IBM WebSphere Real Time for RT Linux V 3

用户指南

IBM WebSphere Real Time for RT Linux V 3

用户指南

注意

在使用本资料及其支持的产品之前，请阅读第 139 页的『声明』中的信息。

第五版（2014 年 2 月）

本版本的用户指南适用于 IBM WebSphere Real Time for RT Linux V3 以及所有后续发行版和修订版，直到在新版本中另有声明为止。

目录

图

表

前言

本用户指南提供关于 IBM® WebSphere® Real Time for RT Linux 的常规信息。

第 1 章 简介

本文为您介绍 IBM WebSphere Real Time for RT Linux。

对该用户指南做出的任何新修改均通过更改左侧的竖线来指示。

本指南中未提供的有关 IBM WebSphere Real Time for RT Linux 的最新信息可以在以下位置中找到: http://www.ibm.com/support/docview.wss?uid=swg21501145

WebSphere Real Time for RT Linux 概述

WebSphere Real Time for RT Linux 将实时功能与 IBM J9 虚拟机 (JVM) 捆绑在一起。

WebSphere Real Time for RT Linux 是一种 Java™ 运行时环境，且带有以实时功能扩展 IBM SDK for Java 的软件开发工具包 (SDK)。依赖于精确响应时间的应用程序可利用随 WebSphere Real Time for RT Linux（基于标准 Java 技术）提供的实时功能。

功能

实时应用程序需要的是稳定的运行时，而不是绝对速度。

当 JVM 在实时方式下运行时，除垃圾回收的堆外还有其他内存区域可供使用。程序可能请求或指定任意数目的可复用作用域和不可复用永久内存区域（这些区域不是垃圾回收的）。该功能为应用程序提供更强的内存使用情况控制。它还使用 Metronome 垃圾回收器 以实现基于时间的回收。当 JVM 在传统的吞吐量方式下运行时，可使用各种基于工作的垃圾回收器，这样可优化吞吐量，但是所产生的个别延迟比 Metronome 垃圾回收器 大。

使用传统 JVM 部署实时应用程序时的主要考虑事项如下所述:
- 由垃圾回收 (GC) 活动导致的不可预测（可能非常长）延迟。
- 出现即时 (JIT) 编译和重新编译时方法运行时出现延迟，且执行时间有变化。
- 任意的操作系统调度。

WebSphere Real Time for RT Linux 通过提供以下项除去这些障碍:
- Metronome 垃圾回收器，一种递增的确定性垃圾回收器，暂停时间极短。
- 提前 (AOT) 编译
- 基于优先级的 FIFO 调度。

另外，WebSphere Real Time 还为实时程序员提供 RTSJ 功能；请参阅第 9 页的『对 RTSJ 的支持』。

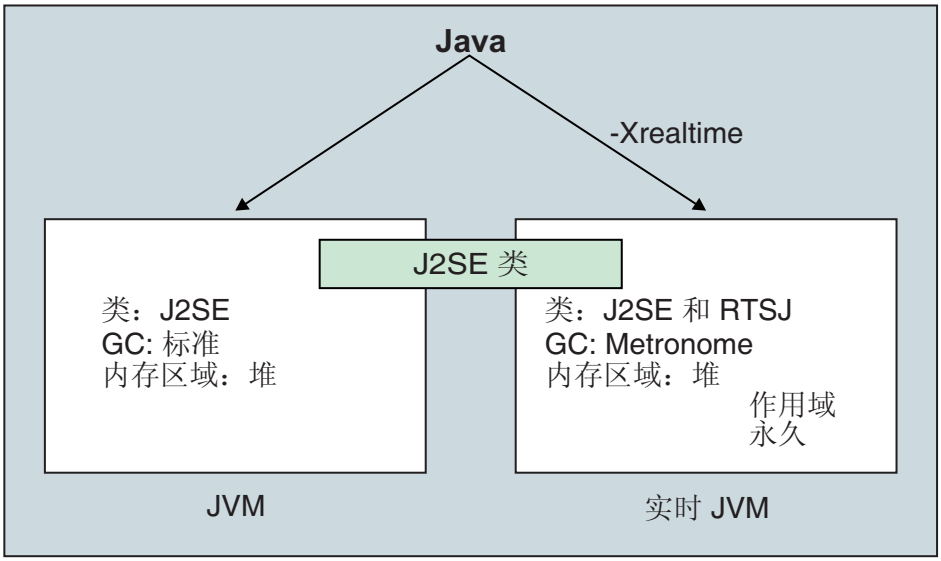

图 1. WebSphere Real Time for RT Linux 概述

运行 JVM 或任何提供的工具时，通过使用 -Xrealtime 选项即可启用实时功能。缺省情况下，运行 JVM 和提供的工具时未启用实时功能。图 1 显示随 WebSphere Real Time for RT Linux 提供的两个 JVM 的关系。

以下 Java 命令识别 -Xrealtime 选项:

表 1. 实时方式中使用的 Java 命令

命令	功能
java	缺省情况下以标准方式运行，但是当指定 -Xrealtime 选项时也以实时方式运行。在实时方式中，程序员访问 javax.realtime 程序包中的类。您可以使用预编译的 JAR 文件和 Metronome 确定性垃圾回收技术。
javac、javah 和 javap	缺省情况下以标准方式运行；但是当指定 -Xrealtime 选项时，它将 javax.realtime.* 类包含在类路径中。
admincache	可在指定或不指定 -Xrealtime 的情况下运行，但是要使用 **admincache** 工具填充共享高速缓存只能在实时方式下操作。在常规方式中，只有高速缓存实用程序（如 listAllCaches 或 printStats）可用。和 **jdmpview** 一样，**admincache** 必须使用 -Xrealtime 运行才可访问实时 JVM 的高速缓存，不使用 -Xrealtime 运行时才可访问常规 JVM 的高速缓存。
jextract	缺省情况下，jextract 以标准方式运行，但是以实时方式处理 JVM 生成的系统转储时必须指定 -Xrealtime 选项运行。

新增内容

本主题介绍 IBM WebSphere Real Time for RT Linux 的更改内容。

WebSphere Real Time for RT Linux V3

WebSphere Real Time for RT Linux V3 是对 IBM SDK for Java V7 的扩展，以本发行版所提供的特性和功能为构建基础，包含实时功能。WebSphere Real Time for RT Linux 的先前版本均以 IBM SDK for Java 的先前发行版为基础。

要了解有关 IBM SDK for Java V7 中新增功能的更多信息，请参阅 IBM SDK for Java 7 信息中心中的新增内容。

对该用户指南做出的任何新修改均通过更改左侧的竖线来指示。

实时 Linux 操作系统支持

目前支持 Red Hat Enterprise MRG 2.2 for Red Hat Enterprise Linux 6 Update 3。要了解有关受支持的硬件和软件的更多信息（包括任何所需的更新或补丁），请参阅第 21 页的『硬件和软件先决条件』。

使用大页面

在服务刷新 4 中，IBM SDK for Java V7 引入了缺省情况下在 x86 和 AMD64/EM64T 平台上使用大页面的功能。虽然 WebSphere Real Time for RT Linux 是对 IBM SDK for Java V7 的扩展，但由于一个已知问题，缺省情况下是禁用大页面的。

符号解析

缺省情况下，JVM 会立即对用户本机库中的每个函数进行符号解析。使用 **-XX:+LazySymbolResolution** 选项可强制 JVM 延迟对用户本机库中的所有函数进行符号解析，直至该函数被调用。有关更多信息，请参阅第 133 页的『非标准选项』。

实时 Linux 操作系统支持

目前支持 SUSE Linux Enterprise Real Time V11。要了解有关受支持的硬件和软件的更多信息（包括任何所需的更新或补丁），请参阅第 21 页的『硬件和软件先决条件』。

jxeinajar

WebSphere Real Time for RT Linux V3 不再支持使用 jxeinajar。要参考有关 jxeinajar 的早期信息，尤其是如何迁移到 admincache 的信息，请访问 WebSphere Real Time for RT Linux V2 文档。

优势

实时环境的优点是 Java 应用程序能够以比使用标准 JVM 更高的可预测性运行，并提供一致的 Java 应用程序计时行为。会在指定的时间执行后台活动（如编译和垃圾回收），从而在运行应用程序时除去任何意外的后台活动峰值。

使用以下功能扩展 JVM 即可获得这些优势:

- Metronome 实时垃圾回收技术
- 提前 (AOT) 编译
- 对 Real-Time Specification for Java (RTSJ) 的支持

所有 Java 应用程序均无需修改即可在实时环境中运行，这受益于 Metronome 垃圾回收器及其定期执行的确定性垃圾回收。要想充分利用 WebSphere Real Time for RT Linux 的所有优势，可同时使用实时线程和非堆实时线程来编写特定于实时环境的应用程序。您采取的方法取决于应用程序的计时规范。

许多实时 Java 应用程序都可利用 Metronome 垃圾回收器的低暂停时间和 AOT 实现其目标，同时保留 Java 可移植性的优点。需求较严格的应用程序必须将实时线程和非堆实时线程的 RTSJ 功能与作用域内存和永久内存结合使用。此方法会对应用程序产生限制使其只能在实时环境中运行，从而丧失了 JSE Java 的可移植性优势。还必须开发更为复杂的编程模型。

辅助功能选项

辅助功能选项功能部件帮助带有某些缺陷（如灵活性受限制或视力有限）的用户成功地使用信息技术。

IBM 力求为不同年龄或能力的所有人提供可访问的产品。

例如，可以在没有鼠标的情况下，通过只使用键盘来操作 WebSphere Real Time for RT Linux。

要读取有关影响底层 IBM SDK for Java V7 的辅助功能选项问题的信息，请参阅 IBM 信息中心。WebSphere Real Time for RT Linux 中没有影响独特功能部件和功能的辅助功能选项问题。

键盘导航

本产品使用标准的 Microsoft Windows 导航键。

对于需要使用键盘导航的用户，可在 Swing密钥绑定处找到 Swing 应用程序的有用击键的描述。

IBM 和辅助功能选项

请参阅 IBM Human Ability and Accessibility Center 以获取有关 IBM 对辅助功能选项的承诺的更多信息。

第 2 章 了解 IBM WebSphere Real Time for RT Linux

本部分介绍 IBM WebSphere Real Time for RT Linux 的关键组件。

Metronome 垃圾回收器简介

Metronome 垃圾回收器用于取代 WebSphere Real Time for RT Linux 中的标准垃圾回收器。

Metronome 垃圾回收与标准垃圾回收之间的关键差异在于 Metronome 垃圾回收发生在较小的可中断步骤中，但标准垃圾回收会在标记和回收垃圾的过程中停止应用程序。

例如:

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

该示例指定应用程序在每 60 毫秒中运行 80% 的时间。其余 20% 的时间可用于垃圾回收（如果有垃圾要回收）。如果给予 Metronome 垃圾回收器足够的资源，那么它可保证达到利用率级别。当堆中的可用空间量降低到动态确定的阈值以下时，会开始进行垃圾回收。

垃圾回收和优先级

垃圾回收线程必须以高于在堆中生成垃圾的最高优先级线程的优先级运行，否则，它可能无法按已配置的利用率所指定的那样运行。常规 Java 线程和实时线程均可能会生成垃圾，因此垃圾回收必须以高于所有常规和实时线程的优先级运行。此优先级划分由 JVM 自动处理，而垃圾回收以高出所有常规和实时线程的最高优先级 0.5 的优先级运行。然而，确保非堆实时线程 (NHRT) 不受垃圾回收影响很重要。请以高于最高优先级实时线程的优先级运行所有 NHRT。这意味着 NHRT 以高于垃圾回收的优先级运行，而不会被延迟。

表 2 显示了您可以定义的优先级的典型示例以及根据您的选择而得出的相关垃圾回收优先级。

有关 Java 优先级和操作系统优先级的比较，请参阅第 11 页的『优先级映射和继承』。

表 2. 垃圾回收和优先级的示例

线程	优先级（示例）
如果最高优先级实时线程为:	20（操作系统优先级 43）
那么垃圾回收器为:	20.5（操作系统优先级 44）

表 2. 垃圾回收和优先级的示例 (续)

线程	优先级（示例）
要确保 NHRT 独立于垃圾回收器运行，请设置高于 GC 的优先级:	21（操作系统优先级 45）或更高。
Metronome 警报线程为:	优先级 46（操作系统优先级 89）

注: 即使使用该配置，非堆实时线程也不会完全不受垃圾回收的影响，因为 Metronome 警报线程在系统中以最高优先级运行，以确保它能够定期唤醒并计算出垃圾回收是否需要执行任何操作。当然，为达到此目的要做的工作很少，因此不是很重要的考虑因素。

Metronome 垃圾回收和类卸载

Metronome 垃圾回收器不会卸载 IBM WebSphere Real Time 中的类，因为这可能需要非确定性工作量，从而导致出现暂停时间界外值。

Metronome 垃圾回收器线程

Metronome 垃圾回收器由两种类型的线程构成: 单个警报线程以及若干个回收 (GC) 线程。缺省情况下，只有一个 GC 线程。您可以使用 **-Xgcthreads** 选项来设置 JVM 的 GC 线程数。

您不能更改 JVM 的警报线程数。

Metronome 垃圾回收器定期检查 JVM 以查看堆内存是否具有足够的可用空间。当可用空间量降低至限制值以下时，Metronome 垃圾回收器将促使 JVM 开始垃圾回收。

警报线程

单个警报线程可保证使用最少资源。它定期“唤醒”，并执行以下检查:

- 堆内存中的可用空间量
- 垃圾回收当前是否正在进行

如果没有足够的可用空间，并且未在执行垃圾回收，那么警报线程将触发回收线程以开始垃圾回收。警报线程在其检查 JVM 的下一安排时间之前不会执行任何操作。

回收线程

每个回收线程均在 Java 和实时线程中检查对象。它们按以下顺序检查内存区域:

1. 设置了作用域的内存，以确认并标记堆中正在由设置了作用域的内存中对象使用的任何活动对象。
2. 永久内存，以确认并标记堆中正在由永久内存中对象使用的任何活动对象。
3. 堆内存，以确认并标记活动对象。

标记了活动对象后，未标记的对象将可供回收。

垃圾回收周期完成后，Metronome 垃圾回收器将检查可用堆空间量。如果仍没有足够的可用堆空间，那么将使用同一触发器标识来启动另一个垃圾回收周期。如果有足够的可用堆空间，那么触发器将结束，而垃圾回收线程将停止。警报线程会继续监控可用堆空间，并将在需要时触发另一个垃圾回收周期。

有关使用 Metronome 垃圾回收器的更多信息，请参阅第 61 页的『使用 Metronome 垃圾回收器』。

编译器

IBM WebSphere Real Time for RT Linux 支持多种代码编译的模型，提供了不同级别的代码性能和确定性。

在 IBM WebSphere Real Time for RT Linux 中有关编译 Java 代码的可用选项包括:

低优先级即时 (JIT) 编译

WebSphere Real Time for RT Linux 中的缺省编译模式使用即时编译器，它在应用程序运行时编译 Java 应用程序的重要方法。在这种方式下，JIT 编译器的工作方式与 JIT 编译器在非实时 JVM 中的操作相似。所不同的是 WebSphere Real Time for RT Linux JIT 编译器运行的优先级比任何实时线程都要低。较低的优先级意味着，当应用程序不需要执行实时任务时，JIT 编译器才会使用系统资源。带来的效果是 JIT 编译器不会明显影响实时任务的性能。

提前 (AOT) 预编译的代码

在运行应用程序之前，WebSphere Real Time for RT Linux 会在预编译步骤中将 Java 方法编译为本机代码。使用 AOT 预编译代码提供了最高级别的确定性和良好的性能。

混合方式，结合 AOT 预编译代码和低优先级 JIT 编译

当应用程序运行时，可以同时使用 AOT 和 JIT 编译的代码。这种操作方式可以提供非常好的确定性和良好的性能，并且为频繁运行的方法提供了非常高的性能。

经过解释的操作

解释器运行 Java 应用程序，但完全不使用任何代码编译。

有关使用经过编译的代码的更多信息，请参阅第 36 页的『在 WebSphere Real Time for RT Linux 中使用经过编译的代码』。

比较 JIT 和 AOT 编译

提前 (AOT) 编译使您能够在运行代码之前编译 Java 类和方法。AOT 编译避免了 JIT 编译器可能对性能敏感路径造成的不可预测的计时影响。为了保证您的代码在执行之前经过编译，并且实现最高级别的预期性能，您可以使用 AOT 编译器将代码预编译到共享类高速缓存中。

注: AOT 编译的代码运行速度通常没有 JIT 编译的代码速度快。

即时 (JIT) 编译器作为高优先级的 SCHED_OTHER 线程运行，运行的优先级高于标准 Java 线程，但低于实时线程的优先级。因此，即时编译不会在实时节点中引起非预期的延迟。这样一来，重要的实时任务就能按时完成，因为 JIT 编译器不会占先。但是，由于 JIT 没有足够的时间来编译已排队的热点方法，因此实时节点可能会作为解释节点来运行。第 8 页的图 2 显示了比较结果。

通常，如果您的应用程序有一个预热阶段，则使用 JIT 运行会更高效，如有必要，请在预热阶段完成之后禁用 JIT。这种方法使 JIT 编译器能够为运行应用程序的环境生成代码。

如果应用程序没有预热阶段，并且不清楚是否会在标准应用程序操作的过程中编译执行的关键路径，则 AOT 编译在这种环境中更加有效。

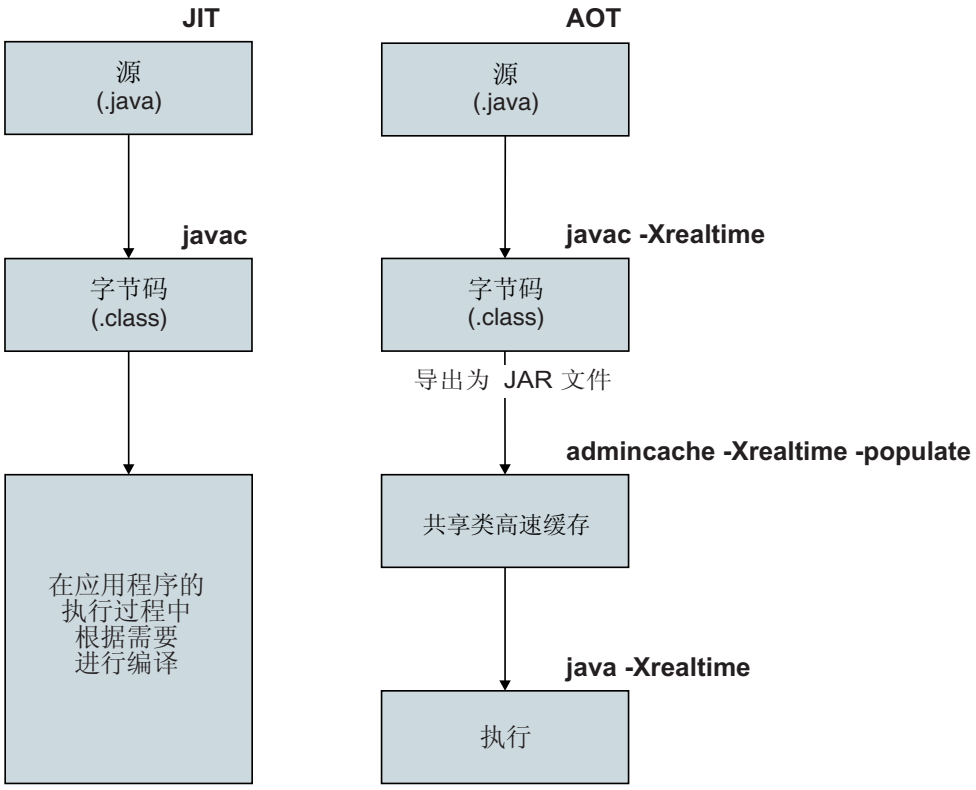

图 2. 比较 JIT 编译器和 AOT 编译器。

线程调度

Linux 调度策略可以与常规 Java 线程以及实时 Java 线程一起使用以调优实时应用程序。

对于 WebSphere Real Time for RT Linux，您可以使用 SCHED_RR 或 SCHED_FIFO 调度策略运行常规 Java 线程。使用 SCHED_RR 或 SCHED_FIFO 策略可以使您更好地控制应用程序，这可改善 Java 线程的实时性能。当使用 SCHED_RR 或 SCHED_FIFO 策略启动 Java 时，JVM 会检测主线程的优先级和策略。JVM 会相应地变更优先级和策略映射。有关变更常规 Java 线程优先级和策略的更多信息，请参阅第 33 页的『线程调度和分派』。

实时 Java 线程的线程调度和分派是 Real Time Specification for Java (RTSJ) 的一部分。可以在第 9 页的『对 RTSJ 的支持』中找到本主题，包括实时 Java 线程的调度策略和优先级处理。

Linux 调度策略包括:

SCHED_OTHER

大多数线程使用的缺省通用时间共享调度策略。必须为这些线程分配优先级零。

SCHED_OTHER 使用时间分片，这意味着每个线程只运行一段有限的时间，该时间过后将允许下一个线程运行。

SCHED_FIFO

只能用于大于零的优先级。当 SCHED_FIFO 线程变为可用时，该线程具有高于任何普通 SCHED_OTHER 线程的优先级。

如果具有较高优先级的 SCHED_FIFO 线程变为可用，那么该线程具有高于具有较低优先级的现有 SCHED_FIFO 线程的优先级。这样，该线程将因其优先级而被保持在队列顶部。

无时间分片。

SCHED_RR

是对 SCHED_FIFO 的增强。差异在于每个线程只允许运行一段有限的时间。如果线程超过该时间，那么将按其优先级返回到列表。

有关这些 Linux 调度策略的更多详细信息，请参阅 **sched_setscheduler** 的联机帮助页。

有关将 Linux 调度策略与 WebSphere Real Time for RT Linux 结合使用的更多信息，请参阅第 33 页的『线程调度和分派』。

对 RTSJ 的支持

WebSphere Real Time for RT Linux 实施了“Real-Time Specification for Java (RTSJ)”。

WebSphere Real Time for RT Linux V3.0 已认证为符合 RTSJ Technology Compatibility Kit V3.1.0 FCS 的 RTSJ 1.0.2 标准，并且符合 Java Compatibility Kit (JCK) for V7.0。

实时线程调度和分派

实时 Java 线程的线程调度和分派是 Real Time Specification for Java 的一部分。调度策略 SCHED_FIFO 用于划分实时 Java 线程（使用 Linux 操作系统优先级 11 - 89）的优先级。

有关 Linux 调度策略的信息位于第 33 页的『线程调度和分派』中。

可调度对象及其参数

有两种主要类型的实时可调度对象：实时线程和异步事件处理程序。

这些可调度对象具有以下与其关联的参数:

SchedulingParameters

PriorityParameters 根据优先级调度实时可调度对象。

ReleaseParameters

- **PeriodicParameters** 描述实时可调度对象的定期释放。定期实时线程是定期进行释放的线程。

- **AperiodicParameters** 描述实时可调度对象的释放。不定期实时线程是不定期释放的线程。

MemoryParameters
描述实时可调度对象的内存分配约束。

ProcessingGroupParameters
在 WebSphere Real Time for RT Linux 中不受支持。

优先级调度程序

在 WebSphere Real Time for RT Linux 中，调度程序是优先级调度程序。如其名称暗示的那样，它根据活动优先级来管理可调度对象的运行。

调度程序维护可调度对象的列表，并确定何时释放每个对象以在 CPU 中运行。调度程序必须遵守与每个可调度对象关联的各种参数。针对此目的，提供了方法 addToFeasibility、isFeasible 和 removeFromFeasibility。

优先级和策略

常规 Java 线程（即作为 `java.lang.Thread` 对象分配的线程）可以使用调度策略 SCHED_OTHER、SCHED_RR 或 SCHED_FIFO。实时线程（即作为 `java.lang.RealtimeThread` 分配的线程）和异步事件处理程序使用 SCHED_FIFO 调度策略。

常规 Java 线程使用缺省调度策略 SCHED_OTHER，除非 JVM 由使用策略 SCHED_RR 或 SCHED_FIFO 的线程启动。使用策略 SCHED_OTHER 的常规 Java 线程的操作系统线程优先级设置为 0。使用策略 SCHED_RR 或 SCHED_FIFO 的常规 Java 线程继承启动 JVM 的线程的优先级。有关常规 Java 线程的优先级和策略的更多信息，请参阅第 34 页的『常规 Java 线程优先级和策略』。

对于实时线程，SCHED_FIFO 策略无时间分片，并且支持从 1（最低优先级）到 99（最高优先级）的 99 级优先级。该 WebSphere Real Time for RT Linux 实施支持 28 个用户优先级（范围为 11 - 38），所以:

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

返回 11，而:

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

返回 38。

OS 操作系统 81 - 89 由 IBM JVM 用以分派工作程序线程。这些线程全部旨在返回休眠状态前执行少量工作。线程如下所述:

- Metronome 垃圾回收器警报线程以优先级 89 运行。该线程定期运行，并分派 GC 工作单元。
- 两个用以处理异步信号的异步信号线程，一个为优先级 88 的非堆实时 (NHRT) 线程，另一个优先级为 87。
- 两个用以分派计时器事件的计时器线程，一个为用于非堆计时器的非堆实时线程，优先级为 85，另一个优先级为 83。
- 异步事件处理程序线程，分派为运行异步事件处理程序，并且线程在运行异步事件处理程序时使用该处理程序的优先级。系统以两个非堆实时处理程序线程启动，其中一个线程优先级为 85，另一个为 83。

- 异步信号非堆实时线程（优先级为 88）处理堆转储、核心转储和 javacore 转储的请求。它在创建转储文件时会临时将优先级升至 89。

Metronome GC 跟踪线程以 OS 优先级 12 运行，JIT 样本线程（以 Java 方法为样本进行编译）以 OS 优先级 13 运行。

JIT 编译线程（不同于 JIT 采样器线程）使用 SCHED_OTHER 策略以 OS 优先级 0 运行。

如果指定了 **-Xnojit** 或 **-Xint**，那么 JIT 编译和 JIT 采样器线程均将禁用。

Metronome 垃圾回收器和完成器优先级在每次回收循环前会不断更改，使其优先级高于最高优先级堆分配线程。必须确保堆分配线程的优先级低于 NoHeapRealtimeThreads 的优先级。

堆分配线程是任意非 NHRT 用户线程，该线程在监视器上未处于睡眠或阻塞状态。不会将在 JNI 接口外运行本机代码的用户线程视作堆分配。如果堆分配线程唤醒、在监视器上不再阻塞或离开 JNI 时正在执行垃圾回收，那么会强制其等待直至垃圾回收完成，再开始继续。

OS 优先级 81 为从堆中分配的内部 JVM 线程保留。如果内部 JVM 线程为 OS 优先级 81，那么垃圾回收器以 OS 优先级 82 运行。当唯一的堆分配用户线程不是实时线程时，GC 优先级以 OS 优先级 11 运行。否则，GC 以高于最高优先级堆分配用户线程的 OS 优先级运行。

GC 优先级在回收循环前进行调整。

优先级映射和继承

每个 Java 优先级均映射到关联的操作系统基本优先级，并且每个操作系统优先级均与调度策略相关联。 WebSphere Real Time for RT Linux Linux 操作系统调度策略是 SCHED_OTHER、SCHED_RR 和 SCHED_FIFO。

实时 Java 线程使用策略 SCHED_FIFO，而常规 Java 线程使用启动 JVM 的线程的策略。常规 Java 线程的缺省调度策略是 SCHED_OTHER，但是可以使用如 **chrt** 的实用程序来设置策略 SCHED_RR 或 SCHED_FIFO。有关线程优先级和策略的更多信息，请参阅第 33 页的『线程调度和分派』。

下表显示如何将 Java 优先级映射到本机操作系统优先级。保留部分 Java 优先级以供 JVM 使用，而无对应 Java 优先级的部分本机优先级也供 JVM 使用。

注:

- 优先级 1-10 供常规 Java 线程使用。
  - 对于策略 SCHED_OTHER，Java 优先级 1-10 映射到操作系统优先级 0。
  - 对于策略 SCHED_FIFO 或 SCHED_RR，Java 优先级 1-10 继承启动 JVM 的线程的优先级。
- 优先级 11 及以上由实时线程和非堆实时线程使用
- 可调度对象始终以其活动优先级运行。活动优先级最初是可调度对象的基本优先级，但是活动优先级可通过优先级继承进行临时提升。可调度对象的基本优先级可在运行时进行更改。

用户基本优先级:

```
Java 优先级 1-10: SCHED_OTHER, OS 优先级 0

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

内部基本优先级:

```
内部 Java 优先级 39: SCHED_FIFO, OS 优先级 81
内部 Java 优先级 40: SCHED_FIFO, OS 优先级 83
内部 Java 优先级 41: SCHED_FIFO, OS 优先级 84
内部 Java 优先级 42: SCHED_FIFO, OS 优先级 85
内部 Java 优先级 43: SCHED_FIFO, OS 优先级 86
内部 Java 优先级 44: SCHED_FIFO, OS 优先级 87
内部 Java 优先级 45: SCHED_FIFO, OS 优先级 88
OS 优先级 11、12、13
OS 优先级偶数 26、28、30... 82
OS 优先级 89
```

另请访问 http://www.rtsj.org/specjavadoc/book_index.html 中的“同步”部分。

优先级继承:

由于线程持有更高优先级线程所需的锁，因此可临时提升线程的活动优先级。这些锁可能是内部 JVM 锁或与同步方法或同步块相关联的用户级别的监控器。因此常规 Java 线程的优先级可临时具有实时优先级，直至线程释放锁时为止。

优先级继承的结果之一就是 SCHED_OTHER 线程的线程策略临时更改为 SCHED_FIFO。

有关基本和活动优先级的更多信息，请参阅 RTSJ 规范中的“同步”部分。

内存管理

垃圾回收内存堆一直以来都被视为实时编程的障碍，因为垃圾回收会带来不可预测的行为。IBM WebSphere Real Time for RT Linux 中的 Metronome 垃圾回收器可以提

供较高的确定性垃圾回收性能。同时，Real-Time Specification for Java (RTSJ) 为已回收垃圾的堆之外的对象提供对内存模型的若干扩展，以便 Java 程序员能够显式管理短生命周期对象和长生命周期对象。

内存区域

RTSJ 引入了可用于对象分配的内存区域概念。某些内存区域存在于堆外部，并对系统和垃圾回收器能对于对象执行的操作加以限制。例如，某些内存区域中的对象永远不会被进行垃圾回收，但垃圾回收器可扫描这些内存区域以查找是否存在对堆中任何对象的引用以保持堆的完整性。

内存管理有三种基本类型:

- 堆内存是传统的 Java 堆，但由 Metronome 垃圾回收器管理。
- 设置了作用域的内存必须由应用程序来具体请求，并只能供实时线程使用，包括非堆实时线程和非堆异步事件处理程序。
- 永久内存表示一个内存区域，其中包含可由任何可调度对象所引用的对象，具体包括非堆实时线程和非堆异步事件处理程序。即使应用程序不使用永久内存，类加载和静态初始化也会予以使用。

永久或设置了作用域的内存可以被指派为使用物理内存（由具有特定特征（例如访问速度显著更快）的内存区域构成）。一般而言，物理内存不经常使用，并且不太可能影响标准 JVM 用户。

堆内存

内存大小受 **-Xmx** 控制，但请记得不要设置初始堆大小 (**-Xms**) 或将其设置为等于最大堆大小 **-Xmx**，因为在实时情况下，堆永远不会从初始堆大小扩展到最大堆大小。当达到最大堆大小而没有可用空间时，将产生 OutOf MemoryError。一般而言，实时 JVM 耗用的堆内存多于传统 JVM，因为支持确定性的回收要求对象的组织方式不同，从而导致更多的堆分段。此外，数组会被拆分成多个片段，而其中每个片段都具有一个头。这取决于大小对象之间的比率和数组使用量，但很可能会发现应用程序还需要 20% 的堆空间。

Metronome 垃圾回收器 类似于主流 JVM 中存在的“大多并发”回收器，因为它在应用程序运行期间回收垃圾。在理想情况下，回收周期会在应用程序内存耗尽之前完成，但某些具有非常高分配速度的应用程序能够以高于 Metronome 垃圾回收器回收速度的速度来进行分配。有多种不同的详细控制都会影响回收速度，但有一种控制会强制 Metronome 在最终抛出 OutOf MemoryError 之前还原为传统的 stop-the-world 垃圾回收器。运行时参数是 **-Xgc:synchronousGCOnOOM**，而与其相对的参数是 **-Xgc:nosynchronousGCOnOOM**。缺省值为 **-Xgc:synchronousGCOnOOM**。

设置了作用域的内存

RTSJ 引入了设置了作用域的内存这一概念。它可供具有明确定义的生命周期的对象使用。作用域可显式输入，或者可以附加到可调度对象（实时线程或异步事件处理程序，在运行对象的 run() 方法之前有效进入作用域）。每个作用域都有一个引用计数，当该引用计数达到零时，该作用域中驻留的对象便可以关闭（或最终完成），而与该作用域关联的内存将得以释放。在最终化处理完成之前会阻止复用该作用域。

设置了作用域的内存可以分为两种类型：VTMemory 和 LTMemory。设置了作用域的这些内存类型因从区域中分配对象所需的时间不同而不同。当内存区域中的内存消耗小于内存区域的初始大小时，LTMemory 可保证线性时间分配。VTMemory 不提供此类保证。

作用域可以进行嵌套。进入嵌套的作用域后，将从与新作用域关联的内存中获取所有后续分配。嵌套的作用域完成后，将复原上一作用域，并将再次从该作用域获取后续分配。

由于设置了作用域的对象的生命周期，有必要通过受限的一组分配规则来限制对设置了作用域的对象的引用。对设置了作用域的对象的引用不能分配给外层作用域中的变量，也不能分配给堆或永久区域中的对象的字段。对设置了作用域的对象的引用只能分配到同一作用域中或内层作用域中。虚拟机会检测错误的分配尝试，并在这些尝试发生时抛出 IllegalAssignmentError 异常。选择设置了作用域的内存类型所带来的灵活性允许应用程序使用具有某些特征的内存区域，这些特征适合于代码的特定语法定义区域。

此内存区域的大小必须在区域构造期间进行定义，而命令行参数 **-Xgc:scopedMemoryMaximumSize** 控制最大值。缺省值是 8 MB，足够满足大多数用途。

永久内存

永久内存是在应用程序中所有可调度对象和线程之间共享的内存资源。分配到永久内存中的对象始终可用于非堆线程和异步事件处理程序，并且不受由垃圾回收所导致的延迟的影响。程序终止时系统会释放对象。

大小受 **-Xgc:immortalMemorySize** 控制；例如，**-Xgc:immortalMemorySize=20m** 设置 20 MB。缺省值是 16 MB，这通常已足够，除非您要执行大量的类加载。类加载是大多数 OutOfMemoryError 异常的可能原因。

估算内存需求

如何获取要分配足够内存而所需的信息

一种合理的方法是确定保存预期对象所需的内存，并外加合理的安全储备。对应用程序的分析可帮助确定所需对象的数量和性质，尽管对象所需的实际大小可能会因不同系统而异。使用 SizeEstimator 类会将实际对象大小考虑在内，从而提供更具可移植性的信息。

SizeEstimator 类

SizeEstimator 类提供关于存储对象所需内存量的指导信息。此估算指示应该为对象自身分配的最小内存空间，而不考虑对象可能需要（例如，在其构建期间）的任何其他资源的内存需求。

有关该类的详细信息，请访问 http://www.rtsj.org/specjavadoc/book_index.html。

使用内存

Java 线程、实时线程和非堆实时线程的比较。

Real-Time Specification for Java (RTSJ) 添加两个类以支持实时线程：RealtimeThread 类和 NoHeapRealtimeThread 类。

- 实时线程和非堆实时线程都是可调度对象。作为可调度对象，它们具有以下参数：发布、调度、内存和处理组。
- 实时线程可以访问堆内存、设置了作用域的内存和永久内存中的对象。非堆实时线程只能访问设置了作用域的内存区域和永久内存区域。
- 非堆实时线程只能访问设置了作用域的和永久内存区域。
- 非堆实时线程需要高于其他实时线程的优先级。如果它们的优先级低于其他实时线程，那么它们会失去在不受垃圾回收器干预的情况下运行的优势。

注：优先级高于其他实时线程的非堆实时线程不会被垃圾回收中断。

表 3. 实时和非堆实时线程的内存访问

线程	永久内存	设置了作用域的内存	堆内存
普通线程	✓	✗	✓
实时线程	✓	✓	✓
非堆实时线程	✓	✓	✗

内存区域的类型

永久内存

永久内存不受垃圾回收的影响。将空间分配到永久内存中之后，此空间在应用程序退出之前不能被回收。

- 由于永久内存的性质，您可能希望找到复用内存的方法。一种可能的方法是创建可复用对象的池。使用设置了作用域的内存是另一种方法。
- 永久内存中的对象不能引用设置了作用域的内存中的任何内容。如果向永久内存中的对象的字段分配设置了作用域的内存中的对象，那么将抛出 IllegalAssignmentError 异常。

设置了作用域的内存

设置了作用域的内存可以用作可调度对象的初始内存区域或可以让可调度对象进入。不再被引用时，将清空此区域中的所有对象。设置了作用域的内存区域中运行的可调度对象从该区域中执行其所有对象分配。当设置了作用域的内存区域未被使用时，其中的对象会最终完成而内存会被回收，从而对作用域进行准备以供复用。当设置了作用域的内存区域不再可用于任何可调度对象时，将回收内存以作他用。

ScopedMemory 实例所描述的内存区域不存在于 Java 堆中，并且不受垃圾回收影响。可以安全地将 ScopedMemory 对象用作与 NoHeapRealtimeThread 关联的初始内存区域，或者通过在 NoHeapRealtimeThread 内使用 ScopedMemory.enter 方法进入此内存区域。

物理内存

如果内存自身的特征很重要，那么请使用物理内存；例如，不可分页的或非易失性的内存。

线性时间分配方案 (LTMemory)

LTMemory 表示一种内存区域，系统保证在该内存区域中的内存消耗小于其初始大小时，该内存区域具有线性时间分配。用于分配的运行时可以在内存消耗介于该区域的初始大小与最大大小之间时有所变化。此外，底层系统无需保证介于初始值与最大值之间的内存始终可用。

可变时间分配方案 (VTMemory)

VTMemory 类似于 LTMemory，不同之处在于 **VTMemory** 区域中的分配所用的运行时间无需在线性时间内完成。

堆内存 堆内存中的对象不能引用设置了作用域的内存中的任何内容。如果向堆内存中的对象的字段分配设置了作用域的内存中的对象，那么将抛出 IllegalAssignmentError 异常。

同步和资源共享

在实时系统中，三个（或三个以上）彼此同步的线程以不同优先级运行时，有时会出现称为优先级反转的情况，此时具有较高优先级的线程运行会被具有较低优先级的线程阻塞一段时间。WebSphere Real Time for RT Linux 使用称为优先级继承的方案来避免出现此情况。

如果具有较高优先级的任务运行被较低优先级任务阻塞，那么较低优先级任务的优先级会临时提升以匹配较高优先级直至较高优先级的任务不再被阻塞。

定期和不定期参数

实时线程有大量释放参数，用于确定可调度对象的释放周期。定期和不定期参数是释放参数的示例。

定期参数

此类适用于那些定期释放的可调度对象。

AbsoluteTime

以毫秒和纳秒表示。

RelativeTime

该值是指定事件的时间长度，以毫秒和纳秒表示。例如，在事件启动及完成时测量绝对时间。随后根据两个测量值之间的差值，计算相对时间。

不定期参数

此类适用于那些不定期释放的可调度对象。由于第二个不定期事件可能在第一个不定期事件完成前就发生了，因此您可以定义未完成请求队列的长度。

异步事件处理

异步事件处理程序对线程外部发生的事件作出反应；例如，来自应用程序界面的输入。在实时系统中，这些事件必须在您为应用程序设置的截止期限内作出反应。

异步事件可与系统中断和 POSIX 信号相关联，并且异步事件可链接到计时器。

和实时线程类似，异步事件处理程序拥有多个与其关联的参数。有关这些参数的列表，请参阅第 9 页的『可调度对象及其参数』。

信号处理程序

POSIXSignalHandler 支持信号 SIGQUIT、SIGTERM 和 SIGABRT。 SIGQUIT 的缺省行为导致生成 Javadump。 Javadump 的生成不会妨碍运行程序的操作，除了 CPU 时间和文件读写。 Javadump 的生成会中断程序，直至 Javadump 完成；在生成 Javadump 期间无法预测应用程序性能。

要在发生故障后禁止所有内核和 Javadump 生成，请使用 **-Xdump:none**。

要仅禁止有关 SIGQUIT 信号的系统转储和 Javadump 生成，请指定 **-Xdump:java:none -Xdump:java:events=gpf+abort**。

以下信号可通过 POSIXSignalHandler 机制连接到 异步事件处理程序 (AEH)（信号描述如 /usr/include/bits/signum.h 中定义所示）：

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

目前不支持其他信号。先前列出的所有信号均为异步信号，并且不可能支持连接到同步信号（如 SIGILL 和 SIGSEGV），原因是它们指示应用程序或 JVM 代码故障，而不是外部生成的事件。

注: 缺省情况下 SIGQUIT 导致 Java 应用程序在 JVM 接收后生成转储（例如 Javadump）。虽然它还传递给任何已连接的 AEH，但该传递可能导致出现混淆或非预期的行为，可以在 Java 命令行上使用 **-Xdump:none:events=user** 选项来禁用该传递。

所需文档

WebSphere Real Time for RT Linux 实施了“Real-Time Specification for Java (RTSJ)”。

WebSphere Real Time for RT Linux V2.0 已认证为符合 RTSJ Technology Compatibility Kit V3.0.13 FCS 的 RTSJ 1.0.2 标准，并且符合 Java Compatibility Kit (JCK) for V6.0。

受支持的功能

支持以下功能:

- 对堆分配的分配率强制限制，目的是限制可调度对象在堆中创建对象的比率。

不受支持的功能

以下功能不受支持:

- 优先级上限模拟协议（Priority Ceiling Emulation Protocol）。例如，不允许将 PriorityCeilingEmulation 用作监视器控制策略。
- 原子访问支持（符合规范时除外）。
- 基本优先级调度程序之外的任何其他调度程序对应用程序都不可用。
- 成本执行。

Real-Time Specification for Java 所需的文档

Real-Time Specification for Java (RTSJ) 的*所需文档*部分在本部分中引述。与 RTSJ 标准实施间的所有不同均有注释。

1. **可行性测试算法为缺省值。**

   “如果可行性测试算法不是缺省值，记录可行性测试算法。”

2. **只有基本优先级调度程序对应用程序可用。**

   “如果非基本优先级调度程序的调度程序对应用程序可用，记录调度程序行为及其与其他调度程序的交互（如“调度”章节中详述）。还需记录调度程序中组成可调度对象的类列表，除非该列表与基本调度程序的可调度对象列表完全相同。”

3. **被较高优先级可调度对象抢占的可调度对象将放置在其优先级的队列前端。**

   “被较高优先级可调度对象抢占的可调度对象放置在其活动优先级队列中，其位置由实施确定。如果抢占的可调度对象未放置在相应队列的前端，那么实施必须记录此类放置的算法。本规范的未来版本可能要求放置在队列前端。”

4. **不支持成本执行。**

   “如果实施支持成本执行，那么需要实施记录当前 CPU 耗用的更新粒度。”

5. **支持简单的顺序映射。**

   “除非是连续字节的简单顺序映射，否则任何物理内存类型过滤器实施的内存映射均必须进行记录。”

6. **没有随 WebSphere Real Time for RT Linux 提供用于 Metronome 垃圾回收器的子类。**

   “实施必须完全记录任何 GarbageCollector 子类的行为。”

7. **没有随 WebSphere Real Time for RT Linux 提供任何 MonitorControl 子类。**

   “提供任何未在该规范中详述的 MonitorControl 子类的实施必须记录其影响，尤其是关于优先级反转控制以及哪些调度程序（如果有）无法支持新策略。”

8. **持有较高优先级可调度对象所需监视器的可调度对象将其优先级提升至较高优先级，直至其释放监视器时为止。如果此时可调度对象不能再运行（即要完成一个更高优先级的工作），那么当在早于 Linux Enterprise Real Time 10 SP2 Update Kernel V2.6.22.19-0.16 和 Red Hat Enterprise Linux 5.1 MRG 2.6.24.7-73 Errata 1 的内核上运行时，会将该对象置于最初（未提升）优先级队列的末端。当内核等于或高于这些版本时，会将可调度对象置于队列前端。**

   “如果由于优先级反转避免算法导致失去“提升的”优先级，可调度对象未置于其新队列的前端，那么实施必须记录排队行为。”

9. **基本调度程序是随 WebSphere Real Time for RT Linux 提供的唯一调度程序。**

   “对于除基本调度程序外的任何可用调度程序，实施必须记录同步语义与为缺省 PriorityInheritance 实例定义的规则之间的不同之处（如果有）。必须提供有关具有优先级继承（如果受支持，遵守优先级上限模拟协议）的新调度程序行为的文档，优先级继承相当于“同步”章节中基本优先级调度程序的语义。”

10. **假定不存在有竞争的可调度对象，或具有相同或更高优先级的系统活动，而且垃圾回收也不会产生干扰，那么从触发某事件到调度关联绑定事件处理程序的最坏情况平均时间为 40µs（不超过 100µs）。如果驱动触发方法的可调度对象、AsyncEvent 对象或处理程序引用堆，那么垃圾回收可能产生的影响如 (A) 中所述。假定代码正在解释中，且事件中已配置单个的处理程序（绑定的处理程序）。**

"针对某些引用体系结构，必须记录自发生绑定而触发 AsyncEvent 到释放关联 AsyncEventHandler 间（假定不存在可运行的具有更高优先级的可调度对象）最坏情况的响应时间间隔。"

11. **假定不存在有竞争的可调度对象，或具有相同或更高优先级的系统活动，且垃圾回收也不会产生干扰，那么在启用 ATC 的线程中触发 AsynchronouslyInterruptedException 到首次传递该异常间的最坏情况平均时间间隔为 35µs（不超过 160µs）。此种情况下启用 ATC 意为线程正在某个区域中以启用 AI 的方法执行，该区域为非 ATC 延迟，且这些条件为 true 直到传递异常为止。垃圾回收可能产生的影响如（A）中所述。如果目标线程位于本机代码中，那么延迟可能一直持续。假定正在解释代码。**

"针对某些引用体系结构，必须记录自启用 ATC 的线程中触发 AsynchronouslyInterruptedException 到首次传递该异常间的时间间隔（假定不存在可运行的具有更高优先级的可调度对象）。"

12. **无适用情况，请参阅响应 4。**

"如果支持成本执行，并且实施将作用域内存中对象的运行完成器的成本分配给任意可调度对象（不包括离开作用域导致作用域引用计数降为 0 的可调度对象），那么应记录分配成本的规则。"

13. **未对 RealtimeSecurity 的标准实施进行更改。**

"如果 RealtimeSecurity 实施比所需实施更受限制，或拥有运行时配置选项，那么应记录这些特征。"

14. **作用域内存区域中的对象完成器将由最后一个引用该区域的线程运行，即当线程将引用计数从 1 减少至 0 时运行。任何与运行完成器相关联的成本将分配给该线程。**

"在重新进入作用域前，并且从调用该作用域的 getReferenceCount() 返回前，实施可运行该作用域内存中对象的完成器。但是必须记录运行这些完成器的时间"。

15. **精度不可设置。**

"对于每个受支持的时钟，文档必须指定是否可设置精度，如果可设置，那么文档必须指明受支持的值。"

16. **除实时时钟外，未随 WebSphere Real Time for RT Linux 提供任何其他时钟。**

"如果实施包含任何非所需实时时钟的时钟，其文档必须指明这些时钟可在何种上下文中使用"。

注:

A 用于测试的引用体系结构为 LS20，4 路，2 GHz，1 MB 高速缓存，4 GB 内存。

B 垃圾回收可导致与堆关联的线程中任一点出现延迟。回收器可以两种基本方式之一运行，这两种方式用于控制发生堆内存耗尽时采取的行为。如果回收器设置为在这些情况下立即抛出 OutOfMemoryError，那么最坏情况的垃圾回收延迟一般小于 1 ms。目前，在某些情况下，延迟时间可能更高；例如，存在多个线程带有深层嵌套堆栈或大量的大型作用域的情况。如果回收器设置为在抛出 OutOfMemoryError 前执行同步 GC，那么可能的回收延迟与堆中的活动对象数以及其他内存区域中的对象数相关。在这些情况中，由于典型堆大小的延迟可以持续许多秒，因此可将其视为无限延迟。

第 3 章 规划

在安装 WebSphere Real Time for RT Linux 之前请先阅读本部分。

- 『迁移』
-
- 『硬件和软件先决条件』
- 第 22 页的『注意事项』

迁移

WebSphere Real Time for RT Linux 在针对实时应用程序修改的 Linux 环境中运行。您可以在实时环境中使用标准 Java 应用程序。或者，您也可以修改应用程序以利用 WebSphere Real Time 的功能。

系统迁移

请按照 Linux 支持团队提供的指示信息进行操作。

硬件和软件先决条件

使用该列表来查看 WebSphere Real Time for RT Linux 支持的硬件、操作系统和 Java 环境。

硬件

WebSphere Real Time for RT Linux 认可的硬件配置是以下系统的多处理器变体:

- IBM BladeCenter® LS20（型号 8850-76U、8850-55U、7971 和 7972）
- IBM eServer™ xSeries 326m（型号 7969-65U、7969-85U、7984-52U 和 7984-6AU）
- IBM BladeCenter LS21（型号 7971-6AU）
- IBM BladeCenter HS21 XM 双核或四核（型号 7995）

为保持得到 WebSphere Real Time for RT Linux 的认可，具有超线程的 IBM 系统不得启用超线程。

此外，WebSphere Real Time for RT Linux 在运行受支持操作系统并具有以下特征的硬件上受支持:

- 至少 512 MB 物理内存。
- 最低 Intel Pentium 4、AMD Opteron 或 Intel Atom 处理器。

对于非认可硬件配置的系统，IBM 不做任何性能保证。 第 85 页的第 7 章，『性能』详细说明了经认可硬件配置的性能注意事项。

在支持超线程的系统上，请确保未启用超线程以避免在使用 WebSphere Real Time for RT Linux 时产生不良性能影响。

操作系统

- Red Hat Enterprise Messaging, Realtime, Grid (MRG) 1.3 for Red Hat Enterprise Linux 5 Update 5。
- Red Hat Enterprise MRG 2.2 for Red Hat Enterprise Linux (RHEL) 6 Update 5。请参阅注释。
- SUSE Linux Enterprise Real Time (SLERT) 10。
- SLERT 11 service pack 1（含补丁级别 2.6.33.18）。

注: 在高工作负载的情况下，当在 IBM WebSphere Real Time V3 for Real Time Linux on MRG 2.2（带 RHEL 6 Update 3）下运行应用程序时，要注意间歇出现的问题。此问题在 RHEL 6 Update 5（这是推荐的基本级别）中得到了解决。有关更多信息，请参阅http://www.ibm.com/support/docview.wss?uid=swg21624408。

请参阅第 25 页的『安装实时 Linux 环境』。

注意事项

使用 WebSphere Real Time for RT Linux 时有许多因素需要注意。

- 可能的情况下，请勿在同一个系统上运行多个实时 JVM。原因是这样做会导致有多个垃圾回收器。每个 JVM 不知道其他 JVM 的内存区域。影响之一是在 JVM 之间无法协调 GC 循环和暂停时间，即意为某个 JVM 可能对其他 JVM 的 GC 性能产生不利影响。 如果您必须使用多个 JVM，请确保使用 **taskset** 命令将每个 JVM 绑定到处理器的特定子集。
- 不能对通过提前编译器进行预编译的代码使用 **-Xdebug** 选项和 **-Xnojit** 选项。因为 **-Xdebug** 会以与 AOT 编译器不同的方式来编译代码，所以不受支持。

  要调试您的代码，请使用经过解释或 JIT 编译的代码。
- 如果使用 com.sun.tools.javac.Main 接口来编译使用 javax.realtime 程序包的 Java 源代码，那么必须确保 `sdk/jre/lib/i386/realtime/jclSC170/realtime.jar` 包含在类路径中。该类型编译的一个常见示例是 Ant 编译。
- 可选 JavaComm 程序包可安装到 WebSphere Real Time for RT Linux 中，通过实时和非实时 JVM 均可对其访问。有关安装和配置的更多信息，请参阅 http://publib.boulder.ibm.com/infocenter/java7sdk/v7r0/topic/com.ibm.java.lnx.70.doc/user/jcommchapter.html。WRT 中的实时 JVM 支持 JavaComm API 与常规 Java 线程结合使用。但是不能保证使用 JavaComm 访问外部设备时的确定性或实时性能。同样，请勿将 JavaComm 与非堆实时线程和实时线程结合使用，需要实时行为时也请勿使用 JavaComm。
- 先前 WebSphere Real Time for RT Linux 发行版用于存储预编译代码和类的共享高速缓存与本发行版的 WebSphere Real Time for RT Linux 使用的高速缓存不兼容。您必须重新生成先前高速缓存的内容。
- 在使用共享类高速缓存时，高速缓存名称不能超出 53 个字符。
- **ps** 命令截断 Java 线程名称。

  ps 命令限于 15 个字符。如果您设置的线程名称多于 15 个字符，该名称由 **ps** 命令截断。
- WebSphere Real Time for RT Linux 不支持 NTLoginModule (NTLM) 认证。

NTLoginModule (NTLM) 用于帮助认证 Windows 服务的访问权限。使用 NTLM 的认证只在 Windows 平台上受支持。这意为 WebSphere Real Time for RT Linux 不支持 NTLM 认证。

第 4 章 安装 WebSphere Real Time for RT Linux

请按照以下步骤来安装产品。

- 『安装文件』
- 『安装实时 Linux 环境』
- 第 26 页的『从 InstallAnywhere 程序包进行安装』
  - 第 27 页的『完成有人照管安装』
  - 第 28 页的『完成无人照管安装』
  - 第 29 页的『已知问题和限制』
- 第 30 页的『设置路径』
- 第 30 页的『设置类路径』
- 第 31 页的『测试安装』
- 第 32 页的『卸载 WebSphere Real Time for RT Linux』

安装文件

您需要这些安装文件。

IBM WebSphere Real Time for RT Linux 通过两种类型的 InstallAnywhere 程序包提供。

可安装的程序包

可安装程序包用于配置系统。例如，程序可能会设置环境变量。

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

归档程序包

这些程序包将文件解压缩到系统，但不会执行任何配置。

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

安装实时 Linux 环境

在安装 WebSphere Real Time for RT Linux 之前，必须安装 64 位版本的实时 Linux。

Red Hat Enterprise Messaging, Realtime, Grid (MRG 1.3) for RHEL 5.5

有关安装 Red Hat Enterprise Linux 5.5 MRG 1.3 的实时组件的更多信息，请参阅 RT-Linux RHEL 5.5 MRG 1.3 的安装指示信息: https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_MRG/1.3/html/Realtime_Installation_Guide/index.html

Red Hat Enterprise MRG 2.2 for RHEL 6.3

有关安装 Red Hat Enterprise MRG 2.2 的实时组件的更多信息，请参阅安装指示信息: https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_MRG/2/html/Realtime_Installation_Guide/index.html

SUSE Linux Enterprise Real Time (SLERT) 10

有关安装 SLERT 10 的更多信息，请访问: http://www.novell.com/products/realtime/eval.html

SUSE Linux Enterprise Real Time (SLERT) 11 service pack 1

您可以从 http://download.novell.com/ 获取 SLERT 11。需要 service pack 1 才能正确用于 WebSphere Real Time for RT Linux。您还必须应用补丁级别 2.6.33.18（可从 Real Time Linux Kernel 5075 下载）。有关 SLERT 11 的更多信息，请访问 https://www.suse.com/products/realtime/。

在使用大量文件描述符来装入不同的类实例时，您可能会看到错误消息 "java.util.zip.ZipException: error in opening zip file"，或者某些其他格式的 IOException，提示无法打开文件。解决方案是使用 **ulimit** 命令来增加文件描述符的规定。要查找已打开文件的当前限制，请使用命令:

```
ulimit -a
```

要允许更多打开文件，请使用命令:

```
ulimit -n 8196
```

从 InstallAnywhere 程序包进行安装

这些程序包提供了一个交互式程序，它指导您完成安装选项。您可以通过图形用户界面或系统控制台运行该程序。

开始之前

您的系统必须具有以下两个共享库:

- GNU C 库 V2.3 (glibc)
- libstdc++.so.5

如果您没有 libstdc++.so.5 共享库，那么在安装时可能会看到 Java 核心转储，并包含以下错误:

```
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
```

如果您在安装可安装的程序包，那么必须在系统上安装 rpm 构建工具，否则安装程序无法在 RPM 数据库中注册此新程序包。要确定是否已安装 rpm 构建工具，请输入以下命令:

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

关于此任务

InstallAnywhere 程序包具有 .bin 文件扩展名。

有两种类型的程序包:

可安装程序包

安装这些程序包也会配置您的系统，例如，通过设置环境变量进行配置。

归档　安装这些程序包会将文件解抽取到系统，但不会执行任何配置。

过程

- 要以交互方式安装程序包，请完成有人照管安装。
- 要在无任何额外用户交互的情况下安装程序包，请完成无人照管安装。 如果要安装许多系统，那么可以选择该选项。
- 安装过程完成后，按照本部分中的配置步骤进行操作，例如设置路径和类路径环境变量。

结果

产品已安装。

注: 请勿中断安装过程，例如通过按 Ctrl+C 来中断。如果中断该过程，那么可能必须重新安装产品。有关更多信息，请参阅第 29 页的『中断的安装』。

如果您在使用可安装程序包，那么可能会看到一些消息，告知您发现了问题。安装归档程序包不会产生任何消息。以下列表中显示了安装可安装程序包时可能会看到的一些消息:

The installer cannot run on your configuration. It will now quit.
如果未授权您的用户标识运行安装过程，那么会出现该错误消息。安装过程因为无法继续，所以会结束。要解决该问题，请使用具有 root 用户权限的用户标识来重新开始安装。

An RPM package is already installed. Uninstall the package before proceeding.
该消息指示 RPM 程序包已安装。安装过程因为无法继续，所以会结束。要解决该问题，请先卸载 RPM 程序包，然后再继续。

完成有人照管安装

以交互方式从 InstallAnywhere 程序包安装产品。

开始之前

在开始安装过程之前，请先确保满足以下条件:

- 如果之前已从 RPM 程序包安装了 WebSphere Real Time for RT Linux，那么必须先卸载该程序包，然后再继续。
- 您必须拥有一个具备 root 用户权限的用户标识。

过程

1. 将安装包文件下载到临时目录。
2. 切换到临时目录。
3. 通过在 shell 提示符处输入 `./package`（其中 *package* 是要安装的程序包的名称）来开始安装过程。
4. 从安装程序窗口中显示的列表中选择语言，然后单击**下一步**。可用语言的列表取决于系统的语言环境设置。
5. 阅读许可协议，并使用滚动条到达许可文本末尾。要继续安装，必须接受许可协议的条款。要接受这些条款，请选择相应单选按钮，然后单击**确定**。

注: 您必须首先阅读到许可文本末尾，然后才能选择用于接受许可协议的单选按钮。

6. 将要求您选择安装的目标目录。如果您不希望安装到缺省目录，请单击**选择**以通过使用浏览器窗口来选择其他目录。选择了安装目录后，请单击**下一步**以继续。
7. 将要求您复查所做选择。要更改所做的选择，请单击**上一步**。如果选择正确，请单击**安装**以继续安装。
8. 安装过程完成后，请单击**完成**以完成安装。

完成无人照管安装

如果您要安装多个系统，并且已知道要使用的安装选项，那么可能希望使用无人照管安装过程。您通过使用有人照管安装过程安装一次，然后使用生成的响应文件来完成以后的安装而无需任何额外的用户交互。

过程

1. 通过完成有人照管安装来创建响应文件。使用以下选项之一:
   - 使用 GUI 并指定安装程序创建响应文件。该响应文件名为 installer.properties，并创建于安装目录中。
   - 使用命令行并向有人照管安装命令附加 -r 选项，同时指定响应文件的完整路径。例如:

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

   示例响应文件内容:

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

   在该示例中，*/my_directory* 是您为安装而选择的目标安装目录。
2. 可选: 如有需要，可编辑响应文件以更改选项。

   注: 归档程序包具有以下已知问题: 即使您在响应文件中更改了目录，使用响应文件的安装仍会使用缺省目录。如果缺省目录中存在先前安装，那么会予以覆盖。

   如果您要创建多个响应文件，并且每个都具有不同的安装选项，请为每个响应文件都指定格式为 *myfile*.properties 的唯一名称。
3. 可选: 生成日志文件。 因为要进行静默安装，所以安装过程结束时将不会显示任何状态消息。要生成包含安装状态的日志文件，请完成以下步骤:
   a. 通过使用以下命令来设置所需的系统属性。

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

   b. 设置以下环境变量以将日志输出发送到控制台。

      ```
      export LAX_DEBUG=1
      ```
4. 通过 **-i** 静默选项以及用于指定响应文件的 **-f** 选项运行程序包安装程序，以开始无人照管安装。 例如:

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

您可以使用属性文件的标准路径或相对路径。在这些示例中，字符串 1>console.txt 2>&1 将安装过程信息从 stderr 和 stdout 流重定向到当前目录中的 console.txt 日志文件。如果您认为安装存在问题，请复查该日志文件。

注：如果您的安装目录包含多个响应文件，那么将使用缺省响应文件 installer.properties。

中断的安装

如果在安装期间程序包安装程序意外停止（例如，如果您按了 Ctrl+C），那么安装会中断并且您无法卸载或重新安装产品。如果您尝试进行卸载或重新安装，那么可能会看到消息 Fatal Application Error。

关于此任务

要解决该问题，请删除文件并重新安装，如以下步骤中所述。

过程

1. 删除 /var/.com.zerog.registry.xml 注册表文件。
2. 如果创建了包含所安装内容的目录，请将其删除。例如 opt/IBM/javawrt3/。
3. 重新运行安装程序。

已知问题和限制

InstallAnywhere 程序包具有一些已知问题和限制。

- 如果您的系统中没有 libstdc++.so.5 共享库，那么安装将失败，并产生 Java 核心转储。有关更多信息，请参阅第 26 页的『从 InstallAnywhere 程序包进行安装』。
- 安装包 GUI 不支持 Orca 屏幕朗读程序。您可以使用无人照管安装方式作为 GUI 的替代方式。
- 如果安装后，您输入 *./package* 以再次启动安装程序，那么安装程序将显示以下消息:

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

  如果您按 Enter 键以接受缺省值，那么安装程序将没有响应。请输入一个数字，然后按 Enter 键。
- 如果您安装程序包，然后尝试以其他方式（例如控制台或静默方式）重新安装，那么可能会看到以下错误消息:

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

  如果您已通过使用 GUI 方式安装并要以控制台方式重新运行安装程序，那么不应该看到此消息。如果您看到该错误，并在运行程序以选择卸载选项（仅限可安装程序包），请改用 **./_uninstall/uninstall** 命令，如第 32 页的『卸载 WebSphere Real Time for RT Linux』中所述。

仅限可安装程序包

- 您无法通过使用 InstallAnywhere 程序包来升级现有安装。要升级 WebSphere Real Time for RT Linux，必须首先卸载任何先前版本。

- 即使您使用不同安装目录，也不能同时在同一系统上安装同一版本的 WebSphere Real Time for RT Linux 的两个不同实例。例如，您不能同时在 /previous 目录中有 WebSphere Real Time for RT Linux V3 并在 /current 目录中有 WebSphere Real Time for RT Linux 服务刷新安装。安装程序会检查版本号。如果安装程序发现具有同一版本号的现有程序包，那么会要求您卸载该现有程序包。
- 如果程序包已安装，并且您通过使用 GUI 再次运行程序包安装程序，那么可以选择卸载该程序包。该卸载选项在无人照管方式下不可用。如果您以无人照管方式再次运行程序包安装程序，那么该程序会运行但不会执行任何操作。

仅限归档程序包

- 如果您更改响应文件中的安装目录，然后通过使用该响应文件来运行无人照管安装，那么安装程序将忽略该新安装目录并改用缺省目录。如果缺省目录中存在先前安装，那么会予以覆盖。

设置路径

设置 **PATH** 环境变量后，您可通过在 shell 提示符中输入名称来运行应用程序或程序。

关于此任务

注: 如果按本部分所述更改 **PATH** 环境变量，将覆盖路径中任何现有的 Java 可执行文件。

每次在工具名称前输入路径即可指定到该工具的路径。例如，如果 SDK 安装在 opt/IBM/javawrt3/ 中，您可以在 shell 提示符中输入以下命令来编译名为 *myfile.java* 的文件:

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

要避免每次均需输入完整路径:

1. 在主目录（根据 shell，通常是 **.bashrc**）中编辑 shell 启动文件，并将绝对路径添加到 **PATH** 环境变量；例如:

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

2. 再次登录或运行更新后的 shell 脚本以激活新的 **PATH** 设置。
3. 使用 **javac** 工具来编译文件。例如，要编译文件 *myfile.java*，在 shell 提示符处输入:

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

   PATH 环境变量使 Linux 能够从当前目录中找到可执行文件，如 **javac**、**java** 和 **javadoc** 工具。要显示路径的当前值，请在命令提示符中输入以下命令:

   ```
   echo $PATH
   ```

下一步做什么

请参阅『设置类路径』以确定是否需要设置 CLASSPATH 环境变量。

设置类路径

CLASSPATH 环境变量会告知 SDK 工具（例如 **java**、**javac** 和 **javadoc** 工具）哪里可以找到 Java 类库。

关于此任务

仅当以下条件之一适用时，才可显式设置 **CLASSPATH** 环境变量:

- 您需要使用一个不同的库或类文件（例如您自行开发的），而它又不在当前目录中。
- 您更改了 bin 和 lib 目录的位置并且它们不再具有相同的父目录。
- 您计划开发或运行在同一系统上使用不同运行时环境的应用程序。

要显示 **CLASSPATH** 的当前值，请在 shell 提示符处输入以下命令:

```
echo $CLASSPATH
```

如果开发并运行使用不同运行时环境的应用程序（包括已经单独安装的其他版本），那么必须为每个应用程序显式地设置 **CLASSPATH** 和 **PATH**。如果您同时运行多个应用程序并使用不同运行时环境，那么每个应用程序都必须在它自己的 shell 中运行。

如果一次只运行一个 Java 版本，那么可以使用 shell 脚本在不同的运行时环境间进行切换。

下一步做什么

请参阅『测试安装』以验证是否成功安装。

测试安装

如果安装成功，请使用 **-version** 选项进行检查。

关于此任务

Java 安装包括标准 JVM 和实时 JVM。

过程

通过完成以下步骤来测试您的安装:

1. 要查看标准 JVM 的版本信息，请在 shell 提示符处输入以下命令:

   ```
   java -version
   ```

   如果该命令成功，将返回以下消息:

   ```
   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
   ```

   如果您打算使用标准 JVM 而非实时 JVM，请参阅 IBM User Guides for Java v7 on Linux。

   注: 版本信息正确，但日期可能晚于该示例中的日期。日期字符串格式为: yyyymmdd，后面可能跟有特定于组件的其他信息。
2. 要查看实时 JVM 的版本信息，请在 shell 提示符处输入以下命令:

```
java -Xrealtime -version
```

如果该命令成功，将返回以下消息:

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

注: 版本信息正确，但平台体系结构和日期可能与该示例不同。日期字符串格式为: yyyymmdd，后面可能跟有特定于组件的其他信息。

卸载 WebSphere Real Time for RT Linux

用于移除 WebSphere Real Time for RT Linux 的过程取决于使用的安装类型。

开始之前

对于 InstallAnywhere 可安装程序包，您的用户标识必须具有 root 用户权限。

关于此任务

没有针对 InstallAnywhere 归档程序包的卸载过程。要移除系统中的归档程序包，请删除安装程序包时选择的目标目录。对于 InstallAnywhere 可安装程序包，使用命令或再次运行安装程序来卸载产品，如以下步骤所述。

过程

- 可选: 使用 **uninstall** 命令来手动卸载。
  1. 切换到包含安装的目录。 例如:

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

  2. 输入以下命令来启动卸载过程: `./_uninstall/uninstall`
- 可选: 如果卸载程序不好找，作为替代方法，您可以运行其他有人照看的安装。安装程序会检测产品是否已安装，然后给您选择是否卸载先前安装的机会。

第 5 章 运行 IBM WebSphere Real Time for RT Linux 应用程序

可在运行实时应用程序时帮助您的重要信息。

- 『线程调度和分派』
- 第 36 页的『在 WebSphere Real Time for RT Linux 中使用经过编译的代码』
- 第 52 页的『使用非堆实时线程』
- 第 60 页的『JVM 之间的类数据共享』
- 第 61 页的『使用 Metronome 垃圾回收器』

线程调度和分派

Linux 操作系统支持各种调度策略。缺省通用时间共享策略是 SCHED_OTHER（供大多数线程使用）。SCHED_RR 和 SCHED_FIFO 可供实时应用程序中的线程使用。。

内核决定哪一个是处理器要运行的下一个可运行线程。内核维护可运行线程列表。它会查找具有最高优先级的线程并选择该线程作为下一个要运行的线程。

可使用以下命令列出线程优先级和策略:

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

其中策略:

- TS 是 SCHED_OTHER
- RR 是 SCHED_RR
- FF 是 SCHED_FIFO
- - 未报告任何策略

输出与以下示例相似:

```
  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
```

该输出显示 Java 进程、已生效的调度策略、优先级为“-”（其他）的主线程以及优先级为 11 到 89 的一些实时线程。

要查询当前调度策略，请使用 **sched_getscheduler** 或该示例中显示的 **ps** 命令。

有关进程的更多信息，请参阅第 90 页的『常用调试方法』。

常规 Java 线程优先级和策略

常规 Java 线程（即作为 `java.lang.Thread` 对象分配的线程）使用缺省调度策略 SCHED_OTHER。从 WebSphere Real Time for RT Linux V3 服务刷新 1 开始，您可以使用 SCHED_RR 或 SCHED_FIFO 调度策略运行常规 Java 线程。

缺省情况下，Java 线程使用缺省的 SCHED_OTHER 策略运行。该策略将 Java 线程映射到操作系统优先级 0。

使用 SCHED_RR 或 SCHED_FIFO 策略可以使您更好地控制应用程序，这可改善 Java 线程的实时性能。当使用 SCHED_RR 或 SCHED_FIFO 策略启动 Java 时，JVM 会检测主线程的优先级和策略。JVM 会相应地变更优先级和策略映射。所有 Java 线程以与主线程相同的操作系统优先级运行。虽然 SCHED_RR 或 SCHED_FIFO 可分配优先级 1 - 99，但是可用于 WebSphere Real Time for RT Linux V3 的 SCHED_RR 或 SCHED_FIFO 优先级为 1 - 10。如果将优先级设置为高于 10，那么主线程的优先级降至 10，并基于值 10 应用映射。

在命令行中更改进程的实时调度属性方法之一是使用命令 **chrt**。在以下示例中，主 Java 线程的优先级设为使用 SCHED_FIFO 调度策略，且操作系统优先级为 6。

```
chrt -f 6 java
```

可能需要对系统进行配置以允许更改优先级。请参阅 第 35 页的『配置系统以允许优先级更改』以获取更多信息。

表 4. Java 和操作系统优先级

Java 优先级	线程的 Java 优先级值	操作系统优先级
1	MIN_PRIORITY	6
2		6
3		6
4		6
5	NORM_PRIORITY（缺省值）	6
6		6
7		6
8		6
9		6
10	MAX_PRIORITY	6

所有与主 Java 线程关联的线程均以相同的操作系统优先级运行。Metronome 警报线程及 JVM 内部具有精确实时计时需求的其他线程可以高于常规 Java 线程使用的优先级运行。

如果您运行命令 `chrt -f 11 java`，那么结果与运行 `chrt -f 10 java` 的结果相同。这是由于您无法将 10 以上的优先级应用于 JVM 线程使用的优先级映射，但是启动 JVM 并等待 JVM 终止的线程将始终保持优先级 11。

有关 **chrt** 命令的更多信息，请参阅 http://publib.boulder.ibm.com/infocenter/lnxinfo/v3r0m0/index.jsp?topic=/liaai/realtime/liaairtchrt.htm。

配置系统以允许优先级更改

缺省情况下，Linux 上的非 root 用户无法提高线程或进程的优先级。您可以更改系统配置以允许使用 Linux 的可插入认证模块 (PAM) 的 pam_limits 模块更改优先级。

如果您不能使用 **chrt** 实用程序更改线程或进程的优先级，那么通常会看到以下消息:

```
sched_setscheduler: Operation not permitted
```

在最新的 Linux 内核上，您可以更改系统配置以允许使用 pam_limits 模块更改优先级。该模块允许您在限制配置文件中配置对系统资源的限制。缺省文件是 /etc/security/limits.conf。

/etc/security/limits.conf 文件中的条目具有以下格式:

```
<domain> <type> <item> <value>
```

其中:

<domain> 可以是以下任意一项:

- 系统上可以变更资源限制的用户名。
- 其成员可以变更资源限制的组名，语法为 @group。
- 通配符“*”，用于指示任何用户或组都可以变更资源限制。

<type> 可以是以下任意一项:

- hard，其中硬限制由内核强制实施。
- soft，其中软限制适用，这些限制可在硬限制指定的范围内予以变更。
- 破折号“-”，用于指示硬限制和软限制。

<item> 是:

- 资源。将 rtprio 用于实时优先级。

<value> 是:

- 限制。使用范围 1 - 100 内的值以指示实时优先级设置的最大限制。

例如,

