IBM WebSphere Real Time for RT Linux
Version 3

Benutzerhandbuch

IBM WebSphere Real Time for RT Linux
Version 3

Benutzerhandbuch

Hinweis

Vor Verwendung dieser Informationen und des darin beschriebenen Produkts sollten die Informationen unter Kapitel 11, „Bemerkungen", auf Seite 163 gelesen werden.

Erste Ausgabe (August 2011)

Diese Ausgabe des Benutzerhandbuchs bezieht sich auf IBM WebSphere Real Time for RT Linux, Version 3 und auf alle nachfolgenden Releases und Änderungen, bis in neuen Ausgaben etwas anderes angegeben wird.

Diese Veröffentlichung ist eine Übersetzung des Handbuchs
IBM WebSphere Real time for RT Linux Version 3, User's Guide,
herausgegeben von International Business Machines Corporation, USA

Informationen, die nur für bestimmte Länder Gültigkeit haben und für Deutschland, Österreich und die Schweiz nicht zutreffen, wurden in dieser Veröffentlichung im Originaltext übernommen.

Möglicherweise sind nicht alle in dieser Übersetzung aufgeführten Produkte in Deutschland angekündigt und verfügbar; vor Entscheidungen empfiehlt sich der Kontakt mit der zuständigen IBM Geschäftsstelle.

Änderung des Textes bleibt vorbehalten.

Herausgegeben von:
TSC Germany
Kst. 2877
August 2011

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellen

Vorwort

In diesem Handbuch bzw. Benutzerhandbuch erhalten Sie allgemeine Informationen zu IBM® WebSphere Real Time for RT Linux.

Kapitel 1. Einführung

Diese Informationen führen Sie in IBM WebSphere Real Time for RT Linux ein.

Übersicht über WebSphere Real Time for RT Linux

WebSphere Real Time for RT Linux bündelt Echtzeitfunktionalität mit der IBM J9 Virtual Machine (JVM).

WebSphere Real Time for RT Linux ist eine Java Runtime Environment mit einem Software-Development-Kit, das IBM SDK for Java durch Echtzeitfunktionalität erweitert. Anwendungen, die von präzisen Antwortzeiten abhängig sind, können die Echtzeitfunktionen von WebSphere Real Time for RT Linux über Standard-Java-Technologie nutzen.

Features

Echtzeitanwendungen benötigen eine konsistente Ausführung anstatt absoluter Geschwindigkeit.

Wird die JVM im Echtzeitmodus ausgeführt, sind neben dem Heapspeicher, für den eine Garbage-Collection durchgeführt wird, zusätzliche Hauptspeicherbereiche verfügbar. Programme fordern möglicherweise eine Anzahl wiederverwendbarer Speicherbereiche für Objekte mit beschränkter Lebensdauer und nicht wiederverwendbare Speicherbereiche für Objekte mit unbeschränkter Lebensdauer an bzw. geben sie an. Für diese Speicherbereiche wird keine Garbage-Collection durchgeführt. Durch diese Funktionalität kann die Anwendung die Speicherbelegung besser steuern. Außerdem werden mit dem Metronom-Garbage-Collector zeitbasierte Garbage-Collections erzielt. Wenn die JVM in einem traditionellen Durchsatzmodus ausgeführt wird, können verschiedene arbeitsbasierte Garbage-Collector verwendet werden, die den Durchsatz optimieren. Sie haben möglicherweise jedoch längere einzelne Verzögerungen als der Metronom-Garbage-Collector.

Die Hauptproblemstellungen bei der Implementierung von Echtzeitanwendungen mit traditionellen JVMs lauten wie folgt:
- Unvorhersehbare (potenziell lange) Verzögerungen aufgrund der Garbage-Collection-Aktivität
- Verzögerungen für die Methodenlaufzeit bei JIT-Kompilierung und erneuter Kompilierung mit unterschiedlicher Ausführungszeit
- Willkürliche Betriebssystemzeitplanung

WebSphere Real Time for RT Linux entfernt diese Hindernisse durch die Bereitstellung folgender Komponenten:
- Metronom-Garbage-Collector, ein inkrementeller und deterministischer Garbage-Collector mit sehr kurzen Pausezeiten
- AOT-Kompilierung
- Prioritätsbasierte FIFO-Planung

Außerdem bietet WebSphere Real Time dem Echtzeitprogrammierer RTSJ-Funktionen. Weitere Informationen hierzu finden Sie in „Unterstützung für RTSJ” auf Seite 9.

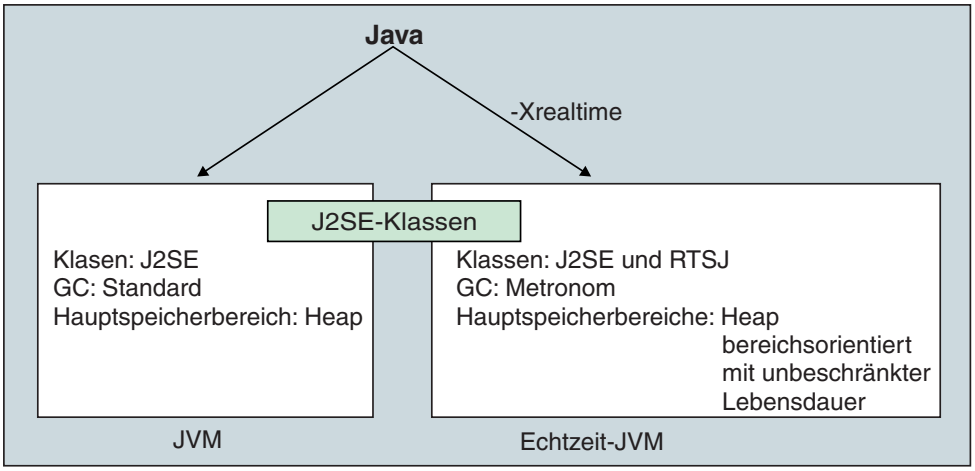

Abbildung 1. Übersicht über WebSphere Real Time for RT Linux

Sie aktivieren die Echtzeitfunktionalität über die Option '-Xrealtime', wenn Sie die JVM oder eins der gelieferten Tools ausführen. Die JVM und gelieferten Tools werden standardmäßig ohne aktivierte Echtzeitfunktionalität ausgeführt. Abb. 1 zeigt die Beziehungen der beiden JVMs, die mit WebSphere Real Time for RT Linux geliefert werden.

Die folgenden Java-Befehle erkennen die Option '-Xrealtime':

Tabelle 1. Im Echtzeitmodus verwendete Java-Befehle

Befehl	**Funktion**
java	Ausführung im Standardmodus, aber auch im Echtzeitmodus bei Angabe der Option '-Xrealtime'. Im Echtzeitmodus greift der Programmierer auf Klassen im Paket javax.realtime zu. Sie können vorkompilierte JAR-Dateien und die deterministische Metronom-Garbage-Collection-Technologie verwenden.
javac, javah, javap	Ausführung im Standardmodus, aber bei Angabe der Option '-Xrealtime' werden die Klassen javax.realtime.* in den Klassenpfad aufgenommen.
admincache	Ausführung mit und ohne '-Xrealtime' möglich, aber ein gemeinsam genutzter Cache kann mithilfe des Tools **admincache** nur im Echtzeitmodus aufgefüllt werden. Im regulären Modus sind nur die Cachedienstprogramme verfügbar (beispielsweise listAllCaches oder printStats). Wie **jdmpview** muss **admincache** mit '-Xrealtime' ausgeführt werden, um auf Caches für die Echtzeit-JVM zuzugreifen, und der Befehl muss ohne '-Xrealtime' ausgeführt werden, um auf Caches für die reguläre JVM zuzugreifen.
jextract	jextract wird im Standardmodus ausgeführt, muss aber mit der Option '-Xrealtime' ausgeführt werden, wenn Systemspeicherauszüge verarbeitet werden, die von der JVM im Echtzeitmodus generiert wurden.

Neuerungen

Dieser Abschnitt enthält eine Einführung in Änderungen für IBM WebSphere Real Time for RT Linux.

WebSphere Real Time for RT LinuxV3

WebSphere Real Time for RT Linux Version 3 ist eine Erweiterung von IBM SDK for Java 7, die auf den in diesem Release verfügbaren Features und Funktionen aufbaut, um Echtzeit-Funktionalität einzuschließen. Ältere Versionen von WebSphere Real Time for RT Linux basierten auf früheren Releases von IBM SDK for Java.

Informationen zu den Neuerungen finden Sie in Neuerungen im Information Center für IBM SDK for Java 7.

jxeinajar

WebSphere Real Time for RT Linux Version 3 unterstützt nicht mehr die Verwendung von 'jxeinajar'. Frühere Informationen zu 'jxeinajar' und insbesondere zum Migrieren auf 'admincache' finden Sie zu Referenzzwecken in der Dokumentation zu WebSphere Real Time for RT Linux Version 2.

Vorteile

Die Vorteile der Echtzeitumgebung bestehen darin, dass Java-Anwendungen mit einem höheren Grad an Vorhersagbarkeit ausgeführt werden als die Standard-JVM und konsistentes Ablaufsteuerungsverhalten für Ihre Java-Anwendung liefern. Hintergrundaktivitäten wie Kompilierung und Garbage-Collection finden zu angegebenen Zeiten statt. Dadurch entfallen nicht erwartete Hintergrundaktivitätsspitzen, wenn Ihre Anwendung ausgeführt wird.

Sie können von diesen Vorteilen profitieren, indem Sie die JVM durch die folgenden Funktionen erweitern:

- Metronom-Garbage-Collection-Echtzeittechnologie
- AOT-Kompilierung
- Unterstützung für Real-Time Specification for Java (RTSJ)

Alle Java-Anwendungen können unverändert in einer Echtzeitumgebung ausgeführt werden und profitieren vom Metronom-Garbage-Collector und seiner deterministischen Garbage-Collection, die in regelmäßigen Intervallen stattfindet. Sie nutzen WebSphere Real Time for RT Linux maximal, indem Sie unter Verwendung von Echtzeitthreads und No-Heap-Echtzeitthreads auf die Echtzeitumgebung ausgerichtete Anwendungen schreiben. Ihre Vorgehensweise hängt von der Ablaufsteuerungsspezifikation der Anwendung ab.

Viele Java-Echtzeitanwendungen können die niedrigen Pausezeiten des Metronom-Garbage-Collectors und von AOT zum Erreichen ihrer Ziele nutzen und die Vorteile der Java-Portierbarkeit beibehalten. Anwendungen mit höheren Anforderungen müssen die RTSJ-Funktionen von Echtzeitthreads und No-Heap-Echtzeitthreads mit dem Speichern für Objekte mit beschränkter und unbeschränkter Lebensdauer verwenden. Bei diesem Ansatz kann Ihre Anwendung nur in einer Echtzeitumgebung ausgeführt werden und nicht von der Portierbarkeit von JSE Java profitieren. Sie müssen außerdem ein komplexeres Programmiermodell entwickeln.

Kapitel 2. Einführung in IBM WebSphere Real Time for RT Linux

In diesem Abschnitt werden die Schlüsselkomponenten von IBM WebSphere Real Time for RT Linux eingeführt.

Einführung in den Metronom-Garbage-Collector

Der Metronom-Garbage-Collector ersetzt den Standard-Garbage-Collector in WebSphere Real Time for RT Linux.

Der Hauptunterschied zwischen der Metronom-Garbage-Collection und der Standard-Garbage-Collection ist, dass die Metronom-Garbage-Collection in kleinen unterbrechbaren Schritten durchgeführt wird, wohingegen die Standard-Garbage-Collection die Anwendung stoppt, während sie Garbage markiert und erfasst.

Beispiel:

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

Das Beispiel gibt an, dass Ihre Anwendung alle 60 ms 80 % ausgeführt wird. Die verbleibenden 20 % der Zeit kann für die Garbage-Collection verwendet werden, sofern zu erfassender Garbage vorhanden ist. Der Metronom-Garbage-Collector garantiert bestimmte Auslastungen, vorausgesetzt ihm wurden ausreichende Ressourcen zugeteilt. Die Garbage-Collection fängt an, wenn der freie Speicherplatz im Heapspeicher unter einen dynamisch ermittelten Schwellenwert fällt.

Garbage-Collection und Prioritäten

Der Garbage-Collection-Thread muss mit einer höheren Priorität ausgeführt werden als der Thread mit der höchsten Priorität, der Garbage im Heapspeicher generiert. Andernfalls kann er möglicherweise nicht so wie von der konfigurierten Auslastung angegeben ausgeführt werden. Reguläre Java-Threads und Echtzeitthreads können Garbage generieren. Daher muss die Garbage-Collection mit einer Priorität ausgeführt werden, die über der Priorität aller regulären Threads und Echtzeitthreads liegt. Diese Priorisierung wird von der JVM automatisch verarbeitet und die Garbage-Collection wird mit einer Priorität ausgeführt, die 0,5 über der höchsten Priorität aller regulären und Echtzeitthreads liegt. Sie müssen jedoch sicherstellten, dass No-Heap-Echtzeitthreads (NHRT-Threads) von der Garbage-Collection nicht betroffen sind. Führen Sie alle NHRT-Threads mit einer höheren Priorität als die Echtzeitthreads mit der höchsten Priorität aus. Dies bedeutet, dass NHRT-Threads mit einer höheren Priorität ausgeführt werden als die Garbage-Collection und nicht verzögert werden.

Tabelle 2 zeigt ein typisches Beispiel für die Prioritäten, die Sie definieren können, und die zugehörigen Garbage-Collection-Prioritäten, die sich aus Ihrer Auswahl ergeben.

Einen Vergleich zwischen Java-Prioritäten und Betriebssystemprioritäten finden Sie in „Prioritätszuordnung und -übernahme" auf Seite 12.

Tabelle 2. Beispiel für Garbage-Collection und Prioritäten

Threads	**Prioritäten (Beispiele)**
Echtzeitthread mit der höchsten Priorität:	20 (Betriebssystempriorität 43)
Garbage-Collector:	20,5 (Betriebssystempriorität 44)
Legen Sie eine höhere Priorität als die Priorität der GC fest, um sicherzustellen, dass ein NHRT-Thread unabhängig vom Garbage-Collector ausgeführt wird:	21 (Betriebssystempriorität 45) oder höher
Metronomalarmthread:	Priorität 46 (Betriebssystempriorität 89)

Anmerkung: Selbst bei dieser Konfiguration sind No-Heap-Echtzeitthreads von der Garbage-Collection betroffen, weil der Metronomalarmthread mit der höchsten Priorität im System ausgeführt wird, um sicherzustellen, dass er regelmäßig aktiviert werden und ermitteln kann, ob die Garbage-Collection tätig werden muss. Die hierfür zu verrichtende Arbeit ist geringfügig und kann daher vernachlässigt werden.

Metronom-Garbage-Collection und Klassenentladung

Der Metronom-Garbage-Collector entlädt keine Klassen in IBM WebSphere Real Time, weil für ihn eine nicht deterministische Arbeitslast erforderlich sein kann, die Pausezeitausreißer verursacht.

Metronom-Garbage-Collector-Threads

Der Metronom-Garbage-Collector besteht aus zwei Typen von Threads: ein einzelner Alarmthread und eine Anzahl GC-Threads. Es gibt standardmäßig einen GC-Thread. Sie können die Anzahl GC-Threads für die JVM mit der Option **`-Xgcthreads`** festlegen.

Sie können die Anzahl Alarmthreads für die JVM nicht ändern.

Der Metronom-Garbage-Collector prüft die JVM periodisch daraufhin, ob der Heapspeicher über ausreichend freien Speicherplatz verfügt. Wenn die freie Speicherplatzkapazität unter den Grenzwert fällt, löst der Metronom-Garbage-Collector den Start der Garbage-Collection durch die JVM aus.

Alarmthread

Der einzelne Alarmthread garantiert die Verwendung minimaler Ressourcen. Er wird in regelmäßigen Intervallen aktiviert und prüft Folgendes:

- Freie Speicherplatzkapazität im Heapspeicher
- Ob die Garbage-Collection zurzeit durchgeführt wird

Wenn der freie Speicherplatz nicht ausreicht und keine Garbage-Collection durchgeführt wird, weist der Alarmthread die Collection-Threads an, die Garbage-Collection zu starten. Der Alarmthread führt keine Aktivitäten aus, bis er das nächste Mal die JVM prüft.

Collection-Threads

Jeder Collection-Thread prüft Java-Threads und Echtzeitthreads auf Heapspeicherobjekte. Sie prüfen die Hauptspeicherbereiche in der folgenden Reihenfolge:

1. Speicher für Objekte mit beschränkter Lebensdauer, um Liveobjekte im Heapspeicher, die von Objekten vom Speicher für Objekte mit beschränkter Lebensdauer verwendet werden, zu ermitteln und zu markieren.
2. Speicher für Objekte mit unbeschränkter Lebensdauer, um Liveobjekte im Heapspeicher, die von Objekten vom Speicher für Objekte mit unbeschränkter Lebensdauer verwendet werden, zu ermitteln und zu markieren.
3. Heapspeicher, um Liveobjekte zu ermitteln und zu markieren.

Wenn die Liveobjekte markiert wurden, sind die nicht markierten Objekte für Collection verfügbar.

Nach dem Abschluss des Garbage-Collection-Zyklus prüft der Metronom-Garbage-Collector, wie viel Heapspeicher frei ist. Wenn weiterhin nicht genügend Heapspeicher frei ist, wird ein weiterer Garbage-Collection-Zyklus mit derselben Trigger-ID gestartet. Wenn ausreichend Heapspeicher frei ist, wird der Trigger beendet und werden die Garbage-Collection-Threads gestoppt. Der Alarmthread überwacht weiterhin den freien Heapspeicher und löst bei Bedarf einen weiteren Garbage-Collection-Zyklus aus.

Weitere Informationen zum Verwenden des Metronom-Garbage-Collectors finden Sie in „Metronom-Garbage-Collector verwenden" auf Seite 72.

Compiler

IBM WebSphere Real Time for RT Linux unterstützt mehrere Codekompilierungsmodelle, die verschiedene Ebenen der Codeleistung und der Bestimmungsfunktion liefern.

Für das Kompilieren von Java-Code mit IBM WebSphere Real Time for RT Linux sind die folgenden Optionen verfügbar:

JIT-Kompilierung mit niedriger Priorität

Das Standardkompilierungsmodell in WebSphere Real Time for RT Linux verwendet einen JIT-Compiler, um die wichtigen Methoden einer Java-Anwendung während der Anwendungsausführung zu kompilieren. In diesem Modus funktioniert der JIT-Compiler ähnlich wie die Operation des JIT-Compilers in einer Nicht-Echtzeit-JVM. Der Unterschied ist, dass der JIT-Compiler von WebSphere Real Time for RT Linux auf einer niedrigeren Prioritätsebene ausgeführt wird als Echtzeitthreads. Die niedrigere Priorität bedeutet, dass der JIT-Compiler Systemressourcen verwendet, wenn die Anwendung keine Echtzeittasks auszuführen braucht. Hierdurch wirkt sich der JIT-Compiler nicht wesentlich auf die Leistung von Echtzeittasks aus.

Vorkompilierter AOT-Code

WebSphere Real Time for RT Linux kompiliert Java-Methoden in einem Vorkompilierungsschritt zu nativem Code, bevor die Anwendung ausgeführt wird. Die Verwendung von mit AOT vorkompiliertem Code bietet die höchste Bestimmungsfunktionsebene mit guter Leistung.

Gemischter Modus, der mit AOT vorkompilierten Code und JIT-Kompilierung mit niedriger Priorität kombiniert
Mit AOT und JIT kompilierter Code kann während der Anwendungsausführung zusammen verwendet werden. Diese Betriebsart kann eine sehr gute Bestimmungsfunktion mit guter Leistung und eine sehr hohe Leistung für häufig ausgeführte Methoden bieten.

Interpretierte Operation
Der Interpreter führt eine Java-Anwendung aus, verwendet die Codekompilierung allerdings nicht.

Weitere Informationen zum Verwenden von kompiliertem Code finden Sie in „Für WebSphere Real Time for RT Linux kompilierten Code verwenden" auf Seite 40.

JIT- und AOT-Kompilierung vergleichen

Mit der Ahead-of-Time-Kompilierung (AOT-Kompilierung) können Sie Java-Klassen und -Methoden vor der Ausführung Ihres Codes kompilieren. Die AOT-Kompilierung verhindert die unvorhersehbare Ablaufsteuerungsauswirkung, die der Just-in-time-Compiler (JIT-Compiler) auf sensible Leistungspfade haben kann. Sie können Ihren Code mit dem AOT-Compiler in einem Cache für gemeinsam genutzte Klassen vorkompilieren, um sicherzustellen, dass Ihr Code vor seiner Ausführung kompiliert ist und um die höchste Ebene deterministischer Leistung zu erzielen.

Anmerkung: Mit AOT kompilierter Code wird in der Regel nicht so schnell ausgeführt wie mit JIT kompilierter Code.

Der JIT-Compiler wird als SCHED_OTHER-Thread mit hoher Priorität ausgeführt. Diese Priorität ist höher als die Priorität von Standard-Java-Threads, allerdings niedriger als die Priorität von Echtzeitthreads. Die JIT-Kompilierung verursacht daher im Echtzeitcode keine nicht deterministischen Verzögerungen. Wichtige Echtzeitarbeit wird deshalb termingerecht ausgeführt, weil sie vom JIT-Compiler nicht zurückgestellt wird. Der Echtzeitcode wird möglicherweise jedoch als interpretierter Code ausgeführt, weil der JIT-Compiler nicht genug Zeit hatte, die aufgelaufenen Methoden mit einem hohen Ressourcenverbrauch zu kompilieren. Ein Vergleich zwischen JIT und AOT finden Sie in Abb. 2 auf Seite 9.

Wenn Ihre Anwendung eine Einstiegsphase hat, ist es im Allgemeinen effizienter, den Code mit dem JIT-Compiler auszuführen und den JIT-Compiler bei Bedarf zu inaktivieren, wenn die Einstiegsphase abgeschlossen ist. Hierdurch kann der JIT-Compiler Code für die Umgebung generieren, in der die Anwendung ausgeführt wird.

Wenn die Anwendung keine Einstiegsphase hat und wenn nicht klar ist, ob Schlüsselpfade der Ausführung über die Standardanwendungsoperation kompiliert werden, funktioniert die AOT-Kompilierung in dieser Umgebung gut.

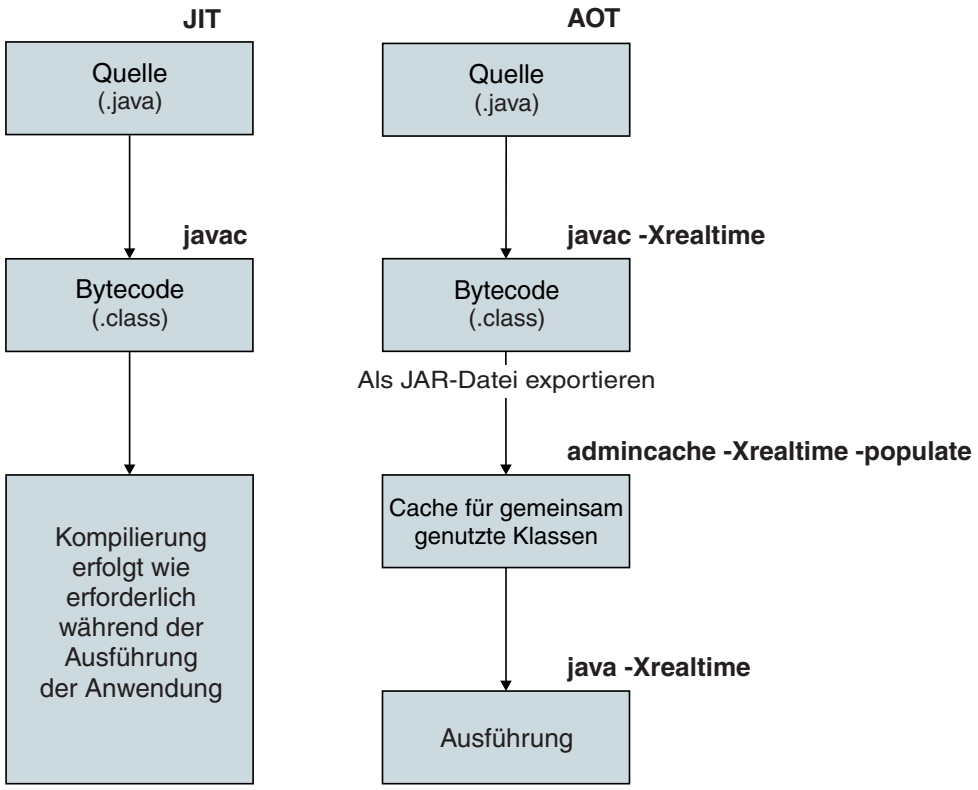

Abbildung 2. Vergleich zwischen dem JIT-Compiler und AOT-Compiler

Unterstützung für RTSJ

WebSphere Real Time for RT Linux implementiert Real-Time Specification for Java (RTSJ).

WebSphere Real Time for RT Linux Version 3.0 wurde als RTSJ-kompatibel mit RTSJ Technology Compatibility Kit 1.0.2 Version J9 3.1.0 FCS zertifiziert und ist mit Java Compatibility Kit (JCK) Version 7.0 kompatibel.

Echtzeitthreadplanung und -zuteilung

Die Threadplanung und -zuteilung von Java-Echtzeitthreads ist Teil von Real Time Specification for Java. Mit der Planungsrichtlinie SCHED_FIFO und den Prioritäten 11-89 des Betriebssystems Linux werden dieJava-Echtzeitthreads priorisiert.

Informationen zu Linux-Planungsrichtlinien finden Sie in „Threadplanung und -zuteilung" auf Seite 39.

Planbare Objekte und ihre Parameter

Es gibt zwei Haupttypen von planbaren Echtzeitobjekten: Echtzeitthreads und Handler für asynchrone Ereignisse.

Diesen planbaren Objekten sind die folgenden Parameter zugeordnet:

SchedulingParameters
PriorityParameters plant planbare Echtzeitobjekte nach der Priorität.

ReleaseParameters

- **PeriodicParameters** beschreibt die periodische Freigabe von planbaren Echtzeitobjekten. Ein periodischer Echtzeitthread ist ein Thread, der in regelmäßigen Intervallen freigegeben wird.
- **AperiodicParameters** beschreibt die Freigabe von planbaren Echtzeitobjekten. Aperiodische Echtzeitthreads werden in unregelmäßigen Intervallen freigegeben.

MemoryParameters
Beschreibt die Einschränkungen der Hauptspeicherzuordnungen für planbare Echtzeitobjekte.

ProcessingGroupParameters
In WebSphere Real Time for RT Linux nicht unterstützt.

Prioritätsscheduler

In WebSphere Real Time for RT Linux ist der Planer ein Prioritätsscheduler. Wie sein Name angibt, verwaltet er die Ausführung von planbaren Objekten auf der Grundlage ihrer aktiven Prioritäten.

Der Scheduler verwaltet die Liste planbarer Objekte und ermittelt, wann die einzelnen Objekte zur Ausführung in der CPU freigegeben werden können. Der Scheduler muss die verschiedenen Parameter für die einzelnen planbaren Objekte einhalten. Zu diesem Zweck werden die Methoden addToFeasibility, isFeasible und removeFromFeasibility bereitgestellt.

Prioritäten und Richtlinien

Reguläre Java-Threads, d. h. als `java.lang.Thread`-Objekte zugeordnete Threads, können die Planungsrichtlinie SCHED_OTHER, SCHED_RR oder SCHED_FIFO verwenden. Echtzeitthreads, d. h. als `java.lang.RealtimeThread`-Objekte zugeordnete Threads, und Handler für asynchrone Ereignisse verwenden die Planungsrichtlinie SCHED_FIFO.

Reguläre Java-Threads verwenden die Standardplanungsrichtlinie SCHED_OTHER, außer die JVM wird durch einen Thread mit der Richtlinie SCHED_RR oder SCHED_FIFO gestartet. Die Betriebssystempriorität von regulären Java-Threads, die die Richtlinie SCHED_OTHER verwenden, ist auf 0 gesetzt. Reguläre Java-Threads, die die Richtlinie SCHED_RR oder SCHED_FIFO verwenden, übernehmen die Priorität des Threads, der die JVM startet.

Bei Echtzeitthreads verfügt die Richtlinie SCHED_FIFO über kein Zeitscheibenverfahren und unterstützt sie 99 Prioritäten von 1 (niedrigste Priorität) bis 99 (höchste Priorität). Diese WebSphere Real Time for RT Linux-Implementierung unterstützt 28 Benutzerprioritäten zwischen 11 und 38 einschließlich. Beispiel 1:

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

Hierdurch wird 11 zurückgegeben. Beispiel 2:

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

Hierdurch wird 38 zurückgegeben.

Die Betriebssystemprioritäten 81-89 werden von der IBM JVM für die Zuteilung von Worker-Threads verwendet. Diese Threads sind so konzipiert, dass sie wenig Arbeit verrichten, bevor sie inaktiviert werden. Hierbei handelt es sich um die folgenden Threads:

- Der Metronom-Garbage-Collector-Alarmthread wird mit der Priorität 89 ausgeführt. Dieser Thread wird regelmäßig ausgeführt und teilt eine GC-Arbeitseinheit zu.
- Zwei Threade, die asynchrone Signale verarbeiten. Einer von ihnen ist der No-Heap-Echtzeitthread (NHRT-Thread). Er hat die Priorität 88 und der andere Thread hat die Priorität 87.
- Zwei Zeitgeberthreads, die Zeitgeberereignisse zuteilen. Einer von ihnen ist der No-Heap-Echtzeitthread für No-Heap-Zeitgeber. Er hat die Priorität 85 und der andere Thread hat die Priorität 83.
- Die Threads für Handler für asynchrone Ereignisse, die für die Ausführung von Handlern für asynchrone Ereignisse zugeteilt werden und denen bei der Ausführung eines Handlers für asynchrone Ereignisse dessen Priorität zugeordnet wird. Das System wird mit zwei No-Heap-Echtzeit-Handler-Threads mit den Prioritäten 85 und 8 gestartet, während andere Threads die Priorität 83 haben.
- Der No-Heap-Echtzeitthread für asynchrone Signale mit der Priorität 88 verarbeitet Anforderungen für Heapspeicherauszüge, Kernspeicherauszüge und Auszüge des Java-Kernspeichers. Seine Priorität wird beim Erstellen von Speicherauszugsdateien vorübergehend auf 89 erhöht.

TraceDer Metronom-GC-Trace-Thread wird mit der Betriebssystempriorität 12 ausgeführt und der JIT-Sampler-Thread, der Java-Methoden für die Kompilierung stichprobenmäßig ermittelt, wird mit der Betriebssystempriorität 13 ausgeführt.

Der JIT-Kompilierungsthread (der sich vom JIT-Sampler-Thread unterscheidet) wird mit der Richtlinie SCHED_OTHER und der Betriebssystempriorität 0 ausgeführt.

Die JIT-Kompilierungs- und JIT-Sampler-Threads sind inaktiviert, wenn **-Xnojit** oder **-Xint** angegeben wird.

Die Metronom-Garbage-Collector- und Finalizerpriorität wird ständig (d. h. vor jeder Erfassung) geändert, sodass sie über der höchsten Priorität des Threads liegt, der den Heapspeicher zuordnet. Sie müssen sicherstellen, dass die Priorität der Threads, die den Heapspeicher zuordnen, die Priorität von NoHeapRealtime-Threads unterschreitet.

Ein Thread, der den Heapspeicher zuordnet, ist ein beliebiger Nicht-NHRT-Benutzerthread, der nicht inaktiviert oder in einem Monitor nicht geblockt ist. Ein Benutzerthread, der nativen Code außerhalb der JNI-Schnittstelle ausführt, wird nicht als Thread erachtet, der Heapspeicher zuordnet. Läuft eine Garbage-Collection, wenn ein Heapspeicher zuordnender Thread aktiviert wird, im Monitor nicht mehr geblockt wird oder JNI verlässt, muss der Thread warten, bis die Garbage-Collection beendet ist, bevor er fortfahren kann.

Die Betriebssystempriorität 81 ist für interne JVM-Threads reserviert, die Zuordnungen vom Heapspeicher vornehmen. Wenn ein interner JVM-Thread die Betriebssystempriorität 81 hat, wird der Garbage-Collector mit der Betriebssystempriorität 82 ausgeführt. Wenn die Benutzerthreads, die nur Heapspeicher zuordnen, keine Echtzeitthreads sind, wird die GC-Priorität mit der Betriebssystempriorität 11 ausgeführt. Andernfalls wird die GC mit einer Priorität ausgeführt, die eine Betriebssystempriorität höher ist als die höchste Priorität der Benutzerthreads, die Heapspeicher zuordnen.

Die GC-Priorität wird vor jeder Erfassung angepasst.

Prioritätszuordnung und -übernahme

Jede Java-Priorität wird einer entsprechenden Betriebssystembasispriorität zugeordnet und jede Betriebssystempriorität wird einer Planungsrichtlinie zugeordnet. Die Planungsrichtlinien des Betriebssystems Linux lauten SCHED_OTHER, SCHED_RR und SCHED_FIFO.

Java-Echtzeitthreads verwenden die Richtlinie SCHED_FIFO, während reguläre Java-Threads die Priorität des Threads verwenden, der die JVM startet. Die Standardplanungsrichtlinie für reguläre Java-Threads ist SCHED_OTHER, über ein Dienstprogramm wie **chrt** können Sie jedoch die Richtlinie SCHED_RR oder SCHED_FIFO festlegen. Weitere Informationen zu Threadprioritäten und -richtlinien finden Sie in „Threadplanung und -zuteilung" auf Seite 39.

In der folgenden Tabelle wird gezeigt, wie die Java-Prioritäten den nativen Betriebssystemprioritäten zugeordnet werden. Einige Java-Prioritäten sind für die JVM-Verwendung reserviert und einige native Prioritäten ohne entsprechende Java-Prioritäten werden auch von der JVM verwendet.

Anmerkung:

- Die Prioritäten 1-10 werden von regulären Java-Threads verwendet.
  - Bei der Richtlinie SCHED_OTHER werden die Java-Prioritäten 1-10 der Betriebssystempriorität 0 zugeordnet.
  - Bei der Richtlinie SCHED_FIFO oder SCHED_RR übernehmen die Java-Prioritäten 1-10 die Priorität des Threads, der die JVM startet.
- Prioritäten ab 11 werden von Echtzeitthreads und No-Heap-Echtzeitthreads verwendet.
- Ein planbares Objekt wird immer mit seiner aktiven Priorität ausgeführt. Die aktive Priorität ist anfänglich die Basispriorität des planbaren Objekts, sie kann jedoch durch Prioritätsübernahme vorübergehend erhöht werden. Die Basispriorität eines planbaren Objekts kann während seiner Ausführung geändert werden.

Benutzerbasisprioritäten:

```
Java-Prioritäten 1-10: SCHED_OTHER, Betriebssystempriorität 0

Java-Priorität 11: SCHED_FIFO, Betriebssystempriorität 25
Java-Priorität 12: SCHED_FIFO, Betriebssystempriorität 27
Java-Priorität 13: SCHED_FIFO, Betriebssystempriorität 29
Java-Priorität 14: SCHED_FIFO, Betriebssystempriorität 31
Java-Priorität 15: SCHED_FIFO, Betriebssystempriorität 33
Java-Priorität 16: SCHED_FIFO, Betriebssystempriorität 35
Java-Priorität 17: SCHED_FIFO, Betriebssystempriorität 37
Java-Priorität 18: SCHED_FIFO, Betriebssystempriorität 39
Java-Priorität 19: SCHED_FIFO, Betriebssystempriorität 41
Java-Priorität 20: SCHED_FIFO, Betriebssystempriorität 43
Java-Priorität 21: SCHED_FIFO, Betriebssystempriorität 45
Java-Priorität 22: SCHED_FIFO, Betriebssystempriorität 47
Java-Priorität 23: SCHED_FIFO, Betriebssystempriorität 49
Java-Priorität 24: SCHED_FIFO, Betriebssystempriorität 51
Java-Priorität 25: SCHED_FIFO, Betriebssystempriorität 53
Java-Priorität 26: SCHED_FIFO, Betriebssystempriorität 55
Java-Priorität 27: SCHED_FIFO, Betriebssystempriorität 57
Java-Priorität 28: SCHED_FIFO, Betriebssystempriorität 59
Java-Priorität 29: SCHED_FIFO, Betriebssystempriorität 61
Java-Priorität 30: SCHED_FIFO, Betriebssystempriorität 63
Java-Priorität 31: SCHED_FIFO, Betriebssystempriorität 65
Java-Priorität 32: SCHED_FIFO, Betriebssystempriorität 67
Java-Priorität 33: SCHED_FIFO, Betriebssystempriorität 69
Java-Priorität 34: SCHED_FIFO, Betriebssystempriorität 71
Java-Priorität 35: SCHED_FIFO, Betriebssystempriorität 73
```

```
Java-Priorität 36: SCHED_FIFO, Betriebssystempriorität 75
Java-Priorität 37: SCHED_FIFO, Betriebssystempriorität 77
Java-Priorität 38: SCHED_FIFO, Betriebssystempriorität 79
```

Interne Basisprioritäten:

```
Interne Java-Priorität 39: SCHED_FIFO, Betriebssystempriorität 81
Interne Java-Priorität 40: SCHED_FIFO, Betriebssystempriorität 83
Interne Java-Priorität 41: SCHED_FIFO, Betriebssystempriorität 84
Interne Java-Priorität 42: SCHED_FIFO, Betriebssystempriorität 85
Interne Java-Priorität 43: SCHED_FIFO, Betriebssystempriorität 86
Interne Java-Priorität 44: SCHED_FIFO, Betriebssystempriorität 87
Interne Java-Priorität 45: SCHED_FIFO, Betriebssystempriorität 88
Betriebssystemprioritäten 11, 12, 13
Betriebssystemprioritäten mit den geraden Zahlen 26, 28, 30, ..., 82
Betriebssystempriorität 89
```

Weitere Informationen finden Sie im Abschnitt zur Synchronisation in http://www.rtsj.org/specjavadoc/book_index.html.

Prioritätsübernahme:

Die aktive Priorität eines Threads kann vorübergehend aufgewertet werden, weil er über eine Sperre verfügt, die von einem Thread mit einer höheren Priorität benötigt wird. Diese Sperren können interne JVM-Sperren oder Monitore auf Benutzerebene sein, die synchronisierten Methoden oder synchronisierten Blöcken zugeordnet sind. Die Priorität eines regulären Java-Threads kann daher vorübergehend eine Echtzeitpriorität sein, bis der Thread die Sperre freigegeben hat.

Eine Folge der Prioritätsübernahme ist, dass die Threadrichtlinie eines SCHED_OTHER-Threads vorübergehend in SCHED_FIFO geändert wird.

Weitere Informationen zu Basisprioritäten und aktiven Prioritäten finden Sie im Abschnitt zur Synchronisation in der RTSJ-Spezifikation.

Speicherverwaltung

Garbage-Collection-Heapspeicher sind aufgrund des unvorhersehbaren Verhaltens, das die Garbage-Collection verursacht, immer als Hindernis für Echtzeitprogrammierung betrachtet worden. Der Metronom-Garbage-Collector in IBM WebSphere Real Time for RT Linux kann hohe deterministische GC-Leistung liefern. Außerdem stellt Real-Time Specification for Java (RTSJ) mehrere Erweiterungen für das Speichermodell für Objekte außerhalb des Garbage-Collection-Heapspeichers bereit, damit ein Java-Programmierer kurz- und langlebige Objekte explizit verwalten kann.

Hauptspeicherbereiche

RTSJ führt das Konzept eines Hauptspeicherbereichs ein, der für die Zuordnung von Objekten verwendet werden kann. Einige Hauptspeicherbereiche sind außerhalb des Heapspeichers vorhanden und beschränken die Aktivität des Systems und Garbage-Collectors für Objekte. Beispielsweise wird für Objekte in einigen Hauptspeicherbereichen keine Garbage-Collection durchgeführt, der Garbage-Collector kann diese Hauptspeicherbereiche jedoch auf Verweise auf ein Objekt im Heapspeicher durchsuchen, um die Integrität des Heapspeichers zu bewahren.

Für die Speicherverwaltung gibt es drei Basistypen:

- Der Heapspeicher ist der traditionelle Java-Heapspeicher, er wird jedoch vom Metronom-Garbage-Collector verwaltet.

- Der Speicher für Objekte mit beschränkter Lebensdauer muss von Anwendungen angefordert werden und kann nur von Echtzeitthreads einschließlich No-Heap-Echtzeitthreads und Handler für asynchrone No-Heap-Ereignisse verwendet werden.
- Der Speicher für Objekte mit unbeschränkter Lebensdauer stellt einen Hauptspeicherbereich mit Objekten dar, auf den ein beliebiges planbares Objekt einschließlich No-Heap-Echtzeitthreads und Handler für asynchrone No-Heap-Ereignisse verweisen kann. Dieser Speicher wird beim Laden von Klassen und bei der statischen Initialisierung verwendet, selbst wenn die Anwendung ihn nicht verwendet.

Für den Speicher für Objekte mit unbeschränkter bzw. beschränkter Lebensdauer kann die Verwendung von physischem Hauptspeicher angegeben werden, der aus Hauptspeicherbereichen mit bestimmten Merkmalen wie wesentlich schnellerer Zugriff besteht. Im Allgemeinen wird der physische Hauptspeicher nicht häufig verwendet und wirkt sich gewöhnlich nicht auf den Standard-JVM-Benutzer aus.

Heapspeicher

Die maximale Größe wird von **-Xmx** gesteuert, legen Sie die Anfangsgröße des Heapspeichers (**-Xms**) jedoch *nicht* fest oder setzen Sie sie auf die maximale Größe des Heapspeichers (**-Xmx**), weil der Heapspeicher in Echtzeit nie von seiner Anfangsgröße auf seine maximale Größe erweitert wird. Wenn Sie die maximale Größe des Heapspeichers erreichen und kein Speicherbereich frei ist, führt dies zu OutOf MemoryError. Im Allgemeinen belegt die Echtzeit-JVM mehr Heapspeicher als die traditionelle JVM, weil Objekte für die unterstützende deterministische Datenerfassung anders organisiert werden müssen. Dies führt zu einer höheren Heapspeicherfragmentierung. Außerdem werden Arrays in Fragmente mit jeweils einem Header aufgeteilt. Es ist wahrscheinlich, dass Sie eine Anwendung finden, die 20 % mehr Heapspeicher benötigt, obwohl dies im Einzelnen vom Verhältnis von großen zu kleinen Objekten und der Array-Nutzung abhängt.

Der Metronom-Garbage-Collector ähnelt dem „meistens gleichzeitig ablaufenden" Collector in der Standard-JVM insofern, als dass er eine Garbage-Collection durchführt, während die Anwendung aktiv ist. Im Idealfall wird der Erfassungszyklus abgeschlossen, bevor für die Anwendung kein Speicher mehr verfügbar ist. Einige Anwendungen mit sehr hohen Zuordnungsraten können jedoch schneller Objekte zuordnen, als der Metronom-Garbage-Collector Objekte erfassen kann. Verschiedene detaillierte Steuerelemente wirken sich auf die Erfassungsrate aus, ein Steuerelement zwingt Metronom allerdings zu einer traditionellen STW-Garbage-Collection, bevor OutOf MemoryError ausgelöst wird. Der Laufzeitparameter lautet **-Xgc:synchronousGCOnOOM** und das Pendant lautet **-Xgc:nosynchronousGCOnOOM**. Der Standardparameter ist **-Xgc:synchronousGCOnOOM**.

Speicher für Objekte mit beschränkter Lebensdauer

RTSJ führt das Konzept von Speicher für Objekte mit beschränkter Lebensdauer ein. Dieser Speicher kann von Objekten mit einer klar definierten Lebensdauer verwendet werden. Eine Lebensdauer kann explizit eingegeben oder an ein planbares Objekt (Echtzeitthread oder Handler für asynchrone Ereignisse) angehängt werden, das somit in den Bereich aufgenommen wird, bevor die Objektmethode run() ausgeführt wird. Jede Lebensdauer ist durch einen Referenzzähler ausgewiesen. Wenn dieser null erreicht, können die im entsprechenden Bereich befindlichen Objekte (ab)geschlossen werden, woraufhin der diesem Bereich zugeordnete Speicher freigegeben wird. Die Wiederverwendung des Bereichs wird geblockt, bis die Endbearbeitung abgeschlossen ist.

Der Speicher für Objekte mit beschränkter Lebensdauer kann in zwei Typen unterteilt werden: VTMemory und LTMemory. Diese Typen von Speicher für Objekte mit beschränkter Lebensdauer unterscheiden sich durch die Zeit, die zum Zuordnen von Objekten aus dem Bereich erforderlich ist. LTMemory garantiert lineare Zeitzuordnung, wenn die Speicherbelegung des Hauptspeicherbereichs kleiner ist als die Anfangsgröße des Hauptspeicherbereichs. VTMemory bietet diese Garantie nicht.

Bereiche können verschachtelt werden. Wenn Daten in einem verschachtelten Bereich verarbeitet werden, kommen alle nachfolgenden Zuordnungen aus dem Speicher, der dem neuen Bereich zugeordnet ist. Wenn der verschachtelte Bereich abgeschlossen ist, wird der vorherige Bereich wiederhergestellt, und nachfolgende Zuordnungen kommen wieder aus diesem Bereich.

Da die Lebensdauer der Objekte beschränkt ist, müssen die Verweise auf die Objekte durch eine Gruppe von einschränkenden Zuordnungsregeln begrenzt werden. Ein Verweis auf ein Objekt mit beschränkter Lebensdauer kann keiner Variablen aus einem übergeordneten Bereich bzw. keinem Feld eines Objekts im Heapspeicher oder Bereich für Objekte mit unbeschränkter Lebensdauer zugeordnet werden. Ein Verweis auf ein Objekt mit beschränkter Lebensdauer kann nur im selben Bereich oder in einem untergeordneten Bereich zugeordnet werden. Die VM erkennt falsche Zuordnungsversuche und löst die Ausnahmebedingung IllegalAssignmentError aus, wenn sie auftreten. Aufgrund der Auswahlflexibilität hinsichtlich der Speichertypen für Objekte mit beschränkter Lebensdauer kann die Anwendung einen Hauptspeicherbereich verwenden, dessen Merkmale zu einem bestimmten syntaktisch definierten Bereich des Codes passen.

Die Größe des Bereichs muss während seiner Erstellung angegeben werden und der Befehlszeilenparameter **-Xgc:scopedMemoryMaximumSize** steuert den Maximalwert. Der Standardwert beträgt 8 MB und ist für die meisten Zwecke angemessen.

Speicher für Objekte mit unbeschränkter Lebensdauer

Der Speicher für Objekte mit unbeschränkter Lebensdauer ist eine von allen planbaren Objekten und Threads in einer Anwendung gemeinsam genutzte Speicherressource. Im Speicher für Objekte mit unbeschränkter Lebensdauer zugeordnete Objekte sind für No-Heap-Threads und Handler für asynchrone Ereignisse immer verfügbar und werden durch die Garbage-Collection nicht verzögert. Objekte werden vom System freigegeben, wenn das Programm beendet wird.

Die Größe wird durch **-Xgc:immortalMemorySize** gesteuert.
-Xgc:immortalMemorySize=20m z. B. legt 20 MB fest. Der Standardwert ist 16 MB. Er ist in der Regel angemessen, sofern Sie nicht viele Klassen laden. Das Laden von Klassen ist die wahrscheinlichste Ursache der meisten Ausnahmebedingungen OutOfMemoryError.

Speicherbedarf schätzen

Vorgehensweise beim Anfordern von Informationen, die zum Zuordnen von ausreichendem Speicher erforderlich sind.

Ein angemessener Ansatz besteht darin, den Speicher, der zum Aufbewahren der erwarteten Objekte erforderlich ist, unter Berücksichtigung eines realistischen Puffers zu ermitteln. Die Analyse der Anwendung hilft Ihnen beim Ermitteln der Anzahl und Art von erforderlichen Objekten, obwohl die tatsächliche Größe, die für eine Größe erforderlich ist, zwischen verschiedenen Systemen unterschiedlich sein

kann. Die Verwendung der Klasse SizeEstimator berücksichtigt die tatsächliche Objektgröße und bietet portierbarere Informationen.

Klasse SizeEstimator

Die Klasse SizeEstimator liefert Informationen zur Speicherkapazität, die zum Speichern eines Objekts benötigt wird. Die Schätzung ist eine Indikation der minimalen Hauptspeicherkapazität, die für das Objekt selbst zugeordnet werden muss. Sie berücksichtigt nicht den Speicherbedarf für andere Ressourcen, die möglicherweise vom Objekt benötigt werden, z. B. während der Erstellung.

Details zu dieser Klasse finden Sie in http://www.rtsj.org/specjavadoc/book_index.html.

Speicher verwenden

Es folgt ein Vergleich von Java-Threads, Echtzeitthreads und No-Heap-Echtzeitthreads.

Real-Time Specification for Java (RTSJ) fügt zwei Klassen zur Unterstützung von Echtzeitthreads hinzu: RealtimeThread und NoHeapRealtimeThread.

- Echtzeitthreads und No-Heap-Echtzeitthreads sind planbare Objekte. Als planbare Objekte haben sie die folgenden Parameter: Release, Planung, Speicher und Verarbeitungsgruppe.
- Echtzeitthreads können auf Objekte im Heapspeicher sowie im Speicher für Objekte mit beschränkter Lebensdauer und im Speicher für Objekte mit unbeschränkter Lebensdauer zugreifen.
- No-Heap-Echtzeitthreads greifen nur auf Speicherbereiche für Objekte mit beschränkter und unbeschränkter Lebensdauer zu.
- No-Heap-Echtzeitthreads benötigen eine höhere Priorität als andere Echtzeitthreads. Wenn ihre Priorität niedriger als die Priorität anderer Echtzeitthreads ist, können sie nicht mehr ohne Interferenz vom Garbage-Collector ausgeführt werden.

Anmerkung: Ein No-Heap-Echtzeitthread mit einer höheren Priorität als andere Echtzeitthreads wird von der Garbage-Collection nicht unterbrochen.

Tabelle 3. Speicherzugriff durch Echtzeit- und No-Heap-Echtzeitthreads

Threads	Speicher für Objekte mit unbeschränkter Lebensdauer	Speicher für Objekte mit beschränkter Lebensdauer	Heapspeicher
Normale Threads	✓	✗	✓
Echtzeitthreads	✓	✓	✓
No-Heap-Echtzeitthreads	✓	✓	✗

Typen des Hauptspeicherbereichs

Speicher für Objekte mit unbeschränkter Lebensdauer
Für den Speicher für Objekte mit unbeschränkter Lebensdauer wird keine Garbage-Collection durchgeführt. Ein Speicherbereich

kann nach seiner Zuordnung im Speicher für Objekte mit unbeschränkter Lebensdauer erst konsolidiert werden, nachdem die Anwendung beendet wurde.

- Aufgrund dieser Merkmale empfiehlt es sich, Wege der Wiederverwendung für den Speicher für Objekte mit unbeschränkter Lebensdauer zu finden. Eine Möglichkeit ist, einen Pool wiederverwendbarer Objekte zu erstellen. Die Verwendung von Speicher für Objekte mit beschränkter Lebensdauer ist eine Alternative.
- Objekte im Speicher für Objekte mit unbeschränkter Lebensdauer können nicht auf Objekte im Speicher für Objekte mit beschränkter Lebensdauer verweisen. Wenn einem Feld eines Objekts im Speicher für Objekte mit unbeschränkter Lebensdauer ein Objekt aus dem Speicher für Objekte mit beschränkter Lebensdauer zugeordnet wird, wird eine Ausnahmebedingung IllegalAssignmentError ausgelöst.

Speicher für Objekte mit beschränkter Lebensdauer

Der Speicher für Objekte mit beschränkter Lebensdauer kann als Anfangsspeicher eines planbaren Objekts verwendet werden oder kann ein planbares Objekt aufnehmen. Wenn nicht mehr auf den Bereich verwiesen wird, können alle darin befindlichen Objekte gelöscht werden. In einem Speicherbereich für Objekte mit beschränkter Lebensdauer ausgeführte planbare Objekte führen alle Objektzuordnungen von diesem Bereich aus. Wenn ein Speicher für Objekte mit beschränkter Lebensdauer nicht verwendet wird, werden die darin befindlichen Objekte abgeschlossen, und der Speicher wird konsolidiert, um den Bereich für Wiederverwendung vorzubereiten. Wenn der Speicherbereich für Objekte mit beschränkter Lebensdauer nicht mehr für planbare Objekte verfügbar ist, wird der Speicher für andere Verwendungen konsolidiert.

Der durch eine ScopedMemory-Instanz beschriebene Hauptspeicherbereich ist im Java-Heapspeicher nicht vorhanden und für ihn wird keine Garbage-Collection durchgeführt. Sie können ein ScopedMemory-Objekt problemlos als Anfangsspeicherbereich verwenden, der einem NoHeapRealtimeThread zugeordnet ist, oder mit der Methode ScopedMemory.enter in einem NoHeapRealtimeThread in den Hauptspeicherbereich eintreten.

Physischer Hauptspeicher

Verwenden Sie den physischen Hauptspeicher, wenn die Merkmale des Speichers wichtig sind, z. B. wenn er nicht auslagerbar oder nicht flüchtig ist.

