ej technologies

JProfiler权威指南

性能专家需要知道的全部

Index

JProfiler介绍

什么是JProfiler?

JProfiler是一个专业工具，用于分析正在运行的JVM内部发生的事情。您可以在开发、质量保证以及当生产系统出现问题时的紧急任务中使用它。

JProfiler处理的四个主要主题是：

- **方法调用**

  这通常被称为“CPU分析”。方法调用可以通过不同的方式进行测量和可视化。方法调用的分析帮助您了解应用程序在做什么，并找到提高其性能的方法。

- **分配**

  在堆上分析对象的分配、引用链和垃圾回收属于“内存分析”类别。此功能使您能够修复内存泄漏，通常使用更少的内存，并分配更少的临时对象。

- **线程和锁**

  线程可以通过在对象上同步来持有锁。当多个线程协作时，可能会发生死锁，JProfiler可以为您可视化它们。此外，锁可能会被争用，这意味着线程在获取它们之前必须等待。JProfiler提供了对线程及其各种锁定情况的深入了解。

- **高级子系统**

  许多性能问题发生在更高的语义层次。例如，对于JDBC调用，您可能想找出哪个SQL语句是最慢的。对于这样的子系统，JProfiler提供了“探针”，它们将特定的有效负载附加到调用树。

JProfiler的UI作为桌面应用程序交付。您可以交互式地分析一个实时的JVM，或者在不使用UI的情况下自动分析。分析数据保存在快照中，可以用JProfiler UI打开。此外，命令行工具和构建工具集成帮助您自动化分析会话。

我该如何继续?

本文档旨在按顺序阅读，后面的帮助主题建立在前面内容的基础上。

首先，架构 [p. 5]的技术概述将帮助您理解分析是如何工作的。

关于安装JProfiler [p. 7]和分析JVM [p. 11]的帮助主题将帮助您快速上手。

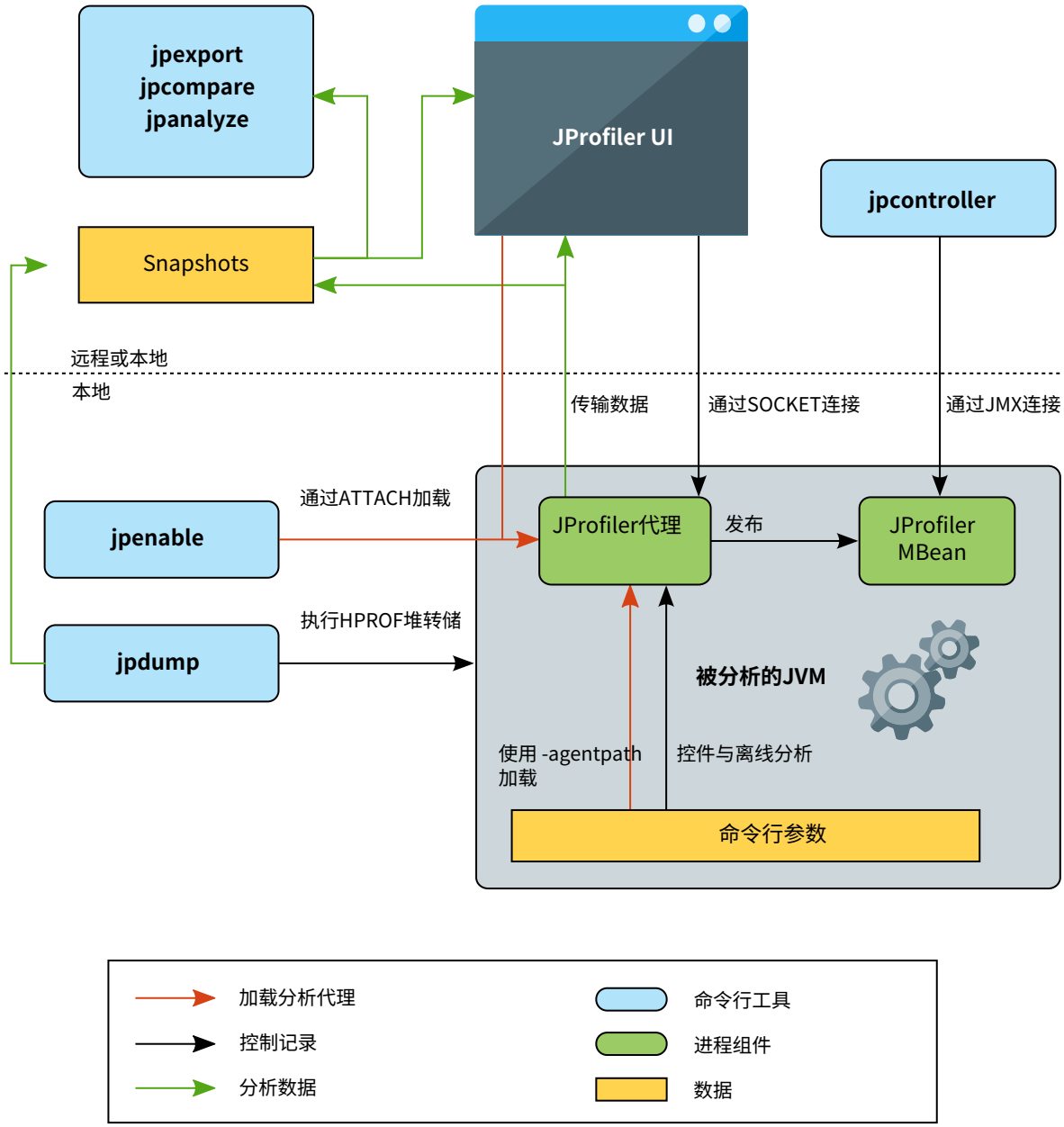
随后，数据记录 [p. 27]和快照 [p. 39]的讨论将带您达到可以自行探索JProfiler的理解水平。

后续章节将根据JProfiler的不同功能建立您的专业知识。最后的部分是可选阅读，如果您需要某些功能，应进行咨询。

我们欢迎您的反馈。如果您觉得某个领域的文档不足，或者发现文档中的不准确之处，请不要犹豫，通过support@ej-technologies.com联系我们。

JProfiler 架构

下图展示了涉及被分析 (profiled) 应用程序、JProfiler UI 和所有命令行实用程序的重要交互的全貌。



分析代理

"JVM 工具接口" (JVMTI) 是一个本机接口，分析器使用它来访问信息并添加钩子以插入自己的检测工具。这意味着至少部分分析代理必须实现为本机代码，因此 JVM 分析器不是平台独立的。JProfiler 支持一系列平台，列表在此处 [p. 7]。

JVM分析器实现为一个本机库，该库在启动时或稍后某个时间点加载。要在启动时加载它，需要在命令行中添加VM参数 `-agentpath:<  >`。您很少需要手动添加此参数，因为JProfiler会为您添加，例如，在IDE集成、集成向导中或直接启动JVM时。然而，了解这就是启用分析的关键是很重要的。

如果 JVM 成功加载了本机库，它会调用库中的一个特殊函数，让分析代理有机会初始化自己。然后 JProfiler 会打印一些以 `JProfiler>` 为前缀的诊断消息，以便您知道分析已激活。底线是，如果您传递了 `-agentpath` VM 参数，分析代理要么成功加载，要么 JVM 无法启动。

一旦加载，分析代理请求JVMTI通知各种事件，例如线程创建或类加载。这些事件中的一些直接提供分析数据。使用类加载事件，分析代理在类加载时检测类并插入自己的字节码以执行其测量。

JProfiler 可以将代理加载到已经运行的 JVM 中，方法是使用 JProfiler UI 或 `bin/jpenable` 命令行工具。在这种情况下，可能需要重新转换大量已经加载的类以应用所需的检测工具。

记录数据

JProfiler 代理仅收集分析数据。JProfiler UI 是单独启动的，并通过 SOCKET 连接到分析代理。对于远程服务器的安全连接，您可以配置 JProfiler 自动创建 SSH 隧道。

从JProfiler UI，您可以指示代理记录数据，在UI中显示分析数据并将快照保存到磁盘。作为UI的替代方案，分析代理可以通过其MBean[1]控制。使用此MBean的命令行工具是`bin/jpcontroller`。

控制分析代理的另一种方法是使用预定义的触发器集和动作。这样，分析代理可以在无人值守模式下运行。这在 JProfiler 中称为 "离线分析"，对于自动化分析会话非常有用。

快照

虽然 JProfiler UI 可以显示实时分析数据，但通常需要保存所有记录的分析数据的快照。快照可以在JProfiler UI 中手动保存，也可以通过触发器动作自动保存。

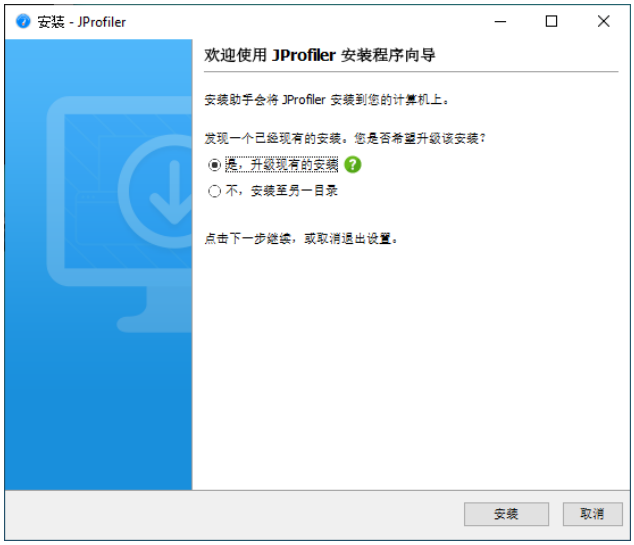
快照可以在 JProfiler UI 中打开和比较。对于自动化处理，可以使用命令行工具 `bin/jpexport` 和 `bin/jpcompare` 从先前保存的快照中提取数据并创建 HTML 报告。

从运行中的 JVM 获取堆快照的一种低开销方法是使用 `bin/jpdump` 命令行工具。它使用 JVM 的内置功能保存一个 HPROF 快照，该快照可以由 JProfiler 打开，并且不需要加载分析代理。

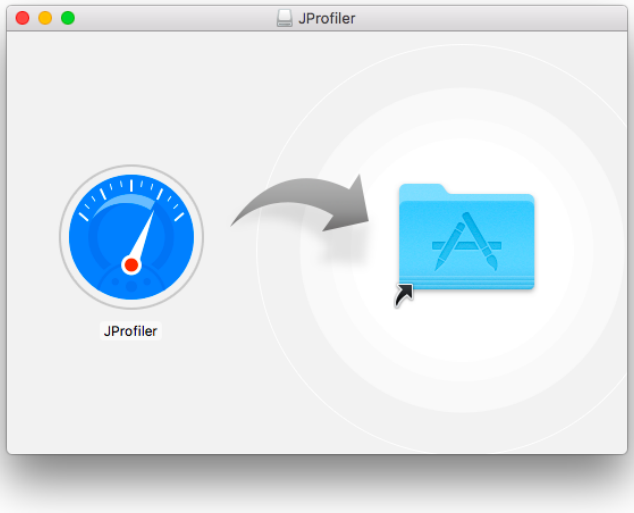
[1] https://en.wikipedia.org/wiki/Java_Management_Extensions

安装 JProfiler

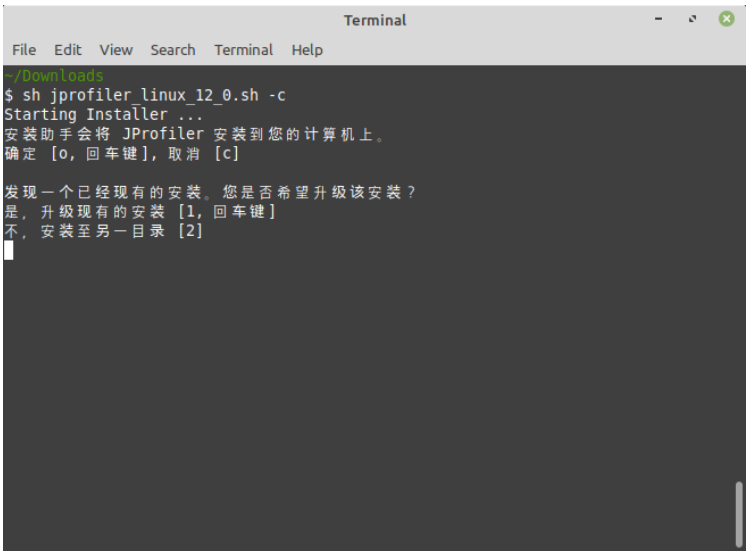
Windows 和 Linux/Unix 提供可执行安装程序，引导您逐步完成安装。 如果检测到先前的安装，安装过程将被简化。



在 macOS 上，JProfiler 使用 UI 应用程序的标准安装程序：一个 DMG 文件，您可以通过双击在 Finder 中挂载， 然后将 JProfiler 应用程序包拖动到 `/Applications` 文件夹。该文件夹在 DMG 本身中显示为符号链接。



在 Linux/Unix 上，下载后安装程序不可执行，因此您需要在执行时添加 sh。 如果传递参数 `-c`，安装程序将执行命令行安装。 使用参数 `-q` 可以在 Windows 和 Linux/Unix 上执行完全无人值守的安装。 在这种情况下，您可以传递附加参数 `-dir <directory>` 以选择安装目录。



运行安装程序后，它将保存一个文件 `.install4j/response.varfile`，其中包含整个用户输入。您可以使用该文件通过在命令行上传递参数 `-varfile <path to response.varfile>` 来自动化无人值守的安装。

要为无人值守的安装设置许可信息，请传递 `-Vjprofiler.licenseKey=<license key>` `-Vjprofiler.licenseName=<user name>`，并可选地 `-Vjprofiler.licenseCompany=<company name>` 作为命令行参数。如果您有浮动许可证，请使用 `FLOAT:<server name or IP address>` 代替许可证密钥。

还提供了 Windows 的 ZIP 文件和 Linux 的 .tar.gz 文件作为存档。

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

将 .tar.gz 存档解压缩到一个单独的顶级目录中。 要启动 JProfiler，请在解压缩的目录中执行 `bin/jprofiler`。在 Linux/Unix 上，文件 `jprofiler.desktop` 可用于将 JProfiler 可执行文件集成到您的窗口管理器中。 例如，在 Ubuntu 上，您可以将桌面文件拖动到启动器侧边栏中以创建永久启动器项目。

将分析代理分发到远程机器

JProfiler 有两个部分：一方面是桌面 UI 和用于快照操作的命令行实用程序， 另一方面是分析代理和用于控制被分析 JVM 的命令行实用程序。您从网站下载的安装程序和存档包含这两个部分。

然而，对于远程分析，您只需要在远程端安装分析代理。虽然您可以简单地在远程机器上提取 JProfiler 分发的存档，但您可能希望限制所需文件的数量，特别是在自动化部署时。此外，分析代理是自由可再分发的，因此您可以将其与您的应用程序一起分发或安装在客户机器上以进行故障排除。

要获取包含分析代理的最小包，远程集成向导会向您显示适当代理存档的下载链接以及所有支持平台的代理存档下载页面。在 JProfiler GUI 中，调用 Session->Integration Wizards->New Server/Remote Integration， 选择“Remote”选项，然后继续到 Remote installation directory 步骤。

特定 JProfiler 版本的 HTML 概览页面的 URL 是

```
https://www.ej-technologies.com/jprofiler/agent?version=15.0.1
```

单个代理存档的下载 URL 格式是

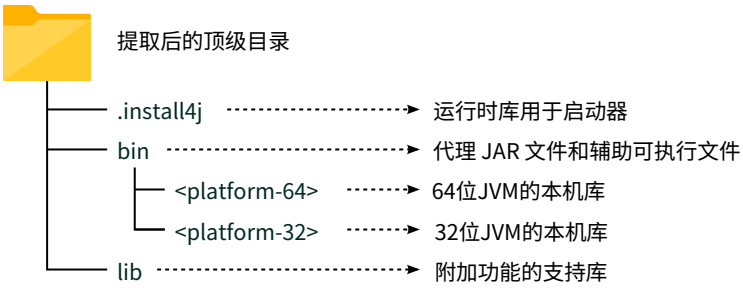
```
https://download.ej-technologies.com/jprofiler/jprofiler_agent_<platform>_15_0_1.<extension>
```

其中 `platform` 对应于 `bin` 目录中的代理目录名称， `extension` 是 Windows 上的 `zip`，macOS 上的 `.tgz`， Linux/Unix 上的 `.tar.gz`。对于 Linux，x86 和 x64 被分组在一起，因此对于 Linux x64，URL 是

```
https://download.ej-technologies.com/jprofiler/jprofiler_agent_linux-x86_15_0_1.tar.gz
```

代理存档包含所需的本机代理库以及 `jpenable`、`jpdump` 和 `jpcontroller` 可执行文件。存档中的可执行文件以及分析代理仅需要 Java 8 作为最低版本。

在远程机器上提取代理存档后看到的子目录如下所述。 它们是相应目标平台上完整 JProfiler 安装的子集。



支持的平台

因为 JProfiler 利用 JVM (JVMTI) 的本机分析接口，所以其分析代理是一个本机库。

JProfiler 支持在以下平台上进行分析：

操作系统	架构	支持的 JVM	版本
Windows 11/10 Windows Server 2025/2022/2019/2016	x86 x64/AMD64	Hotspot (OpenJDK) IBM/OpenJ9	1.8 - 24 1.8 - 24
macOS 10.12 - 15	Intel, Apple	Hotspot (OpenJDK) IBM/OpenJ9	1.8 - 24 1.8 - 24
Linux	x86 x64/AMD64	Hotspot (OpenJDK) IBM/OpenJ9	1.8 - 24 1.8 - 24
Linux	PPC64LE	Hotspot (OpenJDK) IBM/OpenJ9	1.8 - 24 1.8 - 24
Linux	ARMv7 ARMv8	Hotspot (OpenJDK)	1.8 - 24

JProfiler GUI 前端需要 Java 21 VM 才能运行。为此目的，JProfiler 在 Windows、macOS 和 Linux x64 上捆绑了 Java 21 JRE。 附加命令行工具 jpenable、jdump 和 jpcontroller 仅需要 Java 8 VM。

JVMのプロファイリング

JVMをプロファイリングするには、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。これには2つの方法があります：起動スクリプトで`-agentpath`VMパラメータを指定するか、すでに実行中の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方法はプロファイリングの推奨方法であり、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が提供する一般的な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)]でのみプロファイリングできることを意味します。32ビットと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:`の後には、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。たとえば、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からの接続をリッスンす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なので、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をプロファイリングしたい場合は、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、プロファイリングエージェントに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するためだけのもので、どのように開始するかを指示し、デバッグフラグを渡すためのものです。

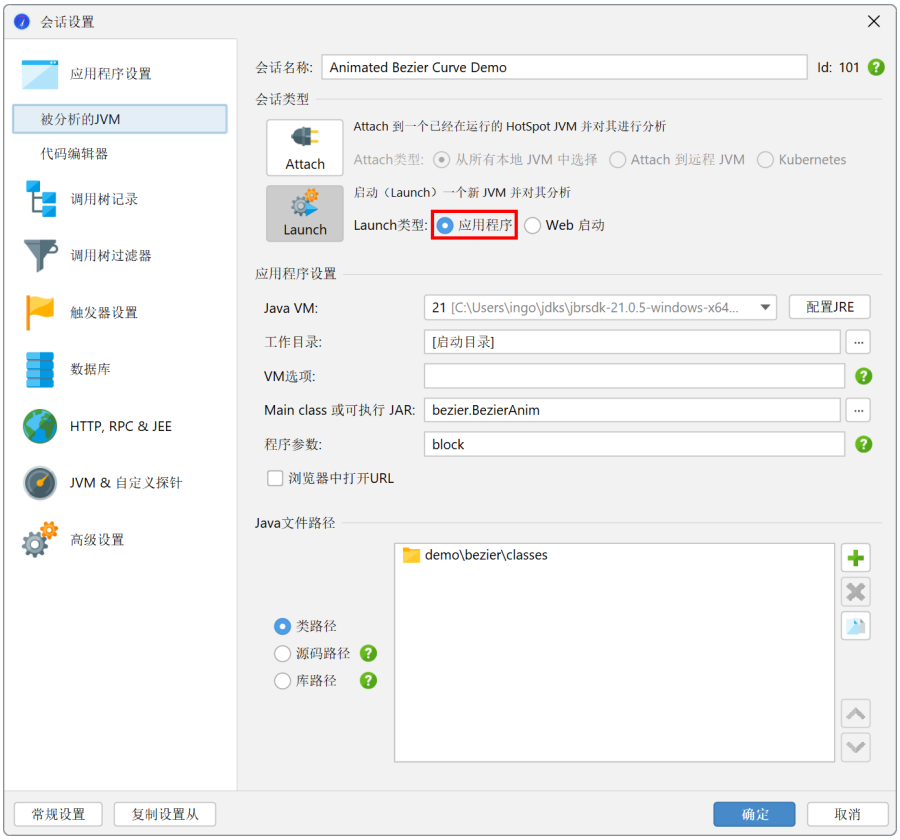
特定の状況下では、起動時にプロファイリング設定を設定する [p. 229]ことが必要であり、 これを達成するためにいくつかの手作業が必要になる場合があります。

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、通信ソケットをすべての利用可能なネットワークインターフェースにバインドするために`address=0.0.0.0`を追加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

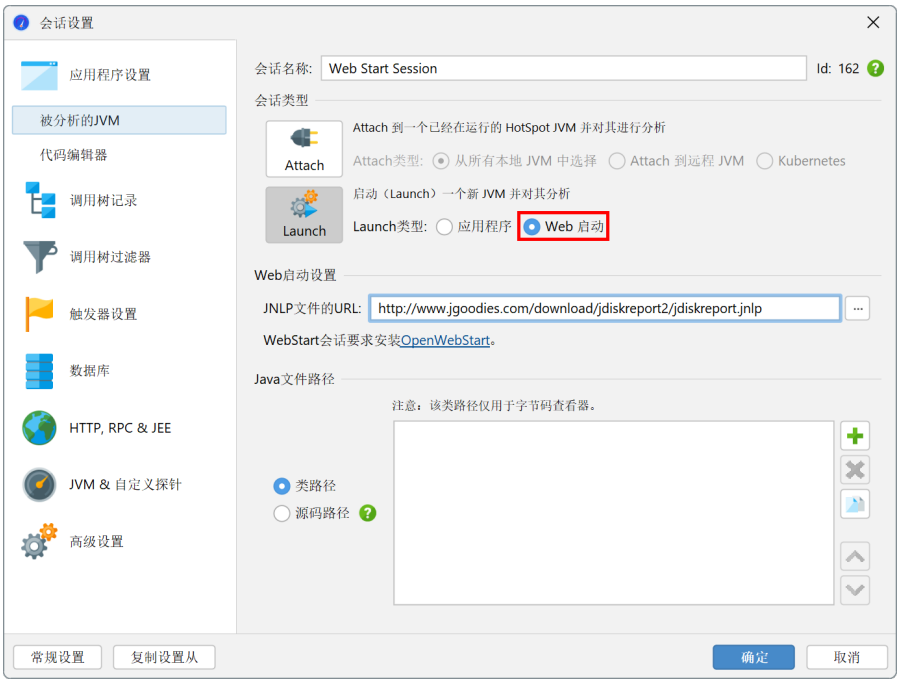
ローカルで起動されたセッション

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。

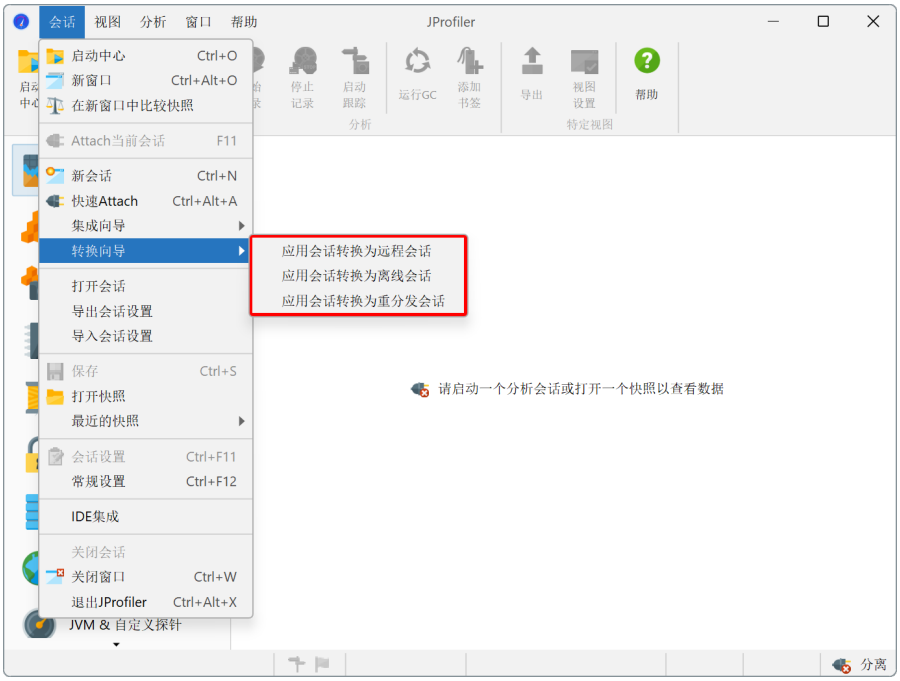


特別な起動モードは「Web Start」で、JNLPファイルのURLを選択し、JProfilerがそれをプロファイリングするためのjvmを起動します。この機能は OpenWebStart [(2)]をサポートし、Java 9以前のOracle JREのレガシーWebStartはサポートされていません。

[(2)] https://openwebstart.com/



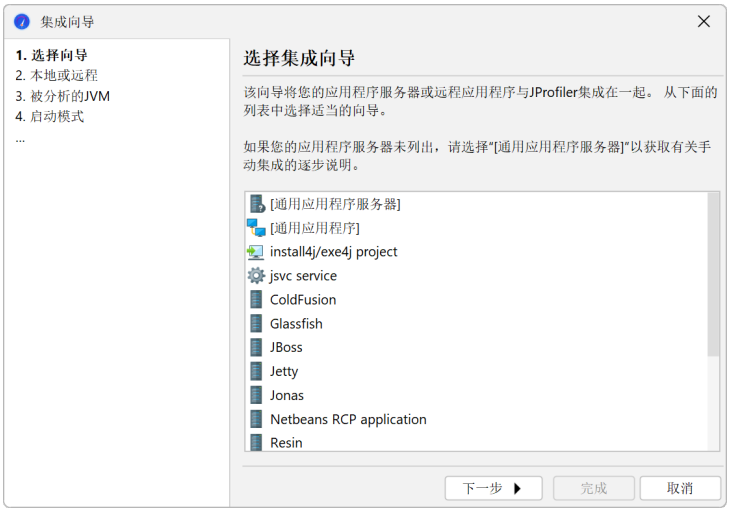
ローカルで起動されたセッションは、メインメニューからSession->Conversion Wizardsを呼び出すことで変換ウィザードを使用してスタンドアロンセッションに変換できます。Convert Application Session to Remoteは単に開始スクリプトを作成し、`-agentpath` VMパラメータをJava呼び出しに挿入します。 Convert Application Session to Offlineは`offline profiling`のための開始スクリプトを作成し、設定が起動時にロードされ、JProfiler GUIが必要ありません。Convert Application Session to Redistributed Session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信できます。



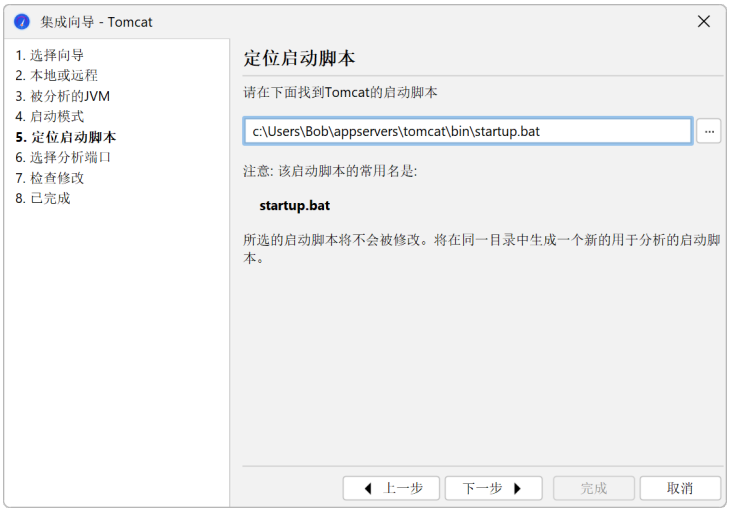
プロファイリングするアプリケーションを自分で開発している場合は、起動されたセッションの代わりにIDE integrationを使用することを検討してください。それはより便利で、より良いソースコードナビゲーションを提供します。アプリケーションを自分で開発していないが、すでに開始スクリプトを持っている場合は、リモート統合ウィザードを使用することを検討してください。それはJava呼び出しに追加する必要がある正確なVMパラメータを教えてくれます。

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製コンテナを処理します。 一部の製品では、VMパラメータを引数として渡すか、環境変数を介して渡す開始スクリプトを生成できます。



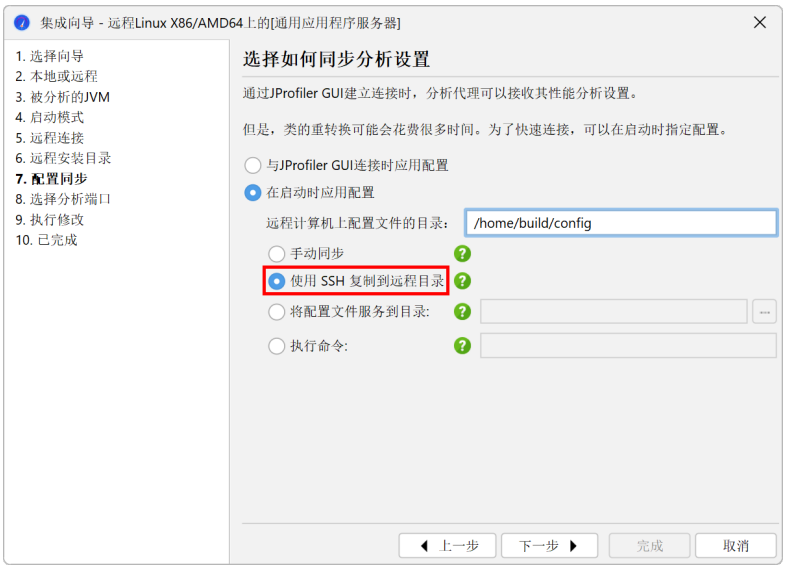
すべての場合において、サードパーティ製品の特定のファイルを見つける必要があるため、JProfilerが変更を行うために必要なコンテキストを持つことができます。一部の一般的なウィザードは、プロファイリングを有効にするために何をする必要があるかについての指示を提供するだけです。



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リングするか、リモートマシンでプロファイリングするかの選択です。ローカルマシンの場合、JProfilerはすでにプラットフォーム、JProfilerがインストールされている場所、および設定ファイルがどこにあるかを知っているため、提供する情報が少なくて済みます。

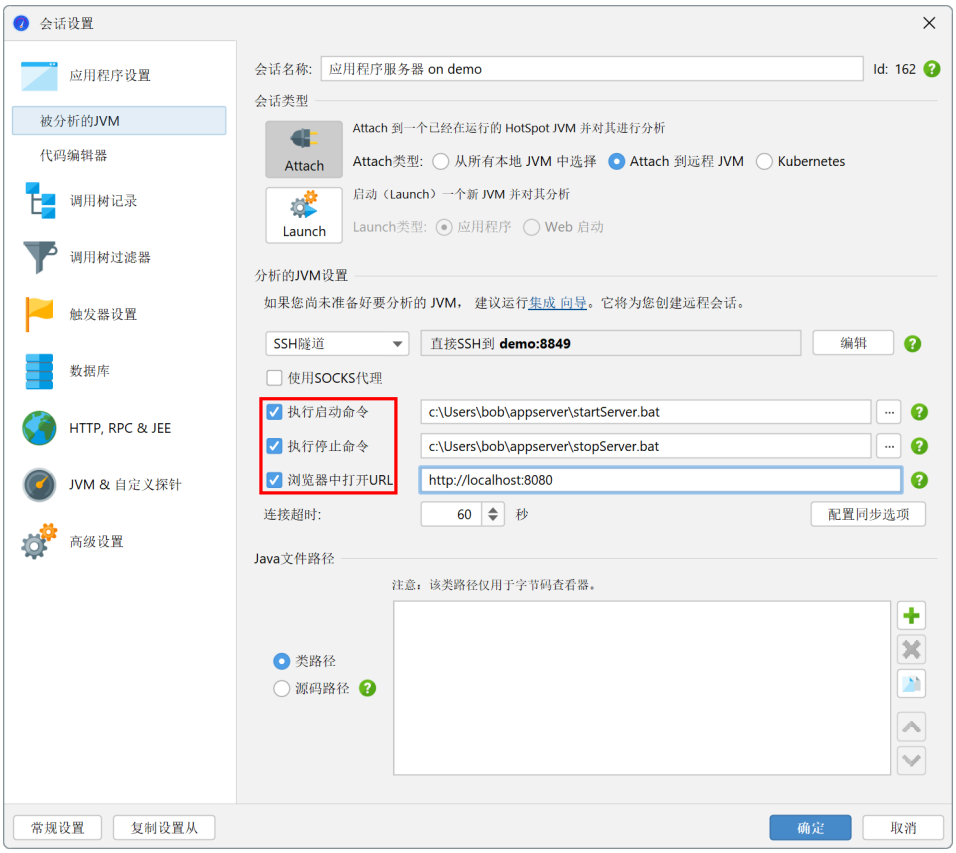
重要な決定は、上記で説明した「起動モード」です。デフォルトでは、プロファイリング設定は起動時にJProfiler UIから送信されますが、JVMをすぐに起動させるようにプロファイリングエージェントに指示することもできます。後者の場合、プロファイリング設定はJProfiler GUIが接続されたときに適用されます。

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、これははるかに効率的です。これはConfig synchronizationステップで行われます。 この場合の主な問題は、ローカルでプロファイリング設定を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、Remote address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経由で自動的に転送できるようにすることです。

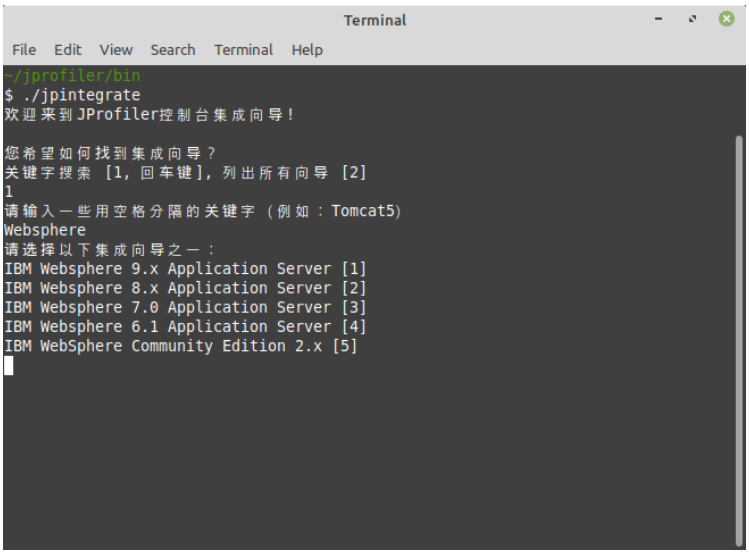


統合ウィザードの最後に、プロファイリングを開始するセッションが作成され、非一般的なケースではサードパーティ製品（アプリケーションサーバーなど）も開始されます。

外部開始スクリプトは、セッション設定ダイアログのApplication settingsタブのExecute start scriptおよびExecute stop script オプションで処理され、URLはOpen browser with URLチェックボックスを選択することで表示できます。ここでは、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することもできます。



統合ウィザードはすべて、プロファイリングされるJVMがリモートマシンで実行されている場合を処理します。しかし、設定ファイルや開始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。

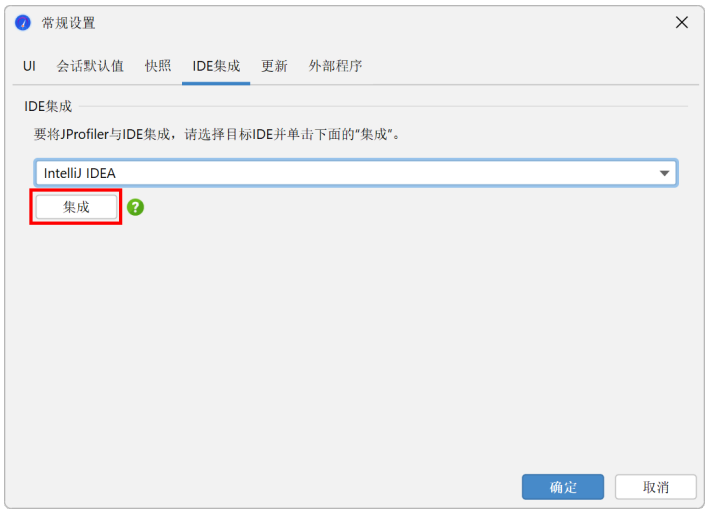


リモートプロファイリングセッションを開始する際にエラーが発生した場合は、問題を解決するために取るべき手順のチェックリストを含む トラブルシューティングガイド [p. 222]を参照してください。

IDE統合

アプリケーションをプロファイリング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、 プロファイリングされた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推奨方法は、 メインメニューからSession->IDE integrations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に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、IDEの関連ウィンドウに簡単にジャンプできるようにします。 すべてのShow Sourceアクションは、JProfilerの組み込みソースビューアの代わりにIDEで直接ソースを表示します。

IDE統合については、後の章 [p. 129]で詳しく説明されています。

アタッチモード

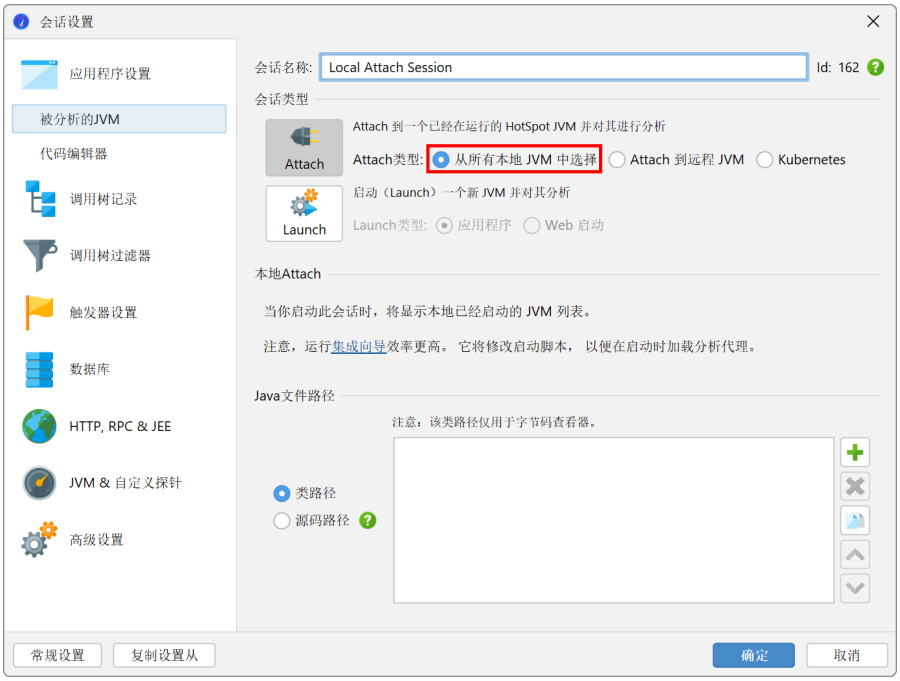
JVMをプロファイリングするつもりであることを事前に決定する必要はありません。JProfilerのアタッチ機能を使用すると、実行中のJVMを選択し、プロファイリングエージェントをその場でロードできます。 アタッチモードは便利ですが、いくつかの欠点があることを知っておくべきです:

- プロファイリングしたいJVMを実行中のJVMのリストから特定する必要があります。これは、同じマシンで多くのJVMが実行されている場合には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が必要です。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場合はデフォルトで無効になっています。
- 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いけません。

JProfilerのスタートセンターのQuick Attachタブには、プロファイリング可能なすべてのJVMがリストされます。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。

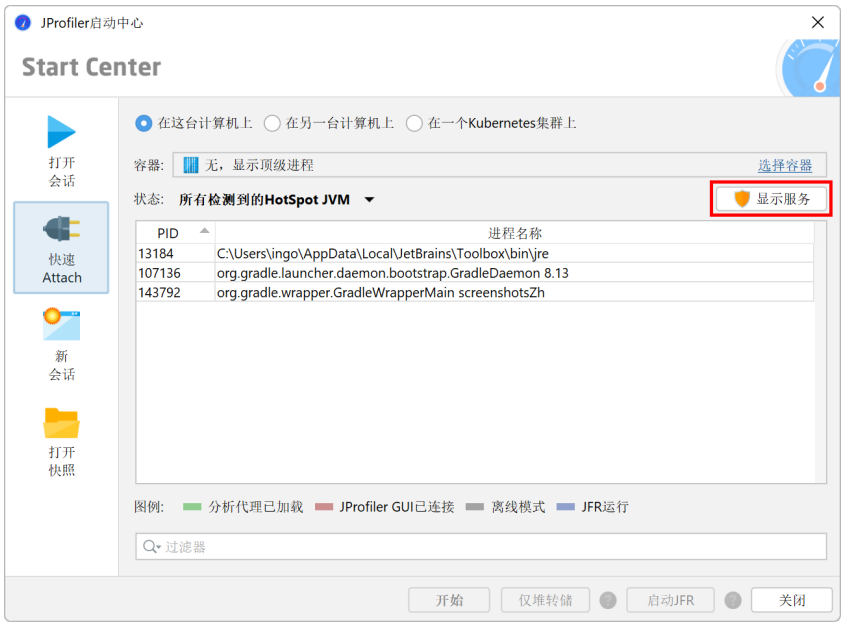
プロファイリングセッションを開始すると、セッション設定ダイアログでプロファイリング設定を構成できます。同じプロセスを繰り返しプロファイリングする場合、同じ設定を何度も再入力したくないので、クイックアタッチ機能で作成されたセッションを閉じるときに永続的なセッションを保存できます。次回このプロセスをプロファイリングしたい場合は、Quick Attachタブの代わりにOpen Sessionタブから保存されたセッションを開始します。 実行中のJVMを選択する必要がありますが、プロファイリング設定は以前に構成したものと同じです。