```
* - rtprio 100
```

允许所有用户使用 **chrt** 或其他机制更改实时进程的优先级。

缺省情况下，root 用户可以无限制地提高实时优先级。要对 root 用户应用限制，必须显式指定 root 用户。配置文件中的组和通配符限制不适用于 root 用户。

如果在该文件中指定单独用户限制，那么这些限制的优先级高于组限制。

对 limits.conf 的更改不会立即生效。您必须重新启动受影响的服务或重新引导系统才能使配置更改生效。

要在实时 Linux 系统上启用优先级更改，您可以向 realtime 组添加用户，如 limits.conf 文件中所示。

启动辅助进程

Java 虚拟机 (JVM) API 中的 java.lang.Runtime.exec 方法为您的 Java 应用程序提供在单独进程中执行命令的能力。

通过该方法调用，会创建新的 java.lang.Process 对象。该对象可以用于控制新进程，或获取与其相关的信息。

为达到此目的，exec 方法会创建若干线程。在 IBM WebSphere Real Time for RT Linux 中，过程修改在实时环境中支持更具确定性的行为。

Runtime.exec 调用为每个派生子进程都创建一个“收获器”线程。收获器线程是唯一等待子进程终止的线程。当子进程终止时，收获器线程会记录子进程退出状态。收获器线程会衍生出新的进程，并赋予该进程与最初调用 Runtime.exec 的线程相同的调度参数。

如果衍生的进程是另一个 WebSphere Real Time for RT Linux JVM，并且 Runtime.exec 方法由另一个通过 Linux 实时策略和优先级运行的方法进行了调用，那么新虚拟机的主线程会将其策略和优先级映射到同一 Linux 实时策略和优先级。该 Java 线程优先级介于 1 到 10 之间。

收获器线程还会创建两个侦听新进程的 stdout 和 stderr 流的新线程。这两个新线程是常规的 Java 线程，而非实时 Java 线程。stdout 和 stderr 数据会保存到这两个线程所使用的缓冲区中。这些缓冲区的保留时间会超过所衍生的进程的生命周期。这种持久性确保衍生进程所持有的资源在该进程终止时立即被清除。

如果您希望 stdout 和 stderr 阅读器线程以 Linux 实时优先级运行，请通过 Linux SCHED_FIFO 或 SCHED_RR 策略和优先级来启动拥有这些线程的原始 JVM。效果是将所有常规线程映射到 Linux 操作系统中的实时策略和高至 10 的优先级。

实时 Java 线程优先级和策略

实时线程（即作为 java.realtime.RealtimeThread 分配的线程）和异步事件处理程序使用 SCHED_FIFO 调度策略。

实时 Java 线程的线程调度和分派是 Real Time Specification for Java (RTSJ) 的一部分。本主题（包括实时 Java 线程的调度策略和优先级处理）在第 9 页的『对 RTSJ 的支持』部分中进行了讨论。

在 WebSphere Real Time for RT Linux 中使用经过编译的代码

IBM WebSphere Real Time for RT Linux 支持多种代码编译的模型，提供了不同级别的代码性能和确定性。

经过解释的操作

这是最简单的代码编译模型。解释器运行 Java 应用程序，但完全不使用任何代码编译。解释器会呈现好的确定性，但提供的性能非常低，因此请避免在生产系统中使用这种操作方式。

要使用经过解释的操作，请在 Java 命令行中指定 **-Xint** 选项。

低优先级即时 (JIT) 编译

WebSphere Real Time for RT Linux 中的缺省编译模式使用即时编译器，它在应用程序运行时编译 Java 应用程序的重要方法。在这种方式下，JIT 编译器的工作方式与 JIT 编译器在非实时 JVM 中的操作相似。所不同的是 WebSphere Real Time for RT Linux JIT 编译器运行的优先级比任何实时线程都要低。较低的优先级意味着，当应用程序不需要执行实时任务时，JIT 编译器才会使用系统资源。带来的效果是 JIT 编译器不会明显影响实时任务的性能。

JIT 编译器为与编译相关的活动使用两个线程: 编译线程和采样器线程。这些线程的运行优先级低于实时任务。编译线程以与应用程序异步的方式运行。这意味着应用程序线程不会在任何时刻等待编译线程完成对方法的编译。采样器线程定期向应用程序线程发送异步消息，以识别当前每个线程上运行的方法。在应用程序线程上处理消息几乎不用花费多少时间。如果由于较高优先级的实时任务，而使得采样线程无法运行，那么不会发送任何消息。使用 JIT 编辑器会对确定性有很小的影响，但这种编译方式为许多用户提供了最佳性能。

要在低优先级下使用 JIT 来运行应用程序，请参阅第 51 页的『启用 JIT』。

提前 (AOT) 预编译的代码

在运行应用程序之前，WebSphere Real Time for RT Linux 会在预编译步骤中将 Java 方法编译为本机代码。在 WebSphere Real Time for RT Linux V2 之前，预编译步骤使用 jxeinajar 工具来编译使用提前编译器的方法，并将结果存储在特殊的 Java 可执行文件中。这些文件可能会收集到绑定 JAR 文件中。在运行应用程序时，绑定 JAR 文件会添加到应用程序类路径中，这样当从 JXE 中装入方法的类时，JVM 可以装入 AOT 代码。利用这种方法，通过在命令行中指定 **-Xnojit** 选项使得 JIT 编译器完全不可用。应用程序可以使用已创建的任何预编译 AOT 代码，并为其他方法使用解释器。这种操作方式提供了高确定性，因为 JIT 编译器不存在，所以不会造成采样线程或上下文切换性能降低。提前编译 Java 代码的困难，同时又要符合 Java 规范，这意味着 AOT 编译的代码的执行速度通常要比 JIT 编译的代码慢，尽管它通常要比解释快得多。

WebSphere Real Time for RT Linux V2 和后续版本使用 IBM Java 6 JVM 中提供的共享类技术，将 AOT 代码存储在共享类高速缓存而不是 JXE 文件中。admincache 工具使您能够查询高速缓存的内容、列出全部现有的高速缓存以及使用类和 AOT 代码填充高速缓存。存储 AOT 编译的代码的优点在于，运行应该程序时不会修改应用程序 JAR 文件，也不需要更改类路径。

根据可用的虚拟地址空间，共享类高速缓存有实际大小限制。这意味着为所有 JAR 文件进行 AOT 编译并不实际。必须有选择性的执行 AOT 编译。

当共享类高速缓存中包含 AOT 代码的应用程序运行时，类方法的 AOT 代码会在类装入 JVM 时自动装入。装入类以安装其方法的 AOT 代码所带来额外开销，使得在应用程序的关键性能部分运行之前，预装入尽可能多的类显得十分重要。

使用 AOT 预编译代码提供了最高级别的确定性和良好的性能。通过指定 **-Xshareclasses** 和 **-Xaot** 选项，您的应用程序在运行时可以使用 AOT 代码。缺省情况下，**-Xaot** 选项是启用的。

要使用 admincache 工具在共享类高速缓存中存储并使用 AOT 代码，请参阅第 39 页的『使用 admincache 工具』。有关从 jxeinajar 迁移到 admincache 的信息可以在 WebSphere Real Time for RT Linux V2 文档中找到。

有关使用 AOT 编译的代码运行应用程序的示例，请参阅第 79 页的『使用 AOT 的同时运行样本应用程序』。

混合方式，结合 AOT 预编译代码和低优先级 JIT 编译

当应用程序运行时，可以同时使用 AOT 和 JIT 编译的代码。这种操作方式可以提供非常好的确定性和良好的性能，并且为频繁运行的方法提供了非常高的性能。这种方式的主要优点在于，AOT 预编译用于确保应用程序中最重要的部分不会在解释器中运行，解释器中运行的速度通常会远逊于 AOT 或 JIT 编译的代码。您不需要预编译每个方法，因为 JIT 编译器可以动态的识别任何频繁

运行的经过解释的方法，而不会明显影响应用程序的性能。向命令行中添加 **-Xshareclasses** 选项时，混合方式是缺省方式。

要使用混合 AOT 和 JIT 编译来运行您的应用程序，请参阅第 79 页的『使用 AOT 的同时运行样本应用程序』

显式管理编译

在启用 JIT 编译器的编译方式中，java.lang.Compiler API 可以用于显式控制 JIT 编译器操作。JIT 编译器编译使用 compileClass() 方法传递的类中的方法。compileClass() 是同步的，因此所提供的方法完成编译之后，它才会返回。通过遍历在应用程序运行时的主要阶段使用的类，应用程序可以在初始化阶段使用 compileClass()。当初始化阶段完成之后，调用 Compiler.disable() 方法来完全禁用编译和采样线程。此技术的主要困难在于，管理在应用程序初始化阶段中要装入和编译的类列表的问题，尤其是在应用程序开发的过程中。

有关管理应用程序中的编译的更多信息，请参阅 IBM Real-Time Class Analysis Tool for Java。

编译命令行选项的概述

运行应用程序时，您可以使用 **-Xjit** 选项来启用 JIT，或者使用 **-Xnojit** 选项来禁用 JIT。**-Xjit** 是缺省方式。

您可以使用 **-Xshareclasses -Xaot** 选项以在运行应用程序时启用 AOT 代码。使用 **-Xnoaot** 选项即禁用 AOT 代码。**-Xaot** 是缺省选项，但如果未指定 **-Xshareclasses** 选项则不会生效，因为 AOT 代码必须存储在共享类高速缓存中。

使用 AOT 编译器

使用这些步骤来预编译您的 Java 代码。该过程描述了 **javac** 命令中的 **-Xrealtime** 选项、**admincache** 工具以及 **java** 命令中的 **-Xrealtime** 与 **-Xnojit** 选项的使用。

关于此任务

使用提前编译器意味着编译是与应用程序运行时分开的。另外，您可以同时编译更多方法，而不仅仅是频繁使用的方法。如以下步骤中所示，您可以编译应用程序中一切，也可以编译单个类。

注: 在使用共享类高速缓存时，高速缓存的名称不能超出 53 个字符。

过程

1. 在 shell 提示符处输入:

   ```
   javac -Xrealtime source
   ```

   该命令从您的源创建 Java 字节码，以便在实时环境中使用。请参阅第 8 页的图 2。
2. 将生成的类文件封装到 JAR 文件中。例如，要创建 test.jar:

   ```
   jar cvf test.jar source
   ```
3. 在 shell 提示符处输入:

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

   该命令会预编译 test.jar 文件，并将输出写入到输出目录 ./aot。
4. 要使用共享类高速缓存中的 AOT 代码来运行文件，请在 shell 提示符处输入:

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

要使用共享类高速缓存中的 AOT 代码来运行文件，请重新编译频繁调用的方法。然后，运行以下命令，而不创建新的 JAR 文件:

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

这些命令使用您在第 3 步中预编译的相同 JAR 文件。

使用 admincache 工具

admincache 工具用于管理工作站上的共享类高速缓存。

在 IBM WebSphere Real Time for RT Linux 产品中，admincache 工具可以用于创建共享类高速缓存，其中包含一个或多个类以及 AOT 编译的代码。在创建了高速缓存之后，此工具还可以用于检验现有的高速缓存。

共享类高速缓存用于在多 JVM 场景中减少内存占用量，并且加速应用程序启动。

WebSphere Real Time for RT Linux 可以通过非实时和实时方式来使用共享类高速缓存，但高速缓存格式、创建和填充方法会有所不同。实时方式的高速缓存与非实时方式的高速缓存不兼容。在非实时方式中，创建和填充高速缓存的方法与标准 JVM 相同。这表示 JVM 在其运行应用程序时创建和填充高速缓存，这一过程对用户是透明的。在实时方式中，使用 **-Xrealtime** 选项时，必须通过 admincache 使用 **-populate** 选项来创建和预填充共享类高速缓存。以实时方式运行的应用程序可以读取预填充高速缓存中的内容，但无法修改其内容。

仅当应用程序以实时方式运行时，才能使用以实时方式创建的共享类高速缓存。仅当应用程序以非实时方式运行时，才能使用以非实时方式创建的共享类高速缓存。这个规则也适用于 admincache 工具。要管理 JVM 以实时方式创建的高速缓存，请将 admincache 与 **-Xrealtime** 选项一起使用。要管理 JVM 以非实时方式创建的高速缓存，请勿使用 **-Xrealtime** 选项。要在运行时连接到共享类高速缓存，请向命令行中添加 **-Xshareclasses** 选项。

工作站上可以创建多个共享类高速缓存，每个都具有特定的名称并且位于特定的目录。创建新的高速缓存时，可以使用 **-cacheName** <*name*> 选项来指定其名称。高速缓存名称不能超出 53 个字符。

缺省情况下，共享类高速缓存在 /tmp/javasharedresources 目录中创建，但可以通过指定 **-cacheDir** <*directory*> 选项来覆盖此位置。共享类高速缓存的内部格式取决于创建时所在工作站的特征。这意味着无法在网络驱动器上创建共享类高速缓存来作为安全措施。此限制的另一个原因是，从网络文件系统访问共享类高速缓存的速度较慢和不可预测的性能影响。

如果命令行中未指定高速缓存名称，那么缺省值为 **sharedcc_<user_login>**

有关非实时方式中的共享高速缓存操作的更多信息，请参阅第 85 页的『非实时方式下 JVM 之间的类数据共享』。

注: 从 IBM WebSphere Real Time for RT Linux V2 SR1 及更高版本开始，您必须将 **-classpath** 选项与 **-populate** 选项一起使用。

创建实时共享类高速缓存:

admincache 工具可以用于创建能以实时方式访问的共享类高速缓存。

注: 在使用缺省设置创建共享类高速缓存时，您必须注意安全性注意事项。请参阅第 87 页的『共享类高速缓存的安全注意事项』以获取有关共享类高速缓存的安全性注意事项的更多信息，以及有关更改缺省许可权的信息。

admincache 工具的 **-populate** 选项可以用于创建共享类高速缓存。该选项与 .jar 文件列表或搜索 .jar 文件的目录或目录树一起使用。对于指定或找到的每个 .jar 文件，admincache 将 .jar 文件中的每个存储在共享类高速缓存中。除非您指定了 **-noaot** 选项，否则类方法还会使用 AOT 编译并存储在共享类高速缓存中。

您必须将 **-classpath** 选项与 **-populate** 一起使用，否则您将会看到以下错误消息:

```
-populate 操作需要指定 -classpath <class path> 选项
```

admincache **-help** 选项会列出可以用于控制 admincache 如何填充高速缓存的子选项。

```
$ admincache -Xrealtime -help
用法: admincache [option]*
  其中 [option] 可以是:
     -help | -?                操作: 显示此帮助
     -Xrealtime                在实时环境中使用
     -cacheName <name>         指定共享高速缓存的名称（使用 %%u 替代用户名）
     -cacheDir <dir>           设置 JVM 缓存文件的位置
     -listAllCaches            操作: 列出所有现有的共享类高速缓存
     -printStats               操作: 打印高速缓存统计信息
     -printAllStats            操作: 打印更详细的高速缓存统计信息
     -destroy                  操作: 销毁指定名称（或缺省）的高速缓存
     -destroyAll               操作: 销毁所有高速缓存
     -populate                 操作: 创建新的高速缓存并填充
       -searchPath <path>      指定用于查找文件的目录（如果未指定任何文件）
                               （缺省值为 .）
                               只能指定一个 -searchPath 选项
       -classpath <class path> 指定将在运行时用于访问该高速缓存的类路径
                               -classpath 选项是必需的
       -[no]recurse            [不]递归到子目录以查找要转换的文件
                               （缺省情况下为不递归）
       -[no]grow               如果指定的高速缓存已存在，[不]添加到其中（缺省值为不增长）
                               如果未选择 -grow 选项，在存在指定的高速缓存的情况下，将会被移除
       -verbose                打印出每个 jar 的进度消息
       -noisy                  打印出每个 jar 中的每个类的进度消息
       -quiet                  禁止所有输出
       -[no]aot                在将类存储到高速缓存中之后，还对方法执行 AOT 编译
       -aotFilter <signature>  仅对匹配的方法执行 AOT 编译并存储到高速缓存中
          例如 -aotFilter {mypackage/myclass.mymethod(I)I} 仅编译 mymethod(I)I
          例如 -aotFilter {mypackage/myclass.mymethod*} 编译所有 mymethod
          例如 -aotFilter {mypackage/myclass.*} 编译 myclass 中的所有方法
       -aotFilterFile <file>   仅对文件中匹配那些条件的方法执行 AOT 编译并存储到
                               高速缓存中（必须已使用 -Xjit:verbose={precompile},
                               vlog=<file> 创建了输入文件）
       -printvmargs            打印在运行时访问已填充的高速缓存所需要的 VM 参数
       [jar file]*.[jar][zip]  要填充到高速缓存中的 jar 文件的显式列表
                               如果未指定任何文件，则将会转换 searchPath 中的
                               所有 files.[jar][zip]。
必须仅指定一个操作选项
```

注: 在使用共享类高速缓存时，**-cacheName** 选项指定的名称不能超出 53 个字符。

可以指定 .jar 文件的列表，在这种情况下，只有那些 .jar 文件中的类将会添加到共享类高速缓存中。如果您未指定 .jar 文件的列表，请使用 **-searchPath <path>** 选项指定

目录树以搜索 .jar 或 .zip 文件。**-recurse** 选项是缺省的，表示以递归方式搜索目录树以查找 .jar 或 .zip 文件。**-norecurse** 选项表示仅搜索指定的目录。指定 **-classpath <class path>** 选项以使 admincache 可以找到用于处理指定的 .jar 文件所需要的所有类。在填充共享类高速缓存的过程中，这些类会装入 JVM，因此当 admincache 尝试从 .jar 文件中装入类时，确保它发现所有引用的类和超类是十分重要的。

-grow 选项指定在高速缓存目录中存在具有相同名称的共享类高速缓存时，向现有的高速缓存内容中添加新的 .jar 文件。**-nogrow** 选项指定在原有高速缓存目录中存在具有相同名称的共享类高速缓存时，创建新的 .jar 文件替换原有的高速缓存内容。**-grow** 选项用于在共享类高速缓存中添加当前不存在的新的 .jar 文件，而不替换已更改的类。如果类由于应用程序修改而发生更改，若要更新高速缓存中已存在的这些类，请勿使用 **-grow** 选项。要更新现有的类，请使用当前的类内容创建全新的高速缓存。当您更改某个类时，若不更新共享类高速缓存，您的应用程序仍将使用新的类内容正常运行，但无法发挥共享类高速缓存的优点。这是由于更改的类将会从磁盘中装入，而不是从共享类高速缓存。从磁盘中装入类表示无法为该类使用 AOT 编译的代码。因此当您更改类时，请重新生成共享类高速缓存。

使用 **-quiet**、**-verbose** 和 **-noisy** 选项可以控制 admincache 提供的信息的详细级别。

要为填充共享类高速缓存的类中的方法指定提前 (AOT) 预编译，请使用 **-aot** 选项。要阻止 AOT 预编译并且仅将类存储到共享类高速缓存中，请使用 **-noaot** 选项。**-aot** 选项是缺省设置。

要有选择地预编译部分方法，请使用 **-aotFilter** <*signature*> 或 **-aotFilterFile** <*file*> 选项。<*signature*> 是对方法特征符的一条简化的正则表达式，放在大括号中，其中“*”可以替换任何字符序列。您可能需要将 <*signature*> 放在单引号中，以使 shell 不会解析方法特征符中的任何字符。

表 5 显示了 <*signature*> 选项的一些示例。

表 5. <signature> 选项的示例

特征符	含义
`-aotFilter '{java/lang/*}'`	对 java/lang 程序包中的方法执行 AOT 编译。
`-aotFilter '{*.sample*}'`	对以“sample”开头的方法执行 AOT 编译。
`-aotFilter '{mypackage/myclass.mymethod(I)I}'`	对与该特征符完全匹配的方法执行 AOT 编译。

-aotFilterFile <*file*> 选项使用 <*file*> 的内容来选择执行 AOT 编译的方法。任何其他方法都不执行 AOT 编译。<*file*> 的内容会在之前应用程序的运行过程中使用 **-Xjit:verbose={precompile},vlog=**<*file*> 选项生成。详细的输出使用内部格式存储在 <*file*> 中。使用 **-aotFilterFile** 选项时需要该格式。

注: **-vlog=**<*file*> 选项不会直接生成名称为“file”的文件。生成详细的输出时会在“file”上附加日期和进程标识。通过指定选项 **-Xjit:verbose={precompile},vlog=my_file**，生成的文件名会类似于 `my_file.<date>.<#>.<process id>`。额外的字段使它更便于在多 JVM 场景中生成单独的详细日志文件，在这些场景中很难对某个特定 JVM 提供命令行选项，或难以为不同 JVM 使用不同的 **-Xjit** 命令行选项。在单 JVM 场景中，这些数字会附加到命令行提供的文件名中。

生成的文件可以与 **-aotFilterFile** 选项一起使用，不需要任何编辑。由多次应用程序运行使用 **-Xjit:verbose={precompile},vlog=<file>** 选项生成的多个详细日志文件，可以合并并提供给使用 **-aotFilterFile** 选项的 admincache。

-printvmargs 选项可以帮助确保在应用程序运行时为命令行提供正确的参数。

```
$ 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
```

在本示例中，最后一行输出显示了运行应用程序时应向命令行中添加的选项，以便能够使用共享类高速缓存中存储的类和 AOT 方法。要在本示例中使用这些选项，请输入命令:

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

使用 admincache 管理共享类高速缓存:

admincache 工具包含多个实用程序用于您的管理系统上的共享类高速缓存。

admincache 工具提供了实用程序以帮助完成多种任务。

- 列出高速缓存中现有的共享类高速缓存。
- 提供有关共享类高速缓存的内容的详细信息。
- 移除特定高速缓存目录中的部分或所有高速缓存。

列出可用的共享类高速缓存:

admincache 工具提供了高速缓存中现有共享类高速缓存的列表。

要获取高速缓存中现有的全部共享类高速缓存的列表，请使用 **-listAllCaches** 选项并使用 **-cacheDir** 选项指定高速缓存目录。

```
$ admincache -Xrealtime -listAllCaches

admincache 1.02

正在列出高速缓存目录 /tmp/javasharedresources/ 中的所有高速缓存

高速缓存名称              级别          持久性      上次分离时间

兼容的共享高速缓存
sharedcc_username       Java6 32 位   是          2008 年 10 月 16 日，周四，17:02:39
rtCache                 Java6 32 位   是          2008 年 10 月 16 日，周四，17:03:12

不兼容的共享高速缓存
nonrtCache              Java6 32 位   是          2008 年 10 月 16 日，周四，17:17:32
```

在本示例中，缺省高速缓存目录中有两个兼容的共享类高速缓存:

- 使用登录名 *username* 的用户的缺省命名的高速缓存

- 另一个名称为 *rtCache* 的高速缓存

该示例还显示了一个名称为 *nonrtCache* 的不兼容高速缓存。*nonrtCache* 是由 JVM 在以非实时方式执行时创建的。这表示不能使用 **-Xrealtime** 选项来访问它。

实时方式的 JVM 可以查看以非实时方式创建的高速缓存。非实时方式的 JVM 不能查看以实时方式创建的高速缓存。

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

© 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


正在列出高速缓存目录 /tmp/javasharedresources/ 中的所有高速缓存

高速缓存名称              级别          持久性       上次分离时间

兼容的共享高速缓存
nonrtCache                Java6 32 位   是           2008 年 10 月 16 日，周四，17:17:32
```

在本示例中，由于未指定 **-Xrealtime**，因此列出 *nonrtCache* 并显示为兼容。

检验共享类高速缓存的内容:

admincache 工具可以描述共享类高速缓存的内容。

您可以使用 admincache 工具的 **-printStats** 选项来获取一份概述，其中描述了共享类高速缓存的主要内容。有关特定高速缓存目录中特定高速缓存的信息，请使用 **-cacheName** 和 **-cacheDir** 选项。以下示例提供了有关缺省高速缓存目录中 *nonrtCache* 高速缓存的信息。

```
$ admincache -cacheName nonrtCache -printStats

admincache 1.02

Current statistics for cache "nonrtCache":


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

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

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

Cache is 16% full
```

注: 在使用共享类高速缓存时，高速缓存的名称不能超出 53 个字符。

其中有数条有关此高速缓存的有用信息:

- 高速缓存的大小，显示为高速缓存大小 = 16776852。
- 高速缓存中的可用空间，显示为可用字节数 = 14070360。您可以计算出该高速缓存大约使用了 16%。
- 高速缓存中存储的类的数量，显示为 # ROMClass 数 = 372。
- 高速缓存中存储的 AOT 方法的数量，显示为 # AOT 方法数 = 981。

有关 admincache 工具中 **-printStats** 选项所提供的信息的更多详细信息，请参阅 printStats 实用程序。

-printAllStats 选项对共享类高速缓存的内容提供了更详细的描述。信息中包含了高速缓存中存储的类和 AOT 方法的列表。**-printAllStats** 选项的输出十分详细。

高速缓存中包含的类会用类似以下的行来表示:

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

这一行表示类 java/lang/ClassLoader 包含在高速缓存中。地址相对于共享类高速缓存为内部，除了诊断用途以外几乎很少使用。

高速缓存中包含的 AOT 方法会用类似以下的行来表示:

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

这几行表示 java/lang/ClassLoader 类中的 callerClassLoader 方法包含在高速缓存中。列出的地址是内部共享高速缓存地址。**-printAllStats** 选项的输出不会显示高速缓存中每个 AOT 方法的特征符，该特征符由参数类型和返回类型组成。

有关 admincache 工具中 **-printAllStats** 选项所提供的信息的更多详细信息，请参阅 printAllStats 实用程序。

销毁共享类高速缓存:

admincache 工具具有擦除指定高速缓存目录中的特定高速缓存或所有高速缓存的选项。

在用户具有相应许可权的情况下，admincache 工具 **-destroy** 选项可以用于擦除某一具体高速缓存目录中的特定高速缓存。在用户具有相应许可权的情况下，**-destroyAll** 选项可以用于擦除所有高速缓存。例如:

```
$ admincache -Xrealtime -destroy

admincache 1.02

JVMSHRC256I 已销毁持久性共享高速缓存"sharedcc_username"
```

擦除高速缓存之后，缺省高速缓存目录中可用共享类高速缓存的列表将会显示已擦除的高速缓存不再出现:

```
$ admincache -Xrealtime -listAllCaches

admincache 1.02

正在列出高速缓存目录 /tmp/javasharedresources/ 中的所有高速缓存
```

```
高速缓存名称                级别          持久性      上次分离时间

兼容的共享高速缓存
rtCache                     Java6 32 位   是          2008 年 10 月 16 日，周四，17:03:12

不兼容的共享高速缓存
nonrtCache                  Java6 32 位   是          2008 年 10 月 16 日，周四，17:17:32
```

-destroyAll 选项会移除指定高速缓存目录中的所有高速缓存，不论它们与当前的 JVM 是否兼容。**-destroyAll** 选项必须谨慎使用:

```
$ admincache -Xrealtime -destroyAll

admincache 1.02

正在尝试销毁高速缓存目录 /tmp/javasharedresources/ 中的所有高速缓存

JVMSHRC256I 已销毁持久性共享高速缓存"rtCache"
JVMSHRC256I 已销毁持久性共享高速缓存"nonrtCache"
```

结果是机器上不再有任何共享类高速缓存可用:

```
$ admincache -Xrealtime -listAllCaches

admincache 1.02

JVMSHRC005I 没有可用的共享类高速缓存
```

如果当前的用户没有访问高速缓存的许可权，那么 **-destroy** 或 **-destroyAll** 选项都不会销毁高速缓存。

共享类高速缓存的实际大小:

admincache 工具提供有关为共享类高速缓存确定大小的信息。

对于较小的应用程序，即使将所有类和方法都填充到共享类高速缓存中，也不会产生大小无法接受的高速缓存。对于较大的应用程序，所生成的共享类高速缓存的大小对于实际用途可能会过大。这是因为 JVM 进程必须具有足够的虚拟地址空间，才能对共享类高速缓存的全部内容进行寻址。在使用共享类高速缓存时技术时，有一些注意事项。

共享类高速缓存整体在连接到它的任何 JVM 中都必须实际均可寻址。这意味着要避免使用大小超过 700 MB 的共享类高速缓存。admincache 工具可以预测高速缓存的大小。如果工具指示高速缓存的大小将会超过 700 MB 限制，则会显示消息建议您存储更少数量的类，或者更严格的选择存储在高速缓存中的 AOT 方法。

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

admincache 1.02

警告: 预测的高速缓存大小 (15960MB) 超过了建议的最大共享类高速缓存大小 (700MB)
如果您的 JAR 文件中包含了主要类文件，那么可能无法创建此大小的高速缓存，
        或者可能在运行应用程序时无法连接到创建的高速缓存。
对此，您可以通过使用 -aotFilterFile 来更有选择性地编译 AOT 方法
要覆盖此警告消息，请在命令行中直接指定 -Xscmx15960M，
        但请注意，导致的失败可能会在填充过程的最后
        才会出现。
```

admincache 工具会根据为填充指定或找到的 .jar 文件的大小总和，预测一个保守的高速缓存大小。这意味当 .jar 文件包含的许多文件并非类文件时，预测就可能不准确。要获取对高速缓存大小更加准确的预测，请为 .jar 文件创建仅包含类文件的临时版本。如果

admincache 工具仍然产生警告消息，您可以通过使用 **-aotFilter** <*pattern*> 或 **-aotFilterFile** <*file*> 选项，来更加有选择性地考虑对 .jar 文件中的方法进行 AOT 预编译。admincache 工具消息会提醒您预测不会将这些选项过滤掉的 AOT 方法考虑在内。

要覆盖警告消息并继续执行高速缓存填充步骤，请向 admincache 命令行中添加指示的 **-Xscmx** 选项。如果预测大小非常大，那么 admincache 工具可能无法创建所需大小的共享类高速缓存。要解决此问题，请减小高速缓存大小，直到 admincache 工具能够继续执行。

当最终的高速缓存写入磁盘中时，它的大小恰好能满足容纳指定的类和 AOT 方法的需要。这意味着指定一个较大的初始高速缓存大小并不会有问题。

在共享类高速缓存中存储 SDK 类:

并非所有应用程序都需要创建一个包含 SDK 中所有 JAR 文件的高速缓存。

SDK 中的 JAR 文件的数量和大小意味着尝试创建包含所有这些 JAR 文件的高速缓存会导致一条警告消息，告知您生成的高速缓存将会过大。对于许多应用程序而言，大多数 SDK JAR 文件从不会被引用。

主要的 SDK JAR 文件位于 SDK/jre/lib 目录中。对于大多数应用程序，这些 JAR 文件中最重要的是 rt.jar，它是 Java 6 发行版中新增的。rt.jar 是一个类的集合，在 Java 6 发行版之前，这些类存储在单独的 JAR 文件中。单独使用 rt.jar 来填充共享类高速缓存，并使用 AOT 编译器编译它的所有方法，这样创建的高速缓存大小大约是 300 MB。典型的应用程序将不会引用中 rt.jar 类中大部分方法。要使用 rt.jar 来填充共享类高速缓存:

1. 仅将 rt.jar 中的类填充到共享类高速缓存中。这大约会消耗高速缓存中 50 MB 的空间。
2. 使用 **-aotFilterFile** <*file*> 选项，以仅编译您的程序可能使用的方法。您可以通过运行应用程序来生成 <*file*>。

SDK 中还有其他一些常用和重要的 JAR 文件，包括:

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

realtime.jar 包含了“Java 实现规范 (RTSJ)”的 IBM 实现。如果您的应用程序使用任何 RTSJ 功能，请将 realtime.jar 文件存储在共享类高速缓存中，以获得更好的预期性能。vm.jar 包含多个内部 JVM 类，通常所有的应用程序中都会使用。java.util.jar 包含多个容器类，必须存储到每个应用程序的共享类高速缓存中，以获得更好的预期性能。

如果应用程序使用 sdk/jre/lib 和 sdk/jre/lib/ext 目录中的其他 JAR 文件的那些类，可以将这些 JAR 文件存储到共享类高速缓存中。识别您的应用程序是否使用这些类的最简便的方法，就是在运行程序时使用 **-verbose:dynload** 选项。**-verbose:dynload** 选项仅会描述应用程序的当前运行所装入的类。例如:

```
<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>
```

本示例输出显示了从两个不同 SDK JAR 文件装入三个类。java/io/InputStreamReader 类从 `rt.jar` 中装入。java/util/LinkedHashSet 和 java/util/HashSet 类从 `java.util.jar` 中装入。

其他 admincache 注意事项:

有关使用 admincache 的有用信息。

高速缓存填充和永久内存大小调整

当 **admincache** 工具以实时方式填充共享类高速缓存时，它必须在完成填充的过程中装入每个类。每个类都会消耗一定的永久内存，因此有可能出现缺省的永久内存大小无法容纳已请求的所有类。如果 **admincache** 工具在向高速缓存中填充许多类时抛出 OutOfMemory 错误，请尝试使用 **-Xgc:immortalMemorySize=32M** 选项，以增加永久内存大小使其超过缺省值 16 MB。

在您更改类时

如果磁盘上的类文件发生更改，共享类高速缓存技术会自动检测到不能使用共享类高速缓存中该类的版本。您的程序将会正常运行，但无法完全发挥共享类高速缓存的优点，并且将不会使用该类的任何 AOT 方法。如果您更改应用程序中的某个类，请重新创建共享类高速缓存。请勿尝试使用 **-grow** 选项来仅重新填充包含所修改类的 .jar 文件，因为此选项并不是旨在用于 .jar 文件已存在于高速缓存中的场景。

管理共享类

即使没有文件要装入，共享高速缓存仍然需要地址空间。请参阅第 96 页的『IBM JVM 管理内存的方式』，以获取有关共享类高速缓存在 JVM 进程中如何消耗内存的更多信息。

将预编译的 JAR 文件存储到共享类高速缓存中

您可以将全部或部分由 IBM 提供的 Java 类存储或包含在共享类高速缓存中。这一过程使用 **-Xrealtime** 选项与 **javac** 以及 **admincache** 工具来将类存储到共享类高速缓存中。

开始之前

仅当使用 **-Xrealtime** 选项时，并且使用 **-Xrealtime** 选项运行 Java 时，才支持提前将 JAR 文件存储到共享类高速缓存中。使用或不使用 **-Xrealtime** 选项运行时，您都可以使用相同的 JAR 文件，但只有指定了 **-Xrealtime** 选项时，才能使用已存储到高速缓存中的 JAR 文件。

注: 在使用共享类高速缓存时，高速缓存名称不能超出 53 个字符。

关于此任务

您可以使用 **admincache** 工具将 JAR 文件存储到共享类高速缓存中。**admincache** 使您能够以三种方式之一来构建您的应用程序。

注:
- 如果您在 Linux 系统上设置了超时，您可能需要在预编译大型 JAR 文件时覆盖它；否则，编译将会超时并且将不会创建 JAR 文件。

预编译应用程序中的所有类和方法:

此过程会预编译应用程序中的所有类。它将 JAR 文件的集合存储到共享类高速缓存中。那些 JAR 文件中的所有类中的所有方法也都存储到高速缓存中。经过优化的 JAR 文件的所有方法都已经过编译。

关于此任务

针对本示例，应用程序位于环境变量 *$APP_HOME* 指定的目录下，而 JAR 文件位于子目录 *$APP_HOME*/lib 中。该应用程序还使用部分由 IBM 提供的 core.jarrt.jar 中的类。在这种情况下，您只能预编译应用程序代码，也就是 main.jar 和 util.jar。

缺省情况下，共享类高速缓存位于 /tmp/javasharedresources 中。使用 **-cacheDir** 选项可以将高速缓存放入其他目录中。您无法在网络文件系统上创建高速缓存。

过程

1. 在 shell 提示符处输入： cd *$APP_HOME*

   其中 *$APP_HOME* 是您的应用程序的目录。
2. 在 shell 提示符处输入：cd *$APP_HOME*/lib。 *$APP_HOME*/lib 是存储 main.jar 和 util.jar 的目录。
3. 在 shell 提示符处输入：admincache -Xrealtime -populate -aot -classpath *$APP_HOME*/lib -searchPath *$APP_HOME*/lib -norecurse . 此过程会优化在 *$APP_HOME*/lib 中发现的每个 JAR 文件，将进度信息输出在屏幕上，然后在 *$APP_HOME*/aot 目录中创建新的 JAR 文件。您可以使用 **-cacheName <name>** 来指定高速缓存名称，但如果未指定名称，那么将使用基于用户登录名的缺省名称。

   注: 使用 **-cacheName** 选项指定的名称不能超出 53 个字符。
4. 在 shell 提示符处输入 admincache -Xrealtime -listAllCaches 会显示高速缓存是否存在。

下一步做什么

要查看更多选项，请输入：admincache -Xrealtime -help。

预编译频繁使用的方法:

您可以使用概要文件指导的 AOT 编译，仅对应用程序频繁使用的方法进行预编译。通过使用特殊的选项 **-Xjit:verbose={precompile},vlog=optFile** 来运行应用程序，AOT 编译可以使用生成的选项文件将一组 JAR 文件存储到共享类高速缓存中。只有选项文件中列出的方法会经过预编译。

开始之前

在您开始之前，创建通常由 JIT 编译器编译的方法列表。

关于此任务

您可以编辑由 **-Xjit:verbose={precompile}** 选项生成的文件。该文件是要进行预编译的方法的明确规范。这些方法是特定的；也就是说，它们包含要进行编译的每个方法的完整特征符，这个特征符使您编译 com/acme/sample.myMethod(J)V 而不编译 com/acme/sample.myMethod(I)V。

注：在使用共享类高速缓存时，高速缓存的名称不能超出 53 个字符。

过程

1. 在 shell 提示符处输入：

   ```
   cd $APP_HOME
   ```

   其中 *$APP_HOME* 是您的应用程序的目录。

2. 在 shell 提示符处输入：

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

   其中：

   - *app.precompileOpts* 是日志文件的名称，其中列出了使用 JIT 编译的方法。
   - *applicationName* 是您的应用程序的名称。

   该命令会创建使用 JIT 编译的方法列表。

3. 在 shell 提示符处输入：

   ```
   cd $APP_HOME/lib
   ```

   $APP_HOME/lib 是存储用于应用程序的 JAR 文件的目录。

4. 要将所有样本应用程序方法编译到高速缓存中，请输入：

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

5. 要将 realtime.jar 和 vm.jar 编译到高速缓存中，请输入：

```
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. 要将 rt.jar 编译到高速缓存中，请输入：

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

7. 要测试此命令，请使用 **-nojit** 选项运行您的应用程序，这样会使用高速缓存中的代码。在 shell 提示符处输入：

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

   其中 *applicationName* 是您的应用程序的名称。

预编译 IBM 提供的文件:

您可以预编译 IBM 提供的文件（例如，rt.jar），以实现性能和可预测性之间的平衡。

关于此任务

预编译类似于预编译您的应用程序 JAR 的任务，但在运行时有额外的需求；您必须确保正确指定了引导类路径，以便使用这些文件而不是 JRE 中文件。您可以使用 **-Xshareclasses** 选项来完成该操作，它指示 JVM 首先在指定的类高速缓存中查找，然后才查找缺省的类路径位置。

注: 在使用共享类高速缓存时，高速缓存的名称不能超出 53 个字符。

预编译 rt.jar 以便在应用程序中使用:

过程

1. 在 shell 提示符处输入: cd *$JAVA_HOME*/lib 其中 *$JAVA_HOME* 是您的 Java 主目录。
2. 运行 **admincache** 工具。在 shell 提示符处，输入:

   ```
   admincache -Xrealtime -populate -cacheName myCache
       -classpath <class path> rt.jar
   ```

   该命令会使用对 IBM 提供的名称为 rt.jar 的文件进行预编译的结果，来填充名称为 myCache 的高速缓存。
3. 运行您的应用程序，指定 **-Xshareclasses** 选项来指定高速缓存名称。要运行您的应用程序，请输入:

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

Just-In-Time (JIT) 编译器

您可以使用随标准 SDK 类库提供的 java.lang.Compiler 类来控制 JIT 编译器的运行时间和方式。IBM 完全支持 Compile.compileClass()、Compiler.enable() 和 Compiler.disable() 方法。

例如，如果您想热启应用程序并知道已编译应用程序中的主要方法，那么您可在热启应用程序后并确信在其余的应用程序执行期间不会发生 JIT 编译时调用 Compiler.disable() 方法。

您可用以下两种方式来控制方法编译:

- 指定可以编译的方法集:

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

  其中 *<method specification>* 是已在此时装入并即将编译的所有方法列表。*<method specification>* 描述标准方法名称。星号表示通配符匹配。

  例如，要编译所有已装入的并以 java.lang.String 开头的方法，请指定:

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

  注: 该命令不仅编译 java.lang.String 类中的方法，还编译 java.lang.StringBuffer 类中的方法，这可能并不是您所希望的。要仅编译 java.lang.String 类中的方法，请指定:

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

- 指定在该线程运行并继续前对编译队列中的所有方法进行编译:

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

可能希望确保在禁用编译器前编译队列为空。编译方法和类集的典型方法是:

```
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
```

JNI 转换期间的确定性

缺省情况下，JIT 生成优化代码，用于实现高性能的 Java 到本机 (J2N) JNI 转换。使用以下代码序列来重新装入本机库时可能出现确定性降低。

```
RegisterNatives / UnregisterNatives / RegisterNatives
```

要还原较慢、更确定的代码，请使用命令行选项 **-Xjit:disableDirectToJNI**。

启用 JIT

您可以使用多种不同的方法来显式地启用 JIT。这两个命令行选项都将覆盖 **JAVA_COMPILER** 环境变量。

过程

- 在运行 Java 应用程序之前，将 **JAVA_COMPILER** 环境变量设置为“jitc”。 在 shell 提示符处，输入:
  - **对于 Korn shell 程序**: `export JAVA_COMPILER=jitc`

    注: 除非另行声明，否则在该信息中使用 Korn shell 程序命令。
  - **对于 Bourne shell 程序**:

    ```
    JAVA_COMPILER=jitc
    export JAVA_COMPILER
    ```
  - **对于 C shell 程序**: `setenv JAVA_COMPILER jitc`

