IBM WebSphere Real Time for RT Linux
버전 3

사용자 안내서

IBM

IBM WebSphere Real Time for RT Linux
버전 3

사용자 안내서

IBM

참고

이 정보 및 이 정보가 지원하는 제품을 사용하기 전에 165 페이지의 제 11 장 『주의사항』의 정보를 읽으십시오.

초판(2011년 8월)

이 사용자 안내서 개정판은 새 개정판에 별도로 명시하지 않는 한 IBM WebSphere Real Time for RT Linux, 버전 3 및 모든 후속 릴리스와 수정사항에 적용됩니다.

목차

그림

표

서문

이 사용자 안내서는 IBM® WebSphere® Real Time for RT Linux에 대한 일반적인 정보를 제공합니다.

제 1 장 소개

이 정보는 IBM WebSphere Real Time for RT Linux에 대해 설명합니다.

- 『WebSphere Real Time for RT Linux 개요』
- 3 페이지의 『새로운 기능』
- 3 페이지의 『장점』

WebSphere Real Time for RT Linux 개요

WebSphere Real Time for RT Linux는 IBM J9 가상 머신(JVM)과 실시간 기능을 번들로 제공합니다.

WebSphere Real Time for RT Linux는 실시간 기능이 있는 IBM SDK for Java를 확장하는 SDK(Software Development Kit)가 있는 Java 런타임 환경입니다. 정확한 응답 시간에 따르는 애플리케이션은 표준 Java 기술의 WebSphere Real Time for RT Linux와 함께 제공되는 실시간 기능을 이용할 수 있습니다.

기능

실시간 애플리케이션은 절대적 속도보다 일관된 런타임을 필요로 합니다.

JVM이 실시간 모드에서 실행되는 경우 가비지 콜렉션된 힙 외에 추가 메모리 영역을 사용할 수 있습니다. 프로그램이 재사용 가능한 범위 및 재사용 불가능한 영구 메모리 영역을 요청하거나 지정할 수 있으며 이는 가비지 콜렉션되지 않습니다. 이 기능은 애플리케이션에 메모리 사용에 대한 더욱 강력한 제어를 제공합니다. 또한 메트로놈 가비지 콜렉터를 사용하여 시간 기반 콜렉션을 수행합니다. JVM이 기존의 처리량 모드에서 실행되는 경우 처리량을 최적화하는 다양한 작업 기반 가비지 콜렉터를 사용할 수 있지만 메트로놈 가비지 콜렉터보다 지연 시간이 길어질 수 있습니다.

기존의 JVM으로 실시간 애플리케이션을 배치할 때 주요 관심 사항은 다음과 같습니다.

- 가비지 콜렉션(GC) 활동에 따른 예측 불가능한 지연(장기화될 수 있음).
- JIT(Just-In-Time) 컴파일 및 재컴파일이 발생할 때 메소드 런타임이 지연되고 실행 시간이 변경될 수 있음.
- 임의의 운영 체제 스케줄링.

WebSphere Real Time for RT Linux는 다음을 제공하여 이 문제를 해결합니다.

- 일시정지 시간이 매우 적은 증분식 결정적 가비지 콜렉터인 메트로놈 가비지 콜렉터
- AOT(Ahead-of-Time) 컴파일.
- 우선순위 기반 FIFO 스케줄링.

또한, WebSphere Real Time은 실시간 프로그래머에게 RTSJ 기능을 제공합니다. 9 페이지의 『RTSJ 지원』의 내용을 참조하십시오.

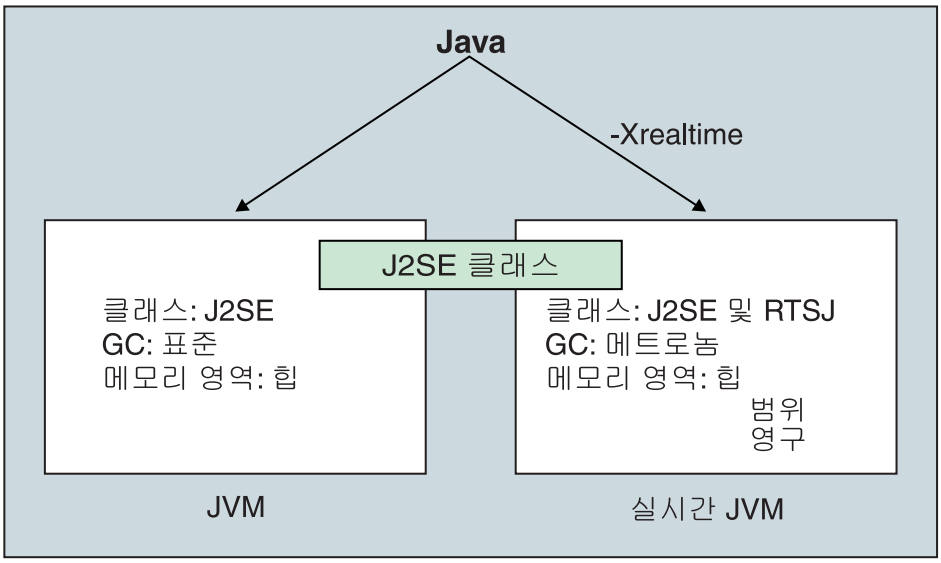

그림 1. WebSphere Real Time for RT Linux 개요

실시간 기능을 사용하려면 JVM 또는 제공된 도구를 실행할 때 -Xrealtime 옵션을 사용합니다. 기본적으로 JVM 및 제공된 도구는 실시간 기능을 사용하지 않은 채 실행됩니다. 그림 1은 WebSphere Real Time for RT Linux에서 제공되는 두 가지 JVM의 관계를 보여줍니다.

다음 Java 명령은 -Xrealtime 옵션을 인식합니다.

표 1. 실시간 모드에 사용되는 Java 명령

명령	기능
java	기본적으로 표준 모드에서 실행되지만 -Xrealtime 옵션을 지정하면 실시간 모드에서 실행됩니다. 실시간 모드에서는 프로그래머가 javax.realtime 패키지의 클래스에 액세스합니다. 사전 컴파일된 jar 파일 및 메트로놈 결정적 가비지 콜렉션 기술을 사용할 수 있습니다.
javac, javah, javap	기본적으로 표준 모드에서 실행되지만 -Xrealtime 옵션을 지정하면 클래스 경로에 javax.realtime.* 클래스가 포함됩니다.
admincache	-Xrealtime의 유무에 상관없이 실행할 수 있지만 **admincache** 도구를 사용하여 공유 캐시를 채우는 것은 실시간 모드에서만 가능합니다. 일반 모드에서는 캐시 유틸리티만 사용할 수 있습니다(예: listAllCaches 또는 printStatsIn). **jdmpview**와 달리 **admincache**는 실시간 JVM의 캐시에 액세스하려면 -Xrealtime과 함께 실행하고, 일반 JVM의 캐시에 액세스하려면 -Xrealtime 없이 실행해야 합니다.
jextract	jextract는 기본적으로 표준 모드에서 실행되지만 실시간 모드에서 JVM이 생성한 시스템 덤프를 처리할 때는 -Xrealtime 옵션과 함께 실행해야 합니다.

새로운 기능

이 주제에서는 IBM WebSphere Real Time for RT Linux의 변경사항을 소개합니다.

WebSphere Real Time for RT Linux V3

WebSphere Real Time for RT Linux V3은 실시간 기능을 포함하도록 이 릴리스에서 사용 가능한 기능에 빌드하는 IBM SDK for Java 7에 대한 확장입니다. 이전 버전의 WebSphere Real Time for RT Linux는 IBM SDK for Java의 이전 릴리스에 기반합니다.

새로운 기능에 대해 더 자세히 알아보려면 IBM SDK for Java 7 Information Center의 새로운 기능을 참조하십시오.

jxeinajar

WebSphere Real Time for RT Linux V3이 더 이상 jxeinajar 사용을 지원하지 않습니다. 참조용으로 이전의 jxeinajar 관련 정보 및 admincache로 마이그레이션하는 방법에 대한 정보를 WebSphere Real Time for RT Linux V2 문서에서 찾을 수 있습니다.

장점

실시간 환경의 이점은 표준 JVM보다 높은 수준의 예측 가능성으로 Java 애플리케이션을 실행하고 사용자의 Java 애플리케이션에 일관된 시간의 동작을 제공하는 것입니다. 컴파일, 가비지 콜렉션과 같은 백그라운드 활동이 지정된 시간에 발생하므로 애플리케이션 실행 시 백그라운드 활동이 예기치 않게 많아지는 것을 방지합니다.

JVM을 다음 기능으로 확장하면 이러한 이점을 얻을 수 있습니다.

- 메트로놈 실시간 가비지 콜렉션 기술
- AOT(Ahead-of-Time) 컴파일
- Real-Time Specification for Java(RTSJ) 지원

모든 Java 애플리케이션은 수정사항 없이 실시간 환경에서 실행할 수 있으며 메트로놈 가비지 콜렉터 및 정기적인 간격으로 발생하는 결정적 가비지 콜렉션의 이점을 활용합니다. WebSphere Real Time for RT Linux에서 최대 이점을 얻으려면 실시간 스레드 및 힙이 없는 실시간 스레드를 사용하여 실시간 환경에 맞는 애플리케이션을 작성합니다. 사용하는 방법은 애플리케이션의 시간 스펙에 따라 다릅니다.

많은 실시간 Java 애플리케이션이 메트로놈 가비지 콜렉터 및 AOT의 짧은 일시정지 시간을 이용하여 목표를 달성하고, Java 이식성의 이점을 활용합니다. 요구사항이 엄격한 애플리케이션은 범위 및 영구 메모리와 함께 실시간 스레드 및 힙이 없는 실시간 스

레드의 RTSJ 기능을 사용해야 합니다. 이 방법은 애플리케이션을 실시간 환경에서만 실행하도록 제한하므로 JSE Java에 대한 이식성 이점을 잃을 수 있습니다. 또한 복합 프로그래밍 모델을 개발해야 합니다.

제 2 장 IBM WebSphere Real Time for RT Linux 이해

이 섹션에서는 IBM WebSphere Real Time for RT Linux의 주요 컴포넌트를 소개합니다.

- 『메트로놈 가비지 콜렉터 소개』
- 7 페이지의 『컴파일러』
  - 8 페이지의 『JIT 및 AOT 컴파일 비교』
- 9 페이지의 『RTSJ 지원』
  - 9 페이지의 『실시간 스레드 스케줄링 및 디스패치』
  - 14 페이지의 『메모리 관리』

메트로놈 가비지 콜렉터 소개

WebSphere Real Time for RT Linux에서 표준 가비지 콜렉터 대신 메트로놈 가비지 콜렉터를 사용합니다.

메트로놈 가비지 콜렉션과 표준 가비지 콜렉션의 주요 차이점은 메트로놈 가비지 콜렉션은 인터럽트 가능한 작은 단계에서 발생하고 표준 가비지 콜렉션은 가비지를 표시하고 수집하는 동안 애플리케이션을 중지시킨다는 것입니다.

예를 들면 다음과 같습니다.

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

예제에서 애플리케이션이 60밀리초마다 80% 실행되도록 지정합니다. 수집할 가비지가 있는 경우 나머지 20% 시간은 가비지 콜렉션에 사용됩니다. 메트로놈 가비지 콜렉터는 충분한 자원이 제공될 경우에 한해 이용 수준을 보장합니다. 힙의 여유 공간 *크기*가 동적으로 판별되는 임계값보다 작아지면 가비지 콜렉션이 시작됩니다.

가비지 콜렉션 및 우선순위

가비지 콜렉션 스레드는 힙에 가비지를 생성하는 가장 우선순위가 높은 스레드보다 높은 우선순위로 실행해야 합니다. 그렇지 않으면 구성된 이용률이 지정하는 대로 실행되지 않을 수 있습니다. 일반 Java 스레드 및 실시간 스레드 모두 가비지를 생성하므로 모든 일반 및 실시간 스레드보다 높은 우선순위로 가비지 콜렉션을 실행해야 합니다. 이 우선순위는 JVM에서 자동으로 처리되어 모든 일반 및 실시간 스레드의 가장 높은 우선순위보다 높은 0.5 우선순위로 가비지 콜렉션이 실행됩니다. 그러나, 힙이 없는 실시간 스레드(NHRT)는 가비지 콜렉션의 영향을 받지 않도록 해야 합니다. 모든 NHRT를 가장 높은 우선순위 실시간 스레드보다 높은 우선순위로 실행하십시오. 이는 NHRT가 가비지 콜렉션보다 높은 우선순위로 실행되며 지연되지 않음을 의미합니다.

표 2은 정의할 수 있는 우선순위의 일반적인 예제와 사용자 선택에 따른 관련 가비지 콜렉션 우선순위를 보여줍니다.

Java 우선순위 및 OS 우선순위 비교를 보려면 12 페이지의 『우선순위 맵핑 및 상속』의 내용을 참조하십시오.

표 2. 가비지 콜렉션 및 우선순위 예제

스레드	우선순위(예제)
우선순위가 가장 높은 실시간 스레드:	20 (OS 우선순위 43)
가비지 콜렉터:	20.5 (OS 우선순위 44)
NHRT가 가비지 콜렉터와 상관 없이 실행되게 하려면 GC보다 높은 우선순위 설정	21 (OS 우선순위 45) 또는 그 이상
메트로놈 알람 스레드:	우선순위 46 (OS 우선순위 89)

주: 이 구성을 사용하더라도 메트로놈 알람 스레드가 정기적으로 활성화되고 가비지 콜렉션이 작업을 수행해야 할 때 제대로 작동하도록 가장 높은 우선순위로 실행되기 때문에 힙이 없는 실시간 스레드는 가비지 콜렉션의 영향을 전혀 받지 않는 것은 아닙니다. 물론 수행할 작업이 매우 작으므로 주요 고려사항이 아닙니다.

메트로놈 가비지 콜렉션 및 클래스 로드 해제

메트로놈 가비지 콜렉터는 일시정지 시간 이탈을 유발하는 비결정적 작업량을 필요로 하기 때문에 IBM WebSphere Real Time에서 클래스를 로드 해제하지 않습니다.

메트로놈 가비지 콜렉터 스레드

메트로놈 가비지 콜렉터는 단일 알람 스레드와 많은 콜렉션(GC) 스레드라는 두 가지 유형의 스레드로 구성됩니다. 기본적으로, 한 개의 GC 스레드가 있습니다. **-Xgcthreads** 옵션을 사용하여 JVM의 GC 스레드 수를 설정할 수 있습니다.

JVM에 대한 알람 스레드 수는 변경할 수 없습니다.

메트로놈 가비지 콜렉터는 힙 메모리에 충분한 여유 공간이 있는지 확인하기 위해 JVM을 정기적으로 검사합니다. 여유 공간의 *크기*가 한계보다 적으면 메트로놈 가비지 콜렉터가 JVM을 트리거하여 가비지 콜렉션을 시작합니다.

알람 스레드

단일 알람 스레드는 최소 자원 사용을 보장합니다. 정기적인 간격으로 『활성화』되고 다음을 검사합니다.

- 힙 메모리의 여유 공간 *크기*
- 가비지 콜렉션이 현재 진행 중인지 여부

여유 공간이 충분하지 않고 가비지 콜렉션이 진행 중이지 않으면 알람 스레드가 가비지 콜렉션을 시작하도록 콜렉션 스레드를 트리거합니다. JVM을 검사하는 다음 스케줄 시간까지 알람 스레드는 아무런 작업을 수행하지 않습니다.

콜렉션 스레드

각 콜렉션 스레드는 Java 및 실시간 스레드에서 힙 오브젝트를 검사합니다. 메모리 영역을 검사할 때는 다음 순서를 따릅니다.

1. 힙에서 범위 메모리의 오브젝트가 사용하는 라이브 오브젝트를 식별하고 표시하기 위해 범위 메모리
2. 힙에서 영구 메모리의 오브젝트가 사용하는 라이브 오브젝트를 식별하고 표시하기 위해 영구 메모리
3. 라이브 오브젝트를 식별하고 표시하기 위해 힙 메모리

라이브 오브젝트가 표시된 경우 표시되지 않은 오브젝트가 콜렉션 대상이 됩니다.

가비지 콜렉션 주기가 완료된 후 메트로놈 가비지 콜렉터가 여유 힙 공간의 *크기*를 확인합니다. 여유 힙 공간이 여전히 충분하지 않으면 동일한 트리거 ID를 사용하여 다른 가비지 콜렉션 주기를 시작합니다. 여유 힙 공간이 충분하면 트리거가 종료하고 가비지 콜렉션 스레드가 중지됩니다. 알람 스레드가 여유 힙 공간을 계속 모니터링하고 필요하면 다른 가비지 콜렉션 주기를 트리거합니다.

메트로놈 가비지 콜렉터를 사용하는데 관한 자세한 정보는 72 페이지의 『메트로놈 가비지 콜렉터 사용』의 내용을 참조하십시오.

컴파일러

IBM WebSphere Real Time for RT Linux는 다양한 레벨의 코드 성능 및 판별을 제공하는 일부 코드 컴파일 모델을 지원합니다.

IBM WebSphere Real Time for RT Linux와 함께 Java 코드 컴파일에 사용 가능한 옵션은 다음 사항이 포함됩니다.

우선순위가 낮은 JIT(Just-In-Time) 컴파일

WebSphere Real Time for RT Linux의 기본 컴파일 모델은 JIT(Just-In-Time) 컴파일러를 사용하여 애플리케이션을 실행하는 동안 Java 애플리케이션의 중요한 메소드를 컴파일합니다. 이 모드에서는 JIT 컴파일러가 실시간 이외 JVM의 JIT 컴파일러 조작과 비슷한 방식으로 작동합니다. 차이점은 WebSphere Real Time for RT Linux JIT 컴파일러가 실시간 스레드보다 낮은 우선순위로 실행된다는 점입니다. 우선순위가 낮으면 애플리케이션이 실시간 태스크를 수행할 필요가 없을 때 JIT 컴파일러가 시스템 자원을 사용하게 됩니다. 따라서 JIT 컴파일러는 실시간 태스크의 성능에 커다란 영향을 미치지 않습니다.

AOT(Ahead-of-time) 사전 컴파일된 코드

WebSphere Real Time for RT Linux는 애플리케이션 실행 전에 사전 컴파

일 단계에서 원시 코드에 대해 Java 메소드를 컴파일합니다. AOT 사전 로드된 코드를 사용하면 성능이 향상되고 판별력이 극대화됩니다.

AOT 사전 컴파일된 코드와 우선순위가 낮은 JIT 컴파일을 결합한 혼합 모드
애플리케이션을 실행하는 동안 AOT 및 JIT 컴파일된 코드를 함께 사용할 수 있습니다. 이 조작 모드는 성능 향상, 특히 자주 실행하는 메소드의 성능 향상과 판별력 극대화를 제공합니다.

해석된 조작
인터프리터가 Java 애플리케이션을 실행하지만 코드 컴파일은 사용하지 않습니다.

컴파일된 코드 사용에 대한 자세한 정보는 40 페이지의 『WebSphere Real Time for RT Linux에서 컴파일된 코드 사용』의 내용을 참조하십시오.

JIT 및 AOT 컴파일 비교

AOT(Ahead-of-Time) 컴파일을 사용하면 코드를 실행하기 전에 Java 클래스 및 메소드를 컴파일할 수 있습니다. AOT 컴파일에서는 JIT 컴파일러가 중요한 성능 경로에 미치는 예측 불가능한 시간적 영향을 피할 수 있습니다. 실행 전에 코드가 컴파일되고 최고 수준의 성능을 보장하기 위해 AOT 컴파일러를 사용하여 코드를 공유 클래스 캐시로 사전 컴파일합니다.

주: AOT 컴파일된 코드는 일반적으로 JIT 컴파일된 코드만큼 신속하게 실행되지는 않습니다.

JIT(Just-In-Time) 컴파일러는 우선순위가 높은 SCHED_OTHER 스레드로 실행되며, 표준 Java 스레드보다 높지만 실시간 스레드보다 낮은 우선순위로 실행됩니다. 따라서 JIT(Just-In-Time) 컴파일에서는 실시간 코드로 비결정적 지연이 발생하지 않습니다. 따라서, JIT 컴파일에서 미리 차지하지 않기 때문에 중요한 실시간 작업을 제시간에 수행합니다. 그러나, JIT에서 큐에 있는 핫 메소드를 컴파일할 충분한 시간이 없으므로 실시간 코드가 해석된 코드로 실행될 수 있습니다. 비교 내용이 9 페이지의 그림 2에 나와 있습니다.

일반적으로, 애플리케이션에 준비 단계가 있으면 JIT로 실행하는 것이 효율적이며 필요에 따라 준비 단계가 완료될 때 JIT를 사용 불가능하게 하십시오. 이 방식에서 JIT 컴파일러는 애플리케이션이 실행되는 환경에 대한 코드를 생성할 수 있습니다.

애플리케이션에 준비 단계가 없고 주요 실행 경로가 표준 애플리케이션 조작을 통해 컴파일되었는지 확실하지 않으면 이 환경에서는 AOT 컴파일이 제대로 작동합니다.

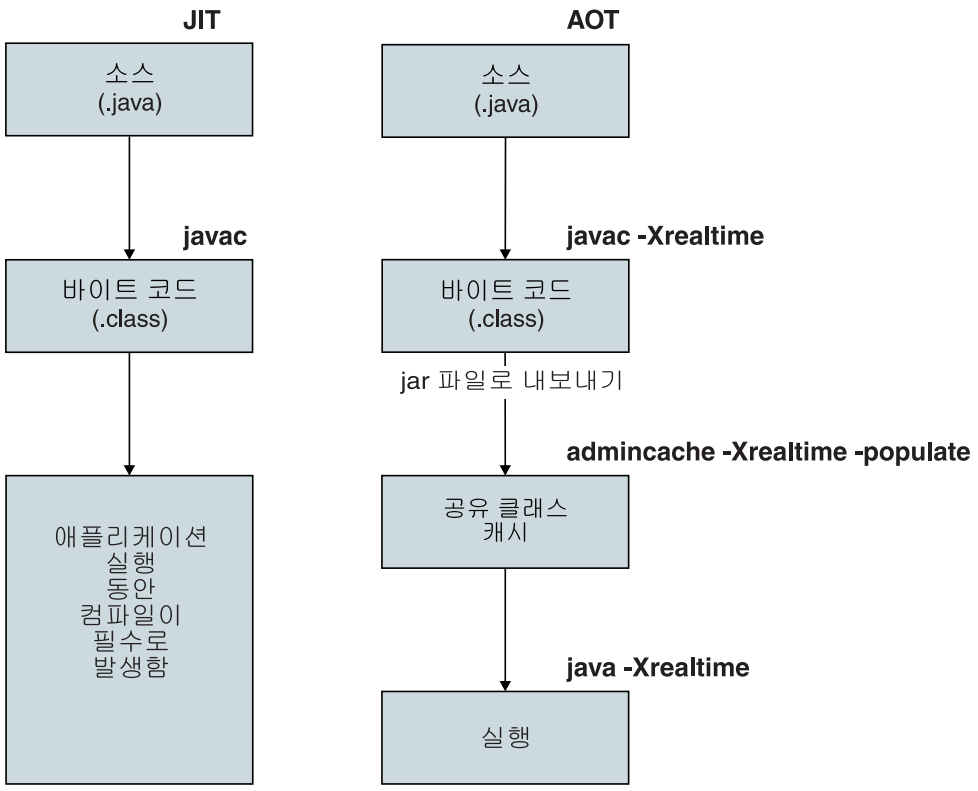

그림 2. JIT 컴파일러 및 AOT 컴파일러 비교

RTSJ 지원

WebSphere Real Time for RT Linux는 Real-Time Specification for Java(RTSJ)를 구현합니다.

WebSphere Real Time for RT Linux 버전 3.0은 RTSJ Technology Compatibility Kit 1.0.2 버전 J9 3.1.0 FCS에 호환 가능한 RTSJ로 인증되고 Java Compatibility Kit(JCK) 버전 7.0과 호환됩니다.

실시간 스레드 스케줄링 및 디스패치

실시간 Java 스레드의 스레드 스케줄링 및 디스패치는 Java용 Real Time Specification의 부분입니다. 스케줄링 정책 SCHED_FIFO는 Linux 운영 체제 우선순위 11 - 89를 사용하는 실시간 Java 스레드의 우선순위를 지정하는 데 사용됩니다.

Linux 스케줄링 정책에 대한 정보가 39 페이지의 『스레드 스케줄링 및 디스패치』에 있습니다.

스케줄 가능 및 매개변수

실시간 스케줄 가능 오브젝트는 실시간 스레드 및 비동기 이벤트 핸들러의 두 가지 기본 유형이 있습니다.

이 스케줄 가능 오브젝트는 다음과 같은 연관된 매개변수를 가집니다.

SchedulingParameters

PriorityParameters는 실시간 스케줄 가능 오브젝트를 우선순위에 따라 스케줄링합니다.

ReleaseParameters

- **PeriodicParameters**는 실시간 스케줄 가능 오브젝트의 정기적 릴리스를 설명합니다. 정기적 실시간 스레드는 정기적인 간격으로 릴리스되는 스레드입니다.
- **AperiodicParameters**는 실시간 스케줄 가능 오브젝트의 릴리스를 설명합니다. 비정기적 실시간 스레드는 비정기적인 간격으로 릴리스됩니다.

MemoryParameters

실시간 스케줄 가능 오브젝트의 메모리 할당 제한조건을 설명합니다.

ProcessingGroupParameters

WebSphere Real Time for RT Linux에서 지원되지 않습니다.

우선순위 스케줄러

WebSphere Real Time for RT Linux에서 스케줄러는 우선순위 스케줄러입니다. 이름이 암시하듯이 활성 우선순위에 따라 스케줄 가능 오브젝트의 실행을 관리합니다.

스케줄러는 스케줄 가능 오브젝트 목록을 관리하고 CPU에서 실행하기 위해 각 오브젝트를 릴리스하는 시기를 판별합니다. 스케줄러는 각 스케줄 가능 오브젝트와 연관된 다양한 매개변수를 준수해야 합니다. 이를 위해 addToFeasibility, isFeasible 및 removeFromFeasibility 메소드가 제공됩니다.

우선순위 및 정책

`java.lang.Thread` 오브젝트로 할당되는 스레드인 일반 Java 스레드는 스케줄링 정책 SCHED_OTHER, SCHED_RR 또는 SCHED_FIFO를 사용할 수 있습니다. `java.lang.RealtimeThread`로 할당되는 스레드인 실시간 스레드 및 비동기 이벤트 핸들러는 SCHED_FIFO 스케줄링 정책을 사용합니다.

일반 Java 스레드는 JVM이 SCHED_RR 또는 SCHED_FIFO 정책을 사용하는 스레드로 시작되지 않는 한, SCHED_OTHER의 기본 스케줄링 정책을 사용합니다. SCHED_OTHER 정책을 사용하는 일반 Java 스레드는 운영 체제 스레드 우선순위를 0으로 설정합니다. 정책 SCHED_RR 또는 SCHED_FIFO를 사용하는 일반 Java 스레드는 JVM을 시작하는 스레드의 우선순위를 상속받습니다.

실시간 스레드의 경우, SCHED_FIFO 정책에 시간 분할이 없으며 1(최하) - 99(최고) 범위의 99개 우선순위를 지원합니다. 이 WebSphere Real Time for RT Linux 구현은 11 - 38 범위(11과 38 포함)에서 28개 사용자 우선순위를 지원합니다.

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

11을 리턴합니다. 및

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

38을 리턴합니다.

OS 우선순위 81 - 89는 작업자 스레드 디스패치를 위해 IBM JVM에서 사용합니다. 이 스레드는 휴면 상태가 되돌아가기 전에 약간의 작업을 수행하기 위해 설계되었습니다. 스레드는 다음과 같습니다.

- 메트로놈 가비지 콜렉터 알람 스레드는 우선순위 89로 실행됩니다. 이 스레드는 정기적으로 실행되고 GC 작업 단위를 디스패치합니다.
- 비동기 신호를 처리하는 두 개의 비동기 신호 스레드. 한 개는 우선순위 88의 힙이 없는 실시간(NHRT) 스레드이고 다른 하나는 우선순위 87의 스레드입니다.
- 타이머 이벤트를 디스패치하는 두 개의 타이머 스레드. 한 개는 우선순위 85의 힙이 없는 타이머를 위한 힙이 없는 실시간 스레드이고 다른 하나는 우선순위 83의 스레드입니다.
- 비동기 이벤트 핸들러를 실행하기 위해 그리고 비동기 이벤트 핸들러를 실행하는 동안 디스패치되고 비동기 이벤트 핸들러에는 핸들러의 우선순위가 지정됩니다. 시스템이 우선순위 85의 두 개의 힙이 없는 실시간 핸들러 스레드 및 우선순위 83의 다른 8개 스레드와 함께 시작됩니다.
- 우선순위 88의 비동기 신호 힙이 없는 실시간 스레드는 힙 덤프, 코어 덤프 및 javacore 덤프에 대한 요청을 처리합니다. 일시적으로 덤프 파일을 작성하는 동안 우선순위를 89로 올립니다.

메트로놈 GC 추적 스레드는 OS 우선순위 12로 실행되고 비교를 위해 Java 메소드를 샘플링하는 JIT 샘플러 스레드는 OS 우선순위 13으로 실행됩니다.

JIT 컴파일 스레드(JIT 샘플러 스레드와 다름)는 OS 우선순위 0의 SCHED_OTHER 정책으로 실행됩니다.

JIT 컴파일 및 JIT 샘플러 스레드는 **-Xnojit** 또는 **-Xint**가 지정된 경우 둘 다 사용불가능합니다.

메트로놈 가비지 콜렉터 및 완료자 우선순위는 끊임없이 변경되어 우선순위가 가장 높은 힙 할당 스레드 이상이 됩니다(각 콜렉션 라운드 전에). 힙 할당 스레드의 우선순위가 NoHeapRealtimeThreads의 우선순위보다 낮은지 확인해야 합니다.

힙 할당 스레드는 모니터에서 차단되거나 휴면 상태가 아닌 NHRT 이외 사용자 스레드입니다. JNI 인터페이스 외부에서 원시 코드를 실행하는 사용자 스레드는 힙 할당으로 간주되지 않습니다. 힙 할당 스레드가 활성화될 때 진행 중인 가비지 콜렉션이 더 이상 모니터에서 차단되지 않거나 JNI를 벗어나면 가비지 콜렉션이 완료될 때까지 대기했다가 계속해야 합니다.

OS 우선순위 81은 힙에서 할당되는 내부 JVM 스레드를 위해 예약됩니다. 내부 JVM 스레드가 OS 우선순위 81이면 가비지 콜렉터가 OS 우선순위 82로 실행됩니다. 유일한 힙 할당 사용자 스레드가 실시간 스레드가 아니면 GC 우선순위가 OS 우선순위 11로 실행됩니다. 그렇지 않으면 GC는 우선순위가 가장 높은 힙 할당 사용자 스레드보다 하나 높은 OS 우선순위로 실행됩니다.

GC 우선순위는 콜렉션 라운드 전에만 조정합니다.

우선순위 맵핑 및 상속

각 Java 우선순위가 연관된 운영 체제 기본 우선순위에 맵핑되고 각 운영 체제 우선순위는 스케줄링 정책과 연관되어 있습니다. WebSphere Real Time for RT Linux Linux 운영 체제 스케줄링 정책은 SCHED_OTHER, SCHED_RR 및 SCHED_FIFO입니다.

실시간 Java 스레드는 SCHED_FIFO 정책을 사용하고, 일반 Java 스레드는 JVM을 시작하는 스레드의 정책을 사용합니다. 일반 Java 스레드의 기본 스케줄링 정책이 SCHED_OTHER이지만 **chrt** 같은 유틸리티를 사용하여 SCHED_RR 또는 SCHED_FIFO 정책을 설정할 수 있습니다. 스레드 우선순위 및 정책에 대한 자세한 정보는 39 페이지의 『스레드 스케줄링 및 디스패치』의 내용을 참조하십시오.

다음 표는 Java 우선순위와 기본 운영 체제 우선순위의 맵핑을 보여줍니다. 일부 Java 우선순위는 JVM에서 사용하도록 예약되고, 상응하는 Java 우선순위가 없는 일부 기본 우선순위도 JVM에서 사용합니다.

주:

- 일반 Java 스레드에서 우선순위 1-10을 사용합니다.
  - SCHED_OTHER 정책의 경우, Java 우선순위 1-10이 운영 체제 우선순위 0에 맵핑됩니다.
  - SCHED_FIFO 또는 SCHED_RR 정책의 경우, Java 우선순위 1-10이 JVM을 시작한 스레드의 우선순위를 상속받습니다.
- 우선순위 11 이상은 실시간 스레드 및 힙이 없는 실시간 스레드에서 사용합니다.
- 스케줄 가능 오브젝트는 항상 활성 우선순위로 실행됩니다. 활성 우선순위는 처음에 스케줄 가능 오브젝트의 기본 우선순위이지만 우선순위 상속으로 활성 우선순위를 일시적으로 올릴 수 있습니다. 스케줄 가능 오브젝트의 기본 우선순위는 실행하는 동안 변경 가능합니다.

사용자 기본 우선순위:

```
Java priorities 1-10: SCHED_OTHER, OS priority 0

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

내부 기본 우선순위:

```
Internal Java priority 39: SCHED_FIFO, OS priority 81
Internal Java priority 40: SCHED_FIFO, OS priority 83
Internal Java priority 41: SCHED_FIFO, OS priority 84
Internal Java priority 42: SCHED_FIFO, OS priority 85
Internal Java priority 43: SCHED_FIFO, OS priority 86
Internal Java priority 44: SCHED_FIFO, OS priority 87
Internal Java priority 45: SCHED_FIFO, OS priority 88
OS priorities 11, 12, 13
OS priorities even numbers 26, 28, 30, ..., 82
OS priority 89
```

참고 항목: http://www.rtsj.org/specjavadoc/book_index.html의 "동기화" 절

우선순위 상속:

스레드의 활성 우선순위는 우선순위가 더 높은 스레드에 필요한 잠금을 포함하고 있으므로 일시적으로 올릴 수 있습니다. 이 잠금은 동기화된 메소드 또는 동기화된 블록과 연관된 사용자 레벨 모니터 또는 내부 JVM 잠금입니다. 따라서, 일반 Java 스레드의 우선순위는 스레드가 잠금을 해제하는 시점까지 일시적으로 실시간 우선순위를 가질 수 있습니다.

우선순위 상속의 한 가지 결과로, SCHED_OTHER 스레드의 스레드 정책이 일시적으로 SCHED_FIFO로 변경됩니다.

기본 및 활성 우선순위에 대한 자세한 정보는 RTSJ 스펙에서 "동기화" 절을 참조하십시오.

메모리 관리

메모리 힙을 수집하는 가비지는 가비지 콜렉션으로 인한 예측 불가능한 동작 때문에 실시간 프로그래밍의 장애 요소로 간주됩니다. IBM WebSphere Real Time for RT Linux의 메트로놈 가비지 콜렉터는 높은 결정적 GC 성능을 제공할 수 있습니다. 동시에 Real-Time Specification for Java(RTSJ)는 Java 프로그래머가 단기 및 장기 오브젝트를 모두 명시적으로 관리할 수 있도록 가비지 수집된 힙 외부의 오브젝트에 대한 메모리 모델에 몇 가지 확장기능을 제공합니다.

메모리 영역

RTSJ는 오브젝트 할당을 위해 사용하는 메모리 영역의 개념을 도입했습니다. 일부 메모리 영역은 힙 외부에 있고 시스템 및 가비지 콜렉터가 오브젝트에 수행할 수 있는 작업 내용을 제한합니다. 예를 들어, 일부 메모리 영역의 오브젝트는 가비지 수집되지 않지만 가비지 콜렉터가 힙의 무결성 보존을 위해 이 메모리 영역에서 힙의 오브젝트 참조를 스캔할 수 있습니다.

메모리 관리의 기본 유형은 다음 3가지입니다.

- 힙 메모리는 전통적인 Java 힙이지만 메트로놈 가비지 콜렉터에서 관리합니다.
- 범위 메모리는 애플리케이션에서 특별히 요청해야 하며 힙이 없는 실시간 스레드 및 힙이 없는 비동기 이벤트 핸들러를 포함하여 실시간 스레드에서만 사용할 수 있습니다.
- 영구 메모리는 힙이 없는 실시간 스레드 및 힙이 없는 비동기 이벤트 핸들러 등 스케줄 가능 오브젝트에서 참조할 수 있는 오브젝트를 포함한 메모리 영역을 나타냅니다. 애플리케이션이 사용하지 않더라도 클래스 로딩 및 static 초기화에서 사용됩니다.

영구 메모리나 범위 메모리는 실제 메모리를 사용하도록 지정할 수 있으며 매우 빠른 액세스와 같은 특성을 가진 메모리 영역으로 구성됩니다. 일반적으로, 실제 메모리는 자주 사용되지 않으며 표준 JVM 사용자에게 영향을 미칠 가능성이 낮습니다.

힙 메모리

최대 *크기*를 **-Xmx**로 제어하지만 초기 힙 *크기*(**-Xms**)를 설정하거나 최대 힙 *크기*(**-Xmx**)와 같게 설정하지 않도록 하십시오. 힙이 초기 힙 *크기*에서 최대 힙 *크기*로 실시간으로 확장되지 않기 때문입니다. 여유 공간이 없는 최대 힙 *크기*에 도달하면 OutOf MemoryError 결과가 나타납니다. 일반적으로, 결정적 콜렉션을 지원하려면 오브젝트를

다르게 구성해야 하고 이로 인해 힙 단편화가 늘어나므로 실시간 JVM은 기존의 JVM보다 많은 힙 메모리를 이용합니다. 또한, 배열이 각각 헤더가 있는 단편으로 분해됩니다. 이는 대소형 오브젝트의 비율과 배열 사용 정도에 따라 다르지만, 애플리케이션에 20% 추가 힙 공간이 필요할 가능성이 높습니다.

메트로놈 가비지 콜렉터는 애플리케이션이 실행되는 동안 가비지를 수집한다는 점에서 메인프레임 JVM에 있는『동시』콜렉터와 비슷합니다. 완벽한 환경에서는 애플리케이션 메모리가 부족해지기 전에 콜렉션 주기가 완료되지만 할당률이 높은 일부 애플리케이션은 메트로놈 가비지 콜렉터의 수집 속도보다 빠르게 할당할 수 있습니다. 다양한 제어가 콜렉션 속도에 영향을 미치지만, 메트로놈이 OutOf MemoryError가 발생하기 전에 기존의 중지 GC로 되돌아가도록 강제하는 한 제어가 있습니다. 런타임 매개변수는 **-Xgc:synchronousGCOnOOM**이고 이에 상응하는 매개변수는 **-Xgc:nosynchronousGCOnOOM**입니다. 기본값은 **-Xgc:synchronousGCOnOOM**입니다.

범위 메모리

RTSJ는 범위 메모리의 개념을 도입했습니다. 수명이 정의된 오브젝트에서 사용할 수 있습니다. 범위를 명시적으로 입력하거나 오브젝트의 run() 메소드를 실행하기 전에 입력을 입력하는 스케줄 가능 오브젝트(실시간 스레드 또는 비동기 이벤트 핸들러)에 첨부할 수 있습니다. 각 범위는 참조 계수를 가지며 참조 계수가 0이 되면 해당 범위에 있는 오브젝트가 처리완료되고(종료됨) 이 범위와 연관된 메모리가 해제됩니다. 완료될 때까지 범위 재사용이 차단됩니다.

범위 메모리를 VTMemory 및 LTMemory라는 두 가지 유형으로 분리할 수 있습니다. 이 유형의 범위 메모리는 영역에서 오브젝트 할당에 필요한 시간에 따라 다릅니다. LTMemory는 메모리 영역에서 메모리 이용이 초기 메모리 영역 크기보다 작을 경우 선형 시간 할당을 보장합니다. VTMemory는 어떠한 보장도 제공하지 않습니다.

범위는 중첩할 수 있습니다. 중첩된 범위를 입력하면 새 범위와 연관된 메모리에서 모든 후속 할당을 가져옵니다. 중첩된 범위가 완료되면 이전의 범위가 복원되고 해당 범위에서 후속 할당을 다시 가져옵니다.

범위 오브젝트의 수명 때문에 제한된 지정 규칙을 사용하여 참조를 범위 오브젝트로 제한해야 합니다. 범위 오브젝트에 대한 참조는 외부 범위의 변수 또는 힙이나 영구 영역의 오브젝트 필드에 지정할 수 없습니다. 범위 오브젝트에 대한 참조를 동일한 범위 또는 내부 범위에만 지정할 수 있습니다. 가상 머신은 잘못된 지정 시도를 발견하고 이 경우 IllegalAssignmentError 예외를 발생시킵니다. 범위 메모리 유형을 선택할 경우 유연성이 향상되어 애플리케이션이 코드의 구문적으로 정의된 특정 영역에 적합한 특성을 가진 메모리 영역을 사용할 수 있습니다.

영역을 구성하는 동안 영역의 크기를 지정해야 하며 명령행 매개변수 **-Xgc:scopedMemoryMaximumSize**가 최대값을 제어합니다. 기본값은 8MB이고 대부분의 용도에 적합합니다.

영구 메모리

영구 메모리는 애플리케이션의 모든 스케줄 가능 오브젝트 및 스레드에서 공유되는 메모리 자원입니다. 영구 메모리에 할당된 오브젝트는 힙이 없는 스레드 및 비동기 이벤트 핸들러에서 항상 사용할 수 있으며 가비지 콜렉션으로 인한 지연의 영향을 받지 않습니다. 프로그램이 종료될 때 시스템에서 오브젝트가 자유롭게 됩니다.

해당 크기는 **-Xgc:immortalMemorySize**로 제어합니다. 예를 들어, **-Xgc:immortalMemorySize=20m**은 20MB를 설정합니다. 기본값은 16MB이며, 사용자가 많은 클래스 로딩을 수행하지 않는 한 대체로 적합합니다. 클래스 로딩이 대부분의 OutOfMemoryError 예외의 원인이 됩니다.

메모리 요구사항 예측

충분한 메모리 할당에 필요한 정보를 얻는 방법

합리적인 방법은 중요한 안전 여백으로 예상 오브젝트를 저장하는 데 필요한 메모리를 파악하는 것입니다. 애플리케이션을 분석하면 오브젝트에 필요한 실제 크기가 시스템별로 다르더라도 필요한 오브젝트의 수와 특성을 파악하는 데 도움이 됩니다. SizeEstimator 클래스를 사용하면 실제 오브젝트 크기를 고려하여 보다 유용한 정보를 제공합니다.

SizeEstimator 클래스

SizeEstimator 클래스는 오브젝트 저장에 필요한 메모리 크기를 안내하는 정보를 제공합니다. 예측치는 오브젝트 자체에 할당되어야 하는 최소 메모리 공간을 나타내며, 구성 등의 작업 중에 오브젝트에 필요한 다른 자원의 메모리 요구사항은 고려하지 않은 것입니다.

이 클래스에 대한 자세한 정보는 http://www.rtsj.org/specjavadoc/book_index.html를 참조하십시오.

메모리 사용

Java 스레드, 실시간 스레드 및 힙이 없는 실시간 스레드를 비교합니다.

Real-Time Specification for Java(RTSJ)는 실시간 스레드를 지원하기 위해 RealtimeThread 클래스 및 NoHeapRealtimeThread 클래스라는 두 가지 클래스를 추가합니다.

- 실시간 스레드 및 힙이 없는 실시간 스레드는 둘 다 스케줄 가능한 오브젝트입니다. 스케줄 가능한 오브젝트는 릴리스, 스케줄링, 메모리 및 처리 그룹 등의 매개변수를 가집니다.

- 실시간 스레드는 힙 메모리, 범위 메모리 및 영구 메모리에 있는 오브젝트에 액세스할 수 있습니다.
- 힙이 없는 실시간 스레드는 범위 메모리 및 영구 메모리 영역에만 액세스합니다.
- 힙이 없는 실시간 스레드는 다른 실시간 스레드보다 우선순위가 높아야 합니다. 우선순위가 실시간 스레드보다 낮으면 가비지 콜렉터의 방해 없이 실행할 수 있는 이점이 적용되지 않습니다.

주: 다른 실시간 스레드보다 우선순위가 높고 힙이 없는 실시간 스레드는 가비지 콜렉션에 의해 인터럽트되지 않습니다.

표 3. 실시간 스레드 및 힙이 없는 실시간 스레드의 메모리 액세스

스레드	영구 메모리	범위 메모리	힙 메모리
정상 스레드	✓	✗	✓
실시간 스레드	✓	✓	✓
힙이 없는 실시간 스레드	✓	✓	✗

메모리 영역 유형

영구 메모리

영구 메모리는 가비지 콜렉션의 영향을 받지 않습니다. 영구 메모리에 할당된 공간은 애플리케이션을 종료할 때까지 재확보할 수 없습니다.

- 이러한 영구 메모리의 특성 때문에 메모리를 재사용하는 방법을 원할 수도 있습니다. 한 가지 방법은 재사용 가능한 오브젝트 풀을 작성하는 것입니다. 또는 범위 메모리를 대신 사용합니다.
- 영구 메모리에 있는 오브젝트는 범위 메모리에 있는 항목을 참조할 수 없습니다. 영구 메모리의 오브젝트 필드에 범위 메모리의 오브젝트가 지정되면 IllegalAssignmentError 예외가 발생합니다.

범위 메모리

범위 메모리는 스케줄 가능한 오브젝트의 초기 메모리 영역으로 사용하거나 입력될 수 있습니다. 더 이상 참조되지 않으면 영역에서 모든 오브젝트를 지웁니다. 범위 메모리 영역에서 실행되는 스케줄 가능한 오브젝트가 해당 영역으로부터의 모든 오브젝트 할당을 수행합니다. 범위 메모리 영역을 사용하지 않으면 해당 영역 내부의 오브젝트가 완료되고 메모리를 재확보하여 재사용할 범위를 준비합니다. 스케줄 가능한 오브젝트에서 범위 메모리 영역을 더 이상 사용할 수 없으면 다른 용도로 사용하기 위해 메모리를 재확보합니다.

ScopedMemory 인스턴스로 설명되는 메모리 영역은 Java 힙에 존재하지 않고 가비지 콜렉션의 영향을 받지 않습니다. ScopedMemory 오

브젝트를 NoHeapRealtimeThread와 연관된 초기 메모리 영역으로 사용하거나 NoHeapRealtimeThread 내부의 ScopedMemory.enter 메소드를 사용하여 메모리 영역을 입력하는 것이 안전합니다.

실제 메모리

메모리 자체의 특성(예: 페이지 불가능 또는 비휘발성)이 중요한 경우 실제 메모리를 사용하십시오.

선형 시간 할당 스키마(LTMemory)

LTMemory는 메모리 영역에서 메모리 이용이 초기 메모리 영역 크기보다 작을 경우 선형 시간 할당이 보장되는 메모리 영역을 의미합니다. 영역의 초기 크기와 최대 크기 사이에서 메모리를 사용할 경우 할당 런타임이 다양할 수 있습니다. 더우기, 초기 및 최대 크기 범위의 메모리가 항상 사용 가능하도록 기본 시스템이 보장할 필요가 없습니다.

가변 시간 할당 스키마(VTMemory)

VTMemory 영역으로부터의 할당 실행 시간이 선형 시간으로 완료될 필요가 없다는 점을 제외하고 VTMemory는 LTMemory와 비슷합니다.

힙 메모리

힙 메모리에 있는 오브젝트는 범위 메모리에 있는 항목을 참조할 수 없습니다. 힙 메모리의 오브젝트 필드에 범위 메모리의 오브젝트가 지정되면 IllegalAssignmentError 예외가 발생합니다.

동기화 및 자원 공유

실시간 시스템에서 다른 우선순위로 실행되는 3개 이상의 스레드가 서로 동기화될 경우, 우선순위 변경이라는 조건이 발생하여 확장된 기간 동안 우선순위가 높은 스레드가 우선순위가 낮은 스레드에 의해 실행되지 않고 차단됩니다. WebSphere Real Time for RT Linux는 이 조건을 방지하기 위해 우선순위 상속이라는 스키마를 사용합니다.

우선순위가 높은 태스크가 우선순위가 낮은 태스크에 의해 실행되지 않게 차단되면 낮은 우선순위 태스크의 우선순위는 높은 우선순위 태스크가 더 이상 차단되지 않을 때까지 높은 우선순위와 일치하도록 일시적으로 순위가 올라갑니다.

정기적 및 비정기적 매개변수

실시간 스레드에는 스케줄 가능 오브젝트가 릴리스되는 빈도를 판별하는 많은 릴리스 매개변수가 있습니다. 정기적 및 비정기적 매개변수가 릴리스 매개변수의 예입니다.

정기적 매개변수

이 클래스는 정기적인 간격으로 릴리스되는 스케줄 가능 오브젝트를 위한 것입니다.

AbsoluteTime

밀리초(ms) 및 나노초(ns)로 표현합니다.

RelativeTime

밀리초(ms) 및 나노초(ns)로 표현된 특정 이벤트의 시간입니다. 예를 들어, 이벤트가 시작하고 완료될 때 절대 시간을 측정할 수 있습니다. 그런 다음 두 측정값의 차이로 상대 시간을 계산합니다.

비정기적 매개변수

이 클래스는 비정기적인 간격으로 릴리스되는 스케줄 가능 오브젝트에서 사용합니다. 첫 번째 이벤트가 완료되기 전에 두 번째 비정기적 이벤트가 발생하므로 미해결 요청의 큐 길이를 정의할 수 있습니다.

비동기 이벤트 처리

비동기 이벤트 핸들러는 스레드 외부에서 발생하는 이벤트에 상호 작용합니다(예: 애플리케이션 인터페이스로부터의 입력). 실시간 시스템에서 이 이벤트는 애플리케이션에 설정한 최종 기한 안에 응답해야 합니다.

비동기 이벤트는 시스템 인터럽트 및 POSIX 신호화 연관되어 있으며 비동기 이벤트를 타이머에 링크할 수 있습니다.

실시간 스레드와 같이 비동기 이벤트 핸들러는 연관된 많은 매개변수가 있습니다. 이 매개변수 목록은 10 페이지의 『스케줄 가능 및 매개변수』의 내용을 참조하십시오.

신호 핸들러

POSIXSignalHandler는 SIGQUIT, SIGTERM 및 SIGABRT 신호를 지원합니다. SIGQUIT의 기본 동작으로 Java 덤프가 생성됩니다. Java 덤프 생성은 CPU 시간, 파일 읽기 및 쓰기와 별도이며, 실행 중인 프로그램 조작을 방해하지 않습니다. Java 덤프가 완료될 때까지 Java 덤프 생성이 프로그램을 인터럽트합니다. Java 덤프를 생성하는 동안 애플리케이션 성능이 예측 불가능합니다.

장애 시 모든 코어 및 Java 덤프 생성을 억제하려면 **-Xdump:none**을 사용하십시오.

SIGQUIT 신호에서만 시스템 덤프 및 Java 덤프 생성을 억제하려면 **-Xdump:java:none -Xdump:java:events=gpf+abort**를 지정하십시오.

다음 신호는 POSIXSignalHandler 메커니즘에 따라 비동기 이벤트 핸들러(AEH)에 첨부할 수 있습니다(/usr/include/bits/signum.h에 정의된 신호 설명).

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

다른 신호는 현재 지원되지 않습니다. 앞에 나열된 모든 신호는 비동기 신호이고 외부에서 생성된 이벤트가 아닌 애플리케이션 또는 JVM 코드 장애를 나타내므로 동기 신호(예: SIGILL 및 SIGSEGV)에 첨부를 지원할 수 없습니다.

주: 기본적으로 SIGQUIT는 Java 애플리케이션이 JVM에 수신될 때 덤프(예: Java 덤프)를 생성하도록 합니다. 첨부된 AEH에 추가로 전달되더라도 이 전달은 혼란과 원하지 않는 동작을 유발하며 Java 명령행에서 **-Xdump:none:events=user** 옵션을 사용하여 이를 사용 불가능하게 할 수 있습니다.

필수 문서

WebSphere Real Time for RT Linux는 Real-Time Specification for Java(RTSJ)를 구현합니다.

WebSphere Real Time for RT Linux 버전 2.0은 RTSJ Technology Compatibility Kit 버전 3.0.13 FCS에 호환 가능한 RTSJ 1.0.2로 인증되고 Java Compatibility Kit(JCK) 버전 6.0과 호환됩니다.

지원되는 기능

지원되는 기능은 다음과 같습니다.

- 스케줄 가능 오브젝트가 힙에 오브젝트를 작성하는 속도를 제한하기 위해 힙 할당에 할당률 적용.

지원되지 않는 기능

지원되지 않는 기능은 다음과 같습니다.

- Priority Ceiling Emulation 프로토콜. 예를 들어, PriorityCeilingEmulation을 모니터 제어 정책으로 허용하지 않습니다.
- 스펙 준수를 위해 필요한 경우를 제외하고 원자 액세스 지원
- 기본 우선순위 스케줄로 이외의 스케줄러는 애플리케이션에서 사용할 수 없음
- 비용 적용

Real-Time Specification for Java의 필수 문서

Real-Time Specification for Java(RTSJ)의 *필수 문서* 절이 이 절에 인용되어 있습니다. 표준 RTSJ 구현의 변형이 나와 있습니다.

1. **실행 가능성 테스트 알고리즘이 기본값입니다.**

   『실행 가능성 테스트 알고리즘이 기본값이 아니면 실현 가능성 테스트 알고리즘을 설명십시오.』

2. **기본 우선순위 스케줄러만 애플리케이션에서 사용할 수 있습니다.**

   『기본 우선순위 스케줄로 이외의 스케줄러를 애플리케이션에서 사용할 수 있으면 해당 스케줄러의 동작과 스케줄러 절에 설명된 대로 다른 스케줄러와의 상호작용

을 설명하십시오. 또한, 목록이 기본 스케줄러의 스케줄 가능 오브젝트 목록과 동일하지 않는 한 스케줄러의 스케줄 가능 오브젝트를 구성하는 클래스 목록에 대해 설명합니다.』

3. **우선순위가 높은 스케줄 가능 오브젝트에서 선점하는 스케줄 가능 오브젝트는 이 우선순위로 큐의 맨 앞에 배치됩니다.**

   『우선순위가 높은 스케줄 가능 오브젝트에서 선점하는 스케줄 가능 오브젝트는 구현에서 판별한 위치에 활성 우선순위로 큐에 배치됩니다. 선점한 스케줄 가능 오브젝트가 적합한 큐의 맨 앞에 배치되지 않으면 구현에서 이러한 배치에 사용된 알고리즘을 설명해야 합니다. 이 스펙의 이후 버전에서도 큐 맨 앞에 배치해야 할 수 있습니다.』

4. **비용 적용이 지원되지 않습니다.**

   『구현이 비용 적용을 지원할 경우, 구현에서 현재 CPU 이용이 업데이트되는 세부 단위를 설명해야 합니다.』

5. **단순 순차 맵핑이 지원됩니다.**

   『인접 바이트의 단순 순차 맵핑이 아닌 한 실제 메모리 유형 필터에 의해 구현된 메모리 맵핑을 설명해야 합니다.』

6. **WebSphere Real Time for RT Linux에서 제공하는 메트로놈 가비지 콜렉터에 대한 서브클래스가 없습니다.**

   『구현에서 가비지 콜렉터 서브클래스의 동작을 전체 설명해야 합니다. 』

7. **WebSphere Real Time for RT Linux에서 제공하는 MonitorControl 서브클래스가 없습니다.**

   『이 스펙에 설명되지 않은 MonitorControl 서브클래스를 제공하는 구현이 우선순위 변경 제어와 관련된 영향 등 그 영향과 새 정책을 지원하지 않는 스케줄러를 설명해야 합니다. 』

8. **우선순위가 높은 스케줄 가능 오브젝트에 필요한 모니터를 보유한 스케줄 가능 오브젝트는 모니터를 릴리스할 때까지 우선순위를 더 높게 올립니다. 해당 시점에 스케줄 가능 오브젝트는 더 이상 실행 가능하지 않고(수행할 우선순위가 더 높은 작업이 있음) SUSE Linux Enterprise Real Time 10 SP2 업데이트 커널 버전 2.6.22.19-0.16 및 Red Hat Enterprise Linux 5.1 MRG 2.6.24.7-73 Errata 1 이전 커널에서 실행될 때 원래 (올리지 않은) 우선순위로 큐 뒤에 배치됩니다. 이 레벨 또는 그 이상 레벨의 커널은 스케줄 가능 오브젝트를 큐 앞쪽에 배치합니다.**

   『우선순위 변경 무시 알고리즘으로 인해 "올린" 우선순위가 손실될 경우 스케줄 가능 오브젝트가 새 큐의 앞쪽에 배치되지 않으면 구현이 큐 동작을 설명해야 합니다.』

9. **기본 스케줄러는 WebSphere Real Time for RT Linux에서 제공하는 스케줄러입니다.**

   『기본 스케줄러가 아닌 스케줄러의 경우, 동기화 시맨틱이 기본 PriorityInheritance 인스턴스에 정의된 규칙과 어떻게 다른지 구현에서 설명해야 합니다. 동기화 장에 있는 기본 우선순위 스케줄러 시맨틱과 동등한 우선순위 상속을 가진 새 스케줄러 동작에 대한 설명을 제공해야 합니다(지원되는 경우 Priority Ceiling Emulation 프로토콜 포함).』

10. **연관된 바운드 이벤트 핸들러 스케줄링에 대한 이벤트 실행에서 가장 나쁜 사례는 평균 40µs이며, 경쟁 스케줄 가능 오브젝트 또는 우선순위가 더 높거나 같은 시스템 활동이 없고 가비지 콜렉션이 방해하지 않는 경우 100µs를 초과하지 않습니다. 실행 메소드를 구동하는 스케줄 가능 오브젝트, AsyncEvent 오브젝트 또는 핸들러가 힙을 참조할 경우, 가비지 콜렉션의 잠재적인 영향이 (A)에 설명되어 있습니다. 여기서는 코드가 해석되고 단일 핸들러(바인드됨)가 이벤트에 구성된다고 가정합니다.**

    『(실행 가능한 높은 우선순위의 스케줄 가능 오브젝트가 없다는 가정 하에) 연관된 AsyncEventHandler를 릴리스하는 바운드로 인한 AsyncEvent 실행 간 가장 나쁜 응답 간격은 일부 참조 아키텍처를 위해 설명해야 합니다.』

11. **ATC 지원 스레드 및 첫 번째 예외 전달에서 AsynchronouslyInterruptedException 실행 간 가장 나쁜 간격은 평균 35µs이며, 경쟁 스케줄 가능 오브젝트 또는 우선순위가 더 높거나 같은 시스템 활동이 없고 가비지 콜렉션이 방해하지 않는 경우 160µs를 초과하지 않습니다. 이 경우 ATC 지원은 스레드가 ATC 유예되지 않은 영역에서 AI 지원 메소드를 실행하고 있음을 의미하며 이러한 조건은 예외 전달까지 적용됩니다. 가비지 콜렉션의 잠재적인 영향이 (A)에 설명되어 있습니다. 대상 스레드가 원시 코드에 있으면 무한정 지연될 수 있습니다. 코드를 해석하고 있다고 가정합니다.**

    『(실행 가능한 높은 우선순위의 스케줄 가능 오브젝트가 없다는 가정 하에) ATC 지원 스레드 및 첫 번째 예외 전달에서 AsynchronouslyInterruptedException 실행 간격은 일부 참조 아키텍처를 위해 설명해야 합니다. 』

12. **해당 사항 없음은 응답 4를 참조하십시오.**

    『비용 적용이 지원되고 구현이 범위 메모리에 있는 오브젝트의 완료자 실행 비용을 범위를 벗어나 범위 참조 계수를 0으로 떨어트린 항목 이외의 스케줄 가능 오브젝트에 지정할 경우, 비용 지정 규칙을 설명합니다.』

13. **RealtimeSecurity의 표준 구현 변경사항이 없습니다.**

    『RealtimeSecurity 구현이 필요한 구현보다 더 제한되거나 런타임 구성 옵션이 있으면 이 기능을 설명합니다.』

14. **범위 메모리 영역의 오브젝트 완료자는 해당 영역을 참조하는 마지막 스레드에 의해 실행됩니다. 즉, 스레드가 참조 계수를 1에서 0으로 떨어트릴 때 실행됩니다. 완료자 실행과 관련된 비용은 해당 스레드에 지정됩니다.**

   『범위에 다시 들어가고 해당 범위에 대한 getReferenceCount() 호출에서 리턴하기 전에 구현이 범위 메모리에 있는 오브젝트에 완료자를 실행할 수 있습니다. 그러나 해당 완료자를 실행할 때는 설명해야 합니다.』

15. **해상도가 설정 가능하지 않습니다.**

   『지원되는 클럭에 대해 문서는 해상도가 설정 가능한지 여부를 지정하고 설정 가능한 경우 지원되는 값을 표시해야 합니다.』

16. **WebSphere Real Time for RT Linux에서 제공하는 실시간 클럭 이외 다른 클럭이 없습니다.**

   『구현에 필수 실시간 클럭 이외의 다른 클럭이 포함되어 있으면 문서에서 해당 클럭이 사용되는 컨텍스트를 표시해야 합니다.』

주:

A 이 테스트의 참조 아키텍처는 1MB 캐시와 4GB 메모리가 장착된 LS20, 4–way, 2GHz입니다.

B 가비지 콜렉션으로 인해 힙과 연관된 스레드의 어떤 지점에서 지연이 발생할 수 있습니다. 콜렉터는 힙 메모리가 소진될 때 동작을 통제하는 두 가지 기본 모드 중 하나로 작동합니다. 이 상황에서 OutOfMemoryError를 즉시 발생시키도록 콜렉터가 설정된 경우 가장 나쁜 사례 가비지 콜렉션 지연이 일반적으로 1ms 미만입니다. 현재 일부 상황에서 더 지연될 수 있습니다. 예를 들어, 깊게 중첩된 스택 또는 많은 대형 범위를 포함한 스레드가 많은 경우입니다. OutOfMemoryError 발생 전에 동기 GC를 수행하도록 콜렉터가 설정된 경우에는 콜렉션 지연이 힙의 라이브 오브젝트 수 및 다른 메모리 영역의 오브젝트 수와 관련이 있습니다. 이 상황에서는 일반 힙 *크기* 때문에 많은 시간(초)이 소요되므로 무제한 지연으로 간주됩니다.

제 3 장 계획

WebSphere Real Time for RT Linux 설치 전에 이 섹션을 읽으십시오.

- 『마이그레이션』
-
- 『하드웨어 및 소프트웨어 전제조건』
- 26 페이지의 『고려사항』

마이그레이션

WebSphere Real Time for RT Linux는 실시간 애플리케이션을 위해 수정한 Linux 환경에서 실행됩니다. 실시간 환경에서 표준 Java 애플리케이션을 사용할 수 있습니다. 또는, WebSphere Real Time의 기능을 이용하도록 애플리케이션을 수정할 수 있습니다.

시스템 마이그레이션

Linux 지원팀에서 제공하는 지시사항을 따르십시오.

하드웨어 및 소프트웨어 전제조건

WebSphere Real Time for RT Linux에 지원되는 하드웨어, 운영 체제 및 Java 환경을 확인하려면 이 목록을 사용하십시오.

하드웨어

WebSphere Real Time for RT Linux 인증된 하드웨어 구성은 다음 시스템에서 멀티프로세서를 변형한 것입니다.

- IBM BladeCenter® LS20 (유형 8850-76U, 8850-55U, 7971, 7972)
- IBM eServer™ xSeries® 326m (유형 7969-65U, 7969-85U, 7984-52U, 7984-6AU)
- IBM BladeCenter LS21 (유형 7971-6AU)
- IBM BladeCenter HS21 XM 듀얼 쿼드 코어 (유형 7995)

WebSphere Real Time for RT Linux에 대해 인증을 유지하려면 하이퍼스레딩이 지원되는 IBM 시스템에서 하이퍼스레딩을 사용 불가능하게 설정해야 합니다.

또한, WebSphere Real Time for RT Linux는 지원되는 운영 체제를 실행하고 다음 특성을 가진 하드웨어에서 지원됩니다.

- 최소 512MB의 실제 메모리

- 최소 Intel Pentium 4, AMD Opteron 또는 Intel Atom 프로세서.

인증된 하드웨어 구성이 아닌 시스템의 경우, IBM은 성능에 관한 어떠한 언급도 하지 않습니다. 인증된 하드웨어 구성의 성능 고려사항은 101 페이지의 제 7 장 『성능』에 자세히 설명되어 있습니다.

하이퍼스레딩이 지원되는 시스템에서는 WebSphere Real Time for RT Linux 사용 시 성능 악영향을 방지하기 위해 하이퍼스레딩를 사용하지 않도록 설정하십시오.

운영 체제

- Red Hat Enterprise Linux 5.3 MRG. 29 페이지의 『Real Time Linux 환경 설치』의 내용을 참조하십시오.
- SUSE Linux Enterprise Real Time (SLERT) 10. 29 페이지의 『Real Time Linux 환경 설치』의 내용을 참조하십시오.

고려사항

WebSphere Real Time for RT Linux를 사용할 때 많은 요소를 알고 있어야 합니다.

- 가능하면, 동일한 시스템에서 둘 이상의 실시간 JVM을 실행하지 마십시오. 여러 개의 가비지 콜렉터가 있을 수 있기 때문입니다. 각 JVM은 서로의 메모리 영역을 알지 못합니다. 이 경우 JVM 전반에서 GC 사이클 및 일시정지 시간을 조정할 수 없으며 이는 한 개의 JVM이 또 다른 JVM의 GC 성능에 부정적인 영향을 줄 수 있음을 의미합니다. 여러 JVM을 사용해야 하는 경우, **taskset** 명령을 사용하여 각 JVM이 프로세서의 특정 서브세트로 바인드되는지 확인하십시오.
- AOT(Ahead-of-Time) 컴파일러를 사용하여 사전 컴파일한 코드와 함께 **-Xdebug** 옵션 및 **-Xnojit** 옵션을 사용할 수 없습니다. **-Xdebug**가 AOT(Ahead-of-Time) 컴파일러와 다른 방법으로 코드를 컴파일하며 지원되지 않기 때문입니다.

  코드를 디버그하려면 해석되거나 JIT 컴파일된 코드를 사용하십시오.
- com.sun.tools.javac.Main 인터페이스를 사용하여 javax.realtime 패키지를 사용하는 Java 소스 코드를 컴파일할 경우, `sdk/jre/lib/i386/realtime/jclSC170/realtime.jar`이 클래스 경로에 있는지 확인해야 합니다. 이 유형 컴파일의 한 가지 일반적인 예로 ant 컴파일이 있습니다.
- 선택적 JavaComm 패키지를 WebSphere Real Time for RT Linux에 설치하고 실시간 및 실시간 이외 JVM 모두에서 액세스할 수 있습니다. 설치 및 구성에 대한 자세한 정보는 http://publib.boulder.ibm.com/infocenter/java7sdk/v7r0/topic/com.ibm.java.lnx.70.doc/user/jcommchapter.html의 내용을 참조하십시오. WRT의 실시간 JVM은 일반 Java 스레드와 함께 사용하는 JavaComm API를 지원합니다. 그러나, JavaComm으로 외부 디바이스에 액세스할 때 판별력 또는 실시간 성능과 관

련하여 어떤한 보장도 없습니다. 따라서, 실시간 동작이 필요한 경우 또는 힙이 없는 실시간 스레드 및 실시간 스레드의 경우 JavaComm을 사용하지 마십시오.

- 이전 WebSphere Real Time for RT Linux 릴리스에서 사전 컴파일된 코드 및 클래스를 저장하도록 사용한 공유 캐시는 이 WebSphere Real Time for RT Linux 릴리스에서 사용한 캐시와 호환되지 않습니다. 이전 캐시의 컨텐츠를 다시 생성해야 합니다.
- 공유 클래스 캐시를 사용할 때는 캐시 이름이 53자를 초과하지 않아야 합니다.
- **ps** 명령은 Java 스레드 이름을 자릅니다.

  ps 명령은 15자로 제한됩니다. 스레드 이름을 16자 이상으로 설정하면 **ps** 명령에 의해 이름이 잘립니다.
- WebSphere Real Time for RT Linux는 NTLoginModule(NTLM) 인증을 지원하지 않습니다.

  NTLoginModule(NTLM)은 Windows 서비스에 대한 액세스를 인증하는 데 사용됩니다. NTLM을 사용한 인증은 Windows 플랫폼에서만 지원됩니다. 즉, WebSphere Real Time for RT Linux는 NTLM 인증을 지원하지 않습니다.

제 4 장 WebSphere Real Time for RT Linux 설치

제품을 설치하려면 다음 단계를 따르십시오.

- 『설치 파일』
- 『Real Time Linux 환경 설치』
- 30 페이지의 『InstallAnywhere 패키지에서 설치』
  - 32 페이지의 『수동 설치 완료』
  - 33 페이지의 『자동 설치 완료』
  - 34 페이지의 『알려진 문제와 제한사항』
- 35 페이지의 『경로 설정』
- 36 페이지의 『클래스 경로 설정』
- 37 페이지의 『설치 테스트』
- 38 페이지의 『WebSphere Real Time for RT Linux 설치 제거』

설치 파일

이 설치 파일이 필요합니다.

IBM WebSphere Real Time for RT Linux는 두 가지 유형의 InstallAnywhere 패키지로 제공됩니다.

설치 가능 패키지

설치 가능 패키지는 시스템을 구성합니다. 예를 들어, 프로그램이 환경 변수를 설정할 수 있습니다.

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

아카이브 패키지

이 패키지는 파일을 시스템에 추출하지만 구성은 수행하지 않습니다.

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

Real Time Linux 환경 설치

WebSphere Real Time for RT Linux를 설치하기 전에 Real Time Linux를 설치해야 합니다.

시작하기 전에

WebSphere Real Time for RT Linux를 설치하기 전에 64비트 버전의 Real Time Linux를 설치해야 합니다.

Red Hat Enterprise Linux 5.3 MRG

- Red Hat Enterprise Linux 5.3 MRG의 Real Time 구성요소 설치에 대한 자세한 정보는 RT-Linux RHEL 5.3 MRG 1.1.2의 설치 지침을 참조하십시오(https://www.redhat.com/docs/en-US/Red_Hat_Enterprise_MRG/1.1/html/Realtime_Installation_Guide/index.html).

SUSE Linux Enterprise Real Time 10

- SUSE Linux Enterprise Real Time 10 설치에 대한 자세한 정보는 http://www.novell.com/products/realtime/eval.html의 내용을 참조하십시오.

다양한 클래스 인스턴스를 로드할 때 많은 수의 파일 디스크립터를 사용하는 경우, "java.util.zip.ZipException: error in opening zip file"와 같은 오류 메시지가 표시되거나 , 파일을 열 수 없다고 보고하는 IOException 예외가 다른 형태로 표시될 수 있습니다. 해결 방법은 파일 디스크립터 규정을 **ulimit** 명령을 사용하여 늘리는 것입니다. 열린 파일의 현재 한계를 알려면 다음 명령을 사용하십시오.

```
ulimit -a
```

더 많은 파일을 열려면

```
ulimit -n 8196
```

명령을 사용하십시오.

InstallAnywhere 패키지에서 설치

이 패키지는 설치 옵션을 안내하는 대화식 프로그램을 제공합니다. 그래픽 사용자 인터페이스 또는 시스템 콘솔에서 프로그램을 실행할 수 있습니다.

시작하기 전에

시스템에 다음 공유 라이브러리가 둘 다 있어야 합니다.

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

libstdc++.so.5 공유 라이브러리가 없으면 설치할 때 다음 오류를 포함한 Java 코어 덤프가 표시될 수 있습니다.

```
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
```

설치 가능한 패키지를 설치하는 경우 rpm-build 도구가 시스템에 설치되어 있어야 하며, 그렇지 않으면 설치 프로그램이 새 패키지를 RPM 데이터베이스에서 등록할 수 없습니다. rpm-build 도구가 설치되어 있는지 알아보려면 다음 명령을 입력하십시오.

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

이 태스크 정보

InstallAnywhere 패키지는 `.bin` 파일 확장자를 가집니다.

두 가지 유형의 패키지가 있습니다.

설치 가능
: 이러한 패키지를 설치하면 예를 들면 환경 변수를 설정하여 시스템도 구성합니다.

아카이브
: 이러한 패키지를 설치하면 파일을 시스템에 추출하지만, 구성은 수행하지 않습니다.

프로시저

- 패키지를 대화식으로 설치하려면, 수동 설치를 완료하십시오.
- 추가 사용자 상호작용 없이 패키지를 설치하려면, 자동 설치를 완료하십시오. 많은 시스템을 설치하려는 경우 이 옵션을 선택할 수 있습니다.
- 설치 프로세스가 완료되면 경로 및 클래스 경로 환경 변수 설정과 같은 섹션에서 구성 단계를 따르십시오.

결과

제품이 설치됩니다.

주: 예를 들면 Ctrl+C를 눌러 설치 프로세스를 인터럽트하지 마십시오. 프로세스를 인터럽트하면 제품을 다시 설치해야 할 수 있습니다. 자세한 정보는 웹 사이트 34 페이지의 『인터럽트된 설치』의 내용을 참조하십시오.

설치 가능한 메시지를 사용하는 경우 문제점이 발견되었음을 알리는 메시지가 표시될 수도 있습니다. 아카이브 패키지의 설치에서는 메시지가 생성되지 않습니다. 설치 가능한 패키지를 사용할 때 표시될 수 있는 메시지 중 일부를 다음 목록에서 보여줍니다.

`The installer cannot run on your configuration. It will now quit.`
: 이 오류 메시지는 설치 프로세스를 실행할 수 있는 권한이 사용자 ID에 부여되지 않은 경우 발생합니다. 계속할 수 없으므로 설치 프로그램이 종료됩니다. 문제점을 수정하려면 루트 권한이 있는 사용자 ID로 설치를 다시 시작하십시오.

An RPM package is already installed. Uninstall the package before proceeding.

이 메시지는 RPM 패키지가 이미 설치되어 있음을 표시합니다. 계속할 수 없으므로 설치 프로그램이 종료됩니다. 문제점을 수정하려면 진행하기 전에 RPM 패키지를 설치 제거하십시오.

수동 설치 완료

InstallAnywhere 패키지에서 대화식으로 제품을 설치합니다.

시작하기 전에

설치 프로세스를 시작하기 전에 다음 조건을 확인하십시오.

- 이전에 RPM 패키지에서 WebSphere Real Time for RT Linux를 설치한 경우에는 진행하기 전에 이 패키지를 설치 제거해야 합니다.
- 루트 권한을 가진 사용자 ID가 있어야 합니다.

프로시저

1. 설치 패키지 파일을 임시 디렉토리에 다운로드하십시오.
2. 임시 디렉토리로 변경하십시오.
3. 쉘 프롬프트에서 *./package*를 입력하여 설치 프로세스를 시작하십시오. 여기서 *package*는 설치하는 패키지의 이름입니다.
4. 설치 프로그램 창에 표시된 목록에서 언어를 선택한 후 **다음**을 클릭하십시오. 사용 가능한 언어의 목록은 시스템의 로케일 설정에 기반합니다.
5. 화면이동 막대를 사용하여 라이센스 텍스트의 끝까지 도달하면서 라이센스 계약을 읽으십시오. 설치를 진행하려면 라이센스 계약의 이용 약관에 동의해야 합니다. 이용 약관에 동의하려면 단일 선택 단추를 선택한 후 **확인**을 클릭하십시오.

   주: 라이센스 텍스트의 끝까지 읽어야 단일 선택 단추를 선택하여 라이센스 계약에 동의할 수 있습니다.
6. 설치의 대상 디렉토리 선택을 묻는 메시지가 나타납니다. 기본 디렉토리에 설치하지 않으려면, 브라우저 창을 사용하여 대체 디렉토리를 선택하도록 **선택**을 클릭하십시오. 설치 디렉토리를 선택했으면 **다음**을 클릭하여 계속하십시오.
7. 선택사항 검토를 묻는 메시지가 나타납니다. 선택사항을 변경하려면 **이전**을 클릭하십시오. 선택사항이 올바르면 **설치**를 클릭하여 설치를 진행하십시오.
8. 설치 프로세스가 완료되면 **완료**를 클릭하여 종료하십시오.

자동 설치 완료

설치할 시스템이 둘 이상이고 사용할 옵션을 이미 알고 있는 경우, 자동 설치 프로세스를 사용할 수 있습니다. 수동 설치를 사용하여 한 번 설치한 다음, 결과 응답 파일을 사용하여 추가적인 사용자 상호작용 없이 추가 설치를 완료합니다.

프로시저

1. 수동 설치를 완료하여 응답 파일을 작성하십시오. 다음 옵션 중 하나를 사용하십시오.
   - GUI를 사용하여 설치 프로그램에서 응답 파일을 작성함을 지정합니다. 응답 파일은 installer.properties라는 파일이며, 설치 디렉토리에 작성됩니다.
   - 명령행을 사용하고 수동 설치 명령에 -r 옵션을 추가하여 응답 파일에 대한 전체 경로를 지정합니다. 예를 들면 다음과 같습니다.

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

   예제 응답 파일 컨텐츠:

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

   이 예제에서 */my_directory*는 설치에 선택한 대상 설치 디렉토리입니다.
2. 옵션: 필요한 경우, 옵션을 변경하려면 응답 파일을 편집하십시오.

   주: 아카이브 패키지에 다음의 알려진 문제가 있습니다. 응답 파일을 사용하는 설치에서 응답 파일에 있는 디렉토리를 변경해도 기본 디렉토리를 사용합니다. 이전 설치가 기본 디렉토리에 있으면 겹쳐씁니다.

   설치 옵션이 각각 다른 응답 파일을 두 개 이상 작성하는 경우, 각 응답 파일에 고유한 이름을 *myfile*.properties 형식으로 지정하십시오.
3. 옵션: 로그 파일을 생성하십시오. 자동으로 설치하는 중이므로, 설치 프로세스의 끝에 상태 메시지가 표시되지 않습니다. 설치 상태를 포함하는 로그 파일을 생성하려면 다음 단계를 완료하십시오.
   a. 다음 명령을 사용하여 필수 시스템 특성을 설정하십시오.

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

   b. 로그 출력을 콘솔로 보내기 위해 다음 환경 변수를 설정하십시오.

      ```
      export LAX_DEBUG=1
      ```
4. 패키지 설치 프로그램을 **-i** 자동 옵션 및 응답 파일을 지정하는 **-f** 옵션으로 실행하여 자동 설치를 시작하십시오. 예를 들면 다음과 같습니다.

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

특성 파일에 대한 완전한 경로 또는 상대 경로를 사용할 수 있습니다. 이러한 예제에서 `1>console.txt 2>&1` 문자열은 설치 프로세스 정보를 stderr 및 stdout 스트림에서 현재 디렉토리의 `console.txt` 로그 파일로 경로 재지정합니다. 설치에 문제점이 있다고 판단되면 이 로그 파일을 검토하십시오.

주: 설치 디렉토리에 여러 응답 파일이 포함된 경우, 기본 응답 파일인 `installer.properties`가 사용됩니다.

인터럽트된 설치

패키지 설치 프로그램이 설치 중에 예상치 못하게 중지되는 경우(예: Ctrl+C를 누르는 경우), 설치가 손상되어 제품을 설치 제거하거나 다시 설치할 수 없습니다. 설치 제거하거나 다시 설치하려고 시도하면 심각한 애플리케이션 오류 메시지가 표시됩니다.

이 태스크 정보

이 문제점을 해결하려면 다음 단계에서 설명된 대로 파일을 삭제하고 다시 설치하십시오.

프로시저

1. `/var/.com.zerog.registry.xml` 레지스트리 파일을 삭제하십시오.
2. 설치가 들어 있는 디렉토리를 삭제(작성된 경우)하십시오. (예: opt/IBM/javawrt3/)
3. 설치 프로그램을 다시 실행하십시오.

알려진 문제와 제한사항

InstallAnywhere 패키지에는 알려진 문제와 제한사항이 있습니다.

- `libstdc++.so.5` 공유 라이브러리가 시스템에 없으면 Java 코어 덤프가 생성되면서 설치에 실패합니다. 자세한 정보는 30 페이지의 『InstallAnywhere 패키지에서 설치』의 내용을 참조하십시오.
- 설치 패키지 GUI에서는 Orca 화면 판독 프로그램이 지원되지 않습니다. GUI에 대한 대체로서 자동 설치 모드를 사용할 수 있습니다.
- 설치 후 다음 작업을 수행하십시오. `./package`를 입력하여 프로그램을 다시 시작하면 프로그램이 다음 메시지를 표시합니다.

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

  Enter를 눌러 기본값을 수락하는 경우 프로그램이 응답하지 않습니다. 숫자를 입력한 후 Enter를 누르십시오.
- 패키지를 설치한 후 다른 모드(예: 콘솔 또는 자동)에서 다시 설치하려고 시도하면, 다음 오류 메시지가 표시될 수 있습니다.

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

GUI 모드를 사용하여 설치했고 콘솔 모드에서 설치 프로그램을 다시 실행하는 경우에는 이 메시지가 표시되지 않습니다. 이 오류가 표시되며 설치 제거 옵션(설치 가능 패키지만)을 선택하기 위해 프로그램을 실행하는 경우, 38 페이지의 『WebSphere Real Time for RT Linux 설치 제거』에 설명된 대로 **./_uninstall/uninstall** 명령을 대신 사용하십시오.

설치 가능 패키지만

- InstallAnywhere 패키지를 사용하여 기존 설치를 업그레이드할 수 없습니다. WebSphere Real Time for RT Linux를 업그레이드하려면 우선 기존 버전을 설치 제거해야 합니다.
- 다른 설치 디렉토리를 사용하는 경우라도 동시에 동일한 시스템에서 동일한 버전의 WebSphere Real Time for RT Linux의 다른 두 인스턴스를 동시에 설치할 수 없습니다. 예를 들어, `/previous` 디렉토리에 있는 WebSphere Real Time for RT Linux V3와 `/current` 디렉토리에 있는 WebSphere Real Time for RT Linux 서비스 새로 고치기 설치를 동시에 실행할 수 없습니다. 설치 프로그램이 버전 번호를 확인합니다. 프로그램이 동일한 버전 번호의 기존 패키지를 찾으면 기존 패키지를 설치 제거할지 여부를 선택해야 합니다.
- 패키지가 설치되어 있고 GUI를 사용하여 패키지 설치 프로그램을 다시 실행하는 경우, 패키지를 설치 제거하도록 선택할 수 있습니다. 이 설치 제거 옵션은 무인 모드에서는 사용할 수 없습니다. 패키지 설치 프로그램을 무인 모드에서 다시 실행하는 경우, 프로그램은 실행되지만 조치는 수행되지 않습니다.

아카이브 패키지만

- 응답 파일의 설치 디렉토리를 변경한 후 해당 응답 파일을 사용하여 자동 설치를 실행하면, 설치 프로그램은 새 설치 디렉토리를 무시하고 기본 디렉토리를 대신 사용합니다. 이전 설치가 기본 디렉토리에 있으면 겹쳐씁니다.

경로 설정

PATH 환경 변수를 설정한 경우, 쉘 프롬프트에 이름을 입력하여 애플리케이션 또는 프로그램을 실행할 수 있습니다.

이 태스크 정보

주: 이 절에 설명된 대로 **PATH** 환경 변수를 변경하면 경로에 있는 기존 Java 실행 파일이 대체됩니다.

매번 도구 이름 앞에 경로를 입력하여 경로를 도구에 지정할 수 있습니다. 예를 들어, SDK가 `opt/IBM/javawrt3/`에 설치된 경우 쉘 프롬프트에 다음을 입력하여 *myfile.java* 파일을 컴파일할 수 있습니다.

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

전체 경로를 매번 입력하지 않으려면 다음을 수행하십시오.

1. 홈 디렉토리에 있는 쉘 시작 파일(쉘에 따라 다르며, 일반적으로 **.bashrc**)을 편집하고 절대 경로를 **PATH** 환경 변수에 추가하십시오. 예:

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

2. 다시 로그온하거나 업데이트된 쉘 스크립트를 실행하여 새 **PATH** 설정을 활성화하십시오.
3. 파일을 **javac** 도구와 컴파일하십시오. 예를 들어, *myfile.java* 파일을 컴파일하려면 쉘 프롬프트에서 다음을 입력하십시오.

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

   PATH 환경 변수를 사용하면 Linux에서 **javac**, **java** 및 **javadoc** 도구와 같은 실행 파일을 현재 디렉토리에서 찾을 수 있습니다. 경로의 현재 값을 표시하려면 명령 프롬프트에서 다음을 입력하십시오.

   ```
   echo $PATH
   ```

다음에 수행할 작업

CLASSPATH 환경 변수를 설정해야 하는지 여부를 판별하려면 『클래스 경로 설정』의 내용을 참조하십시오.

클래스 경로 설정

CLASSPATH 환경 변수는 SDK 도구(예: **java**, **javac** 및 **javadoc** 도구)에 Java 클래스 라이브러리가 있는 위치를 알려줍니다.

이 태스크 정보

다음 중 하나에 해당되는 경우에만 **CLASSPATH** 환경 변수를 명시적으로 설정하십시오.

- 직접 개발하거나 현재 디렉토리에 없는 새로운 라이브러리 또는 클래스 파일이 필요한 경우
- `bin` 및 `lib` 디렉토리의 위치를 변경하여 상위 디렉토리가 더 이상 동일하지 않은 경우.
- 동일한 시스템에서 다른 런타임 환경을 사용하는 애플리케이션을 개발하거나 실행할 계획인 경우.

CLASSPATH의 현재 값을 표시하려면 쉘 프롬프트에 다음을 입력하십시오.

```
echo $CLASSPATH
```

다른 런타임 환경(별도로 설치한 다른 버전 포함)을 사용하는 애플리케이션을 개발하고 실행하는 경우, 각 애플리케이션에 대해 **CLASSPATH** 및 **PATH**를 명시적으로 설정해야

합니다. 여러 애플리케이션을 동시에 실행하며 다른 런타임 환경을 사용하는 경우, 각 애플리케이션을 자체 쉘에서 실행해야 합니다.

한 번에 한 개의 Java 버전만 실행하려면 쉘 스크립트를 사용하여 다른 런타임 환경 간에 전환할 수 있습니다.

다음에 수행할 작업

성공적으로 설치되었는지 확인하려면 『설치 테스트』의 내용을 참조하십시오.

설치 테스트

성공적으로 설치되었는지 확인하려면 **-version** 옵션을 사용하십시오.

이 태스크 정보

Java 설치는 표준 JVM 및 실시간 JVM으로 구성됩니다.

프로시저

설치를 테스트하려면 다음 단계를 따르십시오.

1. 표준 JVM의 버전 정보를 알려면 쉘 프롬프트에서 다음 명령을 입력하십시오.

   ```
   java -version
   ```

   이 명령은 성공할 경우 다음 메시지를 리턴합니다.

   ```
   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
   ```

   실시간 JVM이 아닌 표준 JVM을 사용하려는 경우 Linux기반 Java v7용 IBM 사용자 안내서를 참조하십시오.

   주: 버전 정보는 올바르지만 날짜가 이 예제에 나온 날짜보다 이후일 수 있습니다. 날짜 문자열의 형식은 yyyymmdd이며 뒤에 구성요소별 추가 정보가 나올 수도 있습니다.

2. 실시간 JVM의 버전 정보를 알려면 쉘 프롬프트에서 다음 명령을 입력하십시오.

   ```
   java -Xrealtime -version
   ```

   이 명령은 성공할 경우 다음 메시지를 리턴합니다.

```
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
```

주: 버전 정보는 올바르지만 플랫폼 아키텍처 및 날짜가 예제와 다를 수 있습니다. 날짜 문자열의 형식은 yyyymmdd이며 뒤에 구성요소별 추가 정보가 나올 수도 있습니다.

WebSphere Real Time for RT Linux 설치 제거

WebSphere Real Time for RT Linux의 제거에 사용하는 프로세스는 사용한 설치 유형에 따라서 달라집니다.

시작하기 전에

InstallAnywhere 설치 가능 패키지의 경우 루트 권한을 가진 사용자 ID가 있어야 합니다.

이 태스크 정보

InstallAnywhere 아카이브 패키지의 경우 설치 제거 프로세스가 없습니다. 시스템에서 아카이브 패키지를 제거하려면, 패키지를 설치할 때 선택한 대상 디렉토리를 삭제하십시오. InstallAnywhere 설치 가능 패키지의 경우, 다음 단계에서 설명된 대로 명령을 사용하거나 설치 프로그램을 다시 실행하여 제품을 설치 제거합니다.

프로시저

- 옵션: **uninstall** 명령을 사용하여 수동으로 설치 제거하십시오.
  1. 설치가 포함된 디렉토리로 변경하십시오. 예를 들면 다음과 같습니다.

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

  2. `./_uninstall/uninstall` 명령을 입력하여 설치 제거 프로세스를 시작하십시오.
- 옵션: 설치 제거 프로그램을 쉽게 찾을 수 없는 경우, 대체하여 다른 수동 설치를 실행할 수 있습니다. 설치 프로그램에서 제품이 이미 설치되었음을 발견하면 이전 설치를 설치 제거하는 기회를 제공합니다.

제 5 장 IBM WebSphere Real Time for RT Linux 애플리케이션 실행

실시간 애플리케이션 실행 시 유용한 중요한 정보입니다.

- 40 페이지의『WebSphere Real Time for RT Linux에서 컴파일된 코드 사용』
- 61 페이지의『힙이 없는 실시간 스레드 사용』
- 70 페이지의『JVM 간 클래스 데이터 공유』
- 72 페이지의『메트로놈 가비지 콜렉터 사용』

스레드 스케줄링 및 디스패치

Linux 운영 체제는 다양한 스케줄링 정책을 지원합니다. 기본 유니버셜 시간 공유 스케줄링 정책은 대부분의 스레드에 사용되는 SCHED_OTHER입니다. SCHED_RR 및 SCHED_FIFO는 실시간 애플리케이션의 스레드가 사용할 수 있습니다. 에서 사용합니다.

커널은 프로세서에서 실행할 다음 실행 가능 스레드를 결정합니다. 커널이 실행 가능 스레드 목록을 관리합니다. 우선순위가 가장 높은 스레드를 찾고 실행할 다음 스레드로 해당 스레드를 선택합니다.

다음 명령을 사용하여 스레드 우선순위 및 정책을 나열할 수 있습니다.

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

여기서 policy:

- TS는 SCHED_OTHER입니다.
- RR은 SCHED_RR입니다.
- FF는 SCHED_FIFO입니다.
- -는 보고된 정책이 없습니다.

출력은 다음 예제와 같습니다.

