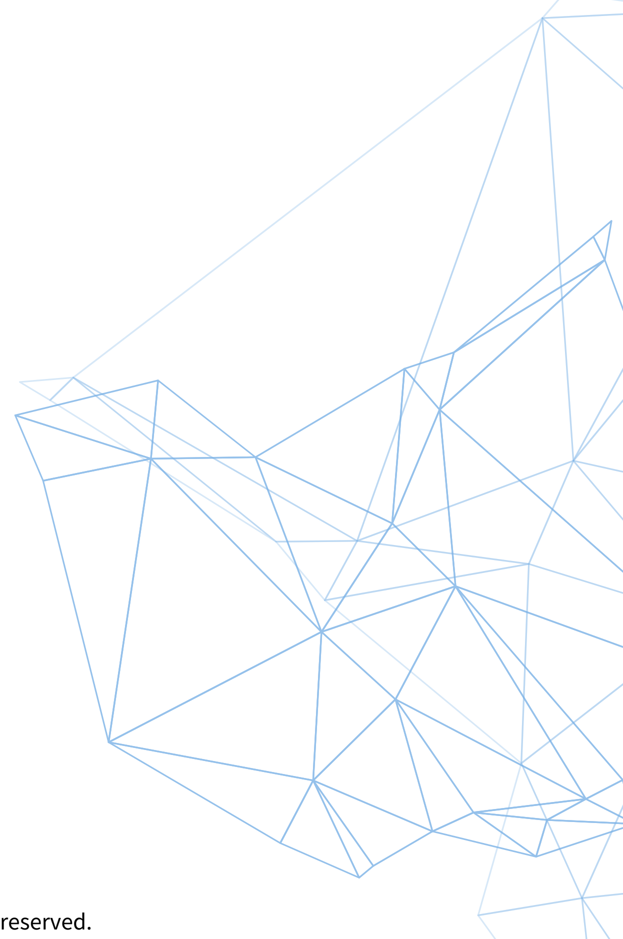
ej technologies

# JProfilerの決定版ガイド

パフォーマンスのプロフェッショナルとして知っておくべきすべてのこと



© 2025 ej-technologies GmbH. All rights reserved.

# Index


イントロダクション .......... 4

アーキテクチャ .......... 6

インストール .......... 8

JVMのプロファイリング .......... 12

データの記録 .......... 28

スナップショット .......... 41

テレメトリー .......... 46

CPUプロファイリング .......... 54

メソッドコールの記録 .......... 67

メモリープロファイリング .......... 72

ヒープウォーカー .......... 81

スレッドプロファイリング .......... 98

プローブ .......... 105

GC分析 .......... 119

MBeanブラウザ .......... 125

オフラインプロファイリング .......... 130

スナップショットの比較 .......... 135

IDE統合 .......... 141

A カスタムプローブ .......... 152
- A.1 プローブの概念 .......... 152
- A.2 スクリプトプローブ .......... 159
- A.3 インジェクションプローブ .......... 163
- A.4 埋め込みプローブ .......... 168

B 呼び出しツリーの詳細機能 .......... 172
- B.1 インストゥルメンテーションの自動調整 .......... 172
- B.2 非同期およびリモートリクエストトラッキング .......... 175
- B.3 呼び出しツリーの一部を表示 .......... 181
- B.4 呼び出しツリーの分割 .......... 186
- B.5 呼び出しツリーの分析 .......... 190

C 高度なCPU分析ビュー .......... 196
- C.1 異常値検出 .......... 196

C.2 複雑性分析 ... 200
C.3 コールトレーサー ... 202
C.4 JavaScript XHR ... 204

D ヒープウォーカーの詳細機能 ... 207
D.1 HPROFスナップショット ... 207
D.2 オーバーヘッドの最小化 ... 210
D.3 フィルターとライブインタラクション ... 212
D.4 メモリーリークの発見 ... 216

E JDKフライトレコーダー (JFR) ... 223
E.1 JFR概要 ... 223
E.2 JFRスナップショットの記録 ... 225
E.3 JFRイベントブラウザ ... 229
E.4 JFRビュー ... 236

F 詳細な設定 ... 243
F.1 接続問題のトラブルシューティング ... 243
F.2 スクリプト ... 245
F.3 カスタムヘルプ ... 249
F.4 起動時のプロファイリング設定 ... 250

G コマンドラインリファレンス ... 253
G.1 プロファイリング用実行ファイル ... 253
G.2 スナップショット用実行ファイル ... 256
G.3 Gradleタスク ... 265
G.4 Antタスク ... 269

# JProfilerの紹介

### JProfilerとは？

JProfilerは、実行中のJVM内部で何が起こっているかを分析するためのプロフェッショナルなツールです。開発、品質保証、そして本番システムで問題が発生した際の緊急対応に使用できます。

JProfilerが扱う主なトピックは4つあります：

- **メソッドコール**
  これは一般的に「CPUプロファイリング」と呼ばれます。メソッドコールはさまざまな方法で測定および視覚化できます。メソッドコールの分析は、アプリケーションが何をしているのかを理解し、そのパフォーマンスを改善する方法を見つけるのに役立ちます。
- **割り当て**
  ヒープ上のオブジェクトをその割り当て、参照チェーン、ガベージコレクションに関して分析することは、「メモリプロファイリング」のカテゴリに入ります。この機能により、メモリリークを修正し、一般的にメモリを少なく使用し、一時的なオブジェクトの割り当てを減らすことができます。
- **スレッドとロック**
  スレッドは、例えばオブジェクトに同期することでロックを保持することができます。複数のスレッドが協力すると、デッドロックが発生する可能性があり、JProfilerはそれを視覚化できます。また、ロックは競合することがあり、スレッドがそれを取得する前に待たなければならないことがあります。JProfilerはスレッドとそのさまざまなロック状況についての洞察を提供します。
- **高レベルのサブシステム**
  多くのパフォーマンス問題は、より高いセマンティックレベルで発生します。例えば、JDBCコールでは、どのSQLステートメントが最も遅いかを知りたいかもしれません。そのようなサブシステムのために、JProfilerは特定のペイロードを呼び出しツリーにアタッチする「プローブ」を提供します。

JProfilerのUIはデスクトップアプリケーションとして提供されます。ライブJVMをインタラクティブにプロファイルすることも、UIを使用せずに自動的にプロファイルすることもできます。プロファイリングデータはスナップショットに保存され、JProfiler UIで開くことができます。さらに、コマンドラインツールやビルドツールの統合により、プロファイリングセッションの自動化を支援します。

### 次にどうすればいいですか？

このドキュメントは順番に読むことを意図しており、後のヘルプトピックは前の内容に基づいています。

まず、アーキテクチャ [p. 6]に関する技術的な概要が、プロファイリングの仕組みを理解するのに役立ちます。

JProfilerのインストール [p. 8]と JVMのプロファイリング [p. 12]に関するヘルプトピックが、あなたを迅速に始動させます。

その後、データ記録 [p. 28]と スナップショット [p. 41]の議論が、JProfilerを自分で探求できるレベルの理解に導きます。

後続の章は、JProfilerのさまざまな機能に関する専門知識を構築します。最後のセクションはオプションの読み物であり、特定の機能が必要な場合に参照するべきです。

フィードバックをお待ちしております。特定の領域でドキュメントが不足していると感じたり、ドキュメントに不正確さを見つけた場合は、support@ej-technologies.comまでお気軽にご連絡ください。

# JProfiler アーキテクチャ

プロファイルされたアプリケーション、JProfiler　UI、およびすべてのコマンドラインユーティリティに関与する重要な相互作用の全体像が以下に示されています。

jpexport
jpcompare
jpanalyze

JProfiler UI

jpcontroller

Snapshots

リモートまたはローカル

ローカル

データを
送信します

ソケット経由で
接続します

JMX経由で
接続します

jpenable

アタッチ経由で
ロードされます

JProfiler
エージェント

公開する

JProfiler
MBean

jpdump

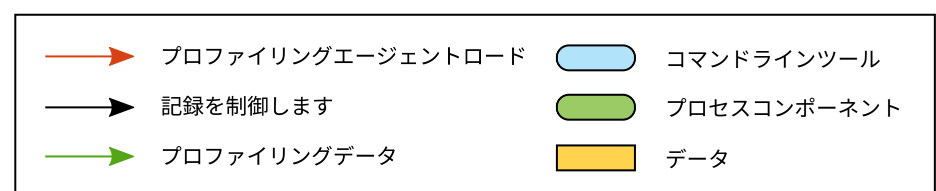
HPROFヒープダンプ
取得

プロファイルJVM

-agentpathで
ロード

オフラインプロファイリングで
制御します

コマンドライン引数




## プロファイリングエージェント

"JVM tool interface" (JVMTI) は、プロファイラが情報にアクセスし、自身の計測を挿入するためのフックを追加するために使用するネイティブインターフェースです。これは、プロファイリング

エージェントの少なくとも一部がネイティブコードとして実装されなければならないことを意味し、したがって JVM プロファイラはプラットフォームに依存しません。JProfiler は、ここ [p. 8] にリストされているさまざまなプラットフォームをサポートしています。

JVM プロファイラは、起動時または後のある時点でロードされるネイティブライブラリとして実装されます。起動時にロードするには、VM パラメータ -agentpath:< > をコマンドラインに追加します。このパラメータを手動で追加する必要はほとんどありません。なぜなら、JProfiler が IDE 統合、統合ウィザード、または JVM を直接起動する場合などに自動的に追加するからです。しかし、これがプロファイリングを可能にするものであることを知っておくことが重要です。

JVM がネイティブライブラリのロードに成功すると、プロファイリングエージェントが初期化する機会を与えるためにライブラリ内の特別な関数を呼び出します。その後、JProfiler は JProfiler> で始まるいくつかの診断メッセージを出力し、プロファイリングがアクティブであることを確認できます。結論として、-agentpath VM パラメータを渡すと、プロファイリングエージェントは正常にロードされるか、JVM が起動しません。

一度ロードされると、プロファイリングエージェントはスレッドの作成やクラスのロードなど、さまざまなイベントの通知を JVMTI に要求します。これらのイベントの一部は直接プロファイリングデータを提供します。クラスロードイベントを使用して、プロファイリングエージェントはロードされたクラスを計測し、独自のバイトコードを挿入して測定を行います。

JProfiler は、JProfiler UI を使用するか、bin/jpenable コマンドラインツールを使用して、すでに実行中の JVM にエージェントをロードできます。その場合、必要な計測を適用するために、多数のすでにロードされたクラスを再変換する必要があるかもしれません。

### データの記録

JProfiler エージェントはプロファイリングデータのみを収集します。JProfiler UI は別途起動され、ソケットを介してプロファイリングエージェントに接続します。リモートサーバーへの安全な接続のために、JProfiler を設定して SSH トンネルを自動的に作成することができます。

JProfiler UI から、エージェントにデータを記録するよう指示し、UI にプロファイリングデータを表示し、スナップショットをディスクに保存することができます。UI の代替として、プロファイリングエージェントはその MBean [(1)] を通じて制御することができます。この MBean を使用するコマンドラインツールは bin/jpcontroller です。

プロファイリングエージェントを制御するもう一つの方法は、事前定義されたトリガーとアクションのセットを使用することです。この方法では、プロファイリングエージェントは無人モードで動作することができます。これは JProfiler では "オフラインプロファイリング" と呼ばれ、プロファイリングセッションを自動化するのに役立ちます。

### スナップショット

JProfiler UI はライブプロファイリングデータを表示できますが、記録されたすべてのプロファイリングデータのスナップショットを保存する必要があることがよくあります。スナップショットは JProfiler UI で手動で保存されるか、トリガーアクションによって自動的に保存されます。

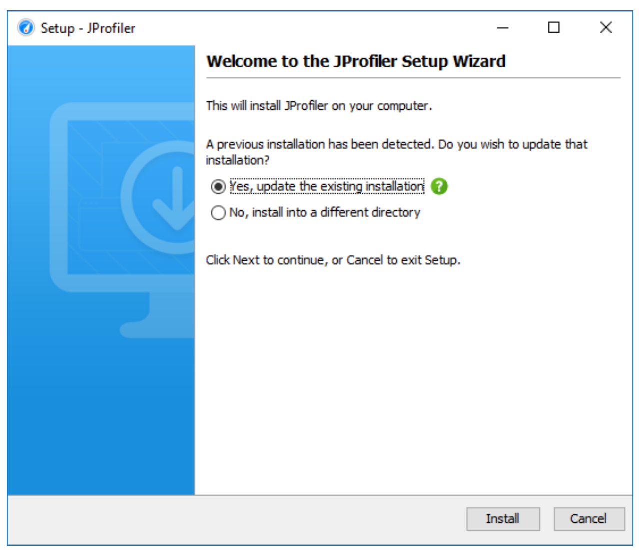
スナップショットは JProfiler UI で開いて比較することができます。自動処理のために、コマンドラインツール bin/jpexport および bin/jpcompare を使用して、以前に保存されたスナップショットからデータを抽出し、HTML レポートを作成することができます。

実行中の JVM からヒープスナップショットを取得する低オーバーヘッドの方法は、bin/jpdump コマンドラインツールを使用することです。これは、JVM の組み込み機能を使用して HPROF スナップショットを保存し、JProfiler で開くことができ、プロファイリングエージェントをロードする必要はありません。

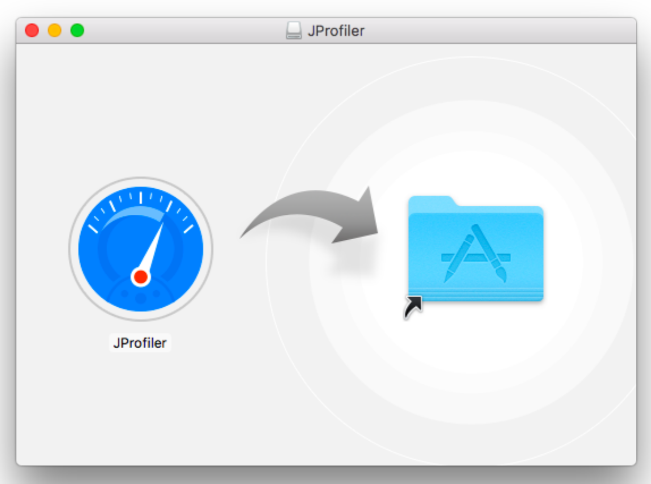
(1) https://en.wikipedia.org/wiki/Java_Management_Extensions

# JProfilerのインストール

WindowsおよびLinux/Unix用の実行可能インストーラーが提供されており、インストールをステップバイステップで案内します。以前のインストールが検出された場合、インストールは簡略化されます。




macOSでは、JProfilerはUIアプリケーションの標準インストール手順を使用します。DMGアーカイブをダブルクリックでFinderにマウントし、JProfilerアプリケーションバンドルを`/Applications`フォルダにドラッグできます。そのフォルダはDMG自体にシンボリックリンクとして表示されます。




Linux/Unixでは、インストーラーはダウンロード後に実行可能ではないため、実行時に`sh`を前に付ける必要があります。インストーラーは、パラメーター`-c`を渡すとコマンドラインインストールを実行します。 WindowsおよびLinux/Unixの完全に無人のインストールは、パラメーター`-q`で実行されます。その場合、インストールディレクトリを選択するために追加の引数`-dir <directory>`を渡すことができます。

```
ingo@ubuntu: ~/Downloads
ingo@ubuntu:~/Downloads$ sh jprofiler_linux_10_0_2.sh -c
Starting Installer ...
This will install JProfiler on your computer.
OK [o, Enter], Cancel [c]

A previous installation has been detected. Do you wish to update that installati
on?
Yes, update the existing installation [1, Enter]
No, install into a different directory [2]
```

インストーラーを実行すると、.install4j/response.varfileというファイルにすべてのユーザー入力が保存されます。そのファイルを使用して、コマンドラインで引数-varfile <path to response.varfile>を渡すことで無人インストールを自動化できます。

無人インストールのライセンス情報を設定するには、 -Vjprofiler.licenseKey=<license key> -Vjprofiler.licenseName=<user name>をコマンドライン引数として渡し、必要に応じて-Vjprofiler.licenseCompany=<company name>を渡します。 フローティングライセンスをお持ちの場合は、ライセンスキーの代わりにFLOAT:<server name or IP address>を使用してください。

アーカイブはWindows用のZIPファイルおよびLinux用の.tar.gzファイルとしても提供されます。コマンド

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

は.tar.gzアーカイブを別のトップレベルディレクトリに展開します。 JProfilerを開始するには、展開したディレクトリでbin/jprofilerを実行します。Linux/Unixでは、 jprofiler.desktopファイルを使用してJProfilerの実行可能ファイルをウィンドウマネージャーに統合できます。 例えば、Ubuntuではデスクトップファイルをランチャーサイドバーにドラッグして、永続的なランチャーアイテムを作成できます。

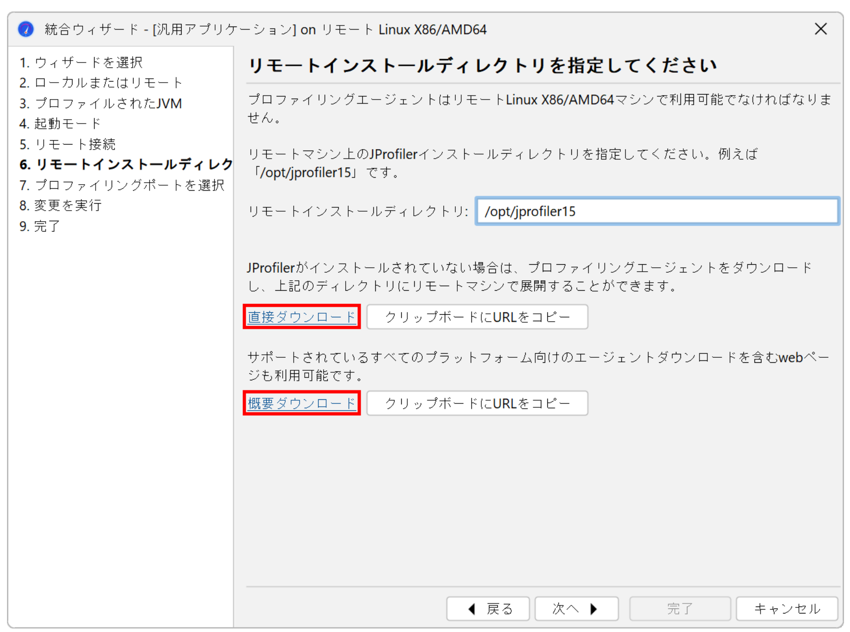
**プロファイリングエージェントをリモートマシンに配布する**

JProfilerには2つの部分があります。一方はスナップショットを操作するコマンドラインユーティリティとデスクトップUIで、もう一方はプロファイリングエージェントとプロファイルされたJVMを制御するコマンドラインユーティリティです。 ウェブサイトからダウンロードするインストーラーとアーカイブには両方の部分が含まれています。

しかし、リモートプロファイリングの場合、リモート側にインストールする必要があるのはプロファイリングエージェントだけです。リモートマシンにJProfilerのディストリビューションをアーカイブから抽出するだけでも良いですが、特にデプロイメントを自動化する際には、必要なファイルの数を制限したいかもしれません。また、プロファイリングエージェントは自由に再配布可能なので、アプリケーションと一緒に出荷したり、トラブルシューティングのために顧客のマシンにインストールすることができます。

プロファイリングエージェントの最小パッケージを取得するには、リモート統合ウィザードで適切なエージェントアーカイブのダウンロードリンクと、サポートされているすべてのプラットフォーム用のエージェントアーカイブのダウンロードページを表示します。 JProfiler GUIでセッション->

統合ウィザード->新しいサーバー/リモート統合を呼び出し、 「リモート」オプションを選択し、リモートインストールディレクトリステップに進みます。




特定のJProfilerバージョンのHTML概要ページのURLは

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

単一のエージェントアーカイブのダウンロードURLの形式は

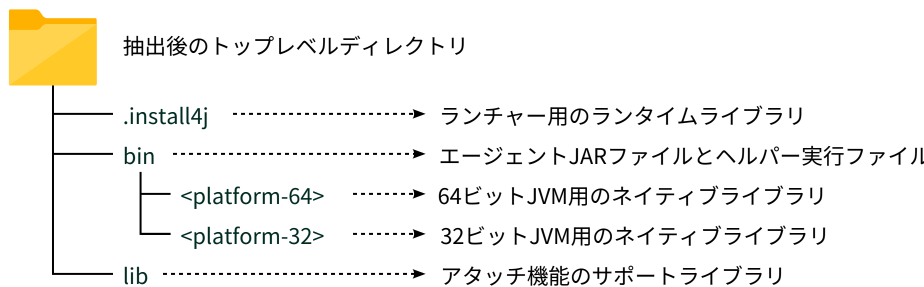
```
https://download.ej-technologies.com/jprofiler/jprofiler_agent_<platform>_15_0.<extension>
```

ここで`platform`はbinディレクトリ内のエージェントディレクトリ名に対応し、 `extension`はWindowsでは`zip`、macOSでは`.tgz`、 Linux/Unixでは`.tar.gz`です。Linuxではx86とx64が一緒にグループ化されているため、 Linux x64のURLは

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

エージェントアーカイブには、必要なネイティブエージェントライブラリと`jpenable`、 `jpdump`、 `jpcontroller`の実行可能ファイルが含まれています。 アーカイブ内の実行可能ファイルとプロファイリングエージェントは、最低限Java 8が必要です。

リモートマシンでエージェントアーカイブを抽出した後に表示されるサブディレクトリは以下に説明されています。それらは、対応するターゲットプラットフォームでの完全なJProfilerインストールのサブセットです。




**サポートされているプラットフォーム**

JProfilerはJVM（JVMTI）のネイティブプロファイリングインターフェースを利用しているため、そのプロファイリングエージェントはネイティブライブラリです。

JProfilerは以下のプラットフォームでのプロファイリングをサポートしています:

| OS | アーキテクチャ | サポートされているJVM | バージョン |
|---|---|---|---|
| Windows 11/10<br>Windows Server 2025/2022/2019/2016 | x86<br>x64/AMD64 | Hotspot (OpenJDK)<br>IBM/OpenJ9 | 1.8 - 24<br>1.8 - 24 |
| macOS 10.12 - 15 | Intel, Apple | Hotspot (OpenJDK)<br>IBM/OpenJ9 | 1.8 - 24<br>1.8 - 24 |
| Linux | x86<br>x64/AMD64 | Hotspot (OpenJDK)<br>IBM/OpenJ9 | 1.8 - 24<br>1.8 - 24 |
| Linux | PPC64LE | Hotspot (OpenJDK)<br>IBM/OpenJ9 | 1.8 - 24<br>1.8 - 24 |
| Linux | ARMv7<br>ARMv8 | Hotspot (OpenJDK) | 1.8 - 24 |

JProfiler GUIフロントエンドはJava 21 VMを必要とします。Windows、macOS、Linux x64用にJava 21 JREがJProfilerにバンドルされています。アタッチコマンドラインツールjpenable、jdump、jpcontrollerはJava 8 VMのみを必要とします。

# JVMのプロファイリング

JVMをプロファイルするには、JProfilerのプロファイリングエージェントをJVMにロードする必要があります。これは2つの異なる方法で行うことができます：開始スクリプトで`-agentpath`VMパラメータを指定するか、既に実行中のJVMにエージェントをロードするためにattach APIを使用します。

JProfilerは両方のモードをサポートしています。VMパラメータを追加することはプロファイルするための推奨方法であり、統合ウィザード、IDEプラグイン、およびJProfiler内からJVMを起動するセッション設定で使用されます。アタッチはローカルでもSSHを介してリモートでも動作します。

### -agentpath VMパラメータ

プロファイリングエージェントをロードするVMパラメータがどのように構成されているかを理解することは有用です。`-agentpath`は、JVMTIインターフェースを使用する任意のネイティブライブラリをロードするためにJVMによって提供される一般的なVMパラメータです。プロファイリングインターフェースJVMTIはネイティブインターフェースであるため、プロファイリングエージェントはネイティブライブラリでなければなりません。これは、明示的にサポートされているプラットフォーム[(1)]でのみプロファイルできることを意味します。32ビットおよび64ビットのJVMも異なるネイティブライブラリを必要とします。一方、Javaエージェントは`-javaagent` VMパラメータでロードされ、限られた機能セットにのみアクセスできます。

`-agentpath:`の後に、ネイティブライブラリへのフルパス名が追加されます。プラットフォーム固有のライブラリ名のみを指定する`-agentlib:`という同等のパラメータがありますが、その場合はライブラリがライブラリパスに含まれていることを確認する必要があります。ライブラリへのパスの後に等号を追加し、カンマで区切ってエージェントにオプションを渡すことができます。例えば、Linuxでは、全体のパラメータは次のようになります：

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

最初の等号はパス名をパラメータから分離し、2番目の等号はパラメータ`port=8849`の一部です。この一般的なパラメータは、プロファイリングエージェントがJProfiler GUIからの接続を待ち受けるポートを定義します。8849は実際にはデフォルトのポートであるため、そのパラメータを省略することもできます。同じマシンで複数のJVMをプロファイルしたい場合は、異なるポートを割り当てる必要があります。IDEプラグインとローカルで起動されたセッションはこのポートを自動的に割り当てますが、統合ウィザードではポートを明示的に選択する必要があります。

2番目のパラメータ`nowait`は、JProfiler GUIが接続するまでJVMを起動時にブロックしないようにプロファイリングエージェントに指示します。起動時にブロックするのがデフォルトです。なぜなら、プロファイリングエージェントはコマンドラインパラメータとしてプロファイリング設定を受け取るのではなく、JProfiler GUIからまたは代替として設定ファイルから受け取るからです。コマンドラインパラメータはプロファイリングエージェントをブートストラップし、開始方法を指示し、デバッグフラグを渡すためだけのものです。

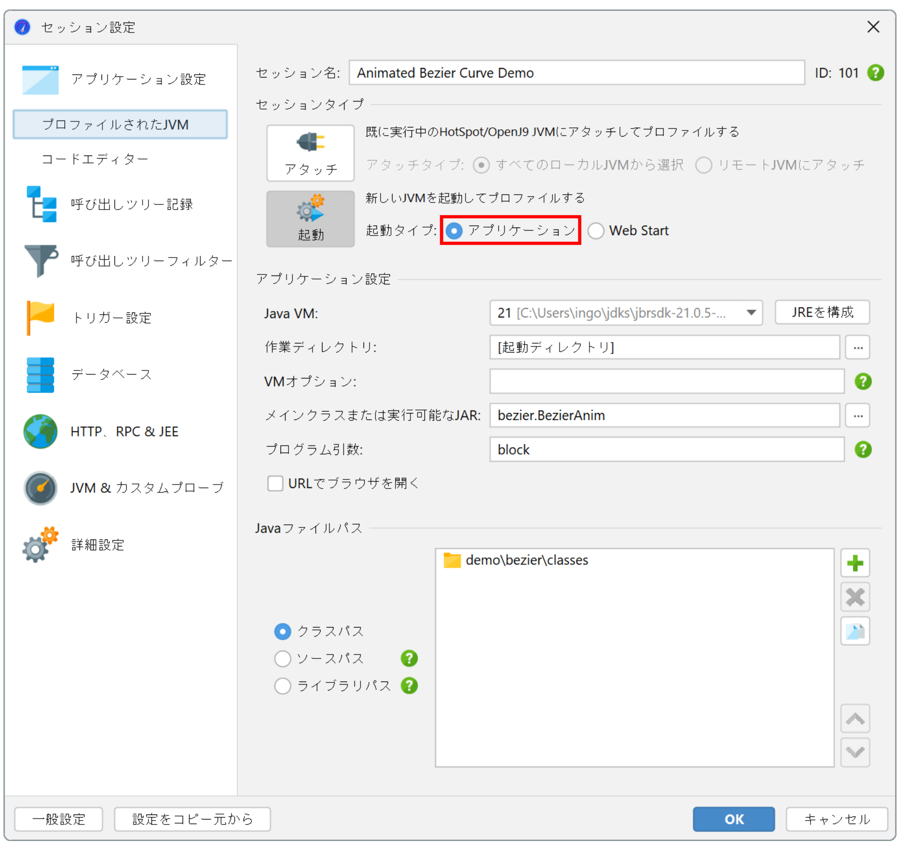
一部の状況では、起動時にプロファイリング設定を設定する[p. 250]必要があり、これを達成するために手動で作業が必要な場合があります。

デフォルトでは、JProfilerエージェントは通信ソケットをループバックインターフェースにバインドします。特定のインターフェースを選択するために`address=[IP address]`オプションを追加するか、`address=0.0.0.0`を追加して通信ソケットを利用可能なすべてのネットワークインターフェースにバインドすることができます。これは、dockerコンテナからプロファイリングポートを公開したい場合に必要になることがあります。

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

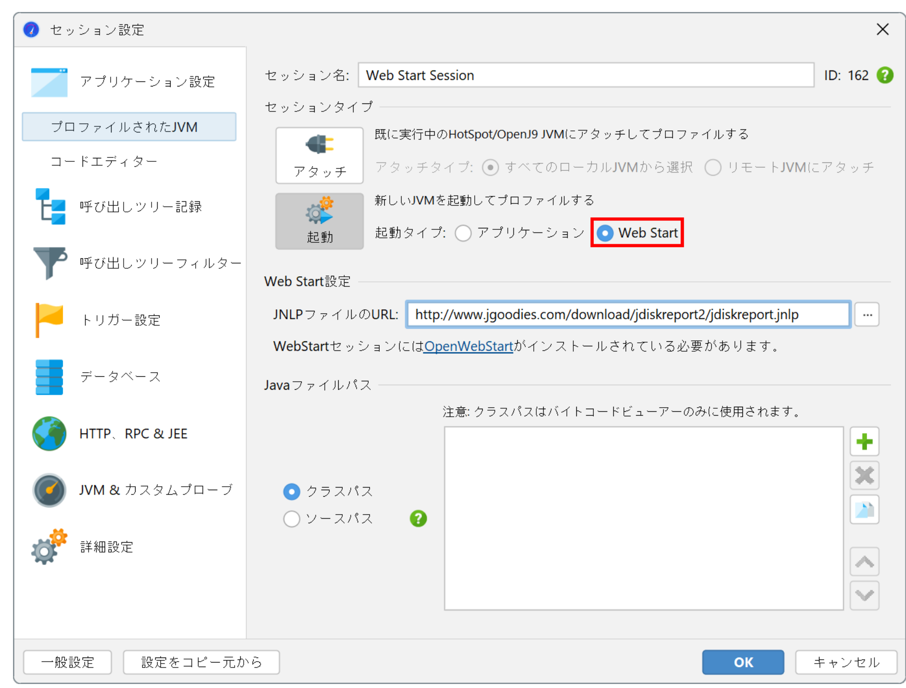
### ローカルで起動されたセッション

IDEの「実行構成」と同様に、JProfilerでローカルで起動されたセッションを直接構成できます。クラスパス、メインクラス、作業ディレクトリ、VMパラメータと引数を指定し、JProfilerがセッションを起動します。JProfilerに付属するすべてのデモセッションはローカルで起動されたセッションです。

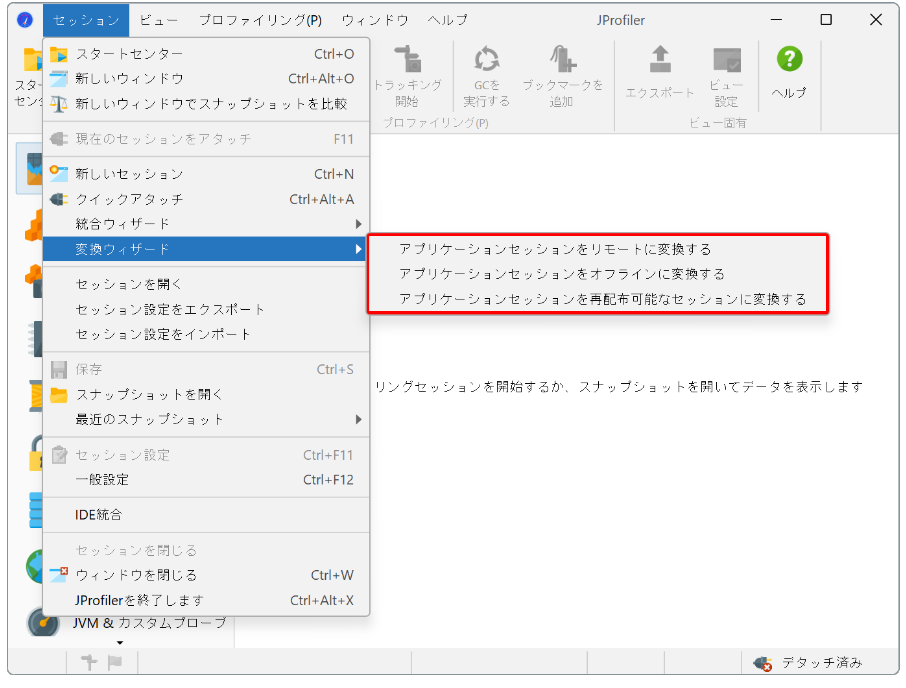



特別な起動モードは「Web Start」で、JNLPファイルのURLを選択し、JProfilerがそれをプロファイルするためにJVMを起動します。この機能はOpenWebStart [(2)] をサポートしており、Java 9以前のOracle JREからのレガシーWebStartはサポートされていません。

(2) https://openwebstart.com/



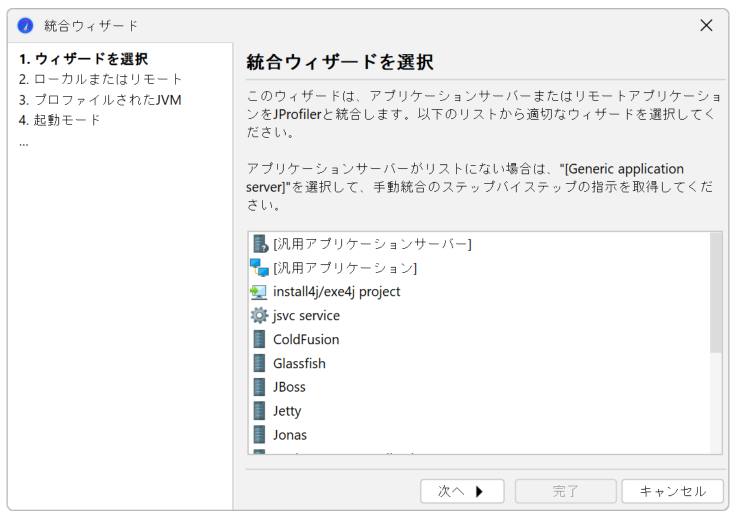

ローカルで起動されたセッションは、メインメニューからセッション->変換ウィザードを呼び出すことで変換ウィザードによってスタンドアロンセッションに変換できます。アプリケーションセッションをリモートに変換は単に開始スクリプトを作成し、`-agentpath` VMパラメータをJava呼び出しに挿入します。アプリケーションセッションをオフラインに変換は、オフラインプロファイリング[p. 130]のための開始スクリプトを作成します。これは、設定が起動時にロードされ、JProfiler GUIが必要ないことを意味します。アプリケーションセッションを再配布セッションに変換は同じことを行いますが、プロファイリングエージェントと設定ファイルを含むディレクトリ`jprofiler_redist`をその隣に作成し、JProfilerがインストールされていない別のマシンにそれを送ることができます。



プロファイルされたアプリケーションを自分で開発する場合は、起動されたセッションの代わりにIDE統合[p. 141]を使用することを検討してください。それはより便利で、より良いソースコードナビゲーションを提供します。アプリケーションを自分で開発していないが、既に開始スクリプトを持っている場合は、リモート統合ウィザードを使用することを検討してください。それはJava呼び出しに追加する必要がある正確なVMパラメータを教えてくれます。

### 統合ウィザード

JProfilerの統合ウィザードは、スタートスクリプトや設定ファイルをプログラム的に変更して追加のVMパラメータを含めることができる多くの有名なサードパーティコンテナを扱います。いくつかの製品では、VMパラメータが引数として渡されるか、環境変数を介して渡されるスタートスクリプトを生成できます。




すべてのケースで、サードパーティ製品から特定のファイルを見つける必要があります。これにより、JProfilerは必要なコンテキストを持って変更を行うことができます。いくつかの一般的なウィザードは、プロファイリングを有効にするために何をする必要があるかについての指示を与えるだけです。

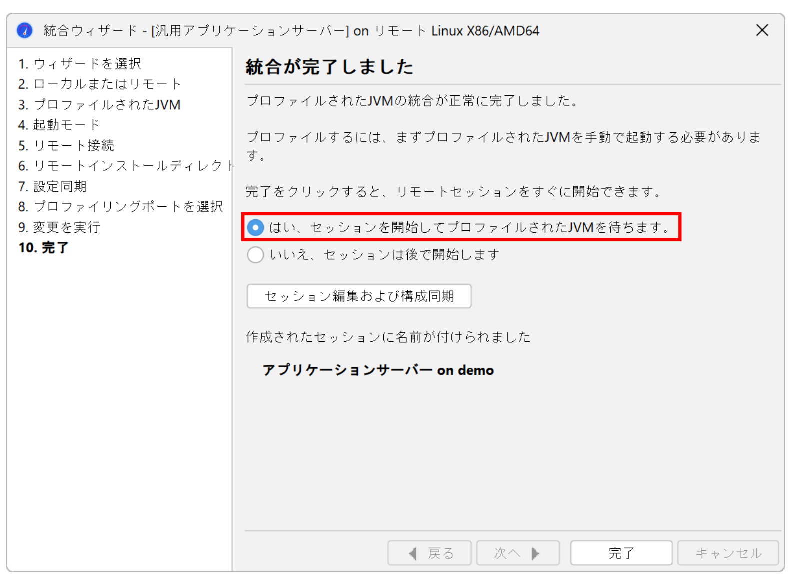
各統合ウィザードの最初のステップは、ローカルマシンでプロファイルするか、リモートマシンでプロファイルするかの選択です。ローカルマシンの場合、JProfilerは既にプラットフォームを知っ

ており、JProfilerがインストールされている場所と設定ファイルがどこにあるかを知っているため、提供する情報は少なくて済みます。

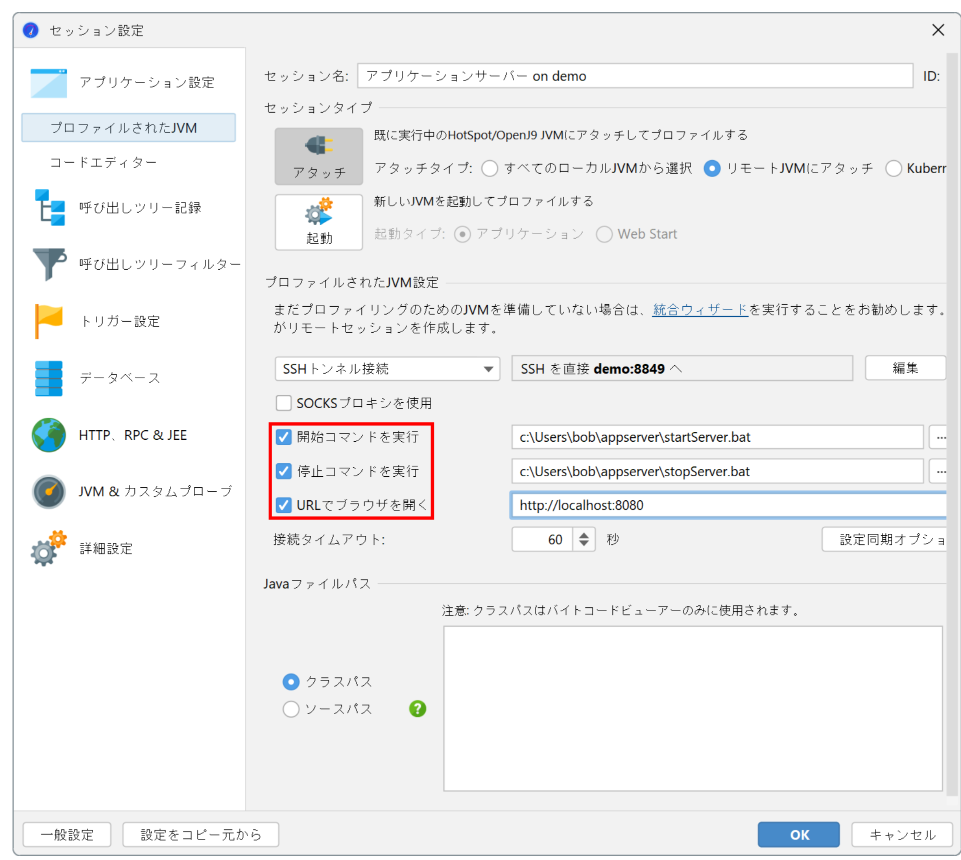
重要な決定は、上記で議論された「起動モード」です。デフォルトでは、プロファイリング設定は起動時にJProfiler　UIから送信されますが、JVMをすぐに起動させるようにプロファイリングエージェントに指示することもできます。後者の場合、プロファイリング設定はJProfiler GUIが接続したときに適用できます。

しかし、プロファイリング設定を含む設定ファイルを指定することもできます。これははるかに効率的です。これは設定の同期ステップで行われます。この場合の主な問題は、プロファイリング設定をローカルで編集するたびに設定ファイルをリモート側と同期する必要があることです。最もエレガントな方法は、リモートアドレスステップでSSHを介してリモートマシンに接続し、設定ファイルをSSHを介して自動的に転送できるようにすることです。

統合ウィザードの最後に、プロファイリングを開始するセッションが作成され、非一般的なケースでは、アプリケーションサーバーなどのサードパーティ製品も起動されます。




外部スタートスクリプトは、セッション設定ダイアログのアプリケーション設定タブでスタートスクリプトを実行およびストップスクリプトを実行オプションによって処理され、URLでブラウザを開くチェックボックスを選択することでURLを表示できます。ここは、リモートマシンのアドレスと設定同期オプションを変更できる場所でもあります。




統合ウィザードはすべて、プロファイルされたJVMがリモートマシンで実行されているケースを処理します。しかし、設定ファイルやスタートスクリプトを変更する必要がある場合は、それをローカルマシンにコピーし、変更されたバージョンをリモートマシンに戻す必要があります。コマンドラインツール`jpintegrate`をリモートマシンで直接実行し、その場で変更を行う方が便利かもしれません。`jpintegrate`はJProfilerの完全なインストールを必要とし、JProfiler GUIと同じJRE要件があります。

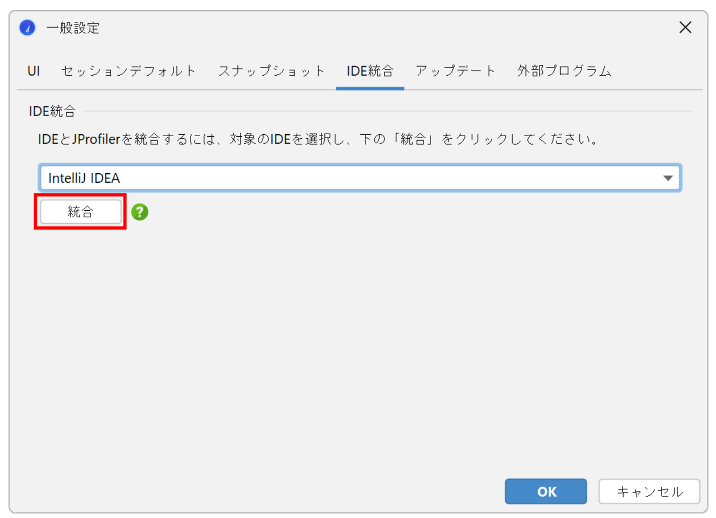
```
ingo@ubuntu: ~
ingo@ubuntu:~$ jprofiler10/bin/jpintegrate
Welcome to the JProfiler console integration wizard!

How do you want to find your integration wizard?
Search by keyword [1, Enter], List all wizards [2]
1
Please enter a number of keywords separated by spaces (for example: Tomcat 5)
Websphere
Please choose one of the following integration wizards:
IBM Websphere 9.x Application Server [1]
IBM Websphere 8.x Application Server [2]
IBM Websphere 7.0 Application Server [3]
IBM Websphere 6.1 Application Server [4]
IBM WebSphere Community Edition 2.x [5]
```

リモートプロファイリングセッションを開始する際にエラーが発生した場合は、トラブルシューティングガイド[p.243]を参照して、問題を解決するために取ることができるステップのチェックリストを確認してください。

### IDE統合

アプリケーションをプロファイルする最も便利な方法は、IDE統合を通じて行うことです。開発中に通常IDEからアプリケーションを起動する場合、IDEはすでに必要な情報をすべて持っており、JProfilerプラグインは単にプロファイリング用のVMパラメータを追加し、必要に応じてJProfilerを起動し、プロファイルされたJVMをJProfilerメインウィンドウに接続することができます。

すべてのIDE統合は、JProfilerインストールの`integrations`ディレクトリに含まれています。原則として、そのディレクトリ内のアーカイブは、各IDEのプラグインインストールメカニズムを使用して手動でインストールできます。しかし、IDE統合をインストールするための推奨方法は、メインメニューからセッション->IDE統合を呼び出すことです。




IDEからのプロファイリングセッションは、JProfilerで独自のセッションエントリを取得しません。なぜなら、そのようなセッションはJProfiler GUIから開始できないからです。プロファイリング設定は、IDEの設定に応じて、プロジェクトごとまたは実行構成ごとに永続化されます。

IDEに接続されているとき、JProfilerはツールバーにウィンドウスイッチャーを表示し、関連するウィンドウに簡単に戻ることができます。すべてのソースを表示アクションは、JProfilerの組み込みソースビューアではなく、IDEで直接ソースを表示します。

IDE統合は、後の章[p. 141]で詳しく説明されています。

### アタッチモード

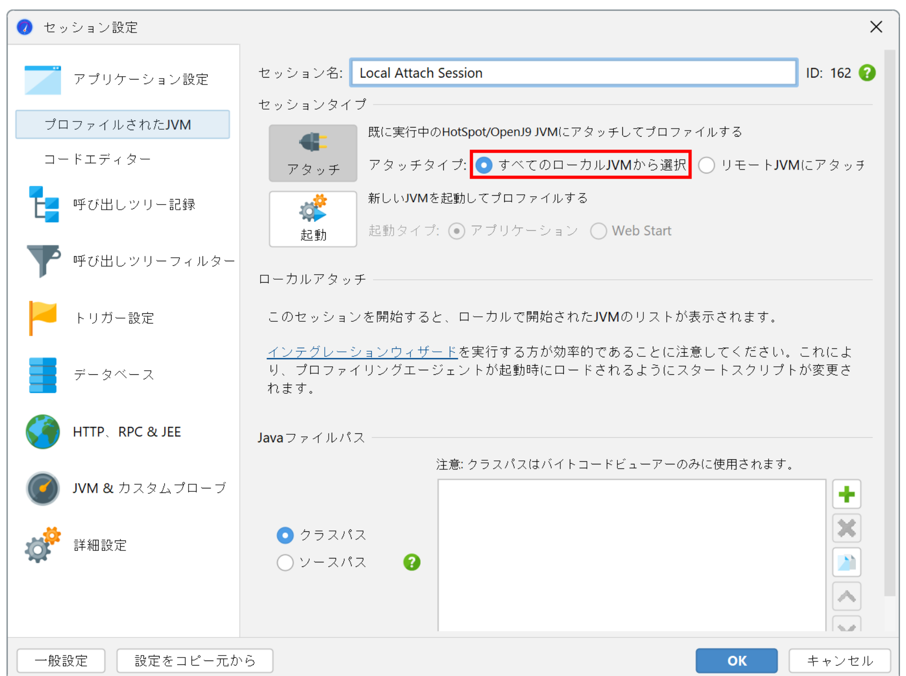
JVMをプロファイルするつもりであることを事前に決定する必要はありません。JProfilerのアタッチ機能を使用すると、実行中のJVMを選択してプロファイリングエージェントを動的にロードできます。アタッチモードは便利ですが、注意すべきいくつかの欠点があります：

- プロファイルしたいJVMを実行中のJVMのリストから識別する必要があります。同じマシンで多くのJVMが実行されている場合、これは時々難しいことがあります。
- 多くのクラスを再定義してインストルメンテーションを追加する必要があるため、追加のオーバーヘッドがあります。
- JProfilerの一部の機能はアタッチモードでは利用できません。これは主に、JVMTIの一部の機能はJVMが初期化されるときにのみオンにでき、JVMのライフサイクルの後の段階では利用できないためです。

- 一部の機能は、すべてのクラスの大部分にインストルメンテーションを必要とします。クラスがロードされている間にインストルメンテーションを追加するのは安価ですが、クラスがすでにロードされているときに後からインストルメンテーションを追加するのはそうではありません。そのような機能は、アタッチモードを使用するときにデフォルトで無効になります。
- アタッチ機能は、OpenJDK JVM、バージョン6以上のOracle JVM、最近のOpenJ9 JVM（8u281+、11.0.11+またはJava 17+）、またはそのようなリリースに基づくIBM JVMでサポートされています。JVMには`-XX:+PerfDisableSharedMem`および`-XX:+DisableAttachMechanism`のVMパラメータを指定してはなりません。

JProfilerのスタートセンターのクイックアタッチタブには、プロファイル可能なすべてのJVMがリストされています。リストエントリの背景色は、プロファイリングエージェントがすでにロードされているか、JProfiler GUIが現在接続されているか、オフラインプロファイリングが設定されているかを示します。

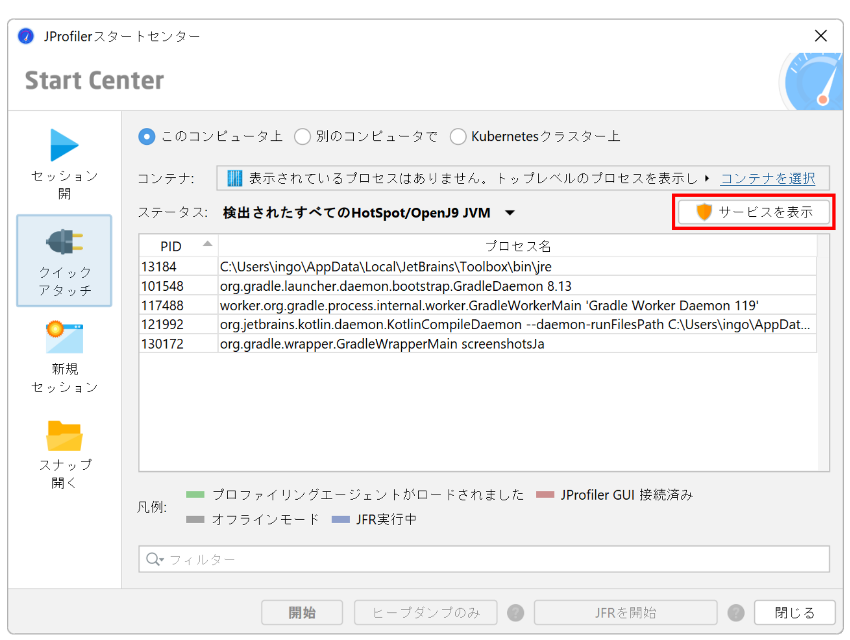
プロファイリングセッションを開始するとき、セッション設定ダイアログでプロファイリング設定を構成できます。同じプロセスを繰り返しプロファイルする場合、同じ設定を何度も再入力したくないので、クイックアタッチ機能で作成されたセッションを閉じるときに永続的なセッションを保存できます。次回このプロセスをプロファイルしたいときは、クイックアタッチタブではなくセッションを開くタブから保存されたセッションを開始します。実行中のJVMを選択する必要がありますが、プロファイリング設定は以前に構成したものと同じです。



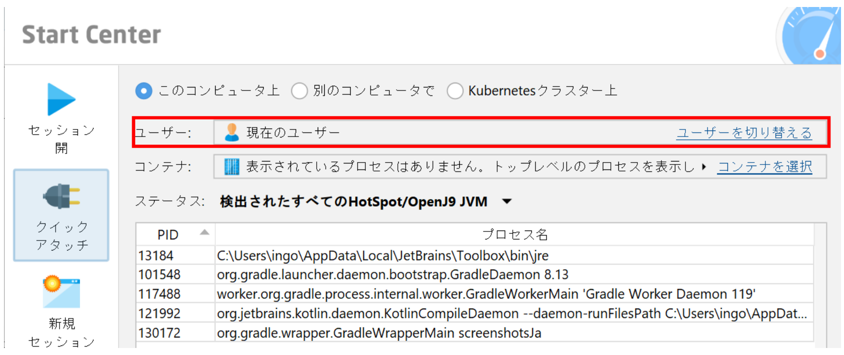

## ローカルサービスへのアタッチ

JVMのアタッチAPIは、呼び出しプロセスがアタッチしたいプロセスと同じユーザーとして実行されることを要求するため、JProfilerによって表示されるJVMのリストは現在のユーザーに限定されます。異なるユーザーによって起動されたプロセスは主にサービスです。サービスにアタッチする方法は、Windows、Linux、およびUnixベースのプラットフォームで異なります。

Windowsでは、アタッチダイアログにサービスを表示ボタンがあり、すべてのローカルで実行中のサービスがリストされます。JProfilerは、どのユーザーで実行されていても、それらのプロセスにアタッチできるようにブリッジ実行ファイルを起動します。




Linuxでは、JProfilerはPolicyKitを通じてUIで直接ユーザーを切り替えることをサポートしています。これはほとんどのLinuxディストリビューションの一部です。アタッチダイアログでユーザーを切り替えるをクリックすると、別のユーザー名を入力し、システムパスワードダイアログで認証できます。




macOSを含むUnixベースのプラットフォームでは、`su`または`sudo`を使用して、異なるユーザーとしてコマンドラインツール`jpenable`を実行できます。UnixバリアントまたはLinuxディストリビューションに応じて、macOSおよびUbuntuのようなDebianベースのLinuxディストリビューションでは`sudo`が使用されます。

sudoを使用する場合、次のように呼び出します：

```
sudo -u userName jpenable
```

suを使用する場合、必要なコマンドラインは次のとおりです：

```
su userName -c jpenable
```

`jpenable`はJVMを選択し、プロファイリングエージェントが待ち受けているポートを教えてくれます。その後、JProfiler　UIからローカルセッションで接続するか、jpenableによって指定されたポートに直接接続するSSH接続を行うことができます。

### リモートマシン上のJVMへのアタッチ

プロファイリングの最も要求の厳しいセットアップはリモートプロファイリングです。JProfiler GUIはローカルマシンで実行され、プロファイルされたJVMは別のマシンで実行されます。プロファイルされたJVMに-agentpath VMパラメータを渡すセットアップでは、リモートマシンにJProfilerをインストールし、ローカルマシンでリモートセッションを設定する必要があります。JProfilerのリモートアタッチ機能では、そのような変更は必要ありません。リモートマシンにログインするためのSSH資格情報が必要なだけです。

SSH接続により、JProfilerは`-agentpath` VMパラメータをプロファイルされたJVMに渡すセットアップでは、リモートマシンにJProfilerをインストールし、ローカルマシンでリモートセッションを設定する必要があります。JProfilerのリモートアタッチ機能では、そのような変更は必要ありません。リモートマシンにログインするためのSSH資格情報が必要なだけです。

SSH接続により、JProfilerは`-agentpath` VMパラメータをプロファイルされたJVMに渡すセットアップでは、リモートマシンにJProfilerをインストールし、ローカルマシンでリモートセッションを設定する必要があります。JProfilerのリモートアタッチ機能では、そのような変更は必要ありません。リモートマシンにログインするためのSSH資格情報が必要なだけです。

自動検出は、SSHログインユーザーとして開始されたリモートマシン上のすべてのJVMをリストします。ほとんどの場合、これはプロファイルしたいサービスを開始したユーザーではありません。サービスを開始するユーザーは通常SSH接続を許可されていないため、JProfilerはユーザーを切り替えるハイパーリンクを追加し、`sudo`または`su`を使用してそのユーザーに切り替えることができます。

複雑なネットワークトポロジーでは、リモートマシンに直接接続できないことがあります。その場合、JProfilerにGUIでマルチホップSSHトンネルで接続するように指示できます。SSHトンネルの終わりで、通常は「127.0.0.1」に直接ネットワーク接続を行うことができます。

他の認証メカニズムについては、OpenSSHトンネルモードを使用できます。OpenSSH実行ファイルを使用する場合のように、コマンドラインでホスト名を入力します。これもOpenSSH設定ファイルで設定されたエイリアスである可能性があります。Windowsでは、Microsoftによる組み込みのOpenSSHクライアントのみがサポートされています。

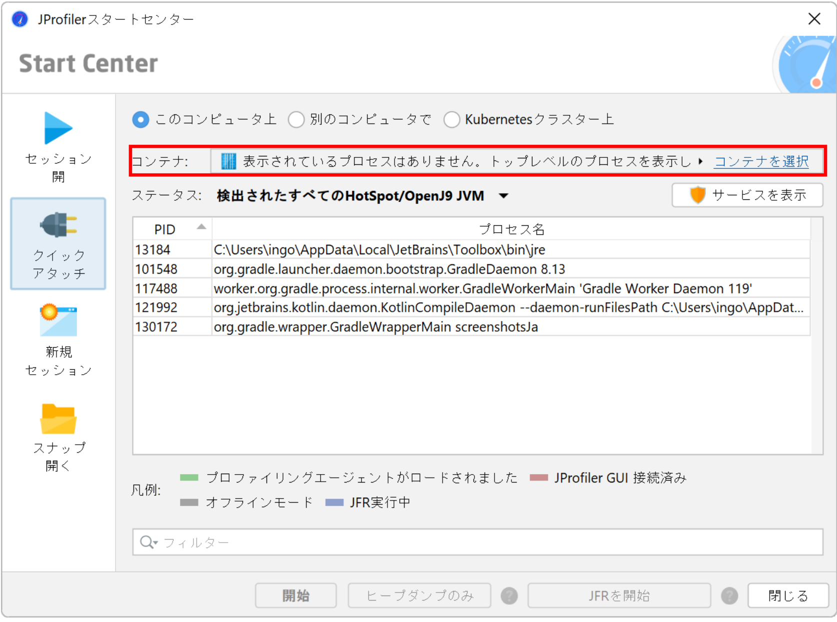
SSHオプションのテキストフィールドは、コマンドラインで指定するOpenSSH実行ファイルへの任意の追加引数を受け取ります。これは、たとえばAWSセッションマネージャーを介してSSH接続をトンネルする方法に関するチュートリアルの指示に従っている場合に特に便利です。

HPROFスナップショットは、SSHログインユーザーとして開始されたJVMに対してのみ取得できます。これは、HPROFスナップショットがJVMを開始したユーザーのアクセス権で書き込まれる中間ファイルを必要とするためです。セキュリティ上の理由から、ダウンロードのためにSSHログインユーザーにファイル権限を転送することはできません。完全なプロファイリングセッションにはそのような制限はありません。

## Dockerコンテナで実行されているJVMへのアタッチ

Dockerコンテナには通常SSHサーバーがインストールされておらず、Dockerコンテナでjpenableを使用することはできますが、Dockerファイルで指定しない限り、プロファイリングポートは外部からアクセスできません。

JProfilerでは、WindowsまたはmacOSのローカルDocker Desktopインストールで実行されているJVMにアタッチすることができ、クイックアタッチダイアログでDockerコンテナを選択します。デフォルトでは、JProfilerは`docker`実行ファイルへのパスを自動的に検出します。あるいは、一般設定ダイアログの「外部ツール」タブでそれを設定できます。



コンテナを選択すると、Dockerコンテナ内で実行されているすべてのJVMが表示されます。JVMを選択すると、JProfilerはDockerコマンドを使用して選択したコンテナにプロファイリングエージェントを自動的にインストールし、プロファイリングの準備をし、プロファイリングプロトコルを外部にトンネルします。

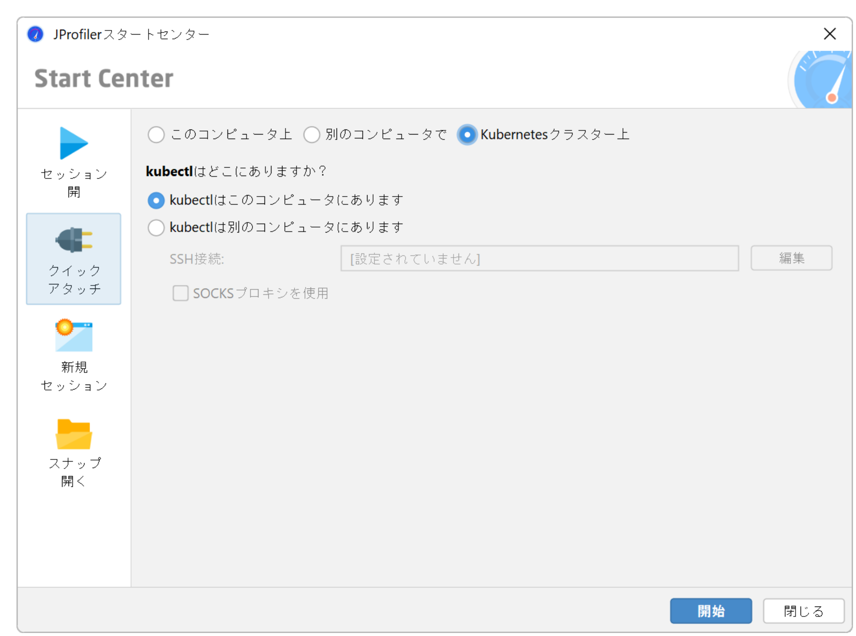
リモートDockerインストールの場合、SSHリモートアタッチ機能を使用してリモートマシン上のDockerコンテナを選択できます。ログインユーザーがdockerグループに属していない場合は、上記のようにユーザーを切り替えることができます。

リモートアタッチダイアログのコンテナを選択ハイパーリンクを使用して、実行中のDockerコンテナを選択し、その中で実行されているすべてのJVMを表示できます。

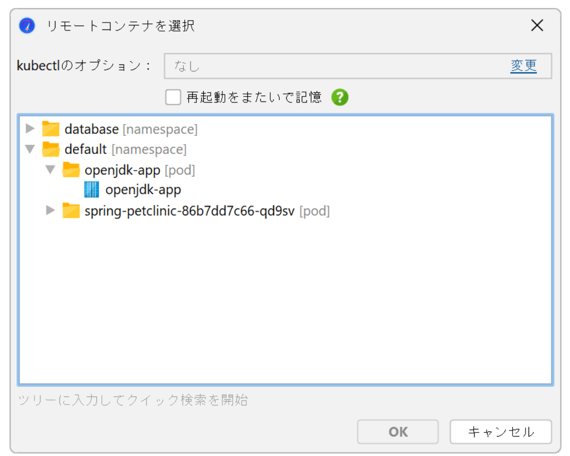
### Kubernetesクラスターで実行されているJVMへのアタッチ

Kubernetesクラスターで実行されているJVMをプロファイルするには、JProfilerは`kubectl`コマンドラインツールを使用して、ポッドやコンテナを発見し、コンテナに接続し、そのJVMをリストし、最終的に選択されたJVMに接続します。

`kubectl`コマンドラインツールは、ローカルコンピュータまたはSSHアクセスを持つリモートマシンで利用可能である場合があります。JProfilerは両方のシナリオを直接サポートしています。ローカルインストールの場合、JProfilerは`kubectl`へのパスを自動的に検出しようとしますが、一般設定ダイアログの「外部ツール」タブで明示的なパスを設定できます。




JProfilerは、検出されたコンテナを3レベルのツリーでリストします。トップにはネームスペースノードがあり、検出されたポッドを含む子ノードがあります。リーフノードはコンテナ自体であり、実行中のJVMの選択に進むために1つを選択する必要があります。




`kubectl`は、Kubernetesクラスターに接続するための認証に追加のコマンドラインオプションを必要とする場合があります。これらのオプションは、コンテナ選択ダイアログの上部に入力できます。これらのオプションは機密情報である可能性があるため、再起動時にそれらを記憶するチェックボックスを明示的に選択した場合にのみディスクに保存されます。このチェックボックスを選択解除すると、以前に保存された情報がすぐにクリアされます。

### 実行中のJVMの表示名の設定

JVM選択テーブルでは、表示されるプロセス名はプロファイルされたJVMのメインクラスとその引数です。exe4jまたはinstall4jによって生成されたランチャーの場合、実行ファイル名が表示されます。

たとえば、同じメインクラスを持つ複数のプロセスがあり、それらを区別できない場合は、表示名を自分で設定することができます。VMパラメータ`-Djprofiler.displayName=[name]`を設定します。名前にスペースが含まれている場合は、シングルクォートを使用します：`-Djprofiler.`

displayName='My name with spaces'。必要に応じて、ダブルクォートでVMパラメータ全体を引用します。-Djprofiler.displayNameに加えて、JProfilerは-Dvisualvm.display.nameも認識します。

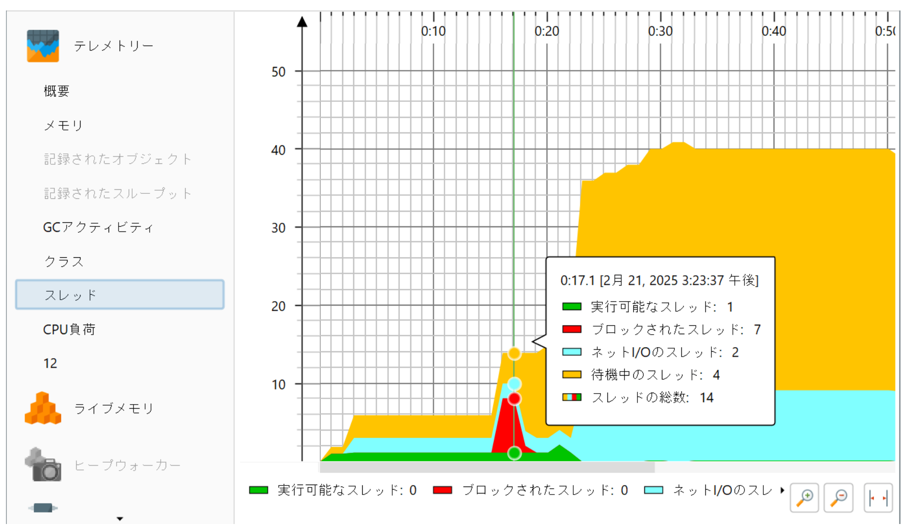
# データの記録

プロファイラの主な目的は、一般的な問題を解決するのに役立つさまざまなソースからのランタイムデータを記録することです。このタスクの主な問題は、実行中のJVMが膨大な量のデータを生成することです。プロファイラが常にすべての種類のデータを記録していた場合、許容できないオーバーヘッドが発生するか、すぐに利用可能なメモリを使い果たしてしまいます。また、特定のユースケースに関連するデータを記録し、無関係なアクティビティを見ないようにしたいことがよくあります。

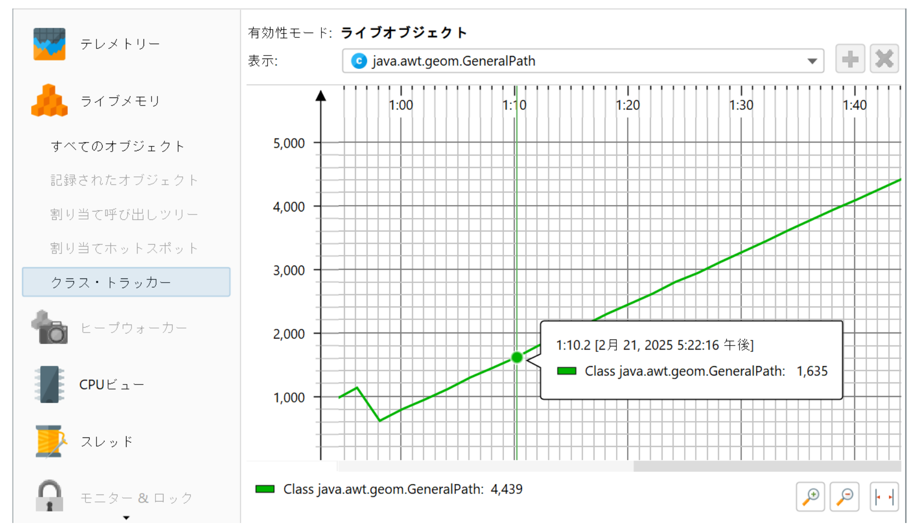
これが、JProfilerが実際に興味のある情報の記録を制御するための細かいメカニズムを提供する理由です。

## スカラー値とテレメトリー

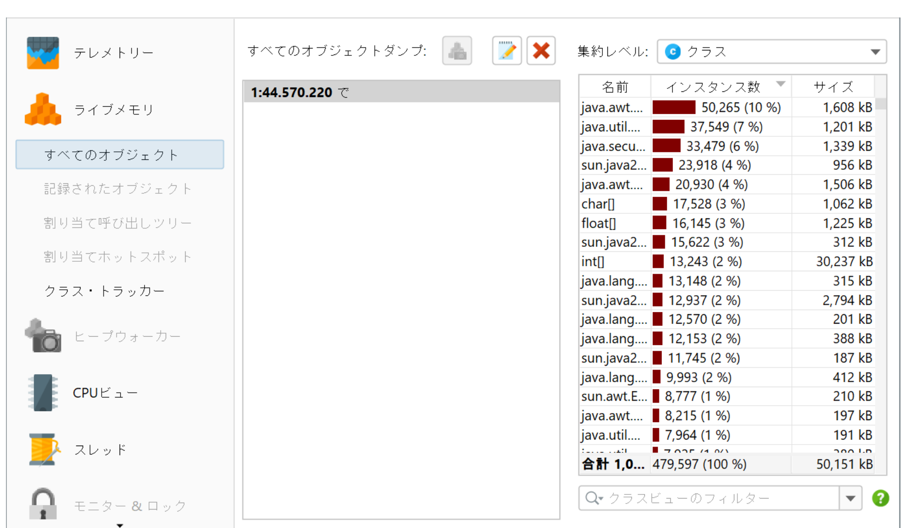
プロファイラの観点から、最も問題の少ないデータ形式はスカラー値です。たとえば、アクティブなスレッドの数やオープンなJDBC接続の数です。JProfilerは通常1秒に1回の固定されたマクロ的な頻度でそのような値をサンプリングし、時間の経過に伴う変化を表示できます。JProfilerでは、そのようなデータを表示するビューは　　　と呼ばれます。ほとんどのテレメトリーは常に記録されます。なぜなら、測定のオーバーヘッドとメモリ消費が小さいからです。データが長時間記録される場合、古いデータポイントは統合され、メモリ消費が時間とともに線形に増加しないようにします。



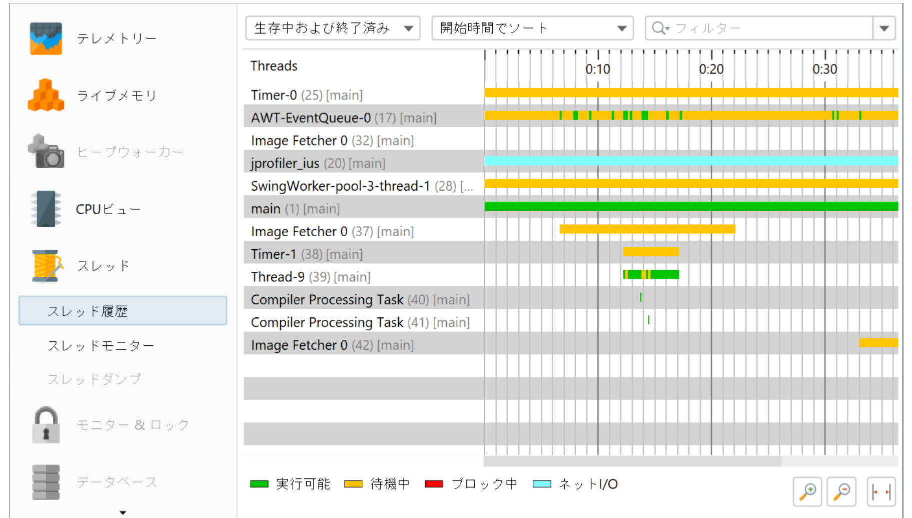

また、各クラスのインスタンス数などのパラメータ化されたテレメトリーもあります。この追加の次元により、恒久的な時系列記録は持続不可能になります。JProfilerに選択されたクラスのインスタンス数のテレメトリーを記録するよう指示できますが、すべてのクラスについて記録するわけではありません。




前の例を続けると、JProfilerはすべてのクラスのインスタンス数を表示できますが、時系列情報はありません。これは「すべてのオブジェクト」ビューであり、各クラスをテーブルの行として表示します。ビューの更新頻度は1秒に1回よりも低く、測定が引き起こすオーバーヘッドに応じて自動的に調整される場合があります。すべてのクラスのインスタンス数を決定することは比較的高価であり、ヒープ上のオブジェクトが多いほど時間がかかります。これが「すべてのオブジェクト」が自動的に更新されず、すべてのオブジェクトの新しいダンプを手動で作成する必要がある理由です。




一部の測定は、スレッドが現在いる実行ステータスのような列挙型の値をキャプチャします。この種の測定は、カラー化されたタイムラインとして表示でき、数値テレメトリーよりもはるかに少ないメモリを消費します。スレッドステータスの場合、「スレッド履歴」ビューはJVM内のすべてのスレッドのタイムラインを表示します。数値値を持つテレメトリーと同様に、古い値は統合され、メモリ消費を減らすためにより粗い粒度にされます。

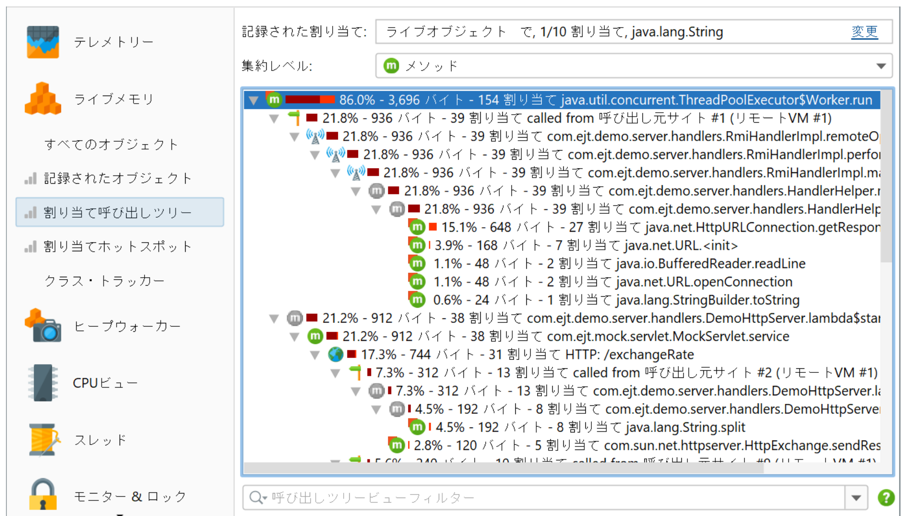



## 割り当て記録

特定の時間間隔に割り当てられたインスタンス数に興味がある場合、JProfilerはすべての割り当てを追跡する必要があります。「すべてのオブジェクト」ビューとは異なり、JProfilerはヒープ内のすべてのオブジェクトを反復して情報をオンデマンドで取得できますが、単一の割り当てを追跡するには、各オブジェクト割り当てに対して追加のコードを実行する必要があります。これにより、非常に高価な測定が行われ、特に多くのオブジェクトを割り当てる場合、プロファイルされたアプリケーションのランタイム特性、特にパフォーマンスホットスポットを大幅に変更する可能性があります。これが、割り当て記録を明示的に開始および停止する必要がある理由です。

記録が関連付けられたビューは、最初は記録ボタンがある空のページを表示します。同じ記録ボタンはツールバーにもあります。

割り当て記録は、割り当てられたインスタンスの数だけでなく、割り当てスタックトレースも記録します。各割り当て記録のスタックトレースをメモリに保持することは過剰なオーバーヘッドを生むため、JProfilerは記録されたスタックトレースをツリーに累積します。これにより、データをはるかに簡単に解釈できるという利点もあります。ただし、時系列の側面は失われ、データから特定の時間範囲を抽出する方法はありません。




## メモリ分析

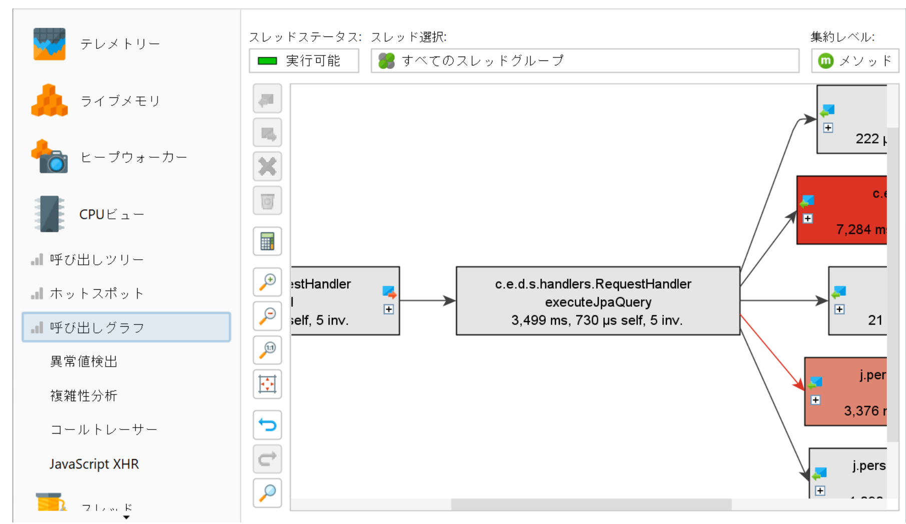
割り当て記録は、オブジェクトがどこで割り当てられたかを測定することしかできず、オブジェクト間の参照に関する情報はありません。メモリリークの解決など、参照を必要とするメモリ分析は、ヒープウォーカーで行われます。ヒープウォーカーはヒープ全体のスナップショットを撮り、それを分析します。これはJVMを一時停止させる侵襲的な操作であり、長時間かかる可能性があり、大量のメモリを必要とします。

より軽量な操作は、ユースケースを開始する前にヒープ上のすべてのオブジェクトをマークし、後でヒープスナップショットを撮るときに新しく割り当てられたオブジェクトを見つけることができるようにすることです。

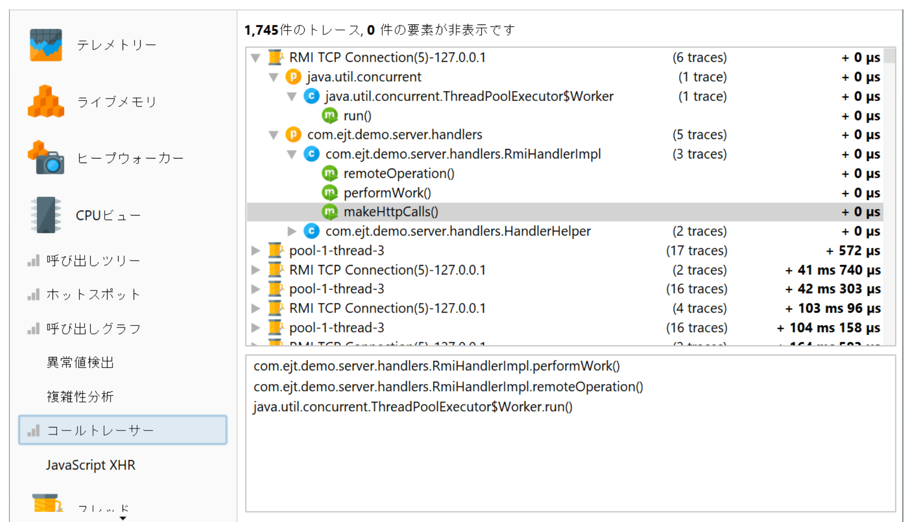
JVMには、古いHPROFプロファイリングエージェントにちなんで名付けられたファイルにヒープ全体をダンプするための特別なトリガーがあります。これはプロファイリングインターフェースとは関係がなく、その制約下で動作しません。このため、HPROFヒープダンプはより高速で、リソースを少なく使用します。欠点は、ヒープウォーカーでヒープスナップショットを表示するときにJVMとのライブ接続がないことと、一部の機能が利用できないことです。

### メソッドコール記録

メソッドコールがどれくらいの時間かかるかを測定することは、割り当て記録と同様にオプションの記録です。メソッドコールはツリーに累積され、呼び出しグラフなど、さまざまな視点から記録されたデータを表示するビューがあります。このタイプのデータの記録は、JProfilerでは「CPU記録」と呼ばれます。

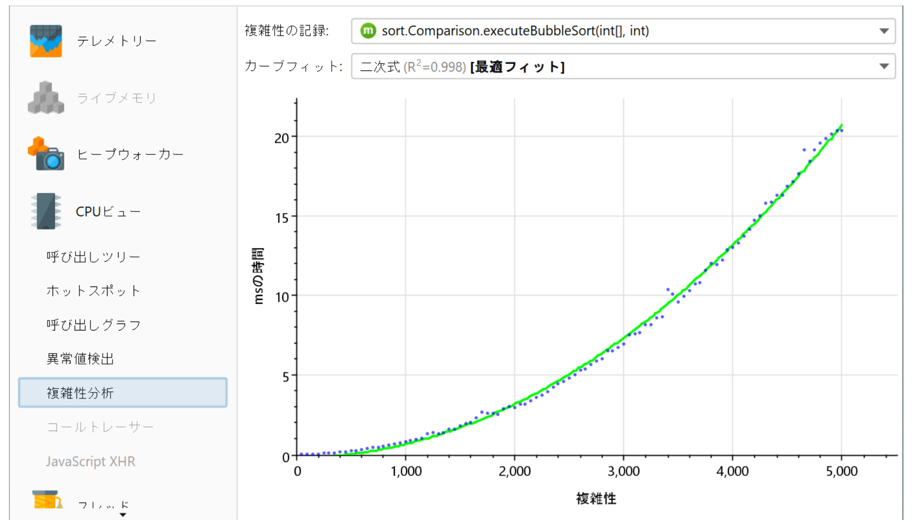



特定の状況下では、特に複数のスレッドが関与している場合、メソッドコールの時系列を確認することが有用です。これらの特別なケースのために、JProfilerは「コールトレーサー」ビューを提供します。このビューには、より一般的なCPU記録に結び付けられていない別の記録タイプがあります。コールトレーサーは、パフォーマンス問題を解決するのに役立つデータを生成しすぎるため、特定の形式のデバッグのみに使用されます。




コールトレーサーはCPU記録に依存しており、必要に応じて自動的にオンにします。

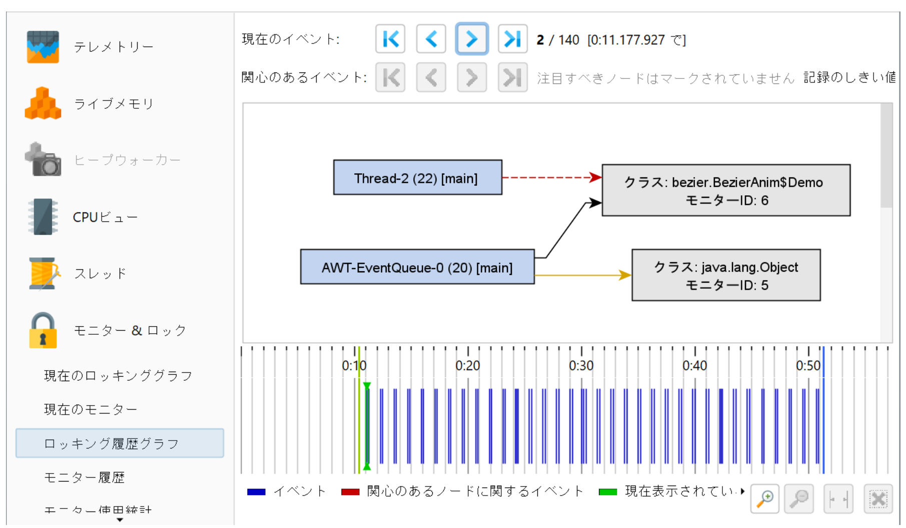
独自の記録を持つ別の特殊なビューは「複雑性分析」です。これは選択されたメソッドの実行時間のみを測定し、CPU記録を有効にする必要はありません。追加のデータ軸は、スクリプトで計算できるメソッドコールのアルゴリズム的複雑性の数値値です。このようにして、メソッドの実行時間がそのパラメータにどのように依存するかを測定できます。




## モニター記録

スレッドが待機またはブロックしている理由を分析するには、対応するイベントを記録する必要があります。そのようなイベントの発生率は大きく異なります。スレッドが頻繁にタスクを調整したり、共通のリソースを共有したりするマルチスレッドプログラムでは、そのようなイベントの数が膨大になる可能性があります。このため、そのような時系列データはデフォルトでは記録されません。

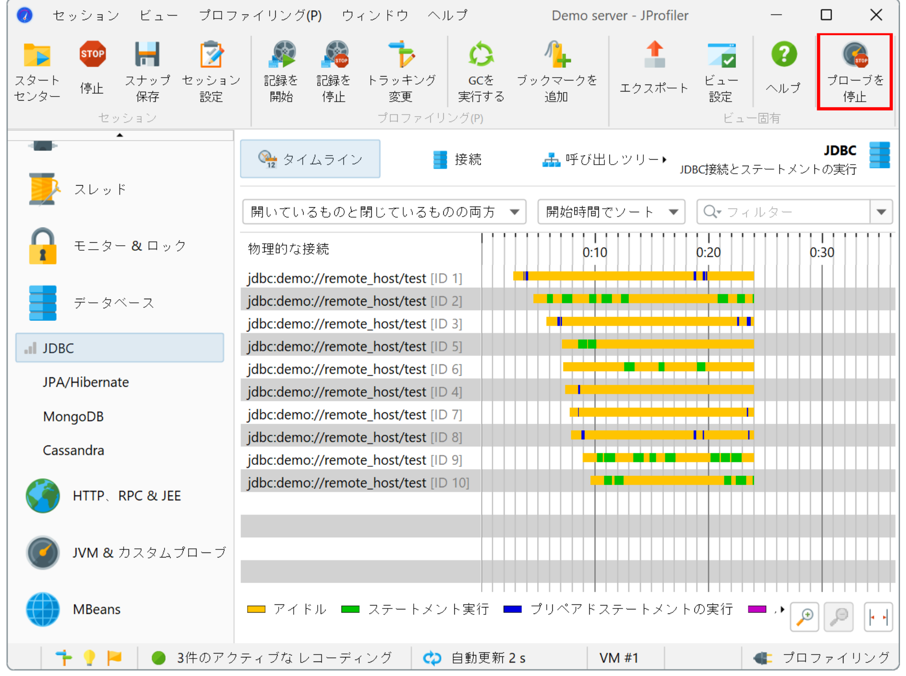
モニター記録をオンにすると、「ロック履歴グラフ」と「モニター履歴」ビューがデータを表示し始めます。



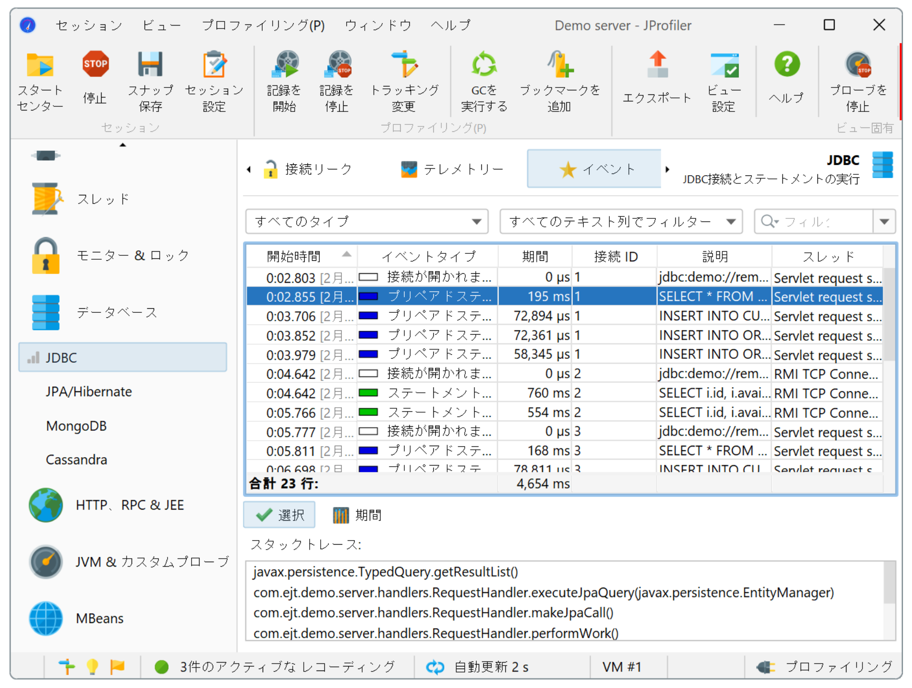

ノイズを排除し、メモリ消費を減らすために、非常に短いイベントは記録されません。ビュー設定でこれらのしきい値を調整することができます。

## プローブ記録

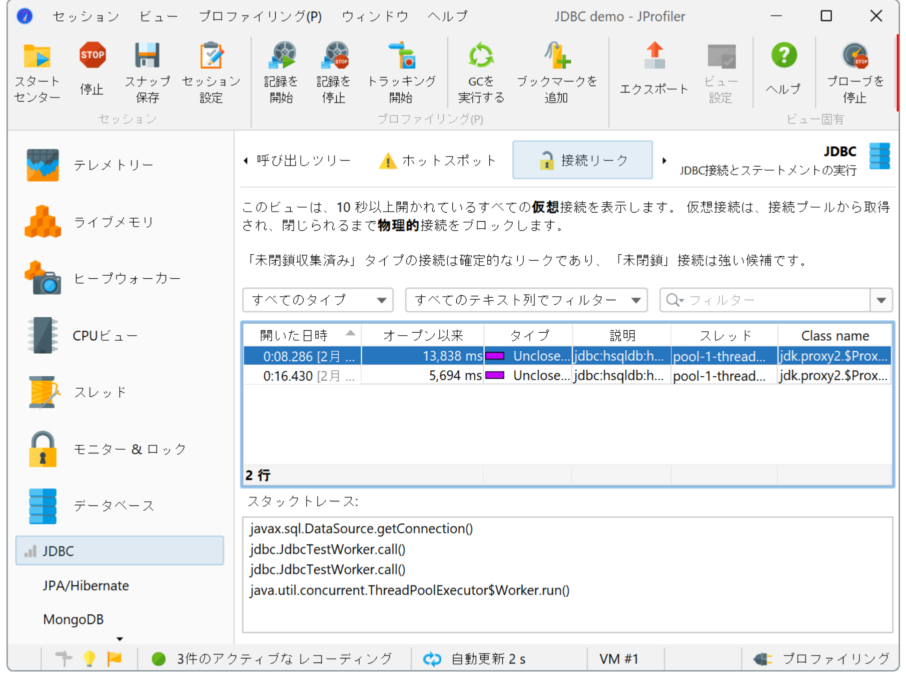
プローブは、JVM内のJDBCコールやファイル操作などの高レベルのサブシステムを示します。デフォルトでは、プローブは記録されず、各プローブの記録を個別に切り替えることができます。アプリケーションが何をしているか、プローブがどのように構成されているかによって、一部のプローブは非常に少ないまたはまったくオーバーヘッドを追加しない場合もあれば、かなりの量のデータを生成する場合もあります。




割り当て記録やメソッドコール記録と同様に、プローブデータは累積され、時系列情報はタイムラインやテレメトリーを除いて破棄されます。ただし、ほとんどのプローブには「イベント」ビューもあり、単一のイベントを調査できます。これにより、潜在的に大きなオーバーヘッドが追加され、別の記録アクションがあります。その記録アクションのステータスは永続的であるため、プローブ記録を切り替えると、以前にオンにしていた場合は関連するイベント記録も切り替わります。

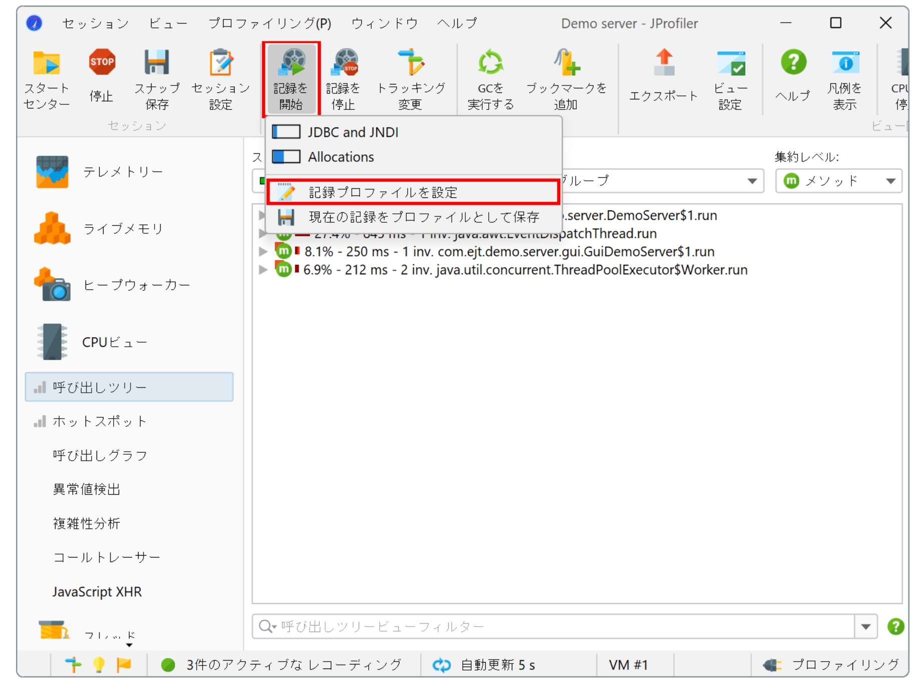


JDBCプローブには、JDBC接続リークを記録するための3番目の記録アクションがあります。接続リークを調査しようとしている場合にのみ、接続リークを探すことに関連するオーバーヘッドが発生します。イベント記録アクションと同様に、リーク記録アクションの選択状態は永続的です。

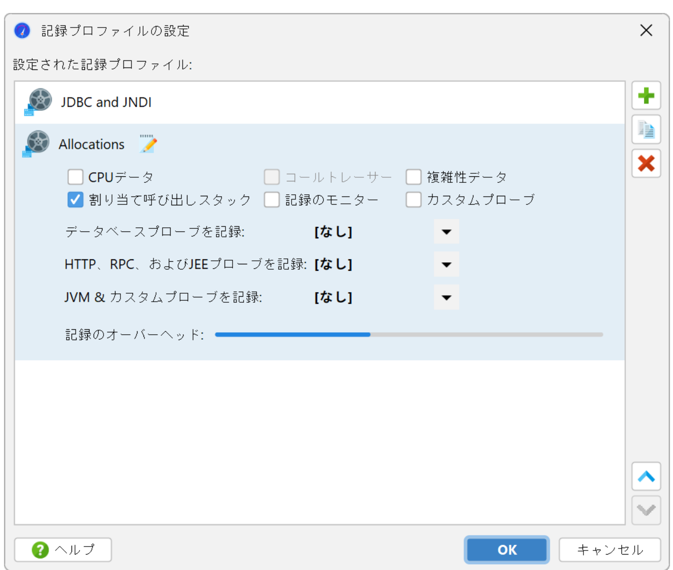


## 記録プロファイル

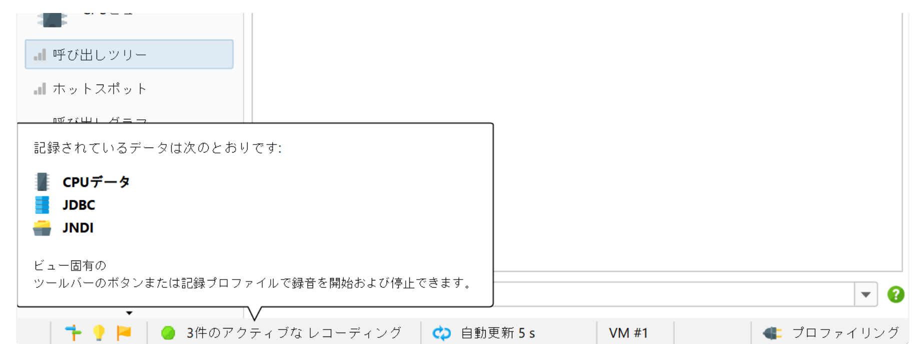
多くの状況で、単一のクリックでさまざまな記録を開始または停止したいことがあります。対応するビューをすべて訪れて、記録ボタンを一つずつ切り替えるのは非現実的です。これが、JProfilerに記録プロファイルがある理由です。記録プロファイルは、ツールバーの記録開始ボタンをクリックして作成できます。




記録プロファイルは、原子的にアクティブ化できる特定の記録の組み合わせを定義します。JProfilerは、選択した記録によって引き起こされるオーバーヘッドについて大まかな印象を与えようとし、問題のある組み合わせを避けるようにします。特に、割り当て記録とCPU記録は一緒にうまく機能しません。なぜなら、割り当て記録がCPUデータのタイミングを大幅に歪めるからです。




セッションが実行中のときにいつでも記録プロファイルをアクティブ化できます。記録プロファイルは加算的ではなく、記録プロファイルに含まれていないすべての記録を停止します。記録停止ボタンを使用すると、どのようにアクティブ化されたかに関係なく、すべての記録を停止します。現在アクティブな記録を確認するには、ステータスバーの記録ラベルにマウスをホバーします。

