ej technologies

JProfiler에 대한 결정적인 가이드

성능 전문가로서 알아야 할 모든 것

Index

JProfiler 소개

JProfiler란 무엇인가요?

JProfiler는 실행 중인 JVM 내부에서 무슨 일이 일어나고 있는지를 분석하기 위한 전문 도구입니다. 개발, 품질 보증, 그리고 운영 시스템에 문제가 발생했을 때의 긴급 대응에 사용할 수 있습니다.

JProfiler가 다루는 네 가지 주요 주제가 있습니다:

- **메서드 호출**

  이는 일반적으로 "CPU 프로파일링"이라고 불립니다. 메서드 호출은 다양한 방식으로 측정되고 시각화될 수 있습니다. 메서드 호출의 분석은 애플리케이션이 무엇을 하고 있는지 이해하고 성능을 개선할 방법을 찾는 데 도움을 줍니다.

- **할당**

  힙에 있는 객체를 할당, 참조 체인 및 가비지 컬렉션과 관련하여 분석하는 것은 "메모리 프로파일링" 범주에 속합니다. 이 기능은 메모리 누수를 수정하고, 일반적으로 메모리 사용을 줄이며, 임시 객체의 할당을 줄이는 데 도움을 줍니다.

- **스레드 및 잠금**

  스레드는 예를 들어 객체에 대해 동기화하여 잠금을 보유할 수 있습니다. 여러 스레드가 협력할 때 교착 상태가 발생할 수 있으며, JProfiler는 이를 시각화할 수 있습니다. 또한, 잠금은 경쟁 상태가 될 수 있으며, 이는 스레드가 잠금을 획득하기 전에 대기해야 함을 의미합니다. JProfiler는 스레드와 다양한 잠금 상황에 대한 통찰력을 제공합니다.

- **상위 수준의 하위 시스템**

  많은 성능 문제는 더 높은 의미 수준에서 발생합니다. 예를 들어, JDBC 호출의 경우 가장 느린 SQL 문이 무엇인지 알고 싶을 것입니다. 그러한 하위 시스템에 대해 JProfiler는 호출 트리에 특정 페이로드를 attach하는 "프로브"를 제공합니다.

JProfiler의 UI는 데스크톱 애플리케이션으로 제공됩니다. 라이브 JVM을 인터랙티브하게 프로파일링하거나 UI를 사용하지 않고 자동으로 프로파일링할 수 있습니다. 프로파일링 데이터는 JProfiler UI로 열 수 있는 스냅샷에 저장됩니다. 또한, 명령줄 도구와 빌드 도구 통합은 프로파일링 세션을 자동화하는 데 도움을 줍니다.

어떻게 계속해야 하나요?

이 문서는 순차적으로 읽도록 의도되었으며, 이후의 도움말 주제는 이전 내용에 기반하여 작성되었습니다.

먼저, 아키텍처 [p. 5]에 대한 기술적 개요는 프로파일링이 어떻게 작동하는지 이해하는 데 도움을 줍니다.

JProfiler 설치 [p. 7] 및 JVM 프로파일링 [p. 11]에 대한 도움말 주제는 시작하는 데 도움을 줍니다.

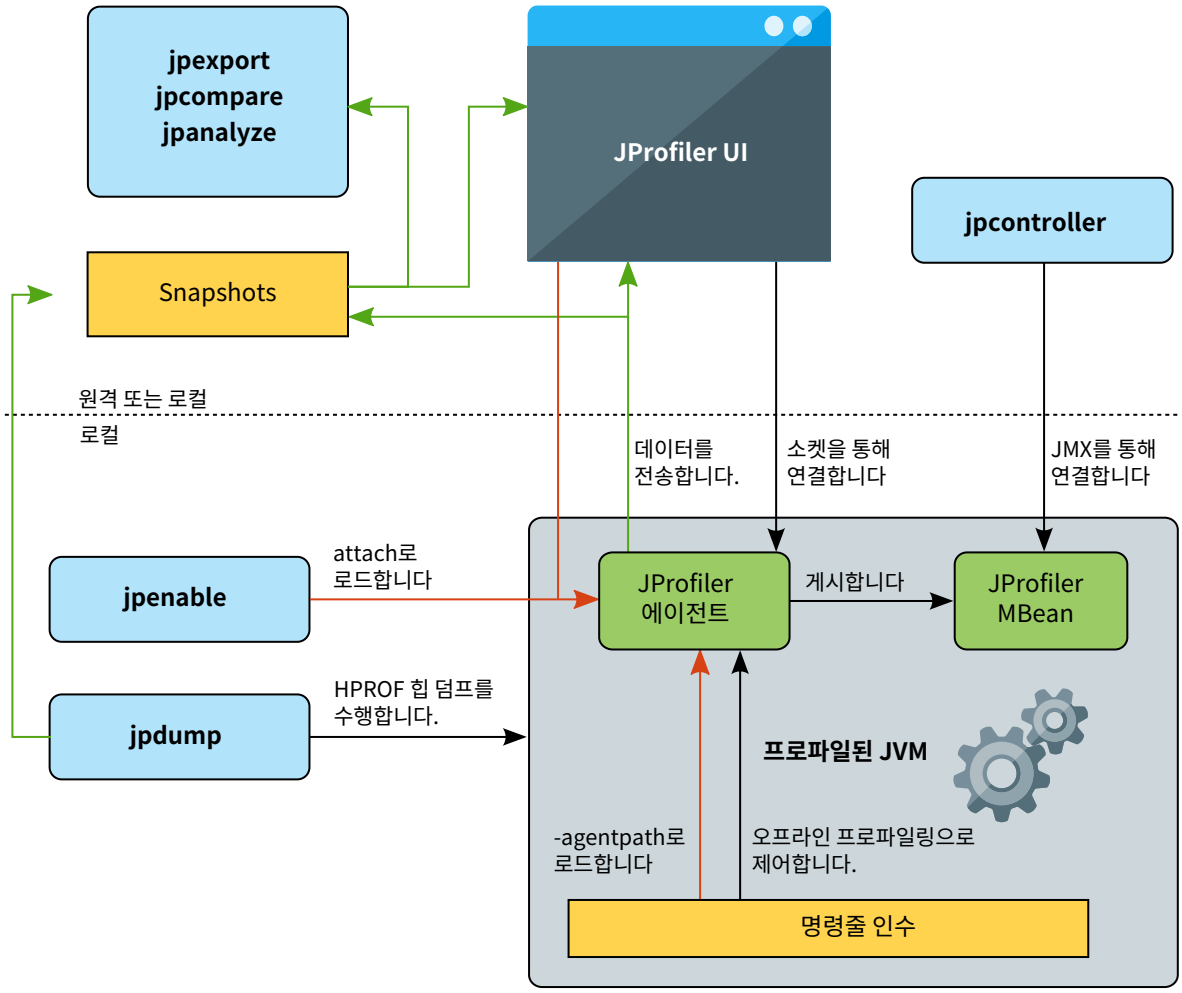
그 후, 데이터 녹화 [p. 26] 및 스냅샷 [p. 39]에 대한 논의는 JProfiler를 스스로 탐색할 수 있는 수준의 이해를 제공합니다.

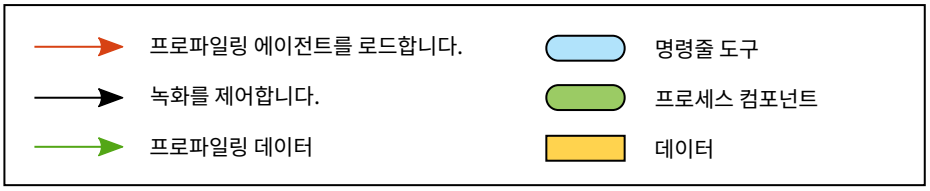
이후의 장에서는 JProfiler의 다양한 기능에 대한 전문 지식을 쌓을 수 있습니다. 마지막 섹션은 특정 기능이 필요할 때 참조해야 할 선택적 읽기 자료입니다.

여러분의 피드백을 환영합니다. 특정 영역에서 문서가 부족하다고 느끼거나 문서에서 부정확성을 발견한 경우, 주저하지 말고 support@ej-technologies.com으로 연락해 주세요.

JProfiler 아키텍처

프로파일된 애플리케이션, JProfiler UI 및 모든 명령줄 유틸리티와 관련된 중요한 상호작용의 전체 그림이 아래에 제공됩니다.





프로파일링 에이전트

"JVM tool interface" (JVMTI)는 프로파일러가 정보를 얻고 자체 계측을 삽입하기 위한 훅을 추가하기 위해 사용하는 네이티브 인터페이스입니다. 이는 프로파일링 에이전트의 적어도 일부가 네이티브 코드로 구현되어

야 하며, 따라서 JVM 프로파일러는 플랫폼 독립적이지 않다는 것을 의미합니다. JProfiler는 여기 [p. 7]에 나열된 다양한 플랫폼을 지원합니다.

JVM 프로파일러는 시작 시 또는 나중에 로드되는 네이티브 라이브러리로 구현됩니다. 시작 시 로드하려면 VM 매개변수 `-agentpath:<  >`를 명령줄에 추가합니다. 이 매개변수를 수동으로 추가할 필요는 거의 없습니다. JProfiler가 IDE 통합, 통합 마법사 또는 JVM을 직접 실행할 때 이를 추가하기 때문입니다. 그러나 이것이 프로파일링을 가능하게 하는 것임을 아는 것이 중요합니다.

JVM이 네이티브 라이브러리를 로드하는 데 성공하면, 라이브러리 내의 특별한 함수를 호출하여 프로파일링 에이전트가 초기화할 기회를 제공합니다. JProfiler는 `JProfiler>`로 시작하는 몇 가지 진단 메시지를 출력하여 프로파일링이 활성화되었음을 알립니다. 요점은 `-agentpath` VM 매개변수를 전달하면 프로파일링 에이전트가 성공적으로 로드되거나 JVM이 시작되지 않는다는 것입니다.

로드되면 프로파일링 에이전트는 JVMTI에 스레드 생성이나 클래스 로딩과 같은 모든 종류의 이벤트에 대해 알림을 요청합니다. 이러한 이벤트 중 일부는 직접 프로파일링 데이터를 제공합니다. 클래스 로딩 이벤트를 사용하여 프로파일링 에이전트는 로드된 클래스에 계측을 삽입하고 자체 바이트코드를 삽입하여 측정을 수행합니다.

JProfiler는 이미 실행 중인 JVM에 에이전트를 로드할 수 있으며, JProfiler UI를 사용하거나 `bin/jpenable` 명령줄 도구를 사용할 수 있습니다. 이 경우, 이미 로드된 상당수의 클래스가 필요한 계측을 적용하기 위해 다시 변환되어야 할 수 있습니다.

데이터 기록

JProfiler 에이전트는 프로파일링 데이터만 수집합니다. JProfiler UI는 별도로 시작되며 소켓을 통해 프로파일링 에이전트에 연결됩니다. 원격 서버에 대한 안전한 연결을 위해 JProfiler를 구성하여 SSH 터널을 자동으로 생성할 수 있습니다.

JProfiler UI에서 에이전트에게 데이터를 기록하도록 지시하고, UI에 프로파일링 데이터를 표시하며, 스냅샷을 디스크에 저장할 수 있습니다. UI의 대안으로, 프로파일링 에이전트는 MBean [(1)]을 통해 제어될 수 있습니다. 이 MBean을 사용하는 명령줄 도구는 `bin/jpcontroller`입니다.

프로파일링 에이전트를 제어하는 또 다른 방법은 미리 정의된 트리거 및 액션 세트를 사용하는 것입니다. 이렇게 하면 프로파일링 에이전트가 무인 모드로 작동할 수 있습니다. 이는 JProfiler에서 "오프라인 프로파일링"이라고 하며, 프로파일링 세션을 자동화하는 데 유용합니다.

스냅샷

JProfiler UI는 실시간 프로파일링 데이터를 표시할 수 있지만, 모든 기록된 프로파일링 데이터의 스냅샷을 저장해야 하는 경우가 종종 있습니다. 스냅샷은 JProfiler UI에서 수동으로 저장되거나 트리거 액션에 의해 자동으로 저장됩니다.

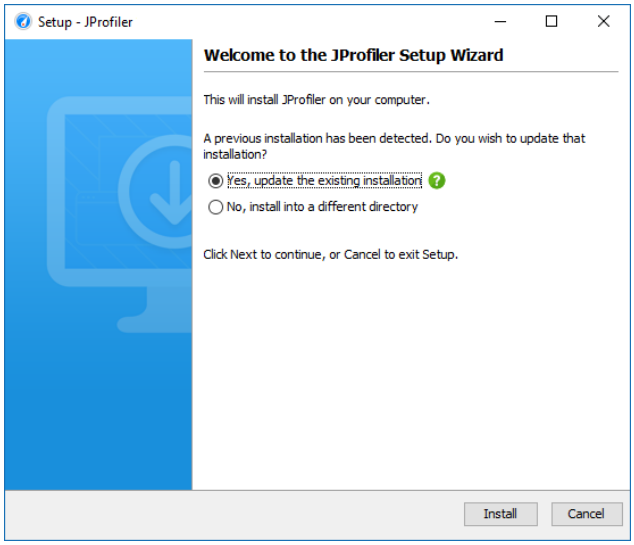
스냅샷은 JProfiler UI에서 열고 비교할 수 있습니다. 자동화된 처리를 위해 명령줄 도구 `bin/jpexport` 및 `bin/jpcompare`를 사용하여 이전에 저장된 스냅샷에서 데이터를 추출하고 HTML 보고서를 생성할 수 있습니다.

실행 중인 JVM에서 힙 스냅샷을 얻는 저비용 방법은 `bin/jpdump` 명령줄 도구를 사용하는 것입니다. 이는 JVM의 내장 기능을 사용하여 JProfiler에서 열 수 있는 HPROF 스냅샷을 저장하며, 프로파일링 에이전트를 로드할 필요가 없습니다.

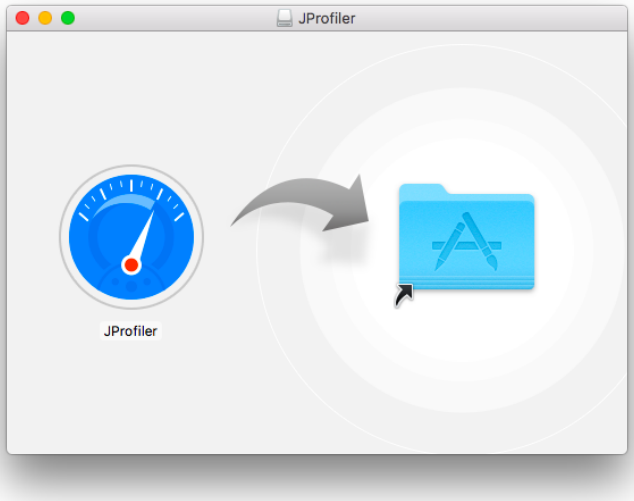
[(1)] https://en.wikipedia.org/wiki/Java_Management_Extensions

JProfiler 설치

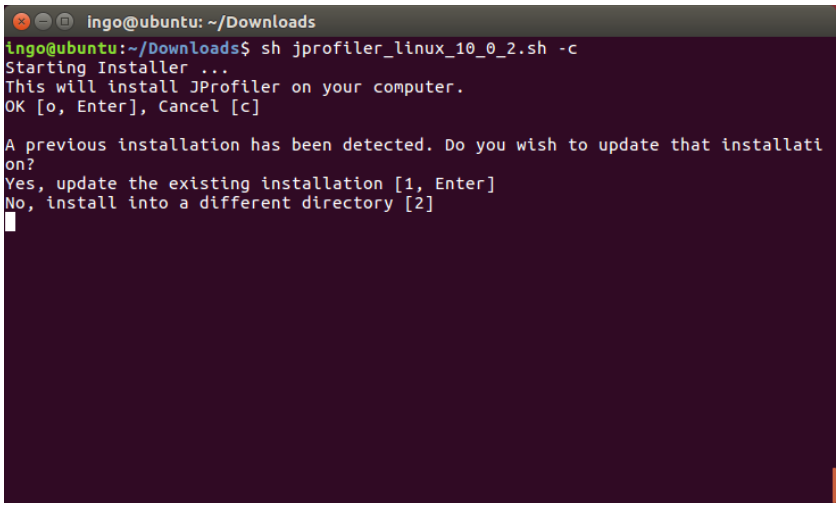
Windows 및 Linux/Unix용 실행 가능한 설치 프로그램이 제공되어 설치 과정을 단계별로 안내합니다. 이전 설치가 감지되면 설치가 간소화됩니다.



macOS에서는 JProfiler가 UI 애플리케이션에 대한 표준 설치 절차를 사용합니다: Finder에서 더블 클릭하여 마운트할 수 있는 DMG 아카이브, 그런 다음 JProfiler 애플리케이션 번들을 `/Applications` 폴더로 드래그할 수 있습니다. 해당 폴더는 DMG 자체에 심볼릭 링크로 표시됩니다.



Linux/Unix에서는 다운로드 후 설치 프로그램이 실행 가능하지 않으므로 실행 시 sh를 앞에 붙여야 합니다. 설치 프로그램은 `-c` 매개변수를 전달하면 명령줄 설치를 수행합니다. Windows 및 Linux/Unix에 대한 완전한 무인 설치는 `-q` 매개변수로 수행됩니다. 이 경우 설치 디렉토리를 선택하기 위해 추가 인수 `-dir <directory>`를 전달할 수 있습니다.



설치 프로그램을 실행한 후에는 전체 사용자 입력을 포함하는 `.install4j/response.varfile` 파일을 저장합니다. 해당 파일을 가져와 명령줄에서 `-varfile <path to response.varfile>` 인수를 전달하여 무인 설치를 자동화할 수 있습니다.

무인 설치에 대한 라이선스 정보를 설정하려면 `-Vjprofiler.licenseKey=<license key> -Vjprofiler.licenseName=<user name>` 및 선택적으로 `-Vjprofiler.licenseCompany=<company name>`을 명령줄 인수로 전달하십시오. 부동 라이선스가 있는 경우 라이선스 키 대신 `FLOAT:<server name or IP address>`를 사용하십시오.

아카이브는 Windows용 ZIP 파일 및 Linux용 .tar.gz 파일로도 제공됩니다. 명령어

```
tar xzvf filename.tar.gz
```

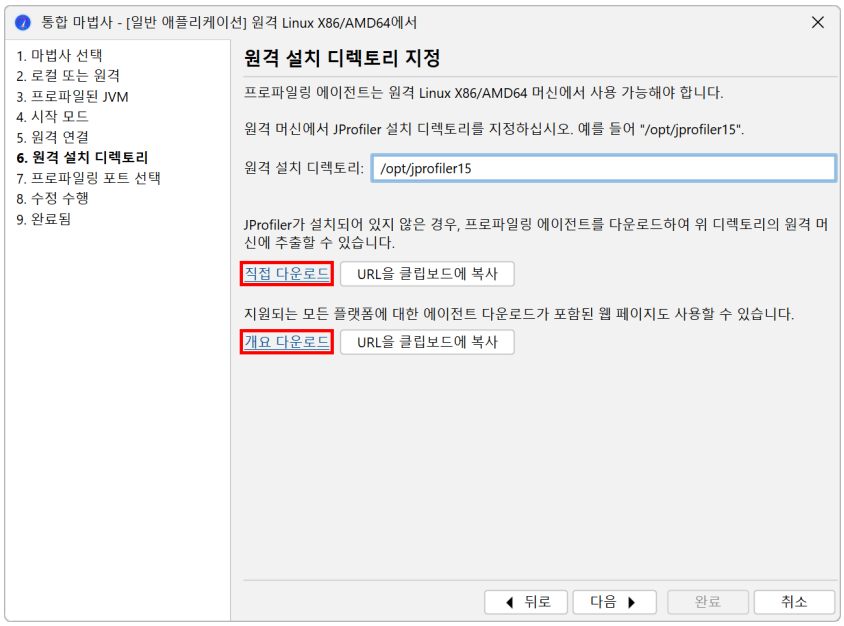
는 .tar.gz 아카이브를 별도의 최상위 디렉토리로 추출합니다. JProfiler를 시작하려면 추출된 디렉토리에서 `bin/jprofiler`를 실행하십시오. Linux/Unix에서는 파일 `jprofiler.desktop`을 사용하여 JProfiler 실행 파일을 윈도우 관리자에 통합할 수 있습니다. 예를 들어, Ubuntu에서는 런처 사이드바로 데스크톱 파일을 드래그하여 영구적인 런처 항목을 만들 수 있습니다.

원격 머신에 프로파일링 에이전트 배포

JProfiler는 두 부분으로 구성됩니다: 스냅샷을 조작하는 명령줄 유틸리티와 함께 데스크톱 UI가 한쪽에 있고, 프로파일된 JVM을 제어하는 명령줄 유틸리티와 함께 프로파일링 에이전트가 다른 쪽에 있습니다. 웹사이트에서 다운로드한 설치 프로그램과 아카이브에는 두 부분이 모두 포함되어 있습니다.

그러나 원격 프로파일링의 경우 원격 측에 프로파일링 에이전트만 설치하면 됩니다. 원격 머신에서 JProfiler 배포 아카이브를 단순히 추출할 수 있지만, 특히 배포를 자동화할 때 필요한 파일 수를 제한하고 싶을 수 있습니다. 또한, 프로파일링 에이전트는 자유롭게 재배포할 수 있으므로 애플리케이션과 함께 제공하거나 고객 머신에 설치하여 문제를 해결할 수 있습니다.

프로파일링 에이전트가 포함된 최소 패키지를 얻으려면 원격 통합 마법사가 적절한 에이전트 아카이브의 다운로드 링크와 지원되는 모든 플랫폼에 대한 에이전트 아카이브의 다운로드 페이지를 보여줍니다. JProfiler GUI에서 세션->통합 마법사->새 서버/원격 통합을 호출하고 "원격" 옵션을 선택한 다음 원격 설치 디렉토리 단계로 진행하십시오.



특정 JProfiler 버전에 대한 HTML 개요 페이지의 URL은 다음과 같습니다:

```
https://www.ej-technologies.com/jprofiler/agent?version=15.0
```

단일 에이전트 아카이브의 다운로드 URL 형식은 다음과 같습니다:

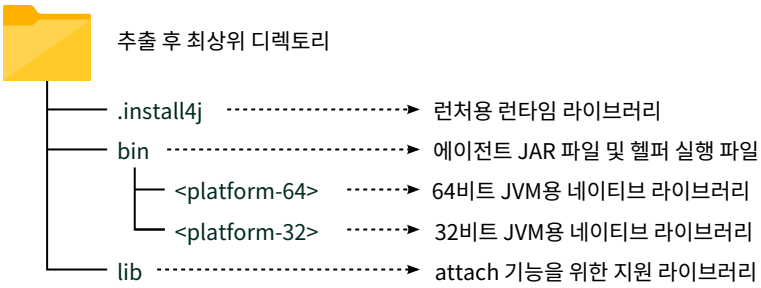
```
https://download.ej-technologies.com/jprofiler/jprofiler_agent_<platform>_15_0.<extension>
```

여기서 `platform`은 `bin` 디렉토리의 에이전트 디렉토리 이름에 해당하고, `extension`은 Windows에서는 `zip`, macOS에서는 `.tgz`, Linux/Unix에서는 `.tar.gz`입니다. Linux의 경우 x86과 x64가 함께 그룹화되므로 Linux x64의 URL은 다음과 같습니다:

```
https://download.ej-technologies.com/jprofiler/jprofiler_agent_linux-x86_15_0.tar.gz
```

에이전트 아카이브에는 필요한 네이티브 에이전트 라이브러리와 함께 `jpenable`, `jpdump` 및 `jpcontroller` 실행 파일이 포함되어 있습니다. 아카이브의 실행 파일과 프로파일링 에이전트는 최소 Java 8 버전만 필요합니다.

원격 머신에서 에이전트 아카이브를 추출한 후 볼 수 있는 하위 디렉토리는 아래에 설명되어 있습니다. 이는 해당 대상 플랫폼에서의 전체 JProfiler 설치의 하위 집합입니다.



지원되는 플랫폼

JProfiler는 JVM(JVMTI)의 네이티브 프로파일링 인터페이스를 활용하기 때문에 프로파일링 에이전트는 네이티브 라이브러리입니다.

JProfiler는 다음 플랫폼에서 프로파일링을 지원합니다:

OS	아키텍처	지원되는 JVM	버전
Windows 11/10 Windows Server 2025/2022/2019/2016	x86 x64/AMD64	Hotspot (OpenJDK) IBM/OpenJ9	1.8 - 24 1.8 - 24
macOS 10.12 - 15	Intel, Apple	Hotspot (OpenJDK) IBM/OpenJ9	1.8 - 24 1.8 - 24
Linux	x86 x64/AMD64	Hotspot (OpenJDK) IBM/OpenJ9	1.8 - 24 1.8 - 24
Linux	PPC64LE	Hotspot (OpenJDK) IBM/OpenJ9	1.8 - 24 1.8 - 24
Linux	ARMv7 ARMv8	Hotspot (OpenJDK)	1.8 - 24

JProfiler GUI 프론트엔드는 실행을 위해 Java 21 VM이 필요합니다. Windows, macOS 및 Linux x64용 JProfiler에는 이 목적을 위한 Java 21 JRE가 번들로 제공됩니다. attach 명령줄 도구 jpenable, jdump 및 jpcontroller는 Java 8 VM만 필요합니다.

JVM 프로파일링

JVM을 프로파일링하려면 JProfiler의 프로파일링 에이전트를 JVM에 로드해야 합니다. 이는 두 가지 방법으로 가능합니다: 시작 스크립트에서 `-agentpath` VM 매개변수를 지정하거나 attach API를 사용하여 이미 실행 중인 JVM에 에이전트를 로드하는 방법입니다.

JProfiler는 두 가지 모드를 모두 지원합니다. VM 매개변수를 추가하는 것이 프로파일링의 선호되는 방법이며, 통합 마법사, IDE 플러그인 및 JProfiler 내에서 JVM을 시작하는 세션 구성에서 사용됩니다. attach는 로컬 및 SSH를 통한 원격에서도 작동합니다.

-agentpath VM 매개변수

프로파일링 에이전트를 로드하는 VM 매개변수가 어떻게 구성되는지 이해하는 것이 유용합니다. `-agentpath`는 JVMTI 인터페이스를 사용하는 모든 종류의 네이티브 라이브러리를 로드하기 위해 JVM에서 제공하는 일반적인 VM 매개변수입니다. 프로파일링 인터페이스 JVMTI는 네이티브 인터페이스이기 때문에 프로파일링 에이전트는 네이티브 라이브러리여야 합니다. 이는 명시적으로 지원되는 플랫폼 [(1)]에서만 프로파일링할 수 있음을 의미합니다. 32비트 및 64비트 JVM도 다른 네이티브 라이브러리가 필요합니다. 반면, Java 에이전트는 `-javaagent` VM 매개변수로 로드되며 제한된 기능 세트에만 액세스할 수 있습니다.

`-agentpath:` 뒤에는 네이티브 라이브러리의 전체 경로 이름이 추가됩니다. 플랫폼별 라이브러리 이름만 지정하는 `-agentlib:`라는 동등한 매개변수가 있지만, 이 경우 라이브러리가 라이브러리 경로에 포함되어 있는지 확인해야 합니다. 라이브러리 경로 뒤에는 등호를 추가하고 옵션을 에이전트에 전달할 수 있으며, 쉼표로 구분됩니다. 예를 들어, Linux에서는 전체 매개변수가 다음과 같이 보일 수 있습니다:

```
-agentpath:/opt/jprofiler15/bin/linux-x64/libjprofilerti.so=port=8849,nowait
```

첫 번째 등호는 경로 이름과 매개변수를 구분하고, 두 번째 등호는 매개변수 `port=8849`의 일부입니다. 이 일반적인 매개변수는 프로파일링 에이전트가 JProfiler GUI로부터 연결을 수신하는 포트를 정의합니다. 8849는 실제로 기본 포트이므로 해당 매개변수를 생략할 수도 있습니다. 동일한 머신에서 여러 JVM을 프로파일링하려면 다른 포트를 할당해야 합니다. IDE 플러그인과 로컬로 시작된 세션은 이 포트를 자동으로 할당하며, 통합 마법사에서는 포트를 명시적으로 선택해야 합니다.

두 번째 매개변수 `nowait`는 프로파일링 에이전트가 시작 시 JVM을 차단하지 않고 JProfiler GUI가 연결되기를 기다리지 않도록 지시합니다. 시작 시 차단은 기본값입니다. 왜냐하면 프로파일링 에이전트는 명령줄 매개변수로 프로파일링 설정을 받지 않고 JProfiler GUI 또는 대안적으로 구성 파일에서 받기 때문입니다. 명령줄 매개변수는 프로파일링 에이전트를 부트스트랩하는 데만 사용되며, 시작 방법과 디버그 플래그를 전달하는 데 사용됩니다.

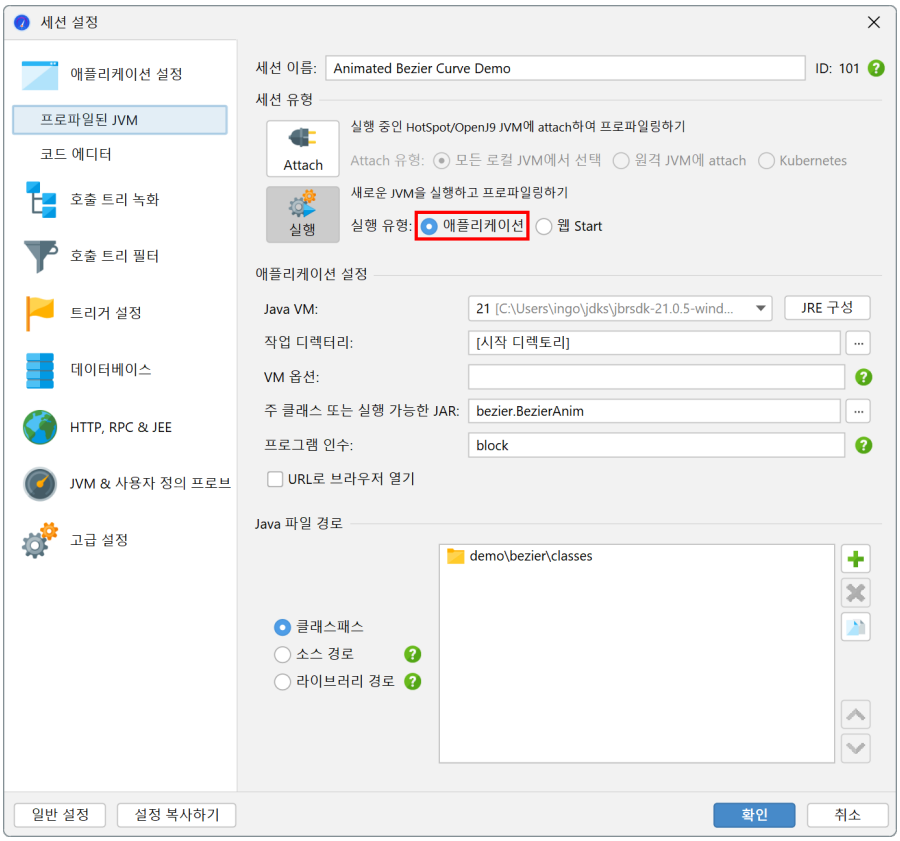
특정 상황에서는 시작 시 프로파일링 설정을 설정 [p. 235]해야 하며, 이를 달성하기 위해 수동 작업이 필요할 수 있습니다.

기본적으로 JProfiler 에이전트는 통신 소켓을 루프백 인터페이스에 바인딩합니다. 특정 인터페이스를 선택하려면 `address=[IP ]` 옵션을 추가하거나 `address=0.0.0.0`을 사용하여 통신 소켓을 모든 사용 가능한 네트워크 인터페이스에 바인딩할 수 있습니다. 이는 도커 컨테이너에서 프로파일링 포트를 게시하려는 경우 필요할 수 있습니다.

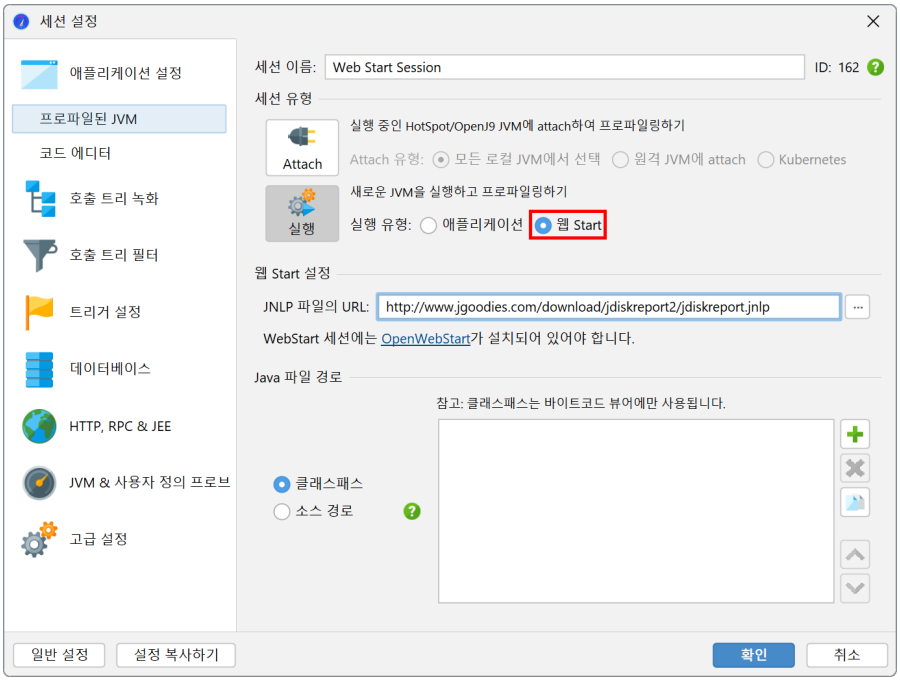
로컬로 시작된 세션

IDE의 "실행 구성"처럼, JProfiler에서 로컬로 시작된 세션을 직접 구성할 수 있습니다. 클래스패스, 메인 클래스, 작업 디렉토리, VM 매개변수 및 인수를 지정하면 JProfiler가 세션을 시작합니다. JProfiler와 함께 제공되는 모든 데모 세션은 로컬로 시작된 세션입니다.

[(1)] https://www.ej-technologies.com/products/jprofiler/featuresPlatforms.html

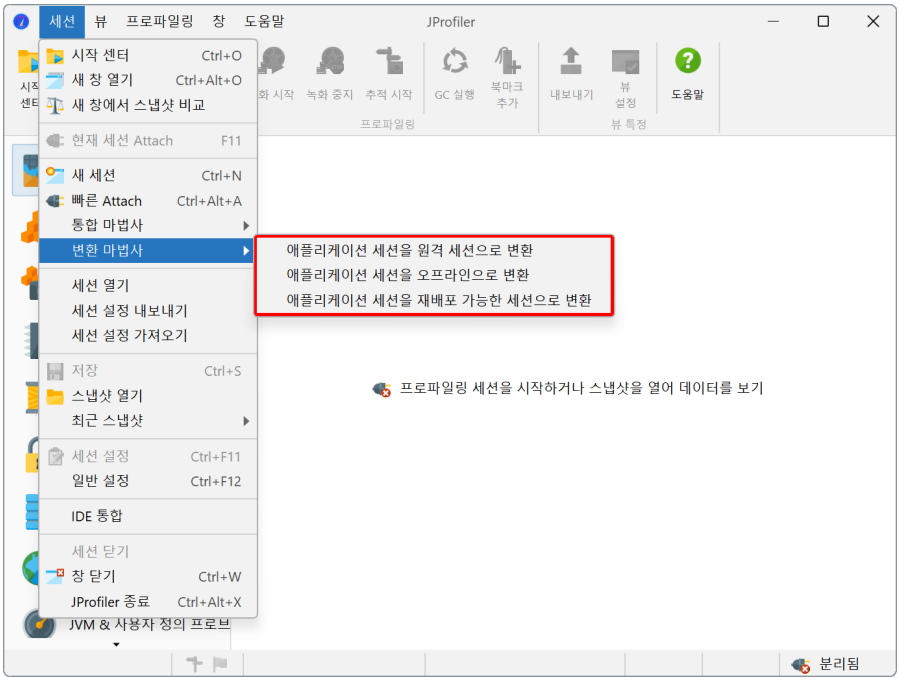


특별한 시작 모드는 "Web Start"로, JNLP 파일의 URL을 선택하면 JProfiler가 이를 프로파일링하기 위해 JVM을 시작합니다. 이 기능은 OpenWebStart [2]를 지원하며, Java 9 이전 Oracle JRE의 레거시 WebStart는 지원되지 않습니다.



[2] https://openwebstart.com/

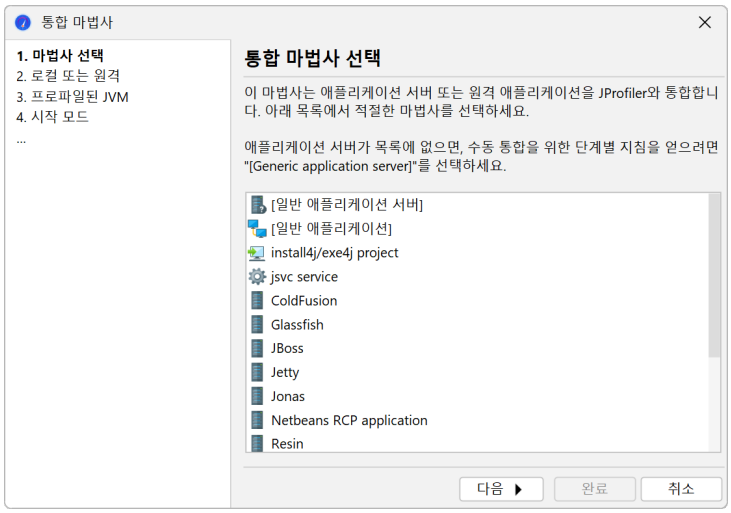
로컬로 시작된 세션은 변환 마법사를 통해 독립 실행형 세션으로 변환할 수 있으며, 주 메뉴에서 세션->변환 마법사를 호출하여 수행할 수 있습니다. 애플리케이션 세션을 원격으로 변환은 시작 스크립트를 생성하고 `-agentpath` VM 매개변수를 Java 호출에 삽입합니다. 애플리케이션 세션을 오프라인으로 변환은 오프라인 프로파일링 [p. 122]을 위한 시작 스크립트를 생성하며, 이는 시작 시 구성이 로드되고 JProfiler GUI가 필요하지 않음을 의미합니다. 애플리케이션 세션을 재배포 세션으로 변환은 동일한 작업을 수행하지만, 프로파일링 에이전트와 구성 파일이 포함된 디렉토리 `jprofiler_redist`를 생성하여 JProfiler가 설치되지 않은 다른 머신으로 전송할 수 있습니다.



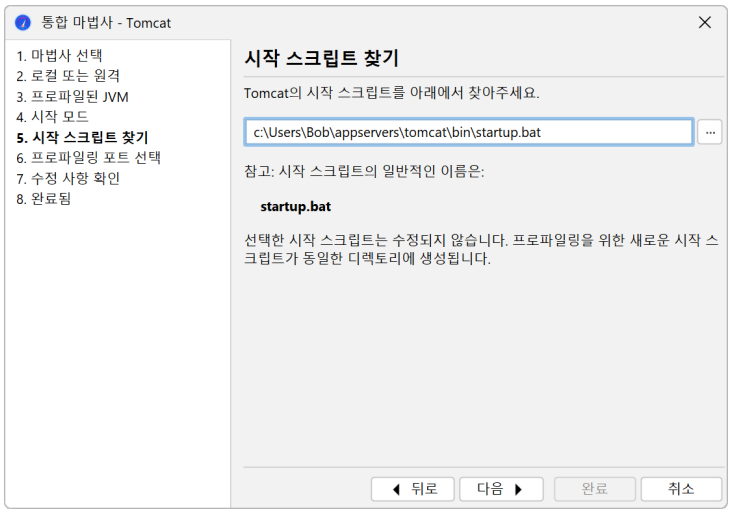
프로파일링된 애플리케이션을 직접 개발하는 경우, 시작된 세션 대신 IDE 통합 [p. 133]을 사용하는 것을 고려하십시오. 이는 더 편리하고 더 나은 소스 코드 탐색을 제공합니다. 애플리케이션을 직접 개발하지 않지만 시작 스크립트가 이미 있는 경우, 원격 통합 마법사를 사용하는 것을 고려하십시오. 이는 Java 호출에 추가해야 하는 정확한 VM 매개변수를 알려줍니다.

통합 마법사

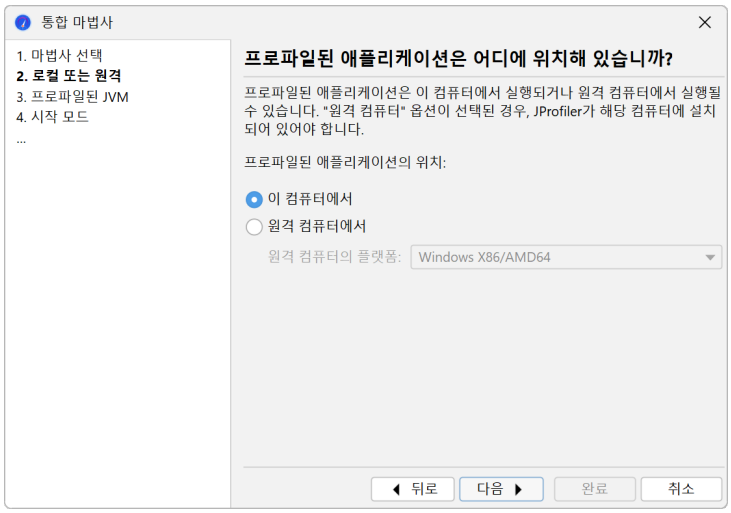
JProfiler의 통합 마법사는 시작 스크립트나 구성 파일을 프로그래밍적으로 수정하여 추가 VM 매개변수를 포함할 수 있는 잘 알려진 서드파티 컨테이너를 처리합니다. 일부 제품의 경우, VM 매개변수가 인수로 전달되거나 환경 변수로 전달되는 시작 스크립트를 생성할 수 있습니다.



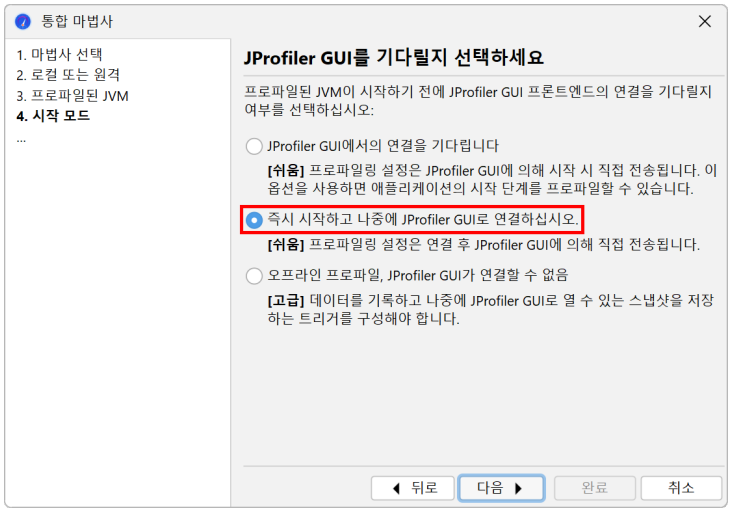
모든 경우에, JProfiler가 수정 작업을 수행할 수 있도록 서드파티 제품의 특정 파일을 찾아야 합니다. 일부 일반 마법사는 프로파일링을 활성화하기 위해 수행해야 할 작업에 대한 지침만 제공합니다.



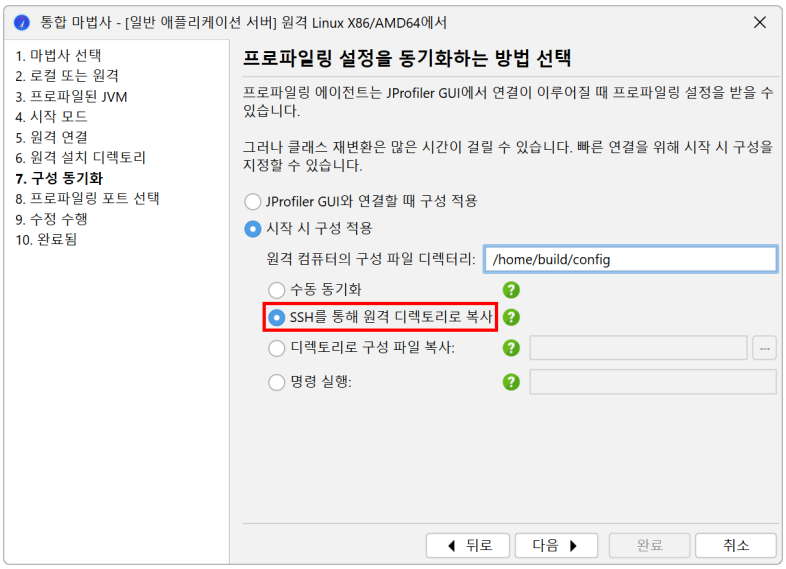
각 통합 마법사의 첫 번째 단계는 로컬 머신에서 프로파일링할지 원격 머신에서 프로파일링할지를 선택하는 것입니다. 로컬 머신의 경우, JProfiler가 이미 플랫폼, JProfiler가 설치된 위치 및 구성 파일의 위치를 알고 있기 때문에 제공해야 할 정보가 적습니다.



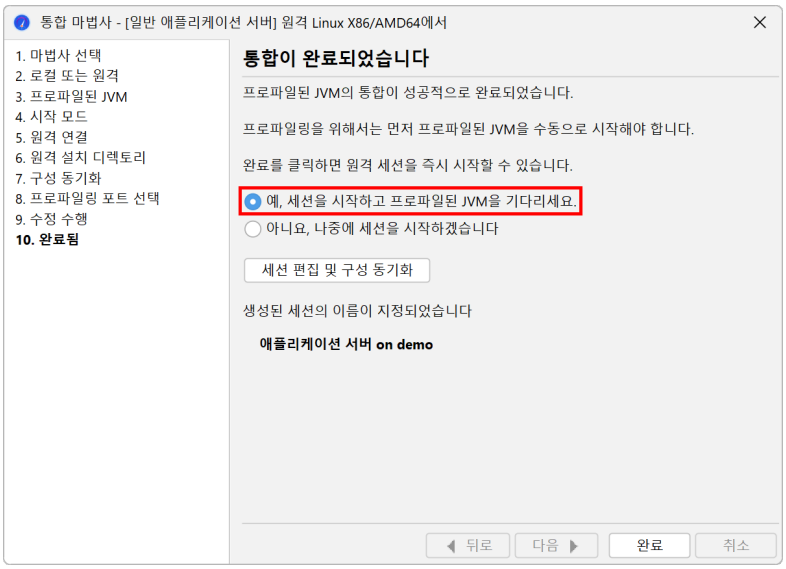
중요한 결정은 위에서 논의된 "시작 모드"입니다. 기본적으로 프로파일링 설정은 시작 시 JProfiler UI에서 전송되지만, JVM이 즉시 시작되도록 프로파일링 에이전트에 지시할 수도 있습니다. 후자의 경우, JProfiler GUI가 연결되면 프로파일링 설정을 적용할 수 있습니다.



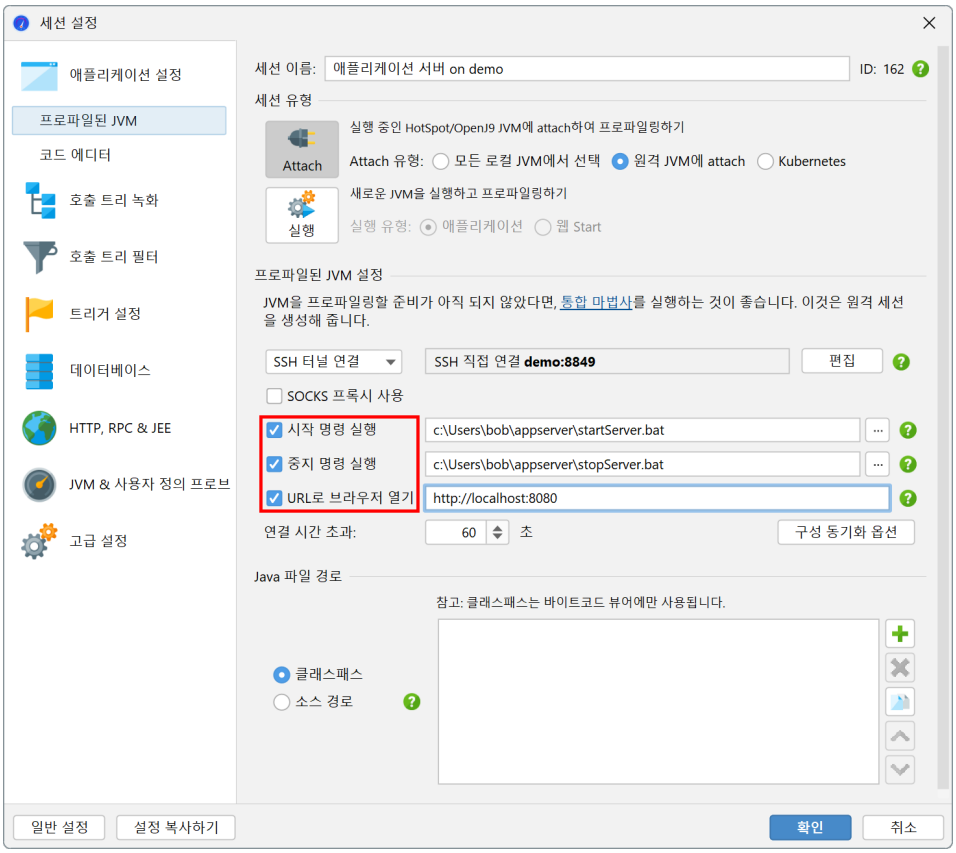
그러나 프로파일링 설정이 포함된 구성 파일을 지정할 수도 있으며, 이는 훨씬 더 효율적입니다. 이는 구성 동기화 단계에서 수행됩니다. 이 경우 주요 문제는 로컬에서 프로파일링 설정을 편집할 때마다 구성 파일을 원격 측과 동기화해야 한다는 것입니다. 가장 우아한 방법은 원격 주소 단계에서 SSH를 통해 원격 머신에 연결하는 것이며, 그러면 구성 파일이 SSH를 통해 자동으로 전송될 수 있습니다.



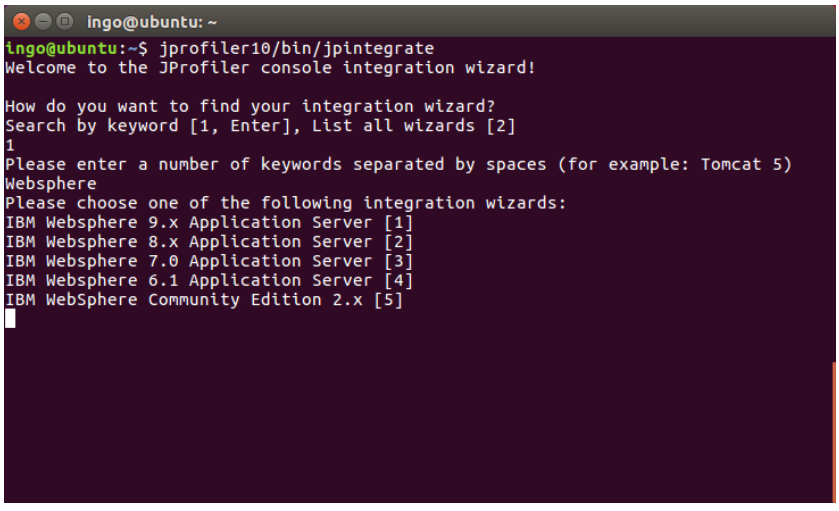
통합 마법사의 끝에서는 프로파일링을 시작하는 세션이 생성되며, 비일반적인 경우에는 애플리케이션 서버와 같은 서드파티 제품도 시작됩니다.



외부 시작 스크립트는 세션 구성 대화 상자의 애플리케이션 설정 탭에서 시작 스크립트 실행 및 중지 스크립트 실행 옵션으로 처리되며, URL은 URL로 브라우저 열기 체크 박스를 선택하여 표시할 수 있습니다. 여기에서 원격 머신의 주소와 구성 동기화 옵션을 변경할 수도 있습니다.



통합 마법사는 모두 프로파일링된 JVM이 원격 머신에서 실행되는 경우를 처리합니다. 그러나 구성 파일이나 시작 스크립트를 수정해야 하는 경우, 로컬 머신으로 복사하여 수정된 버전을 원격 머신으로 다시 전송해야 합니다. 명령줄 도구 `jpintegrate`를 원격 머신에서 직접 실행하여 수정 작업을 수행하는 것이 더 편리할 수 있습니다. `jpintegrate`는 JProfiler의 전체 설치가 필요하며 JProfiler GUI와 동일한 JRE 요구 사항을 가지고 있습니다.

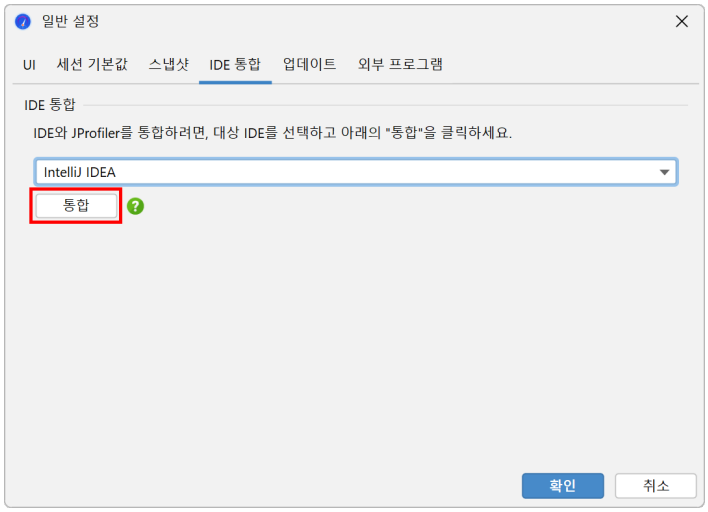


원격 프로파일링 세션을 시작할 때 오류가 발생하면, 문제를 해결하기 위해 수행할 수 있는 단계의 체크리스트를 보려면 문제 해결 가이드 [p. 228]를 참조하십시오.

IDE 통합

애플리케이션을 프로파일링하는 가장 편리한 방법은 IDE 통합을 통해서입니다. 개발 중에 IDE에서 애플리케이션을 시작하는 경우, IDE는 이미 필요한 모든 정보를 가지고 있으며 JProfiler 플러그인은 프로파일링을 위한 VM 매개변수를 간단히 추가하고, 필요시 JProfiler를 시작하고 프로파일링된 JVM을 JProfiler 메인 창에 연결할 수 있습니다.

모든 IDE 통합은 JProfiler 설치의 `integrations` 디렉토리에 포함되어 있습니다. 원칙적으로 해당 디렉토리의 아카이브는 해당 IDE의 플러그인 설치 메커니즘을 통해 수동으로 설치할 수 있습니다. 그러나 IDE 통합을 설치하는 선호되는 방법은 주 메뉴에서 세션->IDE 통합을 호출하는 것입니다.



IDE에서 프로파일링 세션은 JProfiler에서 자체 세션 항목을 얻지 못합니다. 이는 그러한 세션이 JProfiler GUI에서 시작될 수 없기 때문입니다. 프로파일링 설정은 프로젝트별 또는 실행 구성별로 설정에 따라 지속됩니다.

IDE에 연결되면 JProfiler는 도구 모음에 창 전환기를 표시하여 IDE의 관련 창으로 쉽게 이동할 수 있습니다. 모든 소스 보기 작업은 이제 JProfiler의 내장 소스 뷰어 대신 IDE에서 직접 소스를 표시합니다.

IDE 통합은 후속 장 [p. 133]에서 자세히 논의됩니다.

Attach 모드

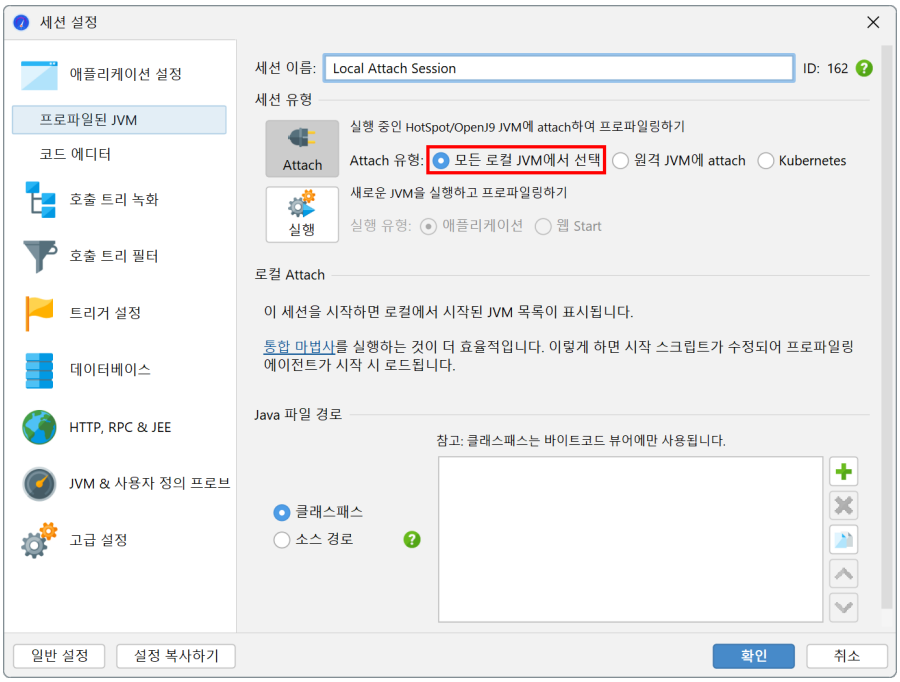
JVM을 프로파일링하려는 의도를 미리 결정할 필요는 없습니다. JProfiler의 attach 기능을 사용하면 실행 중인 JVM을 선택하고 프로파일링 에이전트를 즉시 로드할 수 있습니다. attach 모드는 편리하지만, 인지해야 할 몇 가지 단점이 있습니다:

- 프로파일링하려는 JVM을 실행 중인 JVM 목록에서 식별해야 합니다. 동일한 머신에서 많은 JVM이 실행 중인 경우 이는 때때로 까다로울 수 있습니다.
- 많은 클래스에 계측을 추가하기 위해 잠재적으로 많은 클래스가 재정의되어야 하기 때문에 추가적인 오버헤드가 있습니다.
- JProfiler의 일부 기능은 attach 모드에서 사용할 수 없습니다. 이는 주로 JVMTI의 일부 기능이 JVM이 초기화될 때만 켜질 수 있으며 JVM의 라이프사이클의 후반 단계에서는 사용할 수 없기 때문입니다.
- 일부 기능은 모든 클래스의 큰 부분에서 계측이 필요합니다. 클래스가 로드되는 동안 계측을 추가하는 것은 저렴하지만, 클래스가 이미 로드된 후에 계측을 추가하는 것은 그렇지 않습니다. 이러한 기능은 attach 모드를 사용할 때 기본적으로 비활성화됩니다.
- Attach 기능은 OpenJDK JVM, 버전 6 이상의 Oracle JVM, 최근 OpenJ9 JVM(8u281+, 11.0.11+ 또는 Java 17+) 또는 해당 릴리스를 기반으로 한 IBM JVM에서 지원됩니다. JVM에 대해 `-XX:`

`+PerfDisableSharedMem` 및 `-XX:+DisableAttachMechanism` VM 매개변수를 지정해서는 안 됩니다.

JProfiler의 시작 센터에 있는 빠른 Attach 탭은 프로파일링할 수 있는 모든 JVM을 나열합니다. 목록 항목의 배경색은 프로파일링 에이전트가 이미 로드되었는지, JProfiler GUI가 현재 연결되어 있는지 또는 오프라인 프로파일링이 구성되었는지를 나타냅니다.

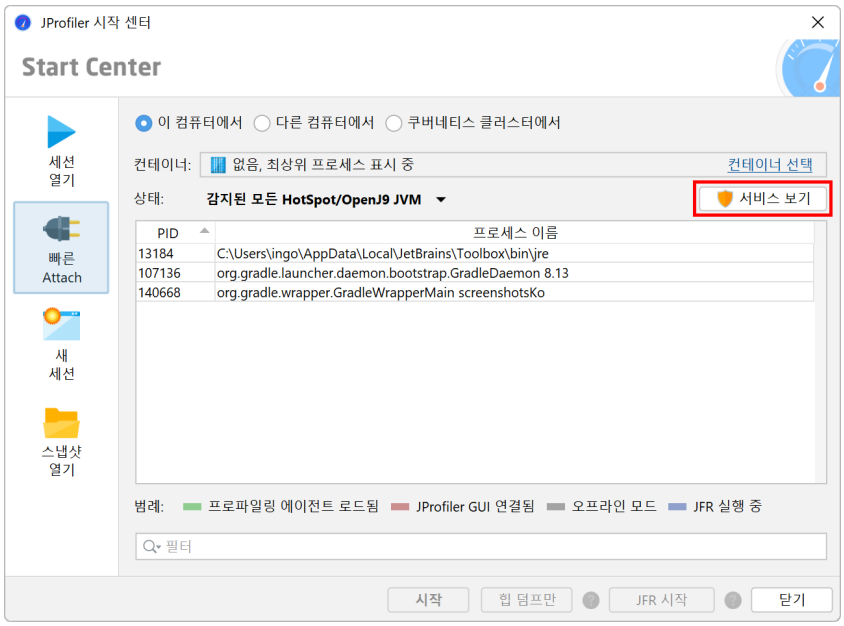
프로파일링 세션을 시작할 때 세션 설정 대화 상자에서 프로파일링 설정을 구성할 수 있습니다. 동일한 프로세스를 반복적으로 프로파일링할 때 동일한 구성을 반복해서 입력하고 싶지 않으므로, 빠른 attach 기능으로 생성된 세션을 닫을 때 영구 세션을 저장할 수 있습니다. 다음에 이 프로세스를 프로파일링하려면 빠른 Attach 탭 대신 세션 열기 탭에서 저장된 세션을 시작하십시오. 여전히 실행 중인 JVM을 선택해야 하지만, 프로파일링 설정은 이전에 구성한 것과 동일합니다.



로컬 서비스에 attach

JVM의 attach API는 호출 프로세스가 attach하려는 프로세스와 동일한 사용자로 실행되어야 하므로 JProfiler에 표시되는 JVM 목록은 현재 사용자로 제한됩니다. 다른 사용자가 시작한 프로세스는 대부분 서비스입니다. 서비스에 attach하는 방법은 Windows, Linux 및 Unix 기반 플랫폼에서 다릅니다.

Windows에서는 attach 대화 상자에 서비스 표시 버튼이 있어 로컬에서 실행 중인 모든 서비스를 나열합니다. JProfiler는 이러한 프로세스에 attach할 수 있도록 브리지 실행 파일을 시작하여 어떤 사용자로 실행되든 상관없이 attach할 수 있습니다.



Linux에서는 JProfiler가 대부분의 Linux 배포판의 일부인 PolicyKit을 통해 UI에서 직접 사용자를 전환하는 것을 지원합니다. attach 대화 상자에서 사용자 전환을 클릭하면 다른 사용자 이름을 입력하고 시스템 비밀번호 대화 상자로 인증할 수 있습니다.

macOS를 포함한 Unix 기반 플랫폼에서는 su 또는 sudo를 사용하여 다른 사용자로 명령줄 도구 `jpenable`을 실행할 수 있습니다. 이는 Unix 변형 또는 Linux 배포판에 따라 다릅니다. macOS 및 Ubuntu와 같은 Debian 기반 Linux 배포판에서는 sudo가 사용됩니다.

sudo를 사용하여 호출

```
sudo -u userName jpenable
```

su를 사용하면 필요한 명령줄은

```
su userName -c jpenable
```

`jpenable`은 JVM을 선택하고 프로파일링 에이전트가 수신 대기 중인 포트를 알려줍니다. 그 후에는 JProfiler UI의 로컬 세션 또는 jpenable이 제공한 포트에 직접 연결하는 SSH 연결로 연결할 수 있습니다.

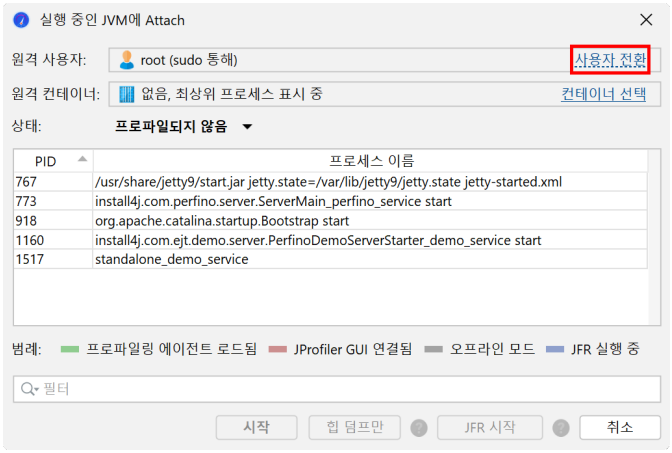
원격 머신의 JVM에 attach

프로파일링을 위한 가장 까다로운 설정은 원격 프로파일링입니다 - JProfiler GUI는 로컬 머신에서 실행되고 프로파일링된 JVM은 다른 머신에서 실행됩니다. 프로파일링된 JVM에 -agentpath VM 매개변수를 전달하는 설정의 경우, 원격 머신에 JProfiler를 설치하고 로컬 머신에 원격 세션을 설정해야 합니다. JProfiler의 원격 attach 기능을 사용하면 이러한 수정이 필요하지 않습니다. 원격 머신에 로그인하기 위한 SSH 자격 증명만 필요합니다.

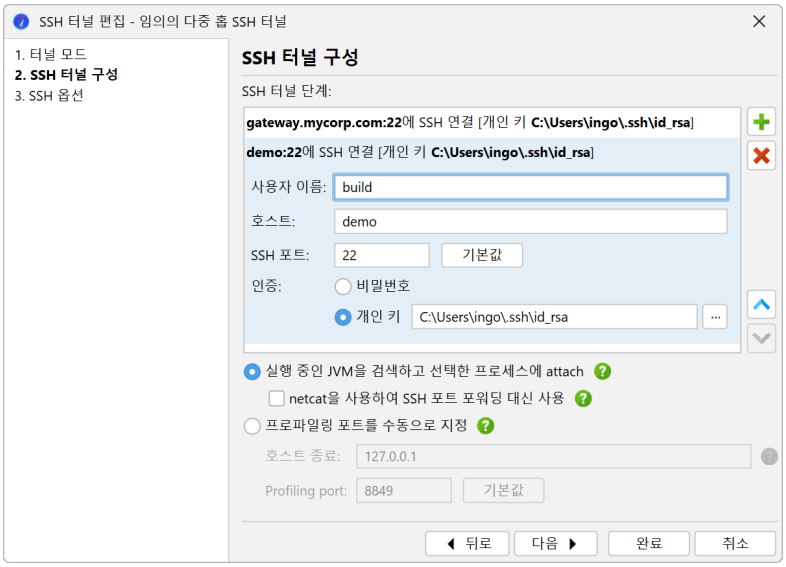
SSH 연결을 통해 JProfiler는 "JProfiler 설치" [p. 7] 도움말 주제에서 논의된 에이전트 패키지를 업로드하고 원격 머신에서 포함된 명령줄 도구를 실행할 수 있습니다. 로컬 머신에 SSH가 설정되어 있을 필요는 없으며, JProfiler는 자체 구현을 제공합니다. 가장 간단한 설정에서는 호스트, 사용자 이름 및 인증을 정의하기만 하면 됩니다.

SSH 연결을 통해 JProfiler는 실행 중인 JVM을 자동으로 검색하거나 프로파일링 에이전트가 이미 수신 대기 중인 특정 포트에 연결할 수 있습니다. 후자의 경우, 위에서 설명한 대로 원격 머신에서 `jpenable` 또는 `jpintegrate`를 사용하여 프로파일링을 위한 특수 JVM을 준비할 수 있습니다. 그런 다음, SSH 원격 attach를 구성하여 구성된 프로파일링 포트에 직접 연결할 수 있습니다.

자동 검색은 SSH 로그인 사용자로 시작된 원격 머신의 모든 JVM을 나열합니다. 대부분의 경우, 이는 프로파일링하려는 서비스를 시작한 사용자가 아닐 것입니다. 서비스 시작 사용자는 일반적으로 SSH 연결이 허용되지 않기 때문에, JProfiler는 사용자 전환 하이퍼링크를 추가하여 sudo 또는 su를 사용하여 해당 사용자로 전환할 수 있습니다.

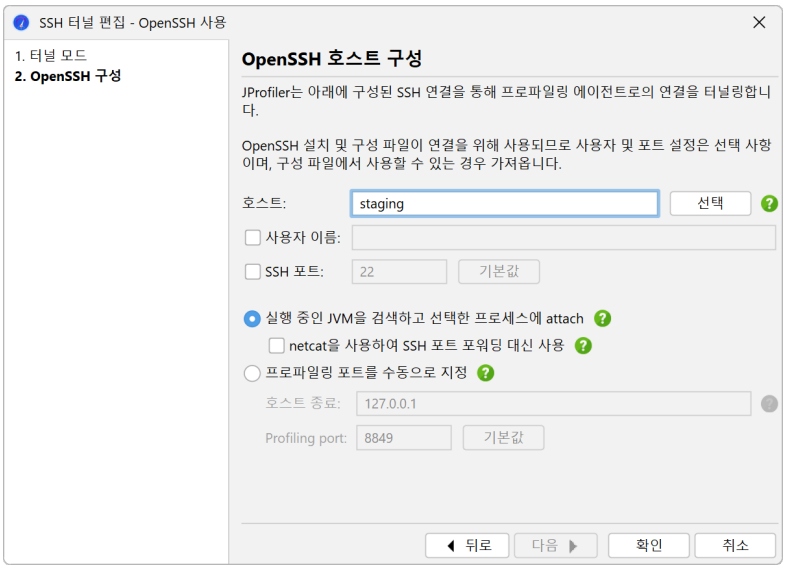


복잡한 네트워크 토폴로지에서는 원격 머신에 직접 연결할 수 없는 경우가 있습니다. 이 경우, JProfiler에 GUI에서 멀티 홉 SSH 터널로 연결하도록 지시할 수 있습니다. SSH 터널의 끝에서 하나의 직접 네트워크 연결을 수행할 수 있으며, 일반적으로 "127.0.0.1"로 연결합니다.



다른 인증 메커니즘의 경우, OpenSSH 터널 모드를 사용할 수 있습니다. OpenSSH 실행 파일을 사용할 때 명령줄에서 입력하는 것처럼 호스트 이름을 입력합니다. 이는 OpenSSH 구성 파일에 구성된 별칭일 수도 있습니다. Windows에서는 Microsoft의 내장 OpenSSH 클라이언트만 지원됩니다.

SSH 옵션 텍스트 필드는 명령줄에서 OpenSSH 실행 파일에 지정할 임의의 추가 인수를 허용합니다. 이는 특히 AWS 세션 관리자를 통해 SSH 연결을 터널링하는 방법에 대한 튜토리얼에서 제공된 지침을 따를 때 유용합니다.

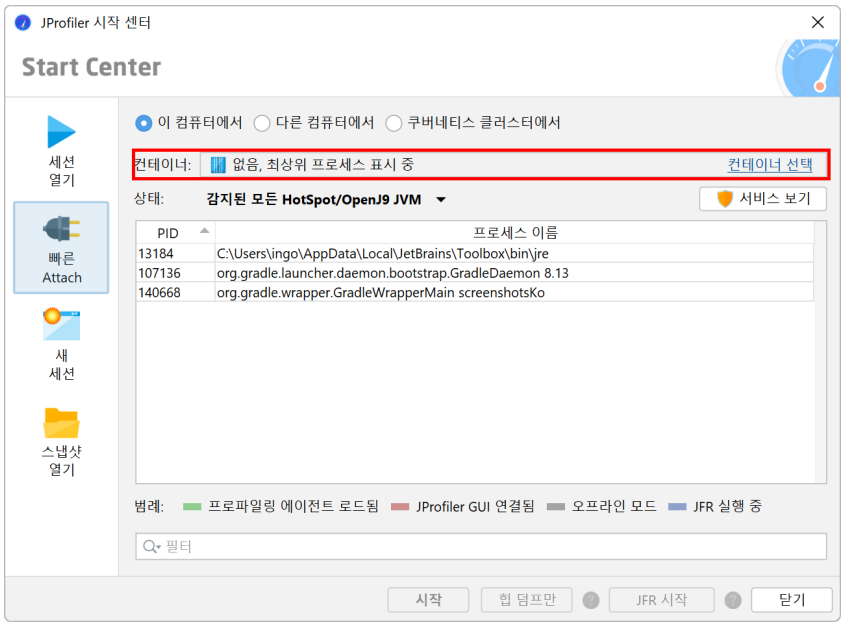


HPROF 스냅샷은 SSH 로그인 사용자로 시작된 JVM에 대해서만 촬영할 수 있습니다. 이는 HPROF 스냅샷이 JVM을 시작한 사용자의 액세스 권한으로 작성된 중간 파일을 필요로 하기 때문입니다. 보안상의 이유로 다운로드를 위해 SSH 로그인 사용자에게 파일 권한을 전송하는 것은 불가능합니다. 전체 프로파일링 세션에는 이러한 제한이 없습니다.

Docker 컨테이너에서 실행 중인 JVM에 attach

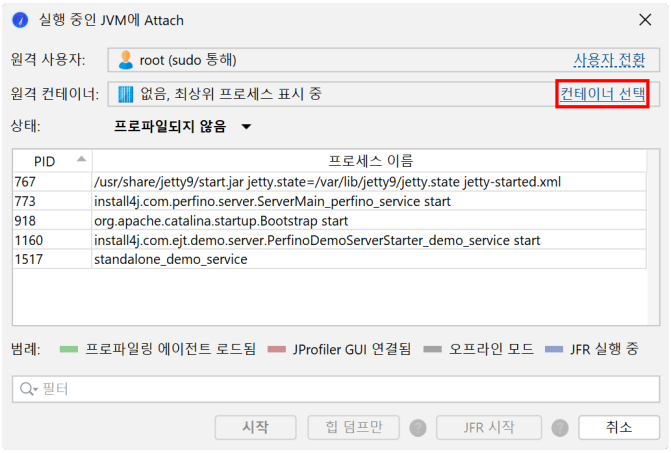
Docker 컨테이너에는 일반적으로 SSH 서버가 설치되어 있지 않으며, Docker 컨테이너에서 jpenable을 사용할 수 있지만, Docker 파일에 명시하지 않는 한 프로파일링 포트는 외부에서 액세스할 수 없습니다.

JProfiler에서는 Windows 또는 macOS의 로컬 Docker Desktop 설치에서 실행 중인 JVM에 attach할 수 있으며, 빠른 attach 대화 상자에서 Docker 컨테이너를 선택하여 수행할 수 있습니다. 기본적으로 JProfiler는 docker 실행 파일의 경로를 자동으로 감지합니다. 또는 일반 설정 대화 상자의 "외부 도구" 탭에서 이를 구성할 수 있습니다.



컨테이너를 선택한 후에는 Docker 컨테이너 내에서 실행 중인 모든 JVM이 표시됩니다. JVM을 선택하면 JProfiler는 Docker 명령을 사용하여 선택한 컨테이너에 프로파일링 에이전트를 자동으로 설치하고, JVM을 프로파일링할 준비를 하고, 프로파일링 프로토콜을 외부로 터널링합니다.

원격 Docker 설치의 경우, SSH 원격 attach 기능을 사용한 다음 원격 머신에서 Docker 컨테이너를 선택할 수 있습니다. 로그인 사용자가 docker 그룹에 속하지 않은 경우, 위에서 설명한 대로 먼저 사용자를 전환할 수 있습니다.

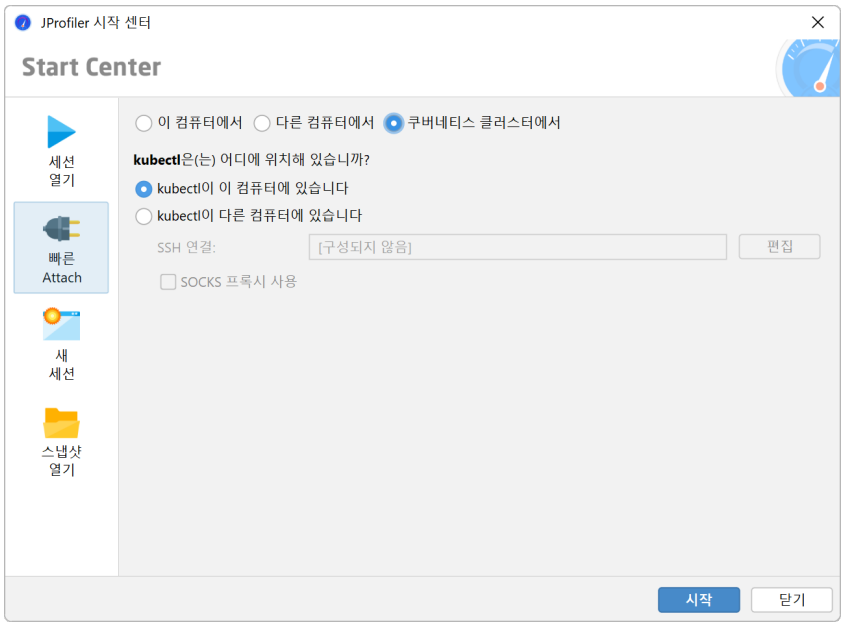


원격 attach 대화 상자의 컨테이너 선택 하이퍼링크를 사용하여 실행 중인 Docker 컨테이너를 선택하고 그 안에서 실행 중인 모든 JVM을 표시할 수 있습니다.

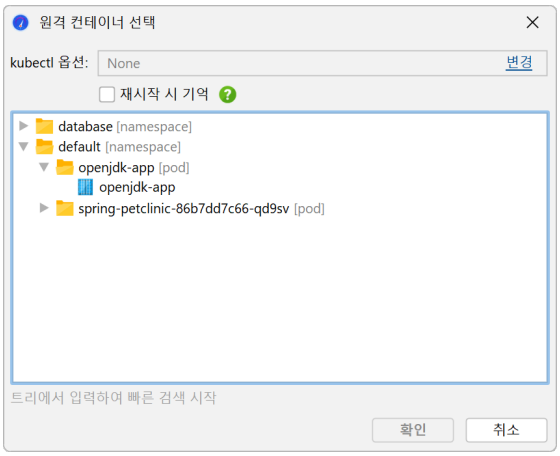
Kubernetes 클러스터에서 실행 중인 JVM에 attach

Kubernetes 클러스터에서 실행 중인 JVM을 프로파일링하기 위해 JProfiler는 `kubectl` 명령줄 도구를 사용하여 포드 및 컨테이너를 검색하고, 컨테이너에 연결하여 JVM을 나열하고, 선택한 JVM에 연결합니다.

`kubectl` 명령줄 도구는 로컬 컴퓨터에 있거나 SSH 액세스 권한이 있는 원격 머신에 있을 수 있습니다. JProfiler는 두 시나리오를 직접 지원합니다. 로컬 설치의 경우, JProfiler는 `kubectl`의 경로를 자동으로 감지하려고 하지만, 일반 설정 대화 상자의 "외부 도구" 탭에서 명시적인 경로를 구성할 수 있습니다.



JProfiler는 세 레벨의 트리로 감지된 모든 컨테이너를 나열합니다. 최상위에는 네임스페이스 노드가 있으며, 감지된 포드를 포함하는 자식 노드가 있습니다. 리프 노드는 컨테이너 자체이며, 실행 중인 JVM을 선택하려면 이 중 하나를 선택해야 합니다.



`kubectl`은 Kubernetes 클러스터에 연결할 수 있도록 인증을 위한 추가 명령줄 옵션이 필요할 수 있습니다. 이러한 옵션은 컨테이너 선택 대화 상자의 상단에 입력할 수 있습니다. 이러한 옵션은 민감한 정보일 수 있으므로, 재시작 시 이를 기억하도록 선택한 경우에만 디스크에 저장됩니다. 이 체크박스를 선택 해제하면 이전에 저장된 정보가 즉시 지워집니다.

실행 중인 JVM의 표시 이름 설정

JVM 선택 테이블에서 표시되는 프로세스 이름은 프로파일링된 JVM의 메인 클래스와 해당 인수입니다. exe4j 또는 install4j에 의해 생성된 런처의 경우 실행 파일 이름이 표시됩니다.

예를 들어 동일한 메인 클래스를 가진 여러 프로세스가 있어 구분할 수 없는 경우, 표시 이름을 직접 설정하려면 `-Djprofiler.displayName=[name]` VM 매개변수를 설정할 수 있습니다. 이름에 공백이 포함된 경우, 단일 인용부호를 사용하십시오: `-Djprofiler.displayName='My name with spaces'` 및 필요한 경우 전체 VM 매개변수를 큰따옴표로 묶으십시오. `-Djprofiler.displayName` 외에도 JProfiler는 `-Dvisualvm.display.name`을 인식합니다.

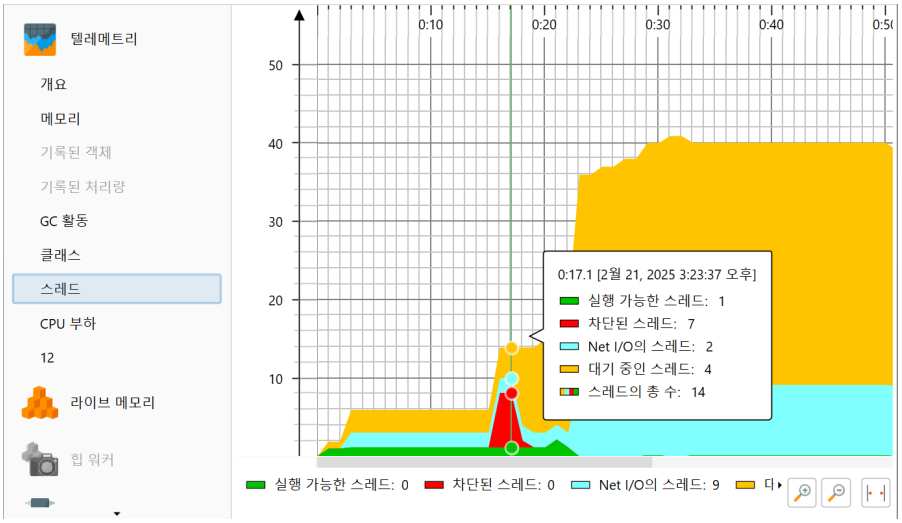
데이터 녹화

프로파일러의 주요 목적은 일반적인 문제를 해결하는 데 유용한 다양한 소스의 런타임 데이터를 기록하는 것입니다. 이 작업의 주요 문제는 실행 중인 JVM이 엄청난 속도로 데이터를 생성한다는 것입니다. 프로파일러가 항상 모든 유형의 데이터를 기록한다면, 이는 용납할 수 없는 오버헤드를 발생시키거나 사용 가능한 모든 메모리를 빠르게 소모할 것입니다. 또한, 특정 사용 사례 주위의 데이터를 기록하고 관련 없는 활동은 보지 않기를 원할 때가 많습니다.

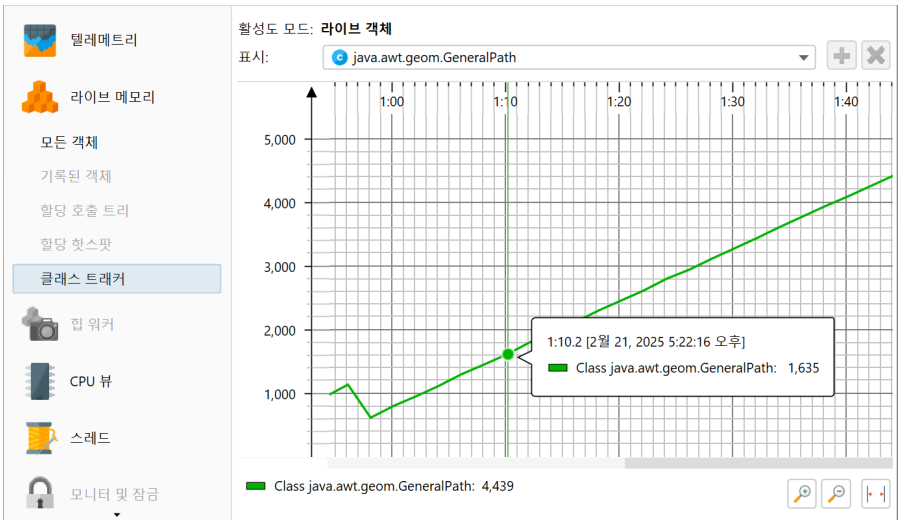
이러한 이유로 JProfiler는 실제로 관심 있는 정보의 기록을 제어하기 위한 세밀한 메커니즘을 제공합니다.

스칼라 값 및 텔레메트리

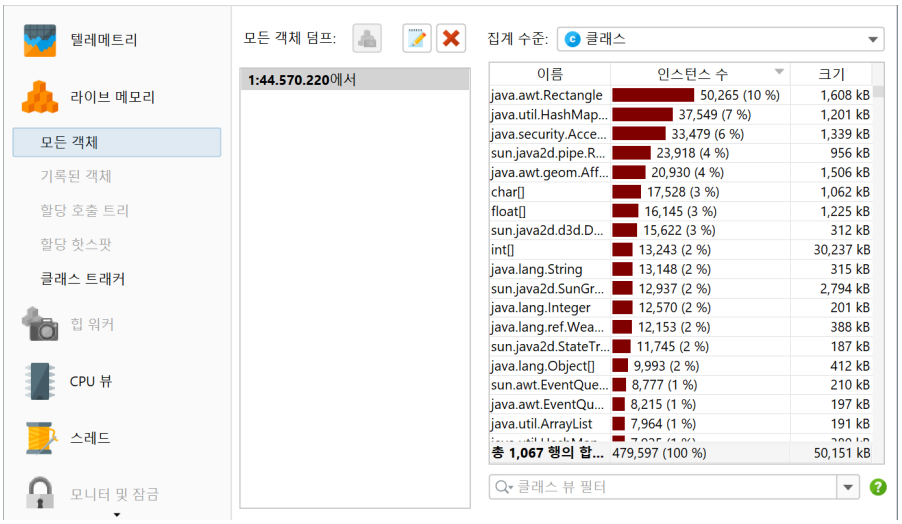
프로파일러의 관점에서 가장 문제가 적은 데이터 형태는 스칼라 값입니다. 예를 들어, 활성 스레드의 수나 열린 JDBC 연결의 수입니다. JProfiler는 이러한 값을 고정된 거시적 빈도로 샘플링할 수 있으며, 일반적으로 초당 한 번씩 시간 경과에 따른 변화를 보여줍니다. JProfiler에서 이러한 데이터를 보여주는 뷰는 텔레메트리 [p. 44] 라고 합니다. 대부분의 텔레메트리는 측정의 오버헤드와 메모리 소비가 적기 때문에 항상 기록됩니다. 데이터가 오랜 시간 동안 기록되면, 오래된 데이터 포인트는 통합되어 메모리 소비가 시간에 따라 선형적으로 증가하지 않도록 합니다.



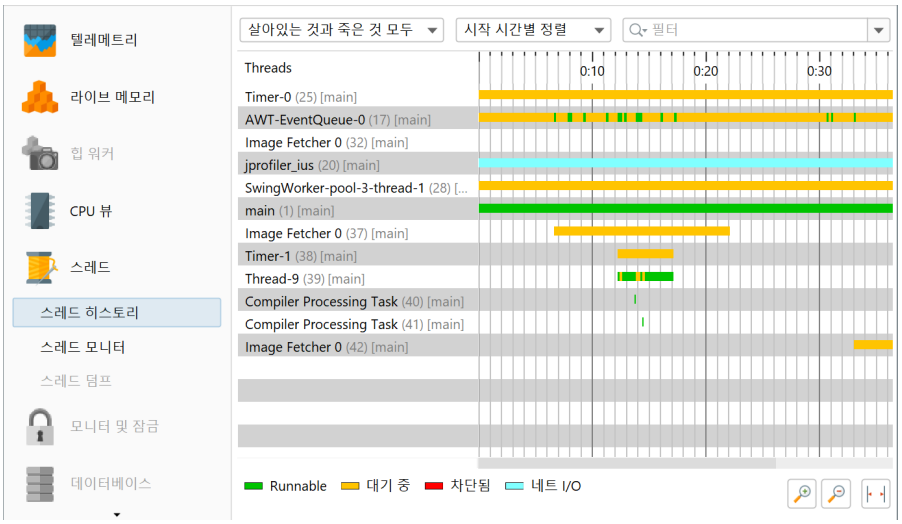
각 클래스의 인스턴스 수와 같은 매개변수가 있는 텔레메트리도 있습니다. 추가 차원은 영구적인 연대기적 기록을 지속 불가능하게 만듭니다. JProfiler에게 선택한 여러 클래스의 인스턴스 수에 대한 텔레메트리를 기록하도록 지시할 수 있지만, 모든 클래스에 대해서는 기록하지 않습니다.



이전 예를 계속해서, JProfiler는 모든 클래스의 인스턴스 수를 보여줄 수 있지만, 연대기적 정보는 없습니다. 이것이 "모든 객체" 뷰이며, 각 클래스를 테이블의 행으로 보여줍니다. 뷰 업데이트 빈도는 초당 한 번보다 낮으며, 측정이 얼마나 많은 오버헤드를 초래하는지에 따라 자동으로 조정될 수 있습니다. 모든 클래스의 인스턴스 수를 결정하는 것은 상대적으로 비용이 많이 들며, 힙에 객체가 많을수록 더 오래 걸립니다. 이 때문에 "모든 객체"는 자동으로 업데이트되지 않으며, 모든 객체의 새로운 덤프를 수동으로 생성해야 합니다.



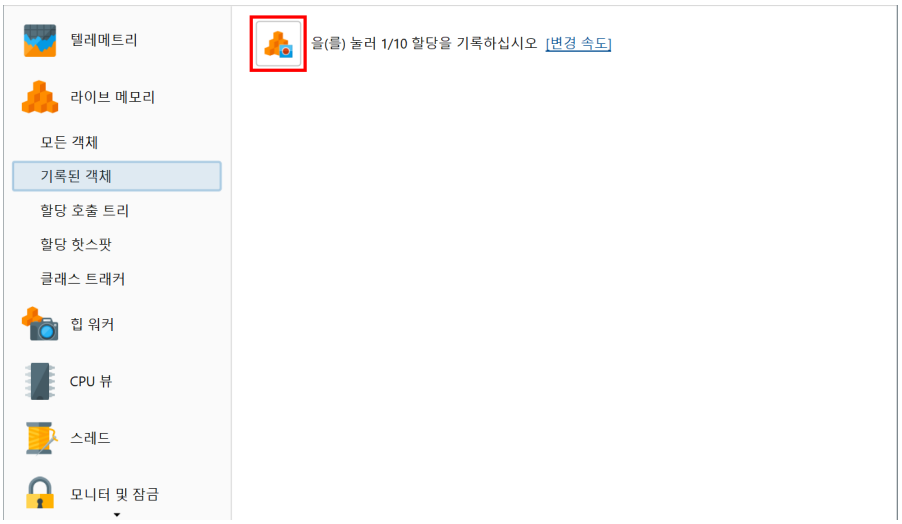
일부 측정은 스레드가 현재 있는 실행 상태와 같은 열거형 값을 캡처합니다. 이러한 종류의 측정은 색상화된 타임라인으로 표시될 수 있으며, 수치 텔레메트리보다 메모리를 훨씬 적게 소비합니다. 스레드 상태의 경우, "스레드 히스토리" 뷰는 JVM의 모든 스레드에 대한 타임라인을 보여줍니다. 수치 값이 있는 텔레메트리와 마찬가지로, 오래된 값은 통합되어 메모리 소비를 줄이기 위해 더 거칠게 만들어집니다.



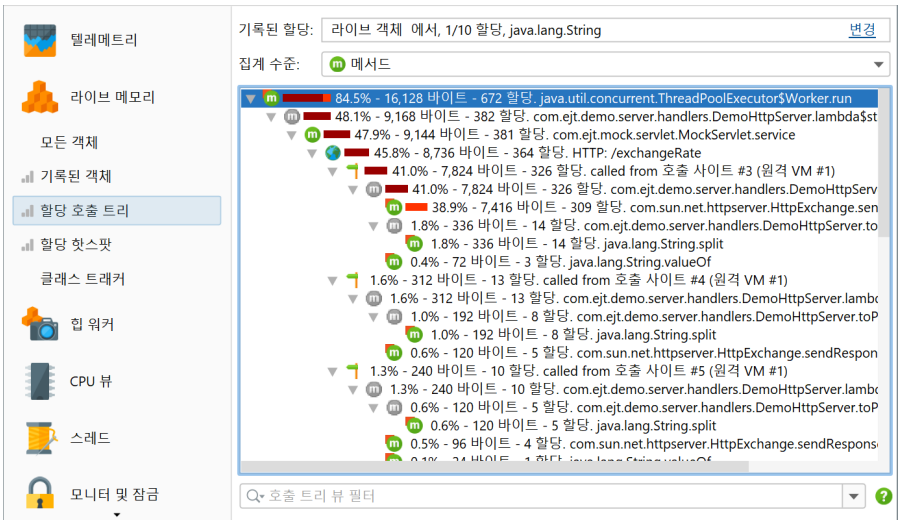
할당 기록

특정 시간 간격 동안 할당된 인스턴스 수에 관심이 있다면, JProfiler는 모든 할당을 추적해야 합니다. JProfiler가 힙의 모든 객체를 반복하여 필요할 때 정보를 얻을 수 있는 "모든 객체" 뷰와 달리, 단일 할당을 추적하려면 각 객체 할당에 대해 추가 코드가 실행되어야 합니다. 이는 매우 비용이 많이 드는 측정으로, 많은 객체를 할당할 경우 프로파일된 애플리케이션의 런타임 특성, 특히 성능 핫스팟을 크게 변경할 수 있습니다. 이 때문에 할당 기록은 명시적으로 시작하고 중지해야 합니다.

관련된 녹화가 있는 뷰는 처음에 녹화 버튼이 있는 빈 페이지를 보여줍니다. 동일한 녹화 버튼은 도구 모음에서도 찾을 수 있습니다.



할당 기록은 할당된 인스턴스 수뿐만 아니라 할당 스택 트레이스도 기록합니다. 메모리에 할당된 각 녹화의 스택 트레이스를 유지하면 과도한 오버헤드를 생성하므로, JProfiler는 기록된 스택 트레이스를 트리로 누적합니다. 이렇게 하면 데이터를 훨씬 쉽게 해석할 수 있는 장점이 있습니다. 그러나 연대기적 측면은 사라지고 특정 시간 범위를 데이터에서 추출할 수 있는 방법은 없습니다.

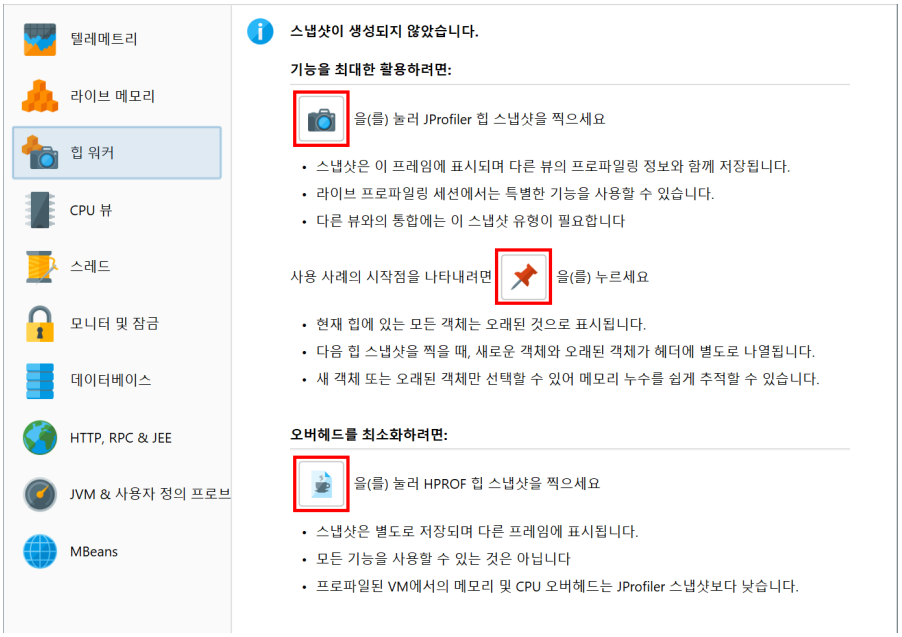


메모리 분석

할당 기록은 객체가 할당된 위치만 측정할 수 있으며, 객체 간의 참조에 대한 정보는 없습니다. 메모리 누수 해결과 같은 참조가 필요한 모든 메모리 분석은 힙 워커에서 수행됩니다. 힙 워커는 전체 힙의 스냅샷을 찍고 분석합니다. 이는 JVM을 일시 중지시키는 침습적인 작업으로, 잠재적으로 오랜 시간이 걸리며 많은 메모리가 필요합니다.

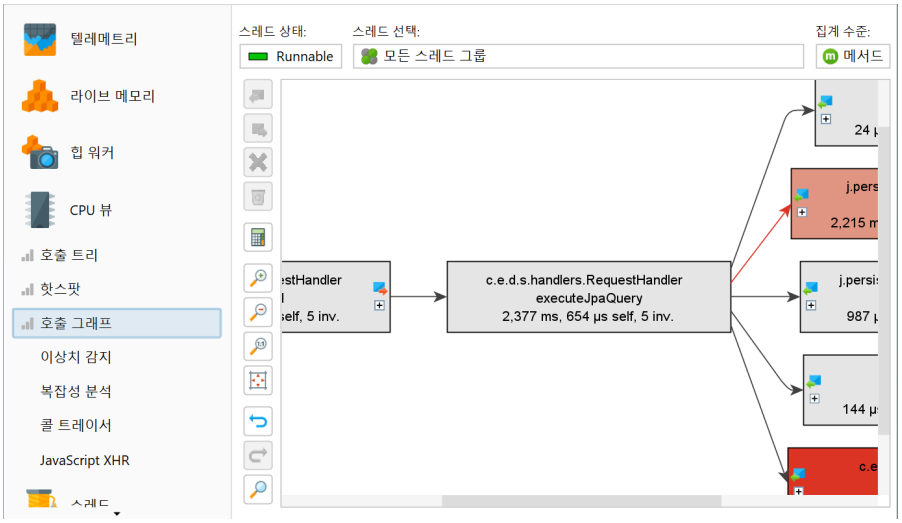
더 가벼운 작업은 사용 사례를 시작하기 전에 힙의 모든 객체를 표시하여 나중에 힙 스냅샷을 찍을 때 새로 할당된 모든 객체를 찾을 수 있도록 하는 것입니다.

JVM에는 전체 힙을 파일로 덤프하는 특별한 트리거가 있으며, 이는 오래된 HPROF 프로파일링 에이전트의 이름을 따서 명명되었습니다. 이는 프로파일링 인터페이스와 관련이 없으며, 그 제약 하에 작동하지 않습니다. 이 때문에 HPROF 힙 덤프는 더 빠르고 적은 리소스를 사용합니다. 단점은 힙 워커에서 힙 스냅샷을 볼 때 JVM과의 실시간 연결이 없으며 일부 기능을 사용할 수 없다는 것입니다.

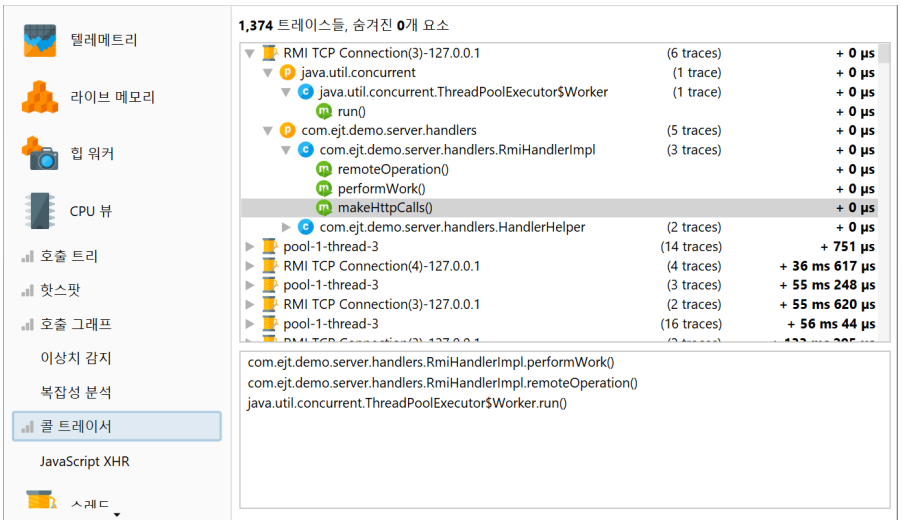


메서드 호출 기록

메서드 호출이 얼마나 오래 걸리는지를 측정하는 것은 할당 기록과 마찬가지로 선택적 녹화입니다. 메서드 호출은 트리에 누적되며, 호출 그래프와 같은 다양한 관점에서 기록된 데이터를 보여주는 다양한 뷰가 있습니다. 이 유형의 데이터에 대한 녹화는 JProfiler에서 "CPU 녹화"라고 합니다.

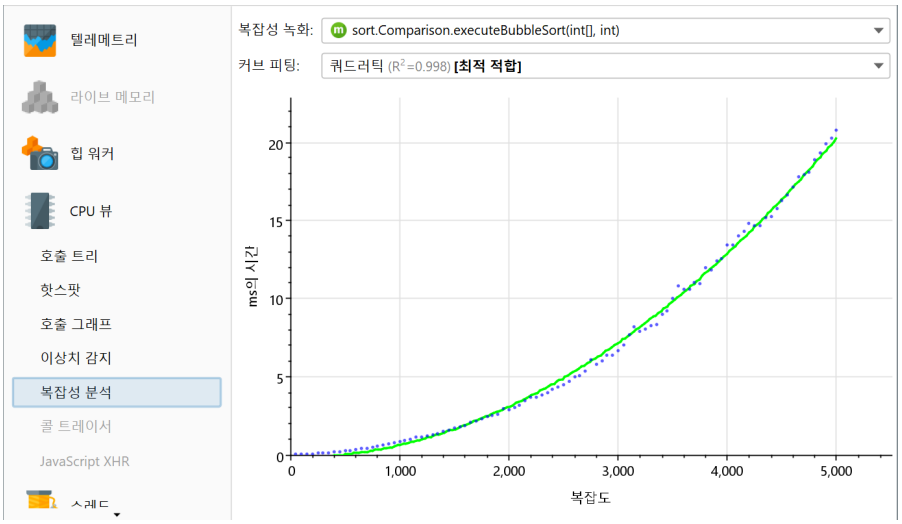


특정 상황에서는 특히 여러 스레드가 관련된 경우 메서드 호출의 연대기적 순서를 보는 것이 유용할 수 있습니다. 이러한 특별한 경우를 위해 JProfiler는 "콜 트레이서" 뷰를 제공합니다. 해당 뷰는 더 일반적인 CPU 녹화와 연결되지 않은 별도의 녹화 유형을 가지고 있습니다. 콜 트레이서는 성능 문제를 해결하는 데 유용할 만큼 많은 데이터를 생성하지 않으며, 이는 전문화된 디버깅 형태에만 사용됩니다.



콜 트레이서는 CPU 녹화에 의존하며, 필요할 경우 자동으로 이를 켭니다.

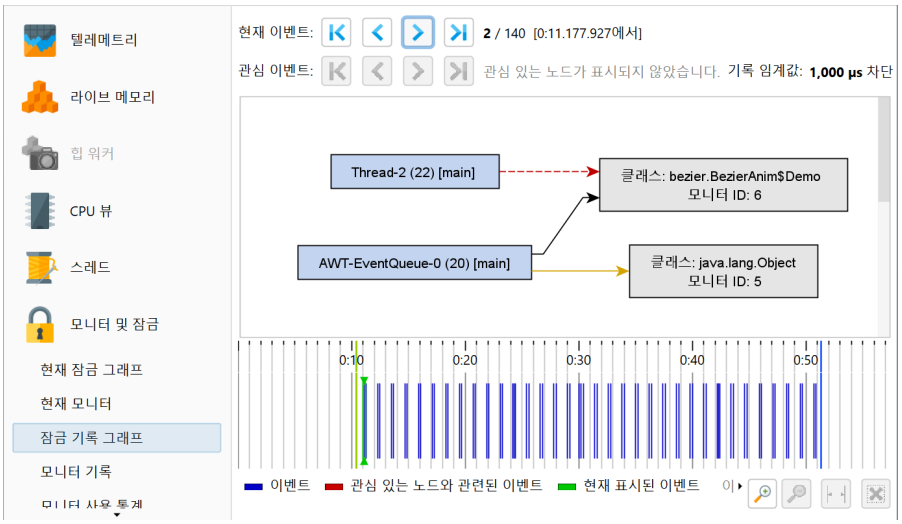
자체 녹화를 가진 또 다른 전문화된 뷰는 "복잡도 분석"입니다. 이는 선택된 메서드의 실행 시간만 측정하며, CPU 녹화가 활성화될 필요는 없습니다. 추가 데이터 축은 스크립트를 사용하여 계산할 수 있는 메서드 호출의 알고리즘 복잡성에 대한 수치 값입니다. 이렇게 하면 메서드의 실행 시간이 매개변수에 어떻게 의존하는지를 측정할 수 있습니다.



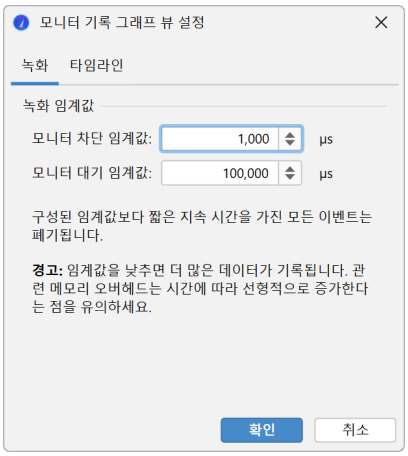
모니터 기록

스레드가 대기하거나 차단되는 이유를 분석하려면 해당 이벤트를 기록해야 합니다. 이러한 이벤트의 비율은 크게 다릅니다. 스레드가 자주 작업을 조정하거나 공통 자원을 공유하는 멀티스레드 프로그램의 경우, 이러한 이벤트가 엄청나게 많을 수 있습니다. 이 때문에 이러한 연대기적 데이터는 기본적으로 기록되지 않습니다.

모니터 기록을 켜면 "잠금 히스토리 그래프"와 "모니터 히스토리" 뷰가 데이터를 보여주기 시작합니다.

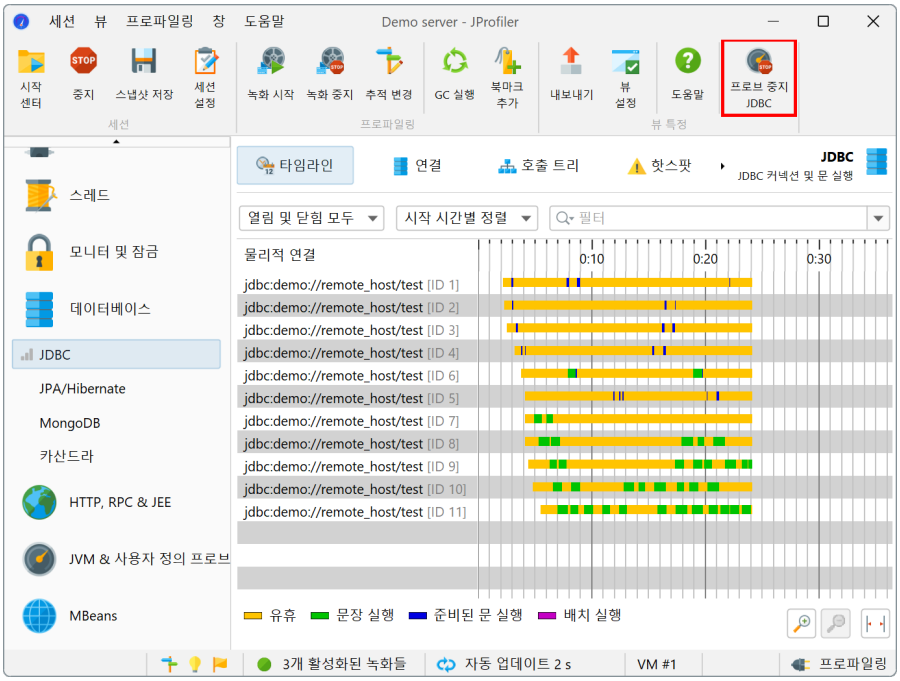


잡음을 제거하고 메모리 소비를 줄이기 위해 매우 짧은 이벤트는 기록되지 않습니다. 뷰 설정에서 이러한 임계값을 조정할 수 있습니다.

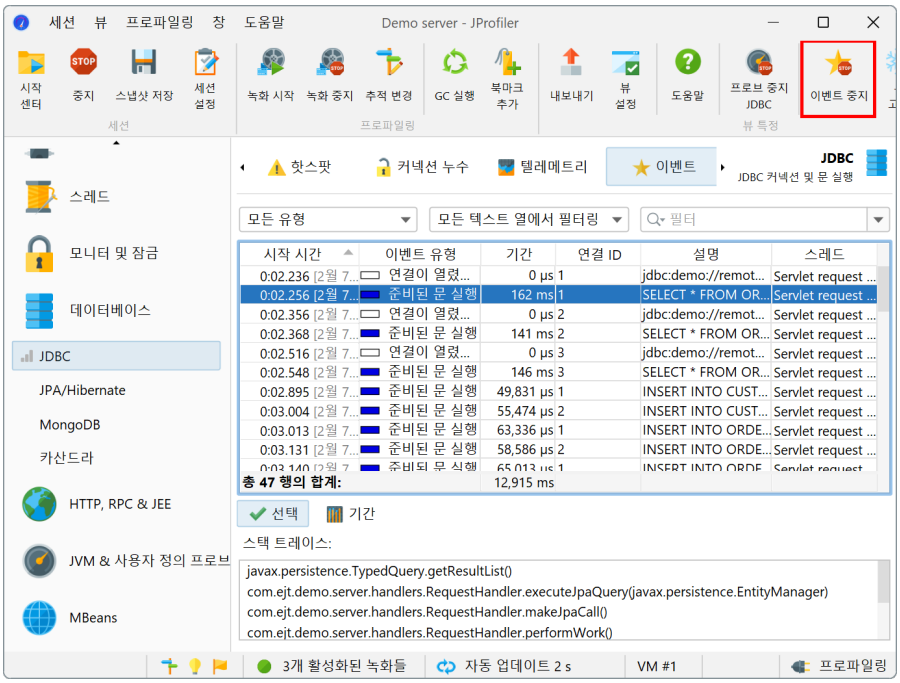


프로브 기록

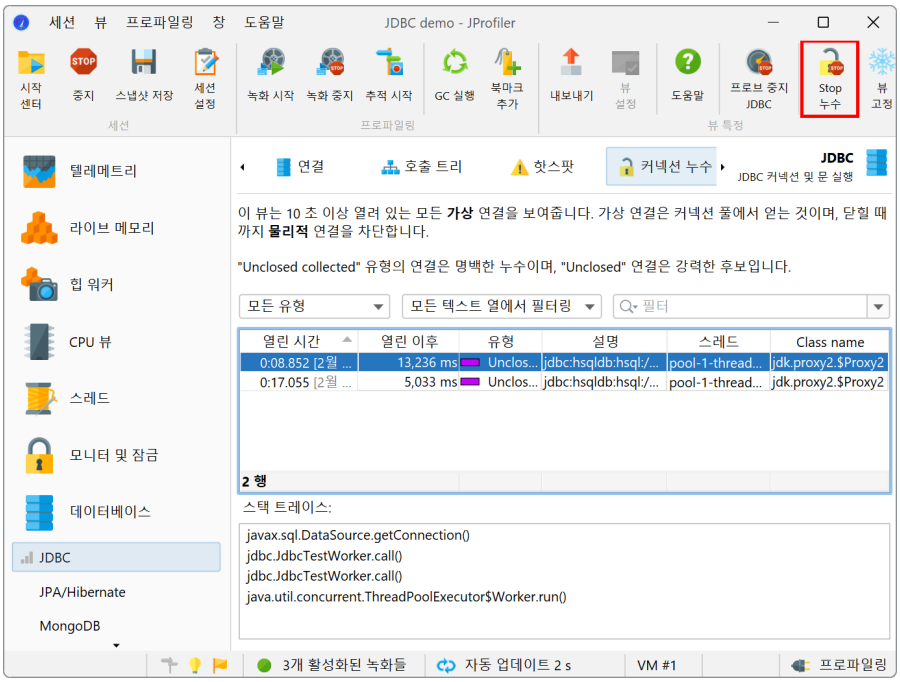
프로브는 JDBC 호출이나 파일 작업과 같은 JVM의 상위 수준 하위 시스템을 보여줍니다. 기본적으로 프로브는 기록되지 않으며, 각 프로브에 대해 별도로 기록을 전환할 수 있습니다. 일부 프로브는 매우 적은 오버헤드를 추가하거나 전혀 추가하지 않으며, 일부는 애플리케이션이 수행하는 작업과 프로브가 구성된 방식에 따라 상당한 양의 데이터를 생성할 수 있습니다.



할당 기록 및 메서드 호출 기록과 마찬가지로 프로브 데이터는 누적되며, 타임라인 및 텔레메트리를 제외하고는 연대기적 정보가 폐기됩니다. 그러나 대부분의 프로브에는 단일 이벤트를 검사할 수 있는 "이벤트" 뷰가 있으며, 이는 잠재적으로 큰 오버헤드를 추가하고 별도의 녹화 작업을 가지고 있습니다. 해당 녹화 작업의 상태는 지속되므로, 프로브 기록을 전환할 때 이전에 켰던 경우 관련 이벤트 기록도 함께 전환됩니다.

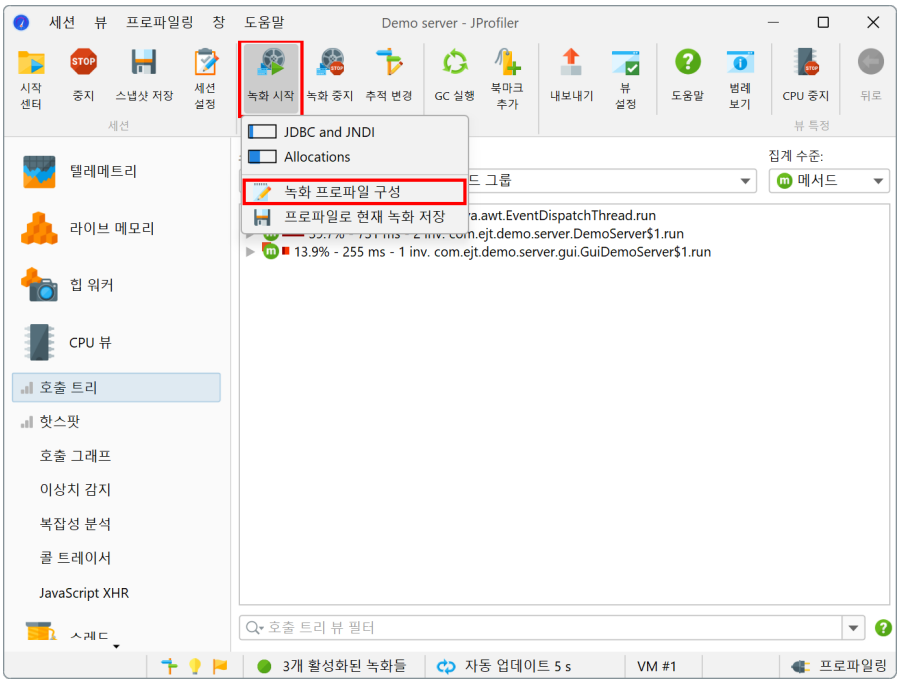


JDBC 프로브에는 JDBC 연결 누수를 기록하기 위한 세 번째 녹화 작업이 있습니다. 연결 누수를 찾는 것과 관련된 오버헤드는 실제로 그러한 문제를 조사하려고 할 때만 발생합니다. 이벤트 기록 작업과 마찬가지로 누수 기록 작업의 선택 상태는 지속됩니다.