```
  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
```

이 출력은 Java 프로세스, 적용되는 스케줄링 정책, 우선순위 『-』(기타)의 기본 스레드 및 우선순위 11 - 89의 몇 가지 실시간 스레드를 보여줍니다.

현재 스케줄링 정책을 조회하려면 예제에 표시된 **sched_getscheduler** 또는 **ps** 명령을 사용하십시오.

프로세스에 대한 자세한 정보는 106 페이지의 『일반 디버깅 기술』를 참조하십시오.

실시간 Java 스레드 우선순위 및 정책

java.realtime.RealtimeThread로 할당되는 스레드인 실시간 스레드 및 비동기 이벤트 핸들러는 SCHED_FIFO 스케줄링 정책을 사용합니다.

실시간 Java 스레드의 스레드 스케줄링 및 디스패치는 Real Time Specification for Java(RTSJ)의 부분입니다. 실시간 Java 스레드의 스케줄링 정책 및 우선순위 처리를 포함한 이 주제는 9 페이지의 『RTSJ 지원』 절에서 설명합니다.

WebSphere Real Time for RT Linux에서 컴파일된 코드 사용

IBM WebSphere Real Time for RT Linux는 몇 가지 코드 컴파일 모델을 지원하고 다양한 레벨의 코드 성능 및 판별을 제공합니다.

해석된 조작

가장 간단한 코드 컴파일 모델입니다. 인터프리터가 Java 애플리케이션을 실행하지만 코드 컴파일은 사용하지 않습니다. 인터프리터는 유용한 판별력을 제공하지만, 성능이 낮아지므로 프로덕션 시스템에서는 이 모드의 조작을 사용하지 않는 것이 좋습니다.

해석된 조작을 사용하려면 Java 명령행에 **-Xint** 옵션을 지정하십시오.

우선순위가 낮은 JIT(Just-In-Time) 컴파일

WebSphere Real Time for RT Linux의 기본 컴파일 모델은 JIT(Just-In-Time) 컴파일러를 사용하여 애플리케이션을 실행하는 동안 Java 애플리케이션의 중요한 메소드를 컴파일합니다. 이 모드에서는 JIT 컴파일러가 실시간 이외 JVM의 JIT 컴파일러 조작과 비슷한 방식으로 작동합니다. 차이점은 WebSphere Real Time for RT Linux JIT 컴파일러가 실시간 스레드보다 낮은 우선순위로 실행된다는 점입니다. 우선순위가 낮으면 애플리케이션이 실시간 태스크를 수행할 필요가 없을 때 JIT 컴파일러가 시스템 자원을 사용하게 됩니다. 따라서 JIT 컴파일러는 실시간 태스크의 성능에 커다란 영향을 미치지 않습니다.

JIT 컴파일러는 컴파일 관련 활동에 대해 컴파일 스레드와 샘플러 스레드라는 두 가지 스레드를 사용합니다. 이 스레드는 실시간 태스크보다 낮은 우선순위로 실행됩니다. 컴파일 스레드는 애플리케이션에 비동기 방식으로 실행됩니다. 이는 컴파일 스레드가 메소드 컴파일을 완료할 때까지 애플리케이션 스레드가

기다리지 않음을 의미합니다. 샘플러 스레드는 각 스레드에서 현재 실행 중인 메소드를 식별할 수 있도록 비동기 메시지를 애플리케이션 스레드로 정기적으로 보냅니다. 애플리케이션 스레드에서 메시지를 처리하는 데에는 시간이 거의 걸리지 않습니다. 실시간 태스크의 우선순위가 높아 샘플링 스레드를 실행할 수 없으면 메시지를 보내지 않습니다. JIT 컴파일러를 사용하면 판별력에 작은 영향이 있지만 이 컴파일 모드는 많은 사용자에게 최상의 성능을 제공합니다.

JIT를 사용하여 낮은 우선순위로 애플리케이션을 실행하려면 59 페이지의 『JIT 사용』의 내용을 참조하십시오.

AOT(Ahead-of-time) 사전 컴파일된 코드

WebSphere Real Time for RT Linux는 애플리케이션 실행 전에 사전 컴파일 단계에서 원시 코드에 대해 Java 메소드를 컴파일합니다. WebSphere Real Time for RT Linux V2 전에 사전 컴파일 단계는 jxeinajar 도구를 통해 AOT(Ahead-of-Time) 컴파일러를 사용하여 메소드를 컴파일하고 결과를 특수 Java 실행 파일에 저장합니다. 이 파일을 바인드된 jar 파일에 수집할 수 있습니다. 애플리케이션을 실행할 때 바인드된 jar 파일을 애플리케이션 클래스 경로에 추가하여 메소드의 클래스를 JXE에서 로드할 때 JVM이 AOT 코드를 로드할 수 있도록 합니다. 이 방법을 사용하면 명령행에 **-Xnojit** 옵션을 지정하여 JIT 컴파일러가 완전히 사용 불가능하게 됩니다. 애플리케이션은 다른 메소드의 인터프리터 및 작성된 사전 컴파일된 AOT 코드를 사용할 수 있습니다. 이 조작 모드는 JIT 컴파일러가 없어 높은 판별력을 제공하므로 샘플링 스레드나 컨텍스트 전환 성능의 저하가 일어나지 않습니다. Java 스펙을 준수하는 동안 Java 코드를 AOT(Ahead-of-Time) 컴파일하는 것이 어려운 현상은 AOT 컴파일된 코드가 해석보다는 많이 빠르지만 JIT 컴파일된 코드보다 느리게 수행됨을 의미합니다.

WebSphere Real Time for RT Linux V2 및 후속 버전은 AOT 코드를 IBM Java 6 JVM에 제공되는 공유 클래스 기술을 사용하여 JXE 파일이 아닌 공유 클래스 캐시에 저장합니다. admincache 도구를 사용하면 캐시의 컨텐츠를 조회하고 모든 기존 캐시를 나열하며 캐시를 클래스 및 AOT 코드로 채울 수 있습니다. AOT 컴파일된 코드를 저장하면 애플리케이션 jar 파일이 수정되지 않고 애플리케이션을 실행할 때 클래스 경로 변경이 필요하지 않다는 이점이 있습니다.

공유 클래스 캐시는 사용 가능한 가상 주소 공간을 기반으로 실제 크기 한계를 가집니다. 즉, 모든 jar 파일에 대한 AOT 컴파일이 실질적이지 않습니다. 선택적으로 AOT 컴파일을 수행해야 합니다.

애플리케이션이 공유 클래스 캐시의 AOT 코드를 사용하여 실행될 경우, 클래스를 JVM에 로드할 때 클래스 로드 메소드에 대한 AOT 코드가 자동으로 로드됩니다. 메소드에 대한 AOT 코드를 설치하기 위해 클래스를 로드할 때 드

는 추가 비용 때문에 애플리케이션에서 성능에 중요한 부분을 실행하기 전에 가능하면 많은 클래스를 사전 로드하는 것이 중요합니다.

AOT 사전 로드된 코드를 사용하면 성능이 향상되고 판별력이 극대화됩니다. **-Xshareclasses** 및 **-Xaot** 옵션을 지정하여 애플리케이션을 실행할 때 AOT 코드를 사용할 수 있습니다. **-Xaot** 옵션은 기본적으로 켜져 있습니다.

admincache 도구를 사용하여 공유 클래스 캐시에 AOT 코드를 저장하고 사용하려면 44 페이지의 『admincache 도구 사용』의 내용을 참조하십시오. jxeinajar에서 admincache로 마이그레이션하는 데 관한 정보는 WebSphere Real Time for RT Linux V2 문서에 있습니다.

AOT 컴파일된 코드로 애플리케이션을 실행하는 예제는 94 페이지의 『AOT를 사용하는 동안 샘플 애플리케이션 실행』의 내용을 참조하십시오.

AOT 사전 컴파일된 코드와 우선순위가 낮은 JIT 컴파일을 결합한 혼합 모드

애플리케이션을 실행하는 동안 AOT 및 JIT 컴파일된 코드를 함께 사용할 수 있습니다. 이 조작 모드는 성능 향상, 특히 자주 실행하는 메소드의 성능 향상과 판별력 극대화를 제공합니다. 이 모드의 주요 이점은 AOT 사전 컴파일을 사용하여 애플리케이션의 가장 중요한 부분이 AOT 또는 JIT 컴파일된 코드보다 대개 훨씬 느린 인터프리터에서 실행되지 않도록 보장한다는 점입니다. JIT 컴파일러가 애플리케이션 성능에 큰 지장을 주지 않으면서 자주 실행되는 해석된 메소드를 동적으로 식별할 수 있으므로 모든 메소드를 사전 컴파일할 필요가 없습니다. **-Xshareclasses** 옵션을 명령행에 추가할 경우 혼합 모드가 기본 모드입니다.

혼합 AOT 및 JIT 컴파일로 애플리케이션을 실행하려면 94 페이지의 『AOT를 사용하는 동안 샘플 애플리케이션 실행』의 내용을 참조하십시오.

명시적으로 컴파일 관리

JIT 컴파일러가 사용되는 컴파일 모드에서 java.lang.Compiler API를 사용하여 JIT 컴파일러 조작을 명시적으로 제어할 수 있습니다. JIT 컴파일러는 compileClass() 메소드를 사용하여 전달된 클래스의 메소드를 컴파일합니다. compileClass()가 동기적이므로 제공된 메소드가 컴파일될 때까지 리턴되지 않습니다. 애플리케이션은 애플리케이션 런타임의 주요 단계에 사용되는 클래스를 반복하여 초기화 단계에서 compileClass()를 사용할 수 있습니다. 초기화 단계가 완료되면 Compiler.disable() 메소드를 호출하여 컴파일 및 샘플링 스레드를 전부 사용 불가능하게 하십시오. 애플리케이션 개발 동안 애플리케이션 초기화 단계에서 로드 및 컴파일할 클래스 목록을 관리하는 문제가 이 기술의 기본적인 어려움으로 남아 있습니다.

애플리케이션에서 컴파일 관리에 대한 자세한 정보는 Java용 IBM 실시간 클래스 분석 도구를 참조하십시오.

컴파일 명령행 옵션 개요

-Xjit 옵션을 사용하여 JIT를 설정한 채로 또는 **-Xnojit** 옵션을 사용하여 JIT를 설정하지 않고 애플리케이션을 실행할 수 있습니다. **-Xjit**가 기본 모드입니다.

-Xshareclasses -Xaot 옵션을 사용하면 AOT 코드를 사용하여 애플리케이션을 실행할 수 있습니다. AOT 코드를 사용 불가능하게 하려면 **-Xnoaot** 옵션을 사용하십시오. **-Xaot**가 기본 옵션이지만, AOT 코드를 공유 클래스 캐시에 저장해야 하므로 **-Xshareclasses** 옵션도 함께 지정하지 않는 한 아무런 영향을 주지 않습니다.

AOT 컴파일러 사용

이 단계를 사용하여 Java 코드를 사전 컴파일하십시오. 이 프로시저는 **javac** 명령에 **-Xrealtime** 옵션 사용, **admincache** 도구 사용 및 **java** 명령에 **-Xrealtime** 및 **-Xnojit** 옵션 사용에 대해 설명합니다.

이 태스크 정보

AOT(Ahead-of-Time) 컴파일러를 사용하면 컴파일이 애플리케이션 런타임과 구분됨을 의미합니다. 또한, 자주 사용하는 메소드만이 아니라 더 많은 메소드를 동시에 컴파일할 수 있습니다. 다음 단계에 표시된 대로 애플리케이션의 모든 항목 또는 개별 클래스만 컴파일할 수 있습니다.

주: 공유 클래스 캐시를 사용할 경우 캐시 이름이 53자를 초과하지 않아야 합니다.

프로시저

1. 쉘 프롬프트에서 다음을 입력하십시오.

   ```
   javac -Xrealtime source
   ```

   이 명령은 실시간 환경에서 사용하기 위해 소스에서 Java 바이트 코드를 작성합니다. 9 페이지의 그림 2의 내용을 참조하십시오.

2. 생성된 클래스 파일을 jar 파일로 패키징하십시오. 예를 들어, test.jar을 작성하려면 다음을 실행하십시오.

   ```
   jar cvf test.jar source
   ```

3. 쉘 프롬프트에서 다음을 입력하십시오.

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

   이 명령은 test.jar 파일을 사전 컴파일하고 출력을 출력 디렉토리 **./aot**에 기록합니다.

4. 쉘 프롬프트에서 다음을 입력하십시오. 공유 클래스 캐시에서 AOT 코드를 사용하여 파일을 실행하려면 쉘 프롬프트에서 다음을 입력하십시오.

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

공유 클래스 캐시에서 AOT 코드를 사용하여 파일을 실행하려면 자주 호출하는 메소드를 사전 컴파일하고 새 jar 파일 없이 쉘 프롬프트에서 다음을 입력하십시오.

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

이 명령은 3단계에서 사전 컴파일한 동일한 jar 파일을 사용합니다.

admincache 도구 사용

admincache 도구는 워크스테이션에서 공유 클래스 캐시를 관리하는 데 사용됩니다.

IBM WebSphere Real Time for RT Linux 제품에서 admincache 도구를 사용하여 클래스 및 AOT 컴파일된 코드가 들어 있는 공유 클래스 캐시를 작성할 수 있습니다. 캐시를 작성한 후 기존 캐시를 검사하는 데에도 이 도구를 사용할 수 있습니다.

공유 클래스 캐시는 다양한 JVM 시나리오에서 메모리 풋프린트를 줄이고 애플리케이션 시작 속도를 높이는 데 사용됩니다.

WebSphere Real Time for RT Linux에서 실시간 및 실시간 이외 모드로 공유 클래스 캐시를 사용할 수 있지만 캐시 형식, 작성 및 채우기 기술이 서로 다릅니다. 실시간 모드 캐시는 실시간 이외 모드 캐시와 호환되지 않습니다. 실시간 이외 모드에서는 표준 JVM과 동일한 방식으로 캐시를 작성하고 채웁니다. 이는 애플리케이션을 실행할 때 사용자에게 투명하게 JVM에서 캐시를 작성하고 채움을 의미합니다. 실시간 모드에서 **-Xrealtime** 옵션을 사용하면 admincache에서 **-populate** 옵션을 사용하여 공유 클래스 캐시를 작성하고 사전에 채워야 합니다. 실시간 모드에서 실행되는 애플리케이션은 사전 채워진 캐시의 컨텐츠를 읽지만 컨텐츠를 수정할 수는 없습니다.

실시간 모드에서 작성한 공유 클래스 캐시는 애플리케이션을 실시간 모드로 실행할 경우에만 사용할 수 있습니다. 실시간 이외 모드에서 작성한 공유 클래스 캐시는 애플리케이션을 실시간 모드 이외 모드에서 실행할 경우에만 사용할 수 있습니다. 이는 admincache 도구에도 적용됩니다. JVM이 작성한 캐시를 실시간 모드로 관리하려면 admincache를 **-Xrealtime** 옵션과 함께 사용하십시오. JVM이 작성한 캐시를 실시간 이외 모드로 관리하려면 **-Xrealtime** 옵션을 사용하지 마십시오. 런타임에 공유 클래스 캐시에 연결하려면 **-Xshareclasses** 옵션을 명령행에 추가하십시오.

워크스테이션에서 특정 이름을 사용하여 지정된 디렉토리에 여러 공유 클래스 캐시를 작성할 수 있습니다. 새 캐시를 작성할 때는 **-cacheName** <*name*> 옵션으로 캐시의 이름을 지정할 수 있습니다. 캐시 이름이 53자를 초과하지 않아야 합니다.

기본적으로 공유 클래스 캐시가 `/tmp/javasharedresources` 디렉토리에 작성되지만 이 위치는 **-cacheDir** <*directory*> 옵션을 지정하여 대체할 수 있습니다. 공유 클래스 캐시의 내부 형식은 작성하는 워크스테이션의 특성에 따라 다릅니다. 즉, 안전 조치로 공유 클래스 캐시를 네트워크로 연결된 드라이브에 작성할 수 없음을 의미합니다. 이렇게 제한하는 다른 이유는 네트워크로 연결된 파일 시스템에서 공유 클래스 캐시에 액세스할 때 성능이 예측 불가능하고 느려질 수 있기 때문입니다.

명령행에 캐시 이름을 지정하지 않은 경우 기본값이 **sharedcc_<user_login>**입니다.

실시간 이외 모드에서 공유 캐시 조작에 대한 자세한 정보는 101 페이지의 『실시간 이외 모드에서 JVM 사이의 클래스 데이터 공유』의 내용을 참조하십시오.

주: IBM WebSphere Real Time for RT Linux V2 SR1 이상에서는 **-populate** 옵션과 함께 **-classpath** 옵션을 사용해야 합니다.

실시간 공유 클래스 캐시 작성:

admincache 도구를 사용하여 실시간 모드에서 액세스 가능한 공유 클래스 캐시를 작성합니다.

주: 기본 설정으로 공유 클래스 캐시 파일을 작성할 때 보안 고려사항을 숙지하고 있어야 합니다. 공유 클래스 캐시의 보안 고려사항 및 기본 권한 변경에 대한 자세한 정보는 103 페이지의 『공유 클래스 캐시의 보안 고려사항』의 내용을 참조하십시오.

공유 클래스 캐시를 작성할 때는 admincache 도구 **-populate** 옵션을 사용합니다. jar 파일 목록, 디렉토리 또는 디렉토리 트리 조합에서 이 옵션을 사용하여 jar 파일을 검색할 수 있습니다. 지정되거나 찾은 각 jar 파일에 대해 admincache가 jar 파일의 각 클래스를 공유 클래스 캐시에 저장합니다. 또한 클래스 메소드는 **-noaot** 옵션을 지정하지 않는 한, AOT 컴파일되고 공유 클래스 캐시에 저장됩니다.

-classpath 옵션은 **-populate**와 함께 사용해야 합니다. 그렇지 않으면 다음과 같은 오류 메시지가 나타납니다.

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

admincache **-help** 옵션은 admincache가 캐시를 채우는 방식을 제어하기 위해 사용하는 하위 옵션을 나열합니다.

```
$ admincache -Xrealtime -help
Usage: admincache [option]*
  where [option] can be:
     -help | -?                Action: show this help
     -Xrealtime                use in real time environment
     -cacheName <name>         specify name of shared cache (Use %%u to substitute username)
     -cacheDir <dir>           set the location of the JVM cache files
     -listAllCaches            Action: list all existing shared class caches
     -printStats               Action: print cache statistics
     -printAllStats            Action: print more verbose cache statistics
     -destroy                  Action: destroy the named (or default) cache
     -destroyAll               Action: destroy all caches
     -populate                 Action: Create a new cache and populate it
       -searchPath <path>      specify the directory in which to find files if no files specified (default i
                               only one -searchPath option can be specified
            -classpath <class path> specify the classpath that will be used at runtime to access this cache
                               the -classpath option is required
       -[no]recurse            [do not] recurse into subdirectories to find files to convert
                               (기본 반복하지 않음)
```

```
       -[no]grow               if specified cache exists already, [do not] add to it (default no grow)
                               if -grow is not selected, specified cache will be removed if present
       -verbose                print out progress messages for each jar
       -noisy                  print out progress messages for each class in each jar
       -quiet                  suppress all output
       -[no]aot                also perform AOT compilation on methods after storing classes into cache
       -aotFilter <signature>  only matching methods will be AOT compiled and stored into cache
          e.g. -aotFilter {mypackage/myclass.mymethod(I)I} compiles only mymethod(I)I
          e.g. -aotFilter {mypackage/myclass.mymethod*} compiles any mymethod
          e.g. -aotFilter {mypackage/myclass.*} compiles all methods from myclass
       -aotFilterFile <file>   only methods matching those in file will be AOT compiled and stored into
                               cache (input file must have been created by -Xjit:verbose={precompile},
                               vlog=<file>)
       -printvmargs            print VM arguments needed to access populated cache at runtime
       [jar file]*.[jar][zip]  explicit list of jar files to populate into cache
                               if no files are specified, all files.[jar][zip] in the searchPath will be conver
Exactly one action option must be specified
```

주: 공유 클래스 캐시를 사용할 경우 **-cacheName** 옵션이 지정하는 이름이 53자를 초과하지 않아야 합니다.

jar 파일 목록을 지정할 수 있으며, 이 경우 jar 파일의 클래스만 공유 클래스 캐시에 추가됩니다. jar 파일 목록을 지정하지 않으면 **-searchPath <path>** 옵션을 사용하여 .jar 또는 .zip 파일을 검색할 디렉토리 트리를 지정하십시오. 기본값은 디렉토리 트리에서 .jar 또는 .zip 파일을 반복적으로 검색하는 **-recurse** 옵션입니다. **-norecurse** 옵션은 지정된 디렉토리만 검색함을 의미합니다. admincache가 지정된 jar 파일을 처리하는 데 필요한 모든 클래스를 찾을 수 있도록 **-classpath <class path>** 옵션을 지정하십시오. 공유 클래스 캐시를 채우는 과정에서 클래스가 JVM에 로드됩니다. 따라서 jar 파일에서 클래스를 로드할 때 admincache가 모든 참조된 클래스 및 수퍼클래스를 찾을 수 있다는 사실은 중요합니다.

-grow 옵션은 캐시 디렉토리에 이름이 동일한 기존의 공유 클래스 캐시가 있는 경우 새 jar 파일을 기존 캐시 컨텐츠에 추가하도록 지정합니다. **-nogrow** 옵션은 이전 캐시 디렉토리에 이름이 동일한 기존의 공유 클래스 캐시가 있는 경우 새 jar 파일이 이존 캐시 컨텐츠를 대체하도록 지정합니다. **-grow** 옵션은 변경된 클래스를 대체하지 않고 공유 클래스 캐시에 현재 존재하지 않는 새 jar 파일을 추가하는 데 사용합니다. 캐시에 이미 있지만 애플리케이션 수정사항 때문에 변경된 클래스를 업데이트하기 위해 **-grow** 옵션을 사용하지는 마십시오. 기존 클래스를 업데이트하려면 현재 클래스 컨텐츠로 완전히 새로운 캐시를 작성하십시오. 클래스를 변경할 때 공유 클래스 캐시를 업데이트하지 않은 경우 애플리케이션이 새 클래스 컨텐츠로 제대로 실행되지만 공유 클래스 캐시를 이용하지 않습니다. 이유는 변경된 클래스는 공유 클래스 캐시가 아닌 디스크에서 로드되기 때문입니다. 디스크에서 클래스를 로드하는 것은 AOT 컴파일된 코드를 해당 클래스에 사용할 수 없음을 의미합니다. 클래스를 변경할 때 공유 클래스 캐시를 재생성하십시오.

admincache에서 제공하는 세부사항 레벨을 제어하려면 **-quiet**, **-verbose** 및 **-noisy** 옵션을 사용하십시오.

공유 클래스 캐시를 채우는 클래스에서 메소드에 대한 AOT(Ahead-Of-Time) 사전 컴파일을 지정하려면 **-aot** 옵션을 사용하십시오. AOT 사전 컴파일을 방지하고 클래스를 공유 클래스 캐시에만 저장하려면 **-noaot** 옵션을 사용하십시오. **-aot** 옵션이 기본 설정입니다.

일부 메소드를 선별적으로 사전 컴파일하려면 **-aotFilter** <*signature*> 또는 **-aotFilterFile** <*file*> 옵션을 사용하십시오. <*signature*>는 메소드 서명을 위한 간결한 정규식으로, 중괄호로 묶으며 '*'는 일련의 문자로 대체할 수 있습니다. 쉘이 메소드 서명의 문자를 해석하지 않도록 <*signature*>를 작은따옴표로 묶어야 할 수도 있습니다.

표 4은 <*signature*> 옵션의 몇 가지 예제를 보여줍니다.

표 4. <signature> 옵션 예제

서명	의미
`-aotFilter '{java/lang/*}'`	java/lang 패키지의 메소드를 AOT 컴파일합니다.
`-aotFilter '{*.sample*}'`	"sample"로 시작되는 메소드를 AOT 컴파일합니다.
`-aotFilter '{mypackage/myclass.mymethod(I)I}'`	이 정확한 서명을 포함한 메소드를 AOT 컴파일합니다.

-aotFilterFile <*file*> 옵션은 <*file*>의 컨텐츠를 사용하여 AOT 컴파일할 메소드를 선택합니다. 다른 메소드는 AOT 컴파일되지 않습니다. **-Xjit:verbose={precompile},vlog=**<*file*> 옵션을 사용하여 애플리케이션을 초기 실행하는 동안 <*file*>의 컨텐츠가 생성됩니다. <*file*>에 저장되는 상세 출력은 내부 형식을 사용합니다. 이 형식은 **-aotFilterFile** 옵션에 필요합니다.

주: -vlog=<*file*> 옵션은 "file"이라는 파일을 직접 생성하지 않습니다. 상세 출력이 생성될 때 날짜 및 프로세스 ID 문자열이 "file"에 추가됩니다. **-Xjit:verbose={precompile},vlog=my_file** 옵션을 지정하면 생성된 파일 이름이 `my_file.<date>.<#>.<process id>`와 비슷합니다. 추가 필드를 통해 한 개 특정 JVM에 명령행 옵션을 제공하거나 다른 JVM과 다른 **-Xjit** 명령행 옵션을 사용하기 어려운 다중 JVM 시나리오에서 개별 상세 로그 파일을 손쉽게 생성할 수 있습니다. 단일 JVM 시나리오에서는 이 번호가 명령행에 제공되는 파일 이름에 추가됩니다.

생성된 파일을 편집할 필요 없이 **-aotFilterFile** 옵션과 함께 사용할 수 있습니다. **-Xjit:verbose={precompile},vlog=<file>** 옵션을 사용하여 실행한 몇 개의 애플리케이션에서 생성한 여러 상세 로그 파일은 잘릴 수 있으며 **-aotFilterFile** 옵션을 사용하여 admincache에 제공합니다.

-printvmargs 옵션은 애플리케이션을 실행할 때 명령행에 올바른 인수가 제공되는지 확인하는 데 유용합니다.

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

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
```

이 예제에서 출력의 마지막 행은 공유 클래스 캐시에 저장된 클래스 및 AOT 메소드가 사용되도록 애플리케이션을 실행할 때 명령행에 추가해야 하는 옵션을 보여줍니다. 이 예제의 옵션을 사용하려면 다음 명령을 입력하십시오.

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

admincache를 사용하여 공유 클래스 캐시 관리:

admincache 도구는 시스템에서 공유 클래스 캐시를 관리하는 몇 가지 유틸리티를 포함하고 있습니다.

admincache 도구는 몇 가지 활동에 도움이 되는 유틸리티를 제공합니다.

- 캐시에 있는 공유 클래스 캐시 나열
- 공유 클래스 캐시 컨텐츠에 대한 세부사항 제공
- 특정 캐시 디렉토리에 있는 캐시의 일부 또는 전부 제거

사용 가능한 공유 클래스 캐시 나열:

admincache 도구는 캐시에 있는 공유 클래스 캐시 목록을 제공합니다.

캐시에 있는 모든 공유 클래스 캐시 목록을 구하려면 **-listAllCaches** 옵션을 사용하고, **-cacheDir** 옵션을 사용하여 캐시 디렉토리를 지정하십시오.

```
$ admincache -Xrealtime -listAllCaches

admincache 1.02

Listing all caches in cacheDir /tmp/javasharedresources/

Cache name              level         persistent  last detach time

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

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

이 예제에서 기본 캐시 디렉토리에 다음과 같은 호환 가능한 공유 클래스 캐시가 두 개 있습니다.

- 로그인 *username*을 가진 사용자의 기본 캐시
- *rtCache*라는 다른 캐시

*nonrtCache*라는 호환 불가능한 캐시도 예제에 있습니다. *nonrtCache*는 실시간 이외 모드로 실행하는 동안 JVM에서 작성한 것입니다. 이는 **-Xrealtime** 옵션을 사용하여 액세스할 수 없음을 의미합니다.

실시간 모드 JVM은 실시간 이외 모드로 작성된 캐시를 볼 수 있습니다. 실시간 이외 모드의 JVM은 실시간 모드로 작성된 캐시를 볼 수 없습니다.

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

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


Listing all caches in cacheDir /tmp/javasharedresources/

Cache name              level         persistent  last detach time

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

이 예제에서는 *nonrtCache*가 나열되고 **-Xrealtime**이 지정되지 않았기 때문에 호환 가능으로 표시됩니다.

공유 클래스 캐시의 컨텐츠 검사:

admincache 도구는 공유 클래스 캐시 컨텐츠를 설명합니다.

admincache 도구 **-printStats** 옵션을 사용하여 공유 클래스 캐시의 주요 컨텐츠에 대한 설명이 있는 개요를 볼 수 있습니다. 특정 캐시에 대한 정보는 지정된 캐시 디렉토리에서 **-cacheName** 및 **-cacheDir** 옵션을 사용하십시오. 다음 예제는 기본 캐시 디렉토리에 있는 *nonrtCache* 캐시에 대한 정보를 제공합니다.

```
$ 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
```

주: 공유 클래스 캐시를 사용할 경우 캐시 이름이 53자를 초과하지 않아야 합니다.

다음은 이 캐시에 대한 몇 가지 유용한 정보입니다.

- 캐시 *크기*는 `cache size = 16776852`로 표시됩니다.
- 캐시에서 사용 가능한 공간은 `free bytes = 14070360`으로 표시됩니다. 캐시가 대략 16% 찬 것을 계산할 수 있습니다.
- 캐시에 저장된 클래스의 수는 `# ROMClasses = 372`로 표시됩니다.
- 캐시에 저장된 AOT 메소드의 수는 `# AOT Methods = 981`로 표시됩니다.

admincache 도구의 **-printStats** 옵션이 제공하는 정보에 대한 세부사항은 printStats 유틸리티를 참조하십시오.

-printAllStats 옵션은 공유 클래스 캐시 컨텐츠에 대한 자세한 설명을 제공합니다. 캐시에 저장된 클래스 및 AOT 메소드 목록도 이 정보에 포함됩니다. **-printAllStats** 옵션의 출력은 상세합니다.

캐시에 있는 클래스는 다음과 같은 행으로 표시됩니다.

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

이 행은 java/lang/ClassLoader 클래스가 캐시에 있음을 보여줍니다. 주소는 공유 클래스 캐시에 내부적이며 진단 목적을 제외하고 자주 사용되지 않습니다.

캐시에 포함된 AOT 메소드는 다음과 같은 행으로 표시됩니다.

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

이 행은 java/lang/ClassLoader 클래스의 callerClassLoader 메소드가 캐시에 있음을 나타냅니다. 나열된 주소는 내부 공유 캐시 주소입니다. 서명이 매개변수 유형과 리턴 유형으로 구성된 경우 **-printAllStats** 옵션의 출력에 캐시에 있는 각 AOT 메소드에 대한 서명이 표시되지 않습니다.

admincache 도구의 **-printAllStats** 옵션이 제공하는 정보의 세부사항은 printAllStats 유틸리티를 참조하십시오.

공유 클래스 캐시 제거:

admincache 도구에는 지정된 캐시 디렉토리에서 특정 캐시 또는 모든 캐시를 지우는 옵션이 있습니다.

admincache 도구 **-destroy** 옵션은 특정 캐시 디렉토리에서 캐시를 지우는 데 사용합니다(사용자에게 관련 권한이 있는 경우). **-destroyAll** 옵션은 모든 캐시를 지우는 데 사용합니다(사용자에게 관련 권한이 있는 경우). 예를 들면 다음과 같습니다.

```
$ admincache -Xrealtime -destroy

admincache 1.02

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

캐시를 지운 후 기본 캐시 디렉토리의 사용 가능 공유 클래스 캐시 목록에서 지워진 캐시는 더 이상 표시되지 않음을 알 수 있습니다.

```
$ admincache -Xrealtime -listAllCaches

admincache 1.02

Listing all caches in cacheDir /tmp/javasharedresources/

Cache name              level         persistent  last detach time

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

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

-destroyAll 옵션은 현재 JVM과 호환 가능한지 여부에 상관없이 지정된 캐시 디렉토리에서 모든 캐시를 제거합니다. **-destroyAll** 옵션을 사용할 때는 주의해야 합니다.

```
$ admincache -Xrealtime -destroyAll

admincache 1.02

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

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

결과적으로, 시스템에 사용할 수 있는 공유 클래스 캐시가 더 이상 없습니다.

```
$ admincache -Xrealtime -listAllCaches

admincache 1.02

JVMSHRC005I No shared class caches available
```

현재 사용자에게 캐시에 액세스할 권한이 없으면 **-destroy** 또는 **-destroyAll** 옵션으로 캐시를 제거할 수 없습니다.

공유 클래스 캐시의 실제 크기:

admincache 도구는 공유 클래스 캐시의 *크기*를 조정하는 정보를 제공합니다.

소형 애플리케이션의 경우, 캐시 *크기*가 너무 커지지 않으면서 공유 클래스 캐시를 모든 클래스 및 메소드로 채울 수 있습니다. 대형 애플리케이션의 경우, 실제 사용할 때 공유 클래스 캐시의 *크기*가 너무 커질 수 있습니다. 이는 JVM 프로세스에 공유 클래스 캐시의 전체 컨텐츠를 처리할 만큼 충분한 가상 주소 공간이 있어야 하기 때문입니다. 공유 클래스 캐시 기술을 사용할 때 적용하는 몇 가지 고려사항이 있습니다.

공유 클래스 캐시는 연결하는 모든 JVM에서 가상으로 처리할 수 있어야 합니다. 이는 700MB보다 더 큰 공유 클래스 캐시를 사용하지 않아야 함을 의미합니다. admincache 도구가 캐시의 *크기*를 예측할 수 있습니다. 캐시가 700MB 한계보다 *크다*고 도구에 표시되면 소수의 클래스를 저장하거나 캐시에 저장되는 AOT 메소드에 대해 보다 *까다로운* 조건을 적용하도록 조언하는 메시지가 표시됩니다.

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

admincache 1.02

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

admincache 도구는 채우기 위해 지정하거나 찾은 jar 파일의 총 *크기*에 따라 보수적으로 캐시 *크기*를 예측합니다. 이는 jar 파일에 클래스 파일이 아닌 파일이 많으면 예측이 정확하지 않을 수 있음을 의미합니다. 캐시 *크기*를 보다 정확히 예측하려면 클래스 파일만 있는 임시 버전의 jar 파일을 작성하십시오. admincache 도구가 경고 메시지를 계속 생성하면 **-aotFilter** <*pattern*> 또는 **-aotFilterFile** <*file*> 옵션을 사용하여 jar 파일의 메소드를 보다 *까다롭게* AOT 사전 컴파일하는 것을 고려합니다. admincache 도구 메시지가 예측할 때 이 옵션에 의해 필터링된 AOT 메소드를 고려하지 않았다고 알려줍니다.

경고 메시지를 대체하고 캐시 채우기 단계로 진행하려면 표시된 **-Xscmx** 옵션을 admincache 명령행에 추가하십시오. 예측 *크기*가 너무 *크면* admincache 도구가 필요한 *크기*의 공유 클래스 캐시를 작성하지 못할 수도 있습니다. 이 문제를 해결하려면 admincache 도구가 처리할 수 있을 때까지 캐시 *크기*를 줄이십시오.

최종 캐시가 디스크에 작성될 때는 지정된 클래스 및 AOT 메소드를 저장하는 데 필요한 만큼의 *크기*입니다. 즉, *초기 캐시 크기*를 크게 지정하는 것은 문제가 되지 않습니다.

공유 클래스 캐시에 SDK 클래스 저장:

모든 애플리케이션에서 SDK의 모든 jar 파일이 있는 캐시를 작성할 필요는 없습니다.

SDK jar 파일의 *크기*와 개수는 모든 jar 파일을 포함한 캐시를 작성하려고 하면 캐시가 너무 커질 수 있다는 경고 메시지가 나타남을 의미합니다. 많은 애플리케이션에서 SDK jar 파일의 대부분은 참조하지 않습니다.

기본 SDK jar 파일은 `SDK/jre/lib` 디렉토리에 있습니다. 대부분의 애플리케이션에서 가장 중요한 jar 파일은 Java 6 릴리스에 새로 제공되는 `rt.jar`입니다. `rt.jar`은 Java 6 릴리스 이전에 별도의 jar 파일에 이전에 저장한 클래스 콜렉션입니다. 공유 클래스 캐시를 `rt.jar`로만 채우고 AOT 컴파일러를 사용하여 모든 메소드를 컴파일하면 대략 300MB *크기*의 캐시가 작성됩니다. 일반 애플리케이션에서는 `rt.jar` 클래스 메소드의 대부분을 참조하지 않습니다. `rt.jar`로 공유 클래스 캐시를 채우려면 다음을 수행하십시오.

1. `rt.jar`의 클래스만 공유 클래스 캐시에 채우십시오. 이렇게 하면 캐시에서 대략 50MB의 공간이 사용됩니다.
2. 프로그램이 사용하는 메소드만 컴파일하려면 **-aotFilterFile** <*file*> 옵션을 사용하십시오. 애플리케이션을 실행하여 <*file*>을 생성할 수 있습니다.

SDK에 다음과 같은 기타 자주 사용하는 중요한 jar 파일이 있습니다.

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

`realtime.jar`은 Java용 Real Time Specification(RTSJ)의 IBM 구현을 포함합니다. 애플리케이션이 RTSJ의 기능을 사용할 경우, 확실한 성능 향상을 위해 공유 클래스 캐시에 `realtime.jar` 파일을 저장하십시오. `vm.jar`은 모든 애플리케이션에 자주 사용되는 몇 가지 내부 JVM 클래스를 포함하고 있습니다. `java.util.jar`은 몇 가지 컨테이너 클래스를 포함하며 확실한 성능 향상을 위해 모든 애플리케이션의 공유 클래스 캐시에 저장해야 합니다.

애플리케이션이 해당 클래스를 사용할 경우 `sdk/jre/lib` 및 `sdk/jre/lib/ext` 디렉토리의 기타 jar 파일을 공유 클래스 캐시에 저장할 수 있습니다. 애플리케이션이 이 클래스를 사용하는지 여부를 확인하는 가장 간단한 방법은 프로그램을 실행할 때 **-verbose:dynload** 옵션을 사용하는 것입니다. **-verbose:dynload** 옵션은 현재 애플리케이션 실행에서 로드된 클래스만 설명합니다. 예를 들면 다음과 같습니다.

```
<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>
```

이 예제 출력은 두 가지 SDK jar 파일에서 로드된 3개의 클래스를 보여줍니다. java/io/InputStreamReader 클래스는 `rt.jar`에서 로드된 것입니다. java/util/LinkedHashSet 및 java/util/HashSet 클래스는 `java.util.jar`에서 로드된 것입니다.

기타 admincache 고려사항:

admincache로 작업 시 유용한 정보

캐시 채우기 및 영구 메모리 *크기* 조정

admincache 도구가 공유 클래스 캐시를 실시간 모드에서 채울 때는 채우기 프로세스를 수행하는 동안 각 클래스를 로드해야 합니다. 각 클래스가 일부 영구 메모리를 이용하므로 기본 영구 메모리 *크기*가 요청된 모든 클래스에 충분하지 않을 수 있습니다. 캐시를 많은 클래스로 채우는 동안 **admincache** 도구가 OutOfMemory 오류를 발생시키면 **-Xgc:immortalMemorySize=32M** 옵션을 사용하여 영구 메모리 *크기*를 기본값 16MB보다 *크게* 늘리십시오.

클래스 변경 시

디스크에서 클래스 파일을 변경하면 공유 클래스 캐시 기술이 공유 클래스 캐시에서 해당 클래스의 캐싱된 버전을 사용하지 않아야 함을 자동으로 감지합니다. 프로그램이 올바르게 작동하지만 공유 클래스 캐시를 전부 이용할 수 없고 해당 클래스의 AOT 메소드가 사용되지 않습니다. 애플리케이션에서 클래스를 변경한 경우, 공유 클래스 캐시를 다시 작성하십시오. 수정된 클래스가 포함된 jar 파일만 사전에 채우기 위해 **-grow** 옵션을 사용하지 마십시오. 이 옵션은 jar 파일이 이미 캐시에 있는 시나리오용으로 설계되지 않았습니다.

공유 캐시 관리

공유 캐시는 로드되는 파일이 없더라도 주소 공간이 필요합니다. 공유 클래스 캐시가 JVM 프로세스에서 메모리를 이용하는 방식에 대한 자세한 정보는 115 페이지의 『IBM JVM의 메모리 관리 방식』의 내용을 참조하십시오.

공유 클래스 캐시에 사전 컴파일된 jar 파일 저장

IBM에서 제공하는 Java 클래스의 전부 또는 일부를 저장하거나 공유 클래스 캐시에 포함할 수 있습니다. 이 프로세스는 **javac**와 함께 **-Xrealtime** 옵션 및 **admincache** 도구를 사용하여 클래스를 공유 클래스 캐시에 저장합니다.

시작하기 전에

AOT(Ahead-of-Time) 방식으로 공유 클래스 캐시에 저장된 Jar 파일은 **-Xrealtime** 옵션을 사용하고 **-Xrealtime** 옵션으로 java를 실행할 경우에만 지원됩니다. **-Xrealtime**의 사용 여부에 상관없이 동일한 jar 파일을 사용할 수 있지만, 캐시에 저장된 jar 파일은 **-Xrealtime**을 지정한 경우에만 사용할 수 있습니다.

주: 공유 클래스 캐시를 사용할 때는 캐시 이름이 53자를 초과하지 않아야 합니다.

이 태스크 정보

admincache 도구를 사용하여 jar 파일을 공유 클래스 캐시에 저장할 수 있습니다. **admincache**을 사용하면 3가지 방법 중 하나로 애플리케이션을 빌드할 수 있습니다.

주:

- Linux 시스템에 제한시간을 설정한 경우, 대형 jar 파일을 사전 컴파일할 때 대체해야 합니다. 그렇지 않으면 컴파일 제한시간이 초과되고 jar 파일이 작성되지 않습니다.

애플리케이션에서 모든 클래스 및 메소드 사전 컴파일:

이 프로시저는 애플리케이션의 모든 클래스를 사전 컴파일합니다. jar 파일 세트를 공유 클래스 캐시에 저장합니다. 해당 jar 파일에 있는 모든 클래스의 모든 메소드가 캐시에 저장됩니다. 최적화된 jar 파일에서 모든 메소드가 컴파일됩니다.

이 태스크 정보

이 예제를 위해 애플리케이션이 환경 변수 *$APP_HOME*에서 지정하는 디렉토리 아래에 있고 jar 파일이 서브디렉토리 *$APP_HOME*/lib에 있습니다. 또한 애플리케이션은 core.jarrt.jar에서 IBM이 제공하는 항목의 몇 가지 클래스를 사용합니다. 이 경우, 애플리케이션 코드만 사전 컴파일할 수 있습니다(main.jar 및 util.jar).

기본적으로, 공유 클래스 캐시는 /tmp/javasharedresources에 있습니다. 캐시를 다른 디렉토리에 두려면 **-cacheDir** 옵션을 사용하십시오. 네트워크로 연결된 파일 시스템에는 캐시를 작성할 수 없습니다.

프로시저

1. 쉘 프롬프트에서 다음을 입력하십시오. cd *$APP_HOME*

   여기서 *$APP_HOME*은 사용자 애플리케이션의 디렉토리입니다.

2. 쉘 프롬프트에서 cd *$APP_HOME*/lib를 입력하십시오. *$APP_HOME*/lib는 main.jar 및 util.jar이 저장된 디렉토리입니다.
3. 쉘 프롬프트에서 admincache -Xrealtime -populate -aot -classpath *$APP_HOME*/lib -searchPath *$APP_HOME*/lib -norecurse를 입력하십시오. . 이 프로시저에서는 *$APP_HOME*/lib에 있는 각 jar 파일을 최적화하고 진행상태 정보를 화면에 쓴 다음 *$APP_HOME*/aot 디렉토리에 새 jar 파일을 작성합니다. **-cacheName <name>**을 사용하여 캐시 이름을 지정할 수 있지만, 지정한 항목이 없으면 사용자 로그인에 기반한 기본 이름이 사용됩니다.

   주: -cacheName 옵션이 지정하는 이름은 53자를 초과하지 않아야 합니다.
4. 쉘 프롬프트에서 admincache -Xrealtime -listAllCaches를 입력하면 캐시의 존재가 표시됩니다.

다음에 수행할 작업

추가 옵션은 admincache -Xrealtime -help를 지정하십시오.

자주 사용하는 메소드 사전 컴파일:

프로파일에 따른 AOT 컴파일을 사용하여 애플리케이션이 자주 사용하는 메소드만 사전 컴파일할 수 있습니다. AOT 컴파일은 특수 옵션 **-Xjit:verbose={precompile},vlog=optFile**을 사용하여 실행하여 생성한 옵션 파일을 사용하여 jar 파일 세트를 공유 클래스 캐시에 저장합니다. 옵션 파일에 나열된 메소드만 사전 컴파일됩니다.

시작하기 전에

시작하기 전에 JIT 컴파일러가 일반적으로 컴파일하는 메소드 목록을 작성하십시오.

이 태스크 정보

-Xjit:verbose={precompile} 옵션이 생성하는 파일을 편집할 수 있습니다. 해당 파일은 사전 컴파일할 메소드의 명시적 스펙입니다. 이 메소드는 특정적입니다. 즉, 컴파일할 각 메소드에 대한 전체 서명을 포함하고 있으며, 이를 사용하여 com/acme/sample.myMethod(J)V를 컴파일할 수 있지만 com/acme/sample.myMethod(I)V는 컴파일하지 못합니다.

주: 공유 클래스 캐시를 사용할 경우 캐시 이름이 53자를 초과하지 않아야 합니다.

프로시저

1. 쉘 프롬프트에서 다음을 입력하십시오.

   cd *$APP_HOME*

   여기서 *$APP_HOME*은 사용자 애플리케이션의 디렉토리입니다.

2. 쉘 프롬프트에서 다음을 입력하십시오.

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

여기서, 각 인수는 다음을 의미합니다.

- *app.precompileOpts*는 JIT로 컴파일한 메소드를 보여주는 로그 파일의 이름입니다.
- *applicationName*은 사용자 애플리케이션의 이름입니다.

이 명령은 JIT를 사용하여 컴파일한 메소드 목록을 작성합니다.

3. 쉘 프롬프트에서 다음을 입력하십시오.

```
cd $APP_HOME/lib
```

*$APP_HOME/lib*는 사용자 애플리케이션의 jar 파일이 저장된 디렉토리입니다.

4. 모든 샘플 애플리케이션 메소드를 캐시에 컴파일하려면 다음을 입력하십시오.

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

5. realtime.jar 및 vm.jar을 캐시에 컴파일하려면 다음을 입력하십시오.

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

6. rt.jar을 캐시에 컴파일하려면 다음을 입력하십시오.

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

7. 이 명령을 테스트하려면 **-nojit** 옵션과 함께 애플리케이션을 실행하십시오. 캐시의 코드를 사용합니다. 쉘 프롬프트에서 다음을 입력하십시오.

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

여기서 *applicationName*은 사용자 애플리케이션의 이름입니다.

IBM이 제공하는 파일 사전 컴파일:

IBM에서 제공하는 파일(예: rt.jar)을 사전 컴파일하여 성능과 사전 예측 능력을 균형 있게 높일 수 있습니다.

이 태스크 정보

사전 컴파일은 애플리케이션 jar을 사전 컴파일하는 태스크와 비슷하지만 런타임에 추가 요구사항이 적용됩니다. JRE의 파일 대신 이 파일을 사용하려면 부팅 클래스 경로

가 올바르게 지정되었는지 확인해야 합니다. **-Xshareclasses** 옵션은 기본 클래스 경로 위치보다 지정된 클래스 캐시를 먼저 살펴보도록 JVM에 지시합니다.

주: 공유 클래스 캐시를 사용할 경우 캐시 이름이 53자를 초과하지 않아야 합니다.

애플리케이션과 함께 사용하기 위해 rt.jar 사전 컴파일:

프로시저

1. 쉘 프롬프트에서 다음을 입력하십시오. cd *$JAVA_HOME*/lib 여기서 *$JAVA_HOME*은 사용자의 Java 홈 디렉토리입니다.
2. **admincache** 도구를 실행하십시오. 쉘 프롬프트에서 다음을 입력하십시오.

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

   이 명령은 IBM 제공 파일 rt.jar을 사전 컴파일한 결과로 myCache라는 캐시를 채웁니다.
3. 해당 캐시 이름을 지정하도록 **-Xshareclasses** 옵션을 지정하여 애플리케이션을 실행하십시오. 애플리케이션을 실행하려면 다음을 입력하십시오.

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

JIT(just-in-time) 컴파일러

표준 SDK 클래스 라이브러리의 부분으로 제공되는 java.lang.Compiler 클래스를 사용하여 JIT 컴파일러가 작동되는 시기 및 방식을 제어할 수 있습니다. IBM은 Compile.compileClass(), Compiler.enable() 및 Compiler.disable() 메소드를 전부 지원합니다.

예를 들어, 애플리케이션을 준비하고 애플리케이션의 주요 메소드가 컴파일되었음을 알고 있는 경우 애플리케이션을 준비한 후 Compiler.disable() 메소드를 호출하여 나머지 애플리케이션 실행 동안 JIT 컴파일이 발생하지 않도록 할 수 있습니다.

다음 두 가지 방법으로 메소드 컴파일을 제어합니다.

- 컴파일할 메소드 세트를 지정합니다.

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

  여기서 <*method specification*>은 이 시점에 로드되고 컴파일할 모든 메소드 목록입니다. <*method specification*>은 완전한 메소드 이름을 설명합니다. 별표는 와일드카드 일치를 지시합니다.

  예를 들어, 이미 로드되고 java.lang.String으로 시작되는 모든 메소드를 컴파일하려면 다음을 지정합니다.

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

주: 이 명령은 java.lang.String 클래스뿐만 아니라 java.lang.StringBuffer 클래스의 메소드도 컴파일하는데, 이는 사용자가 원하는 사항이 아닐 수도 있습니다. java.lang.String 클래스의 메소드만 컴파일하려면 다음을 지정합니다.

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

- 이 스레드가 실행되고 계속 진행되기 전에 컴파일 큐의 모든 메소드가 컴파일되도록 지정하십시오.

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

  컴파일러를 사용 불가능하게 하기 전에 컴파일 큐가 비어 있는지 확인할 수도 있습니다. 메소드 및 클래스를 컴파일하는 일반 기술은 다음과 같습니다.

```
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 전이 동안 판별력

기본적으로 JIT는 고성능 Java 및 원시(J2N) JNI간 전이를 위해 최적화된 코드를 생성합니다. 다음 코드 시퀀스를 사용하여 원시 라이브러리를 다시 로드하면 판별력이 줄어들 수 있습니다.