Lineares Zeitzuordnungsschema (LTMemory)

LTMemory stellt einen Hauptspeicherbereich dar, dem vom System lineare Zeitzuordnung garantiert wird, wenn die Speicherbelegung des Hauptspeicherbereichs kleiner als die Anfangsgröße des Hauptspeicherbereichs ist. Die Ausführung für die Zuordnung darf variieren, wenn sich die Speicherbelegung zwischen der Anfangsgröße und Maximalgröße für den Bereich befindet. Außerdem braucht das zugrunde liegende System nicht zu garantieren, dass Speicher zwischen der Anfangs- und Maximalgröße immer verfügbar ist.

Variables Zeitzuordnungsschema (VTMemory)
VTMemory ähnelt LTMemory mit der Ausnahme, dass die Ausführung einer Zuordnung von einem **VTMemory**-Bereich nicht in Linearzeit abgeschlossen werden muss.

Heapspeicher
Objekte im Heapspeicher können nicht auf Objekte im Speicher für Objekte mit beschränkter Lebensdauer verweisen. Wenn einem Feld eines Objekts im Heapspeicher ein Objekt aus dem Speicher für Objekte mit beschränkter Lebensdauer zugeordnet wird, wird eine Ausnahmebedingung IllegalAssignmentError ausgelöst.

Synchronisation und gemeinsame Nutzung von Ressourcen

In einem Echtzeitsystem, in dem mindestens drei mit einander synchronisierte Threads mit unterschiedlichen Prioritäten ausgeführt werden, kann es gelegentlich zu einer Bedingung kommen, die als Prioritätsumkehrung bezeichnet wird. Dabei wird die Ausführung eines Threads mit einer höheren Priorität von einem Thread mit einer niedrigeren Priorität einen längeren Zeitraum lang geblockt. WebSphere Real Time for RT Linux verwendet ein Schema, das als Prioritätsübernahme bezeichnet wird, um diese Bedingung zu vermeiden.

Wenn die Ausführung einer Task mit einer höheren Priorität von einer Task mit einer niedrigeren Priorität geblockt wird, wird die niedrigere Priorität vorübergehend aufgewertet, um mit der höheren Priorität übereinzustimmen, bis die Task mit der höheren Priorität nicht mehr geblockt ist.

Periodische und aperiodische Parameter

Echtzeitthreads verfügen über eine Anzahl Freigabeparameter, die festlegen, wie häufig ein planbares Objekt freigegeben wird. Periodische und aperiodische Parameter sind Beispiele für Freigabeparameter.

Periodische Parameter

Diese Klasse gilt für die planbaren Objekte, die in regelmäßigen Intervallen freigegeben werden.

AbsoluteTime
Wird in Millisekunden und Nanosekunden ausgedrückt.

RelativeTime
Ist die Länge eines gegebenen Ereignisses, die in Millisekunden und Nanosekunden ausgedrückt wird. Beispielsweise können Sie die absolute Zeit eines Ereignisstarts und -endes messen. Sie können dann die relative Zeit als Differenz zischen den beiden Messwerten berechnen.

Aperiodische Parameter

Diese Klasse wird von den planbaren Objekten verwendet, die in unregelmäßigen Intervallen freigegeben werden. Da ein zweites aperiodisches Ereignis vor dem Ende des ersten aperiodischen Ereignisses auftreten kann, können Sie die Länge der Warteschlange für ausstehende Anforderungen definieren.

Handhabung von asynchronen Ereignissen

Handler für asynchrone Ereignisse reagieren auf Ereignisse, die außerhalb eines Threads auftreten, z. B. die Eingabe von einer Anwendungsschnittstelle. In Echtzeitsystemen müssen diese Ereignisse innerhalb der Termine antworten, die Sie für Ihre Anwendung festlegen.

Asynchrone Ereignisse können Systeminterrupts sowie POSIX-Signalen zugeordnet und mit einem Zeitgeber verknüpft werden.

Wie Echtzeitthreads können Handlern für asynchrone Ereignisse eine Anzahl Parameter zugeordnet werden. Eine Liste dieser Parameter finden Sie in „Planbare Objekte und ihre Parameter" auf Seite 9.

Signalhandler

POSIXSignalHandler unterstützt die Signale SIGQUIT, SIGTERM und SIGABRT. Durch das Standardverhalten von SIGQUIT wird ein Java-Speicherauszug generiert. Die Generierung des Java-Speicherauszugs kollidiert nicht mit der Operation eines aktiven Programms, abgesehen von der CPU-Zeit und dem Lesen von sowie Schreiben in Dateien. Die Generierung eines Java-Speicherauszugs unterbricht das Programm, bis der Java-Speicherauszug abgeschlossen wurde. Die Anwendungsleistung ist nicht vorhersehbar, während Java-Speicherauszüge generiert werden.

Sie können die gesamte Kern- und Java-Speicherauszugsgenerierung bei einem Fehler über **-Xdump:none** unterdrücken.

Soll nur die Generierung eines Systemspeicherauszugs und Java-Speicherauszugs bei einem Signal SIGQUIT unterdrückt werden, geben Sie **-Xdump:java:none -Xdump:java:events=gpf+abort** an.

Die folgenden Signale können durch den POSIXSignalHandler-Mechanismus an Handler für asynchrone Ereignisse angehängt werden (Signalbeschreibungen so wie in `/usr/include/bits/signum.h` definiert):

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

Zurzeit werden keine anderen Signale unterstützt. Alle oben aufgeführten Signale sind asynchron. Das Anhängen an synchrone Signale (wie SIGILL und SIGSEGV) kann nicht unterstützt werden, weil sie einen Fehler in Ihrer Anwendung oder im JVM-Code, kein extern generiertes Ereignis angeben.

Anmerkung: Wird SIGQUIT von der JVM empfangen, weist dieses Signal die Java-Anwendung standardmäßig an, Speicherauszüge (z. B. einen Java-Speicherauszug) zu generieren. Das Signal wird außerdem an jeden angehängten Handler für asynchrone Ereignisse übergeben. Diese Übergabe kann verwirrendes oder unerwünschtes Verhalten verursachen. Sie können es über die Option **-Xdump:none:events=user** in der Java-Befehlszeile inaktivieren.

Erforderliche Dokumentation

WebSphere Real Time for RT Linux implementiert Real-Time Specification for Java (RTSJ).

WebSphere Real Time for RT Linux Version 2.0 wurde als RTSJ 1.0.2-kompatibel mit RTSJ Technology Compatibility Kit Version 3.0.13 FCS zertifiziert und ist mit Java Compatibility Kit (JCK) für Version 6.0 kompatibel.

Unterstützte Funktionen

Die folgenden Funktionen werden unterstützt:

- Durchsetzung der Zuordnungsrate für Heapspeicherzuordnung zum Begrenzen der Rate, mit der ein planbares Objekt Objekte im Heapspeicher erstellt.

Nicht unterstützte Funktionen

Die folgenden Funktionen werden nicht unterstützt:

- Protokoll der Prioritätsobergrenzenemulation. Beispielsweise darf PriorityCeilingEmulation nicht als Überwachungssteuerungsrichtlinie verwendet werden.
- Atomare Zugriffsunterstützung, außer wo für Übereinstimmung mit der Spezifikation erforderlich.
- Für Anwendungen sind außer dem Basisprioritätsscheduler keine anderen Scheduler verfügbar.
- Aufwandsdurchsetzung.

Erforderliche Dokumentation für Real-Time Specification for Java

In diesem Abschnitt wird eine übersetzte Version des Abschnitts zur erforderlichen Dokumentation von Real-Time Specification for Java (RTSJ) zitiert. Bei Abweichungen von der RTSJ-Standardimplementierung werden entsprechende Hinweise gegeben.

1. **Der Durchführbarkeitstestalgorithmus ist der Standard.**

   „Wenn der Durchführbarkeitstestalgorithmus nicht der Standard ist, dokumentieren Sie den Durchführbarkeitstestalgorithmus."
2. **Nur der Basisprioritätsscheduler ist für Anwendungen verfügbar.**

   „Wenn andere Scheduler als der Basisprioritätsscheduler für Anwendungen verfügbar sind, dokumentieren Sie das Verhalten des Schedulers und seine Interaktion mit jedem anderen Scheduler wo wie im Kapitel zu Planung ausgeführt. Dokumentieren Sie auch die Liste von Klassen, die planbare Objekte für den Scheduler darstellen, außer diese Liste entspricht der Liste planbarer Objekte für den Basisscheduler."
3. **Ein planbares Objekt, das durch ein planbares Objekt mit einer höheren Priorität zurückgestellt wird, wird aufgrund seiner Priorität an den Anfang der Warteschlange gestellt.**

   „Ein planbares Objekt, das durch ein planbares Objekt mit einer höheren Priorität zurückgestellt wird, wird aufgrund seiner Priorität an eine Position in der Warteschlange gestellt, die durch die Implementierung bestimmt wird. Wenn das zurückgestellte planbare Objekt nicht an den Anfang der entsprechenden Warteschlange gestellt wird, muss die Implementierung den für diese Platzierung verwendeten Algorithmus dokumentieren. Die Platzierung an den Anfang der Warteschlange kann ein einer zukünftigen Version dieser Spezifikation erforderlich sein."
4. **Aufwandsdurchsetzung wird nicht unterstützt.**

   „Wenn die Implementierung die Aufwandsdurchsetzung unterstützt, muss die Implementierung die Granularität dokumentieren, mit der die aktuelle CPU-Belegung aktualisiert wird."

5. **Einfache sequenzielle Zuordnung wird nicht unterstützt.**

   „Die durch einen Filter des physischen Hauptspeichers implementierte Speicherzuordnung muss dokumentiert werden, außer es handelt sich um eine einfache sequenzielle Zuordnung von zusammenhängenden Byte."

6. **Mit WebSphere Real Time for RT Linux wurden keine Unterklassen für den Metronom-Garbage-Collector geliefert.**

   „Die Implementierung muss das Verhalten von Unterklassen des Garbage-Collectors vollständig dokumentieren."

7. **Mit WebSphere Real Time for RT Linux wurden keine MonitorControl-Unterklassen geliefert.**

   „Eine Implementierung, die in dieser Spezifikation nicht erläuterte MonitorControl-Unterklassen liefert, muss ihre Auswirkungen dokumentieren, vor allem hinsichtlich der Steuerung der Prioritätsumkehrung und der Frage, welche Scheduler die neue Richtlinie nicht unterstützen (sofern zutreffend)."

8. **Einem planbaren Objekt mit einem Monitor, der von einem planbaren Objekt mit einer höheren Priorität benötigt wird, wird solange die höhere Priorität zugeordnet, bis der Monitor freigegeben wird. Ist das planbare Objekt an diesem Punkt nicht mehr ausführbar (d. h., es muss Arbeit mit einer höheren Priorität ausführen), wird es aufgrund seiner ursprünglichen (nicht aufgewerteten) Priorität an das Ende der Warteschlange gestellt, wenn die Ausführung auf einem Kernel vor SUSE Linux Enterprise Real Time 10 SP2 Update Kernelversion 2.6.22.19-0.16 und Red Hat Enterprise Linux 5.1 MRG 2.6.24.7-73 Errata 1 stattfindet. Kernel dieser oder späterer Versionen stellen das planbare Objekt an den Anfang der Warteschlange.**

   „Wenn das planbare Objekt die aufgewertete Priorität aufgrund eines Algorithmus zur Vermeidung der Prioritätsumkehrung verliert und nicht an den Anfang der neuen Warteschlange gestellt wird, muss die Implementierung das Warteschlangensteuerungsverhalten dokumentieren."

9. **Der Basisscheduler ist der einzige mit WebSphere Real Time for RT Linux gelieferte Scheduler.**

   „Bei jedem verfügbaren Scheduler außer dem Basisscheduler muss eine Implementierung dokumentieren, wie sich die Semantik der Synchronisation von den Regeln unterscheidet, die für die Standardinstanz PriorityInheritance definiert sind. Sie muss das Verhalten des neuen Schedulers mit der Prioritätsübernahme (und bei Unterstützung mit dem Protokoll der Prioritätsobergrenzenemulation) dokumentieren, das der Semantik für den Basisprioritätsscheduler entspricht, die im Kapitel zur Synchronisation beschrieben wird."

10. **Die schlechteste Zeit von der Auslösung eines Ereignisses bis zur Planung eines zugeordneten gebundenen Ereignishandlers beträgt durchschnittlich 40µs und überschreitet nicht 100µs, vorausgesetzt, es sind keine konkurrierenden planbaren Objekte bzw. es ist keine Systemaktivität mit mindestens gleicher Priorität vorhanden, und vorausgesetzt, die Garbage-Collection stört nicht. Wenn das planbare Objekt, das die Auslösermethode steuert, das AsyncEvent-Objekt oder der Handler auf den Heapspeicher verweist, ist der potenzielle Einfluss der Garbage-Collection so wie in (A) dokumentiert. Hierbei wird davon ausgegangen, dass der Code interpretiert wird und dass ein einzelner (gebundener) Handler für das Ereignis konfiguriert ist.**

    „Das schlechteste Antwortintervall zwischen dem Auslösen von AsyncEvent aufgrund eines gebundenen Ereignisses zur Freigabe eines zugeordneten AsyncEventHandler (angenommen, es sind keine planbaren Objekte mit höherer Priorität ausführbar) muss für die Referenzarchitektur dokumentiert werden."

11. **Das schlechteste Intervall zwischen dem Auslösen von AsynchronouslyInterruptedException in einem ATC-aktivierten Thread und der ersten Übergabe dieser Ausnahmebedingung beträgt durchschnittlich 35µs und überschreitet nicht 160µs, vorausgesetzt, es sind keine konkurrierenden planbaren Objekte bzw. es ist keine Systemaktivität mit mindestens gleicher Priorität vorhanden, und vorausgesetzt, die Garbage-Collection stört nicht. ATC-aktiviert bedeutet in diesem Fall, dass der Thread in einer AI-aktivierten Methode in einem Bereich ausgeführt wird, der nicht ATC-verzögert ist, und dass diese Bedingungen bis zur Übergabe der Ausnahmebedingung wahr bleiben. Der potenzielle Einfluss der Garbage-Collection ist so wie in (A) dokumentiert. Wenn sich der Zielthread im nativen Code befindet, ist die Verzögerung möglicherweise unbegrenzt. Hierbei wird davon ausgegangen, dass der Code interpretiert wird.**

    „Das Intervall zwischen dem Auslösen von AsynchronouslyInterruptedException in einem ATC-aktivierten Thread und der ersten Übergabe dieser Ausnahmebedingung (vorausgesetzt, es sind keine planbaren Objekte mit höherer Priorität ausführbar) muss für die Referenzarchitektur dokumentiert werden."

12. **Nicht zutreffend. Siehe Antwort 4.**

    „Wird die Aufwandsdurchsetzung unterstützt und ordnet die Implementierung den Aufwand bei der Ausführung von Finalizern für Objekte im Speicher für Objekte mit beschränkter Lebensdauer einem anderen planbaren Objekt zu als dem Objekt, das den Bereichsreferenzzähler durch Verlassen des Bereichs auf null setzt, müssen die Regeln für das Zuordnen des Aufwands dokumentiert werden."

13. **Die Standardimplementierung von RealtimeSecurity wurde nicht geändert.**

    „Wenn die Implementierung von RealtimeSecurity restriktiver ist als die erforderliche Implementierung oder über Laufzeitkonfigurationsoptionen verfügt, müssen diese Features dokumentiert werden."

14. **Die Finalizer für Objekte in einem Speicherbereich für Objekte mit beschränkter Lebensdauer werden vom letzten Thread ausgeführt, der auf diesen Bereich verweist, d. h., sie werden ausgeführt, wenn der Thread den Referenzzähler von 1 auf 0 senkt. Der Aufwand für die Ausführung der Finalizer wird diesem Thread zugeordnet.**

    „Eine Implementierung kann Finalizer für Objekte im Speicher für Objekte mit beschränkter Lebensdauer ausführen, bevor der Bereich wieder aktiviert wird und bevor sie von einem Aufruf von getReferenceCount() für diesen Bereich zurückkehrt. Sie muss jedoch dokumentieren, wann sie diese Finalizer ausführt."

15. **Die Auflösung ist nicht festlegbar.**

    „Die Dokumentation muss für jeden unterstützten Taktgeber angeben, ob die Auflösung festlegbar ist. Ist dies der Fall, muss die Dokumentation die unterstützten Werte angeben."

16. **Es gibt keine anderen Taktgeber als den mit WebSphere Real Time for RT Linux gelieferten Taktgeber.**

    „Wenn eine Implementierung andere Taktgeber als den erforderlichen Taktgeber enthält, muss die Dokumentation angeben, in welchem Kontext diese Taktgeber verwendet werden können."

Anmerkung:

A Die Referenzarchitektur für die Tests ist LS20, 4-Way, 2 GHz mit 1 MB Cache und 4 GB Hauptspeicher.

B Die Garbage-Collection kann jederzeit in einem Thread, der dem Heapspeicher zugeordnet ist, eine Verzögerung verursachen. Der Collector kann in einem von zwei Basismodi fungieren, die das Verhalten regeln, wenn kein Heapspeicher mehr verfügbar ist. Wenn der Collector so festgelegt ist, dass er unter diesen Umständen unverzüglich OutOfMemoryError auslöst, liegt die schlechteste Garbage-Collection-Verzögerung gewöhnlich unter 1 ms.

Die Verzögerung kann zurzeit unter einigen Umständen höher sein, z. B. wenn viele Threads mit tief verschachtelten Stacks oder eine hohe Anzahl großer Bereiche vorliegen. Wenn der Collector so festgelegt ist, dass er eine synchrone GC durchführt, bevor er OutOfMemoryError auslöst, ist die potenzielle Garbage-Collection-Verzögerung mit der Anzahl Liveobjekte im Heapspeicher und der Anzahl Objekte in anderen Hauptspeicherbereichen verbunden. Unter diesen Umständen wird die Verzögerung als unbegrenzt eingestuft, weil sie für typische Größen des Heapspeichers viele Sekunden betragen kann.

Kapitel 3. Planung

Lesen Sie diesen Abschnitt vor der Installation von WebSphere Real Time for RT Linux.

- „Migration"
-
- „Hardware- und Softwarevoraussetzungen"
- „Wichtige Faktoren" auf Seite 26

Migration

WebSphere Real Time for RT Linux wird in einer Linux-Umgebung ausgeführt, die für Echtzeitanwendungen modifiziert wurde. Sie können Standard-Java-Anwendungen in einer Echtzeitumgebung verwenden. Sie können Ihre Anwendungen auch modifizieren, um die Funktionen von WebSphere Real Time zu nutzen.

Systemmigration

Befolgen Sie die vom Linux-Unterstützungsteam gelieferten Anweisungen.

Hardware- und Softwarevoraussetzungen

Prüfen Sie anhand der folgenden Liste die Hardware, das Betriebssystem und die Java-Java-Umgebung, die bzw. das für WebSphere Real Time for RT Linux unterstützt wird.

Hardware

Zertifizierte WebSphere Real Time for RT Linux-Hardwarekonfigurationen sind Multiprozessorvarianten der folgenden Systeme:

- IBM BladeCenter LS20 (Typen 8850-76U, 8850-55U, 7971, 7972)
- IBM eServer xSeries 326m (Typen 7969-65U, 7969-85U, 7984-52U, 7984-6AU)
- IBM BladeCenter LS21 (Typ 7971-6AU)
- IBM BladeCenter HS21 XM Dual Quad Core (Typ 7995)

Bei IBM Systemen mit HT darf HT nicht aktiviert sein, um für WebSphere Real Time for RT Linux zertifiziert zu bleiben.

WebSphere Real Time for RT Linux wird außerdem auf Hardware unterstützt, die ein unterstütztes Betriebssystem ausführt und die folgenden Merkmale aufweist:

- Mindestens 512 MB physischer Hauptspeicher
- Mindestens Intel Pentium 4-, AMD Opteron- oder Intel Atom-Prozessor

IBM macht für Systeme, die keine zertifizierte Hardwarekonfigurationen sind, keine Aussagen zur Leistung. Leistungsaspekte für zertifizierte Hardwarekonfigurationen werden in Kapitel 7, „Leistung", auf Seite 101 beschrieben.

Stellen Sie bei Systemen mit HT-Unterstützung sicher, dass HT nicht aktiviert ist, um bei der Verwendung von WebSphere Real Time for RT Linux Leistungseinbußen zu vermeiden.

Betriebssystem

- Red Hat Enterprise Linux 5.3 MRG. Weitere Informationen hierzu finden Sie in „Real Time Linux-Umgebung installieren" auf Seite 29.
- SUSE Linux Enterprise Real Time (SLERT) 10. Weitere Informationen hierzu finden Sie in „Real Time Linux-Umgebung installieren" auf Seite 29.

Wichtige Faktoren

Sie müssen bei der Verwendung von WebSphere Real Time for RT Linux eine Anzahl Faktoren beachten.

- Sofern möglich, führen Sie nicht mehrere Echtzeit-JVMs auf demselben System aus. Der Grund hierfür ist, dass Sie dann über mehrere Garbage-Collector verfügen würden. Jede einzelne JVM kennt nicht die Hauptspeicherbereiche der anderen JVMs. Eine Auswirkung ist, dass GC-Zyklen und -Pausezeiten JVM-übergreifend nicht koordiniert werden können. Dies bedeutet, dass sich eine JVM negativ auf die GC-Leistung einer anderen JVM auswirken kann. Wenn Sie mehrere JVMs verwenden müssen, stellen Sie sicher, dass jede JVM mithilfe des Befehls **taskset** an eine bestimmte Untergruppe von Prozessoren gebunden wurde.
- Sie können die Optionen **-Xdebug** und **-Xnojit** nicht mit dem Code verwenden, der über den Ahead-of-time-Compiler (AOT-Compiler) vorkompiliert wurde. Der Grund hierfür ist, dass **-Xdebug** den Code anders kompiliert als der AOT-Compiler und nicht unterstützt wird.

  Verwenden Sie für das Debugging Ihres Codes interpretierten oder mit JIT kompilierten Code.
- Wenn Sie den Java-Quellcode, der das Paket javax.realtime verwendet, über die Schnittstelle com.sun.tools.javac.Main kompilieren, müssen Sie sicherstellen, dass `sdk/jre/lib/i386/realtime/jclSC170/realtime.jar` in den Klassenpfad eingeschlossen ist. Ein allgemeines Beispiel für diesen Typ von Kompilierung ist eine Ant-Kompilierung.
- Das optionale Paket JavaComm kann in WebSphere Real Time for RT Linux installiert werden und Zugriff ist über die Echtzeit-JVM und die Nicht-Echtzeit-JVM möglich. Weitere Informationen zur Installation und Konfiguration finden Sie in http://publib.boulder.ibm.com/infocenter/java7sdk/v7r0/topic/com.ibm.java.lnx.70.doc/user/jcommchapter.html. Die Echtzeit-JVM in WRT unterstützt die JavaComm-API für die Verwendung mit regulären Java-Threads. Es gibt jedoch keine Garantie hinsichtlich Bestimmungsfunktion oder Leistung in Echtzeit für den Zugriff auf externe Einheiten über JavaComm. Verwenden Sie JavaComm nicht mit No-Heap-Echtzeitthreads und Echtzeitthreads oder wenn Echtzeitverhalten erforderlich ist.
- Die gemeinsam genutzten Caches, die von früheren Releases von WebSphere Real Time for RT Linux zum Speichern von vorkompiliertem Code und Klassen verwendet wurden, sind nicht mit den Caches kompatibel, die von diesem Release von WebSphere Real Time for RT Linux verwendet werden. Sie müssen den Inhalt der früheren Caches neu generieren.
- Bei Verwendung von Caches für gemeinsam genutzte Klassen darf der Cachename 53 Zeichen nicht überschreiten.
- Der Befehl **ps** schneidet Java-Threadnamen ab.

  Der Befehl **ps** ist auf 15 Zeichen beschränkt. Wenn Sie für einen Threadnamen mehr als 15 Zeichen festlegen, wird der Name vom Befehl **ps** abgeschnitten.
- WebSphere Real Time for RT Linux unterstützt nicht NTLM-Authentifizierung.

Über NTLM wird der Zugriff auf einen Windows-Dienst authentifiziert. Die Authentifizierung über NTLM wird nur auf der Windows-Plattform unterstützt. Dies bedeutet, dass WebSphere Real Time for RT Linux die NTLM-Authentifizierung nicht unterstützt.

Kapitel 4. WebSphere Real Time for RT Linux installieren

Führen Sie folgende Schritte aus, um das Produkt zu installieren.

Installationsdateien

Sie benötigen die folgenden Installationsdateien.

IBM WebSphere Real Time for RT Linux wird in zwei InstallAnywhere-Pakettypen geliefert.

Installierbare Pakete

Installierbare Pakete konfigurieren Ihr System. Beispielsweise können die Programme Umgebungsvariablen festlegen.

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

Archivierungspakete

Diese Pakete extrahieren die Dateien auf Ihr System, führen jedoch keine Konfiguration aus.

- `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-Umgebung installieren

Sie müssen Real Time Linux installieren, bevor Sie WebSphere Real Time for RT Linux installieren können.

Vorbereitende Schritte

Sie müssen eine 64-Bit-Version von Real Time Linux installieren, bevor Sie WebSphere Real Time for RT Linux installieren.

Red Hat Enterprise Linux 5.3 MRG

- Weitere Informationen zum Installieren der Echtzeitkomponente von Red Hat Enterprise Linux 5.3 MRG finden Sie in den Installationsanweisungen für RT-Linux RHEL 5.3 MRG 1.1.2: 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

- Weitere Informationen zum Installieren von SUSE Linux Enterprise Real Time 10 finden Sie in http://www.novell.com/products/realtime/eval.html.

Werden sehr viele Dateideskriptoren zum Laden unterschiedlicher Klasseninstanzen genutzt, wird möglicherweise die Fehlernachricht "`java.util.zip.ZipException: error in opening zip file`" oder eine andere Ausnahmebedingung des Typs IOException angezeigt, in der Ihnen mitgeteilt wird, dass eine Datei nicht geöffnet werden konnte. Die Lösung besteht darin, die Bedingungen für Dateideskriptor mit dem Befehl **ulimit** zu verbessern. Mit folgendem Befehl können Sie den aktuellen Grenzwert für offene Dateien ermitteln:

```
ulimit -a
```

Wenn Sie eine höhere Anzahl an offenen Dateien zulassen möchten, geben Sie folgenden Befehl ein:

```
ulimit -n 8196
```

Installation über ein InstallAnywhere-Paket durchführen

Diese Pakete stellen ein interaktives Programm zur Verfügung, das Sie durch die Installationsoptionen führt. Sie können das Programm über eine grafische Benutzerschnittstelle oder über eine Systemkonsole ausführen.

Vorbereitende Schritte

Die folgenden beiden gemeinsam genutzten Bibliotheken müssen auf Ihrem System vorhanden sein:

- GNU-C-Bibliothek Version 2.3 (glibc)
- `libstdc++.so.5`

Ist die gemeinsam genutzte Bibliothek `libstdc++.so.5` nicht vorhanden, wird bei der Installation möglicherweise ein Java-Kernspeicherauszug mit den folgenden Fehlern ausgegeben:

```
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
```

Wenn Sie ein installierbares Paket installieren, muss das Tool **rpm-build** auf Ihrem System installiert sein. Andernfalls kann das Installationsprogramm das neue Paket nicht in der RPM-Datenbank registrieren. Durch Eingabe des folgenden Befehls können Sie ermitteln, ob das Tool **rpm-build** installiert ist:

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

Informationen zu diesem Vorgang

Die InstallAnywhere-Pakete haben die Dateierweiterung `.bin`.

Es gibt zwei Typen von Paketen:

Installierbares Paket
: Im Rahmen der Installation dieser Pakete erfolgt auch eine Konfiguration Ihres Systems, zum Beispiel durch Festlegen von Umgebungsvariablen.

Archivierungspaket
Im Rahmen der Installation dieser Pakete werden die Dateien auf Ihr System extrahiert, es wird jedoch keine Konfiguration durchgeführt.

Vorgehensweise

- Soll das Paket interaktiv installiert werden, führen Sie eine beaufsichtigte Installation durch.
- Soll das Paket ohne weitere Benutzerinteraktion installiert werden, führen Sie eine unbeaufsichtigte Installation durch. Die Auswahl dieser Option kann sinnvoll sein, wenn mehrere Systeme installiert werden sollen.
- Wenn der Installationsprozess abgeschlossen ist, führen Sie die Konfigurationsschritte in diesem Abschnitt wie das Festlegen der Umgebungsvariablen für Pfad und Klassenpfad aus.

Ergebnisse

Das Produkt wird installiert.

Anmerkung: Unterbrechen Sie den Installationsprozess nicht (z. B. durch Drücken von Strg+C). Wenn Sie den Prozess unterbrechen, müssen Sie das Produkt möglicherweise erneut installieren. Weitere Informationen finden Sie in „Unterbrochene Installation" auf Seite 33.

Wenn Sie ein installierbares Paket verwenden, werden möglicherweise Nachrichten ausgegeben, die darauf hinweisen, dass ein Problem aufgetreten ist. Bei der Installation der Archivierungspakete werden keine Nachrichten generiert. Einige der Nachrichten, die bei Verwendung eines installierbaren Pakets unter Umständen ausgegeben werden, sind in der folgenden Liste aufgeführt:

`Das Installationsprogramm kann Ihre Konfiguration nicht ausführen und wird jetzt beendet.`
Diese Fehlernachricht wird ausgegeben, wenn die Benutzer-ID nicht zur Ausführung des Installationsprozesses berechtigt ist. Da das Installationsprogramm nicht fortgesetzt werden kann, wird es beendet. Starten Sie die Installation erneut und verwenden Sie dabei eine Benutzer-ID, die über Rootberechtigung verfügt, um dieses Problem zu beheben.

`An RPM package is already installed. Uninstall the package before proceeding.`
Diese Nachricht weist darauf hin, dass bereits ein RPM-Paket installiert ist. Da das Installationsprogramm nicht fortgesetzt werden kann, wird es beendet. Deinstallieren Sie zur Behebung des Problems das RPM-Paket, bevor Sie fortfahren.

Beaufsichtigte Installation durchführen

Installieren Sie das Produkt interaktiv aus einem InstallAnywhere-Paket.

Vorbereitende Schritte

Überprüfen Sie vor Beginn des Installationsprozesses, ob die folgenden Bedingungen gegeben sind:

- Wurde WebSphere Real Time for RT Linux zuvor aus einem RPM-Paket installiert, müssen Sie dieses Paket erst deinstallieren, bevor sie fortfahren.
- Sie müssen über eine Benutzer-ID mit Rootberechtigung verfügen.

Vorgehensweise

1. Laden Sie die Installationspaketdatei in ein temporäres Verzeichnis herunter.
2. Wechseln Sie in das temporäre Verzeichnis.
3. Starten Sie den Installationsprozess, indem Sie `./package` an einer Shelleingabeaufforderung eingeben. Dabei steht *Paket* für den Namen des Pakets, das installiert wird.
4. Wählen Sie eine Sprache aus der im Fenster des Installationsprogramms angezeigten Liste aus und klicken Sie dann auf **Next**. Die Liste der verfügbaren Sprachen basiert auf der Ländereinstellung Ihres Systems.
5. Lesen Sie die Lizenzvereinbarung. Führen Sie dabei mithilfe der Bildlaufleiste einen Bildlauf bis zum Ende der Lizenzinformationen durch. Wenn Sie mit der Installation fortfahren wollen, müssen Sie die Bedingungen der Lizenzvereinbarung akzeptieren. Klicken Sie zum Akzeptieren der Bedingungen auf das Optionsfeld und klicken Sie dann auf **OK**.

   Anmerkung: Das Optionsfeld zum Akzeptieren der Lizenzvereinbarung können Sie erst auswählen, wenn Sie bis zum Ende der Lizenzvereinbarung gelesen haben.
6. Sie werden aufgefordert, das Zielverzeichnis für die Installation auszuwählen. Wenn die Installation nicht im Standardverzeichnis durchgeführt werden soll, klicken Sie auf **Choose**, um über das Browserfenster ein alternatives Verzeichnis auszuwählen. Klicken Sie nach der Auswahl des Installationsverzeichnisses auf **Next**, um fortzufahren.
7. Sie werden aufgefordert, die von Ihnen gewählten Einstellungen zu prüfen. Wenn Sie die gewählten Einstellungen ändern wollen, klicken Sie auf **Previous**. Sind die von Ihnen gewählten Einstellungen richtig, klicken Sie auf **Install**, um mit der Installation fortzufahren.
8. Wenn der Installationsprozess abgeschlossen ist, klicken Sie auf **Done**, um den Vorgang zu beenden.

Unbeaufsichtigte Installation durchführen

Wenn Sie mehrere System installieren müssen und bereits wissen, welche Installationsoptionen Sie verwenden wollen, haben Sie auch die Option, einen unbeaufsichtigten Installationsprozess durchzuführen. Dazu führen Sie zunächst *eine* Installation über den beaufsichtigten Installationsprozess durch und verwenden anschließend die daraus resultierende Antwortdatei, um weitere Installationen ohne Benutzerinteraktion durchzuführen.

Vorgehensweise

1. Erstellen Sie eine Antwortdatei, indem Sie eine beaufsichtigte Installation durchführen. Verwenden Sie eine der folgenden Optionen:
   - Verwenden Sie die grafische Benutzerschnittstelle (GUI) und geben Sie an, dass das Installationsprogramm eine Antwortdatei erstellen soll. Die Antwortdatei hat den Namen `installer.properties` und wird im Installationsverzeichnis erstellt.
   - Hängen Sie unter Verwendung der Befehlszeile die Option -r an den Befehl für die beaufsichtigte Installation an und geben Sie dabei den vollständigen Pfad zur Antwortdatei an. Beispiel:

     ```
     ./Paket -r /Pfad/installer.properties
     ```

   Beispielinhalt einer Antwortdatei:

   ```
   INSTALLER_UI=silent
   USER_INSTALL_DIR=/Mein_Verzeichnis
   ```

In diesem Beispiel steht */Mein_Verzeichnis* für das von Ihnen gewählte Zielinstallationsverzeichnis für die Installation.

2. Optional: Ändern Sie in der Antwortdatei nach Bedarf die Optionen.

   Anmerkung: Bei Archivierungspaketen tritt das folgende bekannte Problem auf: Bei Installationen, die mithilfe einer Antwortdatei durchgeführt werden, wird auch nach einer Änderung des Verzeichnisses in der Antwortdatei weiterhin das Standardverzeichnis verwendet. Ist im Standardverzeichnis eine vorherige Installation vorhanden, wird sie überschrieben.

   Wenn Sie mehrere Antwortdateien erstellen, jede mit jeweils anderen Installationsoptionen, geben Sie für jede Antwortdatei einen eindeutigen Namen im Format *MeineDatei*.properties an.

3. Optional: Generieren Sie eine Protokolldatei. Da Sie die Installation im Hintergrund durchführen, werden am Ende des Installationsprozesses keine Statusnachrichten angezeigt. Führen Sie die folgenden Schritte durch, damit eine Protokolldatei generiert wird, in der der Installationsstatus angegeben ist:

   a. Legen Sie mithilfe des folgenden Befehls die erforderlichen Systemeigenschaften fest.

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

   b. Legen Sie die folgende Umgebungsvariable fest, um die Protokollausgabe an die Konsole zu senden.

      ```
      export LAX_DEBUG=1
      ```

4. Starten Sie eine unbeaufsichtigte Installation, indem Sie das Installationsprogramm für das Paket mit der Option **-i** (Installation im Hintergrund) und der Option **-f** (Angabe der Antwortdatei) festlegen. Beispiel:

   ```
   ./Paket -i silent -f /Pfad/installer.properties 1>Konsole.txt 2>&1
   ./Paket -i silent -f /Pfad/MeineDatei.properties 1>Konsole.txt 2>&1
   ```

   Sie können einen vollständig qualifizierten Pfad oder einen relativen Pfad zu der Eigenschaftendatei verwenden. In diesen Beispielen leitet die Zeichenfolge `1>Konsole.txt 2>&1` die Informationen zum Installationsprozess von den Datenströmen stderr und stdout an die Protokolldatei `Konsole.txt` im aktuellen Verzeichnis um. Prüfen Sie diese Protokolldatei, wenn Sie denken, dass bei der Installation ein Problem aufgetreten ist.

   Anmerkung: Wenn Ihr Installationsverzeichnis mehrere Antwortdateien enthält, wird die Standardantwortdatei `installer.properties` verwendet.

Unterbrochene Installation

Wird das Installationsprogramm für das Paket während der Installation unerwartet gestoppt (z. B. durch Drücken von Strg+C), wird die Installation beschädigt und das Produkt kann nicht deinstalliert oder neu installiert werden. Wenn Sie versuchen, eine Deinstallation oder Neuinstallation durchzuführen, wird möglicherweise eine Nachricht ausgegeben, dass ein schwerwiegender Anwendungsfehler aufgetreten ist.

Informationen zu diesem Vorgang

Dieses Problem können Sie beheben, indem Sie Dateien löschen und wie in den folgenden Schritten beschrieben eine Neuinstallation durchführen.

Vorgehensweise

1. Löschen Sie die Registry-Datei `/var/.com.zerog.registry.xml`.

2. Löschen Sie das Verzeichnis, das die Installation enthält, sofern eines erstellt wurde. Beispielsweise opt/IBM/javawrt3/.
3. Führen Sie das Installationsprogramm erneut aus.

Bekannte Probleme und Einschränkungen

In Verbindung mit den InstallAnywhere-Paketen sind bekannte Probleme und Einschränkungen zu beachten.

- Ist die gemeinsam genutzte Bibliothek `libstdc++.so.5` nicht auf Ihrem System vorhanden, schlägt die Installation fehl und es wird ein Java-Kernspeicherauszug erstellt. Weitere Informationen finden Sie in „Installation über ein InstallAnywhere-Paket durchführen" auf Seite 30.
- Die grafische Benutzerschnittstelle des Installationspakets unterstützt das Sprachausgabeprogramm Orca nicht. Alternativ zu der grafischen Benutzerschnittstelle können Sie den Modus für unbeaufsichtigte Installationen verwenden.
- Wenn Sie `./Paket` nach einer Installation eingeben, um das Programm erneut zu starten, wird vom Programm die folgende Nachricht angezeigt:

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

  Wenn Sie die Eingabetaste drücken, um den Standardwert zu übernehmen, reagiert das Programm nicht. Geben Sie eine Zahl ein und drücken Sie dann die Eingabetaste.
- Wenn Sie das Paket installieren und dann versuchen, eine weitere Installation in einem anderen Modus durchzuführen (z. B. im Konsolen- oder Hintergrundsmodus), wird möglicherweise die folgende Fehlernachricht ausgegeben:

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

  Diese Nachricht wird in der Regel nicht ausgegeben, wenn nach einer Installation im GUI-Modus beim erneuten Ausführen des Installationsprogramms der Konsolenmodus verwendet wird. Tritt dieser Fehler auf, wenn Sie das Programm lediglich ausführen, um die Deinstallationsoption auszuwählen (nur bei installierbaren Paketen), verwenden Sie stattdessen den Befehl **`./_uninstall/uninstall`** wie in „WebSphere Real Time for RT Linux deinstallieren" auf Seite 37 beschrieben.

Nur installierbare Pakete

- Die InstallAnywhere-Pakete können nicht verwendet werden, um ein Upgrade für eine vorhandene Installation durchzuführen. Wenn Sie für WebSphere Real Time for RT Linux ein Upgrade durchführen wollen, müssen Sie zuerst alle vorherigen Versionen deinstallieren.
- Sie können nicht zwei verschiedene Instanzen derselben Version von WebSphere Real Time for RT Linux auf demselben System installieren, auch nicht, wenn Sie verschiedene Installationsverzeichnisse verwenden. Es kann z. B. nicht gleichzeitig WebSphere Real Time for RT Linux Version 3 im Verzeichnis `/previous` und eine Serviceaktualisierung von WebSphere Real Time for RT Linux im Verzeichnis `/current` installiert werden. Das Installationsprogramm überprüft die Versionsnummer. Findet das Programm ein vorhandenes Paket mit derselben Versionsnummer, werden Sie aufgefordert, das vorhandene Paket zu deinstallieren.
- Ist das Paket installiert und führen Sie das Installationsprogramm für das Paket erneut unter Verwendung der grafischen Benutzerschnittstelle aus, können Sie auswählen, dass das Paket deinstalliert werden soll.

Diese Deinstallationsoption ist im nicht überwachten Modus nicht verfügbar. Wenn Sie das Installationsprogramm für das Paket erneut im nicht überwachten Modus ausführen, wird das Programm ausgeführt, führt jedoch keine Aktionen aus.

Nur Archivierungspakete

- Wenn Sie das Installationsverzeichnis in einer Antwortdatei ändern und dann eine unbeaufsichtigte Installation unter Verwendung dieser Antwortdatei ausführen, ignoriert das Installationsprogramm das neue Installationsverzeichnis und verwendet stattdessen das Standardverzeichnis. Ist im Standardverzeichnis eine vorherige Installation vorhanden, wird sie überschrieben.

Pfad festlegen

Wenn Sie die Umgebungsvariable **PATH** festgelegt haben, können Sie eine Anwendung oder ein Programm durch die Eingabe des entsprechenden Namens an einer Shelleingabeaufforderung ausführen.

Informationen zu diesem Vorgang

Anmerkung: Alle vorhandenen ausführbaren Java-Programme in dem von Ihnen verwendeten Pfad werden überschrieben, wenn Sie die Umgebungsvariable **PATH** so wie in diesem Abschnitt beschrieben ändern.

Sie können den Pfad für ein Tool angeben, indem Sie den Pfad jedes Mal vor dem Toolnamen eingeben. Wenn z. B. das SDK in `opt/IBM/javawrt3/` installiert ist, können Sie die Datei *meineDatei.java* kompilieren, indem Sie Folgendes an einer Shelleingabeaufforderung eingeben:

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

Gehen Sie wie folgt vor, um die jeweilige Eingabe des vollständigen Pfads zu vermeiden:

1. Bearbeiten Sie die Shellstartdatei im Ausgangsverzeichnis (in der Regel **.bashrc**, je nach der von Ihnen verwendeten Shell), und fügen Sie der Umgebungsvariablen **PATH** die absoluten Pfade hinzu. Beispiel:

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

2. Melden Sie sich erneut an, oder führen Sie das aktualisierte Shell-Script aus, um die neue Einstellung der Umgebungsvariablen **PATH** zu aktivieren.
3. Kompilieren Sie die Datei mit dem Tool **javac**. Geben Sie beispielsweise zum Kompilieren der Datei *MeineDatei.java* an einer Shelleingabeaufforderung Folgendes ein:

   ```
   javac -Xrealtime meineDatei.java
   ```

   Über die Umgebungsvariable **PATH** kann Linux ausführbare Dateien wie **javac** und **java** sowie das Tool **javadoc** im aktuellen Verzeichnis finden. Geben Sie zum Anzeigen des aktuellen Werts Ihres Pfads Folgendes an einer Eingabeaufforderung ein:

   ```
   echo $PATH
   ```

Nächste Schritte

Mithilfe der Informationen in „Klassenpfad festlegen” auf Seite 36 können Sie ermitteln, ob Sie die Umgebungsvariable CLASSPATH festlegen müssen.

Klassenpfad festlegen

Die Umgebungsvariable **CLASSPATH** teilt den SDK-Tools (z. B. **java**, **javac** und **javadoc**) die Speicherposition der Java-Klassenbibliotheken mit.

Informationen zu diesem Vorgang

Legen Sie die Umgebungsvariable **CLASSPATH** nur explizit fest, wenn eine der folgenden Bedingungen zutrifft:

- Sie benötigen eine andere Bibliothek oder Klassendatei (ähnlich der, die Sie entwickeln), und diese befindet sich nicht im aktuellen Verzeichnis.
- Sie ändern die Speicherposition der Verzeichnisse `bin` und `lib`, und sie befinden sich nicht mehr im selben übergeordneten Verzeichnis.
- Sie wollen Anwendungen entwickeln oder ausführen, die unterschiedliche Laufzeitumgebungen auf demselben System verwenden.

Geben Sie Folgendes an einer Shelleingabeaufforderung ein, um den aktuellen Wert der Umgebungsvariable **CLASSPATH** anzuzeigen:

```
echo $CLASSPATH
```

Wenn Sie Anwendungen entwickeln und ausführen wollen, die unterschiedliche Laufzeitumgebungen verwenden, wie z. B. andere Versionen, die separat installiert wurden, müssen Sie **CLASSPATH** und **PATH** für jede Anwendung explizit festlegen. Wenn Sie mehrere Anwendungen gleichzeitig ausführen und unterschiedliche Laufzeitumgebungen verwenden, muss jede Anwendung in einem separaten Befehlsfenster in einer separaten Shell ausgeführt werden.

Wenn Sie nur jeweils eine Java-Version ausführen, können Sie mit einem Shell-Script zwischen den verschiedenen Laufzeitumgebungen umschalten.

Nächste Schritte

Mithilfe der Informationen in „Installation testen” können Sie prüfen, ob Ihre Installation erfolgreich ist.

Installation testen

Mit der Option **-version** können Sie prüfen, ob Ihre Installation erfolgreich ist.

Informationen zu diesem Vorgang

Die Java-Installation besteht aus einer Standard-JVM und einer Echtzeit-JVM.

Vorgehensweise

Führen Sie die folgenden Schritte aus, um Ihre Installation zu testen:

1. Geben Sie den folgenden Befehl an einer Shelleingabeaufforderung ein, um Versionsinformationen für die Standard-JVM anzuzeigen:

   ```
   java -version
   ```

   Dieser Befehl gibt bei erfolgreicher Ausführung die folgende Nachricht zurück:

   ```
   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
```

Wenn Sie die Standard-JVM anstatt der Echtzeit-JVM verwenden wollen, ziehen Sie die IBM Benutzerhandbücher für Java Version 7 unter Linux zu Rate.

Anmerkung: Diese Versionsinformationen sind korrekt, allerdings können neuere Datumsangaben als in diesem Beispiel angezeigt werden. Das Datumsformat ist JJJMMTT möglicherweise gefolgt von zusätzlichen Informationen zur jeweiligen Komponente.

2. Geben Sie den folgenden Befehl an einer Shelleingabeaufforderung ein, um Versionsinformationen für die Echtzeit-JVM anzuzeigen:

   ```
   java -Xrealtime -version
   ```

   Dieser Befehl gibt bei erfolgreicher Ausführung die folgende Nachricht zurück:

   ```
   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
   ```

   Anmerkung: Diese Versionsinformationen sind korrekt, allerdings können die Plattformarchitektur und Daten vom Beispiel abweichen. Das Datumsformat ist JJJMMTT möglicherweise gefolgt von zusätzlichen Informationen zur jeweiligen Komponente.

WebSphere Real Time for RT Linux deinstallieren

Welchen Prozess Sie zum Entfernen von WebSphere Real Time for RT Linux verwenden müssen, hängt davon ab, welcher Installationstyp verwendet wurde.

Vorbereitende Schritte

Bei installierbaren InstallAnywhere-Paketen ist eine Benutzer-ID mit Rootberechtigung erforderlich.

Informationen zu diesem Vorgang

Bei InstallAnywhere-Archivierungspaketen gibt es keinen Deinstallationsprozess. Wenn Sie ein Archivierungspaket von Ihrem System entfernen wollen, löschen Sie das beim Installieren des Pakets gewählte Zielverzeichnis. Bei installierbaren InstallAnywhere-Paketen deinstallieren Sie das Produkt mithilfe eines Befehls oder durch erneutes Ausführen des Installationsprogramms wie in den folgenden Schritten beschrieben.

Vorgehensweise

- Optional: Führen Sie die Deinstallation manuell mithilfe des Befehls **uninstall** durch.
  1. Wechseln Sie in das Verzeichnis, in dem sich die Installation befindet. Beispiel:

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

2. Starten Sie den Deinstallationsprozess durch Eingabe des folgenden Befehls:
   `./_uninstall/uninstall`

- Optional: Ist das Deinstallationsprogramm nicht leicht auffindbar, können Sie alternativ auch eine weitere beaufsichtigte Installation ausführen. Das Installationsprogramm erkennt das bereits installierte Produkt und gibt Ihnen die Gelegenheit, die vorherige Installation zu deinstallieren.

Kapitel 5. IBM WebSphere Real Time for RT Linux-Anwendungen ausführen

Wichtige Informationen für die Ausführung von Echtzeitanwendungen.

- „Für WebSphere Real Time for RT Linux kompilierten Code verwenden" auf Seite 40
- „No-Heap-Echtzeitthreads verwenden" auf Seite 60
- „Gemeinsame Nutzung von Klassendaten zwischen JVMs" auf Seite 70
- „Metronom-Garbage-Collector verwenden" auf Seite 72

Threadplanung und -zuteilung

Das Betriebssystem Linux unterstützt verschiedene Planungsrichtlinien. Die universelle Standard-Time-Sharing-Planungsrichtlinie ist SCHED_OTHER, die von den meisten Threads verwendet wird. SCHED_RR und SCHED_FIFO können von Threads in Echtzeitanwendungen verwendet werden. verwendet.

Der Kernel entscheidet, welcher ausführbare Thread vom Prozessor als nächster Thread ausgeführt wird. Der Kernel verwaltet eine Liste von ausführbaren Threads. Er sucht nach dem Thread mit der höchsten Priorität und wählt diesen Thread als nächsten auszuführenden Thread aus.

Threadprioritäten und -richtlinien können mit dem folgenden Befehl aufgelistet werden:

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

Dabei gilt für 'policy' Folgendes:

- `TS` ist SCHED_OTHER.
- `RR` ist SCHED_RR.
- `FF` ist SCHED_FIFO.
- Bei `-` wurde keine Richtlinie zurückgemeldet.

Die Ausgabe sieht wie folgendes Beispiel aus:

```
  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
```

Diese Ausgabe zeigt den Java-Prozess, die geltende Planungsrichtlinie, den Hauptthread mit der Priorität „-" (andere) und einige Echtzeitthreads mit Prioritäten zwischen 11 und 89 an.

Sie können die aktuelle Planungsrichtlinie mit **`sched_getscheduler`** oder dem im Beispiel gezeigten Befehl **`ps`** abfragen.

Weitere Informationen zu Prozessen finden Sie in „Allgemeine Debugging-Verfahren" auf Seite 106.

Prioritäten und Richtlinien von Java-Echtzeitthreads

Echtzeitthreads, d. h. als `java.realtime.RealtimeThread`-Objekte zugeordnete Threads, und Handler für asynchrone Ereignisse verwenden die Planungsrichtlinie SCHED_FIFO.

Die Threadplanung und -zuteilung von Java-Echtzeitthreads ist Teil von Real Time Specification for Java (RTSJ). Dieses Thema einschließlich Planungsrichtlinien und Prioritätshandhabung von Java-Echtzeitthreads wird im Abschnitt „Unterstützung für RTSJ" auf Seite 9 erläutert.

Für WebSphere Real Time for RT Linux kompilierten Code verwenden

IBM WebSphere Real Time for RT Linux unterstützt mehrere Codekompilierungsmodelle, die verschiedene Ebenen der Codeleistung und der Bestimmungsfunktion liefern.

Interpretierte Operation

Dies ist das einfachste Codekompilierungsmodell. Der Interpreter führt eine Java-Anwendung aus, verwendet die Codekompilierung allerdings nicht. Der Interpreter zeichnet sich durch eine gute Bestimmungsfunktion aus, seine Leistung ist jedoch sehr niedrig. Vermeiden Sie diese Betriebsart daher für Produktionssysteme.

Geben Sie die Option **`-Xint`** in der Java-Befehlszeile an, um die interpretierte Operation zu verwenden.

JIT-Kompilierung mit niedriger Priorität

Das Standardkompilierungsmodell in WebSphere Real Time for RT Linux verwendet einen JIT-Compiler, um die wichtigen Methoden einer Java-Anwendung während der Anwendungsausführung zu kompilieren. In diesem Modus funktioniert der JIT-Compiler ähnlich wie die Operation des JIT-Compilers in einer Nicht-Echtzeit-JVM. Der Unterschied ist, dass der JIT-Compiler von WebSphere Real Time for RT Linux auf einer niedrigeren Prioritätsebene ausgeführt wird als Echtzeitthreads. Die niedrigere Priorität bedeutet, dass der JIT-Compiler Systemressourcen verwendet, wenn die Anwendung keine Echtzeittasks auszuführen braucht. Hierdurch wirkt sich der JIT-Compiler nicht wesentlich auf die Leistung von Echtzeittasks aus.

Der JIT-Compiler verwendet zwei Threads für kompilierungsbezogene Aktivitäten: den Kompilierungsthread und den Sampler-Thread. Diese Threads werden mit einer niedrigeren Priorität ausgeführt als Echtzeittasks. Der Kompilierungsthread wird für die Anwendung asynchron ausgeführt. Dies bedeutet, dass ein Anwendungsthread nie auf den Kompilierungsthread wartet, um die Kompilierung einer Methode abzuschließen. Der Sampler-Thread sendet periodisch eine asynchrone Nachricht an die Anwendungsthreads, um die zurzeit aktive Methode für jeden Thread zu ermitteln. Die Verarbeitung der Nachricht im Anwendungsthread dauert nicht lange. Es werden keine Nachrichten gesendet, wenn der Sampling-Thread aufgrund von Echtzeittasks mit einer höheren Priorität nicht ausgeführt werden kann. Die Verwendung des JIT-Compilers wirkt sich ein wenig auf die Bestimmungsfunktion aus, dieser Kompilierungsmodus liefert jedoch die beste Leistung für viele Benutzer.

