。

永久メモリー域のサイズを大きくしても、永久メモリーの OutOfMemoryError の発生を遅らせる効果しかない場合には、クラス・ロードまたは他のアプリケーション・オブジェクトに関連して行われる永久データの継続的な割り振りのパターンを詳しく調べてください。

スコープ・メモリー・スペースでの OutOfMemoryError の例

この例では、スコープ・メモリー・スペースでの OutOfMemoryError を特定する方法を示し、問題の発生を防ぐための手順について説明します。

コマンド行オプション **-Xdump:heap:request=multiple** を使用すると、以下のように、メモリー・スペースごとに別々のダンプが生成されます。

```
VMDUMP006I Processing Dump Event "uncaught", detail "java/lang/OutOfMemoryError" - Please Wait.
JVMDUMP007I JVM Requesting Snap Dump using '/home/test/snap-0001.trc'
JVMDUMP010I Snap Dump written to /home/test/snap-0001.trc
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Default0809DCD8-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Default0809DCD8-0002.phd
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Immortal0809DCF4-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Immortal0809DCF4-0002.phd
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Scope0809DD10-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Scope0809DD10-0002.phd
JVMDUMP007I JVM Requesting Java Dump using '/home/test/javacore-0003.txt'
JVMDUMP010I Java Dump written to /home/test/javacore-0003.txt
JVMDUMP013I Processed Dump Event "uncaught", detail "java/lang/OutOfMemoryError".
Exception in thread "RTJ Memory Consumer (thread_type=Realtime)" java.lang.OutOfMemoryError
   at tests.com.ibm.jtc.ras.runnable.DepleteMemory.depleteMemory(DepleteMemory.java:57)
   at tests.com.ibm.jtc.ras.runnable.DepleteMemory.run(DepleteMemory.java:26)
<< 行省略... >>
```

以下のスナップ・ダンプは、スコープ・メモリー域 id=0x809dd10 で 2 つの割り振り要求が失敗したことを示しています。

```
16:14:45.887176823 08480900       j9mm.100  Event       J9AllocateObject() returning NULL!
  16 bytes requested for object of class 0x8110e38 from memory space 'Scoped' id=0x809dd10
16:14:45.887252747 08480900       j9mm.100  Event       J9AllocateObject() returning NULL!
  32 bytes requested for object of class 0x81180c8 from memory space 'Scoped' id=0x809dd10
```

以下の Javadump では、id=0x809dd10 のスコープ・メモリー域に関して、メモリー域の割り振りサイズが 60 KB と非常に小さいことが示されています。この場合は、アプリケーション・コードでスコープ・メモリー域のサイズを大きくしてください。

```
0SECTION       MEMINFO subcomponent dump routine
NULL           =================================
1STHEAPFREE    Bytes of Heap Space Free: 3eb0000
1STHEAPALLOC   Bytes of Heap Space Allocated: 4000000
1STHEAPFREE    Bytes of Immortal Space Free: f47474
1STHEAPALLOC   Bytes of Immortal Space Allocated: 1000000
1STHEAPFREE    Bytes of Scoped Space ID=0809DD10 Free: eb00
1STHEAPALLOC   Bytes of Scoped Space Allocated: eb00
.......
1STSEGTYPE     Object Memory
NULL           segment  start    alloc    end       type     bytes
1STSEGSUBTYPE  Scoped Segment  ID=0809DD10
```

```
1STSEGMENT     0809D560 08416350 08424E50 08424E50  00002008 eb00
1STSEGSUBTYPE  Immortal Segment  ID=0809DCF4
1STSEGMENT     0809D4E8 B2857008 B3857008 B3857008  00001008 1000000
```

この Javadump の例では、スコープ・メモリー域が空であるように見えます。空に見えるのは、Javadump が、OutOfMemoryError が JVM に到達した際に生成されるためです。その時点では、スコープは終了し、クリーンアップされています。**-Xdump:java:events=throw,filter=java/lang/OutOfMemoryError** コマンド行オプションを使用することにより、障害点で Javadump を生成できます。このオプションを使用すると、スコープ・メモリー域のフリー・スペースが正しく報告されます。

スコープ・メモリー用に使用可能なすべてのスペースを使い果たしてしまう場合もあります。その場合は、コマンド行オプション **-Xgc:scopedMemoryMaximumSize=<size>** を使用して、スコープ・メモリー域のサイズを大きくしてください。スコープ・メモリー域のデフォルト・サイズは、8 MB です。スコープ・メモリー用に使用可能なすべてのスペースがなくなった場合は、例えば以下のような別のメッセージがコンソールに表示されます。

```
Exception in thread "main" java.lang.OutOfMemoryError: Creating (LTMemory) Scoped memory # 0 size=16777216
        at javax.realtime.MemoryArea.create(MemoryArea.java:808)
        at javax.realtime.MemoryArea.create(MemoryArea.java:798)
        at javax.realtime.ScopedMemory.create(ScopedMemory.java:1359)
        at javax.realtime.ScopedMemory.create(ScopedMemory.java:1351)
        at javax.realtime.ScopedMemory.initialize(ScopedMemory.java:1705)
        at javax.realtime.ScopedMemory.<init>(ScopedMemory.java:216)
        at javax.realtime.ScopedMemory.<init>(ScopedMemory.java:164)
```

複数のヒープでの問題診断

Javadump にあるアドレス範囲と Heapdump にある占有率情報は、複数の RTSJ メモリー域での OutOfMemoryError を分析する際に役立ちます。

以下の Javadump では、永久セグメントが 0xB281C008 から 0xB381C008 の範囲にあり、通常のヒープ・セグメントが 0xB381D008 から 0xB781D008 の範囲にあります。

```
0SECTION       MEMINFO subcomponent dump routine
NULL           =================================
1STHEAPFREE    Bytes of Heap Space Free: 58000
1STHEAPALLOC   Bytes of Heap Space Allocated: 4000000
1STHEAPFREE    Bytes of Immortal Space Free: b319d8
1STHEAPALLOC   Bytes of Immortal Space Allocated: 1000000
NULL
1STSEGTYPE     Internal Memory
<< 行省略... >>
1STSEGTYPE     Object Memory
NULL           segment  start    alloc    end       type     bytes
1STSEGSUBTYPE  Immortal Segment  ID=0809C68C
1STSEGMENT     0809BE80 B281C008 B381C008 B381C008  00001008 1000000
1STSEGSUBTYPE  Heap Segment  ID=0809C670
1STSEGMENT     0809BE08 B381D008 B781D008 B781D008  00000009 4000000
NULL
1STSEGTYPE     Class Memory
NULL           segment  start    alloc    end       type     bytes
1STSEGMENT     08158154 083FFD68 083FFEF0 08407D68  00010040 8004
```

Heapdump は、すべてのオブジェクトのリストと、それらのオブジェクト・クラス、サイズ、および参照が含まれている圧縮バイナリー・ファイルです。 Heapdump は、IBM Support Assistant (ISA) からダウンロードできる Java 用メモリー・ダンプ診断ツール (MDD4J) を使用して分析します。

MDD4J によってリストされるオブジェクト・メモリーの場所を使用して、オブジェクトが置かれているメモリー・スペースを判別します。 0xB28*nnnnn* の範囲にあるアドレスは、永久メモリー域にあります。0xB61*nnnnn* の範囲にあるアドレスは、通常のヒープ内にあります。

メモリー・リークの回避

ガーベッジ・コレクターは、永久メモリー域およびスコープ・メモリー域に対する処理を行いません。永久メモリーの場合、JVM の終了時にのみ、メモリーが解放されます。スコープ・メモリー域は、その参照数がゼロになった場合にのみ解放されます。これらのコンテキストで長時間実行されるタスクは、タスクのウォームアップ後に、永久メモリー域のメモリーがさらに割り振られないような方法で、書き込む必要があります。

クラスのロードには、少量の永久メモリーが使用されます。リアルタイム環境では、クラスに対してガーベッジ・コレクションは行われません。したがって、アプリケーションで必要のないクラスをロードすると、アプリケーションによって必要以上の永久メモリーが使用される可能性があります。

ご使用のアプリケーションに Serializable インターフェースを実装するクラスが含まれている場合は、生成されたクラスの占有スペースを占める永久メモリーの初期サイズを調整してください。コンストラクターにはそれぞれクラスごとに 1 つの生成済みオブジェクトがあり、「GeneratedSerializationConstructorAccessorXXX」(XXX は数値) という形式になっています。これは、オブジェクトが初めて直列化されるときに永久メモリーにロードされます。

永久メモリーから割り振られたオブジェクトに対してはガーベッジ・コレクションが行われないため、永久メモリーを使うことは避けた方がよいでしょう。永久メモリー域がほとんど使用されない場合は、そのメモリー域でのオブジェクトのプーリングを考慮してください。

言語機能による非表示メモリーの割り振り

スコープ・メモリー・コンテキストまたは永久メモリー・コンテキストでは、これらのメソッドにより非表示メモリーが割り振られるため、可変引数の言語機能は使用しないようにしてください。

可変引数 (vararg)

可変引数は、Java 言語を使用して、配列としてメソッドに渡して実装されます。コンパイラーを使用してこの配列を作成し、初期化することで、可変引数メソッドを簡単に呼び出すことができます。

可変引数メソッドを呼び出すと、永久メモリー・コンテキストまたはスコープ・メモリー・コンテキストのメモリーが失われる場合があります。スコープ・メモリー・コンテキストまたは永久メモリー・コンテキストでは可変引数を使用しないでください。可変引数の代わりに、配列を明示的に作成して使用してください。

以下の 2 つの例は、可変引数メソッドを呼び出すための同等の方法を示しています。

```
public class VarargEx {

    public static void main(String[] args) {
        System.out.println("Sum: "+ sum(1.0, 2.0 , 3.0, 4.0));

     }
    static double sum(double... params) {
        double total=0.0;

        for(double num : params) {
            total += num;
        }

        return total;
    }
}
```

```
public class VarargEx {

    public static void main(String[] args) {
        double array[] = new double[4];

        array[0] = 1.0; array[1] = 2.0; array[2] = 3.0; array[3] = 4.0;
        System.out.println("Sum: " + sum(array));
    }

    static double sum(double... params) {
        double total=0.0;

        for(double num : params) {
            total += num;
        }

        return total;
    }
}
```

好ましいのは 2 番目の例です。これは、double 配列の割り振りがコード中で可視化されており、この割り振りを特定のメモリー域に誘導することが可能だからです。

ストリングの連結

ストリングを既存のストリングに追加してより長いストリングを生成するには、java.lang.StringBuilder オブジェクトを使用します。この場合、メモリーを割り振る必要があります。

自動ボックス化

自動ボックス化では、基本型を格納するためのオブジェクトが作成されます。この場合、メモリーを割り振る必要があります。

メモリー・コンテキスト全体でのリフレクションの使用

スコープ・メモリー域で作成された constructor オブジェクトは、同じスコープまたは内部スコープでのみ使用できます。永久、ヒープ、または外部スコープのメモリー・コンテキストでその constructor オブジェクトを使用しようとすると、失敗します。

複数のメモリー・コンテキストでリフレクションが行われたときにスローされる例外は、次に示すものに類似しています。

```
Exception in thread "NoHeapRealtimeThread-14" javax.realtime.IllegalAssignmentError
   at java.lang.reflect.Constructor$1.<init>(Constructor.java:570)
   at java.lang.reflect.Constructor.acquireConstructorAccessor(Constructor.java:568)
   at java.lang.reflect.Constructor.newInstance(Constructor.java:521)
   at testMain$TestRunnable$1.run(testMain.java:40)
   at javax.realtime.MemoryArea.activateNewArea(MemoryArea.java:597)
   at javax.realtime.MemoryArea.doExecuteInArea(MemoryArea.java:612)
   at javax.realtime.ImmortalMemory.executeInArea(ImmortalMemory.java:77)
   at testMain$TestRunnable.allocate(testMain.java:36)
   at testMain$TestRunnable.run(testMain.java:12)
   at java.lang.Thread.run(Thread.java:875)
   at javax.realtime.ScopedMemory.runEnterLogic(ScopedMemory.java:280)
   at javax.realtime.MemoryArea.enter(MemoryArea.java:159)
   at javax.realtime.ScopedMemory.enterAreaWithCleanup(ScopedMemory.java:194)
   at javax.realtime.ScopedMemory.enter(ScopedMemory.java:186)
   at javax.realtime.RealtimeThread.runImpl(RealtimeThread.java:1824)
```

この制限は、割り振り先と同じスコープで constructor を使用することによって回避できます。

スコープ・メモリー域での内部クラスの使用

スコープ・メモリー域のコンテキストで内部クラスを使用する際には、外部オブジェクトと内部オブジェクトが異なるメモリー域に存在する場合は、内部クラス・オブジェクトのインスタンス化の際に注意する必要があります。内部オブジェクトが外部オブジェクトへの参照を保管できない場合には、元のソース・コードで可視ではないコンパイラー生成コードから IllegalAssignmentError が出ます。

内部クラス・オブジェクトは、外部クラス・オブジェクトへの暗黙的な参照を保管できる必要があります。参照が RTSJ メモリー参照規則に違反している場合、IllegalAssignmentError が生成されます。

ほとんどの内部クラス (ローカルおよび匿名内部クラスを含む) には、字句的に囲んでいる外部クラスのインスタンス用のコンパイラー生成 (合成) の非静的フィールドが含まれます。唯一の例外は、内部クラス・インスタンスに、囲んでいる外部オブジェクトがない場合です (例えば、静的イニシャライザー・ブロックでインスタンス化された匿名クラス・オブジェクト)。内部オブジェクトの合成フィールドには、外部オブジェクトへの参照が含まれます。 Java プログラマーにとって便利なように、これはコンパイラーによって実装されます。このフィールドは元のソース・コードでは可視ではありません。ただし、可視の参照を持つ静的ネスト・クラスを使用して同じようなコードを作成することは可能です。暗黙的な参照が RTSJ メモリー域規則に違反している場合は、内部オブジェクトの構成時に、外部オブジェクトへの参照を保管しようとするため、IllegalAssignmentError がスローされます。

一般的に、内部クラスの使用時に RTSJ メモリー参照規則に違反することはできません。関連外部オブジェクトへの参照が RTSJ メモリー参照規則に違反している場合には、内部オブジェクトを作成できません。この規則は、永久メモリーまたはヒープに割り振られた内部オブジェクトが、スコープ・メモリーからの外部オブジェクトへの参照を持つことができないことを意味します。スコープ・メモリーからの内部オブジェクトは、スコープ・メモリーからの外部オブジェクトへの参照を持つことができますが、外部オブジェクトは、同じスコープ・メモリー域または外部スコープ・メモリー域から割り振られる必要があります。

回避方法が存在し、例えば以下の方法があります。

- 静的ネスト・クラスを使用して、暗黙的な参照を除去する
- 内部オブジェクトと外部オブジェクトの関係がメモリー域参照制約に違反しないようにメモリー域を選択する

診断ツールの使用

IBM WebSphere Real Time for RT Linux JVM に関連した問題の診断に利用できる診断ツールは、数多くあります。

IBM SDK for Java 7 には、IBM WebSphere Real Time for RT Linux JVM に関連する問題の診断に利用できる診断ツールが数多く用意されています。このセクションでは使用可能なツールを紹介し、それらのツールの使い方に関する詳細情報へのリンクを記載しています。

SDK 診断ツールを使う際には、覚えておく必要のある重要な点があります。リアルタイム JVM を呼び出す際には、以下のオプションを使用してください。

```
java -Xrealtime
```

リアルタイム JVM の診断ツールを実行する際には、このオプションを指定する必要があります。例えば、IBM WebSphere Real Time for RT Linux JVM に登録されているダンプ・エージェントを表示するには、次のように入力します。

```
java -Xrealtime -Xdump:what
```

IBM WebSphere Real Time for RT Linux に付属するこれらのツールを使う際の詳しい相違点については、以下のセクションで補足情報として説明し、同時に診断に役立つサンプル出力も提供します。

IBM SDK for Java 7 が生成する診断情報のまとめについては、 診断情報のまとめ (Summary of diagnostic information) を参照してください。

ダンプ・エージェントの使用

ダンプ・エージェントは、JVM の初期化時にセットアップされます。ダンプ・エージェントを使用すると、JVM で発生しているガーベッジ・コレクション、スレッド開始、JVM 終了などのイベントに応じて、ダンプを開始したり外部ツールを起動したりできます。