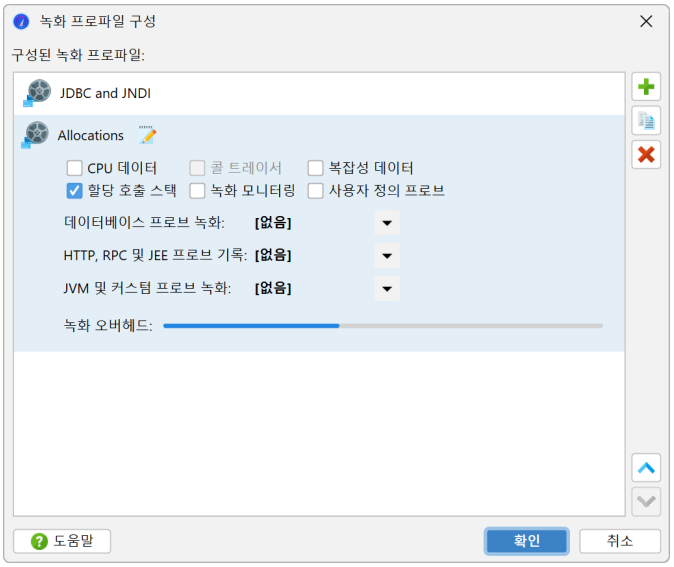


녹화 프로파일

많은 상황에서 단일 클릭으로 다양한 녹화를 시작하거나 중지하고 싶을 것입니다. 모든 관련 뷰를 방문하고 녹화 버튼을 하나씩 전환하는 것은 비실용적일 것입니다. 이러한 이유로 JProfiler에는 녹화 프로파일이 있습니다. 도구 모음에서 녹화 시작 버튼을 클릭하여 녹화 프로파일을 생성할 수 있습니다.

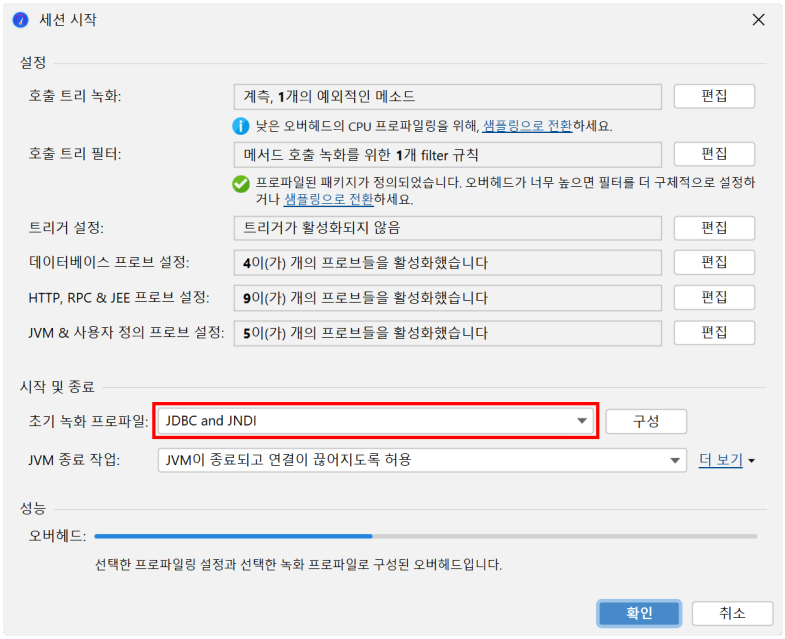


녹화 프로파일은 원자적으로 활성화할 수 있는 특정 녹화 조합을 정의합니다. JProfiler는 선택한 녹화로 인해 생성되는 오버헤드에 대한 대략적인 인상을 주려고 하며, 문제 있는 조합을 피하도록 합니다. 특히, 할당 기록과 CPU 기록은 잘 맞지 않으며, 할당 기록은 CPU 데이터의 타이밍을 크게 왜곡할 것입니다.



세션이 실행 중일 때 언제든지 녹화 프로파일을 활성화할 수 있습니다. 녹화 프로파일은 추가적이지 않으며, 녹화 프로파일에 포함되지 않은 모든 녹화를 중지합니다. 녹화 중지 버튼을 사용하여 어떻게 활성화되었든 모든 녹화를 중지할 수 있습니다. 현재 활성화된 녹화를 확인하려면 상태 표시줄의 녹화 레이블 위로 마우스를 가져가십시오.

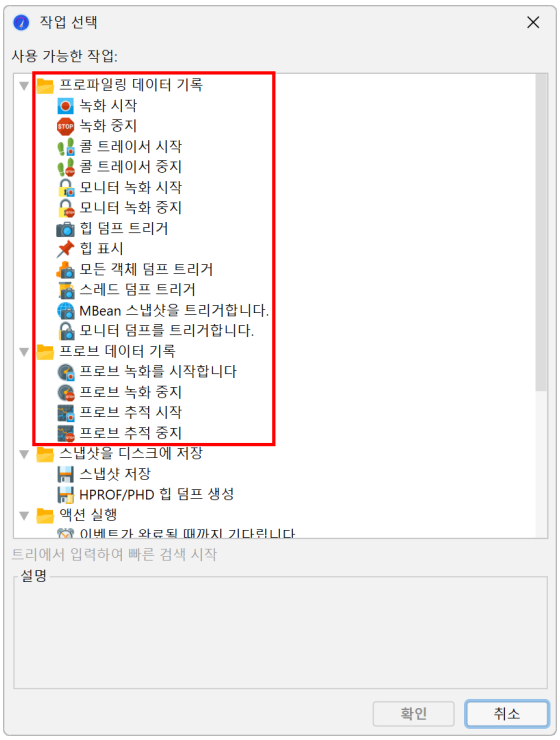
녹화 프로파일은 프로파일링을 시작할 때 직접 활성화할 수도 있습니다. "세션 시작" 대화 상자에는 초기 녹화 프로파일 드롭다운이 있습니다. 기본적으로 녹화 프로파일이 선택되지 않지만, JVM의 시작 단계에서 데이터를 필요로 하는 경우 필요한 녹화를 구성할 수 있는 곳입니다.



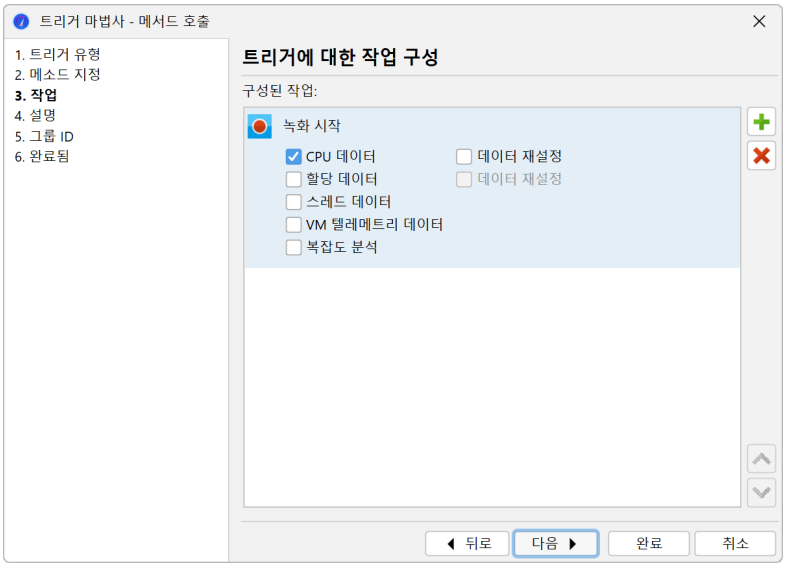
트리거를 사용한 녹화

때때로 특정 조건이 발생할 때 녹화를 시작하고 싶을 수 있습니다. JProfiler에는 작업 목록을 실행하는 트리거를 정의하는 시스템 [p. 122]이 있습니다. 사용 가능한 트리거 작업에는 활성 녹화의 변경도 포함됩니다.

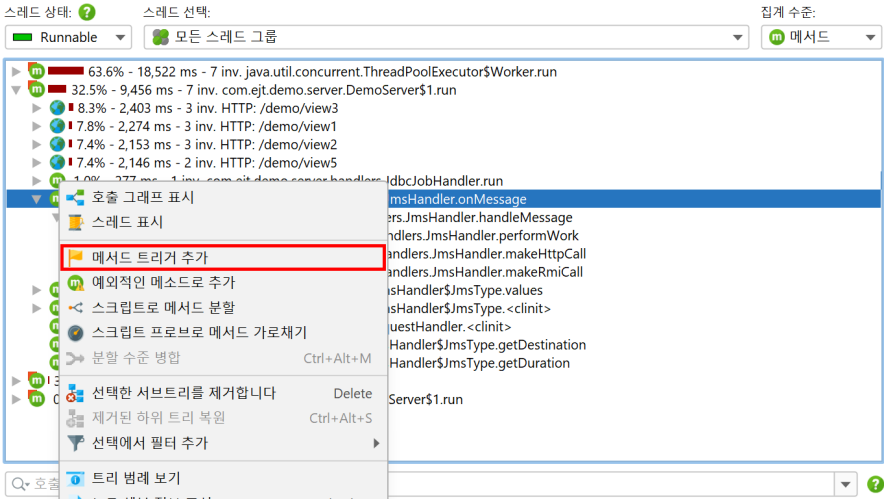
예를 들어, 특정 메서드가 실행될 때만 녹화를 시작하고 싶을 수 있습니다. 그런 경우, 세션 설정 대화 상자로 이동하여 트리거 설정 탭을 활성화하고 해당 메서드에 대한 메서드 트리거를 정의합니다. 작업 구성에서는 다양한 녹화 작업을 사용할 수 있습니다.



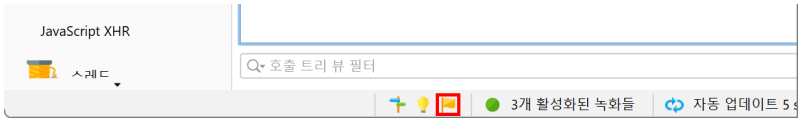
"녹화 시작" 작업은 매개변수 없이 해당 녹화를 제어합니다. 일반적으로 녹화를 중지하고 다시 시작할 때, 이전에 기록된 모든 데이터는 지워집니다. "CPU 데이터" 및 "할당 데이터" 녹화의 경우, 이전 데이터를 유지하고 여러 간격에 걸쳐 누적을 계속할 수 있는 옵션도 있습니다.



메서드 트리거는 호출 트리에서 "메서드 트리거 추가" 작업을 사용하여 편리하게 추가할 수 있습니다. 동일한 세션에 이미 메서드 트리거가 있는 경우, 기존 트리거에 메서드 인터셉션을 추가할 수 있습니다.



기본적으로, 트리거는 프로파일링을 위해 JVM이 시작될 때 활성화됩니다. 시작 시 트리거를 비활성화하는 방법은 두 가지가 있습니다: 트리거 구성에서 개별적으로 비활성화하거나 세션 시작 대화 상자의 시작 시 트리거 활성화 확인란을 선택 해제합니다. 라이브 세션 중에는 메뉴에서 프로파일링->(활성화|비활성화) 트리거를 선택하거나 상태 표시줄의 트리거 녹화 상태 아이콘을 클릭하여 모든 트리거를 활성화하거나 비활성화할 수 있습니다.



때때로, 동일한 그룹의 트리거에 대한 활성화를 동시에 전환해야 할 필요가 있습니다. 이는 관심 있는 트리거에 동일한 그룹 ID를 할당하고 메뉴에서 프로파일링->트리거 그룹 활성화를 호출하여 가능합니다.

jpcontroller를 사용한 녹화

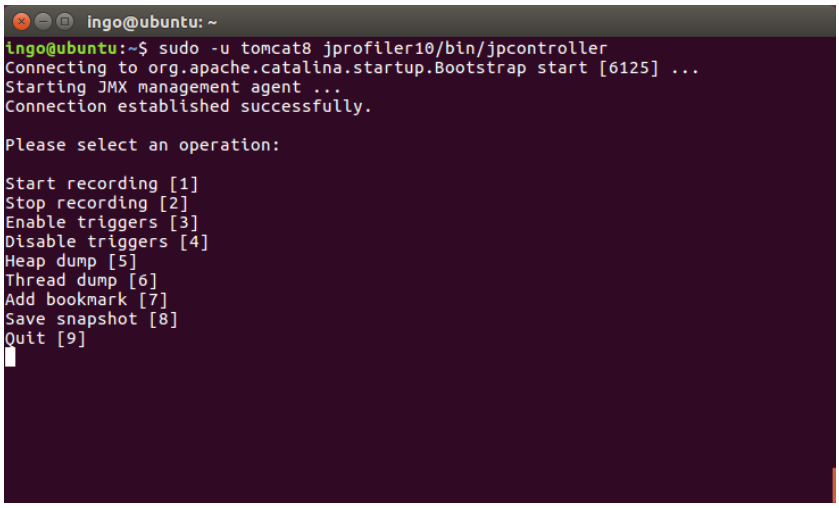
JProfiler에는 이미 프로파일링 중인 JVM의 녹화를 제어하기 위한 명령줄 실행 파일이 있습니다. `jpcontroller`는 JProfiler MBean이 게시되어야 하며, 그렇지 않으면 프로파일된 JVM에 연결할 수 없습니다. 이는 프로파일링 에이전트가 이미 프로파일링 설정을 받은 경우에만 해당됩니다. 프로파일링 설정이 없으면 에이전트는 정확히 무엇을 기록해야 하는지 알 수 없습니다.

다음 조건 중 하나가 적용되어야 합니다:

- JProfiler GUI로 이미 JVM에 연결했습니다
- 프로파일된 JVM이 `-agentpath` VM 매개변수와 함께 시작되었으며, `nowait` 및 `config` 매개변수를 포함했습니다. 통합 마법사에서는 즉시 시작 모드와 시작 시 구성 적용 옵션에 해당합니다.
- `jpenable` 실행 파일로 프로파일링을 위해 JVM이 준비되었으며, `-offline` 매개변수가 지정되었습니다. 자세한 내용은 `jpenable -help`의 출력을 참조하십시오.

특히, 프로파일된 JVM이 `nowait` 플래그로만 시작된 경우 `jpcontroller`는 작동하지 않습니다. 통합 마법사에서는 JProfiler GUI로 연결할 때 구성 적용 옵션이 구성 동기화 단계에서 이러한 매개변수를 구성합니다. 자세한 내용은 시작 시 프로파일링 설정 설정에 대한 도움말 주제 [p. 235]를 참조하십시오.

jpcontroller는 모든 녹화 및 해당 매개변수에 대한 반복적인 다단계 메뉴를 제공합니다. 이를 사용하여 스냅샷을 저장할 수도 있습니다.



프로그램 방식으로 녹화 시작

녹화를 시작하는 또 다른 방법은 API를 통해서입니다. 프로파일된 VM에서 `com.jprofiler.api.controller.Controller` 클래스를 호출하여 프로그램 방식으로 녹화를 시작하고 중지할 수 있습니다. 자세한 내용과 컨트롤러 클래스를 포함하는 아티팩트를 얻는 방법은 오프라인 프로파일링에 대한 장 [p. 122]을 참조하십시오.

다른 JVM에서 녹화를 제어하려면 `jpcontroller`에서도 사용되는 프로파일된 JVM의 동일한 MBean에 액세스할 수 있습니다. MBean의 프로그램 방식 사용을 설정하는 것은 다소 복잡하며 상당한 절차가 필요하므로, JProfiler는 재사용할 수 있는 예제를 제공합니다. 파일 `api/samples/mbean/src/MBeanProgrammaticAccessExample.java`를 확인하십시오. 이는 다른 프로파일된 JVM에서 5초 동안 CPU 데이터를 기록하고 디스크에 스냅샷을 저장합니다.

스냅샷

지금까지 우리는 JProfiler GUI가 프로파일된 JVM 내부에서 실행 중인 프로파일링 에이전트로부터 데이터를 얻는 라이브 세션만 살펴보았습니다. JProfiler는 또한 모든 프로파일링 데이터를 파일에 기록하는 스냅샷을 지원합니다. 이는 여러 시나리오에서 유리할 수 있습니다:

- 예를 들어 테스트의 일부로 프로파일링 데이터를 자동으로 기록하여 JProfiler GUI와 연결할 수 없는 경우.
- 다른 프로파일링 세션의 프로파일링 데이터를 비교하거나 이전 녹화를 보고 싶은 경우.
- 다른 사람과 프로파일링 데이터를 공유하고 싶은 경우.

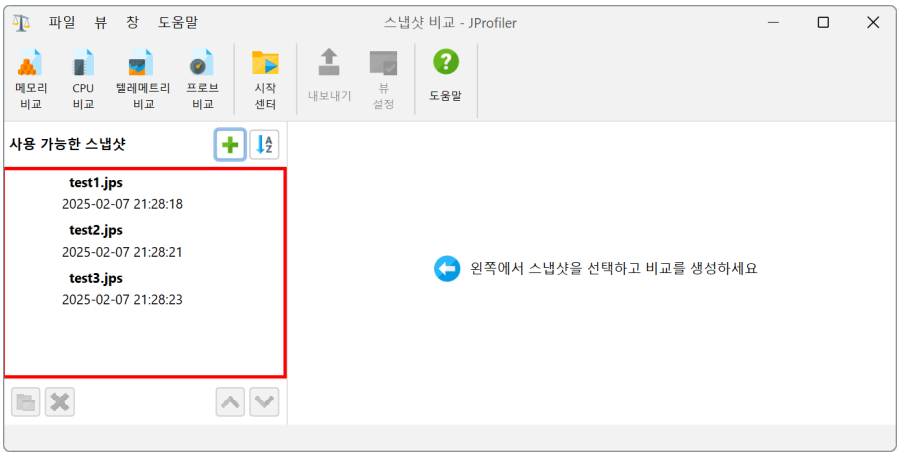
스냅샷에는 힙 스냅샷을 포함한 모든 녹화의 데이터가 포함됩니다. 디스크 공간을 절약하기 위해 스냅샷은 압축되며, 힙 워커 데이터는 직접 메모리 매핑을 허용하기 위해 압축되지 않은 상태로 유지됩니다.

JProfiler GUI에서 스냅샷 저장 및 열기

라이브 세션을 프로파일링할 때 스냅샷 저장 도구 모음 버튼으로 스냅샷을 생성할 수 있습니다. JProfiler는 원격 에이전트로부터 모든 프로파일링 데이터를 가져와 ".jps" 확장자를 가진 로컬 파일에 저장합니다. 라이브 세션 동안 여러 개의 스냅샷을 저장할 수 있습니다. 이들은 자동으로 열리지 않으며 프로파일링을 계속할 수 있습니다.

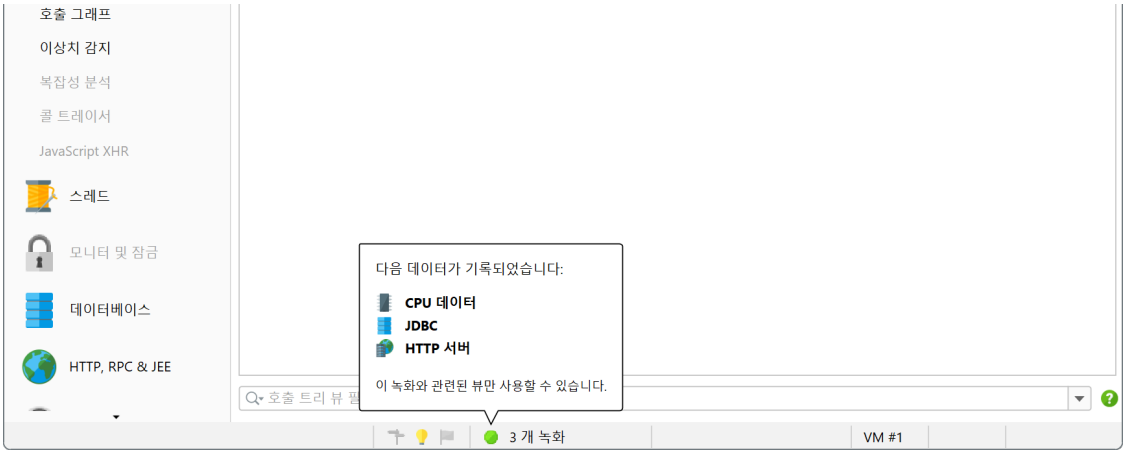
저장된 스냅샷은 파일->최근 스냅샷 메뉴에 자동으로 추가되어 방금 저장한 스냅샷을 편리하게 열 수 있습니다. 라이브 세션이 여전히 실행 중일 때 스냅샷을 열면 라이브 세션을 종료하거나 다른 JProfiler 창을 열 수 있습니다.

JProfiler에서 스냅샷 비교 기능을 사용할 때, 현재 라이브 세션에 대해 저장한 모든 스냅샷으로 목록이 채워집니다. 이를 통해 다양한 사용 사례를 쉽게 비교할 수 있습니다.



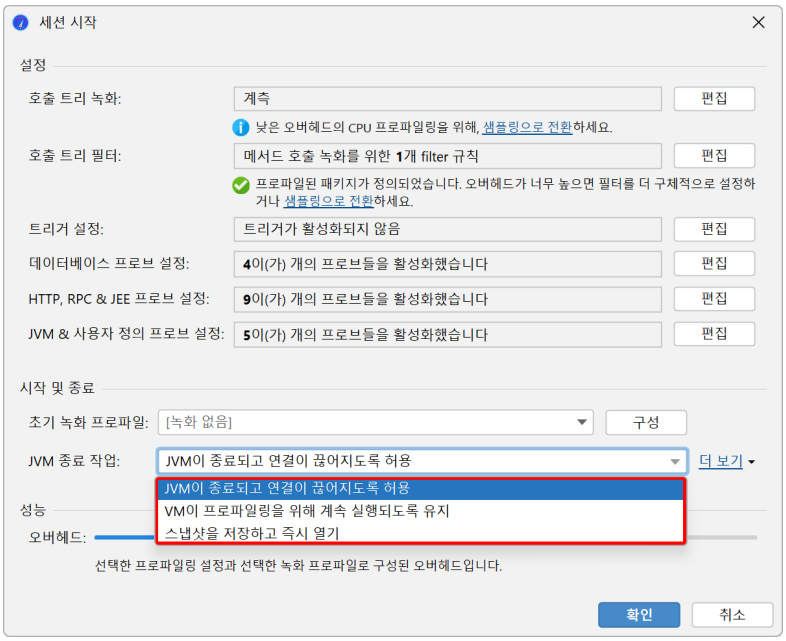
일반적으로, 메인 메뉴에서 세션->스냅샷 열기를 호출하거나 파일 관리자에서 스냅샷 파일을 더블 클릭하여 스냅샷을 열 수 있습니다. JProfiler의 IDE 통합은 IDE 자체의 일반적인 파일 열기 동작을 통해 JProfiler 스냅샷 열기를 지원합니다. 이 경우, 내장된 소스 코드 뷰어 대신 IDE로 소스 코드 탐색을 할 수 있습니다.

스냅샷을 열면 모든 녹화 작업이 비활성화되고 녹화된 데이터가 있는 뷰만 사용할 수 있습니다. 어떤 종류의 데이터가 녹화되었는지 알아보려면 상태 표시줄의 녹화 레이블 위에 마우스를 올려보세요.



단명 프로그램 프로파일링

라이브 세션의 경우, 모든 프로파일링 데이터는 프로파일된 JVM의 프로세스에 존재합니다. 따라서 프로파일된 JVM이 종료되면 JProfiler의 프로파일링 세션도 종료됩니다. JVM이 종료될 때 프로파일링을 계속하려면 세션 시작 대화 상자에서 활성화할 수 있는 두 가지 옵션이 있습니다.



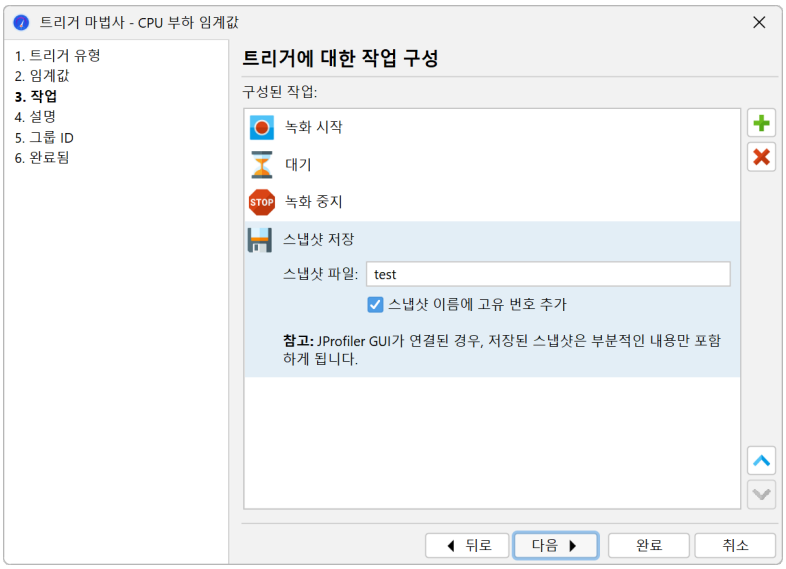
- JVM이 실제로 종료되지 않도록 하고 JProfiler GUI가 연결되어 있는 한 인위적으로 활성 상태를 유지할 수 있습니다. 이는 IDE에서 테스트 케이스를 프로파일링하고 IDE의 테스트 콘솔에서 상태와 총 시간을 보고 싶을 때 바람직하지 않을 수 있습니다.
- JVM이 종료될 때 JProfiler에게 스냅샷을 저장하고 즉시 전환하도록 요청할 수 있습니다. 스냅샷은 임시이며 세션을 닫을 때 스냅샷 저장 작업을 먼저 사용하지 않으면 삭제됩니다.

트리거로 스냅샷 저장

자동화된 프로파일링 세션의 최종 결과는 항상 스냅샷 또는 일련의 스냅샷입니다. 트리거에서는 "스냅샷 저장" 작업을 추가하여 프로파일된 JVM이 실행 중인 머신에 스냅샷을 저장할 수 있습니다. 라이브 세션 중에 트리거가 실행되면 스냅샷은 원격 머신에도 저장되며 이미 JProfiler GUI로 전송된 데이터의 일부를 포함하지 않을 수 있습니다.

트리거로 스냅샷을 저장하는 두 가지 기본 전략이 있습니다:

- 테스트 케이스의 경우, "JVM 시작" 트리거에서 녹화를 시작하고 JVM이 종료될 때 스냅샷을 저장하는 "JVM 종료" 트리거를 추가합니다.
- "CPU 부하 임계값" 트리거와 같은 예외적인 조건이나 "타이머 트리거"를 사용한 주기적인 프로파일링의 경우, 중간에 "슬립" 작업으로 데이터를 녹화한 후 스냅샷을 저장합니다.

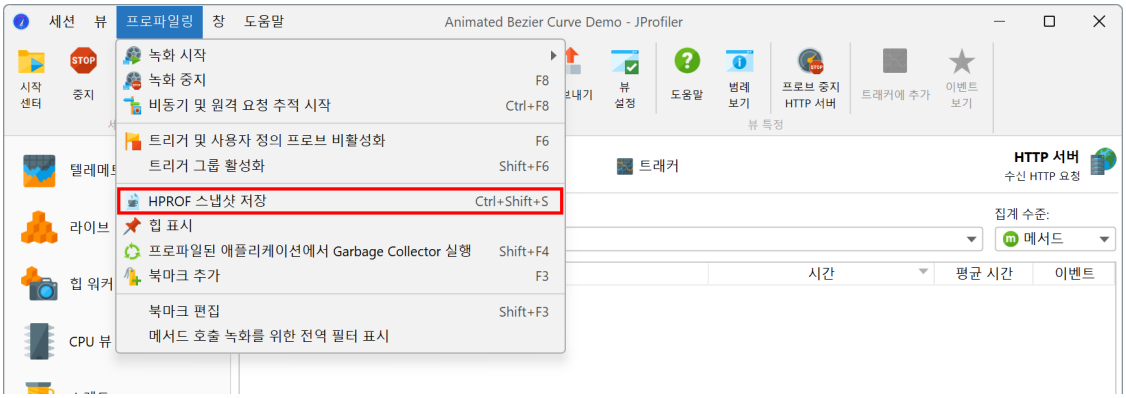


HPROF 힙 스냅샷

힙 스냅샷을 찍는 것이 너무 많은 오버헤드를 발생시키거나 메모리를 너무 많이 소비하는 상황에서는 JVM이 제공하는 내장 기능인 HPROF 힙 스냅샷을 사용할 수 있습니다. 이 작업에는 프로파일링 에이전트가 필요하지 않기 때문에, 프로덕션에서 실행 중인 JVM의 메모리 문제를 분석하는 데 흥미롭습니다.

JProfiler를 사용하면 다음과 같은 세 가지 방법으로 이러한 스냅샷을 얻을 수 있습니다:

- 라이브 세션의 경우, JProfiler GUI는 메인 메뉴에서 HPROF 힙 덤프를 트리거하는 작업을 제공합니다.

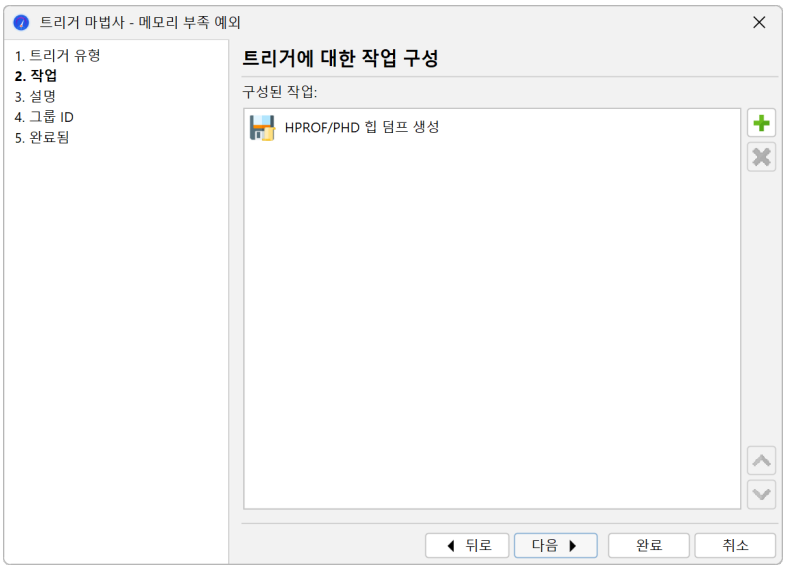


- JProfiler에는 `OutOfMemoryError`가 발생할 때 HPROF 스냅샷을 저장하는 특별한 "메모리 부족 예외" 트리거가 있습니다. 이는 HotSpot JVM이 지원하는 VM 매개변수 [(1)]

  ```
  -XX:+HeapDumpOnOutOfMemoryError
  ```

  에 해당합니다.

[(1)] http://docs.oracle.com/javase/9/troubleshoot/command-line-options1.htm#JSTGD592



- JDK의 jmap 실행 파일 [2] 을 사용하여 실행 중인 JVM에서 HPROF 힙 덤프를 추출할 수 있습니다.

  JProfiler에는 jmap보다 더 다양한 명령줄 도구인 `jpdump`이 포함되어 있습니다. 이 도구는 프로세스를 선택할 수 있게 하고, Windows에서 서비스로 실행 중인 프로세스에 연결할 수 있으며, 혼합 32비트/64비트 JVM에서도 문제가 없고 HPROF 스냅샷 파일을 자동으로 번호 매깁니다. 더 많은 정보를 얻으려면 `-help` 옵션으로 실행하세요.

JDK Flight Recorder 스냅샷

JProfile은 Java Flight Recorder (JFR)에서 저장된 스냅샷 열기를 완벽하게 지원합니다. 이 경우 UI는 상당히 다르며 JFR의 기능에 맞게 조정됩니다. 자세한 내용은 JFR 도움말 주제 [p. 209]를 참조하세요.

[2] https://docs.oracle.com/en/java/javase/11/tools/jmap.html#GUID-D2340719-82BA-4077-B0F3-2803269B7F41

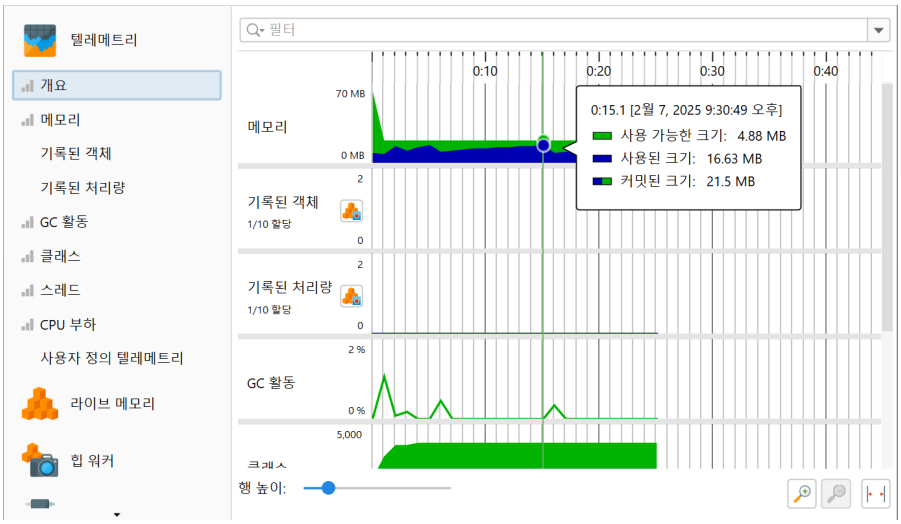
텔레메트리

프로파일링의 한 측면은 시간에 따른 스칼라 측정을 모니터링하는 것입니다. 예를 들어, 사용된 힙 크기입니다. JProfiler에서는 이러한 그래프를 텔레메트리라고 합니다. 텔레메트리를 관찰하면 프로파일된 소프트웨어에 대한 이해를 높이고, 다양한 측정치 간의 중요한 이벤트를 연관시킬 수 있으며, 예상치 못한 동작을 발견하면 JProfiler의 다른 뷰를 사용하여 더 깊은 분석을 수행하도록 유도할 수 있습니다.

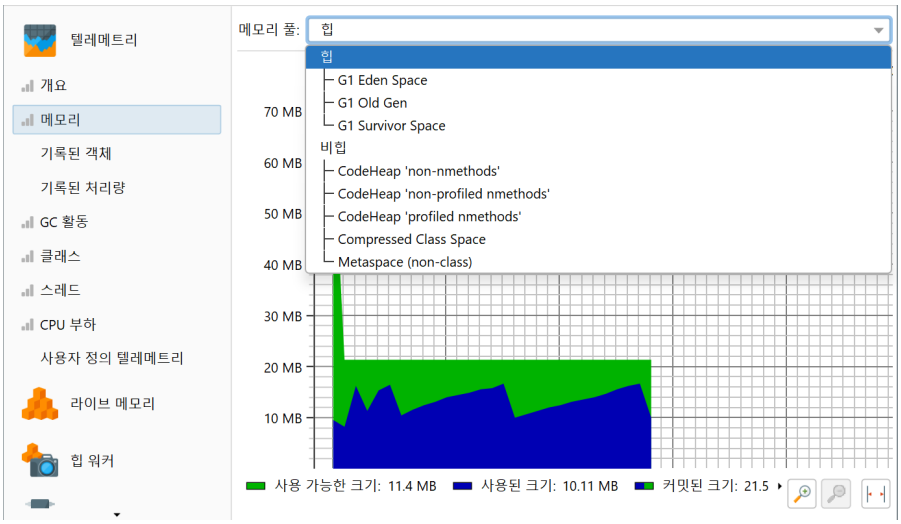
표준 텔레메트리

JProfiler UI의 "텔레메트리" 섹션에서는 기본적으로 여러 텔레메트리가 기록됩니다. 대화형 세션에서는 항상 활성화되어 있습니다. 일부 텔레메트리는 특별한 유형의 데이터가 기록되어야 합니다. 이 경우, 텔레메트리에 녹화 작업이 표시됩니다.

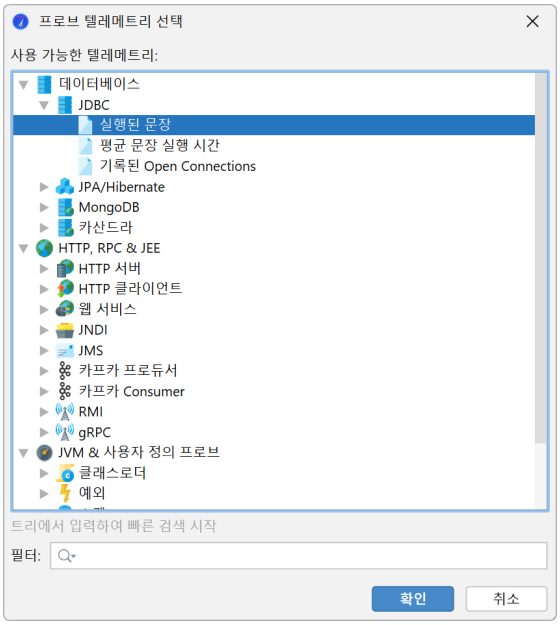
동일한 시간 축에서 여러 텔레메트리를 비교하려면, 개요에서 여러 소규모 텔레메트리를 서로 위에 표시하며, 행 높이를 구성할 수 있습니다. 텔레메트리 제목을 클릭하면 전체 텔레메트리 뷰가 활성화됩니다. 개요에서 텔레메트리의 기본 순서는 적합하지 않을 수 있습니다. 예를 들어, 선택한 텔레메트리를 나란히 연관시키고 싶을 수 있습니다. 이 경우, 개요에서 드래그 앤 드롭으로 순서를 변경할 수 있습니다.



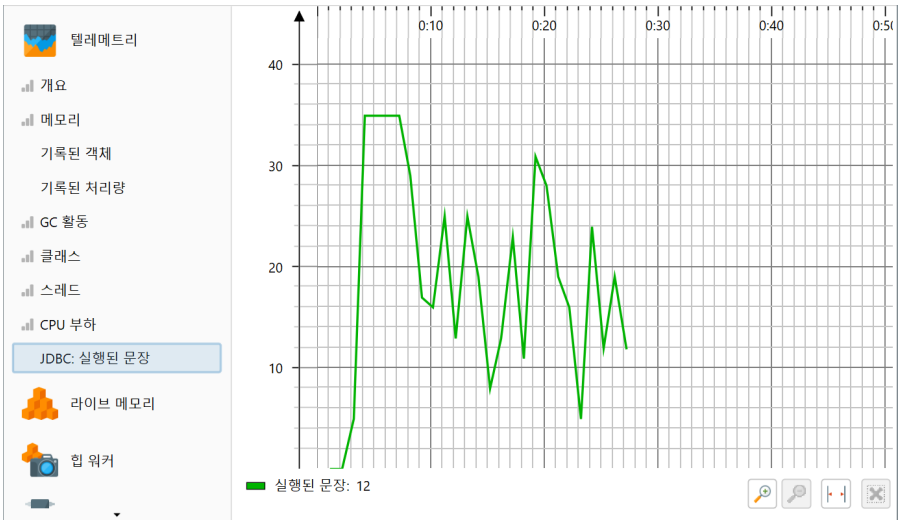
전체 뷰는 현재 값이 있는 범례를 표시하며, 개요에서 보이는 것보다 더 많은 옵션을 가질 수 있습니다. 예를 들어, "메모리" 텔레메트리는 단일 메모리 풀을 선택할 수 있습니다.



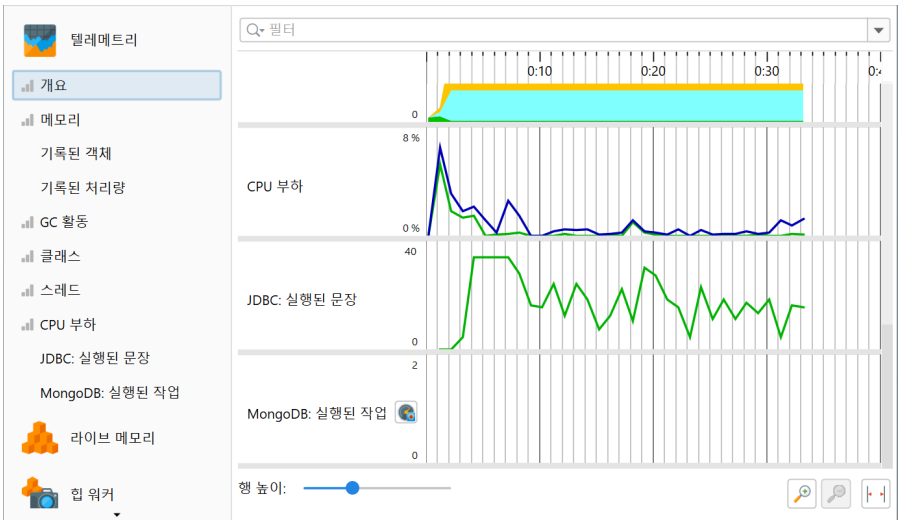
JProfiler는 JVM 및 중요한 프레임워크의 고급 시스템에서 이벤트를 기록하는 많은 프로브 [p. 98]를 가지고 있습니다. 프로브는 해당 프로브 뷰에 표시되는 텔레메트리를 가지고 있습니다. 이러한 텔레메트리를 시스템 텔레메트리와 비교하려면, 선택한 프로브 텔레메트리를 최상위 텔레메트리 섹션에 추가할 수 있습니다. 툴바에서 ✚ 텔레메트리 추가->프로브 텔레메트리 를 선택하고 하나 이상의 프로브 텔레메트리를 선택하십시오.



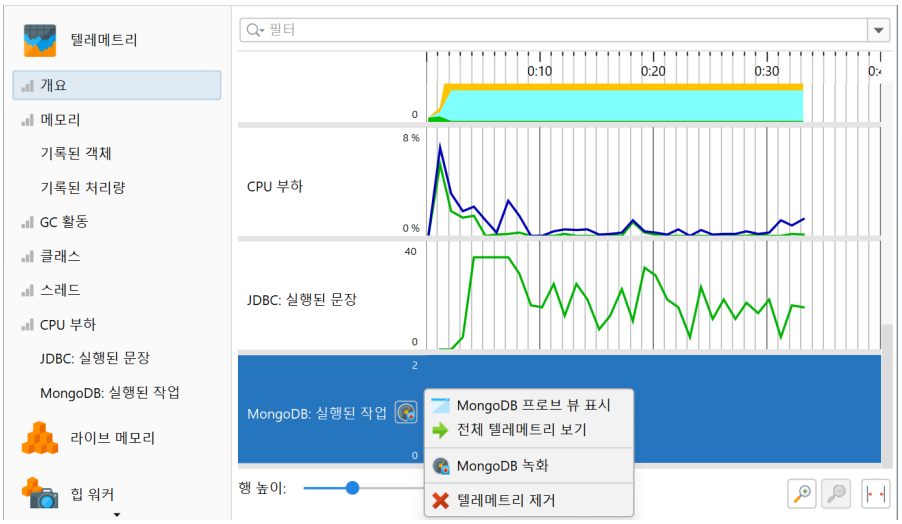
추가된 각 프로브 텔레메트리는 텔레메트리 섹션에 자체 뷰를 가지며, 개요에도 표시됩니다.



프로브 텔레메트리가 추가되면, 프로브 데이터가 기록된 경우에만 데이터가 표시됩니다. 그렇지 않으면, 텔레메트리 설명에 녹화를 시작하는 인라인 버튼이 포함됩니다.

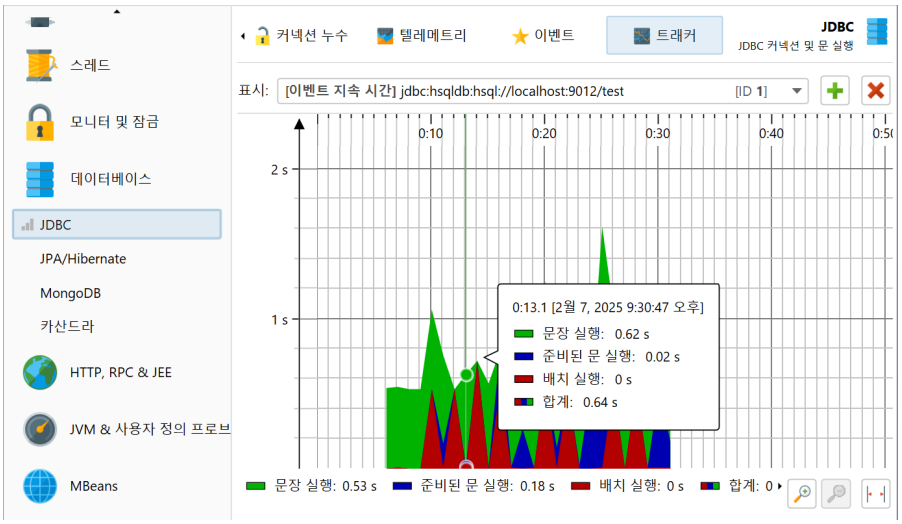


프로브 텔레메트리의 컨텍스트 메뉴에는 녹화 작업과 해당 프로브 뷰를 표시하는 작업이 포함되어 있습니다.



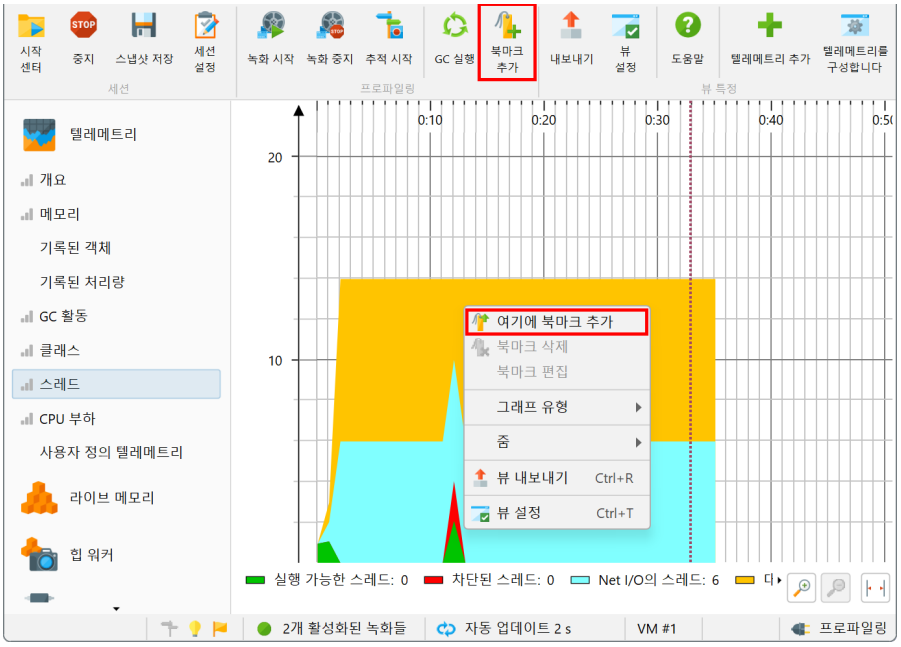
프로브 뷰와 유사하게, 기록된 객체의 VM 텔레메트리는 메모리 기록에 따라 달라지며, 녹화 버튼과 유사한 컨텍스트 메뉴를 가지고 있습니다.

마지막으로, 다른 뷰에서 선택된 스칼라 값을 모니터링하는 "추적" 텔레메트리가 있습니다. 예를 들어, 클래스 트래커 뷰에서는 클래스를 선택하고 시간에 따른 인스턴스 수를 모니터링할 수 있습니다. 또한, 각 프로브에는 선택된 핫스팟이나 제어 객체를 모니터링하는 "트래커" 뷰가 있습니다.



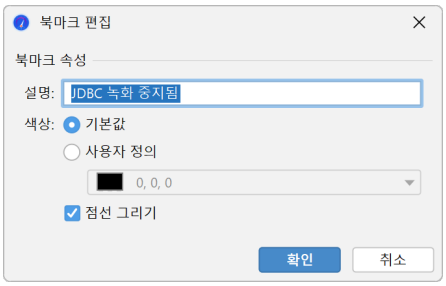
북마크

JProfiler는 모든 텔레메트리에 표시되는 북마크 목록을 유지합니다. 대화형 세션에서는 북마크 추가 툴바 버튼을 클릭하거나, 컨텍스트 메뉴의 여기에 북마크 추가 기능을 사용하여 현재 시간에 북마크를 추가할 수 있습니다.

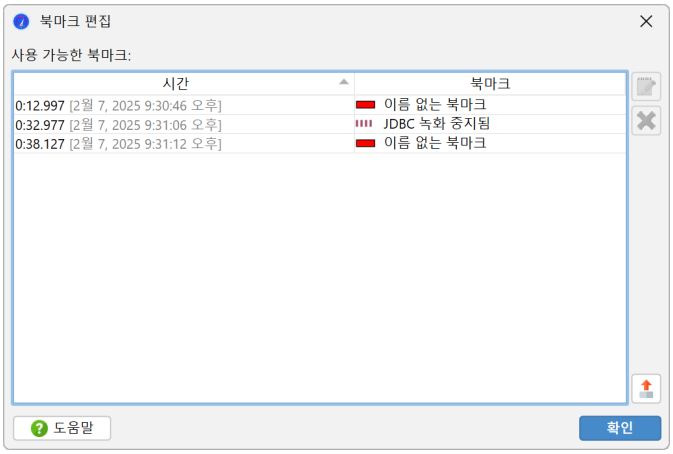


북마크는 수동으로 생성될 뿐만 아니라, 특정 녹화의 시작과 끝을 나타내기 위해 녹화 작업에 의해 자동으로 추가됩니다. 트리거 작업이나 컨트롤러 API를 사용하여 프로그래밍 방식으로 북마크를 추가할 수 있습니다.

북마크에는 색상, 선 스타일 및 툴팁에 표시되는 이름이 있습니다. 기존 북마크를 편집하고 이러한 속성을 변경할 수 있습니다.



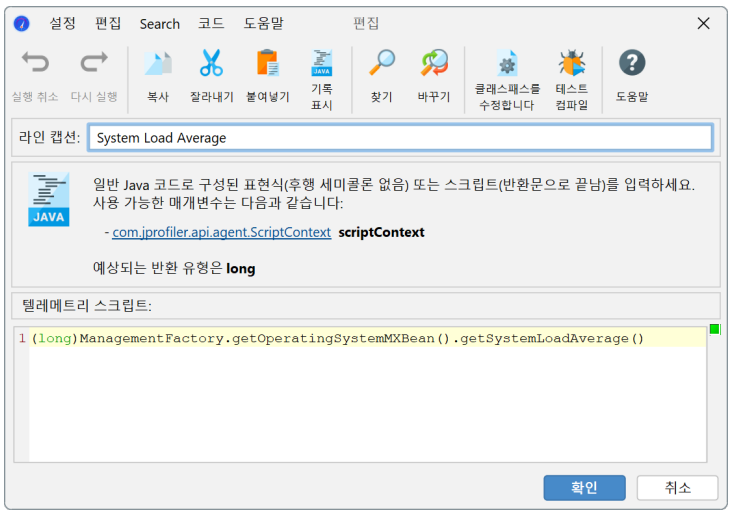
텔레메트리에서 여러 북마크를 마우스 오른쪽 버튼으로 클릭하는 것이 불편하다면, 메뉴에서 프로파일링->북마크 편집 작업을 사용하여 북마크 목록을 얻을 수 있습니다. 여기에서 북마크를 HTML 또는 CSV로 내보낼 수도 있습니다.



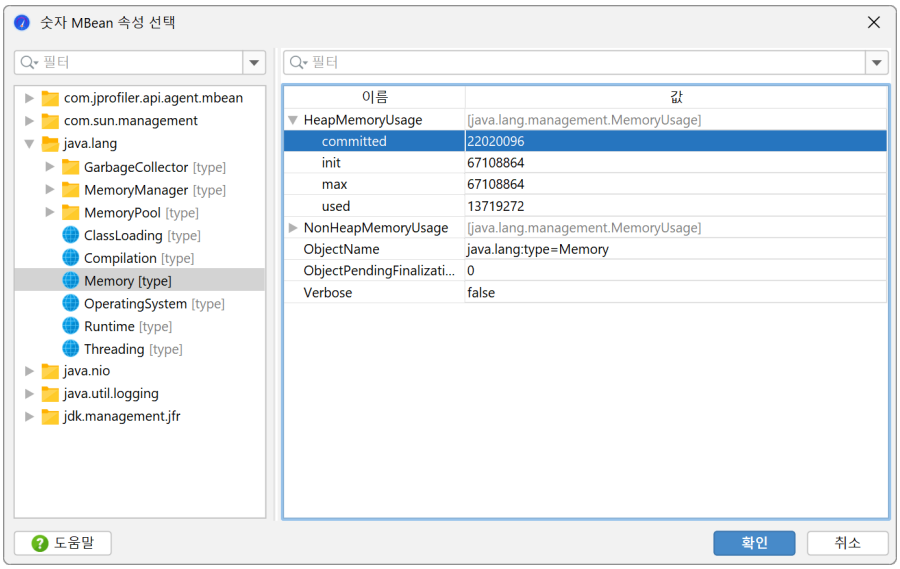
사용자 정의 텔레메트리

사용자 정의 텔레메트리를 추가하는 방법은 두 가지가 있습니다: JProfiler UI에서 스크립트를 작성하여 숫자 값을 제공하거나, 숫자 MBean 속성을 선택하는 것입니다.

사용자 정의 텔레메트리를 추가하려면, "텔레메트리" 섹션에 표시되는 텔레메트리 구성 툴바 버튼을 클릭하십시오. 스크립트 텔레메트리에서는 현재 JProfiler 세션의 클래스패스에 구성된 모든 클래스에 액세스할 수 있습니다. 값이 직접적으로 사용 가능하지 않은 경우, 이 스크립트에서 호출할 수 있는 정적 메서드를 애플리케이션에 추가하십시오.

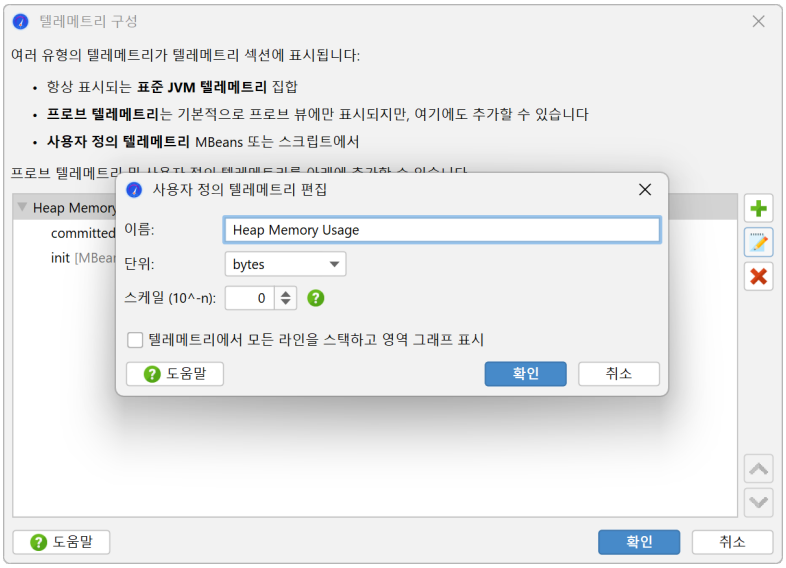


위의 예는 플랫폼 MBean에 대한 호출을 보여줍니다. MBean의 스칼라 값을 그래프로 그리는 것은 MBean 텔레메트리로 더 편리하게 수행됩니다. 여기서 MBean 브라우저를 사용하여 적절한 속성을 선택할 수 있습니다. 속성 값은 숫자여야 합니다.

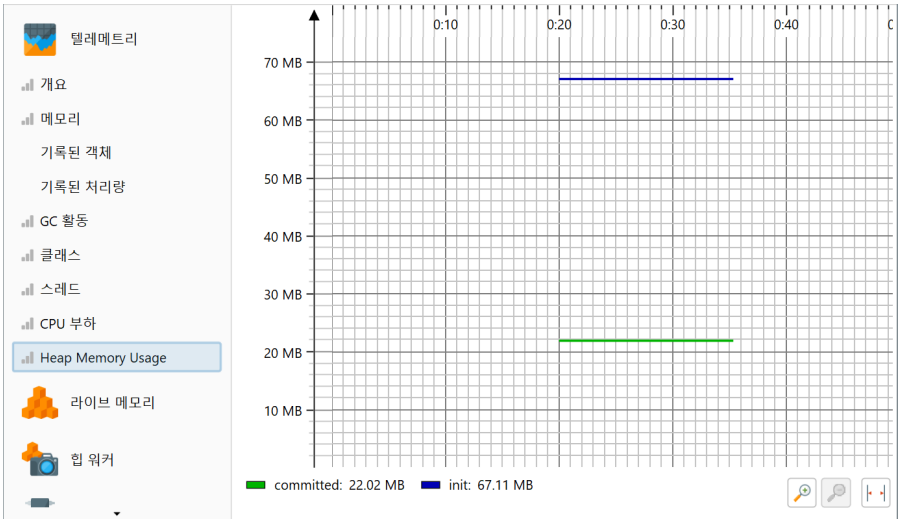


여러 텔레메트리 라인을 단일 텔레메트리로 묶을 수 있습니다. 그래서 구성이 두 부분으로 나뉩니다: 텔레메트리 자체와 텔레메트리 라인입니다. 텔레메트리 라인에서는 데이터 소스와 라인 캡션을 편집하고, 텔레메트리에서는 단위, 스케일 및 스태킹을 구성할 수 있으며, 이는 포함된 모든 라인에 적용됩니다.

스택된 텔레메트리에서는 단일 텔레메트리 라인이 추가적이며, 영역 그래프를 표시할 수 있습니다. 스케일 팩터는 값을 지원되는 단위로 변환하는 데 유용합니다. 예를 들어, 데이터 소스가 kB를 보고하는 경우, JProfiler에는 일치하는 "kB" 단위가 없습니다. 스케일 팩터를 -3으로 설정하면 값이 바이트로 변환되고, 텔레메트리의 단위로 "바이트"를 선택하면 JProfiler는 텔레메트리에서 적절한 집계 단위를 자동으로 표시합니다.



사용자 정의 텔레메트리는 "텔레메트리" 섹션의 끝에 구성된 순서대로 추가됩니다. 이를 재정렬하려면, 개요에서 원하는 위치로 드래그하십시오.



오버헤드 고려사항

처음에는 텔레메트리가 시간에 따라 메모리를 선형적으로 소비할 것처럼 보일 수 있습니다. 그러나 JProfiler는 오래된 값을 통합하고 점진적으로 더 거친 그래인으로 만들어 텔레메트리당 소비되는 총 메모리 양을 제한합니다.

텔레메트리의 CPU 오버헤드는 값이 초당 한 번만 폴링된다는 사실로 인해 제한됩니다. 표준 텔레메트리의 경우, 이 데이터 수집에 대한 추가 오버헤드는 없습니다. 사용자 정의 텔레메트리의 경우, 오버헤드는 기본 스크립트 또는 MBean에 의해 결정됩니다.

CPU 프로파일링

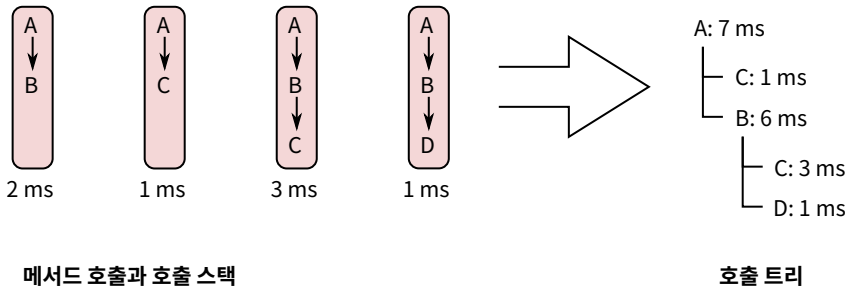
JProfiler가 메서드 호출의 실행 시간과 호출 스택을 측정할 때, 이를 "CPU 프로파일링"이라고 합니다. 이 데이터는 다양한 방식으로 제공됩니다. 해결하려는 문제에 따라, 어느 한 가지 표현이 가장 유용할 것입니다. CPU 데이터는 기본적으로 기록되지 않으며, 흥미로운 사용 사례를 캡처하려면 CPU 기록을 켜야 [p. 26] 합니다.

호출 트리

모든 메서드 호출과 호출 스택을 추적하면 상당한 양의 메모리를 소비하며, 모든 메모리가 소진될 때까지 짧은 시간 동안만 유지할 수 있습니다. 또한, 바쁜 JVM에서 메서드 호출 수를 직관적으로 파악하기 쉽지 않습니다. 일반적으로 그 수가 너무 많아 추적을 찾고 따르는 것이 불가능합니다.

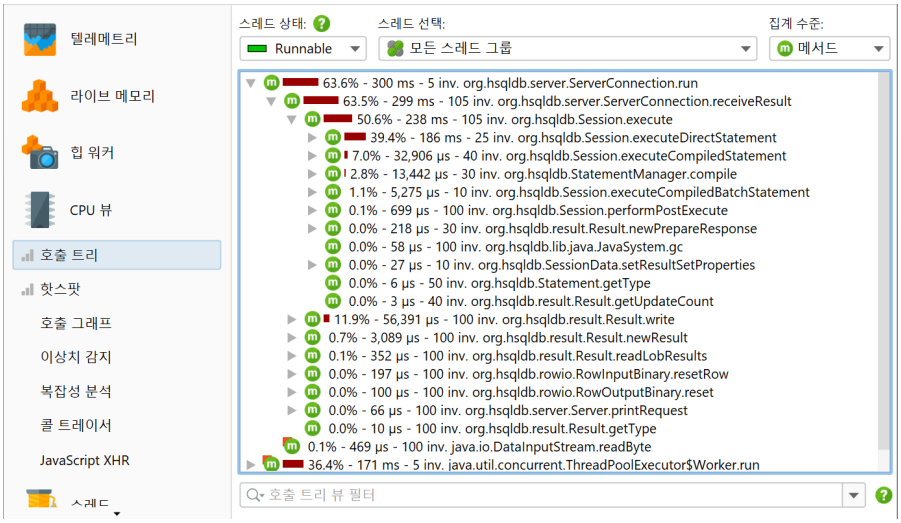
또 다른 측면은 많은 성능 문제가 수집된 데이터가 집계될 때만 명확해진다는 것입니다. 이렇게 하면 특정 시간 기간 동안 전체 활동에 대해 메서드 호출이 얼마나 중요한지 알 수 있습니다. 단일 추적으로는 보고 있는 데이터의 상대적 중요성을 알 수 없습니다.

이러한 이유로 JProfiler는 관찰된 모든 호출 스택의 누적 트리를 구축하고, 관찰된 타이밍과 호출 횟수로 주석을 달아줍니다. 시간적 측면은 제거되고 총 숫자만 유지됩니다. 트리의 각 노드는 적어도 한 번 이상 관찰된 호출 스택을 나타냅니다. 노드는 해당 호출 스택에서 본 모든 아웃고잉 호출을 나타내는 자식을 가지고 있습니다.

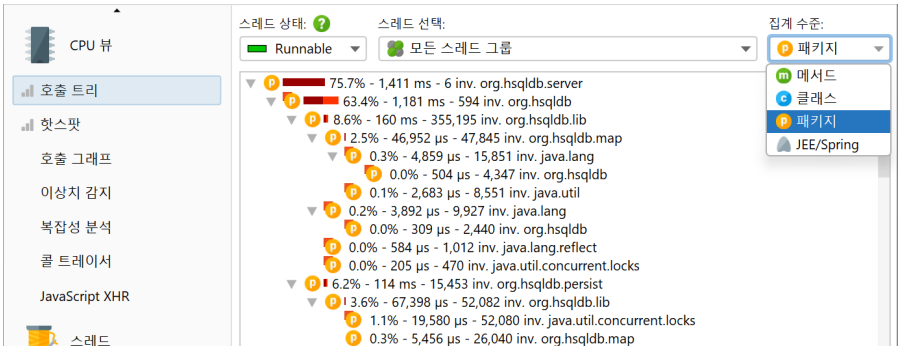


메서드 호출과 호출 스택 **호출 트리**

호출 트리는 "CPU 뷰" 섹션의 첫 번째 뷰이며, CPU 프로파일링을 시작할 때 좋은 출발점입니다. 메서드 호출을 시작점에서 가장 세분화된 끝점까지 따라가는 상향식 뷰가 가장 쉽게 이해되기 때문입니다. JProfiler는 자식을 총 시간으로 정렬하므로, 트리를 깊이 우선으로 열어 성능에 가장 큰 영향을 미치는 트리 부분을 분석할 수 있습니다.



모든 측정은 메서드에 대해 수행되지만, JProfiler는 클래스 또는 패키지 수준에서 호출 트리를 집계하여 더 넓은 관점을 제공할 수 있습니다. 집계 수준 선택기에는 "JEE/Spring 구성 요소" 모드도 포함되어 있습니다. 애플리케이션이 JEE 또는 Spring을 사용하는 경우, 이 모드를 사용하여 클래스 수준에서 JEE 및 Spring 구성 요소만 볼 수 있습니다. URL과 같은 분할 노드는 모든 집계 수준에서 유지됩니다.

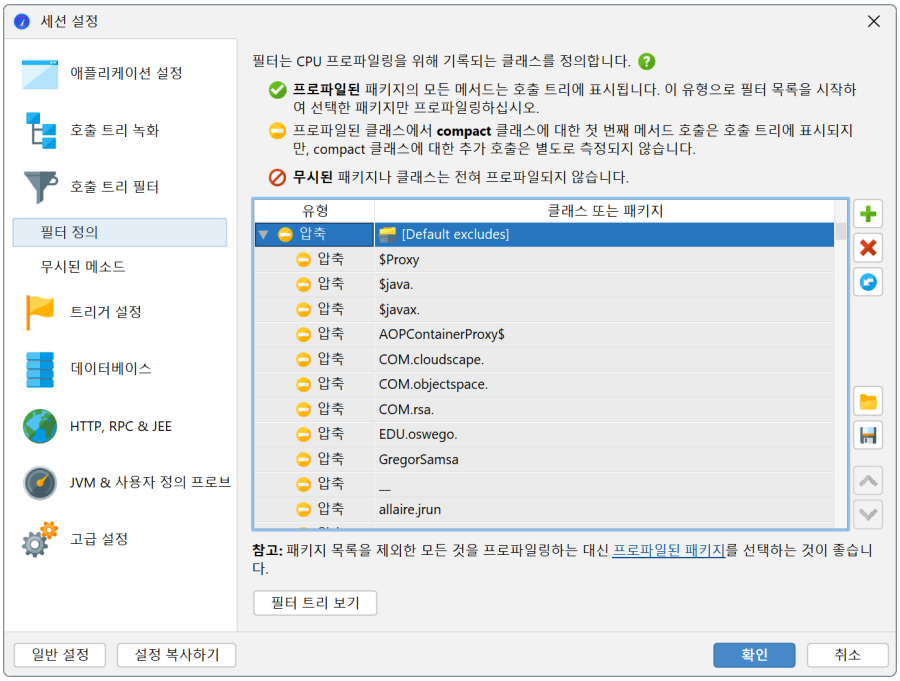


호출 트리 필터

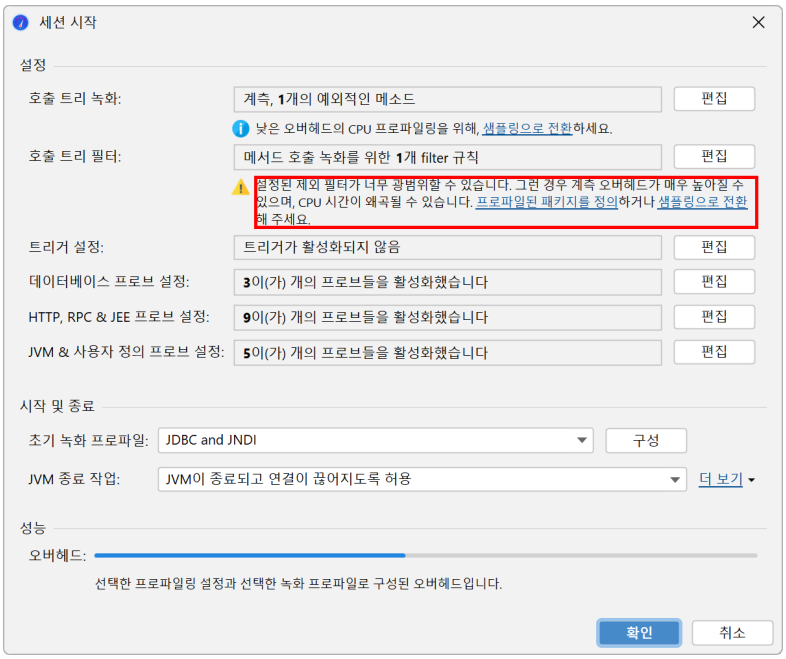
모든 클래스의 메서드가 호출 트리에 표시되면 트리가 너무 깊어 관리하기 어려울 수 있습니다. 애플리케이션이 프레임워크에 의해 호출되는 경우, 호출 트리의 상단은 관심 없는 프레임워크 클래스들로 구성되며, 자신의 클래스는 깊이 묻힐 것입니다. 라이브러리로의 호출은 내부 구조를 보여주며, 수백 개의 메서드 호출 레벨이 있을 수 있으며, 이는 익숙하지 않고 영향을 미칠 수 없는 것입니다.

이 문제의 해결책은 호출 트리에 필터를 적용하여 일부 클래스만 기록되도록 하는 것입니다. 긍정적인 부작용으로는 수집해야 할 데이터가 줄어들고, 계측해야 할 클래스가 줄어들어 오버헤드가 감소합니다.

기본적으로 프로파일링 세션은 일반적으로 사용되는 프레임워크 및 라이브러리에서 제외된 패키지 목록으로 구성됩니다.



물론 이 목록은 불완전하므로 삭제하고 관심 있는 패키지를 직접 정의하는 것이 훨씬 좋습니다. 실제로 계측 [p. 63]과 기본 필터의 조합은 매우 바람직하지 않으므로 JProfiler는 세션 시작 대화 상자에서 이를 변경할 것을 제안합니다.

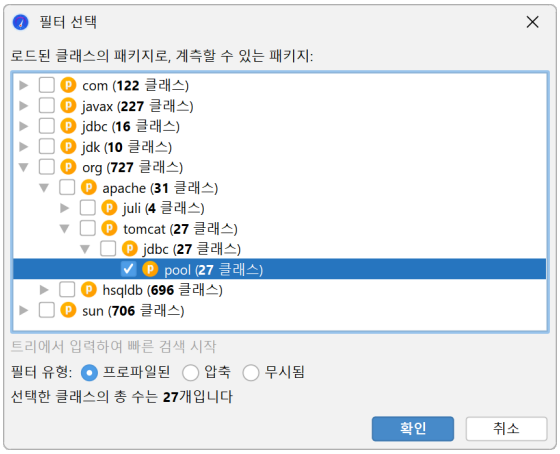


필터 표현식은 완전한 클래스 이름과 비교되므로 `com.mycorp.`는 `com.mycorp.myapp.Application`과 같은 모든 중첩 패키지의 클래스를 일치시킵니다. 필터에는 "프로파일된", "압축", "무시됨"이라는 세 가지 유형이 있습니다. "프로파일된" 클래스의 모든 메서드는 측정됩니다. 이는 자신의 코드에 필요합니다.

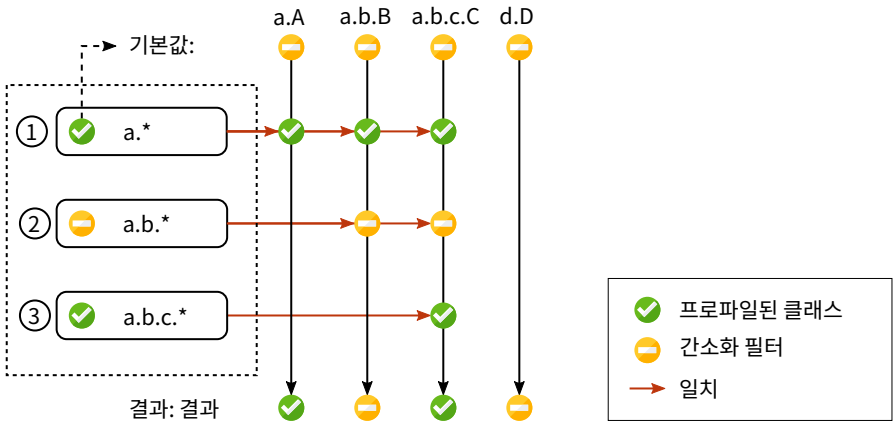
"압축" 필터에 포함된 클래스에서는 해당 클래스에 대한 첫 번째 호출만 측정되며, 추가 내부 호출은 표시되지 않습니다. "압축"은 JRE를 포함한 라이브러리에 적합합니다. 예를 들어, `hashMap.put(a, b)`를 호출할 때 호출 트리에서 `HashMap.put()`을 보고 싶지만 그 이상은 보고 싶지 않습니다. 내부 작동은 맵 구현의 개발자가 아닌 한 불투명하게 처리되어야 합니다.

마지막으로, "무시됨" 메서드는 전혀 프로파일링되지 않습니다. 오버헤드 고려 사항으로 인해 계측이 바람직하지 않을 수 있으며, 동적 호출 사이에 삽입된 내부 Groovy 메서드와 같이 호출 트리에서 단순히 방해가 될 수 있습니다.

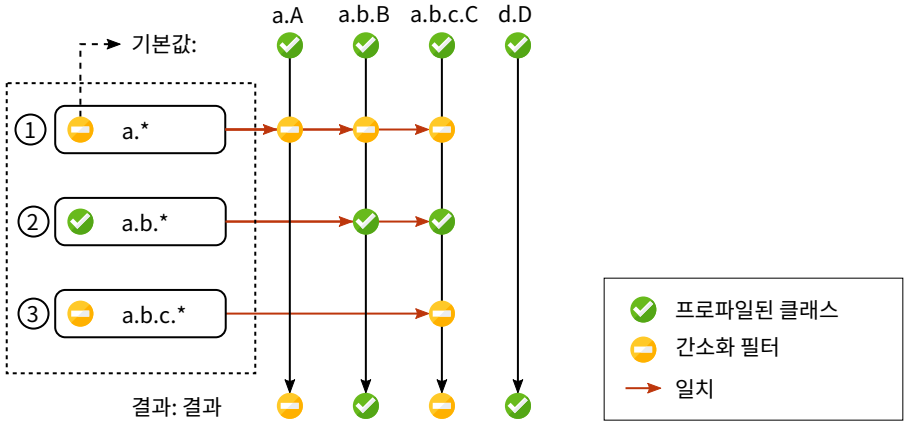
패키지를 수동으로 입력하는 것은 오류가 발생하기 쉬우므로 패키지 브라우저를 사용할 수 있습니다. 세션을 시작하기 전에 패키지 브라우저는 구성된 클래스 경로에 있는 패키지만 표시할 수 있으며, 이는 종종 실제로 로드된 모든 클래스를 포함하지 않습니다. 런타임 시 패키지 브라우저는 로드된 모든 클래스를 표시합니다.



구성된 필터 목록은 각 클래스에 대해 위에서 아래로 평가됩니다. 각 단계에서 일치하는 경우 현재 필터 유형이 변경될 수 있습니다. 필터 목록의 시작 필터 유형이 중요합니다. "프로파일된" 필터로 시작하면 클래스의 초기 필터 유형은 "압축"이며, 명시적인 일치 항목만 프로파일링됩니다.



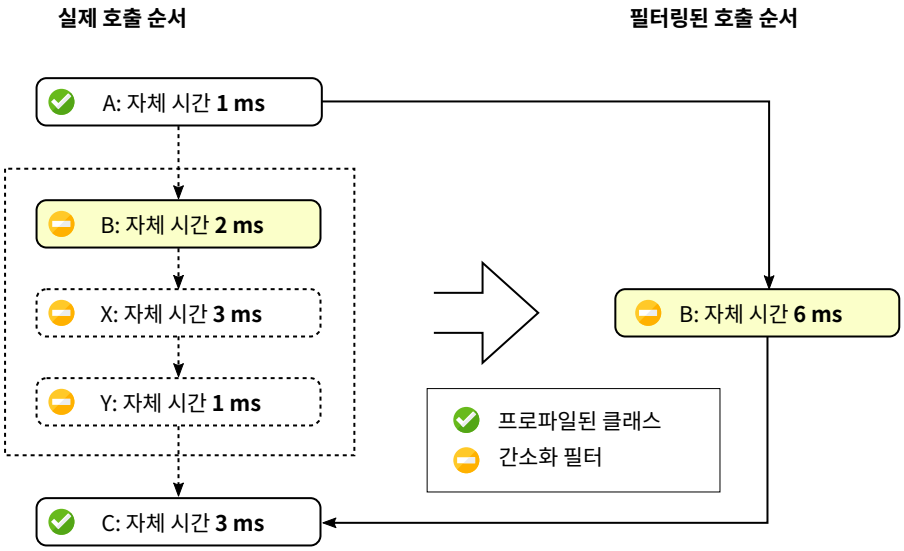
"압축" 필터로 시작하면 클래스의 초기 필터 유형은 "프로파일된"입니다. 이 경우 명시적으로 제외된 클래스를 제외하고 모든 클래스가 프로파일링됩니다.



호출 트리 시간

호출 트리를 올바르게 해석하려면 호출 트리 노드에 표시되는 숫자를 이해하는 것이 중요합니다. 모든 노드에 대해 흥미로운 두 가지 시간이 있으며, 총 시간과 자체 시간이 있습니다. 자체 시간은 노드의 총 시간에서 중첩된 노드의 총 시간을 뺀 것입니다.

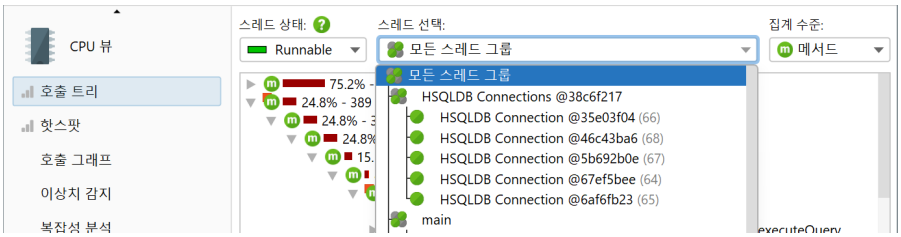
일반적으로 자체 시간은 작으며, 특히 압축 필터링된 클래스에서는 그렇습니다. 대부분의 경우 압축 필터링된 클래스는 리프 노드이며, 자식 노드가 없기 때문에 총 시간은 자체 시간과 같습니다. 때때로 압축 필터링된 클래스는 콜백을 통해 또는 호출 트리의 진입점이기 때문에 프로파일된 클래스를 호출할 수 있습니다. 이 경우 프로파일되지 않은 메서드가 시간을 소비했지만 호출 트리에는 표시되지 않습니다. 그 시간은 호출 트리에서 사용할 수 있는 첫 번째 조상 노드로 거품처럼 올라가며, 압축 필터링된 클래스의 자체 시간에 기여합니다.



호출 트리의 퍼센트 바는 총 시간을 표시하지만, 자체 시간 부분은 다른 색상으로 표시됩니다. 메서드는 동일한 레벨에서 두 메서드가 오버로드되지 않는 한 서명 없이 표시됩니다. 호출 트리 노드의 표시를 사용자 정의하는 다양한 방법이 뷰 설정 대화 상자에 있습니다. 예를 들어, 자체 시간 또는 평균 시간을 텍스트로 표시하거나, 항상 메서드 서명을 표시하거나, 사용된 시간 스케일을 변경할 수 있습니다. 또한, 퍼센트 계산은 전체 호출 트리의 시간이 아닌 부모 시간에 기반할 수 있습니다.

스레드 상태

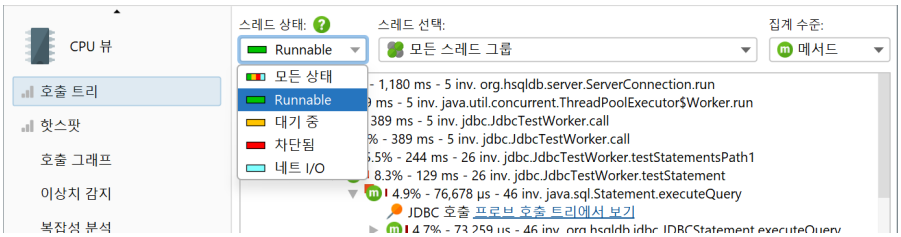
호출 트리의 상단에는 표시된 프로파일링 데이터의 유형과 범위를 변경하는 여러 뷰 매개변수가 있습니다. 기본적으로 모든 스레드는 누적됩니다. JProfiler는 스레드별로 CPU 데이터를 유지하며, 단일 스레드 또는 스레드 그룹을 표시할 수 있습니다.



모든 시간 동안 각 스레드에는 관련된 스레드 상태가 있습니다. 스레드가 바이트코드 명령을 처리할 준비가 되었거나 현재 CPU 코어에서 실행 중인 경우, 스레드 상태는 "Runnable"이라고 합니다. 이 스레드 상태는 성능 병목 현상을 찾을 때 관심이 있으므로 기본적으로 선택됩니다.

대안으로, 스레드는 `Object.wait()` 또는 `Thread.sleep()`을 호출하여 모니터를 기다리고 있을 수 있으며, 이 경우 스레드 상태는 "Waiting"이라고 합니다. `synchronized` 코드 블록의 경계에서 모니터를 획득하려고 할 때 차단된 스레드는 "Blocking" 상태에 있습니다.

마지막으로, JProfiler는 스레드가 네트워크 데이터를 기다릴 때의 시간을 추적하는 합성 "Net I/O" 상태를 추가합니다. 이는 서버 및 데이터베이스 드라이버를 분석할 때 중요합니다. 이 시간은 느린 SQL 쿼리를 조사하는 것과 같은 성능 분석에 관련될 수 있습니다.

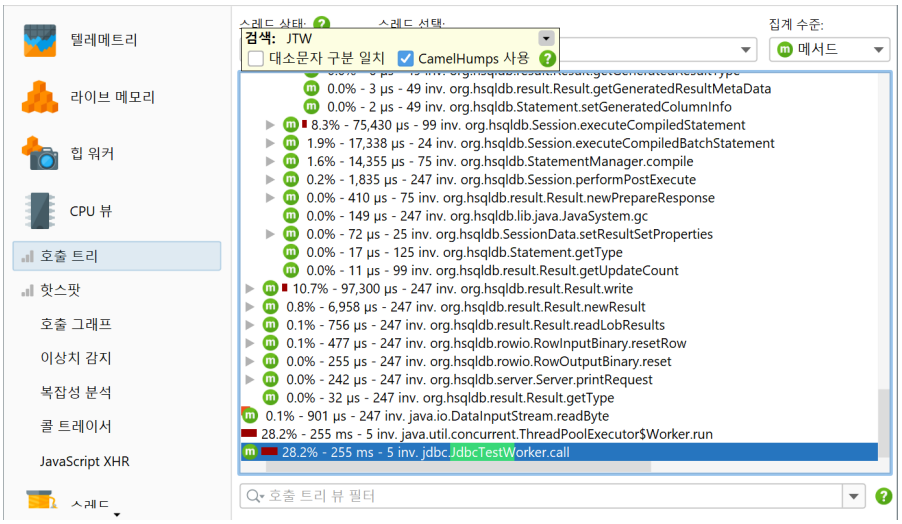


벽 시계 시간을 알고 싶다면, 스레드 상태 "모든 상태"를 선택하고 단일 스레드를 선택해야 합니다. 그래야만 코드에서 `System.currentTimeMillis()` 호출로 계산한 시간과 비교할 수 있습니다.

선택한 메서드를 다른 스레드 상태로 이동하려면 메서드 트리거와 "스레드 상태 재정의" 트리거 동작을 사용하거나 임베디드 [p. 158] 또는 인젝션된 [p. 153] 프로브 API의 `ThreadStatus` 클래스를 사용할 수 있습니다.

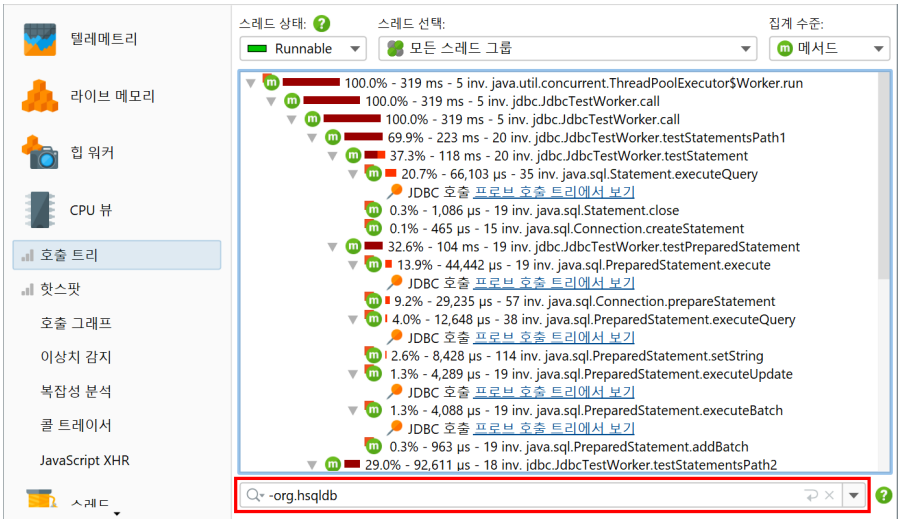
호출 트리에서 노드 찾기

호출 트리에서 텍스트를 검색하는 두 가지 방법이 있습니다. 첫째, 보기->찾기 메뉴를 호출하거나 호출 트리에서 직접 입력을 시작하여 활성화되는 빠른 검색 옵션이 있습니다. 일치 항목이 강조 표시되며, `PageDown`을 누르면 검색 옵션을 사용할 수 있습니다. `ArrowUp` 및 `ArrowDown` 키를 사용하여 다른 일치 항목을 순환할 수 있습니다.



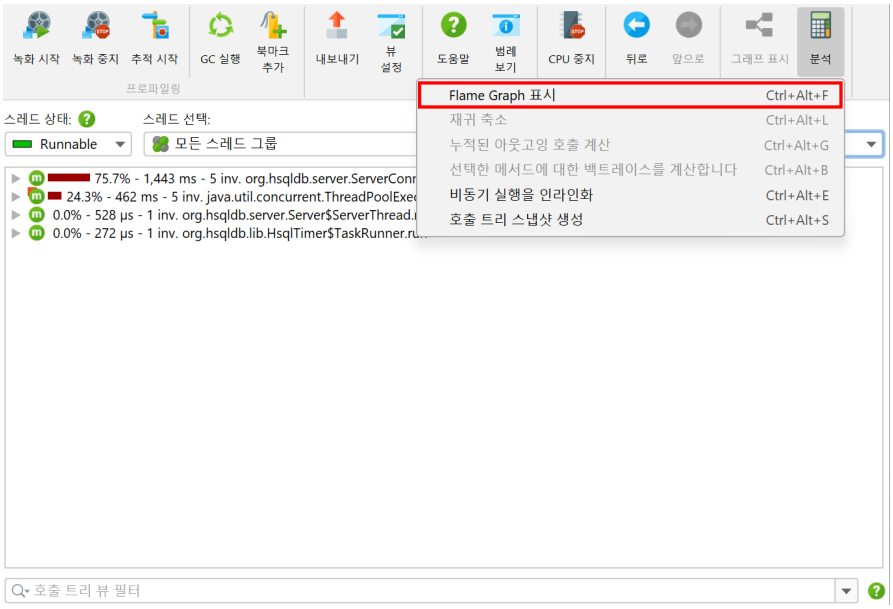
메서드, 클래스 또는 패키지를 검색하는 또 다른 방법은 호출 트리 하단의 뷰 필터를 사용하는 것입니다. 여기에서 쉼표로 구분된 필터 표현식 목록을 입력할 수 있습니다. "-"로 시작하는 필터 표현식은 무시된 필터와 같습니다. "!"로 시작하는 표현식은 압축 필터와 같습니다. 다른 모든 표현식은 프로파일된 필터와 같습니다. 필터 설정과 마찬가지로 초기 필터 유형은 클래스가 기본적으로 포함되거나 제외되는지를 결정합니다.

뷰 설정 텍스트 필드 왼쪽의 아이콘을 클릭하면 뷰 필터 옵션이 표시됩니다. 기본적으로 일치 모드는 "포함"이지만, 특정 패키지를 검색할 때 "시작"이 더 적합할 수 있습니다.

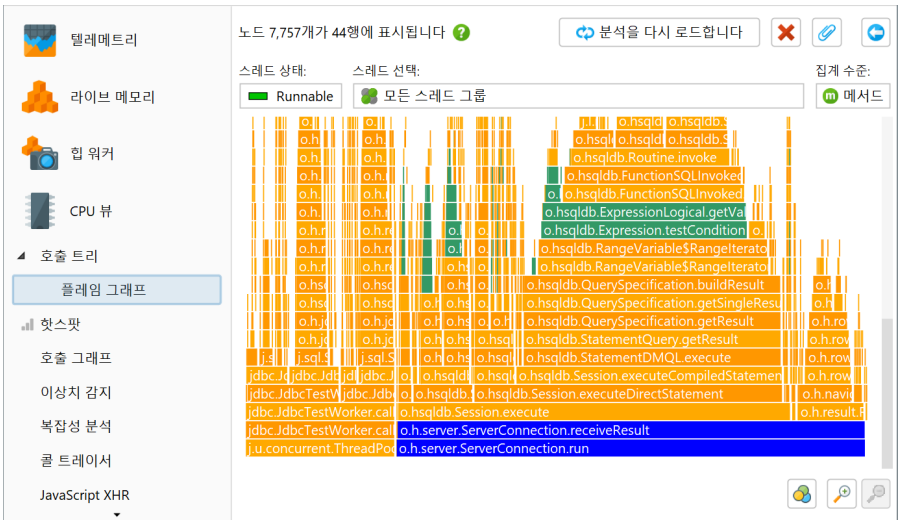


플레임 그래프

호출 트리를 보는 또 다른 방법은 플레임 그래프로 보는 것입니다. 관련 호출 트리 분석 [p. 179]을 호출하여 전체 호출 트리 또는 그 일부를 플레임 그래프로 표시할 수 있습니다.



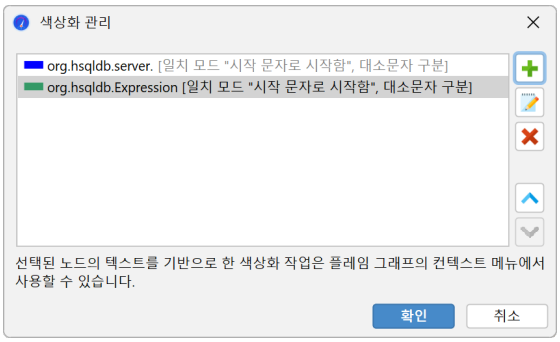
플레임 그래프는 하나의 이미지에 호출 트리의 전체 내용을 보여줍니다. 호출은 플레임 그래프의 하단에서 시작하여 상단으로 전파됩니다. 각 노드의 자식은 바로 위의 행에 배치됩니다. 자식 노드는 알파벳순으로 정렬되며 부모 노드에 중심을 맞춥니다. 각 노드에서 소비된 자체 시간으로 인해 "플레임"은 상단으로 갈수록 점점 좁아집니다. 노드에 대한 추가 정보는 도구 팁에 표시되며, 텍스트를 마크하여 클립보드에 복사할 수 있습니다.



마우스 커서 근처의 도구 팁이 분석을 방해하면, 오른쪽 상단의 버튼으로 잠글 수 있으며, 도구 팁 상단의 그리퍼로 편리한 위치로 이동할 수 있습니다. 동일한 버튼 또는 플레임 그래프를 두 번 클릭하면 도구 팁이 닫힙니다.

플레임 그래프는 정보 밀도가 매우 높으므로, 선택한 노드와 그 하위 계층 구조에 집중하여 표시된 내용을 좁힐 필요가 있을 수 있습니다. 관심 있는 영역을 확대할 수 있을 뿐만 아니라, 노드를 두 번 클릭하거나 컨텍스트 메뉴를 사용하여 새 루트 노드를 설정할 수 있습니다. 여러 번 연속으로 루트를 변경할 때, 루트의 기록에서 다시 이동할 수 있습니다.

플레임 그래프를 분석하는 또 다른 방법은 클래스 이름, 패키지 이름 또는 임의의 검색어를 기반으로 색상화를 추가하는 것입니다. 색상화는 컨텍스트 메뉴에서 추가할 수 있으며, 색상화 대화 상자에서 관리할 수 있습니다. 각 노드에 대해 첫 번째 일치하는 색상화가 사용됩니다. 색상화는 프로파일링 세션 간에 지속되며, 모든 세션 및 스냅샷에 대해 전역적으로 사용됩니다.

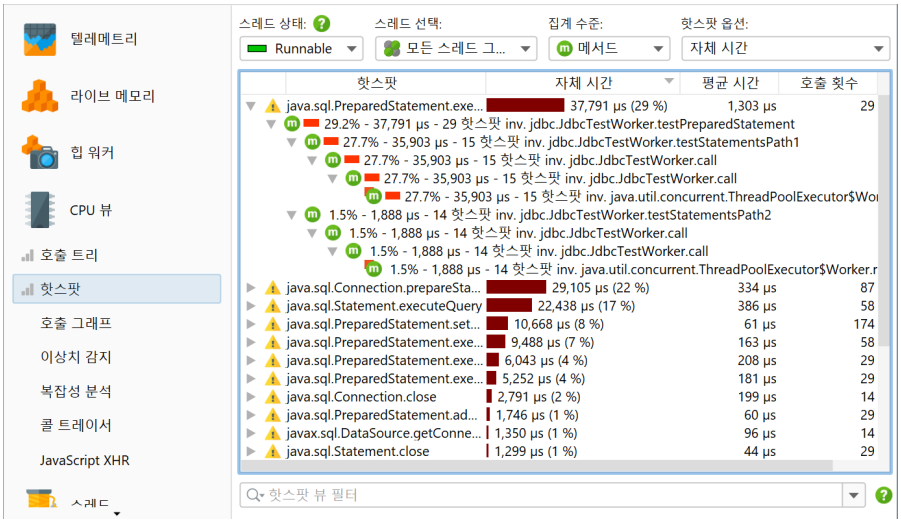


색상화 외에도, 빠른 검색 기능을 사용하여 관심 있는 노드를 찾을 수 있습니다. 커서 키를 사용하여 일치 결과를 순환할 수 있으며, 현재 강조 표시된 일치 항목에 대한 도구 팁이 표시됩니다.

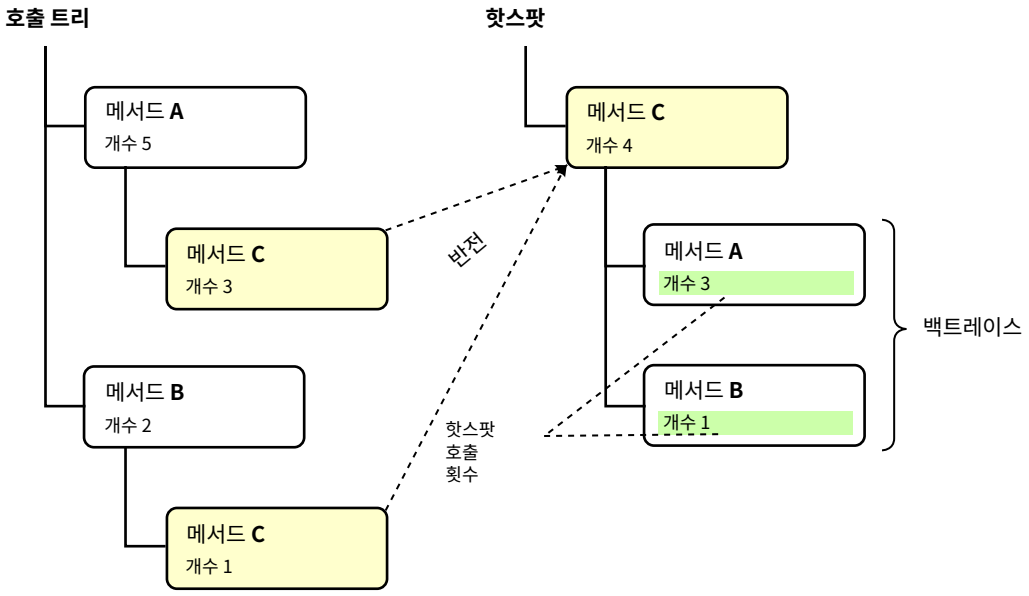
핫스팟

애플리케이션이 너무 느리게 실행되는 경우, 대부분의 시간을 소비하는 메서드를 찾고 싶을 것입니다. 호출 트리로는 이러한 메서드를 직접 찾을 수 있는 경우도 있지만, 종종 호출 트리가 넓고 리프 노드가 많아 작동하지 않을 수 있습니다.

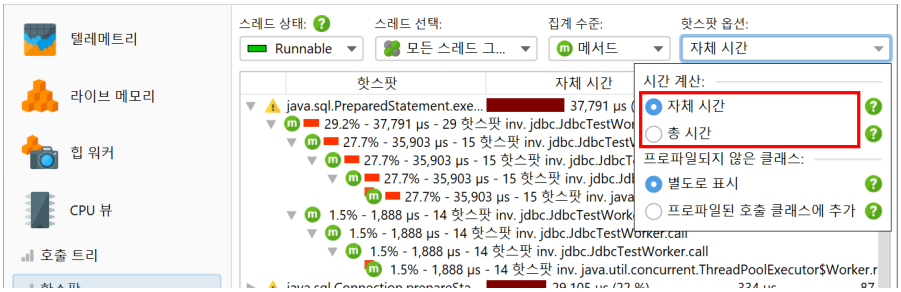
이 경우, 호출 트리의 반대가 필요합니다: 모든 메서드를 총 자체 시간으로 정렬한 목록으로, 모든 다른 호출 스택에서 누적되며, 메서드가 호출된 방법을 보여주는 백트레이스를 포함합니다. 핫스팟 트리에서는 리프가 진입점이며, 애플리케이션의 `main` 메서드나 스레드의 `run` 메서드와 같습니다. 핫스팟 트리의 가장 깊은 노드에서 호출은 최상위 노드로 위로 전파됩니다.



백트레이스의 호출 횟수와 실행 시간은 메서드 노드가 아니라, 이 경로를 따라 최상위 핫스팟 노드가 호출된 횟수를 나타냅니다. 이것은 이해하는 것이 중요합니다: 대충 보면, 노드의 정보를 해당 노드에 대한 호출을 정량화할 것으로 예상할 수 있습니다. 그러나 핫스팟 트리에서는 그 정보가 최상위 노드에 대한 노드의 **기여**를 보여줍니다. 따라서 이 반전된 호출 스택을 따라 최상위 핫스팟이 n번 호출되었고 총 지속 시간은 t초였다고 읽어야 합니다.



기본적으로 핫스팟은 자체 시간에서 계산됩니다. 총 시간에서 계산할 수도 있습니다. 이는 성능 병목 현상을 분석하는 데는 그다지 유용하지 않지만, 모든 메서드 목록을 보고 싶을 때 흥미로울 수 있습니다. 핫스팟 뷰는 오버헤드를 줄이기 위해 최대 메서드 수만 표시하므로 찾고 있는 메서드가 전혀 표시되지 않을 수 있습니다. 이 경우, 하단의 뷰 필터를 사용하여 패키지 또는 클래스를 필터링하십시오. 호출 트리와 달리 핫스팟 뷰 필터는 최상위 노드만 필터링합니다. 핫스팟 뷰의 컷오프는 전역적으로 적용되지 않으며, 표시된 클래스에 따라 적용되므로 필터를 적용한 후 새 노드가 나타날 수 있습니다.

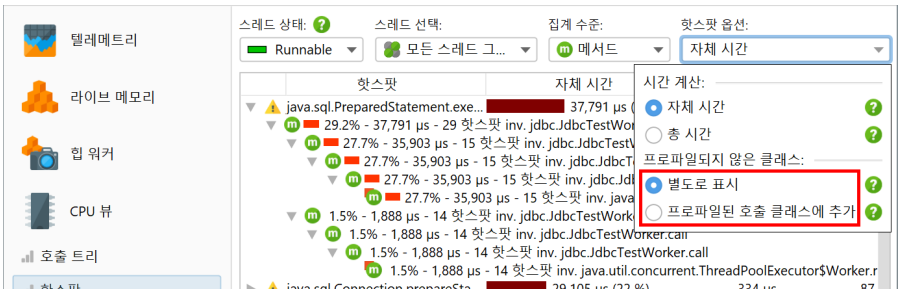


핫스팟과 필터

핫스팟의 개념은 절대적이지 않으며 호출 트리 필터에 따라 다릅니다. 호출 트리 필터가 전혀 없는 경우, 가장 큰 핫스팟은 문자열 조작, I/O 루틴 또는 컬렉션 작업과 같은 JRE의 코어 클래스의 메서드일 가능성이 큽니다. 이러한 핫스팟은 매우 유용하지 않을 것입니다. 왜냐하면 이러한 메서드의 호출을 직접 제어할 수 없으며, 속도를 높일 방법도 없기 때문입니다.

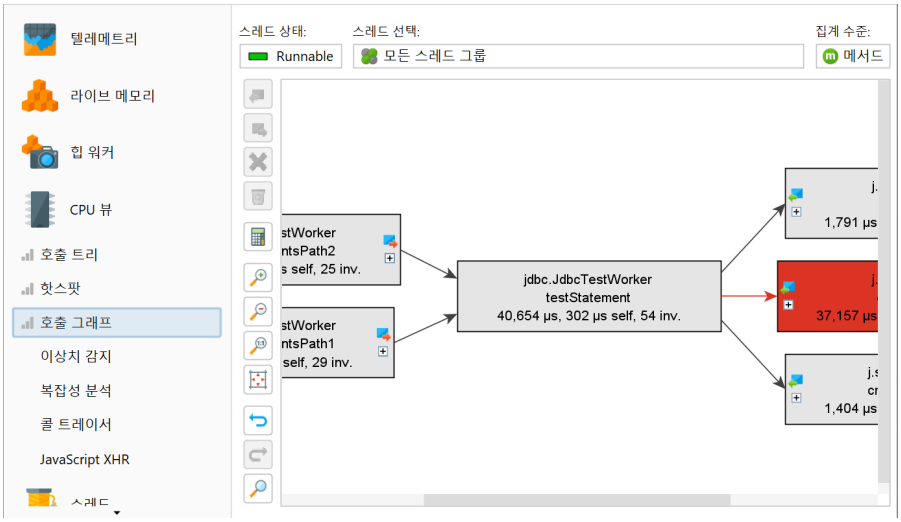
유용하려면 핫스팟은 자신의 클래스의 메서드이거나 직접 호출하는 라이브러리 클래스의 메서드여야 합니다. 호출 트리 필터의 관점에서 자신의 클래스는 "프로파일된" 필터에 있고, 라이브러리 클래스는 "압축" 필터에 있습니다.

성능 문제를 해결할 때 라이브러리 계층을 제거하고 자신의 클래스만 보고 싶을 수 있습니다. 핫스팟 옵션 팝업에서 호출된 프로파일된 클래스에 추가 라디오 버튼을 선택하여 호출 트리에서 그 관점으로 빠르게 전환할 수 있습니다.



호출 그래프

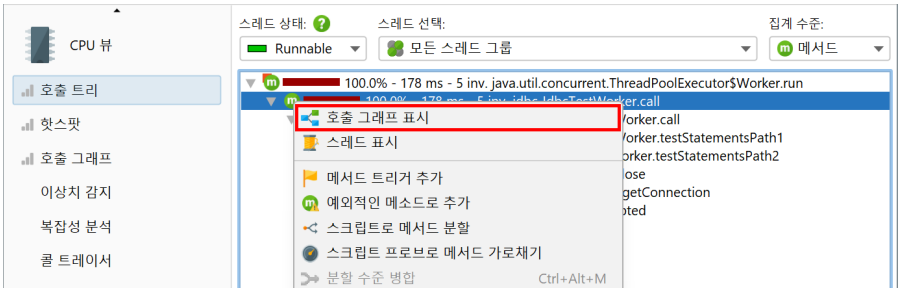
호출 트리와 핫스팟 뷰 모두에서 각 노드는 특히 재귀적으로 호출될 때 여러 번 발생할 수 있습니다. 어떤 상황에서는 각 메서드가 한 번만 발생하고 모든 인바운드 및 아웃바운드 호출이 보이는 메서드 중심의 통계에 관심이 있을 수 있습니다. 이러한 뷰는 그래프로 표시하는 것이 가장 좋으며, JProfiler에서는 호출 그래프라고 합니다.



그래프의 단점 중 하나는 시각적 밀도가 트리보다 낮다는 것입니다. 이 때문에 JProfiler는 기본적으로 패키지 이름을 축약하고, 기본적으로 총 시간의 1% 미만인 아웃고잉 호출을 숨깁니다. 노드에 아웃고잉 확장 아이콘이 있는 한, 모든 호출을 표시하기 위해 다시 클릭할 수 있습니다. 뷰 설정에서 이 임계값을 구성하고 패키지 축약을 끌 수 있습니다.

호출 그래프를 확장할 때, 특히 여러 번 백트랙할 경우 매우 빠르게 혼란스러워질 수 있습니다. 그래프의 이전 상태를 복원하려면 실행 취소 기능을 사용하십시오. 호출 트리와 마찬가지로 호출 그래프는 빠른 검색을 제공합니다. 그래프에 입력하여 검색을 시작할 수 있습니다.

그래프와 트리 뷰는 각각 장단점이 있으므로 때로는 한 뷰 유형에서 다른 뷰 유형으로 전환하고 싶을 수 있습니다. 대화형 세션에서는 호출 트리와 핫스팟 뷰가 실시간 데이터를 표시하며 주기적으로 업데이트됩니다. 그러나 호출 그래프는 요청 시 계산되며 노드를 확장할 때 변경되지 않습니다. 호출 트리의 호출 그래프에서 보기 작업은 새 호출 그래프를 계산하고 선택한 메서드를 표시합니다.



그래프에서 호출 트리로 전환하는 것은 나중에 데이터가 더 이상 비교할 수 없기 때문에 불가능합니다. 그러나 호출 그래프는 보기->분석 작업을 통해 누적된 아웃고잉 호출 및 각 선택된 노드에 대한 백트레이스를 보여주는 호출 트리 분석을 제공합니다.

기본을 넘어

호출 트리, 핫스팟 뷰 및 호출 그래프의 앙상블에는 다른 장 [p. 162]에서 자세히 설명된 많은 고급 기능이 있습니다. 또한, 별도로 [p. 184] 제공되는 다른 고급 CPU 뷰도 있습니다.

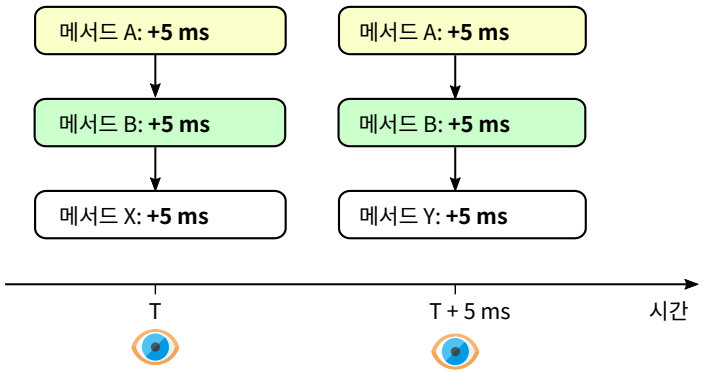
메서드 호출 녹화

메서드 호출을 녹화하는 것은 프로파일러에게 가장 어려운 작업 중 하나입니다. 왜냐하면 이는 상충되는 제약 조건 하에서 작동하기 때문입니다: 결과는 정확하고 완전해야 하며, 측정된 데이터에서 도출한 결론이 잘못되지 않도록 매우 적은 오버헤드를 생성해야 합니다. 불행히도, 모든 유형의 애플리케이션에 대해 이러한 모든 요구 사항을 충족하는 단일 측정 유형은 없습니다. 이 때문에 JProfiler는 사용할 메서드를 결정하도록 요구합니다.

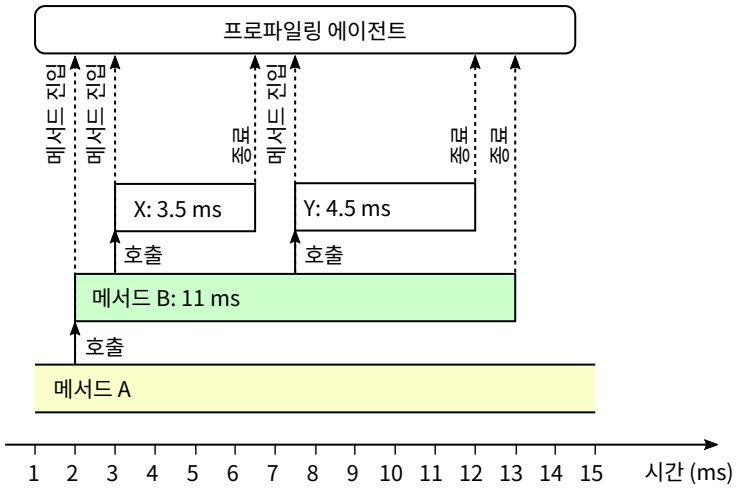
샘플링 대 계측

메서드 호출을 측정하는 것은 "샘플링"과 "계측"이라는 두 가지 근본적으로 다른 기술로 수행될 수 있으며, 각각 장점과 단점이 있습니다: 샘플링의 경우, 스레드의 현재 호출 스택이 주기적으로 검사됩니다. 계측의 경우, 선택된 클래스의 바이트코드가 수정되어 메서드 진입 및 종료를 추적합니다. 계측은 모든 호출을 측정할 수 있으며 모든 메서드에 대한 호출 횟수를 생성할 수 있습니다.

샘플링 데이터를 처리할 때, 전체 샘플링 주기(일반적으로 5ms)가 샘플링된 호출 스택에 할당됩니다. 많은 샘플이 있을 경우, 통계적으로 정확한 그림이 나타납니다. 샘플링의 장점은 드물게 발생하기 때문에 오버헤드가 매우 낮다는 것입니다. 바이트코드를 수정할 필요가 없으며, 샘플링 주기는 메서드 호출의 일반적인 지속 시간보다 훨씬 깁니다. 단점은 메서드 호출 횟수를 결정할 수 없다는 것입니다. 또한, 몇 번만 호출되는 짧은 실행 메서드는 전혀 나타나지 않을 수 있습니다. 이는 성능 병목을 찾고자 할 때는 문제가 되지 않지만, 코드의 세부적인 런타임 특성을 이해하고자 할 때는 불편할 수 있습니다.



반면에, 계측은 많은 짧은 실행 메서드가 계측될 경우 큰 오버헤드를 초래할 수 있습니다. 이 계측은 시간 측정의 고유한 오버헤드 때문에 성능 핫스팟의 상대적 중요성을 왜곡하지만, 핫스팟 컴파일러에 의해 인라인될 메서드가 이제 별도의 메서드 호출로 남아야 하기 때문이기도 합니다. 시간이 더 오래 걸리는 메서드 호출의 경우, 오버헤드는 무시할 수 있습니다. 주로 고수준 작업을 수행하는 좋은 클래스 집합을 찾을 수 있다면, 계측은 매우 낮은 오버헤드를 추가하며 샘플링보다 선호될 수 있습니다. JProfiler의 오버헤드 핫스팟 감지는 몇 번의 실행 후 상황을 개선할 수도 있습니다. 또한, 호출 횟수는 종종 중요한 정보로, 상황을 훨씬 쉽게 파악할 수 있게 해줍니다.



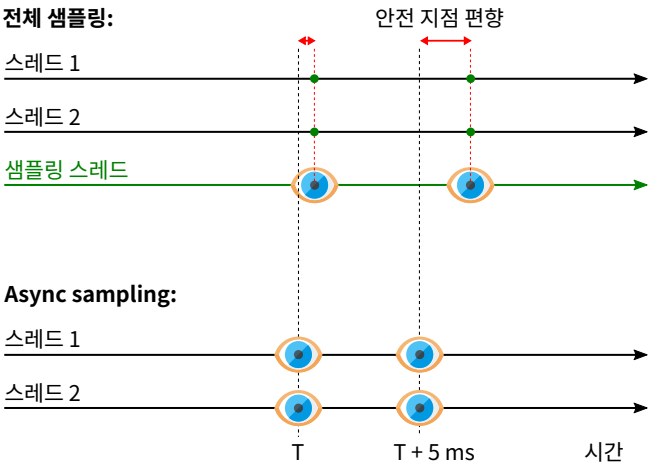
전체 샘플링 대 비동기 샘플링

JProfiler는 샘플링을 위한 두 가지 다른 기술적 솔루션을 제공합니다: "전체 샘플링"은 별도의 스레드로 수행되며, JVM의 모든 스레드를 주기적으로 일시 중지하고 스택 추적을 검사합니다. 그러나 JVM은 특정 "안전 지점"에서만 스레드를 일시 중지하므로 편향이 발생합니다. 고도로 멀티스레드 CPU 바운드 코드가 있는 경우, 프로파일된 핫스팟 분포가 왜곡될 수 있습니다. 반면에, 코드가 상당한 I/O를 수행하는 경우, 이 편향은 일반적으로 문제가 되지 않습니다.

고도로 CPU 바운드된 코드에 대한 정확한 숫자를 얻기 위해, JProfiler는 비동기 샘플링도 제공합니다. 비동기 샘플링에서는 프로파일링 신호 핸들러가 실행 중인 스레드 자체에서 호출됩니다. 프로파일링 에이전트는 네이티브 스택을 검사하고 Java 스택 프레임을 추출합니다. 주요 이점은 이 샘플링 방법에는 안전 지점 편향이 없으며, 고도로 멀티스레드 CPU 바운드 애플리케이션의 오버헤드가 낮다는 것입니다. 그러나 CPU 뷰에서는 "실행 중" 스레드 상태만 관찰할 수 있으며, "대기 중", "차단 중" 또는 "네트워크 I/O" 스레드 상태는 이 방법으로 측정할 수 없습니다. 프로브 데이터는 항상 바이트코드 계측으로 수집되므로 JDBC 및 유사한 데이터에 대한 모든 스레드 상태를 계속 얻을 수 있습니다.