```
RegisterNatives / UnregisterNatives / RegisterNatives
```

느리지만 보다 결정적인 코드로 되돌리려면 명령행 옵션 **-Xjit:disableDirectToJNI**를 사용하십시오.

JIT 사용

몇 가지 방법으로 JIT를 사용 가능하게 설정할 수 있습니다. 두 명령행 옵션 모두 **JAVA_COMPILER** 환경 변수를 재지정합니다.

프로시저

- Java 애플리케이션을 실행하기 전에 **JAVA_COMPILER** 환경 변수를 "jitc"로 설정하십시오. 쉘 프롬프트에서 다음을 입력하십시오.
  - **Korn 쉘의 경우:** `export JAVA_COMPILER=jitc`

    주: Korn 쉘 명령은 따로 언급이 없으면 이 정보에 사용됩니다.
  - **Bourne 쉘의 경우:**

    ```
    JAVA_COMPILER=jitc
    export JAVA_COMPILER
    ```
  - **C 쉘의 경우:** `setenv JAVA_COMPILER jitc`

  JAVA_COMPILER 환경 변수가 빈 문자열인 경우 JIT는 그대로 사용되지 않습니다. 환경 변수를 사용하지 않으려면 쉘 프롬프트에서 `unset` **JAVA_COMPILER**를 입력하십시오.

- JVM 명령행에서 **-D** 옵션을 사용하여 java.compiler 특성을 "jitc"로 설정하십시오. 쉘 프롬프트에서 `java -Djava.compiler=jitc <MyApp>`를 입력하십시오.
- JVM 명령행에서 **-Xjit** 옵션을 사용하십시오. **-Xint** 옵션을 동시에 지정하지 *않아야* 합니다. 쉘 프롬프트에서 `java -Xjit <MyApp>`를 입력하십시오.

JIT 사용 안함

몇 가지 방법으로 JIT를 사용 불가능하게 설정할 수 있습니다. 두 명령행 옵션 모두 **JAVA_COMPILER** 환경 변수를 재지정합니다.

이 태스크 정보

프로시저

- Java 애플리케이션을 실행하기 전에 **JAVA_COMPILER** 환경 변수를 "NONE" 또는 빈 문자열로 설정하십시오. 쉘 프롬프트에서 다음을 입력하십시오.
  - **Korn 쉘의 경우:** `export JAVA_COMPILER=NONE`

    주: Korn 쉘 명령은 이 정보의 나머지 부분에서 사용됩니다.
  - **Bourne 쉘의 경우:**

    ```
    JAVA_COMPILER=NONE
    export JAVA_COMPILER
    ```
  - **C 쉘의 경우:** `setenv JAVA_COMPILER NONE`
- JVM 명령행에서 **-D** 옵션을 사용하여 java.compiler 특성을 "NONE" 또는 빈 문자열로 설정하십시오. 쉘 프롬프트에서 `java -Djava.compiler=NONE <MyApp>`를 입력하십시오.
- JVM 명령행에서 **-Xint** 옵션을 사용하십시오. 쉘 프롬프트에서 `java -Xint <MyApp>`를 입력하십시오.

JIT가 사용 가능한지 여부 판별

-version 옵션을 사용하여 JIT의 상태를 판별할 수 있습니다.

프로시저

쉘 프롬프트에서 다음을 입력하십시오.

```
java -version
```

JIT가 사용되고 있지 않으면 다음이 포함된 메시지가 표시됩니다.
`(JIT disabled)`
JIT가 사용되고 있으면 다음이 포함된 메시지가 표시됩니다.
`(JIT enabled)`

힙이 없는 실시간 스레드 사용

메트로놈 가비지 콜렉션은 보다 일관된 응답 시간을 제공하지만 가비지 콜렉션으로 인한 인터럽트를 완벽히 피하기에 적합한 경우가 많습니다.

NoHeapRealtimeThreads (NHRT)는 RealtimeThreads에 대한 예외입니다. 힙 메모리에 액세스할 수 없다는 점에서 RealtimeThreads와 다릅니다. 힙에 액세스하지 않고도 NHRT를 가비지 콜렉션 주기 동안 몇 가지 제한사항으로 계속 실행할 수 있습니다. 힙에 액세스하지 않고 따르므로 프로그래밍 모델이 실시간 스레드의 모델과 다릅니다.

NHRT 사용 시 고려사항

NHRT에 대한 다음 사항을 고려하십시오.

- NHRT를 사용하는 주요 이유는 가비지 콜렉션을 허용할 수 없는 태스크 때문입니다. 예를 들어, 시간이 중요한 애플리케이션인 경우 인터럽트를 허용할 수 없습니다.
- 시간이 중요하여 NHRT를 사용할 경우 AOT(ahead-of time) 컴파일러를 사용하는 것도 고려해 보십시오(**-Xnojit** 옵션 사용).
- **-Xrealtime** 옵션을 사용할 경우, 메트로놈 가비지 콜렉터를 자동으로 사용하게 됩니다. 메트로놈 가비지 콜렉터의 이점은 엔터프라이즈에 충분하므로 NHRT를 코딩할 필요가 최소화된다는 것입니다.
- NHRT 스레드는 가비지 콜렉터보다 우선순위가 높으므로 가비지 콜렉터와 독립적으로 실행됩니다. Java 스레드는 1 - 10 범위의 우선순위를 가집니다. NHRT가 있으면 프로그램에 설정된 우선순위에 상관 없이 Java 스레드의 우선순위가 0으로 설정됩니다. 가비지 콜렉터는 가장 높은 실시간 스레드보다 0.5 높은 단계로 자동으로 설정됩니다. NHRT의 우선순위를 가장 높은 실시간 스레드보다 적어도 한 단계 높게 설정합니다. 이렇게 하면 NHRT가 가비지 콜렉터와 독립성을 유지합니다.

  주: 메트로놈 알람 스레드 가비지 콜렉터가 시스템에서 가장 높은 우선순위로 실행되기 때문에 NHRT가 가비지 콜렉션으로부터 완전히 자유롭지는 않습니다. 이 우선순위는 가비지 콜렉터가 작업을 수행해야 하는지 여부를 확인하기 위해 JVM을 활성화할 수 있도록 합니다. 메트로놈 알람 스레드를 실행하는 작업은 매우 작으므로 성능에 크게 영향을 미치지 않습니다. 멀티 프로세서 시스템에서 알람 스레드는 NHRT 스레드와 동시에 실행되므로 가비지 콜렉션 인터럽트가 발생하지 않습니다.
- NHRT는 범위 및 영구 메모리 영역으로 제한되기 때문에 Java 메소드가 힙에서 할당되지 않았는지 확인합니다. 시작 메소드가 확인하고 NHRT가 힙에서 할당될 경우 예외를 리턴합니다(`MemoryAccessError`). NHRT는 `ImmortalMemory` 및 `ScopedMemory`에만 액세스할 수 있습니다.
- 잠금이 공유될 경우 일반 스레드가 NHRT 스레드를 차단할 수 있도록 잠금 시맨틱이 변경되지 않습니다.

- 힙을 사용하는 스레드는 NHRT가 동일한 메소드를 사용할 때 동기화된 메소드에서 우선순위가 향상될 수 있습니다.
- NHRT와 힙 스레드 간의 통신을 위해 비차단 큐를 사용합니다. 그렇지 않으면, 두 유형의 스레드를 구분하십시오.

예외

NHRT를 사용할 때 다음과 같은 예외가 발생할 수 있습니다.

- `IllegalAssignmentError`. 예로, 영구 메모리에 범위 메모리에 대한 참조를 작성하려고 할 때 이 오류가 발생할 수 있습니다.
- `MemoryAccessError`. 예로, NHRT가 힙 메모리를 참조하려고 할 때 이 오류가 발생할 수 있습니다.

비동기 이벤트 처리 제한조건

가비지 콜렉션 동안 NHRT가 차단될 수 있는 여러 사례가 있습니다.

1. NHRT가 힙 메모리에서 할당된 핸들러와 이미 연관된 AsyncEvent에서 fire(), setHandler() 또는 addHandler()를 호출하는 경우
2. NHRT가 힙 메모리에서 할당된 핸들러와 연관된 타이머에서 destroy(), start() 또는 stop()을 호출하는 경우
3. NHRT는 범위를 종료하는 마지막 스레드이고 범위에서 타이머 또는 AsyncEvents를 종료합니다. 그러나 Timers 또는 AsyncEvents는 힙 메모리에서 할당된 핸들러와 연관되어 있습니다.

NHRT와 관련하여 이러한 상황을 피하려면 다음을 수행하십시오.

1. 힙에서 할당된 핸들러를 NHRT에 의해 발생할 수 있는 AsyncEvents 또는 Timers에 추가하지 않도록 하십시오.
2. 힙 메모리에서 할당된 핸들러가 있는 AsyncEvents 또는 Timers가 포함된 범위에서 마지막에 NHRT가 있는 조건을 사용하지 마십시오.

메모리 및 스케줄링 제한조건

JVM에서는 힙이 없는 실시간 스레드가 힙의 오브젝트 참조를 피연산자 스택에 로드할 수 없습니다. 이렇게 하려면 javax.realtime.MemoryAccessError를 발생시키십시오.

또한 JVM은 범위 메모리에 있고 힙 또는 영구 메모리에 저장되는 오브젝트 참조가 발생하지 않도록 주의합니다. NHRT에서 범위 메모리를 단독으로 사용하지 않지만 영구 메모리가 올바르지 않거나 NHRT 컨텍스트에서 메모리 할당 해제가 필요할 경우에는 사용할 수 있습니다.

NHRT를 실행하는 동안 오브젝트 참조로 필드를 채우는 경우 해당 필드에서 이미 존재하는 힙의 오브젝트 참조를 겹쳐쓸 수 있습니다. MemoryAccessError를 생성하지 않고도 NHRT에서 이미 존재하는 참조를 겹쳐쓰게 됩니다.

클래스 로딩 제한조건

클래스가 클래스 로더와 동일한 메모리 영역에 로드됩니다. 클래스 로더의 기본 영역은 영구 메모리입니다.

애플리케이션이 예상되는 응답 시간을 제공하려면 "준비된" 상태여야 합니다. 클래스 로딩이 실시간 스레드와 비동기 이벤트 핸들러를 나중에 인터럽트하지 않도록 애플리케이션이 클래스를 먼저 로드해야 합니다.

NHRT와 함께 실행될 때 Java 스레드에 대한 제한조건

시스템 특성을 JVM에서 공유하고 모든 스레드가 시스템 특성에 액세스할 수 있으므로 NHRT가 실행되는 JVM에서 getProperties 및 setProperties 메소드를 사용할 때 다소 주의해야 합니다. NHRT에서 시스템 특성에 액세스 가능하려면 영구 메모리에 있어야 합니다.

`java.lang.System` 클래스는 다음 메소드를 포함하여 스레드가 시스템 특성과 상호 작용할 수 있는 몇 가지 메소드를 제공합니다.

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

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

실시간 JVM은 실시간 JVM 오브젝트가 모든 시스템 특성을 저장하도록 특별히 작성된 com.ibm.realtime.ImmortalProperties 클래스의 인스턴스를 사용합니다. 이 인스턴스를 사용하면 System.setProperty() 또는 System.getProperties.setProperty() 메소드를 호출할 때 특성이 영구 메모리에 저장됩니다. 이 경우 특별한 사용자 코드가 필요하지 않지만 특성을 설정할 때마다 일부 영구 메모리가 이용되는 것을 이해해야 합니다.

시스템 특성 저장에 공유 특성 오브젝트를 사용하므로 setProperties() 메소드에 대한 호출이 약간 더 까다롭습니다. NHRT가 실행되는 실시간 JVM에서 애플리케이션을 실행하는 경우, setProperties 메소드에 대한 호출이 영구 메모리에 작성된 com.ibm.realtime.ImmortalProperties 클래스 또는 서브클래스의 인스턴스로 전달되어야 합니다. 이 인스턴스를 사용하면 setProperties 메소드로 설정된 모든 특성이 영구 메모리에 저장됩니다.

주: setProperties(null)를 호출하면 새 ImmortalProperties 오브젝트가 기본 특성 세트로 내부에 작성되며 이는 추가 영구 메모리를 이용합니다.

getProperties() 메소드를 호출하면 설정된 오브젝트 또는 기본 특성 오브젝트(com.ibm.realtime.ImmortalProperties 오브젝트)가 리턴됩니다. getProperties() 메소드를 호출하는 기존 코드와의 호환성을 최대화하기 위해 ImmortalProperties 오브젝트가 오브젝트를 직렬화하고 표준 JVM에서 직렬화 해제합니다. 표준 JVM에 ImmortalProperties 오브젝트가 없고 직렬화 해제가 실패하므로 ImmortalProperties 직렬화의 기본 동작은 일반 특성 오브젝트를 직렬화하는 것입니다. 이 기본 동작을 대체하기 위해 ImmortalProperties 클래스가 enabledReplacement(boolean) 메소드를 제공하는데 이는 false로 호출될 경우 기본 동작을 사용 불가능하게 합니다. 이 경우 직렬화에서 ImmortalPropeties 오브젝트를 직렬화한 다음 직렬화 해제하고 결과 오브젝트를 실시간 JVM에서 System.setProperties 메소드 호출에 사용할 수 있습니다.

주: 직렬화 해제는 영구 메모리에서 발생하며 제한된 자원을 너무 많이 사용할 수 있습니다.

보안 관리자

JVM에 있는 모든 유형의 스레드가 시스템에 설정된 보안 관리자를 사용합니다. 이러한 이유로, NHRT가 실행되는 실시간 JVM에서 보안 관리자가 영구 메모리에 할당되어야 합니다. 실시간 JVM은 명령행 옵션에 지정된 보안 관리자가 영구 메모리에 할당되도록 합니다. System.setSecurityManager(SecurityManager) 메소드를 호출하여 보안 관리자를 설정할 수도 있습니다. 애플리케이션이 이런 방식으로 보안 관리자를 설정한 경우 NHRT가 올바르게 실행할 수 있도록 보안 관리자가 영구 메모리에서 할당되었는지 확인해야 합니다.

발생된 예외 및 보안 관리자가 리턴한 오브젝트는 영구 메모리에 있거나 캐싱된 경우 현재 할당 컨텍스트에 할당되어야 합니다.

동기화

MonitorControl 클래스 및 서브클래스 PriorityInheritance는 동기화, 특히 우선순위 변경 제어를 관리합니다. 이 클래스에서는 우선순위 변경 제어 정책을 기본값으로 또는 특정 오브젝트에 대해 설정할 수 있습니다.

WaitFreeReadQueue, WaitFreeWriteQueue 및 WaitFreeDequeue 클래스에서는 스케줄 가능한 오브젝트(특히 NoHeapRealtimeThread 인스턴스)와 일반 Java 스레드 간에 대기 없이 통신할 수 있습니다.

WaitFree 클래스는 NoHeapRealtimeThread 인스턴스와 가비지 콜렉션 지연의 영향을 받는 스케줄 가능한 오브젝트 간에 공유되는 데이터에 대한 안전한 동시 액세스를 제공합니다.

힙이 없는 실시간 클래스 안전

환경에 따라 힙이 없는 컨텍스트에서는 일부 JSE API를 사용하지 못할 수 있습니다. 힙 스레드 및 힙이 없는 스레드 간에 공유하는 클래스에 제한이 적용됩니다. JVM에서 제공하는 안전하게 사용할 수 있는 클래스를 숙지하십시오.

오브젝트 공유

힙이 없는 실시간 스레드에서 실행되는 메소드는 힙의 오브젝트 참조를 로드할 때마다 javax.realtime.MemoryAccessError를 발생시킵니다.

그림 3은 사용하지 않아야 하는 코드 예제입니다.

```
/**
 * 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();
        }

    }
```

그림 3. 힙 오브젝트 참조에 액세스하는 NHRT 예제

```
    /* 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);
        }
    }

}
```

그림 4. 힙 오브젝트 참조에 액세스하는 NHRT 예제(그림 1에서 계속됨)

65 페이지의 그림 3에서 **javax.realtime.MemoryAccessError**를 생성합니다.

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

오브젝트가 힙이 없는 실시간 스레드 및 표준 Java 스레드에 액세스 가능한 경우 오브젝트를 영구 메모리에 할당해야 합니다. 이와 비슷하게, 오브젝트가 힙이 없는 실시간 스레드 및 실시간 스레드에 액세스 가능한 경우 오브젝트를 범위 메모리 영역에 저장할 수도 있습니다.

65 페이지의 그림 3에서 "This on the heap." 문자열 참조가 클래스 변수에 있습니다. 모든 클래스가 영구 메모리에 할당되므로 NHRT에서 이 변수에 액세스할 수 있습니다. 또는, 해당 문자열이 NHRT 생성자로 전달되었을 수 있습니다.

대부분의 오브젝트는 다른 오브젝트 참조를 포함하므로 일반 스레드와 NHRT 간에 이러한 오브젝트를 공유할 때 주의를 기울여야 합니다. 일반적인 예제로 일반 스레드와 NHRT 사이에 공유되고 영구 메모리에 할당되는 LinkedList가 있습니다. 충분한 주의를 기울이지 않을 경우, 표준 스레드가 힙에 있는 오브젝트를 LinkedList에 도입할 수 있습니다. 오브젝트 추적을 위해 LinkedList가 할당하는 데이터 구조가 일반 스레드에 의해 힙에 할당되면 NHRT에 MemoryAccessError가 쉽게 발생할 수 있다는 점에 주의해야 합니다.

몇 가지 클래스는 개별 인스턴스를 할당하는 위치에 상관 없이 NHRT와 다른 스레드 간에 안전하게 공유할 수 없습니다. 대개 캐싱을 위해 클래스 변수에 저장된 오브젝트를 사용하는 클래스입니다. InetAddress는 주소를 캐싱하는 일반적인 예제입니다. InetAddress에서 특정 메소드를 호출하는 첫 번째 스레드가 힙에서 실행 중이면 향후에 NHRT에서 동일한 메소드를 안전하게 호출할 수 없습니다.

NHRT로 오브젝트 잠금

NHRT는 다른 스레드와 동기화를 피해야 합니다. 다음 시나리오를 고려하십시오.

- 우선순위가 낮은 실시간 스레드가 동기화된 블록 또는 메소드를 입력하고 오브젝트에서 동기화합니다.
- 우선순위가 높은 NHRT가 동일한 오브젝트에서 동기화할 때 차단됩니다.
- 우선순위 상속으로 인해 실시간 스레드가 NHRT와 동일한 우선순위를 일시적으로 사용합니다.
- 그런 다음 가비지 콜렉션은 NHRT보다 높은 우선순위로 실행되므로 NHRT를 인터럽트할 수 있습니다. NHRT를 사용하는 이유는 가비지 콜렉션으로 인한 인터럽트를 방지하기 위한 것이므로 이 시나리오는 NHRT 사용을 무효화합니다.

NHRT 및 기타 스레드가 동일한 오브젝트에서 동기화되는 것을 피할 수 없는 경우도 있지만 가능성은 최소화해야 합니다. 오브젝트를 공유할 때 불필요한 동기화를 피하도록 주의하십시오.

안전 클래스에 대한 제한사항

애플리케이션에 실시간 스레드 및 힙이 없는 실시간 스레드 오브젝트가 모두 포함된 경우 몇 가지 고려사항이 적용됩니다.

- 실시간 스레드와 상호작용하면 힙이 없는 실시간 스레드에서 MemoryAccessError가 발생할 수 있습니다.
- 실시간 스레드로 인해 가비지 콜렉션이 발생하여 힙이 없는 실시간 스레드가 지연될 수 있습니다.

힙이 없는 실시간 스레드에서 발생한 MemoryAccessError

두 가지 유형의 스레드가 모두 동일한 클래스에서 메소드를 호출하면 실시간 스레드가 클래스의 static 변수를 힙에서 할당된 오브젝트로 『오염』시킬 수 있습니다. 힙이 없는 실시간 스레드가 해당 힙 오브젝트에 액세스하려고 하면 MemoryAccessError가 수신됩니다. 클래스 인스턴스에서도 오염이 발생할 수 있습니다. 불행하게도, 일반적인 코딩 패턴에서 두 문제점이 나타날 가능성이 크므로 몇 가지 사례를 좀 더 살펴보는 것이 좋습니다.

클래스가 시간 소모적인 조작을 수행할 경우 후속 조작의 성능 향상을 위해 대개 결과를 캐싱하도록 선택합니다. 캐시는 일반적으로 클래스의 정적 변수에 고정된 HashMap 같은 콜렉션입니다. 힙 컨텍스트에서 작동하는 실시간 스레드는 이 콜렉션에 힙 오브젝트를 저장할 수 있으며, 이 경우 오브젝트 자체뿐만 아니라 인프라 오브젝트도 콜렉션에 추가합니다(예: 색인의 일부). 나중에 힙이 없는 실시간 스레드가 콜렉션에 액세스하려고 하면 다른 스레드가 추가한 오브젝트에 액세스하지 않더라도 인프라 오브젝트를 로드하려고 시도하므로 MemoryAccessError를 수신합니다. 클래스 라이브러리가 개발되어 성능에 맞게 조정되므로 이 캐시가 더 일반화되고 있습니다.

클래스 인스턴스도 다양한 방법으로 힙 오브젝트에 의해 오염됩니다. 영구 메모리에 빌드되어 두 가지 유형의 스레드에서 액세스 가능한 인스턴스를 생각해 보십시오. 첫 번

째 오브젝트 사용이 힙 컨텍스트의 실시간 스레드에 의해서이면 보조 오브젝트가 원본 오브젝트의 필드에 저장됩니다. 보조 오브젝트가 힙 컨텍스트에 있으면 힙이 없는 실시간 스레드의 후속 사용에서 MemoryAccessError가 다시 표시됩니다. 이 보조 오브젝트는 첫 번째 사용 시 항상 추가되는 것은 아니지만 여러 번 사용한 후 자주 사용되는 메소드의 성능 향상을 위해 설계되었습니다.

가비지 콜렉션에 의해 지연된 힙이 없는 스레드

가비지 콜렉션에 의해 지연되지 않게 하려면 힙이 없는 스레드에 다른 스레드보다 높은 우선순위를 지정해야 합니다.

추가로, 클래스에 동기화된 메소드가 있으면 이러한 메소드를 호출하는 힙이 없는 실시간 스레드가 가비지 콜렉션으로 인해 의도하지 않게 지연될 수도 있습니다. 이 시나리오에 대한 설명이 66 페이지의 『NHRT로 오브젝트 잠금』에 있습니다.

클래스에 동기화된 메소드(static 또는 인스턴스 메소드)가 있으면 이러한 메소드를 호출하는 힙이 없는 실시간 스레드가 가비지 콜렉션으로 인해 의도하지 않게 지연될 수도 있습니다. 힙이 없는 실시간 스레드가 다른 스레드가 완료될 때까지 대기를 차단하는 다른 동기화된 스레드를 호출하는 시점에 실시간 스레드가 동기화된 메소드(static 또는 인스턴스)에 액세스할 경우 문제점이 발생합니다. 힙이 없는 실시간 가 실시간 스레드보다 우선순위가 높으면 실시간 스레드의 우선순위가 높아집니다. 그런 다음 해당 스레드가 가비지 콜렉션 인터럽트를 대기하도록 강제되면 가비지 콜렉터 스레드가 우선순위가 가장 높은 실시간 스레드보다 높은 우선순위를 가지기 때문에 우선순위가 변경됩니다. 이때, 현재 동기화된 메소드 입력 대기가 차단된 힙이 없는 실시간 스레드만큼 높지는 않습니다.

이러한 문제점을 해결하는 유일한 방법은 힙이 없는 실시간 스레드가 다른 스레드 유형과 공유되는 클래스 또는 인스턴스에서 동기화된 메소드를 호출하지 않도록 하는 것입니다. 불행하게도, 메소드가 동기화되었는지 메소드 서명에서 명확하게 알 수 없습니다. 예를 들어, 동기화된 블록을 포함하거나 동기화된 메소드를 호출할 수 있습니다.

요약

NoHeapRealtimeThread 클래스는 런타임 환경을 더욱 복잡하게 하며 혼합된 유형의 스레드가 환경에서 작동할 경우 많은 문제점을 유발할 수 있습니다. 애플리케이션을 개발하는 동안 다른 스레드 유형과 클래스 사용을 공유하는 영역을 주의하여 설계해야 합니다. 무엇보다도 중요한 점은 이 스레드가 SDK의 클래스를 사용한다는 것입니다. 분석의 복잡함 때문에 SDK에 제공되는 모든 클래스가 이러한 공유 사용 시 안전하다고 보장할 수 없습니다. 대신, 몇 개의 클래스는 검증되었습니다. 초기에 MemoryAccessError 측면을 중심으로 검증되었고, 힙이 없는 스레드와 다른 유형의 스레드에서 사용할 수 있도록 하는 데 필요한 경우 분석, 테스트 및 수정하는 클래스 목록이 그 결과입니다.

안전 클래스

이 절에서는 NoHeapRealtimeThread 및 기타 스레드 유형에서 안전하게 사용할 수 있는 클래스 세트를 설명합니다.

MemoryAccessError 안전 측면에 주요 관심이 맞춰져 있습니다. 다음 목록은 동일한 JVM의 3가지 스레드 유형에서 사용할 수 있는 클래스에 대한 설명입니다.

주: 클래스의 개별 인스턴스를 항상 안전하게 공유할 수 있는 것은 아닙니다.

이 규칙을 따라서 모든 스레드 유형별로 클래스를 안전하게 사용할 수 있는지 확인하십시오.

- 인스턴스를 액세스가 의도된 스레드에 액세스할 수 있는 메모리 영역에 빌드해야 합니다.
- 클래스에 공용 static 필드가 있는 경우 힙 오브젝트를 이 필드에 저장하지 마십시오.
- 클래스에 공용 instance 필드가 있는 경우 힙 오브젝트를 이 필드에 저장하지 마십시오.

IBM이 제공하는 모든 클래스가 NHRT 안전한 것은 아닙니다. 다음 패키지에는 NHRT 안전한 클래스가 포함되어 있습니다.

- java.lang 패키지
- java.lang.reflect 패키지
- java.lang.ref 패키지(모든 클래스)
- java.net 패키지
- java.io 패키지
- java.math 패키지

이 테이블에는 NHRT 안전하지 않은 이 패키지 내의 클래스가 표시됩니다.

표 5. NHRT 안전하지 않은 java.lang 패키지의 클래스

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

표 6. NHRT 안전하지 않은 java.lang.reflect 패키지의 클래스

클래스	메소드
java.lang.reflect.Proxy.*	*

표 7. NHRT 안전하지 않은 java.net 패키지의 클래스

클래스	메소드
java.net.SocketPermission.*	newPermissionCollection()Ljava.net.SocketPermissionCollection;

표 8. NHRT 안전하지 않은 java.io 패키지의 클래스

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

표 9. NHRT 안전하지 않은 java.math 패키지의 클래스

클래스	메소드
java.math.BigInteger	*

이 패키지에는 안전하지 않은 클래스가 있는 하위 패키지를 포함될 수 있습니다. 예를 들어, 다음 클래스는 NHRT 안전하지 않습니다.

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

클래스가 NHRT 안전해 보여도 클래스가 NHRT에서 사용하기에 적합하지 않은 경우가 있습니다. 애플리케이션 개발자가 클래스의 NHRT 안전 여부에 상관 없이 사용할 때마다 클래스의 실시간 요구사항을 판별해야 합니다.

JVM 간 클래스 데이터 공유

JVM(Java Virtual Machine)을 사용하면 디스크의 메모리 맵핑된 캐시 파일에 저장하여 JVM 간에 클래스 데이터를 공유할 수 있습니다.

둘 이상의 JVM이 캐시를 공유하는 경우 데이터를 공유하면 전체 가상 스토리지 이용이 줄어듭니다. 또한 데이터를 공유하면 캐시를 작성한 후 JVM의 시작 시간도 단축됩니다. 공유 클래스 캐시는 활성 JVM에 종속되지 않으며 제거할 때까지 지속됩니다. 공유 캐시에는 다음이 포함될 수 있습니다.

- 부트스트랩 클래스
- 애플리케이션 클래스
- 클래스를 설명하는 메타데이터
- AOT(Ahead-of-time) 컴파일된 코드

IBM WebSphere Real Time for RT Linux에서 실시간 및 실시간 이외 모드로 공유 클래스 캐시를 사용할 수 있지만 캐시 형식, 작성 및 채우기 기술이 다릅니다. 실시간 모드 캐시는 실시간 이외 모드 캐시와 호환되지 않습니다. 실시간 이외 모드에서는 표준 JVM과 동일한 방식으로 캐시를 작성하고 채웁니다. 이는 애플리케이션을 실행할 때 사용자에게 투명하게 JVM에서 캐시를 작성하고 채움을 의미합니다. 실시간 모드에서 **-Xrealtime** 옵션을 사용하면 **admincache**에서 **-populate** 옵션을 사용하여 공유 클래스 캐시를 작성하고 사전에 채워야 합니다. 실시간 모드에서 실행되는 애플리케이션은 사전 채워진 캐시의 컨텐츠를 읽지만 컨텐츠를 수정할 수는 없습니다.

캐시를 작성 및 채우고 제거하려면 **admincache** 도구를 사용하십시오.

애플리케이션에서 공유 클래스 캐시를 사용하게 하려면 **-Xshareclasses** 옵션을 명령행에 추가하십시오. 실시간 모드 캐시는 읽기 전용이므로 **-Xshareclasses**의 일부 실시간 이외 모드 하위 옵션은 실시간 모드에서 사용할 수 없습니다.

자세한 정보는 44 페이지의 『admincache 도구 사용』, 101 페이지의 『실시간 이외 모드에서 JVM 사이의 클래스 데이터 공유』 및 154 페이지의 『공유 클래스 진단』의 내용을 참조하십시오.

공유 클래스 캐시로 애플리케이션 실행

공유 클래스 캐시를 사용하여 애플리케이션을 실행하려면 명령행에서 **-Xshareclasses** 옵션을 사용하십시오.

표 10은 **-Xshareclasses** 옵션을 사용하여 애플리케이션을 실시간 모드로 실행할 때 사용 가능한 하위 옵션을 보여줍니다.

표 10. 애플리케이션을 실시간 모드로 실행할 때 사용 가능한 하위 옵션

옵션	의미
cacheDir=<*directory*>	공유 클래스 캐시 데이터를 읽고 기록하는 디렉토리를 설정합니다. 기본적으로 <*directory*>는 /tmp/javasharedresources입니다. 디렉토리 이름이 캐시 작성을 위해 admincache 명령에 사용된 **-cacheDir** 옵션에 지정된 것과 일치해야 합니다.
name=<*name*>	사용할 공유 클래스 캐시의 이름입니다. 해당 이름이 캐시 작성을 위해 admincache 명령에 사용된 **-cacheName** 옵션에 지정된 것과 일치해야 합니다. 이름이 53자를 초과하지 않아야 합니다.
none	클래스 공유를 명시적으로 사용 불가능하게 합니다. 명령행 끝에 추가하여 클래스 데이터 공유를 사용하지 않게 할 수 있습니다. 이 하위 옵션은 명령행에서 이전에 발견된 클래스 공유 인수를 대체합니다.

표 10. 애플리케이션을 실시간 모드로 실행할 때 사용 가능한 하위 옵션 (계속)

옵션	의미
nonfatal	오류나 경고에 상관없이 JVM을 항상 시작합니다. 클래스 데이터 공유가 실패하는 경우에도 JVM을 시작할 수 있습니다. 클래스 데이터 공유에 실패하는 경우 JVM의 일반적인 동작은 시작을 거부하는 것입니다. **nonfatal**이 선택되고 공유 클래스 캐시가 초기화에 실패하는 경우 JVM은 읽기 전용 모드로 캐시에 연결을 시도합니다. 이 시도가 실패하는 경우 JVM은 클래스 데이터 공유 없이 시작됩니다.
silent	모든 출력 메시지를 표시하지 않습니다. 오류 메시지를 포함하여 모든 공유 클래스 메시지를 표시하지 않습니다. JVM이 초기화되지 않도록 방지하는 복구 불가능 오류 메시지가 표시됩니다.
verbose	공유 클래스 캐시의 전체 상태와 자세한 오류 메시지를 제공하는 상세 출력을 사용합니다.
verboseAOT	캐시에서 컴파일된 AOD 코드를 찾고 있는 경우 상세 출력을 사용합니다(예: AOT 메소드 로드 요청 동안).
verboseHelper	Java Helper API에 대해 상세 출력을 사용합니다. 이 출력은 클래스 로더에서 Helper API를 사용하는 방식을 보여줍니다.
verboseIO	클래스 로드 요청의 상세 출력을 사용합니다. 이 옵션은 찾은 클래스에 대한 정보를 나열하고 캐시 I/O 활동에 대한 상세 출력을 제공합니다.

이 옵션이 올바른지 확인하려면 admincache에 **-printvmargs** 옵션을 사용하십시오 (자세한 정보는 **-printvmargs** 참조). **nonfatal** 옵션은 JVM이 공유 클래스 캐시에 대한 경고 및 오류를 강제로 무시하도록 하므로 일반 용도로는 적합하지 않습니다. **none** 옵션은 클래스 공유를 명시적으로 사용 불가능하게 하고 명령행에서 **-Xshareclasses** 옵션을 생략하는 것과 동일합니다.

-Xshareclasses 하위 옵션에 대한 자세한 정보는 클래스 데이터 공유 명령행 옵션을 참조하십시오.

메트로놈 가비지 콜렉터 사용

WebSphere Real Time for RT Linux에서 표준 가비지 콜렉터 대신 메트로놈 가비지 콜렉터를 사용합니다.

프로세서 사용 제어

메트로놈 가비지 콜렉터에 사용 가능한 처리 성능 크기를 제한할 수 있습니다.

가비지 콜렉터에서 사용되는 CPU 시간을 한정하도록 **-Xgc:targetUtilization=N** 옵션을 사용하여 메트로놈 가비지 콜렉터로 가비지 콜렉션을 제어할 수 있습니다.

예를 들면 다음과 같습니다.

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

예제에서는 애플리케이션이 60밀리초마다 80%에 대해 실행됨을 지정합니다. 나머지 20% 시간은 가비지 콜렉션에 사용됩니다. 메트로놈 가비지 콜렉터는 충분한 자원이 제공될 경우에 한해 이용 수준을 보장합니다. 힙의 여유 공간 *크기*가 동적으로 판별되는 임계값보다 작아지면 가비지 콜렉션이 시작됩니다.

메트로놈 가비지 콜렉터 조정

애플리케이션이 사용하는 메모리 양을 제어하여 실시간 환경을 조정할 수 있습니다. 예를 들어, **-Xmx**, **-Xgc:immortalMemorySize=size**, **-Xgc:scopedMemoryMaximumSize=size** 및 **-Xgc:targetUtilization=N** 옵션을 사용합니다.

- 힙의 *크기*를 제한하려면 **-Xmx** 옵션을 사용하십시오.

  선택한 값은 힙 *크기*의 상한으로 사용되므로 시간에 따른 사용을 반영합니다. 너무 작은 값을 선택하면 가비지 콜렉션 빈도가 늘어나 메모리 풋프린트를 줄이더라도 전체 처리량이 작아질 수 있습니다. 실시간 성능 향상을 위해 페이징을 사용하지 마십시오. 머신에서 실행되는 모든 프로세스의 풋프린트가 실제 메모리 *크기*를 초과하지 않도록 하는 것이 일반적입니다.
- 영구 메모리 영역의 *크기*를 제어하려면 **-Xgc:immortalMemorySize=size** 옵션을 사용하십시오.

  영구 메모리 사용을 주의하여 분석해야 합니다. 『이상적인』 애플리케이션은 시작하는 동안 영구 메모리를 사용하지만 이후에 사용을 중지합니다. 영구 오브젝트 할당이 계속되면 영구 메모리가 소진될 때까지 애플리케이션이 계속 실행될 수 있습니다. 다음을 코드에 추가하여

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

  현재 사용을 확보합니다.
- 애플리케이션이 범위 메모리를 과도하게 요청하지 않도록 하려면 **-Xgc:scopedMemoryMaximumSize=size** 옵션을 사용하십시오. 조정이 아닌 진단을 위해 이 옵션을 사용하십시오.
- 최악의 조건(힙 오브젝트의 최대 할당률)에서 가비지 콜렉터가 애플리케이션이 생성하는 것보다 높은 속도로 가비지를 콜렉션하게 하려면 **-Xgc:targetUtilization=N** 옵션을 설정하십시오.

  일반적으로 기본값이면 충분하지만, 콜렉터가 애플리케이션이 생성하는 것보다 빠르게 가비지를 콜렉션하는 지점으로 이용률을 높이면 애플리케이션 성능이 향상됩니다.
- 가비지 콜렉션을 병렬로 실행하기 위해 추가 스레드를 작성하려면 **-Xgcthreads <n>** 옵션을 사용하십시오.

기본값은 한 개 스레드를 사용하는 것입니다. 워크로드의 가비지 생성 속도가 높고 CPU 주기가 사용 가능한 시맨틱 멀티프로세서에서 실행될 경우 이 매개변수를 >1 로 설정하면 성능 이점이 있습니다.

주: 이 매개변수를 너무 높게 설정하면 처리량에 부정적인 영향을 미칠 수 있습니다.

메트로놈 가비지 콜렉터 제한사항

드문 상황에서, 가비지 콜렉션 중에 예상된 일시정지보다 더 길어질 수도 있습니다.

가비지 콜렉션 동안 루트 스캔 프로세스가 사용됩니다. 가비지 콜렉터가 알려진 라이브 참조에서 시작하여 힙을 지나갑니다. 이 참조는 다음과 같습니다.

- 활성 스레드 호출 스택의 라이브 참조 변수
- Static 참조
- 영구 메모리 및 범위 메모리의 모든 오브젝트 참조

애플리케이션 스레드 스택에서 모든 라이브 오브젝트 참조를 찾기 위해 가비지 콜렉터가 스레드 호출 스택에서 모든 스택 프레임을 스캔합니다. 각 활성 스레드 스택을 인터럽트 불가능한 단계에서 스캔합니다. 따라서 스캔은 개별 GC 일시정지 내에서 발생해야 합니다.

그 영향으로, 매우 깊은 스택을 포함한 스레드가 몇 개 있는 경우 시스템 성능이 예상보다 저하질 수 있는데 이유는 콜렉션 주기를 시작할 때 가비지 콜렉션 일시정지가 길어지기 때문입니다.

영구 메모리는 증분식으로 처리됩니다. 다른 모든 범위 메모리 영역은 한 번의 원자 인터럽트 불가능한 단계로 처리됩니다. 이는 루트 스캔이 범위 메모리를 처리할 때 가비지 콜렉션 일시정지가 길어지므로 범위 메모리 영역을 많이 사용하면 시스템 성능이 예상보다 저하될 수 있음을 의미합니다.

제 6 장 애플리케이션 개발

코드 샘플을 포함한 실시간 애플리케이션 작성에 대한 중요 정보입니다.

- 『실시간 이용을 위해 Java 애플리케이션 작성』
- 88 페이지의 『샘플 애플리케이션』
- 97 페이지의 『샘플 실시간 해시 맵』
- 97 페이지의 『Eclipse를 사용하여 WebSphere Real Time for RT Linux 애플리케이션 개발』

실시간 이용을 위해 Java 애플리케이션 작성

이 예제는 실시간 환경을 이용하는 방법을 설명합니다. 코드 수정 없이 실시간으로 Java 애플리케이션을 실행하는 가장 간단한 예제에서 힙이 없는 실시간 스레드를 계획 및 작성하는 복잡한 프로세스까지 다양합니다. 사용자 애플리케이션에 가장 적합한 방법을 결정하는 데 도움이 되는 이유가 제공됩니다.

실시간 애플리케이션 작성 소개

실시간 기술의 기능을 이용하기 위해 정교한 힙이 없는 실시간 애플리케이션을 작성할 필요는 없습니다. 기존 코드를 거의 변경하지 않고 일부 이점을 사용할 수 있습니다.

애플리케이션 프로그래머의 경우, WebSphere Real Time for RT Linux 이용을 위해 다음 단계를 수행합니다.

1. 실시간 JVM에서 표준 Java 애플리케이션을 실행하면 메트로놈 가비지 콜렉션의 이점을 이용하고 애플리케이션 런타임 예측 기능을 대폭 향상시킬 수 있습니다.
2. AOT(Ahead-of-Time) 컴파일러를 사용하기 위해 코드를 사전 컴파일한 후 **-Xnojit** 옵션을 추가합니다. 55 페이지의 『공유 클래스 캐시에 사전 컴파일된 jar 파일 저장』의 내용을 참조하십시오.
3. 애플리케이션에서 java.lang.Thread를 javax.realtime.RealtimeThread로 대체합니다. AOT 옵션과 비교할 때 약간 향상된 점을 볼 수 있습니다.

   실시간 스레드 사용 시 주요 이점은 각 스레드에 제공하는 우선순위를 제어할 수 있다는 것입니다. 실시간 스레드를 정기적으로 설정할 수도 있습니다. 이 이점을 이용하려면 애플리케이션 자체를 변경하기 위해 준비해야 합니다.
4. 타이머 또는 외부 이벤트를 처리하기 위해 실시간 스레드 및 비동기 이벤트 핸들러를 사용하도록 특정 애플리케이션을 계획하고 작성합니다. 다음 3가지 요소를 고려하십시오.
   - 실시간 스레드에 지정하는 우선순위 계획

- 오브젝트를 저장할 메모리 영역 결정
- 이벤트 핸들러와 통신

5. 힙이 없는 실시간 스레드를 사용하도록 특정 애플리케이션 계획 및 작성. 힙이 없는 실시간 스레드는 실시간 스레드의 확장이며 지정한 우선순위 및 메모리 영역을 고려해야 합니다. 일반적으로, 애플리케이션이 GC 일시정지 시간(서브밀리초)과 비슷한 시간에 이벤트를 처리해야 할 경우 이 단계를 수행하십시오. 힙이 없는 실시간 스레드를 사용한 개발의 복잡함을 과소 평가하지 마십시오.

그림 5은 이전에 설명한 단계를 보여줍니다.

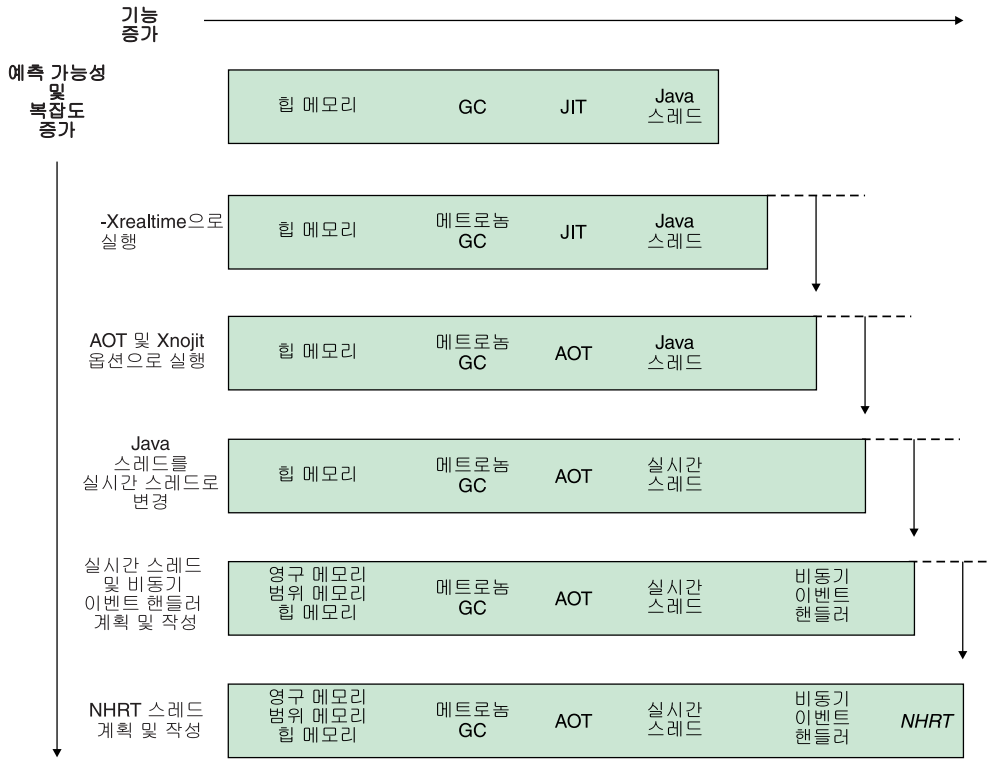

그림 5. 예측 가능성이 향상된 RTSJ 기능 비교.

WebSphere Real Time for RT Linux 애플리케이션 계획

실시간 Java 애플리케이션 작성을 위해 준비할 때 Java 스레드, 실시간 스레드 또는 힙이 없는 실시간 스레드를 사용할 것인지 고려해야 합니다. 또한, 스레드가 사용할 메모리 영역을 결정합니다.

이 태스크 정보

애플리케이션을 계획할 때 다음 단계는 필수 의사 결정을 설명합니다.

프로시저

1. 태스크를 식별하십시오.
2. 타이밍 기간을 결정하십시오.
   - 응답이 10ms를 초과하면 Java 스레드를 선택하고 메트로놈 가비지 콜렉터를 이용하십시오.

     이 스레드는 저장 공간으로 힙 메모리만 사용합니다. 그러나, 메트로놈 가비지 콜렉터에서 제어하기 때문에 가비지 콜렉션이 애플리케이션을 인터럽트하는 단점이 있으며 인터럽트의 길이와 시간은 예측 가능합니다.
   - 응답이 10ms 미만이면 실시간 스레드를 선택하십시오.

     실시간 스레드를 힙, 범위 또는 영구 메모리에 놓을 수 있습니다. 실시간 스레드 사용 시 이점은 다음과 같습니다.
     – 표준 Java 스레드보다 높은 우선순위로 실행할 수 있습니다.
     – 메트로놈 가비지 콜렉터가 가비지 콜렉션을 제어합니다. 그러나, 가비지 콜렉터는 우선순위가 가장 높은 실시간 스레드보다 높은 우선순위로 실행되므로 프로그램의 실행을 인터럽트합니다.
   - 응답이 1ms 미만이면 힙이 없는 실시간 스레드를 선택하십시오.

     힙이 없는 실시간 스레드의 우선순위를 가비지 콜렉션보다 높게 설정할 수 있으므로 메트로놈에 의해 인터럽트되지 않습니다. 메트로놈 알람 스레드는 높은 우선순위로만 실행되며 CPU를 적게 사용합니다.
3. 애플리케이션에 비동기 이벤트 핸들러가 필요한지 여부를 판별하십시오. 이 요구사항은 프로그램의 구조에 따라 다릅니다.
   - 응답 시간이 10ms 미만이면 실시간 스레드를 선택하십시오.
   - 응답 응답이 1ms 미만이면 힙이 없는 실시간 스레드를 선택하십시오.
4. 스레드 우선순위를 판별하십시오. 일반적으로 기간이 짧을 수록 우선순위가 높아집니다.
5. 메모리 특성을 결정하십시오.
   - 태스크에 GC를 압도하는 변수 또는 높은 할당률이 있으면 비율 한계를 적용하거나(MemoryParameters 사용) 범위 메모리 영역에 할당하는 것을 고려하십시오.
   - 태스크가 계산하는 동안 임시 데이터를 대량 생성할 경우, 범위 메모리 영역 사용을 고려하십시오.
   - 태스크가 시작하는 동안 JVM 수명 동안 필요한 몇 가지 데이터를 생성할 경우, 영구 메모리 사용을 고려하십시오. 오브젝트가 JVM 수명 동안 계속 작성될 경우 영구 메모리를 사용하지 마십시오.
   - 태스크가 통신해야 할 경우, 특히 힙이 없는 실시간 스레드 아래에서 실행 중이면 통신을 위해 범위 메모리 영역 사용을 고려하십시오.

- 태스크가 힙이 없는 실시간 스레드 아래에서 실행 중이면 범위 메모리 영역(예: LTMemory)을 빌드하여 힙이 없는 스레드, 런타임 매개변수, 태스크와 통신하는 데 사용되는 대기 없는 큐를 포함하는 것을 고려하십시오. LTMemory 오브젝트는 힙이 없는 스레드가 참조하려고 할 때 오류가 발생하지 않도록 영구 또는 다른 범위에 빌드해야 합니다.

6. 애플리케이션의 구조 및 컨텐츠를 결정한 경우, 애플리케이션의 성능을 높이기 위해 런타임 옵션을 수정하십시오. 다음 단계에서 그 방법을 설명합니다.
   a. 초기 애플리케이션 테스트 동안 **-Xmx**, **-Xgc:immortalMemorySize=size** 및 **-Xgc:scopedMemoryMaximumSize=size** 옵션을 사용하여 힙, 범위 및 영구 메모리에 여유 공간을 설정하십시오.

      주: 메트로놈 GC를 사용할 경우 메트로놈 GC가 힙의 *크기*를 늘리지 않으므로 초기 및 최대 힙 *크기*가 동일해야 합니다. 힙을 늘리는 것은 비결정적 조작입니다.
   b. **-verbose:gc** 옵션을 사용하여 사용된 메모리 양을 판별하십시오.
   c. 가비지 콜렉션을 위한 충분한 시간을 허용하도록 **-Xgc:targetUtilization** 옵션을 수정하십시오. 기본값은 70%이고 이 백분률은 대부분의 애플리케이션에서 적합합니다. 가비지 콜렉션 비율이 할당율보다 약간 높은지 확인하십시오.
   d. **-Xmx** 옵션을 사용하여 힙 메모리의 실제 *크기*를 설정하십시오.

Java 애플리케이션 수정

실시간 Java 기능을 이용하는 코드를 작성하려면 `javax.realtime.RealtimeThread`를 사용하여 스레드를 `java.lang.Thread`로 대체하십시오.

시작하기 전에

이 예제는 `demo/realtime/sample_application.zip` 파일에 있는 JavaRadar.java 클래스에 기반합니다.

이 태스크 정보

실시간 스레드의 프로그래밍 모델은 표준 Java 애플리케이션의 모델과 비슷합니다. 그러나, 실시간 스레드를 프로그램에 추가하는 조잡한 방법은 WebSphere Real Time for RT Linux의 기능을 전부 사용하지 않습니다. 이를 위해 스레드를 수정하여 우선순위를 연관시키고 사용할 메모리 영역도 고려해야 합니다.

스레드의 클래스를 변경하여 애플리케이션의 몇몇 이점만 확보합니다. 실시간 스레드의 기본 우선순위가 표준 Java 스레드의 우선순위보다 *크기* 때문입니다.

JavaRadar를 RealtimeThread로 변경하려면 확장하는 클래스를 `Thread`에서 `RealtimeThread`로 변경합니다.

*javax.realtime.RealtimeThread*가 *java.lang.Thread* 대체

샘플 애플리케이션의 JavaRadar 클래스는 java.lang.Thread를 확장합니다. 예를 들면 다음과 같습니다.

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

이 Java 스레드를 실시간 스레드로 설정하려면 이 클래스 정의를 다음과 같이 재정의합니다.

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

실시간 스레드 작성

지금까지 애플리케이션을 수정했고 이제는 몇 가지 코드를 작성할 차례입니다. 실시간 스레드를 사용하여 실시간 우선순위 레벨 및 메모리 영역을 이용하는 애플리케이션을 작성할 수 있습니다.

시작하기 전에

이 예제는 demo/realtime/sample_application.zip 파일에 있는 JavaRadar.java, RTJavaRadar.java 및 RTJavaControlLauncher.java 클래스에 기반합니다.

이 샘플은 78 페이지의 『Java 애플리케이션 수정』에 설명된 동일 샘플에서 영구 메모리를 사용하는 방법을 보여줍니다.

이 태스크 정보

실시간 스레드의 프로그래밍 모델은 표준 Java 애플리케이션의 모델과 비슷합니다.

실시간 스레드 사용 시 이점은 다음과 같습니다.

- 실시간 스레드에서 OS 레벨 스레드 우선순위를 전면 지원합니다.
- 범위 또는 영구 메모리 영역을 사용합니다.
  - 범위 메모리를 사용하면 가비지 콜렉션에 영향을 주지 않으면서 메모리 할당 해제를 명시적으로 제어할 수 있습니다.
  - 힙이 없는 실시간 스레드를 사용하면 가비지 콜렉션이 일시정지되지 않도록 영구 메모리를 사용할 수 있습니다.
  - 힙의 오브젝트를 참조하는 해당 실시간 스레드는 힙 메모리에 저장된 실시간 스레드처럼 가빕지 콜렉션의 영향을 받습니다.
  - 힙이 없는 실시간 스레드는 힙 메모리의 오브젝트를 참조할 수 없으며, 따라서 가비지 콜렉션의 영향을 받지 않습니다.

80 페이지의 표 11에서는 SimulationThread의 우선순위가 가장 높다는 기반으로 우선순위가 지정됩니다. 이 항목은 외부 이벤트를 나타내며 프로그램의 다른 항목이 선점하도록 허용되지 않아야 하기 때문입니다. RadarThread는 제어기의 ping에 빠르게 응답

해야 합니다. 응답이 빠를 수록 달 착륙선의 높이 측정이 더 정확해집니다. 또한 ListenThread는 제어기의 명령에 빠르게 응답해야 하지만 RadarThread로 보조 위치를 가져갑니다.

이 3개 스레드는 시뮬레이션이 서버로 실행되므로 범위 메모리에 있습니다. 서버가 시뮬레이션을 실행한 후 범위 메모리 영역을 종료한 다음 다른 시뮬레이션 실행을 대기하기 위해 다시 들어갈 수 있습니다. 자체 설정할 수 있도록 서버가 범위 메모리를 사용합니다.

RTJavaRadarthread에는 시간에 보다 민감하고 이 시간을 사용하여 높이 파생값을 구하므로 우선순위가 가장 높은 제어기 스레드가 있습니다. NHRT로 실행 중이고 제어기는 한 번만 실행되며 JVM 종료 시 메모리가 해제되기 때문에 영구적입니다.

RTJavaControlThread 및 RTJavaEventThread의 경우, 시간 제한조건이 많이 중요하지 않기 때문에 힙 메모리를 사용할 수 있습니다.

RTLoadThread는 달 착륙선에 유용한 기능을 수행하지 않습니다. 그러나 RTLoadThread는 달 착륙선의 가장 우선순위가 높은 스레드 성능에 영향을 미치지 않으면서 상당한 메모리 할당 및 할당 해제가 다른 스레드보다 낮은 우선순위로 수행될 수 있음을 보여줍니다.

표 11. 샘플 애플리케이션에서 메모리 영역 및 스레드의 관계

메모리	스레드	우선순위
범위	demo.sim.SimulationThread	38
	demo.sim.RadarThread	37
	demo.sim.SimulationThread.ListenThread	36
영구	demo.controller.RTJavaRadarThread	15
힙	demo.controller.RTJavaControlThread	14
	demo.controller.RTJavaEventThread	13
범위 및 힙	demo.controller.RTLoadThread	12

예

demo.sim.SimulationThread의 이 코드는 우선순위 38이 설정된 경우를 보여줍니다. **1** 이 코드 행은 JVM에서 사용 가능한 최대 우선순위를 검색합니다.