  如果 **JAVA_COMPILER** 环境变量是空字符串，那么仍禁用 JIT。要禁用该环境变量，在 shell 提示符处输入 `unset` **JAVA_COMPILER**。
- 在 JVM 命令行上使用 **-D** 选项将 java.compiler 属性设置为“jitc”。 在 shell 提示符处输入: `java -Djava.compiler=jitc <MyApp>`
- 在 JVM 命令行上使用 **-Xjit** 选项。 不要同时指定 **-Xint** 选项。 在 shell 提示符处输入: `java -Xjit <MyApp>`

禁用 JIT

您可以使用多种不同的方法来禁用 JIT。这两个命令行选项都将覆盖 **JAVA_COMPILER** 环境变量。

关于此任务

过程

- 在运行 Java 应用程序之前，将 **JAVA_COMPILER** 环境变量设置为“NONE”或空字符串。在 shell 提示符处输入:
  - **对于 Korn shell 程序**: `export JAVA_COMPILER=NONE`

注: Korn shell 程序命令用于该信息的其余部分。

– **对于 Bourne shell 程序**:

```
JAVA_COMPILER=NONE
export JAVA_COMPILER
```

– **对于 C shell 程序**: `setenv JAVA_COMPILER NONE`

- 在 JVM 命令行上使用 **-D** 选项将 java.compiler 属性设置为“NONE”或空字符串。 在 shell 提示符处输入: `java -Djava.compiler=NONE <MyApp>`
- 在 JVM 命令行上使用 **-Xint** 选项。 在 shell 提示符处输入: `java -Xint <MyApp>`

确定是否已启用 JIT

您可以使用 **-version** 选项确定 JIT 的状态。

过程

在 shell 提示符中输入以下命令:

```
java -version
```

如果 JIT 没在使用，将显示一条包含以下文本的消息:
（JIT 已禁用）
如果 JIT 正在使用，将显示一条包含以下文本的消息:
（JIT 已启用）

使用非堆实时线程

Metronome 垃圾回收提供更加一致的响应时间，但有时候完全避免因垃圾回收所造成的中断是适当的做法。

NoHeapRealtimeThread (NHRT) 是对 RealtimeThread 的扩展。它们与 RealtimeThread 的不同之处在于它们不具有对堆内存的访问权。如果没有对堆的访问权，那么即使在垃圾回收周期内 NHRT 也能够继续运行，但存在一些限制。由此可知，如果没有对堆的访问权，那么编程模型将不同于实时线程的编程模型。

使用 NHRT 时的注意事项

请考虑关于 NHRT 的以下几点事项:

- 使用 NHRT 的主要原因是有无法容忍垃圾回收的任务。例如，如果您的应用程序是时间关键型应用程序，并且无法容忍任何中断。
- 如果时间非常关键，以致于您要使用 NHRT，那么还请考虑提前 (AOT) 编译器，即使用 **-Xnojit** 选项。
- 在使用 **-Xrealtime** 选项时，会自动使用 Metronome 垃圾回收器。Metronome 垃圾回收器的优势对于贵企业可能已足够，从而可减少对 NHRT 编码的需求。
- NHRT 线程独立于垃圾回收器来运行，因为它们的优先级高于垃圾回收器的优先级。Java 线程的优先级可在范围 1 - 10 内。如果存在 NHRT，那么无论程序中的优先级设置如何，Java 线程的优先级都会重置为 0。垃圾回收器会自动设置为比最高实时线程高出半步。您将 NHRT 的优先级设置为至少比最高实时线程高出 1。通过这种方式，NHRT 便独立于垃圾回收器。

  注: NHRT 并非完全不受垃圾回收的影响，因为 Metronome 警报线程垃圾回收器在系统中以最高优先级运行。该优先级确保能够激活 JVM 以检查垃圾回收器是否必须

执行某操作。运行 Metronome 警报线程的工作较少，因而不会显著影响性能。在多处理器系统上，警报线程可以与 NHRT 线程同时运行，因此不会发生垃圾回收中断。

- 因为 NHRT 被限制在设置了作用域的和永久内存区域，所以 Java 方法执行检查以确保它们不是分配于堆。启动方法会进行检查，并且如果 NHRT 是分配于堆，那么会返回异常 (`MemoryAccessError`)。NHRT 只能访问 `ImmortalMemory` 和 `ScopedMemory`。
- 锁定语义不变，以便在共享了锁定的情况下，常规线程可以阻止 NHRT 线程。
- 正在使用堆的线程可以在 NHRT 尝试使用同步的方法时提升其自身对于该方法的优先级。
- 请对 NHRT 与堆线程之间的通信使用无阻止队列。否则，请将这两种类型的线程分开。

异常

使用 NHRT 时可能会发生以下异常:

- `IllegalAssignmentError`。例如，尝试在永久内存中创建对设置了作用域的内存的引用时可能会发生该错误。
- `MemoryAccessError`。例如，NHRT 尝试引用堆内存时可能会发生该错误。

异步事件处理程序约束

在很多种情况下，在垃圾回收期间都会阻止 NHRT，包括以下情况:

1. NHRT 对与分配于堆内存的处理程序关联的 AsyncEvent 调用 fire()、setHandler() 或 addHandler()
2. NHRT 对与分配于堆内存的处理程序关联的 Timer 调用 destroy()、start() 或 stop()
3. NHRT 是最后一个退出作用域的线程，并正在关闭该作用域中的 Timer 或 AsyncEvent。但是，这些 Timer 或 AsyncEvent 具有分配于堆内存的关联处理程序

要避免对于 NHRT 发生这些情况:

1. 请避免向可能被 NHRT 触发的 AsyncEvent 或 Timer 添加分配于堆的处理程序。
2. 请避免 NHRT 最后一个从具有 AsyncEvent 或 Timer（它们具有分配于堆内存的处理程序）的作用域中退出的情况。

内存和调度约束

JVM 阻止非堆实时线程将对堆中对象的引用加载到其操作数堆栈上。如果这么做，那么会抛出 javax.realtime.MemoryAccessError。

JVM 还会防止在堆或永久内存中存储对设置了作用域的内存中对象的引用。虽然设置了作用域的内存并非专门供 NHRT 使用，但在永久内存不适当并且 NHRT 上下文中需要释放内存的情况下，还是很可能被使用。

当 NHRT 正在执行时，如果它使用对于对象的引用来填充字段，那么可以成功覆盖该字段中对堆中对象的任何预先已存在的引用。此预先已存在的引用将成功被 NHRT 覆盖而不会生成 MemoryAccessError。

类加载约束

类加载到与类装入器相同的内存区域中。类装入器的缺省区域是永久内存。

为了使应用程序能够提供预期响应时间，它们必须已“热身”。应用程序应该较早就加载其类，这样类加载便不会在稍后中断实时线程和异步事件处理程序。

Java 线程在与 NHRT 一起运行时的约束

因为系统属性在 JVM 中共享，并且任何线程都能访问系统属性，所以在运行 NHRT 的 JVM 中使用 getProperties 和 setProperties 方法时需要谨慎小心。为使系统属性可供 NHRT 访问，它们必须位于永久内存中。

java.lang.System 类提供使线程能够与系统属性交互的若干方法，包括以下方法:

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

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

实时 JVM 使用特别为实时 JVM 对象创建的 com.ibm.realtime.ImmortalProperties 类实例来存储所有系统属性。使用该实例可确保对 System.setProperty() 或 System.getProperties.setProperty() 方法的任何调用都会使属性存储在永久内存中。此情况下无需特殊用户代码，但请务必了解每次设置一个属性都会消耗一些永久内存。

对 setProperties() 方法的调用稍微困难一些，因为共享 Properties 对象用于存储系统属性。如果应用程序在运行着 NHRT 的实时 JVM 中运行，那么对 setProperties 方法的调用必须在已于永久内存中创建的 com.ibm.realtime.ImmortalProperties 类或子类的实例中传递。使用该实例可确保通过使用 setProperties 方法所设置的所有属性都存储在永久内存中。

注: 调用 setProperties(null) 会致使通过缺省属性集在内部创建新的 ImmortalProperties 对象，这样会消耗更多永久内存。

对 getProperties() 方法的调用会返回已设置的对象或缺省属性对象（即 com.ibm.realtime.ImmortalProperties 对象）。为了最大限度地提高与调用 getProperties() 方法的现有代码的兼容性，ImmortalProperties 对象将序列化此对象，然后在标准 JVM 中进行反序列化。ImmortalProperties 序列化的缺省行为是序列化常规 Properties 对象，因为标准 JVM 不具有 ImmortalProperties 对象而反序列化会失败。为覆盖此缺省行为，ImmortalProperties 类提供 enabledReplacement(boolean) 方法，如果调用此方法false，那么会禁用该缺省行为。在此情况下，序列化操作会对 ImmortalPropeties 对象进行序列化，然后可以对此进行反序列化，并在实时JVM 中调用 System.setProperties 方法时使用所生成的对象。

注: 反序列化发生在永久内存中，这可能会消耗过多的这种有限资源。

安全管理器

为系统设置的安全管理器供 JVM 中所有类型的线程使用。基于这个原因，在运行 NHRT 的实时 JVM 中，安全管理器必须分配到永久内存中。实时 JVM 可确保命令行选项中指定的任何安全管理器都分配到永久内存中。安全管理器还可以通过对 System.setSecurityManager(SecurityManager) 方法的调用来设置。如果应用程序通过这种方法设置安全管理器，那么必须确保安全管理分配于永久内存，以便 NHRT 能够正确运行。

抛出的异常和安全管理器返回的任何对象都必须位于永久内存中（如果经过了高速缓存），或者分配到当前分配上下文中。

同步

MonitorControl 类及其子类 PriorityInheritance 管理同步，尤其是优先级逆转控制。这些类允许将优先级逆转控制策略设置为缺省设置或针对特定对象的设置。

WaitFreeReadQueue、WaitFreeWriteQueue 和 WaitFreeDequeue 类允许可调度对象（尤其是 NoHeapRealtimeThread 的实例）与常规 Java 线程之间进行无等待通信。

WaitFree 类以并发方式安全地访问 NoHeapRealtimeThread 实例与可调度对象（受垃圾回收延迟的影响）之间共享的数据。

非堆实时类安全性

在某些情况下，JSE API 的一些部分未必就能在非堆上下文中使用。对于堆线程与非堆线程之间共享的类施加了一些限制。请注意随 JVM 提供的可安全使用的类。

共享对象

非堆实时线程中运行的方法只要尝试装入对堆中对象的引用就会抛出 javax.realtime.MemoryAccessError。

第 56 页的图 3 是要避免的代码类型的示例:

```
/**
 * 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();
        }

    }
```

图 3. 访问堆对象引用的 NHRT 的示例

```
    /* 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("Message: " + NHRTError1.message);
        }
    }

}
```

图 4. 访问堆对象引用的 NHRT 的示例（图 1 续）

图 3 会产生 **javax.realtime.MemoryAccessError**:

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

如果某个对象要对非堆实时线程和标准 Java 线程均可访问，那么该对象必须分配到永久内存中。相似地，如果某个对象要对非堆实时线程和实时线程均可访问，那么该对象还可以保存在设置了作用域的内存区域内。

在第 56 页的图 3 中，对字符串“This on the heap.”的引用保存在类变量中。该变量对于 NHRT 可访问，因为所有类均已分配到永久内存中。或者，该字符串也可以传递到 NHRT 构造方法。

大多数对象都包含对其他对象的引用，因此在普通线程与 NHRT 之间共享此类对象时必须谨慎。典型示例是已分配到永久内存中的在普通线程与 NHRT 之间共享的 LinkedList。如果不够谨慎，那么标准线程可能会向 LinkedList 中引入位于堆中的对象。更应该关注的是 LinkedList 所分配以用于跟踪对象的数据结构被普通线程分配到堆中，从而很容易导致 NHRT 中发生 MemoryAccessError。

一些类无法在 NHRT 与其他线程之间安全共享，无论这些类的单独实例可能被分配到哪里均如此。这些是依赖于类变量中所存储对象的类，通常用于高速缓存目的。InetAddress 是对地址进行高速缓存的典型示例；如果用于调用 InetAddress 中特定方法的第一个线程正在堆中运行，那么将来 NHRT 无法安全地调用同样这些方法。

通过 NHRT 锁定对象

NHRT 必须避免与其他线程同步。请考虑以下场景:

- 低优先级的实时线程进入同步的块或方法，从而同步对象。
- 高优先级的 NHRT 在尝试同步同一对象时被阻止。
- 优先级继承致使实时线程暂时获得与 NHRT 相同的优先级。
- 然后垃圾回收以高于 NHRT 的优先级运行，并因此能够中断 NHRT。使用 NHRT 的原因是为了避免因垃圾回收而导致的中断，因此该场景否定了对 NHRT 的使用。

有时候 NHRT 和其他线程同步同一对象的情况不可避免，但您必须最小化这种可能性。在共享对象时请小心避免不必要的同步。

对安全类的限制

如果应用程序同时包含实时线程和非堆实时线程对象，那么一些注意事项适用。

- 非堆实时线程可以容忍因与实时线程的交互而导致的 MemoryAccessError。
- 非堆实时线程可能会被实时线程所引起的垃圾回收意外延迟。

非堆实时线程上导致的 MemoryAccessError

当这两种类型的线程都调用同一个类中的方法时，实时线程可能会用从堆中分配的对象来“污染”类的静态变量。非堆实时线程将在尝试访问这些堆对象时收到 MemoryAccessError。此污染还可能会发生在类的实例中。遗憾的是，这两个问题都很可能会在典型编码模式中出现，因此值得研究几个案例。

如果类正在执行耗时操作，那么通常会选择对结果进行高速缓存以提高后续操作的性能。该高速缓存通常是集合，例如锚定在类中的静态变量内的散列映射。堆上下文中运行的实时线程可以在此集合中存储堆对象，这不仅将该对象本身，还将基础结构对象添加到此集合，例如索引的一些部分。当非堆实时线程稍后尝试访问此集合时，即

使不会尝试访问另一个线程所添加的对象，也会尝试装入基础结构对象并因此收到 MemoryAccessError。随着类库的发展并进行调整以提高性能，这些高速缓存会变得更普遍。

类实例也可能会通过各种方式遭到堆对象的污染。考虑一个实例，它在永久内存中构建并因此可供这两种类型的线程访问。如果该对象的首次使用是被实时线程在堆上下文中使用，那么您可能会发现有辅助对象存储在原始对象的字段中。如果辅助对象在堆上下文中，那么非堆实时线程之后再次使用该对象将显示 MemoryAccessError。这些辅助对象可能不会始终在首次使用时添加，但数次使用之后，可能设计为提高频繁使用的方法的性能。

垃圾回收所延迟的非堆线程

必须为非堆线程分配高于其他线程的优先级才能避免因垃圾回收而被延迟。

此外，如果类包含任何同步的方法，那么调用此类方法的非堆实时线程都可能因垃圾回收而被意外延迟。该场景在第 57 页的『通过 NHRT 锁定对象』中进行了描述。

如果类包含任何同步的方法（无论是静态还是实例方法），那么调用此类方法的非堆实时线程都可能因垃圾回收而被意外延迟。导致该问题的原因是实时线程在访问同步的方法（静态或实例）的同时，非堆实时线程尝试调用另一个同步的方法（而该方法将阻止等待另一个线程的完成）。如果非堆实时线程的优先级高于实时线程，那么将提高实时线程的优先级。如果之后强制该线程等待垃圾回收中断，那么可能会发生优先级逆转，因为垃圾回收器线程的优先级高于最高优先级实时线程，而这可能没有当前被阻止等待进入同步的方法的非堆实时线程那么高。

解决此类问题的唯一方法是确保非堆实时线程永远不调用与其他线程类型共享的类或实例中的同步的方法。遗憾的是，不能始终根据方法特征符来清晰地判断方法是否同步；例如，它可能包含同步的块或调用同步的方法。

摘要

NoHeapRealtimeThread 类向实时环境增添了很多复杂性，并且不同类型的线程在环境中混合运行可能会导致大量问题。在应用程序开发期间，您必须谨慎地设计其中有不同类型线程共同使用类的区域。特别重要的是这些线程在 SDK 中如何使用类。由于分析的复杂性，对于 SDK 中提供的所有类是否都能够安全地进行此类共同使用，无法提供任何保证。然而，这些类中的一小部分已经过验证。最初，验证侧重于 MemoryAccessError 方面，得出了一列类，这些类已经过分析、测试和修改（如有必要）以确保它们可供非堆和其他类型的线程使用。

安全类

本部分列出了旨在供 NoHeapRealtimeThread 和其他线程类型安全使用的一组类。

主要问题聚焦于安全性的 MemoryAccessError 方面。下面的列表详细列出了可供同一 JVM 中所有三种线程类型使用的类。

注: 类的单独实例可能无法始终安全共享。

遵守以下规则可确保类可供所有线程类型安全使用:

- 实例必须在有意访问实例的线程可访问的内存区域中构建。
- 如果类具有公共的静态字段，请避免在这些字段中存储堆对象。

- 如果类具有公共的实例字段，请避免在这些字段中存储堆对象。

并非所有 IBM 提供的类都是 NHRT 安全型。以下包中含有 NHRT 安全型类:

- java.lang 包
- java.lang.reflect 包
- java.lang.ref 包（所有类）
- java.net 包
- java.io 包
- java.math 包

以下各表显示了这些包内的非 NHRT 安全型类:

表 6. java.lang 包中的非 NHRT 安全型类

类	方法
java.lang.ProcessBuilder	*
java.lang.Thread	getAllStackTraces()Ljava.util.Map;
java.lang.ThreadGroup	*
java.lang.ThreadLocal	*
java.lang.InheritableThreadLocal	*

表 7. java.lang.reflect 包中的非 NHRT 安全型类

类	方法
java.lang.reflect.Proxy.*	*

表 8. java.net 包中的非 NHRT 安全型类

类	方法
java.net.SocketPermission.*	newPermissionCollection()Ljava.net.SocketPermissionCollection;

表 9. java.io 包中的非 NHRT 安全型类

类	方法
java.io.ExpiringCache	*
java.io.SequenceInputStream	*
java.io.FilePermission	newPermissionCollection()Ljava.io.FilePermissionCollection;
java.io.ObjectInputStream	*
java.io.ObjectOutputStream	*
java.io.ObjectStreamClass	*

表 10. java.math 包中的非 NHRT 安全型类

类	方法
java.math.BigInteger	*

这些包可能包含具有非安全型类的子包。例如，以下类是非 NHRT 安全型类:

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

- java.lang.instrument.*

即使类被视为 NHRT 安全型类，该类可能仍不适合在 NHRT 中使用。应用程序开发者必须依具体情况来确定类的实时需求，而不考虑类是否为 NHRT 安全型类。

JVM 之间的类数据共享

Java 虚拟机 (JVM) 使您能够通过将类数据存储在磁盘上的内存映射的缓存文件中，在 JVM 之间共享类数据。

当多个 JVM 共享一个高速缓存时，共享会降低总体虚拟存储器耗用。当创建高速缓存后，类共享还会降低 JVM 的启动时间。共享类高速缓存独立于任何活动 JVM，并且持续存在直到它被销毁。共享的高速缓存可以包含:

- 引导程序类
- 应用程序类
- 描述了的元数据
- 提前（AOT）编译的代码

IBM WebSphere Real Time for RT Linux 可以通过非实时和实时方式来使用共享类高速缓存，但高速缓存格式、创建和填充方法会有所不同。实时方式的高速缓存与非实时方式的高速缓存不兼容。在非实时方式中，创建和填充高速缓存的方法与标准 JVM 相同。这表示 JVM 在其运行应用程序时创建和填充高速缓存，这一过程对用户是透明的。在实时方式中，使用 **-Xrealtime** 选项时，必须通过 **admincache** 使用 **-populate** 选项来创建和预填充共享类高速缓存。以实时方式运行的应用程序可以读取预填充高速缓存中的内容，但无法修改其内容。

使用 **admincache** 工具创建、填充和销毁高速缓存。

要使应用程序能够使用共享类高速缓存，请向其命令行中添加 **-Xshareclasses** 选项。由于实时方式高速缓存是只读的，因此 **-Xshareclasses** 的部分非实时方式子选项在实时方式中不可用。

有关进一步信息，请参阅第 39 页的『使用 admincache 工具』、第 85 页的『非实时方式下 JVM 之间的类数据共享』和第 129 页的『共享类诊断数据』。

使用共享类高速缓存运行应用程序

要使用共享类高速缓存运行应用程序，请在命令行中使用 **-Xshareclasses** 选项。

表 11 显示了在实时方式中，使用 **-Xshareclasses** 选项运行应用程序时可用的子选项。

表 11. 以实时方式运行应用程序时可用的子选项

选项	含义
cacheDir=<*directory*>	设置在其中读写共享类高速缓存数据的目录。缺省情况下，<*directory*> 为 /tmp/javasharedresources。该目录名称必须与创建高速缓存的 admincache 命令中使用的 **-cacheDir** 选项中指定的目录名称相匹配。

表 11. 以实时方式运行应用程序时可用的子选项 (续)

选项	含义
name=<*name*>	要使用的共享类高速缓存的名称。该名称必须与创建高速缓存的 admincache 命令中使用的 **-cacheName** 选项中指定的名称相匹配。该名称不能超出 53 个字符。
none	显式禁用类共享。可以添加到命令行的结尾来禁用类数据共享。此子选项覆盖命令行中先前找到的类共享参数。
nonfatal	不管错误或警告，始终启动 JVM。即便在类数据共享失败的情况下仍允许 JVM 启动。如果类数据共享失败，JVM 的典型行为是拒绝启动。如果选择 **nonfatal** 并且共享类高速缓存未能初始化，那么 JVM 尝试以只读方式连接到高速缓存。如果该尝试失败，那么 JVM 会在没有类数据共享的情况下启动。
silent	禁止所有输出消息。关闭所有共享类消息，包括错误消息。将显示不可恢复错误消息，这些消息会阻止 JVM 初始化。
verbose	启用详细输出，提供有关共享类高速缓存的总体状态和更多详细的错误消息。
verboseAOT	当高速缓存中发现经过编译的 AOT 代码时启用详细输出，例如在 AOT 方法装入请求的过程中。
verboseHelper	启用 Java 助手 API 的详细输出。该输出显示类装入器如何使用助手 API。
verboseIO	启用类装入请求的详细输出。该选项提供了有关高速缓存 I/O 活动的详细输出，其中列出了有关所找到类的信息。

要确保这些选项都正确无误，请在 admincache 中使用 **-printvmargs** 选项（请参阅 **-printvmargs** 以获取更多信息）。**nonfatal** 选项不适合于常规使用，因为它强制 JVM 忽略有关共享类高速缓存的警告和错误。**none** 选项显式地禁用类共享，等同于在命令行中省略 **-Xshareclasses** 选项。

有关 **-Xshareclasses** 子选项的更多信息信息，请参阅类数据共享命令行选项。

使用 Metronome 垃圾回收器

Metronome 垃圾回收器用于取代 WebSphere Real Time for RT Linux 中的标准垃圾回收器。

控制处理器利用率

您可以限制可供 Metronome 垃圾回收器使用的处理能力大小。

您可以使用 **-Xgc:targetUtilization=N** 选项限制 Metronome 垃圾回收器所使用的 CPU 量，以控制该垃圾回收器的垃圾回收。

例如:

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

该示例指定应用程序在每 60 毫秒中运行 80% 的时间。其余 20% 的时间用于垃圾回收。如果给予 Metronome 垃圾回收器足够的资源，那么它可保证达到利用率级别。当堆中的可用空间量降低到动态确定的阈值以下时，会开始进行垃圾回收。

调整 Metronome 垃圾回收器

您可通过控制应用程序使用的内存量来调整实时环境。例如，使用 **-Xmx**、**-Xgc:immortalMemorySize=size**、**-Xgc:scopedMemoryMaximumSize=size** 和 **-Xgc:targetUtilization=N** 选项。

- 使用 **-Xmx** 选项来限制堆的大小。

  选定的值用作堆大小的上限，从而反映随时间推移的可能使用情况。选择的值过低会提高垃圾回收的频率，并导致整体吞吐量下降（虽然它降低了内存占用量）。要实现优良的实时性能，请避免页面调度。一般情况下需要确保机器上所有正运行的进程的占用量不会超出物理内存大小。

- 使用 **-Xgc:immortalMemorySize=size** 选项来控制永久内存区域的大小。

  必须仔细分析永久内存的使用。“理想的”应用程序在启动期间使用永久内存，随后会停止使用。如果继续分配永久对象，那么应用程序可以继续运行，直至永久内存耗尽为止。可将以下内容添加到代码来获取当前使用情况:

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

  。

- 使用 **-Xgc:scopedMemoryMaximumSize=size** 选项来确保应用程序不会请求过量的作用域内存。使用该选项进行诊断（而非调整）。

- 设置 **-Xgc:targetUtilization=N** 选项以确保在最坏的情况下（堆对象的最大分配率），垃圾回收器可以高于应用程序生成垃圾的速度对其进行回收。

  通常情况下，缺省值足够使用，但是可通过以下方式来改善应用程序的性能: 将利用率提高到回收器能够以略高于应用程序生成垃圾的速度来回收垃圾。

- 使用 **-Xgcthreads <n>** 选项来创建额外的线程，从而以并行方式运行垃圾回收。

  缺省值为使用一个线程。如果您的工作负载具有较高的垃圾生成率，且在提供 CPU 周期的对称式多处理器上运行，那么将该参数设置为 >1 可改善性能。

  注: 将该参数设置过高会对吞吐量造成负面影响。

Metronome 垃圾回收器限制

本主题描述了影响 Metronome GC 策略的任何已知问题或限制。

x86 平台上的 AESNI 支持

x86 体系结构中 AESNI 指令的软件开发当前不支持与 Metronome GC 策略一起使用。

垃圾回收期间的长暂停时间

在极少数情况下，您在垃圾回收期间可能会遇到比预期更长的暂停。在垃圾回收期间，会使用根扫描进程。垃圾回收器会从已知的活动引用开始走遍整个堆。这些引用包括:

- 活动线程调用堆栈中的活动引用变量。
- 静态引用。
- 永久和设置了作用域的内存中的所有对象引用。

为查找应用程序线程的堆栈上的所有活动对象引用，垃圾回收器会扫描该线程的调用堆栈中的所有堆栈帧。将在不可中断的步骤中扫描每个活动线程堆栈。因此，扫描必须在单次 GC 暂停内进行。

其效果是如果您的一些线程具有非常深的堆栈，那么系统性能可能会比预期差，这是由回收周期开始时的延长了的垃圾回收暂停所致。

永久内存以递增方式处理。所有其他设置了作用域的内存区域都在一个不可中断的原子步骤中进行处理。因此，大量使用设置了作用域的内存区域可能会导致系统性能比预期差，这是由根扫描在处理设置了作用域的内存时的延长了的垃圾回收暂停所致。

第 6 章 开发应用程序

关于编写实时应用程序（包括代码样本）的重要信息。

- 『编写 Java 应用程序以利用实时环境』
- 第 75 页的『样本应用程序』
- 第 82 页的『样本实时散列映射』
- 第 82 页的『使用 Eclipse 开发 WebSphere Real Time for RT Linux 应用程序』

编写 Java 应用程序以利用实时环境

这些示例介绍如何利用实时环境。它们涵盖的示例极为全面，既包括最简单的示例，如在实时环境中运行未对代码进行任何修改的 Java 应用程序，也包含更为复杂的规划和编写非堆实时线程的过程。会为您提供原因，帮助您确定哪种方法最适合自己的应用程序。

编写实时应用程序介绍

无需编写详细的非堆实时应用程序来利用实时技术的功能。对现有代码稍作更改即可使用某些优势。

对于应用程序编程人员，您可执行此处所述步骤来利用 WebSphere Real Time for RT Linux:

1. 您可在实时 JVM 中运行标准 Java 应用程序，从而获取 Metronome 垃圾回收的优势，同时显著提升应用程序运行时的可预测性。
2. 预编译代码完成后添加 **-Xnojit** 选项，以使用提前 (AOT) 编译器。请参阅第 47 页的『将预编译的 JAR 文件存储到共享类高速缓存中』。
3. 将应用程序中的 java.lang.Thread 替换为 javax.realtime.RealtimeThread。与 AOT 选项相比，您会看到少许改进。

   使用实时线程的主要优势是能够对指定给每个线程的优先级进行控制。还可定期运行实时线程。要利用这些优势，必须准备对应用程序本身进行更改。
4. 规划并编写特定应用程序以使用实时线程和异步事件处理程序来处理计时器或外部事件。考虑以下三项因素:
   - 规划分配给实时线程的优先级
   - 确定使用哪些内存区域来存放对象
   - 与事件处理程序进行通信
5. 规划并编写特定应用程序以使用非堆实时线程。非堆实时线程是实时线程的扩展，您必须考虑分配的优先级和内存区域。一般来说，仅当应用程序必须以与 GC 暂停时间（亚毫秒）相当的时间处理事件时才执行此步骤。请勿低估使用非堆实时线程进行开发的复杂性。

第 66 页的图 5 显示先前所述的步骤。

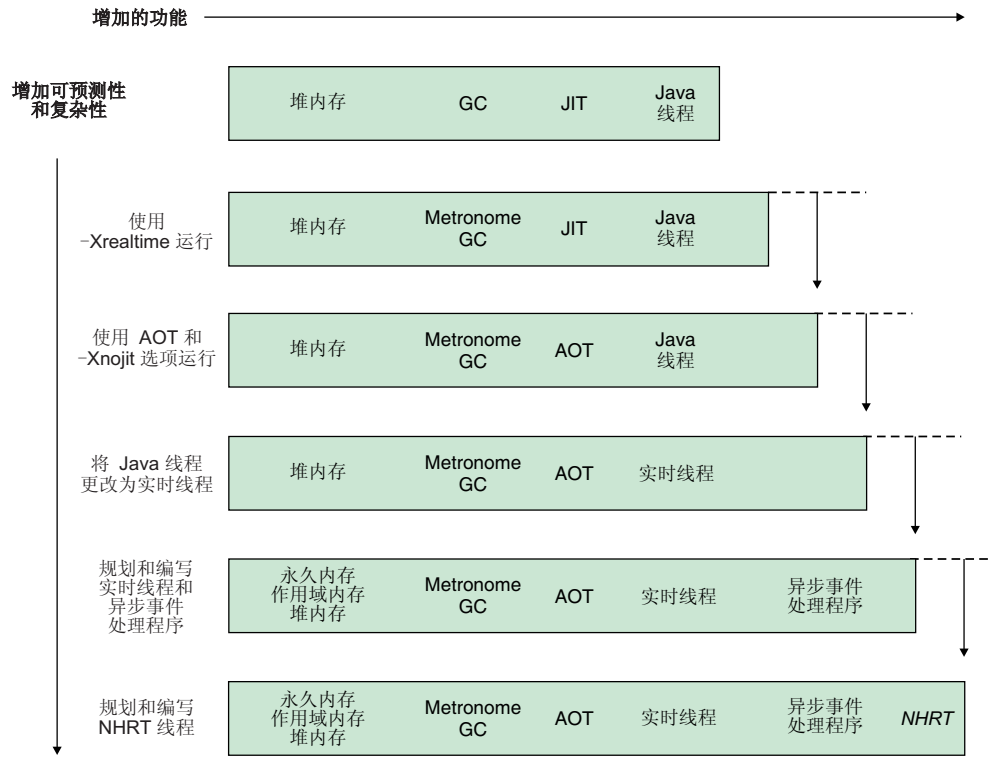

图 5. RTSJ 功能与增强的可预测性进行比较。

规划 WebSphere Real Time for RT Linux 应用程序

准备编写实时 Java 应用程序时，必须考虑是使用 Java 线程、实时线程还是使用非堆实时线程。此外，您还可以确定线程将要使用哪个内存区域。

关于此任务

规划应用程序时，以下步骤描述您需要做的决策:

过程

1. 确定任务。
2. 确定计时周期:
   - 响应大于 10 ms，选择 Java 线程，只采用 Metronome 垃圾回收器。

     这些线程仅使用堆内存进行存储。其劣势在于垃圾回收会中断应用程序，但是由于它由 Metronome 垃圾回收器控制，所以中断的长度和计时是可以预测的。
   - 响应小于 10 ms，选择实时线程。

     实时线程可放置在堆、作用域或永久内存中。使用实时线程的优势如下所述:
     – 它们可以高于标准 Java 线程的优先级运行。
     – 垃圾回收由 Metronome 垃圾回收器进行控制。但是，垃圾回收器会以高于实时线程的最高优先级的优先级运行，并且会中断程序运行。
   - 响应小于 1 ms，选择非堆实时线程。

非堆实时线程的优先级可设置为高于垃圾回收的优先级，因此不会被 Metronome 明显中断。只有 Metronome 警报线程以顶级优先级运行，且使用极少量 CPU。

3. 确定应用程序是否需要异步事件处理程序。该需求取决于程序的结构。
   - 时间响应小于 10 ms，选择实时线程。
   - 时间响应小于 1 ms，选择非堆实时线程
4. 确定线程优先级。 一般来说，时间段越短，优先级越高。
5. 确定内存特征。
   - 如果任务具有可变或高分配率，可能会对 GC 产生极大影响，请考虑强制指定比率限制（使用 MemoryParameters），或考虑分配到作用域内存区域。
   - 如果任务在计算期间生成大量临时数据，请考虑使用作用域内存区域。
   - 如果任务在启动时生成 JVM 生命周期所需的某些数据，请考虑使用永久内存。在 JVM 生命周期外继续创建对象的情况下请避免使用永久内存。
   - 如果任务需要进行通信（尤其当某个任务在非堆实时线程下运行时），考虑使用作用域内存区域进行通信。
   - 如果任务正以非堆实时线程方式运行，考虑构建作用域内存区域（例如 LTMemory）以包含非堆线程、运行时参数和可能用于与任务进行通信的无等待队列。LTMemory 对象必须在永久或其他作用域内存中进行构建，从而避免在非堆线程尝试对其进行引用时出错。
6. 如果已确定应用程序的结构和内容，请修改运行时选项以改善应用程序的性能。接下来的步骤描述如何操作:
   a. 最初测试应用程序期间，使用 **-Xmx**、**-Xgc:immortalMemorySize=size** 和 **-Xgc:scopedMemoryMaximumSize=size** 选项在堆、作用域和永久内存中设置大量空间。

      注: 利用 Metronome GC 时，最初和最大的堆大小必须完全相同，原因是 Metronome GC 不会增加堆的大小。增加堆大小是一项非确定操作。
   b. 使用 **-verbose:gc** 选项来确定使用的内存量。
   c. 修改 **-Xgc:targetUtilization** 选项以为垃圾回收的执行提供充足时间。 缺省值为 70%，该百分比适用于大多数应用程序。请确保垃圾回收率略高于分配率。
   d. 使用 **-Xmx** 选项设置堆内存的实际大小。

修改 Java 应用程序

要编写使用实时 Java 功能的代码，可将线程的 `java.lang.Thread` 替换为 `javax.realtime.RealtimeThread`。

开始之前

该示例基于 `demo/realtime/sample_application.zip` 文件中的 JavaRadar.java 类。

关于此任务

实时线程的编程模型类似于标准 Java 应用程序的模型。但是将实时线程添加到程序这一拙劣方法不能充分利用 WebSphere Real Time for RT Linux 的功能。要想充分利用产品的所有功能，必须修改线程，以使其拥有与其关联的优先级，同时还要考虑使用何种内存区域。

只更改线程的类只能获取极少一部分有关应用程序的优势功能，原因在于实时线程的缺省优先级大于标准 Java 线程。

要将 JavaRadar 更改为 RealtimeThread，请更改类，将其从 Thread 扩展到 RealtimeThread。

将 java.lang.Thread 替换为 javax.realtime.RealtimeThread

样本应用程序中的 JavaRadar 类扩展 java.lang.Thread。例如:

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

要将该 Java 线程变为实时线程，需要重新定义该类定义，如下所示:

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

编写实时线程

目前为止，您只修改了应用程序；现在可编写一些代码。您可以编写使用实时线程的应用程序，以利用实时优先级级别和内存区域。

开始之前

该示例基于 demo/realtime/sample_application.zip 文件中的 JavaRadar.java、RTJavaRadar.java 和 RTJavaControlLauncher.java 类。

该样本显示如何将永久内存与第 67 页的『修改 Java 应用程序』中所述的相同样本结合使用。

关于此任务

实时线程的编程模型类似于标准 Java 应用程序的模型。

使用实时线程的优势如下所述:

- 完全支持实时线程上的 OS 级别线程优先级。
- 作用域内存或永久内存区域的使用。
  - 借助作用域内存，您可在不影响垃圾回收的情况下显式控制内存的释放。
  - 通过非堆实时线程，您可以使用永久内存来避免垃圾回收暂停。
  - 就像存储在堆内存中的实时线程一样，这些引用堆中对象的实时线程受垃圾回收的影响。
  - 非堆实时线程无法引用堆内存中的对象，因此，它们不受垃圾回收的影响。

在第 69 页的表 12 中，以 SimulationThread 具有最高优先级为基础分配优先级，原因是它表示外部事件，且决不允许被程序中的任何项抢占。RadarThread 需要快速响应来自控制器的 ping。响应速度越快，月球登陆器的高度测量就越精确。ListenThread 还必须快速响应来自控制器的命令，响应优先级仅次于 RadarThread。

由于模拟以服务器方式运行，因此这三个线程位于作用域内存中。服务器运行模拟后，可退出作用域内存区域，然后再重新进入以等待模拟另一次运行。服务器使用作用域内存，因此可自行重置。

由于 RTJavaRadarthread 使用该时间来计算高度，因此它对计时更为敏感，所以拥有最高优先级的控制器线程。由于正以 NHRT 方式运行，并且控制器仅运行一次，且 JVM 退出时释放内存，因此它是永久的。

对于 RTJavaControlThread 和 RTJavaEventThread 而言，时间约束并不十分严格，因此允许使用堆内存。

RTLoadThread 不执行月球登陆器的有用功能。但是 RTLoadThread 演示可以低于其他线程的优先级执行大量内存分配和释放，同时并不会对更高优先级的线程性能产生影响。

表 12. 样本应用程序中内存区域与线程的关系

内存	线程	优先级
作用域	demo.sim.SimulationThread	38
	demo.sim.RadarThread	37
	demo.sim.SimulationThread.ListenThread	36
永久	demo.controller.RTJavaRadarThread	15
堆	demo.controller.RTJavaControlThread	14
	demo.controller.RTJavaEventThread	13
作用域和堆	demo.controller.RTLoadThread	12