Informationen zur Ausführung einer Anwendung mit JIT mit niedriger Priorität finden Sie in „JIT-Compiler aktivieren" auf Seite 59.

Vorkompilierter AOT-Code

WebSphere Real Time for RT Linux kompiliert Java-Methoden in einem Vorkompilierungsschritt zu nativem Code, bevor die Anwendung ausgeführt wird. Vor WebSphere Real Time for RT Linux Version 2 verwendete der Vorkompilierungsschritt das Tool 'jxeinajar', um Methoden mit einem AOT-Compiler zu kompilieren, und speicherte die Ergebnisse in speziellen ausführbaren Java-Dateien. Diese Dateien werden möglicherweise in gebundenen JAR-Dateien erfasst. Bei der Ausführung einer Anwendung werden dem Anwendungsklassenpfad gebundene JAR-Dateien hinzugefügt, damit die JVM den AOT-Code laden kann, wenn die Methodenklassen aus JXE geladen werden. Aufgrund dieses Ansatzes ist der JIT-Compiler nicht verfügbar, wenn die Option **-Xnojit** in der Befehlszeile angegeben wird. Die Anwendung kann jeden vorkompilierten AOT-Code, der erstellt wurde, und den Interpreter für andere Methoden verwenden. Diese Betriebsart bietet eine hohe Bestimmungsfunktion, weil der JIT-Compiler nicht vorhanden ist. Daher kommt es zu keinen Leistungseinbußen aufgrund des Sampling-Threads oder eines Kontextwechsels. Aufgrund der Schwierigkeit, den Java-Code unter Einhaltung der Java-Spezifikation vorzeitig zu kompilieren, wird der mit AOT kompilierte Code in der Regel langsamer ausgeführt als der mit JIT kompilierte Code. Die Ausführung ist jedoch gewöhnlich viel schneller als das Interpretieren.

In WebSphere Real Time for RT Linux Version 2 und nachfolgenden Versionen wird der AOT-Code mithilfe der Technologie für gemeinsam genutzte Klassen gespeichert, die in den JVMs von IBM Java 6 bereitgestellt wird, in einem Cache für gemeinsam genutzte Klassen anstatt in JXE-Dateien. Mit dem Tool 'admincache' können Sie den Inhalt eines Cache abfragen, alle vorhandenen Caches auflisten und einen Cache mit Klassen und AOT-Code auffüllen. Die Speicherung von mit AOT kompiliertem Code hat den Vorteil, dass die JAR-Dateien der Anwendung nicht modifiziert werden und dass die Klassenpfade bei der Anwendungsausführung nicht geändert werden müssen.

Für einen Cache für gemeinsam genutzte Klassen gibt es auf der Basis des verfügbaren virtuellen Adressraums eine Begrenzung für eine praktikable Größe. Daher ist die AOT-Kompilierung für alle JAR-Dateien nicht praktikabel. Sie müssen eine selektive AOT-Kompilierung ausführen.

Wird eine Anwendung mit dem AOT-Code in einem Cache für gemeinsam genutzte Klassen ausgeführt, wird der AOT-Code für Methoden einer Klasse automatisch geladen, wenn die Klasse in die JVM geladen wird. Aufgrund des zusätzlichen Aufwands beim Laden einer Klasse zum Installieren von AOT-Code für die entsprechenden Methoden ist es wichtig, vor der Ausführung der leistungskritischen Anwendungskomponenten so viele Klassen wie möglich zu laden.

Die Verwendung von mit AOT vorkompiliertem Code bietet die höchste Bestimmungsfunktionsebene mit guter Leistung. Der AOT-Code kann verwendet werden, wenn Ihre Anwendung bei Angabe der Optionen **-Xshareclasses** und **-Xaot** ausgeführt wird. Die Option **-Xaot** ist standardmäßig aktiviert.

Informationen zum Speichern und Verwenden von AOT-Code mit einem Cache für gemeinsam genutzte Klassen über das Tool 'admincache' finden Sie in „Tool 'admincache' verwenden” auf Seite 43. Informationen zur Migration von 'jxeinajar' auf 'admincache' finden Sie in der Dokumentation für WebSphere Real Time for RT Linux Version.

Ein Beispiel für die Ausführung einer Anwendung anhand von Code, der mit AOT kompiliert ist, finden Sie in „Musteranwendung über AOT ausführen" auf Seite 92.

Gemischter Modus, der mit AOT vorkompilierten Code und JIT-Kompilierung mit niedriger Priorität kombiniert

Mit AOT und JIT kompilierter Code kann während der Anwendungsausführung zusammen verwendet werden. Diese Betriebsart kann eine sehr gute Bestimmungsfunktion mit guter Leistung und eine sehr hohe Leistung für häufig ausgeführte Methoden bieten. Der Hauptvorteil dieses Modus ist, dass mit der AOT-Vorkompilierung sichergestellt wird, dass die wichtigsten Komponenten Ihrer Anwendungen nie im Interpreter ausgeführt werden, der gewöhnlich viel langsamer ist als der mit AOT oder JIT kompilierte Code. Sie brauchen nicht alle Methoden vorzukompilieren, weil der JIT-Compiler alle interpretierten Methoden, die häufig ausgeführt werden, dynamisch ermitteln kann, ohne die Anwendungsleistung wesentlich zu senken. Der gemischte Modus ist der Standardmodus, wenn der Befehlszeile die Option **-Xshareclasses** hinzugefügt wird.

Informationen zur Ausführung Ihrer Anwendung mit gemischter AOT- und JIT-Kompilierung finden Sie in „Musteranwendung über AOT ausführen" auf Seite 92.

Kompilierung explizit verwalten

Bei Kompilierungsmodi mit aktiviertem JIT-Compiler können Sie die JIT-Compiler-Operation über die API java.lang.Compiler explizit steuern. Der JIT-Compiler kompiliert Methoden der übergebenen Klasse mithilfe der Methode compileClass(). compileClass() ist synchron, d. h., diese Methode wird erst zurückgegeben, nachdem die angegebenen Methode kompiliert wurden. Eine Anwendung verwendet compileClass() möglicherweise in einer Initialisierungsphase durch das Iterieren der Klassen, die von der Hauptphase der Anwendungsausführungszeit verwendet werden. Wenn die Initialisierungsphase beendet ist, rufen Sie die Methode Compiler.disable() auf, um die Kompilierungs- und Sampling-Threads vollständig zu inaktivieren. Die Hauptschwierigkeit bei diesem Verfahren ist die Verwaltung der Liste der in der Anwendungsinitialisierungsphase zu ladenden und zu kompilierenden Klassen, vor allem während der Anwendungsentwicklung.

Weitere Informationen zur Verwaltung der Kompilierung in einer Anwendung finden Sie in IBM Real-Time Class Analysis Tool for Java.

Übersicht der Befehlszeilenoptionen für Kompilierung

Sie können eine Anwendung über die Option **-Xjit** mit aktivierter JIT-Funktion bzw. über die Option **-Xnojit** ohne JIT ausführen. **-Xjit** ist der Standardmodus.

Sie können eine Anwendung über die Optionen **-Xshareclasses -Xaot** mit aktiviertem AOT-Code ausführen. Sie können den AOT-Code über die Option **-Xnoaot** inaktivieren. **-Xaot** ist die Standardoption, sie hat jedoch keine Auswirkung, sofern nicht auch die Option **-Xshareclasses** angegeben wird, weil der AOT-Code in einem Cache für gemeinsam genutzte Klassen gespeichert werden muss.

AOT-Compiler verwenden

Mit den folgenden Schritten vorkompilieren Sie Ihren Java-Code. Die folgende Prozedur beschreibt die Verwendung der Option **-Xrealtime** in einem Befehl **javac**, das Tool **admincache** und die Optionen **-Xrealtime** sowie **-Xnojit** für den Befehl **java**.

Informationen zu diesem Vorgang

Die Verwendung des AOT-Compilers bedeutet, dass die Kompilierung von der Ausführungszeit der Anwendung getrennt ist. Sie können außerdem zur selben Zeit weitere Methoden kompilieren anstatt nur die gängigsten Methoden. Sie können alle Klassen in einer Anwendung oder nur einzelne Klassen kompilieren, wie in den folgenden Schritten gezeigt wird.

Anmerkung: Bei Verwendung von Caches für gemeinsam genutzte Klassen darf der Name des Cache 53 Zeichen nicht überschreiten.

Vorgehensweise

1. Geben Sie an einer Shelleingabeaufforderung Folgendes ein:

   ```
   javac -Xrealtime Quelle
   ```

   Dieser Befehl erstellt den Java-Bytecode aus Ihrer Quelle für die Verwendung in der Echtzeitumgebung. Weitere Informationen hierzu finden Sie in Abb. 2 auf Seite 9.
2. Paketieren Sie die generierten Klassendateien in eine JAR-Datei. Geben Sie z. B. Folgendes ein, um 'test.jar' zu erstellen:

   ```
   jar cvf test.jar Quelle
   ```
3. Geben Sie an einer Shelleingabeaufforderung Folgendes ein:

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

   Dieser Befehl vorkompiliert die Datei test.jar und schreibt die Ausgabe in das Ausgabeverzeichnis `./aot`.
4. Geben Sie an einer Shelleingabeaufforderung Folgendes ein: Geben Sie an einer Shelleingabeaufforderung Folgendes ein, um die Datei mit dem AOT-Code im Cache für gemeinsam genutzte Klassen auszuführen:

   ```
   java -Xrealtime -Xshareclasses:name=meinCache -cp test.jar -Xnojit MeineTestklasse
   ```

   Geben Sie an einer Shelleingabeaufforderung Folgendes ein, um die Datei mit dem AOT-Code im Cache für gemeinsam genutzte Klassen auszuführen und häufig aufgerufene Methoden erneut zu kompilieren, ohne eine neue JAR-Datei zu erstellen:

   ```
   java -Xrealtime -Xshareclasses:name=meinCache -cp test.jar MeineTestklasse
   ```

   Diese Befehle verwenden die in Schritt 3 vorkompilierten JAR-Dateien.

Tool 'admincache' verwenden

Mit dem Tool 'admincache' können Sie die Caches für gemeinsam genutzte Klassen auf einer Workstation verwalten.

Bei IBM WebSphere Real Time for RT Linux können Sie mit dem Tool 'admincache' einen Cache für gemeinsam genutzte Klassen erstellen, der nur Klassen bzw. Klassen und mit AOT kompilierten Code enthält. Nach der Erstellung der Caches können Sie mit diesem Tool auch vorhandene Caches überprüfen.

Mit dem Cache für gemeinsam genutzte Klassen wird der Speicherbedarf bei Szenarios mit mehreren JVMs gesenkt und der Anwendungsstart beschleunigt.

Caches für gemeinsam genutzte Klassen können von WebSphere Real Time for RT Linux im Nicht-Echtzeitmodus und Echtzeitmodus verwendet werden, das Cacheformat, die Erstellung und die Füllverfahren sind jedoch unterschiedlich.

Caches im Echtzeitmodus sind nicht kompatibel mit Caches im Nicht-Echtzeitmodus. Im Nicht-Echtzeitmodus werden Caches auf dieselbe Weise wie bei der Standard-JVM erstellt und aufgefüllt. Dies bedeutet, dass der Cache von der JVM während der Ausführung einer Anwendung auf für den Benutzer transparente Weise erstellt und aufgefüllt wird. Bei Verwendung der Option **-Xrealtime** im Echtzeitmodus müssen Caches für gemeinsam genutzte Klassen von 'admincache' unter Verwendung der Option **-populate** erstellt und vorab aufgefüllt werden. Anwendungen, die im Echtzeitmodus ausgeführt werden, lesen möglicherweise Inhalt aus dem vorab aufgefüllten Cache, können diesen Inhalt jedoch nicht ändern.

Im Echtzeitmodus erstellte Caches für gemeinsam genutzte Klassen können nur verwendet werden, wenn eine Anwendung im Echtzeitmodus ausgeführt wird. Im Nicht-Echtzeitmodus erstellte Caches für gemeinsam genutzte Klassen können nur verwendet werden, wenn eine Anwendung im Nicht-Echtzeitmodus ausgeführt wird. Dies gilt auch für das Tool 'admincache'. Verwenden Sie 'admincache' mit der Option **-Xrealtime**, um von der JVM im Echtzeitmodus erstellte Caches zu verwalten. Verwenden Sie nicht die Option **-Xrealtime**, um von der JVM im Nicht-Echtzeitmodus erstellte Caches zu verwalten. Fügen Sie der Befehlszeile die Option **-Xshareclasses** hinzu, um eine Verbindung zu einem Cache für gemeinsam genutzte Klassen während der Ausführung herzustellen.

Auf einer Workstation können mehrere Caches für gemeinsam genutzte Klassen mit individuellen Namen in einem bestimmten Verzeichnis erstellt werden. Wenn ein neuer Cache erstellt wird, kann sein Name mit der Option **-cacheName** <*Name*> angegeben werden. Der Cachename darf nicht mehr als 53 Zeichen umfassen.

Caches für gemeinsam genutzte Klassen werden standardmäßig im Verzeichnis `/tmp/javasharedresources` erstellt, diese Speicherposition kann jedoch durch die Angabe der Option **-cacheDir** <*Verzeichnis*> überschrieben werden. Das interne Format für einen Cache für gemeinsam genutzte Klassen hängt von den Merkmalen der Workstation ab, auf der er erstellt wird. Dies bedeutet, dass Caches für gemeinsam genutzte Klassen nicht als Sicherheitsmaßnahme auf Netzlaufwerken erstellt werden können. Ein weiterer Grund für diese Einschränkung ist langsamere und unvorhersehbare Leistung, wenn von einem Netzdateisystem auf einen Cache für gemeinsam genutzte Klassen zugegriffen wird.

Wird in der Befehlszeile kein Cachename angegeben, ist der Standard **sharedcc_<Benutzeranmeldung>**

Weitere Informationen zur Operation des gemeinsam genutzten Cache im Nicht-Echtzeitmodus finden Sie in „Gemeinsame Nutzung von Klassendaten auf verschiedenen JVMs im Nicht-Echtzeitmodus” auf Seite 101.

Anmerkung: Ab IBM WebSphere Real Time for RT Linux Version 2 SR1 und später müssen Sie die Option **-classpath** mit der Option **-populate** verwenden.

Echtzeitorientierten Cache für gemeinsam genutzte Klassen erstellen:

Mit dem Tool 'admincache' können Sie Caches für gemeinsam genutzte Klassen erstellen, auf die im Echtzeitmodus zugegriffen werden kann.

Anmerkung: Sie müssen sich der Sicherheitsaspekte bewusst sein, wenn Sie Cachedateien für gemeinsam genutzte Klassen mit Standardeinstellungen erstellen. Weitere Informationen zu Sicherheitsaspekten für den Cache für gemeinsam genutzte Klassen und zum Ändern der Standardberechtigungen finden Sie in „Sicherheitsaspekte für den Cache für gemeinsam genutzte Klassen” auf Seite 103.

Mit der Option **-populate** des Tools 'admincache' können Sie Caches für gemeinsam genutzte Klassen erstellen. Über diese Option wird zusammen mit einer Liste von JAR-Dateien, einem Verzeichnis oder einer Verzeichnisstruktur nach JAR-Dateien gesucht. Für jede angegebene oder gefundene JAR-Datei speichert 'admincache' alle darin gefundenen Klassen im Cache für gemeinsam genutzte Klassen. Die Klassenmethoden werden mit AOT kompiliert und im Cache für gemeinsam genutzte Klassen gespeichert, außer Sie geben die Option **-noaot** an.

Sie müssen die Option **-classpath** mit **-populate** verwenden. Andernfalls wird die folgende Fehlernachricht angezeigt:

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

Die Option **-help** von 'admincache' listet die Unteroptionen auf, mit denen Sie steuern können, wie 'admincache' den Cache auffüllt.

```
$ 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
```

Anmerkung: Bei der Verwendung von Caches für gemeinsam genutzte Klassen darf der durch die Option **-cacheName** angegebene Name 53 Zeichen nicht überschreiten.

Sie können eine Liste von JAR-Dateien angeben. In diesem Fall werden dem Cache für gemeinsam genutzte Klassen nur die die Klassen aus diesen JAR-Dateien hinzugefügt. Wenn Sie keine Liste von JAR-Dateien angeben, geben Sie über die Option **-searchPath <Pfad>** eine Verzeichnisstruktur zum Suchen nach JAR- oder ZIP-

Dateien an. Die Option **-recurse** ist der Standardwert und bedeutet, dass die Verzeichnisstruktur rekursiv nach JAR- oder ZIP-Dateien durchsucht wird. Die Option **-norecurse** bedeutet, dass nur das angegebene Verzeichnis durchsucht wird. Geben Sie die Option **-classpath <Klassenpfad>** an, damit 'admincache' nach allen Klassen suchen kann, die zur Verarbeitung der angegebenen JAR-Dateien benötigt werden. Die Klassen werden beim Auffüllen des Cache für gemeinsam genutzte Klassen in die JVM geladen. Daher ist es wichtig, dass alle referenzierten Klassen und Superklassen von 'admincache' gefunden werden können, wenn dieses Tool versucht, eine Klasse aus einer JAR-Datei zu laden.

Die Option **-grow** gibt an, dass dem vorhandenen Cacheinhalt eine neue JAR-Datei hinzugefügt wird, wenn im Cacheverzeichnis ein gleichnamiger Cache für gemeinsam genutzte Klassen vorhanden ist. Die Option **-nogrow** gibt an, dass eine neue JAR-Datei den alten Cacheinhalt ersetzt, wenn im alten Cacheverzeichnis ein gleichnamiger Cache für gemeinsam genutzte Klassen vorhanden ist. Mit der Option **-grow** können Sie neue JAR-Dateien hinzufügen, die zurzeit im Cache für gemeinsam genutzte Klassen nicht vorhanden sind, jedoch nicht geänderte Klassen ersetzen. Verwenden Sie die Option **-grow** nicht, um Klassen zu aktualisieren, die bereits im Cache vorhanden sind, aufgrund von Anwendungsänderungen jedoch geändert wurden. Wenn Sie vorhandene Klassen aktualisieren wollen, erstellen Sie einen vollständig neuen Cache mit dem aktuellen Cacheinhalt. Wenn Sie Ihren Cache für gemeinsam genutzte Klassen beim Ändern einer Klasse nicht aktualisieren, wird Ihre Anwendung mit dem neuen Klasseninhalt ordnungsgemäß ausgeführt, nutzt jedoch nicht den Cache für gemeinsam genutzte Klassen. Dies liegt daran, dass die geänderte Klasse von der Platte und nicht aus dem Cache für gemeinsam genutzte Klassen geladen wird. Wird die Klasse von der Platte geladen, kann der mit AOT kompilierte Code nicht für diese Klasse verwendet werden. Generieren Sie Ihren Cache für gemeinsam genutzte Klassen neu, wenn Sie eine Klasse ändern.

Mit den Optionen **-quiet**, **-verbose** und **-noisy** können Sie die von 'admincache' bereitgestellte Detaillierungsebene steuern.

Mit der Option **-aot** können Sie die AOT-Vorkompilierung für die Methoden in den Klassen angeben, die den Cache für gemeinsam genutzte Klassen auffüllen. Mit der Option **-noaot** können Sie die AOT-Vorkompilierung verhindern, sodass nur Klassen im Cache für gemeinsam genutzte Klassen gespeichert werden. Die Option **-aot** ist die Standardeinstellung.

Mit der Option **-aotFilter** <*Signatur*> bzw. **-aotFilterFile** <*Datei*> können Sie einige Methoden selektiv vorkompilieren. Die <*Signatur*> ist ein vereinfachter regulärer Ausdruck für eine Methodensignatur, der in geschweifte Klammern eingeschlossen ist. Dabei kann '*' eine beliebige Zeichenfolge ersetzen. Sie müssen <*Signatur*> möglicherweise in Hochkommas einschließen, damit die Shell die Zeichen in der Methodensignatur nicht interpretiert.

Tabelle 4 zeigt einige Beispiele der Option <*Signatur*>.

Tabelle 4. Beispiele der Option <Signatur>

Signatur	Bedeutung
`-aotFilter '{java/lang/*}'`	AOT kompiliert Methoden im Paket java/lang.
`-aotFilter '{*.sample*}'`	AOT kompiliert Methoden, die mit "sample" anfangen.

Tabelle 4. Beispiele der Option <Signatur> (Forts.)

Signatur	Bedeutung
`-aotFilter '{mypackage/ myclass.mymethod(I)I}'`	AOT kompiliert die Methode mit dieser Signatur.

Die Option **-aotFilterFile** <*Datei*> verwendet den Inhalt von <*Datei*> zum Auswählen der Methoden für die AOT-Kompilierung. Keine anderen Methoden werden mit AOT kompiliert. Der Inhalt von <*Datei*> wird während einer früheren Ausführung der Anwendung mit der Option **-Xjit:verbose={precompile},vlog=**<*Datei*> generiert. Die in <*Datei*> gespeicherte ausführliche Ausgabe verwendet ein internes Format. Dieses Format wird von der Option **-aotFilterFile** benötigt.

Anmerkung: Die Option **-vlog=**<*Datei*> generiert nicht direkt eine Datei namens "Datei". An "Datei" werden ein Datum und eine Prozess-ID angehängt, wenn die ausführliche Ausgabe generiert wird. Durch die Angabe der Option **-Xjit:verbose={precompile},vlog=meine_Datei** ähnelt der generierte Dateiname `meine_Datei.<Datum>.<#>.<Prozess-ID>`. Die zusätzlichen Felder vereinfachen die Generierung einzelner ausführlicher Protokolldateien bei Szenarios mit mehreren JVMs, bei denen es schwierig sein kann, einer bestimmten JVM Befehlszeilenoptionen bereitzustellen oder verschiedene **-Xjit**-Befehlszeilenoptionen für verschiedene JVMs zu verwenden. Bei einem Szenario mit einer einzelnen JVM werden diese Zahlen an den Dateinamen angehängt, der in der Befehlszeile angegeben wird.

Mit der Option **-aotFilterFile** kann eine generierte Datei ohne Bearbeitung verwendet werden. Mehrere ausführliche Protokolldateien, die von mehreren Anwendungsausführungen mit der Option **-Xjit:verbose={precompile},vlog=<Datei>** generiert werden, können verknüpft und 'admincache' über die Option **-aotFilterFile** bereitgestellt werden.

Mit der Option **-printvmargs** können Sie sicherstellen, dass bei der Anwendungsausführung die richtigen Argumente in der Befehlszeile angegeben sind.

```
$ 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
```

In diesem Beispiel zeigt die letzte Ausgabezeile die Optionen an, die der Befehlszeile bei der Anwendungsausführung hinzugefügt werden müssen, damit die im Cache für gemeinsam genutzte Klassen gespeicherten Klassen und AOT-Methoden verwendet werden. Geben Sie den folgenden Befehl ein, um die Optionen aus diesem Beispiel zu verwenden:

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

Caches für gemeinsam genutzte Klassen mit 'admincache' verwalten:

Das Tool 'admincache' enthält mehrere Dienstprogramme zum Verwalten der Caches für gemeinsam genutzte Klassen auf Ihrem System.

Das Tool 'admincache' liefert Dienstprogramme für mehrere Aktivitäten:

- Die in einem Cache vorhandenen Caches für gemeinsam genutzte Klassen auflisten.
- Details zum Inhalt eines Cache für gemeinsam genutzte Klassen bereitstellen.
- Einige oder alle Caches in einem bestimmten Cacheverzeichnis entfernen.

Verfügbare Caches für gemeinsam genutzte Klassen auflisten:

Das Tool 'admincache' liefert eine Liste von Caches für gemeinsam genutzte Klassen, die in einem Cache vorhanden sind.

Mit der Option **-listAllCaches** zusammen mit der Option **-cacheDir** zur Angabe des Cacheverzeichnisses können Sie eine Liste aller Caches für gemeinsam genutzte Klassen anfordern, die in einem Cache vorhanden sind.

```
$ 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
```

In diesem Beispiel enthält das Standardcacheverzeichnis zwei kompatible Caches für gemeinsam genutzte Klassen:

- Der Standardcache für den Benutzer mit der Anmeldung *username*
- Ein weiterer Cache namens *rtCache*

Das Beispiel zeigt auch einen inkompatiblen Cache namens *nonrtCache*. Dieser Cache wurde von der JVM während der Ausführung im Nicht-Echtzeitmodus erstellt. Dies bedeutet, dass Sie über die Option **-Xrealtime** nicht auf ihn zugreifen können.

Die JVM im Echtzeitmodus kann im Nicht-Echtzeitmodus erstellte Caches erkennen. Die JVM im Nicht-Echtzeitmodus kann im Echtzeitmodus erstellte Caches nicht erkennen.

```
$ 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
```

In diesem Beispiel ist *nonrtCache* als kompatibel aufgelistet, weil **-Xrealtime** nicht angegeben ist.

Inhalt von Caches für gemeinsam genutzte Klassen überprüfen:

Das Tool 'admincache' beschreibt den Inhalt eines Cache für gemeinsam genutzte Klassen.

Mit der Option **-printStats** des Tools 'admincache' können Sie eine Übersicht anfordern, die den Hauptinhalt eines Cache für gemeinsam genutzte Klassen beschreibt. Informationen zu einem bestimmten Cache in einem bestimmten Cacheverzeichnis können Sie über die Optionen **-cacheName** und **-cacheDir** abrufen. Das folgende Beispiel liefert Informationen zum Cache *nonrtCache* im Standardcacheverzeichnis.

```
$ 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
```

Anmerkung: Bei Verwendung von Caches für gemeinsam genutzte Klassen darf der Name des Cache 53 Zeichen nicht überschreiten.

Es gibt mehrere nützliche Informationen zu diesem Cache:

- Die Größe des Cache, die als `cache size = 16776852` angezeigt wird.
- Der im Cache verfügbare Speicherplatz, der als `free bytes = 14070360` angezeigt wird. Sie können berechnen, dass der Cache ungefähr 16 % voll ist.
- Die Anzahl der im Cache gespeicherten Klassen, die als `# ROMClasses = 372` angezeigt wird.
- Die Anzahl der im Cache gespeicherten AOT-Methoden, die als `# AOT Methods = 981` angezeigt wird.

Weitere Details zu den Informationen, die von der Option **-printStats** im Tool 'admincache' bereitgestellt werden, finden Sie in Dienstprogramm 'printStats'.

Die Option **-printAllStats** liefert eine detailliertere Beschreibung des Inhalts eines Cache für gemeinsam genutzte Klassen. Die Informationen enthalten die Liste von Klassen und AOT-Methoden, die im Cache gespeichert sind. Die Ausgabe der Option **-printAllStats** ist ausführlich.

Im Cache enthaltene Klassen werden durch Zeilen angegeben, die dem Folgendem ähneln:

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

Diese Zeile gibt an, dass die Klasse java/lang/ClassLoader im Cache enthalten ist. Die Adressen sind interne Adressen des Cache für gemeinsam genutzte Klassen und eignen sich gewöhnlich nur für Diagnosezwecke.

Im Cache enthaltene AOT-Methoden werden durch Zeilen angegeben, die dem Folgendem ähneln:

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

Diese Zeilen geben an, dass die Methode callerClassLoader aus der Klasse java/lang/ClassLoader im Cache enthalten ist. Die aufgelisteten Adressen sind interne Adressen des gemeinsam genutzten Cache. Die Ausgabe der Option **-printAllStats** zeigt die aus Parametertypen und Rückgabetyp bestehende Signatur für jede AOT-Methode im Cache nicht an.

Weitere Details zu den Informationen, die von der Option **-printAllStats** im Tool 'admincache' bereitgestellt werden, finden Sie in Dienstprogramm 'printAllStats'.

Caches für gemeinsam genutzte Klassen löschen:

Das Tool 'admincache' verfügt über Optionen zum Löschen eines bestimmten Cache oder aller Caches in einem angegebenen Cacheverzeichnis.

Mit der Option **-destroy** des Tools 'admincache' kann ein bestimmter Cache in einem bestimmten Cacheverzeichnis gelöscht werden, sofern der Benutzer über die entsprechende Berechtigung verfügt. Mit der Option **-destroyAll** können alle Caches gelöscht werden, sofern der Benutzer über die entsprechende Berechtigung verfügt. Beispiel:

```
$ admincache -Xrealtime -destroy

admincache 1.02

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

Nach dem Löschen des Cache zeigt eine Liste der im Standardcacheverzeichnis verfügbaren Caches für gemeinsam genutzte Klassen, dass der gelöschte Cache nicht mehr angezeigt wird:

```
$ 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
```

Die Option **-destroyAll** entfernt alle Caches im angegebenen Cacheverzeichnis. Dabei spielt es keine Rolle, ob sie mit der aktuellen JVM kompatibel sind oder nicht. Die Option **-destroyAll** muss mit Sorgfalt verwendet werden:

```
$ 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
```

Das Ergebnis ist, dass auf der Maschine keine Caches für gemeinsam genutzte Klassen mehr verfügbar sind:

```
$ admincache -Xrealtime -listAllCaches

admincache 1.02

JVMSHRC005I No shared class caches available
```

Wenn der aktuelle Benutzer nicht zum Zugreifen auf einen Cache berechtigt ist, wird der Cache durch die Option **-destroy** oder **-destroyAll** nicht gelöscht.

Praktikable Größen für Caches für gemeinsam genutzte Klassen:

Das Tool 'admincache' liefert Informationen für die Dimensionierung der Caches für gemeinsam genutzte Klassen.

Bei kleineren Anwendungen kann ein Cache für gemeinsam genutzte Klassen mit allen Klassen und Methoden aufgefüllt werden, ohne einen übermäßig großen Cache zu erzeugen. Bei größeren Anwendungen wird der sich ergebende Cache für gemeinsam genutzte Klassen möglicherweise zu groß und damit unpraktisch. Der Grund hierfür ist, dass ein JVM-Prozess über ausreichenden virtuellen Adressraum verfügen muss, um den gesamten Inhalt des Cache für gemeinsam genutzte Klassen verarbeiten zu können. Wenn Sie die Technologie des Cache für gemeinsam genutzte Klassen verwenden, müssen Sie gewisse Aspekte beachten.

Praktisch der gesamte Cache für gemeinsam genutzte Klassen muss in der JVM, zu der eine Verbindung hergestellt wird, adressierbar sein. Vermeiden Sie daher die Verwendung von Caches für gemeinsam genutzte Klassen, die größer als 700 MB sind. Das Tool 'admincache' kann die Größe eines Cache vorhersagen. Wenn das Tool angibt, dass der Cache den Grenzwert von 700 MB überschreitet, wird Ihnen in einer Nachricht empfohlen, eine geringere Anzahl Klassen zu speichern bzw. die Anzahl der im Cache gespeicherten AOT-Methoden zu beschränken.

```
$ admincache -Xrealtime -populate veryBigJar.jar -cp <mein Klassenpfad>

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.
```

Das Tool 'admincache' sagt eine konservative Cachegröße basierend auf der Gesamtgröße der JAR-Dateien vorher, die für die Auffüllung angegeben oder gefunden wurden. Dies bedeutet, dass die Vorhersage möglicherweise nicht korrekt ist, wenn die JAR-Datei viele Dateien enthält, die keine Klassendateien sind. Erstellen Sie temporäre Versionen der JAR-Dateien, die nur die Klassendateien enthalten, um eine genauere Vorhersage der Cachegröße zu erhalten. Wenn das Tool 'admincache' weiterhin eine Warnung erzeugt, vorkompilieren Sie die Methoden in der JAR-Datei mit AOT selektiver über die Option **-aotFilter** <*Muster*> oder **-aotFilterFile** <*Datei*>. Die Nachricht des Tools 'admincache' erinnert Sie daran, dass die Vorhersage die durch diese Optionen gefilterten AOT-Methoden nicht berücksichtigt.

Fügen Sie die angegebene Option **-Xscmx** der Befehlszeile von 'admincache' hinzu, um die Warnung außer Kraft zu setzen und mit dem Schritt zum Auffüllen des Cache fortzufahren. Wenn die vorhergesagte Größe sehr hoch ist, kann das Tool 'admincache' möglicherweise keinen Cache für gemeinsam genutzte Klassen mit der erforderlichen Größe erstellen. Sie lösen dieses Problem, indem Sie die Cachegröße senken, bis das Tool 'admincache' fortfahren kann.

Wenn der endgültige Cache auf die Platte geschrieben wird, ist er nur so groß, wie er zum Speichern der angegebenen Klassen und AOT-Methoden sein muss. Dies bedeutet, dass die Angabe einer hohen Anfangscachegröße kein Problem ist.

SDK-Klassen in einem Cache für gemeinsam genutzte Klassen speichern:

Die Erstellung eines Cache, der alle SDK-JAR-Dateien enthält, ist möglicherweise nicht für alle Anwendungen notwendig.

Die Anzahl und Größe der JAR-Dateien im SDK bedeuten, dass der Versuch, einen Cache mit allen JAR-Dateien zu erstellen, zu einer Warnung führt, in der Ihnen mitgeteilt wird, dass der sich ergebende Cache zu groß ist. Bei vielen Anwendungen wird auf die meisten SDK-JAR-Dateien nie verwiesen.

Die wesentlichen SDK-JAR-Dateien sind im Verzeichnis `SDK/jre/lib` angeordnet. Für die meisten Anwendungen ist `rt.jar` die wichtigste JAR-Datei. Sie ist in Java 6-Releases neu. `rt.jar` ist eine Sammlung von Klassen, die vor dem Java 6-Release in separaten JAR-Dateien gespeichert wurden. Durch die Auffüllung eines Cache für gemeinsam genutzte Klassen nur mit `rt.jar` und die Kompilierung aller entsprechenden Methoden mit dem AOT-Compiler wird ein Cache erstellt, der ungefähr 300 MB groß ist.

Eine Standardanwendung verweist nicht auf die meisten Methoden aus den Klassen von rt.jar. Gehen Sie wie folgt vor, um einen Cache für gemeinsam genutzte Klassen mit rt.jar aufzufüllen:

1. Füllen Sie den Cache für gemeinsam genutzte Klassen nur mit den Klassen aus rt.jar auf. Hierdurch werden ungefähr 50 MB Speicherplatz im Cache belegt.
2. Kompilieren Sie über die Option **-aotFilterFile** <*Datei*> nur die Methoden, die Ihr Programm verwendet. Sie können <*Datei*> generieren, indem Sie die Anwendung ausführen.

Das SDK enthält andere gängige und wichtige JAR-Dateien wie beispielsweise:
- sdk/jre/lib/i386/realtime/jclSC160/realtime.jar
- sdk/jre/lib/i386/realtime/jclSC160/vm.jar
- sdk/jre/lib/java.util.jar

realtime.jar enthält die IBM Implementierung von Real Time Specification for Java (RTSJ). Wenn Ihre Anwendung RTSJ-Funktionen verwendet, speichern Sie die Datei realtime.jar für bessere deterministische Leistung im Cache für gemeinsam genutzte Klassen. vm.jar enthält mehrere interne JVM-Klassen, die gewöhnlich in allen Anwendungen verwendet werden. java.util.jar enthält mehrere Containerklassen und muss für bessere deterministische Leistung im Cache für gemeinsam genutzte Klassen jeder Anwendung gespeichert werden.

Andere JAR-Dateien in den Verzeichnissen sdk/jre/lib und sdk/jre/lib/ext können in einem Cache für gemeinsam genutzte Klassen gespeichert werden, wenn eine Anwendung diese Klassen verwendet. Führen Sie Ihr Programm mit der Option **-verbose:dynload** aus, um zu ermitteln, ob Ihre Anwendung diese Klassen verwendet. Die Option **-verbose:dynload** beschreibt nur die von der aktuellen Ausführung der Anwendung geladenen Klassen. Beispiel:

```
<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>
```

Diese Beispielausgabe zeigt drei Klassen, die aus zwei verschiedenen SDK-JAR-Dateien geladen wurden. Die Klasse java/io/InputStreamReader wurde aus rt.jar geladen. Die Klassen java/util/LinkedHashSet und java/util/HashSet wurden aus java.util.jar geladen.

Andere Aspekte von 'admincache':

Nützliche Informationen für das Arbeiten mit 'admincache'.

Cachefüllung und Dimensionierung des Speichers für Objekte mit unbeschränkter Lebensdauer

Wenn das Tool **admincache** einen Cache für gemeinsam genutzte Klassen im Echtzeitmodus auffüllt, muss während des Füllprozesses jede Klasse geladen werden. Jede Klasse belegt Speicher für Objekte mit unbeschränkter Lebensdauer.

Daher ist es möglich, dass die Standardgröße des Speichers für Objekte mit unbeschränkter Lebensdauer für alle angeforderten Klassen nicht ausreicht. Wenn das Tool **admincache** den Fehler OutOfMemory beim Auffüllen eines Cache mit vielen Klassen auslöst, versuchen Sie, den Speicher für Objekte mit unbeschränkter Lebensdauer mithilfe der Option **-Xgc:immortalMemorySize=32M** über den Standardwert von 16 MB hinaus zu erhöhen.

Klassen ändern

Wenn eine Klassendatei auf der Platte geändert wird, erkennt die Technologie des Cache für gemeinsam genutzte Klassen automatisch, dass die zwischengespeicherte Version dieser Klasse in einem Cache für gemeinsam genutzte Klassen nicht verwendet werden darf. Ihr Programm funktioniert ordnungsgemäß, kann den Cache für gemeinsam genutzte Klassen jedoch nicht optimal nutzen, und die AOT-Methoden für diese Klasse werden nicht verwendet. Wenn Sie eine Klasse in Ihrer Anwendung ändern, erstellen Sie Ihren Cache für gemeinsam genutzte Klassen erneut. Versuchen Sie nicht, über die Option **-grow** nur die JAR-Datei mit der geänderten Klasse erneut aufzufüllen, weil diese Option nicht für eine bereits im Cache vorhandene JAR-Datei gilt.

Gemeinsam genutzte Caches verwalten

Gemeinsam genutzte Caches erfordern Adressraum, selbst wenn keine Dateien geladen sind. Weitere Informationen zur Belegung von Speicher durch Caches für gemeinsam genutzte Klassen im JVM-Prozess finden Sie in „Speicherverwaltung durch die IBM JVM" auf Seite 114.

Vorkompilierte JAR-Dateien in einem Cache für gemeinsam genutzte Klassen speichern

Sie können alle oder einige der von IBM bereitgestellten Java-Klassen in einen Cache für gemeinsam genutzte Klassen einschließen. Dieser Prozess verwendet die Option **-Xrealtime** mit **javac** und das Tool **admincache**, um die Klassen in einem Cache für gemeinsam genutzte Klassen zu speichern.

Vorbereitende Schritte

Das Vorabspeichern von JAR-Dateien in einem Cache für gemeinsam genutzte Klassen wird nur unterstützt, wenn die Option **-Xrealtime** verwendet wird und Java mit der Option **-Xrealtime** ausgeführt wird. Sie können dieselben JAR-Dateien bei der Ausführung mit oder ohne **-Xrealtime** verwenden, die im Cache gespeicherten JAR-Dateien können jedoch nur verwendet werden, wenn **-Xrealtime** angegeben wird.

Anmerkung: Bei Verwendung von Caches für gemeinsam genutzte Klassen darf der Cachename 53 Zeichen nicht überschreiten.

Informationen zu diesem Vorgang

Sie können JAR-Dateien mit dem Tool **admincache** in einem Cache für gemeinsam genutzte Klassen speichern. Mit **admincache** können Sie Ihre Anwendung auf eine von drei Arten erstellen.

Anmerkung:
- Wenn Sie für Ihr Linux-System ein Zeitlimit festgelegt haben, müssen Sie es bei der Kompilierung großer JAR-Dateien möglicherweise überschreiben. Andernfalls wird das Kompilierungszeitlimit überschritten und die JAR-Datei nicht erstellt.

Alle Klassen und Methoden in einer Anwendung vorkompilieren:

Diese Prozedur vorkompiliert alle Klassen in einer Anwendung. Sie speichert eine Gruppe von JAR-Dateien in einem Cache für gemeinsam genutzte Klassen. Alle Methoden in allen Klassen in diesen JAR-Dateien werden im Cache gespeichert. Alle Methoden der optimierten JAR-Dateien sind kompiliert.

Informationen zu diesem Vorgang

Im folgenden Beispiel ist die Anwendung im durch die Umgebungsvariable *$APP_HOME* angegebenen Verzeichnis angeordnet und die JAR-Dateien befinden sich im Unterverzeichnis *$APP_HOME*/lib. Die Anwendung verwendet auch einige der von IBM in core.jarrt.jar bereitgestellten Klassen. In diesem Fall können Sie nur den Anwendungscode vorkompilieren, also main.jar und util.jar.

Der Cache für gemeinsam genutzte Klassen befindet sich standardmäßig in /tmp/javasharedresources. Mit der Option **-cacheDir** können Sie den Cache in ein anderes Verzeichnis stellen. In einem Netzdateisystem können Sie keinen Cache erstellen.

Vorgehensweise

1. Geben Sie an einer Shelleingabeaufforderung Folgendes ein: cd *$APP_HOME*
   Dabei ist *$APP_HOME* das Verzeichnis Ihrer Anwendung.

2. Geben Sie an einer Shelleingabeaufforderung Folgendes ein: `cd $APP_HOME/lib`. `$APP_HOME/lib` ist das Verzeichnis, in dem `main.jar` und `util.jar` gespeichert werden.
3. Geben Sie an einer Shelleingabeaufforderung Folgendes ein: `admincache -Xrealtime -populate -aot -classpath $APP_HOME/lib -searchPath $APP_HOME/lib -norecurse` . Diese Prozedur optimiert alle in `$APP_HOME/lib` gefundenen JAR-Dateien. Dabei werden Fortschrittsinformationen auf dem Bildschirm angezeigt und wird die neue JAR-Datei dann im Verzeichnis `$APP_HOME/aot` erstellt. Mit **-cacheName <Name>** können Sie einen Namen angeben. Wird kein Name angegeben, wird ein Standardname auf der Basis der Benutzeranmeldung verwendet.

   Anmerkung: Der mit der Option **-cacheName** angegebene Name darf nicht mehr als 53 Zeichen umfassen.
4. Durch die Eingabe von `admincache -Xrealtime -listAllCaches` an einer Shelleingabeaufforderung wird das Vorhandensein des Cache angezeigt.

Nächste Schritte

Geben Sie Folgendes an, um weitere Optionen abzurufen: `admincache -Xrealtime -help`.

Häufig verwendete Methoden vorkompilieren:

Mit der profilgesteuerten AOT-Kompilierung können Sie nur die Methoden vorkompilieren, die von der Anwendung häufig verwendet werden. Die AOT-Kompilierung speichert eine Gruppe von JAR-Dateien in einem Cache für gemeinsam genutzte Klassen. Dabei wird eine Optionsdatei verwendet, die durch die Ausführung der Anwendung mit der Sonderoption **-Xjit:verbose={precompile},vlog=optFile** generiert wird. Nur die in der Optionsdatei aufgelisteten Methoden werden vorkompiliert.

Vorbereitende Schritte

Erstellen Sie vor dem Start eine Liste der Methoden, die gewöhnlich von einem JIT-Compiler kompiliert werden.

Informationen zu diesem Vorgang

Sie können die durch die Option **-Xjit:verbose={precompile}** generierte Datei bearbeiten. Die Datei ist eine explizite Spezifikation der Methoden, die vorkompiliert werden sollen. Diese Methoden sind spezifisch, d. h., sie enthalten die volle Signatur für jede zu kompilierende Methode. Deshalb können Sie `com/acme/sample.myMethod(J)V`, jedoch nicht `com/acme/sample.myMethod(I)V` kompilieren.

Anmerkung: Bei Verwendung von Caches für gemeinsam genutzte Klassen darf der Name des Cache 53 Zeichen nicht überschreiten.

Vorgehensweise

1. Geben Sie an einer Shelleingabeaufforderung Folgendes ein:
   `cd $APP_HOME`

   Dabei ist *$APP_HOME* das Verzeichnis Ihrer Anwendung.
2. Geben Sie an einer Shelleingabeaufforderung Folgendes ein:

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

Dabei gilt Folgendes:

- *app.precompileOpts* ist der Name der Protokolldatei, die die mit JIT kompilierten Methoden auflistet.
- *applicationName* ist der Name Ihrer Anwendung.

Dieser Befehl erstellt eine Liste der Methoden, die über JIT kompiliert werden.

3. Geben Sie an einer Shelleingabeaufforderung Folgendes ein:

   ```
   cd $APP_HOME/lib
   ```

   $APP_HOME/lib ist das Verzeichnis, in dem die JAR-Dateien für Ihre Anwendung gespeichert werden.

4. Geben Sie Folgendes ein, um alle Musteranwendungsmethoden im Cache zu kompilieren:

   ```
   admincache -Xrealtime -populate -cacheName myCache \
              -aotFilterFile $APP_HOME/app.precompileOpts \
              -cp $APP_HOME/lib/demo.jar
   ```

5. Geben Sie Folgendes ein, um `realtime.jar` und `vm.jar` im Cache zu kompilieren:

```
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. Geben Sie Folgendes ein, um `rt.jar` im Cache zu kompilieren:

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

7. Führen Sie Ihre Anwendung mit der Option **-nojit** aus, die den Code im Cache verwendet, um diesen Befehl zu testen. Geben Sie an der Shelleingabeaufforderung Folgendes ein:

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

Dabei ist *Anwendungsname* der Name Ihrer Anwendung.

Von IBM bereitgestellte Dateien vorkompilieren:

Sie können von IBM bereitgestellte Dateien wie `rt.jar` vorkompilieren, um einen Kompromiss zwischen Leistung und Vorhersagbarkeit zu erzielen.

Informationen zu diesem Vorgang

Die Vorkompilierung ähnelt der Vorkompilierung der JAR-Dateien Ihrer Anwendung, während der Ausführung gilt jedoch eine zusätzliche Voraussetzung. Sie müssen sicherstellen, dass Ihr Bootklassenpfad ordnungsgemäß angegeben ist, sodass diese Dateien anstatt der Dateien in der JRE verwendet werden. Sie können hierfür die Option **-Xshareclasses** verwenden, die die JVM anweist, zuerst im angegebenen Klassencache anstatt an den Standardklassepfadpositionen zu suchen.

Anmerkung: Bei Verwendung von Caches für gemeinsam genutzte Klassen darf der Name des Cache 53 Zeichen nicht überschreiten.

Vorkompilieren Sie `rt.jar` für die Verwendung mit der Anwendung:

Vorgehensweise

1. Geben Sie an einer Shelleingabeaufforderung Folgendes ein: `cd $JAVA_HOME/lib`. Dabei ist *$JAVA_HOME* Ihr Java-Ausgangsverzeichnis.
2. Führen Sie das Tool **admincache** aus. Geben Sie an einer Shelleingabeaufforderung Folgendes ein:

   ```
   admincache -Xrealtime -populate -cacheName meinCache -classpath <Klassenpfad> rt.jar
   ```

   Dieser Befehl füllt den Cache `meinCache` mit den Ergebnissen der Vorkompilierung für die von IBM bereitgestellte Datei `rt.jar` auf.
3. Führen Sie Ihre Anwendung mit **-Xshareclasses** aus, um den Cachenamen anzugeben. Geben Sie zum Ausführen Ihrer Anwendung Folgendes ein:

   ```
   java -Xrealtime -Xnojit -Xshareclasses:name=meinCache
            -classpath:$APP_HOME/main.jar:$APP_HOME/util.jar ...
   ```

JIT-Compiler

Sie können mit der als Teil der Standard-SDK-Klassenbibliothek gelieferten Klasse java.lang.Compiler steuern, wann und wie der Just-in-time-Compiler (JIT-Compiler) fungiert. IBM unterstützt die Methoden Compile.compileClass(), Compiler.enable() und Compiler.disable() vollständig.

Wenn Sie z. B. Ihre Anwendung aktivieren wollen und wissen, dass die Schlüsselmethoden in Ihrer Anwendung kompiliert worden sind, können Sie die Methode Compiler.disable() aufrufen, nachdem Sie Ihre Anwendung aktiviert haben und zuversichtlich sind, dass die JIT-Kompilierung während der restlichen Ausführung Ihrer Anwendung nicht auftritt.

Sie können die Methodenkompilierung auf zwei Arten steuern:

- Geben Sie eine Gruppe von Methoden an, die Sie kompilieren können:

  ```
  Compiler.command("{<Methodenspezifikation>}(compile)");
  ```

  Dabei ist <*Methodenspezifikation*> eine Liste aller Methoden, die zu diesem Zeitpunkt geladen wurden und die kompiliert werden sollen. <*Methodenspezifikation*> beschreibt einen vollständig qualifizierten Methodennamen. Ein Stern gibt einen Platzhalterabgleich an.

  Sollen z. B. alle Methoden kompiliert werden, die mit java.lang.String anfangen und bereits geladen wurden, geben Sie Folgendes an:

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

  Anmerkung: Dieser Befehl kompiliert nicht nur Methoden in der Klasse java.lang.String, sondern auch in der Klasse java.lang.StringBuffer, was möglicherweise nicht beabsichtigt war. Sie müssen Folgendes angeben, um nur Methoden in der Klasse java.lang.String zu kompilieren:

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

- Geben Sie an, dass alle Methoden in der Kompilierungswarteschlange kompiliert werden, bevor dieser Thread ausgeführt und fortgesetzt wird:

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

  Sie stellen am besten sicher, dass die Kompilierungswarteschlange leer war, bevor Sie den Compiler inaktivieren. Ein Standardverfahren für das Kompilieren einer Gruppe von Methoden und Klassen sieht wie folgt aus:

```
Compiler.enable();                              // ensure compiler is active
Compiler.command("{com.mycompany.*}(compile)"); // queue up all the methods you want to compile
Compiler.command("waitOnCompilationQueue");     // wait until all those methods are compiled
Compiler.disable();                             // turn the compiler off
```

Bestimmungsfunktion bei JNI-Übergängen

JIT generiert standardmäßig optimierten Code für J2N-JNI-Übergänge mit hoher Leistung. Es kommt möglicherweise zu einer reduzierten Bestimmungsfunktion, wenn eine native Bibliothek mit der folgenden Codesequenz erneut geladen wird:

```
RegisterNatives / UnregisterNatives / RegisterNatives
```

Mit der Befehlszeilenoption **-Xjit:disableDirectToJNI** können Sie zum langsameren, mehr deterministischen Code zurückkehren.

JIT-Compiler aktivieren

Sie können JIT auf verschiedene Arten explizit aktivieren. Beide Befehlszeilenoptionen überschreiben die Umgebungsvariable **JAVA_COMPILER**.

Vorgehensweise

- Setzen Sie die Umgebungsvariable **JAVA_COMPILER** auf "jitc", bevor Sie die Java-Anwendung ausführen. Geben Sie einer Shelleingabeaufforderung Folgendes ein:
  - **Für die Korn-Shell:** `export JAVA_COMPILER=jitc`

    Anmerkung: In diesen Informationen werden Korn-Shell-Befehle verwendet, sofern nicht anders angegeben.
  - **Für die Bourne-Shell:**

    ```
    JAVA_COMPILER=jitc
    export JAVA_COMPILER
    ```
  - **Für die C-Shell:** `setenv JAVA_COMPILER jitc`

  Wenn die Umgebungsvariable **JAVA_COMPILER** eine leere Zeichenfolge ist, bleibt der JIT-Compiler inaktiviert. Geben Sie an einer Shelleingabeaufforderung `unset` **JAVA_COMPILER** ein, um die Umgebungsvariable zu inaktivieren.
- Setzen Sie die Eigenschaft java.compiler mit der Option **-D** in der JVM-Befehlszeile auf "jitc". Geben Sie an einer Shelleingabeaufforderung Folgendes ein: `java -Djava.compiler=jitc <MeineAnwendung>`
- Verwenden Sie die Option **-Xjit** in der JVM-Befehlszeile. Sie dürfen die Option **-Xint** *nicht* zur selben Zeit angeben. Geben Sie an einer Shelleingabeaufforderung Folgendes ein: `java -Xjit <MeineAnwendung>`

JIT-Compiler inaktivieren

Sie können JIT auf verschiedene Arten inaktivieren. Beide Befehlszeilenoptionen überschreiben die Umgebungsvariable **JAVA_COMPILER**.

Informationen zu diesem Vorgang

Vorgehensweise

- Setzen Sie die Umgebungsvariable **JAVA_COMPILER** auf NONE oder auf eine leere Zeichenfolge, bevor Sie die Java-Anwendung ausführen. Geben Sie an einer Shelleingabeaufforderung Folgendes ein:
  - **Für die Korn-Shell:** `export JAVA_COMPILER=NONE`

    Anmerkung: Korn-Shell-Befehle werden für den Rest dieser Informationen verwendet.
  - **Für die Bourne-Shell:**

    ```
    JAVA_COMPILER=NONE
    export JAVA_COMPILER
    ```
  - **Für die C-Shell:** `setenv JAVA_COMPILER NONE`

- Verwenden Sie die Option **-D** in der JVM-Befehlszeile, um die Eigenschaft java.compiler auf "NONE" oder auf eine leere Zeichenfolge zu setzen. Geben Sie an einer Shelleingabeaufforderung Folgendes ein: `java -Djava.compiler=NONE <MeineAnwendung>`
- Verwenden Sie die Option **-Xint** in der JVM-Befehlszeile. Geben Sie an einer Shelleingabeaufforderung Folgendes ein: `java -Xint <MeineAnwendung>`

Prüfen, ob der JIT-Compiler aktiviert ist

Sie können den Status des JIT-Compilers mit der Option **-version** ermitteln.