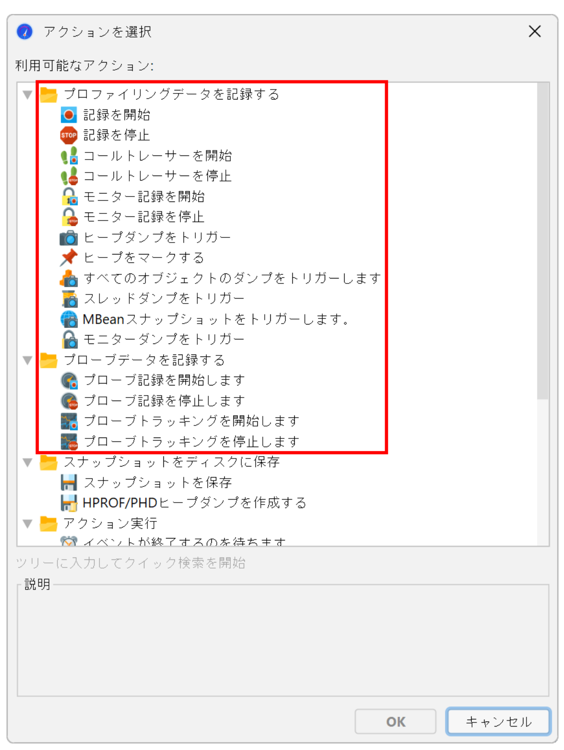



記録プロファイルは、プロファイリングを開始するときに直接アクティブ化することもできます。「セッション開始」ダイアログには、初期記録プロファイルドロップダウンがあります。デフォルトでは、記録プロファイルは選択されていませんが、JVMの起動フェーズからデータが必要な場合は、ここで必要な記録を設定します。

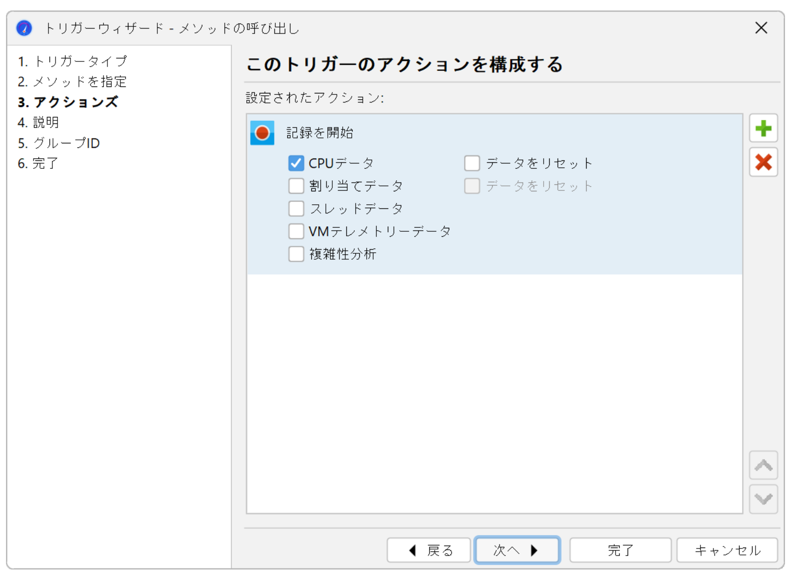
## トリガーを使用した記録

特定の条件が発生したときに記録を開始したい場合があります。JProfilerには、アクションのリストを実行するトリガーを定義するためのシステム[p.130]があります。利用可能なトリガーアクションには、アクティブな記録の変更も含まれます。

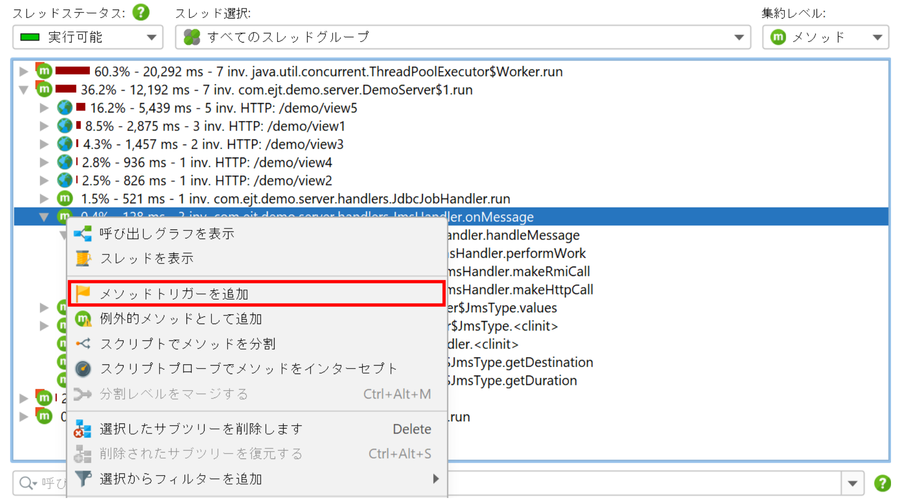
たとえば、特定のメソッドが実行されたときにのみ記録を開始したい場合があります。その場合、セッション設定ダイアログに移動し、トリガー設定タブをアクティブにして、そのメソッドのメソッドトリガーを定義します。アクション設定では、さまざまな記録アクションを選択できます。



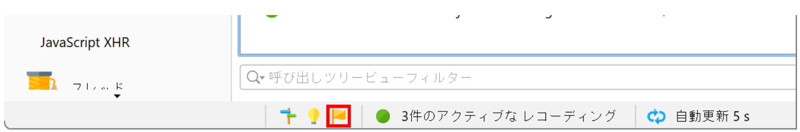

「記録開始」アクションは、パラメータなしでそれらの記録を制御します。通常、記録を停止して再開すると、以前に記録されたデータはすべてクリアされます。「CPUデータ」と「割り当てデータ」の記録では、以前のデータを保持し、複数の間隔にわたって累積を続けるオプションもあります。




メソッドトリガーは、コンテキストメニューの「メソッドトリガーを追加」アクションを使用して、呼び出しツリーで便利に追加できます。同じセッションにすでにメソッドトリガーがある場合は、既存のトリガーにメソッドインターセプションを追加することを選択できます。




デフォルトでは、トリガーはプロファイリングのためにJVMが開始されたときにアクティブです。起動時にトリガーを無効にする方法は2つあります。トリガー設定で個別に無効にするか、セッション開始ダイアログの起動時にトリガーを有効にするチェックボックスをオフにします。ライブセッション中に、メニューからプロファイリング->(有効|無効) トリガーを選択するか、ステータスバーの トリガー記録状態アイコンをクリックして、すべてのトリガーを有効または無効にできます。




時には、同時にトリガーのグループのアクティベーションを切り替える必要があります。これは、興味のあるトリガーに同じグループIDを割り当てて、メニューからプロファイリング->トリガーグループを有効にするを呼び出すことで可能です。

## jpcontrollerを使用した記録

JProfilerには、すでにプロファイルされている任意のJVMで記録を制御するためのコマンドライン実行可能ファイルがあります。`jpcontroller`は、JProfiler MBeanが公開されていることを要求します。そうでない場合、プロファイルされたJVMに接続できません。これは、プロファイリングエージェントがすでにプロファイリング設定を受け取っている場合にのみ当てはまります。プロファイリング設定がない場合、エージェントは正確に何を記録するかを知りません。

次のいずれかの条件が適用される必要があります。

- JProfiler GUIでJVMにすでに接続している
- プロファイルされたJVMが、`-agentpath` VMパラメータを使用して開始され、`nowait`および`config`パラメータの両方が含まれている。この場合、統合ウィザードでは、すぐに起動モードと起動時に設定を適用オプションに対応します。
- JVMが`jpenable`実行可能ファイルでプロファイリングの準備がされ、`-offline`パラメータが指定されている。詳細については、`jpenable -help`の出力を参照してください。

特に、プロファイルされたJVMが`nowait`フラグのみで開始された場合、`jpcontroller`は機能しません。統合ウィザードでは、JProfiler GUIで接続時に設定を適用オプションが設定同期ステップでそのようなパラメータを構成します。詳細については、起動時のプロファイリング設定の設定に関するヘルプトピック[p. 250]を参照してください。

jpcontrollerは、すべての記録とそのパラメータのためのループするマルチレベルメニューを提供します。また、スナップショットを保存することもできます。

```
ingo@ubuntu: ~
ingo@ubuntu:~$ sudo -u tomcat8 jprofiler10/bin/jpcontroller
Connecting to org.apache.catalina.startup.Bootstrap start [6125] ...
Starting JMX management agent ...
Connection established successfully.

Please select an operation:

Start recording [1]
Stop recording [2]
Enable triggers [3]
Disable triggers [4]
Heap dump [5]
Thread dump [6]
Add bookmark [7]
Save snapshot [8]
Quit [9]
```

### プログラムによる記録の開始方法

記録を開始するもう一つの方法は、APIを通じて行うことです。プロファイルされたVMでは、`com.jprofiler.api.controller.Controller`クラスを呼び出して、プログラムで記録を開始および停止できます。詳細とコントローラークラスを含むアーティファクトの取得方法については、オフラインプロファイリングに関する章[p. 130]を参照してください。

別のJVMで記録を制御したい場合、`jpcontroller`でも使用されるプロファイルされたJVMの同じMBeanにアクセスできます。MBeanのプログラムによる使用を設定することはやや複雑で、かなりの手間がかかるため、JProfilerには再利用できる例が付属しています。ファイル`api/samples/mbean/src/MBeanProgrammaticAccessExample.java`を確認してください。これは、別のプロファイルされたJVMで5秒間CPUデータを記録し、スナップショットをディスクに保存します。

# スナップショット

これまで、JProfiler GUIがプロファイルされたJVM内で実行されているプロファイリングエージェントからデータを取得するライブセッションのみを見てきました。JProfilerは、すべてのプロファイリングデータがファイルに書き込まれるスナップショットもサポートしています。これは、いくつかのシナリオで有利です：

- プロファイリングデータを自動的に記録する場合、例えばテストの一部として、JProfiler GUIに接続することができない場合。
- 異なるプロファイリングセッションからのプロファイリングデータを比較したり、古い記録を見たりしたい場合。
- 他の人とプロファイリングデータを共有したい場合。

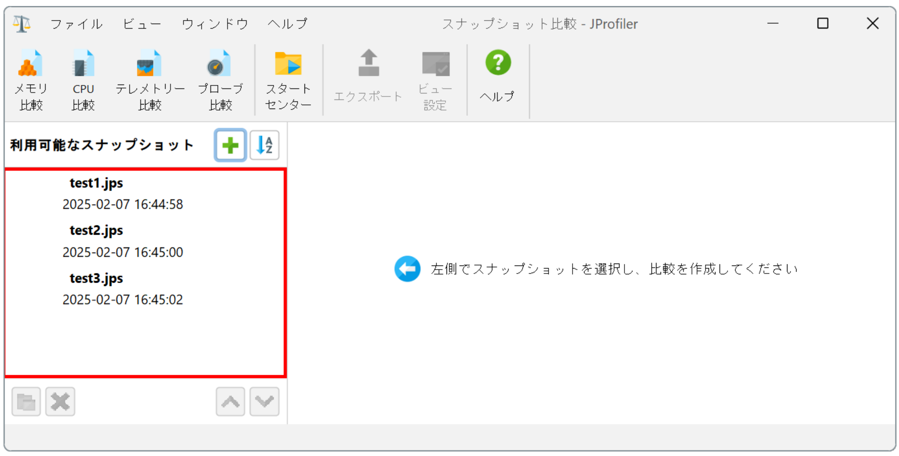
スナップショットには、ヒープスナップショットを含むすべての記録からのデータが含まれています。ディスクスペースを節約するために、スナップショットは圧縮されますが、ヒープウォーカーデータは直接メモリマッピングを可能にするために非圧縮のままにする必要があります。

## JProfiler GUIでのスナップショットの保存と開く

ライブセッションをプロファイリングしているとき、スナップショットを保存ツールバーボタンでスナップショットを作成できます。JProfilerはリモートエージェントからすべてのプロファイリングデータを取得し、".jps"拡張子のローカルファイルに保存します。ライブセッション中に複数のスナップショットを保存することができます。それらは自動的に開かれず、プロファイルを続けることができます。

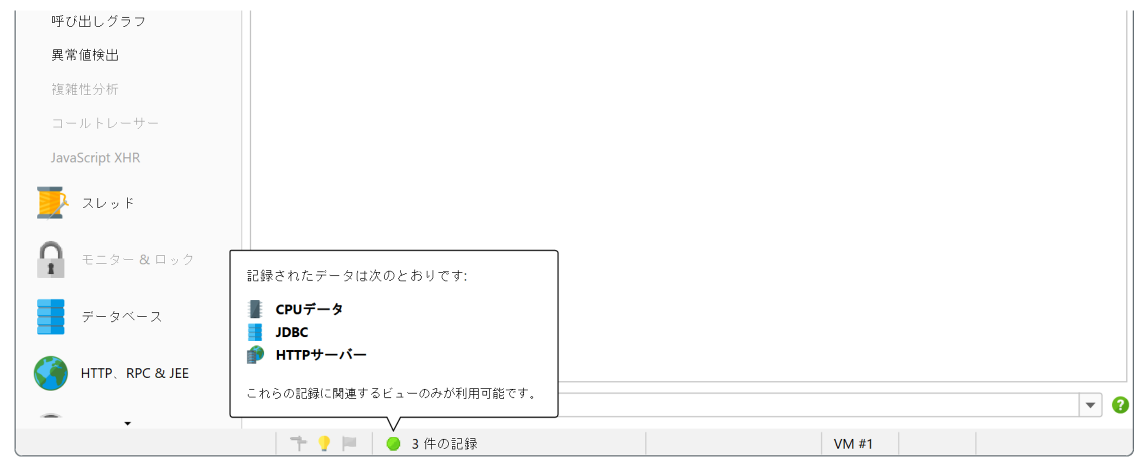
保存されたスナップショットはファイル->最近のスナップショットメニューに自動的に追加されるので、保存したばかりのスナップショットを便利に開くことができます。ライブセッションがまだ実行中のときにスナップショットを開くと、ライブセッションを終了するか、別のJProfilerウィンドウを開くかを選択できます。

JProfilerでスナップショット比較機能を使用すると、現在のライブセッションのために保存したすべてのスナップショットでリストが埋められます。これにより、異なるユースケースを簡単に比較できます。



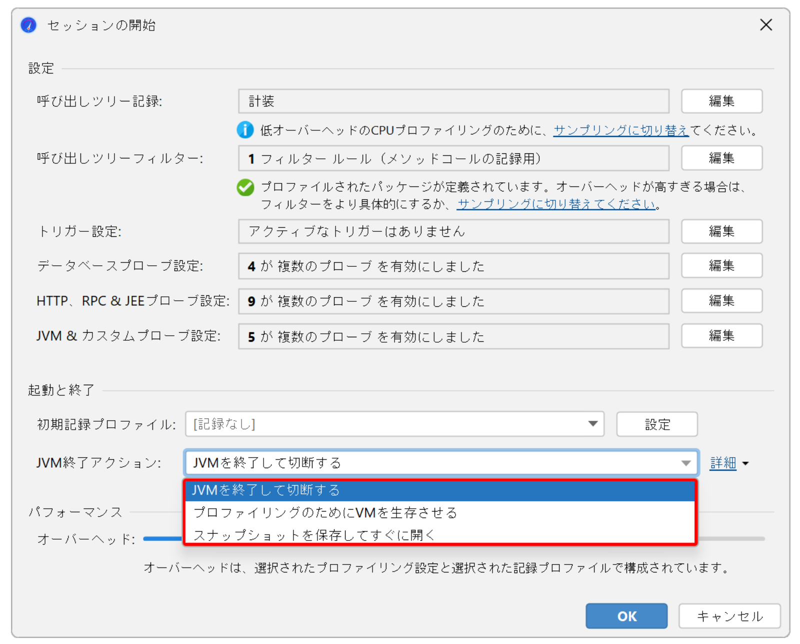

一般的に、スナップショットはメインメニューのセッション->スナップショットを開くを呼び出すか、ファイルマネージャーでスナップショットファイルをダブルクリックすることで開くことができます。JProfilerのIDE統合も、IDE自体の一般的なファイルを開くアクションを通じてJProfilerスナップショットを開くことをサポートしています。その場合、組み込みのソースコードビューアーの代わりにIDEへのソースコードナビゲーションが得られます。

スナップショットを開くと、すべての記録アクションが無効になり、記録されたデータを持つビューのみが利用可能です。どの種類のデータが記録されたかを確認するには、ステータスバーの記録ラベルにマウスをホバーします。




## 短命プログラムのプロファイリング

ライブセッションでは、すべてのプロファイリングデータがプロファイルされたJVMのプロセスに存在します。そのため、プロファイルされたJVMが終了すると、JProfilerのプロファイリングセッションも閉じられます。JVMが終了したときにプロファイリングを続けるには、セッション開始ダイアログで有効にできる2つのオプションがあります。



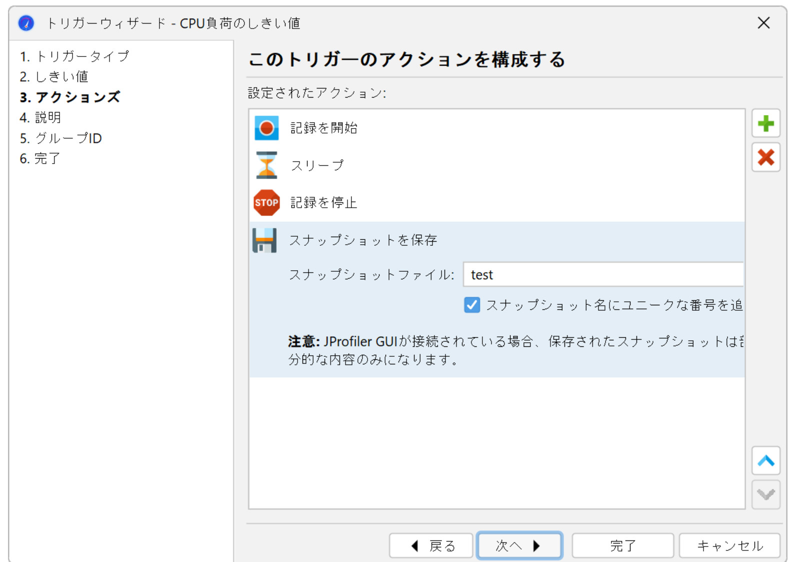

- JVMが実際に終了するのを防ぎ、JProfiler GUIが接続されている限り人工的に生かし続けることができます。これは、IDEからテストケースをプロファイリングしているときに、IDEのテストコンソールでステータスと合計時間を確認したい場合には望ましくないかもしれません。
- JVMが終了したときにJProfilerにスナップショットを保存させ、すぐにそれに切り替えるように依頼できます。スナップショットは一時的なもので、最初にスナップショットを保存アクションを使用しない限り、セッションを閉じると破棄されます。

### トリガーでスナップショットを保存する

自動プロファイリングセッションの最終結果は常にスナップショットまたは一連のスナップショットです。トリガーでは、プロファイルされたJVMが実行されているマシンにスナップショットを保存する「スナップショットを保存」アクションを追加できます。トリガーがライブセッション中に実行されると、スナップショットはリモートマシンにも保存され、すでにJProfiler GUIに送信されたデータの一部を含まない場合があります。

トリガーでスナップショットを保存するための基本的な戦略は2つあります：

- テストケースの場合、「JVM起動」トリガーで記録を開始し、JVMが終了したときにスナップショットを保存する「JVM終了」トリガーを追加します。
- 「CPU負荷しきい値」トリガーのような例外的な条件や、「タイマートリガー」を使用した定期的なプロファイリングの場合、「スリープ」アクションを挟んでデータを記録した後にスナップショットを保存します。




## HPROFヒープスナップショット

ヒープスナップショットを取ることが過剰なオーバーヘッドを生じたり、メモリを消費しすぎたりする状況では、JVMが提供するビルトイン機能としてのHPROFヒープスナップショットを使用できます。この操作にはプロファイリングエージェントが必要ないため、プロダクションで実行されているJVMのメモリ問題を分析するのに興味深いです。

JProfilerを使用すると、そのようなスナップショットを取得する方法は3つあります：

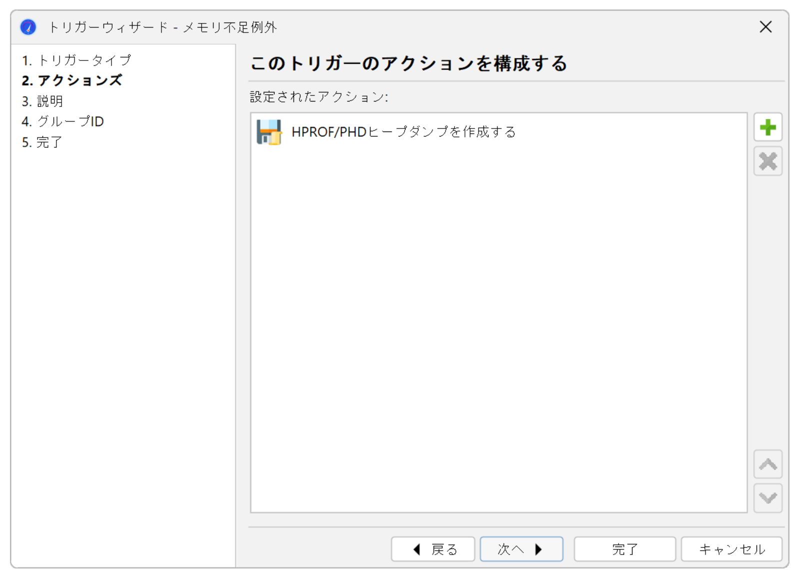
- ライブセッションの場合、JProfiler GUIはメインメニューにHPROFヒープダンプをトリガーするアクションを提供します。

- JProfilerには、`OutOfMemoryError`がスローされたときにHPROFスナップショットを保存する特別な「メモリ不足例外」トリガーがあります。これは、HotSpot JVMがサポートする VMパラメータ [(1)]

```
-XX:+HeapDumpOnOutOfMemoryError
```

  に対応しています。

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




- JDKのjmap実行可能ファイル [2]を使用して、実行中のJVMからHPROFヒープダンプを抽出できます。

  JProfilerには、jmapよりも多用途なコマンドラインツール`jpdump`が含まれています。これにより、プロセスを選択でき、Windowsでサービスとして実行されているプロセスに接続でき、32ビット/64ビットの混在JVMに問題がなく、HPROFスナップショットファイルを自動的に番号付けします。詳細については、`-help`オプションで実行してください。

### JDKフライトレコーダースナップショット

JProfileは、Java Flight Recorder (JFR)によって保存されたスナップショットの開封を完全にサポートしています。この場合、UIは著しく異なり、JFRの機能に合わせて調整されています。詳細については、JFRヘルプトピック [p. 223]を参照してください。

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

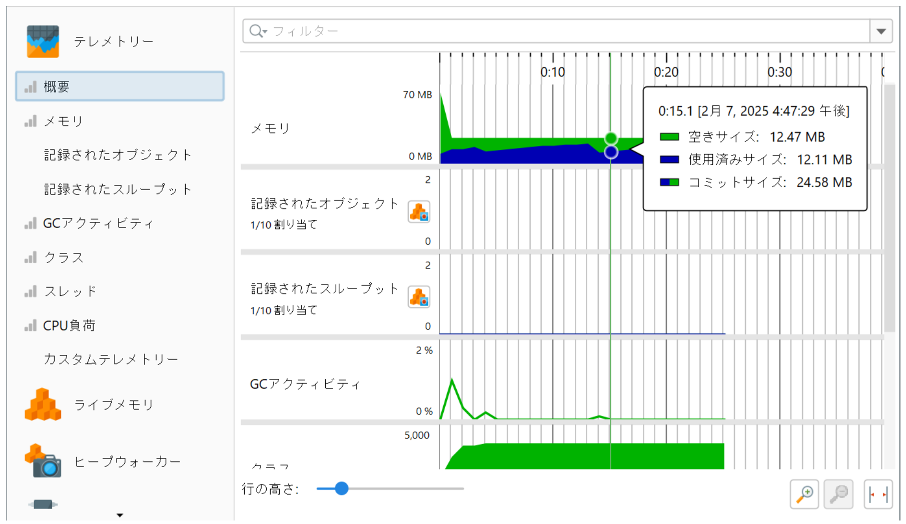
# テレメトリー

プロファイリングの一環として、時間経過に伴うスカラー測定値を監視することがあります。例えば、使用されたヒープサイズです。JProfilerでは、このようなグラフをテレメトリーと呼びます。テレメトリーを観察することで、プロファイルされたソフトウェアの理解が深まり、重要なイベントを異なる測定値で関連付けることができ、予期しない動作に気付いた場合にはJProfilerの他のビューでより深い分析を行うことができます。

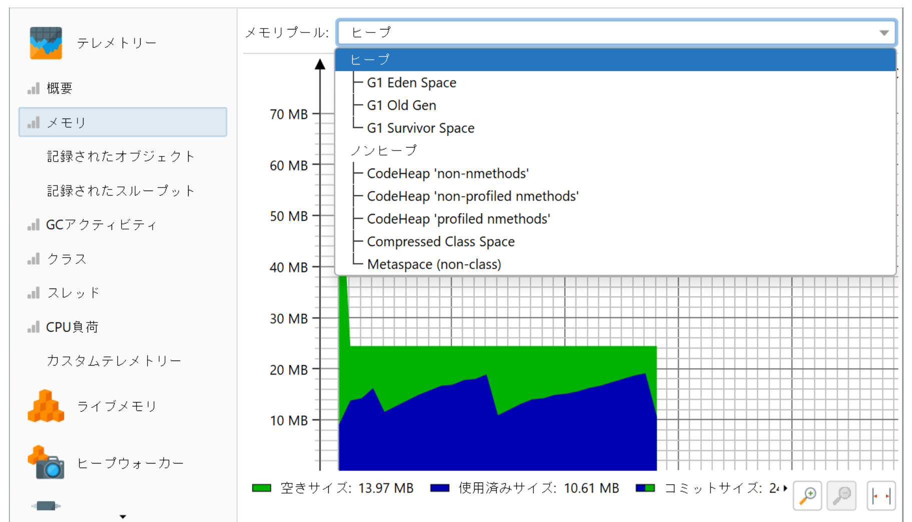
## 標準テレメトリー

JProfiler UIの「テレメトリー」セクションでは、デフォルトでいくつかのテレメトリーが記録されます。インタラクティブセッションでは、常に有効です。特定のデータタイプが記録される必要があるテレメトリーもあります。 その場合、テレメトリーに記録アクションが表示されます。

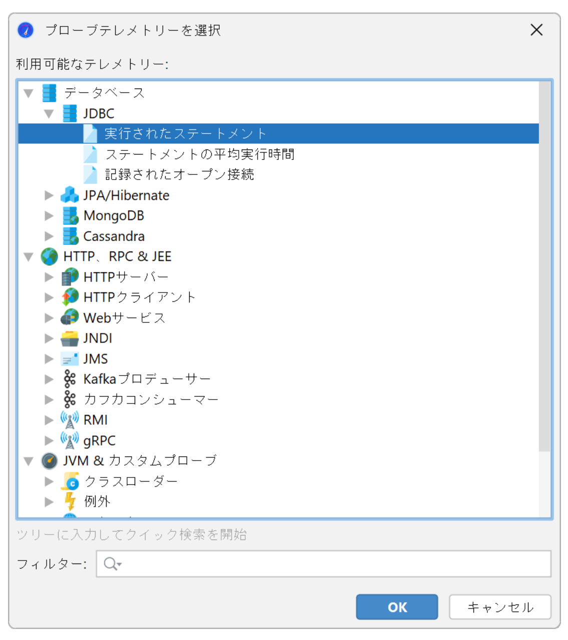
同じ時間軸で複数のテレメトリーを比較するために、概要には複数の小規模なテレメトリーが上下に表示され、行の高さを設定できます。 テレメトリーのタイトルをクリックすると、完全なテレメトリービューがアクティブになります。概要でのテレメトリーのデフォルト順序は必ずしも適切ではないかもしれません。例えば、選択したテレメトリーを並べて関連付けたい場合です。その場合、ドラッグ＆ドロップで 概要で順序を変更できます。



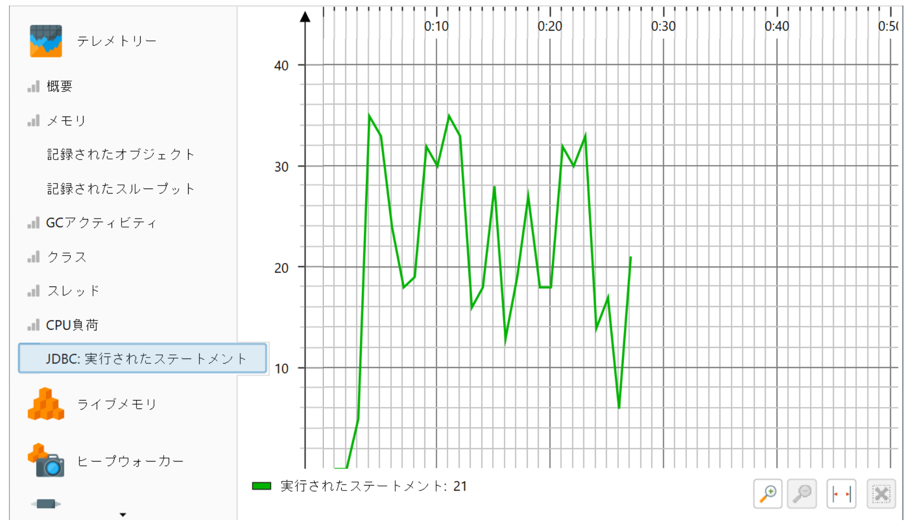

完全なビューには現在の値を示す凡例が表示され、概要に表示されている以上のオプションがある場合があります。 例えば、「メモリ」テレメトリーでは、単一のメモリプールを選択できます。



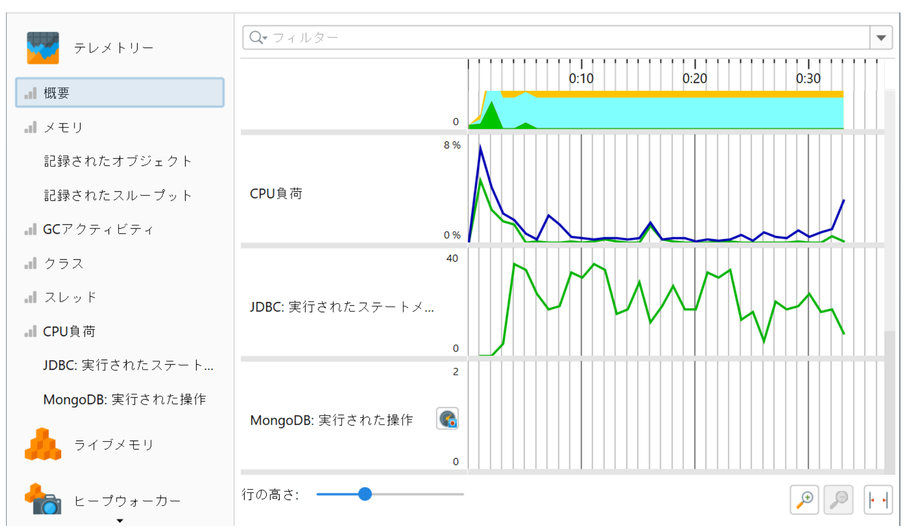

JProfilerには、JVMや重要なフレームワークの高レベルシステムからイベントを記録する多くのプローブ [p. 105]があります。 プローブには対応するプローブビューに表示されるテレメトリーがあります。これらのテレメトリーをシステムテレメトリーと比較するために、選択したプローブテレメトリーをトップレベルのテレメトリーセクションに追加できます。ツールバーから ➕ テレメトリーを追加->プローブテレメトリー を選択し、1つ以上のプローブテレメトリーを選択します。



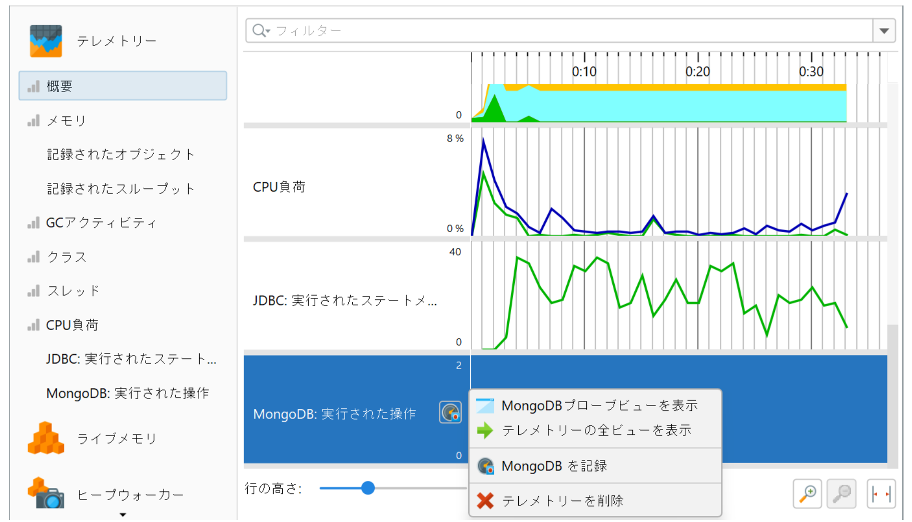

追加された各プローブテレメトリーはテレメトリーセクションに独自のビューを持ち、概要にも表示されます。




プローブテレメトリーが追加されると、プローブデータが記録されている場合にのみデータが表示されます。そうでない場合、テレメトリーの説明には記録を開始するためのインラインボタンが含まれています。

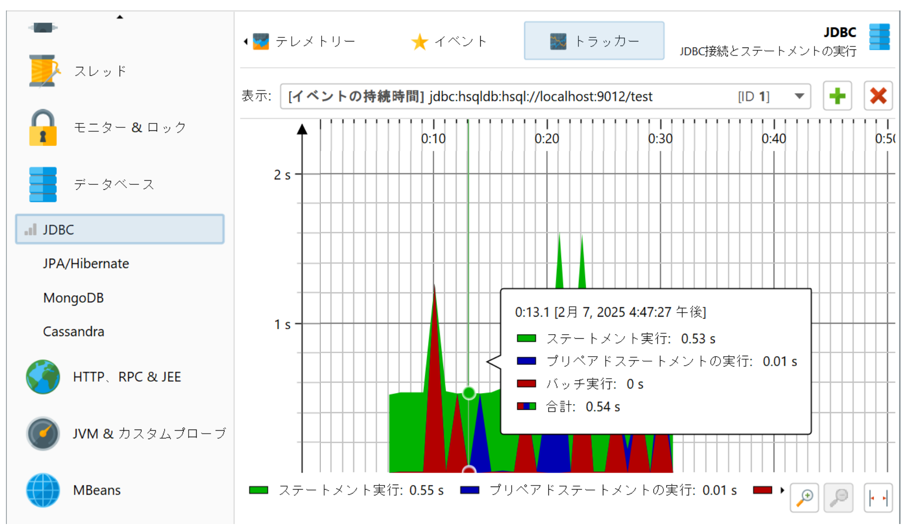



プローブテレメトリーのコンテキストメニューには、記録アクションと対応するプローブビューを表示するアクションが含まれています。



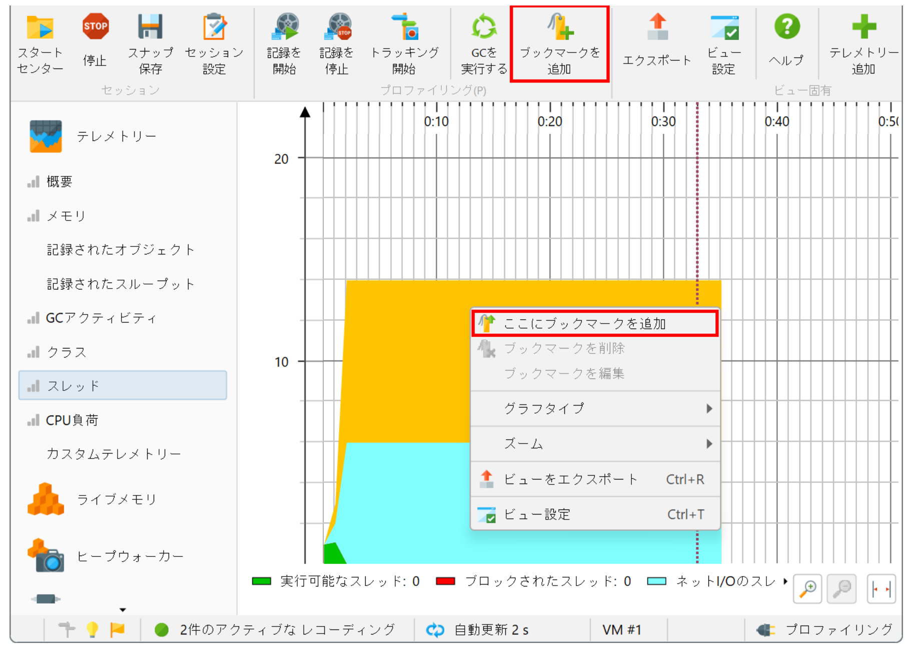

プローブビューと同様に、記録されたオブジェクトのVMテレメトリーはメモリ記録に依存し、記録ボタンと同様のコンテキストメニューを持っています。

最後に、他のビューで選択されたスカラー値を監視する「トラッキング」テレメトリーがあります。例えば、クラス・トラッカービューでは、クラスを選択してそのインスタンス数を時間と共に監視できます。　また、各プローブには選択されたホットスポットや制御オブジェクトを監視する「トラッカー」ビューがあります。




## ブックマーク

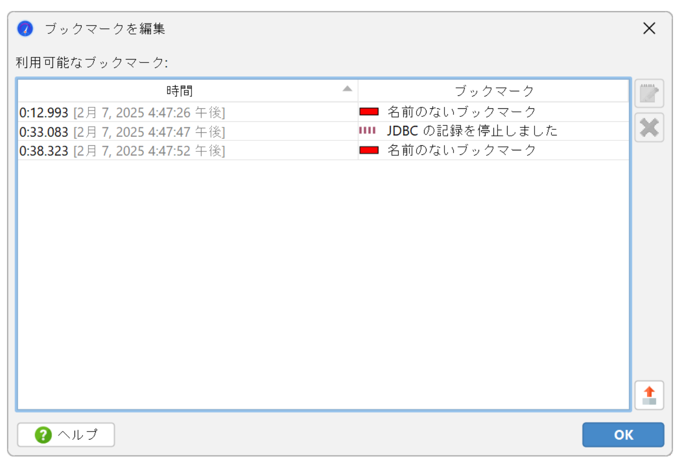
JProfilerはすべてのテレメトリーに表示されるブックマークのリストを維持しています。インタラクティブセッションでは、ブックマークを追加ツールバーボタンをクリックするか、コンテキストメニューのここにブックマークを追加　機能を使用して、現在の時間にブックマークを追加できます。




ブックマークは手動で作成するだけでなく、特定の記録の開始と終了を示すために記録アクションによって自動的に追加されます。 トリガーアクションやコントローラAPIを使用して、プログラムでブックマークを追加できます。

ブックマークには色、線のスタイル、ツールチップに表示される名前があります。既存のブックマークを編集してこれらのプロパティを変更できます。

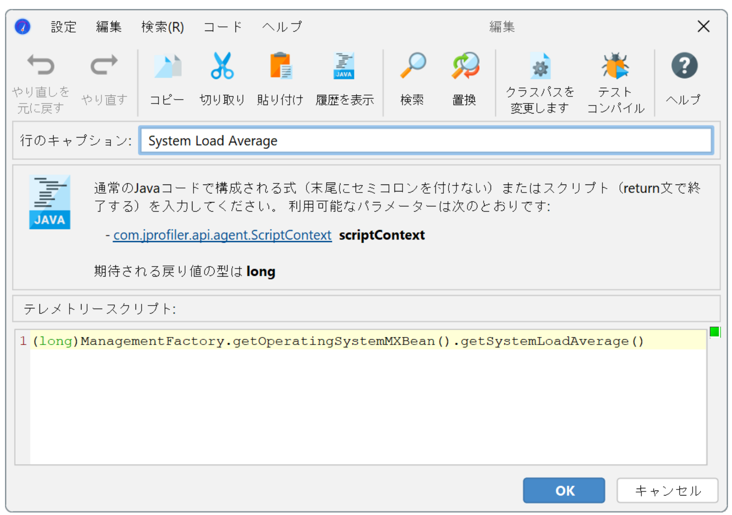
テレメトリーで複数のブックマークを右クリックするのが不便な場合は、 プロファイリング->ブックマークを編集アクションをメニューから使用してブックマークのリストを取得できます。 ここでは、ブックマークをHTMLまたはCSVにエクスポートすることもできます。



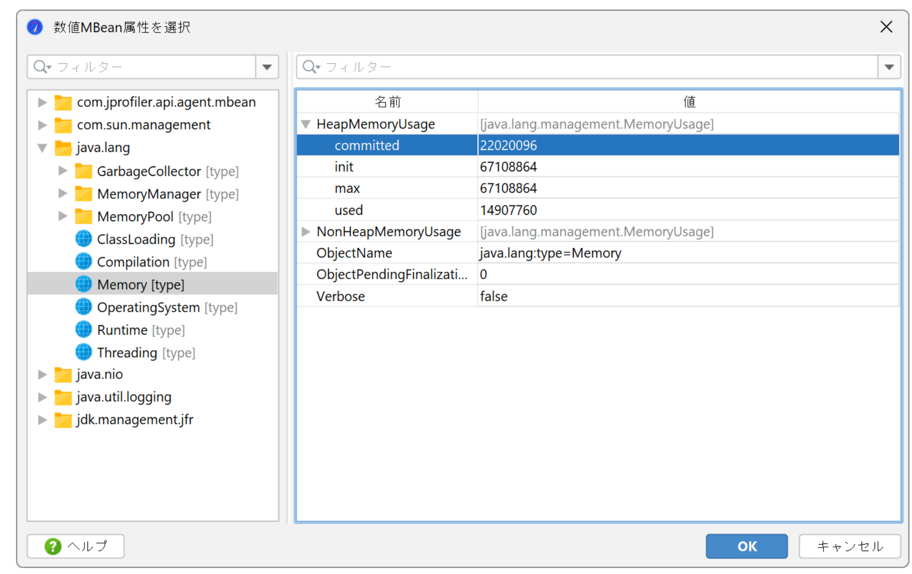

### カスタムテレメトリー

独自のテレメトリーを追加する方法は2つあります。JProfiler UIでスクリプトを書いて数値を提供するか、 数値のMBean属性を選択するかです。

カスタムテレメトリーを追加するには、 「テレメトリー」セクションに表示されるテレメトリーを設定ツールバーボタンをクリックします。 スクリプトテレメトリーでは、現在のJProfilerセッションのクラスパスに設定されているすべてのクラスにアクセスできます。 値が直接利用できない場合は、アプリケーションに静的メソッドを追加し、このスクリプトで呼び出すことができます。

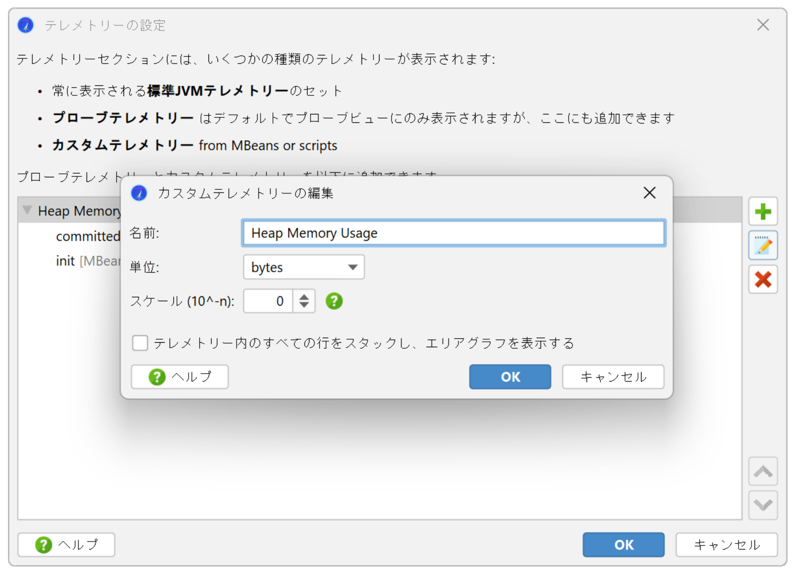



上記の例は、プラットフォームMBeanへの呼び出しを示しています。MBeanのスカラー値をグラフ化するには、MBeanテレメトリーを使用する方が便利です。 ここでは、MBeanブラウザを使用して適切な属性を選択できます。属性値は数値でなければなりません。

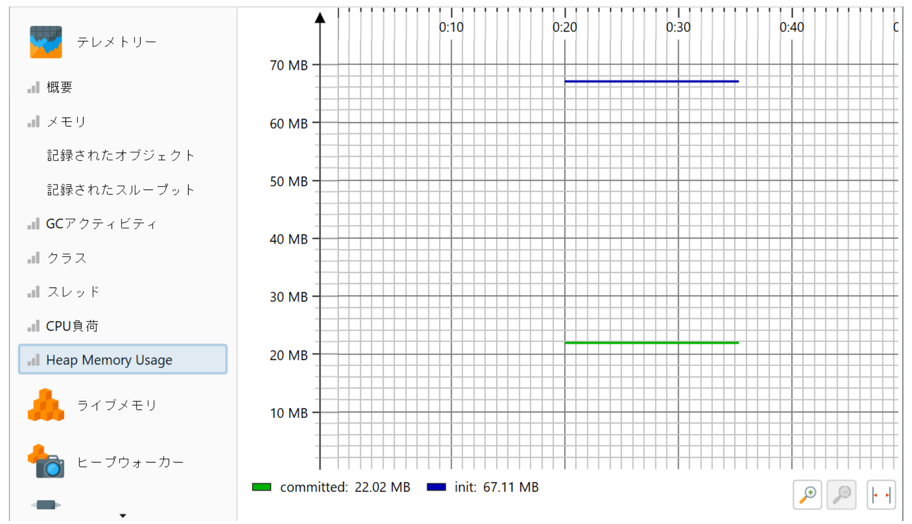



複数のテレメトリーラインを1つのテレメトリーにまとめることができます。そのため、設定は2つの部分に分かれています：テレメトリー自体とテレメトリーラインです。　テレメトリーラインでは、データソースとラインキャプションを編集するだけで、テレメトリーでは単位、スケール、スタッキングを設定できます。 これらはすべて含まれるラインに適用されます。

スタックされたテレメトリーでは、単一のテレメトリーラインが加算され、エリアグラフを表示できます。スケールファクターは、値をサポートされている単位に変換するのに役立ちます。　例えば、データソースがkBを報告する場合、問題はJProfilerに一致する「kB」単位がないことです。スケールファクターを-3に設定すると、値はバイトに変換され、　テレメトリーの単位として「バイト」を選択することで、JProfilerは自動的に適切な集計単位をテレメトリーに表示します。




カスタムテレメトリーは「テレメトリー」セクションの最後に、設定された順序で追加されます。それらを並べ替えるには、概要で目的の位置にドラッグします。




### オーバーヘッドの考慮事項

一見すると、テレメトリーは時間と共にメモリを線形に消費するように見えるかもしれません。しかし、JProfilerは古い値を統合し、テレメトリーごとに消費されるメモリの総量を制限するために徐々に粗粒化します。

テレメトリーのCPUオーバーヘッドは、値が1秒に1回しかポーリングされないという事実によって制限されています。　標準テレメトリーでは、このデータ収集に追加のオーバーヘッドはありません。カスタムテレメトリーでは、オーバーヘッドは基礎となるスクリプトまたはMBeanによって決まります。

# CPU プロファイリング

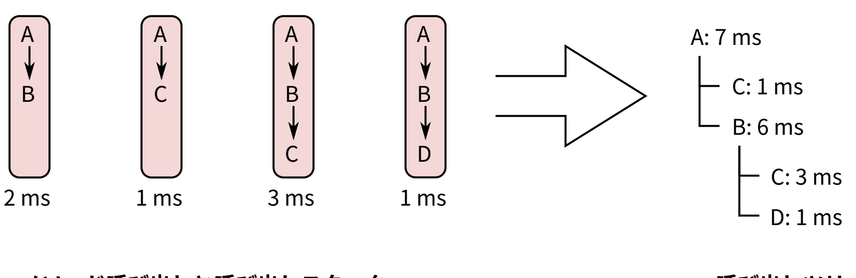
JProfilerがメソッドコールの実行時間をその呼び出しスタックと共に測定する場合、それを「CPU プロファイリング」と呼びます。このデータはさまざまな方法で提示されます。解決しようとしている問題に応じて、どのプレゼンテーションが最も役立つかが決まります。CPUデータはデフォルトでは記録されず、興味深いユースケースをキャプチャするには CPU 記録をオンにする [p. 28] 必要があります。

### 呼び出しツリー

すべてのメソッドコールとその呼び出しスタックを追跡すると、かなりの量のメモリを消費し、すべてのメモリが使い果たされるまで短時間しか維持できません。また、忙しい JVM でのメソッドコールの数を直感的に把握するのは簡単ではありません。通常、その数は非常に多いため、トレースを特定して追跡することは不可能です。

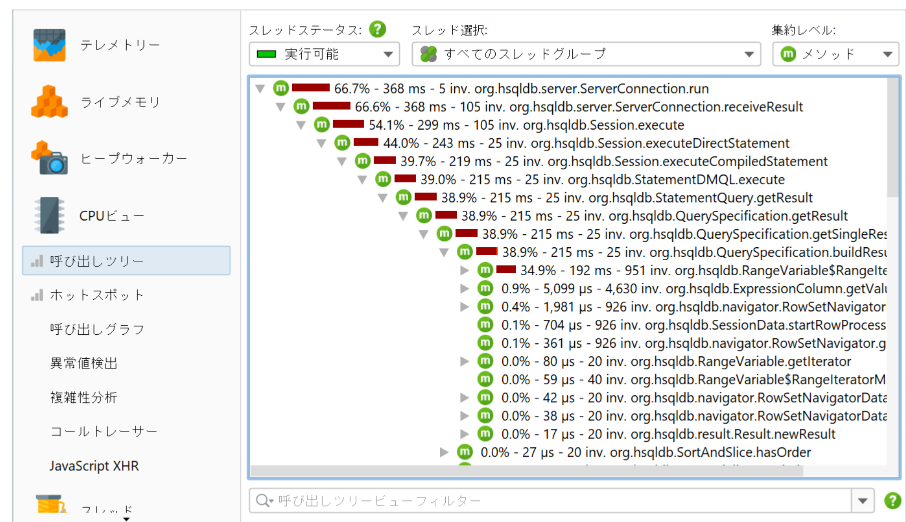
もう一つの側面は、多くのパフォーマンス問題が収集されたデータを集約することで初めて明確になることです。このようにして、特定の時間期間における全体の活動に対するメソッドコールの重要性を判断できます。単一のトレースでは、見ているデータの相対的な重要性を把握することはできません。

これが、JProfilerが観察されたすべての呼び出しスタックの累積ツリーを構築し、観察されたタイミングと呼び出し回数で注釈を付ける理由です。時間的な側面は排除され、合計数のみが保持されます。ツリー内の各ノードは、少なくとも一度は観察された呼び出しスタックを表します。ノードには、その呼び出しスタックで見られたすべての発信コールを表す子がいます。

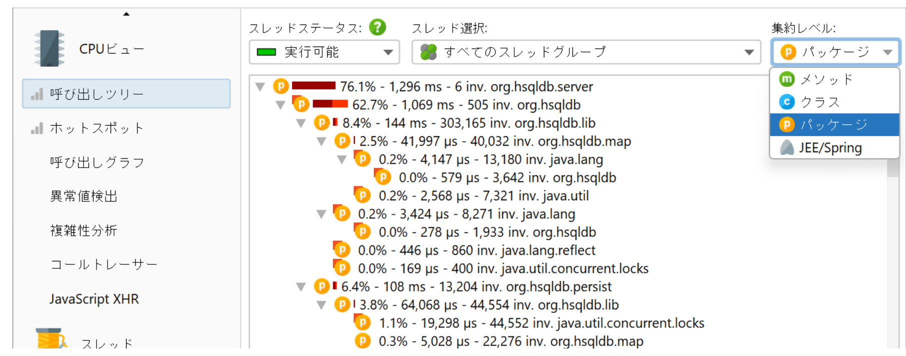



メソッド呼び出しと呼び出しスタック　　呼び出しツリー

呼び出しツリーは「CPUビュー」セクションの最初のビューであり、CPUプロファイリングを開始する際の良い出発点です。メソッドコールを開始点から最も細かい終点まで追跡するトップダウンビューは最も理解しやすいです。JProfilerは子を合計時間でソートするため、ツリーを深さ優先で開いて、パフォーマンスに最も影響を与える部分を分析できます。




すべての測定はメソッドに対して行われますが、JProfilerはクラスまたはパッケージレベルで呼び出しツリーを集約することで、より広い視点を提供します。集約レベルセレクターには「JEE/Springコンポーネント」モードも含まれています。アプリケーションが JEE または Spring を使用している場合、このモードを使用してクラスレベルで JEE および Spring コンポーネントのみを表示できます。URL のような分割ノードはすべての集約レベルで保持されます。

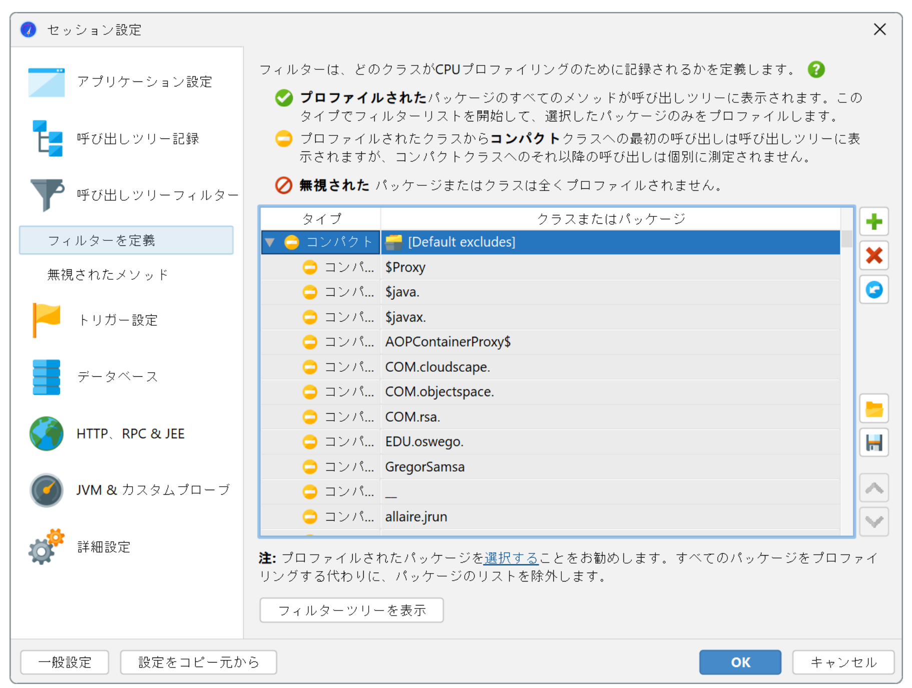



## 呼び出しツリーフィルター

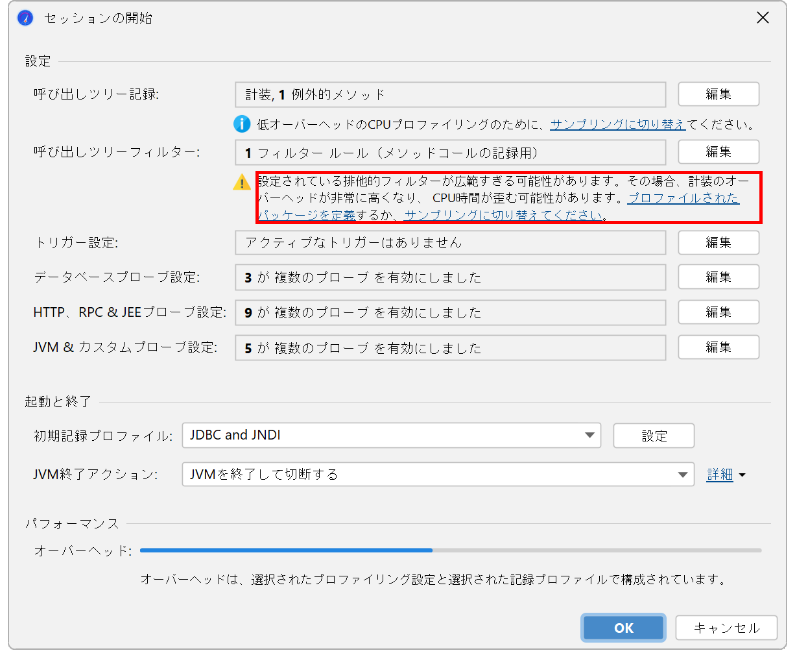
すべてのクラスのメソッドが呼び出しツリーに表示される場合、ツリーは通常管理できないほど深くなります。アプリケーションがフレームワークによって呼び出される場合、呼び出しツリーの上部はフレームワーククラスで構成され、関心がなく、自分のクラスは深く埋もれています。ライブラリへのコールは、その内部構造を示し、数百レベルのメソッドコールが表示される可能性があり、これらを影響を与えることはできません。

この問題の解決策は、呼び出しツリーにフィルターを適用し、一部のクラスのみを記録することです。ポジティブな副作用として、収集するデータが少なくなり、計測するクラスが少なくなるため、オーバーヘッドが削減されます。

デフォルトでは、プロファイリングセッションは一般的に使用されるフレームワークおよびライブラリから除外されたパッケージのリストで構成されています。




もちろん、このリストは不完全なので、削除して自分で関心のあるパッケージを定義する方が良いです。実際、計測[p.67]とデフォルトフィルターの組み合わせは非常に望ましくないため、JProfilerはセッション開始ダイアログで変更を提案します。

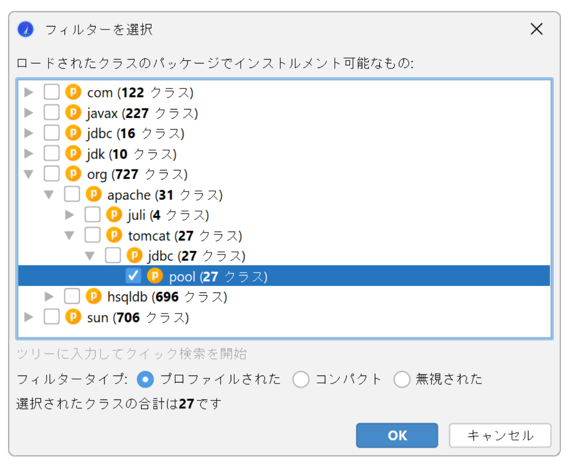



フィルター式は完全修飾クラス名と比較されるため、`com.mycorp.` は `com.mycorp.myapp.Application` のようなすべてのネストされたパッケージ内のクラスに一致します。フィルターには「プロファイルされた」、「コンパクト」、「無視された」という3つのタイプがあります。「プロファイルされた」クラス内のすべてのメソッドが測定されます。これは自分のコードに必要なものです。

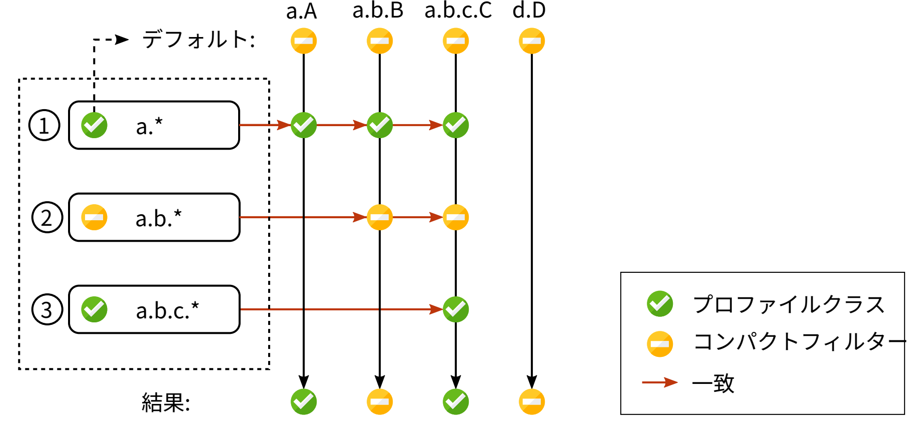
「コンパクト」フィルターに含まれるクラスでは、そのクラスへの最初のコールのみが測定されますが、内部コールは表示されません。「コンパクト」はライブラリ、特にJREに適しています。たとえば、`hashMap.put(a, b)` を呼び出すとき、呼び出しツリーに `HashMap.put()` を表示したいかもしれませんが、それ以上はありません。内部動作は不透明として扱われるべきです。

最後に、「無視された」メソッドはまったくプロファイルされません。オーバーヘッドの考慮から計測が望ましくない場合や、動的コールの間に挿入される内部 Groovy メソッドのように、呼び出しツリーで単に気を散らす場合があります。

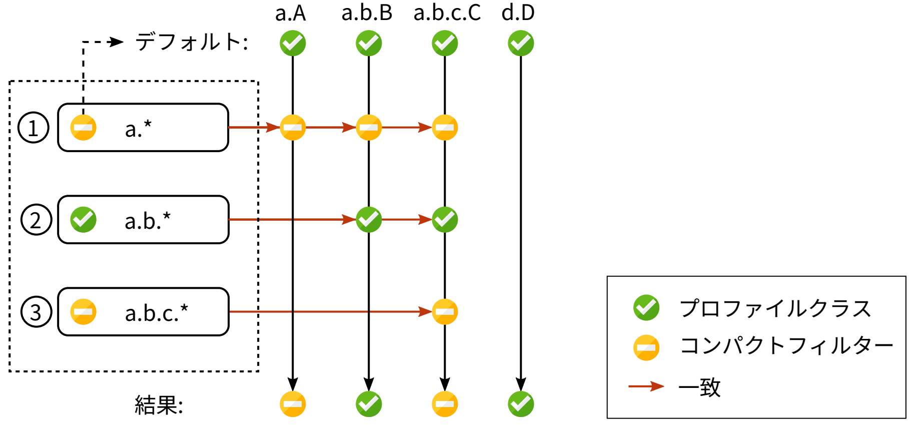
パッケージを手動で入力するのはエラーが発生しやすいため、パッケージブラウザを使用できます。セッションを開始する前に、パッケージブラウザは構成されたクラスパス内のパッケージのみを表示できますが、実際にロードされるすべてのクラスをカバーすることはできません。実行時には、パッケージブラウザはすべてのロードされたクラスを表示します。




構成されたフィルターのリストは、各クラスに対して上から下に評価されます。各段階で、一致がある場合、現在のフィルタータイプが変更されることがあります。フィルターリストを開始するフィルターの種類が重要です。「プロファイルされた」フィルターで開始すると、クラスの初期フィルタータイプは「コンパクト」になり、明示的な一致のみがプロファイルされます。



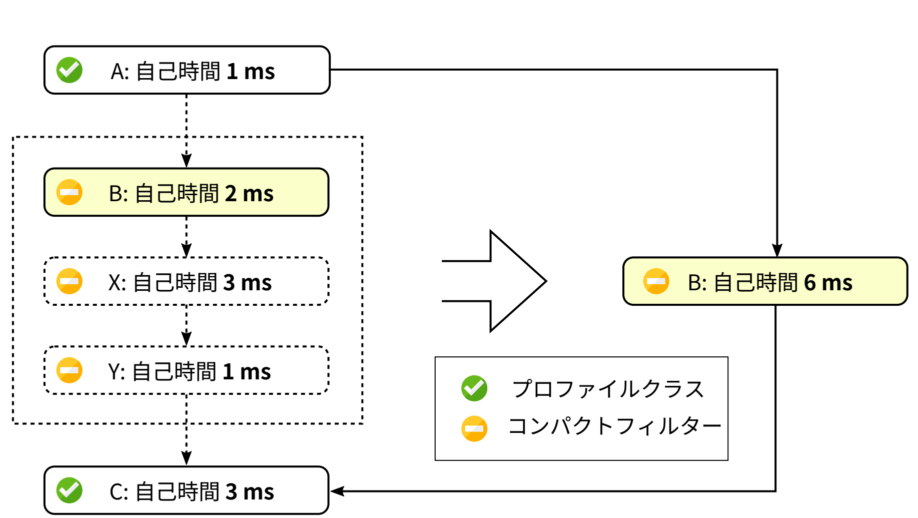

「コンパクト」フィルターで開始すると、クラスの初期フィルタータイプは「プロファイルされた」になります。この場合、明示的に除外されたクラスを除いて、すべてのクラスがプロファイルされます。




### 呼び出しツリーの時間

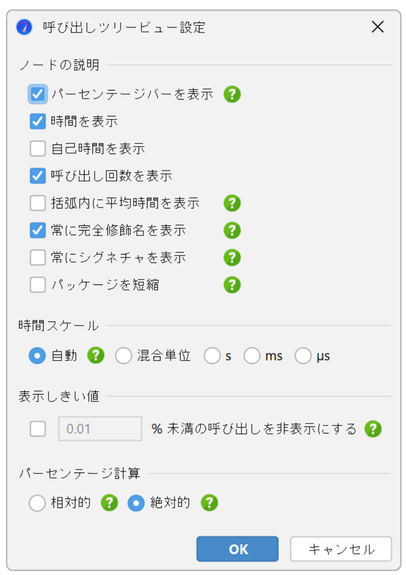
呼び出しツリーを正しく解釈するには、呼び出しツリーノードに表示される数値を理解することが重要です。任意のノードにとって興味深い時間は2つあります。合計時間と自己時間です。自己時間は、ノードの合計時間からネストされたノードの合計時間を引いたものです。

通常、自己時間は小さいですが、コンパクトフィルターされたクラスではそうではありません。ほとんどの場合、コンパクトフィルターされたクラスはリーフノードであり、合計時間は自己時間と等しくなります。時々、コンパクトフィルターされたクラスがプロファイルされたクラスを呼び出すことがあります。たとえば、コールバックを介して、または呼び出しツリーのエントリーポイントであるためです。この場合、プロファイルされていないメソッドが時間を消費しましたが、呼び出しツリーには表示されません。その時間は呼び出しツリーの最初の利用可能な祖先ノードにバブルアップし、コンパクトフィルターされたクラスの自己時間に寄与します。



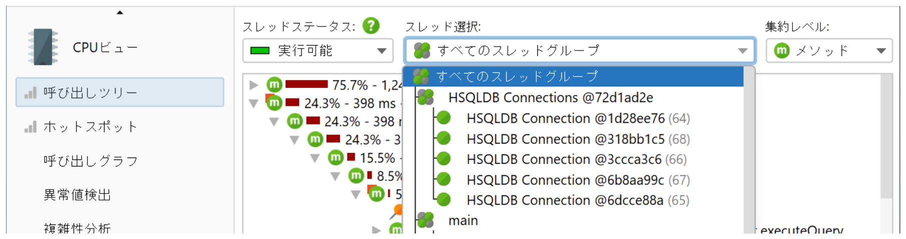

呼び出しツリーのパーセンテージバーは合計時間を示しますが、自己時間の部分は異なる色で表示されます。メソッドは、同じレベルで2つのメソッドがオーバーロードされていない限り、シグネチャなしで表示されます。ビュー設定ダイアログで呼び出しツリーノードの表示をカスタマイズするさまざまな方法があります。たとえば、自己時間や平均時間をテキストとして表示したり、メ

ソッドシグネチャを常に表示したり、使用する時間スケールを変更したりできます。また、パーセンテージ計算は、親時間に基づいて行うこともできます。




### スレッドステータス

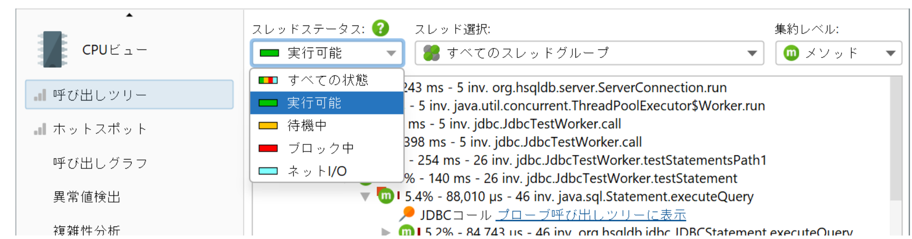
呼び出しツリーの上部には、表示されるプロファイリングデータのタイプと範囲を変更するいくつかのビューパラメータがあります。デフォルトでは、すべてのスレッドが累積されます。JProfiler はスレッドごとにCPUデータを維持し、単一のスレッドまたはスレッドグループを表示できます。




常に、各スレッドには関連付けられたスレッドステータスがあります。スレッドがバイトコード命令を処理する準備ができているか、現在 CPU コアで実行している場合、スレッドステータスは「Runnable」と呼ばれます。このスレッド状態は、パフォーマンスのボトルネックを探す際に興味深いものであるため、デフォルトで選択されています。

代わりに、スレッドがモニターを待機している場合、たとえば `Object.wait()` または `Thread.sleep()` を呼び出している場合、スレッド状態は「Waiting」と呼ばれます。`synchronized` コードブロックの境界でモニターを取得しようとしてブロックされているスレッドは、「Blocking」状態にあります。

最後に、JProfiler はスレッドがネットワークデータを待機している時間を追跡する合成「Net I/O」状態を追加します。これはサーバーやデータベースドライバーを分析する際に重要です。なぜなら、その時間はパフォーマンス分析、たとえば遅いSQLクエリの調査に関連する可能性があるからです。

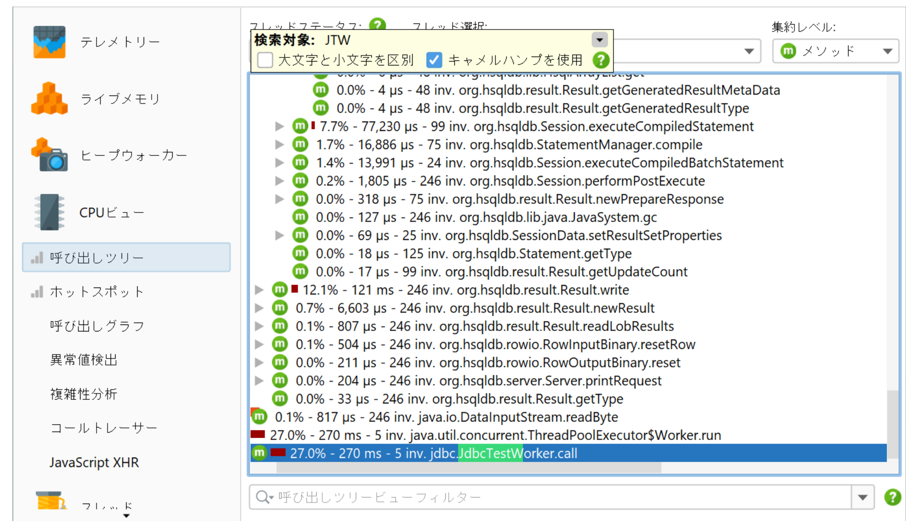



実時間を知りたい場合は、スレッドステータス「すべての状態」を選択し、単一のスレッドも選択する必要があります。その場合にのみ、コード内で `System.currentTimeMillis()` を呼び出して計算した時間と比較できます。

選択したメソッドを別のスレッドステータスにシフトしたい場合は、メソッドトリガーと「スレッドステータスを上書きする」トリガーアクションを使用するか、埋め込み[p. 168]またはインジェクション[p. 163]プローブ API の `ThreadStatus` クラスを使用して行うことができます。

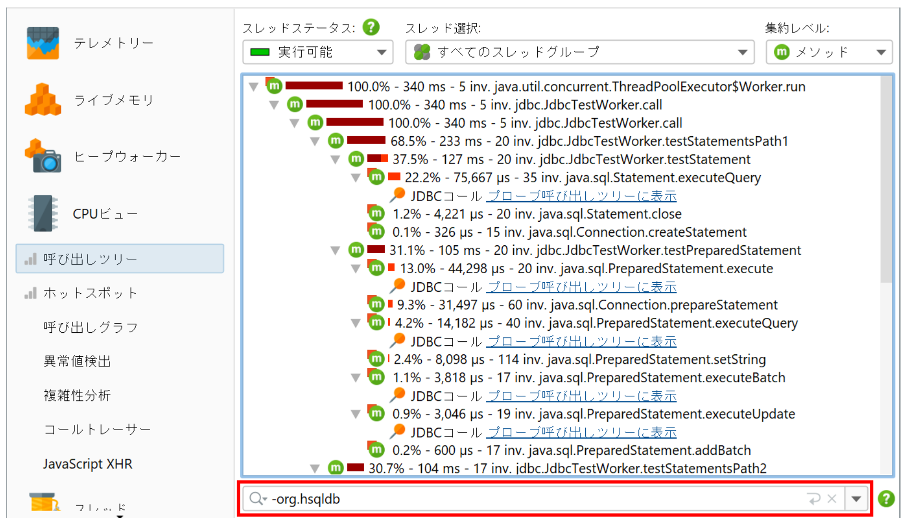
### 呼び出しツリーでのノードの検索

呼び出しツリーでテキストを検索する方法は2つあります。まず、ビュー->検索 をメニューから呼び出すか、呼び出しツリーに直接入力を開始することでアクティブになるクイック検索オプションがあります。マッチがハイライトされ、`PageDown` を押すと検索オプションが利用可能になります。`ArrowUp` および `ArrowDown` キーを使用して、異なるマッチを循環できます。



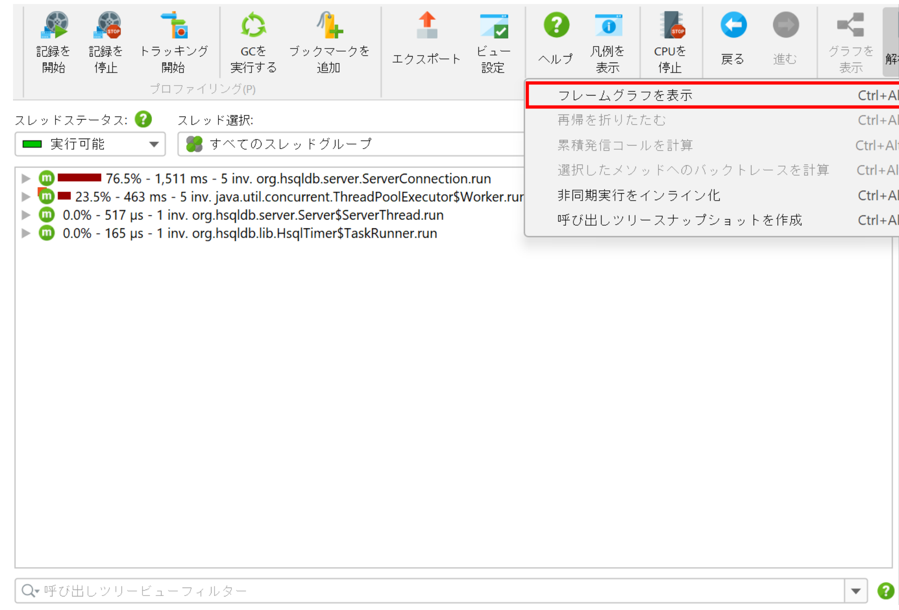

メソッド、クラス、またはパッケージを検索するもう一つの方法は、呼び出しツリーの下部にあるビューフィルターを使用することです。ここでは、カンマ区切りのフィルター式のリストを入力できます。"-" で始まるフィルター式は無視されたフィルターのようなものです。"!" で始まる式はコンパクトフィルターのようなものです。他のすべての式はプロファイルされたフィルターのようなものです。フィルター設定と同様に、初期フィルタータイプはクラスがデフォルトで含まれるか除外されるかを決定します。

ビュー設定テキストフィールドの左側にあるアイコンをクリックすると、ビュー フィルター オプションが表示されます。デフォルトでは、マッチングモードは「含む」ですが、特定のパッケージを検索する場合、「開始」である方が適切かもしれません。

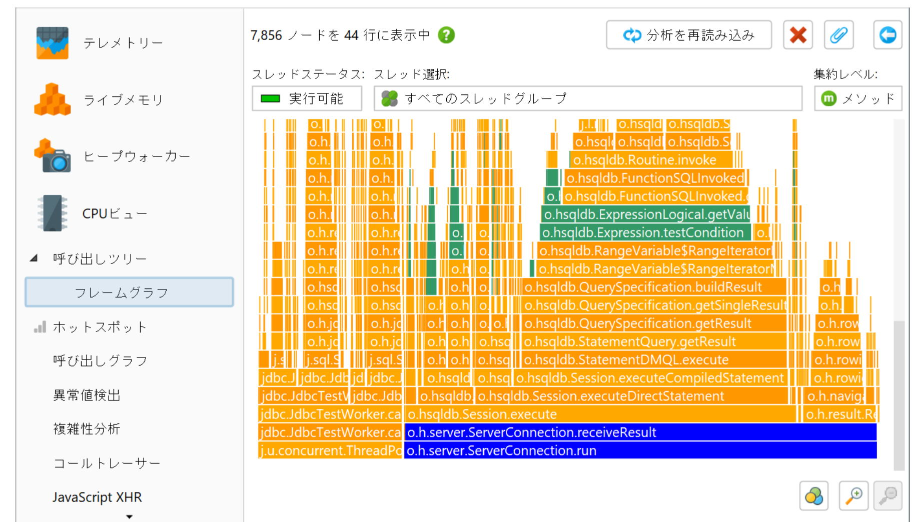



## フレームグラフ

呼び出しツリーをフレームグラフとして表示する別の方法があります。関連する呼び出しツリー分析 [p. 190] を呼び出すことで、呼び出しツリー全体またはその一部をフレームグラフとして表示できます。




フレームグラフは、呼び出しツリーの全内容を1つの画像で表示します。コールはフレームグラフの下部から始まり、上部に向かって伝播します。各ノードの子は、その直上の行に配置されます。子ノードはアルファベット順にソートされ、親ノードの中心に配置されます。各ノードで費やされた自己時間のために、「炎」は上に向かって徐々に狭くなります。ノードに関する詳細情報は、ツールチップに表示され、テキストをマークしてクリップボードにコピーできます。




マウスカーソルの近くのツールチップが分析を妨げる場合は、右上のボタンでロックし、ツールチップの上部にあるグリッパーで便利な場所に移動できます。同じボタンまたはフレームグラフをダブルクリックすると、ツールチップが閉じます。

フレームグラフは非常に高い情報密度を持っているため、選択したノードとその子孫ノードの階層に焦点を当てることで表示されるコンテンツを絞り込む必要があるかもしれません。興味のある領域をズームインすることもできますが、コンテキストメニューを使用して新しいルートノードを設定することもできます。連続してルートを変更する場合、ルートの履歴で戻ることができます。

フレームグラフを分析する別の方法は、クラス名、パッケージ名、または任意の検索語に基づいてカラー化を追加することです。カラー化はコンテキストメニューから追加でき、カラー化ダイアログで管理できます。各ノードには最初に一致するカラー化が使用されます。カラー化はプロファイリングセッション間で永続化され、すべてのセッションとスナップショットに対してグローバルに使用されます。

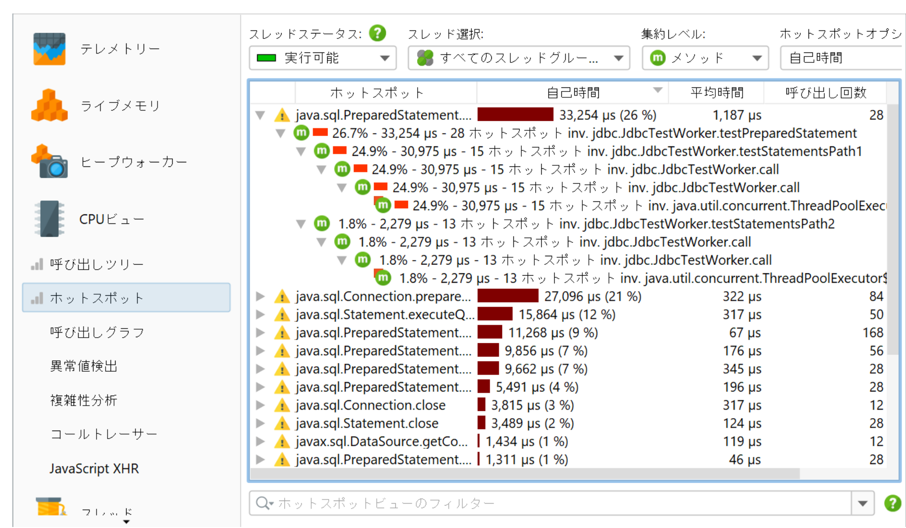
カラー化に加えて、クイック検索機能を使用して興味のあるノードを見つけることができます。カーソルキーを使用してマッチ結果を循環し、現在ハイライトされているマッチのツールチップが表示されます。

### ホットスポット

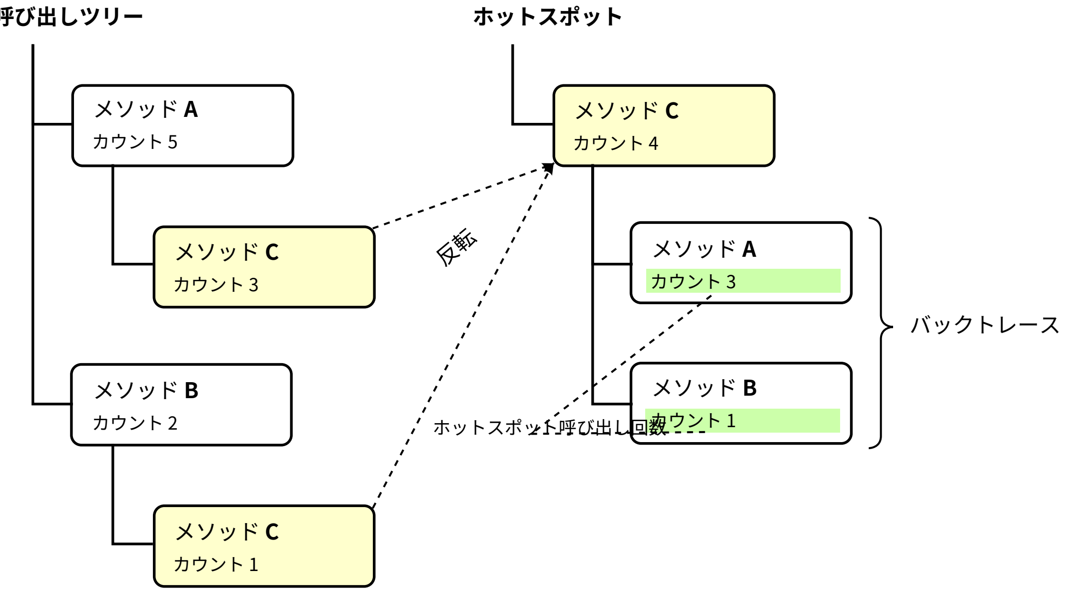
アプリケーションが遅すぎる場合、最も時間を要するメソッドを見つけたいと思うでしょう。呼び出しツリーを使用すると、これらのメソッドを直接見つけることができる場合がありますが、多くの場合、呼び出しツリーが広く、リーフノードの数が非常に多いため、うまくいきません。

その場合、呼び出しツリーの逆が必要です。すべてのメソッドをその合計自己時間でソートしたリストで、異なる呼び出しスタックから累積され、メソッドがどのように呼び出されたかを示すバッ

クトレースがあります。ホットスポットツリーでは、リーフはエントリーポイントであり、アプリケーションの`main`メソッドやスレッドの`run`メソッドのようなものです。ホットスポットツリーの最も深いノードから、コールはトップレベルノードに向かって上に伝播します。

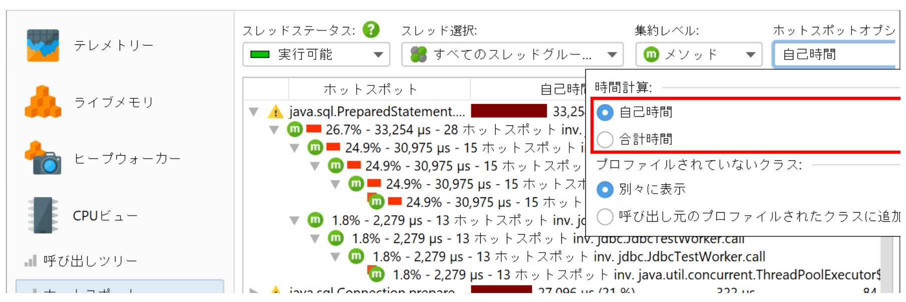



バックトレースの呼び出し回数と実行時間はメソッドノードを指しているのではなく、このパスに沿ってトップレベルホットスポットノードが呼び出された回数を指しています。これは理解することが重要です。表面的には、ノードの情報がそのノードへの呼び出しを定量化することを期待するかもしれません。しかし、ホットスポットツリーでは、その情報はトップレベルノードへのノードの**貢献**を示しています。この逆の呼び出しスタックに沿って、トップレベルホットスポットが`n`回呼び出され、合計`t`秒の期間がありました。




デフォルトでは、ホットスポットは自己時間から計算されます。合計時間から計算することもできます。これはパフォーマンスのボトルネックを分析するのにはあまり役立ちませんが、すべてのメソッドのリストを見たい場合には興味深いかもしれません。ホットスポットビューはオーバーヘッ

ドを減らすために最大数のメソッドのみを表示するため、探しているメソッドが表示されない場合があります。その場合、パッケージまたはクラスをフィルターするために下部のビューフィルターを使用してください。呼び出しツリーとは異なり、ホットスポットビューのフィルターはトップレベルノードのみをフィルターします。ホットスポットビューのカットオフはグローバルに適用されるのではなく、表示されるクラスに対して適用されるため、フィルターを適用した後に新しいノードが表示されることがあります。

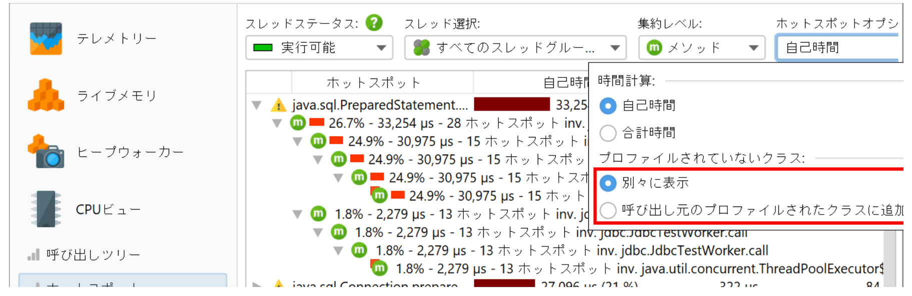



## ホットスポットとフィルター

ホットスポットの概念は絶対的なものではなく、呼び出しツリーフィルターに依存します。呼び出しツリーフィルターがまったくない場合、最大のホットスポットはおそらく常にJREのコアクラスのメソッド、たとえば文字列操作、I/O ルーチン、コレクション操作などになります。このようなホットスポットはあまり役に立ちません。なぜなら、これらのメソッドの呼び出しを直接制御することはほとんどなく、それらを高速化する方法もないからです。

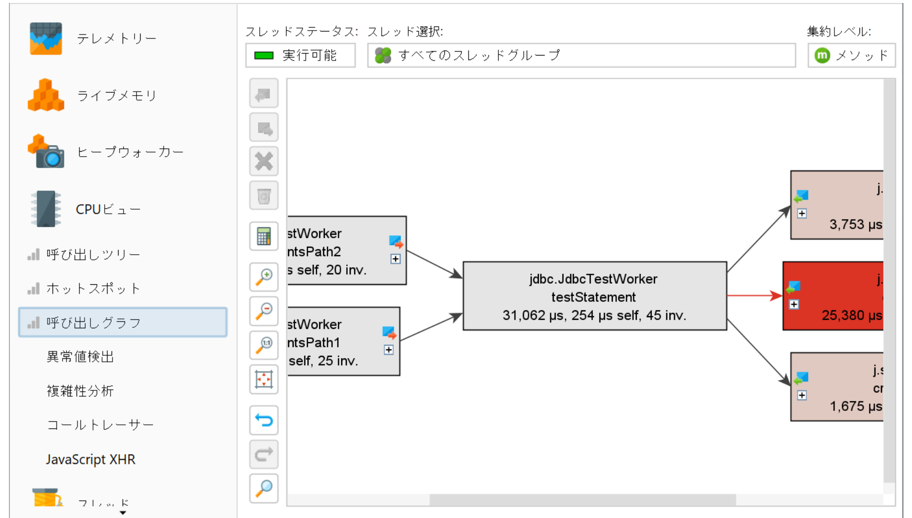
あなたにとって役立つホットスポットは、自分のクラス内のメソッドか、直接呼び出すライブラリクラス内のメソッドである必要があります。呼び出しツリーフィルターの観点から、自分のクラスは「プロファイルされた」フィルターにあり、ライブラリクラスは「コンパクト」フィルターにあります。

パフォーマンスの問題を解決する際には、ライブラリレイヤーを排除し、自分のクラスのみを見たいかもしれません。ホットスポットオプションポップアップで呼び出しプロファイルされたクラスに追加 ラジオボタンを選択することで、その視点にすばやく切り替えることができます。




## 呼び出しグラフ

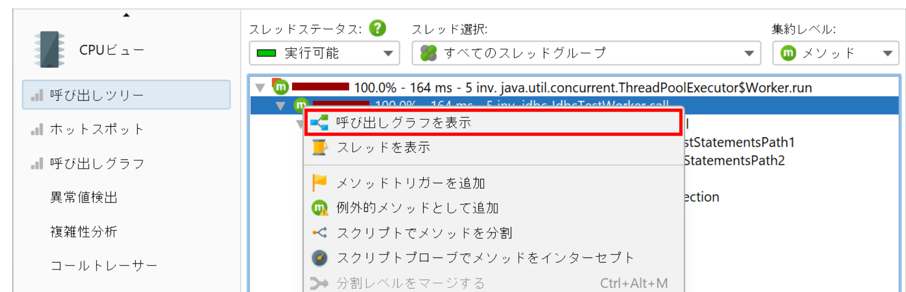
呼び出しツリーとホットスポットビューの両方で、特に再帰的に呼び出される場合、各ノードが複数回出現することがあります。ある状況では、各メソッドが一度だけ出現し、すべての入出力コールが表示されるメソッド中心の統計に興味があるかもしれません。このようなビューはグラフとして表示するのが最適であり、JProfiler ではこれを呼び出しグラフと呼びます。




グラフの欠点の一つは、ツリーよりも視覚的な密度が低いことです。このため、JProfilerはデフォルトでパッケージ名を省略し、合計時間の 1% 未満の発信コールをデフォルトで非表示にします。ノードに発信拡張アイコンがある限り、再度クリックしてすべてのコールを表示できます。ビュー設定では、このしきい値を設定し、パッケージの省略をオフにすることができます。

呼び出しグラフを展開すると、特に複数回バックトラックする場合、非常に混乱する可能性があります。元の状態を復元するには、元に戻す機能を使用してください。呼び出しツリーと同様に、呼び出しグラフもクイック検索を提供します。グラフに入力することで検索を開始できます。

グラフとツリービューにはそれぞれ利点と欠点があるため、ビュータイプを切り替えたい場合があります。インタラクティブセッションでは、呼び出しツリーとホットスポットビューはライブデータを表示し、定期的に更新されます。ただし、呼び出しグラフはリクエストに応じて計算され、ノードを展開しても変更されません。呼び出しツリーの 呼び出しグラフで表示 アクションは、新しい呼び出しグラフを計算し、選択したメソッドを表示します。




グラフから呼び出しツリーに切り替えることはできません。なぜなら、データは通常、後の時点で比較できなくなるからです。ただし、呼び出しグラフは、ビュー->分析アクションを使用して、選択したノードごとに累積された発信コールとバックトレースのツリーを表示できる呼び出しツリー分析を提供します。

## 基本を超えて

呼び出しツリー、ホットスポットビュー、呼び出しグラフのセットには、多くの高度な機能があり、別の章 [p. 172]で詳細に説明されています。また、別々に [p. 196]紹介される他の高度な CPU ビューもあります。

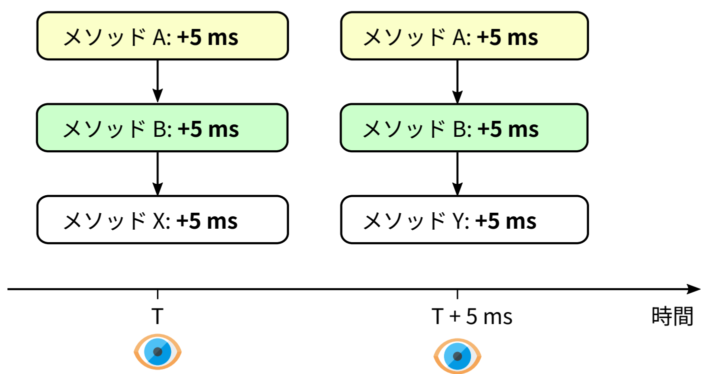
## メソッドコール記録

メソッドコールを記録することはプロファイラーにとって最も難しいタスクの一つです。なぜなら、結果は正確で完全であり、測定されたデータから導き出される結論が誤らないようにするために、非常に小さなオーバーヘッドで行われる必要があるからです。残念ながら、すべてのアプリケーションタイプに対してこれらすべての要件を満たす単一の測定方法は存在しません。このため、JProfilerではどの方法を使用するかを決定する必要があります。

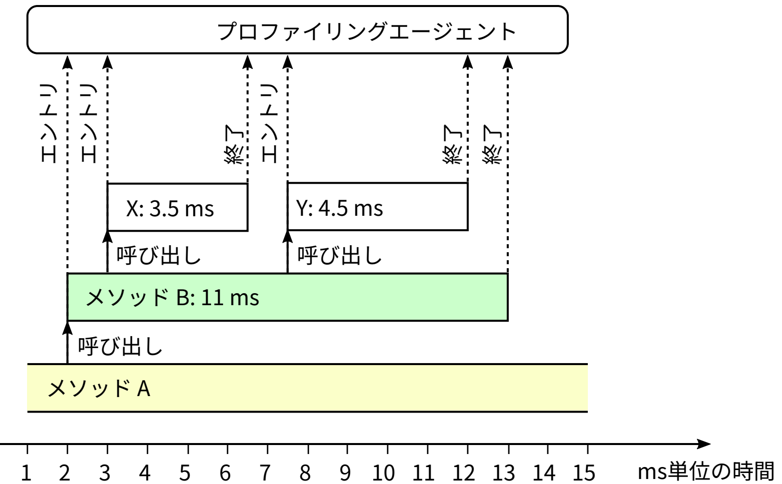
### サンプリング対インストゥルメンテーション

メソッドコールの測定は、「サンプリング」と「インストゥルメンテーション」と呼ばれる2つの基本的に異なる技術で行うことができます。それぞれに利点と欠点があります。サンプリングでは、スレッドの現在の呼び出しスタックを定期的に検査します。インストゥルメンテーションでは、選択されたクラスのバイトコードを修正してメソッドのエントリとエグジットを追跡します。インストゥルメンテーションはすべての呼び出しを測定し、すべてのメソッドの呼び出し回数を生成できます。

サンプリングデータを処理する際、フルサンプリング期間（通常5ms）はサンプリングされた呼び出しスタックに帰属します。多くのサンプルがあると、統計的に正しい図が現れます。サンプリングの利点は、頻度が低いため非常に低いオーバーヘッドを持つことです。バイトコードを修正する必要がなく、サンプリング期間はメソッドコールの典型的な持続時間よりもはるかに長いです。欠点は、メソッドの呼び出し回数を決定できないことです。また、数回しか呼び出されない短時間実行のメソッドはまったく表示されないかもしれません。これはパフォーマンスのボトルネックを探している場合には問題ありませんが、コードの詳細な実行時特性を理解しようとしている場合には不便です。




一方、インストゥルメンテーションは、多くの短時間実行のメソッドがインストゥルメントされると大きなオーバーヘッドを引き起こす可能性があります。このインストゥルメンテーションは、時間測定の固有のオーバーヘッドのために、パフォーマンスホットスポットの相対的重要性を歪めますが、ホットスポットコンパイラによってインライン化されるはずの多くのメソッドが、今や別々のメソッドコールとして残らなければならないためでもあります。長時間かかるメソッドコールの場合、オーバーヘッドは無視できます。主に高レベルの操作を行う良いクラスセットを見つけることができれば、インストゥルメンテーションは非常に低いオーバーヘッドを追加し、サンプリングよりも好ましい場合があります。JProfilerのオーバーヘッドホットスポット検出は、いくつかの実行後に状況を改善することもできます。また、呼び出し回数は、何が起こっているのかを理解するのに非常に重要な情報であることが多いです。



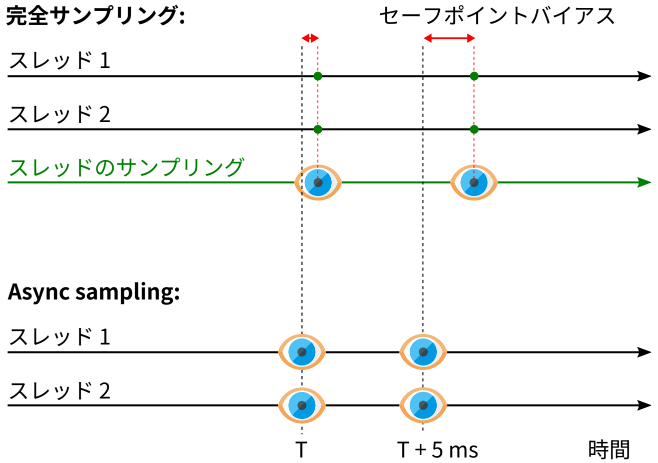

**フルサンプリング対非同期サンプリング**

JProfilerはサンプリングのために2つの異なる技術的解決策を提供します。「フルサンプリング」は、JVM内のすべてのスレッドを定期的に一時停止し、そのスタックトレースを検査する別のスレッドで行われます。ただし、JVMは特定の「セーフポイント」でのみスレッドを一時停止するため、バイアスが導入されます。高度にマルチスレッド化されたCPUバウンドコードを持っている場合、プロファイルされたホットスポットの分布が歪む可能性があります。一方、コードが重要なI/Oを実行する場合、このバイアスは一般的に問題になりません。