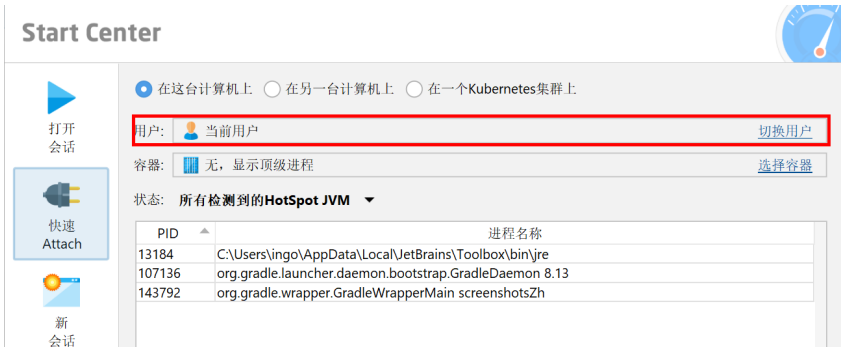
ローカルサービスへのアタッチ

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Show Servicesボタンがあり、ローカルで実行中のすべて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るかに関係なく、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でユーザーを直接切り替えることをサポートしており、これはほとんどのLinuxディストリビューションに含まれています。 アタッチダイアログでSwitch user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バリアントや Linuxディストリビューションに応じて、 macOSや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
```

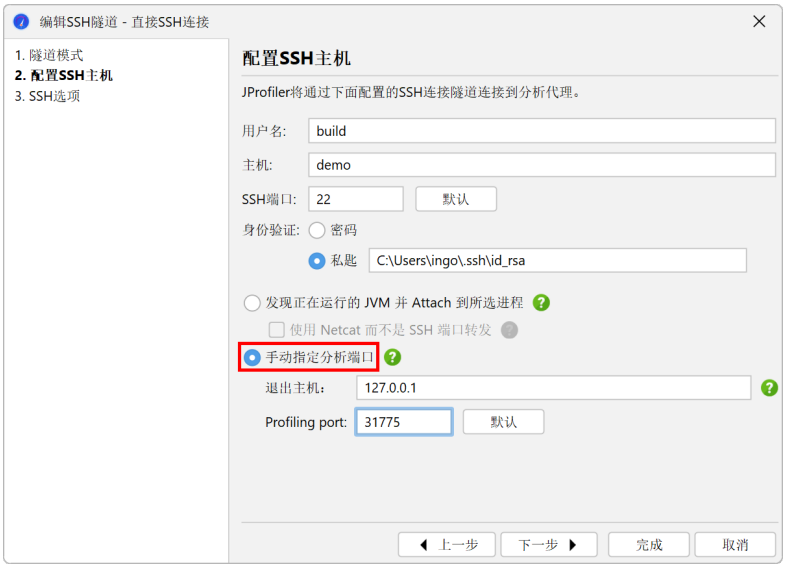
`jpenable`はJVMを選択し、プロファイリングエージェントがリッスンしているポートを教えてくれます。その後、 JProfiler UIからのローカルセッションまたはjpenableによって指定されたポートに直接接続するSSH接続で接続できます。

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、 プロファイリングされるJVMは別のマシンで実行されます。プロファイリングされる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を使用すると、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`Installing JProfiler`ヘルプトピックで説明したエージェントパッケージをアップロードし、リモートマシンで含まれているコマンドラインツールを実行できます。ローカルマシンでSSHを設定する必要はありません。JProfilerには独自の実装が付属しています。最も簡単なセットアップでは、ホスト、ユーザー名、および認証を定義するだけです。

SSH接続を使用すると、JProfilerは実行中のJVMを自動的に検出するか、プロファイリングエージェントがすでにリッスンしている特定のポートに接続できます。　　後者の場合、リモートマシンで`jpenable`または`jpintegrate`を使用し、 プロファイリング用の特別なJVMを準備します。その後、SSHリモートアタッチを設定して、設定されたプロファイリングポートに直接接続できます。



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、これはプロファイリング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Switch User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、GUIでマルチホップSSHトンネルを使用してJProfilerに接続するように指示できます。 SSHトンネルの終点では、通常「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、コマンドラインで入力するように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。ホスト名のほかに、ポートとユーザー名をオプションで指定できます。 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。

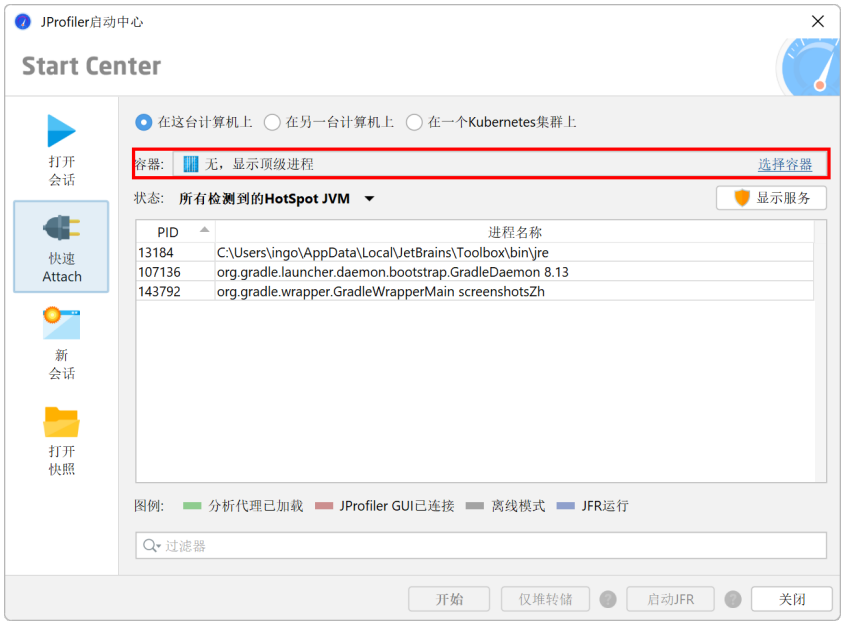
SSHオプションテキストフィールドは、OpenSSH実行ファイルに指定する追加の引数を任意に受け取ります。これは、たとえばAWSセッションマネージャーを介してSSH接続をトンネルする方法についての チュートリアルで指示された手順に従う場合に特に便利です。

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移譲することはできません。フル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。 代わりに、一般設定ダイアログの「外部ツール」タブで設定することもできます。



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、JVMをプロファイリング用に準備し、プロファイリングプロトコルを外部にトンネルします。

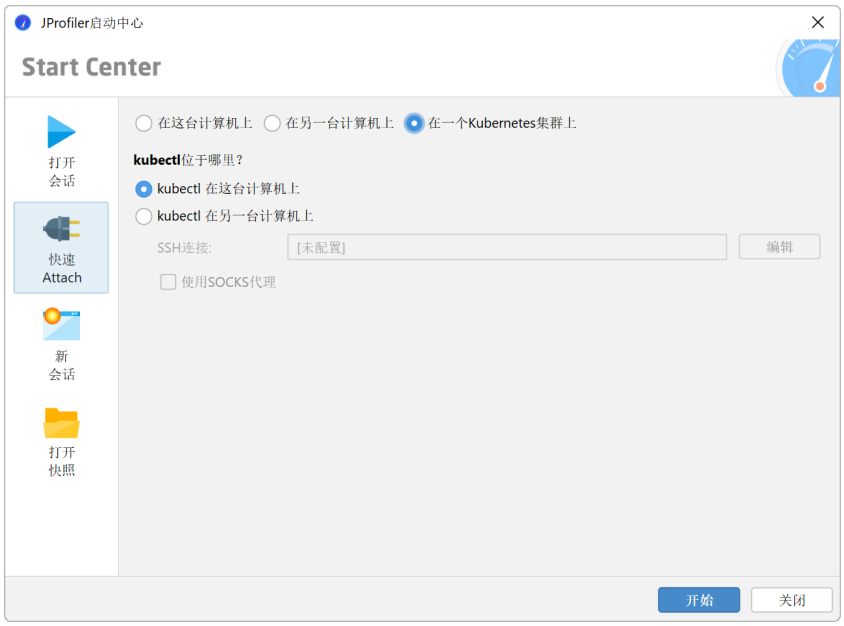
リモートDockerインストールの場合、SSHリモートアタッチ機能を使用し、リモートマシン上のDockerコンテナを選択できます。ログインユーザーがdockerグループに属していない場合は、 上記のようにユーザーを切り替えることができます。

リモートアタッチダイアログのSelect container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。

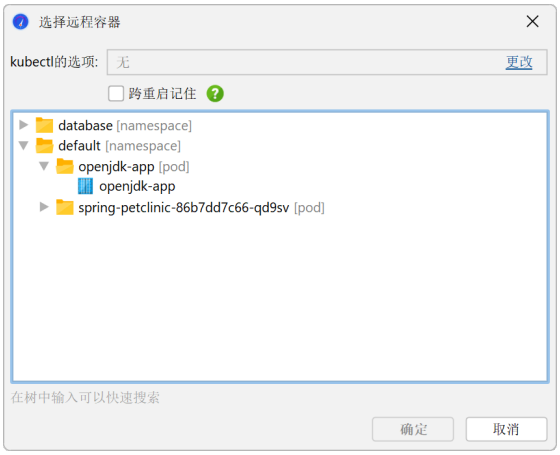
Kubernetesクラスターで実行されているJVMへのアタッチ

Kubernetesクラスターで実行されているJVMをプロファイリングするには、JProfilerは`kubectl`コマンドラインツールを使用して、ポッドとコンテナを発見し、コンテナに接続してそのJVMをリストし、最終的に選択した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可能性があります。 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することもできます。



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オフにすると、 以前に保存された情報が直ちにクリアされます。

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、プロファイリングされた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に、表示名を自分で設定したい場合は、 VMパラメータ`-Djprofiler.displayName=[name]`を設定できます。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。

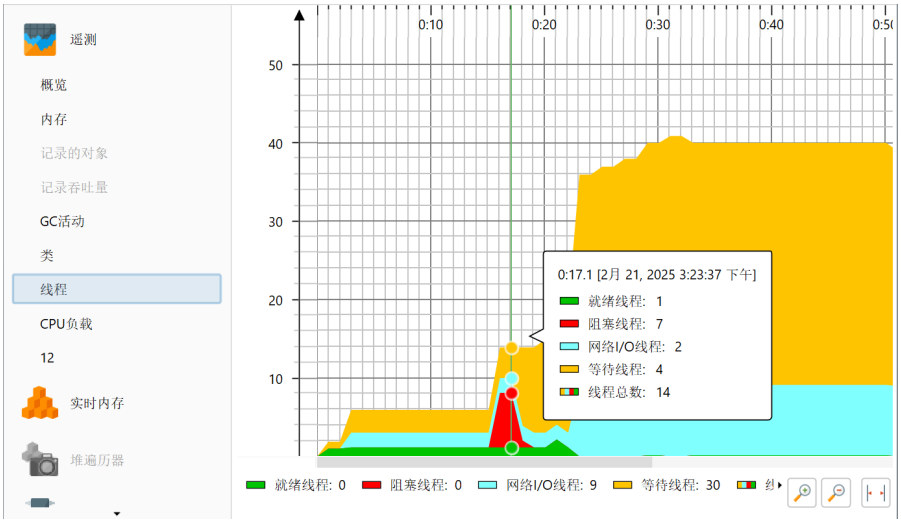
记录数据

Profiler 的主要目的是从各种来源记录运行时数据，这些数据对于解决常见问题非常有用。此任务的主要问题是运行中的 JVM 以惊人的速度生成此类数据。如果 Profiler 始终记录所有类型的数据，它将产生不可接受的开销或迅速耗尽所有可用内存。此外，您通常希望围绕特定用例记录数据，而不查看任何不相关的活动。

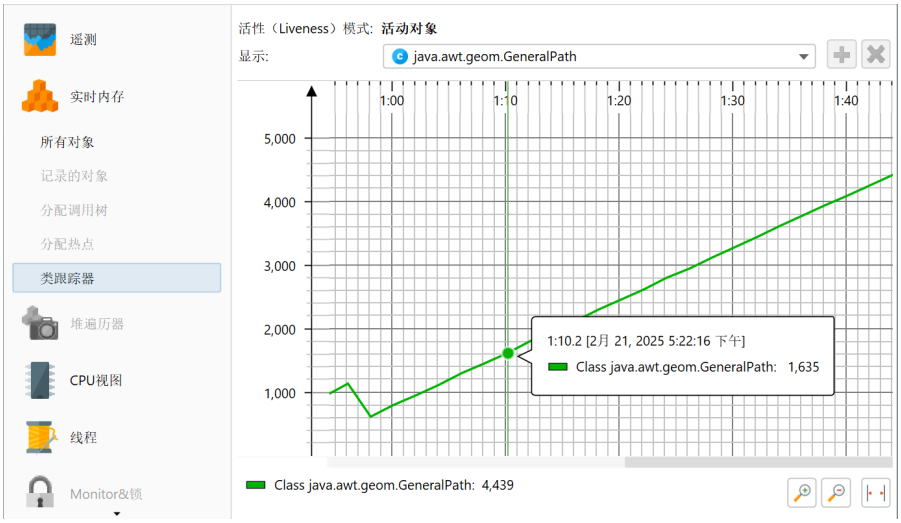
这就是 JProfiler 提供细粒度机制来控制您实际感兴趣的信息记录的原因。

标量值和遥测

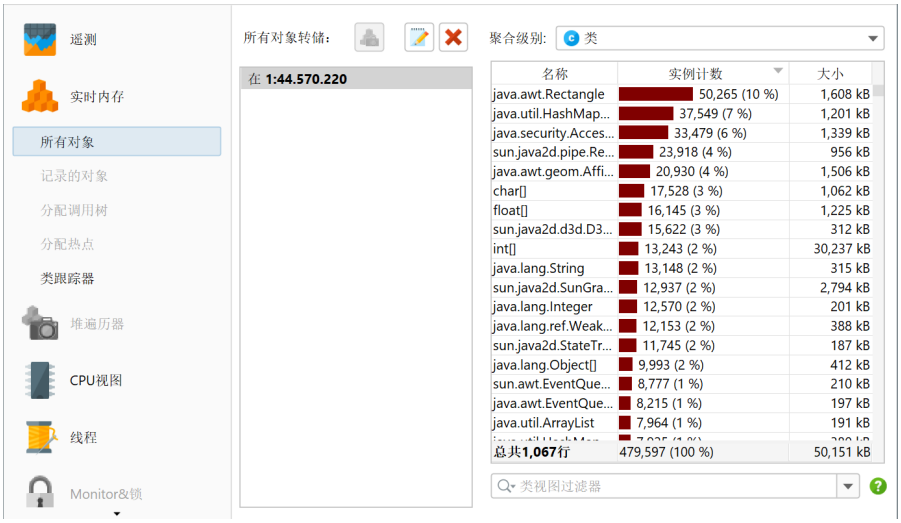
从 Profiler 的角度来看，数据中最不成问题的形式是标量值，例如活动线程的数量或打开的 JDBC 连接的数量。JProfiler 可以以固定的宏观频率（通常为每秒一次）对这些值进行采样，并向您展示随时间的演变。在 JProfiler 中，显示此类数据的视图称为 遥测 [p. 44]。大多数遥测始终被记录，因为测量的开销和内存消耗很小。如果长时间记录数据，较旧的数据点会被合并，以便内存消耗不会随时间线性增长。



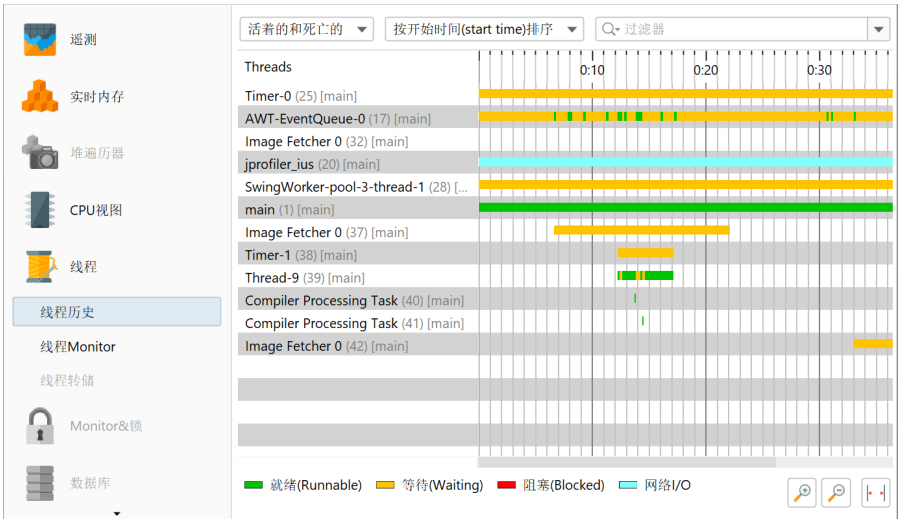
还有参数化的遥测，例如每个类的实例数量。额外的维度使永久的时间顺序记录不可持续。您可以告诉 JProfiler 记录选定类的实例计数的遥测，但不是每个类的。



继续前面的例子，JProfiler 能够向您展示所有类的实例计数，但没有时间顺序信息。这是“所有对象”视图，它将每个类显示为表中的一行。更新视图的频率低于每秒一次，并且可能会根据测量造成的开销自动调整。确定所有类的实例计数相对昂贵，并且对象越多，所需时间越长。这就是为什么“所有对象”不会自动更新，您需要手动创建所有对象的新转储。



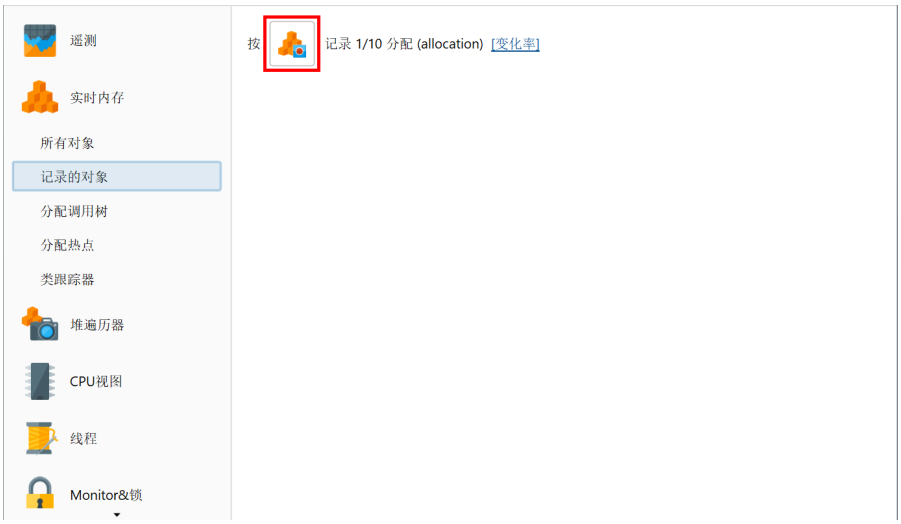
一些测量捕获枚举类值，例如线程当前的执行状态。这种类型的测量可以显示为彩色时间线，并且消耗的内存比数值遥测少得多。在线程状态的情况下，“线程历史”视图显示 JVM 中所有线程的时间线。就像数值遥测一样，较旧的值被合并并变得更加粗粒度，以减少内存消耗。



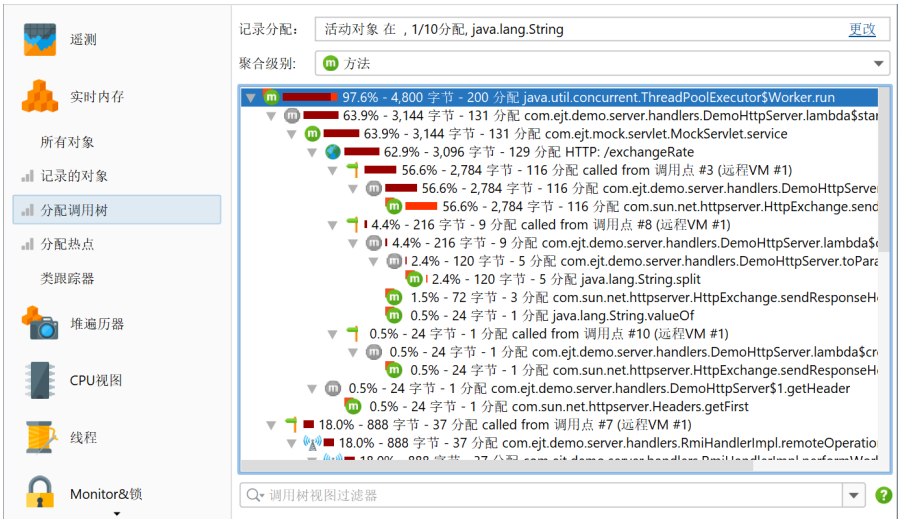
分配记录

如果您对在某个时间间隔内分配的实例计数感兴趣，JProfiler必须跟踪所有分配。与“所有对象”视图相反，JProfiler可以遍历堆中的所有对象以按需获取信息，跟踪单个分配需要为每个对象分配执行额外的代码。这使得它成为一种非常昂贵的测量，可能会显著改变被分析应用程序的运行时特性，例如性能热点，特别是如果您分配了许多对象。这就是为什么分配记录必须明确启动和停止的原因。

具有相关记录的视图最初显示一个带有记录按钮的空白页面。工具栏中也可以找到相同的记录按钮。



分配记录不仅记录分配的实例数量，还记录分配的堆栈跟踪。为每个分配记录保留堆栈跟踪在内存中会产生过多的开销，因此JProfiler将记录的堆栈跟踪累积到树中。这也有一个优势，您可以更轻松地解释数据。但是，时间顺序方面丢失，无法从数据中提取某些时间范围。



内存分析

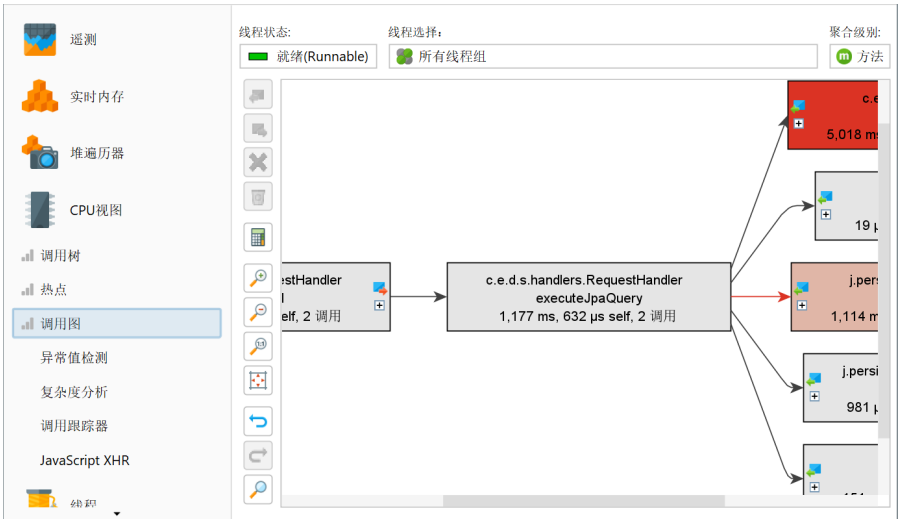
分配记录只能测量对象的分配位置，并且没有关于对象之间引用的信息。任何需要引用的内存分析，例如解决内存泄漏，都是在堆漫游器中完成的。堆漫游器会对整个堆进行快照并进行分析。这是一种侵入性操作，会暂停 JVM（可能会持续很长时间），并且需要大量内存。

一种更轻量级的操作是在开始用例之前标记堆上的所有对象，以便您可以在稍后进行堆快照时找到所有新分配的对象。

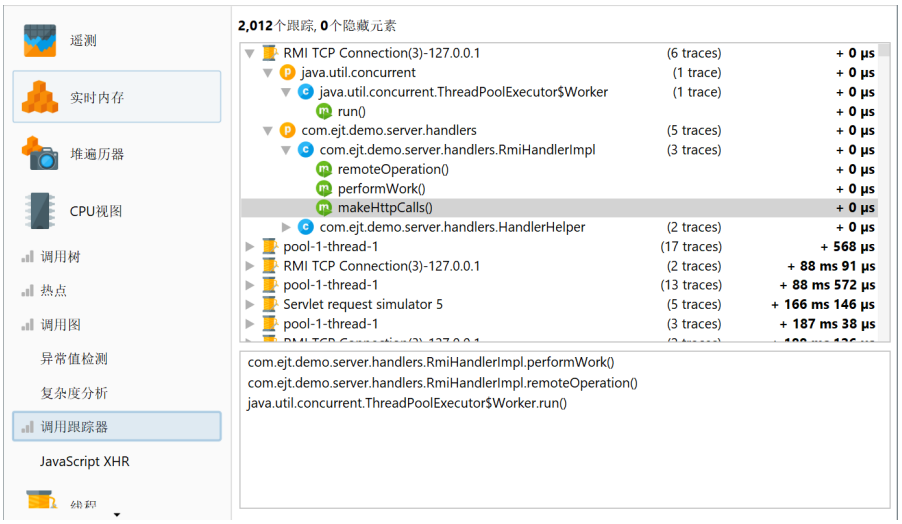
JVM 具有一个特殊的触发器，用于将整个堆转储到一个以旧 HPROF Profiler 代理命名的文件中。这与分析接口无关，也不受其约束。因此，HPROF 堆转储速度更快，使用的资源更少。缺点是当在堆漫游器中查看堆快照时，您将没有与 JVM 的实时连接，并且某些功能不可用。

方法调用记录

测量方法调用所需的时间是一种可选记录，就像分配记录一样。方法调用被累积到树中，并且有各种视图从不同的角度显示记录的数据，例如调用图。在 JProfiler 中，这种类型的数据记录称为“CPU 记录”。

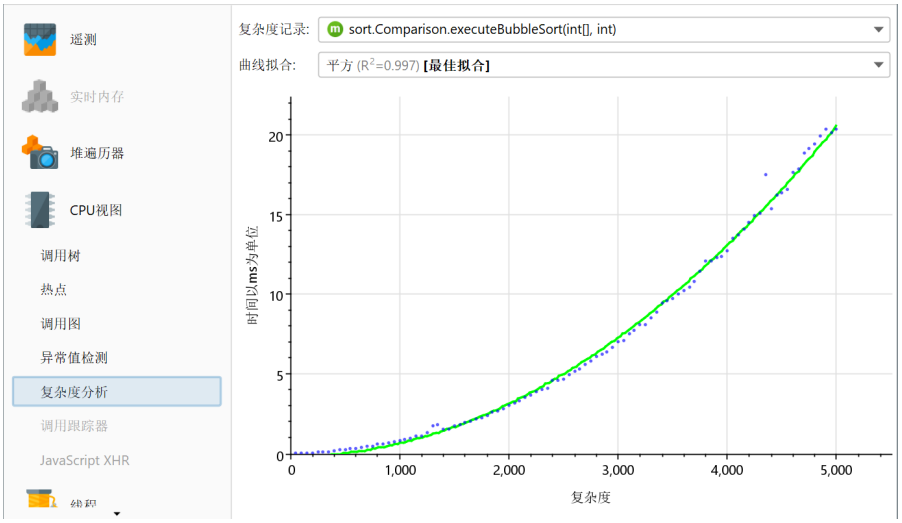


在特定情况下，查看方法调用的时间顺序可能很有用，尤其是在涉及多个线程时。对于这些特殊情况，JProfiler 提供了“调用跟踪器”视图。该视图具有与更通用的 CPU 记录无关的单独记录类型。请注意，调用跟踪器生成的数据过多，无法用于解决性能问题，它仅用于一种特殊形式的调试。



调用跟踪器依赖于 CPU 记录，并在必要时自动打开。

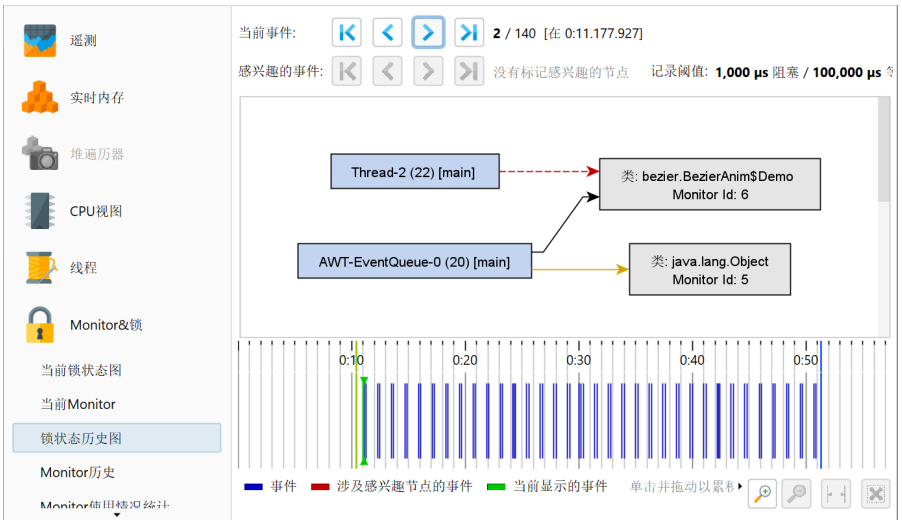
另一个具有自己记录的专用视图是“复杂性分析”。它仅测量选定方法的执行时间，并且不需要启用 CPU 记录。其附加数据轴是方法调用的算法复杂性的数值，您可以使用脚本计算该值。通过这种方式，您可以测量方法的执行时间如何依赖于其参数。



Monitor 记录

要分析线程等待或阻塞的原因，必须记录相应的事件。此类事件的速率差异很大。对于多线程程序，其中线程经常协调任务或共享公共资源，可能会有大量此类事件。这就是为什么默认情况下不记录此类时间顺序数据的原因。

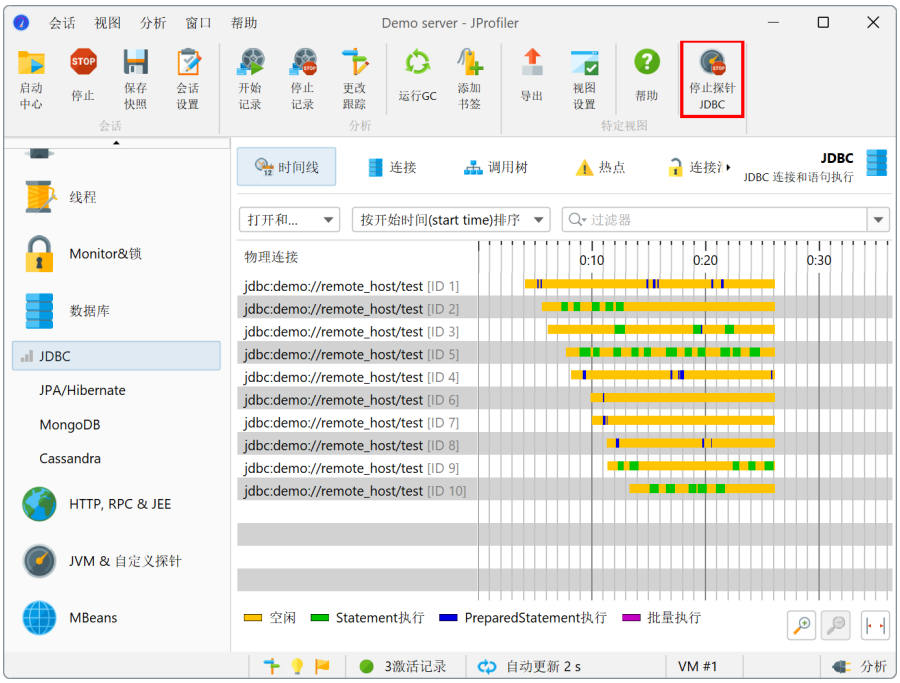
当您打开 Monitor 记录时，“锁定历史图”和“Monitor 历史”视图将开始显示数据。



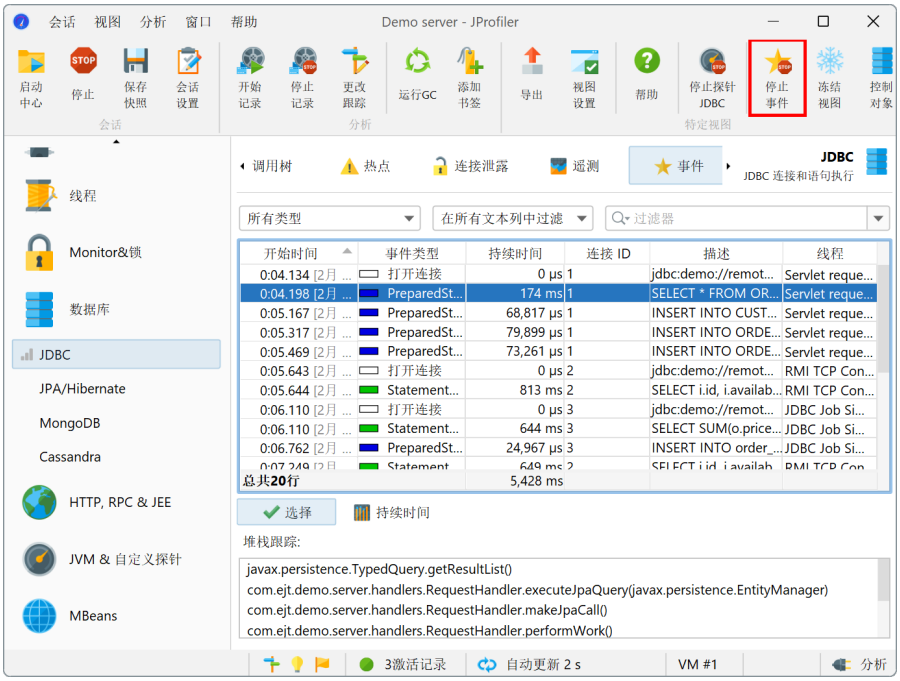
为了消除噪音并减少内存消耗，非常短的事件不会被记录。视图设置为您提供了调整这些阈值的可能性。

探针记录

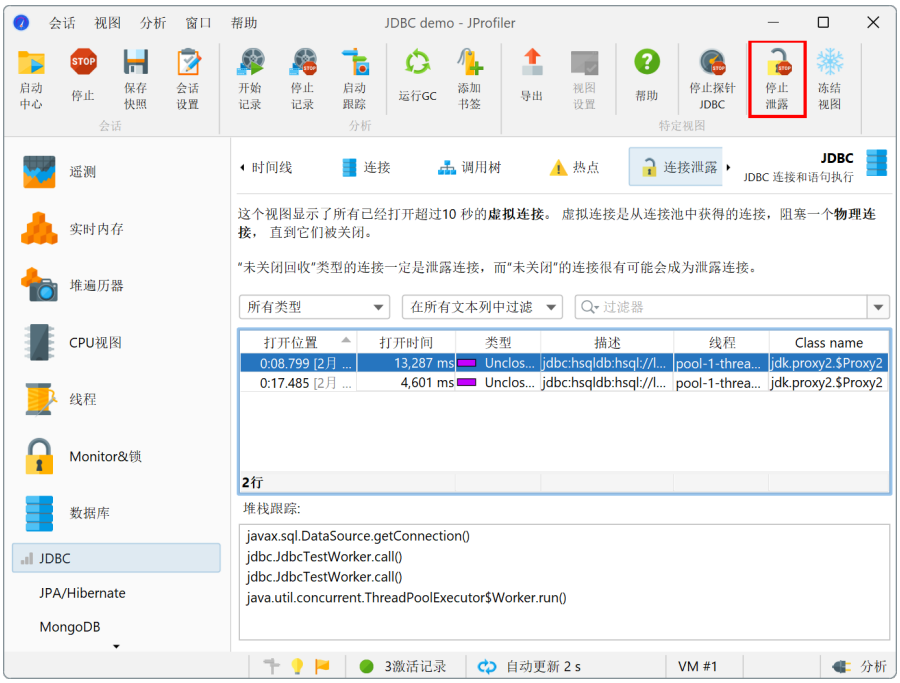
探针显示 JVM 中的高级子系统，例如 JDBC 调用或文件操作。默认情况下，不记录任何探针，您可以分别为每个探针切换记录。某些探针会增加很少或没有开销，而某些探针会根据您的应用程序正在执行的操作以及探针的配置产生大量数据。



就像分配记录和方法调用记录一样，探针数据被累积，时间顺序信息被丢弃，除了时间线和遥测。然而，大多数探针还有一个“事件”视图，允许您检查单个事件。这会增加潜在的大量开销，并具有单独的记录操作。该记录操作的状态是持久的，因此当您切换探针记录时，如果您之前已打开它，相关的事件记录也会被切换。

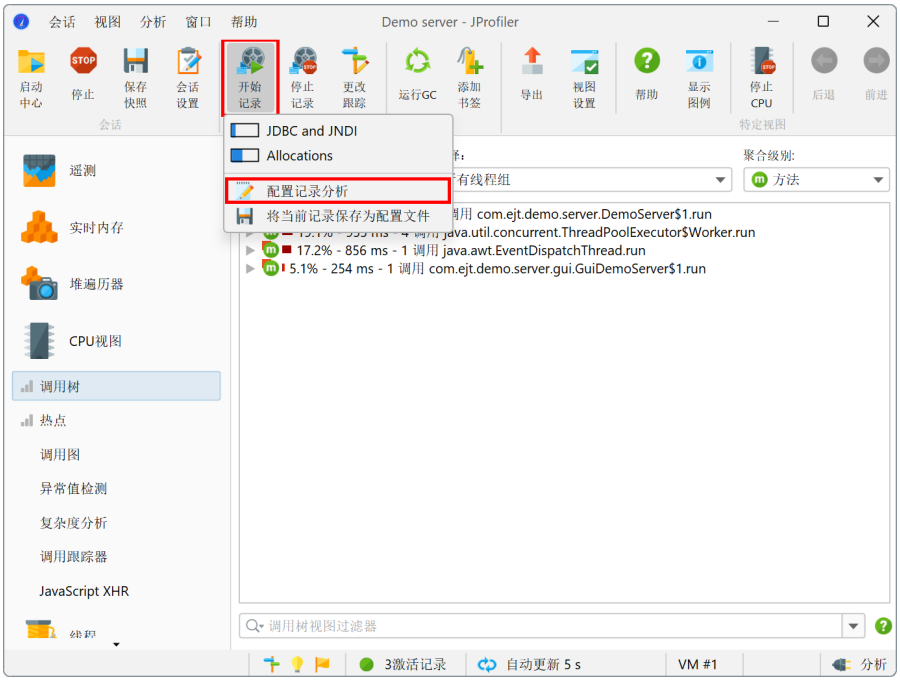


JDBC 探针具有第三个记录操作，用于记录 JDBC 连接泄漏。查找连接泄漏的相关开销仅在您实际尝试调查此类问题时才会产生。就像事件记录操作一样，泄漏记录操作的选择状态是持久的。

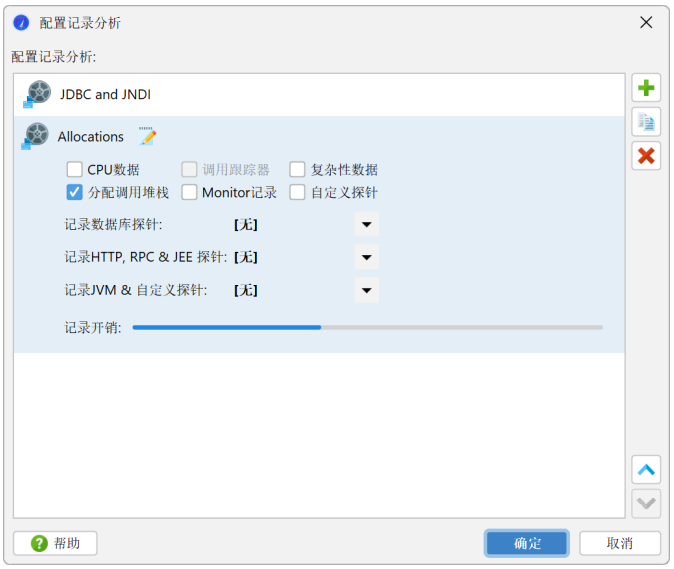


记录配置文件

在许多情况下，您希望通过单击一次来启动或停止各种记录。逐一访问所有相应的视图并切换记录按钮是不切实际的。这就是 JProfiler 具有记录配置文件的原因。可以通过单击工具栏中的 开始记录 按钮来创建记录配置文件。



记录配置文件定义可以原子激活的特定记录组合。JProfiler尝试向您提供有关所选记录所产生的开销的粗略印象，并尝试阻止有问题的组合。特别是，分配记录和 CPU 记录不能很好地结合在一起，因为分配记录会显著扭曲 CPU 数据的时间。



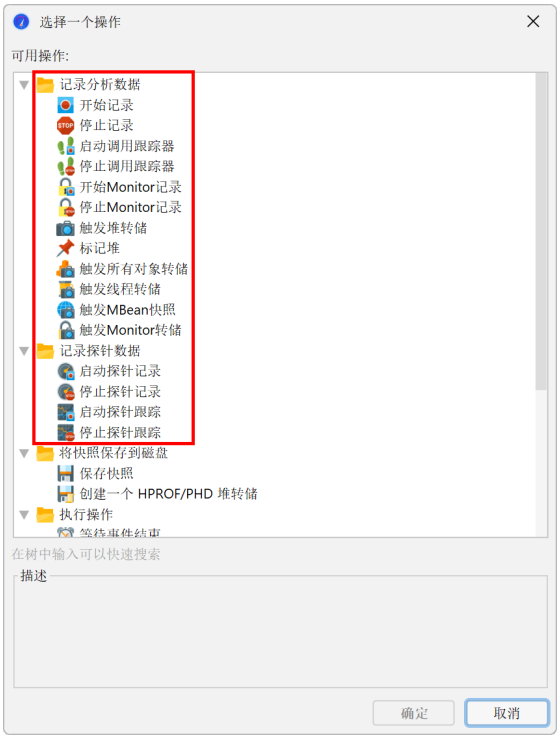
您可以在会话运行时随时激活记录配置文件。记录配置文件不是累加的，它们会停止所有不包含在记录配置文件中的记录。使用 停止记录 按钮，您可以停止所有记录，无论它们是如何激活的。要检查当前激活的记录，请将鼠标悬停在状态栏中的记录标签上。

在开始分析时也可以直接激活记录配置文件。“会话启动”对话框具有一个 初始记录配置文件 下拉菜单。默认情况下，未选择任何记录配置文件，但如果您需要 JVM 启动阶段的数据，这是配置所需记录的地方。

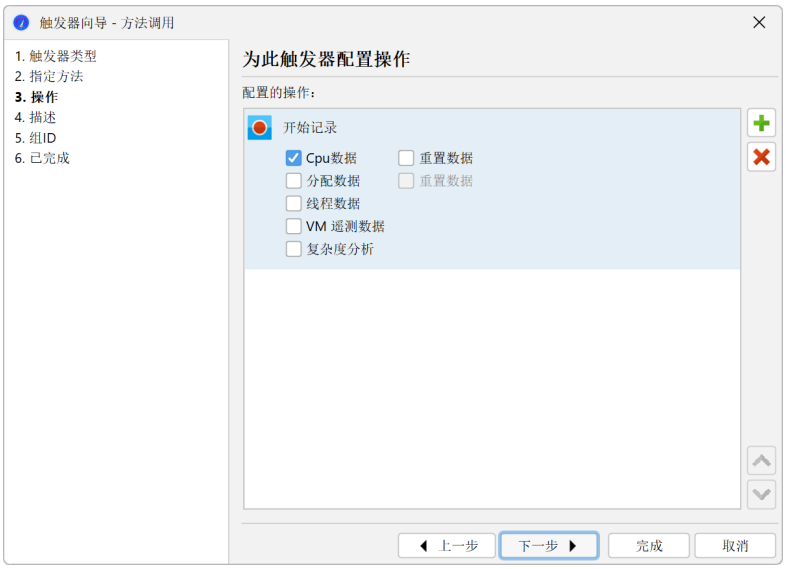
使用触发器记录

有时您希望在特定条件发生时开始记录。JProfiler 具有一个 定义触发器的系统 [p. 119]，该系统执行一系列操作。可用的触发器操作还包括对活动记录的更改。

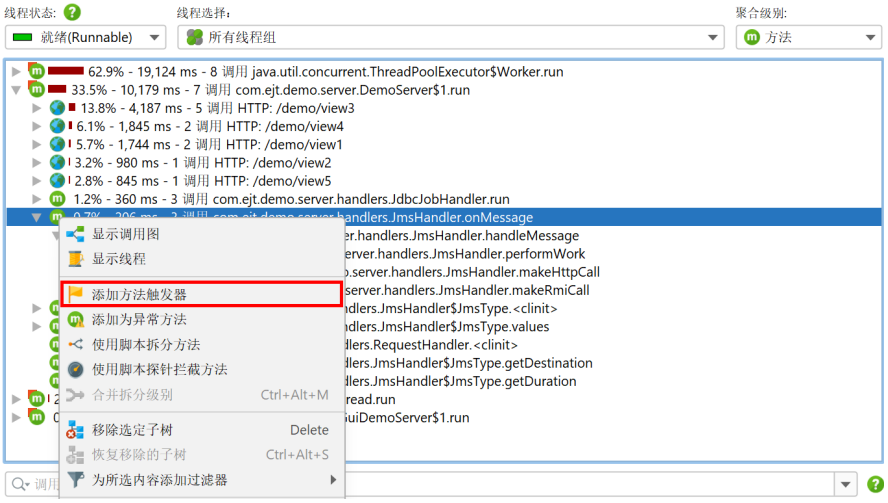
例如，您可能希望仅在执行特定方法时开始记录。在这种情况下，您可以转到会话设置对话框，激活触发器设置选项卡，并为该方法定义一个方法触发器。对于操作配置，您可以使用多种不同的记录操作。



“开始记录”操作控制这些记录，没有任何参数。通常，当您停止并重新启动记录时，所有先前记录的数据都会被清除。对于“CPU 数据”和“分配数据”记录，您还可以选择保留先前的数据并继续跨多个时间间隔累积。



可以通过在调用树中使用上下文菜单中的“添加方法触发器”操作方便地添加方法触发器。如果您在同一会话中已经有一个方法触发器，您可以选择将方法拦截添加到现有触发器。



默认情况下，触发器在 JVM 启动进行分析时处于活动状态。有两种方法可以在启动时禁用触发器：您可以在触发器配置中单独禁用它们，或者在会话启动对话框中取消选择 启动时启用触发器 复选框。在实时会话期间，您可以通过从菜单中选择 分析->(启用|禁用) 触发器 或单击状态栏中的 触发器记录状态图标来启用或禁用所有触发器。

有时，您需要同时切换触发器组的激活。这可以通过将相同的组ID分配给感兴趣的触发器并从菜单中调用 分析->启用触发器组 来实现。

使用 jpcontroller 记录

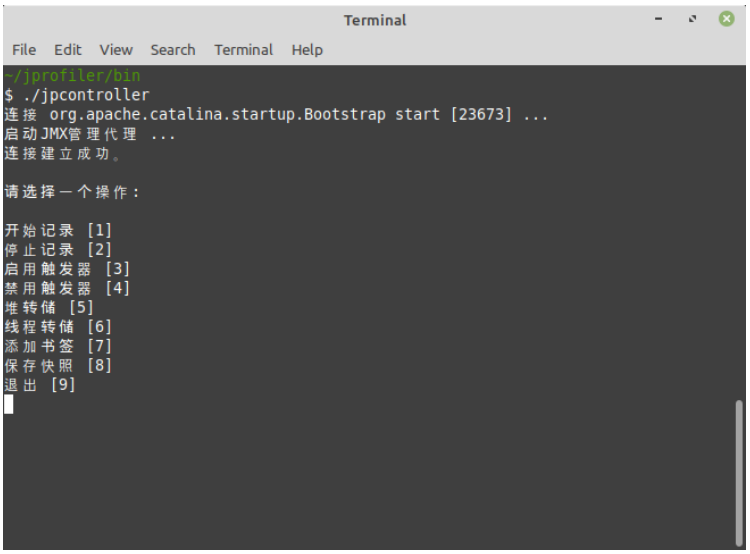
JProfiler 具有一个命令行可执行文件，用于控制任何已被分析的 JVM 中的记录。`jpcontroller` 要求 JProfiler MBean 已发布，否则它将无法连接到被分析的 JVM。只有在分析代理已经接收到分析设置的情况下才会出现这种情况。如果没有分析设置，代理将不知道要准确记录什么。

必须满足以下条件之一：

- 您已经使用 JProfiler GUI 连接到 JVM
- 被分析的 JVM 是使用包含 `nowait` 和 `config` 参数的 `-agentpath` VM 参数启动的。在集成向导中，这对应于 立即启动 模式和 启动时应用配置 选项在 配置同步 步骤中。
- JVM 已使用 `jpenable` 可执行文件准备进行分析，并指定了 `-offline` 参数。有关更多信息，请参见 `jpenable -help` 的输出。

具体来说，如果被分析的 JVM 仅使用 `nowait` 标志启动，则 `jpcontroller` 将无法工作。在集成向导中，连接 JProfiler GUI 时应用配置 选项在 配置同步 步骤中将配置此类参数。有关更多信息，请参见 有关在启动时设置分析设置的帮助主题 [p. 229]。

jpcontroller 为您提供了一个循环的多级菜单，用于所有记录及其参数。您还可以使用它保存快照。



以编程方式启动记录

另一种启动记录的方法是通过 API。在被分析的 VM 中，您可以调用 `com.jprofiler.api.controller.Controller` 类以编程方式启动和停止记录。有关更多信息以及如何获取包含控制器类的工件，请参见 离线分析章节 [p. 119]。

如果您想在不同的 JVM 中控制记录，您可以访问被分析的 JVM 中使用的相同 MBean，该 MBean 也由 `jpcontroller` 使用。设置 MBean 的编程使用相当复杂，需要相当多的步骤，因此 JProfiler 附带了一个您可以重用的示例。查看文件 `api/samples/mbean/src/MBeanProgrammaticAccessExample.java`。它在另一个被分析的 JVM 中记录 CPU 数据 5 秒钟，并将快照保存到磁盘。

快照

到目前为止，我们只看过实时会话，其中 JProfiler GUI 从正在被分析的 JVM 内运行的分析代理获取数据。JProfiler 还支持快照，其中所有分析数据都写入文件。这在几种情况下是有利的：

- 您自动记录分析数据，例如，作为测试的一部分，因此无法连接 JProfiler GUI。
- 您希望比较不同分析会话的数据或查看较早的记录。
- 您希望与其他人共享分析数据。

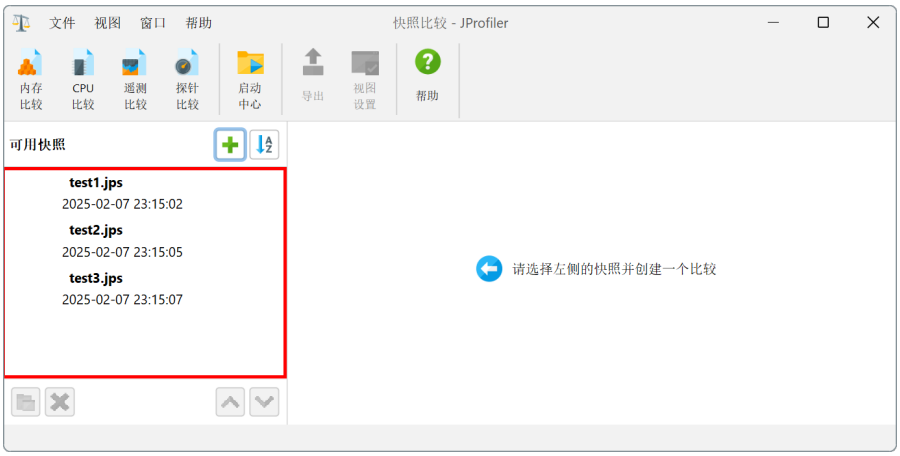
快照包括所有记录的数据，包括堆快照。为了节省磁盘空间，快照被压缩，但堆漫游器数据必须保持未压缩以允许直接内存映射。

在 JProfiler GUI 中保存和打开快照

当您分析实时会话时，可以使用 Save Snapshot 工具栏按钮创建快照。JProfiler 从远程代理提取所有分析数据并将其保存到具有 ".jps" 扩展名的本地文件中。您可以在实时会话过程中保存多个这样的快照。它们不会自动打开，您可以继续进行分析。

保存的快照会自动添加到 File->Recent Snapshots 菜单中，因此您可以方便地打开刚刚保存的快照。当实时会话仍在运行时打开快照时，您可以选择终止实时会话或打开另一个 JProfiler 窗口。

当您在 JProfiler 中使用快照比较功能时，快照列表会填充您为当前实时会话保存的所有快照。这使得比较不同的用例变得容易。

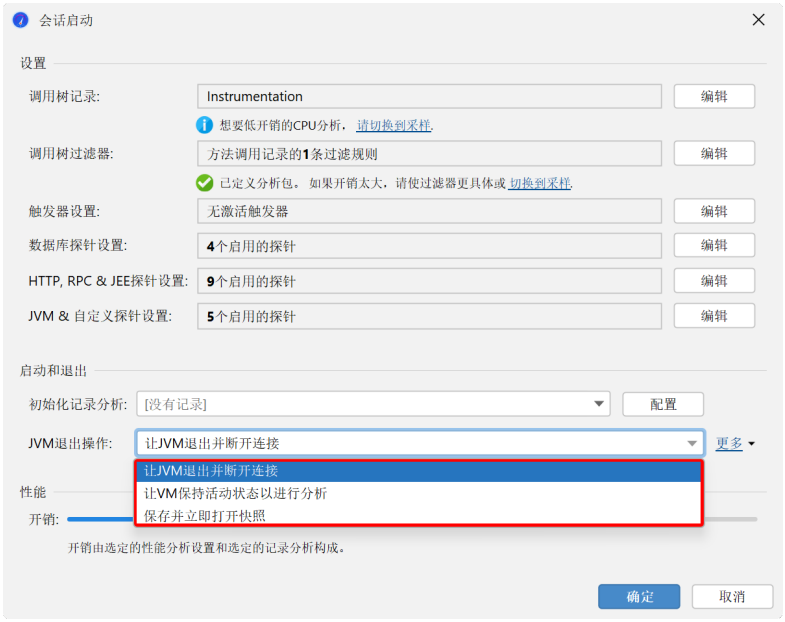


通常，您可以通过从主菜单中调用 Session->Open Snapshot 或在文件管理器中双击快照文件来打开快照。JProfiler 的 IDE 集成还支持通过 IDE 自身的通用 Open File 操作打开 JProfiler 快照。在这种情况下，您将获得进入 IDE 的源代码导航，而不是内置的源代码查看器。

当您打开快照时，所有记录操作都被禁用，并且只有记录了数据的视图可用。要发现记录了哪些数据，请将鼠标悬停在状态栏中的记录标签上。

分析短生命周期程序

对于实时会话，所有分析数据都驻留在被分析的 JVM 的进程中。因此，当被分析的 JVM 终止时，JProfiler 中的分析会话也会关闭。要在 JVM 退出时继续分析，您有两个选项，这两个选项都可以在会话启动对话框中激活。



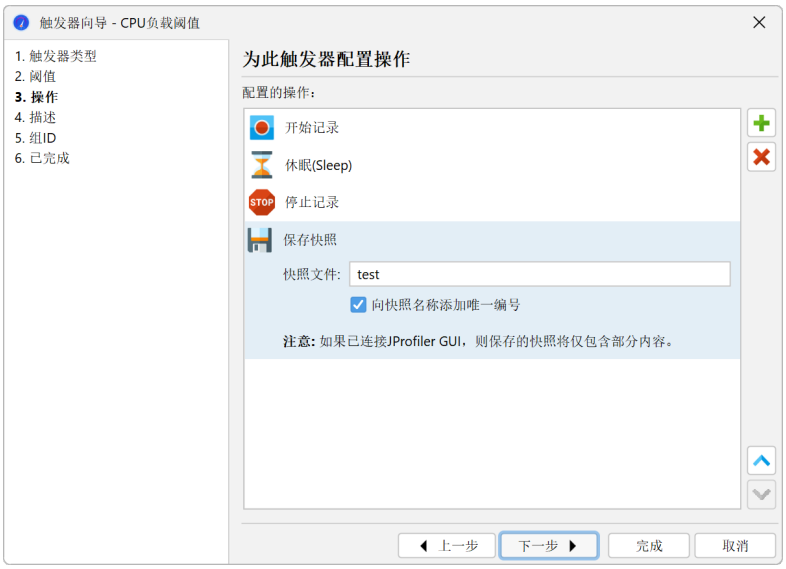
- 您可以防止 JVM 实际退出，并在 JProfiler GUI 连接的情况下保持其人为存活。这在您从 IDE 分析测试用例并希望在 IDE 的测试控制台中查看状态和总时间时可能是不希望的。
- 您可以要求 JProfiler 在 JVM 终止时保存快照并立即切换到它。快照是临时的，除非您首先使用 Save Snapshot 操作，否则将在您关闭会话时被丢弃。

使用触发器保存快照

自动分析会话的最终结果始终是一个快照或一系列快照。在触发器中，您可以添加一个“保存快照”操作，将快照保存到正在运行被分析 JVM 的机器上。当触发器在实时会话期间运行时，快照也会保存在远程机器上，并且可能不包括已经传输到 JProfiler GUI 的数据部分。

使用触发器保存快照有两种基本策略：

- 对于测试用例，在“JVM 启动”触发器中开始记录，并添加一个“JVM 退出”触发器以在 JVM 终止时保存快照。
- 对于异常条件，如“CPU 负载阈值”触发器或使用“计时器触发器”进行定期分析，在记录一些数据后保存快照，并在两者之间使用“睡眠”操作。

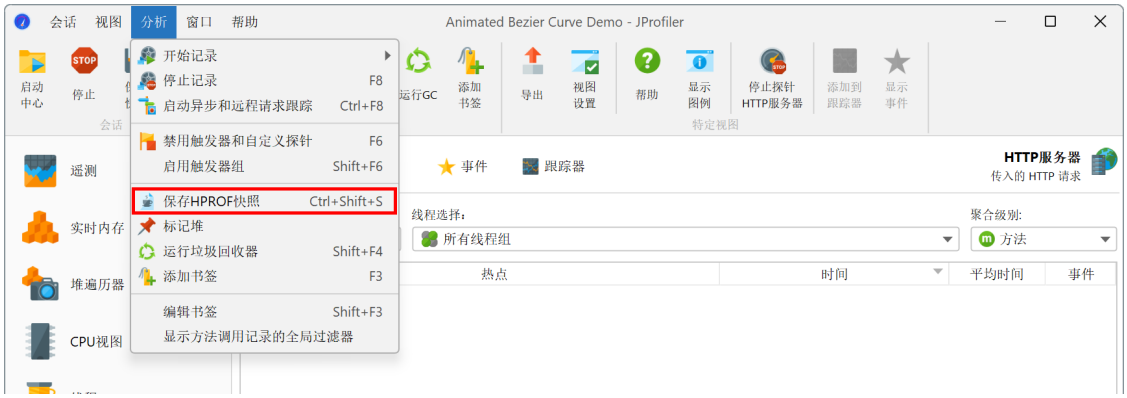


HPROF 堆快照

在堆快照产生过多开销或消耗过多内存的情况下，您可以使用 JVM 提供的 HPROF 堆快照作为内置功能。因为此操作不需要分析代理，所以这对于分析在生产中运行的 JVM 的内存问题很有趣。

使用 JProfiler，有三种方法可以获得这样的快照：

- 对于实时会话，JProfiler GUI 在主菜单中提供了一个操作来触发 HPROF 堆转储。

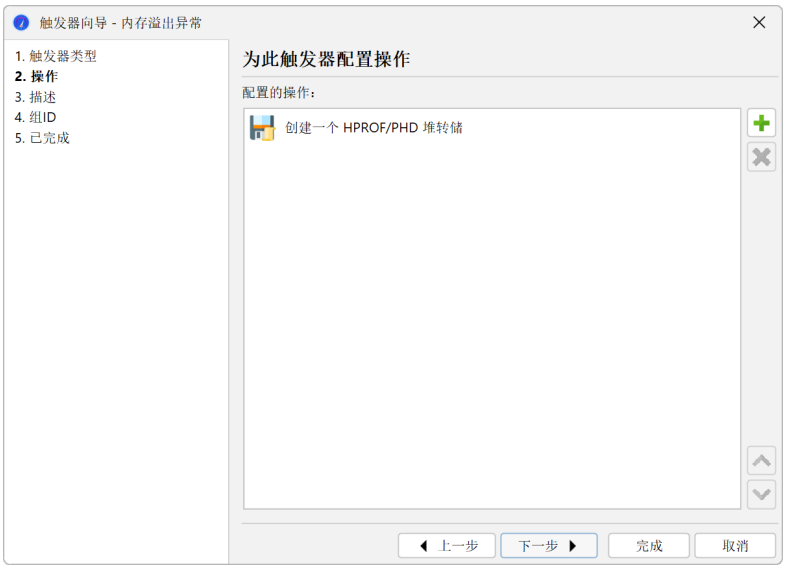


- JProfiler 有一个特殊的“内存不足异常”触发器，当抛出 `OutOfMemoryError` 时保存 HPROF 快照。这对应于 HotSpot JVM 支持的 VM 参数[(1)]