```
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.SimLauncher의 이 코드는 범위 메모리가 정의된 경우를 보여줍니다. **2** 는 선형 시간으로 메모리를 할당하는 범위 메모리 영역인 LTMemory의 할당을 보여줍니다.

```
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;
```

NHRT가 ScopedMemoryArea를 나타내는 오브젝트를 참조할 수 있어야 하기 때문에 simMemArea에 참조되는 ScopedMemoryArea 오브젝트는 영구 메모리에 할당됩니다. 힙에 할당하면 메모리 영역 인수가 힙에 있으므로 NHRT 생성자가 IllegalArgumentException을 발생시킵니다.

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

demo.controller.RTJavaControlLauncher의 이 코드는 RTJavaRadar가 영구 메모리를 정의하고 사용하는 경우를 보여줍니다. 제어기 JVM의 전체 수명 동안 RTJavaRadar가 한 번 실행되므로 시작 시에만 메모리를 할당하도록 설계되었습니다. 영구 메모리에서 안전하게 실행할 수 있습니다. 제어기가 범위 메모리 영역에 먼저 들어갈 필요 없이 RTJavaRadar 메소드에 액세스할 수 있으므로 이러한 애플리케이션 설계가 유용합니다. 제어기가 일반 Java 및 실시간 Java에서 실행되도록 작성되었기 때문에 범위 메모리 영역에 들어가는 것이 어렵습니다.

```
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;
```

비동기 이벤트 핸들러 작성

비동기 이벤트 핸들러는 스레드 외부에서 발생하는 이벤트 또는 타이머 이벤트에 상호작용합니다(예: 애플리케이션 인터페이스로부터의 입력). 실시간 시스템에서 이 이벤트는 애플리케이션에 설정한 최종 기한 안에 응답해야 합니다.

시작하기 전에

이 예제는 `demo/realtime/sample_application.zip` 파일에 있는 RTJavaEventThread.java 및 RTJavaControlLauncher.java 클래스에 기반합니다.

이 태스크 정보

샘플 애플리케이션에서 이벤트 스레드가 Crash(추락) 또는 Land(착륙)을 신호하는 시뮬레이션으로부터 이벤트를 대기합니다. 이 스레드의 실시간 버전에서 AsyncEvent 메커니즘이 사용됩니다. 이 이벤트는 적합한 상태 메시지를 인쇄하고 제어기를 종료시키는 데 사용됩니다.

RTJavaEventThread에는 두 개의 비동기 이벤트가 정의되어 있습니다. 둘 다 매개변수는 없습니다.

```
public class RTJavaEventThread extends RealtimeThread {

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

이 이벤트는 두 개의 비동기 이벤트 핸들러를 작성하고 등록합니다.

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

Crash(추락) 또는 Land(착륙) 메시지를 수신하면 상응하는 비동기 이벤트 핸들러가 실행되어 Runnable 오브젝트가 릴리스됩니다.

```
                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;
        }
```

결과

RTJavaControlLauncher.java는 addLandHandler 및 addCrashHandler 메소드에 대한 호출을 포함합니다. 전달된 Runnable 오브젝트로 인해 메시지가 콘솔에 인쇄되고 연관된 비동기 이벤트 핸들러가 실행될 때 제어 스레드가 중지됩니다. 트리거되는 시점은 RTJavaEventThread.java를 참조하십시오.

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

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

NHRT 스레드 작성

힙이 없는 실시간 스레드(NHRT)를 Java 애플리케이션에 추가하려면 이 학습서를 사용하여 사용자 프로그램을 개발하거나 수정하십시오.

시작하기 전에

이 예제는 `demo/realtime/sample_application.zip` 파일에 있는 SimulationThread.java 및 SimLauncher.java 클래스에 기반합니다.

이 태스크 정보

demo.sim.SimulationThread 클래스는 데모 애플리케이션에 있는 시뮬레이션의 부분입니다. 실제 사용 시 대체 방법으로 작동하므로 나머지 시스템의 인터럽트 없이 실행됩니다. 가비지 콜렉션 또는 시스템의 다른 스레드가 해당 스레드를 인터럽트하지 않도록 스레드가 가용성 우선순위가 가장 높은 NoHeapRealtimeThread로 작성됩니다.

SimulationThread에서 다음 생성자는 수퍼 생성자 『NoHeapRealtimeThread (SchedulingParameters scheduling, MemoryArea area)』를 호출하고, 그 이전에 SchedulingParameters 및 ReleaseParameters를 따로 설정합니다.

```
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;
    }
```

시뮬레이션의 다른 활성 스레드도 힙이 없는 실시간 스레드(NHRT)로 작성되지만, 우선순위가 약간 낮습니다. 우선순위 정렬은 79 페이지의 『실시간 스레드 작성』의 내용을 참조하십시오.

시뮬레이션은 시뮬레이션이 완료된 후 시작하도록 무제한 실행 옵션을 제공합니다. 시뮬레이션이 NHRT로 구성되므로 ScopedMemory 또는 ImmortalMemory를 선택할 수 있습니다. 시뮬레이션 완료 시 할당된 ScopeMemoryArea를 종료하고 다음 실행을 대기하기 위해 다시 입력하는 것이 적합하기 때문에 샘플 애플리케이션은 해당 시뮬레이션에 ScopedMemory를 사용합니다. 이 경우, 각 실행 간에 상태가 전달되지 않습니다.

대부분의 클래스는 NHRT 안전합니다. 그러나, 대부분의 클래스는 NHRT 안전하지 않는 방식으로 실행할 수 있습니다. 예를 들어, DatagramSockets을 영구 메모리 또는 외부 범위 메모리 영역에 보관하는 경우, 메모리 영역을 포괄하도록 설계되지 않기 때문에 문제점이 발생할 수 있습니다. 샘플 애플리케이션은 이러한 문제점을 방지하기 위해 한 개 ScopedMemory 영역만 사용합니다.

RTSJ에서 메모리 할당

RTSJ에서 다양한 방법으로 특정 메모리 영역에 오브젝트를 할당할 수 있으며 특정 시점에 어떤 방법을 선택할지는 명백하지 않은 경우가 있습니다.

각 방법은 고유의 특성이 있으며, RTSJ 구현에 따라 다르고 궁긍적인 메모리 풋프린트 또는 성능의 차이를 가져옵니다. 이 절에서는 사용 가능한 옵션을 대략 설명하고 오브젝트 할당을 위해 가장 적합한 선택사항인 경우를 제안합니다.

static 초기화 프로그램

오브젝트를 영구 메모리 영역에 할당하는 가장 간단한 방법은 static 초기화 프로그램에 할당하는 것입니다. 변경되는 메모리 컨텍스트 문제를 처리할 필요가 없는 이점이 있지만, 이 패턴이 적합한 상황은 매우 제한되어 있습니다. 이 방법은 사용되는 영구 메

모리 크기가 오브젝트 자체에 필요한 크기로 제한되는 점에서 효율적입니다.

MemoryArea.newInstance(Class c)

이 방법은 스레드가 메모리 컨텍스트에 있고 오브젝트(스레드의 범위 스택에 이미 있어야 함)를 다른 영역에 할당하려는 경우 간단합니다. 인스턴스화할 클래스에만 액세스하는 이점이 있지만 newInstance 메소드가 적합한 생성자를 빌드해야 합니다. 이 패턴은 특정 클래스의 오브젝트를 가끔 할당해야 할 경우 적합하며, 그렇지 않으면 메모리 사용이 늘어나는 경향이 있습니다.

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

다시, 이 방법은 스레드가 메모리 컨텍스트에 있고 오브젝트(스레드의 범위 스택에 이미 있어야 함)를 다른 컨텍스트에 할당하려는 경우 간단합니다. 이 경우, 생성자 및 일부 인수를 전달해야 하고 생성자가 현재 메모리 컨텍스트에서 유효한지 확인하는 과정을 수행해야 합니다. newInstance 메소드는 생성자를 빌드할 필요가 없기 때문에 메모리 사용이 newInstance(Class c)보다 적으므로 이 패턴은 오브젝트를 자주 할당하고 생성자를 미리 할당하여 ImmortalMemory 같이 다른 곳에 저장할 수고를 감내할 용의가 있는 경우 적합합니다.

새 <class>()가 뒤에 있는 MemoryArea.enter(Runnable r)

이 방법은 특정 메모리 영역을 새 할당 기본값으로 설정하고 반영 요구 및 참가 생성자 오브젝트를 제거합니다. 따라서 오브젝트 자체 이상으로 추가 메모리 사용이 발생하지 않기 때문에 많은 오브젝트를 작성할 경우 가장 적합합니다. 이 방법은 원하는 영역이 스레드의 범위 스택에서 아직 활성이 아닌 경우에만 작동합니다. 실행 가능 메모리 영역을 빌드해야 하므로 이 방법은 newInstance를 사용할 때보다 더 복잡합니다. 실행 가능에서 또는 static 또는 인스턴스 필드에서 매개변수를 전달해야 하기 때문입니다.

새 <class>()가 뒤에 있는 MemoryArea.executeInArea (Runnable r)

다시, 이 방법은 특정 메모리 영역을 새 할당 기본값으로 설정하고 반영 요구 및 참가 생성자 오브젝트를 제거합니다. 따라서 오브젝트 자체 이상으로 추가 메모리 사용이 발생하지 않기 때문에 많은 오브젝트를 작성할 경우 가장 적합합니다. 원하는 영역이 현재 스레드의 범위 스택에 이미 있는 경우 이 방법을 사용할 수 있으며 MemoryArea.enter보다 유연합니다. 실행 가능 메모리 영역을 빌드해야 하므로 이 방법은 newInstance를 사용할 때보다 더 복잡합니다. 실행 가능에서 또는 static 또는 인스턴스 필드에서 매개변수를 전달해야 하기 때문입니다.

Class.newInstance()

이 방법은 현재 메모리 영역에 새 인스턴스를 빌드하므로 MemoryArea.enter 또는 executeInArea와 함께 사용해야 합니다. 오브젝트 자체 이상으로 추가 메모리 사용이 발생하지 않습니다.

고해상도 타이머 사용

실시간 클럭은 표준 JVM과 연관된 클럭의 정밀도를 높여 줍니다.

시작하기 전에

이 예제는 `demo/realtime/sample_application.zip` 파일에 있는 RTJavaRadar.java 클래스에 기반합니다.

이 태스크 정보

일반 Java에서는 클럭 및 타이머를 처리하는 기능이 제한되어 있습니다. Real-Time Specification for Java에서는 나노초의 정밀도와 충분한 클럭 시간으로 절대 시간을 보다 구체적으로 설정할 수 있습니다. javax.realtime.HighResolutionTime 및 서브클래스는 밀리초 및 나노초라는 두 개 컴포넌트로 시간을 표현하는 데 사용됩니다.

WebSphere Real Time for RT Linux는 고해상도 시간을 제공하기 위해 기본 운영체제 지원을 사용합니다. Current® Linux 커널은 클럭에 최대 4밀리초의 정밀도를 보장합니다. WebSphere Real Time for RT Linux에서 제공되는 Linux 패치는 1마이크로초에 근접한 정밀도를 클럭에 제공합니다.

RTJavaRadar 클래스는 고해상도 타이머 사용을 증명합니다.

- **1** 은 실시간 클럭을 가져옵니다.
- **2** 는 현재 절대 시간을 가져옵니다.
- **3** 은 나노초 시간 컴포넌트를 가져옵니다. 실시간 클럭의 정확도는 나노초 사용이 합리적임을 의미합니다.
- **4** 는 ping 이전 및 이후 시간을 가져옵니다.
- **5** 는 착륙선의 하강 속도를 리턴합니다.
- **6** 은 다른 반복을 수행하기 전에 5밀리초 동안 스레드가 대기하도록 합니다.

```
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.
                }
            }
```

앞의 코드를 JavaRadar 클래스의 다음 표준 JVM 코드와 비교할 수 있습니다.

```
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;
    }
```

샘플 애플리케이션

샘플 애플리케이션은 다양한 예제를 사용하여 Java 프로그램의 실시간 특성을 향상시킬 수 있는 WebSphere Real Time for RT Linux의 기능을 증명합니다.

샘플 애플리케이션의 소스 파일이 `demo/realtime/sample_application.zip` 파일에 있습니다.

샘플은 다음 두 가지 주요 컴포넌트로 구성됩니다.

- **시뮬레이션** 달 착륙기의 간단한 예제입니다. 위치는 지상으로부터의 높이와 착륙 지역으로부터의 거리로 정의됩니다. 90 페이지의 그림 6의 내용을 참조하십시오.

  시뮬레이션 클래스는 힙이 없는 실시간 스레드(NHRT)를 사용하여 작성하고 이 문서에서는 더 이상 수정하지 않습니다.

- 시뮬레이션에 명령을 보내는 **제어기**. 착륙기의 높이를 판단하고 이 정보를 기반으로 착륙기의 하강 속도를 제어하는 레이더 ping을 보냅니다. 또한 제어기는 착륙기로부터 정보 스트림을 수신합니다(예: 착륙 지역으로부터 착륙기의 거리).

  제어기는 처음에 표준 Java로 작성됩니다. 78 페이지의 『Java 애플리케이션 수정』에서는 실시간 Java 프로그램으로 개발됩니다.

착륙 결과에 따라 추락(crash) 또는 착륙(land)이라는 두 메시지 중 하나를 제어기에 보냅니다.

샘플 애플리케이션을 사용하여 이 조작을 수행할 수 있습니다.

- 시뮬레이션 및 제어기를 함께 실행하면 함께 실행되는 실시간 및 표준 Java 클래스의 조합이 증명됩니다. 자세한 정보는 91 페이지의 『샘플 애플리케이션 빌드』 및 91 페이지의 『샘플 애플리케이션 실행』의 내용을 참조하십시오. 여기에서 샘플 애플리케이션에서 예상되는 출력을 볼 수도 있습니다.

  주: LaunchBoth 클래스를 사용하면 시뮬레이션 및 제어기를 동시에 시작할 수 있습니다.

- 메트로놈 가비지 콜렉터 및 표준 가비지 콜렉터를 사용할 때 차이점을 비교하십시오. 자세한 정보는 91 페이지의 『Real Time 없이 샘플 애플리케이션 실행』 및 93 페이지의 『메트로놈 가비지 콜렉터를 사용하여 샘플 애플리케이션 실행』의 내용을 참조하십시오.

- AOT(Ahead-of-Time) 컴파일러를 사용하여 애플리케이션을 실행하십시오. 자세한 정보는 94 페이지의 『AOT를 사용하는 동안 샘플 애플리케이션 실행』의 내용을 참조하십시오.

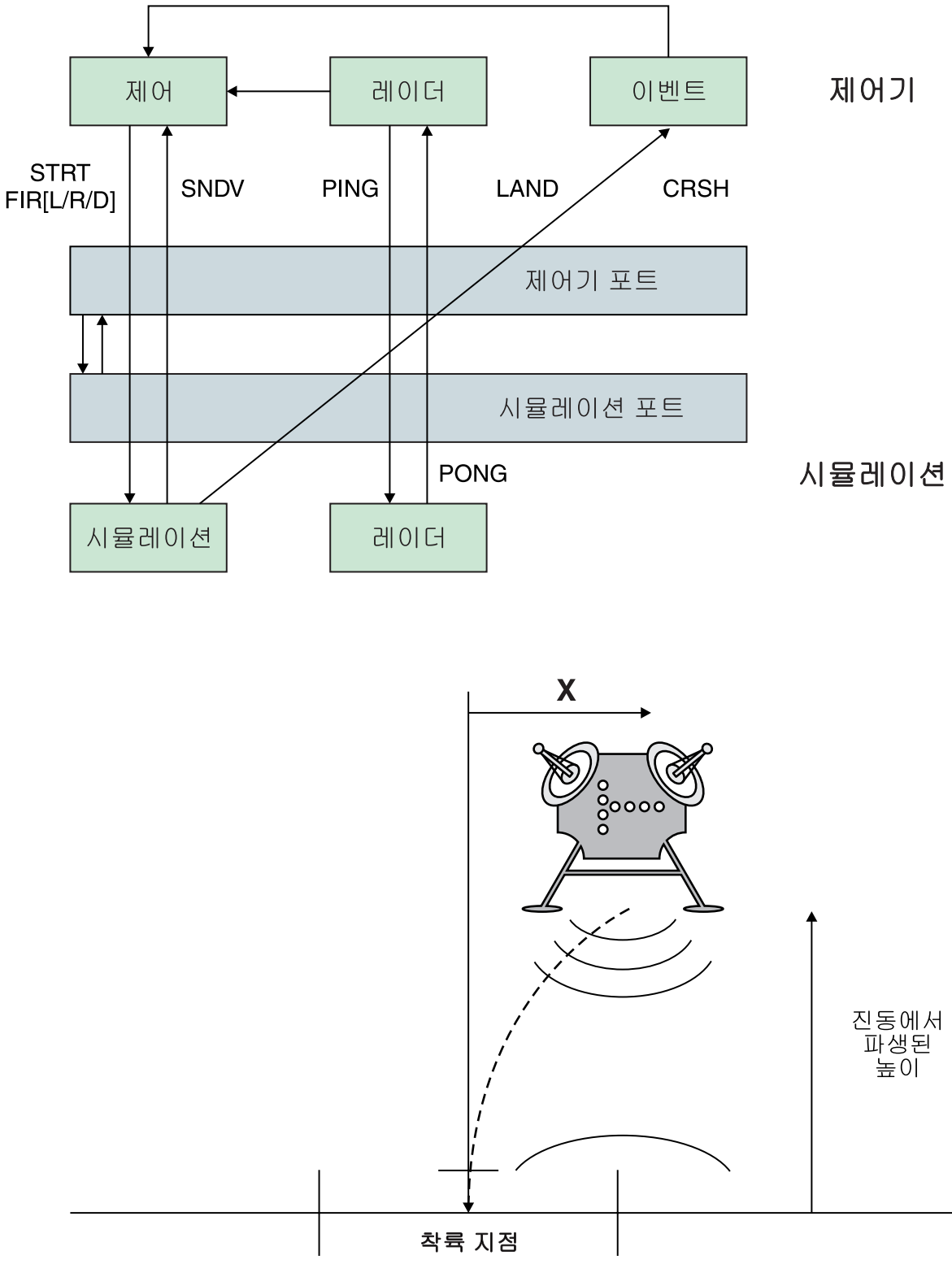

그림 6. 달 착륙기 다이어그램

이 다이어그램은 샘플에서 제공되는 모듈의 관계를 보여줍니다. 다이어그램의 상단에는 제어기 및 시뮬레이터가 표시됩니다. 제어기에는 제어, 레이더 및 이벤트의 3가지 스레

드가 있습니다. 시뮬레이터에는 시뮬레이터 및 레이더의 2가지 스레드가 있습니다. 다이어그램의 하단에는 달 착륙기가 표시되고 착륙기의 높이를 판별하는 진동과 함께 오른쪽 및 왼쪽의 두 제어 기능을 나타냅니다.

샘플 애플리케이션 빌드

참고할 수 있는 샘플 애플리케이션 소스 코드가 제공됩니다. 실행하기 전에 준비 과정에서 Java 소스 코드를 풀고 컴파일해야 합니다.

프로시저

1. 작업 디렉토리를 작성하십시오.
2. 작업 디렉토리에 샘플 애플리케이션을 푸십시오.

   ```
   unzip sample_application.zip
   ```

3. 출력을 위한 새 디렉토리를 작성하십시오.

   ```
   mkdir classes
   ```

4. 소스를 컴파일하십시오.
   a. 파일 목록을 생성하십시오.

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

   b. 소스를 컴파일하십시오.

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

   c. 클래스 파일의 jar 파일을 작성하십시오.

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

다음에 수행할 작업

이제 샘플 애플리케이션을 실행할 수 있습니다.

샘플 애플리케이션 실행

WebSphere Real Time은 표준 JVM 및 **-Xrealtime** 명령행 인수로 시작되는 실시간 JVM을 제공합니다.

샘플 애플리케이션에는 별도의 JVM에서 실행되는 두 개의 컴포넌트가 있습니다.

- Real-Time Java에서만 실행되는 시뮬레이션
- 실시간 이외 또는 실시간 Java에서 실행되는 제어기

제어기 코드를 다양한 모드에서 실행하면 IBM Real-Time Java 기술의 이점을 알 수 있습니다.

Real Time 없이 샘플 애플리케이션 실행

이 프로시저에서 IBM WebSphere Real Time을 사용하지 않고 샘플 애플리케이션을 실행합니다.

시작하기 전에

샘플 애플리케이션을 실행하려면 먼저 샘플 소스 코드를 빌드해야 합니다. 자세한 정보는 웹 사이트 91 페이지의 『샘플 애플리케이션 빌드』의 내용을 참조하십시오.

프로시저

1. 시뮬레이션을 시작하십시오.

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

   이 명령에서, <*port*>는 워크스테이션의 할당 해제된 포트입니다.

2. 제어기를 시작하십시오.

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

   이 명령에서 <*host*>는 시뮬레이션을 실행하는 워크스테이션의 호스트 이름이고 <*port*>는 이전 단계에서 지정된 포트입니다.

결과

애플리케이션이 시뮬레이션 및 제어기가 시작되었음을 표시하는 메시지를 생성합니다.

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

제어기의 몇 가지 값 포인트 샘플이 콘솔에 인쇄됩니다.

```
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
```

이 시뮬레이션이 중지되기 전에 이벤트 요약 메시지가 발행됩니다.

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

포인트 샘플 및 이벤트 요약 메시지 외에, 제어기는 동일한 디렉토리에 graph.svg라는 그래프를 생성합니다. 그래프에는 포인트 샘플의 구상이 포함됩니다. 그래프는 표준 비실시간 JVM으로 애플리케이션을 실행할 때 JavaRadar 스레드에 대한 가비지 콜렉션 일시정지의 영향을 표시합니다. 레이더 높이를 나타내는 데이터에는 스파이크가 있습니다. 스파이크는 제어기 애플리케이션에 영향을 미치는 표준 가비지 콜렉션 일시정

지에 의해서 발생합니다. 일부 실행에서 가비지 콜렉션 일시정지가 장애를 유발할 만큼 장시간 지속되면 다음 메시지가 표시됩니다.

```
CRASH!
```

가비지 콜렉션에 의해 발생한 일시정지 시간을 보려면 **-verbose:gc** 옵션을 제어기 시작 명령에 추가하십시오.

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

메트로놈 가비지 콜렉터를 사용하여 샘플 애플리케이션 실행

-Xrealtime 옵션을 추가하여 코드를 재작성할 필요 없이 실시간 환경에서 표준 Java 애플리케이션을 실행할 수 있습니다. 이 옵션으로 Real-Time Java 언어 기능 및 메트로놈 가비지 콜렉터를 모두 사용할 수 있습니다.

시작하기 전에

샘플 애플리케이션을 실행하려면 먼저 샘플 소스 코드를 빌드해야 합니다. 자세한 정보는 웹 사이트 91 페이지의 『샘플 애플리케이션 빌드』의 내용을 참조하십시오.

프로시저

1. 시뮬레이션을 시작하십시오.

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

   이 명령에서, <*port*>는 워크스테이션의 할당 해제된 포트입니다.

2. 제어기를 시작하십시오.

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

   이 명령에서 <*host*>는 시뮬레이션을 실행하는 워크스테이션의 호스트 이름이고 <*port*>는 이전 단계에서 지정된 포트입니다. 동일한 워크스테이션에서 JVM을 실행하면 동작의 확실성이 떨어질 수 있습니다. 자세한 정보는 웹 사이트 26 페이지의 『고려사항』의 내용을 참조하십시오.

결과

애플리케이션이 실행되고 다음을 포함한 결과물이 생성됩니다.

1. 시뮬레이션 및 제어기가 시작되었음을 알리는 메시지.
2. 제어기 값의 포인트 샘플.
3. 포인트 샘플 구상을 포함하는 동일한 디렉토리 내의 그래프, `graph.svg`.
4. 이벤트 요약 메시지.

메트로놈 가비지 콜렉션으로 애플리케이션을 실행하는 경우, 포인트 샘플 및 해당 그래프는 다음과 같이 표시되는 경향이 있습니다.

- 레이더 높이 데이터에 스파이크가 없습니다.
- 실제 높이 데이터의 추적이 정확합니다.

이는 현재 제어기 코드가 상대적으로 짧은 가비지 콜렉션 일시정지를 사용하여 실행 중이기 때문입니다.

메트로놈 가비지 콜렉션 일시정지의 빈도가 높지만 일반적으로 기간이 1밀리초보다 짧습니다. 비실시간 가비지 콜렉션 일시정지는 적게 수행되지만 일반적으로 수십 또는 수백 밀리초 동안 지속됩니다. 일시정지간의 차이는 제어기 실행 명령에 **-verbose:gc** 옵션을 추가하면 볼 수 있습니다.

상세 가비지 콜렉션 출력에 대한 자세한 정보는 147 페이지의 『verbose:gc 정보 사용』의 내용을 참조하십시오.

AOT를 사용하는 동안 샘플 애플리케이션 실행

이 프로시저에서는 코드를 재작성할 필요 없이 AOT(ahead-of-time) 컴파일러를 사용하는 동안 표준 Java 애플리케이션을 실시간 환경에서 실행합니다. JIT 컴파일러를 사용하는 동안 동일한 애플리케이션 실행을 비교하려면 이 샘플을 사용하십시오.

AOT(Ahead-of-Time) 컴파일에 대한 자세한 정보는 40 페이지의 『WebSphere Real Time for RT Linux에서 컴파일된 코드 사용』의 내용을 참조하십시오.

시작하기 전에

샘플 애플리케이션을 실행하려면 먼저 샘플 소스 코드를 빌드해야 합니다. 자세한 정보는 웹 사이트 91 페이지의 『샘플 애플리케이션 빌드』의 내용을 참조하십시오.

이 태스크 정보

AOT(Ahead-of-Time) 컴파일러는 실행 전에 Java 애플리케이션을 원시 코드에 컴파일합니다. JIT(Just-In-Time) 컴파일로 인한 방해가 없기 때문에 애플리케이션을 보다 정확하게 실행하는 방법을 예측할 수 있습니다.

프로시저

1. 애플리케이션 바이트 코드를 원시 코드로 변환하십시오.
   a. 변환은 일반 JIT 컴파일러로 샘플을 먼저 실행하여 발생합니다.

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

      이 명령에서, *<port>*는 워크스테이션의 할당 해제된 포트입니다.

   b. 다른 창에서 애플리케이션을 실행하십시오.

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

이 명령에서 *<port>*는 이전 단계에 지정한 포트입니다. 애플리케이션 출력 결과는 다음 메시지와 유사합니다.

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

및

```
Land!
```

c. 이전 단계에 작성한 AOT 옵션 파일을 결합하십시오.

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

이전 단계에서 작성한 로그 파일에 사용한 이름에는 파일 이름에 추가된 프로세스 ID 및 날짜 정보가 있습니다. 파일 이름의 형식은 **vlog=** 옵션에 지정되어 있습니다. 예를 들어, **vlog=sim.aotOpts**는 **sim.aotOpts.20081014.234958.13205**와 비슷한 파일 이름을 생성합니다.

d. realtime.jar, vm.jar, rt.jar 및 애플리케이션 demo.jar의 sample.aotOpts 파일에서 해당 파일을 컴파일하십시오. 공유 클래스 캐시를 사용할 경우 캐시 이름이 53자를 초과하지 않아야 합니다.

```
admincache -Xrealtime -populate -cacheName "sample" -aotFilterFile sample.aotOpts
$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
```

컴파일 결과가 보고되었습니다.

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

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

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

Processing complete
```

주: 행:

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

는 "sample"이라는 기존의 캐시가 이 명령으로 제거되고 지정된 캐시가 작성됨을 의미합니다.

e. 채워진 캐시의 컨텐츠를 표시하십시오.

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

2. 시뮬레이션을 시작하십시오.

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

이 명령에서, *<port>*는 이 워크스테이션의 할당 해제된 포트입니다.

3. 제어기를 시작하십시오.

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

이 명령에서 *<host>*는 시뮬레이션을 실행하는 워크스테이션의 호스트 이름이고 *<port>*는 이전 단계에서 지정된 포트입니다. 동일한 워크스테이션에서 JVM을 실행하면 동작의 확실성이 떨어질 수 있습니다. 자세한 정보는 웹 사이트 26 페이지의 『고려사항』의 내용을 참조하십시오.

결과

애플리케이션이 실행되고 다음을 포함한 결과물이 생성됩니다.

1. 시뮬레이션 및 제어기가 시작되었음을 알리는 메시지.
2. 제어기 값의 포인트 샘플.
3. 포인트 샘플 구상을 포함하는 동일한 디렉토리 내의 그래프, `graph.svg`.
4. 이벤트 요약 메시지.

AOT(Ahead-of-Time) 컴파일로 애플리케이션을 실행하는 경우, 포인트 샘플 및 해당 그래프는 다음과 같이 표시되는 경향이 있습니다.

- 레이더 높이 데이터에 스파이크가 없습니다.
- 실제 높이 데이터의 추적이 정확합니다.

이는 제어기 코드가 상대적으로 짧은 가비지 콜렉션 일시정지를 사용하고 JIT(Just-In-Time) 컴파일 방해가 없기 때문입니다.

공유 클래스 캐시를 사용하여 이 애플리케이션을 실행하면 제어기 및 시뮬레이션 JVM 이 두 JVM에서 로드한 클래스에 사용되는 메모리의 일부를 공유하는 이점이 있습니다.

샘플 실시간 해시 맵

WebSphere Real Time for RT Linux에는 IBM SDK for Java 7에서 표준 HashMap 보다 put 메소드에 대해 더 일정한 성능을 제공하는 HashMap 및 HashSet 구현이 포함되어 있습니다.

IBM이 제공하는 표준 java.util.HashMap은 처리량이 높은 애플리케이션에서 잘 작동합니다. 또한 해시 맵이 확장되어야 하는 최대 *크기*를 애플리케이션이 알 수 있도록 지원합니다. 가변 *크기*로 확장할 수 있는 해시 맵이 필요한 애플리케이션의 경우, 사용량에 따라 표준 해시 맵에서 성능 문제점이 발생할 수 있습니다. 표준 해시 맵은 put 메소드를 사용하여 새 항목을 해시 맵에 추가하기 위한 좋은 응답 시간을 제공합니다. 그러나, 해시 맵이 꽉 차면 더 큰 지원 저장소를 할당해야 합니다. 즉, 현재 지원 저장소에 있는 항목을 마이그레이션해야 합니다. 해시 맵이 크면 put 수행 시간이 길어질 수 있습니다. 예를 들어, 이 조작은 몇 밀리초가 걸릴 수 있습니다.

WebSphere Real Time for RT Linux에 샘플 실시간 해시 맵이 포함되어 있습니다. 표준 java.util.HashMap과 동일한 기능 인터페이스를 제공하지만 put 메소드에 대해 보다 일관된 성능을 허용합니다. 지원 저장소를 작성하고 해시 맵이 꽉 차면 모든 항목을 마이그레이션하는 대신, 샘플 해시 맵이 추가 지원 저장소를 작성합니다. 새 지원 저장소는 해시 맵의 다른 지원 저장소에 체인 형식으로 연결됩니다. 체인 형식의 연결로 인해 빈 지원 저장소를 할당하고 다른 지원 저장소에 체인 형식으로 연결하는 동안 초기에 성능이 약간 저하됩니다. 지원 해시 맵을 업데이트하고 나면 모든 항목을 마이그레이션하는 것보다 빨라집니다. 실시간 해시 맵의 단점은 get, put 및 remove 조작이 다소 느려질 수 있다는 점입니다. 각 검색을 단일 항목이 아닌 지원 해시 맵 세트에서 진행해야 하기 때문에 조작이 느려집니다.

실시간 해시 맵을 사용해 보려면 `RTHashMap.jar` 파일을 부트클래스 경로의 시작 부분에 추가하십시오. WebSphere Real Time for RT Linux를 디렉토리 `$WRT_ROOT`에 설치한 경우 표준 해시 맵 대신 다음 옵션을 추가하여 애플리케이션에서 실시간 해시 맵을 사용하도록 하십시오.

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

실시간 해시 맵 구현의 소스 및 클래스 파일이 `demo/realtime/RTHashMap.jar` 파일에 포함되어 있습니다. 또한, 실시간 java.util.LinkedHashMap 및 java.util.HashSet 구현도 제공됩니다.

Eclipse를 사용하여 WebSphere Real Time for RT Linux 애플리케이션 개발

Eclipse를 사용하면 실시간 애플리케이션을 개발할 때 완전한 IDE가 제공됩니다.

시작하기 전에

처음으로 Eclipse 애플리케이션 개발 환경을 사용하여 실시간 애플리케이션을 개발하는 경우 이 프로시저를 사용하여 환경을 구성하십시오.

WebSphere Real Time for RT Linux는 표준 Oracle **javac** 컴파일러를 제공합니다. 사용하는 컴파일러에 대한 제한사항이 없지만 올바른 Java 5.0 클래스 파일을 생성해야 합니다. 그러나, javax.realtime.* Java 클래스가 빌드 경로에 있어야 합니다.

이 태스크 정보

Eclipse에서 애플리케이션을 개발하려면 다음 지시사항을 따르십시오.

프로시저

1. http://www.eclipse.org/downloads/에서 Eclipse를 다운로드하십시오. 올바른 Java 5.0 컴파일을 위해 Eclipse 3.1.2를 사용할 것을 권장합니다.
2. IBM SDK and Runtime Environment for Linux 플랫폼, Java 2 Technology Edition, 버전 5.0 호환 JVM(Eclipse 실행용)을 다운로드하십시오.
3. WebSphere Real Time for RT Linux 패키지에서 `opt/IBM/javawrt3/jre/lib/i386/realtime/jclSC160/realtime.jar` 파일을 푸십시오.
4. Eclipse를 열고 프로젝트를 작성하십시오. **파일** > **새로 작성**을 클릭하십시오. **새 프로젝트** 패널에서 **Java 프로젝트**를 선택하십시오.
5. **다음**을 클릭하여 **새 Java 프로젝트** 패널을 표시하십시오.
   a. 프로젝트 이름을 입력하십시오(예: RTSJ-Tests).
   b. JDK 컴파일러가 5.0으로 설정되었는지 확인하십시오.
6. **완료**를 클릭하십시오.
7. 작업 디렉토리를 작성하고 `opt/IBM/javawrt3/jre/lib/i386/realtime/jclSC160/realtime.jar` 파일을 가져오십시오.
8. **파일** > **새로 작성** > **폴더**를 클릭하여 **새 폴더** 패널을 여십시오. 새 폴더 이름을 입력하십시오(예: *deplib*).
9. **완료**를 클릭하십시오.
10. `realtime.jar` 파일을 가져오려면 **파일** > **가져오기**를 클릭하여 **가져오기** 패널을 여십시오.
11. **파일 시스템** 및 **다음**을 클릭하십시오.
12. 파일 시스템에서 JVM 압축을 푼 `opt/IBM/javawrt3/jre/lib/i386/realtime/jclSC160/` 디렉토리를 여십시오.
13. realtime.jar 파일 옆의 선택란을 선택하고 가져올 폴더를 지정한 다음(예: `RTSJ-Tests/deplib`) **선택한 폴더만 작성** 옵션이 선택되어 있는지 확인하십시오.
14. **완료**를 클릭하십시오.

15. jar 파일을 라이브러리 경로에 추가하십시오. 프로젝트를 마우스 오른쪽 단추로 클릭하고 **특성**을 클릭하여 **특성** 패널을 여십시오.
16. **Java 빌드 경로** 및 **라이브러리** 탭을 클릭하십시오. **Jars 추가**를 클릭하십시오.
17. 프로젝트 디렉토리 아래에서 **realtime.jar**을 클릭하십시오. **확인**을 클릭하십시오.

결과

이 프로시저가 성공하면 realtime.jar 파일이 **라이브러리** 탭의 .jar 파일 목록에 표시됩니다.

예

Eclipse는 `realtime.src.jar`을 사용하여 RTSJ 클래스에 대한 추가 정보를 제공할 수 있습니다. 이를 위해 가져온 `realtime.jar` 파일에 대한 특성 창을 열고 **Java 소스 첨부**를 클릭한 다음 **위치 경로:**에 `realtime.src.jar` 파일 위치를 입력하십시오.

다음에 수행할 작업

Eclipse 기반 Apache Ant를 사용하여 애플리케이션을 빌드하려면 `realtime.jar` 파일을 Ant 빌드 스크립트의 클래스 경로에 추가하십시오. 예를 들면 다음과 같습니다.

```
<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>
```

ant 빌드 스크립트의 일부입니다.

애플리케이션 디버깅

Eclipse 애플리케이션 개발자를 사용하여 애플리케이션을 로컬로 또는 원격으로 디버깅할 수 있습니다.

이 태스크 정보

실시간 애플리케이션을 원격으로 디버깅하려면 디버깅하는 JVM에 다음 옵션이 필요합니다. `-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=10100`

프로시저

1. 애플리케이션이 실행되는 Linux 환경에서 다음을 입력하십시오.

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

여기서, 각 인수는 다음을 의미합니다.

- `server=y`는 JVM이 디버거의 연결을 승인하고 있음을 나타냅니다.
- `suspend=y`는 JVM이 실행 전에 디버거 접속을 대기하도록 설정합니다.
- `address=10100`은 디버거가 JVM에 첨부해야 하는 포트 번호입니다. 이 숫자는 일반적으로 1024 이상입니다.

JVM이 다음 메시지를 표시합니다.

```
Listening for transport dt_socket at address: 10100
```

2. Eclipse에서 애플리케이션을 열고 **디버그**를 선택하십시오.
3. 원격 애플리케이션 디버깅을 위한 새 구성을 작성해야 합니다. 동일한 프로젝트의 애플리케이션이 실행되고 각 실행에서 동일한 포트를 인식하는 경우에만 하나를 작성해야 합니다.
4. 구성을 작성한 경우 구성 이름, 디버깅하는 애플리케이션을 포함한 프로젝트 이름, 애플리케이션이 실행되는 워크스테이션의 *hostname* 및 **-agentlib** 옵션에 전달한 포트 번호를 채우십시오.
5. **디버그**를 클릭하여 디버깅 세션을 시작하십시오. 원격으로 디버깅된 JVM의 상태를 보려면 **디버그** 퍼스펙티브를 열어야 합니다.

JVM으로 Eclipse 실행

이 섹션에서는 WebSphere Real Time for RT Linux JVM으로 Eclipse를 실행하는 방법을 설명합니다.

JVM으로 Eclipse를 실행하려면 "eclipse" 명령에서 다음을 지정해야 합니다.

- 사용할 WebSphere Real Time for RT Linux JVM의 java 실행 파일에 대한 완전한 디렉토리
- **-Xrealtime** JVM 옵션
- Eclipse가 사용할 영구 메모리의 *크기*. 최소한 128M이어야 합니다.

JVM으로 Eclipse 실행 예제

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

주: Eclipse SDK는 WebSphere Real Time for RT Linux 애플리케이션에서 사용 가능한 다양한 실시간 메모리 옵션을 이용하지 않으며, 이에 따른 결과로 Eclipse를 다시 시작하지 않고 몇 시간 또는 몇 일 동안 사용하는 경우 등에 영구 메모리가 소진됩니다. **OutOfMemory** 오류가 발생하면 **-Xgc:immortalMemorySize** 옵션의 값을 높여 Eclipse가 사용할 영구 메모리 양을 늘릴 수 있습니다.

제 7 장 성능

WebSphere Real Time for RT Linux는 가장 높은 처리량 성능 또는 가장 작은 메모리 풋프린트보다 일관되게 짧은 GC 일시정지를 위해 최적화되었습니다.

하이퍼스레딩이 지원되는 시스템에서는 이를 사용하지 않도록 확인해야 합니다. 이는 WebSphere Real Time for RT Linux 사용 시 성능 악영향을 방지하기 위함입니다.

시간 변동을 줄이고 Real-Time Specification for Java(RTSJ)를 지원하려면 몇 가지 표준 IBM Java Runtime 최적화를 사용 불가능하게 해야 합니다. 따라서, 표준 Java 애플리케이션이 **-Xrealtime** 매개변수로 실행될 경우 전체 성능의 저하를 목격할 수 있습니다.

인증된 하드웨어 구성에서의 성능

인증된 시스템에서 WebSphere Real Time for RT Linux 성능 목표를 지원하기에 충분한 클럭 단위 및 프로세서 속도가 제공됩니다. 예를 들어, 적합한 힙 *크기*를 사용하고 오버로드되지 않은 시스템에서 실행 중인 잘 작성된 애플리케이션에서는 일반적으로 1밀리초 또는 약 500마이크로초 이하의 GC 일시정지가 발생합니다. GC 주기 동안 기본 환경 설정의 애플리케이션은 슬라이딩 10 밀리초 시간대 동안 경과 시간의 30% 이상 일시정지되지 않습니다. 10밀리초를 초과하는 GC 일시정지의 집합 시간은 일반적으로 총 3밀리초 이하입니다.

시간 변동 줄이기

WebSphere Real Time for RT Linux에서 표준 JVM에서 두 가지 주요 변동 원인을 다음과 같이 처리합니다.