Vorgehensweise

Geben sie Folgendes an einer Shelleingabeaufforderung ein:

```
java -version
```

Wenn der JIT-Compiler derzeit nicht verwendet wird, wird folgende Nachricht angezeigt:
`(JIT disabled)`
Wenn der JIT-Compiler verwendet wird, wird folgende Nachricht angezeigt:
`(JIT enabled)`

No-Heap-Echtzeitthreads verwenden

Die Metronom-Garbage-Collection liefert konsistentere Antwortzeiten, manchmal ist es jedoch angebracht, Unterbrechungen durch die Garbage-Collection vollständig zu vermeiden.

No-Heap-Echtzeitthreads (NHRT-Threads) sind eine Erweiterung von Echtzeitthreads. Sie unterscheiden sich von Echtzeitthreads dadurch, dass sie keinen Zugriff auf den Heapspeicher haben. NHRT-Threads können ohne Zugriff auf den Heapspeicher selbst während eines Garbage-Collection-Zyklus mit einigen Einschränkungen weiterhin ausgeführt werden. Das Programmiermodell unterscheidet sich aufgrund des fehlenden Zugriffs auf den Heapspeicher vom Programmiermodell für Echtzeitthreads.

Zu beachtende Punkte bei der Verwendung von NHRT-Threads

Beachten Sie die folgenden Punkte für NHRT-Threads:

- NHRT-Threads werden in erster Linie für Tasks verwendet, für die eine Garbage-Collection nicht zulässig ist. Dies ist z. B. der Fall, wenn Ihre Anwendung zeitkritisch ist und nicht unterbrochen werden darf.
- Wenn die Zeit so kritisch ist, dass Sie NHRT-Threads verwenden, erwägen Sie auch die Verwendung des AOT-Compilers über die Option **-Xnojit**.
- Wenn Sie die Option **-Xrealtime** verwenden, verwenden Sie automatisch den Metronom-Garbage-Collector. Die Vorteile des Metronom-Garbage-Collectors reichen für Ihr Unternehmen möglicherweise aus, sodass Sie weniger NHRT-Threads codieren müssen.
- NHRT-Threads werden unabhängig vom Garbage-Collector ausgeführt, weil ihre Priorität höher ist als die Priorität des Garbage-Collectors. Java-Threads können eine Priorität im Bereich von 1 bis 10 haben. Wenn NHRT-Threads vorhanden sind, wird die Priorität von Java-Threads unabhängig von der in Ihrem Programm festgelegten Priorität auf 0 zurückgesetzt. Der Garbage-Collector wird automatisch auf einen halben Schritt höher als der höchste Echtzeitthread ge-

setzt. Sie setzen die Priorität Ihrer NHRT-Threads auf mindestens einen Schritt höher als den höchsten Echtzeitthread. Dadurch sind die NHRT-Threads vom Garbage-Collector unabhängig.

Anmerkung: NHRT-Threads sind nicht vollständig von der Garbage-Collection unberührt, weil der Alarmthread des Metronom-Garbage-Collectors mit der höchsten Priorität im System ausgeführt wird. Diese Priorität stellt sicher, dass die JVM aktiviert werden kann, um zu prüfen, ob der Garbage-Collector aktiv werden muss. Die zur Ausführung des Metronomalarmthreads erforderliche Arbeit ist geringfügig und wirkt sich nicht wesentlich auf die Leistung aus. Auf einem Multiprozessorsystem kann der Alarmthread gleichzeitig mit NHRT-Threads ausgeführt werden, weshalb keine Garbage-Collection-Unterbrechungen auftreten.

- Da NHRT-Threads auf die Speicherbereiche für Objekte mit beschränkter und unbeschränkter Lebensdauer eingeschränkt sind, führen Java-Methoden Prüfungen aus, um sicherzustellen, dass sie nicht aus dem Heapspeicher zugeordnet werden. Die Startmethode führt eine Prüfung durch und gibt eine Ausnahmebedingung zurück (`MemoryAccessError`) zurück, wenn NHRT-Threads aus dem Heapspeicher zugeordnet werden. NHRT-Threads können nur auf den Speicher für Objekte mit unbeschränkter Lebensdauer (`ImmortalMemory`) und den Speicher für Objekte mit beschränkter Lebensdauer (`ScopedMemory`) zugreifen.
- Die Sperrsemantik bleibt unverändert, sodass NHRT-Threads von normalen Threads geblockt werden können, wenn eine Sperre gemeinsam genutzt wird.
- Einem Thread, der den Heapspeicher verwendet, kann für eine synchronisierte Methode eine höhere Priorität zugeordnet werden, wenn ein NHRT-Thread versucht, dieselbe Methode zu verwenden.
- Verwenden Sie nicht blockierende Warteschlangen für die Kommunikation zwischen NHRT-Threads und Heapspeicherthreads. Trennen Sie andernfalls die beiden Threadtypen.

Ausnahmebedingungen

Die folgenden Ausnahmebedingungen können bei der Verwendung von NHRT-Threads auftreten:

- `IllegalAssignmentError`. Dieser Fehler kann beispielsweise auftreten, wenn versucht wird, einen Verweis auf den Speicher für Objekte mit beschränkter Lebensdauer im Speicher für Objekte mit unbeschränkter Lebensdauer zu erstellen.
- `MemoryAccessError`. Dieser Fehler kann beispielsweise auftreten, wenn ein NHRT-Thread versucht, auf den Heapspeicher zu verweisen.

Einschränkungen bei der Handhabung von asynchronen Ereignissen

Es gibt mehrere Fälle, wo NHRT-Threads während der Garbage-Collection geblockt werden können:

1. Wenn ein NHRT-Thread fire(), setHandler() oder addHandler() für ein asynchrones Ereignis aufruft, das bereits Handlern zugeordnet ist, die aus dem Heapspeicher zugeordnet sind.
2. Wenn ein NHRT-Thread destroy(), start() oder stop() für einen Zeitgeber aufruft, der bereits Handlern zugeordnet ist, die aus dem Heapspeicher zugeordnet sind.

3. Ein NHRT-Thread verlässt als letzter Thread einen Bereich und beendet Zeitgeber oder asynchrone Ereignisse aus dem Bereich. Den Zeitgebern oder asynchronen Ereignissen wurden jedoch Handler zugeordnet, die aus dem Heapspeicher zugeordnet werden.

Gehen Sie wie folgt vor, um diese Situationen mit NHRT-Threads zu vermeiden:

1. Vermeiden Sie das Hinzufügen von Handlern, die asynchronen Ereignissen oder Zeitgebern, die durch einen NHRT-Thread ausgelöst werden können, aus dem Heapspeicher zugeordnet sind.
2. Vermeiden Sie Bedingungen, unter denen ein NHRT-Thread als letzter Thread einen Bereich verlässt, der asynchrone Ereignisse oder Zeitgeber enthält, denen Handler aus dem Heapspeicher zugeordnet sind.

Einschränkungen für den Speicher und die Planung

Die JVM hindert No-Heap-Echtzeitthreads am Laden von Verweisen auf Objekte, die sich im Heapspeicher befinden, in ihren Operandenstack. Andernfalls würde javax.realtime.MemoryAccessError ausgelöst.

Die JVM sorgt auch dafür, dass Verweise auf Objekte im Speicher für Objekte mit beschränkter Lebensdauer im Heapspeicher oder im Speicher für Objekte mit unbeschränkter Lebensdauer gespeichert werden. Der Speicher für Objekte mit beschränkter Lebensdauer wird zwar nicht exklusiv von NHRT-Threads verwendet, aber er wird wahrscheinlich verwendet, wenn der Speicher für Objekte mit unbeschränkter Lebensdauer ungeeignet ist und Speicherfreigabe in einem NHRT-Kontext erforderlich ist.

Wenn ein NHRT-Thread während seiner Ausführung ein Feld mit einem Verweis auf ein Objekt auffüllt, kann er einen bereits vorhandenen Verweis auf ein Objekt im Heapspeicher in diesem Feld erfolgreich überschreiben. Der bereits vorhandene Verweis wird vom NHRT-Thread erfolgreich überschrieben, ohne einen MemoryAccessError zu generieren.

Einschränkungen beim Laden von Klassen

Klassen werden in dieselben Hauptspeicherbereiche geladen wie das Klassenladeprogramm. Der Standardbereich für Klassenladeprogramme ist der Speicher für Objekte mit unbeschränkter Lebensdauer.

Anwendungen müssen aktiviert sein, damit sie die erwarteten Antwortzeiten liefern. Anwendungen müssen ihre Klassen früh laden, damit das Laden von Klassen Echtzeitthreads und Handler für asynchrone Ereignisse später nicht unterbricht.

Einschränkungen für Java-Threads bei Ausführung mit NHRT-Threads

Da die Systemeigenschaften in einer JVM gemeinsam genutzt werden und jeder beliebige Thread auf die Systemeigenschaften zugreifen kann, muss sorgfältig vorgegangen werden, wenn die Methoden getProperties und setProperties in JVMs verwendet werden, in denen NHRT-Threads ausgeführt werden. NHRT-Threads können nur auf Systemeigenschaften zugreifen, wenn sie sich im Speicher für Objekte mit unbeschränkter Lebensdauer befinden.

Die Klasse `java.lang.System` enthält mehrere Methoden, mit denen Threads mit den Systemeigenschaften interagieren können.

Hierzu gehören die folgenden Methoden:

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

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

Die Echtzeit-JVM verwendet eine Instanz der Klasse com.ibm.realtime.ImmortalProperties, die für das JVM-Echtzeitobjekt zum Speichern aller Systemeigenschaften erstellt wurde. Die Verwendung dieser Instanz stellt sicher, dass alle Aufrufe der Methoden System.setProperty() oder System.getProperties.setProperty() dazu führen, dass die Eigenschaft im Speicher für Objekte mit unbeschränkter Lebensdauer gespeichert wird. In diesem Fall ist kein besonderer Benutzercode erforderlich, aber Sie müssen bedenken, dass bei jeder Festlegung einer Eigenschaft Speicher für Objekte mit unbeschränkter Lebensdauer belegt wird.

Aufrufe der Methode setProperties() sind etwas schwieriger, weil das gemeinsam genutzte Objekt Properties zum Speichern der Systemeigenschaften verwendet wird. Bei der Ausführung einer Anwendung in einer Echtzeit-JVM mit aktiven NHRT-Threads müssen Aufrufe der Methode setProperties an eine Instanz einer Klasse com.ibm.realtime.ImmortalProperties oder Unterklasse übergeben werden, die im Speicher für Objekte mit unbeschränkter Lebensdauer erstellt wurde. Die Verwendung dieser Instanz stellt sicher, dass alle Eigenschaften, die über die Methode setProperties festgelegt werden, im Speicher für Objekte mit unbeschränkter Lebensdauer gespeichert werden.

Anmerkung: Der Aufruf von setProperties(null) führt zur internen Erstellung eines neuen ImmortalProperties-Objekts mit einer Standardgruppe von Eigenschaften. Hierdurch wird zusätzlicher Speicher für Objekte mit unbeschränkter Lebensdauer belegt.

Aufrufe der Methode getProperties() geben das festgelegte Objekt oder das Standardeigenschaftenobjekt zurück, das ein com.ibm.realtime.ImmortalProperties-Objekt ist. Das ImmortalProperties-Objekt serialisiert das Objekt und deserialisiert es dann in einer Standard-JVM, um die Kompatibilität mit vorhandenem Code zu maximieren, der die Methode getProperties() aufruft. Das Standardverhalten für die Serialisierung von ImmortalProperties ist die Serialisierung eines regulären Properties-Objekts, weil Standard-JVMs nicht über das ImmortalProperties-Objekt verfügen und die Deserialisierung fehlschlägt. Die Klasse ImmortalProperties stellt zum Überschreiben dieses Standardverhaltens die Methode enabledReplacement-(boolean) bereit, die das Standardverhalten beim Aufruf mit 'false' inaktiviert. In diesem Fall serialisiert die Serialisierung das ImmortalPropeties-Objekt, das daraufhin deserialisiert werden kann. Das sich ergebende Objekt kann dann in einem Aufruf der Methode System.setProperties in einer Echtzeit-JVM verwendet werden.

Anmerkung: Die Deserialisierung findet im Speicher für Objekte mit unbeschränkter Lebensdauer statt, wodurch diese begrenzte Ressource zu stark ausgelastet wird.

Sicherheitsmanager

Der für das System festgelegte Sicherheitsmanager wird von allen Threadtypen in der JVM verwendet. Daher muss der Sicherheitsmanager in einer Echtzeit-JVM, in der NHRT-Threads ausgeführt werden, im Speicher für Objekte mit unbeschränkter Lebensdauer zugeordnet werden. Die Echtzeit-JVM stellt sicher, dass jeder in den Befehlszeilenoptionen angegebener Sicherheitsmanager im Speicher für Objekte mit unbeschränkter Lebensdauer zugeordnet wird. Der Sicherheitsmanager kann auch über Aufrufe der Methode System.setSecurityManager(SecurityManager) festgelegt werden. Wenn die Anwendung den Sicherheitsmanager so festlegt, muss sie sicherstellen, dass der Sicherheitsmanager aus dem Speicher für Objekte mit unbeschränkter Lebensdauer zugeordnet wurde, damit NHRT-Threads ordnungsgemäß ausgeführt werden können.

Die ausgelösten Ausnahmebedingungen und alle vom Sicherheitsmanager zurückgegebenen Objekte müssen sich im Speicher für Objekte mit unbeschränkter Lebensdauer befinden (sofern sie zwischengespeichert sind) oder im aktuellen Zuordnungskontext zugeordnet werden.

Synchronisation

Die Klasse MonitorControl und ihre Unterklasse PriorityInheritance verwalten die Synchronisation, vor allem die Steuerung der Prioritätsumkehrung. Diese Klassen ermöglichen die Festlegung einer Steuerungsrichtlinie für Prioritätsumkehrung als Standard oder für bestimmte Objekte.

Die Klassen WaitFreeReadQueue, WaitFreeWriteQueue und WaitFreeDequeue ermöglichen wartefreie Kommunikation zwischen planbaren Objekten (vor allem Instanzen von NoHeapRealtimeThread) und regulären Java-Threads.

Die WaitFree-Klassen bieten in Abhängigkeit von Garbage-Collection-Verzögerungen sicheren gleichzeitigen Zugriff auf Daten, die von NoHeapRealtimeThread und planbaren Objekten gemeinsam genutzt werden.

Sicherheit der No-Heap-Echtzeitklassen

Unter einigen Umständen können Teile der JSE-API nicht notwendigerweise in einem No-Heap-Kontext verwendet werden. Für Klassen, die von Heapspeicher- und No-Heap-Threads gemeinsam genutzt werden, gelten gewisse Einschränkungen. Sie müssen die mit der JVM gelieferten Klassen kennen, die problemlos verwendet werden können.

Objekte gemeinsam nutzen

In No-Heap-Echtzeitthreads ausgeführte Methoden lösen einen javax.realtime.MemoryAccessError aus, wenn sie versuchen, einen Verweis auf ein Objekt in einem Heapspeicher zu laden.

Abb. 3 auf Seite 65 ist ein Beispiel für die Art von Code, die vermieden werden muss:

```
/**
 * NHRTError1
 *
 * This example is a simple demonstration of an NHRT accessing
 * a heap object reference.
 *
 * The error generated is:
 *
 * 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; /* The NHRT can access static fields directly - they are always Immortal. */
    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( );
        }

    }
```

Abbildung 3. Beispiel eines NHRT-Threads, der auf einen Heapspeicherobjektverweis zugreift

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

        /* Prints the String via the static reference in NHRTError1.message */
        public void run() {
            System.out.println("Nachricht: " + NHRTError1.message);
        }
    }

}
```

Abbildung 4. Beispiel eines NHRT-Threads, der auf einen Heapspeicherobjektverweis zugreift (Fortsetzung von Abbildung 1)

Abb. 3 erzeugt einen **javax.realtime.MemoryAccessError**:

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

Wenn ein Objekt für einen No-Heap-Echtzeitthread und einen Standard-Java-Thread zugänglich sein soll, muss es im Speicher für Objekte mit unbeschränkter Lebensdauer zugeordnet werden. Wenn ein Objekt für einen No-Heap-Echtzeitthread und einen Echtzeitthread zugänglich sein soll, kann es in einem Speicherbereich für Objekte mit beschränkter Lebensdauer angeordnet sein.

In Abb. 3 auf Seite 65 befindet sich der Verweis auf die Zeichenfolge "This on the heap." in einer Klassenvariable. NHRT-Threads können auf diese Variable zugreifen, weil alle Klassen im Speicher für Objekte mit unbeschränkter Lebensdauer zugeordnet werden. Die Zeichenfolge hätte auch an den Konstruktor der NHRT-Threads übergeben werden können.

Die meisten Objekte enthalten Verweise auf andere Objekte. Daher müssen Sie sorgfältig vorgehen, wenn derartige Objekte von gewöhnlichen Threads und NHRT-Threads gemeinsam genutzt werden. Ein typisches Beispiel ist eine im Speicher für Objekte mit unbeschränkter Lebensdauer zugeordnete verkettete Liste (LinkedList), die von einem gewöhnlichen Thread und NHRT-Threads gemeinsam genutzt wird. Wenn Sie nicht sorgfältig genug sind, führt der Standardthread möglicherweise Objekte in die LinkedList ein, die sich im Heapspeicher befinden. Die Datenstrukturen, die von der LinkedList zum Protokollieren von Objekten zugeordnet werden, werden durch den gewöhnlichen Thread im Heapspeicher zugeordnet, was im NHRT-Thread häufig einen MemoryAccessError verursacht.

Einige Klassen können nicht problemlos von NHRT-Threads und anderen Threads gemeinsam genutzt werden, unabhängig davon, wo einzelne ihrer Instanzen zugeordnet sind. Diese Klassen hängen von Objekten ab, die gewöhnlich zu Caching-Zwecken in Klassenvariablen gespeichert sind. InetAddress ist ein typisches Beispiel, das Adressen in den Cache stellt. Wenn der erste Thread zum Aufrufen bestimmter Methoden in InetAddress im Heapspeicher ausgeführt wird, ist es nicht sicher, dieselben Methoden zukünftig von NHRT-Threads aufrufen zu lassen.

Sperrung für Objekte mit NHRT-Threads

NHRT-Threads dürfen nicht mit anderen Threads synchronisiert werden. Betrachten Sie das folgende Szenario:

- Ein Echtzeitthread mit einer niedrigen Priorität greift auf einen synchronisierten Block bzw. eine synchronisierte Methode zu und wird mit einem Objekt synchronisiert.
- Ein NHRT-Thread mit einer hohen Priorität wird geblockt, wenn versucht wird, eine Synchronisation für dieselben Objekte durchzuführen.
- Aufgrund der Prioritätsübernahme erwirbt der Echtzeitthread vorübergehend dieselbe Priorität wie der NHRT-Thread.
- Die Garbage-Collection wird dann mit einer höheren Priorität ausgeführt als der NHRT-Thread und kann daher den NHRT-Thread unterbrechen. Der NHRT-Thread wird verwendet, um eine Unterbrechung durch die Garbage-Collection zu vermeiden, was bei diesem Szenario nicht der Fall ist.

Manchmal kann es nicht vermieden werden, dass NHRT-Threads und andere Threads für dasselbe Objekt synchronisiert werden, Sie müssen die Möglichkeit jedoch minimieren. Vermeiden Sie jede unnötige Synchronisation, wenn Sie Objekte gemeinsam nutzen.

Einschränkungen für sichere Klassen

Sie müssen einige Aspekte berücksichtigen, wenn eine Anwendung Echtzeit- und No-Heap-Echtzeitthreadobjekte enthält.

- Im No-Heap-Echtzeitthread können aufgrund der Interaktion mit Echtzeitthreads MemoryAccessErrors auftreten.
- Der No-Heap-Echtzeitthread kann durch die vom Echtzeitthread verursachte Garbage-Collection versehentlich verzögert werden.

In einem No-Heap-Echtzeitthread verursachte MemoryAccessErrors

Wenn die beiden Threadtypen Methoden in derselben Klasse aufrufen, kann der Echtzeitthread die statischen Variablen der Klasse mit aus dem Heapspeicher zugeordneten Objekten „verschmutzen". Der No-Heap-Echtzeitthread empfängt einen MemoryAccessError, wenn er versucht, auf diese Heapspeicherobjekte zuzugreifen. Die Verschmutzung kann auch in Instanzen der Klasse geschehen. Beide Probleme treten wahrscheinlich in typischen Codierungsmustern auf und werden in den beiden folgenden Fällen illustriert.

Wenn eine Klasse eine zeitaufwendige Operation ausführt, stellt sie das Ergebnis häufig in den Cache, um die Leistung der nachfolgenden Operationen zu steigern. Der Cache ist gewöhnlich eine Objektgruppe wie eine Hashzuordnung, die in einer statischen Variable in der Klasse verankert ist. Ein im Heapspeicherkontext fungierender Echtzeitthread kann ein Heapspeicherobjekt in dieser Objektgruppe speichern. Hierdurch wird der Objektgruppe nicht nur das Objekt, sondern werden auch Infrastrukturobjekt hinzugefügt, z. B. Teile des Index. Wenn ein No-Heap-Echtzeitthread später versucht, auf die Objektgruppe, jedoch nicht auf das vom anderen Thread hinzugefügte Objekt zuzugreifen, versucht er, die Infrastrukturobjekte zu laden, und empfängt einen MemoryAccessError. Durch die Weiterentwicklung von Klassenbibliotheken und ihre stetige Leistungsoptimierung werden diese Caches immer gängiger.

Eine Klasseninstanz kann auf verschiedene Arten auch durch Heapspeicherobjekte verschmutzt werden. Angenommen, eine Instanz ist in den Speicher für Objekte mit unbeschränkter Lebensdauer integriert und ist daher für beide Threadtypen zugänglich. Wenn der Echtzeitthread im Heapspeicherkontext das Objekt zuerst verwendet, kann ein Sekundärobjekt in einem Feld des ursprünglichen Objekts gespeichert werden. Wenn sich das Sekundärobjekt im Heapspeicherkontext befindet, führt die nachfolgende erneute Verwendung durch den No-Heap-Echtzeitthread zu einem MemoryAccessError. Diese Sekundärobjekte werden nicht immer bei der Erstverwendung, aber nach eine Anzahl Verwendungen hinzugefügt. Sie wurden möglicherweise für die Leistungssteigerung von stark verwendeten Methoden konzipiert.

Durch die Garbage-Collection verzögerter No-Heap-Thread

No-Heap-Threads müssen Prioritäten zugeordnet werden, die höher sind als die Prioritäten anderer Threads, um ihre Verzögerung durch die Garbage-Collection zu vermeiden.

Wenn außerdem eine Klasse synchronisierte Methoden enthält, ist es möglich, dass ein No-Heap-Echtzeitthread, der derartige Methoden aufruft, durch die Garbage-Collection versehentlich verzögert wird. Dieses Szenario wird in „Sperrung für Objekte mit NHRT-Threads" auf Seite 66 beschrieben.

Wenn eine Klasse synchronisierte Methoden (statische Methoden oder Instanzdefinitionsmethoden) enthält, ist es möglich, dass ein No-Heap-Echtzeitthread, der derartige Methoden aufruft, durch die Garbage-Collection versehentlich verzögert wird. Das Problem tritt auf, wenn ein Echtzeitthread auf eine synchronisierte Methode zugreift, während ein No-Heap-Echtzeitthread versucht, auf eine weitere

synchronisierte Methode zuzugreifen, die das Warten auf den Abschluss des anderen Threads blockiert. Wenn der No-Heap-Echtzeitthread eine höhere Priorität hat als der Echtzeitthread, wird die Priorität des Echtzeitthreads erhöht. Wenn dieser Thread dann gezwungen wird, auf einen Garbage-Collection-Interrupt zu warten, ist eine Prioritätsumkehrung möglich, weil der Garbage-Collector-Thread eine höhere Priorität hat als der Echtzeitthread mit der höchsten Priorität. Diese Priorität ist möglicherweise nicht so hoch wie die Priorität des zurzeit blockierten No-Heap-Echtzeitthreads, der auf Zugriff auf die synchronisierte Methode wartet.

Diese Probleme können nur behoben werden, indem Sie sicherstellen, dass No-Heap-Echtzeitthreads nie synchronisierte Methoden in Klassen oder Instanzen aufrufen, die mit anderen Threadtypen gemeinsam genutzt werden. Leider kann aus einer Methodensignatur nicht immer abgelesen werden, ob eine Methode synchronisiert ist. Sie kann z. B. einen synchronisierten Block enthalten oder eine synchronisierte Methode aufrufen.

Zusammenfassung

Die Klasse NoHeapRealtimeThread erhöht die Komplexität der Echtzeitumgebung wesentlich und es kann zu vielen Problemen kommen, wenn eine Mischung von Threadtypen in einer Umgebung fungieren. Während der Entwicklung einer Anwendung müssen Sie sorgfältig Bereiche entwerfen, in denen die verschiedenen Threadtypen Klassen gemeinsam nutzen können. Besonders wichtig ist die Verwendung von Klassen im SDK durch diese Threads. Aufgrund der Analysekomplexität kann nicht garantiert werden, dass alle im SDK bereitgestellten Klassen für die gemeinsame Nutzung sicher sind. Allerdings wurde eine kleine Untergruppe der Klassen geprüft. Die Prüfung bezog sich auf den MemoryAccessError-Aspekt. Das Ergebnis ist eine Liste von Klassen, die analysiert, getestet und bei Bedarf modifiziert wurden, um sicherzustellen, dass sie von No-Heap-Threads und anderen Typen von Threads verwendet werden können.

Sichere Klassen

In diesem Abschnitt wird die Gruppe von Klassen aufgelistet, die von NoHeapRealtimeThread und anderen Threadtypen problemlos verwendet werden können.

Am wichtigsten ist der MemoryAccessError-Aspekt der Sicherheit. In der folgenden Liste werden die Klassen aufgeführt, die von allen drei Threadtypen in derselben JVM verwendet werden können.

Anmerkung: Einzelne Instanzen der Klasse können möglicherweise nicht immer problemlos gemeinsam genutzt werden.

Beachten Sie die folgenden Regeln, um sicherzustellen, dass eine Klasse von allen Threadtypen sicher verwendet werden kann:

- Die Instanz muss in einem Hauptspeicherbereich erstellt werden, der für den Thread, der auf die Instanz zugreifen will, zugänglich ist.
- Wenn die Klasse über öffentliche statische Felder verfügt, vermeiden Sie das Speichern von Heapspeicherobjekten in diesen Feldern.
- Wenn die Klasse über öffentliche Instanzfelder verfügt, vermeiden Sie das Speichern von Heapspeicherobjekten in diesen Feldern.

Nicht alle von IBM gelieferten Klassen sind NHRT-sicher. Die folgenden Pakete enthalten NHRT-sichere Klassen:

- Paket 'java.lang'
- Paket 'java.lang.reflect'

- Paket 'java.lang.ref' (alle Klassen)
- Paket 'java.net'
- Paket 'java.io'
- Paket 'java.math'

Die folgenden Tabellen zeigen Klassen in diesen Paketen, die nicht NHRT-sicher sind:

Tabelle 5. Klassen im Paket 'java.lang', die nicht NHRT-sicher sind

Class (Klasse)	Method (Methode)
java.lang.ProcessBuilder	*
java.lang.Thread	getAllStackTraces()Ljava.util.Map;
java.lang.ThreadGroup	*
java.lang.ThreadLocal	*
java.lang.InheritableThreadLocal	*

Tabelle 6. Klassen im Paket 'java.lang.reflect', die nicht NHRT-sicher sind

Class (Klasse)	Method (Methode)
java.lang.reflect.Proxy.*	*

Tabelle 7. Klassen im Paket 'java.net', die nicht NHRT-sicher sind

Class (Klasse)	Method (Methode)
java.net.SocketPermission.*	newPermissionCollection()Ljava.net.SocketPermissionCollection;

Tabelle 8. Klassen im Paket 'java.io', die nicht NHRT-sicher sind

Class (Klasse)	Method (Methode)
java.io.ExpiringCache	*
java.io.SequenceInputStream	*
java.io.FilePermission	newPermissionCollection()Ljava.io.FilePermissionCollection;
java.io.ObjectInputStream	*
java.io.ObjectOutputStream	*
java.io.ObjectStreamClass	*

Tabelle 9. Klassen im Paket 'java.math', die nicht NHRT-sicher sind

Class (Klasse)	Method (Methode)
java.math.BigInteger	*

Die Pakete enthalten möglicherweise Unterpakete, die nicht sichere Klassen enthalten. Beispielsweise sind die folgenden Klassen nicht NHRT-sicher:
- java.lang.management.*
- java.lang.annotation.*
- java.lang.instrument.*

Selbst wenn eine Klasse als NHRT-sicher eingestuft wird, eignet sie sich eventuell nicht für die Verwendung in einem NHRT-Thread. Anwendungsentwickler müssen die Echtzeitanforderungen der Klassen unabhängig davon, ob eine Klasse NHRT-sicher ist, individuell ermitteln.

Gemeinsame Nutzung von Klassendaten zwischen JVMs

Java Virtual Machine (JVM) ermöglicht Ihnen die gemeinsame Nutzung von Klassendaten auf verschiedenen JVMs, indem die Daten in eine im Speicher abgelegte Cachedatei auf Platte gestellt werden.

Durch die gemeinsame Nutzung verringert sich die gesamte virtuelle Speicherbelegung, wenn mehrere JVMs einen Cache gemeinsam nutzen. Außerdem verkürzt sich durch die gemeinsame Nutzung der Systemstart von JVM, nachdem der Cache erstellt wurde. Der Cache für gemeinsam genutzte Klassen ist unabhängig von aktiven JVMs und bleibt persistent, bis er gelöscht wird. Ein gemeinsam genutzter Cache kann Folgendes enthalten:

- Bootprogrammklassen
- Anwendungsklassen
- Metadaten, die die Klassen beschreiben
- Kompilierter AOT-Code (Ahead-of-time)

Caches für gemeinsam genutzte Klassen können von IBM WebSphere Real Time for RT Linux im Nicht-Echtzeitmodus und Echtzeitmodus verwendet werden, das Cacheformat, die Erstellung und die Füllverfahren sind jedoch unterschiedlich. Caches im Echtzeitmodus sind nicht kompatibel mit Caches im Nicht-Echtzeitmodus. Im Nicht-Echtzeitmodus werden Caches auf dieselbe Weise wie bei der Standard-JVM erstellt und aufgefüllt. Dies bedeutet, dass der Cache von der JVM während der Ausführung einer Anwendung auf für den Benutzer transparente Weise erstellt und aufgefüllt wird. Bei der Verwendung der Option **-Xrealtime** im Echtzeitmodus müssen Caches für gemeinsam genutzte Klassen von **admincache** unter Verwendung der Option **-populate** erstellt und vorab aufgefüllt werden. Anwendungen, die im Echtzeitmodus ausgeführt werden, lesen möglicherweise Inhalt aus dem vorab aufgefüllten Cache, können diesen Inhalt jedoch nicht ändern.

Mit dem Tool **admincache** können Sie Caches erstellen, auffüllen und löschen.

Fügen Sie der Befehlszeile die Option **-Xshareclasses** hinzu, um eine Anwendung für die Verwendung eines Cache für gemeinsam genutzte Klassen zu aktivieren. Da Caches im Echtzeitmodus schreibgeschützt sind, sind einige Unteroptionen des Nicht-Echtzeitmodus von **-Xshareclasses** im Echtzeitmodus nicht verfügbar.

Weitere Informationen finden Sie in „Tool 'admincache' verwenden" auf Seite 43, „Gemeinsame Nutzung von Klassendaten auf verschiedenen JVMs im Nicht-Echtzeitmodus" auf Seite 101 und „Diagnose bei gemeinsam genutzten Klassen" auf Seite 152.

Anwendungen mit einem Cache für gemeinsam genutzte Klassen ausführen

Sie führen eine Anwendung mit einem Cache für gemeinsam genutzte Klassen über die Option **-Xshareclasses** in der Befehlszeile aus.

Tabelle 10 auf Seite 71 zeigt die Unteroptionen an, die bei der Ausführung einer Anwendung im Echtzeitmodus mit der Option **-Xshareclasses** verfügbar sind.

Tabelle 10. Bei der Ausführung einer Anwendung im Echtzeitmodus verfügbare Unteroptionen

Option	Bedeutung
`cacheDir=<Verzeichnis>`	Legt das Verzeichnis fest, aus dem Daten des Cache für gemeinsam genutzte Klassen gelesen werden bzw. in das diese Daten geschrieben werden. <*Verzeichnis*> ist standardmäßig `/tmp/javasharedresources`. Der Verzeichnisname muss mit dem Verzeichnisnamen übereinstimmen, der in der Option **-cacheDir** des Befehls 'admincache' zum Erstellen des Cache angegeben wurde.
`name=<Name>`	Der Name des zu verwendenden Cache für gemeinsam genutzte Klassen. Der Name muss mit dem Namen übereinstimmen, der in der Option **-cacheName** des Befehls 'admincache' zum Erstellen des Cache angegeben wurde. Der Name darf nicht mehr als 53 Zeichen umfassen.
`none`	Inaktiviert explizit die gemeinsame Nutzung von Klassen. Kann dem Ende einer Befehlszeile hinzugefügt werden, um die gemeinsame Nutzung von Klassendaten zu inaktivieren. Diese Unteroption überschreibt Argumente der gemeinsamen Nutzung von Klassen, die in einem früheren Stadium in der Befehlszeile enthalten waren.
`nonfatal`	Startet die JVM unabhängig von Fehlern oder Warnungen immer. Ermöglicht das Starten der JVM, auch wenn die gemeinsame Nutzung von Klassendaten fehlschlägt. Das ntypische Verhalten der JVM wäre, bei Fehlschlagen der gemeinsamen Nutzung von Klassendaten nicht zu starten. Wenn **nonfatal** ausgewählt ist und die Initialisierung des Cache für gemeinsam genutzte Klassen fehlschlägt, versucht die JVM, die Verbindung zum Cache im Nur-Lese-Modus herzustellen. Schlägt dieser Versuch fehl, startet die JVM ohne gemeinsame Nutzung der Klassendaten.
`silent`	Unterdrückt alle Ausgabenachrichten. Inaktiviert alle Nachrichten der gemeinsam genutzten Klassen, einschließlich der Fehlernachrichten. Nicht behebbare Fehler, die die Initialisierung der JVM verhindern, werden allerdings angezeigt.
`verbose`	Aktiviert die ausführliche Ausgabe, mit der der Gesamtstatus des Cache für gemeinsam genutzte Klassen und ausführlichere Fehlernachrichten bereitgestellt werden.
`verboseAOT`	Aktiviert die ausführliche Ausgabe, wenn kompilierter AOT-Code im Cache gefunden wird, z. B. bei AOT-Methodenladeanforderungen.

Tabelle 10. Bei der Ausführung einer Anwendung im Echtzeitmodus verfügbare Unteroptionen (Forts.)

Option	Bedeutung
`verboseHelper`	Aktiviert die ausführliche Ausgabe für die Java-Helper-API. Diese Ausgabe zeigt Ihnen, wie die Helper-API von Ihrem Klassenladeprogramm verwendet wird.
`verboseIO`	Aktiviert die ausführliche Ausgabe von Klassenladeanforderungen. Diese Option liefert eine ausführliche Ausgabe zu E/A-Aktivitäten im Cache, wobei Informationen zu gefundenen Klassen aufgelistet werden.

Verwenden Sie 'admincache' mit der Option **-printvmargs**, um sicherzustellen, dass diese Optionen richtig sind. (Weitere Informationen finden Sie in **-printvmargs**). Die Option **nonfatal** eignet sich nicht für die allgemeine Verwendung, weil sie die JVM zwingt, Warnungen und Fehler zum Cache für gemeinsam genutzte Klassen zu ignorieren. Die Option **none** inaktiviert die gemeinsame Klassennutzung explizit und entspricht dem Auslassen der Option **-Xshareclasses** in der Befehlszeile.

Detailliertere Informationen zu den Unteroptionen von **-Xshareclasses** finden Sie in Befehlszeilenoptionen für gemeinsame Klassendatennutzung.

Metronom-Garbage-Collector verwenden

Der Metronom-Garbage-Collector ersetzt den Standard-Garbage-Collector in WebSphere Real Time for RT Linux.

Prozessorauslastung steuern

Sie können die Menge der dem Metronom-Garbage-Collector zur Verfügung stehenden Verarbeitungskapazität begrenzen.

Sie können die Garbage-Collection mit dem Metronom-Garbage-Collector unter Verwendung der Option **-Xgc:targetUtilization=N** steuern. Mit dieser Option können Sie die vom Garbage-Collector verwendete CPU-Kapazität begrenzen.

Beispiel:

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

Das Beispiel gibt an, dass Ihre Anwendung in jedem 60-Millisekunden-Intervall 80 % der Zeit belegt. Die verbleibenden 20 % der Zeit werden für die Garbage-Collection verwendet. Der Metronom-Garbage-Collector garantiert bestimmte Auslastungen, vorausgesetzt ihm wurden ausreichende Ressourcen zugeteilt. Die Garbage-Collection fängt an, wenn der freie Speicherplatz im Heapspeicher unter einen dynamisch ermittelten Schwellenwert fällt.

Metronom-Garbage-Collector optimieren

Sie können die Echtzeitumgebung optimieren, indem Sie die Speicherkapazität steuern, die Ihre Anwendung verwendet. Verwenden Sie z. B. die Optionen **-Xmx**, **-Xgc:immortalMemorySize=Größe**, **-Xgc:scopedMemoryMaximumSize=Größe** und **-Xgc:targetUtilization=N**.

- Mit der Option **-Xmx** können Sie die Größe des Heapspeichers begrenzen.

  Der ausgewählte Wert wird als Obergrenze der Größe des Heapspeichers verwendet und spiegelt daher die wahrscheinliche Belegung über einen bestimmten

Zeitraum wider. Die Auswahl eines zu niedrigen Werts erhöht die Garbage-Collection-Häufigkeit und führt zu einem niedrigeren Gesamtdurchsatz und Speicherbedarf. Vermeiden Sie Paging, um eine gute Echtzeitleistung zu erzielen. Stellen Sie sicher, dass der Speicherbedarf aller aktiven Prozesse auf einer Maschine die Größe des physischen Hauptspeichers nicht überschreitet.

- Mit der Option **-Xgc:immortalMemorySize=Größe** können Sie die Größe des Speicherbereichs für Objekte mit unbeschränkter Lebensdauer steuern.

  Sie müssen die Belegung des Speichers für Objekte mit unbeschränkter Lebensdauer sorgfältig analysieren. Die „ideale" Anwendung verwendet während des Starts den Speicher für Objekte mit unbeschränkter Lebensdauer, danach jedoch nicht mehr. Wenn die Zuordnung von Objekten mit unbeschränkter Lebensdauer fortgesetzt wird, kann die Anwendung weiterhin ausgeführt werden, bis kein Speicher für Objekte mit unbeschränkter Lebensdauer mehr verfügbar ist. Die aktuelle Belegung kann durch das Hinzufügen von Folgendem zu Ihrem Code abgerufen werden:

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

- Mit der Option **-Xgc:scopedMemoryMaximumSize=Größe** können Sie sicherstellen, dass Anwendungen keine übermäßige Speicherkapazität für Objekte mit beschränkter Lebensdauer anfordern. Verwenden Sie diese Option für die Diagnose anstatt für die Optimierung.
- Legen Sie die Option **-Xgc:targetUtilization=N** fest, um sicherzustellen, dass der Garbage-Collector unter den schlechtesten Bedingungen (maximale Zuordnungsrate von Heapspeicherobjekten) Garbage mit einer höheren Rate erfassen kann, als die Anwendung ihn generiert.

  Der Standardwert reicht gewöhnlich aus, die Anwendungsleistung kann jedoch möglicherweise gesteigert werden, indem die Auslastung bis zu dem Punkt erhöht wird, an dem der Collector Garbage etwas schneller erfassen kann, als die Anwendung ihn erstellen kann.
- Mit der Option **-Xgcthreads <n>** können Sie zusätzliche Threads zur parallelen Ausführung der Garbage-Collection erstellen.

  Standardmäßig wird ein Thread verwendet. Wenn Ihre Workload eine hohe Garbagegenerierungsrate hat und auf einem symmetrischen Multiprozessor mit verfügbaren CPU-Zyklen ausgeführt wird, kann die Leistung von dem Setzen dieses Parameters auf >1 profitieren.

  Anmerkung: Wenn Sie diesen Parameter zu hoch festlegen, wird der Durchsatz negativ beeinflusst.

Einschränkung für den Metronom-Garbage-Collector

In seltenen Fällen kann es während der Garbage-Collection zu längeren Pausen als erwartet kommen.

Während der Garbage-Collection wird ein Root-Scan-Vorgang verwendet. Der Garbage-Collector führt für den Heapspeicher eine Walk-Operation durch und startet mit bekannten zeitnahen Verweisen. Diese Verweise können folgender Art sein:

- Zeitnahe Verweisvariablen in den aktiven Thread-Aufruf-Stacks
- Statische Verweise
- Alle Objektverweise in den Speichern für Objekte mit unbeschränkter und beschränkter Lebensdauer

Der Garbage-Collector durchsucht alle Stack-Frames im Aufrufstack des Anwendungsthreads, um alle Verweise auf Liveobjekte im Stack dieses Threads zu finden. Jeder aktive Thread-Stack wird in einem unterbrechungsfreien Schritt durchsucht. Daher muss der Suchlauf innerhalb einer einzelnen GC-Pause stattfinden.

Die Systemleistung ist aufgrund von erweiterten Garbage-Collection-Pausen am Anfang eines Erfassungszyklus möglicherweise schlechter als erwartet, wenn einige Threads mit sehr tiefen Stacks vorliegen.

Der Speicher für Objekte mit unbeschränkter Lebensdauer wird inkrementell verarbeitet. Alle anderen Speicherbereiche für Objekte mit beschränkter Lebensdauer werden in einem atomaren unterbrechungsfreien Schritt verarbeitet. Daher führt eine signifikante Verwendung von Speicherbereichen für Objekte mit beschränkter Lebensdauer aufgrund von erweiterten Garbage-Collection-Pausen bei der Verarbeitung des Speichers für Objekte mit beschränkter Lebensdauer durch den Rootsuchlauf möglicherweise zu schlechterer Systemleistung als erwartet.

Kapitel 6. Anwendungen entwickeln

Wichtige Informationen zum Schreiben von Echtzeitanwendungen, einschließlich Codemuster.

Java-Anwendungen zum Nutzen der Echtzeit schreiben

Die folgenden Beispiele beschreiben, wie Sie die Echtzeitumgebung nutzen können. Sie reichen von einem einfachen Beispiel (Ausführung einer Java-Anwendung in Echtzeit ohne Codeänderungen) bis zu dem komplexeren Prozess für das Planen und Schreiben von No-Heap-Echtzeitthreads. Ihnen werden Gründe genannt, die Ihnen bei der Entscheidungsfindung für den besten Ansatz für Ihre Anwendung helfen sollen.

Einführung in das Schreiben von Echtzeitanwendungen

Sie brauchen keine komplizierten No-Heap-Echtzeitanwendungen zu schreiben, um die Features der Echtzeittechnologie nutzen zu können. Einige Vorteile können mit geringfügiger Änderung Ihres vorhandenen Codes verwendet werden.

Im Folgenden werden für Anwendungsprogrammierer die Schritte aufgelistet, mit denen Sie WebSphere Real Time for RT Linux optimaler nutzen können:

1. Sie können eine Standard-Java-Anwendung in einer Echtzeit-JVM ausführen, um die Metronom-Garbage-Collection zu nutzen und eine wesentliche Verbesserung bei der Vorhersagbarkeit der Ausführungszeit Ihrer Anwendung zu erzielen.
2. Fügen Sie die Option **-Xnojit** hinzu, nachdem Sie Ihren Code für die Verwendung mit dem AOT-Compiler vorkompiliert haben. Informationen hierzu finden Sie in „Vorkompilierte JAR-Dateien in einem Cache für gemeinsam genutzte Klassen speichern" auf Seite 55.
3. Ersetzen Sie java.lang.Thread durch javax.realtime.RealtimeThread in Ihrer Anwendung. Im Vergleich zur AOT-Option stellen Sie möglicherweise eine leichte Verbesserung fest.

   Die Verwendung von Echtzeitthreads hat den Hauptvorteil, dass Sie die Priorität steuern können, die Sie jedem einzelnen Thread zuordnen. Echtzeitthreads können auch periodisch sein. Sie müssen Änderungen an der Anwendung vornehmen, um diese Vorteile nutzen zu können.
4. Planen und schreiben Sie eine Anwendung, um Echtzeitthreads und Handler für asynchrone Ereignisse zum Verarbeiten von Zeitgebern oder externen Ereignissen zu verwenden. Beachten Sie die folgenden drei Faktoren:
   - Planung der Priorität, die Sie Ihren Echtzeitthreads zuordnen
   - Wahl der Hauptspeicherbereiche zum Speichern von Objekten
   - Kommunikation mit den Ereignishandlern
5. Planen und schreiben Sie eine Anwendung, um No-Heap-Echtzeitthreads zu verwenden. No-Heap-Echtzeitthreads sind Erweiterungen von Echtzeitthreads und Sie müssen die Priorität, die Sie zuordnen, sowie den Hauptspeicherbe-

reich auswählen. Im Allgemeinen sollten Sie diesen Schritt nur ausführen, wenn die Anwendung Ereignisse in Zeiträumen verarbeiten muss, die mit den GC-Pausezeiten (Submillisekunden) vergleichbar sind. Unterschätzen Sie nicht die Komplexität von Entwicklungen mit No-Heap-Echtzeitthreads.

Abb. 5 zeigt die zuvor beschriebenen Schritte.

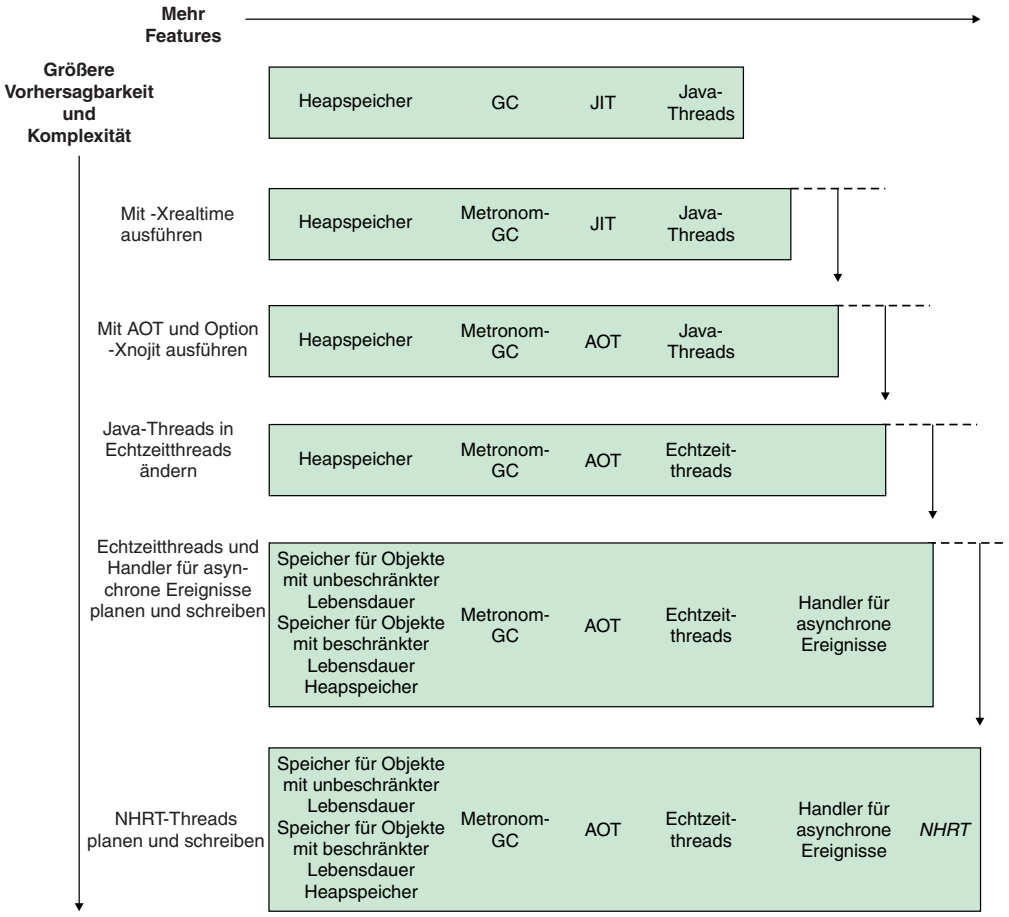

Abbildung 5. Vergleich der RTSJ-Features mit erhöhter Vorhersagbarkeit

WebSphere Real Time for RT Linux-Anwendung planen

Wenn Sie das Schreiben von Java-Echtzeitanwendungen vorbereiten, müssen Sie erwägen, ob Sie Java-Threads, Echtzeitthreads oder No-Heap-Echtzeitthreads verwenden wollen. Sie können außerdem entscheiden, welchen Hauptspeicherbereich Ihre Threads verwenden.

Informationen zu diesem Vorgang

Bei der Planung Ihrer Anwendung beschreiben die folgenden Schritte die Entscheidungen, die Sie treffen müssen:

Vorgehensweise

1. Ermitteln Sie Ihre Tasks.
2. Legen Sie die Ablaufsteuerungszeiträume fest:

- Wählen Sie bei Antworten, die länger als 10 ms sind, Java-Threads aus, wodurch Sie nur den Metronom-Garbage-Collector nutzen.

  Diese Threads verwenden nur den Heapspeicher für Speicherung. Der Nachteil ist, dass die Garbage-Collection Ihre Anwendung unterbricht. Da sie jedoch vom Metronom-Garbage-Collector gesteuert wird, sind die Dauer und Ablaufsteuerung der Unterbrechungen vorhersehbar.
- Wählen Sie bei Antworten, die kürzer als 10 ms sind, Echtzeitthreads aus.

  Echtzeitthreads können im Heapspeicher, Speicher für Objekte mit beschränkter Lebensdauer oder Speicher für Objekte mit unbeschränkter Lebensdauer angeordnet werden. Die Vorteile der Verwendung von Echtzeitthreads sind:
  - Sie können mit höherer Priorität als Standard-Java-Threads ausgeführt werden.
  - Die Garbage-Collection wird durch den Metronom-Garbage-Collector gesteuert. Der Garbage-Collector wird jedoch mit einer höheren Priorität ausgeführt als die höchste Priorität eines Echtzeitthreads und unterbricht die Ausführung Ihres Programms.
- Wählen Sie bei Antworten, die kürzer als eine Millisekunde sind, No-Heap-Echtzeitthreads aus.

  Die Priorität von No-Heap-Echtzeitthreads kann höher als die Priorität der Garbage-Collection festgelegt werden und wird daher vom Metronom nicht wesentlich geändert. Lediglich der Metronomalarmthread wird mit der höchsten Priorität ausgeführt, wodurch die CPU nur geringfügig belastet wird.

3. Ermitteln Sie, ob Ihre Anwendung Handler für asynchrone Ereignisse benötigt. Diese Anforderung hängt von der Struktur Ihres Programms ab.
   - Wählen Sie bei einer Antwortzeit, die kürzer als 10 ms ist, Echtzeitthreads aus.
   - Wählen Sie bei einer Antwortzeit, die kürzer als eine Millisekunde ist, No-Heap-Echtzeitthreads aus.
4. Ermitteln Sie Threadprioritäten. Im Allgemeinen gilt Folgendes: Je kürzer der Zeitraum ist, desto höher ist die Priorität.
5. Legen Sie Speichermerkmale fest.
   - Wenn eine Task eine variable oder hohe Zuordnungsrate aufweist, die die GC möglicherweise überlastet, erwägen Sie eine Ratenbegrenzung (über MemoryParameters) oder die Zuordnung in einem Speicherbereich für Objekte mit beschränkter Lebensdauer.
   - Wenn eine Task während einer Berechnung eine große Menge von temporären Daten generiert, erwägen Sie die Verwendung eines Speicherbereichs für Objekte mit beschränkter Lebensdauer.
   - Wenn eine Task während des Starts einige Daten generiert, die während der Lebensdauer der JVM benötigt werden, erwägen Sie die Verwendung von Speicher für Objekte mit unbeschränkter Lebensdauer. Vermeiden Sie die Verwendung von Speicher für Objekte mit unbeschränkter Lebensdauer in Fällen, wo Objekte während der Lebensdauer der JVM weiterhin erstellt werden.
   - Wenn Tasks kommunizieren müssen, vor allem, wenn eine Task unter einem No-Heap-Echtzeitthread ausgeführt wird, erwägen Sie die Verwendung eines Speicherbereichs für Objekte mit beschränkter Lebensdauer für die Kommunikation.
   - Wenn eine Task unter einem No-Heap-Echtzeitthread ausgeführt wird, erwägen Sie die Erstellung eines Speicherbereichs für Objekte mit beschränkter

Lebensdauer wie z. B. LTMemory zur Aufnahme des No-Heap-Threads, der Laufzeitparameter und möglicherweise der Warteschlangen ohne Wartezeit, über die mit der Task kommuniziert wird. Das LTMemory-Objekt muss im Bereich für Objekte mit unbeschränkter Lebensdauer oder in einem anderen Bereich erstellt werden, um Fehler zu vermeiden, wenn der No-Heap-Thread versucht, darauf zu verweisen.

6. Modifizieren Sie die Laufzeitoptionen, um die Leistung Ihrer Anwendung zu steigern, wenn Sie die Struktur und den Inhalt Ihrer Anwendung festgelegt haben. Dieser Vorgang wird in den nächsten Schritten erläutert:
   a. Legen Sie während des ursprünglichen Tests Ihrer Anwendung über die Optionen **-Xmx**, **-Xgc:immortalMemorySize=Größe** und **-Xgc:scopedMemoryMaximumSize=Größe** eine großzügige Speicherkapazität für den Heapspeicher, Speicher für Objekte mit beschränkter Lebensdauer und Speicher für Objekte mit unbeschränkter Lebensdauer fest.

      Anmerkung: Bei der Metronom-GC müssen die anfänglichen und maximalen Größen des Heapspeichers identisch sein, weil die Metronom-GC die Größe des Heapspeichers nicht erhöht. Die Vergrößerung des Heapspeichers ist eine nicht deterministische Operation.
   b. Ermitteln Sie mit der Option **-verbose:gc** die verwendete Speicherkapazität.
   c. Modifizieren Sie die Option **-Xgc:targetUtilization**, damit für die Garbage-Collection genügend Zeit verfügbar ist. Der Standardwert beträgt 70 % und dieser Prozentsatz ist gewöhnlich für die meisten Anwendungen adäquat. Stellen Sie sicher, dass die Garbage-Collection-Rate etwas höher ist als die Zuordnungsrate.
   d. Legen Sie mit der Option **-Xmx** eine realistische Größe für den Heapspeicher fest.

Java-Anwendungen modifizieren

Soll Code geschrieben werden, der die JavaEchtzeitfeatures nutzt, ersetzen Sie bei Threads `java.lang.Thread` durch `javax.realtime.RealtimeThread`.

Vorbereitende Schritte

Das folgende Beispiel basiert auf der Klasse JavaRadar.java, die sich in der Datei `demo/realtime/sample_application.zip` befindet.

Informationen zu diesem Vorgang

Das Programmiermodell für Echtzeitthreads ähnelt dem Programmiermodell für Standard-Java-Anwendungen. Diese eher rudimentäre Art, Ihrem Programm Echtzeitthreads hinzuzufügen, nutzt die Features von WebSphere Real Time for RT Linux jedoch nicht optimal. Hierzu müssen Sie die Threads modifizieren, sodass ihnen eine Priorität zugeordnet wird, und auch entscheiden, welche Hauptspeicherbereiche sie verwenden.

Ihre Anwendung profitiert nur geringfügig von der Änderung der Klassen Ihrer Threads, weil die Standardpriorität von Echtzeitthreads größer ist als die Priorität von Standard-Java-Threads.

Sie ändern JavaRadar in RealtimeThread, indem Sie die Klasse, die erweitert wird, von `Thread` in `RealtimeThread` ändern.

Ersetzung von java.lang.Thread durch javax.realtime.RealtimeThread

Die Klasse JavaRadar in der Musteranwendung erweitert java.lang.Thread. Beispiel:

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

Soll aus diesem Java-Thread ein Echtzeitthread werden, definieren Sie diese Klassendefinition wie folgt erneut:

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

Echtzeitthreads schreiben

Bislang haben Sie nur eine Anwendung modifiziert. Nun ist es an der Zeit, eigenen Code zu schreiben. Sie können Anwendungen schreiben, die Echtzeitthreads verwenden, um die Echtzeitprioritätsebenen und Hauptspeicherbereiche zu nutzen.

Vorbereitende Schritte

Das folgende Beispiel basiert auf den Klassen JavaRadar.java, RTJavaRadar.java und RTJavaControlLauncher.java, die sich in der Datei demo/realtime/sample_application.zip befinden.

Das folgende Muster zeigt Ihnen, wie Sie den Speicher für Objekte mit unbeschränkter Lebensdauer mit demselben Muster verwenden können, das in „Java-Anwendungen modifizieren" auf Seite 78 beschrieben ist.

Informationen zu diesem Vorgang

Das Programmiermodell für Echtzeitthreads ähnelt dem Programmiermodell für Standard-Java-Anwendungen.

Die Vorteile der Verwendung von Echtzeitthreads sind:

- Vollständige Unterstützung für Threadprioritäten auf Betriebssystemebene in Echtzeitthreads
- Verwendung der Speicherbereiche für Objekte mit beschränkter und unbeschränkter Lebensdauer
  - Über den Speicher für Objekte mit beschränkter Lebensdauer können Sie die Freigabe von Speicher explizit steuern, ohne die Garbage-Collection zu beeinträchtigen.
  - Mit No-Heap-Echtzeitthreads können Sie den Speicher für Objekte mit unbeschränkter Lebensdauer verwenden, um Garbage-Collection-Pausen zu vermeiden.
  - Für jene Echtzeitthreads, die auf Objekte im Heapspeicher verweisen, und für jene Echtzeitthreads, die im Heapspeicher gespeichert sind, wird eine Garbage-Collection durchgeführt.
  - No-Heap-Echtzeitthreads können nicht auf Objekte im Heapspeicher verweisen und sind daher nicht von der Garbage-Collection betroffen.

In Tabelle 11 auf Seite 80 sind die Prioritäten auf der Basis zugeordnet, dass SimulationThread die höchste Priorität hat, weil dieser Thread externe Ereignisse darstellt und durch keine Aktivität im Programm zurückgestellt werden darf. RadarThread muss schnell auf die Pingsignale vom Controller antworten. Je schneller die Antwort ist, desto genauer ist die Messung der Höhe des Landefahrzeugs. ListenThread muss auch schnell auf Befehle vom Controller antworten, RadarThread hat aber Vorrang.

Diese drei Threads befinden sich im Speicher für Objekte mit beschränkter Lebensdauer, weil die Simulation als Server ausgeführt wird. Nachdem der Server eine Simulation ausgeführt hat, kann er den Speicherbereich für Objekte mit beschränkter Lebensdauer verlassen und dann erneut in ihn eintreten, um auf eine weitere Ausführung der Simulation zu warten. Der Server verwendet den Speicher für Objekte mit beschränkter Lebensdauer, damit er sich selbst zurücksetzen kann.

RTJavaRadarthread hat die höchste Priorität der Controller-Threads, weil für ihn das Timing wichtig ist. Dies liegt daran, dass er anhand der angegebenen Zeit die Höhe ableitet. Er befindet sich im Speicher für Objekte mit unbeschränkter Lebensdauer, weil er als NHRT-Thread ausgeführt wird, der Controller nur einmal ausgeführt wird und der Speicher freigegeben wird, wenn die JVM beendet wird.

Für RTJavaControlThread und RTJavaEventThread sind die Zeitvorgaben nicht so kritisch. Daher ist die Verwendung des Heapspeichers annehmbar.

RTLoadThread führt keine nützliche Funktion für das Mondlandefahrzeug aus. RTLoadThread veranschaulicht jedoch, dass wesentliche Hauptspeicherzuordnung und -freigabe mit einer niedrigeren Priorität ausgeführt werden kann als für andere Threads, ohne sich auf die Leistung der Threads mit höherer Priorität auszuwirken.

Tabelle 11. Beziehung von Threads zu Hauptspeicherbereichen in der Musteranwendung

Speicher	Thread	Priorität
Speicher für Objekte mit beschränkter Lebensdauer	demo.sim.SimulationThread	38
	demo.sim.RadarThread	37
	demo.sim.SimulationThread.ListenThread	36
Speicher für Objekte mit unbeschränkter Lebensdauer	demo.controller.RTJavaRadarThread	15
Heapspeicher	demo.controller.RTJavaControlThread	14
	demo.controller.RTJavaEventThread	13
Speicher für Objekte mit beschränkter Lebensdauer und Heapspeicher	demo.controller.RTLoadThread	12

Beispiele

Der folgende Code aus demo.sim.SimulationThread zeigt, wo die Priorität 38 festgelegt wurde. **1** Diese Codezeile ruft die maximale Priorität ab, die in der JVM verfügbar ist.