IBM SDK for Java 7 ユーザー・ガイドには、ダンプ・エージェントについての次のような有用なガイダンスが記載されています。

- **-Xdump** オプションの使用
- ダンプ・エージェント
- ダンプ・イベント
- ダンプ・エージェントの拡張制御
- ダンプ・エージェントのトークン
- デフォルトのダンプ・エージェント
- ダンプ・エージェントの削除
- ダンプ・エージェントの環境変数

- シグナルのマッピング
- ダンプ・エージェントのデフォルトの場所

この情報は、IBM SDK for Java 7 - ダンプ・エージェントの使用で参照できます。

IBM WebSphere Real Time for RT Linux の補足情報は以下に記載されています。

ダンプ・イベント

ダンプ・エージェントは、JVM の実行中に発生したイベントによって起動されます。 IBM WebSphere Real Time for RT Linux の場合、slow イベントのデフォルト値は 5 ミリ秒です。

一部のイベントをフィルターに掛けることで、出力の妥当性を向上させることができます。詳しくは、 124 ページの『filter オプション』を参照してください。

注: 現在のところ、WebSphere Real Time で unload イベントおよび expand イベントは発生しません。クラスは永久メモリー内にあるため、アンロードできません。

注: gpf イベントおよび abort イベントによって、Heapdump の起動、ヒープの準備 (request=prepwalk)、またはヒープの圧縮 (request=compact) を実行することはできません。

以下の表に、ダンプ・エージェントのトリガーとして使用可能なイベントを示します。

イベント	起動条件	フィルター操作
gpf	一般保護違反 (GPF) が発生したとき。	
user	JVM がオペレーティング・システムから SIGQUIT シグナルを受け取ったとき。	
abort	JVM がオペレーティング・システムから SIGABRT シグナルを受け取ったとき。	
vmstart	仮想マシンが開始されたとき。	
vmstop	仮想マシンが停止されたとき。	終了コードに対するフィルター (例えば、`filter=#129..#192#-42#255`)
load	クラスがロードされたとき。	クラス名に対するフィルター (例えば、`filter=java/lang/String`)
unload	クラスがアンロードされたとき。	
throw	例外がスローされたとき。	例外クラス名に対するフィルター (例えば、`filter=java/lang/OutOfMem*`)
catch	例外が catch されたとき。	例外クラス名に対するフィルター (例えば、`filter=*Memory*`)
uncaught	Java 例外がアプリケーションによって catch されなかったとき。	例外クラス名に対するフィルター (例えば、`filter=*MemoryError`)
systhrow	JVM が Java 例外をスローしようとしているとき。これは「throw」イベントとは異なり、JVM の内部で検出されたエラー条件に対してのみ起動されます。	例外クラス名に対するフィルター (例えば、`filter=java/lang/OutOfMem*`)
thrstart	新規スレッドが開始されたとき。	
blocked	スレッドがブロック状態になったとき。	

イベント	起動条件	フィルター操作
thrstop	スレッドが停止されたとき。	
fullgc	ガーベッジ・コレクション・サイクルが開始されたとき。	
slow	内部 JVM 要求に対してスレッドの応答時間が 5ms を超えたとき。	低速と判断されるイベントについては所要時間を変更してください。例えば、**filter=#300ms** にした場合は、内部 JVM 要求に対してスレッドの応答時間が 300ms を超えたときに起動します。
allocation	所定のフィルター指定と一致するサイズの Java オブジェクトが割り振られたとき。	オブジェクト・サイズに対するフィルター。フィルターの指定は必須です。例えば、**filter=#5m** の場合は、オブジェクトのサイズが 5 Mb より大きいときに起動します。範囲もサポートしています。例えば、**filter=#256k..512k** の場合は、オブジェクトのサイズが 256 Kb から 512 Kb までの間であるときに起動します。
traceassert	JVM で内部エラーが発生したとき。	適用されません。
corruptcache	JVM が共有クラス・キャッシュの破損を検出したとき。	適用されません。

filter オプション

一部の JVM イベントは、アプリケーションの存続期間中に数千回発生します。ダンプ・エージェントでは、フィルターおよび範囲を使用して、必要以上のダンプが生成されないようにすることができます。

ワイルドカード

例外イベント・フィルターには、ワイルドカードとしてアスタリスクを使用できます。使用できるのは、フィルターの先頭または末尾のみです。以下のコマンドは、2 つ目のアスタリスクが末尾以外の場所に置かれているため、機能しません。

```
-Xdump:java:events=vmstop,filter=*InvalidArgumentException#*.myVirtualMethod
```

このフィルターを機能させるためには、以下のように変更する必要があります。

```
-Xdump:java:events=vmstop,filter=*InvalidArgumentException#MyApplication.*
```

クラス・ロード・イベントおよび例外イベント

クラス・ロード (load) イベントおよび例外 (throw、catch、uncaught、systhrow) イベントは、以下のように、Java クラス名を基準にしてフィルターに掛けることができます。

```
-Xdump:java:events=throw,filter=java/lang/OutOfMem*
-Xdump:java:events=throw,filter=*MemoryError
-Xdump:java:events=throw,filter=*Memory*
```

throw、uncaught、および systhrow の各例外イベントは、Java メソッド名を基準にしてフィルターに掛けることができます。

```
-Xdump:java:events=throw,filter=ExceptionClassName[#ThrowingClassName.
throwingMethodName[#stackFrameOffset]]
```

任意指定部分は大括弧で囲んで示しています。

catch 例外イベントは、Java メソッド名を基準にしてフィルターに掛けることができます。

```
-Xdump:java:events=catch,filter=ExceptionClassName[#CatchingClassName.
catchingMethodName]
```

任意指定部分は大括弧で囲んで示しています。

vmstop イベント

JVM のシャットダウン・イベントは、以下のように、1 つ以上の終了コードを使用してフィルターに掛けることができます。

```
-Xdump:java:events=vmstop,filter=#129..192#-42#255
```

slow イベント

slow イベントは、フィルターに掛けることで時間しきい値をデフォルトの 5 ミリ秒から変更することができます。

```
-Xdump:java:events=slow,filter=#300ms
```

デフォルトの時間よりも短い時間にフィルターを設定することはできません。

allocation イベント

allocation イベントは、フィルターに掛けて、起動条件となるオブジェクトのサイズを指定する必要があります。フィルターのサイズは、32 ビット・プラットフォームの場合にはゼロから 32 ビット・ポインターの最大値まで、64 ビット・プラットフォームの場合にはゼロから 64 ビット・ポインターの最大値までの間に設定できます。フィルターの下限値をゼロに設定すると、すべての割り振りでダンプが起動されます。

例えば、割り振りのサイズが 5 Mb より大きい場合にダンプを起動するには、以下の設定を使用します。

```
-Xdump:stack:events=allocation,filter=#5m
```

割り振りのサイズが 256Kb から 512Kb までの間である場合にダンプを起動するには、以下の設定を使用します。

```
-Xdump:stack:events=allocation,filter=#256k..512k
```

その他のイベント

フィルターをサポートしていないイベントにフィルターを適用した場合、そのフィルターは無視されます。

request オプション

ダンプ・エージェントを開始する前に、状態を整えるように JVM に要求するには、request オプションを使用します。 IBM WebSphere Real Time for RT Linux の場合は、**multiple** という追加の request オプションがあります。

使用可能なオプションを以下の表にリストします。

オプション値	説明
exclusive	JVM への排他的アクセスを要求します。
compact	ガーベッジ・コレクションを実行します。このオプションを使用すると、ダンプを生成する前に、到達不能なすべてのオブジェクトがヒープから削除されます。
prepwalk	ヒープ・ウォーキングの準備を整えます。このオプションを使用する場合は、**exclusive** も指定する必要があります。
serial	このダンプが終了するまで他のダンプは延期します。
multiple	RTSJ メモリー域ごとに別々の Heapdump を生成します。
preempt	Java ダンプ・エージェントに適用され、スタック・トレースの収集にプロセス内のネイティブ・スレッドを強制的に優先使用するかどうかを制御します。このオプションが指定されていない場合は、Javadump 内の Java スタック・トレースのみが収集されます。

例えば、javadump の request オプションのデフォルト設定は、`request=exclusive+preempt` です。ネイティブ・スタック・トレースを収集する際にスレッドを優先使用せずに Javadump が生成されるように設定を変更するには、以下のオプションを使用します。

```
-Xdump:java:request=exclusive
```

通常は、デフォルトの request オプションで十分です。

`+` を使用して、複数の request オプションを指定することができます。例えば、次のようにします。

```
-Xdump:heap:request=exclusive+compact+prepwalk
```

Javadump の使用

Javadump は、実行中のある一時点で JVM および Java アプリケーションに関して収集した診断情報を記録したファイルを生成します。例えば、オペレーティング・システム、アプリケーション環境、スレッド、スタック、ロック、およびメモリーなどに関する情報です。

IBM SDK for Java 7 ユーザー・ガイドには、Javadump についての次のような有用なガイダンスが記載されています。

- Javadump の有効化
- Javadump の起動
- Javadump の解釈
- 環境変数と Javadump

この情報は、IBM SDK for Java 7 - Javadump の使用で参照できます。

IBM WebSphere Real Time for RT Linux の補足情報と出力例は、以下のトピックに記載されています。

ストレージ管理 (MEMINFO)

`MEMINFO` セクションには、ヒープ・メモリー域、永久メモリー域、スコープ・メモリー域を含む、メモリー・マネージャーに関する情報が示されます。

Javadump の MEMINFO セクションには、メモリー・マネージャーに関する情報が示されます。メモリー・マネージャー・コンポーネントの動作について詳しくは、Metronome ガーベッジ・コレクターの使用 (Using the Metronome Garbage Collector) を参照してください。

Javadump のこの部分には、次のようなさまざまなストレージ管理の値が示されます。

- 空きメモリーの量
- 使用メモリーの量
- ヒープの現在のサイズ
- 永久メモリー域の現在のサイズ
- スコープ・メモリー域の現在のサイズ

このセクションには、ガーベッジ・コレクションの履歴データも含まれています。このデータは、タイム・スタンプが記された一連のトレース・ポイントとして表されます。最新のトレース・ポイントが先頭になります。

標準の JVM が生成した Javadump には、「GC History」セクションがあります。この情報は、リアルタイム JVM 使用時に生成された Javadump には含まれていません。 GC の動作に関する情報を取得するには、**-verbose:gc** オプションまたは JVM スナップ・トレースを使用してください。詳しくは、145 ページの『verbose:gc 情報の使用』、および IBM SDK for Java 7 ユーザー・ガイドのダンプ・エージェントのセクションを参照してください。

スコープ・メモリーを使用するプログラムを実行中に、OutOfMemoryError がスローされると、Javadump にリストされた一部のメモリー域が空になっていることがあります。他のスコープ内にネストされたスコープでメモリー不足が発生した場合、Javadump の生成時にはその内側のスコープが削除されている可能性があります。OutOfMemoryError がスローされた時点のメモリー域の状態に関する情報を取得するには、以下のコマンド行オプションを使用してプログラムを実行します。

```
-Xdump:java:events=throw,filter=java/lang/OutOfMemoryError,range=1..1
```

このコマンドにより、uncaught 例外が検出された時点 (これは若干遅れて発生します) ではなく、OutOfMemoryError がスローされた時点で、追加の Javadump が生成されます。この Javadump では、内側のすべてのスコープを含め、OutOfMemoryError がスローされた時点でアクティブであったすべてのメモリー域を確認することができます。 **-Xdump** オプションの使用について詳しくは、IBM SDK for Java 7 ユーザー・ガイドを参照してください。

Javadump で、セグメントは Java ランタイムによって、大量のメモリーを使用するタスクに割り振られているメモリーのブロックです。例として次のようなタスクがあります。

- JIT キャッシュの保守
- Java クラスの保管

Java ランタイムは、MEMINFO セクションにリストされていない、その他のネイティブ・メモリーも割り振ります。 Java ランタイム・セグメントによって使用される合計メモリーが、必ずしも Java ランタイムの完全なメモリー占有スペースを表してい

るとは限りません。Java ランタイム・セグメントは、セグメント・データ構造および関連するネイティブ・メモリーのブロックで構成されています。

典型的な出力の例を以下に示します。値はすべて 16 進値として示されています。MEMINFO セクションの列見出しには、以下の意味があります。

```
0SECTION       MEMINFO subcomponent dump routine
NULL           =================================
NULL
1STHEAPTYPE    Object Memory
NULL           id         start      end        size       space/region
1STHEAPSPACE   0x00497030     --         --         --     Generational
1STHEAPREGION  0x004A24F0 0x02850000 0x05850000 0x03000000 Generational/Tenured Region
1STHEAPREGION  0x004A2468 0x05850000 0x06050000 0x00800000 Generational/Nursery Region
1STHEAPREGION  0x004A23E0 0x06050000 0x06850000 0x00800000 Generational/Nursery Region
NULL
1STHEAPTOTAL   Total memory:           67108864 (0x04000000)
1STHEAPINUSE   Total memory in use:    33973024 (0x02066320)
1STHEAPFREE    Total memory free:      33135840 (0x01F99CE0)
NULL
1STSEGTYPE     Internal Memory
NULL           segment    start      alloc      end        type       size
1STSEGMENT     0x073DFC9C 0x0761B090 0x0761B090 0x0762B090 0x01000040 0x00010000
       (lines removed for clarity)
1STSEGMENT     0x00497238 0x004FA220 0x004FA220 0x0050A220 0x00800040 0x00010000
NULL
1STSEGTOTAL    Total memory:             873412 (0x000D53C4)
1STSEGINUSE    Total memory in use:           0 (0x00000000)
1STSEGFREE     Total memory free:        873412 (0x000D53C4)
NULL
1STSEGTYPE     Class Memory
NULL           segment    start      alloc      end        type       size
1STSEGMENT     0x0731C858 0x0745C098 0x07464098 0x07464098 0x00010040 0x00008000
        (lines removed for clarity)
1STSEGMENT     0x00498470 0x070079C8 0x07026DC0 0x070279C8 0x00020040 0x00020000
NULL
1STSEGTOTAL    Total memory:            2067100 (0x001F8A9C)
1STSEGINUSE    Total memory in use:    1839596 (0x001C11EC)
1STSEGFREE     Total memory free:        227504 (0x000378B0)
NULL
1STSEGTYPE     JIT Code Cache
NULL           segment    start      alloc      end        type       size
1STSEGMENT     0x004F9168 0x06960000 0x069E0000 0x069E0000 0x00000068 0x00080000
NULL
1STSEGTOTAL    Total memory:             524288 (0x00080000)
1STSEGINUSE    Total memory in use:      524288 (0x00080000)
1STSEGFREE     Total memory free:             0 (0x00000000)
NULL
1STSEGTYPE     JIT Data Cache
NULL           segment    start      alloc      end        type       size
1STSEGMENT     0x004F92E0 0x06A60038 0x06A6839C 0x06AE0038 0x00000048 0x00080000
NULL
1STSEGTOTAL    Total memory:             524288 (0x00080000)
1STSEGINUSE    Total memory in use:       33636 (0x00008364)
1STSEGFREE     Total memory free:        490652 (0x00077C9C)
NULL
1STGCHTYPE     GC History
3STHSTTYPE     15:18:14:901108829 GMT j9mm.134 -   Allocation failure end: newspace=7356368/8388608
oldspace=32038168/50331648 loa=3523072/3523072
3STHSTTYPE     15:18:14:901104380 GMT j9mm.470 -   Allocation failure cycle end: newspace=7356416/8388608
oldspace=32038168/50331648 loa=3523072/3523072
3STHSTTYPE     15:18:14:901097193 GMT j9mm.65 -   LocalGC end: rememberedsetoverflow=0
causedrememberedsetoverflow=0 scancacheoverflow=0 failedflipcount=0 failedflipbytes=0 failedtenurecount=0
failedtenurebytes=0 flipcount=11454 flipbytes=991056 newspace=7356416/8388608 oldspace=32038168/50331648
loa=3523072/3523072 tenureage=1
3STHSTTYPE     15:18:14:901081108 GMT j9mm.140 -   Tilt ratio: 50
```

```
| 3STHSTTYPE     15:18:14:893358658 GMT j9mm.64 -   LocalGC start: globalcount=3 scavengecount=24 weakrefs=0
| soft=0 phantom=0 finalizers=0
| 3STHSTTYPE     15:18:14:893354551 GMT j9mm.63 -   Set scavenger backout flag=false
| 3STHSTTYPE     15:18:14:893348733 GMT j9mm.135 -   Exclusive access: exclusiveaccessms=0.002
| meanexclusiveaccessms=0.002 threads=0 lastthreadtid=0x00495F00 beatenbyotherthread=0
| 3STHSTTYPE     15:18:14:893348391 GMT j9mm.469 -   Allocation failure cycle start: newspace=0/8388608
| oldspace=38199368/50331648 loa=3523072/3523072 requestedbytes=48
| 3STHSTTYPE     15:18:14:893347364 GMT j9mm.133 -   Allocation failure start: newspace=0/8388608
| oldspace=38199368/50331648 loa=3523072/3523072 requestedbytes=48
| 3STHSTTYPE     15:18:14:866523613 GMT j9mm.134 -   Allocation failure end: newspace=2359064/8388608
| oldspace=38199368/50331648 loa=3523072/3523072
| 3STHSTTYPE     15:18:14:866519507 GMT j9mm.470 -   Allocation failure cycle end: newspace=2359296/8388608
| oldspace=38199368/50331648 loa=3523072/3523072
| 3STHSTTYPE     15:18:14:866513004 GMT j9mm.65 -   LocalGC end: rememberedsetoverflow=0
| causedrememberedsetoverflow=0 scancacheoverflow=0 failedflipcount=5056 failedflipbytes=445632
| failedtenurecount=0 failedtenurebytes=0 flipcount=9212 flipbytes=6017148 newspace=2359296/8388608
| oldspace=38199368/50331648 loa=3523072/3523072 tenureage=1
| 3STHSTTYPE     15:18:14:866493839 GMT j9mm.140 -   Tilt ratio: 64
| 3STHSTTYPE     15:18:14:859814852 GMT j9mm.64 -   LocalGC start: globalcount=3 scavengecount=23 weakrefs=0
| soft=0 phantom=0 finalizers=0
| 3STHSTTYPE     15:18:14:859808692 GMT j9mm.63 -   Set scavenger backout flag=false
| 3STHSTTYPE     15:18:14:859801848 GMT j9mm.135 -   Exclusive access: exclusiveaccessms=0.004
| meanexclusiveaccessms=0.004 threads=0 lastthreadtid=0x00495F00 beatenbyotherthread=0
| 3STHSTTYPE     15:18:14:859801163 GMT j9mm.469 -   Allocation failure cycle start: newspace=0/10747904
| oldspace=38985800/50331648 loa=3523072/3523072 requestedbytes=232
| 3STHSTTYPE     15:18:14:859800479 GMT j9mm.133 -   Allocation failure start: newspace=0/10747904
| oldspace=38985800/50331648 loa=3523072/3523072 requestedbytes=232
| 3STHSTTYPE     15:18:14:652219028 GMT j9mm.134 -   Allocation failure end: newspace=2868224/10747904
| oldspace=38985800/50331648 loa=3523072/3523072
| 3STHSTTYPE     15:18:14:650796714 GMT j9mm.470 -   Allocation failure cycle end: newspace=2868224/10747904
| oldspace=38985800/50331648 loa=3523072/3523072
| 3STHSTTYPE     15:18:14:650792607 GMT j9mm.475 -   GlobalGC end: workstackoverflow=0 overflowcount=0
| memory=41854024/61079552
| 3STHSTTYPE     15:18:14:650784052 GMT j9mm.90 -   GlobalGC collect complete
| 3STHSTTYPE     15:18:14:650780971 GMT j9mm.57 -   Sweep end
| 3STHSTTYPE     15:18:14:650611567 GMT j9mm.56 -   Sweep start
| 3STHSTTYPE     15:18:14:650610540 GMT j9mm.55 -   Mark end
| 3STHSTTYPE     15:18:14:645222792 GMT j9mm.54 -   Mark start
| 3STHSTTYPE     15:18:14:645216632 GMT j9mm.474 -   GlobalGC start: globalcount=2
|             (lines removed for clarity)
| NULL
| NULL           ------------------------------------------------------------------------
```