高度にCPUバウンドされたコードの正確な数値を取得するのを助けるために、JProfilerは非同期サンプリングも提供します。非同期サンプリングでは、プロファイリングシグナルハンドラが実行中のスレッド自体で呼び出されます。プロファイリングエージェントはネイティブスタックを検査し、Javaスタックフレームを抽出します。主な利点は、このサンプリング方法ではセーフポイントバイアスがなく、高度にマルチスレッド化されたCPUバウンドアプリケーションのオーバーヘッドが低いことです。ただし、CPUビューでは「Running」スレッド状態のみが観察でき、「Waiting」、「Blocking」または「Net I/O」スレッド状態はこの方法では測定できません。プローブデータは常にバイトコードインストゥルメンテーションで収集されるため、JDBCや類似のデータに対してはすべてのスレッド状態を取得できます。

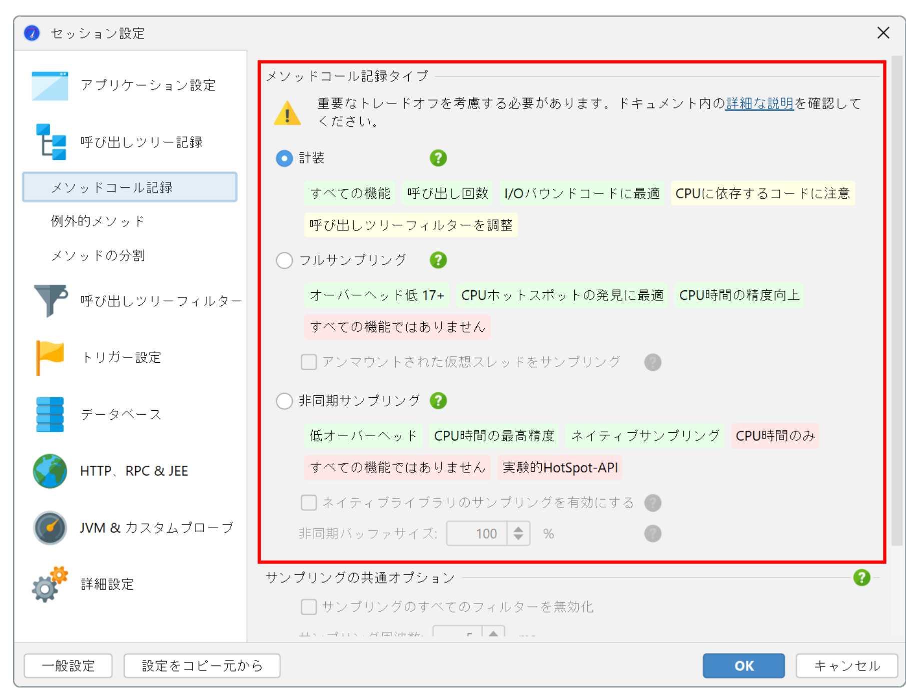
非同期サンプリングは、呼び出しスタックの終わりだけが利用可能な切り詰められたトレースに悩まされます。このため、非同期サンプリングでは呼び出しツリーはホットスポットビューほど有用ではないことがよくあります。非同期サンプリングはLinuxとmacOSでのみサポートされています。

Java 17以降、JProfilerはHotspot JVMでサンプリングにグローバルセーフポイントを使用せず、ほぼゼロオーバーヘッドでフルサンプリングを実行できます。非同期サンプリングと比較して、単一のスレッドに対してはまだある種のセーフポイントバイアスを導入しますが、JVM内のすべてのスレッドに対するグローバルセーフポイントのオーバーヘッドはもはやありません。非同期サンプリングの欠点を考慮すると、Java 17+ではフルサンプリングを使用することが推奨されます。




### メソッドコール記録タイプの選択

プロファイリングに使用するメソッドコール記録タイプは重要な決定であり、すべての状況に対して正しい選択はありませんので、情報に基づいた決定を下す必要があります。新しいセッションを作成するとき、セッション開始ダイアログで使用するメソッドコール記録タイプを尋ねられます。その後いつでも、セッション設定ダイアログでメソッドコール記録タイプを変更できます。




簡単なガイドとして、アプリケーションがスペクトラムの反対側にある2つの明確なカテゴリのいずれかに該当するかどうかをテストする次の質問を考慮してください：

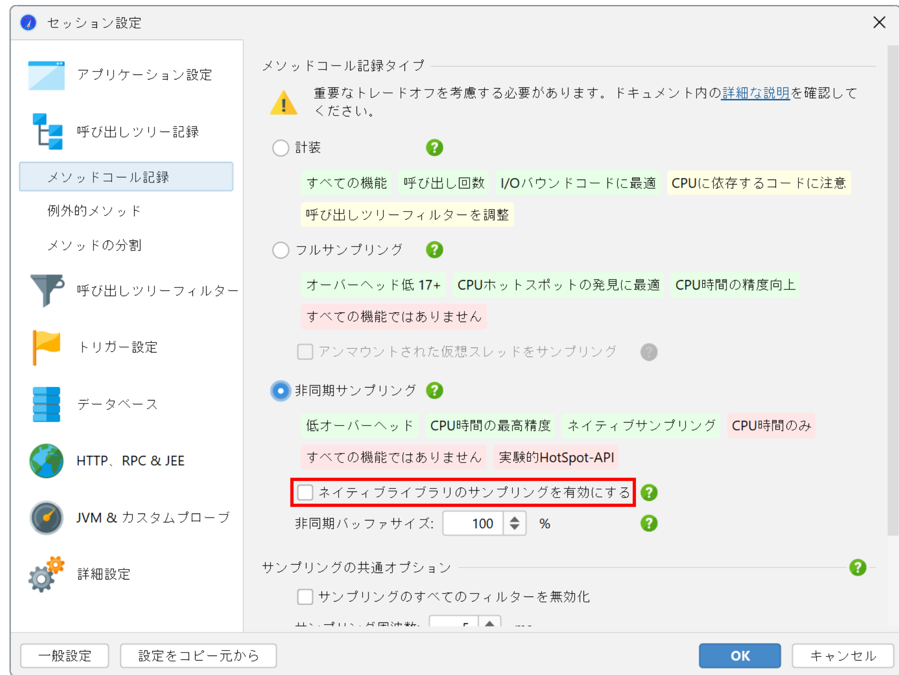
- **プロファイルされたアプリケーションはI/Oバウンドですか？**
  これは、多くのWebアプリケーションがRESTサービスやJDBCデータベースコールを待機している場合に当てはまります。その場合、インストゥルメンテーションは、呼び出しツリーフィルタを慎重に選択して自分のコードのみを含める条件下で、最良のオプションとなります。

- **プロファイルされたアプリケーションは高度にマルチスレッド化され、CPUバウンドですか？**
  例えば、これはコンパイラ、画像処理アプリケーション、または負荷テストを実行しているWebサーバーの場合に当てはまるかもしれません。LinuxまたはmacOSでプロファイリングしている場合、この場合、最も正確なCPU時間を得るために非同期サンプリングを選択する必要があります。

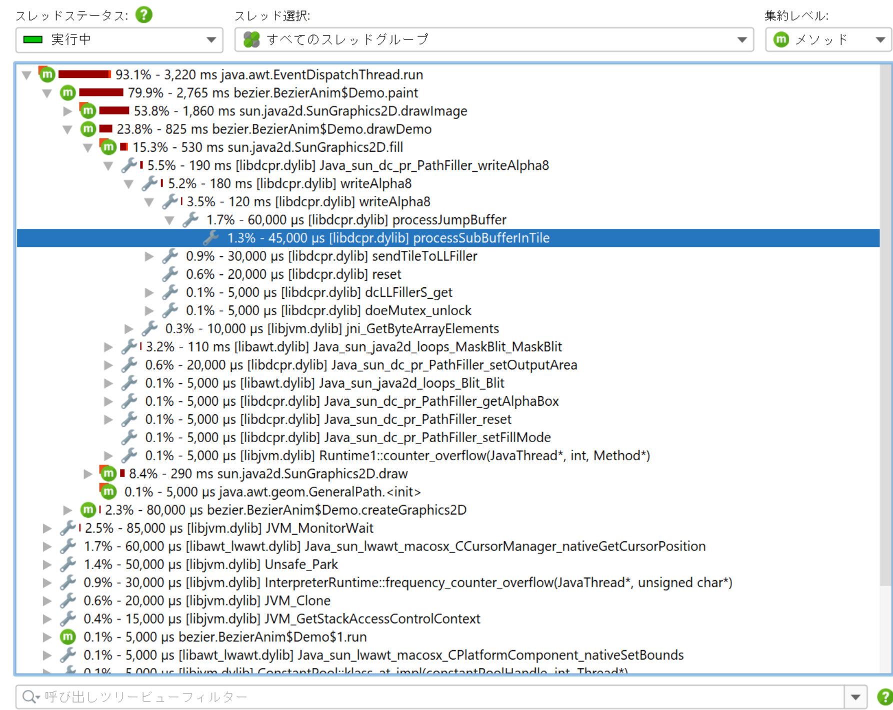
それ以外の場合、「フルサンプリング」が一般的に最も適したオプションであり、新しいセッションのデフォルトとして提案されます。

### ネイティブサンプリング

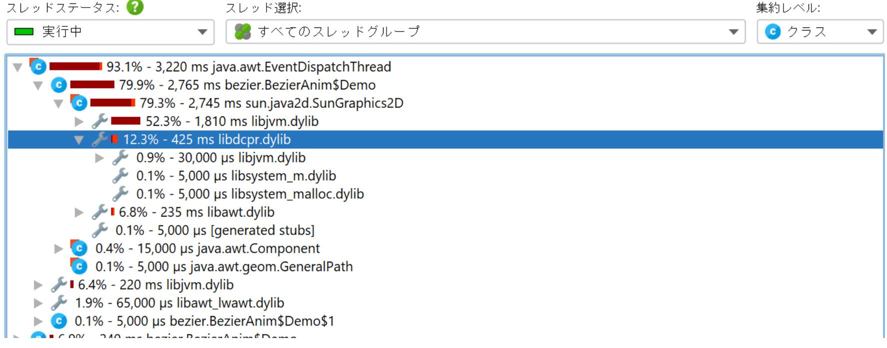
非同期サンプリングはネイティブスタックにアクセスできるため、ネイティブサンプリングも実行できます。デフォルトでは、ネイティブサンプリングは有効になっていません。なぜなら、呼び出しツリーに多くのノードを導入し、ホットスポット計算の焦点をネイティブコードにシフトさせるからです。ネイティブコードにパフォーマンスの問題がある場合、非同期サンプリングを選択し、セッション設定でネイティブサンプリングを有効にすることができます。




JProfilerは、各ネイティブスタックフレームに属するライブラリのパスを解決します。呼び出しツリーのネイティブメソッドノードでは、JProfilerはネイティブライブラリのファイル名を角括弧で始めに表示します。




集約レベルに関しては、ネイティブライブラリはクラスのように機能するため、「クラス」集約レベルでは、同じネイティブライブラリ内のすべての後続の呼び出しが単一のノードに集約されます。「パッケージ」集約レベルは、ネイティブライブラリに関係なく、すべての後続のネイティブメソッドコールを単一のノードに集約します。




選択したネイティブライブラリを排除するには、そのネイティブライブラリからノードを削除[p.181]し、クラス全体を削除することを選択できます。

# メモリプロファイリング

ヒープ上のオブジェクトに関する情報を取得する方法は2つあります。1つは、プロファイリングエージェントが各オブジェクトの割り当てとガベージコレクションを追跡する方法です。JProfilerでは、これを「割り当て記録」と呼びます。これにより、オブジェクトがどこで割り当てられたかを知ることができ、一時オブジェクトに関する統計を作成することもできます。もう1つは、JVMのプロファイリングインターフェースが「ヒープスナップショット」を取得し、すべてのライブオブジェクトとその参照を調査する方法です。この情報は、オブジェクトがなぜガベージコレクションされないのかを理解するために必要です。

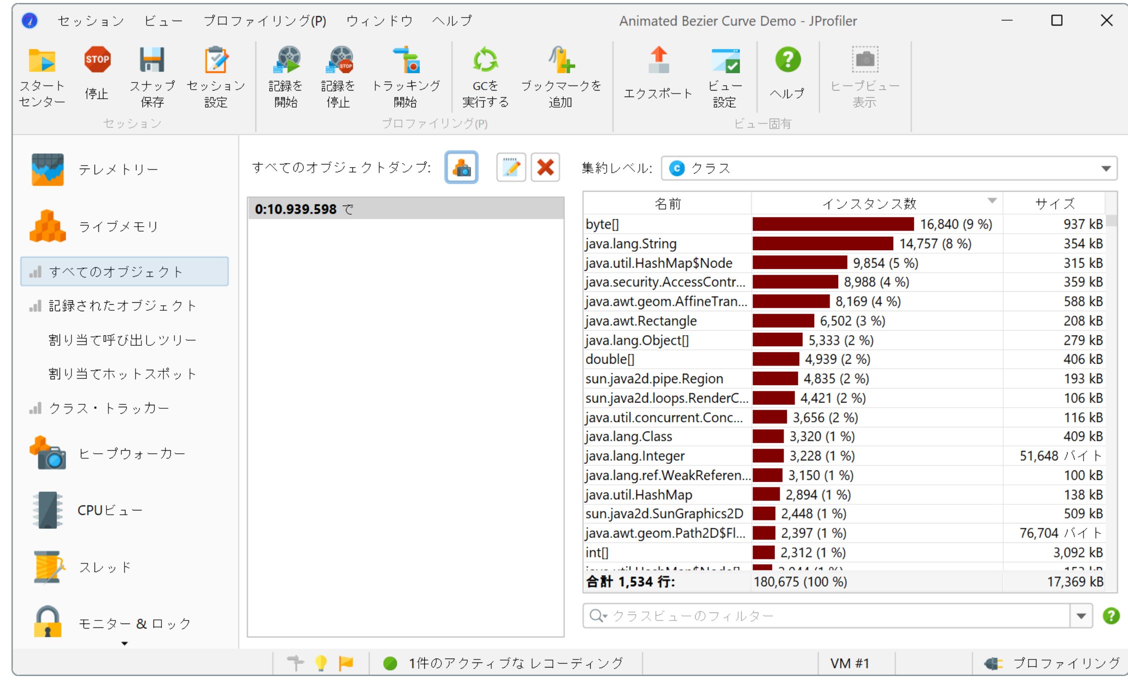
割り当て記録とヒープスナップショットの両方は高価な操作です。割り当て記録は、`java.lang.Object` コンストラクタを計測し、ガベージコレクタがプロファイリングインターフェースに継続的に報告しなければならないため、ランタイム特性に大きな影響を与えます。これが、デフォルトで割り当てが記録されない理由であり、記録を開始および停止[p. 28]する必要があります。ヒープスナップショットを取得するのは一度限りの操作です。しかし、JVMを数秒間停止させる可能性があり、取得したデータの分析には、ヒープのサイズに応じて比較的長い時間がかかることがあります。

JProfilerはメモリ分析を2つのビューセクションに分割します。「ライブメモリ」セクションは定期的に更新できるデータを表示し、「ヒープウォーカー」セクションは静的なヒープスナップショットを表示します。割り当て記録は「ライブメモリ」セクションで制御されますが、記録されたデータはヒープウォーカーによっても表示されます。

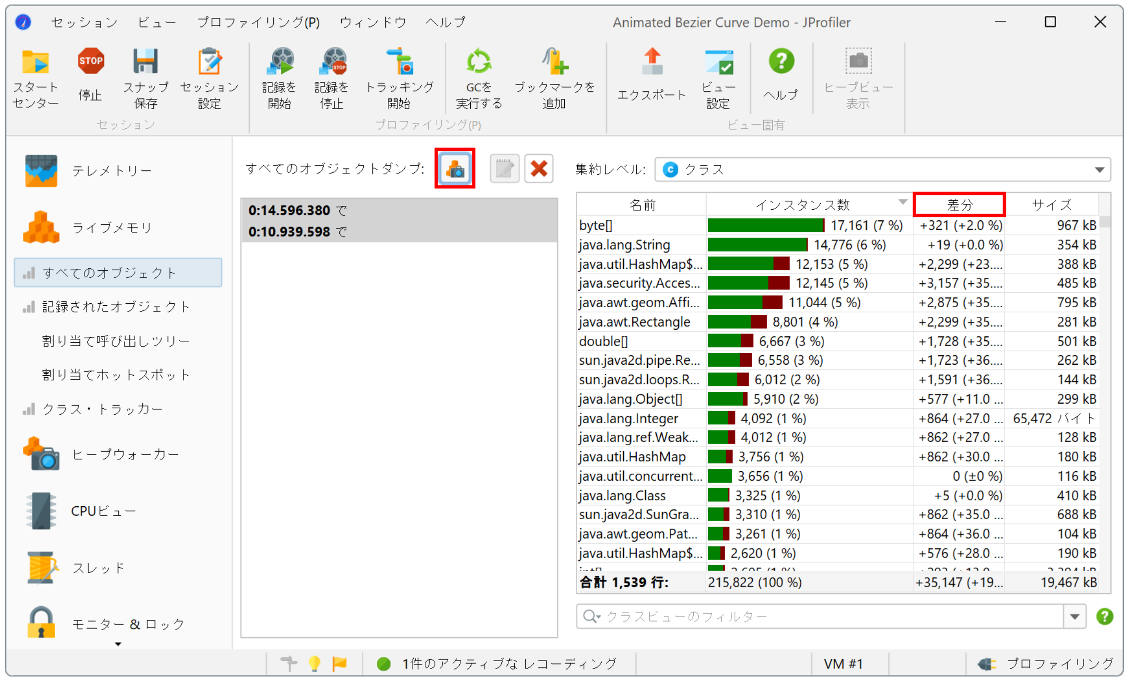
メモリプロファイリングで解決できる最も一般的な3つの問題は、メモリリークの発見[p. 216]、メモリ消費の削減、一時オブジェクトの作成の削減です。最初の2つの問題については、主にヒープウォーカーを使用し、主にJVM内の最大のオブジェクトを保持しているのが誰であるか、そしてそれらがどこで作成されたかを調べます。最後の問題については、ガベージコレクションされたオブジェクトを含むライブビューにのみ依存することができます。

### インスタンス数の追跡

ヒープ上にどのようなオブジェクトがあるかを把握するために、「すべてのオブジェクト」ビューはすべてのクラスとそのインスタンス数のヒストグラムを表示します。このビューに表示されるデータは、割り当て記録ではなく、インスタンス数のみを計算するミニヒープスナップショットを実行することで収集されます。ヒープが大きいほど、この操作を実行するのに時間がかかるため、このビューは現在の値で自動的に更新されません。



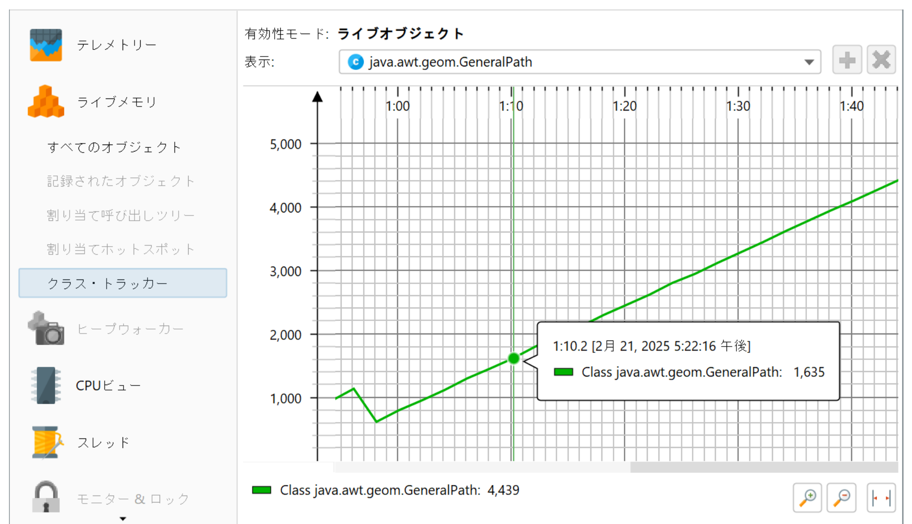
メモリリークを探す際には、インスタンス数を時間とともに比較したいことがよくあります。すべてのクラスについてそれを行うには、ビューの差分機能を使用できます。すべてのオブジェクトの2つのダンプが同時に選択されると、差分列が挿入され、インスタンス数のヒストグラムは緑色でマーク時の基準値を示します。すべてのオブジェクトの新しいダンプを取得すると、最も古い選択されたダンプが選択されたままになり、すべてのオブジェクトの新しいダンプとの違いが表示されます。



ダンプセレクタは、ダンプが取得された時刻を示します。ダブルクリックでラベルを追加して、識別を容易にすることができます。すべてのオブジェクトのダンプは、トリガーアクション[p. 28]またはController APIでもトリガーでき、ラベルを指定することもできます。

一方、「記録されたオブジェクト」ビューは、割り当て記録を開始した後に割り当てられたオブジェクトのインスタンス数のみを表示します。割り当て記録を停止すると、新しい割り当ては追加されませんが、ガベージコレクションは追跡され続けます。このようにして、特定のユースケースでヒープに残るオブジェクトを確認できます。オブジェクトが長時間ガベージコレクションされないことに注意してください。GC実行ツールバーボタンを使用して、このプロセスを加速できます。動的に更新されるほとんどのビューと同様に、フリーズツールバーボタンを使用して表示されるデータの更新を停止できます。

現在をマークツールバーボタンを使用すると、「記録されたオブジェクト」ビューでも選択された基準に対する差分列を表示できます。選択されたクラスについては、コンテキストメニューの選択をクラス・トラッカーに追加アクションを使用して、時間解決されたグラフを表示することもできます。




### 割り当てスポット

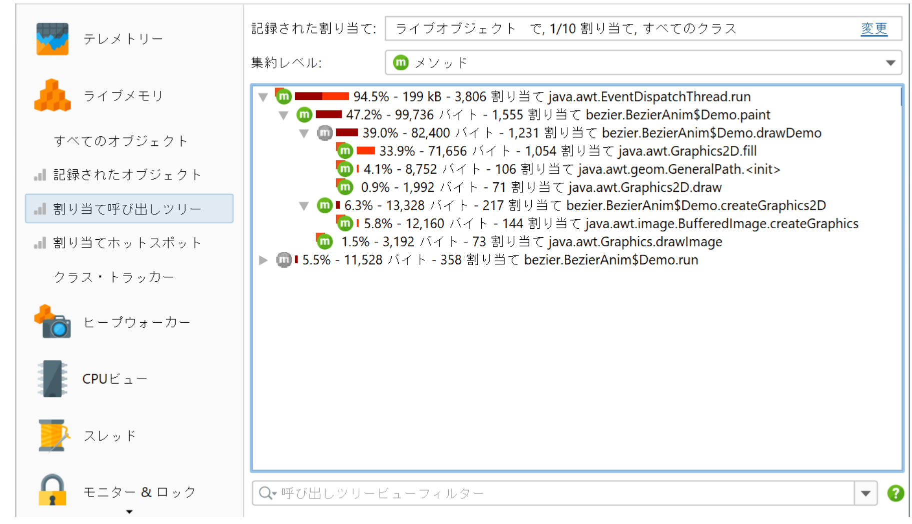
割り当て記録がアクティブな場合、JProfilerはオブジェクトが割り当てられるたびに呼び出しスタックを記録します。スタックウォーキングAPIからの正確な呼び出しスタックは使用しません。なぜなら、それは非常に高価だからです。代わりに、CPUプロファイリングに設定されたのと同じメカニズムが使用されます。つまり、呼び出しスタックは呼び出しツリーフィルタ[p. 54]に従ってフィルタリングされ、実際の割り当てスポットは、無視されたクラスやコンパクトフィルタされたクラスからのメソッドであるため、呼び出しスタックに存在しない場合があります。しかし、これらの変更は直感的に理解しやすいです。コンパクトフィルタされたメソッドは、コンパクトフィルタされたクラスへのさらなる呼び出しで行われるすべての割り当てに対して責任を負います。

サンプリングを使用すると、割り当てスポットが近似的になり、混乱を招くことがあります。時間測定とは異なり、特定のクラスがどこで割り当てられるか、どこで割り当てられないかについて明確なアイデアを持っていることがよくあります。サンプリングは統計的な絵を描くため、`java.util.HashMap.get`が自分のクラスを割り当てるような、明らかに不可能な割り当てスポットが表示されることがあります。正確な数値や呼び出しスタックが重要な分析には、計測とともに割り当て記録を使用することをお勧めします。

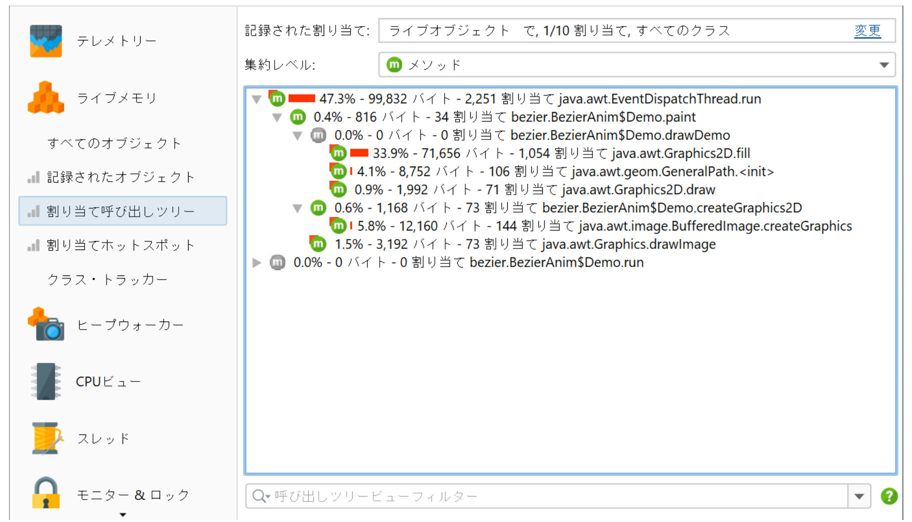
CPUプロファイリングと同様に、割り当て呼び出しスタックは呼び出しツリーとして表示され、呼び出し回数や時間ではなく、割り当て数と割り当てられたメモリが表示されます。CPU呼び出しツリーとは異なり、割り当て呼び出しツリーは自動的に表示および更新されません。なぜなら、ツリーの計算がより高価だからです。JProfilerは、すべてのオブジェクトだけでなく、選択されたク

ラスやパッケージの割り当てツリーも表示できます。他のオプションとともに、現在のデータから割り当てツリーを計算するようにJProfilerに依頼した後に表示されるオプションダイアログで設定されます。

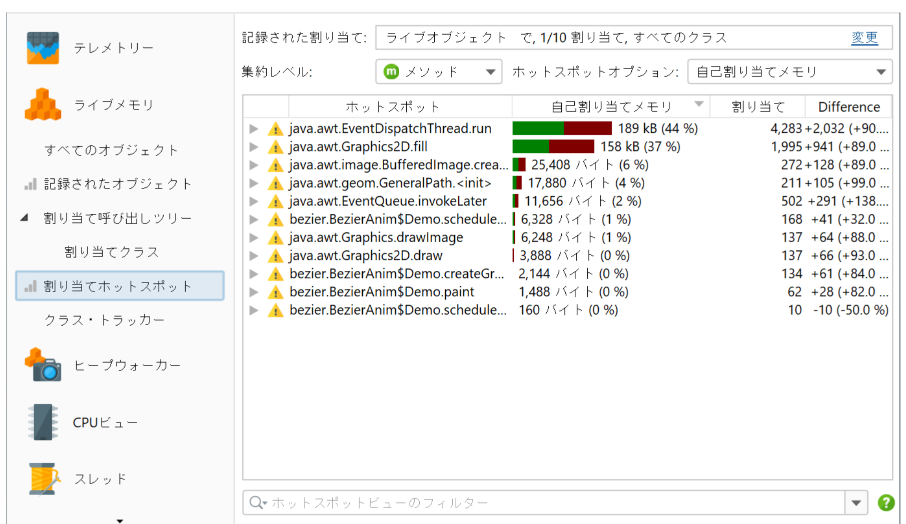
CPU呼び出しツリーの便利な特性は、各ノードが子ノードで費やされた時間を含むため、上から下に累積時間を追跡できることです。デフォルトでは、割り当てツリーも同じように動作し、各ノードが子ノードによって行われた割り当てを含みます。たとえ割り当てが呼び出しツリーの深い葉ノードでのみ行われたとしても、数値は上に伝播します。このようにして、割り当て呼び出しツリーのブランチを開く際にどのパスを調査する価値があるかを常に確認できます。「自己割り当て」は、実際にノードによって行われたものであり、その子孫によって行われたものではありません。CPU呼び出しツリーと同様に、パーセンテージバーは異なる色で表示されます。




割り当て呼び出しツリーでは、特に選択されたクラスの割り当てを表示する場合、割り当てがまったく行われないノードが多く存在します。これらのノードは、実際の割り当てが行われたノードに至る呼び出しスタックを示すためにのみ存在します。これらのノードはJProfilerでは「ブリッジ」ノードと呼ばれ、上のスクリーンショットのように灰色のアイコンで表示されます。場合によっては、割り当ての累積が邪魔になり、実際の割り当てスポットのみを表示したいことがあります。割り当てツリーのビュー設定ダイアログには、その目的のために非累積数値を表示するオプションがあります。有効にすると、ブリッジノードは常にゼロの割り当てを表示し、パーセンテージバーを持ちません。

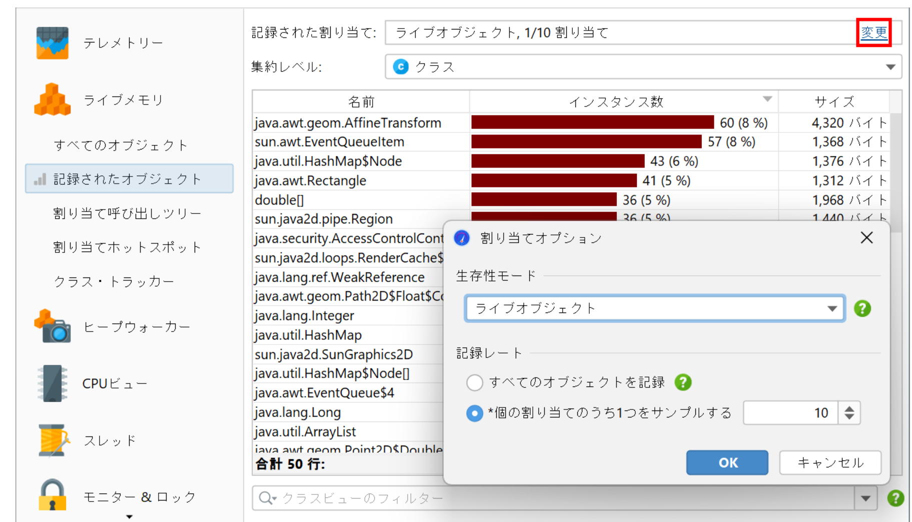



割り当てホットスポットビューは、割り当て呼び出しツリーとともにポピュレートされ、選択されたクラスを作成する責任があるメソッドに直接焦点を当てることができます。記録されたオブジェクトビューと同様に、割り当てホットスポットビューは現在の状態をマークし、時間とともに変化を観察することをサポートします。差分列がビューに追加され、現在の値をマークアクションが呼び出された時点からホットスポットがどの程度変化したかを示します。デフォルトでは、割り当てビューは定期的に更新されないため、計算ツールバーボタンをクリックして新しいデータセットを取得し、基準値と比較する必要があります。オプションダイアログで自動更新が利用可能ですが、大きなヒープサイズには推奨されません。




## 割り当て記録率

すべての割り当てを記録することは、かなりのオーバーヘッドを追加します。多くの場合、割り当ての総数は重要ではなく、相対的な数値が問題を解決するのに十分です。これが、JProfilerがデフォルトで10回に1回の割り当てのみを記録する理由です。これにより、すべての割り当てを記録するのと比較して、オーバーヘッドが約1/10に減少します。すべての割り当てを記録したい場合や、目的に対してさらに少ない割り当てが十分である場合は、記録されたオブジェクトビューや割り当て呼び出しツリーおよびホットスポットビューのパラメータダイアログで記録率を変更できます。

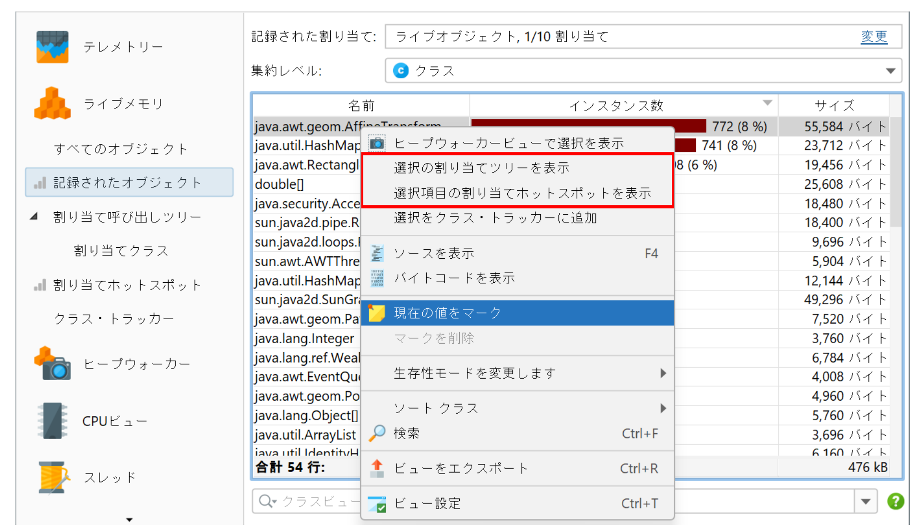



この設定は、「詳細設定->メモリプロファイリング」ステップのセッション設定ダイアログでも見つけることができ、オフラインプロファイリングセッション用に調整できます。

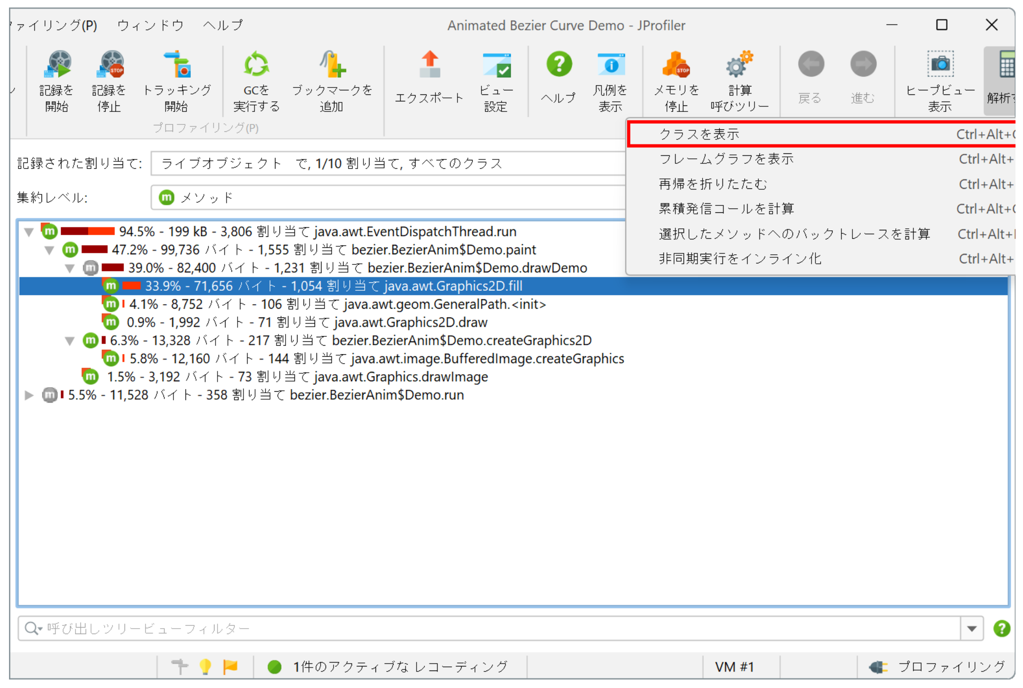
割り当て記録率は、「記録されたオブジェクト」と「記録されたスループット」のVMテレメトリーに影響を与え、その値は設定された分数で測定されます。スナップショットを比較[p.135]する際には、最初のスナップショットの割り当て率が報告され、必要に応じて他のスナップショットがそれに応じてスケーリングされます。

## 割り当てられたクラスの分析

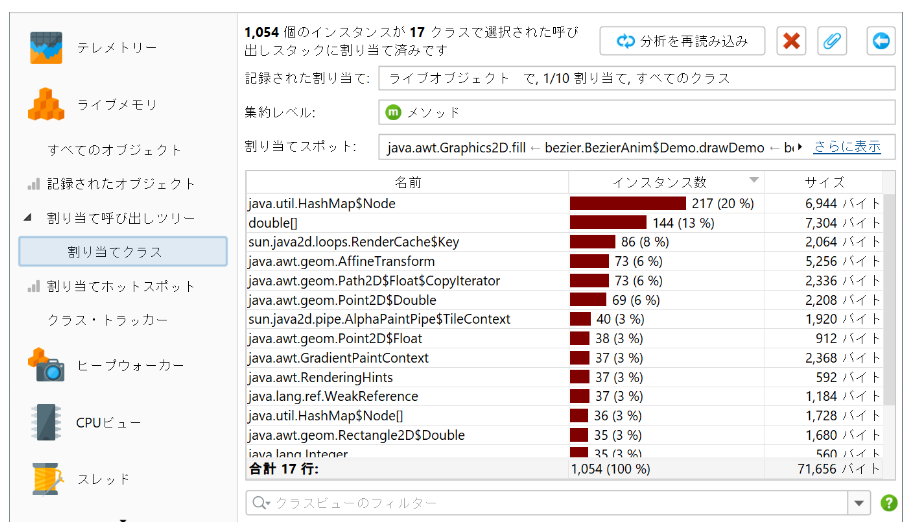
割り当てツリーと割り当てホットスポットビューを計算する際には、事前に見たいクラスやパッケージを指定する必要があります。特定のクラスにすでに焦点を当てている場合にはうまく機能しますが、事前の予想なしに割り当てホットスポットを見つけようとする場合には不便です。1つの方法は、「記録されたオブジェクト」ビューを見て、選択したクラスやパッケージの割り当てツリーまたは割り当てホットスポットビューに切り替えるためのコンテキストメニューのアクションを使用することです。




もう1つの方法は、すべてのクラスの割り当てツリーまたは割り当てホットスポットから始めて、クラスを表示アクションを使用して、選択された割り当てスポットまたは割り当てホットスポットのクラスを表示することです。




割り当てられたクラスのヒストグラムは、呼び出しツリー分析[p. 190]として表示されます。このアクションは他の呼び出しツリー分析からも機能します。



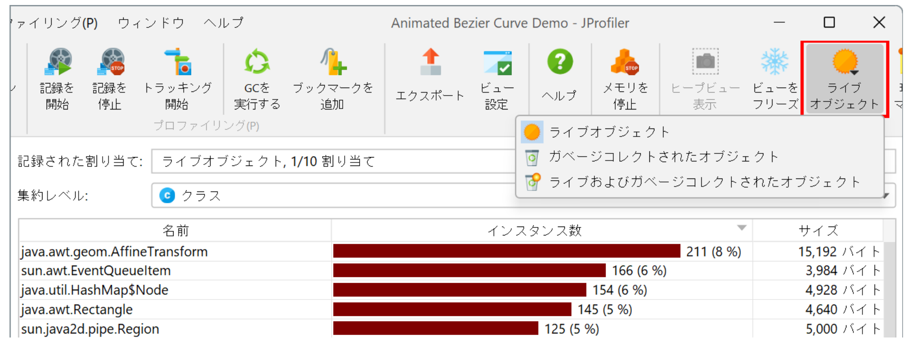

クラス分析ビューは静的であり、割り当てツリーとホットスポットビューが再計算されても更新されません。分析を再読み込みアクションは、最初に割り当てツリーを更新し、新しいデータから現在の分析ビューを再計算します。

### ガベージコレクションされたオブジェクトの分析

割り当て記録は、ライブオブジェクトを表示するだけでなく、ガベージコレクションされたオブジェクトに関する情報も保持します。これは、一時的な割り当てを調査する際に役立ちます。多くの一時オブジェクトを割り当てることは、かなりのオーバーヘッドを生む可能性があるため、割り当て率を下げることでパフォーマンスが大幅に向上することがあります。

記録されたオブジェクトビューでガベージコレクションされたオブジェクトを表示するには、生存性セレクタをガベージコレクションされたオブジェクトまたはライブおよびガベージコレクション

されたオブジェクトに変更します。割り当て呼び出しツリーおよび割り当てホットスポットビューのオプションダイアログには同等のドロップダウンがあります。

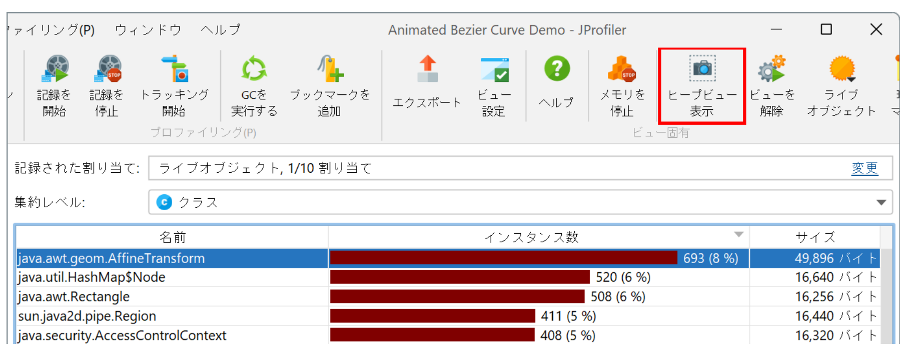



しかし、JProfilerはデフォルトでガベージコレクションされたオブジェクトの割り当てツリー情報を収集しません。なぜなら、ライブオブジェクトのみのデータは、はるかに少ないオーバーヘッドで維持できるからです。「割り当て呼び出しツリー」または「割り当てホットスポット」ビューでガベージコレクションされたオブジェクトを含むモードに生存性セレクタを切り替えると、JProfilerは記録タイプを変更することを提案します。これはプロファイリング設定の変更であるため、変更をすぐに適用することを選択した場合、以前に記録されたすべてのデータがクリアされます。この設定を事前に変更したい場合は、セッション設定ダイアログの「詳細設定」->「メモリプロファイリング」で行うことができます。

## 次のステップ: ヒープウォーカー

より高度なタイプの質問は、オブジェクト間の参照を含むことになります。たとえば、記録されたオブジェクト、割り当てツリー、および割り当てホットスポットビューに表示されるサイズは**浅いサイズ**です。それらはクラスのメモリレイアウトのみを含み、参照されたクラスは含まれません。クラスのオブジェクトが実際にどれだけ重いかを確認するために、**保持サイズ**、つまりそれらのオブジェクトがヒープから削除された場合に解放されるメモリ量を知りたいことがよくあります。

この種の情報は、ヒープ上のすべてのオブジェクトを列挙し、高価な計算を行う必要があるため、ライブメモリビューでは利用できません。その作業はヒープウォーカーによって処理されます。ライブメモリビューの興味のあるポイントからヒープウォーカーにジャンプするには、ヒープウォーカーで表示ツールバーボタンを使用できます。それにより、ヒープウォーカーの同等のビューに移動します。




ヒープスナップショットが利用できない場合は、新しいヒープスナップショットが作成されます。それ以外の場合は、JProfilerが既存のヒープスナップショットを使用するかどうかを尋ねます。

いずれにせよ、ライブメモリビューとヒープウォーカーの数値が非常に異なることがよくあることを理解することが重要です。ヒープウォーカーは、ライブメモリビューとは異なる時点でのスナップショットを表示するだけでなく、すべての参照されていないオブジェクトを排除します。ガベージコレクタの状態によっては、参照されていないオブジェクトがヒープのかなりの部分を占めることがあります。

# ヒープウォーカー

## ヒープスナップショット

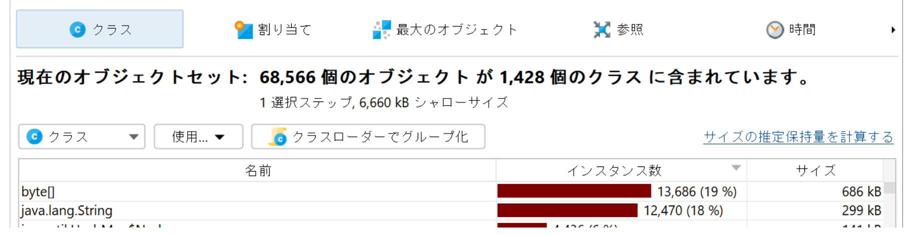
オブジェクト間の参照を含むヒープ分析にはヒープスナップショットが必要です。なぜなら、JVMにオブジェクトへのインカミング参照を尋ねることはできないからです。その質問に答えるには、ヒープ全体を反復処理する必要があります。そのヒープスナップショットから、JProfilerはヒープウォーカーのビューを提供するために必要なデータを生成するために最適化された内部データベースを作成します。

ヒープスナップショットには2つのソースがあります：JProfilerヒープスナップショットとHPROF/PHDヒープスナップショットです。JProfilerヒープスナップショットはヒープウォーカーで利用可能なすべての機能をサポートしています。プロファイリングエージェントはプロファイリングインターフェースJVMTIを使用してすべての参照を反復処理します。プロファイルされたJVMが異なるマシンで実行されている場合、すべての情報はローカルマシンに転送され、そこでさらに計算が行われます。HPROF/PHDスナップショットはJVMの組み込みメカニズムで作成され、JProfilerが読み取ることができる標準フォーマットでディスクに書き込まれます。HotSpot JVMはHPROFスナップショットを作成でき、Eclipse OpenJ9 JVMはPHDスナップショットを提供します。

ヒープウォーカーの概要ページでは、JProfilerヒープスナップショットまたはHPROF/PHDヒープスナップショットを作成するかを選択できます。デフォルトでは、JProfilerヒープスナップショットが推奨されます。HPROF/PHDヒープスナップショットは、別の章[p.207]で説明されている特別な状況で役立ちます。

## 選択ステップ

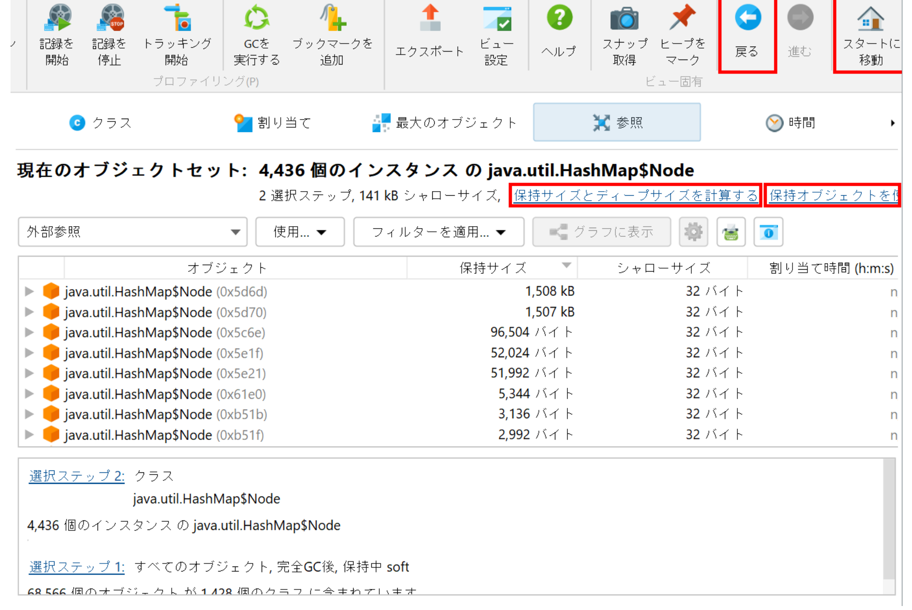
ヒープウォーカーは、選択されたオブジェクトセットの異なる側面を示すいくつかのビューで構成されています。ヒープスナップショットを取得した直後は、ヒープ上のすべてのオブジェクトを見ています。各ビューには、選択されたオブジェクトを**現在のオブジェクトセット**に変えるためのナビゲーションアクションがあります。ヒープウォーカーのヘッダーエリアには、現在のオブジェクトセットに含まれるオブジェクトの数に関する情報が表示されます。




最初は、ライブメモリセクション[p.72]の「すべてのオブジェクト」ビューに似た「クラス」ビューを見ています。クラスを選択して使用->選択されたインスタンスを呼び出すことで、そのクラスのインスタンスのみを含む新しいオブジェクトセットを作成します。ヒープウォーカーでは、「使用する」とは常に新しいオブジェクトセットを作成することを意味します。

新しいオブジェクトセットに対して、ヒープウォーカーのクラスビューを表示することは興味深くありません。なぜなら、それは実質的に以前に選択されたクラスにテーブルをフィルタリングするだけだからです。代わりに、JProfilerは「新しいオブジェクトセット」ダイアログで別のビューを提案します。このダイアログをキャンセルして新しいオブジェクトセットを破棄し、以前のビューに戻ることができます。アウトゴーイング参照ビューが提案されますが、別のビューを選択することもできます。これは最初に表示されるビューに過ぎません。その後、ヒープウォーカーのビューセレクターでビューを切り替えることができます。

ヘッダーエリアには、現在2つの選択ステップがあることが示され、保持サイズとディープサイズを計算するためのリンクや、現在のオブジェクトセットによって保持されているすべてのオブジェクトを使用するためのリンクが含まれています。後者は別の選択ステップを追加し、オブジェクトセットに複数のクラスが含まれる可能性があるため、クラスビューを提案します。

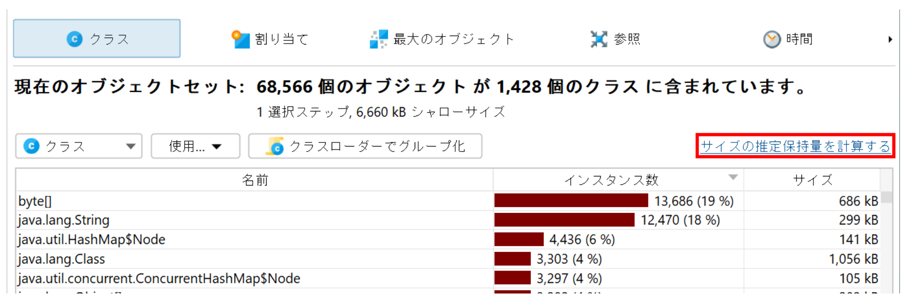



ヒープウォーカーの下部には、これまでの選択ステップが一覧表示されています。ハイパーリンクをクリックすると、任意の選択ステップに戻ることができます。最初のデータセットは、ツールバーのスタートに移動ボタンでもアクセスできます。ツールバーの戻るボタンと進むボタンは、分析でバックトラックする必要がある場合に便利です。

### クラスビュー

ヒープウォーカーの上部にあるビューセレクターには、現在のオブジェクトセットに対する異なる情報を示す5つのビューが含まれています。その最初のものが「クラス」ビューです。

クラスビューは、ライブメモリセクションの「すべてのオブジェクト」ビューに似ており、クラスをパッケージにグループ化できる集約レベルの選択肢があります。さらに、クラスの推定保持サイズを表示することができます。これは、クラスのすべてのインスタンスがヒープから削除された場合に解放されるメモリの量です。推定保持サイズを計算ハイパーリンクをクリックすると、新しい保持サイズ列が追加されます。表示される保持サイズは推定された下限であり、正確な数値を計算するには時間がかかりすぎます。本当に正確な数値が必要な場合は、興味のあるクラスまたはパッケージを選択し、新しいオブジェクトセットのヘッダーにある保持サイズとディープサイズを計算ハイパーリンクを使用してください。




1つ以上のクラスまたはパッケージを選択すると、インスタンス自体、関連する`java.lang.Class`オブジェクト、またはすべての保持オブジェクトを選択できます。ダブルクリックは最も迅速な選択モードであり、選択されたインスタンスを使用します。複数の選択モードが利用可能な場合、このケースのように、ビューの上に使用ドロップダウンメニューが表示されます。

クラスローダー関連の問題を解決する際には、インスタンスをクラスローダーでグループ化する必要があることがよくあります。インスペクションタブには、クラスビューで利用可能な「クラスローダーでグループ化」インスペクションがあり、そのコンテキストで特に重要です。その分析を実行すると、上部にあるグループ化テーブルにすべてのクラスローダーが表示されます。クラスローダーを選択すると、以下のビューでデータがそれに応じてフィルタリングされます。別の選択ステップを実行するまで、グループ化テーブルはヒープウォーカーの他のビューに切り替えてもそのまま残ります。その後、クラスローダーの選択がその選択ステップの一部になります。

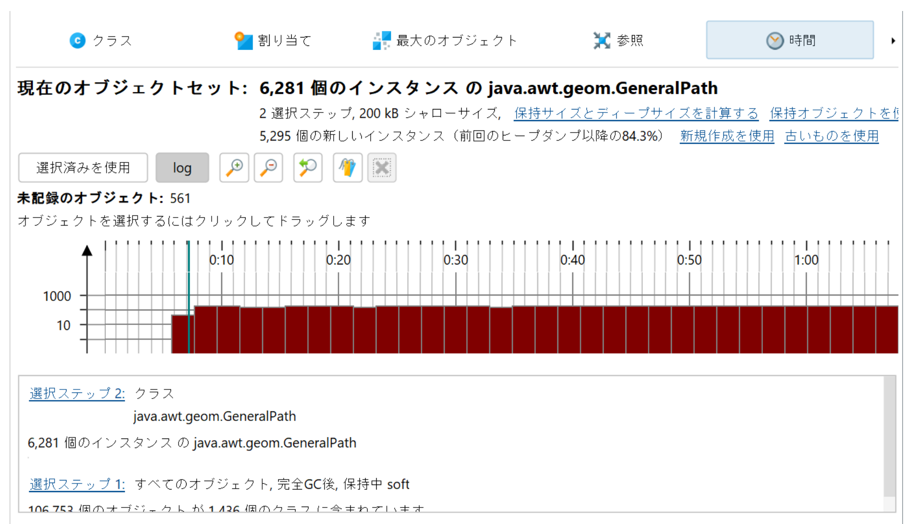
## 割り当て記録ビュー

オブジェクトがどこで割り当てられたかの情報は、メモリリークの疑いを絞り込む際やメモリ消費を削減しようとする際に重要です。JProfilerヒープスナップショットの場合、「割り当て」ビューは、割り当てコールツリーと割り当てホットスポットを示します。割り当てが記録されたオブジェクトに対して、割り当てコールツリーに「未記録オブジェクト」ノードにグループ化されます。HPROF/PHDスナップショットの場合、このビューは利用できません。

クラスビューと同様に、複数のノードを選択し、上部の選択されたものを使用ボタンを使用して新しい選択ステップを作成できます。「割り当てホットスポット」ビューのモードでは、バックトレース内のノードも選択できます。これにより、選択されたバックトレースで終了するコールス

タックで割り当てられた、関連するトップレベルホットスポットのオブジェクトのみが選択されます。

JProfilerが割り当てを記録する際に保存できるもう1つの情報は、オブジェクトが割り当てられた時間です。ヒープウォーカーの「時間」ビューは、現在のオブジェクトセット内のすべての記録されたインスタンスの割り当て時間のヒストグラムを示します。クリックしてドラッグすることで、1つまたは複数の間隔を選択し、選択されたものを使用ボタンで新しいオブジェクトセットを作成できます。



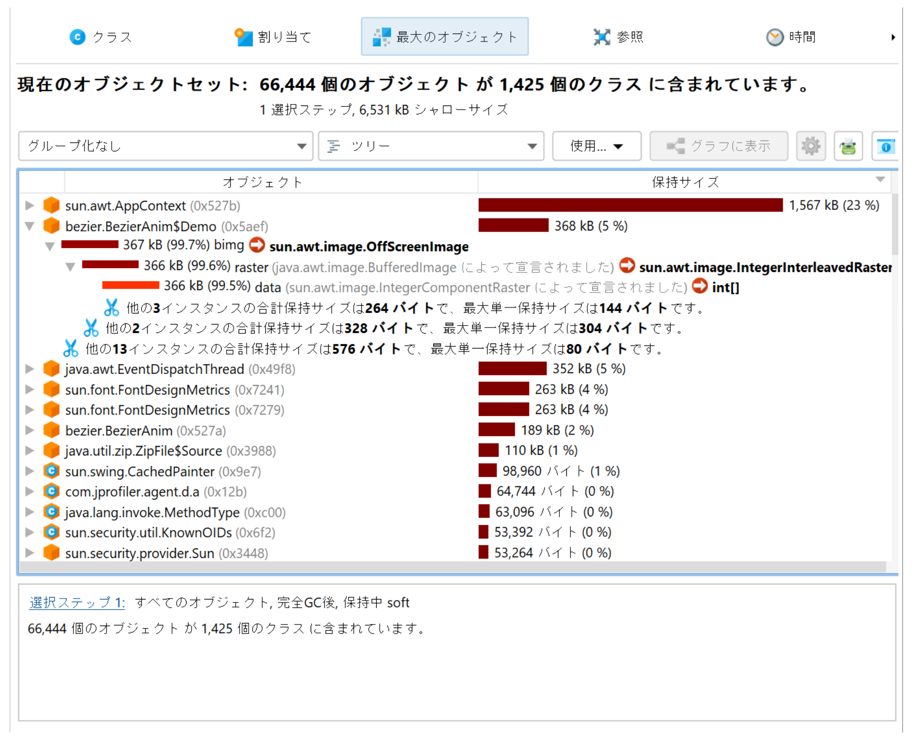

時間間隔をより正確に選択するために、ブックマーク[p. 46]の範囲を指定できます。最初と最後に選択されたブックマークの間のすべてのオブジェクトがマークされます。

時間ビューに加えて、割り当て時間は参照ビューの別の列として表示されます。ただし、割り当て時間の記録はデフォルトでは有効になっていません。時間ビューで直接オンにするか、セッション設定ダイアログの詳細設定 -> メモリプロファイリングで設定を編集できます。

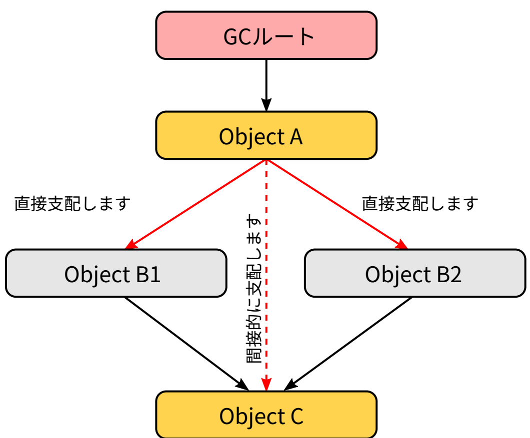
### 最大オブジェクトビュー

最大オブジェクトビューは、現在のオブジェクトセット内の最も重要なオブジェクトのリストを表示します。このコンテキストでの「最大」とは、ヒープから削除された場合に最も多くのメモリを解放するオブジェクトを意味します。そのサイズは**保持サイズ**と呼ばれます。対照的に、**ディープサイズ**は、強い参照を通じて到達可能なすべてのオブジェクトの合計サイズです。

各オブジェクトは、他のオブジェクトへのアウトゴーイング参照を表示するために展開できます。この方法で、祖先の1つが削除された場合にガベージコレクトされる保持オブジェクトのツリーを再帰的に展開できます。この種のツリーは「ドミネーターツリー」と呼ばれます。このツリー内の各オブジェクトに表示される情報は、アウトゴーイング参照ビューと似ていますが、支配的な参照のみが表示されます。




すべての支配されたオブジェクトがその支配者によって直接参照されるわけではありません。たとえば、次の図の参照を考えてみましょう：




オブジェクトAはオブジェクトB1とB2を支配し、オブジェクトCへの直接参照を持っていません。B1とB2の両方がCを参照しています。B1もB2もCを支配していませんが、Aは支配しています。この場合、B1、B2、およびCはドミネーターツリーのAの直接の子としてリストされ、CはB1とB2の子としてリストされません。B1とB2の場合、Aで保持されているフィールド名が表示されます。Cの場合、参照ノードに「[推移的参照]」が表示されます。

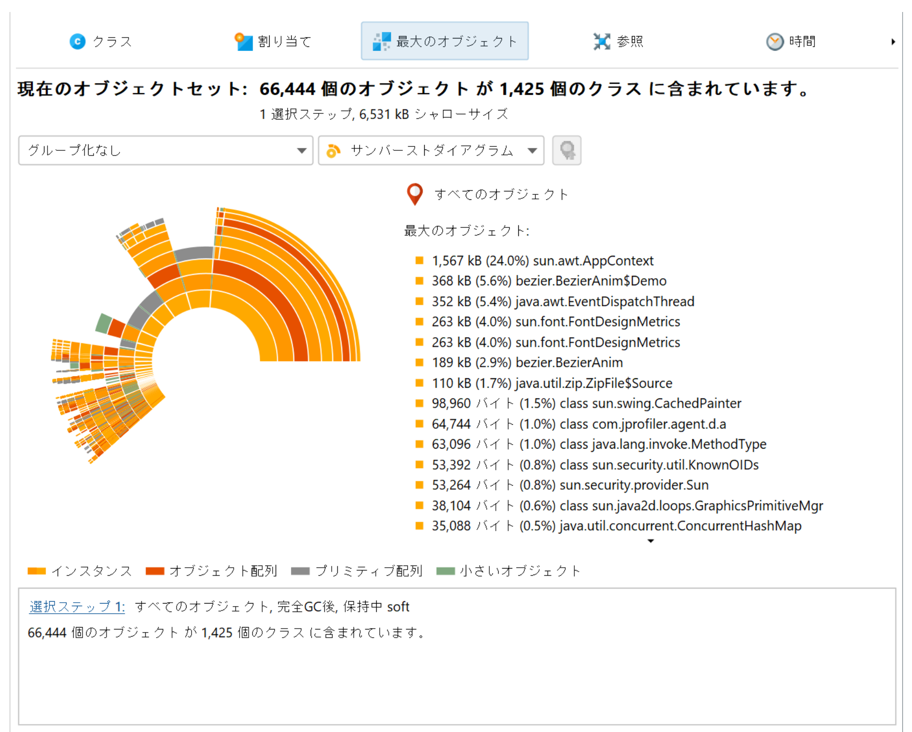
ドミネーターツリーの各参照ノードの左側には、トップレベルオブジェクトの保持サイズの何パーセントがターゲットオブジェクトによってまだ保持されているかを示すサイズバーがあります。ツリーをさらに掘り下げると、数値は減少します。ビュー設定では、パーセンテージの基準をヒープ全体のサイズに変更できます。

ドミネーターツリーには、親オブジェクトの保持サイズの0.5%未満の保持サイズを持つすべてのオブジェクトを排除する組み込みのカットオフがあります。これは、重要なオブジェクトから気をそらす小さな支配されたオブジェクトの非常に長いリストを避けるためです。このようなカットオフが発生すると、このレベルで表示されていないオブジェクトの数、その合計保持サイズ、および単一オブジェクトの最大保持サイズを通知する特別な「カットオフ」子ノードが表示されます。

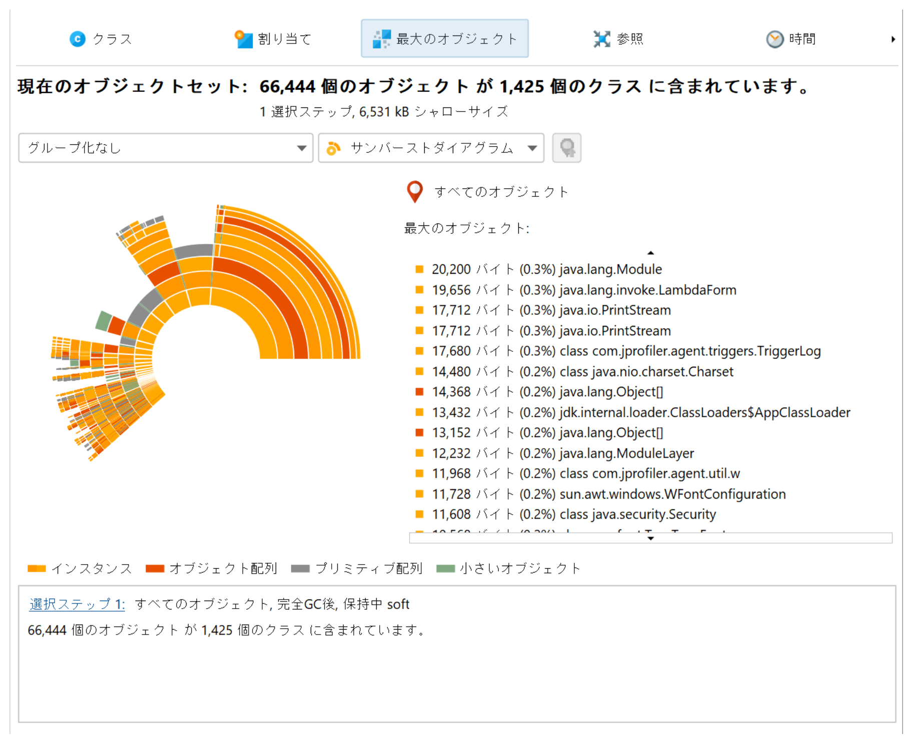
単一のオブジェクトを表示する代わりに、ドミネーターツリーは最大オブジェクトをクラスにグループ化することもできます。ビューの上部にあるグループ化ドロップダウンには、この表示モードをアクティブにするチェックボックスが含まれています。さらに、トップレベルでクラスローダーのグループ化を追加できます。クラスローダーのグループ化は、最大オブジェクトが計算された後に適用され、最大オブジェクトのクラスを誰がロードしたかを示します。特定のクラスローダーの最大オブジェクトを分析したい場合は、最初に「クラスローダーでグループ化」インスペクションを使用できます。

最大オブジェクトビューの上にある表示モードセレクターを使用すると、サンバーストダイアグラムに切り替えることができます。このダイアグラムは、一連の同心円状のセグメントリングで構成されており、最大深度までのドミネーターツリーの全体の内容を1つの画像で示します。参照は最も内側のリングから始まり、円の外縁に向かって伝播します。この視覚化は、高情報密度の平坦な視点を提供し、参照パターンを発見し、特別な色分けを通じて大きなプリミティブおよびオブジェクト配列を一目で確認できるようにします。

現在のオブジェクトセットがヒープ全体である場合、円の全周は使用されたヒープサイズに対応します。最大オブジェクトビューは、ヒープ全体の0.1%以上を保持するオブジェクトのみを表示するため、最大オブジェクトによって保持されていないすべてのオブジェクトに対応する実質的なセクターが空になります。




任意のリングセグメントをクリックすると、円の新しいルートが設定され、ダイアグラムで確認できる最大深度が拡張されます。ダイアグラムの中空の中心をクリックすると、以前のルートが復元されます。新しいルートが設定されている場合、円の全周はルートオブジェクトの保持サイズに対応します。空のセクターは、ルートオブジェクトの自己サイズと、最大保持オブジェクトのリストに存在しない追加のオブジェクトを表します。現在のオブジェクトセットがヒープ全体でない場合、円の全周は表示されている最大オブジェクトの合計に対応し、空のセクターは表示されません。




インスタンスとその直ちに保持されたオブジェクトに関する詳細情報は、マウスを上に移動するとダイアグラムの右側に表示されます。マウスがリングセグメントの外にある場合、右側のリストには最も内側のリングの最大オブジェクトが表示されます。そのリストにマウスを移動すると、対応するリングセグメントがハイライトされ、リスト項目をクリックするとダイアグラムの新しいルートが設定されます。新しいオブジェクトセットを作成するには、リングセグメントやリスト項目のコンテキストメニューのアクションから選択できます。

### 参照ビュー

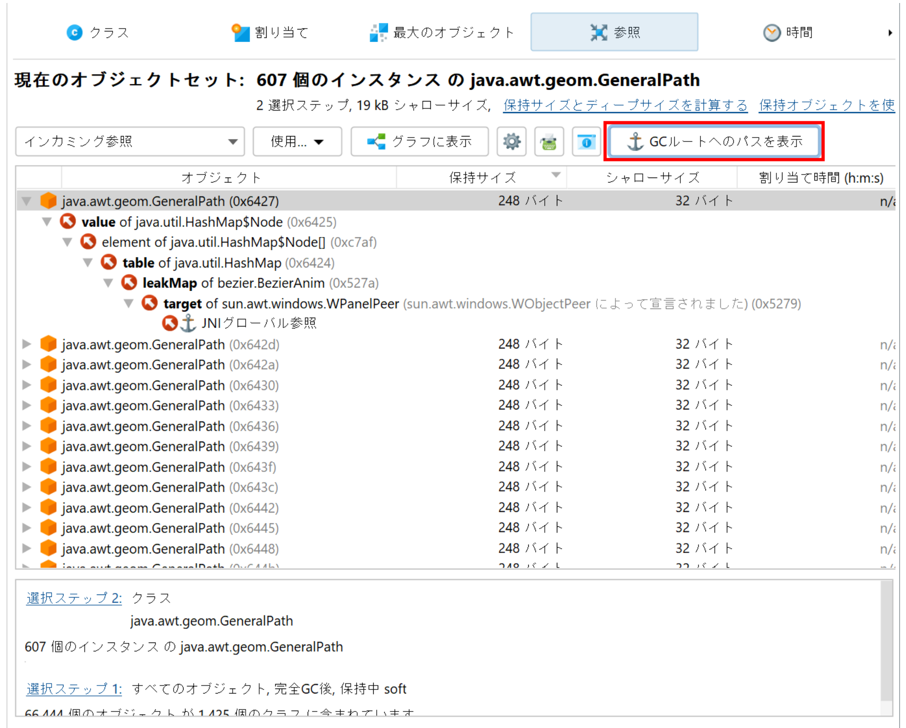
前のビューとは異なり、参照ビューは少なくとも1つの選択ステップを実行した場合にのみ利用可能です。初期オブジェクトセットに対しては、これらのビューは役に立ちません。なぜなら、インカミングおよびアウトゴーイング参照ビューはすべての個々のオブジェクトを表示し、マージされた参照ビューは特定のオブジェクトセットに対してのみ解釈できるからです。

アウトゴーイング参照ビューは、IDEのデバッガーが表示するビューに似ています。オブジェクトを開くと、プリミティブデータと他のオブジェクトへの参照が表示されます。任意の参照タイプを新しいオブジェクトセットとして選択でき、複数のオブジェクトを一度に選択できます。クラスビューと同様に、保持オブジェクトや関連する`java.lang.Class`オブジェクトを選択できます。選択されたオブジェクトが標準コレクションの場合、単一のアクションで含まれるすべての要素を選択することもできます。クラスローダーオブジェクトの場合、ロードされたすべてのインスタンスを選択するオプションがあります。

null参照を持つフィールドは、メモリ分析の際に気をそらす可能性があるため、デフォルトでは表示されません。デバッグ目的でフィールドをすべて表示したい場合は、ビュー設定でこの動作を変更できます。

表示されたインスタンスの単純な選択に加えて、アウトゴーイング参照ビューには、強力なフィルタリング機能[p.212]があります。ライブセッションでは、アウトゴーイングおよびインカミング参照ビューの両方に、同じ章で説明されている高度な操作および表示機能があります。

インカミング参照ビューは、メモリリークを解決するための主要なツールです。オブジェクトがなぜガベージコレクトされないのかを調べるために、GCルートへのパスを表示ボタンは、ガベージコレクタールートへの参照チェーンを見つけます。メモリリーク[p.216]に関する章には、この重要なトピックに関する詳細情報があります。

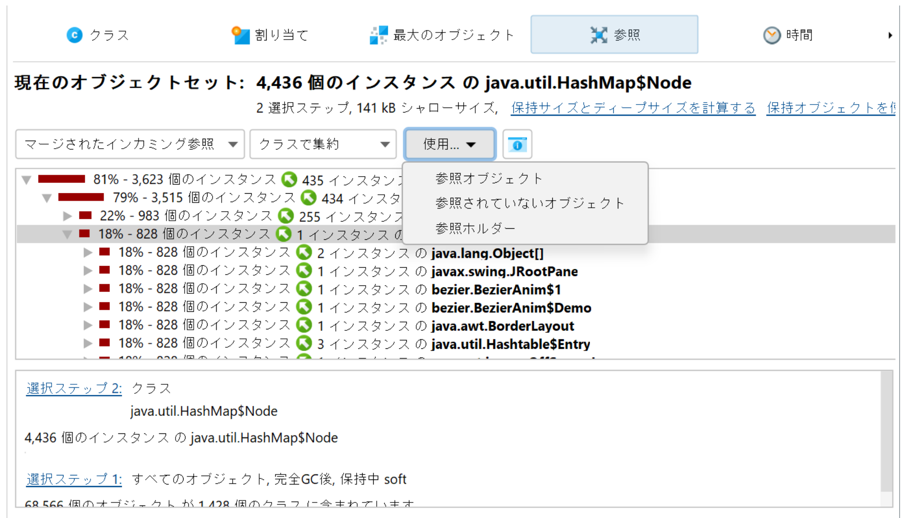



## マージされた参照

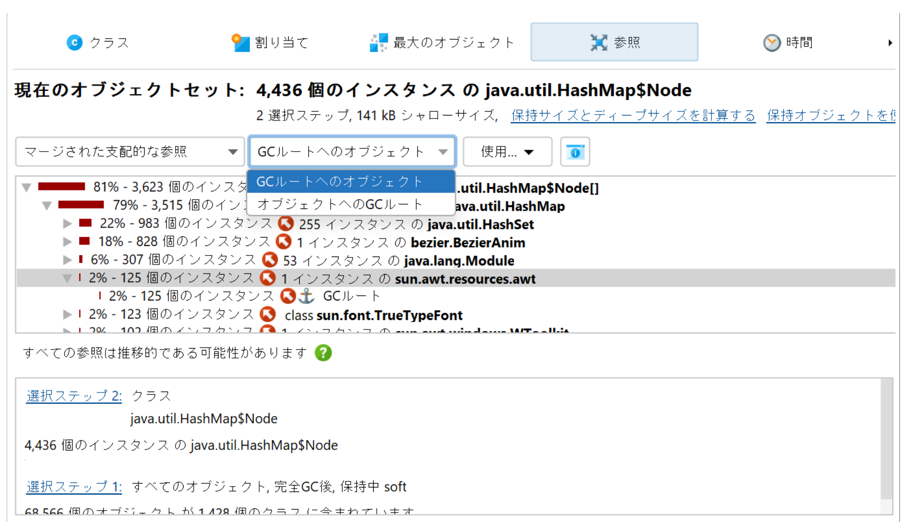
多くの異なるオブジェクトの参照を確認するのは面倒な場合があるため、JProfilerは現在のオブジェクトセット内のすべてのオブジェクトのマージされたアウトゴーイングおよびインカミング参照を表示できます。デフォルトでは、参照はクラスごとに集約されます。クラスのインスタンスが同じクラスの他のインスタンスによって参照されている場合、特別なノードが挿入され、これらのクラス再帰参照からの元のインスタンスとインスタンスを示します。このメカニズムは、リンクリストのような一般的なデータ構造内の内部参照チェーンを自動的に折りたたみます。

フィールドごとにグループ化されたマージされた参照を表示することもできます。その場合、各ノードはクラスの特定のフィールドや配列の内容などの参照タイプです。標準コレクションの場合、累積を妨げる内部参照チェーンが圧縮されるため、「java.lang.HashMapのマップ値」のような参照タイプが表示されます。クラス集約の場合とは異なり、このメカニズムはJREの標準ライブラリから明示的にサポートされているコレクションに対してのみ機能します。

「マージされたアウトゴーイング参照」ビューでは、インスタンスカウントは参照されたオブジェクトを指します。「マージされたインカミング参照」ビューでは、各行に2つのインスタンスカウントが表示されます。最初のインスタンスカウントは、現在のオブジェクトセット内のインスタンスがこのパスに沿って参照されている数を示します。ノードの左側にあるバーアイコンは、この割合を視覚化します。矢印アイコンの後の2番目のインスタンスカウントは、親ノードへの参照を保持しているオブジェクトを指します。選択ステップを実行する際に、選択された方法で参照されている現在のオブジェクトセットからオブジェクトを選択するか、選択された参照を持つオブジェクト（参照ホルダー）に興味があるかを選択できます。




「マージされた支配参照」ビューを使用すると、現在のオブジェクトセット内の一部またはすべてのオブジェクトがガベージコレクトされるために削除する必要がある参照を見つけることができます。支配参照ツリーは、最大オブジェクトビューのドミネーターツリーのマージされた逆として解釈でき、クラスごとに集約されます。参照矢印は、2つのクラス間の直接参照を表していない場合がありますが、支配的でない参照を保持している他のクラスが間にある場合があります。複数のガベージコレクタールートがある場合、現在のオブジェクトセット内の一部またはすべてのオブジェクトに対して支配参照が存在しない場合があります。



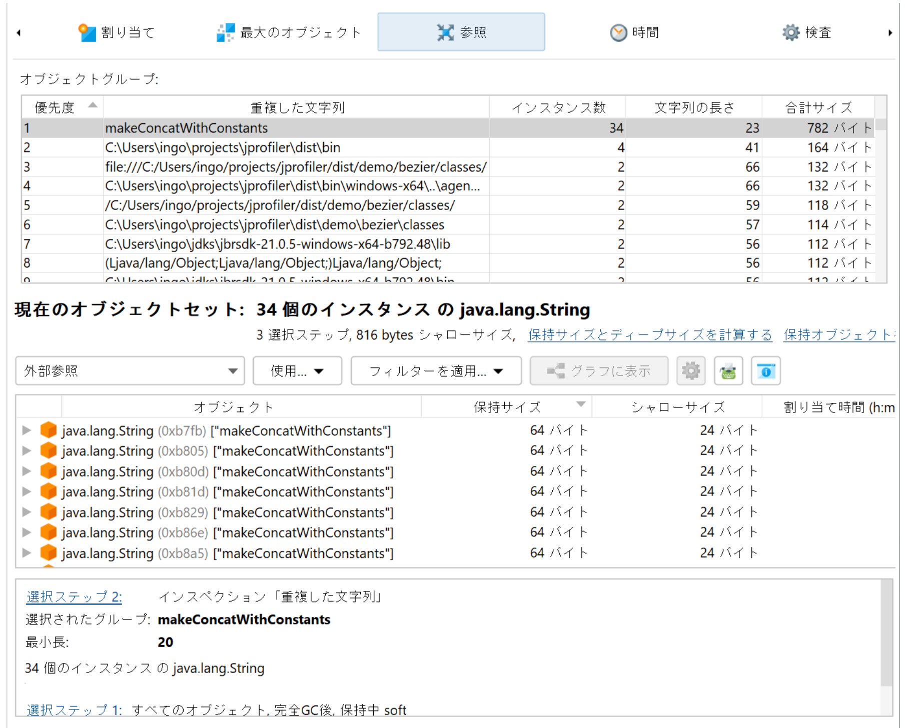

デフォルトでは、「マージされた支配参照」ビューはインカミング支配参照を表示し、ツリーを開くことで、GCルートによって保持されているオブジェクトに到達できます。時には、参照ツリーが多くの異なるパスに沿って同じルートオブジェクトに導くことがあります。ビューの上部にあるドロップダウンで「GCルートからオブジェクトへのビュー」モードを選択することで、ルートがトップレベルにあり、現在のオブジェクトセットがリーフノードにある逆の視点を見ることができます。その場合、参照はトップレベルからリーフノードに向かって進みます。どの視点がより良いかは、削除したい参照が現在のオブジェクトセットに近いか、GCルートに近いかによります。

### インスペクション

「インスペクション」ビューはデータ自体を表示しません。他のビューでは利用できないルールに従って新しいオブジェクトセットを作成するいくつかのヒープ分析を提示します。たとえば、ス

レッドローカルによって保持されているすべてのオブジェクトを表示したい場合があります。これは参照ビューでは不可能です。インスペクションは、いくつかのカテゴリにグループ化され、その説明で説明されています。

インスペクションは、計算されたオブジェクトセットをグループに分割することができます。グループは、ヒープウォーカーの上部にあるテーブルに表示されます。たとえば、「重複する文字列」インスペクションは、重複する文字列値をグループとして表示します。参照ビューにいる場合、選択された文字列値を持つ`java.lang.String`インスタンスを以下に表示できます。最初は、グループテーブルの最初の行が選択されています。選択を変更することで、現在のオブジェクトセットが変更されます。グループテーブルのインスタンスカウントおよびサイズ列は、行を選択したときに現在のオブジェクトセットがどれだけ大きくなるかを示します。




グループ選択は、ヒープウォーカーの別の選択ステップではありませんが、インスペクションによって行われた選択ステップの一部になります。グループ選択は、下部の選択ステップペインに表示されます。グループ選択を変更すると、選択ステップペインが即座に更新されます。

グループを作成する各インスペクションは、インスペクションのコンテキストで最も重要なグループを決定します。これは、他の列の自然なソート順と常に一致するわけではないため、グループテーブルの優先度列には、インスペクションのソート順を強制する数値が含まれています。

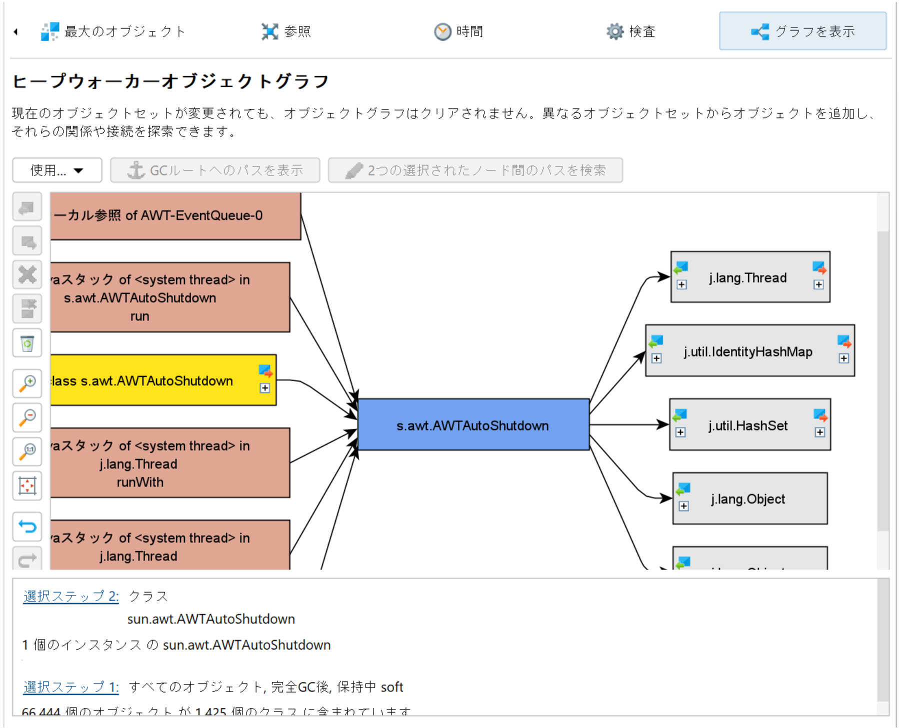
インスペクションは、大きなヒープに対して計算するのに高価な場合があるため、結果はキャッシュされます。このようにして、履歴に戻って、以前に計算されたインスペクションの結果を待たずに確認できます。

## ヒープウォーカーグラフ

インスタンスとその参照を一緒に最も現実的に表現するのはグラフです。グラフは視覚的な密度が低く、一部のタイプの分析には不向きですが、それでもオブジェクト間の関係を視覚化する最良の方法です。たとえば、循環参照はツリーでは解釈が難しいですが、グラフではすぐに明らかです。また、ツリーストラクチャではどちらか一方しか表示できないため、インカミングおよびアウトゴーイング参照を一緒に表示することが有益な場合があります。

ヒープウォーカーグラフは、現在のオブジェクトセットから自動的にオブジェクトを表示することはなく、現在のオブジェクトセットを変更してもクリアされません。アウトゴーイング参照ビュー、インカミング参照ビュー、または最大オブジェクトビューから選択したオブジェクトを手動でグラフに追加し、1つ以上のインスタンスを選択してグラフに表示アクションを使用します。

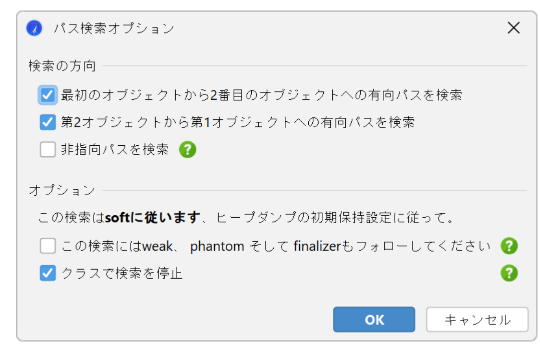
グラフ内のパッケージ名はデフォルトで短縮されています。CPUコールグラフと同様に、ビュー設定で完全な表示を有効にできます。参照は矢印として描かれます。参照上にマウスを移動すると、特定の参照の詳細を示すツールチップウィンドウが表示されます。参照ビューから手動で追加されたインスタンスは青い背景を持っています。インスタンスが追加された時期が新しいほど、背景色は暗くなります。ガベージコレクタールートは赤い背景を持ち、クラスは黄色い背景を持ちます。




デフォルトでは、参照グラフは現在のインスタンスの直接のインカミングおよびアウトゴーイング参照のみを表示します。任意のオブジェクトをダブルクリックすることでグラフを展開できます。これにより、そのオブジェクトの直接のインカミングまたは直接のアウトゴーイング参照が展開されます。インスタンスの左側と右側にある展開コントロールを使用して、インカミングおよびアウトゴーイング参照を選択的に開くことができます。バックトラックする必要がある場合は、グラフの以前の状態を復元するために元に戻す機能を使用して、ノードが多すぎて気が散らないようにします。グラフをトリミングするには、すべての接続されていないノードを削除するアクションや、すべてのオブジェクトを削除するアクションがあります。

インカミング参照ビューと同様に、グラフにはGCルートへのパスを表示ボタンがあり、利用可能な場合は1つ以上の参照チェーンをガベージコレクタールート[p.216]に展開します。さらに、2つのインスタンスが選択されている場合にアクティブになる2つの選択されたノード間のパスを見つける

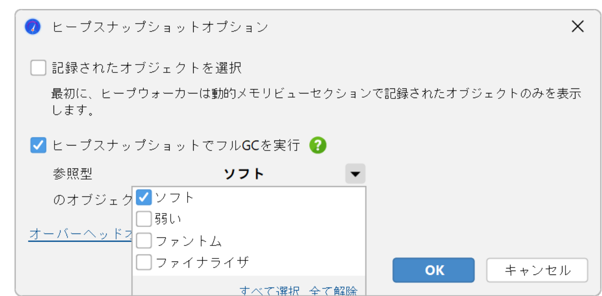
アクションがあります。これは、弱い参照に沿っても、指向性および非指向性のパスを検索できます。適切なパスが見つかると、それは赤で表示されます。




### 初期オブジェクトセット

ヒープスナップショットを取得する際に、初期オブジェクトセットを制御するオプションを指定できます。割り当てを記録している場合、記録されたオブジェクトを選択チェックボックスは、最初に表示されるオブジェクトを記録されたものに制限します。通常、これらの数値はライブメモリビューのものとは異なります。なぜなら、ヒープウォーカーによって参照されていないオブジェクトが削除されるからです。未記録のオブジェクトはヒープスナップショットにまだ存在していますが、初期オブジェクトセットには表示されません。さらに選択ステップを進めることで、未記録のオブジェクトに到達できます。

さらに、ヒープウォーカーはガベージコレクションを実行し、ソフト参照を除いて弱参照オブジェクトを削除します。これは通常、メモリリークを探している場合に望ましいです。なぜなら、弱参照オブジェクトは気をそらす可能性があり、強参照オブジェクトのみが関連しているからです。ただし、弱参照オブジェクトに興味がある場合は、ヒープウォーカーにそれらを保持させることができます。JVMの4つの弱参照タイプは「ソフト」、「弱」、「ファントム」、「ファイナライザ」であり、ヒープスナップショットでオブジェクトを保持するのに十分なものを選択できます。

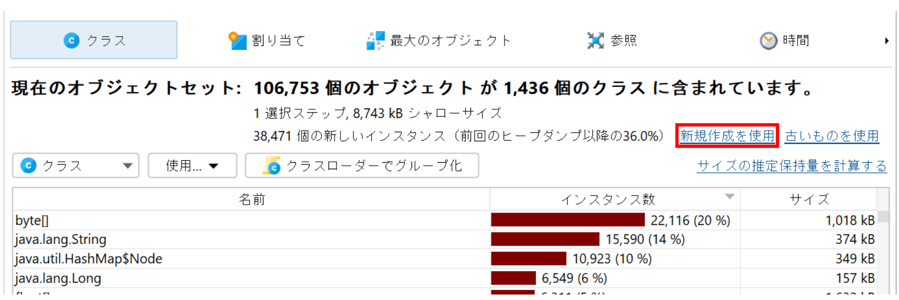



存在する場合、弱参照オブジェクトはヒープウォーカーの「弱参照」インスペクションを使用して現在のオブジェクトセットから選択または削除できます。

### ヒープのマーク

特定のユースケースのために割り当てられたオブジェクトを確認したい場合があります。割り当て記録をそのユースケースの周りで開始および停止することでこれを行うこともできますが、オーバーヘッドが少なく、他の目的のために割り当て記録機能を保持するはるかに優れた方法があります：ヒープウォーカーの概要で宣伝されているヒープをマークアクションであり、プロファイリングメニューやトリガーアクションとしても利用可能です。これにより、ヒープ上のすべてのオブジェクトが「古い」としてマークされます。次のヒープスナップショットを取得すると、「新しい」オブジェクトが何であるべきかが明確になります。

以前のヒープスナップショットまたはヒープマークの呼び出しがあった場合、ヒープウォーカーのタイトルエリアには新しいインスタンスカウントと、新しいものを使用および古いものを使用というタイトルの2つのリンクが表示されます。これにより、その時点から割り当てられたインスタンス、またはそれ以前に割り当てられた生存インスタンスのいずれかを選択できます。この情報は各オブジェクトセットに対して利用可能であり、最初に掘り下げてから後で新しいまたは古いインスタンスを選択できます。



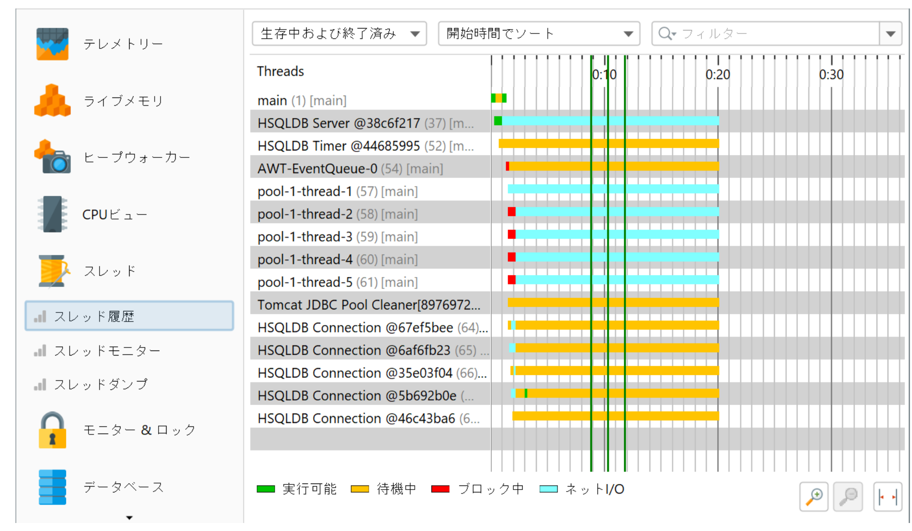
# スレッドプロファイリング

スレッドを誤って使用すると、さまざまな種類の問題が発生する可能性があります。アクティブなスレッドが多すぎるとスレッド飢餓が発生し、スレッドが互いにブロックし合ってアプリケーションの生存性に影響を与えたり、ロックを誤った順序で取得するとデッドロックが発生する可能性があります。さらに、スレッドに関する情報はデバッグ目的で重要です。

JProfilerでは、スレッドプロファイリングが2つのビューセクションに分かれています。「スレッド」セクションはスレッドのライフサイクルとスレッドダンプのキャプチャを扱います。「モニター＆ロック」セクションは、複数のスレッドの相互作用を分析するための機能を提供します。

## スレッドの検査

スレッド履歴ビューでは、各スレッドがタイムラインの中で色付きの行として表示され、色は記録されたスレッドの状態を示します。スレッドは作成時間、名前、またはスレッドグループでソートされ、名前でフィルタリングすることができます。また、ドラッグ＆ドロップでスレッドの順序を自分で再配置することもできます。モニターイベントが記録されている場合、「待機中」または「ブロック中」の状態にあるスレッドの部分にカーソルを合わせると、関連するスタックトレースが表示され、モニター履歴ビューへのリンクが表示されます。

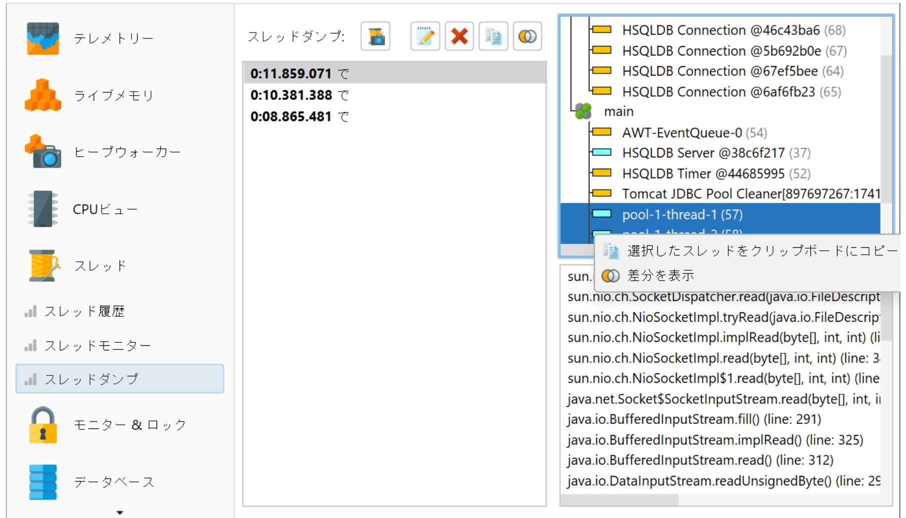



すべてのスレッドの表形式のビューはスレッドモニタービューで利用可能です。スレッドが作成されている間にCPU記録がアクティブである場合、JProfilerは作成スレッドの名前を保存し、テーブ

ルに表示します。下部には、作成スレッドのスタックトレースが表示されます。パフォーマンス上の理由から、JVMから実際のスタックトレースは要求されず、CPU記録からの現在の情報が使用されます。これは、スタックトレースが呼び出しツリー収集のフィルタ設定を満たすクラスのみを表示することを意味します。

プロファイリング設定で推定CPU時間の記録を有効にすると、CPU Time列がテーブルに追加されます。CPU時間は、CPUデータを記録しているときにのみ測定されます。

ほとんどのデバッガーと同様に、JProfilerもスレッドダンプを取得できます。スレッドダンプのスタックトレースは、JVMによって提供される完全なスタックトレースであり、CPU記録に依存しません。異なるスレッドダンプは、2つのスレッドダンプを選択してShow Differenceボタンをクリックすると、差分ビューアで比較できます。また、単一のスレッドダンプから2つのスレッドを選択して、コンテキストメニューからShow Differenceを選択することで比較することも可能です。

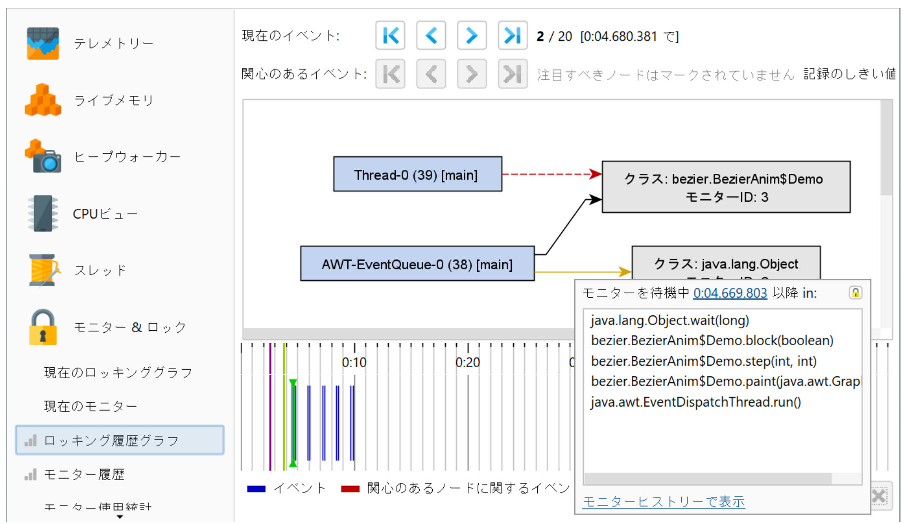



スレッドダンプは、「Triggerthreaddump」トリガーアクションまたはAPIを介しても取得できます。

### ロック状況の分析

すべてのJavaオブジェクトには、2つの同期操作に使用できる関連するモニターがあります。スレッドは、他のスレッドが通知を発行するまでモニターで待機するか、モニターのロックを取得し、他のスレッドがロックの所有権を放棄するまでブロックすることができます。さらに、Javaは、より高度なロック戦略を実装するための`java.util.concurrent.locks`パッケージ内のクラスを提供します。このパッケージ内のロックは、オブジェクトのモニターを使用せず、異なるネイティブ実装を使用します。