```
-XX:+HeapDumpOnOutOfMemoryError
```

。

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



- JDK 中的 jmap 可执行文件 [2] 可用于从正在运行的 JVM 中提取 HPROF 堆转储。

  JProfiler包含命令行工具`jpdump`，比jmap更通用。它允许您选择一个进程，可以连接到Windows上作为服务运行的进程，没有混合 32 位/64 位 JVM 的问题，并自动编号 HPROF 快照文件。使用`-help`选项执行它以获取更多信息。

JDK Flight Recorder 快照

JProfile 完全支持打开由 Java Flight Recorder (JFR) 保存的快照。在这种情况下，UI 显著不同，并根据 JFR 的功能进行了调整。有关更多详细信息，请参阅 JFR 帮助主题 [p. 205]。

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

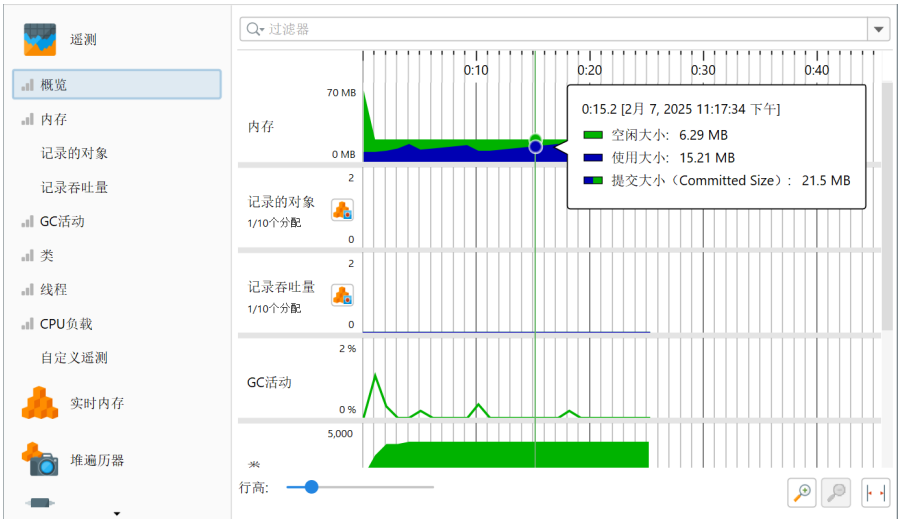
遥测 (Telemetries)

分析的一个方面是监控随时间变化的标量测量，例如，使用的堆大小。在JProfiler中，这样的图形被称为遥测。观察遥测可以让您更好地理解被分析 (profiled) 的软件， 允许您在不同测量之间关联重要事件，并可能在您注意到意外行为时提示您使用JProfiler中的其他视图进行更深入的分析。

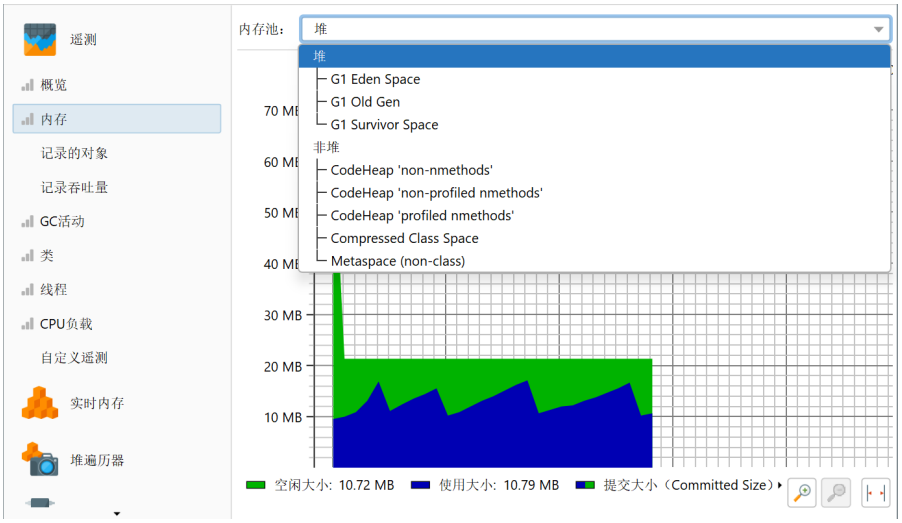
标准遥测 (Standard telemetries)

在JProfiler UI的“遥测”部分，默认情况下会记录一些遥测。 对于交互会话，它们始终是启用的。某些遥测需要记录特殊类型的数据。 在这种情况下，遥测中将显示一个记录操作。

要在相同时间轴上比较多个遥测，概览会在彼此之上显示多个小规模遥测，并具有可配置的行高。点击遥测标题会激活完整的遥测视图。概览中遥测的默认顺序可能不合适，例如，因为您希望将选定的遥测并排关联。 在这种情况下，您可以在概览中通过拖放重新排序它们。

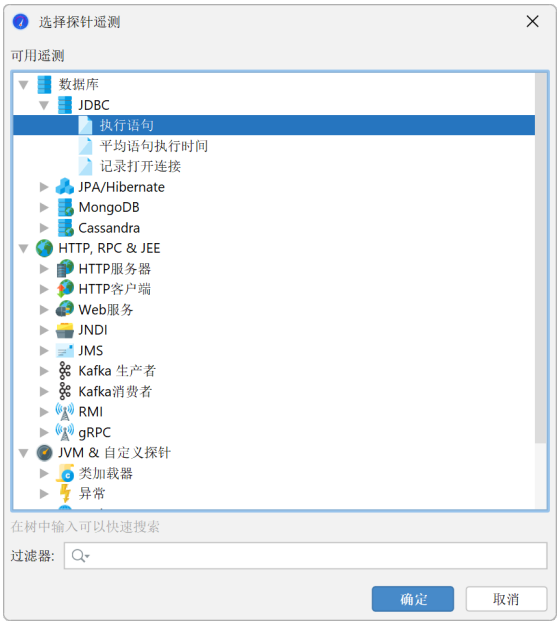


完整视图显示当前值的图例，并可能具有比概览中可见的更多选项。 例如， “Memory”遥测允许您选择单个内存池。

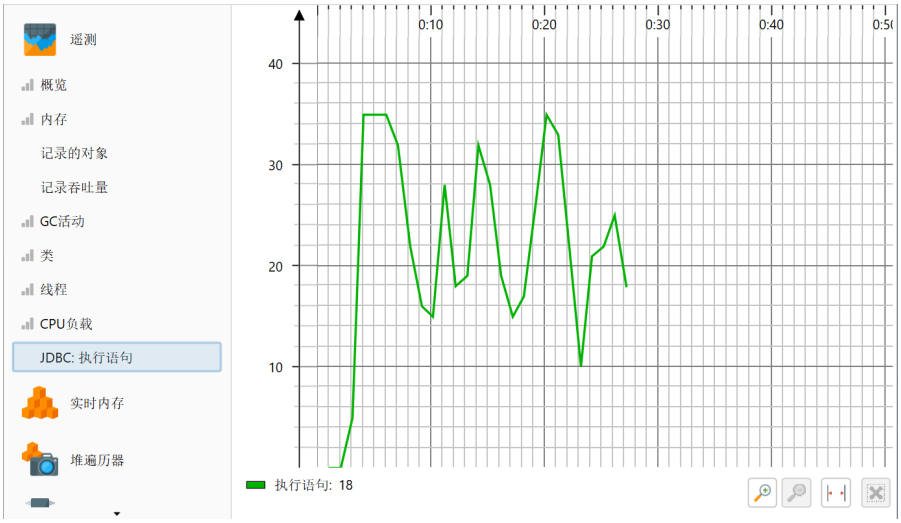


JProfiler有大量的探针 (probes) [p. 96]，记录JVM中的高级系统和重要框架的事件。 探针具有在相应探针视图中显示的遥测。要将这些遥测与系统遥测进行比较，您可以将选定的探针遥测添加到顶级

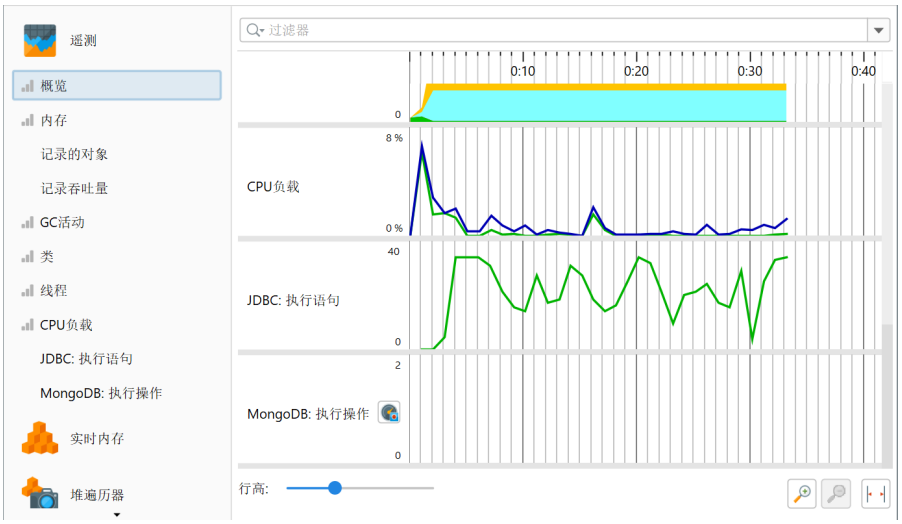
遥测部分。 从工具栏中选择 ✚ 添加遥测 (Add telemetries)->探针遥测 (Probe Telemetry) 并选择一个或多个探针遥测。



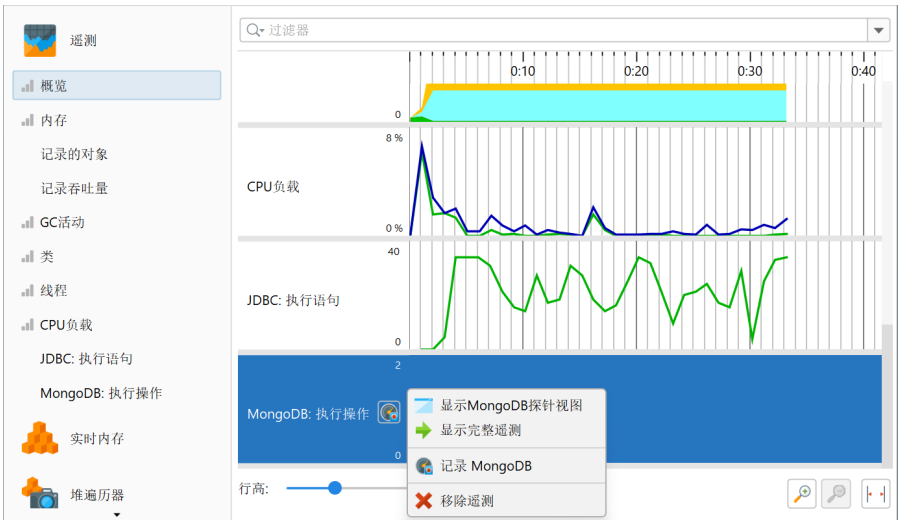
每个添加的探针遥测在遥测部分都有自己的视图，并且也显示在概览中。



一旦添加了探针遥测，只有在记录了探针数据时才会显示数据。如果没有，遥测描述中包含一个内联按钮以开始记录。

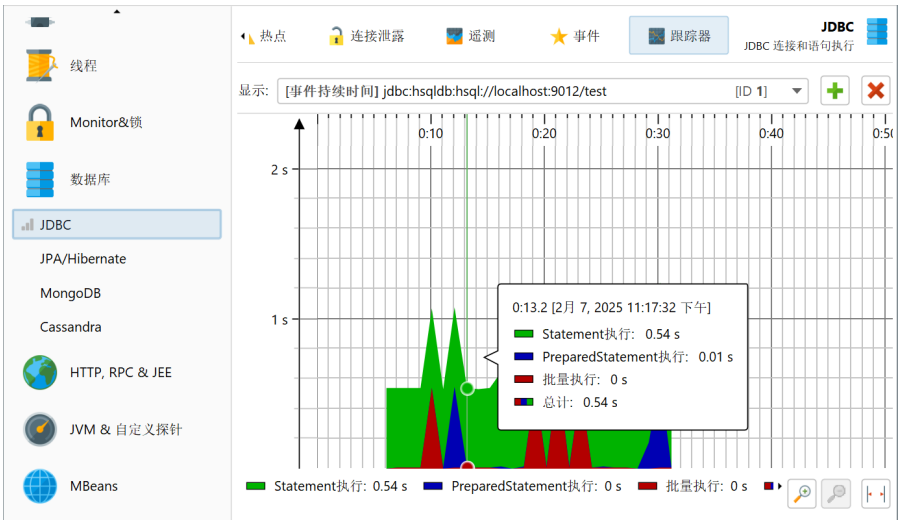


探针遥测的上下文菜单包含记录操作以及显示相应探针视图的操作。



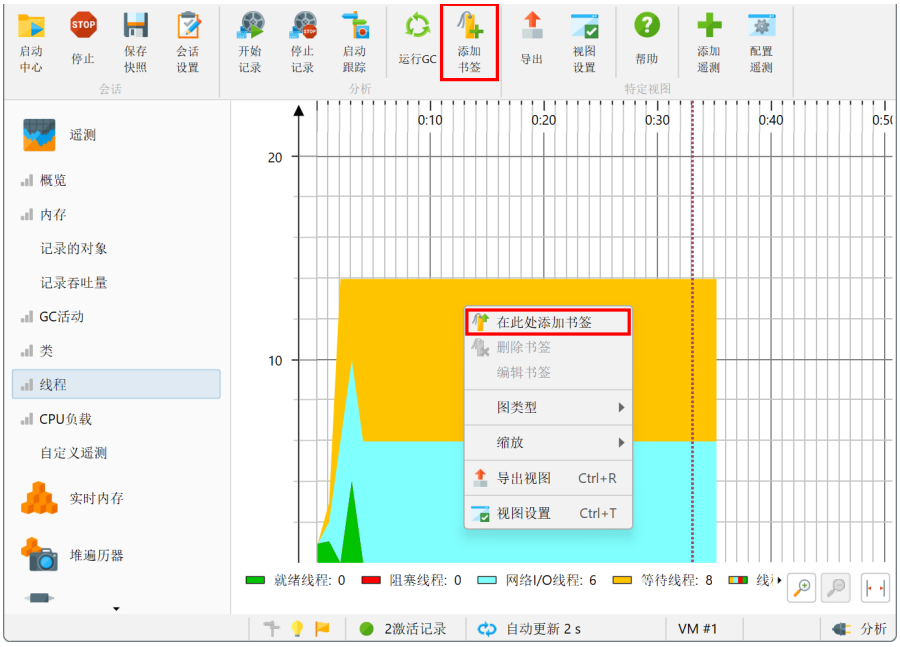
类似于探针视图，记录对象的VM遥测依赖于内存记录，并且也有一个记录按钮和类似的上下文菜单。

最后，还有“跟踪”遥测，用于监控在另一个视图中选择的标量值。 例如，类追踪器 (class tracker) 视图允许您选择一个类并监控其实例数量随时间的变化。此外，每个探针都有一个“追踪器(Tracker)”视图，其中监控选定的热点 (hot spots) 或控制对象 (control objects)。



书签 (Bookmarks)

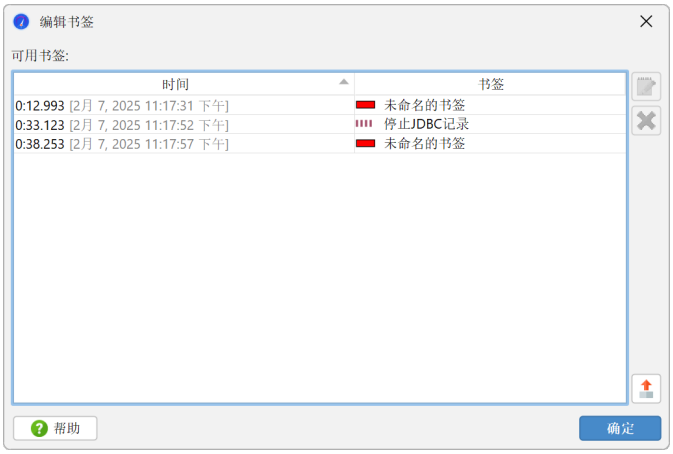
JProfiler维护一个在所有遥测中显示的书签列表。在交互会话中，您可以通过点击添加书签 (Add Bookmark)工具栏按钮， 或使用上下文菜单中的在此添加书签 (Add Bookmark Here)功能，在当前时间添加书签。



书签不仅可以手动创建，它们还会由记录操作自动添加，以指示特定记录的开始和结束。通过触发器 (trigger) 操作或控制器API，您可以以编程方式添加书签。

书签具有颜色、线条样式以及在工具提示中显示的名称。您可以编辑现有书签并更改这些属性。

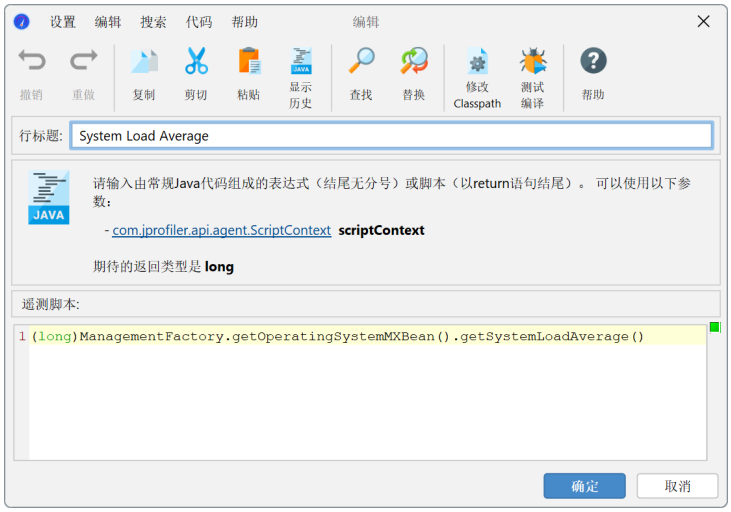
如果在遥测中右键点击多个书签太不方便，您可以使用菜单中的分析 (Profiling)->编辑书签 (Edit Bookmarks)操作获取书签列表。 这也是您可以将书签导出为HTML或CSV的地方。



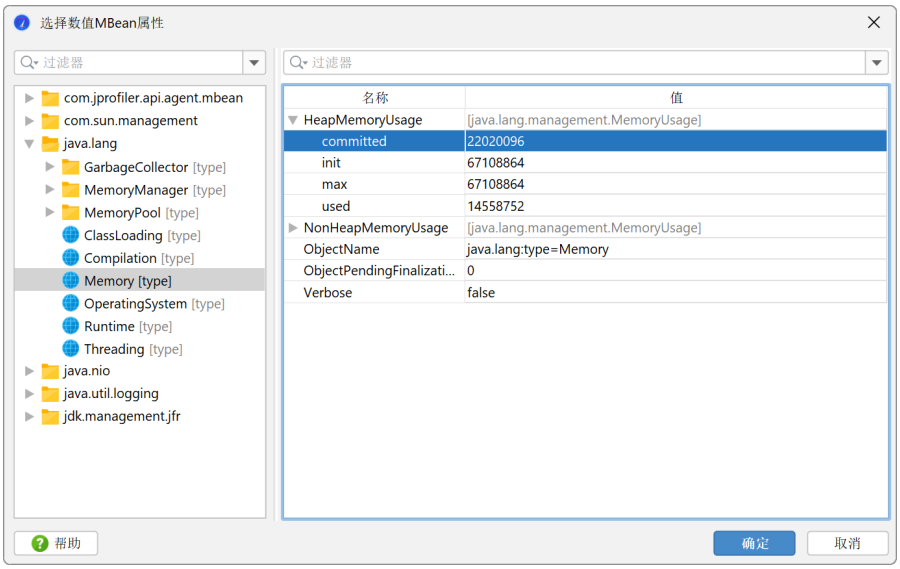
自定义遥测 (Custom telemetries)

有两种方法可以添加您自己的遥测：要么在JProfiler UI中编写脚本以提供数值，要么选择一个数值的MBean属性。

要添加自定义遥测，请点击“遥测”部分中可见的配置遥测 (Configure Telemetries)工具栏按钮。在脚本遥测中，您可以访问当前JProfiler会话的类路径中配置的所有类。如果值不能直接获得，请向您的应用程序添加一个静态方法，您可以在此脚本中调用。

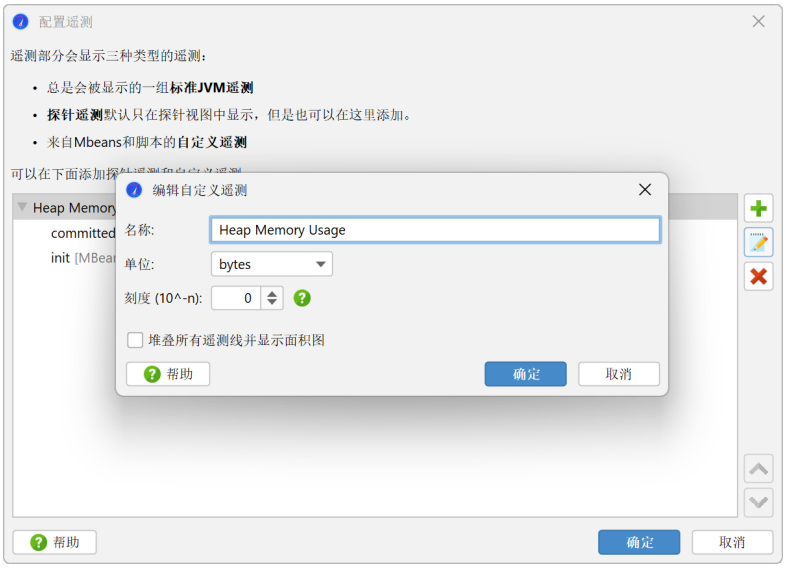


上面的示例显示了对平台MBean的调用。绘制MBean的标量值更方便地通过MBean遥测完成。 在这里，MBean浏览器允许您选择合适的属性。属性值必须是数值。

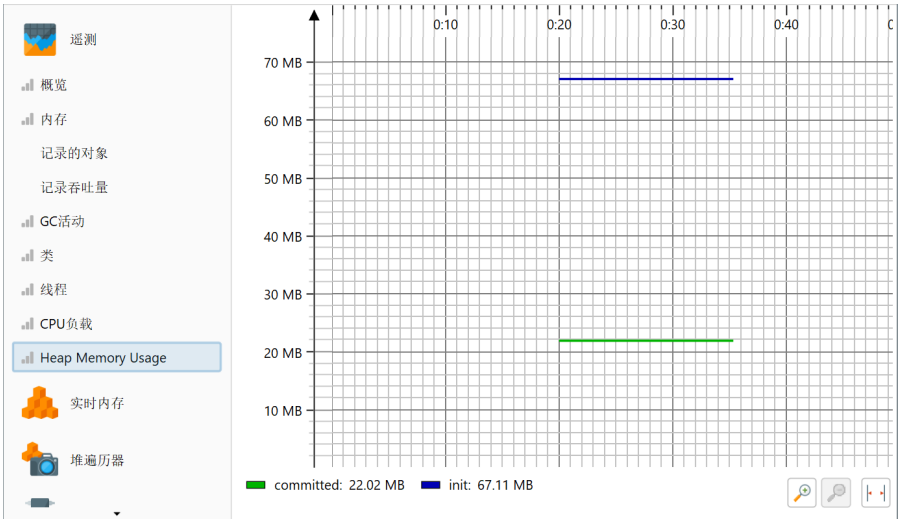


您可以将多个遥测线捆绑成一个遥测。这就是为什么配置分为两部分：遥测本身和遥测线。在遥测线中，您只需编辑数据源和线条标题，在遥测中，您可以配置单位、比例和堆叠，这些适用于所有包含的线条。

在堆叠遥测中，单个遥测线是累加的，并且可以显示区域图。比例因子用于将值转换为支持的单位。例如，如果数据源报告kB，问题是JProfiler中没有匹配的“kB”单位。 如果将比例因子设置为-3，值将被转换为字节，并通过选择“字节(bytes)”作为遥测的单位，JProfiler将自动在遥测中显示适当的聚合单位。



自定义遥测按配置顺序添加到“遥测”部分的末尾。要重新排序它们，请在概览中将它们拖动到所需位置。



开销考虑 (Overhead considerations)

乍一看，似乎遥测会随着时间线性消耗内存。然而，JProfiler会合并较旧的值，并使它们逐渐变得更粗粒度，以限制每个遥测消耗的总内存量。

遥测的CPU开销受到限制，因为它们的值每秒只轮询一次。 对于标准遥测，此数据收集没有额外的开销。 对于自定义遥测，开销由底层脚本或MBean决定。

CPU Profiling

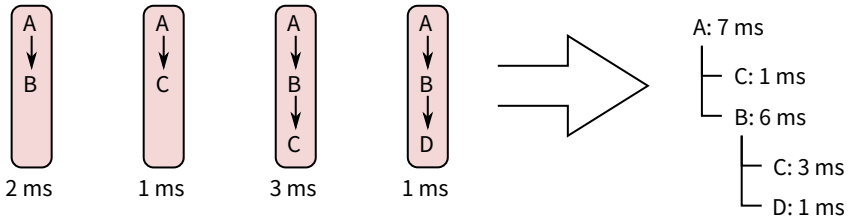
当 JProfiler 测量方法调用的执行时间及其调用栈时，我们称之为“CPU Profiling”。这些数据以多种方式呈现。根据您要解决的问题，某种展示方式可能会更有帮助。默认情况下，CPU数据不会被记录，您必须 开启 CPU 记录 [p. 27] 以捕获有趣的用例。

调用树

跟踪所有方法调用及其调用栈会消耗大量内存，并且只能维持很短的时间，直到所有内存耗尽。此外，直观地理解繁忙 JVM 中的方法调用数量也不容易。通常，这个数字大到无法定位和跟踪踪迹。

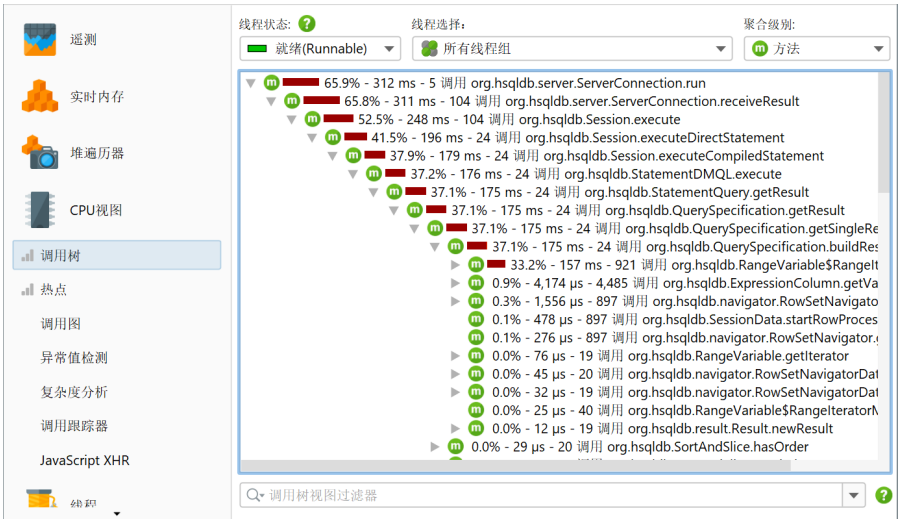
另一个方面是，许多性能问题只有在数据被聚合后才会变得清晰。通过这种方式，您可以判断在特定时间段内方法调用相对于整个活动的重要性。单个踪迹中，您无法了解所查看数据的相对重要性。

这就是为什么JProfiler构建了一个累积的调用栈树，并注释了观察到的时间和调用次数。时间顺序被消除，只保留总数。树中的每个节点代表至少观察到一次的调用栈。节点有子节点，代表在该调用栈中看到的所有外部调用。

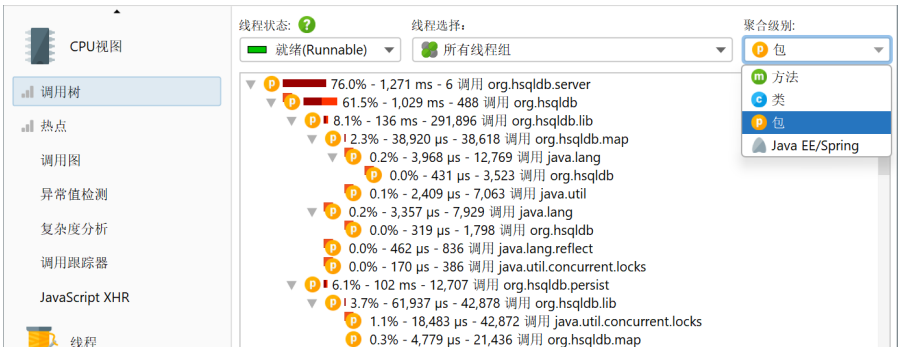


方法调用与调用栈 **调用树**

调用树是“CPU 视图”部分中的第一个视图，当您开始 CPU Profiling 时，这是一个很好的起点，因为从起始点到最细粒度终点的方法调用的自上而下视图最容易理解。JProfiler按总时间对子节点进行排序，因此您可以深度优先打开树以分析对性能影响最大的部分。



虽然所有测量都是针对方法进行的，JProfiler允许您通过在类或包级别上聚合调用树来获得更广泛的视角。聚合级别选择器还包含一个“JEE/Spring组件”模式。如果您的应用程序使用JEE或Spring，您可以使用此模式在类级别上仅查看 JEE 和 Spring 组件。像 URL 这样的拆分节点在所有聚合级别中都保留。

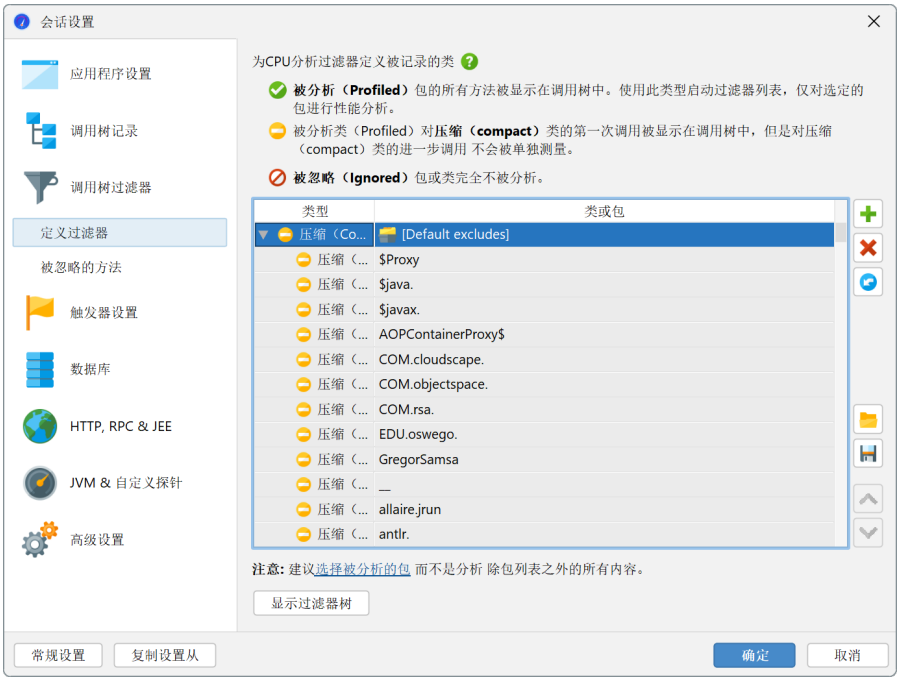


调用树过滤器

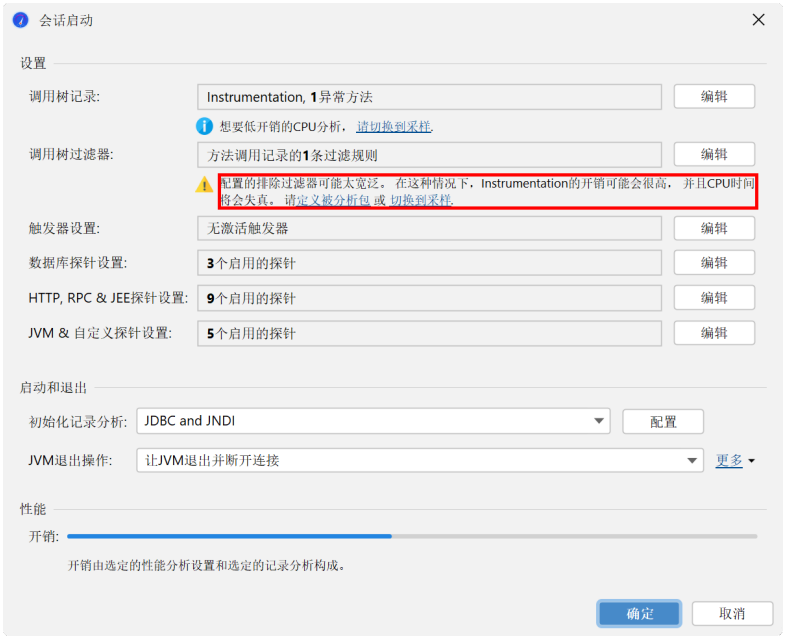
如果调用树中显示了所有类的方法，树通常太深而无法管理。如果您的应用程序由框架调用，调用树的顶部将由您不关心的框架类组成，而您自己的类将深埋其中。调用库将显示其内部结构，可能有数百个您不熟悉且无法影响的方法调用级别。

解决此问题的方法是对调用树应用过滤器，以便仅记录某些类。作为一个积极的副作用，收集的数据更少，必须被检测的类更少，因此开销减少。

默认情况下，分析会话配置了一个排除常用框架和库包的列表。



当然，这个列表是不完整的，所以最好删除它并自己定义感兴趣的包。事实上，检测[p. 63]和默认过滤器的组合是如此不理想，以至于 JProfiler 建议在会话启动对话框中更改它。



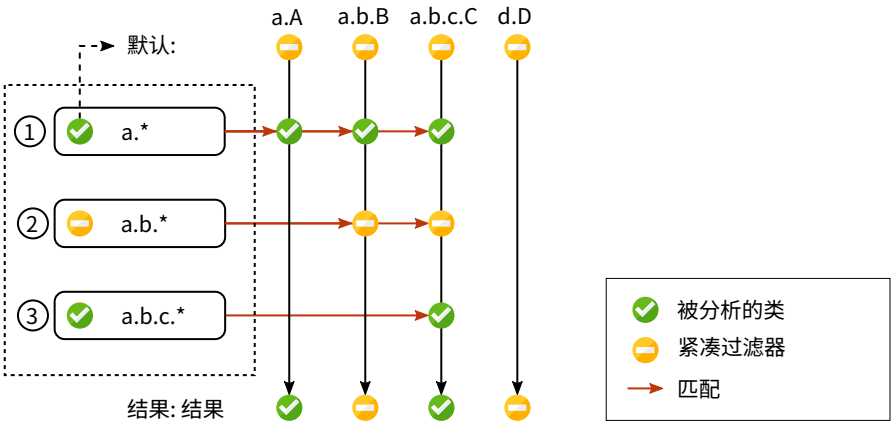
过滤器表达式与完全限定类名进行比较，因此 `com.mycorp.` 匹配所有嵌套包中的类，例如 `com.mycorp.myapp.Application`。有三种类型的过滤器，称为“被分析”、“紧凑”和“忽略”。“被分析”类中的所有方法都被测量。这是您自己的代码所需的。

在“紧凑”过滤器包含的类中，仅测量对该类的第一次调用，但不显示进一步的内部调用。“紧凑”是您对库（包括 JRE）所需要的。例如，当调用 `hashMap.put(a, b)` 时，您可能希望在调用树中看到 `HashMap.put()`，但不希望看到更多——除非您是映射实现的开发者，否则其内部工作应被视为不透明。

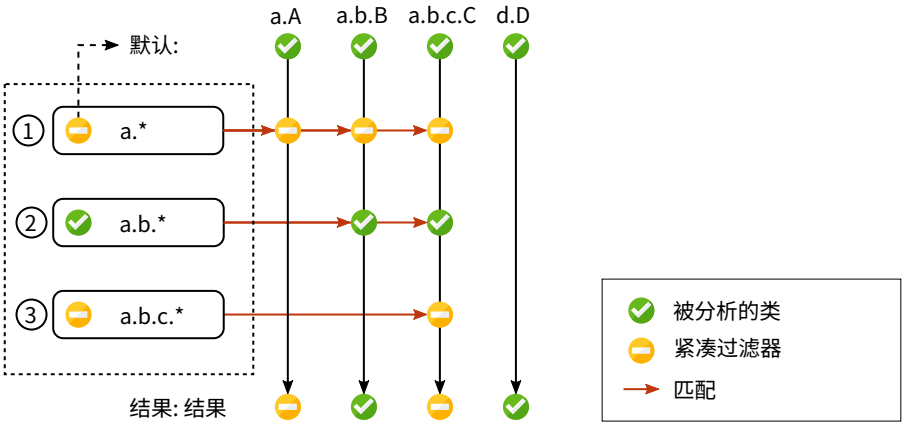
最后，“忽略”方法根本不被分析。由于开销考虑，它们可能不适合检测，或者它们可能在调用树中只是分散注意力，例如动态调用之间插入的内部 Groovy 方法。

手动输入包容易出错，因此您可以使用包浏览器。在您启动会话之前，包浏览器只能显示配置类路径中的包，这通常不包括所有实际加载的类。在运行时，包浏览器将显示所有已加载的类。

配置的过滤器列表从上到下对每个类进行评估。在每个阶段，如果有匹配项，当前过滤器类型可能会更改。重要的是过滤器列表的起始类型。如果您以“被分析”过滤器开始，类的初始过滤器类型是“紧凑”，这意味着只有显式匹配项被分析。



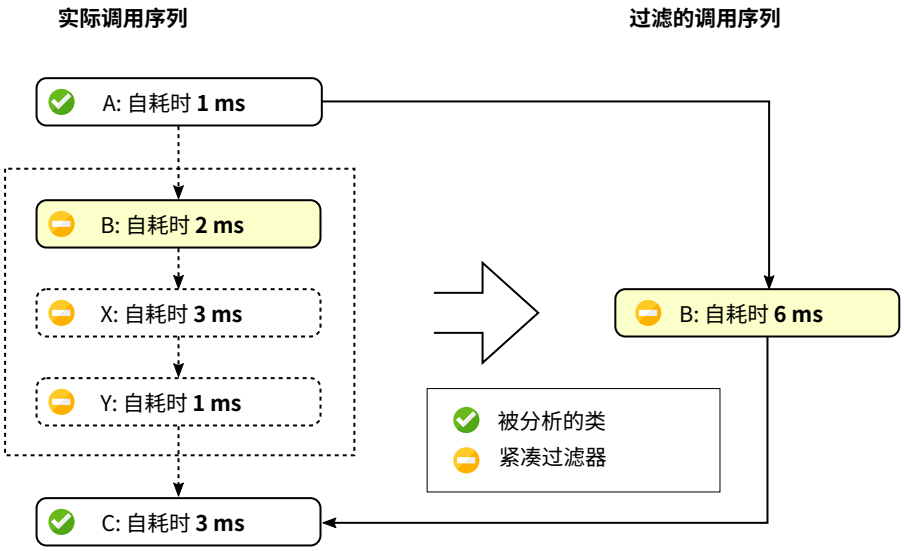
如果您以“紧凑”过滤器开始，类的初始过滤器类型是“被分析”。在这种情况下，除显式排除的类外，所有类都被分析。



调用树时间

为了正确解释调用树，理解显示在调用树节点上的数字非常重要。对于任何节点，有两个时间是有趣的，总时间和自耗时。自耗时是节点的总时间减去嵌套节点的总时间。

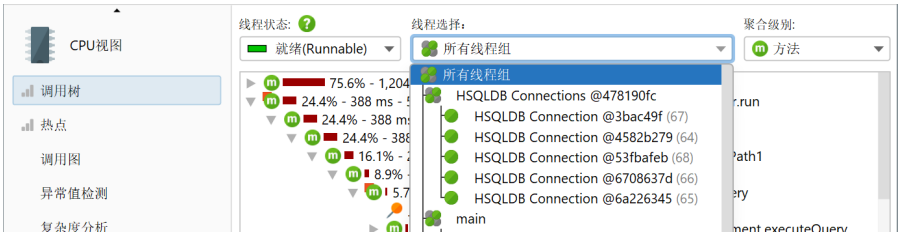
通常，自耗时很小，除了紧凑过滤的类。大多数情况下，紧凑过滤的类是叶节点，总时间等于自耗时，因为没有子节点。有时，紧凑过滤的类会调用被分析的类，例如通过回调或因为它是调用树的入口点，如当前线程的 `run` 方法。在这种情况下，一些未分析的方法消耗了时间，但在调用树中未显示。该时间会冒泡到调用树中第一个可用的祖先节点，并贡献给紧凑过滤类的自耗时。



调用树中的百分比条显示总时间，但自耗时部分以不同颜色显示。除非同一级别的两个方法被重载，否则方法显示时不带签名。在视图设置对话框中，有多种方法可以自定义调用树节点的显示。例如，您可能希望将自耗时或平均时间显示为文本，始终显示方法签名或更改使用的时间尺度。此外，百分比计算可以基于父时间而不是整个调用树的时间。

线程状态

在调用树的顶部，有几个视图参数可以更改显示的分析数据的类型和范围。默认情况下，所有线程都是累积的。JProfiler 按线程维护 CPU 数据，您可以显示单个线程或线程组。



在任何时候，每个线程都有一个关联的线程状态。如果线程准备好处理字节码指令或当前正在 CPU 核心上执行它们，则线程状态称为“Runnable”。在寻找性能瓶颈时，该线程状态是感兴趣的，因此默认选择它。

或者，线程可能正在等待监视器，例如，通过调用 `Object.wait()` 或 `Thread.sleep()`，在这种情况下，线程状态称为“Waiting”。尝试获取监视器时被阻塞的线程，例如在`synchronized`代码块的边界处，则处于“Blocking”状态。

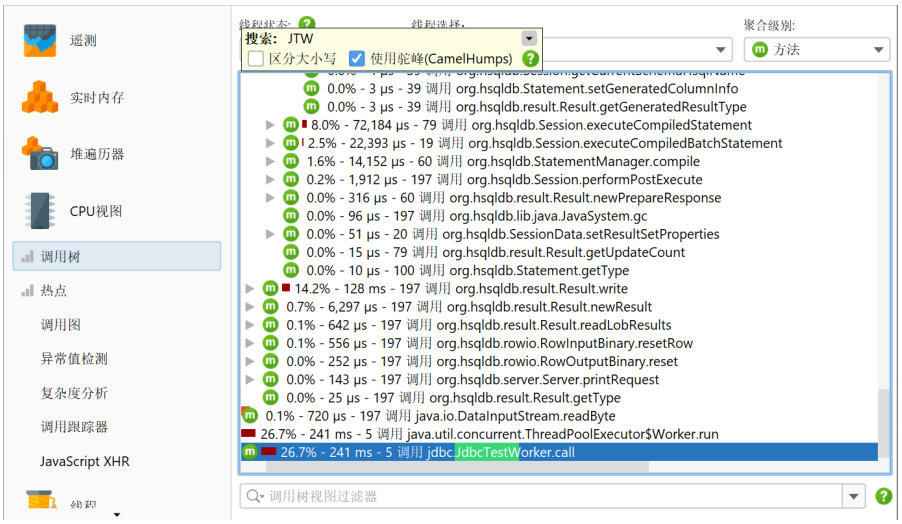
最后，JProfiler 添加了一个合成的“Net I/O”状态，用于跟踪线程等待网络数据的时间。这对于分析服务器和数据库驱动程序很重要，因为该时间可能与性能分析相关，例如调查慢速 SQL 查询。

如果您对挂钟时间感兴趣，您必须选择线程状态“所有状态”并选择单个线程。只有这样，您才能将时间与您在代码中通过调用 `System.currentTimeMillis()` 计算的持续时间进行比较。

如果您想将选定的方法移到不同的线程状态，可以使用方法触发器和“覆盖线程状态”触发器操作，或者使用 `ThreadStatus` 类在 嵌入式 [p. 154] 或 注入式 [p. 149] 探针 API 中。

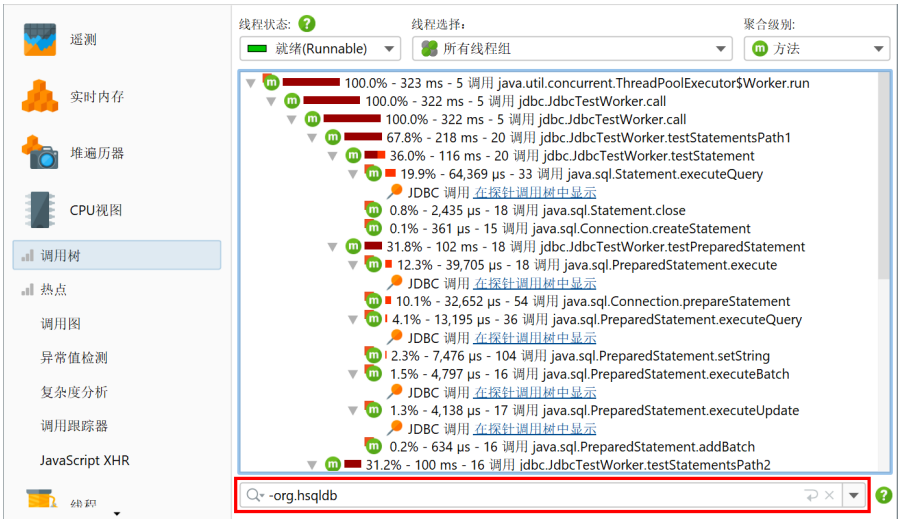
在调用树中查找节点

有两种方法可以在调用树中搜索文本。首先，有快速搜索选项，可以通过从菜单中调用 视图->查找 或直接开始在调用树中键入来激活。匹配项将被高亮显示，按下`PageDown`后可以使用搜索选项。使用 `ArrowUp` 和 `ArrowDown` 键可以循环浏览不同的匹配项。



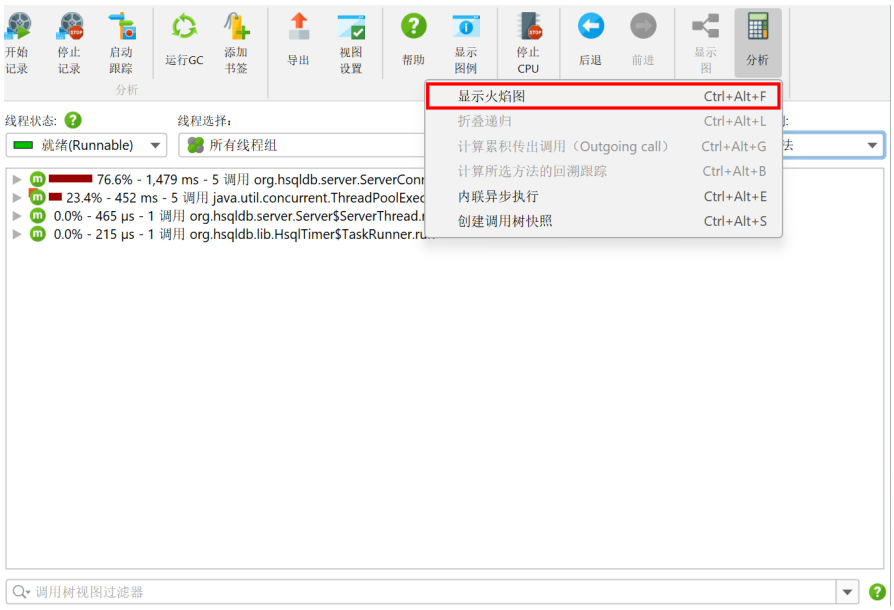
另一种搜索方法、类或包的方法是使用调用树底部的视图过滤器。在这里，您可以输入以逗号分隔的过滤器表达式列表。以“-”开头的过滤器表达式类似于忽略过滤器。以“!”开头的表达式类似于紧凑过滤器。所有其他表达式类似于被分析过滤器。就像过滤器设置一样，初始过滤器类型决定类是默认包含还是排除。

单击视图设置文本字段左侧的图标会显示视图过滤器选项。默认情况下，匹配模式是“包含”，但在搜索特定包时，“以...开头”可能更合适。

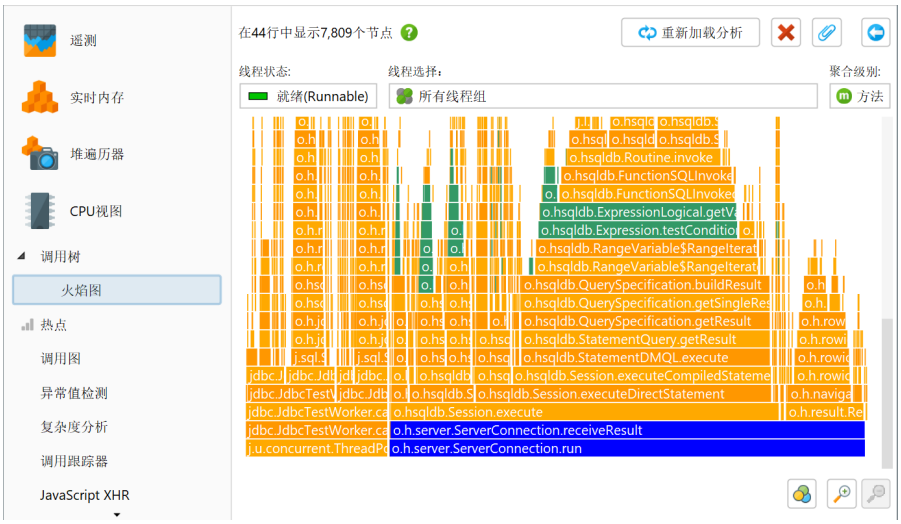


火焰图

查看调用树的另一种方法是作为火焰图。您可以通过调用关联的 调用树分析 [p. 175]，将整个调用树或其一部分显示为火焰图。



火焰图在一张图像中显示调用树的全部内容。调用从火焰图的底部开始，并向顶部传播。每个节点的子节点排列在其正上方的行中。子节点按字母顺序排序，并以其父节点为中心。由于每个节点中花费的自耗时，“火焰”向顶部逐渐变窄。有关节点的更多信息显示在工具提示中，您可以在其中标记文本以将其复制到剪贴板。



如果鼠标光标附近的工具提示干扰了您的分析，您可以使用其右上角的按钮锁定它，然后使用工具提示顶部的握柄将其移动到方便的位置。相同的按钮或双击火焰图关闭工具提示。

火焰图的信息密度非常高，因此可能需要通过关注选定节点及其后代节点的层次结构来缩小显示的内容。虽然您可以放大感兴趣的区域，但也可以通过双击或使用上下文菜单设置新的根节点。当连续多次更改根节点时，您可以在根节点历史中再次向后移动。

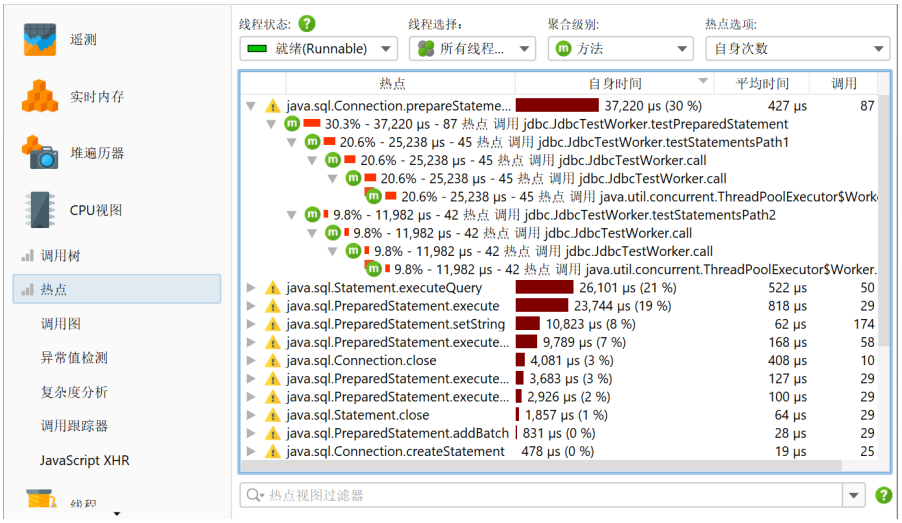
分析火焰图的另一种方法是根据类名、包名或任意搜索词添加着色。可以从上下文菜单中添加着色，并可以在着色对话框中进行管理。对于每个节点，使用第一个匹配的着色。着色在分析会话之间持久化，并在所有会话和快照中全局使用。

除了着色，您还可以使用快速搜索功能查找感兴趣的节点。使用光标键可以在匹配结果之间循环，同时为当前高亮显示的匹配项显示工具提示。

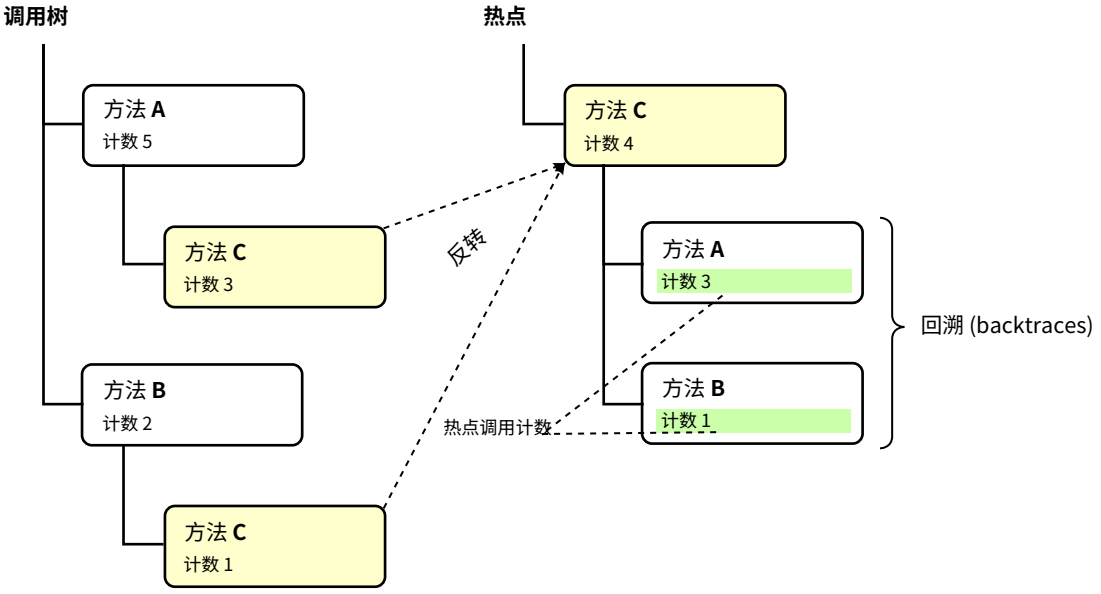
热点

如果您的应用程序运行得太慢，您会想找到占用大部分时间的方法。使用调用树，有时可以直接找到这些方法，但通常这不起作用，因为调用树可能很宽，叶节点数量巨大。

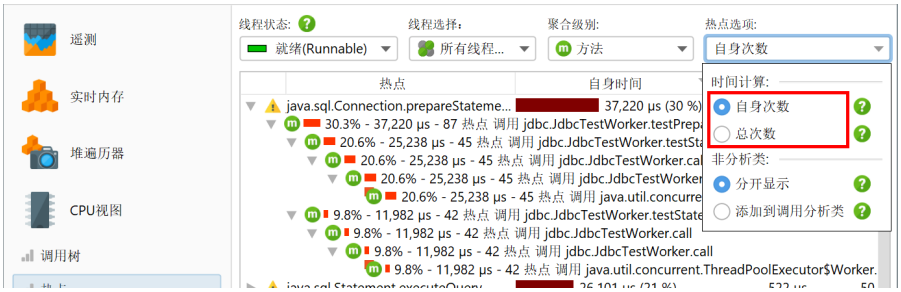
在这种情况下，您需要调用树的逆：按总自耗时排序的所有方法列表，从所有不同的调用栈累积，并显示方法的调用方式。在热点树中，叶子是入口点，如应用程序的 `main` 方法或线程的 `run` 方法。从热点树中最深的节点，调用向上传播到顶级节点。



回溯中的调用次数和执行时间不指代方法节点，而是指沿此路径调用顶级热点节点的次数。这一点很重要：乍一看，您可能会期望节点上的信息量化对该节点的调用。然而，在热点树中，该信息显示了节点对顶级节点的**贡献**。因此，您必须这样阅读数字：沿着这个反向调用栈，顶级热点被调用了 n 次，总持续时间为 t 秒。



默认情况下，热点是根据自耗时计算的。您也可以根据总时间计算它们。这对于分析性能瓶颈不是很有用，但如果您想查看所有方法的列表可能会很有趣。热点视图仅显示最大数量的方法以减少开销，因此您要查找的方法可能根本不显示。在这种情况下，使用底部的视图过滤器过滤包或类。与调用树相反，热点视图过滤器仅过滤顶级节点。热点视图中的截止点不是全局应用的，而是相对于显示的类应用的，因此应用过滤器后可能会出现新节点。

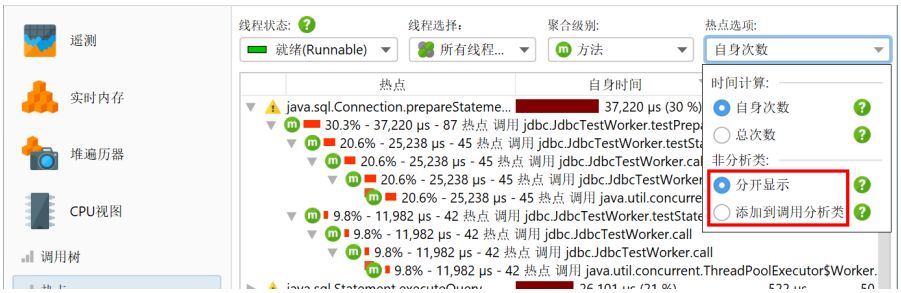


热点和过滤器

热点的概念不是绝对的，而是取决于调用树过滤器。如果您没有任何调用树过滤器，最大的热点很可能总是 JRE 核心类中的方法，如字符串操作、I/O 例程或集合操作。这样的热点不会很有用，因为您通常无法直接控制这些方法的调用，也无法加速它们。

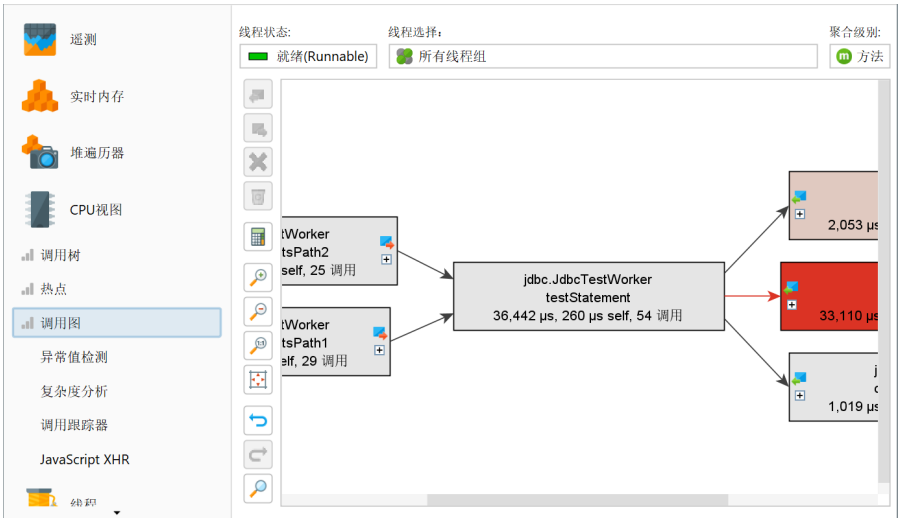
为了对您有用，热点必须是您自己类中的方法或您直接调用的库类中的方法。在调用树过滤器方面，您自己的类在“被分析”过滤器中，库类在“紧凑”过滤器中。

在解决性能问题时，您可能希望消除库层，只查看您自己的类。您可以通过在热点选项弹出窗口中选择 添加到调用的被分析类 单选按钮快速切换到该视角。



调用图

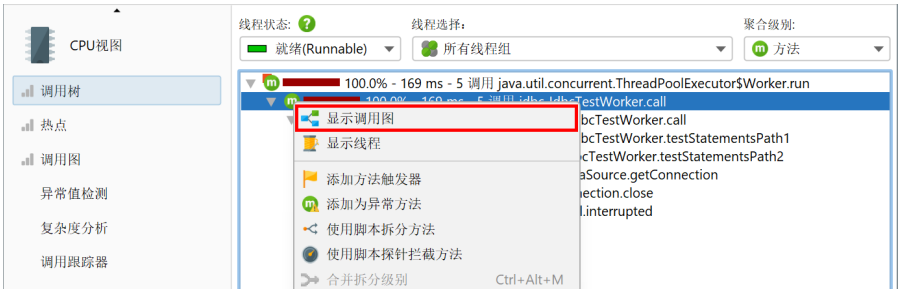
在调用树和热点视图中，每个节点可能会多次出现，尤其是在递归调用时。在某些情况下，您对方法中心的统计感兴趣，其中每个方法只出现一次，所有传入和传出调用都可见。这样的视图最好显示为图形，在 JProfiler 中，它被称为调用图。



图形的一个缺点是其视觉密度低于树。这就是为什么 JProfiler 默认缩写包名并默认隐藏总时间少于 1% 的传出调用。只要节点有传出扩展图标，您可以再次单击它以显示所有调用。在视图设置中，您可以配置此阈值并关闭包缩写。

扩展调用图时，它可能会很快变得混乱，尤其是如果您多次回溯。使用撤销功能恢复图形的先前状态。就像调用树一样，调用图提供快速搜索。通过在图形中键入，您可以开始搜索。

图形和树视图各有优缺点，因此有时您可能希望从一种视图类型切换到另一种。在交互式会话中，调用树和热点视图显示实时数据并定期更新。然而，调用图是在请求时计算的，并且在您扩展节点时不会更改。调用树中的 在调用图中显示 操作计算新的调用图并显示选定的方法。



从图形切换到调用树是不可能的，因为数据通常在稍后时间不再可比。然而，调用图提供调用树分析，其 视图->分析 操作可以为您显示累积传出调用和每个选定节点的回溯树。

超越基础

调用树、热点视图和调用图的组合具有许多高级功能，这些功能在 不同章节 [p. 158] 中详细解释。此外，还有其他高级 CPU 视图在 单独 [p. 180] 介绍。

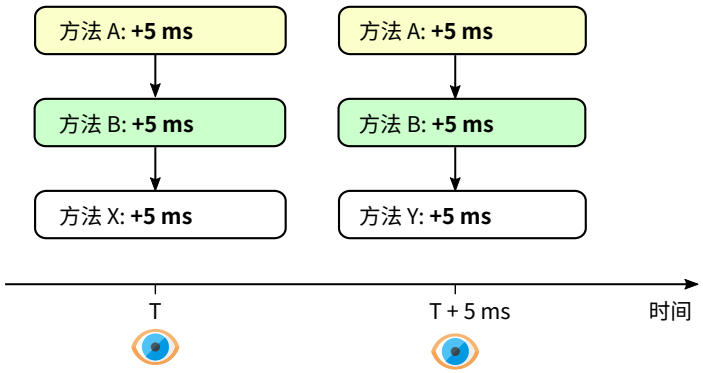
方法调用记录

记录方法调用是分析器最困难的任务之一，因为它在相互冲突的约束下运行：结果应该准确、完整，并产生如此小的开销，以至于您从测量数据中得出的结论不会变得不正确。不幸的是，没有一种测量类型可以满足所有类型应用程序的所有这些要求。这就是为什么JProfiler要求您决定使用哪种方法。

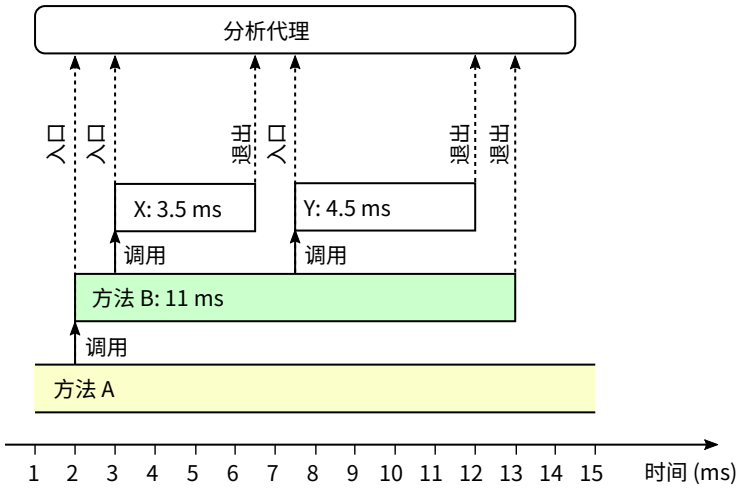
采样与插桩

方法调用的测量可以通过两种根本不同的技术完成，称为“采样”和“插桩”，每种技术都有其优点和缺点：通过采样，线程的当前调用栈会定期检查。通过插桩，选定类的字节码被修改以跟踪方法的进入和退出。插桩测量所有调用，并且可以为所有方法生成调用计数。

在处理采样数据时，完整的采样周期（通常为5毫秒）归因于采样的调用栈。随着大量样本的出现，统计上正确的图像会浮现。采样的优点是它的开销非常低，因为它很少发生。不需要修改字节码，并且采样周期比方法调用的典型持续时间要长得多。缺点是您无法确定任何方法调用计数。此外，仅调用几次的短时间运行方法可能根本不会出现。如果您正在寻找性能瓶颈，这并不重要，但如果您试图了解代码的详细运行时特性，这可能会带来不便。



另一方面，如果对许多短时间运行的方法进行插桩，可能会引入大量开销。这种插桩会扭曲性能热点的相对重要性，因为时间测量的固有开销，但也因为许多本来会被热点编译器内联的方法现在必须保持为单独的方法调用。对于需要较长时间的方法调用，开销是微不足道的。如果您可以找到一组主要执行高级操作的类，插桩将增加非常低的开销，并且可能比采样更可取。JProfiler的开销热点检测也可以在一些运行后改善情况。此外，调用计数通常是重要的信息，使得更容易看出发生了什么。



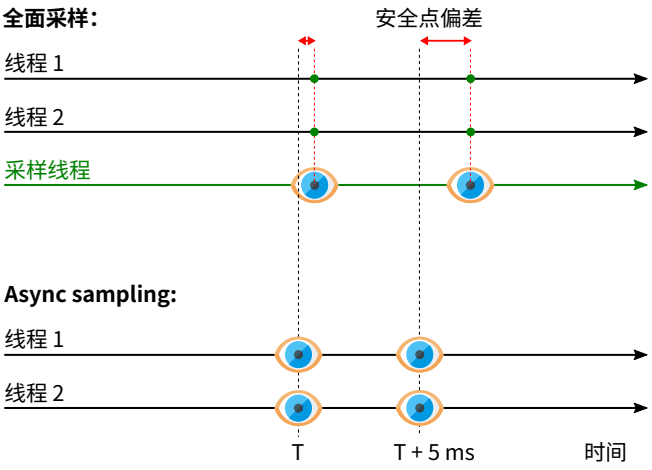
完整采样与异步采样

JProfiler为采样提供了两种不同的技术解决方案：“完整采样”是通过一个单独的线程完成的，该线程定期暂停JVM中的所有线程并检查它们的栈跟踪。然而，JVM仅在某些“安全点”暂停线程，从而引入偏差。如果您有高度多线程的CPU绑定代码，被分析的热点分布可能会被扭曲。另一方面，如果代码也执行了大量I/O，这种偏差通常不会成为问题。

为了帮助获得高度CPU绑定代码的准确数字，JProfiler还提供异步采样。通过异步采样，分析信号处理程序在运行的线程本身上被调用。分析代理然后检查本机栈并提取Java栈帧。主要的好处是这种采样方法没有安全点偏差，并且对于高度多线程的CPU绑定应用程序，开销较低。然而，对于CPU视图来说，只能观察到“Running”线程状态，而“Waiting”、“Blocking”或“Net I/O”线程状态无法以这种方式测量。探针数据始终通过字节码插桩收集，因此您仍然可以获得JDBC和类似数据的所有线程状态。

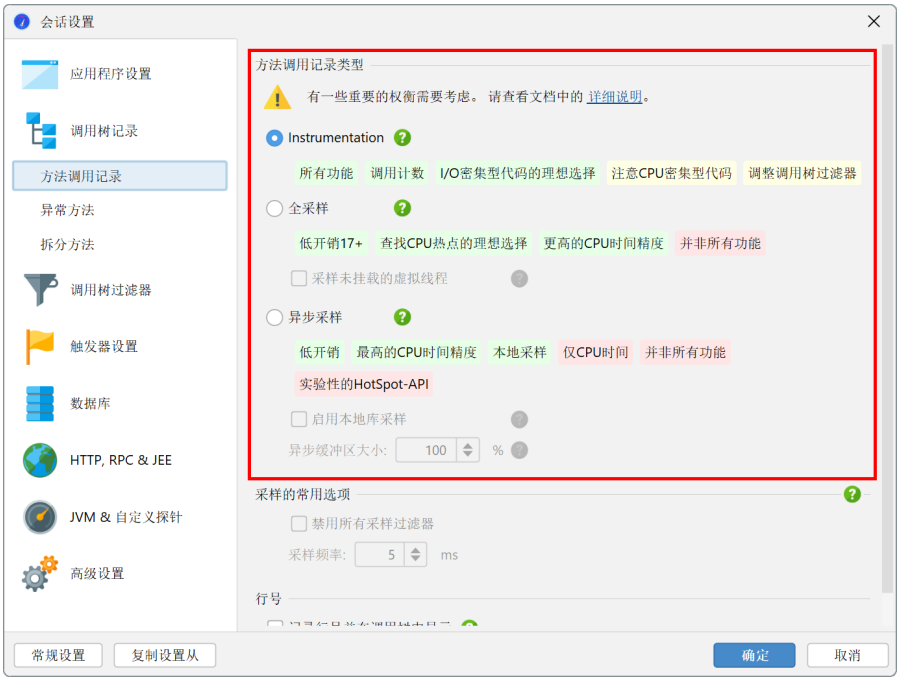
异步采样存在截断的跟踪，其中仅可用调用栈的末尾。这就是为什么调用树对于异步采样通常不如热点视图有用的原因。异步采样仅在Linux和macOS上受支持。

从Java 17开始，JProfiler可以避免在Hotspot JVM上使用全局安全点进行采样，并以接近零的开销进行完整采样。与异步采样相比，它仍然为单个线程引入某种安全点偏差，但不再为JVM中的所有线程引入全局安全点的开销。考虑到异步采样的缺点，建议在Java 17+上使用完整采样。



选择方法调用记录类型

选择用于分析的方法调用记录类型是一个重要的决定，并且没有适用于所有情况的正确选择，因此您需要做出明智的决定。当您创建新会话时，会话启动对话框会询问您要使用哪种方法调用记录类型。在任何以后的时间点，您都可以在会话设置对话框中更改方法调用记录类型。



作为一个简单的指南，请考虑以下问题，以测试您的应用程序是否属于光谱两端的两个明确类别之一：

- **被分析的应用程序是否I/O绑定?**

  许多Web应用程序大部分时间都在等待REST服务和JDBC数据库调用。在这种情况下，插桩将在您仔细选择调用树过滤器以仅包含您自己的代码的条件下是最佳选择。

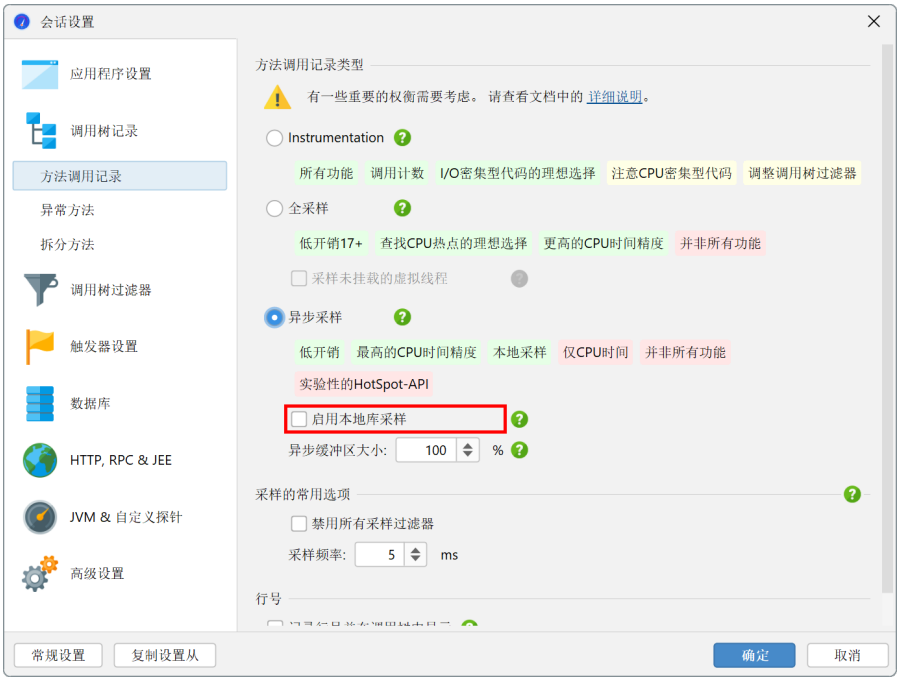
- **被分析的应用程序是否高度多线程且CPU绑定?**

  例如，这可能是编译器、图像处理应用程序或正在运行负载测试的Web服务器的情况。如果您在Linux或macOS上进行分析，您应该选择异步采样以在这种情况下获得最准确的CPU时间。

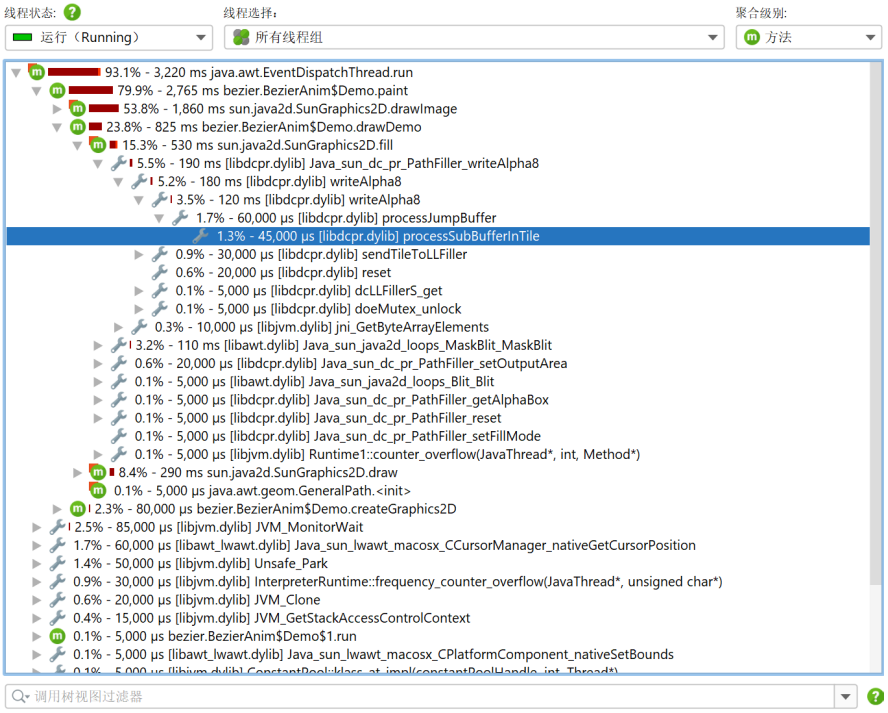
否则，“完整采样”通常是最合适的选项，并建议作为新会话的默认选项。

本机采样

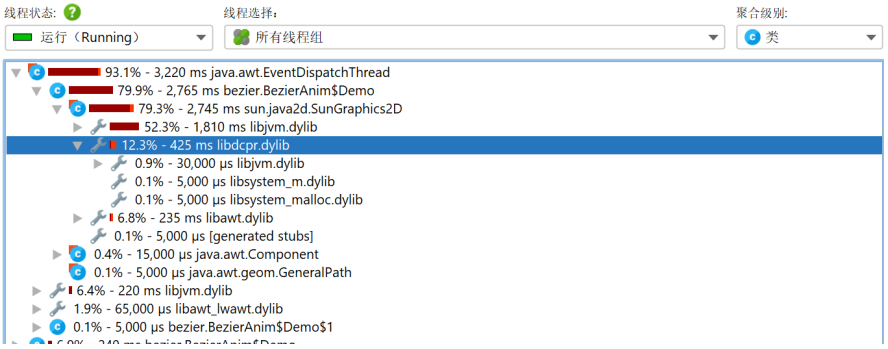
因为异步采样可以访问本机栈，所以它也可以执行本机采样。默认情况下，本机采样未启用，因为它会在调用树中引入大量节点，并将热点计算的重点转移到本机代码。如果您确实在本机代码中遇到性能问题，您可以选择异步采样并在会话设置中启用本机采样。



JProfiler解析属于每个本机栈帧的库路径。在调用树中的本机方法节点上，JProfiler在开头以方括号显示本机库的文件名。



关于聚合级别，本机库的作用类似于类，因此在“类”聚合级别中，同一本机库内的所有后续调用将聚合到一个节点中。“包”聚合级别将所有后续本机方法调用聚合到一个节点中，而不考虑本机库。



要消除选定的本机库，您可以从该本机库中移除一个节点 [p. 166]并选择移除整个类。

内存分析

有两种方式可以获得堆上对象的信息。一方面，分析代理可以跟踪每个对象的分配和垃圾回收情况。在JProfiler中，这被称为"分配记录"。它会告诉你对象被分配的位置，也可以用来创建有关临时对象的统计。另一方面，JVM的分析接口允许分析代理生成"堆快照"，以便检查所有活动对象及其引用。这些信息对于理解为什么对象不能被垃圾回收不可缺少。

分配记录和堆快照都是成本很高的操作。分配记录对运行时的特性有很大的影响，因为`java.lang.Object`构造函数必须被插入指令（Instrument），而垃圾回收器必须不断向分析接口报告。这就是为什么默认情况下不记录分配，你必须显式地开始和停止记录 [p. 27]。 生成堆快照是一次性操作。然而，它可能会使JVM暂停几秒钟，并且对获取的数据进行分析可能需要相对较长的时间，堆越大需要的时间越长。

JProfiler将其内存分析分为两个视图部分:"实时内存"部分显示的是可以定期更新的数据，而"堆遍历器"部分显示的是静态的堆快照。 分配记录是在"Live memory"部分控制的，但记录的数据也在堆遍历器中显示。

通过内存分析可以解决三个最常见的问题，分别是： 查找内存泄漏 [p. 199]、减少内存消耗和减少临时对象的创建。 对于前两个问题，你主要会使用堆遍历器，主要是看谁持有JVM中最大对象，以及它们在哪里被创建。 对于最后一个问题，你只能依靠显示记录的分配的实时视图，因为它涉及已经被垃圾回收的对象。

跟踪实例数

为了了解堆上有哪些对象，"所有对象"视图向你显示了所有类及其实例计数的直方图。该视图中显示的数据不是通过分配记录收集的，而是通过执行一个迷你堆快照，只计算实例计数。堆越大，执行该操作所需的时间就越长，因此视图的更新频率会根据测量的开销自动降低。当视图不活动时，不收集数据，视图不会产生任何开销。与大多数动态更新的视图一样， 一个 冻结工具栏按钮可以停止更新显示的数据。

而"记录的对象"视图，只显示你启动分配记录后被分配的对象实例计数。当你停止分配记录时，不会添加新的分配，但会继续跟踪垃圾回收。 通过这种方式，你可以看到某个用例的堆上还剩下哪些对象。注意，对象可能在很长一段时间内都不会被垃圾回收。 通过运行GC工具栏按钮，你可以加快这个过程。

当寻找内存泄漏时，你通常想比较一段时间内的实例数。要对所有的类进行比较，可以使用视图的差值功能。 使用 标记当前工具栏按钮，会插入一个相差列，实例计数的直方图以绿色显示标记时的基线值。

对于所选类，你还可以通过使用上下文菜单中的将所选内容添加到类跟踪器操作显示一个时间解析图。

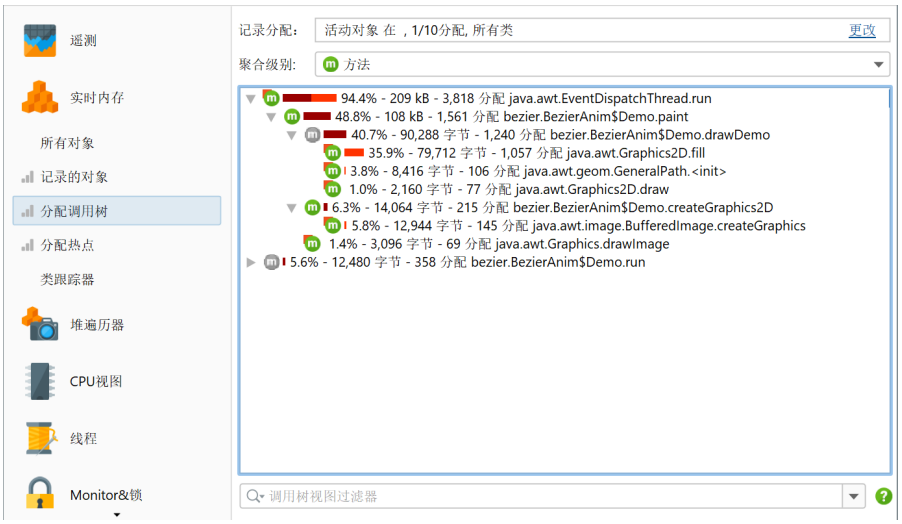
分配点

当分配记录处于激活状态时，JProfiler会在每次分配对象时记录调用堆栈。它不会使用确切的调用堆栈，例如来自堆遍历API的调用堆栈， 因为那样成本太高。而是使用与CPU分析配置相同的机制。这意味着调用堆栈会根据调用树过滤器 [p. 51]进行过滤， 而且实际的分配点可能在调用堆栈中没有显示的方法中，因为它来自一个被忽略（Ignored）或压缩（Compact）过滤的类。然而，这些变化直观上很容易理解:一个压缩（Compact）过滤方法负责所有进一步调用压缩（Compact）过滤类的分配。

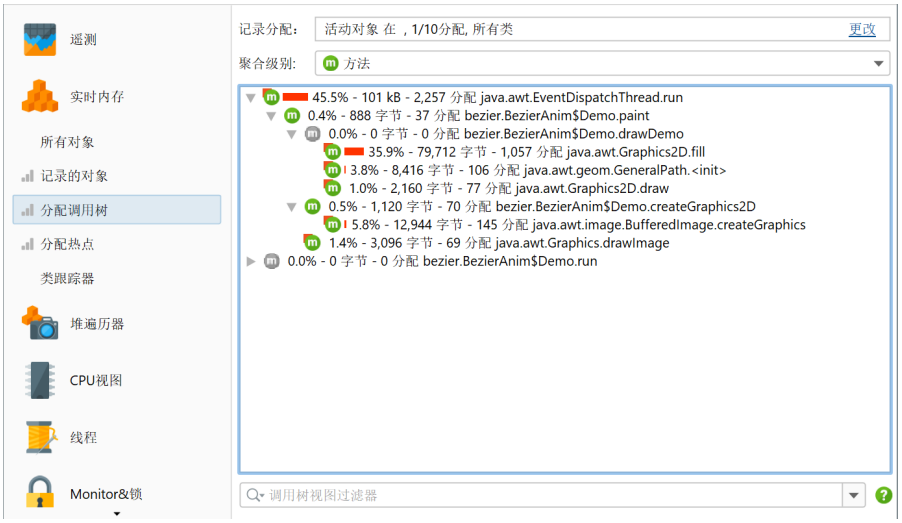
如果你使用采样，分配点就会变得粗略估计不准确，可能会令人困惑。不像时间测量，你通常会清楚地知道某些类在哪里可以分配，哪里不能分配。因为采样描绘的是统计而不是精确的图，你可能会看到一些看似不可能的分配点， 比如`java.util.HashMap.get`分配你自己的一个类。对于任何一种对精确数字和调用堆栈很重要的分析， 建议将分配记录和Instrumentation一起使用。

就像CPU分析一样，分配调用堆栈也是以调用树的形式显示的，只是显示的是分配计数和分配的内存，而不是调用计数和时间。与CPU调用树不同的是，分配调用树不会自动显示和更新，因为该树的计算成本较高。JProfiler不仅可以显示所有对象的分配树，还可以显示所选类或包的分配树。与其他选项一起，这在你要求JProfiler从当前数据计算分配树后显示的选项对话框中进行配置。

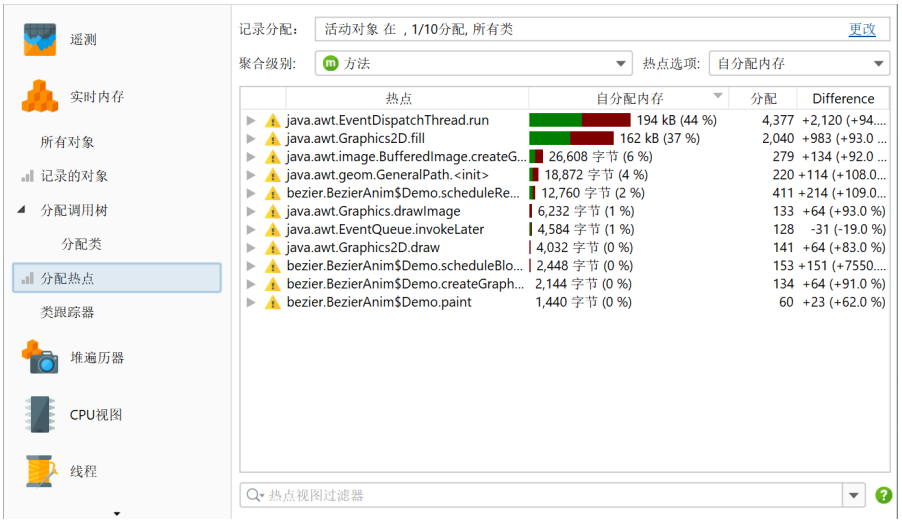
CPU调用树的一个有用的特性是，你可以沿着从上到下累计的时间，因为每个节点都包含了子节点所花费的时间。默认情况下，分配树的行为也是如此，这意味着每个节点都包含了子节点进行的分配。即使分配只由调用树深处的叶节点执行， 数字也会向上传递到顶部。这样一来，你在打开分配调用树的分支时，总能看到哪条路径值得调查。 "自身分配"是指那些实际由节点而不是由其子节点执行的分配。像在CPU调用树中一样，百分比栏用不同的颜色显示它们。



在分配调用树中，往往有很多节点完全没有执行分配，特别是当你为一个选定类显示分配时。这些节点存在只是为了向你展示到实际执行分配节点的完整调用堆栈路径。这样的节点在JProfiler中被称为"桥接"节点，并以灰色图标显示，正如你在上面的截图中看到的那样。在某些情况下，分配的累积可能会妨碍你，因为你只想看到实际的分配点。为此，分配树的视图设置提供了一个选项以显示非累计数。如果激活，桥接节点将始终显示零分配，并且没有百分比条。

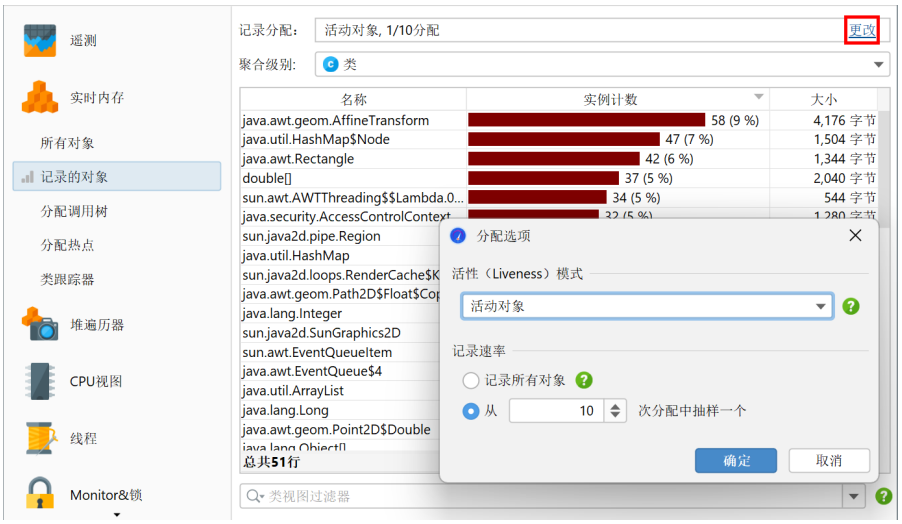


分配热点视图与分配调用树一起，允许你直接关注负责创建所选类的方法。就像记录的对象视图，分配热点视图也支持标记当前状态和观察一段时间内的差值。视图中会添加一个差值列，它显示了热点自当标记当前值操作被调用后的变化。因为默认情况下，分配视图不会定期更新，所以你必须单击计算工具栏按钮以获得一个新数据集然后与基线值比较。 在选项对话框中可以有一个自动更新更新选项可用，但对于大堆，不建议使用。