```
super(null, area);

// Set priority separately, as we are using "this".
// Note that PriorityScheduler.MAX_PRIORITY has been deprecated.
this.setSchedulingParameters(new PriorityParameters(PriorityScheduler
        .getMaxPriority(this)));    1
```

Der folgende Code aus demo.sim.SimLauncher zeigt, wo der Speicher für Objekte mit beschränkter Lebensdauer definiert wurde. **2** zeigt die Zuordnung von LTMemory an, ein Speicherbereich für Objekte mit beschränkter Lebensdauer, der Hauptspeicher in linearer Zeit reserviert.

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

/*
 * The LTMemory object has to be created in a memory area that the
 * NHRTs can access.
 */
ImmortalMemory.instance().enter(new Runnable() {
    public void run() {
        myMemRef.ref = new LTMemory(10000000); 2
    }
});

final MemoryArea simMemArea = myMemRef.ref;
```

Das ScopedMemoryArea-Objekt, auf das `simMemArea` verweist, wird im Speicher für Objekte mit unbeschränkter Lebensdauer zugeordnet, weil der NHRT-Thread in der Lage sein muss, auf das Objekt zu verweisen, das ScopedMemoryArea darstellt. Durch die Zuordnung im Heapspeicher löst der NHRT-Thread-Konstruktor IllegalArgumentException aus, weil sich sein Hauptspeicherbereichsargument im Heapspeicher befand.

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

Der folgende Code aus demo.controller.RTJavaControlLauncher zeigt, wo der Speicher für Objekte mit unbeschränkter Lebensdauer definiert ist und von RTJavaRadar verwendet wird. Da RTJavaRadar während der gesamten Lebensdauer der Controller-JVM einmal ausgeführt wird, reserviert diese Klasse nur beim Start Hauptspeicher. Sie kann problemlos im Speicher für Objekte mit unbeschränkter Lebensdauer ausgeführt werden. Der Anwendungsentwurf profitiert hiervon, weil der Controller auf die RTJavaRadar-Methoden zugreifen kann, ohne zuerst in den Speicherbereich für Objekte mit beschränkter Lebensdauer einzutreten. Der Eintritt in den Speicherbereich für Objekte mit beschränkter Lebensdauer ist schwierig, weil der Controller für die Ausführung in gewöhnlichem Java sowie in Echtzeit-Java geschrieben wurde.

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

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

// Create RTJavaRadar in Immortal, it is an NHRT.
// If it was in scoped, it's interaction with the other threads would
// be more complex.
ImmortalMemory.instance().enter(new Runnable() {
    public void run() {
        // Realtime version of Radar.
        radarRef.ref = new RTJavaRadar(radarPort, ImmortalMemory
                .instance());
    }
});

RTJavaRadar radarJava = radarRef.ref;
```

Handler für asynchrone Ereignisse schreiben

Handler für asynchrone Ereignisse reagieren auf Zeitgeberereignisse oder auf Ereignisse, die außerhalb eines Threads auftreten, z. B. Eingabe von einer Anwen-

dungsschnittstelle. In Echtzeitsystemen müssen diese Ereignisse vor dem Ablauf der Termine antworten, die Sie für Ihre Anwendung festlegen.

Vorbereitende Schritte

Das folgende Beispiel basiert auf den Klassen RTJavaEventThread.java und RTJavaControlLauncher.java, die sich in der Datei `demo/realtime/sample_application.zip` befinden.

Informationen zu diesem Vorgang

In der Musteranwendung wartet der Ereignisthread auf Ereignisse von der Simulation, die einen Absturz oder eine Landung signalisieren. In der Echtzeitversion dieses Threads wird der Mechanismus AsyncEvent verwendet. Die folgenden Ereignisse werden verwendet, um die entsprechende Statusnachricht auszugeben und und den Controller zu beenden.

Für RTJavaEventThread sind zwei asynchrone Ereignisse definiert. Beide Ereignisse haben keine Parameter.

```
public class RTJavaEventThread extends RealtimeThread {

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

Die folgenden Ereignisse erstellen und registrieren zwei Handler für asynchrone Ereignisse:

```
    /**
     * Pass a runnable object that will be fired when the land event occurs.
     *
     * @param runnable code to be executed when land event is triggered.
     */
    public void addLandHandler(Runnable runnable) {
        AsyncEventHandler handler = new AsyncEventHandler(runnable);
        this.landEvent.addHandler(handler);
    }

    /**
     * Pass a runnable object that will be run when the crash event occurs.
     *
     * @param runnable code to be executed when crash event is triggered.
     */
    public void addCrashHandler(Runnable runnable) {
        AsyncEventHandler handler = new AsyncEventHandler(runnable);
        this.crashEvent.addHandler(handler);
    }
```

Wenn die Absturz- bzw. Landenachricht empfangen wird, wird der entsprechende Handler für asynchrone Ereignisse ausgelöst, wodurch ausführbare Objekte (Runnable) freigegeben werden.

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

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

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

Ergebnisse

RTJavaControlLauncher.java enthält Aufrufe der Methoden addLandHandler und addCrashHandler. Durch die übergebenen ausführbaren Objekte (Runnable) wird auf der Konsole eine Nachricht ausgegeben und der Steuerthread wird gestoppt, wenn die zugeordneten Handler für asynchrone Ereignisse ausgelöst werden. Informationen zum Zeitpunkt der Auslösung finden Sie in RTJavaEventThread.java.

```
// AEH runnable for land handler.
javaEventThread.addLandHandler(new Runnable() {
    public void run() {
        System.out.println("LAND!");
    }
});

// AEH runnable for crash handler.
javaEventThread.addCrashHandler(new Runnable() {
    public void run() {
        System.out.println("CRASH!");
    }
});
```

NHRT-Threads schreiben

Wollen Sie einer Java-Anwendung No-Heap-Echtzeitthreads (NHRT-Threads) hinzufügen, entwickeln bzw. modifizieren Sie Ihre eigenen Programme anhand dieses Lernprogramms.

Vorbereitende Schritte

Das folgende Beispiel basiert auf den Klassen SimulationThread.java und SimLauncher.java, die sich in der Datei `demo/realtime/sample_application.zip` befinden.

Informationen zu diesem Vorgang

Die Klasse demo.sim.SimulationThread ist Teil der Simulation in der Demoanwendung. Sie ist eine realistische Konzeption und wird daher wahrscheinlich ohne Unterbrechung vom restlichen System ausgeführt. Der Thread wird als NoHeapRealtimeThread mit der höchsten verfügbaren Priorität erstellt, um sicherzustellen, dass der Thread durch die Garbage-Collection oder andere Threads im System nicht unterbrochen wird.

In SimulationThread ruft der folgende Konstruktor den Superkonstruktor „NoHeapRealtimeThread(SchedulingParameters scheduling, MemoryArea area)" auf, bevor er SchedulingParameters und ReleaseParameters separat festlegt:

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

        super(null, area);

        // Set priority separately, as we are using "this".
        // Note that PriorityScheduler.MAX_PRIORITY has been deprecated.
        this.setSchedulingParameters(new PriorityParameters(PriorityScheduler
                .getMaxPriority(this)));

        ReleaseParameters releaseParms = new PeriodicParameters(null,
                new RelativeTime(period, 0)); // 20ms cycle (50Hz)
```

```
        this.setReleaseParameters(releaseParms);

        // It is good practice to identify each of the threads.
        this.setName("SimulationThread");

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

Die anderen aktiven Threads in der Simulation werden auch als No-Heap-Echtzeitthreads (NHRT-Threads) erstellt, allerdings mit einer etwas niedrigeren Priorität. Informationen zur Zuordnung von Prioritäten finden Sie in „Echtzeitthreads schreiben" auf Seite 79.

Die Simulation kann unendlich ausgeführt werden, d. h., sie wird nach ihrer Beendigung erneut gestartet. Da die Simulation aus NHRT-Threads besteht, können Sie den Speicher für Objekte mit beschränkter Lebensdauer (ScopedMemory) oder den Speicher für Objekte mit unbeschränkter Lebensdauer (ImmortalMemory) auswählen. Die Musteranwendung verwendet den Speicher für Objekte mit beschränkter Lebensdauer für die Simulation, weil es am besten ist, den zugeordneten Speicher für Objekte mit beschränkter Lebensdauer zu verlassen, wenn die Simulation abgeschlossen ist, und wieder in ihn einzutreten, um auf die nächste Ausführung zu warten. In diesem Fall wird von der nächsten Ausführung kein Status der vorherigen Ausführung übernommen.

Die meisten Klassen sind NHRT-sicher, sie können jedoch auf eine Art ausgeführt werden, die nicht NHRT-sicher ist. Wenn z. B. DatagramSockets im Speicher für Objekte mit unbeschränkter Lebensdauer oder in einem übergeordneten Speicherbereich für Objekte mit beschränkter Lebensdauer verbleiben, treten möglicherweise Probleme auf, weil sie nicht auf mehrere Hauptspeicherbereiche verteilt werden können. Die Musteranwendung verwendet nur den Speicherbereich für Objekte mit beschränkter Lebensdauer, um derartige Probleme zu verhindern.

Hauptspeicherzuordnung in RTSJ

In RTSJ können Sie ein Objekt einem bestimmten Hauptspeicherbereich auf verschiedene Arten zuordnen. Es ist nicht immer offensichtlich, welche Art zu einem gegebenen Zeitpunkt gewählt werden soll.

Jeder Ansatz hat gewisse Merkmale, die zwischen RTSJ-Implementierungen variieren und sich auf die Leistung oder den endgültigen Speicherbedarf auswirken. In diesem Abschnitt werden die verfügbaren Optionen und Umstände vorgestellt, unter denen sie möglicherweise die beste Wahl für das Zuordnen eines Objekts sind.

Statischer Initialisierungsoperator

Die einfachste Art, ein Objekt im Speicher für Objekte mit unbeschränkter Lebensdauer zuzuordnen, ist die Zuordnung in einem statischen Initialisierungsoperator. Der Vorteil ist, dass Sie den Speicherkontext nicht zu ändern brauchen, allerdings sind die Umstände, unter denen dieses Muster geeignet ist, recht begrenzt. Dieser Ansatz ist insofern effizient, als dass die belegte Speicherkapazität für Objekte mit unbeschränkter Lebensdauer auf die Speicherkapazität begeschränkt ist, die für das Objekt benötigt wird.

MemoryArea.newInstance(Class c)

Dieser Ansatz ist unkompliziert, wenn sich ein Thread in einem Speicherkontext befindet und ein Objekt in einem anderen Bereich zuordnen will, der sich bereits im Bereichsstack des Threads befinden muss. Der Vorteil ist, dass Sie nur auf die zu instanziierende Klasse Zugriff benötigen. Die Methode newInstance muss jedoch einen entsprechenden Konstruktor erstellen. Dieses Muster ist am geeignetsten, wenn Objekte einer bestimmten Klasse selten zugeordnet werden müssen. Andernfalls führt dieser Ansatz zu hoher Speicherbelegung.

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

Auch dieser Ansatz ist einfach, wenn sich ein Thread in einem Speicherkontext befindet und ein Objekt in einem anderen Kontext zuordnen will, der sich bereits im Bereichsstack des Threads befinden muss. In diesem Fall müssen Sie einen Konstruktor sowie einige Argumente übergeben und sicherstellen, dass der Konstruktor im aktuellen Speicherkontext gültig ist. Da die Methode newInstance keinen Konstruktor zu erstellen braucht, ist die Speicherbelegung niedriger als bei newInstance(Class c). Daher ist dieses Muster geeigneter, wenn Objekte häufiger zugeordnet werden und Sie bereit sind, den Konstruktor im Voraus zuzuordnen und beispielsweise im Speicher für Objekte mit unbeschränkter Lebensdauer (ImmortalMemory) zu speichern.

MemoryArea.enter(Runnable r) gefolgt von neuer <Klasse>()

Dieser Ansatz macht den angegebenen Hauptspeicherbereich zum neuen Standard für Zuordnungen und macht die Reflexion und die zugehörigen Konstruktorobjekte unnötig. Daher ist dieser Ansatz am geeignetsten, wenn viele Objekte erstellt werden sollen, weil keine zusätzliche Speicherbelegung oberhalb des Objekts auftritt. Dieser Ansatz funktioniert nur, wenn der gewünschte Bereich im Bereichsstack eines Threads nicht bereits aktiv ist. Dieser Ansatz ist aufgrund der erforderlichen Erstellung eines ausführbaren Hauptspeicherbereichs (Runnable) komplexer als die Verwendung von newInstance, weil Sie im Allgemeinen Parameter im ausführbaren Hauptspeicherbereich oder über statische Felder bzw. Instanzfelder übergeben müssen.

MemoryArea.executeInArea(Runnable r) gefolgt von neuer <Klasse>()

Auch dieser Ansatz macht den angegebenen Hauptspeicherbereich zum neuen Standard für Zuordnungen und macht die Reflexion und die zugehörigen Konstruktorobjekte unnötig. Daher ist dieser Ansatz am geeignetsten, wenn viele Objekte erstellt werden sollen, weil keine zusätzliche Speicherbelegung oberhalb des Objekts auftritt. Sie können diesen Ansatz verwenden, wenn sich der gewünschte Bereich bereits im Bereichsstack des aktuellen Threads befindet und daher flexibler ist als MemoryArea.enter. Dieser Ansatz ist aufgrund der erforderlichen Erstellung eines ausführbaren Hauptspeicherbereichs (Runnable) komplexer als die Verwendung von newInstance, weil Sie im Allgemeinen Parameter im ausführbaren Hauptspeicherbereich oder über statische Felder bzw. Instanzfelder übergeben müssen.

Class.newInstance()

Dieser Ansatz erstellt die neue Instanz im aktuellen Hauptspeicherbereich und muss daher mit MemoryArea.enter oder MemoryArea.executeInArea verwendet werden. Es tritt keine zusätzliche Speicherbelegung oberhalb des Objekts auf.

Zeitgeber mit hoher Auflösung verwenden

Der Echtzeittaktgeber bietet höhere Genauigkeit als die Taktgeber der Standard-JVM.

Vorbereitende Schritte

Das folgende Beispiel basiert auf der Klasse RTJavaRadar.java, die sich in der Datei `demo/realtime/sample_application.zip` befindet.

Informationen zu diesem Vorgang

Gewöhnliches Java hat eine begrenzte Fähigkeit, Takt- und Zeitgeber zu verarbeiten. Mithilfe von Real-Time Specification for Java können absolute Zeiten bis auf die Nanosekunde genau angegeben werden und die reale Prozesslaufzeit widerspiegeln. Mit javax.realtime.HighResolutionTime und den entsprechenden Unterklassen wird die Zeit mit zwei Komponenten dargestellt: Millisekunden und Nanosekunden.

WebSphere Real Time for RT Linux verwendet die Unterstützung des zugrunde liegenden Betriebssystems, um die Zeit mit hoher Auflösung zu liefern. Aktuelle Linux-Kernel liefern einen Taktgeber mit einer garantierten maximalen Genauigkeit von 4 Millisekunden. Die mit WebSphere Real Time for RT Linux gelieferten Linux-Patches bieten einen Taktgeber mit einer Genauigkeit von ungefähr 1 Mikrosekunde.

Die Klasse RTJavaRadar veranschaulicht die Verwendung des Zeitgebers mit hoher Auflösung:

- **1** ruft den Echtzeittaktgeber ab.
- **2** ruft die aktuelle absolute Zeit ab.
- **3** ruft die Nanosekunden der Zeit ab. Die Genauigkeit des Echtzeittaktgebers bedeutet, dass die Verwendung von Nanosekunden angemessen ist.
- **4** ruft die Zeit vor und nach dem Pingsignal ab.
- **5** gibt die Sinkgeschwindigkeit des Landefahrzeugs zurück.
- **6** lässt den Thread 5 Millisekunden lang warten, bevor eine weitere Iteration ausgeführt wird.

```
public void run() {
       // The following objects are created in advance and reused each
       // iteration.
       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;

               // Rather than use the time = rtClock.getTime() form, this
               // method
               // replaces the values in a preexisting AbsoluteTime object.
               rtClock.getTime(time);                  4
               millis = time.getMilliseconds();
```

```
            nanos = time.getNanoseconds();

            // We time the time it takes to send the ping and receive the
            // pong.
            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) {
                // This is not important.
            }
        }
```

Der vorstehende Code kann mit dem folgenden Standard-JVM-Code in der Klasse JavaRadar verglichen werden:

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

            long nanos = System.nanoTime(), lastnanos;
            while (this.running) {
                /* Set the height every x milliseconds */
                Thread.sleep(5);
                lastnanos = nanos;
                lastheight = height;

                nanos = System.nanoTime();

                this.radarPort.ping();

                // Time scale is height units per millisecond
                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;
            }
```

Musteranwendung

Die Musteranwendung verwendet eine Reihe von Beispielen zur Veranschaulichung der Features von WebSphere Real Time for RT Linux, die zur Verbesserung der Echtzeitmerkmale von Java-Programmen verwendet werden können.

Die Quellendateien für die Musteranwendung sind in der Dateidemo/realtime/sample_application.zip enthalten.

Die Musteranwendung besteht aus zwei Hauptkomponenten:

- **Eine Simulation**, ein einfaches Beispiel eines Mondlandefahrzeugs. Seine Position wird durch seine Höhe über dem Boden und die Entfernung vom Landebereich definiert. Weitere Informationen hierzu finden Sie in Abb. 6 auf Seite 89.

  Die Simulation wurde mit No-Heap-Echtzeitthreads (NHRT-Threads) geschrieben und wird in dieser Dokumentation nicht modifiziert.
- **Ein Controller**, der Befehle an die Simulation sendet. Er sendet Radarpingsignale, um die Höhe des Landefahrzeugs zu ermitteln und die Sinkgeschwindigkeit des Landefahrzeugs auf der Basis dieser Informationen zu steuern. Der Controller empfängt einen Datenstrom vom Landefahrzeug, z. B. die Entfernung des Landefahrzeugs zum Landebereich.

  Der Controller wurde anfänglich in Standard-Java geschrieben. In „Java-Anwendungen modifizieren" auf Seite 78 wird er zu einem Java-Echtzeitprogramm entwickelt.

Je nach dem Ergebnis der Landung wird dem Controller eine von zwei Nachrichten gesendet: Absturz oder Landung.

Mit der Musteranwendung können Sie die folgenden Operationen ausführen:

- Die Simulation und den Controller zusammen ausführen, um die gemeinsame Ausführung von Echtzeit- und Standard-Java-Klassen zu veranschaulichen. Informationen hierzu finden Sie in „Musteranwendung erstellen" auf Seite 90 und „Musteranwendung ausführen" auf Seite 90, wo die Ausgabe der Musteranwendung angezeigt wird.

  Anmerkung: Sie können die Simulation und den Controller über die Klasse LaunchBoth gleichzeitig starten.
- Den Unterschied zwischen dem Metronom-Garbage-Collector und dem Standard-Garbage-Collector vergleichen. Informationen hierzu finden Sie in „Musteranwendung ohne Real Time ausführen" auf Seite 90 und „Musteranwendung mit dem Metronom-Garbage-Collector ausführen" auf Seite 91.
- Die Anwendung mit dem AOT-Compiler ausführen. Informationen hierzu finden Sie in „Musteranwendung über AOT ausführen" auf Seite 92.

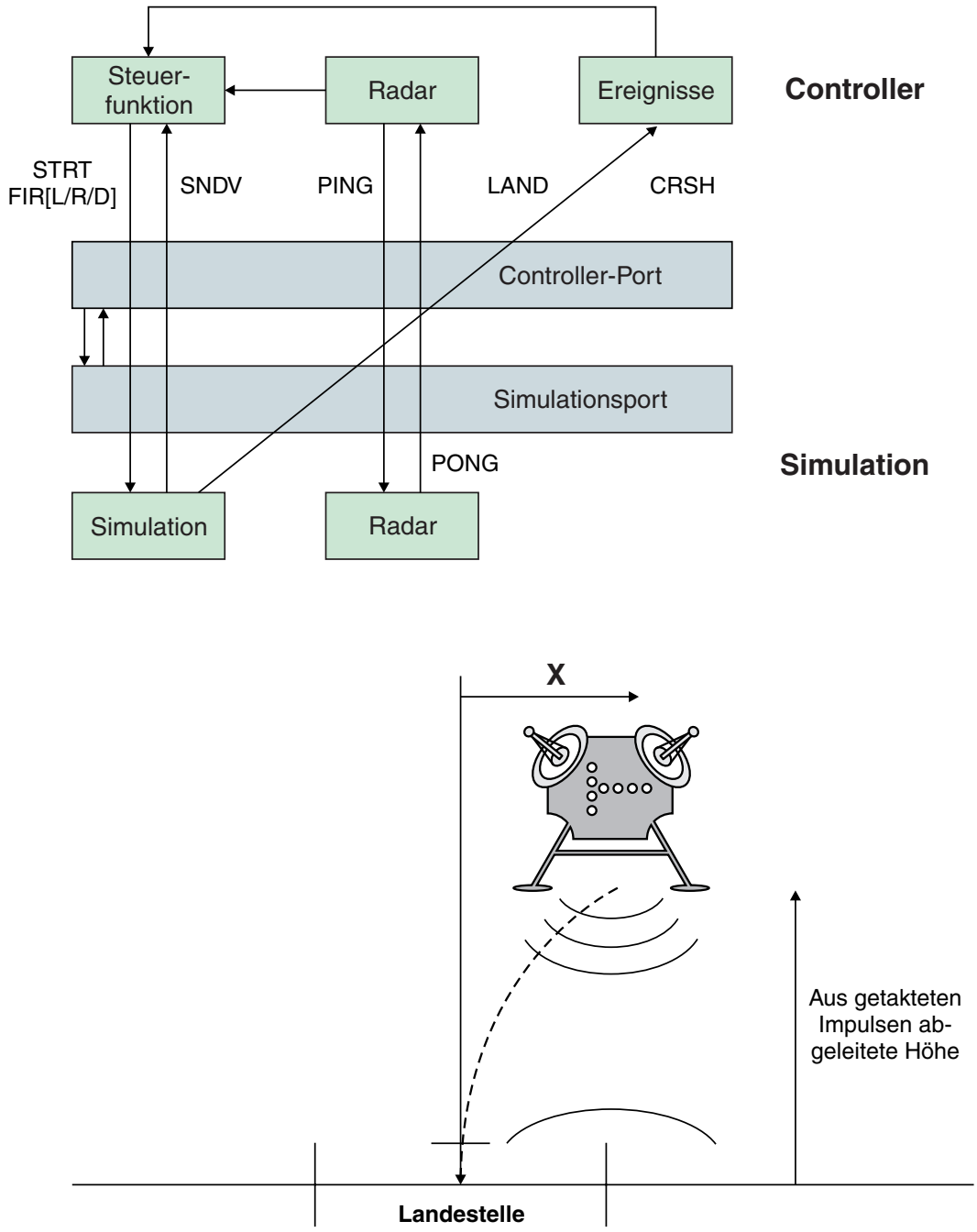

Abbildung 6. Diagramm des Landefahrzeugs

Dieses Diagramm zeigt die Beziehung der in diesem Muster enthaltenen Module. Der obere Teil des Diagramms zeigt den Controller und Simulator. Der Controller verfügt über drei Threads: Steuer-, Radar- und Ereignisthreads. Der Simulator verfügt über zwei Threads: den Simulator- und den Radarthread. Der untere Teil des Diagramms zeigt das Mondlandefahrzeug an und gibt die beiden Steuerfunktionen 'Links' und 'Rechts' zusammen mit den Impulsen an, die die Höhe des Landefahrzeugs ermitteln.

Musteranwendung erstellen

Der Quellcode der Musteranwendung dient als Leitfaden. Der Java-Quellcode muss entpackt und kompiliert werden, bevor er ausgeführt werden kann.

Vorgehensweise

1. Erstellen Sie ein Arbeitsverzeichnis.
2. Extrahieren Sie die Musteranwendung in Ihr Arbeitsverzeichnis:
   ```
   unzip sample_application.zip
   ```
3. Erstellen Sie ein neues Verzeichnis für Ihre Ausgabe:
   ```
   mkdir classes
   ```
4. Kompilieren Sie die Quelle.
   a. Generieren Sie eine Liste der Dateien:
      ```
      find -name "*.java" > source
      ```
   b. Kompilieren Sie die Quelle:
      ```
      javac -Xrealtime -Xlint:deprecated -g -d classes @source
      ```
   c. Erstellen Sie eine JAR-Datei der Klassendateien:
      ```
      jar cf demo.jar -C classes/ .
      ```

Nächste Schritte

Sie können die Musteranwendung nun ausführen.

Musteranwendung ausführen

WebSphere Real Time bietet eine Standard-JVM und eine Echtzeit-JVM, die mit dem Befehlszeilenargument **-Xrealtime** gestartet wird.

Die Musteranwendung enthält zwei Komponenten, die für die Ausführung in separaten JVMs konzipiert wurden:

- Die Simulation, die nur in Echtzeit-Java ausgeführt wird.
- Der Controller, der in Nicht-Echtzeit- oder Echtzeit-Java ausgeführt werden kann.

Die Ausführung des Controller-Codes in einer Vielzahl von Modi veranschaulicht die Vorteile derIBM Real Time Java-Technologie.

Musteranwendung ohne Real Time ausführen

In dieser Prozedur führen Sie die Musteranwendung aus, ohne IBM WebSphere Real Time zu nutzen.

Vorbereitende Schritte

Zum Ausführen der Musteranwendung müssen Sie zuerst den Musterquellcode erstellen. Weitere Informationen hierzu finden Sie in „Musteranwendung erstellen".

Vorgehensweise

1. Starten Sie die Simulation:
   ```
   java -Xrealtime -classpath ./demo.jar -Xgc:scopedMemoryMaximumSize=11m
   demo.sim.SimLauncher <Port>
   ```

   In diesem Befehl ist <*Port*> ein für die Workstation freigegebener Port.

2. Starten Sie den Controller:

   ```
   java -classpath ./demo.jar -mx300m demo.controller.JavaControlLauncher <Host> <Port>
   ```

   In diesem Befehl ist *<Host>* der Hostname der Workstation, auf der die Simulation ausgeführt wird, und *<Port>* ist der im vorherigen Schritt angegebene Port.

Ergebnisse

Die Anwendung erzeugt eine Nachricht, die angibt, dass die Simulation und der Controller gestartet wurden:

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

Einige Positionsmuster der Werte im Controller werden in der Konsole ausgegeben:

```
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
```

Vor dem Stopp der Simulation wird eine Ereignisübersichtsnachricht ausgegeben:

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

Zusätzlich zu den Positionsmustern und zur Ereignisübersichtsnachricht erzeugt der Controller das Diagramm `graph.svg` im selben Verzeichnis. Das Diagramm enthält eine Darstellung der Positionsmuster. Das Diagramm zeigt die Auswirkung der Garbage-Collection-Pausen auf den Thread JavaRadar bei Ausführung der Anwendung mit einer Standard-Nicht-Echtzeit-JVM. Die Daten, die die Radarhöhe darstellen, weisen Spitzen auf. Die Spitzen werden durch Standard-Garbage-Collection-Pausen verursacht, die sich auf die Controller-Anwendung auswirken. Bei einigen Ausführungen sind die Garbage-Collection-Pausen so lang, dass sie Fehler verursachen und zur folgenden Nachricht führen:

```
CRASH!
```

Fügen Sie dem Controllerstartbefehl die Option **-verbose:gc** hinzu, um die durch die Garbage-Collection verursachten Pausezeiten anzuzeigen:

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

Musteranwendung mit dem Metronom-Garbage-Collector ausführen

Sie können eine Standard-Java-Anwendung in einer Echtzeitumgebung ausführen, ohne den Code neu schreiben zu müssen, indem Sie die Option -Xrealtime hinzufügen. Die Option aktiviert die Java-Echtzeitsprachenfunktionen und den Metronom-Garbage-Collector.

Vorbereitende Schritte

Zum Ausführen der Musteranwendung müssen Sie zuerst den Musterquellcode erstellen. Weitere Informationen hierzu finden Sie in „Musteranwendung erstellen" auf Seite 90.

Vorgehensweise

1. Starten Sie die Simulation:

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

   In diesem Befehl ist *<Port>* ein auf der Workstation freigegebener Port.
2. Starten Sie den Controller:

   ```
   java -Xrealtime -classpath ./demo.jar -mx300m
   demo.controller.JavaControlLauncher <Host> <Port>
   ```

   In diesem Befehl ist *<Host>* der Hostname der Workstation, auf der die Simulation ausgeführt wird, und *<Port>* ist der im vorherigen Schritt angegebene Port. Die Ausführung von beiden JVMs auf derselben Workstation kann zu weniger deterministischem Verhalten führen. Weitere Informationen finden Sie in „Wichtige Faktoren" auf Seite 26.

Ergebnisse

Die Anwendung wird ausgeführt und generiert mehrere Ausgaben, einschließlich der folgenden:

1. Nachrichten, die zeigen, dass die Simulation und der Controller gestartet wurden.
2. Positionsmuster der Werte im Controller.
3. Ein Diagram `graph.svg` im selben Verzeichnis mit einer Darstellung der Positionsmuster der Werte im Controller.
4. Eine Nachricht mit einer Ereigniszusammenfassung.

Wird die Anwendung mit der Metronom-Garbage-Collection ausgeführt, wird in den Positionsmustern und entsprechenden Diagrammen Folgendes angezeigt:

- Keine Spitzen in den Daten für die Radarhöhe
- Präzise Verfolgung der Daten für die Radarhöhe

Der Grund hierfür ist, dass der Controller-Code nun mit kürzeren Garbage-Collection-Pausen ausgeführt wird.

Die Metronom-Garbage-Collection-Pausen sind häufig, jedoch gewöhnlich kürzer als 1 Millisekunde. Nicht-Echtzeit-Garbage-Collection-Pausen treten weniger häufig auf, dauern gewöhnlich jedoch Zehnte oder Hunderste von Millisekunden. Sie können den Unterschied zwischen den Pausen anzeigen, indem Sie dem Controller-Ausführungsbefehl die Option **`-verbose:gc`** hinzufügen.

Weitere Informationen zur ausführlichen Garbage-Collection-Ausgabe finden Sie in „Informationen von 'verbose:gc' verwenden" auf Seite 146.

Musteranwendung über AOT ausführen

Diese Prozedur führt eine Standard-Java-Anwendung in einer Echtzeitumgebung mit dem Ahead-of-time-Compiler (AOT-Compiler) aus, ohne dass Code neu geschrieben werden muss. Verwenden Sie das folgende Muster, um die Ausführung

der Anwendung über den AOT-Compiler mit der Ausführung derselben Anwendung über den JIT-Compiler zu vergleichen.

Weitere Details zur AOT-Kompilierung finden Sie in „Für WebSphere Real Time for RT Linux kompilierten Code verwenden" auf Seite 40.

Vorbereitende Schritte

Zum Ausführen der Musteranwendung müssen Sie zuerst den Musterquellcode erstellen. Weitere Informationen hierzu finden Sie in „Musteranwendung erstellen" auf Seite 90.

Informationen zu diesem Vorgang

Der AOT-Compiler kompiliert Ihre Java-Anwendung vor der Ausführung zu nativem Code. Sie können genauer vorhersagen, wie die Anwendung ausgeführt wird, weil die JIT-Kompilierung keine Unterbrechungen verursacht.

Vorgehensweise

1. Konvertieren Sie die Anwendungsbytecodes in nativen Code.
   a. Die Konvertierung findet statt, indem Sie zuerst die Musteranwendung mit dem normalen JIT-Compiler ausführen.

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

      In diesem Befehl ist *<Port>* ein für die Workstation freigegebener Port.
   b. Führen Sie die Anwendung in einem anderen Fenster aus.

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

      In diesem Befehl ist *<Port>* der im vorherigen Schritt angegebene Port. Die Anwendungsausgabeergebnisse ähneln den folgenden Nachrichten:

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

      und:

      ```
      Land!
      ```
   c. Kombinieren Sie die in den vorherigen Schritten erstellten AOT-Optionsdateien.

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

      An die Namen der in den vorherigen Schritten erstellten Protokolldateien sind Datums- und Prozess-ID-Informationen angehängt. Das Format für den Dateinamen wird durch die Option **vlog=** angegeben. Beispielsweise generiert **vlog=sim.aotOpts** einen Dateinamen, der **sim.aotOpts.20081014.234958.13205** ähnelt:
   d. Kompilieren Sie die Dateien in der Datei `sample.aotOpts` in `realtime.jar`, `vm.jar`, `rt.jar` und in der Anwendung `demo.jar`. Bei Verwendung von Caches für gemeinsam genutzte Klassen darf der Name des Cache 53 Zeichen nicht überschreiten.

```
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
```

Die Kompilierungsergebnisse werden zurückgemeldet:

```
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
```

Anmerkung: Die Zeile:

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

bedeutet, dass ein eventuell vorhandener Cache mit dem Namen "sample" durch diesen Befehl zerstört wird, um den angegebenen Cache zu erstellen.

e. Zeigen Sie den Inhalt des gefüllten Cache an.

```
admincache -Xrealtime -cacheName "sample" -printStats
```

2. Starten Sie die Simulation:

```
java -Xrealtime -Xnojit -Xmx300m -Xshareclasses:name="sample" \
     -classpath ./demo.jar -Xgc:scopedMemoryMaximumSize=11m \
     demo.sim.SimLauncher <Port>
```

In diesem Befehl ist <*Port*> ein für diese Workstation freigegebener Port.

3. Starten Sie den Controller:

```
java -Xrealtime -Xnojit -Xmx300m -Xshareclasses:name="sample" \
     -classpath ./demo.jar \
     demo.controller.JavaControlLauncher <Host> <Port>
```

In diesem Befehl ist <*Host*> der Hostname der Workstation, auf der die Simulation ausgeführt wird, und <*Port*> ist der im vorherigen Schritt angegebene Port. Die Ausführung von beiden JVMs auf derselben Workstation kann zu weniger deterministischem Verhalten führen. Weitere Informationen finden Sie in „Wichtige Faktoren” auf Seite 26.

Ergebnisse

Die Anwendung wird ausgeführt und generiert mehrere Ausgaben, einschließlich der folgenden:

1. Nachrichten, die zeigen, dass die Simulation und der Controller gestartet wurden.

2. Positionsmuster der Werte im Controller.
3. Ein Diagramm `graph.svg` im selben Verzeichnis mit einer Darstellung der Positionsmuster der Werte im Controller.
4. Eine Nachricht mit einer Ereigniszusammenfassung.

Wird die Anwendung mit der AOT-Kompilierung ausgeführt, wird in den Positionsmustern und entsprechenden Diagrammen Folgendes angezeigt:

- Keine Spitzen in den Daten für die Radarhöhe
- Präzise Verfolgung der Daten für die Radarhöhe

Der Grund hierfür ist, dass der Controller-Code nun mit kürzeren Garbage-Collection-Pausen und ohne JIT-Kompilierungsunterbrechungen ausgeführt wird.

Ein Vorteil bei der Ausführung dieser Anwendung über den Cache für gemeinsam genutzte Klassen besteht darin, dass die Controller- und Simulations-JVMs einige Speicherbereiche gemeinsam nutzen, die von den Klassen verwendet werden, die von beiden JVMs geladen werden.

Echtzeitorientierte Musterhashzuordnung

WebSphere Real Time for RT Linux enthält HashMap- und HashSet-Implementierungen, die eine konsistentere Leistung für die Methode put als die HashMap-Standardimplementierung in IBM SDK for Java 7 liefern.

Der von IBM gelieferte Standard java.util.HashMap eignet sich gut für Anwendungen mit hohem Durchsatz. Er eignet sich auch für Anwendungen, bei denen die maximale Größe der Hashzuordnung erhöht werden muss. Bei Anwendungen, die eine Hashzuordnung benötigen, die nutzungsabhängig verschiedentlich groß sein können, gibt es ein potenzielles Leistungsproblem mit der Standardhashzuordnung. Die Standardhashzuordnung bietet gute Antwortzeiten beim Hinzufügen von neuen Einträgen zur Hashzuordnung mit der Methode put. Wenn diese Hashzuordnung jedoch aufgefüllt ist, muss ein größerer Sicherungsspeicher zugeordnet werden. Dies bedeutet, dass die Einträge im aktuellen Sicherungsspeicher migriert werden müssen. Wenn die Hashzuordnung groß ist, kann die Ausführung einer put-Operation auch lange dauern. Beispielsweise kann die Operation mehrere Millisekunden dauern.

WebSphere Real Time for RT Linux enthält eine echtzeitorientierte Musterhashzuordnung. Sie bietet dieselbe Funktionsschnittstelle wie der Standard java.util.HashMap, ermöglicht jedoch konsistentere Leistung für die Methode put. Die Musterhashzuordnung erstellt einen zusätzlichen Sicherungsspeicher, anstatt nur einen Sicherungsspeicher zu erstellen und alle Einträge zu migrieren, wenn die Hashzuordnung aufgefüllt ist. Der neue Sicherungsspeicher ist mit den anderen Sicherungsspeichern in der Hashzuordnung verkettet. Die Verkettung verursacht anfänglich eine leichte Leistungssenkung, während der leere Sicherungsspeicher zugeordnet und mit den anderen Sicherungsspeichern verkettet wird. Nachdem die Sicherungshashzuordnung aktualisiert wurde, ist sie schneller als die Migration aller Einträge. Ein Nachteil der echtzeitorientierten Hashzuordnung ist, dass die get-, put- und remove-Operationen etwas langsamer ausgeführt werden. Die Operationen sind langsamer, weil jede Suche eine Gruppe von Sicherungshashzuordnungen anstatt nur einer Sicherungshashzuordnung durcharbeiten muss.

Fügen Sie am Anfang Ihres Bootklassenpfads die Datei `RTHashMap.jar` hinzu, um die echtzeitorientierte Hashzuordnung auszuprobieren. Wenn Sie WebSphere Real Time for RT Linux im Verzeichnis `$WRT_ROOT` installiert haben, fügen Sie die folgen-

de Option hinzu, um die echtzeitorientierte Hashzuordnung anstatt der Standardhashzuordnung mit Ihrer Anwendung zu verwenden:

```
-Xbootclasspath/p:$WRT_ROOT/demo/realtime/RTHashMap.jar
```

Die Quellen- und Klassendateien für die Implementierung der echtzeitorientierten Hashzuordnung sind in die Datei `demo/realtime/RTHashMap.jar` eingeschlossen. Außerdem werden die Echtzeitimplementierungen java.util.LinkedHashMap und java.util.HashSet bereitgestellt.

WebSphere Real Time for RT Linux-Anwendungen mit Eclipse entwickeln

Die Verwendung von Eclipse bietet Ihnen eine mit vielen Funktionen ausgestattete integrierte Entwicklungsumgebung für die Entwicklung Ihrer Echtzeitanwendungen.

Vorbereitende Schritte

Wenn Sie die Eclipse-Anwendungsentwicklungsumgebung zum ersten Mal zum Entwickeln von Echtzeitanwendungen verwenden, konfigurieren Sie Ihre Umgebung mithilfe der folgenden Prozedur.

WebSphere Real Time for RT Linux enthält den Standard-Oracle-Compiler **`javac`**. Sie können einen beliebigen Compiler verwenden, er muss jedoch gültige Klassendateien für Java 5.0 erzeugen. Die Java-Klassen javax.realtime.* müssen sich jedoch im Erstellungspfad befinden.

Informationen zu diesem Vorgang

Befolgen Sie die folgenden Anweisungen, um Ihre Anwendungen in Eclipse zu entwickeln:

Vorgehensweise

1. Laden Sie Eclipse von http://www.eclipse.org/downloads/ herunter. Ihnen wird empfohlen, Eclipse 3.1.2 zu verwenden, damit Java 5.0 Kompilierungen ordnungsgemäß durchführt.
2. Laden Sie IBM SDK and Runtime Environment for Linux Platforms herunter, das mit Java 2 Technology Edition Version 5.0 kompatibel ist.
3. Extrahieren Sie die Datei `opt/IBM/javawrt3/jre/lib/i386/realtime/jclSC160/realtime.jar` aus dem WebSphere Real Time for RT Linux-Paket.
4. Öffnen Sie Eclipse und erstellen Sie ein Projekt. Klicken Sie auf **Datei** > **Neu**. Wählen Sie **Java-Projekt** in der Anzeige **Neues Projekt** aus.
5. Klicken Sie auf **Nächste**, um die Anzeige **Neues Java-Projekt** anzuzeigen.
   a. Geben Sie einen Projektnamen wie z. B. RTSJ-Tests ein.
   b. Prüfen Sie, ob der JDK-Compiler auf 5.0 gesetzt ist.
6. Klicken Sie auf **Fertig stellen**.
7. Erstellen Sie ein Arbeitsverzeichnis und importieren Sie die Datei `opt/IBM/javawrt3/jre/lib/i386/realtime/jclSC160/realtime.jar`.
8. Klicken Sie auf **Datei** > **Neu** > **Ordner**, um die Anzeige **Neuer Ordner** zu öffnen. Geben Sie einen Ordnernamen ein, z. B. *deplib*.
9. Klicken Sie auf **Fertig stellen**.

10. Klicken Sie zum Importieren Ihrer Datei `realtime.jar` auf **Datei** > **Importieren**, wodurch die Anzeige **Importieren** geöffnet wird.
11. Klicken Sie auf **Dateisystem** und dann auf **Nächste**.
12. Öffnen Sie das Verzeichnis `opt/IBM/javawrt3/jre/lib/i386/realtime/jclSC160/` auf dem Dateisystem, auf dem die JVM entpackt wurde.
13. Wählen Sie das Kontrollkästchen neben der Datei 'realtime.jar' aus, geben Sie einen Ordner als Importziel an (z. B. `RTSJ-Tests/deplib`) und stellen Sie sicher, dass die Option **Nur ausgewählte Ordner erstellen** ausgewählt ist.
14. Klicken Sie auf **Fertig stellen**.
15. Fügen Sie die JAR-Datei dem Bibliothekspfad hinzu. Klicken Sie mit der rechten Maustaste auf Ihr Projekt und klicken Sie auf **Eigenschaften**, um die Anzeige **Eigenschaften** zu öffnen.
16. Klicken Sie auf **Java-Buildpfad** und die Registerkarte **Bibliotheken**. Klicken Sie auf **JARs hinzufügen**.
17. Klicken Sie auf **realtime.jar** unter Ihrem Projektverzeichnis. Klicken Sie auf **OK**.

Ergebnisse

Wenn diese Prozedur erfolgreich ist, wird die Datei 'realtime.jar' in der Liste der JAR-Dateien auf der Registerkarte **Bibliotheken** angezeigt.

Beispiel

Eclipse kann mit `realtime.src.jar` weitere Informationen zu den RTSJ-Klassen darstellen. Öffnen Sie hierzu das Fenster **Eigenschaften** für die importierte Datei `realtime.jar`, klicken Sie auf **Java-Quellenzuordnung** und geben Sie die Speicherposition der Datei `realtime.src.jar` in **Positionspfad:** ein.

Nächste Schritte

Wenn Sie Anwendungen über Apache Ant mit Eclipse erstellen wollen, fügen Sie die Datei `realtime.jar` dem Klassenpfad in Ihrem Ant-Erstellungsscript hinzu. Beispiel:

```
<property name="rtsj.src" location="." />
<property name="rtsj.deplib" location="deplib" />
<property name="rtsj.jar.dir" location="build/rtsj-jar.dir" />

<!-- Generate .class files for this package -->
<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>
```

Dies ist nur ein Teil eines Erstellungsscripts.

Anwendungsfehler beheben

Mit Eclipse Application Developer können Sie Anwendungsfehler lokal oder über Fernzugriff beheben.

Informationen zu diesem Vorgang

Wollen Sie Fehler in Ihrer Echtzeitanwendung über Fernzugriff beheben, benötigt die JVM, für die der Debugger ausgeführt wird, die folgende Option.
`-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=10100`

Vorgehensweise

1. Geben Sie in der Linux-Umgebung, in der Ihre Anwendung ausgeführt wird, Folgendes ein:

   ```
   java -Xrealtime -agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=10100
   ```

   Dabei gilt Folgendes:
   - `server=y` gibt an, dass die JVM Verbindungen von Debuggern akzeptiert.
   - `suspend=y` lässt die JVM auf die Herstellung einer Debuggerverbindung warten, bevor sie ausgeführt wird.
   - `address=10100` ist die Portnummer, über die der Debugger die Verbindung zur JVM herstellen soll. Diese Nummer liegt normalerweise über 1024.

   Die JVM zeigt die folgende Nachricht an:

   ```
   Empfangsbereit für Transport dt_socket bei Adresse: 10100
   ```
2. Öffnen Sie Ihre Anwendung in Eclipse und wählen Sie **Debug** aus.
3. Für die Fehlerbehebung von fernen Anwendungen muss eine neue Konfiguration erstellt werden. Sie müssen eine Konfiguration nur erstellen, wenn eine Anwendung im selben Projekt ausgeführt wird und bei jeder Ausführung denselben Port überwacht.
4. Nach der Erstellung der Konfiguration müssen Sie den Namen der Konfiguration, den Namen des Projekts, das die Anwendung enthält, für die Sie die Fehlerbehebung ausführen, den *Hostnamen* der Workstation, auf der die Anwendung ausgeführt wird, und die über die Optionen von **`-agentlib`** übergebene Portnummer angeben.
5. Klicken Sie auf **Debug**, um die Debugsitzung zu starten. Die Perspektive **Debug** muss geöffnet sein, damit Sie den Status der JVM anzeigen können, für die die Fehlerbehebung über Fernzugriff ausgeführt wird.

Eclipse mit der JVM ausführen

In diesem Abschnitt wird erläutert, wie Eclipse mit WebSphere Real Time for RT Linux JVM ausgeführt werden kann.

Für die Ausführung von Eclipse mit der JVM müssen Sie Folgendes im Befehl 'eclipse' angeben:

- Das vollständig qualifizierte Verzeichnis für die ausführbare Java-Datei von WebSphere Real Time for RT Linux JVM, die verwendet werden soll.
- Die JVM-Option **`-Xrealtime`**.
- Die Größe des Speichers für Objekte mit unbeschränkter Lebensdauer, den Eclipse verwenden soll. Sie muss mindestens 128 MB betragen.

Beispiel für die Ausführung von Eclipse mit der JVM:

```
eclipse -vm $JAVA_HOME/jre/bin/java -vmargs -Xrealtime -Xgc:immortalMemorySize=128M
```

Anmerkung: Eclipse SDK nutzt nicht die verschiedenen Echtzeitspeicheroptionen, die für WebSphere Real Time for RT Linux-Anwendungen verfügbar sind. Daher wird der Speicher für Objekte mit unbeschränkter Lebensdauer schnell belegt, vor allem, wenn Eclipse vielen Stunden oder Tage lang verwendet wird, ohne erneut gestartet zu werden. Wenn ein **OutOfMemory**-Fehler auftritt, können Sie den Wert der Option **-Xgc:immortalMemorySize** erhöhen, um die Kapazität des Speichers für Objekte mit unbeschränkter Lebensdauer zu erhöhen, den Eclipse verwenden soll.

Kapitel 7. Leistung

WebSphere Real Time for RT Linux ist für konsistent kurze GC-Pausen anstatt für die höchste Durchsatzleistung oder den kleinsten Speicherbedarf optimiert.

Sie müssen sicherstellen, dass HT auf Systemen, die es unterstützen, nicht aktiviert ist. Hierdurch werden Leistungseinbußen bei der Verwendung von WebSphere Real Time for RT Linux vermieden.

Einige IBM Java Runtime-Standardoptimierungen mussten inaktiviert werden, um die Ablaufsteuerungsvariabilität zu reduzieren und für Unterstützung für Real-Time Specification for Java (RTSJ) zu sorgen. Daher kommt es wahrscheinlich zu einer Senkung der Gesamtleistung, wenn eine Standard-Java-Anwendung mit dem Parameter **-Xrealtime** ausgeführt wird.

Leistung bei zertifizierten Hardwarekonfigurationen

Zertifiziere Systeme haben eine ausreichende Taktgebergranularität und Prozessorgeschwindigkeit, um die WebSphere Real Time for RT Linux-Leistungsziele zu unterstützen. Beispielsweise würde es für eine gut geschriebene Anwendung, die auf einem nicht überladenen System und mit einer adäquaten Größe des Heapspeichers normalerweise GC-Pausezeiten geben, die stark unter 1 Millisekunde liegen, gewöhnlich ca. 500 Mikrosekunden. Im Verlauf von GC-Zyklen wird eine Anwendung mit Standardumgebungseinstellungen nicht länger als 30 % der abgelaufenen Zeit während eines 10 Millisekunden langen gleitenden Fensters angehalten. Die in GC-Pausen im Verlauf eines 10 Millisekunden langen Zeitraums verbrachte Gesamtzeit beträgt normalerweise weniger als 3 Millisekunden.

Ablaufsteuerungsvariabilität reduzieren

Die zwei Hauptquellen von Variabilität in einer Standard-JVM werden in WebSphere Real Time for RT Linux wie folgt gehandhabt:

- Java-Codevorbereitung: Das Laden und die JIT-Kompilierung wird von der AOT-Kompilierung gehandhabt. Weitere Informationen hierzu finden Sie in „AOT-Compiler verwenden" auf Seite 42.
- Garbage-Collection-Pausen: Die möglicherweise langen Pausen im Vergleich zu den Garbage-Collector-Standardmodi werden durch die Verwendung des Metronom-Garbage-Collectors vermieden. Weitere Informationen hierzu finden Sie in „Metronom-Garbage-Collector verwenden" auf Seite 72.

Gemeinsame Nutzung von Klassendaten auf verschiedenen JVMs im Nicht-Echtzeitmodus

Die gemeinsame Nutzung von Klassen wird im Nicht-Echtzeitmodus unterstützt, läuft jedoch anders ab als im Echtzeitmodus.

Sie können Klassendaten zwischen Java Virtual Machines (JVMs) gemeinsam nutzen, indem Sie sie in einer im Speicher abgelegten Cachedatei speichern. Durch die gemeinsame Nutzung verringert sich die gesamte virtuelle Speicherbelegung, wenn mehrere JVMs einen Cache gemeinsam nutzen.

Außerdem verkürzt sich durch die gemeinsame Nutzung der Systemstart von JVM, nachdem der Cache erstellt wurde. Der Cache für gemeinsam genutzte Klassen ist unabhängig von aktiven JVMs und bleibt persistent, bis er gelöscht wird.

Ein gemeinsam genutzter Cache kann Folgendes enthalten:

- Bootprogrammklassen
- Anwendungsklassen
- Metadaten, die die Klassen beschreiben
- Kompilierter AOT-Code (Ahead-of-time)

Kapitel 8. Sicherheit

Dieser Abschnitt enthält wichtige Informationen zur Sicherheit.

Sicherheitsaspekte für den Cache für gemeinsam genutzte Klassen

Der Cache für gemeinsam genutzte Klassen wurde entwickelt, um die Cacheverwaltung und die Nutzbarkeit zu erleichtern. Die Standardsicherheitsrichtlinie ist hierbei jedoch möglicherweise nicht geeignet.

Wenn Sie den Cache für gemeinsam genutzte Klassen verwenden, müssen Sie die Standardberechtigungen für neue Dateien beachten, um die Sicherheit durch das Einschränken des Zugriffs verbessern zu können.

Datei	Standardberechtigungen
neue gemeinsam genutzte Caches	Leseberechtigungen für Gruppe und andere
Verzeichnis `javasharedresources`	Allgemeine Lese-, Schreib- und Ausführungsberechtigungen

Sie benötigen Schreibberechtigung für die Cachedatei und für das Cacheverzeichnis, um einen Cache zu löschen oder weiter zu füllen.

Dateiberechtigungen für die Cachedatei ändern

Sie können den Zugriff auf einen Cache für gemeinsam genutzte Klassen mit dem Befehl **`chmod`** einschränken.

Erforderliche Änderung	Befehl
Zugriff auf den Benutzer und die Gruppe einschränken	`chmod 770 /tmp/javasharedresources`
Zugriff auf den Benutzer einschränken	`chmod 700 /tmp/javasharedresources`
Benutzer auf Lese- und Schreibzugriff für einen bestimmten Cache einschränken	`chmod 600 /tmp/javasharedresources/` `<`*`Datei für gemeinsam genutzten Cache`*`>`
Benutzer und Gruppe auf Lese- und Schreibzugriff für einen bestimmten Cache einschränken	`chmod 660 /tmp/javasharedresources/` `<`*`Datei für gemeinsam genutzten Cache`*`>`

Weitere Informationen zum Erstellen eines Cache für gemeinsam genutzte Klassen finden Sie in „Echtzeitorientierten Cache für gemeinsam genutzte Klassen erstellen” auf Seite 44.

Verbindung zu einem Cache herstellen, für den Sie keine Zugriffsberechtigung haben

Wenn Sie versuchen, eine Verbindung zu einem Cache herzustellen, für den Sie nicht die entsprechenden Zugriffsberechtigungen haben, wird eine Fehlernachricht angezeigt:

```
JVMSHRC226E Fehler beim Öffnen der Cachedatei für eine gemeinsam genutzte Klasse
JVMSHRC220E Fehlercode für Portschicht = -302
JVMSHRC221E Fehlernachricht für Plattform: Zugriff verweigert
JVMJ9VM015W Initialisierungsfehler für Bibliothek j9shr25(11): JVMJ9VM009E J9VMDllMain
fehlgeschlagen
Java Virtual Machine konnte nicht erstellt werden
```

Kapitel 9. Fehlerbehebung und Unterstützung

Fehlerbehebung und Unterstützung für WebSphere Real Time for RT Linux

Allgemeine Problembestimmungsmethoden

Mithilfe der Fehlerbestimmung können Sie feststellen, welche Art von Fehler vorliegt und wie Sie am besten vorgehen.

Wenn Sie wissen, welche Art von Fehler vorliegt, können Sie eine oder mehrere der folgenden Tasks ausführen:
- Fehler beheben
- Passende Fehlerumgehung finden
- Erforderliche Daten für die Generierung eines Fehlerberichts für IBM erfassen

Fehlerbestimmung unter Linux

In diesem Abschnitt wird die Fehlerbestimmung unter Linux beschrieben.

Das Benutzerhandbuch für IBM SDK for Java 7 enthält nützliche Informationen zum Diagnostizieren von Problemen unter Linux:
- Linux-Umgebung einrichten und prüfen
- Allgemeine Debugging-Verfahren
- Abstürze diagnostizieren
- Debugging von Blockierungen
- Debugging von Speicherlecks
- Debugging von Leistungsproblemen

Sie finden diese Informationen an folgender Stelle: IBM SDK for Java 7 - Linux-Fehlerbestimmung.

Die folgenden Informationen sind für IBM WebSphere Real Time for RT Linux ergänzend.

Linux-Umgebung einrichten und prüfen

Prüfen Sie in IBM WebSphere Real Time for RT Linux, ob die JVM ordnungsgemäß für das Generieren eines Systemspeicherauszugs konfiguriert ist.

Linux-Systemspeicherauszüge (Kerndateien)

Bei einem Absturz ist der Linux-Systemspeicherauszug (Kerndatei) die wichtigste Quelle für Diagnosedaten. Wenn Sie sicherstellen möchten, dass diese Datei generiert wird, müssen Sie Ihre Betriebssystemeinstellungen und den verfügbaren Plattenspeicher so wie im Benutzerhandbuch für IBM SDK for Java 7 beschrieben prüfen.

Java Virtual Machine-Einstellungen

Die JVM muss so konfiguriert sein, dass sie im Fall eines Absturzes Kerndateien generiert. Führen Sie `java -Xrealtime -Xdump:what` in der Befehlszeile aus. Die Ausgabe dieser Option sieht wie folgt aus:

```
-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
```

Bei den gezeigten Werten handelt es sich um die Standardeinstellungen. Es muss mindestens die Option `events=gpf` eingestellt sein, damit bei einem Absturz eine Kerndatei generiert wird. Mit der Befehlszeilenoption **`-Xdump:system[:name1=value1,name2=value2 ...]`** können Sie Optionen ändern und festlegen.

Allgemeine Debugging-Verfahren

Da Java-Threadnamen im Betriebssystem angezeigt werden, können Sie den Befehl **ps** beim Debugging einsetzen. Wenn Sie Traceerstellungstools verwenden, müssen Sie die richtigen Befehle für IBM WebSphere Real Time for RT Linux verwenden.

Prozessinformationen prüfen

Bei der Ausführung des Befehls **ps** in IBM WebSphere Real Time for RT Linux wird die folgende Ausgabe generiert:

```
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
```

U Wählt alle Prozesse aus.

L Zeigt Threads an.

o Gibt ein vordefiniertes Format für Spalten an, die angezeigt werden sollen. Bei den angegebenen Spalten handelt es sich um die Prozess-ID, die Thread-ID, die Planungsrichtlinie, die Echtzeitthreadpriorität und den Befehl, der dem Prozess zugeordnet ist. Diese Informationen sind hilfreich, um festzustellen, welche virtuelle Maschine und welche Threads in Ihrer Anwendung zu einem bestimmten Zeitpunkt aktiv sind.

Traceerstellungstools

Unter Linux stehen die drei Traceerstellungstools **strace**, **ltrace** und **mtrace** zur Verfügung. Mit dem Befehl `man strace` werden eine ganze Reihe verfügbarer Optionen angezeigt.

strace
Das Tool 'strace' verfolgt Systemaufrufe. Sie können es für einen Prozess verwenden, der bereits verfügbar ist, oder mit einem neuen Prozess starten. 'strace' erfasst die Systemaufrufe, die von einem Programm ausgeführt werden, und die Signale, die von einem Prozess empfangen werden. Für jeden Systemaufruf werden der Name, Argumente und der Rückgabewert verwendet. Mit 'strace' können Sie ein Programm ohne die Quelle verfolgen (es ist keine Neukompilierung erforderlich). Wenn Sie 'strace' mit der Option **-f** verwenden, werden untergeordnete Prozesse verfolgt, die als Ergebnis eines verzweigten Systemaufrufs erstellt wurden. Mit 'strace' können Sie Plug-in-Probleme untersuchen oder versuchen zu verstehen, warum Programme nicht ordnungsgemäß gestartet werden.

Um 'strace' mit einer Java-Anwendung zu verwenden, geben Sie `strace java -Xrealtime <class-name>` ein.

Sie können die Traceausgabe vom Tool 'strace' in eine Datei übertragen, indem Sie die Option `-o` verwenden.

ltrace
Das Tool 'ltrace' ist variantenabhängig. Es ist dem Tool 'strace' sehr ähnlich. Dieses Tool fängt die Aufrufe der dynamischen Bibliothek durch den ausführenden Prozess ab und zeichnet sie auf. 'strace' führt für die Signale, die vom ausführenden Prozess empfangen werden, dieselbe Aktion aus.

Um 'ltrace' mit einer Java-Anwendung zu verwenden, geben Sie `ltrace java -Xrealtime <class-name>` ein.

mtrace
'mtrace' ist im GNU-Toolset enthalten. Es installiert spezielle Steuerroutinen für 'malloc', 'realloc' und 'free' und macht es möglich, dass alle Verwendungen dieser Funktionen verfolgt und in einer Datei erfasst werden. Durch diese Traceerstellung wird die Programmeffizienz verringert. Sie sollte deshalb nicht während der normalen Verwendung aktiviert werden. Um 'mtrace' verwenden zu können, setzen Sie **IBM_MALLOCTRACE** auf 1 und legen Sie **MALLOC_TRACE** so fest, dass auf eine gültige Datei verwiesen wird, in der die Traceinformationen gespeichert werden. Sie müssen über Schreibzugriff auf diese Datei verfügen.

Um 'mtrace' mit einer Java-Anwendung zu verwenden, geben Sie Folgendes ein:

```
export IBM_MALLOCTRACE=1
export MALLOC_TRACE=/tmp/file
java -Xrealtime <class-name>
mtrace /tmp/file
```

Abstürze diagnostizieren

Befolgen Sie beim Zusammenstellen von Informationen zur Ausführung von Prozessen und der Java-Umgebung vor einem Absturz diese Richtlinien.

Prozessinformationen zusammenstellen

Zeigen Sie beim Erforschen der Umstände vor dem Absturz mithilfe der Befehle **gdb** und **bt** den Stack-Trace des fehlgeschlagenen Threads an, anstatt die Kerndatei zu analysieren.

Informationen zur Java-Umgebung ermitteln

Mithilfe des Java-Speicherauszugs können Sie feststellen, welche Operationen die einzelnen Threads ausgeführt haben und welche Java-Methoden ausgeführt wurden. Gleichen Sie Funktionsadressen mit Bibliotheksadressen ab, um die Quelle des Codes zu ermitteln, der an verschiedenen Punkten ausgeführt wird.

Überprüfen Sie mithilfe der Option **-verbose:gc** den Status des Java-Heapspeichers und der Speicherbereiche für Objekte mit unbeschränkter und beschränkter Lebensdauer. Stellen Sie die folgenden Fragen:

- Gab es einen Speicherengpass in einem der Hauptspeicherbereiche, der möglicherweise den Absturz verursacht hat?
- Ist der Absturz während der Garbage-Collection aufgetreten? Dies deutet auf einen möglichen Garbage-Collection-Fehler hin.
- Ist der Absturz nach der Garbage-Collection aufgetreten? Dies deutet auf einen möglichen Datenverlust im Hauptspeicher hin.

Debugging von Leistungsproblemen

Beachten Sie beim Debugging von Leistungsproblemen die folgenden Punkte für IBM WebSphere Real Time for RT Linux zusätzlich zu den Themen im Benutzerhandbuch für IBM SDK for Java 7.

Größe von Hauptspeicherbereichen ändern

Die JVM kann durch Ändern der Größe des Heapspeichers sowie des Speichers für Objekte mit unbeschränkter und beschränkter Lebensdauer optimiert werden. Wählen Sie die richtige Größe, um die Leistung zu optimieren. Wenn Sie die richtige Größe verwenden, kann der Garbage-Collector leichter die erforderliche Auslastung bereitstellen.

Weitere Informationen zum Ändern der Größe von Hauptspeicherbereichen finden Sie in „Fehlerbehebung für den Metronom-Garbage-Collector" auf Seite 146.

JIT-Kompilierung und Leistung

Bei der Verwendung von JIT sollten Sie die Auswirkungen auf das Echtzeitverhalten berücksichtigen.

Wenn Sie vorhersehbares Verhalten, jedoch auch eine bessere Leistung benötigen, sollten Sie die Verwendung der AOT-Kompilierung (AOT - Ahead-Of-Time) in Betracht ziehen. Weitere Informationen finden Sie in „Für WebSphere Real Time for RT Linux kompilierten Code verwenden" auf Seite 40.

Bekannte Einschränkungen unter Linux

Bei Linux hat eine schnelle Entwicklung stattgefunden und es gab verschiedene Probleme bei der Interaktion der JVM und des Betriebssystems, insbesondere im Bereich der Threads.

Beachten Sie die folgenden Einschränkungen, die sich möglicherweise auf Ihr Linux-System auswirken.

Threads als Prozesse

Wenn die Anzahl der Java-Threads die maximal zulässige Anzahl an Prozessen überschreitet, hat dies möglicherweise die folgenden Folgen für Ihr Programm:

- Es erhält eine Fehlernachricht.
- Es erhält einen **SIGSEGV**-Fehler.
- Es wird gestoppt.

Weitere Informationen finden Sie in *The Volano Report* auf der Website http://www.volano.com/report/index.html.

Einschränkungen für variabel verknüpfte Stacks

Bei einer Ausführung ohne variabel verknüpfte Stacks wird unabhängig von der Einstellung für **-Xss** eine Minimalgröße von 256 KB für native Stacks für die einzelnen Threads bereitgestellt.

Auf einem Linux-System mit variabel verknüpften Stacks werden die Werte für **-Xss** verwendet. Wenn Sie ein Linux-System ohne variabel verknüpfte Stacks migrieren, müssen alle Werte für **-Xss** groß genug sein und es darf kein Minimum von 256 KB erforderlich sein.

glibc-Einschränkungen

Wenn Sie eine Nachricht erhalten, die angibt, dass die Bibliothek `libjava.so` nicht geladen werden konnte, da ein Symbol nicht gefunden wurde (z. B. __bzero), ist möglicherweise eine frühere Version der GNU C-Laufzeitbibliothek, glibc, installiert. Das SDK für die Linux-Threadimplementierung erfordert glibc Version 2.3.2 oder höher.

Einschränkungen für Schriftarten

Wenn Sie die Installation auf einem Red Hat-System durchführen, führen Sie (beispielsweise unter Linux IA32) Folgendes aus, damit der Schriftartenserver die Java-TrueType-Schriftarten finden kann:

```
/usr/sbin/chkfontpath --add opt/IBM/javawrt3/jre/lib/fonts
```

Sie müssen dies während der Installation ausführen und Sie müssen als „Root" angemeldet sein, um den Befehl ausführen zu können. Weitere Informationen zu Problemen mit Schriftarten finden Sie im SDK- und Runtime Environment-Benutzerhandbuch für Linux.

Leistungsprobleme bei Linux Red Hat MRG-Kerneln

Ein Konfigurationsproblem mit Red Hat MRG-Kerneln kann unerwartete Pausen bei Anwendungsthreads verursachen, wenn WebSphere Real Time mit aktivierter ausführlicher Garbage-Collection gestartet wird. Diese Pausen werden nicht in der Ausgabe der ausführlichen GC dokumentiert, Sie können jedoch je nach Netzkonfiguration mehrere Millisekunden dauern. JVMs, die von fern definierten LDAP-Benutzern gestartet werden, sind am meisten betroffen, da der Cachedämon des Namensservice (`nscd`) nicht gestartet wird, was zu Verzögerungen bei der

Netzübertragung führt. Beheben Sie das Problem, indem Sie `nscd` starten. Führen Sie folgende Schritte aus, um den Status des Service `nscd` zu überprüfen und das Problem zu beheben:

1. Überprüfen Sie, ob der Dämon `nscd` aktiv ist, indem Sie den folgenden Befehl eingeben:

   ```
   /sbin/service nscd status
   ```

   Wenn der Dämon nicht aktiv ist, wird folgende Nachricht angezeigt:

   ```
   nscd is stopped
   ```

2. Starten Sie den Service `nscd` als Rootbenutzer mit dem folgenden Befehl:

   ```
   /sbin/service nscd start
   ```

3. Ändern Sie die Startinformationen für den Service `nscd` als Rootbenutzer mit dem folgenden Befehl:

   ```
   /sbin/chkconfig nscd on
   ```

Der Prozess `nscd` ist jetzt aktiv und wird nach einem Warmstart automatisch gestartet.

Fehlerbestimmung für NLS

Die JVM enthält integrierte Unterstützung für verschiedene Ländereinstellungen.

Das Benutzerhandbuch für IBM SDK for Java 7 enthält nützliche Informationen zum Diagnostizieren von NLS-Problemen:

- Schriftarten - Übersicht
- Dienstprogramme für Schriftarten
- Häufig auftretende NLS-Probleme und mögliche Ursachen

Sie finden diese Informationen an folgender Stelle: IBM SDK for Java 7 - NLS-Problembestimmung.

Fehlerbestimmung für ORB

Eine der ersten Aufgaben beim Debugging eines ORB-Fehlers besteht darin zu bestimmen, ob der Fehler bei der verteilten Anwendung clientseitig oder serverseitig auftritt. Stellen Sie sich eine typische RMI-IIOP-Sitzung als eine einfache synchrone Übertragung zwischen einem Client, der den Zugriff auf ein Objekt anfordert, und einem Server, der ihn gewährt, vor.

Das Benutzerhandbuch für IBM SDK for Java 7 enthält nützliche Informationen zum Diagnostizieren von ORB-Problemen:

- ORB-Fehler bestimmen
- Stack-Trace interpretieren
- ORB-Traces interpretieren
- Häufig auftretende Probleme
- IBM ORB-Service: Daten erfassen

Sie finden diese Informationen an folgender Stelle: IBM SDK for Java 7 - ORB-Problembestimmung.

Die folgenden Informationen sind für IBM WebSphere Real Time for RT Linux ergänzend.

IBM ORB-Service: Daten erfassen

Führen Sie beim Erfassen der Java-Versionsausgabe für den Service den folgenden Befehl aus:

```
java -Xrealtime -version
```

Vortests

Wenn ein Problem auftritt, generiert ORB möglicherweise eine Ausnahmebedingung `org.omg.CORBA.*`, die Folgendes einschließt:

- Text zur Angabe der Ursache
- Nebencode
- Fertigstellungsstatus

Bevor Sie davon ausgehen, dass die Fehlerursache bei ORB liegt, prüfen Sie Folgendes:

- Das Szenario kann in ähnlicher Konfiguration reproduziert werden.
- Der JIT-Compiler ist inaktiviert.
- Es wird kein AOT-kompilierter Code verwendet

Weitere Aktionen:

- Inaktivieren Sie zusätzliche Prozessoren.
- Inaktivieren Sie simultanes Multithreading (SMT), wo dies möglich ist.
- Beseitigen Sie Speicherabhängigkeiten bei Client oder Server. Der Mangel an physischem Hauptspeicher kann die Ursache langsamer Verarbeitung, scheinbarer Blockierungen oder von Abstürzen sein. Stellen Sie sicher, dass Sie über ein angemessenes Speichervolumen verfügen, um diese Probleme zu lösen.
- Überprüfen Sie Fehler im physischen Netz wie Firewalls, Übertragungsleitungen, Router und DNS-Namensserver. Dies sind die Hauptursachen für die Ausnahmebedingung COMM_FAILURE in CORBA. Überprüfen Sie als Test den Namen Ihrer eigenen Workstation mit Ping.
- Wenn die Anwendung eine Datenbank wie DB2 verwendet, wechseln Sie zum zuverlässigsten Treiber. Um zum Beispiel den DB2 AppDriver zu isolieren, wechseln Sie zu 'Net Driver', der zwar langsamer ist und Sockets verwendet, dafür aber zuverlässiger ist.

Fehlerbehebung von OutOfMemory-Fehlern

Handhabung von Ausnahmebedingungen OutOfMemoryError, Speicherlecks und ausgeblendeten Hauptspeicherzuordnungen.

Allgemeine Fehlerbehebungsinformationen zum Metronom-Garbage-Collector finden Sie in „Fehlerbehebung für den Metronom-Garbage-Collector" auf Seite 146.

Fehler aufgrund abnormaler Speicherbedingungen (OutOfMemoryErrors) diagnostizieren

Das Diagnostizieren von Ausnahmebedingungen OutOfMemoryError im Metronom-Garbage-Collector kann aufgrund des periodischen Charakters des Garbage-Collectors komplexer sein als in einer Standard-JVM.

Die Merkmale der verschiedenen Heapspeichertypen werden in „Speicherverwaltung" auf Seite 13 beschrieben. Im Allgemeinen benötigt eine RTSJ-Anwendung ungefähr 20 % mehr Heapspeicher als eine Standard-Java-Anwendung.

Die JVM erzeugt standardmäßig die folgende Diagnoseausgabe, wenn ein nicht abgefangener OutOfMemoryError auftritt:

- Kurzspeicherauszug; siehe „Speicherauszugsagenten verwenden" auf Seite 123.
- Heapspeicherauszug; siehe „Heapspeicherauszug verwenden" auf Seite 132.
- Java-Speicherauszug; siehe „Java-Speicherauszug verwenden" auf Seite 127.
- Systemspeicherauszug; siehe „Systemspeicherauszüge und die Anzeigefunktion für Speicherauszüge verwenden" auf Seite 136.

Die Speicherauszugsdateinamen werden in der Konsolausgabe angegeben:

```
JVMDUMP006I Speicherauszugsereignis "systhrow", Detail "java/lang/OutOfMemoryError" wird verarbeitet
- bitte warten.
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 Speicherauszugsereignis "systhrow", Detail "java/lang/OutOfMemoryError" wurde verarbeitet.
```

Der in der Konsolausgabe gezeigte und im Java-Speicherauszug enthaltene Java-Backtrace gibt an, wo der OutOfMemoryError in der Java-Anwendung aufgetreten ist. Im nächsten Schritt muss ermittelt werden, welcher RTSJ-Hauptspeicherbereich voll ist. Die JVM-Speicherverwaltungskomponente gibt einen Tracepunkt aus, der die Größe, die Klassenblockadresse und den Speicherbereichsnamen der fehlgeschlagenen Zuordnung angibt. Diesen Tracepunkt finden Sie im Kurzspeicherauszug:

```
<< Zeilen ausgelassen... >>
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
```

Die Tracepunkt-ID und Datenfelder weichen je nach dem zugeordneten Objekttyp möglicherweise von den angezeigten Elementen ab. In diesem Beispiel zeigt der Tracepunkt, dass der Zuordnungsfehler auftrat, als die Anwendung versuchte, ein 33,6 MB großes Objekt des Typs `class 0x81312d8` im Speichersegment `id=0x809c5f0` des Metronomheapspeichers zuzuordnen.

Sie können anhand der Speicherverwaltungsinformationen im Java-Speicherauszug ermitteln, welcher Hauptspeicherbereich (RTSJ) betroffen ist:

```
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
<< Zeilen aus Gründen der Übersichtlichkeit entfernt >>
```

Sie können den Objekttyp ermitteln, der zugeordnet wird, indem Sie den Klassenabschnitt im Java-Speicherauszug prüfen:

```
NULL           ------------------------------------------------------------------
0SECTION       CLASSES subcomponent dump routine
NULL           ================================
<< Zeilen ausgelassen... >>
1CLTEXTCLLOD    ClassLoader loaded classes
2CLTEXTCLLOAD    Loader *System*(0xF182BB80)
<< Zeilen ausgelassen... >>
3CLTEXTCLASS      [C(0xF1632D80)
```

Die Informationen im Java-Speicherauszug bestätigen, dass die versuchte Zuordnung für ein Zeichenarray im normalen Heapspeicher (`ID=0xF288B584`) vorgenommen werden sollte und dass die in der entsprechenden Zeile `1STHEAP` angegebene Gesamtgröße zugeordneten Heapspeichers 67108864 Dezimalbyte oder 0x04000000 Hexadezimalbyte ist, d. h., sie beträgt 64 MB.

In diesem Beispiel ist die fehlgeschlagene Zuordnung in Beziehung zur Gesamtgröße des Heapspeichers groß. Wenn Ihre Anwendung 33 MB große Objekte erstellen soll, ist der nächste Schritt die Erhöhung des Heapspeichers mit der Option **-Xmx**.

Gewöhnlich ist die fehlgeschlagene Zuordnung in Beziehung zur Gesamtgröße des Heapspeichers klein. Dies liegt daran, dass vorherige Zuordnungen den Heapspeicher auffüllen. In diesen Fällen ist der nächste Schritt die Verwendung des Heapspeicherauszugs, um die Speicherkapazität zu prüfen, die vorhandenen Objekten zugeordnet ist.

Der Heapspeicherauszug ist eine komprimierte Binärdatei, die eine Liste aller Objekte mit der zugehörigen Objektklasse, der Größe und Verweisen enthält. Analysieren Sie den Heapspeicherauszug mit dem Tool 'Memory Dump Diagnostics for Java' (MDD4J), das über IBM Support Assistant (ISA) heruntergeladen werden kann.

Mit MDD4J können Sie einen Heapspeicherauszug laden und Baumstrukturen nach Objekten durchsuchen, von denen Sie vermuten, dass sie viel Heapspeicher belegen. Das Tool bietet verschiedene Ansichten für Objekte im Heapspeicher. Beispielsweise kann MDD4J eine Ansicht anzeigen, die wahrscheinliche Leckkandidaten aufführt und die fünf Objekte und Pakete angibt, die am meisten zur Größe des Heapspeichers beitragen. Durch die Auswahl der Baumstrukturansicht erhalten Sie weitere Informationen zur Art des Containerobjektlecks.

Standardmäßig wird eine einzelne Heapspeicherauszugsdatei mit allen Objekten in allen RTSJ-Speicherbereichen erzeugt. Fordern Sie mit der Befehlszeilenoption **-Xdump:heap:request=multiple** einen separaten Heapspeicherauszug für jeden Speicherbereich an. Bei mehreren Speicherauszügen können Sie eine Gruppe von Objekten prüfen, die in einem bestimmten Hauptspeicherbereich zugeordnet sind. Sie geben die Heapspeicherauszüge durch den in der Konsolenausgabe aufgelisteten Dateinamen an:

```
JVMDUMP006I Speicherauszugsereignis "uncaught", Detail "java/lang/OutOfMemoryError" wird verarbeitet
- bitte warten.
<< Zeilen ausgelassen... >>
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
<< Zeilen ausgelassen... >>
JVMDUMP013I Speicherauszugsereignis "uncaught", Detail "java/lang/OutOfMemoryError" wurde verarbeitet.
Exception in thread "RTJ Memory Consumer (thread_type=Realtime)" java.lang.OutOfMemoryError
 at tests.com.ibm.jtc.ras.runnable.DepleteMemory.depleteMemory(DepleteMemory.java:57)
<< Zeilen ausgelassen... >>
```

Speicherverwaltung durch die IBM JVM

Die IBM JVM benötigt Speicher für mehrere verschiedene Komponenten, einschließlich Speicherbereiche für Klassen, kompilierten Code, Java-Objekte, Java-Stacks und JNI-Stacks. Einige dieser Speicherbereiche müssen in zusammenhängendem Speicher angeordnet sein. Andere Speicherbereiche können in kleinere Speicherbereiche segmentiert und verknüpft werden.

Dynamisch geladene Klassen und kompilierter Code werden werden in segmentierten Speicherbereichen für dynamisch geladene Klassen gespeichert. Klassen sind in beschreibbare Speicherbereiche (RAM-Klassen) und Nur-Lese-Speicherbereiche (ROM-Klassen) unterteilt. Während der Ausführung wird für den Klassencache Speicher zugeordnet, er wird beim Anwendungsstart jedoch nicht notwendigerweise in einen zusammenhängenden Speicherbereich geladen. Beim Verweisen auf Klassen durch die Anwendung werden Klassen und kompilierter Code im Klassencache dem Speicher zugeordnet. Die ROM-Komponente der Klasse wird von mehreren auf diese Klasse verweisenden Prozessen gemeinsam genutzt. Die RAM-Komponente der Klasse wird in den segmentierten Speicherbereichen für dynamisch geladene Klassen erstellt, wenn die JVM zum ersten Mal auf die Klasse verweist. Der mit AOT kompilierte Code für die Methoden einer Klasse im Klassencache wird in einen Speicherbereich für ausführbaren dynamischen Code kopiert, weil dieser Code von Prozessen nicht gemeinsam genutzt wird. Klassen, die nicht aus dem Klassencache geladen werden, ähneln zwischengespeicherten Klassen, außer dass die ROM-Klasseninformationen in segmentierten Speicherbereichen für dynamisch geladene Klassen erstellt werden. Dynamisch generierter Code wird in denselben Speicherbereichen für dynamischen Code gespeichert, die den AOT-Code für zwischengespeicherte Klassen enthalten.

Alle Java-Objekte werden im Standardheapspeicher gespeichert, wenn die JVM ohne die Option **-Xrealtime** ausgeführt wird. Wird die Option **-Xrealtime** verwendet, können Objekte auch aus zwei zusätzlichen Speicherbereichen (Speicher für Objekte mit unbeschränkter Lebensdauer und Speicher für Objekte mit beschränkter Lebensdauer) zugeordnet werden.

Der Stack für jeden Java-Thread kann ein segmentierter Speicherbereich sein. Der JNI-Stack für jeden Thread belegt einen zusammenhängenden Speicherbereich.

Führen Sie Ihre JVM mit der Option **-verbose:sizes** aus, um zu ermitteln, wie die JVM konfiguriert ist. Diese Option gibt Informationen zu Speicherbereichen aus, deren Größe Sie verwalten können. Für nicht zusammenhängende Speicherbereiche wird ein Inkrement ausgegeben, das beschreibt, wie viel Speicher angefordert wird, wenn der Bereich vergrößert werden muss.

Es folgt eine Beispielausgabe, die die Optionen **-Xrealtime -verbose:sizes** verwendet:

```
-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
```

Dieses Beispiel gibt an, dass das RAM-Klassensegment anfänglich 0 ist, jedoch bei Bedarf um 32-KB-Blöcke vergrößert wird. Das ROM-Klassensegment ist anfänglich 0 und wird bei Bedarf um 128-KB-Blöcke vergrößert. Sie können diese Größen mit den Optionen **-Xmca** und **-Xmco** steuern. RAM- und ROM-Klassensegmente werden bei Bedarf vergrößert. Daher brauchen Sie diese Optionen in der Regel nicht zu ändern.

Der Speicher für Objekte mit unbeschränkter Lebensdauer ist ein zusammenhängender Bereich, dem möglicherweise vorab mehr Speicherplatz zugeordnet werden muss. In diesem Beispiel werden dem Speicherbereich für Objekte mit unbeschränkter Lebensdauer vorab 16 MB zugeordnet. Wenn Sie versuchen, mehr als 16 MB von Objekten in diesen Speicherbereich für Objekte mit unbeschränkter Lebensdauer zu schreiben, empfangen Sie die Ausnahmebedingung OutOfMemory, weil für diesen Speicherbereich definitionsgemäß keine Garbage-Collection durchgeführt wird.

Der Speicherbereich für Objekte mit beschränkter Lebensdauer ist zusammenhängend und in diesem Beispiel mit 8 MB vorab zugeordnet. Wenn bei der Programmausführung viele Speicherbereiche für Objekte mit beschränkter Lebensdauer aktiv sind, müssen Sie möglicherweise einen größeren Speicherbereich für Objekte mit beschränkter Lebensdauer angeben.

Mit dem Dienstprogramm 'admincache' können Sie ermitteln, wie groß Ihr dem Speicher zugeordnete Bereich ist, wenn Sie den Klassencache verwenden. Es folgt ein Beispiel für die Ausgabe vom Befehl `admincache -Xrealtime -printStats -nologo`:

```
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
```

Die Cachegröße gibt an, dass der dem Speicher zugeordnete Bereich 7 MB etwas überschreitet. Die ROM-Klasse und die AOT-Byte belegen den meisten Speicherplatz, jeweils etwas mehr als 3 MB.

Beispiel für OutOfMemoryError im Speicherbereich für Objekte mit unbeschränkter Lebensdauer

Dieses Beispiel zeigt, wie ein OutOfMemoryError im Speicherbereich für Objekte mit unbeschränkter Lebensdauer ermittelt werden kann, und beschreibt die zu ergreifenden Schritte, um das Problem zu vermeiden.

Der Kurzspeicherauszug zeigt, dass die beiden Zuordnungsanforderungen im Speicherbereich für Objekte mit unbeschränkter Lebensdauer id=0x809dd1c fehlgeschlagen sind:

```
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
```

Der Java-Speicherauszug zeigt, dass der Speicherbereich für Objekte mit unbeschränkter Lebensdauer voll ist:

```
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
<< Zeilen ausgelassen... >>
1STSEGTYPE    Object Memory
NULL          segment  start    alloc    end      type     bytes
1STSEGSUBTYPE  Immortal Segment  ID=0809DD1C
1STSEGMENT     0809D510 B279D008 B379D008 B379D008  00001008 1000000
```

Eine MDD4J-Analyse zeigt, dass eine sehr große verkettete Liste (LinkedList) zugeordnet wurde, die einen hohen Anteil des verfügbaren Hauptspeichers belegt.

Ihnen wird empfohlen, die Anzahl der im Speicherbereich für Objekte mit unbeschränkter Lebensdauer zugeordneten Objekte zu minimieren, weil für Objekte im Bereich für Objekte mit unbeschränkter Lebensdauer keine Garbage-Collection durchgeführt wird. Die gängigste Verwendung des Speichers für Objekte mit unbeschränkter Lebensdauer ist das Laden von Klassen, das eine begrenzte Aktivität ist, die am häufigsten während der JVM- und Anwendungsinitialisierung auftritt. Anwendungen mit einer hohen Anzahl geladener Klassen (oder einer anderen Verwendung des Speichers für Objekte mit unbeschränkter Lebensdauer) können den Speicherbereich für Objekte mit unbeschränkter Lebensdauer vergrößern. Verwenden Sie hierzu die Option **-Xgc:immortalMemorySize=<Größe>**. Die Standardgröße des Speicherbereichs für Objekte mit unbeschränkter Lebensdauer beträgt 16 MB.

Wenn die Vergrößerung des Speicherbereichs für Objekte mit unbeschränkter Lebensdauer den OutOfMemoryError für den Speicher für Objekte mit unbeschränkter Lebensdauer nur verzögert, prüfen Sie das Muster für die kontinuierliche Zuordnung von Daten für Objekte mit unbeschränkter Lebensdauer, das sich auf das Laden von Klassen oder andere Anwendungsobjekte bezieht.

Beispiel für OutOfMemoryError im Speicherbereich für Objekte mit beschränkter Lebensdauer

Dieses Beispiel zeigt, wie ein OutOfMemoryError im Speicherbereich für Objekte mit beschränkter Lebensdauer ermittelt werden kann, und beschreibt die zu ergreifenden Schritte, um das Problem zu vermeiden.

Erzeugen Sie mit der Befehlszeilenoption **-Xdump:heap:request=multiple** separate Speicherauszüge für jeden Hauptspeicherbereich:

```
VMDUMP006I Speicherauszugsereignis "uncaught", Detail "java/lang/OutOfMemoryError" wird verarbeitet
- bitte warten.
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 Speicherauszugsereignis "uncaught", Detail "java/lang/OutOfMemoryError" wurde verarbeitet.
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)
<< Zeilen ausgelassen... >>
```

Der Kurzspeicherauszug zeigt, dass die beiden Zuordnungsanforderungen im Speicherbereich für Objekte mit beschränkter Lebensdauer id=0x809dd10 fehlgeschlagen sind:

```
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
```

Der Java-Speicherauszug zeigt, dass die zugeordnete Hauptspeicherbereichsgröße für den Speicherbereich für Objekte mit beschränkter Lebensdauer id=0x809dd10 recht klein ist, und zwar nur 60 KB. Vergrößern Sie in diesem Fall den Speicherbereich für Objekte mit beschränkter Lebensdauer im Anwendungscode.

```
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
```

Im Beispiel-Java-Speicherauszug scheint der Speicherbereich für Objekte mit beschränkter Lebensdauer leer zu sein. Dies ist der Fall, weil der Java-Speicherauszug erzeugt wird, wenn der OutOfMemoryError die JVM erreicht. Zu diesem Zeitpunkt wurde der Bereich verlassen und bereinigt. Sie können einen Java-Speicherauszug über die Befehlszeilenoption **-Xdump:java:events=throw,filter=java/lang/OutOfMemoryError** am Fehlerpunkt erzeugen. Durch die Verwendung dieser Option wird der freie Speicherplatz im Speicherbereich für Objekte mit beschränkter Lebensdauer ordnungsgemäß zurückgemeldet.

Es ist auch möglich, den für den Speicher für Objekte mit beschränkter Lebensdauer verfügbaren Gesamtspeicherplatz zu belegen. Vergrößern Sie in diesem Fall den

Speicherbereich für Objekte mit beschränkter Lebensdauer über die Befehlszeilenoption **-Xgc:scopedMemoryMaximumSize=<Größe>**. Die Standardgröße des Speicherbereichs für Objekte mit beschränkter Lebensdauer beträgt 8 MB. Wenn der für den Speicher für Objekte mit beschränkter Lebensdauer verfügbare Gesamtspeicherplatz belegt ist, werden auf der Konsole verschiedene Nachrichten angezeigt, z. B.:

```
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)
```

Probleme in mehreren Heapspeichern diagnostizieren

Mit den im Java-Speicherauszug bereitgestellten Adressbereichen und den Belegungsinformationen im Heapspeicherauszug können Sie Fehler aufgrund abnormaler Speicherbedingungen (OutOfMemoryErrors) in mehreren RTSJ-Hauptspeicherbereichen analysieren.

Im folgenden Java-Speicherauszug reicht das Segment für Objekte mit unbeschränkter Lebensdauer von 0xB281C008 bis 0xB381C008 und das normale Heapsegment reicht von 0xB381D008 bis 0xB781D008:

```
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
<< Zeilen ausgelassen... >>
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
```

Der Heapspeicherauszug ist eine komprimierte Binärdatei, die eine Liste aller Objekte mit der zugehörigen Objektklasse, der Größe und Verweisen enthält. Analysieren Sie den Heapspeicherauszug mit dem Tool 'Memory Dump Diagnostics for Java' (MDD4J), das über IBM Support Assistant (ISA) heruntergeladen werden kann.

Sie können mithilfe der von MDD4J aufgelisteten Objektspeicherpositionen den Hauptspeicherbereich ermitteln, in dem sich ein Objekt befindet. Adressen im Bereich 0xB28*nnnnn* befinden sich im Speicherbereich für Objekte mit unbeschränkter Lebensdauer. Adressen im Bereich 0xB61*nnnnn* befinden sich im normalen Heapspeicher.

Speicherlecks vermeiden

Der Garbage-Collector verarbeitet keine Speicherbereiche für Objekte mit unbeschränkter oder beschränkter Lebensdauer. Im Fall von Speicher für Objekte mit unbeschränkter Lebensdauer wird der Speicher nur freigegeben, wenn die JVM beendet wird. Speicherbereiche für Objekte mit beschränkter Lebensdauer werden nur freigegeben, nachdem ihr Referenzzähler auf null gesunken ist. Tasks mit langer Laufzeit in diesen Kontexten müssen so geschrieben werden, dass nach dem Taskbeginn kein zusätzlicher Speicher aus dem Speicherbereich für Objekte mit unbeschränkter Lebensdauer zugeordnet wird.

Das Laden von Klassen belegt einen kleinen Anteil des Speichers für Objekte mit unbeschränkter Lebensdauer. Für diese Klassen wird in der Echtzeitumgebung keine Garbage-Collection durchgeführt. Durch das Laden von Klassen, die von der Anwendung nicht benötigt werden, kann die Anwendung mehr Speicher für Objekte mit unbeschränkter Lebensdauer verwenden als notwendig.

Wenn Ihre Anwendung Klassen enthält, die die Schnittstelle Serializable implementieren, passen Sie die Anfangsgröße des Speichers für Objekte mit unbeschränkter Lebensdauer mit Hinblick auf den Speicherbedarf von generierten Klassen an. Jeder Konstruktor verfügt über ein generiertes Objekt pro Klasse im Format "GeneratedSerializationConstructorAccessorXXX" (wobei XXX eine Zahl ist), das bei seiner ersten Serialisierung in den Speicher für Objekte mit unbeschränkter Lebensdauer geladen wird.

Vermeiden Sie die Verwendung von Speicher für Objekte mit unbeschränkter Lebensdauer, weil für Objekte, die aus dem Speicher für Objekte mit unbeschränkter Lebensdauer zugeordnet wurden, keine Garbage-Collection durchgeführt wird. Erwägen Sie Objektpooling im Speicher für Objekte mit unbeschränkter Lebensdauer, wenn der Speicherbereich für Objekte mit unbeschränkter Lebensdauer mehr als gelegentlich verwendet wird.

Zuordnung von ausgeblendetem Speicher durch Sprachenfunktionen

Vermeiden Sie bei einem Speicherkontext für Objekte mit beschränkter bzw. unbeschränkter Lebensdauer die Sprachenfunktionen von Variablenargumenten, weil diese Methoden ausgeblendeten Speicher zuordnen.

Variablenargumente (vararg)

Die Sprache Java implementiert Variablenargumente, indem sie sie als Array an die Methode übergibt. Der Compiler vereinfacht das Aufrufen von Variablenargumentmethoden durch die Erstellung und Initialisierung des Arrays.

Der Speicher kann durch den Aufruf einer Variablenargumentmethode in einem Speicherkontext für Objekte mit beschränkter bzw. unbeschränkter Lebensdauer verloren gehen. Verwenden Sie in Speicherkontexten für Objekte mit beschränkter bzw. unbeschränkter Lebensdauer keine Variablenargumente. Erstellen Sie anstelle dessen explizit ein Array und verwenden Sie es anstatt der Variablenargumente.

Es folgen zwei Beispiele mit äquivalenten Arten, eine Variablenargumentmethode aufzurufen:

```
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;
    }
}
```

Das zweite Beispiel wird bevorzugt. Da die doppelte Array-Zuordnung im Code sichtbar wird, kann die Zuordnung in einen bestimmten Hauptspeicherbereich geleitet werden.

Verkettung von Zeichenfolgen

Das Hinzufügen einer Zeichenfolge zum Erzeugen von längeren Zeichenfolgen ist über java.lang.StringBuilder-Objekte implementiert und erfordert die Zuordnung von Hauptspeicher.

Auto-Boxing

Zum Auto-Boxing gehört die Erstellung eines Objekts, das einen Basistyp enthält. Dies erfordert erfordert Zuordnungen von Hauptspeicher.

Reflexion in Speicherkontexten verwenden

Wenn ein Konstruktorobjekt in einem Speicherbereich für Objekte mit beschränkter Lebensdauer erstellt wurde, kann es nur im selben Bereich oder in einem untergeordneten Bereich verwendet werden. Alle Versuche, dieses Konstruktorobjekt im Speicher für Objekte mit unbeschränkter Lebensdauer, im Heapspeicher oder in einem übergeordneten Bereichsspeicher zu verwenden, schlagen fehl.

Die Ausnahmebedingung, die ausgelöst wird, wenn in Speicherkontexten Reflexion aufgetreten ist, ähnelt Folgendem:

```
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)
```

Sie können diese Einschränkung umgehen, indem Sie den Konstruktor im selben Bereich verwenden, dem er zugeordnet wurde.

Untergeordnete Klassen mit Speicherbereichen für Objekte mit beschränkter Lebensdauer verwenden

Bei der Verwendung von untergeordneten Klassen im Kontext von Speicherbereichen für Objekte mit beschränkter Lebensdauer müssen Sie sorgfältig vorgehen, wenn Sie untergeordnete Klassenobjekte instanziieren und sich die über- und untergeordneten Objekte in unterschiedlichen Hauptspeicherbereichen befinden. Der vom Compiler generierte Code, der im ursprünglichen Quellcode nicht sichtbar ist, verursacht einen IllegalAssignmentError, wenn das untergeordnete Objekt einen Verweis auf das übergeordnete Objekt nicht speichern kann.

Ein Objekt einer untergeordneten Klasse muss in der Lage sein, einen impliziten Verweis auf das Objekt der übergeordneten Klasse zu speichern. Wenn der Verweis gegen die RTSJ-Speicherverweisregeln verstößt, wird ein IllegalAssignmentError generiert.

Die meisten untergeordneten Klassen (einschließlich lokale und anonyme untergeordnete Klassen) enthalten ein vom Compiler generiertes (synthetisches) nicht statisches Feld für die Instanz der lexikalisch einschließenden übergeordneten Klasse. Die einzige Ausnahme tritt auf, wenn eine Instanz einer untergeordneten Klasse über kein einschließendes übergeordnetes Objekt wie ein anonymes Klassenobjekt verfügt, das in einem statischen Initialisierungsoperatorblock instanziiert ist. Das synthetische Feld des untergeordneten Objekts enthält einen Verweis auf das übergeordnete Objekt. Dies wird vom Compiler für den Java-Programmierer implementiert. Das Feld ist im ursprünglichen Quellcode nicht sichtbar. Es ist jedoch möglich, ähnlichen Code mithilfe von statischen verschachtelten Klassen mit einem Verweis zu schreiben, der sichtbar ist. Verstößt der implizite Verweis gegen die RTSJ-Speicherverweisregeln, wird ein IllegalAssignmentError ausgelöst, wenn das untergeordnete Objekt bei seiner Erstellung versucht, den Verweis auf das übergeordnete Objekt zu speichern.

Im Allgemeinen können Sie nicht gegen die RTSJ-Speicherverweisregeln verstoßen, wenn Sie untergeordnete Klassen verwenden. Sie können kein untergeordnetes Objekt erstellen, wenn ein Verweis auf das zugeordnete übergeordnete Objekt gegen die RTSJ-Speicherverweisregeln verstößt. Diese Regel bedeutet, dass ein dem Speicher für Objekte mit unbeschränkter Lebensdauer oder im Heapspeicher zugeordnetes untergeordnetes Objekt keinen Verweis auf ein übergeordnetes Objekt im Speicher für Objekte mit beschränkter Lebensdauer haben kann. Ein untergeordnetes Objekt im Speicher für Objekte mit beschränkter Lebensdauer kann einen Verweis auf ein übergeordnetes Objekt im Speicher für Objekte mit beschränkter Lebensdauer haben, das übergeordnete Objekt muss jedoch aus demselben Speicherbereich für Objekte mit beschränkter Lebensdauer oder aus einem übergeordneten Speicherbereich für Objekte mit beschränkter Lebensdauer zugeordnet werden.

Es gibt Fehlerumgehungen, einschließlich folgende:

- Statische verschachtelte Klassen verwenden, um den impliziten Verweis zu eliminieren.
- Hauptspeicherbereiche auswählen, um sicherzustellen, dass die Beziehungen von unter- und übergeordneten Objekten nicht gegen die Einschränkungen für Verweise auf Hauptspeicherbereiche verstoßen.

Diagnosetools verwenden

Ihnen stehen zahlreiche Diagnosetools für das Diagnostizieren von Problemen mit IBM WebSphere Real Time for RT Linux JVM zur Verfügung.

IBM SDK for Java 7 bietet zahlreiche Diagnosetools für das Diagnostizieren von Problemen mit IBM WebSphere Real Time for RT Linux JVM. In diesem Abschnitt werden die verfügbaren Tools eingeführt und Links zu weiteren Informationen zur Verwendung der Tools bereitgestellt.

Bei der Verwendung der SDK-Diagnosetools müssen Sie einen wichtigen Punkt beachten. Wenn Sie die Echtzeit-JVM aufrufen, verwenden Sie die folgende Option:

```
java -Xrealtime
```

Diese Option muss verwendet werden, wenn Sie Diagnosetools für die Echtzeit-JVM ausführen. Sollen z. B. die registrierten Speicherauszugsagenten für IBM WebSphere Real Time for RT Linux JVM angezeigt werden, geben Sie Folgendes ein:

```
java -Xrealtime -Xdump:what
```

Weitere Unterschiede bei der Verwendung dieser Tools mit IBM WebSphere Real Time for RT Linux werden als ergänzende Informationen geliefert. Außerdem wird eine Musterausgabe bereitgestellt, um Sie bei der Diagnose zu unterstützen.

Eine Zusammenfassung der von IBM SDK for Java 7 generierten Diagnoseinformationen finden Sie in Zusammenfassung der Diagnoseinformationen.

Speicherauszugsagenten verwenden

Speicherauszugsagenten werden während der JVM-Initialisierung eingerichtet. Mit ihrer Hilfe können Sie anhand von Ereignissen, die in der JVM auftreten (z. B. Garbage-Collection, Threadstart oder JVM-Beendigung), Speicherauszüge auslösen oder ein externes Tool starten.

Das Benutzerhandbuch für IBM SDK for Java 7 enthält nützliche Informationen zu Speicherauszugsagenten:

- Option **-Xdump** verwenden
- Speicherauszugsagenten
- Speicherauszugsereignisse
- Erweiterte Steuerung von Speicherauszugsagenten
- Tokens für Speicherauszugsagenten
- Standardspeicherauszugsagenten
- Speicherauszugsagenten entfernen
- Umgebungsvariablen für Speicherauszugsagenten
- Signalzuordnungen
- Standardpositionen von Speicherauszugsagenten

Sie finden diese Informationen an folgender Stelle: IBM SDK for Java 7 - Speicherauszugsagenten verwenden.

Ergänzende Informationen für IBM WebSphere Real Time for RT Linux finden Sie an folgender Stelle:

Speicherauszugsereignisse

Speicherauszugsagenten werden durch Ereignisse ausgelöst, die während des JVM-Betriebs auftreten. Bei IBM WebSphere Real Time for RT Linux beträgt der Standardwert für das langsame Ereignis 5 Millisekunden.

Einige Ereignisse können gefiltert werden, um die Relevanz der Ausgabe zu verbessern. Weitere Informationen finden Sie in „Option 'filter'" auf Seite 125.

Anmerkung: Die Ereignisse 'unload' und 'expand' treten in WebSphere Real Time zurzeit nicht auf. Klassen befinden sich im Immortal Memory und können nicht entladen werden.

Anmerkung: Die Ereignisse 'gpf' und 'abort' können keinen Heapspeicherauszug auslösen und den Heapspeicher nicht vorbereiten (request=prepwalk) oder komprimieren (request=compact).

In der folgenden Tabelle sind Ereignisse aufgeführt, die als Auslöser für Speicherauszugsagenten verfügbar sind:

Event	Auslöser	Filteroperation
gpf	Es gibt ein Problem beim allgemeinen Zugriffsschutz (General Protection Fault, GPF).	
Benutzer	Die JVM empfängt das Signal SIGQUIT vom Betriebssystem.	
abort	Die JVM empfängt das Signal SIGABRT vom Betriebssystem.	
vmstart	Die virtuelle Maschine wird gestartet.	
vmstop	Die virtuelle Maschine wird gestoppt.	Filterung nach Exit-Code, z. B. **`filter=#129..#192#-42#255`**
load	Eine Klasse wird geladen.	Filterung nach Klassenname, z. B. **`filter=java/lang/String`**
unload	Eine Klasse wird entladen.	
throw	Es wird eine Ausnahmebedingung ausgelöst.	Filterung nach Ausnahmeklassenname, z. B. **`filter=java/lang/OutOfMem*`**
catch	Es wird eine Ausnahmebedingung abgefangen.	Filterung nach Ausnahmeklassenname, z. B. **`filter=*Memory*`**
uncaught	Eine Java-Ausnahmebedingung wird nicht von der Anwendung abgefangen.	Filterung nach Ausnahmeklassenname, z. B. **`filter=*MemoryError`**
systhrow	Eine Java-Ausnahmebedingung steht kurz davor, von der JVM ausgelöst zu werden. Dies unterscheidet sich vom Ereignis 'throw', weil es nur für Fehlerbedingungen ausgelöst wird, die intern in der JVM erkannt werden.	Filterung nach Ausnahmeklassenname, z. B. **`filter=java/lang/OutOfMem*`**
thrstart	Ein neuer Thread wird gestartet.	
blocked	Ein Thread wird geblockt.	
thrstop	Ein Thread wird gestoppt.	

Event	Auslöser	Filteroperation
fullgc	Es wird ein Garbage-Collection-Zyklus gestartet.	
slow	Ein Thread benötigt mehr als 5 ms, um auf eine interne JVM-Anforderung zu antworten.	Ändert die für ein Ereignis benötigte Zeit, ab der es als langsam gilt; Beispiel: **filter=#300ms** führt zu einer Auslösung, wenn ein Thread mehr als 300 ms benötigt, um auf eine interne JVM-Anforderung zu antworten.
allocation	Es wird ein Java-Objekt mit einer Größe zugeordnet, die der angegebenen Filterspezifikation entspricht.	Filterung nach Objektgröße; es muss ein Filter angegeben werden. Beispiel: **filter=#5m** führt bei Objekten mit einer Größe von mehr als 5 MB zu einer Auslösung. Es werden auch Bereichsangaben unterstützt; beispielsweise führt **filter=#256k..512k** bei Objekten mit einer Größe zwischen 256 KB und 512 KB zu einer Auslösung.
traceassert	In der JVM ist ein interner Fehler aufgetreten.	Nicht zutreffend.
corruptcache	Die JVM stellt fest, dass der Cache für gemeinsam genutzte Klassen beschädigt ist.	Nicht zutreffend.

Option 'filter'

Einige JVM-Ereignisse treten während der Lebensdauer einer Anwendung unzählige Male auf. Speicherauszugsagenten können mithilfe von Filtern und Bereichen die übermäßige Erstellung von Speicherauszügen verhindern.

Platzhalterzeichen

Sie können in einem Ausnahmebedingungsereignisfilter ein Platzhalterzeichen verwenden, indem Sie nur am Anfang oder Ende des Filters einen Stern angeben. Der folgende Befehl funktioniert nicht, weil der zweite Stern nicht am Ende des Filters steht:

```
-Xdump:java:events=vmstop,filter=*InvalidArgumentException#*.myVirtualMethod
```

Damit dieser Filter funktioniert, muss er wie folgt geändert werden:

```
-Xdump:java:events=vmstop,filter=*InvalidArgumentException#MyApplication.*
```

Klassenlade- und Ausnahmebedingungsereignisse

Sie können Klassenladeereignisse (load) und Ausnahmebedingungsereignisse (throw, catch, uncaught, systhrow) nach dem Java-Klassennamen filtern:

```
-Xdump:java:events=throw,filter=java/lang/OutOfMem*
-Xdump:java:events=throw,filter=*MemoryError
-Xdump:java:events=throw,filter=*Memory*
```

Sie können die Ausnahmebedingungsereignisse throw, uncaught und systhrow nach dem Java-Methodennamen filtern:

```
-Xdump:java:events=throw,filter=ExceptionClassName[#ThrowingClassName.throwing
MethodName[#stackFrameOffset]]
```

Optionale Angaben stehen in eckigen Klammern.

Sie können 'catch'-Ausnahmebedingungsereignisse nach dem Java-Methodennamen filtern:

```
-Xdump:java:events=catch,filter=ExceptionClassName[#CatchingClassName.catchingMethodName]
```

Optionale Angaben stehen in eckigen Klammern.

Ereignis vmstop

Sie können das JVM-Systemabschlussereignis mithilfe eines oder mehrerer Exit-Codes filtern:

```
-Xdump:java:events=vmstop,filter=#129..192#-42#255
```

Ereignis 'slow'

Sie können das Ereignis 'slow' filtern, um den Standardschwellenwert von 5 ms für die Zeit zu ändern:

```
-Xdump:java:events=slow,filter=#300ms
```

Sie können den Filter auf eine Zeit setzen, die unter der Standardzeit liegt.

Ereignis 'allocation'

Sie müssen das Ereignis 'allocation' filtern, um die Größe von Objekten anzugeben, die einen Auslöser verursachen. Sie können die Filtergröße auf einen Wert von null bis zum Maximalwert eines 32-Bit-Zeigers auf 32-Bit-Plattformen bzw. bis zum Maximalwert eines 64-Bit-Zeigers auf 64-Bit-Plattformen setzen. Wird der niedrigere Filterwert auf null gesetzt, löst dies einen Speicherauszug für alle Zuordnungen aus.

Geben Sie beispielsweise Folgendes an, um Speicherauszüge für Zuordnungen mit einer Größe von mehr als 5 MB auszulösen:

```
-Xdump:stack:events=allocation,filter=#5m
```

Geben Sie Folgendes an, um Speicherauszüge für Zuordnungen mit einer Größe zwischen 256 KB und 512 KB auszulösen:

```
-Xdump:stack:events=allocation,filter=#256k..512k
```

Sonstige Ereignisse

Wenn Sie einen Filter auf ein Ereignis anwenden, das keine Filterung unterstützt, wird der Filter ignoriert.

Option 'request'

Mit der Option 'request' können Sie die JVM auffordern, den Status vor dem Starten des Speicherauszugsagenten vorzubereiten. Bei IBM WebSphere Real Time for RT Linux gibt es für 'request' die zusätzliche Option **multiple**.

Die verfügbaren Optionen sind in der folgenden Tabelle aufgelistet:

Optionswert	Beschreibung
exclusive	Fordert exklusiven Zugriff auf die JVM an.
compact	Führt eine Garbage-Collection aus. Diese Option entfernt alle nicht erreichbaren Objekte aus dem Heapspeicher, bevor der Speicherauszug generiert wird.
prepwalk	Bereitet den Heapspeicher für Walk-Operationen vor. Bei Verwendung dieser Option muss auch **exclusive** angegeben werden.
serial	Andere Speicherauszüge zurückstellen, bis dieser fertig gestellt ist.

Optionswert	Beschreibung
multiple	Erzeugt separate Heapspeicherauszüge für jeden RTSJ-Hauptspeicherbereich.
preempt	Wird auf den Java-Speicherauszugsagenten angewendet und steuert, ob native Threads im Prozess zurückgestellt werden, um Stack-Traces zu erfassen. Wenn diese Option nicht angegeben wird, werden nur Java-Stack-Traces im Java-Speicherauszug (Javadump) erfasst.

Beispielsweise ist die Standardeinstellung der Option 'request' für Java-Speicherauszüge `request=exclusive+preempt`. Mithilfe der folgenden Option können Sie die Einstellungen so ändern, dass Java-Speicherauszüge erzeugt werden, ohne Threads für die Erfassung von nativen Stack-Traces zurückzustellen:

```
-Xdump:java:request=exclusive
```

In der Regel reichen die 'request'-Standardoptionen aus.

Mithilfe von + können mehrere 'request'-Optionen angegeben werden. Beispiel:

```
-Xdump:heap:request=exclusive+compact+prepwalk
```

Java-Speicherauszug verwenden

Bei einem Java-Speicherauszug werden Dateien erstellt, die Diagnoseinformationen zur JVM und zu einer Java-Anwendung enthalten, die zu einem Zeitpunkt während der Ausführung erfasst wurden. Dies können beispielsweise Informationen über das Betriebssystem, die Anwendungsumgebung, Threads, Stacks, Sperren und den Speicher sein.

Das Benutzerhandbuch für IBM SDK for Java 7 enthält nützliche Informationen zu Java-Speicherauszügen:

- Java-Speicherauszug aktivieren
- Java-Speicherauszüge auslösen
- Java-Speicherauszug interpretieren
- Umgebungsvariablen und Java-Speicherauszug

Sie finden diese Informationen an folgender Stelle: IBM SDK for Java 7 - Java-Speicherauszug verwenden.

Ergänzende Informationen und eine Musterausgabe für IBM WebSphere Real Time for RT Linux finden Sie in den folgenden Themen.

Speichermanagement (MEMINFO)

Der Abschnitt `MEMINFO` enthält Informationen zum Memory Manager (Speichermanager), einschließlich Speicherbereiche für den Heapspeicher sowie Speicher für Objekte mit unbeschränkter und beschränkter Lebensdauer.

Der Abschnitt `MEMINFO` eines Java-Speicherauszugs enthält Informationen zum Memory Manager (Speichermanager). Informationen zur Funktionsweise der Speichermanagerkomponente finden Sie in Using the Metronome Garbage Collector.

Dieser Teil des Java-Speicherauszugs enthält verschiedene Werte zum Speichermanagement:

- Freie Speicherkapazität
- Belegte Speicherkapazität

- Aktuelle Größe des Heapspeichers
- Aktuelle Größe der Speicherbereiche für Objekte mit unbeschränkter Lebensdauer
- Aktuelle Größe der Speicherbereiche für Objekte mit beschränkter Lebensdauer

Darüber hinaus enthält dieser Abschnitt Protokolldaten zur Garbage-Collection. Die Daten werden in Form von Tracepunkten angezeigt, die mit einer Zeitmarke versehen sind. Dabei steht der aktuellste Tracepunkt an erster Stelle.

Java-Speicherauszüge, die mit der Standard-JVM erstellt wurden, enthalten einen Abschnitt mit der Bezeichnung „GC History" (Protokolldaten zur Garbage-Collection). Diese Informationen sind nicht in Java-Speicherauszügen enthalten, die mit der Echtzeit-JVM erstellt wurden. Informationen zum Verhalten der Garbage-Collection erhalten Sie durch Angabe der Option **-verbose:gc** oder über den JVMn-Snap-Trace. Weitere Details finden Sie in „Informationen von 'verbose:gc' verwenden" auf Seite 146 und im Abschnitt zu Speicherauszugsagenten des Benutzerhandbuchs für IBM SDK for Java 7.

Wenn bei der Ausführung eines Programms, das Speicher für Objekte mit begrenzter Lebensdauer (Scoped Memory) verwendet, eine Ausnahmebedingung OutOfMemoryError ausgelöst wird, sind einige der im Java-Speicherauszug aufgelisteten Hauptspeicherbereiche möglicherweise leer. Wenn ein Bereich, der in einen anderen Bereich eingebettet ist, über keinen freien Speicher mehr verfügt, dann wurde dieser eingebettete Bereich zu dem Zeitpunkt, an dem der Java-Speicherauszug erstellt wurde, möglicherweise bereits gelöscht. Informationen zum Status der Hauptspeicherbereiche zu dem Zeitpunkt, an dem die Ausnahmebedingung OutOfMemoryError ausgelöst wird, erhalten Sie, indem Sie das Programm mit der folgenden Befehlszeilenoption ausführen:

```
-Xdump:java:events=throw,filter=java/lang/OutOfMemoryError,range=1..1
```

Dieser Befehl generiert bereits beim Auslösen der Ausnahmebedingung OutOfMemoryError einen zusätzlichen Java-Speicherauszug, nicht erst, wenn die nicht abgefangene Ausnahmebedingung zu einem etwas späteren Zeitpunkt erkannt wird. In diesem Java-Speicherauszug werden alle Speicherbereiche (einschließlich der eingebetteten Bereiche) aufgeführt, die beim Auslösen der Ausnahmebedingung OutOfMemoryError aktiv waren. Weitere Informationen zur Verwendung der Option **-Xdump** finden Sie im Benutzerhandbuch für IBM SDK for Java 7.

In einem Java-Speicherauszug sind Segmente jeweils Blöcke von Speicher, denen die Java-Laufzeitumgebung Tasks mit großem Speicherbedarf zuordnet. Beispieltasks:

- JIT-Caches verwalten
- Java-Klassen speichern

Java Runtime ordnet auch anderen nativen Speicher zu, der im Abschnitt `MEMINFO` nicht aufgelistet ist. Der von den Segmenten der Java-Laufzeitumgebung verwendete Gesamtspeicher entspricht nicht zwangsläufig dem vollständigen Speicherbedarf der Java-Laufzeitumgebung. Ein Java Runtime-Segment besteht aus der Segmentdatenstruktur und einem zugeordneten Block von nativem Speicher.

Bei dem folgenden Beispiel handelt es sich um eine typische Ausgabe. Alle Werte werden als Hexadezimalwerte bereitgestellt. Die Spaltenüberschriften im Abschnitt `MEMINFO` haben die folgenden Bedeutungen:

```
| 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
|        (Zeilen zur Verdeutlichung entfernt)
| 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
|        (Zeilen zur Verdeutlichung entfernt)
| 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
|         (Zeilen zur Verdeutlichung entfernt)
| NULL
| NULL           ------------------------------------------------------------------------
```

Threads und Stack-Trace (THREADS)

Der Abschnitt THREADS ist für Anwendungsprogrammierer einer der hilfreichsten Abschnitte in einem Java-Speicherauszug. Er enthält eine Liste der Java-Threads, der nativen Threads und der Stack-Traces. Außerdem werden Echtzeitthreads und No-Heap-Echtzeitthreads von IBM WebSphere Real Time for RT Linux angezeigt.

Java-Threads werden durch native Threads des Betriebssystems implementiert. Jeder Thread wird durch eine Gruppe von Zeilen wie die folgenden dargestellt:

```
"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)
```

Die Namen der Java-Threads können über den Befehl **ps** im Betriebssystem angezeigt werden. Weitere Informationen zur Verwendung des Befehls **ps** finden Sie in „Allgemeine Debugging-Verfahren” auf Seite 106.

In einem Java-Speicherauszug, der mit einem No-Heap-Echtzeitthread erstellt wird, fehlen unter Umständen einige Informationen. Ist das Threadnamensobjekt aus dem No-Heap-Echtzeitthread nicht ersichtlich, wird anstelle des tatsächlichen Threadnamens der Text „(access error)” (Zugriffsfehler) angezeigt.

Bei den Eigenschaften in der ersten Zeile handelt es sich um den Threadnamen, die Adressen der JVM-Threadstrukturen und das Java-Threadobjekt, den Threadstatus und die Priorität des Java-Threads. Bei den Eigenschaften in der zweiten Zeile handelt es sich um die ID und die Priorität des nativen Betriebssystemthreads sowie um die native Planungsrichtlinie des Betriebssystems.

Es gibt drei Möglichkeiten, die Threadnamen zu finden:

- In Javacore-Dateien. Allerdings sind nicht alle Threads in den Javacore-Dateien enthalten.
- Beim Auflisten der Threads über das Betriebssystem mit dem Befehl **ps**.
- Bei Verwendung der Methode java.lang.Thread.getName().

In der folgenden Tabelle werden Informationen zu Threadnamen in IBM WebSphere Real Time for RT Linux aufgelistet.

Tabelle 12. Threadnamen in IBM WebSphere Real Time for RT Linux

Informationen zum Thread	**Threadname**
Ein interner JVM-Thread, mit dem das Garbage-Collection-Modul die Endbearbeitung von Objekten durch sekundäre Threads zuteilt.	`Finalizer master`
Der vom Garbage-Collector verwendete Alarmthread.	`GC Alarm`
Die für die Garbage-Collection verwendeten Slave-Threads.	`GC Slave`
Ein interner JVM-Thread, mit dem das JIT-Compilermodul (Just-in-time) Stichproben zur Nutzung von Methoden in der Anwendung nimmt.	`JIT Sampler`
Ein Thread, mit dem die virtuelle Maschine die von der Anwendung empfangenen Signale verwaltet; dabei spielt es keine Rolle, ob die Signale extern oder intern generiert wurden.	`Signal Reporter`

Die Standardnamen der in Java-Code erstellten Echtzeitthreads (javax.realtime-.RealtimeThread) lauten `RTThread-x`, wobei „x" die Threadnummer ist.

Die Standardnamen von No-Heap-Echtzeitthreads lauten `NHRTThread-x`, wobei „x" die Threadnummer ist.

Die Priorität der Java-Threads wird plattformabhängig einem Prioritätswert des Betriebssystems zugeordnet. Eine Java-Threadpriorität mit einem großen Wert besagt, dass es sich um einen Thread mit hoher Priorität handelt. Dieser Thread wird häufiger ausgeführt als Threads mit niedrigerer Priorität. Weitere Informationen zu Prioritäten und Java-, Echtzeit- und No-Heap-Echtzeitthreads finden Sie in „Prioritätszuordnung und -übernahme" auf Seite 12.

Folgende Statuswerte sind möglich:

- R (Runnable): Der Thread kann bei Bedarf ausgeführt werden.
- CW (Condition Wait): Der Thread befindet sich im Wartestatus, beispielsweise aus den folgenden Gründen:
  - Ein sleep()-Aufruf wurde ausgegeben.
  - Der Thread wurde für eine Ein-/Ausgabe geblockt.
  - Eine wait()-Methode wurde aufgerufen, um auf die Benachrichtigung eines Monitors zu warten.
  - Der Thread wird über einen join()-Aufruf mit einem anderen Thread synchronisiert.

- S (Suspended): Der Thread wurde von einem anderen Thread ausgesetzt.
- Z (Zombie): Der Thread wurde abgebrochen.
- P (Parked): Der Thread wurde von der neuen Concurrency-API von Java (java.util.concurrent) vorgehalten bzw. "geparkt".
- B (Blocked): Der Thread wartet auf die Übernahme einer Sperre, die gerade anderweitig vergeben ist.

Ist ein Thread geparkt oder blockiert, enthält die Ausgabe eine mit 3XMTHREADBLOCK beginnende Zeile für diesen Thread, in der die Ressource, auf die der Thread wartet und (sofern möglich) der Thread aufgelistet ist, der zurzeit Eigner dieser Ressource ist. Weitere Informationen finden Sie im Thema zu geblockten Threads im Benutzerhandbuch für IBM SDK for Java 7.

Wenn Sie einen Java-Speicherauszug zum Abrufen von Diagnoseinformationen initialisieren, versetzt die JVM Java-Threads vor der Erzeugung des Java-Kernspeichers in den Wartemodus. In der Zeile 1TIPREPSTATE des Abschnitts TITLE wird der Vorbereitungsstatus exclusive_vm_access angezeigt.

```
1TIPREPSTATE Prep State: 0x4 (exclusive_vm_access)
```

Threads, die beim Auslösen des Java-Kernspeichers Java-Code ausführten, befinden sich im Status CW (Condition Wait).

```
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)
```

Der Abschnitt LOCKS des Java-Kernspeichers zeigt, dass diese Threads auf eine interne JVM-Sperre warten.

```
2LKREGMON          Thread public flags mutex lock (0x002A5234): <unowned>
3LKNOTIFYQ            Waiting to be notified:
3LKWAITNOTIFY            "main" (0x41481900)
```

Heapspeicherauszug verwenden

Der Begriff Heapspeicherauszug beschreibt den IBM Virtual Machine for Java-Mechanismus, bei dem ein Auszug von allen Liveobjekten, die sich im Java-Heapspeicher befinden, erstellt wird, d. h., von den Objekten, die gerade von der aktiven Java-Anwendung verwendet werden.

Das Benutzerhandbuch für IBM SDK for Java 7 enthält nützliche Informationen zu Heapspeicherauszügen:
- Heapspeicherauszüge abrufen
- Tools für die Verarbeitung von Heapspeicherauszügen
- **-Xverbose:gc** zum Abrufen von Heapspeicherinformationen verwenden
- Umgebungsvariablen und Heapspeicherauszug
- Textformat (klassisches Format) der Heapspeicherauszugsdatei
- PHD-Dateiformat

Sie finden diese Informationen an folgender Stelle: IBM SDK for Java 7 - Heapspeicherauszug verwenden.

Ergänzende Informationen für IBM WebSphere Real Time for RT Linux:

Mehrere Heapspeicherauszüge für echtzeitorientierte JVMs aktivieren

Der generierte Heapspeicherauszug ist standardmäßig eine einzelne Datei, die Informationen zu allen Java-Objekten in allen Hauptspeicherbereichen, im Heapspeicher, im Speicher für Objekte mit unbeschränkter Lebensdauer und im Speicher für Objekte mit beschränkter Lebensdauer enthält. Mehrere Speicherauszüge werden in erster Linie erzeugt, damit jeder einzelne Heapspeicherbereich mithilfe der traditionellen Heapspeicherauszugstools ohne Änderung analysiert werden kann.

Informationen zu diesem Vorgang

Standardmäßig enthalten Heapspeicherauszüge Informationen zu allen Objekten in den Hauptspeicherbereichen der JVM (Heapspeicher, Immortal Memory und Speicher für Objekte mit beschränkter Lebensdauer). Sie können separate Heapspeicherauszüge mit Informationen zu Java-Objekten in jedem Hauptspeicherbereich über die Option **request=multiple** mit **-Xdump:heap** anfordern. Beachten Sie, dass Sie die Standardeinstellungen der Option 'request' auch wiederholen müssen. Sie müssen also `request=multiple+exclusive+prepwalk+compact` angeben. Durch Folgendes wird eine Gruppe von Heapspeicherauszügen mit einem zusätzlichen Feld im Namen erzeugt, das den Hauptspeicherbereich angibt:

```
heapdump.%id.%Y%m%d.%H%M%S.%pid.phd
```

Dabei gibt *%id* die Heapspeicherauszugsdatei mit Objekten im Heapspeicher, im Speicher für Objekte mit unbeschränkter Lebensdauer oder in einem bestimmten Bereich im Speicher für Objekte mit beschränkter Lebensdauer an.

Es gibt 4 Typen von Heapspeicher, die durch die folgenden Namen dargestellt sind: „Default", „Immortal", „Scope" und „Other". Der Heapspeicherauszugscode ersetzt %id im Heapspeicherkennsatz durch einen dieser Namen, der mit einer (gewöhnlich numerischen) ID verknüpft ist, z. B.:
`heapdump.Immortal12994208.20060807.093653.7684.txt.`

Beispiel

```
java -Xrealtime -Xdump:heap:defaults:request=multiple+exclusive+compact+prepwalk
<Java-Programm>
```

Die Verwendung dieser Zusatzoption erzeugt mehrere Heapspeicherauszüge im PHD-Format (PHD - Portable Heapdump).

```
java -Xrealtime -Xdump:heap:defaults:request=multiple+exclusive+compact+prepwalk,
opts=CLASSIC <Java-Programm>
```

Die Verwendung dieser Zusatzoption erzeugt mehrere Heapspeicherauszüge im klassischen Textformat.

Die Verwendung der Option '-Xdump:what' zeit die Speicherauszugsagenten beim JVM-Start an und eignet sich für die Prüfung der vorhandenen Speicherauszugsoptionen.

Textformat (klassisches Format) der Heapspeicherauszugsdatei

Der Heapspeicherauszug im Textformat oder klassischen Format ist eine Liste aller Objektinstanzen im Heapspeicher (einschließlich Objekttyp, Größe und Verweise zwischen Objekten).

Kopfsatz

Der Kopfsatz ist ein einzelner Datensatz, der eine Zeichenfolge aus Versionsinformationen enthält.

```
// Version: <Versionszeichenfolge mit SDK-Version, Plattform und
JVM-Buildstufe>
```

Beispiel:

```
// Version: J2RE 7.0 IBM J9 2.6 Linux x86-32 build 20101016_024574_lHdRSr
```

Objektdatensätze

Objektdatensätze bestehen aus mehreren Datensätzen (einer für jede Objektinstanz im Heapspeicher), die Informationen wie Objektadresse, Größe, Typ und Verweise aus dem Objekt bereitstellen.

```
<Objektadresse, hexadezimal> [<Länge der Objektinstanz in Bytes, dezimal>]
OBJ <Objekttyp> <Klassenblockverweise, hexadezimal>
<Heapspeicherverweis, hexadezimal <Heapspeicherverweis,
hexadezimal> ...
```

Die Objektadresse und Heapspeicherverweise befinden sich im Heapspeicher, die Klassenblockadresse jedoch außerhalb des Heapspeichers. Alle in der Objektinstanz gefundenen Verweise werden aufgelistet, einschließlich der Verweise mit Nullwerten. Der Objekttyp ist entweder ein Klassenname (einschließlich Paket) oder ein primitives Array oder Klassen-Array, dessen JVM-Standardtypkennung angezeigt wird (siehe „Java VM-Typkennungen" auf Seite 136). Objektdatensätze können außerdem zusätzliche Klassenblockverweise enthalten, zum Beispiel bei Reflexionsklasseninstanzen.

Beispiele:

Objektinstanz mit einer Länge von 28 Bytes und dem Typ 'java/lang/String':

```
0x00436E90 [28] OBJ java/lang/String
```

Klassenblockadresse des Typs 'java/lang/String', gefolgt von einem Verweis auf eine Array-Instanz des Typs 'char':

```
0x415319D8 0x00436EB0
```

Objektinstanz mit einer Länge von 44 Bytes und dem Array-Typ 'char':

```
0x00436EB0 [44] OBJ [C
```

Klassenblockadresse von char-Array:

```
0x41530F20
```

Objekt des Typs Array der untergeordneten Klasse 'java/util/Hashtable Entry':

```
0x004380C0 [108] OBJ [Ljava/util/Hashtable$Entry;
```

Objekt des Typs untergeordnete Klasse 'java/util/Hashtable Entry':

```
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
```

Klassenblockadresse und Heapspeicherverweise, einschließlich Verweise mit Nullwerten:

```
0x4158CB88 0x004219B8 0x004341F0 0x00000000
```

Klassendatensätze

Klassendatensätze bestehen aus mehreren Datensätzen (einer für jede geladene Klasse), die Informationen wie Klassenblockadresse, Größe, Typ und Verweise aus der Klasse bereitstellen.

```
<Klassenblockadresse,
hexadezimal> [<Länge des Klassenblocks in Bytes, dezimal>]
CLS <Klassentyp>
<Klassenblockverweis, hexadezimal> <Klassenblockverweis, hexadezimal> ...
<Heapspeicherverweis, hexadezimal> <Heapspeicherverweis, hexadezimal>...
```

Die Klassenblockadresse und Klassenblockverweise befinden sich außerhalb des Heapspeichers, aber der Klassendatensatz kann auch Verweise auf den Heapspeicher enthalten, zum Beispiel bei statischen Klassendatenelementen. Alle im Klassenblock gefundenen Verweise werden aufgelistet, einschließlich der Verweise mit Nullwerten. Der Klassentyp ist entweder ein Klassenname (einschließlich Paket) oder ein primitives Array oder Klassen-Array, dessen JVM-Standardtypkennung angezeigt wird (siehe „Java VM-Typkennungen” auf Seite 136).

Beispiele:

Klassenblock mit einer Länge von 32 Byte, für Klasse 'java/lang/Runnable':

```
0x41532E68 [32] CLS java/lang/Runnable
```

Verweise auf andere Klassenblöcke und Heapspeicherverweise, einschließlich Verweise mit Nullwerten:

```
0x4152F018 0x41532E68 0x00000000 0x00000000 0x00499790
```

Klassenblock mit einer Länge von 168 Bytes, für Klasse 'java/lang/Math':

```
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
```

Trailerdatensatz 1

Trailerdatensatz 1 ist ein einzelner Datensatz, der Satzzählungen enthält.

```
// Breakdown - Classes: <Klassendatensatzzählung, dezimal>,
Objects: <Objektdatensatzzählung, dezimal>,
ObjectArrays: <Objekt-Array-Datensatzzählung, dezimal>,
PrimitiveArrays: <Primitiv-Array-Datensatzzählung, dezimal>
```

Beispiel:

```
// Breakdown - Classes: 321, Objects: 3718, ObjectArrays: 169,
PrimitiveArrays: 2141
```

Trailerdatensatz 2

Trailerdatensatz 2 ist ein einzelner Datensatz, der Gesamtzahlen enthält.

```
// EOF:  Total 'Objects',Refs(null) :
<Gesamtzahl Objektzählung, dezimal>,
<Gesamtzahl Verweiszählung, dezimal>
(,Gesamtzahl Nullverweiszählung, dezimal>)
```

Beispiel:

```
// EOF:  Total 'Objects',Refs(null) : 6349,23240(7282)
```

Java VM-Typkennungen

Die Java VM-Typkennungen sind Abkürzungen für die Java-Typen. Sie werden in der folgenden Tabelle aufgeführt:

Java VM-Typkennung	**Java-Typ**
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double
L *<vollständig qualifizierte Klasse>* ;	*<vollständig qualifizierte Klasse>*
[*<Typ>*	*<Typ>*[ ] (Array von *<Typ>*)
(*<arg-Typen>*) *<ret-Typ>*	method

Systemspeicherauszüge und die Anzeigefunktion für Speicherauszüge verwenden

Die JVM kann unter konfigurierbaren Bedingungen native Systemspeicherauszüge, die auch als Kernspeicherauszüge bekannt sind, generieren. Systemspeicherauszüge sind in der Regel umfangreich. Die meisten Tools zum Analysieren von Systemspeicherauszügen sind außerdem plattformspezifisch.Verwenden Sie das Tool **gdb** zum Analysieren eines Systemspeicherauszugs unterLinux.

Das Benutzerhandbuch für IBM SDK for Java 7 enthält nützliche Informationen zur Verwendung von Speicherauszugsagenten und der Anzeigefunktion für Speicherauszüge:

- Übersicht über Systemspeicherauszüge
- Standardeinstellungen für Systemspeicherauszüge
- Anzeigefunktion für Speicherauszüge verwenden
  - **jextract** verwenden
  - Mit der Anzeigefunktion für Speicherauszüge zu lösende Probleme
  - In **jdmpview** verfügbare Befehle
  - Beispielsitzung
  - **jdmpview**jdmpview-Befehle - Kurzübersicht

Sie finden diese Informationen an folgender Stelle: IBM SDK for Java 7 - Speicherauszüge und Anzeigefunktion für Speicherauszüge verwenden.

Ergänzende Informationen für IBM WebSphere Real Time for RT Linux:

`jextract` verwenden

Wenn Sie einen Systemspeicherauszug von einer Echtzeit-JVM verarbeiten, müssen Sie die Option **-Xrealtime** einschließen. Beispiel:

```
jextract -Xrealtime <Kerndateiname> [<ZIP-Datei>]
```

Wenn Sie **jextract** in einer JVM ausführen, die sich von der unterscheidet, für die der Speicherauszug erzeugt wurde, werden folgende Fehlernachrichten ausgegeben:

```
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).
```

Diese Nachricht wird auch erzeugt, wenn Sie Java mit der Standard-JVM ausgeführt, jedoch die Option **-Xrealtime** bei der Verarbeitung des Speicherauszugs mit **jextract** verwendet haben.

In `jdmpview` verfügbare Befehle

jdmpview ist ein interaktives Befehlszeilentool zur Untersuchung der Informationen aus einem JVM-Systemspeicherauszug und zur Ausführung verschiedener Analysefunktionen.

info jitm
Listet mit AOT und JIT kompilierte Methoden und deren Adressen auf:
- Methodenname und Signatur
- Startadresse der Methode
- Endadresse der Methode

Alle anderen Befehlsoptionen finden Sie im Benutzerhandbuch für IBM SDK for Java 7.

Tracefunktion von Java-Anwendungen und der JVM

'JVM trace' ist eine in IBM WebSphere Real Time for RT Linux bereitgestellte Tracefunktion, die die Leistung nur unwesentlich beeinträchtigt. In den meisten Fällen werden die Tracedaten in einem kompakten Binärformat gespeichert, das mit dem mitgelieferten Java-Formatierungsprogramm formatiert werden kann.

Die Tracefunktion ist standardmäßig aktiviert, zusammen mit einer kleinen Anzahl an Tracepunkten, die auf Hauptspeicherpuffer verweisen. Sie können Tracepunkte während der Ausführung aktivieren, indem Sie Stufen, Komponenten, Gruppennamen oder einzelne Tracepunktkennungen verwenden.

Das Benutzerhandbuch für IBM SDK for Java 7 enthält ausführliche Informationen zur Erstellung von Anwendungstraces:

- Traceziele
- Tracepunkt-Arten
- Standardtrace
- Tracedaten erfassen
- Tracesteuerung

- Traceerstellung für Java-Anwendungen
- Traceerstellung für Java-Methoden

Bei der IBM WebSphere Real Time for RT Linux-Traceerstellung müssen Sie die Echtzeit-JVM ordnungsgemäß aufrufen, wenn Sie Traceoptionen einschließen. Geben Sie z. B. beim Angeben von Traceoptionen Folgendes ein:

```
java -Xrealtime -Xtrace:<Optionen>
```

Sie finden Informationen zu IBM SDK for Java 7 an folgender Stelle: Traceerstellung für Java-Anwendungen und JVM.

Fehlerbestimmung für JIT und AOT

Mithilfe von Befehlszeilenoptionen können Sie Probleme und Fehler in Zusammenhang mit dem JIT-Compiler und dem AOT-Compiler diagnostizieren und die Leistung der Compiler optimieren.

IBM WebSphere Real Time for RT Linux nutzt zwar einige allgemeine Komponenten gemeinsam mit IBM SDK for Java 7, das Verhalten von JIT und AOT ist jedoch unterschiedlich. In diesem Abschnitt wird die Fehlerbehebung für JIT- und AOT-Probleme unter IBM WebSphere Real Time for RT Linux behandelt.

JIT- oder AOT-Probleme diagnostizieren

Hin und wieder kann es vorkommen, dass gültiger Bytecode bei der Kompilierung zu ungültigem nativen Code wird, sodass das Java-Programm abstürzt. Sie können dem Java-Service-Team wertvolle Hinweise geben, mit deren Hilfe es feststellen kann, ob der Fehler beim JIT- oder AOT-Compiler liegt und wenn ja, *wo* genau die Fehlerursache liegt.

Informationen zu diesem Vorgang

Mit der Option **-Xaot:verbose** in der admincache-Befehlszeile können Sie die Methoden ermitteln, die beim Auffüllen des Cache für gemeinsam genutzte Klassen kompiliert werden. Beispiel:

```
admincache -Xrealtime -Xaot:verbose -populate -aot my.jar -cp <Ihr_Klassenpfad>
```

In diesem Abschnitt wird beschrieben, wie Sie feststellen können, ob ein Problem durch den Compiler verursacht wurde. Darüber hinaus werden mögliche Strategien zur Fehlerbehebung sowie Debugverfahren aufgeführt, mit denen Probleme in Zusammenhang mit dem Compiler behoben werden können.

JIT- oder AOT-Compiler inaktivieren:

Wenn Sie vermuten, dass ein Problem beim JIT- oder AOT-Compiler liegt, sollten Sie den Kompiliervorgang inaktivieren, um festzustellen, ob das Problem weiterhin auftritt. Ist dies der Fall, liegt das Problem nicht beim Compiler.

Informationen zu diesem Vorgang

Der JIT-Compiler ist standardmäßig aktiviert. Der AOT-Compiler ist ebenfalls aktiviert, jedoch nur aktiv, wenn die gemeinsam genutzten Klassen aktiviert wurden. Aus Gründen der Effizienz werden nicht alle Methoden in einer Java-Anwendung kompiliert. Die Java Virtual Machine (JVM) zeichnet auf, wie oft die einzelnen Methoden in der Anwendung aufgerufen werden; bei jedem Aufruf und bei jeder Interpretation einer Methode wird der Aufrufzähler der betreffenden Methode ent-

sprechend erhöht. Erreicht der Zähler den Grenzwert für die Kompilierung, wird die Methode kompiliert und nativ ausgeführt.

Durch die Aufrufzähler wird die Kompilierung von Methoden über die gesamte Lebensdauer der Anwendung verteilt, wobei Methoden, die häufiger aufgerufen werden, eine höhere Priorität erhalten. Methoden dagegen, die nur selten aufgerufen werden, werden möglicherweise nie kompiliert. Wenn ein Java-Programm daher fehlschlägt, kann das Problem zwar beim JIT- oder AOT-Compiler liegen, es kann aber auch an anderer Stelle in der JVM verursacht werden.

Bei der Fehlerdiagnose muss in einem ersten Schritt festgestellt werden, *wo* das Problem liegt. Dazu müssen Sie das Java-Programm zunächst im reinen Interpretationsmodus ausführen (d. h. mit inaktiviertem JIT- und AOT-Compiler).

Vorgehensweise

1. Entfernen Sie alle **-Xjit**- und **-Xaot**-Optionen (sowie alle zugehörigen Parameter) aus der Befehlszeile.
2. Inaktivieren Sie mit der Befehlszeilenoption **-Xint** den JIT- und den AOT-Compiler. Aus leistungstechnischen Gründen sollte die Option **-Xint** in Produktionsumgebungen nicht verwendet werden.

Nächste Schritte

Die Ausführung des Java-Programms mit inaktivierter Kompilierung kann folgendes Ergebnis haben:

- Das Problem besteht weiterhin, d. h., es wird nicht durch den JIT- oder AOT-Compiler verursacht. In einigen Fällen wird das Programm möglicherweise auf andere Weise abgebrochen, trotzdem liegt der Fehler nicht beim Compiler.
- Das Problem tritt nicht mehr auf. In diesem Fall wird es höchstwahrscheinlich durch den JIT- oder AOT-Compiler verursacht.

  Wenn Sie keine gemeinsam verwendeten Klassen verwenden, ist der JIT-Compiler fehlerhaft. Bei Verwendung gemeinsam genutzter Klassen müssen Sie feststellen, welcher Compiler den Fehler verursacht; dazu müssen Sie die Anwendung nur mit aktivierter JIT-Kompilierung ausführen. Führen Sie die Anwendung mit der Option **-Xnoaot** anstelle der Option **-Xint** aus. Das kann folgendes Ergebnis haben:
  - Das Problem besteht weiterhin, d. h., es wird durch den JIT-Compiler verursacht. Sie können auch mithilfe der Option **-Xnojit** anstelle der Option **-Xnoaot** feststellen, ob das Problem allein beim JIT-Compiler liegt.
  - Das Problem tritt nicht mehr auf, d. h., es wird durch den AOT-Compiler verursacht.

JIT-Compiler gezielt inaktivieren:

Wenn der Fehler Ihres Java-Programms auf ein Problem mit dem JIT-Compiler hinweist, können Sie versuchen, den Fehler weiter einzugrenzen.

Informationen zu diesem Vorgang

Standardmäßig optimiert der JIT-Compiler Methoden auf unterschiedlichen Optimierungsstufen. Auf verschiedene Methoden werden je nach der entsprechenden Aufrufanzahl verschiedene Optimierungsoptionen angewendet. Methoden, die häufiger aufgerufen werden, werden mit einer höheren Stufe optimiert. Indem Sie Parameter des JIT-Compilers ändern, können Sie die Optimierungsstufe der Metho-

den steuern. Sie können ermitteln, ob das Optimierungsprogramm fehlerhaft ist, und wenn ja, welche Optimierung problematisch ist.

JIT-Parameter werden in Form einer durch Kommas getrennten Liste an die Option **-Xjit** angehängt: Die Syntax lautet wie folgt:
-Xjit:<*Parameter1*>,<*Parameter2*>=<*Wert*>. Beispiel:

```
java -Xjit:verbose,optLevel=noOpt HelloWorld
```

Es wird das Programm HelloWorld ausgeführt, die ausführliche Ausgabe vom JIT-Compiler aktiviert und der native Code vom JIT-Compiler generiert, ohne dass Optimierungen ausgeführt werden.

So können Sie feststellen, wo im Compiler das Problem verursacht wird:

Vorgehensweise

1. Setzen Sie Grenzwert für die Kompilierung über den JIT-Parameter **count=0** auf null. Dieser Parameter bewirkt, dass jede Java-Methode vor ihrer Ausführung kompiliert wird. Verwenden Sie **count=0** nur zur Diagnose von Fehlern, weil viele weitere Methoden kompiliert werden, einschließlich Methoden, die nur selten verwendet werden. Die zusätzliche Kompilierung verbraucht mehr IT-Ressourcen und verlangsamt Ihre Anwendung. Mit **count=0** schlägt Ihre Anwendung sofort fehl, wenn der Problembereich erreicht wird. In einigen Fällen kann ein Fehlschlagen zuverlässiger mit **count=1** erreicht werden.
2. Fügen Sie den JIT-Compilerparametern **disableInlining** hinzu. Mit **disableInlining** wird die Generierung von umfangreicherem und komplexerem Code inaktiviert. Tritt das Problem nicht mehr auf, können Sie **disableInlining** als Fehlerumgehung einsetzen, bis das Compilerproblem vom Java-Service-Team analysiert und behoben wurde.
3. Verringern Sie die Optimierungsstufen, indem Sie den Parameter **optLevel** hinzufügen und das Programm erneut ausführen, bis der Fehler nicht mehr auftritt oder Sie die Stufe „noOpt" erreichen. Starten Sie bei einem Problem mit dem JIT-Compiler mit der Optimierungsstufe „scorching" und fahren Sie anschließend mit der nächstniedrigeren Stufe usw. fort. Hier die Optimierungsstufen in absteigender Reihenfolge:
   a. scorching
   b. veryHot
   c. hot
   d. warm
   e. cold
   f. noOpt

Nächste Schritte

Wenn mithilfe einer dieser Einstellungen der Fehler behoben wird, verfügen Sie über eine Fehlerumgehung, die Sie verwenden können. Diese Fehlerumgehung ist vorläufig einsetzbar, während das Compilerproblem vom Java-Service-Team analysiert und behoben wird. Wenn das Entfernen von **disableInlining** aus der JIT- Parameterliste nicht dazu führt, dass das Problem erneut auftritt, sollten Sie 'disableInlining' entfernen, um die Leistung zu erhöhen. Gehen Sie entsprechend den Anweisungen im Abschnitt „Methode ermitteln, bei der der Fehler auftritt" auf Seite 141 vor, um die Leistung der Fehlerumgehung zu erhöhen.

Tritt das Problem auch bei der Optimierungsstufe „noOpt" auf, müssen Sie den JIT-Compiler inaktivieren, um den Fehler zu umgehen.

Methode ermitteln, bei der der Fehler auftritt:

Wenn Sie die niedrigste Optimierungsstufe ermittelt haben, bei der der JIT- oder AOT-Compiler bei der Kompilierung von Methoden ein Problem auslöst, können Sie herausfinden, welcher Teil des Java-Programms dieses Problem bei der Kompilierung verursacht. Anschließend können Sie den Compiler anweisen, die Fehlerumgehung auf eine bestimmte Methode oder Klasse oder auf ein bestimmtes Paket zu beschränken, sodass der Compiler den Rest des Programms normal kompilieren kann. Wenn ein JIT-Compilerproblem bei **-Xjit:optLevel=noOpt** auftritt, können Sie den Compiler auch anweisen, die Methoden, die der Auslöser sind, gar nicht zu kompilieren.

Vorbereitende Schritte

Mithilfe einer Fehlernachricht wie der folgenden können Sie die Methode ermitteln, die das Problem verursacht:

```
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;
```

Folgende Zeilen sind von Bedeutung:

vmState=0x00000000
: Gibt an, dass es sich bei dem Code, der fehlerhaft ist, nicht um JVM-Laufzeitcode handelt.

Module= oder Module_base_address=
: Nicht in der Ausgabe (kann leer oder null sein), da der Code von JIT außerhalb einer DLL oder Bibliothek kompiliert wurde.

Compiled_method=
: Gibt die Java-Methode an, für die der kompilierte Code erstellt wurde.

Informationen zu diesem Vorgang

Wird die Methode, bei der der Fehler aufgetreten ist, nicht in der Ausgabe angezeigt, können Sie sie wie folgt ermitteln:

Vorgehensweise

1. Führen Sie das Java-Programm unter Angabe der JIT-Parameter **verbose** und **vlog=**<*Dateiname*> mit der Option **-Xjit** oder **-Xaot** aus. Bei Angabe dieser Parameter listet der Compiler die kompilierten Methoden in einer Protokolldatei mit dem Namen *<Dateiname>.<Datum>.<Uhrzeit>.<PID>* (auch als *Begrenzungsdatei* (LimitFile) bezeichnet) auf. Eine Begrenzungsdatei enthält typischerweise Zeilen, die kompilierten Methoden entsprechen; Beispiel:

   ```
   + (hot) java/lang/Math.max(II)I @ 0x10C11DA4-0x10C11DDD
   ```

   Zeilen, die nicht mit einem Pluszeichen beginnen, werden in den anschließenden Schritten vom Compiler ignoriert und können aus der Datei entfernt werden. Methoden, für die AOT-Code aus dem Cache für gemeinsam genutzte Klassen geladen wird, beginnen mit `+(AOT load)`.

2. Führen Sie das Programm erneut mit dem JIT- oder AOT-Parameter **limitFile=**(<*Dateiname*>,<*m*>,<*n*>) aus; dabei ist <*Dateiname*> der Pfad zur Begrenzungsdatei und <*m*> und <*n*> sind die Zeilennummern, die die erste und die letzte Methode in der Begrenzungsdatei angeben, die kompiliert werden sollen. Der Compiler kompiliert nur die in der Begrenzungsdatei zwischen den Zeilen <*m*> und <*n*> aufgelisteten Methoden. Methoden, die nicht in der Begrenzungsdatei aufgeführt sind und nicht innerhalb des angegebenen Bereichs liegen, werden nicht kompiliert und es wird auch kein AOT-Code im Cache für gemeinsam genutzte Daten für diese Methoden geladen. Schlägt das Programm nicht mehr fehl, müssen eine oder mehrere der in der letzten Iteration entfernten Methoden das Problem verursacht haben.
3. Wiederholen Sie diesen Vorgang unter Angabe verschiedener Werte für <*m*> und <*n*> so oft, bis Sie die Mindestanzahl an Methoden ermittelt haben, bei deren Kompilierung das Problem auftritt. Sie können eine Binärsuche nach der Methode durchführen, die das Problem verursacht, indem Sie die Anzahl der ausgewählten Zeilen jedesmal halbieren. Häufig kann die Datei auf eine einzige Zeile reduziert werden.

Nächste Schritte

Wenn Sie die betreffende Methode ermittelt haben, können Sie den JIT- oder AOT-Compiler gezielt nur für diese Methode inaktivieren. Schlägt das Programm bei einer JIT-Kompilierung mit Optimierungsstufe **optLevel=hot** beispielsweise aufgrund der Methode java/lang/Math.max(II)I fehl, führen Sie das Programm wie folgt aus:

```
-Xjit:{java/lang/Math.max(II)I}(optLevel=warm,count=0)
```

Damit wird nur die Methode, die das Problem verursacht, unter Verwendung der Optimierungsstufe „warm" kompiliert; alle anderen Methoden dagegen werden ganz normal kompiliert.

Schlägt eine Methode bei einer JIT-Kompilierung mit Optimierungsstufe „noOpt" fehl, können Sie sie unter Angabe des Parameters **exclude**={<*Methode*>} ganz von der Kompilierung ausschließen:

```
-Xjit:exclude={java/lang/Math.max(II)I}
```

Verursacht eine Methode einen Programmfehler, wenn AOT-Code aus dem Cache für gemeinsam genutzte Daten geladen wird, können Sie die Methode mit dem Parameter **exclude**={<*Methode*>} von dem AOT-Ladevorgang ausschließen:

```
-Xaot:exclude={java/lang/Math.max(II)I}
```

AOT-Methoden werden nur zu dem Zeitpunkt in den Cache für gemeinsam genutzte Klassen kompiliert, zu dem der Cache durch **admincache** gefüllt wird. Daher ist ein Verhindern des AOT-Ladevorgangs die beste Möglichkeit, Probleme in Zusammenhang mit diesen Methoden zu diagnostizieren.

JIT- und AOT-Kompilierfehler identifizieren:

Untersuchen Sie bei JIT-Compilerfehlern zunächst die Fehlernachricht, um festzustellen, ob ein Fehler bei dem Versuch des JIT-Compilers auftritt, eine Methode zu kompilieren.

Wenn die Java Virtual Machine (JVM) abstürzt und Sie erkennen können, dass der Fehler in der JIT-Bibliothek (`libj9jit26.so`) aufgetreten ist, ist der JIT-Compiler möglicherweise bei dem Versuch ausgefallen, eine Methode zu kompilieren.

Mithilfe einer Fehlernachricht wie der folgenden können Sie die betreffende Methode ermitteln:

```
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;)
```

Folgende Zeilen sind von Bedeutung:

vmState=0x00050000
Gibt an, dass der JIT-Compiler Code kompiliert. Eine Liste der vmState-Codenummern finden Sie in der Tabelle der Java-Speicherauszugstags im Benutzerhandbuch für IBM SDK for Java 7, http://publib.boulder.ibm.com/infocenter/java7sdk/v7r0/topic/com.ibm.java.lnx.70.doc/diag/tools/javadump_tags_info.html.

Module=/home/test/sdk/jre/bin/libj9jit26.so
Gibt an, dass der Fehler in `libj9jit26.so`, dem JIT-Compilermodul, aufgetreten ist.

Method_being_compiled=
Gibt die Java-Methode an, die kompiliert wird.

Wird die Methode, bei der der Fehler aufgetreten ist, nicht angezeigt, geben Sie die Option **verbose** mit den folgenden zusätzlichen Einstellungen an:

```
-Xjit:verbose={compileStart|compileEnd}
```

Mit diesen **verbose**-Einstellungen wird dokumentiert, wann der JIT- oder AOT-Compiler mit dem Kompilieren einer Methode beginnt und wann er den Vorgang beendet. Schlägt der JIT- oder AOT-Compiler bei einer bestimmten Methode fehl (d. h., der Kompiliervorgang wird zwar gestartet, aber vorzeitig beendet), können Sie diese Methode über die Angabe des Parameters **exclude** von der JIT- oder AOT-Kompilierung ausschließen (Informationen hierzu finden Sie in „Methode ermitteln, bei der der Fehler auftritt" auf Seite 141). Bei Problemen in Zusammenhang mit der AOT-Kompilierung müssen Sie vor einer Verwendung der Option **exclude** den Cache mit den gemeinsam genutzten Klassen löschen. Kann durch den Ausschluss der Methode ein vorzeitiger Absturz vermieden werden, können Sie mit dieser Fehlerumgehung arbeiten, bis das Problem vom Service-Team behoben wurde.

AOT-Kompilierungsfehler im Nicht-Echtzeitmodus identifizieren:

Die Fehlerbestimmung für den AOT-Compiler im Nicht-Echtzeitmodus verläuft sehr ähnlich wie für den JIT-Compiler.

Informationen zu diesem Vorgang

Führen Sie zunächst wie bei JIT Ihre Anwendung mit **-Xnoaot** aus. Dies stellt sicher, dass der AOT'ed-Code bei der Ausführung der Anwendung nicht verwendet wird.

Wird das Problem dadurch behoben, müssen Sie die AOT-JAR-Dateien anhand des im Abschnitt „Methode ermitteln, bei der der Fehler auftritt" auf Seite 141 beschriebenen Verfahrens erneut erstellen, wobei die Option **-Xaot** zur AOT-Erstellungszeit anstatt während der Anwendungsausführung angegeben wird.

AOT-Kompilierungsfehler im Echtzeitmodus ermitteln:

Die AOT-Fehlerbestimmung sucht mithilfe des Tools 'admincache' nach dem Fehler.

Informationen zu diesem Vorgang

Im Gegensatz zu JIT-Kompilierungsfehlern, die während der Anwendungsausführung auftreten, treten AOT-Kompilierungsfehler während des Füllschritts von 'admincache' auf.

Führen Sie das Tool 'admincache' mit der Option **-Xnoaot** aus, um zu ermitteln, wo der Fehler aufgetreten ist. Hierdurch wird sichergestellt, dass die Anwendung nicht mit Code ausgeführt wird, der mit AOT kompiliert wurde.

Wenn die Verwendung der Option **-Xnoaot** den Fehler korrigiert, prüfen Sie die Ausgabe des ursprünglichen Absturzes. Die Ausgabe enthält Informationen, die angeben, welche Methode die Fehlerursache ist. Suchen Sie nach einer Zeile, die der folgenden Zeile ähnelt:

```
Method_being_compiled=myAppClass.main(Ljava/lang/String;)V
```

Diese Methode muss aus der AOT-Kompilierung ausgeschlossen werden, um den Fehler zu vermeiden. Fügen Sie hierzu der admincache-Befehlszeile wie im Folgenden gezeigt eine Option hinzu:

```
-Xaot:exclude={myAppClass.main(Ljava/lang/String;)V}
```

Dieser Ausschluss verhindert die AOT-Kompilierung der fehlerhaften Methode.

Leistung von Anwendungen mit kurzer Laufzeit

Der JIT-Compiler von IBM ist für Anwendungen mit langer Laufzeit optimiert, wie sie in der Regel auf einem Server eingesetzt werden. Mit der **-Xquickstart**-Befehlszeilenoption im Nicht-Echtzeitmodus können Sie die Leistung von Anwendungen mit kurzer Laufzeit erhöhen; dies gilt besonders für Anwendungen, bei denen die Verarbeitung nicht auf eine geringe Anzahl von Methoden konzentriert ist.

Bei Verwendung der Option **-Xquickstart** verwendet der JIT-Compiler standardmäßig eine niedrigere Optimierungsstufe und kompiliert weniger Methoden. Werden weniger Methoden schneller kompiliert, kann dies den Anwendungsstart beschleunigen. Ist der AOT-Compiler aktiv (gemeinsam genutzte Klassen und AOT-Kompilierung sind aktiviert), werden bei Angabe der Option **-Xquickstart** alle für eine Kompilierung ausgewählten Methoden mit dem AOT-Compiler kompiliert, wodurch der Start nachfolgender Ausführungen verkürzt wird. **-Xquickstart** kann bei der Verwendung für Anwendungen mit langer Laufzeit, die Methoden mit vielen Verarbeitungsressourcen enthalten, die Leistung vermindern. Änderungen an der Implementierung der Option **-Xquickstart** in künftigen Releases sind vorbehalten.

Sie können auch versuchen, die Startzeiten durch Anpassung des JIT-Schwellenwerts zu verbessern. Weitere Informationen finden Sie in „JIT-Compiler gezielt inaktivieren" auf Seite 139.

-Xquickstart hat keine Auswirkungen auf die Verwendung von AOT-Code mit **-Xrealtime**.

Verhalten der Java Virtual Machine bei Inaktivität

Sie können die CPU-Zyklen einer inaktiven Java Virtual Machine (JVM) über die Option **-XsamplingExpirationTime** reduzieren, mit der der JIT-Sampling-Thread inaktiviert wird.

Über den JIT-Sampling-Thread wird die aktive Java-Anwendung auf häufig verwendete Methoden überprüft. Die Hauptspeicherbelegung und die Prozessorauslastung des Sampling-Threads sind geringfügig und die durchgeführten Stichproben werden automatisch reduziert, wenn die JVM inaktiv ist.

In einigen Fällen ist es aber vielleicht wünschenswert, dass keine CPU-Zyklen durch eine inaktive JVM verbraucht werden. Dazu müssen Sie die Option **-XsamplingExpirationTime**<*Zeitraum*> angeben. Dabei wird <*Zeitraum*> auf die Anzahl an Sekunden gesetzt, die der Sampling-Thread ausgeführt werden soll. Allerdings sollte diese Option mit Vorsicht gehandhabt werden, da ein inaktivierter Sampling-Thread nicht wieder aktiviert werden kann. Der Sampling-Thread sollte zumindest lange genug aktiv sein, um wichtige Optimierungen zu erkennen.

Diagnostics-Collector

Der Diagnostics-Collector erfasst die Java-Diagnosedateien zu einem Problemereignis.

Das Zusammenstellen der vom IBM Service benötigten Dateien kann die Zeit reduzieren, die für das Lösen der gemeldeten Probleme aufgewendet wird. Das Benutzerhandbuch für IBM SDK for Java 7 enthält ausführliche Informationen zur Diagnostics-Collector-Verwendung.

Sie finden diese Informationen an folgender Stelle: IBM SDK for Java 7 - Diagnostics-Collector.

Garbage-Collector-Diagnose

In diesem Abschnitt wird die Vorgehensweise zur Diagnose von Problemen bei der Garbage-Collection beschrieben.

Das Benutzerhandbuch für IBM SDK for Java 7 enthält nützliche Informationen zum Diagnostizieren von Garbage-Collector-Problemen:

- Ausführliche Protokollierung bei Garbage-Collection
- Garbage-Collection-Tracing mit **-Xtgc**

Sie finden diese Informationen an folgender Stelle: IBM SDK for Java 7 - Garbage-Collector-Diagnose.

Ergänzende Informationen zum Metronom-Garbage-Collector von IBM WebSphere Real Time for RT Linux finden Sie in den folgenden Abschnitten.

Fehlerbehebung für den Metronom-Garbage-Collector

Mit den Befehlszeilenoptionen können Sie die Häufigkeit der Metronom-Garbage-Collection, Ausnahmebedingungen aufgrund abnormaler Speicherbedingungen und das Metronomverhalten bei expliziten Systemaufrufen steuern.

Informationen von 'verbose:gc' verwenden:

Sie können die Option **-verbose:gc** mit der Option **-Xgc:verboseGCCycleTime=N** verwenden, um Informationen zur Metronom-Garbage-Collector-Aktivität auf der Konsole auszugeben. Nicht alle XML-Eigenschaften in der **-verbose:gc**-Ausgabe von der Standard-JVM werden erstellt oder gelten für die Ausgabe des Metronom-Garbage-Collectors.

Mit der Option **-verbose:gc** können Sie den minimalen, maximalen und durchschnittlichen freien Speicherplatz im Heapspeicher anzeigen. Dadurch können Sie den Grad der Aktivität sowie die Verwendung des Heapspeichers prüfen und anschließend die Werte bei Bedarf anpassen. Die Option **-verbose:gc** gibt Metronom-statistikdaten auf der Konsole aus.

Die Option **-Xgc:verboseGCCycleTime=N** steuert die Informationsabrufhäufigkeit. Sie ermittelt die Zeit in Millisekunden, die bis zum Erstellen eines Speicherauszugs der Zusammenfassungen verstreicht. Der Standardwert für N ist 1000 Millisekunden. Die Zykluszeit bedeutet nicht, dass ein Speicherauszug der Zusammenfassung genau zu dieser Zeit erstellt wird, sondern wenn das letzte Garbage-Collection-Ereignis stattfindet, das dieses Zeitkriterium erfüllt. Die Erfassung und Anzeige dieser Statistikdaten kann die Pausezeiten des Metronom-Garbage-Collectors erhöhen. Je kleiner N wird, desto größer können die Pausezeiten werden.

Ein Quantum ist ein einzelner Zeitraum von Metronom-Garbage-Collector-Aktivität, der für eine Anwendung eine Unterbrechung, d. h. Pausezeit, verursacht.

Beispiel für die Ausgabe von 'verbose:gc'

Geben Sie Folgendes ein:

```
java -Xrealtime -verbose:gc -Xgc:verboseGCCycleTime=N meineAnwendung
```

Das folgende Beispiel zeigt die Anfangsausgabe von 'verbose:gc' an, die die Versions- und Garbage-Collection-Einstellungen enthält:

```
<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=/mein_Verzeichnis/pxi3270hrt-20110719_02/sdk/jre/lib/i386/realtime/options.default"/>
      <vmarg name="-Xjcl:jclse7b_26"/>
      <vmarg
  name="-Dcom.ibm.oti.vm.bootstrap.library.path=/mein_Verzeichnis/pxi3270hrt-20110719_02/sdk/jre/lib/i386/realtime:/
  mein_Verzeichnis/pxi3270hrt-2011071..."/>
      <vmarg
  name="-Dsun.boot.library.path=/mein_Verzeichnis/pxi3270hrt-20110719_02/sdk/jre/lib/i386/realtime:/mein_Verzeichnis/
  pxi3270hrt-20110719_02/sdk/jre/lib..."/>
      <vmarg
  name="-Djava.library.path=/mein_Verzeichnis/pxi3270hrt-20110719_02/sdk/jre/lib/i386/realtime:/mein_Verzeichnis/
  pxi3270hrt-20110719_02/sdk/jre/lib/i38..."/>
      <vmarg name="-Djava.home=/mein_Verzeichnis/pxi3270hrt-20110719_02/sdk/jre"/>
      <vmarg name="-Djava.ext.dirs=/mein_Verzeichnis/pxi3270hrt-20110719_02/sdk/jre/lib/ext"/>
      <vmarg name="-Duser.dir=/mein_Verzeichnis/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>
```

Wird eine Garbage-Collection ausgelöst, tritt ein Ereignis `trigger start` gefolgt von einer Anzahl `heartbeat`-Ereignissen und einem Ereignis `trigger end` auf, wenn der Trigger abgeschlossen ist. Das folgende Beispiel zeigt einen ausgelösten Garbage-Collection-Zyklus als Ausgabe von 'verbose:gc' an:

```
<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" />
```

Die folgenden Ereignistypen können auftreten:

<trigger-start ...>
: Der Anfang eines Garbage-Collection-Zyklus, als die belegte Speicherkapazität den Triggerschwellenwert überschritt. Der Standardschwellenwert ist 50 % des Heapspeichers. Das Attribut `intervalms` ist das Intervall zwischen dem vorherigen Ereignis `trigger end` (mit der ID -1) und diesem Ereignis `trigger start`.

<trigger-end ...>
: Ein Garbage-Collection-Zyklus senkte den verwendeten Speicher erfolgreich unter den Triggerschwellenwert. Wenn ein Garbage-Collection-Zyklus beendet wird, der verwendete Speicher den Triggerschwellenwert jedoch nicht unterschreitet, wird ein neuer Garbage-Collection-Zyklus mit derselben Kontext-ID gestartet. Für jedes Ereignis `trigger start` gibt es ein entsprechendes Ereignis `trigger end` mit derselben Kontext-ID. Das Attribut `intervalms` ist das Intervall zwischen dem vorherigen Ereignis `trigger start` und dem aktuellen Ereignis `trigger end`. Während dieser Zeit wird mindestens ein Garbage-Collection-Zyklus beendet, bis der verwendete Speicher den Triggerschwellenwert unterschreitet.

<gc-op id="28" type="heartbeat"...>
: Ein periodisches Ereignis, das Speicher- und Zeitinformationen zu allen Garbage-Collection-Quanten für den abgedeckten Zeitraum zusammenstellt. Ein Ereignis `heartbeat` kann nur zwischen einem übereinstimmenden Paar von `trigger start`- und `trigger end`-Ereignissen auftreten, d. h., während ein Garbage-Collection-Zyklus aktiv ist. Das Attribut `intervalms` ist das Intervall zwischen dem vorherigen Ereignis `heartbeat` (mit `id` -1) und diesem Ereignis `heartbeat`.

<gc-op id="29" type="syncgc"...>
: Ein synchrones (nicht deterministisches) Garbage-Collection-Ereignis. Weitere Informationen hierzu finden Sie in „Synchrone Garbage-Collections" auf Seite 149.

Die XML-Tags in diesem Beispiel haben die folgenden Bedeutungen:

<quanta ...>
: Eine Zusammenfassung der Quantumpausezeiten während des Heartbeatintervalls, einschließlich die Länge der Pausen in Millisekunden.

<free-mem type="heap" ...>
: Eine Zusammenfassung der freien Heapspeicherkapazität während des Heartbeatintervalls, die am Ende jeden Garbage-Collection-Quantums stichprobenmäßig ermittelt wird.

<classunload-info classloadersunloaded=9 classesunloaded=156 />
: Die Anzahl der Klassenladeprogramme und der Klassen, die während des Heartbeatintervalls entladen wurden.

<references type="weak" cleared="660 />
Die Anzahl und der Typ der Java-Verweisobjekte, die während des Heartbeatintervalls gelöscht wurden.

Anmerkung:

- Wenn nur ein Garbage-Collection-Quantum im Intervall zwischen zwei Heartbeats aufgetreten ist, wird der freie Speicher nur am Ende dieses einen Quantums stichprobenmäßig ermittelt. Daher sind die in der Heartbeatzusammenfassung angegebenen minimalen, maximalen und durchschnittlichen Beträge gleich.
- Das Intervall zwischen zwei `heartbeat`-Ereignissen ist möglicherweise wesentlich größer ist als die angegebene Zykluszeit, wenn der Heapspeicher nicht voll genug ist, um eine Garbage-Collection-Aktivität auszulösen. Wenn z. B. Ihr Programm eine Garbage-Collection-Aktivität nur alle paar Sekunden erfordert, wird ein Heartbeat wahrscheinlich nur alle paar Sekunden angezeigt.
- Es ist möglich, dass das Intervall wesentlich größer ist als die angegebene Zykluszeit, weil die Garbage-Collection für einen Heapspeicher, der nicht voll genug ist, keine Arbeit ausführen muss. Wenn z. B. Ihr Programm eine Garbage-Collection-Aktivität nur alle paar Sekunden erfordert, wird ein Heartbeat wahrscheinlich nur alle paar Sekunden angezeigt.

  Wenn ein Ereignis wie eine synchrone Garbage-Collection oder eine Änderung der Priorität auftritt, werden die Ereignisdetails und anstehende Ereignisse wie Heartbeats unverzüglich als Ausgabe erzeugt.
- Wenn das maximale Garbage-Collection-Quantum für einen angegebenen Zeitraum zu groß ist, können Sie die Zielauslastung mit der Option **-Xgc:targetUtilization** senken. Diese Aktion gibt dem Garbage-Collector mehr Arbeitszeit. Alternativ können Sie die Größe des Heapspeichers mit der Option **-Xmx** erhöhen. Wenn Ihre Anwendung längere Verzögerungen tolerieren kann als zurzeit aufgelistet, können Sie die Zielauslastung erhöhen bzw. die Größe des Heapspeichers senken.
- Die Ausgabe kann mit der Option **-Xverbosegclog:<Datei>** von der Konsole in eine Protokolldatei umgeleitet werden. Beispielsweise schreibt **-Xverbosegclog:out** die Ausgabe von **-verbose:gc** in die Datei *out*.
- Die in `thread-priority` aufgelistete Priorität ist die Priorität des zugrunde liegenden Betriebssystemthreads, keine Java-Threadpriorität.

Synchrone Garbage-Collections

Ein Eintrag wird auch in das Protokoll von **-verbose:gc** geschrieben, wenn eine synchrone (nicht deterministische) Garbage-Collection auftritt. Für dieses Ereignis gibt es drei mögliche Ursachen:

- Ein expliziter Aufruf von `System.gc()` im Code.
- Für die JVM ist kein Speicher mehr verfügbar und es wird eine synchrone Garbage-Collection durchgeführt, um eine Bedingung OutOfMemoryError zu vermeiden.
- Die JVM wird während einer fortlaufenden Garbage-Collection beendet. Die JVM kann die Collection nicht abbrechen. Daher schließt sie die Collection synchron ab und wird dann beendet.

Es folgt ein Beispiel für einen Eintrag `System.gc()`:

```
<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>
```

Es folgt ein Beispiel für einen Eintrag einer synchronen Garbage-Collection als Ergebnis der Beendigung der JVM:

```
| <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>
```

Die XML-Tags und Attribute in diesem Beispiel haben die folgenden Bedeutungen:

| **<gc-op id="9" type="syncgc" timems="6.439" ...**
| Diese Zeile gibt an, dass der Ereignistyp eine synchrone Garbage-Collection ist. Das Attribut `timems` ist die Dauer der synchronen Garbage-Collection in Millisekunden.

| **<syncgc-info reason="..."/>**
| Die Ursache der synchronen Garbage-Collection.

| **<free-mem-delta.../>**
| Der freie Java-Heapspeicher vor und nach der synchronen Garbage-Collection in Byte.

| **<finalization .../>**
| Die Anzahl Objekte, die auf die Endbearbeitung warten.

| **<classunload-info .../>**
| Die Anzahl der Klassenladeprogramme und der Klassen, die während des Heartbeatintervalls entladen wurden.

| **<references type="weak" cleared="53" .../>**
| Die Anzahl und der Typ der Java-Verweisobjekte, die während des Heartbeatintervalls gelöscht wurden.

Es kann nur zu einer synchronen Garbage-Collection aufgrund von abnormalen Speicherbedingungen oder einer VM-Beendigung kommen, wenn der Garbage-Collector aktiv ist. Hierbei muss ein Ereignis `trigger start` vorangehen, obwohl nicht unbedingt unmittelbar. Einige `heartbeat`-Ereignisse treten wahrscheinlich zwischen einem Ereignis `trigger start` und dem Ereignis `synchgc` auf. Eine durch `System.gc()` verursachte synchrone Garbage-Collection kann jederzeit auftreten.

Alle GC-Quanten protokollieren

Einzelne GC-Quanten können durch die Aktivierung der Tracepunkte `GlobalGCStart` und `GlobalGCEnd` protokolliert werden. Diese Tracepunkte werden am Anfang und Ende der Metronom-Garbage-Collector-Aktivität einschließlich synchrone Garbage-Collections erzeugt. Die Ausgabe für diese Tracepunkte ähnelt der folgenden Ausgabe:

```
| 03:44:35.281 0x833cd00 j9mm.52 - GlobalGC start: globalcount=3
|
| 03:44:35.284 0x833cd00 j9mm.91 - GlobalGC end: workstackoverflow=0 overflowcount=0
```

Änderungen der Priorität

Zusätzlich zu den Zusammenfassungen wird ein Eintrag in das Protokoll von **-verbose:gc** geschrieben, wenn sich die Priorität des Garbage-Collector-Threads ändert (weil die Anwendung die Threadprioritäten ändert oder mindestens ein Thread in einer Anwendung beendet wird). Die aufgelistete Priorität ist die Priorität des zugrunde liegenden Betriebssystemthreads, keine Java-Threadpriorität. Es folgt ein Beispiel für einen Eintrag zur Änderung der Garbage-Collector-Threadpriorität:

```
<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>
```

Änderungen der Priorität können in Real Time protokolliert werden, indem die Tracepunktinformationen zu den Prioritäten des Garbage-Collector-Threads erzeugt werden. Die Ausgabe ähnelt der folgenden Ausgabe:

```
15:58:25.493*0x8286e00    j9mm.102       - setGCThreadPriority()
called with newGCThreadPriority = 11
```

Diese Ausgabe kann durch die Verwendung der ID aktiviert werden: **-Xtrace:iprint=tpnid{j9mm.102}**.

Einträge wegen Fehlern aufgrund abnormaler Speicherbedingungen

Wenn für einen der Hauptspeicherbereiche kein Speicherplatz mehr verfügbar ist, wird ein Eintrag in das Protokoll von **-verbose:gc** geschrieben, bevor die Ausnahmebedingung OutOfMemoryError ausgelöst wird. Es folgt ein Beispiel für diese Ausgabe:

```
<out-of-memory id="71" timestamp="2011-07-23T08:32:51.435" memorySpaceName="Scoped"
  memorySpaceAddress="080EED9C"/>
```

Standardmäßig wird ein Java-Speicherauszug als Ergebnis einer Ausnahmebedingung OutOfMemoryError erzeugt. Der folgende Speicherauszug enthält Informationen zum von Ihrem Programm verwendeten Speicher und zu den Bereichen. Zusammen mit dem Wert `J9MemorySpace`, der in der Ausgabe von **-verbose:gc** angegeben wird, können Sie mit den folgenden Information im Speicherauszug den Hauptspeicherbereich ermitteln, für den kein Speicherplatz mehr verfügbar ist:

```
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)
```

Im Beispiel oben kann die in der Ausgabe von **-verbose:gc** angegebene Hauptspeicherkapazitäts-ID (0x080EED9C) mit der ID des Hauptspeicherbereichs 'Scoped Segment' im Java-Speicherauszug abgeglichen werden. Dieser Abgleich kann nützlich sein, wenn Sie über mehrere Bereiche verfügen und ermitteln müssen, für welchen Bereich kein Speicherplatz mehr verfügbar ist, weil die Ausgabe von **-verbose:gc** nur angibt, ob OutOfMemoryError im Speicher für Objekte mit unbeschränkter Lebensdauer, Speicher für Objekte mit beschränkter Lebensdauer oder Heapspeicher aufgetreten ist.

Metronom-Garbage-Collector-Verhalten bei abnormalen Speicherbedingungen:

Der Metronom-Garbage-Collector löst eine uneingeschränkte, nicht deterministische Garbage-Collection aus, wenn für die JVM kein Speicher mehr verfügbar ist. Sie können das nicht deterministische Verhalten mit der Option **-Xgc:noSynchronousGCOnOOM** verhindern, durch die ein OutOfMemoryError ausgelöst wird, wenn für die JVM kein Speicher mehr verfügbar ist.

Die uneingeschränkte Standard-Garbage-Collection wird ausgeführt, bis der gesamte Garbage in einer einzelnen Operation erfasst wurde. Die erforderliche Pausezeit ist gewöhnlich viele Millisekunden länger als ein normales inkrementelles Metronomquantum.

Zugehörige Informationen

Mit '-Xverbose:gc' synchrone Garbage-Collections analysieren

Metronom-Garbage-Collector-Verhalten bei expliziten System.gc()-Aufrufen:

Läuft ein Garbage-Collection-Zyklus, schließt der Metronom-Garbage-Collector den Zyklus auf synchrone Art ab, wenn `System.gc()` aufgerufen wird. Läuft kein Garbage-Collection-Zyklus, wird ein voller synchroner Zyklus ausgeführt, wenn `System.gc()` aufgerufen wird. Mit `System.gc()` können Sie den Heapspeicher auf kontrollierte Weise bereinigen. Hierbei handelt es sich um eine nicht deterministische Operation, weil sie vor ihrer Rückgabe eine vollständige Garbage-Collection ausführt.

Einige Anwendungen rufen die Software anderer Anbieter auf, die über System.gc()-Aufrufe verfügt. Für sie ist es jedoch nicht zulässig, diese nicht deterministischen Verzögerungen zu erstellen. Mit der Option **-Xdisableexplicitgc** können Sie alle System.gc()-Aufrufe inaktivieren.

Die ausführliche Garbage-Collection-Ausgabe für einen System.gc()-Aufruf weist als Ursache „system garbage collect" auf und hat wahrscheinlich eine lange Dauer:

```
| <gc-op id="9" type="syncgc" timems="6.439" contextid="8" timestamp="2011-07-12T09:41:40.808">
|   <syncgc-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>
```

Diagnose bei gemeinsam genutzten Klassen

Für die Verwendung des Modus gemeinsam genutzter Klassen ist ein Verständnis der Vorgehensweise bei der Diagnose von eventuell auftretenden Problemen hilfreich.

Eine Einführung in gemeinsam genutzte Klassen finden im Abschnitt Class data sharing between JVMs.

Das Benutzerhandbuch für IBM SDK for Java 7 enthält nützliche Informationen zum Diagnostizieren von Problemen mit gemeinsam genutzten Klassen:

- Gemeinsam genutzte Klassen implementieren
- Umgang mit Änderungen am Bytecode zur Ausführungszeit
- Dynamische Aktualisierungen - Grundlagen

- Java-Helper-API verwenden
- Diagnosenachrichten für gemeinsam genutzte Klassen - Grundlagen
- Probleme in Zusammenhang mit gemeinsam genutzten Klassen beheben

Sie finden diese Informationen an folgender Stelle: IBM SDK for Java 7 - Diagnose bei gemeinsam genutzten Klassen.

Einige Informationen im Benutzerhandbuch für IBM SDK for Java 7 gelten unter Umständen nicht für IBM WebSphere Real Time for RT Linux. Dies gilt insbesondere für Folgendes:

- Im Echtzeitmodus haben Anwendungen nur Lesezugriff (keinen Lese- und Schreibzugriff) auf Caches für gemeinsam genutzte Klassen.
- Caches können ausschließlich mit dem Tool **admincache** geändert werden.
- Nicht permanente Caches sind im Echtzeitmodus nicht verfügbar.

Mit der JVMTI arbeiten

JVMTI ist eine bidirektionale Schnittstelle, die die Kommunikation zwischen der Java Virtual Machine (JVM) und einem nativen Agenten ermöglicht. Sie ersetzt die JVMDI- und die JVMPI-Schnittstelle.

Mithilfe der JVMTI können andere Anbieter Tools zur Fehlerermittlung, Profilerstellung und Überwachung für die Java Virtual Machine (JVM) entwickeln. Die Schnittstelle enthält Mechanismen, mit der der Agent der JVM mitteilen kann, welche Informationen benötigt werden. Darüber hinaus ermöglicht die Schnittstelle auch den Empfang von Benachrichtigungen. Es können jederzeit mehrere Agenten mit der JVM verbunden werden.

Das Benutzerhandbuch für IBM SDK for Java 7 enthält ausführliche Informationen zur Verwendung von JVMTI, einschließlich einen API-Referenzabschnitt zu IBM Erweiterungen für JVMTI.

Sie finden diese Informationen an folgender Stelle: IBM SDK for Java 7 - Mit JVMTI arbeiten.

Diagnostic Tool Framework for Java verwenden

Das Diagnostic Tool Framework for Java (DTFJ) ist eine Java-Anwendungsprogrammierschnittstelle (API) von IBM zur Unterstützung der Erstellung von Java-Diagnosetools. DTFJ arbeitet mit Daten aus einem Systemspeicherauszug oder einem Java-Speicherauszug.

Das Benutzerhandbuch für IBM SDK for Java 7 enthält ausführliche Informationen zur DTFJ-Verwendung. Folgen Sie diesem Link: Diagnostic Tool Framework for Java verwenden

IBM Monitoring and Diagnostic Tools for Java - Health Center verwenden

IBM Monitoring and Diagnostic Tools for Java - Health Center ist ein Diagnosetool zur Überwachung des Status einer aktiven Java Virtual Machine (JVM).

Informationen zu IBM Monitoring and Diagnostic Tools for Java - Health Center sind in developerWorks und in einem Infocenter verfügbar.

Kapitel 10. Referenz

In diesen Themen werden die Optionen und Klassenbibliotheken aufgelistet, die mit WebSphere Real Time for RT Linux verwendet werden können.

Befehlszeilenoptionen

Sie können beim Starten von Java Optionen in der Befehlszeile angeben. Standardoptionen wurden für die beste allgemeine Verwendung ausgewählt.

Java-Optionen und Systemeigenschaften angeben

Sie können Java-Eigenschaften und Systemeigenschaften auf drei verschiedene Arten angeben.

Informationen zu diesem Vorgang

Sie können Java-Optionen und Systemeigenschaften auf folgende Arten angeben. Sie lauten nach Vorrang wie folgt:

1. Durch Angeben der Option oder der Eigenschaft in der Befehlszeile. Beispiel:

   ```
   java -Dmysysprop1=tcpip -Dmysysprop2=wait -XdisablejavadumpMeineJavaKlasse
   ```

2. Durch Erstellen einer Datei, die die Optionen enthält, und Angeben dieser Datei in der Befehlszeile mit der Option **-Xoptionsfile=**<*Dateiname*>.

   Geben Sie in der Optionsdatei jede Option in einer neuen Zeile an. Sie können das Zeichen '\' als Fortsetzungszeichen verwenden, wenn sich eine Option über mehrere Zeilen erstrecken soll. Verwenden Sie das Zeichen '#', um Kommentarzeilen zu definieren. Sie können die Option **-classpath** nicht in einer Optionsdatei angeben. Im Folgenden finden Sie ein Beispiel für eine Optionsdatei:

   ```
   #My options file
   -X<Option1>
   -X<Option2>=\
   <Wert1>,\
   <Wert2>
   -D<sysprop1>=<Wert1>
   ```

3. Durch Erstellen einer Umgebungsvariable mit der Bezeichnung **IBM_JAVA_OPTIONS**, die die Optionen enthält. Beispiel:

   ```
   export IBM_JAVA_OPTIONS="-Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump"
   ```

Die letzte in der Befehlszeile angegebene Option hat Vorrang vor der ersten Option. Wenn Sie z. B. die Optionen **-Xint -Xjit meineKlasse** angeben, hat die Option **-Xjit** Vorrang vor **-Xint**.

Systemeigenschaften

Systemeigenschaften sind für Anwendungen verfügbar und liefern Informationen zur Laufzeitumgebung.

com.ibm.jvm.realtime
: Mit dieser Eigenschaft können Java-Anwendungen ermitteln, ob Sie in einer WebSphere Real Time for RT Linux-Umgebung ausgeführt werden.

  Wenn Ihre Anwendung in der IBM WebSphere Real Time for RT Linux-Laufzeitumgebung ausgeführt wird und mit der Option **-Xrealtime** gestartet wurde, hat die Eigenschaft **com.ibm.jvm.realtime** den Wert „hard".

Wenn Ihre Anwendung in der IBM WebSphere Real Time for RT Linux-Laufzeitumgebung ausgeführt wird, allerdings nicht mit der Option **-Xrealtime** gestartet wurde, ist die Eigenschaft **com.ibm.jvm.realtime** nicht festgelegt.

Wenn Ihre Anwendung in der IBM WebSphere Real Time-Laufzeitumgebung ausgeführt wird, hat die Eigenschaft **com.ibm.jvm.realtime** den Wert „soft".

Standardoptionen

Die Definitionen für die Standardoptionen.

-agentlib:<*Bibliotheksname*>**[=**<*Optionen*>**]**
Lädt die native Agentenbibliothek <*Bibliotheksname*>; z. B. **-agentlib:hprof**. Geben Sie **-agentlib:jdwp=help** und **-agentlib:hprof=help** in der Befehlszeile an, um weitere Informationen zu erhalten.

-agentpath:*Bibliotheksname***[=**<*Optionen*>**]**
Lädt die native Agentenbibliothek mit dem vollständigen Pfadnamen.

-assert Zeigt Hilfe für Optionen für 'assert' an.

-cp oder -classpath <*Verzeichnisse und ZIP- oder JAR-Dateien getrennt durch* **:**>
Legt den Suchpfad für Anwendungsklassen und -ressourcen fest. Wenn **-classpath** und **-cp** nicht verwendet werden und **CLASSPATH** nicht definiert wurde, ist der Benutzerklassenpfad standardmäßig das aktuelle Verzeichnis (.).

-D<property_name>=<*Wert*>
Definiert eine Systemeigenschaft.

-help oder -?
Gibt einen Verwendungshinweis aus.

-javaagent:<*JAR-Pfad*>**[=**<*Optionen*>**]**
Lädt einen Agenten der Programmiersprache Java. Weitere Informationen finden Sie in der Dokumentation zur API java.lang.instrument.

-jre-restrict-search
Schließt private Benutzer-JREs in die Versionssuche ein.

-no-jre-restrict-search
Nimmt private Benutzer-JREs in die Versionssuche auf.

-showversion
Zeigt die Produktversion an und setzt den Vorgang fort.

-verbose:[*class***,***gc***,***dynload***,***sizes***,***stack***,***jni***]**
Aktiviert die ausführliche Ausgabe.

-verbose:class
Schreibt für jede geladene Klasse einen Eintrag in 'stderr'.

-verbose:gc
Siehe „Informationen von 'verbose:gc' verwenden" auf Seite 146.

-verbose:dynload
Stellt ausführliche Informationen bereit, während die einzelnen Klassen von der JVM geladen werden:
- Klassenname und -paket
- Bei Klassendateien, die sich in einer `.jar`-Datei befanden, der Name und der Verzeichnispfad der `.jar`-Datei

- Details zur Klassengröße und der zum Laden der Klasse benötigten Zeit

Die Daten werden in 'stderr' ausgegeben. Im Folgenden finden Sie eine Beispielausgabe:

```
<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>
```

Anmerkung: Aus dem Cache für gemeinsam genutzte Klassen geladene Klassen werden in der Ausgabe von **-verbose:dynload** nicht angezeigt. Verwenden Sie **-verbose:class**, um Informationen zu diesen Klassen zu erhalten.

-verbose:sizes
Schreibt Informationen in 'stderr', die die für die Stacks und Heapspeicher in der JVM verwendete Speicherkapazität beschreiben.

-verbose:stack
Schreibt Informationen in 'stderr', die die Java- und C-Stack-Belegung beschreiben.

-verbose:jni
Schreibt Informationen in 'stderr', die die von der Anwendung und der JVM aufgerufenen JNI-Services beschreiben.

-version
Gibt Versionsinformationen für den Nicht-Echtzeitmodus aus. Bei Verwendung mit der Option '-Xrealtime' werden Versionsinformationen für den Echtzeitmodus ausgegeben.

-version:<*Wert*>
Erfordert, dass die angegebene Version ausgeführt wird.

-X Zeigt Hilfe für vom Standard abweichende Optionen an.

Vom Standard abweichende Optionen

Optionen mit dem Präfix **-X** weichen vom Standard ab und können jederzeit geändert werden.

Das Benutzerhandbuch für IBM SDK for Java 7 enthält ausführliche Informationen zu vom Standard abweichenden Optionen. Sie finden diese Informationen an folgender Stelle: IBM SDK for Java 7 - Befehlszeilenoptionen.

Ergänzende Informationen für IBM WebSphere Real Time for RT Linux finden Sie in den folgenden Abschnitten.

Echtzeitoptionen

Die Definition der Option **-Xrealtime** in WebSphere Real Time for RT Linux.

Die folgenden **-X**-Optionen sind in der WebSphere Real Time for RT Linux-Umgebung gültig.

-Xrealtime
Startet den Echtzeitmodus. Diese Option ist erforderlich, wenn Sie den Metronom-Garbage-Collector ausführen und die RTSJ-Services verwenden wollen. Wenn Sie diese Option nicht angeben, wird die JVM im Nicht-Echtzeitmodus gestartet, der äquivalent ist zu IBM SDK and Runtime Environment for Linux Platforms, Java 2 Technology Version 7.

Die Option **-Xrealtime** ist mit **-Xgcpolicy:metronome** austauschbar. Sie können eine der beiden Optionen angeben, um den Echtzeitmodus zu aktivieren.

Ahead-of-time-Optionen

Die Definitionen für die Ahead-of-time-Optionen (AOT-Optionen).

Zweck

Keine Option angegeben:
Wird mit dem Interpreter und dynamisch kompilierten Code ausgeführt. Wird AOT-Code erkannt, wird er nicht verwendet. Er wird anstelle dessen bei Bedarf dynamisch kompiliert. Dies ist für Nicht-Echtzeitanwendungen und Echtzeitanwendungen besonders nützlich. Diese Option bietet optimale Leistung und optimalen Durchsatz, kann jedoch nicht-deterministischen Verzögerungen während der Ausführung unterliegen, wenn die Kompilierung auftritt.

-Xjit: Diese Option entspricht dem Standardwert.

-Xint: Führt nur den Interpreter aus, ignoriert den für AOT geschriebenen Code, der in einer vorkompilierten JAR-Datei gefunden wird, und führt den dynamischen Compiler nicht aus. Dieser Modus wird selten benötigt, in der Regel nur für die Behebung von Fehlern, die mit der Kompilierung zusammenhängen, oder für sehr kurze Stapelanwendungen, die von der Kompilierung nicht profitieren.

-Xnojit:
Führt den Interpreter aus und verwendet für AOT geschriebenen Code, wenn er in einer vorkompilierten JAR-Datei gefunden wird. Diese Option führt den dynamischen Compiler nicht aus. Dieser Modus funktioniert gut für einige Echtzeitanwendungen, bei denen Sie sicherstellen wollen, dass es zu keinen nicht-deterministischen Verzögerungen während der Ausführung aufgrund der Kompilierung kommt. Für AOT geschriebener Code kann nur bei Ausführung mit der Option **-Xrealtime** verwendet werden. Er wird bei der Ausführung in einer Standard-JVM, d. h., wenn **-Xrealtime** nicht angegeben wird, nicht unterstützt.

Beispiel

```
java -Xrealtime -Xnojit outputtest.jar.
```

Optionen für den Metronom-Garbage-Collector

Die Definitionen der Optionen für den Metronom-Garbage-Collector.

-Xgc:immortalMemorySize=*Größe*
Gibt die Größe des Heapspeicherbereichs für Objekte mit unbeschränkter Lebensdauer an. Der Standardwert ist 16 MB.

-Xgc:scopedMemoryMaximumSize=*Größe*
Gibt die Größe des Heapspeicherbereichs für Objekte mit beschränkter Lebensdauer an. Der Standardwert ist 8 MB.

-Xgc:synchronousGCOnOOM | -Xgc:nosynchronousGCOnOOM
Die Garbage-Collection tritt auf, wenn im Heapspeicher kein Speicherplatz mehr verfügbar ist. Ist im Heapspeicher kein Speicherplatz mehr verfügbar, stoppt die Verwendung von **-Xgc:synchronousGCOnOOM** Ihre Anwendung, während die Garbage-Collection nicht verwendete Objekte entfernt. Ist erneut kein Speicherplatz mehr verfügbar, reduzieren Sie die Zielauslastung, um der Garbage-Collection mehr Zeit zum Abschließen zu geben. Wird **-Xgc:nosynchronousGCOnOOM** angegeben und ist der Heapspeicher voll, wird

Ihre Anwendung gestoppt und setzt eine Nachricht zu einem Fehler aufgrund abnormaler Speicherbedingungen ab. Der Standardparameter ist **-Xgc:synchronousGCOnOOM**.

-Xnoclassgc
Inaktiviert die Garbage-Collection für Klassen. Diese Option inaktiviert die Garbage-Collection für Speicher, der Java-Klassen zugeordnet ist, die nicht mehr von der JVM verwendet werden sollen. Das Standardverhalten ist **-Xnoclassgc**.

-Xgc:targetUtilization=*N*
Setzt die Anwendungsauslastung auf N %. Der Garbage-Collector versucht, höchstens (100-N) % jeden Zeitintervalls zu verwenden. Angemessene Werte liegen im Bereich zwischen 50-80 %. Anwendungen mit niedrigen Zuordnungsraten können mit 90 % ausgeführt werden. Der Standardwert beträgt 70 %.

Das folgende Beispiel zeigt, dass die maximale Größe des Heapspeichers 30 MB beträgt. Der Garbage-Collector versucht, 25 % jeden Zeitintervalls zu verwenden, weil die Zielauslastung für die Anwendung 75 % ist.

```
java -Xrealtime -Xmx30m -Xgc:targetUtilization=75 Test
```

-Xgc:threads=*N*
Gibt die Anzahl auszuführender GC-Threads an. Der Standardwert ist 1.

-Xgc:verboseGCCycleTime=N
N ist die Zeit in Millisekunden, die bis zum Erstellen eines Speicherauszugs der Übersichtsdaten verstreichen soll.

Anmerkung: Die Zykluszeit bedeutet nicht, dass ein Speicherauszug der Übersichtsdaten genau zu dieser Zeit erstellt wird, sondern wenn das letzte Garbage-Collection-Ereignis stattfindet, das dieses Zeitkriterium erfüllt.

-Xmx<*Größe*>
Gibt die Java-Heapspeichergröße an. Im Gegensatz zu anderen Garbage-Collection-Strategien unterstützt die Echtzeit-Metronom-Garbage-Collection nicht die Heapspeichererweiterung. Sie können nicht zwischen einer Anfangsgröße des Heapspeichers und einer Maximalgröße des Heapspeichers wählen. Sie können nur die Maximalgröße des Heapspeichers angeben.

-Xthr:metronomeAlarm=os*xx*
Steuert die Priorität, mit der der Alarmthread des Metronom-Garbage-Collectors ausgeführt wird.

Dabei ist *xx* eine Zahl zwischen 11 und 89, die die Priorität angibt, mit der der Metronomalarmthread ausgeführt werden soll. Gehen Sie sorgfältig vor, wenn Sie die Betriebssystempriorität modifizieren, mit der der Alarmthread ausgeführt wird. Wenn Sie eine Betriebssystempriorität angeben, die unter der Priorität eines Echtzeitthreads liegt, treten OutOfMemory-Fehler auf, weil der Garbage-Collector mit einer Priorität ausgeführt wird, die unter der Priorität der Echtzeitthreads liegt, die Garbage zuordnen. Der Standardalarmthread des Metronom-Garbage-Collectors wird mit der Betriebssystempriorität 89 ausgeführt.

Standardeinstellungen für die JVM

Die Standardeinstellungen gelten für die Echtzeit-JVM, wenn an der Umgebung, in der die JVM ausgeführt wird, keine Änderungen vorgenommen werden. Allgemeine Einstellungen werden zu Referenzzwecken angezeigt.

Die Standardeinstellungen können über Umgebungsvariablen oder Befehlszeilenparameter beim JVM-Start geändert werden. Die Tabelle zeigt einige der allgemeinen JVM-Einstellungen an. Die letzte Spalte gibt an, wie Sie das Verhalten ändern können. Dabei gelten die folgenden Schlüssel:

- **U**: nur von der Umgebungsvariable gesteuerte Einstellung
- **B**: nur vom Befehlszeilenparameter gesteuerte Einstellung
- **UB**: von der Umgebungsvariable und vom Befehlszeilenparameter gesteuerte Einstellung, wobei der Befehlszeilenparameter Vorrang hat

Die Informationen diesen als Kurzübersicht und sind nicht umfassend.

JVM-Einstellung	Standard-einstellung	Einstellung beeinflusst durch:
Java-Speicherauszüge	aktiviert	UB
Java-Speicherauszüge bei nicht genügend Speicherkapazität	aktiviert	UB
Heapspeicherauszüge	inaktiviert	UB
Heapspeicherauszüge bei nicht genügend Speicherkapazität	aktiviert	UB
Systemspeicherauszüge	aktiviert	UB
Wo Speicherauszugsdateien erstellt werden	aktuelles Verzeichnis	UB
Ausgabe über 'verbose'	inaktiviert	B
Suche im Bootklassenpfad	inaktiviert	B
JNI-Prüfungen	inaktiviert	B
Fernes Debugging	inaktiviert	B
Genaue Übereinstimmungsprüfungen	inaktiviert	B
Schnellstart	inaktiviert	B
Ferner Debuginformationsserver	inaktiviert	B
Reduzierte Signalisierung	inaktiviert	B
Verkettung von Signalhandlern	aktiviert	B
Klassenpfad	nicht festgelegt	UB
Gemeinsame Nutzung von Klassendaten	inaktiviert	B
Unterstützung von Eingabehilfen	aktiviert	U
JIT-Compiler	aktiviert	UB
AOT-Compiler (AOT wird von JVM nur verwendet, wenn auch gemeinsam genutzte Klassen aktiviert sind)	aktiviert	B
JIT-Debugoptionen	inaktiviert	B
Java2D-Schriftarten mit algorithmischer Fettschrift in maximaler Größe	14 Punkte	U

JVM-Einstellung	Standard-einstellung	Einstellung beeinflusst durch:
Verwendung von wiedergegebenen Bitmapdateien in skalierbaren Java2D-Schriftarten	aktiviert	U
Rastern von Java2D-FreeType-Schriftarten	aktiviert	U
Verwendung von Java2D-AWT-Schriftarten	inaktiviert	U
Länderspezifische Standardeinstellung	keine	U
Wartezeit vor Start des Plug-in	null	U
Temporäres Verzeichnis	/tmp	U
Umleitung des Plug-ins	keine	U
IM-Switching	inaktiviert	U
IM-Modifikatoren	inaktiviert	U
Threading-Modell	nicht zutreffend	U
Anfängliche Stackgröße für 32-Bit-Java-Threads. Verwenden Sie: **-Xiss<Größe>**	2 KB	B
Maximale Stackgröße für 32-Bit-Java-Threads. Verwenden Sie: **-Xss<Größe>**	256 KB	B
Stackgröße für 32-Bit-Betriebssystemthreads. Verwenden Sie **-Xmso<Größe>**	256 KB	B
Anfangsgröße des Heapspeichers. Verwenden Sie **-Xms<Größe>**	64 MB	B
Maximale Größe des Java-Heapspeichers. Verwenden Sie **-Xmx<Größe>**	Die Hälfte der verfügbaren Hauptspeichergröße, Minimum 16 MB, Maximum 512 MB	B
Nutzung des Sollzeitintervalls für eine Anwendung. Der Garbage-Collector versucht, den Rest zu verwenden. Verwenden Sie **-Xgc:targetUtilization=<Prozentsatz>**.	70 %	B
Die Anzahl der auszuführenden Garbage-Collector-Threads. Verwenden Sie **-Xgc:threads=<Wert>**	1	B
Maximale Speicherkapazität, die Bereichsspeichern im Modus **-Xrealtime** zugeordnet werden kann. Verwenden Sie **-Xgc:scopedMemoryMaximumSize=<Größe>**.	8 MB	B
Größe des Speicherbereichs für Objekte mit unbeschränkter Lebensdauer im Modus **-Xrealtime**. Verwenden Sie **-Xgc:immortalMemorySize=<Größe>**	16 MB	B

Anmerkung: Der „verfügbare Hauptspeicher" ist entweder die Kapazität des Realspeichers (physischen Speichers) oder der Wert von **RLIMIT_AS**, wobei der niedrigere Wert ausschlaggebend ist.

Klassenbibliotheken von WebSphere Real Time for RT Linux

Ein Verweis auf die von WebSphere Real Time for RT Linux verwendeten Java-Klassenbibliotheken.

Die von WebSphere Real Time for RT Linux verwendeten Java-Klassenbibliotheken werden in http://www.rtsj.org/specjavadoc/book_index.html beschrieben.

Mit TCK ausführen

Wenn Sie Real-Time Specification for Java (RTSJ) Technology Compatibility Kit (TCK) mit WebSphere Real Time for RT Linux ausführen, müssen Sie `demo/realtime/TCKibm.jar` in den Klassenpfad aufnehmen, damit Tests erfolgreich abgeschlossen werden können.

`TCKibm.jar` enthält die Klasse **`VibmcorProcessorLock`**, die die IBM Erweiterung der Klasse TCK.ProcessorLock ist. Diese Klasse stellt das Einzelprozessorverhalten bereit, das in einer kleinen Gruppe von TCK-Tests erforderlich ist. Weitere Informationen zur Klasse TCK.ProcessorLock und zu anbieterspezifischen Erweiterungen für diese Klasse finden Sie in der Readme-Datei, die mit der TCK-Verteilung geliefert wird.

Kapitel 11. Bemerkungen

Die vorliegenden Informationen wurden für Produkte und Services entwickelt, die auf dem deutschen Markt angeboten werden. Möglicherweise bietet IBM die in dieser Dokumentation beschriebenen Produkte, Services oder Funktionen in anderen Ländern nicht an. Informationen über die gegenwärtig im jeweiligen Land verfügbaren Produkte und Services sind beim zuständigen IBM Ansprechpartner erhältlich. Hinweise auf IBM Lizenzprogramme oder andere IBM Produkte bedeuten nicht, dass nur Programme, Produkte oder Services von IBM verwendet werden können. Anstelle der IBM Produkte, Programme oder Services können auch andere, ihnen äquivalente Produkte, Programme oder Services verwendet werden, solange diese keine gewerblichen oder anderen Schutzrechte von IBM verletzen. Die Verantwortung für den Betrieb von Produkten, Programmen und Services anderer Anbieter liegt beim Kunden.

Für in dieser Dokumentation beschriebene Erzeugnisse und Verfahren kann es IBM Patente oder Patentanmeldungen geben. Mit der Auslieferung dieser Dokumentation ist keine Lizenzierung dieser Patente verbunden. Lizenzanforderungen sind schriftlich an folgende Adresse zu richten (Anfragen an diese Adresse müssen auf Englisch formuliert werden):

IBM Director of Licensing
IBM Europe, Middle East & Africa
Tour Descartes
2, avenue Gambetta
92066 Paris La Defense
France

Trotz sorgfältiger Bearbeitung können technische Ungenauigkeiten oder Druckfehler in dieser Veröffentlichung nicht ausgeschlossen werden. Die hier enthaltenen Informationen werden in regelmäßigen Zeitabständen aktualisiert und als Neuausgabe veröffentlicht. IBM kann ohne weitere Mitteilung jederzeit Verbesserungen und/oder Änderungen an den in dieser Veröffentlichung beschriebenen Produkten und/oder Programmen vornehmen.

Verweise in diesen Informationen auf Websites anderer Anbieter werden lediglich als Service für den Kunden bereitgestellt und stellen keinerlei Billigung des Inhalts dieser Websites dar. Das über diese Websites verfügbare Material ist nicht Bestandteil des Materials für dieses IBM Produkt. Die Verwendung dieser Websites geschieht auf eigene Verantwortung.

Werden an IBM Informationen eingesandt, können diese beliebig verwendet werden, ohne dass eine Verpflichtung gegenüber dem Einsender entsteht.

Lizenznehmer des Programms, die Informationen zu diesem Produkt wünschen mit der Zielsetzung: (i) den Austausch von Informationen zwischen unabhängig voneinander erstellten Programmen und anderen Programmen (einschließlich des vorliegenden Programms) sowie (ii) die gemeinsame Nutzung der ausgetauschten Informationen zu ermöglichen, wenden sich an folgende Adresse:

- JIMMAIL@uk.ibm.com [Ansprechpartner im Hursley Java Technology Center (JTC)]

Die Bereitstellung dieser Informationen kann unter Umständen von bestimmten Bedingungen - in einigen Fällen auch von der Zahlung einer Gebühr - abhängig sein.

Die Lieferung des in diesem Dokument beschriebenen Lizenzprogramms sowie des zugehörigen Lizenzmaterials erfolgt auf der Basis der IBM Rahmenvereinbarung bzw. der Allgemeinen Geschäftsbedingungen von IBM, der IBM Internationalen Nutzungsbedingungen für Programmpakete oder einer äquivalenten Vereinbarung.

Alle in diesem Dokument enthaltenen Leistungsdaten stammen aus einer kontrollierten Umgebung. Die Ergebnisse, die in anderen Betriebsumgebungen erzielt werden, können daher erheblich von den hier erzielten Ergebnissen abweichen. Einige Daten stammen möglicherweise von Systemen, deren Entwicklung noch nicht abgeschlossen ist. Eine Gewährleistung, dass diese Daten auch in allgemein verfügbaren Systemen erzielt werden, kann nicht gegeben werden. Darüber hinaus wurden einige Daten unter Umständen durch Extrapolation berechnet. Die tatsächlichen Ergebnisse können davon abweichen. Benutzer dieses Dokuments sollten die entsprechenden Daten in ihrer spezifischen Umgebung prüfen.

Alle Informationen zu Produkten anderer Anbieter stammen von den Anbietern der aufgeführten Produkte, deren veröffentlichten Ankündigungen oder anderen allgemein verfügbaren Quellen. IBM hat diese Produkte nicht getestet und kann daher keine Aussagen zu Leistung, Kompatibilität oder anderen Merkmalen machen. Fragen zu den Leistungsmerkmalen von Produkten anderer Anbieter sind an den jeweiligen Anbieter zu richten.

Marken

IBM, das IBM Logo und ibm.com sind Marken oder eingetragene Marken der IBM Corporation in den USA und/oder anderen Ländern. Sind diese und weitere Markennamen von IBM bei ihrem ersten Vorkommen in diesen Informationen mit einem Markensymbol (® oder ™) gekennzeichnet, bedeutet dies, dass IBM zum Zeitpunkt der Veröffentlichung dieser Informationen Inhaber der eingetragenen Marken oder der Common-Law-Marken (common law trademarks) in den USA war. Diese Marken können auch eingetragene Marken oder Common-Law-Marken in anderen Ländern sein. Eine aktuelle Liste der IBM Marken finden Sie auf der Webseite "Copyright and trademark information" unter http://www.ibm.com/legal/copytrade.shtml.

Adobe, das Adobe-Logo, PostScript und das PostScript-Logo sind Marken oder eingetragene Marken der Adobe Systems Incorporated in den USA und/oder anderen Ländern.

Intel und Itanium sind Marken der Intel Corporation oder ihrer Tochtergesellschaften in den USA oder anderen Ländern.

Linux ist eine Marke von Linus Torvalds in den USA und/oder anderen Ländern.

Java und alle auf Java basierenden Marken und Logos sind Marken oder eingetragene Marken der Oracle Corporation und/oder ihrer verbundenen Unternehmen.

Weitere Unternehmens-, Produkt- oder Servicenamen können Marken anderer Hersteller sein.

Index

Sonderzeichen

A

B

C

D

E

F

G

H

I

T

V

Z