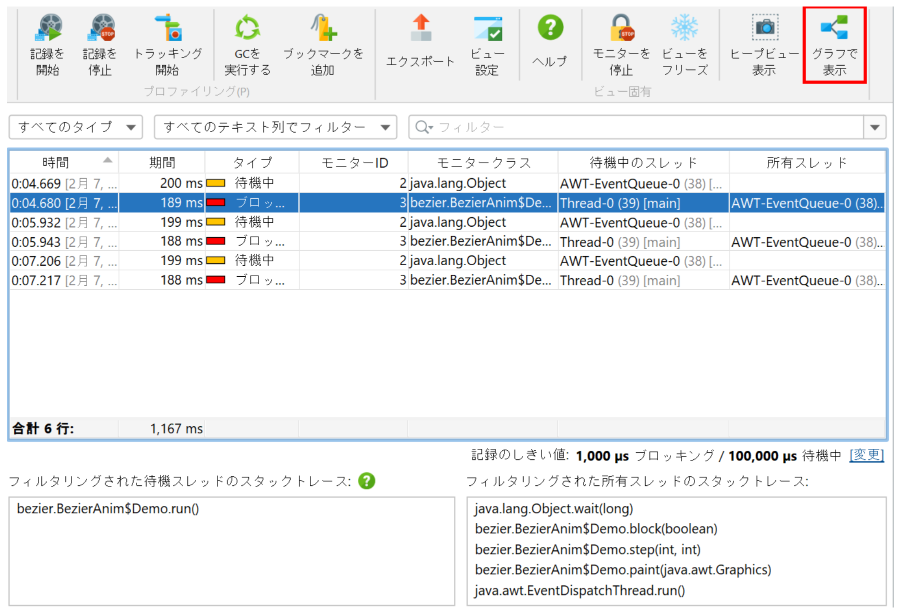
JProfilerは、上記の両方のメカニズムに対してロック状況を記録できます。ロック状況では、1つまたは複数のスレッド、モニター、または`java.util.concurrent.locks.Lock`のインスタンス、および一定の時間を要する待機またはブロック操作があります。これらのロック状況は、モニター履歴ビューで表形式で表示され、ロック履歴グラフで視覚的に表示されます。




ロック履歴グラフは、孤立したモニターイベントの期間よりも、関与するすべてのモニターとスレッドの関係全体に焦点を当てています。ロック状況に参加しているスレッドとモニターは青と灰色の長方形として描かれ、デッドロックの一部である場合は赤で描かれます。黒い矢印はモニター

の所有権を示し、黄色の矢印は待機中のスレッドから関連するモニターに伸び、破線の赤い矢印はスレッドがモニターを取得しようとして現在ブロックしていることを示します。CPUデータが記録されている場合、ブロックまたは待機中の矢印にカーソルを合わせるとスタックトレースが表示されます。これらのツールチップには、モニター履歴ビューの対応する行に移動するハイパーリンクが含まれています。

表形式のモニター履歴ビューはモニターイベントを表示します。これらには列として表示される期間があり、テーブルをソートすることで最も重要なイベントを見つけることができます。表形式のビューで選択された行に対して、Show in Graphアクションを使用してグラフにジャンプできます。



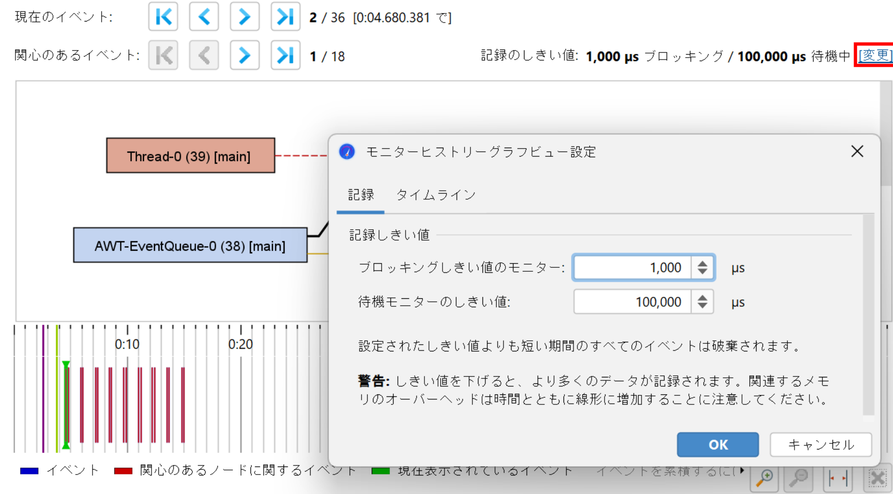

各モニターイベントには関連するモニターがあります。Monitor Class列には、モニターが使用されているインスタンスのクラス名が表示され、Javaオブジェクトがモニターに関連付けられていない場合は「[raw monitor]」と表示されます。いずれの場合も、モニターには一意のIDがあり、別の列に表示されるため、複数のイベントにわたる同じモニターの使用を関連付けることができます。各モニターイベントには、操作を実行している待機スレッドと、操作をブロックしている所有スレッドがオプションであります。利用可能な場合、それらのスタックトレースはビューの下部に表示されます。

モニターインスタンスについてさらに質問がある場合は、モニター履歴ビューとロック履歴グラフの両方でShow in Heap Walkerアクションを使用すると、ヒープウォーカーへのリンクが提供され、モニターインスタンスが新しいオブジェクトセットとして選択されます。

### 関心のあるイベントの制限

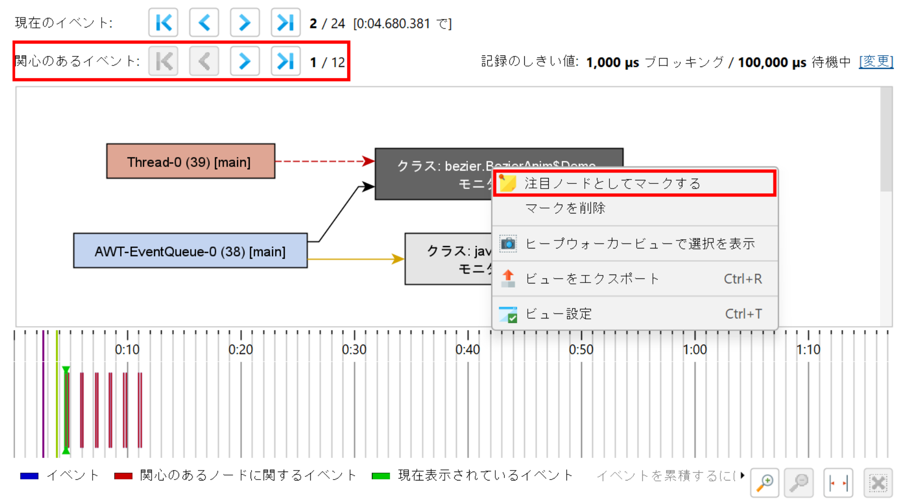
モニターイベントを分析する際の基本的な問題の1つは、アプリケーションが非常に高い頻度でモニターイベントを生成する可能性があることです。そのため、JProfilerには待機およびブロックイベントのデフォルトのしきい値があり、これを下回るイベントは即座に破棄されます。これらのし

きい値はビュー設定で定義されており、より長いイベントに焦点を当てるために増やすことができます。




記録されたイベントには、さらにフィルタを適用することができます。モニター履歴ビューでは、しきい値、イベントタイプ、およびテキストフィルタがビューの上部に提供されています。ロック履歴グラフでは、関心のあるスレッドまたはモニターを選択し、マークされたエンティティを含むロック状況のみを表示することができます。関心のあるイベントはタイムラインで異なる色で表示され、これらのイベントを通過するための二次ナビゲーションバーがあります。現在のイベントが関心のあるイベントでない場合、現在のイベントと次の関心のあるイベントの間にどれだけのイベントがあるかを確認できます。

選択したスレッドまたはモニターが存在するロック状況に加えて、グラフから削除されたロック状況も表示されます。これは、各モニターイベントが、操作が開始されたロック状況と終了したロック状況の2つによって定義されるためです。完全に空のグラフも、JVMにロックがないことを示す有効なロック状況です。




注意が必要なイベントの数を減らすための別の戦略は、ロック状況を累積することです。ロック履歴グラフでは、下部にすべての記録されたイベントを示すタイムラインがあります。これをクリックしてドラッグすると、時間範囲が選択され、含まれるすべてのイベントのデータが上のロックグラフに表示されます。累積グラフでは、各矢印に同じタイプの複数のイベントが含まれることがあります。その場合、ツールチップウィンドウにはイベントの数と含まれるすべてのイベントの合計

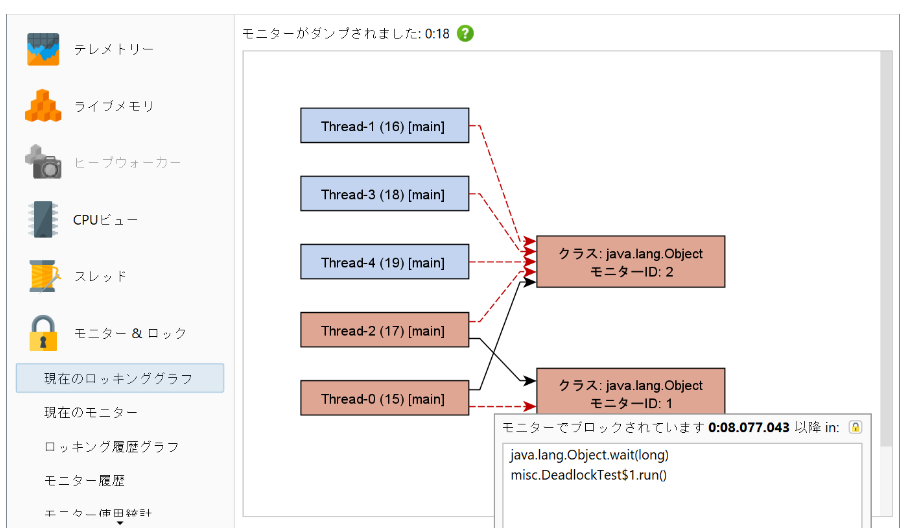
時間が表示されます。ツールチップウィンドウのドロップダウンリストにはタイムスタンプが表示され、異なるイベント間を切り替えることができます。

### デッドロック検出

「現在のロックグラフ」と「現在のモニター」ビューは、JProfiler UIのアクションでトリガーされる「モニターダンプ」で動作します。モニターダンプを使用すると、進行中のイベントを検査できます。これには、決して終了しないイベントであるデッドロックが含まれ、履歴ビューには表示されません。

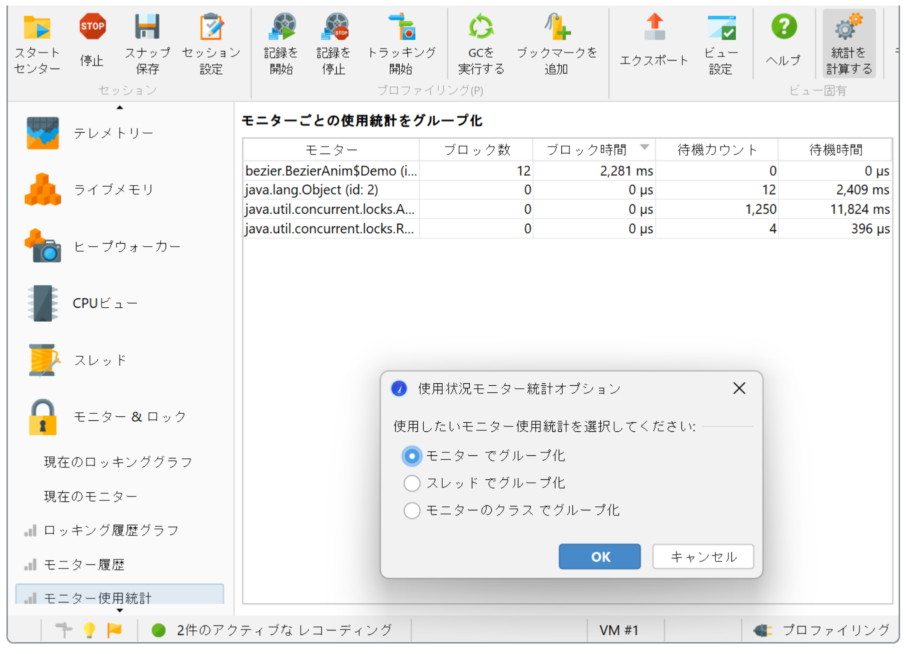
ブロック操作は通常短命ですが、デッドロックが発生した場合、両方のビューは問題の永続的なビューを表示します。さらに、現在のロックグラフは、デッドロックを引き起こすスレッドとモニターを赤で表示するため、そのような問題をすぐに見つけることができます。

新しいモニターダンプを取得すると、両方のビューのデータが置き換えられます。モニターダンプは、「Trigger monitor dump」トリガーアクションまたはAPIを介してもトリガーできます。




### モニター使用統計

より高い視点からブロックおよび待機操作を調査するために、モニター統計ビューはモニター記録データからレポートを計算します。モニターイベントをモニター、スレッド名、またはモニターのクラスでグループ化し、各行の累積カウントと期間を分析できます。


スタート
センター
停止
スナップ
保存
セッション
設定
セッション
記録を
開始
記録を
停止
トラッキング
開始
GCを
実行する
ブックマークを
追加
プロファイリング(P)
エクスポート
ビュー
設定
ヘルプ
統計を
計算する
ビュー固有
テレメトリー
ライブメモリ
ヒープウォーカー
CPUビュー
スレッド
モニター & ロック
現在のロッキンググラフ
現在のモニター
ロッキング履歴グラフ
モニター履歴
モニター使用統計
モニターごとの使用統計をグループ化
モニター
ブロック数
ブロック時間
待機カウント
待機時間
bezier.BezierAnim$Demo (i...
12
2,281 ms
0
0 µs
java.lang.Object (id: 2)
0
0 µs
12
2,409 ms
java.util.concurrent.locks.A...
0
0 µs
1,250
11,824 ms
java.util.concurrent.locks.R...
0
0 µs
4
396 µs
使用状況モニター統計オプション
使用したいモニター使用統計を選択してください:
モニター でグループ化
スレッド でグループ化
モニターのクラス でグループ化
OK
キャンセル
2件のアクティブな レコーディング
VM #1
プロファイリング

# プローブ

CPUとメモリのプロファイリングは主にオブジェクトとメソッドコールに関心があり、JVM上のアプリケーションの基本的な構成要素です。いくつかの技術では、実行中のアプリケーションからセマンティックデータを抽出し、プロファイラに表示するより高レベルのアプローチが必要です。

これの最も顕著な例は、JDBCを使用したデータベースへのコールのプロファイリングです。呼び出しツリーは、JDBCAPIを使用したときとそのコールがどれくらいの時間を要するかを示します。しかし、異なるSQLステートメントが各コールで実行される可能性があり、どのコールがパフォーマンスのボトルネックの原因であるかはわかりません。また、JDBCコールはアプリケーションの多くの異なる場所から発生することが多く、一般的な呼び出しツリーを検索する代わりに、すべてのデータベースコールを表示する単一のビューを持つことが重要です。

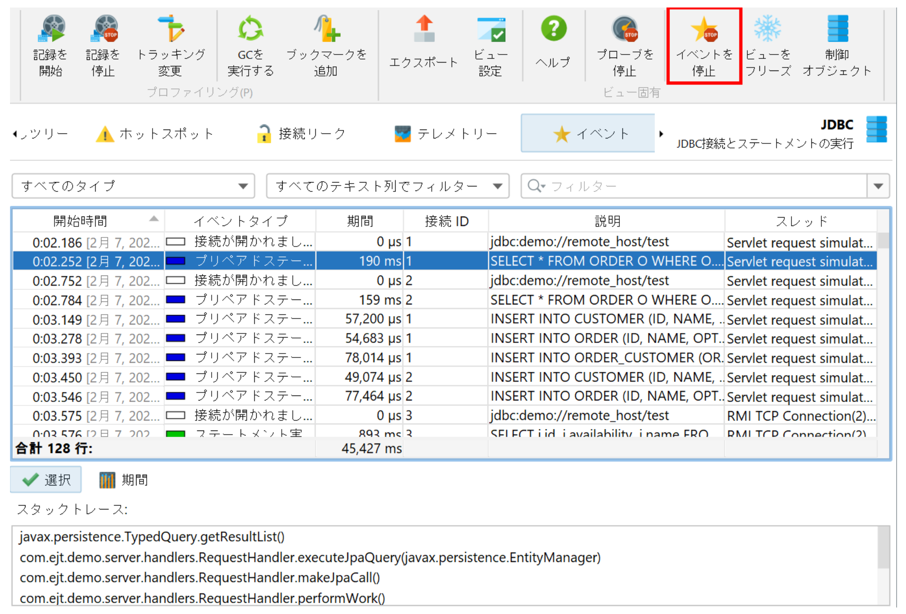
この問題を解決するために、JProfilerはJREの重要なサブシステムのためのいくつかのプローブを提供します。プローブは特定のクラスに計測を追加してデータを収集し、「Databases」および「JEE＆Probes」ビューセクションに専用のビューで表示します。さらに、プローブは呼び出しツリーにデータを注釈付けすることができ、一般的なCPUプロファイリングと高レベルのデータを同時に見ることができます。

JProfilerが直接サポートしていない技術についての詳細情報を取得したい場合は、そのための独自のプローブ[p.163]を書くことができます。いくつかのライブラリ、コンテナ、またはデータベースドライバは、アプリケーションで使用されるとJProfilerに表示される埋め込みプローブ[p.168]を提供することがあります。

### プローブイベント

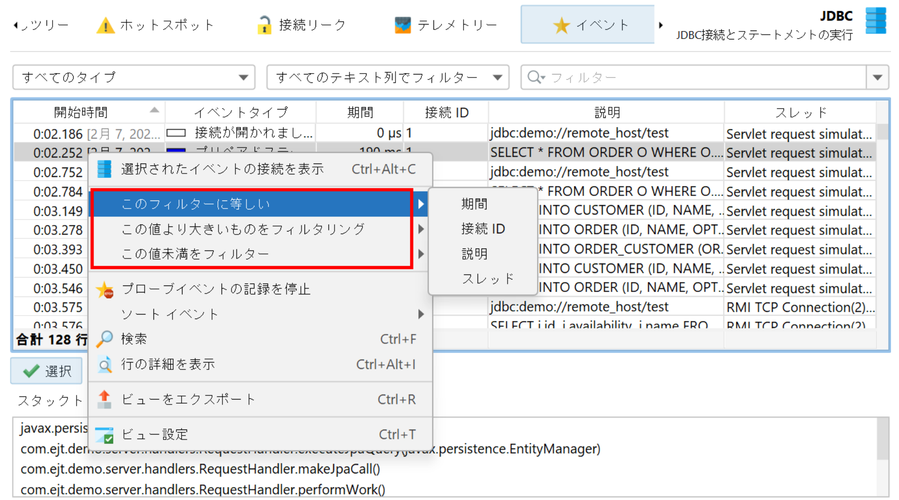
プローブはオーバーヘッドを追加するため、デフォルトでは記録されませんが、各プローブの記録を開始[p. 28]する必要があります。手動または自動で行うことができます。

プローブの能力に応じて、プローブデータは複数のビューに表示されます。最も低いレベルではプローブイベントがあります。他のビューはプローブイベントを累積したデータを表示します。デフォルトでは、プローブイベントはプローブが記録されている場合でも保持されません。単一のイベントが重要になる場合、プローブイベントビューでそれらを記録することができます。ファイルプローブのような一部のプローブでは、通常高いレートでイベントを生成するため、これは一般的に推奨されません。HTTPサーバープローブやJDBCプローブのような他のプローブは、はるかに低いレートでイベントを生成する可能性があるため、単一のイベントを記録することが適切な場合があります。



プローブイベントは、メソッドパラメータ、戻り値、計測されたオブジェクト、スローされた例外など、さまざまなソースからプローブ文字列をキャプチャします。プローブは複数のメソッドコールからデータを収集することができます。たとえば、JDBCプローブは、実際のSQL文字列を構築するために準備されたステートメントのすべてのセッターコールをインターセプトする必要があります。プローブ文字列は、プローブによって測定される高レベルのサブシステムに関する基本情報です。さらに、イベントには開始時間、オプションの期間、関連するスレッド、およびスタックトレースが含まれます。

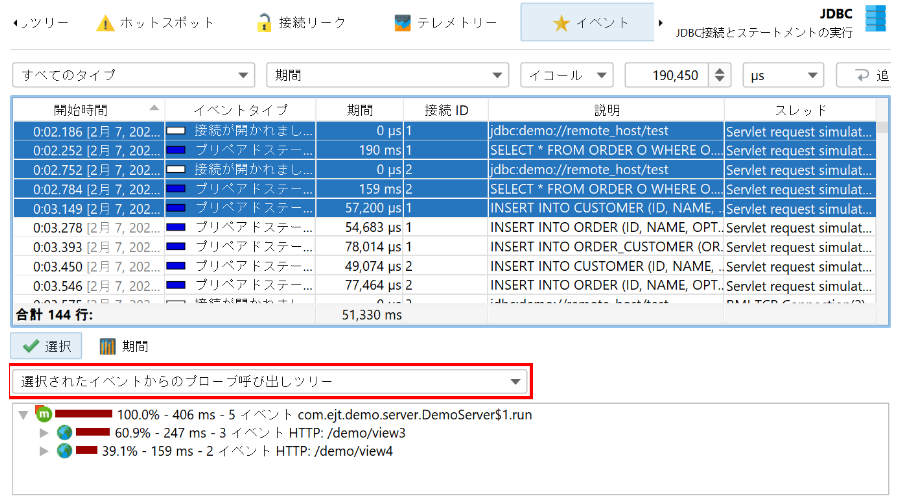
テーブルの下部には、表示されたイベントの総数を示し、テーブル内のすべての数値列を合計する特別な行があります。デフォルトの列では、これはDuration列のみを含みます。テーブルの上のフィルタセレクタと共に、選択されたイベントのサブセットの収集データを分析することができます。デフォルトでは、テキストフィルタはすべてのテキストフィールド列で機能しますが、テキストフィールドの前のドロップダウンから特定のフィルタ列を選択できます。フィルタオプションはコンテキストメニューからも利用可能で、たとえば、選択されたイベントの期間よりも長いすべてのイベントをフィルタリングすることができます。



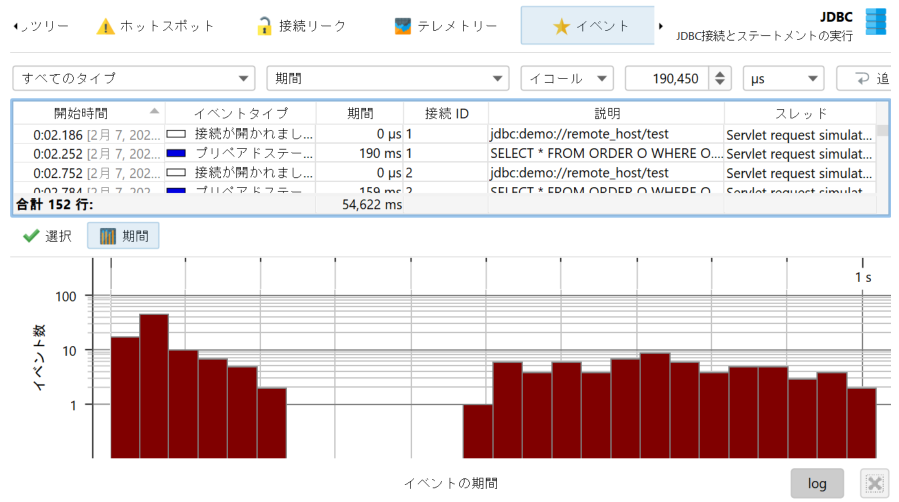
他のプローブビューもプローブイベントをフィルタリングするオプションを提供します。プローブテレメトリービューでは時間範囲を選択でき、プローブ呼び出しツリービューでは選択された呼び

出しスタックからイベントをフィルタリングできます。プローブホットスポットビューは、選択されたバックトレースまたはホットスポットに基づいてプローブイベントフィルタを提供し、制御オブジェクトとタイムラインビューは選択された制御オブジェクトのプローブイベントをフィルタリングするアクションを提供します。

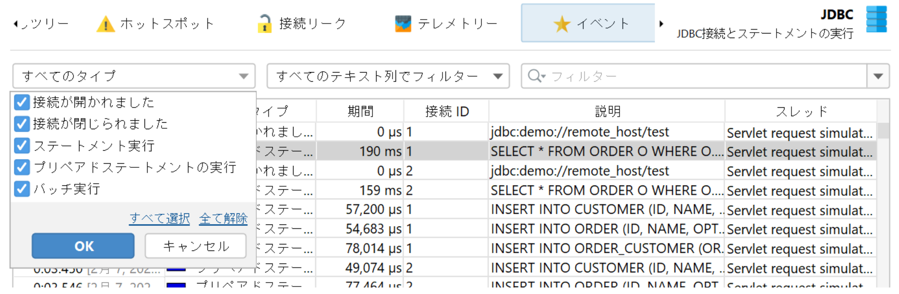
選択されたプローブイベントのスタックトレースは下部に表示されます。複数のプローブイベントが選択されている場合、スタックトレースは累積され、呼び出しツリー、バックトレース付きプローブホットスポット、またはバックトレース付きCPUホットスポットとして表示されます。




スタックトレースビューの横には、イベント期間のヒストグラムビューと、オプションで記録されたスループットが表示されます。これらのヒストグラムでマウスを使用して期間範囲を選択し、上のテーブルでプローブイベントをフィルタリングすることができます。



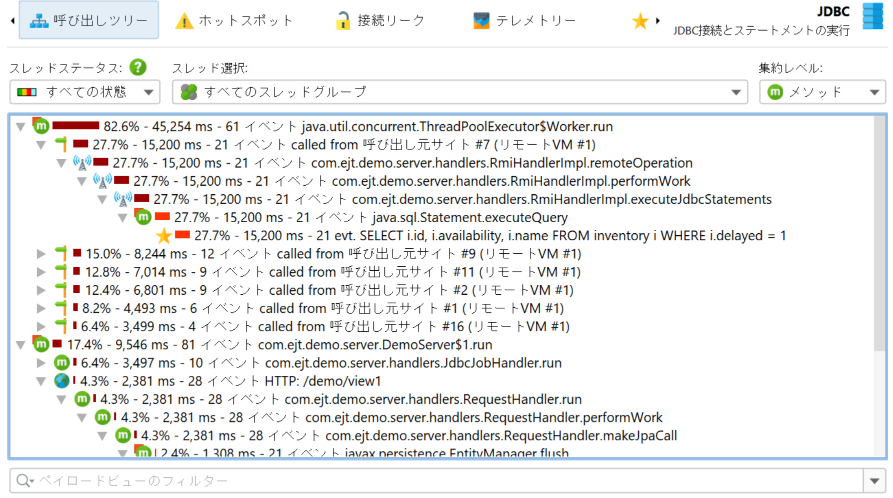

プローブはさまざまな種類のアクティビティを記録し、プローブイベントにイベントタイプを関連付けることができます。たとえば、JDBCプローブは、ステートメント、準備されたステートメント、およびバッチ実行を異なる色のイベントタイプとして表示します。




単一のイベントが記録されるときの過剰なメモリ使用量を防ぐために、JProfilerはイベントを統合します。イベントキャップはプロファイリング設定で設定され、すべてのプローブに適用されます。最新のイベントのみが保持され、古いイベントは破棄されます。この統合は高レベルのビューには影響しません。

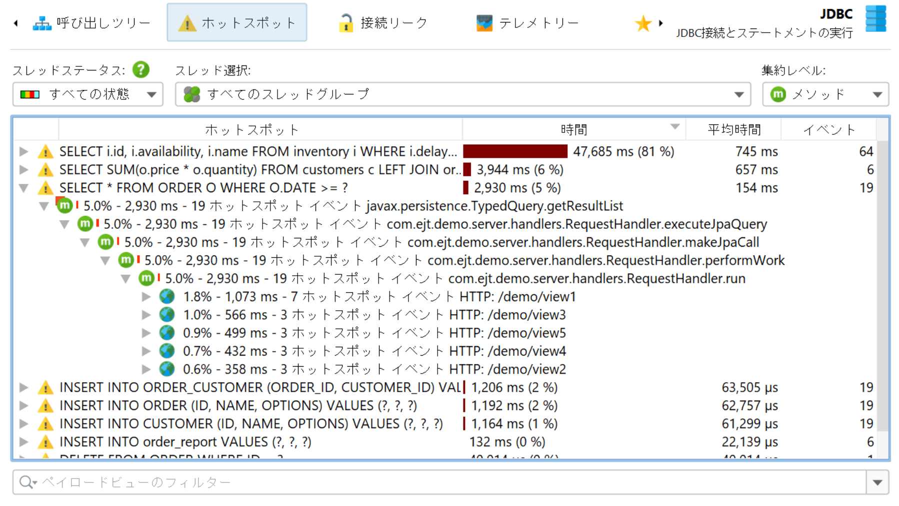
### プローブ呼び出しツリーとホットスポット

プローブ記録はCPU記録と密接に連携しています。プローブイベントはプローブ呼び出しツリーに集約され、プローブ文字列はリーフノード（ペイロードと呼ばれる）になります。プローブイベントが作成された呼び出しスタックのみがそのツリーに含まれます。メソッドノードの情報は記録されたペイロード名を参照します。たとえば、特定の呼び出しスタックでSQLステートメントが42回実行され、合計時間が9000 msであった場合、これはすべての祖先呼び出しツリーノードに42のイベントカウントと9000 msの時間を追加します。記録されたすべてのペイロードの累積は、プローブ固有の時間を最も消費する呼び出しパスを示す呼び出しツリーを形成します。プローブツリーの焦点はペイロードであるため、ビューのフィルタはデフォルトでペイロードを検索しますが、そのコンテキストメニューにはクラスをフィルタリングするモードも提供されています。




CPU記録がオフになっている場合、バックトレースには「No CPU data was recorded」ノードのみが含まれます。CPUデータが部分的にしか記録されていない場合、これらのノードと実際のバックトレースが混在している可能性があります。サンプリングが有効になっている場合でも、JProfilerはデフォルトでプローブペイロードの正確な呼び出しトレースを記録します。このオーバーヘッドを避けたい場合は、プロファイリング設定でオフにすることができます。プローブ記録のデータ収集を増やしたりオーバーヘッドを減らしたりするために調整できる他のいくつかのチューニングオプションがあります。

ホットスポットはプローブ呼び出しツリーから計算できます。ホットスポットノードは、CPUビューセクション [p. 54]のようにメソッドコールではなく、ペイロードになります。これはプローブの最も即時に役立つビューであることがよくあります。CPU記録がアクティブな場合、トップレベルのホットスポットを開いてメソッドバックトレースを分析することができます。通常のCPUホットスポットビューと同様です。バックトレースノードの数字は、ホットスポットの直下のノードから最も深いノードまでの呼び出しスタックに沿って測定されたプローブイベントの数と合計時間を示します。




プローブ呼び出しツリーとプローブホットスポットビューの両方で、スレッドまたはスレッドグループ、スレッドステータス、およびメソッドノードの集約レベルを選択できます。対応するCPUビューと同様です。CPUビューからデータを比較するために来るとき、プローブビューのデフォルトのスレッドステータスが「すべての状態」であり、CPUビューのように「実行可能」ではないことを覚えておくことが重要です。これは、プローブイベントがデータベースコール、ソケット操

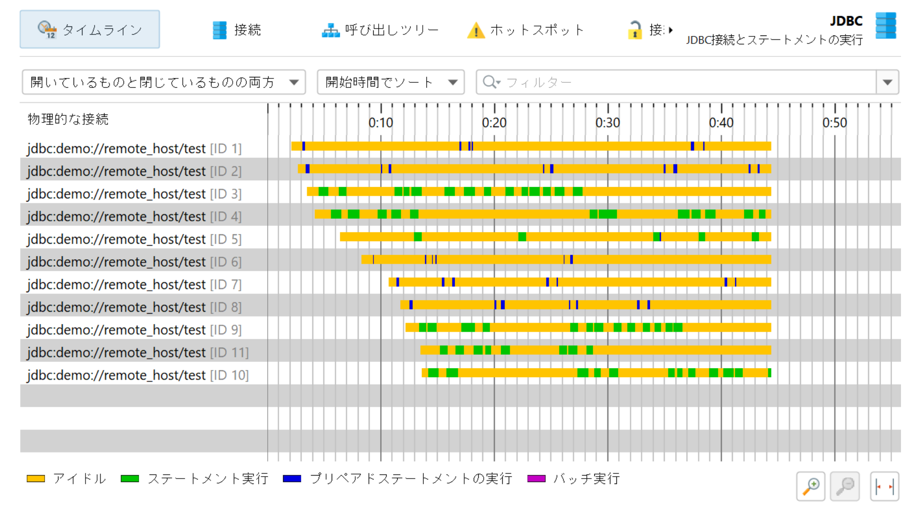
作、プロセス実行などの外部システムを含むことが多く、現在のJVMがそれに取り組んでいる時間だけでなく、合計時間を見ることが重要だからです。

**制御オブジェクト**

外部リソースへのアクセスを提供する多くのライブラリは、リソースと対話するために使用できる接続オブジェクトを提供します。たとえば、プロセスを開始すると、`java.lang.Process`オブジェクトを使用して出力ストリームから読み取り、入力ストリームに書き込むことができます。JDBCを使用する場合、SQLクエリを実行するには`java.sql.Connection`オブジェクトが必要です。この種のオブジェクトに対してJProfilerで使用される一般的な用語は「制御オブジェクト」です。

プローブイベントを制御オブジェクトとグループ化し、そのライフサイクルを表示することで、問題の発生源をよりよく理解するのに役立ちます。また、制御オブジェクトの作成はしばしば高価であるため、アプリケーションがそれらを多く作成しすぎず、適切に閉じることを確認したいです。この目的のために、制御オブジェクトをサポートするプローブには、「タイムライン」と「制御オブジェクト」ビューがあり、後者はより具体的に命名されることがあります。たとえば、JDBCプローブでは「Connections」となります。制御オブジェクトが開かれたり閉じられたりすると、プローブはイベントビューに表示される特別なプローブイベントを作成し、関連するスタックトレースを調査することができます。

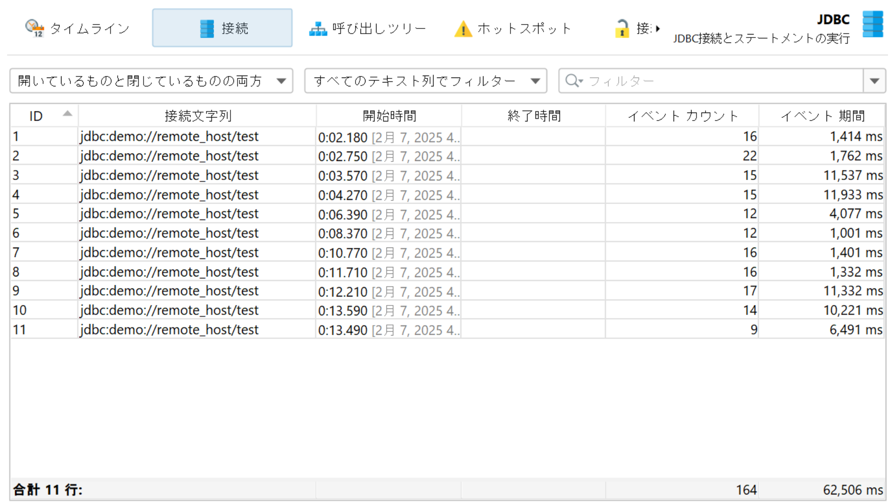
タイムラインビューでは、各制御オブジェクトがバーとして表示され、その着色は制御オブジェクトがアクティブであったときの状態を示します。プローブは異なるイベントタイプを記録でき、タイムラインはそれに応じて色分けされます。このステータス情報は、イベントのリストから取得されるのではなく、統合されていないか、利用できない場合もありますが、最後のステータスから100 msごとにサンプリングされます。制御オブジェクトには、それらを識別するための名前があります。たとえば、ファイルプローブはファイル名で制御オブジェクトを作成し、JDBCプローブは接続文字列を制御オブジェクトの名前として表示します。




制御オブジェクトビューは、すべての制御オブジェクトを表形式で表示します。デフォルトでは、開いている制御オブジェクトと閉じている制御オブジェクトの両方が表示されます。上部のコントロールを使用して、開いている制御オブジェクトまたは閉じている制御オブジェクトのみを表示するように制限したり、特定の列の内容をフィルタリングしたりできます。制御オブジェクトの基本的なライフサイクルデータに加えて、テーブルは各制御オブジェクトの累積アクティビティに関するデータを表示します。たとえば、イベント数と平均イベント期間です。

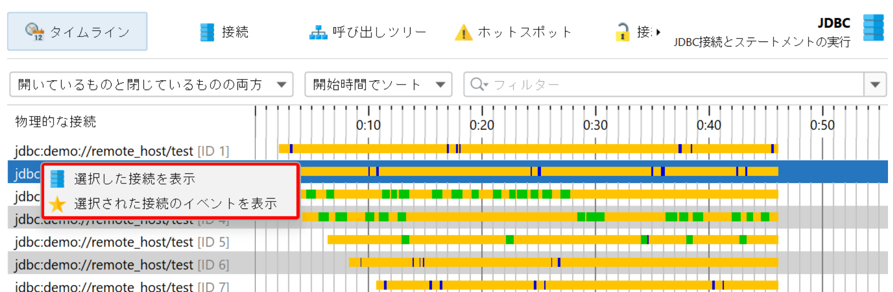
プローブによっては、ここに異なる列を表示します。たとえば、プロセスプローブは読み取りイベントと書き込みイベントのための別々の列セットを表示します。この情報は、単一イベント記録が

無効になっている場合でも利用可能です。イベントビューと同様に、下部の合計行はフィルタリングと共に使用して、制御オブジェクトの部分セットに関する累積データを取得できます。



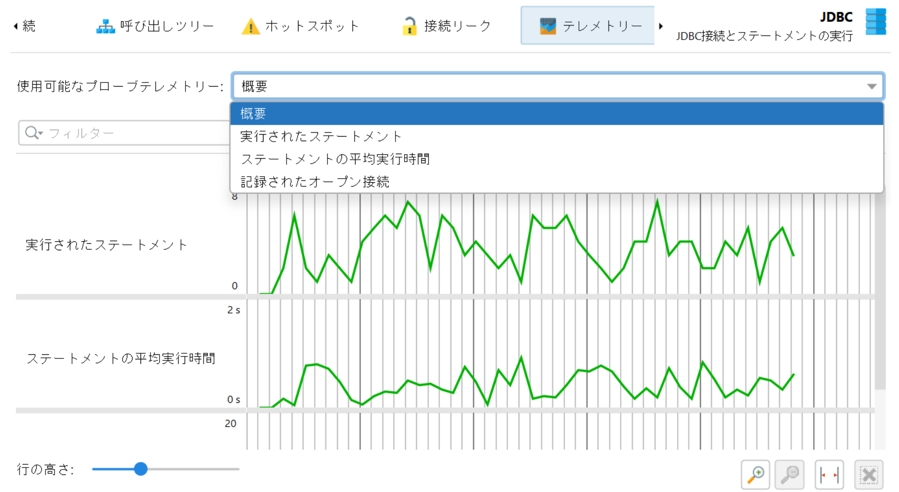

プローブは、ネストされたテーブルで特定のプロパティを公開することができます。これは、メインテーブルの情報過多を軽減し、テーブル列により多くのスペースを与えるために行われます。ネストされたテーブルが存在する場合、たとえばファイルおよびプロセスプローブの場合、各行には左側に展開ハンドルがあり、その場でプロパティ値テーブルを開きます。

タイムライン、制御オブジェクトビュー、およびイベントビューは、ナビゲーションアクションで接続されています。たとえば、タイムラインビューでは、行を右クリックして、他のビューのいずれかにジャンプし、選択された制御オブジェクトのデータのみが表示されるようにすることができます。これは、制御オブジェクトIDを選択された値にフィルタリングすることによって実現されます。

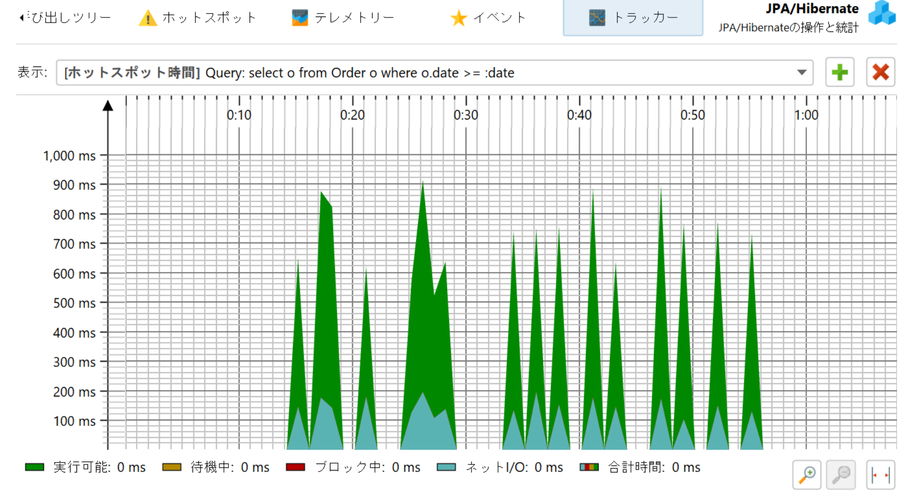



## テレメトリーとトラッカー

プローブによって収集された累積データから、いくつかのテレメトリーが記録されます。任意のプローブに対して、秒あたりのプローブイベント数と、プローブイベントの平均期間やI/O操作のスループットなどの平均測定値が利用可能です。制御オブジェクトを持つプローブの場合、開いている制御オブジェクトの数も標準的なテレメトリーです。各プローブは追加のテレメトリーを追加することができ、たとえばJPAプローブはクエリ数とエンティティ操作数のための別々のテレメトリーを表示します。



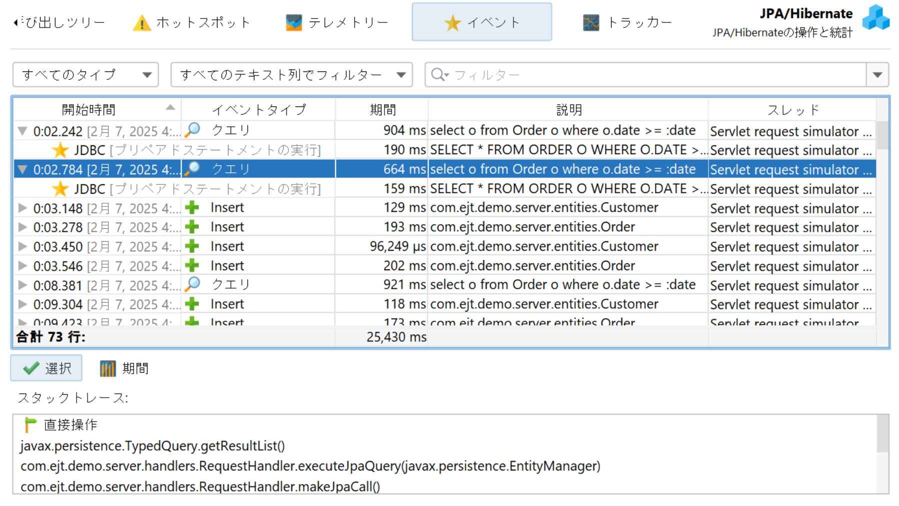

ホットスポットビューと制御オブジェクトビューは、時間の経過とともに追跡するのに興味深い累積データを表示します。これらの特別なテレメトリーはプローブトラッカーで記録されます。トラッキングを設定する最も簡単な方法は、ホットスポットまたは制御オブジェクトビューからAdd Selection to Trackerアクションを使用して新しいテレメトリーを追加することです。どちらの場合も、時間またはカウントを追跡するかどうかを選択する必要があります。制御オブジェクトを追跡する場合、テレメトリーはすべての異なるプローブイベントタイプのための積み重ねられたエリアグラフです。追跡されたホットスポットの場合、追跡された時間は異なるスレッド状態に分割されます。



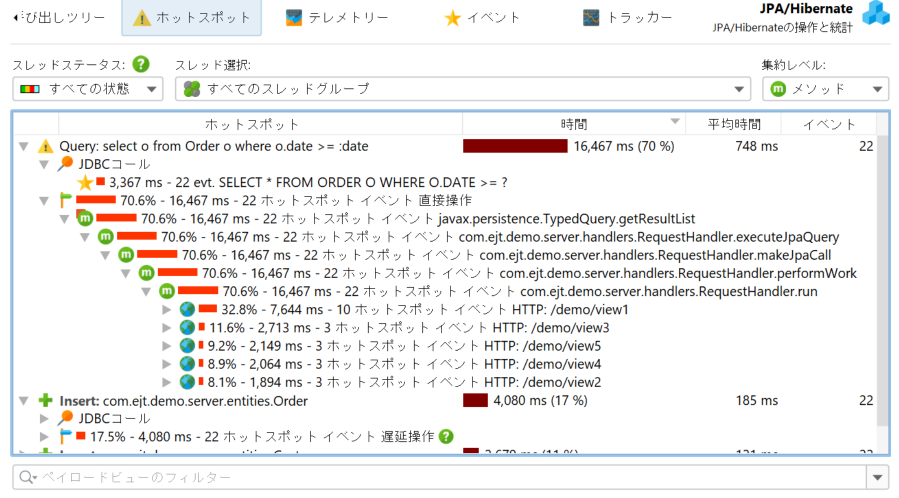

プローブテレメトリーは、"Telemetries"セクションに追加 [p. 46]して、システムテレメトリーやカスタムテレメトリーと比較することができます。その後、テレメトリー概要のコンテキストメニューアクションでプローブ記録を制御することもできます。

## JDBCとJPA

JDBCとJPAプローブは連携して動作します。JPAプローブのイベントビューでは、JPAプローブと共にJDBCプローブが記録されている場合、関連するJDBCイベントを表示するために単一のイベントを展開できます。



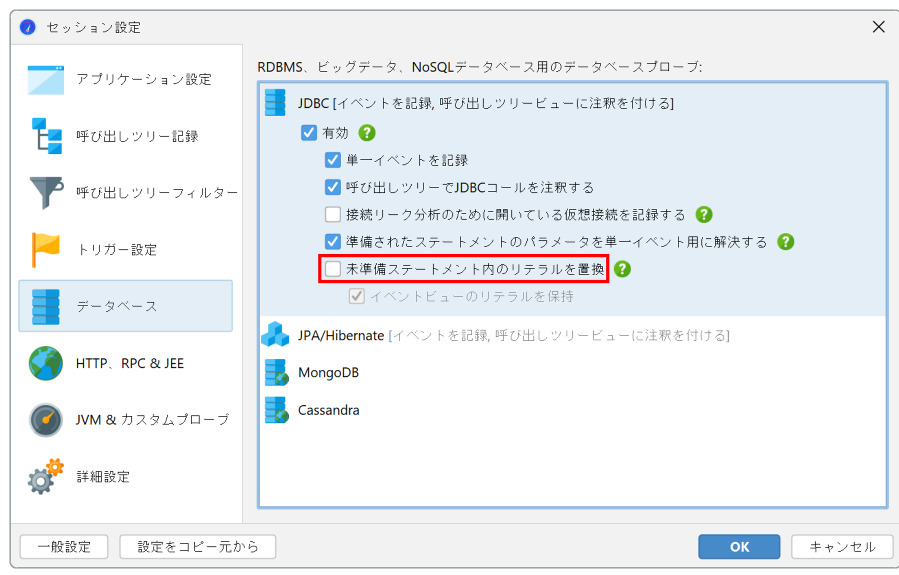

同様に、ホットスポットビューはすべてのホットスポットに「JDBC calls」ノードを追加し、JPA操作によってトリガーされたJDBCコールを含みます。一部のJPA操作は非同期であり、すぐに実行されず、セッションがフラッシュされたときに任意の後の時点で実行されます。パフォーマンスの問題を探すとき、そのフラッシュのスタックトレースは役に立たないので、JProfilerは既存のエンティティが取得された場所や新しいエンティティが永続化された場所のスタックトレースを記憶し、それらをプローブイベントに結びつけます。その場合、ホットスポットのバックトレースは「Deferred operations」とラベル付けされたノード内に含まれ、それ以外の場合は「Direct operations」ノードが挿入されます。




MongoDBプローブのような他のプローブは、直接操作と非同期操作の両方をサポートしています。非同期操作は現在のスレッドで実行されず、別の場所で、同じJVMの1つまたは複数の他のスレッドで、または別のプロセスで実行されます。そのようなプローブでは、ホットスポットのバックトレースは「Direct operations」と「Async operation」コンテナノードに分類されます。

JDBCプローブの特別な問題は、IDのようなリテラルデータがSQL文字列に含まれていない場合にのみ良いホットスポットを取得できることです。これは準備されたステートメントが使用されている場合には自動的にそうなりますが、通常のステートメントが実行される場合にはそうではありません。後者の場合、ほとんどのクエリが1回だけ実行されるホットスポットのリストを取得する可能性があります。これを解決するために、JProfilerは準備されていないステートメントのリテラルを置き換えるための非デフォルトオプションをJDBCプローブ設定で提供しています。デバッグ目的で、イベントビューでリテラルを表示したい場合があります。そのオプションを無効にすると、

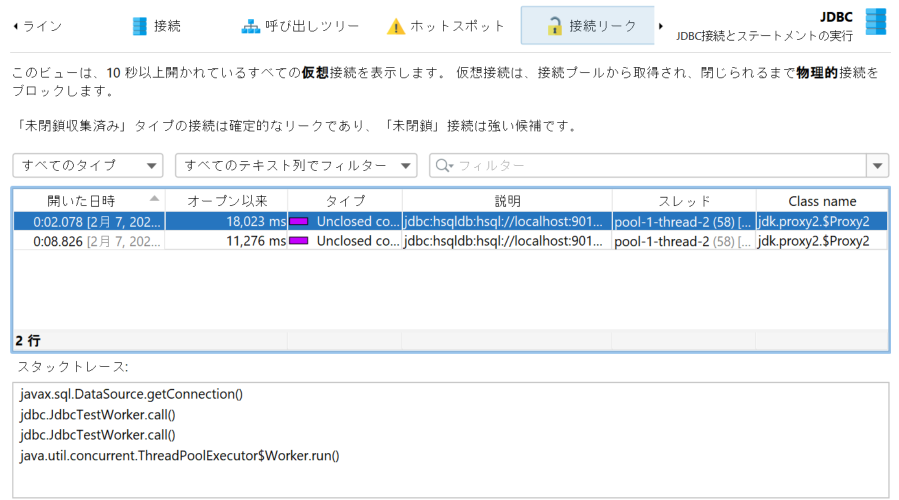
JProfilerが多くの異なる文字列をキャッシュする必要がなくなるため、メモリオーバーヘッドが減少します。




一方、JProfilerは準備されたステートメントのパラメータを収集し、イベントビューでプレースホルダーなしの完全なSQL文字列を表示します。これもデバッグ時に役立ちますが、必要ない場合は、メモリを節約するためにプローブ設定でオフにすることができます。

## JDBC接続リーク

JDBCプローブには、データベースプールに返されていない開いている仮想データベース接続を表示する「Connection leaks」ビューがあります。これは、プールされたデータベースソースによって作成された仮想接続にのみ影響します。仮想接続は、閉じられるまで物理接続をブロックします。




リーク候補には、「unclosed」接続と「unclosed collected」接続の2種類があります。どちらのタイプも、データベースプールによって提供された接続オブジェクトがまだヒープ上にあり、`close()`がそれらに対して呼び出されていない仮想接続です。「unclosed collected」接続はガベージコレクトされており、明確な接続リークです。

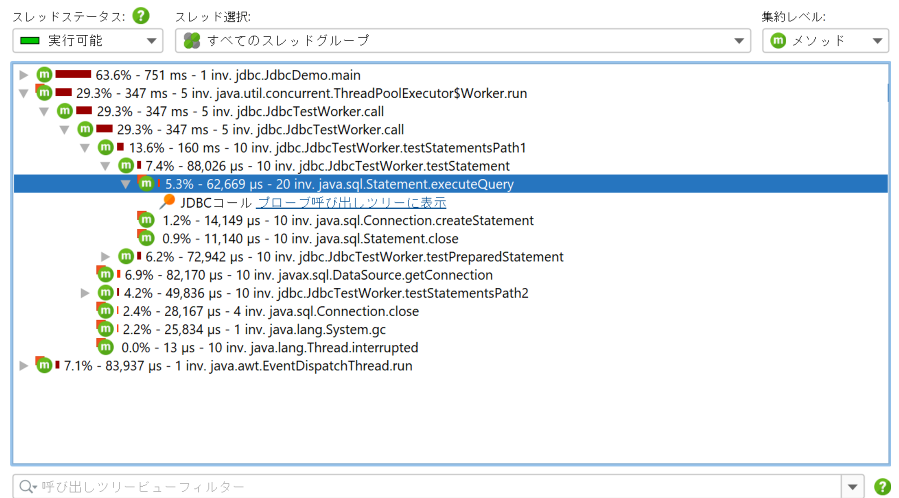
「unclosed」接続オブジェクトはまだヒープ上にあります。Open Since期間が長いほど、そのような仮想接続がリーク候補である可能性が高くなります。仮想接続は、10秒以上開いている場合に潜在的なリークと見なされます。ただし、`close()`がまだ呼び出される可能性があり、その場合、「Connection leaks」ビューのエントリは削除されます。

接続リークテーブルには、接続クラスの名前を示すClass Name列が含まれています。これにより、どのタイプのプールが接続を作成したかがわかります。JProfilerは、多くのデータベースドライバと接続プールを明示的にサポートしており、どのクラスが仮想接続であり、物理接続であるかを知っています。未知のプールやデータベースドライバの場合、JProfilerは物理接続を仮想接続と誤解する可能性があります。物理接続はしばしば長寿命であるため、「Connection leaks」ビューに表示されることがあります。この場合、接続オブジェクトのクラス名がそれを誤検知として識別するのに役立ちます。

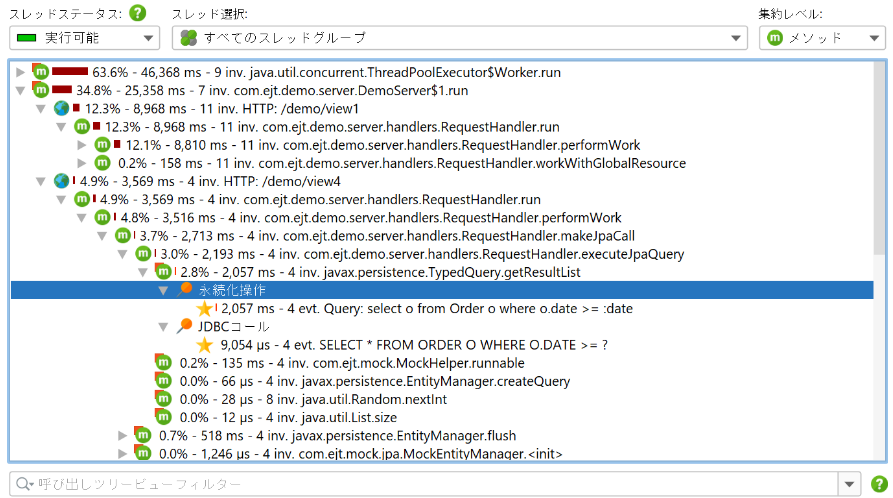
デフォルトでは、プローブ記録を開始するとき、接続リーク分析は有効になっていません。接続リークビューには、JDBCプローブ設定のRecord open virtual connections for connection leak analysisチェックボックスに対応する別の記録ボタンがあります。イベント記録と同様に、ボタンの状態は永続的であるため、一度分析を開始すると、次のプローブ記録セッションでも自動的に開始されます。

## 呼び出しツリー内のペイロードデータ

CPU呼び出しツリーを見ているとき、プローブがペイロードデータを記録した場所を見ることは興味深いです。そのデータは、測定されたCPU時間を解釈するのに役立つかもしれません。そのため、多くのプローブはCPU呼び出しツリーにクロスリンクを追加します。たとえば、クラスローダープローブは、クラスのロードがトリガーされた場所を示すことができます。これは、呼び出しツリーでは見えず、予期しないオーバーヘッドを追加する可能性があります。呼び出しツリービューで不透明なデータベースコールは、対応するプローブでワンクリックでさらに分析できます。これにより、プローブリンクをクリックすると、プローブ呼び出しツリービューのコンテキストで分析が自動的に繰り返される呼び出しツリー分析でも機能します。



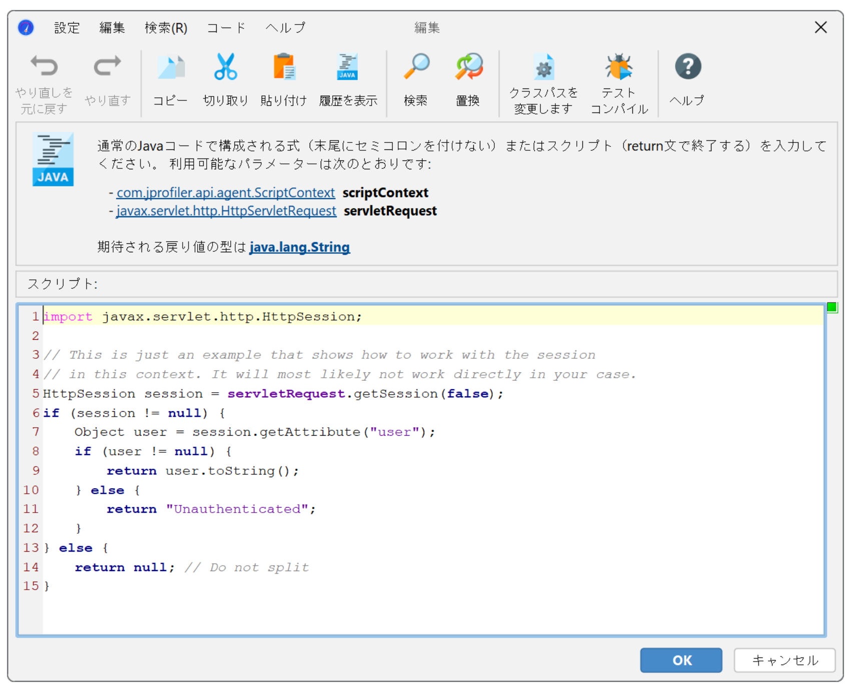
もう一つの可能性は、CPU呼び出しツリーにペイロード情報を直接インラインで表示することです。すべての関連するプローブには、その目的のためにAnnotate in call treeオプションがあります。その場合、プローブ呼び出しツリーへのリンクは利用できません。各プローブには独自のペイロードコンテナノードがあります。同じペイロード名を持つイベントは集約され、呼び出し回数と合計時間が表示されます。ペイロード名は、呼び出しスタックごとに統合され、最も古いエントリは「[earlier calls]」ノードに集約されます。呼び出しスタックごとに記録されるペイロード名の最大数は、プロファイリング設定で設定可能です。



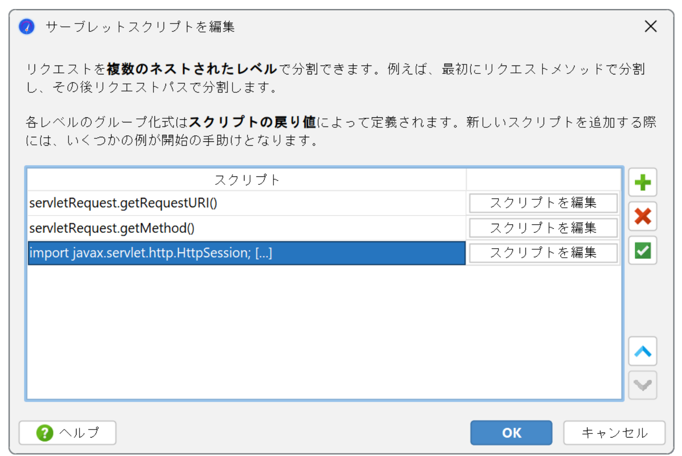

## 呼び出しツリーの分割

一部のプローブは、呼び出しツリーにペイロードデータを注釈付けするためにプローブ文字列を使用しません。むしろ、各異なるプローブ文字列のために呼び出しツリーを分割します。これは、サーバータイプのプローブに特に有用であり、異なるタイプの受信リクエストごとに呼び出しツリーを個別に表示したい場合に役立ちます。「HTTPサーバー」プローブはURLをインターセプトし、URLのどの部分を呼び出しツリーの分割に使用するかを細かく制御できます。デフォルトでは、パラメータなしでリクエストURIパスのみを使用します。

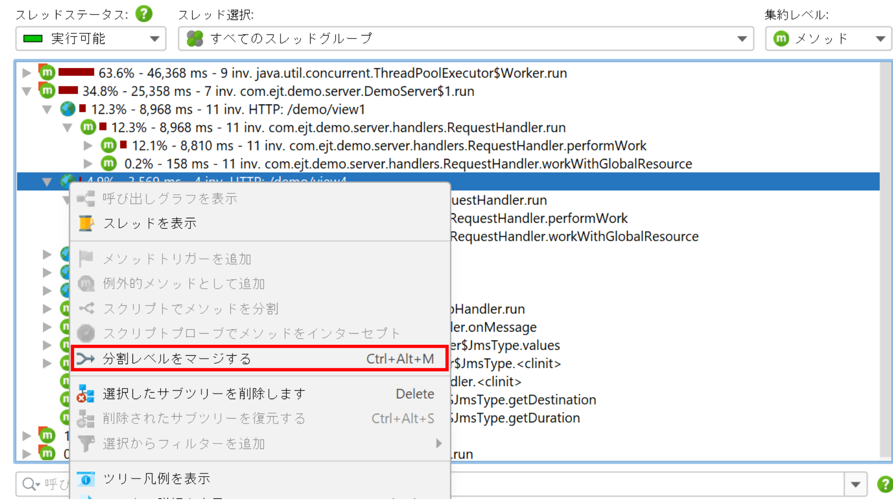
より柔軟性を持たせるために、分割文字列を決定するスクリプトを定義することができます。スクリプトでは、現在の`javax.servlet.http.HttpServletRequest`をパラメータとして受け取り、希望する文字列を返します。




さらに、単一の分割レベルに制限されず、複数のネストされた分割を定義することができます。たとえば、最初にリクエストURIパスで分割し、次にHTTPセッションオブジェクトから抽出されたユーザー名で分割することができます。また、リクエストメソッドでグループ化してからリクエストURIで分割することもできます。




ネストされた分割を使用することで、呼び出しツリーの各レベルに対して個別のデータを見ることができます。呼び出しツリーを見ているとき、あるレベルが邪魔になることがあり、「HTTPサーバー」プローブ設定からそれを排除する必要があるかもしれません。より便利に、記録されたデータを失うことなく、対応する分割ノードのコンテキストメニューを使用して、呼び出しツリー内で分割レベルを一時的にマージおよびアンマージすることができます。




呼び出しツリーを分割すると、かなりのメモリオーバーヘッドが発生する可能性があるため、慎重に使用する必要があります。メモリオーバーロードを避けるために、JProfilerは分割の最大数を制限します。特定の分割レベルの分割キャップが達成された場合、特別な「[capped nodes]」分割ノードが追加され、キャップカウンタをリセットするためのハイパーリンクが表示されます。デフォルトのキャップが目的に対して低すぎる場合は、プロファイリング設定でそれを増やすことができます。

## ガベージコレクタ分析

ガベージコレクタ（GC）のランタイム特性を理解し分析することは、いくつかの理由で重要です。まず第一に、GCの停止はアプリケーションの応答性に直接影響を与える可能性があります。ガベージコレクタの動作を理解することで、これらの停止を減らすために設定を最適化することができます。 一般的に、頻繁で長いGCサイクルは、ヒープが小さすぎるか、一時オブジェクトが多すぎることを示している可能性があります。

ガベージコレクタプローブを使用することで、これらの問題を解決し、適切なガベージコレクタの選択、ヒープサイズ、その他のJVMパラメータなど、JVM設定を調整する際により情報に基づいた決定を下すことができます。

ガベージコレクタプローブは他のプローブとは異なるビューを持ち、異なるデータソースを使用します。 そのデータはJVMのプロファイリングインターフェースから取得されるのではなく、JFRストリーミングを使用してGC関連のイベントをJDKフライトレコーダー[1]から分析します。 JFRイベントストリーミングに依存しているため、GCプローブはJava 17以上のHotspot JVMでプロファイルされた場合にのみ利用可能です。 JFRスナップショットを開く [p. 223]と、使用されているJavaバージョンに関係なく、同じプローブが利用可能です。

### ガベージコレクションビュー

ガベージコレクタプローブのメインビューは「ガベージコレクション」テーブルです。これは、記録されたすべてのガベージコレクションを行として表示し、最も重要なメトリクスを列として表示します。

ガベージコレクション　テレメトリー　GCサマリー　GC設定　GCフラグ　**ガベージコレクター** GCコレクションと構成

すべてのテキスト列でフィルター　フィルター

| GC ID | 開始時間 | 期間 | 原因 | コレクター | 最長ポーズ | 一時停止の合計 | 最終参照 | 弱い参照 | ソフト参照 | ファントム参照 |
|---|---|---|---|---|---|---|---|---|---|---|
| 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 |
| **合計 112 行:** | | 1,618 ms | | | | 645 ms | 152 | 12,588 | 4,731 | 3,539 |

「原因」列は、なぜガベージコレクションがトリガーされたかを示します。例えば、`System.gc()`の呼び出しが完全なガベージコレクションをトリガーしました。それは「コレクタ」列の関連する「G1Full」値からわかります。また、20msの大幅な停止を引き起こしたため、一般的に`System.gc()`を呼び出すことは良い考えではありません。 他の原因は、若い世代スペースのコレクション（「G1New」）や、古い世代の未参照オブジェクトをクリーンアップするG1コレクタの古いGCコレクション（「G1Old」）をトリガーします。 古いGCコレクションは、若い世代のコレクションよりも一貫して長くかかりますが、若い世代のコレクションはより多くのオブジェクトを収集します。

特殊なGC処理を伴う収集された参照は、「final」、「weak」、「soft」、「phantom」参照として別々の列に表示されます。

[1] https://en.wikipedia.org/wiki/JDK_Flight_Recorder

最長の停止と停止の合計のために別々の列がある理由は、各ガベージコレクションが複数のフェーズで構成され、それぞれが別々の停止を生成するためです。　また、ガベージコレクションの「期間」は停止の合計と等しくありません。ガベージコレクションは実行中にJVMを部分的にしか停止しないためです。スクリーンショットの「G1Old」コレクションは、期間の約5分の1しか停止しないことがわかります。

ガベージコレクションのさまざまなフェーズを調査するには、「GC ID」列のツリーアイコンを切り替えることができます。

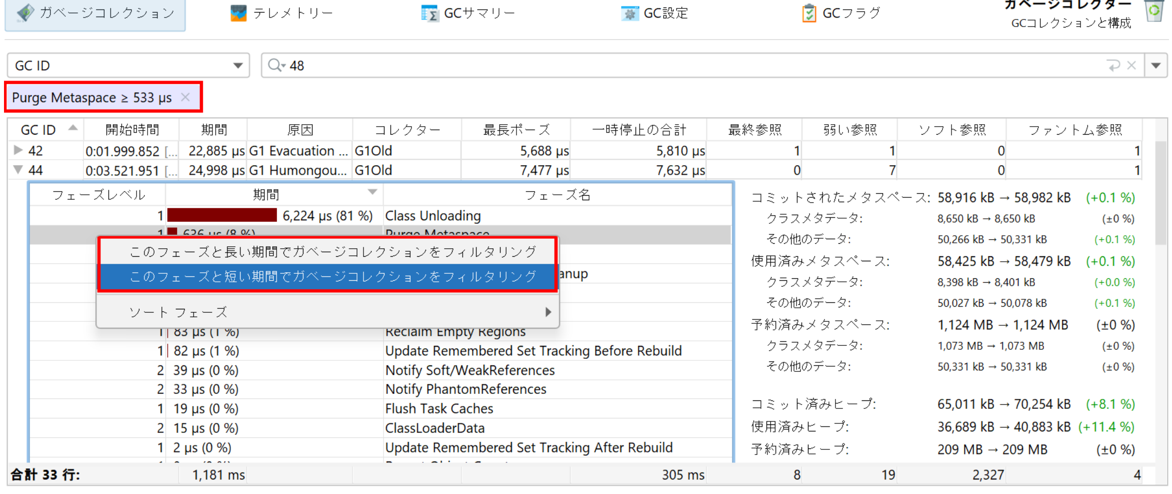
上のスクリーンショットでは、G1コレクタの混合GCコレクション（「G1Old」）が展開されています。ほとんどの時間が「クラスアンロード」に費やされており、これはJVMを停止しません。右側には、ガベージコレクションのさらなる統計が表示されています。ここでは、使用されたヒープは同じままで、使用されたメタスペースは0.1%増加しました。

各コレクタのフェーズは異なります。上のスクリーンショットでは、完全なコレクションが表示されています。ヒープ全体のライブオブジェクトをマークするのに多くの時間を費やしています。コレクションの終わりには、使用されたヒープが15.7%減少し、メタスペースは同じままでした。

ガベージコレクションを分析する際には、**フィルタリング**が異なるガベージコレクションのサブセットを比較するための重要なツールです。テーブルの上部にはフィルタセレクタがあり、任意の列を選択して対応するフィルタを設定できます。　同様のガベージコレクションを簡単に見る方法

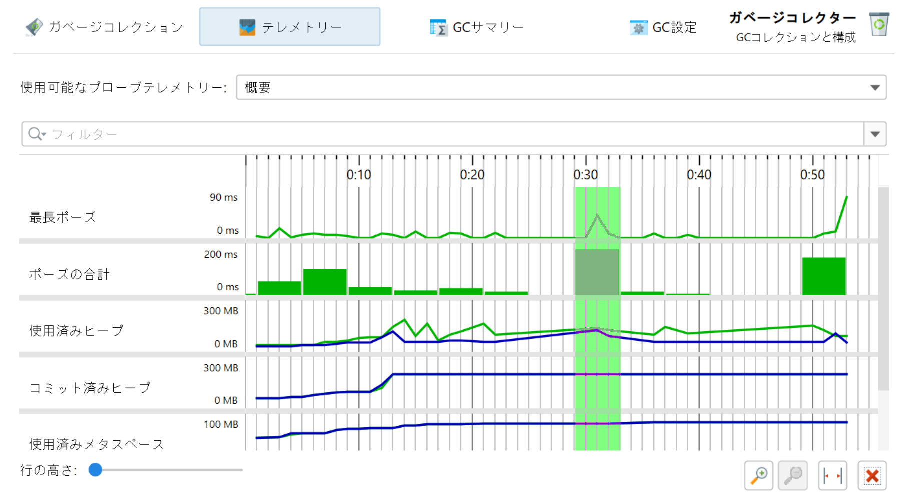
は、テーブルのコンテキストメニューを使用して、選択した行の列値に基づいてフィルタ条件を選択することです。

興味のあるガベージコレクションを絞り込むために複数のフィルタを追加できます。アクティブなフィルタはテーブルの上部にラベルとして表示されます。　ネストされたGCフェーズテーブルからフィルタを追加することも可能です。

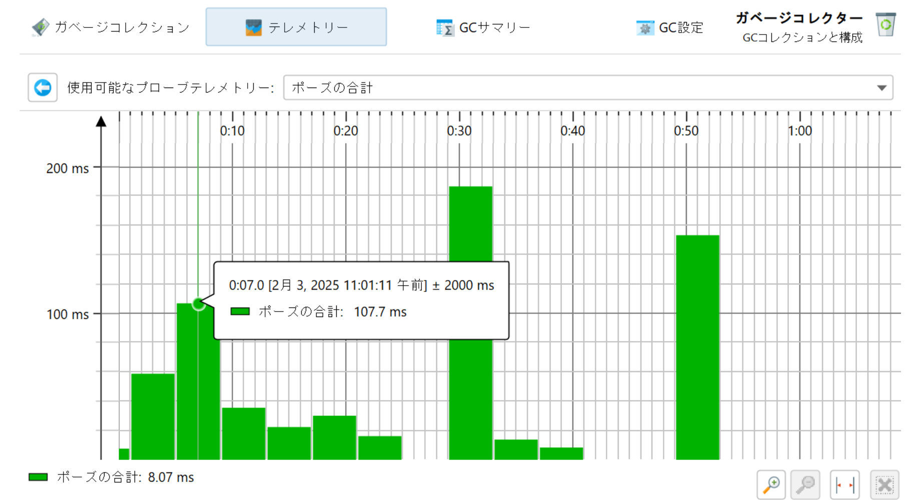


### テレメトリー

GCプローブは「テレメトリー」プローブビューで利用可能な多くのテレメトリーを生成します。




GCの停止を最小限に抑えたい場合は、上部の「最長停止」テレメトリーが最も興味深いものになるでしょう。テレメトリーの時間軸に沿ってドラッグすると、「ガベージコレクション」ビューで対応するガベージコレクションを選択できます。　垂直方向の解像度を向上させるために、上部のドロップダウンから単一のテレメトリーを選択するか、テレメトリーの名前をクリックすることができます。




上のスクリーンショットでは、時間にわたる停止の合計が表示されています。JProfilerは記録されたデータの**ヒストグラムを構築**することで合計可能な測定を提示します。ビンの幅は利用可能な水平スペースに依存するため、ヒストグラムのビンはズームレベルに応じて変化し、「スケールに合わせる」が有効な場合はウィンドウの幅に応じて変化します。変わらないのは、すべてのヒストグラムビンの下の総面積です。

**ヒープとメタスペースのテレメトリー**は、ガベージコレクションを展開するときに見ることができる統計に基づいています。これは、完全なプロファイリングセッションのメモリテレメトリーのように定期的にサンプリングされるデータではありません。ある期間にガベージコレクションが発生しない場合、データはありません。割り当て活動が少ないJVMでは、時間軸に沿って長い間隔があり、グラフは2つのガベージコレクションの間で補間されるだけです。

これらのテレメトリーのそれぞれには、「GC前」と「GC後」の2つのデータラインがあります。「使用済みヒープ」テレメトリーでは、通常、違いが大きいです。各時点で、2つのデータラインの値を比較することで、ガベージコレクションがどれだけの作業を行ったかを確認できます。正確

な値を取得するには、ツールチップを見てください。「コミット済みヒープ」テレメトリーとメタスペーステレメトリーでは、両方のラインの違いはしばしば小さいです。

JFRスナップショット[p.223]を分析している場合、`jdk.GCHeapSummary` JFRイベントタイプからの同じデータがテレメトリーセクションの「メモリ」テレメトリーでも使用されます。ただし、その場合、「GC前」と「GC後」の値は同じデータラインに表示され、GCプローブテレメトリーのように1秒ごとの粒度に集約されないため、グラフは異なって見えます。

### GCサマリー

GCサマリーは、記録期間全体にわたって集約された測定値を示します。 各測定は、ガベージコレクションの数、平均値、最大値、および合計値を提供します。最も重要なデータは上部の「停止時間」で、これはアプリケーションの生存性に直接影響を与えます。

他のトップレベルカテゴリは、すべてのコレクションの合計時間を示し、それが若いコレクションと古いコレクションの2つのサブカテゴリに分割されます。

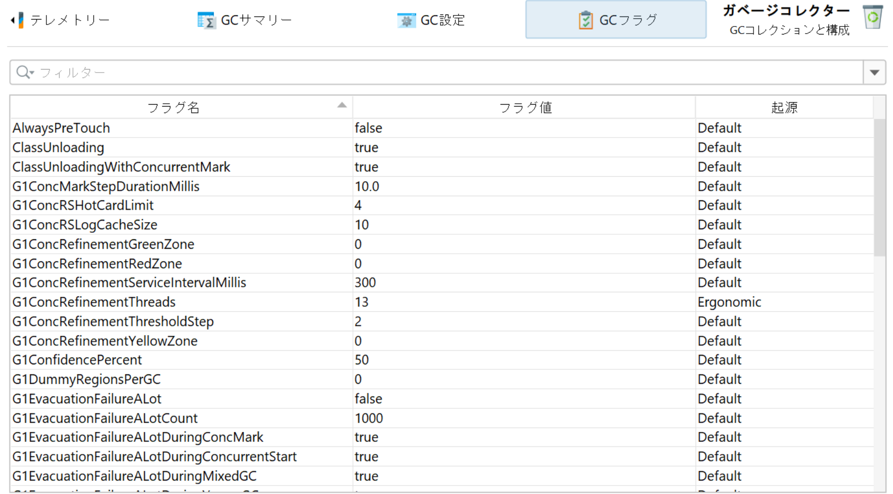
### GC設定

ガベージコレクタを調整する際には、明示的に設定されるか、ガベージコレクタ自体によって暗黙的に設定される一般的なプロパティを調査したいかもしれません。

これらのプロパティはすべてのガベージコレクタに共通であり、ガベージコレクタ間の違いを理解するのに役立ちます。

### GCフラグ

最後に、GC固有のフラグは、ガベージコレクタのどのプロパティを調整できるかを示し、実際の値を確認することができます。




| フラグ名 | フラグ値 | 起源 |
|---|---|---|
| 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 |

「起源」列は、フラグがどのように設定されたかを示します。「デフォルト」値は標準設定から変更されていないものであり、「エルゴノミック」フラグはガベージコレクタによって自動的に調整されたものです。 コマンドラインで特定のGCフラグを設定した場合、それらは起源として「コマンドライン」として報告されます。

## MBeanブラウザ

多くのアプリケーションサーバーやフレームワーク、例えば Apache Camel [(1)] は、JMXを使用して設定およびモニタリングの目的で多数のMBeanを公開しています。JVM自体も、JVM内の低レベル操作に関する興味深い情報を提供する プラットフォームMXBeans [(2)] を公開しています。

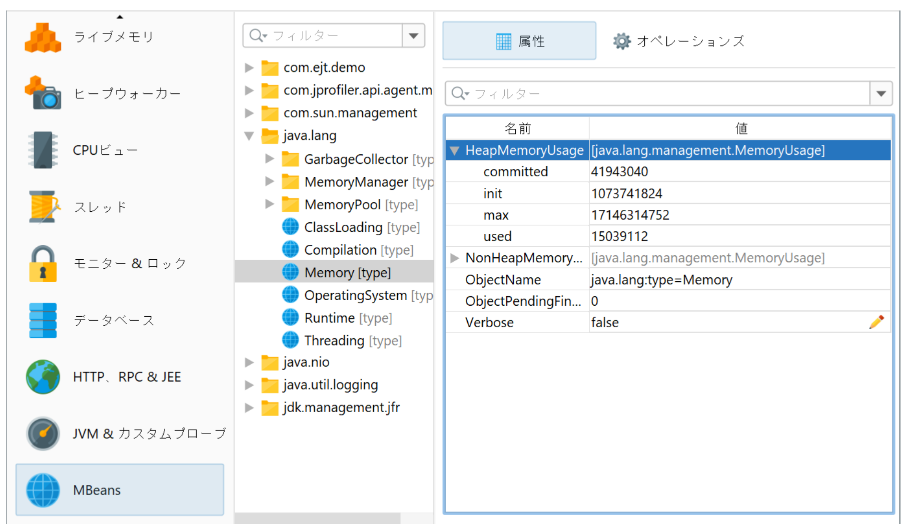
JProfilerには、プロファイルされたVM内のすべての登録されたMBeanを表示するMBeanブラウザが含まれています。MBeanサーバーにアクセスするためのJMXのリモート管理レベルは必要ありません。なぜなら、JProfilerエージェントはすでにインプロセスで実行されており、すべての登録されたMBeanサーバーにアクセスできるからです。

JProfilerは**Open MBeans**の型システムをサポートしています。Open MBeansは、いくつかの単純な型を定義するだけでなく、カスタムクラスを含まない複雑なデータ型を定義できます。また、配列やテーブルがデータ構造として利用可能です。 **MXBeans**を使用すると、JMXはJavaクラスからOpen MBeansを自動的に作成する簡単な方法を提供します。 例えば、JVMが提供するMBeansはMXBeansです。

MBeansには階層がありませんが、JProfilerはオブジェクトドメイン名を最初のコロンまでの部分を最初のツリーレベルとして取り、すべてのプロパティを再帰的にネストされたレベルとして使用してツリーに整理します。プロパティ値は最初に表示され、プロパティキーは末尾に括弧で示されます。`type`プロパティは、トップレベルノードのすぐ下に表示されるように 優先されます。

### 属性

MBeanコンテンツを表示するツリーテーブルのトップレベルでは、MBean属性が表示されます。




次のデータ構造はネストされた行として表示されます：

- **配列**

  プリミティブ配列とオブジェクト配列の要素は、キー名としてインデックスを使用してネストされた行に表示されます。

- **複合データ**

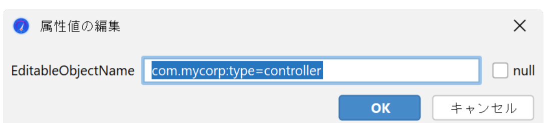
  複合データ型のすべてのアイテムはネストされた行として表示されます。各アイテムは任意の型である可能性があるため、 ネストは任意の深さまで続けることができます。

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

- **表形式データ**
  最も頻繁に遭遇するのは、`java.util.Map`のインスタンスが1つのキー列と1つの値列を持つ表形式データ型に マッピングされるMXBeansでの表形式データです。キーの型が単純型の場合、マップは「インライン」で表示され、各キーと値のペアはネストされた行として表示されます。キーが複雑な型を持つ場合、「マップエントリ」要素のレベルがネストされたキーと値のエントリと共に挿入されます。これは、複合キーと複数の値を持つ一般的な表形式タイプの場合も同様です。

オプションとして、MBean属性は編集可能であり、その場合は ✏ 編集アイコンが値の横に表示され、値を編集アクションがアクティブになります。複合型および表形式型はMBeanブラウザで編集できませんが、 配列または単純型は編集可能です。

値がnullableである場合、例えば配列のように、エディタにはnull状態を選択するためのチェックボックスがあります。



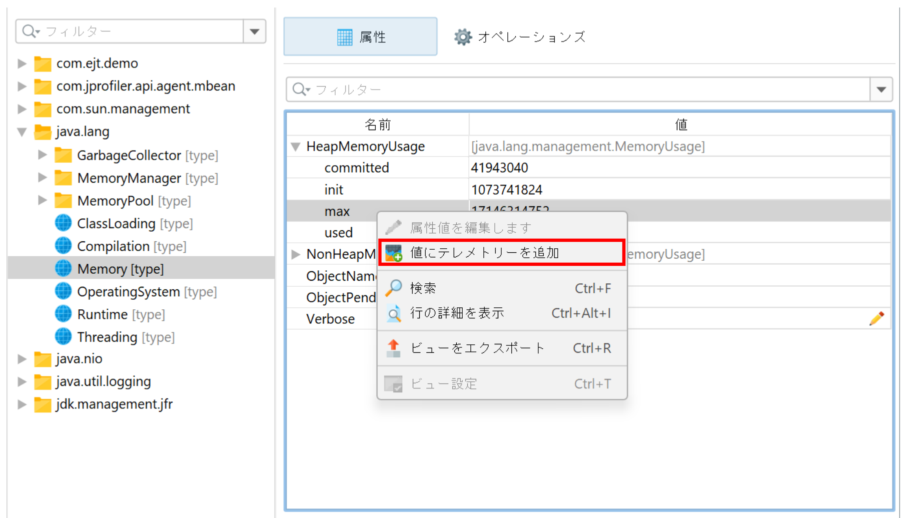

配列要素はセミコロンで区切られます。1つの末尾のセミコロンは無視できるため、 `1`と`1;`は同等です。セミコロンの前に値がない場合、オブジェクト配列ではnull値として扱われます。文字列配列では、ダブルクォート("")を使用して空の要素を作成し、セミコロンを含む要素は要素全体を引用符で 囲むことで作成できます。文字列要素内のダブルクォートは2倍にする必要があります。 例えば、次の文字列配列値を入力すると

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

次の文字列配列が作成されます

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

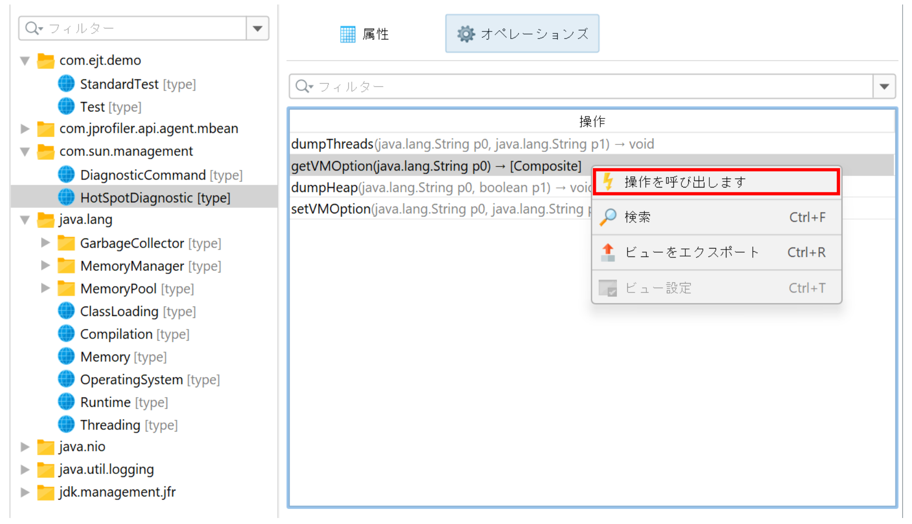
JProfilerは数値MBean属性値からカスタムテレメトリーを作成できます。カスタムテレメトリーのために MBeanテレメトリーラインを定義 [p. 46]すると、 テレメトリーデータを提供する属性を選択できるMBean属性ブラウザが表示されます。 すでにMBeanブラウザで作業している場合、コンテキストメニューの値のためのテレメトリーを追加アクションは、新しいカスタムテレメトリーを作成する便利な方法を提供します。



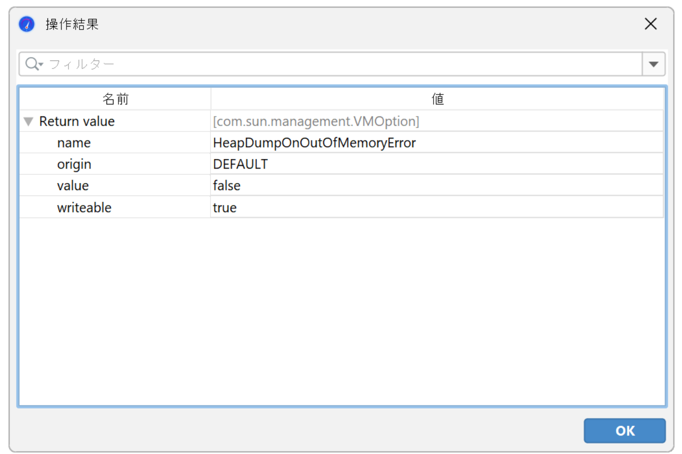

テレメトリーは、単純なキーと単一の値を持つ複合データまたは表形式データのネストされた値を追跡することもできます。ネストされた行を選択すると、パスコンポーネントがスラッシュで区切られた値パスが構築されます。

## 操作

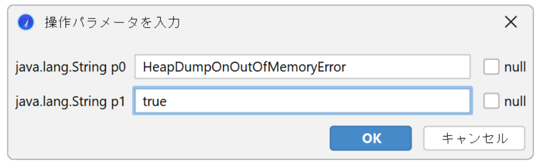
MBean属性の検査と変更に加えて、MBean操作を呼び出し、その戻り値を確認できます。 MBean操作は、MBeanインターフェース上のメソッドであり、セッターやゲッターではありません。




操作の戻り値は、複合型、表形式型、または配列型である可能性があるため、MBean属性ツリーテーブルに似た内容の新しいウィンドウが表示されます。単純な戻り値型の場合、「戻り値」という名前の行が1つだけ表示されます。他の型の場合、「戻り値」は結果が追加されるルート要素です。

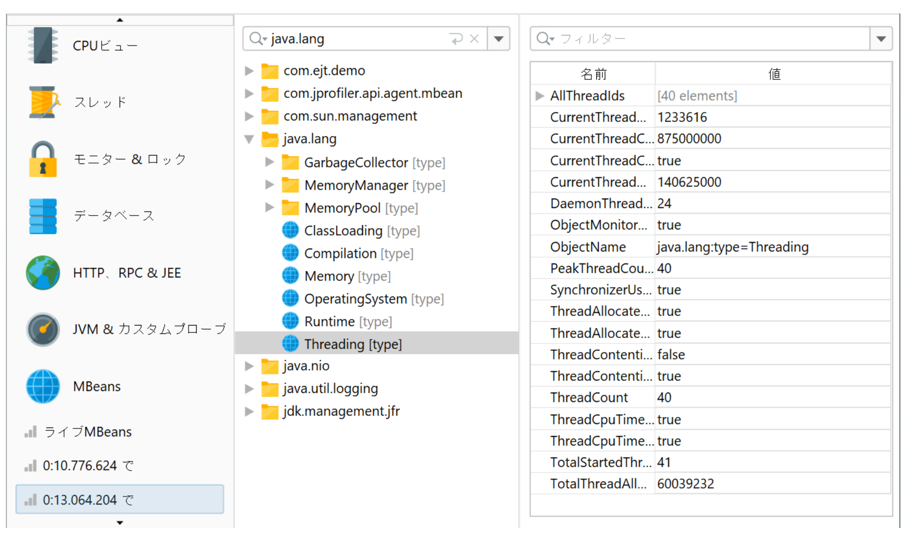



MBean操作には1つ以上の引数がある場合があります。それらを入力する際には、MBean属性を編集する場合と同じルールと制限が適用されます。



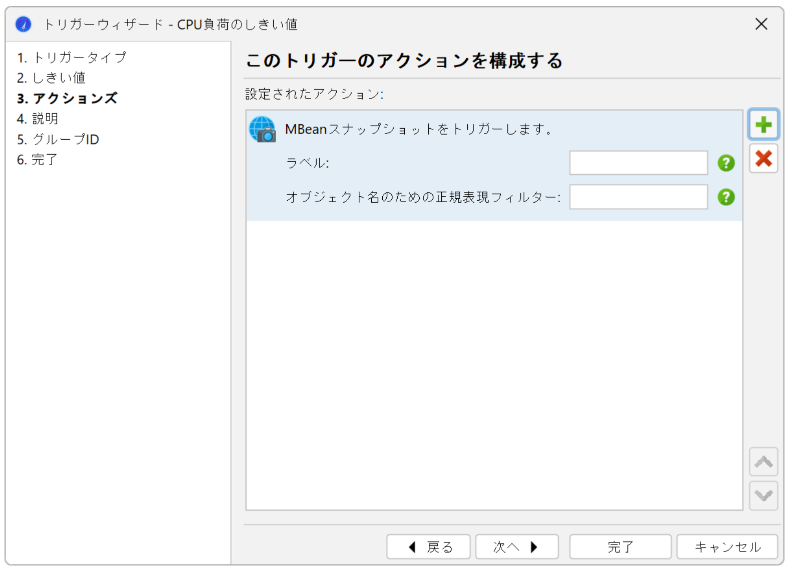

### MBeanスナップショット

MBeansのライブ値を表示することに加えて、現在の状態のスナップショットを撮ることもできます。各新しいスナップショットはMBeanビューセクションに別のビューとして追加され、カスタムラベルを割り当てることができます。スナップショットが撮影されると、現在のフィルターに従って表示されるMBeansのみが含まれます。 この方法で、特定のMBeansに集中し、目的に関連しないMBeansのクエリのオーバーヘッドを減らすことができます。




JProfiler UIでスナップショットを保存する際、すべてのMBeanスナップショットも保存されますが、ライブMBeanビューは保存されません。オフラインプロファイリング[p.130]の場合、Controller APIまたは「MBeanスナップショットを保存」トリガーアクションを使用してプログラム的にMBeanスナップショットを撮ることができます。

コントローラーAPIとトリガーアクションの両方は、ビューセレクターに表示されるオプションのラベルと、 含まれるMBeansをフィルタリングするためのオプションの正規表現をサポートしています。



# オフラインプロファイリング

JProfilerでアプリケーションをプロファイルするには、基本的に2つの異なる方法があります。デフォルトでは、JProfiler GUIをアタッチしてプロファイルします。JProfiler GUIは、記録の開始と停止のためのボタンを提供し、すべての記録されたプロファイリングデータを表示します。

JProfiler GUIを使用せずにプロファイルし、後で結果を分析したい場合があります。このシナリオのために、JProfilerはオフラインプロファイリングを提供します。オフラインプロファイリングでは、プロファイルされたアプリケーションをプロファイリングエージェントで開始し、JProfiler GUIと接続する必要がありません。

しかし、オフラインプロファイリングでもいくつかのアクションを実行する必要があります。少なくとも1つのスナップショットを保存しなければ、後で分析するためのプロファイリングデータは利用できません。また、CPUや割り当てデータを見るためには、ある時点で記録を開始する必要があります。同様に、保存されたスナップショットでヒープウォーカーを使用したい場合は、ヒープダンプをトリガーする必要があります。

### プロファイリングAPI

この問題に対する最初の解決策はコントローラーAPIです。APIを使用すると、コード内でプログラム的にすべてのプロファイリングアクションを呼び出すことができます。`api/samples/offline`ディレクトリには、コントローラーAPIを実際にどのように使用するかを示す実行可能な例があります。そのディレクトリで`../gradlew`を実行してコンパイルおよび実行し、テストプログラムがどのように呼び出されるかを理解するためにGradleビルドファイル`build.gradle`を学習してください。

コントローラーAPIは、実行時にプロファイリングアクションを管理するための主要なインターフェースです。これは、JProfilerインストール内の`bin/agent.jar`に含まれているか、Maven依存関係として以下の座標で利用できます。

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

およびリポジトリ

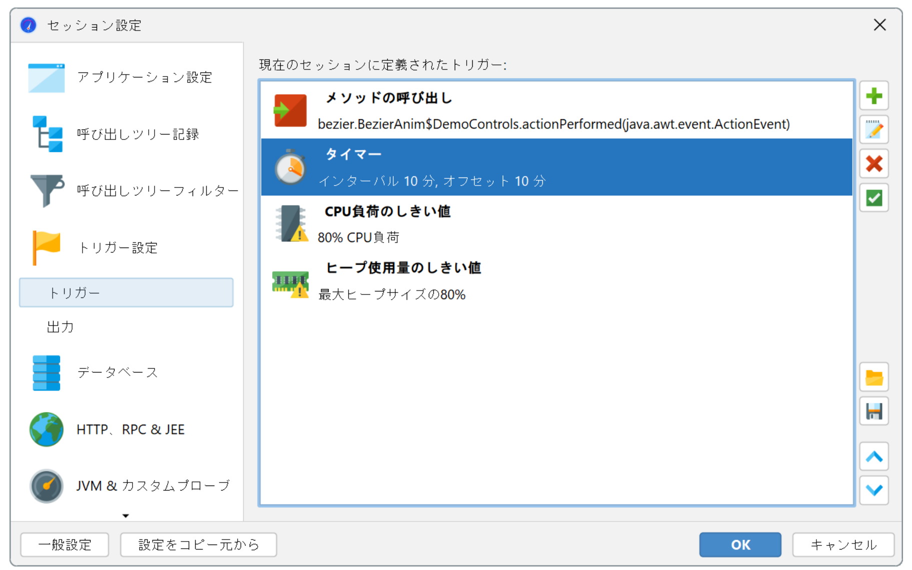
```
https://maven.ej-technologies.com/repository
```

プロファイリングAPIがアプリケーションの通常の実行中に使用される場合、API呼び出しは静かに何も行いません。

このアプローチの欠点は、開発中にJProfilerエージェントライブラリをアプリケーションのクラスパスに追加し、ソースコードにプロファイリング指示を追加し、プログラム的なプロファイリングアクションを変更するたびにコードを再コンパイルする必要があることです。

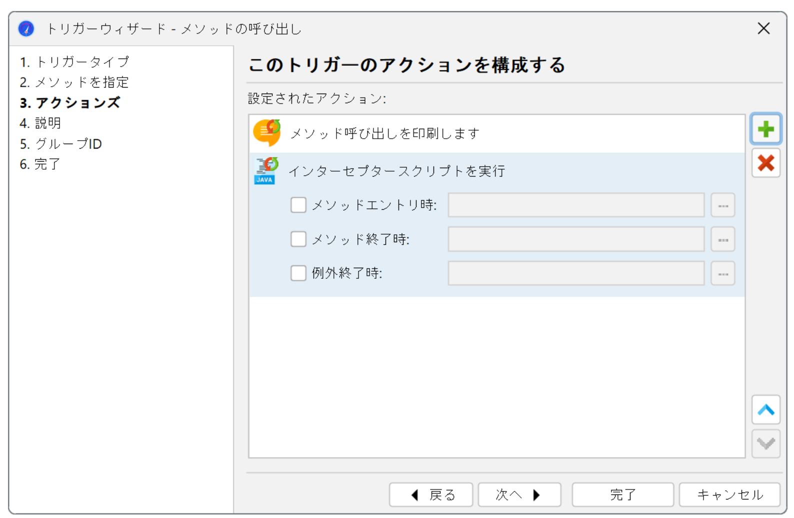
### トリガー

トリガー [p. 28]を使用すると、ソースコードを変更せずにJProfiler GUIでプロファイリングアクションをすべて指定できます。トリガーはJProfiler設定ファイルに保存されます。設定ファイルとセッションIDは、オフラインプロファイリングが有効になっている状態で開始するときにコマンドラインでプロファイリングエージェントに渡され、プロファイリングエージェントはそれらのトリガー定義を読み取ることができます。