- Java 코드 준비: 로딩 및 JIT(Just-In-Time) 컴파일을 AOT(Ahead-of-Time) 컴파일에서 처리합니다. 43 페이지의 『AOT 컴파일러 사용』의 내용을 참조하십시오.
- 가비지 콜렉션 일시정지: 표준 가비지 콜렉터 모드에서의 장시간 일시정지는 메트로놈 가비지 콜렉터를 사용하면 방지할 수 있습니다. 72 페이지의 『메트로놈 가비지 콜렉터 사용』의 내용을 참조하십시오.

실시간 이외 모드에서 JVM 사이의 클래스 데이터 공유

클래스 공유는 실시간 모드가 아닌 경우에도 지원되지만 실시간 모드인 경우와 다르게 작동합니다.

공유 클래스 데이터를 디스크의 메모리 맵핑된 캐시 파일에 저장하여 JVM(Java Virtual Machine)사이에서 해당 데이터를 공유할 수 있습니다. 둘 이상의 JVM이 캐시를 공유

하는 경우 데이터를 공유하면 전체 가상 스토리지 이용이 줄어듭니다. 또한 데이터를 공유하면 캐시를 작성한 후 JVM의 시작 시간도 단축됩니다. 공유 클래스 캐시는 실행 중인 JVM에 종속되지 않으며 삭제할 때까지 지속됩니다.

공유 캐시에는 다음이 포함될 수 있습니다.

- 부트스트랩 클래스
- 애플리케이션 클래스
- 클래스를 설명하는 메타데이터
- AOT(Ahead-of-time) 컴파일된 코드

제 8 장 보안

이 섹션에는 보안에 대한 중요한 정보가 있습니다.

공유 클래스 캐시의 보안 고려사항

공유 클래스 캐시는 각 캐시 관리 및 사용을 위해 설계되었지만 기본 보안 정책이 적합하지 않을 수 있습니다.

공유 클래스 캐시를 사용할 때는 액세스 제한으로 보안이 강화되도록 사용자가 새 파일에 대한 기본 권한을 알고 있어야 합니다.

파일	기본 권한
새 공유 캐시	그룹 및 other의 읽기 권한
javasharedresources 디렉토리	읽기, 쓰기 및 실행 권한

캐시를 제거하거나 확장하려면 캐시 파일 및 캐시 디렉토리 둘 다에 대한 쓰기 권한이 필요합니다.

캐시 파일의 파일 권한 변경

공유 클래스 캐시에 대한 액세스를 제한하려면 **chmod** 명령을 사용합니다.

필요한 변경	명령
사용자 및 그룹으로 액세스 제한	chmod 770 /tmp/javasharedresources
사용자로 액세스 제한	chmod 700 /tmp/javasharedresources
사용자를 특정 캐시 전용의 읽기 및 쓰기 액세스로 제한	chmod 600 /tmp/javasharedresources/<*file for shared cache*>
사용자 및 그룹을 특정 캐시 전용의 읽기 및 쓰기 액세스로 제한	chmod 660 /tmp/javasharedresources/<*file for shared cache*>

공유 클래스 캐시 작성에 대한 자세한 정보는 45 페이지의 『실시간 공유 클래스 캐시 작성』의 내용을 참조하십시오.

액세스 권한이 없는 캐시에 연결

올바른 액세스 권한이 없는 캐시에 연결하려고 하면 오류 메시지가 표시됩니다.

```
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.
```

제 9 장 문제점 해결 및 지원

WebSphere Real Time for RT Linux에 대한 문제점 해결 및 지원 방법을 설명합니다.

- 『일반 문제점 판별 메소드』
- 112 페이지의 『OutOfMemory 오류 문제점 해결』
- 123 페이지의 『진단 도구 사용』

일반 문제점 판별 메소드

문제점 판별은 사용자가 결함의 종류와 적절한 조치 순서에 대해 이해할 수 있도록 도와줍니다.

문제점의 종류가 무엇인지 알고 있으면 다음 태스크를 수행할 수 있습니다.

- 문제점을 수정하십시오.
- 유용한 해결 방법을 찾으십시오.
- IBM으로 버그 보고서를 생성하는 데 필요한 데이터를 콜렉트하십시오.

Linux 문제점 판별

이 섹션에서는 Linux에서의 문제저 판별에 대해 설명합니다.

IBM SDK for Java 7 사용자 안내서에는 Linux에서 발생하는 문제점 진단에 필요한 정보가 있으며 다음 내용을 다루고 있습니다.

- Linux 환경 설정 및 확인
- 일반 디버깅 기술
- 크래쉬 진단
- 정지 디버깅
- 메모리 누수 디버깅
- 성능 문제 디버깅

이 정보는 여기에서 찾을 수 있습니다: IBM SDK for Java 7 - Linux 문제 판별.

다음 정보는 IBM WebSphere Real Time for RT Linux에 대한 보충 정보입니다.

Linux 환경 설정 및 확인

IBM WebSphere Real Time for RT Linux에서 JVM이 시스템 덤프를 생성하도록 올바르게 구성되어 있는지 확인합니다.

Linux 시스템 덤프(코어 파일)

클래쉬가 발생하는 경우, 획득해야 하는 가장 중요한 진단 데이터는 Linux 시스템 덤프(코어 파일)입니다. 이 파일이 생성되도록 하려면 IBM SDK for Java 7 사용자 안내서에 설명된 대로 운영 체제 설정과 사용 가능한 디스크 공간을 확인해야 합니다.

JVM(Java Virtual Machine) 설정

크래쉬가 발생할 때 코어 파일을 생성할 수 있도록 JVM을 구성해야 합니다. 명령행에서 `java -Xrealtime -Xdump:what`을 실행하십시오. 이 옵션의 출력은 다음과 같습니다.

```
-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
```

표시된 값이 기본 설정입니다. 최소한 `events=gpf`가 설정되어야 시스템 크래쉬 발생 시에 코어 파일을 생성할 수 있습니다. 명령행 옵션 **`-Xdump:system[:name1=value1,name2=value2 ...]`**를 사용하여 옵션을 변경하고 설정할 수 있습니다.

일반 디버깅 기술

운영 체제에서 Java 스레드 이름이 보이므로 ps 명령을 사용하여 디버깅을 지원할 수 있습니다. 추적 도구를 사용할 때 IBM WebSphere Real Time for RT Linux에 대해 올바른 명령을 사용해야 합니다.

프로세스 정보 조사

IBM WebSphere Real Time for RT Linux에서 ps 명령을 실행할 때 표시될 수 있는 예상 출력은 다음과 같습니다.

```
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 모든 프로세스를 선택합니다.

L 스레드를 표시합니다.

o 표시할 열의 사전 정의된 형식을 제공합니다. 지정된 열은 프로세스 ID, 스레드 ID, 스케줄 정책, 실시간 스레드 우선순위 및 프로세스와 연관된 명령입니다. 이 정보는 가상 시스템뿐 아니라 사용자 애플리케이션에서 주어진 시간에 어떤 스레드가 실행 중인지를 이해하는 데 유용합니다.

추적 도구

Linux의 세 가지 추적 도구는 **strace**, **ltrace** 및 **mtrace**입니다. 명령 `man strace`는 사용 가능한 전체 옵션 세트를 표시합니다.

strace

strace 도구는 시스템 호출을 추적합니다. 사용 가능한 프로세스에서 이를 사용하거나 새 프로세스로 이를 시작할 수 있습니다. strace는 프로그램에서 작성된 시스템 호출 및 프로세스에서 수신한 신호를 기록합니다. 각 시스템 호출에 대해 이름, 인수 및 리턴 값이 사용됩니다. strace를 사용하면 소스 없이도 프로그램을 추적할 수 있습니다(다시 컴파일할 필요 없음). strace를 **-f** 옵션과 함께 사용할 경우 포크된 시스템 호출의 결과로 작성된 하위 프로세스를 추적합니다. Plug-in 문제점을 조사하거나 프로그램이 제대로 시작되지 않은 이유를 파악할 때 strace를 사용합니다.

Java 애플리케이션과 함께 strace를 사용하려면 `strace java -Xrealtime <class-name>`을 입력하십시오.

`-o` 옵션을 사용하여 strace 도구를 통한 추적 출력을 파일에 넣을 수 있습니다.

ltrace

ltrace 도구는 배포 의존적으로, strace와 매우 유사합니다. 이 도구는 실행 프로세스에 의해 호출된 대로 동적 라이브러리 호출을 가로채어 기록합니다. strace는 실행 프로세스에 의해 수신되는 신호에 대해 동일합니다.

Java 애플리케이션과 함께 Itrace를 사용하려면 `ltrace java -Xrealtime <class-name>`를 입력하십시오.

mtrace

mtrace는 GNU 도구 세트에 포함되어 있습니다. 이는 malloc, realloc 및 free용 특수 핸들러를 설치하며, 이러한 기능의 모든 사용을 추적하고 파일에 기록하도록 해줍니다. 이 추적은 프로그램 효율성을 떨어뜨리므로 일반 사용 중에는 사용하지 않도록 하십시오. mtrace를 사용하려면 **IBM_MALLOCTRACE**를 1로 설정하고 **MALLOC_TRACE**를 추적 정보가 저장되는 올바른 파일을 가리키도록 설정하십시오. 이 파일에 대한 쓰기 액세스가 있어야 합니다.

Java 애플리케이션과 함께 mtrace를 사용하려면 다음을 입력하십시오.

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

크래쉬 진단

클래쉬 이전의 Java 환경 및 실행 프로세스에 대한 정보를 수집하려면 다음 가이드라인을 수행하십시오.

프로세스 정보 수집

크래쉬가 발생하기 전의 상태를 수집할 때, 코어 파일을 분석하는 대신 **gdb** 및 **bt** 명령을 사용하여 실패한 스레드의 스택 추적을 표시하십시오.

Java 환경 정보 찾기

각 스레드가 어떤 작업을 수행 중이었고 어떤 Java 메소드가 실행 중이었는지 판별하려면 Javadump를 사용하십시오. 다양한 지점에서 실행 중인 코드의 소스를 판별하려면 라이브러리 주소에 대해 함수 주소를 일치시키십시오.

-verbose:gc 옵션을 사용하여 Java 힙과 영구 및 범주 메모리 영역의 상태를 확인하십시오. 다음 질문을 고려하십시오.

- 메모리 영역 중 한 곳에 메모리가 부족한지, 그리고 이것이 크래쉬의 원인이 되었는지 여부.
- 가비지 콜렉션 동안 가능한 가비지 콜렉션 결함을 나타내는 크래쉬가 발생했는지 여부.
- 가비지 콜렉션 후에 가능한 메모리 충돌을 나타내는 크래쉬가 발생했는지 여부.

성능 문제 디버깅

성능 문제를 디버깅할 때, IBM SDK for Java 7 사용자 안내서 외에도 이러한 IBM WebSphere Real Time for RT Linux의 특정 항목을 고려하십시오.

메모리 영역 크기 조정

JVM은 힙, 영구 및 범위 메모리 *크기*에 따라 조정할 수 있습니다. 성능을 최적화하려면 올바른 *크기*를 선택하십시오. 올바른 *크기*를 사용하면 가비지 콜렉터가 필요한 용도를 제공하기가 더 쉬워집니다.

메모리 영역의 *크기* 다양화에 대한 자세한 정보는 147 페이지의 『메트로놈 가비지 콜렉터 문제점 해결』의 내용을 참조하십시오.

JIT 컴파일 및 성능

JIT를 사용할 경우, 실시간 동작을 위한 구현을 고려해야 합니다.

예측 가능한 동작을 필요로 하지만 보다 나은 성능도 필요한 경우 AOT(Ahead-Of-Time) 컴파일 사용을 고려해야 합니다. 자세한 정보는 40 페이지의 『WebSphere Real Time for RT Linux에서 컴파일된 코드 사용』의 내용을 참조하십시오.

Linux의 알려진 제한사항

Linux는 빠르게 진화해 왔으며 그에 따라 특히 스레드 영역에서 JVM과 운영 체제의 상호작용과 관련하여 다양한 문제가 생겨났습니다.

Linux 시스템에 영향을 줄 수 있는 다음 제한사항에 주의하십시오.

프로세스로서의 스레드

Java 스레드의 수가 허용되는 최대 프로세스 수를 초과하는 경우 프로그램이 다음과 같이 될 수 있습니다.

- 오류 메시지를 수신합니다.
- **SIGSEGV** 오류가 발생합니다.
- 중지됩니다.

자세한 정보는 http://www.volano.com/report/index.html의 *Volano 보고서*를 참조하십시오.

플로팅 스택 제한사항

플로팅 스택 없이 실행 중인 경우 **-Xss**에 설정된 값에 관계없이 각 스레드에 대해 최저 256KB의 원본 스택 *크기*가 제공됩니다.

플로팅 스택 Linux 시스템에서는 **-Xss** 값이 주어집니다. 플로팅이 아닌 Linux 시스템에서 마이그레이션하는 경우, **-Xss** 값이 충분히 큰지 그리고 최소 256KB에 의존하고 있지 않은지 확인하십시오.

glibc 제한사항

__bzero 같은 기호를 찾을 수 없으므로 libjava.so 라이브러리를 로드할 수 없다는 메시지가 표시된다는 것은 이전 버전의 GNU C 런타임 라이브러리인 glibc가 설치되어 있다는 의미입니다. Linux 스레드 구현용 SDK를 사용하려면 glibc 버전 2.3.2 이상이 있어야 합니다.

글꼴 제한사항

Red Hat 시스템에 설치할 경우 글꼴 서버로 Java 트루타입 글꼴을 찾고 다음을 실행할 수 있도록 하십시오(예: Linux IA32).

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

이는 설치 시 수행해야 하며 명령을 실행하려면 『루트』로 로그인되어 있어야 합니다. 글꼴 문제에 대한 자세한 정보는 *Linux SDK and Runtime User Guide*를 참조하십시오.

Linux Red Hat MRG 커널의 성능 문제

Red Hat MRG 커널과 관련된 구성 문제가 발생하면 WebSphere Real Time이 사용 가능한 상세한 가비지 콜렉션을 시작할 때 애플리케이션 스레드가 예기치 않게 일시정지될 수 있습니다. 이러한 일시정지는 상세한 GC 출력에 기록되지는 않지만 네트워크 구성에 따라 수 밀리초 동안 지속될 수 있습니다. 원격으로 정의된 LDAP 사용자로부터 시작된 JVM은 이름 서비스 캐시 디먼(nscd)이 지원되지 않으므로 네트워크 지연에 영향을 줄 수 있습니다. nscd를 시작하여 문제점을 해결하십시오. nscd 서비스의 상태를 확인하고 문제점을 정정하려면 다음 단계를 따르십시오.

1. 다음 명령을 입력하여 nscd 디먼이 실행 중인지 검사하십시오.

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

   디먼이 실행되고 있지 않으면 다음 메시지가 표시됩니다.

   ```
   nscd is stopped
   ```

2. 루트 사용자로서 다음 명령으로 nscd 서비스를 시작하십시오.

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

3. 루트 사용자로서 다음 명령으로 nscd 서비스에 대한 시작 정보를 변경하십시오.

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

nscd 프로세스는 현재 실행 중이며 다시 부팅한 후에 자동으로 시작됩니다.

NLS 문제점 판별

JVM에는 다른 로케일에 대한 기본 제공 지원이 포함되어 있습니다.

IBM SDK for Java 7 사용자 안내서에는 NLS 문제점 진단에 필요한 정보가 있으며 다음 내용을 다루고 있습니다.

- 글꼴 개요
- 글꼴 유틸리티
- 공통 NLS 문제점 및 가능한 원인

이 정보는 여기에서 찾을 수 있습니다: IBM SDK for Java 7 - NLS 문제점 판별.

ORB 문제점 판별

ORB 문제점을 디버깅하는 경우 첫 번째 태스크 중 하나는 문제점이 분산 애플리케이션의 클라이언트 측에 있는지 또는 서버 측에 있는지 여부를 판별하는 것입니다. 일반 RMI-IIOP 세션을 오브젝트 액세스를 요청하는 클라이언트와 액세스를 제공하는 서버 사이의 단순 동기 통신으로 간주하십시오.

IBM SDK for Java 7 사용자 안내서에는 ORB 문제점 진단에 필요한 정보가 있으며 다음 내용을 다루고 있습니다.

- ORB 문제 식별
- 스택 추적 해석
- ORB 추적 해석
- 공통 문제점
- IBM ORB 서비스: 데이터 콜렉트

이 정보는 여기에서 찾을 수 있습니다: IBM SDK for Java 7 - ORB 문제점 판별.

다음 정보는 IBM WebSphere Real Time for RT Linux에 대한 보충 정보입니다..

IBM ORB 서비스: 데이터 콜렉트

서비스의 Java 버전 출력을 수집할 때 다음 명령을 실행하십시오.

```
java -Xrealtime -version
```

사전 테스트

문제가 발생하는 경우 ORB가 다음 사항을 포함하는 `org.omg.CORBA.*` 예외를 생성할 수 있습니다.

- 원인을 표시하는 텍스트
- 보조 코드
- 완료 상태

ORB가 문제점의 원인이라고 가정하기 전에 다음을 먼저 확인하십시오.

- 유사한 구성으로 시나리오를 재생성할 수 있어야 합니다.

- JIT가 사용 안함으로 설정되어 있어야 합니다.
- AOT 컴파일 코드를 사용하지 않아야 합니다.

기타 조치에는 다음이 포함됩니다.

- 추가 프로세서를 끄십시오.
- 가능한 경우 SMT(Simultaneous Multithreading)를 끄십시오.
- 클라이언트 또는 서버와의 메모리 종속성을 제거하십시오. 실제 메모리 부족으로 인해 성능 저하, 명백한 정지 또는 크래시가 발생할 수 있습니다. 이러한 문제점을 제거하려면 메모리에 적절한 여유 공간이 있는지 확인하십시오.
- 실제 네트워크 문제점(예: 방화벽, 통신 링크, 라우터, DNS 이름 서버 등)을 확인하십시오. 이러한 문제점은 CORBA COMM_FAILURE 예외의 주요 원인입니다. 테스트로 자체 워크스테이션 이름에 대해 Ping을 실행하십시오.
- 애플리케이션이 DB2®와 같은 데이터베이스를 사용하는 경우 가장 신뢰할 수 있는 드라이버로 전환하십시오. 예를 들어, DB2 AppDriver를 분리하려면 속도가 더 느리고 소켓을 사용하지만 더 신뢰할 수 있는 네트 드라이버로 전환하십시오.

OutOfMemory 오류 문제점 해결

OutOfMemoryError 예외, 메모리 누수 및 숨겨진 메모리 할당을 처리합니다.

메트로놈 가비지 콜렉터를 사용하는데 관한 일반적인 문제점 해결 정보는 147 페이지의 『메트로놈 가비지 콜렉터 문제점 해결』의 내용을 참조하십시오.

OutOfMemoryError 진단

메트로놈 가비지 콜렉터에서 OutOfMemoryError 예외를 진단하는 것은 가비지 콜렉터의 정기적인 특성 때문에 표준 JVM보다 더 복잡할 수 있습니다.

다양한 유형의 힙에 대한 특성이 14 페이지의 『메모리 관리』에 설명되어 있습니다. 일반적으로 RTSJ 애플리케이션은 표준 Java 애플리케이션보다 대략 20% 추가 힙 공간을 필요로 합니다.

기본적으로 JVM은 발견되지 않은 OutOfMemoryError 발생 시 다음과 같은 진단 출력을 생성합니다.

- 스냅 덤프: 123 페이지의 『덤프 에이전트 사용』 참조
- 힙 덤프: 133 페이지의 『힙 덤프 사용』 참조
- Java 덤프: 127 페이지의 『Java 덤프 사용』 참조
- 시스템 덤프: 137 페이지의 『시스템 덤프 및 덤프 뷰어 사용』 참조

덤프 파일 이름이 콘솔 출력에 제공됩니다.

```
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".
```

콘솔 출력에 표시되고 Java 덤프에서도 사용 가능한 Java 백추적은 Java 애플리케이션에서 OutOfMemoryError가 발생한 위치를 나타냅니다. 다음 단계는 꽉 찬 RTSJ 메모리 영역을 찾는 것입니다. JVM 메모리 관리 구성요소가 할당 실패한 메모리 공간 이름, *크기*, 클래스 블록 주소를 제공하는 추적포인트를 발행합니다. 이 추적포인트는 스냅 덤프에 있습니다.

```
<< 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
```

추적포인트 ID 및 데이터 필드는 할당하는 오브젝트의 유형에 따라 다를 수 있습니다. 이 예제에서 추적포인트는 애플리케이션이 `Metronome heap`, 메모리 세그먼트 `id=0x809c5f0`에 `class 0x81312d8` 유형의 33.6MB 오브젝트를 할당하려고 할 때 할당 장애가 발생했음을 보여줍니다.

Java 덤프에서 메모리 관리 정보를 확인하여 영향을 받는 RTSJ 메모리 영역을 판별할 수 있습니다.

```
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 >>
```

Java 덤프의 클래스 섹션을 확인하여 할당되는 오브젝트의 유형을 판별할 수 있습니다.

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

Java 덤프의 정보는 시도된 할당이 정상 힙 (ID=0xF288B584)에서 문자 배열에 대한 것이 해당 1STHEAP 행에 표시된 힙의 총 할당된 *크기*가 67108864 10진수 바이트 또는 0x04000000 16진수 바이트 또는 64MB임을 확인합니다.

이 예제에서 총 힙 *크기*와 비교하여 실패한 할당이 큽니다. 애플리케이션이 33MB 오브젝트를 작성할 것으로 예상되면 다음 단계는 **-Xmx** 옵션을 사용하여 힙의 *크기*를 늘리는 것입니다.

총 힙 *크기*와 비교하여 실패한 할당이 작은 것이 일반적입니다. 힙을 채우는 이전 할당 때문입니다. 이 경우, 다음 단계는 힙 덤프를 사용하여 기존 오브젝트에 할당되는 메모리의 양을 조사하는 것입니다.

힙 덤프는 오브젝트 클래스, *크기* 및 참조를 포함한 모든 오브젝트 목록이 있는 압축된 2진 파일입니다. Java용 메모리 덤프 진단 도구(MDD4J)를 사용하여 힙 덤프를 분석하십시오.(IBM Support Assistant(ISA)에서 다운로드할 수 있음).

MDD4J를 사용하여 대량의 힙 공간 이용이 의심되는 오브젝트의 트리 구조를 찾고 힙 덤프를 로드할 수 있습니다. 도구에서 힙의 오브젝트에 대한 다양한 보기를 제공합니다. 예를 들어, MDD4J는 누수 의심에 대해 설명하는 보기를 표시할 수 있으며 힙 크기에 기여한 상위 5개 오브젝트 및 패키지를 알려줍니다. 트리 보기를 선택하면 누수 컨테이너 오브젝트의 특성에 대한 자세한 정보가 제공됩니다.

기본적으로 모든 RTSJ 메모리 공간의 모든 오브젝트가 들어 있는 단일 힙 덤프 파일이 생성됩니다. 각 메모리 공간에 대해 별도의 힙 덤프를 요청하려면 명령행 옵션 **-Xdump:heap:request=multiple**을 사용하십시오. 여러 개의 덤프를 사용하면 특정 메모리 영역에 할당된 오브젝트 세트만 검사할 수 있습니다. 콘솔 출력에 제공된 파일 이름별로 힙 덤프를 식별합니다.

```
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 JVM의 메모리 관리 방식

IBM JVM은 클래스, 컴파일된 코드, Java 오브젝트, Java 스택, JNI 스택용 메모리 영역을 포함하여 다양한 구성요소에 대한 메모리가 필요합니다. 이 메모리 영역 중 일부는 인접 메모리에 있어야 합니다. 기타 메모리 영역은 더 작은 메모리 영역으로 세그먼트화하고 서로 링크할 수 있습니다.

동적으로 로드된 클래스 및 컴파일된 코드는 동적으로 로드된 클래스용으로 세그먼트화된 메모리 영역에 저장됩니다. 클래스는 쓰기 가능 메모리 영역(RAM 클래스) 및 읽기 전용 메모리 영역(ROM 클래스)로 다시 세분화됩니다. 런타임에 클래스 캐시가 맵핑된 메모리이지만, 애플리케이션 시작 시 인접 메모리 영역으로 반드시 로드되는 것은 아닙니다. 애플리케이션이 클래스를 참조하므로 클래스 및 클래스 캐시의 컴파일된 코드가 저장 공간에 맵핑됩니다. 클래스의 ROM 구성요소는 이 클래스를 참조하는 여러 프로세스 간에 공유됩니다. JVM이 클래스를 처음 참조할 때 클래스의 RAM 구성요소가 동적으로 로드된 클래스의 세그먼트화된 메모리 영역에 작성됩니다. 클래스 캐시의 클래스 메소드에 대한 AOT 컴파일된 코드는 프로세스에서 공유하지 않으므로 실행 가능한 동적 코드 메모리 영역에 복사됩니다. 클래스 캐시에서 로드되지 않은 클래스는 ROM 클래스 정보가 동적으로 로드된 클래스의 세그먼트화된 메모리 영역에 작성되는 점을 제외하고 캐싱된 클래스와 비슷합니다. 동적으로 생성된 코드는 캐싱된 클래스의 AOT 코드가 저장된 동일한 동적 코드 메모리 영역에 저장됩니다.

-Xrealtime 옵션 없이 JVM을 실행할 경우 모든 Java 오브젝트가 표준 힙 메모리에 저장됩니다. **-Xrealtime** 옵션을 사용하면 영구 메모리 및 범위 메모리라는 두 개의 추가 메모리 영역에서 오브젝트가 할당될 수도 있습니다.

각 Java 스레드의 스택이 세그먼트화된 메모리 영역에 걸쳐 있습니다. 각 스레드의 JNI 스택이 인접 메모리 영역을 차지합니다.

JVM의 구성 방식을 판별하려면 **-verbose:sizes** 옵션을 사용하여 실행하십시오. 이 옵션은 *크기*를 관리할 수 있는 메모리 영역에 대한 정보를 인쇄합니다. 인접하지 않은 메모리 영역의 경우, 영역을 확장해야 될 때마다 획득되는 메모리의 *크기*를 설명하는 증분 정보가 인쇄됩니다.

다음은 **-Xrealtime -verbose:sizes** 옵션을 사용한 예제 출력입니다.

```
-Xmca32K                          RAM class segment increment
-Xmco128K                         ROM class segment increment
-Xms64M                           initial memory size-Xgc:immortalMemorySize=16M     immorta
-Xgc:scopedMemoryMaximumSize=8M scoped memory space maximum size
-Xmx64M                           memory maximum
-Xmso256K                         operating system thread stack size
-Xiss2K                           java thread stack initial size
-Xssi16K                          java thread stack increment
-Xss256K                          java thread stack maximum size
```

이 예제는 RAM 클래스 세그먼트가 *초기*에 0이고 필요에 따라 32KB 블록씩 확장되는 것을 보여줍니다. ROM 클래스 세그먼트가 *초기*에 0이고 필요에 따라 128KB 블록씩 확장됩니다. **-Xmca** 및 **-Xmco** 옵션을 사용하면 이 *크기*를 제어할 수 있습니다. 대개 이 옵션을 변경할 필요가 없도록 RAM 클래스 및 ROM 클래스 세그먼트가 필요에 따라 확장됩니다.

영구 메모리는 인접 영역으로 더 큰 공간에 사전 할당되어야 합니다. 이 예제에서는 영구 메모리 영역이 16MB로 사전 할당됩니다. 16MB 이상의 오브젝트를 영구 메모리 영역에 쓸 경우, 이 메모리 영역이 가비지 수집되지 않기 때문에 OutOfMemory 예외를 수신하게 됩니다.

범위 메모리 영역은 인접하며 이 예제에서는 8MB로 사전 할당됩니다. 프로그램을 실행할 때 많은 범위 메모리 영역이 활성인 경우 범위 메모리 영역을 더 *크*게 지정해야 할 수도 있습니다.

클래스 캐시를 사용할 경우 메모리 맵핑된 영역의 *크기*를 판별하려면 admincache 유틸리티를 사용하십시오. 다음은 `admincache -Xrealtime -printStats -nologo` 명령의 출력 샘플입니다.

```
J9 Java(TM) admincache 1.0

Current statistics for cache "sharedcc_localuser":


base address       = 0xA52B4000
end address        = 0xA59B7000
allocation pointer = 0xA59B4000

cache size         = 7356040
free bytes         = 330604
ROMClass bytes     = 3798460
AOT bytes          = 3101560
Data bytes         = 3812
Metadata bytes     = 121604
```

```
Metadata % used  = 1%

# ROMClasses       = 1044
# AOT Methods      = 1652
# Classpaths       = 2
# URLs             = 1
# Tokens         = 0
# Stale classes = 0
% Stale classes  = 0%

Cache is 95% full
```

캐시 크기는 메모리 맵핑된 영역이 공간에서 7MB를 약간 초과함을 보여줍니다. ROM 클래스 및 AOT 바이트가 각각 3MB를 약간 초과하여 이 공간의 대부분을 차지합니다.

영구 메모리 공간의 예제 OutOfMemoryError

이 예제는 영구 메모리 공간에서 OutOfMemoryError를 식별하는 방법과 문제점 방지를 위해 수행하는 단계를 설명합니다.

스냅 덤프는 영구 메모리 영역 id=0x809dd1c에서 두 개의 작은 할당 요청이 실패했음을 보여줍니다.

```
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
```

Java 덤프는 영구 메모리 공간이 꽉 찼음을 보여줍니다.

```
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 분석은 매우 큰 LinkedList가 할당되어 사용 가능한 메모리의 많은 부분을 이용하고 있음을 보여줍니다.

영구 영역의 오브젝트는 가비지 수집되지 않으므로 영구 메모리 영역에 할당되는 오브젝트의 수를 최소화하는 것이 좋습니다. 가장 일반적인 영구 메모리 용도는 JVM 및 애플리케이션 초기화 중에 주로 발생하는 무한 활동인 클래스 로딩입니다. 로드된 클래스(또는 기타 영구 메모리 사용)가 많은 애플리케이션은

-Xgc:immortalMemorySize=<size> 옵션을 사용하여 영구 메모리 영역의 *크기*를 늘릴 수 있습니다. 영구 메모리 영역의 기본 *크기*는 16MB입니다.

영구 메모리 영역의 *크기*를 늘리면 영구 메모리에 대해서만 OutOfMemoryError가 지연되고 클래스 로딩 또는 기타 애플리케이션 오브젝트와 관련된 영구 데이터의 지속된 할당 패턴을 조사합니다.

범위 메모리 공간의 예제 OutOfMemoryError

이 예제는 범위 메모리 공간에서 OutOfMemoryError를 식별하는 방법과 문제점 방지를 위해 수행하는 단계를 설명합니다.

각 메모리 공간에 대해 별도의 덤프를 생성하려면 명령행 옵션 **-Xdump:heap:request=multiple**을 사용하십시오.

```
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... >>
```

스냅 덤프는 범위 메모리 영역 id=0x809dd10에서 두 개의 할당 요청이 실패했음을 보여줍니다.

```
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
```

Java 덤프는 id=0x809dd10를 포함한 범위 메모리 영역에 대해 할당된 메모리 영역 *크기*가 매우 작음을 보여줍니다(60KB). 이 경우, 애플리케이션 코드에서 범위 메모리 영역의 *크기*를 늘리십시오.

```
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
```

예제 Java 덤프에서 범위 메모리 영역이 비어 있는 것으로 보입니다. OutOfMemoryError가 JVM에 도달할 때(범위가 종료되고 정리됨) Java 덤프가 생성되므로 비어 있는 것처럼 보입니다. **-Xdump:java:events=throw,filter=java/lang/OutOfMemoryError** 명령행 옵션을 사용하여 장애 지점에 Java 덤프를 생성할 수 있습니다. 이 옵션을 사용하면 범위 메모리 영역의 여유 공간이 올바르게 보고됩니다.

범위 메모리에 사용 가능한 총 공간을 모두 사용할 수도 있습니다. 이 경우, 명령행 옵션 **-Xgc:scopedMemoryMaximumSize=<size>**를 사용하여 범위 메모리 영역의 크기를 늘리십시오. 범위 메모리 영역의 기본 크기는 8MB입니다. 범위 메모리에 사용 가능한 총 범위를 모두 사용한 경우 콘솔에 다른 메시지가 표시됩니다. 예:

```
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)
```

여러 힙에서 문제점 진단

Java 덤프에 있는 주소 범위와 힙 덤프에 있는 거주 정보를 함께 사용하면 여러 RTSJ 메모리 영역에서 OutOfMemoryError를 분석하는 데 도움이 됩니다.

이 Java 덤프에서 영구 세그먼트의 범위는 0xB281C008에서 0xB381C008까지이고, 정상 힙 세그먼트의 범위는 0xB381D008에서 0xB781D008까지입니다.

```
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
```

힙 덤프는 오브젝트 클래스, 크기 및 참조를 포함한 모든 오브젝트 목록이 있는 압축된 2진 파일입니다. Java용 메모리 덤프 진단 도구(MDD4J)를 사용하여 힙 덤프를 분석하십시오.(IBM Support Assistant(ISA)에서 다운로드할 수 있음).

MDD4J가 나열하는 오브젝트 메모리 위치를 사용하여 오브젝트가 있는 메모리 공간을 판별할 수 있습니다. 0xB28*nnnnn* 범위의 주소는 영구 메모리 영역에 있습니다. 0xB61*nnnnn* 범위의 주소는 일반 힙에 있습니다.

메모리 누수 방지

가비지 콜렉터는 영구 또는 범위 메모리 영역을 처리하지 않습니다. 영구 메모리의 경우, JVM이 종료될 때에만 메모리가 해제됩니다. 범위 메모리 영역은 참조 계수가 0이 된 후에만 해제됩니다. 이 컨텍스트에서 오랫동안 실행되는 태스크는 태스크를 준비한 후 영구 메모리 영역에서 추가 메모리를 할당하지 않는 방식으로 작성해야 합니다.

클래스 로딩은 영구 메모리를 적게 사용합니다. 실시간 환경에서 이 클래스는 가비지 수집되지 않습니다. 마찬가지로, 애플리케이션에 필요하지 않은 클래스 로딩으로 인해 애플리케이션이 필요한 것보다 더 많은 영구 메모리를 사용하게 될 수 있습니다.

애플리케이션에 Serializable 인터페이스를 구현하는 클래스가 있으면 생성된 클래스를 풋프린트를 설명하도록 초기 영구 메모리 크기를 조정하십시오. 각 생성자는 클래스당 한 개의 생성된 오브젝트를 가지며, 오브젝트를 처음 직렬화할 때 영구 메모리에 로드되고 "GeneratedSerializationConstructorAccessorXXX"의 형식입니다(여기서 XXX는 번호임).

영구 메모리에서 할당한 오브젝트는 가비지 수집될 수 없으므로 가비지 영구 메모리를 사용하지 마십시오. 영구 메모리 영역을 자주 사용할 경우 영구 메모리에서 오브젝트 풀링을 고려해 보십시오.

언어 기능을 통한 숨겨진 메모리 할당

범위 메모리 또는 영구 메모리 컨텍스트에서는 가변 인수 언어 기능을 사용하지 마십시오. 이 메소드가 숨겨진 메모리를 할당하게 됩니다.

가변 인수(vararg)

Java 언어는 가변 인수를 메소드에 배열로 전달하여 이를 구현합니다. 컴파일러는 배열을 작성 및 초기화하여 가변 인수 메소드를 간편하게 호출할 수 있도록 합니다.

영구 메모리 또는 범위 메모리 컨텍스트에서 가변 인수 메소드를 호출하면 메모리가 손상될 수 있습니다. 범위 메모리 또는 영구 메모리 컨텍스트에서는 가변 인수를 사용하지 마십시오. 대신, 배열을 명시적으로 작성하여 가변 변수 대신 사용하십시오.

다음은 가변 인수 메소드를 호출하는 방식에 대한 두 예제입니다.

```
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;
    }
}
```

선호하는 방식은 2번째 예제입니다. 코드에서 이중 배열 할당을 표시하게 되므로 할당을 특정 메모리 영역으로 지정할 수 있습니다.

문자열 연결

긴 문자열을 만들기 위해 기존 문자열에 추가하려면 java.lang.StringBuilder 오브젝트를 사용하는데, 이때 메모리 할당이 필요합니다.

오토박싱

오토박싱(autoboxing)은 기본 유형을 가진 오브젝트 작성을 포함하는데, 이때에도 메모리 할당이 필요합니다.

메모리 컨텍스트에서 리플렉션 사용

생성자 오브젝트가 범위 메모리 영역에 빌드된 경우 동일한 범위 또는 내부 범위에서만 사용할 수 있습니다. 영구, 힙 또는 외부 범위 메모리 컨텍스트에서 해당 constructor 오브젝트를 사용하려고 하면 실패합니다.

메모리 컨텍스트에서 리플렉션이 발생할 때 나타나는 예외는 다음과 유사합니다.

```
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)
```

할당된 것과 동일한 범위에서 생성자를 사용하면 이 제한사항을 해결할 수 있습니다.

범위 메모리 영역과 내부 클래스 사용

범위 메모리 영역 컨텍스트에서 내부 클래스를 사용할 때 내부 및 외부 오브젝트가 다른 메모리 영역에 있으면 내부 클래스 오브젝트를 인스턴스화할 경우 주의해야 합니다. 내부 오브젝트가 외부 오브젝트에 대한 참조를 저장할 수 없을 경우 원본 소스 코드에 표시되지 않는 컴파일러 생성 코드에서 IllegalAssignmentError가 발생합니다.

내부 클래스 오브젝트가 암시적 참조를 외부 클래스 오브젝트에 저장할 수 있어야 합니다. 참조가 RTSJ 메모리 참조 규칙을 위반하면 IllegalAssignmentError가 생성됩니다.

대부분의 내부 클래스(로컬 및 익명 내부 클래스 포함)는 엔클로징 외부 클래스 인스턴스에 대해 컴파일러가 생성한 (합성) static 이외 필드를 포함합니다. 내부 클래스 인스턴스에 static 초기자 블록에서 인스턴스화된 익명 클래스 오브젝트 같은 엔클로징 외부 오브젝트가 없는 경우에만 예외가 발생합니다. 내부 오브젝트의 합성 필드는 외부 오브젝트에 대한 참조를 포함합니다. Java 프로그래머의 편의를 위해 컴파일러가 이를 구현합니다. 표시되는 참조와 함께 static 중첩 클래스를 사용하여 비슷한 코드를 작성할 수 있더라도 원본 소스 코드에는 해당 필드가 표시되지 않습니다. 암시적 참조가 RTSJ 메모리 영역 규칙을 위반하면 내부 오브젝트를 생성할 때 외부 오브젝트에 대한 참조를 저장하려고 하므로 IllegalAssignmentError가 발생합니다.

일반적으로, 내부 클래스를 사용할 때 RTSJ 메모리 참조 규칙을 위반할 수 없습니다. 연관된 외부 오브젝트 참조가 RTSJ 메모리 참조 규칙을 위반할 경우 내부 오브젝트를 작성할 수 없습니다. 이 규칙은 영구 또는 힙에서 할당되는 내부 오브젝트가 범위 메모리의 외부 오브젝트 참조를 가질 수 없음을 의미합니다. 범위 메모리의 내부 오브젝트가 범위 메모리의 외부 오브젝트 참조를 가질 수 있지만 외부 오브젝트는 동일한 범위 메모리 영역 또는 외부 범위 메모리 영역에서 할당되어야 합니다.

다음과 같은 해결책이 있습니다.

- static 중첩 클래스를 사용하여 암시적 참조를 제거합니다.
- 내부 및 외부 오브젝트 관계가 메모리 영역 참조 제한사항을 위반하지 않도록 메모리 영역을 선택합니다.

진단 도구 사용

IBM WebSphere Real Time for RT Linux JVM 문제점 진단에 사용할 수 있는 진단 도구가 있습니다.

IBM SDK for Java 7은 IBM WebSphere Real Time for RT Linux JVM 문제점 진단에 사용할 수 있는 진단 도구를 제공합니다. 이 섹션에서는 사용 가능한 도구를 소개하고 해당 도구를 사용하는 데 필요한 추가 정보에 대한 링크를 제공합니다.

SDK 진단 도구 사용 시 기억해야 할 중요한 사항이 있습니다. 실시간 JVM을 호출할 때 다음 옵션을 사용합니다.

```
java -Xrealtime
```

이 옵션은 실시간 JVM에 대한 진단 도구를 실행할 때 사용해야 합니다. 예를 들어, IBM WebSphere Real Time for RT Linux의 등록된 덤프 에이전트를 표시하려면 다음을 입력합니다.

```
java -Xrealtime -Xdump:what
```

이 도구를 IBM WebSphere Real Time for RT Linux와 함께 사용하는 데 관한 차이점은 여기에서 진단을 돕는 샘플 출력과 함께 보충 정보로 제공됩니다.

IBM SDK for Java 7에서 생성된 진단 정보의 요약은 진단 정보 요약을 참조하십시오.

덤프 에이전트 사용

JVM 초기화 중에 덤프 에이전트를 설정합니다. 덤프 에이전트를 통해 JVM에서 발생하는 이벤트(예: 가비지 콜렉션, 스레드 시작 또는 JVM 종료)를 사용하고 덤프를 시작하거나 외부 도구를 시작할 수 있습니다.

IBM SDK for Java 7 사용자 안내서에는 덤프 에이전트에 대한 유용한 정보를 포함하고 있으며 다음 내용을 다루고 있습니다.

- **-Xdump** 옵션 사용
- 덤프 에이전트
- 덤프 에이전트
- 덤프 에이전트 고급 제어
- 덤프 에이전트 토큰
- 기본 덤프 에이전트
- 덤프 에이전트 제거
- 덤프 에이전트 환경 변수
- 신호 맵핑
- 덤프 에이전트 기본 위치

이 정보는 여기에서 찾을 수 있습니다: IBM SDK for Java 7 - 덤프 에이전트 사용.

IBM WebSphere Real Time for RT Linux의 보충 정보는 다음 위치에 있습니다.

덤프 에이전트

덤프 에이전트는 JVM 운영 중에 발생하는 이벤트에 의해 트리거됩니다. IBM WebSphere Real Time for RT Linux의 경우, 느린 이벤트의 기본값은 5밀리초입니다.

일부 이벤트를 필터하여 출력의 연관성을 향상시킬 수 있습니다. 자세한 정보는 웹 사이트 125 페이지의 『필터 옵션』를 참조하십시오.

주: 로드 해제 및 확장 이벤트는 현재 WebSphere Real Time에서 발생하지 않습니다. 클래스는 영구 메모리 상태이며 로드 해제될 수 없습니다.

주: gpf 및 중단 이벤트는 힙 덤프를 트리거하지 않거나, 힙을 준비(request=prepwalk) 또는 압축(request=compact)하지 않습니다.

다음 테이블은 덤프 에이전트 트리거로 사용할 수 있는 이벤트를 표시합니다.

이벤트	트리거하는 경우	필터 작업
gpf	GPF(General Protection Fault)가 발생함	
user	JVM이 운영 체제에서 SIGQUIT 신호를 수신합니다.	
abort	JVM이 운영 체제에서 IGABRT 신호를 수신함	
vmstart	가상 머신이 시작됨	
vmstop	가상 머신이 중지됨	종료 코드에 대해 필터링(예: **filter=#129..#192#-42#255**)
load	클래스가 로드됨	클래스 이름에 대해 필터링(예:**filter=java/lang/String**)

이벤트	트리거하는 경우	필터 작업
unload	클래스가 언로드됨	
throw	예외가 발생함	예외 클래스 이름에 대해 필터링(예: **filter=java/lang/OutOfMem***)
catch	예외를 발견함	예외 클래스 이름에 대해 필터링(예: **filter=*Memory***)
uncaught	애플리케이션에서 Java 예외를 발견하지 못함	예외 클래스 이름에 대해 필터링(예: **filter=*MemoryError**)
systhrow	JVM에서 Java 예외가 발생함. 이 이벤트는 JVM 내부에서 발견한 오류 조건에 대해서만 트리거되므로 'throw' 이벤트와는 다름니다.	예외 클래스 이름에 대해 필터링(예: **filter=java/lang/OutOfMem***)
thrstart	새로운 스레드가 시작됨	
blocked	스레드가 블록됨	
thrstop	스레드가 중지됨	
fullgc	가비지 콜렉션 순환이 시작됨	
slow	스레드가 내부 JVM 요청에 대한 응답에 5ms 이상을 소요합니다.	'slow'로 간주하는 이벤트 소요 시간을 변경할 수 있습니다. 예를 들어 **filter=#300ms**는 내부 JVM 요청에 대한 응답에 스레드가 300ms 시간 이상 소요하면 트리거합니다.
allocation	Java 오브젝트가 주어진 필터 스펙과 일치하는 크기로 할당됨	오브젝트 크기에 대해 필터링하고 필터를 제공해야 합니다. 예를 들어 **filter=#5m**는 5MB 이상인 오브젝트를 트리거합니다. **filter=#256k..512k**는 크기가 256KB에서 512KB 사이인 오브젝트에 대해 트리거합니다.
traceassert	내부 오류가 JVM에서 발생합니다.	적용할 수 없습니다.
corruptcache	JVM에서 공유 클래스 캐시의 손상을 발견합니다.	적용할 수 없습니다.

필터 옵션

일부 JVM 이벤트는 애플리케이션 실행 시간 중 여러 번 발생합니다. 덤프 에이전트는 필터 및 범위를 사용하여 초과 덤프가 생성되지 않도록 방지합니다.

와일드 카드

필터의 시작 또는 끝에만 별표를 지정하여 예외 이벤트 필터에서 와일드 카드를 사용할 수 있습니다. 다음 명령은 두 번째 별표가 끝에 있지 않으므로 작동하지 않습니다.

```
-Xdump:java:events=vmstop,filter=*InvalidArgumentException#*.myVirtualMethod
```

이 필터를 작동하게 하려면 다음과 같이 변경되어야 합니다.

```
-Xdump:java:events=vmstop,filter=*InvalidArgumentException#MyApplication.*
```

클래스 로딩 및 예외 이벤트

클래스 로딩(load) 및 예외(throw, catch, uncaught, systhrow) 이벤트를 Java 클래스 이름에 따라 필터할 수 있습니다.

```
-Xdump:java:events=throw,filter=java/lang/OutOfMem*
-Xdump:java:events=throw,filter=*MemoryError
-Xdump:java:events=throw,filter=*Memory*
```

throw, uncaught 및 systhrow 예외 이벤트를 Java 메소드 이름에 따라 필터링할 수 있습니다.

```
-Xdump:java:events=throw,filter=ExceptionClassName[#ThrowingClassName.
throwingMethodName[#stackFrameOffset]]
```

선택적 부분은 대괄호로 표시되어 있습니다.

발견 예외 이벤트를 Java 메소드 이름에 따라 필터링할 수 있습니다.

```
-Xdump:java:events=catch,filter=ExceptionClassName[#CatchingClassName.
catchingMethodName]
```

선택적 부분은 대괄호로 표시되어 있습니다.

vmstop 이벤트

하나 이상의 종료 코드를 사용하여 JVM 시스템 종료 이벤트를 필터할 수 있습니다.

```
-Xdump:java:events=vmstop,filter=#129..192#-42#255
```

느린 이벤트

느린 이벤트를 필터링하여 기본 5ms에서 시간 임계값을 변경할 수 있습니다.