示例

demo.sim.SimulationThread 中的代码显示已设置优先级 38。 **1** 该代码行检索 JVM 中可用的最大优先级。

```
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
```

demo.sim.SimLauncher 中的代码显示已定义作用域内存。 **2** 显示 LTMemory 的分配，这是线性时间内分配内存的作用域内存区域。

```
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;
```

simMemArea 引用的 ScopedMemoryArea 对象分配到永久内存中，原因是 NHRT 必须能够引用代表 ScopedMemoryArea 的对象。在堆上分配会导致 NHRT 构造函数抛出 IllegalArgumentException，原因是其内存区域参数位于堆中。

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

demo.controller.RTJavaControlLauncher 中的代码显示 RTJavaRadar 已定义并使用永久内存。由于 RTJavaRadar 在控制器 JVM 的整个生命周期内运行一次，因此只在启动时分

配内存；它可在永久内存中安全运行。由于控制器无需先进入作用域内存区域即可访问 RTJavaRadar 方法，因此有益于应用程序设计。因为控制器编写为以常规 Java 和实时 Java 运行，因此进入作用域内存区域是非常困难的。

```
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;
```

编写异步事件处理程序

异步事件处理程序对计时器事件或线程外部发生的事件作出反应；例如，来自应用程序界面的输入。在实时系统中，这些事件必须在您为应用程序设置的截止期限内作出反应。

开始之前

该示例基于 demo/realtime/sample_application.zip 文件中的 RTJavaEventThread.java 和 RTJavaControlLauncher.java 类。

关于此任务

在样本应用程序中，事件线程等待来自模拟（发出坠毁或着陆信号）的事件。在该线程的实时版本中使用 AsyncEvent 机制。这些事件用于打印输出相应的状态消息，并使控制器退出。

RTJavaEventThread 具有两个已定义的异步事件。这两个事件均无参数。

```
public class RTJavaEventThread extends RealtimeThread {

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

这些事件创建并注册两个异步事件处理程序:

```
    /**
     * 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);
}
```

如果接收到坠毁或着陆消息，就会触发相应的异步事件处理程序，从而释放 Runnable 对象。

```
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;
}
```

结果

RTJavaControlLauncher.java 包含对 addLandHandler 和 addCrashHandler 方法的调用。传递的 Runnable 对象导致将消息打印输出到控制台，当关联的异步事件处理程序被触发时控制线程停止。请参阅 RTJavaEventThread.java，以了解触发点。

```
// 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!");
    }
});
```

编写 NHRT 线程

要将非堆实时线程 (NHRT) 添加到 Java 应用程序，可使用该教程来开发或修改您自己的程序。

开始之前

该示例基于 demo/realtime/sample_application.zip 文件中的 SimulationThread.java 和 SimLauncher.java 类。

关于此任务

demo.sim.SimulationThread 类是演示应用程序中模拟的一部分。可将其视为真实世界的替代方法，因此在运行时不会从系统的其余部分中断。线程创建为 NoHeapRealtimeThread，且具有最高可用优先级，目的是确保线程不会被垃圾回收或系统中的其他线程所中断。

在 SimulationThread 中，以下构造函数调用超级构造函数“NoHeapRealtimeThread (SchedulingParameters scheduling, MemoryArea area)”，在此之前分别设置其 SchedulingParameters 和 ReleaseParameters:

```
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;
    }
```

模拟中的其他活动线程也将创建为非堆实时线程 (NHRT)，但是其优先级稍低。请参阅第 68 页的『编写实时线程』，了解优先级的分配策略。

模拟具有无限期运行的选项，以便在模拟完成后重新启动。由于模拟由 NHRT 组成，您可以选择 ScopedMemory 或 ImmortalMemory。样本应用程序将 ScopedMemory 用于模拟，因为退出在模拟完成时分配的 ScopeMemoryArea 然后重新进入以等待下一次运行比较合适。在这种情况下，不会使某次运行的状态持续至下一次运行。

大多数类都属于“NHRT 安全”；但是大部分类可以非“NHRT 安全”方式运行。例如，如果 DatagramSockets 保留在永久内存或外部作用域的内存区域中，那么由于其设计不支持跨内存区域，因此可能会出现问题。样本应用程序只使用一个 ScopedMemory 区域来防止此类问题。

RTSJ 中的内存分配

在 RTSJ 中，您可以使用多种方法在某个特定的内存区域中分配对象，并且大多数情况下选择哪种方法并没有明显的区别。

每种方法均有某些特征（因 RTSJ 实施而有所不同），并使性能或最终的内存占用量产生差异。该部分概述了可用选项及何时用作分配对象最佳选项的建议情况。

静态初始化程序

在永久内存区域中分配对象的最简单方式是在静态初始化程序中进行分配。其优势在于无需处理不断变化的内存上下文，但是该模式的适用环境相当有限。当耗用的永久内存量限于对象本身所需时该方法很有效。

MemoryArea.newInstance(Class c)

如果线程位于内存上下文中，且希望在其他区域（该区域必须已位于线程的作用域堆栈中）中分配对象，那么该方法很简单。其优势在于您只需访问要实例化的类，但是 newInstance 方法必须构建相应的构造函数。如果指定类的对象必须极少进行分配，那么

该模式是最佳选择，但是其内存使用率却较高。

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

再次重申，如果线程位于内存上下文中，且希望在其他上下文（该上下文必须位于线程的作用域堆栈中）中分配对象，那么该方法很简单。在这种情况下，必须传递 Constructor 和某些参数，并确保 Constructor 在当前内存上下文中有效。由于 newInstance 方法无需构建 Constructor，因此内存使用率低于 newInstance(Class c)，如果对象分配较为频繁，希望提前分配构造函数并将其存储在某处（如 ImmortalMemory），那么该模式更为适用。

MemoryArea.enter(Runnable r) followed by new <class>()

该方法使得指定的 MemoryArea 成为分配的新缺省值，同时无需反映并除去伴生的 Constructor 对象。因此在以下情况中该方法最为适用: 由于没有出现超出对象本身的额外内存使用情况而要创建多个对象。仅当任意线程的作用域堆栈中的指定区域尚未处于活动状态时该方法才起作用。由于通常需要在 Runnable 或整个静态或实例字段中传递参数，因此构建 Runnable 内存区域的需求使得该方法要比使用 newInstance 更为复杂。

MemoryArea.executeInArea(Runnable r) followed by new <class>()

再次重申，该方法使得指定的 MemoryArea 成为分配的新缺省值，同时无需反映并除去伴生的 Constructor 对象。因此在以下情况中该方法最为适用: 由于没有出现超出对象本身的额外内存使用情况而要创建多个对象。如果指定的区域已位于当前线程的作用域堆栈中，那么您可以使用该方法，该方法比 MemoryArea.enter 更为灵活。构建 Runnable 的需求使得该方法要比使用 newInstance 更为复杂，原因是通常需要在 Runnable 或整个静态或实例字段中传递参数。

Class.newInstance()

该方法在当前内存区域中构建新实例，因此必须与 MemoryArea.enter 或 executeInArea 结合使用。无超出对象本身的额外内存使用情况。

使用高精度计时器

实时时钟为与标准 JVM 关联的时钟提供更高精度。

开始之前

该示例基于 `demo/realtime/sample_application.zip` 文件中的 RTJavaRadar.java 类。

关于此任务

常规 Java 处理时钟和计时器的能力受限。Real-Time Specification for Java 允许绝对时间精确到纳秒级精度，并支持挂钟时间具有足够的量级。
javax.realtime.HighResolutionTime 及其子类用于表示时间，包括毫秒和纳秒两个部分。

WebSphere Real Time for RT Linux 借助底层操作系统的支持来提供高精度时间。Current® Linux 内核提供最优可实现 4 毫秒保证精度的时钟。随 WebSphere Real Time for RT Linux 提供的 Linux 补丁提供精确到 1 毫秒的时钟。

RTJavaRadar 类演示高精度计时器的使用:

- 1 获取实时时钟。
- 2 获取当前绝对时间。
- 3 获取时间的纳秒组成部分。实时时钟的准确性表示适合使用纳秒。
- 4 获取 ping 之前和之后的时间。
- 5 返回登陆器下降的速度。
- 6 使线程在执行其他迭代前等待 5 毫秒。

```
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.
               }
           }
```

先前的代码可与 JavaRadar 类中的以下标准 JVM 代码进行比较:

```
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;
    }
```

样本应用程序

样本应用程序使用一系列示例来演示 WebSphere Real Time for RT Linux 的功能，您可以使用这些功能来提高 Java 程序的实时特征。

样本应用程序的源文件位于 `demo/realtime/sample_application.zip` 文件中。

样本包含两个主要组件:

- **模拟**，月球登陆器的简单示例。登陆器的位置由其与地面间高度（源于时控脉冲）及其与着陆区域的水平距离决定。请参阅第 76 页的图 6。

  模拟类 使用 非堆实时线程 (NHRT) 编写，且在本文档中未对其进行进一步修改。
- **控制器**，将命令发送给模拟。控制器发送雷达 ping，目的是判断登陆器的高度，并基于该信息来控制登陆器的下降速度。控制器还接收来自登陆器的信息流；例如，登陆器与着陆区域的距离。

  控制器最初以标准 Java 编写。在第 67 页的『修改 Java 应用程序』中，它已开发成为实时 Java 程序。

根据登陆的结果信息，向控制器发送两条消息之一：坠毁或着陆。

使用样本应用程序，您可以执行这些操作:

- 同时运行模拟和控制器来演示实时和标准 Java 类一起运行的组合。有关更多信息，请参阅第 77 页的『构建样本应用程序』和第 77 页的『运行样本应用程序』，其中您还将看到期望来自样本应用程序的输出。

  注: 您可使用 LaunchBoth 类来同时启动模拟和控制器。
- 比较使用 Metronome 垃圾回收器 和标准垃圾回收器间的差异。有关信息，请参阅第 77 页的『无 Real Time 的情况下运行样本应用程序』和第 78 页的『运行带有 Metronome 垃圾回收器的样本应用程序』。
- 使用提前 (AOT) 编译器运行应用程序。有关信息，请参阅第 79 页的『使用 AOT 的同时运行样本应用程序』。

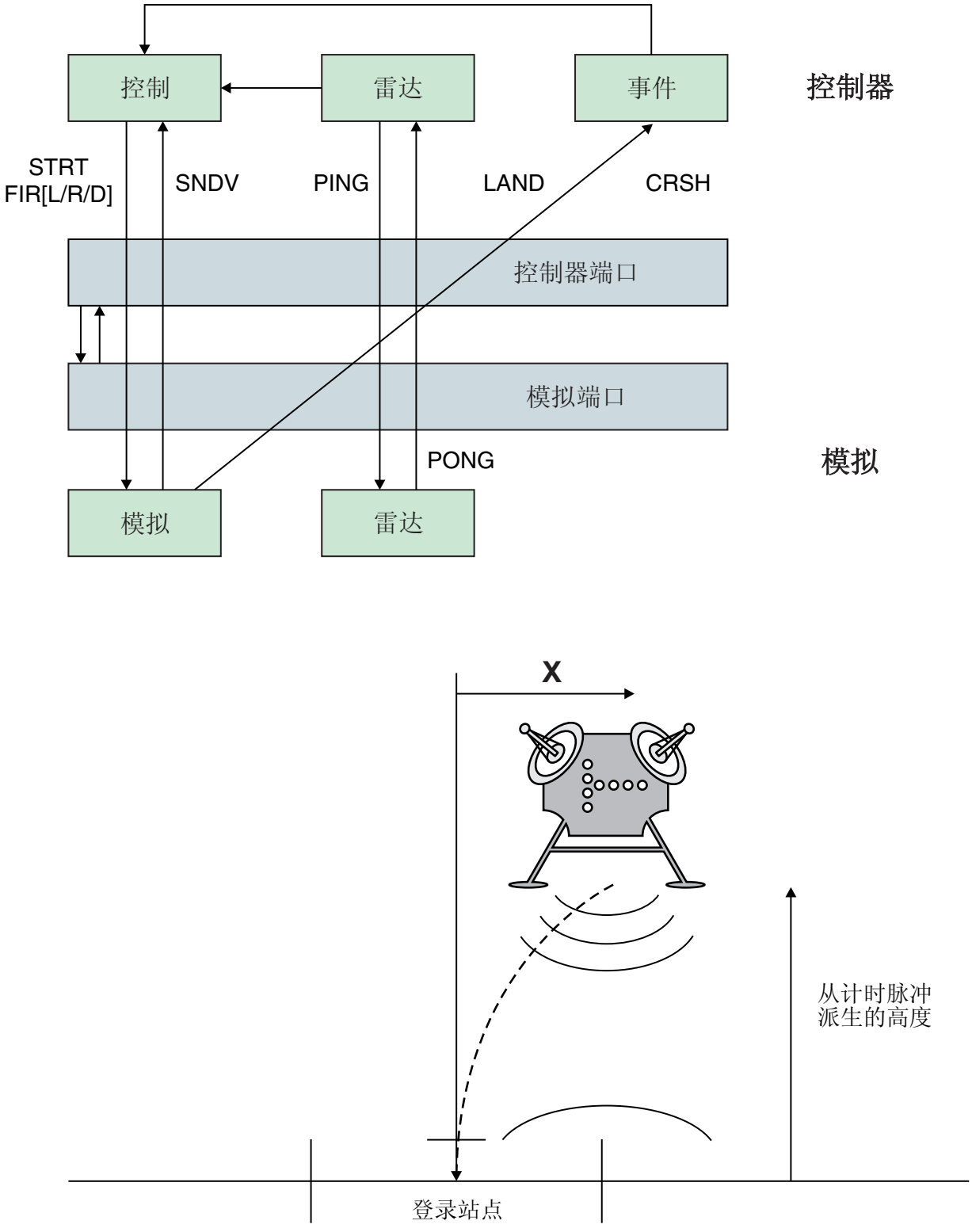

图 6. 月球登陆器图

该图显示本样本中提供的模块间的关系。控制器和模拟组件通过控制器和模拟端口相互通信。控制器组件有三个线程: Control、Radar 和 Events。模拟组件有两个线程:

Simulation 和 Radar。Control 线程启动模拟，随后将触发消息发送给模拟组件以控制登陆器的方向。Simulation 线程发回表示其状态的值。每个组件的 Radar 线程将 PING 和 PONG 消息发送给彼此。Control 线程使用这些消息交换间的时间来计算登陆器的高度。Simulation 线程还将相应的结束事件（坠毁或着陆）发送给 Events 线程。Events 线程将结束事件传递回 Control 线程，后者随后结束模拟。

构建样本应用程序

提供有样本应用程序源代码以用于指导。在运行前需要对 Java 源代码进行解压和编译。

过程

1. 创建工作目录。
2. 将样本应用程序抽取到您的工作目录:

   ```
   unzip sample_application.zip
   ```

3. 创建一个新目录进行输出:

   ```
   mkdir classes
   ```

4. 编译源。
   a. 生成文件列表:

      ```
      find -name "*.java" > source
      ```

   b. 编译源:

      ```
      javac -Xrealtime -Xlint:deprecated -g -d classes @source
      ```

   c. 创建类文件的 JAR 文件:

      ```
      jar cf demo.jar -C classes/ .
      ```

下一步做什么

现在可以运行样本应用程序。

运行样本应用程序

WebSphere Real Time 提供标准 JVM 和实时 JVM，以 **-Xrealtime** 命令行参数启动。

样本应用程序由两个部分组成，以便在单独的 JVM 中运行:

- Simulation，只能在实时 Java 中运行。
- Controller，可在非实时或实时 Java 中运行。

在多种方式下运行 Controller 代码体现了 IBM Real-Time Java 技术的优势。

无 Real Time 的情况下运行样本应用程序

该过程中，在不使用 IBM WebSphere Real Time 的情况下运行样本应用程序。

开始之前

要运行样本应用程序，首先必须构建样本源代码。请参阅 『构建样本应用程序』以获取更多信息。

过程

1. 启动模拟:

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

在该命令中，*<port>* 是工作站的释放端口。

2. 启动控制器:

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

在该命令中，*<host>* 是运行模拟的工作站主机名，*<port>* 是前一步骤中指定的端口。

结果

应用程序生成一条消息，该消息表明模拟和控制器已启动:

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

控制器中值的部分定点样本打印输出到控制台:

```
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
```

模拟停止前，会发出事件摘要消息:

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

除定点样本和事件摘要消息外，控制器还在同一目录下生成名为 `graph.svg` 的图。该图包含定点样本。 该图显示使用标准非实时 JVM 运行应用程序时，垃圾回收暂停对 JavaRadar 线程的影响。代表雷达高度的数据具有峰值。峰值是由影响 Controller 应用程序的标准垃圾回收暂停导致的。在某些运行情况下，垃圾回收暂停时间足够长可导致出现故障，从而显示消息:

```
CRASH!
```

要查看垃圾回收导致的暂停时间，将 **-verbose:gc** 选项添加到控制器启动命令:

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

运行带有 Metronome 垃圾回收器的样本应用程序

您可通过添加 -Xrealtime 选项以在实时环境中运行标准 Java 应用程序，无需重新编写代码。该选项同时启用实时 Java 语言功能和 Metronome 垃圾回收器。

开始之前

要运行样本应用程序，首先必须构建样本源代码。请参阅 第 77 页的『构建样本应用程序』以获取更多信息。

过程

1. 启动模拟:

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

   在该命令中，*<port>* 是工作站的释放端口。

2. 启动控制器:

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

   在该命令中，*<host>* 是运行模拟的工作站主机名，*<port>* 是前一步骤中指定的端口。在同一工作站上同时运行两个 JVM 会导致确定性行为降低。请参阅 第 22 页的『注意事项』以获取更多信息。

结果

应用程序运行并生成若干输出，包括:

1. 表明模拟和控制器已启动的消息。
2. 控制器值的定点样本。
3. 同一目录中名为 `graph.svg` 的图，包含定点样本。
4. 事件摘要消息。

在使用 Metronome 垃圾回收运行应用程序时，定点样本和对应的图显示:

- “雷达高度”数据中无峰值。
- “真实高度”数据的精确跟踪。

原因是 Controller 代码目前正在垃圾回收暂停时间较短的情况下运行。

Metronome 垃圾回收暂停十分频繁，但是持续时间通常少于 1 毫秒。非实时垃圾回收暂停并不常见，但是其持续时间通常长达几十或几百毫秒。将 **-verbose:gc** 选项添加到 Controller 运行命令即可看到暂停间的差异。

请参阅第 124 页的『使用 verbose:gc 信息』，以了解有关详细垃圾回收输出的更多信息。

使用 AOT 的同时运行样本应用程序

该过程在使用提前（Ahead-of-Time，AOT）编译器的同时，在实时环境中运行标准 Java 应用程序，且无需重新编写代码。将该样本与使用 JIT 编译器运行的相同应用程序进行对比。

请参阅第 36 页的『在 WebSphere Real Time for RT Linux 中使用经过编译的代码』，以获取有关提前编译的更多信息。

开始之前

要运行样本应用程序，首先必须构建样本源代码。请参阅 第 77 页的『构建样本应用程序』以获取更多信息。

关于此任务

提前编译器先将 Java 应用程序编译为本机代码，然后运行。由于即时编译不会造成中断，因此您可以更准确地预测应用程序的运行方式。

过程

1. 将应用程序字节码转换成本机代码。
   a. 先使用常规 JIT 编译器运行样本来进行转换。

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

      在该命令中，*<port>* 是工作站的释放端口。
   b. 在不同窗口中，运行应用程序。

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

      在该命令中，*<port>* 是前一步骤中指定的端口。 该应用程序输出结果类似于以下消息:

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

      和:

      ```
      Land!
      ```
   c. 合并先前步骤中创建的 AOT 选项文件。

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

      用于先前步骤中创建的日志文件的名称在其文件名后附加有日期和进程标识信息。该文件名的格式由 **vlog=** 选项指定。例如，**vlog=sim.aotOpts** 生成的文件名类似于 **sim.aotOpts.20081014.234958.13205**:
   d. 编译 realtime.jar、vm.jar、rt.jar 和应用程序demo.jar 中的 sample.aotOpts 文件。在使用共享类高速缓存时，高速缓存的名称不能超出 53 个字符。

      ```
      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
      ```

      编译结果报告如下:

```
J9 Java™ admincache 1.0
Licensed Materials - Property of IBM

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

```
JVMSHRC256I Persistent shared cache "sample" has been destroyed
Converting files
Converting /team/mstoodle/demo/sdk/jre/lib/i386/realtime/jclSC160/vm.jar into shared class cache
Succeeded to convert jar file /team/mstoodle/demo/sdk/jre/lib/i386/realtime/jclSC160/vm.jar
Converting /team/mstoodle/demo/sdk/jre/lib/i386/realtime/jclSC160/realtime.jar into shared class cache
Succeeded to convert jar file /team/mstoodle/demo/sdk/jre/lib/i386/realtime/jclSC160/realtime.jar
Converting /team/mstoodle/demo/sdk/jre/lib/rt.jar into shared class cache
Succeeded to convert jar file /team/mstoodle/demo/sdk/jre/lib/rt.jar
Converting /team/mstoodle/demo/demo.jar into shared class cache
Succeeded to convert jar file /team/mstoodle/demo/demo.jar

Processing complete
```

注: 行:

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

意为该命令已破坏任何名为“sample”的现有高速缓存，以创建指定的高速缓存。

e. 显示填充的高速缓存的内容。

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

2. 启动模拟:

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

在该命令中，*<port>* 是该工作站的释放端口。

3. 启动控制器:

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

在该命令中，*<host>* 是运行模拟的工作站主机名，*<port>* 是前一步骤中指定的端口。在同一工作站上同时运行两个 JVM 会导致确定性行为降低。请参阅 第 22 页的『注意事项』以获取更多信息。

结果

应用程序运行并生成若干输出，包括:

1. 表明模拟和控制器已启动的消息。
2. 控制器值的定点样本。
3. 同一目录中名为 `graph.svg` 的图，包含定点样本。
4. 事件摘要消息。

在使用提前编译运行应用程序时，定点样本和对应的图显示:

- “雷达高度”数据中无峰值。
- “真实高度”数据的精确跟踪。

原因是 Controller 代码目前正在垃圾回收暂停时间较短且不存在即时编译中断的情况下运行。

使用共享类高速缓存运行该应用程序的优势在于控制器和模拟 JVM 共享由两个 JVM 装入的类使用的部分内存。

样本实时散列映射

WebSphere Real Time for RT Linux 包括 HashMap 和 HashSet 实施，它们为 put 方法提供的性能比 IBM SDK for Java 7 中的标准 HashMap 更为稳定。

IBM 提供的标准 java.util.HashMap 可为高吞吐量应用程序提供优良的服务。它还对知道自身散列映射需要增加到的最大大小的应用程序提供帮助。对于需要增加到可变大小的散列映射的应用程序而言，根据不同的用法，标准散列映射存在潜在性能问题。对于使用 put 方法将新条目添加到散列映射而言，标准散列映射能够提供良好的响应时间。但是当散列映射填满后，必须分配一个更大的后备存储器。这意味着必须迁移当前后备存储器中的条目。如果散列映射较大，那么执行 put 的时间也较长。例如，操作可能需要几毫秒。

WebSphere Real Time for RT Linux 包含一个样本实时散列映射。它提供的功能接口与标准 java.util.HashMap 完全相同，但是为 put 方法提供能够提供更为稳定的性能。样本散列映射会另创建一个后备存储器，而不是在散列映射填满时创建后备存储器并迁移所有条目。新建的后备存储器会链接到该散列映射中的其他后备存储器。链接最初在分配空后备存储器并链接到其他后备存储器时会导致性能略微下降。后备散列映射更新后，迁移所有条目的速度比以前更快。实时散列映射的劣势在于 get、put 和 remove 操作的速度稍微变慢。操作速度变慢的原因在于每一项查找均必须在整个后备散列映射集（而不是某一个散列映射）中执行。

要试用实时散列映射，请将 `RTHashMap.jar` 文件添加到引导类路径的开头。如果将 WebSphere Real Time for RT Linux 安装到目录 `$WRT_ROOT` 中，请添加以下选项以将实时散列映射用于您的应用程序，而不是标准散列映射:

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

用于实时散列映射实施的源和类文件包含在 `demo/realtime/RTHashMap.jar` 文件中。此外，还提供了实时 java.util.LinkedHashMap 和 java.util.HashSet 实施。

使用 Eclipse 开发 WebSphere Real Time for RT Linux 应用程序

开发实时应用程序时可使用 Eclipse 为您提供功能全面的 IDE。

开始之前

如果这是您首次使用 Eclipse 应用程序开发环境来开发实时应用程序，请使用该过程来配置环境。

WebSphere Real Time for RT Linux 提供标准 Oracle **`javac`** 编译器。对使用何种编译器无限制，但是它必须生成有效的 Java 5.0 类文件。但是，javax.realtime.* Java 类必须位于构建路径中。

关于此任务

要开发基于 Eclipse 的应用程序，请执行以下指示信息:

过程

1. 从 http://www.eclipse.org/downloads/ 下载 Eclipse。 建议您使用 Eclipse 3.1.2 以获取正确的 Java 5.0 编译。

2. 下载符合 JVM 标准的 IBM SDK and Runtime Environment for Linux platforms, Java 2 Technology Edition, V5.0 以运行 Eclipse。
3. 从 WebSphere Real Time for RT Linux 程序包中抽取文件 `opt/IBM/javawrt3/jre/lib/i386/realtime/jclSC160/realtime.jar`。
4. 打开 Eclipse，创建项目。单击**文件 > 新建**。 选择**新建项目**面板中的 **Java 项目**。
5. 单击**下一步**以显示**新建 Java 项目**面板。
   a. 输入项目名称，例如 RTSJ-Tests。
   b. 检查 JDK 编译器是否设置为 5.0。
6. 单击**完成**。
7. 创建工作目录，并导入 `opt/IBM/javawrt3/jre/lib/i386/realtime/jclSC160/realtime.jar` 文件。
8. 单击**文件 > 新建 > 文件夹**以打开**新建文件夹**面板。输入新的文件夹名称，例如 *deplib*。
9. 单击**完成**。
10. 要导入 `realtime.jar` 文件，请单击**文件 > 导入**以打开**导入**面板。
11. 单击**文件系统**，然后单击**下一步**。
12. 打开解压 JVM 的文件系统上的 `opt/IBM/javawrt3/jre/lib/i386/realtime/jclSC160/` 目录。
13. 选择 realtime.jar 文件旁的复选框，指定要导入其中的文件夹，例如 `RTSJ-Tests/deplib`，并确保已选中**仅创建已选中的文件夹**选项。
14. 单击**完成**。
15. 将 JAR 文件添加到库路径。右键单击项目，然后单击**属性**以打开**属性**面板。
16. 单击 **Java 构建路径**和**库**选项卡。单击**添加 JAR**。
17. 单击项目目录下的 **realtime.jar**。单击**确定**。

结果

如果过程成功完成，那么 realtime.jar 文件会出现在**库**选项卡的 .jar 文件列表上。

示例

Eclipse 可以使用 `realtime.src.jar` 来显示 RTSJ 类的其他信息。要实现此目的，请打开已导入的 `realtime.jar` 文件的属性窗口，单击 **Java 源附件**，然后在**位置路径:** 中输入 `realtime.src.jar` 文件的位置。

下一步做什么

如果希望使用基于 Eclipse 的 Apache Ant 来构建应用程序，请将 `realtime.jar` 文件添加到 Ant 构建脚本的类路径中。例如:

```
<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>
```

这只是 Ant 构建脚本的一部分。

调试应用程序

使用 Eclipse 应用程序开发程序，您可以远程或本地调试应用程序。

关于此任务

要远程调试实时应用程序，要调试的 JVM 需要以下选项。`-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=10100`

过程

1. 在运行应用程序的 Linux 环境中，输入:

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

   其中:
   - `server=y` 指示 JVM 正在接受来自调试器的连接。
   - `suspend=y` 使得 JVM 在运行前等待调试器进行连接。
   - `address=10100` 是调试器应连接到 JVM 的端口号。此数字一般应高于 1024。

   JVM 显示以下消息:

   ```
   Listening for transport dt_socket at address: 10100
   ```
2. 在 Eclipse 中打开应用程序，并选择**调试**。
3. 应创建用于调试远程应用程序的新配置。如果同一项目中的应用程序正在运行并且正在同一端口上侦听每次运行，那么您只需创建一个配置。
4. 创建配置后，填写以下内容: 配置的名称，包含正调试的应用程序的项目名称，运行应用程序的工作站的*主机名*，以及在 **-agentlib** 选项中传入的端口号。
5. 单击**调试**以启动调试会话。应为您打开**调试**透视图以查看远程调试的 JVM 的状态。

使用 JVM 运行 Eclipse

本部分介绍如何使用 WebSphere Real Time for RT Linux JVM 来运行 Eclipse。

要使用 JVM 来运行 Eclipse，请使用 **eclipse** 命令指定以下项:

- 计划使用的 WebSphere Real Time for RT Linux JVM 的 Java 可执行文件的标准目录
- **-Xrealtime** JVM 选项
- 希望 Eclipse 使用的永久内存大小。大小最小为 128M。

例如:

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

注: Eclipse SDK 不能利用对 WebSphere Real Time for RT Linux 应用程序可用的各种实时内存选项。该行为导致的结果之一就是永久内存耗尽，尤其是 Eclipse 已在无重新启动的情况下连续使用了好几个小时或好几天。如果出现 **OutOfMemory** 错误，您可增加 **-Xgc:immortalMemorySize** 选项的值，从而增加希望 Eclipse 使用的永久内存量。

第 7 章 性能

对 WebSphere Real Time for RT Linux 针对始终短暂的 GC 暂停，而非最高吞吐量性能或最小内存占用量进行优化。

在支持超线程的系统上，您必须确保未启用超线程。原因是要避免使用 WebSphere Real Time for RT Linux 时产生不良性能影响。

降低计时可变性以及对 Real-Time Specification for Java (RTSJ) 的支持要求禁用一些标准 IBM Java 运行时优化。因此，当标准 Java 应用程序在使用 **-Xrealtime** 参数运行时很可能会注意到整体性能下降。

经认可的硬件配置上的性能

经认可的系统具有足够的时钟粒度和处理器速度来支持 WebSphere Real Time for RT Linux 性能目标。例如，未超负荷的系统上运行的编写良好并具有足够堆大小的应用程序一般会经历远低于 1 毫秒且通常约为 500 微秒的 GC 暂停时间。在 GC 周期内，具有缺省环境设置的应用程序不会暂停超过任何滑动的 10 毫秒时间窗内所耗用时间的 30%。在任意 10 毫秒周期内的 GC 暂停中所耗用的总时间通常总共为少于 3 毫秒。

降低计时可变性

标准 JVM 中的两种主要的可变性源按如下所述在 WebSphere Real Time for RT Linux 中处理:

- Java 代码准备: 加载和及时 (JIT) 编译由提前 (AOT) 编译来处理。请参阅第 38 页的『使用 AOT 编译器』。
- 垃圾回收暂停: 通过使用Metronome 垃圾回收器来避免标准垃圾回收器方式所带来的可能长时间暂停。请参阅第 61 页的『使用 Metronome 垃圾回收器』。

非实时方式下 JVM 之间的类数据共享

在非实时方式下支持类共享，但与在实时方式下的运行有所不同。

通过将类数据存储在磁盘上的内存映射的缓存文件中，您可以在 Java 虚拟机 (JVM) 之间共享这些数据。当多个 JVM 共享一个高速缓存时，共享会降低总体虚拟存储器耗用。当创建高速缓存后，类共享还会降低 JVM 的启动时间。共享类高速缓存独立于任何运行的 JVM，并且持续存在直到它被删除。

共享的高速缓存可以包含:

- 引导程序类
- 应用程序类
- 描述了的元数据
- 提前（AOT）编译的代码

注: 非实时 JVM 不能移除实时共享类高速缓存。

第 8 章 安全性

本部分包含关于安全性的重要信息。

共享类高速缓存的安全注意事项

共享类高速缓存旨在更方便地执行高速缓存的管理和使用，但是缺省的安全策略可能并不适用。

使用共享类高速缓存时，必须了解新文件的缺省许可权，这样可通过限制访问权限来提高安全性。

文件	缺省许可权
新共享高速缓存	对组和其他条目的读许可权
javasharedresources 目录	读、写和执行许可权

您需要同时具有缓存文件和缓存目录的写许可权才可破坏或增加高速缓存。

更改缓存文件的文件许可权

要限制共享类高速缓存的访问权，可使用 **chmod** 命令。

所需更改	命令
限制用户和组的访问权限	chmod 770 /tmp/javasharedresources
限制用户的访问权限	chmod 700 /tmp/javasharedresources
仅限制用户对某特定高速缓存的读和写访问权	chmod 600 /tmp/javasharedresources/*<file for shared cache>*
仅限制用户和组对某特定高速缓存的读和写访问权	chmod 660 /tmp/javasharedresources/*<file for shared cache>*

请参阅第 40 页的『创建实时共享类高速缓存』，以获取有关创建共享类高速缓存的更多信息。

连接到无访问许可权的高速缓存

如果尝试连接到无相应访问许可权的高速缓存，会看到一条错误消息:

```
JVMSHRC226E Error opening shared class cache file
JVMSHRC220E Port layer error code = -302
JVMSHRC221E Platform error message: Permission denied
JVMJ9VM015W Initialization error for library j9shr25(11): JVMJ9VM009E J9VMDllMain
failed
Could not create the Java virtual machine.
```

第 9 章 故障诊断与支持

WebSphere Real Time for RT Linux 的故障诊断与支持

- 『常规问题确定方法』
- 第 94 页的『OutOfMemory 错误故障诊断』
- 第 103 页的『使用诊断工具』

常规问题确定方法

确定问题可以帮助您了解故障类型以及适当的纠正措施。

了解问题类型后，可能要执行以下一个或多个任务:

- 解决问题。
- 找到良好的变通方法。
- 收集用于向 IBM 生成错误报告的必要数据。

Linux 问题确定

本部分描述 Linux 上的问题确定。

IBM SDK for Java V7 用户指南包含了关于 Linux 上问题诊断的有用指南，涵盖以下内容:

- 设置并检查您的 Linux 环境
- 常用调试方法
- 崩溃诊断
- 调试挂起
- 调试内存泄漏
- 调试性能问题

您可以在以下位置找到此信息: IBM SDK for Java 7 - Linux problem determination。

以下信息是对 IBM WebSphere Real Time for RT Linux 的补充。

设置并检查您的 Linux 环境

在 IBM WebSphere Real Time for RT Linux 上，检查 JVM 是否已正确配置以生成系统转储。

Linux 系统转储（核心文件）

发生崩溃时，要获取的最重要诊断数据是 Linux 系统转储（核心文件）。要确保生成该文件，您必须检查操作系统设置以及可用磁盘空间，如 IBM SDK for Java V7 用户指南中所述。

Java 虚拟机设置

JVM 必须配置为在发生崩溃时生成核心文件。在命令行上运行 `java -Xrealtime -Xdump:what`。该选项的输出是:

```
-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
```

所显示的值是缺省设置。在发生崩溃时，必须至少设置 events=gpf 以生成核心文件。您可以使用以下命令行选项来更改和设置选项
-Xdump:system[:name1=value1,name2=value2 ...]