分配记录率

记录每一次分配都会增加很大的开销。在很多情况下，分配的总数量并不重要，相对数量足以解决问题。这就是为什么JProfiler默认每10次分配只记录一次。这会使开销减少到记录所有分配的大概1/10。如果你想记录所有的分配，或者即使记录更少的分配也足以满足你的目的，你可以在记录的对象视图以及分配调用树和热点视图的参数对话框中改变记录率。

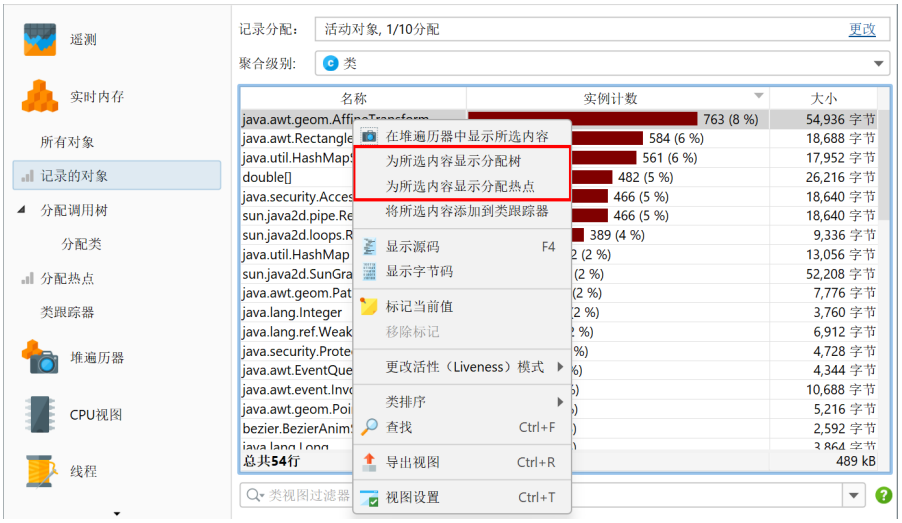


该设置也可以在会话设置对话框的"高级设置->内存分析"步骤中找到，在那里可以为离线分析会话进行调整。

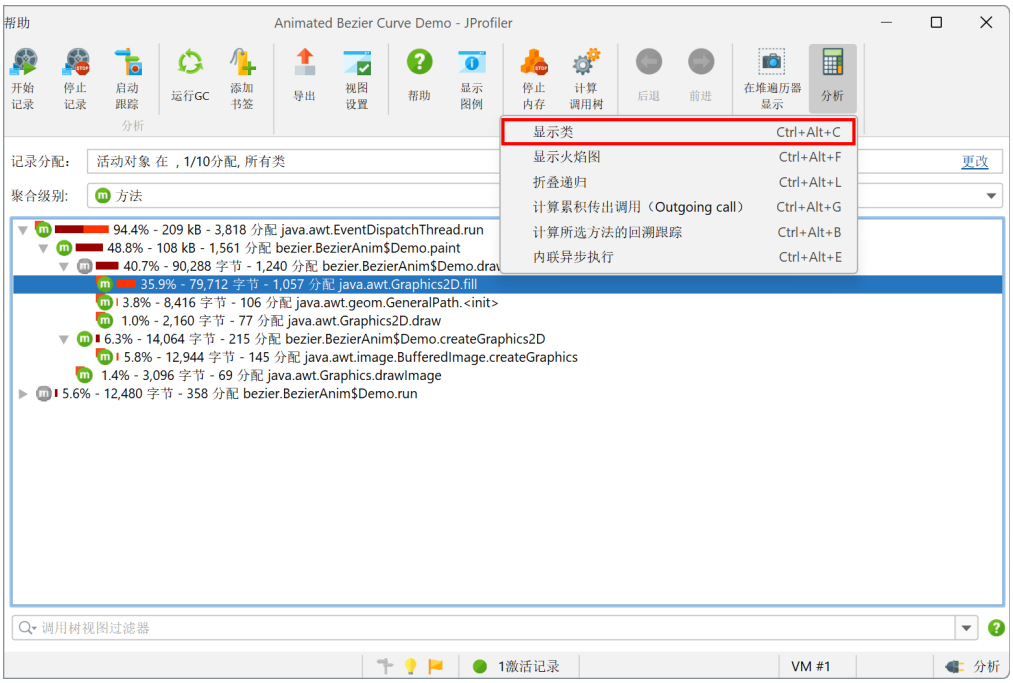
分配记录率会影响"记录的对象"和"记录吞吐量"VM遥测，这些值将以配置的分数来测量。 比较快照 [p. 123]时，将报告第一个快照的分配率，如有必要，将对其他快照进行相应的缩放。

分析分配的类

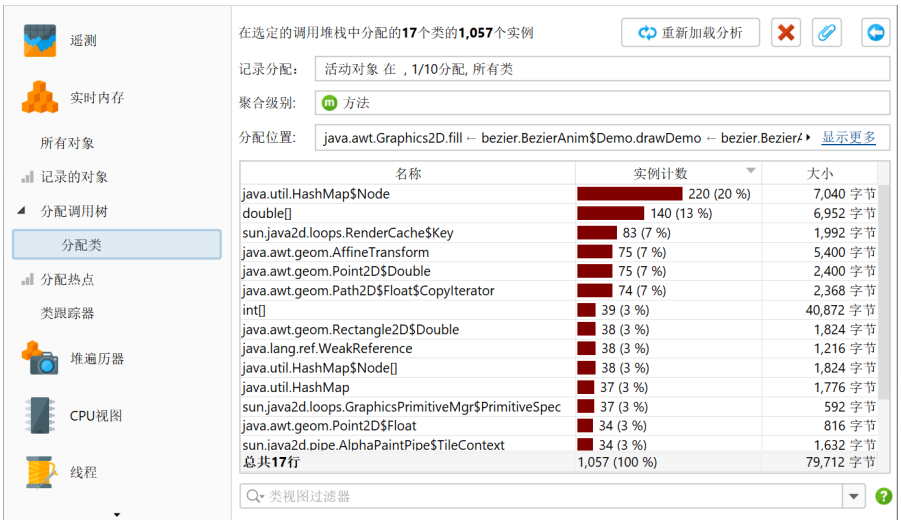
当计算分配树和分配热点视图时，你必须预先指定你想看到其分配的类或包。如果你已经在关注特定类，这工作的很好。 但当你想在没有任何预设的情况下查找分配热点时，就不方便了。一种方式是从"记录的对象"视图开始查看，然后使用上下文菜单中的操作切换到所选类或包的分配树或分配热点视图。



另一种方式是从所有类的分配树或分配热点开始，用显示类操作来显示所选分配点或分配热点的类。



分配的类的直方图显示为调用树分析 [p. 175]。这个操作对于其他调用树分析也有效。

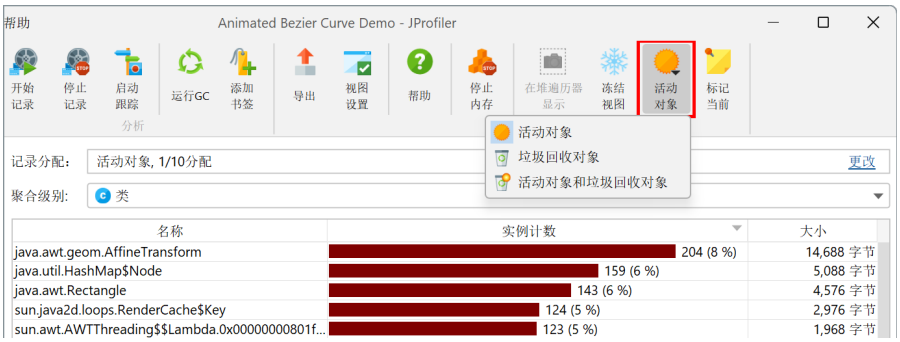


类分析视图是静态的，当重新计算分配树和热点视图时，不会更新。重新加载分析操作将首先更新分配树， 然后根据新数据重新计算当前分析视图。

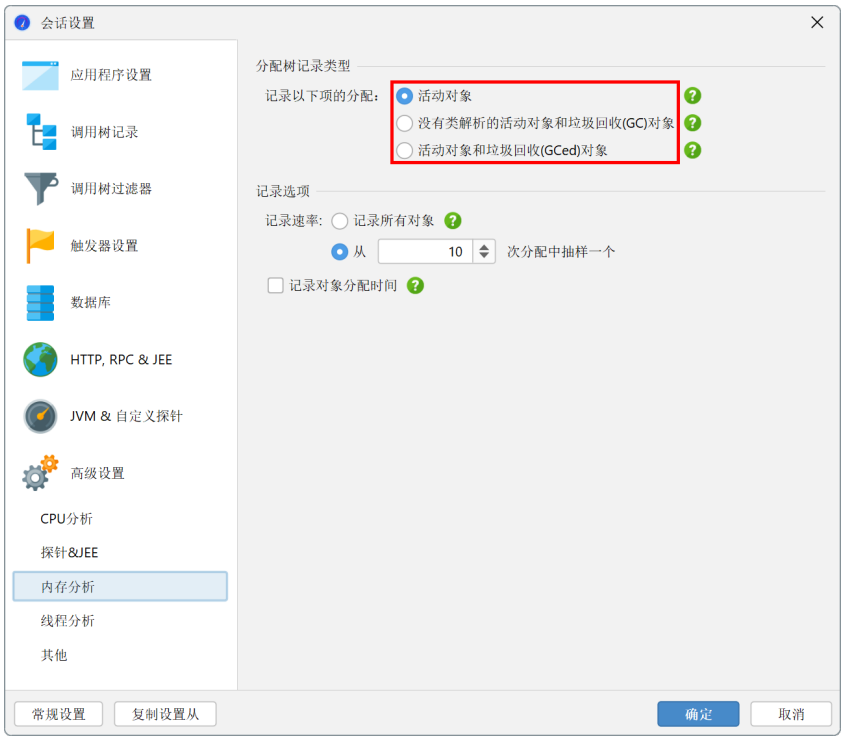
分析垃圾回收对象

分配记录不仅可以显示实时对象，还可以保留垃圾回收对象的信息。这在调查临时分配时很有用。分配大量的临时对象会产生很大的开销， 所以降低分配率往往能大大改善性能。

要在记录的对象视图中显示垃圾回收对象，请将活性选择器更改为垃圾回收对象或活动对象和垃圾回收对象。 分配调用树和分配热点视图的选项对话框有一个类似的下拉菜单。



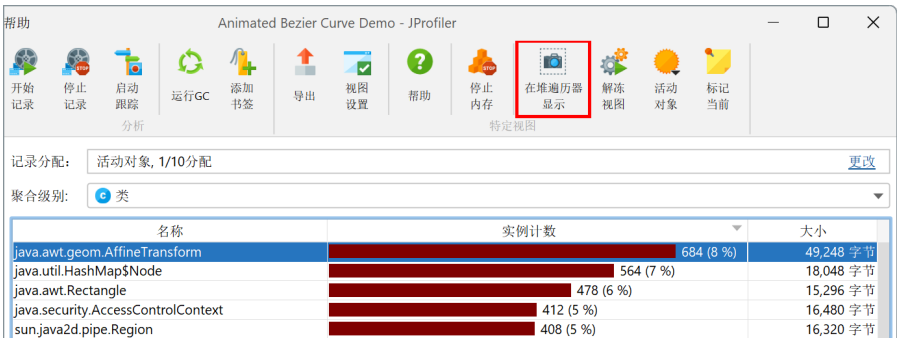
但是，JProfiler默认情况下并不收集垃圾回收对象的分配树信息，因为只维护活动对象的数据可以减少很多开销。 当将"分配调用树"或"分配热点"视图中的活性选择器切换到包括垃圾回收对象的模式时，JProfiler建议更改记录类型。这是分析设置的一个改变，因此如果你选择立即应用该改变，所有以前记录的数据将被清除。 如果你想提前更改这个设置，可以在会话设置对话框的"高级设置"->"内存分析"中进行更改。



下一站：堆遍历器

任何深入一点的问题都会涉及到对象之间的引用。例如，在记录的对象、分配树和分配热点视图中显示的大小都是**浅层大小**，它们只是包括了类的内存布局，但不包括任何引用的类。要想知道一个类的对象到底有多大，你通常需要知道**保留大小**， 也就是如果从堆中删除这些对象所释放的内存量。

这种信息在实时内存视图中是没有的，因为它需要枚举堆上的所有对象并执行成本昂贵的计算。这项工作由堆遍历器处理。要从实时内存视图中的感兴趣点跳转到堆遍历器中，可以使用在堆遍历器中显示所选内容工具栏按钮。 它会将你带入堆遍历器中的相应视图。



如果没有可用的堆快照，则会创建一个新的堆快照，否则JProfiler会询问你是否使用现有堆快照。

在任何情况下，重要的是要理解实时内存视图和堆遍历器中的数字通常会有很大的不同。除了堆步行器显示的是与实时内存视图不同的时间点的快照外，它还排除了所有未引用的对象。根据垃圾回收器的状态，未引用的对象可能会占据堆的很大一部分。

堆遍历器

堆快照

任何涉及对象之间引用的堆分析都需要堆快照，因为无法询问JVM对象的传入引用是什么。您必须遍历整个堆来回答这个问题。从该堆快照中，JProfiler创建了一个内部数据库，该数据库经过优化以生成堆遍历器中视图所需的数据。

堆快照有两个来源：JProfiler堆快照和HPROF/PHD堆快照。JProfiler堆快照支持堆遍历器中的所有可用功能。分析代理使用分析接口JVMTI遍历所有引用。如果被分析的JVM在不同的机器上运行，所有信息将传输到本地机器，并在那里进行进一步计算。HPROF/PHD快照是通过JVM中的内置机制创建的，并以JProfiler可以读取的标准格式写入磁盘。HotSpot JVM可以创建HPROF快照，而Eclipse OpenJ9 JVM提供PHD快照。

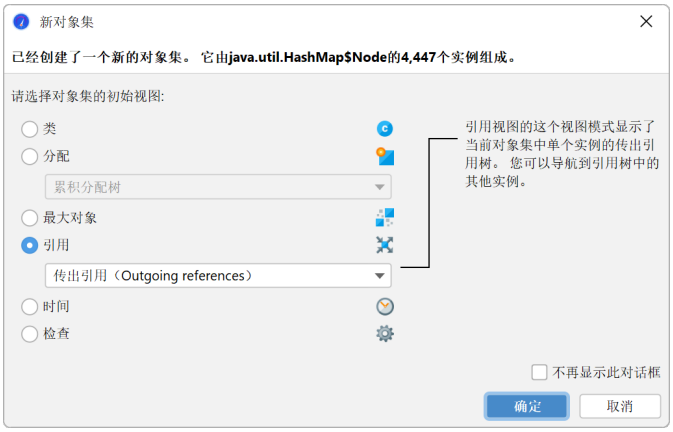
在堆遍历器的概览页面上，您可以选择是创建JProfiler堆快照还是HPROF/PHD堆快照。默认情况下，推荐使用JProfiler堆快照。HPROF/PHD堆快照在另一章 [p. 191]中讨论的特殊情况下很有用。

选择步骤

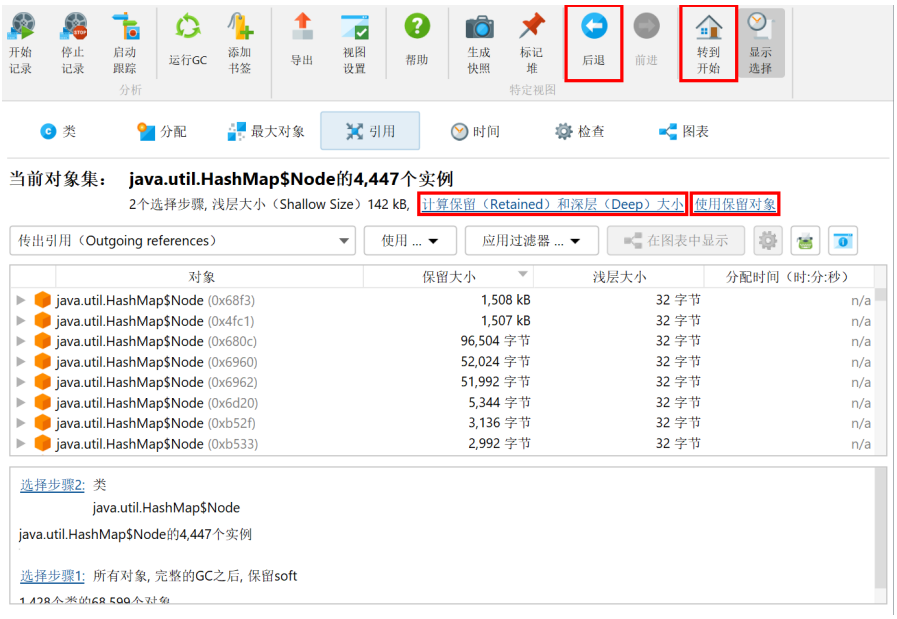
堆遍历器由几个视图组成，这些视图显示所选对象集的不同方面。拍摄堆快照后，您正在查看堆上的所有对象。每个视图都有导航操作，用于将一些选定的对象转换为**当前对象集**。堆遍历器的标题区域显示当前对象集中包含多少个对象的信息。

最初，您正在查看“Classes”视图，该视图类似于live memory section中的“All objects”视图。通过选择一个类并调用Use->Selected Instances，您可以创建一个仅包含该类实例的新对象集。在堆遍历器中，“使用”总是意味着创建一个新的对象集。

对于新的对象集，显示堆遍历器的类视图并不有趣，因为它实际上只是将表过滤到先前选择的类。相反，JProfiler建议使用“New object set”对话框中的另一个视图。您可以取消此对话框以放弃新对象集并返回到上一个视图。建议使用传出引用视图，但您也可以选择其他视图。这只是最初显示的视图，您可以在堆遍历器的视图选择器中切换视图。



标题区域现在告诉您有两个选择步骤，并包括用于计算保留和深度大小或使用当前对象集保留的所有对象的链接。后者将添加另一个选择步骤，并建议类视图，因为在该对象集中可能会有多个类。



在堆遍历器的下部，列出了到目前为止的选择步骤。单击超链接将带您返回到任何选择步骤。第一个数据集也可以通过工具栏中的Go To Start按钮到达。如果您需要回溯分析，工具栏中的后退和前进按钮非常有用。

Classes视图

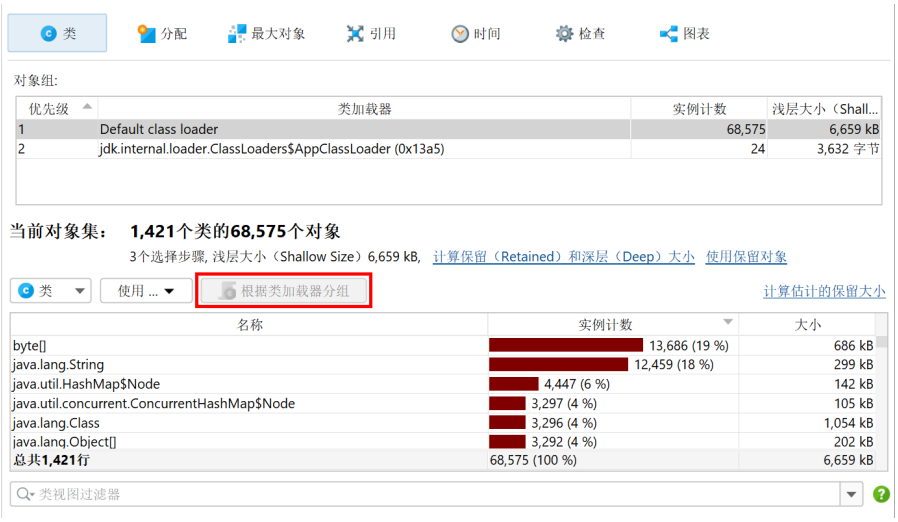
堆遍历器顶部的视图选择器包含五个视图，这些视图显示当前对象集的不同信息。其中第一个是“Classes”视图。

类视图类似于live memory section中的“All objects”视图，并具有可以将类分组到包中的聚合级别选择器。此外，它可以显示类的估计保留大小。这是如果从堆中删除一个类的所有实例，将释放的内存量。如果单击Calculate estimated retained sizes超链接，将添加一个新的Retained Size列。显示

的保留大小是估计的下限，计算确切的数字会太慢。如果您确实需要确切的数字，请选择感兴趣的类或包，并使用新对象集标题中的Calculate retained and deep sizes超链接。

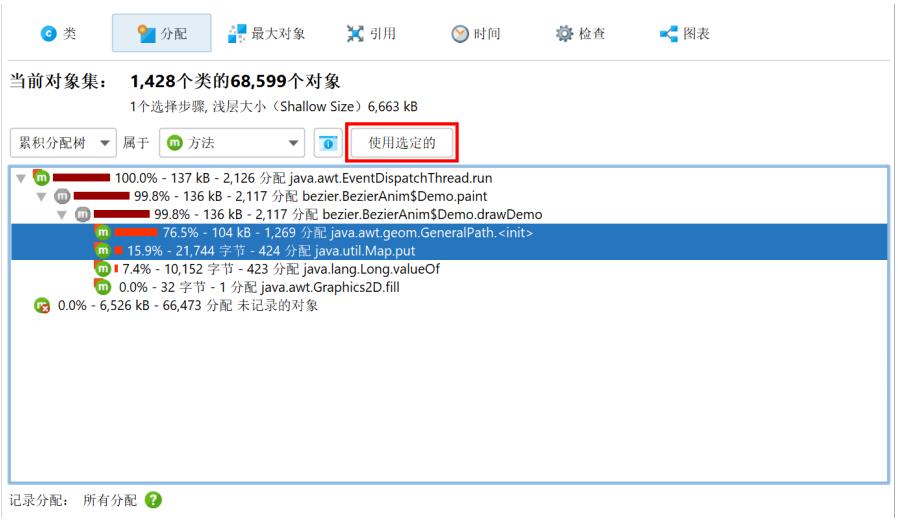
基于您选择的一个或多个类或包，您可以选择实例本身、关联的`java.lang.Class`对象或所有保留的对象。双击是最快的选择模式，并使用选定的实例。如果有多个选择模式可用，如在这种情况下，视图上方将显示一个Use下拉菜单。

在解决类加载器相关问题时，您通常需要按类加载器对实例进行分组。Inspections选项卡提供了一个“Group by class loaders”检查，该检查在类视图中可用，因为在该上下文中特别重要。如果您执行该分析，顶部的分组表将显示所有类加载器。选择一个类加载器会相应地过滤下面视图中的数据。当您切换到堆遍历器的其他视图时，分组表仍然存在，直到您执行另一个选择步骤。然后，类加载器选择成为该选择步骤的一部分。



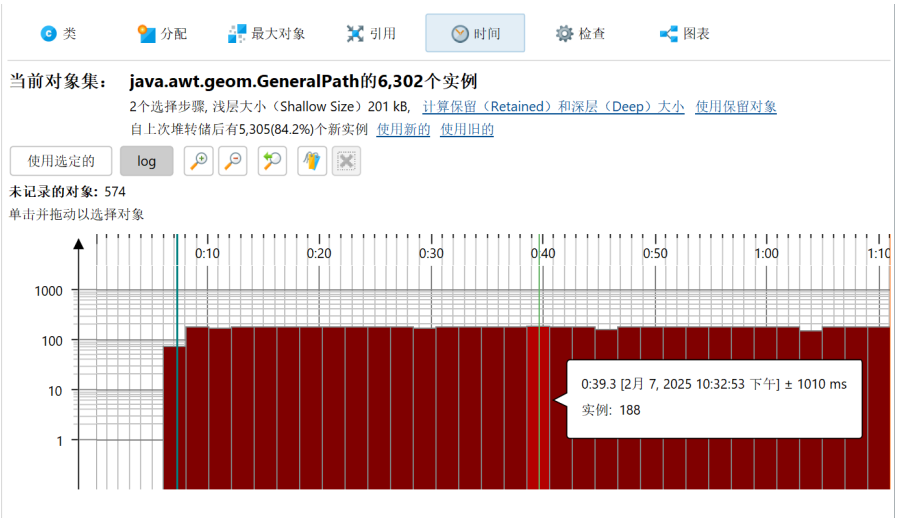
Allocation recording视图

当缩小内存泄漏嫌疑对象或尝试减少内存消耗时，了解对象的分配位置可能很重要。对于JProfiler堆快照，“Allocations”视图显示分配调用树和分配热点，适用于那些记录了分配的对象。其他对象在分配调用树中被分组到“unrecorded objects”节点中。对于HPROF/PHD快照，此视图不可用。



与类视图一样，您可以选择多个节点，并使用顶部的Use Selected按钮创建新的选择步骤。在“Allocation hot spots”视图模式中，您还可以选择回溯中的节点。这将仅选择在关联的顶级热点上分配的对象，这些对象的调用栈以所选回溯结束。

JProfiler在记录分配时可以保存的另一条信息是对象分配的时间。堆遍历器中的“Time”视图显示当前对象集中所有记录实例的分配时间直方图。您可以单击并拖动以选择一个或多个间隔，然后使用Use Selected按钮创建一个新的对象集。



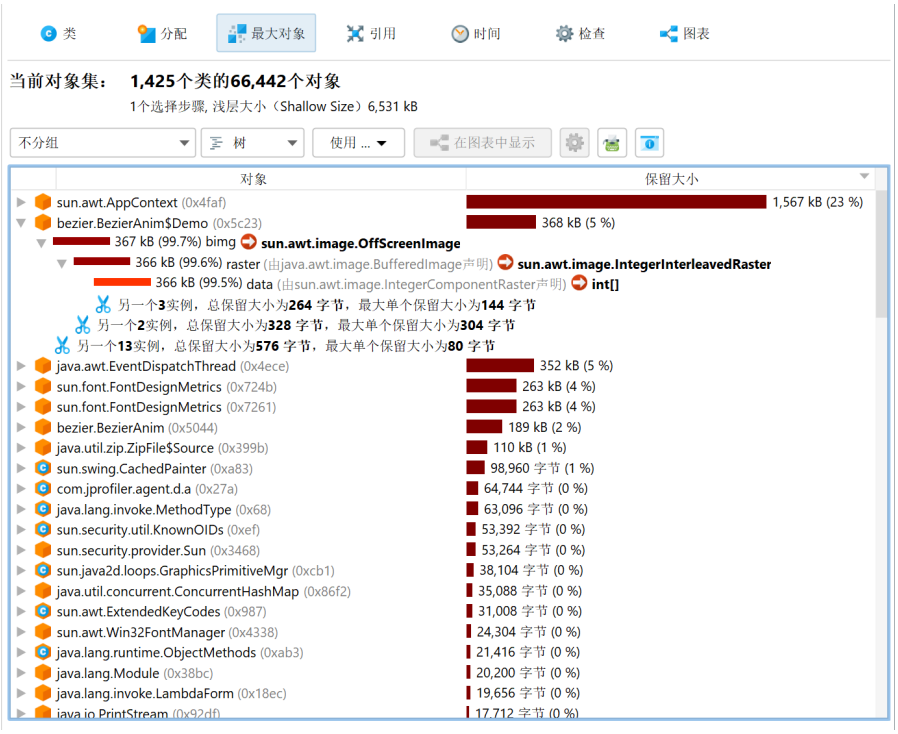
为了更精确地选择时间间隔，您可以指定bookmarks的范围。所有在第一个和最后一个选定书签之间的对象将被标记。

除了时间视图，分配时间还显示为引用视图中的单独列。然而，默认情况下不启用分配时间记录。您可以直接在时间视图中打开它，或者在会话设置对话框中的Advanced Settings -> Memory Profiling中编辑设置。

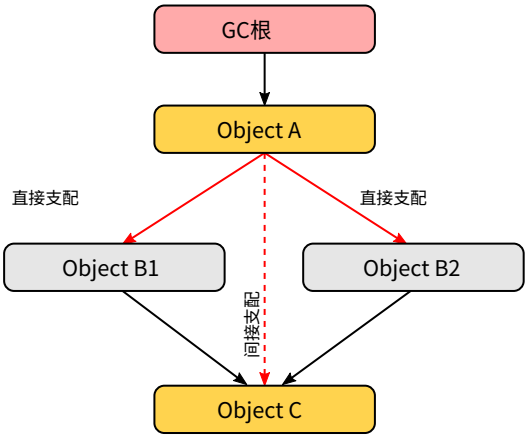
Biggest objects视图

最大对象视图显示当前对象集中最重要对象的列表。在此上下文中，“最大”意味着如果从堆中删除这些对象，将释放最多内存的对象。该大小称为**retained size**。相反，**deep size**是通过强引用可达的所有对象的总大小。

每个对象都可以展开以显示由该对象保留的其他对象的传出引用。通过这种方式，您可以递归展开保留对象的树，如果删除其中一个祖先，这些对象将被垃圾回收。这种树称为“dominatortree”。在此树中显示的每个对象的信息类似于传出引用视图，只是仅显示主导引用。



并非所有被支配的对象都直接由其支配者引用。例如，考虑下图中的引用：



对象A支配对象B1和B2，并且它没有直接引用对象C。B1和B2都引用C。B1和B2都不支配C，但A支配。在这种情况下，B1、B2和C被列为A在支配树中的直接子节点，C不会被列为B1和B2的子节点。对于B1和B2，显示它们在A中被持有的字段名称。对于C，在引用节点上显示“[transitive reference]”。

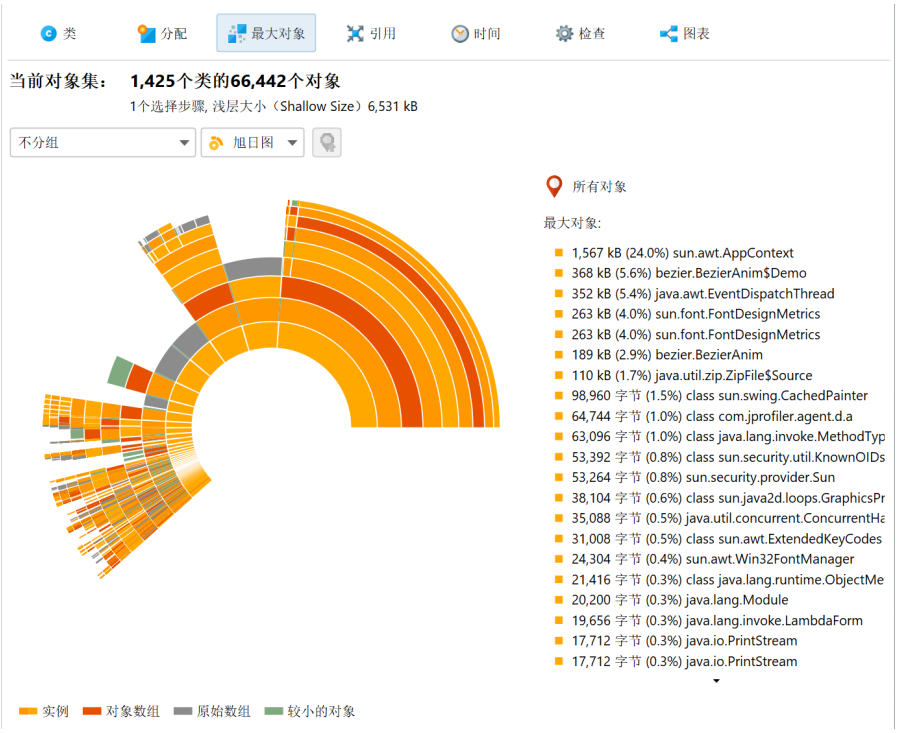
在支配树的每个引用节点的左侧，大小条显示顶级对象的保留大小中仍由目标对象保留的百分比。随着您进一步深入树中，数字将减少。在视图设置中，您可以将百分比基数更改为总堆大小。

支配树有一个内置的截止点，消除了所有保留大小低于父对象保留大小0.5%的对象。这是为了避免过长的小支配对象列表，这些对象会分散对重要对象的注意力。如果发生这样的截止，将显示一个特殊的“cutoff”子节点，通知您在此级别上未显示的对象数量、它们的总保留大小以及单个对象的最大保留大小。

除了显示单个对象，支配树还可以将最大对象分组到类中。视图顶部的分组下拉菜单包含一个复选框，用于激活此显示模式。此外，您可以在顶级添加类加载器分组。类加载器分组在计算最大对象后应用，并显示谁加载了最大对象的类。如果您想分析特定类加载器的最大对象，可以先使用“Group by class loader”检查。

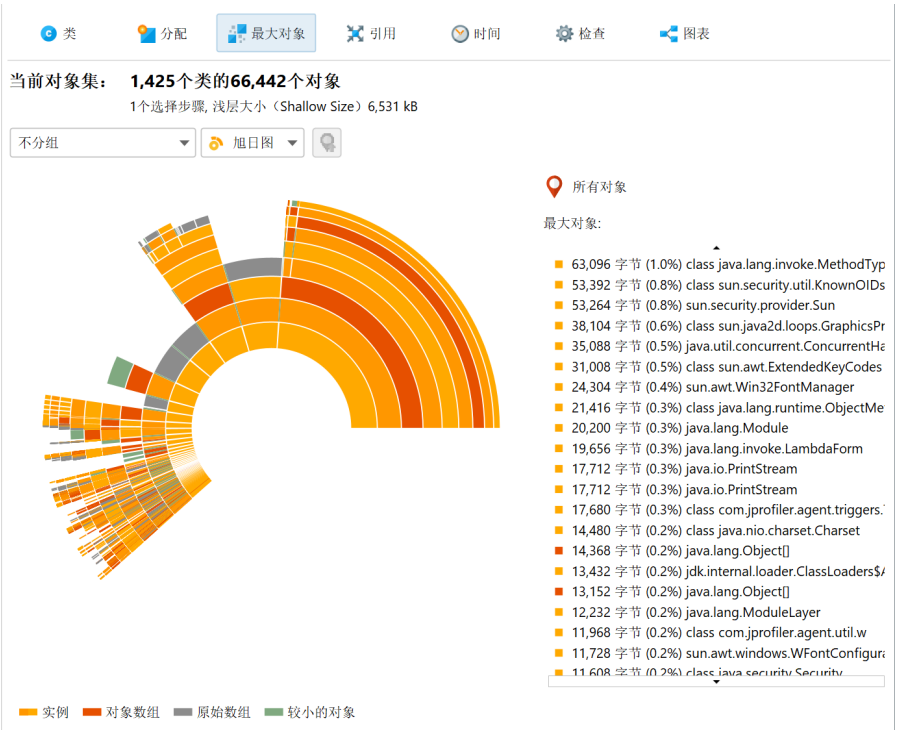
最大对象视图上方的视图模式选择器允许您切换到旭日图。该图由一系列同心分段环组成，显示支配树的整个内容，最多深度为一个图像。引用起源于最内层的环，并向圆的外缘传播。此可视化为您提供了一个扁平化的视角，具有高信息密度，使您能够发现引用模式，并通过其特殊的颜色编码一目了然地看到大型原始和对象数组。

如果当前对象集是整个堆，则圆的总周长对应于已用堆大小。因为最大对象视图仅显示保留超过总堆0.1%的对象，这意味着一个重要的扇区将是空的，对应于所有未被这些最大对象保留的对象。



单击任何环段都会为圆设置一个新的根，从而扩展您在图中可以看到的最大深度。单击图的空心中心会恢复先前的根。如果设置了新根，则圆的总周长对应于根对象的保留大小。空扇区表示根对象的自

大小和在最大保留对象列表中不存在的其他对象。如果当前对象集不是整个堆，则圆的总周长对应于所有显示的最大对象的总和，并且不会显示空扇区。

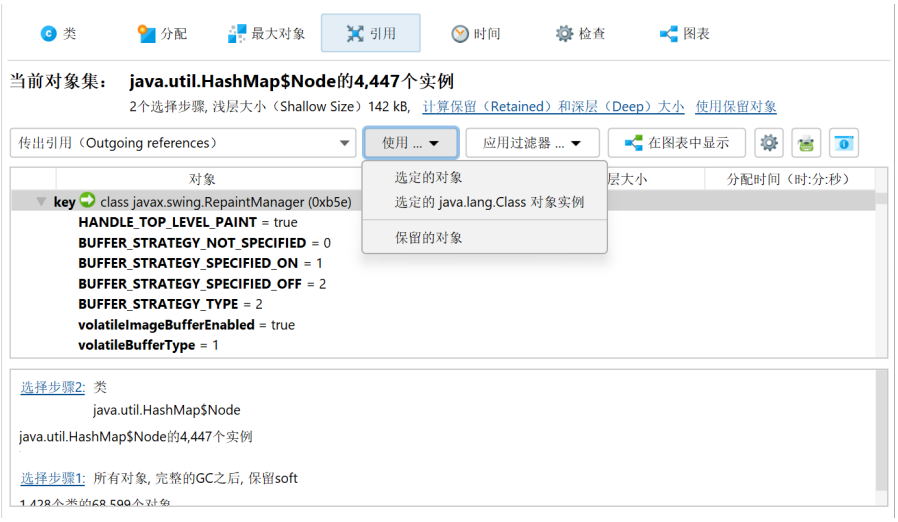


当您将鼠标悬停在它们上面时，图的右侧会显示有关实例及其立即保留对象的更多信息。当鼠标不在任何环段上时，右侧的列表显示最内层环中的最大对象。将鼠标悬停在该列表上会突出显示相应的环段，单击列表项会为图设置一个新根。要创建一个新的对象集，您可以从上下文菜单中的操作中选择，无论是在环段上还是在列表项上。

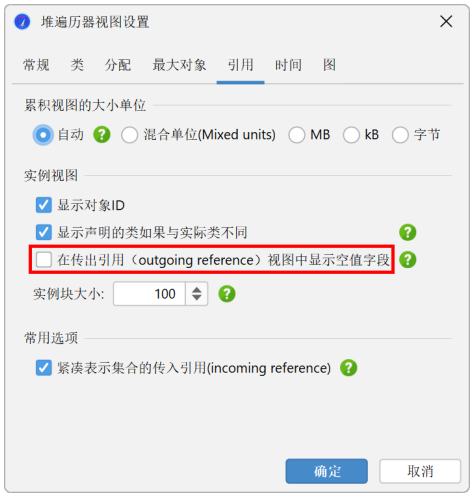
Reference视图

与之前的视图不同，引用视图仅在您执行至少一个选择步骤后可用。对于初始对象集，这些视图没有用，因为传入和传出引用视图显示所有单个对象，并且合并引用视图只能针对专注的对象集进行解释。

传出引用视图类似于IDE中的调试器会显示的视图。打开对象时，您可以看到原始数据和对其他对象的引用。任何引用类型都可以选择为新的对象集，您可以一次选择多个对象。与类视图一样，您可以选择保留的对象或关联的`java.lang.Class`对象。如果选定的对象是标准集合，您还可以通过单个操作选择所有包含的元素。对于类加载器对象，有一个选项可以选择所有加载的实例。

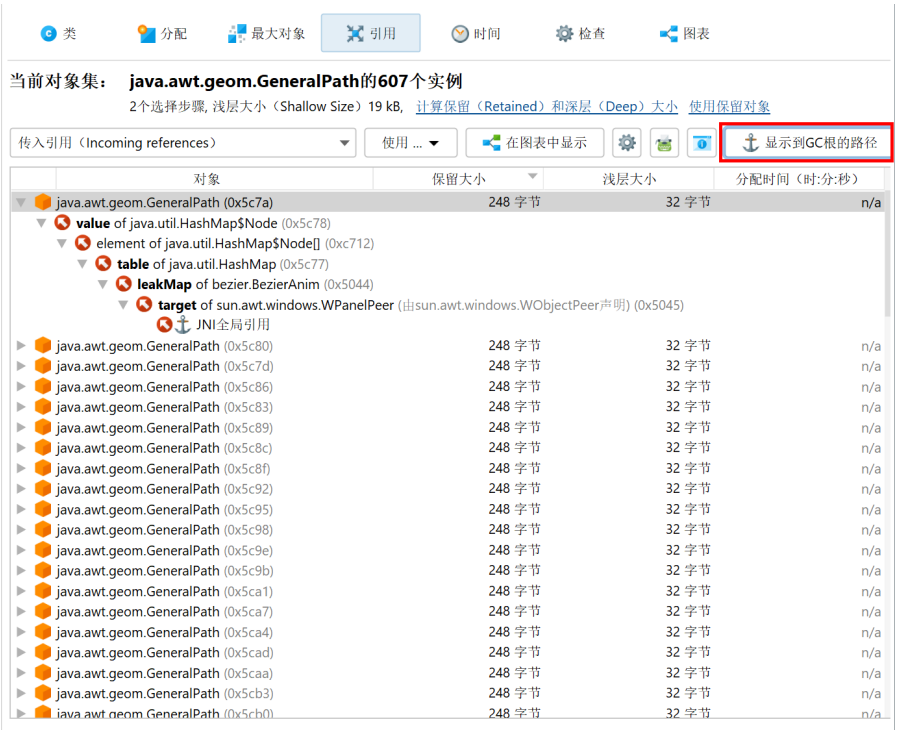


默认情况下，不显示具有空引用的字段，因为该信息可能会分散内存分析的注意力。如果您想出于调试目的查看所有字段，可以在视图设置中更改此行为。



除了简单选择显示的实例，传出引用视图还具有 强大的过滤功能 [p. 195]。 对于实时会话，传出和传入引用视图都具有高级的操作和显示功能，在同一章中进行了讨论。

传入引用视图是解决内存泄漏的主要工具。要找出为什么对象没有被垃圾回收，Show Paths To GC Root按钮将找到到垃圾收集器根的引用链。关于这个重要主题的详细信息，请参见内存泄漏 [p. 199] 章节。

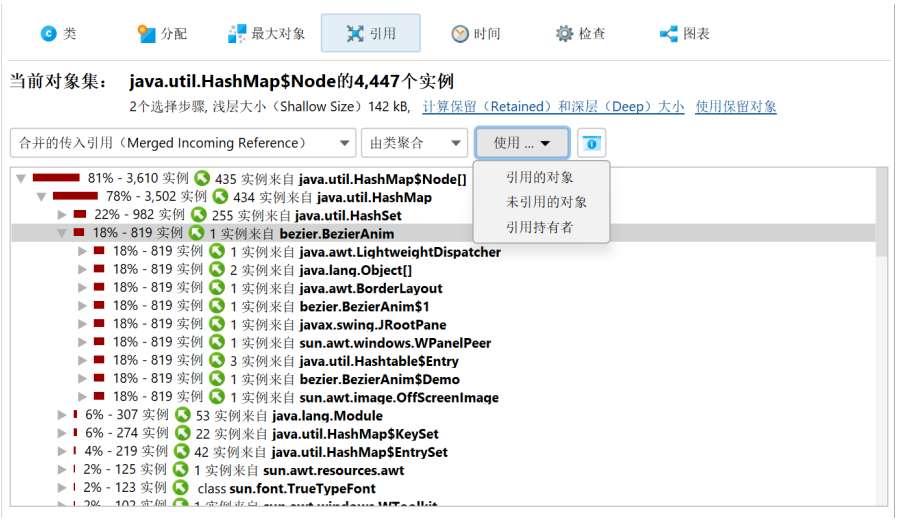


Merged references

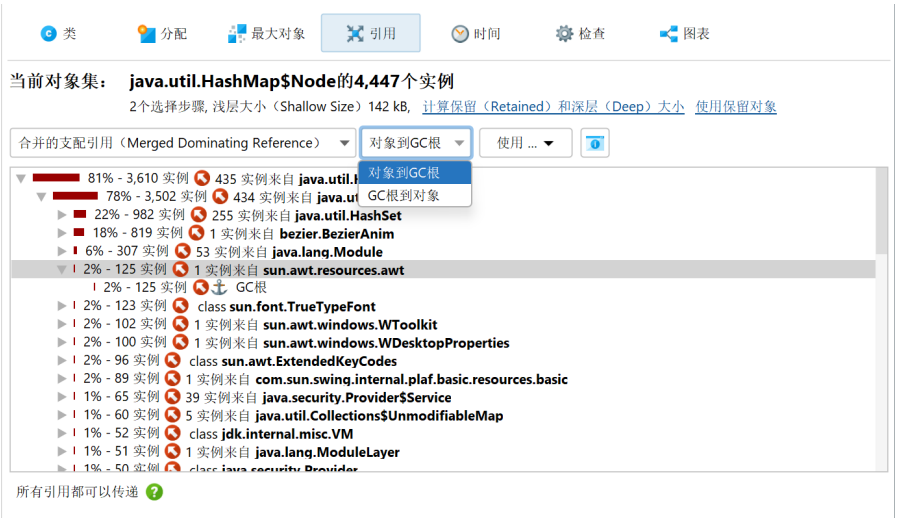
检查许多不同对象的引用可能很繁琐，因此JProfiler可以向您显示当前对象集中所有对象的合并传出和传入引用。默认情况下，引用按类聚合。如果一个类的实例被同一类的其他实例引用，则插入一个特殊节点，显示原始实例加上这些类递归引用的实例。此机制自动折叠常见数据结构中的内部引用链，例如在链表中。

您还可以选择按字段分组显示合并引用。在这种情况下，每个节点都是一个引用类型，例如类的特定字段或数组的内容。对于标准集合，内部引用链会破坏累积，因此您会看到类似“map value of java.lang.HashMap”的引用类型。与类聚合不同，此机制仅适用于JRE标准库中显式支持的集合。

在“Merged outgoing references”视图中，实例计数指的是引用的对象。在“Merged incoming references”视图中，您会在每行看到两个实例计数。第一个实例计数显示当前对象集中沿此路径引用的实例数量。节点左侧的条形图标可视化此比例。箭头图标后的第二个实例计数指的是持有对父节点引用的对象。在执行选择步骤时，您可以选择是要选择当前对象集中以选定方式引用的对象，还是对选定引用感兴趣的对象——引用持有者。



使用“Merged dominating references”视图，您可以找出必须删除哪些引用，以便当前对象集中的某些或所有对象可以被垃圾回收。支配引用树可以解释为最大对象视图中支配树的合并逆，按类聚合。引用箭头可能不表示两个类之间的直接引用，但可能有其他类在中间持有非支配引用。在多个垃圾收集器根的情况下，可能没有支配引用存在于当前对象集中的某些或所有对象。

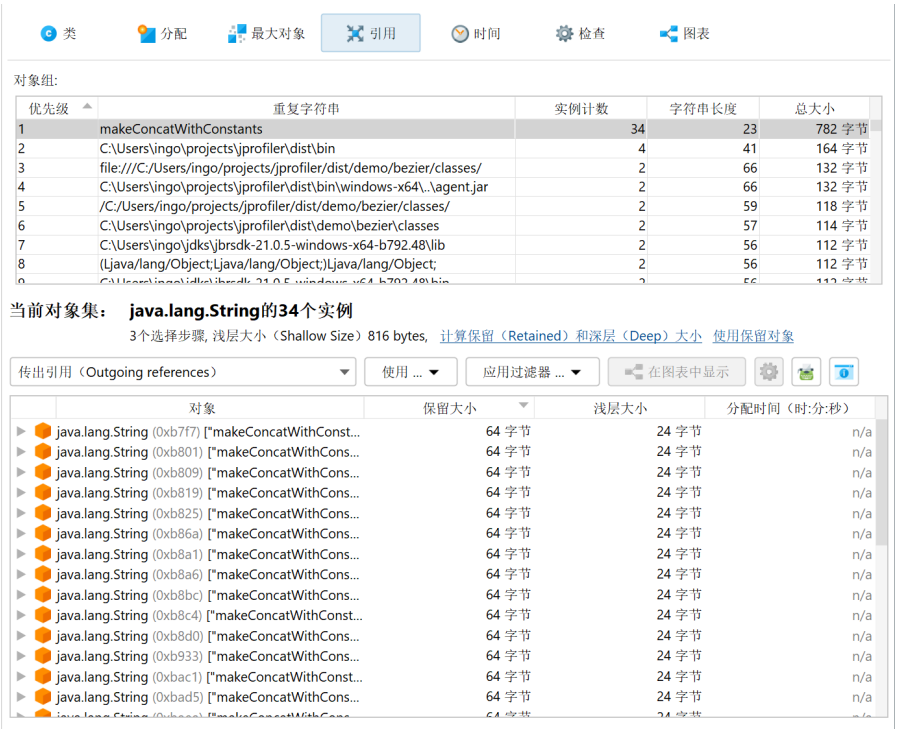


默认情况下，“Merged dominating references”视图显示传入的支配引用，通过打开树，您可以到达由GC根持有的对象。有时，引用树可能通过许多不同的路径引导到相同的根对象。通过在视图顶部的下拉菜单中选择“GC roots to objects”视图模式，您可以看到反向视角，其中根位于顶级，当前对象集中的对象位于叶节点。在这种情况下，引用从顶级向叶节点传播。哪个视角更好取决于您要消除的引用是接近当前对象集还是接近GC根。

Inspections

“Inspections”视图本身不显示数据。它提供了一些堆分析，根据其他视图中不可用的规则创建新的对象集。例如，您可能想查看由线程本地保留的所有对象。这在引用视图中是不可能做到的。检查被分为几个类别，并在其描述中进行解释。

检查可以将计算的对象集划分为组。组显示在堆遍历器顶部的表中。例如，“Duplicate strings”检查将重复的字符串值显示为组。如果您在引用视图中，您可以看到具有选定字符串值的`java.lang.String`实例。最初，组表中的第一行被选中。通过更改选择，您更改了当前对象集。组表的Instance Count和Size列告诉您选择一行时当前对象集的大小。



组选择不是堆遍历器中的单独选择步骤，但它成为检查所做选择步骤的一部分。您可以在底部的选择步骤窗格中看到组选择。当您更改组选择时，选择步骤窗格会立即更新。

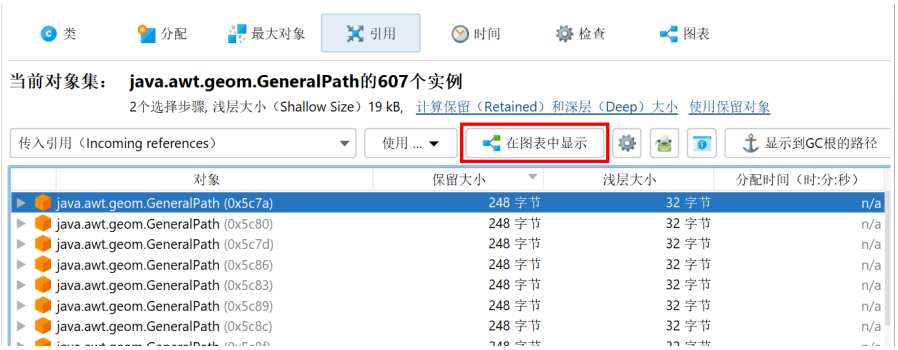
每个创建组的检查决定哪些组在检查的上下文中最重要。因为这并不总是与其他列的自然排序顺序相对应，组表中的Priority列包含一个数值，用于强制检查的排序顺序。

对于大堆，检查可能计算成本高，因此结果会被缓存。这样，您可以回到历史记录中查看先前计算的检查结果，而无需等待。

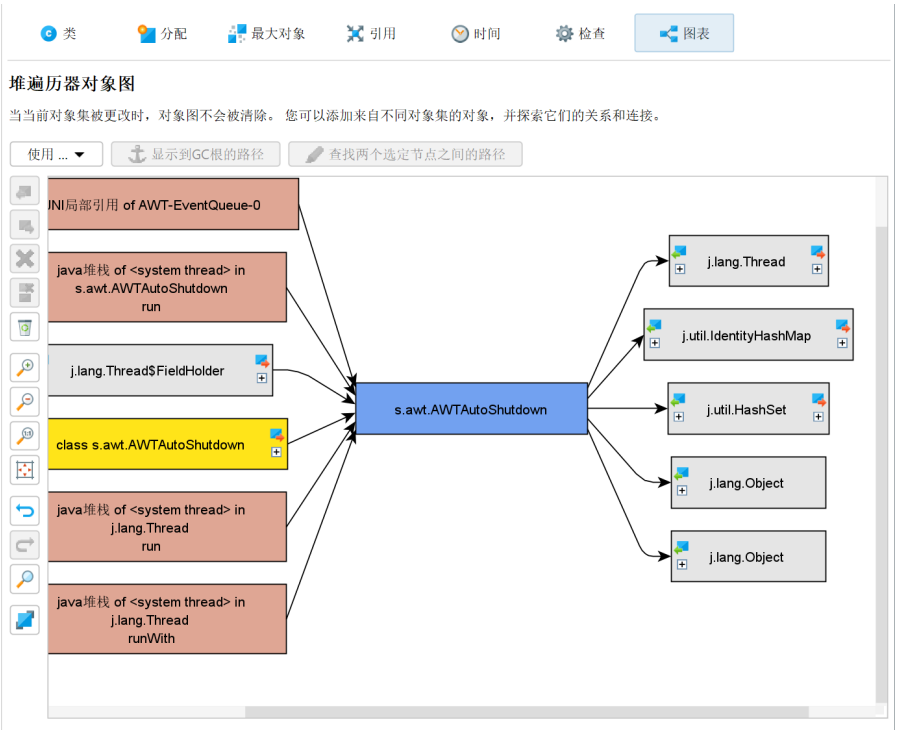
Heap walker graph

实例及其引用的最真实表示是图形。虽然图形的视觉密度低，并且对于某些类型的分析不实用，但它仍然是可视化对象之间关系的最佳方式。例如，循环引用在树中难以解释，但在图中立即显而易见。此外，看到传入和传出引用可能是有益的，这在树结构中是不可能的，在树结构中您可以看到其中之一。

堆遍历器图形不会自动显示当前对象集中的任何对象，也不会在您更改当前对象集时清除。您可以从传出引用视图、传入引用视图或最大对象视图中手动将选定对象添加到图形中，方法是选择一个或多个实例并使用Show In Graph操作。



默认情况下，图中的包名称会缩短。与CPU调用图一样，您可以在视图设置中启用完整显示。引用以箭头形式绘制。如果您将鼠标移动到引用上，将显示一个工具提示窗口，显示特定引用的详细信息。从引用视图手动添加的实例具有蓝色背景。实例添加得越晚，背景颜色越深。垃圾收集器根具有红色背景，类具有黄色背景。



默认情况下，引用图仅显示当前实例的直接传入和传出引用。您可以通过双击任何对象来扩展图形。这将根据您移动的方向，展开该对象的直接传入或直接传出引用。使用实例左右两侧的扩展控件，您可以选择性地打开传入和传出引用。如果您需要回溯，请使用撤销功能恢复图形的先前状态，以免被太多节点分散注意力。要修剪图形，有删除所有未连接节点或甚至删除所有对象的操作。

与传入引用视图一样，图形具有Show Path To GC Root按钮，如果可用，将扩展到垃圾收集器根的一个或多个引用链。此外，还有一个Find Path Between Two Selected Nodes操作，如果选择了两个实例，则处于活动状态。它可以搜索有向和无向路径，并可选地沿着弱引用。如果找到合适的路径，将以红色显示。

Initial object set

当您拍摄堆快照时，您可以指定控制初始对象集的选项。如果您记录了分配，Select recorded objects复选框将初始显示的对象限制为那些已记录的对象。数字通常会与live memory视图中的数字不同，因为堆遍历器会删除未引用的对象。未记录的对象仍然存在于堆快照中，只是没有显示在初始对象集中。通过进一步的选择步骤，您可以到达未记录的对象。

此外，堆遍历器会执行垃圾收集并删除弱引用对象，软引用除外。这通常是可取的，因为在寻找内存泄漏时，弱引用对象会分散注意力，而只有强引用对象才相关。然而，在那些您对弱引用对象感兴趣

的情况下，您可以告诉堆遍历器保留它们。JVM中的四种弱引用类型是“soft”、“weak”、“phantom”和“finalizer”，您可以选择其中哪种类型应该足以在堆快照中保留对象。

如果存在，弱引用对象可以通过使用堆遍历器中的“Weak reference”检查从当前对象集中选择或删除。

Marking the heap

通常，您希望查看为特定用例分配的对象。虽然您可以通过在该用例周围启动和停止分配记录来实现，但有一种更好的方法，具有更少的开销，并为其他目的保留分配记录功能：Mark Heap操作，该操作在堆遍历器概览中进行宣传，并且也可以在Profiling菜单中或作为触发器操作使用，将堆上的所有对象标记为“旧”。当您拍摄下一个堆快照时，现在很清楚“新”对象应该是什么。

如果存在先前的堆快照或标记堆调用，堆遍历器的标题区域会显示新的实例计数和两个标题为Use new和Use old的链接，允许您选择自那时起分配的实例，或在此之前分配的幸存实例。此信息适用于每个对象集，因此您可以先深入研究，然后选择新实例或旧实例。


类
分配
最大对象
引用
时间
检查
图表
当前对象集： 1,436个类的106,822个对象
1个选择步骤, 浅层大小（Shallow Size）8,748 kB
自上次堆转储后有38,507(36.0%)个新实例 使用新的 使用旧的
类
使用 ...
根据类加载器分组
计算估计的保留大小
名称
实例计数
大小
byte[] 22,119 (20 %) 1,018 kB
java.lang.String 15,572 (14 %) 373 kB
java.util.HashMap$Node 10,944 (10 %) 350 kB
java.lang.Long 6,570 (6 %) 157 kB

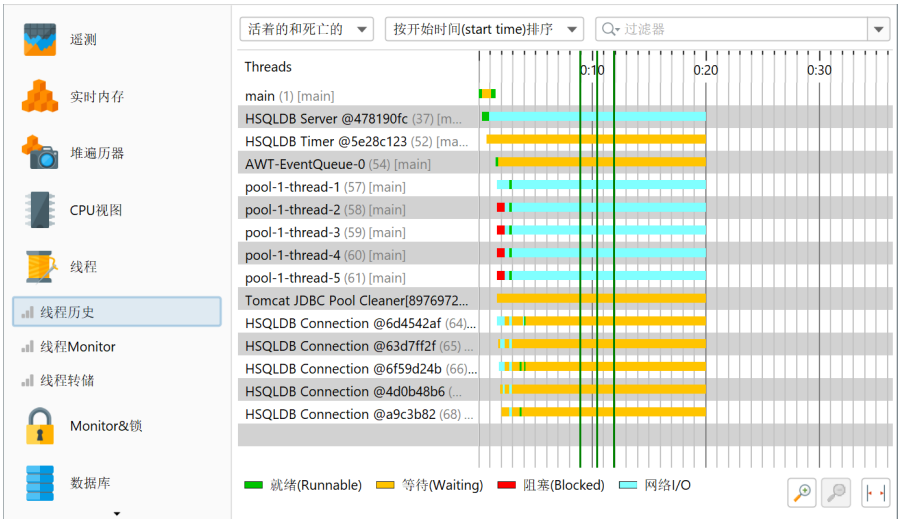
线程分析

不正确地使用线程会导致许多不同类型的问题。过多的活动线程可能导致线程饥饿，线程可能会互相阻塞并影响应用程序的活性，或者以错误的顺序获取锁可能导致死锁。此外，关于线程的信息对于调试目的也很重要。

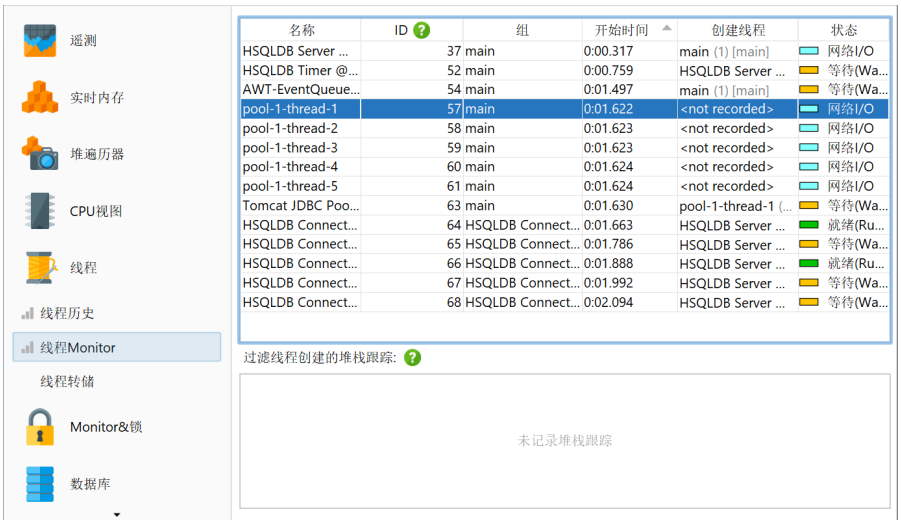
在 JProfiler 中，线程分析分为两个视图部分：“Threads”部分处理线程的生命周期和捕获线程转储。“Monitors & locks”部分提供分析多个线程交互的功能。

检查线程

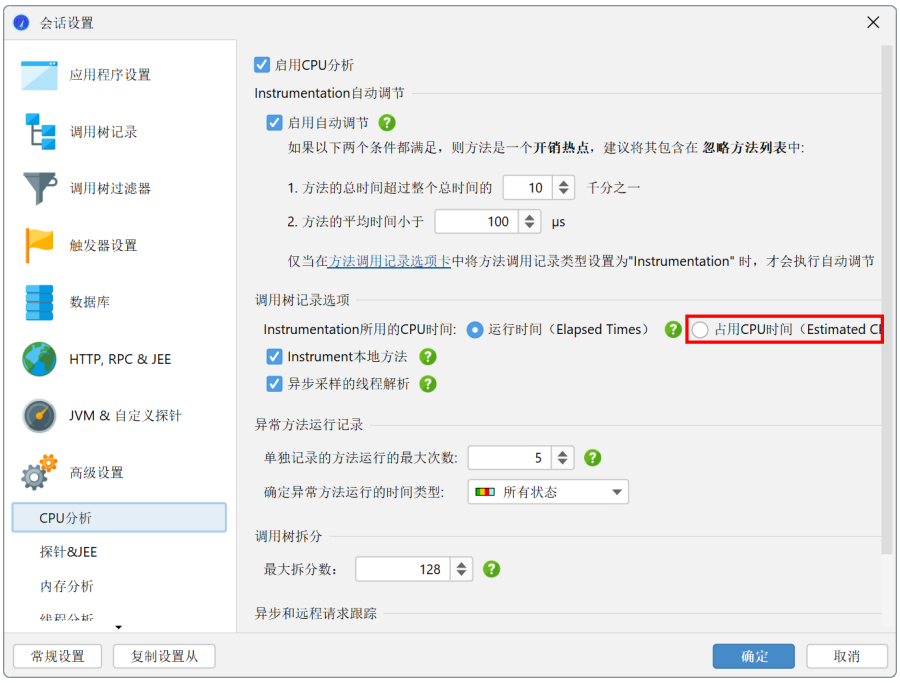
线程历史视图在时间线上显示每个线程为一条彩色行，其中颜色表示记录的线程状态。线程按创建时间、名称或线程组排序，并可以按名称过滤。您还可以通过拖放自行重新排列线程的顺序。当监视器事件被记录时，您可以悬停在线程的“Waiting”或“Blocked”状态部分，并查看与监视器历史视图的链接的相关调用栈。



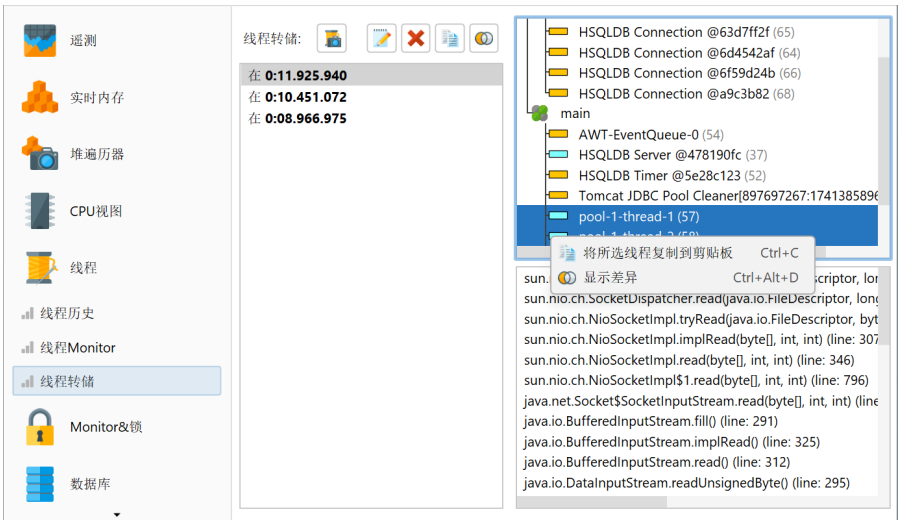
所有线程的表格视图可在线程监视器视图中使用。如果在创建线程时CPU记录处于活动状态，JProfiler会保存创建线程的名称并在表中显示。在底部，显示创建线程的调用栈。出于性能原因，不会从JVM请求实际的调用栈，而是使用 CPU 记录中的当前信息。这意味着调用栈将仅显示满足调用树收集过滤器设置的类。



如果您在分析设置中启用估算 CPU 时间的记录，则会在表中添加一个CPU Time列。仅在记录 CPU 数据时测量 CPU 时间。



像大多数调试器一样，JProfiler也可以进行线程转储。线程转储的调用栈是JVM提供的完整调用栈，不依赖于 CPU 记录。选择两个线程转储并单击Show Difference按钮时，可以在差异查看器中比较不同的线程转储。还可以通过选择单个线程转储中的两个线程并从上下文菜单中选择Show Difference来比较它们。

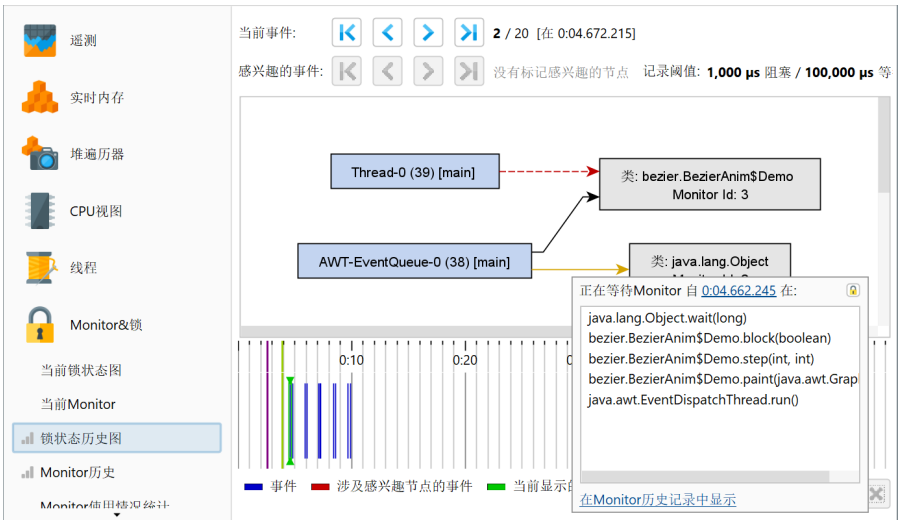


线程转储还可以通过“Trigger thread dump”触发器操作或通过API进行。

分析锁定情况

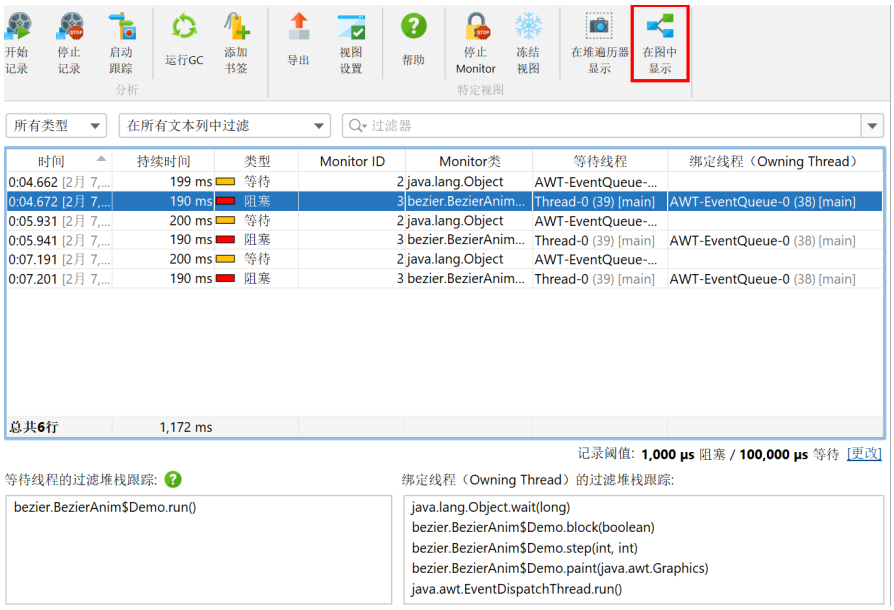
每个 Java 对象都有一个关联的监视器，可用于两种同步操作：线程可以在监视器上等待，直到另一个线程在其上发出通知，或者它可以获取监视器上的锁，可能会阻塞，直到另一个线程放弃锁的所有权。此外，Java在`java.util.concurrent.locks`包中提供了用于实现更高级锁定策略的类。该包中的锁不使用对象的监视器，而是使用不同的本机实现。

JProfiler可以记录上述两种机制的锁定情况。在锁定情况下，有一个或多个线程、一个监视器或`java.util.concurrent.locks.Lock`的实例以及一个需要一定时间的等待或阻塞操作。这些锁定情况以表格形式在监视器历史视图中呈现，并在锁定历史图中以可视化方式呈现。



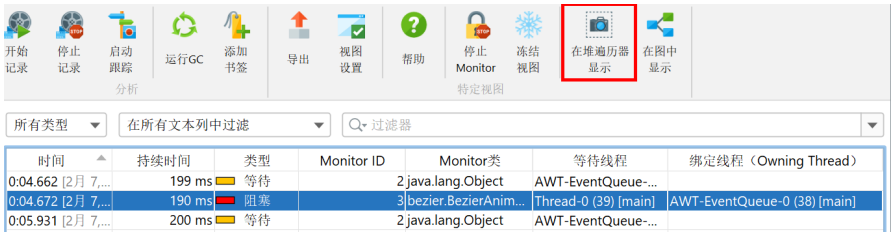
锁定历史图关注所有涉及的监视器和线程的整个关系集，而不是孤立监视器事件的持续时间。参与锁定情况的线程和监视器被绘制为蓝色和灰色矩形，如果它们是死锁的一部分，则绘制为红色。黑色箭头表示监视器的所有权，黄色箭头从等待线程延伸到关联的监视器，而虚线红色箭头表示线程想要获取监视器并且当前正在阻塞。如果记录了 CPU 数据，则在悬停在阻塞或等待箭头上时可以查看调用栈。这些工具提示包含超链接，可带您到监视器历史视图中的相应行。

表格监视器历史视图显示监视器事件。它们有一个显示为列的持续时间，因此您可以通过对表进行排序来找到最重要的事件。对于表格视图中选择的任何行，您可以通过Show in Graph操作跳转到图形。



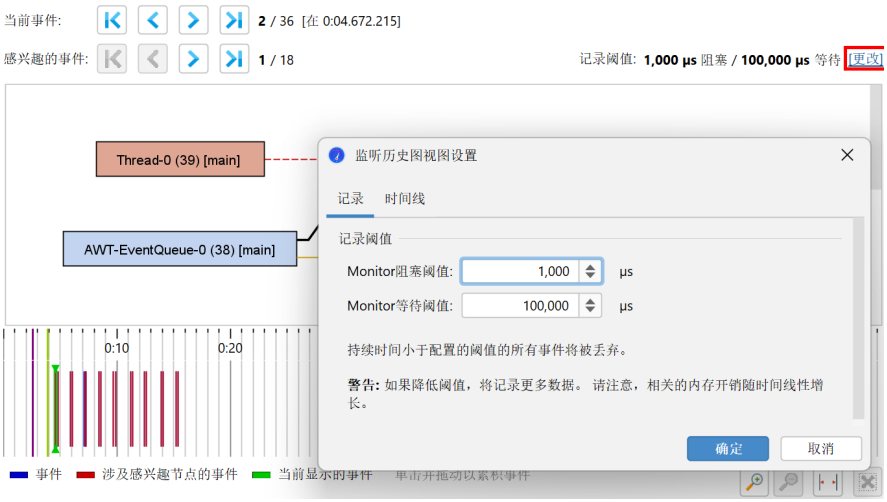
每个监视器事件都有一个关联的监视器。Monitor　Class列显示使用监视器的实例的类名，如果没有Java 对象与监视器关联，则显示“[raw monitor]”。无论如何，监视器都有一个唯一的 ID，显示在单独的列中，因此您可以关联多个事件中相同监视器的使用情况。每个监视器事件都有一个执行操作的等待线程和一个可能阻塞操作的拥有线程。如果可用，它们的调用栈显示在视图的下部。

如果您对监视器实例有进一步的问题，监视器历史视图和锁定历史图中的Show in Heap Walker操作提供了一个链接到堆行走器，并选择监视器实例作为新的对象集。



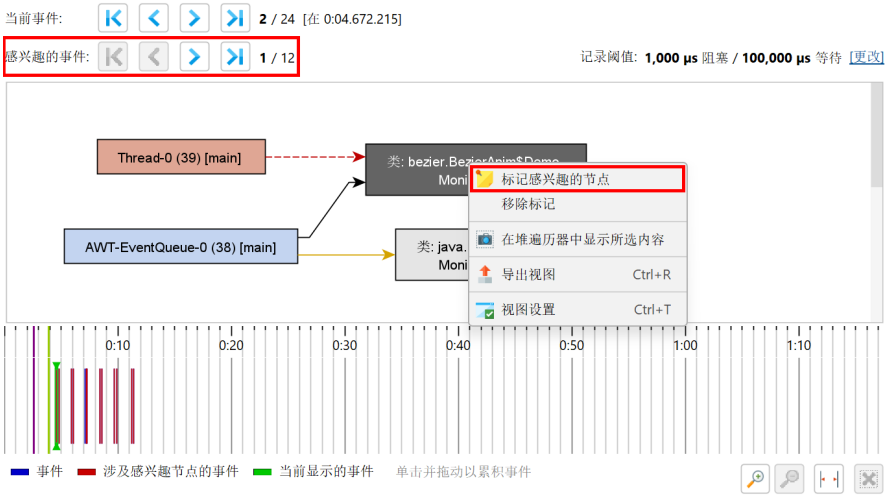
限制感兴趣的事件

分析监视器事件的一个基本问题是应用程序可能会以极高的速率生成监视器事件。这就是为什么JProfiler对等待和阻塞事件有默认阈值，低于该阈值的事件会立即被丢弃。这些阈值在视图设置中定义，可以增加以便专注于更长的事件。



对于记录的事件，您可以进一步应用过滤器。监视器历史视图在视图顶部提供了一个阈值、一个事件类型和一个文本过滤器。锁定历史图允许您选择感兴趣的线程或监视器，并仅显示涉及标记实体的锁定情况。感兴趣的事件在时间线上以不同的颜色显示，并且有一个次级导航栏可以逐步浏览这些事件。如果当前事件不是感兴趣的事件，您可以看到当前事件与任一方向上的下一个感兴趣事件之间有多少事件。

除了选择的线程或监视器存在的锁定情况外，还显示从图中删除它的锁定情况。这是因为每个监视器事件由两个这样的锁定情况定义，一个是操作开始时，一个是操作结束时。这也意味着完全空的图形是一个有效的锁定情况，表示 JVM 中没有更多的锁。



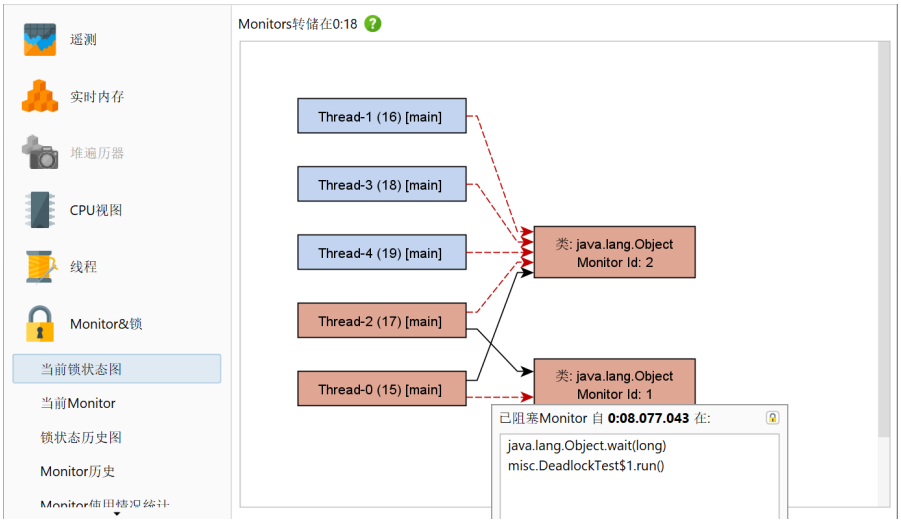
另一种减少需要您关注的事件数量的策略是累积锁定情况。在锁定历史图中，底部有一个时间线，显示所有记录的事件。单击并拖动它可以选择一个时间范围，并在上面的锁定图中显示所有包含事件的数据。在累积图中，每个箭头可以包含多个相同类型的事件。在这种情况下，工具提示窗口显示事件的数量以及所有包含事件的总时间。工具提示窗口中的下拉列表显示时间戳，并允许您在不同事件之间切换。

死锁检测

“Current locking graph”和“Current monitors”视图在 JProfiler UI 中通过一个动作触发的“monitor dump”上运行。通过监视器转储，您可以检查仍在进行中的事件。这包括死锁，这些事件永远不会完成，无法在历史视图中显示。

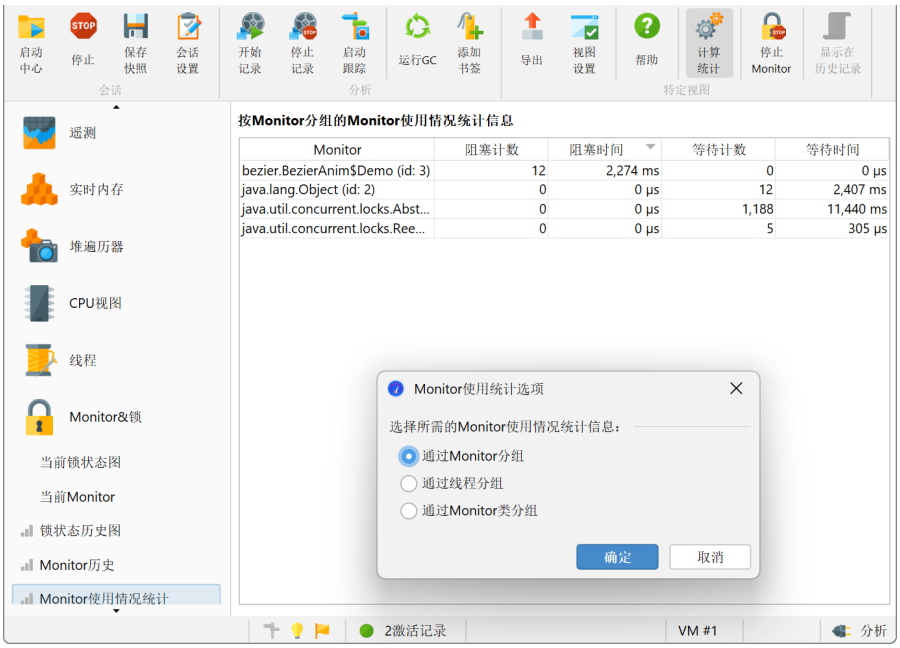
阻塞操作通常是短暂的，但在发生死锁时，这两个视图将显示问题的永久视图。此外，当前锁定图显示导致死锁的线程和监视器为红色，因此您可以立即发现此类问题。

进行新的监视器转储将替换两个视图中的数据。您还可以通过“Trigger monitor dump”触发器操作或通过API触发监视器转储。



监视器使用统计

为了从更高的角度调查阻塞和等待操作，监视器统计视图从监视器记录数据中计算报告。您可以按监视器、线程名称或监视器类对监视器事件进行分组，并分析每行的累积计数和持续时间。



探针

CPU 和内存分析主要关注对象和方法调用，这是 JVM 上应用程序的基本构建块。对于某些技术，需要一种更高级别的方法来从运行的应用程序中提取语义数据并在分析器中显示。

最显著的例子是使用 JDBC 对数据库调用进行分析。调用树显示您何时使用 JDBC API 以及这些调用需要多长时间。然而，不同的SQL语句可能会为每个调用执行，您不知道哪些调用导致了性能瓶颈。此外，JDBC 调用通常源自应用程序中的许多不同位置，因此重要的是要有一个显示所有数据库调用的单一视图，而不是在通用调用树中搜索它们。

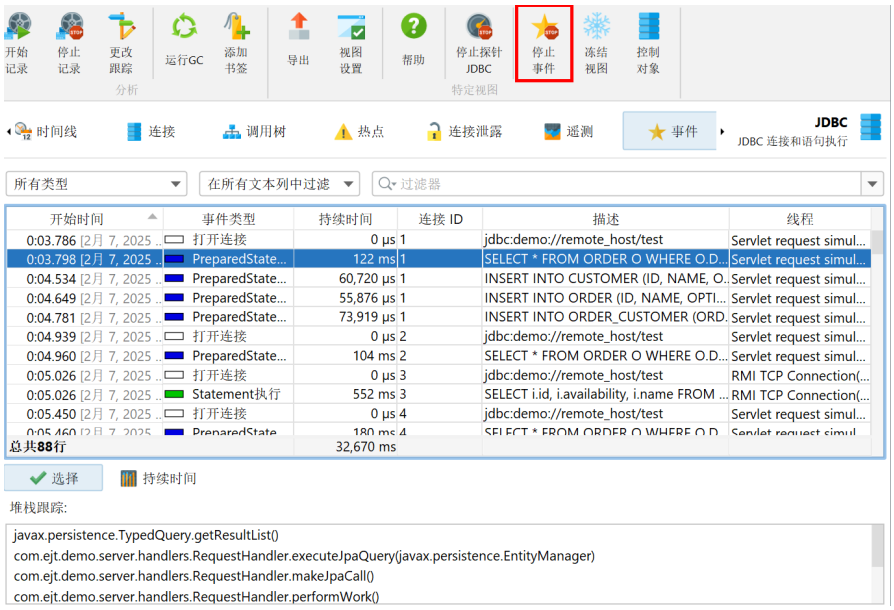
为了解决这个问题，JProfiler为JRE中的重要子系统提供了许多探针。探针在特定类中添加了检测，以收集它们的数据并在“数据库”和“JEE＆Probes”视图部分的专用视图中显示。此外，探针可以将数据注释到调用树中，以便您可以同时看到通用 CPU 分析和高级数据。

如果您有兴趣获取 JProfiler 不直接支持的技术的更多信息，您可以为其编写自己的探针 [p. 149]。一些库、容器或数据库驱动程序可能会附带自己的嵌入式探针 [p. 154]，当您的应用程序使用它们时，这些探针会在 JProfiler 中可见。

探针事件

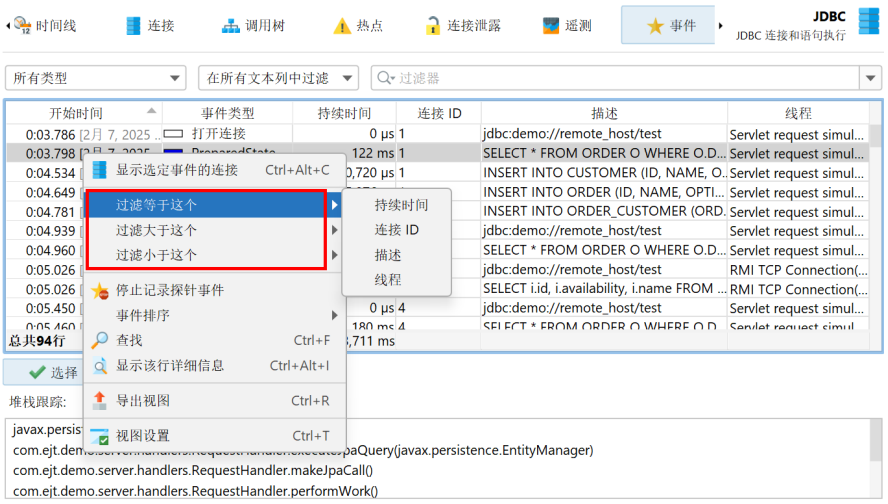
由于探针会增加开销，因此默认情况下不会记录它们，但您必须为每个探针单独开始记录 [p. 27]，可以手动或自动进行。

根据探针的功能，探针数据会显示在多个视图中。最低级别是探针事件。其他视图显示累积探针事件的数据。默认情况下，即使正在记录探针，也不会保留探针事件。当单个事件变得重要时，您可以在探针事件视图中记录它们。对于某些探针，如文件探针，通常不建议这样做，因为它们通常以高频率生成事件。其他探针，如“HTTP 服务器”探针或 JDBC 探针，可能会以较低的频率生成事件，因此记录单个事件可能是合适的。