プロファイリングAPIとは対照的に、API呼び出しをソースコードに追加するのではなく、トリガーはJVMで特定のイベントが発生したときにアクティブになります。たとえば、特定のプロファイリングアクションのためにメソッドの開始または終了時にAPI呼び出しを追加する代わりに、メソッド呼び出しトリガーを使用できます。別の使用例として、スナップショットを定期的に保存するために独自のタイマースレッドを作成する代わりに、タイマートリガーを使用できます。

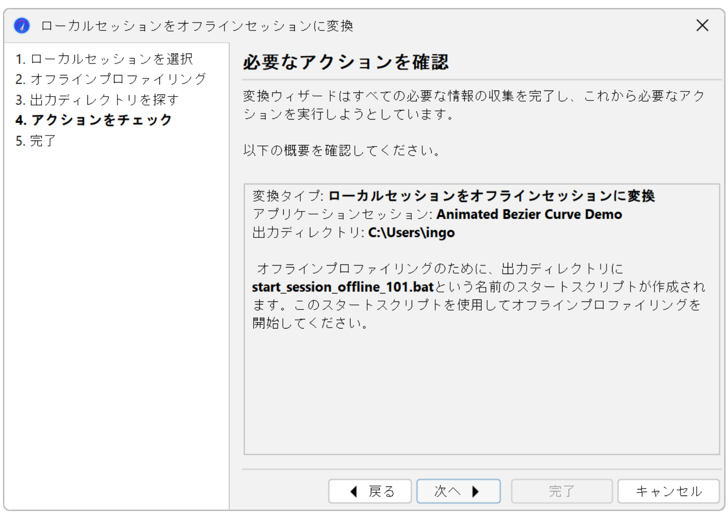
各トリガーには、関連するイベントが発生したときに実行されるアクションのリストがあります。これらのアクションの一部は、コントローラーAPIのプロファイリングアクションに対応しています。さらに、メソッド呼び出しをパラメータと戻り値と共に印刷するアクションや、メソッドのインターセプタースクリプトを呼び出すアクションなど、コントローラー機能を超える他のアクションもあります。



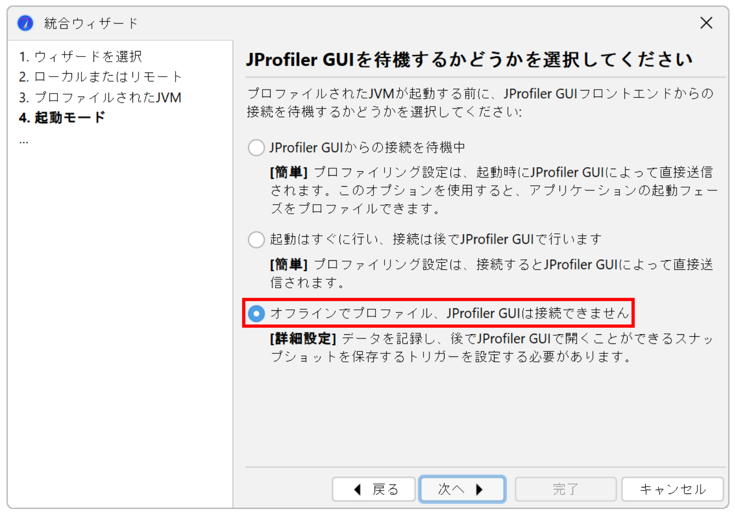

## オフラインプロファイリングの設定

JProfilerで起動されたセッションを設定している場合、メインメニューからセッション->変換ウィザード->アプリケーションセッションをオフラインに変換を呼び出すことで、オフラインセッションに変換できます。これにより、適切なVMパラメータを持つ開始スクリプトが作成され、JProfiler UIで使用するのと同じセッションからプロファイリング設定が取得されます。呼び出しを別のコンピュータに移動したい場合は、セッション->セッション設定のエクスポートを使用してセッション

を設定ファイルにエクスポートし、開始スクリプトのVMパラメータがそのファイルを参照していることを確認する必要があります。




統合ウィザードを使用してアプリケーションサーバーをプロファイルするときは、常に開始スクリプトまたは設定ファイルが変更され、プロファイリングのためのVMパラメータがJava呼び出しに挿入されます。すべての統合ウィザードには、「スタートアップ」ステップで「オフラインプロファイル」オプションがあり、インタラクティブプロファイリングの代わりにオフラインプロファイリングのためにアプリケーションサーバーを設定できます。




統合ウィザードで処理されない開始スクリプトがある場合など、Java呼び出しに自分でVMパラメータを渡したい場合があります。そのVMパラメータの形式は次のとおりです。

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

これは`[Generic application]`ウィザードから利用できます。

ライブラリパラメータとして`offline`を渡すと、オフラインプロファイリングが有効になります。この場合、JProfiler GUIとの接続はできません。`session`パラメータは、プロファイリング設定に使用する設定ファイルからのセッションを決定します。セッションのIDは、セッション設定ダイア

ログのアプリケーション設定タブの右上隅に表示されます。オプションのconfigパラメータは設定ファイルを指します。これは、セッション->セッション設定のエクスポートを呼び出してエクスポートできるファイルです。このパラメータを省略すると、標準の設定ファイルが使用されます。そのファイルは、ユーザーホームディレクトリの.jprofiler15ディレクトリにあります。

### GradleとAntを使用したオフラインプロファイリング

GradleまたはAntからオフラインプロファイリングを開始する場合、対応するJProfilerプラグインを使用して作業を簡単にすることができます。テストをプロファイルするためのGradleタスクの典型的な使用例を以下に示します。

```
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タスクは、標準のJavaExecタスクで実行するのと同じ方法で任意のJavaクラスをプロファイルします。JProfilerで直接サポートされていない他の方法でJVMを起動する場合、com.jprofiler.gradle.SetAgentPathPropertyタスクは必要なVMパラメータをプロパティに書き込むことができます。JProfilerプラグインを適用するとデフォルトで追加されるため、単に次のように書くことができます。

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

その後、タスクが実行された後にagentPathPropertyをプロジェクトプロパティ参照として他の場所で使用できます。すべてのGradleタスクと対応するAntタスクの機能は、別の章[p.253]で詳細に文書化されています。

### 実行中のJVMに対するオフラインプロファイリングの有効化

コマンドラインユーティリティbin/jpenableを使用すると、バージョン8以上の任意の実行中のJVMでオフラインプロファイリングを開始できます。VMパラメータと同様に、offlineスイッチ、セッションID、およびオプションの設定ファイルを指定する必要があります。

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

このような呼び出しでは、実行中のJVMのリストからプロセスを選択する必要があります。追加の引数--pid=<PID> --noinputを使用すると、プロセスを自動化し、ユーザー入力をまったく必要としないようにすることができます。

一方、オンザフライでオフラインプロファイリングを有効にする場合、いくつかの記録を手動で開始したり、スナップショットを保存したりする必要があるかもしれません。これは、bin/jpcontrollerコマンドラインツールで可能です。

プロファイリングエージェントがロードされているだけで、プロファイリング設定が適用されていない場合、記録アクションをオンにすることはできず、jpcontrollerは接続できません。これは、jpenableを使用してプロファイリングを有効にした場合でも、offlineパラメータを指定しない場合を含みます。オフラインモードを有効にすると、プロファイリング設定が指定され、jpcontrollerを使用できます。

jpenableおよびjpcontroller実行ファイルに関する詳細情報は、コマンドラインリファレンス[p. 253]で利用できます。

# スナップショットの比較

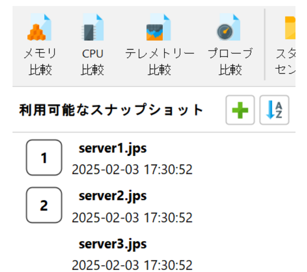
現在のアプリケーションのランタイム特性を以前のバージョンと比較することは、パフォーマンスの退行を防ぐための一般的な品質保証技術です。また、単一のプロファイリングセッション内でパフォーマンス問題を解決する際にも役立ちます。異なるユースケースを比較して、なぜ一方が他方よりも遅いのかを見つけることができます。どちらの場合も、関心のある記録データを含むスナップショットを保存し、セッション->新しいウィンドウでスナップショットを比較メニューから呼び出すか、スタートセンターのスナップショットを開くタブで複数のスナップショットを比較ボタンをクリックして、JProfilerのスナップショット比較機能を使用します。

## スナップショットの選択

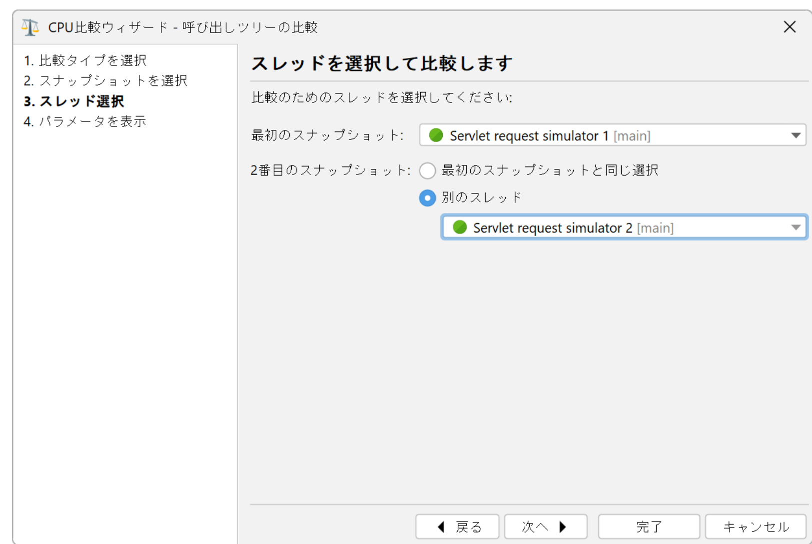
比較は、別のトップレベルウィンドウで作成および表示されます。まず、スナップショットセレクタにいくつかのスナップショットを追加します。その後、関心のあるスナップショットを選択し、比較ツールバーのボタンをクリックすることで、リストされたスナップショットのうち2つ以上から比較を作成できます。リスト内のスナップショットファイルの順序は重要です。すべての比較は、リストの下の方にあるスナップショットが後の時間に記録されたと仮定します。スナップショットを手動で配置する以外に、名前や作成時間でソートすることもできます。

JProfilerのメインウィンドウのビューとは異なり、比較ビューには固定されたビューのパラメータがあり、上部に表示されます。これにより、パラメータをその場で調整するドロップダウンリストはありません。すべての比較は、比較のためのパラメータを収集するウィザードを表示し、同じパラメータで複数回同じ比較を実行できます。ウィザードは以前の呼び出しからパラメータを記憶しているため、複数のスナップショットセットを比較する場合、設定を繰り返す必要はありません。いつでも完了ボタンでウィザードをショートカットするか、インデックスのステップをクリックして別のステップにジャンプできます。

比較がアクティブな場合、分析されたスナップショットは番号のプレフィックスと共に表示されます。2つのスナップショットで動作する比較では、表示される差異はスナップショット2の測定値からスナップショット1の測定値を引いたものです。

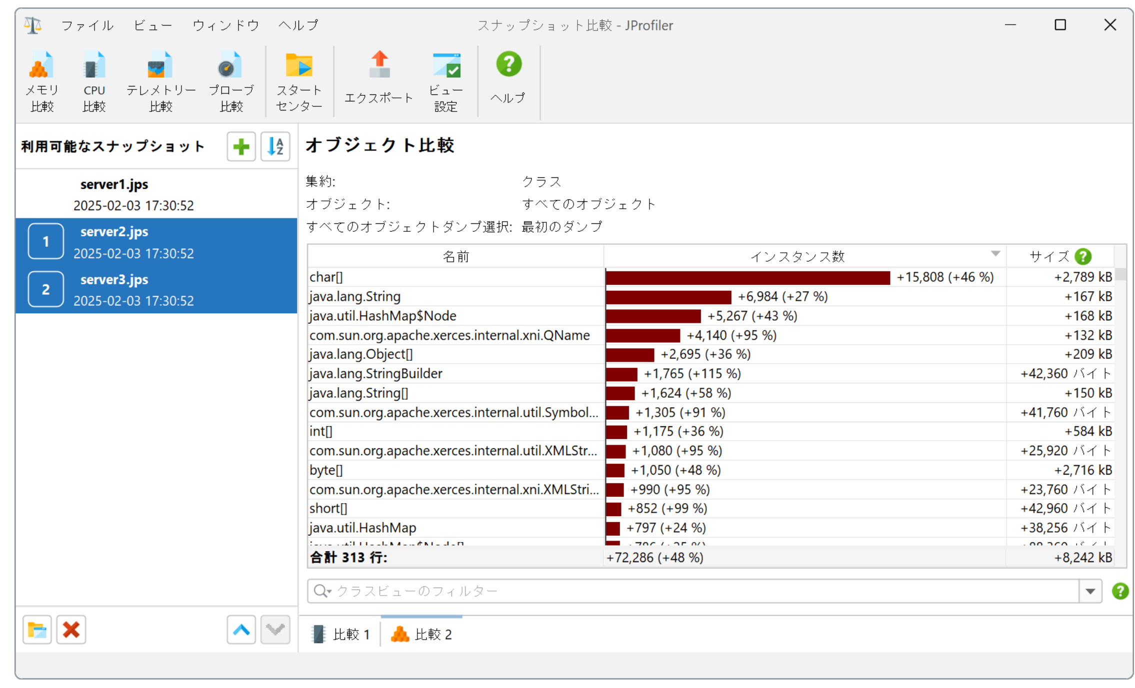



CPUの比較では、同じスナップショットを最初と2番目のスナップショットとして使用し、ウィザードで異なるスレッドやスレッドグループを選択できます。

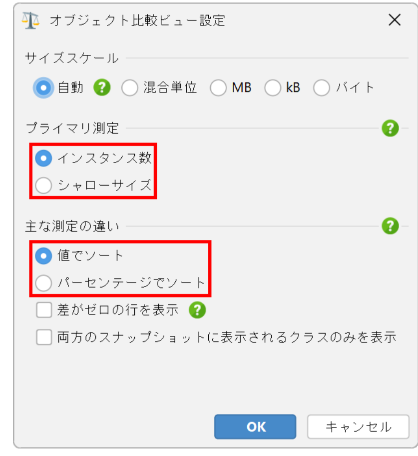



## テーブルによる比較

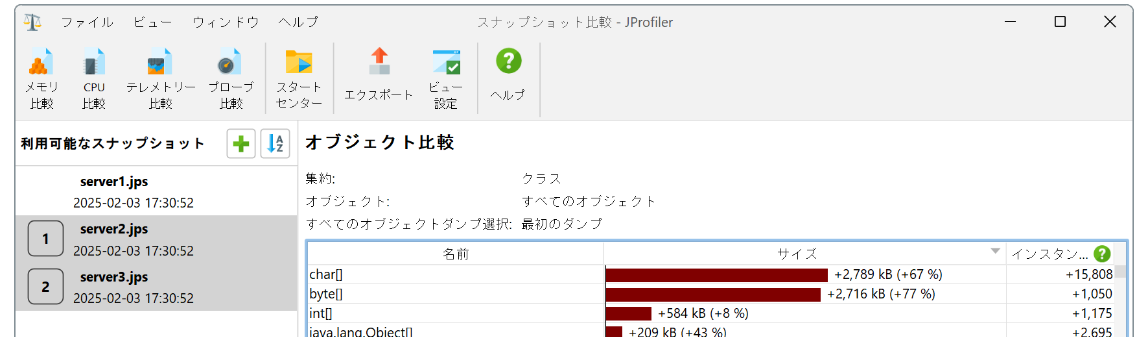
最も簡単な比較は「オブジェクト」メモリ比較です。「すべてのオブジェクト」、「記録されたオブジェクト」、またはヒープウォーカーの「クラス」ビューからデータを比較できます。比較の列はインスタンス数とサイズの差異を示しますが、インスタンス数列のみが双方向の棒グラフを表示し、増加は赤で右に、減少は緑で左に描かれます。




ビュー設定ダイアログで、この棒グラフが絶対変化を表示するかパーセンテージを表示するかを選択できます。他の値は括弧内に表示されます。この設定は、列がどのようにソートされるかも決定します。




最初のデータ列の測定値はプライマリメジャーと呼ばれ、ビュー設定でデフォルトのインスタンス数からシャローサイズに切り替えることができます。




テーブルのコンテキストメニューから、同じ比較パラメータで選択されたクラスの他のメモリ比較にショートカットできます。

**オブジェクト比較**

集約: クラス
オブジェクト: すべてのオブジェクト
すべてのオブジェクトダンプ選択: 最初のダンプ

| 名前 | インスタンス数 | サイズ |
|---|---|---|
| char[] | +15,808 (+46 %) | +2,789 kB |
| java.lang.String | +6,984 (+27 %) | +167 kB |
| java.util.HashMap$Node | +5,267 (+43 %) | +168 kB |
| com.sun.org.apache.xer | (+95 %) | +132 kB |
| java.lang.Object[] | 5 %) | +209 kB |
| java.lang.StringBuilder | %) | +42,360 バイト |
| java.lang.String[] | ) | +150 kB |
| com.sun.org.apache.xer | | +41,760 バイト |
| int[] | | +584 kB |
| com.sun.org.apache.xer | | +25,920 バイト |
| byte[] | | +2,716 kB |
| com.sun.org.apache.xer | | +23,760 バイト |
| short[] | | +42,960 バイト |
| java.util.HashMap | | +38,256 バイト |
| 合計 313 行: | | +8,242 kB |

割り当て呼び出しツリー比較を作成
割り当てホットスポット比較を作成
ソースを表示 F4
バイトコードを表示
ソート クラス
検索 Ctrl+F
ビューをエクスポート Ctrl+R
ビュー設定 Ctrl+T

クラスビューのフィルター

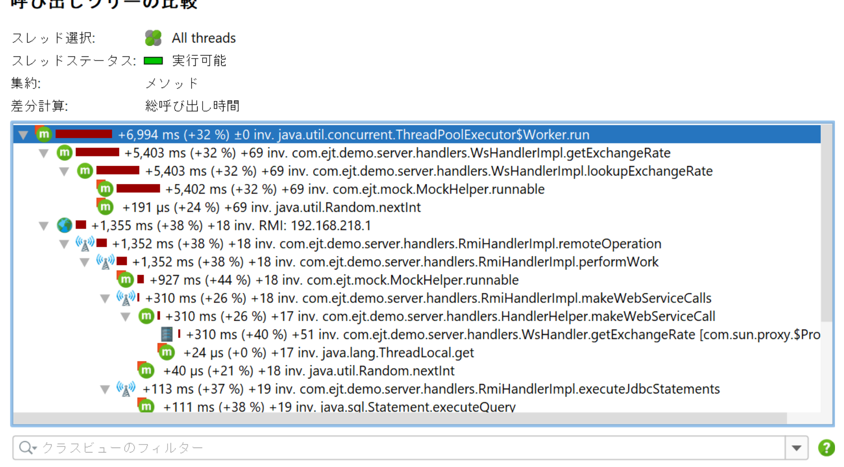
オブジェクト比較と同様に、CPUホットスポット、プローブホットスポット、割り当てホットスポットの比較も同様のテーブルで表示されます。

## ツリーによる比較

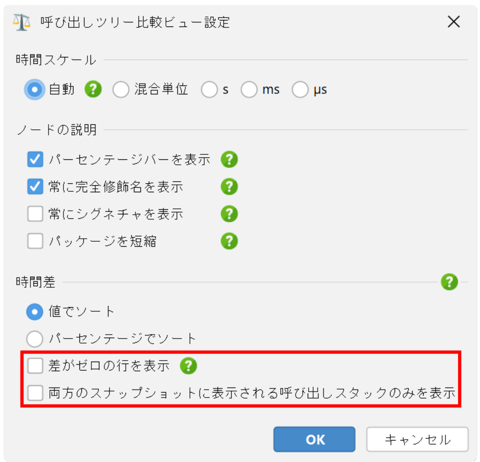
CPU呼び出しツリー、割り当て呼び出しツリー、プローブ呼び出しツリーのそれぞれについて、選択されたスナップショット間の差異を示す別のツリーを計算できます。通常の呼び出しツリービューとは対照的に、インラインバー図は今や変化を表示し、増加は赤、減少は緑で示されます。

**呼び出しツリーの比較**

スレッド選択: All threads
スレッドステータス: 実行可能
集約: メソッド
差分計算: 総呼び出し時間



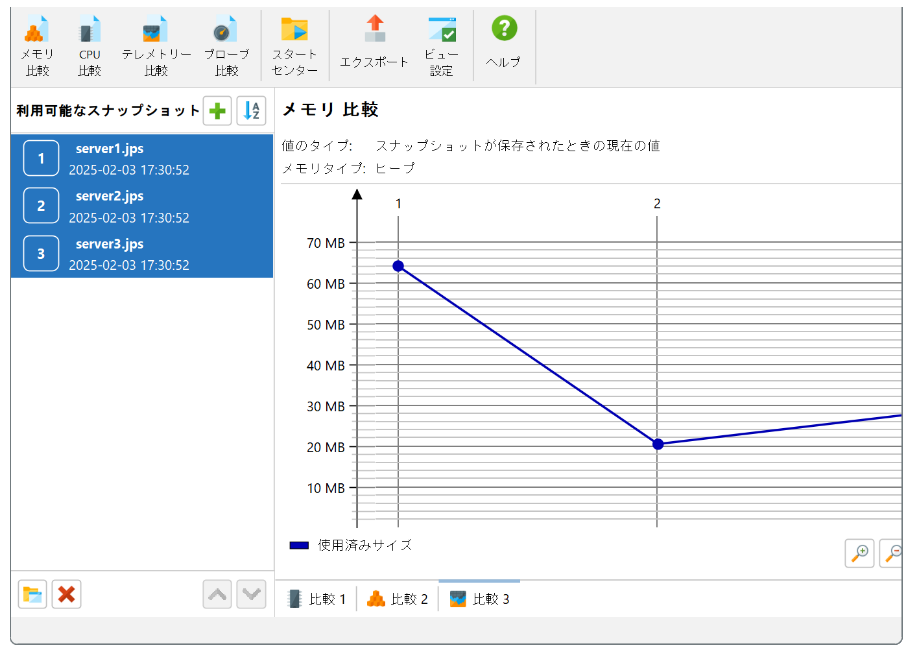
手元のタスクに応じて、両方のスナップショットファイルに存在し、一方のスナップショットファイルから他方に変化した呼び出しスタックのみを表示する方が簡単になる場合があります。この動作はビュー設定ダイアログで変更できます。



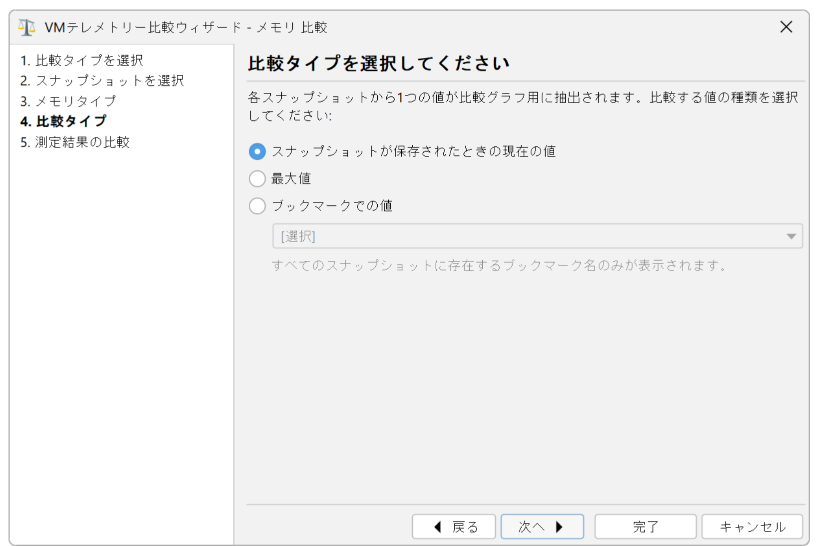

CPUおよびプローブ呼び出しツリーの比較では、合計時間ではなく平均時間を比較することが興味深いかもしれません。これはウィザードの「ビューのパラメータ」ステップで選択できるオプションです。

## テレメトリー比較

テレメトリー比較では、同時に2つ以上のスナップショットを比較できます。スナップショットセレクタでスナップショットを選択しない場合、ウィザードはすべてのスナップショットを比較するものと仮定します。テレメトリー比較には時間軸がなく、番号付きの選択されたスナップショットが順序付きのx軸として表示されます。ツールチップにはスナップショットの完全な名前が含まれています。




比較は各スナップショットから1つの数値を抽出します。テレメトリーデータは時間解決されているため、これを行う方法は複数あります。ウィザードの「比較タイプ」ステップでは、スナップショットが保存されたときの値を使用するか、最大値を計算するか、選択されたブックマークでの値を見つけるオプションを提供します。

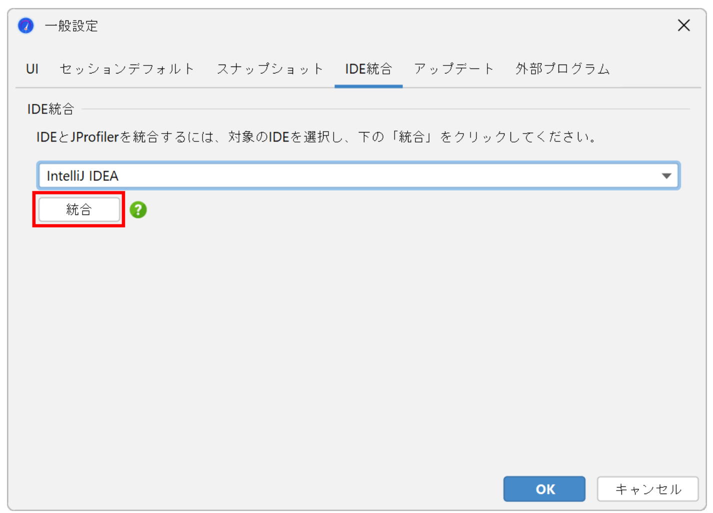


# IDE統合

アプリケーションをプロファイルするとき、JProfilerのビューに表示されるメソッドやクラスは、しばしばそのソースコードを見ないと答えられない質問を引き起こします。JProfilerはその目的のために組み込みのソースコードビューアを提供していますが、機能は限定されています。また、問題が見つかった場合、次のステップは通常、問題のあるコードを編集することです。理想的には、JProfilerのプロファイリングビューからIDEへの直接のパスがあるべきで、手動での検索なしにコードを検査し改善することができます。

## IDE統合のインストール

JProfilerはIntelliJ IDEA、eclipse、NetBeansのためのIDE統合を提供しています。IDEプラグインをインストールするには、メインメニューからセッション->IDE統合を呼び出します。IntelliJ IDEAのプラグインインストールはIDEのプラグイン管理で行われ、他のIDEではプラグインはJProfilerによって直接インストールされます。インストーラーもこのアクションを提供し、JProfilerのインストールとともにIDEプラグインを簡単に更新できるようにします。統合ウィザードは、プラグインをJProfilerの現在のインストールディレクトリに接続します。IDEプラグインの設定では、いつでも使用するJProfilerのバージョンを変更できます。プラグインとJProfiler GUIの間のプロトコルは後方互換性があり、古いバージョンのJProfilerとも動作します。

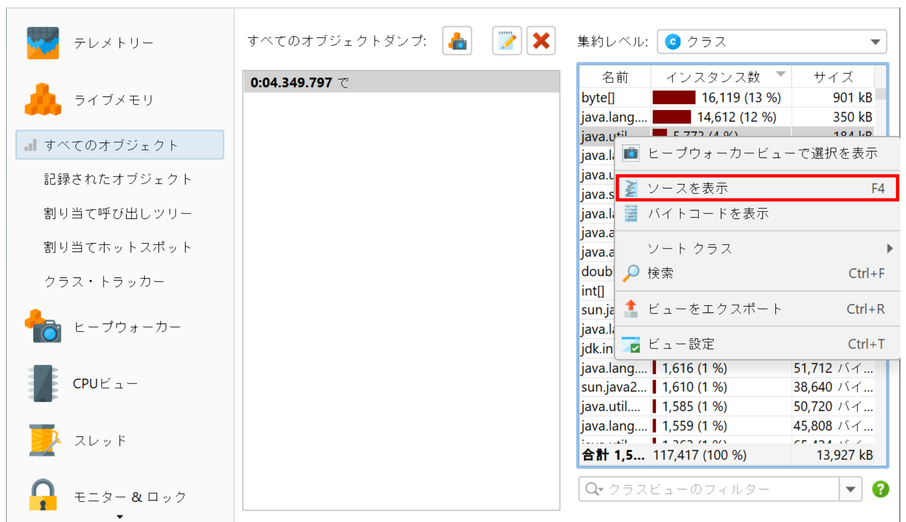



IntelliJ IDEAの統合はプラグインマネージャからもインストールできます。その場合、初めてプロファイルするときにプラグインがJProfilerの実行可能ファイルの場所を尋ねます。

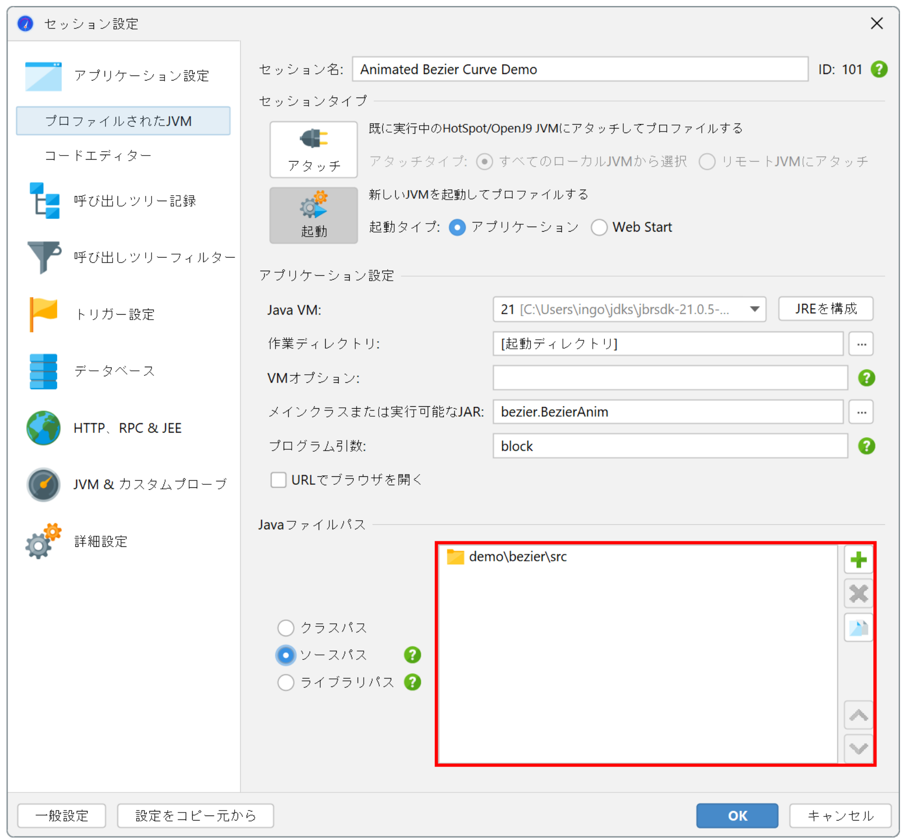
異なるプラットフォームでは、JProfilerの実行可能ファイルは異なるディレクトリに配置されています。Windowsでは`bin\jprofiler.exe`、LinuxまたはUnixでは`bin/jprofiler`、macOSではIDE統合用にJProfilerアプリケーションバンドル内に特別なヘルパーシェルスクリプト`Contents/Resources/app/bin/macos/jprofiler.sh`があります。

## ソースコードナビゲーション

JProfilerでクラス名やメソッド名が表示されるすべての場所で、コンテキストメニューにはソースを表示アクションが含まれています。

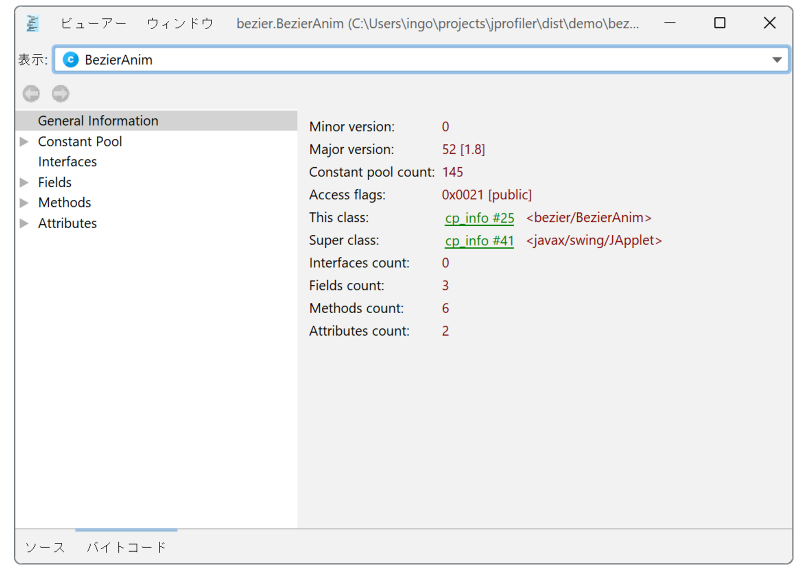



セッションがIDEから開始されていない場合、組み込みのソースコードビューアが表示され、コンパイルされたクラスファイルの行番号テーブルを使用してメソッドを見つけます。ソースファイルは、そのルートディレクトリまたはZIPファイルがアプリケーション設定で設定されている場合にのみ見つかります。




ソースコード表示とともに、jclasslibバイトコードビューア[(1)]に基づくバイトコードビューアがコンパイルされたクラスファイルの構造を表示します。

(1) https://github.com/ingokegel/jclasslib



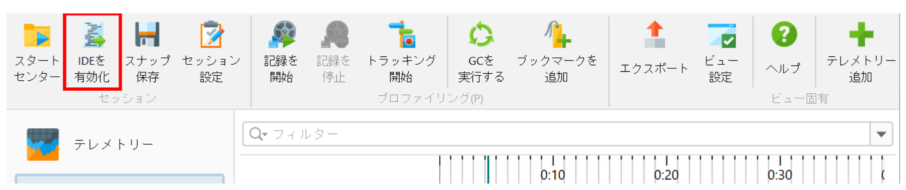

セッションがIDEから起動された場合、統合されたソースコードビューアは使用されず、ソースを表示アクションはIDEプラグインに委ねられます。IDE統合は起動されたプロファイリングセッション、保存されたスナップショットのオープン、実行中のJVMへのアタッチをサポートします。

ライブプロファイリングセッションでは、プロファイルされたアプリケーションをIDEのために実行またはデバッグするのと同様に開始します。JProfilerプラグインはプロファイリングのためのVMパラメータを挿入し、JProfilerウィンドウを接続します。JProfilerは別プロセスとして実行され、必要に応じてプラグインによって開始されます。JProfilerからのソースコードナビゲーションリクエストはIDEの関連プロジェクトに送信されます。JProfilerとIDEプラグインは、タスクバーのエントリが点滅せず、単一プロセスを扱っているかのようにウィンドウ切り替えをシームレスに行うために協力します。

セッションを開始するとき、「セッションスタートアップ」ダイアログでプロファイリング設定をすべて構成できます。起動されたセッションに使用される構成済みのプロファイリング設定は、IDE統合に応じてプロジェクトごとまたは実行構成ごとにJProfilerによって記憶されます。セッションが初めてプロファイルされるとき、IDEプラグインはソースファイルのパッケージ階層の最上位クラスに基づいてプロファイルされたパッケージのリストを自動的に決定します。後の任意の時点で、セッション設定ダイアログのフィルター設定ステップに移動し、リセットボタンを使用してこの計算を再度実行できます。

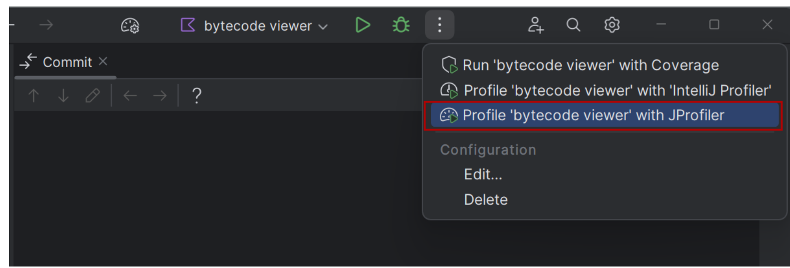
スナップショットの場合、IDE統合はIDE内からスナップショットファイルをファイル->開くアクションで開くか、プロジェクトウィンドウでダブルクリックすることで設定されます。JProfilerからのソースコードナビゲーションは現在のプロジェクトに向けられます。最後に、IDEプラグインはIDEにJVMにアタッチアクションを追加し、実行中のJVMを選択してIDEにソースコードナビゲーションを取得できます。これはスナップショットのメカニズムと似ています。

時には特定のクラスやメソッドを考慮せずにIDEに切り替えたいことがあります。そのために、JProfilerウィンドウのツールバーにはIDE統合によって開かれたプロファイリングセッションに対してIDEをアクティブ化ボタンが表示されます。このアクションは`F11`キーにバインドされており、IDEでのJProfilerアクティベーションアクションと同様に、同じキーのバインディングでIDEとJProfilerの間を行き来できます。

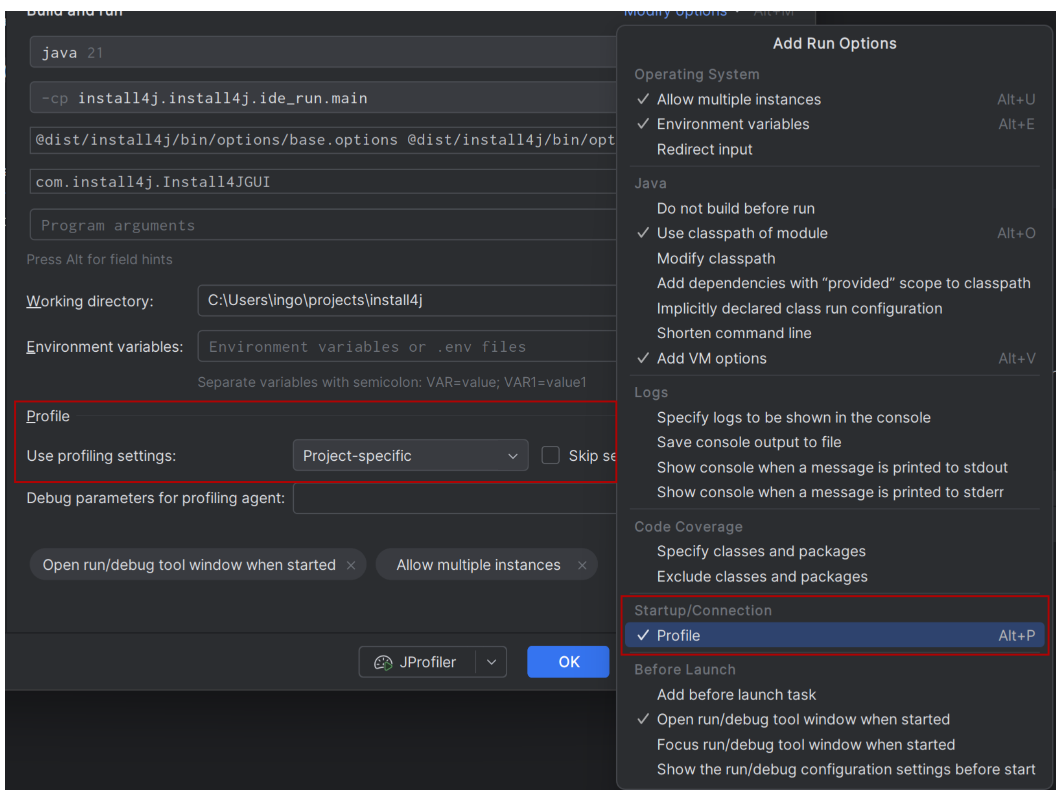


## IntelliJ IDEA統合

IntelliJ IDEAからアプリケーションをプロファイルするには、実行メニューのプロファイリングコマンドのいずれかを選択するか、メインツールバーの実行またはデバッグアクションの横にあるドロップダウンメニューをクリックして「JProfilerでプロファイル」アクションを選択します。JProfilerはアプリケーションサーバーを含むほとんどの実行IDEA構成タイプをプロファイルできます。




JProfilerプラグインは、すぐには見えない実行構成に追加の設定を追加します。これらの設定にアクセスするには、「オプションを変更」ドロップダウンで「プロファイル」オプションを選択します。他のすべてのプロファイリング設定は、JProfilerウィンドウのスタートアップダイアログで構成できます。



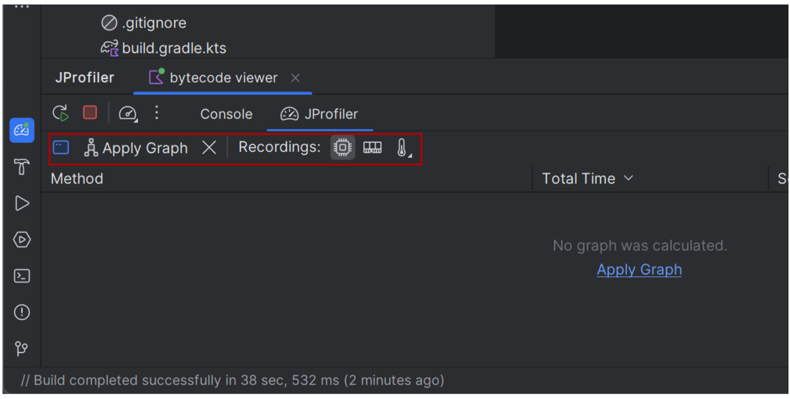

プロファイリングセッションが開始されると、出力は別のJProfilerツールウィンドウに表示されます。そのツールウィンドウは、通常の実行ツールウィンドウのようにコンソール出力を表示し、JProfiler UIに接続した後に使用できる「JProfiler」タブを含みます：

```
JProfiler> Listening locally on port: 34558.
JProfiler> Enabling native methods instrumentation.
JProfiler> Can retransform classes.
JProfiler> Can retransform any class.
JProfiler> Native library initialized
JProfiler> VM initialized
JProfiler> Retransforming 7 base class files.
JProfiler> Base classes instrumented.
JProfiler> Waiting for a connection from the JProfiler GUI ...
JProfiler> Using instrumentation
JProfiler> Time measurement: elapsed time
JProfiler> CPU profiling enabled
```

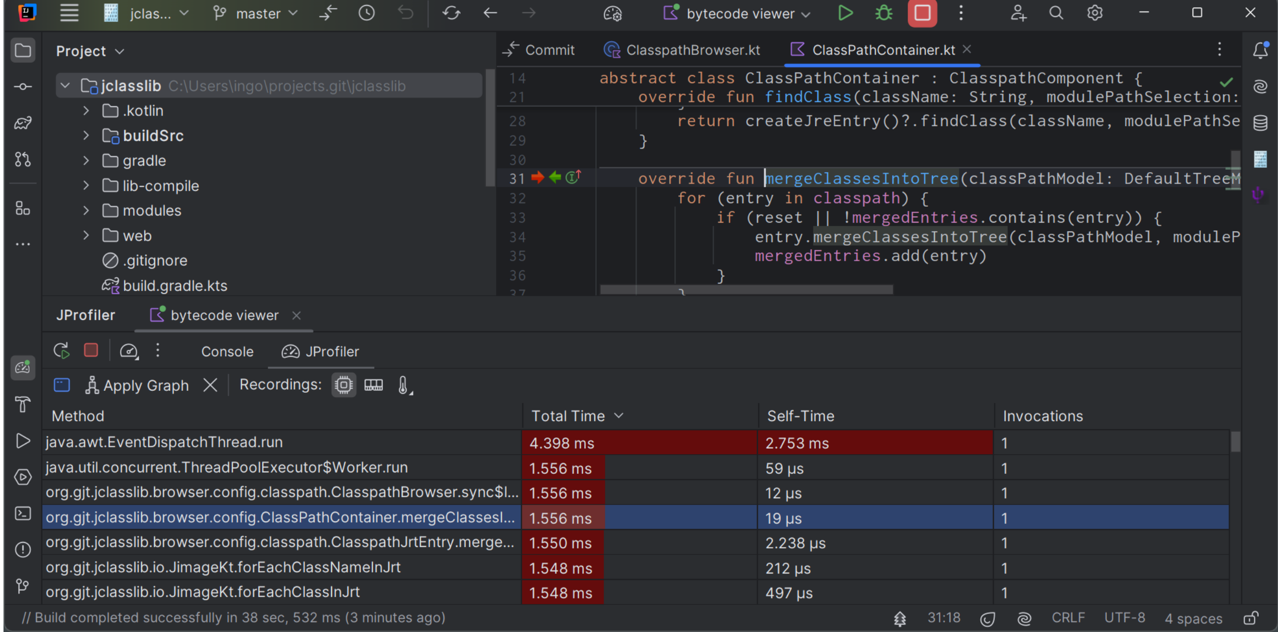
JProfilerツールウィンドウは、IntelliJ IDEAでJProfilerスナップショットを開いたときや、「JProfilerでJVMにアタッチ」アクションで実行中のJVMにアタッチしたときにも表示されます。

「JProfiler」タブには、CPUデータ、割り当てデータ、プローブイベントのデータ記録を開始および停止するアクションが含まれています。さらに、JProfilerウィンドウに切り替えるアクションもあります。JProfilerウィンドウには、IDEAウィンドウに戻るための同様のアクションが含まれており、2つの別々のウィンドウで作業するのが便利になります。JProfilerからIntelliJ IDEAへの正確なソースコードナビゲーションはJavaとKotlinに対して実装されています。

プロファイリング情報は通常JProfilerウィンドウに表示されますが、CPUグラフデータはIntelliJ IDEA UIにも統合されています。これは、このデータをソースコード内で直接表示するのが理にかなっているからです。IntelliJ IDEAで「グラフを適用」アクションを使用するか、JProfilerでCPUグラフを生成して、IntelliJ IDEA内でCPUデータを表示します。スレッド選択などの高度なパラメータを構成したり、呼び出しツリールート、呼び出しツリー削除、呼び出しツリービューフィルター設定を呼び出しツリービューから使用するには、JProfilerウィンドウでグラフを生成する必要があります。



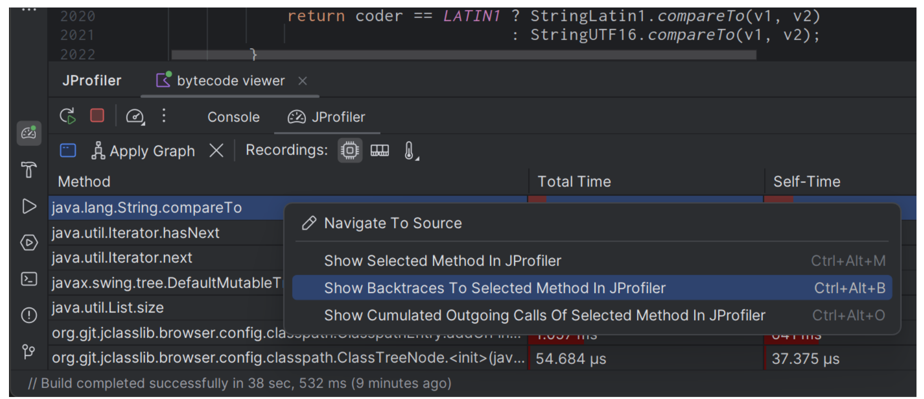

CPUデータが適用されると、「JProfiler」タブには記録されたメソッドのリストが表示されます。メソッドをダブルクリックすると、ソースコードに移動します。ソースコードエディタのガターには、着信および発信呼び出しの矢印が追加されます。



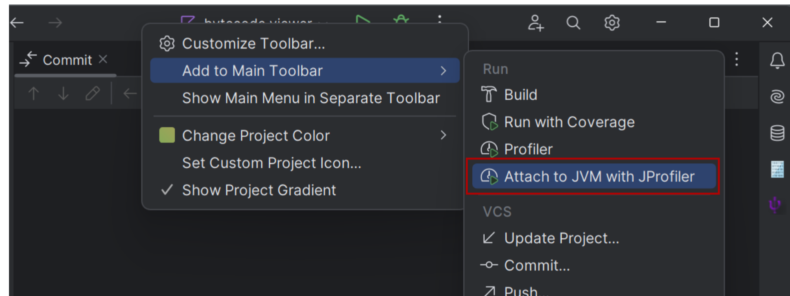

ガターアイコンをクリックすると、ポップアップウィンドウに着信または発信メソッドが表示され、記録された時間を示す棒グラフが表示されます。ポップアップ内の行をクリックすると、対応するメソッドに移動します。

また、ターゲットメソッドの総記録時間と呼び出し回数がポップアップの下部に表示されます。ポップアップの右下隅にある「JProfilerで表示」ドロップダウンは、JProfiler UIへのコンテキスト依存のナビゲーションアクションを提供します。選択したノードまたは対応する呼び出しツリー分析をメソッドグラフで表示できます。発信呼び出しの場合、「累積発信呼び出し」分析が提供され、着信呼び出しの場合は「バックトレース」分析が提供されます。

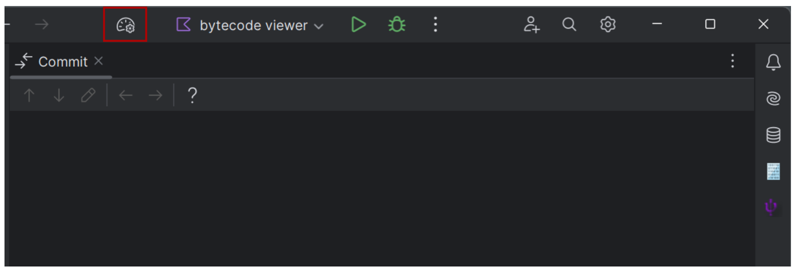
同じナビゲーションアクションは、「JProfiler」タブのメソッドテーブルのコンテキストメニューでも利用できます：



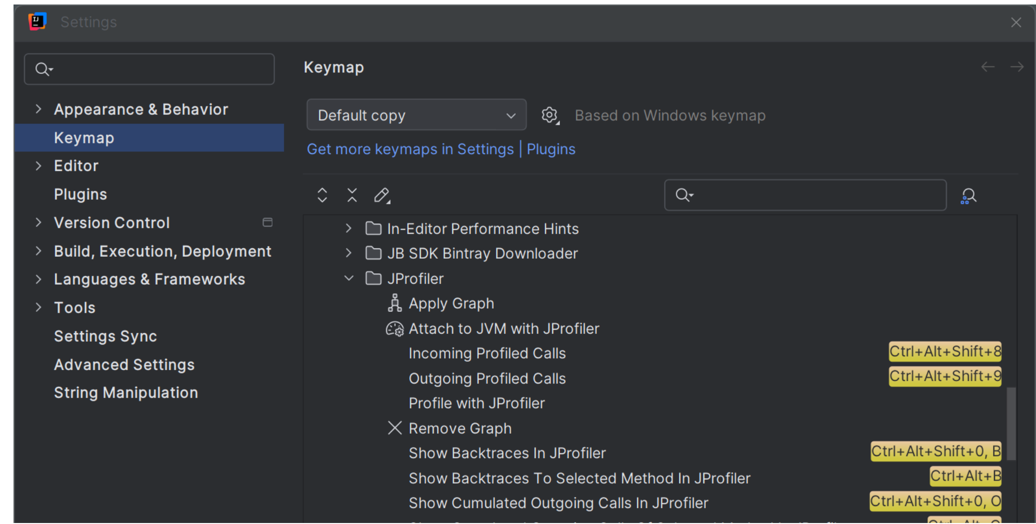

JProfilerプラグインは、「JProfilerでJVMにアタッチ」アクションのためのツールバークイックアクションを提供し、メインツールバーに追加できます。そのアクションを使用すると、すでに実行中のプロセスにアタッチし、JProfiler UIからIntelliJ IDEAへのソースコードナビゲーションを取得し、ソースコードエディタ内にインラインCPUグラフデータを取得できます：



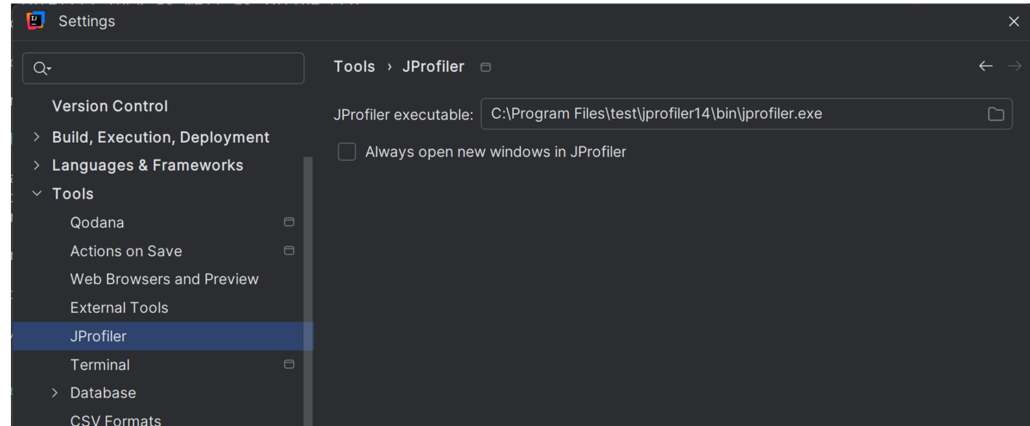

アクションボタンを追加すると、このように見えます：




JProfilerのすべてのアクションのキーのバインディングは、IntelliJ IDEAの「キーマップ」設定でカスタマイズできます。競合しないキーボードショートカットの限られた利用可能性を考慮すると、ソースコードエディタからJProfiler UIへのナビゲーションアクションは、最初に`Ctrl-Alt-Shift-O`を押してから、ナビゲーションアクションを選択するための別のキーを押すチェーンショートカットです。この機能を頻繁に使用する場合は、より簡単なキーボードショートカットを割り当てることを検討してください。



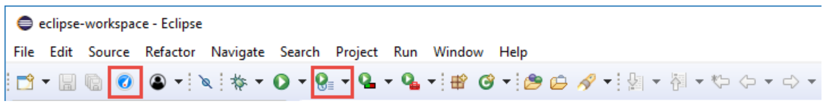

IDE設定のツール->JProfilerページで、使用するJProfiler実行可能ファイルと、新しいプロファイリングセッションのために常に新しいウィンドウをJProfilerで開くかどうかを調整できます。

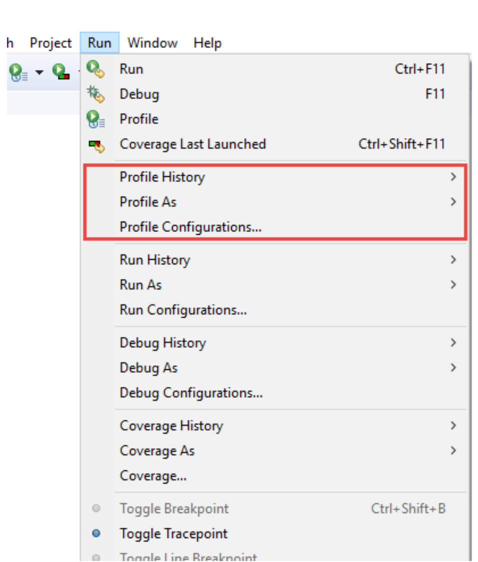



## eclipse統合

eclipseプラグインは、テスト実行構成やWTP実行構成を含む、最も一般的な起動構成タイプをプロファイルできます。eclipseプラグインは、eclipseフレームワークの部分的なインストールではなく、完全なeclipse SDKでのみ動作します。

eclipseからアプリケーションをプロファイルするには、実行メニューのプロファイリングコマンドのいずれかを選択するか、対応するツールバーボタンをクリックします。プロファイルコマンドは、eclipseのデバッグおよび実行コマンドと同等であり、eclipseのインフラストラクチャの一部ですが、実行->JProfilerをJVMにアタッチメニュー項目はJProfilerプラグインによって追加されます。



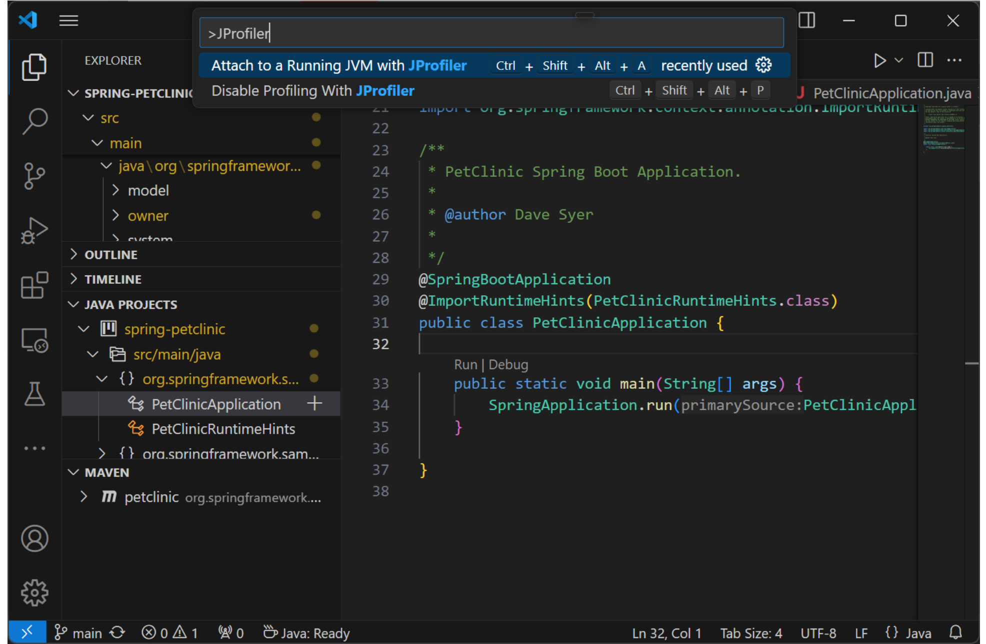



Javaパースペクティブに実行->プロファイル...メニュー項目が存在しない場合は、ウィンドウ->パースペクティブ->パースペクティブのカスタマイズでこのパースペクティブの「プロファイル」アクションを有効にし、アクションセットの可用性タブを前面に持ってきて、プロファイルチェックボックスを選択します。

JProfiler関連の設定のいくつかは、eclipseのウィンドウ->設定->JProfilerで調整できます。

## VS Code統合

VS Code拡張機能は、`JProfiler`　　　　　　　　　アクションを追加します。呼び出されると、デバッグおよび実行アクションはJava起動構成のプロファイリングを開始します。JProfilerが開始され、プロファイリング設定を構成できるセッションスタートアップダイアログが表示されます。セッションスタートアップダイアログが確認されると、アプリケーションが開始されます。



JProfiler アクションを使用すると、実行およびデバッグアクションのデフォルトの動作が復元されます。プロファイリングモードの変更に関する通知は、エディタの右下隅にトーストメッセージとしてVS Codeに表示されます。JProfiler とJProfiler は同じデフォルトのキーのバインディングを持っているため、プロファイリングモードを切り替えるために使用できます。

すでに実行中のJVMをプロファイルするには、JProfiler JVM アクションを使用します。

JProfilerのソースナビゲーションアクションは、対応するソースコードをVS Codeに表示します。JProfilerスナップショットのためにVS Codeにソースナビゲーションを取得するには、VS Code内からファイル->開くでスナップショットを開きます。

### NetBeans統合

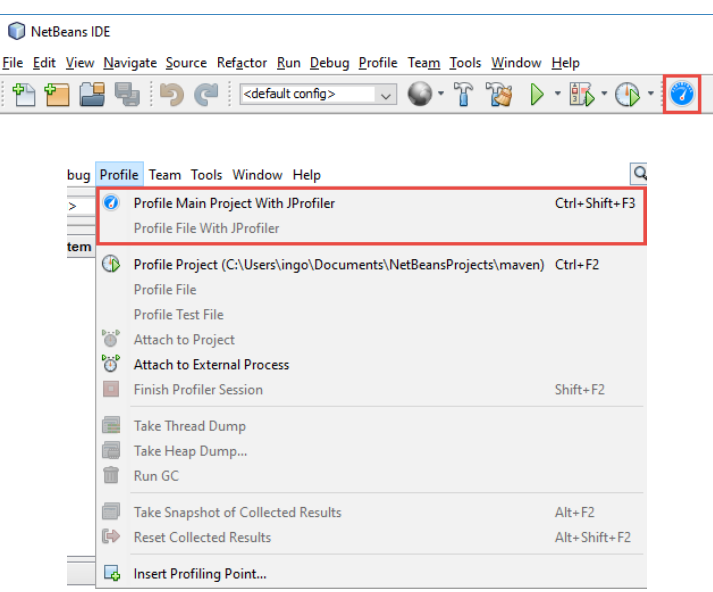
NetBeansでは、標準、フリーフォーム、およびexec Mavenプラグインを使用するMavenプロジェクトをプロファイルできます。NetBeansからアプリケーションをプロファイルするには、実行メニューのプロファイリングコマンドのいずれかを選択するか、対応するツールバーボタンをクリックします。別の方法でアプリケーションを開始するMavenプロジェクトやGradleプロジェクトの場合は、プロジェクトを通常どおり開始し、メニューのプロファイル->JProfilerを実行中のJVMにアタッチアクションを使用します。

フリーフォームプロジェクトの場合、プロファイルを試みる前に一度アプリケーションをデバッグする必要があります。必要なファイルnbproject/ide-targets.xmlはデバッグアクションによって設定されます。JProfilerはそれに「profile-jprofiler」という名前のターゲットを追加し、デバッグターゲットと同じ内容で、必要に応じてVMパラメータを変更しようとします。フリーフォームプロジェクトのプロファイリングに問題がある場合は、このターゲットの実装を確認してください。

統合されたTomcatまたはNetBeansで設定された他のTomcatサーバーでWebアプリケーションをプロファイルできます。メインプロジェクトがWebプロジェクトの場合、JProfilerでメインプロジェクトをプロファイルを選択すると、プロファイリングが有効になったTomcatサーバーが開始されます。

バンドルされたGlassFishサーバーを使用してNetBeansを使用している場合、メインプロジェクトがGlassFishサーバーを使用するように設定されている場合、JProfilerでメインプロジェクトをプロファイルを選択すると、プロファイリングが有効になったアプリケーションサーバーが開始されます。

JProfiler実行可能ファイルの場所と新しいJProfilerウィンドウを開くポリシーは、オプションダイアログのその他->JProfilerで調整できます。


NetBeans IDE
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
<default config>
bug Profile Team Tools Window Help
Profile Main Project With JProfiler Ctrl+Shift+F3
Profile File With JProfiler
Profile Project (C:\Users\ingo\Documents\NetBeansProjects\maven) Ctrl+F2
Profile File
Profile Test File
Attach to Project
Attach to External Process
Finish Profiler Session Shift+F2
Take Thread Dump
Take Heap Dump...
Run GC
Take Snapshot of Collected Results Alt+F2
Reset Collected Results Alt+Shift+F2
Insert Profiling Point...

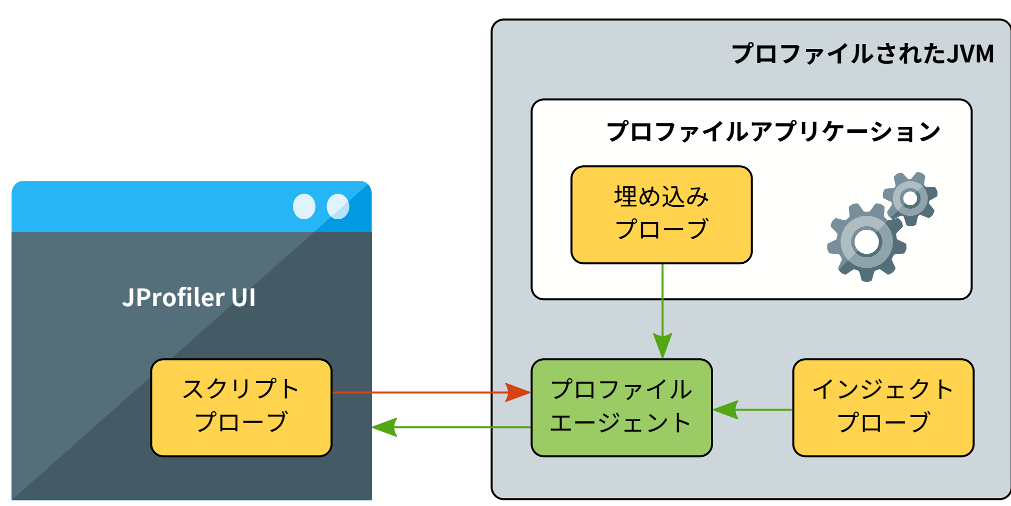
# A カスタムプローブ

## A.1 プローブの概念

JProfilerのカスタムプローブを開発するには、いくつかの基本的な概念と用語を理解しておく必要があります。JProfilerのすべてのプローブの共通の基盤は、特定のメソッドをインターセプトし、インターセプトしたメソッドのパラメータや他のデータソースを使用して、JProfiler UIで見たいと思う興味深い情報を含む文字列を構築することです。

プローブを定義する際の最初の問題は、インターセプトするメソッドをどのように指定し、文字列を構築するためにメソッドのパラメータや他の関連オブジェクトを使用できる環境をどのように得るかです。JProfilerでは、これを行うための3つの異なる方法があります：

- **スクリプトプローブ** [p. 159]は、完全にJProfiler UIで定義されます。呼び出しツリーでメソッドを右クリックし、スクリプトプローブアクションを選択し、組み込みのコードエディタで文字列の式を入力します。これはプローブを試すのに最適ですが、カスタムプローブの機能のごく一部しか公開しません。
- **埋め込みプローブ** [p. 168] APIは、自分のコードから呼び出すことができます。ライブラリ、データベースドライバ、またはサーバーを書く場合、プロダクトと一緒にプローブを出荷することができます。JProfilerでプロファイルされた誰でも、JProfiler UIに自動的にプローブが追加されます。
- **インジェクションプローブ** [p. 163] APIを使用して、IDEでサードパーティソフトウェア用のプローブを書くことができます。APIは、インターセプションを定義し、メソッドパラメータや他の有用なオブジェクトを注入するためにアノテーションを使用します。

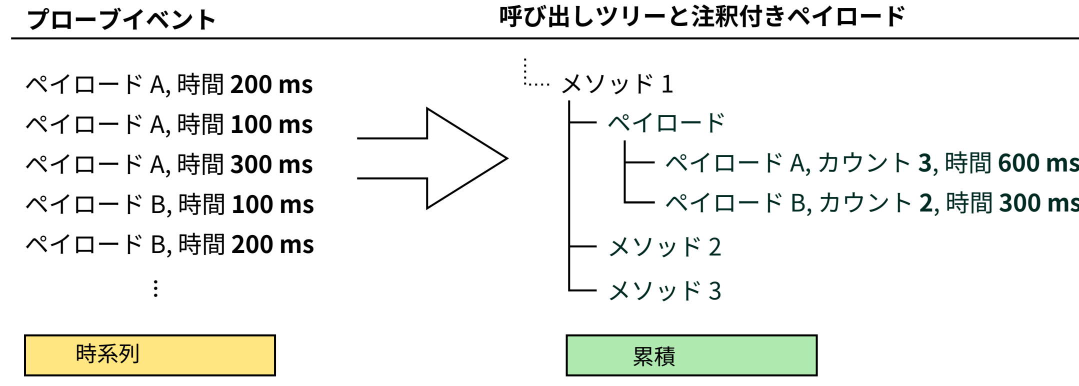



次の質問は、作成した文字列をJProfilerがどのように扱うべきかです。利用可能な2つの異なる戦略は、ペイロードの作成または呼び出しツリーの分割です。

#### ペイロードの作成

プローブによって構築された文字列は、**プローブイベント**を作成するために使用できます。イベントには、その文字列に設定された説明、インターセプトされたメソッドの呼び出し時間と等しい期

間、および関連する呼び出しスタックがあります。対応する呼び出しスタックで、プローブの説明とタイミングは累積され、**ペイロード**として呼び出しツリーに保存されます。イベントは一定の最大数を超えると統合されますが、呼び出しツリーの累積ペイロードは、記録期間全体の総数を示します。CPUデータとプローブの両方が記録されている場合、プローブの呼び出しツリービューはペイロード文字列をリーフノードとして持つマージされた呼び出しスタックを表示し、CPU呼び出しツリービューにはプローブ呼び出しツリービューへの**注釈付きリンク**が含まれます。




CPUデータと同様に、ペイロードは呼び出しツリーまたはホットスポットビューで表示できます。ホットスポットは、どのペイロードが最も多くの時間を費やしているかを示し、バックトレースはどの部分のコードがこれらのペイロードを作成しているかを示します。ホットスポットの良いリストを得るためには、ペイロード文字列に一意のIDやタイムスタンプを含めないようにする必要があります。すべてのペイロード文字列が異なる場合、累積も明確なホットスポットの分布もありません。たとえば、準備されたJDBCステートメントの場合、パラメータはペイロード文字列に含めるべきではありません。

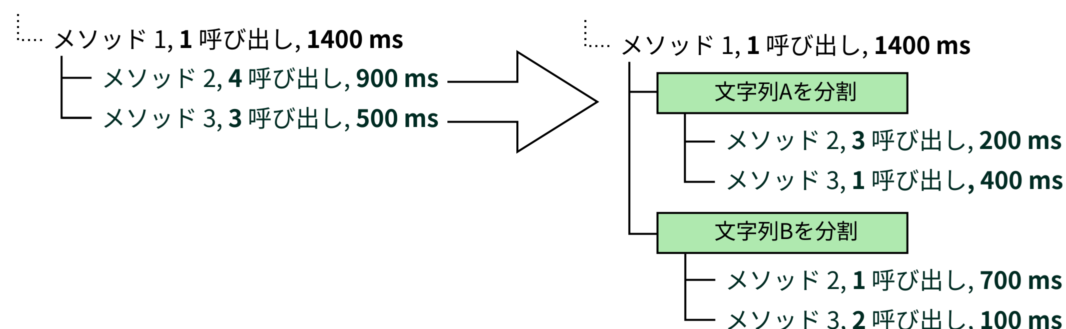
スクリプトプローブは、設定されたスクリプトの戻り値から自動的にペイロードを作成します。インジェクションプローブも同様で、`PayloadInterception`で注釈されたインターセプションハンドラメソッドから文字列または高度な機能のための`Payload`オブジェクトとしてペイロードの説明を返します。一方、埋め込みプローブは、`Payload.exit`をペイロードの説明を引数として呼び出すことによってペイロードを作成し、`Payload.enter`と`Payload.exit`の間の時間がプローブイベントの期間として記録されます。

ペイロードの作成は、異なる呼び出し元サイトで発生するサービスへの呼び出しを記録している場合に最も有用です。典型的な例は、ペイロード文字列がクエリ文字列やコマンドの形式であるデータベースドライバです。プローブは、測定された作業が別のソフトウェアコンポーネントによって実行される呼び出し元サイトの視点を取ります。

### 呼び出しツリーの分割

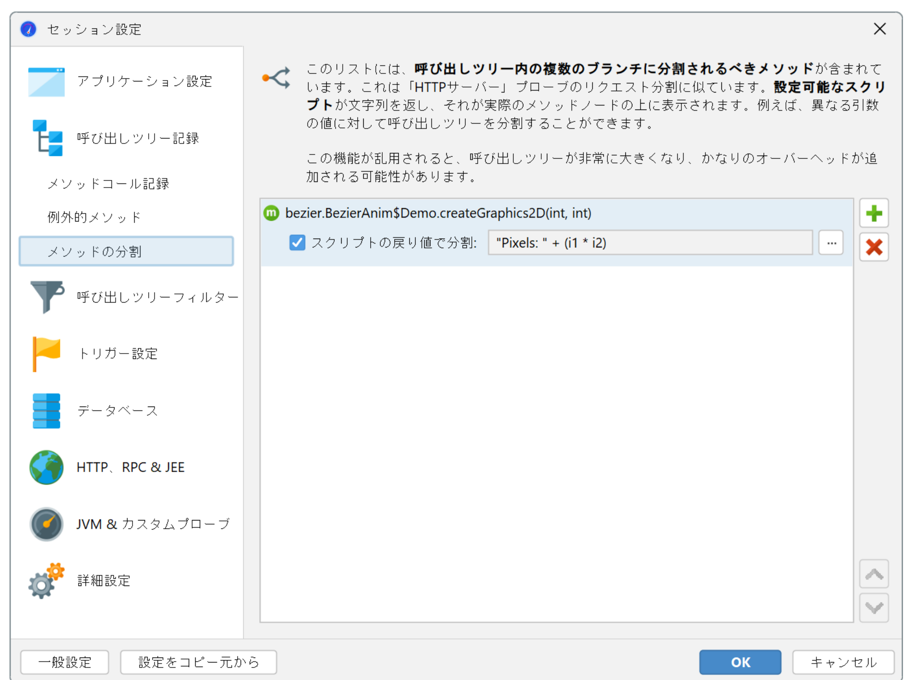
プローブはまた、実行サイトの視点を取ることができます。その場合、インターセプトされたメソッドがどのように呼び出されるかではなく、その後にどのメソッド呼び出しが実行されるかが重要です。典型的な例は、抽出された文字列がURLであるサーブレットコンテナのプローブです。

ペイロードを作成することよりも重要なのは、プローブによって構築された各異なる文字列に対して呼び出しツリーを分割する能力です。そのような文字列ごとに、対応する呼び出しの累積呼び出しツリーを含む分割ノードが呼び出しツリーに挿入されます。そうでなければ1つの累積呼び出しツリーしかないところに、呼び出しツリーを異なる部分に分割する分割ノードのセットがあり、個別に分析することができます。




複数のプローブがネストされた分割を生成することができます。単一のプローブはデフォルトで1つの分割レベルしか生成しませんが、スクリプトプローブではサポートされていない**リエントラント**として設定されている場合を除きます。

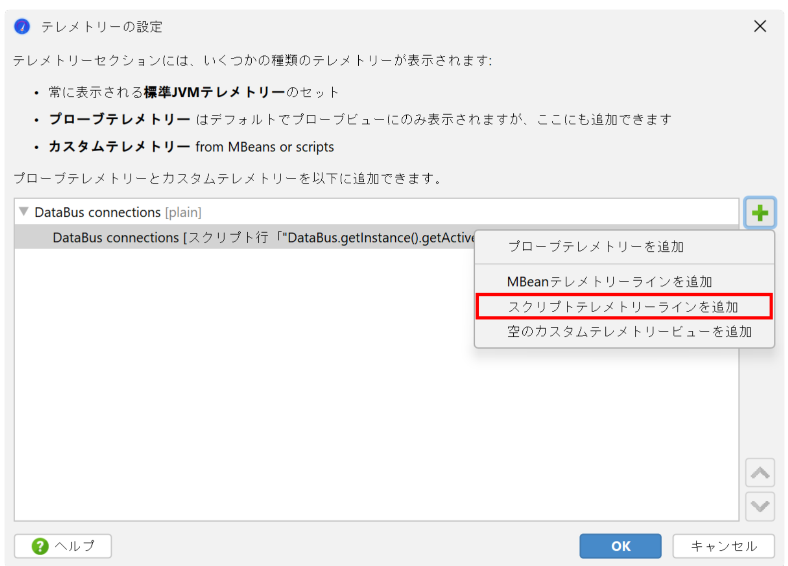
JProfiler UIでは、呼び出しツリーの分割はスクリプトプローブ機能とバンドルされていませんが、「メソッドの分割」[p. 186]と呼ばれる別の機能です。これらはペイロードを作成せずに呼び出しツリーを分割するだけなので、名前と説明が必要なプローブビューは必要ありません。インジェクションプローブは、`SplitInterception`で注釈されたインターセプションハンドラメソッドから分割文字列を返し、埋め込みプローブは分割文字列で`Split.enter`を呼び出します。




## テレメトリー

カスタムプローブには、イベント頻度と平均イベント期間の2つのデフォルトテレメトリーがあります。インジェクションプローブと埋め込みプローブは、プローブ設定クラスの注釈付きメソッドで作成される追加のテレメトリーをサポートしています。JProfiler UIでは、スクリプトテレメト

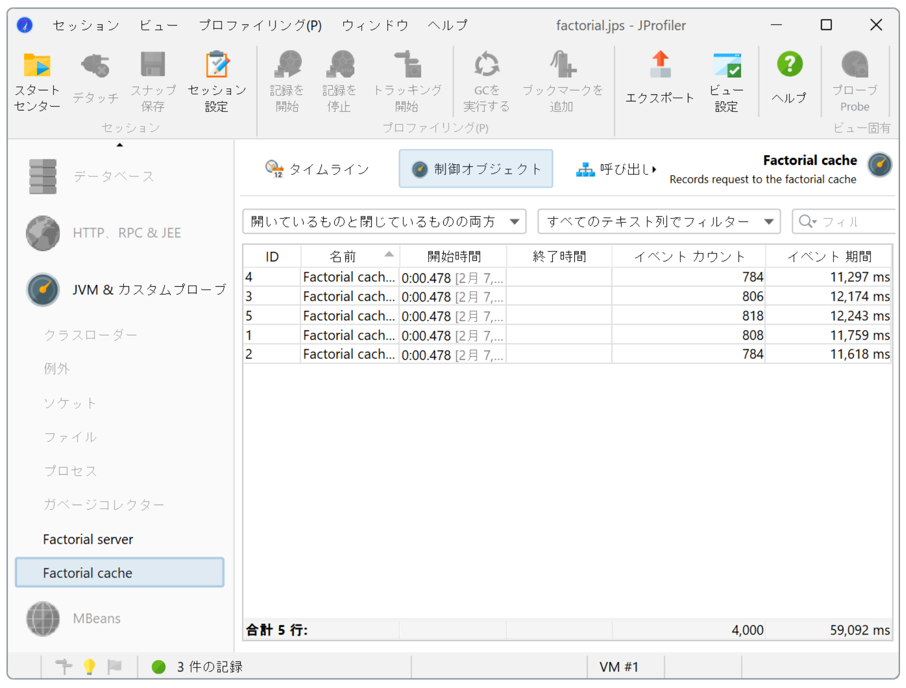
リーはスクリプトプローブ機能とは独立しており、ツールバーのテレメトリーの設定ボタンの下にある「テレメトリー」セクションにあります。




テレメトリーメソッドは1秒ごとにポーリングされます。`Telemetry`アノテーションでは、単位とスケールファクターを設定できます。`line`属性を使用すると、複数のテレメトリーを単一のテレメトリービューに結合できます。`TelemetryFormat`の`stacked`属性を使用すると、ラインを加算してスタックされたライングラフとして表示できます。埋め込みプローブとインジェクションプローブのテレメトリー関連APIは同等ですが、それぞれのプローブタイプにのみ適用されます。

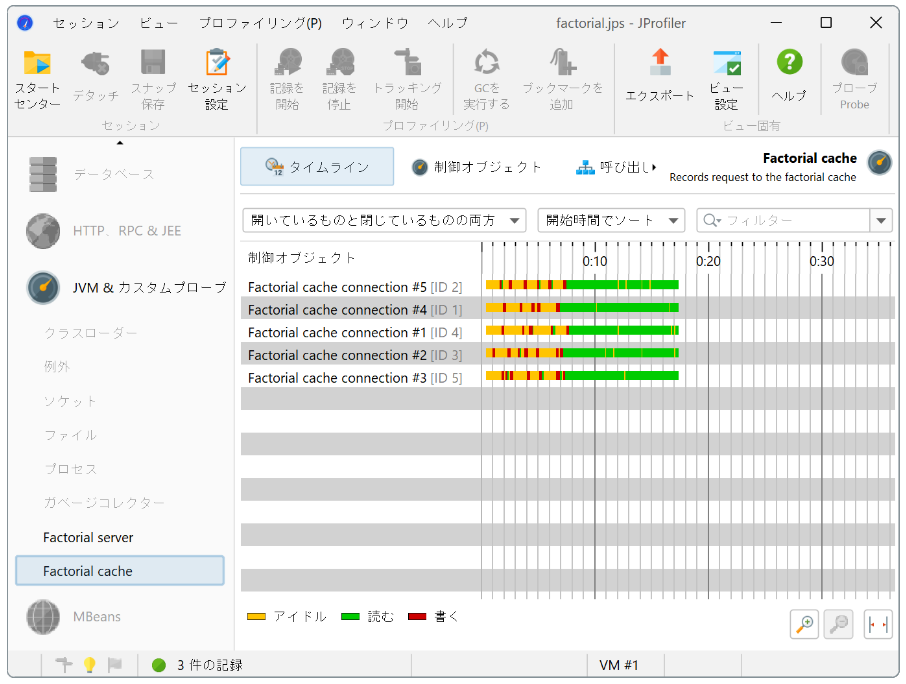
### 制御オブジェクト

時には、プローブイベントを関連する長寿命のオブジェクトに結びつけることが興味深いことがあります。JProfilerではこれを「制御オブジェクト」と呼びます。たとえば、データベースプローブでは、クエリを実行する物理接続がその役割を果たします。このような制御オブジェクトは、埋め込みAPIとインジェクションプローブAPIで開閉でき、プローブイベントビューで対応するイベントを生成します。プローブイベントが作成されるとき、制御オブジェクトを指定することができ、プローブの「制御オブジェクト」ビューで表示される統計に貢献します。



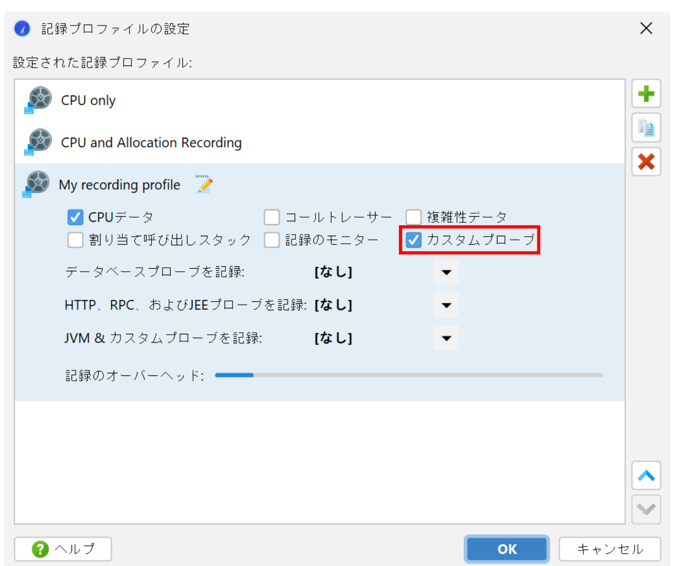

制御オブジェクトには、開かれたときに指定する必要がある表示名があります。新しい制御オブジェクトがプローブイベントの作成に使用される場合、プローブはその設定で名前リゾルバを提供する必要があります。

さらに、プローブはenumクラスを介してカスタムイベントタイプを定義できます。プローブイベントが作成されるとき、これらのタイプの1つを指定でき、イベントビューに表示され、単一のイベントタイプをフィルタリングできます。さらに重要なのは、時間軸上にラインとして制御オブジェクトを表示するプローブのタイムラインビューがイベントタイプに応じてカラーリングされることです。カスタムタイプのないプローブでは、イベントが記録されていないアイドル状態とプローブイベントの期間のデフォルトイベント状態が表示されます。カスタムタイプを使用すると、たとえば「読み取り」と「書き込み」のように状態を区別できます。

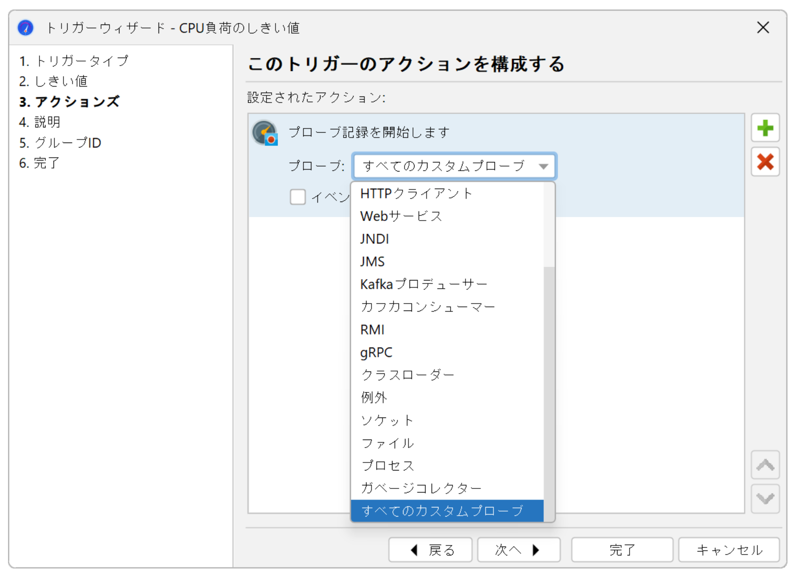



## 記録

すべてのプローブと同様に、カスタムプローブはデフォルトではデータを記録しませんが、必要に応じて記録を有効または無効にする必要があります。プローブビューで手動の開始/停止アクションを使用することもできますが、プローブ記録を開始する必要があることがよくあります。JProfilerはカスタムプローブを事前に知らないため、記録プロファイルにはすべてのカスタムプローブに適用されるカスタムプローブチェックボックスがあります。




同様に、プローブ記録を開始および停止するトリガーアクションにすべてのカスタムプローブを選択できます。




プログラムによる記録のために、すべてのカスタムプローブを記録するには`Controller.startProbeRecording(Controller.PROBE_NAME_ALL_CUSTOM, ProbeRecordingOptions.EVENTS)`を呼び出すか、より具体的にするためにプローブのクラス名を渡すことができます。

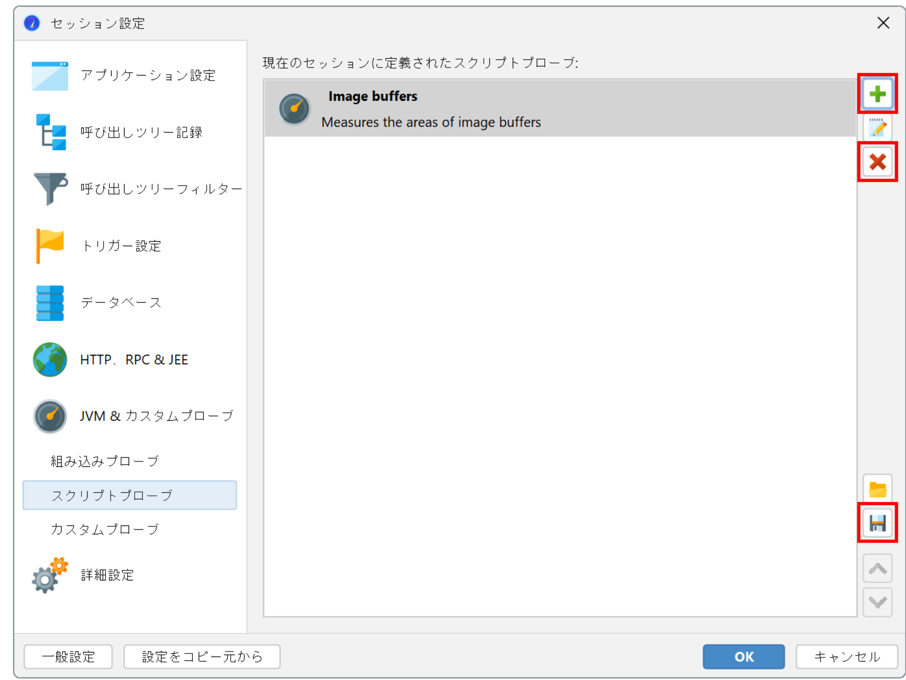
## A.2 スクリプトプローブ

IDEでカスタムプローブを開発するには、インターセプションポイントとプローブが提供する利点を明確に理解する必要があります。一方、スクリプトプローブを使用すると、JProfiler GUIで直接簡単なプローブを迅速に定義し、APIを学ぶことなく実験できます。埋め込みまたは注入されたカスタムプローブとは異なり、スクリプトプローブは実行中のプロファイリングセッション中に再定義でき、迅速な編集-コンパイル-テストループを実現します。

### スクリプトプローブの定義

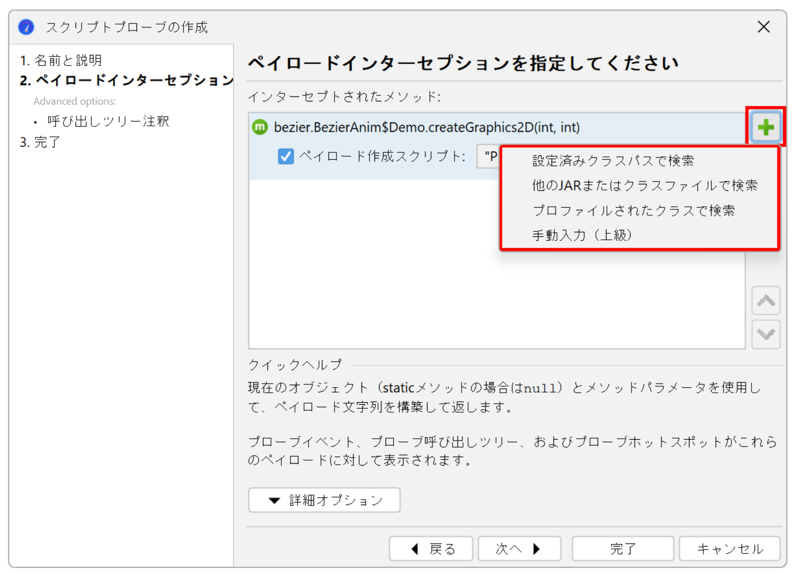
スクリプトプローブは、インターセプトされたメソッドを選択し、プローブのペイロード文字列を返すスクリプトを入力することで定義されます。このようなメソッドとスクリプトのペアを複数まとめて1つのプローブにすることができます。

スクリプトプローブの設定はセッション設定でアクセスできます。ここでスクリプトプローブを作成および削除し、他のプロファイリングセッションでインポートできるセットにスクリプトプローブを保存します。

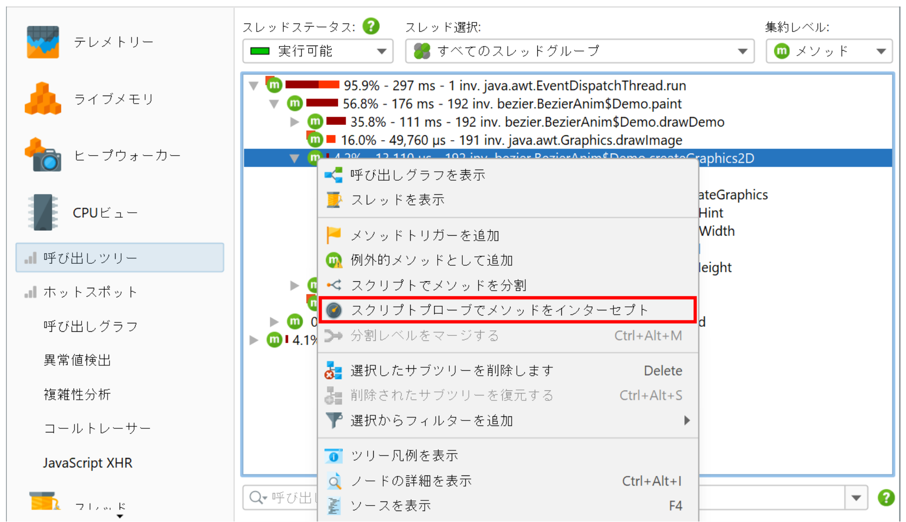



各スクリプトプローブには名前とオプションで説明が必要です。名前はJProfilerのビューセレクタの「JEE＆Probes」セクションにプローブビューを追加するために使用されます。説明はプローブビューのヘッダーに表示され、その目的を簡潔に説明する必要があります。

メソッドを選択するには、設定されたクラスパスからクラスを選択するか、プロファイルされたクラスから選択するなど、複数のオプションがあります。プロファイリングセッションがすでに実行中の場合は、選択したクラスからメソッドを選択できます。




インターセプトされたメソッドを選択するはるかに簡単な方法は、呼び出しツリービューからです。コンテキストメニューで、スクリプトプローブでメソッドをインターセプトアクションを選択すると、新しいプローブを作成するか、既存のプローブにインターセプションを追加するかを尋ねられます。

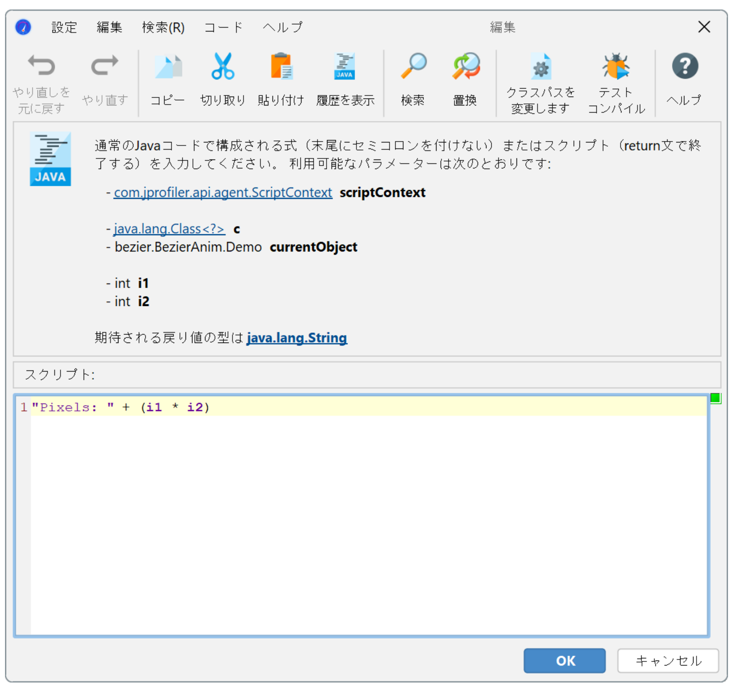



## プローブスクリプト

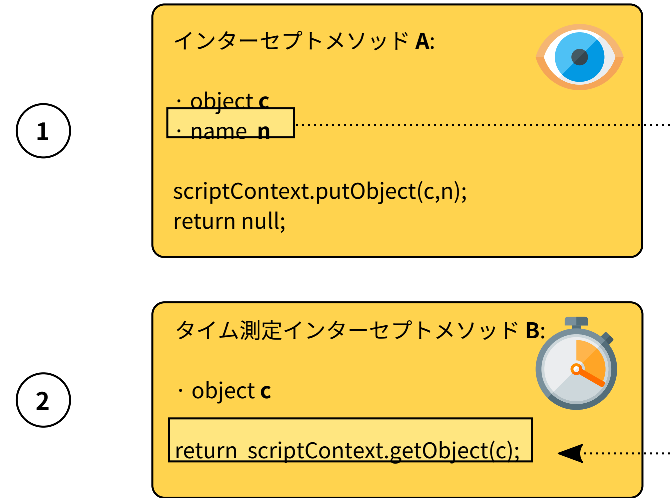
スクリプトエディタでは、インターセプトされたメソッドのすべてのパラメータと、メソッドが呼び出されたオブジェクトにアクセスできます。インターセプトされたメソッドの戻り値やスローされた例外にアクセスする必要がある場合は、埋め込みまたは注入されたプローブを書く必要があります。

この環境では、スクリプトはペイロード文字列を構築できます。これは、式として、またはreturn文を含むステートメントのシーケンスとして構築できます。このようなスクリプトの最も単純なバージョンは、1つのパラメータに対して`parameter.toString()`を返すか、プリミティブ型のパラメータに対して`String.valueOf(parameter)`を返します。`null`を返すと、ペイロードは作成されません。

CPUとプローブデータを同時に記録する場合、CPUセクションの呼び出しツリービューは、適切な呼び出しスタックでプローブビューへのリンクを表示します。あるいは、CPU呼び出しツリービューでペイロード文字列をインラインで表示することを選択できます。プローブウィザードの「ペイロードインターセプション->呼び出しツリー注釈」ステップにはこのオプションが含まれています。




スクリプトに利用可能なもう1つのパラメータはスクリプトコンテキストです。これは、現在のプローブに定義された任意のスクリプトの呼び出し間でデータを保存できる`com.jprofiler.api.agent.ScriptContext`型のオブジェクトです。たとえば、インターセプトされたメソッドAが良いテキスト表現を持たないオブジェクトのみを参照する場合、オブジェクトと表示名の関連付けはメソッドBをインターセプトすることで取得できます。その場合、同じプローブでメソッドBをインターセプトし、オブジェクトとテキストの関連付けをスクリプトコンテキストに直接保存できます。メソッドAでは、その表示テキストをスクリプトコンテキストから取得し、ペイロード文字列を構築するために使用します。




これらの種類の懸念があまりにも複雑になる場合は、埋め込みまたは注入されたプローブAPIに切り替えることを検討してください。

### 欠けている機能

スクリプトプローブはカスタムプローブ開発への簡単なエントリを促進するように設計されていますが、完全なプローブシステムからいくつかの機能が欠けていることに注意する必要があります:

- スクリプトプローブは呼び出しツリーの分割を行うことができません。JProfiler UIでは、これは で説明されているように別の機能です。埋め込みおよび注入されたプローブは、呼び出しツリー分割機能を直接提供します。
- スクリプトプローブは制御オブジェクトを作成したり、カスタムプローブイベントタイプを作成したりすることができません。これは埋め込みまたは注入されたプローブでのみ可能です。
- スクリプトプローブは戻り値やスローされた例外にアクセスすることができません。埋め込みおよび注入されたプローブとは異なります。
- スクリプトプローブは再入可能なインターセプションを処理できません。メソッドが再帰的に呼び出される場合、最初の呼び出しのみがインターセプトされます。埋め込みおよび注入されたプローブは、再入可能な動作に対する詳細な制御を提供します。
- デフォルトのテレメトリー以外のテレメトリーをプローブビューにバンドルすることはできません。代わりに、スクリプトテレメトリ機能を使用することができます。これは で示されています。

## A.3 インジェクトされたプローブ

スクリプトプローブと同様に、インジェクトされたプローブは選択されたメソッドのインターセプションハンドラを定義します。 しかし、インジェクトされたプローブはJProfiler GUIの外でIDE内で開発され、JProfilerが提供するインジェクトされたプローブAPIに依存します。 このAPIは寛容なApache License, version 2.0の下でライセンスされており、関連するアーティファクトを簡単に配布できます。

インジェクトされたプローブを始める最良の方法は、JProfilerインストールの`api/samples/simple-injected-probe`ディレクトリにある例を学ぶことです。そのディレクトリで`../gradlew`を実行してコンパイルし、実行します。gradleビルドファイル`build.gradle`にはサンプルに関するさらなる情報が含まれています。`api/samples/advanced-injected-probe`にある例は、制御オブジェクトを含むプローブシステムのより多くの機能を示しています。

### 開発環境

インジェクトされたプローブを開発するために必要なプローブAPIは、maven座標を持つ単一のアーティファクトに含まれています。

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

このマニュアルに対応するJProfilerのバージョンは15.0です。

Jar、ソース、およびjavadocアーティファクトは、次のリポジトリに公開されています。

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

GradleやMavenのようなビルドツールを使用して開発クラスパスにプローブAPIを追加するか、JARファイルを使用することができます。

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

JProfilerインストール内で。

Javadocを閲覧するには、次に移動します。

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

そのjavadocは、JProfilerによって公開されているすべてのAPIのjavadocを組み合わせたものです。`com.jprofiler.api.probe.injected`パッケージの概要は、APIを探索するための良い出発点です。

### プローブ構造

インジェクトされたプローブは、`com.jprofiler.api.probe.injected.Probe`で注釈されたクラスです。その注釈の属性は、プローブ全体の設定オプションを公開します。たとえば、個別の検査には興味がない多くのプローブイベントを作成する場合、 `events`属性を使用してプローブイベントビューを無効にし、オーバーヘッドを削減できます。

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

プローブクラスには、インターセプションハンドラを定義するために特別に注釈された静的メソッドを追加します。 PayloadInterception注釈はペイロードを作成し、SplitInterception注釈は呼び出しツリーを分割します。ハンドラの戻り値は、注釈に応じてペイロードまたは分割文字列として使用されます。スクリプトプローブと同様に、nullを返すと、インターセプションは効果がありません。タイミング情報はインターセプトされたメソッドに対して自動的に計算されます。

```
@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();
    }
}
```

上記の例でわかるように、両方の注釈には、MethodSpecを使用してインターセプトされたメソッドを定義するためのmethod属性があります。 スクリプトプローブとは異なり、MethodSpecはクラス名が空でもよく、特定のシグネチャを持つすべてのメソッドがクラス名に関係なくインターセプトされます。 または、MethodSpecのsubtypes属性を使用して、クラス階層全体をインターセプトすることもできます。

スクリプトプローブとは異なり、すべてのパラメータが自動的に利用可能である場合、ハンドラメソッドは関心のある値を要求するためにパラメータを宣言します。 各パラメータは、com.jprofiler.api.probe.injected.parameterパッケージの注釈で注釈されているため、プロファイリングエージェントはどのオブジェクトまたはプリミティブ値をメソッドに渡す必要があるかを知っています。たとえば、ハンドラメソッドのパラメータに@Parameter(0)を注釈すると、インターセプトされたメソッドの最初のパラメータが注入されます。

インターセプトされたメソッドのメソッドパラメータは、すべてのインターセプションタイプで利用可能です。ペイロードインターセプションは、メソッドのエントリではなく出口をインターセプトするようにプロファイリングエージェントに指示した場合、@ReturnValueまたは@ExceptionValueを使用して戻り値やスローされた例外にアクセスできます。 これは、PayloadInterception注釈のinvokeOn属性を使用して行います。

スクリプトプローブとは異なり、インジェクトされたプローブハンドラは、インターセプトされたメソッドの再帰的な呼び出しに対して呼び出されることができます。 インターセプション注釈のreentrant属性をtrueに設定した場合です。ハンドラメソッドにProbeContext型のパラメータ

を使用すると、ProbeContext.getOuterPayload()やProbeContext.restartTiming()を呼び出すことで、ネストされた呼び出しに対するプローブの動作を制御できます。

**高度なインターセプション**

プローブ文字列を構築するために必要なすべての情報を収集するには、単一のインターセプションでは不十分な場合があります。　　　　　そのため、プローブには、ペイロードや分割を作成しないInterceptionで注釈された任意の数のインターセプションハンドラを含めることができます。情報はプローブクラスの静的フィールドに保存できます。マルチスレッド環境でのスレッドセーフティのために、参照型を保存するためにThreadLocalインスタンスを使用し、 カウンタを維持するためにjava.util.concurrent.atomicパッケージのアトミック数値型を使用するべきです。

ある状況下では、メソッドのエントリとメソッドの出口の両方に対してインターセプションが必要です。一般的なケースは、inMethodCallのような状態変数を維持し、別のインターセプションの動作を変更する場合です。　デフォルトのインターセプションタイプであるエントリインターセプションでinMethodCallをtrueに設定できます。そのインターセプションの直下に別の静的メソッドを定義し、@AdditionalInterception(invokeOn = InvocationType.EXIT)で注釈します。インターセプトされたメソッドは上記のインターセプションハンドラから取得されるため、再度指定する必要はありません。　メソッドボディ内で、inMethodCall変数をfalseに設定できます。

```
...