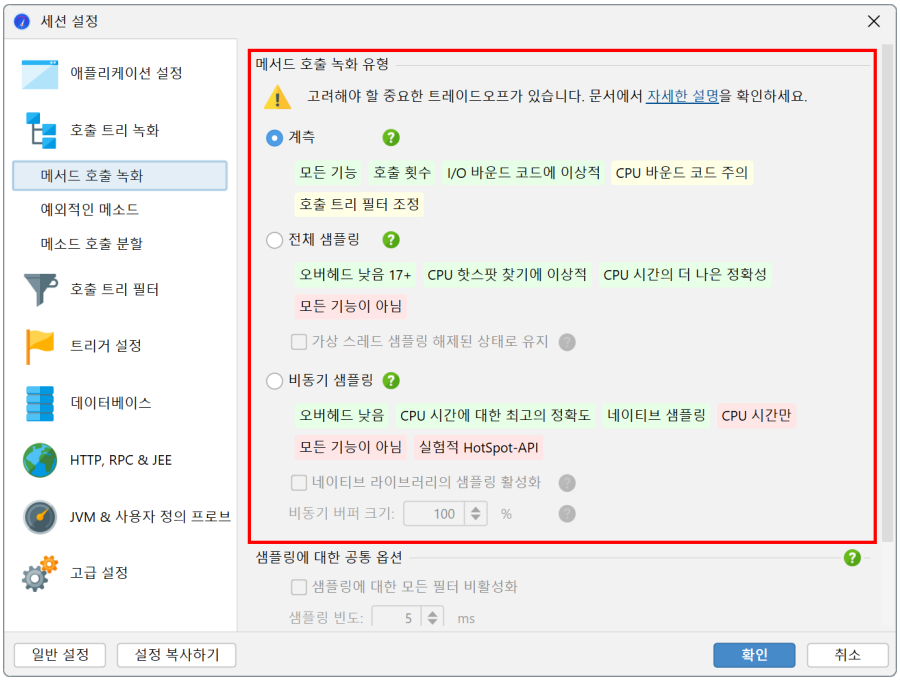
비동기 샘플링은 호출 스택의 끝부분만 사용할 수 있는 잘린 추적으로 인해 고통받습니다. 이 때문에 호출 트리는 비동기 샘플링에 대해 핫스팟 뷰만큼 유용하지 않을 수 있습니다. 비동기 샘플링은 Linux와 macOS에서만 지원됩니다.

Java 17부터 JProfiler는 Hotspot JVM에서 샘플링을 위해 전역 안전 지점을 사용하지 않고 거의 제로 오버헤드로 전체 샘플링을 수행할 수 있습니다. 비동기 샘플링과 비교할 때, 여전히 단일 스레드에 대한 안전 지점 편향을 도입하지만, JVM의 모든 스레드에 대한 전역 안전 지점의 오버헤드는 더 이상 없습니다. 비동기 샘플링의 단점을 고려할 때, Java 17+에서는 전체 샘플링을 사용하는 것이 권장됩니다.



메서드 호출 녹화 유형 선택

프로파일링을 위한 메서드 호출 녹화 유형을 선택하는 것은 중요한 결정이며 모든 상황에 맞는 올바른 선택은 없으므로 정보에 입각한 결정을 내려야 합니다. 새 세션을 생성할 때, 세션 시작 대화 상자가 사용할 메서드 호출 녹화 유형을 묻습니다. 이후 언제든지 세션 설정 대화 상자에서 메서드 호출 녹화 유형을 변경할 수 있습니다.



간단한 가이드로, 애플리케이션이 스펙트럼의 반대쪽에 있는 두 가지 명확한 범주 중 하나에 속하는지 테스트하는 다음 질문을 고려하십시오:

- **프로파일된 애플리케이션이 I/O 바운드인가요?**

  이는 대부분의 시간을 REST 서비스 및 JDBC 데이터베이스 호출을 기다리는 많은 웹 애플리케이션의 경우입니다. 이 경우, 계측은 호출 트리 필터를 신중하게 선택하여 자신의 코드만 포함하도록 하는 조건 하에 최선의 옵션이 될 것입니다.

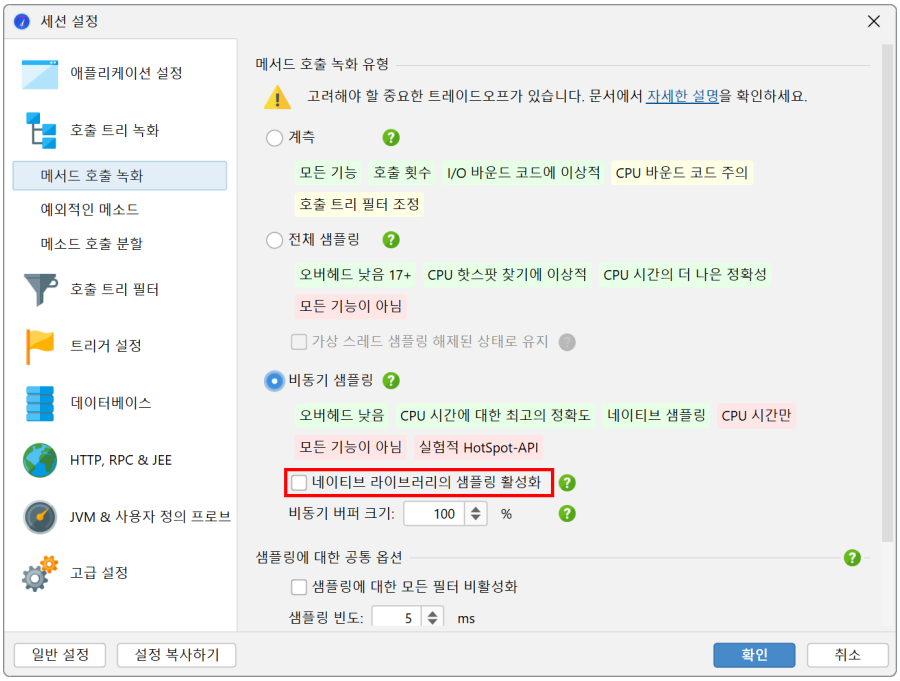
- **프로파일된 애플리케이션이 멀티스레드 및 CPU 바운드인가요?**

  예를 들어, 이는 컴파일러, 이미지 처리 애플리케이션 또는 부하 테스트를 실행하는 웹 서버의 경우일 수 있습니다. Linux 또는 macOS에서 프로파일링하는 경우, 이 경우 가장 정확한 CPU 시간을 얻기 위해 비동기 샘플링을 선택해야 합니다.

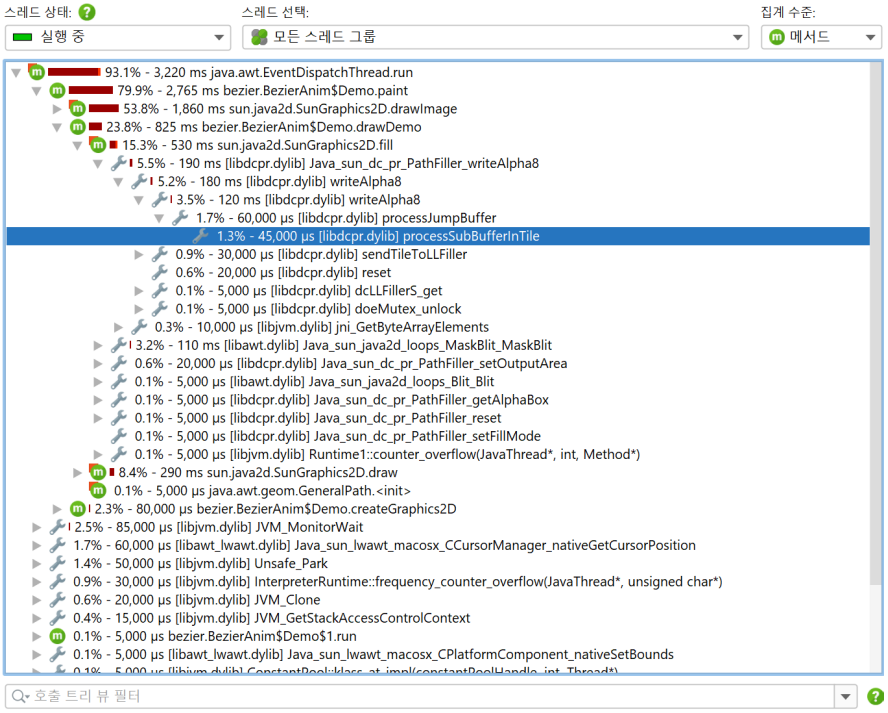
그렇지 않으면, "전체 샘플링"이 일반적으로 가장 적합한 옵션이며 새 세션의 기본값으로 제안됩니다.

네이티브 샘플링

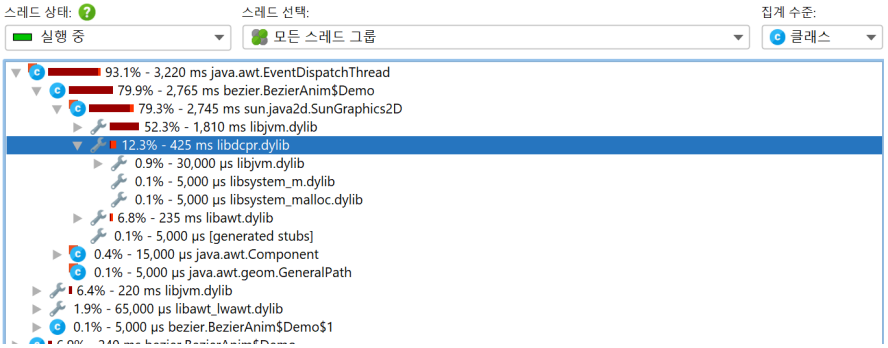
비동기 샘플링은 네이티브 스택에 접근할 수 있기 때문에 네이티브 샘플링도 수행할 수 있습니다. 기본적으로 네이티브 샘플링은 활성화되어 있지 않습니다. 왜냐하면 이는 호출 트리에 많은 노드를 도입하고 핫스팟 계산의 초점을 네이티브 코드로 이동시키기 때문입니다. 네이티브 코드에서 성능 문제가 있는 경우, 비동기 샘플링을 선택하고 세션 설정에서 네이티브 샘플링을 활성화할 수 있습니다.



JProfiler는 각 네이티브 스택 프레임에 속하는 라이브러리의 경로를 해결합니다. 호출 트리의 네이티브 메서드 노드에서 JProfiler는 네이티브 라이브러리의 파일 이름을 대괄호 안에 표시합니다.



집계 수준과 관련하여, 네이티브 라이브러리는 클래스처럼 작동하므로 "클래스" 집계 수준에서는 동일한 네이티브 라이브러리 내의 모든 후속 호출이 단일 노드로 집계됩니다. "패키지" 집계 수준은 네이티브 라이브러리와 상관없이 모든 후속 네이티브 메서드 호출을 단일 노드로 집계합니다.



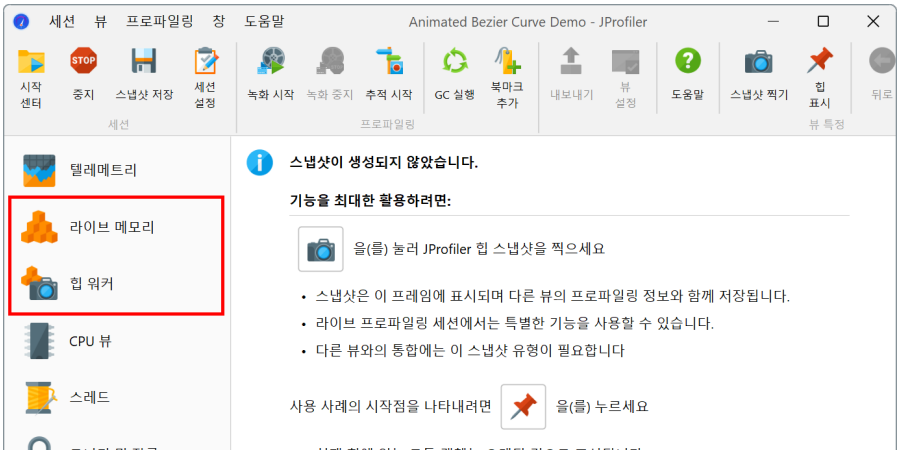
선택한 네이티브 라이브러리를 제거하려면, 해당 네이티브 라이브러리에서 노드를 제거 [p. 170]하고 전체 클래스를 제거하도록 선택할 수 있습니다.

메모리 프로파일링

힙에 있는 객체에 대한 정보를 얻는 방법은 두 가지가 있습니다. 한편으로는 프로파일링 에이전트가 각 객체의 할당 및 가비지 수집을 추적할 수 있습니다. JProfiler에서는 이를 "할당 녹화"라고 합니다. 이는 객체가 어디에서 할당되었는지를 알려주며, 임시 객체에 대한 통계를 생성하는 데에도 사용할 수 있습니다. 다른 한편으로는 JVM의 프로파일링 인터페이스가 프로파일링 에이전트가 모든 활성 객체와 그 참조를 함께 검사하기 위해 "힙 스냅샷"을 찍을 수 있게 합니다. 이 정보는 객체가 왜 가비지 수집되지 않는지를 이해하는 데 필요합니다.

할당 녹화와 힙 스냅샷 모두 비용이 많이 드는 작업입니다. 할당 녹화는 런타임 특성에 큰 영향을 미치는데, 이는 `java.lang.Object` 생성자를 계측해야 하고 가비지 수집기가 프로파일링 인터페이스에 지속적으로 보고해야 하기 때문입니다. 이 때문에 할당은 기본적으로 녹화되지 않으며, 녹화를 시작하고 중지 [p. 26]해야 합니다. 힙 스냅샷을 찍는 것은 일회성 작업입니다. 그러나 이는 JVM을 몇 초 동안 멈출 수 있으며, 획득한 데이터를 분석하는 데 상대적으로 오랜 시간이 걸릴 수 있으며, 이는 힙의 크기에 따라 달라집니다.

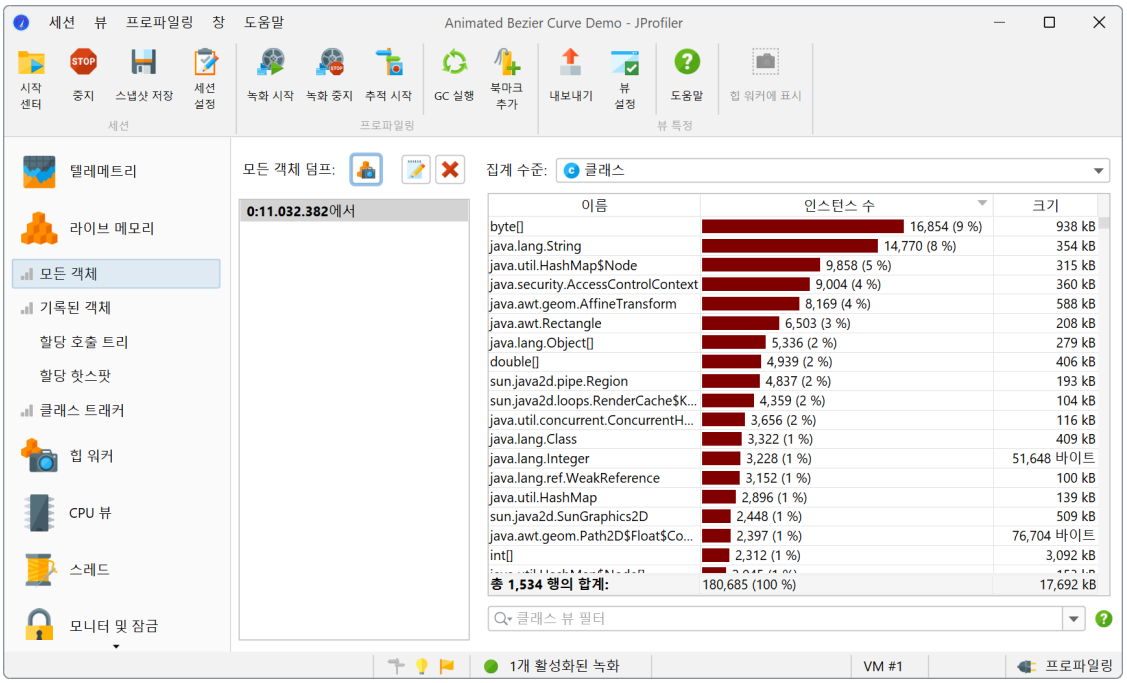
JProfiler는 메모리 분석을 두 개의 뷰 섹션으로 나눕니다: "라이브 메모리" 섹션은 주기적으로 업데이트될 수 있는 데이터를 제공하며, "힙 워커" 섹션은 정적 힙 스냅샷을 보여줍니다. 할당 녹화는 "라이브 메모리" 섹션에서 제어되지만, 녹화된 데이터는 힙 워커에서도 표시됩니다.



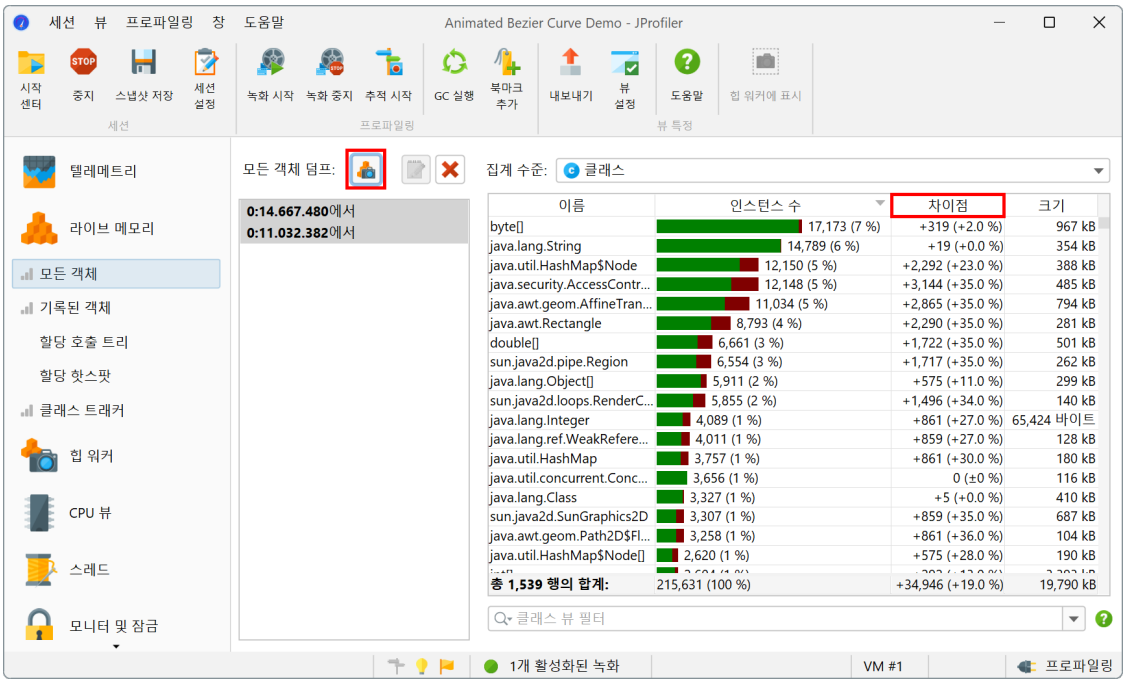
메모리 프로파일링으로 해결할 수 있는 세 가지 가장 일반적인 문제는 다음과 같습니다: 메모리 누수 찾기 [p. 203], 메모리 소비 줄이기 및 임시 객체 생성 줄이기. 첫 번째 두 문제의 경우, 주로 힙 워커를 사용하여 JVM에서 가장 큰 객체를 누가 보유하고 있는지와 그것들이 어디에서 생성되었는지를 주로 살펴봅니다. 마지막 문제의 경우, 이미 가비지 수집된 객체를 포함하기 때문에 녹화된 할당을 보여주는 라이브 뷰에만 의존할 수 있습니다.

인스턴스 수 추적

힙에 어떤 객체가 있는지 개요를 얻으려면, "모든 객체" 뷰가 모든 클래스와 그 인스턴스 수의 히스토그램을 보여줍니다. 이 뷰에 표시되는 데이터는 할당 녹화로 수집되는 것이 아니라 인스턴스 수만 계산하는 미니 힙 스냅샷을 수행하여 수집됩니다. 힙이 클수록 이 작업을 수행하는 데 시간이 더 오래 걸리므로, 이 뷰는 현재 값으로 자동 업데이트되지 않습니다.



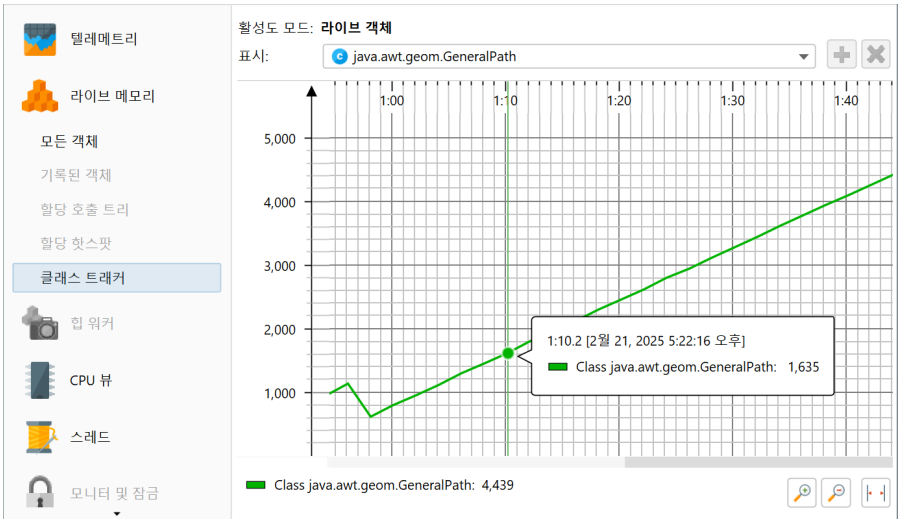
메모리 누수를 찾을 때, 종종 인스턴스 수를 시간에 따라 비교하고 싶습니다. 모든 클래스에 대해 그렇게 하려면, 뷰의 차이 기능을 사용할 수 있습니다. 모든 객체의 두 덤프가 동시에 선택되면, 차이 열이 삽입되고 인스턴스 수의 히스토그램은 마킹 시점의 기준선 값을 녹색으로 표시합니다. 모든 객체의 새 덤프를 찍을 때, 가장 오래된 선택된 덤프는 선택된 상태로 남아 있으며, 모든 객체의 새 덤프와의 차이가 표시됩니다.



덤프 선택기는 덤프가 찍힌 시간 스탬프를 보여줍니다. 덤프를 더블 클릭하여 더 쉽게 식별할 수 있도록 레이블을 추가할 수 있습니다. 모든 객체의 덤프는 트리거 액션 [p. 26] 또는 Controller API로도 트리거할 수 있으며, 여기서도 레이블을 지정할 수 있습니다.

반면, "녹화된 객체" 뷰는 할당 녹화를 시작한 후에 할당된 객체의 인스턴스 수만 보여줍니다. 할당 녹화를 중지하면 새로운 할당은 추가되지 않지만, 가비지 수집은 계속 추적됩니다. 이렇게 하면 특정 사용 사례에 대해 힙에 남아 있는 객체를 볼 수 있습니다. 객체가 오랫동안 가비지 수집되지 않을 수 있음을 유의하십시오. GC 실행 도구 모음 버튼을 사용하여 이 프로세스를 가속화할 수 있습니다. 대부분의 동적으로 업데이트되는 뷰와 마찬가지로, 동결 도구 모음 버튼이 표시된 데이터를 업데이트하는 것을 중지하기 위해 제공됩니다.

현재 마크 도구 모음 버튼을 사용하여 "녹화된 객체" 뷰에서도 선택된 기준선에 대한 차이 열을 표시할 수 있습니다. 선택된 클래스에 대해, 컨텍스트 메뉴에서 선택 항목을 클래스 트래커에 추가 액션을 사용하여 시간에 따른 그래프를 표시할 수도 있습니다.

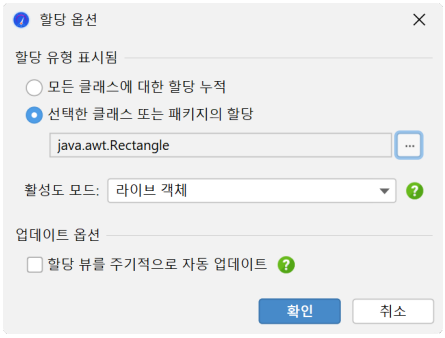


할당 지점

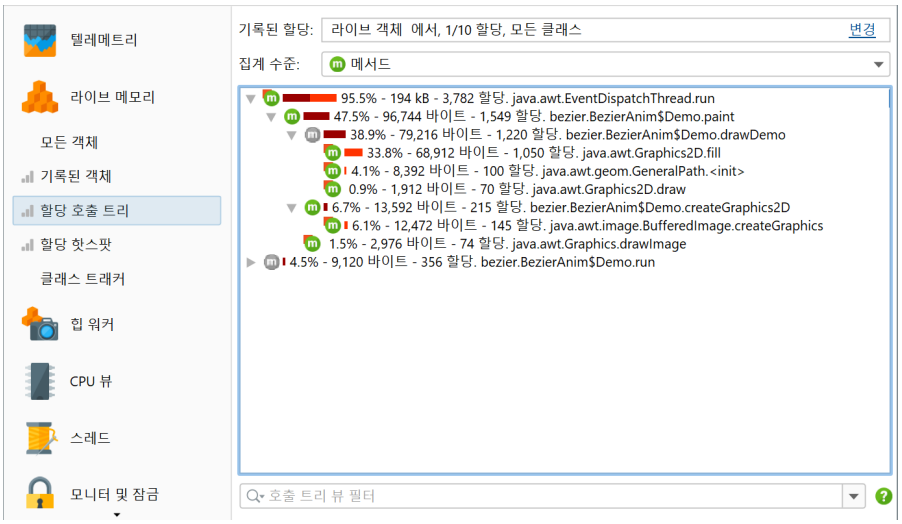
할당 녹화가 활성화되면, JProfiler는 객체가 할당될 때마다 호출 스택을 기록합니다. 스택-워킹 API와 같은 정확한 호출 스택을 사용하지 않는데, 이는 비용이 너무 많이 들기 때문입니다. 대신, CPU 프로파일링에 대해 구성된 것과 동일한 메커니즘이 사용됩니다. 이는 호출 스택이 호출 트리 필터 [p. 51]에 따라 필터링되며, 실제 할당 지점이 호출 스택에 존재하지 않는 메서드에 있을 수 있음을 의미합니다. 이는 무시되거나 압축-필터링된 클래스에서 비롯된 것입니다. 그러나 이러한 변경 사항은 직관적으로 이해하기 쉽습니다: 압축-필터링된 메서드는 압축-필터링된 클래스에 대한 추가 호출에서 수행되는 모든 할당에 대해 책임이 있습니다.

샘플링을 사용하는 경우, 할당 지점이 대략적이 되어 혼란스러울 수 있습니다. 시간 측정과 달리, 특정 클래스가 어디에서 할당될 수 있고 어디에서 할당될 수 없는지에 대한 명확한 아이디어가 있는 경우가 많습니다. 샘플링은 통계적 그림을 그리므로, `java.util.HashMap.get`이 자신의 클래스를 할당하는 것과 같은 불가능해 보이는 할당 지점을 볼 수 있습니다. 정확한 숫자와 호출 스택이 중요한 분석의 경우, 계측과 함께 할당 녹화를 사용하는 것이 좋습니다.

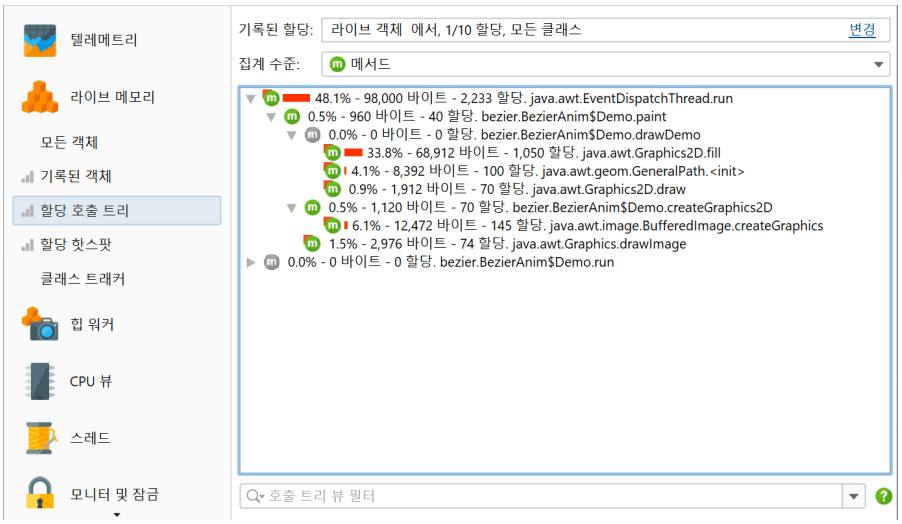
CPU 프로파일링과 마찬가지로, 할당 호출 스택은 호출 트리로 표시되며, 호출 수와 시간이 아닌 할당 수와 할당된 메모리로 표시됩니다. CPU 호출 트리와 달리, 할당 호출 트리는 자동으로 표시 및 업데이트되지 않는데, 이는 트리 계산이 더 비용이 많이 들기 때문입니다. JProfiler는 모든 객체뿐만 아니라 선택된 클래스나 패키지에 대해서도 할당 트리를 보여줄 수 있습니다. 다른 옵션과 함께, 이는 현재 데이터에서 할당 트리를 계산하도록 JProfiler에 요청한 후 표시되는 옵션 대화 상자에서 구성됩니다.



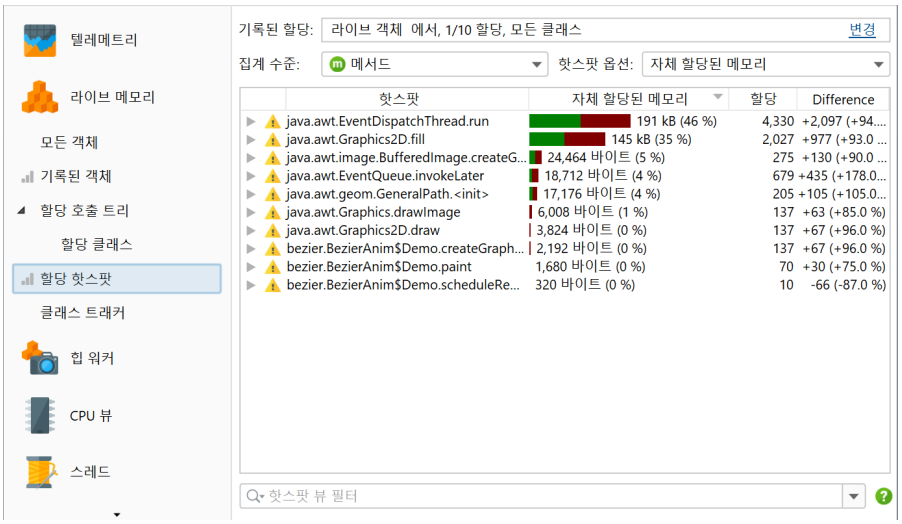
CPU 호출 트리의 유용한 속성은 각 노드가 자식 노드에서 소비된 시간을 포함하기 때문에 위에서 아래로 누적된 시간을 따라갈 수 있다는 것입니다. 기본적으로 할당 트리는 동일한 방식으로 작동하며, 각 노드는 자식 노드에 의해 수행된 할당을 포함합니다. 할당이 호출 트리 깊숙한 곳의 리프 노드에서만 수행되더라도, 숫자는 위로 전파됩니다. 이 방식으로, 할당 호출 트리의 가지를 열 때 조사할 가치가 있는 경로를 항상 볼 수 있습니다. "자체 할당"은 실제로 노드에 의해 수행된 것이며, 자손에 의해 수행된 것이 아닙니다. CPU 호출 트리와 마찬가지로, 퍼센티지 막대는 다른 색상으로 표시됩니다.



할당 호출 트리에서는 특히 선택된 클래스에 대한 할당을 표시할 때 할당이 수행되지 않는 많은 노드가 자주 있습니다. 이러한 노드는 실제 할당이 이루어진 노드로 이어지는 호출 스택을 보여주기 위해 존재합니다. 이러한 노드는 JProfiler에서 "브리지" 노드라고 하며, 위의 스크린샷에서 볼 수 있듯이 회색 아이콘으로 표시됩니다. 어떤 경우에는 할당의 누적이 방해가 될 수 있으며, 실제 할당 지점만 보고 싶을 수 있습니다. 할당 트리의 뷰 설정 대화 상자는 그 목적을 위해 누적되지 않은 숫자를 표시하는 옵션을 제공합니다. 활성화되면, 브리지 노드는 항상 0 할당을 표시하고 퍼센티지 막대가 없습니다.

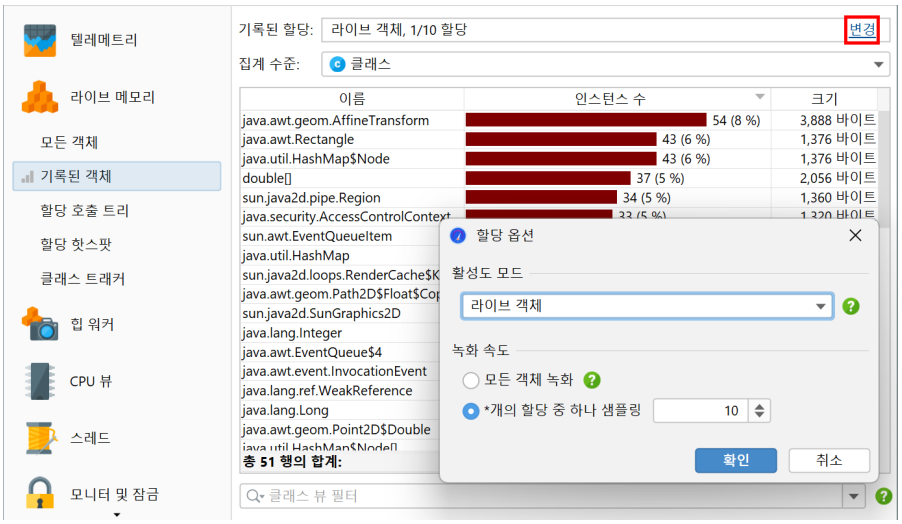


할당 핫스팟 뷰는 할당 호출 트리와 함께 채워지며, 선택된 클래스 생성을 담당하는 메서드에 직접 집중할 수 있게 해줍니다. 녹화된 객체 뷰와 마찬가지로, 할당 핫스팟 뷰는 현재 상태를 마킹하고 시간에 따른 차이를 관찰하는 것을 지원합니다. 차이 열이 뷰에 추가되어 현재 값 마크 액션이 호출된 시점 이후 핫스팟이 얼마나 변경되었는지를 보여줍니다. 할당 뷰는 기본적으로 주기적으로 업데이트되지 않으므로, 계산 도구 모음 버튼을 클릭하여 기준선 값과 비교할 새 데이터 세트를 얻어야 합니다. 자동 업데이트는 옵션 대화 상자에서 사용할 수 있지만, 큰 힙 크기에는 권장되지 않습니다.



할당 녹화 비율

모든 할당을 녹화하는 것은 상당한 오버헤드를 추가합니다. 많은 경우, 할당의 총 숫자는 중요하지 않으며, 상대적인 숫자가 문제를 해결하는 데 충분합니다. 그래서 JProfiler는 기본적으로 10번째 할당만 녹화합니다. 이는 모든 할당을 녹화하는 것에 비해 오버헤드를 대략 1/10로 줄입니다. 모든 할당을 녹화하거나, 목적에 따라 더 적은 할당이 충분한 경우, 녹화된 객체 뷰와 할당 호출 트리 및 핫스팟 뷰의 매개변수 대화 상자에서 녹화 비율을 변경할 수 있습니다.

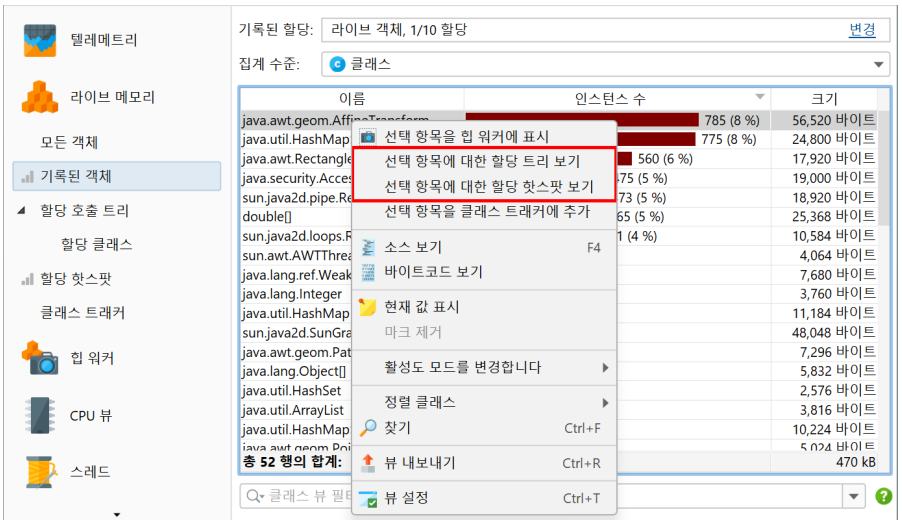


이 설정은 세션 설정 대화 상자의 "고급 설정->메모리 프로파일링" 단계에서도 찾을 수 있으며, 오프라인 프로파일링 세션에 대해 조정할 수 있습니다.

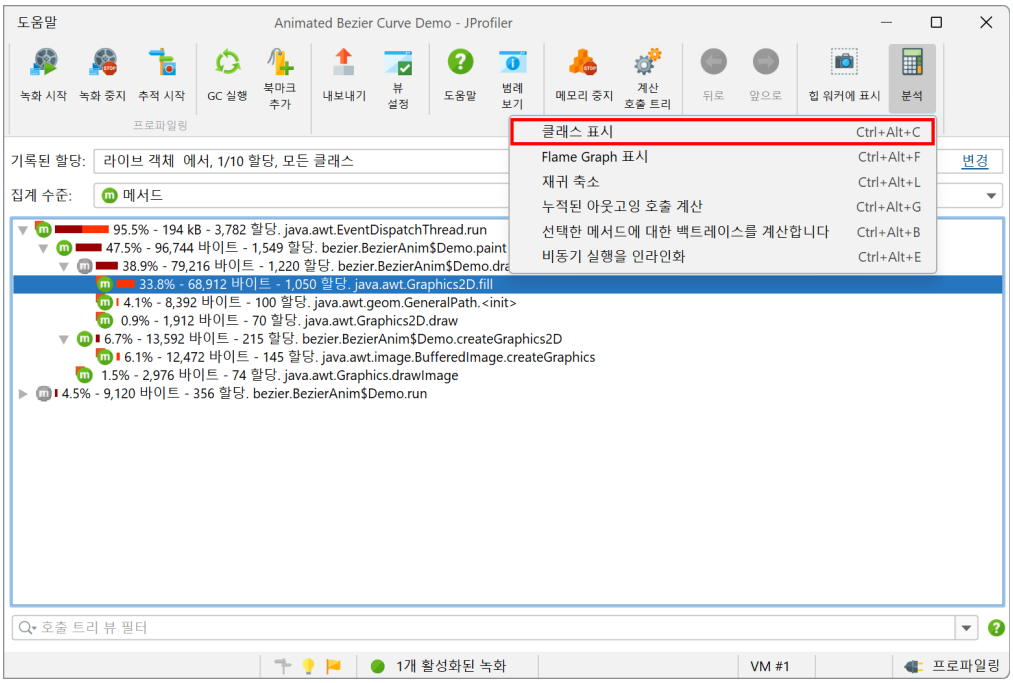
할당 녹화 비율은 "녹화된 객체" 및 "녹화된 처리량"에 대한 VM 텔레메트리에 영향을 미치며, 해당 값은 구성된 비율로 측정됩니다. 스냅샷을 비교 [p. 127]할 때, 첫 번째 스냅샷의 할당 비율이 보고되며, 필요한 경우 다른 스냅샷은 이에 따라 조정됩니다.

할당된 클래스 분석

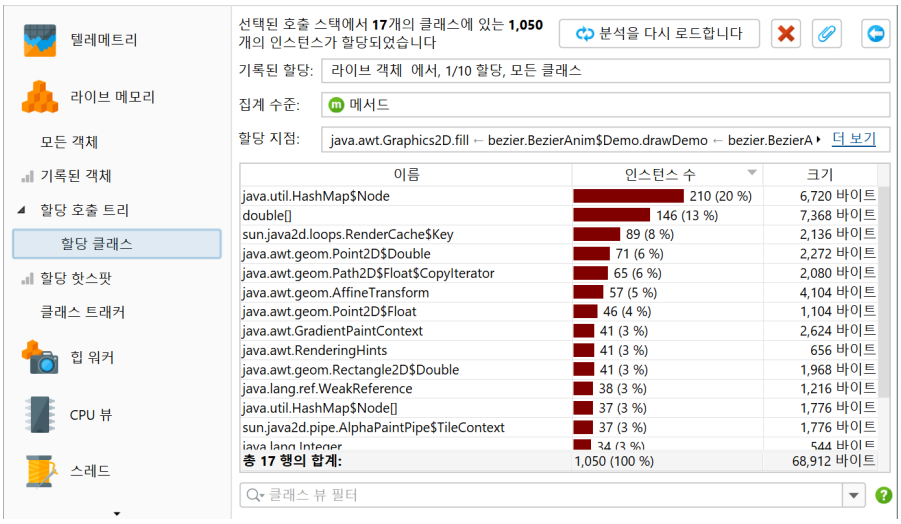
할당 트리 및 할당 핫스팟 뷰를 계산할 때, 미리 보고 싶은 클래스나 패키지를 지정해야 합니다. 이는 이미 특정 클래스에 집중한 경우 잘 작동하지만, 사전 개념 없이 할당 핫스팟을 찾으려고 할 때는 불편합니다. 한 가지 방법은 "녹화된 객체" 뷰를 보고 선택된 클래스나 패키지에 대한 할당 트리 또는 할당 핫스팟 뷰로 전환하기 위한 컨텍스트 메뉴의 액션을 사용하는 것입니다.



또 다른 방법은 모든 클래스에 대한 할당 트리 또는 할당 핫스팟으로 시작하고, 클래스 표시 액션을 사용하여 선택된 할당 지점 또는 할당 핫스팟에 대한 클래스를 표시하는 것입니다.



할당된 클래스의 히스토그램은 호출 트리 분석 [p. 179]으로 표시됩니다. 이 액션은 다른 호출 트리 분석에서도 작동합니다.

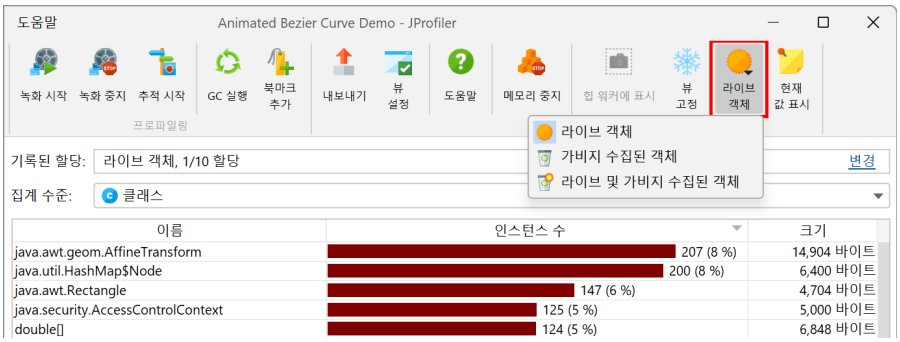


클래스 분석 뷰는 정적이며, 할당 트리 및 핫스팟 뷰가 재계산될 때 업데이트되지 않습니다. 분석 다시 로드 액션은 먼저 할당 트리를 업데이트한 다음 새 데이터에서 현재 분석 뷰를 재계산합니다.

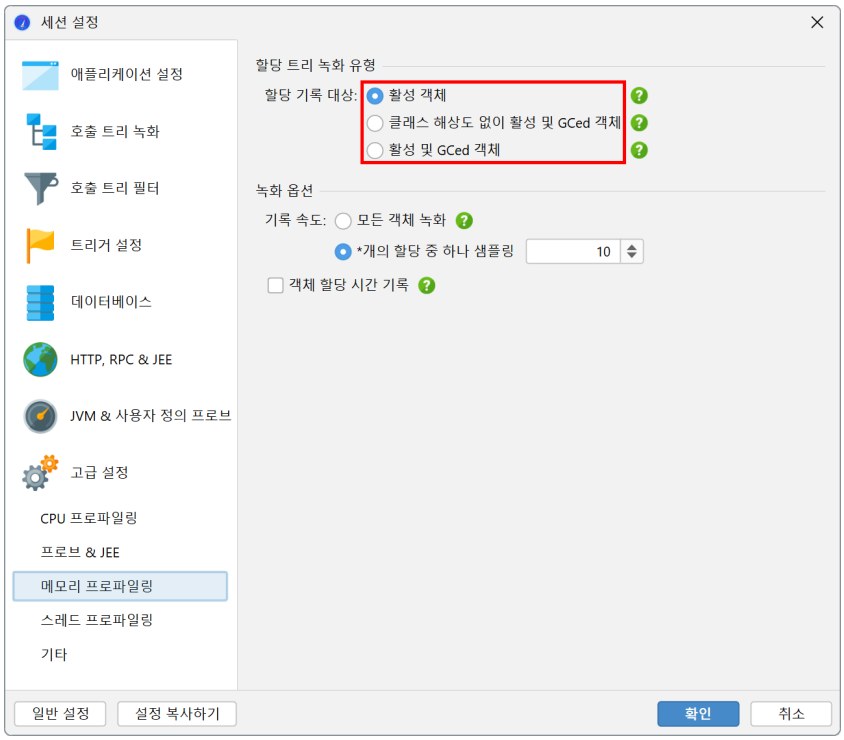
가비지 수집된 객체 분석

할당 녹화는 활성 객체를 보여줄 뿐만 아니라, 가비지 수집된 객체에 대한 정보도 유지합니다. 이는 임시 할당을 조사할 때 유용합니다. 많은 임시 객체를 할당하는 것은 상당한 오버헤드를 발생시킬 수 있으므로, 할당 비율을 줄이면 성능이 크게 향상됩니다.

녹화된 객체 뷰에서 가비지 수집된 객체를 표시하려면, 활성도 선택기를 가비지 수집된 객체 또는 활성 및 가비지 수집된 객체로 변경하십시오. 할당 호출 트리 및 할당 핫스팟 뷰의 옵션 대화 상자에는 동등한 드롭다운이 있습니다.



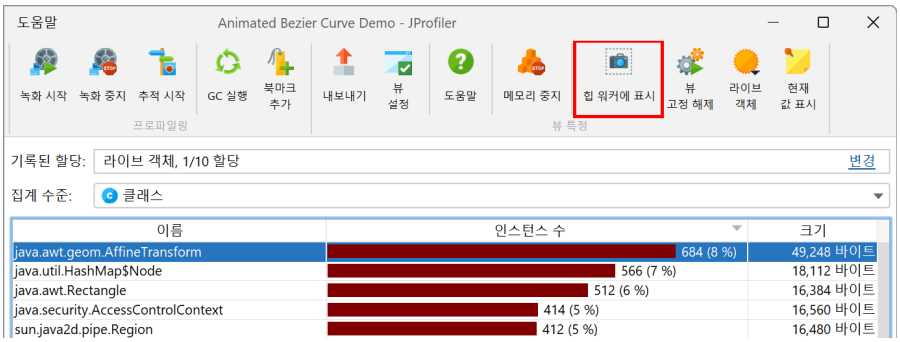
그러나 JProfiler는 기본적으로 가비지 수집된 객체에 대한 할당 트리 정보를 수집하지 않습니다. 이는 활성 객체에 대한 데이터만 유지하는 것이 훨씬 적은 오버헤드로 가능하기 때문입니다. "할당 호출 트리" 또는 "할당 핫스팟" 뷰에서 가비지 수집된 객체를 포함하는 모드로 활성도 선택기를 전환할 때, JProfiler는 녹화 유형을 변경할 것을 제안합니다. 이는 프로파일링 설정의 변경 사항이므로, 즉시 변경을 적용하기로 선택하면 이전에 녹화된 모든 데이터가 지워집니다. 미리 이 설정을 변경하고 싶다면, 세션 설정 대화 상자의 "고급 설정" -> "메모리 프로파일링"에서 그렇게 할 수 있습니다.



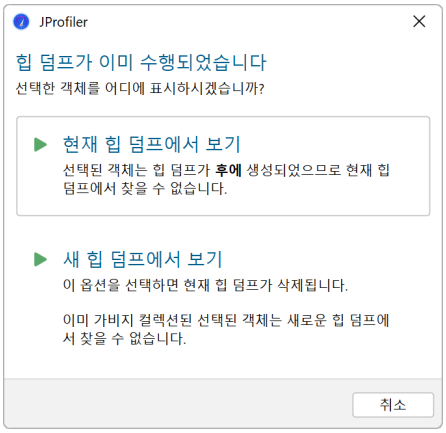
다음 정류장: 힙 워커

더 고급 유형의 질문은 객체 간의 참조를 포함합니다. 예를 들어, 녹화된 객체, 할당 트리 및 할당 핫스팟 뷰에 표시되는 크기는 **얕은 크기**입니다. 이는 클래스의 메모리 레이아웃만 포함하며, 참조된 클래스는 포함하지 않습니다. 클래스의 객체가 실제로 얼마나 무거운지를 보려면, 종종 **유지된 크기**를 알고 싶습니다. 이는 해당 객체가 힙에서 제거될 경우 해제될 메모리 양을 의미합니다.

이러한 종류의 정보는 라이브 메모리 뷰에서는 사용할 수 없습니다. 이는 힙의 모든 객체를 열거하고 비용이 많이 드는 계산을 수행해야 하기 때문입니다. 이 작업은 힙 워커가 처리합니다. 라이브 메모리 뷰의 관심 지점에서 힙 워커로 이동하려면, 힙 워커에서 표시 도구 모음 버튼을 사용할 수 있습니다. 이는 힙 워커의 동등한 뷰로 이동합니다.



힙 스냅샷이 없는 경우, 새 힙 스냅샷이 생성되며, 그렇지 않으면 JProfiler는 기존 힙 스냅샷을 사용할지 여부를 묻습니다.



어쨌든, 라이브 메모리 뷰와 힙 워커의 숫자가 종종 매우 다를 것임을 이해하는 것이 중요합니다. 힙 워커가 라이브 메모리 뷰와 다른 시점의 스냅샷을 보여주는 것 외에도, 모든 참조되지 않은 객체를 제거하기 때문입니다. 가비지 수집기의 상태에 따라, 참조되지 않은 객체는 힙의 상당 부분을 차지할 수 있습니다.

힙 워커

힙 스냅샷

객체 간의 참조를 포함하는 모든 힙 분석은 힙 스냅샷이 필요합니다. JVM에 객체로의 들어오는 참조를 물어볼 수 없기 때문입니다. 해당 질문에 답하기 위해 전체 힙을 반복해야 합니다. 그 힙 스냅샷에서 JProfiler는 힙 워커의 뷰를 제공하는 데 필요한 데이터를 생성하는 내부 데이터베이스를 만듭니다.

힙 스냅샷의 두 가지 소스가 있습니다: JProfiler 힙 스냅샷과 HPROF/PHD 힙 스냅샷. JProfiler 힙 스냅샷은 힙 워커의 모든 사용 가능한 기능을 지원합니다. 프로파일링 에이전트는 프로파일링 인터페이스 JVMTI를 사용하여 모든 참조를 반복합니다. 프로파일된 JVM이 다른 머신에서 실행 중인 경우, 모든 정보는 로컬 머신으로 전송되고 추가 계산이 거기서 수행됩니다. HPROF/PHD 스냅샷은 JVM의 내장 메커니즘으로 생성되며 JProfiler가 읽을 수 있는 표준 형식으로 디스크에 기록됩니다. HotSpot JVM은 HPROF 스냅샷을 생성할 수 있으며 Eclipse OpenJ9 JVM은 PHD 스냅샷을 제공합니다.

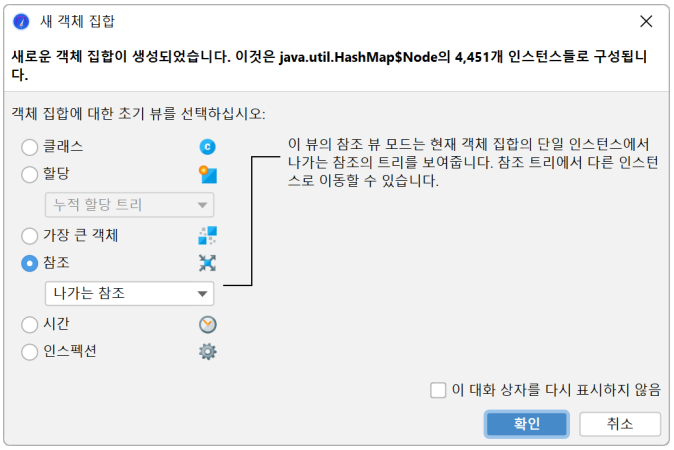
힙 워커의 개요 페이지에서 JProfiler 힙 스냅샷 또는 HPROF/PHD 힙 스냅샷을 생성할지 선택할 수 있습니다. 기본적으로 JProfiler 힙 스냅샷이 권장됩니다. HPROF/PHD 힙 스냅샷은 다른 장 [p. 195]에서 논의된 특별한 상황에서 유용합니다.

선택 단계

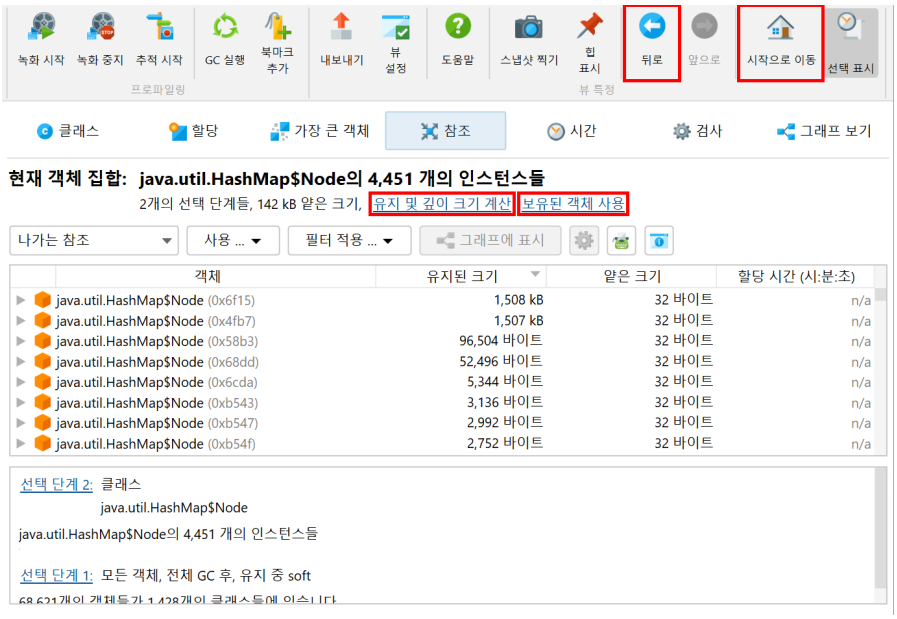
힙 워커는 선택된 객체 집합의 다양한 측면을 보여주는 여러 뷰로 구성됩니다. 힙 스냅샷을 찍은 직후, 힙의 모든 객체를 보고 있습니다. 각 뷰에는 선택된 객체를 **현재 객체 집합**으로 변환하는 탐색 작업이 있습니다. 힙 워커의 헤더 영역에는 현재 객체 집합에 포함된 객체 수에 대한 정보가 표시됩니다.

처음에는 라이브 메모리 섹션 [p. 68]의 "모든 객체" 뷰와 유사한 "클래스" 뷰를 보고 있습니다. 클래스를 선택하고 사용->선택된 인스턴스를 호출하면 해당 클래스의 인스턴스만 포함하는 새로운 객체 집합이 생성됩니다. 힙 워커에서 "사용"은 항상 새로운 객체 집합을 생성하는 것을 의미합니다.

새로운 객체 집합의 경우, 힙 워커의 클래스 뷰를 보여주는 것은 흥미롭지 않을 것입니다. 왜냐하면 이전에 선택한 클래스로 테이블을 필터링하는 것과 같기 때문입니다. 대신 JProfiler는 "새로운 객체 집합" 대화 상자와 함께 다른 뷰를 제안합니다. 이 대화 상자를 취소하여 새로운 객체 집합을 버리고 이전 뷰로 돌아갈 수 있습니다. 나가는 참조 뷰가 제안되지만 다른 뷰를 선택할 수도 있습니다. 이것은 처음 표시된 뷰에만 해당되며, 이후 힙 워커의 뷰 선택기에서 뷰를 전환할 수 있습니다.



헤더 영역은 이제 두 개의 선택 단계가 있음을 알려주고, 현재 객체 집합에 의해 유지되는 모든 객체를 사용하기 위한 링크를 포함합니다. 후자는 또 다른 선택 단계를 추가하고 해당 객체 집합에 여러 클래스가 있을 가능성이 있기 때문에 클래스 뷰를 제안합니다.

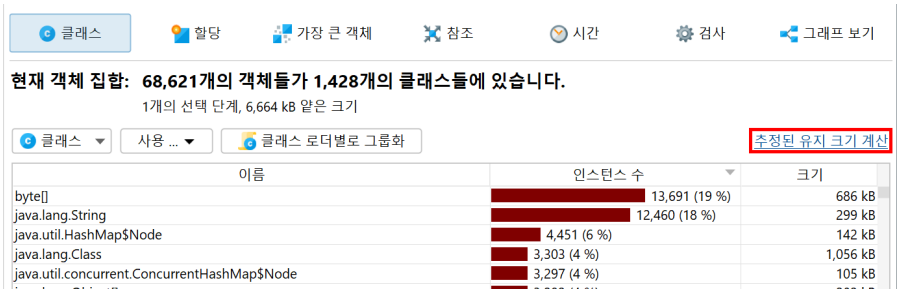


힙 워커의 하단 부분에는 지금까지의 선택 단계가 나열됩니다. 하이퍼링크를 클릭하면 선택 단계로 돌아갈 수 있습니다. 첫 번째 데이터 집합은 도구 모음의 시작으로 이동 버튼으로도 접근할 수 있습니다. 도구 모음의 뒤로 및 앞으로 버튼은 분석에서 되돌아가야 할 때 유용합니다.

클래스 뷰

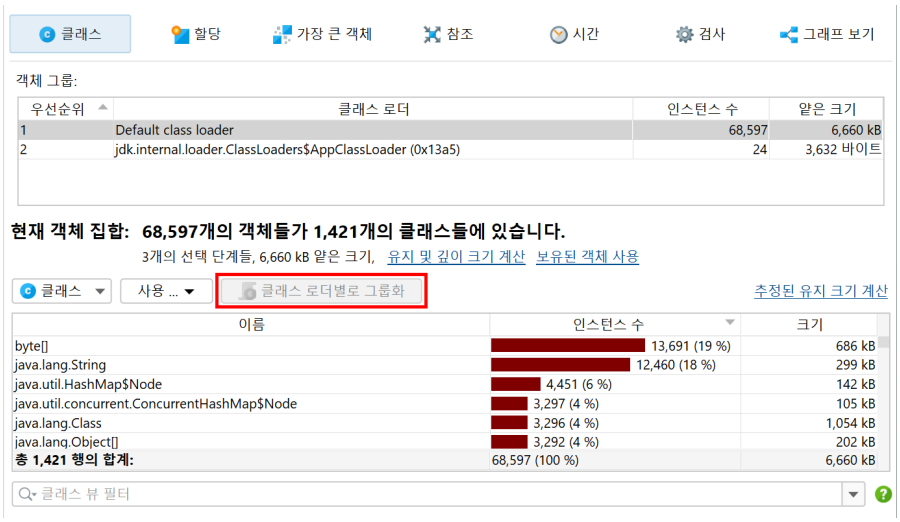
힙 워커 상단의 뷰 선택기에는 현재 객체 집합에 대한 다양한 정보를 보여주는 다섯 개의 뷰가 포함되어 있습니다. 그 중 첫 번째는 "클래스" 뷰입니다.

클래스 뷰는 라이브 메모리 섹션의 "모든 객체" 뷰와 유사하며, 클래스를 패키지로 그룹화할 수 있는 집계 수준 선택기가 있습니다. 또한 클래스에 대한 추정된 유지 크기를 표시할 수 있습니다. 이는 클래스의 모든 인스턴스가 힙에서 제거될 경우 해제될 메모리 양입니다. 추정된 유지 크기 계산 하이퍼링크를 클릭하면 새로운 유지 크기 열이 추가됩니다. 표시된 유지 크기는 추정된 하한이며, 정확한 숫자를 계산하는 것은 너무 느립니다. 정말로 정확한 숫자가 필요하다면, 관심 있는 클래스나 패키지를 선택하고 새로운 객체 집합의 헤더에 있는 유지 및 깊이 크기 계산 하이퍼링크를 사용하세요.



하나 이상의 클래스나 패키지를 선택한 후, 인스턴스 자체, 관련된 `java.lang.Class` 객체 또는 모든 유지 객체를 선택할 수 있습니다. 더블 클릭은 가장 빠른 선택 모드이며 선택된 인스턴스를 사용합니다. 여러 선택 모드가 가능한 경우, 이 경우처럼, 뷰 위에 사용 드롭다운 메뉴가 표시됩니다.

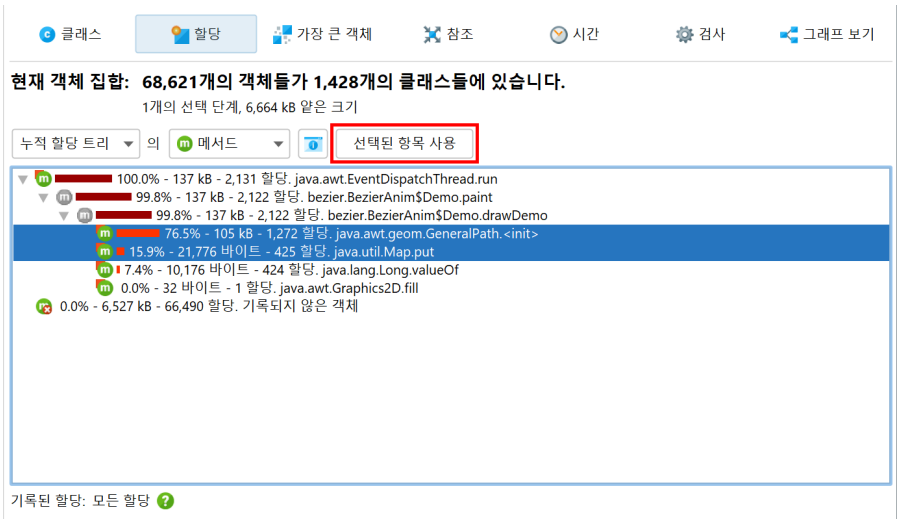
클래스 로더 관련 문제를 해결할 때, 인스턴스를 클래스 로더별로 그룹화해야 하는 경우가 많습니다. 검사 탭은 클래스 뷰에서 사용할 수 있는 "클래스 로더별 그룹화" 검사를 제공합니다. 이는 해당 컨텍스트에서 특히 중요하기 때문입니다. 해당 분석을 실행하면 상단에 모든 클래스 로더를 보여주는 그룹화 테이블이 표시됩니다. 클래스 로더를 선택하면 아래 뷰에서 데이터가 필터링됩니다. 그룹화 테이블은 다른 뷰로 전환할 때도 그대로 유지되며, 다른 선택 단계를 수행할 때까지 유지됩니다. 그런 다음 클래스 로더 선택이 해당 선택 단계의 일부가 됩니다.



할당 녹화 뷰

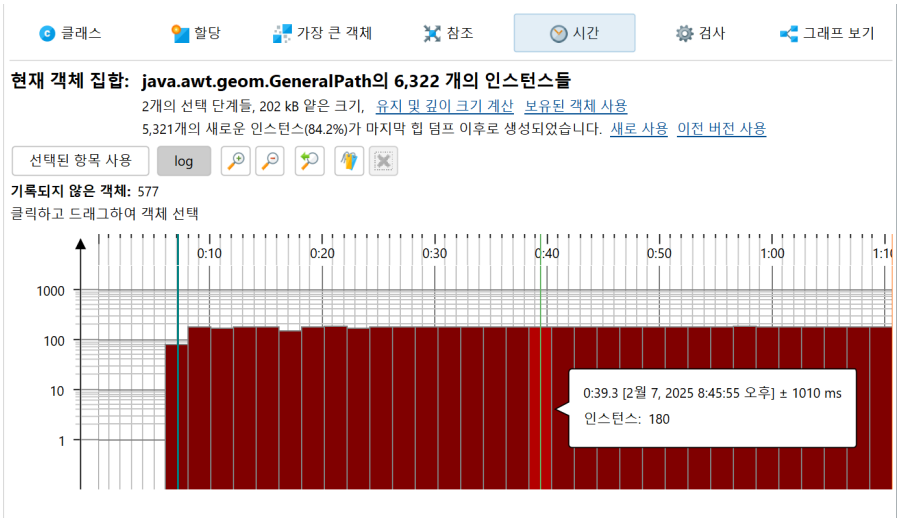
객체가 할당된 위치에 대한 정보는 메모리 누수의 용의자를 좁히거나 메모리 소비를 줄이려 할 때 중요할 수 있습니다. JProfiler 힙 스냅샷의 경우, "할당" 뷰는 할당 호출 트리와 할당 핫스팟을 보여줍니다. 할당이 기록된

객체에 대해 다른 객체는 할당 호출 트리의 "기록되지 않은 객체" 노드에 그룹화됩니다. HPROF/PHD 스냅샷의 경우, 이 뷰는 사용할 수 없습니다.



클래스 뷰와 마찬가지로, 여러 노드를 선택하고 상단의 선택된 사용 버튼을 사용하여 새로운 선택 단계를 만들 수 있습니다. "할당 핫스팟" 뷰 모드에서는 백 트레이스에서 노드를 선택할 수도 있습니다. 이는 선택된 백 트레이스로 끝나는 호출 스택에서 할당된 관련 최상위 핫스팟의 객체만 선택합니다.

JProfiler가 할당을 기록할 때 저장할 수 있는 또 다른 정보는 객체가 할당된 시간입니다. 힙 워커의 "시간" 뷰는 현재 객체 집합의 모든 기록된 인스턴스에 대한 할당 시간의 히스토그램을 보여줍니다. 하나 이상의 간격을 선택하고 선택된 사용 버튼으로 새로운 객체 집합을 만들 수 있습니다.



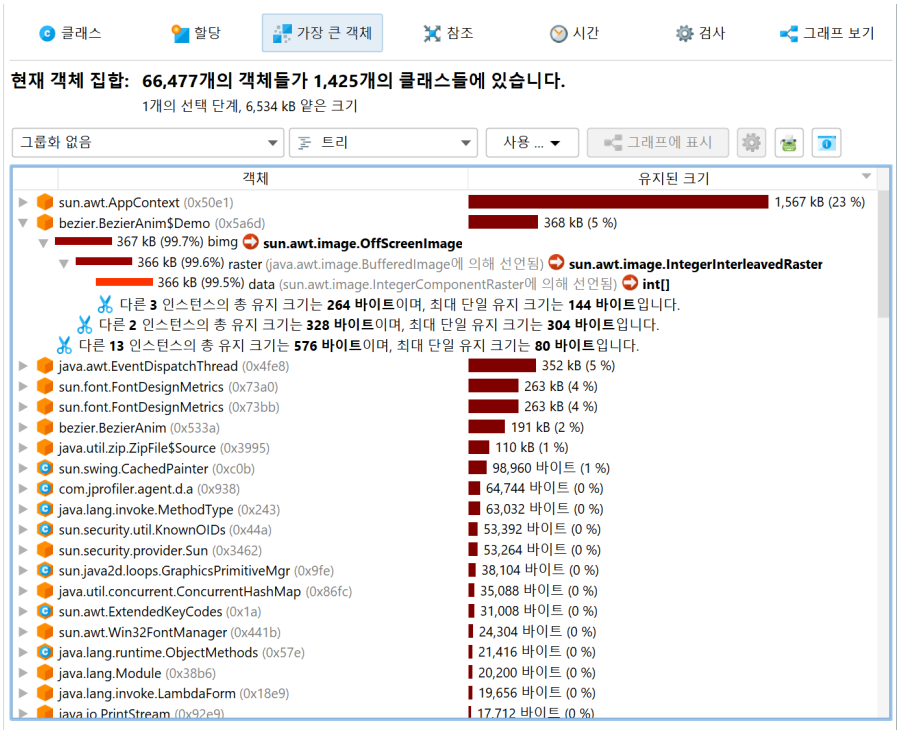
시간 간격을 더 정확하게 선택하려면 북마크 [p. 44] 범위를 지정할 수 있습니다. 선택된 첫 번째 및 마지막 북마크 사이의 모든 객체가 표시됩니다.

시간 뷰 외에도 할당 시간은 참조 뷰에서 별도의 열로 표시됩니다. 그러나 할당 시간 기록은 기본적으로 활성화되어 있지 않습니다. 시간 뷰에서 직접 켜거나 세션 설정 대화 상자의 고급 설정 -> 메모리 프로파일링에서 설정을 편집할 수 있습니다.

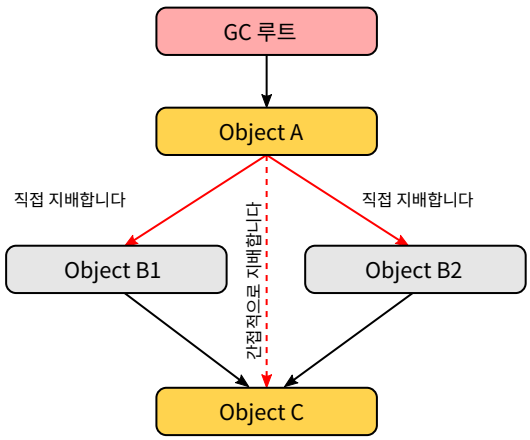
가장 큰 객체 뷰

가장 큰 객체 뷰는 현재 객체 집합에서 가장 중요한 객체 목록을 보여줍니다. 이 컨텍스트에서 "가장 큰"은 힙에서 제거될 경우 가장 많은 메모리를 해제할 객체를 의미합니다. 그 크기는 **유지 크기**라고 합니다. 반면에 **깊이 크기**는 강한 참조를 통해 도달할 수 있는 모든 객체의 총 크기입니다.

각 객체는 이 객체에 의해 유지되는 다른 객체에 대한 나가는 참조를 보여주기 위해 확장할 수 있습니다. 이 방식으로, 조상 중 하나가 제거될 경우 가비지 수집될 유지 객체의 트리를 재귀적으로 확장할 수 있습니다. 이러한 종류의 트리를 "지배자 트리"라고 합니다. 이 트리의 각 객체에 대해 표시되는 정보는 나가는 참조 뷰와 유사하지만 지배 참조만 표시됩니다.



모든 지배된 객체가 지배자에 의해 직접 참조되는 것은 아닙니다. 예를 들어, 다음 그림의 참조를 고려해 보세요:

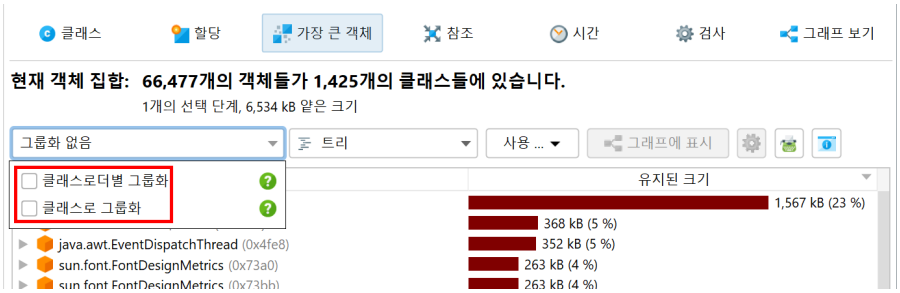


객체 A는 객체 B1과 B2를 지배하며, 객체 C에 대한 직접 참조가 없습니다. B1과 B2 모두 C를 참조합니다. B1이나 B2는 C를 지배하지 않지만 A는 지배합니다. 이 경우, B1, B2 및 C는 지배자 트리에서 A의 직접 자식으로 나열되며, C는 B1 및 B2의 자식으로 나열되지 않습니다. B1 및 B2의 경우, A에서 보유하고 있는 필드 이름이 표시됩니다. C의 경우, 참조 노드에 "[전이 참조]"가 표시됩니다.

지배자 트리의 각 참조 노드 왼쪽에는 상위 수준 객체의 유지 크기의 몇 퍼센트가 대상 객체에 의해 여전히 유지되고 있는지를 보여주는 크기 막대가 있습니다. 트리에서 더 깊이 들어갈수록 숫자가 감소합니다. 뷰 설정에서 퍼센트 기준을 전체 힙 크기로 변경할 수 있습니다.

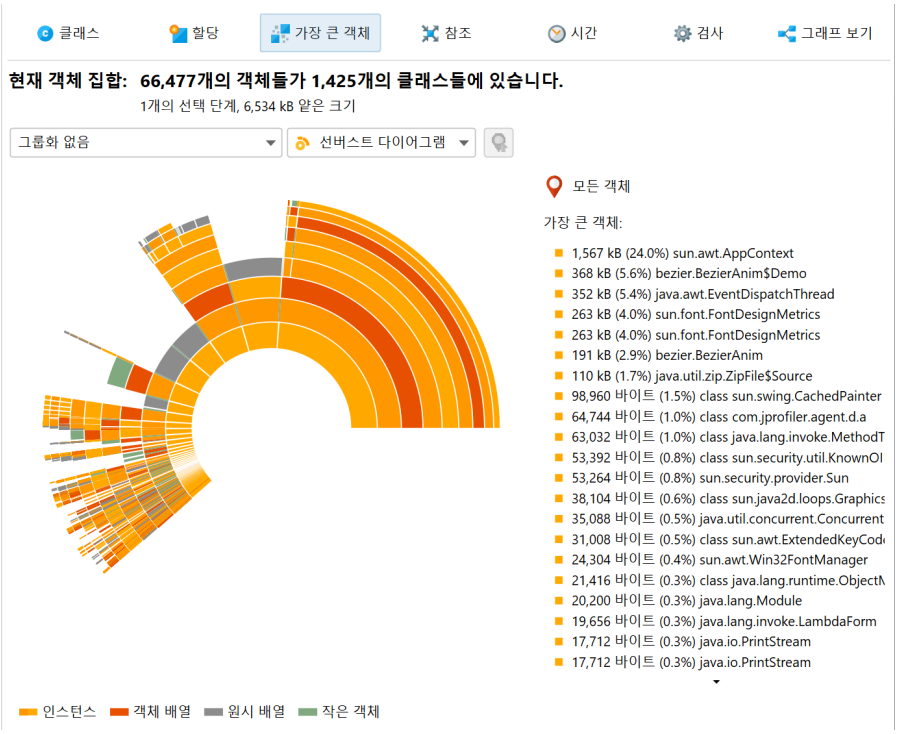
지배자 트리는 유지 크기가 부모 객체의 유지 크기의 0.5% 미만인 모든 객체를 제거하는 내장 컷오프가 있습니다. 이는 중요한 객체에서 주의를 분산시키는 작은 지배 객체의 지나치게 긴 목록을 피하기 위함입니다. 이러한 컷오프가 발생하면, 이 수준에서 표시되지 않는 객체 수, 총 유지 크기 및 단일 객체의 최대 유지 크기에 대해 알리는 특별한 "컷오프" 자식 노드가 표시됩니다.

단일 객체를 표시하는 대신, 지배자 트리는 가장 큰 객체를 클래스별로 그룹화할 수도 있습니다. 뷰 상단의 그룹화 드롭다운에는 이 표시 모드를 활성화하는 체크박스가 포함되어 있습니다. 또한 최상위 수준에 클래스 로더 그룹화를 추가할 수 있습니다. 클래스 로더 그룹화는 가장 큰 객체가 계산된 후 적용되며, 가장 큰 객체의 클래스를 로드한 사람을 보여줍니다. 대신 특정 클래스 로더에 대한 가장 큰 객체를 분석하려면 먼저 "클래스 로더별 그룹화" 검사를 사용할 수 있습니다.

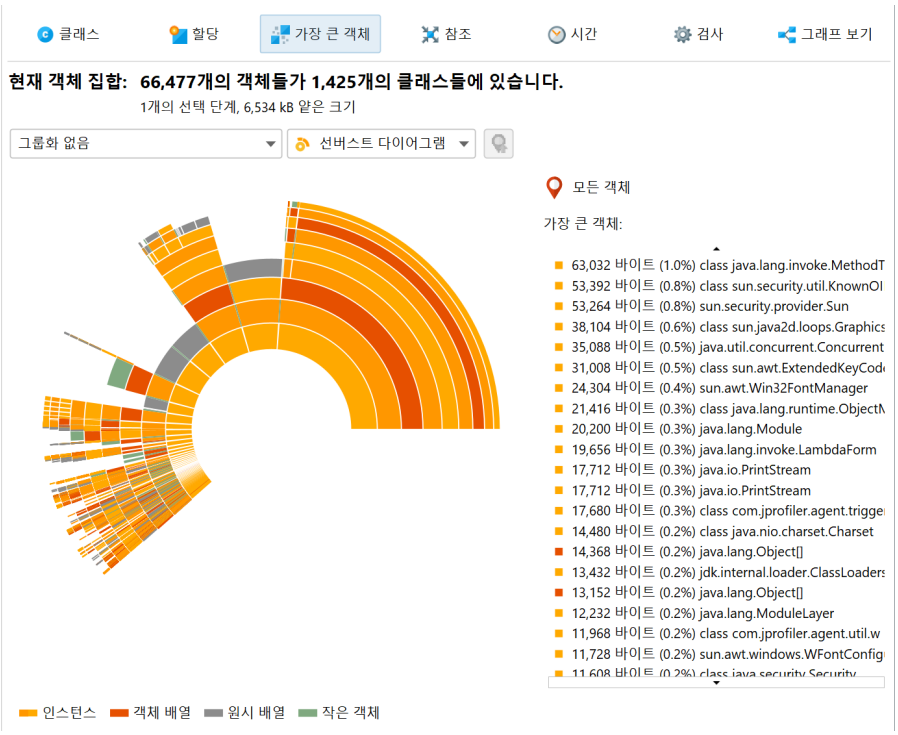


가장 큰 객체 뷰 위의 뷰 모드 선택기를 사용하여 선버스트 다이어그램으로 전환할 수 있습니다. 다이어그램은 일련의 동심원으로 구성되어 있으며, 최대 깊이까지 지배자 트리의 전체 내용을 하나의 이미지로 보여줍니다. 참조는 가장 안쪽의 링에서 시작하여 원의 바깥쪽 가장자리로 전파됩니다. 이 시각화는 높은 정보 밀도로 평평한 관점을 제공하여 참조 패턴을 발견하고 특별한 색상 코딩을 통해 큰 기본 및 객체 배열을 한눈에 볼 수 있습니다.

현재 객체 집합이 전체 힙인 경우, 원의 전체 둘레는 사용된 힙 크기에 해당합니다. 가장 큰 객체 뷰는 전체 힙의 0.1% 이상을 유지하는 객체만 표시하므로, 이는 상당한 섹터가 비어 있음을 의미하며, 이는 가장 큰 객체에 의해 유지되지 않는 모든 객체에 해당합니다.



링 세그먼트를 클릭하면 원의 새로운 루트를 설정하여 다이어그램에서 볼 수 있는 최대 깊이를 확장합니다. 다이어그램의 빈 중앙을 클릭하면 이전 루트가 복원됩니다. 새로운 루트가 설정된 경우, 원의 전체 둘레는 루트 객체의 유지 크기에 해당합니다. 빈 섹터는 루트 객체의 자체 크기와 가장 큰 유지 객체 목록에 없는 추가 객체를 나타냅니다. 현재 객체 집합이 전체 힙이 아닌 경우, 원의 전체 둘레는 표시된 가장 큰 객체의 합계에 해당하며 빈 섹터는 표시되지 않습니다.

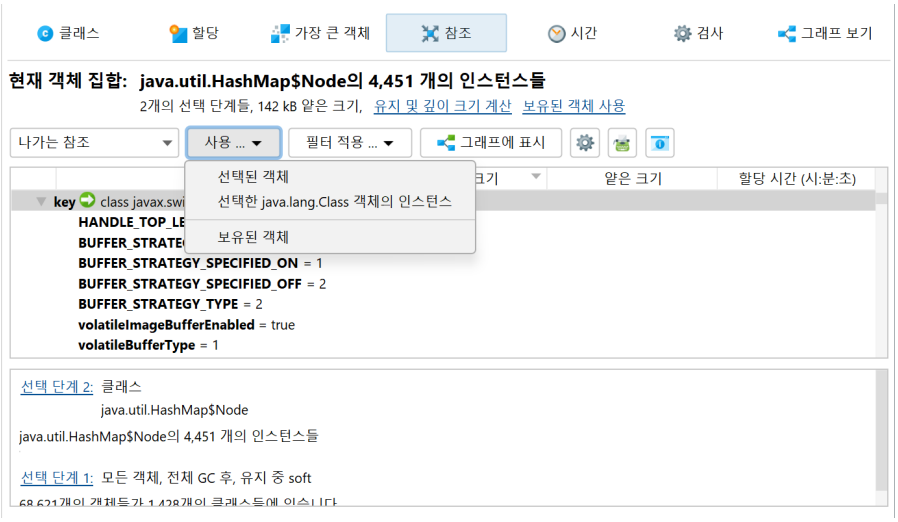


다이어그램 위에 마우스를 올리면 인스턴스와 즉시 유지된 객체에 대한 추가 정보가 오른쪽에 표시됩니다. 마우스가 링 세그먼트 외부에 있을 때, 오른쪽의 목록은 가장 안쪽 링의 가장 큰 객체를 보여줍니다. 해당 목록 위에 마우스를 올리면 해당 링 세그먼트가 강조 표시되고, 목록 항목을 클릭하면 다이어그램의 새로운 루트가 설정됩니다. 새로운 객체 집합을 만들려면 링 세그먼트와 목록 항목 모두에서 컨텍스트 메뉴의 작업을 선택할 수 있습니다.

참조 뷰

이전 뷰와 달리, 참조 뷰는 최소한 하나의 선택 단계를 수행한 경우에만 사용할 수 있습니다. 초기 객체 집합의 경우, 이러한 뷰는 유용하지 않습니다. 왜냐하면 들어오는 및 나가는 참조 뷰는 모든 개별 객체를 보여주고, 병합된 참조 뷰는 집중된 객체 집합에 대해서만 해석할 수 있기 때문입니다.

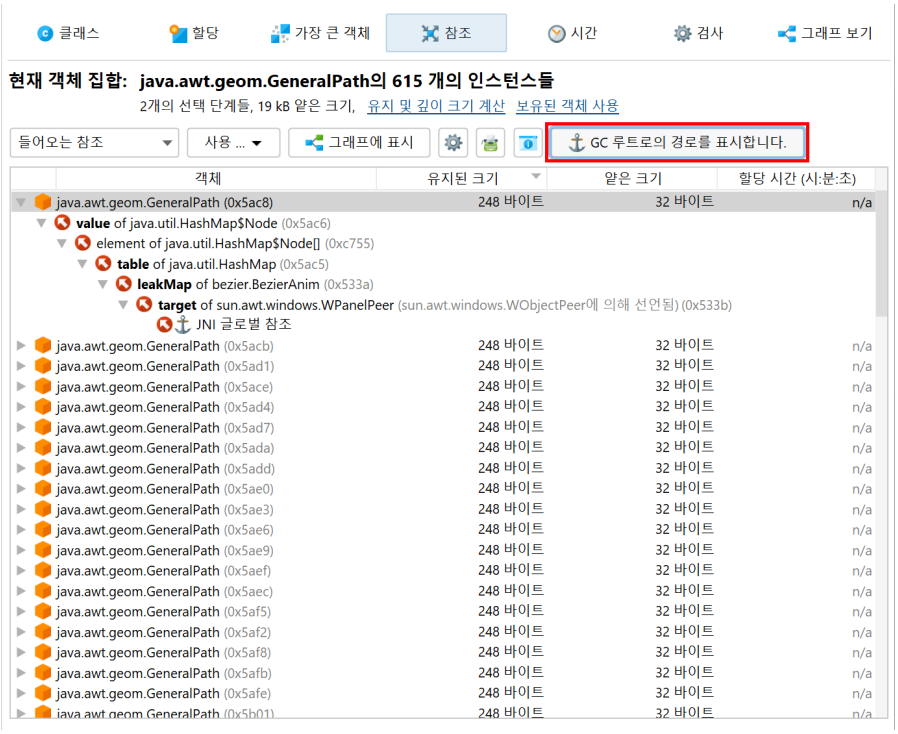
나가는 참조 뷰는 IDE의 디버거가 보여줄 뷰와 유사합니다. 객체를 열면 기본 데이터와 다른 객체에 대한 참조를 볼 수 있습니다. 모든 참조 유형은 새로운 객체 집합으로 선택할 수 있으며, 여러 객체를 한 번에 선택할 수 있습니다. 클래스 뷰와 마찬가지로, 유지 객체 또는 관련된 `java.lang.Class` 객체를 선택할 수 있습니다. 선택된 객체가 표준 컬렉션인 경우, 단일 작업으로 모든 포함된 요소를 선택할 수도 있습니다. 클래스 로더 객체의 경우, 모든 로드된 인스턴스를 선택하는 옵션이 있습니다.



null 참조가 있는 필드는 기본적으로 표시되지 않습니다. 왜냐하면 해당 정보가 메모리 분석에 방해가 될 수 있기 때문입니다. 디버깅 목적으로 모든 필드를 보고 싶다면, 뷰 설정에서 이 동작을 변경할 수 있습니다.

표시된 인스턴스의 간단한 선택 외에도, 나가는 참조 뷰에는 강력한 필터링 기능 [p. 199]이 있습니다. 라이브 세션의 경우, 나가는 참조 뷰와 들어오는 참조 뷰 모두 동일한 장에서 논의된 고급 조작 및 표시 기능을 가지고 있습니다.

들어오는 참조 뷰는 메모리 누수를 해결하기 위한 주요 도구입니다. 객체가 가비지 수집되지 않는 이유를 찾으려면, GC 루트로의 경로 표시 버튼이 가비지 수집기 루트로의 참조 체인을 찾습니다. 메모리 누수 [p. 203]에 대한 장에는 이 중요한 주제에 대한 자세한 정보가 있습니다.

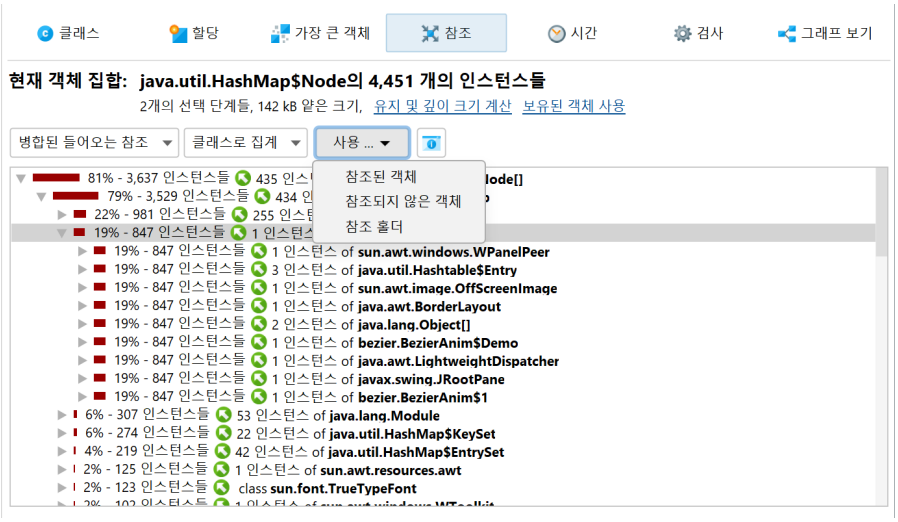


병합된 참조

많은 다른 객체에 대한 참조를 확인하는 것은 번거로울 수 있으므로, JProfiler는 현재 객체 집합의 모든 객체에 대한 병합된 나가는 및 들어오는 참조를 보여줄 수 있습니다. 기본적으로 참조는 클래스별로 집계됩니다. 클래스의 인스턴스가 동일한 클래스의 다른 인스턴스에 의해 참조되는 경우, 특별한 노드가 삽입되어 이러한 클래스 재귀 참조에서 원래 인스턴스와 인스턴스를 보여줍니다. 이 메커니즘은 연결 리스트와 같은 일반적인 데이터 구조의 내부 참조 체인을 자동으로 축소합니다.

필드별로 그룹화된 병합된 참조를 표시하도록 선택할 수도 있습니다. 이 경우, 각 노드는 클래스의 특정 필드나 배열의 내용과 같은 참조 유형입니다. 표준 컬렉션의 경우, 누적을 방해할 내부 참조 체인이 압축되어 "java.lang.HashMap의 맵 값"과 같은 참조 유형을 볼 수 있습니다. 클래스 집계의 경우와 달리, 이 메커니즘은 JRE의 표준 라이브러리에서 명시적으로 지원되는 컬렉션에만 작동합니다.

"병합된 나가는 참조" 뷰에서 인스턴스 수는 참조된 객체를 나타냅니다. "병합된 들어오는 참조" 뷰에서는 각 행에 두 개의 인스턴스 수가 표시됩니다. 첫 번째 인스턴스 수는 현재 객체 집합에서 이 경로를 따라 참조된 인스턴스 수를 보여줍니다. 노드 왼쪽의 막대 아이콘은 이 비율을 시각화합니다. 화살표 아이콘 뒤의 두 번째 인스턴스 수는 상위 노드에 대한 참조를 보유한 객체를 나타냅니다. 선택 단계를 수행할 때, 선택된 방식으로 참조된 현재 객체 집합의 객체를 선택할지, 선택된 참조를 가진 객체(참조 보유자)에 관심이 있는지를 선택할 수 있습니다.

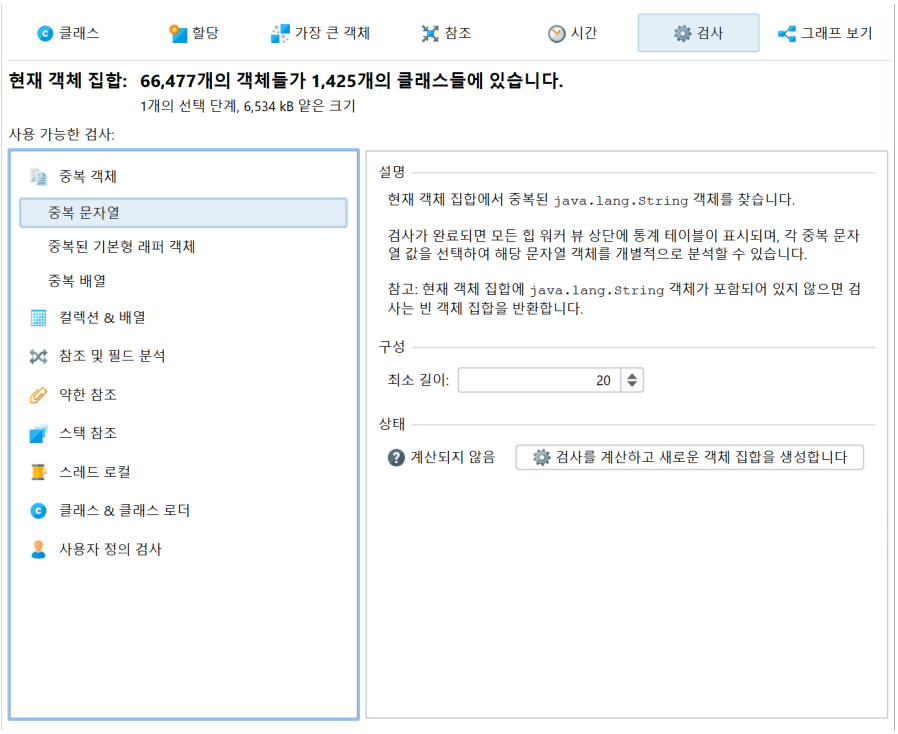


"병합된 지배 참조" 뷰를 사용하면 현재 객체 집합의 일부 또는 전체 객체가 가비지 수집될 수 있도록 제거해야 하는 참조를 찾을 수 있습니다. 지배 참조 트리는 가장 큰 객체 뷰의 지배자 트리의 병합된 역으로 해석될 수 있으며, 클래스별로 집계됩니다. 참조 화살표는 두 클래스 간의 직접 참조를 나타내지 않을 수 있지만, 지배하지 않는 참조를 보유한 다른 클래스가 있을 수 있습니다. 여러 가비지 수집기 루트의 경우, 현재 객체 집합의 일부 또는 전체 객체에 대해 지배 참조가 존재하지 않을 수 있습니다.

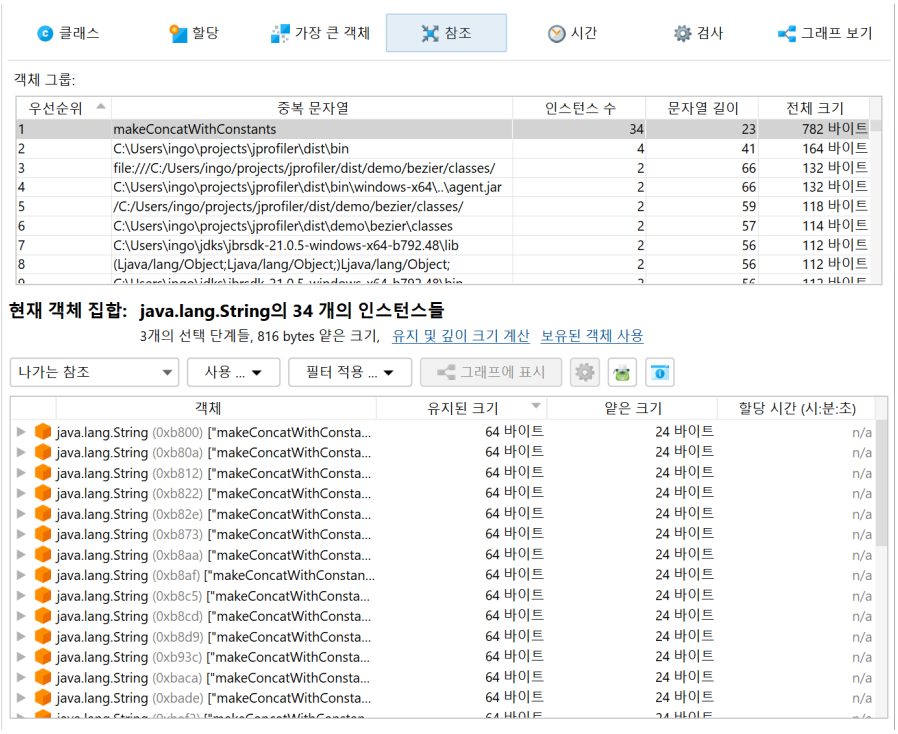
기본적으로 "병합된 지배 참조" 뷰는 들어오는 지배 참조를 보여주며, 트리를 열면 GC 루트에 의해 보유된 객체에 도달할 수 있습니다. 때로는 참조 트리가 여러 다른 경로를 따라 동일한 루트 객체로 이어질 수 있습니다. 뷰 상단의 드롭다운에서 "GC 루트에서 객체로" 뷰 모드를 선택하면, 루트가 최상위 수준에 있고 현재 객체 집합의 객체가 리프 노드에 있는 반대 관점을 볼 수 있습니다. 이 경우, 참조는 최상위 수준에서 리프 노드로 향합니다. 어느 관점이 더 나은지는 제거하려는 참조가 현재 객체 집합에 가까운지, GC 루트에 가까운지에 따라 다릅니다.

검사

"검사" 뷰는 자체적으로 데이터를 표시하지 않습니다. 다른 뷰에서 사용할 수 없는 규칙에 따라 새로운 객체 집합을 생성하는 여러 힙 분석을 제공합니다. 예를 들어, 스레드 로컬에 의해 유지되는 모든 객체를 보고 싶을 수 있습니다. 이는 참조 뷰에서 수행할 수 없습니다. 검사는 여러 범주로 그룹화되어 있으며 설명에서 설명됩니다.



검사는 계산된 객체 집합을 그룹으로 분할할 수 있습니다. 그룹은 힙 워커 상단의 테이블에 표시됩니다. 예를 들어, "중복 문자열" 검사는 중복된 문자열 값을 그룹으로 표시합니다. 참조 뷰에 있는 경우, 선택된 문자열 값으로 `java.lang.String` 인스턴스를 볼 수 있습니다. 처음에는 그룹 테이블의 첫 번째 행이 선택됩니다. 선택을 변경하면 현재 객체 집합이 변경됩니다. 그룹 테이블의 인스턴스 수 및 크기 열은 행을 선택할 때 현재 객체 집합이 얼마나 큰지를 알려줍니다.



그룹 선택은 힙 워커에서 별도의 선택 단계가 아니지만, 검사가 수행한 선택 단계의 일부가 됩니다. 선택 단계를 변경하면 선택 단계 창이 즉시 업데이트됩니다.

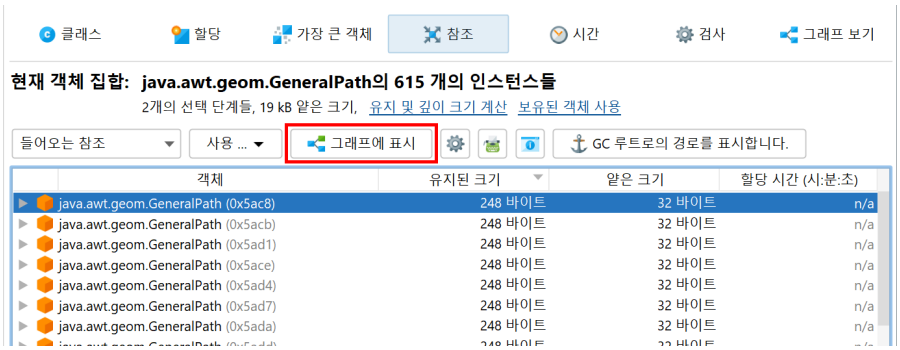
그룹을 생성하는 각 검사는 검사 컨텍스트에서 가장 중요한 그룹을 결정합니다. 이는 항상 다른 열의 자연 정렬 순서와 일치하지 않기 때문에, 그룹 테이블의 우선순위 열에는 검사를 위한 정렬 순서를 강제하는 숫자 값이 포함되어 있습니다.

검사는 큰 힙에 대해 계산하기에 비용이 많이 들 수 있으므로 결과는 캐시됩니다. 이렇게 하면, 기록에서 돌아가 이전에 계산된 검사의 결과를 기다리지 않고 볼 수 있습니다.

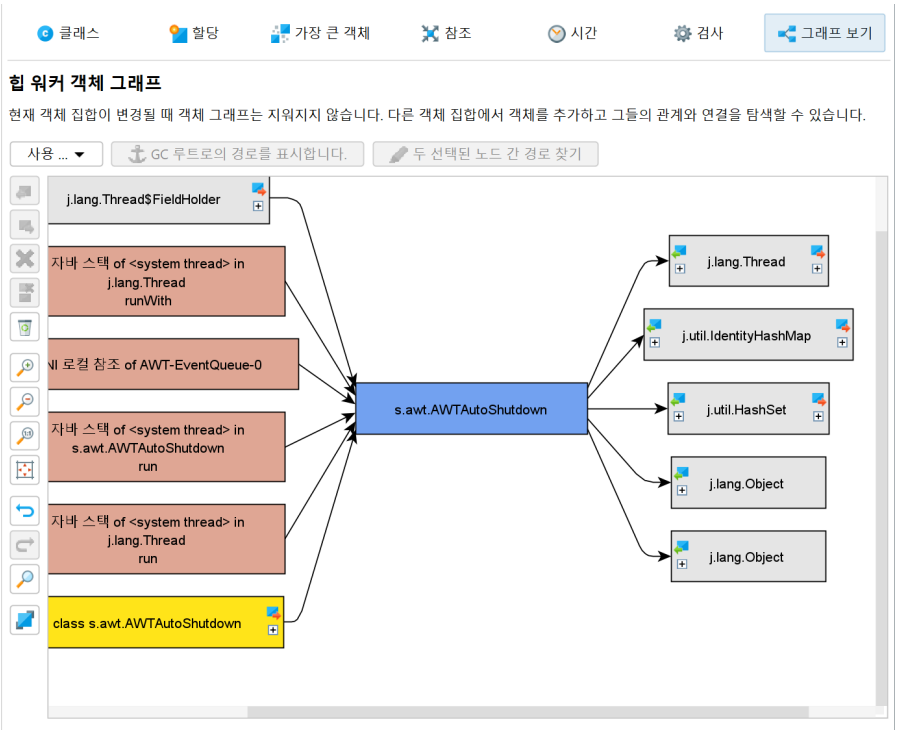
힙 워커 그래프

인스턴스와 그 참조를 함께 가장 현실적으로 표현하는 것은 그래프입니다. 그래프는 시각적 밀도가 낮고 일부 유형의 분석에는 비실용적일 수 있지만, 여전히 객체 간의 관계를 시각화하는 가장 좋은 방법입니다. 예를 들어, 순환 참조는 트리에서 해석하기 어렵지만 그래프에서는 즉시 명백합니다. 또한, 나가는 참조와 들어오는 참조를 함께 보는 것이 유익할 수 있습니다. 이는 트리 구조에서는 하나 또는 다른 것만 볼 수 있기 때문에 불가능합니다.

힙 워커 그래프는 현재 객체 집합의 객체를 자동으로 표시하지 않으며, 현재 객체 집합을 변경할 때 지워지지 않습니다. 나가는 참조 뷰, 들어오는 참조 뷰 또는 가장 큰 객체 뷰에서 선택한 객체를 수동으로 그래프에 추가하여 하나 이상의 인스턴스를 선택하고 그래프에 표시 작업을 사용합니다.

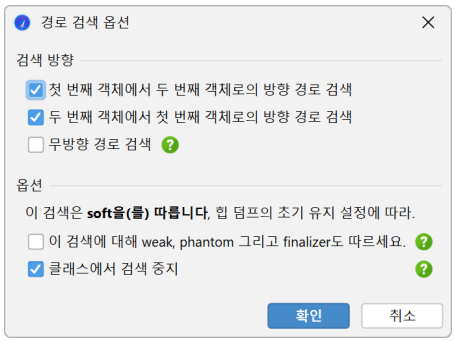


그래프의 패키지 이름은 기본적으로 축약됩니다. CPU 호출 그래프와 마찬가지로, 뷰 설정에서 전체 표시를 활성화할 수 있습니다. 참조는 화살표로 그려집니다. 참조 위에 마우스를 올리면 특정 참조에 대한 세부 정보를 보여주는 툴팁 창이 표시됩니다. 참조 뷰에서 수동으로 추가된 인스턴스는 파란색 배경을 가지고 있습니다. 인스턴스가 더 최근에 추가될수록 배경 색상이 더 어두워집니다. 가비지 수집기 루트는 빨간색 배경을 가지고 있으며 클래스는 노란색 배경을 가지고 있습니다.



기본적으로 참조 그래프는 현재 인스턴스의 직접적인 들어오는 및 나가는 참조만 표시합니다. 객체를 더블 클릭하여 그래프를 확장할 수 있습니다. 이는 해당 객체에 대한 직접적인 들어오는 참조 또는 직접적인 나가는 참조를 확장합니다. 인스턴스의 왼쪽 및 오른쪽에 있는 확장 컨트롤을 사용하여 들어오는 및 나가는 참조를 선택적으로 열 수 있습니다. 되돌아가야 할 경우, 그래프의 이전 상태를 복원하기 위해 실행 취소 기능을 사용하여 너무 많은 노드에 의해 방해받지 않도록 하세요. 그래프를 다듬기 위해 모든 연결되지 않은 노드를 제거하거나 모든 객체를 제거하는 작업이 있습니다.

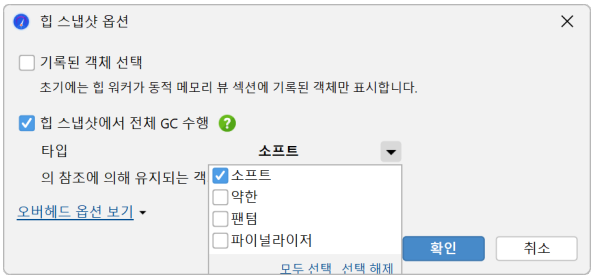
들어오는 참조 뷰와 마찬가지로, 그래프에는 GC 루트로의 경로 표시 버튼이 있어 가비지 수집기 루트로의 참조 체인 [p. 203]을 확장합니다. 또한, 두 인스턴스가 선택된 경우 활성화되는 두 선택된 노드 간의 경로 찾기 작업이 있습니다. 이는 약한 참조를 따라가는 경로를 검색할 수 있습니다. 적절한 경로가 발견되면 빨간색으로 표시됩니다.



초기 객체 집합

힙 스냅샷을 찍을 때, 초기 객체 집합을 제어하는 옵션을 지정할 수 있습니다. 할당을 기록한 경우, 기록된 객체 선택 체크박스는 초기 표시 객체를 기록된 객체로 제한합니다. 힙 워커에 의해 참조되지 않은 객체가 제거되기 때문에, 숫자는 일반적으로 라이브 메모리 뷰의 숫자와 다릅니다. 기록되지 않은 객체는 여전히 힙 스냅샷에 존재하지만, 초기 객체 집합에는 표시되지 않습니다. 추가 선택 단계를 통해 기록되지 않은 객체에 도달할 수 있습니다.

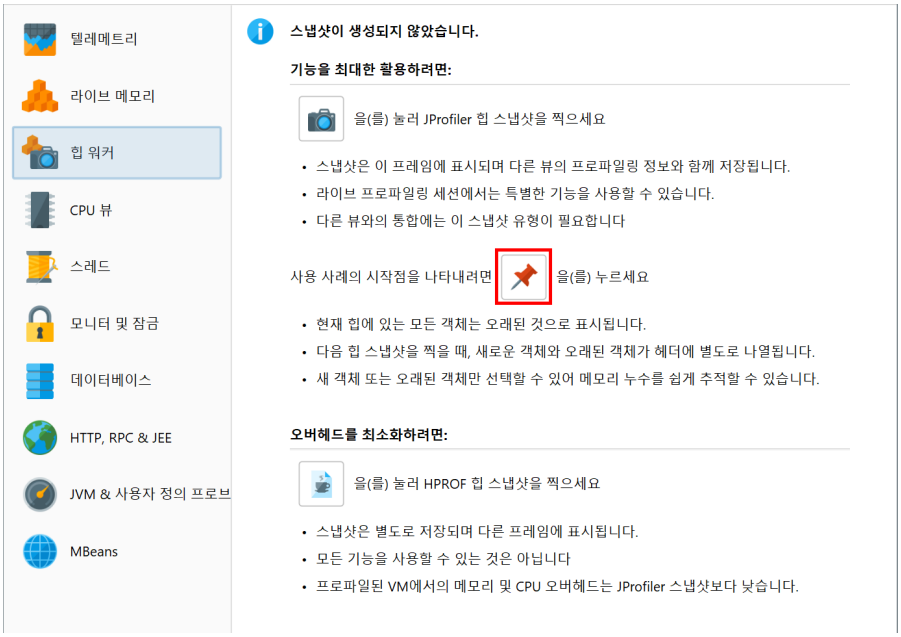
또한, 힙 워커는 가비지 수집을 수행하고 약하게 참조된 객체를 제거합니다. 단, 소프트 참조는 제외됩니다. 이는 약하게 참조된 객체가 메모리 누수를 찾을 때 방해가 될 수 있기 때문에 일반적으로 바람직합니다. 약하게 참조된 객체에 관심이 있는 경우, 힙 워커에 해당 객체를 유지하도록 지시할 수 있습니다. JVM의 네 가지 약한 참조 유형은 "소프트", "약한", "팬텀" 및 "파이널라이저"이며, 힙 스냅샷에서 객체를 유지하기에 충분한 참조 유형을 선택할 수 있습니다.



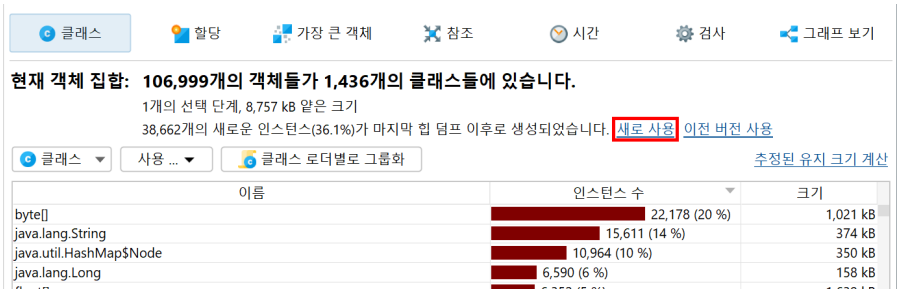
약하게 참조된 객체가 있는 경우, 힙 워커의 "약한 참조" 검사를 사용하여 현재 객체 집합에서 선택하거나 제거할 수 있습니다.

힙 마킹

종종 특정 사용 사례를 위해 할당된 객체를 보고 싶을 때가 있습니다. 할당 기록을 해당 사용 사례 주변에서 시작하고 중지하여 이를 수행할 수 있지만, 오버헤드가 훨씬 적고 다른 용도로 할당 기록 기능을 보존하는 훨씬 더 나은 방법이 있습니다: 힙 워커 개요에서 광고되고 프로파일링 메뉴 또는 트리거 작업으로도 사용할 수 있는 힙 마크 작업은 힙의 모든 객체를 "오래된" 것으로 표시합니다. 다음 힙 스냅샷을 찍을 때, 이제 "새로운" 객체가 무엇인지 명확합니다.



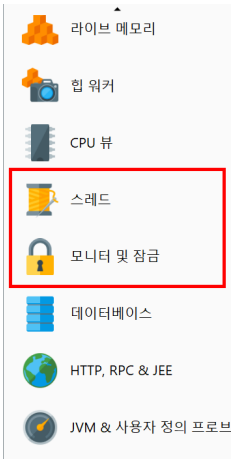
이전 힙 스냅샷이나 힙 마크 호출이 있었던 경우, 힙 워커의 제목 영역에는 새로운 인스턴스 수와 새로운 사용 및 오래된 사용이라는 두 개의 링크가 표시되어, 해당 시점 이후에 할당된 인스턴스 또는 이전에 할당된 생존 인스턴스를 선택할 수 있습니다. 이 정보는 각 객체 집합에 대해 제공되므로, 먼저 드릴다운하고 나중에 새로운 또는 오래된 인스턴스를 선택할 수 있습니다.


클래스
할당
가장 큰 객체
참조
시간
검사
그래프 보기
현재 객체 집합: 106,999개의 객체들가 1,436개의 클래스들에 있습니다.
1개의 선택 단계, 8,757 kB 얕은 크기
38,662개의 새로운 인스턴스(36.1%)가 마지막 힙 덤프 이후로 생성되었습니다. 새로 사용 이전 버전 사용
클래스
사용 ...
클래스 로더별로 그룹화
추정된 유지 크기 계산
이름
인스턴스 수
크기
byte[] 22,178 (20 %) 1,021 kB
java.lang.String 15,611 (14 %) 374 kB
java.util.HashMap$Node 10,964 (10 %) 350 kB
java.lang.Long 6,590 (6 %) 158 kB

스레드 프로파일링

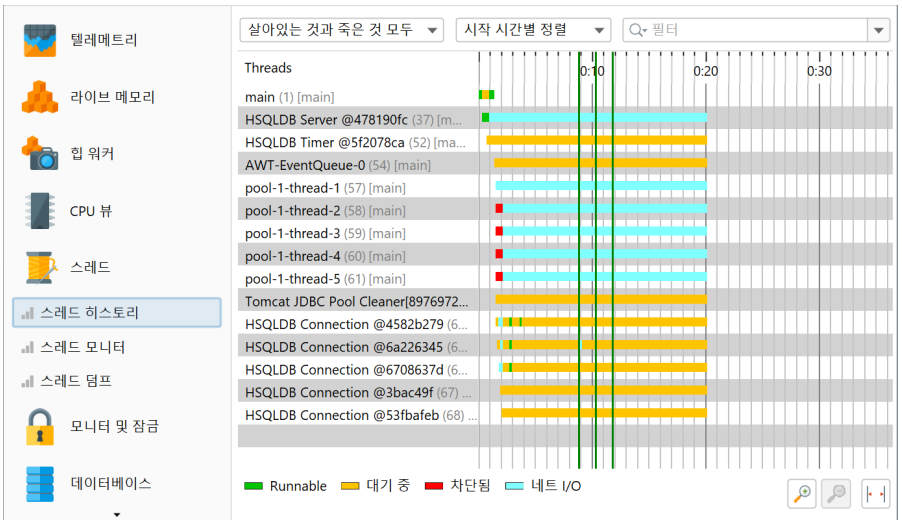
스레드를 잘못 사용하면 여러 가지 문제를 일으킬 수 있습니다. 너무 많은 활성 스레드는 스레드 기아를 초래할 수 있으며, 스레드가 서로를 차단하여 애플리케이션의 활성도에 영향을 미치거나 잘못된 순서로 잠금을 획득하면 교착 상태로 이어질 수 있습니다. 또한, 스레드에 대한 정보는 디버깅 목적으로 중요합니다.

JProfiler에서 스레드 프로파일링은 두 개의 뷰 섹션으로 나뉩니다: "Threads" 섹션은 스레드의 생명 주기와 스레드 덤프 캡처를 다루고, "Monitors & locks" 섹션은 여러 스레드의 상호 작용을 분석하는 기능을 제공합니다.



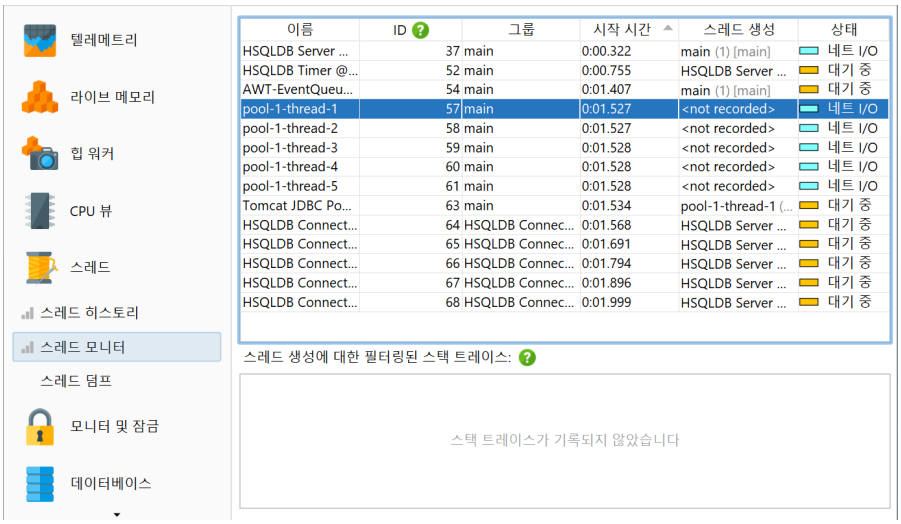
스레드 검사

스레드 히스토리 뷰는 각 스레드를 타임라인에서 색상으로 표시된 행으로 보여주며, 색상은 기록된 스레드 상태를 나타냅니다. 스레드는 생성 시간, 이름 또는 스레드 그룹에 따라 정렬되며 이름으로 필터링할 수 있습니다. 또한, 드래그 앤 드롭을 통해 스레드의 순서를 직접 재배열할 수 있습니다. 모니터 이벤트가 기록된 경우, "Waiting" 또는 "Blocked" 상태에 있는 스레드의 일부에 마우스를 올리면 모니터 히스토리 뷰로 연결되는 관련 스택 트레이스를 볼 수 있습니다.