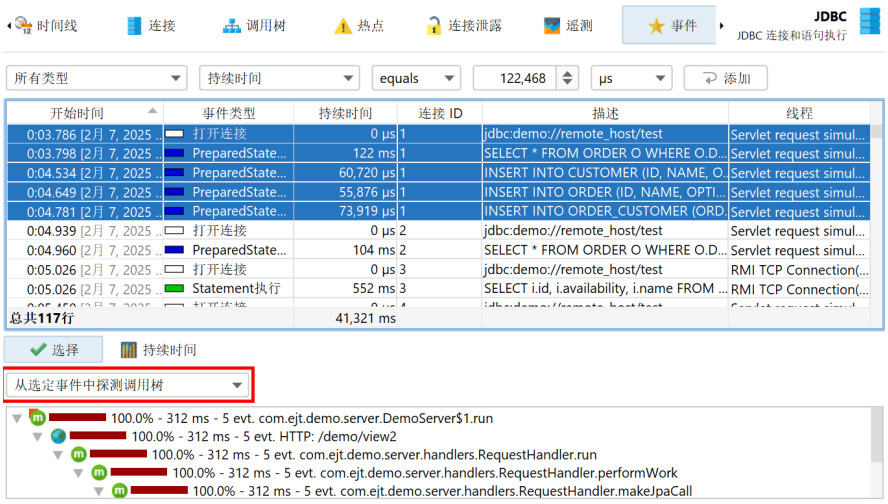
探针事件从多种来源捕获探针字符串，包括方法参数、返回值、检测对象和抛出的异常。探针可以从多个方法调用中收集数据，例如，JDBC 探针必须拦截所有预处理语句的 setter 调用以构建实际的 SQL 字符串。探针字符串是探针测量的高级子系统的基本信息。此外，事件包含开始时间、可选的持续时间、关联的线程和堆栈跟踪。

在表格的底部，有一行特殊行显示显示的事件总数并汇总表格中的所有数字列。对于默认列，这仅包括持续时间列，结合表格上方的过滤器选择器，您可以分析所选事件子集的收集数据。默认情况下，文本过滤器适用于所有文本字段列，但您可以从文本字段前的下拉菜单中选择特定的过滤列。过滤选项也可以从上下文菜单中获得，例如，过滤所有持续时间大于所选事件的事件。

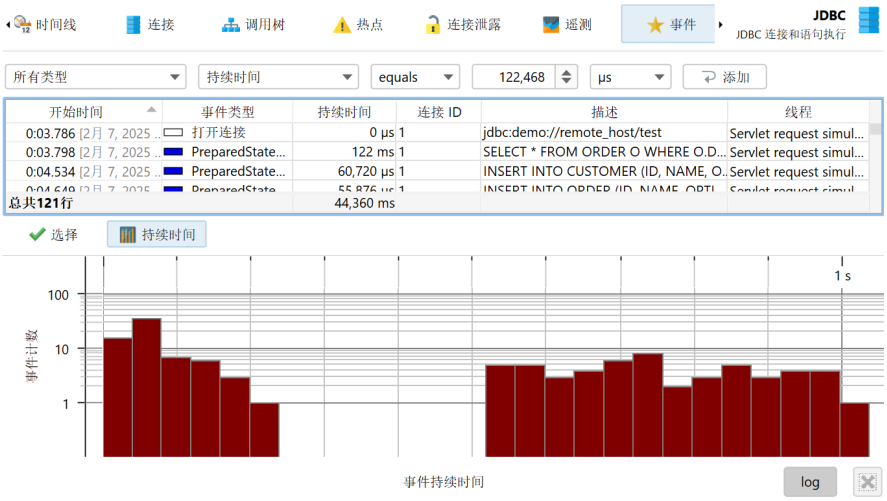


其他探针视图也提供过滤探针事件的选项：在探针遥测视图中，您可以选择一个时间范围；在探针调用树视图中，您可以过滤来自所选调用栈的事件；探针热点视图提供基于所选回溯或热点的探针事件过滤器；控制对象和时间线视图提供操作以过滤所选控制对象的探针事件。

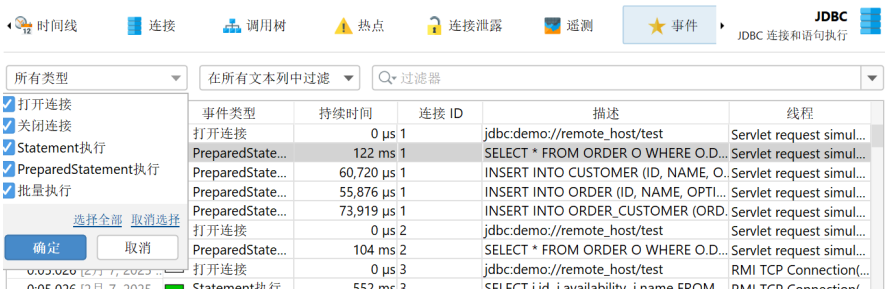
所选探针事件的堆栈跟踪显示在底部。如果选择了多个探针事件，则堆栈跟踪会累积并显示为调用树、带有回溯的探针热点或带有回溯的 CPU 热点。



在堆栈跟踪视图旁边，显示事件持续时间的直方图视图和可选的记录吞吐量。您可以使用鼠标在这些直方图中选择一个持续时间范围，以便在上面的表格中过滤探针事件。



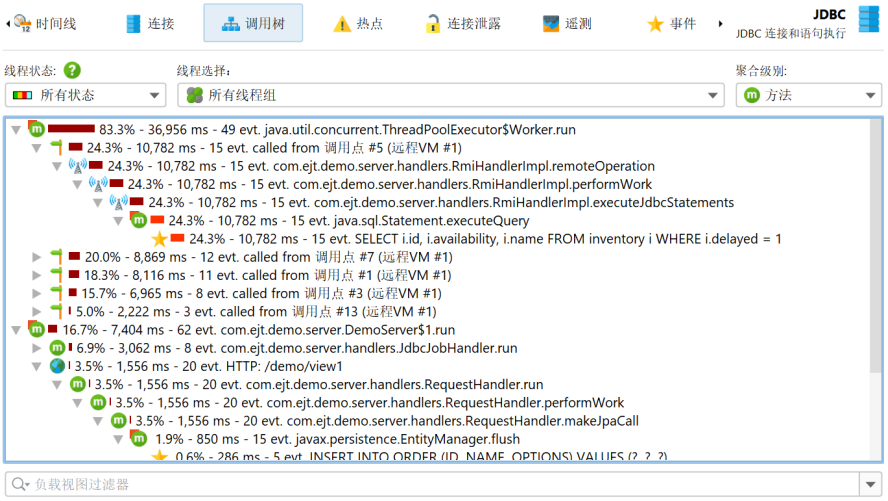
探针可以记录不同类型的活动，并将事件类型与其探针事件关联。例如，JDBC 探针显示语句、预处理语句和批处理执行作为具有不同颜色的事件类型。



为了防止在记录单个事件时过度使用内存，JProfiler会合并事件。事件上限在分析设置中配置，并适用于所有探针。仅保留最新的事件，较旧的事件会被丢弃。这种合并不会影响高级视图。

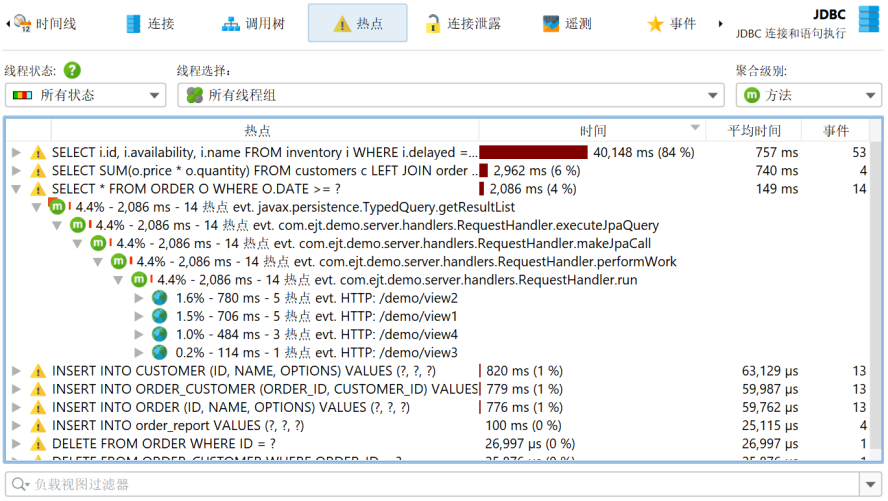
探针调用树和热点

探针记录与 CPU 记录密切配合。探针事件被聚合到探针调用树中，其中探针字符串是叶节点，称为“有效负载”。仅包含创建了探针事件的调用栈才会包含在该树中。方法节点上的信息指的是记录的有效负载名称。例如，如果在特定调用栈上执行了 42 次 SQL 语句，总时间为 9000 毫秒，这会将事件计数 42 和时间 9000 毫秒添加到所有祖先调用树节点。所有记录的有效负载的累积形成调用树，显示哪些调用路径消耗了大部分探针特定的时间。探针树的重点是有效负载，因此视图过滤器默认搜索有效负载，尽管其上下文菜单也提供过滤类的模式。



如果关闭 CPU 记录，回溯将仅包含一个“未记录 CPU 数据”节点。如果仅部分记录了 CPU 数据，可能会混合这些节点与实际回溯。即使启用了采样，JProfiler默认情况下也会记录探针有效负载的精确调用跟踪。如果您想避免这种开销，可以在分析设置中关闭它。探针记录还有其他几个调整选项，可以调整以增加数据收集或减少开销。

可以从探针调用树中计算热点。热点节点现在是有效负载，而不是CPU 中的方法调用。这通常是探针最直接有用的视图。如果 CPU 记录处于活动状态，您可以打开顶级热点并分析方法回溯，就像在常规 CPU 热点视图中一样。回溯节点上的数字指示沿从最深节点到热点下方节点的调用栈测量了多少探针事件及其总持续时间。



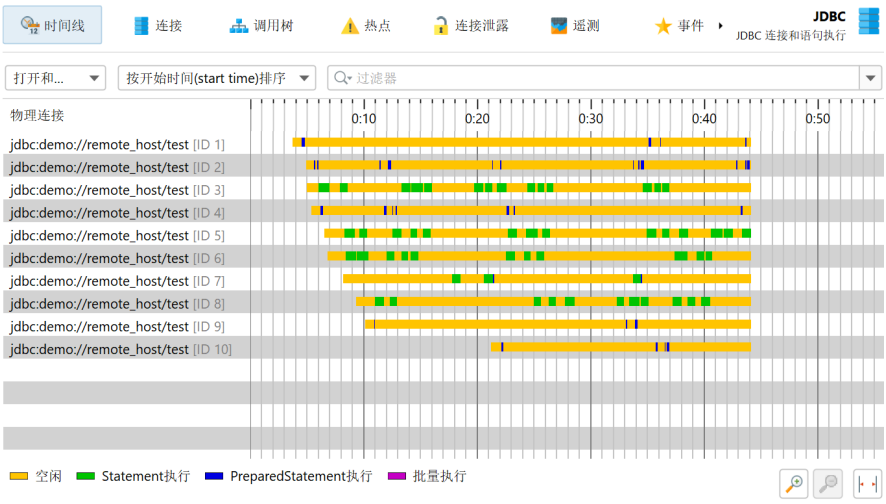
探针调用树和探针热点视图都允许您选择线程或线程组、线程状态和方法节点的聚合级别，就像在相应的 CPU 视图中一样。当您从 CPU 视图中比较数据时，重要的是要记住探针视图中的默认线程状态是“所有状态”，而不是 CPU 视图中的“可运行”。这是因为探针事件通常涉及外部系统，如数据库调用、socket 操作或进程执行，重要的是查看总时间，而不仅仅是当前 JVM 在其上花费的时间。

控制对象

许多提供对外部资源访问的库为您提供一个连接对象，您可以使用它与资源进行交互。例如，当启动一个进程时，`java.lang.Process`对象允许您从输出流中读取并写入输入流。使用JDBC时，您需要一个 `java.sql.Connection` 对象来执行 SQL 查询。在 JProfiler 中用于此类对象的通用术语是“控制对象”。

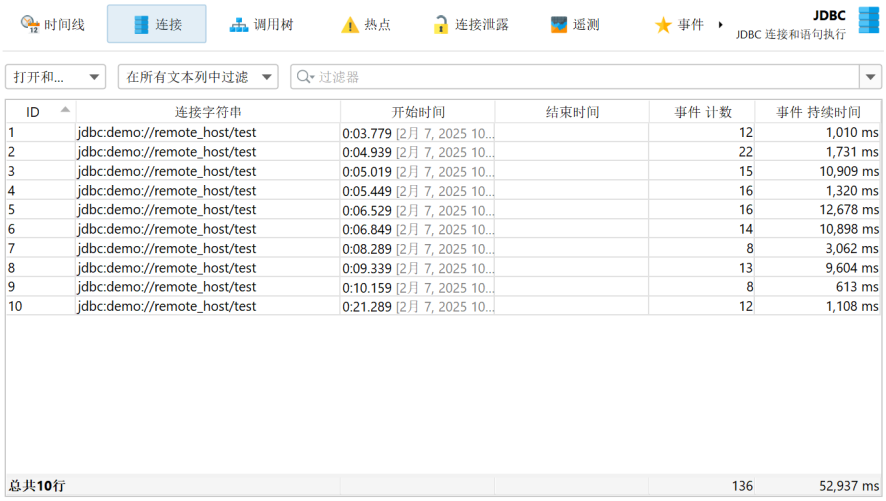
将探针事件与其控制对象分组并显示其生命周期可以帮助您更好地理解问题的来源。此外，创建控制对象通常很昂贵，因此您需要确保您的应用程序不会创建太多并正确关闭它们。为此，支持控制对象的探针具有“时间线”和“控制对象”视图，其中后者可能更具体地命名，例如，JDBC 探针的“连接”。当控制对象打开或关闭时，探针会创建特殊的探针事件，这些事件显示在事件视图中，以便您可以检查关联的堆栈跟踪。

在时间线视图中，每个控制对象显示为一个条，其着色显示控制对象何时处于活动状态。探针可以记录不同的事件类型，时间线会相应地着色。此状态信息不是从事件列表中获取的，事件列表可能已合并或甚至不可用，而是每 100 毫秒从最后一个状态中采样。控制对象有一个名称，允许您识别它们。例如，文件探针使用文件名创建控制对象，而 JDBC 探针显示连接字符串作为控制对象的名称。



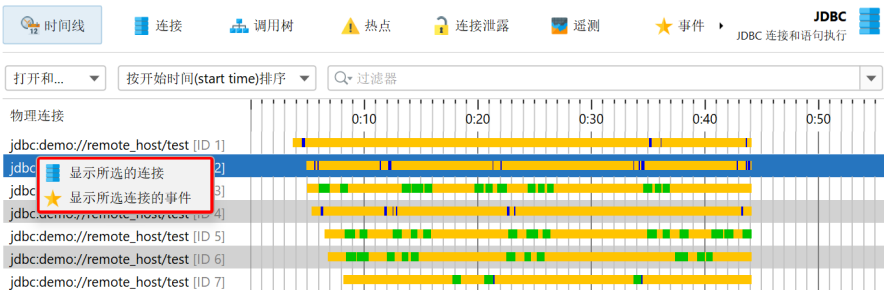
控制对象视图以表格形式显示所有控制对象。默认情况下，打开和关闭的控制对象都存在。您可以使用顶部的控件将显示限制为仅打开或关闭的控制对象，或过滤特定列的内容。除了控制对象的基本生命周期数据外，表格还显示每个控制对象的累积活动数据，例如事件计数和平均事件持续时间。

不同的探针在此处显示不同的列，例如，进程探针显示用于读取和写入事件的单独列集。如果禁用单个事件记录，此信息也可用。就像事件视图一样，底部的总行可以与过滤一起使用，以获取部分控制对象集的累积数据。



探针可以在嵌套表中发布某些属性。这是为了减少主表中的信息过载，并为表列提供更多空间。如果存在嵌套表，例如文件和进程探针，每行左侧都有一个展开句柄，可以在原地打开属性值表。

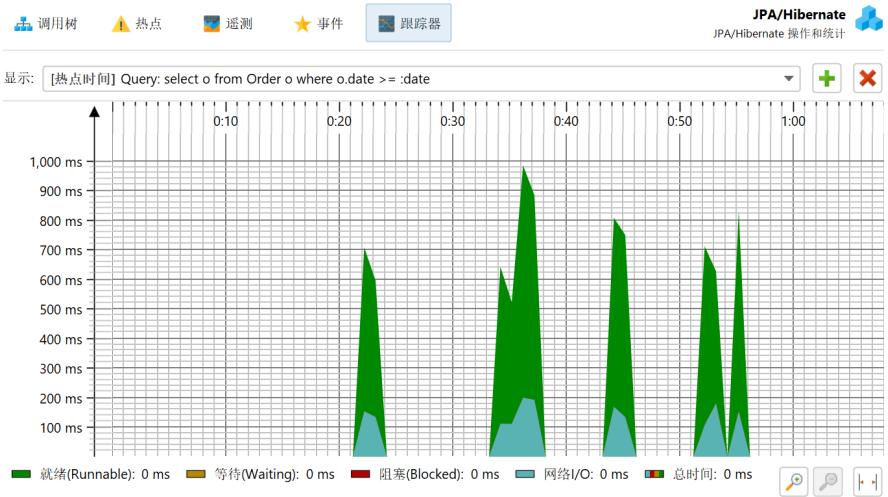
时间线、控制对象视图和事件视图通过导航操作连接。例如，在时间线视图中，您可以右键单击一行并跳转到其他视图，以便仅显示所选控制对象的数据。这是通过将控制对象 ID 过滤到所选值来实现的。



遥测和跟踪器

从探针收集的累积数据中记录了多个遥测。对于任何探针，每秒的探针事件数和探针事件的一些平均度量（如平均持续时间或 I/O 操作的吞吐量）都是可用的。对于具有控制对象的探针，打开的控制对象数量也是一个标准遥测。每个探针可以添加额外的遥测，例如，JPA 探针显示查询计数和实体操作计数的单独遥测。

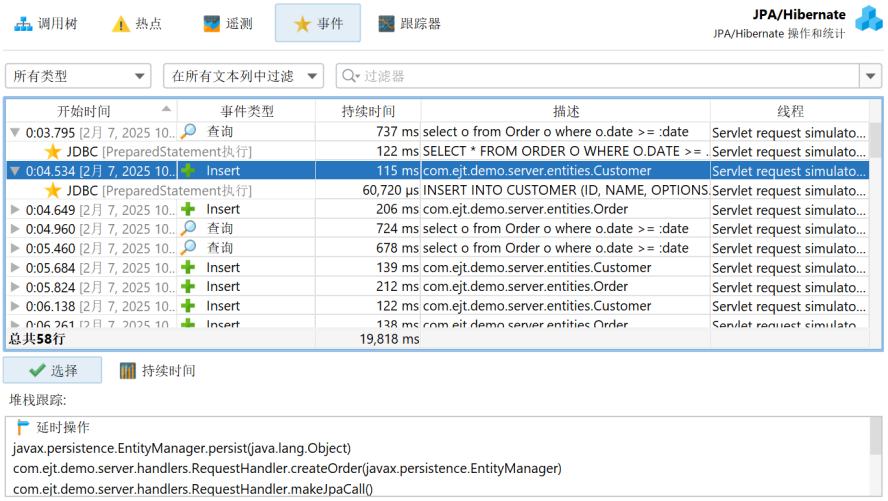
热点视图和控制对象视图显示累积数据，这些数据可能在一段时间内跟踪很有趣。这些特殊的遥测由探针跟踪器记录。设置跟踪的最简单方法是从热点或控制对象视图中使用添加选择到跟踪器操作添加新的遥测。在这两种情况下，您都必须选择是要跟踪时间还是计数。当跟踪控制对象时，遥测是所有不同探针事件类型的堆叠区域图。对于跟踪的热点，跟踪的时间被分成不同的线程状态。



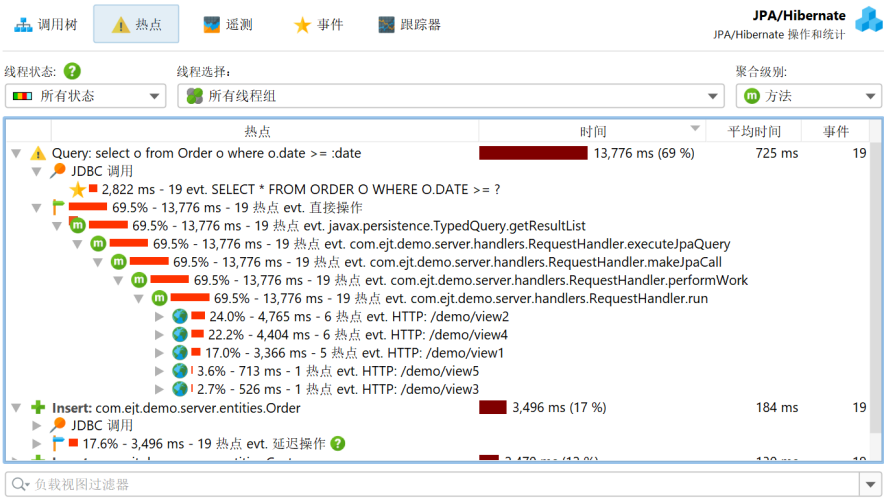
探针遥测可以添加到“遥测”部分 [p. 44]，以便将它们与系统遥测或自定义遥测进行比较。然后，您还可以通过遥测概览中的上下文菜单操作控制探针记录。

JDBC 和 JPA

JDBC 和 JPA 探针协同工作。在 JPA 探针的事件视图中，您可以展开单个事件以查看关联的 JDBC 事件（如果 JDBC 探针与 JPA 探针一起记录）。



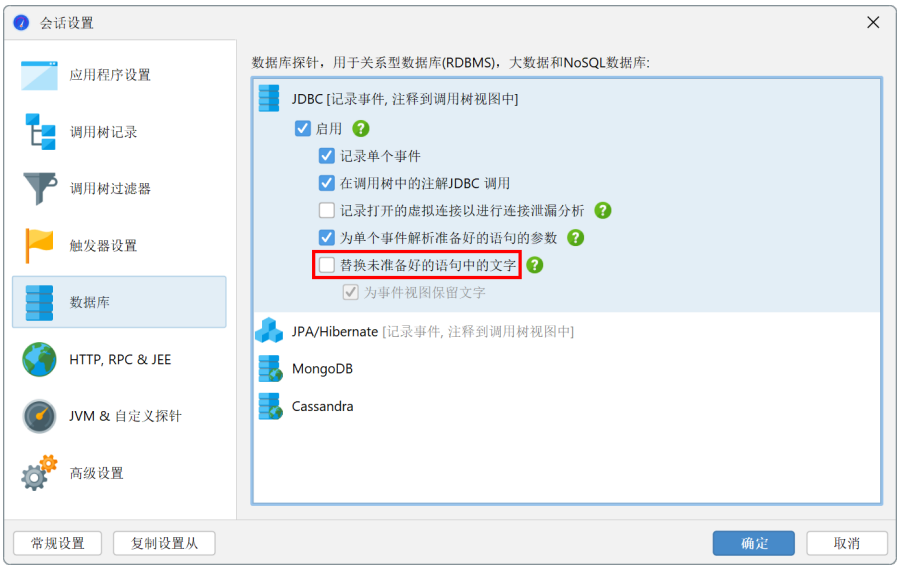
类似地，热点视图向所有热点添加一个特殊的“JDBC 调用”节点，其中包含由 JPA 操作触发的 JDBC 调用。一些 JPA 操作是异步的，并不会立即执行，而是在会话刷新时的某个任意时间点执行。在寻找性能问题时，该刷新堆栈跟踪没有帮助，因此 JProfiler 记住了现有实体的获取位置或新实体的持久化位置的堆栈跟踪，并将它们与探针事件联系起来。在这种情况下，热点的回溯包含在标记为“延迟操作”的节点中，否则会插入一个“直接操作”节点。



其他探针（如 MongoDB 探针）支持直接和异步操作。异步操作不是在当前线程上执行，而是在同一 JVM 中的一个或多个其他线程上或在另一个进程中执行。对于此类探针，热点中的回溯被排序到“直接操作”和“异步操作”容器节点中。

JDBC 探针中的一个特殊问题是，如果 SQL 字符串中包含字面数据（如 ID），您只能获得良好的热点。如果使用预处理语句，这将自动发生，但如果执行常规语句，则不会。在后一种情况下，您可能会得到一个热点列表，其中大多数查询仅执行一次。作为补救措施，JProfiler 在 JDBC 探针配置中提

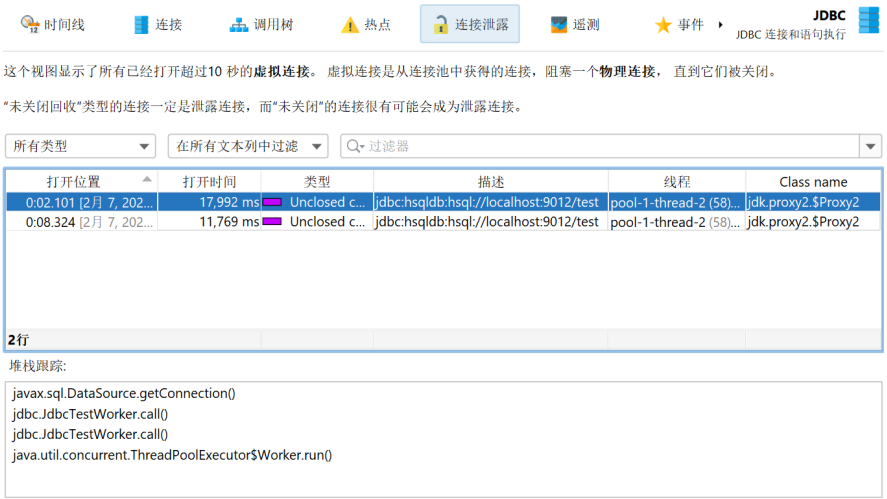
供了一个非默认选项，用于替换未准备好的语句中的字面量。出于调试目的，您可能仍然希望在事件视图中看到字面量。禁用该选项可以减少内存开销，因为 JProfiler 不必缓存那么多不同的字符串。



另一方面，JProfiler收集预处理语句的参数，并在事件视图中显示完整的SQL字符串而不带占位符。同样，这在调试时很有用，但如果您不需要它，可以在探针设置中关闭它以节省内存。

JDBC 连接泄漏

JDBC 探针具有“连接泄漏”视图，显示未返回到其数据库池的打开虚拟数据库连接。这仅影响由池化数据库源创建的虚拟连接。虚拟连接会阻塞物理连接，直到它们关闭。



有两种类型的泄漏候选者，“未关闭”连接和“未关闭收集”连接。两种类型都是虚拟连接，数据库池发放的连接对象仍在堆上，但尚未调用`close()`。已被垃圾收集的“未关闭收集”连接是确定的连接泄漏。

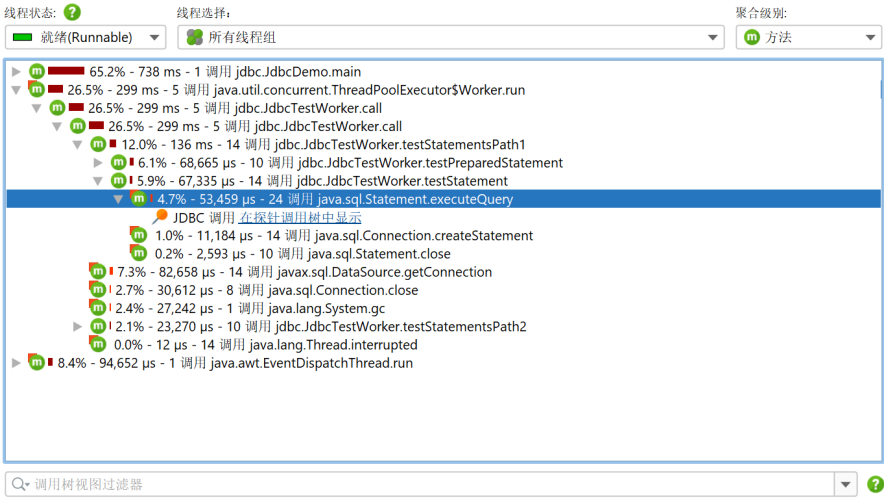
“未关闭”连接对象仍在堆上。自打开以来的持续时间越长，这样的虚拟连接就越可能是泄漏候选者。当虚拟连接已打开超过 10 秒时，它被视为潜在泄漏。然而，`close()` 仍然可以在其上调用，然后“连接泄漏”视图中的条目将被移除。

连接泄漏表包括一个类名列，显示连接类的名称。这将告诉您哪个类型的池创建了连接。JProfiler明确支持大量数据库驱动程序和连接池，并知道哪些类是虚拟和物理连接。对于未知的池或数据库驱动程序，JProfiler可能会误将物理连接视为虚拟连接。由于物理连接通常是长寿命的，因此它会显示在“连接泄漏”视图中。在这种情况下，连接对象的类名将帮助您将其识别为误报。

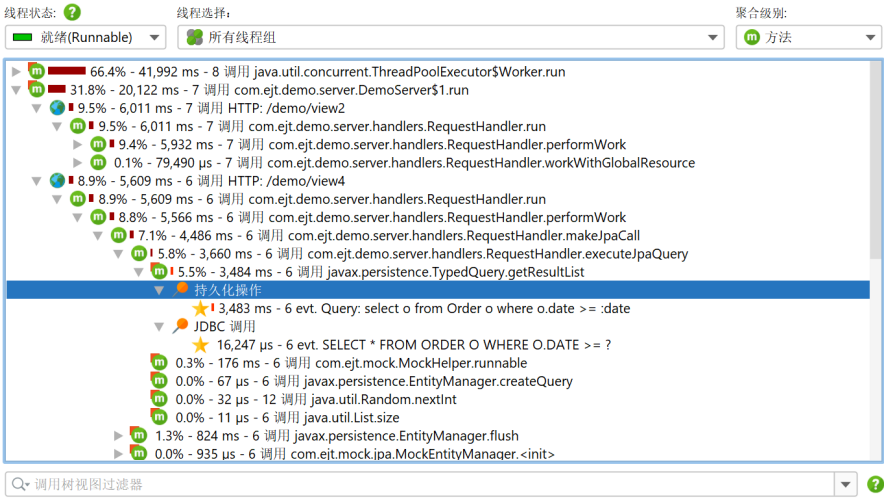
默认情况下，当您开始探针记录时，连接泄漏分析未启用。在连接泄漏视图中有一个单独的记录按钮，其状态对应于 JDBC 探针设置中的记录打开的虚拟连接以进行连接泄漏分析复选框。就像事件记录一样，按钮的状态是持久的，因此如果您启动分析一次，它将自动为下一个探针记录会话启动。

调用树中的有效负载数据

在查看 CPU 调用树时，查看探针记录的有效负载数据的位置很有趣。这些数据可能会帮助您解释测量的 CPU 时间。这就是为什么许多探针在 CPU 调用树中添加交叉链接的原因。例如，类加载器探针可以显示类加载触发的位置。这在调用树中是不可见的，并且可能会增加意外的开销。调用树视图中不透明的数据库调用可以通过单击进一步在相应的探针中分析。这甚至适用于调用树分析，当您单击探针链接时，分析会在探针调用树视图的上下文中自动重复。

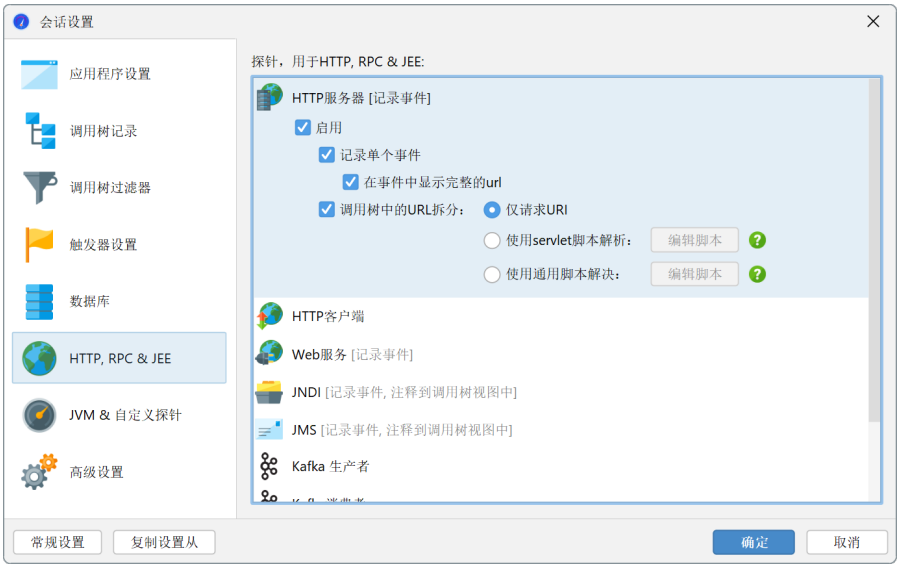


另一种可能性是直接在 CPU 调用树中内联显示有效负载信息。所有相关探针在其配置中都有一个在调用树中注释选项。这样就没有可用的探针调用树链接。每个探针都有自己的有效负载容器节点。具有相同有效负载名称的事件被聚合，并显示调用次数和总时间。有效负载名称在每个调用栈的基础上合并，最旧的条目被聚合到一个“[早期调用]”节点中。每个调用栈记录的有效负载名称的最大数量可以在分析设置中配置。

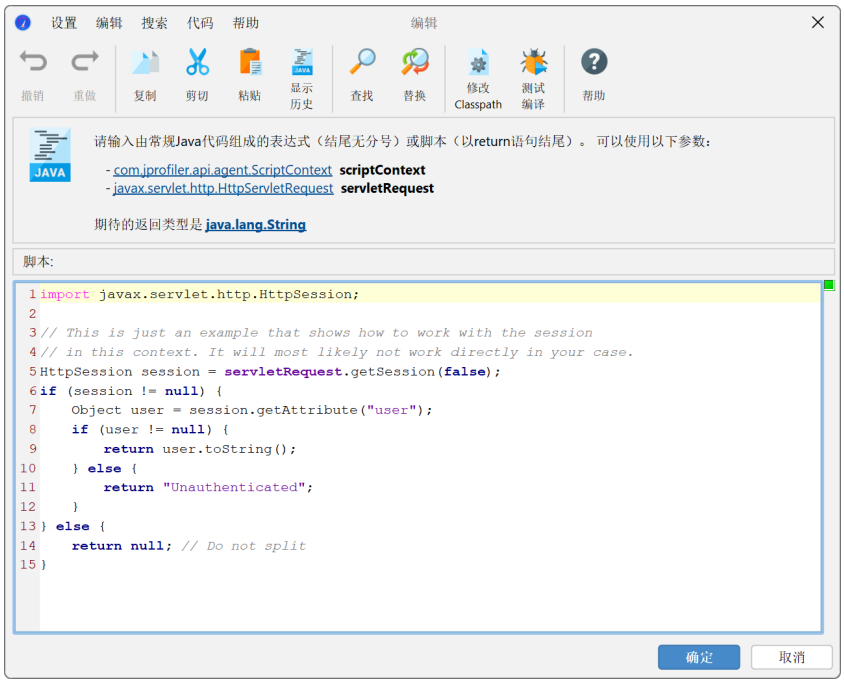


调用树拆分

一些探针不使用其探针字符串将有效负载数据注释到调用树中。相反，它们为每个不同的探针字符串拆分调用树。这对于服务器类型的探针特别有用，您希望为每种不同类型的传入请求分别查看调用树。“HTTP 服务器”探针拦截 URL，并为您提供对 URL 的哪些部分用于拆分调用树的细粒度控制。默认情况下，它仅使用请求 URI 路径而不带任何参数。

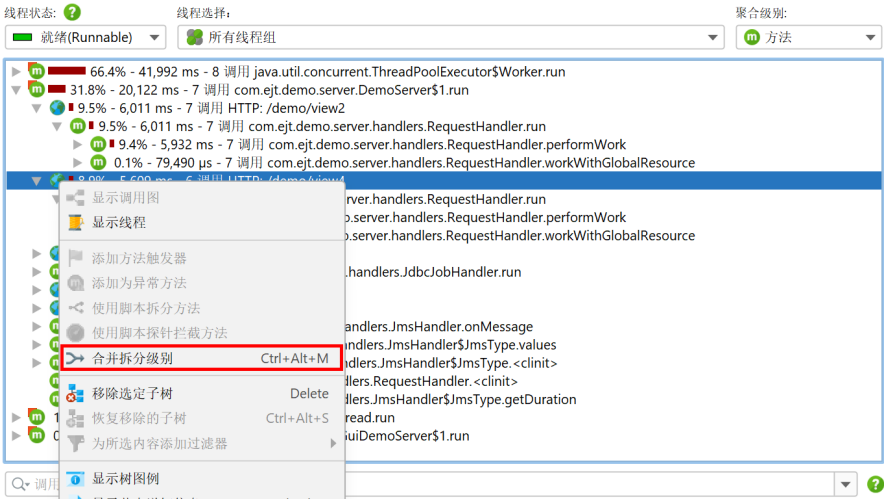


为了更灵活，您可以定义一个脚本来确定拆分字符串。在脚本中，您可以将当前`javax.servlet.http.HttpServletRequest`作为参数并返回所需的字符串。



更重要的是，您不仅限于单个拆分级别，还可以定义多个嵌套拆分。例如，您可以先按请求 URI 路径拆分，然后按从 HTTP 会话对象中提取的用户名拆分。或者，您可以按请求方法对请求进行分组，然后按请求 URI 拆分。

通过使用嵌套拆分，您可以在调用树中查看每个级别的单独数据。在查看调用树时，某个级别可能会妨碍您，您可能需要在“HTTP 服务器”探针配置中消除它。更方便的是，在不丢失记录数据的情况下，您可以通过在相应的拆分节点上使用上下文菜单，动态合并和取消合并调用树中的拆分级别。



拆分调用树可能会导致相当大的内存开销，因此应谨慎使用。为了避免内存过载，JProfiler限制了拆分的最大数量。如果特定拆分级别的拆分上限已达到，将添加一个特殊的“[受限节点]”拆分节点，并带有一个超链接以重置上限计数器。如果默认上限对您的用途来说太低，您可以在分析设置中增加它。

垃圾收集器分析

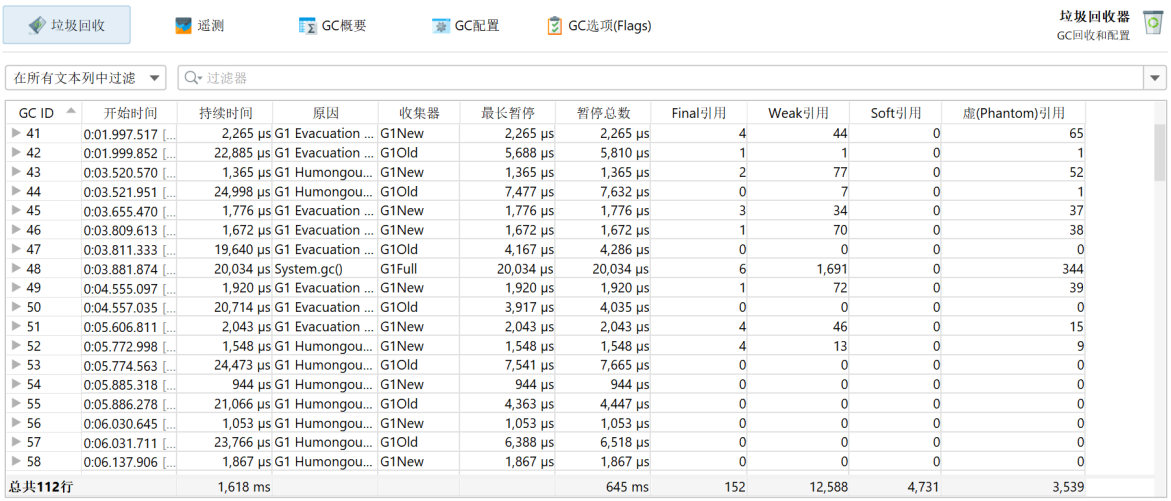
理解和分析垃圾收集器（GC）的运行时特性很重要。首先，GC暂停会直接影响应用程序的响应性。通过了解垃圾收集器的性能，可以优化其设置以减少这些暂停。 通常，频繁且长时间的GC周期可能表明堆太小，或者创建了太多临时对象。

借助垃圾收集器探针，可以解决这些问题，并在调整JVM设置时做出更明智的决策，例如选择合适的垃圾收集器、堆大小或其他JVM参数。

垃圾收集器探针与其他探针有不同的视图，并使用不同的数据源。 它不是从JVM的分析接口获取数据，而是使用JFR流来分析来自JDK Flight Recorder [1]的GC相关事件。 由于依赖于JFR事件流，GC探针仅在分析Java 17或更高版本的Hotspot JVM时可用。 当你打开JFR快照 [p. 205]时，无论使用的Java版本如何，都会提供相同的探针。

垃圾收集视图

垃圾收集器探针中的主要视图是“垃圾收集”表。它将所有记录的垃圾收集显示为行，并将其最重要的指标显示为列。



GC ID	开始时间	持续时间	原因	收集器	最长暂停	暂停总数	Final引用	Weak引用	Soft引用	虚(Phantom)引用
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
总共112行		1,618 ms				645 ms	152	12,588	4,731	3,539

“原因”列显示了触发垃圾收集的原因。例如，调用`System.gc()`触发了完整的垃圾收集。 你可以从“收集器”列中关联的“G1Full”值中看到这一点。它还导致了20毫秒的显著暂停，这就是为什么通常不建议调用`System.gc()`。其他原因触发了年轻代空间的收集（“G1New”）或G1收集器的旧GC收集（“G1Old”），清理旧代中未引用的对象。 你可以看到旧GC收集的时间通常比年轻代收集的时间长，尽管年轻代收集的对象更多。

具有特殊GC处理的收集引用在单独的列中显示为“final”、“weak”、“soft”和“phantom”引用。

之所以有最长暂停和暂停总和的单独列，是因为每次垃圾收集由多个阶段组成，这些阶段会产生单独的暂停。此外，垃圾收集的“持续时间”不等于暂停总和，因为垃圾收集在执行时仅部分暂停JVM。你可以看到截图中的“G1Old”收集仅暂停了其持续时间的五分之一。

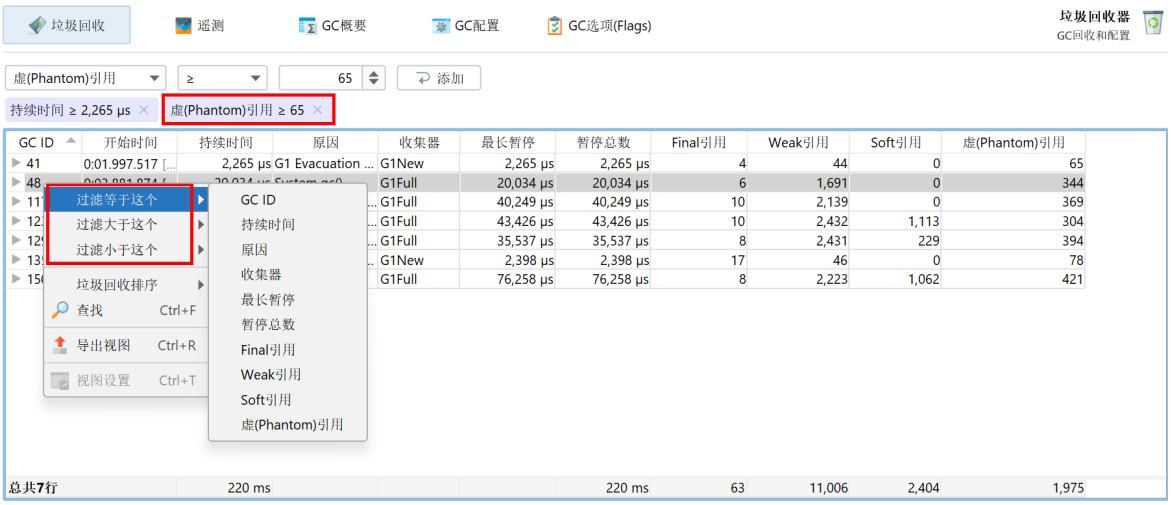
要检查垃圾收集的各个阶段，可以切换“GC ID”列中的树图标。

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

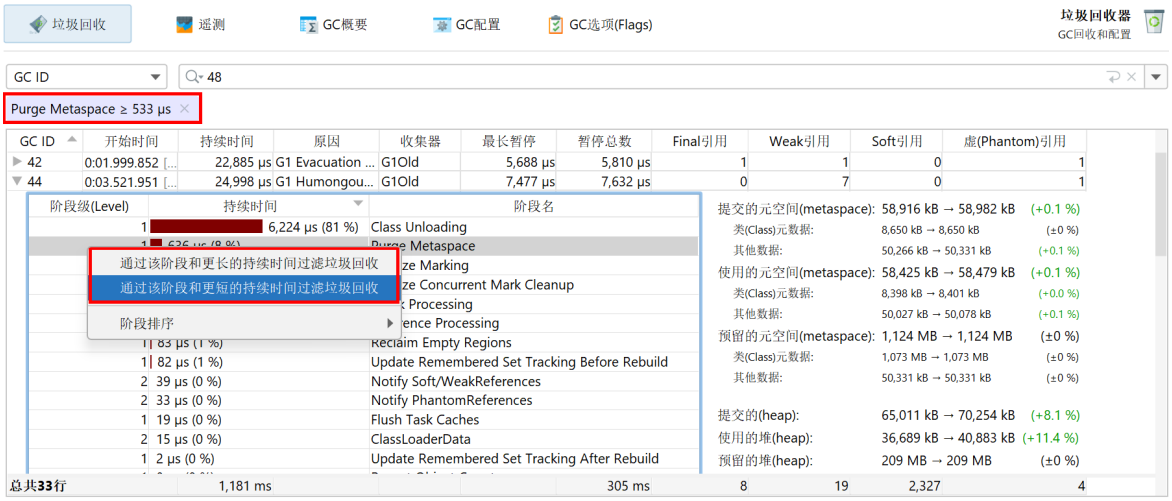
在上面的截图中，G1收集器的混合GC收集（“G1Old”）已展开。你可以看到大部分时间花在“类卸载”上，这不会暂停JVM。在右侧，你可以看到垃圾收集的进一步统计信息。在这里，使用的堆保持不变，而使用的元空间上升了0.1%。

每个收集器的阶段是不同的。在上面的截图中，显示了一个完整的收集。它花费了大量时间标记整个堆中的活动对象。 在收集结束时，使用的堆减少了15.7%，而元空间保持不变。

在分析垃圾收集时，**过滤**是比较不同垃圾收集子集的重要工具。在表的顶部，有一个过滤器选择器，可以让你选择任何列并配置相应的过滤器。 查看相似垃圾收集的更简单方法是使用表上的上下文菜单，并根据选定行中的列值选择过滤条件。

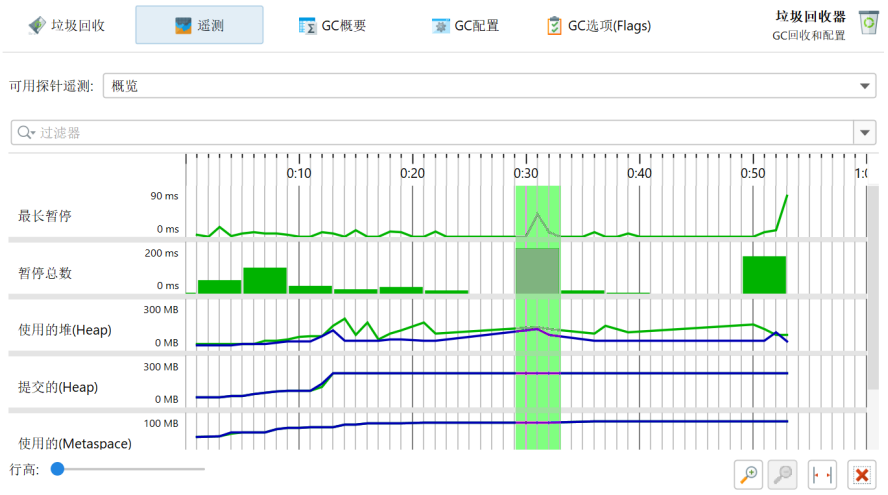


你可以添加多个过滤器以缩小感兴趣的垃圾收集范围。活动过滤器显示为表顶部的标签。 也可以从嵌套的GC阶段表中添加过滤器。

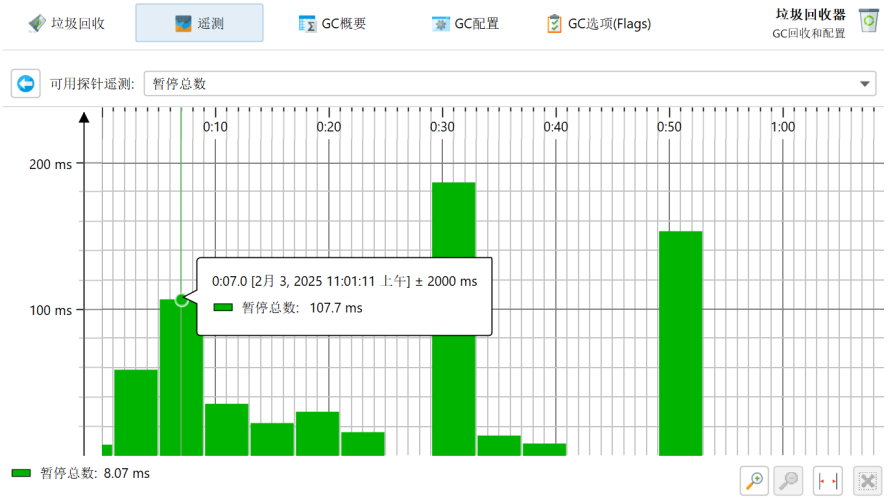


遥测

GC探针生成了一些遥测数据，这些数据在“遥测”探针视图中可用。



如果你对最小化GC暂停感兴趣，顶部的“最长暂停”遥测将是最有趣的。 你可以沿着遥测的时间轴拖动以在“垃圾收集”视图中选择相应的垃圾收集。为了获得更好的垂直分辨率，你可以从顶部的下拉菜单中选择单个遥测，或通过单击遥测的名称。



在上面的截图中，你可以看到随时间的暂停总和。JProfiler通过构建**记录数据的直方图**来呈现可累加的测量值。 直方图的宽度取决于可用的水平空间，因此直方图的宽度会根据缩放级别和窗口的宽度（如果启用了“适应缩放”）而变化。 保持不变的是所有直方图下的总面积。

堆和元空间遥测基于你在展开垃圾收集时可以看到的统计数据。这意味着数据不像完整分析会话中的内存遥测那样定期采样。 如果在某个时间段内没有发生垃圾收集，就不会有数据。 对于分配活动较少的JVM，沿时间轴可能会有长时间的图形仅在两个垃圾收集之间插值。

这些遥测中的每一个都有两条数据线：“GC前”和“GC后”。对于“使用的堆”遥测，差异通常很大。 在每个时间点，你可以通过比较两条数据线的值来查看垃圾收集完成了多少工作。 你可以查看工具提示以获取精确的值。对于“提交的堆”遥测和元空间遥测，两条线之间的差异通常很小。

如果你正在分析JFR快照 [p. 205]，则在遥测部分的“内存”遥测中也使用了相同的 `jdk.GCHeapSummary` JFR事件类型的数据。 但在这种情况下，“GC前”和“GC后”值都显示在同一条数据线上，数据不会像GC探针遥测那样聚合到每秒一次的粒度，因此图形会有所不同。

GC摘要

GC摘要显示了在整个记录期间聚合的测量值。 每个测量值提供垃圾收集的数量，以及平均值、最大值和总值。 最重要的数据在顶部是直接影响应用程序活性的“暂停时间”。

▼ 暂停次数	
暂停计数	143
平均暂停	4,512 µs
最大暂停时间	76,258 µs
总暂停次数	645 ms
▼ 所有回收总次数	
平均GC时间	14,451 µs
最大GC事件	76,258 µs
总GC时间	1,618 ms
▼ 年轻代回收总次数	
GC计数	70
平均GC时间	1,712 µs
最大GC事件	3,460 µs
总GC时间	119 ms
▼ 老年代回收总次数	
GC计数	42
平均GC时间	35,682 µs
最大GC事件	76,258 µs
总GC时间	1,498 ms

另一个顶级类别显示了所有收集的总时间，然后分为年轻和旧收集的两个子类别。

GC配置

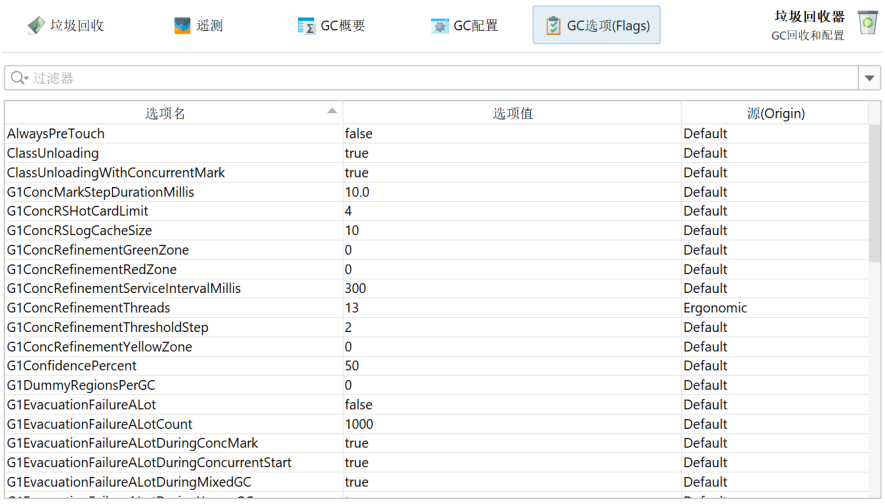
当你调整垃圾收集器时，你可能希望检查可以显式设置或由垃圾收集器本身隐式设置的常见属性。

▼ GC配置	
年轻代垃圾收集器	G1New
年老代垃圾收集器	G1Old
并发 GC 线程	3
并行 GC 线程	13
并发显式 GC	false
已禁用显式 GC	false
使用动态 GC 线程	true
GC 时间比率	12
▼ GC堆(Heap)配置	
初始大小	209 MB
最小堆大小	8,388 kB
最大堆大小	209 MB
是否使用压缩 Oops	true
压缩 Oops 模式	32-bit
堆地址大小	32
对象对齐	8 字节
▼ 年轻代配置	
最小年轻代大小	1,363 kB
最大年轻代大小	125 MB

这些属性是所有垃圾收集器共有的，帮助你理解垃圾收集器之间的差异。

GC标志

最后，GC特定的标志让你了解垃圾收集器可以调整的属性，并让你检查它们的实际值。



选项名	选项值	源(Origin)
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

“来源”列显示了标志是如何设置的。“默认”值未从标准设置中修改，而“自适应”标志已由垃圾收集器自动调整。 如果你在命令行上设置了特定的GC标志，它们将在来源中报告为“命令行”。

MBean 浏览器

许多应用服务器和框架，如 Apache Camel[1] 使用 JMX 来公开一些 MBeans 以进行配置和监控。JVM 本身也发布了一些 平台 MXBeans[2]，它们提供了有关 JVM 低级操作的有趣信息。

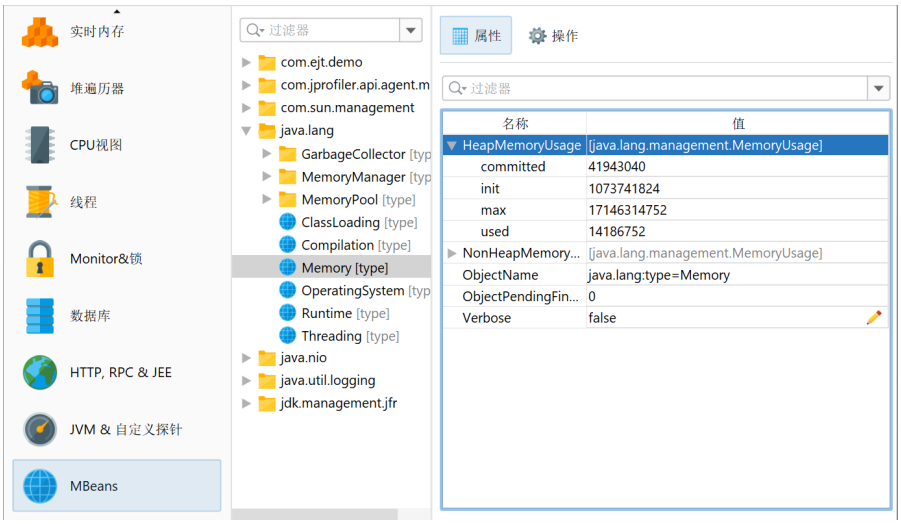
JProfiler 包含一个 MBean 浏览器，显示被分析 VM 中所有注册的 MBeans。访问 MBean 服务器的 JMX 远程管理 级别不是必需的，因为 JProfiler 代理已经在进程中运行，并且可以访问所有注册的 MBean 服务器。

JProfiler 支持 **Open MBeans** 的类型系统。除了定义一些简单类型之外，Open MBeans 可以定义不涉及自定义类的复杂数据类型。此外，数组和表格 作为数据结构是可用的。使用 **MXBeans**，JMX 提供了一种从 Java 类自动创建 Open MBeans 的简便方法。 例如，JVM 提供的 MBeans 是 MXBeans。

虽然 MBeans 没有层次结构，但 JProfiler 通过将对象域名直到第一个冒号作为第一级树节点， 并使用所有属性作为递归嵌套级别来将它们组织成树。属性值首先显示，属性键在括号中显示在末尾。`type` 属性优先显示在顶级节点下方。

属性

在显示 MBean 内容的树表的顶层，您可以看到 MBean 属性。



以下数据结构显示为嵌套行：

- **Arrays**
  原始数组和对象数组的元素显示在嵌套行中，索引作为键名。
- **Composite data**
  复合数据类型中的所有项目显示为嵌套行。每个项目可以是任意类型，因此嵌套可以继续到任意深度。
- **Tabular data**
  在 MXBeans 中，您最常遇到的是将 `java.util.Map` 实例映射到具有一个键列和一个值列的表格数据类型。如果键的类型是简单类型，则地图显示为“内联”，每个键值对显示为嵌套行。如果键具有复杂类型，则插入一个包含嵌套键和值条目的“map entry”元素级别。这也是一般表格类型的情况，具有复合键和多个值。

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

可选地，MBean 属性可以是可编辑的，在这种情况下， 编辑图标将显示在其值旁边，并且 编辑值 操作变为活动状态。复合和表格类型不能在 MBean 浏览器中编辑，但数组或简单类型是可编辑的。

如果值是可为空的，例如数组，编辑器有一个复选框来选择空状态。

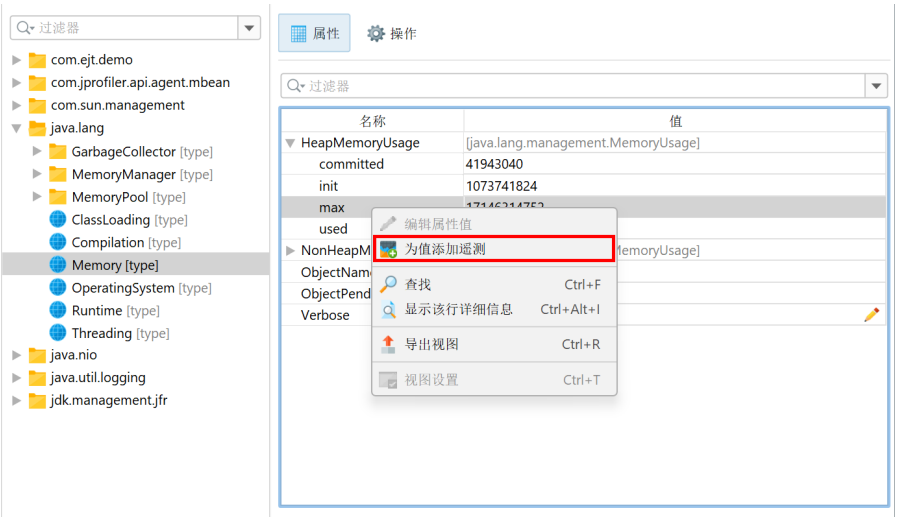
数组元素用分号分隔。可以忽略一个尾随分号，因此 1 和 1; 是等效的。分号前缺少的值将被视为对象数组的空值。对于字符串数组，您可以使用双引号 ("") 创建空元素，并通过引用整个元素来包含分号的元素。字符串元素中的双引号必须加倍。 例如，输入字符串数组值

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

创建字符串数组

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

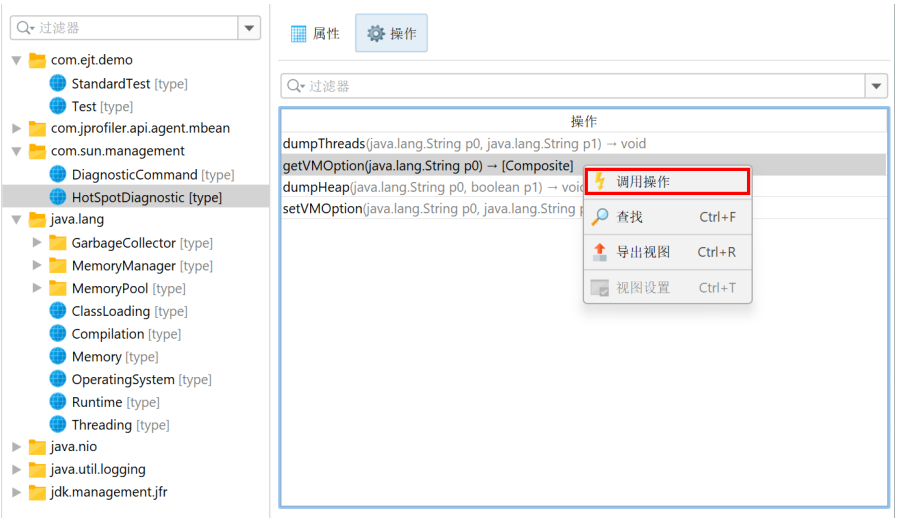
JProfiler 可以从数值 MBean 属性值创建自定义遥测。当您 定义 MBean 遥测线 [p. 44] 以获取自定义遥测时，将显示一个 MBean 属性 浏览器，允许您选择提供遥测数据的属性。当您已经在 MBean 浏览器中工作时，为值添加遥测 操作 在上下文菜单中提供了一种方便的方法来创建新的自定义遥测。



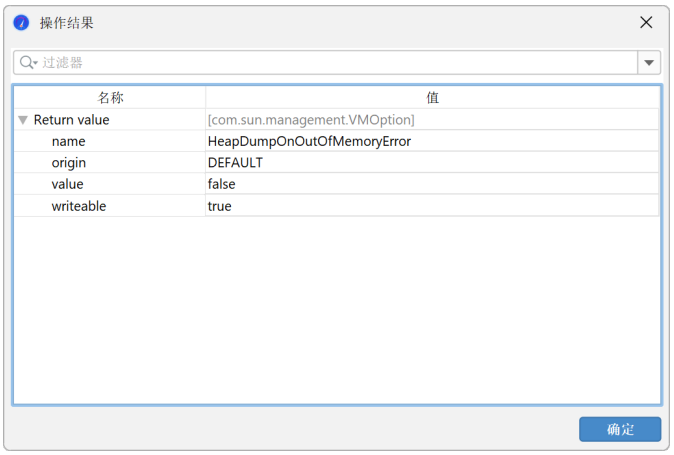
遥测还可以跟踪复合数据或具有简单键和单个值的表格数据中的嵌套值。 当您选择嵌套行时，将构建一个值路径，其中路径组件由正斜杠分隔。

操作

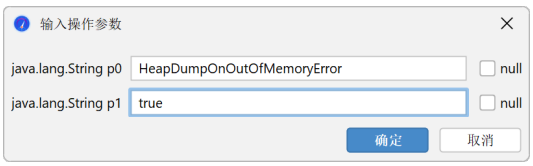
除了检查和修改MBean属性之外，您还可以调用MBean操作并检查其返回值。MBean操作是MBean接口上的方法， 不是设置器或获取器。



操作的返回值可能具有复合、表格或数组类型，因此显示一个内容类似于MBean属性树表的新窗口。对于简单返回类型，只有一个名为“返回值”的行。对于其他类型，“返回值”是根元素，结果被添加到其中。

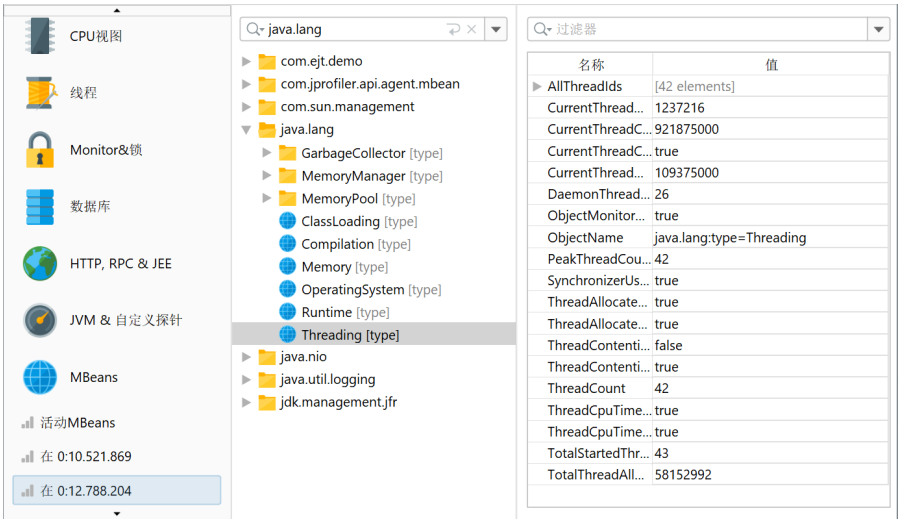


MBean 操作可以有一个或多个参数。当您输入它们时，适用与编辑 MBean 属性时相同的规则和限制。



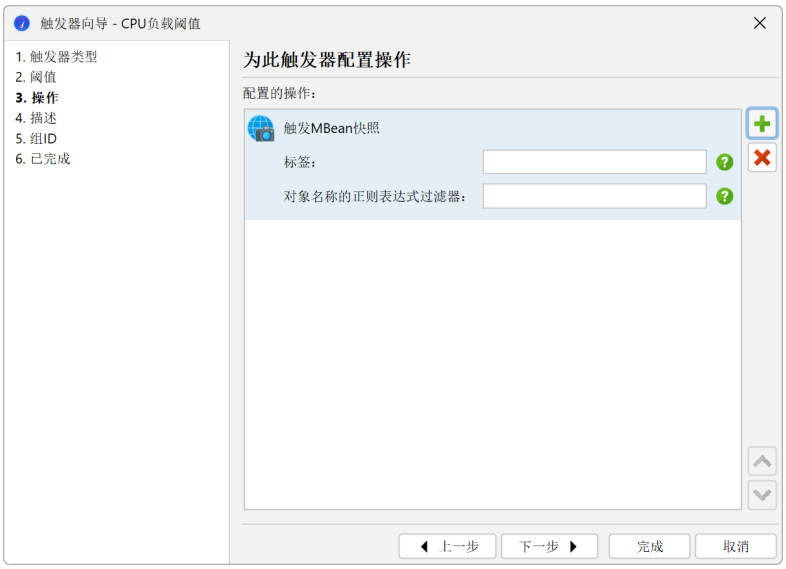
MBean 快照

除了查看 MBeans 的实时值之外，您还可以拍摄其当前状态的快照。每个新快照 将作为 MBean 视图部分中的单独视图添加，并可以分配自定义标签。拍摄快照时，仅根据当前过滤器显示的那些MBeans会被包含。通过这种方式，您可以专注于 特定的 MBeans 并减少查询与您的目的无关的 MBeans 的开销。



在 JProfiler UI 中保存快照时，所有 MBean 快照也会被保存，而实时 MBean 视图不会 被保存。对于离线分析 [p. 119]，您可以使用 Controller API 或“保存 MBean 快照” 触发器操作以编程方式拍摄 MBean 快照。

控制器 API 和触发器操作都支持在视图选择器中显示的可选标签以及 用于过滤包含的 MBeans 的可选正则表达式。



离线分析

使用 JProfiler 分析应用程序有两种根本不同的方法：默认情况下，您可以在附加 JProfiler GUI 的情况下进行分析。JProfiler GUI 为您提供了开始和停止记录的按钮，并显示所有记录的分析数据。

在某些情况下，您可能希望在没有 JProfiler GUI 的情况下进行分析，并在稍后分析结果。对于这种情况，JProfiler 提供了离线分析。离线分析允许您在没有连接 JProfiler GUI 的情况下启动被分析的应用程序。

然而，离线分析仍然需要执行一些操作。至少需要保存一个快照，否则稍后将无法获得分析数据。此外，要查看 CPU 或分配数据，您必须在某个时间点开始记录。同样，如果您希望能够在保存的快照中使用堆遍历器，您必须触发堆转储。

分析 API

解决此问题的第一个方案是控制器 API。通过 API，您可以在代码中以编程方式调用所有分析操作。在 `api/samples/offline` 目录中，有一个可运行的示例，向您展示如何在实践中使用控制器 API。在该目录中执行 `../gradlew` 以编译和运行它，并研究 Gradle 构建文件 `build.gradle` 以了解如何调用测试程序。

Controller API 是在运行时管理分析操作的主要接口。它包含在您的 JProfiler 安装的 `bin/agent.jar` 中，或者作为 Maven 依赖项，其坐标为

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

和仓库

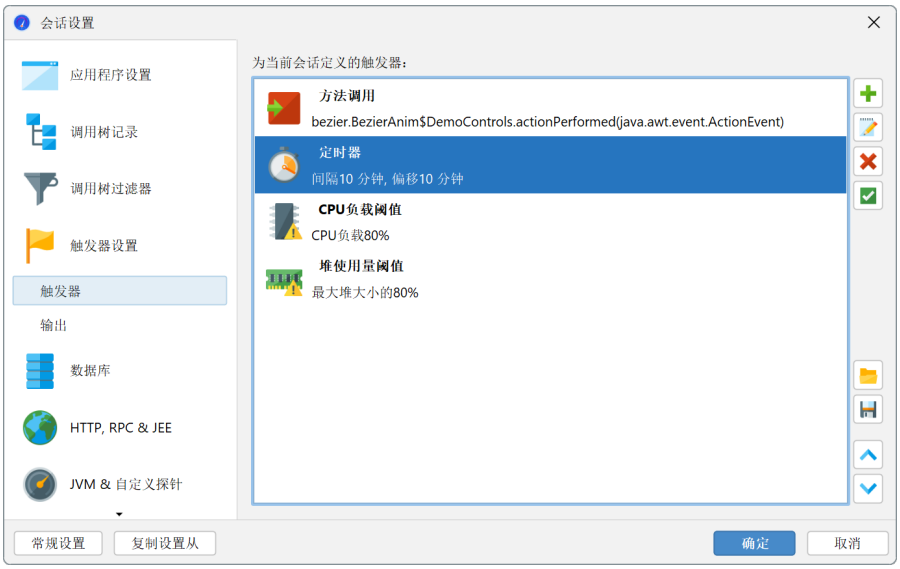
```
https://maven.ej-technologies.com/repository
```

如果在应用程序的正常执行期间使用分析 API，API 调用将静默地不执行任何操作。

这种方法的缺点是您必须在开发期间将 JProfiler 代理库添加到应用程序的类路径中，将分析指令添加到源代码中，并在每次更改编程分析操作时重新编译代码。

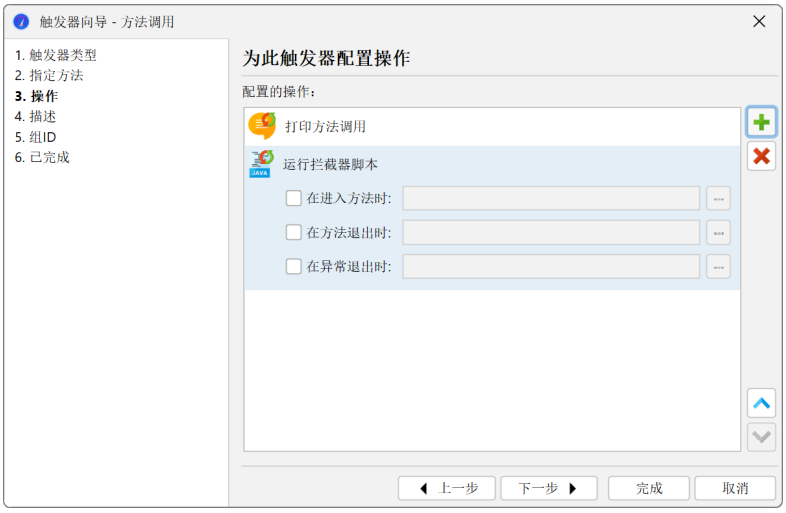
触发器

使用 触发器 [p. 27]，您可以在不修改源代码的情况下在 JProfiler GUI 中指定所有分析操作。触发器保存在 JProfiler 配置文件中。当您在启用离线分析的情况下启动时，配置文件和会话 ID 会通过命令行传递给分析代理，以便分析代理可以读取这些触发器定义。



与分析 API 不同，您需要在源代码中添加 API 调用，触发器是在 JVM 中发生某个事件时激活的。例如，您可以使用方法调用触发器，而不是在方法的开始或结束时为某个分析操作添加 API 调用。作为另一种用例，您可以使用计时器触发器，而不是创建自己的计时器线程来定期保存快照。

每个触发器都有一个动作列表，当关联的事件发生时执行这些动作。其中一些动作对应于控制器 API 中的分析动作。此外，还有其他超出控制器功能的动作，例如打印带参数和返回值的方法调用的动作或调用方法拦截器脚本的动作。



配置离线分析

如果您在 JProfiler 中配置了一个启动会话，可以通过从主菜单中调用 会话->转换向导->将应用程序会话转换为离线将其转换为离线会话。这将创建一个带有适当VM参数的启动脚本，并从您在JProfiler UI中使用的相同会话中获取分析设置。如果您想将调用移动到另一台计算机，您必须使用 会话->导出会话设置 将会话导出到配置文件，并确保启动脚本中的 VM 参数引用该文件。

当使用集成向导分析应用服务器时，总是会修改一个启动脚本或配置文件，以便将分析的 VM 参数插入到 Java 调用中。所有集成向导在“启动”步骤中都有一个“离线分析”选项，以便为离线分析而不是交互式分析配置应用服务器。

您可能希望自己将 VM 参数传递给 Java 调用，例如，如果您有一个不由集成向导处理的启动脚本。该 VM 参数的格式为

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

并且可以从 [Generic application] 向导中获得。

将offline作为库参数传递可以启用离线分析。在这种情况下，无法与JProfilerGUI连接。session参数决定了配置文件中哪个会话应被用于分析设置。会话的 ID 可以在会话设置对话框的 应用程序设置 选项卡的右上角看到。可选的 config 参数指向配置文件。您可以通过调用 会话->导出会话设置 导出该文件。如果省略该参数，将使用标准配置文件。该文件位于用户主目录中的 .jprofiler15 目录中。

使用 Gradle 和 Ant 进行离线分析

当您从 Gradle 或 Ant 启动离线分析时，可以使用相应的 JProfiler 插件使您的工作更轻松。下面显示了用于分析测试的 Gradle 任务的典型用法：