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;
    }
}

...
```

このユースケースの動作する例は、api/samples/advanced-injected-probe/src/main/java/AdvancedAwtEventProbe.javaで見ることができます。

### 制御オブジェクト

Probe注釈のcontrolObjects属性がtrueに設定されていない限り、制御オブジェクトビューは表示されません。制御オブジェクトを操作するには、ハンドラメソッドにその型のパラメータを宣言してProbeContextを取得する必要があります。以下のサンプルコードは、制御オブジェクトを開いてプローブイベントと関連付ける方法を示しています。

```
@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);
    }

    ...

}
```

制御オブジェクトには定義された寿命があり、プローブビューはタイムラインと制御オブジェクトビューにその開閉時間を記録します。可能であれば、ProbeContext.openControlObject()とProbeContext.closeControlObject()を呼び出して制御オブジェクトを明示的に開閉するべきです。そうでない場合は、制御オブジェクトの表示名を解決する@ControlObjectNameで注釈された静的メソッドを宣言する必要があります。

プローブイベントは、ハンドラメソッドがStringの代わりにPayloadのインスタンスを返す場合、制御オブジェクトと関連付けることができます。 ProbeContext.createPayload()メソッドは、制御オブジェクトとプローブイベントタイプを受け取ります。許可されたイベントタイプを持つenumは、Probe注釈のcustomTypes属性で登録する必要があります。

制御オブジェクトは、メソッドエントリに対応する時間測定の開始時に指定する必要があります。場合によっては、ペイロード文字列の名前は戻り値や他のインターセプションに依存するため、メソッドの出口でのみ利用可能になります。 その場合、ProbeContext.createPayloadWithDeferredName()を使用して名前なしでペイロードオブジェクトを作成できます。@AdditionalInterception(invokeOn = InvocationType.EXIT)で注釈されたインターセプションハンドラを直下に定義し、そのメソッドからStringを返すと、 それが自動的にペイロード文字列として使用されます。

### スレッド状態のオーバーライド

データベースドライバや外部リソースへのネイティブコネクタの実行時間を測定する際、JProfilerにいくつかのメソッドを異なるスレッド状態に置くように指示する必要がある場合があります。たとえば、データベース呼び出しを「Net I/O」スレッド状態にすることが有用です。 通信メカニズ

ムが標準のJava I/O機能を使用せず、ネイティブメカニズムを使用する場合、これは自動的には行われません。

`ThreadState.NETIO.enter()`と`ThreadState.exit()`のペアを使用すると、プロファイリングエージェントはスレッド状態を適切に調整します。

```
...

@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();
}

...
```

**デプロイメント**

インジェクトされたプローブをデプロイする方法は2つあり、クラスパスに配置するかどうかによって異なります。 VMパラメータを使用して

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

プロファイリングエージェントによって別のプローブクラスパスが設定されます。 プローブクラスパスにはディレクトリとクラスファイルを含めることができ、Windowsでは';'、他のプラットフォームでは':'で区切ります。 プロファイリングエージェントはプローブクラスパスをスキャンし、すべてのプローブ定義を見つけます。

プローブクラスをクラスパスに配置する方が簡単な場合は、VMパラメータを設定できます。

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

完全修飾クラス名のカンマ区切りリストにします。これらのクラス名のそれぞれについて、プロファイリングエージェントはインジェクトされたプローブをロードしようとします。

## A.4 埋め込みプローブ

プローブのターゲットとなるソフトウェアコンポーネントのソースコードを制御できる場合は、注入プローブではなく埋め込みプローブを書くべきです。

注入プローブを書く際の最初の作業のほとんどは、インターセプトするメソッドを指定し、ハンドラーメソッドのメソッドパラメータとして注入オブジェクトを選択することに費やされます。埋め込みプローブでは、監視対象のメソッドから直接埋め込みプローブAPIを呼び出すことができるため、これが不要です。埋め込みプローブのもう一つの利点は、デプロイが自動であることです。プローブはソフトウェアと一緒に出荷され、アプリケーションがプロファイルされたときにJProfiler UIに表示されます。出荷する必要がある唯一の依存関係は、プロファイリングエージェントのフックとして機能する空のメソッドボディを主に含む、Apache 2.0ライセンスの小さなJARファイルです。

### 開発環境

開発環境は注入プローブと同じですが、アーティファクト名が`jprofiler-probe-embedded`であることと、プローブを別プロジェクトで開発する代わりにアプリケーションと一緒にJARファイルを出荷することが異なります。ソフトウェアコンポーネントに埋め込みプローブを追加するために必要なプローブAPIは、単一のJARアーティファクトに含まれています。javadocでは、APIを探索する際に`com.jprofiler.api.probe.embedded`のパッケージ概要から始めてください。

注入プローブと同様に、埋め込みプローブにも2つの例があります。`api/samples/simple-embedded-probe`には、埋め込みプローブの書き方を始めるための例があります。そのディレクトリで`../gradlew`を実行してコンパイルおよび実行し、実行環境を理解するためにgradleビルドファイル`build.gradle`を学習してください。制御オブジェクトを含むより多くの機能については、`api/samples/advanced-embedded-probe`の例を参照してください。

### ペイロードプローブ

注入プローブと同様に、構成目的でプローブクラスが必要です。プローブクラスは、ペイロードを収集するか呼び出しツリーを分割するかに応じて、`com.jprofiler.api.probe.embedded.PayloadProbe`または`com.jprofiler.api.probe.embedded.SplitProbe`を拡張する必要があります。注入プローブAPIでは、ハンドラーメソッドに対してペイロード収集と分割のための異なるアノテーションを使用します。一方、埋め込みプローブAPIにはハンドラーメソッドがなく、この構成をプローブクラス自体に移す必要があります。

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

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

注入プローブが構成にアノテーションを使用するのに対し、埋め込みプローブはプローブの基底クラスからメソッドをオーバーライドすることで構成します。ペイロードプローブの場合、唯一の抽象メソッドは`getName()`であり、他のすべてのメソッドには必要に応じてオーバーライドできるデフォルトの実装があります。たとえば、オーバーヘッドを減らすためにイベントビューを無効にしたい場合は、`isEvents()`をオーバーライドして`false`を返すことができます。

ペイロードを収集し、それに関連するタイミングを測定するには、`Payload.enter()`と`Payload.exit()`のペアを使用します。

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

Payload.enter()呼び出しはプローブクラスを引数として受け取り、プロファイリングエージェントがどのプローブが呼び出しのターゲットであるかを知ることができ、Payload.exit()呼び出しは自動的に同じプローブに関連付けられ、ペイロード文字列を引数として受け取ります。exit呼び出しを逃すと、呼び出しツリーが壊れるので、これは常にtryブロックのfinally句で行うべきです。

測定されたコードブロックが値を生成しない場合は、ペイロード文字列とRunnableを取るPayload.executeメソッドを呼び出すことができます。Java 8+では、ラムダやメソッド参照を使用してこのメソッド呼び出しを非常に簡潔にすることができます。

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

その場合、ペイロード文字列は事前に知られている必要があります。Callableを取るexecuteのバージョンもあります。

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

Callableを取るシグネチャの問題は、Callable.call()がチェックされたExceptionをスローするため、それをキャッチするか、含むメソッドで宣言する必要があることです。

#### 制御オブジェクト

ペイロードプローブは、Payloadクラスの適切なメソッドを呼び出すことで制御オブジェクトを開閉できます。これらは、制御オブジェクトとカスタムイベントタイプを取るPayload.enter()またはPayload.execute()メソッドのバージョンに渡すことでプローブイベントに関連付けられます。

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

制御オブジェクトビューはプローブ構成で明示的に有効にする必要があり、カスタムイベントタイプもプローブクラスで登録する必要があります。

```
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;
    }
}
```

制御オブジェクトを明示的に開閉しない場合、プローブクラスはすべての制御オブジェクトの表示名を解決するためにgetControlObjectNameをオーバーライドする必要があります。

#### 分割プローブ

分割プローブの基底クラスには抽象メソッドがありません。これは、プローブビューを追加せずに呼び出しツリーを分割するためだけに使用できます。その場合、最小限のプローブ定義は次のようになります。

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

分割プローブの重要な構成の1つは、それが再入可能であるべきかどうかです。デフォルトでは、トップレベルの呼び出しのみが分割されます。再帰呼び出しも分割したい場合は、isReentrant()をオーバーライドしてtrueを返します。分割プローブはまた、プローブビューを作成し、分割文字列をペイロードとして公開することもできます。プローブクラスでisPayloads()をオーバーライドしてtrueを返す場合です。

分割を実行するには、Split.enter()とSplit.exit()のペアを呼び出します。

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

ペイロード収集とは異なり、分割文字列はプローブクラスと一緒にSplit.enter()メソッドに渡す必要があります。再び、Split.exit()が確実に呼び出されることが重要であるため、tryブロックのfinally句にあるべきです。Splitはまた、RunnableおよびCallable引数を持つexecute()メソッドを提供し、単一の呼び出しで分割を実行します。

#### テレメトリー

埋め込みプローブのテレメトリーを公開することは特に便利です。同じクラスパスにあるため、アプリケーション内のすべての静的メソッドに直接アクセスできます。注入プローブと同様に、プローブ構成クラスの静的なパブリックメソッドに`@Telemetry`を注釈し、数値を返します。詳細については、プローブの概念[p.152]の章を参照してください。埋め込みプローブAPIと注入プローブAPIの`@Telemetry`アノテーションは同等であり、異なるパッケージにあるだけです。

埋め込みプローブAPIと注入プローブAPIのもう一つの並行機能は、`ThreadState`クラスを使用してスレッド状態を変更する機能です。再び、クラスは異なるパッケージで両方のAPIに存在します。

#### デプロイメント

JProfiler UIでプロファイリングする際に埋め込みプローブを有効にするための特別な手順は必要ありません。ただし、最初の`Payload`または`Split`への呼び出しが行われたときにのみプローブが登録されます。その時点でのみ、関連するプローブビューがJProfilerに作成されます。組み込みおよび注入プローブの場合のように、最初からプローブビューが表示されることを好む場合は、次のように呼び出すことができます。

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

ペイロードプローブの場合

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

分割プローブの場合。

`Payload`および`Split`のメソッドを条件付きで呼び出すかどうかを検討しているかもしれません。オーバーヘッドを最小限に抑えるためにコマンドラインスイッチで制御するかもしれません。しかし、メソッドボディが空であるため、これは一般的には必要ありません。プロファイリングエージェントがアタッチされていない場合、ペイロード文字列の構築を除いてオーバーヘッドは発生しません。プローブイベントは微視的なスケールで生成されるべきではないことを考慮すると、それらは比較的まれに作成されるため、ペイロード文字列の構築は比較的無視できる努力であるべきです。

コンテナにとってのもう一つの懸念は、クラスパス上の外部依存関係を公開したくないかもしれないということです。コンテナのユーザーが埋め込みプローブAPIを使用することもでき、それが競合を引き起こす可能性があります。その場合、埋め込みプローブAPIを独自のパッケージにシェードすることができます。JProfilerはシェードされたパッケージを認識し、APIクラスを正しくインストゥルメントします。ビルド時のシェードが実用的でない場合は、ソースアーカイブを抽出し、クラスをプロジェクトの一部にすることができます。

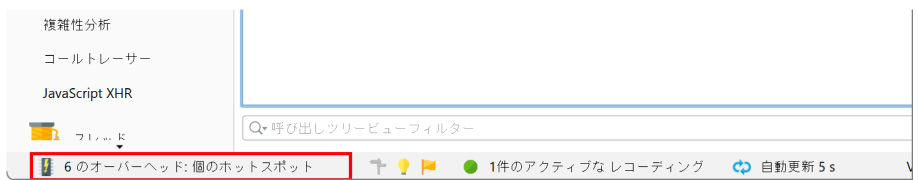
# B 呼び出しツリーの詳細機能

## B.1 自動チューニングと無視されたメソッド

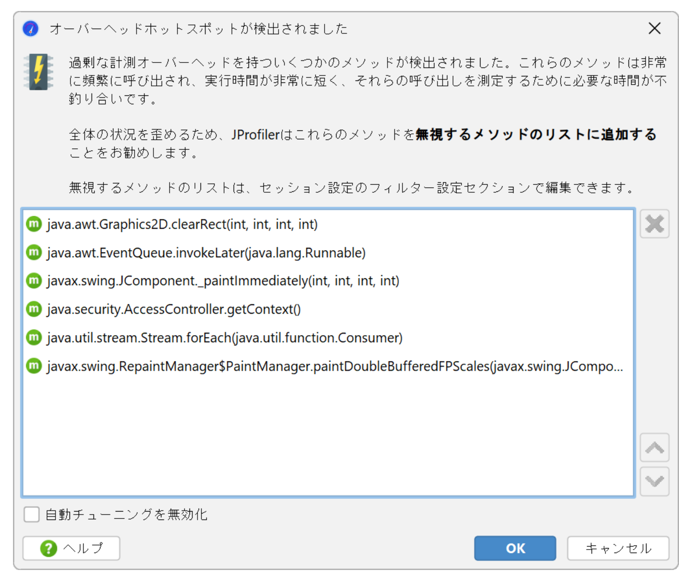
メソッドコール記録タイプがインストルメンテーションに設定されている場合、プロファイルされたクラスのすべてのメソッドがインストルメントされます。これにより、非常に短い実行時間を持つメソッドに対して大きなオーバーヘッドが発生します。もしそのようなメソッドが非常に頻繁に呼び出されると、測定されたそのメソッドの時間は非常に高くなりすぎます。また、インストルメンテーションのために、ホットスポットコンパイラがそれらを最適化するのを妨げる可能性があります。極端な場合、これらのメソッドは支配的なホットスポットとなりますが、これはインストルメントされていない実行では当てはまりません。例としては、次の文字を読み取るXMLパーサーのメソッドがあります。このようなメソッドは非常に迅速に戻りますが、短時間で何百万回も呼び出される可能性があります。

メソッドコール記録タイプがサンプリングに設定されている場合、この問題は発生しません。しかし、サンプリングは呼び出し回数を提供せず、長いメソッドコールのみを表示し、サンプリングを使用するといくつかのビューは完全な機能を持ちません。

インストルメンテーションの問題を軽減するために、JProfilerには自動チューニングと呼ばれるメカニズムがあります。プロファイリングエージェントは時折、高いインストルメンテーションオーバーヘッドを持つメソッドをチェックし、それらをJProfiler GUIに送信します。ステータスバーには、オーバーヘッドホットスポットの存在を知らせるエントリが表示されます。

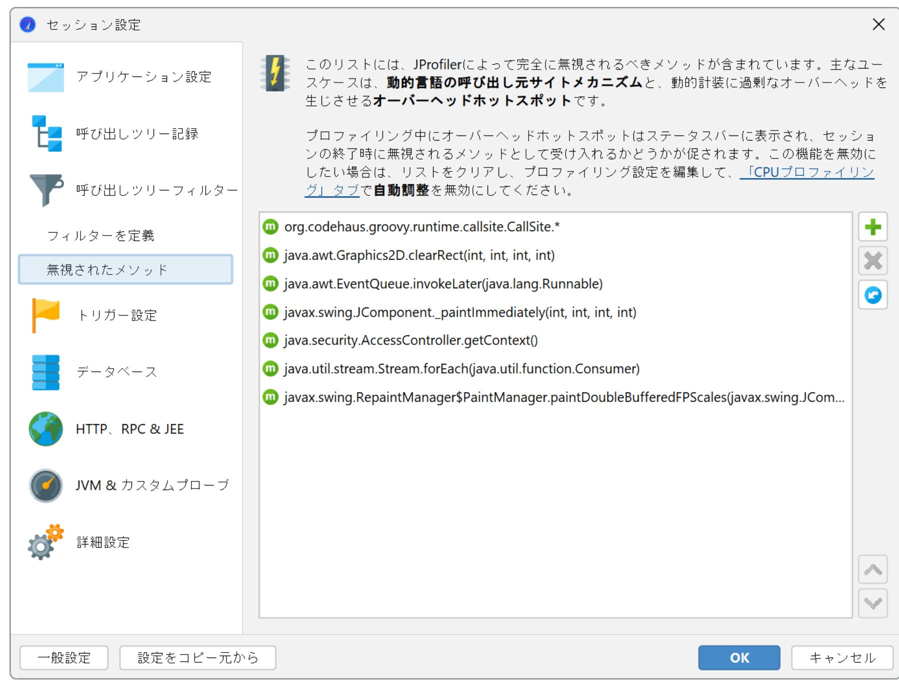



そのステータスバーエントリをクリックすると、検出されたオーバーヘッドホットスポットを確認し、無視されたメソッドのリストに受け入れることができます。これらの無視されたメソッドはインストルメントされません。セッションが終了すると、同じダイアログが表示されます。



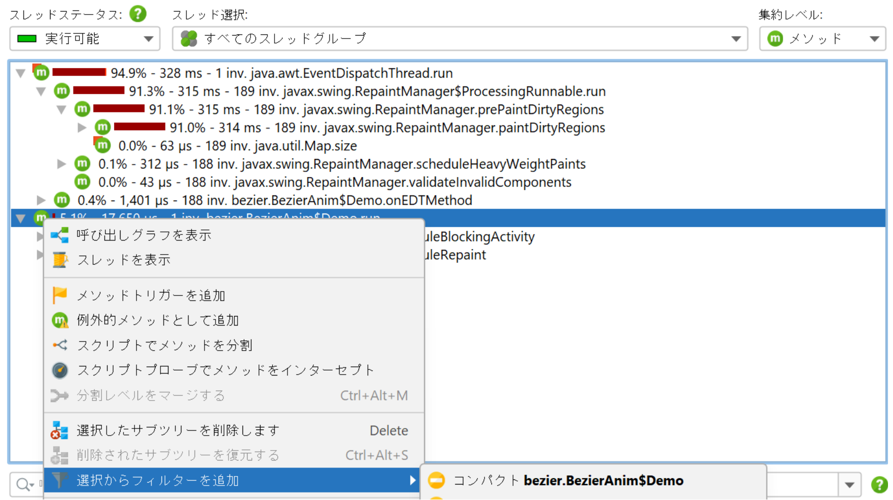

新しいプロファイリング設定を適用すると、すべての無視されたメソッドは呼び出しツリーから欠落します。それらの実行時間は呼び出し元メソッドの自己時間に追加されます。後で無視されたメ

ソッドがプロファイリングビューで不可欠であることが判明した場合は、セッション設定の無視されたメソッドタブでそれらを削除できます。




無視されたメソッドのデフォルト設定には、動的メソッドディスパッチに使用されるGroovyの呼び出し元サイトクラスが含まれていますが、　実際の呼び出しチェーンを追跡するのが難しくなります。

手動で無視されたメソッドを追加したい場合は、セッション設定で行うことができますが、はるかに簡単な方法は、呼び出しツリーでメソッドを選択し、コンテキストメニューからメソッドを無視アクションを呼び出すことです。




フィルター設定では、フィルターエントリのタイプを「無視」に設定することで、クラス全体やパッケージ全体を無視することもできます。選択からフィルターを追加メニューには、選択されたノードに依存するアクションが含まれており、クラスやトップレベルパッケージまでのパッケージ

を無視することを提案します。選択されたノードがコンパクトプロファイルされているかプロファイルされているかに応じて、フィルターを反対のタイプに変更するアクションも表示されます。

自動チューニングに関するメッセージを表示したくない場合は、プロファイリング設定で無効にすることができます。また、オーバーヘッドホットスポットを決定するための基準を設定することもできます。メソッドがオーバーヘッドホットスポットと見なされるのは、以下の条件の両方を満たす場合です：

- すべての呼び出しの合計時間がスレッド全体の合計時間のパーミルでしきい値を超える場合
- 平均時間がマイクロ秒単位の絶対しきい値より低い場合

## B.2 非同期およびリモートリクエストトラッキング

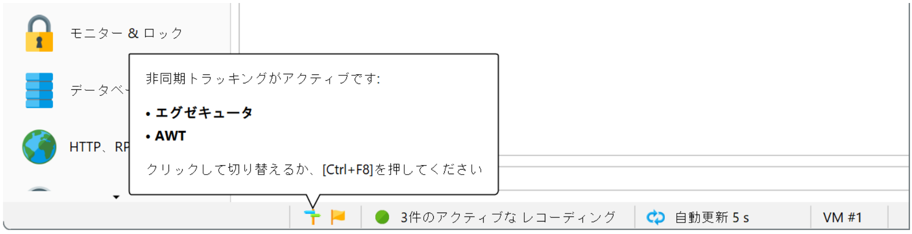
タスクの非同期実行は、プレーンなJavaコードでも、さらにリアクティブフレームワークでも一般的な手法です。ソースファイルで隣接しているコードが、今や2つ以上の異なるスレッドで実行されます。デバッグとプロファイリングにおいて、これらのスレッド変更は2つの問題を引き起こします。一方では、呼び出された操作がどれほど高価であるかが不明です。他方では、高価な操作を引き起こしたコードに遡ることができません。

JProfilerは、この問題に対して、呼び出しが同じJVM内で行われるかどうかに応じて異なる解決策を提供します。非同期実行がそれを呼び出す同じJVM内で行われる場合、「インライン非同期実行」呼び出しツリー分析[p.190]は、呼び出し元サイトと実行サイトの両方を含む単一の呼び出しツリーを計算します。リモートJVMへのリクエストが行われた場合、呼び出しツリー[p.54]には呼び出し元サイトと実行サイトへのハイパーリンクが含まれており、関係するJVMのプロファイリングセッションを表示する異なるJProfilerのトップレベルウィンドウ間をシームレスにナビゲートできます。

### 非同期およびリモートリクエストトラッキングの有効化

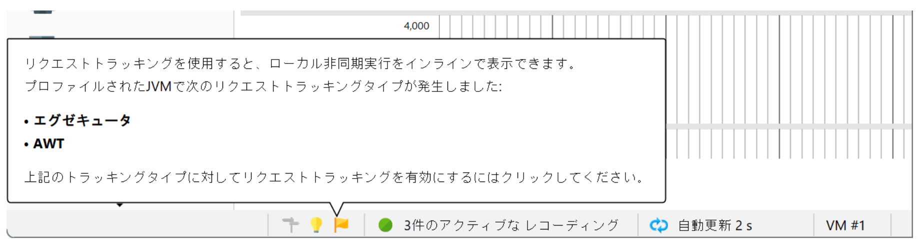
非同期メカニズムはさまざまな方法で実装でき、別のスレッドまたは異なるJVMでタスクを開始するセマンティクスは一般的な方法では検出できません。JProfilerは、いくつかの一般的な非同期およびリモートリクエスト技術を明示的にサポートしています。リクエストトラッキング設定でそれらを有効または無効にすることができます。デフォルトでは、リクエストトラッキングは有効になっていません。また、セッションが開始される直前に表示されるセッション開始ダイアログでリクエストトラッキングを設定することも可能です。

JProfilerのメインウィンドウでは、ステータスバーがいくつかの非同期およびリモートリクエストトラッキングタイプが有効になっているかどうかを示し、設定ダイアログへのショートカットを提供します。



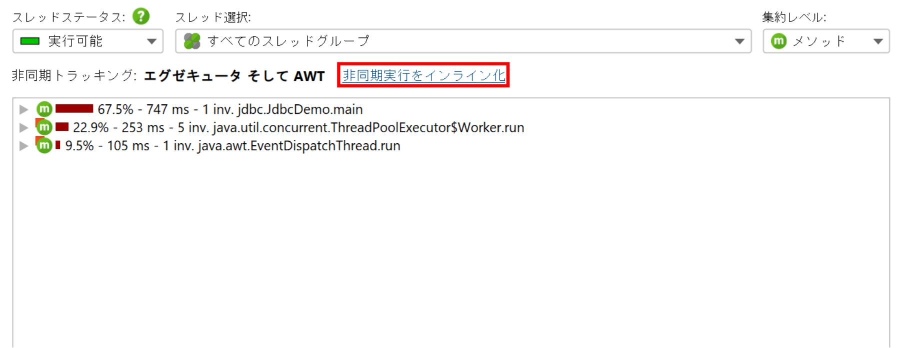

JProfilerは、プロファイルされたJVMで有効化されていない非同期リクエストトラッキングタイプが使用されている場合、 通知アイコンをステータスバーの非同期およびリモートリクエストトラッキングアイコンの横に表示します。通知アイコンをクリックすることで、検出されたトラッキ

ングタイプを有効化できます。非同期およびリモートリクエストトラッキングは大きなオーバーヘッドを生じる可能性があるため、必要な場合にのみ有効化するべきです。




## 非同期トラッキング

少なくとも1つの非同期トラッキングタイプが有効化されている場合、CPU、割り当て、プローブ記録の呼び出しツリーおよびホットスポットビューは、すべての有効化されたトラッキングタイプに関する情報を表示し、「インライン非同期実行」呼び出しツリー分析を計算するボタンを提供します。その分析の結果ビューでは、すべての非同期実行の呼び出しツリーが、 「非同期実行」ノードを介して呼び出し元サイトと接続されています。デフォルトでは、非同期実行の測定値は呼び出しツリーの祖先ノードに追加されません。集約された値を見ることが有用な場合があるため、分析の上部にあるチェックボックスで適切な場合にそれを行うことができます。




別のスレッドでタスクをオフロードする最も簡単な方法は、新しいスレッドを開始することです。JProfilerを使用すると、「スレッド開始」リクエストトラッキングタイプを有効化することで、スレッドの作成から実行サイトまでを追跡できます。ただし、スレッドは重量級オブジェクトであり、通常は繰り返し呼び出しのために再利用されるため、このリクエストトラッキングタイプはデバッグ目的でより有用です。

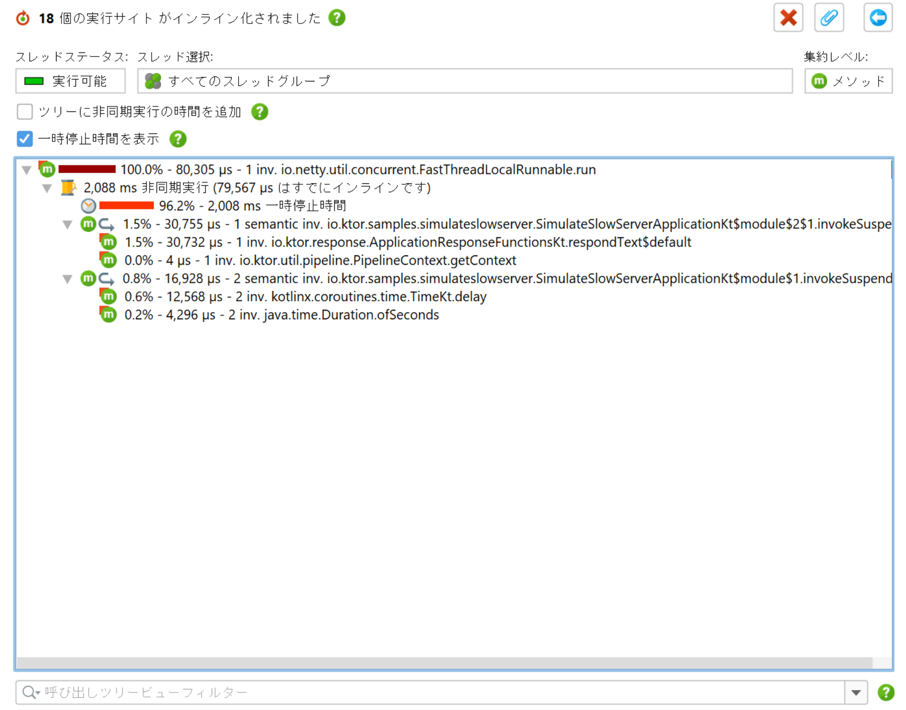
他のスレッドでタスクを開始する最も重要で一般的な方法は、`java.util.concurrent`パッケージのエグゼキュータを使用することです。エグゼキュータは、非同期実行を扱う多くの高レベルのサードパーティライブラリの基盤でもあります。エグゼキュータをサポートすることで、JProfilerはマルチスレッドおよび並列プログラミングを扱うライブラリ全体をサポートします。

上記の一般的なケースに加えて、JProfilerはJVM用の2つのGUIツールキット、AWTとSWTもサポートしています。どちらのツールキットもシングルスレッドであり、GUIウィジェットを操作し描画操作を行うことができる特別なイベントディスパッチスレッドがあります。GUIをブロックしないようにするために、長時間実行されるタスクはバックグラウンドスレッドで実行する必要があります。ただし、バックグラウンドスレッドは進行状況や完了を示すためにGUIを更新する必要があることが多いです。これは、イベントディスパッチスレッドで実行されるように`Runnable`をスケジュールする特別なメソッドで行われます。

GUIプログラミングでは、原因と結果を結びつけるために複数のスレッド変更を追跡する必要があることがよくあります。ユーザーがイベントディスパッチスレッドでアクションを開始し、それがエグゼキュータを介してバックグラウンド操作を開始します。完了後、そのエグゼキュータはイベ

ントディスパッチスレッドに操作をプッシュします。最後の操作がパフォーマンスの問題を引き起こす場合、それは元のイベントから2つのスレッド変更が離れています。

最後に、JProfilerはKotlinコルーチン [(1)] もサポートしています。Kotlinのマルチスレッドソリューションは、すべてのKotlinバックエンドで実装されています。非同期実行自体はコルーチンが起動されるポイントです。Kotlinコルーチンのディスパッチメカニズムは柔軟で、実際には現在のスレッドで開始することも含まれます。この場合、「非同期実行」ノードにはインライン部分があり、そのノードのテキストで別途報告されます。




サスペンドメソッドは実行を中断し、その後異なるスレッドで再開される可能性があります。サスペンドが検出されたメソッドには、追加の「サスペンド」アイコンがあり、ツールチップには実際の呼び出し数とメソッドのセマンティックな呼び出し数が表示されます。Kotlinコルーチンは意図的にサスペンドすることができますが、スレッドにバインドされていないため、待機時間は呼び出しツリーのどこにも表示されません。コルーチンの実行が完了するまでにかかる総時間を確認するために、「非同期実行」ノードの下に「サスペンド」時間ノードが追加され、コルーチンの全サスペンド時間をキャプチャします。非同期実行のCPU時間またはウォールクロック時間に興味があるかどうかに応じて、分析の上部にある「サスペンド時間を表示」チェックボックスでこれらのノードをオンザフライで追加または削除できます。

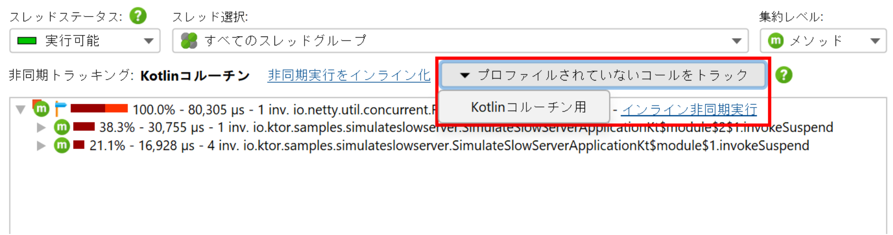
### プロファイルされていない呼び出し元サイトのトラッキング

デフォルトでは、エグゼキュータとKotlinコルーチントラッキングの両方が、呼び出し元サイトがプロファイルされたクラスにある非同期実行のみをトラッキングします。これは、フレームワークやライブラリがこれらの非同期メカニズムをあなた自身のコードの実行に直接関連しない方法で使用する可能性があり、追加された呼び出し元および実行サイトがオーバーヘッドと混乱を引き起こすだけだからです。ただし、プロファイルされていない呼び出し元サイトをトラッキングするユースケースもあります。たとえば、フレームワークがKotlinコルーチンを開始し、その上であなた自身のコードが実行される場合です。

プロファイルされていないクラスでそのような呼び出し元サイトが検出された場合、呼び出しツリーおよびホットスポットビューのトラッキング情報には対応する通知メッセージが表示されま

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

す。ライブセッションでは、これらのビューから直接エグゼキュータおよびKotlinコルーチントラッキングのプロファイルされていない呼び出し元サイトのトラッキングを個別にオンにすることができます。これらのオプションは、セッション設定ダイアログの「CPUプロファイリング」ステップでいつでも変更できます。




Kotlinコルーチンは、CPU記録がアクティブな間に起動された場合にのみトラッキングできることを理解することが重要です。後でCPU記録を開始した場合、Kotlinコルーチンからの非同期実行はインライン化できません。JProfilerは、プロファイルされていないクラスでの呼び出し元サイトの検出と同様に通知します。アプリケーションの開始時に開始される長寿命のコルーチンをプロファイルする必要がある場合、アタッチモードを使用することはできません。その場合、-agentpath VMパラメータ [p. 12]でJVMを起動し、起動時にCPU記録を開始します。

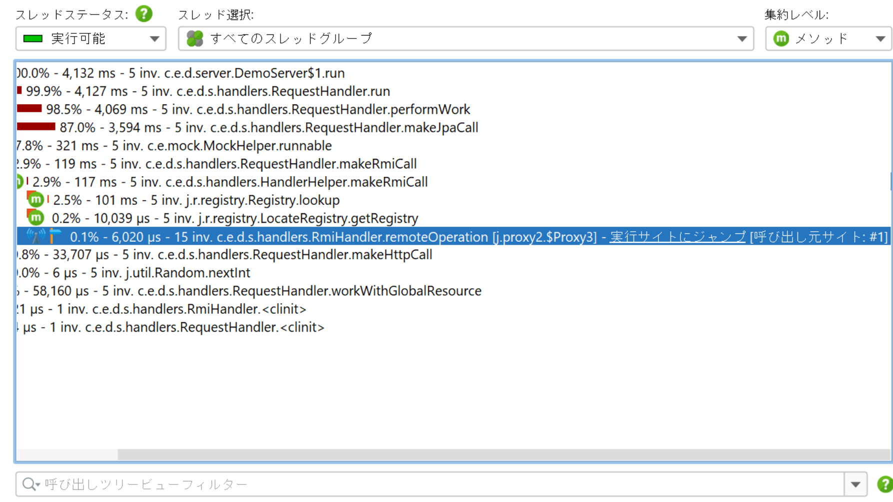
### リモートリクエストトラッキング

選択された通信プロトコルに対して、JProfilerはメタデータを挿入し、JVMの境界を越えてリクエストをトラッキングすることができます。サポートされている技術は次のとおりです：

- HTTP: クライアント側ではHttpURLConnection、java.net.http.HttpClient、Apache Http Client 4.x、Apache Async Http Client 4.x、OkHttp 3.9+、サーバー側では任意のServlet-API実装またはServletなしのJetty
- Jersey Async Client 2.x、RestEasy Async Client 3.x、Cxf Async Client 3.1.1+の非同期JAX-RS呼び出しの追加サポート
- Webサービス: JAX-WS-RI、Apache Axis2、Apache CXF
- RMI
- gRPC
- リモートEJB呼び出し: JBoss 7.1+およびWeblogic 11+

JProfilerでリクエストを追跡するためには、両方のVMをプロファイルし、別々のJProfilerトップレベルウィンドウで同時に開く必要があります。これは、ライブセッションでもスナップショットでも機能します。ターゲットJVMが現在開かれていない場合、またはリモート呼び出しの時点でCPU記録がアクティブでなかった場合、呼び出し元サイトのハイパーリンクをクリックするとエラーメッセージが表示されます。

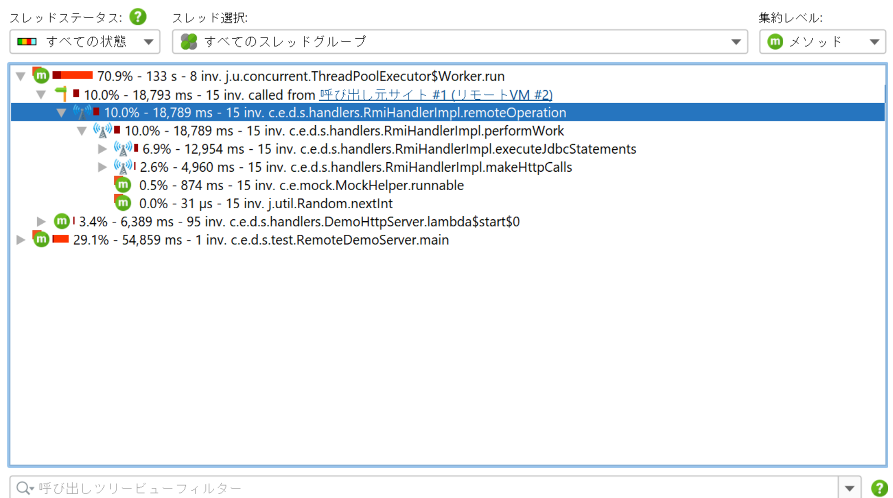
リモートリクエストをトラッキングする際、JProfilerは関与するJVMの呼び出しツリーで呼び出し元サイトと実行サイトを明示的に示します。JProfilerの呼び出し元サイトは、記録されたリモートリクエストが実行される前の最後のプロファイルされたメソッド呼び出しです。それは、異なるVMに位置する実行サイトでタスクを開始します。JProfilerでは、呼び出しツリービューに表示されるハイパーリンクを使用して、呼び出し元サイトと実行サイトの間をジャンプすることができます。



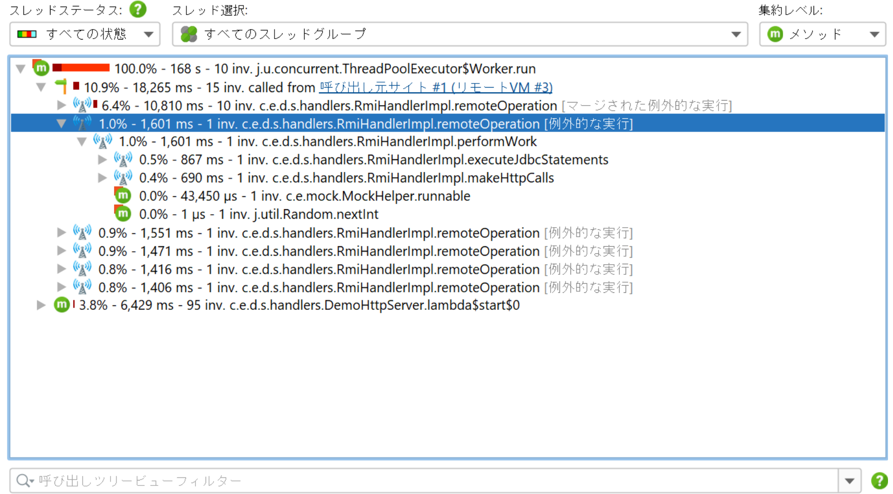

呼び出し元サイトは、すべてのスレッドに対してリモートリクエストトラッキングに関して同じアイデンティティを持っています。これは、呼び出し元サイトから実行サイトへ、またはその逆にジャンプする際に、スレッド解決がなく、ジャンプは常に「すべてのスレッドグループ」および「すべてのスレッド状態」スレッドステータス選択をアクティブにすることを意味し、ターゲットが表示されるツリーの一部であることが保証されます。

呼び出し元サイトと実行サイトは1:nの関係にあります。呼び出し元サイトは、特に異なるリモートVMにある場合、複数の実行サイトでリモートタスクを開始することができます。同じVM内では、単一の呼び出し元サイトに対して複数の実行サイトがあることはあまり一般的ではありません。なぜなら、それらは異なる呼び出しスタックで発生する必要があるからです。呼び出し元サイトが複数の実行サイトを呼び出す場合、ダイアログでそれらの1つを選択できます。

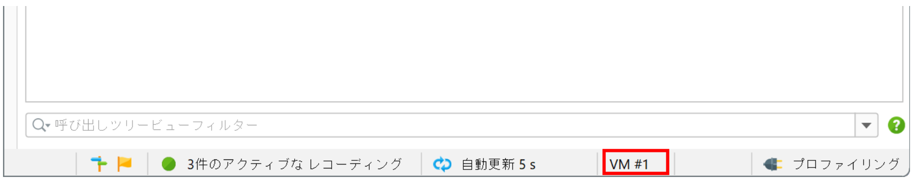
実行サイトは、特定の呼び出し元サイトによって開始されたすべての実行を含む呼び出しツリー内の合成ノードです。実行サイトノードのハイパーリンクは、その呼び出し元サイトに戻ります。




同じ呼び出し元サイトが同じ実行サイトを繰り返し呼び出す場合、実行サイトはすべての呼び出しのマージされた呼び出しツリーを表示します。それが望ましくない場合は、例外的メソッド [p.196] 機能を使用して、以下のスクリーンショットに示すように呼び出しツリーをさらに分割できます。



実行サイトは単一の呼び出し元サイトからのみ参照されるのに対し、呼び出し元サイト自体は複数の実行サイトにリンクすることができます。呼び出し元サイトの数値IDを使用して、異なる実行サイトから参照されている場合でも同じ呼び出し元サイトを認識できます。さらに、呼び出し元サイトはリモートVMのIDを表示します。プロファイルされたVMのIDはステータスバーに表示されます。それはJProfilerが内部で管理するユニークIDではなく、JProfilerで開かれた新しいプロファイルされたVMごとに1から始まりインクリメントされる表示IDです。



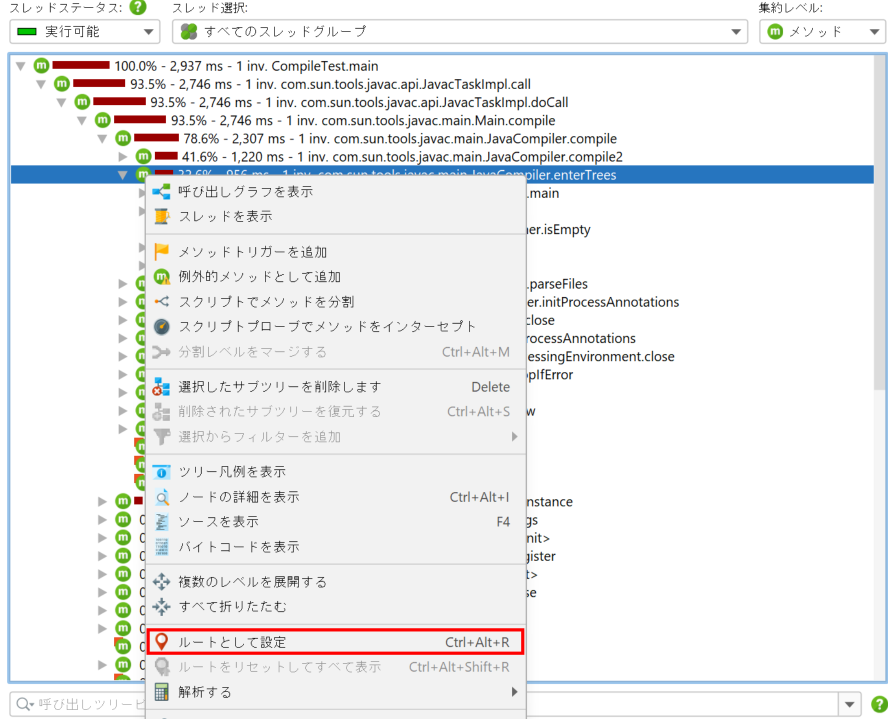
## B.3 呼び出しツリーの部分を表示する

呼び出しツリーにはしばしば情報が多すぎます。表示される詳細を減らしたい場合、いくつかの方法があります。特定のサブツリーに表示データを制限する、不要なデータをすべて削除する、またはメソッドコールを表示するためのより粗いフィルターを使用することができます。これらの戦略はすべてJProfilerによってサポートされています。

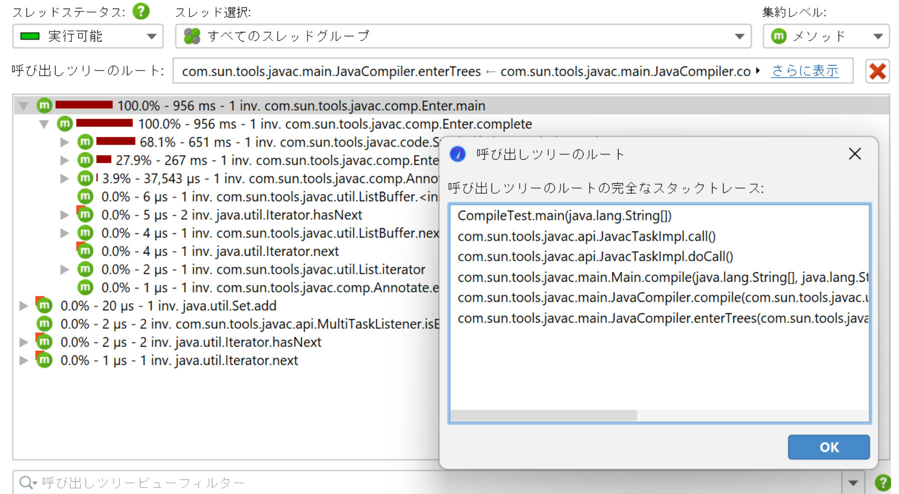
### 呼び出しツリーのルートを設定する

複数のタスクが順次実行されるユースケースをプロファイルする場合、各サブツリーを個別に分析できます。そのようなサブタスクのエントリーポイントを見つけたら、周囲の呼び出しツリーは単なる気を散らすものであり、サブツリー内のタイミングの割合が全体の呼び出しツリーのルートを参照するのは不便です。

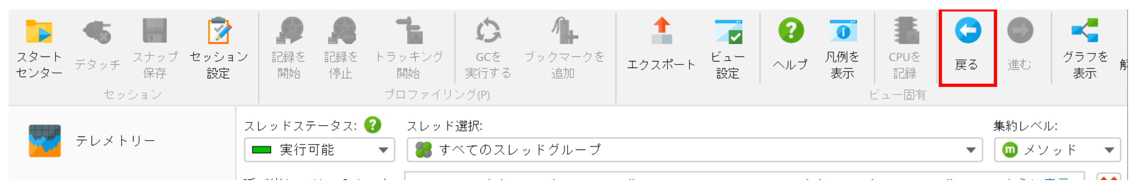
特定のサブツリーに焦点を当てるために、JProfilerは呼び出しツリーと割り当て呼び出しツリービューでルートとして設定コンテキストアクションを提供します。



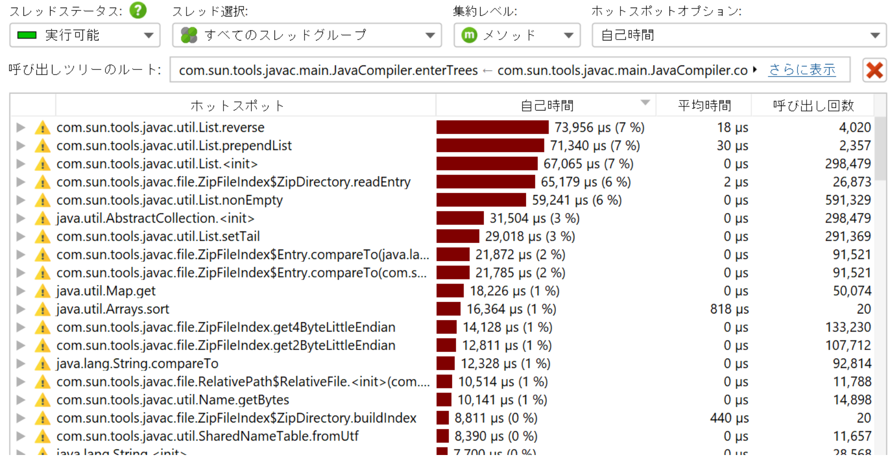

呼び出しツリーのルートを設定した後、選択したルートに関する情報がビューの上部に表示されます。スクロール可能なラベルにはルートに至るまでの最後のスタック要素が表示され、詳細を表示ボタンをクリックすると、呼び出しツールートの全スタックを含む詳細ダイアログが表示されます。




ルート設定アクションを再帰的に使用すると、呼び出しスタックのプレフィックスが単純に連結されます。前の呼び出しツリーに戻るには、呼び出しツリー履歴の戻るボタンを使用して1回のルート変更を元に戻すか、コンテキストメニューのルートをリセットしてすべてを表示アクションを使用して、元のツリーに一度に戻ることができます。




呼び出しツールートを変更する際に最も重要なのは、ホットスポットビューが選択されたルートのみに対して計算されたデータを表示し、ツリー全体に対してではないことです。ホットスポットビューの上部には、呼び出しツリービューと同様に現在の呼び出しツールートが表示され、表示されているデータのコンテキストを思い出させます。

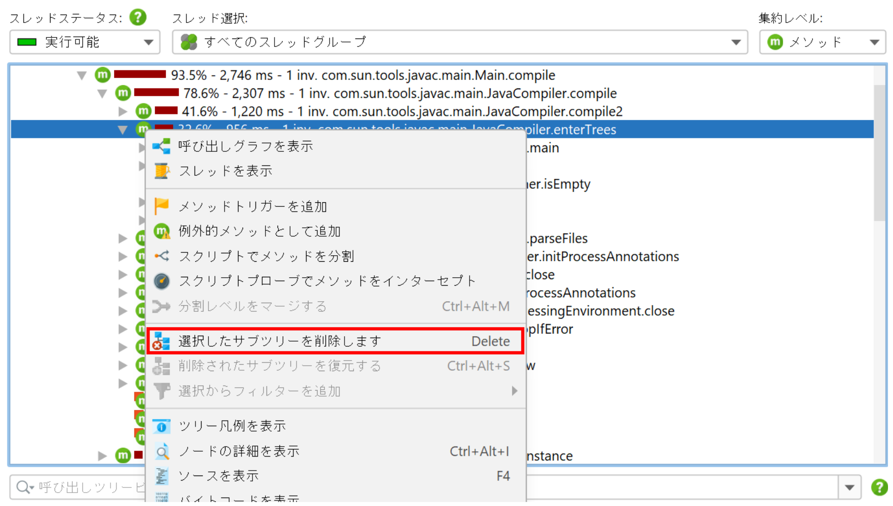



### 呼び出しツリーの部分を削除する

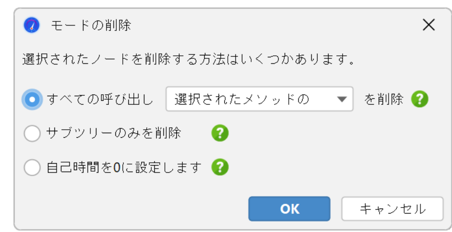
特定のメソッドが存在しない場合に呼び出しツリーがどのように見えるかを確認することが役立つ場合があります。たとえば、開発環境のように迅速に反復できない本番システムのスナップショットを使用しているため、一度に複数のパフォーマンス問題を修正する必要がある場合です。主要な

パフォーマンス問題を解決した後、次の問題を分析したいのですが、それは最初の問題がツリーから除外されている場合にのみ明確に見えます。

呼び出しツリーのノードは、選択して`Delete`キーを押すか、コンテキストメニューから選択されたサブツリーを削除を選択することで、そのサブツリーと共に削除できます。祖先ノードの時間は、隠されたノードが存在しなかったかのように適切に修正されます。

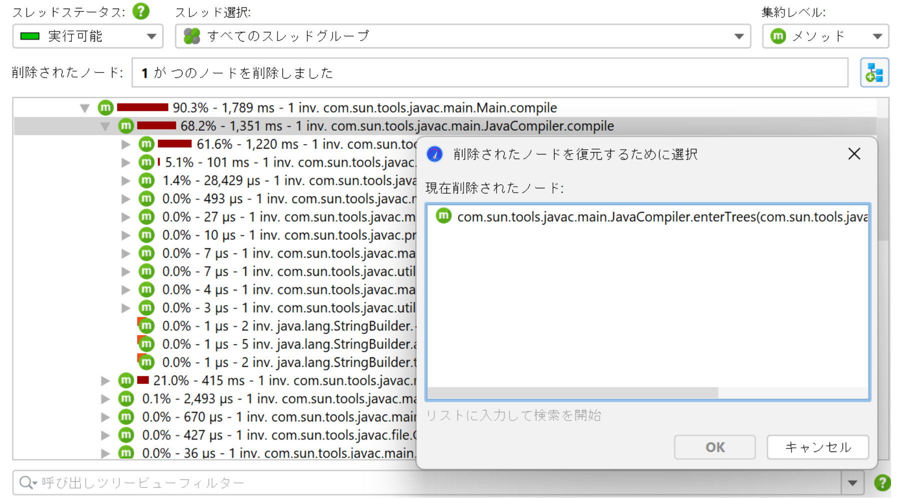



削除モードは3つあります。すべての呼び出しを削除モードでは、JProfilerは選択されたメソッドのすべての呼び出しを呼び出しツリー全体で検索し、それらをサブツリーと共に削除します。サブツリーのみを削除オプションは選択されたサブツリーのみを削除します。最後に、自己時間をゼロに設定は、選択されたノードを呼び出しツリーに残しますが、その自己時間をゼロに設定します。これは、`Thread.run`のような多くの時間をプロファイルされていないクラスから含む可能性のあるコンテナノードに役立ちます。



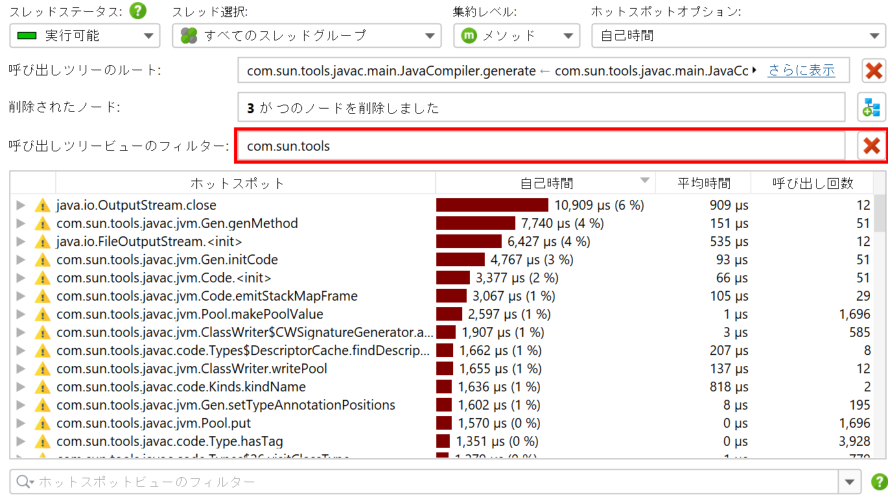

ルートとして設定アクションと同様に、削除されたノードはホットスポットビューに影響を与えます。この方法で、これらのメソッドが重要な貢献をしないように最適化された場合、ホットスポットがどのように見えるかを確認できます。

ノードを削除すると、呼び出しツリーとホットスポットビューの両方のヘッダーエリアに、削除されたノードの数と削除されたサブツリーを復元ボタンを含む行が表示されます。そのボタンをクリックすると、再表示するべき削除された要素を選択できるダイアログが表示されます。



## 呼び出しツリービューフィルター

ホットスポットビューに表示されるデータに影響を与える呼び出しの3番目の機能は、ビューフィルターです。呼び出しツリーフィルターを変更すると、計算されたホットスポット[p.54]に大きな影響を与えます。この呼び出しツリービューとの相互依存性を強調するために、ホットスポットビューはビューの上部に呼び出しツリービューフィルターを表示し、追加のフィルターを削除するボタンを表示します。

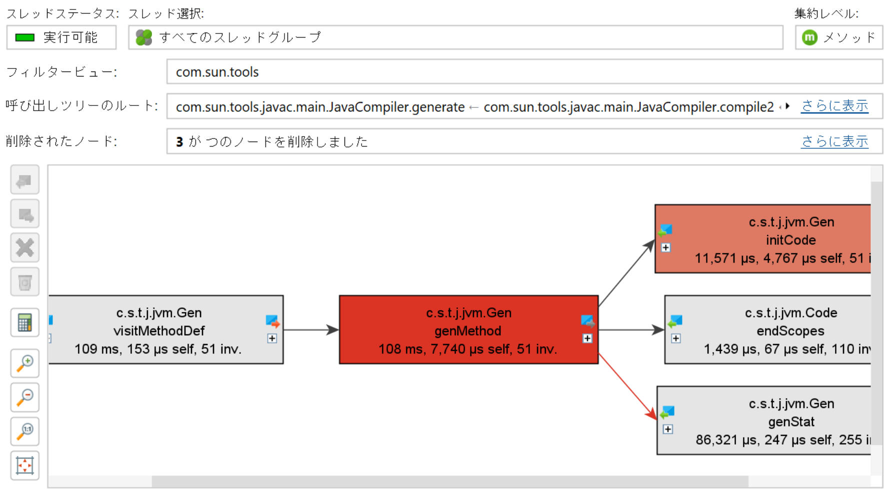


呼び出しツールートの設定、呼び出しツリーの部分の削除、およびビューフィルターは一緒に使用できますが、ビューフィルターは最後に設定する必要があります。呼び出しツリーでビューフィルターが設定されると、ルートとして設定および選択されたサブツリーを削除アクションはもう機能しません。

## 呼び出しグラフとの相互作用

呼び出しツリーまたはホットスポットビューでグラフを表示アクションを呼び出すと、同じ呼び出しツールートに限定され、削除されたメソッドを含まず、設定された呼び出しツリービューフィルターを使用するグラフが表示されます。グラフの上部には、これらの変更に関する情報が呼び出しツリーと同様の形式で表示されます。

グラフビュー自体で新しいグラフを作成する際に、ウィザードのチェックボックスを使用して、呼び出しツリー調整機能のどれを呼び出しグラフの計算に考慮するかを選択できます。各チェックボックスは、対応する機能が現在呼び出しツリービューで使用されている場合にのみ表示されます。



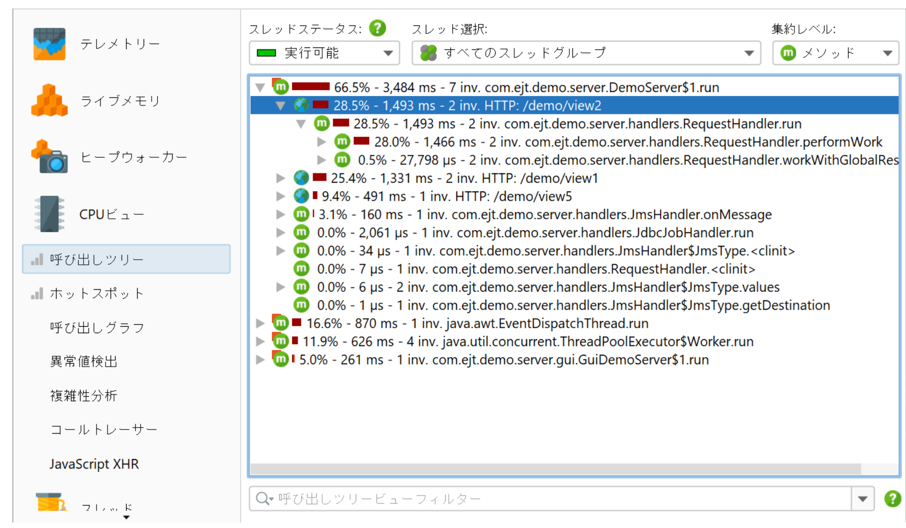
## B.4 呼び出しツリーの分割

呼び出しツリーは、同じ呼び出しスタックの繰り返し呼び出しに対して累積されます。これは、メモリのオーバーヘッドとデータを理解しやすくするために統合する必要があるためです。しかし、時には選択したポイントで累積を分割し、呼び出しツリーの一部を個別に表示したいことがあります。

JProfilerには、呼び出しスタックに特別なノードを挿入し、挿入されたノードの上にあるメソッド呼び出しから抽出されたセマンティック情報を表示することで、呼び出しツリーを分割するという概念があります。これらの分割ノードにより、呼び出しツリー内で追加のペイロード情報を直接確認し、それらのサブツリーを個別に分析することができます。各分割タイプは、コンテキストメニューのアクションで動的にマージおよびアンマージでき、メモリのオーバーヘッドが制限されるように分割ノードの総数に上限があります。

### 呼び出しツリーの分割とプローブ

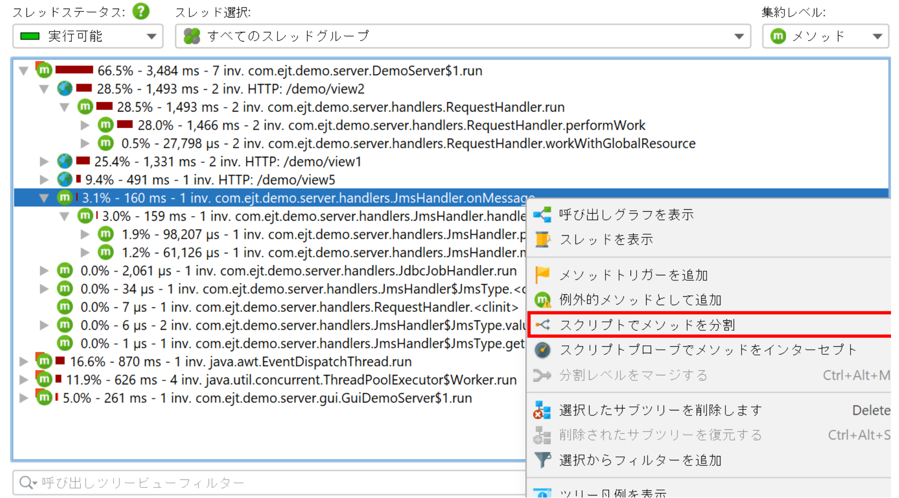
プローブ[p. 105]は、興味のある選択されたメソッドで収集した情報に基づいて呼び出しツリーを分割できます。例えば、「HTTPサーバー」プローブは、各異なるURLに対して呼び出しツリーを分割します。この場合の分割は非常に設定可能で、URLの必要な部分のみを含めたり、サーブレットコンテキストからの他の情報を含めたり、複数の分割レベルを生成したりすることができます。



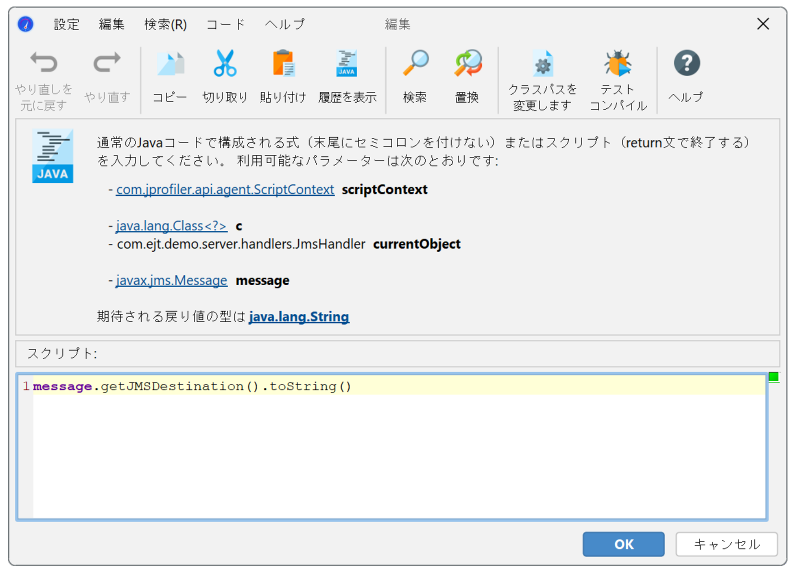

独自のプローブを作成する場合、埋め込み[p. 168]および注入[p. 163]カスタムプローブシステムの両方で、同じ方法で呼び出しツリーを分割できます。

### スクリプトによるメソッドの分割

プローブで利用可能な同じ分割機能を、スクリプトでメソッドを分割アクションを使用して、直接呼び出しツリーで使用できます。以下のスクリーンショットでは、JMSメッセージハンドラの呼び出しツリーを分割して、異なるタイプのメッセージの処理を個別に確認したいと考えています。




プローブを書く代わりに、文字列を返すスクリプトを入力します。この文字列は、選択したメソッドで呼び出しツリーをグループ化するために使用され、分割ノードに表示されます。`null`を返すと、現在のメソッド呼び出しは分割されず、通常通り呼び出しツリーに追加されます。

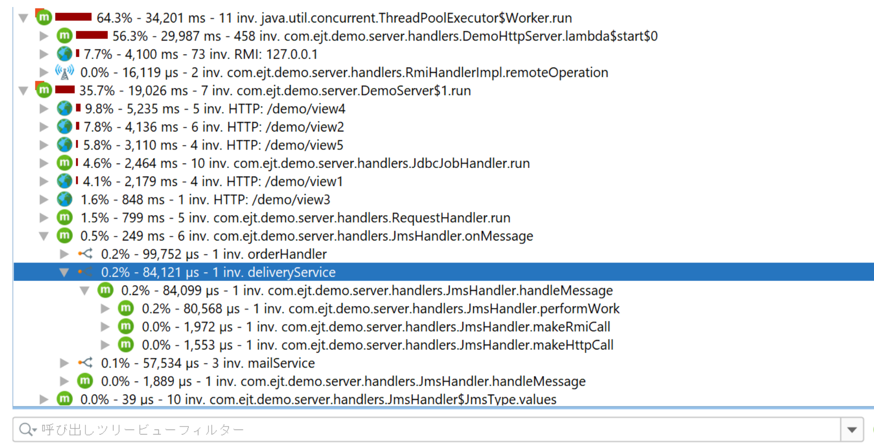



スクリプトは多くのパラメータにアクセスできます。選択したメソッドのクラス、非静的メソッドの場合はインスタンス、およびすべてのメソッドパラメータが渡されます。さらに、データを保存するために使用できる`ScriptContext`オブジェクトを取得します。同じスクリプトの以前の呼び出しからいくつかの値を思い出す必要がある場合、コンテキストで`getObject`/`putObject`および`getLong`/`putLong`メソッドを呼び出すことができます。例えば、特定のメソッドパラメータの値が初めて見られたときにのみ分割したい場合があります。その場合、次のように使用できます。

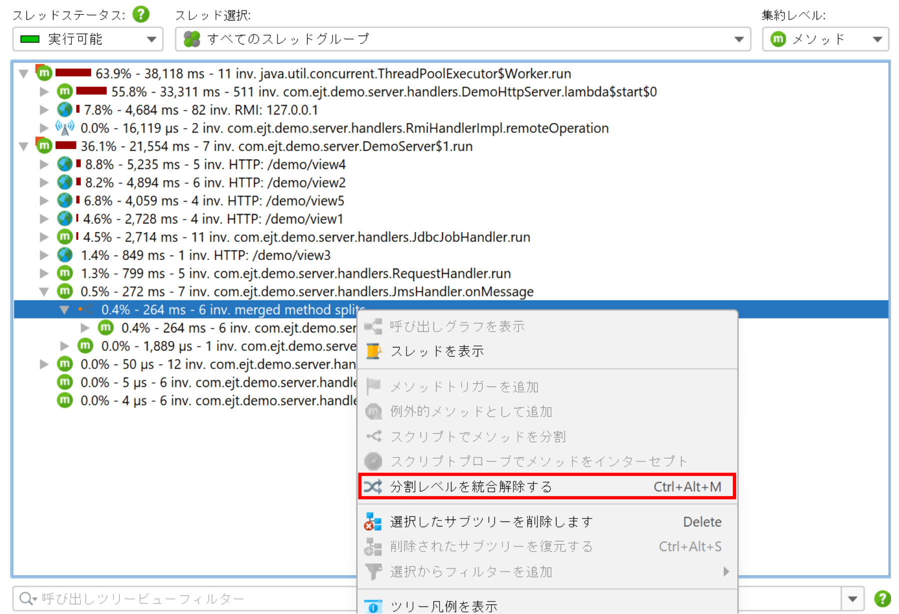
```
if (scriptContext.getObject(text) != null) {
    scriptContext.putObject(text);
    return text;
} else {
    return null;
}
```

これは分割スクリプトの一部です。

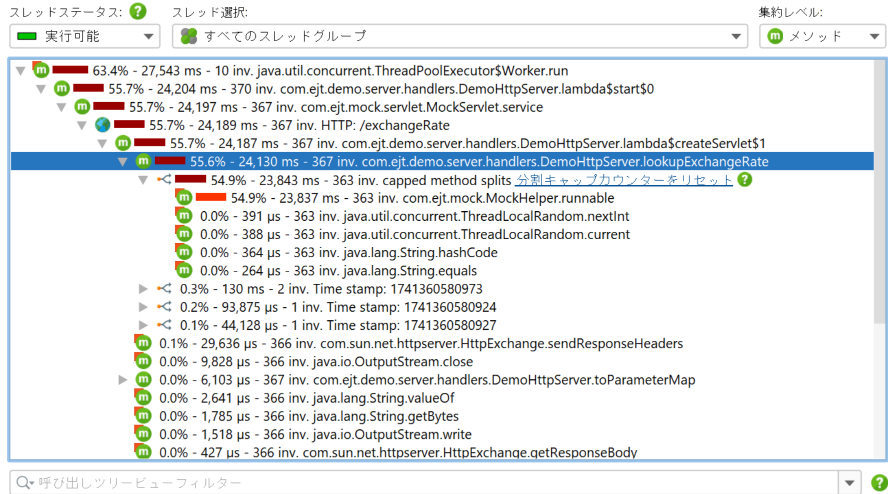
分割ノードは選択したメソッドの下に挿入されます。上記のスクリーンショットの例では、各JMSメッセージの宛先に対する処理コードを個別に確認できます。



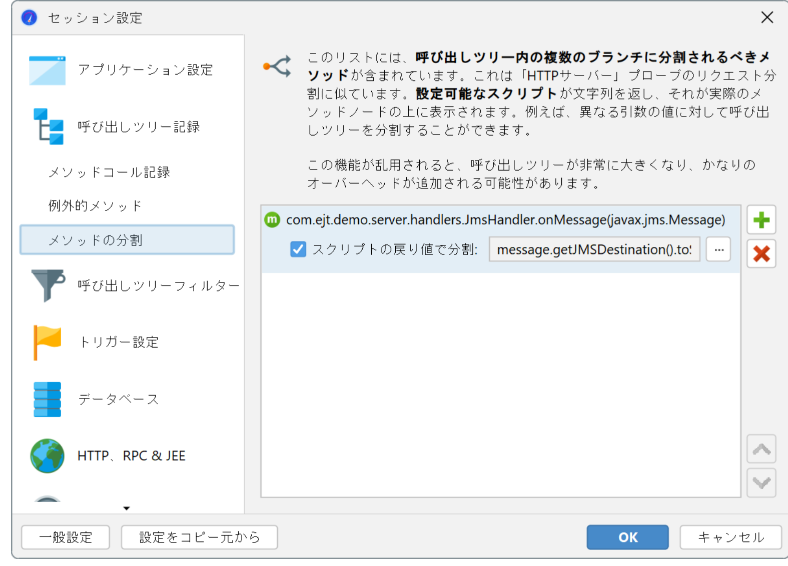

分割の場所は選択した呼び出しスタックではなく、メソッドにバインドされています。同じメソッドが呼び出しツリーの他の場所に存在する場合、それも分割されます。分割レベルをマージアクションを使用すると、すべての分割が単一のノードにマージされます。そのノードは、再度分割を解除する機会を提供します。




あまりにも多くの分割を生成すると、制限されたメソッド分割とラベル付けされたノードに、すべてのさらなる分割呼び出しが単一のツリーに累積されます。ノード内のハイパーリンクを使用して、キャップカウンターをリセットし、さらに分割ノードを記録できます。分割の最大数を恒久的に増やすには、プロファイリング設定でキャップを増やすことができます。



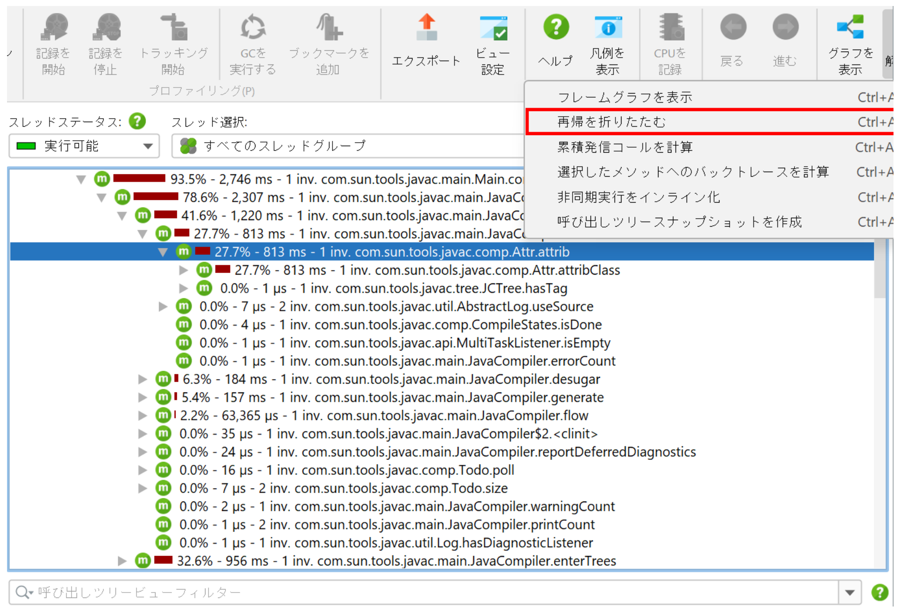

作成後に分割メソッドを編集するには、セッション設定ダイアログに移動します。特定の分割メソッドがもう必要ないが、将来の使用のために保持したい場合、スクリプト設定の前のチェックボックスを使用して無効にできます。これは、呼び出しツリーで単にマージするよりも優れています。なぜなら、記録のオーバーヘッドが重要である可能性があるからです。



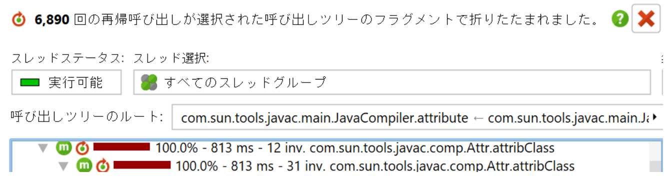
## B.5 呼び出しツリー分析

呼び出しツリー[p.54]は、JProfilerが記録した実際の呼び出しスタックを表示します。呼び出しツリーを分析する際、解釈を容易にするために呼び出しツリーに適用できるいくつかの変換があります。これらの変換は時間がかかり、呼び出しツリービューの機能と互換性のない形式に出力を変更するため、分析結果の新しいビューが作成されます。

このような分析を行うには、呼び出しツリービューでノードを選択し、ツールバーまたはコンテキストメニューから 呼び出しツリー分析アクションのいずれかを選択します。




呼び出しツリービューの下にネストされたビューが作成されます。同じ分析アクションを再度実行すると、分析が置き換えられます。同時に複数の分析結果を保持するには、結果ビューをピン留めすることができます。その場合、同じタイプの次の分析は新しいビューを作成します。ピン留めされたビューでは、ビューセレクタの左側に表示される名前を変更するためのリネームボタンがビューの上部に表示されます。



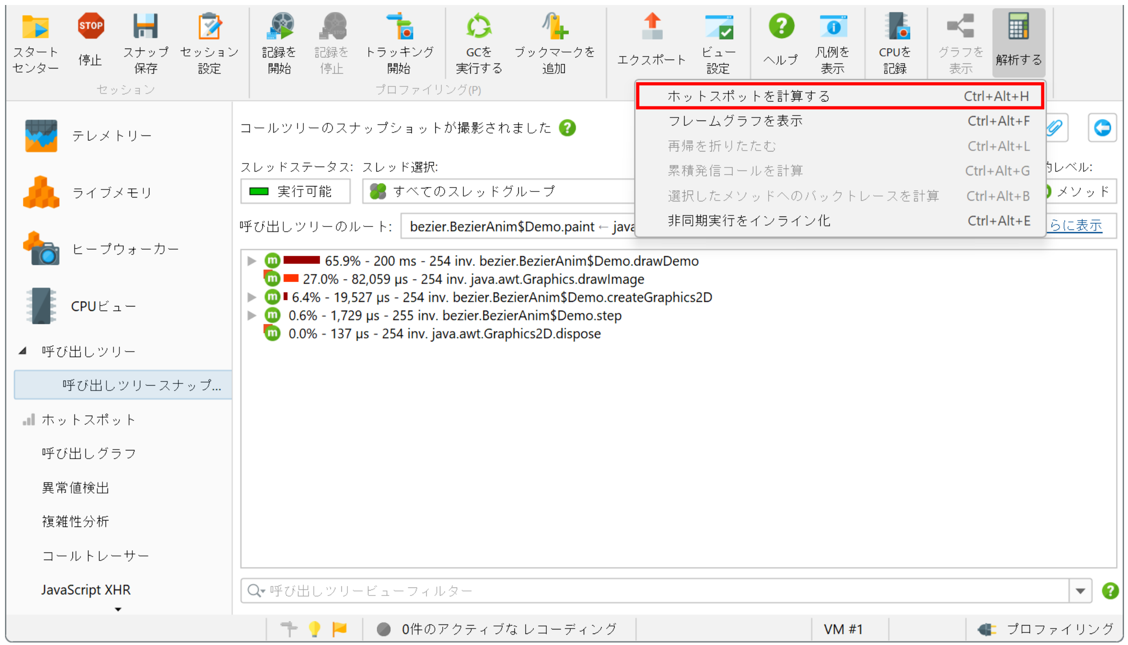

ライブセッションでは、結果ビューは呼び出しツリーと共に更新されず、分析が行われた時点のデータを表示します。現在のデータに対して分析を再計算するには、リロードアクションを使用します。割り当てツリーで自動更新が無効になっている場合のように、呼び出しツリー自体を再計算する必要がある場合も、リロードアクションが それを処理します。

### 呼び出しツリースナップショット

「呼び出しツリースナップショットの作成」分析は、現在の呼び出しツリーの静的コピーを作成するだけです。これは、JProfilerスナップショットを保存して開くことなく、異なるユースケースを比較するのに便利です。　また、記録中の呼び出しツリーの凍結コピーを操作する方法を提供します。

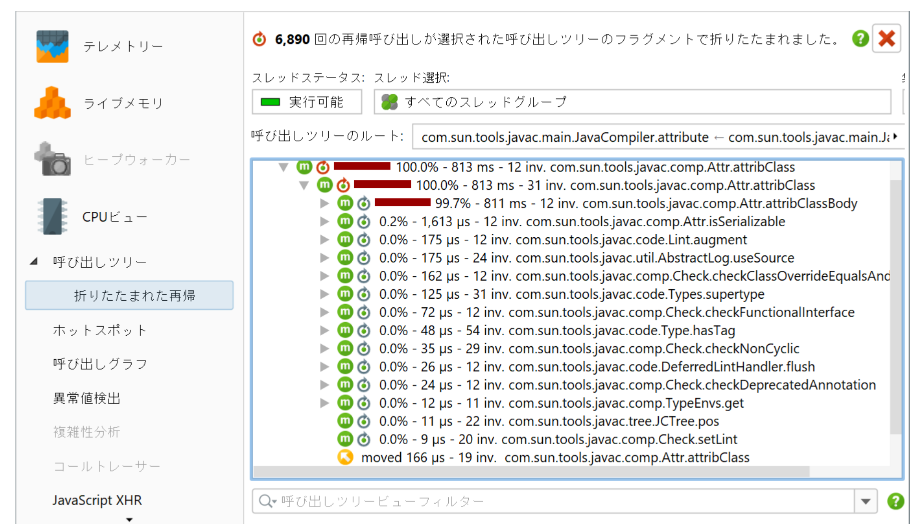
「呼び出しツリースナップショットの作成」分析は、「CPUビュー」セクションの「呼び出しツリー」ビューでのみ利用可能です。呼び出しツリースナップショットビューをピン留めすると、同時に複数の呼び出しツリースナップショットを持つことができます。他の分析とは異なり、呼び出しツリースナップショットは独立したデータセットを構成するため、JProfilerスナップショットに保存されます。

呼び出しツリービューで利用可能な呼び出しツリー分析に加えて、呼び出しツリースナップショットには　親ビューのホットスポットを計算する「ホットスポットの計算」アクションもあります。「CPUビュー」セクションの「ホットスポット」ビューと同様です。呼び出しツリースナップショットビューの下にネストされたビューからアクセス可能なすべての分析は、トップレベルの呼び出しツリービューのデータではなく、親の呼び出しツリースナップショットのデータを使用します。

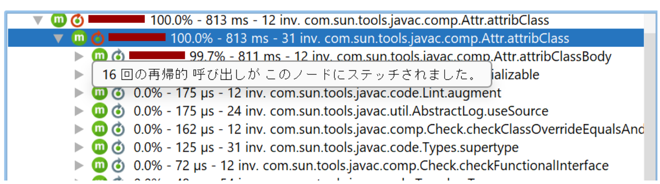



## 再帰の折りたたみ

再帰を利用するプログラミングスタイルは、分析が難しい呼び出しツリーをもたらします。「再帰の折りたたみ」呼び出しツリー分析は、すべての再帰が折りたたまれた呼び出しツリーを計算します。 呼び出しツリーで現在選択されているノードの親ノードが、分析のための 呼び出しツリーのルート [p. 181]として機能します。 呼び出しツリー全体を分析するには、トップレベルのノードのいずれかを選択します。




再帰は、同じメソッドが呼び出しスタックの上位で既に呼び出された場合に検出されます。その場合、サブツリーは呼び出しツリーから削除され、そのメソッドの最初の呼び出しに戻されます。そのノードは、ツールチップに再帰の数を示すアイコンでプレフィックスされます。そのノードの下では、異なる深さからのスタックがマージされます。マージされたスタックの数もツールチップに表示されます。折りたたまれた再帰の総数は、元の呼び出しツリーに設定された呼び出しツリーパラメータに関する情報の上にあるヘッダーに表示されます。

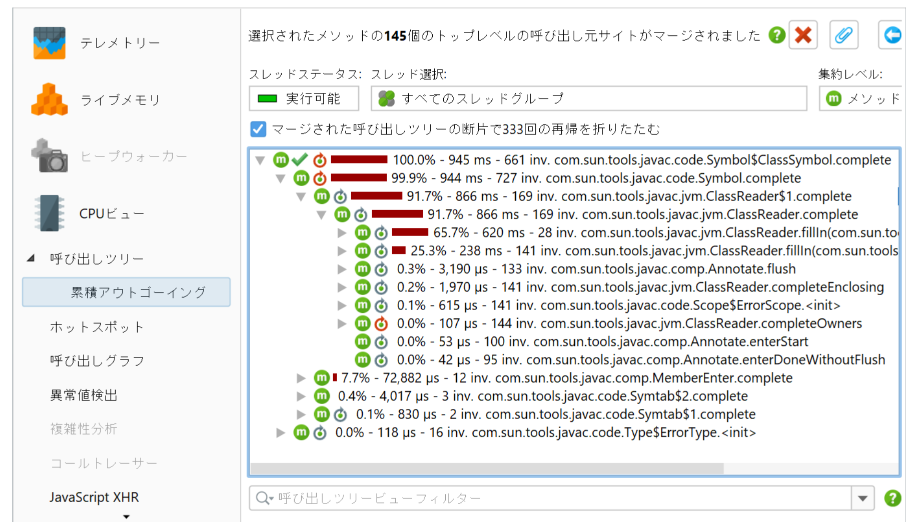



単純な再帰の場合、マージされたスタックの数は再帰の数プラス1です。したがって、 再帰ツールチップに「1再帰」と表示されるノードは、その再帰ツールチップに「2マージスタック」と表示されるノードを含むツリーを持ちます。より複雑なケースでは、再帰がネストされ、重複するマージされた呼び出しツリーを生成するため、スタックの深さごとにマージされたスタックの数が異なります。

サブツリーが呼び出しツリーから削除されて上位でマージされるポイントで、特別な 「移動されたノード」 プレースホルダーが挿入されます。

### 累積されたアウトゴーイングコールの分析

呼び出しツリーでは、選択されたメソッドのアウトゴーイングコールを見ることができますが、そのメソッドが呼び出された特定の呼び出しスタックに対してのみです。同じ関心のあるメソッドが異なる呼び出しスタックで呼び出された可能性があり、より良い統計を得るためにそれらの呼び出しのすべての累積された呼び出しツリーを分析することがしばしば有用です。「累積されたアウトゴーイングコールの計算」分析は、選択されたメソッドのすべてのアウトゴーイングコールを合計した呼び出しツリーを表示します。




選択されたメソッドについて、JProfilerは再帰呼び出しを考慮せずにそのトップレベルの呼び出しをすべて収集し、結果ツリーに累積します。ヘッダーには、そのプロセスで合計されたトップレベルの呼び出し元サイトの数が 表示されます。

ビューの上部には、結果ツリーで再帰を折りたたむことができるチェックボックスがあります。これは「再帰の折りたたみ」分析と似ています。再帰が折りたたまれると、トップレベルノードとアウトゴーイングコールの最初のレベルは、メソッドコールグラフと同じ数値を示します。

### バックトレースの計算

「バックトレースの計算」分析は、「累積されたアウトゴーイングコールの計算」分析を補完します。後者と同様に、再帰呼び出しを考慮せずに選択されたメソッドのすべてのトップレベル呼び出しを合計します。ただし、アウトゴーイングコールを表示する代わりに、選択されたメソッドの呼び出しに寄与するバックトレースを表示します。呼び出しは最も深いノードから始まり、選択されたメソッドに向かって進行します。

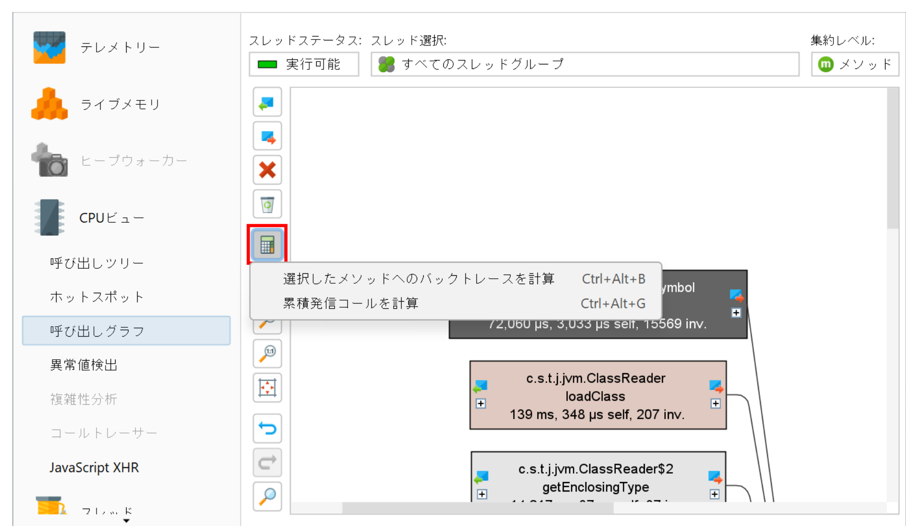
この分析はホットスポットビューに似ていますが、デフォルトでは選択されたメソッドの自己時間ではなく合計時間を合計し、ホットスポットビューは合計時間の重要な部分を占めるメソッドのみを表示します。　ビューの上部には集計モードとラベル付けされたラジオボタングループがあり、自己時間に設定できます。その選択で、選択されたメソッドの合計値はホットスポットビューのデフォルトモードと一致します。

バックトレースでは、バックトレースノードの呼び出し回数と時間は選択されたメソッドにのみ関連しています。それらは、その特定の呼び出しスタックに沿った呼び出しが選択されたメソッドの値にどれだけ寄与したかを示します。　「累積されたアウトゴーイングコールの計算」分析と同様に、再帰を折りたたむことができ、バックトレースの最初のレベルはメソッドコールグラフのインカミングコールと同等です。

#### 呼び出しグラフでの呼び出しツリー分析

呼び出しグラフでは、各メソッドは一意ですが、呼び出しツリーではメソッドが複数の呼び出しスタックで発生することがあります。選択されたメソッドに対して、「累積されたアウトゴーイングコールの計算」と「バックトレースの計算」分析は、呼び出しツリーと呼び出しグラフの視点の橋渡しをします。それらは選択されたメソッドを中心に置き、アウトゴーイングコールとインカミングコールをツリーとして表示します。コールグラフを表示アクションを使用すると、いつでも完全なグラフに切り替えることができます。

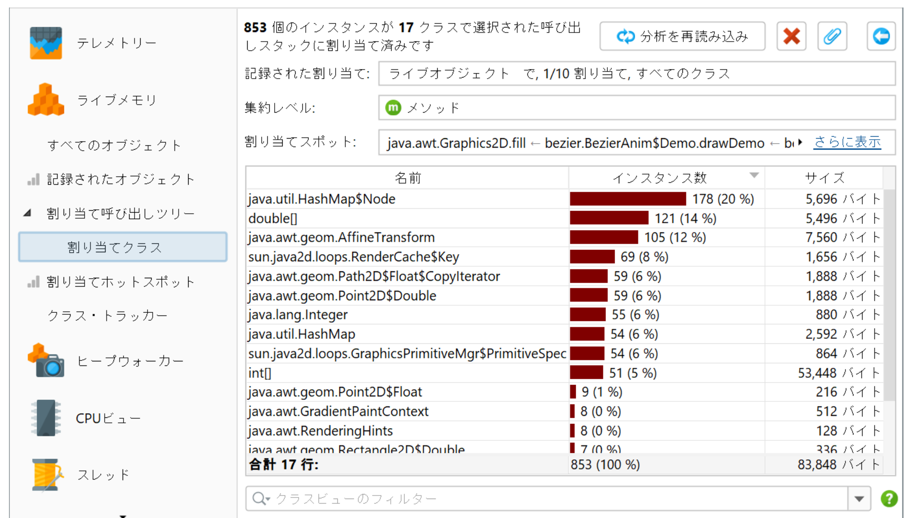
時には、逆の方向に視点を切り替え、グラフからツリービューに変更したいことがあります。呼び出しグラフで作業しているとき、グラフ内の任意の選択されたノードに対して、呼び出しツリー分析と同じ方法で累積されたアウトゴーイングコールとバックトレースをツリーとして表示できます。




IntelliJ IDEA統合 [p. 141]では、エディタのガターに表示される呼び出しグラフには、 これらのツリーを直接表示するアクションが含まれています。

#### 割り当てのためのクラスの表示

以前の呼び出しツリー分析とは少し異なるのは、割り当て呼び出しツリーと割り当てホットスポットビューでの「クラスの表示」分析です。これは呼び出しツリーを別のツリーに変換するのではなく、 割り当てられたすべてのクラスを含むテーブルを表示します。結果ビューは 記録されたオブジェクトビュー [p. 72]に似ていますが、 特定の割り当てスポットに制限されています。




呼び出しツリーを表示する分析結果ビューでは、「累積されたアウトゴーイングコールの計算」と「選択されたメソッドへのバックトレースの計算」分析の両方が利用可能です。それらを呼び出すと、独立したパラメータを持つ新しいトップレベル分析が作成されます。前の分析結果ビューからの呼び出しツリーの削除は、新しいトップレベル分析には反映されません。

一方、クラスの表示アクションは、呼び出しツリー分析結果ビューから使用された場合、 新しいトップレベル分析を作成しません。代わりに、元のビューの2レベル下にネストされた分析を作成します。

# C 高度なCPU分析ビュー

## C.1 異常値検出と例外的メソッド記録

ある状況では、メソッドの平均呼び出し時間が問題ではなく、時折メソッドが誤動作することが問題となることがあります。呼び出しツリーでは、すべてのメソッド呼び出しが累積されるため、頻繁に呼び出されるメソッドが1万回の呼び出しのうち1回だけ期待の100倍の時間がかかる場合、合計時間には明確な痕跡を残しません。

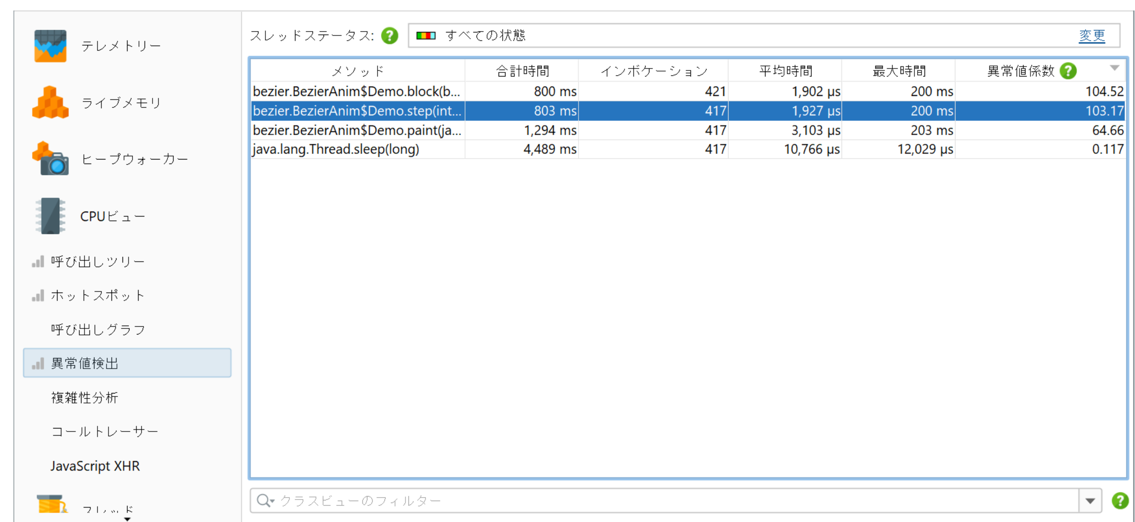
この問題に対処するために、JProfilerは呼び出しツリーにおける異常値検出ビューと例外的メソッド記録機能を提供します。

### 異常値検出ビュー

異常値検出ビューは、各メソッドの呼び出し時間と呼び出し回数に関する情報を、単一の呼び出しで測定された最大時間と共に表示します。最大呼び出し時間と平均時間の偏差は、すべての呼び出し時間が狭い範囲にあるか、または顕著な異常値があるかを示します。異常値係数は次のように計算されます。