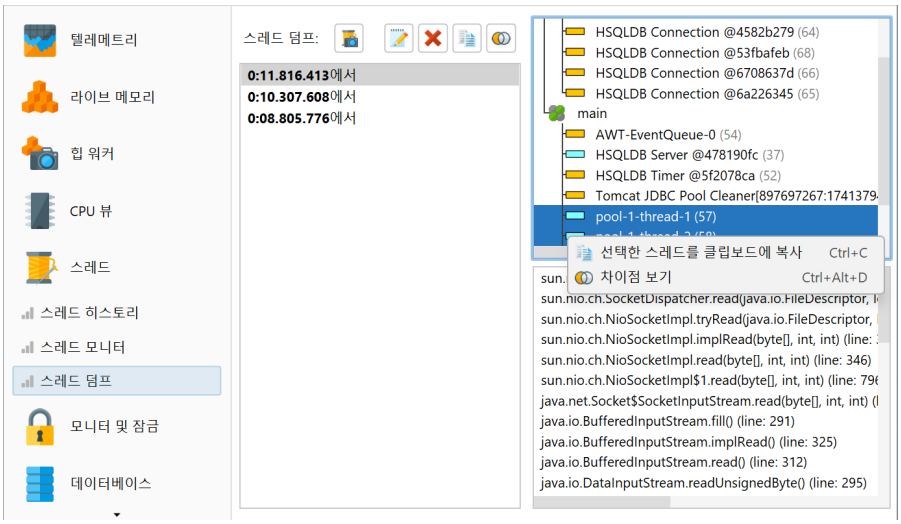
모든 스레드의 표 형식 뷰는 스레드 모니터 뷰에서 사용할 수 있습니다. 스레드가 생성되는 동안 CPU 기록이 활성화되어 있으면, JProfiler는 생성 스레드의 이름을 저장하고 테이블에 표시합니다. 하단에는 생성 스레드의 스택 트레이스가 표시됩니다. 성능상의 이유로 JVM에서 실제 스택 트레이스를 요청하지 않고 CPU 기록의

현재 정보를 사용합니다. 이는 스택 트레이스가 호출 트리 수집에 대한 필터 설정을 만족하는 클래스만 표시됨을 의미합니다.



프로파일링 설정에서 추정된 CPU 시간을 기록하도록 설정하면, CPU Time 열이 테이블에 추가됩니다. CPU 시간은 CPU 데이터를 기록할 때만 측정됩니다.

대부분의 디버거와 마찬가지로, JProfiler는 스레드 덤프를 가져올 수도 있습니다. 스레드 덤프의 스택 트레이스는 JVM에서 제공하는 전체 스택 트레이스이며 CPU 기록에 의존하지 않습니다. 두 개의 스레드 덤프를 선택하고 Show Difference 버튼을 클릭하면 diff 뷰어에서 다른 스레드 덤프를 비교할 수 있습니다. 또한, 단일 스레드 덤프에서 두 스레드를 선택하고 컨텍스트 메뉴에서 Show Difference를 선택하여 비교할 수도 있습니다.

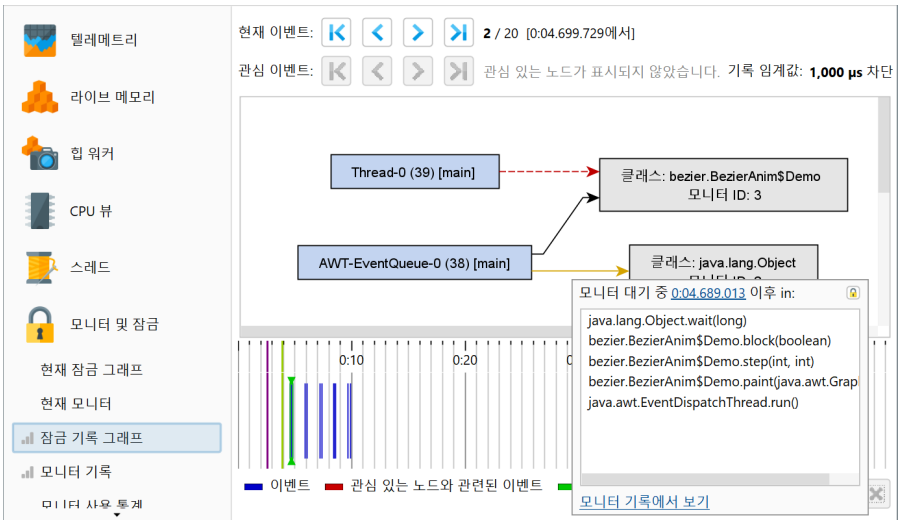


스레드 덤프는 "Trigger thread dump" 트리거 액션이나 API를 통해서도 가져올 수 있습니다.

잠금 상황 분석

모든 Java 객체에는 두 가지 동기화 작업에 사용할 수 있는 관련 모니터가 있습니다: 스레드는 다른 스레드가 모니터에 대한 알림을 발행할 때까지 모니터에서 대기할 수 있거나, 모니터에 대한 잠금을 획득하여 다른 스레드가 잠금 소유권을 포기할 때까지 차단될 수 있습니다. 또한, Java는 `java.util.concurrent.locks` 패키지에서 더 고급 잠금 전략을 구현하기 위한 클래스를 제공합니다. 해당 패키지의 잠금은 객체의 모니터를 사용하지 않고 다른 네이티브 구현을 사용합니다.

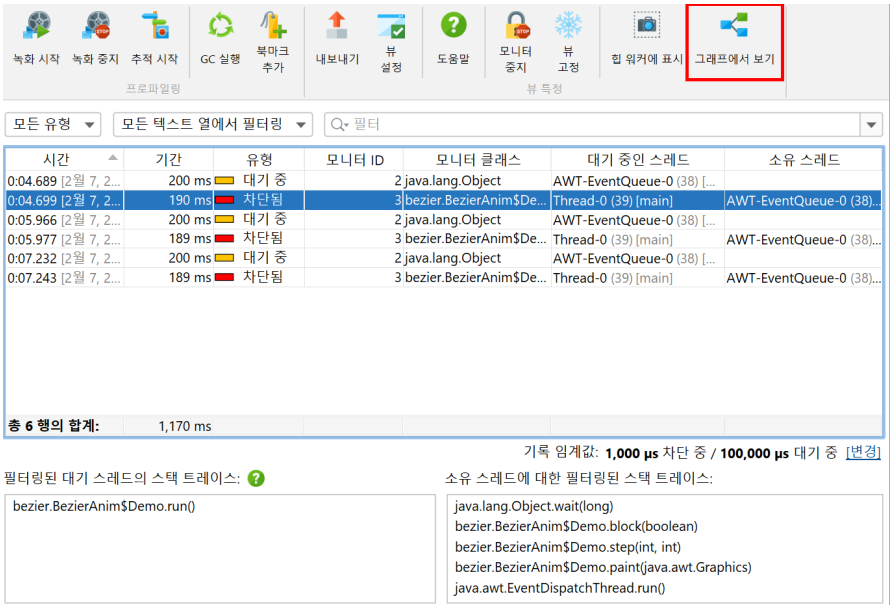
JProfiler는 위의 두 메커니즘 모두에 대한 잠금 상황을 기록할 수 있습니다. 잠금 상황에서는 하나 이상의 스레드, 모니터 또는 `java.util.concurrent.locks.Lock` 인스턴스와 특정 시간 동안 대기 또는 차단 작업이 있습니다. 이러한 잠금 상황은 모니터 히스토리 뷰에서 표 형식으로, 잠금 히스토리 그래프에서 시각적으로 표시됩니다.



잠금 히스토리 그래프는 고립된 모니터 이벤트의 지속 시간보다는 관련된 모든 모니터와 스레드의 전체 관계 집합에 중점을 둡니다. 잠금 상황에 참여하는 스레드와 모니터는 파란색과 회색 직사각형으로 표시되며, 교착 상태의 일부인 경우 빨간색으로 표시됩니다. 검은색 화살표는 모니터의 소유권을 나타내고, 노란색 화살표는 대기 중인 스레드에서 관련 모니터로 확장되며, 점선 빨간색 화살표는 스레드가 모니터를 획득하려고 하며 현재 차단 중임을 나타냅니다. CPU 데이터가 기록된 경우 차단 또는 대기 화살표 위에 마우스를 올리면 스택 트

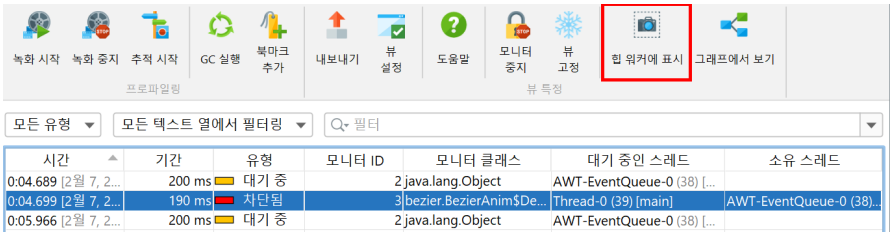
레이스를 볼 수 있습니다. 이러한 도구 팁에는 모니터 히스토리 뷰의 해당 행으로 이동하는 하이퍼링크가 포함되어 있습니다.

표 형식의 모니터 히스토리 뷰는 모니터 이벤트를 보여줍니다. 이들은 열로 표시되는 지속 시간을 가지므로 테이블을 정렬하여 가장 중요한 이벤트를 찾을 수 있습니다. 표 형식 뷰에서 선택한 행에 대해 Show in Graph 액션을 사용하여 그래프로 이동할 수 있습니다.



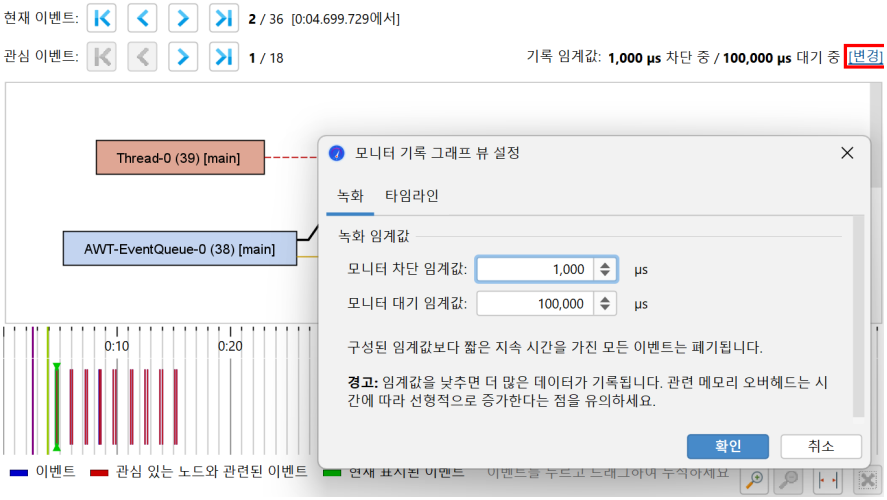
각 모니터 이벤트에는 관련 모니터가 있습니다. Monitor Class 열은 모니터가 사용되는 인스턴스의 클래스 이름을 표시하거나 모니터와 관련된 Java 객체가 없는 경우 "[raw monitor]"를 표시합니다. 어떤 경우든 모니터에는 별도의 열에 표시되는 고유 ID가 있어 여러 이벤트에 걸쳐 동일한 모니터의 사용을 상관시킬 수 있습니다. 각 모니터 이벤트에는 작업을 수행하는 대기 스레드와 선택적으로 작업을 차단하는 소유 스레드가 있습니다. 사용 가능한 경우, 그들의 스택 트레이스는 뷰의 하단에 표시됩니다.

모니터 인스턴스에 대한 추가 질문이 있는 경우, 모니터 히스토리 뷰와 잠금 히스토리 그래프 모두에서 Show in Heap Walker 액션을 사용하여 힙 워커로 연결하고 모니터 인스턴스를 새로운 객체 집합으로 선택할 수 있습니다.



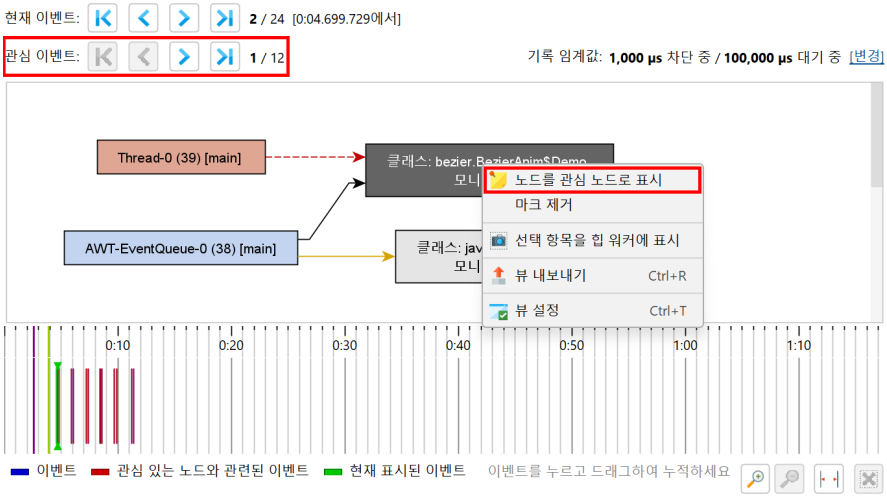
관심 이벤트 제한

모니터 이벤트를 분석하는 데 있어 근본적인 문제 중 하나는 애플리케이션이 비정상적으로 많은 모니터 이벤트를 생성할 수 있다는 것입니다. 그래서 JProfiler는 대기 및 차단 이벤트에 대해 기본 임계값을 설정하여 이벤트가 즉시 폐기되도록 합니다. 이러한 임계값은 뷰 설정에서 정의되며 더 긴 이벤트에 집중하기 위해 증가시킬 수 있습니다.



기록된 이벤트에 대해 추가로 필터를 적용할 수 있습니다. 모니터 히스토리 뷰는 뷰 상단에 임계값, 이벤트 유형 및 텍스트 필터를 제공합니다. 잠금 히스토리 그래프에서는 관심 있는 스레드나 모니터를 선택하고 표시된 엔티티가 포함된 잠금 상황만 표시할 수 있습니다. 관심 이벤트는 타임라인에서 다른 색상으로 표시되며, 해당 이벤트를 단계별로 탐색할 수 있는 보조 탐색 막대가 있습니다. 현재 이벤트가 관심 이벤트가 아닌 경우, 현재 이벤트와 다음 관심 이벤트 사이에 얼마나 많은 이벤트가 있는지 양방향으로 볼 수 있습니다.

선택한 스레드나 모니터가 있는 잠금 상황 외에도 그래프에서 제거된 잠금 상황도 표시됩니다. 이는 각 모니터 이벤트가 작업이 시작된 잠금 상황과 종료된 잠금 상황 두 가지로 정의되기 때문입니다. 또한, 완전히 비어 있는 그래프는 JVM에 더 이상 잠금이 없음을 나타내는 유효한 잠금 상황입니다.



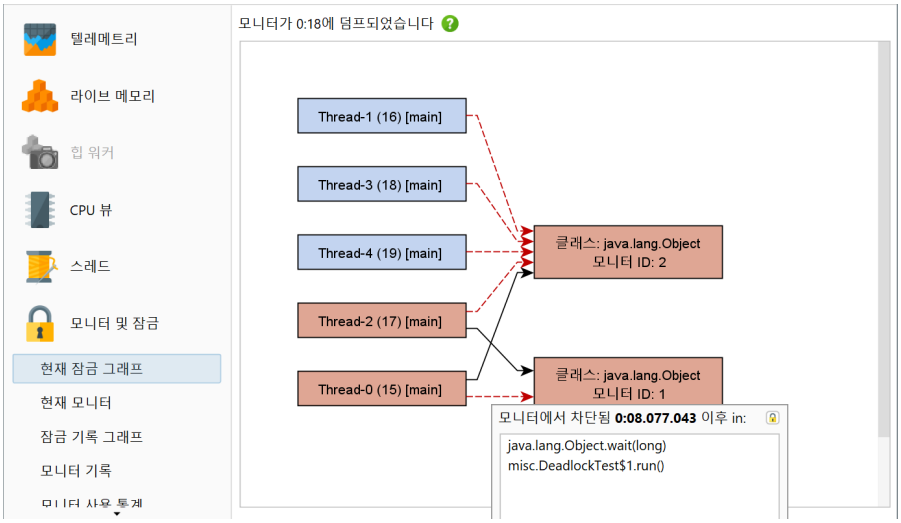
주의를 기울여야 할 이벤트 수를 줄이는 또 다른 전략은 잠금 상황을 누적하는 것입니다. 잠금 히스토리 그래프에서는 하단에 모든 기록된 이벤트를 보여주는 타임라인이 있습니다. 클릭하고 드래그하여 시간 범위를 선택하면 포함된 모든 이벤트의 데이터가 위의 잠금 그래프에 표시됩니다. 누적된 그래프에서는 각 화살표에 동일한 유형의 여러 이벤트가 포함될 수 있습니다. 이 경우 도구 팁 창에는 이벤트 수와 포함된 모든 이벤트의 총 시간이 표시됩니다. 도구 팁 창의 드롭다운 목록에서 타임스탬프를 표시하고 다른 이벤트 간에 전환할 수 있습니다.

교착 상태 감지

"Current locking graph"와 "Current monitors" 뷰는 JProfiler UI에서 액션으로 트리거되는 "모니터 덤프"에서 작동합니다. 모니터 덤프를 사용하면 아직 진행 중인 이벤트를 검사할 수 있습니다. 여기에는 결코 완료되지 않고 히스토리 뷰에 표시될 수 없는 교착 상태가 포함됩니다.

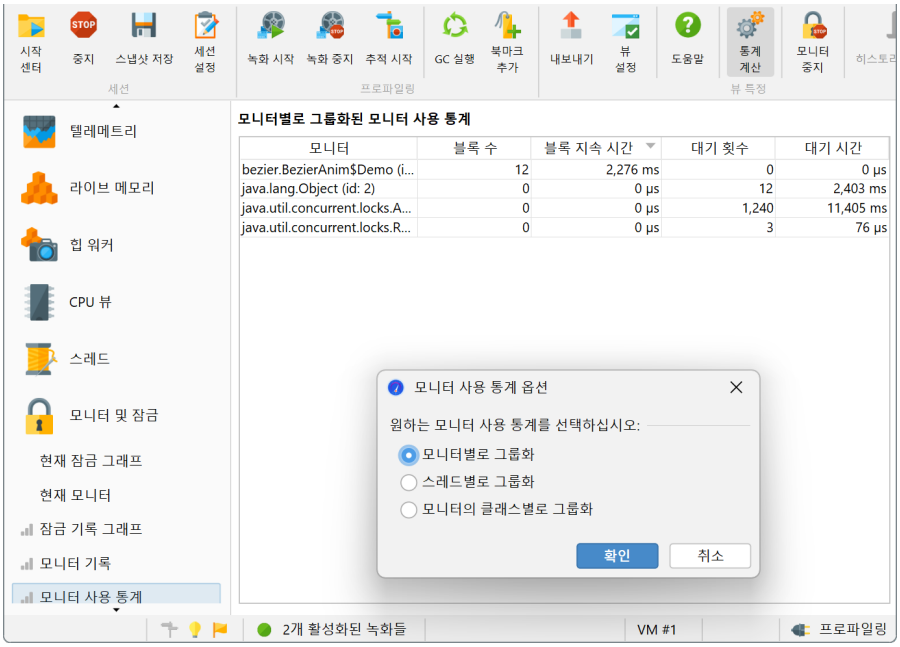
차단 작업은 일반적으로 단명하지만, 교착 상태가 발생하면 두 뷰 모두 문제의 영구적인 뷰를 표시합니다. 또한, 현재 잠금 그래프는 교착 상태를 일으키는 스레드와 모니터를 빨간색으로 표시하여 이러한 문제를 즉시 파악할 수 있습니다.

새로운 모니터 덤프를 가져오면 두 뷰의 데이터가 교체됩니다. "Trigger monitor dump" 트리거 액션이나 API를 통해 모니터 덤프를 트리거할 수도 있습니다.



모니터 사용 통계

차단 및 대기 작업을 더 높은 관점에서 조사하기 위해 모니터 통계 뷰는 모니터 기록 데이터에서 보고서를 계산합니다. 모니터 이벤트를 모니터, 스레드 이름 또는 모니터 클래스별로 그룹화하고 각 행에 대한 누적 횟수와 지속 시간을 분석할 수 있습니다.



프로브

CPU 및 메모리 프로파일링은 주로 객체와 메서드 호출, 즉 JVM에서 애플리케이션의 기본 빌딩 블록에 중점을 둡니다. 일부 기술에서는 실행 중인 애플리케이션에서 의미론적 데이터를 추출하여 프로파일러에 표시하는 보다 고급의 접근 방식이 필요합니다.

가장 두드러진 예는 JDBC를 사용한 데이터베이스 호출 프로파일링입니다. 호출 트리는 JDBC API를 사용할 때와 그 호출이 얼마나 오래 걸리는지를 보여줍니다. 그러나 각 호출에 대해 다른 SQL 문이 실행될 수 있으며, 이러한 호출 중 어느 것이 성능 병목 현상의 원인인지 알 수 없습니다. 또한, JDBC 호출은 애플리케이션의 여러 다른 위치에서 발생하는 경우가 많으며, 일반 호출 트리에서 검색하지 않고 모든 데이터베이스 호출을 보여주는 단일 뷰가 필요합니다.

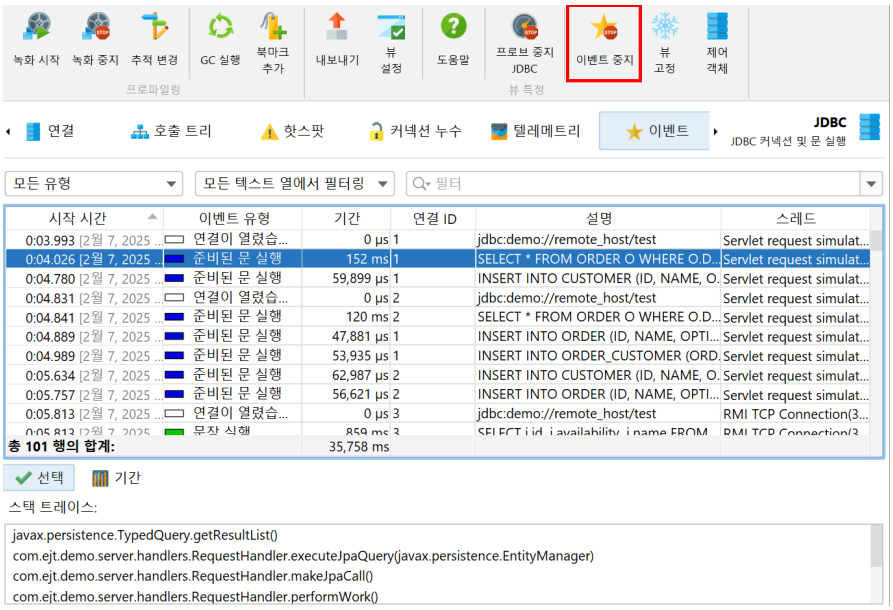
이 문제를 해결하기 위해 JProfiler는 JRE의 중요한 하위 시스템에 대한 여러 프로브를 제공합니다. 프로브는 특정 클래스에 계측을 추가하여 데이터를 수집하고 "Databases" 및 "JEE & Probes" 뷰 섹션의 전용 뷰에 표시합니다. 또한, 프로브는 호출 트리에 데이터를 주석으로 추가하여 일반 CPU 프로파일링과 고급 데이터를 동시에 볼 수 있습니다.

JProfiler에서 직접 지원하지 않는 기술에 대한 추가 정보를 얻고자 하는 경우, 해당 기술에 대한 사용자 정의 프로브 [p. 153]를 작성할 수 있습니다. 일부 라이브러리, 컨테이너 또는 데이터베이스 드라이버는 자체 임베디드 프로브 [p. 158]를 제공할 수 있으며, 애플리케이션에서 사용될 때 JProfiler에 표시됩니다.

프로브 이벤트

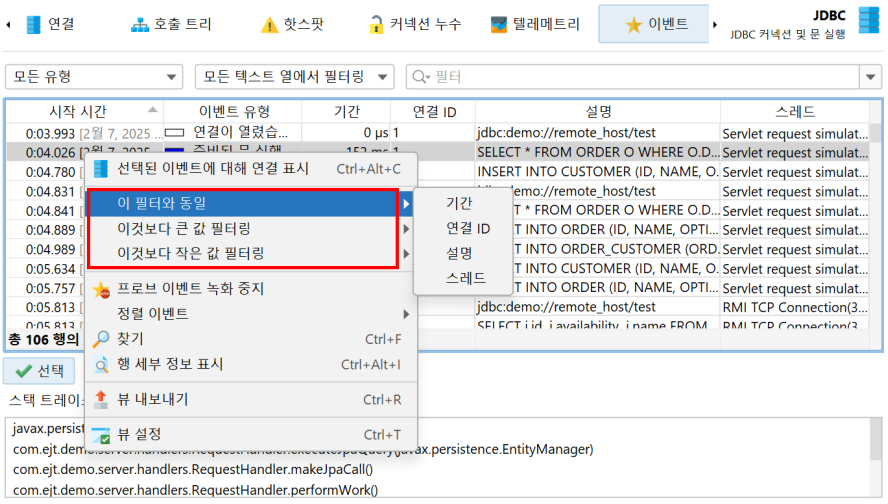
프로브는 오버헤드를 추가하기 때문에 기본적으로 기록되지 않으며, 각 프로브에 대해 수동 또는 자동으로 녹화를 시작 [p. 26]해야 합니다.

프로브의 기능에 따라 프로브 데이터는 여러 뷰에 표시됩니다. 가장 낮은 수준은 프로브 이벤트입니다. 다른 뷰는 프로브 이벤트를 누적하여 데이터를 보여줍니다. 기본적으로 프로브 이벤트는 프로브가 기록 중일 때도 유지되지 않습니다. 단일 이벤트가 중요해지면 프로브 이벤트 뷰에서 기록할 수 있습니다. 파일 프로브와 같은 일부 프로브의 경우 일반적으로 높은 비율로 이벤트를 생성하기 때문에 권장되지 않습니다. "HTTP 서버" 프로브나 JDBC 프로브와 같은 다른 프로브는 훨씬 낮은 비율로 이벤트를 생성하므로 단일 이벤트를 기록하는 것이 적절할 수 있습니다.



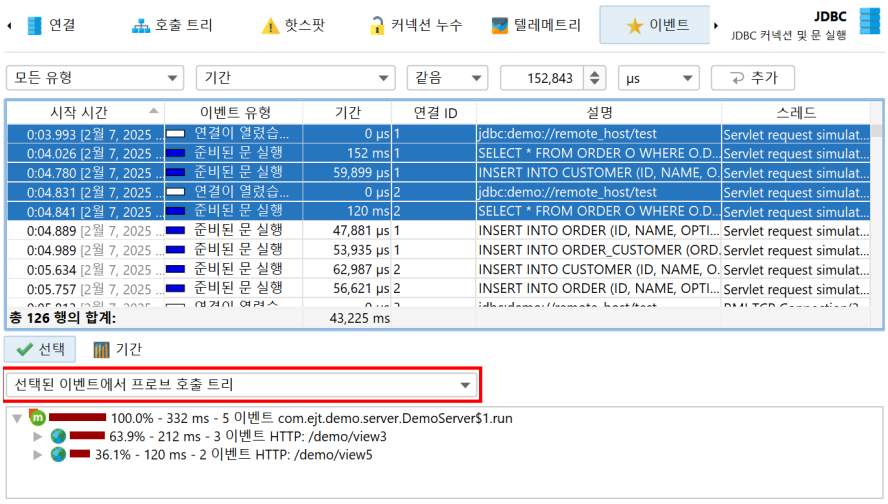
프로브 이벤트는 메서드 매개변수, 반환 값, 계측된 객체 및 발생한 예외를 포함한 다양한 소스에서 프로브 문자열을 캡처합니다. 예를 들어, JDBC 프로브는 실제 SQL 문자열을 구성하기 위해 준비된 문서에 대한 모든 setter 호출을 가로채야 하므로 여러 메서드 호출에서 데이터를 수집할 수 있습니다. 프로브 문자열은 프로브에 의해 측정된 고급 하위 시스템에 대한 기본 정보입니다. 또한, 이벤트에는 시작 시간, 선택적 지속 시간, 관련 스레드 및 스택 트레이스가 포함됩니다.

테이블 하단에는 표시된 이벤트의 총 수를 보여주고 테이블의 모든 숫자 열을 합산하는 특별한 행이 있습니다. 기본 열의 경우, 이는 지속 시간 열만 포함됩니다. 테이블 위의 필터 선택기와 함께 선택된 이벤트 하위 집합에 대한 수집된 데이터를 분석할 수 있습니다. 기본적으로 텍스트 필터는 모든 텍스트 필드 열에서 작동하지만, 텍스트 필드 앞의 드롭다운에서 특정 필터 열을 선택할 수 있습니다. 예를 들어, 선택한 이벤트의 지속 시간보다 큰 모든 이벤트를 필터링하기 위해 컨텍스트 메뉴에서 필터 옵션을 사용할 수도 있습니다.

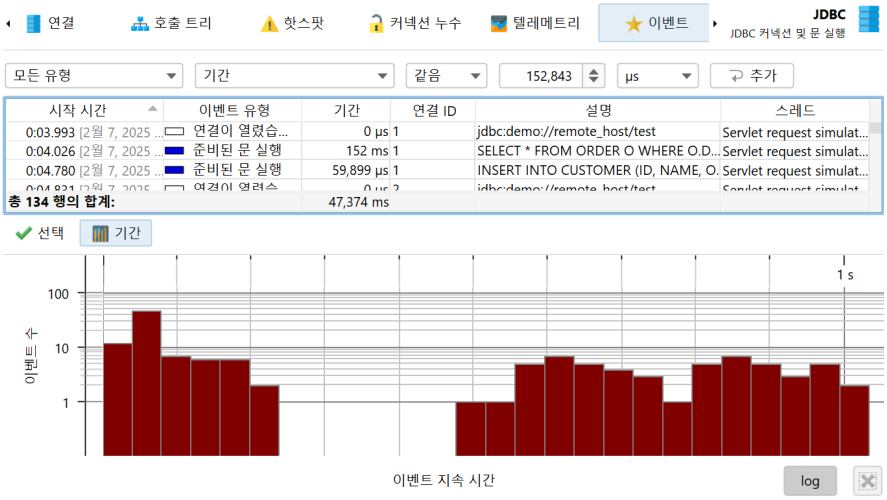


다른 프로브 뷰에서도 프로브 이벤트를 필터링할 수 있는 옵션을 제공합니다: 프로브 텔레메트리 뷰에서는 시간 범위를 선택할 수 있고, 프로브 호출 트리 뷰에서는 선택한 호출 스택에서 이벤트를 필터링할 수 있으며, 프로브 핫스팟 뷰는 선택한 백 트레이스 또는 핫스팟을 기반으로 프로브 이벤트 필터를 제공합니다. 제어 객체 및 타임 라인 뷰는 선택한 제어 객체에 대한 프로브 이벤트를 필터링하는 작업을 제공합니다.

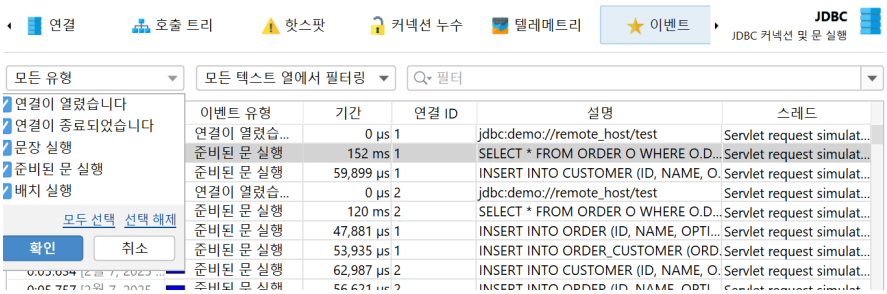
선택한 프로브 이벤트의 스택 트레이스는 하단에 표시됩니다. 여러 프로브 이벤트가 선택된 경우, 스택 트레이스는 누적되어 호출 트리, 백 트레이스가 있는 프로브 핫스팟 또는 백 트레이스가 있는 CPU 핫스팟으로 표시됩니다.



스택 트레이스 뷰 옆에는 이벤트 지속 시간 및 선택적으로 기록된 처리량에 대한 히스토그램 뷰가 표시됩니다. 마우스를 사용하여 이러한 히스토그램에서 지속 시간 범위를 선택하여 위의 테이블에서 프로브 이벤트를 필터링할 수 있습니다.



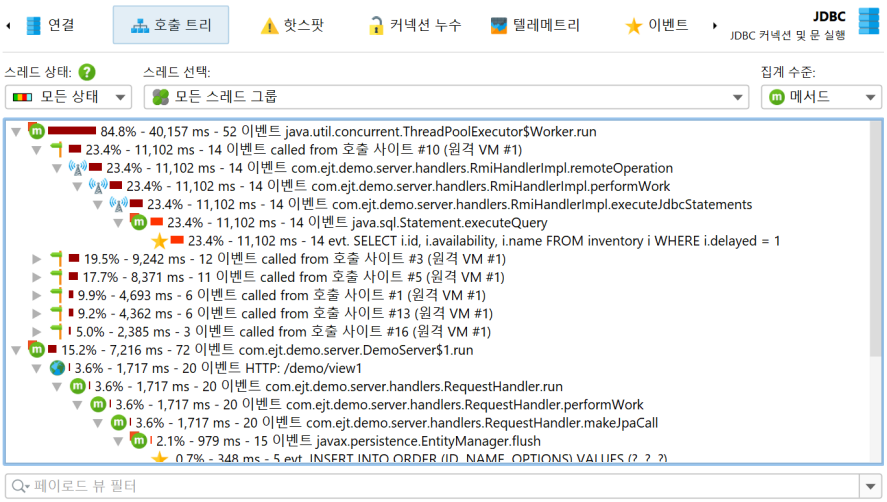
프로브는 다양한 종류의 활동을 기록하고 프로브 이벤트에 이벤트 유형을 연결할 수 있습니다. 예를 들어, JDBC 프로브는 문서, 준비된 문서 및 배치 실행을 다른 색상으로 표시합니다.



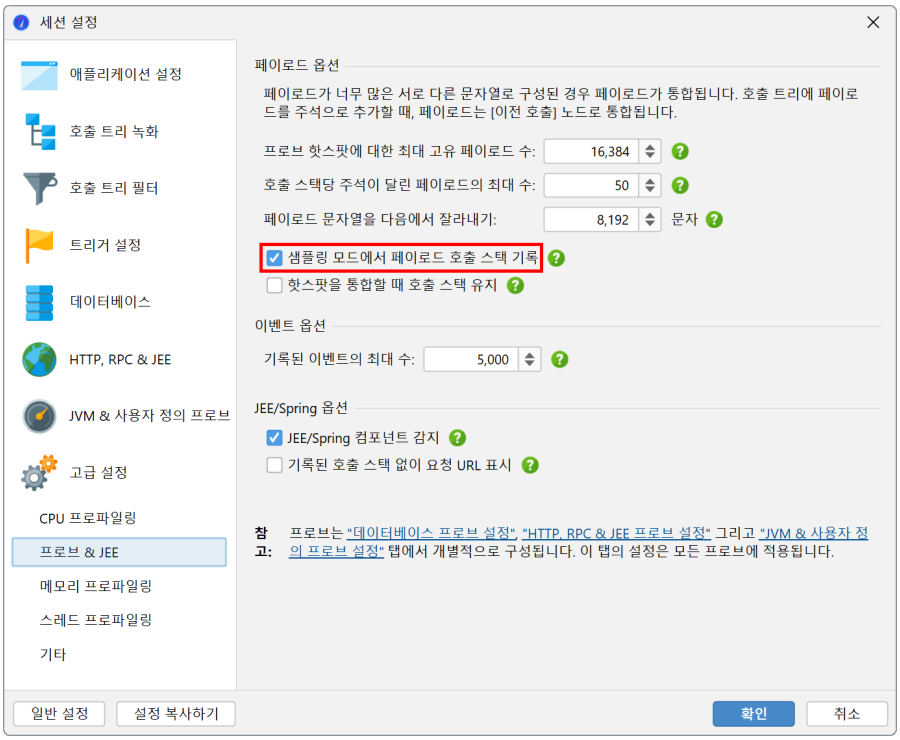
단일 이벤트가 기록될 때 과도한 메모리 사용을 방지하기 위해 JProfiler는 이벤트를 통합합니다. 이벤트 캡은 프로파일링 설정에서 구성되며 모든 프로브에 적용됩니다. 가장 최근의 이벤트만 유지되며, 오래된 이벤트는 삭제됩니다. 이 통합은 고급 뷰에 영향을 미치지 않습니다.

프로브 호출 트리 및 핫스팟

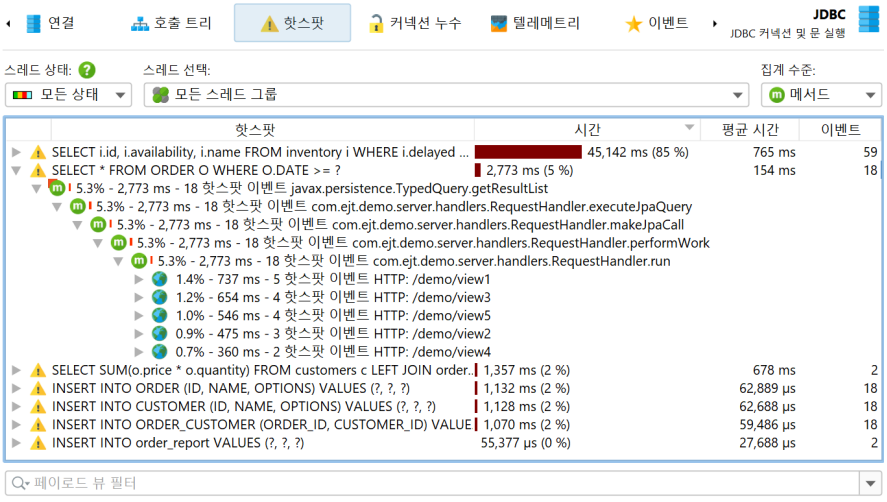
프로브 녹화는 CPU 녹화와 긴밀히 협력합니다. 프로브 이벤트는 프로브 호출 트리에 집계되며, 프로브 문자열은 "페이로드"라고 불리는 리프 노드입니다. 프로브 이벤트가 생성된 호출 스택만 해당 트리에 포함됩니다. 메서드 노드에 대한 정보는 기록된 페이로드 이름을 참조합니다. 예를 들어, 특정 호출 스택에서 SQL 문이 총 9000ms 동안 42회 실행된 경우, 이는 모든 상위 호출 트리 노드에 이벤트 수 42와 시간 9000ms를 추가합니다. 기록된 모든 페이로드의 누적은 프로브 특정 시간을 가장 많이 소비하는 호출 경로를 보여주는 호출 트리를 형성합니다. 프로브 트리의 초점은 페이로드에 있으며, 뷰 필터는 기본적으로 페이로드를 검색하지만, 컨텍스트 메뉴에서도 클래스를 필터링하는 모드를 제공합니다.



CPU 녹화가 꺼져 있으면, 백 트레이스에는 "No CPU data was recorded" 노드만 포함됩니다. CPU 데이터가 부분적으로만 기록된 경우, 이러한 노드와 실제 백 트레이스가 혼합될 수 있습니다. 샘플링이 활성화된 경우에도 JProfiler는 기본적으로 프로브 페이로드에 대한 정확한 호출 트레이스를 기록합니다. 이 오버헤드를 피하고 싶다면, 프로파일링 설정에서 이를 끌 수 있습니다. 데이터 수집을 늘리거나 오버헤드를 줄이기 위해 조정할 수 있는 여러 다른 프로브 녹화 조정 옵션이 있습니다.



핫스팟은 프로브 호출 트리에서 계산할 수 있습니다. 핫스팟 노드는 이제 CPU 뷰 섹션 [p. 51]과 같은 메서드 호출이 아니라 페이로드입니다. 이는 종종 프로브의 가장 즉각적으로 유용한 뷰입니다. CPU 녹화가 활성화된 경우, 최상위 핫스팟을 열고 일반 CPU 핫스팟 뷰와 마찬가지로 메서드 백트레이스를 분석할 수 있습니다. 백트레이스 노드의 숫자는 핫스팟 바로 아래의 노드에서 가장 깊은 노드까지 확장된 호출 스택을 따라 측정된 프로브 이벤트가 몇 개이며 총 지속 시간이 얼마인지 나타냅니다.



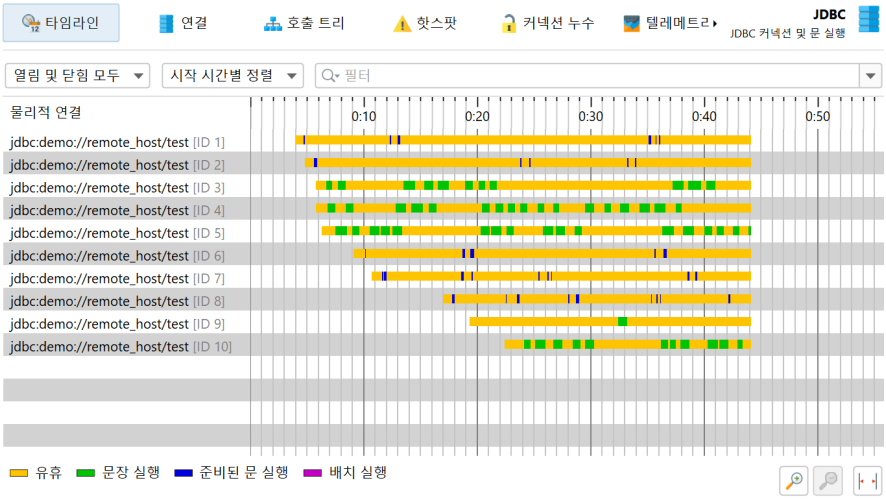
프로브 호출 트리와 프로브 핫스팟 뷰 모두 스레드 또는 스레드 그룹, 스레드 상태 및 메서드 노드에 대한 집계 수준을 선택할 수 있습니다. CPU 뷰와 비교하여 데이터를 비교할 때, 프로브 뷰의 기본 스레드 상태가 "Runnable"이 아닌 "All states"임을 염두에 두는 것이 중요합니다. 이는 프로브 이벤트가 종종 데이터베이스 호출, 소켓 작업 또는 프로세스 실행과 같은 외부 시스템을 포함하기 때문에 현재 JVM이 작업한 시간뿐만 아니라 총 시간을 보는 것이 중요하기 때문입니다.

제어 객체

외부 리소스에 대한 액세스를 제공하는 많은 라이브러리는 리소스와 상호 작용하는 데 사용할 수 있는 연결 객체를 제공합니다. 예를 들어, 프로세스를 시작할 때 `java.lang.Process` 객체를 사용하여 출력 스트림에서 읽고 입력 스트림에 쓸 수 있습니다. JDBC를 사용할 때는 SQL 쿼리를 수행하기 위해 `java.sql.Connection` 객체가 필요합니다. JProfiler에서 이 종류의 객체에 사용되는 일반 용어는 "제어 객체"입니다.

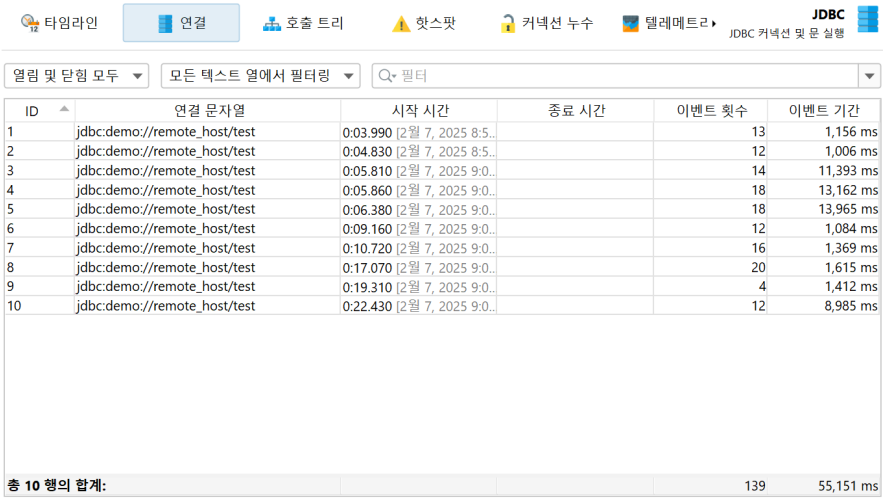
프로브 이벤트를 제어 객체와 함께 그룹화하고 그들의 생명 주기를 보여주는 것은 문제의 원인을 더 잘 이해하는 데 도움이 될 수 있습니다. 또한, 제어 객체를 생성하는 것은 종종 비용이 많이 들기 때문에 애플리케이션이 너무 많이 생성하지 않고 적절히 닫도록 하는 것이 중요합니다. 이를 위해 제어 객체를 지원하는 프로브는 "타임 라인" 및 "제어 객체" 뷰를 제공하며, 후자는 예를 들어 JDBC 프로브의 경우 "Connections"와 같이 더 구체적으로 명명될 수 있습니다. 제어 객체가 열리거나 닫힐 때, 프로브는 이벤트 뷰에 표시되는 특별한 프로브 이벤트를 생성하여 관련 스택 트레이스를 검사할 수 있습니다.

타임 라인 뷰에서는 각 제어 객체가 막대로 표시되며, 색상은 제어 객체가 활성화된 시점을 나타냅니다. 프로브는 다양한 이벤트 유형을 기록할 수 있으며 타임 라인은 이에 따라 색상이 지정됩니다. 이 상태 정보는 이벤트 목록에서 가져오지 않으며, 통합되거나 사용할 수 없을 수도 있지만, 마지막 상태에서 매 100ms마다 샘플링됩니다. 제어 객체에는 식별할 수 있는 이름이 있습니다. 예를 들어, 파일 프로브는 파일 이름으로 제어 객체를 생성하고, JDBC 프로브는 연결 문자열을 제어 객체의 이름으로 표시합니다.



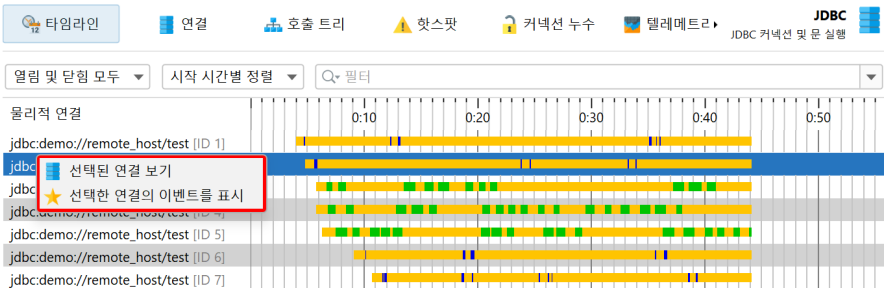
제어 객체 뷰는 모든 제어 객체를 표 형식으로 보여줍니다. 기본적으로 열려 있거나 닫힌 제어 객체가 모두 표시됩니다. 상단의 컨트롤을 사용하여 열려 있거나 닫힌 제어 객체만 표시하거나 특정 열의 내용을 필터링할 수 있습니다. 제어 객체의 기본 생명 주기 데이터 외에도, 테이블은 각 제어 객체의 누적 활동에 대한 데이터를 보여줍니다. 예를 들어, 이벤트 수와 평균 이벤트 지속 시간이 포함됩니다.

다른 프로브는 여기에서 다른 열을 표시합니다. 예를 들어, 프로세스 프로브는 읽기 및 쓰기 이벤트에 대한 별도의 열 세트를 표시합니다. 이 정보는 단일 이벤트 기록이 비활성화된 경우에도 사용할 수 있습니다. 이벤트 뷰와 마찬가지로, 하단의 총 행은 필터링과 함께 사용하여 제어 객체의 부분 집합에 대한 누적 데이터를 얻을 수 있습니다.



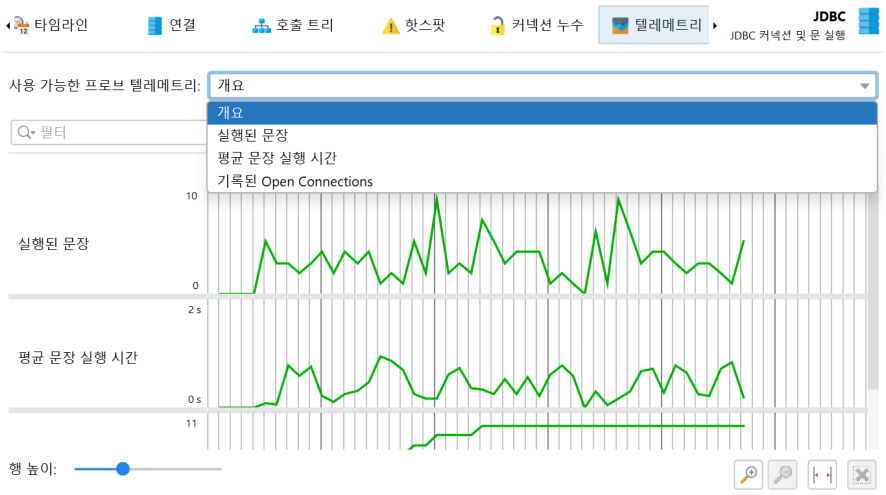
프로브는 중첩된 테이블에서 특정 속성을 게시할 수 있습니다. 이는 메인 테이블의 정보 과부하를 줄이고 테이블 열에 더 많은 공간을 제공하기 위해 수행됩니다. 파일 및 프로세스 프로브와 같은 중첩 테이블이 있는 경우, 각 행에는 왼쪽에 확장 핸들이 있어 속성-값 테이블을 열 수 있습니다.

타임 라인, 제어 객체 뷰 및 이벤트 뷰는 탐색 작업과 연결되어 있습니다. 예를 들어, 타임 라인 뷰에서 행을 마우스 오른쪽 버튼으로 클릭하여 다른 뷰로 이동하여 선택한 제어 객체의 데이터만 표시할 수 있습니다. 이는 제어 객체 ID를 선택한 값으로 필터링하여 달성됩니다.

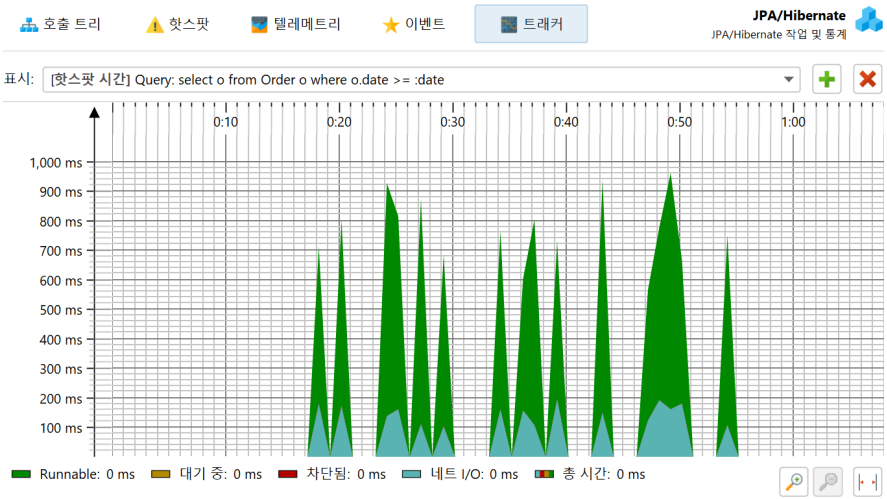


텔레메트리 및 트래커

프로브에 의해 수집된 누적 데이터에서 여러 텔레메트리가 기록됩니다. 모든 프로브에 대해 초당 프로브 이벤트 수와 평균 지속 시간 또는 I/O 작업의 처리량과 같은 프로브 이벤트에 대한 평균 측정값이 제공됩니다. 제어 객체가 있는 프로브의 경우, 열린 제어 객체의 수 또한 표준 텔레메트리입니다. 각 프로브는 추가 텔레메트리를 추가할 수 있으며, 예를 들어 JPA 프로브는 쿼리 수 및 엔티티 작업 수에 대한 별도의 텔레메트리를 보여줍니다.



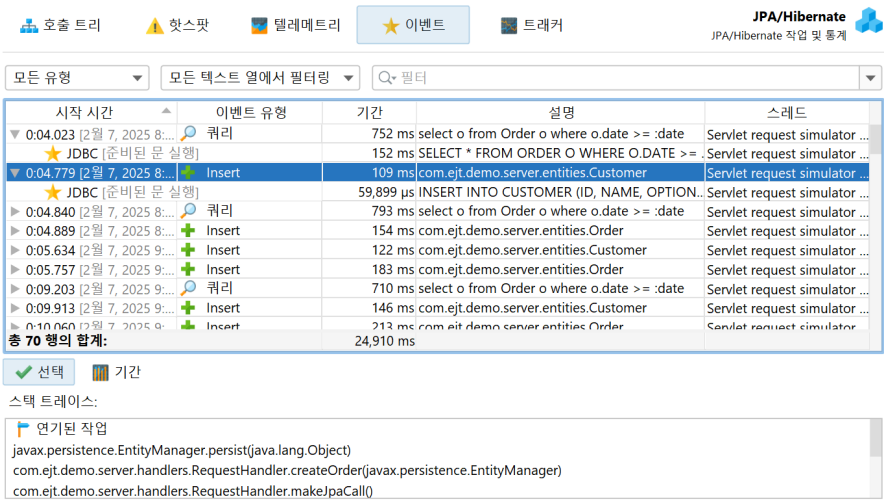
핫스팟 뷰와 제어 객체 뷰는 시간이 지남에 따라 추적할 수 있는 누적 데이터를 보여줍니다. 이러한 특별한 텔레메트리는 프로브 트래커로 기록됩니다. 추적을 설정하는 가장 쉬운 방법은 핫스팟 또는 제어 객체 뷰에서 선택 항목을 트래커에 추가 작업을 사용하여 새 텔레메트리를 추가하는 것입니다. 두 경우 모두 시간을 추적할지 수를 추적할지 선택해야 합니다. 제어 객체를 추적할 때, 텔레메트리는 모든 다른 프로브 이벤트 유형에 대한 누적 영역 그래프입니다. 추적된 핫스팟의 경우, 추적된 시간은 다른 스레드 상태로 나뉩니다.



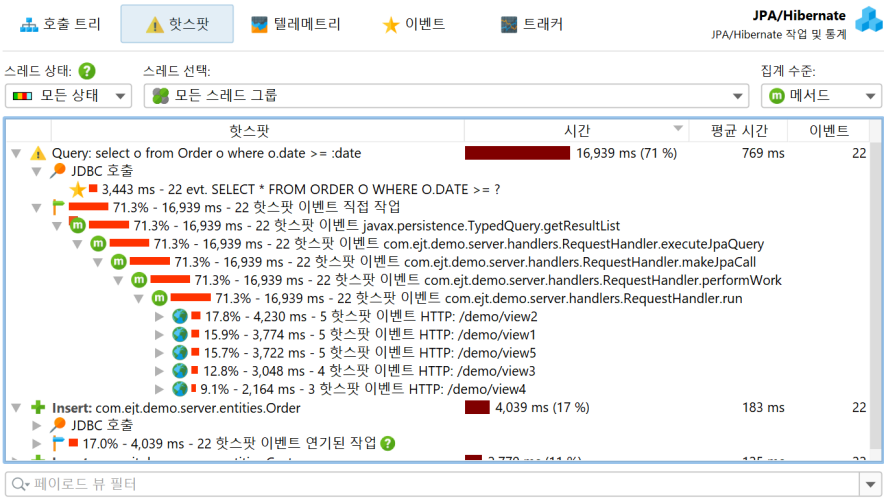
프로브 텔레메트리는 "텔레메트리" 섹션에 추가 [p. 44]하여 시스템 텔레메트리 또는 사용자 정의 텔레메트리와 비교할 수 있습니다. 그런 다음 텔레메트리 개요의 컨텍스트 메뉴 작업을 통해 프로브 녹화를 제어할 수도 있습니다.

JDBC 및 JPA

JDBC 및 JPA 프로브는 협력하여 작동합니다. JPA 프로브의 이벤트 뷰에서, JPA 프로브와 함께 JDBC 프로브가 기록된 경우, 관련된 JDBC 이벤트를 보기 위해 단일 이벤트를 확장할 수 있습니다.

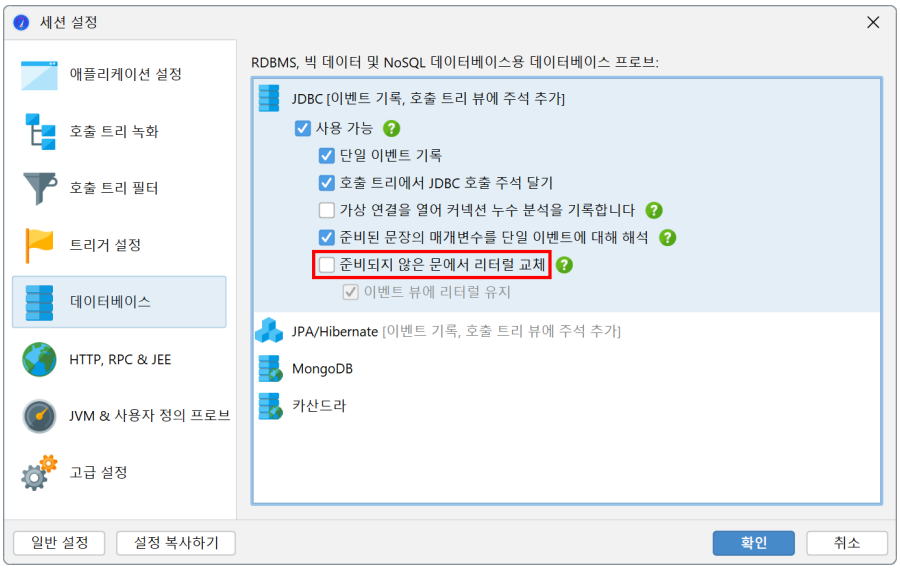


마찬가지로, 핫스팟 뷰는 JPA 작업에 의해 트리거된 JDBC 호출을 포함하는 모든 핫스팟에 특별한 "JDBC 호출" 노드를 추가합니다. 일부 JPA 작업은 비동기적이며 즉시 실행되지 않고 세션이 플러시될 때 임의의 나중 시점에 실행됩니다. 성능 문제를 찾을 때, 해당 플러시의 스택 트레이스는 도움이 되지 않으므로 JProfiler는 기존 엔티티가 획득되었거나 새 엔티티가 지속된 위치의 스택 트레이스를 기억하고 이를 프로브 이벤트에 연결합니다. 이 경우, 핫스팟의 백 트레이스는 "지연된 작업"이라는 레이블이 붙은 노드 내에 포함되며, 그렇지 않으면 "직접 작업" 노드가 삽입됩니다.



MongoDB 프로브와 같은 다른 프로브는 직접 및 비동기 작업을 모두 지원합니다. 비동기 작업은 현재 스레드에서 실행되지 않지만, 동일한 JVM의 하나 이상의 다른 스레드 또는 다른 프로세스에서 실행됩니다. 이러한 프로브의 경우, 핫스팟의 백 트레이스는 "직접 작업" 및 "비동기 작업" 컨테이너 노드로 정렬됩니다.

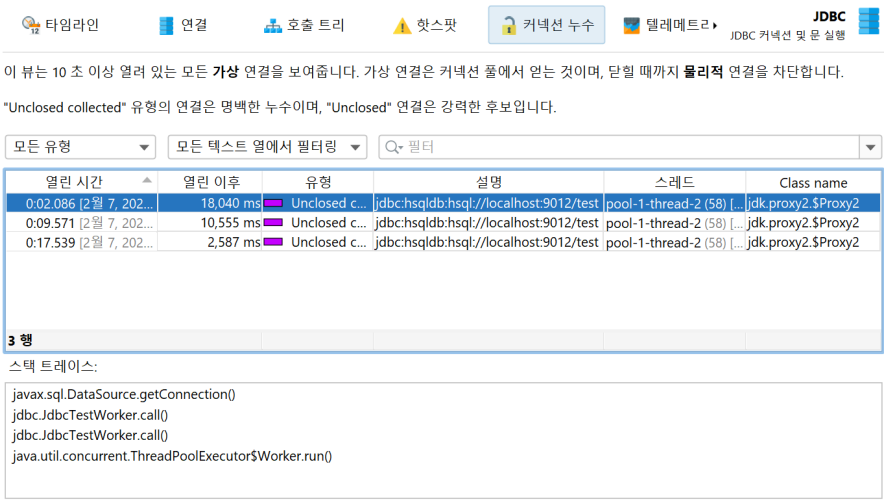
JDBC 프로브의 특별한 문제는 ID와 같은 리터럴 데이터가 SQL 문자열에 포함되지 않은 경우에만 좋은 핫스팟을 얻을 수 있다는 것입니다. 이는 준비된 문서가 사용되는 경우 자동으로 발생하지만, 일반 문서가 실행되는 경우에는 그렇지 않습니다. 후자의 경우, 대부분의 쿼리가 한 번만 실행되는 핫스팟 목록을 얻을 가능성이 높습니다. 해결책으로, JProfiler는 준비되지 않은 문서에서 리터럴을 대체하기 위한 JDBC 프로브 구성에서 기본이 아닌 옵션을 제공합니다. 디버깅 목적으로 이벤트 뷰에서 리터럴을 여전히 보고 싶을 수 있습니다. 해당 옵션을 비활성화하면 JProfiler가 너무 많은 다른 문자열을 캐시할 필요가 없으므로 메모리 오버헤드가 줄어듭니다.



반면에, JProfiler는 준비된 문서의 매개변수를 수집하고 이벤트 뷰에서 자리 표시자가 없는 완전한 SQL 문자열을 보여줍니다. 이는 디버깅 시 유용하지만, 필요하지 않은 경우 메모리를 절약하기 위해 프로브 설정에서 이를 끌 수 있습니다.

JDBC 연결 누수

JDBC 프로브에는 데이터베이스 풀로 반환되지 않은 열린 가상 데이터베이스 연결을 보여주는 "연결 누수" 뷰가 있습니다. 이는 풀링된 데이터베이스 소스에 의해 생성된 가상 연결에만 영향을 미칩니다. 가상 연결은 닫힐 때까지 물리적 연결을 차단합니다.



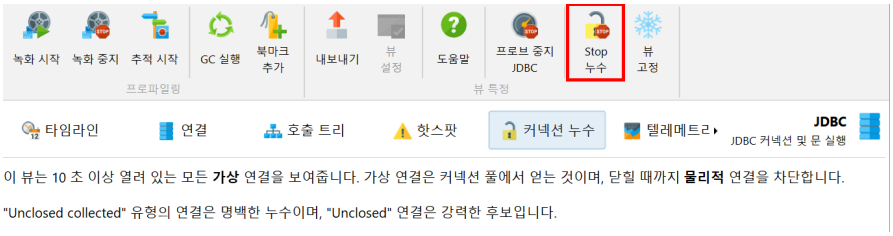
누수 후보에는 "닫히지 않은" 연결과 "수집되지 않은 닫히지 않은" 연결의 두 가지 유형이 있습니다. 두 유형 모두 데이터베이스 풀에서 제공한 연결 객체가 여전히 힙에 있지만 `close()`가 호출되지 않은 가상 연결입니다. "수집되지 않은 닫히지 않은" 연결은 가비지 수집되었으며 확실한 연결 누수입니다.

"닫히지 않은" 연결 객체는 여전히 힙에 있습니다. 열린 이후 지속 시간이 길수록 그러한 가상 연결이 누수 후보일 가능성이 높습니다. 가상 연결은 10초 이상 열려 있을 때 잠재적인 누수로 간주됩니다. 그러나 `close()`가 여전히 호출될 수 있으며, 그러면 "연결 누수" 뷰에서 항목이 제거됩니다.

연결 누수 테이블에는 연결 클래스의 이름을 보여주는 클래스 이름 열이 포함되어 있습니다. 이를 통해 어떤 유형의 풀이 연결을 생성했는지 알 수 있습니다. JProfiler는 많은 수의 데이터베이스 드라이버 및 연결 풀을 명

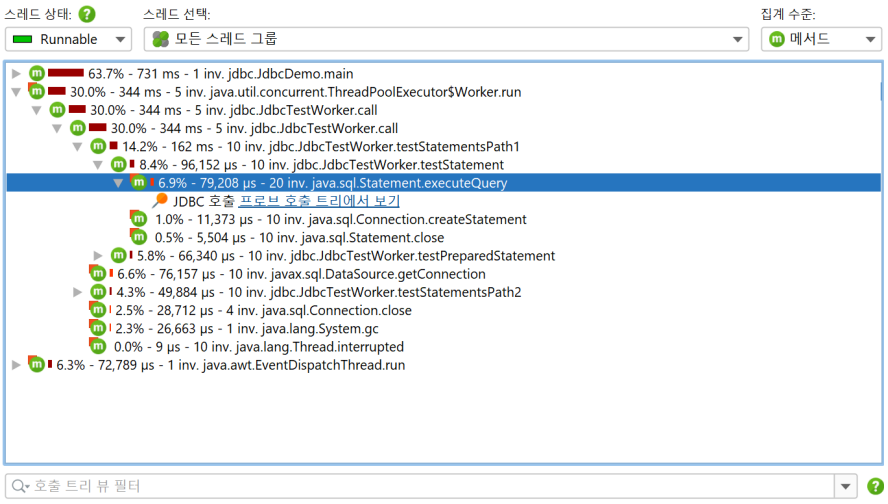
시적으로 지원하며, 가상 및 물리적 연결이 어떤 클래스인지 알고 있습니다. 알려지지 않은 풀이나 데이터베이스 드라이버의 경우, JProfiler는 물리적 연결을 가상 연결로 착각할 수 있습니다. 물리적 연결은 종종 장기간 지속되므로, "연결 누수" 뷰에 표시될 수 있습니다. 이 경우, 연결 객체의 클래스 이름은 이를 잘못된 긍정으로 식별하는 데 도움이 됩니다.

기본적으로 프로브 녹화를 시작할 때, 연결 누수 분석은 활성화되지 않습니다. 연결 누수 뷰에는 JDBC 프로브 설정의 연결 누수 분석을 위한 열린 가상 연결 기록 체크 박스에 해당하는 별도의 녹화 버튼이 있습니다. 이벤트 기록과 마찬가지로, 버튼의 상태는 지속되므로 한 번 분석을 시작하면 다음 프로브 녹화 세션에서도 자동으로 시작됩니다.

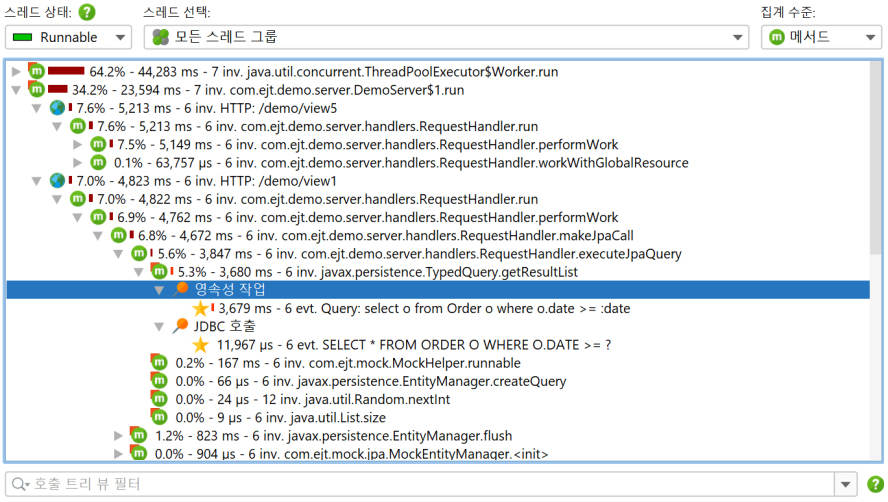


호출 트리의 페이로드 데이터

CPU 호출 트리를 볼 때, 프로브가 페이로드 데이터를 기록한 위치를 보는 것이 흥미롭습니다. 해당 데이터는 측정된 CPU 시간을 해석하는 데 도움이 될 수 있습니다. 이러한 이유로 많은 프로브가 CPU 호출 트리에 교차 링크를 추가합니다. 예를 들어, 클래스 로더 프로브는 클래스 로딩이 트리거된 위치를 보여줄 수 있습니다. 이는 호출 트리에서 보이지 않으며 예상치 못한 오버헤드를 추가할 수 있습니다. 호출 트리 뷰에서 불투명한 데이터베이스 호출은 단일 클릭으로 해당 프로브에서 추가 분석할 수 있습니다. 이는 호출 트리 분석에서도 작동하며, 분석이 프로브 호출 트리 뷰의 컨텍스트에서 자동으로 반복될 때 프로브 링크를 클릭하면 작동합니다.

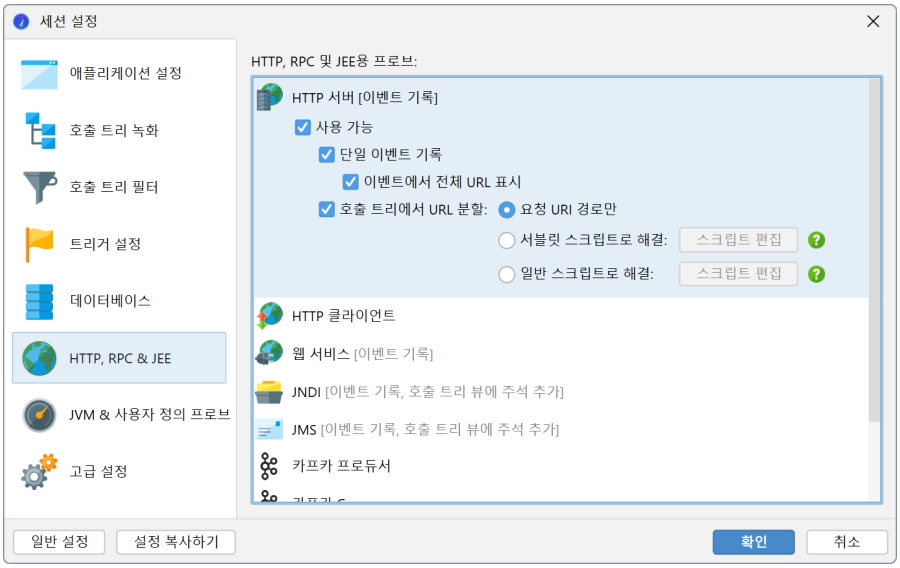


또 다른 가능성은 CPU 호출 트리에 직접 페이로드 정보를 인라인으로 표시하는 것입니다. 모든 관련 프로브는 해당 목적으로 구성에 호출 트리에 주석 추가 옵션을 가지고 있습니다. 이 경우, 프로브 호출 트리에 대한 링크는 사용할 수 없습니다. 각 프로브는 자체 페이로드 컨테이너 노드를 가지고 있습니다. 동일한 페이로드 이름을 가진 이벤트는 집계되며, 호출 횟수와 총 시간이 표시됩니다. 페이로드 이름은 호출 스택별로 통합되며, 가장 오래된 항목은 "[earlier calls]" 노드로 집계됩니다. 호출 스택당 기록된 최대 페이로드 이름 수는 프로파일링 설정에서 구성할 수 있습니다.

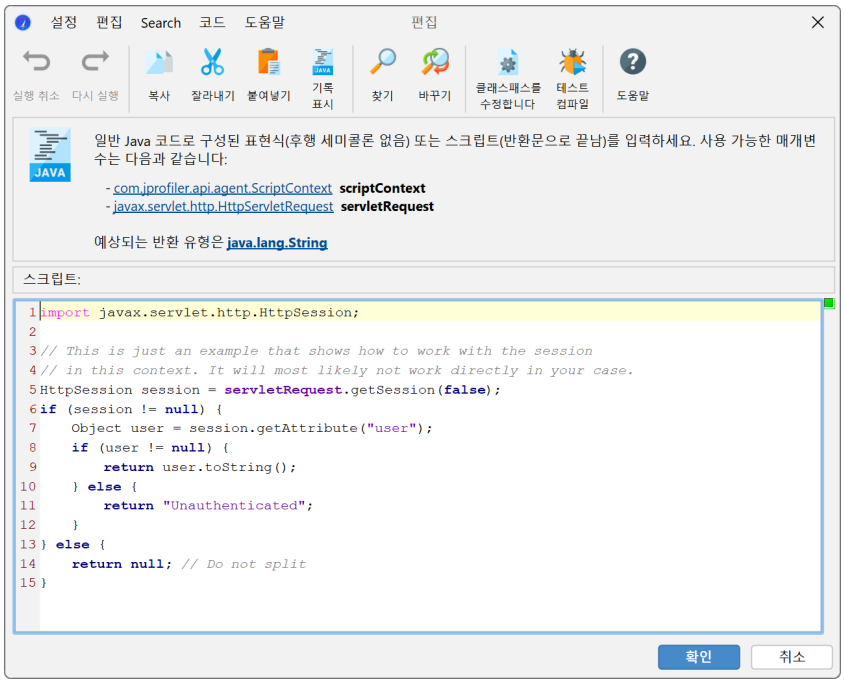


호출 트리 분할

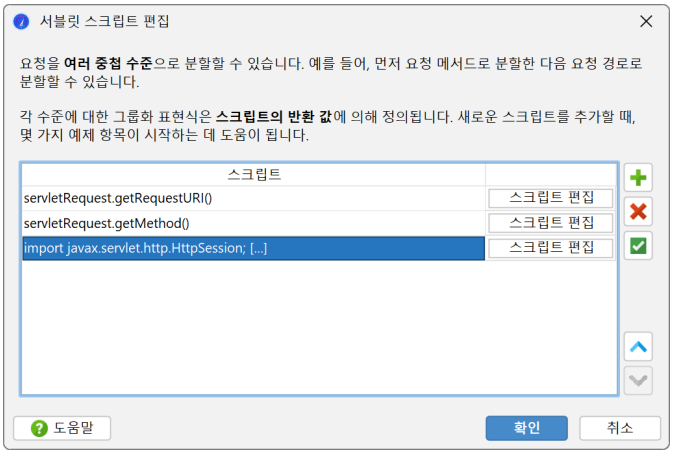
일부 프로브는 호출 트리에 페이로드 데이터를 주석으로 추가하기 위해 프로브 문자열을 사용하지 않습니다. 대신, 각 다른 프로브 문자열에 대해 호출 트리를 분할합니다. 이는 서버 유형 프로브에 특히 유용하며, 각 다른 유형의 수신 요청에 대해 호출 트리를 별도로 보고자 할 때 유용합니다. "HTTP 서버" 프로브는 URL을 가로채고 URL의 어떤 부분이 호출 트리를 분할하는 데 사용되어야 하는지에 대한 세밀한 제어를 제공합니다. 기본적으로 매개변수 없이 요청 URI 경로만 사용합니다.



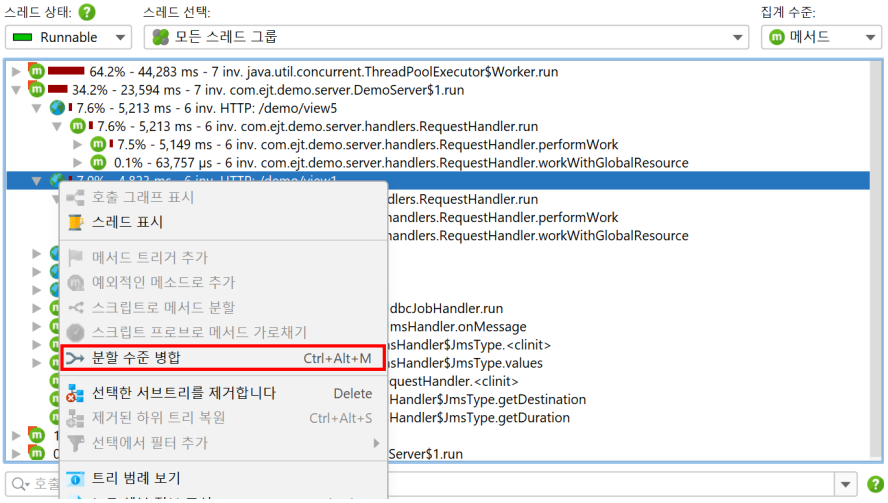
더 유연하게 하기 위해, 분할 문자열을 결정하는 스크립트를 정의할 수 있습니다. 스크립트에서는 현재 `javax.servlet.http.HttpServletRequest`를 매개변수로 받고 원하는 문자열을 반환합니다.



더 나아가, 단일 분할 수준에 국한되지 않고 여러 중첩된 분할을 정의할 수 있습니다. 예를 들어, 요청 URI 경로로 먼저 분할한 다음 HTTP 세션 객체에서 추출한 사용자 이름으로 분할할 수 있습니다. 또는 요청 메서드로 요청을 그룹화한 다음 요청 URI로 분할할 수 있습니다.



중첩된 분할을 사용하면 호출 트리의 각 수준에 대한 별도의 데이터를 볼 수 있습니다. 호출 트리를 볼 때, 특정 수준이 방해가 될 수 있으며 "HTTP 서버" 프로브 구성에서 이를 제거해야 할 필요성을 느낄 수 있습니다. 더 편리하게, 기록된 데이터의 손실 없이, 해당 분할 노드의 컨텍스트 메뉴를 사용하여 호출 트리에서 분할 수준을 일시적으로 병합하고 병합 해제할 수 있습니다.



호출 트리를 분할하면 상당한 메모리 오버헤드가 발생할 수 있으므로 주의해서 사용해야 합니다. 메모리 과부하를 방지하기 위해 JProfiler는 최대 분할 수를 제한합니다. 특정 분할 수준에 대한 분할 캡이 도달하면, 캡 카운터를 재설정하는 하이퍼링크가 있는 특별한 "[capped nodes]" 분할 노드가 추가됩니다. 기본 캡이 목적에 비해 너무 낮은 경우, 프로파일링 설정에서 이를 늘릴 수 있습니다.

가비지 컬렉터 분석

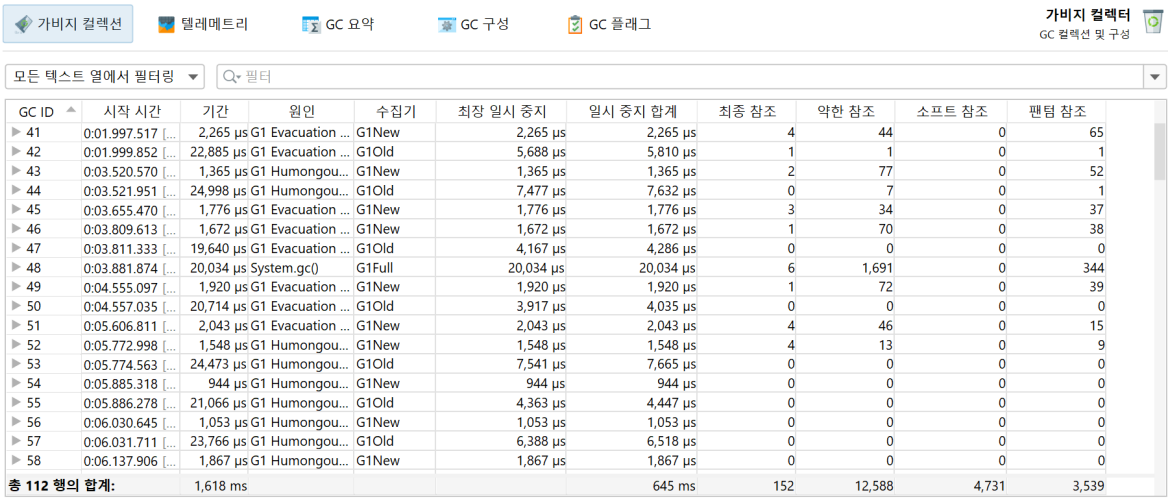
가비지 컬렉터(GC)의 런타임 특성을 이해하고 분석하는 것은 여러 가지 이유로 중요합니다. 첫째, GC 일시 중지는 애플리케이션의 응답성에 직접적인 영향을 줄 수 있습니다. 가비지 컬렉터가 어떻게 작동하는지 이해함으로써 이러한 일시 중지를 줄이기 위해 설정을 최적화할 수 있습니다. 일반적으로, 빈번하고 긴 GC 사이클은 힙이 너무 작거나 너무 많은 임시 객체가 생성되고 있음을 나타낼 수 있습니다.

가비지 컬렉터 프로브의 도움으로 이러한 문제를 해결하고 JVM 설정을 조정할 때 더 나은 결정을 내릴 수 있습니다. 예를 들어 적절한 가비지 컬렉터, 힙 크기 또는 기타 JVM 매개변수를 선택할 수 있습니다.

가비지 컬렉터 프로브는 다른 프로브와 다른 뷰를 가지고 있으며, 다른 데이터 소스를 사용합니다. JVM의 프로파일링 인터페이스에서 데이터를 얻지 않고 JFR 스트리밍을 사용하여 JDK 플라이트 레코더[(1)]의 GC 관련 이벤트를 분석합니다. JFR 이벤트 스트리밍에 의존하기 때문에, GC 프로브는 Java 17 이상에서 Hotspot JVM을 프로파일할 때만 사용할 수 있습니다. JFR 스냅샷을 열 때 [p. 209], 사용된 Java 버전에 관계없이 동일한 프로브를 사용할 수 있습니다.

가비지 컬렉션 뷰

가비지 컬렉터 프로브의 주요 뷰는 "가비지 컬렉션" 테이블입니다. 이 테이블은 모든 기록된 가비지 컬렉션을 행으로 표시하며, 가장 중요한 메트릭을 열로 표시합니다.



GC ID	시작 시간	기간	원인	수집기	최장 일시 중지	일시 중지 합계	최종 참조	약한 참조	소프트 참조	팬텀 참조
41	0:01.997.517 [...	2,265 µs	G1 Evacuation ...	G1New	2,265 µs	2,265 µs	4	44	0	65
42	0:01.999.852 [...	22,885 µs	G1 Evacuation ...	G1Old	5,688 µs	5,810 µs	1	1	0	1
43	0:03.520.570 [...	1,365 µs	G1 Humongou...	G1New	1,365 µs	1,365 µs	2	77	0	52
44	0:03.521.951 [...	24,998 µs	G1 Humongou...	G1Old	7,477 µs	7,632 µs	0	7	0	1
45	0:03.655.470 [...	1,776 µs	G1 Evacuation ...	G1New	1,776 µs	1,776 µs	3	34	0	37
46	0:03.809.613 [...	1,672 µs	G1 Evacuation ...	G1New	1,672 µs	1,672 µs	1	70	0	38
47	0:03.811.333 [...	19,640 µs	G1 Evacuation ...	G1Old	4,167 µs	4,286 µs	0	0	0	0
48	0:03.881.874 [...	20,034 µs	System.gc()	G1Full	20,034 µs	20,034 µs	6	1,691	0	344
49	0:04.555.097 [...	1,920 µs	G1 Evacuation ...	G1New	1,920 µs	1,920 µs	1	72	0	39
50	0:04.557.035 [...	20,714 µs	G1 Evacuation ...	G1Old	3,917 µs	4,035 µs	0	0	0	0
51	0:05.606.811 [...	2,043 µs	G1 Evacuation ...	G1New	2,043 µs	2,043 µs	4	46	0	15
52	0:05.772.998 [...	1,548 µs	G1 Humongou...	G1New	1,548 µs	1,548 µs	4	13	0	9
53	0:05.774.563 [...	24,473 µs	G1 Humongou...	G1Old	7,541 µs	7,665 µs	0	0	0	0
54	0:05.885.318 [...	944 µs	G1 Humongou...	G1New	944 µs	944 µs	0	0	0	0
55	0:05.886.278 [...	21,066 µs	G1 Humongou...	G1Old	4,363 µs	4,447 µs	0	0	0	0
56	0:06.030.645 [...	1,053 µs	G1 Humongou...	G1New	1,053 µs	1,053 µs	0	0	0	0
57	0:06.031.711 [...	23,766 µs	G1 Humongou...	G1Old	6,388 µs	6,518 µs	0	0	0	0
58	0:06.137.906 [...	1,867 µs	G1 Humongou...	G1New	1,867 µs	1,867 µs	0	0	0	0
총 112 행의 합계:		1,618 ms				645 ms	152	12,588	4,731	3,539

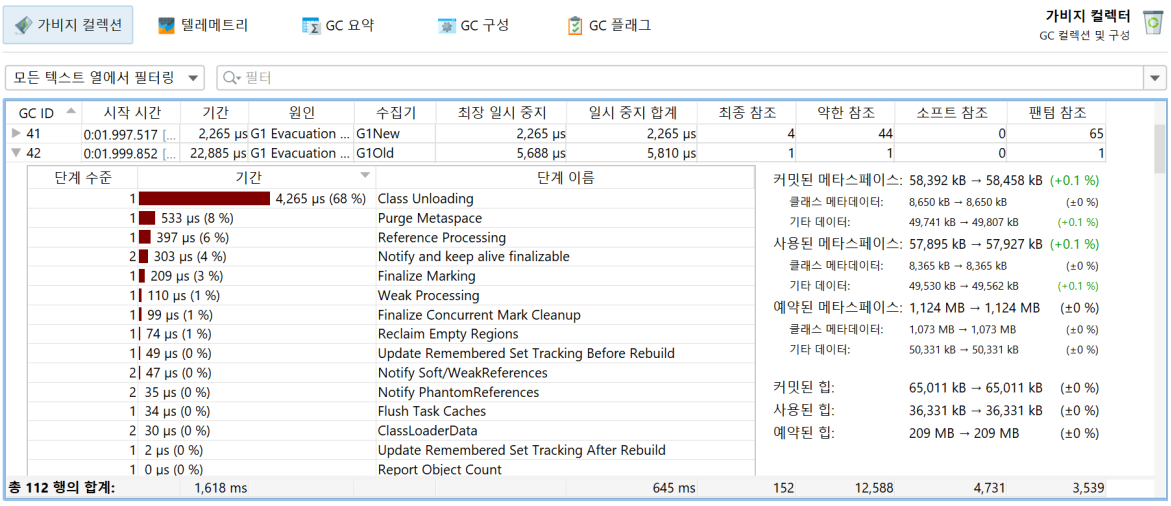
"원인" 열은 가비지 컬렉션이 트리거된 이유를 보여줍니다. 예를 들어, `System.gc()` 호출은 전체 가비지 컬렉션을 트리거했습니다. 이는 "수집기" 열의 "G1Full" 값에서 확인할 수 있습니다. 또한 20ms의 상당한 일시 중지를 유발했기 때문에 일반적으로 `System.gc()`를 호출하는 것은 좋은 생각이 아닙니다. 다른 원인은 젊은 세대 공간("G1New")의 수집을 트리거하거나, 오래된 세대의 참조되지 않은 객체를 정리하는 G1 수집기의 오래된 GC 수집("G1Old")을 트리거합니다. 오래된 GC 수집은 젊은 세대 수집보다 시간이 더 오래 걸리지만, 젊은 세대 수집은 더 많은 객체를 수집합니다.

특별한 GC 처리가 있는 수집된 참조는 "final", "weak", "soft" 및 "phantom" 참조로 별도의 열에 표시됩니다.

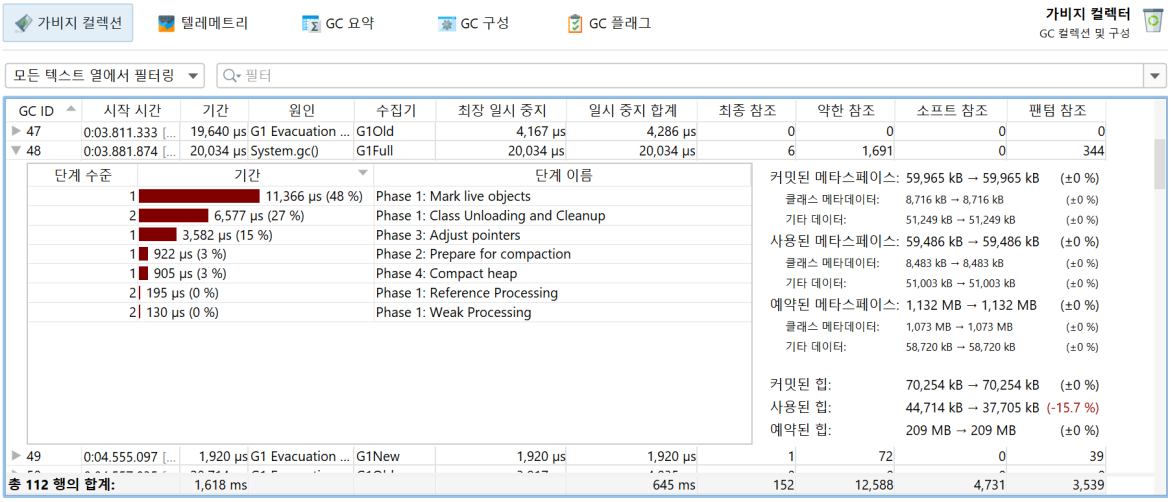
가장 긴 일시 중지와 일시 중지 합계에 대해 별도의 열이 있는 이유는 각 가비지 컬렉션이 여러 단계로 구성되어 별도의 일시 중지를 생성하기 때문입니다. 또한, 가비지 컬렉션의 "지속 시간"은 일시 중지 합계와 같지 않습니다. 가비지 컬렉션은 실행 중에 JVM을 부분적으로만 일시 중지하기 때문입니다. 스크린샷에서 "G1Old" 컬렉션은 지속 시간의 약 5분의 1만큼만 일시 중지됩니다.

가비지 컬렉션의 다양한 단계를 검사하려면 "GC ID" 열의 트리 아이콘을 전환할 수 있습니다.

(1) https://en.wikipedia.org/wiki/JDK_Flight_Recorder

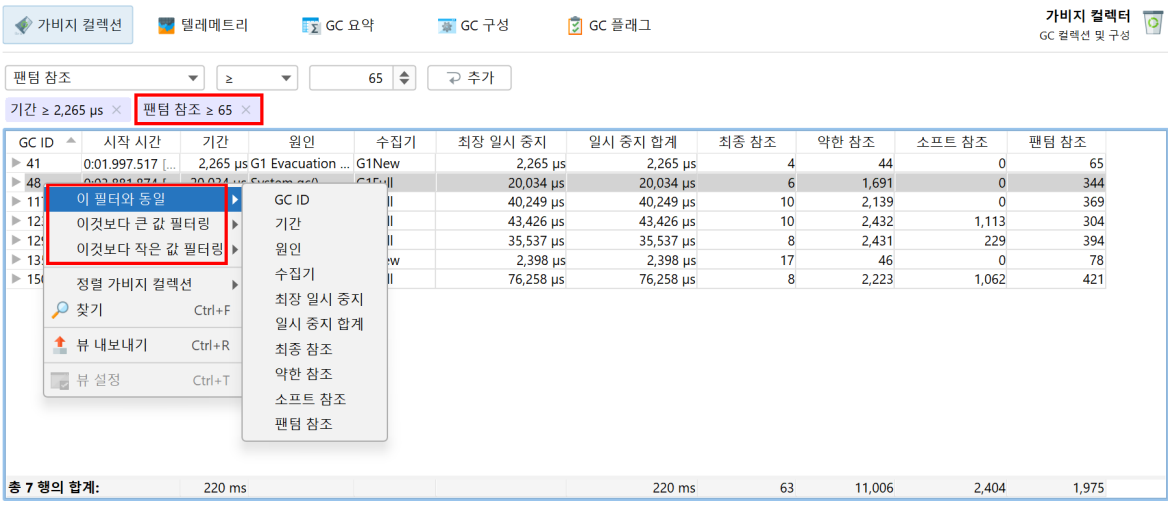


위의 스크린샷에서는 G1 수집기의 혼합 GC 수집("G1Old")이 확장되었습니다. 대부분의 시간이 "클래스 언로딩"에 소비되며, 이는 JVM을 일시 중지하지 않습니다. 오른쪽에서는 가비지 컬렉션에 대한 추가 통계를 볼 수 있습니다. 여기서 사용된 힙은 동일하게 유지되었지만 사용된 메타스페이스는 0.1% 증가했습니다.

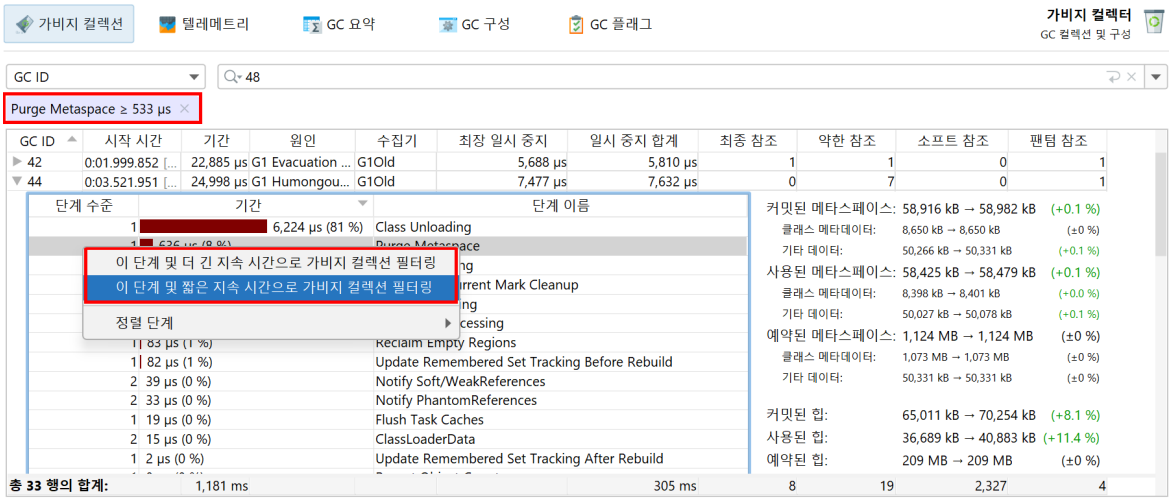


각 수집기의 단계는 다릅니다. 위의 스크린샷에서는 전체 수집이 표시됩니다. 전체 힙에서 살아있는 객체를 마킹하는 데 많은 시간이 소요됩니다. 수집이 끝날 때, 사용된 힙은 15.7% 감소했으며, 메타스페이스는 동일하게 유지되었습니다.

가비지 컬렉션을 분석할 때, **필터링**은 다양한 가비지 컬렉션의 하위 집합을 비교하는 중요한 도구입니다. 테이블 상단에는 필터 선택기가 있어 원하는 열을 선택하고 해당 필터를 구성할 수 있습니다. 유사한 가비지 컬렉션을 쉽게 확인하려면 테이블의 컨텍스트 메뉴를 사용하여 선택한 행의 열 값에 기반한 필터 조건을 선택할 수 있습니다.

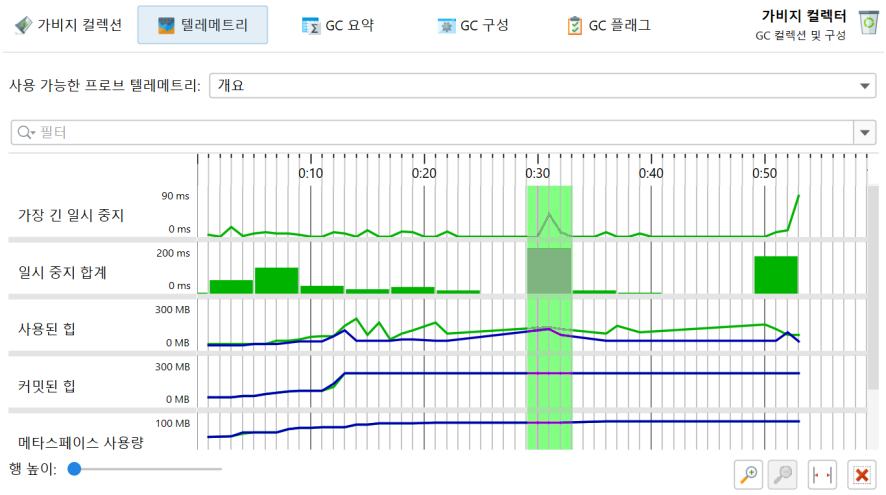


관심 있는 가비지 컬렉션을 좁히기 위해 여러 필터를 추가할 수 있습니다. 활성 필터는 테이블 상단에 레이블로 표시됩니다. 중첩된 GC 단계 테이블에서도 필터를 추가할 수 있습니다.

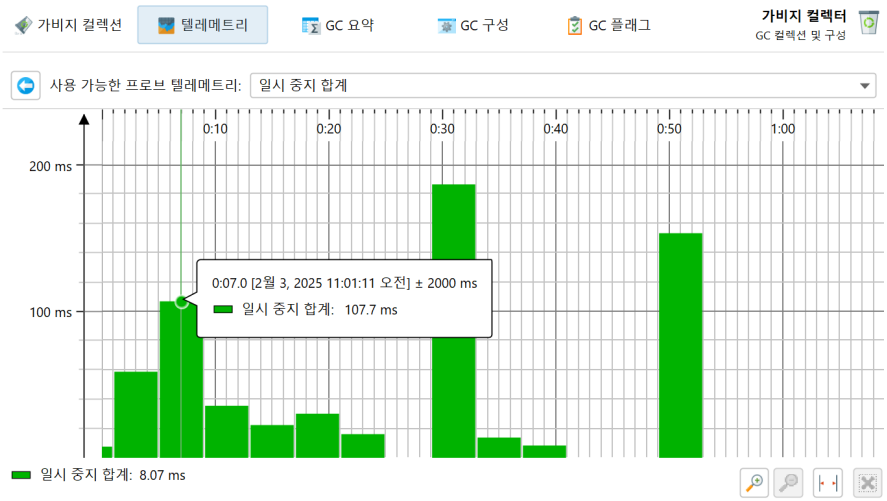


텔레메트리

GC 프로브는 "텔레메트리" 프로브 뷰에서 사용할 수 있는 여러 텔레메트리를 생성합니다.



GC 일시 중지를 최소화하는 데 관심이 있다면, 상단의 "가장 긴 일시 중지" 텔레메트리가 가장 흥미로울 것입니다. 텔레메트리의 시간 축을 따라 드래그하여 "가비지 컬렉션" 뷰에서 해당 가비지 컬렉션을 선택할 수 있습니다. 더 나은 수직 해상도를 위해 상단의 드롭다운에서 단일 텔레메트리를 선택하거나 텔레메트리 이름을 클릭하여 선택할 수 있습니다.



위의 스크린샷에서는 시간에 따른 일시 중지 합계를 볼 수 있습니다. JProfiler는 기록된 데이터의 **히스토그램을 구축**하여 합산 가능한 측정을 제공합니다. 빈 너비는 사용 가능한 수평 공간에 따라 달라지므로, 히스토그램 빈은 확대/축소 수준에 따라, 그리고 "맞춤 비율"이 활성화된 경우 창의 너비에 따라 변경됩니다. 동일하게 유지되는 것은 모든 히스토그램 빈 아래의 총 면적입니다.

힙 및 메타스페이스 텔레메트리는 가비지 컬렉션을 확장할 때 볼 수 있는 통계에 기반합니다. 이는 전체 프로파일링 세션의 메모리 텔레메트리와 같이 정기적으로 샘플링되지 않습니다. 특정 기간 동안 가비지 컬렉션이 발생하지 않으면 데이터가 없습니다. 할당 활동이 적은 JVM의 경우, 그래프가 두 가비지 컬렉션 사이에서 단순히 보간되는 시간 축을 따라 긴 구간이 있을 수 있습니다.

이러한 각 텔레메트리는 "GC 전" 및 "GC 후"의 두 데이터 라인을 가지고 있습니다. 차이는 일반적으로 "사용된 힙" 텔레메트리에서 큽니다. 각 시간에 가비지 컬렉션이 수행한 작업량을 두 데이터 라인의 값을 비교하여 확인할 수 있습니다. 툴팁을 보면 정확한 값을 확인할 수 있습니다. "커밋된 힙" 텔레메트리와 메타스페이스 텔레메트리의 경우, 두 라인 간의 차이는 종종 작습니다.

JFR 스냅샷 [p. 209]을 분석하는 경우, 동일한 데이터가 "메모리" 텔레메트리에서 `jdk.GCHeapSummary` JFR 이벤트 유형으로 사용됩니다. 그러나 이 경우, "GC 전" 및 "GC 후" 값이 동일한 데이터 라인에 표시되며,

GC 프로브 텔레메트리와 같이 초당 한 번의 집계로 데이터가 집계되지 않으므로 그래프가 다르게 보일 것입니다.

GC 요약

GC 요약은 전체 녹화 기간 동안 집계된 측정을 보여줍니다. 각 측정은 가비지 컬렉션의 수와 평균, 최대 및 총 값을 제공합니다. 상단의 가장 중요한 데이터는 애플리케이션의 활성도에 직접 영향을 미치는 "일시 중지 시간"입니다.

▼ 일시 중지 시간	
일시 중지 횟수	143
평균 일시 중지	4,512 µs
최대 일시 중지	76,258 µs
일시 중지 합계	645 ms
▼ 모든 컬렉션 총 시간	
평균 GC 시간	14,451 µs
최대 GC 시간	76,258 µs
총 GC 시간	1,618 ms
▼ 영 콜렉션 총 시간	
GC 횟수	70
평균 GC 시간	1,712 µs
최대 GC 시간	3,460 µs
총 GC 시간	119 ms
▼ 컬렉션 총 시간 (구버전)	
GC 횟수	42
평균 GC 시간	35,682 µs
최대 GC 시간	76,258 µs
총 GC 시간	1,498 ms

다른 최상위 카테고리는 모든 컬렉션의 총 시간을 보여주며, 이는 젊은 컬렉션과 오래된 컬렉션의 두 하위 카테고리로 나뉩니다.

GC 구성

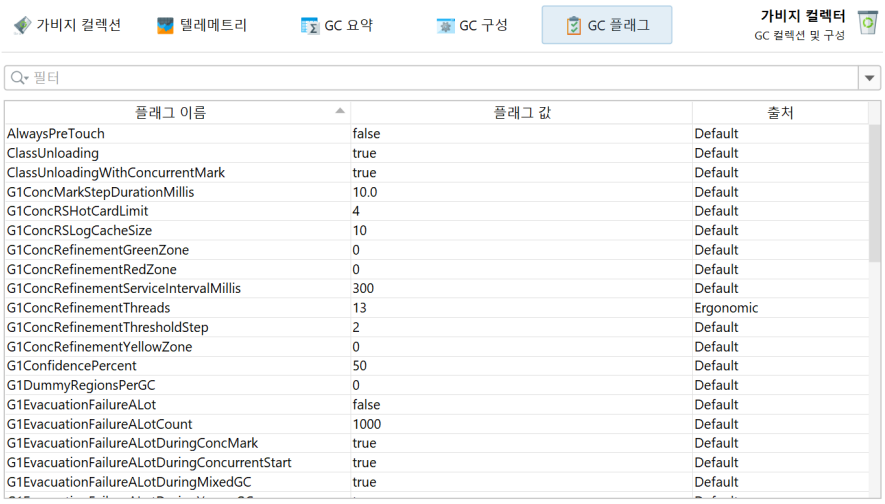
가비지 컬렉터를 조정할 때, 명시적으로 설정할 수 있거나 가비지 컬렉터 자체에 의해 암시적으로 설정된 일반 속성을 검사하고 싶을 수 있습니다.

▼ GC 구성	
Young Garbage Collector	G1New
Old Garbage Collector	G1Old
Concurrent GC Threads	3
Parallel GC Threads	13
Concurrent Explicit GC	false
Disabled Explicit GC	false
Uses Dynamic GC Threads	true
GC Time Ratio	12
▼ GC 힙 구성	
Initial Size	209 MB
Minimum Heap Size	8,388 kB
Maximum Heap Size	209 MB
If Compressed Oops Are Used	true
Compressed Oops Mode	32-bit
Heap Address Size	32
Object Alignment	8 바이트
▼ 영 제너레이션 구성	
Minimum Young Generation Size	1,363 kB
Maximum Young Generation Size	125 MB

이러한 속성은 모든 가비지 컬렉터에 공통적이며, 가비지 컬렉터 간의 차이를 이해하는 데 도움이 됩니다.

GC 플래그

마지막으로, GC 전용 플래그는 가비지 컬렉터의 어떤 속성을 조정할 수 있는지에 대한 아이디어를 제공하며, 실제 값을 확인할 수 있습니다.



플래그 이름	플래그 값	출처
AlwaysPreTouch	false	Default
ClassUnloading	true	Default
ClassUnloadingWithConcurrentMark	true	Default
G1ConcMarkStepDurationMillis	10.0	Default
G1ConcRSHotCardLimit	4	Default
G1ConcRSLogCacheSize	10	Default
G1ConcRefinementGreenZone	0	Default
G1ConcRefinementRedZone	0	Default
G1ConcRefinementServiceIntervalMillis	300	Default
G1ConcRefinementThreads	13	Ergonomic
G1ConcRefinementThresholdStep	2	Default
G1ConcRefinementYellowZone	0	Default
G1ConfidencePercent	50	Default
G1DummyRegionsPerGC	0	Default
G1EvacuationFailureALot	false	Default
G1EvacuationFailureALotCount	1000	Default
G1EvacuationFailureALotDuringConcMark	true	Default
G1EvacuationFailureALotDuringConcurrentStart	true	Default
G1EvacuationFailureALotDuringMixedGC	true	Default

"원본" 열은 플래그가 어떻게 설정되었는지를 보여줍니다. "기본값"은 표준 설정에서 수정되지 않은 값을 나타내며, "인체공학적" 플래그는 가비지 컬렉터에 의해 자동으로 조정된 값을 나타냅니다. 명령줄에서 특정 GC 플래그를 설정한 경우, 원본에서 "명령줄"로 보고됩니다.

MBean 브라우저

Apache Camel [1]과 같은 많은 애플리케이션 서버 및 프레임워크는 JMX를 사용하여 구성 및 모니터링 목적으로 다수의 MBean을 노출합니다. JVM 자체도 JVM의 저수준 작업에 대한 흥미로운 정보를 제공하는 플랫폼 MXBean [2]을 다수 게시합니다.

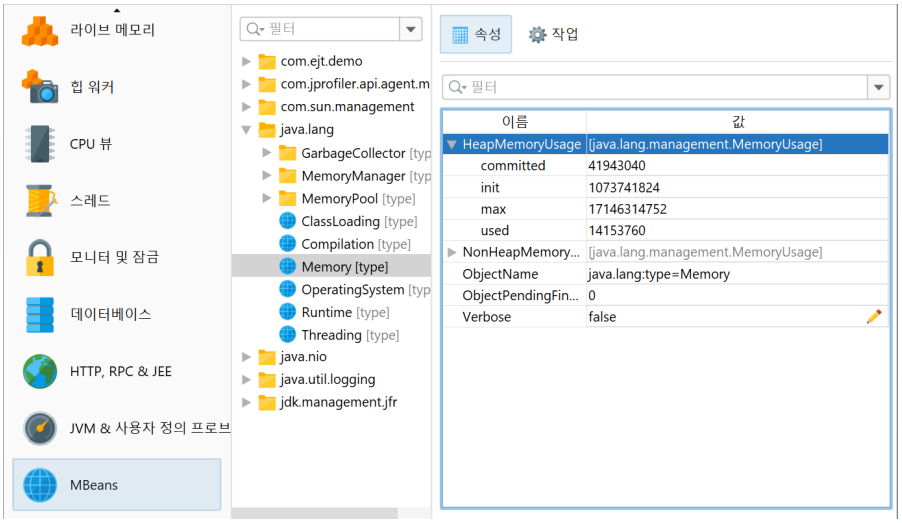
JProfiler는 프로파일된 VM에 등록된 모든 MBean을 보여주는 MBean 브라우저를 포함합니다. MBean 서버에 접근하기 위한 JMX의 원격 관리 수준은 필요하지 않습니다. 왜냐하면 JProfiler 에이전트가 이미 프로세스 내에서 실행 중이며 모든 등록된 MBean 서버에 접근할 수 있기 때문입니다.

JProfiler는 **Open MBean**의 타입 시스템을 지원합니다. 간단한 타입을 정의하는 것 외에도, Open MBean은 사용자 정의 클래스가 포함되지 않은 복잡한 데이터 타입을 정의할 수 있습니다. 또한, 배열과 테이블이 데이터 구조로 사용 가능합니다. **MXBean**을 사용하면 JMX는 Java 클래스에서 Open MBean을 자동으로 생성하는 쉬운 방법을 제공합니다. 예를 들어, JVM이 제공하는 MBean은 MXBean입니다.

MBean은 계층 구조가 없지만, JProfiler는 객체 도메인 이름을 첫 번째 콜론까지 첫 번째 트리 수준으로 사용하고 모든 속성을 재귀적으로 중첩된 수준으로 사용하여 트리로 조직합니다. 속성 값은 속성 키를 괄호 안에 두고 끝에 표시됩니다. `type` 속성은 최상위 노드 바로 아래에 나타나도록 우선 순위가 부여됩니다.

속성

MBean 내용을 보여주는 트리 테이블의 최상위 수준에서 MBean 속성을 볼 수 있습니다.



다음 데이터 구조는 중첩된 행으로 표시됩니다:

- **Arrays**
  기본 배열 및 객체 배열의 요소는 키 이름으로 인덱스를 사용하여 중첩된 행으로 표시됩니다.
- **Composite data**
  복합 데이터 타입의 모든 항목은 중첩된 행으로 표시됩니다. 각 항목은 임의의 타입일 수 있으므로 중첩은 임의의 깊이까지 계속될 수 있습니다.
- **Tabular data**
  대부분의 경우 MXBean에서 `java.util.Map` 인스턴스가 하나의 키 열과 하나의 값 열을 가진 표형 데이터 타입으로 매핑됩니다. 키의 타입이 간단한 타입인 경우, 맵은 "인라인"으로 표시되며 각 키-값 쌍은 중첩

[1] https://camel.apache.org/camel-jmx.html
[2] https://docs.oracle.com/javase/7/docs/technotes/guides/management/mxbeans.html

된 행으로 표시됩니다. 키가 복잡한 타입인 경우, 중첩된 키 및 값 항목이 있는 "맵 항목" 요소 수준이 삽입됩니다. 이는 복합 키와 다중 값을 가진 일반적인 표형 타입의 경우에도 해당됩니다.

선택적으로, MBean 속성은 편집 가능할 수 있으며 이 경우 편집 아이콘이 값 옆에 표시되고 값 편집 작업이 활성화됩니다. 복합 및 표형 타입은 MBean 브라우저에서 편집할 수 없지만 배열이나 간단한 타입은 편집 가능합니다.

값이 널일 수 있는 경우, 예를 들어 배열과 같은 경우, 편집기에는 널 상태를 선택할 수 있는 체크박스가 있습니다.

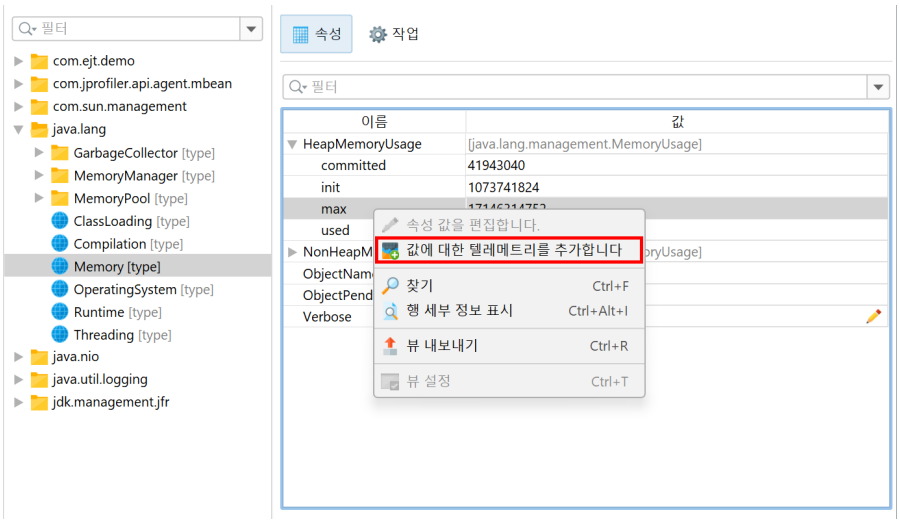
배열 요소는 세미콜론으로 구분됩니다. 하나의 후행 세미콜론은 무시될 수 있으므로 1과 1;은 동일합니다. 세미콜론 앞에 값이 없으면 객체 배열의 경우 널 값으로 처리됩니다. 문자열 배열의 경우, 이중 따옴표("")로 빈 요소를 생성할 수 있으며, 세미콜론을 포함하는 요소는 전체 요소를 인용하여 생성할 수 있습니다. 문자열 요소의 이중 따옴표는 두 번 사용해야 합니다. 예를 들어, 다음과 같은 문자열 배열 값을 입력하면

```
"Test";"";;"embedded "" quote";"A;B";;
```

다음과 같은 문자열 배열이 생성됩니다

```
new String[] {"Test", "", null, "embedded \" quote", "A;B", null}
```

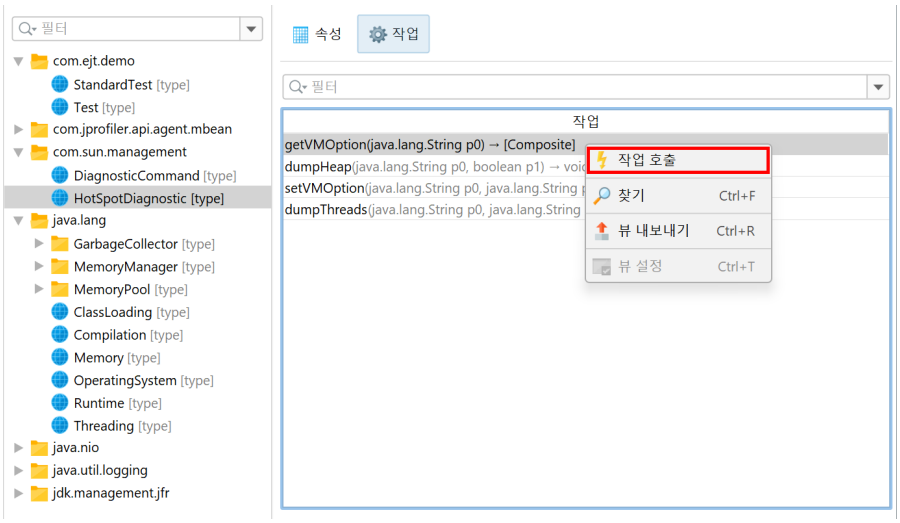
JProfiler는 숫자 MBean 속성 값에서 사용자 정의 텔레메트리를 생성할 수 있습니다. 사용자 정의 텔레메트리에 대해 MBean 텔레메트리 라인 [p. 44]을 정의할 때, 텔레메트리 데이터를 제공하는 속성을 선택할 수 있는 MBean 속성 브라우저가 표시됩니다. 이미 MBean 브라우저에서 작업 중인 경우, 컨텍스트 메뉴의 값에 대한 텔레메트리 추가 작업은 새로운 사용자 정의 텔레메트리를 생성하는 편리한 방법을 제공합니다.



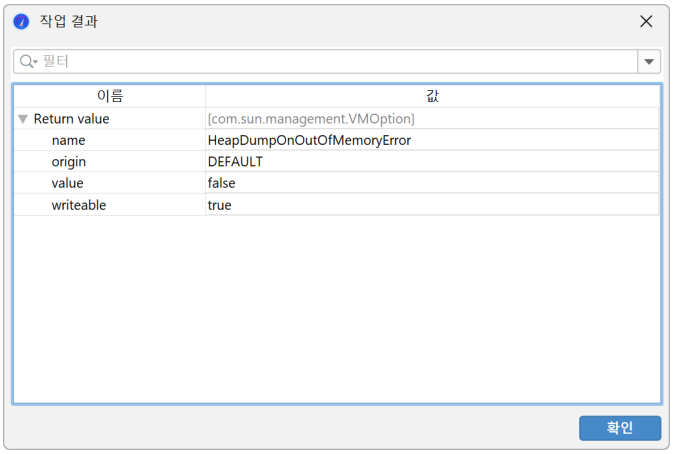
텔레메트리는 복합 데이터 또는 간단한 키와 단일 값을 가진 표형 데이터의 중첩된 값을 추적할 수도 있습니다. 중첩된 행을 선택하면 경로 구성 요소가 슬래시로 구분된 값 경로가 생성됩니다.