スレッドおよびスタック・トレース (THREADS)

アプリケーション・プログラマーにとって、Java ダンプの中で最も有用な部分の 1 つが THREADS セクションです。このセクションには、Java スレッド、ネイティブ・スレッド、およびスタック・トレースのリストが示されます。 IBM WebSphere Real Time for RT Linux については、リアルタイム・スレッドと非ヒープ・リアルタイム・スレッドも示されています。

Java スレッドは、オペレーティング・システムのネイティブ・スレッドによって実装されます。各スレッドは以下のような一連の行で表されます。

```
"main" J9VMThread:0x41D11D00, j9thread_t:0x003C65D8, java/lang/Thread:0x40BD6070, state:CW, prio=5
 (native thread ID:0xA98, native priority:0x5, native policy:UNKNOWN)
 Java callstack:
 at java/lang/Thread.sleep(Native Method)
 at java/lang/Thread.sleep(Thread.java:862)
 at mySleep.main(mySleep.java:31)
```

ps コマンドを使用する際に、オペレーティング・システムで Java スレッド名を表示できます。 **ps** コマンドの使用について詳しくは、 106 ページの『一般的なデバッグ手法』を参照してください。

非ヒープ・リアルタイム・スレッドから生成された Javadump では、一部の情報が欠落している場合があります。非ヒープ・リアルタイム・スレッドからはスレッド名オブジェクトが見えない場合は、実際のスレッド名ではなく、「(access error)」というテキストが示されます。

1 行目のプロパティーは、スレッド名、JVM スレッド構造のアドレスと Java スレッド・オブジェクトのアドレス、スレッドの状態、および Java スレッド優先順位です。2 行目のプロパティーは、ネイティブ・オペレーティング・システムのスレッド ID、ネイティブ・オペレーティング・システムのスレッド優先順位、およびネイティブ・オペレーティング・システムのスケジューリング・ポリシーです。

スレッド名は 3 とおりの方法で表示されます。

- javacore ファイル内にリストされる。すべてのスレッドが javacore ファイルにリストされるわけではありません。
- **ps** コマンドを使用して、オペレーティング・システムからスレッドをリストする場合。
- java.lang.Thread.getName() メソッドを使用している場合

以下の表に、IBM WebSphere Real Time for RT Linux のスレッド名に関する情報を示します。

表 12. IBM WebSphere Real Time for RT Linux のスレッド名

スレッドの詳細	スレッド名
2 次スレッドによるオブジェクトのファイナライズをディスパッチするために、ガーベッジ・コレクション・モジュールが使用する内部 JVM スレッド。	`Finalizer master`
ガーベッジ・コレクターが使用するアラーム・スレッド。	`GC Alarm`
ガーベッジ・コレクション用に使用されるスレーブ・スレッド。	`GC Slave`
アプリケーションにおけるメソッドの使用を抽出するために、JIT (Just-In-Time) コンパイラー・モジュールが使用する内部 JVM スレッド。	`JIT Sampler`
外部で生成されたか内部で生成されたかに関係なく、アプリケーションが受信したシグナルを管理するために VM が使用するスレッド。	`Signal Reporter`

Java コードで作成されたリアルタイム・スレッド (javax.realtime.RealtimeThread) のデフォルト名は、`RTThread-x` (『x』 はスレッド番号) です。

非ヒープ・リアルタイム・スレッドのデフォルト名は、`NHRTThread-x` (『x』 はスレッド番号) です。

Java のスレッド優先順位は、プラットフォームに依存しない方法でオペレーティング・システムの優先順位値にマップされます。 Java のスレッド優先順位の値が大きいほど、そのスレッドの優先順位が高いことを意味します。つまり、優先順位の高

いスレッドほど実行頻度が上がるということです。Java スレッド、リアルタイム・スレッド、および非ヒープ・リアルタイム・スレッドの場合の動作について詳しくは、 12 ページの『優先順位のマッピングおよび継承』を参照してください。

使用される状態値は以下のとおりです。

- R - 実行可能 (Runnable) - スレッドは必要に応じて実行可能です。
- CW - 待機状態 (Condition Wait) - スレッドは待機中です。以下のような理由が考えられます。
  - sleep() 呼び出しが実行された
  - スレッドで入出力がブロックされている
  - モニターに通知があるまで待機する wait() メソッドが呼び出された
  - スレッドが join() 呼び出しによって他のスレッドと同期中である
- S – 中断状態 (Suspended) - スレッドは他のスレッドによって中断されています。
- Z – ゾンビ (Zombie) – スレッドは強制終了されました。
- P – 保留状態 (Parked) – スレッドは新規の並行性 API (java.util.concurrent) によって保留されています。
- B – ブロック状態 (Blocked) – スレッドは現在他のものが所有しているロックの取得を待機しています。

スレッドが保留状態またはブロック状態の場合は、出力にそのスレッドについての行が含まれます。この行は、3XMTHREADBLOCK で始まり、スレッドが待機しているリソースと、そのリソースが現在所有しているスレッド (ある場合) がリストされます。詳しくは、IBM SDK for Java 7 ユーザー・ガイドのブロック状態のスレッドに関するトピックを参照してください。

Javadump を開始して診断情報を取得すると、JVM は javacore を生成する前に Java スレッドを休止します。exclusive_vm_access の準備状態が、TITLE セクションの 1TIPREPSTATE 行に表示されます。

```
1TIPREPSTATE Prep State: 0x4 (exclusive_vm_access)
```

javacore がトリガーされたときに Java コードを実行していたスレッドは、CW (待機状態) にあります。

```
3XMTHREADINFO      "main" J9VMThread:0x41481900, j9thread_t:0x002A54A4, java/lang/Thread:0x004316B8,
state:CW, prio=5
3XMTHREADINFO1            (native thread ID:0x904, native priority:0x5, native policy:UNKNOWN)
3XMTHREADINFO3           Java callstack:
4XESTACKTRACE                at java/lang/String.getChars(String.java:667)
4XESTACKTRACE                at java/lang/StringBuilder.append(StringBuilder.java:207)
```

javacore の LOCKS セクションには、これらのスレッドが内部の JVM ロックを待機していることが示されます。

```
2LKREGMON          Thread public flags mutex lock (0x002A5234): <unowned>
3LKNOTIFYQ            Waiting to be notified:
3LKWAITNOTIFY            "main" (0x41481900)
```

Heapdump の使用

Heapdump とは、Java ヒープ上にあるすべてのライブ・オブジェクトのダンプを生成する IBM Virtual Machine for Java のメカニズムを指します。ライブ・オブジェクトとは、実行中の Java アプリケーションによって使用されているオブジェクトのことです。

IBM SDK for Java 7 ユーザー・ガイドには、Heapdumps についての次のような有用なガイダンスが記載されています。

- Heapdump の取得
- Heapdump を処理するためのツール
- **-Xverbose:gc** を使用したヒープ情報の取得
- 環境変数と Heapdump
- テキスト (標準型) の Heapdump ファイル・フォーマット
- ポータブル Heapdump (PHD) ファイル・フォーマット

この情報は、IBM SDK for Java 7 - Heapdump の使用で参照できます。

IBM WebSphere Real Time for RT Linux の補足情報:

リアルタイム JVM に対する複数の Heapdump の有効化

生成された Heapdump は、デフォルトでは、すべてのメモリー領域、ヒープ・メモリー、永久メモリー、およびスコープ・メモリーにおけるすべての Java オブジェクトに関する情報を含む 1 つのファイルです。複数のダンプを生成する主な理由は、従来の Heapdump ツールを変更せずに使用して、各ヒープ領域を分析できるようにするためです。

このタスクについて

デフォルトでは、Heapdump には、JVM のメモリー領域、ヒープ、永続メモリー、およびスコープ・メモリー内のすべてのオブジェクトに関する情報が含まれています。 **request=multiple** オプションと **-Xdump:heap** を共に使用して、各メモリー領域における Java オブジェクトに関する情報を含む別々の Heapdump を取得できます。request オプションのデフォルト設定も同様に繰り返す必要があるため、`request=multiple+exclusive+prepwalk+compact` を指定する必要があることに注意してください。これにより、特定のメモリー領域を示すフィールドが名前の中に追加された、一連の Heapdump が次のように生成されます。

```
heapdump.%id.%Y%m%d.%H%M%S.%pid.phd
```

ここで、*%id* はヒープ・メモリー、永久メモリーにあるオブジェクト、またはスコープ・メモリーの特定の領域にあるオブジェクトを含む Heapdump ファイルを示します。

「Default」、「Immortal」、「Scope」、および「Other」という名前の 4 種類のヒープがあります。 Heapdump コードでは、ヒープ・ラベルの %id がこれらの名前うちのいずれかで置き換えられ、識別子 (通常は数値) で連結されます。例えば、`heapdump.Immortal12994208.20060807.093653.7684.txt.` などです。

例

```
java -Xrealtime -Xdump:heap:defaults:request=multiple+exclusive+compact+prepwalk
     <java program>
```

この追加オプションを使用すると、複数の Heapdump がポータブル Heapdump (phd) 形式で生成されます。

```
java -Xrealtime -Xdump:heap:defaults:request=multiple+exclusive+compact+prepwalk,
     opts=CLASSIC <java program>
```

この追加オプションを使用すると、複数の Heapdump が標準のテキスト形式で生成されます。

-Xdump:what オプションを使用すると、JVM 起動時にダンプ・エージェントが表示されます。このオプションは、現行のダンプ・オプションを確認するのに便利です。

テキスト (標準型) の Heapdump ファイル・フォーマット

テキスト (標準型) の Heapdump では、ヒープ内のすべてのオブジェクト・インスタンスが、オブジェクトの型とサイズ、およびオブジェクト間の参照を含めてリストされます。

ヘッダー・レコード

ヘッダー・レコードは、一連のバージョン情報が格納された単一のレコードです。

```
// Version:
<SDK レベル、プラットフォーム、および JVM ビルド・レベルが含まれたバージョン文字列>
```

例:

```
// Version: J2RE 7.0 IBM J9 2.6 Linux x86-32 build 20101016_024574_lHdRSr
```

オブジェクト・レコード

オブジェクト・レコードは複数のレコード (ヒープ上のオブジェクト・インスタンスごとに 1 つのレコード) からなり、オブジェクトのアドレス、サイズ、型、およびそのオブジェクトからの参照を表します。

```
<16 進値のオブジェクト・アドレス> [<10 進値のオブジェクト・インスタンスの長さ (バイト数)>]
OBJ <オブジェクト型> <16 進値のクラス・ブロック参照>
<16 進値のヒープ参照 <16 進値のヒープ参照> ...
```

オブジェクト・アドレスとヒープ参照はヒープ内にありますが、クラス・ブロック・アドレスはヒープ外にあります。オブジェクト・インスタンス内で見つかったすべての参照が (NULL 値の参照も含めて) リストされます。オブジェクト型は、パッケージを含むクラス名か、プリミティブ配列型またはクラス配列型であり、その標準 JVM 型シグニチャーによって示されます (135 ページの『Java VM の型シグニチャー』を参照)。オブジェクト・レコードには、追加のクラス・ブロック参照も格納できます (通常はリフレクション・クラス・インスタンスの場合)。

例:

長さが 28 バイトで java/lang/String 型のオブジェクト・インスタンス:

```
0x00436E90 [28] OBJ java/lang/String
```

java/lang/String のクラス・ブロック・アドレスと、char 配列インスタンスへの参照:

```
0x415319D8 0x00436EB0
```

長さが 44 バイトで char 配列型のオブジェクト・インスタンス:

```
0x00436EB0 [44] OBJ [C
```

char 配列のクラス・ブロック・アドレス:

```
0x41530F20
```

java/util/Hashtable Entry 内部クラスの配列型のオブジェクト:

```
0x004380C0 [108] OBJ [Ljava/util/Hashtable$Entry;
```

java/util/Hashtable Entry 内部クラス型のオブジェクト:

```
0x4158CD80 0x00000000 0x00000000 0x00000000 0x00000000 0x00421660 0x004381C0
0x00438130 0x00438160 0x00421618 0x00421690 0x00000000 0x00000000 0x00000000
0x00438178 0x004381A8 0x004381F0 0x00000000 0x004381D8 0x00000000 0x00438190
0x00000000 0x004216A8 0x00000000 0x00438130 [24] OBJ java/util/Hashtable$Entry
```