常用调试方法

因为 Java 线程名称在操作系统中可视，所以您可以使用 ps 命令来帮助进行调试。在使用跟踪工具时，必须对 IBM WebSphere Real Time for RT Linux 使用正确的命令。

检查进程信息

在 IBM WebSphere Real Time for RT Linux 上运行 ps 命令时可预期看到的输出为:

```
ps -eLo pid,tid,rtprio,comm,cmd
29286 29286      - java            jre/bin/java -Xrealtime -jar example.jar
29286 29287      - main            jre/bin/java -Xrealtime -jar example.jar
29286 29290     88 Signal Reporter jre/bin/java -Xrealtime -jar example.jar
29286 29295      - JIT Compilation jre/bin/java -Xrealtime -jar example.jar
29286 29296     13 JIT Sampler     jre/bin/java -Xrealtime -jar example.jar
29286 29297      - Signal Dispatch jre/bin/java -Xrealtime -jar example.jar
29286 29298      - Finalizer maste jre/bin/java -Xrealtime -jar example.jar
29286 29299     11 Gc Slave Thread jre/bin/java -Xrealtime -jar example.jar
29286 29300     89 Metronome GC Al jre/bin/java -Xrealtime -jar example.jar
29286 29301      - Thread-2        jre/bin/java -Xrealtime -jar example.jar
29286 29302     43 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29303     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29304     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29305     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29306     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29307     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29311     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29312     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29313     85 Realtime AEH No jre/bin/java -Xrealtime -jar example.jar
29286 29314     85 Realtime AEH No jre/bin/java -Xrealtime -jar example.jar
29286 29315     87 Realtime Schedu jre/bin/java -Xrealtime -jar example.jar
29286 29316     79 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29317     85 Realtime Non-he jre/bin/java -Xrealtime -jar example.jar
29286 29318     83 Realtime Heap T jre/bin/java -Xrealtime -jar example.jar
29286 29319     83 Realtime Heap T jre/bin/java -Xrealtime -jar example.jar
29286 29321     45 RealtimeThread- jre/bin/java -Xrealtime -jar example.jar
29286 29343     43 RealtimeThread- jre/bin/java -Xrealtime -jar example.jar
29286 29345      - stdout reader j jre/bin/java -Xrealtime -jar example.jar
29286 29346      - stderr reader j jre/bin/java -Xrealtime -jar example.jar
```

e 选择所有进程。

L 显示线程。

o 提供要显示的列的预定义格式。指定的列为进程标识、线程标识、调度策略、实时线程优先级以及与进程关联的命令。这些信息对于了解在给定时间应用程序和虚拟机中的什么线程正在运行很有用。

跟踪工具

Linux 上的三种跟踪工具是 **strace**、**ltrace** 和 **mtrace**。命令 man strace 显示完整一组可用选项。

strace

strace 工具跟踪系统调用。您可以在已经可用的进程上使用该工具，也可以通过新进程启动该工具。 strace 记录程序所发出的系统调用和进程所接收的信号。对于每个系统调用，均将使用名称、参数和返回值。strace 允许您在无需源的情况下跟踪程序（不需要进行重新编译）。如果您使用带有 **-f** 选项的 strace，那么将跟踪因派生系统调用而已创建的子进程。您可以使用 strace 来调查插件问题或尝试了解程序无法正确启动的原因。

要将 strace 用于 Java 应用程序，请输入 `strace java -Xrealtime <class-name>`。

您可以通过使用 `-o`选项来将跟踪输出从 strace 工具定向到文件。

ltrace

ltrace 工具依赖于分发。它与 strace 非常相似。该工具在执行进程进行调用时拦截并记录动态库调用。strace 对于执行进程所接收的信号执行上述同样的操作。

要将 ltrace 用于 Java 应用程序，请输入 `ltrace java -Xrealtime <class-name>`。

mtrace

mtrace 包含在 GNU 工具集中。它用于安装 malloc、realloc 和 free 的处理程序，并且使这些函数的所有用户能够被跟踪并记录到文件。该跟踪会降低程序效率，不应在一般使用情况下启用。要使用 mtrace，请将 **IBM_MALLOCTRACE** 设置为 1，并将 **MALLOC_TRACE** 设置为指向跟踪信息将存储到的有效文件。您必须拥有对该文件的写访问权。

要将 mtrace 用于 Java 应用程序，请输入:

```
export IBM_MALLOCTRACE=1
export MALLOC_TRACE=/tmp/file
java -Xrealtime <class-name>
mtrace /tmp/file
```

崩溃诊断

在收集崩溃之前关于运行中的进程以及 Java 环境的信息时，请遵循以下准则。

收集进程信息

当搜索在发生崩溃之前发生的事件时，请使用 **gdb** 和 **bt** 命令来显示故障线程的堆栈跟踪，而不是分析核心文件。

查明 Java 环境的相关信息

使用 Java 转储来确定各线程所执行的操作以及哪些 Java 方法在运行。将函数地址与库地址进行对照以确定在各个点运行的代码源。

使用 **-verbose:gc** 选项查看 Java 堆以及永久和设置了作用域的内存区域的状态。提出以下问题:

- 在可能导致崩溃的内存区域之一中是否存在内存不足情况？
- 崩溃是否在垃圾回收期间发生，从而指示可能是垃圾回收故障？
- 崩溃是否在垃圾回收之后发生，从而指示可能是内存损坏？

调试性能问题

在调试性能问题时，除了 IBM SDK for Java V7 用户指南中的主题之外，还请考虑 IBM WebSphere Real Time for RT Linux 的以下特定事项。

调整内存区域大小

可通过调整堆、永久和设置了作用域的内存的大小来对 JVM 进行调整。选择正确大小可优化性能。使用正确大小可使垃圾回收器更容易提供所需的利用率。

有关调整内存区域大小的更多信息，请参阅第 124 页的『Metronome 垃圾回收器故障诊断』。

JIT 编译和性能

在使用 JIT 时，您应考虑对实时行为的影响。

如果您需要可预测的行为，但也需要更佳的性能，那么应考虑使用提前 (AOT) 编译。有关更多信息，请参阅第 36 页的『在 WebSphere Real Time for RT Linux 中使用经过编译的代码』。

Linux 上的已知限制

Linux 一直在快速发展，而 JVM 与操作系统之间的交互一直存在各种问题，尤其是在线程领域。

请注意以下可能影响 Linux 系统的限制。

作为进程的线程

如果 Java 线程的数量超过所允许的最大进程数，那么您的程序可能:

- 收到错误消息
- 收到 **SIGSEGV** 错误
- 停止

有关更多信息，请参阅位于以下位置的 *The Volano Report*: http://www.volano.com/report/index.html。

浮动堆栈限制

如果在无浮动堆栈的情况下运行，那么无论 **-Xss** 的设置如何，均会为每个线程提供最低 256 KB 的本机堆栈大小。

在浮动堆栈 Linux 系统上，会使用 **-Xss** 值。如果您从非浮动堆栈 Linux 系统进行迁移，请确保任何 **-Xss** 值都足够大并且不依赖于最小值 256 KB。

glibc 限制

如果您收到一条消息指示由于未找到符号（例如 __bzero）而无法装入 `libjava.so` 库，那么您可能安装了较低版本的 GNU C 运行时库 glibc。用于 Linux 线程实施的 SDK 需要 glibc V2.3.2 或更高版本。

字体限制

当您在 Red Hat 系统上进行安装时，为了让字体服务器能够找到 Java TrueType 字体，请运行（例如，在 Linux IA32 上）：

```
/usr/sbin/chkfontpath --add opt/IBM/javawrt3/jre/lib/fonts
```

您必须在安装时执行该操作，并且必须以“root”用户身份登录来运行该命令。要了解更详细的字体问题，请参阅《Linux SDK 和运行时环境用户指南》。

Linux Red Hat MRG 内核中的性能问题

Red Hat MRG 内核的配置问题可能会导致在启用了详细垃圾回收的情况下启动 WebSphere Real Time 时应用程序线程意外暂停。这些暂停不会在详细 GC 输出中报告，但根据网络配置情况，可能会持续若干毫秒。从远程定义的 LDAP 用户启动的 JVM 最受影响，因为名称服务高速缓存守护程序 (nscd) 未启动，从而导致网络延迟。通过启动 `nscd` 可解决该问题。按照以下步骤来检查 `nscd` 服务的状态并更正此问题：

1. 通过输入以下命令来检查 `nscd` 守护程序是否正在运行：

   ```
   /sbin/service nscd status
   ```

   如果该守护程序未在运行，您将看到以下消息：

   ```
   nscd is stopped
   ```

2. 以 root 用户身份使用以下命令启动 `nscd` 服务：

   ```
   /sbin/service nscd start
   ```

3. 以 root 用户身份使用以下命令更改 `nscd` 服务的启动信息：

   ```
   /sbin/chkconfig nscd on
   ```

`nscd` 进程现正在运行，并会在重新引导之后自动启动。

NLS 问题确定

JVM 包含针对不同语言环境的内置支持。

IBM SDK for Java V7 用户指南包含关于诊断 NLS 问题的有用指南，涵盖以下内容：

- 字体概述
- 字体实用程序
- 常见 NLS 问题和可能的原因

您可以在以下位置找到此信息：IBM SDK for Java 7 - NLS problem determination。

ORB 问题确定

调试 ORB 问题的首要任务是确定问题位于分布式应用程序的客户机端还是服务器端。典型的 RMI-IIOP 会话可看作请求访问某对象的客户机和提供该对象的服务器之间的简单同步通信。

IBM SDK for Java V7 用户指南包含关于诊断 ORB 问题的有用指南，涵盖以下内容：

- 确定 ORB 问题
- 解释堆栈跟踪
- 解释 ORB 跟踪

- 常见问题
- IBM ORB 服务: 收集数据

您可以在以下位置找到此信息: IBM SDK for Java 7 - ORB problem determination。

以下信息是对 IBM WebSphere Real Time for RT Linux 的补充。

IBM ORB 服务: 收集数据

收集用于服务的 Java 版本输出时，请运行以下命令:

```
java -Xrealtime -version
```

初步测试

发生问题时，ORB 可能会生成包含以下内容的 `org.omg.CORBA.*` 异常:

- 指示原因的文本
- 次代码
- 完成状态

在推断 ORB 是问题的原因之前，请确保以下事项成立:

- 场景可在类似配置中重现。
- JIT 已禁用。
- 未在使用任何 AOT 已编译代码

其他操作包括:

- 关闭其他处理器。
- 如果可能，关闭同时多线程 (SMT)。
- 消除客户机或服务器的内存依赖性。物理内存短缺可能是低性能、明显挂起或崩溃的原因。要除去这些问题，请确保拥有合理的内存空间。
- 检查物理网络问题，例如防火墙、通信链路、路由器和 DNS 名称服务器。这些是 CORBA COMM_FAILURE 异常的主要原因。测试时，请 ping 您自己的工作站名称。
- 如果应用程序正在使用 DB2® 等数据库，请切换到最可靠的驱动程序。例如，要隔离 DB2 AppDriver，请切换到 Net Driver，该驱动程序速度较慢且使用套接字，但更加可靠。

OutOfMemory 错误故障诊断

处理 OutOfMemoryError 异常、内存泄漏和隐藏内存分配。

有关 Metronome 垃圾回收器的常规故障诊断信息，请参阅第 124 页的『Metronome 垃圾回收器故障诊断』。

诊断 OutOfMemoryError

在 Metronome 垃圾回收器中诊断 OutOfMemoryError 异常比在标准 JVM 中进行诊断复杂得多，这是由此垃圾收集器的周期性决定的。

第 12 页的『内存管理』对不同类型的堆的特征作了描述。通常，RTSJ 应用程序需要的堆空间大约比标准 Java 应用程序多 20%。

缺省情况下，发生未捕获的 OutOfMemoryError 时，JVM 将生成以下诊断输出:

- 快照转储; 请参阅第 105 页的『使用转储代理程序』。
- Heapdump; 请参阅第 113 页的『使用 Heapdump』。
- Javadump; 请参阅第 108 页的『使用 Javadump』。
- 系统转储; 请参阅第 116 页的『使用系统转储和转储查看器』。

转储文件名在控制台输出中给出:

```
JVMDUMP006I Processing dump event "systhrow", detail "java/lang/OutOfMemoryError" - please wait.
JVMDUMP007I JVM Requesting Snap dump using 'Snap.20081017.104217.13161.0001.trc'
JVMDUMP010I Snap dump written to Snap.20081017.104217.13161.0001.trc
JVMDUMP007I JVM Requesting Heap dump using 'heapdump.20081017.104217.13161.0002.phd'
JVMDUMP010I Heap dump written to heapdump.20081017.104217.13161.0002.phd
JVMDUMP007I JVM Requesting Java dump using 'javacore.20081017.104217.13161.0003.txt'
JVMDUMP010I Java dump written to javacore.20081017.104217.13161.0003.txt
JVMDUMP013I Processed dump event "systhrow", detail "java/lang/OutOfMemoryError".
```

显示在控制台输出中并且在 Javadump 中提供的 Java 回溯指示了 Java 应用程序中发生 OutOfMemoryError 的位置。下一步是确定哪个 RTSJ 内存区域已满。JVM 内存管理组件发出一个跟踪点，该跟踪点提供了失败分配的大小、类块地址和内存空间名称。您可以在快照转储中找到此跟踪点:

```
<< lines omitted... >>
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
```

跟踪点标识和数据字段可能与显示的内容不同，具体取决于所分配的对象的类型。在此示例中，跟踪点指出，当应用程序尝试在 Metronome heap 的内存段 id=0x809c5f0 中分配大小为 33.6 MB 且类型为 class 0x81312d8 的对象时，发生分配故障。

您可以通过查看 Javadump 中的内存管理信息来确定受影响的 RTSJ 内存区域:

```
NULL           ------------------------------------------------------------------
0SECTION       MEMINFO subcomponent dump routine
NULL           ================================
NULL
1STMEMTYPE     Object Memory
NULL           region     start      end        size       name
1STHEAP        0xF288B584 0xF2A1C000 0xF6A1C000 0x04000000 Default
NULL
1STMEMUSAGE    Total memory available: 67108864 (0x04000000)
1STMEMUSAGE    Total memory in use:    66676824 (0x03F96858)
1STMEMUSAGE    Total memory free:      00432040 (0x000697A8)
NULL
NULL           region     start      end        size       name
1STHEAP        0xF288B5A4 0xF17FF008 0xF27FF008 0x01000000 Immortal
NULL
1STMEMUSAGE    Total memory available: 16777216 (0x01000000)
1STMEMUSAGE    Total memory in use:    00450816 (0x0006E100)
1STMEMUSAGE    Total memory free:      16326400 (0x00F91F00)
NULL
1STSEGTYPE     Internal Memory
NULL           segment    start      alloc      end        type       size
1STSEGMENT     0x0808DA48 0x0814A0A8 0x0814A0A8 0x0815A0A8 0x01000040 0x00010000
1STSEGMENT     0x0808DB50 0x08131EB8 0x08131EB8 0x08141EB8 0x01000040 0x00010000
<< lines removed for clarity >>
```

可以通过查看 Javadump 的类部分确定所分配对象的类型:

```
NULL           ------------------------------------------------------------
0SECTION       CLASSES subcomponent dump routine
NULL           ================================
<< lines omitted... >>
1CLTEXTCLLOD    ClassLoader loaded classes
2CLTEXTCLLOAD    Loader *System*(0xF182BB80)
<< lines omitted... >>
3CLTEXTCLASS      [C(0xF1632D80)
```

Javadump 中的信息确认所尝试的分配是针对字符数组在正常堆 (ID=0xF288B584) 中进行，并且该堆的分配总大小（由相应的 1STHEAP 行指示）是十进制字节数 67108864 或者十六进制字节数 0x04000000，即 64 MB。

在此示例中，失败的分配相对于堆的总大小较大。如果应用程序应该创建 33 MB 的对象，那么下一步是使用 **-Xmx** 选项增大堆大小。

更为常见的情况时，失败的分配相对于堆的总大小较小。这是因为，先前的分配已将堆用尽。在这些情况下，下一步是使用 Heapdump 来调查分配给现有对象的内存量。

Heapdump 是一个压缩二进制文件，其中包含所有对象及其对象类、大小和引用的列表。请使用 IBM Monitoring and Diagnostic Tools for Java - Memory Analyzer 工具来分析 Heapdump，此工具可以从 IBM Support Assistant (ISA) 下载。

通过使用 MDD4J，可以装入 Heapdump 并找到可能耗用了大量堆空间的对象的树结构。此工具可以提供堆中的对象的各种视图。例如，MDD4J 可以显示一个视图以详细描述可疑的泄漏，并给出堆中前 5 个最大的对象和包。选择树形视图将提供有关发生泄漏的容器对象的性质的更多信息。

缺省情况下，将生成单一 Heapdump 文件，其中包含所有 RTSJ 内存空间中的所有对象。通过使用命令行选项 **-Xdump:heap:request=multiple**，可以请求针对每个内存空间创建不同的 Heapdump。借助多个转储，可以只检查在特定内存区域中分配的对象的集合。您可以通过控制台输出中给出的文件名来标识 Heapdump:

```
JVMDUMP006I Processing Dump Event "uncaught", detail "java/lang/OutOfMemoryError" - Please Wait.
<< lines omitted... >>
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
<< lines omitted... >>
JVMDUMP013I Processed Dump Event "uncaught", detail "java/lang/OutOfMemoryError".
Exception in thread "RTJ Memory Consumer (thread_type=Realtime)" java.lang.OutOfMemoryError
 at tests.com.ibm.jtc.ras.runnable.DepleteMemory.depleteMemory(DepleteMemory.java:57)
<< lines omitted... >>
```

IBM JVM 管理内存的方式

IBM JVM 需要内存以用于多个不同的组件，包括类、已编译的代码、Java 对象、Java 堆栈和 JNI 堆栈的内存区域。其中的一些内存区域必须在连续内存中。其他内存区域可以分段为较小的区域并链接到一起。

动态装入的类和已编译的代码存储在动态装入的类的分段内存区域中。这些类进一步划分到可写内存区域（RAM 类）和只读内存区域（ROM 类）中。在运行时，应用程序

启动期间，将对类高速缓存进行内存映射（但不必将其装入），从而将其映射到连续内存区域。应用程序引用这些类时，类高速缓存中的类以及已编译的代码将映射到存储器。类的 ROM 组成部分由多个引用了这个类的进程共享。类的 RAM 组成部分在动态装入的类首次被 JVM 引用时，在这个类的分段内存区域中创建。类高速缓存中各个类方法的 AOT 编译代码将复制到可执行动态代码内存区域中，这是因为，各个进程不会共享此代码。并非从类高速缓存中装入的类与高速缓存的类相似，只不过 ROM 类信息在动态装入的类的分段内存区域中创建。动态生成的代码与高速缓存的类的 AOT 代码存储在相同的动态代码内存区域中。

在未指定 **-Xrealtime** 选项的情况下运行 JVM 时，所有 Java 对象都存储在标准的堆内存中。如果使用了 **-Xrealtime** 选项，那么还可以在两个附加的内存区域（分别称为永久内存和作用域限定内存）中分配对象。

每个 Java 线程的堆栈都可以跨分段内存区域。每个线程的 JNI 堆栈都占用连续的内存区域。

要确定如何配置 JVM，请在使用 **-verbose:sizes** 选项的情况下运行。此选项将打印出允许您管理大小的内存区域的相关信息。对于非连续内存区域，将打印一个增量，以描述每次需要增大该区域时获取的内存量。

以下是使用 **-Xrealtime -verbose:sizes** 选项时的输出示例:

```
-Xmca32K                          RAM 类段增量
-Xmco128K                         ROM 类段增量
-Xms64M                           初始内存大小
-Xgc:immortalMemorySize=16M       永久内存空间大小
-Xgc:scopedMemoryMaximumSize=8M 作用域限定内存空间最大大小
-Xmx64M                           最大内存
-Xmso256K                         操作系统线程堆栈大小
-Xiss2K                           Java 线程堆栈初始大小
-Xssi16K                          Java 线程堆栈增量
-Xss256K                          Java 线程堆栈最大大小
```

此示例指示 RAM 类段最初为 0，但根据需要以 32 KB 块为增量增大。ROM 类段最初为 0，并且根据需要以 128 KB 块为增量增大。您可以使用 **-Xmca** 和 **-Xmco** 选项来控制这些大小。RAM 类和 ROM 类段根据需要增大，因此您通常不需要更改这些选项。

永久内存是连续区域，您可能需要为其预先分配较大的空间。在本例中，为永久内存区域预先分配了 16 MB。如果您尝试将 16 MB 以上的对象写入此永久内存区域，那么将接收到 OutOfMemory 异常，这是因为根据定义，不会对此内存区域进行垃圾收集。

作用域限定内存区域是连续的，在本例中，为其预先分配了 8 MB。如果程序运行期间有许多处于活动状态的作用域限定内存区域，那么您可能需要指定较大的作用域限定内存区域。

使用 admincache 实用程序可以确定使用类高速缓存时内存映射区域的大小。以下是 `admincache -Xrealtime -printStats -nologo` 命令的输出样本:

```
J9 Java™ 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
```

高速缓存大小指示空间中的内存映射区域稍大于 7 MB。ROM 类和 AOT 字节占用了其中的大部分空间，它们各使用了略多于 3 MB 的空间。

永久内存空间中的示例 OutOfMemoryError

此示例说明如何识别永久内存空间中的 OutOfMemoryError，并描述了预防此问题所需执行的步骤。

快照转储表明，在永久内存区域 id=0x809dd1c 中，两个小型分配请求失败:

```
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
```

Javadump 表明永久内存空间已满:

```
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
<< lines omitted... >>
1STSEGTYPE     Object Memory
NULL           segment  start    alloc    end       type     bytes
1STSEGSUBTYPE  Immortal Segment  ID=0809DD1C
1STSEGMENT     0809D510 B279D008 B379D008 B379D008  00001008 1000000
```

MDD4J 分析表明，已分配非常大的 LinkedList，因此使用了相当大比例的可用内存。

建议您最大限度地减少在永久内存区域中分配的对象数，这是因为，系统不会对永久区域中的对象进行垃圾收集。最常见的永久内存使用是类装入，这是在 JVM 和应用程序初始化期间最常发生的有限活动。如果应用程序要装入大量的类或者在其他方面使用永久内存，那么可以使用 **-Xgc:immortalMemorySize=<size>** 选项来增大永久内存区域的大小。永久内存区域的缺省大小为 16 MB。

如果增大永久内存区域大小只能推迟永久内存 OutOfMemoryError 的发生，请调查持续分配永久数据的模式，确定是与类装入相关还是与其他应用程序对象相关。

作用域限定内存空间中的示例 OutOfMemoryError

此示例说明如何识别作用域限定内存空间中的 OutOfMemoryError，并描述了预防此问题所需执行的步骤。

通过使用命令行选项 **-Xdump:heap:request=multiple**，针对每个内存空间生成不同的转储:

```
VMDUMP006I Processing Dump Event "uncaught", detail "java/lang/OutOfMemoryError" - Please Wait.
JVMDUMP007I JVM Requesting Snap Dump using '/home/test/snap-0001.trc'
JVMDUMP010I Snap Dump written to /home/test/snap-0001.trc
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Default0809DCD8-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Default0809DCD8-0002.phd
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Immortal0809DCF4-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Immortal0809DCF4-0002.phd
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Scope0809DD10-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Scope0809DD10-0002.phd
JVMDUMP007I JVM Requesting Java Dump using '/home/test/javacore-0003.txt'
JVMDUMP010I Java Dump written to /home/test/javacore-0003.txt
JVMDUMP013I Processed Dump Event "uncaught", detail "java/lang/OutOfMemoryError".
Exception in thread "RTJ Memory Consumer (thread_type=Realtime)" java.lang.OutOfMemoryError
   at tests.com.ibm.jtc.ras.runnable.DepleteMemory.depleteMemory(DepleteMemory.java:57)
   at tests.com.ibm.jtc.ras.runnable.DepleteMemory.run(DepleteMemory.java:26)
<< lines omitted... >>
```

快照转储表明，在作用域限定内存区域 id=0x809dd10 中，两个分配请求失败:

```
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
```

Javadump 表明，对于作用域限定内存区域 id=0x809dd10，分配的内存区域大小相当小，只有 60 KB；在这种情况下，请在应用程序代码中增大作用域限定内存区域的大小。

```
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
```

在示例 Javadump 中，作用域限定内存区域看起来是空的。这是因为，Javadump 在 OutOfMemoryError 到达 JVM 时生成，此时，作用域已退出并被清除。您可以使用 **-Xdump:java:events=throw,filter=java/lang/OutOfMemoryError** 命令行选项在故障点生成 Javadump。通过使用此选项，将正确报告作用域限定内存区域中的可用空间量。

另外，作用域限定内存的总空间也可能耗尽；在这种情况下，请使用 **-Xgc:scopedMemoryMaximumSize=<size>** 命令行选项增大作用域限定内存区域的大小。作用域限定内存区域的缺省大小为 8 MB。如果所有可用于作用域限定内存的空间用尽，那么您将在控制台上看到不同的消息；例如:

```
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)
```

诊断多个堆中的问题

您可以将 Javadump 中提供的地址范围与 Heapdump 中的占用信息配合使用，以帮助分析多个 RTSJ 内存区域中的 OutOfMemoryError。

在以下 Javadump 中，永久段范围是 0xB281C008 到 0xB381C008，正常堆段范围是 0xB381D008 到 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
<< lines omitted... >>
1STSEGTYPE     Object Memory
NULL           segment  start    alloc    end       type     bytes
1STSEGSUBTYPE  Immortal Segment  ID=0809C68C
1STSEGMENT     0809BE80 B281C008 B381C008 B381C008  00001008 1000000
1STSEGSUBTYPE  Heap Segment  ID=0809C670
1STSEGMENT     0809BE08 B381D008 B781D008 B781D008  00000009 4000000
NULL
1STSEGTYPE     Class Memory
NULL           segment  start    alloc    end       type     bytes
1STSEGMENT     08158154 083FFD68 083FFEF0 08407D68  00010040 8004
```

Heapdump 是一个压缩二进制文件，其中包含所有对象及其对象类、大小和引用的列表。请使用 IBM Monitoring and Diagnostic Tools for Java - Memory Analyzer 工具来分析 Heapdump，此工具可以从 IBM Support Assistant (ISA) 下载。

您可以使用 MDD4J 列示的对象内存位置来确定对象所在的内存空间。0xB28*nnnnn* 范围内的地址处于永久内存区域。0xB61*nnnnn* 范围内的地址处于正常堆中。

避免内存泄露

垃圾回收器不处理永久或设置了作用域的内存区域。如果是永久内存，那么将仅在 JVM 退出时释放内存。设置了作用域的内存区域将仅在其引用计数达到零之后得以释放。这些上下文中运行的长期运行任务必须通过如下的方式编写：任务已活跃后，不再从永久内存区域中分配更多的内存。

加载类使用少量的永久内存。在实时环境中不会对这些类进行垃圾回收。这样，应用程序不需要的加载类可能会导致应用程序使用超过必要量的永久内存。

如果您的应用程序包含用于实施 Serializable 接口的类，请调整初始永久内存大小以将所生成类的占用量计算在内。每个构造方法对于每个类都有一个所生成对象，格式为“GeneratedSerializationConstructorAccessorXXX”（其中 XXX 是一个数字），该对象在首次进行序列化时装入永久内存。

请避免使用永久内存，因为无法对从永久内存内分配的对象进行垃圾回收。如果永久内存区域不仅只是偶尔使用，请考虑在永久内存中启用对象池。

通过语言功能隐藏的内存分配

在设置了作用域的或永久内存上下文中，请避免使用变量参数语言功能，因为这些方法会分配隐藏的内存。

变量参数 (vararg)

Java 语言通过将变量参数作为数组传递到方法来实施这些变量参数。编译器通过创建并初始化数组来简化变量参数方法的调用。

在永久或设置了作用域的内存上下文中调用变量参数方法可能会损失内存。请勿在设置了作用域的或永久内存上下文中使用变量参数。请改为显式创建数组并使用它来代替变量参数。

以下两个示例显示了用来调用变量参数方法的两种等效方式:

```
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;
    }
}
```

第二个示例为首选方式。因为双数组分配在代码中变得可视，并且此分配可以定向到特定内存区域中。

字符串并置

通过使用 java.lang.StringBuilder 对象向现有字符串添加内容，实现生成更长字符串的操作，而这需要内存分配。

自动装箱

自动装箱涉及创建对象以包含基本类型，而这需要内存分配。

跨内存上下文使用反射

如果已在设置了作用域的内存区域中构建 constructor 对象，那么只能在同一作用域或一个内部作用域中使用该对象。在永久、堆或外部作用域内存上下文中使用该 constructor 对象的任何尝试都将失败。

跨内存上下文发生反射时抛出的异常将类似于以下输出:

```
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)
```

通过在 constructor 对象所分配到的作用域中使用该对象，可以对该限制进行变通。

在作用域内存区域中使用内部类

在作用域内存区域上下文中使用内部类时，如果外部和内部对象位于不同的内存区域中，那么实例化内部类对象时请格外小心。如果内部对象不能存储对外部对象的引用，那么 IllegalAssignmentError 将产生于编译器生成的代码（其在原始源代码中不可见）。

内部类对象必须能够存储对其外部类对象的隐式引用。如果引用违反 RTSJ 内存引用规则，那么将生成 IllegalAssignmentError。

大多数内部类（包括本地和匿名内部类）将包含一个编译器生成（合成）的非静态字段，用于词法封闭的外部类的实例。唯一例外是当内部类实例不包含封闭外部对象（如静态初始化程序块中实例化的匿名类对象）时。内部对象的合成字段将包含对外部对象的引用。这由编译器进行实施，目的是方便 Java 程序员使用。该字段在原始源代码中不可见，但可以使用静态嵌套类（带有可见引用）来编写相似的代码。如果隐式引用违反 RTSJ 内存区域规则，将抛出 IllegalAssignmentError，当构造了内部对象时，其尝试存储对外部对象的引用。

一般情况下，使用内部类时不能违反 RTSJ 内存引用规则。如果对关联外部对象的引用违反了 RTSJ 内存引用规则，那么无法创建内部对象。该规则意为分配到永久内存或堆中的内部对象不能具有对作用域内存中外部对象的引用。作用域内存中的内部对象可以具有对作用域内存中外部对象的引用，但是外部对象必须从同一作用域内存区域或外部作用域内存区域进行分配。

有两种变通方法，包括:

- 使用静态嵌套类来消除隐式引用
- 选择合适的内存区域，确保内部和外部对象关系不会违反内存区域引用限制

使用诊断工具

有多种可用于帮助诊断 IBM WebSphere Real Time for RT Linux JVM 问题的诊断工具。

IBM SDK for Java 7 提供了多种诊断工具，这些工具可用于诊断 IBM WebSphere Real Time for RT Linux JVM 的问题。本部分介绍可用的工具，并提供关于如何使用这些工具的更多信息的链接。

在使用 SDK 诊断工具时有一个要点需谨记。在调用实时 JVM 时，请使用以下选项:

```
java -Xrealtime
```

在为实时 JVM 运行诊断工具时必须使用该选项。例如，要显示 IBM WebSphere Real Time for RT Linux JVM 的已注册转储代理进程，请输入:

```
java -Xrealtime -Xdump:what
```

此处作为补充信息提供了关于对 IBM WebSphere Real Time for RT Linux 使用这些工具的任何其他差异，还一起提供了用来帮助您进行诊断的样本输出。

要查看 IBM SDK for Java 7 所生成的诊断信息的摘要，请访问 Summary of diagnostic information。

使用 IBM Monitoring and Diagnostic Tools for Java

IBM 提供了工具和文档来帮助您了解、监视和诊断使用 IBM JRE 的应用程序中的问题。

以下是可用的工具:

- 运行状况中心
- 垃圾回收和内存可视化器
- 交互式诊断数据资源管理器
- 内存分析器

垃圾回收和内存可视化器

垃圾回收和内存可视化器 (GCMV) 帮助您了解 Java 应用程序的内存使用、垃圾回收行为以及性能。

GCMV 解析和绘制来自不同类型日志的数据，包括以下类型:

- 详细垃圾回收日志。

- 跟踪垃圾回收日志，通过使用 -Xtgc 参数生成。
- 本机内存日志，通过使用 ps、**svmon** 或 **perfmon** 系统命令生成。

该工具有助于诊断问题（例如内存泄漏）、分析各种可视格式的数据以及提供调优建议。

GCMV 是以 IBM Support Assistant (ISA) 附加组件的形式提供的。有关该附加组件的安装和入门信息，请参阅：http://www.ibm.com/developerworks/java/jdk/tools/gcmv/。

IBM 信息中心提供了有关 GCMV 的进一步信息。

运行状况中心

“运行状况中心”是用于监视正在运行的 Java 虚拟机 (JVM) 状态的诊断工具。

此工具被分成两个部分来提供:

- 可从正在运行的应用程序中收集数据的 Health Center 代理程序。
- 会连接到代理程序的基于 Eclipse 的客户机。客户机可解释数据，还能提供可提高受监控应用程序性能的建议。

运行状况中心是以 IBM Support Assistant (ISA) 附加组件的形式提供的。有关该附加组件的安装和入门信息，请参阅：http://www.ibm.com/developerworks/java/jdk/tools/healthcenter/。

IBM 信息中心提供了有关运行状况中心的进一步信息。

交互式诊断数据资源管理器

交互式诊断数据资源管理器 (IDDE) 是转储查看器（**jdmpview** 命令）的基于 GUI 的替代方法。 IDDE 提供的功能与转储查看器相同，但还具有额外的支持功能，例如保存命令输出的功能。

使用 IDDE 可以更轻松地浏览和检查 JVM 生成的转储文件。在 IDDE 中，您可以在调查日志中输入命令来浏览转储文件。调查日志提供的支持包括以下项:

- 命令帮助
- 自动完成文本和某些参数，例如类名
- 保存命令和输出的功能，您随后可以将其发送给其他人
- 突出显示的文本和问题标记
- 添加个人注释的功能
- IDDE 中支持使用 Memory Analyzer

IDDE 是以 IBM Support Assistant (ISA) 附加组件的形式提供的。有关该附加组件的安装和入门信息，请参阅 developerWorks® 上的 IDDE概述。

IBM信息中心提供了与 IDDE 有关的更多信息。

内存分析器

内存分析器帮助您分析使用操作系统级别转储和“可移植堆转储 (PHD)”的 Java 堆。

该工具可以分析包含上百万对象的转储，从而提供以下信息:

- 对象的保留大小。

- 阻止垃圾回收器回收对象的进程。
- 自动截取可疑的泄漏的报告。

该工具基于 Eclipse Memory Analyzer (MAT) 项目，并且使用 IBM Diagnostic Tool Framework for Java (DTFJ) 功能部件以启用对 IBM JVM 中转储的处理。

内存分析器是以 IBM Support Assistant (ISA) 附加组件的形式提供的。有关该附加组件的安装和入门信息，请参阅：http://www.ibm.com/developerworks/java/jdk/tools/memoryanalyzer/。

IBM 信息中心提供了有关内存分析器的进一步信息。

使用转储代理程序

在 JVM 初始化期间设置转储代理程序。这使您能够使用在 JVM 中发生的事件，例如，垃圾回收、线程启动或 JVM 终止来启动转储或启动外部工具。

《IBM SDK for Java V7 用户指南》包含有关转储代理程序的有用指导，包括：

- 使用 **-Xdump** 选项
- 转储代理程序
- 转储事件
- 对转储代理程序的高级控制
- 转储代理程序令牌
- 缺省转储代理程序
- 除去转储代理程序
- 转储代理程序环境变量
- 信号映射
- 转储代理程序缺省位置

您可以在此处查找信息：IBM SDK for Java 7 - 使用转储代理程序。

此处为 IBM WebSphere Real Time for RT Linux 提供了补充信息：

转储事件

转储代理程序由 JVM 操作期间发生的事件触发。对于 IBM WebSphere Real Time for RT Linux，慢事件的缺省值为 5 毫秒。

可过滤某些事件来提高输出的相关性。请参阅 第 106 页的『filter 选项』以获取更多信息。

注：卸装和扩展事件当前不会在 WebSphere Real Time 中发生。类位于永久内存中，不能对其进行卸装。

注：gpf 和 abort 事件不能触发堆转储、准备堆 (request=prepwalk) 或压缩堆 (request=compact)。

下表显示了可用作转储代理程序触发器的事件：

事件	触发条件...	过滤操作
gpf	发生一般保护故障 (GPF) 。	
user	JVM 从操作系统收到 SIGQUIT 信号。	
abort	JVM 从操作系统收到 SIGABRT 信号。	
vmstart	虚拟机已启动。	
vmstop	虚拟机已停止。	过滤退出码；例如，**filter=#129..#192#-42#255**
load	已装入类。	过滤类名；例如，**filter=java/lang/String**
unload	已卸装类。	
throw	抛出异常。	过滤异常类名；例如，**filter=java/lang/OutOfMem***
catch	捕获到异常。	过滤异常类名；例如，**filter=*Memory***
uncaught	应用程序未捕获到 Java 异常。	过滤异常类名；例如，**filter=*MemoryError**
systhrow	Java 异常将由 JVM 抛出。这与“throw”事件不同，因为它仅针对在 JVM 内部检测到的错误条件触发。	过滤异常类名；例如，**filter=java/lang/OutOfMem***
thrstart	启动了新线程。	
blocked	线程被阻止。	
thrstop	线程停止。	
fullgc	启动了垃圾收集循环。	
slow	线程响应内部 JVM 请求的时间超过 5ms。	更改被视为事件很慢的时间量；例如，如果线程响应内部 JVM 请求的时间超过 300ms，那么将触发 **filter=#300ms**。
allocation	为 Java 对象分配了与给定过滤器规范相匹配的大小	过滤对象大小；必须已提供过滤器。例如，**filter=#5m** 将在大于 5 Mb 的对象上触发。也支持使用范围；例如，**filter=#256k..512k** 将在大小介于 256 Kb 到 512 Kb 之间的对象上触发。
traceassert	JVM 中发生内部错误	不适用。
corruptcache	JVM 发现共享类高速缓存已损坏。	不适用。

filter 选项

某些 JVM 事件在应用程序的生命周期中可出现上千次。转储代理程序可使用过滤器和范围来避免生成过多的转储。

通配符

可通过仅在过滤器的开头或结尾使用星号来在异常事件过滤器中使用通配符。以下命令无效，因为第二个星号不在结尾处:

```
-Xdump:java:events=vmstop,filter=*InvalidArgumentException#*.myVirtualMethod
```

为了使此过滤器有效，必须将其更改为:

```
-Xdump:java:events=vmstop,filter=*InvalidArgumentException#MyApplication.*
```

类装入和异常事件

可以通过 Java 类名来过滤类装入 (load) 和异常（throw、catch、uncaught 和 systhrow）事件:

```
-Xdump:java:events=throw,filter=java/lang/OutOfMem*
-Xdump:java:events=throw,filter=*MemoryError
-Xdump:java:events=throw,filter=*Memory*
```

您可以通过 Java 方法名称来过滤 throw、uncaught 和 systhrow 异常事件:

```
-Xdump:java:events=throw,filter=ExceptionClassName[#ThrowingClassName.
throwingMethodName[#stackFrameOffset]]
```

可选部分将显示在方括号中。

您可以通过 Java 方法名称来过滤捕获异常事件:

```
-Xdump:java:events=catch,filter=ExceptionClassName[#CatchingClassName.
catchingMethodName]
```

可选部分将显示在方括号中。

vmstop 事件

可以通过使用一个或多个退出码来过滤 JVM 关闭事件:

```
-Xdump:java:events=vmstop,filter=#129..192#-42#255
```

慢事件

您可以过滤慢事件以更改缺省值为 5ms 的时间阈值:

```
-Xdump:java:events=slow,filter=#300ms
```

不能将过滤器设置为慢于缺省时间的时间。

分配事件

您必须过滤分配事件以指定导致触发的对象大小。您可以将过滤器大小从零设置为 32 位平台上 32 位指针的最大值，或 64 位平台上 64 位指针的最大值。将较低的过滤器值设置为零将在所有分配上触发转储。

例如，要在大小大于 5 Mb 的分配上触发转储，请使用:

```
-Xdump:stack:events=allocation,filter=#5m
```

要在大小介于 256Kb 到 512Kb 之间的分配上触发转储，请使用:

```
-Xdump:stack:events=allocation,filter=#256k..512k
```

其他事件

如果将过滤器应用于不支持过滤的事件，将忽略过滤器。

请求选项

在启动转储代理程序之前，使用请求选项来要求 JVM 对状态进行准备。对于 IBM WebSphere Real Time for RT Linux 还有额外的请求选项: **multiple**。

下表中列出了可用选项:

选项值	描述
exclusive	请求对 JVM 的互斥存取。
compact	运行垃圾回收。此选项将在生成转储之前从堆中除去所有无法访问的对象。

选项值	描述
prepwalk	准备要移动的堆。使用此选项时，还必须指定 **exclusive**。
serial	在完成此转储之前，暂挂其他转储。
multiple	为每个 RTSJ 内存区域生成单个堆转储。
preempt	应用于 Java 转储代理程序，并控制是否强制性地先占此进程中的本地线程以收集堆栈跟踪。如果未指定此选项，那么仅在 Javadump 中收集 Java 堆栈跟踪。

例如，Javadump 的请求选项的缺省设置是 request=exclusive+preempt。要更改设置以使生成 Javadump 时不抢占线程来收集本机堆栈跟踪，请使用以下选项:

```
-Xdump:java:request=exclusive
```

一般而言，缺省请求选项就已足够。

您可以使用 + 来指定多个请求选项。例如:

```
-Xdump:heap:request=exclusive+compact+prepwalk
```

使用 Javadump

Javadump 生成的文件包含与 JVM 以及在执行期间的某个点捕获的 Java 应用程序相关的信息。例如，此信息可能与操作系统、应用程序环境、线程、堆栈、锁定和内存有关。

《IBM SDK for Java V7 用户指南》包含有关 Javadump 的有用指导，包括:

- 启用 Javadump
- 触发 Javadump
- 解释 Javadump
- 环境变量和 Javadump

您可以在此处查找信息: IBM SDK for Java 7 - 使用 Javadump。

以下主题中提供了 IBM WebSphere Real Time for RT Linux 的补充信息和样本输出。

存储管理 (MEMINFO)

MEMINFO 部分提供了有关 Memory Manager 的信息，包括堆、永久和作用域内存区域。

Javadump 的 MEMINFO 部分显示了有关 Memory Manager 的信息。请参阅使用 Metronome 垃圾回收器，获取有关 Memory Manager 组件工作方式的详细信息。

这部分的 Javadump 提供了不同存储管理值，包括:

- 可用内存量
- 已使用内存量
- 堆的当前大小
- 永久内存区域的当前大小
- 作用域内存区域的当前大小

本部分还包含垃圾回收历史数据。这些数据显示为跟踪点序列，每一个都带有时间戳记，最近的跟踪点排在最前面。

由标准 JVM 生成的 Javadump 包含“GC History”部分。该信息不会包含在使用实时 JVM 时生成的 Javadump 中。使用 **-verbose:gc** 选项或 JVM 快照跟踪获取有关 GC 行为的信息。请参阅第 124 页的『使用 verbose:gc 信息』和《IBM SDK for Java V7 用户指南》的堆代理程序部分以获取更多详细信息。

如果正在运行使用作用域内存的程序，并且抛出 OutOfMemoryError，那么 Javadump 中所列的某些内存区可能为空。当嵌套在另一个作用域内的作用域内存不足时，在生成 Javadump 时可能会删除内部作用域。要获取抛出 OutOfMemoryError 时与内存区状态相关的信息，请使用以下命令行选项运行程序:

```
-Xdump:java:events=throw,filter=java/lang/OutOfMemoryError,range=1..1
```

该命令会在抛出 OutOfMemoryError 时生成另一个 Javadump，而不是在稍后检测到未捕获的异常时。在此 Javadump 中，可以看到抛出 OutOfMemoryError 时处于活动状态的所有内存区，包括任何内部作用域。有关使用 **-Xdump** 选项的进一步信息，请参阅《IBM SDK for Java V7 用户指南》。

在 Javadump 中，分段是 Java 运行时为使用大量内存的任务分配的内存块。示例任务包括:

- 维护 JIT 高速缓存
- 存储 Java 类

Java 运行时环境还会分配其他本机内存，这些内存未在 MEMINFO 部分中列出。Java 运行时分段所使用内存总量并不必然代表 Java 运行时的完全内存占用量。Java 运行时分段由分段数据结构和相关的本机内存块构成。

以下示例显示了一些典型输出。所有值都以十六进制值形式提供。MEMINFO 部分的列标题的含义如下:

- 对象内存部分 (HEAPTYPE):

  id 空间或区域的标识。

  start 堆的此区域的开始地址。

  end 堆的此区域的结束地址。

  size 堆的此区域的大小。

  space/region
  对于仅包含 id 和名称的行，此列显示了内存空间的名称。其他情况下，列显示了内存空间的名称，后面是包含在该内存空间中的特定区域的名称。

- 内部内存部分 (SEGTYPE)，包括类内存、JIT 代码高速缓存和 JIT 数据高速缓存:

  segment
  分段控制数据结构的地址。

  start 本机内存分段的开始地址。

  alloc 本机内存分段中的当前分配地址。

  end 本机内存分段的结束地址。

  type 内部位字段，描述本机内存分段的特征。

  size 本机内存分段的大小。

```
0SECTION       MEMINFO subcomponent dump routine
NULL           ================================
NULL
```

```
1STHEAPTYPE    Object Memory
NULL           id         start      end        size       space/region
1STHEAPSPACE   0x00497030     --         --         --     Generational
1STHEAPREGION  0x004A24F0 0x02850000 0x05850000 0x03000000 Generational/Tenured Region
1STHEAPREGION  0x004A2468 0x05850000 0x06050000 0x00800000 Generational/Nursery Region
1STHEAPREGION  0x004A23E0 0x06050000 0x06850000 0x00800000 Generational/Nursery Region
NULL
1STHEAPTOTAL   Total memory:           67108864 (0x04000000)
1STHEAPINUSE   Total memory in use:    33973024 (0x02066320)
1STHEAPFREE    Total memory free:      33135840 (0x01F99CE0)
NULL
1STSEGTYPE     Internal Memory
NULL           segment    start      alloc      end        type       size
1STSEGMENT     0x073DFC9C 0x0761B090 0x0761B090 0x0762B090 0x01000040 0x00010000
       (lines removed for clarity)
1STSEGMENT     0x00497238 0x004FA220 0x004FA220 0x0050A220 0x00800040 0x00010000
NULL
1STSEGTOTAL    Total memory:           873412 (0x000D53C4)
1STSEGINUSE    Total memory in use:         0 (0x00000000)
1STSEGFREE     Total memory free:      873412 (0x000D53C4)
NULL
1STSEGTYPE     Class Memory
NULL           segment    start      alloc      end        type       size
1STSEGMENT     0x0731C858 0x0745C098 0x07464098 0x07464098 0x00010040 0x00008000
        (lines removed for clarity)
1STSEGMENT     0x00498470 0x070079C8 0x07026DC0 0x070279C8 0x00020040 0x00020000
NULL
1STSEGTOTAL    Total memory:           2067100 (0x001F8A9C)
1STSEGINUSE    Total memory in use:    1839596 (0x001C11EC)
1STSEGFREE     Total memory free:       227504 (0x000378B0)
NULL
1STSEGTYPE     JIT Code Cache
NULL           segment    start      alloc      end        type       size
1STSEGMENT     0x004F9168 0x06960000 0x069E0000 0x069E0000 0x00000068 0x00080000
NULL
1STSEGTOTAL    Total memory:           524288 (0x00080000)
1STSEGINUSE    Total memory in use:    524288 (0x00080000)
1STSEGFREE     Total memory free:           0 (0x00000000)
NULL
1STSEGTYPE     JIT Data Cache
NULL           segment    start      alloc      end        type       size
1STSEGMENT     0x004F92E0 0x06A60038 0x06A6839C 0x06AE0038 0x00000048 0x00080000
NULL
1STSEGTOTAL    Total memory:           524288 (0x00080000)
1STSEGINUSE    Total memory in use:     33636 (0x00008364)
1STSEGFREE     Total memory free:      490652 (0x00077C9C)
NULL
1STGCHTYPE     GC History
3STHSTTYPE     15:18:14:901108829 GMT j9mm.134 -   Allocation failure end: newspace=7356368/8388608
oldspace=32038168/50331648 loa=3523072/3523072
3STHSTTYPE     15:18:14:901104380 GMT j9mm.470 -   Allocation failure cycle end: newspace=7356416/8388608
oldspace=32038168/50331648 loa=3523072/3523072
3STHSTTYPE     15:18:14:901097193 GMT j9mm.65 -   LocalGC end: rememberedsetoverflow=0
causedrememberedsetoverflow=0 scancacheoverflow=0 failedflipcount=0 failedflipbytes=0 failedtenurecount=0
failedtenurebytes=0 flipcount=11454 flipbytes=991056 newspace=7356416/8388608 oldspace=32038168/50331648
loa=3523072/3523072 tenureage=1
3STHSTTYPE     15:18:14:901081108 GMT j9mm.140 -   Tilt ratio: 50
3STHSTTYPE     15:18:14:893358658 GMT j9mm.64 -   LocalGC start: globalcount=3 scavengecount=24 weakrefs=0
soft=0 phantom=0 finalizers=0
3STHSTTYPE     15:18:14:893354551 GMT j9mm.63 -   Set scavenger backout flag=false
3STHSTTYPE     15:18:14:893348733 GMT j9mm.135 -   Exclusive access: exclusiveaccessms=0.002
meanexclusiveaccessms=0.002 threads=0 lastthreadtid=0x00495F00 beatenbyotherthread=0
3STHSTTYPE     15:18:14:893348391 GMT j9mm.469 -   Allocation failure cycle start: newspace=0/8388608
oldspace=38199368/50331648 loa=3523072/3523072 requestedbytes=48
3STHSTTYPE     15:18:14:893347364 GMT j9mm.133 -   Allocation failure start: newspace=0/8388608
oldspace=38199368/50331648 loa=3523072/3523072 requestedbytes=48
3STHSTTYPE     15:18:14:866523613 GMT j9mm.134 -   Allocation failure end: newspace=2359064/8388608
oldspace=38199368/50331648 loa=3523072/3523072
3STHSTTYPE     15:18:14:866519507 GMT j9mm.470 -   Allocation failure cycle end: newspace=2359296/8388608
oldspace=38199368/50331648 loa=3523072/3523072
3STHSTTYPE     15:18:14:866513004 GMT j9mm.65 -   LocalGC end: rememberedsetoverflow=0
causedrememberedsetoverflow=0 scancacheoverflow=0 failedflipcount=5056 failedflipbytes=445632
failedtenurecount=0 failedtenurebytes=0 flipcount=9212 flipbytes=6017148 newspace=2359296/8388608
oldspace=38199368/50331648 loa=3523072/3523072 tenureage=1
3STHSTTYPE     15:18:14:866493839 GMT j9mm.140 -   Tilt ratio: 64
3STHSTTYPE     15:18:14:859814852 GMT j9mm.64 -   LocalGC start: globalcount=3 scavengecount=23 weakrefs=0
soft=0 phantom=0 finalizers=0
3STHSTTYPE     15:18:14:859808692 GMT j9mm.63 -   Set scavenger backout flag=false
3STHSTTYPE     15:18:14:859801848 GMT j9mm.135 -   Exclusive access: exclusiveaccessms=0.004
```

```
meanexclusiveaccessms=0.004 threads=0 lastthreadtid=0x00495F00 beatenbyotherthread=0
3STHSTTYPE     15:18:14:859801163 GMT j9mm.469 -   Allocation failure cycle start: newspace=0/10747904
oldspace=38985800/50331648 loa=3523072/3523072 requestedbytes=232
3STHSTTYPE     15:18:14:859800479 GMT j9mm.133 -   Allocation failure start: newspace=0/10747904
oldspace=38985800/50331648 loa=3523072/3523072 requestedbytes=232
3STHSTTYPE     15:18:14:652219028 GMT j9mm.134 -   Allocation failure end: newspace=2868224/10747904
oldspace=38985800/50331648 loa=3523072/3523072
3STHSTTYPE     15:18:14:650796714 GMT j9mm.470 -   Allocation failure cycle end: newspace=2868224/10747904
oldspace=38985800/50331648 loa=3523072/3523072
3STHSTTYPE     15:18:14:650792607 GMT j9mm.475 -   GlobalGC end: workstackoverflow=0 overflowcount=0
memory=41854024/61079552
3STHSTTYPE     15:18:14:650784052 GMT j9mm.90 -   GlobalGC collect complete
3STHSTTYPE     15:18:14:650780971 GMT j9mm.57 -   Sweep end
3STHSTTYPE     15:18:14:650611567 GMT j9mm.56 -   Sweep start
3STHSTTYPE     15:18:14:650610540 GMT j9mm.55 -   Mark end
3STHSTTYPE     15:18:14:645222792 GMT j9mm.54 -   Mark start
3STHSTTYPE     15:18:14:645216632 GMT j9mm.474 -   GlobalGC start: globalcount=2
            (lines removed for clarity)
NULL
NULL ------------------------------------------------------------------------
```

线程和堆栈跟踪 (THREADS)

对于应用程序员而言，Java 转储最有用的部分之一是 THREADS 节。该节显示 Java 线程、本机线程和堆栈跟踪的列表。对于 IBM WebSphere Real Time for RT Linux，还会显示实时线程和非堆实时线程。

Java 线程由操作系统的本机线程执行。每个线程用诸如以下的行表示:

```
"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)
```

使用 **ps** 命令时，Java 线程名称在操作系统中是可见的。更多关于使用 **ps** 命令的信息，请参阅第 90 页的『常用调试方法』。

第一行中的属性包括 JVM 线程结构和 Java 线程对象的线程名称、地址，以及线程状态和 Java 线程优先级。第二行中的属性包括本机操作系统线程标识、本机操作系统线程优先级和本机操作系统调度策略。

线程名称可在以下三种方式中可见:

- 列出在 javacore 文件中。并非所有线程都会列出在 javacore 文件中。
- 使用 **ps** 命令列出操作系统中的线程时。
- 使用 java.lang.Thread.getName() 方法时。

下表提供了有关 IBM WebSphere Real Time for RT Linux 线程名称的信息。

表 13. IBM WebSphere Real Time for RT Linux 中的线程名称

线程详细信息	线程名称
由垃圾回收模块使用的内部 JVM 线程，用于分派由辅助线程执行的对象最终化。	Finalizer master
由垃圾回收器使用的警报线程。	GC Alarm
用于垃圾回收的从属线程。	GC Slave
由即时编译器模块使用的内部 JVM 线程，用于对应用程序中方法的使用情况进行采样。	IProfiler

表 13. IBM WebSphere Real Time for RT Linux 中的线程名称 (续)

线程详细信息	线程名称
由 VM 使用的线程，用于管理应用程序收到的信号，而无论信号由内部还是外部生成。	Signal Reporter
用于编译 Java 代码的内部 JVM 线程。	JIT Compilation Thread
用于允许将 JVMTI 代理程序附加到运行中的 JVM 的内部 JVM 线程。	Attach API wait loop

在 Java 代码中创建的实时线程 (javax.realtime.RealtimeThread) 的缺省名称为 RTThread-x，其中“x”是线程编号。

非堆实时线程的缺省名称为 NHRTThread-x，其中“x”是线程编号。

Java 线程优先级会根据平台映射至操作系统优先级值。较大的 Java 线程优先级值表明该线程具有较高的优先级。换言之，该线程会比低优先级线程更频繁地运行。要获取这如何对 Java 线程、实时线程和无堆实时线程产生影响的进一步详细信息，请参阅第 11 页的『优先级映射和继承』。

状态值可以是:

- R - 可运行 - 条件满足时，该线程将运行。
- CW - 等待条件 - 该线程正在等待。例如，由于:
  - 调用了 sleep()
  - 针对 I/O 已阻止了该线程
  - 调用了 wait() 方法，以等待通知监视器
  - 该线程通过 join() 调用正在与另一个线程同步
- S – 已暂挂 – 该线程已被另一个线程暂挂。
- Z – 僵死 – 该线程已被结束。
- P – 停放 – 该线程已因新的并发 API (java.util.concurrent) 而被停放。
- B – 已阻塞 – 该线程正在等待获取其他对象当前拥有的锁。

如果某个线程已停发或已阻塞，则输出中包含针对该线程的一行，以 3XMTHREADBLOCK 开始，列出该线程要等待的资源，以及当前拥有该资源的线程（如果可能）。有关更多信息，请参阅《IBM SDK for Java V7 用户指南》中关于已阻塞线程的主题。

当您启动 Javadump 以获取诊断信息时，JVM 会在生成 javacore 之前停顿 Java 线程。TITLE 部分的 1TIPREPSTATE 行中会显示 exclusive_vm_access 的准备状态。

```
1TIPREPSTATE Prep State: 0x4 (exclusive_vm_access)
```

当触发 javacore 时运行 Java 代码的线程目前处于 CW（条件等待）状态。

```
3XMTHREADINFO      "main" J9VMThread:0x41481900, j9thread_t:0x002A54A4, java/lang/Thread:0x004316B8,
state:CW, prio=5
3XMTHREADINFO1            (native thread ID:0x904, native priority:0x5, native policy:UNKNOWN)
3XMTHREADINFO3           Java callstack:
4XESTACKTRACE                at java/lang/String.getChars(String.java:667)
4XESTACKTRACE                at java/lang/StringBuilder.append(StringBuilder.java:207)
```

javacore LOCKS 部分显示了这些线程正在内部 JVM 锁定中等待。

```
2LKREGMON          Thread public flags mutex lock (0x002A5234): <unowned>
3LKNOTIFYQ            Waiting to be notified:
3LKWAITNOTIFY            "main" (0x41481900)
```

使用 Heapdump

术语 Heapdump 描述了 IBM Virtual Machine for Java 机制，该机制生成 Java 堆上所有活动对象的转储，即，正在运行的 Java 应用程序使用的那些活动对象。

《IBM SDK for Java V7 用户指南》包含有关 Heapdump 的有用指导，包括:

- 获取 Heapdump
- 用于处理 Heapdump 的工具
- 使用 **-Xverbose:gc** 来获取堆信息
- 环境变量和 Heapdump
- 文本（经典）Heapdump 文件格式
- 可移植堆转储 (PHD) 文件格式

您可以在此处查找信息: IBM SDK for Java 7 - 使用 Heapdump。

IBM WebSphere Real Time for RT Linux 的补充信息:

为实时 JVM 启用多个 Heapdump

缺省情况下，生成的 Heapdump 是单个文件，包含有关所有内存区域（堆内存、永久内存和作用域内存）中所有 Java 对象的信息。生成多个转储的主要原因，是为了可以使用不需要修改的传统 Heapdump 工具来分析每个独立的堆区域。

关于此任务

缺省情况下，Heapdump 包含 JVM 内存区域（堆内存、不朽内存和作用域内存）中所有对象的相关信息。通过使用 **request=multiple** 选项与 **-Xdump:heap**，您可以获取不同的 Heapdump，其中包含有关每个内存区域中的 Java 对象的信息。请注意，您还必须重复请求选项的缺省设置，因此需要指定 `request=multiple+exclusive+prepwalk+compact`。这样会产生一组 Heapdump，名称中包含额外字段以指示特定的内存区域:

```
heapdump.%id.%Y%m%d.%H%M%S.%pid.phd
```

其中 *%id* 将 Heapdump 文件识别为包含堆内存、永久内存或作用域限定内存的特定区域中的对象。

以下名称代表了 4 种类型的堆: “缺省”、“永久”、“作用域”和“其他”。Heapdump 代码将堆标签中的 %id 替换为这些名称之一，并加上标识符数字（通常为数字），例如: `heapdump.Immortal12994208.20060807.093653.7684.txt.`

示例

```
java -Xrealtime -Xdump:heap:defaults:request=multiple+exclusive+compact+prepwalk
     <java program>
```

使用此额外选项会以可移植 Heapdump (phd) 格式生成多个 Heapdump。

```
java -Xrealtime -Xdump:heap:defaults:request=multiple+exclusive+compact+prepwalk,
     opts=CLASSIC <java program>
```

使用此额外选项会以 CLASSIC 文本格式生成多个 Heapdump。

使用 -Xdump:what 选项可以在 JVM 启动时显示转储代理程序，有助于检查已应用的转储选项。

文本（经典）Heapdump 文件格式

文本或经典 Heapdump 是堆中所有对象实例的列表，包括对象类型、大小以及对象之间的引用。

头记录

头记录是包含版本信息字符串的单条记录。

```
// Version: <版本字符串包含 SDK 级别、平台和 JVM 构建级别>
```

示例:

```
// Version: J2RE 7.0 IBM J9 2.6 Linux x86-32 build 20101016_024574_lHdRSr
```

对象记录

对象记录是多条记录，每条记录用于堆中的每个对象实例，提供对象地址、大小、类型以及对象的引用。

```
<object address, in hexadecimal> [<length in bytes of object instance, in decimal>]
OBJ <object type> <class block reference, in hexadecimal>
<heap reference, in hexadecimal <heap reference, in hexadecimal> ...
```

对象地址和堆引用位于堆中，但是类块地址在堆外。将列出在对象实例中找到的所有引用，包括那些为空值的引用。对象类型是包括软件包或原始阵列的类名或类阵列类型，通过其标准的 JVM 类型签名显示，请参阅 第 116 页的『Java VM 类型签名』。对象记录也可包含其他的类块引用，通常在反映类实例的情况下。

示例:

类型为 java/lang/String 的对象实例，长度为 28 字节:

```
0x00436E90 [28] OBJ java/lang/String
```

java/lang/String 的类块地址，后跟对 char 阵列实例的引用:

```
0x415319D8 0x00436EB0
```

类型为 char 阵列的对象实例，长度为 44 字节:

```
0x00436EB0 [44] OBJ [C
```

char 阵列的类块地址:

```
0x41530F20
```

类型阵列为 java/util/Hashtable Entry 内部类的对象:

```
0x004380C0 [108] OBJ [Ljava/util/Hashtable$Entry;
```

类型为 java/util/Hashtable Entry 内部类的对象:

```
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
```

类块地址和堆引用，包括空引用:

```
0x4158CB88 0x004219B8 0x004341F0 0x00000000
```

类记录

类记录是多条记录，每条记录用于每个已装入的类，提供类块地址、大小、类型以及类的引用。

```
<class block address, in hexadecimal> [<length in bytes of class block, in decimal>]
CLS <class type>
<class block reference, in hexadecimal> <class block reference, in hexadecimal> ...
<heap reference, in hexadecimal> <heap reference, in hexadecimal>...
```

类块地址和类块引用在堆的外部，但是类记录还可以将引用包含到堆中，这通常适用于静态类数据成员。将列出在类块中找到的所有引用，包括那些为空值的引用。类类型是包括软件包或原始阵列的类名或类阵列类型，通过其标准的 JVM 类型签名显示，请参阅 第 116 页的『Java VM 类型签名』。

示例:

用于类 java/lang/Runnable 的长度为 32 字节的类块:

```
0x41532E68 [32] CLS java/lang/Runnable
```

对其他类块的引用和堆引用（包括空引用）:

```
0x4152F018 0x41532E68 0x00000000 0x00000000 0x00499790
```

用于类 java/lang/Math 的长度为 168 字节的类块:

```
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
```

尾部记录 1

尾部记录 1 是包含记录计数的单条记录。

```
// Breakdown - Classes: <类记录计数（十进制）>,
Objects: <对象记录计数（十进制）>,
ObjectArrays: <对象数组记录计数（十进制）>,
PrimitiveArrays: <原语数组记录计数（十进制）>
```

示例:

```
// Breakdown - Classes: 321, Objects: 3718, ObjectArrays: 169,
PrimitiveArrays: 2141
```

尾部记录 2

尾部记录 2 是包含总数的单条记录。

```
// EOF:  Total 'Objects',Refs(null) :
<对象总计（十进制）>,
<引用总计（十进制）>
(空引用总计（十进制）>)
```

示例:

```
// EOF:  Total 'Objects',Refs(null) : 6349,23240(7282)
```

Java VM 类型签名

Java VM 类型签名是 Java 类型的缩写，如下表中所示:

Java VM 类型签名	Java 类型
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double
L <*标准类*> ;	<*标准类*>
[<*类型*>	<*类型*>[ ]（<*类型*> 的阵列）
（<*参数类型*>）<*返回类型*>	*方法*

使用系统转储和转储查看器

JVM 可以在可配置条件下生成本机系统转储，也被称为核心转储。系统转储通常很大。用于分析系统转储的大多数工具是特定于平台的。使用 **gdb** 工具可以分析 Linux 上的系统转储。

《IBM SDK for Java V7 用户指南》包含有关使用系统转储和转储查看器的有用指导，包括:

- 系统转储概述
- 系统转储缺省值
- 使用转储查看器
  - 使用 **jextract**
  - 使用转储查看器跟踪问题
  - **jdmpview** 中的可用命令
  - 示例会话
  - **jdmpview** 命令快速参考

您可以在此处查找信息: IBM SDK for Java 7 - 使用系统转储和转储查看器。

IBM WebSphere Real Time for RT Linux 的补充信息:

使用 jextract

当从实时 JVM 处理系统转储时，您必须包含 **-Xrealtime** 选项。例如:

```
jextract -Xrealtime <core file name> [<zip_file>]
```

当您在产生转储的 JVM 之外的其他 JVM 上运行 **jextract** 时，那么您将看到以下错误消息:

```
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).
```

类似的，如果您使用标准 JVM 来运行 Java，但使用 **jextract** 处理转储时使用了 **-Xrealtime** 选项，也会生成该消息。

jdmpview 中的可用命令

jdmpview 是一种交互式的命令行工具，用于浏览来自 JVM 系统转储的信息以及执行各种分析功能。

info jitm

列出 AOT 和 JIT 编译的方法及其地址:

- 方法名称和特征符
- 方法起始地址
- 方法结束地址

有关所有其他命令选项，请参阅《IBM SDK for Java V7 用户指南》。

跟踪 Java 应用程序和 JVM

JVM 跟踪是 IBM WebSphere Real Time for RT Linux 中提供的一种跟踪工具，其对性能的影响最小。在大多数情况下，将以压缩二进制格式保存跟踪数据，这种格式的数据可使用提供的 Java 格式化程序进行格式化。

缺省情况下将启用跟踪，并且一小组跟踪点将转至内存缓冲区。您可以在运行时使用级别、组件、组名或个别跟踪点标识来启用跟踪点。

《IBM SDK for Java V7 用户指南》包含有关跟踪应用程序的详细信息，包括:

- 可跟踪哪些内容
- 跟踪点类型
- 缺省跟踪
- 记录跟踪数据
- 控制跟踪
- 跟踪 Java 应用程序
- 跟踪 Java 方法

在跟踪 IBM WebSphere Real Time for RT Linux 时，您必须在包含跟踪选项时正确的调用实时 JVM。例如，在指定跟踪选项时，请输入:

```
java -Xrealtime -Xtrace:<options>
```

您可以在此处查找 IBM SDK for Java V7 信息: 跟踪 Java 应用程序和 JVM。

JIT 和 AOT 问题确定

您可以使用命令行选项帮助诊断 JIT 和 AOT 编译器问题，以及调优性能。

尽管 IBM WebSphere Real Time for RT Linux 与 IBM SDK for Java V7 共享一些公共组件，但 JIT 和 AOT 的行为是不同的。本部分涵盖了针对 IBM WebSphere Real Time for RT Linux 上的 JIT 和 AOT 问题的故障诊断。

诊断 JIT 或 AOT 问题

有效字节码偶尔可能会编译为无效的本机代码，从而导致 Java 程序失败。通过确定 JIT 或 AOT 编译器是否出错，以及出错的位置（如果出错），您可以为 Java 服务团队提供有用的帮助。

关于此任务

要确定填充共享类缓存时编译的方法，请在 admincache 命令行上使用 **-Xaot:verbose** 选项。例如:

```
admincache -Xrealtime -Xaot:verbose -populate -aot my.jar -cp <My Class Path>
```

本部分描述了如何确定自己的问题是否与编译器相关。本部分还推荐了一些用于解决编译器相关问题的可能的变通方法和调试方法。

禁用 JIT 或 AOT 编译器:

如果您怀疑 JIT 或 AOT 编译器中出现问题，那么可以禁用编译，以查看问题是否仍然存在。如果问题仍然存在，那么可知问题不是由编译器引起的。

关于此任务

缺省情况下，启用 JIT 编译器。同时也会启用 AOT 编译器，但只有在启用了共享类后该编译器才处于活动状态。由于效率原因，不会编译 Java 应用程序中的所有方法。JVM 保留应用程序中每种方法的调用计数；每当调用并解释方法时，就会增加该方法的调用计数。当计数达到编译阈值时，就会在本机编译和执行该方法。

调用计数机制将方法的编译分布在应用程序的整个生命周期内，赋予最常使用的方法较高的优先级。一些不常使用的方法可能根本不会被编译。因此，当 Java 程序失败时，问题可能出在 JIT 或 AOT 编译器中，也可能出在 JVM 中的其他位置。

诊断故障的第一步就是确定问题的位置。要执行该操作，必须先在纯解释方式下运行 Java 程序（即禁用 JIT 和 AOT 编译器）。

过程

1. 除去命令行中的任何 **-Xjit** 和 **-Xaot** 选项（及伴随的参数）。
2. 使用 **-Xint** 命令行选项禁用 JIT 和 AOT 编译器。 由于性能原因，请勿在生产环境中使用 **-Xint** 选项。

下一步做什么

在禁用编译的情况下运行 Java 程序会导致以下一种情况:

- 故障仍然存在。问题不在 JIT 或 AOT 编译器中。在某些情况下，程序可能以不同的方式失败；不过，问题与编译器无关。
- 故障消失。问题很有可能在 JIT 或 AOT 编译器中。

  如果您未在使用共享类，那么出错的是 JIT 编译器。如果您正在使用共享类，那么必须在只启用 JIT 编译的情况下运行应用程序，从而确定出错的编译器。使用 **-Xnoaot** 选项（而不是 **-Xint** 选项）运行应用程序。这将导致以下一种情况:

  – 故障仍然存在。JIT 编译器中出现问题。您还可以使用 **-Xnojit** 选项（而不是 **-Xnoaot** 选项），以确保只有 JIT 编译器出错。

– 故障消失。AOT 编译器中出现问题。

选择性地禁用 JIT 编译器:

如果 Java 程序故障指向 JIT 编译器问题，您可以尝试进一步缩小问题范围。

关于此任务

缺省情况下，JIT 编译器在各种优化级别对方法进行优化。不同的优化选项将根据不同方法的调用计数应用于这些方法。经常调用的方法会在较高级别上进行优化。通过更改 JIT 编译器参数，您可以控制对方法进行优化时采用的优化级别。您可以确定优化器是否发生故障，如果发生故障，可以确定发生问题的优化。

将 JIT 参数指定为用逗号分隔的列表，并附加到 **-Xjit** 选项。语法为: **-Xjit**:<*param1*>,<*param2*>=<*value*>。例如:

```
java -Xjit:verbose,optLevel=noOpt HelloWorld
```

将运行 HelloWorld 程序，对 JIT 启用详细输出，并使 JIT 生成本机代码而不执行任何优化。

按照以下步骤来确定编译器的哪个部分导致故障:

过程

1. 设置 JIT 参数 **count=0**，以便将编译阈值更改为零。此参数将导致先编译每个 Java 方法，然后再运行这些方法。 仅当诊断问题时才应使用 **count=0**，这是因为此参数会显著增加编译的方法数，包括编译那些并不经常使用的方法。额外的编译将使用更多计算资源，并导致应用程序速度减慢。 如果指定了 **count=0**，那么到达问题区域时，应用程序将立即失败。在某些情况下，采用 **count=1** 可以更可靠地重现故障。
2. 对 JIT 编译器参数添加 **disableInlining**。 **disableInlining** 禁止生成较大较复杂的代码。 如果问题不再发生，请在 Java 服务团队分析并解决编译器问题期间使用 **disableInlining** 作为变通方法。
3. 通过添加 **optLevel** 参数降低优化级别，并再次运行该程序，直到故障不再发生为止，或者使用“noOpt”级别。对于 JIT 编译器问题，请从“scorching”入手并依次使用下一个列表项。 这些优化级别按降序排列为:
   a. scorching
   b. veryHot
   c. hot
   d. warm
   e. cold
   f. noOpt

下一步做什么

如果其中一项设置导致故障消失，表明存在可以采用的变通方法。此变通方法可以在 Java 服务团队分析并解决编译器问题期间暂时采用。如果从 JIT 参数列表中除去 **disableInlining** 并不会导致故障重新出现，那么可以执行该操作以提高性能。遵循第 120 页的『查找失败的方法』中的指示信息，提高变通方法的性能。

如果在“noOpt”优化级别仍发生故障，那么作为变通方法，您必须禁用 JIT 编译器。

查找失败的方法:

确定将会导致 JIT 或 AOT 编译器编译方法时触发故障的最低优化级别后，可以确定哪一部分的 Java 程序在编译时将会引起故障。随后，您可以指示编译器将变通方法限制为某一特定方法、类或程序包，从而允许编译器正常编译程序的其余部分。对于 JIT 编译器故障，如果故障与 **-Xjit:optLevel=noOpt** 有关，那么您还可以指示编译器根本不编译导致故障的方法。

开始之前

如果您看到类似如下示例的错误输出，那么可以使用它来确定失败的方法:

```
Unhandled exception
Type=Segmentation error vmState=0x00000000
Target=2_30_20050520_01866_BHdSMr (Linux 2.4.21-27.0.2.EL)
CPU=s390x (2 logical CPUs) (0x7b6a8000 RAM)
J9Generic_Signal_Number=00000004 Signal_Number=0000000b Error_Value=4148bf20 Signal_Code=00000001
Handler1=00000100002ADB14 Handler2=00000100002F480C InaccessibleAddress=0000000000000000
gpr0=0000000000000006 gpr1=0000000000000006 gpr2=0000000000000000 gpr3=0000000000000006
gpr4=0000000000000001 gpr5=0000000080056808 gpr6=0000010002BCCA20 gpr7=0000000000000000
......
Compiled_method=java/security/AccessController.toArrayOfProtectionDomains([Ljava/lang/Object;
Ljava/security/AccessControlContext;)[Ljava/security/ProtectionDomain;
```

重要的行为:

vmState=0x00000000
指明失败的代码不是 JVM 运行时代码。

Module= or Module_base_address=
不出现在输出中（可能为空或 0），这是因为代码是由 JIT 编译的，位于任何 DDL 或库之外。

Compiled_method=
指明为其生成所编译代码的 Java 方法。

关于此任务

如果输出未指明失败的方法，那么可以按照以下步骤来确定失败的方法:

过程

1. 在对 **-Xjit** 或 **-Xaot** 选项添加了 JIT 参数 **verbose** 和 **vlog=**<*filename*> 的情况下运行 Java 程序。利用这些参数，编译器会在名为 <*filename*>.<*date*>.<*time*>.<*pid*>的日志文件（也称为*界限文件*）中列出已编译的方法。一般的界限文件包含与所编译方法相对应的行，类似如下:

   ```
   + (hot) java/lang/Math.max(II)I @ 0x10C11DA4-0x10C11DDD
   ```

   编译器在接下来的步骤中忽略未以加号开头的行，您可以从文件中除去这些行。为其从共享类高速缓存装入 AOT 代码的方法以 `+ (AOT load)` 开头。
2. 在指定了 JIT 或 AOT 参数 **limitFile=**(<*filename*>,<*m*>,<*n*>) 的情况下再次运行程序，其中 <*filename*> 是限制文件的路径，<*m*> 和 <*n*> 是行号，用于指示限制文件中第一个和最后一个应该编译的方法。 编译器只编译界限文件中 <*m*> 行到 <*n*> 行上所列出的方法。不会编译未列示在限制文件中的方法以及范围外部的行中列示的方法，并且不会装入共享数据高速缓存中这些方法的 AOT 代码。 如果程序不再失败，那么您在最后一次迭代中除去的一个或多个方法必然是导致故障的原因。

3. 使用不同的 <*m*> 和 <*n*> 值按需多次重复该进程，以找到导致故障的必须编译的最少数量方法集。 通过每次将选定的行数减半，您可以对失败的方法执行对分搜索。一般来说，您可以将文件减少到一行。

下一步做什么

找到失败的方法后，可以仅针对失败的方法禁用 JIT 或 AOT 编译器。例如，如果 JIT 通过 **optLevel=hot** 进行编译时，java/lang/Math.max(II)I 方法导致程序失败，那么您可以通过以下选项运行程序:

```
-Xjit:{java/lang/Math.max(II)I}(optLevel=warm,count=0)
```

以便只在『warm』优化级别编译失败的方法，但正常编译所有其他方法。

如果 JIT 在『noOpt』优化级别编译方法时失败，您可以使用 **exclude**={<*method*>} 参数将其全部从编译中排除:

```
-Xjit:exclude={java/lang/Math.max(II)I}
```

如果共享数据高速缓存中装入 AOT 代码时某方法导致程序失败，那么可以使用 **exclude**={<*method*>} 参数将该方法从 AOT 装入中排除。

```
-Xaot:exclude={java/lang/Math.max(II)I}
```

AOT 方法只能在 **admincache** 填充步骤时编译到共享类高速缓存。防止 AOT 装入是诊断这些方法所存在问题的最佳途径。

确定 JIT 和 AOT 编译故障:

对于 JIT 编译器故障，分析错误输出以确定当 JIT 编译器尝试编译方法时是否发生错误。

如果 JVM 崩溃，并且您可以看到故障发生在 JIT 库 (libj9jit26.so) 中，则 JIT 编译器可能在尝试编译某个方式时失败。

如果您看到类似如下示例的错误输出，那么可以使用它来确定失败的方法:

```
Unhandled exception
Type=Segmentation error vmState=0x00050000
Target=2_30_20051215_04381_BHdSMr (Linux 2.4.21-32.0.1.EL)
CPU=ppc64 (4 logical CPUs) (0xebf4e000 RAM)
J9Generic_Signal_Number=00000004 Signal_Number=0000000b Error_Value=00000000 Signal_Code=00000001
Handler1=0000007FE05645B8 Handler2=0000007FE0615C20
R0=E8D4001870C00001 R1=0000007FF49181E0 R2=0000007FE2FBCEE0 R3=0000007FF4E60D70
R4=E8D4001870C00000 R5=0000007FE2E02D30 R6=0000007FF4C0F188 R7=0000007FE2F8C290
......
Module=/home/test/sdk/jre/bin/libj9jit26.so
Module_base_address=0000007FE29A6000
......
Method_being_compiled=com/sun/tools/javac/comp/Attr.visitMethodDef(Lcom/sun/tools/javac/tree/
JCTree$JCMethodDecl;)
```

重要的行为:

vmState=0x00050000

指明 JIT 编译器正在编译代码。有关 vmState 代码编号的列表，请参阅《IBM SDK for Java V7 用户指南》、http://publib.boulder.ibm.com/infocenter/java7sdk/v7r0/topic/com.ibm.java.lnx.70.doc/diag/tools/javadump_tags_info.html中的 Javadump 标记表。

Module=/home/test/sdk/jre/bin/libj9jit26.so
指示错误发生在 JIT 编译器模块 libj9jit26.so 中。

Method_being_compiled=
指明正在编译的 Java 方法。

如果输出未指明失败的方法，那么使用 **verbose** 选项以及以下一些额外设置:

```
-Xjit:verbose={compileStart|compileEnd}
```

这些 **verbose** 设置会报告 JIT 或 AOT 编译器开始编译方法的时间以及结束时间。如果 JIT 或 AOT 编译器在特定方法上失败（即它开始编译，但在可以结束之前崩溃），那么在 **-Xjit** 或 **-Xaot** 命令行选项上使用 **exclude** 参数，将其排除在 JIT 或 AOT 编译之外（请参阅第 120 页的『查找失败的方法』）。 对于 AOT 编译问题，使用 **exclude** 选项之前破坏共享类高速缓存。 如果排除该方法能够防止崩溃，那么您可以在服务团队纠正问题时采用一种变通方法。

识别非实时方式中的 AOT 编译故障:

非实时方式中的 AOT 问题确定与 JIT 问题确定非常相似。

关于此任务

与 JIT 一样，首先利用 **-Xnoaot** 运行应用程序，确保运行应用程序时未使用 AOT 代码。

如果这样能修正问题，那么采用第 120 页的『查找失败的方法』中描述的方法重建 AOT JAR 文件，在 AOT 构建时（而不是应用程序运行时）提供 **-Xaot** 选项。

识别实时方式中的 AOT 编译故障:

AOT 问题确定使用 admincache 工具来找到问题。

关于此任务

与发生在应用程序运行时的 JIT 编译故障相比，AOT 编译故障发生在 admincache 填充步骤的过程中。

要查找问题发生的位置，请使用 **-Xnoaot** 选项来运行 admincache 工具。这样确保了应用程序不会运行提前编译的代码。

如果使用 **-Xnoaot** 选项来修正问题，请检查原始崩溃的输出。该输出提供的信息中可以识别导致问题的方法。查找类似以下的行:

```
Method_being_compiled=myAppClass.main(Ljava/lang/String;)V
```

为了避免出现问题，必须从提前编译中排除该方法。可以通过向 admincache 命令行中添加选项来实现，类似于以下示例:

```
-Xaot:exclude={myAppClass.main(Ljava/lang/String;)V}
```

这样的排除可以避免对有问题的方法进行 AOT 编译。

短时间运行的应用程序的性能

IBM JIT 编译器针对服务器上通常使用的长时间运行的应用程序进行了调优。您可以在非实时方式下使用 **-Xquickstart** 命令行选项来提高短时间运行的应用程序的性能，对于未将处理集中到几个方法中的应用程序而言尤其如此。

-Xquickstart 使 JIT 编译器在缺省情况下使用较低的优化级别，并编译较少的方法。更快速地执行少量编译可以减少应用程序的启动时间。当 AOT 编译器处于活动状态（同时启用了共享类和 AOT 编译）时，**-Xquickstart** 将导致所有选择编译的方法由 AOT 编译，这将缩短后续运行的启动时间。将 **-Xquickstart** 与那些包含使用了大量处理资源的方法的长时间运行应用程序配合使用时，可能会导致性能下降。**-Xquickstart** 的执行会根据将来发行版的变化而相应调整。

您还可以尝试通过调整 JIT 阈值（使用试错法）来改进启动时间。请参阅 第 119 页的『选择性地禁用 JIT 编译器』以获取更多信息。

-Xquickstart 不会通过 **-Xrealtime** 对 AOT 代码造成影响。

空闲期间 JVM 的行为

通过使用 **-XsamplingExpirationTime** 选项关闭 JIT 采样线程，可以减少空闲 JVM 所使用的 CPU 周期。

JIT 采样线程对运行中的 Java 应用程序进行概要分析，以发现常用的方法。采样线程的内存和处理器使用情况可忽略不计，而且当 JVM 空闲时，概要分析的频率将自动降低。

在一些情况下，您可能希望空闲的 JVM 不占用 CPU 周期。要执行此操作，请指定 **-XsamplingExpirationTime**<*time*> 选项。将 <*time*> 设置为希望采样线程运行的秒数。请慎重使用该选项；关闭该选项后，您将不能重新激活采样线程。允许采样线程长时间运行，以确定重要的优化。

诊断收集器

“诊断收集器”收集针对问题事件的 Java 诊断文件。

收集 IBM 服务所需要的文件，可以减少解决报告的问题所需要的时间。《IBM SDK for Java V7 用户指南》包含有关使用“诊断收集器”的详细信息。

您可以在此处查找信息：IBM SDK for Java 7 - 诊断收集器。

垃圾收集器诊断数据

本部分描述了如何诊断垃圾收集问题。

《IBM SDK for Java V7 用户指南》包含有关诊断垃圾收集器问题的有用指导，包括：

- 详细垃圾回收日志记录
- 使用 **-Xtgc** 跟踪垃圾回收

您可以在此处查找信息：IBM SDK for Java 7 - 垃圾收集器诊断数据。

以下部分提供了有关 IBM WebSphere Real Time for RT Linux Metronome 垃圾回收器的补充信息。

Metronome 垃圾回收器故障诊断

通过使用命令行选项，您可以控制 Metronome 垃圾回收的频率、内存耗尽异常以及显式系统调用时的 Metronome 行为。

使用 verbose:gc 信息:

您可以结合使用 **-verbose:gc** 选项和 **-Xgc:verboseGCCycleTime=N** 选项以将关于 Metronome 垃圾回收器活动的信息写到控制台。并非标准 JVM 的 **-verbose:gc** 输出中的 XML 属性，并非所有都会予以创建，也并非全部适用于 Metronome 垃圾回收器的输出。

使用 **-verbose:gc** 选项以查看堆中的最小、最大和平均可用空间。这样，您便可以检查堆的活动级别和使用情况，然后在必要时调整这些值。**-verbose:gc** 选项用于将 Metronome 统计信息写到控制台。

-Xgc:verboseGCCycleTime=N 选项用于控制检索信息的频率。它决定转储摘要的时间（毫秒）。N 的缺省值是 1000 毫秒。周期时间不表示刚好在这一时间转储摘要，而是表示在符合该时间标准的最后一次垃圾回收事件过去时进行转储。对这些统计信息的收集和显示可能会增加 Metronome 垃圾回收器暂停次数，并且随着 N 变小，暂停次数可能会变多。

定量是 Metronome 垃圾回收器活动的单个时间段，会导致应用程序的中断或暂停时间。

verbose:gc 输出示例

输入:

```
java -Xrealtime -verbose:gc -Xgc:verboseGCCycleTime=N myApplication
```

该示例显示 verbose:gc 的初始输出，其中包含版本和垃圾回收设置:

```
<verbosegc
xmlns="http://www.ibm.com/j9/verbosegc" version="R26_Java726_GA_20110716_0946_B87065">

<initialized id="1" timestamp="2011-07-27T14:17:52.277">
  <attribute name="gcPolicy" value="-Xgcpolicy:metronome"/>
  <attribute name="maxHeapSize" value="0x5800000"/>
  <attribute name="initialHeapSize" value="0x4000000"/>
  <attribute name="compressedRefs" value="false"/>
  <attribute name="pageSize" value="0x1000"/>
  <attribute name="requestedPageSize" value="0x1000"/>
  <attribute name="gcthreads" value="1"/>
  <region>
    <attribute name="regionSize" value="16384"/>
    <attribute name="regionCount" value="4096"/>
    <attribute name="arrayletLeafSize" value="2048"/>
  </region>
  <metronome>
    <attribute name="beatsPerMeasure" value="500"/>
    <attribute name="timeInterval" value="10000"/>
    <attribute name="targetUtilization" value="70"/>
    <attribute name="trigger" value="0x2000000"/>
    <attribute name="headRoom" value="0x100000"/>
  </metronome>
  <system>
    <attribute name="physicalMemory" value="12507463680"/>
    <attribute name="numCPUs" value="8"/>
    <attribute name="architecture" value="x86"/>
    <attribute name="os" value="Linux"/>
    <attribute name="osVersion" value="2.6.24.7-75ibmrt2.18"/>
```

```
  </system>
  <vmargs>
    <vmarg
name="-Xoptionsfile=/my_dir/pxi3270hrt-20110719_02/sdk/jre/lib/i386/realtime/options.default"/>
    <vmarg name="-Xjcl:jclse7b_26"/>
    <vmarg
name="-Dcom.ibm.oti.vm.bootstrap.library.path=/my_dir/pxi3270hrt-20110719_02/sdk/jre/lib/i386/realtime:/
my_dir/pxi3270hrt-2011071..."/>
    <vmarg
name="-Dsun.boot.library.path=/my_dir/pxi3270hrt-20110719_02/sdk/jre/lib/i386/realtime:/my_dir/
pxi3270hrt-20110719_02/sdk/jre/lib..."/>
    <vmarg
name="-Djava.library.path=/my_dir/pxi3270hrt-20110719_02/sdk/jre/lib/i386/realtime:/my_dir/
pxi3270hrt-20110719_02/sdk/jre/lib/i38..."/>
    <vmarg name="-Djava.home=/my_dir/pxi3270hrt-20110719_02/sdk/jre"/>
    <vmarg name="-Djava.ext.dirs=/my_dir/pxi3270hrt-20110719_02/sdk/jre/lib/ext"/>
    <vmarg name="-Duser.dir=/my_dir/pxi3270hrt-20110719_02/sdk/jre/bin"/>
    <vmarg name="_j2se_j9=1120000"
value="F76FF700"/>
    <vmarg name="-Djava.runtime.version=pxi3270hrt-20110719_02"/>
    <vmarg name="-Djava.class.path=."/>
    <vmarg name="-Xrealtime"/>
    <vmarg name="-verbose:gc"/>
    <vmarg name="-Dsun.java.launcher=SUN_STANDARD"/>
    <vmarg name="-Dsun.java.launcher.pid=5543"/>
    <vmarg name="_port_library" value="F7701B80"/>
    <vmarg name="_bfu_java" value="F77029A8"/>
    <vmarg name="_org.apache.harmony.vmi.portlib" value="08051DA0"/>
  </vmargs>
</initialized>
```

触发垃圾回收时，将发生 trigger start 事件，后跟任意数量的 heartbeat 事件，然后在触发器得到满足时发生 trigger end 事件。该示例以 verbose:gc 输出方式显示已触发的垃圾回收周期:

```
<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" />
```

可能会发生以下事件类型:

<trigger-start ...>
垃圾回收周期的开始（当已用内存量变得高于触发器阈值时）。缺省阈值为堆的 50%。intervalms 属性是上一个 trigger end 事件（id 为 -1）与此 trigger start 事件之间的时间间隔。

<trigger-end ...>
垃圾回收周期已成功将已用内存量降低至触发器阈值以下。如果垃圾回收周期已结束，但已用内存未降低到触发器阈值以下，那么将使用同一上下文标识开始新的垃圾回收周期。对于每个 trigger start 事件，都有一个具有同一上下文标识的匹配 trigger end 事件。intervalms 属性是上一个 trigger start 事件与当前 trigger end 事件之间的时间间隔。在这段时间内，一个或多个垃圾回收周期将会完成，直到已用内存降低到触发器阈值以下。

<gc-op id="28" type="heartbeat"...>
一个定期事件，它收集所涵盖时间内与所有垃圾回收定量有关的信息（关于内存和时间）。heartbeat 事件只能在一对匹配的 trigger start 与 trigger end 事件之间发生，即在活动垃圾回收周期正在进行期间发生。intervalms 属性是上一个 heartbeat 事件（id 为 -1）与该 heartbeat 事件之间的时间间隔。

<gc-op id="29" type="syncgc"...>
同步（非确定性）垃圾回收事件。请参阅第 127 页的『同步垃圾回收』

该示例中的 XML 标记具有以下含义:

<quanta ...>
脉动信号间隔期间定量暂停时间的摘要，包括暂停的长度（毫秒）。

<free-mem type="heap" ...>
脉动信号间隔期间可用堆空间量的摘要，在每个垃圾回收定量的末尾采样。

<classunload-info classloadersunloaded=9 classesunloaded=156 />
脉动信号间隔期间卸载的类装入器和类的数量。

<references type="weak" cleared="660 />
脉动信号间隔期间已清除的 Java 引用对象的数量和类型。

注:

- 如果两次脉动信号之间的时间间隔中只发生了一个垃圾回收定量，那么将仅在这一个定量的末尾对可用内存进行采样。因此，该脉动信号摘要中给出的最小、最大和平均量全都相等。
- 如果堆没有填满到足以要求执行垃圾回收活动的程度，那么两次 heartbeat 事件之间的时间间隔可能会比指定的周期时间长得多。例如，如果您的程序只是需要每几秒执行一次垃圾回收活动，那么您可能每几秒只会看到一次脉动信号。
- 时间间隔可能会由于以下原因而比指定的周期时间长得多: 垃圾回收对于没有填满到足以有理由进行垃圾回收活动的堆无工作。例如，如果您的程序只是需要每几秒执行一次垃圾回收活动，那么您可能每几秒只会看到一次脉动信号。

  如果发生诸如同步垃圾回收或优先级更改的事件，那么该事件以及任何暂挂事件（如 heartbeat 事件）的详细信息会立即生成为输出。
- 如果给定时间段内的最大垃圾回收定量过大，那么您可能希望使用 **-Xgc:targetUtilization** 选项来降低目标利用率。该操作给予垃圾回收器更多的工作

时间。或者，您也可能希望使用 **-Xmx** 选项来增加堆大小。类似地，如果您的应用程序能够容忍比当前所报告的延迟更长的延迟，那么可以提高目标利用率或降低堆大小。

- 通过 **-Xverbosegclog:<file>** 选项可以将输出重定向到日志文件而非控制台；例如，**-Xverbosegclog:out** 将 **-verbose:gc** 输出写到文件 *out*。
- thread-priority 中所列的优先级是底层操作系统线程优先级，而非 Java 线程优先级。

同步垃圾回收

当发生同步（非确定性）垃圾回收时，也会向 **-verbose:gc** 日志中写入一个条目。该事件有三种可能的原因:

- 代码中有显式 System.gc() 调用。
- JVM 耗尽内存，然后执行同步垃圾回收以避免 OutOfMemoryError 情况。
- JVM 在持续垃圾回收期间关闭。JVM 无法取消此回收，因此同步完成此回收，然后退出。

System.gc() 条目的示例如下:

```
<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>
```

因 JVM 关闭而产生的同步垃圾回收条目的示例如下:

```
<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>
```

该示例中的 XML 标记和属性具有以下含义:

<gc-op id="9" type="syncgc" timems="6.439" ...
: 该行指示事件类型是同步垃圾回收。timems 属性是同步垃圾回收的持续时间（毫秒）。

<syncgc-info reason="..."/>
: 同步垃圾回收的原因。

<free-mem-delta.../>
: 同步垃圾回收前后的可用 Java 堆内存（字节）。

<finalization .../>
: 等待最终化的对象的数量。

<classunload-info .../>
: 脉动信号间隔期间卸载的类装入器和类的数量。

<references type="weak" cleared="53" .../>

脉动信号间隔期间已清除的 Java 引用对象的数量和类型。

因内存耗尽情况或 JVM 关闭而进行的同步垃圾回收只能在垃圾回收器处于活动状态的情况下发生。这之前必须先发生 `trigger start` 事件，尽管两者不必前后紧密相连。某些 heartbeat 事件可能在 `trigger start` 事件和 `synchgc` 事件之间发生。 `System.gc()` 所引起的同步垃圾回收可随时发生。

跟踪所有 GC 定量

对于单独 GC 定量，可通过启用 `GlobalGCStart` 和 `GlobalGCEnd` 跟踪点来进行跟踪。这些跟踪点在所有 Metronome 垃圾回收器活动（包括同步垃圾回收）的开始和结束时生成。这些跟踪点的输出类似于:

```
03:44:35.281 0x833cd00 j9mm.52 - GlobalGC start: globalcount=3

03:44:35.284 0x833cd00 j9mm.91 - GlobalGC end: workstackoverflow=0 overflowcount=0
```

优先级更改

除了摘要，还会在垃圾回收器线程优先级发生更改时（因为应用程序更改了线程优先级，或者因为应用程序中的一个或多个线程已结束）向 **-verbose:gc** 日志中写入条目。所列优先级是底层操作系统线程优先级，而非 Java 线程优先级。垃圾回收器线程优先级更改条目的示例如下:

```
<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>
```

可以通过生成与垃圾回收器线程优先级有关的跟踪点信息来实时跟踪优先级更改。该输出类似于:

```
15:58:25.493*0x8286e00    j9mm.102       - setGCThreadPriority() called with
newGCThreadPriority = 11
```

可通过使用标识来启用该输出，如下所示: **-Xtrace:iprint=tpnid{j9mm.102}**

内存耗尽条目

当内存区域之一耗尽可用空间时，会在抛出 OutOfMemoryError 异常之前向 **-verbose:gc** 日志中写入条目。该输出的示例如下:

```
<out-of-memory id="71" timestamp="2011-07-23T08:32:51.435" memorySpaceName="Scoped"
  memorySpaceAddress="080EED9C"/>
```

缺省情况下，会因 OutOfMemoryError 异常而生成 Java 转储。该转储包含与程序所用的内存区域有关的信息。结合 **-verbose:gc** 输出中给出的 `J9MemorySpace` 值，您可以使用该转储中的这些信息来确定耗尽了空间的具体内存区域:

```
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)
```

在上一个示例中，-verbose:gc 输出中给出的内存空间标识 (0x080EED9C) 可以与 Java 转储中设置了作用域的内存区域的标识相匹配。如果您拥有若干作用域并需要确定哪一个耗尽了内存，那么此匹配可能有用，因为 -verbose:gc 输出仅指示 OutOfMemoryError 是发生在永久内存、设置了作用域的内存还是堆内存中。

内存耗尽情况下的 Metronome 垃圾回收器行为:

缺省情况下，Metronome 垃圾回收器在 JVM 耗尽内存时触发不受限的非确定性垃圾回收。要避免非确定性行为，请使用 **-Xgc:noSynchronousGCOnOOM** 选项以在 JVM 耗尽内存时抛出 OutOfMemoryError。

缺省的不受限回收将一直运行，直到在一次操作中回收完所有可能的垃圾为止。所需暂停时间通常比普通 Metronome 递增定量长许多毫秒。

相关信息:

使用 -Xverbose:gc 来分析同步垃圾回收

显式 System.gc() 调用时的 Metronome 垃圾回收器行为:

如果垃圾回收周期正在进行中，那么在调用 System.gc() 时 Metronome 垃圾回收器会以同步方式完成该周期。如果未在进行任何垃圾回收，那么在调用 System.gc() 时将执行完全同步周期。使用 System.gc() 可通过受控方式来清理堆。这是非确定性操作，因为它在返回之前执行完全垃圾回收。

某些应用程序会调用具有 System.gc() 调用的供应商软件，其中不允许创建这些非确定性延迟。要禁用所有 System.gc() 调用，请使用 **-Xdisableexplicitgc** 选项。

System.gc() 调用的详细垃圾回收输出具有“系统垃圾回收”原因，而可能具有较长的持续时间:

```
<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>
```

共享类诊断数据

了解如何诊断可能出现的问题，将有助于使用共享类方式。

有关共享类的介绍，请参阅JVM 间类数据共享。

《IBM SDK for Java V7 用户指南》包含有关诊断共享类问题的有用指导，包括:

- 部署共享类
- 处理运行时字节码修改
- 了解动态更新
- 使用 Java Helper API

- 了解共享类诊断输出
- 调试有关共享类的问题

您可以在此处查找信息: IBM SDK for Java 7 - 共享类诊断数据。

《IBM SDK for Java V7 用户指南》中的部分资料可能不适用于 IBM WebSphere Real Time for RT Linux。特别是:

- 在实时方式中，应用程序只有对共享类高速缓存的只读访问权，而不是读-写访问权。
- 只能使用 **admincache** 工具来对高速缓存进行修改。
- 实时方式下，非持久性高速缓存不可用。

使用 JVMTI

JVMTI 是一个双路接口，支持 JVM 与本机代理程序之间进行通信。它取代了 JVMDI 和 JVMPI 接口。

JVMTI 支持第三方为 JVM 开发调试、概要分析和监视工具。此接口包含用于通知 JVM 所需的各类信息的代理程序机制。此接口还提供了接收相关通知的方式。任何时候都可以将几个代理程序连接至 JVM。

《IBM SDK for Java V7 用户指南》包含有关使用 JVMTI 的详细信息，包括有关 IBM 对 JVMTI 的扩展的 API 参考部分。

您可以在此处查找信息: IBM SDK for Java 7 - 使用 JVMTI。

使用 Diagnostic Tool Framework for Java

Diagnostic Tool Framework for Java (DTFJ) 是 IBM 提供的 Java 应用程序编程接口 (API)，用于为构建 Java 诊断工具提供支持。DTFJ 可以处理来自系统转储或 Javadump 的数据。

《IBM SDK for Java V7 用户指南》包含有关 DTFJ 的详细信息。跟随此链接: 使用 Diagnostic Tool Framework for Java

第 10 章 参考

这一组主题列出了可用于 WebSphere Real Time for RT Linux 的选项和类库

命令行选项

在您启动 Java 时，可以在命令行中指定选项。最常用的选项已选作缺省选项。

指定 Java 选项和系统属性

有三种方法可指定 Java 属性和系统属性。

关于此任务

您可使用这些方法指定 Java 选项和系统属性。按照优先顺序，这些方法依次为:

1. 在命令行上指定选项或属性。例如:

   ```
   java -Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump MyJavaClass
   ```

2. 创建包含这些选项的文件，并使用 **-Xoptionsfile=**<*filename*> 选项在命令行上指定该文件。

   在该选项文件中，在新的一行中指定每个选项；如果需要单个选项跨多个行，那么可以使用“\”字符作为续行符。使用“#”字符来定义注释行。不能在选项文件中指定 **-classpath**。以下是选项文件的示例:

   ```
   #My options file
   -X<option1>
   -X<option2>=\
   <value1>,\
   <value2>
   -D<sysprop1>=<value1>
   ```

3. 创建包含这些选项的名为 **IBM_JAVA_OPTIONS** 的环境变量。例如:

   ```
   export IBM_JAVA_OPTIONS="-Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump"
   ```

您在命令行中指定的最后一个选项的优先顺序高于第一个选项。例如，如果您指定选项 **-Xint -Xjit myClass**，那么 **-Xjit** 的优先顺序高于 **-Xint**。

系统属性

为应用程序提供了系统属性，帮助提供有关运行时环境的信息。

com.ibm.jvm.realtime

该属性使 Java 应用程序能够确定是否在 WebSphere Real Time for RT Linux 环境下运行。

如果您的应用程序正在 IBM WebSphere Real Time for RT Linux 运行时环境下运行，且使用 **-Xrealtime** 选项启动，那么 **com.ibm.jvm.realtime** 属性的值为“hard”。

如果您的应用程序正在 IBM WebSphere Real Time for RT Linux 运行时环境下运行，但未使用 **-Xrealtime** 选项启动，那么不会设置 **com.ibm.jvm.realtime** 属性。

如果应用程序在 IBM WebSphere Real Time 运行时环境下运行，那么 **com.ibm.jvm.realtime** 属性的值为“soft”。

标准选项

标准选项的定义。

-agentlib:<*libname*>**[=**<*options*>**]**
装入本机代理程序库 <*libname*>；例如 **-agentlib:hprof**。有关更多信息，请在命令行上指定 **-agentlib:jdwp=help** 和 **-agentlib:hprof=help**。

-agentpath:*libname***[=**<*options*>**]**
以完整路径名装入本机代理程序库。

-assert
打印有关与断言相关的选项的帮助。

-cp 或 -classpath <*由 : 隔开的目录和 .zip 或 .jar 文件*>
设置应用程序类和资源的搜索路径。如果不使用 **-classpath** 和 **-cp**，也不设置 **CLASSPATH**，那么缺省情况下，用户的类路径是当前目录（.）。

-D<property_name>=<*value*>
设置系统属性。

-help 或 -?
打印使用消息。

-javaagent:<*jarpath*>**[=**<*options*>**]**
装入 Java 编程语言代理程序。有关更多信息，请参阅 java.lang.instrument API 文档。

-jre-restrict-search
在版本搜索中包含用户专用的 JRE。

-no-jre-restrict-search
在版本搜索中排除用户专用的 JRE。

-showversion
打印产品版本并继续。

-verbose:[*class***,***gc***,***dynload***,***sizes***,***stack***,***jni***]**
启用冗余输出。

-verbose:class
为装入的每个类的 stderr 写入一个条目。

-verbose:gc
请参阅第 124 页的『使用 verbose:gc 信息』。

-verbose:dynload
在 JVM 装入每个类时提供详细信息，包括:
- 类名和程序包
- 对于 .jar 文件中的类文件，.jar 的名称和目录路径
- 类大小和装入类所用时间的详细信息

数据将写入 stderr。输出示例如下所示:

```
<Loaded java/lang/String from /myjdk/sdk/jre/lib/i386/
softrealtime/jclSC160/vm.jar>
<Class size 17258; ROM size 21080; debug size 0>
<Read time 27368 usec; Load time 782 usec; Translate time 927 usec>
```

注: **-verbose:dynload** 输出中不列出从共享类高速缓存装入的类。 使用 **-verbose:class** 获取有关这些类的信息。

-verbose:sizes
向 stderr 写入信息，描述用于 JVM 中堆栈和堆的内存量

-verbose:stack
向 stderr 写入信息，描述 Java 和 C 堆栈使用情况。

-verbose:jni
向 stderr 写入信息，描述由应用程序和 JVM 调用的 JNI 服务。

-version
打印输出非实时方式的版本信息。 与 -Xrealtime 选项结合使用时，会打印输出实时方式的版本信息。

-version:<*value*>
需要运行指定版本。

-X 打印关于非标准选项的帮助。

非标准选项

带有 **-X** 前缀的选项是非标准选项；它们可能发生更改，而不另行通知。

《IBM SDK for Java V7 用户指南》包含有关非标准选项的详细信息。您可在以下地址查找该信息: IBM SDK for Java 7 - 命令行选项。

以下列表记录了《IBM SDK for Java V7 用户指南》中所做的更改:

- **-XX:-LazySymbolResolution** 是 RT Linux 操作系统上的缺省选项，**-XX:+LazySymbolResolution** 不是缺省选项。

以下部分提供有关 IBM WebSphere Real Time for RT Linux 的补充信息。

实时选项

WebSphere Real Time for RT Linux 中使用的 **-Xrealtime** 选项的定义。

以下 **-X** 选项适用于 WebSphere Real Time for RT Linux 环境。

-Xrealtime
启动实时方式。如果希望运行 Metronome 垃圾回收器，并使用 Real-Time Specification for Java (RTSJ) 服务，那么这是必需选项。如果未指定该选项，那么 JVM 以非实时方式启动等同于 IBM SDK and Runtime Environment for Linux Platforms, Java 2 Technology V7。

-Xrealtime 选项可与 **-Xgcpolicy:metronome** 互换。您可以指定其中一个来获取实时方式。

提前选项

提前选项的定义。

用途

未指定选项:

使用解释器和动态编译的代码运行。如果发现了 AOT 代码，并不使用它。而是根据需要进行动态编译。这对非实时和某些实时应用程序而言十分有用。该选项提供优化的性能和吞吐量，但是在运行时会因执行编译而出现非确定性延迟。

-Xjit: 此选项等同于缺省值。

-Xint:

仅运行解释器，忽略为 AOT 编写的代码（可能位于预编译 JAR 文件中），同时不会运行动态编译器。除非需要对您怀疑与编译有关的问题进行调试，或用于不会受益于编译的超短批处理应用程序，否则一般不需要使用该方式。

-Xnojit:

运行解释器，并使用为 AOT 编写的代码（如果其位于预编译的 JAR 文件中）。该选项不会运行动态编译器。该方式特别适用于这样的实时应用程序: 希望确保在运行时不会因编译造成非确定性延迟。为 AOT 编写的代码只能在使用 **-Xrealtime** 选项运行的情况下使用。以标准 JVM 方式运行（即未指定 **-Xrealtime**）的环境下不支持该选项。

示例 `java -Xrealtime -Xnojit outputtest.jar`。

Metronome 垃圾回收器选项

Metronome 垃圾回收器选项的定义。

-Xgc:immortalMemorySize=*size*

指定永久堆区域的大小。缺省值为 16 MB。

-Xgc:scopedMemoryMaximumSize=*size*

指定作用域内存堆区域的大小。缺省值为 8 MB。

-Xgc:synchronousGCOnOOM | -Xgc:nosynchronousGCOnOOM

执行垃圾回收的一种情况是堆内存耗尽。如果堆中没有更多的可用空间，使用 **-Xgc:synchronousGCOnOOM** 可停止应用程序，同时垃圾回收除去未使用的对象。如果可用空间再次耗尽，考虑降低目标利用率以为垃圾回收的完成提供更多时间。设置 **-Xgc:nosynchronousGCOnOOM** 意为当堆内存已满时，应用程序停止，并发出内存耗尽消息。缺省值为 **-Xgc:synchronousGCOnOOM**。

-Xnoclassgc

禁用类垃圾回收。此选项关闭与 JVM 不再使用的 Java 类关联的存储器的垃圾回收。缺省行为是 **-Xnoclassgc**。

-Xgc:targetUtilization=*N*

将应用程序利用率设置为 N%；垃圾回收器尝试最多使用每个时间间隔的 (100-N)%。合理值范围为 50-80%。低分配率的应用程序可能以 90% 运行。缺省值为 70%。

本示例显示堆内存最大大小为 30 MB。垃圾回收器尝试使用每个时间间隔的 25%，原因是应用程序的目标利用率为 75%。

```
java -Xrealtime -Xmx30m -Xgc:targetUtilization=75 Test
```

-Xgc:threads=*N*

指定要运行的 GC 线程数目。缺省值为 1。

-Xgc:verboseGCCycleTime=N

N 为应转储摘要信息的时间（单位为毫秒）。

注: 周期时间不表示刚好在这一时间转储摘要信息，而是表示在符合该时间标准的最后一次垃圾回收事件过去时进行转储。

-Xmx<*size*>

指定 Java 堆大小。与其他垃圾回收策略不同，实时 Metronome GC 并不支持堆扩展。没有初始或最大堆大小选项。您只能指定最大堆大小。

-Xthr:metronomeAlarm=os*xx*

控制运行 Metronome 垃圾回收器 警报线程使用的优先级。

其中 *xx* 为从 11 到 89 的数字，用以指定运行 Metronome 警报线程应使用的优先级。修改运行警报线程使用的 OS 优先级时应格外小心。如果您指定的 OS 优先级低于任意实时线程，那么将出现 OutOfMemory 错误，原因是垃圾回收器结束时运行的优先级低于分配垃圾的实时线程优先级。缺省 Metronome 垃圾回收器 警报线程运行优先级为 OS 优先级 89。

JVM 的缺省设置

当 JVM 的运行环境未发生更改时，缺省设置会应用于实时 JVM 中。下面显示了常用设置以供参考。

在 JVM 启动时可以使用环境变量或命令行参数来更改缺省设置。下表显示了部分常用 JVM 设置。最后一列指示您可以如何更改行为，其中的如下关键字适用:

- **e** - 仅环境变量控制的设置
- **c** - 仅命令行参数控制的设置
- **ec** - 环境变量和命令行参数都可以控制的设置，其中命令行参数优先。

提供的信息仅供快速参考，并不全面。

JVM 设置	缺省值	影响设置的原因
Javadumps	已启用	ec
Javadumps on out of memory	已启用	ec
Heapdumps	已禁用	ec
Heapdumps on out of memory	已启用	ec
Sysdumps	已启用	ec
Where dump files are produced	当前目录	ec
Verbose output	已禁用	c
Boot classpath search	已禁用	c
JNI checks	已禁用	c
Remote debugging	已禁用	c
Strict conformancy checks	已禁用	c
Quickstart	已禁用	c
Remote debug info server	已禁用	c
Reduced signalling	已禁用	c
Signal handler chaining	已启用	c

JVM 设置	缺省值	影响设置的原因
Classpath	未设置	ec
Class data sharing	已禁用	c
Accessibility support	已启用	e
JIT compiler	已启用	ec
AOT compiler (AOT is not used by the JVM unless shared classes are also enabled)	已启用	c
JIT debug options	已禁用	c
Java2D max size of fonts with algorithmic bold	14 磅	e
Java2D use rendered bitmaps in scalable fonts	已启用	e
Java2D freetype font rasterizing	已启用	e
Java2D use AWT fonts	已禁用	e
Default locale	无	e
Time to wait before starting plug-in	0	e
Temporary directory	/tmp	e
Plug-in redirection	无	e
IM switching	已禁用	e
IM modifiers	已禁用	e
Thread model	不适用	e
Initial stack size for Java Threads 32-bit. Use: **-Xiss<size>**	2 KB	c
Maximum stack size for Java Threads 32-bit. Use: **-Xss<size>**	256 KB	c
Stack size for OS Threads 32-bit. Use **-Xmso<size>**	256 KB	c
Initial heap size. Use **-Xms<size>**	64 MB	c
Maximum Java heap size. Use **-Xmx<size>**	可用内存的一半，最小为 16 MB，最大为 512 MB	c
应用程序的目标时间间隔利用率。垃圾回收器会尝试使用剩余部分。使用 **-Xgc:targetUtilization=<percentage>**	70%	c
要运行的垃圾回收器线程的数量。使用 **-Xgc:threads=<value>**	1	c
在 **-Xrealtime** 方式下可以分配到作用域内存的最大内容量。使用 **-Xgc:scopedMemoryMaximumSize=<size>**。	8 MB	c
在 **-Xrealtime** 方式下设置永久内存区域的大小。使用 **-Xgc:immortalMemorySize=<size>**	16 MB	c

注: “可用内存”可以是实际（物理）内存的数量，或者 **RLIMIT_AS** 值，取其中最小的值。

WebSphere Real Time for RT Linux 类库

对由 WebSphere Real Time for RT Linux 使用的 Java 类库的引用。

供 WebSphere Real Time for RT Linux 使用的 Java 类库信息可访问: http://www.rtsj.org/specjavadoc/book_index.html。

通过 TCK 运行

如果通过 Real-Time Specification for Java (RTSJ) Technology Compatibility Kit (TCK) 运行 WebSphere Real Time for RT Linux，那么类路径中应包含 `demo/realtime/TCKibm.jar`，以便成功完成测试。

`TCKibm.jar` 包含类 **`VibmcorProcessorLock`**，该类是 IBM 对 TCK.ProcessorLock 类的扩展。此类提供一小组 TCK 测试中所需的单处理器行为。有关 TCK.ProcessorLock 类及其特定于供应商的扩展的更多信息，请参阅随 TCK 分发版提供的自述文件。

声明

本信息是为在美国提供的产品和服务编写的。IBM 可能在其他国家或地区不提供本文档中讨论的产品、服务或功能特性。有关您当前所在区域的产品和服务的信息，请向您当地的 IBM 代理咨询。任何对 IBM 产品、程序或服务的引用并非意在明示或暗示只能使用 IBM 的产品、程序或服务。只要不侵犯 IBM 的知识产 权，任何同等功能的产品、程序或服务，都可以代替 IBM 产品、程序或服务。但是，评估和验证任何非 IBM 产品、程序或服务，则由用户自行负责。

IBM 公司可能已拥有或正在申请与本文档中主题有关的各项专利。提供本文档并未授予用户使用这些专利的任何许可。您可以用书面方式将许可查询寄往:

IBM Director of Licensing IBM Corporation North Castle DriveArmonk, NY 10504-1785 U.S.A.

有关双字节 (DBCS) 信息的许可证查询，请与您所在国家或地区的 IBM 知识产权部门联系，或用书面方式将查询寄往:

Intellectual Property LicensingLegal and Intellectual Property LawIBM Japan Ltd.19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

本条款不适用英国或任何这样的条款与当地法律不一致的国家或地区:

INTERNATIONAL BUSINESS MACHINES CORPORATION“按现状”提供本出版物，不附有任何种类的（无论是明示的还是暗含的）保证，包括但不限于暗含的有关非侵权、适销和适用于某特定用途的保证。某些国家或地区在某些交易中不允许免除明示或暗含的保证。因此本条款可能不适用于您。

本信息中可能包含技术方面不够准确的地方或印刷错误。此处的信息将定期更改；这些更改将编入本资料的新版本中。IBM 可以随时对本信息中描述的产品和/或程序进行改进和/或更改，而不另行通知。

本信息中对非 IBM Web 站点的任何引用都只是为了方便起见才提供的，不以任何方式充当对那些 Web 站点的保证。那些 Web 站点中的资料不是本 IBM 产品资料的一部分，使用那些 Web 站点带来的风险将由您自行承担。

IBM 可以按它认为适当的任何方式使用或分发您所提供的任何信息而无须对您承担任何责任。

本程序的被许可方如果要了解有关程序的信息以达到如下目的: (i) 允许在独立创建的程序和其他程序（包括本程序）之间进行信息交换，以及 (ii) 允许对已经交换的信息进行相互使用，请与下列地址联系:

- JIMMAIL@uk.ibm.com [Hursley Java Technology Center (JTC) contact]

只要遵守适当的条件和条款，包括某些情形下的一定数量的付费，都可获得这方面的信息。

本资料中描述的许可程序及其所有可用的许可资料均由 IBM 依据 IBM 客户协议、IBM 国际程序许可协议或任何同等协议中的条款提供。

此处包含的任何性能数据都是在受控环境中测得的。因此，在其他操作环境中获得的数据可能会有明显的不同。有些测量可能是在开发集的系统上进行的，因此不保证与一般可用系统上进行的测量结果相同。此外，有些测量是通过推算而估计的，实际结果可能会有差异。本文档的用户应当验证其特定环境的适用数据。

涉及非 IBM 产品的信息可从这些产品的供应商、其出版说明或其他可公开获得的资料中获取。IBM 没有对这些产品进行测试，也无法确认其性能的精确性、兼容性或任何其他关于非 IBM 产品的声明。有关非 IBM 产品性能的问题应当向这些产品的供应商提出。

隐私策略注意事项

IBM 软件产品（包括软件即服务解决方案（“软件产品”））可能会使用 cookie 或其他技术收集产品使用信息，以帮助改善最终用户体验，定制与最终用户的交互或用于其他目的。在许多情况下，软件产品不会收集个人可标识信息 (PII)。我们的某些软件产品可以帮助您收集个人可标识信息 (PII)。如果本软件产品使用 cookie 收集个人可标识信息 (PII)，下面列出了有关本产品的具体 cookie 使用信息。

此软件产品不使用 cookie 或其他技术来收集个人可标识信息。

如果针对该软件产品部署的配置使您作为客户能够通过 cookie 和其他技术从最终用户处收集个人可标识信息，那么您应向自己的法律顾问了解有关适用于此类数据收集的任何法律，包括声明和同意的要求。

要获取有关针对这些目的使用各种技术（包括 cookies）的更多信息，请参阅 (i) IBM 的隐私策略：http://www.ibm.com/privacy；(ii) IBM 的在线隐私声明：http://www.ibm.com/privacy/details（尤其是题为“Cookies, Web Beacons and Other Technologies”的部分）；以及 (iii)“IBM Software Products and Software-as-a-Service Privacy Statement”：http://www.ibm.com/software/info/product-privacy。

商标

IBM、IBM 徽标和 ibm.com 是 International Business Machines Corporation 在美国和/或其他国家或地区的商标或注册商标。如果这些术语和其他 IBM 已注册商标的术语在本信息中首次出现时都使用商标符号（® 或 ™）标记，那么这些符号表示在本信息发布时由 IBM 在美国注册或拥有的普通法商标。这些商标也可能是在其他国家或地区的注册商标或普通法商标。 IBM 商标的最新列表在 Web 页面 http://www.ibm.com/legal/copytrade.shtml 的“Copyright and trademark information”部分中提供。

Adobe、Adobe 徽标、PostScript 和 PostScript 徽标是 Adobe Systems Incorporated 在美国和/或其他国家或地区的注册商标或商标。

Intel 和 Itanium 是 Intel Corporation 或其子公司在美国和其他国家或地区的商标。

Linux 是 Linus Torvalds 在美国和/或其他国家或地区的商标。

Java 和所有基于 Java 的商标和徽标是 Oracle 和/或其子公司的商标或注册商标。

其他公司、产品或服务名称可能是其他公司的商标或服务标记。

索引

O

P

R

S

T

[特别字符]

Printed in China