작업

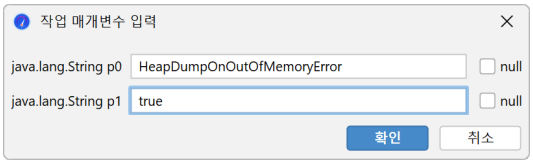
MBean 속성을 검사하고 수정하는 것 외에도 MBean 작업을 호출하고 그 반환 값을 확인할 수 있습니다. MBean 작업은 MBean 인터페이스의 메서드로, 설정자나 가져오기 메서드가 아닙니다.



작업의 반환 값은 복합, 표형 또는 배열 타입일 수 있으므로 MBean 속성 트리 테이블과 유사한 내용의 새 창이 표시됩니다. 간단한 반환 타입의 경우, "반환 값"이라는 이름의 행이 하나만 있습니다. 다른 타입의 경우, "반환 값"은 결과가 추가되는 루트 요소입니다.



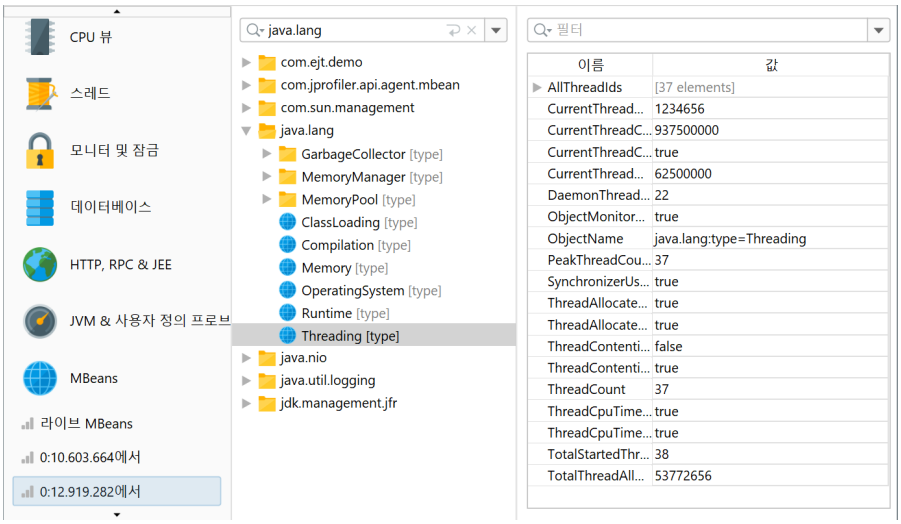
MBean 작업은 하나 이상의 인수를 가질 수 있습니다. 입력할 때, MBean 속성을 편집할 때와 동일한 규칙과 제한이 적용됩니다.



MBean 스냅샷

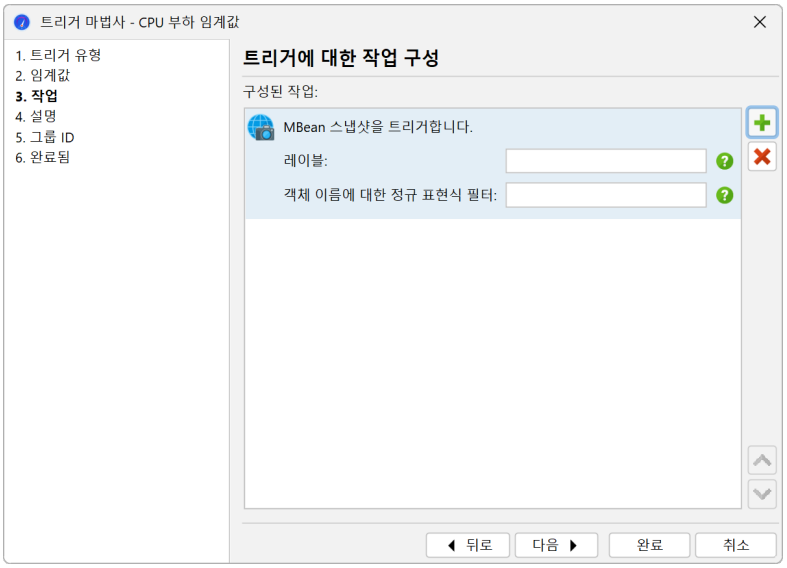
MBean의 실시간 값을 보는 것 외에도 현재 상태의 스냅샷을 찍을 수 있습니다. 각 새로운 스냅샷은 MBean 뷰 섹션의 별도 뷰로 추가되며 사용자 정의 레이블을 지정할 수 있습니다. 스냅샷이 찍힐 때, 현재 필터에 따라 표

시되는 MBean만 포함됩니다. 이렇게 하면 특정 MBean에 집중하고 관련 없는 MBean을 쿼리하는 오버헤드를 줄일 수 있습니다.



JProfiler UI에서 스냅샷을 저장할 때, 모든 MBean 스냅샷도 저장되지만 실시간 MBean 뷰는 저장되지 않습니다. 오프라인 프로파일링 [p. 122]의 경우, Controller API 또는 "MBean 스냅샷 저장" 트리거 작업을 사용하여 MBean 스냅샷을 프로그래밍 방식으로 찍을 수 있습니다.

컨트롤러 API와 트리거 작업 모두 뷰 선택기에 표시되는 선택적 레이블과 포함된 MBean을 필터링하기 위한 선택적 정규 표현식을 지원합니다.



오프라인 프로파일링

JProfiler로 애플리케이션을 프로파일링하는 두 가지 근본적으로 다른 방법이 있습니다: 기본적으로 JProfiler GUI에 attach된 상태로 프로파일링합니다. JProfiler GUI는 녹화를 시작하고 중지할 수 있는 버튼을 제공하며, 모든 녹화된 프로파일링 데이터를 보여줍니다.

JProfiler GUI 없이 프로파일링하고 나중에 결과를 분석하고 싶은 상황이 있을 수 있습니다. 이러한 시나리오를 위해 JProfiler는 오프라인 프로파일링을 제공합니다. 오프라인 프로파일링을 통해 프로파일링 에이전트와 함께 프로파일된 애플리케이션을 시작할 수 있지만 JProfiler GUI와 연결할 필요는 없습니다.

그러나 오프라인 프로파일링은 여전히 몇 가지 작업을 수행해야 합니다. 최소한 하나의 스냅샷을 저장해야 하며, 그렇지 않으면 나중에 분석할 프로파일링 데이터가 제공되지 않습니다. 또한, CPU 또는 할당 데이터를 보려면 어느 시점에서 녹화를 시작해야 합니다. 마찬가지로 저장된 스냅샷에서 힙 워커를 사용하려면 힙 덤프를 트리거해야 합니다.

프로파일링 API

이 문제에 대한 첫 번째 해결책은 컨트롤러 API입니다. API를 사용하면 코드에서 모든 프로파일링 작업을 프로그래밍 방식으로 호출할 수 있습니다. `api/samples/offline` 디렉토리에는 컨트롤러 API를 실제로 사용하는 방법을 보여주는 실행 가능한 예제가 있습니다. 해당 디렉토리에서 `../gradlew`를 실행하여 컴파일하고 실행하며, 테스트 프로그램이 호출되는 방법을 이해하기 위해 Gradle 빌드 파일 `build.gradle`을 공부하십시오.

컨트롤러 API는 런타임에 프로파일링 작업을 관리하는 주요 인터페이스입니다. 이는 JProfiler 설치의 `bin/agent.jar`에 포함되어 있거나 다음 좌표로 Maven 종속성으로 사용할 수 있습니다.

```
group: com.jprofiler
artifact: jprofiler-probe-injected
version: <JProfiler version>
```

그리고 저장소

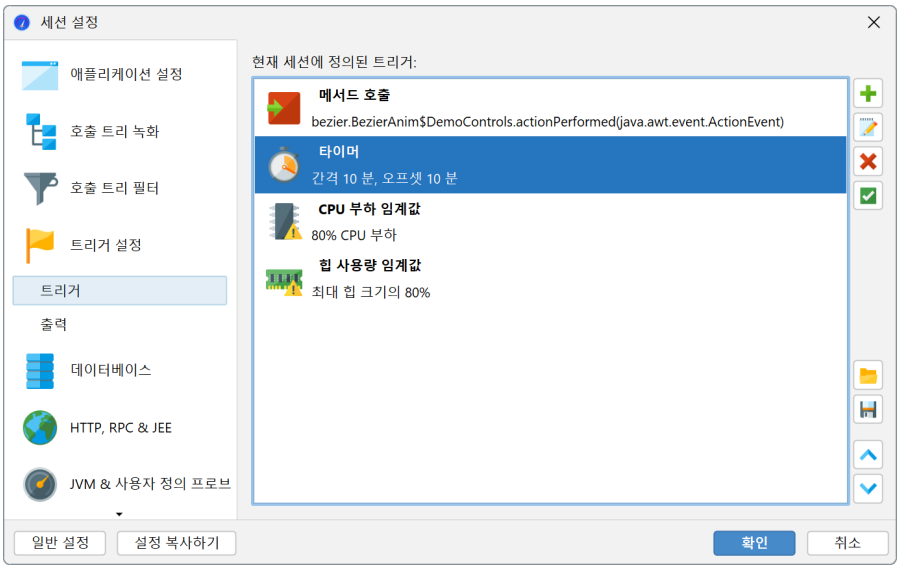
```
https://maven.ej-technologies.com/repository
```

프로파일링 API가 애플리케이션의 일반 실행 중에 사용되는 경우, API 호출은 조용히 아무 작업도 수행하지 않습니다.

이 접근 방식의 단점은 개발 중에 JProfiler 에이전트 라이브러리를 애플리케이션의 클래스패스에 추가하고, 소스 코드에 프로파일링 지침을 추가하고, 프로그래밍 방식의 프로파일링 작업을 변경할 때마다 코드를 다시 컴파일해야 한다는 것입니다.

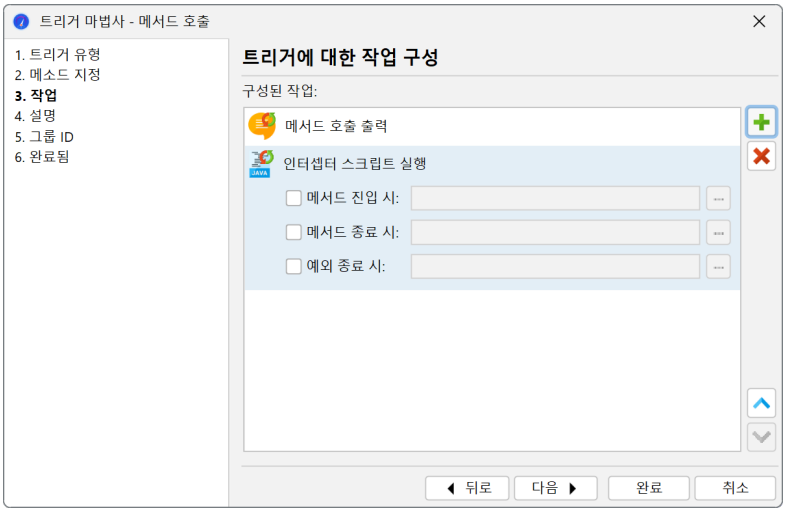
트리거

트리거 [p. 26]를 사용하면 소스 코드를 수정하지 않고 JProfiler GUI에서 모든 프로파일링 작업을 지정할 수 있습니다. 트리거는 JProfiler 구성 파일에 저장됩니다. 구성 파일과 세션 ID는 오프라인 프로파일링이 활성화된 상태로 시작할 때 명령줄에서 프로파일링 에이전트에 전달되어, 프로파일링 에이전트가 해당 트리거 정의를 읽을 수 있습니다.



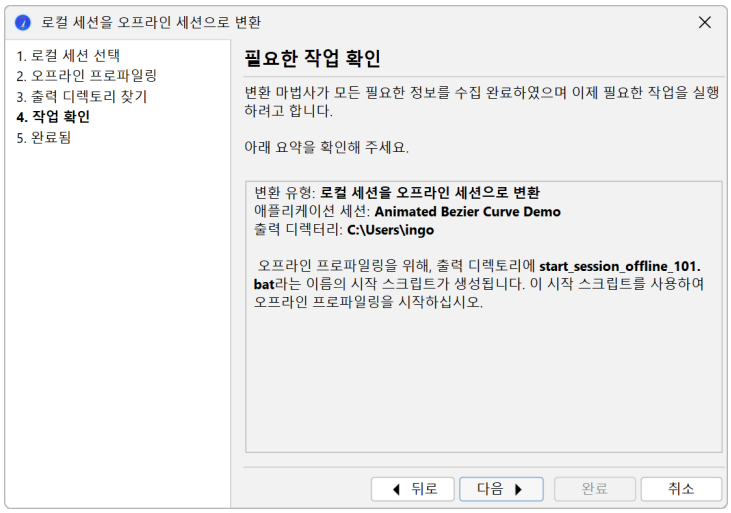
소스 코드에 API 호출을 추가하는 프로파일링 API와 달리, 트리거는 JVM에서 특정 이벤트가 발생할 때 활성화됩니다. 예를 들어, 메소드의 시작 또는 끝에서 특정 프로파일링 작업에 대한 API 호출을 추가하는 대신, 메소드 호출 트리거를 사용할 수 있습니다. 또 다른 사용 사례로는, 주기적으로 스냅샷을 저장하기 위해 자체 타이머 스레드를 생성하는 대신 타이머 트리거를 사용할 수 있습니다.

각 트리거에는 관련 이벤트가 발생할 때 수행되는 작업 목록이 있습니다. 이러한 작업 중 일부는 컨트롤러 API의 프로파일링 작업에 해당합니다. 또한, 메소드 호출을 매개변수 및 반환 값과 함께 출력하는 작업이나 메소드에 대한 인터셉터 스크립트를 호출하는 작업과 같은 컨트롤러 기능을 초과하는 다른 작업도 있습니다.

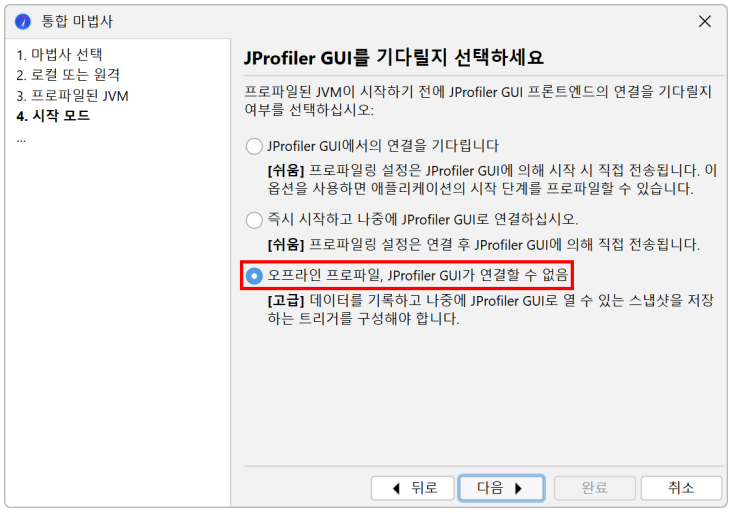


오프라인 프로파일링 구성

JProfiler에서 시작된 세션을 구성한 경우, 메인 메뉴에서 세션->변환 마법사->애플리케이션 세션을 오프라인으로 변환을 호출하여 오프라인 세션으로 변환할 수 있습니다. 이렇게 하면 적절한 VM 매개변수를 사용하여 시작 스크립트를 생성하고, JProfiler UI에서 사용하는 동일한 세션의 프로파일링 설정을 가져옵니다. 호출을 다른 컴퓨터로 이동하려면 세션->세션 설정 내보내기를 사용하여 세션을 구성 파일로 내보내고, 시작 스크립트의 VM 매개변수가 해당 파일을 참조하도록 해야 합니다.



통합 마법사를 사용하여 애플리케이션 서버를 프로파일링할 때, 프로파일링을 위한 VM 매개변수가 Java 호출에 삽입되도록 시작 스크립트나 구성 파일이 항상 수정됩니다. 모든 통합 마법사에는 "시작" 단계에서 오프라인 프로파일링을 구성하기 위한 "오프라인 프로파일링" 옵션이 있습니다.



통합 마법사에서 처리되지 않는 시작 스크립트가 있는 경우, 예를 들어 Java 호출에 VM 매개변수를 직접 전달하고 싶을 수 있습니다. 해당 VM 매개변수의 형식은 다음과 같습니다.

```
-agentpath:<path to jprofilerti library>=offline,id=<ID>[,config=<path>]
```

이는 `[Generic application]` 마법사에서 사용할 수 있습니다.

라이브러리 매개변수로 `offline`을 전달하면 오프라인 프로파일링이 활성화됩니다. 이 경우, JProfiler GUI와의 연결은 불가능합니다. `session` 매개변수는 프로파일링 설정에 사용할 구성 파일의 세션을 결정합니다. 세션의 ID는 세션 설정 대화 상자의 애플리케이션 설정 탭의 오른쪽 상단 모서리에서 볼 수 있습니다. 선택적 `config` 매개변수는 구성 파일을 가리킵니다. 이는 세션->세션 설정 내보내기를 호출하여 내보낼 수 있는 파일입니다. 매개변수를 생략하면 표준 구성 파일이 사용됩니다. 해당 파일은 사용자 홈 디렉토리의 `.jprofiler15` 디렉토리에 위치합니다.

Gradle 및 Ant를 사용한 오프라인 프로파일링

Gradle 또는 Ant에서 오프라인 프로파일링을 시작할 때, 해당 JProfiler 플러그인을 사용하여 작업을 더 쉽게 할 수 있습니다. 테스트를 프로파일링하기 위한 Gradle 작업의 일반적인 사용 예는 아래와 같습니다:

```
plugins {
    id 'com.jprofiler' version 'X.Y.Z'
    id 'java'
}

jprofiler {
    installDir = file('/opt/jprofiler')
}

task run(type: com.jprofiler.gradle.TestProfile) {
    offline = true
    configFile = file("path/to/jprofiler_config.xml")
    sessionId = 1234
}
```

`com.jprofiler.gradle.JavaProfile` 작업은 표준 `JavaExec` 작업으로 실행하는 것과 동일한 방식으로 모든 Java 클래스를 프로파일링합니다. JProfiler에서 직접 지원하지 않는 JVM을 시작하는 다른 방법을 사용하는 경우, `com.jprofiler.gradle.SetAgentPathProperty` 작업을 사용하여 필요한 VM 매개변수를 속성에 기록할 수 있습니다. 이는 JProfiler 플러그인을 적용할 때 기본적으로 추가되므로, 다음과 같이 간단히 작성할 수 있습니다:

```
setAgentPathProperty {
    propertyName = 'agentPathProperty'
    offline = true
    configFile = file("path/to/jprofiler_config.xml")
    sessionId = 1234
}
```

그런 다음 작업이 실행된 후 다른 곳에서 `agentPathProperty`를 프로젝트 속성 참조로 사용할 수 있습니다. 모든 Gradle 작업의 기능과 해당 Ant 작업은 별도의 장 [p. 237]에서 자세히 문서화되어 있습니다.

실행 중인 JVM에 대한 오프라인 프로파일링 활성화

명령줄 유틸리티 `bin/jpenable`를 사용하여 버전 8 이상인 모든 실행 중인 JVM에서 오프라인 프로파일링을 시작할 수 있습니다. VM 매개변수와 마찬가지로, `offline` 스위치, 세션 ID 및 선택적 구성 파일을 지정해야 합니다:

```
jpenable --offline --id=12344 --config=/path/to/jprofiler_config.xml
```

이러한 호출을 통해 실행 중인 JVM 목록에서 프로세스를 선택해야 합니다. 추가 인수 `--pid=<PID>` `--noinput`를 사용하면 프로세스를 자동화하여 사용자 입력이 전혀 필요하지 않도록 할 수 있습니다.

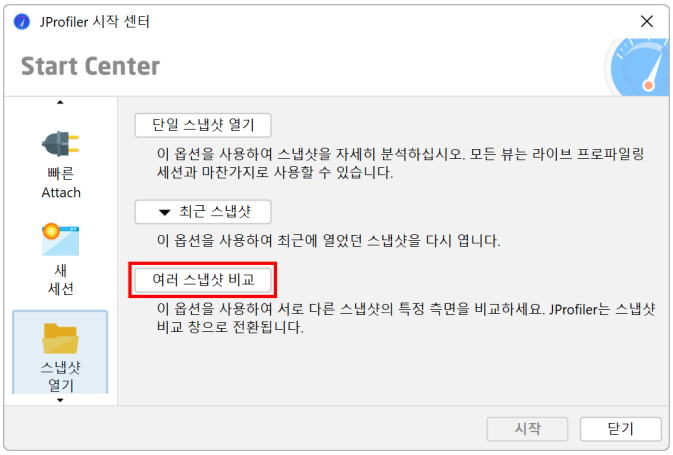
반면에, 실행 중에 오프라인 프로파일링을 활성화할 때는 수동으로 녹화를 시작하거나 스냅샷을 저장해야 할 수도 있습니다. 이는 `bin/jpcontroller` 명령줄 도구로 가능합니다.

프로파일링 에이전트가 로드되었지만 프로파일링 설정이 적용되지 않은 경우, 녹화 작업을 켤 수 없으므로 `jpcontroller`는 연결할 수 없습니다. 이는 `jpenable`을 사용하여 프로파일링을 활성화했지만 `offline` 매개변수 없이 활성화한 경우를 포함합니다. 오프라인 모드를 활성화하면 프로파일링 설정이 지정되며 `jpcontroller`를 사용할 수 있습니다.

`jpenable` 및 `jpcontroller` 실행 파일에 대한 자세한 정보는 명령줄 참조 [p. 237]에서 확인할 수 있습니다.

스냅샷 비교

현재 애플리케이션의 런타임 특성을 이전 버전과 비교하는 것은 성능 회귀를 방지하기 위한 일반적인 품질 보증 기술입니다. 또한, 단일 프로파일링 세션 내에서 성능 문제를 해결하는 데 유용할 수 있으며, 여기서 두 가지 다른 사용 사례를 비교하여 하나가 다른 것보다 느린 이유를 찾을 수 있습니다. 두 경우 모두 관심 있는 기록된 데이터를 포함한 스냅샷을 저장하고 메뉴에서 세션->새 창에서 스냅샷 비교를 호출하거나 시작 센터의 스냅샷 열기 탭에서 여러 스냅샷 비교 버튼을 클릭하여 JProfiler의 스냅샷 비교 기능을 사용합니다.

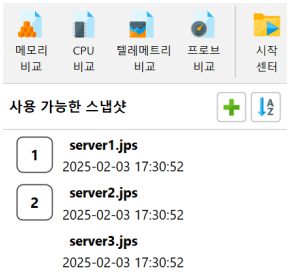


스냅샷 선택

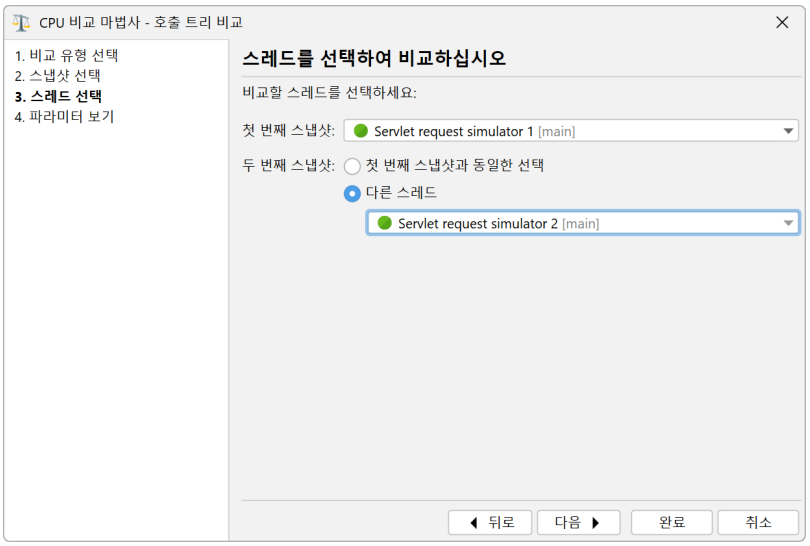
비교는 별도의 최상위 창에서 생성되고 표시됩니다. 먼저 스냅샷 선택기에 여러 스냅샷을 추가합니다. 그런 다음 관심 있는 스냅샷을 선택하고 비교 도구 모음 버튼을 클릭하여 나열된 스냅샷 중 두 개 이상에서 비교를 생성할 수 있습니다. 목록에서 스냅샷 파일의 순서는 중요합니다. 모든 비교는 목록에서 아래쪽에 있는 스냅샷이 나중에 기록되었다고 가정합니다. 스냅샷을 수동으로 배열하는 것 외에도 이름이나 생성 시간으로 정렬할 수 있습니다.

JProfiler의 메인 창에 있는 뷰와 달리, 비교 뷰에는 고정된 뷰 매개변수가 있으며, 이는 매개변수를 즉석에서 조정할 수 있는 드롭다운 목록 대신 상단에 표시됩니다. 모든 비교는 비교를 위한 매개변수를 수집하는 마법사를 표시하며, 동일한 매개변수로 여러 번 동일한 비교를 수행할 수 있습니다. 마법사는 이전 호출에서 매개변수를 기억하므로 여러 세트의 스냅샷을 비교할 때 구성을 반복할 필요가 없습니다. 언제든지 완료 버튼으로 마법사를 단축하거나 인덱스에서 단계를 클릭하여 다른 단계로 이동할 수 있습니다.

비교가 활성화되면 분석된 스냅샷이 번호 접두사와 함께 표시됩니다. 두 개의 스냅샷으로 작동하는 비교의 경우 표시된 차이는 스냅샷 2의 측정값에서 스냅샷 1의 측정값을 뺀 것입니다.

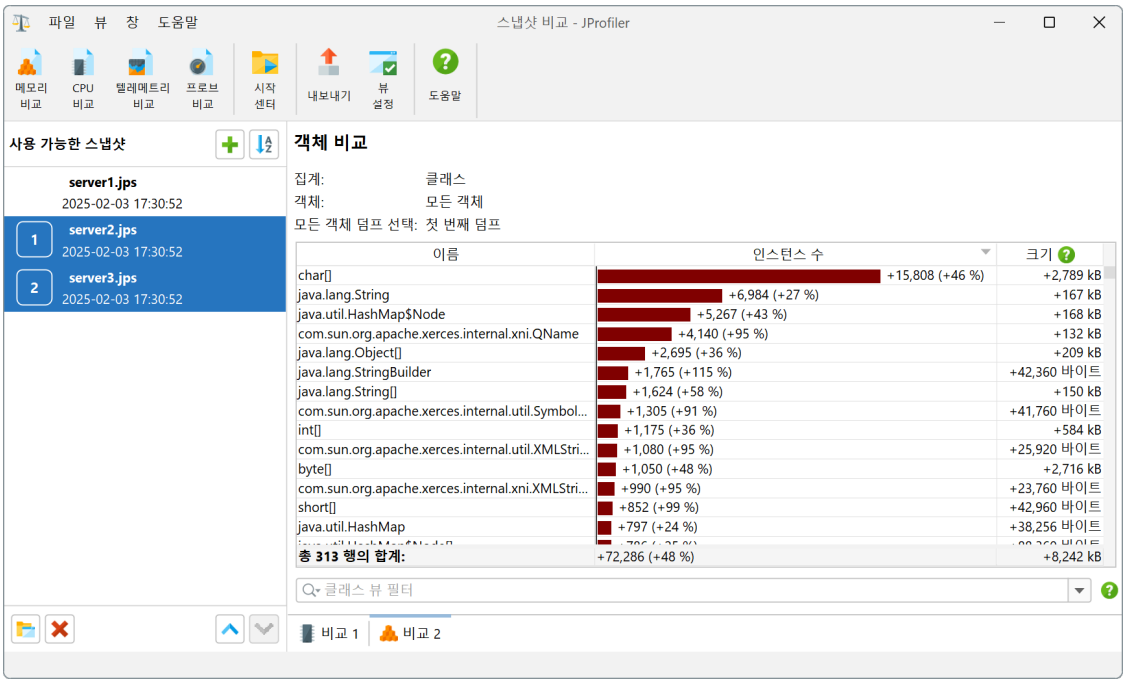


CPU 비교의 경우, 동일한 스냅샷을 첫 번째 및 두 번째 스냅샷으로 사용하고 마법사에서 다른 스레드 또는 스레드 그룹을 선택할 수 있습니다.



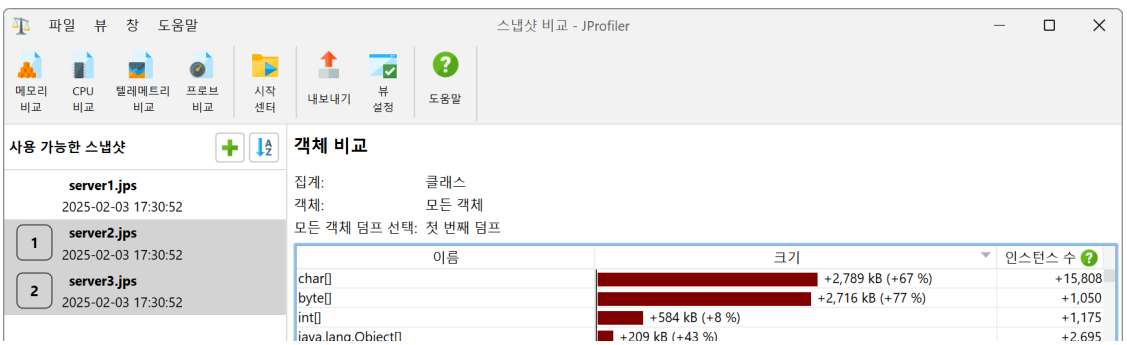
테이블과의 비교

가장 간단한 비교는 "객체" 메모리 비교입니다. 이는 힙 워커의 "모든 객체", "기록된 객체" 또는 "클래스" 뷰의 데이터를 비교할 수 있습니다. 비교의 열은 인스턴스 수와 크기의 차이를 보여주지만, 인스턴스 수 열만이 양방향 막대 차트를 보여주며, 증가가 빨간색으로 오른쪽에, 감소가 녹색으로 왼쪽에 그려집니다.

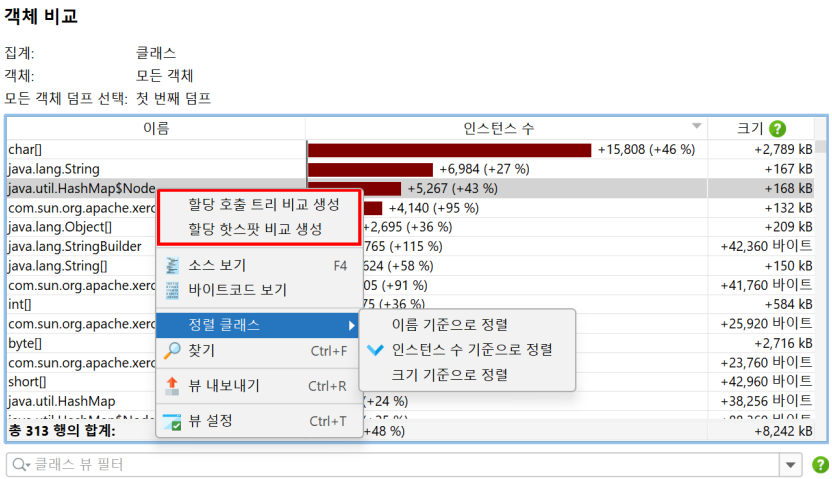


뷰 설정 대화 상자에서 이 막대 차트를 절대 변화량 또는 백분율로 표시할지 선택할 수 있습니다. 다른 값은 괄호 안에 표시됩니다. 이 설정은 또한 열이 정렬되는 방식을 결정합니다.

첫 번째 데이터 열의 측정값은 기본 측정값이라고 하며, 뷰 설정에서 기본 인스턴스 수에서 얕은 크기로 전환할 수 있습니다.



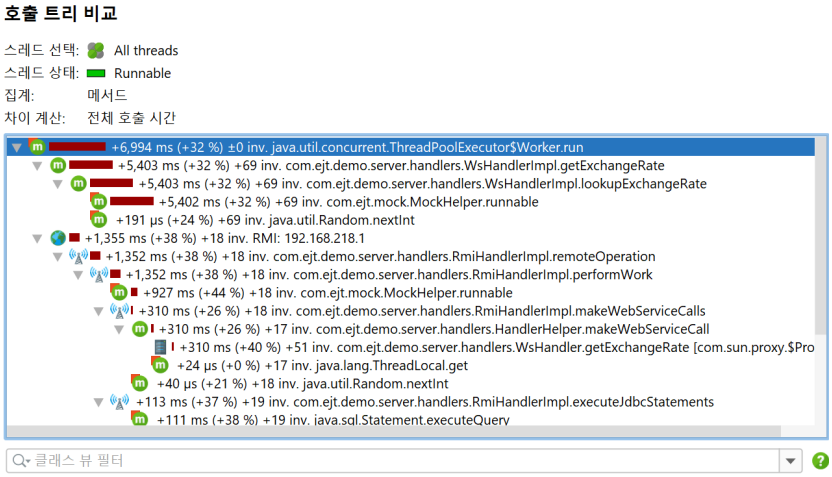
테이블의 컨텍스트 메뉴는 동일한 비교 매개변수와 선택된 클래스에 대해 다른 메모리 비교로의 단축키를 제공합니다.



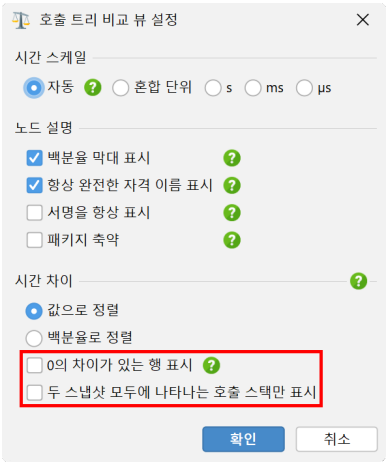
객체 비교와 마찬가지로, CPU 핫스팟, 프로브 핫스팟 및 할당 핫스팟 비교도 유사한 테이블로 표시됩니다.

트리와의 비교

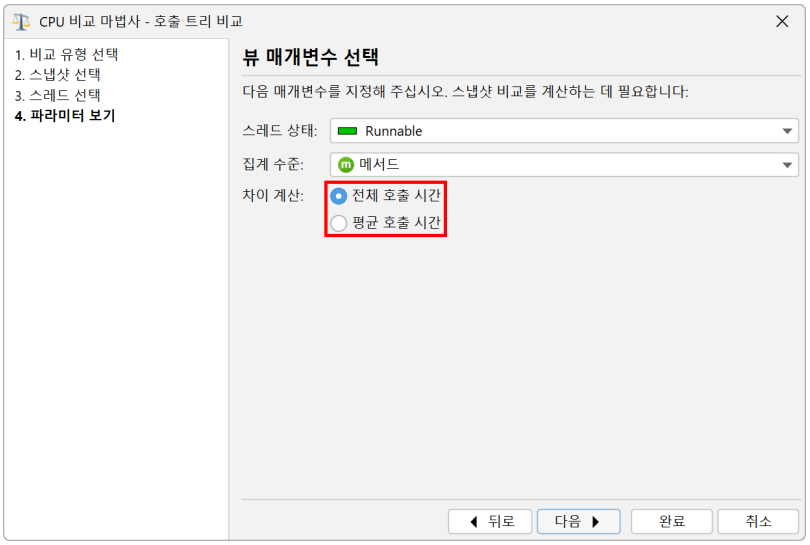
CPU 호출 트리, 할당 호출 트리 및 프로브 호출 트리 각각에 대해 선택된 스냅샷 간의 차이를 보여주는 또 다른 트리를 계산할 수 있습니다. 일반 호출 트리 뷰와 달리, 인라인 막대 다이어그램은 이제 변화량을 표시하며, 증가의 경우 빨간색, 감소의 경우 녹색으로 표시됩니다.



작업에 따라 두 스냅샷 파일 모두에 존재하고 하나의 스냅샷 파일에서 다른 스냅샷 파일로 변경된 호출 스택만 보는 것이 더 쉬울 수 있습니다. 뷰 설정 대화 상자에서 이 동작을 변경할 수 있습니다.

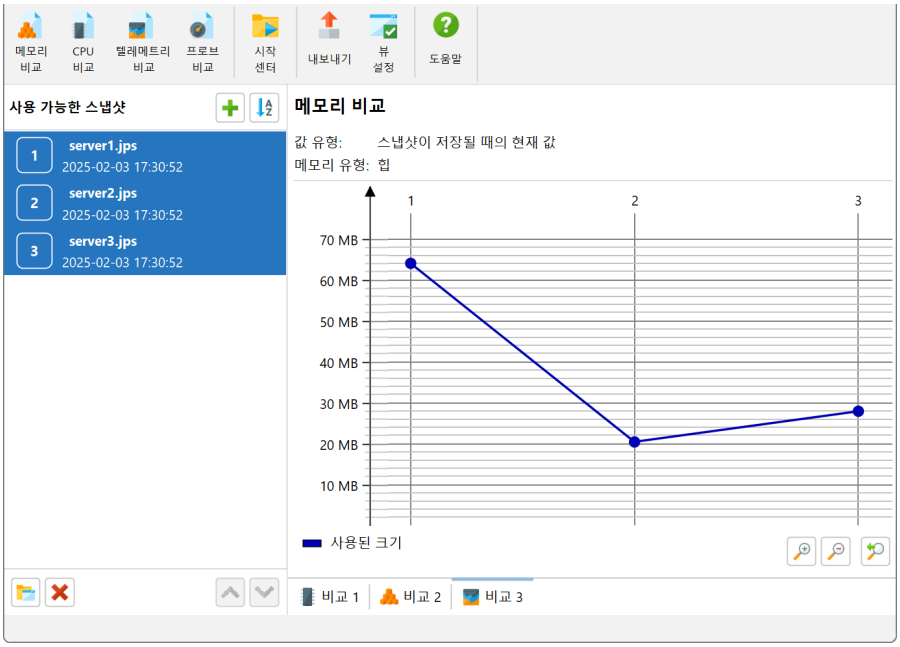


CPU 및 프로브 호출 트리 비교의 경우 총 시간 대신 평균 시간을 비교하는 것이 흥미로울 수 있습니다. 이는 마법사의 "뷰 매개변수" 단계에서 선택할 수 있는 옵션입니다.

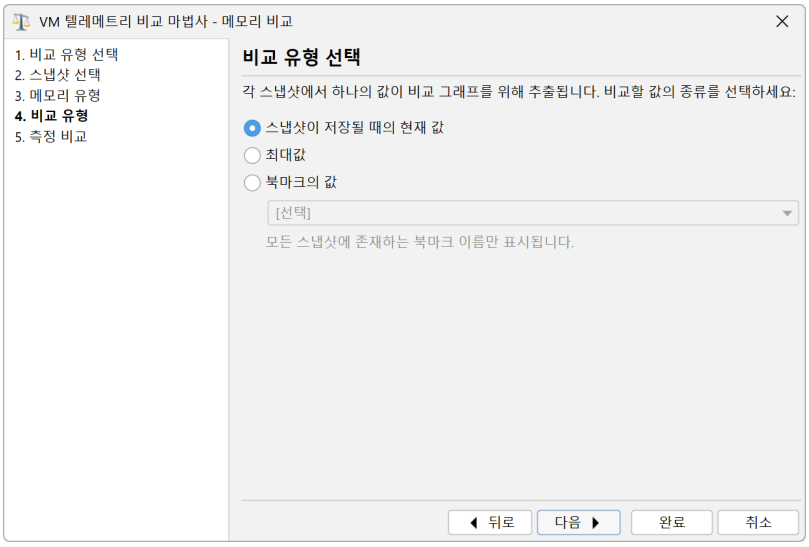


텔레메트리 비교

텔레메트리 비교에서는 동시에 두 개 이상의 스냅샷을 비교할 수 있습니다. 스냅샷 선택기에서 스냅샷을 선택하지 않으면, 마법사는 모든 스냅샷을 비교하려고 한다고 가정합니다. 텔레메트리 비교에는 시간 축이 없으며, 대신 선택된 스냅샷을 순서대로 x축에 표시합니다. 도구 팁에는 스냅샷의 전체 이름이 포함되어 있습니다.



비교는 각 스냅샷에서 하나의 숫자를 추출합니다. 텔레메트리 데이터는 시간에 따라 해상도가 있으므로 여러 가지 방법으로 수행할 수 있습니다. 마법사의 "비교 유형" 단계에서는 스냅샷이 저장될 때의 값을 사용하거나 최대값을 계산하거나 선택된 북마크에서 값을 찾는 옵션을 제공합니다.

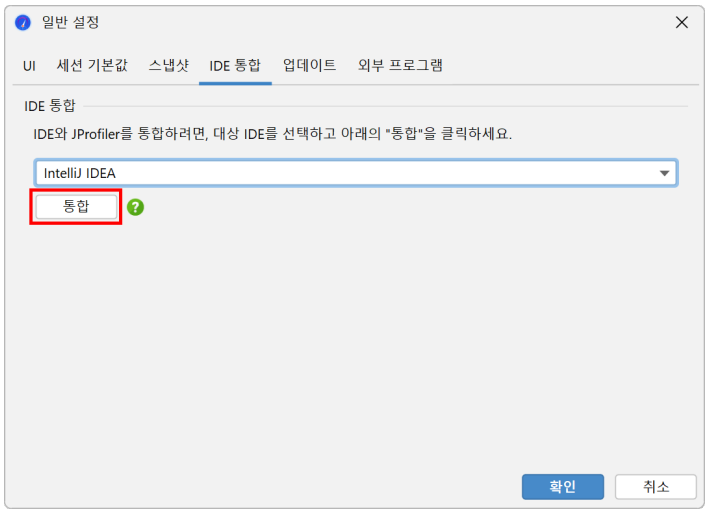


IDE 통합

애플리케이션을 프로파일링할 때, JProfiler의 뷰에 나타나는 메서드와 클래스는 종종 소스 코드를 확인해야만 답을 얻을 수 있는 질문을 유발합니다. JProfiler는 이를 위해 내장된 소스 코드 뷰어를 제공하지만, 기능이 제한적입니다. 또한, 문제가 발견되면 다음 단계는 보통 문제의 코드를 수정하는 것입니다. 이상적으로는 JProfiler의 프로파일링 뷰에서 IDE로 직접 연결되어 수동 검색 없이 코드를 검사하고 개선할 수 있어야 합니다.

IDE 통합 설치

JProfiler는 IntelliJ IDEA, eclipse 및 NetBeans에 대한 IDE 통합을 제공합니다. IDE 플러그인을 설치하려면 메인 메뉴에서 세션->IDE 통합을 호출하십시오. IntelliJ IDEA의 플러그인 설치는 IDE의 플러그인 관리로 수행되며, 다른 IDE의 경우 플러그인은 JProfiler에 의해 직접 설치됩니다. 설치 프로그램은 또한 JProfiler 설치와 함께 IDE 플러그인을 쉽게 업데이트할 수 있도록 이 작업을 제공합니다. 통합 마법사는 플러그인을 JProfiler의 현재 설치 디렉토리와 연결합니다. IDE 플러그인 설정에서 언제든지 사용 중인 JProfiler 버전을 변경할 수 있습니다. 플러그인과 JProfiler GUI 간의 프로토콜은 이전 버전의 JProfiler와도 호환되며 작동할 수 있습니다.

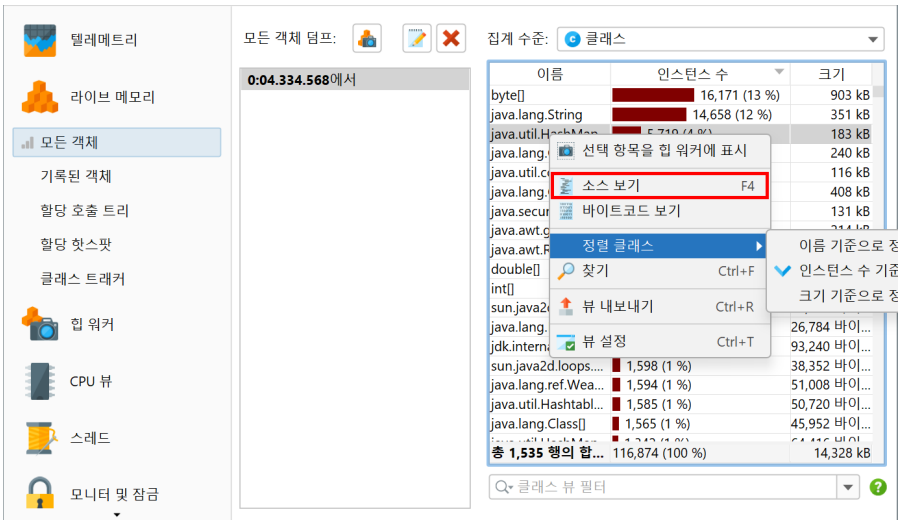


IntelliJ IDEA 통합은 플러그인 관리자에서도 설치할 수 있습니다. 이 경우, 처음 프로파일링할 때 플러그인은 JProfiler 실행 파일의 위치를 묻습니다.

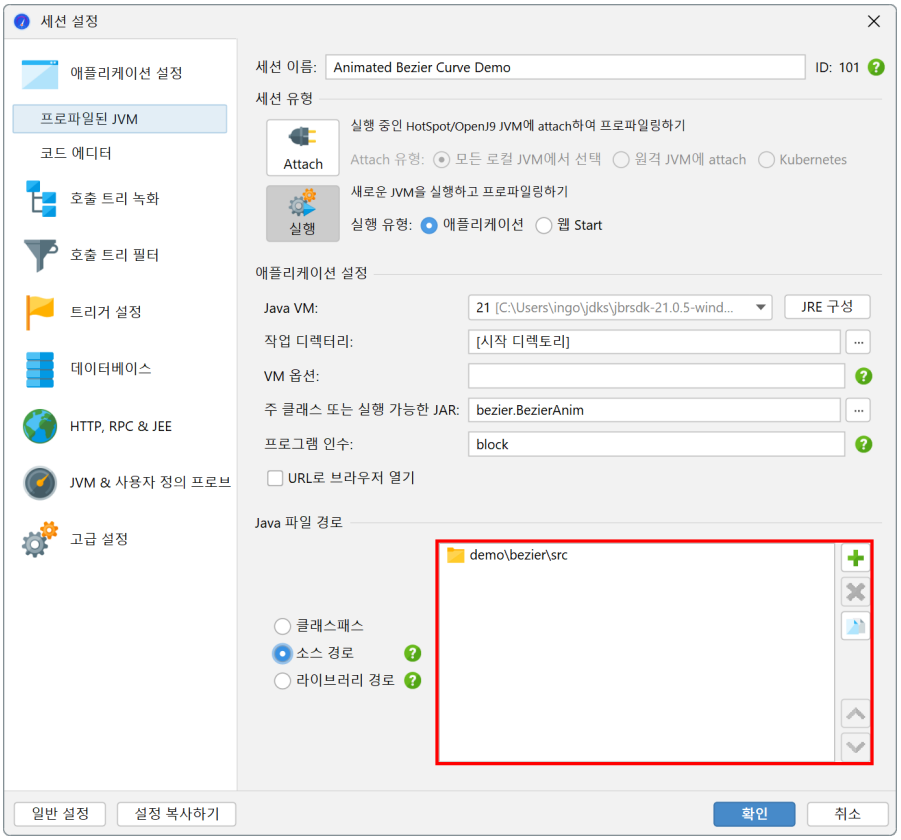
다양한 플랫폼에서 JProfiler 실행 파일은 다른 디렉토리에 위치합니다. Windows에서는 `bin\jprofiler.exe`, Linux 또는 Unix에서는 `bin/jprofiler`, macOS에서는 IDE 통합을 위한 JProfiler 애플리케이션 번들 내에 특별한 헬퍼 셸 스크립트 `Contents/Resources/app/bin/macos/jprofiler.sh`가 있습니다.

소스 코드 탐색

JProfiler에서 클래스 이름이나 메서드 이름이 표시되는 모든 곳에서 컨텍스트 메뉴에는 소스 보기 작업이 포함되어 있습니다.

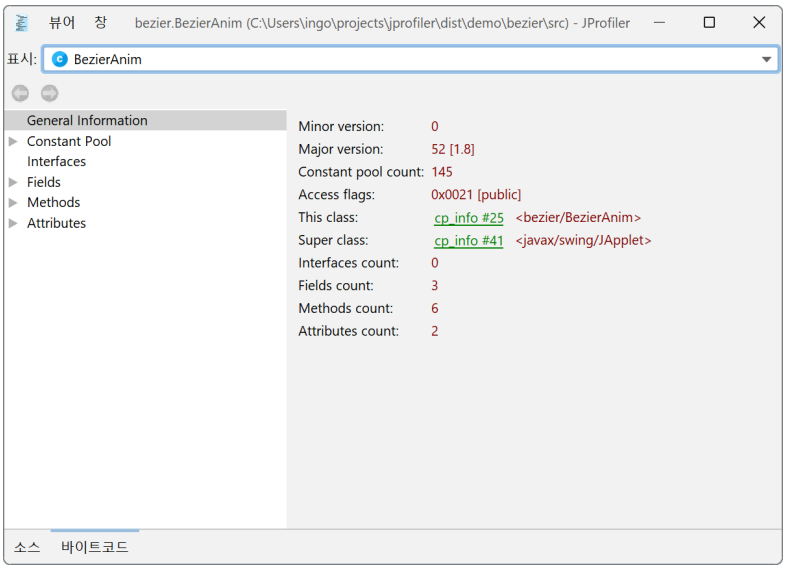


세션이 IDE에서 시작되지 않은 경우, 내장된 소스 코드 뷰어가 컴파일된 클래스 파일의 라인 번호 테이블을 사용하여 메서드를 찾습니다. 소스 파일은 애플리케이션 설정에서 루트 디렉토리 또는 포함된 ZIP 파일이 구성된 경우에만 찾을 수 있습니다.



소스 코드 표시와 함께, jclasslib 바이트코드 뷰어 [1]를 기반으로 한 바이트코드 뷰어가 컴파일된 클래스 파일의 구조를 보여줍니다.

[1] https://github.com/ingokegel/jclasslib



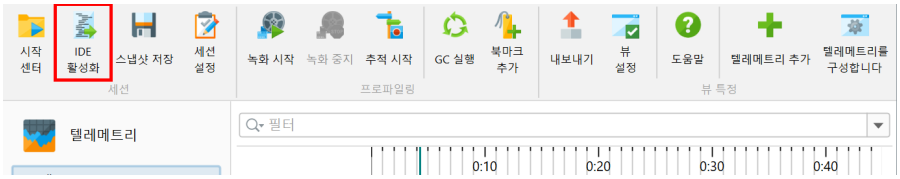
세션이 IDE에서 시작된 경우, 통합된 소스 코드 뷰어는 사용되지 않으며 소스 보기 작업은 IDE 플러그인으로 위임됩니다. IDE 통합은 시작된 프로파일링 세션, 저장된 스냅샷 열기 및 실행 중인 JVM에 attach하는 것을 지원합니다.

라이브 프로파일링 세션의 경우, 프로파일된 애플리케이션을 실행하거나 디버깅하는 것과 유사하게 IDE에서 시작합니다. JProfiler 플러그인은 프로파일링을 위한 VM 매개변수를 삽입하고 JProfiler 창을 연결합니다. JProfiler는 별도의 프로세스로 실행되며 필요할 경우 플러그인에 의해 시작됩니다. JProfiler의 소스 코드 탐색 요청은 IDE의 관련 프로젝트로 전송됩니다. JProfiler와 IDE 플러그인은 작업 표시줄 항목이 깜박이지 않고 창 전환을 매끄럽게 하기 위해 협력하여 단일 프로세스를 다루는 것처럼 보이게 합니다.

세션을 시작할 때, "세션 시작" 대화 상자에서 모든 프로파일링 설정을 구성할 수 있습니다. 시작된 세션에 사용되는 구성된 프로파일링 설정은 IDE 통합에 따라 프로젝트별 또는 실행 구성별로 JProfiler에 의해 기억됩니다. 세션이 처음 프로파일링될 때, IDE 플러그인은 소스 파일의 패키지 계층 구조에서 최상위 클래스에 기반하여 프로파일된 패키지 목록을 자동으로 결정합니다. 이후 언제든지 세션 설정 대화 상자의 필터 설정 단계로 이동하여 이 계산을 다시 수행하기 위해 재설정 버튼을 사용할 수 있습니다.

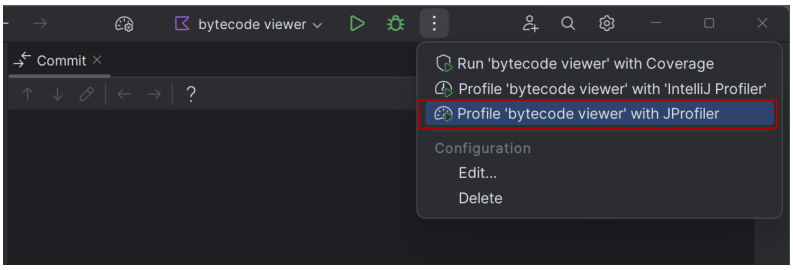
스냅샷의 경우, IDE 통합은 IDE 내에서 파일->열기 작업을 통해 스냅샷 파일을 열거나 프로젝트 창에서 더블 클릭하여 설정됩니다. JProfiler에서의 소스 코드 탐색은 현재 프로젝트로 안내됩니다. 마지막으로, IDE 플러그인은 실행 중인 JVM을 선택하고 스냅샷 메커니즘과 유사하게 IDE로 소스 코드 탐색을 가져오는 JVM에 Attach 작업을 IDE에 추가합니다.

때로는 특정 클래스나 메서드를 염두에 두지 않고 IDE로 전환하고 싶을 수 있습니다. 이를 위해 JProfiler 창의 도구 모음에는 IDE 통합에 의해 열리는 프로파일링 세션에 대해 표시되는 IDE 활성화 버튼이 있습니다. 이 작업은 IDE에서 JProfiler 활성화 작업과 마찬가지로 F11 키에 바인딩되어 있어 동일한 키 바인딩으로 IDE와 JProfiler 간을 자유롭게 전환할 수 있습니다.

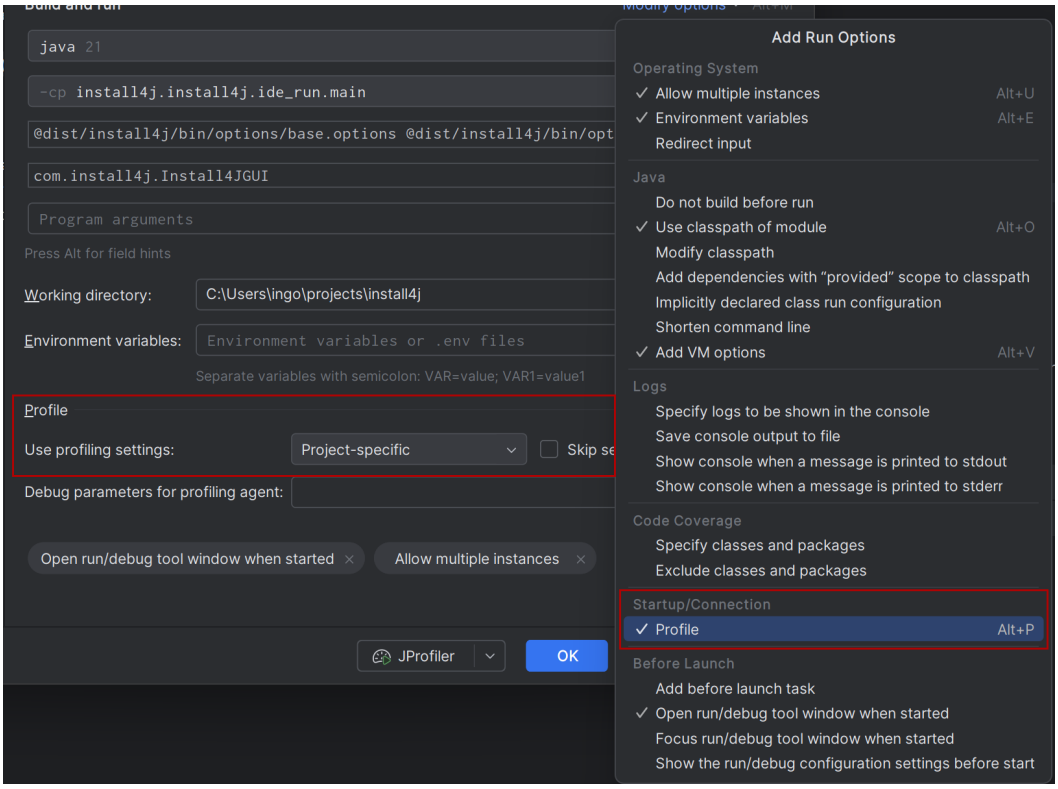


IntelliJ IDEA 통합

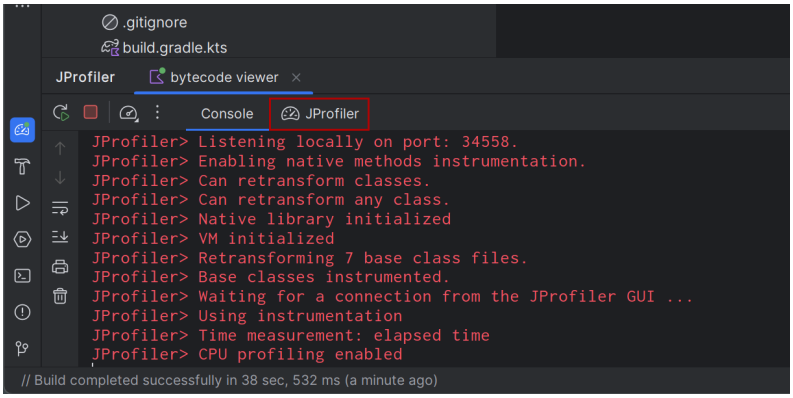
IntelliJ IDEA에서 애플리케이션을 프로파일링하려면 실행 메뉴에서 프로파일링 명령 중 하나를 선택하거나 메인 도구 모음의 실행 또는 디버그 작업 옆 드롭다운 메뉴를 클릭하여 "JProfiler로 프로파일" 작업을 선택하십시오. JProfiler는 애플리케이션 서버를 포함한 대부분의 실행 IDEA 구성 유형을 프로파일링할 수 있습니다.



JProfiler 플러그인은 실행 구성에 추가 설정을 추가하며, 이는 즉시 보이지 않습니다. 이러한 설정에 액세스하려면 "옵션 수정" 드롭다운에서 "프로파일" 옵션을 선택하십시오. 다른 모든 프로파일링 설정은 JProfiler 창의 시작 대화 상자에서 구성할 수 있습니다.



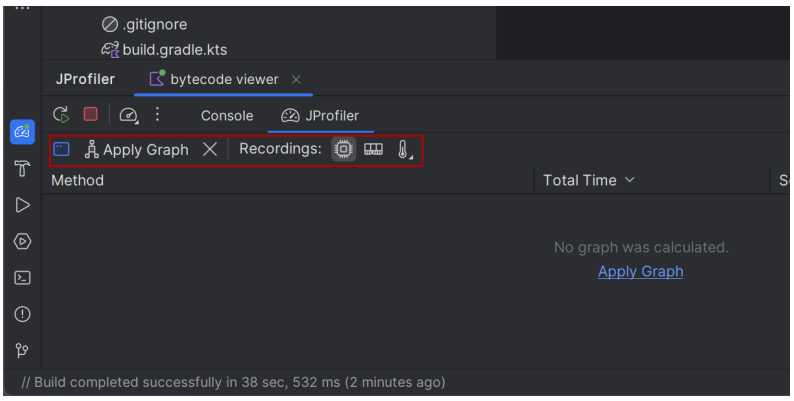
프로파일링 세션이 시작되면 출력은 별도의 JProfiler 도구 창에 나타납니다. 해당 도구 창은 JProfiler UI에 연결한 후 사용할 수 있는 "JProfiler" 탭과 함께 일반 실행 도구 창과 같은 콘솔 출력을 표시합니다:



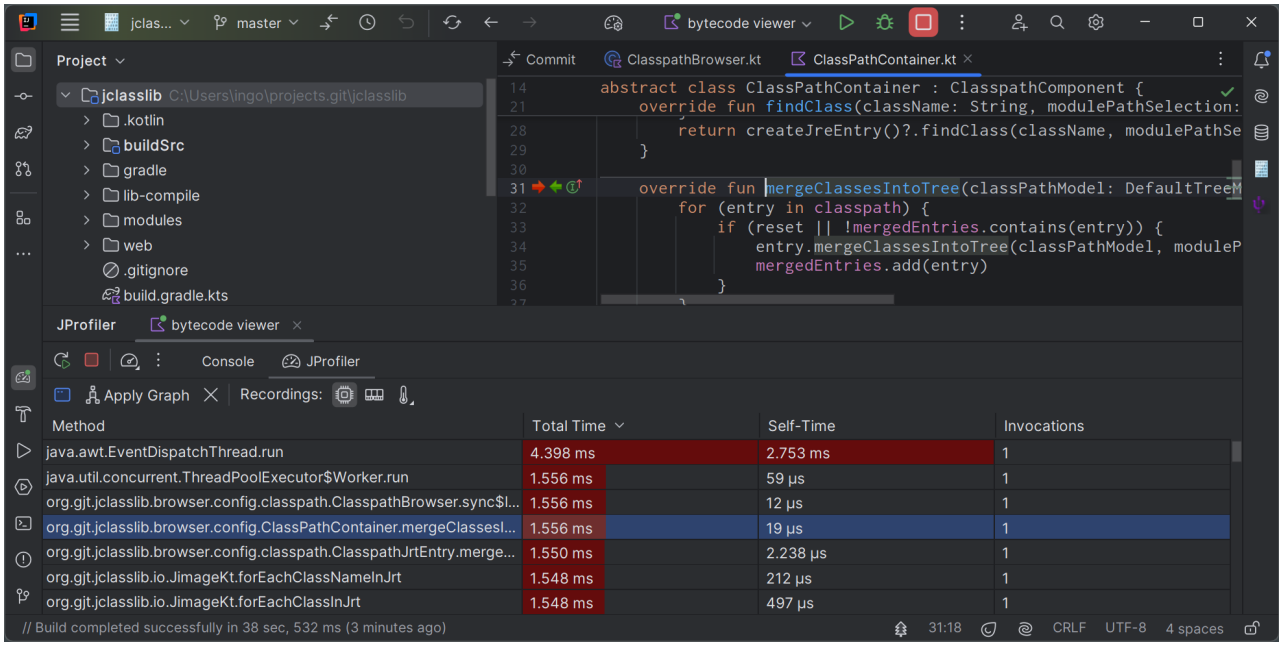
JProfiler 도구 창은 IntelliJ IDEA에서 JProfiler 스냅샷을 열거나 "JProfiler로 JVM에 Attach" 작업으로 실행 중인 JVM에 attach할 때도 표시됩니다.

"JProfiler" 탭에는 CPU 데이터, 할당 데이터 및 프로브 이벤트에 대한 데이터 기록을 시작하고 중지하는 작업이 포함되어 있습니다. 또한 JProfiler 창으로 전환할 수 있는 작업도 포함되어 있습니다. JProfiler 창에는 IDEA 창으로 다시 전환할 수 있는 유사한 작업이 포함되어 있어 두 개의 별도 창을 편리하게 사용할 수 있습니다. JProfiler에서 IntelliJ IDEA로의 정밀한 소스 코드 탐색은 Java와 Kotlin에 대해 구현되어 있습니다.

프로파일링 정보는 일반적으로 JProfiler 창에 표시되지만, CPU 그래프 데이터는 소스 코드에 직접 이 데이터를 표시하는 것이 의미가 있기 때문에 IntelliJ IDEA UI에도 통합됩니다. IntelliJ IDEA에서 "그래프 적용" 작업을 사용하거나 JProfiler에서 CPU 그래프를 생성하여 IntelliJ IDEA 내에서 CPU 데이터를 표시하십시오. 스레드 선택과 같은 고급 매개변수를 구성하거나 호출 트리 루트, 호출 트리 제거 및 호출 트리 뷰 필터 설정을 호출 트리 뷰에서 사용하려면 JProfiler 창에서 그래프를 생성해야 합니다.

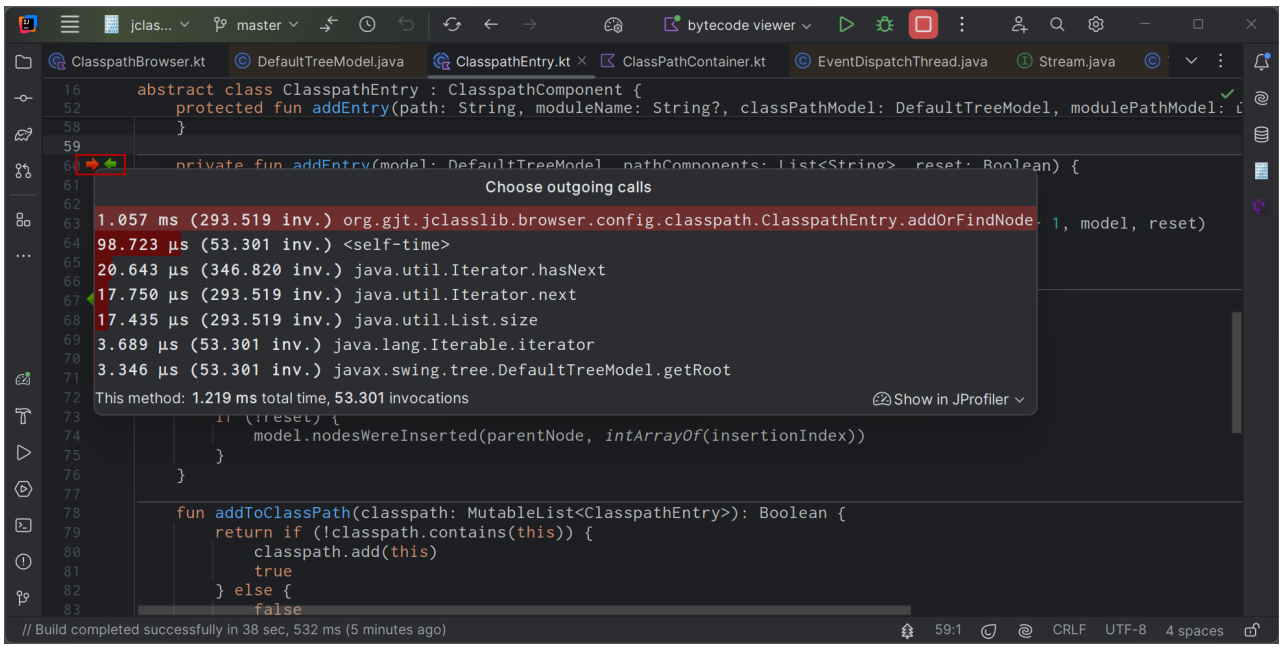


CPU 데이터가 적용되면 "JProfiler" 탭에는 기록된 메서드 목록이 표시됩니다. 메서드를 더블 클릭하면 소스 코드로 이동합니다. 소스 코드 편집기의 구석에는 들어오고 나가는 호출에 대한 화살표가 추가됩니다.

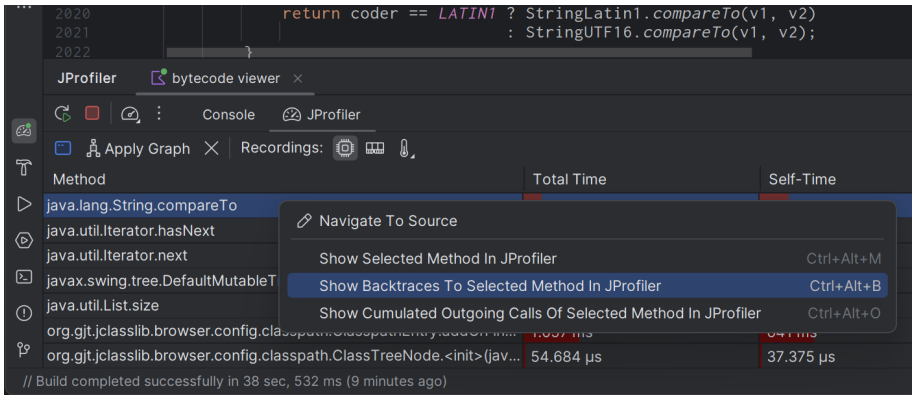


구석 아이콘을 클릭하면 팝업 창에 들어오고 나가는 메서드가 표시되며, 기록된 시간을 보여주는 막대 차트가 함께 표시됩니다. 팝업의 행을 클릭하면 해당 메서드로 이동합니다.

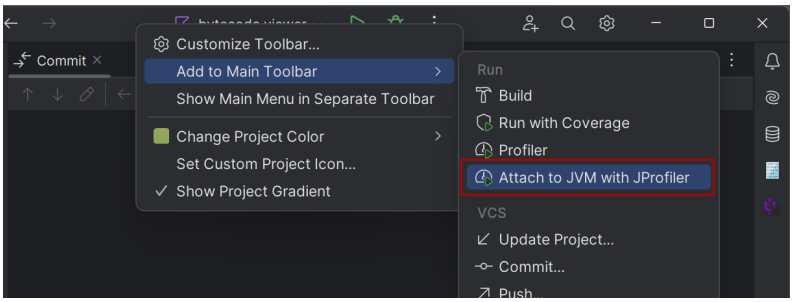
또한 팝업 하단에는 대상 메서드의 총 기록 시간과 호출 횟수가 표시됩니다. 팝업의 오른쪽 하단에 있는 "JProfiler에서 보기" 드롭다운은 JProfiler UI로의 컨텍스트 종속 탐색 작업을 제공합니다. 선택한 노드 또는 메서드 그래프에서 해당 호출 트리 분석을 표시할 수 있습니다. 나가는 호출의 경우 "누적 나가는 호출" 분석이 제공되며, 들어오는 호출의 경우 "백트레이스" 분석이 제공됩니다.



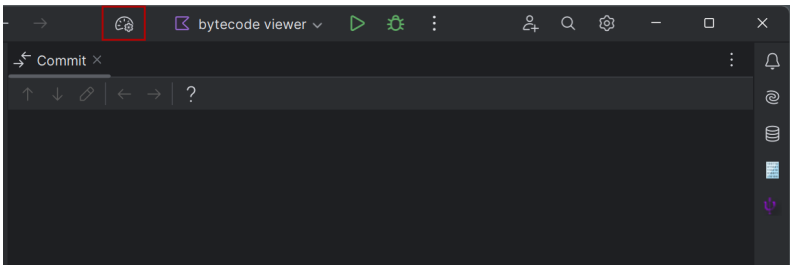
동일한 탐색 작업은 "JProfiler" 탭의 메서드 테이블의 컨텍스트 메뉴에서도 사용할 수 있습니다:



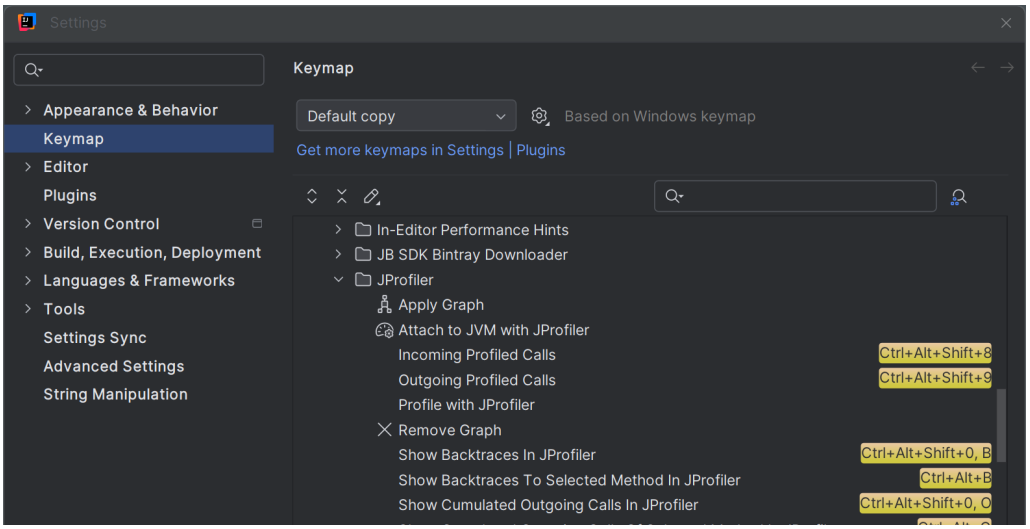
JProfiler 플러그인은 메인 도구 모음에 추가할 수 있는 "JProfiler로 JVM에 Attach" 작업에 대한 도구 모음 빠른 작업을 제공합니다. 이 작업을 통해 이미 실행 중인 프로세스에 attach하고 JProfiler UI에서 IntelliJ IDEA로의 소스 코드 탐색뿐만 아니라 소스 코드 편집기에서 인라인 CPU 그래프 데이터를 얻을 수 있습니다:



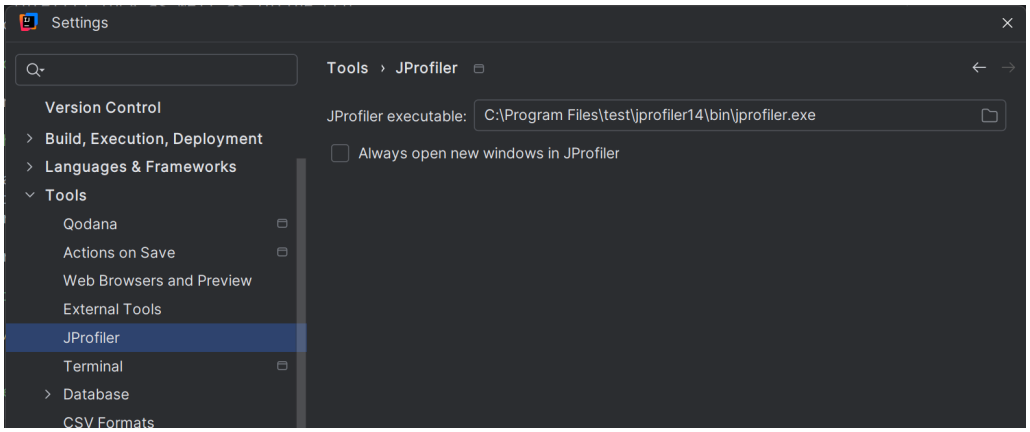
작업 버튼이 추가되면 이렇게 보입니다:



JProfiler의 모든 작업에 대한 키 바인딩은 IntelliJ IDEA의 "키맵" 설정에서 사용자 정의할 수 있습니다. 충돌하지 않는 키보드 단축키의 제한된 가용성을 감안할 때, 소스 코드 편집기에서 JProfiler UI로의 탐색 작업은 `Ctrl-Alt-Shift-O`를 먼저 누르고 탐색 작업을 선택하기 위해 다른 키를 누르는 체인 단축키입니다. 이 기능을 자주 사용하는 경우 더 간단한 키보드 단축키를 할당할 수 있습니다.



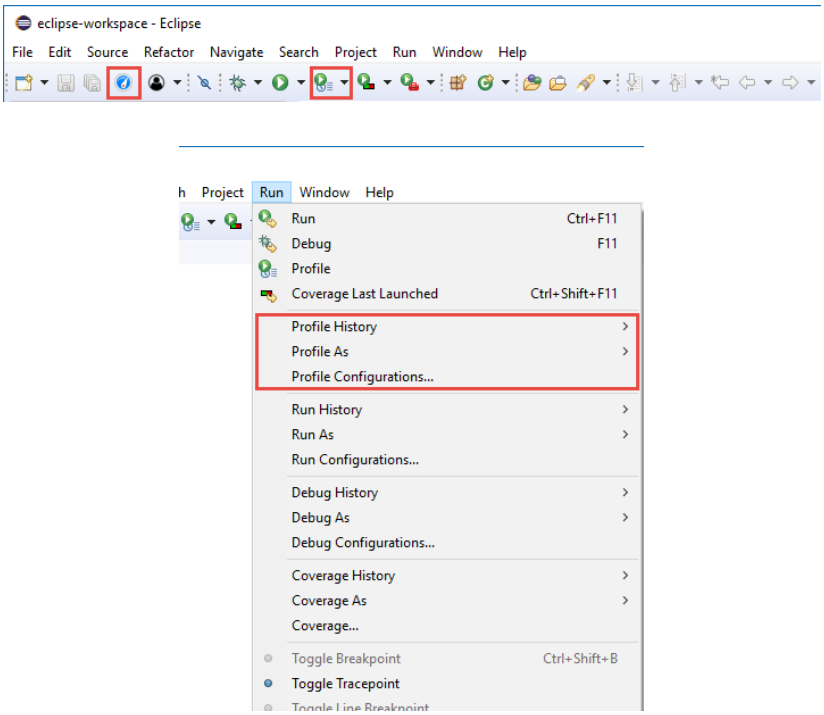
IDE 설정의 도구->JProfiler 페이지에서 사용 중인 JProfiler 실행 파일과 새로운 프로파일링 세션에 대해 항상 새로운 창을 열지 여부를 조정할 수 있습니다.



Eclipse 통합

eclipse 플러그인은 테스트 실행 구성 및 WTP 실행 구성을 포함한 대부분의 일반 실행 구성 유형을 프로파일링할 수 있습니다. eclipse 플러그인은 eclipse 프레임워크의 부분 설치가 아닌 전체 eclipse SDK와 함께만 작동합니다.

eclipse에서 애플리케이션을 프로파일링하려면 실행 메뉴에서 프로파일링 명령 중 하나를 선택하거나 해당 도구 모음 버튼을 클릭하십시오. 프로파일 명령은 eclipse의 디버그 및 실행 명령과 동일하며, 실행->JProfiler를 JVM에 Attach 메뉴 항목을 제외하고 eclipse의 인프라의 일부입니다. 이 메뉴 항목은 JProfiler 플러그인에 의해 추가됩니다.

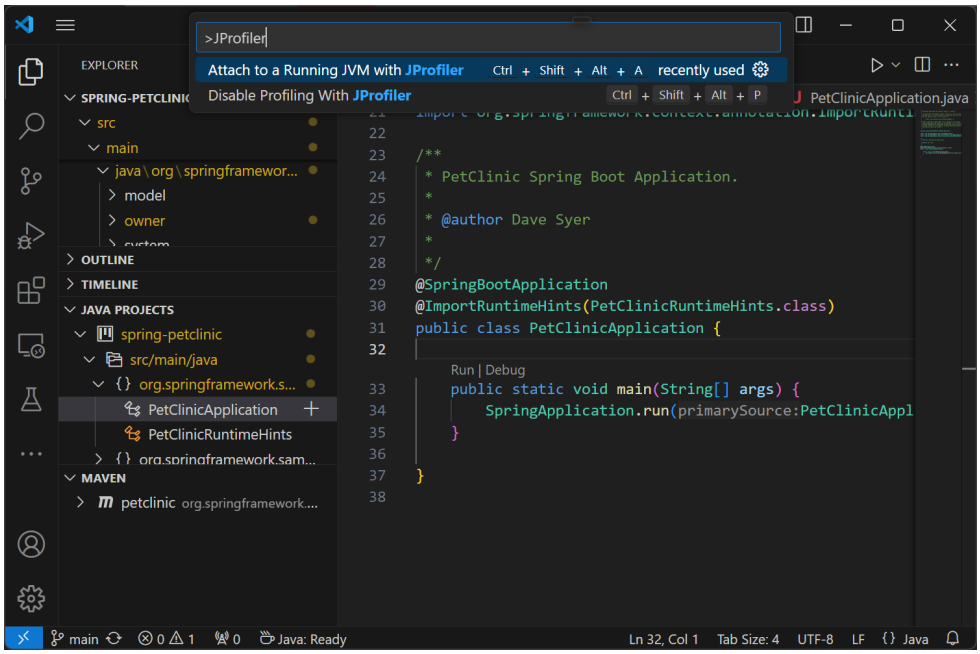


Java 관점에서 실행->프로파일 ... 메뉴 항목이 존재하지 않는 경우, 창->관점->관점 사용자 정의에서 "프로파일" 작업을 이 관점에 대해 활성화하여 작업 세트 가용성 탭을 앞으로 가져오고 프로파일 체크박스를 선택하십시오.

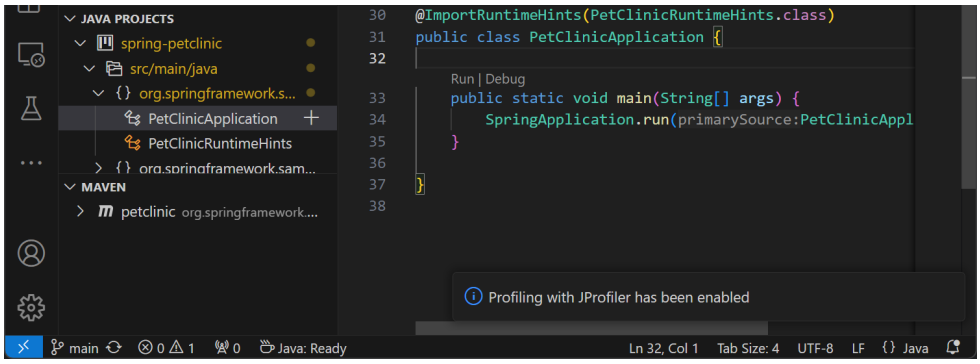
JProfiler 실행 파일의 위치를 포함한 여러 JProfiler 관련 설정은 eclipse의 창->환경 설정->JProfiler에서 조정할 수 있습니다.

VS Code 통합

VS Code 확장은 `JProfiler` 작업을 추가합니다. 호출되면, 디버그 및 실행 작업은 Java 실행 구성을 위한 프로파일링을 시작합니다. JProfiler가 시작되고, 프로파일링 설정을 구성할 수 있는 세션 시작 대화 상자가 표시됩니다. 세션 시작 대화 상자가 확인되면 애플리케이션이 시작됩니다.



`JProfiler` 작업을 사용하면 실행 및 디버그 작업의 기본 동작이 복원됩니다. 프로파일링 모드의 변경 사항에 대한 알림은 VS Code의 편집기 하단 오른쪽 모서리에 토스트 메시지로 표시됩니다. `JProfiler` 및 `JProfiler` 는 동일한 기본 키 바인딩을 가지므로 프로파일링 모드를 토글하는 데 사용할 수 있습니다.



이미 실행 중인 JVM을 프로파일링하려면 `JProfiler` `JVM` `Attach` 작업을 사용하십시오.

JProfiler의 소스 탐색 작업은 VS Code에서 해당 소스 코드를 표시합니다. JProfiler 스냅샷에 대한 VS Code로의 소스 탐색을 얻으려면 VS Code 내에서 파일->열기를 사용하여 스냅샷을 여십시오.

NetBeans 통합

NetBeans에서는 exec Maven 플러그인을 사용하는 표준, 자유 형식 및 Maven 프로젝트를 프로파일링할 수 있습니다. NetBeans에서 애플리케이션을 프로파일링하려면 실행 메뉴에서 프로파일링 명령 중 하나를 선택하거나 해당 도구 모음 버튼을 클릭하십시오. 애플리케이션을 다른 방식으로 시작하는 Maven 프로젝트와

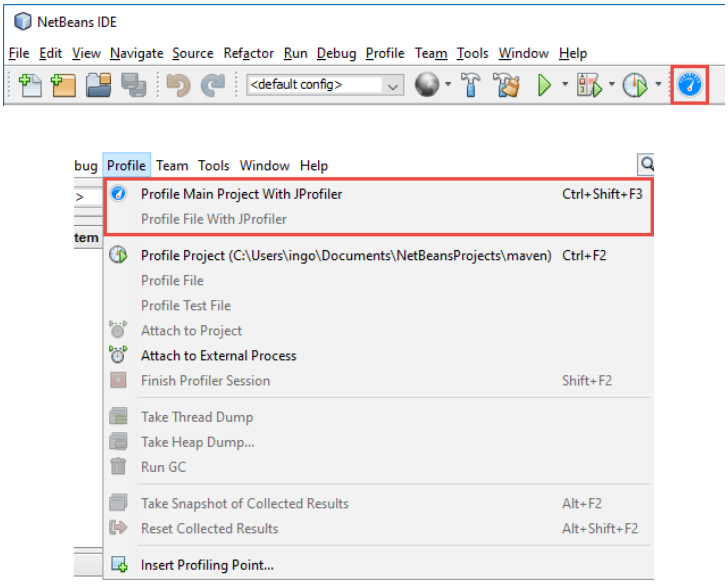
Gradle 프로젝트의 경우, 프로젝트를 정상적으로 시작하고 메뉴에서 프로파일->JProfiler를 실행 중인 JVM에 Attach 작업을 사용하십시오.

자유 형식 프로젝트의 경우, 프로파일링을 시도하기 전에 애플리케이션을 한 번 디버그해야 합니다. 필요한 파일`nbproject/ide-targets.xml`은 디버그 작업에 의해 설정됩니다. JProfiler는 디버그 대상과 동일한 내용을 가진 "profile-jprofiler"라는 대상을 추가하고 필요한 경우 VM 매개변수를 수정하려고 시도합니다. 자유 형식 프로젝트를 프로파일링하는 데 문제가 있는 경우, 이 대상의 구현을 확인하십시오.

통합된 Tomcat 또는 NetBeans에 구성된 다른 Tomcat 서버로 웹 애플리케이션을 프로파일링할 수 있습니다. 메인 프로젝트가 웹 프로젝트인 경우, JProfiler로 메인 프로젝트 프로파일을 선택하면 프로파일링이 활성화된 상태로 Tomcat 서버가 시작됩니다.

번들된 GlassFish Server와 함께 NetBeans를 사용하고 메인 프로젝트가 GlassFish Server를 사용하도록 설정된 경우, JProfiler로 메인 프로젝트 프로파일을 선택하면 프로파일링이 활성화된 상태로 애플리케이션 서버가 시작됩니다.

JProfiler 실행 파일의 위치와 새로운 JProfiler 창을 여는 정책은 옵션 대화 상자의 기타->JProfiler에서 조정할 수 있습니다.



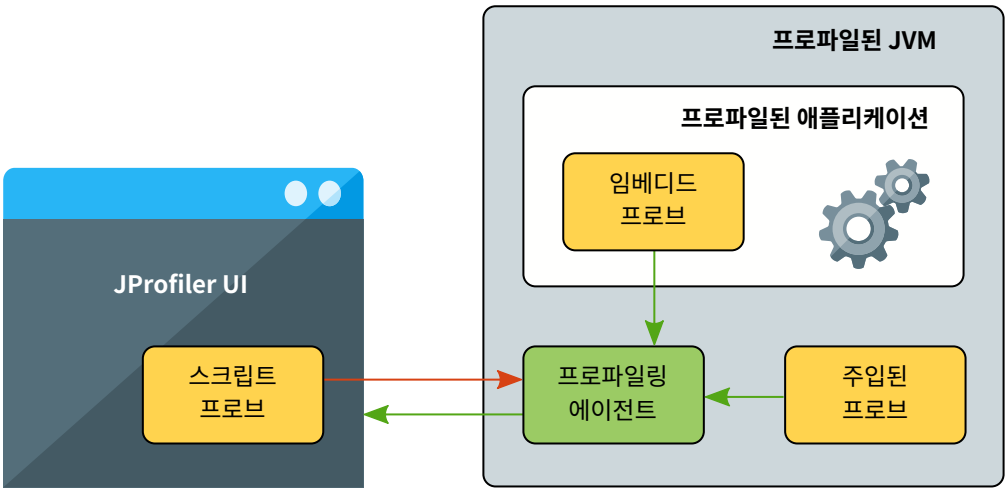
A 사용자 정의 프로브

A.1 프로브 개념

JProfiler에 대한 사용자 정의 프로브를 개발하려면 몇 가지 기본 개념과 용어를 알고 있어야 합니다. JProfiler의 모든 프로브의 공통 기반은 특정 메소드를 가로채고 가로챈 메소드 매개변수 및 기타 데이터 소스를 사용하여 JProfiler UI에서 보고 싶은 흥미로운 정보를 포함하는 문자열을 생성하는 것입니다.

프로브를 정의할 때 초기 문제는 가로챈 메소드를 지정하고 문자열을 생성하기 위해 메소드 매개변수 및 기타 관련 객체를 사용할 수 있는 환경을 얻는 방법입니다. JProfiler에서는 이를 수행하는 세 가지 방법이 있습니다:

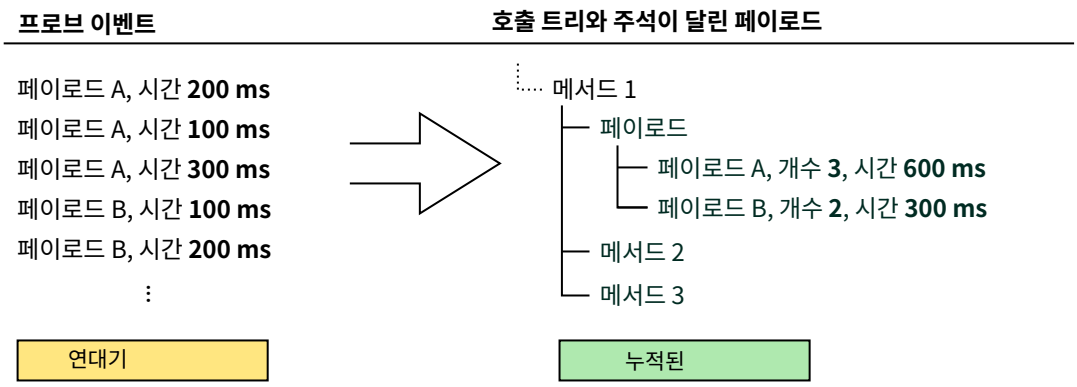
- **스크립트 프로브** [p. 149]는 JProfiler UI에서 완전히 정의됩니다. 호출 트리에서 메소드에 오른쪽 클릭하여 스크립트 프로브 작업을 선택하고 내장된 코드 편집기에서 문자열에 대한 표현식을 입력할 수 있습니다. 이는 프로브를 실험하는 데 유용하지만 사용자 정의 프로브의 기능 중 매우 제한된 부분만 노출합니다.
- **임베디드 프로브** [p. 158] API는 사용자 코드에서 호출할 수 있습니다. 라이브러리, 데이터베이스 드라이버 또는 서버를 작성하는 경우 제품과 함께 프로브를 제공할 수 있습니다. JProfiler로 제품을 프로파일링하는 사람은 JProfiler UI에 자동으로 프로브가 추가됩니다.
- **인젝티드 프로브** [p. 153] API를 사용하여 IDE에서 타사 소프트웨어에 대한 프로브를 작성할 수 있으며, JProfiler의 프로브 시스템의 전체 기능을 사용할 수 있습니다. API는 주석을 사용하여 인터셉션을 정의하고 메소드 매개변수 및 기타 유용한 객체를 주입합니다.



다음 질문은: JProfiler가 생성한 문자열로 무엇을 해야 하는가입니다. 두 가지 다른 전략이 있습니다: 페이로드 생성 또는 호출 트리 분할.

페이로드 생성

프로브에 의해 생성된 문자열은 **프로브 이벤트**를 생성하는 데 사용할 수 있습니다. 이벤트는 해당 문자열로 설정된 설명과 가로챈 메소드의 호출 시간과 동일한 지속 시간, 그리고 관련 호출 스택을 가집니다. 해당 호출 스택에서 프로브 설명과 타이밍이 누적되어 호출 트리에 **페이로드**로 저장됩니다. 이벤트는 일정 최대 수 이후에 통합되지만, 호출 트리의 누적된 페이로드는 전체 녹화 기간 동안의 총 수치를 보여줍니다. CPU 데이터와 프로브가 모두 녹화되면, 프로브 호출 트리 뷰는 페이로드 문자열을 리프 노드로 사용하여 병합된 호출 스택을 보여주고, CPU 호출 트리 뷰는 프로브 호출 트리 뷰로의 **주석 링크**를 포함합니다.



CPU 데이터와 마찬가지로 페이로드는 호출 트리 또는 핫스팟 뷰에서 표시될 수 있습니다. 핫스팟은 대부분의 소모된 시간에 책임이 있는 페이로드를 보여주고, 백 트레이스는 이러한 페이로드를 생성하는 코드의 부분을 보여줍니다. 좋은 핫스팟 목록을 얻으려면 페이로드 문자열에 고유 ID나 타임스탬프가 포함되지 않아야 합니다. 모든 페이로드 문자열이 다르면 누적이 없고 핫스팟의 명확한 분포가 없기 때문입니다. 예를 들어, 준비된 JDBC 문장의 경우 매개변수를 페이로드 문자열에 포함해서는 안 됩니다.

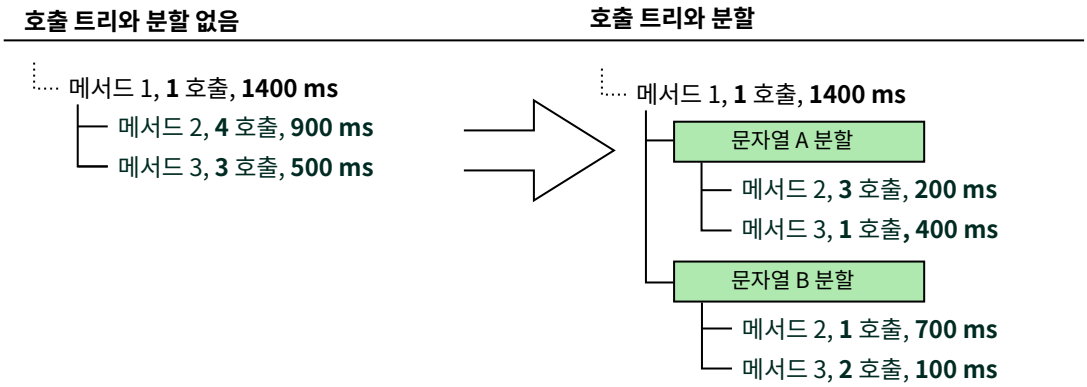
스크립트 프로브는 구성된 스크립트의 반환 값에서 자동으로 페이로드를 생성합니다. 인젝티드 프로브는 유사하게 `PayloadInterception`으로 주석이 달린 인터셉션 핸들러 메소드에서 문자열 또는 고급 기능을 위한 `Payload` 객체로 페이로드 설명을 반환합니다. 반면 임베디드 프로브는 `Payload.exit`를 호출하여 페이로드 설명을 인수로 사용하여 페이로드를 생성하며, `Payload.enter`와 `Payload.exit` 사이의 시간이 프로브 이벤트 지속 시간으로 기록됩니다.

페이로드 생성은 서로 다른 호출 사이트에서 발생하는 서비스 호출을 기록할 때 가장 유용합니다. 일반적인 예는 데이터베이스 드라이버로, 페이로드 문자열은 쿼리 문자열 또는 명령의 형태입니다. 프로브는 작업이 측정되는 호출 사이트의 관점을 취하며, 이 작업은 다른 소프트웨어 구성 요소에 의해 수행됩니다.

호출 트리 분할

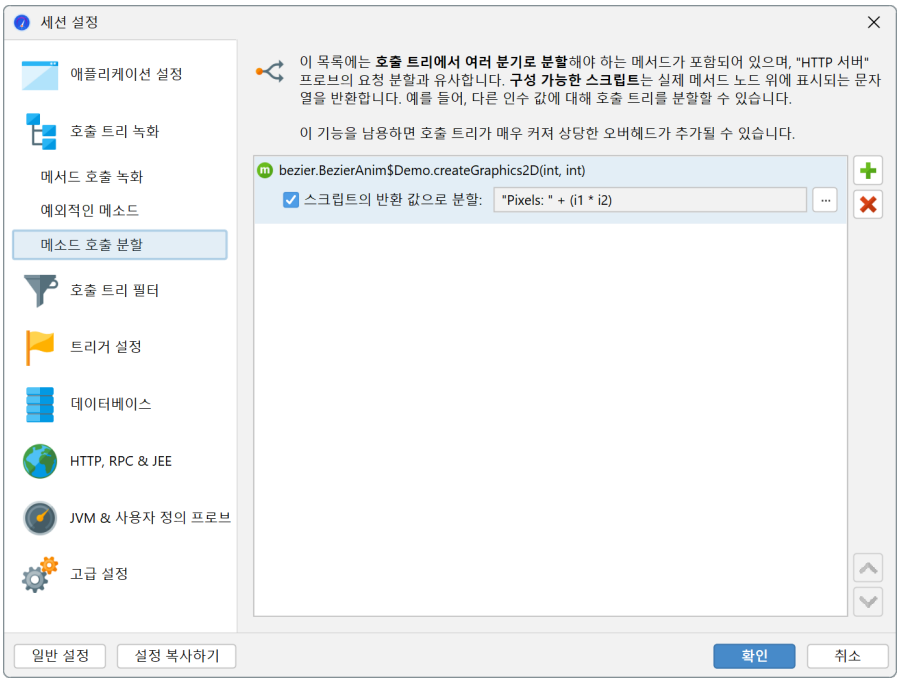
프로브는 실행 사이트의 관점을 취할 수도 있습니다. 이 경우 가로챈 메소드가 어떻게 호출되는지는 중요하지 않으며, 오히려 그 후에 어떤 메소드 호출이 실행되는지가 중요합니다. 일반적인 예는 추출된 문자열이 URL인 서블릿 컨테이너에 대한 프로브입니다.

페이로드를 생성하는 것보다 더 중요한 것은 프로브에 의해 생성된 각 고유 문자열에 대해 호출 트리를 분할할 수 있는 능력입니다. 각 문자열에 대해 호출 트리에 분할 노드가 삽입되어 모든 관련 호출의 누적 호출 트리를 포함합니다. 그렇지 않으면 하나의 누적 호출 트리만 있을 것이지만, 이제는 호출 트리를 서로 다른 부분으로 세분화하여 개별적으로 분석할 수 있는 분할 노드 집합이 있습니다.



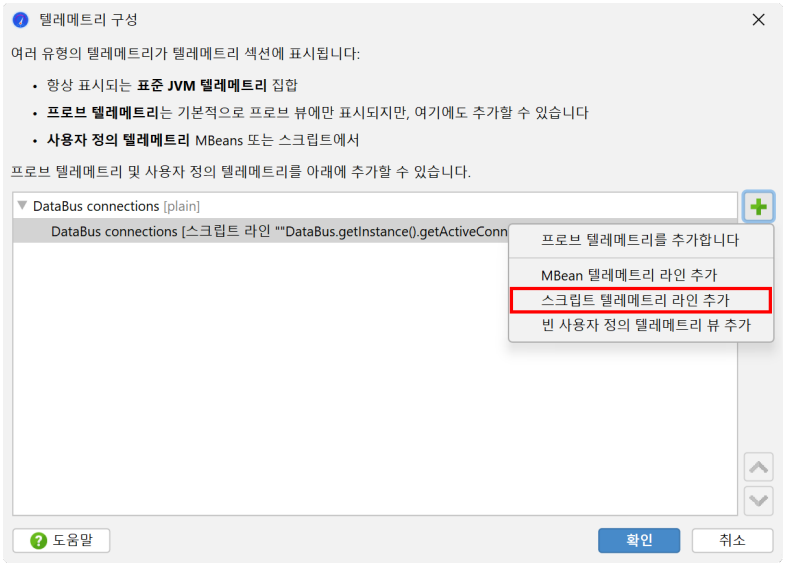
여러 프로브는 중첩된 분할을 생성할 수 있습니다. 단일 프로브는 기본적으로 하나의 분할 수준만 생성하며, 스크립트 프로브에서는 지원되지 않는 **재진입 가능**으로 구성되지 않는 한 그렇습니다.

JProfiler UI에서 호출 트리 분할은 스크립트 프로브 기능과 번들로 제공되지 않으며, "메소드 분할"이라는 별도의 기능 [p. 175]입니다. 이들은 페이로드를 생성하지 않고 호출 트리만 분할하므로 이름과 설명이 있는 프로브 뷰가 필요하지 않습니다. 인젝티드 프로브는 `SplitInterception`으로 주석이 달린 인터셉션 핸들러 메소드에서 분할 문자열을 반환하고, 임베디드 프로브는 분할 문자열로 `Split.enter`를 호출합니다.



텔레메트리

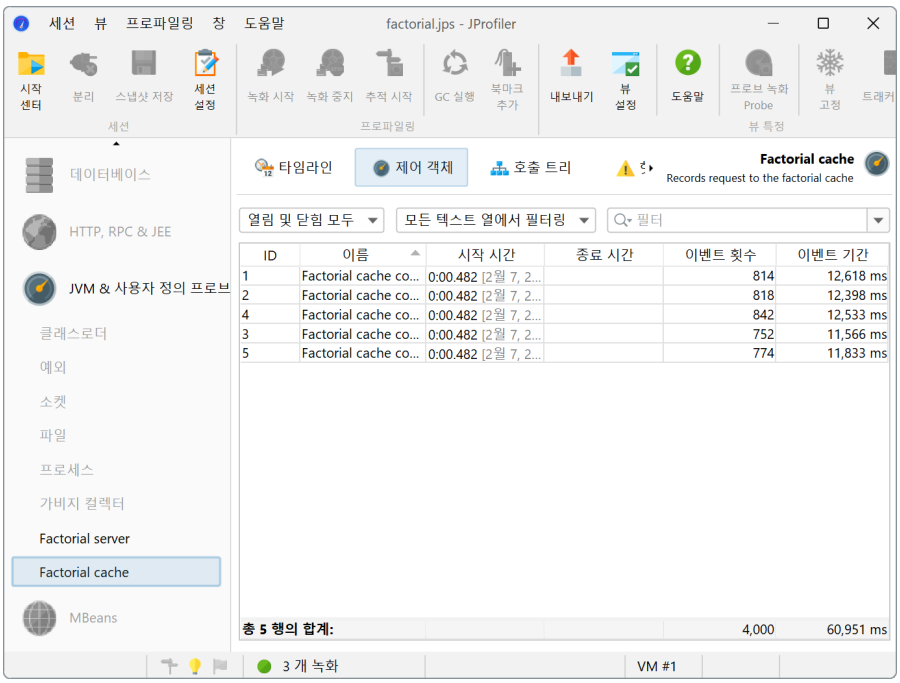
사용자 정의 프로브에는 두 가지 기본 텔레메트리가 있습니다: 이벤트 빈도와 평균 이벤트 지속 시간. 인젝티드 및 임베디드 프로브는 프로브 구성 클래스의 주석이 달린 메소드를 통해 추가 텔레메트리를 지원합니다. JProfiler UI에서 스크립트 텔레메트리는 스크립트 프로브 기능과 독립적이며, 도구 모음의 텔레메트리 구성 버튼 아래 "텔레메트리" 섹션에서 찾을 수 있습니다.



텔레메트리 메소드는 초당 한 번씩 폴링됩니다. `Telemetry` 주석에서 단위와 스케일 팩터를 구성할 수 있습니다. `line` 속성을 사용하여 여러 텔레메트리를 단일 텔레메트리 뷰로 결합할 수 있습니다. `TelemetryFormat`의 `stacked` 속성을 사용하여 선을 추가하고 스택형 선 그래프로 표시할 수 있습니다. 임베디드 및 인젝티드 프로브의 텔레메트리 관련 API는 동등하지만 각각의 프로브 유형에만 적용 가능합니다.

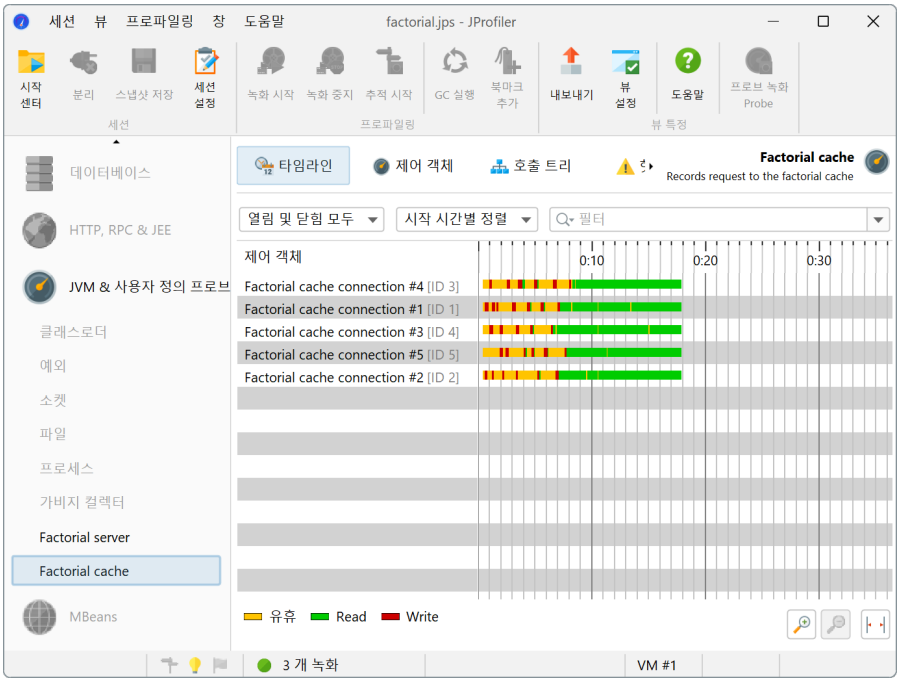
제어 객체

때때로 프로브 이벤트를 JProfiler에서 "제어 객체"라고 하는 관련된 장기 객체에 연결하는 것이 흥미로울 수 있습니다. 예를 들어, 데이터베이스 프로브에서는 쿼리를 실행하는 물리적 연결이 그 역할을 합니다. 이러한 제어 객체는 임베디드 API와 인젝티드 프로브 API로 열고 닫을 수 있으며, 프로브 이벤트 뷰에서 해당 이벤트를 생성합니다. 프로브 이벤트가 생성될 때 제어 객체를 지정할 수 있으며, 프로브의 "제어 객체" 뷰에 표시되는 통계에 기여합니다.



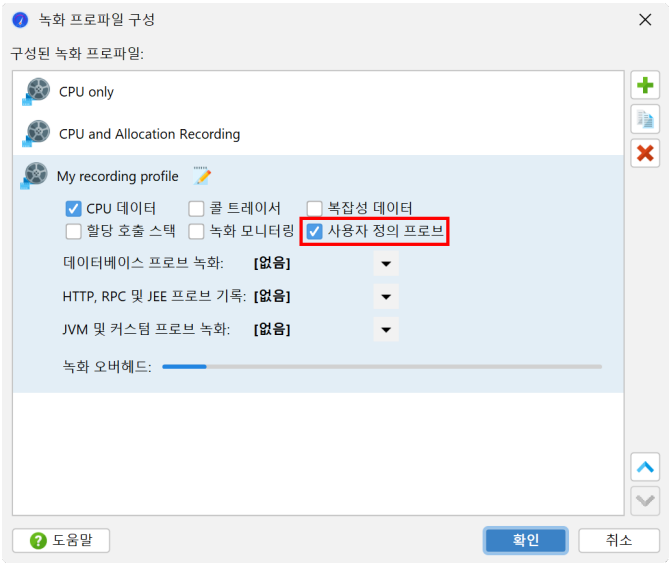
제어 객체는 열릴 때 지정해야 하는 표시 이름을 가집니다. 새 제어 객체가 프로브 이벤트 생성 시 사용되면, 프로브는 구성에서 이름 해석기를 제공해야 합니다.

또한, 프로브는 열거형 클래스를 통해 사용자 정의 이벤트 유형을 정의할 수 있습니다. 프로브 이벤트가 생성될 때 이러한 유형 중 하나를 지정할 수 있으며, 이벤트 뷰에 나타나 단일 이벤트 유형을 필터링할 수 있습니다. 더 중요한 것은 시간 축에 선으로 표시된 제어 객체를 보여주는 프로브의 타임라인 뷰가 이벤트 유형에 따라 색상이 지정된다는 것입니다. 사용자 정의 이벤트 유형이 없는 프로브의 경우, 색상은 이벤트가 기록되지 않는 유휴 상태와 프로브 이벤트 지속 시간 동안의 기본 이벤트 상태를 보여줍니다. 사용자 정의 유형을 사용하면 "읽기" 및 "쓰기"와 같은 상태를 구별할 수 있습니다.

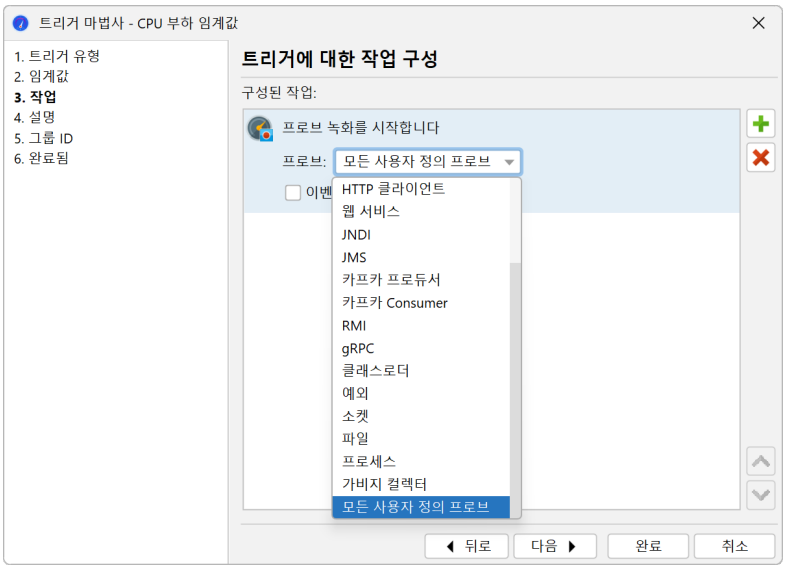


녹화

모든 프로브와 마찬가지로 사용자 정의 프로브는 기본적으로 데이터를 기록하지 않으며, 필요에 따라 녹화를 활성화하고 비활성화해야 합니다. 프로브 뷰에서 수동 시작/중지 작업을 사용할 수 있지만, 시작 시 프로브 녹화를 켜야 하는 경우가 많습니다. JProfiler는 사용자 정의 프로브를 미리 알지 못하기 때문에, 녹화 프로파일에는 모든 사용자 정의 프로브에 적용되는 사용자 정의 프로브 체크 박스가 있습니다.



마찬가지로, 프로브 녹화를 시작하고 중지하는 트리거 작업에 대해 모든 사용자 정의 프로브를 선택할 수 있습니다.



프로그래밍 방식의 녹화를 위해 `Controller.startProbeRecording(Controller.PROBE_NAME_ALL_CUSTOM, ProbeRecordingOptions.EVENTS)` 를 호출하여 모든 사용자 정의 프로브를 기록하거나, 보다 구체적으로 하기 위해 프로브의 클래스 이름을 전달할 수 있습니다.

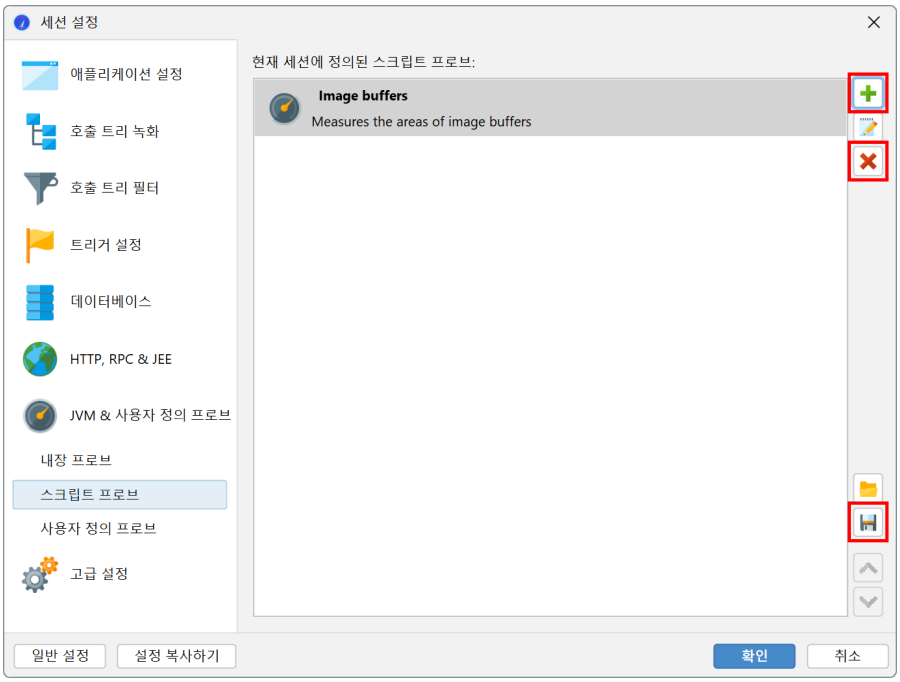
A.2 스크립트 프로브

IDE에서 사용자 정의 프로브를 개발하려면 인터셉션 포인트와 프로브가 제공할 이점에 대한 명확한 이해가 필요합니다. 반면에 스크립트 프로브를 사용하면 JProfiler GUI에서 간단한 프로브를 직접 정의하고 API를 배우지 않고도 실험할 수 있습니다. 임베디드 또는 인젝션된 사용자 정의 프로브와 달리, 스크립트 프로브는 실행 중인 프로파일링 세션 동안 재정의할 수 있어 빠른 편집-컴파일-테스트 루프를 제공합니다.

스크립트 프로브 정의하기

스크립트 프로브는 인터셉트된 메서드를 선택하고 프로브의 페이로드 문자열을 반환하는 스크립트를 입력하여 정의됩니다. 여러 개의 메서드-스크립트 쌍을 단일 프로브로 묶을 수 있습니다.

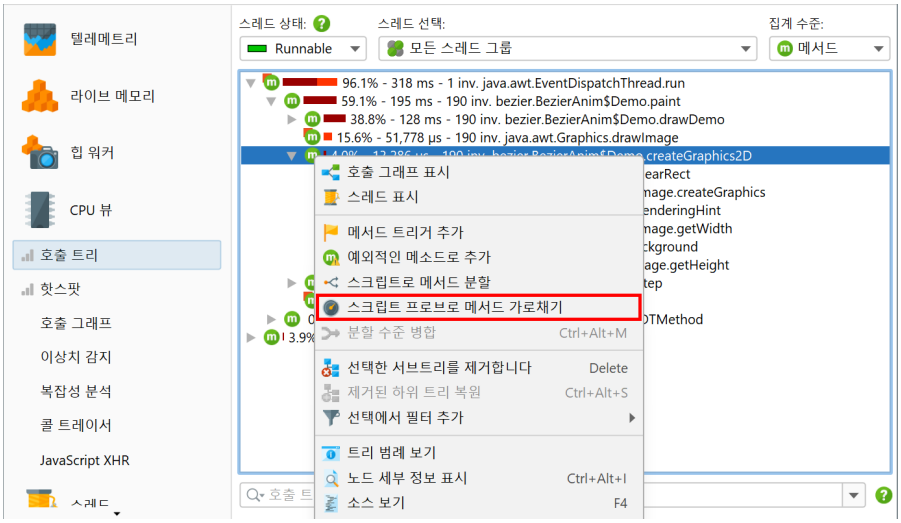
스크립트 프로브 설정은 세션 설정에서 액세스할 수 있습니다. 여기에서 스크립트 프로브를 생성하고 삭제할 수 있으며, 다른 프로파일링 세션에서 가져올 수 있는 세트로 스크립트 프로브를 저장할 수 있습니다.



각 스크립트 프로브에는 이름과 선택적으로 설명이 필요합니다. 이름은 JProfiler의 뷰 선택기에서 "JEE & Probes" 섹션에 프로브 뷰를 추가하는 데 사용됩니다. 설명은 프로브 뷰의 헤더에 표시되며 그 목적에 대한 간단한 설명이어야 합니다.

메서드를 선택할 때는 구성된 클래스패스에서 클래스를 선택하거나 프로파일된 클래스에서 클래스를 선택하는 등 여러 옵션이 있습니다. 두 번째 단계에서는 선택한 클래스에서 메서드를 선택할 수 있습니다.