クラス・ブロック・アドレスとヒープ参照 (NULL 参照を含む):

```
0x4158CB88 0x004219B8 0x004341F0 0x00000000
```

クラス・レコード

クラス・レコードは複数のレコード (読み込まれたクラスごとに 1 つのレコード) からなり、クラス・ブロック・アドレス、サイズ、型、およびそのクラスからの参照を表します。

```
<16 進値のクラス・ブロック・アドレス> [<10 進値のクラス・ブロックの長さ (バイト数)>]
CLS <クラス型>
<16 進値のクラス・ブロック参照> <16 進値のクラス・ブロック参照> ...
<16 進値のヒープ参照> <16 進値のヒープ参照> ...
```

クラス・ブロック・アドレスとクラス・ブロック参照はヒープ外にありますが、クラス・レコードにはヒープ内への参照も格納できます (通常は静的クラス・データ・メンバーについて)。クラス・ブロック内で見つかったすべての参照が (NULL 値のものも含めて) リストされます。クラス型は、パッケージを含むクラス名か、プリミティブ配列型またはクラス配列型であり、その標準 JVM 型シグニチャーによって示されます (135 ページの『Java VM の型シグニチャー』を参照)。

例:

java/lang/Runnable クラスの長さが 32 バイトのクラス・ブロック:

```
0x41532E68 [32] CLS java/lang/Runnable
```

他のクラス・ブロックへの参照とヒープ参照 (NULL 参照を含む):

```
0x4152F018 0x41532E68 0x00000000 0x00000000 0x00499790
```

java/lang/Math クラスの長さが 168 バイトのクラス・ブロック:

```
0x00000000 0x004206A8 0x00420720 0x00420740 0x00420760 0x00420780 0x004207B0
0x00421208 0x00421270 0x00421290 0x004212B0 0x004213C8 0x00421458 0x00421478
0x00000000 0x41589DE0 0x00000000 0x4158B340 0x00000000 0x00000000 0x00000000
0x4158ACE8 0x00000000 0x4152F018 0x00000000 0x00000000 0x00000000
```

トレーラー・レコード 1

トレーラー・レコード 1 は、レコード数が格納された単一のレコードです。

```
// Breakdown - Classes: <10 進値のクラス・レコード数>,
Objects: <10 進値のオブジェクト・レコード数>,
ObjectArrays: <10 進値のオブジェクト配列レコード数>,
PrimitiveArrays: <10 進値のプリミティブ配列レコード数>
```

例:

```
// Breakdown - Classes: 321, Objects: 3718, ObjectArrays: 169,
PrimitiveArrays: 2141
```

トレーラー・レコード 2

トレーラー・レコード 2 は、合計数が格納された単一のレコードです。

```
// EOF:  Total 'Objects',Refs(null) :
<10 進値の合計オブジェクト数>,
<10 進値の合計参照数>
(,10 進値の合計 NULL 参照数>)
```

例:

```
// EOF:  Total 'Objects',Refs(null) : 6349,23240(7282)
```

Java VM の型シグニチャー

Java VM の型シグニチャーは、次の表に示す Java 型の略記です。

Java VM の型シグニチャー	Java 型
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double
L <*完全修飾クラス*> ;	<*完全修飾クラス*>
[<*型*>	<*型*>[ ] (<*型*> の配列)
(<*引数の型*>) <*戻り値の型*>	メソッド

システム・ダンプおよびダンプ・ビューアーの使用

JVM は、ユーザーによって構成可能な状況下でネイティブ・システム・ダンプ (コア・ダンプとも呼ばれます) を生成できます。システム・ダンプは一般的に大きいサイズになります。また、システム・ダンプを分析するために使用されるほとんどのツールは、プラットフォーム固有です。Linux 上でシステム・ダンプを分析するには、**gdb** ツールを使用します。

IBM SDK for Java 7 ユーザー・ガイドには、システム・ダンプとダンプ・ビューアーの使用についての、次のような有用なガイドラインが記載されています。

- システム・ダンプの概要
- システム・ダンプのデフォルト
- ダンプ・ビューアーの使用
  - **jextract** の使用
  - ダンプ・ビューアーを使用して対処すべき問題
  - **jdmpview** で使用可能なコマンド
  - セッション例
  - **jdmpview** コマンドのクイック・リファレンス

この情報は、IBM SDK for Java 7 - システム・ダンプおよびダンプ・ビューアーの使用で参照できます。

IBM WebSphere Real Time for RT Linux の補足情報:

jextract の使用

リアルタイム JVM からシステム・ダンプを処理する場合は、**-Xrealtime** オプションを指定する必要があります。例えば、次のようにします。

```
jextract -Xrealtime <core file name> [<zip_file>]
```

jextract を、ダンプが生成された JVM とは異なる JVM で実行すると、以下のエラー・メッセージが表示されます。

```
J9RAS.buildID is incorrect (found e8801ed67d21c6be, expecting eb4173107d21c673).
This version of jextract is incompatible with this dump.
Failure detected during jextract, see previous message(s).
```

同様に、標準の JVM で Java を実行していても、**jextract** を使用してダンプを処理するときに **-Xrealtime** オプションを使用した場合にも、このメッセージが生成されます。

jdmpview で使用可能なコマンド

jdmpview は対話型のコマンド行ツールであり、JVM システム・ダンプから得られる情報を調べて、各種の分析機能を実行できます。

info jitm
AOT および JIT コンパイル済みメソッドとそのアドレスの一覧を表示します。
- メソッドの名前とシグニチャー
- メソッドの開始アドレス
- メソッドの終了アドレス

その他のコマンド・オプションについて詳しくは、IBM SDK for Java 7 ユーザー・ガイドを参照してください。

Java アプリケーションと JVM のトレース

JVM トレースとは、IBM WebSphere Real Time for RT Linux で提供されるトレース機能であり、パフォーマンスに及ぼす影響を最小限に抑えることができます。大部分の場合、トレース・データは圧縮されたバイナリー・フォーマットで保持されます。これは、提供されている Java フォーマッターでフォーマット設定できます。

デフォルトで、トレースは使用可能に設定されており、トレース・ポイントの小規模なセットがメモリー・バッファーに入れられます。レベル、コンポーネント、グループ名、または個々のトレース・ポイント ID を使用して、実行時にトレース・ポイントを使用可能に設定することができます。

IBM SDK for Java 7 ユーザー・ガイドには、アプリケーションのトレースに関する以下のような詳細情報が記載されています。

- トレース可能なもの
- トレース・ポイントのタイプ
- デフォルト・トレース
- トレース・データの記録
- トレースの制御
- Java アプリケーションのトレース
- Java メソッドのトレース

IBM WebSphere Real Time for RT Linux をトレースする際は、トレース・オプションを指定するときにリアルタイム JVM を正しく起動する必要があります。例えば、トレース・オプションを指定するときは、次のように入力します。

```
java -Xrealtime -Xtrace:<options>
```

IBM SDK for Java 7 の情報については、Java アプリケーションと JVM のトレースで参照できます。

JIT および AOT の問題判別

コマンド行オプションは、JIT および AOT コンパイラーの問題診断に役立つほか、パフォーマンスの調整も行えます。

IBM WebSphere Real Time for RT Linux は一部の共通コンポーネントを IBM SDK for Java 7 と共有しますが、JIT と AOT の振る舞いは異なります。このセクションでは、IBM WebSphere Real Time for RT Linux での JIT と AOT の問題のトラブルシューティングについて説明します。

JIT または AOT の問題診断

場合により、有効なバイトコードをコンパイルした結果、無効なネイティブ・コードが生成され、Java プログラムで障害が発生することがあります。 JIT または AOT コンパイラーに欠陥があるか、また、欠陥がある場合にはどこに 欠陥があるかを判別することによって、Java サービス・チームに有益な情報を提供することができます。

このタスクについて

共有クラス・キャッシュへのデータの追加時にどのメソッドがコンパイルされていたのかを判別するには、admincache コマンド行で **-Xaot:verbose** オプションを使用します。例えば、次のようにします。

```
admincache -Xrealtime -Xaot:verbose -populate -aot my.jar -cp <My Class Path>
```

このセクションでは、問題がコンパイラーに関連したものであるかどうかを判別する方法について説明します。このセクションではまた、コンパイラーに関連した問

題を解決する上で考えられる回避策およびデバッグ技法を提案します。

JIT または AOT コンパイラーの無効化:

問題が JIT または AOT コンパイラーで発生していると疑われる場合には、コンパイルを無効にして、引き続き問題が発生するかを確認します。引き続き問題が発生する場合には、コンパイラーが原因ではないと分かります。

このタスクについて

JIT コンパイラーはデフォルトで有効になっています。 AOT コンパイラーも有効になっていますが、共有クラスが有効にされていない限り、アクティブにはなりません。効率性の点から、Java アプリケーションのすべてのメソッドがコンパイルされるわけではありません。 JVM はアプリケーションの各メソッドの呼び出し回数を保持しています。メソッドの呼び出し回数は、そのメソッドが呼び出されるか解釈されるたびに増えていきます。回数がコンパイルのしきい値に達すると、メソッドがコンパイルされ、ネイティブに実行されます。

呼び出し回数メカニズムにより、メソッドのコンパイルはアプリケーションの存続時間全体にわたって分散されます。使用頻度の高いメソッドほど優先順位が高くなります。使用頻度の低い一部のメソッドは、一度もコンパイルされない可能性があります。結果として、Java プログラムで障害が発生した場合、それは JIT または AOT コンパイラーの問題である可能性も、JVM の他の場所における問題である可能性もあります。

障害を診断する最初のステップは、問題がどこで 発生しているかを判別することです。そのためには、まず、純粋に解釈を行うモードで (つまり、JIT および AOT コンパイラーを無効にして) Java プログラムを実行する必要があります。

手順

1. コマンド行からすべての **-Xjit** オプションと **-Xaot** オプション (およびそのパラメーター) を削除します。
2. **-Xint** コマンド行オプションを使用して、JIT および AOT コンパイラーを無効にします。 パフォーマンス上の理由から、実稼働環境では **-Xint** オプションを使用しないようにしてください。

次のタスク

コンパイルを無効にして Java プログラムを実行すると、以下のいずれかの結果が得られます。

- 引き続き障害が発生します。この問題は、JIT または AOT コンパイラーが原因ではありません。場合によっては、プログラムでの問題の発生の仕方が変わることがありますが、その場合でも、この問題とコンパイラーは関係ありません。
- 障害が発生しなくなります。この問題は、かなりの確率で JIT または AOT コンパイラーが原因です。

  共有クラスを使用していない場合は、JIT コンパイラーに障害の原因があります。共有クラスを使用している場合は、JIT コンパイルのみを有効にしてアプリケーションを実行し、どのコンパイラーに障害の原因があるのかを判別する必要

があります。 **-Xint** オプションの代わりに **-Xnoaot** オプションを使用してアプリケーションを実行します。これにより、以下のいずれかの結果が得られます。
  - 引き続き障害が発生します。この問題は JIT コンパイラーが原因です。 **-Xnoaot** オプションの代わりに **-Xnojit** オプションを使用して、JIT コンパイラーのみが障害の原因であるかを確認することもできます。
  - 障害が発生しなくなります。この問題は AOT コンパイラーが原因です。

JIT (Just-In-Time) コンパイラーの選択的な無効化:

Java プログラム障害の原因が JIT (Just-In-Time) コンパイラーの問題にあると思われる場合は、さらに問題を絞り込んでみてください。

このタスクについて

デフォルトでは、JIT コンパイラーはさまざまな最適化レベルでメソッドを最適化します。それぞれの呼び出し回数に基づいて、各メソッドに異なる最適化が選択されて適用されています。呼び出し頻度の高いメソッドほど、より高いレベルで最適化されます。 JIT コンパイラーのパラメーターを変更することで、メソッドの最適化レベルを制御できます。最適化プログラムが障害の原因であるのか、また、最適化プログラムが原因である場合には、どの最適化に問題があるのかを判別できます。

-Xjit オプションに付加する JIT パラメーターを、コンマ区切りのリストで指定します。構文は、**-Xjit**:<*param1*>,<*param2*>=<*value*> です。例えば、次のようにします。

```
java -Xjit:verbose,optLevel=noOpt HelloWorld
```

この場合は、 HelloWorld プログラムが実行され、JIT からの詳細出力が有効にされて、最適化は一切実行せずに JIT にネイティブ・コードを生成させます。

コンパイラーのどの部分が障害の原因となっているのかを判別するには、以下のステップに従います。

手順

1. JIT のパラメーター **count=0** を設定して、コンパイルのしきい値をゼロに変更します。このパラメーターにより、各 Java メソッドが実行前にコンパイルされるようになります。 **count=0** は、問題を診断する場合にのみ使用してください。これは、使用頻度が低いメソッドを含め、さらに多くのメソッドがコンパイルされるためです。追加のコンパイルにより、より多くの計算リソースが使用され、アプリケーションの処理速度が低下します。 **count=0** の場合、問題の領域に達すると、アプリケーションで即時に障害が発生します。場合によっては、**count=1** を使用することで、より確実に障害を再現できることがあります。
2. **disableInlining** を、JIT (Just-In-Time) コンパイラーのパラメーターに追加します。 **disableInlining** は、比較的大きく複雑なコードの生成を無効にします。問題が発生しなくなった場合は、Java サービス・チームがコンパイラーの問題を分析して修正するまでの間、回避策として **disableInlining** を使用してください。
3. **optLevel** パラメーターを追加することによって最適化レベルを下げ、障害が発生しなくなるか、「noOpt」レベルに達するまでプログラムを再実行します。

JIT (Just-In-Time) コンパイラーの問題の場合は、「scorching」から始めて、リストの降順に作業を進めてください。 最適化レベルを降順に示すと、以下のとおりになります。

a. scorching
b. veryHot
c. hot
d. warm
e. cold
f. noOpt

次のタスク

これらの設定のいずれかで障害が発生しなくなった場合は、それが使用可能な回避策になります。これは、Java サービス・チームがコンパイラーの問題を分析して修正する間の一時的な回避策です。JIT のパラメーター・リストから **disableInlining** を削除しても障害が再発しない場合には、パフォーマンスを改善するために削除してください。『障害が発生したメソッドの特定』の指示に従って、回避策のパフォーマンスを改善してください。

「noOpt」最適化レベルでも引き続き障害が発生する場合は、回避策として JIT (Just-In-Time) コンパイラーを無効にする必要があります。

障害が発生したメソッドの特定:

障害を引き起こすメソッドを JIT または AOT コンパイラーがコンパイルしなければならない最も低い最適化レベルを判別したら、コンパイル時に Java プログラムのどの部分が障害を引き起こしているのかを突き止めることができます。その後は、コンパイラーに命令して特定のメソッド、クラス、またはパッケージに回避策を限定し、コンパイラーがプログラムの残りの部分を通常どおりにコンパイルできるようにします。 JIT コンパイラーの障害で、**-Xjit:optLevel=noOpt** を使用しても障害が発生する場合は、その障害を引き起こしているメソッドを決してコンパイルしないようにコンパイラーに命令することもできます。

始める前に

以下の例のようなエラー出力がある場合は、これを使用して障害が発生しているメソッドを特定できます。

```
Unhandled exception
Type=Segmentation error vmState=0x00000000
Target=2_30_20050520_01866_BHdSMr (Linux 2.4.21-27.0.2.EL)
CPU=s390x (2 logical CPUs) (0x7b6a8000 RAM)
J9Generic_Signal_Number=00000004 Signal_Number=0000000b Error_Value=4148bf20 Signal_Code=00000001
Handler1=00000100002ADB14 Handler2=00000100002F480C InaccessibleAddress=0000000000000000
gpr0=0000000000000006 gpr1=0000000000000006 gpr2=0000000000000000 gpr3=0000000000000006
gpr4=0000000000000001 gpr5=0000000080056808 gpr6=0000010002BCCA20 gpr7=0000000000000000
......
Compiled_method=java/security/AccessController.toArrayOfProtectionDomains([Ljava/lang/Object;
Ljava/security/AccessControlContext;)[Ljava/security/ProtectionDomain;
```

重要な行は以下のとおりです。

vmState=0x00000000
: 障害が発生したコードが、JVM ランタイム・コードではないことを示します。

Module= or Module_base_address=
: このコードは JIT によって、DLL またはライブラリーの外側でコンパイルされたため、この出力にはありません (ブランクやゼロである場合もあります)。