```
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` 任务以与标准 `JavaExec` 任务相同的方式分析任何 Java 类。如果您使用其他方法启动 JVM，而 JProfiler 不直接支持，`com.jprofiler.gradle.SetAgentPathProperty` 任务可以将所需的 VM 参数写入属性。应用 JProfiler 插件时默认添加，因此您可以简单地写：

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

然后在任务执行后将 `agentPathProperty` 用作项目属性引用。所有 Gradle 任务和相应的 Ant 任务的功能在 单独的章节 [p. 231] 中有详细记录。

为运行中的 JVM 启用离线分析

使用命令行工具 `bin/jpenable`，您可以在任何版本为 8 或更高的运行中的 JVM 中启动离线分析。就像 VM 参数一样，您必须指定 `offline` 开关、会话 ID 和可选的配置文件：

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

通过这样的调用，您必须从运行中的 JVM 列表中选择一个进程。通过附加参数 `--pid=<PID>` `--noinput`，您可以自动化该过程，使其完全不需要用户输入。

另一方面，当即时启用离线分析时，可能需要手动启动一些记录或保存快照。这可以通过 `bin/jpcontroller` 命令行工具实现。

如果仅加载了分析代理，但没有应用分析设置，则无法开启任何记录操作，因此 `jpcontroller` 将无法连接。这包括使用 `jpenable` 启用分析的情况，但没有 `offline` 参数。如果启用离线模式，则指定了分析设置，并且可以使用 `jpcontroller`。

有关 `jpenable` 和 `jpcontroller` 可执行文件的更多信息，请参阅 命令行参考 [p. 231]。

比较快照

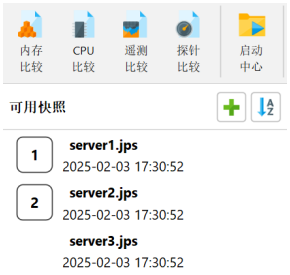
将当前应用程序的运行时特性与以前的版本进行比较是一种常见的质量保证技术，用于防止性能回归。它也可以帮助解决单个分析会话范围内的性能问题，在这种情况下，您可能希望比较两个不同的用例，找出为什么一个比另一个慢。在这两种情况下，您保存包含感兴趣数据的快照，并通过从菜单中调用会话->在新窗口中比较快照或点击启动中心的比较多个快照按钮来使用JProfiler中的快照比较功能。

选择快照

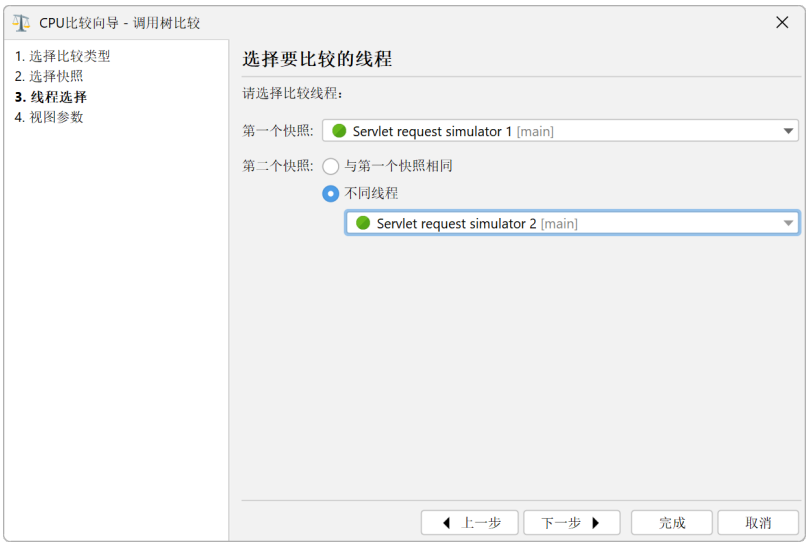
比较是在一个单独的顶级窗口中创建和查看的。首先，您在快照选择器中添加多个快照。然后，您可以通过选择感兴趣的快照并点击比较工具栏按钮，从列出的两个或多个快照中创建比较。列表中快照文件的顺序很重要，因为所有比较都假设列表中较后的快照是在较晚的时间记录的。除了手动排列快照，您还可以按名称或创建时间对它们进行排序。

与JProfiler主窗口中的视图不同，比较视图具有固定的视图参数，这些参数显示在顶部，而不是让您动态调整参数的下拉列表。所有比较都显示用于收集比较参数的向导，您可以使用相同的参数多次执行相同的比较。向导会记住其以前调用的参数，因此如果您比较多个快照集，则无需重复配置。在任何时候，您都可以通过完成按钮快捷跳过向导，或通过点击索引中的步骤跳转到另一个步骤。

当比较处于活动状态时，分析的快照会显示带有数字前缀。对于使用两个快照的比较，显示的差异是快照2的测量值减去快照1的测量值。

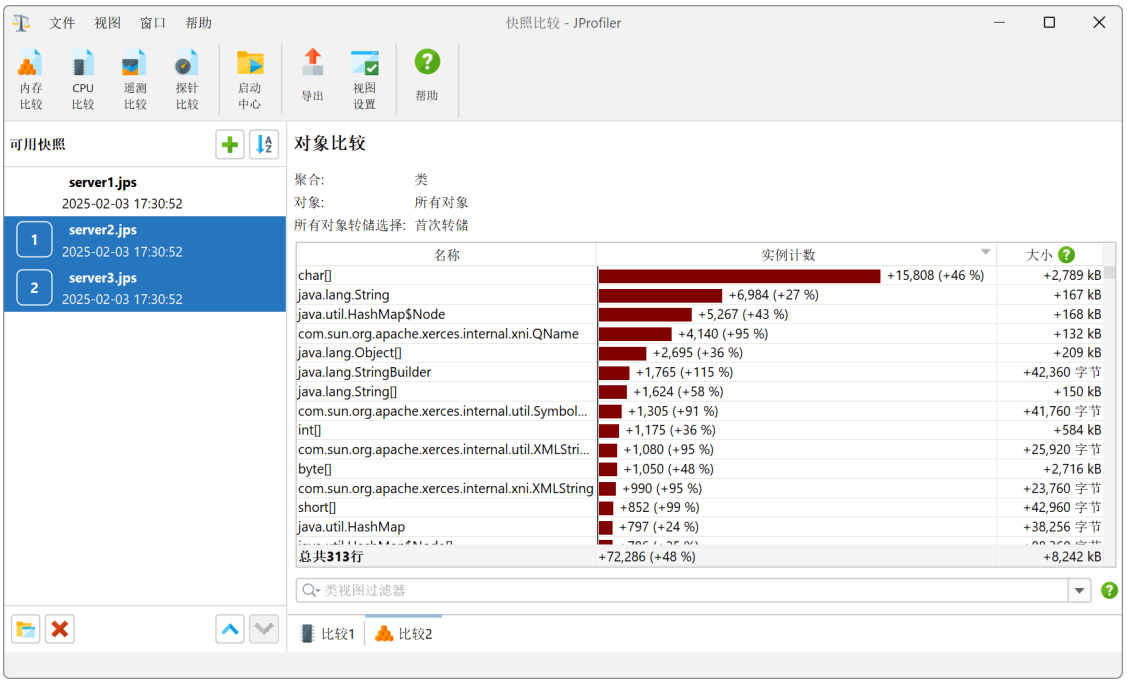


对于CPU比较，您可以使用相同的快照作为第一个和第二个快照，并在向导中选择不同的线程或线程组。

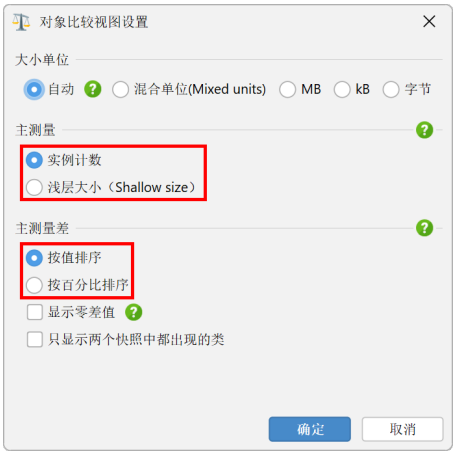


表格比较

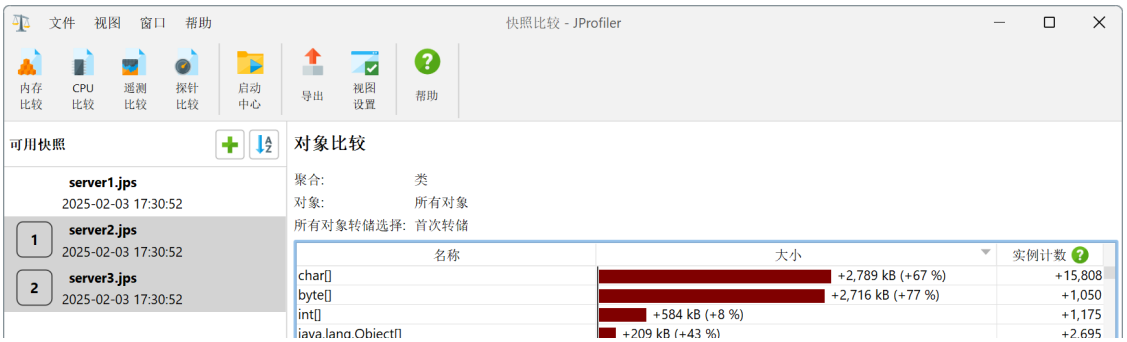
最简单的比较是“对象”内存比较。它可以比较堆行走器的“所有对象”、“记录的对象”或“类”视图中的数据。比较中的列显示实例计数和大小的差异，但只有实例计数列显示双向条形图，其中增加部分以红色向右绘制，而减少部分以绿色向左绘制。



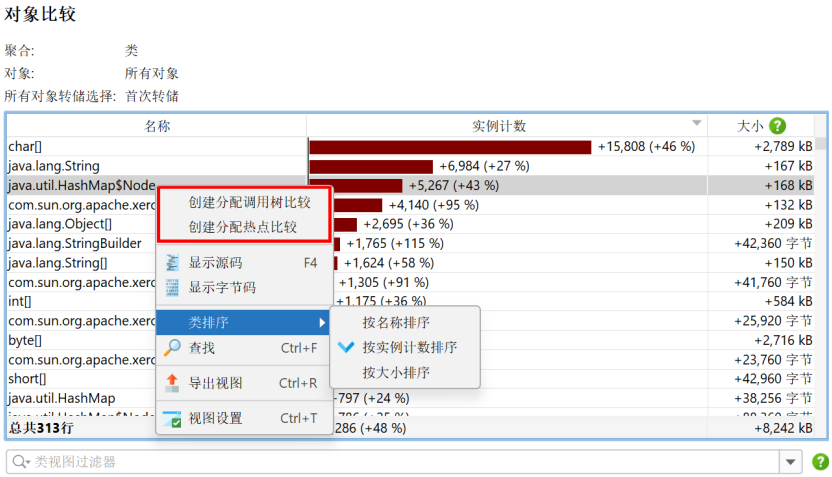
在视图设置对话框中，您可以选择是否希望此条形图显示绝对变化或百分比。另一个值显示在括号中。此设置还决定了列的排序方式。



第一个数据列中的测量称为主要测量，您可以在视图设置中将其从默认的实例计数切换为浅层大小。



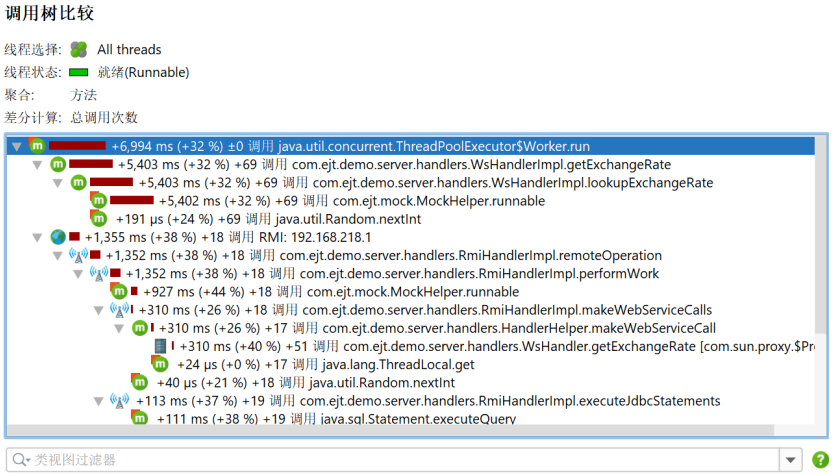
表格的上下文菜单为您提供了一个快捷方式，可以使用相同的比较参数和所选类进入其他内存比较。



像对象比较一样，CPU热点、探针热点和分配热点比较也显示在类似的表格中。

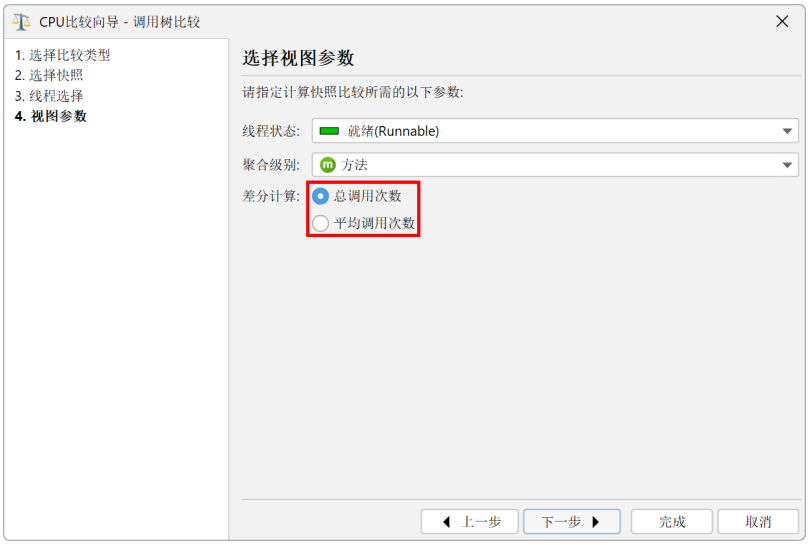
树状比较

对于每个CPU调用树、分配调用树和探针调用树，您可以计算另一个树，显示所选快照之间的差异。与常规调用树视图相比，内联条形图现在显示变化，增加部分以红色显示，减少部分以绿色显示。



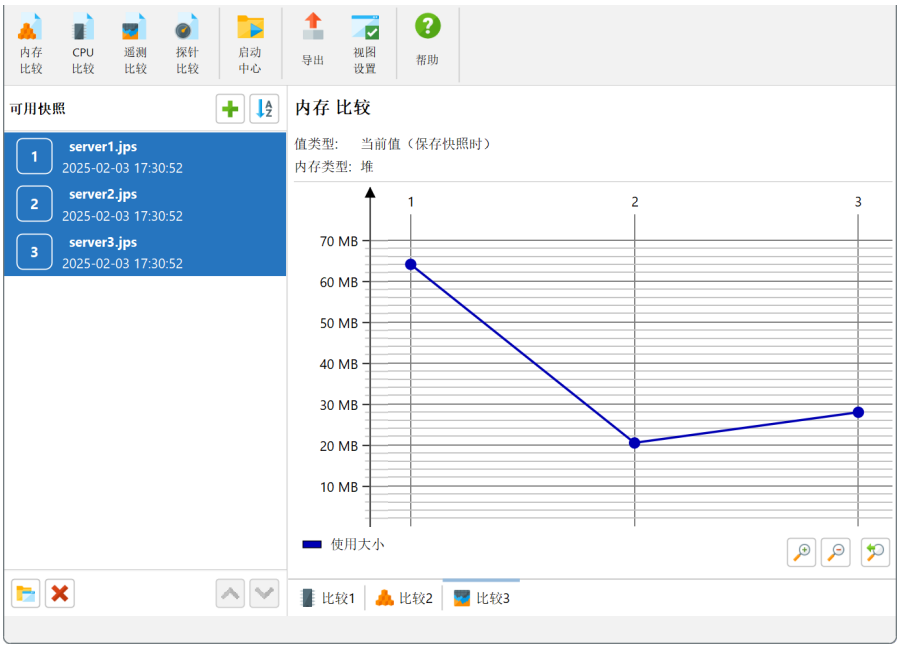
根据手头的任务，如果您只看到在两个快照文件中都存在并且从一个快照文件更改到另一个快照文件的调用栈，可能会让您更容易。您可以在视图设置对话框中更改此行为。

对于CPU和探针调用树比较，比较平均时间而不是总时间可能会很有趣。这是向导的“视图参数”步骤中的一个选项。



遥测比较

对于遥测比较，您可以同时比较多个快照。如果您在快照选择器中未选择任何快照，向导将假定您要比较所有快照。遥测比较没有时间轴，而是将编号的选定快照显示为序数x轴。工具提示包含快照的完整名称。



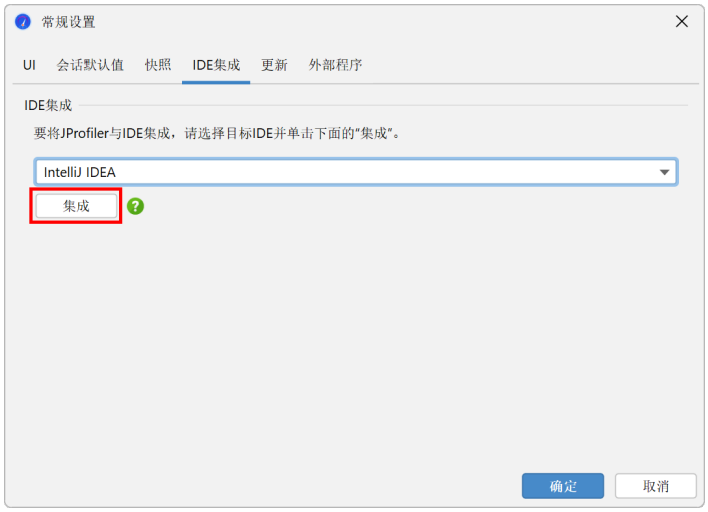
比较从每个快照中提取一个数字。由于遥测数据是时间解析的，因此有多种方法可以做到这一点。向导的“比较类型”步骤为您提供了使用快照保存时的值、计算最大值或在选定书签处找到值的选项。

IDE 集成

当您分析应用程序时，JProfiler的视图中出现的方法和类通常会引发只能通过查看其源代码来回答的问题。虽然JProfiler提供了一个内置的源代码查看器用于此目的，但其功能有限。此外，当发现问题时，通常的下一步是编辑有问题的代码。理想情况下，应该有一条从JProfiler的分析视图直接到IDE的路径，这样您就可以在不进行任何手动查找的情况下检查和改进代码。

安装 IDE 集成

JProfiler 为 IntelliJ IDEA、eclipse 和 NetBeans 提供 IDE 集成。要安装 IDE 插件，请从主菜单中调用会话->IDE 集成。IntelliJ IDEA 的插件安装通过 IDE 中的插件管理进行，其他 IDE 的插件则直接由 JProfiler 安装。安装程序还提供此操作，以便于与 JProfiler 安装一起更新 IDE 插件。集成向导将插件与 JProfiler 的当前安装目录连接。在 IDE 插件设置中，您可以随时更改使用的 JProfiler 版本。插件与 JProfiler GUI 之间的协议是向后兼容的，也可以与旧版本的 JProfiler 一起工作。

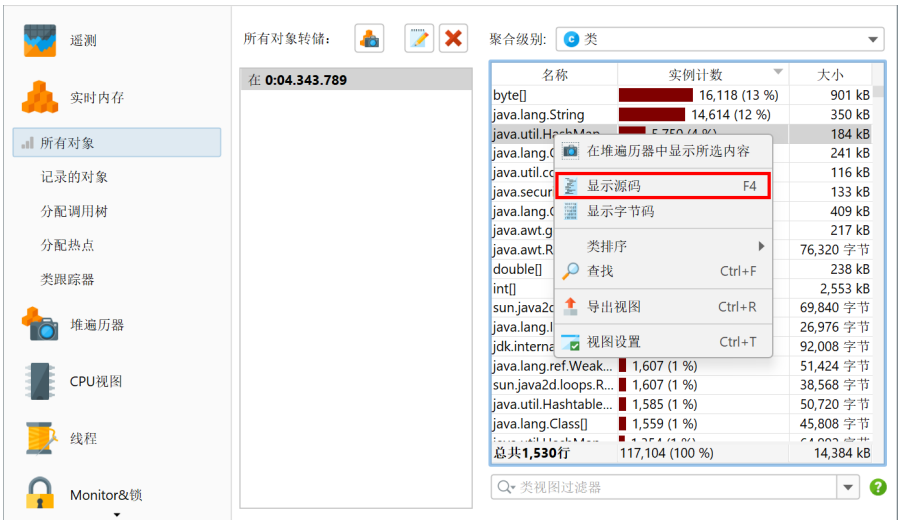


IntelliJ IDEA集成也可以从插件管理器安装。在这种情况下，插件将在您首次分析时询问JProfiler可执行文件的位置。

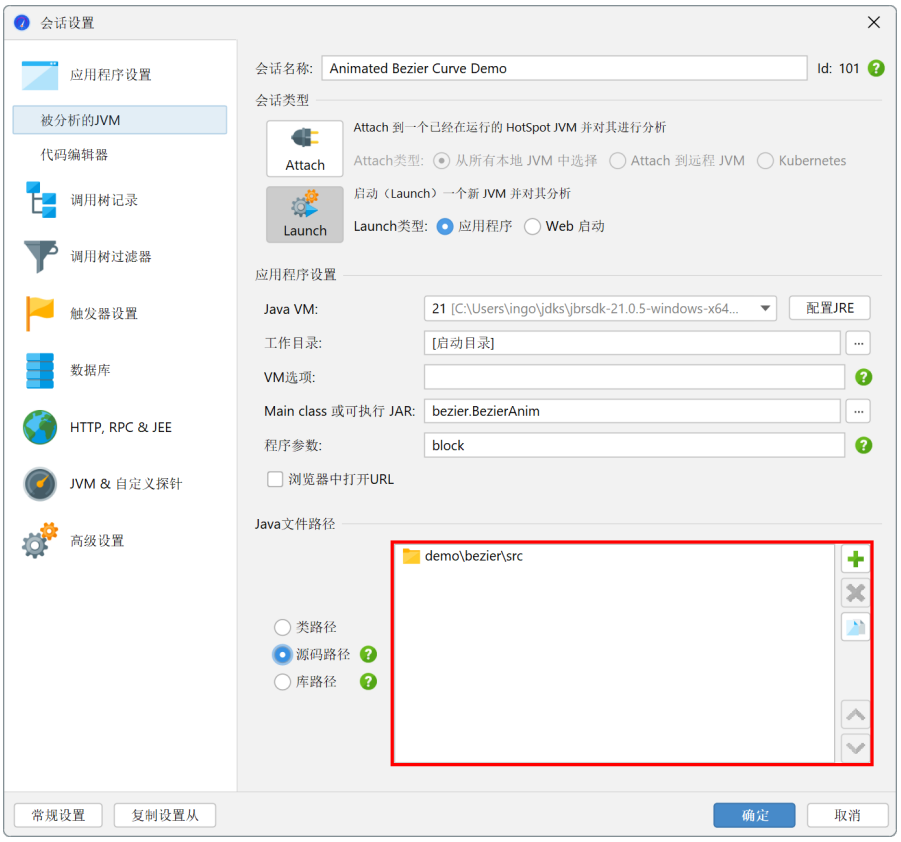
在不同的平台上，JProfiler 可执行文件位于不同的目录。在 Windows 上，它是`bin\jprofiler.exe`，在 Linux 或 Unix 上是`bin/jprofiler`，而在 macOS 上，JProfiler 应用程序包中有一个特殊的帮助 shell 脚本`Contents/Resources/app/bin/macos/jprofiler.sh`用于 IDE 集成。

源代码导航

在 JProfiler 中显示类名或方法名的任何地方，右键菜单中都有一个显示源代码操作。

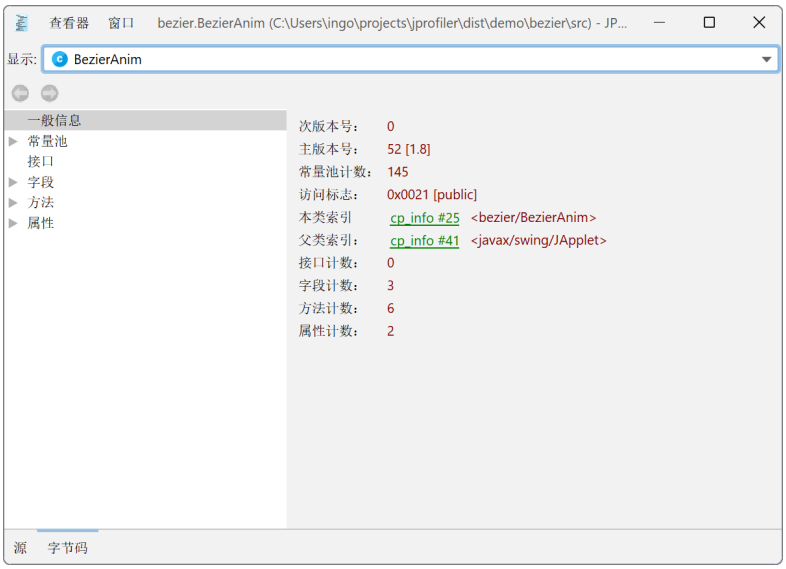


如果会话不是从 IDE 启动的，则会显示内置的源代码查看器，该查看器使用编译类文件中的行号表来查找方法。只有在应用程序设置中配置其根目录或包含的 ZIP 文件时，才能找到源文件。



与源代码显示一起，基于jclasslib 字节码查看器[1]的字节码查看器显示编译类文件的结构。

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



如果会话是从 IDE 启动的，则不会使用集成的源代码查看器，显示源代码操作将交给 IDE 插件。IDE 集成支持启动的分析会话、打开保存的快照以及附加到正在运行的 JVM。

对于实时分析会话，您可以像运行或调试一样启动IDE的被分析应用程序。然后，JProfiler插件将插入用于分析的 VM 参数，并连接一个 JProfiler 窗口。JProfiler 作为一个独立的进程运行，并在需要时由插件启动。JProfiler的源代码导航请求将发送到IDE中的关联项目。JProfiler和IDE插件合作，使窗口切换无缝进行，没有闪烁的任务栏条目，就像您在处理单个进程一样。

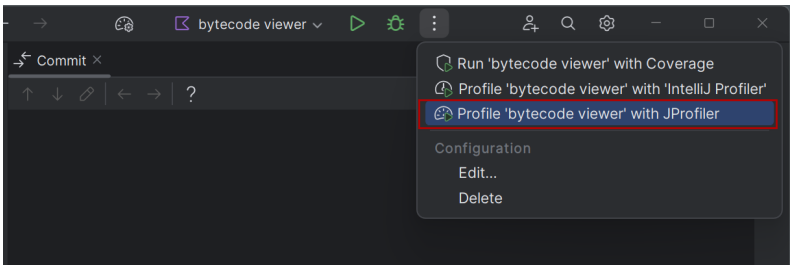
启动会话时，“会话启动”对话框允许您配置所有分析设置。用于启动会话的配置文件设置由JProfiler记住，具体取决于 IDE 集成，是按项目还是按运行配置记住的。当会话首次被分析时，IDE 插件会自动根据源文件包层次结构中的最顶层类确定被分析包的列表。在任何时候，您都可以转到会话设置对话框中的过滤器设置步骤，并使用重置按钮再次执行此计算。

对于快照，IDE 集成通过在 IDE 中使用文件->打开操作打开快照文件或在项目窗口中双击它来设置。来自 JProfiler 的源代码导航将被定向到当前项目。最后，IDE 插件在 IDE 中添加一个附加到 JVM操作，允许您选择一个正在运行的 JVM，并将源代码导航到 IDE，类似于快照的机制。

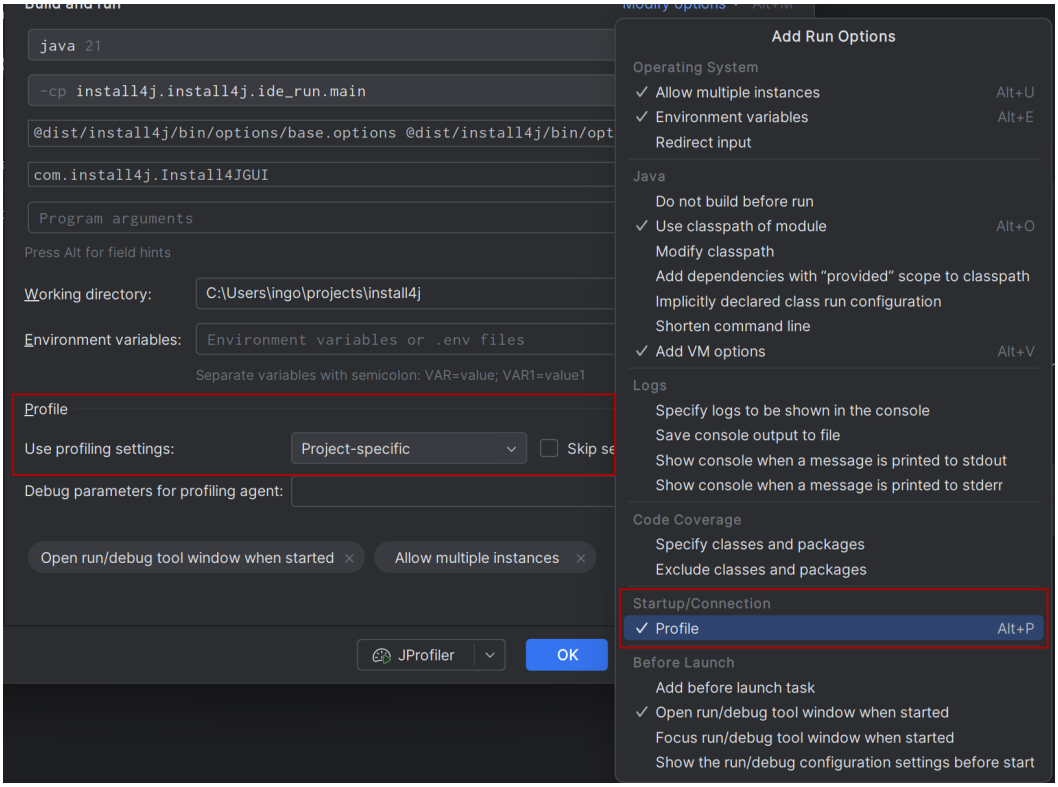
有时您可能希望切换到 IDE 而不考虑特定的类或方法。为此，JProfiler 窗口中的工具栏有一个激活IDE按钮，该按钮显示用于由IDE集成打开的分析会话。该操作绑定到F11键，就像IDE中的JProfiler激活操作一样，因此您可以使用相同的键绑定在 IDE 和 JProfiler 之间来回切换。

IntelliJ IDEA 集成

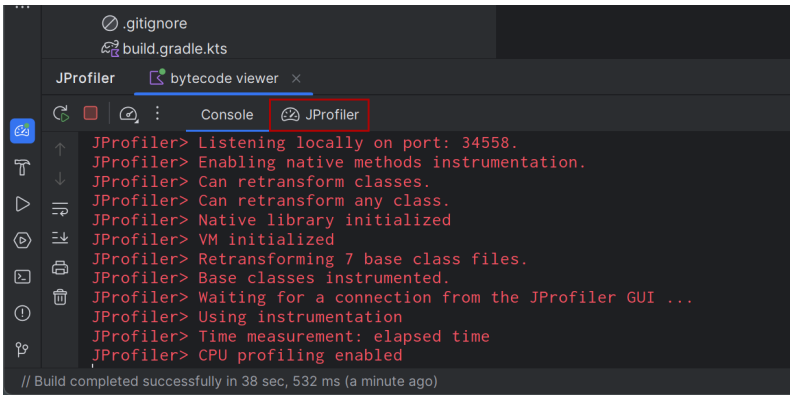
要从 IntelliJ IDEA 分析您的应用程序，请在运行菜单中选择一个分析命令，或单击主工具栏中运行或调试操作旁边的下拉菜单以选择“使用 JProfiler 分析”操作。JProfiler 可以分析大多数运行 IDEA 配置类型，包括应用服务器。



JProfiler插件向运行配置添加了额外的设置，这些设置并不立即可见。要访问这些设置，请在“修改选项”下拉菜单中选择“分析”选项。所有其他分析设置可以在JProfiler窗口的启动对话框中配置。



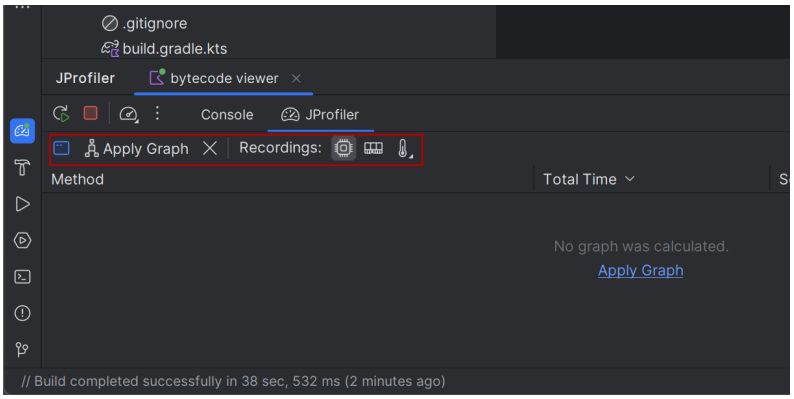
一旦分析会话启动，输出将显示在一个单独的JProfiler工具窗口中。该工具窗口显示控制台输出，就像常规运行工具窗口一样，还有一个“JProfiler”选项卡，可以在您连接到 JProfiler UI 后使用：



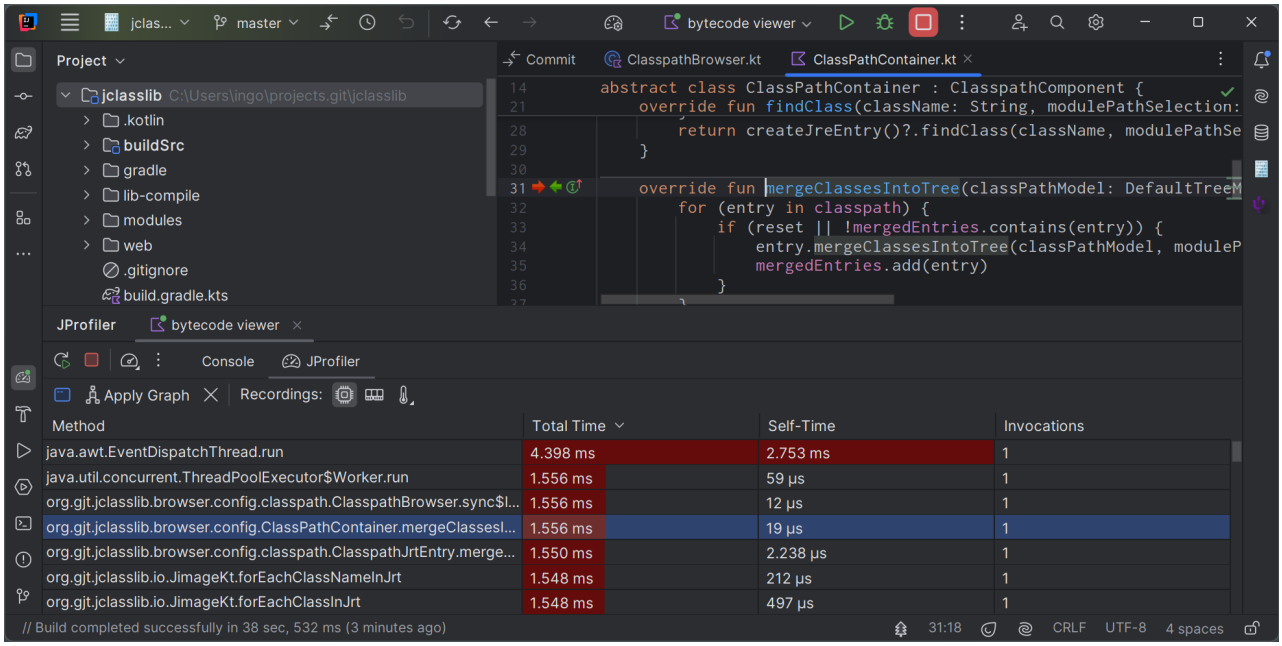
当您在 IntelliJ IDEA 中打开 JProfiler 快照或使用“使用 JProfiler 附加到 JVM”操作附加到正在运行的 JVM 时，也会显示 JProfiler 工具窗口。

“JProfiler”选项卡包含用于启动和停止 CPU 数据、分配数据和探针事件的数据记录的操作。此外，还有一个操作允许您切换到 JProfiler 窗口。JProfiler 窗口包括一个类似的操作，用于切换回 IDEA 窗口，使得使用两个独立的窗口变得方便。JProfiler 到 IntelliJ IDEA 的精确源代码导航针对 Java 和 Kotlin 实现。

分析信息通常显示在 JProfiler 窗口中，但 CPU 图形数据也集成在 IntelliJ IDEA UI 中，因为直接在源代码中显示这些数据是有意义的。在 IntelliJ IDEA 中使用“应用图形”操作或在 JProfiler 中生成 CPU 图形以在 IntelliJ IDEA 中显示 CPU 数据。要配置高级参数，如线程选择或使用调用树根、调用树移除和调用树视图过滤器设置，您应该在 JProfiler 窗口中生成图形。



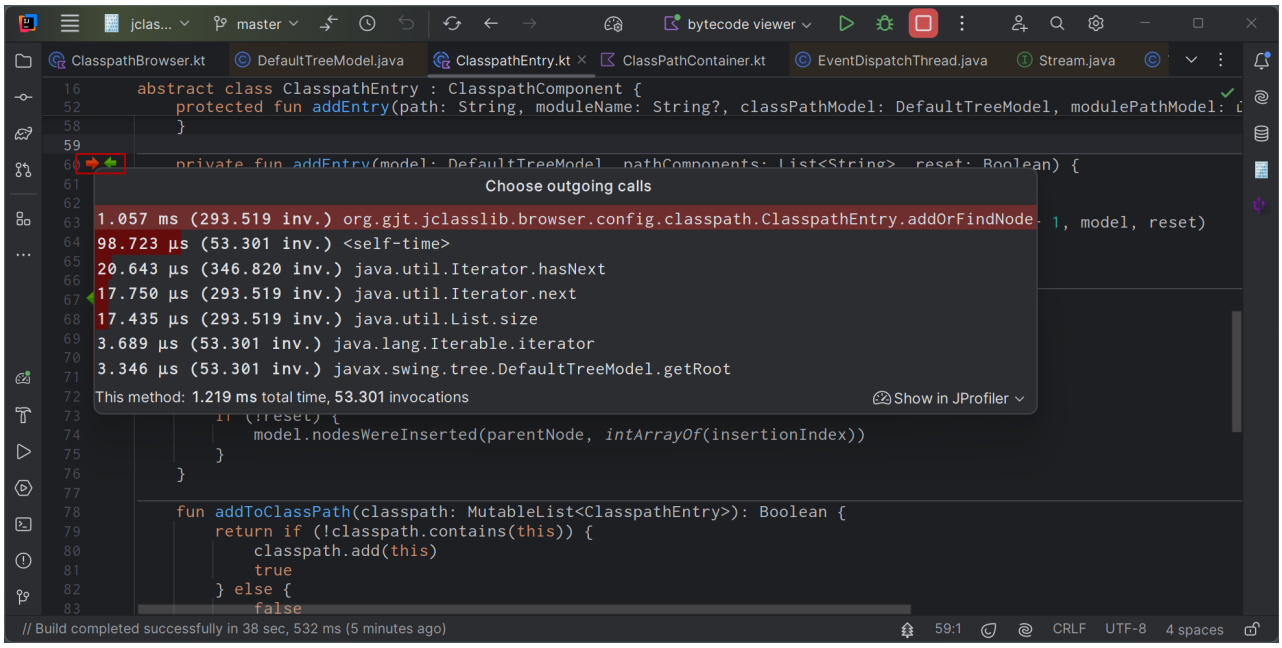
一旦应用了 CPU 数据，“JProfiler”选项卡将显示记录的方法列表。双击方法将带您到源代码。在源代码编辑器的边缘，添加了用于传入和传出调用的箭头。



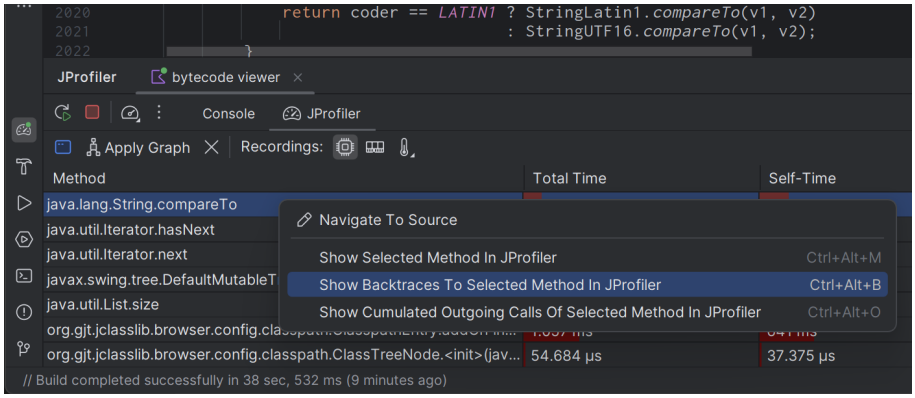
单击边缘图标会在弹出窗口中显示传入或传出方法，以及显示记录时间的条形图。单击弹出窗口中的行将导航到相应的方法。

此外，目标方法的总记录时间和调用次数将显示在弹出窗口的底部。弹出窗口右下角的“在 JProfiler 中显示”下拉菜单提供上下文相关的导航操作到 JProfiler UI。您可以在方法图中显示选定的节点或

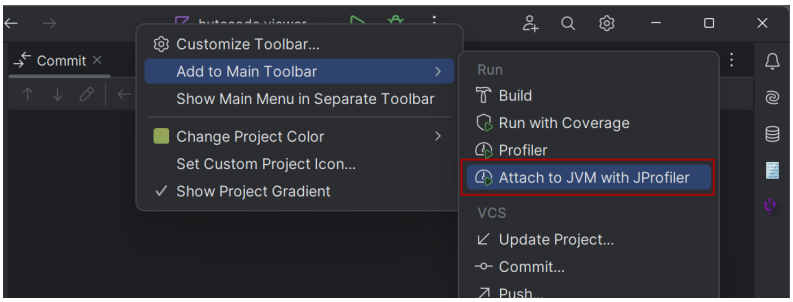
相应的调用树分析。对于传出调用，提供“累积传出调用”分析，对于传入调用，提供“回溯”分析。



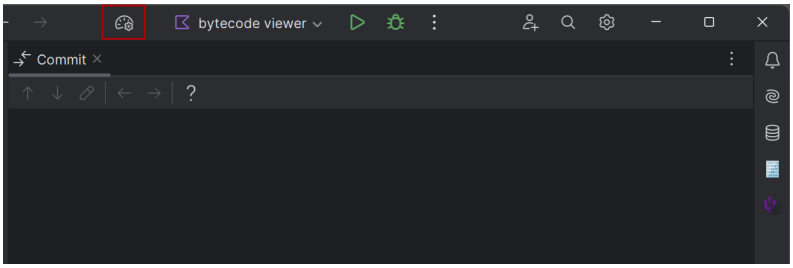
同样的导航操作也可以在“JProfiler”选项卡中的方法表的上下文菜单中使用：



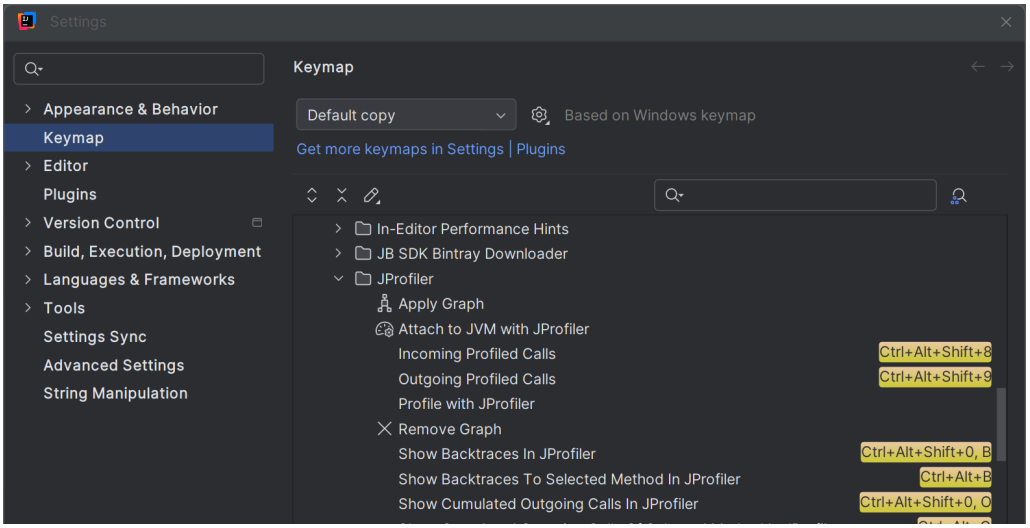
JProfiler 插件提供了一个工具栏快速操作，用于“使用 JProfiler 附加到 JVM”操作，您可以将其添加到主工具栏。通过该操作，您可以附加到已经运行的进程，并仍然从 JProfiler UI 到 IntelliJ IDEA 获取源代码导航，以及在源代码编辑器中内联 CPU 图形数据：