인터셉트된 메서드를 선택하는 훨씬 쉬운 방법은 호출 트리 뷰에서 선택하는 것입니다. 컨텍스트 메뉴에서 스크립트 프로브로 메서드 인터셉트 작업을 통해 새 프로브를 생성할지 기존 프로브에 인터셉션을 추가할지 묻습니다.

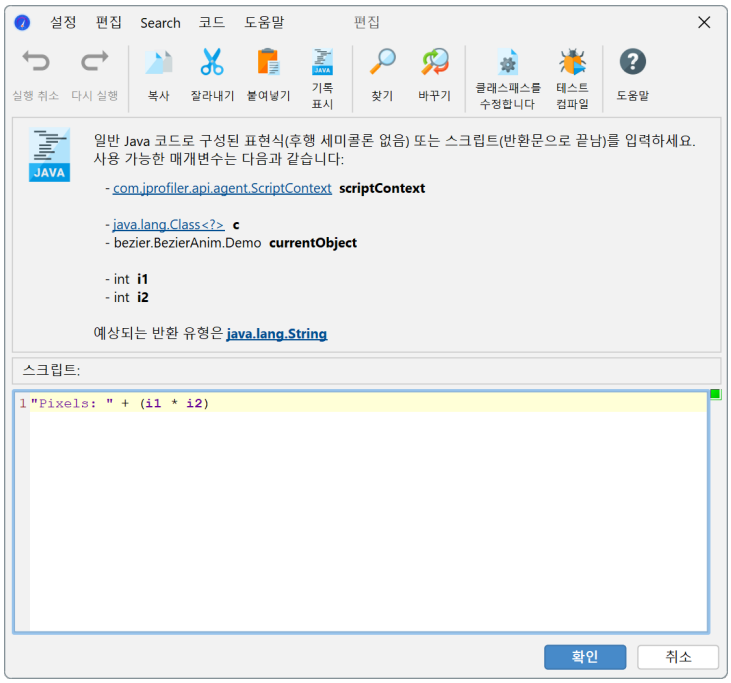


프로브 스크립트

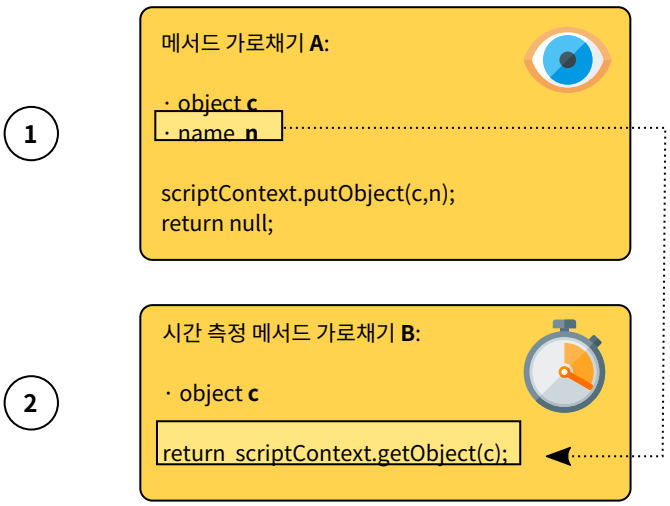
스크립트 편집기에서는 인터셉트된 메서드의 모든 매개변수와 메서드가 호출된 객체에 액세스할 수 있습니다. 인터셉트된 메서드의 반환 값이나 발생한 예외에 액세스해야 하는 경우 임베디드 또는 인젝션된 프로브를 작성해야 합니다.

이 환경에서는 스크립트가 표현식 또는 반환문이 있는 문장 시퀀스로 페이로드 문자열을 구성할 수 있습니다. 이러한 스크립트의 가장 간단한 버전은 하나의 매개변수에 대해 `parameter.toString()`을 반환하거나 기본 유형의 매개변수에 대해 `String.valueOf(parameter)`을 반환합니다. `null`을 반환하면 페이로드가 생성되지 않습니다.

CPU와 프로브 데이터를 동시에 기록하면 CPU 섹션의 호출 트리 뷰에서 적절한 호출 스택에 있는 프로브 뷰로의 링크를 표시합니다. 또는 CPU 호출 트리 뷰에 페이로드 문자열을 인라인으로 표시하도록 선택할 수 있습니다. 프로브 마법사의 "페이로드 인터셉션->호출 트리 주석" 단계에는 이 옵션이 포함되어 있습니다.



스크립트에 사용할 수 있는 또 다른 매개변수는 스크립트 컨텍스트로, 현재 프로브에 대해 정의된 모든 스크립트의 호출 간에 데이터를 저장할 수 있는 `com.jprofiler.api.agent.ScriptContext` 유형의 객체입니다. 예를 들어, 인터셉트된 메서드 A는 텍스트 표현이 좋지 않은 객체만 볼 수 있지만, 객체와 표시 이름 간의 연관성은 메서드 B를 인터셉트하여 얻을 수 있다고 가정해 보겠습니다. 그런 다음 동일한 프로브에서 메서드 B를 인터셉트하고 객체-텍스트 연관성을 스크립트 컨텍스트에 직접 저장할 수 있습니다. 메서드 A에서는 스크립트 컨텍스트에서 해당 표시 텍스트를 가져와 페이로드 문자열을 구성하는 데 사용할 수 있습니다.



이러한 종류의 문제가 너무 복잡해지면 임베디드 또는 인젝션된 프로브 API로 전환하는 것을 고려해야 합니다.

누락된 기능

스크립트 프로브는 사용자 정의 프로브 개발에 쉽게 진입할 수 있도록 설계되었지만, 전체 프로브 시스템에서 몇 가지 기능이 누락되어 있다는 점을 인식해야 합니다:

- 스크립트 프로브는 호출 트리 분할을 수행할 수 없습니다. JProfiler UI에서는 사용자 정의 프로브 개념 [p. 143]에서 설명한 대로 별도의 기능입니다. 임베디드 및 인젝션된 프로브는 호출 트리 분할 기능을 직접 제공합니다.
- 스크립트 프로브는 제어 객체를 생성하거나 사용자 정의 프로브 이벤트 유형을 생성할 수 없습니다. 이는 임베디드 또는 인젝션된 프로브에서만 가능합니다.
- 스크립트 프로브는 반환 값이나 발생한 예외에 액세스할 수 없습니다. 이는 임베디드 및 인젝션된 프로브와 다릅니다.
- 스크립트 프로브는 재진입 인터셉션을 처리할 수 없습니다. 메서드가 재귀적으로 호출되면 첫 번째 호출만 인터셉트됩니다. 임베디드 및 인젝션된 프로브는 재진입 동작에 대한 세밀한 제어를 제공합니다.
- 기본 텔레메트리 외의 텔레메트리를 프로브 뷰에 번들로 묶을 수 없습니다. 대신, 사용자 정의 프로브 개념 [p. 143]에서 설명한 대로 스크립트 텔레메트리 기능을 사용할 수 있습니다.

A.3 Injected Probes

스크립트 프로브와 유사하게, 인젝션된 프로브는 선택된 메서드에 대한 인터셉션 핸들러를 정의합니다. 그러나, 인젝션된 프로브는 JProfiler GUI 외부의 IDE에서 개발되며, JProfiler에서 제공하는 인젝션된 프로브 API에 의존합니다. 이 API는 Apache License, version 2.0 하에 허가되어 있어 관련 아티팩트를 쉽게 배포할 수 있습니다.

인젝션된 프로브를 시작하는 가장 좋은 방법은 JProfiler 설치의 `api/samples/simple-injected-probe` 디렉토리에 있는 예제를 공부하는 것입니다. 해당 디렉토리에서 `../gradlew`를 실행하여 컴파일하고 실행하세요. Gradle 빌드 파일 `build.gradle`에는 샘플에 대한 추가 정보가 포함되어 있습니다. `api/samples/advanced-injected-probe`에 있는 예제는 제어 객체를 포함한 프로브 시스템의 더 많은 기능을 보여줍니다.

Development environment

인젝션된 프로브를 개발하기 위해 필요한 프로브 API는 다음과 같은 Maven 좌표를 가진 단일 아티팩트에 포함되어 있습니다.

```
group: com.jprofiler
artifact: jprofiler-probe-injected
version: <JProfiler version>
```

이 매뉴얼에 해당하는 JProfiler 버전은 15.0입니다.

Jar, 소스 및 javadoc 아티팩트는 다음 저장소에 게시됩니다.

```
https://maven.ej-technologies.com/repository
```

Gradle 또는 Maven과 같은 빌드 도구를 사용하여 개발 클래스 경로에 프로브 API를 추가하거나 JAR 파일을 사용할 수 있습니다.

```
api/jprofiler-probe-injected.jar
```

JProfiler 설치에서.

Javadoc을 보려면 다음으로 이동하세요.

```
api/javadoc/index.html
```

해당 javadoc은 JProfiler에서 게시한 모든 API의 javadoc을 결합한 것입니다. `com.jprofiler.api.probe.injected` 패키지의 개요는 API를 탐색하기 위한 좋은 출발점입니다.

Probe structure

인젝션된 프로브는 `com.jprofiler.api.probe.injected.Probe`로 주석이 달린 클래스입니다. 해당 주석의 속성은 전체 프로브에 대한 구성 옵션을 노출합니다. 예를 들어, 개별 검사가 필요 없는 많은 프로브 이벤트를 생성하는 경우, `events` 속성을 사용하여 프로브 이벤트 뷰를 비활성화하고 오버헤드를 줄일 수 있습니다.

```
@Probe(name = "Foo", description = "Shows foo server requests", events = "false")
public class FooProbe {
    ...
}
```

프로브 클래스에는 인터셉션 핸들러를 정의하기 위해 특별히 주석이 달린 정적 메서드를 추가합니다. PayloadInterception 주석은 페이로드를 생성하고 SplitInterception 주석은 호출 트리를 분할합니다. 핸들러의 반환 값은 주석에 따라 페이로드 또는 분할 문자열로 사용됩니다. 스크립트 프로브와 마찬가지로 null을 반환하면 인터셉션에 아무런 효과가 없습니다. 타이밍 정보는 인터셉션된 메서드에 대해 자동으로 계산됩니다.

```
@Probe(name = "FooBar")
public class FooProbe {
    @PayloadInterception(
        invokeOn = InvocationType.ENTER,
        method = @MethodSpec(className = "com.bar.Database",
                             methodName = "processQuery",
                             parameterTypes = {"com.bar.Query"},
                             returnType = "void"))
    public static String fooRequest(@Parameter(0) Query query) {
        return query.getVerbose();
    }

    @SplitInterception(
        method = @MethodSpec(className = "com.foo.Server",
                             methodName = "handleRequest",
                             parameterTypes = {"com.foo.Request"},
                             returnType = "void"))
    public static String barQuery(@Parameter(0) Request request) {
        return request.getPath();
    }
}
```

위의 예에서 볼 수 있듯이, 두 주석 모두 MethodSpec을 사용하여 인터셉션된 메서드를 정의하는 method 속성을 가지고 있습니다. 스크립트 프로브와 달리, MethodSpec은 클래스 이름이 비어 있을 수 있으며, 특정 서명을 가진 모든 메서드가 클래스 이름에 상관없이 인터셉션됩니다. 또는 MethodSpec의 subtypes 속성을 사용하여 전체 클래스 계층 구조를 인터셉션할 수 있습니다.

스크립트 프로브와 달리 모든 매개변수가 자동으로 사용 가능한 경우, 핸들러 메서드는 관심 있는 값을 요청하기 위해 매개변수를 선언합니다. 각 매개변수는 com.jprofiler.api.probe.injected.parameter 패키지의 주석으로 주석이 달려 있어 프로파일링 에이전트가 메서드에 전달해야 할 객체 또는 기본 값을 알 수 있습니다. 예를 들어, 핸들러 메서드의 매개변수에 @Parameter(0)를 주석으로 달면 인터셉션된 메서드의 첫 번째 매개변수가 주입됩니다.

인터셉션된 메서드의 메서드 매개변수는 모든 인터셉션 유형에 대해 사용 가능합니다. 페이로드 인터셉션은 @ReturnValue 또는 @ExceptionValue를 사용하여 반환 값을 액세스할 수 있으며, 메서드의 출구를 인터셉션하도록 프로파일링 에이전트에 지시한 경우 예외를 던질 수 있습니다. 이는 PayloadInterception 주석의 invokeOn 속성을 사용하여 수행됩니다.

스크립트 프로브와 달리, 인젝션된 프로브 핸들러는 인터셉션된 메서드의 재귀 호출에 대해 호출될 수 있으며, 인터셉션 주석의 reentrant 속성을 true로 설정한 경우에만 가능합니다. 핸들러 메서드에 ProbeContext 유형의 매개변수를 사용하여 ProbeContext.getOuterPayload() 또는 ProbeContext.restartTiming()을 호출하여 중첩 호출에 대한 프로브의 동작을 제어할 수 있습니다.

Advanced interceptions

때로는 단일 인터셉션으로 프로브 문자열을 작성하는 데 필요한 모든 정보를 수집하기에 충분하지 않을 수 있습니다. 이를 위해, 프로브는 페이로드나 분할을 생성하지 않는 Interception으로 주석이 달린 임의의 수의 인터셉션 핸들러를 포함할 수 있습니다. 정보는 프로브 클래스의 정적 필드에 저장할 수 있습니다. 멀티스레드 환경에서 스레드 안전성을 위해, 참조 유형을 저장하기 위해 ThreadLocal 인스턴스를 사용하고, 카운터를 유지하기 위해 java.util.concurrent.atomic 패키지의 원자 숫자 유형을 사용해야 합니다.

특정 상황에서는 메서드 진입과 메서드 종료 모두에 대한 인터셉션이 필요할 수 있습니다. 일반적인 경우는 `inMethodCall`과 같은 상태 변수를 유지하여 다른 인터셉션의 동작을 수정하는 경우입니다. 기본 인터셉션 유형인 진입 인터셉션에서 `inMethodCall`을 `true`로 설정할 수 있습니다. 이제 해당 인터셉션 바로 아래에 정적 메서드를 정의하고 `@AdditionalInterception(invokeOn = InvocationType.EXIT)`으로 주석을 달아야 합니다. 인터셉션 핸들러 위에서 가져온 인터셉션된 메서드이므로 다시 지정할 필요가 없습니다. 메서드 본문에서 `inMethodCall` 변수를 `false`로 설정할 수 있습니다.

```
...

private static final ThreadLocal<Boolean> inMethodCall =
    ThreadLocal.withInitial(() -> Boolean.FALSE);

@Interception(
    invokeOn = InvocationType.ENTER,
    method = @MethodSpec(className = "com.foo.Server",
                         methodName = "internalCall",
                         parameterTypes = {"com.foo.Request"},
                         returnType = "void"))
public static void guardEnter() {
    inMethodCall.set(Boolean.TRUE);
}

@AdditionalInterception(InvocationType.EXIT)
public static void guardExit() {
    inMethodCall.set(Boolean.FALSE);
}

@SplitInterception(
      method = @MethodSpec(className = "com.foo.Server",
                           methodName = "handleRequest",
                           parameterTypes = {"com.foo.Request"},
                           returnType = "void"),
      reentrant = true)
public static String splitRequest(@Parameter(0) Request request) {
    if (!inMethodCall.get()) {
        return request.getPath();
    } else {
        return null;
    }
}

...
```

이 사용 사례의 작동 예제는 `api/samples/advanced-injected-probe/src/main/java/AdvancedAwtEventProbe.java`에서 볼 수 있습니다.

Control objects

`Probe` 주석의 `controlObjects` 속성이 `true`로 설정되지 않으면 제어 객체 뷰는 보이지 않습니다. 제어 객체를 사용하려면 핸들러 메서드에 해당 유형의 매개변수를 선언하여 `ProbeContext`를 얻어야 합니다. 아래 샘플 코드는 제어 객체를 열고 프로브 이벤트와 연결하는 방법을 보여줍니다.

```
@Probe(name = "Foo", controlObjects = true, customTypes = MyEventTypes.class)
public class FooProbe {
    @Interception(
            invokeOn = InvocationType.EXIT,
            method = @MethodSpec(className = "com.foo.ConnectionPool",
                        methodName = "createConnection",
                        parameterTypes = {},
                        returnType = "com.foo.Connection"))
    public static void openConnection(ProbeContext pc, @ReturnValue Connection c) {
        pc.openControlObject(c, c.getId());
    }

    @PayloadInterception(
            invokeOn = InvocationType.EXIT,
            method = @MethodSpec(className = "com.foo.ConnectionPool",
                        methodName = "createConnection",
                        parameterTypes = {"com.foo.Query", "com.foo.Connection"},
                        returnType = "com.foo.Connection"))
    public static Payload handleQuery(
        ProbeContext pc, @Parameter(0) Query query, @Parameter(1) Connection c) {
        return pc.createPayload(query.getVerbose(), c, MyEventTypes.QUERY);
    }

    ...

}
```

제어 객체는 정의된 수명을 가지며, 프로브 뷰는 타임라인과 제어 객체 뷰에서 열고 닫는 시간을 기록합니다. 가능하면 `ProbeContext.openControlObject()`와 `ProbeContext.closeControlObject()`를 호출하여 제어 객체를 명시적으로 열고 닫아야 합니다. 그렇지 않으면 제어 객체의 표시 이름을 해석하는 `@ControlObjectName`으로 주석이 달린 정적 메서드를 선언해야 합니다.

프로브 이벤트는 핸들러 메서드가 `String` 대신 `Payload` 인스턴스를 반환하는 경우 제어 객체와 연결될 수 있습니다. `ProbeContext.createPayload()` 메서드는 제어 객체와 프로브 이벤트 유형을 받습니다. 허용된 이벤트 유형을 가진 열거형은 `Probe` 주석의 `customTypes` 속성에 등록되어야 합니다.

제어 객체는 시간 측정의 시작 시점에 지정되어야 하며, 이는 메서드 진입에 해당합니다. 경우에 따라 페이로드 문자열의 이름은 반환 값이나 다른 인터셉션에 따라 달라지기 때문에 메서드 종료 시에만 사용할 수 있습니다. 이 경우 `ProbeContext.createPayloadWithDeferredName()`을 사용하여 이름 없이 페이로드 객체를 생성할 수 있습니다. `@AdditionalInterception(invokeOn = InvocationType.EXIT)`으로 주석이 달린 인터셉션 핸들러를 바로 아래에 정의하고 해당 메서드에서 `String`을 반환하면 자동으로 페이로드 문자열로 사용됩니다.

Overriding the thread state

데이터베이스 드라이버 또는 외부 리소스에 대한 네이티브 커넥터의 실행 시간을 측정할 때, JProfiler에 일부 메서드를 다른 스레드 상태로 설정하도록 지시해야 할 때가 있습니다. 예를 들어, 데이터베이스 호출을 "Net I/O" 스레드 상태에 두는 것이 유용합니다. 통신 메커니즘이 표준 Java I/O 기능을 사용하지 않고 네이티브 메커니즘을 사용하는 경우, 이는 자동으로 적용되지 않습니다.

`ThreadState.NETIO.enter()`와 `ThreadState.exit()` 호출 쌍을 사용하여 프로파일링 에이전트는 스레드 상태를 적절히 조정합니다.

```
...

@Interception(invokeOn = InvocationType.ENTER, method = ...)
public static void enterMethod(ProbeContext probeContext, @ThisValue JComponent
component) {
    ThreadState.NETIO.enter();
}

@AdditionalInterception(InvocationType.EXIT)
public static void exitMethod() {
    ThreadState.exit();
}

...
```

Deployment

인젝션된 프로브를 배포하는 방법에는 클래스패스에 넣을지 여부에 따라 두 가지 방법이 있습니다. VM 매개변수

```
-Djprofiler.probeClassPath=...
```

별도의 프로브 클래스 경로가 프로파일링 에이전트에 의해 설정됩니다. 프로브 클래스패스는 디렉토리와 클래스 파일을 포함할 수 있으며, Windows에서는 ';'로, 다른 플랫폼에서는 ':'로 구분됩니다. 프로파일링 에이전트는 프로브 클래스패스를 스캔하여 모든 프로브 정의를 찾습니다.

프로브 클래스를 클래스패스에 배치하는 것이 더 쉬운 경우, VM 매개변수를 설정할 수 있습니다.

```
-Djprofiler.customProbes=...
```

쉼표로 구분된 완전한 클래스 이름 목록으로. 이러한 클래스 이름 각각에 대해, 프로파일링 에이전트는 인젝션된 프로브를 로드하려고 시도합니다.

A.4 임베디드 프로브

프로브의 대상이 되는 소프트웨어 컴포넌트의 소스 코드를 제어할 수 있는 경우, 주입된 프로브 대신 임베디드 프로브를 작성해야 합니다.

주입된 프로브를 작성할 때 초기 노력의 대부분은 인터셉트된 메소드를 지정하고 핸들러 메소드의 매개변수로 주입된 객체를 선택하는 데 들어갑니다. 임베디드 프로브를 사용하면 모니터링된 메소드에서 임베디드 프로브 API를 직접 호출할 수 있기 때문에 이러한 작업이 필요하지 않습니다. 임베디드 프로브의 또 다른 장점은 배포가 자동이라는 것입니다. 프로브는 소프트웨어와 함께 제공되며 애플리케이션이 프로파일될 때 JProfiler UI에 나타납니다. 제공해야 하는 유일한 종속성은 주로 프로파일링 에이전트의 훅으로 작동하는 빈 메소드 본문으로 구성된 Apache 2.0 라이선스 하의 작은 JAR 파일입니다.

개발 환경

개발 환경은 주입된 프로브와 동일하지만, 아티팩트 이름이 `jprofiler-probe-embedded`이고 JAR 파일을 별도의 프로젝트에서 프로브를 개발하는 대신 애플리케이션과 함께 제공한다는 점이 다릅니다. 소프트웨어 컴포넌트에 임베디드 프로브를 추가하는 데 필요한 프로브 API는 단일 JAR 아티팩트에 포함되어 있습니다. javadoc에서 API를 탐색할 때 `com.jprofiler.api.probe.embedded` 패키지 개요부터 시작하십시오.

주입된 프로브와 마찬가지로 임베디드 프로브에도 두 가지 예제가 있습니다. `api/samples/simple-embedded-probe`에는 임베디드 프로브 작성을 시작하는 데 도움이 되는 예제가 있습니다. 해당 디렉토리에서 `../gradlew`를 실행하여 컴파일하고 실행하며, 실행 환경을 이해하기 위해 gradle 빌드 파일 `build.gradle`를 연구하십시오. 제어 객체를 포함한 더 많은 기능을 보려면 `api/samples/advanced-embedded-probe`의 예제로 이동하십시오.

페이로드 프로브

주입된 프로브와 유사하게, 여전히 구성 목적으로 프로브 클래스가 필요합니다. 프로브 클래스는 페이로드를 수집하는지 호출 트리를 분할하는지에 따라 `com.jprofiler.api.probe.embedded.PayloadProbe` 또는 `com.jprofiler.api.probe.embedded.SplitProbe`를 확장해야 합니다. 주입된 프로브 API에서는 핸들러 메소드에 대한 페이로드 수집 및 분할에 대해 다른 주석을 사용합니다. 반면에 임베디드 프로브 API에는 핸들러 메소드가 없으며 이 구성을 프로브 클래스 자체로 이동해야 합니다.

```
public class FooPayloadProbe extends PayloadProbe {
    @Override
    public String getName() {
        return "Foo queries";
    }

    @Override
    public String getDescription() {
        return "Records foo queries";
    }
}
```

주입된 프로브가 구성을 위해 주석을 사용하는 반면, 임베디드 프로브는 프로브의 기본 클래스에서 메소드를 재정의하여 구성합니다. 페이로드 프로브의 경우 유일한 추상 메소드는 `getName()`이며, 다른 모든 메소드는 필요에 따라 재정의할 수 있는 기본 구현을 가지고 있습니다. 예를 들어, 오버헤드를 줄이기 위해 이벤트 뷰를 비활성화하려면 `isEvents()`를 재정의하여 `false`를 반환할 수 있습니다.

페이로드를 수집하고 관련된 시간을 측정하기 위해 `Payload.enter()`와 `Payload.exit()` 호출 쌍을 사용합니다.

```
public void measuredCall(String query) {
    Payload.enter(FooPayloadProbe.class);
    try {
        performWork();
    } finally {
        Payload.exit(query);
    }
}
```

`Payload.enter()` 호출은 프로브 클래스를 인수로 받으므로 프로파일링 에이전트가 호출의 대상 프로브를 알 수 있으며, `Payload.exit()` 호출은 자동으로 동일한 프로브와 연결되며 페이로드 문자열을 인수로 받습니다. 종료 호출을 놓치면 호출 트리가 깨질 수 있으므로 항상 try 블록의 finally 절에서 수행해야 합니다.

측정된 코드 블록이 값을 생성하지 않는 경우, 페이로드 문자열과 `Runnable`을 인수로 받는 `Payload.execute` 메소드를 대신 호출할 수 있습니다. Java 8+에서는 람다 또는 메소드 참조를 사용하여 이 메소드 호출을 매우 간결하게 만들 수 있습니다.

```
public void measuredCall(String query) {
  Payload.execute(FooPayloadProbe.class, query, this::performWork);
}
```

이 경우 페이로드 문자열은 사전에 알려져 있어야 합니다. `Callable`을 받는 `execute`의 버전도 있습니다.

```
public QueryResult measuredCall(String query) throws Exception {
    return Payload.execute(PayloadProbe.class, query, () -> query.execute());
}
```

`Callable`을 받는 시그니처의 문제는 `Callable.call()`이 체크된 `Exception`을 던지므로 이를 잡거나 포함하는 메소드에 선언해야 한다는 것입니다.

제어 객체

페이로드 프로브는 `Payload` 클래스의 적절한 메소드를 호출하여 제어 객체를 열고 닫을 수 있습니다. 제어 객체는 제어 객체와 사용자 정의 이벤트 유형을 받는 `Payload.enter()` 또는 `Payload.execute()` 메소드의 버전으로 프로브 이벤트와 연결됩니다.

```
public void measuredCall(String query, Connection connection) {
    Payload.enter(FooPayloadProbe.class, connection, MyEventTypes.QUERY);
    try {
        performWork();
    } finally {
        Payload.exit(query);
    }
}
```

제어 객체 뷰는 프로브 구성에서 명시적으로 활성화해야 하며, 사용자 정의 이벤트 유형은 프로브 클래스에서도 등록해야 합니다.

```
public class FooPayloadProbe extends PayloadProbe {
    @Override
    public String getName() {
        return "Foo queries";
    }

    @Override
    public String getDescription() {
        return "Records foo queries";
    }

    @Override
    public boolean isControlObjects() {
        return true;
    }

    @Override
    public Class<? extends Enum> getCustomTypes() {
        return Connection.class;
    }
}
```

제어 객체를 명시적으로 열고 닫지 않는 경우, 프로브 클래스는 모든 제어 객체에 대한 표시 이름을 해결하기 위해 `getControlObjectName`을 재정의해야 합니다.

분할 프로브

분할 프로브 기본 클래스에는 추상 메소드가 없으므로 프로브 뷰를 추가하지 않고도 호출 트리를 분할하는 데 사용할 수 있습니다. 이 경우 최소한의 프로브 정의는 다음과 같습니다.

```
public class FooSplitProbe extends SplitProbe {}
```

분할 프로브의 중요한 구성 중 하나는 재진입 가능 여부입니다. 기본적으로 최상위 호출만 분할됩니다. 재귀 호출도 분할하려면 `isReentrant()`를 재정의하여 `true`를 반환하도록 합니다. 분할 프로브는 또한 프로브 뷰를 생성하고 분할 문자열을 페이로드로 게시할 수 있으며, 프로브 클래스에서 `isPayloads()`를 재정의하여 `true`를 반환하도록 할 수 있습니다.

분할을 수행하려면 `Split.enter()`와 `Split.exit()` 호출 쌍을 만듭니다.

```
public void splitMethod(String parameter) {
    Split.enter(FooSplitProbe.class, parameter);
    try {
        performWork(parameter);
    } finally {
        Split.exit();
    }
}
```

페이로드 수집과 달리, 분할 문자열은 프로브 클래스와 함께 `Split.enter()` 메소드에 전달되어야 합니다. 다시 말하지만, `Split.exit()`가 신뢰할 수 있게 호출되는 것이 중요하므로, 이는 try 블록의 finally 절에 있어야 합니다. `Split`는 `Runnable` 및 `Callable` 인수를 사용하여 단일 호출로 분할을 수행하는 `execute()` 메소드도 제공합니다.

텔레메트리

임베디드 프로브에 대해 텔레메트리를 게시하는 것은 특히 편리합니다. 동일한 클래스패스에 있기 때문에 애플리케이션의 모든 정적 메소드에 직접 접근할 수 있습니다. 주입된 프로브와 마찬가지로, 프로브 구성 클래스의

정적 공용 메소드에 `@Telemetry`를 주석으로 달고 숫자 값을 반환합니다. 자세한 내용은 프로브 개념 [p. 143] 장을 참조하십시오. 임베디드 및 주입된 프로브 API의 `@Telemetry` 주석은 동일하며, 단지 다른 패키지에 있습니다.

임베디드 및 주입된 프로브 API 간의 또 다른 병렬 기능은 `ThreadState` 클래스를 사용하여 스레드 상태를 수정할 수 있는 기능입니다. 이 클래스는 두 API에 모두 존재하며 다른 패키지에 있습니다.

배포

JProfiler UI로 프로파일링할 때 임베디드 프로브를 활성화하기 위해 특별한 단계는 필요하지 않습니다. 그러나 `Payload` 또는 `Split`에 대한 첫 번째 호출이 이루어질 때만 프로브가 등록됩니다. 그 시점에만 JProfiler에서 관련된 프로브 뷰가 생성됩니다. 기본 제공 및 주입된 프로브의 경우처럼 처음부터 프로브 뷰가 보이기를 원한다면, 다음을 호출할 수 있습니다.

```
PayloadProbe.register(FooPayloadProbe.class);
```

페이로드 프로브의 경우

```
SplitProbe.register(FooSplitProbe.class);
```

분할 프로브의 경우.

오버헤드를 최소화하기 위해 명령줄 스위치로 제어되는 조건부로 `Payload` 및 `Split` 메소드를 호출할지 여부를 고려할 수 있습니다. 그러나 메소드 본문이 비어 있기 때문에 일반적으로 필요하지 않습니다. 프로파일링 에이전트가 attach되지 않은 경우, 페이로드 문자열의 생성 외에는 오버헤드가 발생하지 않습니다. 프로브 이벤트는 미세한 규모로 생성되지 않아야 하므로 비교적 드물게 생성되며, 페이로드 문자열을 생성하는 것은 비교적 사소한 작업이어야 합니다.

컨테이너의 또 다른 우려 사항은 클래스 경로에 외부 종속성을 노출하고 싶지 않을 수 있다는 것입니다. 컨테이너의 사용자가 임베디드 프로브 API를 사용할 수도 있으며, 이는 충돌을 초래할 수 있습니다. 이 경우, 임베디드 프로브 API를 자체 패키지로 셰이딩할 수 있습니다. JProfiler는 여전히 셰이딩된 패키지를 인식하고 API 클래스를 올바르게 계측합니다. 빌드 타임 셰이딩이 실용적이지 않은 경우, 소스 아카이브를 추출하여 클래스를 프로젝트의 일부로 만들 수 있습니다.

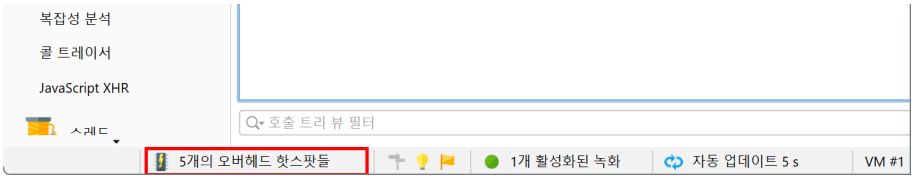
B 호출 트리 기능 상세

B.1 자동 튜닝 및 무시된 메소드

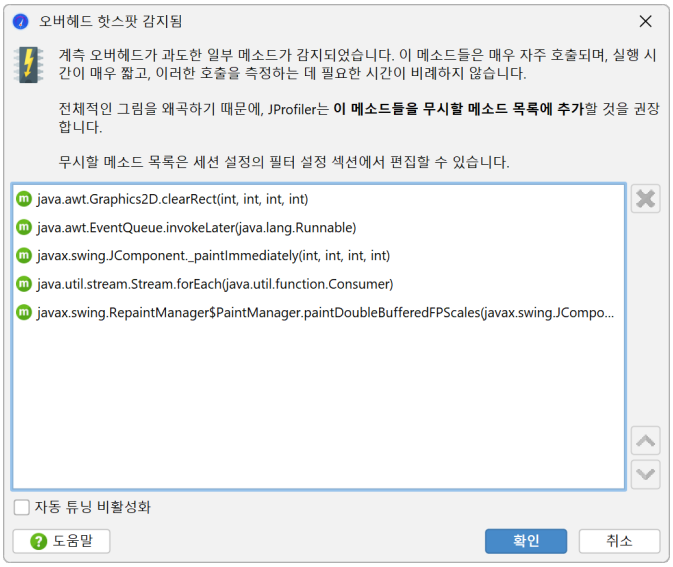
메서드 호출 녹화 유형이 계측으로 설정된 경우, 프로파일된 클래스의 모든 메서드가 계측됩니다. 이는 실행 시간이 매우 짧은 메서드에 대해 상당한 오버헤드를 발생시킵니다. 이러한 메서드가 매우 자주 호출되면, 해당 메서드의 측정된 시간이 너무 높게 나타날 것입니다. 또한, 계측으로 인해 핫스팟 컴파일러가 최적화를 방해할 수 있습니다. 극단적인 경우, 이러한 메서드는 계측되지 않은 실행에서는 사실이 아님에도 불구하고 지배적인 핫스팟이 됩니다. 예를 들어, XML 파서의 다음 문자를 읽는 메서드가 있습니다. 이러한 메서드는 매우 빠르게 반환되지만, 짧은 시간 내에 수백만 번 호출될 수 있습니다.

메서드 호출 녹화 유형이 샘플링으로 설정된 경우에는 이 문제가 발생하지 않습니다. 그러나 샘플링은 호출 횟수를 제공하지 않으며, 더 긴 메서드 호출만 표시하고, 샘플링이 사용될 때 여러 뷰가 완전한 기능을 갖지 못합니다.

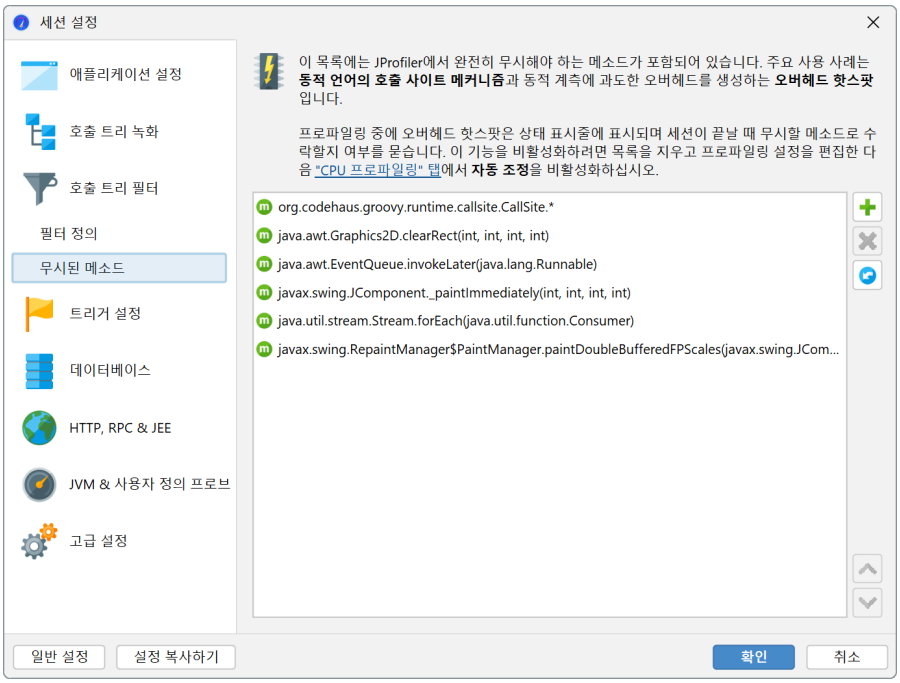
계측 문제를 완화하기 위해 JProfiler에는 자동 튜닝이라는 메커니즘이 있습니다. 프로파일링 에이전트는 주기적으로 높은 계측 오버헤드를 가진 메서드를 확인하고 이를 JProfiler GUI로 전송합니다. 상태 표시줄에는 오버헤드 핫스팟의 존재를 알리는 항목이 표시됩니다.



상태 표시줄 항목을 클릭하여 감지된 오버헤드 핫스팟을 검토하고 무시된 메소드 목록에 추가할 수 있습니다. 이러한 무시된 메소드는 계측되지 않습니다. 세션이 종료되면 동일한 대화 상자가 표시됩니다.

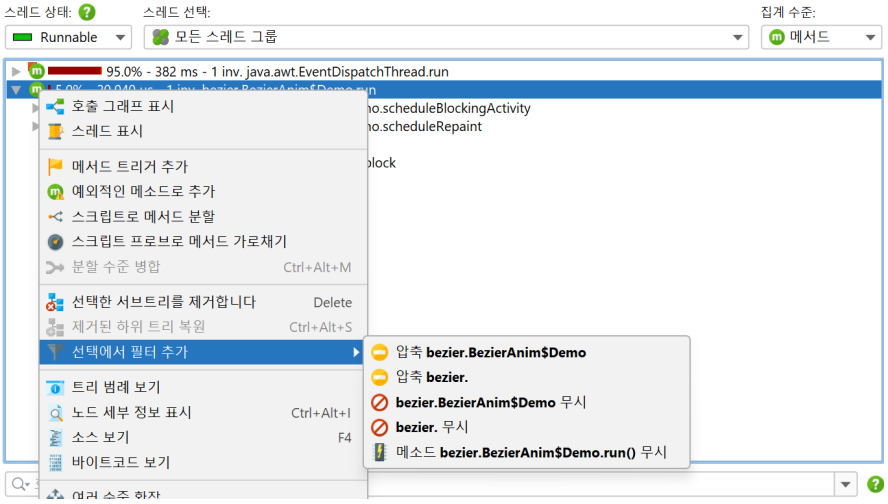


새로운 프로파일링 설정을 적용한 후에는 모든 무시된 메소드가 호출 트리에서 누락됩니다. 그들의 실행 시간은 호출 메소드의 자체 시간에 추가됩니다. 나중에 프로파일링 뷰에서 일부 무시된 메소드가 필수적임을 발견하면, 세션 설정의 무시된 메소드 탭에서 제거할 수 있습니다.



무시된 메소드의 기본 구성에는 실제 호출 체인을 따르기 어렵게 만드는 Groovy의 동적 메소드 디스패치에 사용되는 호출 사이트 클래스가 포함되어 있습니다.

무시된 메소드를 수동으로 추가하려면 세션 설정에서 할 수 있지만, 훨씬 더 쉬운 방법은 호출 트리에서 메소드를 선택하고 컨텍스트 메뉴에서 메소드 무시 작업을 호출하는 것입니다.



필터 설정에서 필터 항목의 유형을 "무시됨"으로 설정하여 전체 클래스나 패키지를 무시할 수도 있습니다. 선택에서 필터 추가 메뉴에는 선택된 노드에 따라 클래스나 패키지를 최상위 패키지까지 무시하도록 제안하는 작업이 포함되어 있습니다. 선택된 노드가 압축 프로파일된 것인지 프로파일된 것인지에 따라 필터를 반대 유형으로 변경하는 작업도 볼 수 있습니다.

자동 튜닝에 대한 메시지를 보고 싶지 않은 경우, 프로파일링 설정에서 이를 비활성화할 수 있습니다. 또한, 오버헤드 핫스팟을 결정하는 기준을 구성할 수 있습니다. 메소드는 다음 두 가지 조건이 모두 충족되면 오버헤드 핫스팟으로 간주됩니다:

- 모든 호출의 총 시간이 스레드의 전체 총 시간의 퍼밀레 기준 임계값을 초과합니다

- 평균 시간이 마이크로초 단위의 절대 임계값보다 낮습니다

B.2 비동기 및 원격 요청 추적

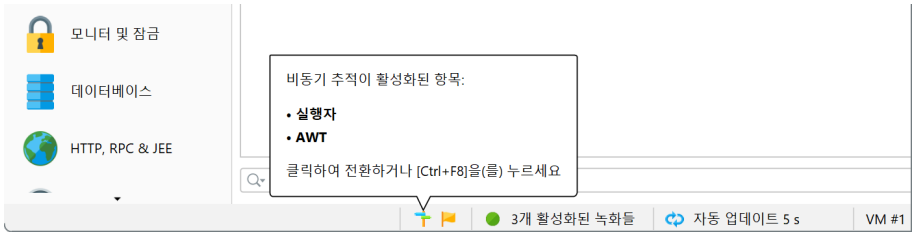
작업의 비동기 실행은 일반 Java 코드에서뿐만 아니라 반응형 프레임워크에서도 일반적인 관행입니다. 소스 파일에서 인접한 코드는 이제 두 개 이상의 다른 스레드에서 실행됩니다. 디버깅 및 프로파일링을 위해 이러한 스레드 변경은 두 가지 문제를 제시합니다. 한편으로는 호출된 작업이 얼마나 비용이 드는지 명확하지 않습니다. 다른 한편으로는 비용이 많이 드는 작업을 실행을 유발한 코드로 추적할 수 없습니다.

JProfiler는 호출이 동일한 JVM에 남아 있는지 여부에 따라 이 문제에 대한 다양한 솔루션을 제공합니다. 비동기 실행이 호출한 동일한 JVM에서 발생하는 경우 "Inline Async Executions" 호출 트리 분석 [p. 179]은 호출 사이트와 실행 사이트를 모두 포함하는 단일 호출 트리를 계산합니다. 원격 JVM에 요청이 있는 경우 호출 트리 [p. 51]에는 호출 사이트와 실행 사이트로의 하이퍼링크가 포함되어 있어 관련 JVM에 대한 프로파일링 세션을 표시하는 JProfiler 최상위 창 간에 원활하게 탐색할 수 있습니다.

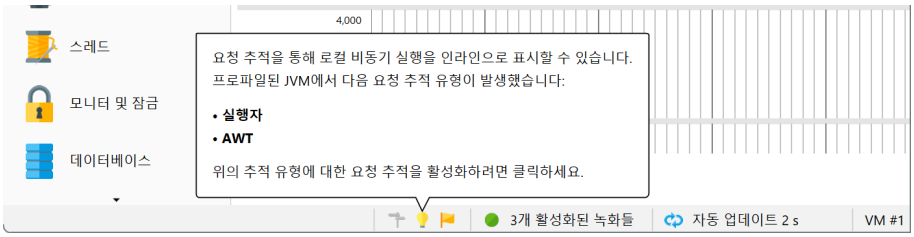
비동기 및 원격 요청 추적 활성화

비동기 메커니즘은 다양한 방식으로 구현될 수 있으며 별도의 스레드 또는 다른 JVM에서 작업을 시작하는 의미를 일반적인 방식으로 감지할 수 없습니다. JProfiler는 여러 일반적인 비동기 및 원격 요청 기술을 명시적으로 지원합니다. 요청 추적 설정에서 이를 활성화하거나 비활성화할 수 있습니다. 기본적으로 요청 추적은 활성화되어 있지 않습니다. 세션이 시작되기 직전에 표시되는 세션 시작 대화 상자에서 요청 추적을 구성할 수도 있습니다.

JProfiler의 메인 창에서 상태 표시줄은 일부 비동기 및 원격 요청 추적 유형이 활성화되어 있는지 여부를 표시하고 구성 대화 상자로의 바로 가기를 제공합니다.

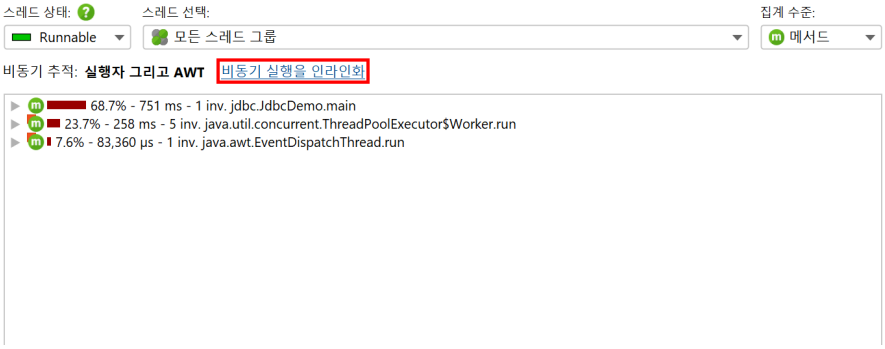


JProfiler는 프로파일된 JVM에서 활성화되지 않은 비동기 요청 추적 유형이 사용되는지 감지하고 상태 표시줄의 비동기 및 원격 요청 추적 아이콘 옆에 💡 알림 아이콘을 표시합니다. 알림 아이콘을 클릭하면 감지된 추적 유형을 활성화할 수 있습니다. 비동기 및 원격 요청 추적은 상당한 오버헤드를 발생시킬 수 있으며 필요한 경우에만 활성화해야 합니다.



비동기 추적

적어도 하나의 비동기 추적 유형이 활성화된 경우 CPU, 할당 및 프로브 녹화를 위한 호출 트리 및 핫스팟 뷰는 활성화된 모든 추적 유형에 대한 정보를 표시하고 "Inline Async Executions" 호출 트리 분석을 계산하는 버튼을 제공합니다. 해당 분석의 결과 뷰에서는 모든 비동기 실행의 호출 트리가 "비동기 실행" 노드를 통해 호출 사이트와 연결됩니다. 기본적으로 비동기 실행 측정값은 호출 트리의 상위 노드에 추가되지 않습니다. 때로는 집계된 값을 보는 것이 유용하기 때문에 분석 상단의 체크박스를 통해 적절한 경우 이를 수행할 수 있습니다.



다른 스레드에서 작업을 오프로드하는 가장 간단한 방법은 새 스레드를 시작하는 것입니다. JProfiler를 사용하면 "Thread start" 요청 추적 유형을 활성화하여 스레드의 생성부터 실행 사이트까지 스레드를 추적할 수 있습니다. 그러나 스레드는 무거운 객체이며 일반적으로 반복 호출을 위해 재사용되므로 이 요청 추적 유형은 디버깅 목적으로 더 유용합니다.

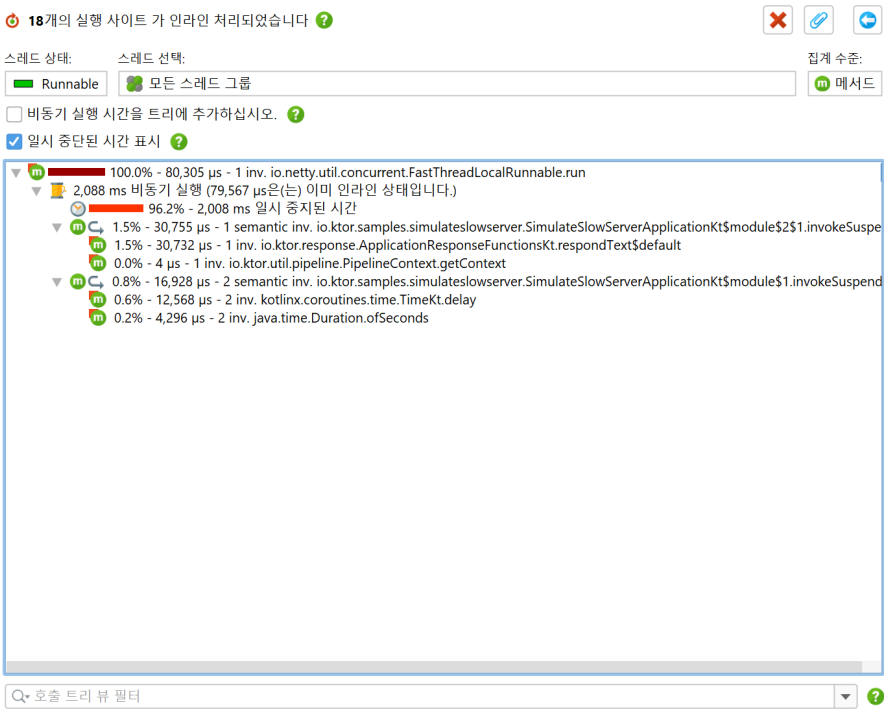
다른 스레드에서 작업을 시작하는 가장 중요하고 일반적인 방법은 `java.util.concurrent` 패키지의 실행기를 사용하는 것입니다. 실행기는 비동기 실행을 처리하는 많은 고급 타사 라이브러리의 기반이기도 합니다. 실행기를 지원함으로써 JProfiler는 멀티 스레드 및 병렬 프로그래밍을 처리하는 전체 클래스의 라이브러리를 지원합니다.

위의 일반적인 경우 외에도 JProfiler는 JVM용 두 가지 GUI 툴킷인 AWT와 SWT도 지원합니다. 두 툴킷은 단일 스레드로, GUI 위젯을 조작하고 그리기 작업을 수행할 수 있는 특별한 이벤트 디스패치 스레드가 있습니다. GUI를 차단하지 않으려면 백그라운드 스레드에서 장기 실행 작업을 수행해야 합니다. 그러나 백그라운드 스레드는 종종 진행 상황이나 완료를 표시하기 위해 GUI를 업데이트해야 합니다. 이는 이벤트 디스패치 스레드에서 실행될 `Runnable`을 예약하는 특별한 메서드를 사용하여 수행됩니다.

GUI 프로그래밍에서는 원인과 결과를 연결하기 위해 여러 스레드 변경을 따라야 하는 경우가 많습니다. 사용자는 이벤트 디스패치 스레드에서 작업을 시작하고, 이는 차례로 실행기를 통해 백그라운드 작업을 시작합니다. 완료 후, 해당 실행기는 이벤트 디스패치 스레드에 작업을 푸시합니다. 마지막 작업이 성능 문제를 일으키면 이는 원래 이벤트에서 두 번의 스레드 변경이 발생한 것입니다.

마지막으로 JProfiler는 모든 Kotlin 백엔드에 대해 구현된 Kotlin의 멀티 스레딩 솔루션인 Kotlin 코루틴 [(1)] 을 지원합니다. 비동기 실행 자체는 코루틴이 시작되는 지점입니다. Kotlin 코루틴의 디스패치 메커니즘은 유연하며 실제로 현재 스레드에서 시작하는 것을 포함할 수 있으며, 이 경우 "비동기 실행" 노드에는 노드의 텍스트에 별도로 보고되는 인라인 부분이 있습니다.

[(1)] https://kotlinlang.org/docs/reference/coroutines.html

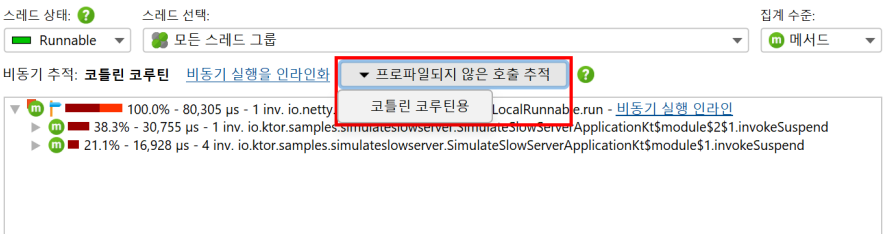


중단 메서드는 실행을 중단할 수 있으며, 이는 다른 스레드에서 다시 시작될 수 있습니다. 중단이 감지된 메서드에는 실제 호출 횟수와 메서드의 의미적 호출 횟수를 보여주는 툴팁이 있는 추가 "중단" 아이콘이 있습니다. Kotlin 코루틴은 의도적으로 중단될 수 있지만 스레드에 바인딩되지 않기 때문에 대기 시간은 호출 트리 어디에도 나타나지 않습니다. 코루틴 실행이 완료될 때까지 소요된 총 시간을 확인하기 위해 "비동기 실행" 노드 아래에 코루틴의 전체 중단 시간을 캡처하는 "중단된" 시간 노드가 추가됩니다. 비동기 실행의 CPU 시간 또는 실제 시간에 관심이 있는지 여부에 따라 분석 상단의 "중단된 시간 표시" 체크박스를 사용하여 이러한 노드를 실시간으로 추가하거나 제거할 수 있습니다.

프로파일되지 않은 호출 사이트 추적

기본적으로 실행기 및 Kotlin 코루틴 추적은 호출 사이트가 프로파일된 클래스에 있는 비동기 실행만 추적합니다. 이는 프레임워크 및 라이브러리가 이러한 비동기 메커니즘을 사용하여 직접적으로 코드 실행과 관련이 없는 방식으로 사용할 수 있으며, 추가된 호출 및 실행 사이트가 오버헤드와 혼란을 초래할 수 있기 때문입니다. 그러나 프로파일되지 않은 호출 사이트를 추적하는 사용 사례가 있습니다. 예를 들어, 프레임워크가 Kotlin 코루틴을 시작하고 그 위에서 코드가 실행되는 경우입니다.

프로파일되지 않은 클래스에서 이러한 호출 사이트가 감지되면 호출 트리 및 핫스팟 뷰의 추적 정보에 해당 알림 메시지가 표시됩니다. 라이브 세션에서는 이러한 뷰에서 직접 실행기 및 Kotlin 코루틴 추적을 위해 프로파일되지 않은 호출 사이트 추적을 개별적으로 켤 수 있습니다. 이러한 옵션은 세션 설정 대화 상자의 "CPU 프로파일링" 단계에서 언제든지 변경할 수 있습니다.



Kotlin 코루틴은 CPU 녹화가 활성화된 동안 시작된 경우에만 추적할 수 있음을 이해하는 것이 중요합니다. 나중에 CPU 녹화를 시작하면 Kotlin 코루틴의 비동기 실행은 인라인될 수 없습니다. JProfiler는 프로파일되지

않은 클래스에서 호출 사이트를 감지한 경우와 마찬가지로 알림을 제공합니다. 애플리케이션 시작 시 시작되는 장기 실행 코루틴을 프로파일링해야 하는 경우 attach 모드를 사용하는 것은 옵션이 아닙니다. 이 경우 -agentpath VM 매개변수 [p. 11]로 JVM을 시작하고 시작 시 CPU 녹화를 시작하십시오.

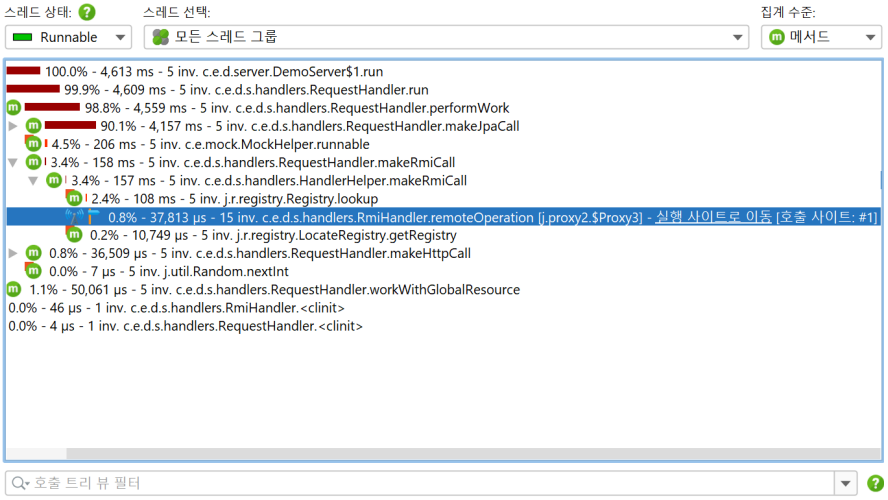
원격 요청 추적

선택된 통신 프로토콜에 대해 JProfiler는 메타데이터를 삽입하고 JVM 경계를 넘어 요청을 추적할 수 있습니다. 지원되는 기술은 다음과 같습니다:

- HTTP: HttpURLConnection, java.net.http.HttpClient, Apache Http Client 4.x, Apache Async Http Client 4.x, OkHttp 3.9+ 클라이언트 측, Servlet-API 구현 또는 Jetty(서블릿 없이) 서버 측
- Jersey Async Client 2.x, RestEasy Async Client 3.x, Cxf Async Client 3.1.1+의 비동기 JAX-RS 호출에 대한 추가 지원
- 웹 서비스: JAX-WS-RI, Apache Axis2 및 Apache CXF
- RMI
- gRPC
- 원격 EJB 호출: JBoss 7.1+ 및 Weblogic 11+

JProfiler에서 요청을 추적하려면 두 VM을 프로파일링하고 별도의 JProfiler 최상위 창에서 동시에 열어야 합니다. 이는 라이브 세션과 스냅샷 모두에서 작동합니다. 대상 JVM이 현재 열려 있지 않거나 원격 호출 시 CPU 녹화가 활성화되지 않은 경우 호출 사이트 하이퍼링크를 클릭하면 오류 메시지가 표시됩니다.

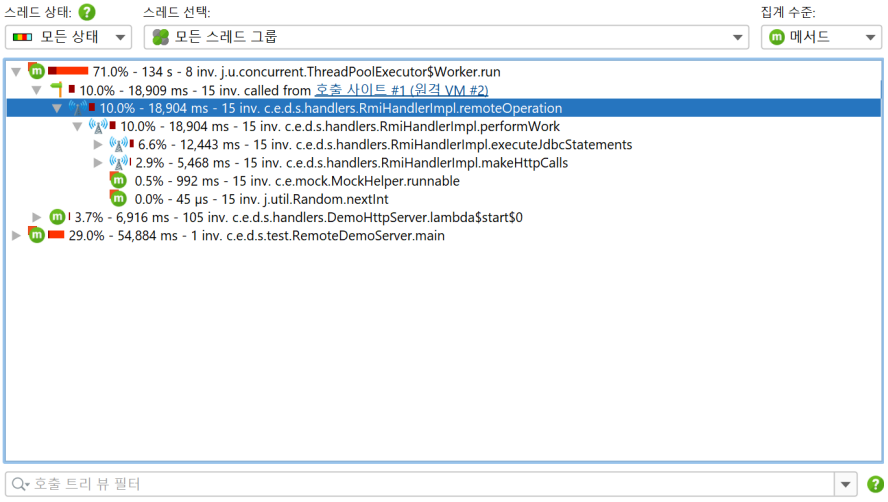
원격 요청을 추적할 때 JProfiler는 관련 JVM의 호출 트리에서 호출 사이트와 실행 사이트를 명시적으로 표시합니다. JProfiler의 호출 사이트는 기록된 원격 요청이 수행되기 전의 마지막 프로파일된 메서드 호출입니다. 이는 다른 VM에 위치한 실행 사이트에서 작업을 시작합니다. JProfiler는 호출 트리 뷰에 표시된 하이퍼링크를 사용하여 호출 사이트와 실행 사이트 간에 이동할 수 있도록 합니다.



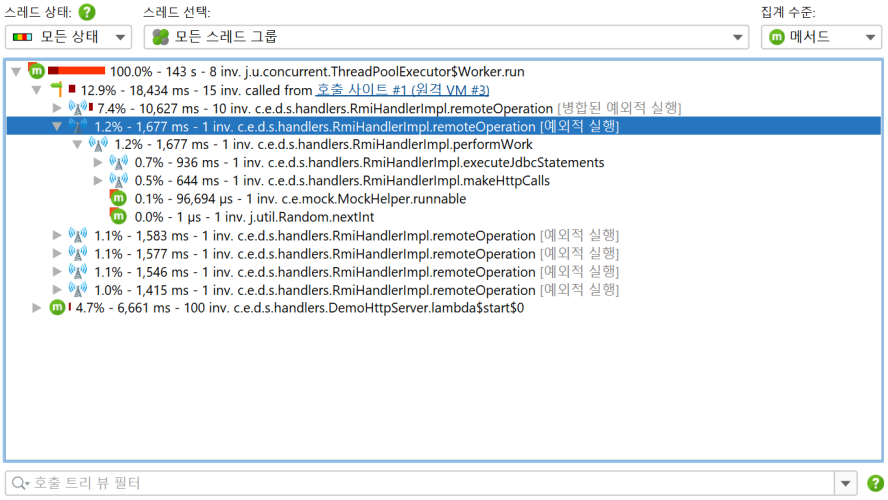
호출 사이트는 모든 스레드에 대해 원격 요청 추적과 관련하여 동일한 ID를 가집니다. 이는 호출 사이트에서 실행 사이트로 또는 그 반대로 이동할 때 스레드 해상도가 없으며 이동 시 항상 "모든 스레드 그룹" 및 "모든 스레드 상태" 스레드 상태 선택이 활성화되어 대상이 표시된 트리의 일부가 되도록 보장됨을 의미합니다.

호출 사이트와 실행 사이트는 1:n 관계에 있습니다. 호출 사이트는 여러 실행 사이트에서 원격 작업을 시작할 수 있으며, 특히 서로 다른 원격 VM에 있는 경우 그렇습니다. 동일한 VM에서는 단일 호출 사이트에 대한 여러 실행 사이트가 발생할 가능성이 적습니다. 호출 스택이 다를 때만 발생할 수 있기 때문입니다. 호출 사이트가 둘 이상의 실행 사이트를 호출하는 경우 대화 상자에서 하나를 선택할 수 있습니다.

실행 사이트는 특정 호출 사이트에 의해 시작된 모든 실행을 포함하는 호출 트리의 합성 노드입니다. 실행 사이트 노드의 하이퍼링크를 통해 해당 호출 사이트로 돌아갈 수 있습니다.



동일한 호출 사이트가 동일한 실행 사이트를 반복적으로 호출하는 경우 실행 사이트는 모든 호출의 병합된 호출 트리를 표시합니다. 원하지 않는 경우 예외적인 메소드 [p. 184] 기능을 사용하여 호출 트리를 더 분할할 수 있습니다. 아래 스크린샷에 표시된 대로입니다.



단일 호출 사이트에서만 참조되는 실행 사이트와 달리 호출 사이트 자체는 여러 실행 사이트에 연결될 수 있습니다. 호출 사이트의 숫자 ID를 통해 다른 실행 사이트에서 참조되는 경우 동일한 호출 사이트를 인식할 수 있습니다. 또한 호출 사이트는 원격 VM의 ID를 표시합니다. 프로파일된 VM의 ID는 상태 표시줄에서 볼 수 있습니다. 이는 JProfiler가 내부적으로 관리하는 고유 ID가 아니라 JProfiler에서 열리는 각 새로운 프로파일된 VM에 대해 1부터 시작하여 증가하는 표시 ID입니다.

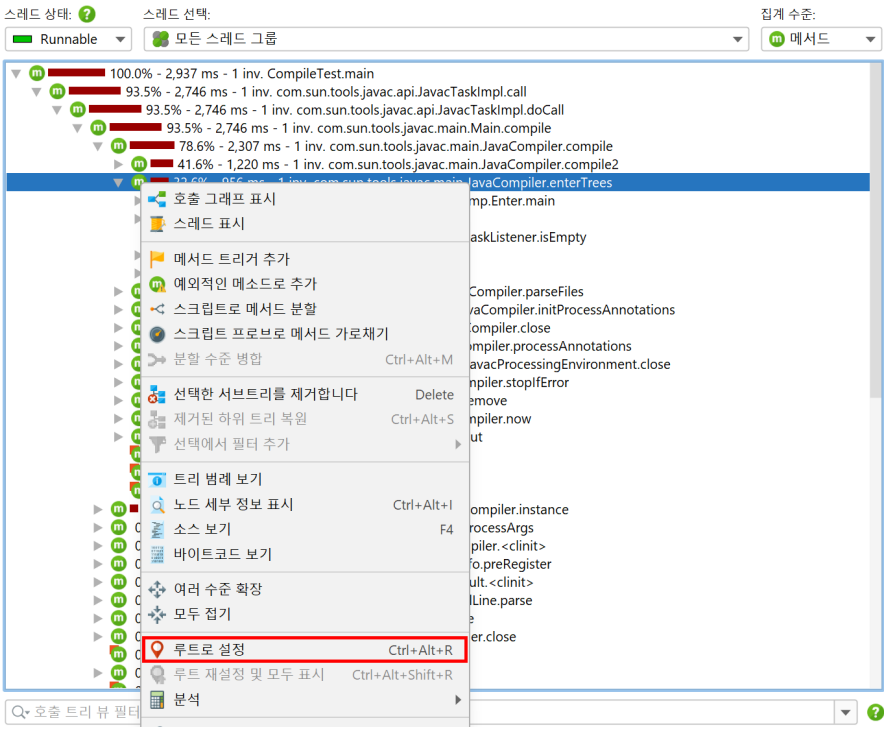
B.3 호출 트리의 일부 보기

호출 트리는 종종 너무 많은 정보를 포함하고 있습니다. 표시된 세부 정보를 줄이고 싶을 때는 여러 가지 방법이 있습니다: 특정 서브트리로 표시된 데이터를 제한하거나, 모든 불필요한 데이터를 제거하거나, 메서드 호출을 표시하기 위한 더 거친 필터를 사용할 수 있습니다. 이러한 모든 전략은 JProfiler에서 지원됩니다.

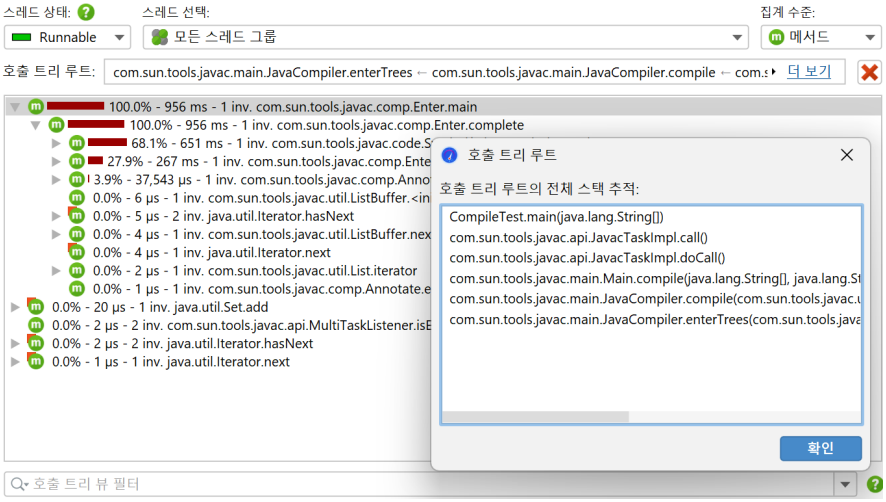
호출 트리 루트 설정

순차적으로 실행되는 여러 작업으로 구성된 사용 사례를 프로파일링할 경우, 각 서브트리를 개별적으로 분석할 수 있습니다. 그러한 하위 작업의 진입점을 찾으면, 주변 호출 트리는 단지 방해 요소가 되며 서브트리의 타이밍 백분율은 전체 호출 트리의 루트를 참조하게 됩니다.

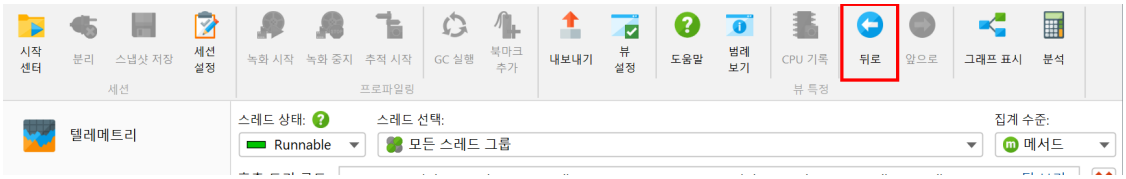
특정 서브트리에 집중하기 위해, JProfiler는 호출 트리와 할당 호출 트리 뷰에서 루트로 설정 컨텍스트 액션을 제공합니다.



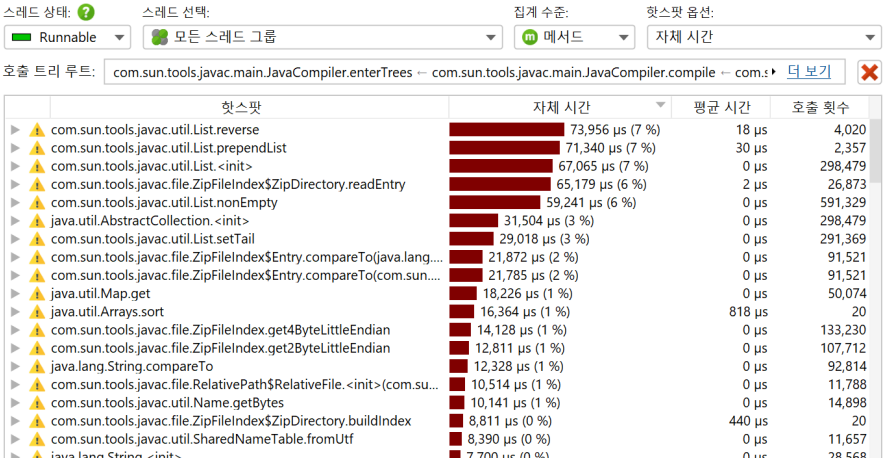
호출 트리 루트를 설정한 후, 선택한 루트에 대한 정보가 뷰 상단에 표시됩니다. 스크롤 가능한 단일 레이블은 루트까지 이어지는 마지막 몇 개의 스택 요소를 보여주며, 자세히 보기 버튼을 클릭하면 호출 트리 루트의 전체 스택을 포함한 세부 대화 상자가 표시됩니다.



루트 설정 액션을 재귀적으로 사용할 때, 호출 스택 접두사는 단순히 연결됩니다. 이전 호출 트리로 돌아가려면, 호출 트리 히스토리의 뒤로 버튼을 사용하여 한 번에 하나의 루트 변경을 실행 취소하거나, 컨텍스트 메뉴에서 루트 재설정 및 모두 보기 액션을 사용하여 한 번에 원래 트리로 돌아갈 수 있습니다.



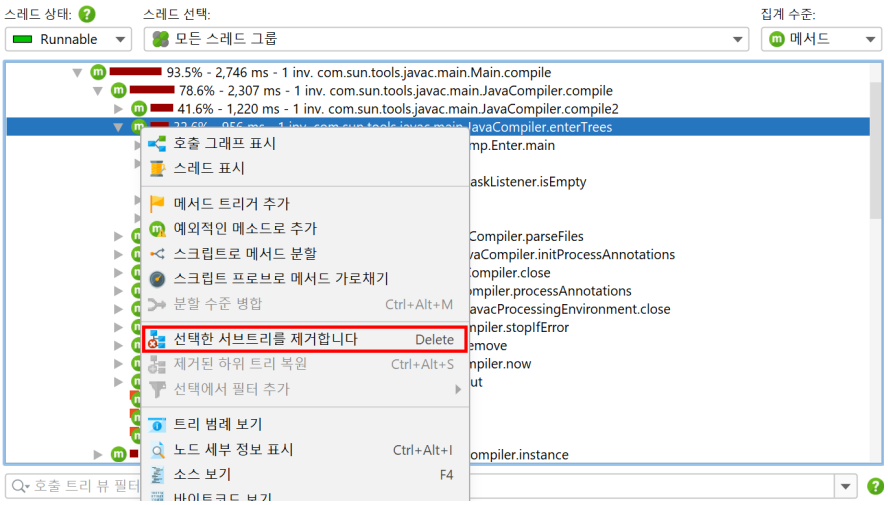
호출 트리 루트를 변경할 때 가장 중요한 것은 핫스팟 뷰가 전체 트리가 아닌 선택한 루트에 대해서만 계산된 데이터를 표시한다는 것입니다. 핫스팟 뷰 상단에는 호출 트리 뷰와 마찬가지로 현재 호출 트리 루트가 표시되어 표시된 데이터의 컨텍스트를 상기시킵니다.



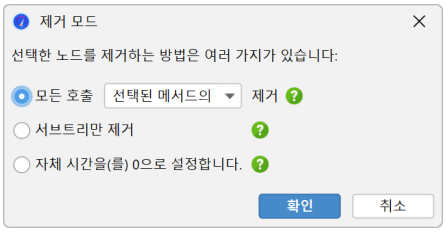
호출 트리의 일부 제거

때로는 특정 메서드가 존재하지 않는 경우 호출 트리가 어떻게 보일지 확인하는 것이 도움이 됩니다. 예를 들어, 개발 환경에서 빠르게 반복할 수 없는 프로덕션 시스템의 스냅샷을 사용하고 있기 때문에 한 번에 여러 성능 문제를 해결해야 하는 경우가 있을 수 있습니다. 주요 성능 문제를 해결한 후에는 두 번째 문제를 분석하고 싶지만, 첫 번째 문제가 트리에서 제거되어야만 명확하게 볼 수 있습니다.

호출 트리의 노드는 선택하여 Delete 키를 누르거나 컨텍스트 메뉴에서 선택한 서브트리 제거를 선택하여 서브트리와 함께 제거할 수 있습니다. 상위 노드의 시간은 숨겨진 노드가 존재하지 않는 것처럼 적절히 수정됩니다.

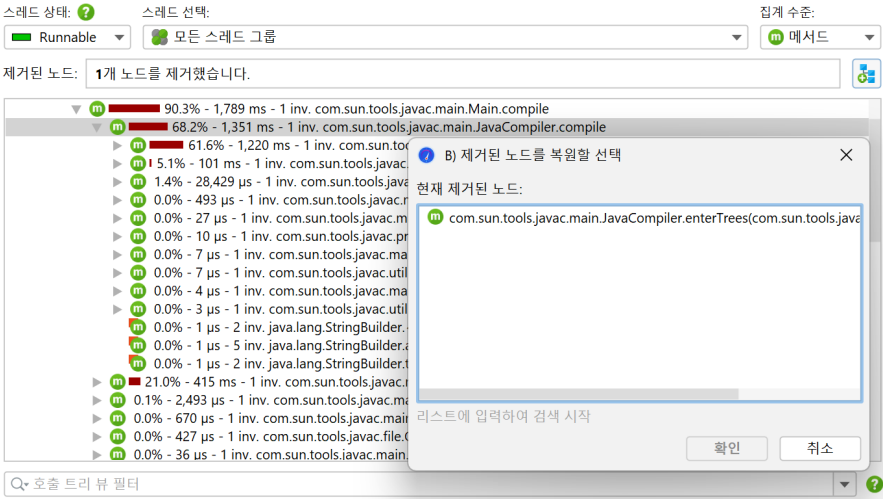


제거 모드는 세 가지가 있습니다. 모든 호출 제거 모드에서는 JProfiler가 전체 호출 트리에서 선택한 메서드의 모든 호출을 검색하고 서브트리와 함께 제거합니다. 서브트리만 제거 옵션은 선택한 서브트리만 제거합니다. 마지막으로, 자체 시간을 0으로 설정은 호출 트리에 선택한 노드를 남겨두지만 자체 시간을 0으로 설정합니다. 이는 Thread.run과 같은 많은 시간을 프로파일되지 않은 클래스에서 포함할 수 있는 컨테이너 노드에 유용합니다.



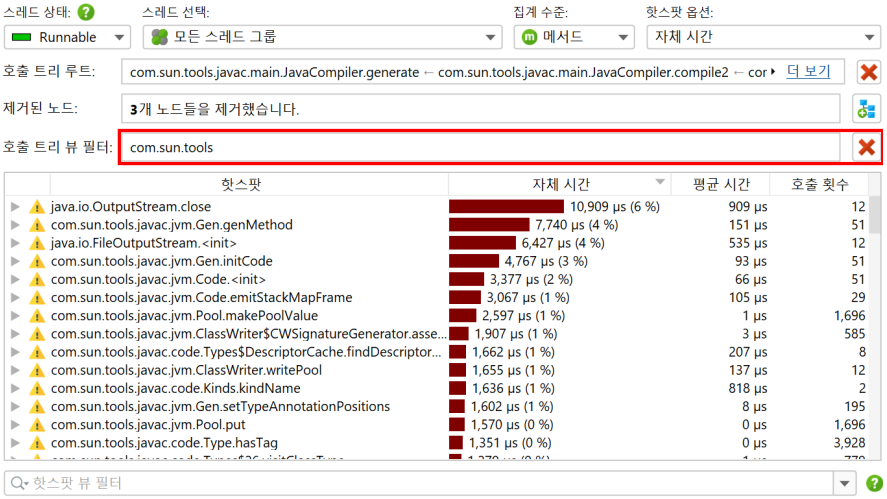
루트로 설정 액션과 마찬가지로, 제거된 노드는 핫스팟 뷰에 영향을 미칩니다. 이 방법으로, 해당 메서드가 중요하지 않은 기여로 최적화된 경우 핫스팟이 어떻게 보일지 확인할 수 있습니다.

노드를 제거하면 호출 트리와 핫스팟 뷰의 헤더 영역에 제거된 노드의 수와 제거된 서브트리 복원 버튼이 표시됩니다. 해당 버튼을 클릭하면 제거된 요소를 다시 표시할 수 있는 대화 상자가 나타납니다.



호출 트리 뷰 필터

핫스팟 뷰에 표시된 데이터에 영향을 미치는 호출의 세 번째 기능은 뷰 필터입니다. 호출 트리 필터를 변경하면 계산된 핫스팟 [p. 51]에 큰 영향을 미칩니다. 호출 트리 뷰와의 상호 의존성을 강조하기 위해, 핫스팟 뷰는 뷰 상단에 호출 트리 뷰 필터를 표시하고 추가 필터를 제거하는 버튼을 함께 제공합니다.

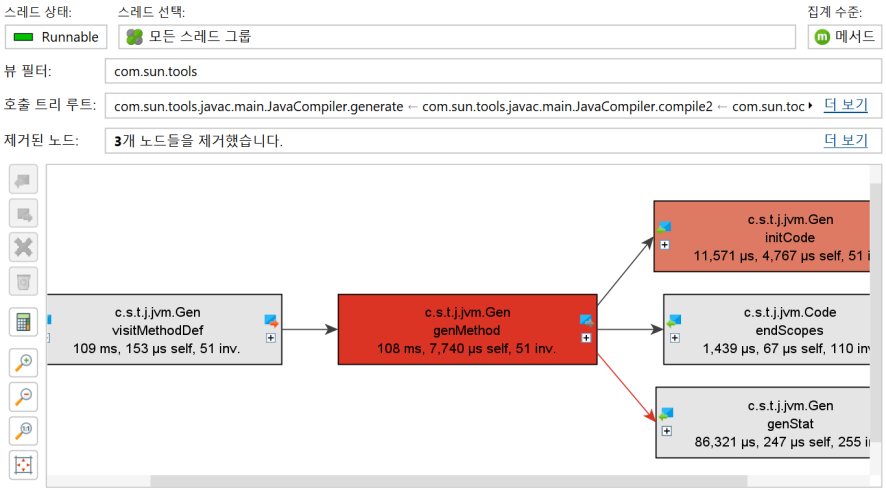


호출 트리 루트 설정, 호출 트리의 일부 제거 및 뷰 필터는 함께 사용할 수 있으며, 뷰 필터는 마지막에 설정해야 한다는 제한이 있습니다. 호출 트리에 뷰 필터가 설정되면, 루트로 설정 및 선택한 서브트리 제거 액션은 더 이상 작동하지 않습니다.

호출 그래프와의 상호작용

호출 트리 또는 핫스팟 뷰에서 그래프 보기 액션을 호출하면 동일한 호출 트리 루트로 제한되고, 제거된 메서드를 포함하지 않으며, 구성된 호출 트리 뷰 필터를 사용하는 그래프가 표시됩니다. 그래프 상단에는 이러한 변경 사항에 대한 정보가 호출 트리와 유사한 형태로 표시됩니다.

그래프 뷰 자체에서 새 그래프를 생성할 때, 마법사의 체크 박스를 통해 호출 그래프 계산에 고려해야 할 호출 트리 조정 기능을 선택할 수 있습니다. 각 체크 박스는 호출 트리 뷰에서 해당 기능이 현재 사용 중인 경우에만 표시됩니다.



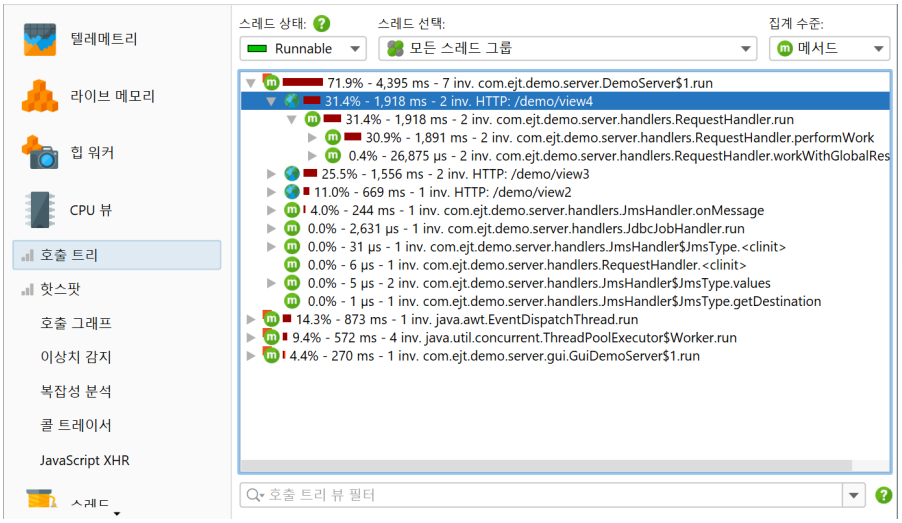
B.4 호출 트리 분할

호출 트리는 동일한 호출 스택의 반복 호출에 대해 누적됩니다. 이는 메모리 오버헤드와 데이터를 이해하기 쉽게 통합할 필요가 있기 때문입니다. 그러나 때로는 선택된 지점에서 누적을 중단하여 호출 트리의 일부를 별도로 볼 수 있습니다.

JProfiler는 호출 스택에 삽입되고 삽입된 노드 위의 메서드 호출에서 추출된 의미 정보를 보여주는 특별한 노드를 사용하여 호출 트리를 분할하는 개념을 가지고 있습니다. 이러한 분할 노드는 호출 트리 내에서 추가적인 페이로드 정보를 직접 볼 수 있게 하며, 포함된 서브트리를 별도로 분석할 수 있습니다. 각 분할 유형은 컨텍스트 메뉴의 작업으로 실시간으로 병합 및 분리할 수 있으며, 메모리 오버헤드를 제한하기 위해 분할 노드의 총 수에 제한이 있습니다.

호출 트리 분할 및 프로브

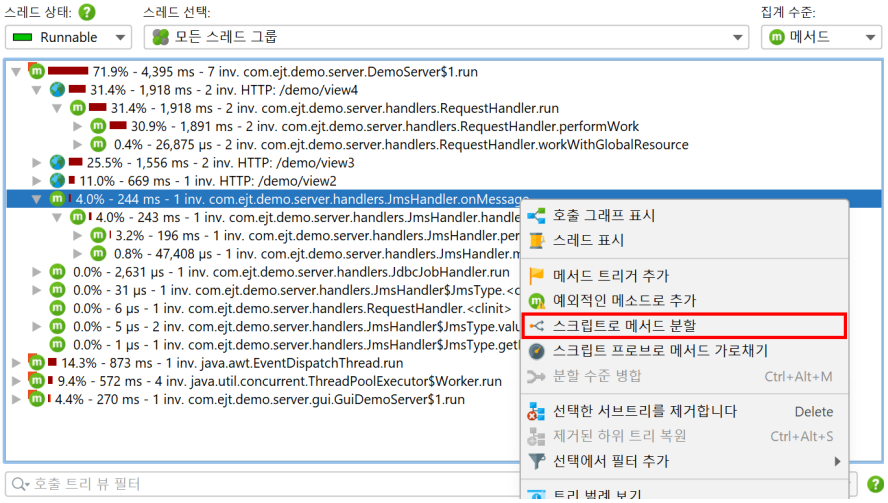
프로브 [p. 98]는 관심 있는 선택된 메서드에서 수집한 정보에 따라 호출 트리를 분할할 수 있습니다. 예를 들어, "HTTP 서버" 프로브는 각 다른 URL에 대해 호출 트리를 분할합니다. 이 경우 분할은 매우 구성 가능하여 URL의 원하는 부분만 포함하거나 서블릿 컨텍스트의 다른 정보 또는 여러 분할 수준을 생성할 수 있습니다.



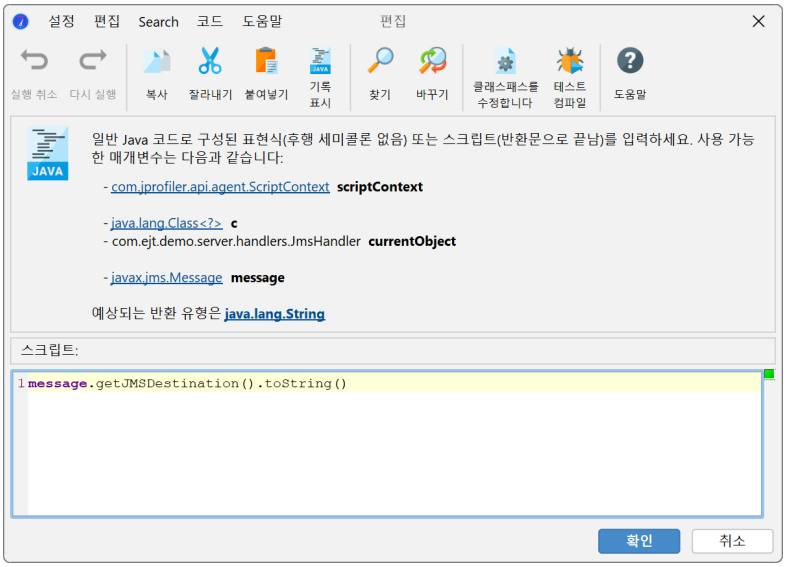
사용자 정의 프로브를 작성하면 임베디드 [p. 158] 및 인젝티드 [p. 153] 사용자 정의 프로브 시스템을 사용하여 동일한 방식으로 호출 트리를 분할할 수 있습니다.

스크립트로 메서드 분할

프로브에 사용할 수 있는 동일한 분할 기능을 호출 트리에서 직접 스크립트로 메서드 분할 작업을 사용하여 사용할 수 있습니다. 아래 스크린샷에서는 다른 유형의 메시지 처리를 별도로 보기 위해 JMS 메시지 핸들러에 대한 호출 트리를 분할하려고 합니다.



프로브를 작성하는 대신 문자열을 반환하는 스크립트를 입력합니다. 이 문자열은 선택된 메서드에서 호출 트리를 그룹화하는 데 사용되며 분할 노드에 표시됩니다. `null`을 반환하면 현재 메서드 호출은 분할되지 않고 일반적으로 호출 트리에 추가됩니다.

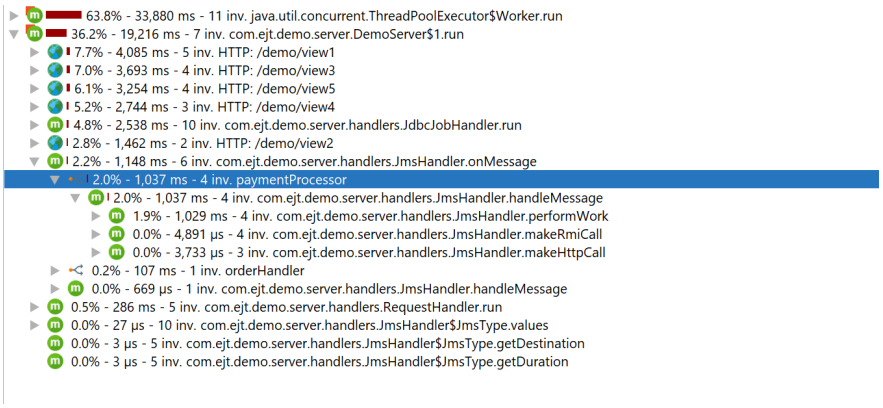


스크립트는 여러 매개변수에 액세스할 수 있습니다. 선택된 메서드의 클래스, 비정적 메서드의 인스턴스, 모든 메서드 매개변수가 전달됩니다. 추가로 데이터를 저장할 수 있는 `ScriptContext` 객체를 얻습니다. 동일한 스크립트의 이전 호출에서 일부 값을 기억해야 하는 경우, 컨텍스트에서 `getObject/putObject` 및 `getLong/putLong` 메서드를 호출할 수 있습니다. 예를 들어, 특정 메서드 매개변수 값이 처음으로 나타날 때만 분할하고 싶을 수 있습니다. 그런 경우 다음을 사용할 수 있습니다.

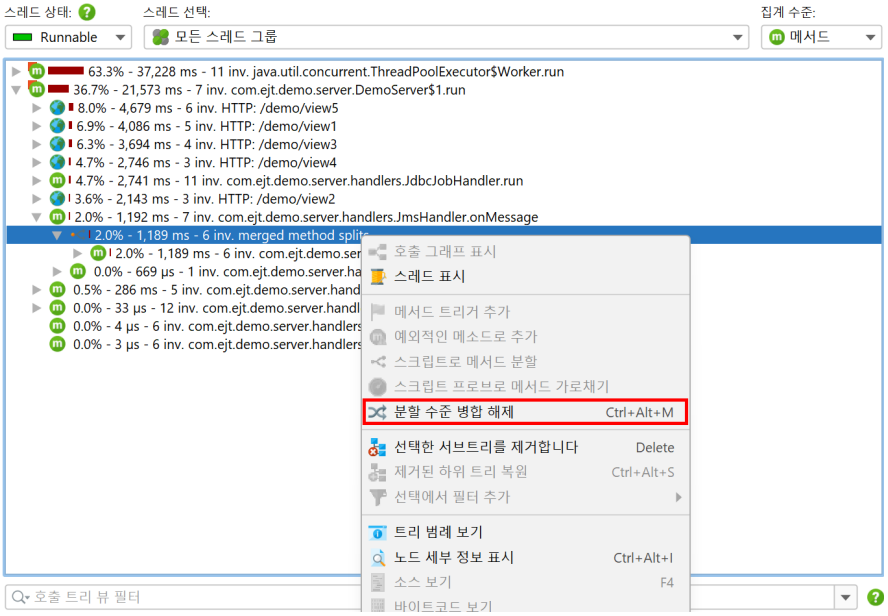
```
if (scriptContext.getObject(text) != null) {
    scriptContext.putObject(text);
    return text;
} else {
    return null;
}
```

분할 스크립트의 일부로 사용할 수 있습니다.

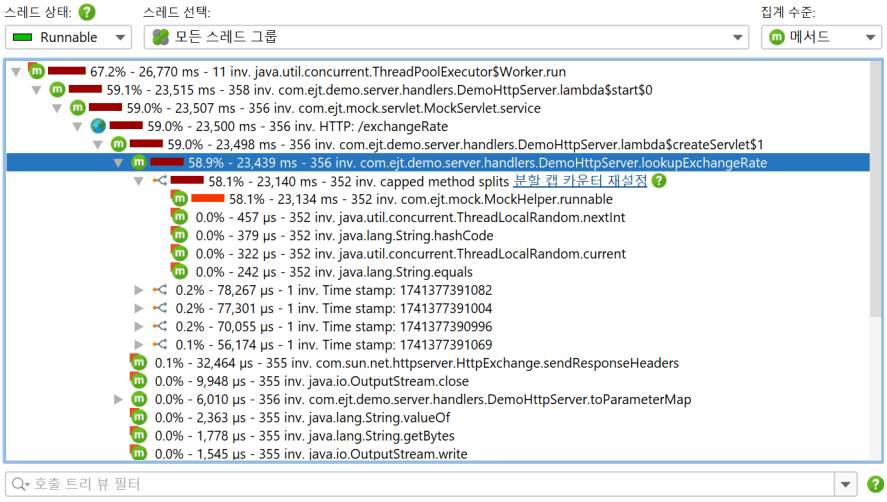
분할 노드는 선택된 메서드 아래에 삽입됩니다. 위의 스크린샷 예제에서는 이제 각 JMS 메시지 대상에 대한 처리 코드를 별도로 볼 수 있습니다.



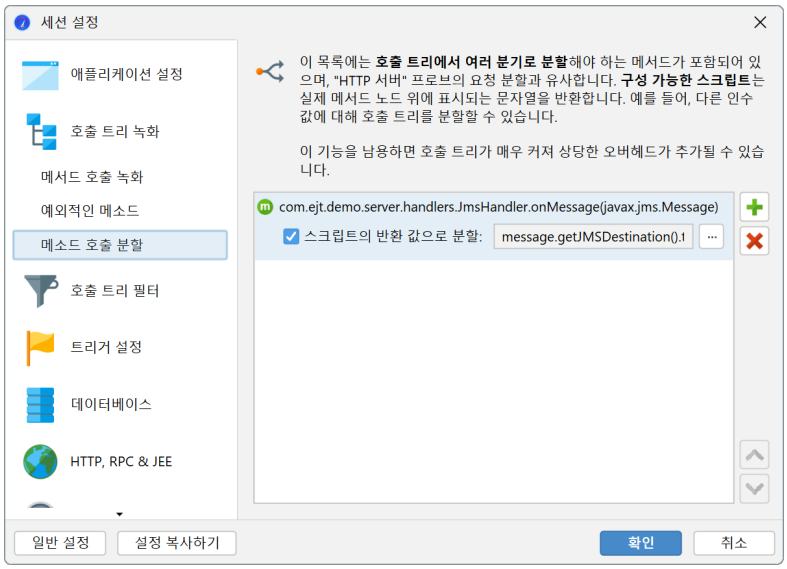
분할 위치는 선택된 호출 스택이 아닌 메서드에 바인딩됩니다. 동일한 메서드가 호출 트리의 다른 곳에 있으면 역시 분할됩니다. 분할 수준 병합 작업을 사용하면 모든 분할이 단일 노드로 병합됩니다. 그 노드는 분할을 다시 분리할 기회를 제공합니다.



너무 많은 분할을 생성하면 제한된 메서드 분할로 레이블이 지정된 노드가 모든 추가 분할 호출을 단일 트리로 누적하여 포함합니다. 노드의 하이퍼링크를 사용하여 캡 카운터를 재설정하고 더 많은 분할 노드를 기록할 수 있습니다. 분할의 최대 수를 영구적으로 늘리려면 프로파일링 설정에서 캡을 늘릴 수 있습니다.



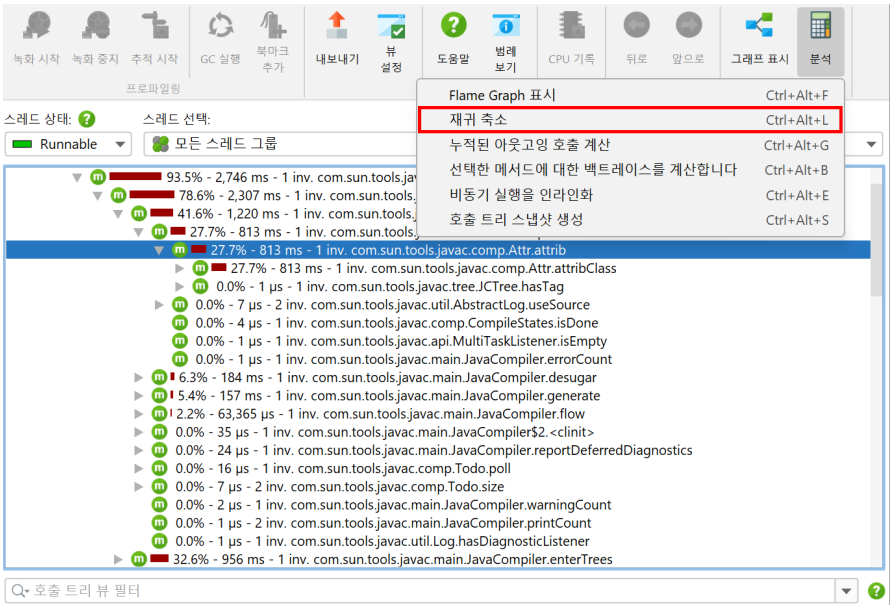
생성한 후 분할 메서드를 편집하려면 세션 설정 대화 상자로 이동하십시오. 특정 분할 메서드가 더 이상 필요하지 않지만 나중에 사용할 수 있도록 유지하려면 스크립트 구성 앞의 체크박스를 사용하여 비활성화할 수 있습니다. 이는 호출 트리에서 병합하는 것보다 낫습니다. 녹화 오버헤드가 상당할 수 있기 때문입니다.



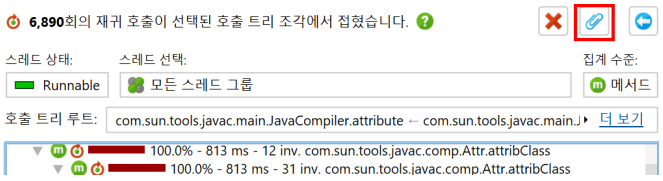
B.5 호출 트리 분석

호출 트리 [p. 51]는 JProfiler가 기록한 실제 호출 스택을 보여줍니다. 호출 트리를 분석할 때, 해석을 용이하게 하기 위해 호출 트리에 적용할 수 있는 몇 가지 변환이 있습니다. 이러한 변환은 시간이 많이 소요될 수 있으며 호출 트리 뷰의 기능과 호환되지 않는 방식으로 출력 형식을 변경하므로 분석 결과가 포함된 새로운 뷰가 생성됩니다.

이러한 분석을 수행하려면 호출 트리 뷰에서 노드를 선택하고 도구 모음 또는 컨텍스트 메뉴에서 호출 트리 분석 작업 중 하나를 선택하십시오.



호출 트리 뷰 아래에 중첩된 뷰가 생성됩니다. 동일한 분석 작업을 다시 호출하면 분석이 대체됩니다. 여러 분석 결과를 동시에 유지하려면 결과 뷰를 고정할 수 있습니다. 이 경우 동일한 유형의 다음 분석은 새로운 뷰를 생성합니다. 고정된 뷰의 경우, 왼쪽의 뷰 선택기에 표시되는 이름을 변경할 수 있는 이름 변경 버튼이 뷰 상단에 표시됩니다.



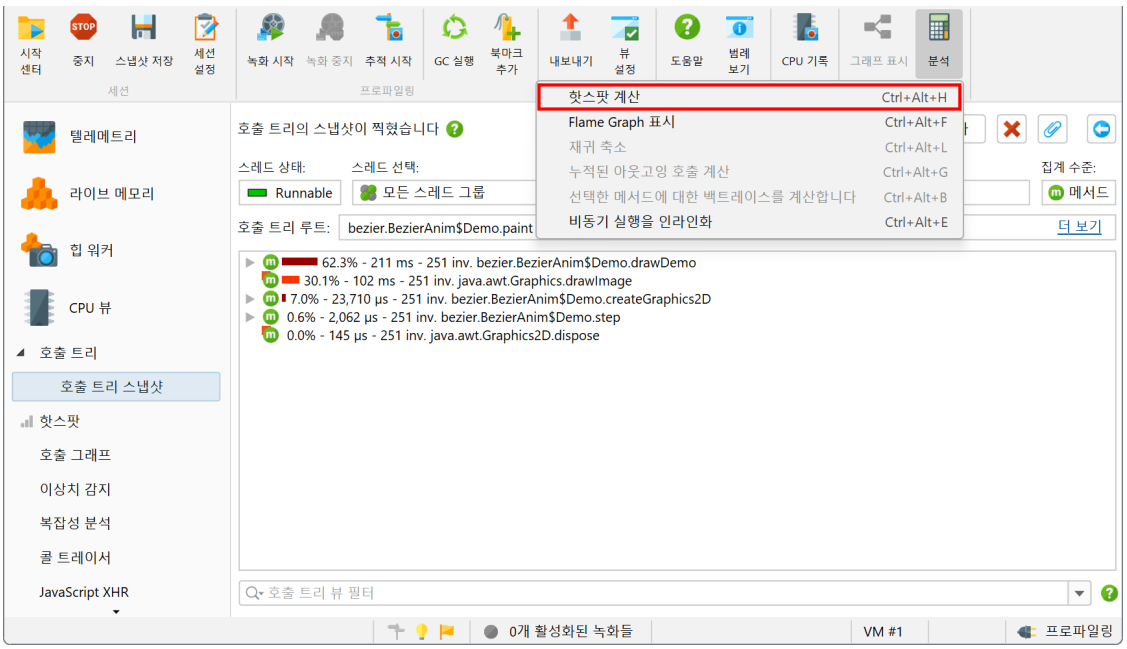
라이브 세션에서는 결과 뷰가 호출 트리와 함께 업데이트되지 않으며 분석이 수행된 시점의 데이터를 표시합니다. 현재 데이터를 위한 분석을 다시 계산하려면 다시 로드 작업을 사용하십시오. 할당 트리에서 자동 업데이트가 비활성화된 경우와 같이 호출 트리 자체를 다시 계산해야 하는 경우에도 다시 로드 작업이 이를 처리합니다.

호출 트리 스냅샷

"호출 트리 스냅샷 생성" 분석은 현재 호출 트리의 정적 복사본을 단순히 생성합니다. 이는 JProfiler 스냅샷을 저장하고 열지 않고도 다양한 사용 사례를 비교하는 데 유용합니다. 또한 호출 트리가 여전히 기록 중일 때 고정된 복사본으로 작업할 수 있는 방법을 제공합니다.

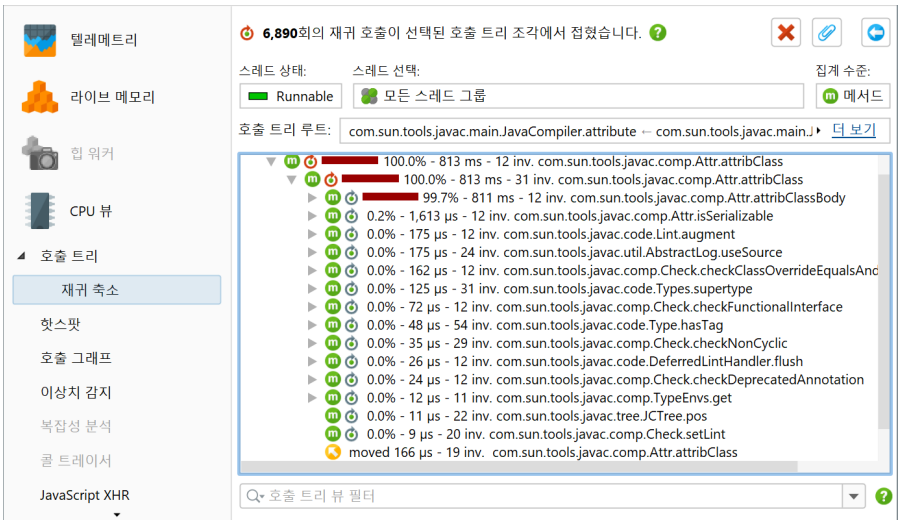
"호출 트리 스냅샷 생성" 분석은 "CPU 뷰" 섹션의 "호출 트리" 뷰에서만 사용할 수 있습니다. 호출 트리 스냅샷 뷰를 고정하면 여러 호출 트리 스냅샷을 동시에 가질 수 있습니다. 다른 분석과 달리 호출 트리 스냅샷은 별도의 데이터 집합을 구성하기 때문에 JProfiler 스냅샷에 저장됩니다.

호출 트리 뷰에서 사용할 수 있는 호출 트리 분석 외에도 호출 트리 스냅샷에는 부모 뷰의 핫스팟을 계산하는 "핫스팟 계산" 작업이 있습니다. 이는 "CPU 뷰" 섹션의 "핫스팟" 뷰와 유사합니다. 호출 트리 스냅샷 뷰 아래에 중첩된 뷰에서 액세스할 수 있는 모든 분석은 최상위 호출 트리 뷰의 데이터가 아닌 부모 호출 트리 스냅샷의 데이터로 작업합니다.

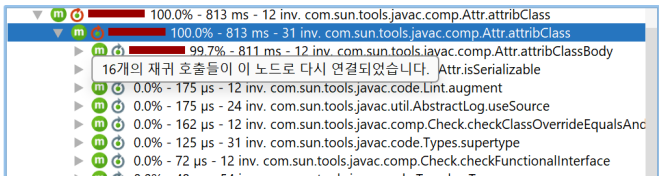


재귀 축소

재귀를 사용하는 프로그래밍 스타일은 분석하기 어려운 호출 트리를 초래합니다. "재귀 축소" 호출 트리 분석은 모든 재귀가 접힌 호출 트리를 계산합니다. 호출 트리에서 현재 선택한 항목의 부모 노드는 분석을 위한 호출 트리 루트 [p. 170] 역할을 합니다. 전체 호출 트리를 분석하려면 최상위 노드 중 하나를 선택하십시오.



동일한 메서드가 호출 스택 상단에서 이미 호출된 경우 재귀가 감지됩니다. 이 경우 하위 트리가 호출 트리에서 제거되고 해당 메서드의 첫 번째 호출로 다시 연결됩니다. 그런 다음 호출 트리의 해당 노드는 재귀 횟수를 보여주는 도구 팁이 있는 아이콘으로 접두사가 붙습니다. 해당 노드 아래에서 서로 다른 깊이의 스택이 병합됩니다. 병합된 스택의 수는 도구 팁에도 표시됩니다. 접힌 재귀의 총 수는 원래 호출 트리에 대해 설정된 호출 트리 매개변수 정보 위의 헤더에 표시됩니다.

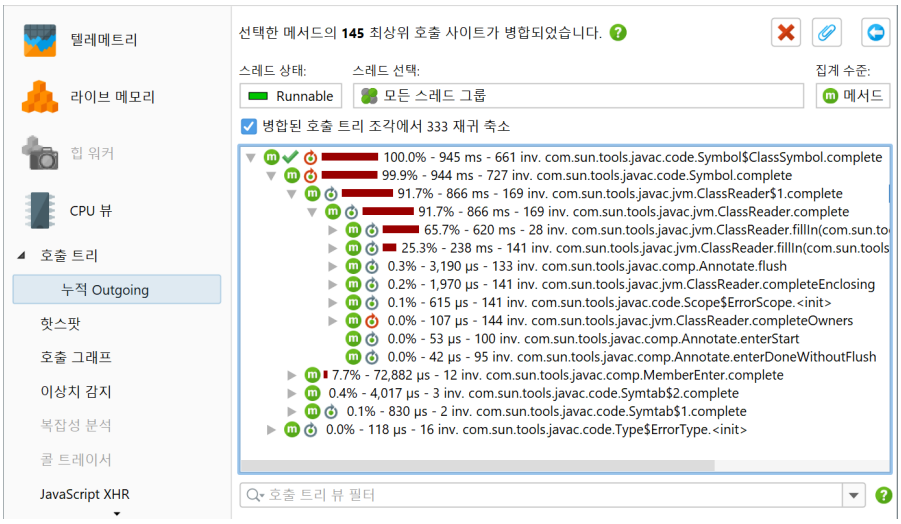


간단한 재귀의 경우 병합된 스택의 수는 재귀 횟수에 1을 더한 값입니다. 따라서 재귀 도구 팁에 "1 재귀"가 표시된 노드는 재귀 도구 팁에 "2 병합된 스택"이 표시된 노드를 포함하는 트리를 포함합니다. 더 복잡한 경우에는 재귀가 중첩되어 겹치는 병합 호출 트리를 생성하므로 병합된 스택의 수는 스택 깊이에 따라 다릅니다.

하위 트리가 호출 트리에서 제거되어 상위로 병합되는 지점에서 특별한 "이동된 노드" 자리 표시자가 삽입됩니다.

누적된 아웃고잉 호출 분석

호출 트리에서 선택한 메서드에 대한 아웃고잉 호출을 볼 수 있지만 해당 메서드가 호출된 특정 호출 스택에 대해서만 볼 수 있습니다. 관심 있는 동일한 메서드가 다른 호출 스택에서 호출되었을 수 있으며, 더 나은 통계를 얻기 위해 이러한 모든 호출의 누적된 호출 트리를 분석하는 것이 종종 유용합니다. "누적된 아웃고잉 호출 계산" 분석은 메서드가 호출된 방식에 관계없이 선택한 메서드의 모든 아웃고잉 호출을 합산한 호출 트리를 보여줍니다.

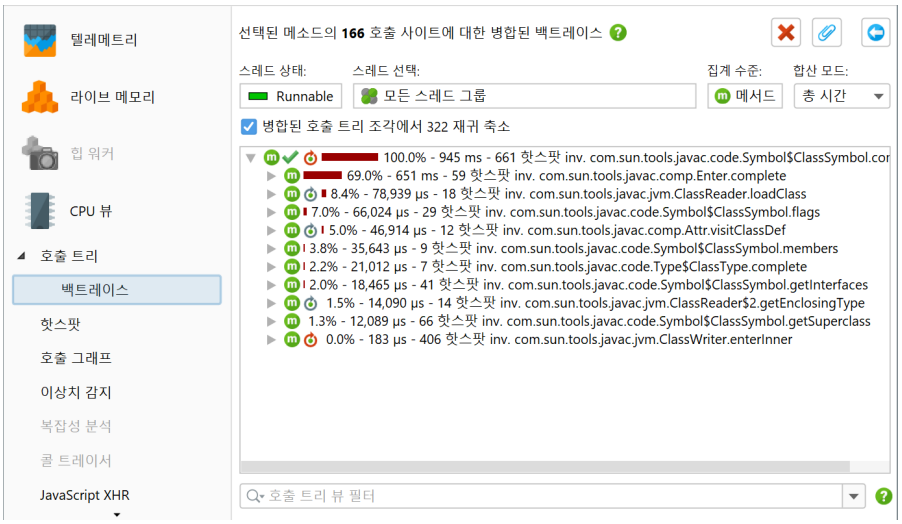


선택한 메서드에 대해 JProfiler는 재귀 호출을 고려하지 않고 모든 최상위 호출을 수집하고 결과 트리에 누적합니다. 헤더에는 해당 프로세스에서 합산된 최상위 호출 사이트의 수가 표시됩니다.

뷰 상단에는 결과 트리에서 재귀를 축소할 수 있는 체크박스가 있으며, 이는 "재귀 축소" 분석과 유사합니다. 재귀가 축소되면 최상위 노드와 아웃고잉 호출의 첫 번째 레벨은 메서드 호출 그래프와 동일한 숫자를 표시합니다.

백트레이스 계산

"백트레이스 계산" 분석은 "누적된 아웃고잉 호출 계산" 분석을 보완합니다. 후자는 재귀 호출을 고려하지 않고 선택한 메서드의 모든 최상위 호출을 합산합니다. 그러나 아웃고잉 호출을 보여주는 대신 선택한 메서드의 호출에 기여하는 백트레이스를 보여줍니다. 호출은 가장 깊은 노드에서 시작하여 선택한 메서드로 진행됩니다.



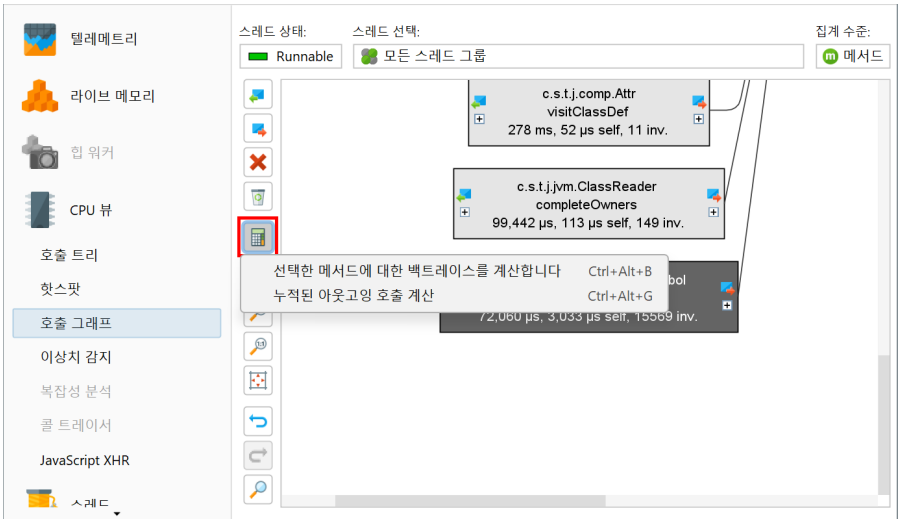
이 분석은 핫스팟 뷰와 유사하지만 기본적으로 선택한 메서드의 자체 시간 대신 총 시간을 합산하며, 핫스팟 뷰는 자체 시간이 총 시간의 중요한 부분인 메서드만 표시합니다. 뷰 상단에는 합산 모드라는 라디오 버튼 그룹이 있으며 자체 시간으로 설정할 수 있습니다. 이 선택으로 선택한 메서드의 합산 값은 핫스팟 뷰의 기본 모드와 일치합니다.

백트레이스에서 백트레이스 노드의 호출 횟수와 시간은 선택한 메서드와만 관련이 있습니다. 특정 호출 스택을 따라 호출이 선택한 메서드의 값에 얼마나 기여했는지를 보여줍니다. "누적된 아웃고잉 호출 계산" 분석과 유사하게 재귀를 축소할 수 있으며 백트레이스의 첫 번째 레벨은 메서드 호출 그래프의 인커밍 호출과 동일합니다.

호출 그래프에서 호출 트리 분석

호출 그래프에서는 각 메서드가 고유하지만 호출 트리에서는 메서드가 여러 호출 스택에 나타날 수 있습니다. 선택한 메서드 하나에 대해 "누적된 아웃고잉 호출 계산" 및 "백트레이스 계산" 분석은 호출 트리와 호출 그래프의 관점 사이의 다리 역할을 합니다. 선택한 메서드를 중심에 두고 아웃고잉 및 인커밍 호출을 트리로 보여줍니다. 호출 그래프 표시 작업을 통해 언제든지 전체 그래프로 전환할 수 있습니다.

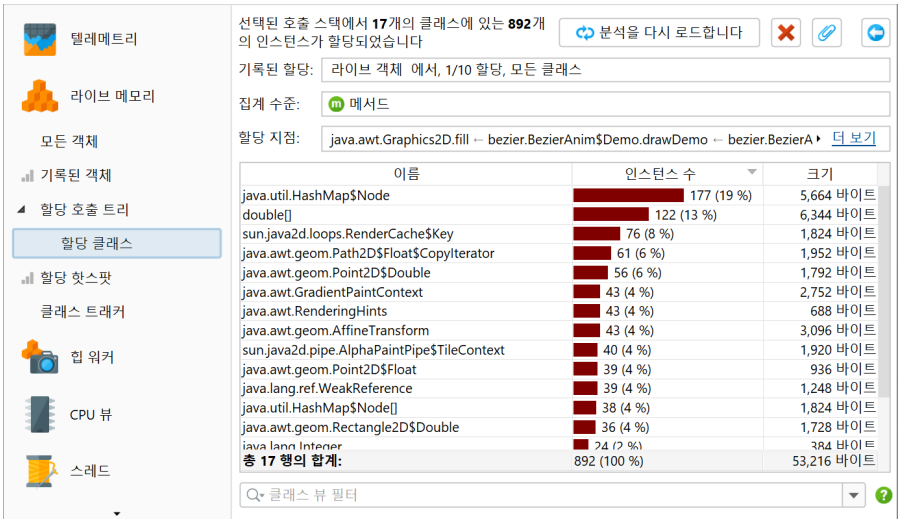
때로는 반대 방향으로 관점을 전환하여 그래프에서 트리 뷰로 변경하고 싶을 때가 있습니다. 호출 그래프에서 작업할 때 그래프의 선택한 노드에 대해 호출 그래프와 동일한 호출 트리 분석을 사용하여 누적된 아웃고잉 호출과 백트레이스를 트리로 표시할 수 있습니다.



IntelliJ IDEA 통합 [p. 133]에서는 편집기의 구석에 표시되는 호출 그래프에 이러한 트리를 직접 표시하는 작업이 포함되어 있습니다.

할당을 위한 클래스 표시

이전 호출 트리 분석과 약간 다른 점은 할당 호출 트리와 할당 핫스팟 뷰에서 "클래스 표시" 분석입니다. 이는 호출 트리를 다른 트리로 변환하지 않고 모든 할당된 클래스를 포함하는 테이블을 표시합니다. 결과 뷰는 기록된 객체 뷰 [p. 68]와 유사하지만 특정 할당 지점에 제한됩니다.