Compiled_method=
: コンパイル済みコードの生成の対象となる Java メソッドを示します。

このタスクについて

障害が発生しているメソッドが出力に示されていない場合は、以下のステップに従って、障害が発生しているメソッドを特定してください。

手順

1. JIT のパラメーター **verbose** および **vlog=**<*filename*> を **-Xjit** または **-Xaot** オプションに追加して Java プログラムを実行します。 これらのパラメーターが使用されている場合、コンパイラーは、コンパイルしたメソッドを <*filename*>.<*date*>.<*time*>.<*pid*> という名前のログ・ファイルにリストします。このファイルは、*しきい値ファイル* とも呼ばれます。典型的なしきい値ファイルには、コンパイルされたメソッドに対応する以下のような行が含まれています。

   ```
   + (hot) java/lang/Math.max(II)I @ 0x10C11DA4-0x10C11DDD
   ```

   先頭に正符号がない行は、以降のステップでコンパイラーによって無視されるため、ファイルから削除して構いません。 AOT コードが共有クラス・キャッシュからロードされたメソッドの先頭には、`+ (AOT load)` が付きます。
2. JIT または AOT パラメーター **limitFile=**(<*filename*>,<*m*>,<*n*>) を使用してプログラムを再度実行します。ここで、<*filename*> はしきい値ファイルのパス、<*m*> および <*n*> はしきい値ファイル内の最初と最後のコンパイル対象メソッドを示す行番号です。 コンパイラーは、しきい値ファイル内の <*m*> から <*n*> までの行にリストされたメソッドのみをコンパイルします。しきい値ファイルにリストされていないメソッド、およびこの範囲外の行にリストされているメソッドはコンパイルされません。また、これらのメソッドの共有データ・キャッシュにある AOT コードはロードされません。 プログラムで障害が発生しなくなった場合、恐らくは最後の反復で削除した 1 つ以上のメソッドが障害の原因であると考えられます。
3. <*m*> および <*n*> に異なる値を使用して必要な回数だけこのプロセスを繰り返し、コンパイル時に障害を引き起こす最小限のメソッド・セットを見つけてください。 毎回、選択する行の数を半数にしていくことで、障害が発生しているメソッドの二分探索を実行できます。 多くの場合、ファイルを 1 行になるまで減らしていくことができます。

次のタスク

障害が発生しているメソッドが見つかった場合には、そのメソッドに対してのみ、JIT または AOT コンパイラーを無効にすることができます。例えば、

optLevel=hot を使用して JIT コンパイルを実行する際にメソッド java/lang/Math.max(II)I がプログラムの障害を引き起こしている場合は、以下を使用してプログラムを実行します。

```
-Xjit:{java/lang/Math.max(II)I}(optLevel=warm,count=0)
```

これにより、障害が発生しているメソッドだけ「warm」最適化レベルでコンパイルされ、他のすべてのメソッドは通常どおりにコンパイルされます。

「noOpt」で JIT コンパイルした際に障害が発生するメソッドについては、**exclude=**{<*method*>} パラメーターを使用して、コンパイルの対象から完全に除外できます。

```
-Xjit:exclude={java/lang/Math.max(II)I}
```

共有データ・キャッシュから AOT コードをロードした際にメソッドがプログラムの障害を引き起こす場合は、**exclude=**{<*method*>} パラメーターを使用して、そのメソッドを AOT ロードから除外します。

```
-Xaot:exclude={java/lang/Math.max(II)I}
```

admincache へのデータ追加ステップでは、コンパイルされた AOT メソッドが必ず共有クラス・キャッシュに格納されます。これらのメソッドに関する問題については、AOT ロードを行わないことが最善の診断方法です。

JIT および AOT コンパイルの障害の特定:

JIT コンパイラーの障害が発生した場合は、エラー出力を分析して、JIT コンパイラーがメソッドのコンパイルを試行した際に障害が発生したのかどうかを判別してください。

JVM が異常終了し、その障害が JIT ライブラリー (libj9jit26.so) で起こったことが分かる場合は、JIT (Just-In-Time) コンパイラーがメソッドをコンパイルしようとしている間に障害が発生した可能性があります。

以下の例のようなエラー出力がある場合は、これを使用して障害が発生しているメソッドを特定できます。

```
Unhandled exception
Type=Segmentation error vmState=0x00050000
Target=2_30_20051215_04381_BHdSMr (Linux 2.4.21-32.0.1.EL)
CPU=ppc64 (4 logical CPUs) (0xebf4e000 RAM)
J9Generic_Signal_Number=00000004 Signal_Number=0000000b Error_Value=00000000 Signal_Code=00000001
Handler1=0000007FE05645B8 Handler2=0000007FE0615C20
R0=E8D4001870C00001 R1=0000007FF49181E0 R2=0000007FE2FBCEE0 R3=0000007FF4E60D70
R4=E8D4001870C00000 R5=0000007FE2E02D30 R6=0000007FF4C0F188 R7=0000007FE2F8C290
......
Module=/home/test/sdk/jre/bin/libj9jit26.so
Module_base_address=0000007FE29A6000
......
Method_being_compiled=com/sun/tools/javac/comp/Attr.visitMethodDef(Lcom/sun/tools/javac/tree/
JCTree$JCMethodDecl;)
```

重要な行は以下のとおりです。

vmState=0x00050000
JIT コンパイラーがコードをコンパイルしていることを示します。 vmState のコード番号のリストについては、「IBM SDK for Java 7 ユーザー・ガイド

(IBM SDK for Java 7 User guide)」 (http://publib.boulder.ibm.com/infocenter/java7sdk/v7r0/topic/com.ibm.java.lnx.70.doc/diag/tools/javadump_tags_info.html) に記載された Javadump タグの表を参照してください。

Module=/home/test/sdk/jre/bin/libj9jit26.so
JIT (Just-In-Time) コンパイラーのモジュール libj9jit26.so でエラーが発生したことを示します。

Method_being_compiled=
コンパイル対象の Java メソッドを示します。

障害が発生しているメソッドが出力に示されていない場合は、以下の追加設定とともに **verbose** オプションを使用してください。

```
-Xjit:verbose={compileStart|compileEnd}
```

これらの **verbose** 設定により、JIT または AOT コンパイラーがメソッドのコンパイルを開始した時刻および終了した時刻が報告されるようになります。 JIT または AOT コンパイラーが特定のメソッドで障害を起こしている場合 (つまり、コンパイルは開始されるが、終了する前に異常終了する場合) は、 **exclude** パラメーターを使用して、そのメソッドを JIT または AOT のコンパイルから除外してください (140 ページの『障害が発生したメソッドの特定』を参照してください)。 AOT のコンパイルに関する問題である場合には、**exclude** オプションを使用する前に、共有クラス・キャッシュを破棄してください。メソッドを除外することで異常終了を防げる場合には、それがサービス・チームが問題を修正するまでの回避策になります。

非リアルタイム・モードでの AOT コンパイルの障害の特定:

非リアルタイム・モードでの AOT の問題判別は、JIT の問題判別によく似ています。

このタスクについて

JIT の場合と同様、まずは **-Xnoaot** を使用してアプリケーションを実行します。これにより、アプリケーションを実行する際に、AOT 化されたコードが使用されなくなります。

これで問題が修正された場合は、 140 ページの『障害が発生したメソッドの特定』に記載されている手法を使用し、アプリケーションの実行時ではなく、AOT のビルド時に **-Xaot** オプションを指定して AOT の JAR ファイルを再ビルドします。

リアルタイム・モードでの AOT コンパイルの障害の特定:

AOT の問題判別は、admincache ツールを使用して問題を特定します。

このタスクについて

アプリケーション・ランタイムで発生する JIT コンパイルの障害とは対照的に、AOT コンパイルの障害が admincache のデータ追加ステップで発生します。

問題が発生した場所を特定するために、admincache ツールを **-Xnoaot** オプションと共に実行します。これにより、アプリケーションは Ahead-Of-Time コンパイル・コードで実行されなくなります。

-Xnoaot オプションを使用することで問題が解決した場合、最初のクラッシュの出力を検査してください。この出力には、どのメソッドが問題の原因であるかを特定する情報が示されています。以下のような行を探してください。

```
Method_being_compiled=myAppClass.main(Ljava/lang/String;)V
```

問題を回避するために、このメソッドは Ahead-Of-Time コンパイルから除外する必要があります。これを実行するには、admincache コマンド行に以下のようなオプションを追加します。

```
-Xaot:exclude={myAppClass.main(Ljava/lang/String;)V}
```

この除外により、問題メソッドの AOT コンパイルが回避されます。

短期実行アプリケーションのパフォーマンス

IBM JIT コンパイラーは、通常はサーバーで使用される長期実行アプリケーションに合わせて調整されています。 **-Xquickstart** コマンド行オプションを非リアルタイム・モードで使用すると、短期実行アプリケーション、特に処理が少数のメソッドに集中していないアプリケーションのパフォーマンスを向上させることができます。

-Xquickstart を使用すると、JIT コンパイラーがデフォルトでより低い最適化レベルを使用するようになり、コンパイルされるメソッドの数が少なくなります。より少ない回数のコンパイルをより短い時間で実行することにより、アプリケーションの開始時間を短縮できます。 AOT コンパイラーがアクティブになっている (共有クラスと AOT コンパイルの両方が有効になっている) 場合に **-Xquickstart** を使用すると、コンパイルの対象として選択されているすべてのメソッドが AOT コンパイルされ、以後実行した場合の開始時間が短縮されます。大量の処理リソースを使用するメソッドが含まれた長期実行アプリケーションで **-Xquickstart** を使用した場合、パフォーマンスが低下する可能性があります。**-Xquickstart** の実装は、今後のリリースで変更されることがあります。

開始時間は、JIT のしきい値を (試行錯誤して) 調整することによって短縮することもできます。詳しくは、 139 ページの『JIT (Just-In-Time) コンパイラーの選択的な無効化』を参照してください。

-Xrealtime を使用している場合、AOT コードに対して **-Xquickstart** を使用しても効果はありません。

アイドル期間中の JVM の動作

-XsamplingExpirationTime オプションを使用して JIT のサンプリング・スレッドをオフにすることで、アイドル状態の JVM が消費する CPU サイクルを削減することができます。

JIT のサンプリング・スレッドは、実行中の Java アプリケーションに関するプロファイルを作成して、頻繁に使用されているメソッドをディスカバーします。サンプリング・スレッドによるメモリーおよびプロセッサーの使用量はごくわずかです。また、プロファイルの作成頻度は、JVM がアイドル状態の場合には自動的に引き下げられます。

アイドル状態の JVM に CPU サイクルを一切消費させないようにする必要がある場合もあります。その場合には、**-XsamplingExpirationTime**<*time*> オプションを指

定します。 <*time*> には、サンプリング・スレッドの実行時間を秒数で設定します。このオプションは慎重に使用してください。このオプションをオフにした後でサンプリング・スレッドを再度アクティブにすることはできません。サンプリング・スレッドの実行には、肝心の最適化を識別できるだけの十分な時間を与えてください。

Diagnostics Collector

Diagnostics Collector は、問題のあるイベントの Java 診断ファイルを収集します。

IBM サービスから要求されるファイルを収集することにより、報告された問題の解決にかかる時間を短縮できます。IBM SDK for Java 7 ユーザー・ガイドには、Diagnostics Collector の使用に関する詳細情報が記載されています。

この情報は、「IBM SDK for Java 7 - Diagnostics Collector」に記載されています。

ガーベッジ・コレクターの診断

このセクションでは、ガーベッジ・コレクションの問題を診断する方法を説明します。

IBM SDK for Java 7 ユーザー・ガイドには、ガーベッジ・コレクターの問題の診断についての次のような有用なガイダンスが記載されています。

- 詳細ガーベッジ・コレクションのロギング
- **-Xtgc** を使用したガーベッジ・コレクションのトレース

この情報は、IBM SDK for Java 7 - ガーベッジ・コレクターの診断で参照できます。

IBM WebSphere Real Time for RT Linux の Metronome ガーベッジ・コレクターに関する補足情報は、以下のセクションに記載されています。

Metronome ガーベッジ・コレクターのトラブルシューティング

コマンド行オプションを使用することで、Metronome ガーベッジ・コレクションの頻度、メモリー不足例外、および明示的なシステム呼び出しでの Metronome の動作を制御できます。

verbose:gc 情報の使用:

-verbose:gc オプションを **-Xgc:verboseGCCycleTime=N** オプションとともに使用して、Metronome ガーベッジ・コレクター・アクティビティーに関する情報をコンソールに書き込むことができます。標準の JVM からの **-verbose:gc** 出力ですべての XML プロパティーが作成されるわけではなく、Metronome ガーベッジ・コレクターの出力に適用されるわけでもありません。

-verbose:gc オプションを使用して、ヒープ内の最小、最大、および平均フリー・スペースを表示します。これにより、ヒープのアクティビティーや使用状況のレベルを確認し、必要に応じて値を調整することができます。この **-verbose:gc** オプションを使用すると、Metronome の統計がコンソールに書き込まれます。

-Xgc:verboseGCCycleTime=N オプションでは、情報の検索頻度を制御します。このオプションにより、ミリ秒単位の要約のダンプ時間が決まります。N のデフォルト

値は 1000 ミリ秒です。要約がダンプされるのは、サイクル・タイムに指定された時点ちょうどではなく、この時間の基準を満たす最後のガーベッジ・コレクション・イベント時です。これらの統計の収集および表示により、Metronome ガーベッジ・コレクターによる休止時間が増加する可能性があります。また、N の値が小さくなるほど、休止時間が長くなる可能性があります。

クォンタは、アプリケーションの中断時間または休止時間を生じさせる、Metronome ガーベッジ・コレクター・アクティビティーの単一期間です。

verbose:gc 出力の例

以下のように入力します。

```
java -Xrealtime -verbose:gc -Xgc:verboseGCCycleTime=N myApplication
```

この例では、バージョンおよびガーベッジ・コレクションの設定を含む、verbose:gc の初期出力が示されています。

```
<verbosegc
xmlns="http://www.ibm.com/j9/verbosegc" version="R26_Java726_GA_20110716_0946_B87065">

< 「initialized id="1" timestamp="2011-07-27T14:17:52.277">
  <attribute name="gcPolicy" value="-Xgcpolicy:metronome"/>
  <attribute name="maxHeapSize" value="0x5800000"/>
  <attribute name="initialHeapSize" value="0x4000000"/>
  <attribute name="compressedRefs" value="false"/>
  <attribute name="pageSize" value="0x1000"/>
  <attribute name="requestedPageSize" value="0x1000"/>
  <attribute name="gcthreads" value="1"/>
  <region>
    <attribute name="regionSize" value="16384"/>
    <attribute name="regionCount" value="4096"/>
    <attribute name="arrayletLeafSize" value="2048"/>
  </region>
  <metronome>
    <attribute name="beatsPerMeasure" value="500"/>
    <attribute name="timeInterval" value="10000"/>
    <attribute name="targetUtilization" value="70"/>
    <attribute name="trigger" value="0x2000000"/>
    <attribute name="headRoom" value="0x100000"/>
  </metronome>
  <system>
    <attribute name="physicalMemory" value="12507463680"/>
    <attribute name="numCPUs" value="8"/>
    <attribute name="architecture" value="x86"/>
    <attribute name="os" value="Linux"/>
    <attribute name="osVersion" value="2.6.24.7-75ibmrt2.18"/>
  </system>
  <vmargs>
    <vmarg
name="-Xoptionsfile=/my_dir/pxi3270hrt-20110719_02/sdk/jre/lib/i386/realtime/options.default"/>
    <vmarg name="-Xjcl:jclse7b_26"/>
    <vmarg
name="-Dcom.ibm.oti.vm.bootstrap.library.path=/my_dir/pxi3270hrt-20110719_02/sdk/jre/lib/i386/realtime:/
my_dir/pxi3270hrt-2011071..."/>
    <vmarg
name="-Dsun.boot.library.path=/my_dir/pxi3270hrt-20110719_02/sdk/jre/lib/i386/realtime:/my_dir/
pxi3270hrt-20110719_02/sdk/jre/lib..."/>
    <vmarg
name="-Djava.library.path=/my_dir/pxi3270hrt-20110719_02/sdk/jre/lib/i386/realtime:/my_dir/
pxi3270hrt-20110719_02/sdk/jre/lib/i38..."/>
    <vmarg name="-Djava.home=/my_dir/pxi3270hrt-20110719_02/sdk/jre"/>
    <vmarg name="-Djava.ext.dirs=/my_dir/pxi3270hrt-20110719_02/sdk/jre/lib/ext"/>
    <vmarg name="-Duser.dir=/my_dir/pxi3270hrt-20110719_02/sdk/jre/bin"/>
```

```
    <vmarg name="_j2se_j9=1120000"
value="F76FF700"/>
    <vmarg name="-Djava.runtime.version=pxi3270hrt-20110719_02"/>
    <vmarg name="-Djava.class.path=."/>
    <vmarg name="-Xrealtime"/>
    <vmarg name="-verbose:gc"/>
    <vmarg name="-Dsun.java.launcher=SUN_STANDARD"/>
    <vmarg name="-Dsun.java.launcher.pid=5543"/>
    <vmarg name="_port_library" value="F7701B80"/>
    <vmarg name="_bfu_java" value="F77029A8"/>
    <vmarg name="_org.apache.harmony.vmi.portlib" value="08051DA0"/>
  </vmargs>
</initialized>
```