```
（最大時間 − 平均時間） / 平均時間
```

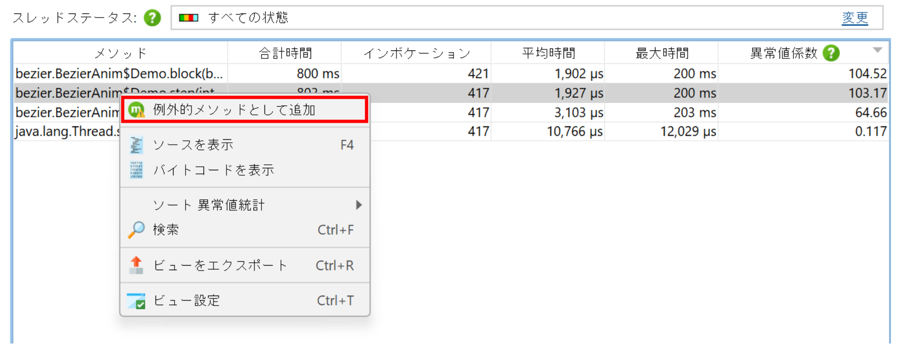
これにより、この点でメソッドを定量化するのに役立ちます。デフォルトでは、異常値係数が最も高いメソッドが上位に表示されるようにテーブルがソートされています。異常値検出ビューのデータは、CPUデータが記録されている場合に利用可能です。




数回しか呼び出されないメソッドや非常に短時間で実行されるメソッドからの過剰な混乱を避けるために、最大時間と呼び出し回数の下限をビュー設定で設定できます。デフォルトでは、最大時間が10ms以上で呼び出し回数が10を超えるメソッドのみが異常値統計に表示されます。

### 例外的メソッド記録の設定

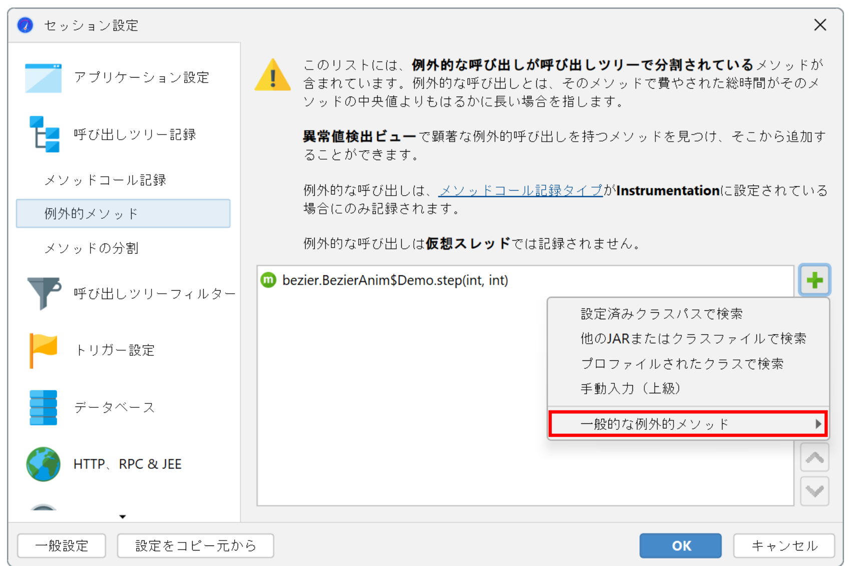
例外的な呼び出し時間を持つメソッドを特定したら、コンテキストメニューでそれを例外的メソッドとして追加できます。同じコンテキストメニューアクションは、呼び出しツリービューでも利用可能です。




例外的メソッド記録のためにメソッドを登録すると、最も遅い呼び出しのいくつかが呼び出しツリーに個別に保持されます。他の呼び出しは通常通り単一のメソッドノードにマージされます。個別に保持される呼び出しの数はプロファイリング設定で設定できます。デフォルトでは5に設定されています。

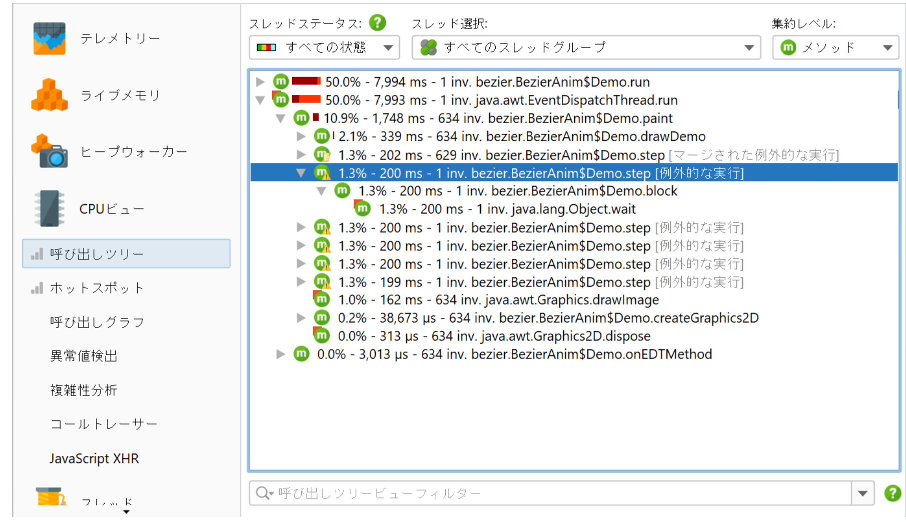
遅いメソッド呼び出しを区別する際には、時間測定のために特定のスレッド状態を使用する必要があります。これはCPUビューのスレッドステータス選択ではなく、表示オプションであり記録オプションではありません。デフォルトではウォールクロック時間が使用されますが、プロファイリング設定で異なるスレッドステータスを設定できます。同じスレッド状態が異常値検出ビューにも使用されます。

セッション設定では、呼び出しツリーや異常値検出ビューのコンテキストなしで例外的メソッドを削除したり、新しいものを追加したりできます。また、例外的メソッド設定では、AWTやJavaFXのイベントディスパッチメカニズムのような、例外的に長時間実行されるイベントが大きな問題となる既知のシステムに対して例外的メソッド定義を追加するオプションを提供します。




### 呼び出しツリー内の例外的メソッド

例外的メソッドの実行は、呼び出しツリービューで異なる表示がされます。




分割されたメソッドノードは、変更されたアイコンと追加のテキストを表示します：

- **[例外的な実行]**

  このノードには、例外的に遅いメソッド実行が含まれています。定義により、呼び出し回数は1になります。後で多くの他のメソッド実行が遅くなると、このノードは消えて、設定された最大数の個別に記録されたメソッド実行に応じて「マージされた例外的実行」ノードに追加される可能性があります。

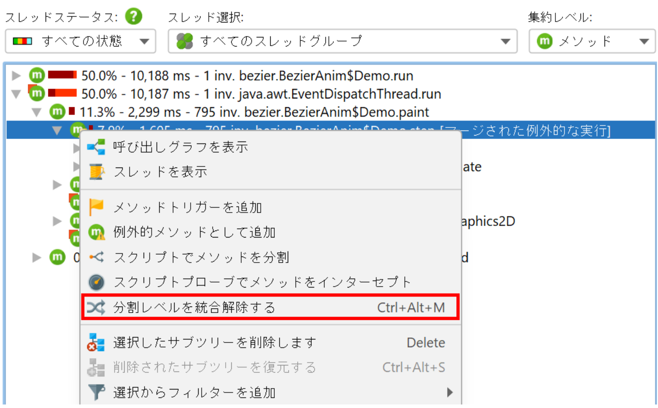
- **[マージされた例外的実行]**

  例外的に遅いと見なされないメソッド呼び出しは、このノードにマージされます。任意の呼び出しスタックに対して、例外的メソッドごとに1つのノードしか存在できません。

- **[現在の例外的実行]**

呼び出しツリービューがJProfiler GUIに送信されたときに呼び出しが進行中であった場合、その呼び出しが例外的に遅いかどうかはまだわかりませんでした。「現在の例外的実行」は、現在の呼び出しのために個別に維持されたツリーを示します。呼び出しが完了すると、それは「例外的実行」ノードとして個別に維持されるか、「マージされた例外的実行」ノードにマージされます。

プローブ[p.105]や分割メソッド[p.186]による呼び出しツリーの分割と同様に、例外的メソッドノードには、コンテキストメニューで呼び出しをその場でマージおよびアンマージするための分割レベルをマージアクションがあります。



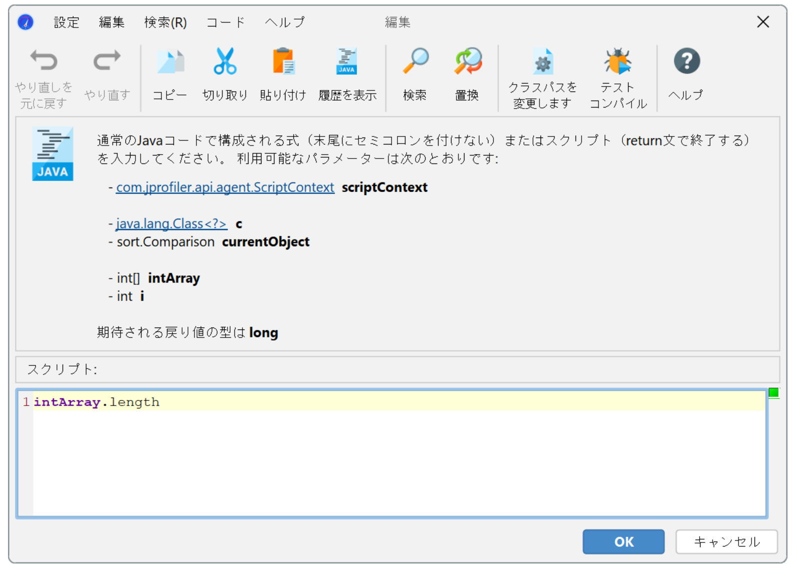
## C.2 複雑性分析

複雑性分析ビューでは、選択したメソッドのアルゴリズムの複雑性をメソッドパラメータに応じて調査することができます。

ビッグO記法の詳細を更新するには、アルゴリズムの複雑性の入門[1]と一般的なアルゴリズムの複雑性の比較ガイド[2]を読むことをお勧めします。

まず、モニターするメソッドを1つ以上選択する必要があります。

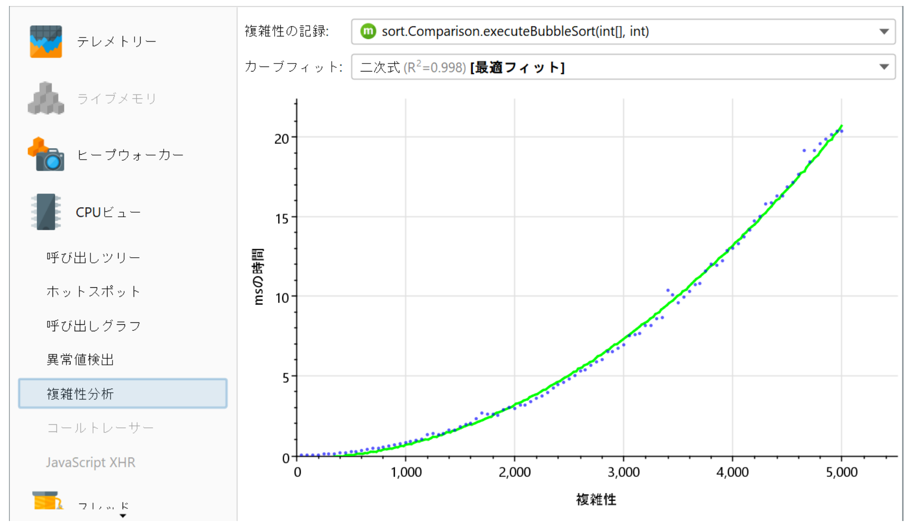
各メソッドに対して、`long`型の戻り値を持つスクリプトを入力できます。この戻り値が現在のメソッドコールの複雑性として使用されます。例えば、`java.util.Collection`型のメソッドパラメータが`inputs`という名前の場合、スクリプトは`inputs.size()`となります。



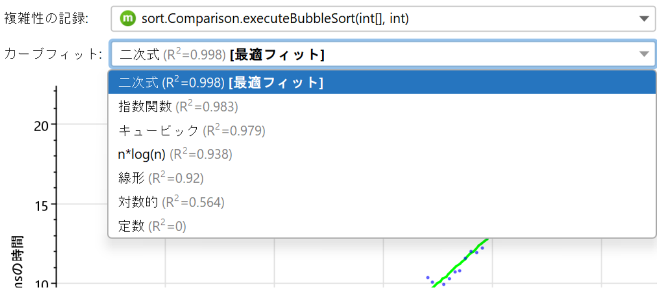

複雑性の記録はCPUの記録とは独立しています。複雑性分析ビューで直接、または記録プロファイルやトリガーアクション [p. 28]を使用して複雑性の記録を開始および停止できます。 記録が停止された後、結果を示すグラフが表示され、x軸に複雑性、y軸に実行時間がプロットされます。メモリ要件を削減するために、JProfilerは異なる複雑性と実行時間を共通のバケットにまとめることができます。上部のドロップダウンで、異なる設定済みのメソッド間を切り替えることができます。

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

グラフはバブルチャートで、各データポイントのサイズはその中の測定数に比例します。すべての測定が異なる場合、通常の散布図が表示されます。反対に、すべてのメソッド呼び出しが同じ複雑性と実行時間を持つ場合、1つの大きな円が表示されます。




少なくとも3つのデータポイントがある場合、一般的な複雑性を持つカーブフィットが表示されます。JProfilerは、いくつかの一般的な複雑性からカーブフィットを試み、最初に最適なフィットを表示します。カーブフィットのドロップダウンで他のカーブフィットモデルを表示することもできます。カーブフィットの説明に埋め込まれた$R^2$値は、フィットの良さを示します。ドロップダウン内のモデルは$R^2$に関して降順にソートされているため、最適なモデルは常に最初の項目です。




$R^2$は負の値を取ることができることに注意してください。これは単なる記法であり、実際に何かの二乗ではありません。負の値は、一定のラインフィットよりも悪いフィットを示します。一定のラインフィットは常に$R^2$値が0であり、完全なフィットは1の値を持ちます。

現在表示されているフィットのパラメータをエクスポートするには、エクスポートダイアログで「プロパティ」オプションを選択します。品質保証環境での自動分析のために、コマンドラインエクスポート [p. 256]はプロパティ形式をサポートしています。

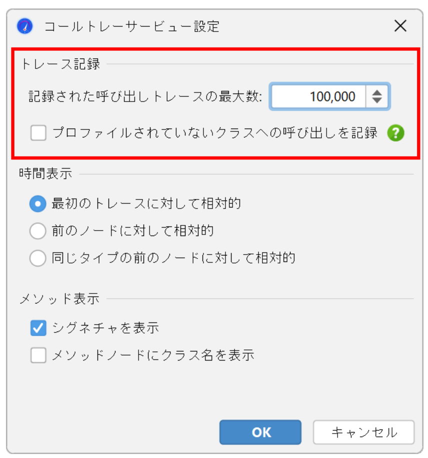
## C.3 コールトレーサー

呼び出しツリーでのメソッドコールの記録は、同じ呼び出しスタックを持つコールを累積します。正確な時系列情報を保持することは通常、メモリ要件が膨大であり、記録されたデータの量が多いため、解釈が非常に困難になるため、実現不可能です。

しかし、限られた状況では、コールをトレースし、完全な時系列を保持することが意味を持ちます。たとえば、複数の協力スレッドのメソッドコールの正確なインターレースを分析したい場合があります。デバッガーはそのようなユースケースをステップスルーすることはできません。あるいは、一連のメソッド呼び出しを分析したいが、デバッガーのように一度だけ見るのではなく、前後に移動できるようにしたい場合もあります。JProfilerはコールトレーサーでこの機能を提供します。

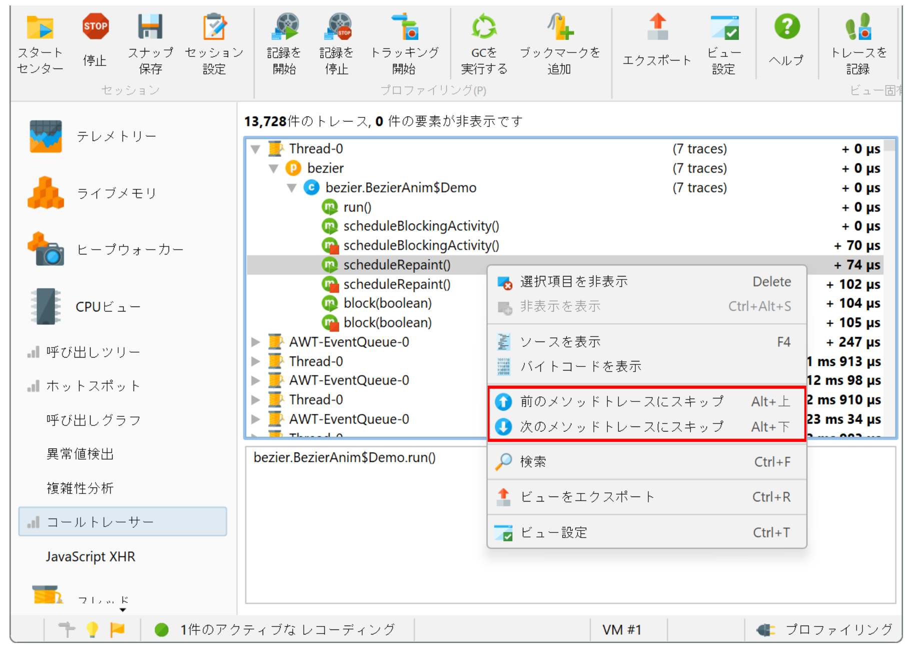
コールトレーサーには、コールトレーサービューでアクティブ化できる別の記録アクションがあり、トリガー [p. 28]またはプロファイリングAPI [p. 130]で使用できます。過剰なメモリ消費の問題を避けるために、収集されるコールトレースの最大数に上限が設定されています。その上限はビュー設定で構成可能です。収集されるトレースの割合は、フィルター設定に大きく依存します。

コールトレーシングは、メソッドコール記録タイプがインストルメンテーションに設定されている場合にのみ機能します。サンプリングは単一のメソッドコールを追跡しないため、サンプリングでコールトレースを収集することは技術的に不可能です。コンパクトフィルタリングされたクラスへのコールは、呼び出しツリーと同様にコールトレーサーで記録されます。自分のクラスにのみ焦点を当てたい場合は、ビュー設定でこれらのコールを除外できます。




トレースされたメソッドコールは、関連するコールを折りたたむことでスキップしやすくするために、3つのレベルのツリーで表示されます。3つのグループは スレッド、 パッケージ および クラスです。これらのグループのいずれかの現在の値が変更されるたびに、新しいグループ化ノードが作成されます。

最下層には メソッドエントリ および メソッド終了ノードがあります。コールトレースのテーブルの下には、現在選択されているメソッドトレースのスタックトレースが表示されます。他のメソッドへのコールトレースが現在のメソッドから記録されている場合、または別のスレッドが現在のメソッドを中断する場合、そのメソッドのエントリと終了ノードは隣接しません。前のメソッドおよび次のメソッドアクションを使用して、メソッドレベルでのみナビゲートできます。




トレースおよびすべてのグループ化ノードに表示されるタイミングは、デフォルトで最初のトレースを指しますが、前のノードからの相対時間を表示するように変更できます。前のノードが親ノードである場合、その差はゼロになります。同じタイプの前のノードに対する相対時間を表示するオプションも利用可能です。

適切なフィルターを使用しても、非常に短時間で膨大な数のトレースが収集される可能性があります。興味のないトレースを排除するために、コールトレーサーは表示されるデータを迅速にトリムすることを可能にします。たとえば、特定のスレッドが関連性がないか、特定のパッケージやクラスのトレースが興味深くないかもしれません。また、再帰的なメソッド呼び出しは多くのスペースを占有する可能性があり、それらの単一のメソッドのみを排除したい場合があります。

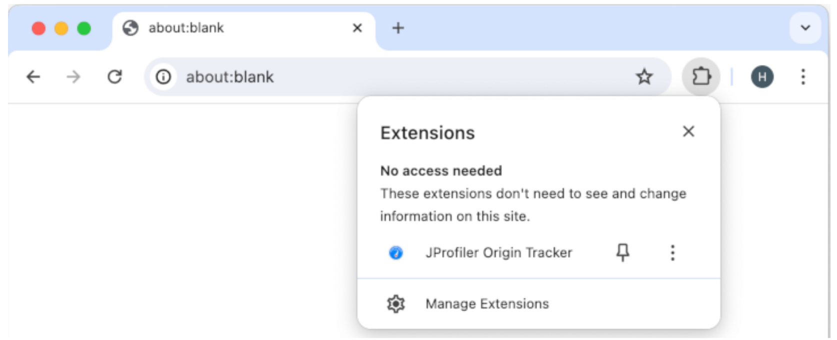
ノードを選択して`delete`キーを押すことでノードを非表示にできます。選択したノードの他のすべてのインスタンスと関連するすべての子ノードも非表示になります。ビューの上部には、記録されたすべてのトレースのうち、まだ表示されているコールトレースの数が表示されます。非表示のノードを再表示するには、非表示を表示ツールバーボタンをクリックできます。

## C.4 JavaScript XHR オリジントラッキング

JavaScript XHR オリジントラッキングを使用すると、ブラウザ内の異なるスタックトレースに対してサーブレット呼び出しを分割できます。 これにより、XMLHttpRequest [(1)] または Fetch [(2)] リクエスト中のプロファイルされた JVM のアクティビティをブラウザ内のアクションとより良く関連付けることができます。 以下では、「XHR」は XMLHttpRequest と Fetch の両方のメカニズムを指します。

### ブラウザプラグイン

この機能を使用するには、ブラウザとして Google Chrome [(3)] を使用し、 JProfiler オリジントラッカー拡張機能 [(4)] をインストールする必要があります。




Chrome 拡張機能は、ツールバーに JProfiler アイコンのボタンを追加し、トラッキングを開始します。トラッキングを開始すると、拡張機能はすべての XHR 呼び出しをインターセプトし、ローカルで実行中の　JProfiler　インスタンスに報告します。トラッキングが開始されていない限り、JProfiler は JavaScript XHR オリジントラッキングの設定方法を示す情報ページを表示します。

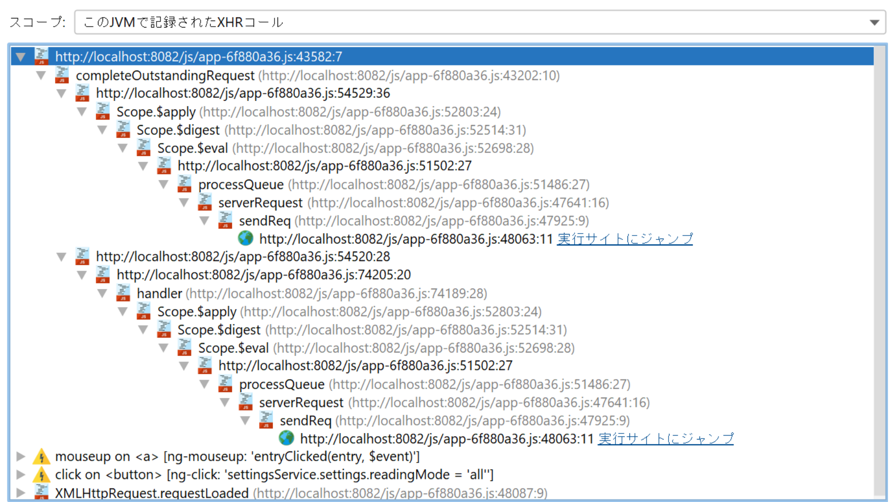
トラッキングがアクティブ化されると、JProfiler 拡張機能はページのリロードを求めます。これは計装を追加するために必要です。 ページをリロードしないことを選択した場合、イベント検出が機能しない可能性があります。

(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

トラッキングのステータスはドメインごとに永続的です。トラッキングがアクティブな状態でブラウザを再起動し、同じ URL を訪れると、 ページをリロードすることなくトラッキングが自動的に有効になります。

### JavaScript XHR ツリー

XHR 呼び出しが JProfiler のアクティブなプロファイリングセッションによってプロファイルされた JVM によって処理される場合、 JavaScript XHR ビューはこれらの呼び出しの累積された呼び出しツリーを表示します。ビューが空のままの場合は、ビューの上部にある「スコープ」を「すべての XHR 呼び出し」に切り替えて、XHR 呼び出しが行われたかどうかを確認できます。




JavaScript 呼び出しスタックノードには、ソースファイルと行番号に関する情報が含まれています。 XHR 呼び出しが行われた関数には、 特別なアイコンと、 プロファイルされた JVM によって処理された場合には隣接するハイパーリンクがあります。ハイパーリンクをクリックすると、呼び出しツリービュー [p. 54] の JavaScript 分割ノードに移動し、このタイプのリクエストを処理するために サーバー側で行われた呼び出しツリーを見ることができます。

ツリーの上部には、 ブラウザイベントノードがあり、 イベント名と要素名を重要な属性と共に表示し、イベントのソースを特定するのに役立ちます。すべてのリクエストに関連するイベントがあるわけではありません。

拡張機能は、いくつかの人気のある JavaScript フレームワークを認識しており、イベントのターゲットノードからイベントリスナーが配置されているノードまで祖先階層をたどり、表示に適した属性を探して呼び出しツリーを分割します。フレームワーク固有の属性が見つからない場合は、`id` 属性で停止します。ID がない場合は、`a`、`button`、`input` などの 「制御要素」を探します。すべて失敗した場合、イベントリスナーが登録されている要素が表示されます。

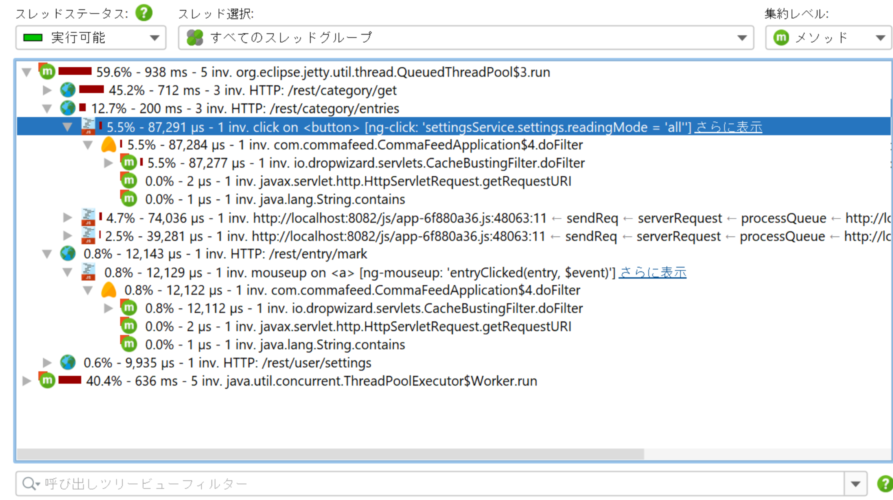
場合によっては、興味深い属性の自動検出が適切でないことがあり、異なる呼び出しツリーの分割を好むかもしれません。 たとえば、一部のフレームワークは自動的に ID を割り当てますが、アクションの意味的な説明と共にすべての要素をグループ化する方が読みやすい場合があります。異なる呼び出しツリーの分割を実現するには、HTML 属性を追加します。

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

ターゲット要素またはターゲットとイベントリスナーの位置の間の要素に追加します。その属性のテキストが分割に使用され、 他の属性は無視されます。

### 呼び出しツリーの分割

呼び出しツリービューでは、XHR呼び出しはブラウザイベントと呼び出しスタックの各組み合わせごとに呼び出しツリーを分割します。 分割ノードはブラウザイベントに関する情報を表示します。 イベントが進行中でない場合、たとえば `setTimeout()` の呼び出しでは、最後の数フレームがインラインで表示されます。




これらのノードの「詳細を表示」ハイパーリンクをクリックすると、ビュー->ノードの詳細を表示アクションによって開かれるのと同じ詳細ダイアログが開きます。JavaScript分割ノードの場合、詳細ダイアログにはノードのテキストではなく、ブラウザの呼び出しスタック全体が表示されます。 他の JavaScript 分割ノードの呼び出しスタックを調べるには、非モーダルの詳細ダイアログを開いたままにして、それらのノードをクリックします。詳細ダイアログは自動的に内容を更新します。

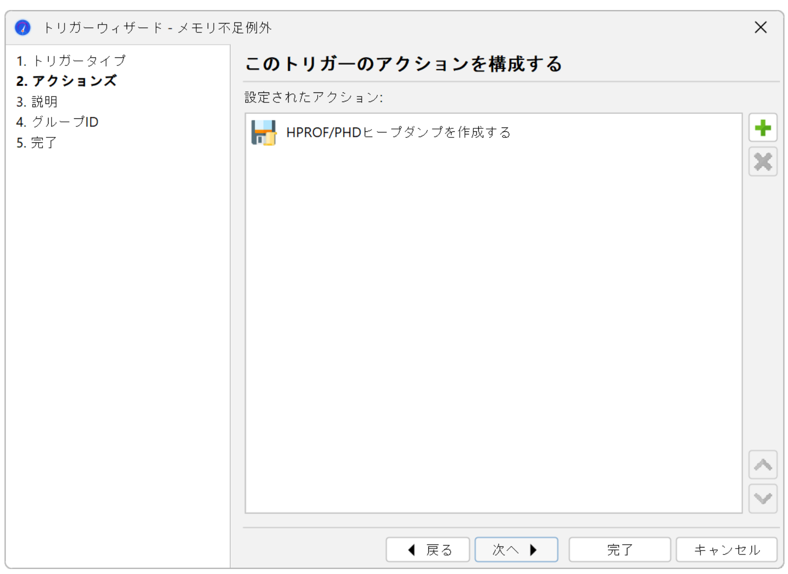
# D ヒープウォーカーの詳細機能

## D.1 HPROFおよびPHDヒープスナップショット

HotSpot JVMとAndroid RuntimeはどちらもHPROF形式でのヒープスナップショットをサポートしており、 IBM J9 JVMはPHD形式でそのようなスナップショットを書き込みます。PHDファイルにはガベージコレクタのルートが含まれていないため、JProfilerはクラスをルートとしてシミュレートします。クラスローダーのメモリリークをPHDファイルで見つけるのは難しいかもしれません。

ネイティブヒープスナップショットはプロファイリングエージェントなしで保存でき、JProfilerヒープスナップショットよりも低いオーバーヘッドを伴います。 これは、一般的なAPIの制約なしに保存されるためです。その反面、ネイティブヒープスナップショットはJProfilerヒープスナップショットよりも機能が少ないです。例えば、割り当て記録情報は利用できないため、オブジェクトがどこに割り当てられたかを見ることができません。HPROFおよびPHDスナップショットは、セッション->スナップショットを開くでJProfilerで開くことができ、JProfilerスナップショットを開くのと同じように操作します。ヒープウォーカーのみが利用可能で、他のセクションはグレーアウトされます。

ライブセッションでは、プロファイリング->HPROF/PHDヒープスナップショットを保存を呼び出すことで HPROF/PHDヒープスナップショットを作成して開くことができます。 オフラインプロファイリング[p.130]の場合、「HPROFヒープダンプを作成する」トリガーアクションがあります。これは通常、「メモリ不足例外」トリガーと共に使用され、`OutOfMemoryError`がスローされたときにHPROFスナップショットを保存します。




これは VMパラメータ [1] に対応しています。

```
-XX:+HeapDumpOnOutOfMemoryError
```

これはHotSpot JVMでサポートされています。

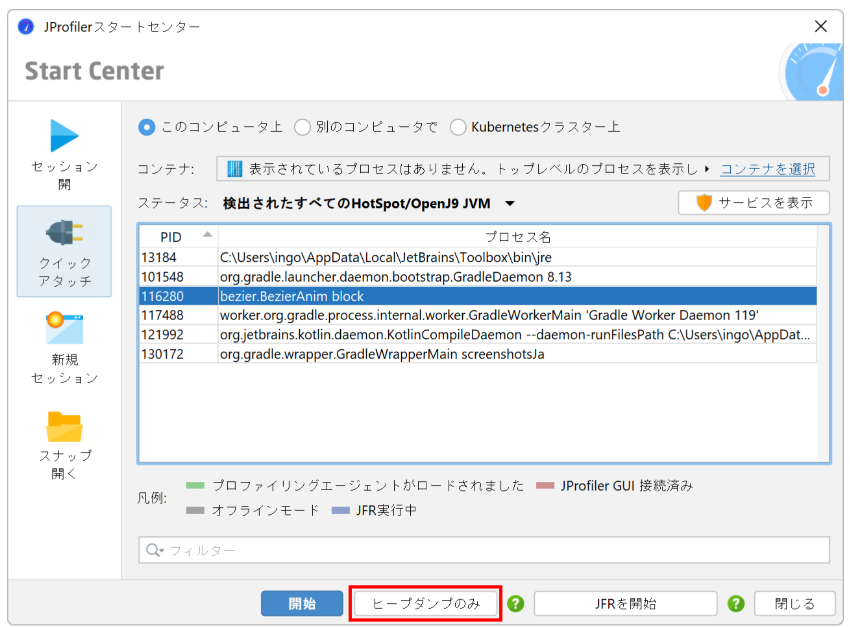
実行中のシステムからHPROFヒープダンプを抽出する別の方法は、JREの一部であるコマンドラインツール `jmap` を使用することです。その呼び出し構文

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

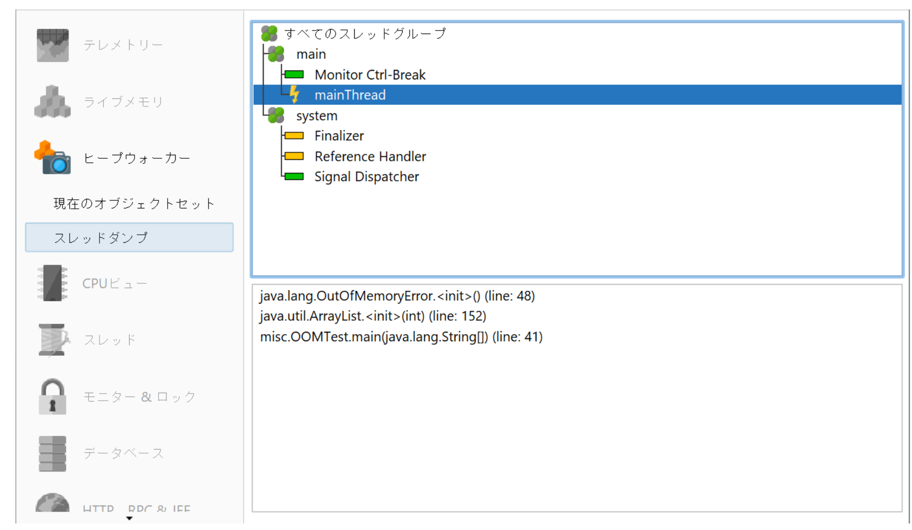
```
jmap -dump:live,format=b,file=<filename> <PID>
```

は覚えにくく、最初に`jps`実行ファイルを使用してPIDを確認する必要があります。 JProfilerにはインタラクティブなコマンドライン実行ファイル`bin/jpdump`が付属しており、これが非常に便利です。プロセスを選択でき、Windowsでサービスとして実行されているプロセスに接続でき、 混在した32ビット/64ビットJVMでも問題なく、HPROFスナップショットファイルを自動的に番号付けします。 より多くの情報を得るには`-help`オプションで実行してください。

プロファイリングエージェントをロードせずにHPROFヒープスナップショットを撮ることは、JProfiler GUIでもサポートされています。 プロセスにアタッチする際、ローカルでもリモートでも、常にHPROFヒープスナップショットを撮ることができます。




HPROFスナップショットにはスレッドダンプが含まれることがあります。HPROFスナップショットが `OutOfMemoryError`の結果として保存された場合、スレッドダンプはエラー発生時にアプリケーションのどの部分がアクティブであったかを伝えることができるかもしれません。エラーをトリガーしたスレッドは特別なアイコンでマークされています。



テレメトリー
ライブメモリ
ヒープウォーカー
現在のオブジェクトセット
スレッドダンプ
CPUビュー
スレッド
モニター & ロック
データベース
すべてのスレッドグループ
main
Monitor Ctrl-Break
mainThread
system
Finalizer
Reference Handler
Signal Dispatcher
java.lang.OutOfMemoryError.<init>() (line: 48)
java.util.ArrayList.<init>(int) (line: 152)
misc.OOMTest.main(java.lang.String[]) (line: 41)

## D.2 ヒープウォーカーでのオーバーヘッドの最小化

小さなヒープの場合、ヒープスナップショットを取得するのに数秒かかりますが、非常に大きなヒープでは、これは長いプロセスになることがあります。物理メモリが不足していると、計算が非常に遅くなることがあります。たとえば、JVMが50 GBのヒープを持ち、ローカルマシンで5 GBの空き物理メモリしかない場合、JProfilerは特定のインデックスをメモリに保持できず、処理時間が不釣り合いに増加します。

JProfilerは主にヒープ分析にネイティブメモリを使用するため、`-Xmx` 値を `bin/jprofiler.vmoptions` ファイルで増やすことは、`OutOfMemoryError` が発生し、JProfilerがそのような変更を指示した場合を除いて推奨されません。ネイティブメモリは利用可能であれば自動的に使用されます。分析が完了し、内部データベースが構築された後、ネイティブメモリは解放されます。

ライブスナップショットの場合、ヒープダンプを取得した直後に分析が計算されます。スナップショットを保存すると、分析はスナップショットファイルの隣に `.analysis` サフィックスを持つディレクトリに保存されます。スナップショットファイルを開くと、ヒープウォーカーは非常に迅速に利用可能になります。`.analysis` ディレクトリを削除すると、スナップショットを開いたときに計算が再度実行されるため、スナップショットを他の人に送る場合、分析ディレクトリを一緒に送る必要はありません。

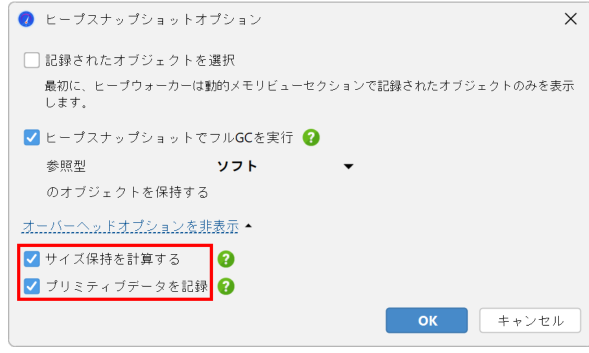
ディスク上のメモリを節約したい場合や、生成された `.analysis` ディレクトリが不便な場合は、一般設定でその作成を無効にすることができます。

HPROFスナップショットとオフラインプロファイリングで保存されたJProfilerスナップショットには、`.analysis` ディレクトリが隣にありません。なぜなら、分析はJProfiler UIによって行われ、プロファイリングエージェントによって行われないからです。そのようなスナップショットを開くときに計算を待ちたくない場合は、`jpanalyze` コマンドライン実行可能ファイルを使用してスナップショットを事前分析 [p. 256]することができます。

スナップショットは書き込み可能なディレクトリから開くことをお勧めします。分析なしでスナップショットを開き、そのディレクトリが書き込み可能でない場合、分析には一時的な場所が使用されます。そのため、スナップショットを開くたびに計算を繰り返す必要があります。

分析の大部分は保持サイズの計算です。処理時間が長すぎる場合で保持サイズが必要ない場合は、ヒープウォーカーオプションダイアログのオーバーヘッドオプションでその計算を無効にすることができます。保持サイズに加えて、その場合「最大オブジェクト」ビューも利用できなくなります。プリミティブデータを記録しないことでヒープスナップショットが小さくなりますが、参照

ビューでそれらを見ることはできません。同じオプションは、ファイル選択ダイアログで分析をカスタマイズを選択した場合にスナップショットを開くときに提示されます。

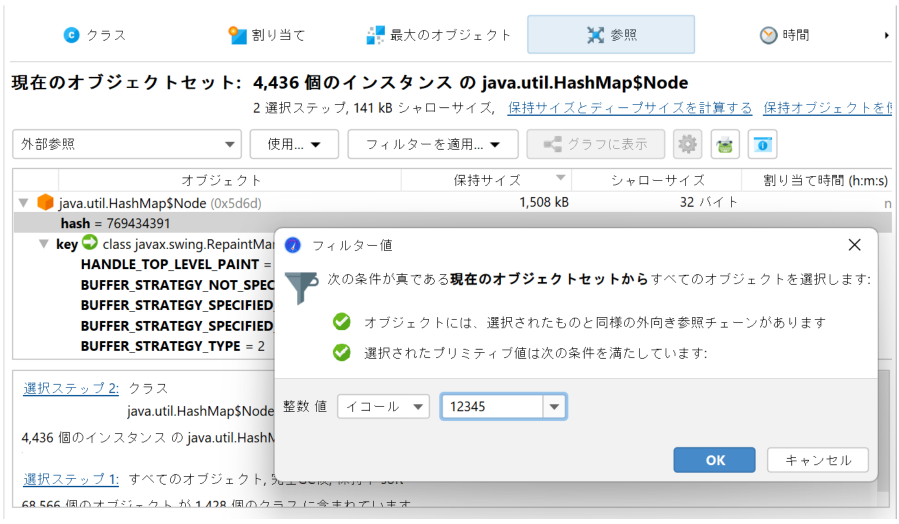


## D.3 フィルターとライブインタラクション

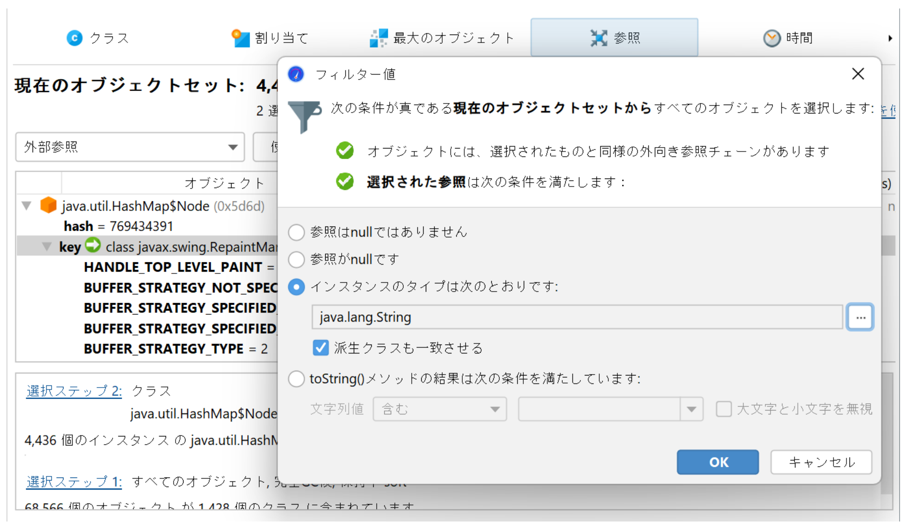
ヒープウォーカーで興味のあるオブジェクトを探す際、同じクラスのインスタンスが多すぎるオブジェクトセットに到達することがあります。特定のフォーカスに応じてオブジェクトセットをさらに絞り込むために、選択基準にはそのプロパティや参照が含まれることがあります。例えば、特定の属性を含むHTTPセッションオブジェクトに興味があるかもしれません。ヒープウォーカーのマージされたアウトゴーイング参照ビューでは、オブジェクトセット全体の参照チェーンを含む選択ステップを実行できます。

しかし、個々のオブジェクトを表示するアウトゴーイング参照ビューでは、参照とプリミティブフィールドを制約する選択ステップを行うためのより強力な機能を提供します。

トップレベルのオブジェクト、プリミティブ値、またはアウトゴーイング参照ビューの参照を選択すると、フィルターを適用->選択した値を制限することでアクションが有効になります。選択に応じて、フィルタ値ダイアログは異なるオプションを提供します。どのオプションを設定しても、新しいオブジェクトセットのオブジェクトが選択されたものと同様のアウトゴーイング参照チェーンを持たなければならないという制約を常に暗黙的に追加します。フィルターは常にトップレベルのオブジェクトに対して機能し、現在のオブジェクトセットをおそらく小さなセットに制限します。

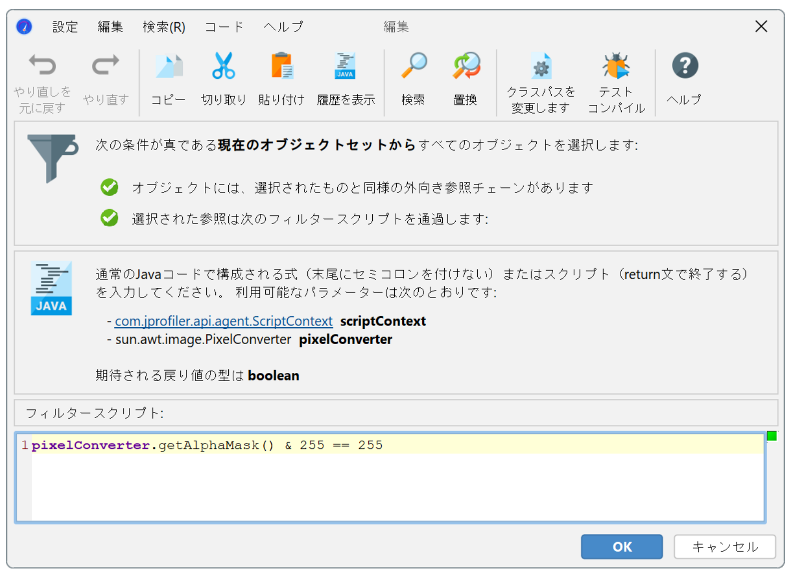



プリミティブ値の制約は、HPROFおよびJProfilerヒープスナップショットの両方で機能します。参照型の場合、JProfilerに非null値、null値、および選択したクラスの値をフィルタリングするように依頼できます。`toString()`メソッドの結果によるフィルタリングは、ライブセッションでのみ利用可能ですが、`java.lang.String`および`java.lang.Class`オブジェクトの場合、JProfilerはこれを自動的に判断できます。




最も強力なフィルタータイプは、コードスニペットを使用するものです。オブジェクトをフィルタリングするための基本的に異なる2つの方法があります：

ライブセッションでは、JProfilerはプロファイルされたJVMでフィルタリングスクリプトを実行し、実際のインスタンスをスクリプトに渡すことができます。ライブオブジェクトでスクリプトを実行してフィルターを適用によって表示されるスクリプトエディタでは、プロパティに直接アクセスする式やスクリプトを書き、そのブール値の戻り値がインスタンスが現在のオブジェクトセットに保持されるべきかどうかを決定します。

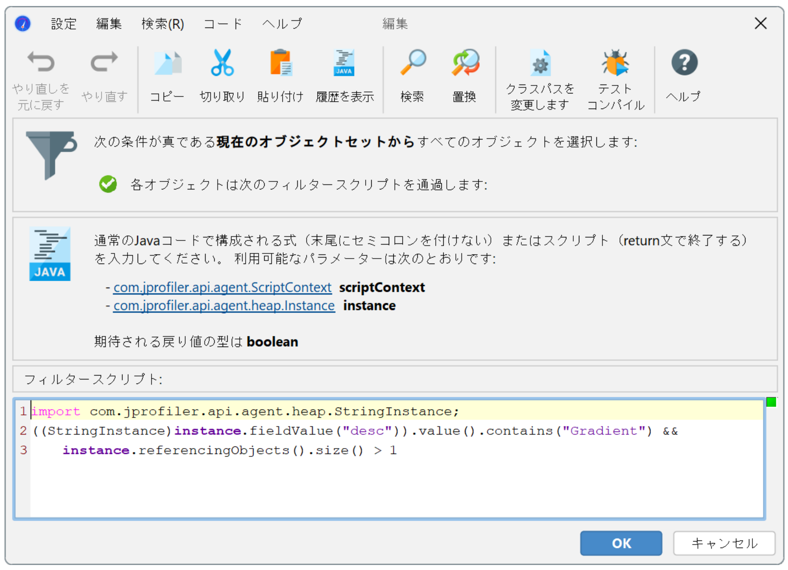



当然、この機能はライブセッションでのみ機能します。なぜなら、JProfilerはライブオブジェクトにアクセスする必要があるからです。考慮すべきもう一つの要因は、ヒープスナップショットが取得された後にオブジェクトがガベージコレクトされている可能性があることです。その場合、コードスニペットフィルターが実行されたときに、そのようなオブジェクトは新しいオブジェクトセットに含まれません。

HPROFやPDHスナップショットを含むスナップショットでも機能するもう一つのオプションは、ダンプされたデータでスクリプトを実行してフィルターを適用アクションです。各インスタンスは、`com.jprofiler.api.agent.heap.HeapObject`のインスタンスとしてスクリプトに渡されます。適用可能な場合、パラメータをダウンキャストできるいくつかのサブインターフェースがあり

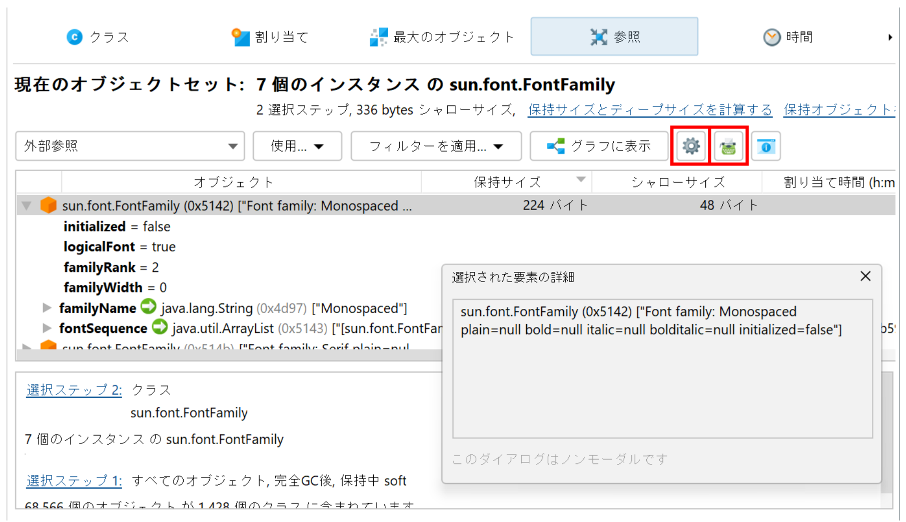
ます。詳細については、Javadocを参照してください。例えば、オブジェクトがオブジェクトインスタンスであり、フィールド値へのアクセスを提供する場合、`com.jprofiler.api.agent.heap.Instance`インターフェースが利用可能です。スクリプトがトップレベルのオブジェクトで動作し、現在のオブジェクトセットのすべてのオブジェクトが同じタイプである場合、スクリプトパラメータは自動的に適切なサブタイプを持ちます。

これらのフィルタースクリプトでは、`HeapObject`パラメータのメソッドを通じて、すべてのインカミングおよびアウトゴーイング参照にもアクセスできます。

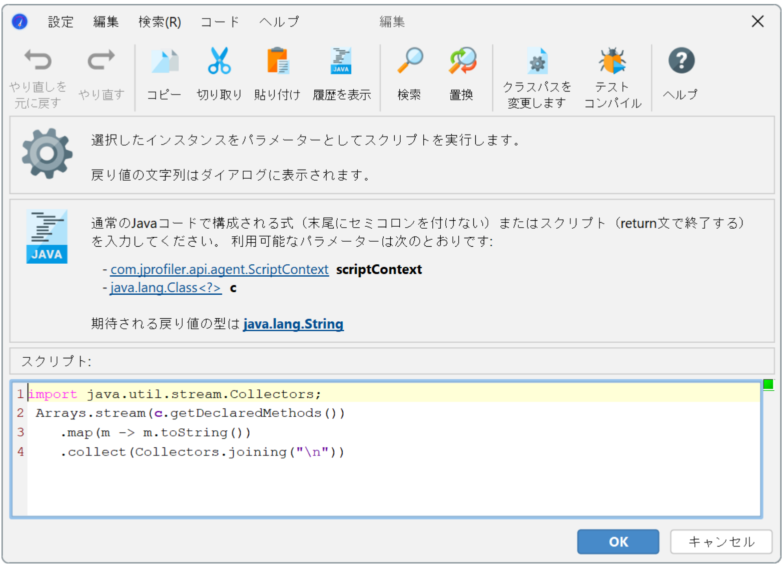



PHDスナップショットにはフィールド情報が含まれていないため、そのようなスナップショットではすべてのインスタンスが`com.jprofiler.api.agent.heap.HeapObject`または`com.jprofiler.api.agent.heap.ClassObject`として渡され、フィールド値は`referencedObjects()`メソッドを通じてのみアクセスできます。

フィルター以外にも、個々のオブジェクトと対話するためのアウトゴーイング参照ビューには2つの他の機能があります：toString()値を表示アクションは、現在ビューに表示されているすべてのオブジェクトで`toString()`メソッドを呼び出し、それらを参照ノードに直接表示します。ノードは非常に長くなる可能性があり、テキストが切り取られることがあります。コンテキストメニューからノードの詳細を表示アクションを使用すると、全体のテキストを確認できます。




`toString()`メソッドを呼び出すよりもオブジェクトから情報を取得するためのより一般的な方法は、文字列を返す任意のスクリプトを実行することです。スクリプトを実行アクションは、toString()値を表示アクションの隣にあり、トップレベルのオブジェクトまたは参照が選択されているときにそれを行うことができます。スクリプトの実行結果は、別のダイアログに表示されます。

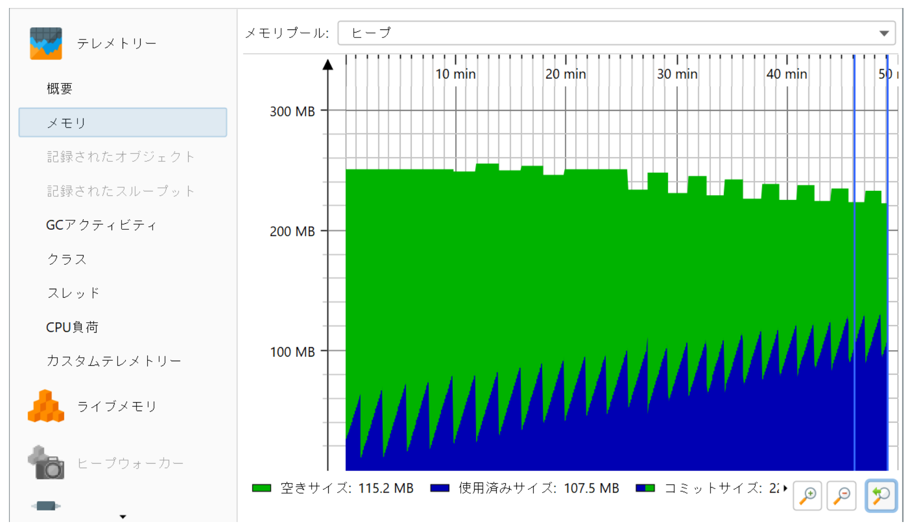


## D.4 メモリリークの発見

通常のメモリ使用量とメモリリークを区別するのは、しばしば簡単ではありません。しかし、過剰なメモリ使用量とメモリリークは同じ症状を持っているため、同じ方法で分析することができます。分析は2つのステップで進行します:疑わしいオブジェクトを特定し、それらのオブジェクトがなぜヒープに残っているのかを見つけ出すことです。

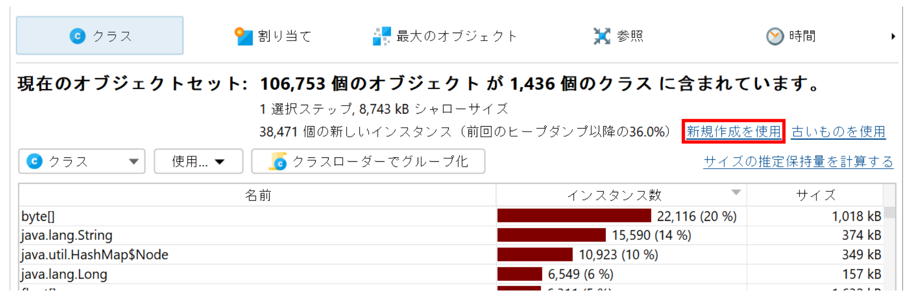
### 新しいオブジェクトの発見

メモリリークのあるアプリケーションが実行されていると、時間とともにますます多くのメモリを消費します。 メモリ使用量の増加を検出するには、VMテレメトリーと 差分機能 [p. 72]を「すべてのオブジェクト」と「記録されたオブジェクト」ビューで使用するのが最適です。これらのビューを使用すると、問題があるかどうか、またその深刻さを判断できます。時には、コールヒストグラムテーブルの差分列が問題の手がかりを与えてくれることもあります。




メモリリークのより深い分析には、ヒープウォーカーの機能が必要です。特定のユースケースに関するメモリリークを詳細に調査するには、「ヒープをマーク」機能 [p. 81]が最適です。これにより、特定の過去の時点からヒープに残っている新しいオブジェクトを特定できます。これらのオブジェクトについては、それらがまだ正当にヒープに存在するのか、誤った参照がそれらを生かし続けているのかを確認する必要があります。

興味のあるオブジェクトのセットを分離するもう一つの方法は、割り当て記録を通じて行うことです。 ヒープスナップショットを取る際に、すべての記録されたオブジェクトを表示するオプションがあります。しかし、割り当て記録を特定のユースケースに限定したくないかもしれません。また、割り当て記録は高いオーバーヘッドを持つため、ヒープをマークアクションは比較的に影響が小さくなります。最後に、ヒープウォーカーは、ヒープをマークした場合、新しいものを使用および古いものを使用ハイパーリンクを使用して、任意の選択ステップで古いオブジェクトと新しいオブジェクトを選択できます。




## 最大のオブジェクトの分析

メモリリークが利用可能なヒープを埋めると、プロファイルされたアプリケーションの他のタイプのメモリ使用量を圧倒します。その場合、新しいオブジェクトを調べる必要はなく、単に最も重要なオブジェクトを分析するだけです。

メモリリークは非常に遅い速度で発生し、長い間支配的にならないことがあります。そのようなメモリリークをプロファイリングして可視化するまで待つのは実用的ではないかもしれません。`OutOfMemoryError`がスローされたときにHPROFスナップショットを自動的に保存するJVMの組み込み機能を使用すると、通常のメモリ消費よりもメモリリークが重要なスナップショットを取得できます。実際、常に追加することをお勧めします。

```
-XX:+HeapDumpOnOutOfMemoryError
```

VMパラメータや本番システムに追加することで、開発環境で再現が難しいメモリリークを分析する方法を持つことができます。

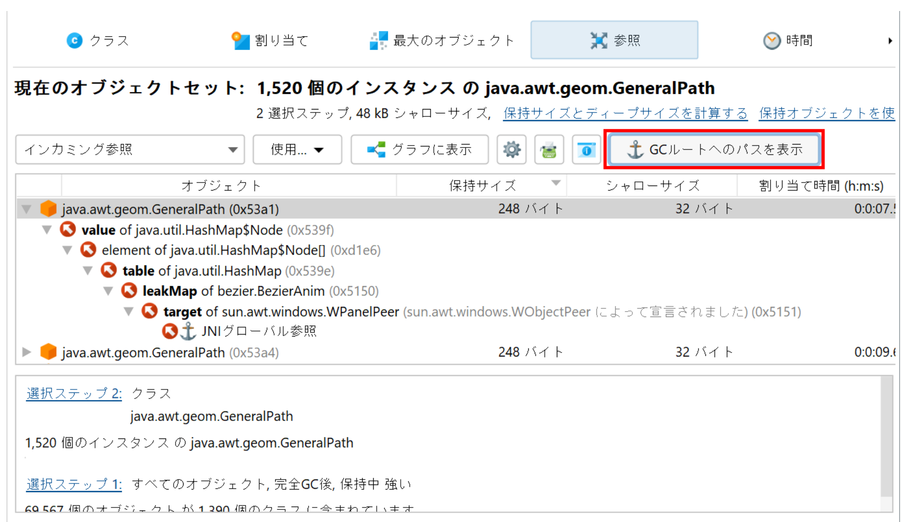
メモリリークが支配的である場合、ヒープウォーカーの「最大のオブジェクト」ビューのトップオブジェクトには、誤って保持されたメモリが含まれます。最大のオブジェクト自体は正当なオブジェクトであるかもしれませんが、それらのドミネーターツリーを開くと、リークされたオブジェクトにたどり着きます。単純な状況では、ヒープの大部分を含む単一のオブジェクトがあります。たとえば、マップがオブジェクトをキャッシュするために使用され、そのキャッシュが決してクリアされない場合、マップは最大のオブジェクトのドミネーターツリーに表示されます。

## ガベージコレクタルートからの強い参照チェーンの発見

オブジェクトが問題になるのは、それが強く参照されている場合のみです。「強く参照されている」とは、ガベージコレクタルートからオブジェクトへの参照チェーンが少なくとも1つあること

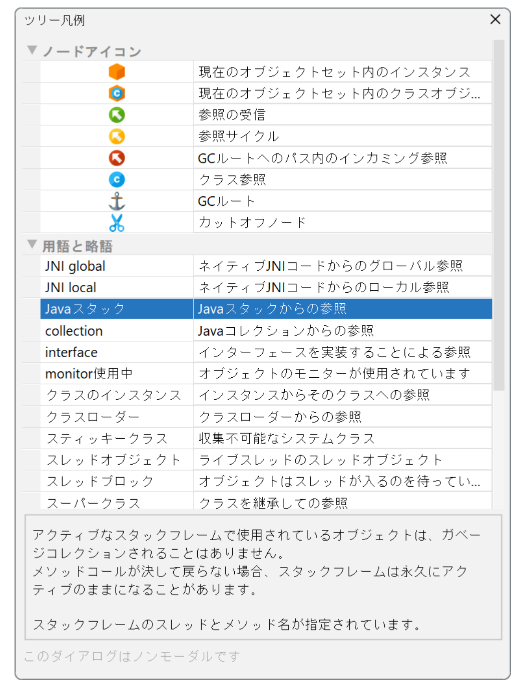
を意味します。「ガベージコレクタルート」（略してGCルート）は、JVMが知っている特別な参照です。

GCルートからの参照チェーンを見つけるには、「インカミング参照」ビューまたはヒープウォーカーグラフでGCルートへのパスを表示アクションを使用できます。このような参照チェーンは実際には非常に長いことがあるため、「インカミング参照」ビューで一般的により簡単に解釈できます。参照は下から上位レベルのオブジェクトに向かって指します。検索の結果である参照チェーンのみが展開され、同じレベルの他の参照はノードが閉じて再度開かれるか、コンテキストメニューですべてのインカミング参照を表示アクションが呼び出されるまで表示されません。




参照ノードで使用されるGCルートのタイプやその他の用語の説明を得るには、ツリーレジェンドを使用してください。

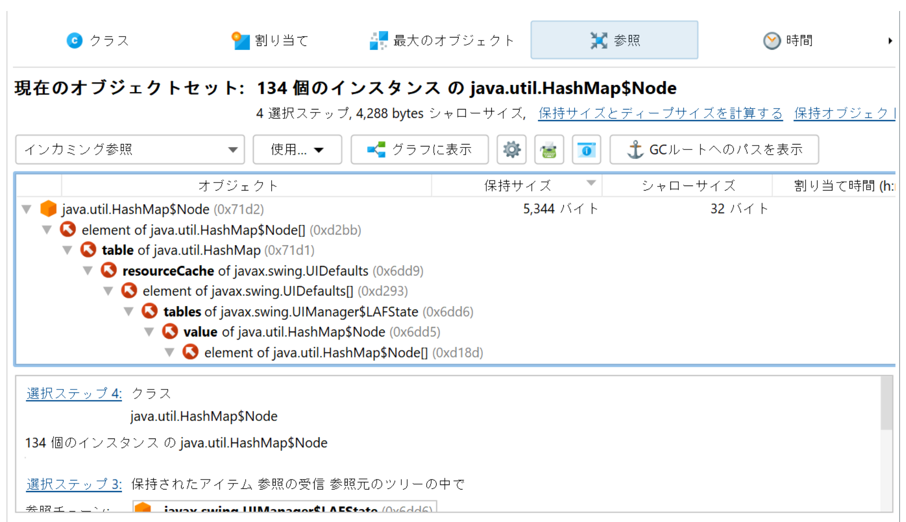
ツリー内のノードを選択すると、非モーダルのツリーレジェンドが選択されたノードで使用されているすべてのアイコンと用語を強調表示します。ダイアログ内の行をクリックすると、下部に説明が表示されます。



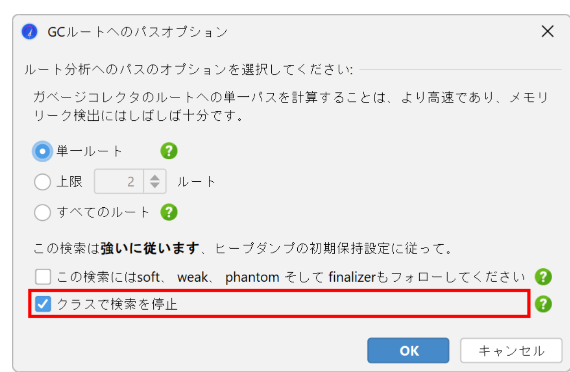

重要なタイプのガベージコレクタルートは、スタックからの参照、JNIを通じてネイティブコードによって作成された参照、および現在使用中のライブスレッドやオブジェクトモニターなどのリソースです。さらに、JVMは重要なシステムを維持するためにいくつかの「スティッキー」参照を追加します。

クラスとクラスローダーには特別な循環参照スキームがあります。クラスは、そのクラスローダーによってロードされたクラスがライブインスタンスを持たない場合に、クラスローダーと一緒にガベージコレクトされます。

- そのクラスローダーによってロードされたクラスがライブインスタンスを持たない
- クラスローダー自体がそのクラスによってのみ参照されている
- `java.lang.Class`オブジェクトがクラスローダーのコンテキストでのみ参照されている

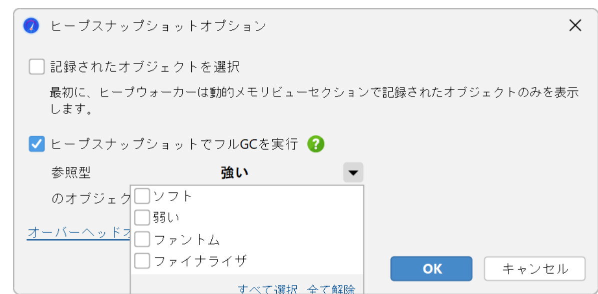


ほとんどの状況では、クラスは興味のあるGCルートへのパスの最後のステップです。 クラス自体はGCルートではありません。しかし、カスタムクラスローダーが使用されていないすべての状況では、それらをそのように扱うのが適切です。これは、ガベージコレクタルートを検索する際のJProfilerのデフォルトモードですが、ルートオプションダイアログへのパスで変更できます。



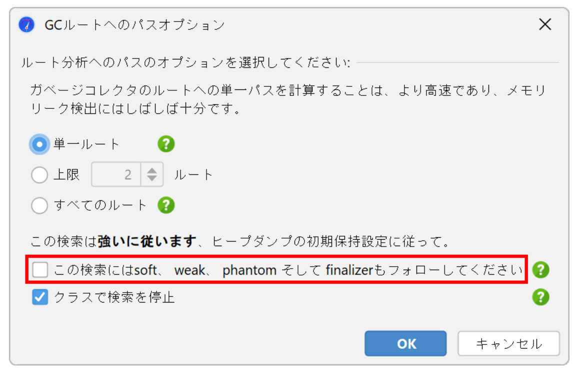

GCルートへの最短パスを解釈するのに問題がある場合は、追加のパスを検索できます。一般的に、GCルートへのすべてのパスを検索することは推奨されません。なぜなら、それは大量のパスを生成する可能性があるからです。

ライブメモリビューとは対照的に、ヒープウォーカーは未参照のオブジェクトを表示しません。しかし、ヒープウォーカーは強く参照されているオブジェクトだけを表示するわけではありません。デフォルトでは、ヒープウォーカーはソフト参照によってのみ参照されているオブジェクトも保持しますが、弱い参照、ファントム参照、またはファイナライザ参照によってのみ参照されているオブジェクトは排除します。ソフト参照はヒープが枯渇しない限りガベージコレクトされないため、それらを含めることで大きなヒープ使用量を説明できるようにしています。ヒープスナップショットを取る際に表示されるオプションダイアログで、この動作を調整できます。



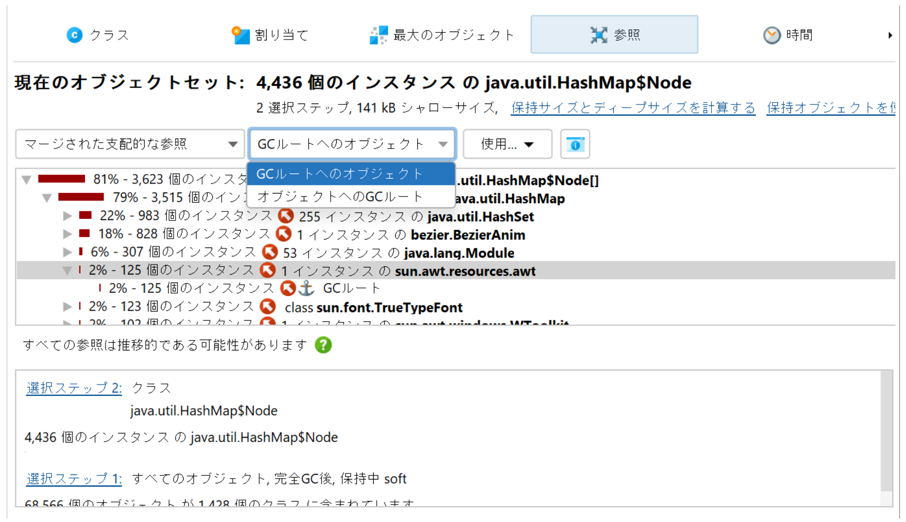

ヒープウォーカーに弱く参照されているオブジェクトを含めることは、デバッグ目的で興味深いかもしれません。後で弱く参照されているオブジェクトを削除したい場合は、「弱い参照によって保持されているオブジェクトを削除」検査を使用できます。

GCルートへのパスを検索する際、ヒープウォーカーオプションダイアログでオブジェクトを保持するために選択された参照タイプが考慮されます。このようにして、GCルート検索へのパスは常にオブジェクトがヒープウォーカーに保持された理由を説明できます。GCルート検索へのパスのオプションダイアログでは、許容される参照タイプをすべての弱い参照に拡大できます。




### オブジェクトセット全体の排除

これまでのところ、単一のオブジェクトについてのみ検討してきました。多くの場合、メモリリークの一部である同じタイプの多くのオブジェクトを持つことがあります。現在のオブジェクトセット内の他のオブジェクトにも有効である場合が多いです。興味のあるオブジェクトが異なる方法で参照されている場合のより一般的なケースでは、「マージされた支配参照」ビューが、現在のオブジェクトセットをヒープに保持している参照を見つけるのに役立ちます。




支配参照ツリーの各ノードは、その参照を排除した場合に現在のオブジェクトセット内のどれだけのオブジェクトがガベージコレクションの対象になるかを示します。複数のガベージコレクタルートによって参照されているオブジェクトは、支配的なインカミング参照を持たない場合があるため、ビューはオブジェクトの一部にしか役立たないか、場合によっては空であることもあります。その場合、マージされたインカミング参照ビューを使用し、ガベージコレクタルートを一つずつ排除する必要があります。

# E JDKフライトレコーダー (JFR)

## E.1 JDK Flight Recorder (JFR) のサポート

JDK Flight Recorder (JFR) [(1)] は、システムレベルのイベントを幅広く記録する構造化されたロギングツールです。航空機のブラックボックスがフライトデータを継続的に記録し、事故調査に使用されるのと同様に、JFRはJVM内のイベントストリームを継続的に記録し、問題の診断に使用されます。このアプローチの利点は、インシデントに至るまでのシステムの詳細な情報を時系列でキャプチャすることです。JFR はパフォーマンスへの影響を最小限に抑えるように設計されており、長期間にわたって本番環境で安全に実行できます。

Java 17 以降、JFR は JProfiler のデータソースの一つでもあります。JVM のプロファイリングインターフェースを使用するネイティブエージェントに加えて、プロファイリングコンテキストで興味深い JVM の高レベルシステムがあります。一つは JProfiler のいくつかのテレメトリーにデータを提供する MBean システムであり、もう一つは ガベージコレクタープローブ [p. 119] に使用される JFR です。その目的のために、JFR と直接やり取りすることはなく、JProfiler が JFR イベントストリーミングを透過的に処理します。

### JProfiler における JFR の統合

JProfiler は JFR 記録 [p. 225] を完全に統合しているため、ローカルマシンまたは JFR 記録が設定されていないリモートマシンで実行中の JVM からデータを簡単にキャプチャできます。

JProfiler UI で JFR スナップショットを開くと、利用可能なビューとセクションは通常のプロファイリングセッションとは異なります。UI の中心はイベントブラウザ [p. 229] です。JFR ビューに利用可能な他のすべてのビューは、別の章で説明されています [p. 236]。

イベントタイプを操作し、フィルタを設定し、分析を表示する際に、JProfiler は時折 JFR スナップショットファイルを再スキャンする必要があります。JFR スナップショットファイルは非常に大きくなる可能性があり、すべてのデータをメモリに保持したり、すべての分析を事前に計算したりすることは現実的ではありません。このため、ネットワークドライブから JFR スナップショットを開くことは推奨されません。

非常に大きな JFR スナップショットを開く際には、ファイル選択ダイアログで「分析をカスタマイズ」チェックボックスをクリックし、分析に必要ないイベントカテゴリを除外することで、スナップショット処理を高速化し、メモリ使用量を削減できます。利用可能なイベントカテゴリは、単一のプローブとビューセクションをカバーしています。CPU ビュー、メモリビュー、およびテレメトリービューのイベントタイプはオプションではなく、ロードする必要があります。

例えば、CPU データのみに興味がある場合は、すべてのプローブとイベントブラウザを除外できます。JProfiler は最速の JFR ビューアを目指しており、典型的な JFR スナップショットを迅速に開きますが、JFR 記録は潜在的に無制限であり、開く速度が問題になる可能性がある数十ギガバイトのスナップショットに直面することがあります。

### JFR スナップショットのスタックトレース

JFR の重要な機能の一つは、特定のイベントタイプに対して効率的に全スタックトレースを記録する能力です。そのようなイベントタイプでは、JFR 設定でスタックトレース記録を切り替えることができます。多くの JVM アプリケーションイベントタイプ、特にスレッドに関するものは、デフォルトでスタックトレース記録が有効になっています。

JFR は固定の深さまでしかスタックトレースを収集しないため、長いスタックトレースは切り捨てられます。切り捨てられたトレースは理解可能な呼び出しツリーを構築するのに適していないため、これらのトレースは特別にマークされたノードの下に表示されます。

```
-XX:FlightRecorderOptions=stackdepth=<nnnn>
```

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

VM パラメータを使用して、JFR で収集されるトレースのサイズを増やし、アプリケーションの切り捨てられたトレースを排除することができます。

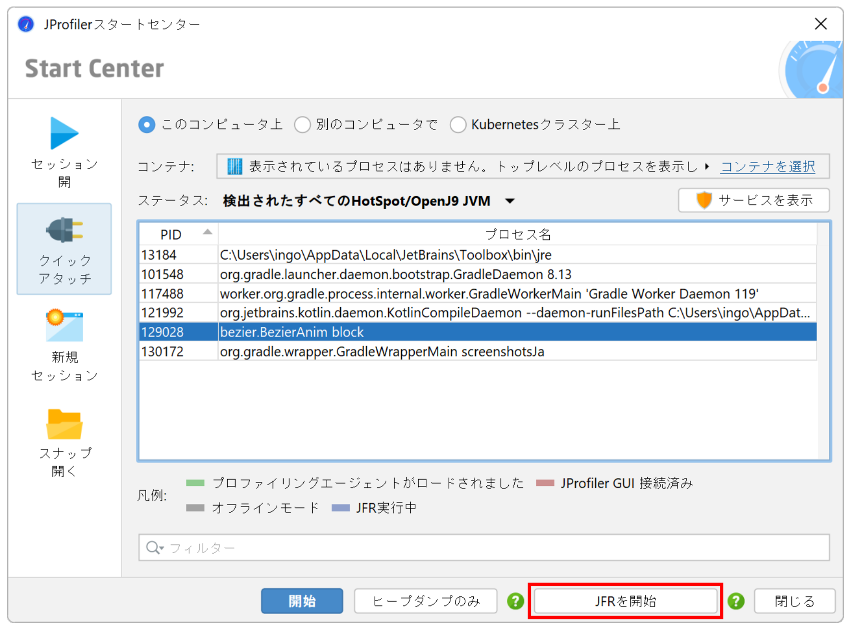
## E.2 JProfilerでJFRスナップショットを記録する

JFRをプロダクションJVMで実行することの利点は、オーバーヘッドが最小限であり、プロファイリングインターフェースを有効にする必要がないことです。JProfilerはUIで直接JFR記録をサポートしています。JFRをプログラムで開始するか、コマンドラインで`-XX:StartFlightRecording` VMパラメータを追加することもできますが、JProfilerは既に実行中のJVMの記録を開始および停止するのに役立ちます。

JProfilerでJVMにアタッチする際、ネイティブプロファイリングエージェントをロードする代わりに、JFR記録を開始および停止することを選択できます。JProfilerの広範なリモート接続機能を使用すると、たとえば、DockerやKubernetesコンテナで実行されているJVMでJFR記録を開始することができ、コンテナを変更する必要はありません。

### JFR記録の開始と停止

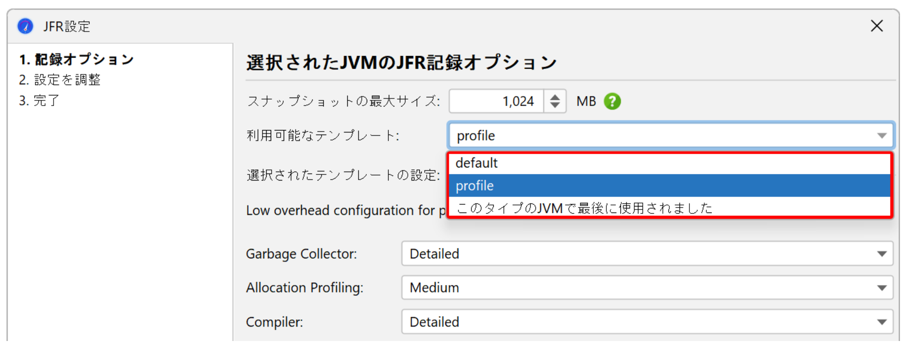
スタートセンターの「クイックアタッチ」タブでJVMを選択し、ダイアログの下部にあるStart JFRボタンをクリックします。ローカルで実行中のJVMがスクリーンショットに表示されていますが、同じボタンはリモートJVMにアタッチする際にも使用できます。




JFR設定ウィザードでは、選択したプロセスで使用されているJREの`lib/jfr`ディレクトリから送信される**イベント設定テンプレート**の1つを選択できます。デフォルトでは、「default」と「profile」という2つのテンプレートがあり、「profile」はより多くのデータを記録し、オーバーヘッドが増加します。そのディレクトリに他のファイルを作成すると、ウィザードで対応するテンプレートを選択できるようになります。

これらのテンプレートファイルには、利用可能なイベントと重要な**高レベル設定**の構成指令が含まれています。各高レベル設定は、さまざまなイベントに結び付けることができます。このUIはテンプレートファイルの内容に基づいて動的に生成されます。異なるプロファイル間を切り替えると、異なるデフォルト値が表示されます。このUIに含まれていない多くのイベントタイプがあり、次のステップでのみ構成可能です。

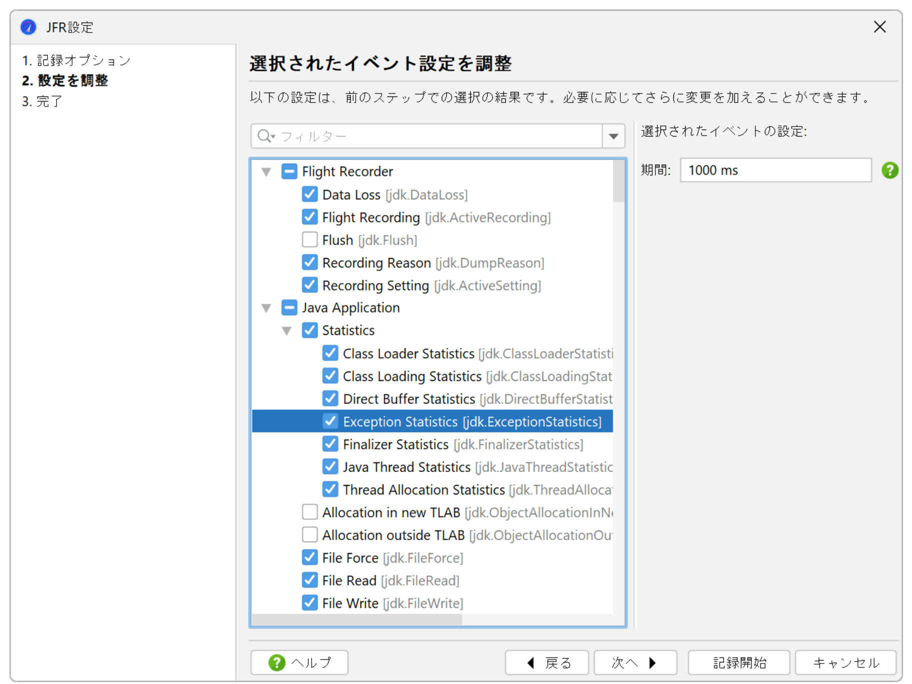
すでに同じイベントタイプのセットでJVMのJFR記録を開始している場合、JProfilerは**最後の設定**を使用するオプションを提供します。




そのオプションを選択すると、高レベルの記録設定は利用できなくなり、次のステップに進んで全体の構成を確認し、さらに変更を加えることができます。

ウィザードのこのステップでのもう1つの重要な設定は**最大スナップショットサイズ**です。JFR記録の性質上、スナップショットのサイズは非常に迅速に増加し、ハードディスク全体を埋め尽くす可能性があります。それを避けるために、最大スナップショットサイズの制約は過剰なストレージ使用を防ぎます。最大サイズに達すると、古いイベントは破棄され、新しいイベントは引き続き記録されます。 このプロセスはJFRの自動メカニズムです。

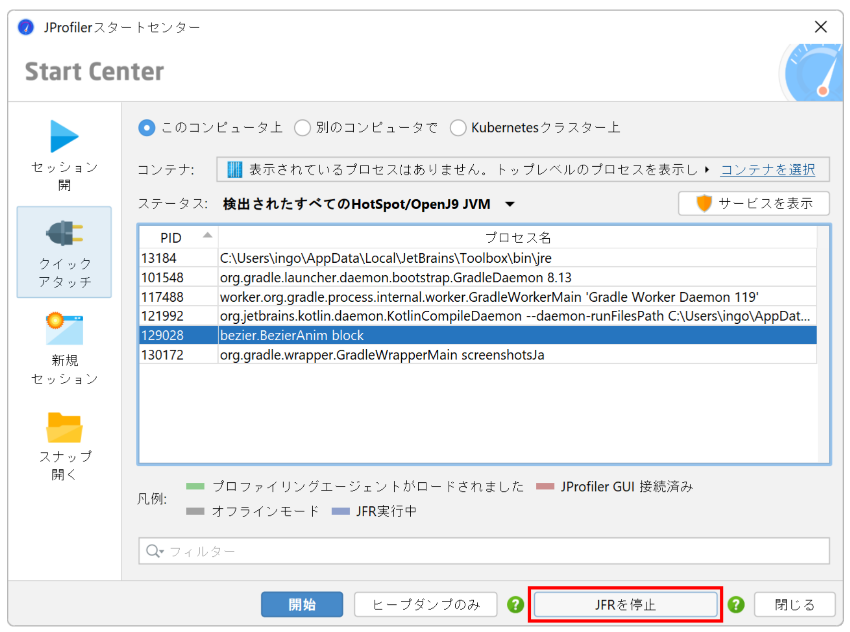
ウィザードの次のステップでは、すべてのイベントタイプのカテゴリ化されたツリーを確認し、右側で各イベントのさらなる構成を行うことができます。




イベントには**期間**、**しきい値**、各イベントの**スタックトレース**を記録するかどうかのフラグの設定がある場合があります。期間としきい値は時間単位の設定であり、downキーを押すと利用可能な単位のオートコンプリートポップアップが表示されます。期間はまた、オートコンプリートポップアップからも利用可能な特別な値「everyChunk」、「beginChunk」、「endChunk」をサポートしています。「チャンク」は、連続したイベントデータとメタデータのセットを保持し、記録における基本的なストレージおよびデータ転送の単位として機能するJFR記録の一部を指します。

ツリーで選択されるイベントが多いほど、記録されるデータが増えます。いくつかのイベントタイプは大量のデータを生成し、いくつかは少量のイベントしか生成しません。

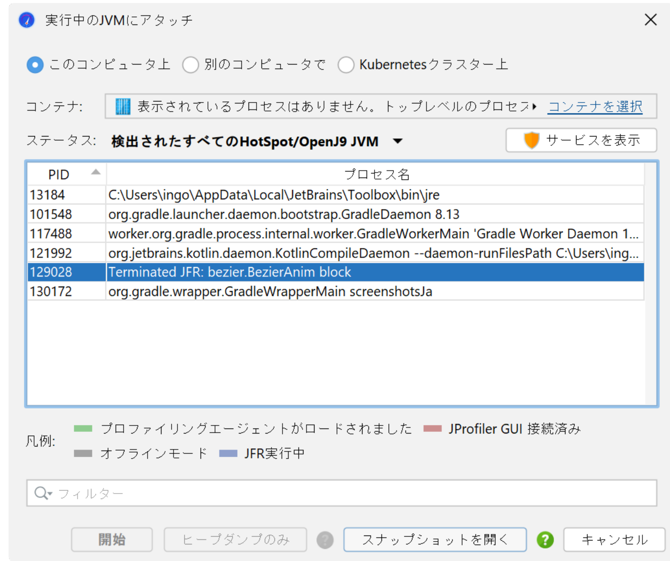
フルプロファイリングモードや「ヒープダンプのみ」モードとは異なり、UIですぐにデータが表示されるわけではありませんが、JFRスナップショットを開始すると、選択されていないときにJVMの背景色が変更され、JProfilerが記録を開始したことがわかります。JVMが選択されている場合、下部のJFRボタンのテキストは記録が停止されることを示します。




JProfilerによって開始されたJFR記録を停止すると、JFRスナップショットが転送され、JProfilerで開かれます。**スナップショットは一時的**であり、ウィンドウを閉じると削除されます。スナップショットを永続的な場所に保存するには、ツールバーの「スナップショットを保存」アクションを使用してください。

## JFR記録を持つ終了したJVM

JFRの使用の1つとして、クラッシュ前の瞬間を調査することが挙げられます。その場合、JVMはJVMテーブルに表示されなくなり、JFR記録を停止してJFRスナップショットを開くことができません。JProfilerでJFR記録が開始され、記録を停止する前にJVMが終了した場合、「Terminated JFR:」というプレフィックスが付いた特別なエントリがJVMテーブルに追加されます。そのエントリをダブルクリックするか、「JFR」ボタンを使用して、JFRスナップショットを開くことができます。



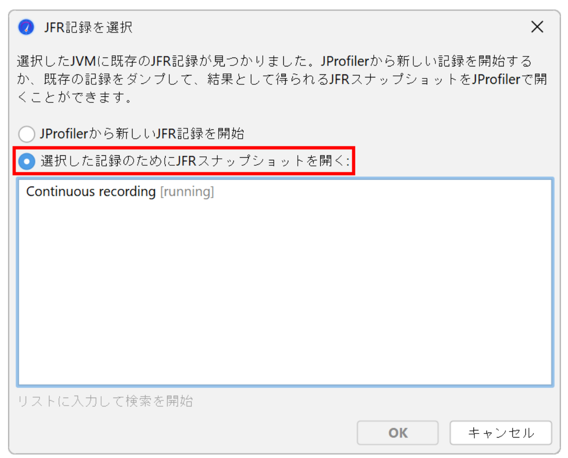

そのようなエントリを開くと、リストから削除されます。手動で停止された記録と同様に、開かれたJFRスナップショットは一時的であり、後で分析するために保持したい場合は保存する必要があります。

### 外部で開始されたJFR記録の表示

上記の例では、JFR記録はJProfilerで開始および停止されました。JProfilerの外部で開始されたJFR記録も表示できます。連続JFR記録は、次のようなVMパラメータで簡単に開始できます。