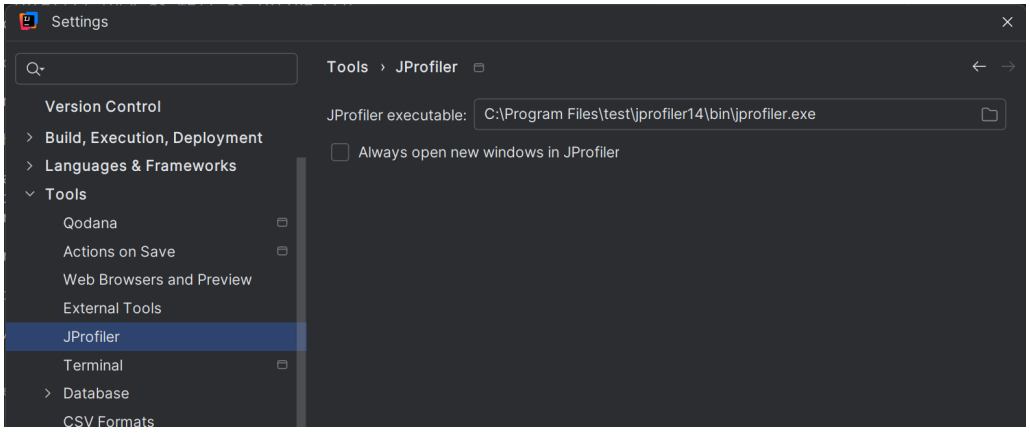
这是添加后操作按钮的外观：



JProfiler中所有操作的键绑定可以在IntelliJ IDEA的“键映射”设置中自定义。鉴于非冲突键盘快捷键的有限可用性，从源代码编辑器到 JProfiler UI 的导航操作是链式快捷键，您首先按 `Ctrl-Alt-Shift-O`，然后按另一个键选择导航操作。如果您经常使用此功能，您可能希望分配更简单的键盘快捷键。



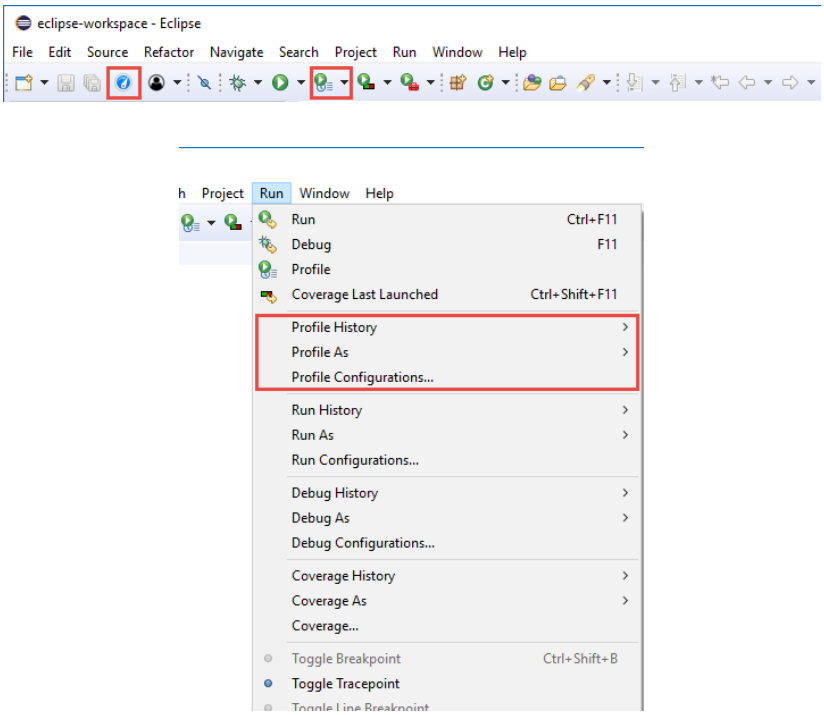
在 IDE 设置的工具->JProfiler页面上，您可以调整使用的 JProfiler 可执行文件以及是否总是希望在 JProfiler 中为新的分析会话打开一个新窗口。



Eclipse 集成

eclipse 插件可以分析大多数常见的启动配置类型，包括测试运行配置和 WTP 运行配置。eclipse 插件仅适用于完整的 eclipse SDK，而不适用于 eclipse 框架的部分安装。

要从 eclipse 分析您的应用程序，请在运行菜单中选择一个分析命令或单击相应的工具栏按钮。分析命令等同于 eclipse 中的调试和运行命令，是 eclipse 基础设施的一部分，除了运行->附加 JProfiler 到 JVM菜单项，这是由 JProfiler 插件添加的。

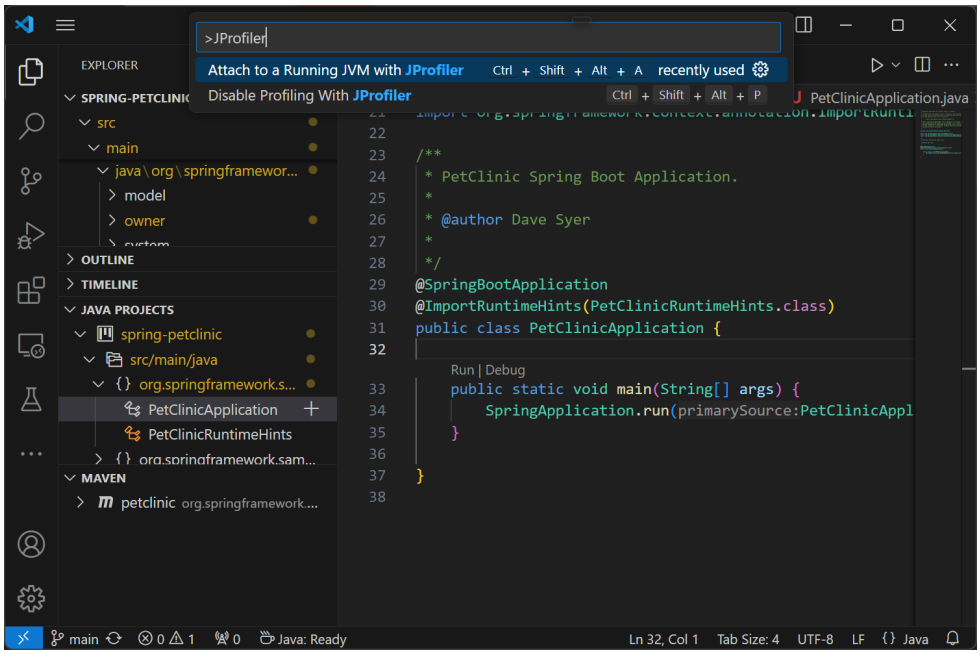


如果 Java 透视图中不存在菜单项运行->分析...，请在窗口->透视图->自定义透视图下启用此透视图的“分析”操作，通过将操作集可用性选项卡置于前面并选择分析复选框。

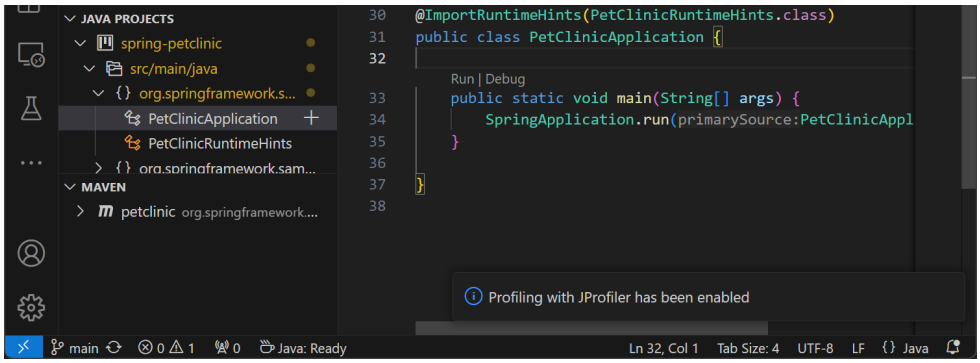
包括 JProfiler 可执行文件位置在内的多个 JProfiler 相关设置可以在 eclipse 的窗口->首选项->JProfiler 下进行调整。

VS Code 集成

VS Code 扩展添加了一个操作 `JProfiler` 。调用时，调试和运行操作将启动 Java 启动配置的分析。JProfiler 将启动，并显示会话启动对话框，您可以在其中配置分析设置。确认会话启动对话框后，应用程序将启动。



使用 `JProfiler` 操作，恢复运行和调试操作的默认行为。有关分析模式更改的通知将作为编辑器右下角的 toast 消息显示在 VS Code 中。 `JProfiler` 和 `JProfiler` 具有相同的默认键绑定，因此可以用于切换分析模式。



要分析已运行的 JVM，请使用 `JVM` `JProfiler` 操作。

JProfiler 中的源代码导航操作将在 VS Code 中显示相应的源代码。要在 VS Code 中获取 JProfiler 快照的源代码导航，请在 VS Code 中使用文件->打开打开快照。

NetBeans 集成

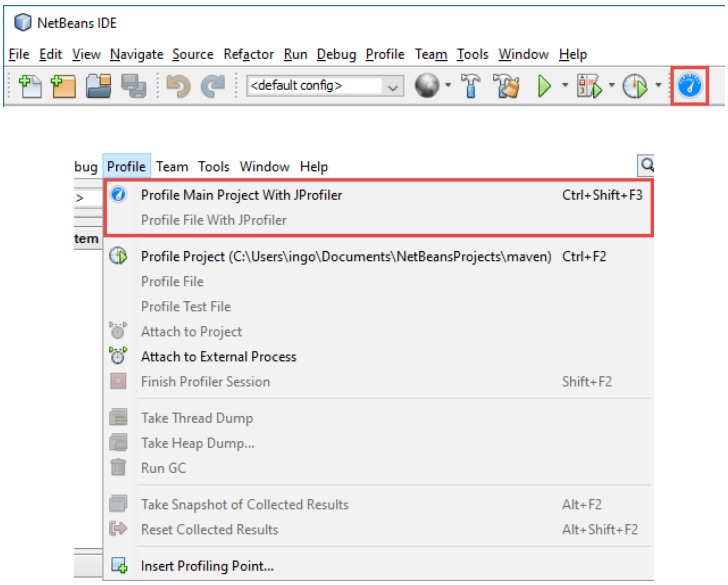
在 NetBeans 中，您可以分析使用 exec Maven 插件的标准、自由格式和 Maven 项目。要从 NetBeans 分析您的应用程序，请在运行菜单中选择一个分析命令或单击相应的工具栏按钮。对于以其他方式启动应用程序的 Maven 项目和 Gradle 项目，请正常启动项目并使用菜单中的分析->附加 JProfiler 到正在运行的 JVM操作。

对于自由格式项目，您必须先调试一次应用程序，然后再尝试分析它，因为所需的文件`nbproject/ide-targets.xml`是由调试操作设置的。JProfiler 将向其添加一个名为“profile-jprofiler”的目标，其内容与调试目标相同，并将尝试根据需要修改 VM 参数。如果您在分析自由格式项目时遇到问题，请检查此目标的实现。

您可以使用集成的 Tomcat 或在 NetBeans 中配置的任何其他 Tomcat 服务器分析 Web 应用程序。当您的主项目是 Web 项目时，选择使用 JProfiler 分析主项目将启动启用分析的 Tomcat 服务器。

如果您使用带有捆绑 GlassFish 服务器的 NetBeans，并且您的主项目设置为使用 GlassFish 服务器，选择使用 JProfiler 分析主项目将启动启用分析的应用服务器。

JProfiler 可执行文件的位置和打开新 JProfiler 窗口的策略可以在选项对话框中的杂项->JProfiler下进行调整。



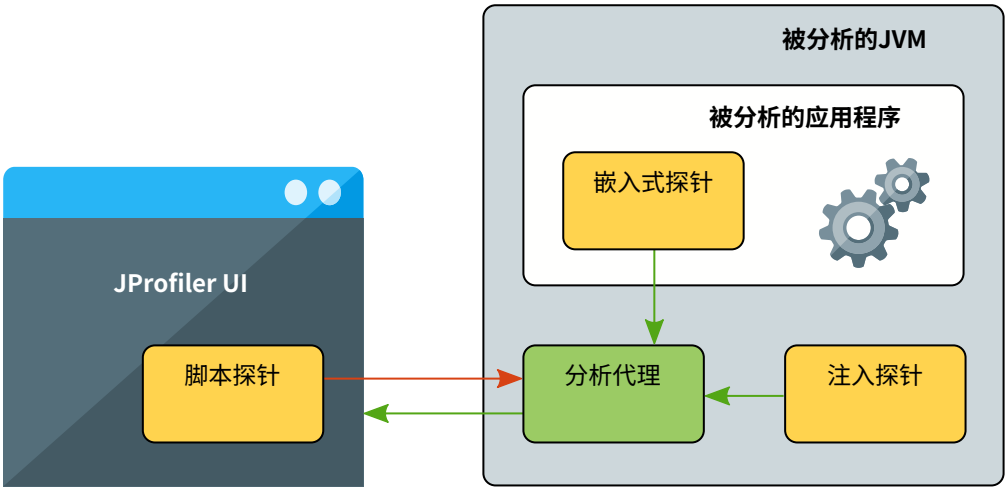
A 自定义探针

A.1 探针概念 (Probe Concepts)

为 JProfiler 开发自定义探针时，您需要了解一些基本概念和术语。JProfiler 所有探针的共同基础是它们拦截特定方法，并使用拦截的方法参数和其他数据源构建一个包含您希望在 JProfiler UI 中看到的有趣信息的字符串。

定义探针时的初始问题是如何指定拦截的方法，并获得一个可以使用方法参数和其他相关对象来构建字符串的环境。在 JProfiler 中，有三种不同的方法可以做到这一点：

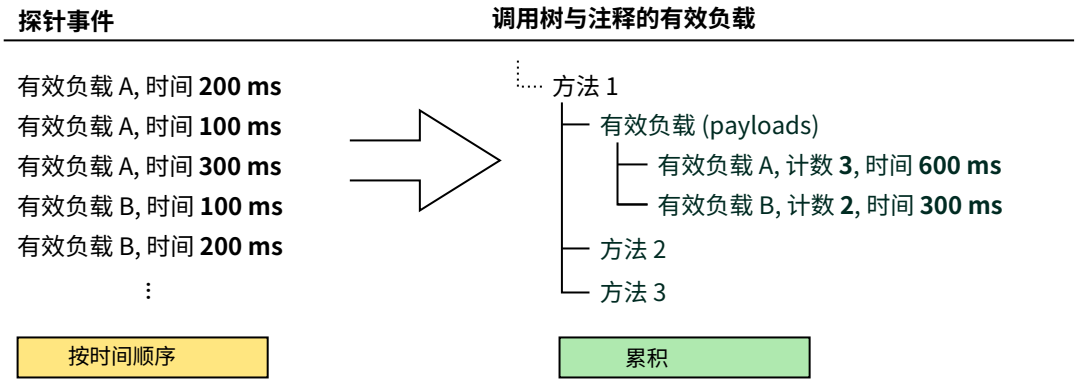
- **脚本探针 (script probe)** [p. 145] 完全在 JProfiler UI 中定义。您可以在调用树中右键单击一个方法，选择脚本探针操作，并在内置代码编辑器中输入字符串表达式。这对于实验探针非常有用，但仅暴露了自定义探针功能的一个非常有限的部分。
- **嵌入探针 (embedded probe)** [p. 154] API 可以从您自己的代码中调用。如果您编写一个库、数据库驱动程序或服务器，您可以随产品一起提供探针。任何使用 JProfiler 分析您的产品的人，都会自动将您的探针添加到 JProfiler UI 中。
- 使用 **注入探针 (injected probe)** [p. 149] API，您可以在 IDE 中为第三方软件编写探针，充分利用 JProfiler 的探针系统。该 API 使用注解来定义拦截并注入方法参数和其他有用对象。



下一个问题是：JProfiler应该如何处理您创建的字符串？有两种不同的策略可用：有效负载创建或调用树拆分。

有效负载创建 (Payload creation)

探针构建的字符串可用于创建一个 **探针事件 (probe event)**。该事件具有设置为该字符串的描述、等于拦截方法调用时间的持续时间以及相关的调用栈。在其相应的调用栈中，探针描述和时间被累积并作为 **有效负载 (payloads)** 保存到调用树中。当事件在达到某个最大数量后被合并时，调用树中累积的有效负载显示整个记录期间的总数。如果同时记录了 CPU 数据和您的探针，探针调用树视图将显示合并的调用栈，其中有效负载字符串作为叶节点，CPU 调用树视图将包含指向探针调用树视图的 **注释链接 (annotated links)**。



就像 CPU 数据一样，有效负载可以在调用树或热点视图中显示。热点显示哪些有效负载负责大部分消耗的时间，反向跟踪显示哪些代码部分负责创建这些有效负载。为了获得良好的热点列表，有效负载字符串不应包含任何唯一ID或时间戳，因为如果每个有效负载字符串都不同，就不会有累积，也没有明确的热点分布。例如，在准备好的 JDBC 语句的情况下，参数不应包含在有效负载字符串中。

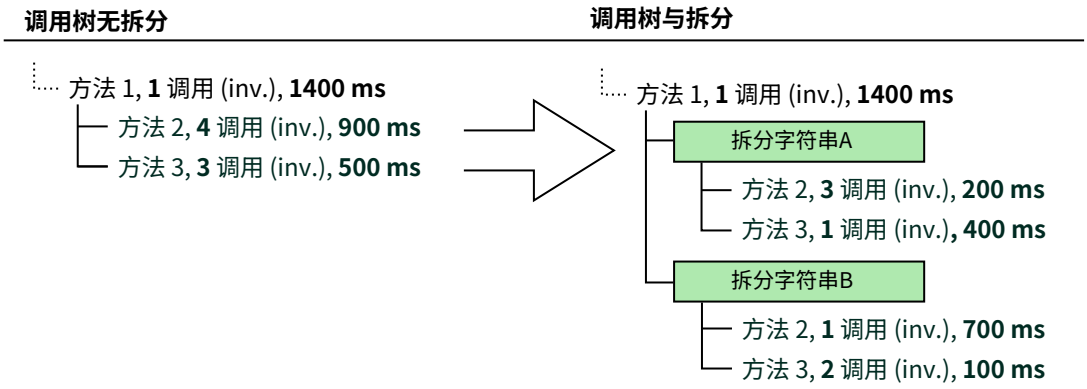
脚本探针会自动从配置的脚本返回值创建有效负载。注入探针类似，它们从带有`PayloadInterception`注解的拦截处理方法返回有效负载描述，作为字符串或`Payload`对象以实现高级功能。另一方面，嵌入探针通过调用 `Payload.exit` 并将有效负载描述作为参数来创建有效负载，其中 `Payload.enter` 和 `Payload.exit` 之间的时间被记录为探针事件持续时间。

如果您正在记录在不同调用点发生的服务调用，有效负载创建最为有用。一个典型的例子是数据库驱动程序，其中有效负载字符串是某种形式的查询字符串或命令。探针从调用点的角度出发，测量的工作由另一个软件组件执行。

调用树拆分 (Call tree splitting)

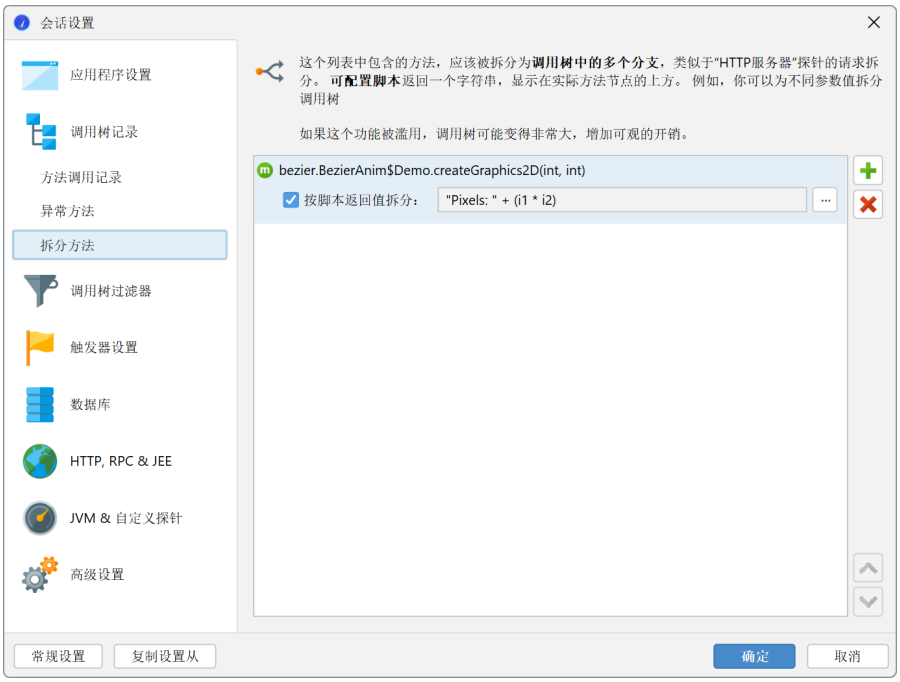
探针也可以从执行站点的角度出发。在这种情况下，重要的不是拦截方法如何被调用，而是拦截方法之后执行了哪些方法调用。一个典型的例子是用于 servlet 容器的探针，其中提取的字符串是一个URL。

比创建有效负载更重要的是现在能够为探针构建的每个不同字符串拆分调用树。对于每个这样的字符串，将在调用树中插入一个拆分节点，其中包含所有相应调用的累积调用树。否则只有一个累积调用树，现在有一组拆分节点将调用树分割成不同的部分，可以分别进行分析。



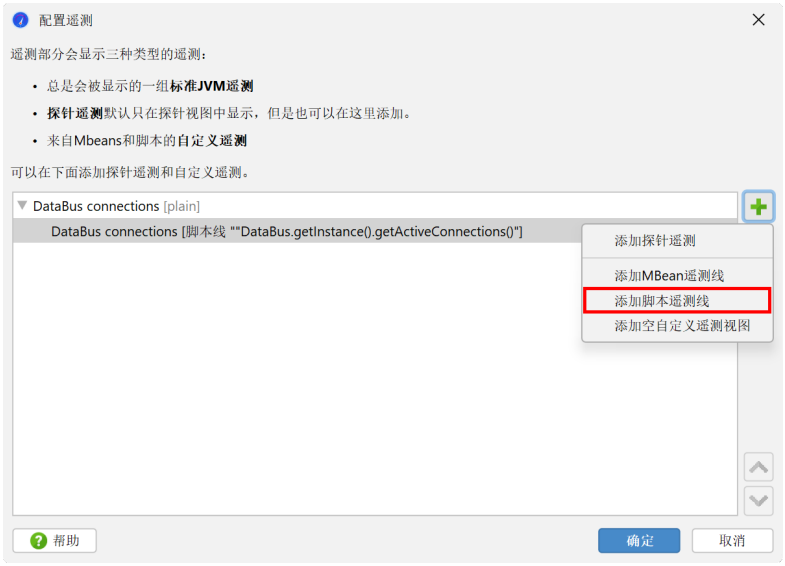
多个探针可以产生嵌套拆分。默认情况下，单个探针仅产生一个拆分级别，除非它被配置为 **可重入 (reentrant)**，这不支持脚本探针。

在 JProfiler UI 中，调用树拆分不与脚本探针功能捆绑在一起，而是一个 单独的功能 (separate feature) [p. 171]，称为 "Split methods"。它们仅拆分调用树而不创建有效负载，因此不需要带有名称和描述的探针视图。注入探针从带有`SplitInterception`注解的拦截处理方法返回拆分字符串，而嵌入探针调用 `Split.enter` 并传入拆分字符串。



遥测 (Telemetries)

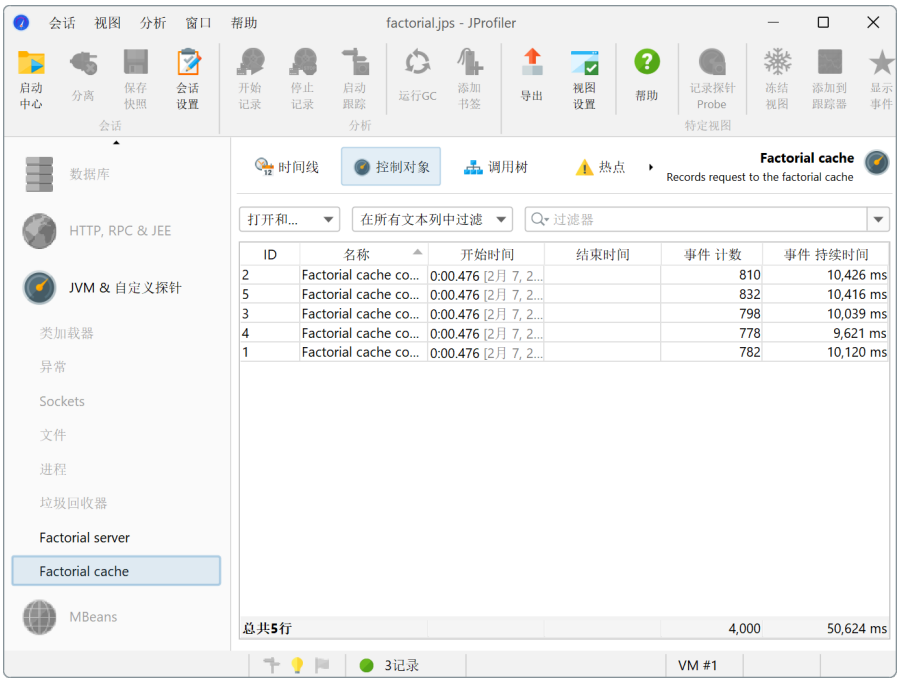
自定义探针有两个默认遥测：事件频率和平均事件持续时间。注入和嵌入探针支持通过探针配置类中的注解方法创建的附加遥测。在 JProfiler UI 中，脚本遥测独立于脚本探针功能，并在 "Telemetries" 部分中找到，位于工具栏中的 Configure Telemetries 按钮下。



遥测方法每秒轮询一次。在 `Telemetry` 注解中，您可以配置单位和比例因子。使用 `line` 属性，可以将多个遥测组合成一个遥测视图。使用 `TelemetryFormat` 的 `stacked` 属性，您可以使线条累加并将其显示为堆叠线图。嵌入和注入探针中的遥测相关 API 是等效的，但仅适用于各自的探针类型。

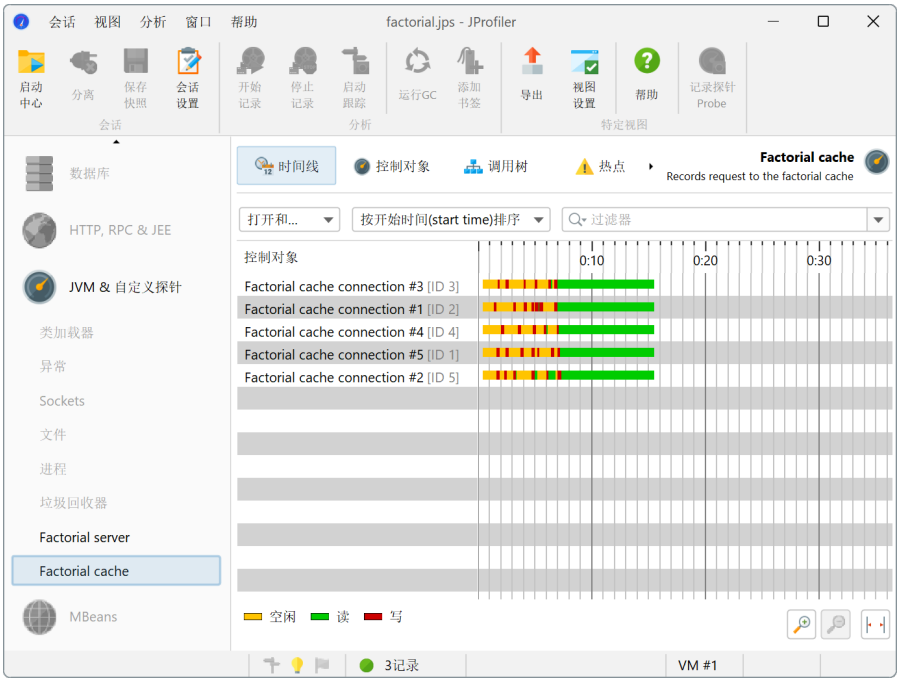
控制对象 (Control objects)

有时将探针事件与相关的长生命周期对象关联是很有趣的，这些对象在 JProfiler 中称为 "控制对象 (control objects)"。例如，在数据库探针中，该角色由执行查询的物理连接承担。这些控制对象可以通过嵌入 API 和注入探针 API 打开和关闭，从而在探针事件视图中生成相应的事件。当创建探针事件时，可以指定控制对象，以便探针事件贡献于探针的 "控制对象" 视图中显示的统计信息。



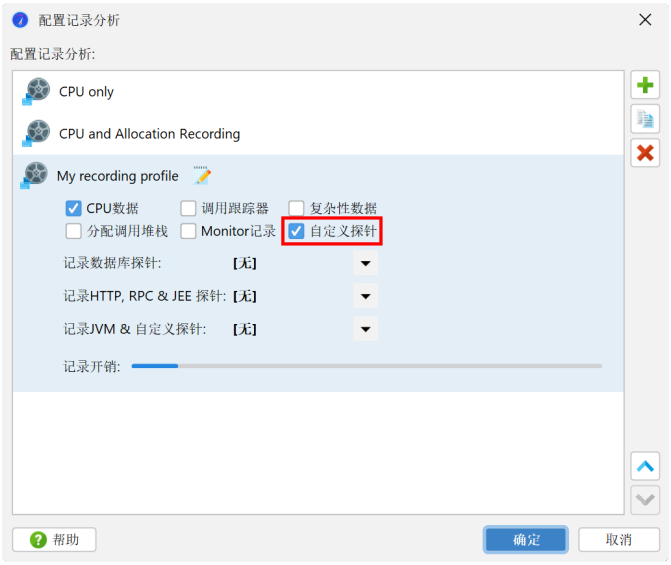
控制对象具有在打开时必须指定的显示名称。如果在创建探针事件时使用了新的控制对象，探针必须在其配置中提供名称解析器。

此外，探针可以通过枚举类定义自定义事件类型。当创建探针事件时，可以指定这些类型之一，并在事件视图中显示，您可以在其中筛选单个事件类型。更重要的是，显示控制对象作为时间轴上线条的探针时间线视图根据事件类型进行着色。对于没有自定义事件类型的探针，着色显示空闲状态，其中没有记录事件，以及探针事件持续时间的默认事件状态。使用自定义类型，您可以区分状态，例如"读取 (read)" 和 "写入 (write)"。

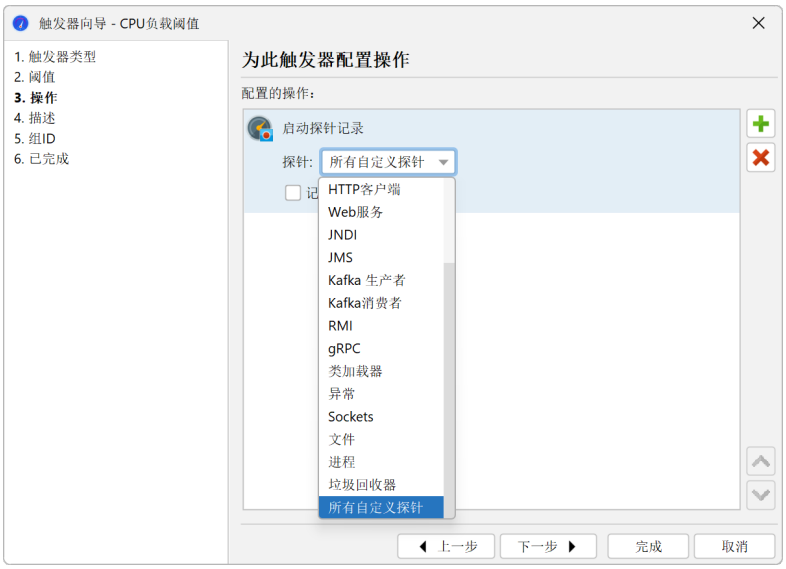


记录 (Recording)

像所有探针一样，自定义探针默认不记录数据，但您必须根据需要启用和禁用记录。虽然您可以在探针视图中使用手动开始/停止操作，但通常需要在开始时打开探针记录。因为 JProfiler 事先不知道自定义探针，记录配置文件中有一个 Custom probes 复选框，适用于所有自定义探针。



同样，您可以为开始和停止探针记录的触发器操作选择 All custom probes。



对于编程记录，您可以调用 `Controller.startProbeRecording(Controller.PROBE_NAME_ALL_CUSTOM, ProbeRecordingOptions.EVENTS)` 来记录所有自定义探针，或者传递探针的类名以更具体。

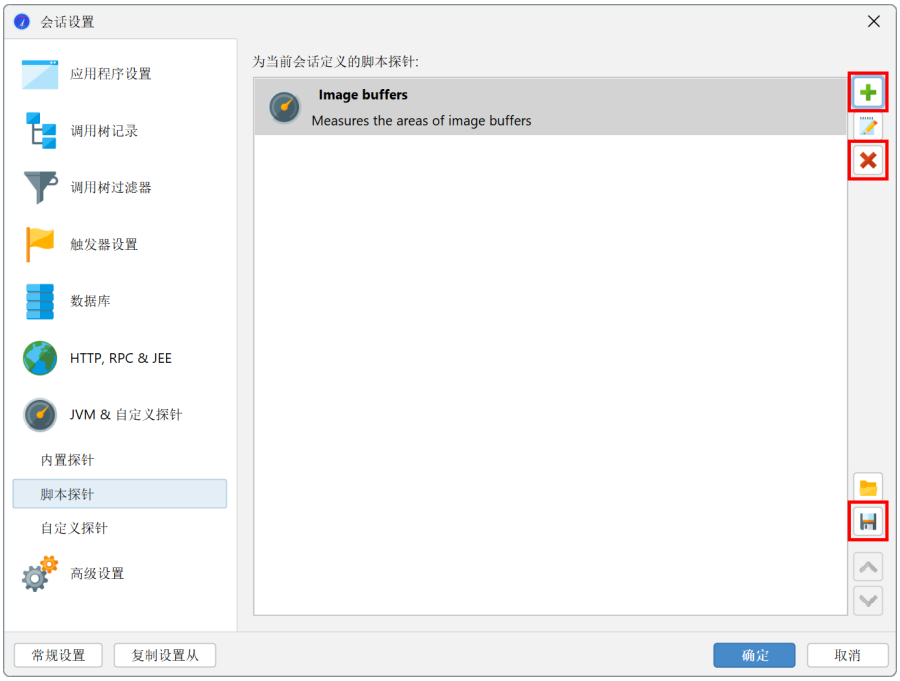
A.2 脚本探针

在您的IDE中开发自定义探针需要清晰理解拦截点及探针将提供的好处。而使用脚本探针，您可以直接在JProfiler GUI中快速定义简单探针并进行实验，而无需学习任何API。与嵌入式或注入式自定义探针不同，脚本探针可以在运行中的分析会话期间重新定义，从而实现快速的编辑-编译-测试循环。

定义脚本探针

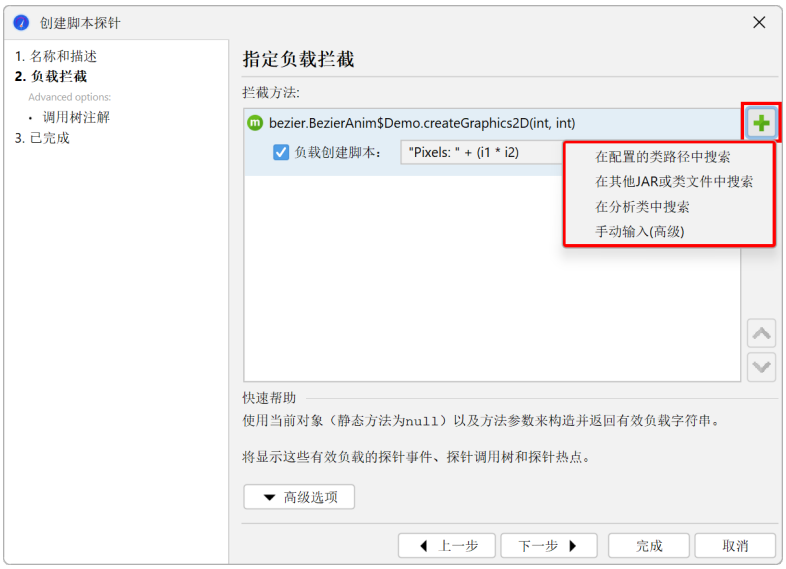
通过选择一个被拦截的方法并输入返回探针有效负载字符串的脚本来定义脚本探针。多个这样的方法-脚本对可以捆绑在一个探针中。

可以在会话设置中访问脚本探针配置。这是创建和删除脚本探针以及将您的脚本探针保存到可以被其他分析会话导入的集合中的地方。

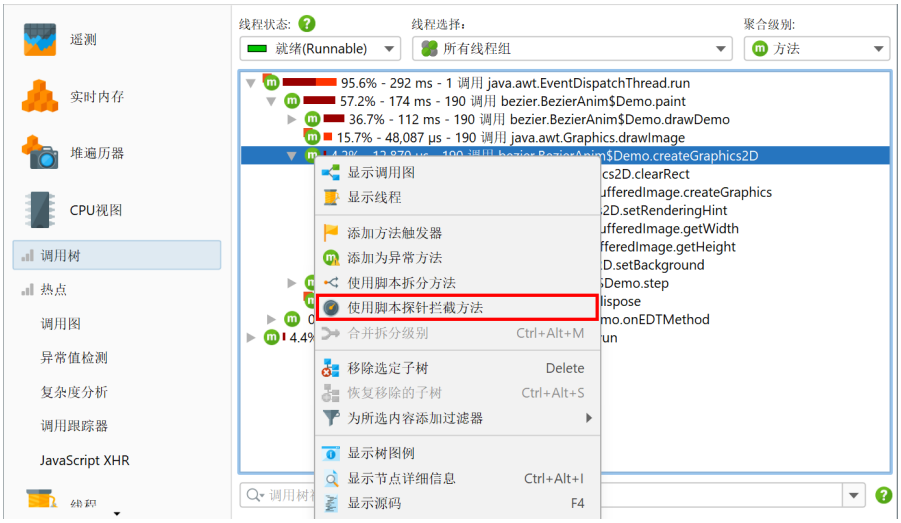


每个脚本探针需要一个名称和可选的描述。名称用于在JProfiler的视图选择器中的"JEE & Probes"部分添加探针视图。描述显示在探针视图的标题中，应为其目的提供简短的解释。

对于选择方法，您有多种选项，包括从配置的类路径中选择一个类或从被分析的类中选择一个类（如果分析会话已经在运行）。在第二步中，您可以从选定的类中选择一个方法。



从调用树视图中选择被拦截的方法要容易得多。在上下文菜单中，使用脚本探针拦截方法操作将询问您是否要创建新探针或将拦截添加到现有探针。

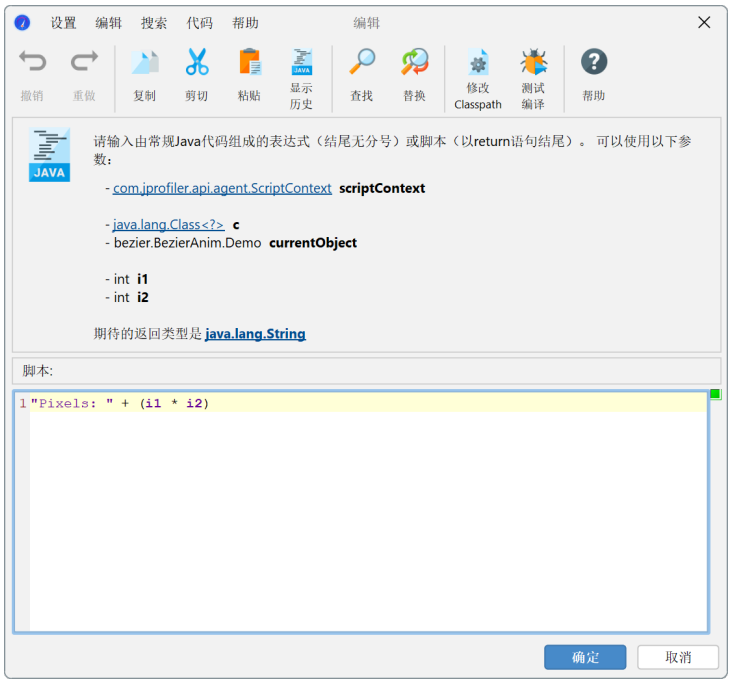


探针脚本

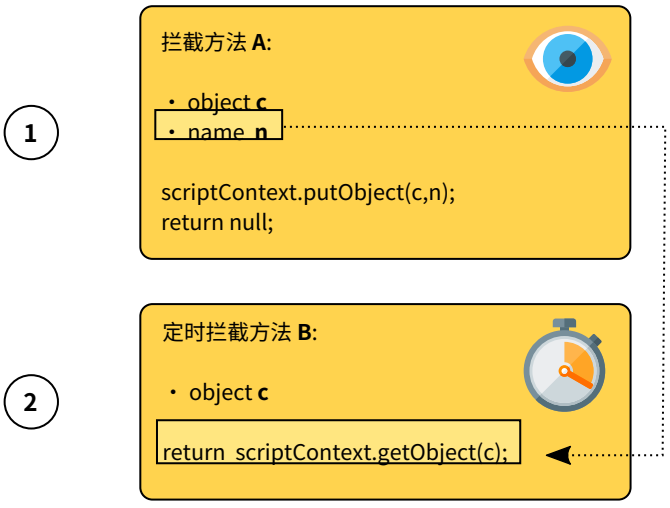
在脚本编辑器中，您可以访问被拦截方法的所有参数以及调用该方法的对象。如果您需要访问被拦截方法的返回值或任何抛出的异常，则必须编写嵌入式或注入式探针。

在此环境中，您的脚本可以构建有效负载字符串，既可以是表达式，也可以是带有返回语句的语句序列。此类脚本的最简单版本只是为一个参数返回`parameter.toString()`或为具有原始类型的参数返回`String.valueOf(parameter)`。如果返回`null`，则不会创建有效负载。

如果您同时记录CPU和探针数据，CPU部分的调用树视图将在适当的调用栈中显示到探针视图的链接。或者，您可以选择在CPU调用树视图中内联显示有效负载字符串。探针向导的"有效负载拦截->调用树注释"步骤包含此选项。



脚本可用的另一个参数是脚本上下文，一个类型为`com.jprofiler.api.agent.ScriptContext`的对象，允许您在为当前探针定义的任何脚本的调用之间存储数据。例如，假设被拦截的方法A只看到没有良好文本表示的对象，但对象和显示名称之间的关联可以通过拦截方法B获得。然后，您可以在同一个探针中拦截方法B，并将对象到文本的关联直接保存到脚本上下文中。在方法A中，您可以从脚本上下文中获取该显示文本并使用它来构建有效负载字符串。



如果这些类型的关注点变得过于复杂，您应该考虑切换到嵌入式或注入式探针API。

缺失的功能

脚本探针旨在促进自定义探针开发的简单入门，但它们缺少完整探针系统中的一些功能，您应该注意：

- 脚本探针无法进行调用树拆分。在JProfilerUI中，这是一个单独的功能，如自定义探针概念 [p. 139] 中所述。嵌入式和注入式探针直接提供调用树拆分功能。
- 脚本探针无法创建控制对象或创建自定义探针事件类型。这只能通过嵌入式或注入式探针实现。
- 脚本探针无法访问返回值或抛出的异常，不像嵌入式和注入式探针。
- 脚本探针无法处理重入拦截。如果一个方法递归调用，则只有第一次调用会被拦截。嵌入式和注入式探针为您提供对重入行为的细粒度控制。
- 不可能将默认遥测以外的遥测捆绑到探针视图中。相反，您可以使用脚本遥测功能，如自定义探针概念 [p. 139]中所示。

A.3 注入探针 (Injected Probes)

类似于脚本探针，注入探针为选定的方法定义拦截处理程序。然而，注入探针是在JProfiler GUI之外的 IDE 中开发的，并依赖于 JProfiler 提供的注入探针 API。该 API 采用宽松的 Apache License 2.0 许可，使得分发相关工件变得容易。

开始使用注入探针的最佳方式是研究 JProfiler 安装目录中的 `api/samples/simple-injected-probe` 示例。在该目录中执行 `../gradlew` 以编译并运行它。Gradle 构建文件 `build.gradle` 包含有关该示例的更多信息。`api/samples/advanced-injected-probe` 中的示例展示了探针系统的更多功能，包括控制对象。

开发环境

开发注入探针所需的探针 API 包含在具有 Maven 坐标的单个工件中

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

本手册对应的 JProfiler 版本为 15.0.1。

Jar、源代码和 Javadoc 工件发布在以下仓库中

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

您可以使用 Gradle 或 Maven 等构建工具将探针 API 添加到开发类路径中，或者使用 JProfiler 安装中的 JAR 文件

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

。要浏览 Javadoc，请访问

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

该Javadoc结合了JProfiler发布的所有API的Javadoc。`com.jprofiler.api.probe.injected` 包的概述是探索 API 的良好起点。

探针结构

注入探针是一个用`com.jprofiler.api.probe.injected.Probe`注解的类。该注解的属性公开了整个探针的配置选项。例如，如果您创建了大量不适合单独检查的探针事件，`events` 属性允许您禁用探针事件视图并减少开销。

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

在探针类中，您可以添加特别注解的静态方法以定义拦截处理程序。`PayloadInterception` 注解创建有效负载，而 `SplitInterception` 注解拆分调用树。处理程序的返回值用作有效负载或拆分字符串，具体取决于注解。与脚本探针一样，如果返回 `null`，拦截将无效。拦截方法的时间信息会自动计算。

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

如上例所示，两个注解都有一个 `method` 属性，用于使用 `MethodSpec` 定义被拦截的方法。与脚本探针不同，`MethodSpec`可以有一个空的类名，因此所有具有特定签名的方法都会被拦截，而不管类名如何。或者，您可以使用 `MethodSpec` 的 `subtypes` 属性来拦截整个类层次结构。

与脚本探针不同，脚本探针会自动提供所有参数，处理程序方法声明参数以请求感兴趣的值。每个参数都用 `com.jprofiler.api.probe.injected.parameter` 包中的注解进行注解，因此分析代理知道必须将哪个对象或原始值传递给方法。例如，将处理程序方法的参数注解为`@Parameter(0)`会注入被拦截方法的第一个参数。

被拦截方法的方法参数可用于所有拦截类型。如果您告诉分析代理拦截方法的退出而不是进入，`@ReturnValue` 或 `@ExceptionValue` 可以访问返回值或抛出的异常。可以通过`PayloadInterception`注解的`invokeOn`属性来实现。

与脚本探针不同，如果您将拦截注解的 `reentrant` 属性设置为 `true`，则注入探针处理程序可以为被拦截方法的递归调用调用。在处理程序方法中使用类型为 `ProbeContext` 的参数，您可以通过调用 `ProbeContext.getOuterPayload()` 或 `ProbeContext.restartTiming()` 来控制探针对嵌套调用的行为。

高级拦截

有时单个拦截不足以收集构建探针字符串所需的所有信息。为此，您的探针可以包含任意数量的用`Interception`注解的拦截处理程序，这些处理程序不会创建有效负载或拆分。信息可以存储在探针类的静态字段中。为了在多线程环境中实现线程安全，您应该使用`ThreadLocal`实例来存储引用类型，并使用`java.util.concurrent.atomic`包中的原子数值类型来维护计数器。

在某些情况下，您需要对方法入口和方法出口进行拦截。一个常见的情况是，如果您维护状态变量，如 `inMethodCall`，它会修改另一个拦截的行为。您可以在入口拦截中将 `inMethodCall` 设置为`true`，这是默认的拦截类型。现在，您可以在该拦截下方直接定义另一个静态方法，并用`@AdditionalInterception(invokeOn = InvocationType.EXIT)`注解它。拦截方法取自上面的拦截处理程序，因此您不必再次指定它。在方法体中，您可以将 `inMethodCall` 变量设置为`false`。

```
...

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;
    }
}

...
```

您可以在 `api/samples/advanced-injected-probe/src/main/java/AdvancedAwtEventProbe.java` 中看到此用例的工作示例。

控制对象

除非 `Probe` 注解的 `controlObjects` 属性设置为 `true`，否则控制对象视图不可见。要使用控制对象，您必须通过在处理程序方法中声明该类型的参数来获取`ProbeContext`。下面的示例代码展示了如何打开控制对象并将其与探针事件关联。

```
@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);
    }

    ...

}
```

控制对象具有定义的生命周期，探针视图在时间轴和控制对象视图中记录其打开和关闭时间。如果可能，您应该通过调用 `ProbeContext.openControlObject()` 和 `ProbeContext.closeControlObject()` 来显式打开和关闭控制对象。否则，您必须声明一个用 `@ControlObjectName` 注解的静态方法来解析控制对象的显示名称。

如果您的处理程序方法返回 `Payload` 实例而不是 `String`，则探针事件可以与控制对象关联。`ProbeContext.createPayload()` 方法接受一个控制对象和一个探针事件类型。允许的事件类型的枚举必须通过 `Probe` 注解的 `customTypes` 属性注册。

控制对象必须在时间测量开始时指定，这对应于方法入口。在某些情况下，有效负载字符串的名称只有在方法退出时才可用，因为它依赖于返回值或其他拦截。在这种情况下，您可以使用`ProbeContext.createPayloadWithDeferredName()` 创建一个没有名称的有效负载对象。在其下方定义一个用 `@AdditionalInterception(invokeOn = InvocationType.EXIT)` 注解的拦截处理程序，并从该方法返回一个 `String`，它将自动用作有效负载字符串。

覆盖线程状态

在测量数据库驱动程序或本机连接器到外部资源的执行时间时，有时需要告诉 JProfiler 将某些方法放入不同的线程状态。例如，将数据库调用放入“Net I/O”线程状态是有用的。如果通信机制不使用标准的 Java I/O 设施，而是使用某种本机机制，这将不会自动发生。

使用 `ThreadState.NETIO.enter()` 和 `ThreadState.exit()` 调用对，分析代理会相应地调整线程状态。

```
...

@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();
}

...
```

部署

部署注入探针有两种方式，具体取决于您是否希望将它们放在类路径上。使用 VM 参数

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

分析代理设置了一个单独的探针类路径。探针类路径可以包含目录和类文件，在 Windows 上用 ';' 分隔，在其他平台上用 ':' 分隔。分析代理将扫描探针类路径并找到所有探针定义。

如果您更容易将探针类放在类路径上，可以设置 VM 参数

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

为一个完全限定类名的逗号分隔列表。对于这些类名中的每一个，分析代理将尝试加载一个注入探针。

A.4 嵌入式探针

如果您控制探针目标软件组件的源代码，您应该编写嵌入式探针而不是注入式探针。

编写注入式探针时，大部分初始工作都集中在指定拦截的方法和选择注入对象作为处理方法的参数。对于嵌入式探针，这不是必需的，因为您可以直接从监控的方法中调用嵌入式探针API。嵌入式探针的另一个优点是部署是自动的。探针与您的软件一起发布，并在应用程序被分析时出现在JProfiler UI中。您唯一需要发布的依赖项是一个小的JAR文件，该文件在Apache 2.0许可证下授权，主要由空方法体组成，作为分析代理的挂钩。

开发环境

开发环境与注入式探针相同，不同之处在于工件名称是`jprofiler-probe-embedded`而不是`jprofiler-probe-injected`，并且您将JAR文件与您的应用程序一起发布，而不是在单独的项目中开发探针。您需要为软件组件添加嵌入式探针的探针API包含在单个JAR工件中。在javadoc中，探索API时从`com.jprofiler.api.probe.embedded`包概览开始。

就像注入式探针一样，嵌入式探针也有两个示例。在`api/samples/simple-embedded-probe`中，有一个示例可以帮助您开始编写嵌入式探针。在该目录中执行`../gradlew`以编译和运行它，并研究gradle构建文件`build.gradle`以了解执行环境。有关更多功能，包括控制对象，请转到`api/samples/advanced-embedded-probe`中的示例。

有效负载探针

类似于注入式探针，您仍然需要一个探针类用于配置目的。探针类必须扩展`com.jprofiler.api.probe.embedded.PayloadProbe`或`com.jprofiler.api.probe.embedded.SplitProbe`，具体取决于您的探针是收集有效负载还是拆分调用树。使用注入式探针API，您可以在处理方法上使用不同的注解进行有效负载收集和拆分。而嵌入式探针API没有处理方法，需要将此配置转移到探针类本身。

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

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

而注入式探针使用注解进行配置，您可以通过重写探针基类的方法来配置嵌入式探针。对于有效负载探针，唯一的抽象方法是`getName()`，所有其他方法都有默认实现，您可以根据需要重写。例如，如果您想禁用事件视图以减少开销，可以重写`isEvents()`以返回`false`。

为了收集有效负载并测量其相关的时间，您可以使用一对`Payload.enter()`和`Payload.exit()`调用。

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

Payload.enter()调用接收探针类作为参数，因此分析代理知道哪个探针是调用的目标，Payload.exit()调用自动与同一探针关联，并接收有效负载字符串作为参数。如果您错过了退出调用，调用树将被破坏，因此这应该始终在try块的finally子句中完成。

如果测量的代码块不产生任何值，您可以调用Payload.execute方法，该方法接收有效负载字符串和一个Runnable。使用Java 8+，lambda或方法引用使得此方法调用非常简洁。

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

在这种情况下，有效负载字符串必须事先知道。还有一个版本的execute接收Callable。

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

接收Callable的签名的问题在于Callable.call()抛出一个受检的Exception，因此您必须捕获它或在包含的方法上声明它。

控制对象

有效负载探针可以通过调用Payload类中的适当方法来打开和关闭控制对象。它们通过将控制对象和自定义事件类型传递给接收控制对象的Payload.enter()或Payload.execute()方法的版本与探针事件相关联。

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

控制对象视图必须在探针配置中显式启用，自定义事件类型也必须在探针类中注册。

```
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;
    }
}
```

如果您没有显式打开和关闭控制对象，探针类必须重写`getControlObjectName`以便解析所有控制对象的显示名称。

拆分探针

拆分探针基类没有抽象方法，因为它可以仅用于拆分调用树而不添加探针视图。在这种情况下，最小的探针定义只是

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

拆分探针的重要配置之一是它们是否应该是可重入的。默认情况下，仅拆分顶级调用。如果您希望也拆分递归调用，可以重写`isReentrant()`以返回`true`。如果您在探针类中重写`isPayloads()`以返回`true`，拆分探针还可以创建探针视图并将拆分字符串发布为有效负载。

要执行拆分，请调用`Split.enter()`和`Split.exit()`。

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

与有效负载收集相反，拆分字符串必须与探针类一起传递给`Split.enter()`方法。同样，`Split.exit()`的调用必须可靠，因此它应该在try块的finally子句中。`Split`还提供了带有`Runnable`和`Callable`参数的`execute()`方法，可以通过一次调用执行拆分。

遥测

发布嵌入式探针的遥测特别方便，因为在相同的类路径中，您可以直接访问应用程序中的所有静态方法。就像注入式探针一样，在探针配置类中为静态公共方法添加`@Telemetry`注解并返回一个数值。有关更多信息，请参阅探针概念[p. 139]一章。嵌入式和注入式探针API的`@Telemetry`注解是等效的，只是它们位于不同的包中。

嵌入式和注入式探针API之间的另一个并行功能是使用ThreadState类修改线程状态的能力。同样，该类在两个API中都存在，只是包不同。

部署

在使用JProfilerUI进行分析时，不需要特殊步骤来启用嵌入式探针。然而，只有在第一次调用Payload或Split时，探针才会被注册。只有在那时，相关的探针视图才会在JProfiler中创建。如果您希望探针视图从一开始就可见，就像内置和注入式探针一样，您可以调用

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

对于有效负载探针和

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

对于拆分探针。

您可能会考虑是否有条件地调用Payload和Split的方法，可能由命令行开关控制以最小化开销。然而，这通常不是必需的，因为方法体是空的。没有附加分析代理，除了构建有效负载字符串之外，不会产生任何开销。考虑到探针事件不应在微观尺度上生成，它们将相对较少地创建，因此构建有效负载字符串应该是一个相对不显著的工作。

对于容器的另一个担忧可能是您不希望在类路径上暴露外部依赖项。您的容器用户也可以使用嵌入式探针API，这将导致冲突。在这种情况下，您可以将嵌入式探针API遮蔽到您自己的包中。JProfiler仍然会识别遮蔽的包并正确地对API类进行检测。如果构建时遮蔽不切实际，您可以提取源代码存档并将类作为项目的一部分。

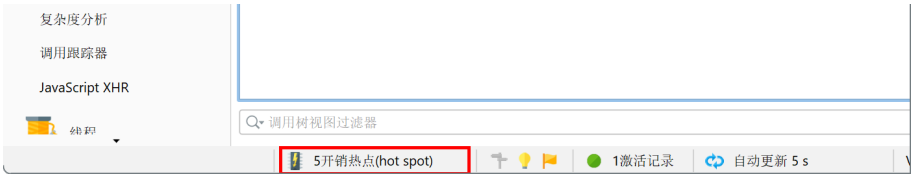
B 调用树功能详解

B.1 自动调优和忽略的方法

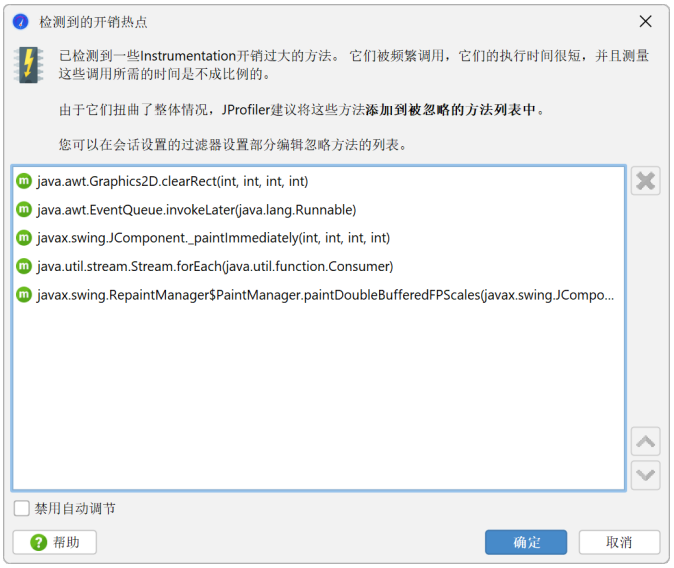
如果方法调用记录类型设置为插桩，所有被分析的类的方法都会被插桩。这会对执行时间非常短的方法造成显著的开销。如果这些方法被非常频繁地调用，那么测量到的这些方法的时间将会过高。此外，由于插桩，热点编译器可能无法优化它们。在极端情况下，这些方法会成为主要的热点，尽管这在未插桩的运行中并不成立。一个例子是读取下一个字符的XML解析器方法。这样的一个方法返回得非常快，但可能在短时间内被调用数百万次。

当方法调用记录类型设置为采样时，这个问题不存在。然而，采样不提供调用次数，只显示较长的方法调用，并且在使用采样时，几个视图没有其完整的功能。

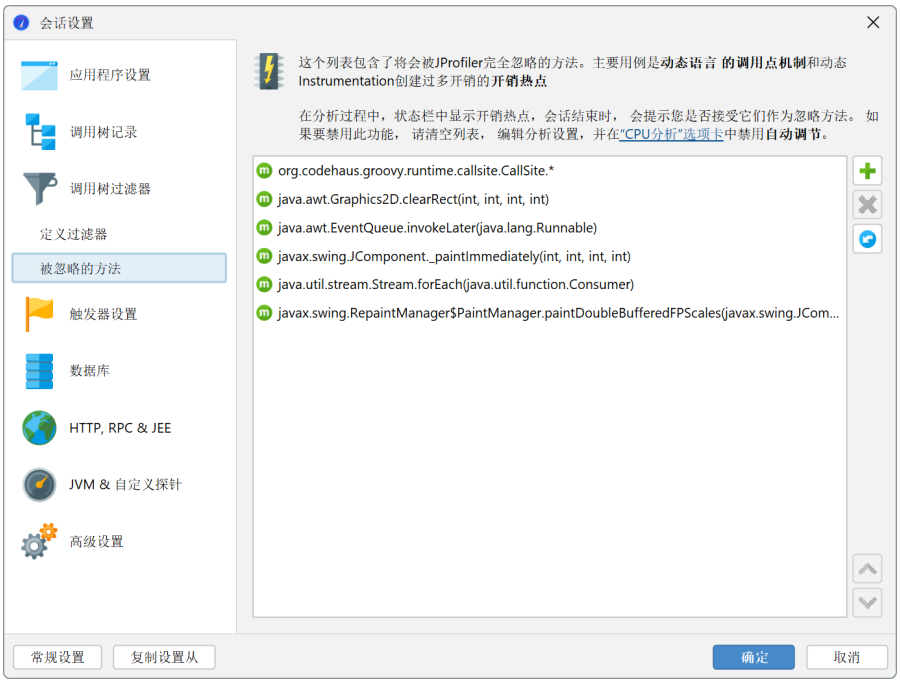
为了缓解插桩的问题，JProfiler有一个称为自动调优的机制。分析代理会不时检查具有高插桩开销的方法，并将它们传输到JProfiler GUI。在状态栏中，会显示一个警告条目，提示存在开销热点。



您可以点击该状态栏条目查看检测到的开销热点，并选择将它们接受到忽略的方法列表中。这些忽略的方法将不会被插桩。当会话结束时，会显示相同的对话框。

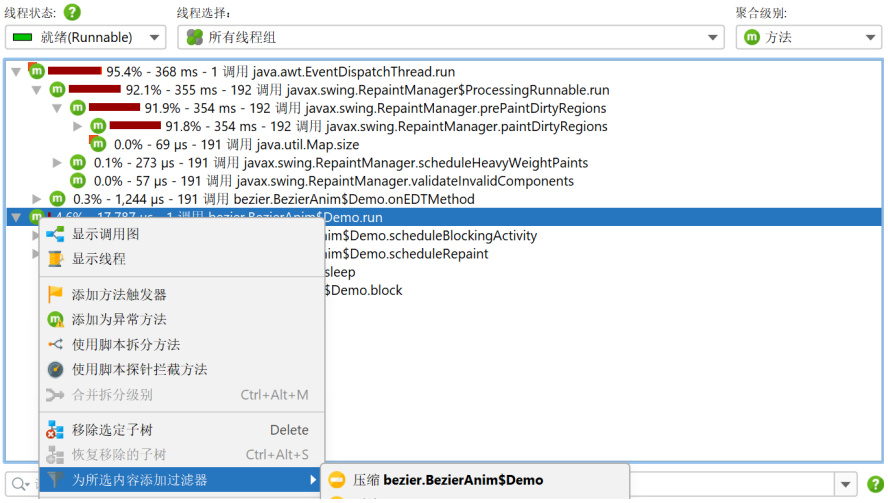


应用新的配置文件设置后，所有忽略的方法将在调用树中缺失。它们的执行时间将被添加到调用方法的自耗时中。如果稍后发现某些忽略的方法在分析视图中是不可或缺的，您可以在会话设置中的忽略的方法选项卡中将它们移除。



忽略的方法的默认配置包括用于动态方法分派的Groovy调用点类，但这使得跟踪实际调用链变得困难。

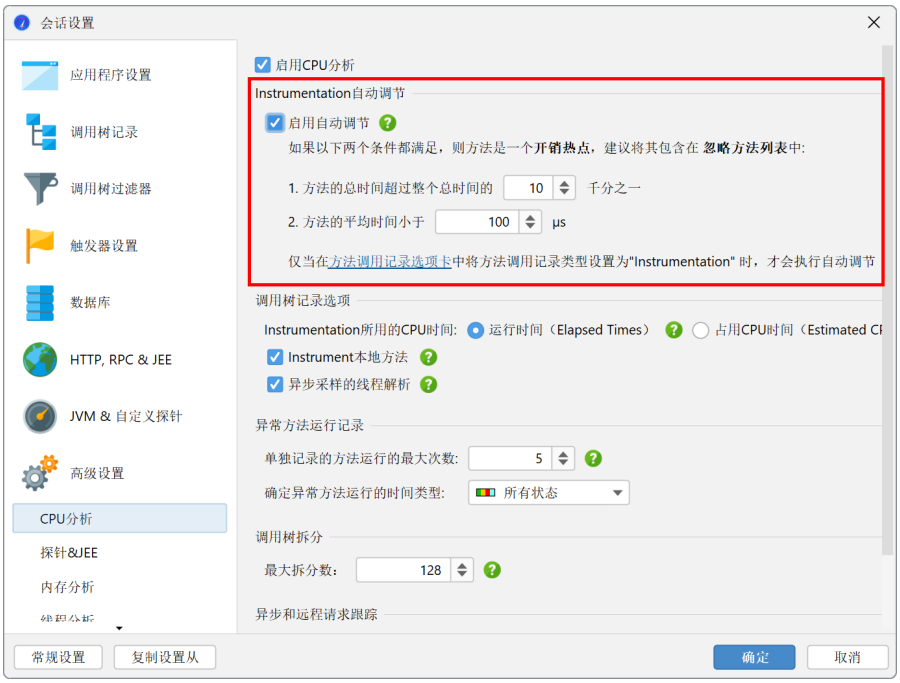
如果您想手动添加忽略的方法，可以在会话设置中进行，但更简单的方法是选择调用树中的一个方法，并从上下文菜单中调用忽略方法操作。



在过滤器设置中，您还可以通过将过滤器条目的类型设置为“忽略”来忽略整个类或包。从选择中添加过滤器菜单包含依赖于所选节点的操作，并建议忽略类或包直到顶级包。根据所选节点是紧凑分析的还是被分析的，您还会看到更改过滤器为相反类型的操作。

如果您不想看到任何关于自动调优的消息，可以在配置文件设置中禁用它。此外，您可以配置用于确定开销热点的标准。如果同时满足以下两个条件，则一个方法被视为开销热点：

- 所有调用的总时间超过了线程中整个总时间的千分之一的阈值
- 其平均时间低于微秒的绝对阈值


会话设置
应用程序设置
调用树记录
调用树过滤器
触发器设置
数据库
HTTP, RPC & JEE
JVM & 自定义探针
高级设置
CPU分析
探针&JEE
内存分析
启用CPU分析
Instrumentation自动调节
启用自动调节
如果以下两个条件都满足，则方法是一个开销热点，建议将其包含在 忽略方法列表中:
1. 方法的总时间超过整个总时间的 10 千分之一
2. 方法的平均时间小于 100 µs
仅当在方法调用记录选项卡中将方法调用记录类型设置为"Instrumentation" 时，才会执行自动调节
调用树记录选项
Instrumentation所用的CPU时间: 运行时间（Elapsed Times） 占用CPU时间（Estimated CP
Instrument本地方法
异步采样的线程解析
异常方法运行记录
单独记录的方法运行的最大次数: 5
确定异常方法运行的时间类型: 所有状态
调用树拆分
最大拆分数: 128
异步和远程请求跟踪
常规设置
复制设置从
确定
取消