호출 트리를 보여주는 분석 결과 뷰에서는 "누적된 아웃고잉 호출 계산" 및 "선택한 메서드로의 백트레이스 계산" 분석을 사용할 수 있습니다. 이를 호출하면 독립적인 매개변수를 가진 새로운 최상위 분석이 생성됩니다. 이전 분석 결과 뷰에서의 호출 트리 제거는 새로운 최상위 분석에 반영되지 않습니다.

반면 클래스 표시 작업은 호출 트리 분석 결과 뷰에서 사용될 때 새로운 최상위 분석을 생성하지 않습니다. 대신 원래 뷰에서 두 레벨 아래에 중첩된 분석을 생성합니다.

C 고급 CPU 분석 뷰

C.1 이상치 감지 및 예외적인 메소드 녹화

어떤 상황에서는 메소드의 평균 호출 시간이 문제가 아니라, 가끔씩 메소드가 잘못 동작하는 것이 문제일 수 있습니다. 호출 트리에서는 모든 메소드 호출이 누적되므로, 10000번 호출 중 한 번 100배 더 오래 걸리는 자주 호출되는 메소드는 총 시간에 뚜렷한 흔적을 남기지 않습니다.

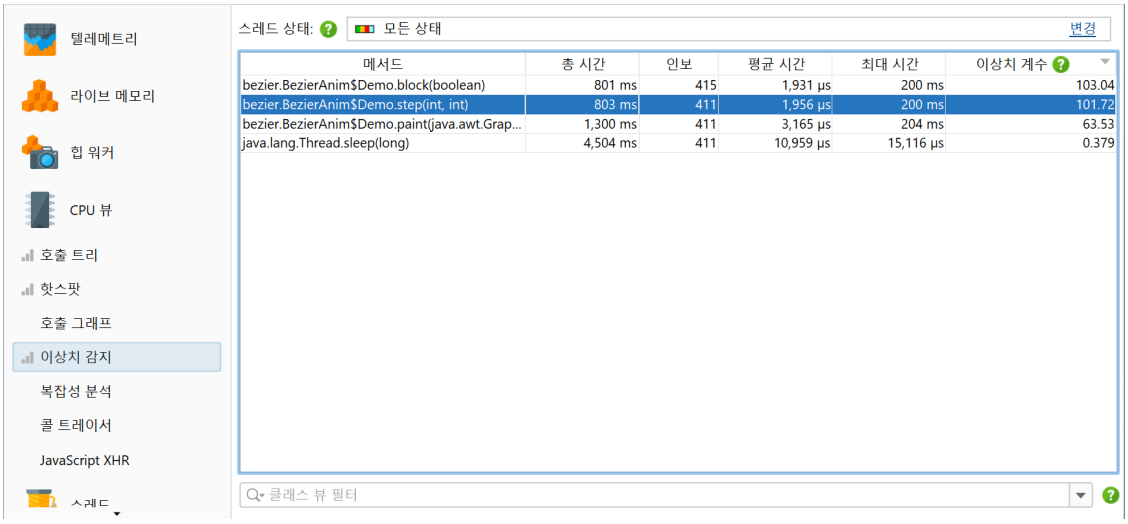
이 문제를 해결하기 위해 JProfiler는 호출 트리에서 이상치 감지 뷰와 예외적인 메소드 녹화 기능을 제공합니다.

이상치 감지 뷰

이상치 감지 뷰는 각 메소드의 호출 지속 시간과 호출 횟수에 대한 정보를 최대 시간과 함께 보여줍니다. 최대 호출 시간이 평균 시간에서 얼마나 벗어나는지를 통해 모든 호출 지속 시간이 좁은 범위에 있는지 아니면 유의미한 이상치가 있는지를 알 수 있습니다. 이상치 계수는 다음과 같이 계산됩니다:

```
(최대 시간 - 평균 시간) / 평균 시간
```

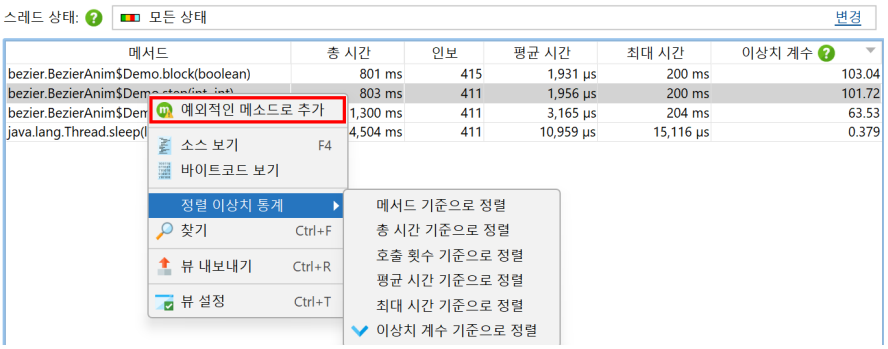
이와 같은 방식으로 메소드를 정량화하는 데 도움이 될 수 있습니다. 기본적으로 테이블은 가장 높은 이상치 계수를 가진 메소드가 상단에 오도록 정렬됩니다. 이상치 감지 뷰의 데이터는 CPU 데이터가 녹화된 경우에 사용할 수 있습니다.



몇 번만 호출되거나 매우 짧게 실행되는 메소드로 인한 과도한 혼잡을 피하기 위해, 최대 시간과 호출 횟수에 대한 하한값을 뷰 설정에서 설정할 수 있습니다. 기본적으로 최대 시간이 10ms 이상이고 호출 횟수가 10을 초과하는 메소드만 이상치 통계에 표시됩니다.

예외적인 메소드 녹화 설정

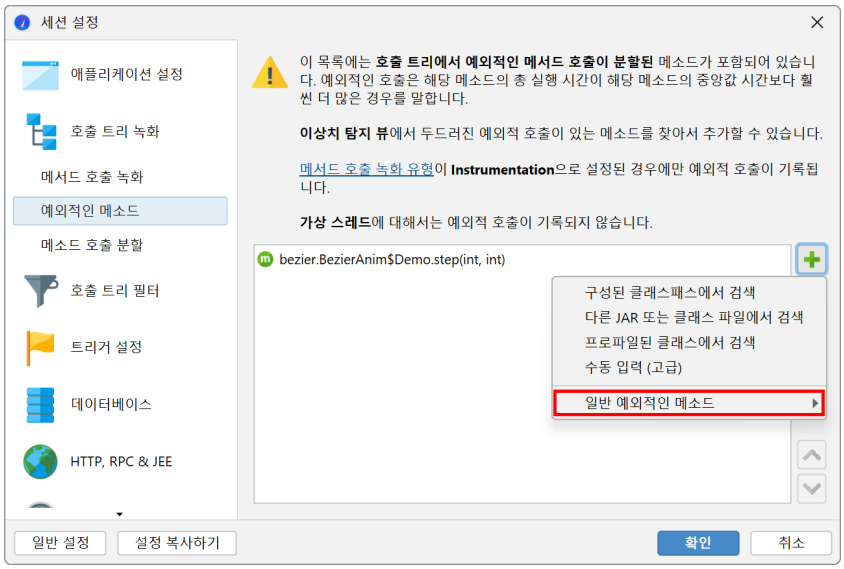
예외적인 호출 지속 시간을 겪는 메소드를 식별한 후에는 컨텍스트 메뉴에서 예외적인 메소드로 추가할 수 있습니다. 동일한 컨텍스트 메뉴 동작은 호출 트리 뷰에서도 사용할 수 있습니다.



예외적인 메소드 녹화를 위해 메소드를 등록하면, 가장 느린 몇 번의 호출이 호출 트리에 별도로 유지됩니다. 다른 호출은 일반적으로 단일 메소드 노드로 병합됩니다. 별도로 유지되는 호출의 수는 프로파일링 설정에서 구성할 수 있으며, 기본값은 5로 설정되어 있습니다.

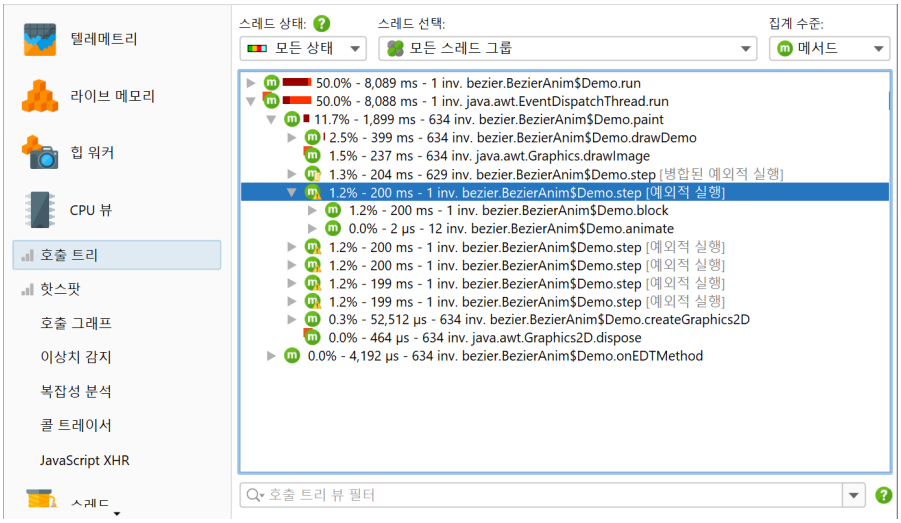
느린 메소드 호출을 구분할 때, 시간 측정을 위해 특정 스레드 상태가 사용되어야 합니다. 이는 CPU 뷰의 스레드 상태 선택이 아니며, 이는 단지 표시 옵션일 뿐 녹화 옵션이 아닙니다. 기본적으로 벽시계 시간이 사용되지만, 프로파일링 설정에서 다른 스레드 상태를 구성할 수 있습니다. 동일한 스레드 상태가 이상치 감지 뷰에도 사용됩니다.

세션 설정에서는 호출 트리나 이상치 감지 뷰의 컨텍스트 없이 예외적인 메소드를 제거하거나 새로 추가할 수 있습니다. 또한, 예외적인 메소드 구성은 AWT 및 JavaFX 이벤트 디스패치 메커니즘과 같이 예외적으로 오래 실행되는 이벤트가 주요 문제가 되는 잘 알려진 시스템에 대한 예외적인 메소드 정의를 추가할 수 있는 옵션을 제공합니다.



호출 트리의 예외적인 메소드

호출 트리 뷰에서는 예외적인 메소드 실행이 다르게 표시됩니다.



분할된 메소드 노드는 수정된 아이콘과 추가 텍스트를 표시합니다:

- **[예외적 실행]**

  이러한 노드는 예외적으로 느린 메소드 실행을 포함합니다. 정의에 따라 호출 횟수는 하나일 것입니다. 나중에 많은 다른 메소드 실행이 더 느려지면, 이 노드는 사라지고 구성된 최대 별도로 녹화된 메소드 실행 수에 따라 "병합된 예외적 실행" 노드에 추가될 수 있습니다.

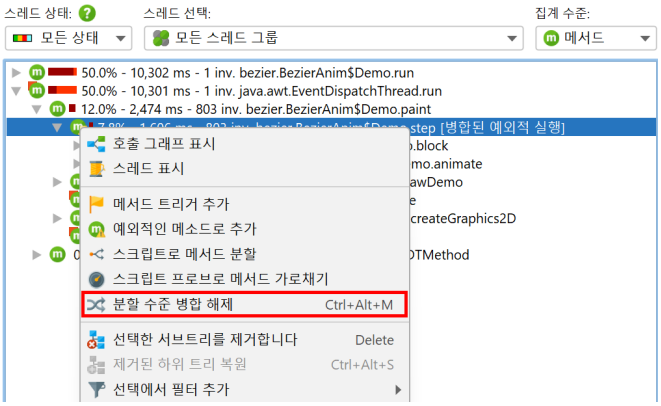
- **[병합된 예외적 실행]**

  예외적으로 느리지 않은 메소드 호출은 이 노드로 병합됩니다. 어떤 호출 스택에서도 예외적인 메소드당 하나의 노드만 있을 수 있습니다.

- **[현재 예외적 실행]**

호출 트리 뷰가 JProfiler GUI로 전송되는 동안 호출이 진행 중이었다면, 호출이 예외적으로 느린지 여부는 아직 알려지지 않았습니다. "현재 예외적 실행"은 현재 호출에 대한 별도로 유지된 트리를 보여줍니다. 호출이 완료된 후, 별도의 "예외적 실행" 노드로 유지되거나 "병합된 예외적 실행" 노드로 병합될 것입니다.

프로브 [p. 98] 및 분할 메소드 [p. 175]에 의한 호출 트리 분할과 마찬가지로, 예외적인 메소드 노드는 컨텍스트 메뉴에서 분할 수준 병합 작업을 통해 모든 호출을 실시간으로 병합 및 분리할 수 있습니다.

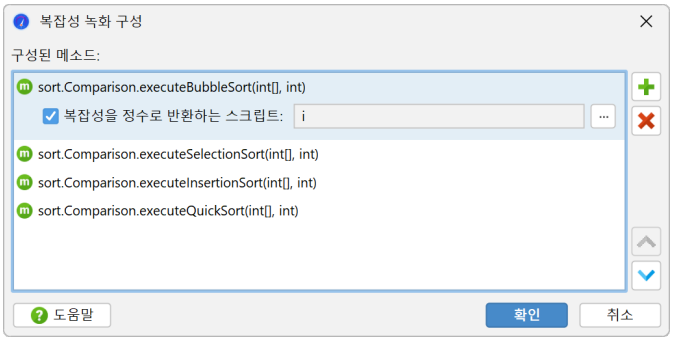


C.2 복잡도 분석

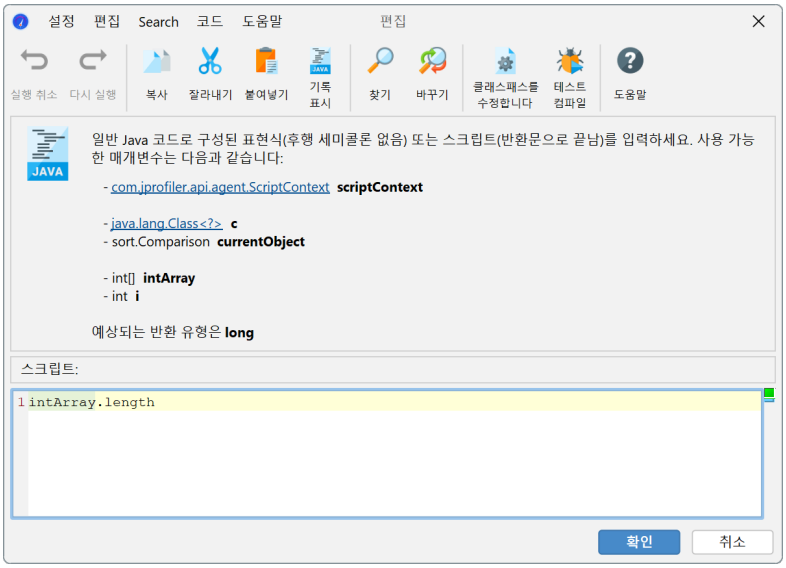
복잡도 분석 뷰는 선택된 메소드의 알고리즘 복잡도를 메소드 매개변수에 따라 조사할 수 있게 합니다.

빅 오 표기법에 대한 세부 사항을 새로 고치려면, 알고리즘 복잡도에 대한 소개 [(1)] 및 일반 알고리즘의 복잡도 비교 가이드 [(2)] 를 읽어보는 것이 좋습니다.

먼저, 모니터링할 메소드를 하나 이상 선택해야 합니다.



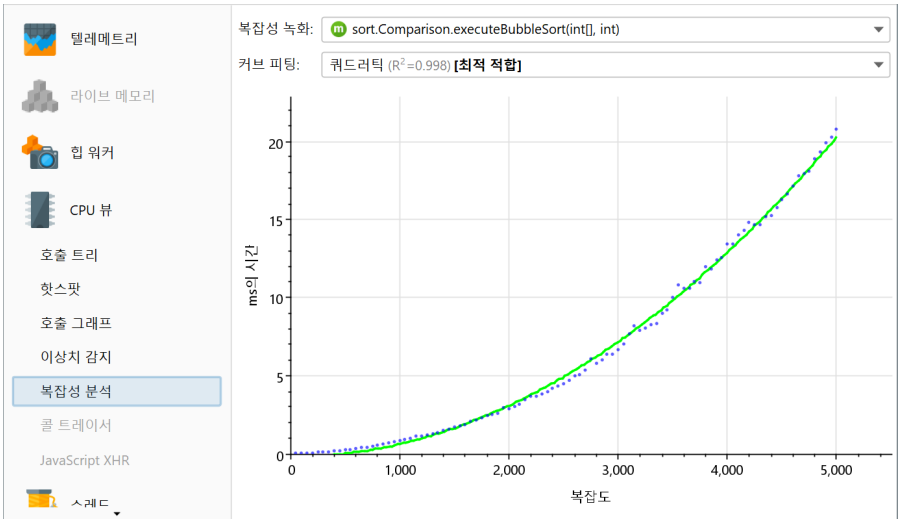
각 메소드에 대해, 현재 메서드 호출의 복잡도로 사용되는 `long` 유형의 반환 값을 가진 스크립트를 입력할 수 있습니다. 예를 들어, `java.util.Collection` 유형의 메소드 매개변수 중 하나가 `inputs`라는 이름이라면, 스크립트는 `inputs.size()`일 수 있습니다.



복잡도 녹화는 CPU 녹화와 독립적입니다. 복잡도 분석 뷰에서 직접 복잡도 녹화를 시작하고 중지할 수 있으며, 녹화 프로파일 또는 트리거 액션 [p. 26]을 사용할 수도 있습니다. 녹화가 중지된 후에는 결과가 x축의 복잡도와 y축의 실행 시간을 플로팅하는 그래프로 표시됩니다. 메모리 요구 사항을 줄이기 위해, JProfiler는 서로 다른 복잡도와 실행 시간을 공통 버킷으로 결합할 수 있습니다. 상단의 드롭다운을 사용하여 구성된 다른 메소드 간에 전환할 수 있습니다.

그래프는 버블 차트로, 각 데이터 포인트의 크기는 그 안에 있는 측정 수에 비례합니다. 모든 측정이 구별되면 일반적인 산포 차트를 볼 수 있습니다. 다른 극단적으로, 모든 메소드 호출이 동일한 복잡도와 실행 시간을 가지면 하나의 큰 원을 보게 됩니다.

[(1)] https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/
[(2)] https://bigocheatsheet.com/



데이터 포인트가 최소 3개 이상인 경우, 일반적인 복잡도로 곡선 피팅이 표시됩니다. JProfiler는 여러 일반적인 복잡도로부터 곡선 피팅을 시도하며 처음에는 최적의 피팅을 보여줍니다. 곡선 피팅을 위한 드롭다운을 사용하여 다른 곡선 피팅 모델도 표시할 수 있습니다. 곡선 피팅 설명에 포함된 R^2 값은 피팅의 적합성을 보여줍니다. 드롭다운의 모델은 R^2에 따라 내림차순으로 정렬되어 있으므로 최적의 모델이 항상 첫 번째 항목입니다.

R^2는 단지 표기법일 뿐 실제로는 아무것도 제곱하지 않기 때문에 음수가 될 수 있습니다. 음수 값은 상수 선과의 피팅보다 나쁜 피팅을 나타냅니다. 상수 선 피팅은 항상 R^2 값이 0이고 완벽한 피팅은 값이 1입니다.

현재 표시된 피팅의 매개변수를 내보내기 대화 상자에서 "속성" 옵션을 선택하여 내보낼 수 있습니다. 품질 보증 환경에서 자동 분석을 위해 명령줄 내보내기 [p. 240]는 속성 형식을 지원합니다.

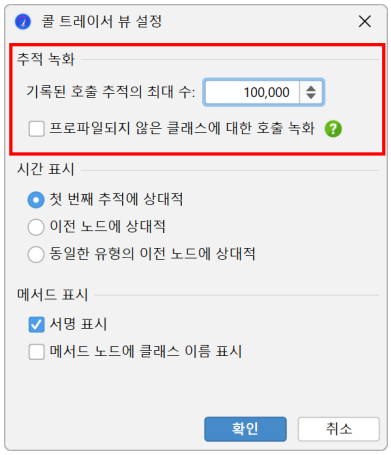
C.3 콜 트레이서

호출 트리에서 메서드 호출 녹화는 동일한 호출 스택을 가진 호출을 누적합니다. 메모리 요구 사항이 방대하고 기록된 데이터의 양이 많아 해석이 어렵기 때문에 정확한 시간 순서 정보를 유지하는 것은 일반적으로 실현 가능하지 않습니다.

그러나 제한된 상황에서는 호출을 추적하고 전체 시간 순서를 유지하는 것이 의미가 있습니다. 예를 들어, 여러 협력 스레드의 메서드 호출의 정확한 교차를 분석하고자 할 수 있습니다. 디버거는 이러한 사용 사례를 단계별로 진행할 수 없습니다. 또는 일련의 메서드 호출을 분석하고 싶지만 디버거에서처럼 한 번만 보는 것이 아니라 앞뒤로 이동할 수 있기를 원할 수 있습니다. JProfiler는 콜 트레이서를 통해 이 기능을 제공합니다.

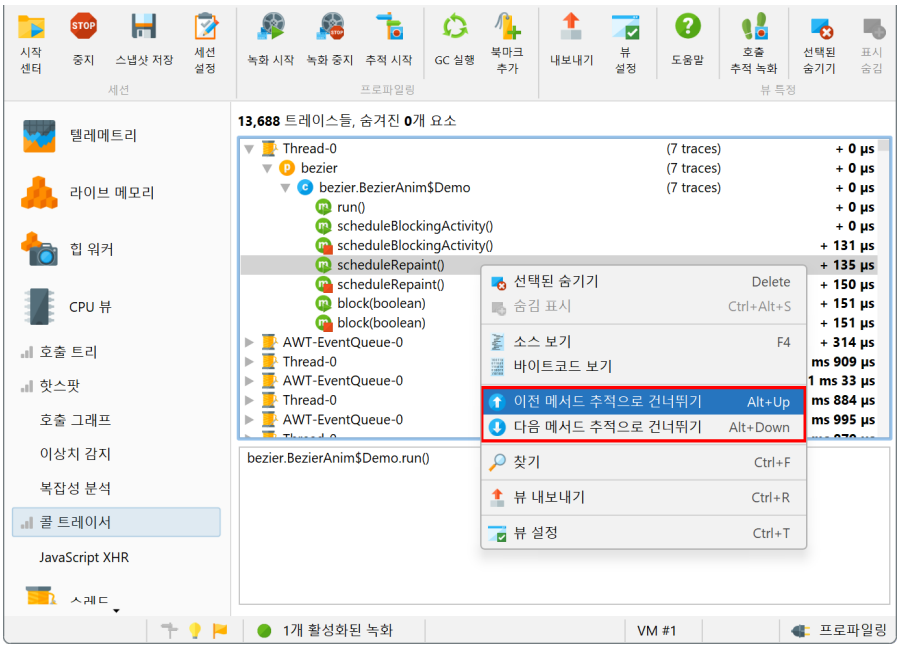
콜 트레이서는 콜 트레이서 뷰에서, 트리거 [p. 26] 또는 프로파일링 API [p. 122]로 활성화할 수 있는 별도의 녹화 작업을 가지고 있습니다. 과도한 메모리 소비 문제를 피하기 위해 수집된 호출 추적의 최대 수에 제한이 설정됩니다. 그 제한은 뷰 설정에서 구성할 수 있습니다. 수집된 추적의 비율은 필터 설정에 크게 의존합니다.

호출 추적은 메서드 호출 녹화 유형이 계측으로 설정된 경우에만 작동합니다. 샘플링은 단일 메서드 호출을 추적하지 않으므로 샘플링으로 호출 추적을 수집하는 것은 기술적으로 불가능합니다. 콜 트리에서와 마찬가지로 콜 트레이서에서 압축 필터링된 클래스에 대한 호출이 기록됩니다. 자신의 클래스에만 집중하고 싶다면 뷰 설정에서 이러한 호출을 제외할 수 있습니다.



추적된 메서드 호출은 관련된 호출을 쉽게 건너뛸 수 있도록 세 가지 수준으로 나누어진 트리로 표시됩니다. 세 그룹은 스레드, 패키지 및 클래스입니다. 이 그룹 중 하나의 현재 값이 변경될 때마다 새로운 그룹화 노드가 생성됩니다.

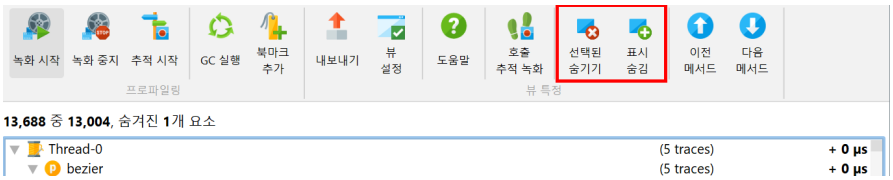
가장 낮은 수준에서는 메서드 진입 및 메서드 종료 노드가 있습니다. 호출 추적 아래에는 현재 선택된 메서드 추적의 스택 추적이 표시됩니다. 현재 메서드에서 다른 메서드로의 호출 추적이 기록되었거나 다른 스레드가 현재 메서드를 중단한 경우, 해당 메서드의 진입 및 종료 노드는 인접하지 않을 것입니다. 이전 메서드 및 다음 메서드 작업을 사용하여 메서드 수준에서만 탐색할 수 있습니다.



추적 및 모든 그룹화 노드에 표시되는 타이밍은 기본적으로 첫 번째 추적을 참조하지만 이전 노드 이후의 상대 시간을 표시하도록 변경할 수 있습니다. 이전 노드가 부모 노드인 경우, 그 차이는 0이 될 것입니다. 또한 동일한 유형의 이전 노드에 대한 상대 시간을 표시하는 옵션도 사용할 수 있습니다.

적절한 필터가 있어도 매우 짧은 시간에 많은 수의 추적이 수집될 수 있습니다. 관심 없는 추적을 제거하기 위해 콜 트레이서는 표시된 데이터를 빠르게 다듬을 수 있도록 합니다. 예를 들어, 특정 스레드는 관련이 없을 수 있으며 특정 패키지나 클래스의 추적은 흥미롭지 않을 수 있습니다. 또한, 재귀 메서드 호출은 많은 공간을 차지할 수 있으며 단일 메서드만 제거하고 싶을 수 있습니다.

노드를 선택하고 `delete` 키를 눌러 숨길 수 있습니다. 선택한 노드의 다른 모든 인스턴스와 관련된 모든 자식 노드도 숨겨질 것입니다. 뷰 상단에서 기록된 모든 추적 중 몇 개의 호출 추적이 여전히 표시되고 있는지 확인할 수 있습니다. 숨겨진 노드를 다시 표시하려면 숨겨진 항목 표시 도구 모음 버튼을 클릭할 수 있습니다.

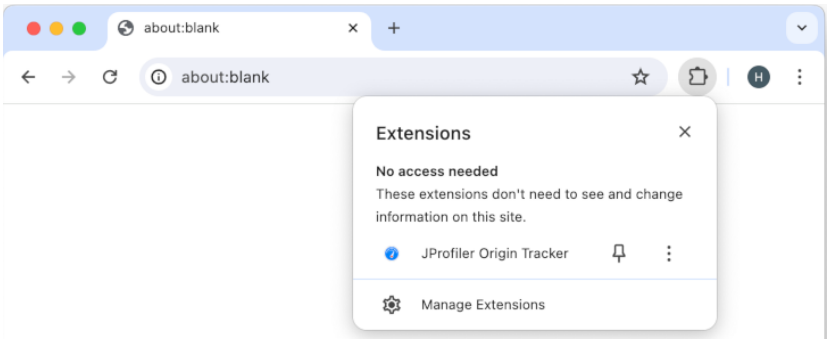


C.4 JavaScript XHR 원본 추적

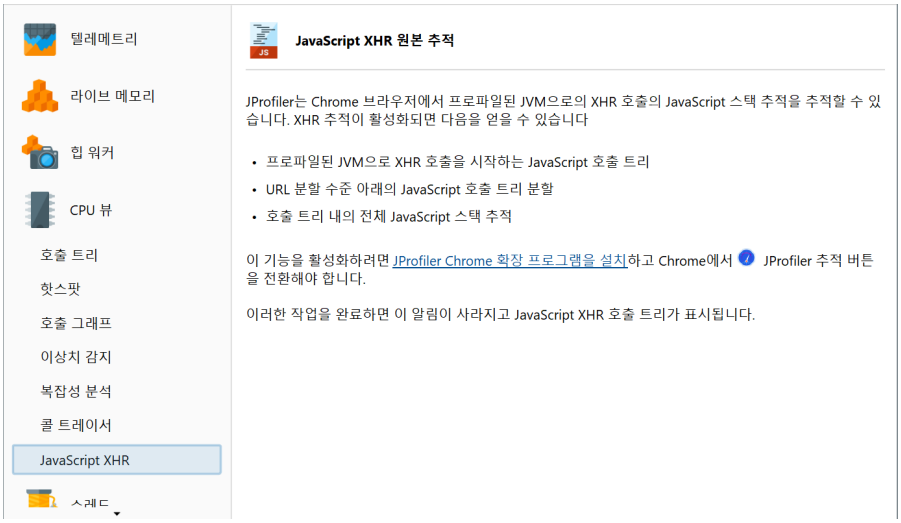
JavaScript XHR 원본 추적을 사용하면 브라우저에서 XMLHttpRequest[1] 또는 Fetch[2] 요청 중에 서로 다른 스택 트레이스에 대해 서블릿 호출을 분할할 수 있어, 프로파일된 JVM의 활동을 브라우저의 동작과 더 잘 연관시킬 수 있습니다. 다음에서 "XHR"은 XMLHttpRequest와 Fetch 메커니즘 모두를 지칭합니다.

브라우저 플러그인

이 기능을 사용하려면 브라우저로 Google Chrome[3]을 사용하고 JProfiler 원본 추적기 확장 프로그램[4]을 설치해야 합니다.



Chrome 확장 프로그램은 툴바에 JProfiler 아이콘이 있는 버튼을 추가하여 추적을 시작합니다. 추적을 시작하면 확장 프로그램은 모든 XHR 호출을 가로채고 로컬에서 실행 중인 JProfiler 인스턴스에 보고합니다. 추적이 시작되지 않은 경우 JProfiler는 JavaScript XHR 원본 추적을 설정하는 방법을 알려주는 정보 페이지를 표시합니다.



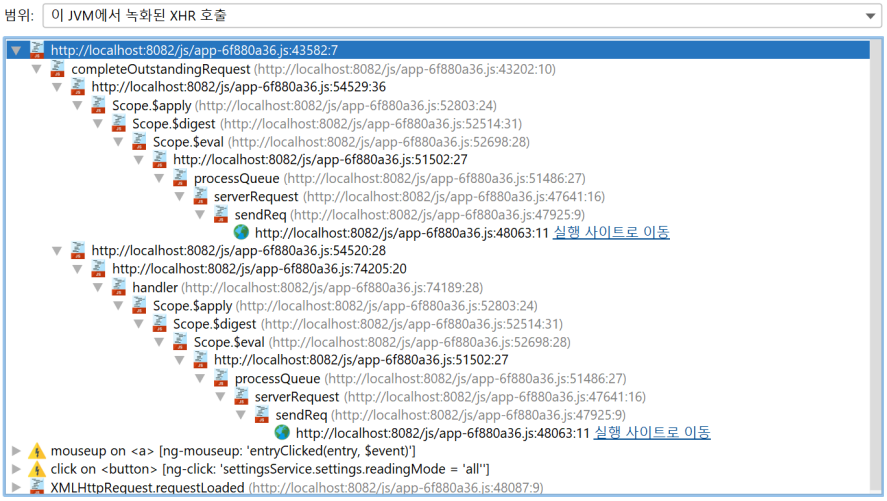
추적이 활성화되면 JProfiler 확장 프로그램은 페이지를 다시 로드하라는 메시지를 표시합니다. 이는 계측을 추가하기 위해 필요합니다. 페이지를 다시 로드하지 않기로 선택하면 이벤트 감지가 작동하지 않을 수 있습니다.

추적 상태는 도메인별로 지속됩니다. 추적이 활성화된 상태에서 브라우저를 다시 시작하고 동일한 URL을 방문하면 페이지를 다시 로드할 필요 없이 추적이 자동으로 활성화됩니다.

[1] https://xhr.spec.whatwg.org/
[2] https://fetch.spec.whatwg.org/
[3] http://www.google.com/chrome/
[4] https://chrome.google.com/webstore/detail/jprofiler-origin-tracker/mnicmpklpjkhohdbcdkflhochdfnmmbm

JavaScript XHR 트리

XHR 호출이 JProfiler의 활성 프로파일링 세션에 의해 프로파일된 JVM에 의해 처리되는 경우, JavaScript XHR 뷰는 이러한 호출의 누적 호출 트리를 표시합니다. 뷰가 비어 있는 경우, 뷰 상단의 "범위"를 "모든 XHR 호출"로 전환하여 XHR 호출이 발생했는지 확인할 수 있습니다.



Javascript 호출 스택 노드는 소스 파일과 라인 번호에 대한 정보를 포함합니다. XHR 호출이 이루어진 함수에는 특수 아이콘과 인접한 하이퍼링크가 있으며, XHR 호출이 프로파일된 JVM에 의해 처리된 경우 하이퍼링크가 제공됩니다. 하이퍼링크를 클릭하면 호출 트리 뷰 [p. 51]의 JavaScript 분할 노드로 이동하여 이 유형의 요청을 처리한 서버 측 호출 트리를 볼 수 있습니다.

트리의 상단에는 브라우저 이벤트 노드가 있으며, 이벤트 이름과 요소 이름을 중요한 속성과 함께 표시하여 이벤트의 출처를 파악하는 데 도움이 됩니다. 모든 요청에 관련된 이벤트가 있는 것은 아닙니다.

확장 프로그램은 여러 인기 있는 JavaScript 프레임워크를 인식하고 이벤트의 대상 노드에서 이벤트 리스너가 위치한 노드까지의 상위 계층을 탐색하여 표시 및 호출 트리 분할에 적합한 속성을 찾습니다. 프레임워크별 속성을 찾지 못하면 `id` 속성에서 멈춥니다. ID가 없는 경우 `a`, `button` 또는 `input`과 같은 "제어 요소"를 검색합니다. 모두 실패하면 이벤트 리스너가 등록된 요소가 표시됩니다.

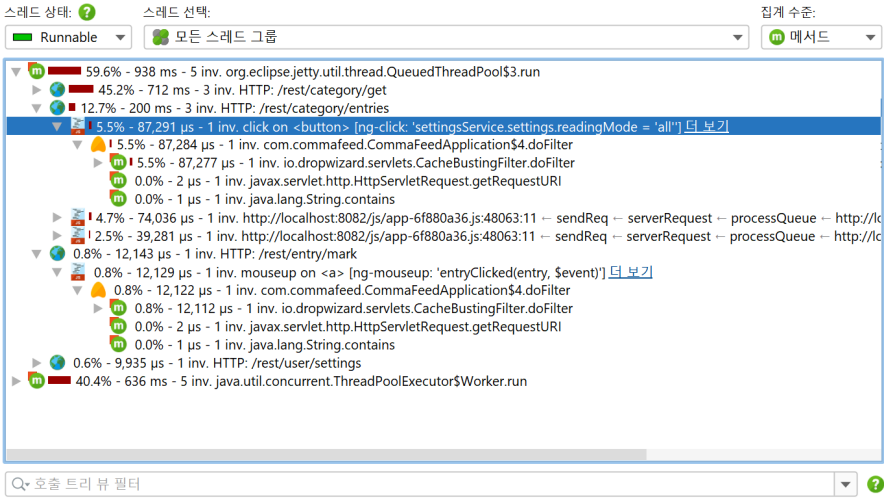
경우에 따라 흥미로운 속성의 자동 감지가 적합하지 않을 수 있으며 다른 호출 트리 분할을 선호할 수 있습니다. 예를 들어, 일부 프레임워크는 자동 ID를 할당하지만 모든 요소를 동작에 대한 의미적 설명과 함께 그룹화하는 것이 더 읽기 쉬울 수 있습니다. 다른 호출 트리 분할을 달성하려면 HTML 속성을 추가하십시오.

```
data-jprofiler="..."
```

대상 요소 또는 대상과 이벤트 리스너의 위치 사이의 요소에 추가합니다. 해당 속성의 텍스트는 분할에 사용되며 다른 속성은 무시됩니다.

호출 트리 분할

호출 트리 뷰에서 XHR 호출은 브라우저 이벤트와 호출 스택의 각 개별 조합에 대해 호출 트리를 분할합니다. 분할 노드는 브라우저 이벤트에 대한 정보를 표시합니다. `setTimeout()` 호출과 같이 진행 중인 이벤트가 없는 경우 마지막 몇 개의 스택 프레임이 인라인으로 표시됩니다.



이러한 노드의 "더 보기" 하이퍼링크는 뷰->노드 세부 정보 표시 작업에 의해 열리는 것과 동일한 세부 정보 대화 상자를 엽니다. JavaScript 분할 노드의 경우 세부 정보 대화 상자는 노드의 텍스트가 아니라 전체 브라우저 호출 스택을 표시합니다. 다른 JavaScript 분할 노드의 호출 스택을 검사하려면 비모달 세부 정보 대화 상자를 열어 두고 해당 노드를 클릭하십시오. 세부 정보 대화 상자는 자동으로 내용을 업데이트합니다.

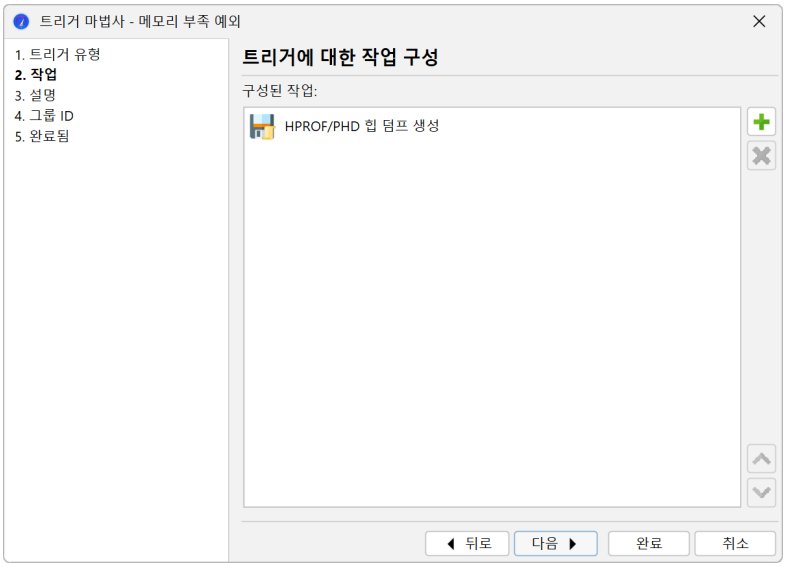
D 힙 워커 기능 상세

D.1 HPROF 및 PHD 힙 스냅샷

HotSpot JVM과 Android Runtime은 모두 HPROF 형식의 힙 스냅샷을 지원하며, IBM J9 JVM은 PHD 형식으로 이러한 스냅샷을 작성합니다. PHD 파일은 가비지 컬렉터 루트를 포함하지 않으므로, JProfiler는 클래스를 루트로 시뮬레이션합니다. PHD 파일로 클래스 로더 메모리 누수를 찾는 것은 어려울 수 있습니다.

네이티브 힙 스냅샷은 프로파일링 에이전트 없이 저장할 수 있으며, 일반적인 API의 제약 없이 저장되기 때문에 JProfiler 힙 스냅샷보다 오버헤드가 낮습니다. 반면에, 네이티브 힙 스냅샷은 JProfiler 힙 스냅샷보다 기능이 적습니다. 예를 들어, 할당 녹화 정보가 제공되지 않으므로 객체가 어디에 할당되었는지 볼 수 없습니다. HPROF 및 PHD 스냅샷은 세션->스냅샷 열기에서 JProfiler 스냅샷을 여는 것처럼 JProfiler에서 열 수 있습니다. 힙 워커만 사용할 수 있으며, 다른 모든 섹션은 회색으로 표시됩니다.

라이브 세션에서는 프로파일링->HPROF/PHD 힙 스냅샷 저장을 호출하여 HPROF/PHD 힙 스냅샷을 생성하고 열 수 있습니다. 오프라인 프로파일링 [p. 122]의 경우 "HPROF 힙 덤프 생성" 트리거 액션이 있습니다. 이는 일반적으로 "메모리 부족 예외" 트리거와 함께 사용되어 `OutOfMemoryError`가 발생할 때 HPROF 스냅샷을 저장합니다.



이는 VM 매개변수 [(1)] 에 해당합니다.

```
-XX:+HeapDumpOnOutOfMemoryError
```

이는 HotSpot JVM에서 지원됩니다.

실행 중인 시스템에서 HPROF 힙 덤프를 추출하는 또 다른 방법은 JRE의 일부인 명령줄 도구 `jmap` 을 사용하는 것입니다. 그 호출 구문은

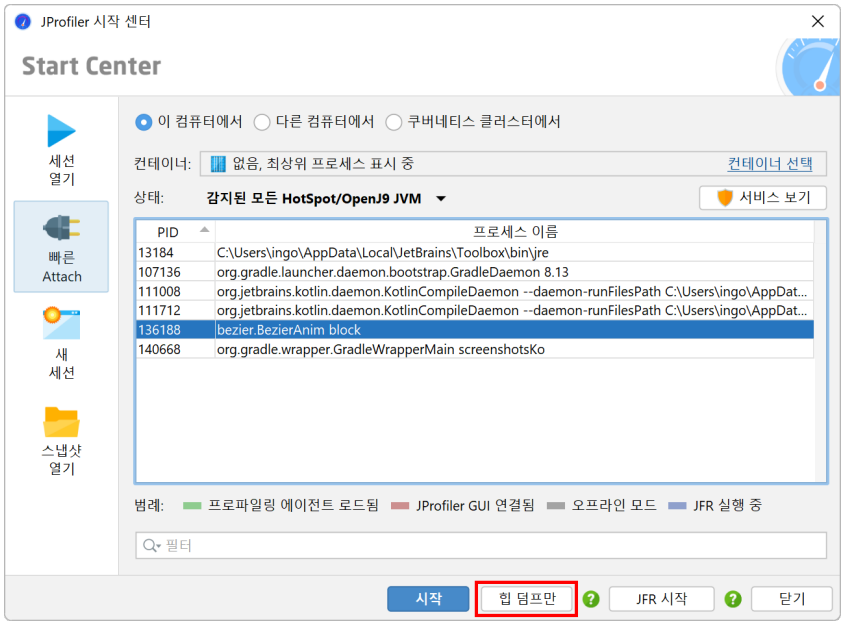
```
jmap -dump:live,format=b,file=<filename> <PID>
```

기억하기 어렵고 PID를 먼저 찾기 위해 `jps` 실행 파일을 사용해야 합니다. JProfiler는 대화형 명령줄 실행 파일 `bin/jpdump` 을 제공하여 훨씬 더 편리합니다. 프로세스를 선택할 수 있으며, Windows에서 서비스로 실

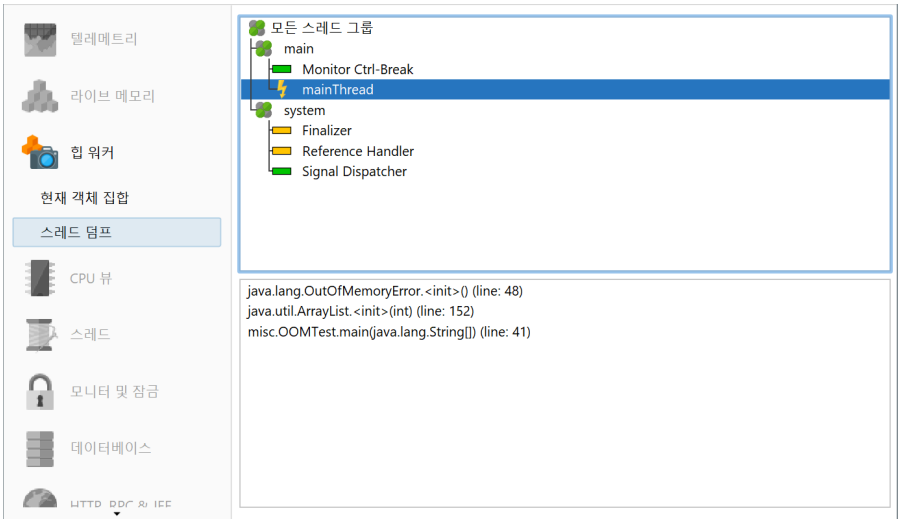
[(1)] http://docs.oracle.com/javase/9/troubleshoot/command-line-options1.htm#JSTGD592

행 중인 프로세스에 연결할 수 있고, 혼합된 32비트/64비트 JVM에서도 문제가 없으며, HPROF 스냅샷 파일을 자동으로 번호 매깁니다. 더 많은 정보를 얻으려면 `-help` 옵션으로 실행하십시오.

프로파일링 에이전트를 로드하지 않고 HPROF 힙 스냅샷을 찍는 것도 JProfiler GUI에서 지원됩니다. 프로세스에 로컬 또는 원격으로 attach할 때, 항상 HPROF 힙 스냅샷을 찍을 수 있는 가능성이 있습니다.



HPROF 스냅샷은 스레드 덤프를 포함할 수 있습니다. HPROF 스냅샷이 `OutOfMemoryError`의 결과로 저장된 경우, 스레드 덤프는 오류 당시 애플리케이션의 어느 부분이 활성화되었는지를 전달할 수 있습니다. 오류를 트리거한 스레드는 특별한 아이콘으로 표시됩니다.



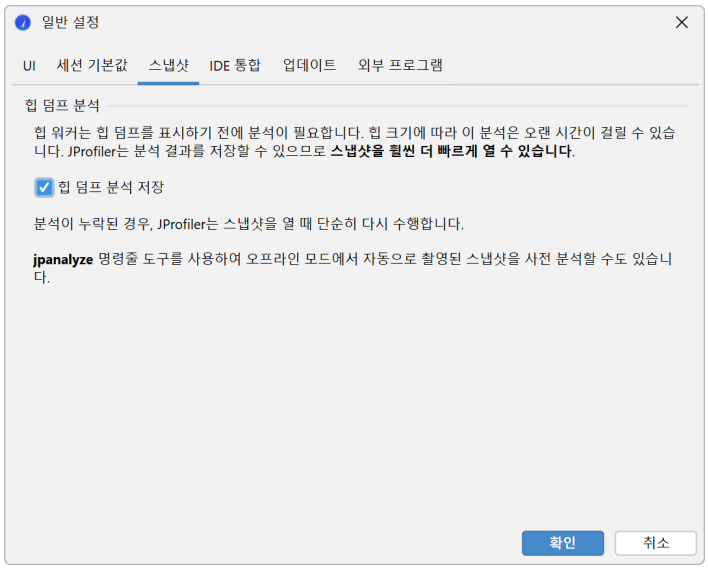
D.2 힙 워커에서 오버헤드 최소화하기

작은 힙의 경우, 힙 스냅샷을 찍는 데 몇 초가 걸리지만, 매우 큰 힙의 경우 이는 시간이 오래 걸릴 수 있습니다. 물리적 메모리가 부족하면 계산이 훨씬 느려질 수 있습니다. 예를 들어, JVM이 50 GB 힙을 가지고 있고 로컬 머신에서 5 GB의 여유 물리 메모리만으로 힙 덤프를 분석하는 경우, JProfiler는 특정 인덱스를 메모리에 보관할 수 없으며 처리 시간이 불균형하게 증가합니다.

JProfiler는 주로 힙 분석을 위해 네이티브 메모리를 사용하기 때문에, `-Xmx` 값을 `bin/jprofiler.vmoptions` 파일에서 증가시키는 것은 `OutOfMemoryError`가 발생했고 JProfiler가 그러한 수정을 지시한 경우가 아니라면 권장되지 않습니다. 네이티브 메모리는 사용 가능할 경우 자동으로 사용됩니다. 분석이 완료되고 내부 데이터베이스가 구축된 후에는 네이티브 메모리가 해제됩니다.

라이브 스냅샷의 경우, 힙 덤프를 찍은 직후에 분석이 계산됩니다. 스냅샷을 저장할 때, 분석은 스냅샷 파일 옆에 `.analysis` 접미사가 있는 디렉토리에 저장됩니다. 스냅샷 파일을 열면 힙 워커가 매우 빠르게 사용 가능합니다. `.analysis` 디렉토리를 삭제하면 스냅샷을 열 때 다시 계산이 수행되므로, 스냅샷을 다른 사람에게 보낼 때 분석 디렉토리를 함께 보낼 필요는 없습니다.

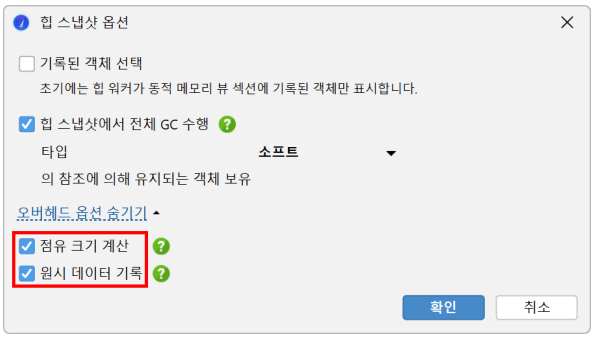
디스크 메모리를 절약하거나 생성된 `.analysis` 디렉토리가 불편한 경우, 일반 설정에서 그 생성을 비활성화할 수 있습니다.



HPROF 스냅샷과 오프라인 프로파일링 [p. 122]으로 저장된 JProfiler 스냅샷은 옆에 `.analysis` 디렉토리가 없습니다. 이는 분석이 프로파일링 에이전트가 아닌 JProfiler UI에 의해 수행되기 때문입니다. 이러한 스냅샷을 열 때 계산을 기다리고 싶지 않다면, `jpanalyze` 명령줄 실행 파일을 사용하여 스냅샷을 사전 분석 [p. 240]할 수 있습니다.

스냅샷을 쓰기 가능한 디렉토리에서 여는 것이 좋습니다. 분석이 없는 스냅샷을 열 때, 디렉토리가 쓰기 가능하지 않으면 임시 위치가 분석에 사용됩니다. 그런 경우 계산은 스냅샷을 열 때마다 반복되어야 합니다.

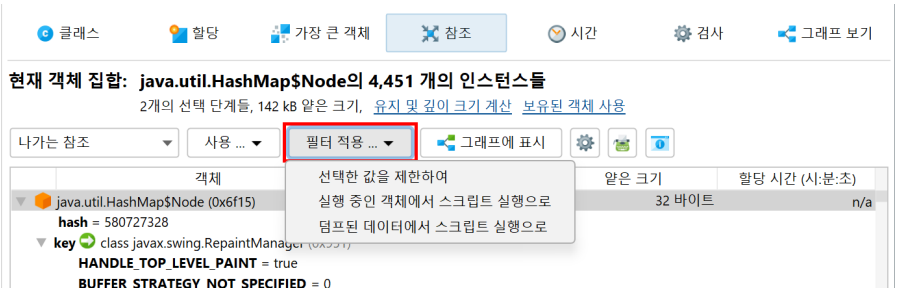
분석의 큰 부분은 유지된 크기의 계산입니다. 처리 시간이 너무 길고 유지된 크기가 필요하지 않다면, 힙 워커 옵션 대화 상자의 오버헤드 옵션에서 그 계산을 비활성화할 수 있습니다. 유지된 크기 외에도 "가장 큰 객체" 뷰도 사용할 수 없게 됩니다. 원시 데이터를 기록하지 않으면 힙 스냅샷이 작아지지만, 참조 뷰에서 이를 볼 수 없게 됩니다. 파일 선택 대화 상자에서 분석 사용자 정의를 선택하면 스냅샷을 열 때 동일한 옵션이 제공됩니다.


힙 스냅샷 옵션
기록된 객체 선택
초기에는 힙 워커가 동적 메모리 뷰 섹션에 기록된 객체만 표시합니다.
힙 스냅샷에서 전체 GC 수행
타입
소프트
의 참조에 의해 유지되는 객체 보유
오버헤드 옵션 숨기기
점유 크기 계산
원시 데이터 기록
확인
취소

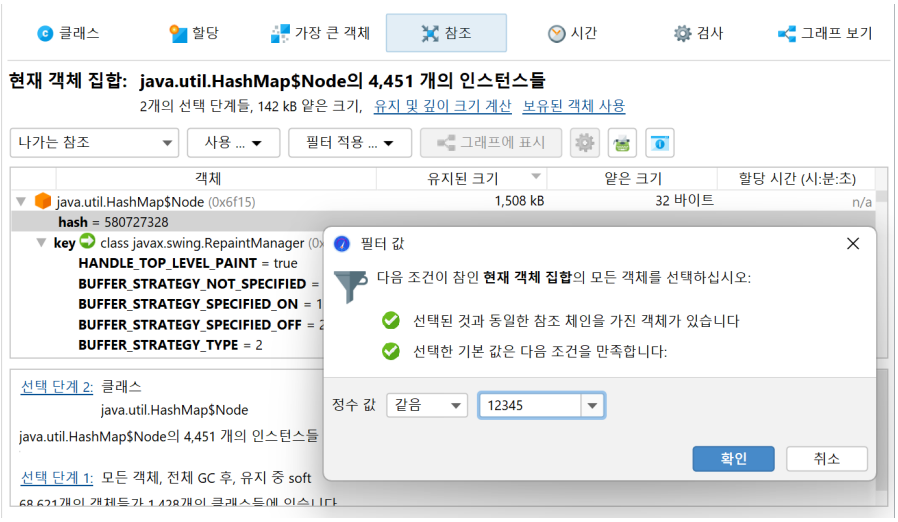
D.3 필터 및 라이브 상호작용

힙 워커에서 관심 있는 객체를 찾을 때, 동일한 클래스의 인스턴스가 너무 많은 객체 집합에 도달하는 경우가 많습니다. 특정 초점에 따라 객체 집합을 더 줄이기 위해 선택 기준은 해당 속성이나 참조를 포함할 수 있습니다. 예를 들어, 특정 속성을 포함하는 HTTP 세션 객체에 관심이 있을 수 있습니다. 힙 워커의 병합된 나가는 참조 뷰에서 전체 객체 집합에 대한 참조 체인을 포함하는 선택 단계를 수행할 수 있습니다.

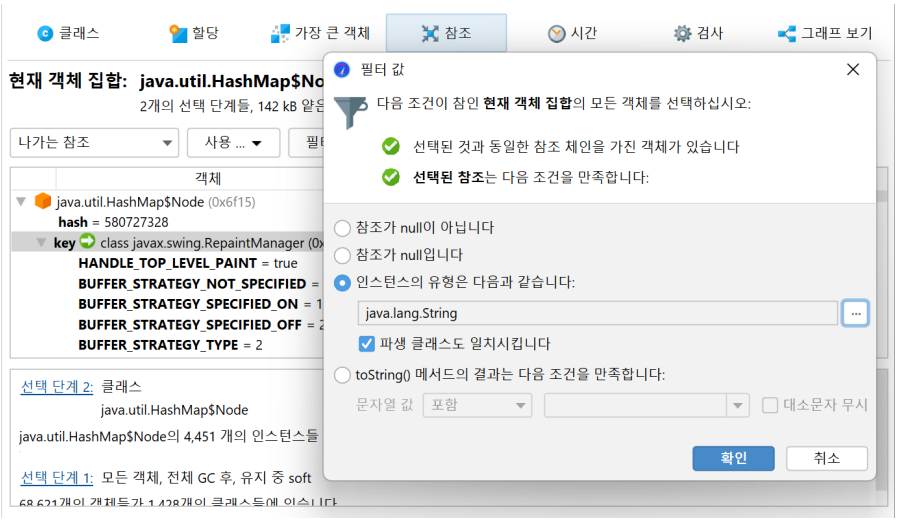
그러나 개별 객체를 볼 수 있는 나가는 참조 뷰에서는 참조와 원시 필드를 제한하는 선택 단계를 수행할 수 있는 훨씬 더 강력한 기능을 제공합니다.



최상위 객체, 원시 값 또는 나가는 참조 뷰의 참조를 선택하면 필터 적용->선택한 값 제한으로 작업이 활성화됩니다. 선택에 따라 필터 값 대화 상자는 다양한 옵션을 제공합니다. 어떤 옵션을 구성하든 새 객체 집합의 객체는 선택한 것과 같은 나가는 참조 체인을 가져야 한다는 제약 조건을 항상 암시적으로 추가합니다. 필터는 항상 현재 객체 집합을 더 작은 집합으로 제한하여 최상위 객체에서 작동합니다.

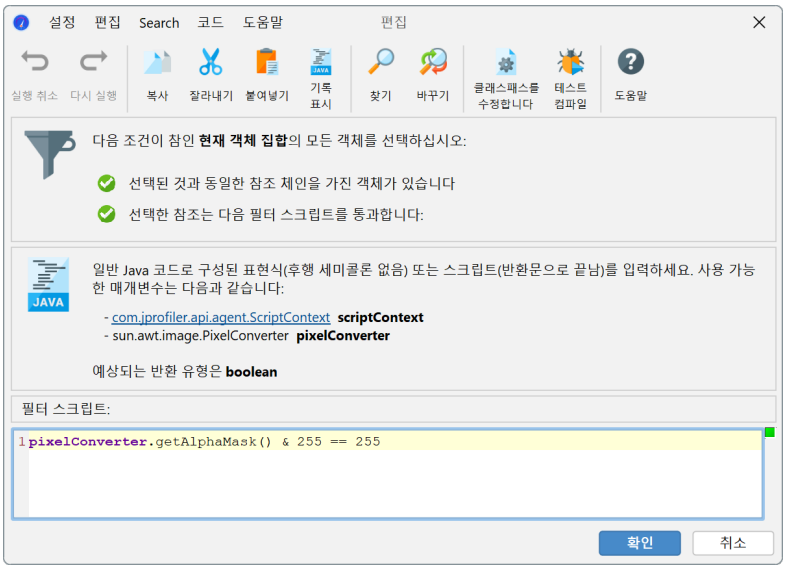


원시 값 제한은 HPROF 및 JProfiler 힙 스냅샷 모두에서 작동합니다. 참조 유형의 경우, JProfiler에 null이 아닌 값, null 값 및 선택한 클래스의 값을 필터링하도록 요청할 수 있습니다. `toString()` 메소드의 결과로 필터링하는 것은 `java.lang.String` 및 `java.lang.Class` 객체를 제외하고는 라이브 세션에서만 사용할 수 있으며, JProfiler는 이를 스스로 파악할 수 있습니다.



가장 강력한 필터 유형은 코드 스니펫을 사용하는 것입니다. 객체를 필터링하는 두 가지 근본적으로 다른 방법이 있습니다:

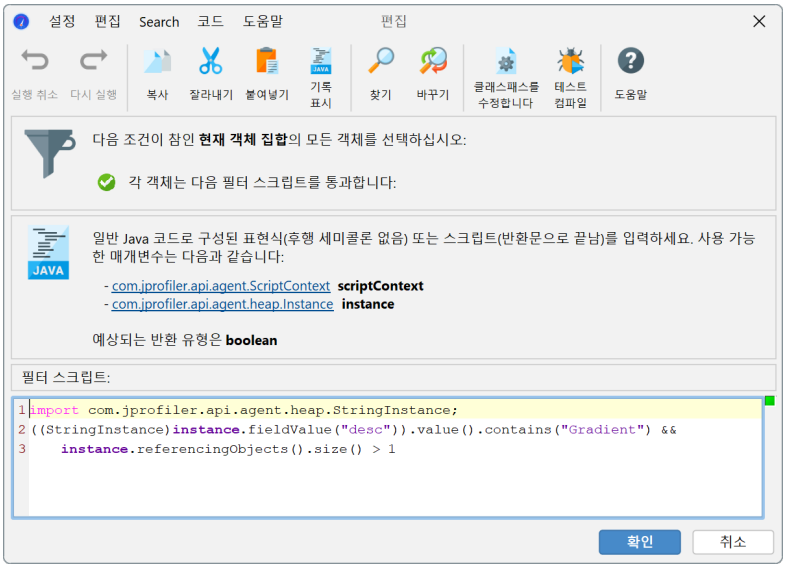
라이브 세션에서는 JProfiler가 프로파일된 JVM에서 필터링 스크립트를 실행하고 실제 인스턴스를 스크립트에 전달할 수 있습니다. 라이브 객체에서 스크립트를 실행하여 필터 적용에 의해 표시되는 스크립트 편집기에서 속성에 직접 액세스하고 부울 반환 값이 인스턴스를 현재 객체 집합에 유지할지 여부를 결정하는 표현식이나 스크립트를 작성할 수 있습니다.



당연히 이 기능은 라이브 세션에서만 작동할 수 있습니다. JProfiler가 라이브 객체에 액세스해야 하기 때문입니다. 고려해야 할 또 다른 요소는 힙 스냅샷이 찍힌 후 객체가 가비지 수집되었을 수 있다는 것입니다. 그런 경우, 코드 스니펫 필터가 실행될 때 해당 객체는 새 객체 집합에 포함되지 않습니다.

HPROF 및 PDH 스냅샷을 포함하여 스냅샷에서도 작동하는 두 번째 옵션은 덤프된 데이터에서 스크립트를 실행하여 필터 적용 작업입니다. 각 인스턴스는 `com.jprofiler.api.agent.heap.HeapObject`의 인스턴스로 스크립트에 전달됩니다. 해당되는 경우 매개변수를 다운캐스트할 수 있는 여러 하위 인터페이스가 있습니다. 자세한 내용은 Javadoc를 참조하십시오. 예를 들어, 객체가 객체 인스턴스인 경우 `com.jprofiler.api.agent.heap.Instance` 인터페이스를 사용할 수 있으며 필드 값에 액세스할 수 있습니다. 스크립트가 최상위 객체에서 작동하고 현재 객체 집합의 모든 객체가 동일한 유형인 경우 스크립트 매개변수는 자동으로 적절한 하위 유형을 갖게 됩니다.

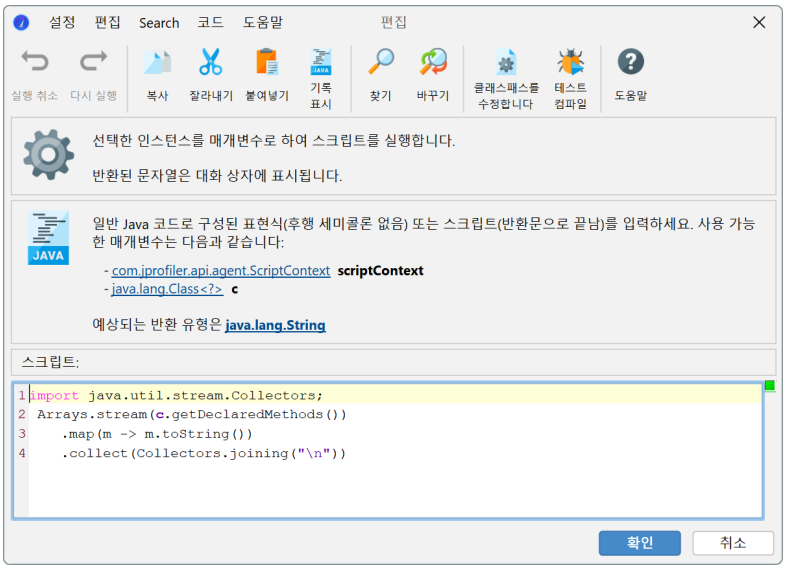
이러한 필터 스크립트에서는 `HeapObject` 매개변수의 메소드를 통해 모든 들어오는 및 나가는 참조에 액세스할 수도 있습니다.



PHD 스냅샷에는 필드 정보가 포함되어 있지 않으므로 모든 인스턴스는 `com.jprofiler.api.agent.heap.HeapObject` 또는 `com.jprofiler.api.agent.heap.ClassObject`로 전달되며, 필드 값은 `referencedObjects()` 메소드를 통해서만 액세스할 수 있습니다.

필터 외에도 개별 객체와 상호작용하기 위한 나가는 참조 뷰에는 두 가지 다른 기능이 있습니다: toString() 값 표시 작업은 현재 뷰에 보이는 모든 객체에 대해 `toString()` 메소드를 호출하고 이를 참조 노드에 직접 표시합니다. 노드는 매우 길어질 수 있으며 텍스트가 잘릴 수 있습니다. 컨텍스트 메뉴에서 노드 세부정보 표시 작업을 사용하면 전체 텍스트를 볼 수 있습니다.

`toString()` 메소드를 호출하는 것보다 객체에서 정보를 얻는 더 일반적인 방법은 문자열을 반환하는 임의의 스크립트를 실행하는 것입니다. toString() 값 표시 작업 옆의 스크립트 실행 작업을 통해 최상위 객체나 참조가 선택된 경우 이를 수행할 수 있습니다. 스크립트 실행 결과는 별도의 대화 상자에 표시됩니다.

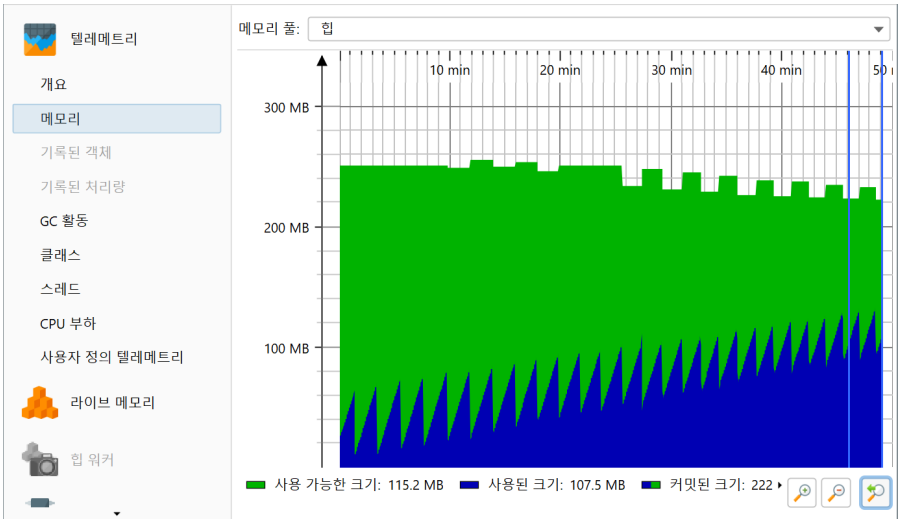

설정
편집
Search
코드
도움말
편집
실행 취소
다시 실행
복사
잘라내기
붙여넣기
기록 표시
찾기
바꾸기
클래스패스를 수정합니다
테스트 컴파일
도움말
선택한 인스턴스를 매개변수로 하여 스크립트를 실행합니다.
반환된 문자열은 대화 상자에 표시됩니다.
JAVA
일반 Java 코드로 구성된 표현식(후행 세미콜론 없음) 또는 스크립트(반환문으로 끝남)를 입력하세요. 사용 가능한 매개변수는 다음과 같습니다:
- com.jprofiler.api.agent.ScriptContext scriptContext
- java.lang.Class<?> c
예상되는 반환 유형은 java.lang.String
스크립트:
1 import java.util.stream.Collectors;
2 Arrays.stream(c.getDeclaredMethods())
3 .map(m -> m.toString())
4 .collect(Collectors.joining("\n"))
확인
취소

D.4 메모리 누수 찾기

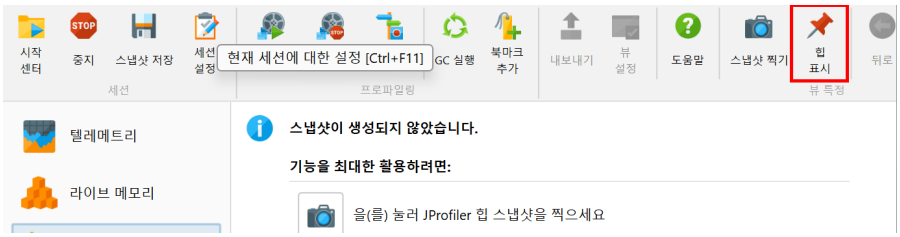
일반적인 메모리 사용과 메모리 누수를 구분하는 것은 종종 간단하지 않습니다. 그러나 과도한 메모리 사용과 메모리 누수는 동일한 증상을 가지므로 동일한 방식으로 분석할 수 있습니다. 분석은 두 단계로 진행됩니다: 의심스러운 객체를 찾고, 왜 그 객체들이 여전히 힙에 있는지 알아내는 것입니다.

새로운 객체 찾기

메모리 누수가 있는 애플리케이션이 실행 중일 때, 시간이 지남에 따라 더 많은 메모리를 소비합니다. 메모리 사용량의 증가를 감지하는 가장 좋은 방법은 VM 텔레메트리와 "모든 객체" 및 "녹화된 객체" 뷰의 차이 기능 [p. 68]을 사용하는 것입니다. 이러한 뷰를 통해 문제가 있는지, 그리고 그 심각성을 판단할 수 있습니다. 때로는 호출 히스토그램 테이블의 차이 열이 이미 문제에 대한 아이디어를 제공합니다.



메모리 누수에 대한 더 깊은 분석은 힙 워커의 기능을 필요로 합니다. 특정 사용 사례 주위의 메모리 누수를 자세히 조사하려면 "힙 표시" 기능 [p. 77]이 가장 적합합니다. 이를 통해 특정 이전 시점 이후 힙에 남아 있는 새로운 객체를 식별할 수 있습니다. 이러한 객체에 대해서는 여전히 힙에 정당하게 있는지, 아니면 객체가 더 이상 목적을 제공하지 않음에도 불구하고 잘못된 참조가 그것들을 활성 상태로 유지하는지 확인해야 합니다.



관심 있는 객체 집합을 격리하는 또 다른 방법은 할당 녹화를 통해서입니다. 힙 스냅샷을 찍을 때, 모든 녹화된 객체를 표시하는 옵션이 있습니다. 그러나 할당 녹화를 특정 사용 사례에만 제한하고 싶지 않을 수도 있습니다. 또한, 할당 녹화는 높은 오버헤드를 가지므로 힙 표시 작업은 상대적으로 훨씬 작은 영향을 미칠 것입니다. 마지막으로, 힙 워커는 힙을 표시한 경우 새로 사용 및 오래된 사용 하이퍼링크를 사용하여 선택 단계에서 오래된 객체와 새로운 객체를 선택할 수 있게 해줍니다.

가장 큰 객체 분석

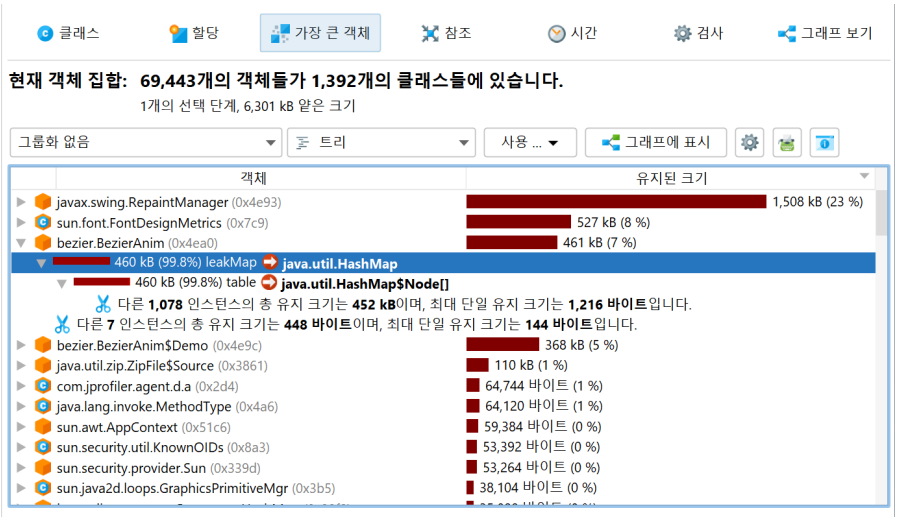
메모리 누수가 사용 가능한 힙을 채우면, 프로파일된 애플리케이션의 다른 유형의 메모리 사용을 압도할 것입니다. 이 경우, 새로운 객체를 조사할 필요가 없으며, 단순히 가장 중요한 객체를 분석하면 됩니다.

메모리 누수는 매우 느린 속도를 가질 수 있으며 오랫동안 지배적이지 않을 수 있습니다. 메모리 누수가 눈에 띄게 될 때까지 프로파일링하는 것은 실용적이지 않을 수 있습니다. `OutOfMemoryError`가 발생할 때 자동으로 HPROF 스냅샷을 저장하는 [p. 195] JVM의 내장 기능을 사용하면, 메모리 누수가 일반적인 메모리 소비보다 더 중요한 스냅샷을 얻을 수 있습니다. 사실, 항상 추가하는 것이 좋습니다.

```
-XX:+HeapDumpOnOutOfMemoryError
```

VM 매개변수나 프로덕션 시스템에 추가하여 개발 환경에서 재현하기 어려운 메모리 누수를 분석할 수 있는 방법을 제공합니다.

메모리 누수가 지배적이라면, 힙 워커의 "가장 큰 객체" 뷰의 상위 객체는 실수로 유지된 메모리를 포함할 것입니다. 가장 큰 객체 자체는 정당한 객체일 수 있지만, 그들의 지배자 트리를 열면 누수된 객체로 이어질 것입니다. 간단한 상황에서는 힙의 대부분을 포함하는 단일 객체가 있을 것입니다. 예를 들어, 맵이 객체를 캐시하는 데 사용되고 그 캐시가 결코 지워지지 않는다면, 그 맵은 가장 큰 객체의 지배자 트리에 나타날 것입니다.

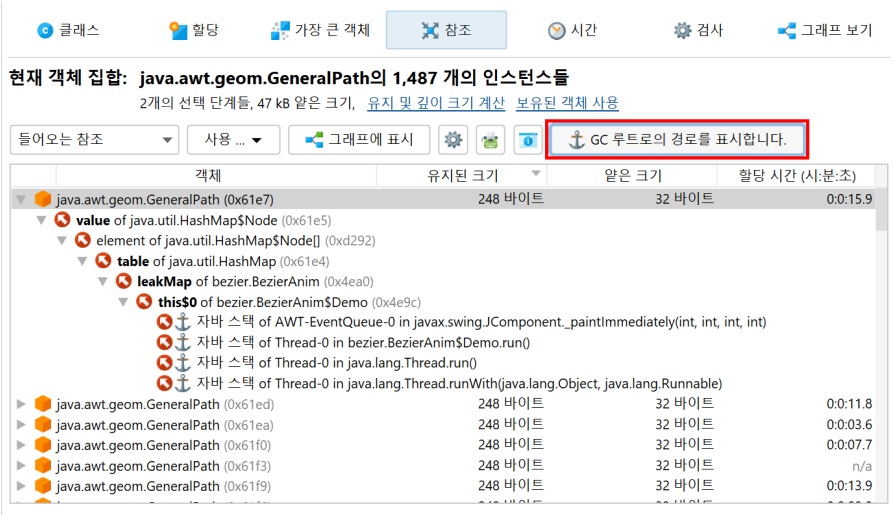


가비지 컬렉터 루트에서 강한 참조 체인 찾기

객체는 강하게 참조될 때만 문제가 될 수 있습니다. "강하게 참조된다"는 것은 가비지 컬렉터 루트에서 객체로의 참조 체인이 적어도 하나 있다는 것을 의미합니다. "가비지 컬렉터" 루트(줄여서 GC 루트)는 가비지 컬렉터가 알고 있는 JVM의 특별한 참조입니다.

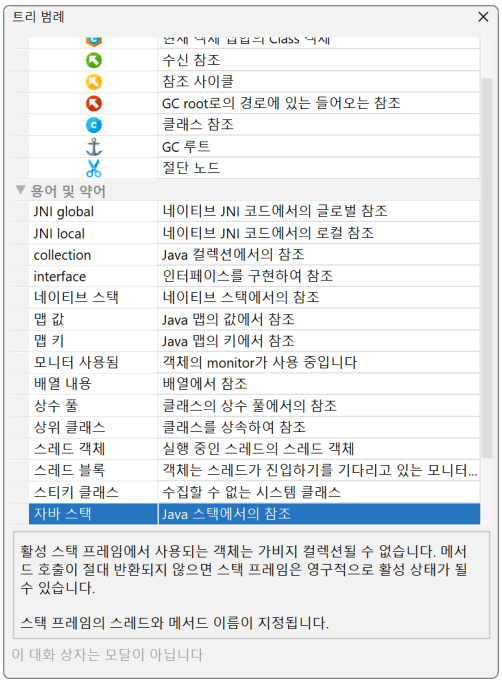
GC 루트에서 참조 체인을 찾으려면, "들어오는 참조" 뷰나 힙 워커 그래프에서 GC 루트로의 경로 표시 작업을 사용할 수 있습니다. 이러한 참조 체인은 실제로 매우 길 수 있으므로 일반적으로 "들어오는 참조" 뷰에서 더 쉽

게 해석할 수 있습니다. 참조는 아래에서 상위 수준의 객체로 향합니다. 검색 결과인 참조 체인만 확장되며, 동일한 수준의 다른 참조는 노드를 닫고 다시 열거나 컨텍스트 메뉴에서 모든 들어오는 참조 표시 작업을 호출할 때까지 보이지 않습니다.



참조 노드에서 사용된 GC 루트 유형 및 기타 용어에 대한 설명을 얻으려면 트리 범례를 사용하십시오.

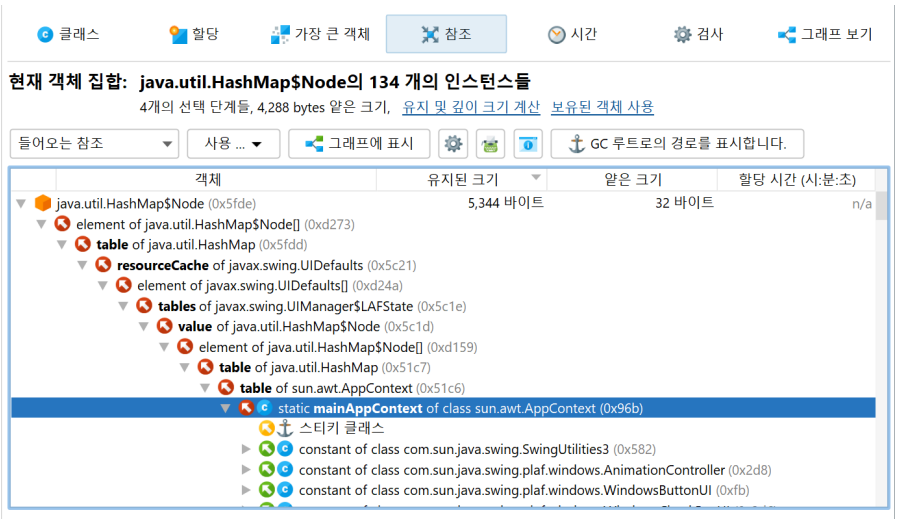
트리에서 노드를 선택하면 비모달 트리 범례가 선택된 노드에서 사용된 모든 아이콘과 용어를 강조 표시합니다. 대화 상자의 행을 클릭하면 하단에 설명이 표시됩니다.



중요한 유형의 가비지 컬렉터 루트는 스택에서의 참조, JNI를 통한 네이티브 코드에 의해 생성된 참조, 현재 사용 중인 라이브 스레드 및 객체 모니터와 같은 리소스입니다. 또한, JVM은 중요한 시스템을 유지하기 위해 몇 가지 "끈적한" 참조를 추가합니다.

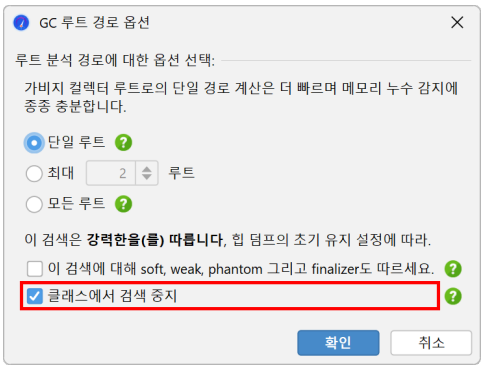
클래스와 클래스로더는 특별한 순환 참조 체계를 가지고 있습니다. 클래스는 클래스로더와 함께 가비지 컬렉션됩니다.

- 해당 클래스로더에 의해 로드된 클래스 중 어떤 것도 라이브 인스턴스를 가지고 있지 않을 때
- 클래스로더 자체가 클래스에 의해서만 참조되지 않을 때
- `java.lang.Class` 객체가 클래스로더의 컨텍스트 외부에서 참조되지 않을 때



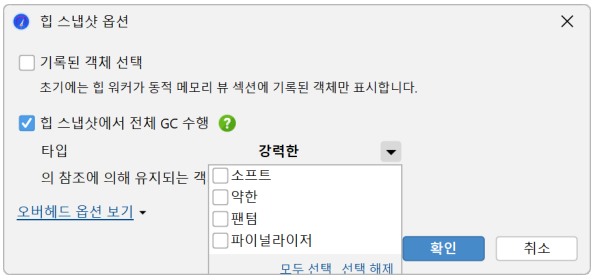
대부분의 경우, 클래스는 관심 있는 GC 루트로의 경로에서 마지막 단계입니다. 클래스 자체는 GC 루트가 아닙니다. 그러나 사용자 정의 클래스로더가 사용되지 않는 모든 상황에서, 그것들을 GC 루트로 취급하는 것이 적

절합니다. 이는 가비지 컬렉터 루트를 검색할 때 JProfiler의 기본 모드이지만, 루트 옵션 대화 상자에서 이를 변경할 수 있습니다.

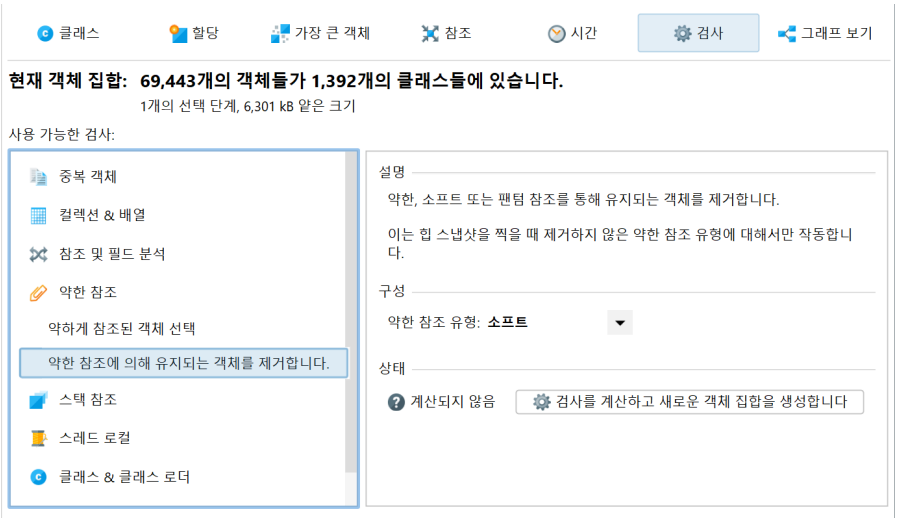


GC 루트로의 최단 경로를 해석하는 데 문제가 있는 경우, 추가 경로를 검색할 수 있습니다. 일반적으로 모든 경로를 GC 루트로 검색하는 것은 권장되지 않습니다. 왜냐하면 많은 경로를 생성할 수 있기 때문입니다.

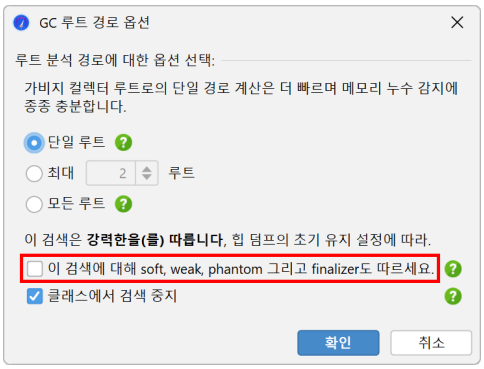
라이브 메모리 뷰와는 달리, 힙 워커는 참조되지 않은 객체를 절대 표시하지 않습니다. 그러나 힙 워커는 강하게 참조된 객체만 표시하지 않을 수도 있습니다. 기본적으로 힙 워커는 약한 참조, 팬텀 참조 또는 파이널라이저 참조에 의해서만 참조된 객체를 제거하지만, 소프트 참조에 의해서만 참조된 객체는 유지합니다. 소프트 참조는 힙이 소진되지 않는 한 가비지 컬렉션되지 않으므로, 큰 힙 사용량을 설명할 수 없을 수 있기 때문에 포함됩니다. 힙 스냅샷을 찍을 때 표시되는 옵션 대화 상자에서 이 동작을 조정할 수 있습니다.



힙 워커에 약하게 참조된 객체가 있는 것은 디버깅 목적으로 흥미로울 수 있습니다. 나중에 약하게 참조된 객체를 제거하려면 "약한 참조에 의해 유지된 객체 제거" 검사를 사용할 수 있습니다.



GC 루트로의 경로를 검색할 때, 힙 워커 옵션 대화 상자에서 객체를 유지하기 위해 선택된 참조 유형이 고려됩니다. 그렇게 함으로써, GC 루트 검색 경로는 항상 객체가 힙 워커에 유지된 이유를 설명할 수 있습니다. GC 루트 검색 경로의 옵션 대화 상자에서 허용 가능한 참조 유형을 모든 약한 참조로 확장할 수 있습니다.



전체 객체 집합 제거

지금까지 우리는 단일 객체만 살펴보았습니다. 종종 메모리 누수의 일부인 동일한 유형의 많은 객체가 있을 것입니다. 많은 경우, 단일 객체의 분석은 현재 객체 집합의 다른 객체에도 유효할 것입니다. 관심 있는 객체가 다양한 방식으로 참조되는 일반적인 경우에는 "병합된 지배 참조" 뷰가 현재 객체 집합을 힙에 유지하는 참조를 찾는 데 도움이 될 것입니다.

지배 참조 트리의 각 노드는 해당 참조를 제거하면 현재 객체 집합에서 가비지 컬렉션에 적합한 객체 수를 알려줍니다. 여러 가비지 컬렉터 루트에 의해 참조된 객체는 지배적인 들어오는 참조를 가질 수 없으므로 뷰는 객체의 일부에만 도움이 되거나 비어 있을 수도 있습니다. 이 경우, 병합된 들어오는 참조 뷰를 사용하고 가비지 컬렉터 루트를 하나씩 제거해야 합니다.

E JDK Flight Recorder (JFR)

E.1 JDK Flight Recorder (JFR) 지원

JDK Flight Recorder (JFR) [(1)]는 시스템 수준의 다양한 이벤트를 기록하는 구조화된 로깅 도구입니다. 항공기의 블랙박스가 비행 데이터를 지속적으로 기록하여 사고 조사에 사용되는 것과 유사하게, JFR은 문제 진단에 사용하기 위해 JVM에서 이벤트 스트림을 지속적으로 기록합니다. 이 접근 방식의 장점은 사건 발생 전 시스템에 대한 시간 순서대로 상세한 정보를 캡처한다는 것입니다. JFR은 성능에 미치는 영향을 최소화하도록 설계되었으며, 프로덕션 환경에서 장기간 실행해도 안전합니다.

Java 17부터 JFR은 JProfiler의 데이터 소스 중 하나로 사용됩니다. JVM의 프로파일링 인터페이스를 사용하는 네이티브 에이전트 외에도, 프로파일링 컨텍스트에서 관심 있는 JVM의 고급 시스템이 있습니다. 하나는 JProfiler의 일부 텔레메트리에 데이터를 제공하는 MBean 시스템이고, 다른 하나는 가비지 컬렉터 프로브 [p. 112]에 사용되는 JFR입니다. 이를 위해 JFR과 상호작용할 필요는 없으며, JProfiler가 JFR 이벤트 스트리밍을 투명하게 처리합니다.

JProfiler에서의 JFR 통합

JProfiler는 JFR 녹화 [p. 210]를 완전히 통합하여, 로컬 머신이나 JFR 녹화가 구성되지 않은 원격 머신에서 실행 중인 JVM에서 데이터를 쉽게 캡처할 수 있습니다.

JProfiler UI에서 JFR 스냅샷을 열면, 사용 가능한 뷰와 섹션이 일반적인 프로파일링 세션과 다릅니다. UI의 중심은 이벤트 브라우저 [p. 214]입니다. JFR 뷰에 사용 가능한 다른 모든 뷰는 별도의 장에서 설명됩니다 [p. 221].

이벤트 유형을 설정하고 필터를 설정하며 분석을 보는 동안, JProfiler는 가끔 JFR 스냅샷 파일을 다시 스캔해야 할 수 있습니다. JFR 스냅샷 파일은 잠재적으로 매우 크며, 모든 데이터를 메모리에 보관하거나 모든 분석을 미리 계산하는 것은 실현 가능하지 않습니다. 이 때문에 네트워크 드라이브에서 JFR 스냅샷을 여는 것은 권장되지 않습니다.

매우 큰 JFR 스냅샷을 열 때, 파일 선택기에서 "분석 사용자 정의" 체크 박스를 클릭하고 분석에 필요하지 않은 이벤트 카테고리를 제외하여 스냅샷 처리 속도를 높이고 메모리 사용량을 줄일 수 있습니다. 사용 가능한 이벤트 카테고리는 단일 프로브와 뷰 섹션을 포함합니다. CPU 뷰, 메모리 뷰 및 텔레메트리 뷰에 대한 이벤트 유형은 선택 사항이 아니며 로드해야 합니다.

예를 들어, CPU 데이터에만 관심이 있는 경우 모든 프로브와 이벤트 브라우저를 제외할 수 있습니다. JProfiler는 가장 빠른 JFR 뷰어가 되는 것을 목표로 하며, 일반적인 JFR 스냅샷을 빠르게 엽니다. 그러나 JFR 녹화는 잠재적으로 무제한이며, 수십 기가바이트 크기의 스냅샷을 마주할 수 있어 열기 속도가 문제가 될 수 있습니다.

JFR 스냅샷의 스택 트레이스

JFR의 중요한 기능 중 하나는 특정 이벤트 유형에 대해 전체 스택 트레이스를 효율적으로 기록할 수 있는 기능입니다. 이러한 이벤트 유형에 대해서는 JFR 설정에서 스택 트레이스 기록을 토글할 수 있습니다. 특히 스레드와 관련된 많은 JVM 애플리케이션 이벤트 유형은 기본적으로 스택 트레이스 기록이 활성화되어 있습니다.

JFR은 고정된 깊이까지의 스택 트레이스만 수집하므로 긴 스택 트레이스는 잘립니다. 잘린 트레이스는 이해할 수 있는 호출 트리를 구축하는 데 적합하지 않으므로, 이러한 트레이스는 특별히 표시된 노드 아래에 표시됩니다.

```
-XX:FlightRecorderOptions=stackdepth=<nnnn>
```

VM 매개변수를 사용하여 JFR에서 수집된 트레이스의 크기를 늘리고 애플리케이션의 잘린 트레이스를 제거할 수 있습니다.

[(1)] https://en.wikipedia.org/wiki/JDK_Flight_Recorder

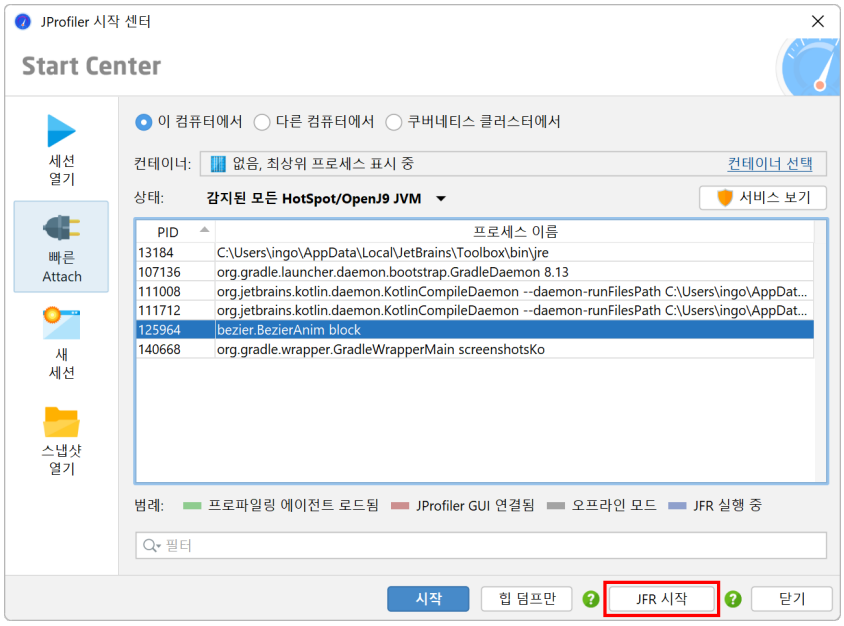
E.2 JProfiler로 JFR 스냅샷 녹화하기

JFR을 프로덕션 JVM에서 최소한의 오버헤드로 실행할 수 있고 프로파일링 인터페이스를 활성화할 필요가 없다는 이점 때문에, JProfiler는 UI에서 직접 JFR 녹화를 지원합니다. JFR을 프로그래밍 방식으로 시작하거나 명령줄에서 `-XX:StartFlightRecording` VM 매개변수를 추가하여 시작할 수 있지만, JProfiler는 이미 실행 중인 JVM에 대해 녹화를 시작하고 중지하는 데 도움을 줍니다.

JProfiler로 JVM에 attach할 때, 네이티브 프로파일링 에이전트를 로드하는 대신 JFR 녹화를 시작하고 중지하도록 선택할 수 있습니다. JProfiler의 광범위한 원격 연결 기능을 통해, 예를 들어 Docker나 Kubernetes 컨테이너에서 실행 중인 JVM에서 컨테이너를 수정할 필요 없이 JFR 녹화를 시작할 수 있습니다.

JFR 녹화 시작 및 중지

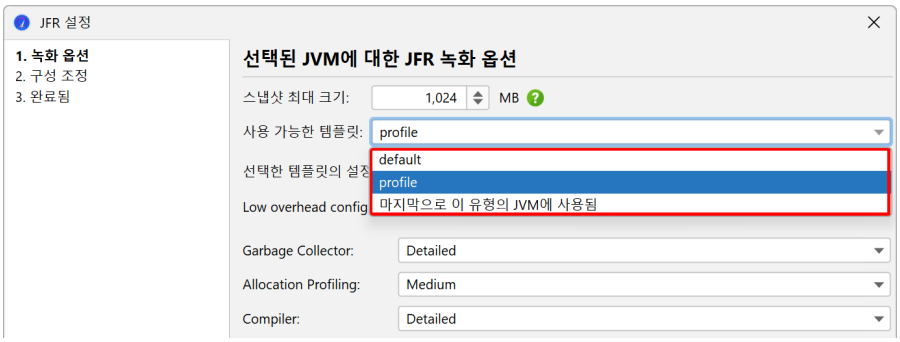
시작 센터의 "Quick attach" 탭에서 JVM을 선택하고 대화 상자 하단의 Start JFR 버튼을 클릭합니다. 로컬에서 실행 중인 JVM이 스크린샷에 표시되지만, 원격 JVM에 attach할 때도 동일한 버튼을 사용할 수 있습니다.



JFR 설정 마법사에서 선택한 프로세스가 사용하는 JRE의 `lib/jfr` 디렉토리에서 전송된 **이벤트 설정 템플릿** 중 하나를 선택할 수 있습니다. 기본적으로 "default"와 "profile"이라는 두 가지 템플릿이 있으며, "profile"은 더 많은 데이터를 기록하고 더 많은 오버헤드를 추가합니다. 해당 디렉토리에 다른 파일을 생성하면 마법사에서 해당 템플릿을 선택할 수 있습니다.

이러한 템플릿 파일에는 사용 가능한 이벤트와 중요한 **고급 설정**에 대한 구성 지침이 포함되어 있습니다. 각 고급 설정은 여러 다른 이벤트와 연결될 수 있습니다. 이 UI는 템플릿 파일의 내용에 따라 동적으로 생성됩니다. 다른 프로필 간 전환을 통해 다른 기본값을 확인할 수 있습니다. 이 UI에 포함되지 않은 더 많은 이벤트 유형이 있으며, 이는 다음 단계에서만 구성할 수 있습니다.

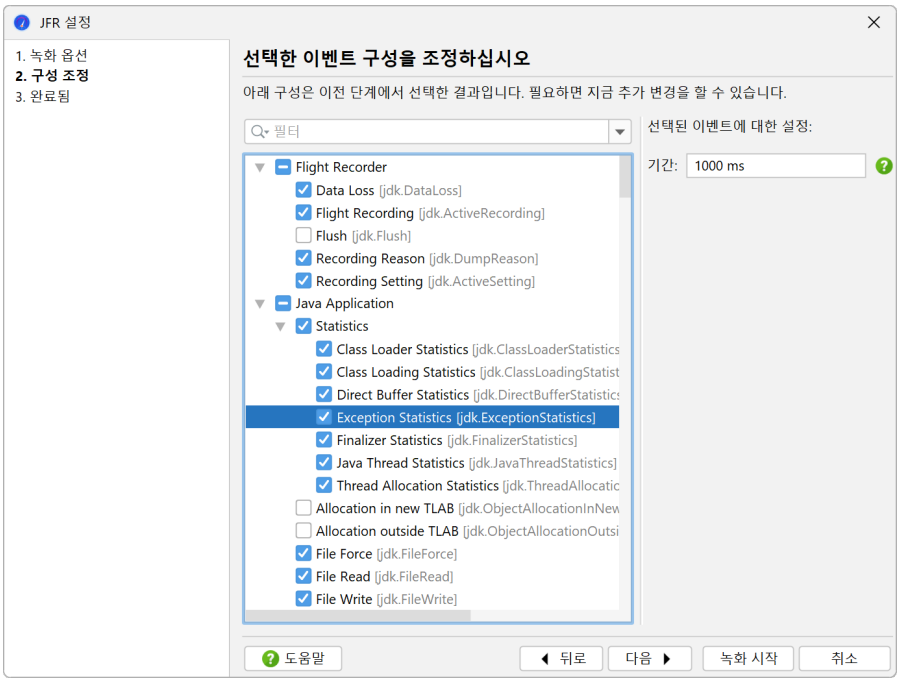
동일한 이벤트 유형 세트로 JVM에 대해 JFR 녹화를 이미 시작한 경우, JProfiler는 **마지막 설정**을 사용할 수 있는 옵션을 제공합니다.



해당 옵션을 선택하면 고급 녹화 설정은 사용할 수 없으며, 다음 단계로 진행하여 전체 구성을 보고 추가 변경을 할 수 있습니다.

마법사의 이 단계에서 또 다른 중요한 설정은 **최대 스냅샷 크기**입니다. JFR 녹화의 특성상 스냅샷의 크기는 매우 빠르게 증가할 수 있으며, 전체 하드 디스크를 채울 수 있습니다. 이를 방지하기 위해 최대 스냅샷 크기 제한은 과도한 저장소 사용을 방지합니다. 최대 크기에 도달하면, 오래된 이벤트는 삭제되고 새로운 이벤트는 계속 기록됩니다. 이 프로세스는 JFR의 자동 메커니즘입니다.

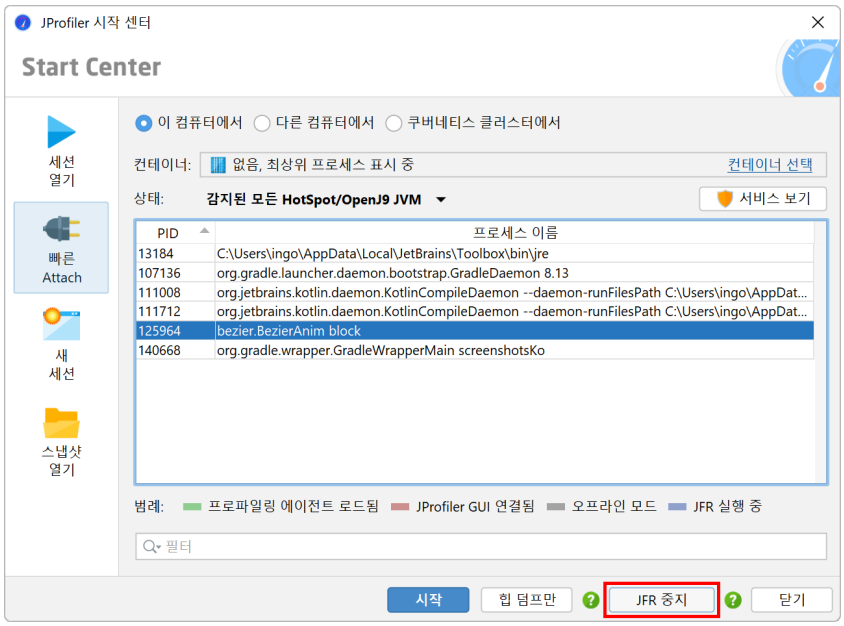
마법사의 다음 단계에서는 모든 이벤트 유형의 분류된 트리를 볼 수 있으며, 오른쪽에서 각 이벤트에 대한 추가 구성을 할 수 있습니다.



이벤트에는 **기간**, **임계값** 및 각 이벤트에 대해 **스택 추적**을 기록할지 여부에 대한 플래그 설정이 있을 수 있습니다. 기간과 임계값 모두 시간 단위로 설정되며, down 키를 눌러 사용 가능한 단위에 대한 자동 완성 팝업을 얻을 수 있습니다. 기간은 또한 자동 완성 팝업에서 사용할 수 있는 특별한 값 "everyChunk", "beginChunk" 및 "endChunk"를 지원합니다. "chunk"는 JFR 녹화의 일부로, 연속적인 이벤트 데이터와 메타데이터 세트를 보유하며 녹화에서 기본 저장 및 데이터 전송 단위로 기능합니다.

트리에서 더 많은 이벤트가 선택될수록 더 많은 데이터가 기록됩니다. 일부 이벤트 유형은 대량의 데이터를 생성하는 반면, 일부는 적은 수의 이벤트만 생성합니다.

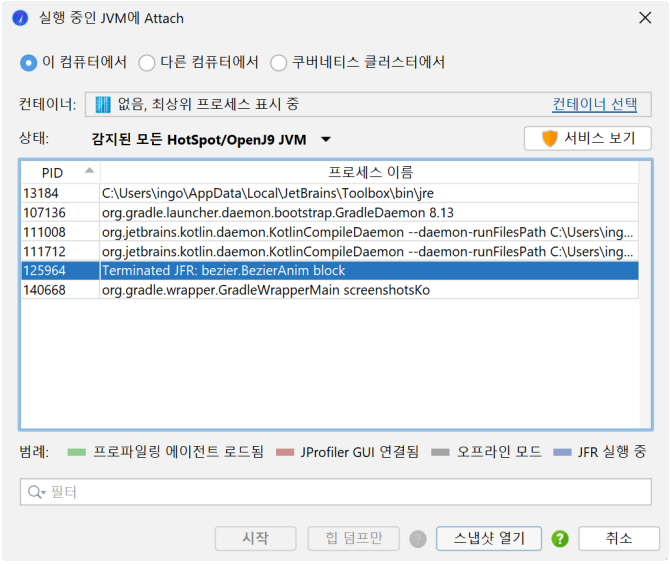
전체 프로파일링 모드나 "Heap dump only" 모드와 달리, JFR 스냅샷을 시작하면 UI에서 즉시 데이터를 볼 수 있는 것이 아니라, 선택되지 않은 경우 JVM의 배경색만 변경되어 JProfiler가 녹화를 시작했음을 알 수 있습니다. JVM이 선택되면, 하단의 JFR 버튼 텍스트가 녹화가 중지될 것임을 보여줍니다.



JProfiler에서 시작한 JFR 녹화를 중지하면, JFR 스냅샷이 전송되어 JProfiler에서 열립니다. **스냅샷은 임시**이며 창을 닫으면 삭제됩니다. 스냅샷을 영구적인 위치에 저장하려면, 툴바의 "Save snapshot" 작업을 사용하세요.

JFR 녹화가 있는 종료된 JVM

JFR의 한 가지 언급된 사용 사례는 충돌 전의 순간을 조사하는 것입니다. 이 경우, JVM은 더 이상 JVM 테이블에 표시되지 않아 JFR 녹화를 중지하고 JFR 스냅샷을 열 수 없습니다. JProfiler에서 JFR 녹화를 시작하고 녹화를 중지하기 전에 JVM이 종료되면, "Terminated JFR:"로 시작하는 특별한 항목이 JVM 테이블에 추가됩니다. 해당 항목을 더블 클릭하거나 "JFR" 버튼을 사용하여 JFR 스냅샷을 열 수 있습니다.



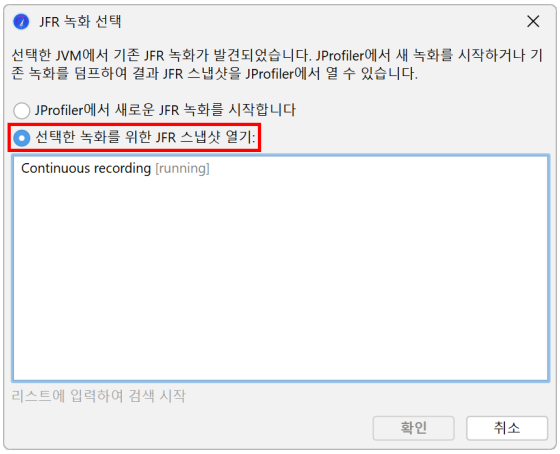
해당 항목을 열면 목록에서 제거됩니다. 수동으로 중지된 녹화와 마찬가지로, 열린 JFR 스냅샷은 임시이며 나중에 분석을 위해 보관하려면 저장해야 합니다.

외부에서 시작된 JFR 녹화 표시

위의 예에서는 JProfiler에서 JFR 녹화를 시작하고 중지했습니다. JProfiler 외부에서 시작된 JFR 녹화도 표시할 수 있습니다. 연속 JFR 녹화는 VM 매개변수로 쉽게 시작할 수 있습니다.

```
"-XX:StartFlightRecording=maxsize=500m=filename=$TEMP/myapp.jfr,name=Continuous
recording"
```

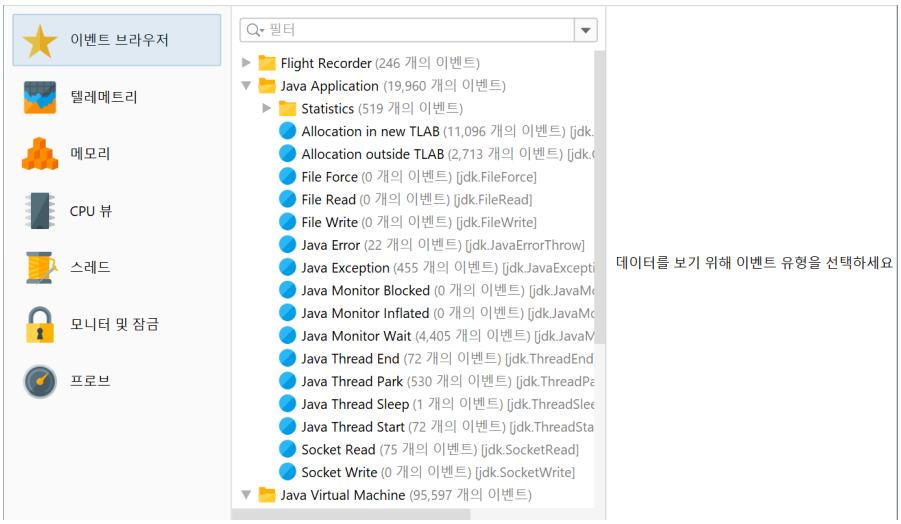
JVM 테이블에서 JFR 녹화가 실행 중임을 나타내는 특별한 배경색은 JProfiler에서 시작된 JFR 녹화에만 해당됩니다. 다른 방법으로 JFR 녹화가 시작된 JVM에 연결하면 다른 대화 상자가 표시됩니다.



이제 JProfiler에서 새 녹화를 시작하거나 기존 녹화를 덤프하여 결과 JFR 스냅샷을 JProfiler에서 표시할 수 있습니다. 외부에서 시작된 JFR 녹화는 별도의 라이프 사이클을 가지며 JProfiler에 의해 중지되지 않습니다.

E.3 JFR 이벤트 브라우저

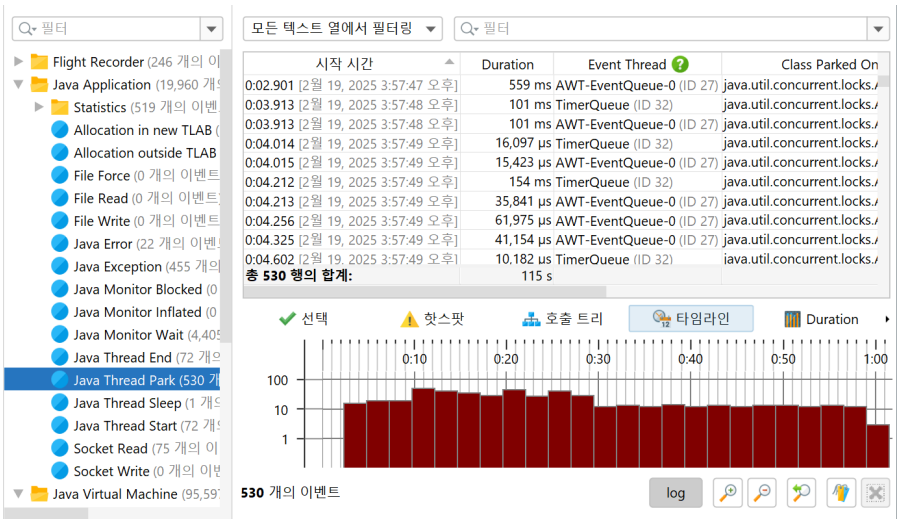
이벤트 브라우저는 JFR 스냅샷에 기록된 모든 데이터를 보여줍니다.



JFR은 이벤트 유형을 계층적 카테고리로 구성하여 이벤트 브라우저의 왼쪽에 트리를 형성합니다. 단일 이벤트 유형을 선택하여 기록된 이벤트를 표시할 수 있습니다. 기본적으로 JProfiler는 모든 등록된 이벤트 유형을 표시하며, 이벤트가 기록되지 않은 경우에도 표시됩니다. 또는 뷰 설정 대화상자에서 빈 이벤트 카테고리를 숨길 수 있습니다.

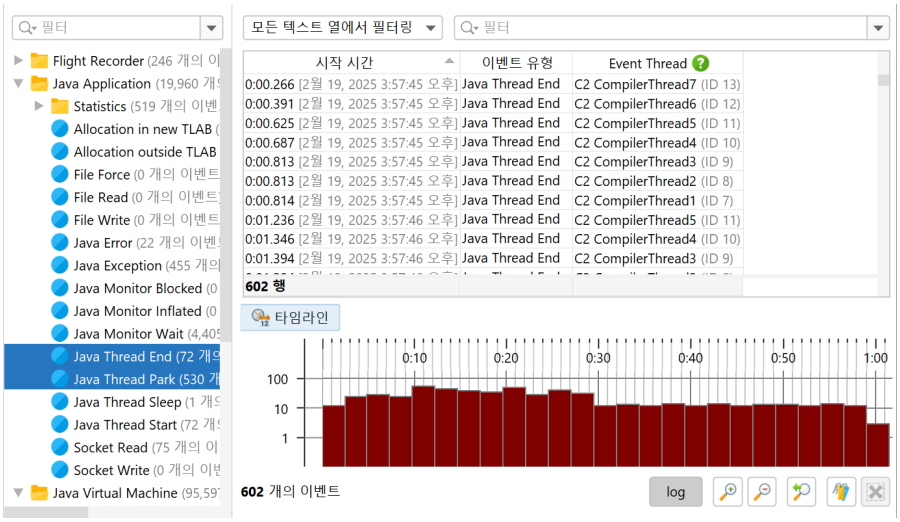
JFR 이벤트

이벤트는 이벤트 유형 트리에서 선택에 따라 열이 달라지는 메인 테이블의 행으로 표시됩니다.



테이블의 이벤트는 기본적으로 시간순으로 정렬됩니다. UI 과부하를 피하기 위해 처음 10000개의 이벤트만 테이블에 표시됩니다. **아래의 분석**은 항상 모든 이벤트에서 계산됩니다. 필터를 설정하면 처음 10000개만이 아닌 모든 이벤트를 확인합니다. 이는 필터를 설정할 때 이전에 표시되지 않았던 이벤트가 테이블에 나타날 수 있음을 의미합니다.

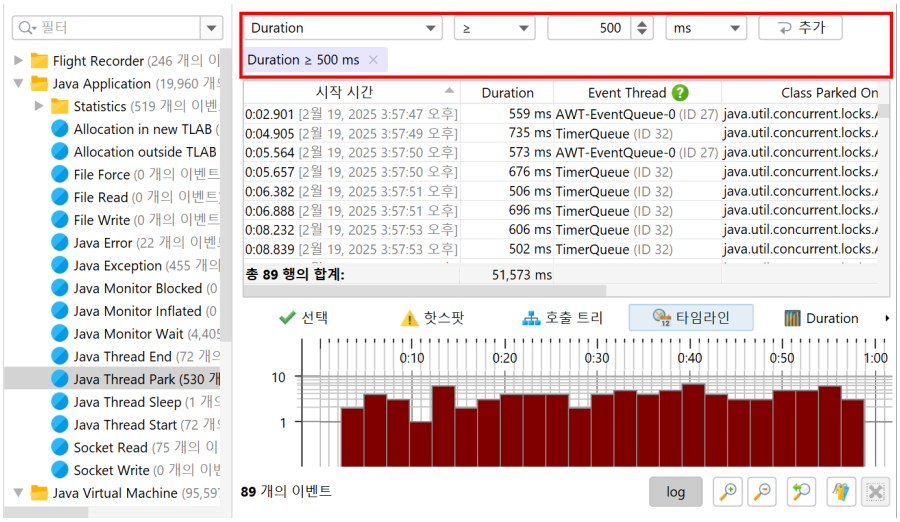
또한 **여러 이벤트 유형** 또는 전체 카테고리를 선택할 수도 있습니다. 이 경우 선택된 모든 이벤트의 합집합이 테이블에 표시됩니다. 각 이벤트 유형은 자체 열 세트를 가지고 있기 때문에, 선택된 모든 이벤트 유형에 공통적인 열만 포함됩니다.



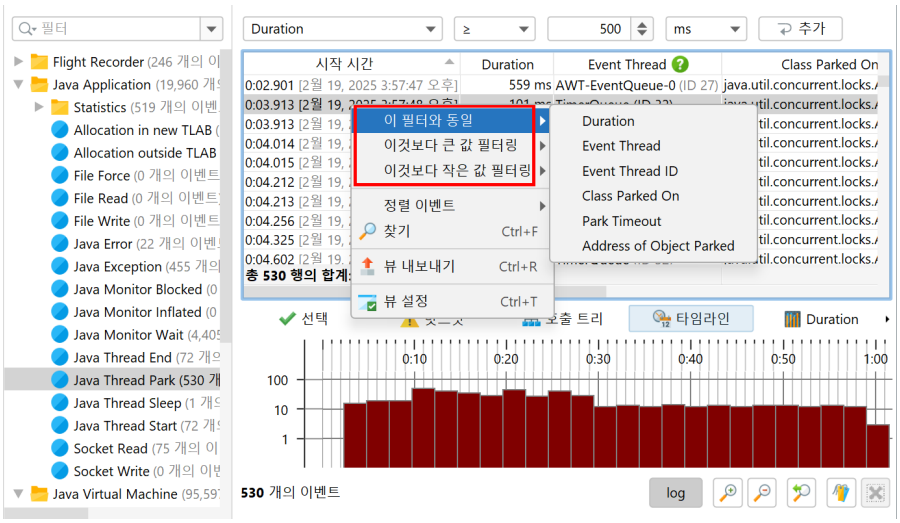
사용 가능한 분석의 수는 또한 사용 가능한 열에 따라 추가되기 때문에 줄어들 수 있습니다.