```
"-XX:StartFlightRecording=maxsize=500m=filename=$TEMP/myapp.jfr,name=Continuous
recording"
```

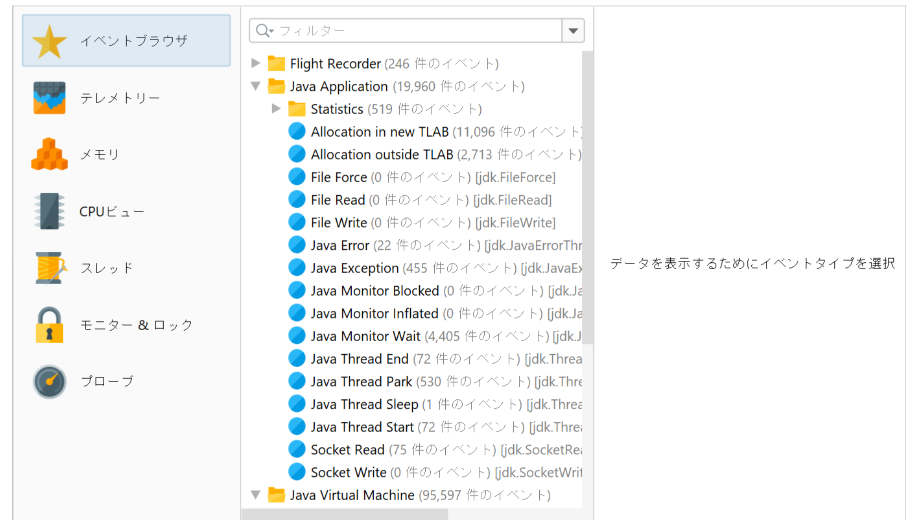
JVMテーブルの特別な背景色による指示は、JProfilerで開始されたJFR記録のみを指します。他の手段でJFR記録が開始されたJVMに接続すると、別のダイアログが表示されます。




JProfilerで新しい記録を開始するか、既存の記録をダンプして結果のJFRスナップショットをJProfilerで表示するかを選択できます。外部で開始されたJFR記録は別のライフサイクルを持ち、JProfilerによって停止されることはありません。

## E.3 JFR イベントブラウザ

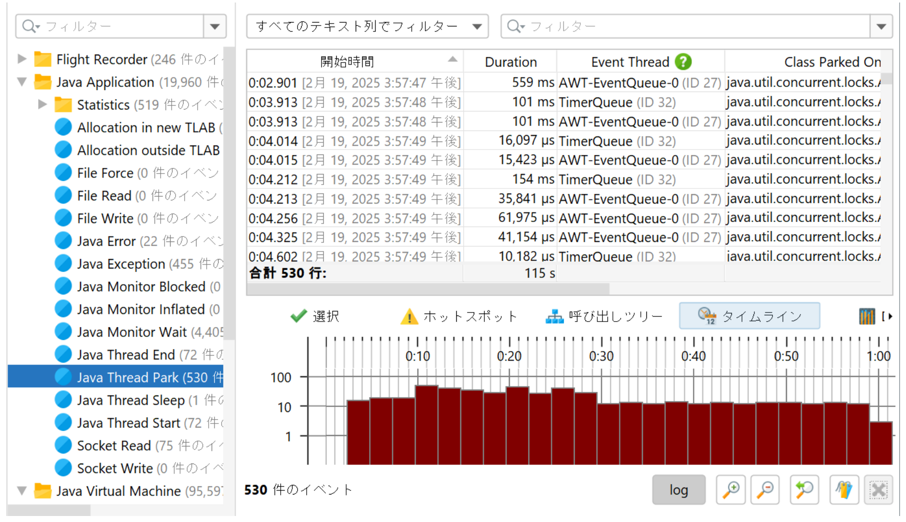
イベントブラウザは、JFR スナップショットに記録されたすべてのデータを表示します。




JFR はイベントタイプを階層的なカテゴリに整理し、イベントブラウザの左側にツリーを構成します。　単一のイベントタイプを選択して、記録されたイベントを表示できます。デフォルトでは、JProfiler はすべての登録されたイベントタイプを表示しますが、イベントが記録されていない場合でも表示されます。代わりに、ビュー設定ダイアログで空のイベントカテゴリを非表示にすることもできます。

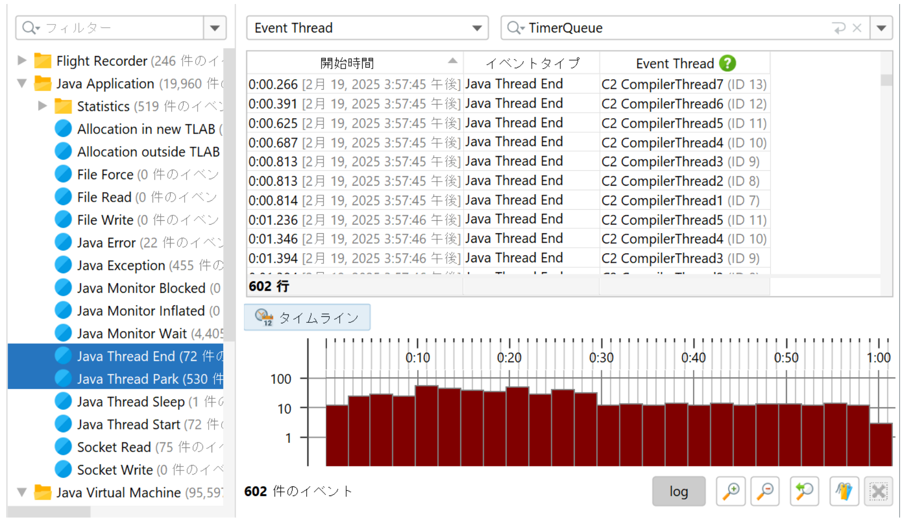
### JFR イベント

イベントはメインテーブルの行として表示され、列はイベントタイプのツリーでの選択に依存します。




テーブル内のイベントはデフォルトで時系列順にソートされます。UI　の過負荷を避けるために、テーブルには最初の10000件のイベントのみが表示されます。**下部の分析**は常にすべてのイベントから計算されます。フィルターを設定すると、最初の　10000　件だけでなく、すべてのイベントをチェックします。つまり、フィルターを設定すると、以前は表示されていなかったイベントがテーブルに表示されることがあります。

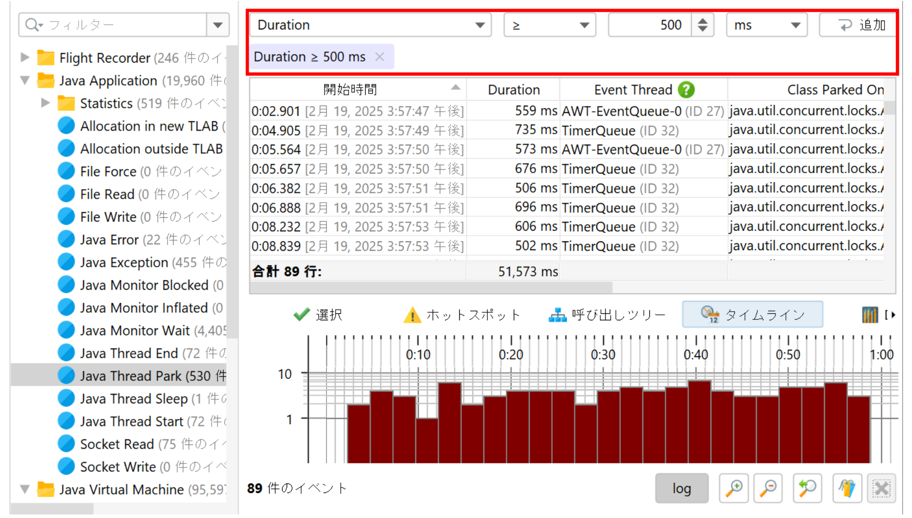
**複数のイベントタイプ**またはカテゴリ全体を選択することもできます。その場合、選択されたすべてのイベントの合計がテーブルに表示されます。各イベントタイプには独自の列セットがあるため、選択されたすべてのイベントタイプに共通する列のみが含まれます。



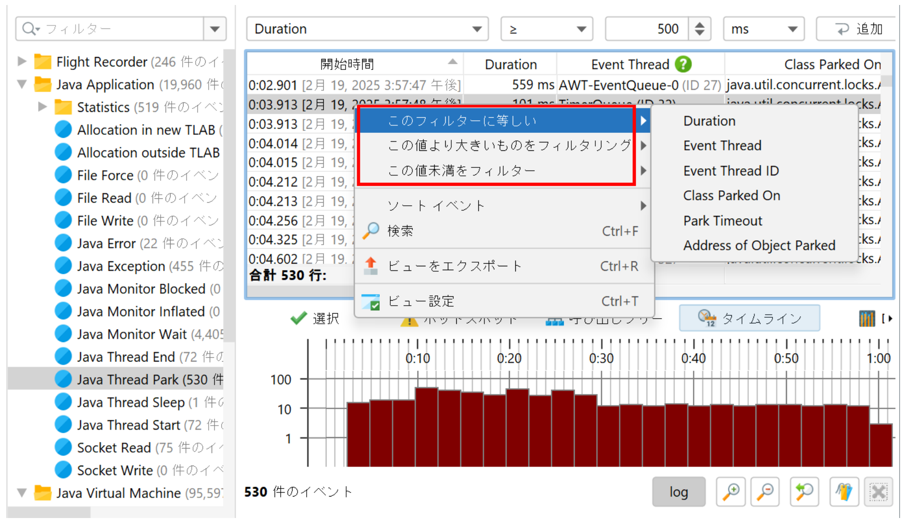

利用可能な分析の数も減少する可能性があります。これは、分析ビューが利用可能な列に基づいて追加されるためです。

**列の幅**は、実際の内容に基づいて自動的に調整されますが、列をリサイズすると、同じ内容タイプの列の幅が選択に固定され、自動的に変更されなくなります。ビュー設定ダイアログで**列の幅をクリア**するまでです。時間やメモリのような単位を持つ列のスケールも各セルごとに自動的に計算されます。列のスケールを固定して比較しやすくする場合、ビュー設定ダイアログで各列に対してオプションが提供されます。この場合、設定は選択された各イベントタイプごとに個別に保存されます。

イベントをフィルタリングする方法はいくつかあります。テーブルの上部にはフィルターセレクターがあり、すべてのテキスト列でフィルターをかけるか、単一の列を選択して列タイプに一致するフィルターを設定できます。

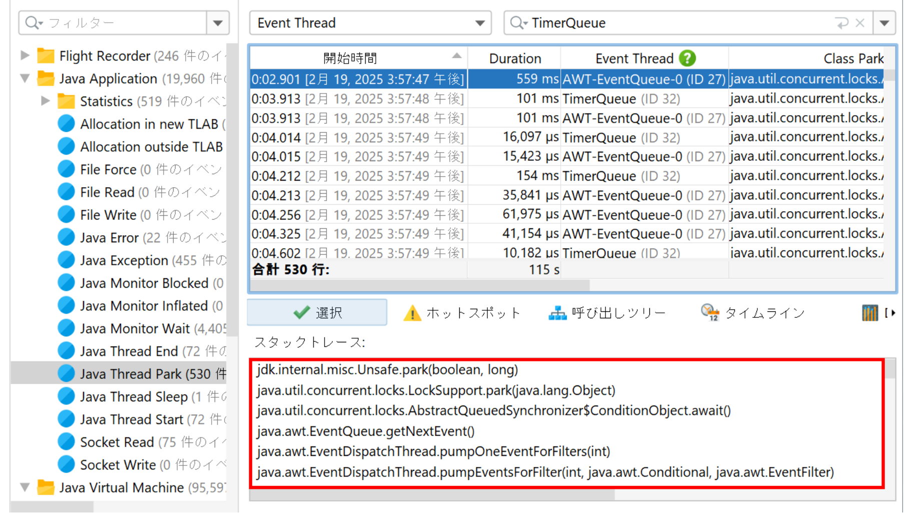



別のフィルタリング方法として、関心のある行を選択し、コンテキストメニューを使用して選択した行の値に基づいて特定のフィルターを選択することができます。上部のフィルターセレクターが調整され、選択内容が表示されます。別の値を選択してフィルターを再度追加すると、同じ列の以前のフィルターが置き換えられます。一般的に、**各フィルタータイプは一度しか存在できず**、同じフィルターを再設定すると以前のフィルターが置き換えられます。

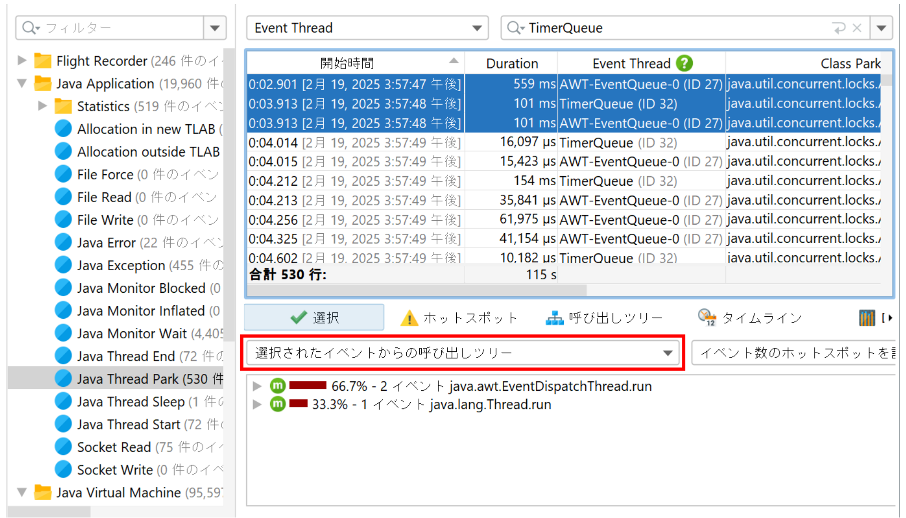



## スタックトレース

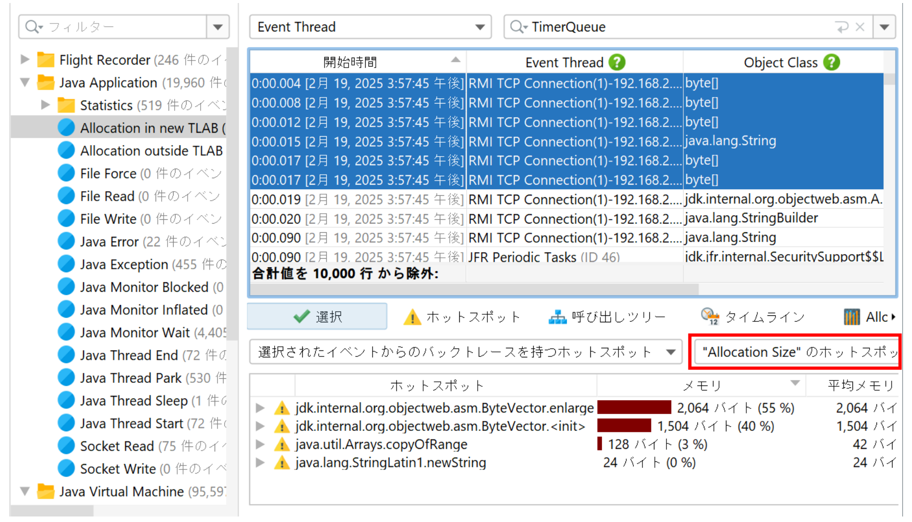
JProfiler　では、選択されたイベントのスタックトレースがイベントテーブルの下の分割ペインの「選択」タブに表示されます。



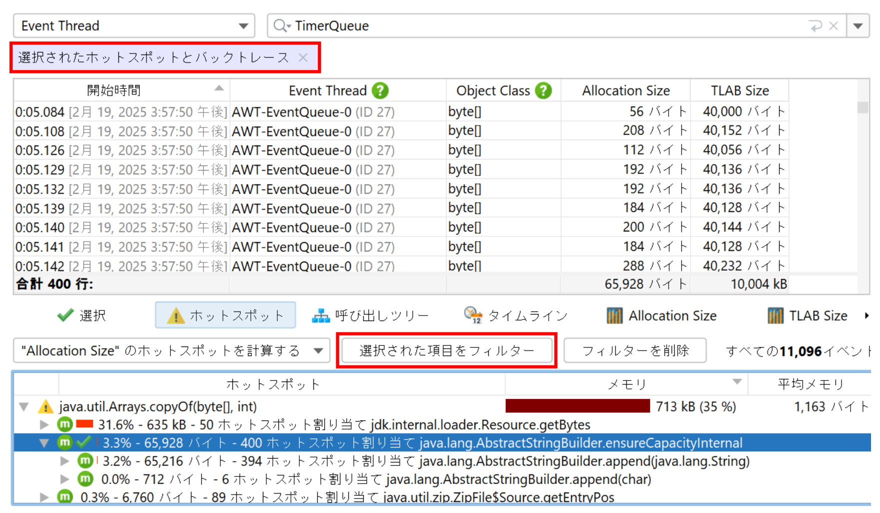

複数のイベントを選択すると、選択タブが変更され、選択されたイベントのスタックトレースから計算されたホットスポットまたは累積呼び出しツリーが表示されます。




デフォルトでは、イベント数が呼び出しツリーとホットスポットビューのノードのパーセンテージを決定します。いくつかのイベントタイプには、この目的に適した他の測定値、例えば持続時間や割り当てられたメモリが含まれています。これらの測定値が利用可能な場合、選択タブの2番目のドロップダウンから**ホットスポットタイプ**として選択できます。




下部の分割ペインにある「ホットスポット」と「呼び出しツリー」ビューには同じビューが含まれていますが、スナップショット内のすべてのイベントに対して**計算されます**。選択タブと同様に、「ホットスポットタイプ」ドロップダウンもあります。すべてのイベントを表示することに加えて、これらのビューから**フィルターを選択**することもできます。呼び出しツリービューでは、特定の呼び出しスタックを選択し、選択したものをフィルターボタンをクリックすると、上のテーブルにはその呼び出しスタックを持つイベントのみが表示されます。ホットスポットビューでは、トップレベルのホットスポットまたはバックトレース内の任意のノードを選択して、選択したノードへの逆呼び出しスタックフラグメントで終わるスタックトレースを持つイベントのみが表示されます。

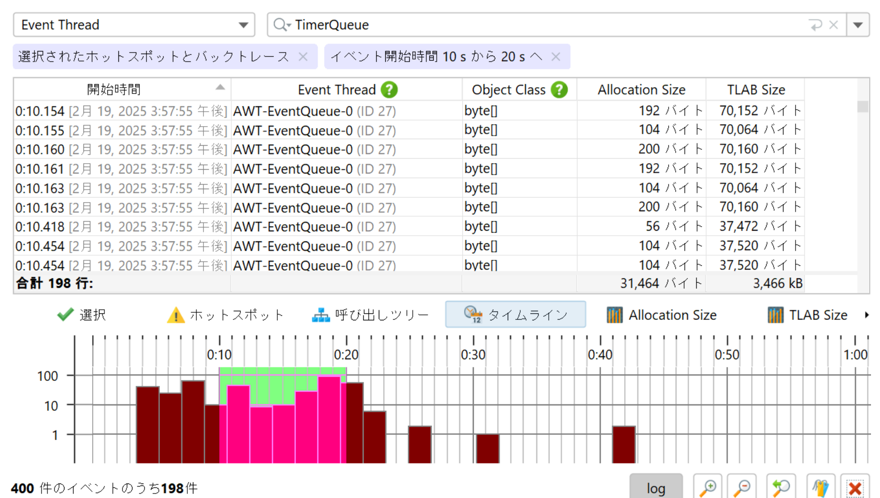



上のスクリーンショットでは、バックトレース内のノードがフィルターノードとして選択されていることがわかります。通常の呼び出しツリーアイコンに加えて、チェックマークも含まれています。フィルターは上部のタグラベルまたはフィルターを削除ボタンで削除できます。テーブル内のイベント数は選択されたノードの数と等しいです。ホットスポットツリーは、ホットスポットビューで設定されたフィルターなしですべてのイベントを表示し続けます。

これは、分析ビューから設定されたフィルターの一般的な機能です。分析ビュー自体は、すべてのフィルターされたイベントから計算されますが、**分析ビューで設定されたフィルターを除外しています**。これにより、分析ビューはより有用になります。なぜなら、選択した部分が全体のイベントセットのどの部分であるかを確認できるからです。

## タイムラインビュー

すべてのJFRイベントには関連する時間があるため、すべてのイベントタイプまたはイベントタイプのセットには、イベントの時間的な分布を示すタイムラインビューがあります。

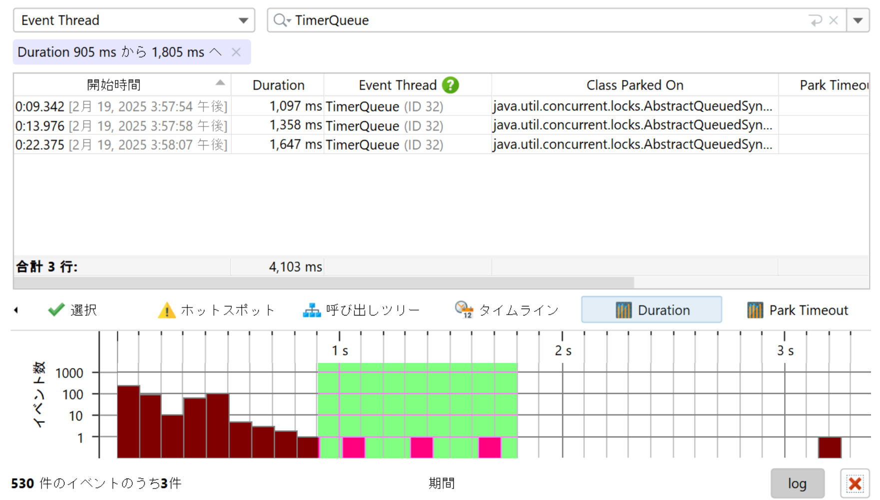



特定の時間範囲に焦点を当てるには、**時間軸に沿ってドラッグ**できます。上の例では、現在、ホットスポットのバックトレースからのフィルターとタイムラインビューからのフィルターの2つのフィルターがあります。再び、タイムラインビューは全体の時間範囲を表示し続けますが、他の分析ビューは選択された時間範囲からのイベントのみを表示します。

デフォルトの表示モードは**対数**であるため、イベント数が少ない領域でもイベント数が多い領域に対して可視性が保たれます。タイムラインの下のlogボタンを選択解除することで線形モードに切り替えることができます。デフォルトでは、全体の時間範囲が利用可能な幅で表示されますが、可変時間範囲に切り替えて、JProfilerの他のテレメトリーと同様にズームやスクロールが可能です。また、**ブックマーク**も利用可能で、選択した時間範囲に垂直マーカーを追加できます。このようにして、異なるイベントタイプ間で時間の瞬間を比較できます。

## ヒストグラムビュー

複数のイベントに対して合計できるすべての測定値、例えば持続時間や割り当てサイズは、特別な方法で処理されます。まず、イベントテーブルのこれらの測定値の列には、下部に合計値があります。次に、呼び出しツリーとホットスポット分析ビューは、イベント数の代わりにこれらの測定値でツリーを計算するための「ホットスポットタイプ」ドロップダウンを提供します。最後に、これらの測定値ごとに、下部の分割パネルにヒストグラム分析が追加されます。




ヒストグラムは、垂直軸にイベント数を表示し、水平軸には選択された測定値を表示し、分布を計算できるように複数のビンに分割されます。ビンサイズとイベント数はツールチップから利用可能です。

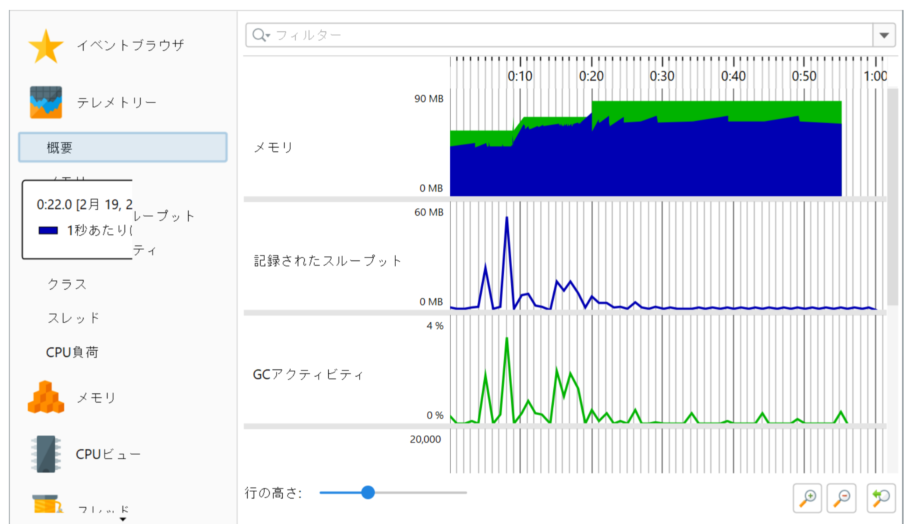
上のスクリーンショットは、ヒストグラムにフィルターが設定された様子を示しています。他の分析ビューと同様に、フィルターは他の分析ビューにのみ適用され、ヒストグラム全体は引き続き表示されます。タイムラインビューと同様に、ヒストグラムはデフォルトで対数垂直軸を持っています。ここで、スクリーンショットで選択されたイベントは、線形軸では表示されません。

## E.4 JFRスナップショットのビュー

JFRイベントブラウザ[p. 229]とは別に、JProfilerはフルプロファイリングセッションで利用可能なビューの一部を使用し、それらをJFRデータで埋めます。これは、JFRがメモリ割り当てとメソッド実行のデータを収集するため可能です。主な制限は記録レートが低いことで、問題のあるホットスポットを確認するために十分なデータを得るのに時間がかかることです。

### テレメトリー

「記録されたオブジェクトのテレメトリー」を除いて、フルプロファイリングセッションのすべてのテレメトリーは、表示されるデータにいくつかの制限があるものの、JFRスナップショットでも利用可能です。メモリテレメトリーはGC特有のプールを表示せず、スレッドテレメトリーはスレッド状態ごとのスレッド数を表示せず、記録されたスループットテレメトリーはオブジェクト数ではなくサイズを表示し、解放されるオブジェクトを表示しません。




以下の表は、さまざまなテレメトリーで使用されるイベントタイプと、それらが「デフォルト」および「プロファイル」テンプレートの両方で有効になっているかどうかを示しています。

| テレメトリー | イベントタイプ | プロファイルで有効 |
|---|---|---|
| メモリ | jdk.GCHeapSummary, jdk.MetaspaceSummary | すべて |
| 記録されたスループット | jdk.ObjectAllocationSample,<br>jdk.ObjectAllocationInNewTLAB,<br>jdk.ObjectAllocationOutsideTLAB | プロファイルのみ |
| GCアクティビティ | jdk.GarbageCollection | すべて |
| クラス | jdk.ClassLoadingStatistics | すべて |
| スレッド | jdk.JavaThreadStatistics | すべて |
| CPU負荷 | jdk.CPULoad | すべて |

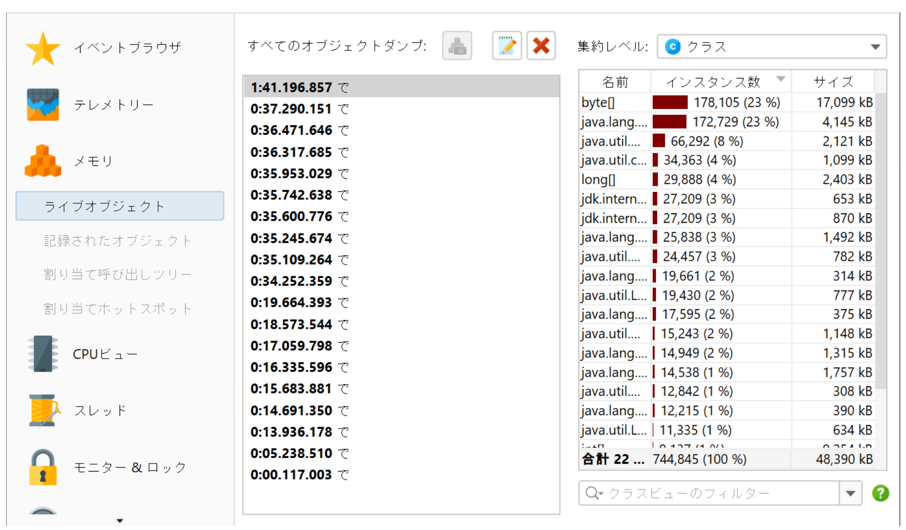
### メモリビュー

「メモリ」セクションでは、2つの異なるイベントタイプを使用してビューにデータを表示します。「ライブオブジェクト」ビューは、フルガベージコレクション後にヒープに残るすべてのクラスとインスタンス数の統計的表現を表示します。このデータは、`jdk.ObjectCount`イベントが有効になっている場合にのみ利用可能で、デフォルトのJFRテンプレートのいずれにも含まれていません。

これは、かなりのオーバーヘッドがあるためです。この設定は、JFRの高レベル設定で「ガベージコレクタ」ドロップダウンを使用して切り替えることもできます。Java 17以前では、このドロップダウンは「メモリプロファイリング」とラベル付けされています。

`jdk.ObjectCount`イベントがスナップショットで複数回記録されている場合、ビューは`jdk.ObjectCount`イベントの**最初と最後の発生の違い**を表示します。このようにして、記録時間中に数値がどのように変化したかを把握し、メモリリークの兆候を提供する可能性があります。これらの時間がスナップショット記録の開始点と終了点と一致しない場合、対応するブックマークがテレメトリービューに追加されます。通常、ヒープの1%以上の合計オブジェクトサイズを持つクラスのみが含まれます。

重大な調査には、フルプロファイリングセッション [p. 72]を使用するか、HPROFスナップショット [p. 207]を取得することを検討してください。

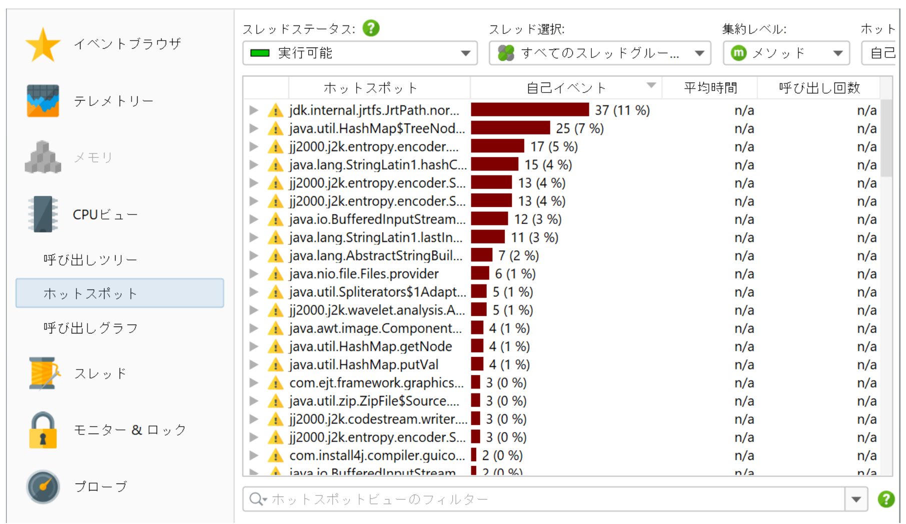



「記録されたオブジェクト」ビューおよび割り当てビューは、Java 16以降の`jdk.ObjectAllocationSample`イベントと、以前のJavaバージョンの`jdk.ObjectAllocationInNewTLAB`および`jdk.ObjectAllocationOutsideTLAB`イベントからデータを表示します。高レベルUIの「割り当てプロファイリング」ドロップダウンでもこれらのイベントタイプを有効にする方法が提供されています。

「ライブオブジェクト」ビューとは異なり、これらは記録がアクティブな間に割り当てられたオブジェクトのみを表示します。割り当てはJFRによってサンプリングされますが、サイズは**割り当てられた総サイズの推定値**として報告されます。この不一致のため、これらのビューで報告されるサイズは、サンプル数に平均インスタンスサイズを掛けたものと一致しません。それ以外の場合、これらのビューは、フルプロファイリングセッションのメモリビュー[p. 72]と同様の機能を持っています。

## CPUビュー

「CPUビュー」には、呼び出しツリー、ホットスポットビュー、および呼び出しグラフが含まれます。「Runnable」スレッド状態のデータは、標準のJFRテンプレートの両方でデフォルトで記録される`jdk.ExecutionSample`イベントに基づいています。ただし、サンプリングレートはデフォルトで20 msに設定されており、これはJFRの高レベルUIの「メソッドサンプリング」設定の「ノーマル」オプションに対応しています。JFRは非常に少数のランダムなスレッドのみをサンプリングするため、ホットスポットが十分に目立つように十分なデータを取得するには非常に長い時間がかかる可能性があります。必要に応じて、`jdk.ExecutionSample`の期間を短縮することを検討してください。これにより、JFRがデータを累積しないため、非常に大きなスナップショットサイズになる可能性があることに注意してください。




スレッドが断続的にサンプリングされるため、フルプロファイリングセッションのように実際の実行時間を推定することはできません。時間の代わりに、呼び出しツリーとホットスポットビューには**イベント数**が表示されます。これは、同じ欠点を持つ非同期サンプリング [p. 67] と似ています。他のJFRスレッド状態は「Waiting」、「Blocking」、「Socket and file I/O」であり、依然として時間を測定します。この不一致のため、スレッド状態セレクタで「すべてのスレッド状態」モードは利用できません。

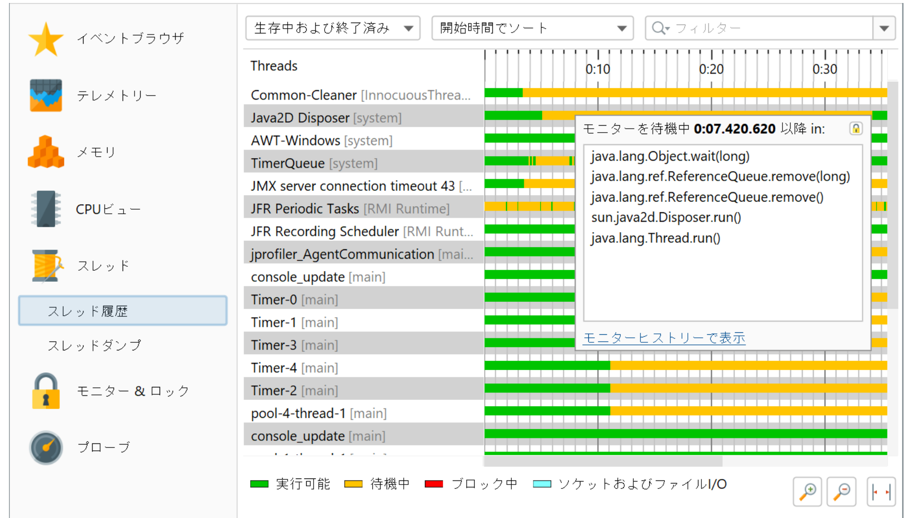
もう一つの考慮事項は、非Runnableスレッド状態が、スレッド状態セレクタのツールチップに表示される設定可能な最小期間しきい値を持つイベントから計算されることです。これらのスレッド状態の実際の合計時間は、かなり大きい可能性があります。スレッド状態を組み立てるために使用されるイベントタイプを示す表は以下に示されています：

| スレッド状態 | イベントタイプ |
|---|---|
| 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 |

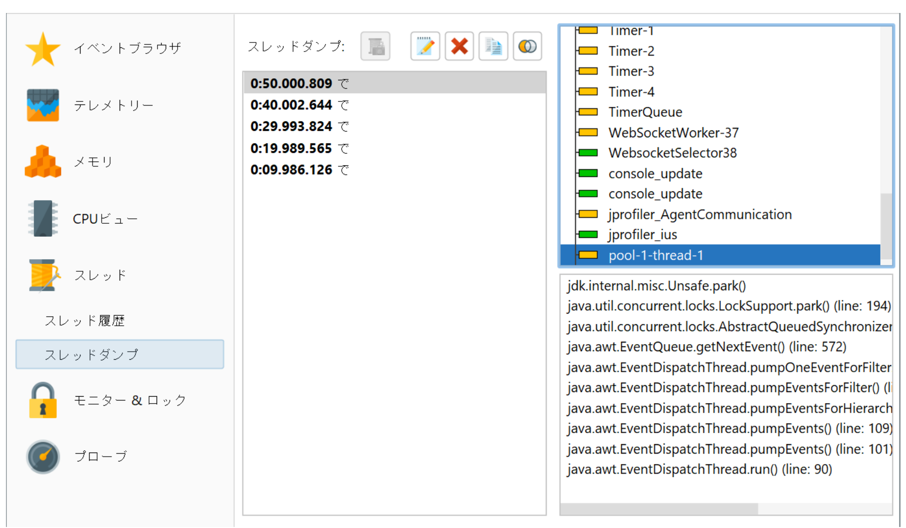
ビューの機能は、CPUビューに関するヘルプトピック [p. 54] で説明されています。フルプロファイリングセッションの多くの機能は、JFRコンテキストでは利用できないことに注意してください。

### スレッドとモニタービュー

時系列のメソッドサンプリングデータから、スレッド履歴ビューを計算でき、待機時間とブロック時間のスタックトレースを表示するツールチップも含まれます。




スレッドダンプは、JFRとJProfilerの両方で利用可能な機能であり、同じビューに表示されます。この場合、イベントブラウザは、`jdk.ThreadDump`イベントのスレッドダンプ列の構造化された内容を表示する方法がないため、代替手段にはなりません。スレッドダンプビューでは、異なるスレッドダンプを比較する[p. 98]こともできます。




`jdk.JavaMonitorWait`、`jdk.ThreadSleep`、`jdk.ThreadPark`イベントから、JProfilerは、フルプロファイリングセッション[p. 98]のものと似たモニター履歴を計算しますが、ブロッキングスレッドの情報は含まれていません。問題を解決するためにその情報が必要な場合は、フルプロファイリングセッションに切り替えてください。これにより、フルプロファイリングセッションのロッキンググラフはJFRスナップショットでは利用できないことも意味します。待機イベントに関する集計情報を示すモニター使用統計は存在し、待機時間のみを表示します。

## プローブ

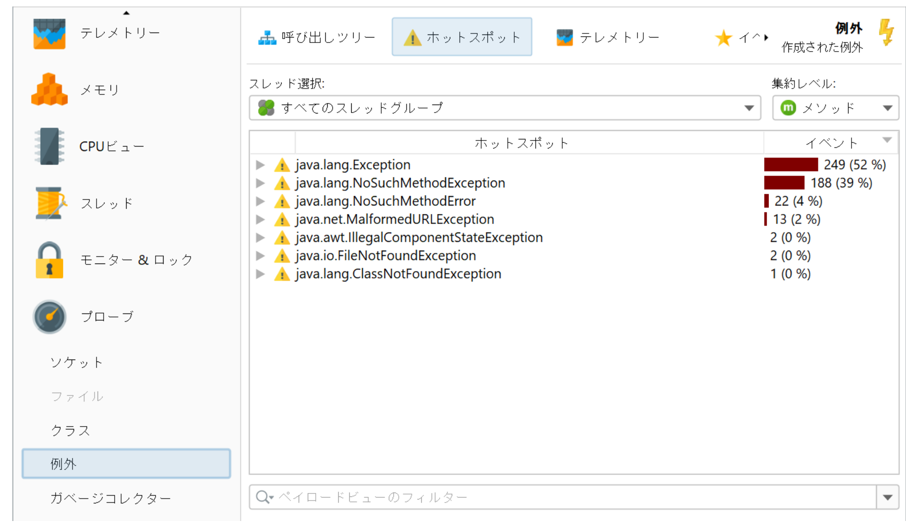
フルプロファイリングセッションの一部のJVMプローブには、JFRスナップショットで同等のデータソースがあります。それらの主な利点は、イベントブラウザと比較して、複数の関連するイベントタイプを組み合わせることです。以下の表は、データソースとして使用されるイベントタイプを持つ利用可能なプローブを示しています。

| プローブ | イベントタイプ | プロファイルで有効 |
|---|---|---|
| ソケット | jdk.SocketRead, jdk.SocketWrite | すべて |
| ファイル | jdk.FileRead, jdk.FileWrite | すべて |
| クラス | jdk.ClassLoad, jdk.ClassUnload, jdk.ClassDefine | なし |
| 例外 | jdk.JavaErrorThrow, jdk.JavaExceptionThrow | エラーは両方で、例外はなし |
| ガベージコレクタ | 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 | すべて |

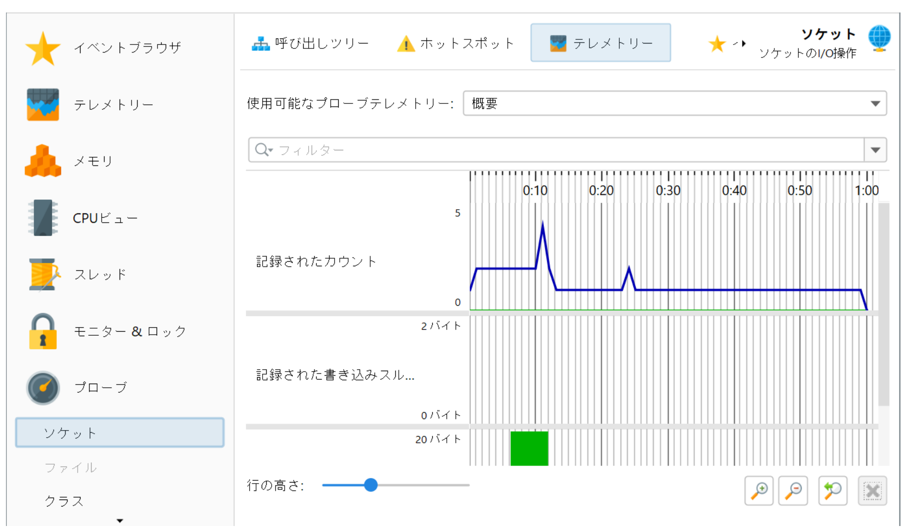
クラスローディングには、高レベルのJFR UIで3つのクラスローディングイベントをすべてオンにするチェックボックスがあります。

各プローブは、いくつかのビューを表示します。イベントブラウザとは対照的に、焦点は単一のイベントではなく、集計データにあります。これが、JProfilerのプローブがJFRデータ収集と概念的に異なる点です。

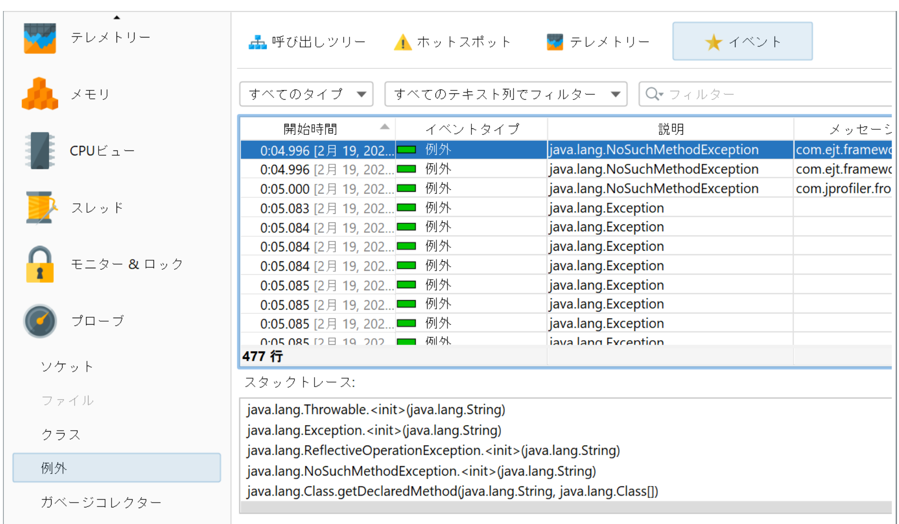
ガベージコレクタープローブを除いて、すべてのプローブには以下のビューがあります：呼び出しツリーとホットスポットビューでは、単一のスレッドまたはスレッドグループ、および集約レベルを選択できます。デフォルトでは、すべてのスレッドが表示され、集約レベルは「メソッド」に設定されています。




テレメトリービューは、記録されたデータから1つ以上のテレメトリーを表示し、すべてを一度に表示する概要ページを示します。テレメトリー名をクリックすると、完全なテレメトリーが開きます。時間軸に沿ってドラッグすることで、イベントビューで対応するイベントを選択できます。




イベントビューは、JFRブラウザのものと似ています。ただし、基になるJFRイベントに対応する複数のイベントタイプを表示し、タイプセレクタを提供します。フィルタリングと単一および複数選択のスタックトレース表示は、イベントブラウザと同様に処理されます。また、時間とメモリ測定のヒストグラムビューがあり、水平軸に沿ってドラッグすることで範囲を選択できます。




ガベージコレクタービューは特別で、Java 17以降のプロファイリングセッションで同じ情報を表示できます。JVMプローブカテゴリのガベージコレクタープローブが記録されると、必要なデータを取得するためにJFRストリーミングが使用されます。詳細については、ガベージコレクタ分析[p.119]の章を参照してください。

# F 詳細な設定

## F.1 接続問題のトラブルシューティング

プロファイリングセッションを確立できない場合、最初に行うべきことは、プロファイルされたアプリケーションまたはアプリケーションサーバーのターミナル出力を確認することです。アプリケーションサーバーの場合、stderrストリームはしばしばログファイルに書き込まれます。これは、アプリケーションサーバーのメインログファイルではなく、別のログファイルである場合があります。例えば、Websphereアプリケーションサーバーは、`native_stderr.log`ファイルにstderr出力のみを含めて書き込みます。stderr出力の内容に応じて、問題の検索は異なる方向に進みます。

#### 接続問題

stderrに`"Waiting for connection ..."`と表示されている場合、プロファイルされたアプリケーションの設定は問題ありません。問題は次の質問に関連している可能性があります：

- ローカルマシンのJProfiler GUIで"Attach to remote JVM"セッションを開始するのを忘れましたか？プロファイリングエージェントが"nowait"オプションで即座に起動するように設定されていない限り、JProfiler GUIが接続するまでVMの起動を待機します。
- セッション設定でホスト名またはIPアドレスが正しく設定されていますか？
- 通信ポートを誤って設定しましたか？通信ポートはHTTPや他の標準ポート番号とは関係なく、既に使用されているポートと同じであってはなりません。プロファイルされたアプリケーションの場合、通信ポートはプロファイリングVMパラメータのオプションとして定義されます。VMパラメータ`-agentpath:<path to jprofilerti library>=port=25000`で、25000のポートが使用されます。
- ループバックインターフェースでのみリッスンする直接接続でエージェントに接続しようとしていますか？デフォルトでは、エージェントはループバックインターフェースでのみリッスンします。JProfilerを設定してリモートマシンへのSSHトンネルを設定することができます。暗号化が必要ない場合は、`address=[IP address]`オプションを`-agentpath`パラメータに使用することもできます。
- ローカルマシンとリモートマシンの間にファイアウォールがありますか？着信接続だけでなく、発信接続や中間のゲートウェイマシン上のファイアウォールも存在する可能性があります。

#### ポートバインディングの問題

stderrにソケットをバインドできないというエラーメッセージが含まれている場合、ポートは既に使用されています。その場合、次の質問を確認してください：

- プロファイルされたアプリケーションを複数回起動しましたか？各プロファイルされたアプリケーションには、別々の通信ポートが必要です。
- 前回のプロファイリング実行のゾンビJavaプロセスがポートをブロックしていますか？
- 通信ポートを使用している別のアプリケーションがありますか？

stderrに`JProfiler>`で始まる行がなく、アプリケーションまたはアプリケーションサーバーが正常に起動する場合、`-agentpath:[path to jprofilerti library]` VMパラメータがJava呼び出しに含まれていません。実際に実行されるスタートアップスクリプト内のJava呼び出しを見つけて、そこにVMパラメータを追加する必要があります。

#### アタッチの問題

実行中のJVMにアタッチする際、すべてのJVMのリストに関心のあるJVMが表示されないことがあります。この問題の原因を見つけるには、アタッチメカニズムがどのように機能するかを理解する

ことが重要です。JVMが起動すると、`hsperfdata_$USER`ディレクトリにPIDファイルを書き込み、それによって発見されます。同じユーザーまたは管理者ユーザーのみがJVMにアタッチできます。JProfilerは、管理者ユーザーとしてJVMに接続するのを助けることができます。

Windowsでは、Show Servicesボタンを使用してすべてのJVMサービスプロセスを表示します。JProfilerは、システムアカウントで実行されるサービスに接続できるシステムアカウントで実行されるヘルパーサービスをインストールします。このサービスの名前は"JProfiler helper"であり、そのボタンをクリックするとインストールされます。サービスのインストールを許可するためにUACプロンプトを確認する必要があります。JProfilerが終了すると、サービスは再びアンインストールされます。

Linuxでは、アタッチダイアログでユーザースイッチャーを使用してrootアカウントでアタッチできます。このユーザースイッチャーは、ローカルJVMをプロファイリングする場合やリモートLinuxまたはmacOSマシンにアタッチする場合に表示されます。リモートアタッチの場合、別の非rootユーザーに切り替えることもできます。rootパスワードがある場合は、サービスを実行している実際のユーザーではなく、常にrootに切り替えてください。

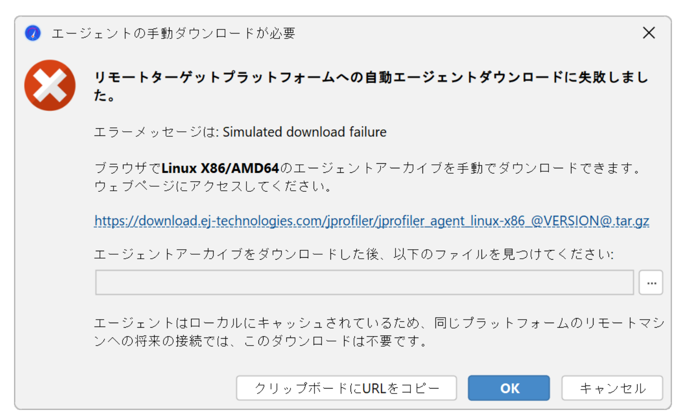
LinuxでJVMが表示されない場合、通常は一時ディレクトリに関連する問題です。`/tmp/hsperfdata_$USER`ディレクトリのアクセス権が間違っている可能性があります。その場合、ディレクトリを削除してJVMを再起動してください。アタッチされるプロセスは/tmpへの書き込みアクセス権を持っている必要があります。そうでない場合、アタッチはサポートされません。

**systemd**を使用している場合、関心のあるプロセスに`PrivateTmp=yes`がsystemdサービスファイルに設定されている可能性があります。その場合、pidファイルは異なる場所に書き込まれます。アタッチダイアログのユーザースイッチャーでrootユーザーに切り替えるか、CLIツールをrootとして使用することで、JProfilerはこれを処理します。

### リモートアタッチのための自動エージェントダウンロード

リモートアタッチの場合、JProfilerはリモートターゲットプラットフォーム用のエージェントライブラリを必要とします。ローカルに利用できない場合、ダウンロードを試みます。この操作は、`https://download.ej-technologies.com`へのHTTPS接続をブロックするファイアウォールがある場合や、トラフィックを復号化するために中間者攻撃スキームでSSL接続を検査するファイアウォールがある場合に失敗することがあります。後者の場合、JProfilerはファイアウォールの証明書を持っていないため、HTTPS接続は失敗します。

エージェントダウンロードエラーが発生した場合、JProfilerは手動の回避策を提供します。ダイアログが表示され、ウェブサイトからエージェントアーカイブを手動でダウンロードし、リモートアタッチ操作を続行する前にダウンロードしたアーカイブを見つけるための指示が示されます。




エージェントファイルはキャッシュされるため、これは各リモートプラットフォームに対して一度だけの操作です。JProfilerが更新されると、エージェントが変更され、ダウンロードを再度行う必要があります。

## F.2 JProfilerのスクリプト

JProfilerの組み込みスクリプトエディタを使用すると、カスタムプローブ設定、メソッドの分割、ヒープウォーカーフィルタなど、JProfiler GUIのさまざまな場所にカスタムロジックを入力できます。

編集エリアの上のボックスには、スクリプトの利用可能なパラメータとその戻り値の型が表示されます。メニューからヘルプ->Javadoc概要を表示を呼び出すことで、`com.jprofiler.api.*`パッケージのクラスに関する詳細情報を取得できます。

いくつかのパッケージは完全修飾クラス名を使用せずに使用できます。これらのパッケージは次のとおりです:

- java.util.*
- java.io.*

完全修飾クラス名の使用を避けるために、テキストエリアの最初の行にいくつかのimport文を配置できます。

すべてのスクリプトには、スクリプトの連続した呼び出し間で状態を保存できる`com.jprofiler.api.agent.ScriptContext`のインスタンスが渡されます。

最大のエディタ機能を得るためには、一般設定でJDKを設定することをお勧めします。デフォルトでは、JProfilerが実行されるJREが使用されます。その場合、コード補完はJREのクラスのパラメータ名とJavadocを提供しません。

### スクリプトタイプ

スクリプトは式であることができます。式には末尾のセミコロンがなく、必要な戻り値の型に評価されます。例えば、

```
object.toString().contains("test")
```

はヒープウォーカーのアウトゴーイングリファレンスビューのフィルタスクリプトとして機能します。

あるいは、スクリプトは一連のJavaステートメントで構成され、最後のステートメントとして必要な戻り値の型のreturnステートメントを持ちます:

```
import java.lang.management.ManagementFactory;
return ManagementFactory.getRuntimeMXBean().getUptime();
```

上記の例はスクリプトテレメトリーに適しています。JProfilerは、式を入力したかスクリプトを入力したかを自動的に検出します。

以前に入力したスクリプトを再利用したい場合は、スクリプト履歴から選択できます。履歴を表示ツールバーボタンをクリックすると、以前に使用したすべてのスクリプトが表示されます。スクリプトはスクリプトシグネチャによって整理され、現在のスクリプトシグネチャがデフォルトで選択されます。

### コード補完

`CTRL-Space`を押すと、コード補完の提案が表示されるポップアップが表示されます。また、ドット（"."）を入力すると、他の文字が入力されない場合に遅延後にこのポップアップが表示されます。遅延はエディタ設定で構成可能です。ポップアップが表示されている間、`Backspace`で文字を入力または削除し続けることができ、ポップアップはそれに応じて更新されます。"Camel-hump"補完がサポートされています。例えば、`NPE`を入力して`CTRL-Space`を押すと、`java.lang.NullPointerException`などのクラスが提案されます。自動的にインポートされないクラスを受け入れると、完全修飾名が挿入されます。

```
1 // This assumes that a query parameter named "action" is used
2 String action = servletRequest.getParameter("action");
3 String uri = servletRequest.getReq
4 if (action != null) {
5     return uri + "?action=" + actio
6 } else {
7     return uri;
8 }
9
```

```
getAuthType()                 String
getCharacterEncoding()        String
getContentType()              String
getContextPath()              String
getHeader(String arg0)        String
getLocalAddr()                String
getLocalName()                String
getMethod()                   String
getParameter(String arg0)     String
getPathInfo()                 String
```

自動補完ポップアップは次のことを提案できます:

- 変数とスクリプトパラメータ。スクリプトパラメータは太字で表示されます。
- import文を入力する際のパッケージ
- クラス
- コンテキストがクラスの場合のフィールド
- コンテキストがクラスまたはメソッドのパラメータリストの場合のメソッド

設定されたセッションクラスパスにも設定されたJDKにも含まれていないクラスを持つパラメータは[　　]としてマークされ、ジェネリックな`java.lang.Object`型に変更されます。そのようなパラメータにメソッドを呼び出し、コード補完を取得するために、アプリケーション設定でクラスパスに欠落しているJARファイルを追加します。

**問題分析**

入力したコードはリアルタイムで分析され、エラーと警告条件がチェックされます。エラーはエディタ内で赤い下線として表示され、右側のガターに赤いストライプとして表示されます。未使用の変数宣言などの警告は、エディタ内で黄色の背景として表示され、ガターに黄色のストライプとして表示されます。エディタ内のエラーまたは警告にマウスをホバーすると、またガターエリアのストライプにマウスをホバーすると、エラーまたは警告メッセージが表示されます。

右側のガターの上部にあるステータスインジケータは、コードに警告やエラーがない場合は緑色、警告がある場合は黄色、エラーが見つかった場合は赤色です。エディタ設定で問題分析のしきい値を構成できます。

ダイアログの右上隅にあるガターアイコンが緑色の場合、エディタ設定でエラー分析を無効にしていない限り、スクリプトはコンパイルされます。いくつかの状況では、実際のコンパイルを試みることができます。メニューからコード->テストコンパイルを選択すると、スクリプトがコンパイルされ、別のダイアログにエラーが表示されます。OKボタンでスクリプトを保存しても、スクリプトがすぐに使用されない限り、スクリプトの構文の正しさはテストされません。

### キーのバインディング

`SHIFT-F1`を押すと、カーソル位置の要素を説明するJavadocページがブラウザで開きます。Javaランタイムライブラリのjavadocは、コードエディタの一般設定で有効なJavadocの場所を持つJDKが設定されている場合にのみ表示できます。

Javaコードエディタのすべてのキーのバインディングは構成可能です。ウィンドウメニューから設定->キー マップを選択して、キー マップ エディタを表示します。キーのバインディングはファイル`$HOME/.jprofiler15/editor_keymap.xml`に保存されます。このファイルは、デフォルトのキーマップがコピーされた場合にのみ存在します。JProfilerのインストールを別のコンピュータに移行する際に、このファイルをコピーしてキーのバインディングを保持できます。

## F.3 カスタムヘルプ

ユーザーに追加のガイダンスを提供する内部ウェブサイトがある場合、ツールバーと「ヘルプ」メニューに追加のヘルプボタンを追加できます。そのためには、次のプロパティを.vmoptionsファイルに追加してください:

```
-Dcustom.help.url=https://www.internal.website.com
-Dcustom.help.toolBarText=Internal#help
-Dcustom.help.actionName=Show internal help
```

3つのプロパティすべてを定義する必要があり、そうすることでアクションがUIに表示されます。`custom.help.toolBarText` プロパティはツールバーに表示されるテキストです。簡潔であるべきで、上記の例のように#セパレータを使用して2行目を追加することができます。

.vmoptionsファイルの場所は、WindowsとLinuxでは`<JProfiler installation directory>/bin/jprofiler.vmoptions`にあり、macOSでは`/Applications/JProfiler.app/Contents/vmoptions.txt`にあります。さらに、ユーザーが書き込み可能な場所として、Windowsでは`%USERPROFILE%\.jprofiler15\jprofiler.vmoptions`、Linuxでは`$HOME/.jprofiler15/jprofiler.vmoptions`、macOSでは`$HOME/Library/Preferences/jprofiler.vmoptions`があります。

## F.4 起動時のプロファイリング設定の設定

プロファイリングエージェントが記録を開始する前に、プロファイリング設定を設定する必要があります。これは、JProfiler UI に接続するときに行われます。場合によっては、プロファイリングエージェントが起動時にプロファイリング設定を知っている必要があります。主なユースケースは次のとおりです。

- **オフラインプロファイリング**
  トリガーまたはAPIを使用してデータを記録し、スナップショットを保存します。このモードでは JProfiler GUI は接続できません。詳細については、オフラインプロファイリングに関するヘルプトピック [p. 130]を参照してください。
- **ヘッドレスマシンでの jpcontroller を使用したプロファイリング**
  コマンドラインユーティリティ jpcontroller [p. 253] を使用して、JProfiler GUI の代わりにデータを記録し、スナップショットをインタラクティブまたは非インタラクティブなコマンドファイルで保存できます。ただし、jpcontrollerにはプロファイリング設定を構成する機能がないため、事前に設定しておく必要があります。
- **古い OpenJ9 および IBM JVM へのリモートアタッチ**
  8u281、11.0.11、および Java 17 より前の古い OpenJ9 および IBM JVM では、プロファイルされたプロセスの安定性を危険にさらすことなくクラスを再定義する機能がないため、プロファイリング設定を起動時に設定する必要があります。JProfilerのリモート統合ウィザードの「Profiled JVM」ステップでは、JVM のタイプを尋ねられ、古い OpenJ9 および IBM JVM を選択すると、以下で説明するオプションがウィザードに追加されます。

一般に、起動時にプロファイリング設定を設定することは、最も効率的な操作モードです。なぜなら、クラスの再定義の数が最小限に抑えられるからです。利便性の低下が問題でない場合、あらゆる種類のプロファイリングセッションに使用できます。

### 起動時のプロファイリング設定の設定

統合ウィザードを使用する場合は、「ローカルまたはリモート」ステップで リモートコンピュータ上 オプションを選択し、「設定同期」ステップで 起動時に設定を適用 オプションを選択します。ウィザードは、以下の段落で説明するのと同じオプションを追加します。

プロファイリングエージェントをロードするために起動スクリプトに `-agentpath` VM パラメータを追加した場合、プロファイリング設定は次のように追加することで設定できます。

```
,config=<設定ファイルへのパス>,id=<セッション ID>
```

`-agentpath` パラメータの完全なパラメータは次のようになります。

```
-agentpath:/path/to/libjprofilerti.so=port=8849,nowait,config=/path/to/config,id=123
```

プロセスが開始された後にプロファイリングエージェントをロードするために `jpenable` を使用する場合、インタラクティブな実行でオフラインモードを選択し、そこで設定とIDを指定できます。あるいは、非インタラクティブな実行のために `--offline`、`--config`、`--id` 引数を渡します。

### 設定ファイルの準備

参照される設定ファイルは、現在のマシン上のJProfilerインストールの設定ファイルである可能性があり、その場合、設定パラメータを指定する必要はありません。JProfiler の設定ファイルは `$HOME/.jprofiler15/jprofiler_config.xml` または `%USERPROFILE%\.jprofiler15\`

jprofiler_config.xml にあり、-agentlib VM パラメータの config オプションのデフォルトです。

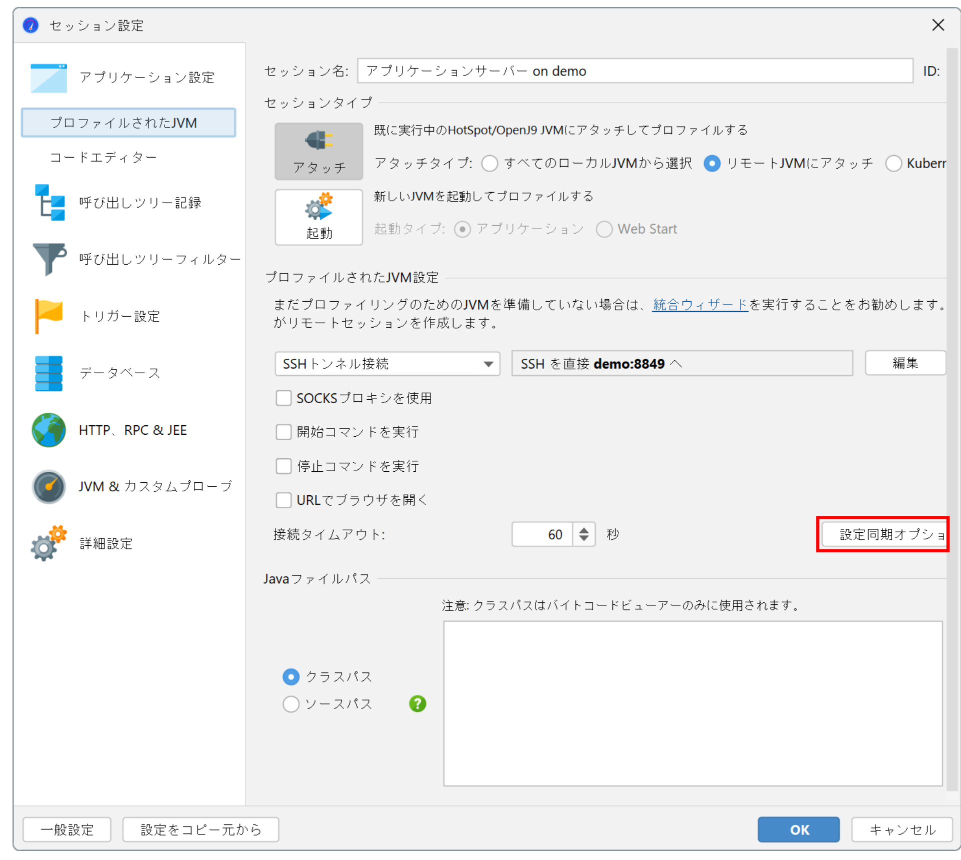
多くの場合、自動プロファイリングは別のマシンで実行され、ローカルの JProfiler 設定ファイルを参照することはできません。その場合、ローカルマシン上の JProfiler UI でプロファイリング設定を含むセッションを準備し、セッション->セッション設定のエクスポート　を介してエクスポートし、JProfiler が実行されているマシンに転送できます。

セッション ID は、セッション設定ダイアログの「アプリケーション設定」タブの右上隅に表示されます（以下のスクリーンショットを参照）。エクスポートされたファイルにセッションが 1 つしか含まれていない場合、id パラメータを指定する必要はありません。

### 設定ファイルの同期

初期設定を完了した後、将来のプロファイリング実行のためにプロファイリング設定を調整したい場合があります。これには、設定ファイルをリモートマシンにコピーする必要があります。

JProfiler のリモートセッションには、「設定同期」機能があり、このプロセスを自動化できます。




セッションが SSH 経由で開始された場合、設定ファイルを SSH 経由で直接リモートマシンにコピーできます。それ以外の場合でも、設定ファイルをローカルディレクトリにコピーし、それをリモートマシンにマウントすることができます。最後に、任意のコマンドを実行して、他の手段で設定ファイルをコピーすることができます。

設定同期オプション
以下で設定された同期アクションは、プロファイリング設定が変更されるたびに実行されます。
希望する同期アクションを選択してください:
手動同期
SSHでリモートディレクトリにコピー:
/home/build/config
ディレクトリに設定ファイルをコピー:
コマンドを実行:
OK
キャンセル

# G コマンドラインリファレンス

## G.1 プロファイリングのためのコマンドライン実行ファイル

JProfilerには、プロファイリングエージェントを設定し、コマンドラインからプロファイリングアクションを制御するためのいくつかのコマンドラインツールが含まれています。

### 実行中のJVMへのプロファイリングエージェントのロード

コマンドラインユーティリティ`bin/jpenable`を使用すると、バージョン6以上の任意の実行中のJVMにプロファイリングエージェントをロードできます。コマンドライン引数を使用して、ユーザー入力を必要としないようにプロセスを自動化できます。サポートされている引数は次のとおりです。

```
使用法: jpenable [オプション]

jpenableは選択されたローカルJVMでプロファイリングエージェントを開始し、
別のコンピュータから接続できるようにします。JProfiler GUIがローカルで実行されている場合、
この実行可能ファイルを実行する代わりにJProfiler GUIから直接アタッチできます。

* 引数が指定されていない場合、jpenableはまだプロファイルされていないローカルJVMを
  検出し、コマンドラインで必要なすべての入力を求めます。
* 以下の引数を使用することで、コマンドラインでユーザー入力を部分的または完全に提供できます:

  -d  --pid=<PID>       プロファイルされるべきJVMのPID
  -n  --noinput         いかなる状況でもユーザー入力を求めない
  -h  --help            このヘルプを表示
      --options=<OPT>   エージェントに渡されるデバッグオプション

  GUIモード: (デフォルト)
  -g  --gui             JProfiler GUIを使用してJVMにアタッチします
  -p  --port=<nnnnn>    プロファイリングエージェントがJProfiler GUIからの
                        接続を待ち受けるポート
  -a  --address=<IP>    プロファイリングエージェントが待ち受けるアドレス。このパラメータが
                        なければ、localhostからのみアタッチが可能です。すべてのアドレスで
                        待ち受けるには0.0.0.0を使用します。

  オフラインモード:
  -o  --offline         JVMはオフラインモードでプロファイルされます
  -c  --config=<PATH>   プロファイリング設定を含む設定ファイルへのパス
  -i  --id=<ID>         設定ファイル内のセッションID。設定ファイルに単一のセッションしか
                        含まれていない場合は不要です。

JVMはjpenableと同じユーザーとして実行されている必要があることに注意してください。
そうでないとJProfilerは接続できません。
例外は、ローカルシステムアカウントで実行されているWindowsサービスです。
jpenableを使用して対話的にリストする場合です。
```

### HPROFスナップショットの保存

ヒープスナップショットが必要な場合は、プロファイリングエージェントをVMにロードせずにHPROFスナップショット [p. 207]を保存する`bin/jpdump`コマンドラインツールを使用することを検討してください。

```
使用法: jpdump [オプション]

jpdumpは、ローカルで実行中のJVMからヒープをファイルにダンプします。
Hotspot VMはHPROFファイルを生成し、OpenJ9 VMはPHDファイルを生成します。
HPROFおよびPHDファイルは、その後JProfiler GUIで開くことができます。
```

```
* 引数が指定されていない場合、jpdumpはローカルで実行中のすべてのJVMを一覧表示します。
* 次の引数を使用して、コマンドラインでユーザー入力を部分的または完全に指定できます:


  -p  --pid=<PID>     ヒープをダンプするJVMのPID
                      PIDが指定されている場合、追加の質問はされません。
  -a  --all           すべてのオブジェクトを保存します。指定されていない場合は、
                      生存オブジェクトのみがダンプされます。
  -f  --file=<PATH>   ダンプファイルのパス。指定されていない場合、
                      ダンプファイルは現在のディレクトリに<VM名>.hprofとして書き込まれます。
                      ファイルが既に存在する場合、番号が追加されます。
  -h  --help          このヘルプを表示

JVMはjpdumpと同じユーザーとして実行されている必要があることに注意してください。
そうでない場合、JProfilerは接続できません。
例外は、ローカルシステムアカウントで実行されているWindowsサービスです。
これらをjpdumpで対話的にリストする場合です。
```

これは、プロファイリングエージェントをロードしてJProfilerヒープスナップショットを保存するよりもオーバーヘッドが低くなります。また、プロファイリングエージェントは決してアンロードされないため、この方法は本番環境で実行されているJVMに適しています。

### プロファイリングエージェントの制御

引数なしでbin/jpcontroller実行ファイルを起動すると、ローカルマシン上のプロファイルされたJVMへの接続を試みます。複数のプロファイルされたJVMが検出された場合、リストから選択できます。

jpcontrollerは、プロファイリング設定が設定されているJVMにのみ接続できるため、-agentpath VMパラメータの"nowait"オプションでJVMが起動された場合には機能しません。このオプションは、統合ウィザードの"Startup mode"画面でStartup immediately, connect later with the JProfiler GUIラジオボタンを選択したときに設定され、まだJProfiler GUIがエージェントに接続していない場合に設定されます。jpenableを--offline引数なしで使用する場合も、jpcontrollerがプロファイルされたプロセスに接続する前にJProfiler GUIからの接続が必要です。

リモートコンピュータ上のプロセスに接続したい場合、またはJVMがバージョン6以上のHotSpot JVMでない場合、VMパラメータ-Djprofiler.jmxServerPort=[port]をプロファイルされたJVMに渡す必要があります。そのポートでMBeanサーバーが公開され、選択したポートをjpcontrollerへの引数として指定できます。追加のVMパラメータ-Djprofiler.jmxPasswordFile=[file]を使用して、user password形式のキーと値のペアを持つプロパティファイルを指定して認証を設定できます（スペースまたはタブで区切られています）。これらのVMパラメータはcom.sun.management.jmxremote.port VMパラメータによって上書きされることに注意してください。

JMXサーバーの明示的なセットアップにより、コマンドラインコントローラーを使用してjpcontroller host:portを呼び出すことでリモートサーバーに接続できます。リモートコンピュータがIPアドレスでのみ到達可能な場合、-Djava.rmi.server.hostname=[IP address]をリモートVMへのVMパラメータとして追加する必要があります。

デフォルトでは、jpcontrollerはインタラクティブなコマンドラインユーティリティですが、手動入力を必要とせずにプロファイリングセッションを自動化することもできます。自動化された呼び出しでは、[pid | host:port]を渡してプロファイルされたJVMを選択し、--non-interactive引数を渡します。さらに、コマンドのリストが読み取られ、stdinからまたは--command-file引数で指定されたコマンドファイルから読み取られます。各コマンドは新しい行で開始され、空白行または"#"コメント文字で始まる行は無視されます。

この非インタラクティブモードのコマンドは、JProfiler MBean[1]のメソッド名と同じです。それらは同じ数のパラメータを必要とし、スペースで区切られます。文字列にスペースが含まれている場合は、ダブルクォートで囲む必要があります。さらに、`sleep <seconds>`コマンドが提供されており、指定した秒数だけ一時停止します。これにより、記録を開始し、しばらく待ってからスナップショットをディスクに保存することができます。

プロファイリング設定はプロファイリングエージェントに設定されている必要があります。これはJProfiler UIに接続するときに行われます。JProfiler UIに接続しない場合、起動コマンドで手動で設定するか、jpenableを使用して設定する必要があります。詳細については、起動時のプロファイリング設定の設定に関するヘルプトピック [p. 250]を参照してください。

jpcontrollerのサポートされている引数は以下に示されています：

```
使用法: jpcontroller [オプション] [ホスト:ポート | pid]

* 引数が指定されていない場合、jpcontrollerはプロファイルされた
  ローカルJVMを検出しようとします
* 単一の数字が指定された場合、jpcontrollerはプロセスID [pid] のJVMに接続を試みます。
  そのJVMがプロファイルされていない場合、jpcontrollerは接続できません。
  その場合は、まずjpenableユーティリティを使用してください。
* それ以外の場合、jpcontrollerは "ホスト:ポート" に接続します。ここでポートは、
  プロファイルされたJVMのVMパラメータ -Djprofiler.jmxServerPort=[port] で
  指定された値です。

使用可能なオプション:
  -n --non-interactive       コマンドのリストがstdinから読み込まれる
                             自動セッションを実行します。
  -f --command-file=<PATH>   stdinの代わりにファイルからコマンドを読み込みます。
                             --non-interactiveと一緒に使用する場合のみ適用されます。

非対話型コマンドの構文:
  RemoteControllerMBeanのjavadoc (https://bit.ly/2DimDN5) を参照して、
  操作のリストを確認してください。パラメータはスペースで区切られ、
  スペースを含む場合は引用符で囲む必要があります。例えば:

    addBookmark "Hello world"
    startCPURecording true
    startProbeRecording builtin.JdbcProbe true true
    sleep 10
    stopCPURecording
    stopProbeRecording builtin.JdbcProbe
    saveSnapshot /path/to/snapshot.jps

  sleep <秒数> コマンドは指定された秒数だけ一時停止します。
  空行と # で始まる行は無視されます。
```

[1] https://www.ej-technologies.com/resources/jprofiler/help/api/javadoc/com/jprofiler/api/agent/mbean/RemoteControllerMBean.html

## G.2 スナップショットを操作するためのコマンドライン実行ファイル

オフラインプロファイリング[p.130]を使用してスナップショットをプログラム的に保存する場合、それらのスナップショットからデータやレポートをプログラム的に抽出する必要があるかもしれません。JProfilerは、スナップショットからビューをエクスポートするためのものと、スナップショットを比較するためのものという、 2つの別々のコマンドライン実行ファイルを提供します。

### スナップショットからビューをエクスポートする

実行ファイル`bin/jpexport`は、ビューのデータをさまざまな形式にエクスポートします。`-help`オプションを使用して実行すると、利用可能なビュー名とビューオプションに関する情報が得られます。 簡潔さのために、以下の出力では重複するヘルプテキストは省略されています。

```
使用法: jpexport "スナップショットファイル" [グローバルオプション]
                 "ビュー名" [オプション] "出力ファイル"
                 "ビュー名" [オプション] "出力ファイル" ...

"snapshot file" は、次のいずれかの拡張子を持つスナップショットファイルです:
           .jps, .hprof, .hpz, .phd, .jfr
      「ビュー名」は、以下に一覧されているビュー名の1つです
      [options] は -option=value 形式のオプションのリストです
      "output file"はエクスポートの出力ファイルです

グローバルオプション:
    -obfuscator=none|proguard|yguard
        選択した難読化ツールに対してデオブスクエートします。デフォルトは「none」で、他の値を使用する
場合はmappingFileオプションを指定
        する必要があります。
    -mappingfile=<file>
        選択した難読化ツールのマッピングファイル。
    -outputdir=<出力ディレクトリ>
        出力ファイルが相対パスの場合に、ビューに使用されるベースディレクトリです。
    -ignoreerrors=true|false
        オプションを設定できないビューで発生するエラーを無視して、次のビューへ進みます。デフォルト値
は"false"です。つまり、最初のエラーが発生
        した時点でエクスポートが終了します。
    -csvseparator=<区切り文字>
        CSVエクスポートのフィールド区切り文字です。デフォルトは「,」です。
    -bitmap=false|true
       適切な場合は、メインコンテンツに対してSVGではなくビットマップ画像をエクスポートします。デフォ
ルト値はfalseです。

利用可能なビュー名とオプション:
* TelemetryHeap, TelemetryObjects, TelemetryThroughput, TelemetryGC,
  TelemetryClasses, TelemetryThreads, TelemetryCPU
  ?????:
    -format=html|csv
        エクスポートされたファイルの出力フォーマットを決定します。存在しない場合、エクスポートフォー
マットは出力ファイルの拡張子から決定されます。
    -minwidth=<ピクセル数>
        graphの最小幅（ピクセル単位）。デフォルト値は800です。
    -minheight=<ピクセル数>
        graphの最小高さ（ピクセル単位）。デフォルト値は600です。

* Bookmarks, ThreadMonitor, CurrentMonitorUsage, MonitorUsageHistory
  ?????:
    -format=html|csv

* AllObjects
  ?????:
    -format=html|csv
    -viewfilters=<カンマ区切りのリスト>
```

```
          エクスポートのためのビュー・フィルターを設定します。ビュー・フィルターを設定すると、指定されたパッケージとそのサブパッケージのみがエクスポー
          トされたビューに表示されます。
     -viewfiltermode=startswith|endswith|contains|equals
          ビューのフィルターモードを設定します。デフォルト値は「contains」です。
     -viewfilteroptions=casesensitive
          ビュー・フィルターのブールオプション。デフォルトでは、オプションは設定されていません。
     -aggregation=class|package|component
          エクスポートの集約レベルを選択します。デフォルト値はクラスです。
     -expandpackages=true|false
          パッケージ集約レベルでパッケージノードを展開して、含まれているクラスを表示します。デフォルト値は「false」です。他の集約レベルやcsv出
          力形式では効果がありません。

* RecordedObjects
  AllObjects?????????????????:
     -liveness=live|gc|all
          エクスポートの生存性モードを選択します。つまり、ライブオブジェクト、ガベージコレクションされたオブジェクト、またはその両方を表示するかどうか
          を指定します。デフォルト値はライブオブジェクトです。

* AllocationTree
  ?????:
     -format=html|xml
     -viewfilters=<カンマ区切りのリスト>
     -viewfiltermode=startswith|endswith|contains|equals
     -viewfilteroptions=casesensitive
     -aggregation=method|class|package|component
          エクスポートの集約レベルを選択します。デフォルト値はメソッドです。
     -class=<fully qualified class name>
          割り当てデータを計算するクラスを指定します。空の場合、すべてのクラスの割り当てが表示されます。package
          オプションと一緒に使用することはできません。
     -package=<fully qualified package name>
          パッケージの割り当てデータを計算するためのパッケージを指定します。空の場合、すべてのパッケージの割り当てが表示されます。パッケージ名に
          .** を追加すると、パッケージが再帰的に選択されます。class オプションと一緒に使用することはできません。
     -liveness=live|gc|all

* AllocationHotSpots
  ?????:
     -format=html|csv|xml
     -viewfilters=<カンマ区切りのリスト>
     -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
          呼び出されていないクラスを別々に表示するか、呼び出し元のクラスに追加するかを選択します。デフォルト値は、呼び出されていないクラスを別々に表示
          することです。
     -valuesummation=self|total
          ホットスポットの時間がどのように計算されるかを決定します。デフォルトは「self」です。
     -expandbacktraces=true|false
          HTMLまたはXML形式でバックトレースを展開します。デフォルト値は「false」です。

* ClassTracker
  TelemetryHeap?????????????????:
     -class
          追跡されたクラス。欠落している場合、最初に追跡されたクラスがエクスポートされます。
```

```
* CallTree
  ?????:
    -format=html|xml
    -viewfilters=<カンマ区切りのリスト>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive
    -aggregation=method|class|package|component
    -threadgroup=<スレッドグループの名前>
        エクスポートするスレッドグループを選択します。"thread"を指定した場合、スレッドはこのスレッ
ドグループ内のみで検索されます。それ以外の
        場合は、スレッドグループ全体が表示されます。
    -thread=<スレッドの名前>
        エクスポートするスレッドを選択します。デフォルトでは、呼び出しツリーはすべてのスレッドに対し
てマージされます。
    -threadstatus=all|running|waiting|blocking|netio
        エクスポートのためのスレッドステータスを選択します。デフォルト値は「running」です。

* HotSpots
  ?????:
    -format=html|csv|xml
    -viewfilters=<カンマ区切りのリスト>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive
    -aggregation=method|class|package|component
    -threadgroup=<スレッドグループの名前>
    -thread=<スレッドの名前>
    -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=<カンマ区切りのリスト>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive

* Complexity
  ?????:
    -format=html|csv|properties
    -fit=best|constant|linear|quadratic|cubic|exponential|logarithmic|n_log_n
        エクスポートするフィット。デフォルト値は「best」です。カーブフィッティングデータはCSVにエク
スポートされません。
    -method=<method name>
        複雑性グラフをエクスポートするメソッド名です。指定されていない場合、最初のメソッドがエクス
ポートされます。それ以外の場合、指定されたテキスト
        で始まる最初のメソッド名がエクスポートされます。
    -width=<ピクセル数>
    -height=<ピクセル数>

* ThreadHistory
  TelemetryHeap??????????????????????
    -format=html

* MonitorUsageStatistics
  ?????:
    -format=html|csv
    -type=monitors|threads|classes
      エンティティを選択して、そのモニタースタティスティクスを計算します。デフォルト値は「monitors」
です。

* ProbeTimeLine
```

```
  ThreadHistory?????????????????:
    -probeid=<id>
        プローブの内部IDをエクスポートする必要があります。すべての利用可能な組み込みプローブを一覧表示し、カスタムプローブ名の説明を得るには「jp
        export --listProbes」を実行してください。

* ProbeControlObjects
  ?????:
    -probeid=<id>
    -format=html|csv

* ProbeCallTree
  ?????:
    -probeid=<id>
    -format=html|xml
    -viewfilters=<カンマ区切りのリスト>
    -viewfiltermode=startswith|endswith|contains|equals|wildcard|regex
    -viewfilteroptions=exclude,casesensitive
    -aggregation=method|class|package|component
    -threadgroup=<スレッドグループの名前>
    -thread=<スレッドの名前>
    -threadstatus=all|running|waiting|blocking|netio
        エクスポートのためのスレッドステータスを選択します。デフォルト値は「all」です。

* ProbeHotSpots
  ProbeCallTree??????????????????????:
    -format=html|csv|xml
    -expandbacktraces=true|false

* ProbeTelemetry
  TelemetryHeap?????????????????:
    -probeid=<id>
    -telemetrygroup
        テレメトリーグループの1から始まるインデックスを設定します。これは、プローブテレメトリービューの上にあるドロップダウンリストで表示されるエン
        トリを指します。デフォルト値は「1」です。

* ProbeEvents
  ?????:
    -probeid=<id>
    -format=html|csv|xml

* ProbeTracker
  TelemetryHeap?????????????????:
    -probeid=<id>
    -index=<number>
        ドロップダウンリスト内のトラッキングされた要素を含む、ゼロベースのインデックスを設定します。デフォルト値は0です。
```

### スナップショットを比較する

実行ファイル`bin/jpcompare`は、異なるスナップショットを比較[p.135]し、それらをHTMLまたは機械可読形式にエクスポートします。`-help` の出力は以下に示されており、 こちらも重複する説明は省略されています。

```
使用法: jpcompare "スナップショットファイル"[,"スナップショットファイル",...] [グローバルオプション]
              "比較名" [オプション] "出力ファイル"
              "比較名" [オプション] "出力ファイル" ...

"snapshot file" は、次のいずれかの拡張子を持つスナップショットファイルです:
```

```
            .jps, .hprof, .hpz, .phd, .jfr
        「比較名」は、以下に一覧されている比較名の1つです
        [options] は -option=value 形式のオプションのリストです
        "output file"はエクスポートの出力ファイルです

グローバルオプション:
    -outputdir=<出力ディレクトリ>
        出力ファイルが相対パスの場合に、比較に使用されるベースディレクトリです。
    -ignoreerrors=true|false
        オプションを設定できない比較で発生するエラーを無視して、次の比較へ進みます。デフォルト値
は"false"です。つまり、最初のエラーが発生した
        時点でエクスポートが終了します。
    -csvseparator=<区切り文字>
        CSVエクスポートのフィールド区切り文字です。デフォルトは「,」です。
    -bitmap=false|true
       適切な場合は、メインコンテンツに対してSVGではなくビットマップ画像をエクスポートします。デフォ
ルト値はfalseです。
    -sortbytime=false|true
        指定されたスナップショットファイルを変更時間でソートします。デフォルト値はfalseです。
    -listfile=<filename>
        スナップショットファイルのパスを含むファイルを読み込みます。1行につき1つのスナップショット
ファイルです。

利用可能な比較名とオプション:
* Objects
  ?????:
    -format=html|csv
        エクスポートされたファイルの出力フォーマットを決定します。存在しない場合、エクスポートフォー
マットは出力ファイルの拡張子から決定されます。
    -viewfilters=<カンマ区切りのリスト>
        エクスポートのためのビュー・フィルターを設定します。ビュー・フィルターを設定すると、指定され
たパッケージとそのサブパッケージのみがエクスポー
        トされたビューに表示されます。
    -viewfiltermode=startswith|endswith|contains|equals
        ビューのフィルターモードを設定します。デフォルト値は「contains」です。
    -viewfilteroptions=casesensitive
        ビュー・フィルターのブールオプション。デフォルトでは、オプションは設定されていません。
    -aggregation=class|package|component
        エクスポートの集約レベルを選択します。デフォルト値はクラスです。
    -liveness=live|gc|all
        エクスポートの生存性モードを選択します。つまり、ライブオブジェクト、ガベージコレクションされ
たオブジェクト、またはその両方を表示するかどうか
        を指定します。デフォルト値はライブオブジェクトです。
    -objects=recorded|heapwalker|all
        記録されたオブジェクト、ヒープウォーカー内のオブジェクト、またはすべてのオブジェクトダンプか
らのオブジェクト数を比較します。デフォルトは
        .jps ファイルの場合は記録されたオブジェクトで、HPROF/PHD ファイルの場合はヒープウォーカー
です。
    -dumpselection=first|last|label
        比較計算に使用されるすべてのオブジェクトダンプ。デフォルトは最後の値です。
    -label
        dumpselection が 'label' に設定されている場合、比較を計算するためのラベルの名前です。

* AllocationHotSpots
  ?????:
    -format=html|csv
    -viewfilters=<カンマ区切りのリスト>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive
    -aggregation=method|class|package|component
        エクスポートの集約レベルを選択します。デフォルト値はメソッドです。
    -liveness=live|gc|all
    -unprofiledclasses=separately|addtocalling
        呼び出されていないクラスを別々に表示するか、呼び出し元のクラスに追加するかを選択します。デ
```

```
フォルト値は、呼び出されていないクラスを別々に表示
        することです。
    -valuesummation=self|total
        ホットスポットの時間がどのように計算されるかを決定します。デフォルトは「self」です。
    -classselection
        特定のクラスまたはパッケージの比較を計算します。パッケージは、単一のパッケージには「.*」、再
帰的なパッケージには「.**」を追加して指定し
        ます。 パッケージをワイルドカード付きで指定してください。例: 'java.awt.*'。

* AllocationTree
  ?????:
    -format=html|xml
    -viewfilters=<カンマ区切りのリスト>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive
    -aggregation=method|class|package|component
    -liveness=live|gc|all
    -classselection

* HotSpots
  ?????:
    -format=html|csv
    -viewfilters=<カンマ区切りのリスト>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive
    -firstthreadselection
        特定のスレッドまたはスレッドグループの比較を計算します。 スレッドグループを 'group.*'
        のように指定し、特定のスレッドグループ内のスレッドを 'group.thread'
        のように指定します。スレッド名のドットはバックスラッシュでエスケープしてください。
    -secondthreadselection
       特定のスレッドまたはスレッドグループの比較を計算します。 メッセージは 'firstthreadselection'

        が設定されている場合にのみ利用可能です。空の場合、'firstthreadselection' と同じ値が使用
されます。 スレッドグループを
        'group.*' のように指定し、特定のスレッドグループ内のスレッドを 'group.thread'
        のように指定します。スレッド名のドットはバックスラッシュでエスケープしてください。
    -threadstatus=all|running|waiting|blocking|netio
        エクスポートのためのスレッドステータスを選択します。デフォルト値は「running」です。
    -aggregation=method|class|package|component
    -differencecalculation=total|average
        呼び出し回数の差分計算方法を選択します。デフォルト値は総回数です。
    -unprofiledclasses=separately|addtocalling
    -valuesummation=self|total

* CallTree
  ?????:
    -format=html|xml
    -viewfilters=<カンマ区切りのリスト>
    -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=<ピクセル数>
        graphの最小幅（ピクセル単位）。デフォルト値は800です。
    -minheight=<ピクセル数>
        graphの最小高さ（ピクセル単位）。デフォルト値は600です。
    -valuetype=current|maximum|bookmark
```

```
          各スナップショットに対して計算される値のタイプ。デフォルトは現在の値です。
     -bookmarkname
          valuetype が 'bookmark' に設定されている場合、値を計算するためのブックマークの名前です。

     -measurements=maximum,free,used
          比較グラフに表示される測定値。複数の値はカンマで連結します。デフォルト値は「used」です。
     -memorytype=heap|nonheap
          メモリの種類を分析する必要があります。デフォルトは「heap」です。
     -memorypool
          特別なメモリプールを分析する場合は、このパラメータでその名前を指定できます。デフォルトは空
で、つまり特別なメモリプールはありません。

* TelemetryObjects
  ?????:
     -format=html|csv
     -minwidth=<ピクセル数>
     -minheight=<ピクセル数>
     -valuetype=current|maximum|bookmark
     -bookmarkname
     -measurements=total,nonarrays,arrays
          比較グラフに表示される測定値。複数の値はカンマで連結します。デフォルト値は「total」です。

* TelemetryClasses
  TelemetryObjects??????????????????????
     -measurements=total,filtered,unfiltered

* TelemetryThreads
  TelemetryObjects??????????????????????
     -measurements=total,runnable,blocked,netio,waiting

* ProbeHotSpots
  ?????:
     -format=html|csv
     -viewfilters=<カンマ区切りのリスト>
     -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>
          プローブの内部IDをエクスポートする必要があります。すべての利用可能な組み込みプローブを一覧表
示し、カスタムプローブ名の説明を得るには「jp
          export --listProbes」を実行してください。

* ProbeCallTree
  ProbeHotSpots???????????????????????
     -format=html|xml

* ProbeTelemetry
  TelemetryObjects???????????????????????:
     -measurements
          計測値を示すテレメトリーグループ内の1ベースのインデックスです。複数の値をカンマで連結します
（例: "1,
          2"）。デフォルト値はすべての計測値を表示します。
     -probeid=<id>
     -telemetrygroup
          テレメトリーグループの1から始まるインデックスを設定します。これは、プローブテレメトリービュー
の上にあるドロップダウンリストで表示されるエン
          トリを指します。デフォルト値は「1」です。
```

### 自動出力形式

ほとんどのビューと比較は、複数の出力形式をサポートしています。デフォルトでは、出力形式は出力ファイルの拡張子から推測されます:

- **.html**

  ビューまたは比較はHTMLファイルとしてエクスポートされます。HTMLページで使用される画像を含む ディレクトリ `jprofiler_images` が作成されます。

- **.csv**

  データはCSVデータとしてエクスポートされ、最初の行には列名が含まれます。

  Microsoft Excelを使用する場合、CSVは安定した形式ではありません。Windows上のMicrosoft Excelは 地域設定からセパレータ文字を取得します。JProfilerは、小数点セパレータとしてカンマを使用するロケールではセミコロンをセパレータとして使用し、小数点セパレータとしてドットを使用するロケールではカンマを使用します。CSVセパレータ文字を上書きする必要がある場合は、グローバルオプション `csvseparator` を設定することで 可能です。

- **.xml**

  データはXMLとしてエクスポートされます。データ形式は自己記述的です。

異なる拡張子を使用したい場合は、`format` オプションを使用して 出力形式の選択を上書きすることができます。

### スナップショットの分析

生成されたスナップショットにヒープダンプが含まれている場合は、`bin/jpanalyze`実行ファイルを使用して、ヒープダンプ分析を事前に準備 [p. 81]することができます。 その後、JProfilerのGUIでスナップショットを開くのが非常に速くなります。ツールの使用情報は以下に示されています:

```
使用法: jpanalyze [オプション] "スナップショットファイル" ["スナップショットファイル" ...]

"snapshot file" は、次のいずれかの拡張子を持つスナップショットファイルです:
           .jps, .hprof, .hpz, .phd, .jfr
     [options] は -option=value 形式のオプションのリストです

オプション:
    -obfuscator=none|proguard|yguard
       選択した難読化ツールに対してデオブスクエートします。デフォルトは「none」で、他の値を使用する場合はmappingFileオプションを指定
        する必要があります。
    -mappingfile=<file>
        選択した難読化ツールのマッピングファイル。
    -removeunreferenced=true|false
        参照されていないオブジェクトまたは弱参照されているオブジェクトを削除する必要がある場合。
    -retained=true|false
        保持サイズを計算します（最大のオブジェクト）。removeunreferencedがtrueに設定されます。
    -retainsoft=true|false
        参照されていないオブジェクトが削除された場合、ソフト参照を保持するかどうかを指定します。
    -retainweak=true|false
        参照されていないオブジェクトが削除された場合、弱い参照を保持するかどうかを指定します。
    -retainphantom=true|false
        参照されていないオブジェクトが削除された場合、ファントム参照を保持するかどうかを指定します。

    -retainfinalizer=true|false
        参照されていないオブジェクトが削除された場合、ファイナライザの参照を保持するかどうかを指定します。
```

removeUnreferenced、retained、およびすべてのretain*　コマンドラインオプションは、ヒープウォーカーオプションダイアログのオプションに対応しています。

## G.3 Gradleタスク

JProfilerは、特別なタスクを使用してGradleからのプロファイリングをサポートします。さらに、JProfilerは スナップショットを操作するためのコマンドライン実行ファイル [p. 256] に対応するGradleタスクを提供します。

### Gradleタスクの使用

JProfiler GradleタスクをGradleビルドファイルで利用可能にするには、`plugins` ブロックを使用します。

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

この目的のためにGradleプラグインリポジトリを使用したくない場合、Gradleプラグインは `bin/gradle.jar`ファイルに配布されています。

次に、JProfiler GradleプラグインにJProfilerがインストールされている場所を伝える必要があります。

```
jprofiler {
    installDir = file('/path/to/jprofiler/home')
}
```

### Gradleからのプロファイリング

`com.jprofiler.gradle.JavaProfile`タイプのタスクを使用して、任意のJavaプロセスをプロファイルできます。 このクラスはGradleの組み込みクラス`JavaExec`を拡張しているため、プロセスを構成するために同じ引数を使用できます。 テストをプロファイルするには、Gradle `Test`タスクを拡張する`com.jprofiler.gradle.TestProfile`タイプのタスクを使用します。

追加の構成なしで、両方のタスクはインタラクティブなプロファイリングセッションを開始し、プロファイリングエージェントはデフォルトポート8849でJProfiler GUIからの接続を待ちます。オフラインプロファイリングの場合、以下の表に示すいくつかの属性を追加する必要があります。

<table>
<tr><th>属性</th><th>説明</th><th>必須</th></tr>
<tr><td>offline</td><td>プロファイリング実行がオフラインモードであるべきかどうか。</td><td rowspan="2">いいえ、<code>offline</code>と<code>nowait</code>の両方を<code>true</code>にすることはできません。</td></tr>
<tr><td>nowait</td><td>プロファイリングをすぐに開始するか、プロファイルされたJVMがJProfiler GUIからの接続を待つべきかどうか。</td></tr>
<tr><td>sessionId</td><td>プロファイリング設定を取得するセッションIDを定義します。<code>nowait</code>もofflineも設定されていない場合、 プロファイリングセッションはGUIで選択されるため、効果はありません。</td><td>必須<ul><li><code>offline</code>が設定されている場合</li><li>1.5 JVMで<code>nowait</code>が設定されている場合</li></ul></td></tr>
</table>

| 属性 | 説明 | 必須 |
|---|---|---|
| configFile | プロファイリング設定を読み込む構成ファイルを定義します。 | いいえ |
| port | プロファイリングエージェントがJProfiler GUIからの接続を待つポート番号を定義します。これはリモートセッション構成で設定されたポートと同じでなければなりません。設定されていないかゼロの場合、デフォルトポート（8849）が使用されます。offlineが設定されている場合、GUIからの接続がないため効果はありません。 | いいえ |
| debugOptions | チューニングやデバッグの目的で追加のライブラリパラメータを渡したい場合、この属性を使用できます。 | いいえ |

メインメソッドを持つJavaクラスをプロファイルする例を以下に示します。このクラスは含まれるプロジェクトによってコンパイルされます。

```
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')
}
```

このタスクの実行可能な例はapi/samples/offlineサンプルプロジェクトで見ることができます。 標準のJavaExecタスクとは異なり、JavaProfileタスクはcreateProcess()を呼び出すことで バックグラウンドで開始することもできます。この機能のデモンストレーションはapi/samples/mbeanサンプルプロジェクトで確認できます。

プロファイリングに必要なVMパラメータが必要な場合、com.jprofiler.gradle.SetAgentpathPropertyタスクはpropertyName属性で設定されたプロパティにそれを割り当てます。JProfilerプラグインを適用すると、自動的にこのタイプのタスクsetAgentPathPropertyがプロジェクトに追加されます。 前述の例で使用されるVMパラメータを取得するには、単に以下を追加します。

```
setAgentPathProperty {
    propertyName = 'profilingVmParameter'
    offline = true
    sessionId = 80
    configFile = file('path/to/jprofiler_config.xml')
}
```

プロジェクトに追加し、setAgentPathPropertyへの依存関係を他のタスクに追加します。そのタスクの実行フェーズでプロジェクトプロパティprofilingVmParameterを使用できます。他のタスクプロパティにプロパティを割り当てる際は、doFirst {...}コードブロックでその使用を囲み、 Gradleの実行フェーズにいることを確認してください。構成フェーズではありません。

### スナップショットからのデータのエクスポート

com.jprofiler.gradle.Exportタスクは、保存されたスナップショットからビューをエクスポートするために使用でき、bin/jpexport コマンドラインツール [p.256]の引数を複製します。次の属性をサポートしています：

| 属性 | 説明 | 必須 |
| --- | --- | --- |
| snapshotFile | スナップショットファイルへのパス。これは.jps拡張子を持つファイルでなければなりません。 | はい |
| ignoreErrors | ビューのオプションを設定できない場合に発生するエラーを無視し、次のビューに進みます。デフォルト値はfalseで、最初のエラーが発生したときにエクスポートが終了します。 | いいえ |
| csvSeparator | CSVエクスポートのフィールドセパレータ文字。デフォルトは","です。 | いいえ |
| obfuscator | 選択されたオブファスケータのためにクラスとメソッド名をデオブファスケートします。デフォルトは"none"で、 他の値の場合はmappingFileオプションを指定する必要があります。 none、proguard、yguardのいずれか。 | いいえ |
| mappingFile | 選択されたオブファスケータのためのマッピングファイル。obfuscator属性が指定されている場合にのみ設定できます。 | obfuscatorが指定されている場合のみ |

エクスポートタスクでは、viewsメソッドを呼び出し、その中でview(name, file[, options])を一度または複数回呼び出すクロージャを渡します。viewの各呼び出しは1つの出力ファイルを生成します。 name引数はビュー名です。利用可能なビュー名のリストについては、 jpexport コマンドライン実行ファイル[p. 256]のヘルプページを参照してください。引数fileは出力ファイルで、絶対ファイルまたはプロジェクトに対する相対ファイルです。 最後に、オプションのoptions引数は、選択されたビューのエクスポートオプションを持つマップです。

エクスポートタスクを使用する例は以下の通りです：

```
task export(type: com.jprofiler.gradle.Export) {
    snapshotFile = file('snapshot.jps')
    views {
        view('CallTree', 'callTree.html')
        view('HotSpots', 'hotSpots.html',
            [threadStatus: 'all', expandBacktraces: 'true'])
    }
}
```

### スナップショットの比較

bin/jpcompare コマンドラインツール[p. 256]のように、com.jprofiler.gradle.Compareタスクは2つ以上のスナップショットを比較できます。 その属性は次の通りです：

| 属性 | 説明 | 必須 |
| --- | --- | --- |
| snapshotFiles | 比較するスナップショットファイル。Gradleがファイルコレクションに解決するオブジェクトを含む Iterableを渡すことができます。 | はい |
| sortByTime | trueに設定すると、提供されたすべてのスナップショットファイルがファイルの変更時間でソートされます。 そうでない場合は、snapshotFiles属性で指定された順序で比較されます。 | いいえ |

| 属性 | 説明 | 必須 |
| --- | --- | --- |
| ignoreErrors | 比較のオプションを設定できない場合に発生するエラーを無視し、次の比較に進みます。 デフォルト値はfalseで、最初のエラーが発生したときにエクスポートが終了します。 | いいえ |

エクスポートされたビューがExportタスクで定義されているのと同様に、Compareタスクにはcomparisonsメソッドがあり、comparison(name, file[, options])のネストされた呼び出しで 実行する比較を定義します。利用可能な比較名のリストは、jpcompare コマンドライン実行ファイル [p. 256]のヘルプページで確認できます。

比較タスクを使用する例は以下の通りです：

```
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'])
    }
}
```

または、複数のスナップショットに対してテレメトリー比較を作成したい場合：

```
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'])
    }
}
```

### ヒープスナップショットの分析

Gradleタスクcom.jprofiler.gradle.Analyzeは、 bin/jpanalyze コマンドラインツール [p. 256]と同じ機能を持っています。

このタスクには、Compareタスクのように処理されるスナップショットを指定するsnapshotFiles属性と、デオブファスケーションのためのExportタスクのようなobfuscatorおよびmappingfile属性があります。属性removeUnreferenced、retainSoft、retainWeak、retainPhantom、retainFinalizer、retainedはコマンドラインツールの引数に対応しています。

Analyzeタスクを使用する例は以下の通りです：

```
task analyze(type: com.jprofiler.gradle.Analyze) {
    snapshotFiles = fileTree(dir: 'snapshots', include: '*.jps')
    retainWeak = true
    obfuscator = 'proguard'
    mappingFile = file('obfuscation.txt')
}
```

## G.4 Ant タスク

JProfiler によって提供される Ant [1] タスクは、Gradle タスクと非常に似ています。この章では、Gradle タスクとの違いを強調し、各 Ant タスクの例を示します。

すべての Ant タスクはアーカイブ `bin/ant.jar` に含まれています。 Ant にタスクを利用可能にするには、まず `taskdef` 要素を挿入して、Ant にタスク定義の場所を知らせる必要があります。以下のすべての例にはそのtaskdefが含まれています。それはビルドファイルごとに一度だけ発生し、プロジェクト要素の下のレベルのどこにでも現れることができます。

`ant.jar` アーカイブを Ant ディストリビューションの `lib` フォルダにコピーすることはできません。タスク定義で JProfiler の完全なインストールを参照する必要があります。

### Ant からのプロファイリング

`com.jprofiler.ant.ProfileTask` は組み込みの `Java` タスクから派生しており、そのすべての属性とネストされた要素をサポートしています。追加の属性は`ProfileJava`Gradleタスク [p.265] と同じです。 Ant の属性は大文字と小文字を区別せず、通常は小文字で書かれます。

```
<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>
```

### スナップショットからのデータのエクスポート

`com.jprofiler.ant.ExportTask` を使用すると、`Export` Gradle タスク [p. 265] と同様に、スナップショットからビューをエクスポートできます。ビューは Gradle タスクとは異なり、タスク要素の直下にネストされ、オプションはネストされた `option` 要素で指定されます。

```
<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>
```

### スナップショットの比較

`com.jprofiler.ant.CompareTask` は `Compare` Gradle タスクに対応しており、2 つ以上のスナップショット間で比較を行います。`com.jprofiler.ant.ExportTask` と同様に、比較は要素の直下にネストされ、オプションは各`comparison`要素にネストされます。スナップショットファイルはネストされたファイルセットで指定されます。

[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>
```

### ヒープスナップショットの分析

`Analyze` Gradleタスクと同様に、Ant用の`com.jprofiler.ant.AnalyzeTask`は、オフラインプロファイリングで保存されたスナップショットのヒープスナップショット分析をGUIでの高速アクセスのために準備します。処理されるべきスナップショットはネストされたファイルセットで指定されます。

```
<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>
```