```
-Xdump:java:events=slow,filter=#300ms
```

사용자는 필터를 기본 시간보다 낮은 시간으로 설정할 수 없습니다.

할당 이벤트

트리거를 발생시키는 오브젝트의 *크기*를 지정하려면 할당 이벤트를 필터해야 합니다. 필터 *크기*는 32비트 플랫폼의 경우 0에서 최대값인 32비트 포인터까지 64비트 플랫폼의 경우 최대값인 64비트 포인터까지 설정할 수 있습니다. 필터 값을 0으로 낮게 설정하면 모든 할당에 대해 덤프가 트리거됩니다.

예를 들어, *크기*가 5Mb를 초과하는 할당에 대해 덤프를 트리거하려면 다음을 사용하십시오.

```
-Xdump:stack:events=allocation,filter=#5m
```

*크기*가 256Kb - 512Kb 사이인 할당에 대해 덤프를 트리거하려면 다음을 사용하십시오.

```
-Xdump:stack:events=allocation,filter=#256k..512k
```

기타 이벤트

필터링을 지원하지 않는 이벤트에 필터를 적용하면 해당 필터가 무시됩니다.

요청 옵션

요청 옵션을 사용하여 덤프 에이전트를 시작하기 전에 JVM이 상태를 준비하도록 할 수 있습니다. IBM WebSphere Real Time for RT Linux의 경우 추가 요청 옵션 **multiple**이 있습니다.

사용 가능한 옵션이 다음 표에 나열되어 있습니다.

옵션 값	설명
exclusive	JVM에 독점적 액세스를 요청합니다.
compact	가비지 콜렉션을 실행합니다. 이 옵션은 덤프가 생성되기 전에 힙의 모든 접근 불가능한 오브젝트를 제거합니다.
prepwalk	워킹(walking)에 대한 힙을 준비합니다. 이 옵션을 사용할 때 **exclusive**도 지정해야 합니다.
serial	하나의 덤프가 완료될 때까지 다른 덤프를 일시중단합니다.
multiple	각 RTSJ 메모리 영역에 대해 별개의 힙 덤프를 생성합니다.
preempt	Java 덤프 에이전트에 적용하여 스택 추적을 수집하기 위해 처리 중인 원시 스레드를 강제 선점하는지 여부를 제어합니다. 이 옵션이 지정되어 있지 않으면 Java 스택 추적만 Javadump에서 콜렉트합니다.

예를 들어, Javadump에 대한 요청 옵션의 기본 설정은 `request=exclusive+preempt` 입니다. 스레드를 선점하지 않고 Javadump가 생성되어 원시 스택 추적을 수집하도록 설정을 변경하려면 다음 옵션을 사용하십시오.

```
-Xdump:java:request=exclusive
```

일반적으로 기본 요청 옵션으로 충분합니다.

하나 이상의 요청 옵션을 지정하려면 +를 사용하십시오. 예를 들면 다음과 같습니다.

```
-Xdump:heap:request=exclusive+compact+prepwalk
```

Java 덤프 사용

Java 덤프는 JVM 및 Java 애플리케이션과 관련하여 실행 중 한 지점에서 수집한 진단 정보를 포함하는 파일을 생성합니다. 예를 들어 운영 체제, 애플리케이션 환경, 스레드, 스택, 잠금 및 메모리에 관한 정보가 포함될 수 있습니다.

IBM SDK for Java 7 사용자 안내서에는 Java 덤프에 대한 유용한 정보를 포함하고 있으며 다음 내용을 다루고 있습니다.

- Java 사용
- Java 덤프 트리거
- Javadump 해석

- 환경 변수 및 Java 덤프

이 정보는 여기에서 찾을 수 있습니다: IBM SDK for Java 7 - Java 덤프 사용.

다음 주제에 IBM WebSphere Real Time for RT Linux의 보충 정보 및 샘플 출력이 있습니다.

스토리지 관리(MEMINFO)

MEMINFO 섹션에서는 힙, 영구 및 범위 메모리 영역을 포함한 Memory Manager 관련 정보를 제공합니다.

Java 덤프의 MEMINFO 섹션은 Memory Manager에 대한 정보를 표시합니다. Memory Manager 컴포넌트 작동 방식에 대한 세부사항은 메트로놈 가비지 콜렉터 사용의 내용을 참조하십시오.

Java 덤프의 이 파트는 다음을 포함한 다양한 스토리지 관리 값을 제공합니다.

- 사용 가능한 메모리 용량
- 사용된 메모리 용량
- 현재 힙 *크기*
- 현재 영구 메모리 영역 *크기*
- 현재 범위 메모리 영역 *크기*

이 섹션에는 가비지 콜렉션 히스토리 데이터도 있습니다. 가비지 콜렉션 히스토리 데이터는 추적점 시퀀스로 표시되며 각 추적점에는 시간소인이 있고 가장 최근 추적점이 맨 처음에 오도록 정렬되어 있습니다.

표준 JVM에서 생성된 Javadump에는『GC 히스토리』섹션이 있습니다. 이러한 정보는 실시간 JVM 사용 시 생성되는 Javadump에는 포함되지 않습니다. **-verbose:gc** 옵션 또는 JVM 스냅 추적을 사용하여 GC 동작에 대한 정보를 얻으십시오. 자세한 내용은 IBM SDK for Java 7 사용자 안내서의 147 페이지의『verbose:gc 정보 사용』및 덤프 에이전트 섹션을 참조하십시오.

범위 지정된 메모리를 사용하는 프로그램을 실행 중이며 OutOfMemoryError가 처리되는 경우, Java 덤프에 나열된 일부 메모리 영역이 비어 있을 수 있습니다. Java 덤프가 생성될 때 다른 범위 내에 중첩된 범위의 메모리가 부족하면 내부 범위가 삭제되었을 수 있습니다. OutOfMemoryError 처리 당시의 메모리 영역 상태와 관련된 정보를 가져오려면 다음 명령행 옵션을 사용하여 프로그램을 실행하십시오.

```
-Xdump:java:events=throw,filter=java/lang/OutOfMemoryError,range=1..1
```

위 명령은 미발견 예외가 발견되는 경우(약간 늦게 발생함)가 아니라 OutOfMemoryError가 처리되는 경우 추가 Java 덤프를 생성합니다. 이 Java 덤프에서 모든 내부 범위를

포함하여 OutOfMemoryError가 처리될 때 활성이었던 모든 메모리 영역을 볼 수 있습니다. **-Xdump** 옵션 사용에 대한 추가 정보는 IBM SDK for Java 7 사용자 안내서를 참조하십시오.

Java 덤프에서 세그먼트는 대량의 메모리를 사용하는 태스크를 위해 Java Runtime이 할당한 메모리 블록입니다. 예제 태스크는 다음과 같습니다.

- JIT 캐시 유지보수
- Java 클래스 저장

또한 Java Runtime은 MEMINFO 섹션에 나열되지 않은 다른 원시 메모리를 할당합니다. Java Runtime 세그먼트에 사용되는 총 메모리가 Java Runtime의 전체 메모리 풋프린트를 반드시 나타내는 것은 아닙니다. Java Runtime 세그먼트는 세그먼트 데이터 구조 및 원시 메모리의 연관된 블록으로 구성됩니다.

다음은 일반 출력의 예입니다. 모든 값이 16진 값으로 제공됩니다. MEMINFO 섹션의 열 표제는 다음 의미를 가집니다.

```
| 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           ------------------------------------------------------------------------
```

스레드 및 스택 추적(THREADS)

애플리케이션 프로그래머에게 가장 유용한 Java 덤프 항목 중 하나는 THREADS 섹션입니다. 이 섹션에는 Java 스레드, 원시 스레드 및 스택 추적에 대한 목록이 있습니다. IBM WebSphere Real Time for RT Linux 실시간 스레드 및 힙이 없는 실시간 스레드도 표시됩니다.

Java 스레드는 운영 체제의 원시 스레드가 구현합니다. 각 스레드는 다음과 같이 일련의 행으로 표시됩니다.

```
"main" J9VMThread:0x41D11D00, j9thread_t:0x003C65D8, java/lang/Thread:0x40BD6070, state:CW, prio=5
 (native thread ID:0xA98, native priority:0x5, native policy:UNKNOWN)
Java callstack:
 at java/lang/Thread.sleep(Native Method)
 at java/lang/Thread.sleep(Thread.java:862)
 at mySleep.main(mySleep.java:31)
```

ps 명령을 사용할 때 Java 스레드 이름이 운영 체제에 표시됩니다. **ps** 명령 사용에 대한 추가 정보는 106 페이지의 『일반 디버깅 기술』의 내용을 참조하십시오.

힙이 없는 실시간 스레드에서 생성된 Java 덤프의 경우 일부 정보가 누락되었을 수 있습니다. 힙이 없는 실시간 스레드에 스레드 이름 오브젝트가 표시되지 않는 경우, 실제 스레드 이름 대신 『(액세스 오류)』 텍스트가 출력됩니다.

첫 번째 행의 특성은 스레드 이름, JVM 스레드 구조의 주소, Java 스레드 오브젝트의 주소, 스레드 상태 및 Java 스레드 우선순위입니다. 두 번째 행의 특성은 원시 운영 체제 스레드 ID, 원시 운영 체제 스레드 우선순위 및 원시 운영 체제 스케줄링 정책입니다.

스레드 이름은 세 가지 방법으로 표시됩니다.

- javacore 파일에 표시됩니다. 모든 파일이 javacore 파일에 나타나는 것은 아닙니다.
- **ps** 명령을 사용하는 운영 체제에서 스레드를 나열하는 경우
- java.lang.Thread.getName() 메소드를 사용하는 경우

다음 테이블에서 IBM WebSphere Real Time for RT Linux 스레드 이름에 대한 정보가 표시됩니다.

표 12. IBM WebSphere Real Time for RT Linux의 스레드 이름

스레드 세부사항	스레드 이름
2차 스레드에 의한 오브젝트 완료를 디스패치하기 위해 가비지 콜렉션이 사용하는 내부 JVM 스레드	`Finalizer master`
가비지 콜렉터가 사용하는 알람 스레드	`GC Alarm`
가비지 콜렉션을 위해 사용되는 Slave 스레드	`GC Slave`
애플리케이션에서 메소드의 사용을 샘플링하기 위해 JIT(just-in-time) 컴파일러 모듈이 사용하는 내부 JVM 스레드	`JIT Sampler`
내부 또는 외부적으로 생성되는 애플리케이션에 의해 수신된 신호를 관리하기 위해 VM이 사용하는 스레드	`Signal Reporter`

Java 코드로 작성된 실시간 스레드(javax.realtime.RealtimeThread)의 기본 이름은 `RTThread-x`입니다. 여기서 『x』는 스레드 번호입니다.

힙이 없는 실시간 스레드의 기본 이름은 `NHRTThread-x`입니다. 여기서 『x』는 스레드 번호입니다.

Java 스레드 우선순위는 플랫폼에 기반을 둔 방식으로 운영 체제 우선순위 값에 맵핑됩니다. Java 스레드 우선순위에서 큰 값은 스레드의 우선순위가 높음을 의미합니다. 즉, 이러한 스레드는 우선순위가 낮은 스레드보다 더 자주 실행됩니다. 우선순위가 Java 스레드, 실시간 스레드 및 힙이 없는 실시간 스레드에서 어떻게 작동하는지 자세히 알아보려면 12 페이지의 『우선순위 맵핑 및 상속』의 내용을 참조하십시오.

상태 값이 될 수 있는 것은 다음과 같습니다.

- R - Runnable - 기회가 되면 스레드를 실행할 수 있습니다.
- CW - Condition Wait - 스레드가 대기 중입니다. 대기 원인의 예를 들면 다음과 같습니다.
  - sleep() 호출이 작성됨
  - 스레드가 I/O에 대해 블록됨
  - wait() 메소드가 호출되어 알림을 받는 모니터에서 대기함
  - 스레드가 join() 호출을 사용하여 다른 스레드와 동기화 중임
- S – Suspended – 스레드가 다른 스레드에 의해 일시중단되었습니다.
- Z – Zombie – 스레드가 강제 종료되었습니다.
- P – Parked – 새 동시성 API(java.util.concurrent)에 의해 스레드가 중지되었습니다.
- B – Blocked – 스레드가 다른 항목이 현재 소유하고 있는 잠금을 얻기 위해 대기 중입니다.

스레드가 대기 또는 차단 상태인 경우, 출력에서 3XMTHREADBLOCK으로 시작하는 해당 스레드 행에는 스레드가 기다리는 자원 및 가능한 경우 해당 자원을 현재 소유하는 스레드가 나열됩니다. 자세한 정보는 IBM SDK for Java 7 사용자 안내서에 있는 차단된 스레드 관련 주제를 참조하십시오.

진단 정보를 얻기 위해 java 덤프를 시작하면 JVM이 javacore를 생성하기 전에 Java 스레드를 중지합니다. exclusive_vm_access의 준비 상태는 TITLE 섹션의 1TIPREPSTATE 행에 표시됩니다.

```
1TIPREPSTATE Prep State: 0x4 (exclusive_vm_access)
```

javacore가 트리거되었을 때 Java 코드를 실행하던 스레드는 CW(조건 대기) 상태입니다.

```
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 섹션은 이 스레드가 내부 JVM 잠금을 대기 중임을 표시합니다.

```
2LKREGMON          Thread public flags mutex lock (0x002A5234): <unowned>
3LKNOTIFYQ            Waiting to be notified:
3LKWAITNOTIFY            "main" (0x41481900)
```

힙 덤프 사용

힙 덤프는 IBM Virtual Machine for Java 메커니즘을 설명하는 용어로, Java 힙에 있는 모든 활성 오브젝트 즉, 실행 중인 Java 애플리케이션에 사용되는 오브젝트의 덤프를 생성하는 메커니즘을 설명합니다.

IBM SDK for Java 7 사용자 안내서에는 힙 덤프에 대한 유용한 정보를 포함하고 있으며 다음 내용을 다루고 있습니다.

- 힙 덤프 가져오기
- 힙 덤프 처리에 사용되는 도구
- 힙 정보를 얻기 위해 **-Xverbose:gc** 사용
- 환경 변수 및 힙 덤프
- 텍스트(classic) 힙 덤프 파일 형식
- PHD(Portable Heap Dump) 파일 형식

이 정보는 여기에서 찾을 수 있습니다: IBM SDK for Java 7 - 힙 덤프 사용하기.

IBM WebSphere Real Time for RT Linux 보충 정보:

실시간 JVM에 대해 여러 힙 덤프 사용

생성된 힙 덤프는 기본적으로 힙 메모리, 영구 메모리 및 범위 메모리 등 모든 메모리 영역의 모든 Java 오브젝트에 대한 정보가 들어 있는 단일 파일입니다. 여러 덤프는 생성하는 주된 이유는 수정사항 없이 기존의 힙 덤프 도구를 사용하여 개별 힙 영역을 분석할 수 있기 때문입니다.

이 태스크 정보

기본적으로 힙 덤프에는 JVM 메모리 영역, 힙, 영구 및 범위 메모리에 있는 모든 오브젝트에 관한 정보를 포함되어 있습니다. **-Xdump:heap**과 함께 **request=multiple** 옵션을 사용하여 각 메모리 영역에 있는 Java 오브젝트에 대한 정보가 들어 있는 별도의 힙 덤프를 구할 수 있습니다. 요청 옵션의 기본 설정을 반복해야 하며, request=multiple+exclusive+prepwalk+compact를 지정해야 합니다. 이렇게 하면 이름에 특정 메모리 영역을 나타내는 추가 필드가 있는 힙 덤프 세트가 생성됩니다.

```
heapdump.%id.%Y%m%d.%H%M%S.%pid.phd
```

여기서 *%id*는 힙 덤프 파일을 힙 메모리, 영구 메모리 또는 범위 메모리의 특정 영역에 있는 오브젝트를 포함한 것으로 식별합니다.

『기본』, 『영구』, 『범위』 및 『기타』라는 이름의 4가지 유형의 힙이 있습니다. 힙 덤프 코드는 힙 레이블의 %id를 ID(대개 숫자)로 연결된 이 이름 중 하나로 대체합니다(예 : heapdump.Immortal12994208.20060807.093653.7684.txt.).

예

```
java -Xrealtime -Xdump:heap:defaults:request=multiple+exclusive+compact+prepwalk <java prog
```

이 추가 옵션을 사용하면 이동 가능한 힙 덤프(phd) 형식에 여러 힙 덤프가 생성됩니다.

```
java -Xrealtime -Xdump:heap:defaults:request=multiple+exclusive+compact+prepwalk,opts=CLASS
```

이 추가 옵션을 사용하면 기본(CLASSIC) 텍스트 형식에 여러 힙 덤프가 생성됩니다.

-Xdump:what 옵션은 JVM 시작 시 덤프 에이전트를 보여주며 보유한 덤프 옵션을 확인하는 데 유용합니다.

텍스트(classic) 힙 덤프 파일 형식

텍스트 또는 classic 힙 덤프는 오브젝트 유형, 크기 및 오브젝트 간 참조를 포함한 힙에 있는 모든 오브젝트 인스턴스의 목록입니다.

헤더 레코드

헤더 레코드는 버전 정보 문자열이 포함된 단일 레코드입니다.

```
// Version: <version string containing SDK level, platform and JVM build level>
```

예:

```
// Version: J2RE 7.0 IBM J9 2.6 Linux x86-32 build 20101016_024574_lHdRSr
```

오브젝트 레코드

오브젝트 레코드는 힙의 각 오브젝트 인스턴스마다 하나씩 있는 다중 레코드입니다. 오브젝트 주소, 크기, 유형 및 참조를 오브젝트에서 제공합니다.

```
<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> ...
```

오브젝트 주소 및 힙 참조는 힙에 있으나 클래스 블록 주소는 힙 외부에 있습니다. 널 값인 참조를 포함하여 오브젝트 인스턴스에서 발견된 모든 참조가 나열됩니다. 오브젝트 유형은 패키지가 포함된 클래스 이름이거나 표준 JVM 유형 서명으로 표시되는 클래스 배열 유형 또는 기본 배열입니다. 137 페이지의 『Java VM 유형 서명』의 내용을 참조하십시오. 리플렉션 클래스 인스턴스의 경우 일반적으로 오브젝트 레코드에 추가 클래스 블록 참조도 포함될 수 있습니다.

예:

다음은 길이가 28바이트인 java/lang/String 유형의 오브젝트 인스턴스입니다.

```
0x00436E90 [28] OBJ java/lang/String
```

java/lang/String의 클래스 블록 주소 뒤에는 문자 배열 인스턴스에 대한 참조가 옵니다.

```
0x415319D8 0x00436EB0
```

다음은 길이가 44바이트인 문자 배열 유형의 오브젝트 인스턴스입니다.

```
0x00436EB0 [44] OBJ [C
```

다음은 문자 배열의 클래스 블록 주소입니다.

```
0x41530F20
```

다음은 java/util/Hashtable 항목 내부 클래스 배열 유형의 오브젝트입니다.

```
0x004380C0 [108] OBJ [Ljava/util/Hashtable$Entry;
```

다음은 java/util/Hashtable 항목 내부 클래스 유형의 오브젝트입니다.

```
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
```

다음은 클래스 블록 주소 및 힙 참조(널 참조 포함)입니다.

```
0x4158CB88 0x004219B8 0x004341F0 0x00000000
```

클래스 레코드

클래스 레코드는 로드된 클래스마다 하나씩 있는 다중 레코드입니다. 클래스 블록 주소, 크기, 유형 및 참조를 클래스에서 제공합니다.

```
<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>...
```

클래스 블록 주소 및 클래스 블록 참조는 힙 외부에 있으나 일반적으로 static 클래스 데이터 멤버인 경우 클래스 레코드에도 힙에 대한 참조가 포함될 수 있습니다. 널값인 참조를 포함하여 클래스 블록에서 발견된 모든 참조가 나열됩니다. 클래스 유형은 패키지가 포함된 클래스 이름이거나 표준 JVM 유형 서명으로 표시되는 클래스 배열 유형 또는 기본 배열입니다. 137 페이지의 『Java VM 유형 서명』의 내용을 참조하십시오.

예:

다음은 길이가 32바이트인 java/lang/Runnable 클래스의 클래스 블록입니다.

```
0x41532E68 [32] CLS java/lang/Runnable
```

다음은 다른 클래스 블록에 대한 참조 및 힙 참조(널 참조 포함)입니다.

```
0x4152F018 0x41532E68 0x00000000 0x00000000 0x00499790
```

다음은 길이가 168바이트인 java/lang/Math 클래스의 클래스 블록입니다.

```
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
```

트레일러 레코드 1

트레일러 레코드 1은 레코드 계수가 포함된 단일 레코드입니다.

```
// Breakdown - Classes: <class record count, in decimal>,
Objects: <object record count, in decimal>,
ObjectArrays: <object array record count, in decimal>,
PrimitiveArrays: <primitive array record count, in decimal>
```

예:

```
// Breakdown - Classes: 321, Objects: 3718, ObjectArrays: 169,
PrimitiveArrays: 2141
```

트레일러 레코드 2

트레일러 레코드 2는 총계가 포함된 단일 레코드입니다.

```
// EOF:  Total 'Objects',Refs(null) :
<total object count, in decimal>,
<total reference count, in decimal>
(,total null reference count, in decimal>)
```

예:

```
// EOF:  Total 'Objects',Refs(null) : 6349,23240(7282)
```

Java VM 유형 서명

Java VM 유형 서명은 Java 유형의 약어이며 아래 테이블에 표시되어 있습니다.

Java VM 유형 서명	Java 유형
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double
L <*fully qualified-class*> ;	<*fully qualified-class*>
[<*type*>	<*type*>[ ] (array of <*type*>)
(<*arg-types*>) <*ret-type*>	메소드

시스템 덤프 및 덤프 뷰어 사용

JVM은 구성 가능한 조건에서 원시 시스템 덤프(코어 덤프)를 생성할 수 있습니다. 시스템 덤프는 일반적으로 상당히 큽니다. 시스템 덤프를 분석하는 데 사용하는 대부분의 도구는 또한 특정 플랫폼을 사용합니다.Linux에서 시스템 덤프를 분석하려면 **gdb** 도구를 사용하십시오.

IBM SDK for Java 7 사용자 안내서에는 시스템 덤프 및 덤프 뷰어 사용에 대한 유용한 정보가 있으며 다음 내용을 다루고 있습니다.

- 시스템 덤프 개요
- 시스템 덤프 기본값
- 덤프 뷰어 사용
  - **jextract** 사용
  - 덤프 뷰어를 사용하여 다룰 문제점
  - **jdmpview**에서 사용 가능한 명령
  - 세션 예
  - **jdmpview** 명령 빠른 참조

이 정보는 여기에서 찾을 수 있습니다: IBM SDK for Java 7 - 시스템 덤프 및 덤프 뷰어 사용.

IBM WebSphere Real Time for RT Linux 보충 정보:

jextract 사용

실시간 JVM에서 시스템 덤프를 처리할 때 **-Xrealtime** 옵션을 포함해야 합니다. 예를 들면 다음과 같습니다.

```
jextract -Xrealtime <core file name> [<zip_file>]
```

덤프가 생성된 JVM과 다른 JVM에서 **jextract**를 실행하면 다음 오류 메시지가 표시됩니다.

```
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).
```

또한 이 메시지는 **jextract**로 덤프를 처리할 때 Java를 표준 JVM으로 실행하지만 **-Xrealtime** 옵션을 사용한 경우에도 발생합니다.

jdmpview에서 사용 가능한 명령

jdmpview는 JVM 시스템 덤프에서 정보를 탐색하고 다양한 분석 기능을 수행하는 대화식 명령행 도구입니다.

info jitm

AOT 및 JIT 컴파일 메소드와 해당 주소를 표시합니다.

- 메소드 이름 및 서명
- 메소드 시작 주소
- 메소드 끝 주소

기타 모든 명령 옵션은 IBM SDK for Java 7 사용자 안내서를 참조하십시오.

Java 애플리케이션 및 JVM 추적

JVM 추적은 IBM WebSphere Real Time for RT Linux에서 제공하는 추적 기능으로 성능에 최소한의 영향을 줍니다. 대부분 경우, 압축 데이터는 압축 2진 형식으로 보관되며 제공된 Java 포맷터로 포맷할 수 있습니다.

추적은 메모리 버퍼로 이동하는 소량의 추적 지점 세트와 함께 기본적으로 사용 가능합니다. 런타임 시 레벨, 컴포넌트, 그룹 이름 또는 개별 추적 지점 ID를 사용하여 추적 지점을 사용할 수 있습니다.

IBM SDK for Java 7 사용자 안내서에는 애플리케이션 추적에 대한 자세한 정보가 있으며 다음 내용을 다루고 있습니다.

- 추적 대상

- 추적 지점 유형
- 기본 추적
- 추적 데이터 레코딩
- 추적 제어
- Java 애플리케이션 추적
- Java 메소드 추적

IBM WebSphere Real Time for RT Linux 추적 시, 추적 옵션을 포함하는 경우 실시간 JVM을 올바르게 호출해야 합니다. 예를 들어, 추적 옵션을 지정하는 경우 다음을 입력해야 합니다.

```
java -Xrealtime -Xtrace:<options>
```

IBM SDK for Java 7 정보를 여기에서 찾을 수 있습니다. Java 애플리케이션 및 JVM 추적.

JIT 및 AOT 문제점 판별

명령행 옵션을 사용하여 JIT 및 AOT 컴파일러 문제를 진단하고 성능을 조정할 수 있습니다.

IBM WebSphere Real Time for RT Linux가 일부 공통 컴포넌트를 IBM SDK for Java 7과 공유하더라도 JIT 와 AOT의 동작은 다릅니다. 이 섹션은 IBM WebSphere Real Time for RT Linux의 JIT 및 AOT 문제를 다룹니다.

JIT 또는 AOT 문제 진단

때때로 올바른 바이트 코드를 올바르지 않은 원시 코드로 컴파일한 경우 Java 프로그램이 실패할 수 있습니다. JIT 또는 AOT 컴파일러에 장애 유무를 판별하여 해당 장애가 있는 *대상*에 Java 서비스 팀 지원을 제공할 수 있습니다.

이 태스크 정보

공유 클래스 캐시가 채워졌을 때 컴파일된 메소드를 판별하려면 관리 캐시 명령행에서 **-Xaot:verbose** 옵션을 사용하십시오. 예를 들면 다음과 같습니다.

```
admincache -Xrealtime -Xaot:verbose -populate -aot my.jar -cp <My Class Path>
```

본 절에서는 컴파일러 관련 문제일 경우 판별하는 방법을 설명합니다. 이 절에서는 또한 가능한 몇몇 해결 방법 및 컴파일러 관련 문제 해결을 위한 디버깅 기술을 제안합니다.

JIT 또는 AOT 컴파일러 사용 안함:

문제가 JIT 또는 AOT 컴파일러에서 발생하는 것으로 의심되면 컴파일을 사용하지 않아도 문제가 지속되는지 보십시오. 문제가 여전히 발생하면 컴파일러가 원인이 아님을 알 수 있습니다.

이 태스크 정보

기본적으로 JIT 컴파일러는 사용 가능합니다. AOT 컴파일러는 기본적으로 사용 가능하지만 공유 클래스가 사용 가능해야 활성화됩니다. Java 애플리케이션에 있는 모든 메소드가 컴파일되는 것은 아니며 이는 효율성을 위해서입니다. JVM은 애플리케이션의 각 메소드에 대한 호출 계수를 유지합니다. 메소드가 호출되고 해석될 때마다 해당 메소드에 대한 호출 계수가 증가됩니다. 계수가 컴파일 임계값에 도달하면 메소드가 컴파일되고 원래대로 실행됩니다.

호출 계수 메커니즘은 가장 자주 사용되는 메소드에 우선순위를 주어 애플리케이션의 실행 주기 동안 메소드 컴파일을 전개합니다. 일부 자주 사용되지 않는 메소드는 전혀 컴파일되지 않을 수도 있습니다. 결과적으로 Java 프로그램이 실패하면 문제가 JIT 또는 AOT 컴파일러에 있거나 JVM의 다른 부분에 있는 것입니다.

실패를 진단하는 첫 번째 단계는 문제가 *어디에* 있는지 판별하는 것입니다. 이렇게 하려면 먼저 단순히 해석된모드(즉, JIT 및 AOT 컴파일러 사용 안함)에서 Java 프로그램을 실행하십시오.

프로시저

1. 명령행에서 **-Xjit** 및 **-Xaot** 옵션(그리고 수반하는 매개변수)을 제거하십시오.
2. **-Xint** 명령행 옵션을 사용하여 JIT 및 AOT 컴파일러를 끄십시오. 성능을 고려하여 프로덕션 환경에서는 **-Xint** 옵션을 사용하지 마십시오.

다음에 수행할 작업

컴파일을 사용하지 않고 Java 프로그램을 실행하면 다음 중 하나의 결과를 얻습니다.

- 실패가 지속됩니다. 이 경우 문제가 JIT 또는 AOT 컴파일러에 있지 않습니다. 일부 경우, 프로그램이 다른 방법으로 시작하지 못할 수 있지만 컴파일러 관련 문제는 아닙니다.
- 실패가 없어졌습니다. 문제가 JIT 또는 AOT 컴파일에 있을 가능성이 많습니다.

  공유 클래스를 사용하지 않는 경우라면 JIT 컴파일러에 문제가 있는 것입니다. 공유 클래스를 사용하는 경우라면 JIT 컴파일만 사용하여 애플리케이션을 실행해서 어떤 컴파일러에 문제가 있는지 판단할 수 있습니다. 애플리케이션을 **-Xint** 옵션이 아닌 **-Xnoaot** 옵션을 사용하여 실행하십시오. 결과는 다음 중 하나입니다.
  - 실패가 지속됩니다. 이 경우 문제가 JIT 컴파일러에 있습니다. JIT 컴파일러에만 문제가 있는지 확인하려면 **-Xnoaot** 옵션 대신에 **-Xnojit** 옵션을 사용하십시오.
  - 실패가 없어졌습니다. 이 경우 문제가 AOT 컴파일러에 있습니다.

선택적으로 JIT 컴파일러 사용 안함:

Java 프로그램 실패가 JIT 컴파일러의 문제를 지정하는 경우 문제의 범위를 더욱 축소할 수 있습니다.

이 태스크 정보

기본적으로 JIT 컴파일러는 다양한 최적화 레벨에서 메소드를 최적화합니다. 다양한 최적화 선택사항이 호출 계수에 기반하여 다양한 메소드에 적용됩니다. 가장 자주 호출되는 메소드가 상위 레벨로 최적화됩니다. JIT 컴파일러 매개변수를 변경하여 메소드를 최적화하는 레벨을 제어할 수 있습니다. 최적화 프로그램에 결함이 있는지 여부와 결함이 있으면 문제가 되는 최적화를 판별할 수 있습니다.

JIT 매개변수를 쉼표로 구분된 목록으로 **-Xjit** 옵션에 추가하여 지정할 수 있습니다. 구문은 **-Xjit**:<*param1*>,<*param2*>=<*value*>입니다. 예를 들면 다음과 같습니다.

```
java -Xjit:verbose,optLevel=noOpt HelloWorld
```

HelloWorld 프로그램을 실행하고 JIT에서 상세 출력을 활성화하여 JIT에서 최적화 수행 없이 원시 코드를 생성하게 합니다.

컴파일러의 어느 부분에 문제가 있는지 판단하려면 다음 단계를 따르십시오.

프로시저

1. JIT 매개변수 **count=0**을 설정하여 컴파일 임계값을 영(0)으로 변경하십시오. 이 매개변수를 사용하면 각 Java 메소드가 실행되기 전에 컴파일됩니다. **count=0**은 문제점 진단 시에만 사용하십시오. 자주 사용하지 않는 메소드를 포함하여 추가로 여러 메소드가 컴파일됩니다. 추가 컴파일에서는 컴퓨팅 자원을 많이 사용하므로 애플리케이션의 속도가 저하됩니다. **count=0**을 사용하면 문제 영역에 도달했을 때 애플리케이션이 즉각 실패합니다. 일부 경우 **count=1**을 사용하여 확증을 위해 실패를 재현할 수 있습니다.
2. **disableInlining**을 JIT 컴파일러 매개변수에 추가하십시오. **disableInlining**은 더 크고 복잡한 코드 생성을 불가능하게 합니다. 더 이상 문제가 발생하지 않는다면 Java 서비스 팀에서 컴파일러 문제를 분석하고 정정하는 동안 **disableInlining**을 임시 해결 방법으로 사용하십시오.
3. **optLevel** 매개변수를 추가하여 최적화 레벨을 줄이고 실패가 더 이상 발생하지 않을 때까지 또는 『noOpt』 레벨에 도달할 때까지 프로그램을 다시 실행하십시오. JIT 컴파일러 문제일 경우 『scorching』으로 시작하고 목록에 대해 작업하십시오. 최적화 레벨은 다음과 같으며 내림차순입니다.
   a. scorching
   b. veryHot
   c. hot

d. warm

e. cold

f. noOpt

다음에 수행할 작업

이러한 설정 중 하나를 사용하여 실패가 발생하지 않았으면, 이 방법은 사용할 수 있는 임시 해결 방법이 됩니다. 이 임시 해결 방법은 Java 서비스 킴이 컴파일러 문제를 분석 및 수정하는 동안 임시로 사용됩니다. JIT 매개변수 목록에서 **disableInlining**을 제거해도 실패가 다시 발생하지 않으면 성능 향상을 위해 그렇게 하십시오. 임시 해결 방법의 성능을 향상시키려면 『실패 메소드 찾기』에 있는 지시사항을 따르십시오.

『noOpt』 최적화 레벨에서 실패가 계속해서 발생한다면 임시 해결 방법으로 JIT 컴파일러를 끄십시오.

실패 메소드 찾기:

가장 낮은 최적화 레벨로 JIT 또는 AOT 컴파일러가 메소드를 컴파일해서 실패를 트리거할 수 있다고 판단하는 경우 컴파일된 Java 프로그램의 어느 부분이 실패의 원인이 되는지 찾을 수 있습니다. 컴파일러를 특정 메소드, 클래스 또는 패키지에 대한 임시 해결 방법으로 한정하도록 지시하여 컴파일이 프로그램의 다른 부분은 정상적으로 컴파일하도록 할 수 있습니다. JIT 컴파일러 실패의 경우, 실패가 **-Xjit:optLevel=noOpt**에서 발생한다면, 컴파일러에 실패를 유발하는 메소드 모두를 컴파일하지 않도록 지시할 수 있습니다.

시작하기 전에

이 예와 유사한 오류 출력을 보고 실패한 메소드를 식별할 수 있습니다.

```
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;
```

중요한 행은 다음과 같습니다.

vmState=0x00000000

실패한 코드가 JVM Runtime 코드가 아님을 표시합니다.

Module= or Module_base_address=

코드가 JIT에서 컴파일되고 DLL 또는 라이브러리 외부에 있어 출력에 표시되지 않습니다(공백 또는 영(0)).

Compiled_method=

컴파일된 코드가 생성된 Java 메소드를 표시합니다.

이 태스크 정보

출력에서 실패한 메소드를 표시하지 않으면 다음 단계를 따라 추가 메소드를 식별하십시오.

프로시저

1. Java 프로그램을 **-Xjit** 또는 **-Xaot** 옵션에 추가된 JIT 매개변수 **verbose** 및 **vlog=**<*filename*>을 적용하여 실행하십시오. 이 매개변수를 사용하면 컴파일러가 <*filename*>.<*date*>.<*time*>.<*pid*> 또는 *한계 파일* 로그 파일에 컴파일된 메소드를 나열합니다. 다음과 같이 일반적 한계 파일에는 컴파일된 메소드에 해당하는 행이 포함됩니다.

   ```
   + (hot) java/lang/Math.max(II)I @ 0x10C11DA4-0x10C11DDD
   ```

   더하기 부호로 시작하지 않는 행은 다음 단계에서 컴파일러가 무시하며 파일에서 이들을 제거할 수 있습니다. 공유 클래스 캐시에서 로드된 AOT 코드에 대한 메소드는 +(AOT load)로 시작합니다.
2. JIT 또는 AOT 매개변수 **limitFile=**(<*filename*>,<*m*>,<*n*>)와 함께 프로그램을 다시 실행하십시오. 여기서 <*filename*>은 한계 파일에 대한 경로이며 <*m*> 및 <*n*>은 컴파일할 한계 파일에 있는 첫 번째 및 마지막 메소드를 표시하는 행 번호입니다. 컴파일러는 한계 파일의 <*m*>행부터 <*n*>행까지 나열된 메소드만 컴파일합니다. 한계 파일에 없는 메소드와 범위 밖의 행에 나열된 메소드는 컴파일되지 않습니다. 그리고 이러한 메소드에 대해 공유 데이터 캐시에 있는 AOT 코드는 로드되지 않습니다. 프로그램이 더 이상 실패하지 않으면 마지막 반복에서 제거한 하나 이상의 메소드가 실패의 원인입니다.
3. 필요한 횟수만큼 <*m*>과 <*n*>에 대해 다른 값을 적용하고 이 프로세스를 반복하여, 실패를 트리거하기 위해 컴파일해야 하는 최소 세트의 메소드를 찾으십시오. 매번 선택한 행의 수를 반으로 줄여서 실패한 메소드에 대해 2진 검색을 수행하십시오. 종종 파일을 한 행으로 줄일 수 있습니다.

다음에 수행할 작업

실패한 메소드의 위치를 찾으면 실패한 메소드에 대해서만 JIT 또는 AOT 컴파일러를 사용하지 마십시오. 예를 들어, 메소드 java/lang/Math.max(II)I가 **optLevel=hot**과 함께 JIT 컴파일되었을 때, 프로그램을 실패하게 한다면 다음을 적용하여 프로그램을 실행할 수 있습니다.

```
-Xjit:{java/lang/Math.max(II)I}(optLevel=warm,count=0)
```

실패 메소드만 최적화 레벨 『warm』으로 컴파일하고 다른 메소드들은 일반적으로 컴파일합니다.

메소드가 『noOpt』에서 JIT 컴파일되었을 때 실패하는 경우 **exclude**={<*method*>} 매개변수를 사용하여 컴파일에서 이를 함께 제거할 수 있습니다.

```
-Xjit:exclude={java/lang/Math.max(II)I}
```

메소드가 AOT 코드 공유 데이터 캐시에서 로드되었을 때 프로그램을 실패하게 하는 경우, AOT 로딩에서 **exclude**={<*method*>} 매개변수를 사용하여 메소드를 제외하십시오.

```
-Xaot:exclude={java/lang/Math.max(II)I}
```

AOT 메소드는 **admincache** 채우기 단계 중에 공유 클래스 캐시로만 컴파일됩니다. AOT 로딩을 방지하는 것이 이러한 메소드에 대한 문제 진단을 접근하는 가장 좋은 방법입니다.

JIT 및 AOT 컴파일 장애 식별:

JIT 컴파일러 장애의 경우 JIT 컴파일러가 메소드 컴파일을 시도할 때 장애가 발생했는지 여부를 판별하기 위해 오류 출력을 분석하십시오.

JVM 크래쉬의 경우 JIT 라이브러리(`libj9jit26.so`)에서 장애가 발생하였다면 JIT 컴파일러가 메소드 컴파일을 시도하는 중에 실패했을 수 있습니다.

이 예와 유사한 오류 출력을 보고 실패한 메소드를 식별할 수 있습니다.

```
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;)
```

중요한 행은 다음과 같습니다.

vmState=0x00050000

JIT 컴파일러가 코드를 컴파일하고 있음을 표시합니다. vmState 코드 번호 목록의 경우 IBM SDK for Java 7사용자 안내서(http://publib.boulder.ibm.com/infocenter/

java7sdk/v7r0/topic/com.ibm.java.lnx.70.doc/diag/tools/javadump_tags_info.html)의 Java 덤프 태그 테이블을 참조하십시오.

Module=/home/test/sdk/jre/bin/libj9jit26.so
JIT 컴파일러 모듈인 libj9jit26.so에서 발생한 오류를 표시합니다.

Method_being_compiled=
Java 메소드가 컴파일되고 있음을 표시합니다.

출력에서 실패한 메소드를 표시하지 않으면 다음 추가 설정과 함께 **verbose** 옵션을 사용하십시오.

```
-Xjit:verbose={compileStart|compileEnd}
```

이러한 **verbose** 설정은 JIT 또는 AOT 컴파일러가 메소드 컴파일을 시작 및 종료한 시기를 보고합니다. JIT 또는 AOT 컴파일러가 특정 메소드에서 실패하면(즉, 컴파일링을 시작했으나 종료하기 전에 크래쉬되면) 에서 **exclude** 매개변수를 사용하여 JIT 또는 AOT 컴파일에서 이를 제외시키십시오(142 페이지의 『실패 메소드 찾기』 참조). AOT 컴파일에 대한 문제의 경우 **exclude** 옵션을 사용하기 전에 공유 클래스 캐시를 폐기하십시오. 제외 메소드가 크래쉬를 방지한다면 서비스 팀에서 사용자 문제를 정정하는 동안 사용할 수 있는 임시 해결 방법으로 사용하십시오.

실시간 모드가 아닌 경우의 AOT 컴파일 실패 식별:

실시간 모드가 아닌 경우의 AOT 문제점 판별은 JIT 문제점 판별과 매우 유사합니다.

이 태스크 정보

JIT의 경우처럼 먼저 애플리케이션 실행 시 AOT 코드가 사용되지 않도록 하는 **-Xnoaot**를 사용하여 애플리케이션을 실행하십시오.

이렇게 하여 문제점이 수정되는 경우 142 페이지의 『실패 메소드 찾기』에 설명된 것과 동일한 기술로 애플리케이션 런타임이 아닌 AOT 빌드 시에 **-Xaot** 옵션을 제공하여 AOT JAR 파일을 다시 빌드하십시오.

실시간 모드에서 AOT 컴파일 실패 식별:

AOT 문제점 판별에서는 admincache 도구를 사용하여 문제점을 찾습니다.

이 태스크 정보

애플리케이션 런타임에 발생하는 JIT 컴파일 실패와 대조적으로, AOT 컴파일 실패는 admincache 채우기 단계에서 발생합니다.

문제점이 발생한 위치를 알려면 **-Xnoaot**과 함께 admincache 도구를 실행하십시오. 그러면 애플리케이션이 AOT(Ahead-of-Time) 컴파일된 코드로 실행되지 않습니다.

-Xnoaot 옵션 사용으로 문제점을 수정할 경우 원본 크래쉬의 출력을 검토하십시오. 해당 출력은 문제점의 원인이 된 메소드를 식별하는 정보를 제공합니다. 다음과 비슷한 행을 찾으십시오.

```
Method_being_compiled=myAppClass.main(Ljava/lang/String;)V
```

문제점을 방지하려면 이 메소드를 AOT(Ahead-of-Time) 컴파일에서 제외해야 합니다. 다음과 같이 옵션을 admincache 명령행에 추가하면 제외됩니다.

```
-Xaot:exclude={myAppClass.main(Ljava/lang/String;)V}
```

이렇게 제외하면 문제점 메소드가 AOT 컴파일되지 않습니다.

단기 실행 애플리케이션 성능

IBM JIT 컴파일러는 주로 서버에서 사용되는 장기 실행 애플리케이션을 위해 조정됩니다. 비실시간 모드에서 **-Xquickstart** 명령행 옵션을 사용해서 적은 수의 메소드로 처리를 집중할 수 없는 애플리케이션의 경우에 단기 실행 어플리케이션의 성능을 향상하도록 사용할 수 있습니다.

-Xquickstart는 JIT 컴파일러가 기본적으로 낮은 최적화 레벨을 사용하도록 하여 적은 수의 메소드를 컴파일하게 합니다. 적은 수의 컴파일을 좀 더 빠르게 수행하면 애플리케이션 시작 시간을 단축할 수 있습니다. AOT 컴파일러가 활성(공유 클래스와 AOT 컴파일이 모두 사용 가능)이면 **-Xquickstart**는 컴파일을 위해 선택된 모든 메소드가 AOT 컴파일되도록 하여 후속 실행의 시작 시간을 단축합니다. **-Xquickstart**는 대량의 처리 자원을 사용하는 메소드가 포함된 장기 실행 애플리케이션에 사용될 경우 성능을 저하시킬 수 있습니다. **-Xquickstart** 구현은 향후 릴리스에서 변경될 수 있습니다.

JIT 임계값을 조정하여(시행 착오를 거쳐) 시작 시간을 향상시킬 수 있습니다. 자세한 정보는 141 페이지의 『선택적으로 JIT 컴파일러 사용 안함』의 내용을 참조하십시오.

-Xquickstart는 **-Xrealtime**에서의 AOT 코드 사용에 영향이 없습니다.

대기 기간 동안 JVM 동작

JIT 샘플링 스레드를 끄기 위해 **-XsamplingExpirationTime** 옵션을 사용하여 대기 JVM에서 소비하는 CPU 주기를 줄일 수 있습니다.

JIT 샘플링 스레드는 공통적으로 사용되는 메소드를 감지하기 위해, 실행 중인 Java 애플리케이션을 프로파일합니다. 샘플링 스레드의 메모리와 프로세서 사용은 미미하며 프로파일링 빈도는 JVM이 대기 상태일 경우 자동으로 감축됩니다.

일부 환경에서 사용자는 대기 JVM이 CPU 주기를 소비하지 않기를 원할 수 있습니다. 그렇게 하려면 **-XsamplingExpirationTime**<*time*> 옵션을 지정하십시오. <*time*>을 샘플링 스레드를 실행하려는 시간(초)으로 설정하십시오. 이 옵션은 주의해서 사용

해야 합니다. 샘플링 스레드를 끄게 되면 다시 활성화할 수 없습니다. 샘플링 스레드가 충분한 시간 동안 실행되어 중요한 최적화를 식별하도록 하십시오.

진단 콜렉터

진단 콜렉터는 문제점 이벤트에 사용할 Java 진단 파일을 수집합니다.

IBM 서비스에 필요한 파일을 수집하면 보고된 문제점을 해결하는 데 드는 시간을 줄일 수 있습니다. IBM SDK for Java 7 사용자 안내서에는 진단 콜렉터의 사용에 대한 자세한 정보가 포함되어 있습니다.

이 정보는 여기에서 찾을 수 있습니다: IBM SDK for Java 7 - 진단 콜렉터.

가비지 콜렉터 진단

이 섹션은 가비지 콜렉션 문제점의 진단 방법을 설명합니다.

IBM SDK for Java 7 사용자 안내서에서는 가비지 콜렉터 문제점의 진단에 필요한 정보가 있으며 다음 내용을 다루고 있습니다.

- 상세 가비지 콜렉션 로깅
- **-Xtgc**를 사용하여 가비지 콜렉션 추적

이 정보는 여기에서 찾을 수 있습니다: IBM SDK for Java 7 - 가비지 콜렉터 진단.

IBM WebSphere Real Time for RT Linux 메트로놈 가비지 콜렉터에 대한 보충 정보는 다음 섹션에 있습니다.

메트로놈 가비지 콜렉터 문제점 해결

명령행 옵션을 사용하여 메트로놈 가비지 콜렉션 빈도, 메모리 부족 예외 및 명시적 시스템 호출에서 메트로놈 동작을 제어할 수 있습니다.

verbose:gc 정보 사용:

-verbose:gc 옵션을 **-Xgc:verboseGCCycleTime=N** 옵션과 함께 사용하여 메트로놈 가비지 콜렉터 활동에 대한 정보를 콘솔에 작성할 수 있습니다. 표준 JVM의 **-verbose:gc** 출력에 있는 모든 XML 특성이 작성되거나 메트로놈 가비지 콜렉터 출력에 적용되는 것은 아닙니다.

힙에서 최소, 최대 및 중간 여유 공간을 보려면 **-verbose:gc** 옵션을 사용하십시오. 이 방식으로 활동 레벨과 힙 사용을 검사하고 필요하면 값을 조정할 수 있습니다. **-verbose:gc** 옵션은 메트로놈 통계를 콘솔에 작성합니다.