ガーベッジ・コレクションが起動されると、`trigger start` イベントが発生し、その後に任意の数の `heartbeat` イベントが続き、起動の完了時に `trigger end` イベントが発生します。この例では、起動されたガーベッジ・コレクション・サイクルが verbose:gc 出力として示されています。

```
<trigger-start id="25" timestamp="2011-07-12T09:32:04.503" />

<cycle-start id="26" type="global" contextid="26" timestamp="2011-07-12T09:32:04.503" intervalms="984.285" />

<gc-op id="27" type="heartbeat" contextid="26" timestamp="2011-07-12T09:32:05.209">
  <quanta quantumCount="321" quantumType="mark" minTimeMs="0.367" meanTimeMs="0.524" maxTimeMs="1.878"
    maxTimestampMs="598704.070" />
  <exclusiveaccess-info minTimeMs="0.006" meanTimeMs="0.062" maxTimeMs="0.147" />
  <free-mem type="heap" minBytes="99143592" meanBytes="114374153" maxBytes="134182032" />
  <free-mem type="immortal" minBytes="44234538" meanBytes="60342344" maxBytes="61219900"/>
  <thread-priority maxPriority="11" minPriority="11" />
</gc-op>

<gc-op id="28" type="heartbeat" contextid="26" timestamp="2011-07-12T09:32:05.458">
  <quanta quantumCount="115" quantumType="sweep" minTimeMs="0.430" meanTimeMs="0.471" maxTimeMs="0.511"
    maxTimestampMs="599475.654" />
  <exclusiveaccess-info minTimeMs="0.007" meanTimeMs="0.067" maxTimeMs="0.173" />
  <classunload-info classloadersunloaded=9 classesunloaded=156  />
  <references type="weak" cleared="660" />
  <free-mem type="heap" minBytes="24281568" meanBytes="55456028" maxBytes="87231320" />
  <free-mem type="immortal" minBytes="38234500" meanBytes="41736440" maxBytes="42233458"/>
  <thread-priority maxPriority="11" minPriority="11" />
</gc-op>

<gc-op id="29" type="syncgc" timems="136.945" contextid="26" timestamp="2011-07-12T09:32:06.046">
  <syncgc-info reason="out of memory" exclusiveaccessTimeMs="0.006" threadPriority="11" />
  <free-mem-delta type="heap" bytesBefore="21290752" bytesAfter="171963656" />
  <free-mem-delta type="immortal" bytesBefore="35735400" bytesAfter="35735400"/>
</gc-op>

<cycle-end id="30" type="global" contextid="26" timestamp="2011-07-12T09:32:06.046" />

<trigger-end id="31" timestamp="2011-07-12T09:32:06.046" />
```

発生する可能性のあるイベントのタイプは以下のとおりです。

<trigger-start ...>
: 使用メモリー量がトリガーしきい値を上回ったことによる、ガーベッジ・コレクション・サイクルの開始。デフォルトのしきい値はヒープの 50% です。`intervalms` 属性は、直前の `trigger end` イベント (ID は -1) と現行の `trigger start` イベントとの間隔を表します。

<trigger-end ...>
: ガーベッジ・コレクション・サイクルにより、使用済みメモリー量がトリガーしきい値未満まで下がりました。ガーベッジ・コレクション・サイクルが

終了しても、使用済みメモリーがトリガーしきい値を下回らなかった場合は、新しいガーベッジ・コレクション・サイクルが、同じコンテキスト ID で開始されます。`trigger start` イベントごとに、対応する同じコンテキスト ID の `trigger end` イベントがあります。`intervalms` 属性は、直前の `trigger start` イベントと現行の `trigger end` イベントとの間隔を表します。この間に、使用済みメモリーがトリガーしきい値を下回るまで、1 つ以上のガーベッジ・コレクション・サイクルが完了します。

<gc-op id="28" type="heartbeat"...>
対象となる期間中のすべてのガーベッジ・コレクション・クォンタに関する (メモリーや時間などの) 情報を収集する定期的なイベント。 heartbeat イベントは、対応する `trigger start` イベントと `trigger end` イベントの組の間 (つまり、アクティブ・ガーベッジ・コレクション・サイクルが進行中である間) にのみ発生する可能性があります。`intervalms` 属性は、直前の heartbeat イベント (ID は -1) と現行の heartbeat イベントとの間隔を示します。

<gc-op id="29" type="syncgc"...>
同期 (非決定論的な) ガーベッジ・コレクション・イベント。 149 ページの『同期ガーベッジ・コレクション』を参照してください。

この例の各 XML タグの意味は以下のとおりです。

<quanta ...>
ハートビート間隔内のクォンタ休止時間の長さの要約。休止時間の長さがミリ秒で示されます。

<free-mem type="heap" ...>
ハートビート間隔でのフリー・ヒープ・スペース量 (各ガーベッジ・コレクション・クォンタの終了時にサンプリングされます) の要約。

<classunload-info classloadersunloaded=9 classesunloaded=156 />
ハートビート間隔においてアンロードされたクラスおよびクラス・ローダーの数。

<references type="weak" cleared="660 />
ハートビート間隔内にクリアされた Java 参照オブジェクトの数とタイプ。

注:

- 2 つのハートビートの間にガーベッジ・コレクション・クォンタが 1 回しか生じなかった場合、空きメモリーのサンプリングは、この 1 回のクォンタの終了時にのみ行われます。このため、ハートビートの要約で示される最小量、最大量、および平均量はすべて同じになります。
- ガーベッジ・コレクション・アクティビティーが必要になるほどヒープが埋まっていない場合、2 つのハートビート・イベントの間隔は、指定されたサイクル・タイムよりも大幅に長くなる可能性があります。例えば、ご使用のプログラムでガーベッジ・コレクション・アクティビティーを数秒ごとに一度だけ行う必要がある場合、ハートビートは数秒ごとに一度だけになると考えられます。
- ガーベッジ・コレクション・アクティビティーに必要なヒープが十分でない場合、2 つのハートビート・イベントの間隔が指定されたサイクル・タイムより大幅に長くなる可能性があります。例えば、ご使用のプログラムでガーベッジ・コ

レクション・アクティビティーを数秒ごとに一度だけ行う必要がある場合、ハートビートは数秒ごとに一度だけになると考えられます。

同期ガーベッジ・コレクションや優先順位変更などのイベントが発生した場合、そのイベントおよび保留中の (ハートビートなどの) すべてのイベントの詳細は、直ちに出力として生成されます。

- 一定期間の最大ガーベッジ・コレクション・クォンタが大きすぎる場合は、**-Xgc:targetUtilization** オプションを使用してターゲットの使用率を減らした方がよい場合があります。このアクションは、ガーベッジ・コレクターの処理時間を増やします。あるいは、**-Xmx** オプションでヒープ・サイズを増やす方法もあります。同様に、ご使用のアプリケーションで、現在報告されている時間よりも長い遅延時間が許容されている場合は、ターゲットの使用率を増やすか、ヒープ・サイズを減らすことができます。
- **-Xverbosegclog:<file>** オプションで、出力をコンソールではなく、ログ・ファイルにリダイレクトできます。例えば、**-Xverbosegclog:out** を使用した場合は、**-verbose:gc** の出力が *out* ファイルに書き込まれます。
- thread-priority にリストされている優先順位は、Java スレッド優先の順位ではなく、基盤となっているオペレーティング・システムのスレッド優先順位です。

同期ガーベッジ・コレクション

同期 (非決定論的な) ガーベッジ・コレクションが発生した場合、項目は **-verbose:gc** ログにも書き込まれます。このイベントの原因として、以下の 3 つが考えられます。

- コードでの明示的な System.gc() 呼び出し。
- JVM でメモリー不足が発生し、OutOfMemoryError 状態を回避するために、同期ガーベッジ・コレクションが実行された。
- JVM が継続的なガーベッジ・コレクション中にシャットダウンした。JVM はガーベッジ・コレクションを取り消すことができないため、ガーベッジ・コレクションを同期的に完了し、終了します。

System.gc() 項目は以下の例のようになります。

```
<gc-op id="9" type="syncgc" timems="12.92" contextid="8" timestamp="2011-07-12T09:41:40.808">
  <syncgc-info reason="system GC" totalBytesRequested="260" exclusiveaccessTimeMs="0.009"
    threadPriority="11" />
  <free-mem-delta type="heap" bytesBefore="22085440" bytesAfter="136023450" />
  <free-mem-delta type="immortal" bytesBefore="62324800" bytesAfter="62324800"/>
  <classunload-info classloadersunloaded="54" classesunloaded="234" />
  <references type="soft" cleared="21" dynamicThreshold="29" maxThreshold="32" />
  <references type="weak" cleared="523" />
  <finalization enqueued="124" />
</gc-op>
```

JVM のシャットダウンの結果として行われた同期ガーベッジ・コレクションの項目は、以下の例のようになります。

```
<gc-op id="24" type="syncgc" timems="6.439" contextid="19" timestamp="2011-07-12T09:43:14.524">
  <syncgc-info reason="VM shut down" exclusiveaccessTimeMs="0.009" threadPriority="11" />
  <free-mem-delta type="heap" bytesBefore="56182430" bytesAfter="151356238" />
  <free-mem-delta type="immortal" bytesBefore="23659200" bytesAfter="23659200"/>
  <classunload-info classloadersunloaded="14" classesunloaded="276" />
  <references type="soft" cleared="154" dynamicThreshold="29" maxThreshold="32" />
  <references type="weak" cleared="53" />   <finalization enqueued="34" />
</gc-op>
```

この例の XML タグおよび属性の意味は以下のとおりです。

<gc-op id="9" type="syncgc" timems="6.439" ...
: この行は、イベント・タイプが同期ガーベッジ・コレクションであることを示しています。 timems 属性は、同期ガーベッジ・コレクションの継続時間です (ミリ秒単位)。

<syncgc-info reason="..."/>
: 同期ガーベッジ・コレクションの原因を示しています。

<free-mem-delta.../>
: 同期ガーベッジ・コレクションの実行前と実行後のフリー Java ヒープ・メモリー (バイト単位)。

<finalization .../>
: ファイナライズを待っているオブジェクトの数。

<classunload-info .../>
: ハートビート間隔においてアンロードされたクラスおよびクラス・ローダーの数。

<references type="weak" cleared="53" .../>
: ハートビート間隔内にクリアされた Java 参照オブジェクトの数とタイプ。

メモリー不足状態または VM のシャットダウンが原因の同期ガーベッジ・コレクションは、ガーベッジ・コレクターがアクティブである場合にのみ発生する可能性があります。即時というわけではありませんが、trigger start イベントが先行する必要があります。場合によっては、複数の heartbeat イベントが、trigger start イベントと synchgc イベントの間に発生する可能性があります。 System.gc() が原因の同期ガーベッジ・コレクションはどの時点でも発生する可能性があります。

すべての GC クォンタのトラッキング

個々の GC クォンタのトラッキングは、GlobalGCStart および GlobalGCEnd のトレース・ポイントを有効にすることで可能になります。これらのトレース・ポイントは、同期ガーベッジ・コレクションを含む、すべての Metronome ガーベッジ・コレクター・アクティビティーの開始時と終了時に生成されます。これらのトレース・ポイントの出力は以下のようになります。

```
03:44:35.281 0x833cd00 j9mm.52 - GlobalGC start: globalcount=3

03:44:35.284 0x833cd00 j9mm.91 - GlobalGC end: workstackoverflow=0 overflowcount=0
```

優先順位の変更

(アプリケーションによりスレッド優先順位が変更されたか、またはアプリケーションの 1 つ以上のスレッドが終了したため) ガーベッジ・コレクターのスレッド優先順位が変更されると、**-verbose:gc** ログには、要約のほかに項目も書き込まれます。リストされている優先順位は、Java のスレッド優先順位ではなく、基礎となる OS のスレッド優先順位です。ガーベッジ・コレクターのスレッド優先順位の変更項目は、以下の例のようになります。

```
<gc type="heartbeat" id="73" timestamp="Feb 26 13:11:35 2007" intervalms"1001.754">
  <summary quantumcount="240">
    <quantum minms="0.022" meanms="0.984" maxms="1.011" />
    <classunloading classloaders="11" classes="17" />
```

```
|       <heap minfree="202833920" meanfree="214184823" maxfree="221102080" />
|       <thread-priority maxPriority="11" minPriority="11" />
|     </summary>
|   </gc>
```

優先順位の変更は、ガーベッジ・コレクターのスレッド優先順位に関するトレース・ポイント情報を生成することにより、リアルタイムでトラッキングできます。この出力は以下のようになります。

```
15:58:25.493*0x8286e00    j9mm.102       - setGCThreadPriority() called with
newGCThreadPriority = 11
```

この出力は、**-Xtrace:iprint=tpnid{j9mm.102}** のように、ID を使用して有効にすることができます。

メモリー不足に関する項目

メモリー域のいずれかでフリー・スペース不足が発生すると、OutOfMemoryError 例外がスローされる前に、**-verbose:gc** ログに項目が書き込まれます。この出力は以下の例のようになります。

```
| <out-of-memory id="71" timestamp="2011-07-23T08:32:51.435" memorySpaceName="Scoped"
|   memorySpaceAddress="080EED9C"/>
```

デフォルトでは、OutOfMemoryError 例外の結果として、Javadump が生成されます。このダンプには、プログラムによって使用されたメモリー域に関する情報が含まれます。以下のように、**-verbose:gc** 出力で指定された J9MemorySpace 値とともに、ダンプ内のこの情報を使用すれば、スペース不足が発生した特定のメモリー域を識別することができます。

```
| NULL           id         start      end        size       space/region
| 1STHEAPSPACE   0x080EED9C     --         --         --     Scoped
| 1STHEAPREGION  0x0810C570 0xF1B09028 0xF2B09028 0x01000000 Scoped/Region
| NULL
| 1STHEAPTOTAL   Total memory:           16777216 (0x01000000)
| 1STHEAPINUSE   Total memory in use:      625952 (0x00098D20)
| 1STHEAPFREE    Total memory free:      16151264 (0x00F672E0)
```

| 上記の例では、-verbose:gc 出力で得られたメモリー・スペース ID (0x080EED9C)
| を、Java ダンプの Scoped メモリー域の ID と突き合わせることができます。この
| 突き合わせは、複数のスコープを扱っており、メモリー不足が発生しているスコー
| プを特定する必要がある場合に役立ちます。-verbose:gc 出力では、永久メモリー、
| スコープ・メモリー、またはヒープ・メモリーで OutOfMemoryError が発生したか
| どうかしか示されないからです。

メモリー不足状態での Metronome ガーベッジ・コレクターの動作:

JVM でメモリー不足が発生した場合、デフォルトでは、Metronome ガーベッジ・コレクターは無制限の非決定論的なガーベッジ・コレクションを起動します。非決定論的な動作を回避するには、**-Xgc:noSynchronousGCOnOOM** オプションを使用して、JVM でメモリー不足が発生したときに OutOfMemoryError をスローするようにします。

デフォルトの無制限コレクションは、考えられるすべてのガーベッジが 1 回の操作で収集されるまで実行されます。必要な休止時間は、通常、Metronome の標準的な増分クォンタよりも長いミリ秒になります。

関連情報

-Xverbose:gc を使用した同期ガーベッジ・コレクションの分析

明示的な System.gc() 呼び出しでの Metronome ガーベッジ・コレクターの動作:

ガーベッジ・コレクション・サイクルが進行中の場合は、`System.gc()` が呼び出されると、Metronome ガーベッジ・コレクターがサイクルを完了させるまでこのメソッドは待機します。進行中のガーベッジ・コレクション・サイクルがない場合は、`System.gc()` が呼び出されたときに完全なサイクルが実行され、このメソッドはサイクルが完了するまで待機します。制御された方法でヒープをクリーンアップするには、`System.gc()` を使用します。これは、ガーベッジ・コレクションが完全に実行されてからコントロールが戻されるため、非決定論的な操作になります。

一部のアプリケーションは、これらの非決定論的な遅延の発生が受け入れられない System.gc() を呼び出すベンダー・ソフトウェアを呼び出します。すべての System.gc() 呼び出しを無効にするには、**-Xdisableexplicitgc** オプションを使用します。

以下のように、System.gc() 呼び出しに対する詳細なガーベッジ・コレクション出力には、『system garbage collect』 という理由が含まれており、duration の時間が長くなる可能性があります。

```
<gc-op id="9" type="syncgc" timems="6.439" contextid="8" timestamp="2011-07-12T09:41:40.808">
  <yncgc-info reason="VM shut down" exclusiveaccessTimeMs="0.009"
threadPriority="11"/>
  <free-mem-delta type="heap" bytesBefore="126082300" bytesAfter="156085440"/>
  <free-mem-delta type="immortal" bytesBefore="5129096" bytesAfter="5129096"/>
  <classunload-info classloadersunloaded="14" classesunloaded="276"/>
  <references type="soft" cleared="154" dynamicThreshold="29" maxThreshold="32"/>
  <references type="weak" cleared="53"/>
  <finalization enqueued="34"/>
</gc-op>
```

共有クラス診断

発生した問題をどのように診断するのかを理解しておくと、共有クラス・モードを使用するときに役立ちます。

共有クラスの概要については、JVM 間でのクラス・データの共有を参照してください。

IBM SDK for Java 7 ユーザー・ガイドには、共有クラスに関する問題の診断についての次のような有用なガイダンスが記載されています。

- 共有クラスの配備
- 実行時バイトコード変更の取り扱い
- 動的更新について
- Java ヘルパー API の使用
- 共有クラスの診断出力について
- 共有クラスに関する問題のデバッグ

この情報は、IBM SDK for Java 7 - 共有クラス診断で参照できます。

IBM SDK for Java 7 ユーザー・ガイドの内容の一部は、IBM WebSphere Real Time for RT Linux には適用できないものもあります。具体的には、以下のとおりです。

- リアルタイム・モードでは、アプリケーションには共有クラス・キャッシュに対して読み取り権限のみがあり、読み取り/書き込み権限はありません。
- キャッシュは、**admincache** ツールのみを使用して変更することができます。
- 非永続キャッシュは、リアルタイム・モードでは使用できません。

JVMTI の使用

JVMTI は、JVM とネイティブ・エージェント間の通信を可能にする、双方向インターフェースです。これにより、JVMDI および JVMPI のインターフェースが置き換えられます。

JVMTI により、JVM 用のデバッグ、プロファイル作成、およびモニター・ツールの作成をサード・パーティーが行うことができます。 必要とされる種類の情報についてエージェントが JVM に通知するメカニズムが、インターフェースに搭載されています。このインターフェースでは、関連した通知を受け取る手段も提供されています。複数のエージェントを、いつでも JVM に接続することができます。

IBM SDK for Java 7 ユーザー・ガイドには、JVMTI に対する IBM 拡張に関する API リファレンスのセクションを含む、JVMTI の使用に関する詳細情報が記載されています。

この情報は、IBM SDK for Java 7 - JVMTI の使用で参照できます。

Diagnostic Tool Framework for Java の使用

Diagnostic Tool Framework for Java (DTFJ) は、IBM が提供する Java アプリケーション・プログラミング・インターフェース (API) であり、Java 診断ツールの作成をサポートするために使用されます。DTFJ は、システム・ダンプまたは Javadump のデータを扱います。

IBM SDK for Java 7 ユーザー・ガイドには、DTFJ に関する詳細情報が記載されています。Diagnostic Tool Framework for Java の使用を参照してください。

IBM Monitoring and Diagnostic Tools for Java - Health Center の使用

IBM Monitoring and Diagnostic Tools for Java - Health Center は、実行中の Java 仮想マシン (JVM) の状況を監視するための診断ツールです。

IBM Monitoring and Diagnostic Tools for Java - Health Center に関する情報は、developerWorks® および インフォメーション・センター から入手できます。

第 10 章 参照

この一連のトピックでは、WebSphere Real Time for RT Linux で使用可能なオプションおよびクラス・ライブラリーのリストを示します。

コマンド行オプション

Java の開始時にコマンド行でオプションを指定できます。デフォルト・オプションは、最も一般的な使用法に応じて選択されています。

Java オプションとシステム・プロパティーの指定

Java プロパティーおよびシステム・プロパティーは、3 とおりの方法で指定できます。

このタスクについて

Java オプションおよびシステム・プロパティーは、以下の方法で指定できます。それらは、優先順に以下のものです。

1. コマンド行でオプションまたはプロパティーを指定します。例えば、次のようにします。

   ```
   java -Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump MyJavaClass
   ```

2. 該当のオプションを含むファイルを作成し、**-Xoptionsfile=**<*filename*> オプションを使用してこのファイルをコマンド行で指定します。

   オプション・ファイルでは、各オプションを改行して指定します。単一オプションを複数行にわたって記述する場合は、継続文字として「¥」文字を使用できます。コメント行を定義するには、「#」文字を使用します。オプション・ファイルで **-classpath** を指定することはできません。オプション・ファイルの例を以下に示します。

   ```
   #My options file
   -X<option1>
   -X<option2>=¥
   <value1>,¥
   <value2>
   -D<sysprop1>=<value1>
   ```

3. オプションを含む **IBM_JAVA_OPTIONS** という環境変数を作成します。例えば、次のようにします。

   ```
   export IBM_JAVA_OPTIONS="-Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump"
   ```

コマンド行では、最後に指定したオプションが最初のオプションより優位になります。例えば、「**-Xint -Xjit myClass**」というオプションを指定した場合、**-Xjit** というオプションは **-Xint** より優位になります。

システム・プロパティー

アプリケーションではシステム・プロパティーを使用することができ、これにより、ランタイム環境の情報を提供できます。

com.ibm.jvm.realtime
このプロパティーにより、Java アプリケーションは、WebSphere Real Time for RT Linux 環境内で実行中であるかどうかを判別できます。

アプリケーションが IBM WebSphere Real Time for RT Linux ランタイム内で実行中であり、**-Xrealtime** オプションを指定して開始されている場合は、**com.ibm.jvm.realtime** プロパティーの値は「hard」になります。

アプリケーションが IBM WebSphere Real Time for RT Linux ランタイム内で実行中で、**-Xrealtime** オプションを指定して開始されていない場合は、**com.ibm.jvm.realtime** プロパティーは設定されません。

アプリケーションが IBM WebSphere Real Time ランタイム内で実行中の場合、 **com.ibm.jvm.realtime** プロパティーの値は「soft」になります。

標準オプション

標準オプションの定義

-agentlib:<*libname*>**[=**<*options*>**]**
ネイティブ・エージェント・ライブラリー <*libname*> をロードします。例えば、**-agentlib:hprof** のように指定します。詳しくは、コマンド行で **-agentlib:jdwp=help** と **-agentlib:hprof=help** を指定してください。

-agentpath:*libname***[=**<*options*>**]**
絶対パス名でネイティブ・エージェント・ライブラリーをロードします。

-assert assert 関連オプションのヘルプを表示します。

-cp または -classpath <: *で区切られたディレクトリーおよび .zip ファイルまたは .jar ファイル*>
アプリケーション・クラスおよびリソースの検索パスを設定します。**-classpath** も **-cp** も使用しないで、**CLASSPATH** を設定しない場合、ユーザー・クラスパスは、デフォルトでは現行ディレクトリー (.) になります。

-D<property_name>=<*value*>
システム・プロパティーを設定します。

-help または -?
使用法メッセージを表示します。

-javaagent:<*jarpath*>**[=**<*options*>**]**
Java プログラミング言語エージェントをロードします。詳しくは、java.lang.instrument API の資料を参照してください。

-jre-restrict-search
バージョン検索の対象にユーザーのプライベート JRE を含めます。

-no-jre-restrict-search
バージョン検索の対象からユーザーのプライベート JRE を除外します。

-showversion
製品のバージョンを表示して継続します。

-verbose:[*class***,***gc***,***dynload***,***sizes***,***stack***,***jni***]**
詳細出力を使用可能にします。

-verbose:class
: ロードされるクラスごとに、項目を stderr に書き込みます。

-verbose:gc
: 145 ページの『verbose:gc 情報の使用』を参照してください。

-verbose:dynload
: 各クラスが JVM によってロードされるときに、以下のような詳細情報を提供します。
- クラス名およびパッケージ
- `.jar` ファイル内にあったクラス・ファイルの場合、`.jar` の名前およびディレクトリー・パス
- クラスのサイズ、およびクラスのロードにかかった時間などの詳細

データは stderr に書き込まれます。出力の例を以下に示します。

```
<Loaded java/lang/String from /myjdk/sdk/jre/lib/i386/
softrealtime/jclSC160/vm.jar>
<Class size 17258; ROM size 21080; debug size 0>
<Read time 27368 usec; Load time 782 usec; Translate time 927 usec>
```

注: 共有クラス・キャッシュからロードされたクラスは **`-verbose:dynload`** 出力に表示されません。 これらのクラスの情報を出力するには、**`-verbose:class`** を使用してください。

-verbose:sizes
: JVM 内のスタックおよびヒープのために使用されたメモリーの量を示す情報を stderr に書き込みます。

-verbose:stack
: Java および C のスタック使用量について説明する情報を stderr に書き込みます。

-verbose:jni
: アプリケーションと JVM によって呼び出される JNI サービスについて説明する情報を stderr に書き込みます。

-version
: 非リアルタイム・モードのバージョン情報を表示します。 -Xrealtime オプションとともに使用された場合、リアルタイム・モードのバージョン情報を表示します。

-version:<*value*>
: 指定したバージョンの実行を要求します。

-X 非標準オプションのヘルプを表示します。

非標準オプション

接頭部に **`-X`** が付いたオプションは非標準であり、予告なしに変更されることがあります。

IBM SDK for Java 7 ユーザー・ガイドに、非標準オプションに関する詳細情報が記載されています。この情報は、IBM SDK for Java 7 - コマンド行オプションで参照できます。

IBM WebSphere Real Time for RT Linux の補足情報は、以下のセクションに記載されています。

リアルタイム・オプション

WebSphere Real Time for RT Linux で使用される **-Xrealtime** オプションの定義。

以下の **-X** オプションは、WebSphere Real Time for RT Linux 環境で使用することができます。

-Xrealtime
: リアルタイム・モードを開始します。Metronome ガーベッジ・コレクターを実行して Real-Time Specification for Java (RTSJ) サービスを使用する場合に指定する必要があります。このオプションを指定しない場合は、JVM は IBM SDK and Runtime Environment for Linux Platforms, Java 2 Technology バージョン 7 に相当する非リアルタイム・モードで開始されます。

  -Xrealtime オプションは **-Xgcpolicy:metronome** と交換可能です。リアルタイム・モードを使用するために、どちらを指定することもできます。

Ahead-of-Time オプション

Ahead-of-Time オプションの定義

目的

オプションが指定されない場合:
: インタープリター、および動的にコンパイルされたコードで実行されます。AOT コードは検出されても使用されません。その代わりに、そのコードは必要に応じて動的にコンパイルされます。これは、非リアルタイム・アプリケーションといくつかのリアルタイム・アプリケーションで特に役に立ちます。このオプションにより、パフォーマンスとスループットが最適化されますが、コンパイルが行われると、非決定論的遅延が実行時に生じることがあります。

-Xjit: このオプションはデフォルトと同じです。

-Xint: インタープリターのみを実行し、プリコンパイルされた jar ファイル内で検出される可能性のある、AOT 用に書かれたコードを無視して、動的コンパイラーを実行しません。このモードは、コンパイルに関連すると思われるデバッグの問題が生じた場合、またはコンパイルによる効果が得られない、きわめて短いバッチ・アプリケーションの場合を除き、あまり必要になることはありません。

-Xnojit:
: インタープリターを実行し、プリコンパイルされた jar ファイル内で AOT 用に書かれたコードが検出された場合にはそれを使用します。動的コンパイラーは実行しません。このモードは、一部のリアルタイム・アプリケーションで、コンパイルに起因する非決定論的遅延が実行時に発生しないようにしたい場合に役立ちます。 AOT 用に書かれたコードは、**-Xrealtime** オプションを指定して実行されているときにのみ使用できます。標準 JVM で実行されているとき、つまり、**-Xrealtime** が指定されていないときにはサポートされません。

例　`java -Xrealtime -Xnojit` *outputtest.jar*。

Metronome ガーベッジ・コレクターのオプション

Metronome ガーベッジ・コレクター・オプションの定義。

-Xgc:immortalMemorySize=*size*
: 永久ヒープ領域のサイズを指定します。デフォルトは 16 MB です。

-Xgc:scopedMemoryMaximumSize=*size*
: スコープ・メモリー・ヒープ領域のサイズを指定します。デフォルトは 8 MB です。

-Xgc:synchronousGCOnOOM | -Xgc:nosynchronousGCOnOOM
: ガーベッジ・コレクションが行われる原因の 1 つとして、ヒープでのメモリー不足があります。ヒープに空き領域がなくなった場合、**`-Xgc:synchronousGCOnOOM`** を使用すると、アプリケーションが停止し、その間にガーベッジ・コレクションによって未使用オブジェクトが削除されます。それでも空き領域が足りなくなる場合は、目標使用率を低くして、より長い時間をかけてガーベッジ・コレクションを完了させることを考慮してください。 **`-Xgc:nosynchronousGCOnOOM`** を設定すると、ヒープ・メモリーがいっぱいになったときにアプリケーションが停止してメモリー不足メッセージを出すようになります。デフォルトは **`-Xgc:synchronousGCOnOOM`** です。

-Xnoclassgc
: クラス・ガーベッジ・コレクションを使用不可にします。このオプションは、JVM で使用されていない Java クラスに関連するストレージのガーベッジ・コレクションをオフに切り替えます。デフォルトの動作は **`-Xnoclassgc`** です。

-Xgc:targetUtilization=*N*
: アプリケーション使用率を N% に設定すると、ガーベッジ・コレクターは最大で各時間間隔の (100-N)% まで使用しようと試みます。妥当な値は 50% から 80% までの間です。割り振り速度の低いアプリケーションは、90% で実行できる可能性があります。デフォルトは 70% です。

  次の例では、ヒープ・メモリーの最大サイズが 30 MB に設定されています。アプリケーションの目標使用率が 75% に設定されているため、ガーベッジ・コレクターは最大で各時間間隔の 25% まで使用するように試みます。

```
java -Xrealtime -Xmx30m -Xgc:targetUtilization=75 Test
```

-Xgc:threads=*N*
: 実行する GC スレッドの数を指定します。デフォルトは 1 です。

-Xgc:verboseGCCycleTime=N
: N は、要約情報をダンプする時間 (ミリ秒単位) です。

  注: 要約情報は、サイクル・タイムに指定された時間ちょうどにダンプされるのではなく、この時間の基準を満たす最後のガーベッジ・コレクション・イベント時にダンプされます。

-Xmx<*size*>
: Java ヒープ・サイズを指定します。他のガーベッジ・コレクション戦略と

は異なり、リアルタイム Metronome GC はヒープ拡張をサポートしません。初期または最大ヒープ・サイズを指定するオプションはありません。最大ヒープ・サイズのみを指定できます。

-Xthr:metronomeAlarm=os*xx*
Metronome ガーベッジ・コレクター アラーム・スレッドの実行優先順位を制御します。

ここで、*xx* は、Metronome アラーム・スレッドの実行優先順位を指定する、11 から 89 までの数値です。アラーム・スレッドが実行される OS 優先順位を変更する際には注意が必要です。リアルタイム・スレッドより低い OS 優先順位を指定すると、ガーベッジを割り振るリアルタイム・スレッドより低い優先順位でガーベッジ・コレクターが実行されることになるため、OutOfMemory エラーが発生します。デフォルトの Metronome ガーベッジ・コレクター アラーム・スレッドは、OS 優先順位である 89 で実行されます。