열 너비는 실제 콘텐츠에 따라 자동으로 조정되며, 열을 조정하면 동일한 콘텐츠 유형의 열 너비가 선택에 고정되어 더 이상 자동으로 변경되지 않습니다. 뷰 설정 대화상자에서 **열 너비 지우기**를 할 때까지는 변경되지 않습니다. 시간이나 메모리와 같은 단위가 있는 열의 스케일도 각 셀에 대해 자동으로 계산됩니다. 비교를 위해 열의 스케일을 고정하려면, 뷰 설정 대화상자에서 각 열에 대한 옵션을 제공합니다. 이 경우 설정은 선택된 각 이벤트 유형에 대해 별도로 저장됩니다.

이벤트를 필터링하는 여러 가지 방법이 있습니다. 테이블 상단에는 필터 선택기가 있어 모든 텍스트 열에서 필터링하거나 단일 열을 선택하고 열 유형에 맞는 필터를 구성할 수 있습니다.

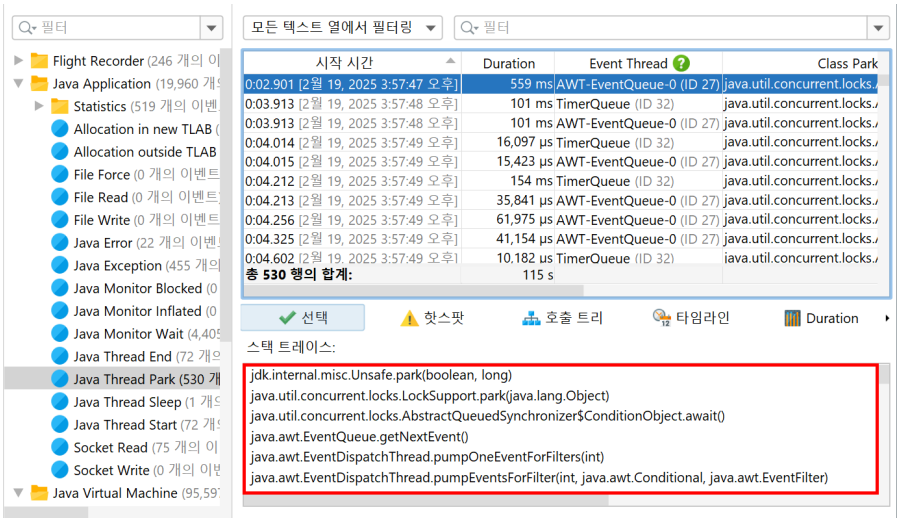


필터링의 또 다른 방법은 관심 있는 행을 선택하고 컨텍스트 메뉴를 사용하여 선택된 행의 값에 기반한 특정 필터를 선택하는 것입니다. 상단의 필터 선택기가 조정되어 선택 사항을 표시합니다. 이제 다른 값을 선택하고 필터를 다시 추가할 수 있으며, 그러면 동일한 열에 대한 이전 필터를 대체합니다. 일반적으로 **각 필터 유형은 한 번만 존재할 수 있으며** 동일한 필터를 다시 설정하면 이전 필터를 대체합니다.

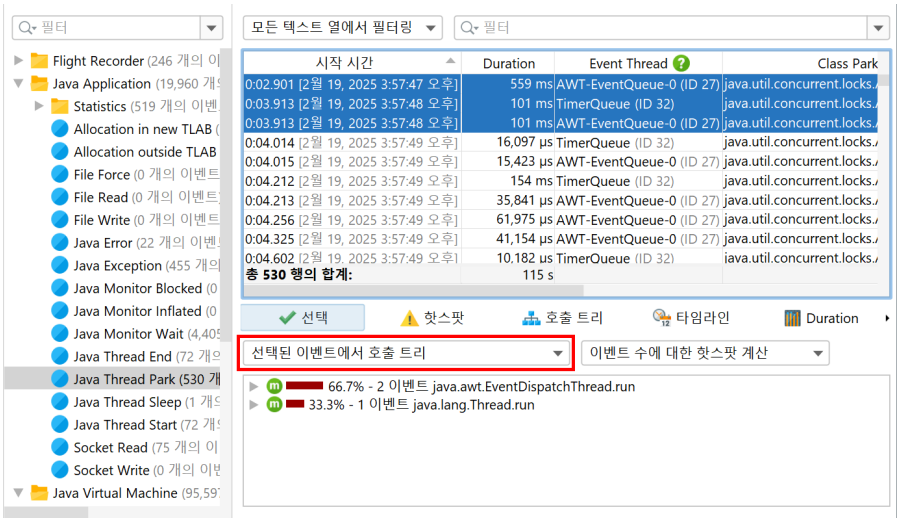


스택 트레이스

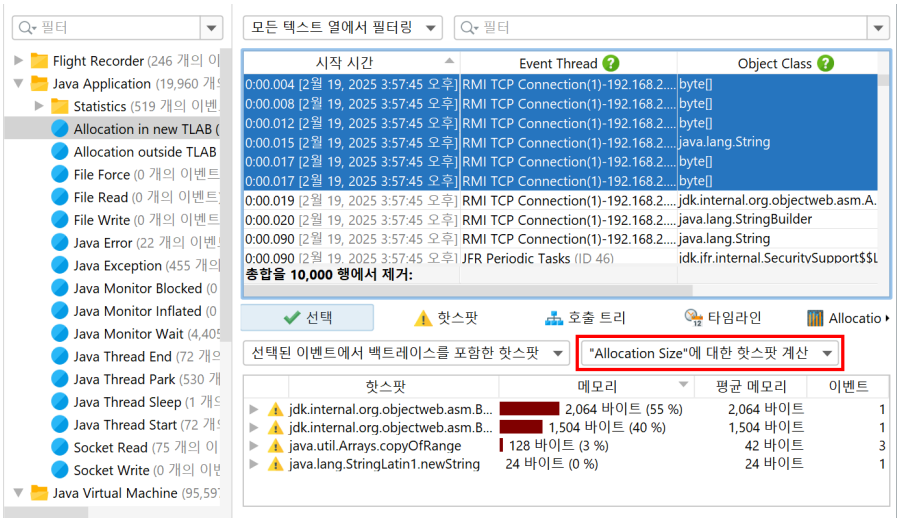
JProfiler에서는 선택된 이벤트의 스택 트레이스가 이벤트 테이블 아래의 분할 창의 "선택" 탭에 표시됩니다.



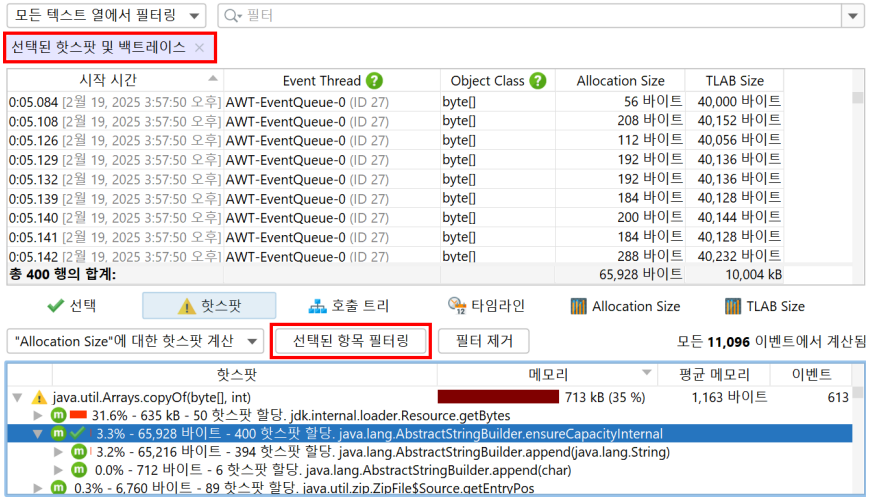
여러 이벤트를 선택하면 선택 탭이 변경되어 선택된 이벤트의 스택 트레이스에서 계산된 핫스팟 또는 누적 호출 트리를 보여주는 뷰로 변경됩니다.



기본적으로 이벤트 수가 호출 트리와 핫스팟 뷰의 노드에 대한 백분율을 결정합니다. 일부 이벤트 유형은 이 목적에 적합한 다른 측정을 포함합니다. 예를 들어, 지속 시간이나 할당된 메모리 등이 있습니다. 이러한 측정이 가능한 경우, 선택 탭의 두 번째 드롭다운에서 **핫스팟 유형**으로 선택할 수 있습니다.



하단 분할 창의 "핫스팟" 및 "호출 트리" 뷰는 동일한 뷰를 포함하지만, 스냅샷의 모든 이벤트에 대해 **계산됩니다**. 선택 탭과 마찬가지로, "핫스팟 유형" 드롭다운도 있습니다. 모든 이벤트를 표시하는 것 외에도, 이러한 뷰에서 **필터를 선택**할 수 있습니다. 호출 트리 뷰에서는 특정 호출 스택을 선택하고 선택된 필터 버튼을 클릭하면 위의 테이블에 해당 호출 스택이 있는 이벤트만 표시됩니다. 핫스팟 뷰에서는 최상위 핫스팟이나 백트레이스의 노드를 선택하여 선택된 노드로의 역 호출 스택 조각으로 끝나는 이벤트만 표시할 수 있습니다.

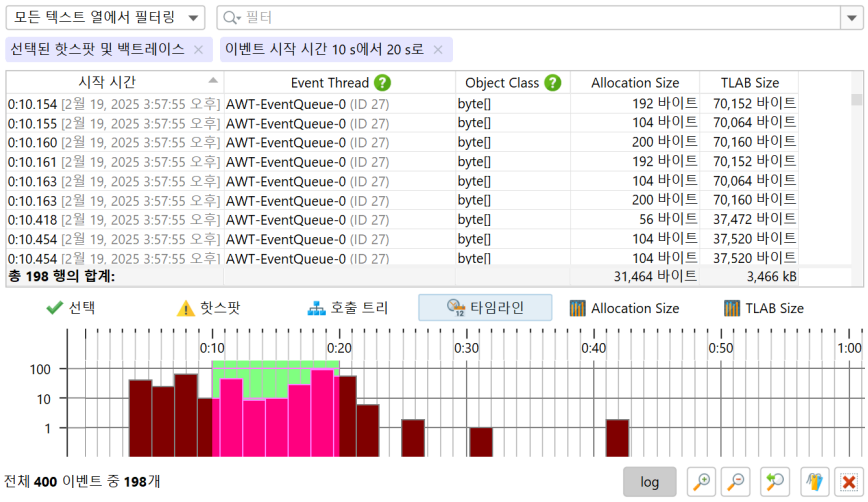


위의 스크린샷에서 백트레이스의 노드가 필터 노드로 선택된 것을 볼 수 있습니다. 일반적인 호출 트리 아이콘 외에도 체크 마크가 포함되어 있습니다. 필터는 상단의 태그 레이블이나 필터 제거 버튼을 통해 제거할 수 있습니다. 테이블의 이벤트 수는 선택된 노드의 수와 같습니다. 핫스팟 트리는 핫스팟 뷰에서 설정된 필터 없이 모든 이벤트를 계속 표시합니다.

이는 분석 뷰에서 설정된 필터의 일반적인 기능입니다: 분석 뷰 자체는 모든 필터링된 이벤트에서 계산되지만, **분석 뷰에서 설정된 필터를 제외하고** 계산됩니다. 이는 분석 뷰를 더 유용하게 만들어 주며, 전체 이벤트 집합에서 선택한 부분을 볼 수 있습니다.

타임라인 뷰

모든 JFR 이벤트는 관련된 시간을 가지고 있으므로, 모든 이벤트 유형 또는 이벤트 유형 집합은 이벤트의 시간 순서 분포를 보여주는 타임라인 뷰를 가지고 있습니다.

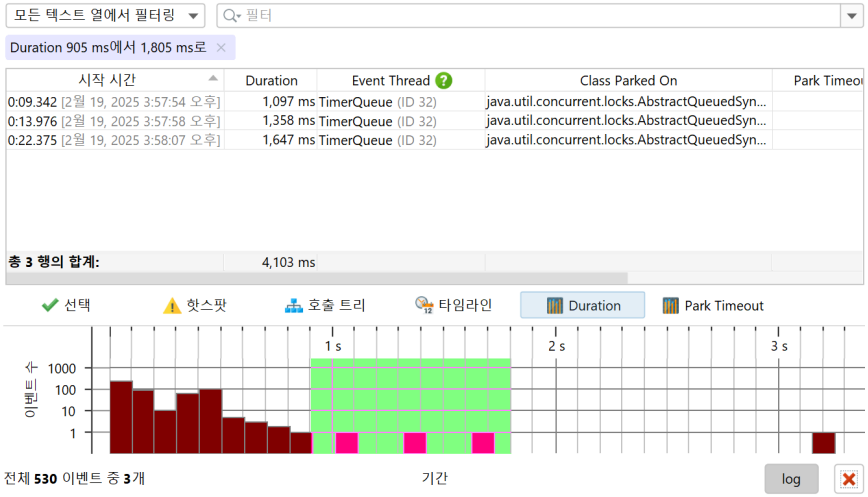


특정 시간 범위에 집중하려면 **시간 축을 따라 드래그**할 수 있습니다. 위의 예에서는 이제 두 개의 필터가 있습니다: 핫스팟의 백트레이스에서의 필터와 타임라인 뷰에서의 필터입니다. 다시 말해, 타임라인 뷰는 전체 시간 범위를 계속 표시하지만, 다른 분석 뷰는 이제 선택된 시간 범위의 이벤트만 표시합니다.

기본 표시 모드는 **로그 모드**로, 이벤트 수가 적은 영역도 이벤트 수가 많은 영역에 대해 여전히 보이도록 합니다. 타임라인 아래의 log 버튼을 선택 해제하여 선형 모드로 전환할 수 있습니다. 기본적으로 전체 시간 범위가 사용 가능한 너비에 표시되지만, 가변 시간 범위로 전환하고 다른 JProfiler의 텔레메트리와 마찬가지로 확대 및 스크롤할 수 있습니다. 또한 **북마크**를 사용할 수 있으며, 선택된 시간 범위에 수직 마커를 추가할 수 있습니다. 이렇게 하면 다른 이벤트 유형 간에 시간을 비교할 수 있습니다.

히스토그램 뷰

지속 시간 및 할당 크기와 같이 여러 이벤트에 대해 합산할 수 있는 모든 측정은 특별한 방식으로 처리됩니다: 첫째, 이벤트 테이블의 이러한 측정의 열은 하단에 총 값을 가지고 있습니다. 둘째, 호출 트리 및 핫스팟 분석 뷰는 이벤트 수 대신 이러한 측정으로 트리를 계산하기 위해 "핫스팟 유형" 드롭다운을 제공합니다. 마지막으로, 각 측정에 대해 하단 분할 패널에 히스토그램 분석이 추가됩니다.



히스토그램은 수직 축에 이벤트 수를 표시하고, 수평 축은 선택된 측정을 보여주며, 분포를 계산할 수 있도록 여러 개의 빈으로 나뉩니다. 빈 크기와 이벤트 수는 툴팁에서 확인할 수 있습니다.

위의 스크린샷은 히스토그램에서 필터가 설정된 방법을 보여줍니다. 다른 분석 뷰와 마찬가지로, 필터는 다른 분석 뷰에만 적용되며, 전체 히스토그램은 여전히 표시됩니다. 타임라인 뷰와 마찬가지로, 히스토그램은 기본

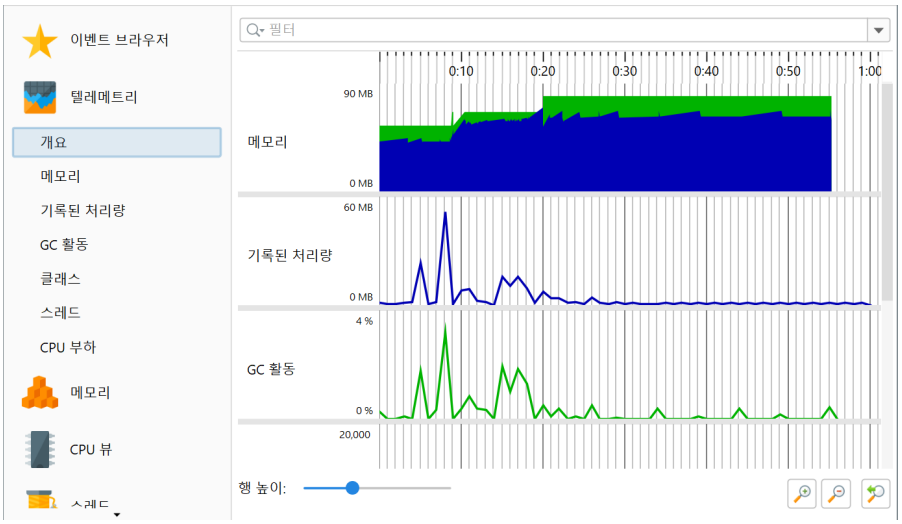
적으로 로그 수직 축을 가지고 있습니다. 여기서 스크린샷의 선택된 이벤트는 선형 축에서는 보이지 않을 것입니다.

E.4 JFR 스냅샷의 뷰

JFR 이벤트 브라우저 [p. 214] 외에도, JProfiler는 전체 프로파일링 세션에서 사용할 수 있는 몇 가지 뷰를 사용하여 JFR 데이터를 채웁니다. 이는 JFR이 메모리 할당 및 메서드 실행에 대한 데이터를 수집하기 때문에 가능합니다. 주요 제한 사항은 녹화 속도가 낮아서 문제 있는 핫스팟을 보기에 충분한 데이터를 얻는 데 오랜 시간이 걸릴 수 있다는 것입니다.

텔레메트리

"기록된 객체 텔레메트리"를 제외하고, 전체 프로파일링 세션의 모든 텔레메트리는 JFR 스냅샷에서도 일부 표시 데이터의 제한과 함께 사용할 수 있습니다. 메모리 텔레메트리는 GC-특정 풀을 표시하지 않으며, 스레드 텔레메트리는 스레드 상태별 스레드 수를 표시하지 않으며, 기록된 처리량 텔레메트리는 객체 수 대신 크기를 표시하고 해제되는 객체를 표시하지 않습니다.



아래 표는 다양한 텔레메트리에서 사용되는 이벤트 유형과 "기본" 및 "프로파일" 템플릿 모두에서 활성화 여부를 보여줍니다.

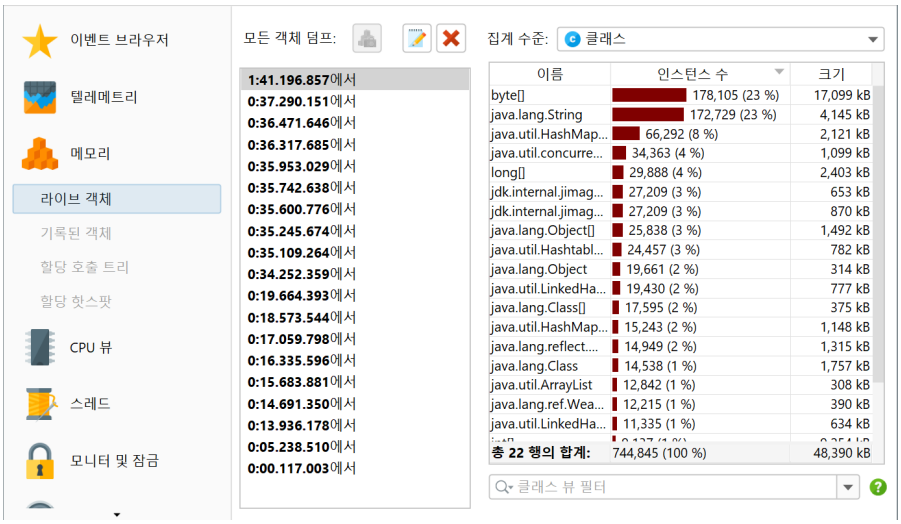
텔레메트리	이벤트 유형	프로파일에서 활성화됨
메모리	jdk.GCHeapSummary, jdk.MetaspaceSummary	모두
기록된 처리량	jdk.ObjectAllocationSample, jdk.ObjectAllocationInNewTLAB, jdk.ObjectAllocationOutsideTLAB	프로파일 전용
GC 활동	jdk.GarbageCollection	모두
클래스	jdk.ClassLoadingStatistics	모두
스레드	jdk.JavaThreadStatistics	모두
CPU 부하	jdk.CPULoad	모두

메모리 뷰

"메모리" 섹션에서는 두 가지 다른 이벤트 유형을 사용하여 뷰에 데이터를 채웁니다. "라이브 객체" 뷰는 전체 가비지 컬렉션 후 힙에 남아 있는 모든 클래스와 인스턴스 수의 통계적 표현을 보여줍니다. 이 데이터는 `jdk.ObjectCount` 이벤트가 활성화된 경우에만 사용할 수 있으며, 이는 기본 JFR 템플릿 중 어느 것도 해당되지 않으며 상당한 오버헤드가 있기 때문입니다. 고급 JFR 구성에서 "Garbage collector" 드롭다운을 사용하여 이 설정을 전환할 수도 있습니다. Java 17 이전에는 이 드롭다운이 "메모리 프로파일링"으로 표시됩니다.

`jdk.ObjectCount` 이벤트가 스냅샷에서 여러 번 기록된 경우, 뷰는 `jdk.ObjectCount` 이벤트의 **첫 번째와 마지막 발생 사이의 차이**를 보여줍니다. 이렇게 하면 녹화 시간 동안 숫자가 어떻게 변했는지에 대한 감각을 얻을 수 있으며 메모리 누수의 징후를 제공할 수 있습니다. 이러한 시간이 스냅샷 녹화의 시작 및 종료 지점과 일치하지 않는 경우, 해당 북마크가 텔레메트리 뷰에 추가됩니다. 힙의 1% 이상인 총 객체 크기를 가진 클래스만 포함됩니다.

심각한 조사를 위해 전체 프로파일링 세션 [p. 68]을 사용하거나 HPROF 스냅샷 [p. 195]을 찍는 것을 고려하십시오.

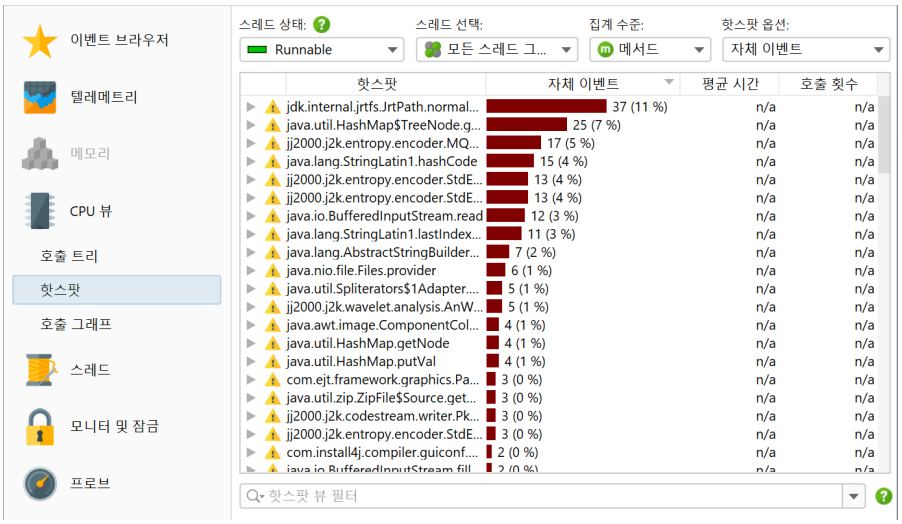


"기록된 객체" 뷰와 할당 뷰는 Java 16 이후 `jdk.ObjectAllocationSample` 이벤트와 이전 Java 버전의 `jdk.ObjectAllocationInNewTLAB` 및 `jdk.ObjectAllocationOutsideTLAB` 이벤트에서 데이터를 보여줍니다. 고급 UI의 "할당 프로파일링" 드롭다운에서도 이러한 이벤트 유형을 활성화할 수 있습니다.

"라이브 객체" 뷰와 달리, 녹화가 활성화된 동안 할당된 객체만 표시됩니다. 할당은 JFR에 의해 샘플링되지만 크기는 **총 할당된 크기에 대한 추정치**로 보고됩니다. 이러한 불일치로 인해, 이러한 뷰에서 보고된 크기는 샘플 수에 평균 인스턴스 크기를 곱한 것과 일치하지 않습니다. 그렇지 않으면, 이러한 뷰는 전체 프로파일링 세션의 메모리 뷰 [p. 68]와 유사한 기능을 가지고 있습니다.

CPU 뷰

"CPU 뷰"에는 호출 트리, 핫스팟 뷰 및 호출 그래프가 포함됩니다. "Runnable" 스레드 상태의 데이터는 기본적으로 표준 JFR 템플릿 모두에서 기록되는 `jdk.ExecutionSample` 이벤트를 기반으로 합니다. 그러나 샘플링 속도는 기본적으로 20ms로 설정되어 있으며, 이는 JFR 고급 UI의 "메서드 샘플링" 설정의 "Normal" 옵션에 해당합니다. JFR은 매우 적은 수의 임의 스레드만 샘플링하기 때문에, 핫스팟이 충분히 두드러지게 보일 만큼 충분한 데이터를 얻는 데 오랜 시간이 걸릴 수 있습니다. 필요한 경우 `jdk.ExecutionSample`의 주기를 줄이는 것을 고려하십시오. 이로 인해 JFR이 데이터를 누적하지 않기 때문에 매우 큰 스냅샷 크기가 될 수 있음을 유의하십시오.



스레드가 간헐적으로 샘플링되기 때문에, 전체 프로파일링 세션에서처럼 실제 실행 시간을 추정할 수 없습니다. 시간 대신, 호출 트리와 핫스팟 뷰에서는 **이벤트 수**가 표시됩니다. 이는 동일한 단점을 가진 비동기 샘플링 [p. 63]과 유사합니다. 다른 JFR 스레드 상태는 "Waiting", "Blocking" 및 "Socket and file I/O"이며 여전히 시간을 측정합니다. 이러한 불일치로 인해, 스레드 상태 선택기에서 "모든 스레드 상태" 모드는 사용할 수 없습니다.

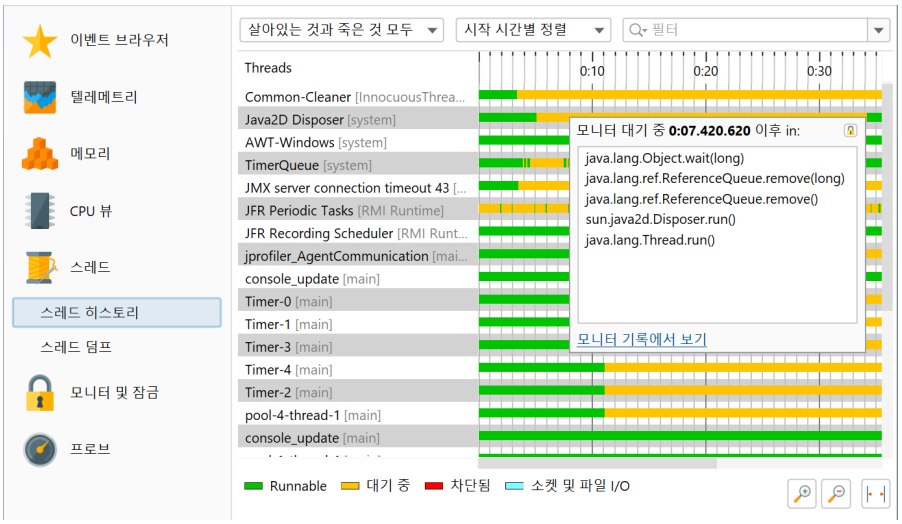
또 다른 고려 사항은 비실행 스레드 상태가 스레드 상태 선택기 옆의 도구 설명에 표시되는 구성 가능한 최소 지속 시간 임계값을 가진 이벤트에서 계산된다는 것입니다. 이러한 스레드 상태의 실제 총 시간은 훨씬 더 클 수 있습니다. 스레드 상태를 조립하는 데 사용되는 이벤트 유형이 포함된 표는 아래에 나와 있습니다:

스레드 상태	이벤트 유형
Runnable	jdk.ExecutionSample
Waiting	jdk.JavaMonitorWait, jdk.ThreadSleep, jdk.ThreadPark
Blocking	jdk.JavaMonitorEnter
Socket and file I/O	jdk.SocketRead, jdk.SocketWrite, jdk.FileRead, jdk.FileWrite

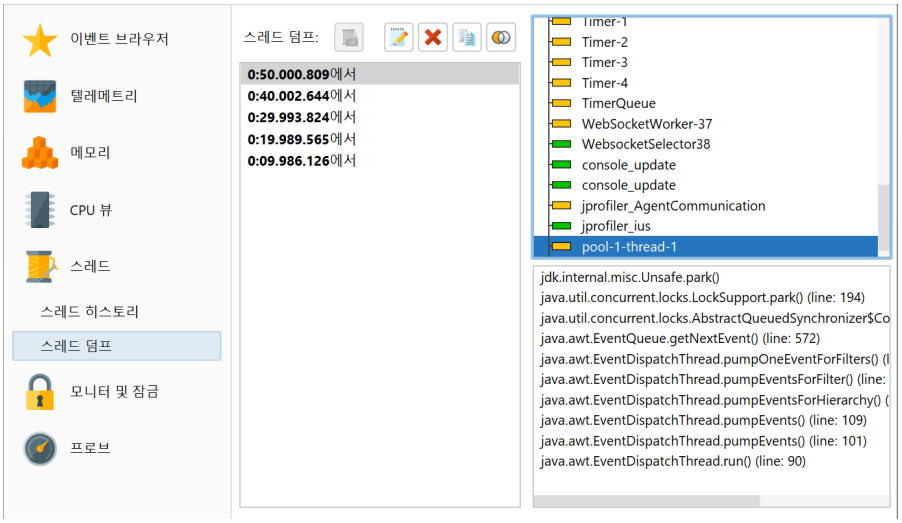
뷰의 기능은 CPU 뷰에 대한 도움말 주제 [p. 51]에서 설명되어 있습니다. 전체 프로파일링 세션의 많은 기능이 JFR 컨텍스트에서는 사용할 수 없음을 유의하십시오.

스레드 및 모니터 뷰

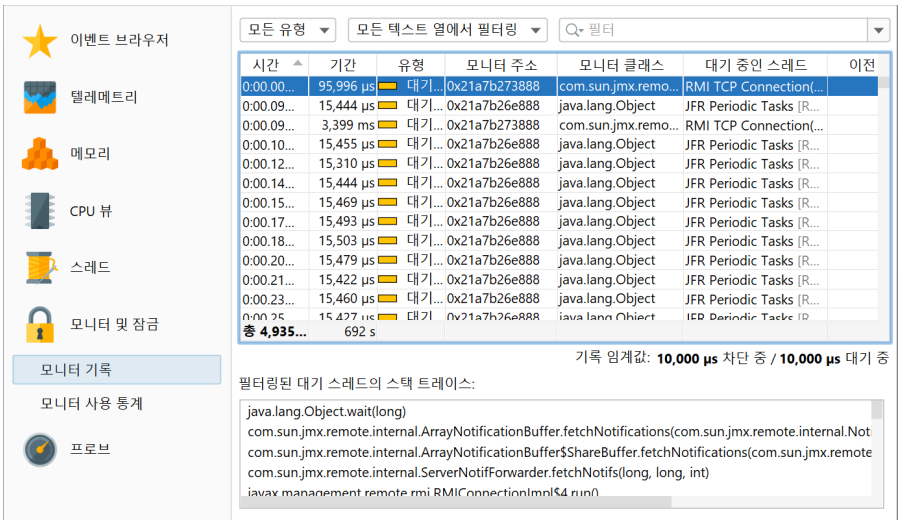
연대기적 메서드 샘플링 데이터에서, 스레드 히스토리 뷰를 계산할 수 있으며, 대기 및 차단 시간에 대한 스택 트레이스를 보여주는 도구 설명도 포함됩니다.



스레드 덤프는 JFR과 JProfiler 모두에서 기능으로 제공되며 동일한 뷰에 표시됩니다. 이 경우, 이벤트 브라우저는 `jdk.ThreadDump` 이벤트의 스레드 덤프 열의 구조화된 내용을 표시할 방법이 없기 때문에 대체가 아닙니다. 스레드 덤프 뷰에서는 다른 스레드 덤프를 비교 [p. 92]할 수도 있습니다.



`jdk.JavaMonitorWait`, `jdk.ThreadSleep` 및 `jdk.ThreadPark` 이벤트에서, JProfiler는 차단 스레드에 대한 정보 없이 전체 프로파일링 세션 [p. 92]과 유사한 모니터 히스토리를 계산합니다. 문제 해결에 해당 정보가 필요한 경우, 전체 프로파일링 세션으로 전환하십시오. 이는 전체 프로파일링 세션의 잠금 그래프가 JFR 스냅샷에서는 사용할 수 없음을 의미하기도 합니다. 대기 이벤트에 대한 집계 정보를 보여주는 모니터 사용 통계가 있으며 대기 시간만 표시됩니다.



프로브

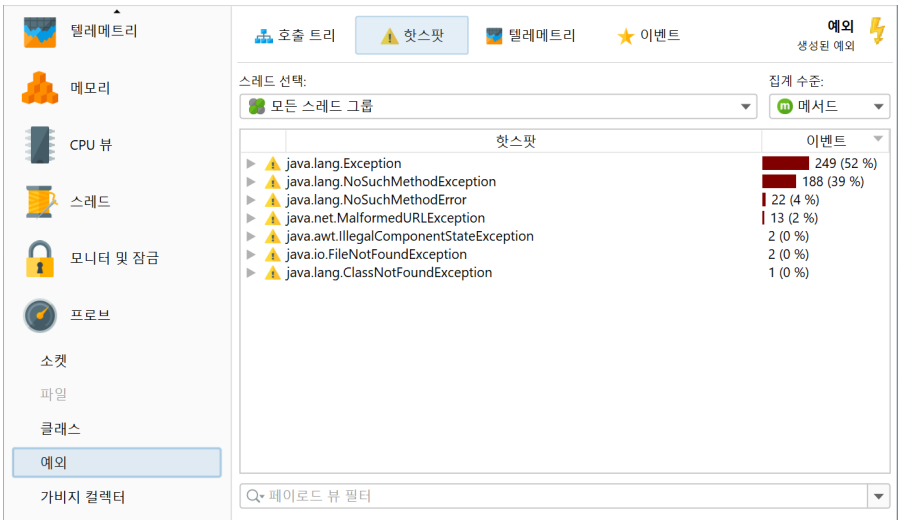
전체 프로파일링 세션의 일부 JVM 프로브는 JFR 스냅샷에서 동등한 데이터 소스를 가지고 있습니다. 이벤트 브라우저에 비해 주요 장점은 여러 관련 이벤트 유형을 결합한다는 것입니다. 아래 표는 데이터 소스로 사용되는 이벤트 유형과 함께 사용할 수 있는 프로브를 보여줍니다.

프로브	이벤트 유형	프로파일에서 활성화됨
Sockets	jdk.SocketRead, jdk.SocketWrite	모두
Files	jdk.FileRead, jdk.FileWrite	모두
클래스	jdk.ClassLoad, jdk.ClassUnload, jdk.ClassDefine	없음
예외	jdk.JavaErrorThrow, jdk.JavaExceptionThrow	오류는 모두, 예외는 없음
Garbage Collector	jdk.GarbageCollection, jdk.GCPhasePause, jdk.YoungGarbageCollection, jdk.OldGarbageCollection, jdk.GCReferenceStatistics, jdk.GCPhasePauseLevel<n>, jdk.GCHeapSummary, jdk.MetaspaceSummary, jdk.GCHeapConfiguration, jdk.GCConfiguration, jdk.YoungGenerationConfiguration, jdk.GCSurvivorConfiguration, jdk.GCTLABConfiguration	모두

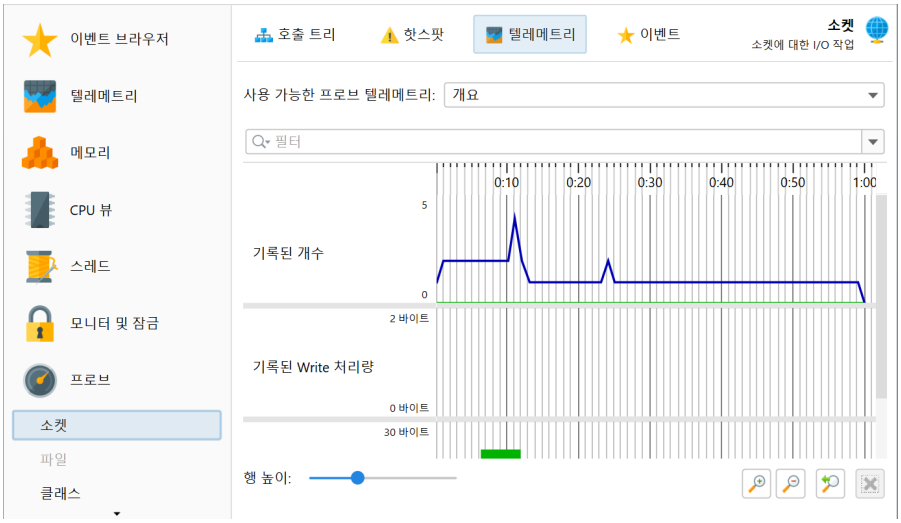
클래스 로딩은 고급 JFR UI에서 세 가지 클래스 로딩 이벤트를 모두 켜는 별도의 체크 박스를 가지고 있습니다.

각 프로브는 여러 개의 뷰를 보여줍니다. 이벤트 브라우저와 달리, 초점은 단일 이벤트가 아니라 집계된 데이터에 맞춰져 있습니다. 이것이 JProfiler의 프로브가 JFR 데이터 수집과 개념적으로 다른 점입니다.

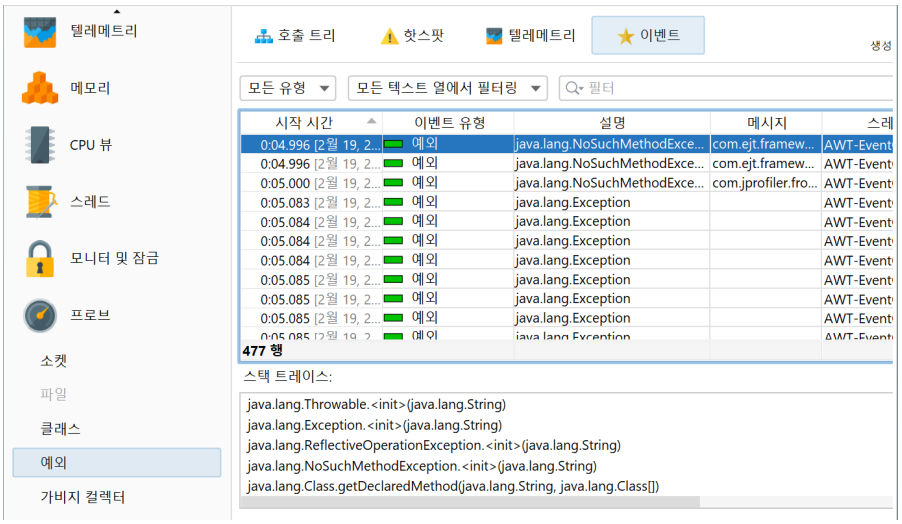
Garbage collector 프로브를 제외하고, 모든 프로브는 다음과 같은 뷰를 가지고 있습니다: 호출 트리와 핫스팟 뷰에서는 단일 스레드 또는 스레드 그룹을 선택할 수 있으며 집계 수준도 선택할 수 있습니다. 기본적으로 모든 스레드가 표시되며 집계 수준은 "메서드"로 설정됩니다.



텔레메트리 뷰는 기록된 데이터에서 하나 이상의 텔레메트리를 표시하며, 모든 텔레메트리를 한 번에 보여주는 개요 페이지가 있습니다. 전체 텔레메트리는 텔레메트리 이름을 클릭하여 열 수 있습니다. 시간 축을 따라 드래그하여 이벤트 뷰에서 해당 이벤트를 선택할 수 있습니다.



이벤트 뷰는 JFR 브라우저의 것과 유사합니다. 그러나 여러 JFR 이벤트에 해당하는 여러 이벤트 유형을 표시하며 유형 선택기를 제공합니다. 단일 및 다중 선택에 대한 필터링 및 스택 트레이스 표시가 이벤트 브라우저와 동일하게 처리됩니다. 또한, 시간 및 메모리 측정을 위한 히스토그램 뷰가 있으며, 수평 축을 따라 드래그하여 범위를 선택할 수 있습니다.



Garbage collector 뷰는 특별하며, Java 17 이상에서 프로파일링 세션과 동일한 정보를 전체 프로파일링 세션에서 보여줄 수 있습니다. JVM 프로브 카테고리에서 garbage collector 프로브가 기록되면, 필요한 데이터를 얻기 위해 JFR 스트리밍이 사용됩니다. 자세한 내용은 garbage collector 분석 [p. 112] 장을 참조하십시오.

F 상세 설정

F.1 연결 문제 해결

프로파일링 세션을 설정할 수 없을 때, 가장 먼저 해야 할 일은 프로파일된 애플리케이션 또는 애플리케이션 서버의 터미널 출력을 확인하는 것입니다. 애플리케이션 서버의 경우, stderr 스트림이 종종 로그 파일에 기록됩니다. 이는 애플리케이션 서버의 주요 로그 파일이 아닌 별도의 로그 파일일 수 있습니다. 예를 들어, Websphere 애플리케이션 서버는 `native_stderr.log` 파일을 작성하여 stderr 출력만 포함합니다. stderr 출력의 내용에 따라 문제를 찾는 방향이 달라집니다:

연결 문제

stderr에 `"Waiting for connection ..."`이 포함되어 있으면, 프로파일된 애플리케이션의 구성은 정상입니다. 문제는 다음 질문과 관련이 있을 수 있습니다:

- 로컬 머신의 JProfiler GUI에서 "Attach to remote JVM" 세션을 시작하는 것을 잊으셨나요? 프로파일링 에이전트가 "nowait" 옵션으로 즉시 시작하도록 구성되지 않은 한, JProfiler GUI가 연결될 때까지 VM이 시작을 계속하지 않습니다.
- 세션 설정에서 호스트 이름이나 IP 주소가 올바르게 구성되어 있습니까?
- 잘못된 통신 포트를 구성했습니까? 통신 포트는 HTTP 또는 다른 표준 포트 번호와 관련이 없으며 이미 사용 중인 포트와 동일해서는 안 됩니다. 프로파일된 애플리케이션의 경우, 통신 포트는 프로파일링 VM 매개변수의 옵션으로 정의됩니다. VM 매개변수 `-agentpath:<path to jprofilerti library>=port=25000`을 사용하면 25000 포트가 사용됩니다.
- 루프백 인터페이스에서만 수신 대기하는 직접 연결로 에이전트에 연결하려고 합니까? 기본적으로 에이전트는 루프백 인터페이스에서만 수신 대기합니다. JProfiler를 구성하여 원격 머신에 SSH 터널을 설정할 수 있습니다. 암호화가 필요하지 않다면 `address=[IP address]` 옵션을 `-agentpath` 매개변수에 사용할 수도 있습니다.
- 로컬 머신과 원격 머신 사이에 방화벽이 있습니까? 들어오는 연결뿐만 아니라 나가는 연결에 대한 방화벽이나 중간의 게이트웨이 머신에 방화벽이 있을 수 있습니다.

포트 바인딩 문제

stderr에 소켓을 바인딩할 수 없다는 오류 메시지가 포함되어 있으면, 포트가 이미 사용 중입니다. 이 경우, 다음 질문을 확인하십시오:

- 프로파일된 애플리케이션을 여러 번 시작했습니까? 각 프로파일된 애플리케이션은 별도의 통신 포트가 필요합니다.
- 이전 프로파일링 실행의 좀비 Java 프로세스가 포트를 차단하고 있습니까?
- 통신 포트를 사용하는 다른 애플리케이션이 있습니까?

stderr에 `JProfiler>`로 시작하는 줄이 없고 애플리케이션 또는 애플리케이션 서버가 정상적으로 시작되면, `-agentpath:[path to jprofilerti library]` VM 매개변수가 Java 호출에 포함되지 않았습니다. 시작 스크립트에서 실제로 실행되는 Java 호출을 찾아 VM 매개변수를 추가해야 합니다.

Attach 문제

실행 중인 JVM에 attach할 때, 관심 있는 JVM이 모든 JVM 목록에 표시되지 않을 수 있습니다. 이 문제의 원인을 찾으려면 attach 메커니즘이 어떻게 작동하는지 이해하는 것이 중요합니다. JVM이 시작되면, 임시 디렉토리의 `hsperfdata_$USER` 디렉토리에 PID 파일을 작성하여 발견됩니다. 동일한 사용자나 관리자 사용자만 JVM에 attach할 수 있습니다. JProfiler는 관리자 사용자로 JVM에 연결하는 데 도움을 줄 수 있습니다.

Windows에서는 Show Services 버튼을 사용하여 모든 JVM 서비스 프로세스를 표시할 수 있습니다. JProfiler는 시스템 계정으로 실행되는 서비스와 구성된 사용자 계정으로 실행되는 서비스에 연결할 수 있는 시스템 계

정으로 실행되는 도우미 서비스를 설치합니다. 해당 서비스의 이름은 "JProfiler helper"이며, 해당 버튼을 클릭하면 설치됩니다. 서비스 설치를 허용하려면 UAC 프롬프트를 확인해야 합니다. JProfiler가 종료되면 서비스는 다시 제거됩니다.

Linux에서는 attach 대화 상자에서 사용자 전환기를 사용하여 root 계정으로 attach할 수 있습니다. 이 사용자 전환기는 로컬 JVM을 프로파일링할 때뿐만 아니라 원격 Linux 또는 macOS 머신에 attach할 때도 표시됩니다. 원격 attach의 경우, 다른 비-root 사용자로 전환할 수도 있습니다. root 비밀번호가 있는 경우, 서비스를 실행하는 실제 사용자보다는 항상 root로 전환하십시오.

Linux에서 JVM이 표시되지 않는 경우, 문제는 일반적으로 임시 디렉토리와 관련이 있습니다. 한 가지 가능성은 `/tmp/hsperfdata_$USER` 디렉토리의 액세스 권한이 잘못된 것입니다. 이 경우, 디렉토리를 삭제하고 JVM을 다시 시작하십시오. attach할 프로세스는 /tmp에 쓰기 권한이 있어야 하며, 그렇지 않으면 attach가 지원되지 않습니다.

systemd를 사용하는 경우, 관심 있는 프로세스가 systemd 서비스 파일에 `PrivateTmp=yes`로 설정되어 있을 수 있습니다. 그러면 pid 파일이 다른 위치에 작성됩니다. attach 대화 상자에서 사용자 전환기를 사용하여 root 사용자로 전환하거나 CLI 도구를 root로 사용하면 JProfiler가 이를 처리합니다.

원격 attach를 위한 자동 에이전트 다운로드

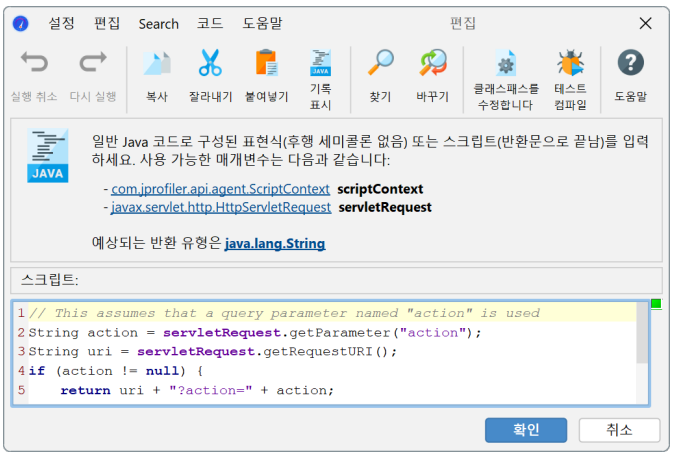
원격 attach의 경우, JProfiler는 원격 대상 플랫폼에 대한 에이전트 라이브러리가 필요합니다. 로컬에 사용 가능한 라이브러리가 없으면 다운로드를 시도합니다. 이 작업은 HTTPS 연결을 `https://download.ej-technologies.com`로 차단하거나 중간자 공격 방식으로 SSL 연결을 검사하여 트래픽을 해독하는 방화벽이 있는 경우 실패할 수 있습니다. 후자의 경우, JProfiler는 방화벽의 인증서를 가지고 있지 않기 때문에 HTTPS 연결이 실패합니다.

에이전트 다운로드 오류가 발생하면 JProfiler는 수동 해결 방법을 제공합니다. 대화 상자가 표시되어 웹사이트에서 에이전트 아카이브를 수동으로 다운로드하고 원격 attach 작업을 계속하기 전에 다운로드한 아카이브를 찾는 방법을 보여줍니다.

에이전트 파일은 캐시되기 때문에, 이는 각 원격 플랫폼에 대한 일회성 작업입니다. JProfiler가 업데이트되면 에이전트가 변경되며 다운로드를 다시 수행해야 합니다.

F.2 JProfiler에서의 스크립트

JProfiler의 내장 스크립트 편집기를 사용하면 JProfiler GUI의 다양한 위치에서 사용자 정의 로직을 입력할 수 있습니다. 여기에는 사용자 정의 프로브 구성, 분할 메소드, 힙 워커 필터 등이 포함됩니다.



편집 영역 위의 상자는 스크립트의 사용 가능한 매개변수와 반환 유형을 보여줍니다. 메뉴에서 도움말->Javadoc 개요 보기를 호출하여 `com.jprofiler.api.*` 패키지의 클래스에 대한 추가 정보를 얻을 수 있습니다.

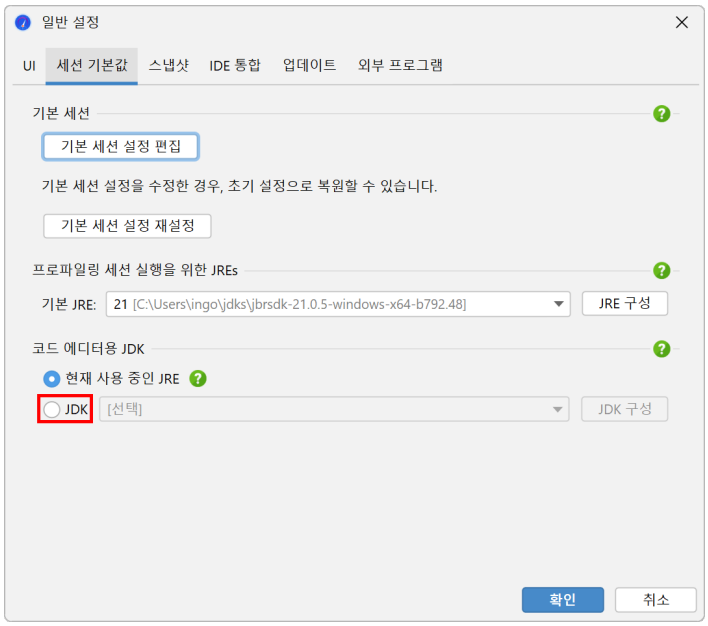
여러 패키지는 완전한 클래스 이름을 사용하지 않고도 사용할 수 있습니다. 이러한 패키지는 다음과 같습니다:

- java.util.*
- java.io.*

완전한 클래스 이름 사용을 피하기 위해 텍스트 영역의 첫 줄에 여러 import 문을 넣을 수 있습니다.

모든 스크립트는 스크립트의 연속 호출 간에 상태를 저장할 수 있는 `com.jprofiler.api.agent.ScriptContext`의 인스턴스를 전달받습니다.

최대 편집기 기능을 얻으려면 일반 설정에서 JDK를 구성하는 것이 좋습니다. 기본적으로 JProfiler가 실행되는 JRE가 사용됩니다. 이 경우 코드 완성은 JRE의 클래스에 대한 매개변수 이름과 Javadoc을 제공하지 않습니다.



스크립트 유형

스크립트는 표현식일 수 있습니다. 표현식은 후행 세미콜론이 없으며 필요한 반환 유형으로 평가됩니다. 예를 들어,

```
object.toString().contains("test")
```

힙 워커의 나가는 참조 뷰에서 필터 스크립트로 작동합니다.

또는 스크립트는 일련의 Java 문으로 구성되며 마지막 문으로 필요한 반환 유형의 return 문이 포함됩니다:

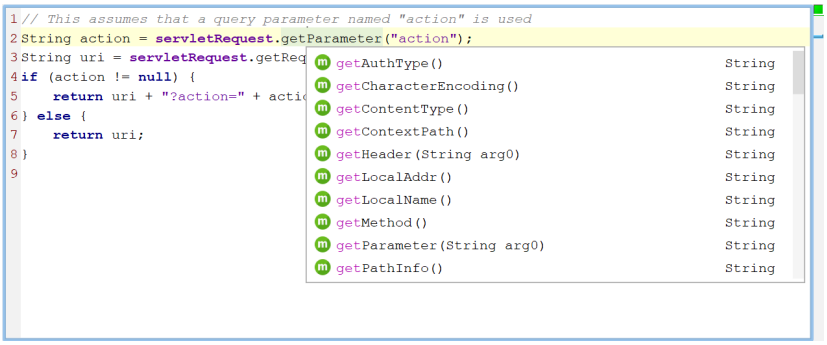
```
import java.lang.management.ManagementFactory;
return ManagementFactory.getRuntimeMXBean().getUptime();
```

위의 예는 스크립트 텔레메트리에 적합합니다. JProfiler는 표현식이나 스크립트를 입력했는지 자동으로 감지합니다.

이전에 입력한 스크립트를 재사용하려면 스크립트 기록에서 선택할 수 있습니다. 기록 보기 도구 모음 버튼을 클릭하면 이전에 사용한 모든 스크립트가 표시됩니다. 스크립트는 스크립트 서명별로 정리되며 현재 스크립트 서명이 기본적으로 선택됩니다.

코드 완성

CTRL-Space를 누르면 코드 완성 제안이 포함된 팝업이 나타납니다. 또한, 점(".")을 입력하면 다른 문자가 입력되지 않을 경우 지연 후에 이 팝업이 표시됩니다. 지연 시간은 편집기 설정에서 구성할 수 있습니다. 팝업이 표시되는 동안 Backspace로 문자를 계속 입력하거나 삭제할 수 있으며 팝업은 이에 따라 업데이트됩니다. "Camel-hump" 완성이 지원됩니다. 예를 들어, NPE를 입력하고 CTRL-Space를 누르면 `java.lang.NullPointerException`을 포함한 다른 클래스가 제안됩니다. 자동으로 import되지 않는 클래스를 수락하면 완전한 이름이 삽입됩니다.



자동 완성 팝업은 다음을 제안할 수 있습니다:

- 변수 및 스크립트 매개변수. 스크립트 매개변수는 굵은 글꼴로 표시됩니다.
- import 문을 입력할 때 패키지
- 클래스
- 클래스가 컨텍스트일 때 필드
- 클래스 또는 메소드의 매개변수 목록이 컨텍스트일 때 메소드

구성된 세션 클래스 경로나 구성된 JDK에 포함되지 않은 클래스가 있는 매개변수는 `[unresolved]`로 표시되며 일반 `java.lang.Object` 유형으로 변경됩니다. 이러한 매개변수에 메소드를 호출하고 코드 완성을 받으려면 누락된 JAR 파일을 애플리케이션 설정의 클래스 경로에 추가하십시오.

문제 분석

입력한 코드는 실시간으로 분석되어 오류 및 경고 조건을 확인합니다. 오류는 편집기에서 빨간색 밑줄로 표시되고 오른쪽 여백에 빨간색 줄무늬로 표시됩니다. 사용되지 않는 변수 선언과 같은 경고는 편집기에서 노란색 배경으로 표시되고 여백에 노란색 줄무늬로 표시됩니다. 편집기에서 오류나 경고 위에 마우스를 올리거나 여백 영역의 줄무늬 위에 마우스를 올리면 오류 또는 경고 메시지가 표시됩니다.

오른쪽 여백 상단의 상태 표시기는 코드에 경고나 오류가 없으면 녹색, 경고가 있으면 노란색, 오류가 발견되면 빨간색입니다. 편집기 설정에서 문제 분석의 임계값을 구성할 수 있습니다.

대화 상자의 오른쪽 상단 모서리에 있는 여백 아이콘이 녹색이면 편집기 설정에서 오류 분석을 비활성화하지 않는 한 스크립트가 컴파일됩니다. 일부 상황에서는 실제 컴파일을 시도해 볼 수 있습니다. 메뉴에서 코드->테스트 컴파일을 선택하면 스크립트를 컴파일하고 별도의 대화 상자에 오류를 표시합니다. 확인 버튼으로 스크립트를 저장해도 스크립트가 바로 사용되지 않는 한 스크립트의 구문적 정확성을 테스트하지 않습니다.

키 바인딩

SHIFT-F1을 누르면 커서 위치의 요소를 설명하는 Javadoc 페이지가 브라우저에서 열립니다. Java 런타임 라이브러리에 대한 Javadoc은 코드 편집기의 일반 설정에서 유효한 Javadoc 위치가 있는 JDK가 구성된 경우에만 표시할 수 있습니다.

Java 코드 편집기의 모든 키 바인딩은 구성할 수 있습니다. 창 메뉴에서 설정->키 맵을 선택하여 키 맵 편집기를 표시합니다. 키 바인딩은 파일 `$HOME/.jprofiler15/editor_keymap.xml`에 저장됩니다. 이 파일은 기본 키 맵이 복사된 경우에만 존재합니다. JProfiler 설치를 다른 컴퓨터로 마이그레이션할 때 이 파일을 복사하여 키 바인딩을 보존할 수 있습니다.

F.3 사용자 정의 도움말

사용자를 위한 추가 지침을 제공하는 내부 웹사이트가 있는 경우, 툴바와 "도움말" 메뉴에 추가 도움말 버튼을 추가할 수 있습니다. 이를 위해서는 .vmoptions 파일에 다음 속성을 추가하십시오:

```
-Dcustom.help.url=https://www.internal.website.com
-Dcustom.help.toolBarText=Internal#help
-Dcustom.help.actionName=Show internal help
```

세 가지 속성 모두 UI에 동작을 표시하려면 정의되어야 합니다. `custom.help.toolBarText` 속성은 툴바에 표시되는 텍스트입니다. 간결해야 하며, 위의 예와 같이 # 구분자를 사용하여 두 번째 줄을 추가할 수 있습니다.

.vmoptions 파일의 위치는 Windows 및 Linux에서는 `<JProfiler >/bin/jprofiler.vmoptions`에 있으며, macOS에서는 `/Applications/JProfiler.app/Contents/vmoptions.txt`에 있습니다. 추가로, Windows에서는 `%USERPROFILE%\.jprofiler15\jprofiler.vmoptions`, Linux에서는 `$HOME/.jprofiler15/jprofiler.vmoptions`, macOS에서는 `$HOME/Library/Preferences/jprofiler.vmoptions`에 사용자 쓰기 가능한 위치가 있습니다.

F.4 시작 시 프로파일링 설정 설정하기

프로파일링 에이전트가 녹화를 시작하기 전에 프로파일링 설정이 설정되어야 합니다. 이는 JProfiler UI에 연결할 때 발생합니다. 특정 상황에서는 프로파일링 에이전트가 시작 시 프로파일링 설정을 알아야 할 필요가 있습니다. 주요 사용 사례는 다음과 같습니다:

- **오프라인 프로파일링**

  트리거 또는 API를 사용하여 데이터를 기록하고 스냅샷을 저장합니다. 이 모드에서는 JProfiler GUI가 연결할 수 없습니다. 자세한 내용은 오프라인 프로파일링에 대한 도움말 주제 [p. 122]를 참조하십시오.

- **헤드리스 머신에서 jpcontroller로 프로파일링**

  명령줄 유틸리티 jpcontroller [p. 237]를 사용하여 JProfiler GUI 대신 데이터를 기록하고 스냅샷을 상호작용적으로 또는 비상호작용적 명령 파일로 저장할 수 있습니다. 그러나 jpcontroller는 프로파일링 설정을 구성할 수 있는 기능이 없으므로 사전에 설정해야 합니다.

- **이전 OpenJ9 및 IBM JVM에 원격 attach**

  8u281, 11.0.11 및 Java 17 이전의 OpenJ9 및 IBM JVM은 프로파일된 프로세스의 안정성을 위협하지 않고 클래스를 재정의할 수 있는 기능이 없으므로 시작 시 프로파일링 설정을 설정해야 합니다. JProfiler의 원격 통합 마법사의 "Profiled JVM" 단계에서 JVM의 유형을 묻고, Older OpenJ9 and IBM JVMs를 선택하면 마법사가 아래에서 논의된 옵션을 추가합니다.

일반적으로 시작 시 프로파일링 설정을 설정하는 것이 가장 효율적인 운영 모드입니다. 이는 수행해야 할 클래스 재정의의 수가 가장 적기 때문입니다. 편리함이 줄어드는 것이 문제가 되지 않는다면, 모든 종류의 프로파일링 세션에 사용할 수 있습니다.

시작 시 프로파일링 설정 설정하기

통합 마법사를 사용하는 경우, "Local or Remote" 단계에서 원격 컴퓨터에서 옵션을 선택한 다음 "Config synchronization" 단계에서 시작 시 구성 적용 옵션을 선택하십시오. 그러면 마법사가 다음 단락에서 논의된 것과 동일한 옵션을 추가합니다.

프로파일링 에이전트를 로드하기 위해 시작 스크립트에 `-agentpath` VM 매개변수를 추가한 경우, 프로파일링 설정은 다음을 추가하여 설정할 수 있습니다:

```
,config=<구성 파일 경로>,id=<세션 ID>
```

`-agentpath` 매개변수에 추가합니다. 완전한 매개변수는 다음과 같습니다:

```
-agentpath:/path/to/libjprofilerti.so=port=8849,nowait,config=/path/to/config,id=123
```

프로세스가 시작된 후 프로파일링 에이전트를 로드하기 위해 `jpenable`을 사용하는 경우, 상호작용적 실행에서 오프라인 모드를 선택하고 구성 및 ID를 지정할 수 있습니다. 또는 비상호작용적 실행을 위해 `--offline`, `--config` 및 `--id` 인수를 전달하십시오.

구성 파일 준비하기

참조된 구성 파일은 현재 머신의 JProfiler 설치의 구성 파일일 수 있으며, 이 경우 구성 매개변수를 전혀 지정할 필요가 없습니다. JProfiler 구성 파일은 `$HOME/.jprofiler15/jprofiler_config.xml` 또는 `%USERPROFILE%\.jprofiler15\jprofiler_config.xml`에 위치하며 `-agentlib` VM 매개변수의 `config` 옵션에 대한 기본값입니다.

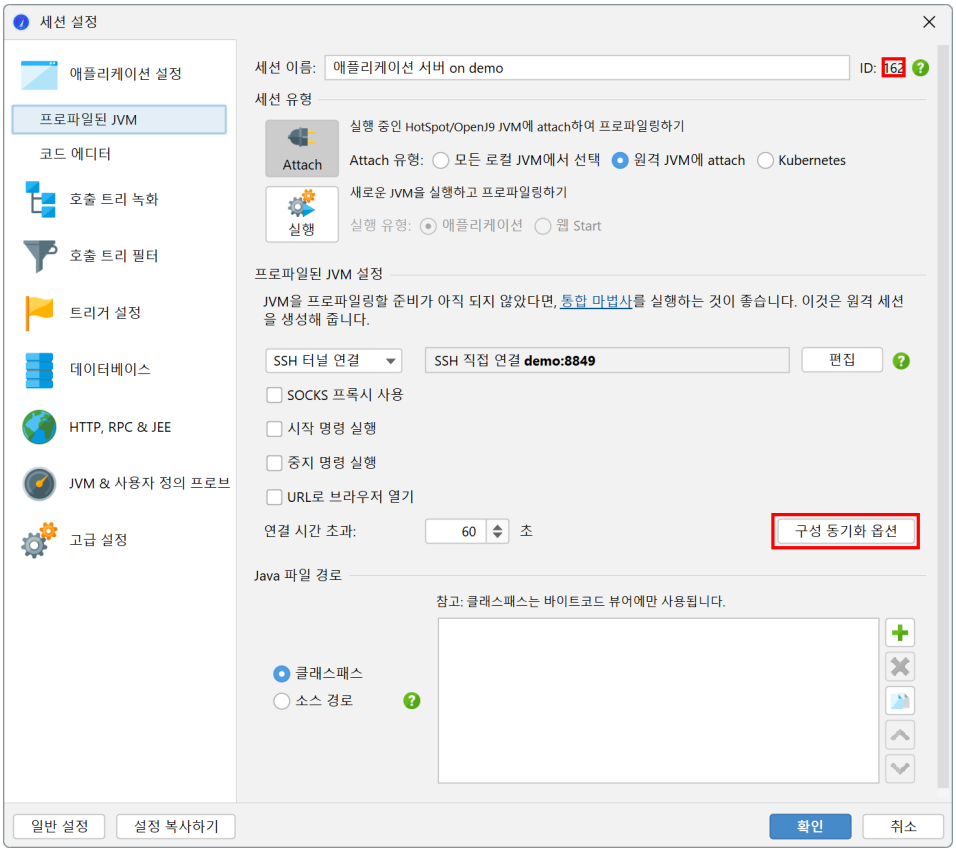
종종 자동화된 프로파일링은 다른 머신에서 수행되어야 하며 로컬 JProfiler 구성 파일을 참조할 수 없습니다. 이 경우 로컬 머신의 JProfiler UI에서 프로파일링 설정으로 세션을 준비하고 세션->세션 설정 내보내기를 통해 내보내고 JProfiler가 실행 중인 머신으로 전송할 수 있습니다.

세션 ID는 세션 설정 대화 상자의 "애플리케이션 설정" 탭의 오른쪽 상단 모서리에서 볼 수 있습니다 (아래 스크린샷 참조). 내보낸 파일에 하나의 세션만 포함된 경우 `id` 매개변수를 지정할 필요가 없습니다.

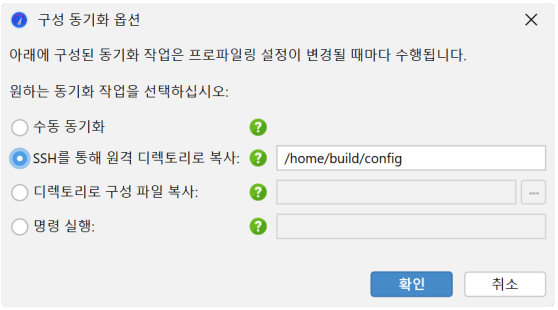
구성 파일 동기화하기

초기 설정을 완료한 후, 향후 프로파일링 실행을 위해 프로파일링 설정을 조정하고자 할 수 있습니다. 이를 위해서는 구성 파일을 수정할 때마다 원격 머신에 복사해야 합니다.

JProfiler의 원격 세션에는 이 프로세스를 자동화할 수 있는 "Config synchronization" 기능이 있습니다.



세션이 SSH를 통해 시작된 경우, SSH를 통해 구성 파일을 원격 머신으로 직접 복사할 수 있습니다. 그렇지 않은 경우, 여전히 구성 파일을 원격 머신에 마운트될 수 있는 로컬 디렉토리에 복사할 수 있습니다. 마지막으로, 구성 파일을 다른 방법으로 복사하기 위해 임의의 명령을 실행할 수 있습니다.



G 명령줄 참조

G.1 프로파일링을 위한 명령줄 실행 파일

JProfiler는 프로파일링 에이전트를 설정하고 명령줄에서 프로파일링 작업을 제어하기 위한 여러 명령줄 도구를 포함하고 있습니다.

실행 중인 JVM에 프로파일링 에이전트 로드하기

명령줄 유틸리티 `bin/jpenable`을 사용하면 버전 6 이상인 모든 실행 중인 JVM에 프로파일링 에이전트를 로드할 수 있습니다. 명령줄 인수를 사용하여 사용자 입력 없이 프로세스를 자동화할 수 있습니다. 지원되는 인수는 다음과 같습니다:

```
사용법: jpenable [옵션]

jpenable은 선택된 로컬 JVM에서 프로파일링 에이전트를 시작하여 다른 컴퓨터에서
연결할 수 있도록 합니다. JProfiler GUI가 로컬에서 실행 중인 경우, 이 실행 파일을
실행하는 대신 JProfiler GUI에서 직접 attach할 수 있습니다.

* 인수가 주어지지 않으면, jpenable은 아직 프로파일되지 않은 로컬 JVM을
  검색하려고 시도하며, 명령줄에서 필요한 모든 입력을 요청합니다.
* 다음 인수와 함께 명령줄에서 사용자 입력을 부분적으로 또는 완전히 제공할 수
  있습니다:

  -d  --pid=<PID>     프로파일링할 JVM의 PID
  -n  --noinput       어떤 경우에도 사용자 입력을 요청하지 않음
  -h  --help          이 도움말 표시
      --options=<OPT> 에이전트에 전달되는 디버깅 옵션

  GUI 모드: (기본값)
  -g  --gui           JProfiler GUI를 사용하여 JVM에 attach합니다
  -p  --port=<nnnnn>  JProfiler GUI에서의 연결을 위해 프로파일링 에이전트가
                      수신 대기할 포트
  -a  --address=<IP>  프로파일링 에이전트가 수신 대기할 주소입니다. 이
                      매개변수가 없으면, localhost에서만 attach할 수 있습니다. 모든
                      주소에서 수신 대기하려면 0.0.0.0을 사용하십시오.

  오프라인 모드:
  -o  --offline       JVM을 오프라인 모드에서 프로파일링합니다
  -c  --config=<PATH> 프로파일링 설정이 포함된 구성 파일의 경로
  -i  --id=<ID>       구성 파일의 세션 ID입니다. 구성 파일에 단일 세션만
                      포함된 경우 필요하지 않습니다.

JVM은 jpenable과 동일한 사용자로 실행되어야 하며, 그렇지 않으면
JProfiler가 연결할 수 없습니다.
예외는 jpenable로 대화형으로 나열할 때 로컬 시스템 계정에서 실행되는 Windows 서비스입니다.
```

HPROF 스냅샷 저장하기

힙 스냅샷만 필요하다면, 프로파일링 에이전트를 VM에 로드하지 않고 HPROF 스냅샷 [p. 195]을 저장하는 `bin/jpdump` 명령줄 도구를 사용하는 것을 고려해 보십시오:

```
사용법: jpdump [옵션]

jpdump는 로컬에서 실행 중인 JVM의 힙을 파일로 덤프합니다.
Hotspot VM은 HPROF 파일을 생성하고, OpenJ9 VM은 PHD 파일을 생성합니다.
HPROF 및 PHD 파일은 JProfiler GUI에서 열 수 있습니다.

* 인수가 주어지지 않으면, jpdump는 로컬에서 실행 중인 모든 JVM을 나열합니다.
```

```
* 다음 인수로 명령줄에서 사용자 입력을 부분적으로 또는 완전히 제공할 수 있습니다:
  전체 사용자 입력을 명령줄에서 제공할 수 있습니다:

  -p  --pid=<PID>    힙을 덤프할 JVM의 PID
                     PID가 지정되면 추가 질문은 없습니다.
  -a  --all          모든 객체를 저장합니다. 지정하지 않으면, 활성 객체만
                     덤프됩니다
  -f  --file=<PATH>  덤프 파일의 경로. 지정하지 않으면, 덤프 파일은
                     <VM 이름>.hprof로 현재 디렉토리에 작성됩니다.
                     파일이 이미 존재하면, 숫자가 추가됩니다.
  -h  --help         이 도움말 표시

JVM은 jpdump와 동일한 사용자로 실행되어야 하며, 그렇지 않으면
JProfiler가 연결할 수 없습니다.
예외는 jpdump로 대화형으로 나열할 때 로컬 시스템 계정에서 실행되는 Windows 서비스입니다.
```

이는 프로파일링 에이전트를 로드하고 JProfiler 힙 스냅샷을 저장하는 것보다 오버헤드가 적습니다. 또한, 프로파일링 에이전트는 절대 언로드될 수 없기 때문에 이 방법은 프로덕션에서 실행 중인 JVM에 적합합니다.

프로파일링 에이전트 제어하기

`bin/jpcontroller` 실행 파일을 인수 없이 시작하면 로컬 머신의 프로파일된 JVM에 연결을 시도합니다. 여러 프로파일된 JVM이 발견되면 목록에서 하나를 선택할 수 있습니다.

`jpcontroller`는 프로파일링 설정이 설정된 JVM에만 연결할 수 있으므로, `-agentpath` VM 매개변수에 대해 "nowait" 옵션으로 JVM이 시작된 경우에는 작동하지 않습니다. 이 옵션은 통합 마법사의 "시작 모드" 화면에서 즉시 시작, 나중에 JProfiler GUI로 연결 라디오 버튼을 선택하고 아직 JProfiler GUI가 에이전트에 연결되지 않은 경우 설정됩니다. `jpenable`을 `--offline` 인수 없이 사용하는 경우에도 `jpcontroller`가 프로파일된 프로세스에 연결하기 전에 JProfiler GUI의 연결이 필요합니다.

원격 컴퓨터의 프로세스에 연결하거나 JVM이 버전 6 이상의 HotSpot JVM이 아닌 경우, VM 매개변수 `-Djprofiler.jmxServerPort=[port]`를 프로파일된 JVM에 전달해야 합니다. 해당 포트에서 MBean 서버가 게시되며, 선택한 포트를 `jpcontroller`의 인수로 지정할 수 있습니다. 추가 VM 매개변수 `-Djprofiler.jmxPasswordFile=[file]`을 사용하여 `user password` 형식의 키-값 쌍이 포함된 속성 파일을 지정하여 인증을 설정할 수 있습니다 (공백 또는 탭으로 구분). 이러한 VM 매개변수는 `com.sun.management.jmxremote.port` VM 매개변수에 의해 재정의된다는 점에 유의하십시오.

JMX 서버의 명시적 설정을 통해 명령줄 컨트롤러를 사용하여 `jpcontroller host:port`를 호출하여 원격 서버에 연결할 수 있습니다. 원격 컴퓨터가 IP 주소를 통해서만 접근 가능한 경우, `-Djava.rmi.server.hostname=[IP address]`을 원격 VM의 VM 매개변수로 추가해야 합니다.

기본적으로, `jpcontroller`는 대화형 명령줄 유틸리티이지만, 수동 입력 없이 프로파일링 세션을 자동화할 수도 있습니다. 자동화된 호출은 `[pid | host:port]`를 전달하여 프로파일된 JVM을 선택하고 `--non-interactive` 인수를 전달합니다. 또한, 명령 목록은 stdin 또는 `--command-file` 인수로 지정된 명령 파일에서 읽습니다. 각 명령은 새 줄에서 시작하며, 빈 줄이나 "#" 주석 문자로 시작하는 줄은 무시됩니다.

이 비대화형 모드의 명령은 JProfiler MBean [(1)]의 메서드 이름과 동일합니다. 이들은 동일한 수의 매개변수를 요구하며, 공백으로 구분됩니다. 문자열에 공백이 포함된 경우, 반드시 큰따옴표로 둘러싸야 합니다. 추가로, `sleep <seconds>` 명령이 제공되어 몇 초 동안 일시 중지할 수 있습니다. 이를 통해 녹화를 시작하고, 잠시 기다린 후 디스크에 스냅샷을 저장할 수 있습니다.

프로파일링 설정은 프로파일링 에이전트에 설정되어야 한다는 점에 유의하십시오. 이는 JProfiler UI에 연결할 때 발생합니다. JProfiler UI에 절대 연결하지 않는 경우, 시작 명령에서 수동으로 설정하거나 jpenable을

[(1)] https://www.ej-technologies.com/resources/jprofiler/help/api/javadoc/com/jprofiler/api/agent/mbean/RemoteControllerMBean.html

사용하여 설정해야 합니다. 자세한 내용은 시작 시 프로파일링 설정 설정에 대한 도움말 주제 [p. 235]를 참조하십시오.

jpcontroller의 지원되는 인수는 아래에 나와 있습니다:

```
사용법: jpcontroller [옵션] [host:port | pid]

* 인수가 주어지지 않으면, jpcontroller는 프로파일된
   로컬 JVM을 검색하려고 시도합니다
* 단일 숫자가 지정되면, jpcontroller는 프로세스 ID [pid]를 가진 JVM에
   연결을 시도합니다. 해당 JVM이 프로파일되지 않은 경우, jpcontroller는
   연결할 수 없습니다. 그런 경우, 먼저 jpenable 유틸리티를 사용하십시오.
* 그렇지 않으면, jpcontroller는 "host:port"에 연결하며, 여기서 port는
   프로파일된 JVM의 VM 매개변수 -Djprofiler.jmxServerPort=[port]에
   지정된 값입니다.

사용 가능한 옵션:
  -n --non-interactive      명령 목록이 stdin에서 읽혀지는 자동화된 세션을 실행합니다.
  -f --command-file=<PATH>  stdin 대신 파일에서 명령을 읽습니다. 이는
                            --non-interactive와 함께 사용할 수 있습니다.

비대화형 명령의 구문:
  RemoteControllerMBean (https://bit.ly/2DimDN5)의 javadoc을 참조하여
  작업 목록을 확인하십시오. 매개변수는 공백으로 구분되며, 공백이 포함된 경우
  인용부호로 묶어야 합니다. 예를 들어:

    addBookmark "Hello world"
    startCPURecording true
    startProbeRecording builtin.JdbcProbe true true
    sleep 10
    stopCPURecording
    stopProbeRecording builtin.JdbcProbe
    saveSnapshot /path/to/snapshot.jps

  sleep <seconds> 명령은 지정된 초 수만큼 일시 중지합니다.
  빈 줄과 #으로 시작하는 줄은 무시됩니다.
```

G.2 스냅샷 작업을 위한 명령줄 실행 파일

스냅샷을 프로그래밍 방식으로 저장하기 위해 오프라인 프로파일링 [p. 122]을 사용할 때, 프로그래밍 방식으로 해당 스냅샷에서 데이터나 보고서를 추출해야 할 수도 있습니다. JProfiler는 스냅샷에서 뷰를 내보내기 위한 하나의 명령줄 실행 파일과 스냅샷을 비교하기 위한 또 다른 명령줄 실행 파일을 제공합니다.

스냅샷에서 뷰 내보내기

실행 파일 `bin/jpexport`는 다양한 형식으로 뷰 데이터를 내보냅니다. `-help` 옵션과 함께 실행하면 사용 가능한 뷰 이름과 뷰 옵션에 대한 정보를 얻을 수 있습니다. 간결성을 위해 아래 출력에서는 중복된 도움말 텍스트가 생략되었습니다.

```
사용법: jpexport "스냅샷 파일" [글로벌 옵션]
              "뷰 이름" [옵션] "출력 파일"
              "뷰 이름" [옵션] "출력 파일" ...

"snapshot file"은 다음 확장자를 가진 스냅샷 파일입니다:
           .jps, .hprof, .hpz, .phd, .jfr
      "뷰 이름"은(는) 아래 나열된 뷰 이름 중 하나입니다.
      [options]는 -option=value 형식의 옵션 목록입니다.
      "output file"은(는) 내보내기의 출력 파일입니다

글로벌 옵션:
    -obfuscator=none|proguard|yguard
        선택한 난독화 도구에 대해 디옵스큐레이트합니다. 기본값은 "none"이며, 다른 값을 사용하려면
mappingFile 옵션을
        지정해야 합니다.
    -mappingfile=<file>
        선택한 난독화 도구에 대한 매핑 파일.
    -outputdir=<출력 디렉토리>
        뷰의 출력 파일이 상대 경로 파일인 경우에 사용할 기본 디렉터리입니다.
    -ignoreerrors=true|false
        옵션을 설정할 수 없는 뷰에서 발생하는 오류를 무시하고 다음 뷰로 계속 진행하십시오. 기본값은
"false"이며, 이는 첫
        번째 오류가 발생하면 내보내기가 중단됨을 의미합니다.
    -csvseparator=<구분자 문자>
        CSV 내보내기의 필드 구분 문자입니다. 기본값은 ','입니다.
    -bitmap=false|true
        적절한 경우, 주요 콘텐츠에 대해 SVG 대신 비트맵 이미지를 내보내십시오. 기본값은 false입니다.

사용 가능한 뷰 이름 및 옵션:
* TelemetryHeap, TelemetryObjects, TelemetryThroughput, TelemetryGC,
  TelemetryClasses, TelemetryThreads, TelemetryCPU
  ??:
    -format=html|csv
        내보낸 파일의 출력 형식을 결정합니다. 존재하지 않으면, 출력 파일의 확장자로부터 내보내기 형식이
결정됩니다.
    -minwidth=<픽셀 수>
        graph의 최소 너비(픽셀 단위)입니다. 기본값은 800입니다.
    -minheight=<픽셀 수>
        픽셀 단위로 graph의 최소 높이입니다. 기본값은 600입니다.

* Bookmarks, ThreadMonitor, CurrentMonitorUsage, MonitorUsageHistory
  ??:
    -format=html|csv

* AllObjects
  ??:
    -format=html|csv
    -viewfilters=<쉼표로 구분된 목록>
        내보내기를 위한 뷰 필터를 설정합니다. 뷰 필터를 설정하면, 내보내기된 뷰에 지정된 패키지와 그 하
위 패키지만 표시됩니다.
    -viewfiltermode=startswith|endswith|contains|equals
```

```
        뷰 필터 모드를 설정합니다. 기본값은 "contains"입니다.
    -viewfilteroptions=casesensitive
        뷰 필터에 대한 Boolean 옵션입니다. 기본적으로 옵션이 설정되어 있지 않습니다.
    -aggregation=class|package|component
        내보내기를 위한 집계 수준을 선택합니다. 기본값은 classes입니다.
    -expandpackages=true|false
       패키지 집계 수준에서 패키지 노드를 확장하여 포함된 클래스를 표시합니다. 기본값은 "false"입니다.
 다른 집계 수준 및
        csv 출력 형식에서는 효과가 없습니다.

* RecordedObjects
  AllObjects? ????? ?? ??? ???? ????:
    -liveness=live|gc|all
        내보내기를 위한 활성도 모드를 선택합니다. 즉, 활성 객체, 가비지 수집된 객체 또는 둘 다 표시할지
를 선택합니다. 기본값은
        활성 객체입니다.

* AllocationTree
  ??:
    -format=html|xml
    -viewfilters=<쉼표로 구분된 목록>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive
    -aggregation=method|class|package|component
        내보내기를 위한 집계 수준을 선택합니다. 기본값은 methods입니다.
    -class=<fully qualified class name>
        클래스에 대한 할당 데이터를 계산할 클래스를 지정합니다. 비어 있으면 모든 클래스의 할당이 표시됩
니다. package 옵션과
        함께 사용할 수 없습니다.
    -package=<fully qualified package name>
        패키지에 대한 할당 데이터를 계산할 패키지를 지정합니다. 비어 있는 경우, 모든 패키지의 할당이 표
시됩니다. 패키지 이름에
        .**를 추가하면 패키지를 재귀적으로 선택합니다. class 옵션과 함께 사용할 수 없습니다.
    -liveness=live|gc|all

* AllocationHotSpots
  ??:
    -format=html|csv|xml
    -viewfilters=<쉼표로 구분된 목록>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive
    -aggregation=method|class|package|component
    -class=<fully qualified class name>
    -package=<fully qualified package name>
    -liveness=live|gc|all
    -unprofiledclasses=separately|addtocalling
        프로파일되지 않은 클래스를 별도로 표시할지 호출 클래스에 추가할지를 선택합니다. 기본값은 프로파일
되지 않은 클래스를 별도로
        표시하는 것입니다.
    -valuesummation=self|total
        핫스팟의 시간이 어떻게 계산되는지를 결정합니다. 기본값은 "자체"입니다.
    -expandbacktraces=true|false
        HTML 또는 XML 형식으로 백트레이스를 확장합니다. 기본값은 "false"입니다.

* ClassTracker
  TelemetryHeap? ????? ?? ??? ???? ????:
    -class
        추적된 클래스입니다. 누락된 경우, 첫 번째로 추적된 클래스가 내보내집니다.

* CallTree
  ??:
    -format=html|xml
    -viewfilters=<쉼표로 구분된 목록>
    -viewfiltermode=startswith|endswith|contains|equals
```

```
    -viewfilteroptions=casesensitive
    -aggregation=method|class|package|component
    -threadgroup=<스레드 그룹의 이름>
        내보내기를 위한 스레드 그룹을 선택합니다. "thread"도 지정하면 스레드는 이 스레드 그룹에서만 검
색되며, 그렇지 않으면
        전체 스레드 그룹이 표시됩니다.
    -thread=<스레드 이름>
        스레드를 내보내기에 선택합니다. 기본적으로 호출 트리는 모든 스레드에 대해 병합됩니다.
    -threadstatus=all|running|waiting|blocking|netio
        내보내기를 위한 스레드 상태를 선택합니다. 기본값은 "running"입니다.

* HotSpots
  ??:
    -format=html|csv|xml
    -viewfilters=<쉼표로 구분된 목록>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive
    -aggregation=method|class|package|component
    -threadgroup=<스레드 그룹의 이름>
    -thread=<스레드 이름>
    -threadstatus=all|running|waiting|blocking|netio
    -expandbacktraces=true|false
    -unprofiledclasses=separately|addtocalling
    -valuesummation=self|total

* OutlierDetection
  ??:
    -format=html|csv
    -threadstatus=all|running|waiting|blocking|netio
    -viewfilters=<쉼표로 구분된 목록>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive

* Complexity
  ??:
    -format=html|csv|properties
    -fit=best|constant|linear|quadratic|cubic|exponential|logarithmic|n_log_n
        내보내야 할 적합도입니다. 기본값은 "best"입니다. 곡선 피팅 데이터는 CSV로 내보내지 않습니다.
    -method=<method name>
        복잡성 그래프를 내보낼 메서드 이름입니다. 지정하지 않으면 첫 번째 메서드가 내보내집니다. 그렇지
않으면 주어진 텍스트로
        시작하는 첫 번째 메서드 이름이 내보내집니다.
    -width=<픽셀 수>
    -height=<픽셀 수>

* ThreadHistory
  TelemetryHeap? ????? ??? ???????:
    -format=html

* MonitorUsageStatistics
  ??:
    -format=html|csv
    -type=monitors|threads|classes
        모니터 통계를 계산할 엔티티를 선택합니다. 기본값은 "monitors"입니다.

* ProbeTimeLine
  ThreadHistory? ????? ?? ??? ???? ????:
    -probeid=<id>
        프로브를 내보내야 하는 내부 ID입니다. 모든 사용 가능한 내장 프로브를 나열하고 사용자 정의 프로
브 이름에 대한 설명을
        보려면 "jpexport --listProbes"을 실행하세요.

* ProbeControlObjects
  ??:
```

```
    -probeid=<id>
    -format=html|csv

* ProbeCallTree
  ??:
    -probeid=<id>
    -format=html|xml
    -viewfilters=<쉼표로 구분된 목록>
    -viewfiltermode=startswith|endswith|contains|equals|wildcard|regex
    -viewfilteroptions=exclude,casesensitive
    -aggregation=method|class|package|component
    -threadgroup=<스레드 그룹의 이름>
    -thread=<스레드 이름>
    -threadstatus=all|running|waiting|blocking|netio
        내보내기를 위한 스레드 상태를 선택합니다. 기본값은 "all"입니다.

* ProbeHotSpots
  ProbeCallTree? ????? ????? ??? ??? ????:
    -format=html|csv|xml
    -expandbacktraces=true|false

* ProbeTelemetry
  TelemetryHeap? ????? ?? ??? ???? ????:
    -probeid=<id>
    -telemetrygroup
        텔레메트리 그룹의 1 기반 인덱스를 설정합니다. 이는 프로브 텔레메트리 뷰 위의 드롭다운 목록에서
볼 수 있는 항목을
        참조합니다. 기본값은 "1"입니다.

* ProbeEvents
  ??:
    -probeid=<id>
    -format=html|csv|xml

* ProbeTracker
  TelemetryHeap? ????? ?? ??? ???? ????:
    -probeid=<id>
    -index=<number>
        드롭다운 목록에서 추적된 요소를 포함하는 항목의 0부터 시작하는 인덱스를 설정합니다. 기본값은 0입
니다.
```

스냅샷 비교

실행 파일 `bin/jpcompare`는 다른 스냅샷을 비교 [p. 127]하고 이를 HTML 또는 기계 판독 가능한 형식으로 내보냅니다. `-help` 출력은 아래에 재현되어 있으며, 중복된 설명은 생략되었습니다.

```
사용법: jpcompare "스냅샷 파일"[,"스냅샷 파일",...] [글로벌 옵션]
              "비교 이름" [옵션] "출력 파일"
              "비교 이름" [옵션] "출력 파일" ...

"snapshot file"은 다음 확장자를 가진 스냅샷 파일입니다:
           .jps, .hprof, .hpz, .phd, .jfr
      "비교 이름"은(는) 아래 나열된 비교 이름 중 하나입니다.
      [options]는 -option=value 형식의 옵션 목록입니다.
      "output file"은(는) 내보내기의 출력 파일입니다

글로벌 옵션:
    -outputdir=<출력 디렉토리>
        비교의 출력 파일이 상대 경로 파일인 경우에 사용할 기본 디렉터리입니다.
    -ignoreerrors=true|false
        옵션을 설정할 수 없는 비교에서 발생하는 오류를 무시하고 다음 비교로 계속 진행하십시오. 기본값은
"false"이며, 이는 첫
```

```
        번째 오류가 발생하면 내보내기가 중단됨을 의미합니다.
    -csvseparator=<구분자 문자>
        CSV 내보내기의 필드 구분 문자입니다. 기본값은 ','입니다.
    -bitmap=false|true
        적절한 경우, 주요 콘텐츠에 대해 SVG 대신 비트맵 이미지를 내보내십시오. 기본값은 false입니다.
    -sortbytime=false|true
        지정된 스냅샷 파일을 수정 시간으로 정렬합니다. 기본값은 false입니다.
    -listfile=<filename>
        스냅샷 파일의 경로를 포함하는 파일을 읽습니다. 각 줄에는 하나의 스냅샷 파일이 포함되어 있습니다.


사용 가능한 비교 이름 및 옵션:
* Objects
  ??:
    -format=html|csv
        내보낸 파일의 출력 형식을 결정합니다. 존재하지 않으면, 출력 파일의 확장자로부터 내보내기 형식이
결정됩니다.
    -viewfilters=<쉼표로 구분된 목록>
        내보내기를 위한 뷰 필터를 설정합니다. 뷰 필터를 설정하면, 내보내기된 뷰에 지정된 패키지와 그 하
위 패키지만 표시됩니다.
    -viewfiltermode=startswith|endswith|contains|equals
        뷰 필터 모드를 설정합니다. 기본값은 "contains"입니다.
    -viewfilteroptions=casesensitive
        뷰 필터에 대한 Boolean 옵션입니다. 기본적으로 옵션이 설정되어 있지 않습니다.
    -aggregation=class|package|component
        내보내기를 위한 집계 수준을 선택합니다. 기본값은 classes입니다.
    -liveness=live|gc|all
        내보내기를 위한 활성도 모드를 선택합니다. 즉, 활성 객체, 가비지 수집된 객체 또는 둘 다 표시할지
를 선택합니다. 기본값은
        활성 객체입니다.
    -objects=recorded|heapwalker|all
        기록된 객체, 힙 워커의 객체 또는 모든 객체 덤프의 객체 수를 비교합니다. 기본값은 .jps 파일의
경우 기록된 객체이고,
        HPROF/PHD 파일의 경우 힙 워커입니다.
    -dumpselection=first|last|label
        비교 계산에 사용되는 모든 객체 덤프입니다. 기본값은 마지막 값입니다.
    -label
        dumpselection이(가) 'label'로 설정된 경우, 비교를 계산해야 하는 레이블의 이름입니다.

* AllocationHotSpots
  ??:
    -format=html|csv
    -viewfilters=<쉼표로 구분된 목록>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive
    -aggregation=method|class|package|component
        내보내기를 위한 집계 수준을 선택합니다. 기본값은 methods입니다.
    -liveness=live|gc|all
    -unprofiledclasses=separately|addtocalling
        프로파일되지 않은 클래스를 별도로 표시할지 호출 클래스에 추가할지를 선택합니다. 기본값은 프로파일
되지 않은 클래스를 별도로
        표시하는 것입니다.
    -valuesummation=self|total
        핫스팟의 시간이 어떻게 계산되는지를 결정합니다. 기본값은 "자체"입니다.
    -classselection
        특정 클래스 또는 패키지에 대한 비교를 계산합니다. 패키지는 단일 패키지의 경우 '.*'를, 재귀 패키
지의 경우 '.**'를
        추가하여 지정합니다. 패키지를 와일드카드와 함께 지정하십시오, 예를 들어 'java.awt.*'.

* AllocationTree
  ??:
    -format=html|xml
    -viewfilters=<쉼표로 구분된 목록>
    -viewfiltermode=startswith|endswith|contains|equals
```

```
    -viewfilteroptions=casesensitive
    -aggregation=method|class|package|component
    -liveness=live|gc|all
    -classselection

* HotSpots
  ??:
    -format=html|csv
    -viewfilters=<쉼표로 구분된 목록>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive
    -firstthreadselection
        특정 스레드 또는 스레드 그룹에 대한 비교를 계산합니다. 스레드 그룹을 'group.*'처럼 지정하고 특
정 스레드 그룹 내의
        스레드를 'group.thread'처럼 지정하세요. 스레드 이름의 점은 백슬래시로 이스케이프 처리하세요.
    -secondthreadselection
        특정 스레드 또는 스레드 그룹에 대한 비교를 계산합니다. 'firstthreadselection'이(가) 설정된
 경우에만
        사용할 수 있습니다. 비어 있는 경우, 'firstthreadselection'에 대한 동일한 값이 사용됩니다.
 스레드 그룹을
        'group.*'처럼 지정하고 특정 스레드 그룹 내의 스레드를 'group.thread'처럼 지정하세요. 스레
드 이름의 점은
        백슬래시로 이스케이프 처리하세요.
    -threadstatus=all|running|waiting|blocking|netio
        내보내기를 위한 스레드 상태를 선택합니다. 기본값은 "running"입니다.
    -aggregation=method|class|package|component
    -differencecalculation=total|average
        호출 시간에 대한 차이 계산 방법을 선택합니다. 기본값은 총 시간입니다.
    -unprofiledclasses=separately|addtocalling
    -valuesummation=self|total

* CallTree
  ??:
    -format=html|xml
    -viewfilters=<쉼표로 구분된 목록>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive
    -firstthreadselection
    -secondthreadselection
    -threadstatus=all|running|waiting|blocking|netio
    -aggregation=method|class|package|component
    -differencecalculation=total|average

* TelemetryHeap
  ??:
    -format=html|csv
    -minwidth=<픽셀 수>
        graph의 최소 너비(픽셀 단위)입니다. 기본값은 800입니다.
    -minheight=<픽셀 수>
        픽셀 단위로 graph의 최소 높이입니다. 기본값은 600입니다.
    -valuetype=current|maximum|bookmark
        스냅샷마다 계산되는 값의 유형입니다. 기본값은 현재 값입니다.
    -bookmarkname
        valuetype이(가) 'bookmark'로 설정된 경우, 값을 계산해야 하는 북마크의 이름입니다.
    -measurements=maximum,free,used
        비교 그래프에 표시되는 측정값입니다. 여러 값을 쉼표로 연결하십시오. 기본값은 'used'입니다.
    -memorytype=heap|nonheap
        분석할 메모리 유형입니다. 기본값은 'heap'입니다.
    -memorypool
        특수 메모리 풀을 분석하려면 이 매개변수에 이름을 지정할 수 있습니다. 기본값은 비어 있으며, 즉
특수 메모리 풀이 없습니다.

* TelemetryObjects
  ??:
```

```
    -format=html|csv
    -minwidth=<픽셀 수>
    -minheight=<픽셀 수>
    -valuetype=current|maximum|bookmark
    -bookmarkname
    -measurements=total,nonarrays,arrays
        비교 그래프에 표시되는 측정값입니다. 여러 값을 쉼표로 연결하십시오. 기본값은 'total'입니다.

* TelemetryClasses
  TelemetryObjects? ????? ??? ???????:
    -measurements=total,filtered,unfiltered

* TelemetryThreads
  TelemetryObjects? ????? ??? ???????:
    -measurements=total,runnable,blocked,netio,waiting

* ProbeHotSpots
  ??:
    -format=html|csv
    -viewfilters=<쉼표로 구분된 목록>
    -viewfiltermode=startswith|endswith|contains|equals|wildcard|regex
    -viewfilteroptions=exclude,casesensitive
    -firstthreadselection
    -secondthreadselection
    -threadstatus=all|running|waiting|blocking|netio
    -aggregation=method|class|package|component
    -differencecalculation=total|average
    -probeid=<id>
        프로브를 내보내야 하는 내부 ID입니다. 모든 사용 가능한 내장 프로브를 나열하고 사용자 정의 프로
브 이름에 대한 설명을
        보려면 "jpexport --listProbes"을 실행하세요.

* ProbeCallTree
  ProbeHotSpots? ????? ??? ???????:
    -format=html|xml

* ProbeTelemetry
  TelemetryObjects? ????? ????? ??? ??? ????:
    -measurements
        측정값 비교 그래프에 표시되는 텔레메트리 그룹의 1 기반 인덱스입니다. 여러 값을 쉼표로 연결하여
"1,2"와 같이
        입력하세요. 기본값은 모든 측정값을 표시하는 것입니다.
    -probeid=<id>
    -telemetrygroup
        텔레메트리 그룹의 1 기반 인덱스를 설정합니다. 이는 프로브 텔레메트리 뷰 위의 드롭다운 목록에서
볼 수 있는 항목을
        참조합니다. 기본값은 "1"입니다.
```

자동 출력 형식

대부분의 뷰와 비교는 여러 출력 형식을 지원합니다. 기본적으로 출력 형식은 출력 파일의 확장자로부터 유추됩니다:

- **.html**

  뷰 또는 비교가 HTML 파일로 내보내집니다. HTML 페이지에 사용된 이미지를 포함하는 `jprofiler_images` 라는 디렉토리가 생성됩니다.

- **.csv**

  데이터는 첫 번째 줄에 열 이름이 포함된 CSV 데이터로 내보내집니다.

Microsoft Excel을 사용할 때, CSV는 안정적인 형식이 아닙니다. Windows의 Microsoft Excel은 지역 설정에서 구분 문자를 가져옵니다. JProfiler는 소수점 구분자로 쉼표를 사용하는 지역에서는 세미콜론을 구분 문자로 사용하고, 소수점 구분자로 점을 사용하는 지역에서는 쉼표를 사용합니다. CSV 구분 문자를 재정의해야 하는 경우, 전역 `csvseparator` 옵션을 설정하여 그렇게 할 수 있습니다.

- **.xml**
  데이터는 XML로 내보내집니다. 데이터 형식은 자체 설명적입니다.

다른 확장자를 사용하고 싶다면, `format` 옵션을 사용하여 출력 형식의 선택을 재정의할 수 있습니다.

스냅샷 분석

생성된 스냅샷에 힙 덤프가 포함되어 있는 경우, `bin/jpanalyze` 실행 파일을 사용하여 힙 덤프 분석을 미리 준비 [p. 77]할 수 있습니다. JProfiler GUI에서 스냅샷을 열면 매우 빠르게 열립니다. 도구의 사용 정보는 아래에 표시됩니다:

```
사용법: jpanalyze [옵션] "스냅샷 파일" ["스냅샷 파일" ...]

"snapshot file"은 다음 확장자를 가진 스냅샷 파일입니다:
           .jps, .hprof, .hpz, .phd, .jfr
      [options]는 -option=value 형식의 옵션 목록입니다.

옵션:
    -obfuscator=none|proguard|yguard
        선택한 난독화 도구에 대해 디옵스큐레이트합니다. 기본값은 "none"이며, 다른 값을 사용하려면
mappingFile 옵션을
        지정해야 합니다.
    -mappingfile=<file>
        선택한 난독화 도구에 대한 매핑 파일.
    -removeunreferenced=true|false
        참조되지 않거나 약하게 참조된 객체를 제거해야 하는 경우.
    -retained=true|false
        가장 큰 객체의 유지 크기를 계산합니다. removeunreferenced가 true로 설정됩니다.
    -retainsoft=true|false
        참조되지 않은 객체가 제거될 경우, 소프트 참조를 유지할지 여부를 지정합니다.
    -retainweak=true|false
        참조되지 않은 객체가 제거될 경우, 약한 참조를 유지할지 여부를 지정합니다.
    -retainphantom=true|false
        참조되지 않은 객체가 제거되면, 팬텀 참조를 유지할지 여부를 지정합니다.
    -retainfinalizer=true|false
        참조되지 않은 객체가 제거될 경우, finalizer 참조를 유지할지 여부를 지정합니다.
```

`removeUnreferenced`, `retained` 및 모든 `retain*` 명령줄 옵션은 힙 워커 옵션 대화 상자의 옵션에 해당합니다.

G.3 Gradle Tasks

JProfiler는 Gradle에서 특별한 태스크를 사용하여 프로파일링을 지원합니다. 또한, JProfiler는 스냅샷 작업을 위한 명령줄 실행 파일 [p. 240]에 해당하는 여러 Gradle 태스크를 제공합니다.

Gradle 태스크 사용하기

JProfiler Gradle 태스크를 Gradle 빌드 파일에서 사용 가능하게 하려면, `plugins` 블록을 사용할 수 있습니다.

```
plugins {
    id 'com.jprofiler' version 'X.Y.Z'
}
```

이 목적을 위해 Gradle 플러그인 저장소를 사용하고 싶지 않다면, Gradle 플러그인은 `bin/gradle.jar` 파일에 배포됩니다.

다음으로, JProfiler Gradle 플러그인에 JProfiler가 설치된 위치를 알려야 합니다.

```
jprofiler {
    installDir = file('/path/to/jprofiler/home')
}
```

Gradle에서 프로파일링하기

`com.jprofiler.gradle.JavaProfile` 유형의 태스크를 사용하여 모든 Java 프로세스를 프로파일링할 수 있습니다. 이 클래스는 Gradle의 내장 `JavaExec`를 확장하므로 프로세스를 구성하기 위해 동일한 인수를 사용할 수 있습니다. 테스트를 프로파일링하려면 Gradle `Test` 태스크를 확장하는 `com.jprofiler.gradle.TestProfile` 유형의 태스크를 사용하십시오.

추가 구성 없이, 두 태스크 모두 프로파일링 에이전트가 JProfiler GUI로부터의 연결을 기다리는 기본 포트 8849에서 대기하는 대화형 프로파일링 세션을 시작합니다. 오프라인 프로파일링을 위해서는 아래 표에 나와 있는 몇 가지 속성을 추가해야 합니다.

Attribute	Description	Required
offline	프로파일링 실행이 오프라인 모드에서 수행되어야 하는지 여부입니다.	아니요, `offline`과 `nowait`는 모두 `true`일 수 없습니다.
nowait	프로파일링이 즉시 시작되어야 하는지 아니면 프로파일된 JVM이 JProfiler GUI로부터의 연결을 기다려야 하는지 여부입니다.	
sessionId	프로파일링 설정이 가져와야 하는 세션 ID를 정의합니다. `nowait` 또는 offline이 설정되지 않은 경우에는 효과가 없습니다. 이 경우 프로파일링 세션이 GUI에서 선택됩니다.	필수, 만약 • `offline`이 설정된 경우 • 1.5 JVM에 대해 `nowait`이 설정된 경우
configFile	프로파일링 설정이 읽혀야 하는 구성 파일을 정의합니다.	아니요

Attribute	Description	Required
port	프로파일링 에이전트가 JProfiler GUI로부터의 연결을 대기해야 하는 포트 번호를 정의합니다. 이는 원격 세션 구성에서 구성된 포트와 동일해야 합니다. 설정되지 않거나 0인 경우 기본 포트(8849)가 사용됩니다. `offline`이 설정된 경우에는 GUI로부터의 연결이 없으므로 효과가 없습니다.	아니요
debugOptions	튜닝이나 디버깅 목적으로 추가 라이브러리 매개변수를 전달하려면, 이 속성을 사용하여 할 수 있습니다.	아니요

메인 메서드를 가진 Java 클래스를 프로파일링하는 예제는 아래에 나와 있습니다:

```
task run(type: com.jprofiler.gradle.JavaProfile) {
    mainClass = 'com.mycorp.MyMainClass'
    classpath sourceSets.main.runtimeClasspath
    offline = true
    sessionId = 80
    configFile = file('path/to/jprofiler_config.xml')
}
```

이 태스크의 실행 가능한 예제는 `api/samples/offline` 샘플 프로젝트에서 볼 수 있습니다. 표준 `JavaExec` 태스크와 달리, `JavaProfile` 태스크는 `createProcess()`를 호출하여 백그라운드에서 시작할 수도 있습니다. 이 기능의 데모는 `api/samples/mbean` 샘플 프로젝트에서 확인할 수 있습니다.

프로파일링에 필요한 VM 매개변수가 필요한 경우, `com.jprofiler.gradle.SetAgentpathProperty` 태스크는 `propertyName` 속성으로 구성된 속성에 할당합니다. JProfiler 플러그인을 적용하면 자동으로 `setAgentPathProperty`라는 이름의 이 유형의 태스크가 프로젝트에 추가됩니다. 이전 예제에서 사용될 VM 매개변수를 얻으려면, 단순히 추가하십시오.

```
setAgentPathProperty {
    propertyName = 'profilingVmParameter'
    offline = true
    sessionId = 80
    configFile = file('path/to/jprofiler_config.xml')
}
```

프로젝트에 추가하고 `setAgentPathProperty`에 대한 종속성을 다른 태스크에 추가하십시오. 그런 다음 해당 태스크의 실행 단계에서 프로젝트 속성 `profilingVmParameter`를 사용할 수 있습니다. 다른 태스크 속성에 속성을 할당할 때는 `doFirst {...}` 코드 블록으로 사용을 감싸서 Gradle 실행 단계에 있고 구성 단계에 있지 않음을 확인하십시오.

스냅샷에서 데이터 내보내기

`com.jprofiler.gradle.Export` 태스크는 저장된 스냅샷에서 뷰를 내보내는 데 사용되며 `bin/jpexport` 명령줄 도구 [p. 240]의 인수를 복제합니다. 다음 속성을 지원합니다:

Attribute	Description	Required
snapshotFile	스냅샷 파일의 경로입니다. 이 파일은 `.jps` 확장자를 가진 파일이어야 합니다.	예

Attribute	Description	Required
ignoreErrors	뷰에 대한 옵션을 설정할 수 없을 때 발생하는 오류를 무시하고 다음 뷰로 계속 진행합니다. 기본값은 `false`이며, 첫 번째 오류가 발생하면 내보내기가 종료됩니다.	아니요
csvSeparator	CSV 내보내기의 필드 구분 문자입니다. 기본값은 ","입니다.	아니요
obfuscator	선택한 난독화 도구에 대해 클래스 및 메서드 이름을 디난독화합니다. 기본값은 "none"이며, 다른 값을 사용하려면 `mappingFile` 옵션을 지정해야 합니다. `none`, `proguard` 또는 `yguard` 중 하나입니다.	아니요
mappingFile	선택한 난독화 도구의 매핑 파일입니다. `obfuscator` 속성이 지정된 경우에만 설정할 수 있습니다.	`obfuscator` 속성이 지정된 경우에만

내보내기 태스크에서는 `views` 메서드를 호출하고, `view(name, file[, options])`를 한 번 또는 여러 번 호출하는 클로저를 전달합니다. `view`를 호출할 때마다 하나의 출력 파일이 생성됩니다. `name` 인수는 뷰 이름입니다. 사용 가능한 뷰 이름 목록은 `jpexport` 명령줄 실행 파일 [p. 240]의 도움말 페이지를 참조하십시오. `file` 인수는 출력 파일로, 절대 파일이거나 프로젝트에 상대적인 파일일 수 있습니다. 마지막으로, 선택적인 `options` 인수는 선택한 뷰에 대한 내보내기 옵션을 포함하는 맵입니다.

내보내기 태스크를 사용하는 예제는 다음과 같습니다:

```
task export(type: com.jprofiler.gradle.Export) {
    snapshotFile = file('snapshot.jps')
    views {
        view('CallTree', 'callTree.html')
        view('HotSpots', 'hotSpots.html',
            [threadStatus: 'all', expandBacktraces: 'true'])
    }
}
```

스냅샷 비교하기

`bin/jpcompare` 명령줄 도구 [p. 240]와 마찬가지로, `com.jprofiler.gradle.Compare` 태스크는 두 개 이상의 스냅샷을 비교할 수 있습니다. 속성은 다음과 같습니다:

Attribute	Description	Required
snapshotFiles	비교할 스냅샷 파일입니다. Gradle이 파일 컬렉션으로 해석하는 객체를 포함하는 `Iterable`을 전달할 수 있습니다.	예
sortByTime	`true`로 설정되면 제공된 모든 스냅샷 파일이 파일 수정 시간에 따라 정렬되고, 그렇지 않으면 `snapshotFiles` 속성에 지정된 순서대로 비교됩니다.	아니요
ignoreErrors	비교에 대한 옵션을 설정할 수 없을 때 발생하는 오류를 무시하고 다음 비교로 계속 진행합니다. 기본값은 `false`이며, 첫 번째 오류가 발생하면 내보내기가 종료됩니다.	아니요

내보낸 뷰가 `Export` 태스크에 대해 정의된 것과 마찬가지로, `Compare` 태스크는 `comparisons` 메서드를 가지고 있으며, `comparison(name, file[, options])`에 대한 중첩 호출이 수행해야 할 비교를 정의

합니다. 사용 가능한 비교 이름 목록은 `jpcompare` 명령줄 실행 파일 [p. 240]의 도움말 페이지에서 확인할 수 있습니다.

비교 태스크를 사용하는 예제는 다음과 같습니다:

```
task compare(type: com.jprofiler.gradle.Compare) {
    snapshotFiles = files('snapshot1.jps', 'snapshot2.jps')
    comparisons {
        comparison('CallTree', 'callTree.html')
        comparison('HotSpots', 'hotSpots.csv',
            [valueSummation: 'total', format: 'csv'])
    }
}
```

또는 여러 스냅샷에 대한 텔레메트리 비교를 생성하려면:

```
task compare(type: com.jprofiler.gradle.Compare) {
    snapshotFiles = fileTree(dir: 'snapshots', include: '*.jps')
    sortByTime = true
    comparisons {
        comparison('TelemetryHeap', 'heap.html', [valueType: 'maximum'])
        comparison('ProbeTelemetry', 'jdbc.html', [probeId: 'JdbcProbe'])
    }
}
```

힙 스냅샷 분석하기

Gradle 태스크 `com.jprofiler.gradle.Analyze`는 `bin/jpanalyze` 명령줄 도구 [p. 240]와 동일한 기능을 가지고 있습니다.

이 태스크는 `Compare` 태스크와 마찬가지로 처리된 스냅샷을 지정하기 위한 `snapshotFiles` 속성과 디난독화를 위한 `Export` 태스크와 같은 `obfuscator` 및 `mappingfile` 속성을 가지고 있습니다. `removeUnreferenced`, `retainSoft`, `retainWeak`, `retainPhantom`, `retainFinalizer` 및 `retained` 속성은 명령줄 도구의 인수에 해당합니다.

`Analyze` 태스크를 사용하는 예제는 다음과 같습니다:

```
task analyze(type: com.jprofiler.gradle.Analyze) {
    snapshotFiles = fileTree(dir: 'snapshots', include: '*.jps')
    retainWeak = true
    obfuscator = 'proguard'
    mappingFile = file('obfuscation.txt')
}
```

G.4 Ant Tasks

JProfiler에서 제공하는 Ant [(1)] 태스크는 Gradle 태스크와 매우 유사합니다. 이 장에서는 Gradle 태스크와의 차이점을 강조하고 각 Ant 태스크에 대한 예제를 보여줍니다.

모든 Ant 태스크는 `bin/ant.jar` 아카이브에 포함되어 있습니다. Ant에서 태스크를 사용 가능하게 하려면 먼저 Ant에 태스크 정의를 찾을 수 있는 위치를 알려주는 `taskdef` 요소를 삽입해야 합니다. 아래의 모든 예제에는 해당 taskdef가 포함되어 있습니다. 이는 빌드 파일당 한 번만 발생해야 하며 프로젝트 요소 아래의 어느 위치에나 나타날 수 있습니다.

`ant.jar` 아카이브를 Ant 배포의 `lib` 폴더로 복사할 수 없으며, 태스크 정의에서 JProfiler의 전체 설치를 참조해야 합니다.

Profiling from Ant

`com.jprofiler.ant.ProfileTask`는 내장된 `Java` 태스크에서 파생되며 모든 속성과 중첩 요소를 지원합니다. 추가 속성은 `ProfileJava` Gradle 태스크 [p. 248]와 동일합니다. Ant 속성은 대소문자를 구분하지 않으며 일반적으로 소문자로 작성됩니다.

```
<taskdef name="profile"
         classname="com.jprofiler.ant.ProfileTask"
         classpath="<path to JProfiler installation>/bin/ant.jar"/>

<target name="profile">
  <profile classname="MyMainClass" offline="true" sessionid="80">
    <classpath>
      <fileset dir="lib" includes="*.jar" />
    </classpath>
  </profile>
</target>
```

Exporting data from snapshots

`com.jprofiler.ant.ExportTask`를 사용하면 `Export` Gradle 태스크 [p. 248]와 마찬가지로 스냅샷에서 뷰를 내보낼 수 있습니다. 뷰는 Gradle 태스크와 다르게 지정됩니다: 태스크 요소 바로 아래에 중첩되고 옵션은 중첩된 `option` 요소로 지정됩니다.

```
<taskdef name="export"
         classname="com.jprofiler.ant.ExportTask"
         classpath="<path to JProfiler installation>/bin/ant.jar"/>

<target name="export">
  <export snapshotfile="snapshots/test.jps">
    <view name="CallTree" file="calltree.html"/>
    <view name="HotSpots" file="hotspots.html">
      <option name="expandbacktraces" value="true"/>
      <option name="aggregation" value="class"/>
    </view>
  </export>
</target>
```

Comparing snapshots

`com.jprofiler.ant.CompareTask`는 `Compare` Gradle 태스크에 해당하며 두 개 이상의 스냅샷 간의 비교를 수행합니다. `com.jprofiler.ant.ExportTask`와 마찬가지로, 비교는 요소 아래에 직접 중첩되고 옵션은 각 `comparison` 요소에 대해 중첩됩니다. 스냅샷 파일은 중첩된 파일 세트로 지정됩니다.

[(1)] http://ant.apache.org

```
<taskdef name="compare"
         classname="com.jprofiler.ant.CompareTask"
         classpath="<path to JProfiler installation>/bin/ant.jar"/>

<target name="compare">
  <compare sortbytime="true">
    <fileset dir="snapshots">
      <include name="*.jps" />
    </fileset>
    <comparison name="TelemetryHeap" file="heap.html"/>
    <comparison name="TelemetryThreads" file="threads.html">
      <option name="measurements" value="inactive,active"/>
      <option name="valuetype" value="bookmark"/>
      <option name="bookmarkname" value="test"/>
    </comparison>
  </compare>
</target>
```

Analyzing heap snapshots

`Analyze` Gradle 태스크와 마찬가지로, Ant에 대한 `com.jprofiler.ant.AnalyzeTask`는 오프라인 프로파일링으로 저장된 스냅샷에서 GUI에서 더 빠르게 액세스할 수 있도록 힙 스냅샷 분석을 준비합니다. 처리할 스냅샷은 중첩된 파일 세트로 지정됩니다.

```
<taskdef name="analyze"
         classname="com.jprofiler.ant.AnalyzeTask"
         classpath="<path to JProfiler installation>/bin/ant.jar"/>

<target name="analyze">
  <analyze>
    <fileset dir="snapshots" includes="*.jps" />
  </analyze>
</target>
```