-Xgc:verboseGCCycleTime=N 옵션은 정보 검색 빈도를 제어하고, 요약을 덤프하는 시간(밀리초)를 판별합니다. N의 기본값은 1000밀리초입니다. 주기 시간이 해당 시간에 정확하게 요약이 덤프됨을 의미하지는 않지만 이 시간 기준을 충족하는 마지막 가비지 콜

렉션 이벤트가 전달됩니다. 이 통계의 수집 및 표시로 메트로놈 가비지 콜렉터 일시정지 시간이 증가되고 N이 작아질수록 일시정지 시간이 길어질 수 있습니다.

퀀텀은 단일 메트로놈 가비지 콜렉터 활동 기간으로, 애플리케이션의 인터럽트 또는 일시정지 시간을 일으킵니다.

verbose:gc 출력 예제

입력:

```
java -Xrealtime -verbose:gc -Xgc:verboseGCCycleTime=N myApplication
```

이 예제는 버전 및 가비지 콜렉션 설정을 포함한 verbose:gc의 초기 출력을 보여줍니다.

```
<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>
```

가비지 콜렉션을 트리거하면 trigger start 이벤트와 많은 heartbeat 이벤트가 차례로 발생하고, 트리거를 충족하면 trigger end 이벤트가 발생합니다. 이 예제는 트리거된 가비지 콜렉션 주기를 verbose:gc 출력으로 보여줍니다.

```
| <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" />
```

다음과 같은 이벤트 유형이 발생할 수 있습니다.

| **<trigger-start ...>**

| 가비지 콜렉션 주기의 시작이며 사용된 메모리 양이 트리거 임계값보다 커질 때

입니다. 기본 임계값은 힙의 50%입니다. intervalms 속성은 이전 trigger end 이벤트(id-1)와 이 trigger start 이벤트 사이의 간격입니다.

<trigger-end ...>
가비지 콜렉션 주기를 사용된 메모리 *크기*를 트리거 임계값 아래로 낮추었습니다. 가비지 콜렉션 주기가 종료되었지만 사용된 메모리가 트리거 임계값 아래로 낮아지지 않으면 새 가비지 콜렉션 주기가 동일한 컨텍스트 ID로 시작됩니다. 각 trigger start 이벤트에 대해 동일한 컨텍스트 ID의 일치하는 trigger end 이벤트가 있습니다. intervalms 속성은 이전 trigger start 이벤트와 현재 trigger end 이벤트 사이의 간격입니다. 이 기간 동안 사용된 메모리가 트리거 임계값 아래로 낮아질 때까지 하나 이상의 가비지 콜렉션 주기가 완료됩니다.

<gc-op id="28" type="heartbeat"...>
관련 기간 동안 모든 가비지 콜렉션 퀀텀에 대한 정보(메모리 및 시간)를 수집하는 정기적 이벤트입니다. 일치하는 trigger start 및 trigger end 이벤트 쌍 사이(즉, 활성 가비지 콜렉션 주기가 진행 중일 때)에서만 하트비트 이벤트가 발생할 수 있습니다. intervalms 속성은 이전 하트비트 이벤트(id -1)와 이 하트비트 이벤트 사이의 간격입니다.

<gc-op id="29" type="syncgc"...>
동기 (비결정적) 가비지 콜렉션 이벤트입니다. 151 페이지의 『동기 가비지 콜렉션』의 내용을 참조하십시오.

이 예제에서 XML 태그는 다음과 같은 의미를 가집니다.

<quanta ...>
일시정지 기간(밀리초)을 포함한 하트비트 간격 동안 퀀텀 일시정지 시간에 대한 요약입니다.

<free-mem type="heap" ...>
하트비트 간격 동안 여유 힙 공간 *크기*에 대한 요약으로, 각 가비지 콜렉션 퀀텀이 끝날 때 샘플링됩니다.

<classunload-info classloadersunloaded=9 classesunloaded=156 />
하트비트 간격 동안 로드 해제된 클래스 로더 및 클래스의 수입니다.

<references type="weak" cleared="660 />
하트비트 간격 동안 해제된 Java 참조 오브젝트의 수 및 유형입니다.

주:

- 두 하트비트 사이의 간격에 한 개의 가비지 콜렉션 퀀텀만 발생한 경우, 이 퀀텀이 끝날 때에만 여유 메모리가 샘플링됩니다. 따라서 하트비트 요약에 제공된 최소값, 최대값 및 중간값이 모두 동일합니다.

- 힙이 가바지 콜렉션 활동이 필요할 만큼 충분히 차지 않은 경우에는 두 하트비트 사이의 간격이 지정된 주기 시간보다 훨씬 길 수 있습니다. 예를 들어, 프로그램에서 몇 초마다 한 번만 가비지 콜렉션 활동이 필요한 경우 몇 초마다 한 번만 하트비트가 표시될 가능성이 높습니다.
- 가비지 콜렉션 활동이 일어날 만큼 충분히 차지 않은 힙에서는 가비지 콜렉션이 작동하지 않기 때문에 가비지 콜렉션 간격이 지정된 주기 시간보다 더 길어질 수 있습니다. 예를 들어, 프로그램에서 몇 초마다 한 번만 가비지 콜렉션 활동이 필요한 경우 몇 초마다 한 번만 하트비트가 표시될 가능성이 높습니다.

  동기 가비지 콜렉션이나 우선순위 변경 같은 이벤트가 발생하면 이벤트 세부사항과 하트비트 같은 보류 중인 이벤트가 출력으로 즉시 생성됩니다.
- 특정 기간 동안 최대 가비지 콜렉션 퀀텀이 너무 크면 **-Xgc:targetUtilization** 옵션을 사용하여 대상 이용률을 줄일 수 있습니다. 이 조치로 가비지 콜렉터에 작업 시간을 더 부여할 수 있습니다. 또는 **-Xmx** 옵션으로 힙 *크기*를 늘일 수도 있습니다. 이와 비슷하게, 애플리케이션이 현재 보고되는 것보다 긴 지연을 허용할 경우 대상 이용률을 높이거나 힙 *크기*를 줄일 수 있습니다.
- **-Xverbosegclog:<file>** 옵션을 사용하면 출력이 콘솔 대신 로그 파일로 경로 재지정됩니다. 예를 들어, **-Xverbosegclog:out**은 **-verbose:gc** 출력을 *out* 파일에 작성합니다.
- thread-priority에 나열된 우선순위는 Java 스레드 우선순위가 아닌 기본 운영 체제 스레드 우선순위입니다.

동기 가비지 콜렉션

동기 (비결정적) 가비지 콜렉션이 발생하면 **-verbose:gc** 로그에도 항목이 작성됩니다. 이 이벤트의 원인은 다음 3가지입니다.

- 코드에 명시적 System.gc() 호출.
- JVM에서 메모리가 부족하여 OutOfMemoryError 조건을 피하기 위해 동기 가비지 콜렉션을 수행합니다.
- JVM이 가비지 콜렉션이 지속되는 동안 종료됩니다. JVM은 콜렉션을 취소할 수 없으므로 콜렉션을 동기적으로 완료한 다음 존재합니다.

System.gc() 항목의 예제입니다.

```
<gc-op id="9" type="syncgc" timems="12.92" contextid="8" timestamp="2011-07-12T09:41:40.808">
  <syncgc-info reason="system GC" totalBytesRequested="260" exclusiveaccessTimeMs="0.009"
    threadPriority="11" />
  <free-mem-delta type="heap" bytesBefore="22085440" bytesAfter="136023450" />
  <free-mem-delta type="immortal" bytesBefore="62324800" bytesAfter="62324800"/>
  <classunload-info classloadersunloaded="54" classesunloaded="234" />
  <references type="soft" cleared="21" dynamicThreshold="29" maxThreshold="32" />
  <references type="weak" cleared="523" />
  <finalization enqueued="124" />
</gc-op>
```

JVM 종료의 결과로 나타나는 동기 가비지 콜렉션 항목의 예제입니다.

```
<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>
```

이 예제에서 XML 태그 및 속성은 다음과 같은 의미를 가집니다.

<gc-op id="9" type="syncgc" timems="6.439" ...
: 이 행은 이벤트 유형이 동기 가비지 콜렉션임을 나타냅니다. timems 속성은 동기 가비지 콜렉션 기간(밀리초)입니다.

<syncgc-info reason="..."/>
: 동기 가비지 콜렉션의 원인입니다.

<free-mem-delta.../>
: 동기 가비지 콜렉션 전후에 여유 Java 힙 메모리(바이트)입니다.

<finalization .../>
: 완료 대기 중인 오브젝트의 수입니다.

<classunload-info .../>
: 하트비트 간격 동안 로드 해제된 클래스 로더 및 클래스의 수입니다.

<references type="weak" cleared="53" .../>
: 하트비트 간격 동안 해제된 Java 참조 오브젝트의 수 및 유형입니다.

메모리 부족 조건이나 VM 종료로 인한 동기 가비지 콜렉션은 가비지 콜렉터가 활성인 경우에만 발생할 수 있습니다. 바로 이어질 필요는 없지만 이전에 trigger start 이벤트가 선행해야 합니다. 일부 하트비트 이벤트는 trigger start 이벤트와 synchgc 이벤트 사이에 발생합니다. System.gc()에 의해 발생한 동기 가비지 콜렉션은 언제든지 발생할 수 있습니다.

모든 GC 퀀텀 추적

GlobalGCStart 및 GlobalGCEnd 추적포인트를 사용하여 개별 GC 퀀텀을 추적할 수 있습니다. 이 추적포인트는 동기 가비지 콜렉션을 포함한 모든 메트로놈 가비지 콜렉터 활동이 시작하고 끝날 때 생성됩니다. 이 추적포인트의 출력은 다음과 유사합니다.

```
03:44:35.281 0x833cd00 j9mm.52 - GlobalGC start: globalcount=3

03:44:35.284 0x833cd00 j9mm.91 - GlobalGC end: workstackoverflow=0 overflowcount=0
```

우선순위 변경

요약 외에 가비지 콜렉터 스레드 우선순위가 변경되면(애플리케이션에서 스레드 우선순위가 변경되거나 하나 이상의 스레드가 종료되는 경우) 항목이 **-verbose:gc** 로그에 기

록됩니다. 나열된 우선순위는 Java 스레드 우선순위가 아닌 기본 OS 스레드 우선순위입니다. 가비지 콜렉터 스레드 우선순위 변경 항목의 예제입니다.

```
<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>
```

가비지 콜렉터 스레드 우선순위와 관련된 추적포인트 정보를 생성하여 우선순위 변경사항을 실시간으로 추적할 수 있습니다. 이 출력은 다음과 유사합니다.

```
15:58:25.493*0x8286e00    j9mm.102       - setGCThreadPriority() called with newGCThread
```

이 출력은 다음과 같이 ID를 사용하여 사용 가능하게 합니다. **-Xtrace:iprint=tpnid{j9mm.102}**

메모리 부족 항목

메모리 영역 중 하나에서 여유 공간이 부족하면 OutOfMemoryError 예외가 발생하기 전에 **-verbose:gc** 로그에 항목이 기록됩니다. 이 출력의 예제는 다음과 같습니다.

```
<out-of-memory id="71" timestamp="2011-07-23T08:32:51.435" memorySpaceName="Scoped"
  memorySpaceAddress="080EED9C"/>
```

기본적으로 Java 덤프가 OutOfMemoryError 예외의 결과로 생성됩니다. 이 덤프는 프로그램에 사용된 메모리 영역에 대한 정보를 포함하고 있습니다. **-verbose:gc** 출력에 나온 J9MemorySpace 값과 함께 덤프에 있는 이 정보를 사용하여 공간이 부족한 특정 메모리 영역을 식별할 수 있습니다.

```
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)
```

이전 예제에서 -verbose:gc 출력(0x080EED9C)에 지정된 메모리 공간 ID가 Java 덤프의 범위 메모리 영역 ID와 일치할 수 있습니다. -verbose:gc 출력은 OutOfMemoryError가 영구, 범위 또는 힙 메모리에서 발생했는지 여부를 표시하므로 여러 개의 범위가 있고 메모리가 부족한 항목을 식별해야 하는 경우 유용합니다.

메모리 부족 조건에서 메트로놈 가비지 콜렉터 동작:

기본적으로 메트로놈 가비지 콜렉터는 JVM에서 메모리가 부족해지면 무제한 비결정적 가비지 콜렉션을 트리거합니다. 비결정적 동작을 방지하려면 **-Xgc:noSynchronousGCOnOOM** 옵션을 사용하여 JVM에서 메모리가 부족해질 때 OutOfMemoryError를 발생시키십시오.

단일 조작에서 모든 가능한 가비지를 수집할 때까지 기본 무제한 콜렉션이 실행됩니다. 대개는 일시정지 시간이 정상 메트로놈 증분 퀀텀보다 몇 밀리초 큽니다.

관련 정보

동기 가비지 콜렉션을 분석하기 위해 -Xverbose:gc 사용

명시적 System.gc() 호출에서 메트로놈 가비지 콜렉터 동작:

가비지 콜렉션 주기가 진행 중이면 System.gc()를 호출할 때 메트로놈 가비지 콜렉터가 동기적으로 주기를 완료합니다. 진행 중인 가비지 콜렉션 주기가 없으면 System.gc()를 호출할 때 전체 동기 주기가 수행됩니다. 제어 상태로 힙을 정리하려면 System.gc()를 사용하십시오. 이 조작은 리턴하기 전에 전체 가비지 콜렉션을 완료하기 때문에 비결정적입니다.

이 비결정적 지연이 용인되지 않는 일부 애플리케이션은 System.gc() 호출을 포함한 벤더 소프트웨어를 호출합니다. 모든 System.gc() 호출을 사용 불가능하게 하려면 **-Xdisableexplicitgc** 옵션을 사용하십시오.

System.gc() 호출에 대한 상세 가비지 콜렉션 출력은 『시스템 가비지 콜렉션』 이유를 포함하고 있으며 지속 기간이 깁니다.

```
| <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>
```

공유 클래스 진단

발생하는 문제점을 진단하는 방법을 이해하면 공유 클래스 모드 사용에 도움이 됩니다.

공유 클래스에 대한 소개는 JVM간 클래스 데이터 공유의 내용을 참조하십시오.

IBM SDK for Java 7 사용자 안내서에는 공유 클래스 문제점 진단에 필요한 정보가 있으며 다음 내용을 다루고 있습니다.

- 공유 클래스 배치
- 런타임 바이트 코드 수정 처리
- 동적 업데이트 이해
- Java Helper API 사용

- 공유 클래스 진단 출력 이해
- 공유 클래스로 문제점 디버깅

이 정보는 여기에서 찾을 수 있습니다: IBM SDK for Java 7 - 공유 클래스 진단.

IBM SDK for Java 7 사용자 안내서의 일부 자료는 IBM WebSphere Real Time for RT Linux에 적용되지 않을 수 있습니다. 특히:

- 실시간 모드에서 애플리케이션은 읽기-쓰기 액세스가 아닌 공유 클래스 캐시에 대한 읽기 전용 액세스만 있습니다.
- 캐시는 **admincache** 도구를 사용해서만 수정될 수 있습니다.
- 비지속적 캐시는 실시간 모드에서 사용할 수 없습니다.

JVMTI 사용

JVMTI는 JVM과 원시 에이전트 간의 통신이 가능한 양방향 인터페이스입니다. 이는 JVMDI와 JVMPI 인터페이스를 대체합니다.

JVMT를 사용하여 써드파티에서 JVM에 대한 모니터링, 프로파일링 및 디버깅 도구를 개발할 수 있습니다. 인터페이스에는 에이전트가 JVM에 필요한 정보의 종류를 통지하는 메커니즘이 포함되어 있습니다. 인터페이스는 또한 관련 통지를 수신하는 수단도 제공합니다. 몇몇 에이전트를 언제든 JVM에 부착할 수 있습니다.

IBM SDK for Java 7 사용자 안내서에는 JVMTI에 대한 IBM 확장기능에 관한 API 참조 섹션을 포함한 자세한 JVMTI 사용 정보가 포함되어 있습니다.

이 정보는 여기에서 찾을 수 있습니다: IBM SDK for Java 7 - JVMTI 사용.

DTFJ(Diagnostic Tool Framework for Java) 사용

DTFJ(Diagnostic Tool Framework for Java)는 IBM의 Java API(Application Programming Interface)로 Java 진단 도구 빌드를 지원하는 데 사용됩니다. DTFJ는 시스템 덤프 또는 Java 덤프의 데이터로 작업합니다.

IBM SDK for Java 7 사용자 안내서에는 자세한 DTFJ 정보가 포함되어 있습니다. 다음 링크를 사용하십시오. Java용 진단 도구 프레임워크 사용

IBM Monitoring and Diagnostic Tools for Java - Health Center 사용

IBM Monitoring and Diagnostic Tools for Java - Health Center는 실행 중인 JVM(Java Virtual Machine)의 상태를 모니터하는 데 사용되는 진단 도구입니다.

IBM Monitoring and Diagnostic Tools for Java - Health Center에 대한 정보는 developerWorks® 및 InfoCenter에서 사용 가능합니다.

제 10 장 참조

이 주제에서는 WebSphere Real Time for RT Linux와 함께 사용할 수 있는 옵션 및 클래스 라이브러리를 보여줍니다.

명령행 옵션

Java를 시작할 때 명령행에서 옵션을 지정할 수 있습니다. 기본 옵션은 가장 적합한 일반 용도에 맞게 선택되었습니다.

Java 옵션 및 시스템 특성 지정

Java 옵션 및 시스템 특성을 지정하는 방법은 3가지가 있습니다.

이 태스크 정보

다음 방법으로 Java 옵션 및 시스템 특성을 지정할 수 있습니다. 우선 순위에 따라 다음 방법을 사용합니다.

1. 명령행에서 옵션 또는 특성 지정. 예를 들면 다음과 같습니다.

   ```
   java -Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump MyJavaClass
   ```

2. 옵션이 있는 파일을 작성하고 **-Xoptionsfile=**<*filename*> 옵션을 사용하여 명령행에서이 파일을 지정합니다.

   옵션 파일에서, 새 행에 각 옵션을 지정하십시오. 백슬래시('\') 문자를 연속 문자로 사용하는 경우 하나의 옵션을 여러 행에 걸쳐 지정할 수 있습니다. 명령행을 정의하려면 '#' 문자를 사용하십시오. 옵션 파일에서 **-classpath**를 지정할 수 없습니다. 옵션 파일의 예:

   ```
   #My options file
   -X<option1>
   -X<option2>=\
   <value1>,\
   <value2>
   -D<sysprop1>=<value1>
   ```

3. 옵션을 포함한 **IBM_JAVA_OPTIONS**라는 환경 변수 작성. 예를 들면 다음과 같습니다.

   ```
   export IBM_JAVA_OPTIONS="-Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump"
   ```

명령행에서 지정하는 마지막 옵션은 첫 번째 옵션보다 우선순위가 높습니다. 예를 들어, **-Xint -Xjit myClass** 옵션을 지정하는 경우 **-Xjit** 옵션이 **-Xint**보다 우선하여 사용됩니다.

시스템 특성

애플리케이션에서 시스템 특성을 사용할 수 있으며, 런타임 환경에 대한 정보를 제공합니다.

com.ibm.jvm.realtime

이 특성은 Java 애플리케이션이 WebSphere Real Time for RT Linux 환경에서 실행되는지 여부를 판별할 수 있도록 지원합니다.

애플리케이션이 IBM WebSphere Real Time for RT Linux 런타임에서 실행되고 **-Xrealtime** 옵션을 사용하여 시작한 경우, **com.ibm.jvm.realtime** 특성의 값이 『hard』입니다.

애플리케이션이 IBM WebSphere Real Time for RT Linux 환경에서 실행되지만 **-Xrealtime** 옵션을 사용하여 시작하지 않은 경우, **com.ibm.jvm.realtime** 특성이 설정되지 않습니다.

애플리케이션이 IBM WebSphere Real Time 런타임에서 실행되면 **com.ibm.jvm.realtime** 특성의 값이 『soft』입니다.

표준 옵션

표준 옵션에 대한 정의입니다.

-agentlib:<*libname*>**[=**<*options*>**]**

원시 에이전트 라이브러리 <*libname*>을 로드합니다(예: **-agentlib:hprof**). 자세한 정보를 보려면 명령행에 **-agentlib:jdwp=help** 및 **-agentlib:hprof=help**를 지정하십시오.

-agentpath:*libname***[=**<*options*>**]**

전체 경로 이름별로 원시 에이전트 라이브러리를 로드합니다.

-assert

assert 관련 옵션에 대한 도움말을 인쇄합니다.

-cp or -classpath <*directories and .zip or .jar files separated by* **:**>

애플리케이션 클래스 및 자원의 검색 경로를 설정합니다. **-classpath** 및 **-cp**가 사용되지 않고 **CLASSPATH**가 설정되지 않은 경우, 기본적으로 사용자 클래스 경로는 현재 디렉토리(.)입니다..

-D<property_name>=<*value*>

시스템 특성을 설정합니다.

-help 또는 **-?**

사용법 메시지를 인쇄합니다.

-javaagent:<*jarpath*>**[=**<*options*>**]**

Java 프로그래밍 언어 에이전트를 로드합니다. 자세한 정보는 java.lang.instrument API 문서를 참조하십시오.

-jre-restrict-search

버전 검색에 사용자 개인용 JRE를 포함합니다.

-no-jre-restrict-search

버전 검색에서 사용자 개인용 JRE를 제외합니다.

-showversion

제품 버전을 인쇄하고 계속 진행합니다.

-verbose:[*class*,*gc*,*dynload*,*sizes*,*stack*,*jni***]**

상세 출력을 사용 가능하게 합니다.

-verbose:class

로드한 각 클래스의 stderr에 항목을 기록합니다.

-verbose:gc

147 페이지의 『verbose:gc 정보 사용』의 내용을 참조하십시오.

-verbose:dynload

다음을 포함하여 JVM에서 각 클래스를 로드할 때 상세한 정보를 제공합니다.

- 클래스 이름과 패키지
- `.jar` 파일에 있던 클래스 파일의 경우, `.jar`의 이름과 디렉토리 경로
- 클래스를 로드할 때 소요된 시간과 클래스 사이즈의 세부사항

데이터가 stderr에 작성됩니다. 출력 예는 다음과 같습니다.

```
<Loaded java/lang/String from /myjdk/sdk/jre/lib/i386/softrealtime/jclSC16
<Class size 17258; ROM size 21080; debug size 0>
<Read time 27368 usec; Load time 782 usec; Translate time 927 usec>
```

주: 공유 클래스 캐시에서 로드된 클래스는 **-verbose:dynload** 출력에 표시되지 않습니다. 그러한 클래스에 대한 정보는 **-verbose:class** 옵션을 사용하십시오.

-verbose:sizes

JVM에서 스택 및 힙에 사용된 메모리 양을 설명하는 정보를 stderr에 기록합니다.

-verbose:stack

Java 및 C 스택 사용량을 설명하는 정보를 stderr에 기록합니다.

-verbose:jni

애플리케이션 및 JVM에서 호출된 JNI 서비스를 설명하는 정보를 stderr에 기록합니다.

-version

실시간 이외 모드의 버전 정보를 인쇄합니다. -Xrealtime 옵션과 함께 사용하면 실시간 모드의 버전 정보를 인쇄합니다.

-version:<*value*>

지정된 버전을 실행해야 합니다.

-X 비표준 옵션에 대한 도움말을 인쇄합니다.

비표준 옵션

접두부에 **-X**가 있는 옵션은 비표준 옵션이며 별도의 통지없이 변경될 수 있습니다.

IBM SDK for Java 7 사용자 안내서에는 비표준 옵션에 대한 자세한 정보가 포함되어 있습니다. 이 정보는 여기에서 찾을 수 있습니다: IBM SDK for Java 7 - 명령행 옵션.

다음 섹션에 IBM WebSphere Real Time for RT Linux에 대한 보충 정보가 있습니다.

실시간 옵션

WebSphere Real Time for RT Linux에 사용되는 **-Xrealtime** 옵션의 정의입니다.

다음 **-X** 옵션은 WebSphere Real Time for RT Linux 환경에 적용 가능합니다.

-Xrealtime

실시간 모드를 시작합니다. 메트로놈 가비지 콜렉터를 실행하고 Real-Time Specification for Java(RTSJ) 서비스를 사용할 경우 필요합니다. 이 옵션을 지정하지 않으면 IBM SDK and Runtime Environment for Linux 플랫폼, Java 2 Technology, 버전 7에 동등한 실시간 이외 모드에서 JVM이 시작됩니다를 사용합니다.

-Xrealtime 옵션은 **-Xgcpolicy:metronome**과 상호 변경 가능합니다. 실시간 모드를 사용하려면 둘 중 하나를 지정합니다.

AOT(Ahead-of-Time) 옵션

AOT(Ahead-of-Time) 옵션의 정의입니다.

용도

옵션이 지정되지 않음:

인터프리터 및 동적으로 컴파일된 코드와 함께 실행합니다. AOT 코드를 발견하면 사용되지 않습니다. 대신, 필요에 따라 동적으로 컴파일됩니다. 실시간 이외 애플리케이션 및 일부 실시간 애플리케이션에 특히 유용합니다. 이 옵션은 최적의 성능 및 처리량을 제공하지만 컴파일 실행 시 런타임에 비결정적 지연이 발생할 수 있습니다.

-Xjit: 이 옵션은 기본값과 동일합니다.

-Xint: 인터프리터만 실행하고, 사전 컴파일된 jar 파일에 있는 AOT용으로 작성된 코드를 무시하며 동적 컴파일러를 실행하지 않습니다. 이 모드는 컴파일과 관련된 것으로 의심되는 문제점을 디버깅하거나 컴파일에서 이점이 파생되지 않는 매우 짧은 일괄처리 애플리케이션의 경우를 제외하고는 자주 필요하지 않습니다.

-Xnojit:

인터프리터를 실행하고 사전 컴파일된 jar 파일에 있는 경우 AOT용으로 작성된 코드를 사용합니다. 동적 컴파일러는 실행하지 않습니다. 이 모드는 컴파일 때문에 런타임에 비결정적 지연이 발생하지 않도록 할 경우 일부 실시간 애플리케이션에서 유용하게 작동합니다. AOT용으로 작성된 코드는 **-Xrealtime** 옵션으로 실행할 경우에만 사용할 수 있습니다. 표준 JVM에서 실행될 경우(즉, **-Xrealtime**을 지정하지 않음) 지원되지 않습니다.

예 `java -Xrealtime -Xnojit outputtest.jar`.

메트로놈 가비지 콜렉터 옵션

메트로놈 가비지 콜렉터 옵션의 정의입니다.

-Xgc:immortalMemorySize=*size*

영구 힙 영역의 크기를 지정합니다. 기본값은 16MB입니다.

-Xgc:scopedMemoryMaximumSize=*size*

범위 메모리 힙 영역의 크기를 지정합니다. 기본값은 8MB입니다.

-Xgc:synchronousGCOnOOM | -Xgc:nosynchronousGCOnOOM

힙 메모리가 부족할 때 가비지 콜렉션이 발생하는 한 경우입니다. 힙에 여유 공간이 없을 때 **-Xgc:synchronousGCOnOOM**을 사용하면 가비지 콜렉션에서 사용되지 않는 오브젝트를 제거하는 동안 애플리케이션이 중지됩니다. 여유 공간이 다시 부족해지면 가비지 콜렉션이 완료될 때까지 추가 시간을 허용하도록 대상 이용률을 낮추는 것을 고려하십시오. **-Xgc:nosynchronousGCOnOOM** 설정은 힙 메모리가 꽉 차면 애플리케이션이 중지되고 메모리 부족 메시지가 발행됨을 의미합니다. 기본값은 **-Xgc:synchronousGCOnOOM**입니다.

-Xnoclassgc

클래스 가비지 콜렉션을 사용 불가능하게 합니다. 이 옵션은 JVM에서 더 이상 사용되지 않는 Java 클래스와 연관된 스토리지의 가비지 콜렉션이 사용되지 않도록 합니다. 기본 동작은 **-Xnoclassgc**입니다.

-Xgc:targetUtilization=*N*

애플리케이션 이용률을 N%로 설정합니다. 가비지 콜렉터가 각 간격의 최대 (100-N)%를 사용하려고 합니다. 합리적인 값은 50-80%의 범위에 있습니다. 할당률이 낮은 애플리케이션은 90% 실행될 수 있습니다. 기본값은 70%입니다.

이 예제는 힙 메모리의 최대 *크기*가 30MB임을 보여줍니다. 애플리케이션의 대상 이용률이 75%이기 때문에 가비지 콜렉터가 각 간격의 25%를 사용하려고 합니다.

```
java -Xrealtime -Xmx30m -Xgc:targetUtilization=75 Test
```

-Xgc:threads=*N*

실행할 GC 스레드의 수를 지정합니다. 기본값은 1입니다.

-Xgc:verboseGCCycleTime=N

N은 요약 정보를 덤프하는 시간(밀리초)입니다.

주: 주기 시간이 해당 시간에 정확하게 요약 정보가 덤프됨을 의미하지는 않지만 이 시간 기준을 충족하는 마지막 가비지 콜렉션 이벤트가 전달됩니다.

-Xmx<*size***>**

Java 힙 *크기*를 지정합니다. 다른 가비지 콜렉션 전략과 달리, 실시간 메트로놈 GC는 힙 확장을 지원하지 않습니다. *초기* 또는 최대 힙 *크기* 옵션이 없습니다. 최대 힙 *크기*만 지정할 수 있습니다.

-Xthr:metronomeAlarm=os*xx*

메트로놈 가비지 콜렉터 알람 스레드가 실행되는 우선순위를 제어합니다.

여기서 *xx*는 11 - 89 범위의 숫자로, 메트로놈 알람 스레드가 실행되는 우선순위를 지정합니다. 알람 스레드가 실행되는 OS 우선순위를 수정할 때는 주의해야 합니다. OS 우선순위를 실시간 스레드의 우선순위보다 낮게 지정하면 가비지 콜렉터가 가비지를 할당하는 실시간 스레드보다 낮은 우선순위로 실행되기 때문에 OutOfMemory 오류가 발생합니다. 기본 메트로놈 가비지 콜렉터 알람 스레드는 OS 우선순위 89로 실행됩니다.

JVM 기본 설정

JVM이 실행되는 환경에 대한 변경사항이 없으면 기본 설정이 실시간 JVM에 적용됩니다. 참조용으로 공통 설정이 표시됩니다.

JVM 시작 시 환경 변수 또는 명령행 매개변수를 사용하여 기본 설정을 변경할 수 있습니다. 다음 표는 몇 가지 공통 JVM 설정을 보여줍니다. 마지막 열은 다음 키가 적용될 경우 동작을 어떻게 변경되는지 알려줍니다.

- **e** - 환경 변수로만 제어하는 설정
- **c** - 명령행 매개변수로만 제어하는 설정
- **ec** - 환경 변수와 명령행 매개변수 둘 다로 제어하는 설정으로, 명령행 매개변수가 우선 적용됩니다.

이 정보는 빠른 참조로 제공되므로, 모든 내용을 포함하고 있지는 않습니다.

JVM 설정	기본	설정에 영향을 주는 주체
Java 덤프	사용	ec
메모리 부족 시 Java 덤프	사용	ec
힙 덤프	사용 안함	ec
메모리 부족 시 힙 덤프	사용	ec
시스템 덤프	사용	ec
덤프 파일을 작성하는 장소	Current 디렉토리	ec
상세 출력	사용 안함	c
부트 클래스 경로 검색	사용 안함	c
JNI 검사	사용 안함	c
원격 디버깅	사용 안함	c
엄격한 적합성 검사	사용 안함	c
빠른 시작	사용 안함	c
원격 디버그 정보 서버	사용 안함	c
신호 감소	사용 안함	c
신호 처리기 체인	사용	c
클래스 경로	설정 없음	ec
클래스 데이터 공유	사용 안함	c
내게 필요한 옵션 지원	사용	e
JIT 컴파일러	사용	ec
AOT 컴파일러(공유 클래스가 함께 사용 가능해야 JVM에서 AOT를 사용합니다.)	사용	c
JIT 디버그 옵션	사용 안함	c
알고리즘 볼드체의 Java2D 최대 사이즈 글꼴	14 포인트	e
Java2D가 확장 가능한 글꼴에서 렌더링된 비트맵 사용	사용	e
Java2D 프리타입 글꼴 래스터라이징	사용	e
Java2D가 AWT 글꼴 사용	사용 안함	e
기본 로케일	없음	e
플러그인 시작 전 대기 시간	0	e
임시 디렉토리	/tmp	e
플러그인 방향 재지정	없음	e
IM 교환	사용 안함	e
IM 수정자	사용 안함	e
스레드 모델	N/A	e
Java 스레드 32비트에 대한 *초기 스택 크기*. **-Xiss<size>** 사용	2KB	c
Java 스레드 32비트에 대한 *최대 스택 크기*. **-Xss<size>** 사용	256KB	c
OS 스레드 32비트에 대한 *스택 크기*. **-Xmso<size>** 사용	256KB	c
초기 힙 크기. **-Xms<size>** 사용	64MB	c

JVM 설정	기본	설정에 영향을 주는 주체
최대 Java 힙 크기. **-Xmx<size>** 사용	최소 16MB와 최대 512MB의 사용 가능 메모리 절반	c
애플리케이션의 대상 시간 간격 이용률. 가비지 콜렉터가 남은 항목 사용 시도. **-Xgc:targetUtilization=<percentage>** 사용	70%	c
실행할 가비지 콜렉터 스레드의 수. **-Xgc:threads=<value>** 사용	1	c
-Xrealtime 모드에서 범위 메모리에 할당할 수 있는 최대 메모리 크기. **-Xgc:scopedMemoryMaximumSize=<size>** 사용	8MB	c
-Xrealtime 모드에서 영구 메모리 영역의 크기 설정. **-Xgc:immortalMemorySize=<size>** 사용	16MB	c

주: 『사용 가능 메모리』는 실제(물리적) 메모리이거나 **RLIMIT_AS** 값이며 가장 작은 값입니다.

WebSphere Real Time for RT Linux 클래스 라이브러리

WebSphere Real Time for RT Linux에 사용되는 Java 클래스 라이브러리에 대한 참조입니다.

WebSphere Real Time for RT Linux에 사용되는 Java 클래스 라이브러리가 http://www.rtsj.org/specjavadoc/book_index.html에 설명되어 있습니다.

TCK로 실행

Real-Time Specification for Java(RTSJ) Technology Compatibility Kit(TCK)를 WebSphere Real Time for RT Linux와 함께 실행할 경우, 테스트가 완료되도록 클래스 경로에 `demo/realtime/TCKibm.jar`을 포함해야 합니다.

`TCKibm.jar`은 `TCK.ProcessorLock` 클래스에 대한 IBM 확장인 **VibmcorProcessorLock** 클래스를 포함하고 있습니다. 이 클래스는 소규모 TCK 테스트에 필요한 유니프로세서 동작을 제공합니다. TCK.ProcessorLock 클래스 및 이 클래스의 공급업체별 확장에 대한 자세한 정보는 TCK 배포판에 포함된 readme 파일을 참조하십시오.

제 11 장 주의사항

이 정보는 미국에서 제공되는 제품 및 서비스용으로 작성된 것입니다. IBM은 다른 국가에서 이 문서에 기술된 제품, 서비스 또는 기능을 제공하지 않을 수도 있습니다. 현재 사용할 수 있는 제품 및 서비스에 대한 정보는 한국 IBM 담당자에게 문의하십시오. 여기에서 IBM 제품, 프로그램 또는 서비스를 언급하는 것이 해당 IBM 제품, 프로그램 또는 서비스만을 사용할 수 있다는 것을 의미하지는 않습니다. IBM의 지적 재산권을 침해하지 않는 한, 기능상으로 동등한 제품, 프로그램 또는 서비스를 대신 사용할 수도 있습니다. 그러나 비IBM 제품, 프로그램 또는 서비스의 운용에 대한 평가 및 검증은 사용자의 책임입니다.

IBM은 이 책에서 다루고 있는 특정 내용에 대해 특허를 보유하고 있거나 현재 특허 출원 중일 수 있습니다. 이 책을 제공한다고 해서 특허에 대한 라이센스까지 부여하는 것은 아닙니다. 라이센스에 대한 의문사항은 다음으로 문의하십시오.

135-700
서울특별시 강남구 도곡동 467-12
군인공제회관빌딩
한국 아이.비.엠 주식회사
고객만족센터
전화번호: 080-023-8080

2바이트(DBCS) 정보에 관한 라이센스 문의는 한국 IBM 고객만족센터에 문의하거나 다음 주소로 서면 문의하시기 바랍니다.

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

다음 단락은 현지법과 상충하는 영국이나 기타 국가에서는 적용되지 않습니다.

IBM은 타인의 권리 비침해, 상품성 및 특정 목적에의 적합성에 대한 묵시적 보증을 포함하여(단, 이에 한하지 않음) 묵시적이든 명시적이든 어떠한 종류의 보증 없이 이 책을 "현상태대로" 제공합니다. 일부 국가에서는 특정 거래에서 명시적 또는 묵시적 보증의 면책사항을 허용하지 않으므로, 이 사항이 적용되지 않을 수도 있습니다.

이 정보에는 기술적으로 부정확한 내용이나 인쇄상의 오류가 있을 수 있습니다. 이 정보는 주기적으로 변경되며, 변경된 사항은 최신판에 통합됩니다. IBM은 이 책에서 설명한 제품 및/또는 프로그램을 사전 통지 없이 언제든지 개선 및/또는 변경할 수 있습니다.

이 정보에서 언급되는 비IBM의 웹 사이트는 단지 편의상 제공된 것으로, 어떤 방식으로든 이들 웹 사이트를 옹호하고자 하는 것은 아닙니다. 해당 웹 사이트의 자료는 본 IBM 제품 자료의 일부가 아니므로 해당 웹 사이트 사용으로 인한 위험은 사용자 본인이 감수해야 합니다.

IBM은 귀하의 권리를 침해하지 않는 범위 내에서 적절하다고 생각하는 방식으로 귀하가 제공한 정보를 사용하거나 배포할 수 있습니다.

(i) 독립적으로 작성된 프로그램과 기타 프로그램(본 프로그램 포함)간의 정보 교환 및 (ii) 교환된 정보의 상호 이용을 목적으로 정보를 원하는 프로그램 라이센스 사용자는 다음 주소로 문의하십시오.

- JIMMAIL@uk.ibm.com [한국 아이.비.엠 주식회사 고객만족센터]

이러한 정보는 해당 조건(예를 들어, 사용료 지불 등)에 따라 사용할 수 있습니다.

이 정보에 기술된 라이센스가 있는 프로그램 및 이 프로그램에 대해 사용 가능한 모든 라이센스가 있는 자료는 IBM이 IBM 기본 계약, IBM 프로그램 라이센스 계약(IPLA) 또는 이와 동등한 계약에 따라 제공한 것입니다.

본 문서에 포함된 모든 성능 데이터는 제한된 환경에서 산출된 것입니다. 따라서 다른 운영 환경에서 얻어진 결과는 상당히 다를 수 있습니다. 일부 성능은 개발 레벨 상태의 시스템에서 측정되었을 수 있으므로 이러한 측정치가 일반적으로 사용되고 있는 시스템에서도 동일하게 나타날 것이라고는 보증할 수 없습니다. 또한 일부 성능은 추정을 통해 추측되었을 수도 있으므로 실제 결과는 다를 수 있습니다. 이 책의 사용자는 해당 데이터를 사용자의 특정 환경에서 검증해야 합니다.

비IBM 제품에 관한 정보는 해당 제품의 공급업체, 공개 자료 또는 다른 기타 범용 소스로부터 얻은 것입니다. IBM에서는 이러한 비IBM 제품을 테스트하지 않았으므로, 이들 제품과 관련된 성능의 정확성, 호환성 또는 기타 주장에 대해서는 확신할 수 없습니다. 비IBM 제품의 성능에 대한 의문사항은 해당 제품의 공급업체에 문의하십시오.

상표

IBM, IBM 로고 및 ibm.com은 미국 또는 기타 국가에서 사용되는 International Business Machines Corporation의 상표 또는 등록상표입니다. 이와 함께 기타 IBM 상표가 기재된 용어가 상표 기호(® 또는 ™)와 함께 이 정보에 처음 표시된 경우, 이와 같은 기호는 이 정보를 발행할 때 미국에서 IBM이 소유한 등록상표 또는 일반 법적

상표입니다. 또한 이러한 상표는 기타 국가에서 등록상표 또는 일반 법적 상표입니다. 현재 IBM 상표 목록은 웹 "저작권 및 상표 정보"(http://www.ibm.com/legal/copytrade.shtml)에 있습니다.

Adobe, Adobe 로고, PostScript 및 PostScript 로고는 미국 또는 기타 국가에서 사용되는 Adobe Systems Incorporated의 상표 또는 등록상표입니다.

Intel 및 Itanium은 미국 및 기타 국가에서 사용되는 Intel Corporation 또는 그 자회사의 상표입니다.

Linux는 미국 또는 기타 국가에서 사용되는 Linus Torvalds의 상표입니다.

Java 및 모든 Java 기반 상표는 Oracle 및/또는 계열사의 상표 또는 등록상표입니다.

기타 회사, 제품 또는 서비스 이름은 타사의 상표 또는 서비스표입니다.

주의사항

이 정보는 미국에서 제공되는 제품 및 서비스용으로 작성된 것입니다. IBM은 다른 국가에서 이 문서에 기술된 제품, 서비스 또는 기능을 제공하지 않을 수도 있습니다. 현재 사용할 수 있는 제품 및 서비스에 대한 정보는 한국 IBM 담당자에게 문의하십시오. 여기에서 IBM 제품, 프로그램 또는 서비스를 언급하는 것이 해당 IBM 제품, 프로그램 또는 서비스만을 사용할 수 있다는 것을 의미하지는 않습니다. IBM의 지적 재산권을 침해하지 않는 한, 기능상으로 동등한 제품, 프로그램 또는 서비스를 대신 사용할 수도 있습니다. 그러나 비IBM 제품, 프로그램 또는 서비스의 운용에 대한 평가 및 검증은 사용자의 책임입니다.

IBM은 이 책에서 다루고 있는 특정 내용에 대해 특허를 보유하고 있거나 현재 특허 출원 중일 수 있습니다. 이 책을 제공한다고 해서 특허에 대한 라이센스까지 부여하는 것은 아닙니다. 라이센스에 대한 의문사항은 다음으로 문의하십시오.

135-700
서울특별시 강남구 도곡동 467-12
군인공제회관빌딩
한국 아이.비.엠 주식회사
고객만족센터
전화번호: 080-023-8080

2바이트(DBCS) 정보에 관한 라이센스 문의는 한국 IBM 고객만족센터에 문의하거나 다음 주소로 서면 문의하시기 바랍니다.

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

다음 단락은 현지법과 상충하는 영국이나 기타 국가에서는 적용되지 않습니다.

IBM은 타인의 권리 비침해, 상품성 및 특정 목적에의 적합성에 대한 묵시적 보증을 포함하여(단, 이에 한하지 않음) 묵시적이든 명시적이든 어떠한 종류의 보증 없이 이 책을 "현상태대로" 제공합니다. 일부 국가에서는 특정 거래에서 명시적 또는 묵시적 보증의 면책사항을 허용하지 않으므로, 이 사항이 적용되지 않을 수도 있습니다.

이 정보에는 기술적으로 부정확한 내용이나 인쇄상의 오류가 있을 수 있습니다. 이 정보는 주기적으로 변경되며, 변경된 사항은 최신판에 통합됩니다. IBM은 이 책에서 설명한 제품 및/또는 프로그램을 사전 통지 없이 언제든지 개선 및/또는 변경할 수 있습니다.

이 정보에서 언급되는 비IBM의 웹 사이트는 단지 편의상 제공된 것으로, 어떤 방식으로든 이들 웹 사이트를 옹호하고자 하는 것은 아닙니다. 해당 웹 사이트의 자료는 본 IBM 제품 자료의 일부가 아니므로 해당 웹 사이트 사용으로 인한 위험은 사용자 본인이 감수해야 합니다.

IBM은 귀하의 권리를 침해하지 않는 범위 내에서 적절하다고 생각하는 방식으로 귀하가 제공한 정보를 사용하거나 배포할 수 있습니다.

(i) 독립적으로 작성된 프로그램과 기타 프로그램(본 프로그램 포함)간의 정보 교환 및 (ii) 교환된 정보의 상호 이용을 목적으로 정보를 원하는 프로그램 라이센스 사용자는 다음 주소로 문의하십시오.

- JIMMAIL@uk.ibm.com [한국 아이.비.엠 주식회사 고객만족센터]

이러한 정보는 해당 조건(예를 들어, 사용료 지불 등)에 따라 사용할 수 있습니다.

이 정보에 기술된 라이센스가 있는 프로그램 및 이 프로그램에 대해 사용 가능한 모든 라이센스가 있는 자료는 IBM이 IBM 기본 계약, IBM 프로그램 라이센스 계약(IPLA) 또는 이와 동등한 계약에 따라 제공한 것입니다.

본 문서에 포함된 모든 성능 데이터는 제한된 환경에서 산출된 것입니다. 따라서 다른 운영 환경에서 얻어진 결과는 상당히 다를 수 있습니다. 일부 성능은 개발 레벨 상태의 시스템에서 측정되었을 수 있으므로 이러한 측정치가 일반적으로 사용되고 있는 시스템에서도 동일하게 나타날 것이라고는 보증할 수 없습니다. 또한 일부 성능은 추정을 통해 추측되었을 수도 있으므로 실제 결과는 다를 수 있습니다. 이 책의 사용자는 해당 데이터를 사용자의 특정 환경에서 검증해야 합니다.

비IBM 제품에 관한 정보는 해당 제품의 공급업체, 공개 자료 또는 다른 기타 범용 소스로부터 얻은 것입니다. IBM에서는 이러한 비IBM 제품을 테스트하지 않았으므로, 이들 제품과 관련된 성능의 정확성, 호환성 또는 기타 주장에 대해서는 확신할 수 없습니다. 비IBM 제품의 성능에 대한 의문사항은 해당 제품의 공급업체에 문의하십시오.

상표

IBM, IBM 로고 및 ibm.com은 미국 또는 기타 국가에서 사용되는 International Business Machines Corporation의 상표 또는 등록상표입니다. 이와 함께 기타 IBM 상표가 기재된 용어가 상표 기호(® 또는 ™)와 함께 이 정보에 처음 표시된 경우, 이와 같은 기호는 이 정보를 발행할 때 미국에서 IBM이 소유한 등록상표 또는 일반 법적

상표입니다. 또한 이러한 상표는 기타 국가에서 등록상표 또는 일반 법적 상표입니다. 현재 IBM 상표 목록은 웹 "저작권 및 상표 정보"(http://www.ibm.com/legal/copytrade.shtml)에 있습니다.

Adobe, Adobe 로고, PostScript 및 PostScript 로고는 미국 또는 기타 국가에서 사용되는 Adobe Systems Incorporated의 상표 또는 등록상표입니다.

Intel 및 Itanium은 미국 및 기타 국가에서 사용되는 Intel Corporation 또는 그 자회사의 상표입니다.

Linux는 미국 또는 기타 국가에서 사용되는 Linus Torvalds의 상표입니다.

Java 및 모든 Java 기반 상표는 Oracle 및/또는 계열사의 상표 또는 등록상표입니다.

기타 회사, 제품 또는 서비스 이름은 타사의 상표 또는 서비스표입니다.

색인

[자]

[차]

[카]

[타]

[파]

[하]

A

C

D

H

I

J

L

N

O

P

R

S

T

[특수 문자]

Printed in Korea