JVM のデフォルト設定

Real Time JVM の実行環境に変更が行われなかった場合、この JVM にはデフォルト設定が適用されます。一般的な設定をリファレンスとして示します。

デフォルト設定は、環境変数を使用するか、JVM の開始時にコマンド行パラメーターを使用することで変更できます。一般的な JVM 設定の一部を以下の表に示します。最後の列は動作の変更方法を示します。この列には以下のキーが適用されます。

- **e** - 環境変数のみが設定を制御します。
- **c** - コマンド行パラメーターのみが設定を制御します。
- **ec** - 環境変数とコマンド行パラメーターの両方が設定を制御しますが、コマンド行パラメーターが優先します。

この情報はクイック・リファレンスとして提供されるもので、包括的なものではありません。

JVM の設定	デフォルト	影響する設定
Javadump	使用可能	ec
メモリー不足時の Javadump	使用可能	ec
Heapdump	使用不可	ec
メモリー不足時の Heapdump	使用可能	ec
Sysdump	使用可能	ec
ダンプ・ファイルの生成場所	現行ディレクトリー	ec
verbose 出力	使用不可	c
ブート・クラスパス検索	使用不可	c
JNI チェック	使用不可	c
リモート・デバッグ	使用不可	c
厳密な適合チェック	使用不可	c
クイック・スタート	使用不可	c

JVM の設定	デフォルト	影響する設定
リモート・デバッグ情報サーバー	使用不可	c
シグナル通知の削減	使用不可	c
シグナル・ハンドラーのチェーニング	使用可能	c
クラスパス	未設定	ec
クラス・データの共有	使用不可	c
アクセシビリティー・サポート	使用可能	e
JIT コンパイラー	使用可能	ec
AOT コンパイラー (AOT は、共有クラスも使用可能でない限り、JVM では使用されません)	使用可能	c
JIT デバッグ・オプション	使用不可	c
アルゴリズムで太字に指定したフォントの Java2D での最大サイズ	14 ポイント	e
Java2D でスケーラブル・フォントのレンダー・ビットマップを使用	使用可能	e
Java2D でのフリータイプ・フォントのラスタライズ	使用可能	e
Java2D で AWT フォントを使用	使用不可	e
デフォルト・ロケール	なし	e
プラグイン開始までの待ち時間	ゼロ	e
一時ディレクトリー	/tmp	e
プラグインのリダイレクト	なし	e
IM 切り替え	使用不可	e
IM 修飾子	使用不可	e
スレッド・モデル	該当なし	e
Java スレッド 32 ビット用の初期スタック・サイズ。 使用法: **-Xiss<サイズ>**	2 KB	c
Java スレッド 32 ビット用の最大スタック・サイズ。 使用法: **-Xss<サイズ>**	256 KB	c
OS スレッド 32 ビット用のスタック・サイズ。 使用法: **-Xmso<サイズ>**	256 KB	c
初期ヒープ・サイズ。 使用法: **-Xms<サイズ>**	64 MB	c
最大 Java ヒープ・サイズ。 使用法: **-Xmx<サイズ>**	使用可能メモリーの半分のサイズ (最小 16 MB、最大 512 MB)	c
アプリケーションのターゲット時間間隔の利用。ガーベッジ・コレクターはその余剰時間を使用しようとします。使用法: **-Xgc:targetUtilization=<percentage>**	70%	c
実行するガーベッジ・コレクター・スレッドの数。使用法: **-Xgc:threads=<value>**	1	c
-Xrealtime モードでスコープ・メモリーに割り振り可能なメモリーの最大量。使用法: **-Xgc:scopedMemoryMaximumSize=<size>**	8 MB	c

JVM の設定	デフォルト	影響する設定
-Xrealtime モードでの永久メモリー域のサイズの設定。使用法: **-Xgc:immortalMemorySize=<size>**	16 MB	c

注: 「使用可能メモリー」とは、実際の (物理) メモリーまたは **RLIMIT_AS** 値のいずれかの最小値です。

WebSphere Real Time for RT Linux クラス・ライブラリー

WebSphere Real Time for RT Linux によって使用される Java クラス・ライブラリーの参照資料です。

WebSphere Real Time for RT Linux によって使用される Java クラス・ライブラリーについては、http://www.rtsj.org/specjavadoc/book_index.htmlで説明されています。

TCK を使用した実行

WebSphere Real Time for RT Linux とともに Real-Time Specification for Java (RTSJ) Technology Compatibility Kit (TCK) を実行している場合、テストを正常に完了させるために demo/realtime/TCKibm.jar をクラスパスに含める必要があります。

TCKibm.jar には、**VibmcorProcessorLock** クラスが含まれています。このクラスは、TCK.ProcessorLock クラスに対して IBM による拡張が加えられたものです。このクラスでは、小規模な TCK テスト・セットで必要なユニプロセッサーの動作が提供されます。 TCK.ProcessorLock クラスおよびこのクラスに対するベンダー固有の拡張について詳しくは、TCK ディストリビューションに含まれている README ファイルを参照してください。

特記事項

本書は米国 IBM が提供する製品およびサービスについて作成したものです。本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒242-8502
神奈川県大和市下鶴間1623番14号
日本アイ・ビー・エム株式会社
法務・知的財産
知的財産権ライセンス渉外

以下の保証は、国または地域の法律に沿わない場合は、適用されません。

IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。 国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。 IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

© Copyright IBM Corp. 2003, 2011

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム（本プログラムを含む）との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

- JIMMAIL@uk.ibm.com (Hursley Java Technology Center (JTC) 連絡先)

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性がありますが、その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確証できません。 IBM 以外の製品の性能に関する質問は、それらの製品の供給者にお願いします。

商標

IBM、IBM ロゴおよび ibm.com は、世界の多くの国で登録された International Business Machines Corp. の商標です。他の製品名およびサービス名等は、それぞれ IBM または各社の商標である場合があります。現時点での IBM の商標リストについては、http://www.ibm.com/legal/copytrade.shtml をご覧ください。

Adobe、Adobe ロゴ、PostScript、PostScript ロゴは、Adobe Systems Incorporated の米国およびその他の国における登録商標または商標です。

Intel および Itanium は、Intel Corporation の米国およびその他の国における商標です。

Linux は、Linus Torvalds の米国およびその他の国における商標です。

Java およびすべての Java 関連の商標およびロゴは Oracle やその関連会社の米国およびその他の国における商標または登録商標です。

索引

日本語, 数字, 英字, 特殊文字の順に配列されています。なお, 濁音と半濁音は清音と同等に扱われています。

［ア行］

アプリケーション
　実行　91, 92
アプリケーションの実行　91, 92
アラーム・スレッド
　Metronome ガーベッジ・コレクター　5
アンインストール　38
　InstallAnywhere　38
安全なクラス
　NHRT　69
イベント
　ダンプ・エージェント　123
インストール　29
永久メモリー　5, 14
オブジェクト・レコード、Heapdump 内の　133
オプション
　-noRecurse　55
　-outPath　55
　-searchPath　55
　-verbose:gc　145
　-Xdump:heap　132
　-Xgc:immortalMemorySize　159
　-Xgc:noSynchronousGCOnOOM　151
　-Xgc:nosynchronousGCOnOOM　159
　-Xgc:scopedMemoryMaximumSize　159
　-Xgc:synchronousGCOnOOM　151, 159
　-Xgc:targetUtilization　159
　-Xgc:threads　159
　-Xgc:verboseGCCycleTime=N　145, 159
　-Xmx　159
　-Xnojit　43
　-Xrealtime　43
オペレーティング・システム　25

［カ行］

ガーベッジ・コレクション
　リアルタイム　5, 73
　Metronome　5, 73
ガーベッジ・コレクターの診断　145
　使用、診断ツールの　145
概念　5
開発、アプリケーションの　75
概要　1
型シグニチャー　135
既知の制限　109
共有クラス
　診断　152
共有クラス・キャッシュ　44, 45, 48, 49, 51, 52, 53, 54, 70, 71
クラスのアンロード
　Metronome　5
クラス・データの共有　101
クラス・レコード、Heapdump 内の　134
クラス・ロード
　NHRT　63
クラッシュ
　Linux　108
クロック
　リアルタイム　86
計画　25
計画、非同期イベント・ハンドラー　19, 82
コア・ファイル　105
コレクション・スレッド
　Metronome ガーベッジ・コレクター　5
コンパイラー
　Ahead-of-Time　8, 43
コンパイル　7, 40
コンパイルの障害、JIT の　142

［サ行］

作業ベースのコレクション　5
作成　55, 56, 58
作成、非同期イベント・ハンドラー　19, 82
作成、プリコンパイルされたファイルの　55, 56, 58
参照　155
サンプル・アプリケーション　87, 95
時間ベースのコレクション
　Metronome　5
シグナル・ハンドリング　19
システム・プロパティー　63
実行、アプリケーションの　39
使用、ダンプ・エージェントの　122
障害が発生したメソッド、JIT の　140
障害が発生したメソッドの特定、JIT の　140
診断ツールの使用　122
　Diagnostics Collector　145
診断ツールの使用 *(続き)*
　DTFJ　153
スケジューリング・ポリシー
　SCHED_FIFO　10, 12, 39, 40
　SCHED_OTHER　10, 12, 39, 40
　SCHED_RR　10, 39, 40
スコープ・メモリー　5, 14
ストレージ管理、Javadump の　127
スレッドおよびスタック・トレース (THREADS)　129
スレッド・スケジューリング　10, 39
スレッド・ディスパッチング　10, 39
制御、プロセッサー使用率の　73
制限
　Metronome　74
セキュリティー　103
セキュリティー・マネージャー　63
設定、デフォルト (JVM)　160
ソフトウェア前提条件　25

［タ行］

短期実行アプリケーション
　JIT　144
ダンプ・エージェント
　イベント　123
　使用　122
　フィルター　124
ダンプ・ビューアー　135
　診断ツールの使用　135
直列化　63
テキスト (標準型) の Heapdump ファイル・フォーマット
　Heapdump　133
デフォルト設定、JVM　160
同期　18
トラブルシューティング
　Metronome　145
トラブルシューティングおよびサポート　105
トレース　137
　診断ツールの使用　137
トレーラー・レコード 1、Heapdump 内の　135
トレーラー・レコード 2、Heapdump 内の　135

［ナ行］

内部ベース優先順位　12

© Copyright IBM Corp. 2003, 2011

［ハ行］

ハードウェア前提条件　25
パッケージ化　29
パフォーマンス上の問題のデバッグ　108
ヒープ・メモリー　14
非直列化　63
非同期イベント・ハンドラー
　計画　19, 82
　作成　19, 82
非ヒープ・リアルタイム
　使用　61
非ヒープ・リアルタイム・スレッド　16
標準型 (テキスト) の Heapdump ファイル・フォーマット
　Heapdump　133
複数の Heapdump　132
プリコンパイルされたファイル　55, 56, 58
ヘッダー・レコード、Heapdump 内の　133
ポリシー　10, 40

［マ行］

メモリー
　所要量　16
　SizeEstimator クラス　16
メモリー域　14
　リフレクション　121
メモリー管理　14
メモリー管理の理解　114
メモリー・リーク
　回避　119
戻りコード　55
問題判別　105

［ヤ行］

ユーザー・ベース優先順位:　12
優先順位　10, 40
　内部ベース　12
　ユーザー・ベース　12
優先順位逆転　18
優先順位継承　18
優先順位スケジューラー　10, 39
優先順位の継承　13

［ラ行］

リアルタイム・ガーベッジ・コレクション　5, 73
リアルタイム・クロック　86
リアルタイム・スレッド　16
　計画　79
リアルタイム・スレッド *(続き)*
　作成　79
リアルタイム・スレッドの計画　79
リアルタイム・スレッドの作成　79
リソース共有　18
リフレクション
　メモリー・コンテキスト　121

A

admincache
　管理　48, 54, 70
　共有クラス・キャッシュ　44, 48, 49, 51, 52, 53, 54, 70, 71
　検査、クラス・キャッシュの　49
　サイズ変更、共有クラス・キャッシュの　52
　削除、キャッシュの　51
　作成、リアルタイム共有クラス・キャッシュの　45
　使用　44, 45, 71
　選択、キャッシュに入れるクラスの　53
　破棄、キャッシュの　51
　リスト、クラス・キャッシュの　48
Ahead-of-Time コンパイル　8, 43
AOT
　無効化　138
AOT (Ahead-Of-Time) コンパイラー　93
AOT コンパイラーの無効化　138

C

CLASSPATH
　設定　36

D

Diagnostics Collector　145
DTFJ　153

H

Health Center　153
　診断ツールの使用　153
Heapdump　132
　診断ツールの使用　132
　テキスト (標準型) の Heapdump ファイル・フォーマット　133

I

IBM 提供のファイル
　プリコンパイル　58
ImmortalProperties　63
InstallAnywhere　38

J

Java アプリケーション
　作成　75
　変更　78
Java クラス・ライブラリー
　RTSJ　162
Javadump　126
　診断ツールの使用　126
　ストレージ管理　127
　スレッドおよびスタック・トレース (THREADS)　129
JIT　137
　アイドル　144
　コンパイルの障害、特定　142
　障害が発生したメソッドの特定　140
　診断ツールの使用　137
　選択的な無効化　139
　短期実行アプリケーション　144
　テスト　60
　無効化　138
JIT コンパイラーの無効化　138
JIT の選択的な無効化　139
just-in-time
　テスト　60
JVMTI　153
　診断ツールの使用　153

L

Linux
　環境のセットアップと確認
　　コア・ファイル　105
　既知の制限　109
　クラッシュ、診断　108
　デバッグ手法　106
　問題判別　105
　　パフォーマンス上の問題のデバッグ　108

M

Metronome
　時間ベースのコレクション　5
　制御、プロセッサー使用率の　73
　制限　74
Metronome ガーベッジ・コレクション　5, 73
Metronome ガーベッジ・コレクター
　アラーム・スレッド　5
　コレクション・スレッド　5
Metronome、クラスのアンロード　5

N

NHRT
　安全なクラス　69
　クラス・ロード　63
　スケジューリング　63
　制約　63
　メモリー　63
NLS
　問題判別　110
NoHeapRealtimeThread　16

O

ORB
　デバッグ　110
OutOfMemoryError　112, 151
OutOfMemoryError、永久　116
OutOfMemoryError、スコープ　117

P

PATH
　設定　35
POSIXSignalHandler　19

R

RealtimeThread　16
RTSJ　14

S

SCHED_FIFO　10, 12, 39, 40
SCHED_OTHER　10, 12, 39, 40
SCHED_RR　10, 39, 40
SIGABRT　19
SIGKILL　19
SIGQUIT　19
SIGTERM　19
SIGUSR1　19
SIGUSR2　19
SizeEstimator　16

T

TCK　162
Technology Compatibility Kit　162

［特殊文字］

-agentlib:　156
-agentpath:　156
-assert　156
-classpath　156
-cp　156
-D　156
-help　156
-javaagent:　156
-jre-restrict-search　156
-noRecurse　55
-no-jre-restrict-search　156
-outPath　55
-searchPath　55
-showversion　156
-verbose:　156
-verbose:gc オプション　145
-version:　156
-X　156
-Xbootclasspath/p　158
-Xdebug　26
-Xdump:heap　132
-Xgc:immortalMemorySize　159
-Xgc:immortalMemorySize=size　73
-Xgc:nosynchronousGCOnOOM　159
-Xgc:noSynchronousGCOnOOM オプション　151
-Xgc:scopedMemoryMaximumSize　159
-Xgc:scopedMemoryMaximumSize=size　73
-Xgc:synchronousGCOnOOM　159
-Xgc:synchronousGCOnOOM オプション　151
-Xgc:targetUtilization　159
-Xgc:threads　159
-Xgc:verboseGCCycleTime=N　159
-Xgc:verboseGCCycleTime=N オプション　145
-Xint　7, 40, 158
-Xjit　7, 40, 158
-Xmx　73, 112, 159
-Xnojit　7, 26, 40, 158
-Xrealtime　7, 40, 158
-Xshareclasses　26
-XsynchronousGCOnOOM　112
-?　156

Printed in Japan

日本アイ・ビー・エム株式会社

〒103-8510 東京都中央区日本橋箱崎町19-21