B.2 异步和远程请求跟踪

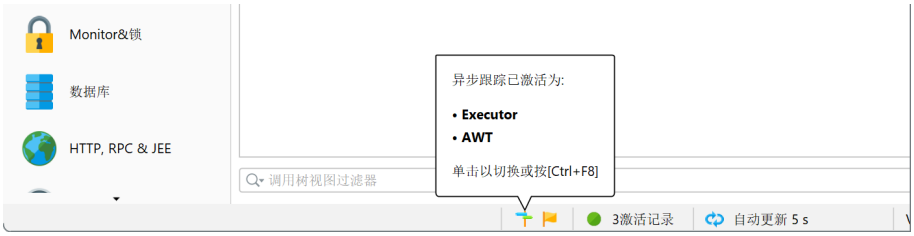
异步任务执行是一种常见的实践，无论是在普通Java代码中，还是在反应式框架中更是如此。源文件中相邻的代码现在在两个或多个不同的线程上执行。对于调试和分析，这些线程变化带来了两个问题：一方面，不清楚调用操作的开销有多大。另一方面，无法将昂贵的操作追溯到导致其执行的代码。

JProfiler根据调用是否留在同一个JVM中提供不同的解决方案。如果异步执行发生在调用它的同一个JVM中，“内联异步执行”调用树分析[p.175]计算出一个包含调用点和执行站点的单一调用树。如果请求发送到远程JVM，调用树[p. 51]包含指向调用点和执行站点的超链接，因此您可以在显示涉及JVM的分析会话的不同JProfiler顶级窗口之间无缝导航。

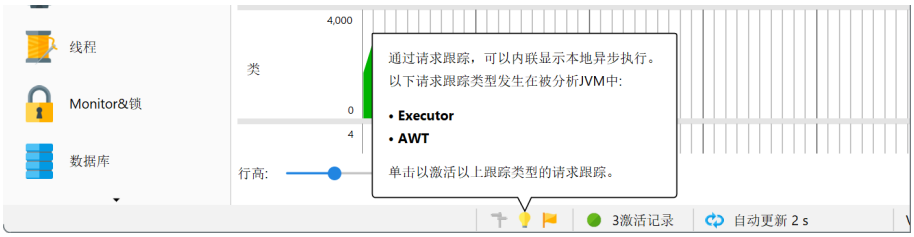
启用异步和远程请求跟踪

异步机制可以通过多种方式实现，无法以通用方式检测在单独线程或不同JVM上启动任务的语义。JProfiler明确支持几种常见的异步和远程请求技术。您可以在请求跟踪设置中启用或禁用它们。默认情况下，请求跟踪未启用。还可以在会话启动对话框中配置请求跟踪，该对话框在会话启动之前直接显示。

在JProfiler的主窗口中，状态栏指示是否启用了某些异步和远程请求跟踪类型，并为您提供配置对话框的快捷方式。

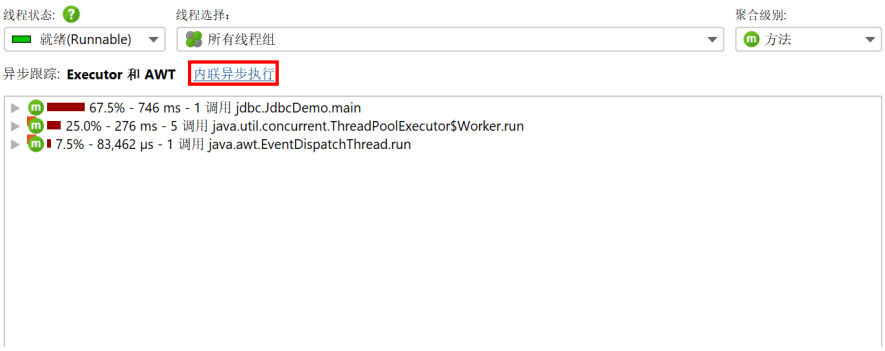


JProfiler检测到在被分析的JVM中使用了未激活的异步请求跟踪类型，并在状态栏中的异步和远程请求跟踪图标旁边显示一个💡通知图标。通过单击通知图标，您可以激活检测到的跟踪类型。异步和远程请求跟踪可能会产生大量开销，只有在必要时才应激活。



异步跟踪

如果至少激活了一种异步跟踪类型，CPU、分配和探针记录的调用树和热点视图将显示有关所有激活的跟踪类型的信息，并带有一个计算“内联异步执行”调用树分析的按钮。在该分析的结果视图中，所有异步执行的调用树通过一个“异步执行”节点与调用点连接。默认情况下，异步执行测量不会添加到调用树中的祖先节点中。因为有时查看聚合值很有用，所以分析顶部的复选框允许您在适当的地方这样做。



在另一个线程上卸载任务的最简单方法是启动一个新线程。使用JProfiler，您可以通过激活“线程启动”请求跟踪类型来跟踪线程从创建到执行站点的过程。然而，线程是重量级对象，通常用于重复调用，因此这种请求跟踪类型更适合调试目的。

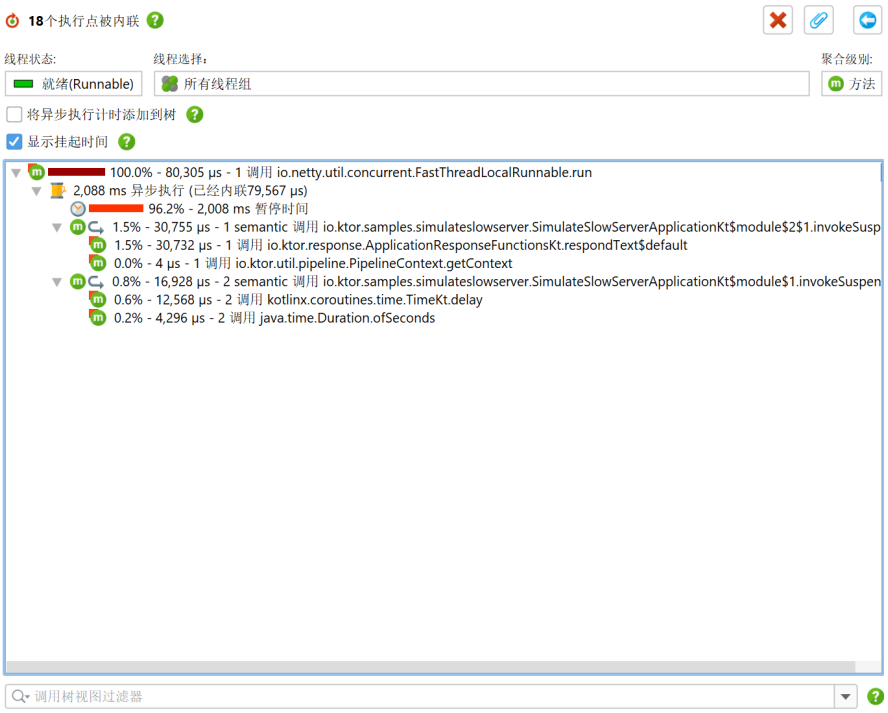
在其他线程上启动任务的最重要和通用的方法是使用`java.util.concurrent`包中的执行器。执行器也是许多处理异步执行的高级第三方库的基础。通过支持执行器，JProfiler支持处理多线程和并行编程的整个类库。

除了上述通用情况，JProfiler还支持JVM的两个GUI工具包：AWT和SWT。两个工具包都是单线程的，这意味着有一个特殊的事件调度线程可以操作GUI小部件并执行绘图操作。为了不阻塞GUI，长时间运行的任务必须在后台线程上执行。然而，后台线程通常需要更新GUI以指示进度或完成。这是通过安排一个`Runnable`在事件调度线程上执行的特殊方法来完成的。

在GUI编程中，您经常需要跟踪多个线程更改以连接原因和结果：用户在事件调度线程上启动一个操作，该操作又通过执行器启动后台操作。完成后，该执行器将一个操作推送到事件调度线程。如果最后一个操作造成性能问题，它距离原始事件有两个线程更改。

最后，JProfiler支持Kotlin协程[1]，Kotlin的多线程解决方案为所有Kotlin后端实现。异步执行本身是启动协程的点。Kotlin协程的调度机制是灵活的，实际上可以涉及在当前线程上启动，在这种情况下，“异步执行”节点有一个内联部分，然后在节点文本中单独报告。

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

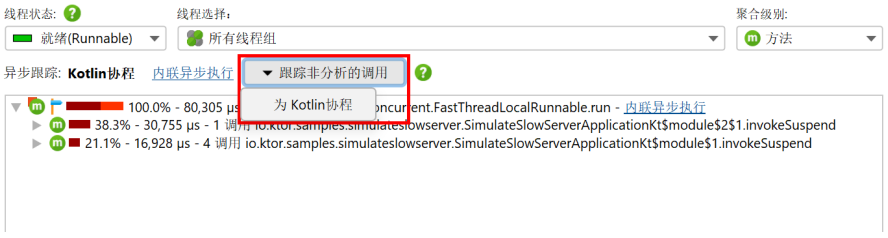


挂起的方法可以中断执行，然后可能在不同的线程上恢复。检测到挂起的方法有一个额外的 “挂起”图标，工具提示显示实际调用次数与方法的语义调用次数。Kotlin协程可以故意挂起，但由于它们不绑定到线程，等待时间不会出现在调用树中的任何地方。要查看协程执行完成所需的总时间，在“异步执行”节点下添加一个 “挂起”时间节点，该节点捕获协程的整个挂起时间。根据您是对CPU时间还是异步执行的挂钟时间感兴趣，您可以通过分析顶部的“显示挂起时间”复选框动态添加或删除这些节点。

跟踪未分析的调用点

默认情况下，执行器和Kotlin协程跟踪仅跟踪调用点位于被分析类中的异步执行。这是因为框架和库可以以与您自己的代码执行无直接关系的方式使用这些异步机制，添加的调用和执行站点只会增加开销和干扰。然而，跟踪未分析的调用点也有用例。例如，框架可以启动一个Kotlin协程，然后在其上执行您自己的代码。

如果检测到未分析类中的此类调用点，调用树和热点视图中的跟踪信息会显示相应的通知消息。在实时会话中，您可以直接从这些视图中分别为执行器和Kotlin协程跟踪启用未分析调用点的跟踪。这些选项可以随时在会话设置对话框的“CPU分析”步骤中更改。



重要的是要理解，Kotlin协程只能在其启动时CPU记录处于活动状态时进行跟踪。如果您稍后开始CPU记录，Kotlin协程的异步执行将无法内联。JProfiler会像检测未分析类中的调用点一样通知您。如果您需要分析在应用程序开始时启动的长寿命协程，则使用附加模式不是一个选项。在这种情况下，使用-agentpath VM参数 [p. 11]启动JVM，并在启动时开始CPU记录。

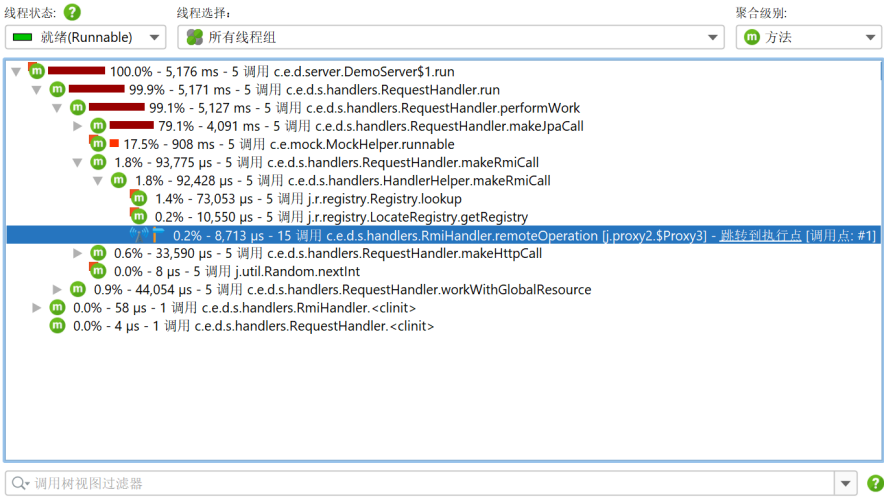
远程请求跟踪

对于选定的通信协议，JProfiler能够插入元数据并跨JVM边界跟踪请求。支持的技术有：

- HTTP：客户端侧的HttpURLConnection、java.net.http.HttpClient、Apache Http Client 4.x、Apache Async Http Client 4.x、OkHttp 3.9+，服务器侧的任何Servlet-API实现或不带Servlets的Jetty
- 额外支持异步JAX-RS调用，适用于Jersey Async Client 2.x、RestEasy Async Client 3.x、Cxf Async Client 3.1.1+
- Web服务：JAX-WS-RI、Apache Axis2和Apache CXF
- RMI
- gRPC
- 远程EJB调用：JBoss 7.1+和Weblogic 11+

为了能够在JProfiler中跟踪请求，您必须分析两个VM并在不同的JProfiler顶级窗口中同时打开它们。这适用于实时会话和快照。如果目标JVM当前未打开，或者在远程调用时CPU记录未激活，单击调用点超链接将显示错误消息。

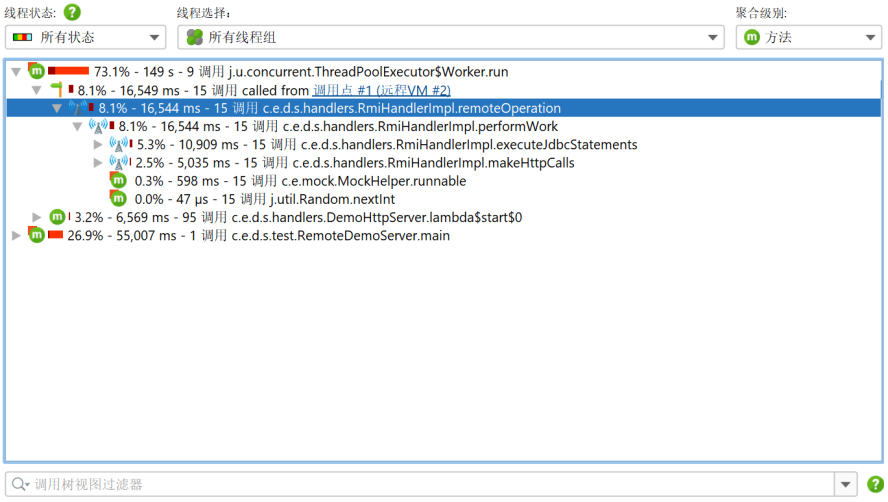
在跟踪远程请求时，JProfiler在涉及的JVM的调用树中明确显示调用点和执行站点。JProfiler中的调用点是记录的远程请求执行之前的最后一个被分析的方法调用。它在位于不同VM的执行站点启动任务。JProfiler允许您通过调用树视图中显示的超链接在调用点和执行站点之间跳转。



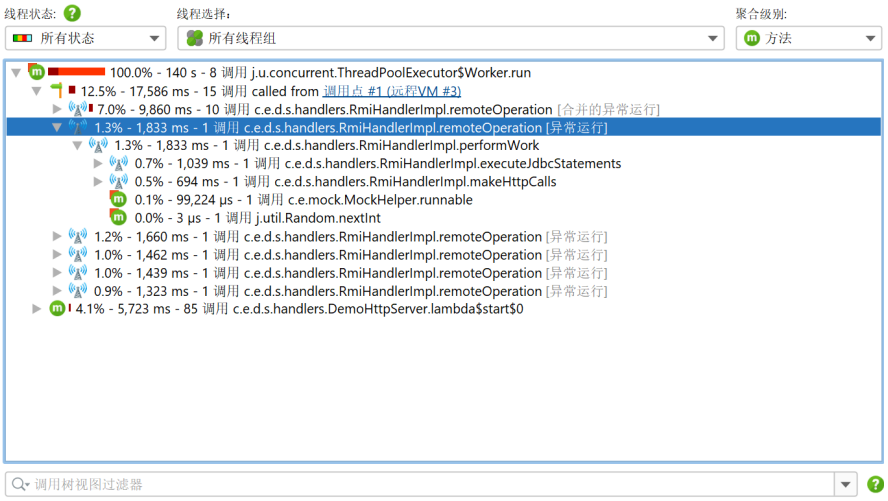
对于所有线程，调用点在远程请求跟踪方面具有相同的身份。这意味着当您从调用点跳转到执行站点或反之时，没有线程解析，跳转总是激活“所有线程组”以及“所有线程状态”线程状态选择，以确保目标是显示树的一部分。

调用点和执行站点之间是1:n关系。一个调用点可以在多个执行站点启动远程任务，尤其是如果它们位于不同的远程VM中。在同一个VM中，单个调用点的多个执行站点不太常见，因为它们必须在不同的调用栈中发生。如果一个调用点调用多个执行站点，您可以在对话框中选择其中一个。

执行站点是调用树中的一个合成节点，包含由一个特定调用点启动的所有执行。执行站点节点中的超链接将您带回到该调用点。



如果同一个调用点重复调用同一个执行站点，执行站点将显示其所有调用的合并调用树。如果不希望这样，您可以使用异常方法 [p. 180]功能进一步拆分调用树，如下图所示。



与仅从单个调用点引用的执行站点不同，调用点本身可以链接到多个执行站点。通过调用点的数字ID，您可以识别相同的调用点，如果您看到它从不同的执行站点引用。此外，调用点显示远程VM的ID。被分析VM的ID可以在状态栏中看到。它不是JProfiler内部管理的唯一ID，而是一个显示ID，从一开始，每打开一个新的被分析VM就递增。

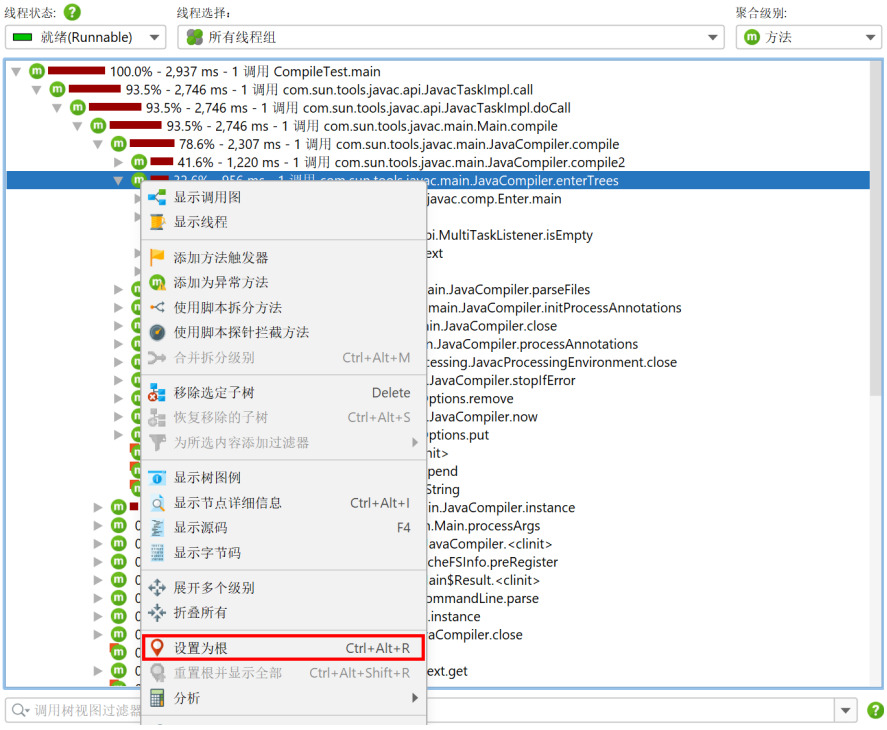
B.3 查看调用树的部分

调用树通常包含过多的信息。当您想减少显示的细节时，有几种可能性：您可以将显示的数据限制为一个特定的子树，移除所有不需要的数据，或使用更粗粒度的过滤器来显示方法调用。所有这些策略都由JProfiler支持。

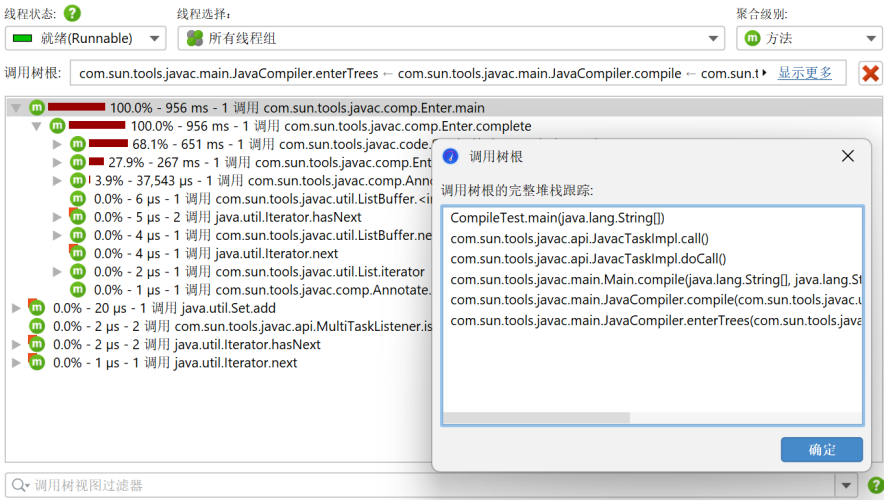
设置调用树根

如果您分析的用例由多个顺序运行的任务组成，每个子树都可以单独分析。一旦找到此类子任务的入口点，周围的调用树就只是一个干扰，而子树中的时间百分比不便地引用了整个调用树的根。

为了专注于特定的子树，JProfiler在调用树和分配调用树视图中提供了设为根上下文操作。

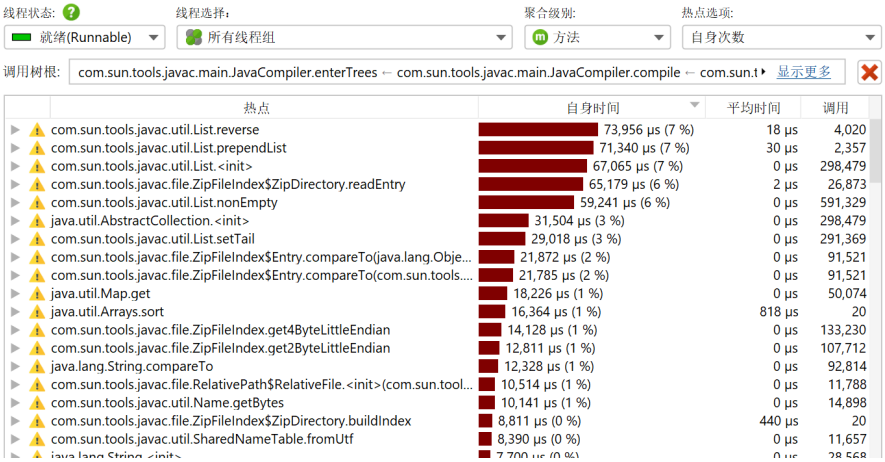


设置调用树根后，关于所选根的信息会显示在视图的顶部。一个可滚动的标签显示了通向根的最后几个栈元素，点击显示更多按钮可以显示整个调用树根的详细对话框。



当您递归使用设为根操作时，调用栈前缀将简单地连接在一起。要返回到以前的调用树，您可以使用调用树历史记录的返回按钮一次撤销一个根更改，或在上下文菜单中使用重置根并显示全部操作一步返回到原始树。

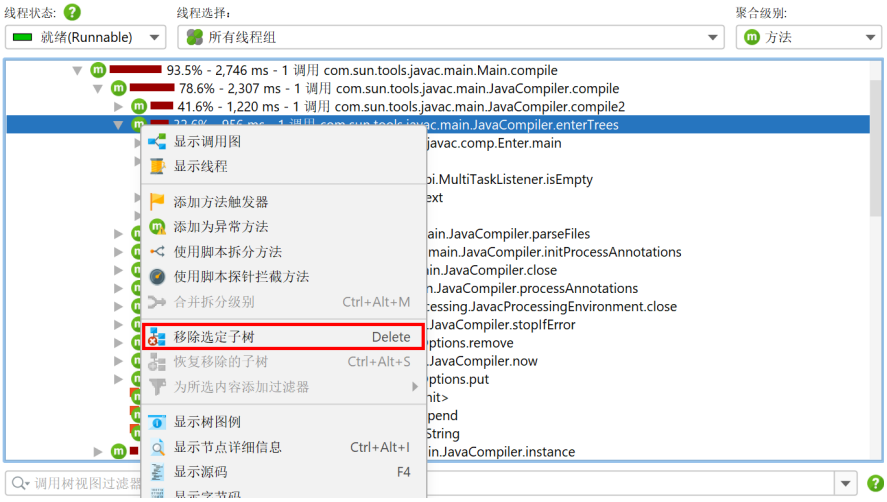
更改调用树根最重要的是，热点视图将仅显示为所选根计算的数据，而不是整个树。在热点视图的顶部，您将看到当前的调用树根，就像在调用树视图中一样，以提醒您显示数据的上下文。



移除调用树的部分

有时查看调用树在某个方法不存在时的样子是有帮助的。例如，当您必须一次性修复多个性能问题时，因为您正在使用无法像开发环境中那样快速迭代的生产系统快照。解决主要性能问题后，您希望分析第二个问题，但只有在第一个问题从树中消除后才能清晰地看到。

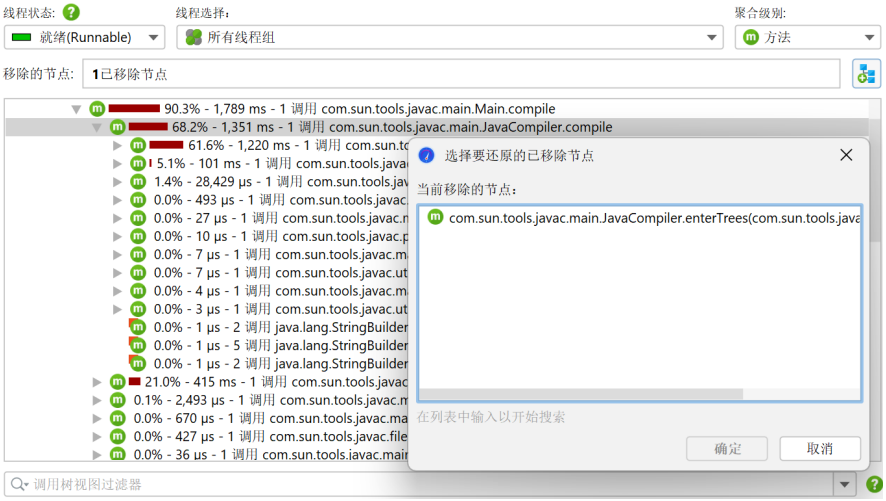
通过选择调用树中的节点并按　键，或从上下文菜单中选择移除选定子树，可以将节点及其子树一起移除。祖先节点中的时间将相应地校正，就好像隐藏的节点不存在一样。



有三种移除模式。使用移除所有调用模式，JProfiler会在整个调用树中搜索所选方法的所有调用，并将其及其整个子树一起移除。仅移除子树选项仅移除选定的子树。最后，将自耗时设为零将所选节点保留在调用树中，但将其自耗时设为零。这对于像Thread.run这样的容器节点很有用，因为它们可能包含大量未被分析的类的时间。

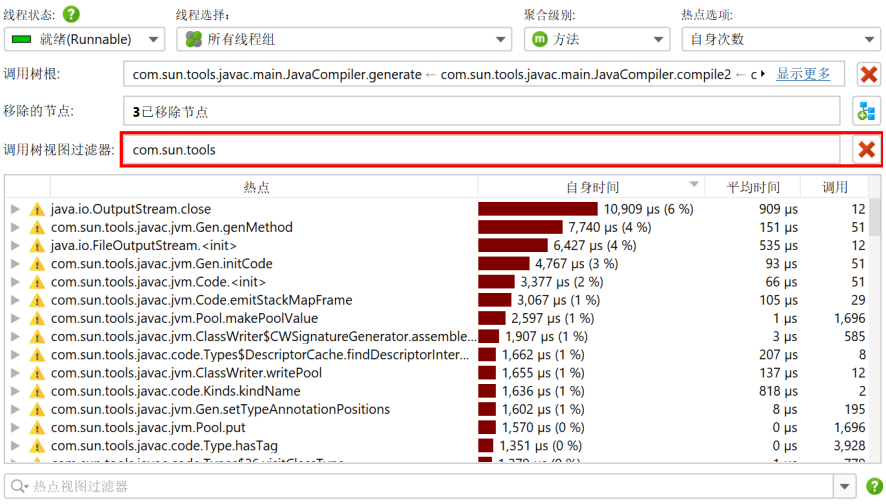
就像设为根操作一样，移除的节点会影响热点视图。通过这种方式，您可以检查如果这些方法被优化到不再是重要贡献的程度，热点将是什么样子。

当您移除一个节点时，调用树和热点视图的标题区域将显示一行，包含移除节点的计数和恢复移除的子树按钮。点击该按钮将弹出一个对话框，您可以在其中选择应再次显示的移除元素。



调用树视图过滤器

调用中对热点视图中显示数据有影响的第三个功能是视图过滤器。当您更改调用树过滤器时，它对计算的热点[p.51]有很大影响。为了强调与调用树视图的这种相互依赖性，热点视图在视图上方显示调用树视图过滤器的一行，并带有一个按钮以移除附加过滤器。

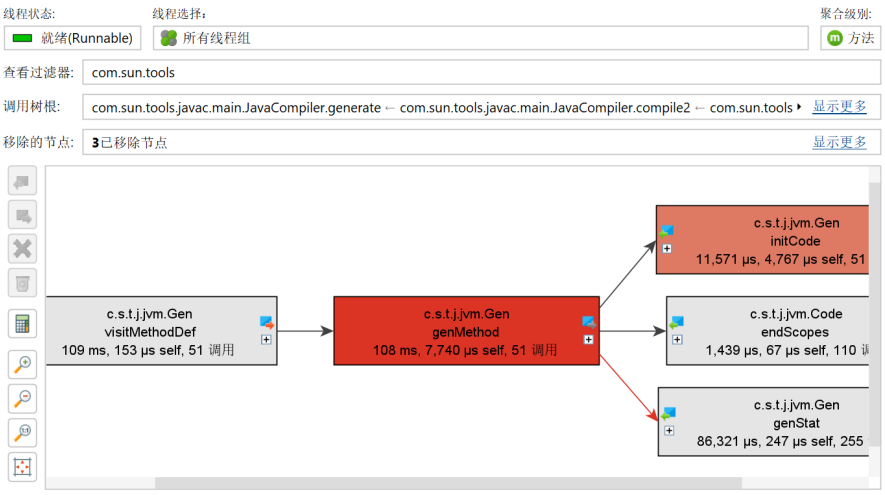


设置调用树根、移除调用树的部分和视图过滤器可以一起使用，限制是视图过滤器必须最后设置。一旦在调用树中配置了视图过滤器，设为根和移除选定子树操作将不再起作用。

与调用图的交互

在调用树或热点视图中调用显示图形操作将显示一个图形，该图形仅限于相同的调用树根，不包括移除的方法，并使用配置的调用树视图过滤器。在图形的顶部，关于这些更改的信息以类似于调用树的形式显示。

在图形视图本身中创建新图形时，向导中的复选框让您选择哪些调用树调整功能应被考虑用于调用图的计算。每个复选框仅在调用树视图中当前使用相应功能时可见。


线程状态:
就绪(Runnable)
线程选择:
所有线程组
聚合级别:
方法
查看过滤器:
com.sun.tools
调用树根:
com.sun.tools.javac.main.JavaCompiler.generate ← com.sun.tools.javac.main.JavaCompiler.compile2 ← com.sun.tools
显示更多
移除的节点:
3已移除节点
显示更多
c.s.t.j.jvm.Gen
visitMethodDef
109 ms, 153 µs self, 51 调用
c.s.t.j.jvm.Gen
genMethod
108 ms, 7,740 µs self, 51 调用
c.s.t.j.jvm.Gen
initCode
11,571 µs, 4,767 µs self, 51
c.s.t.j.jvm.Code
endScopes
1,439 µs, 67 µs self, 110
c.s.t.j.jvm.Gen
genStat
86,321 µs, 247 µs self, 255

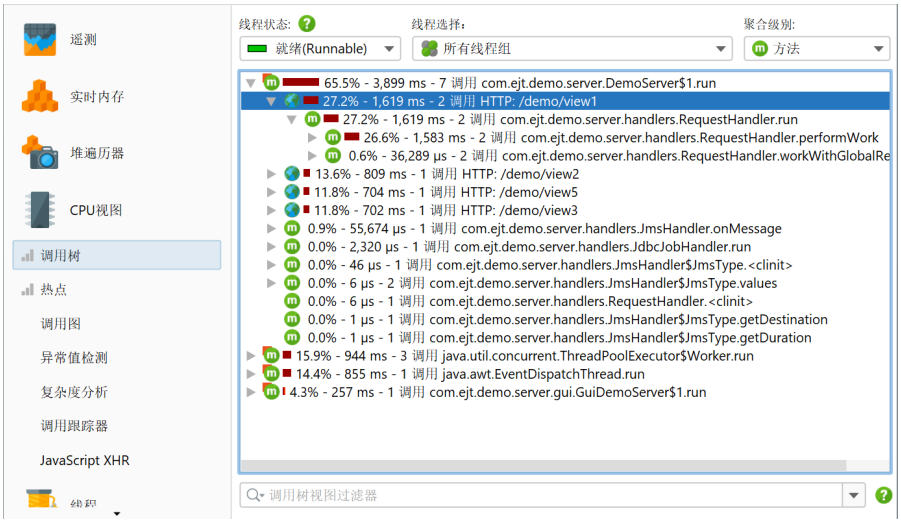
B.4 调用树拆分

调用树是为相同调用栈的重复调用累积的。这是因为内存开销和需要合并数据以便于理解。然而，有时您可能希望在选定的点打破累积，以便可以单独查看调用树的部分。

JProfiler具有通过在调用栈中插入特殊节点来拆分调用树的概念，并显示从插入节点上方的方法调用中提取的语义信息。这些拆分节点允许您直接在调用树中查看额外的有效负载信息，并单独分析其包含的子树。每种拆分类型都可以通过上下文菜单中的操作动态合并和取消合并，并对拆分节点的总数进行限制，以便内存开销是有限的。

调用树拆分和探针

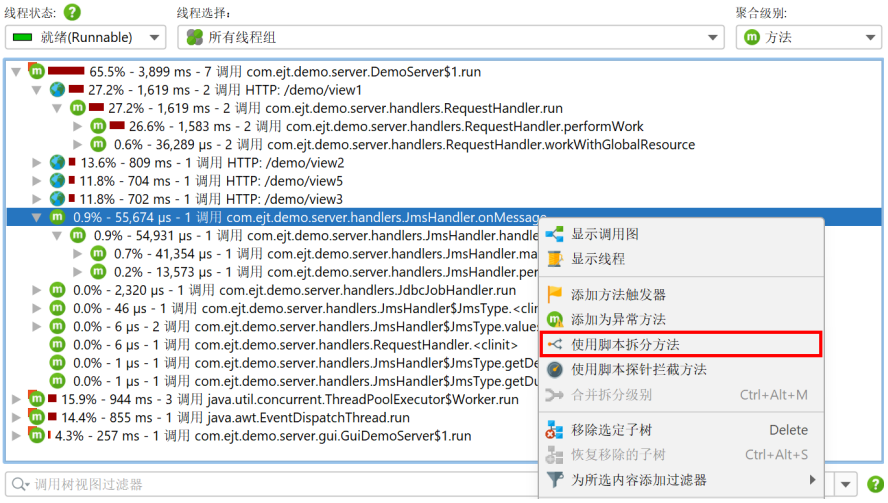
探针 [p. 96]可以根据它们在选定的感兴趣方法中收集的信息拆分调用树。例如，“HTTP服务器”探针为每个不同的URL拆分调用树。在这种情况下，拆分是高度可配置的，因此您可以仅包含URL的所需部分、来自servlet上下文的其他信息，甚至可以生成多个拆分级别。



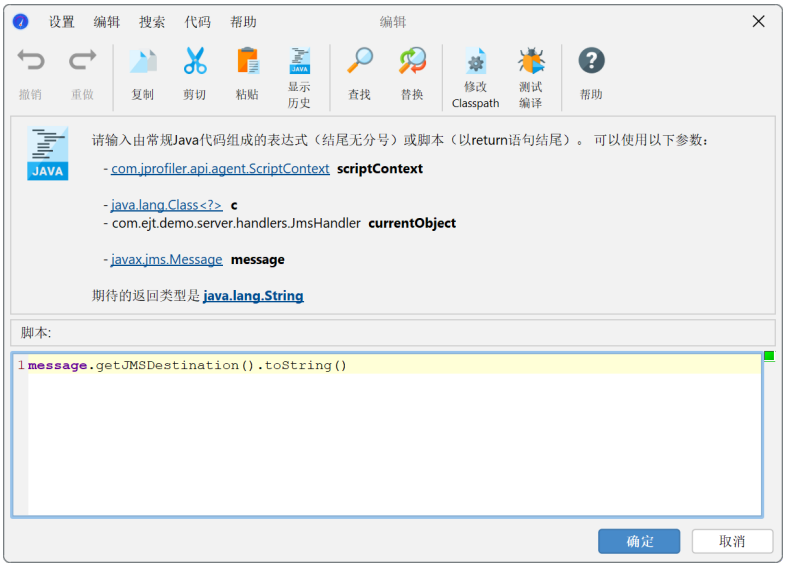
如果您编写自己的探针，可以使用 嵌入式[p. 154]和 注入式[p. 149]自定义探针系统以相同方式拆分调用树。

使用脚本拆分方法

探针可用的相同拆分功能可以直接在调用树中使用，通过使用脚本拆分方法操作。在下面的屏幕截图中，我们希望为JMS消息处理程序拆分调用树，以单独查看不同类型消息的处理。



您只需输入一个返回字符串的脚本，而不是编写探针。该字符串用于在选定方法处对调用树进行分组，并显示在拆分节点中。如果返回null，则当前方法调用不会被拆分，并像往常一样添加到调用树中。

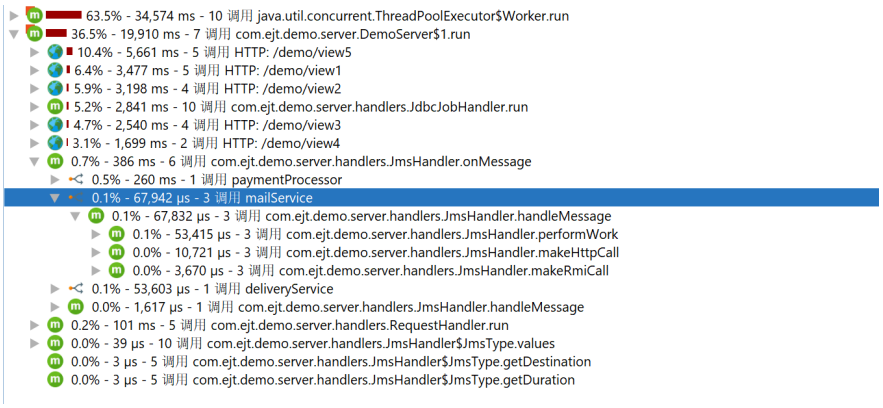


脚本可以访问许多参数。它传递了选定方法的类、非静态方法的实例以及所有方法参数。此外，您还可以获得一个ScriptContext对象，用于存储数据。如果需要从同一脚本的先前调用中回忆一些值，可以在上下文中调用getObject/putObject和getLong/putLong方法。例如，您可能只想在第一次看到特定方法参数值时进行拆分。然后可以使用

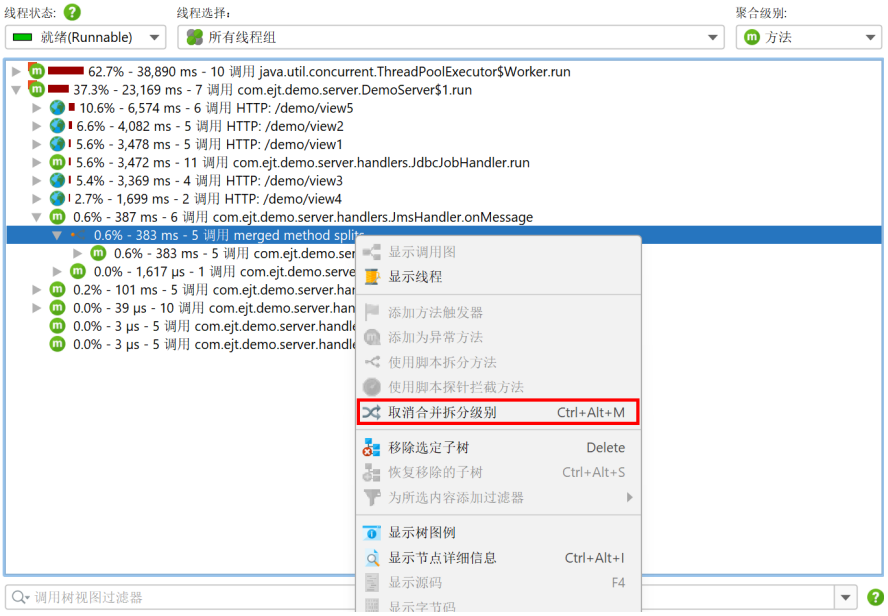
```
if (scriptContext.getObject(text) != null) {
    scriptContext.putObject(text);
    return text;
} else {
    return null;
}
```

作为拆分脚本的一部分。

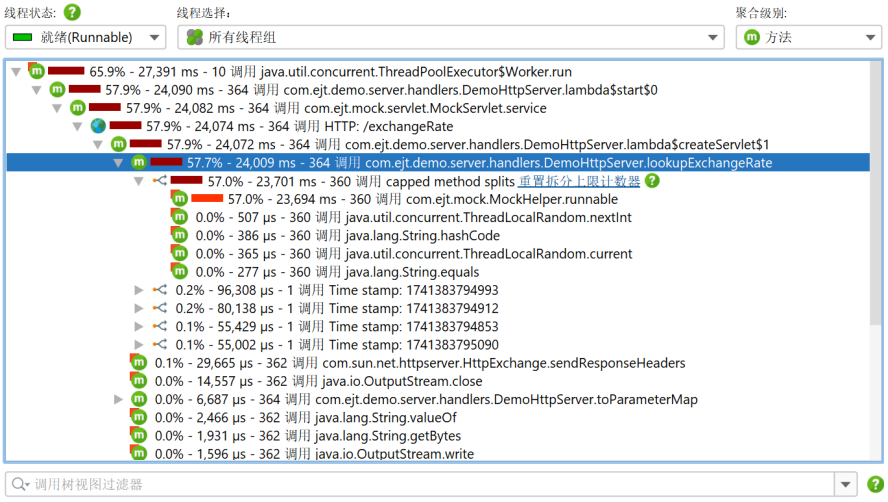
拆分节点插入在选定方法的下方。对于上面屏幕截图中的示例，我们现在可以分别看到每个JMS消息目的地的处理代码。



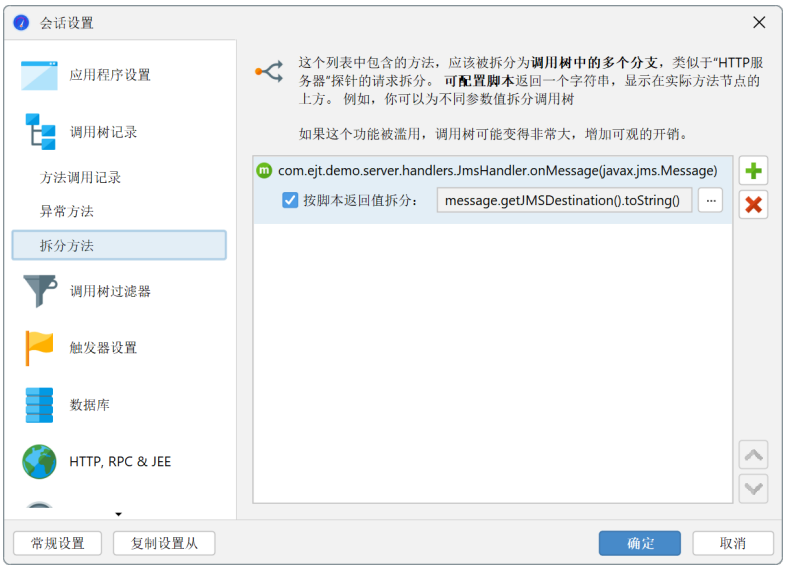
拆分位置绑定到方法，而不是选定的调用栈。如果相同的方法在调用树的其他地方出现，也将被拆分。如果使用合并拆分级别操作，所有拆分将合并为一个节点。该节点为您提供了再次取消合并拆分的机会。



如果生成了太多拆分，一个标记为封顶方法拆分的节点将包含所有进一步的拆分调用，累积为一个单一的树。通过节点中的超链接，您可以重置封顶计数器并记录更多的拆分节点。要永久增加最大拆分数，可以在配置文件设置中增加封顶。



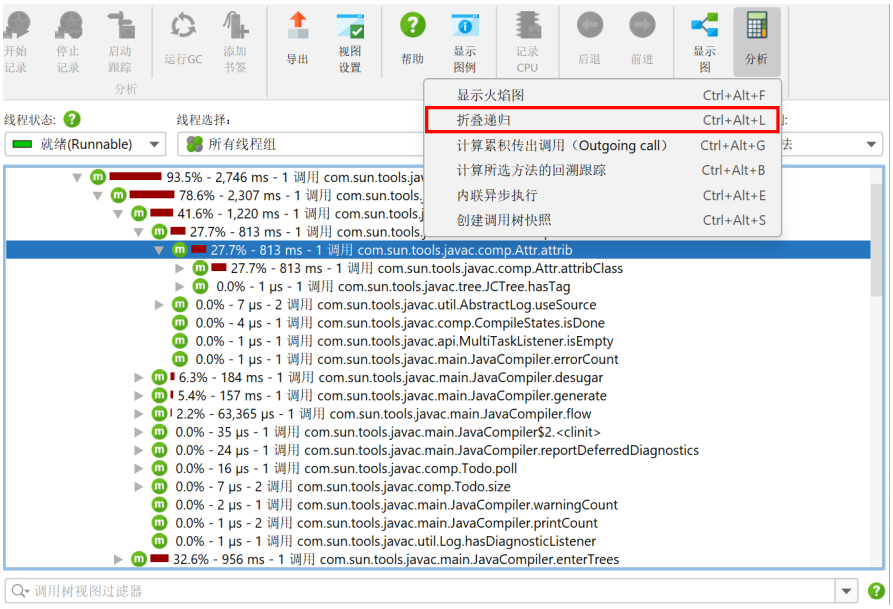
要在创建拆分方法后编辑它们，请转到会话设置对话框。如果不再需要特定的拆分方法，但希望将其保留以备将来使用，可以通过脚本配置前的复选框禁用它。这比仅在调用树中合并它更好，因为记录开销可能很大。



B.5 调用树分析

调用树 [p. 51]显示了JProfiler记录的实际调用栈。在分析调用树时，可以对调用树应用一些转换，以便于解释。这些转换可能耗时，并以一种与调用树视图中的功能不兼容的方式改变输出格式，因此会创建新的视图来显示分析结果。

要执行这样的分析，请在调用树视图中选择一个节点，然后从工具栏或上下文菜单中选择一个调用树分析操作。



在调用树视图下方将创建一个嵌套视图。如果再次调用相同的分析操作，分析将被替换。要同时保留多个分析结果，可以固定结果视图。在这种情况下，下次进行相同类型的分析时将创建一个新视图。对于固定的视图，视图顶部会显示一个重命名按钮，可以用来更改在左侧视图选择器中显示的名称。

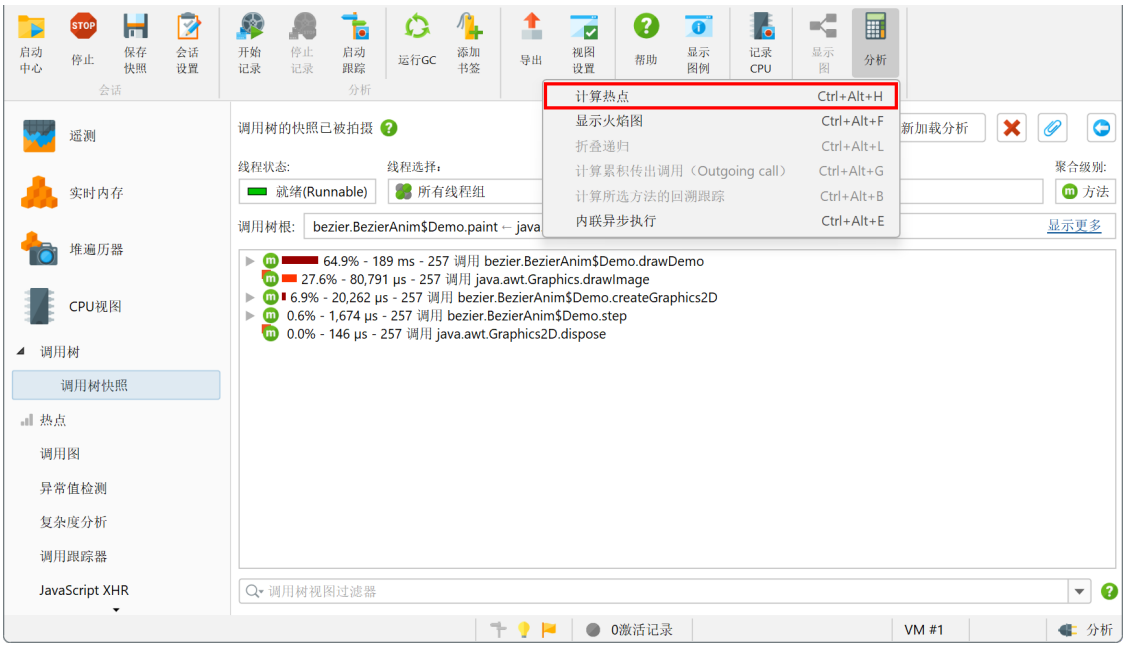
在实时会话中，结果视图不会与调用树一起更新，而是显示分析时的数据。要重新计算当前数据的分析，请使用重新加载操作。如果调用树本身需要重新计算，例如在禁用自动更新的分配树中，重新加载操作也会处理这一点。

调用树快照

"创建调用树快照"分析只是创建当前调用树的静态副本。这对于在不保存和打开JProfiler快照的情况下比较不同的用例非常有用。此外，它还提供了一种在调用树仍在记录时处理其冻结副本的方法。

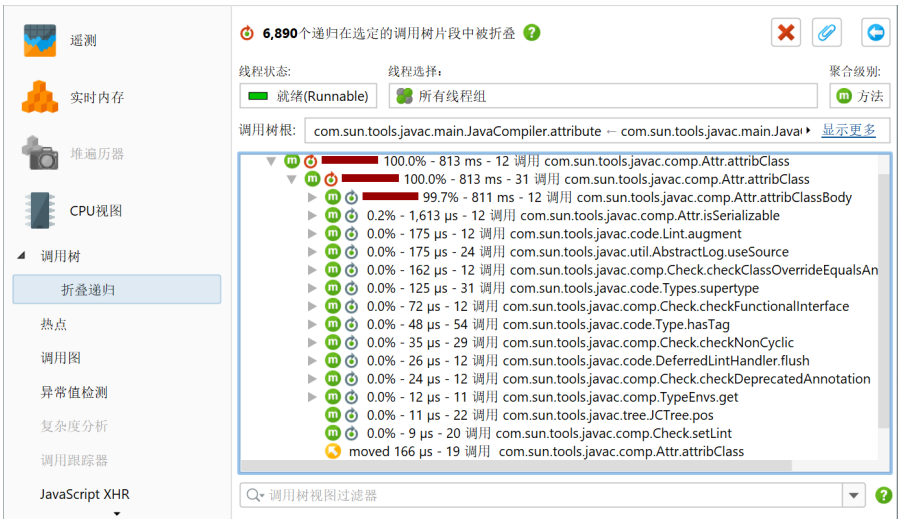
"创建调用树快照"分析仅在"CPU视图"部分的"调用树"视图中可用。如果固定调用树快照视图，可以同时拥有多个调用树快照。与其他分析不同，调用树快照会保存在JProfiler快照中，因为它们构成了独立的数据集。

除了调用树视图中可用的调用树分析外，调用树快照还具有"计算热点"操作，该操作计算父视图的热点，类似于"CPU视图"部分的"热点"视图。从嵌套在调用树快照视图下方的视图中访问的所有分析都使用其父调用树快照的数据，而不是顶级调用树视图的数据。



折叠递归

使用递归的编程风格会导致难以分析的调用树。"折叠递归"调用树分析计算一个将所有递归折叠的调用树。调用树中当前选择的父节点作为分析的 调用树根 [p. 166]。要分析整个调用树，请选择一个顶级节点。



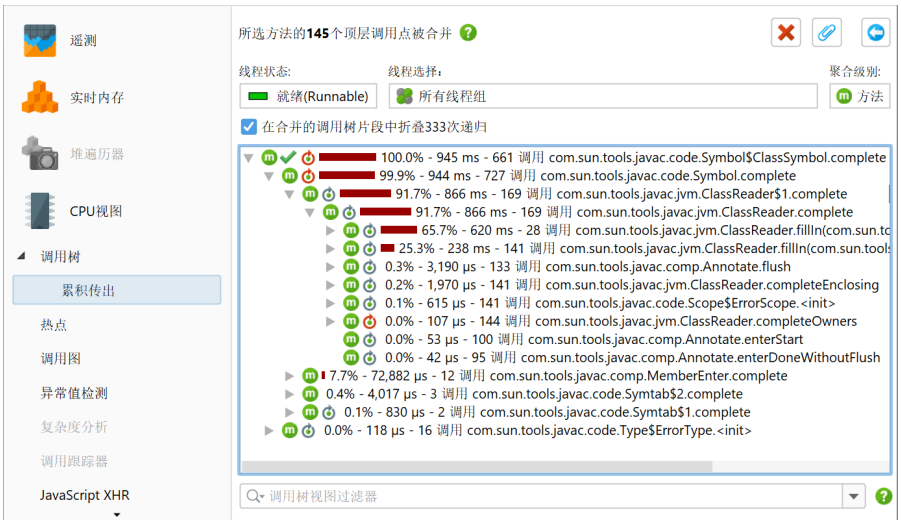
当同一方法在调用栈的更高位置已经被调用时，检测到递归。在这种情况下，子树从调用树中移除并缝合回该方法的第一次调用。调用树中的那个节点前缀一个图标，其工具提示显示递归次数。在该节点下，不同深度的栈被合并。合并栈的数量也显示在工具提示中。折叠递归的总数显示在标题中，位于为原始调用树设置的调用树参数信息上方。

对于简单递归，合并栈的数量是递归次数加一。因此，递归工具提示显示"1递归"的节点将包含一个树，其中节点在其递归工具提示中显示"2合并栈"。在更复杂的情况下，递归是嵌套的，并产生重叠的合并调用树，因此合并栈的数量从栈深度到栈深度不等。

在从调用树中移除子树以在更高位置合并的点上，插入一个特殊的 "移动节点" 占位符。

分析累积的外部调用

在调用树中，可以看到选定方法的外部调用，但仅限于该方法被调用的一个特定调用栈。相同的感兴趣方法可能在不同的调用栈中被调用，通常分析所有这些调用的累积调用树以获得更好的统计数据是有用的。"计算累积外部调用"分析显示一个调用树，该树汇总了选定方法的所有外部调用，而不管该方法是如何被调用的。

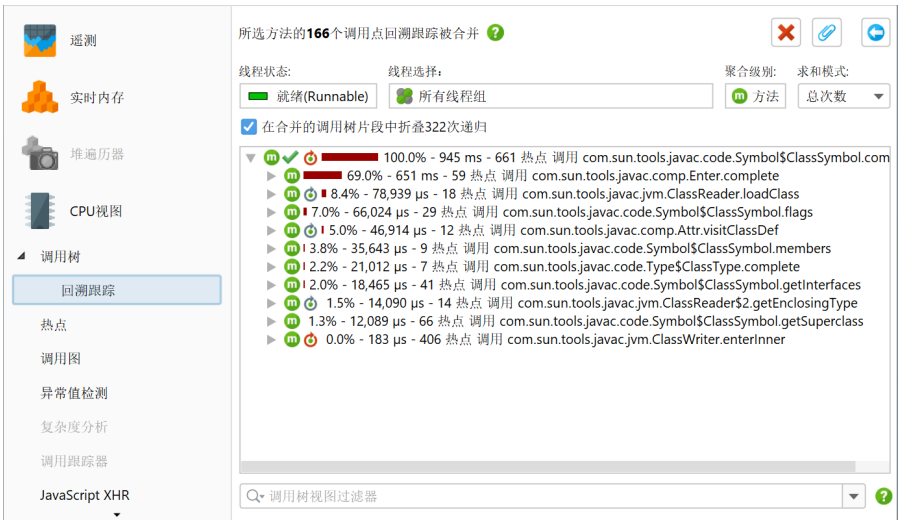


对于选定的方法，JProfiler收集其所有顶级调用，而不考虑递归调用，并在结果树中累积它们。标题显示在该过程中汇总的顶级调用点的数量。

在视图顶部，有一个复选框，允许您在结果树中折叠递归，类似于"折叠递归"分析。如果递归被折叠，顶级节点和外部调用的第一级显示与方法调用图相同的数字。

计算回溯

"计算回溯"分析补充了"计算累积外部调用"分析。与后者一样，它汇总了选定方法的所有顶级调用，而不考虑递归调用。然而，它不是显示外部调用，而是显示促成选定方法调用的回溯。调用起始于最深的节点，并向选定方法的顶部推进。



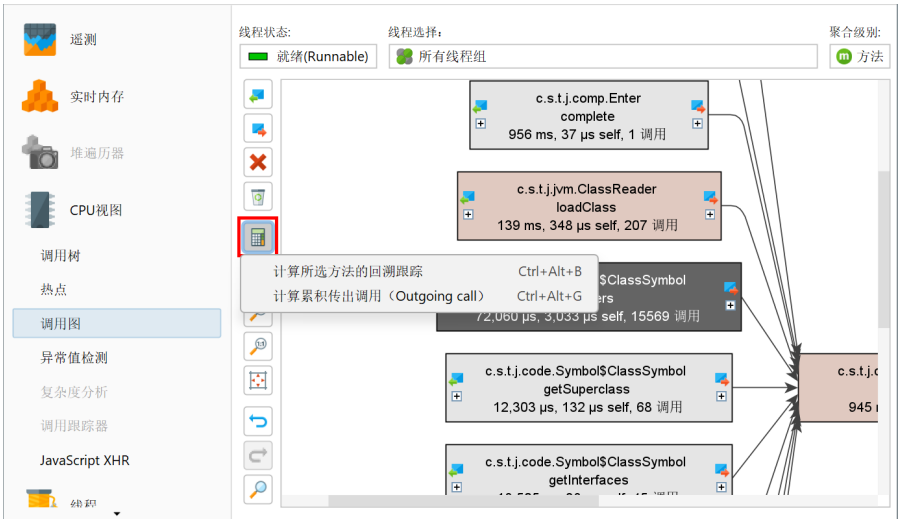
此分析类似于热点视图，只是默认情况下它汇总选定方法的总时间而不是自耗时，而热点视图仅显示自耗时占总时间显著比例的方法。在视图顶部有一个标记为汇总模式的单选按钮组，可以设置为自耗时。选择该选项后，选定方法的汇总值与热点视图中的默认模式匹配。

在回溯中，回溯节点上的调用计数和时间仅与选定方法相关。它们显示了沿着特定调用栈的调用对选定方法值的贡献。类似于"计算累积外部调用"分析，您可以折叠递归，回溯中的第一级相当于方法调用图中的传入调用。

调用图中的调用树分析

在调用图中，每个方法都是唯一的，而在调用树中，方法可以出现在多个调用栈中。对于一个选定的方法，"计算累积外部调用"和"计算回溯"分析是调用树和调用图视角之间的桥梁。它们将选定方法置于中心，并以树的形式显示外部和传入调用。使用显示调用图操作，您可以随时切换到完整图形。

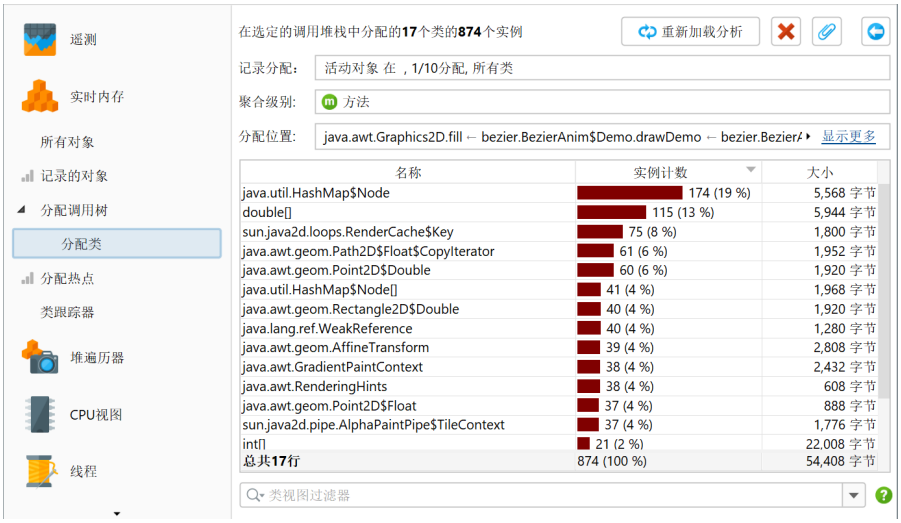
有时，您希望从相反的方向切换视角，从图形切换到树视图。当您在调用图中工作时，可以为图中任何选定节点显示累积外部调用和回溯为树，使用与调用图中相同的调用树分析。



在IntelliJ IDEA集成 [p. 129]中，编辑器边栏中显示的调用图包含直接显示这些树的操作。

显示分配的类

与之前的调用树分析略有不同的是，分配调用树和分配热点视图中的"显示类"分析。它不会将调用树转换为另一棵树，而是显示一个包含所有分配类的表。结果视图类似于 记录对象视图 [p. 68]，但仅限于特定的分配点。



在显示调用树的分析结果视图中，"计算累积外部调用"和"计算回溯到选定方法"分析均可用。调用它们会创建具有独立参数的新顶级分析。先前分析结果视图中的任何调用树移除都不会反映在新的顶级分析中。

另一方面，显示类操作在从调用树分析结果视图中使用时不会创建新的顶级分析。相反，它会创建一个嵌套分析，该分析位于原始视图下方的两级。

C 高级 CPU 分析视图

C.1 异常值检测和异常方法记录

在某些情况下，问题不在于方法的平均调用时间，而在于方法偶尔会表现异常。在调用树中，所有方法调用都是累积的，因此一个每10000次调用中有一次耗时100倍于预期的频繁调用方法不会在总时间中留下明显的痕迹。

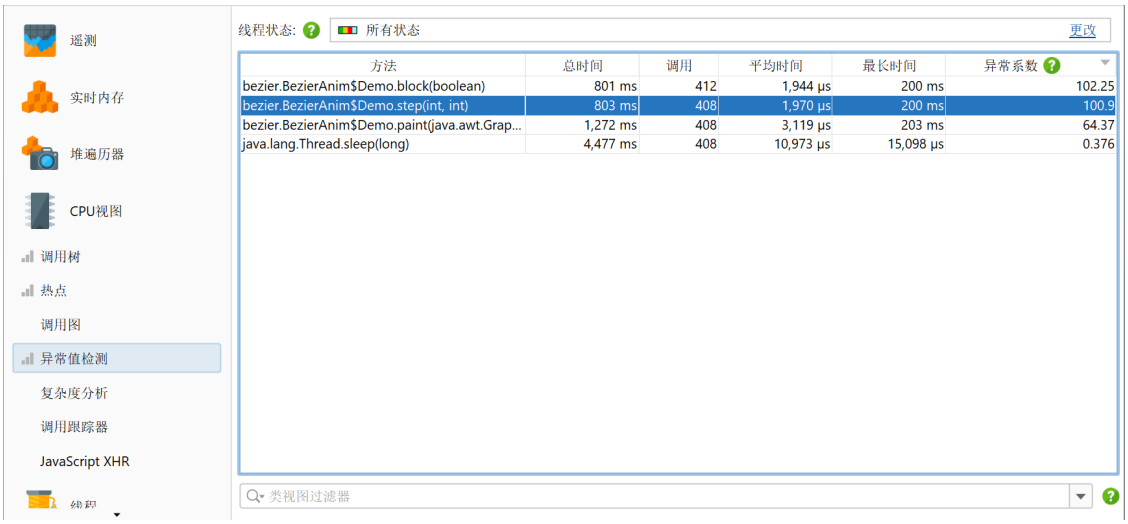
为了解决这个问题，JProfiler在调用树中提供了异常值检测视图和异常方法记录功能。

异常值检测视图

异常值检测视图显示了每个方法的调用持续时间和调用次数的信息，以及单次调用测得的最大时间。最大调用时间与平均时间的偏差显示了所有调用持续时间是否在一个狭窄的范围内，或者是否存在显著的异常值。异常值系数计算为

```
(最大时间 - 平均时间) / 平均时间
```

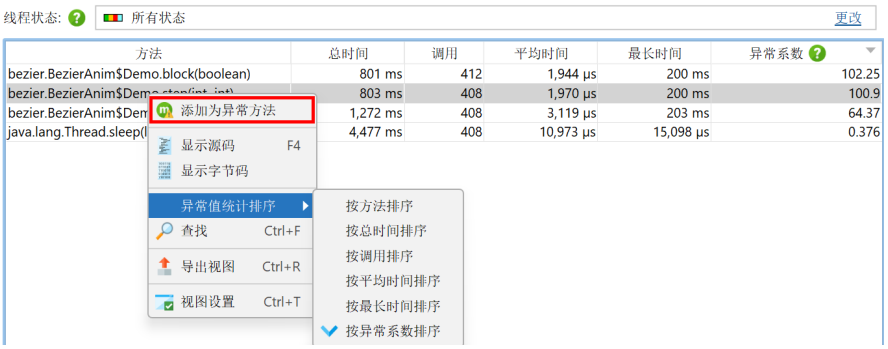
可以帮助您在这方面量化方法。默认情况下，表格按异常值系数从高到低排序。如果已记录CPU数据，则异常值检测视图中的数据可用。



为了避免来自仅调用几次的方法和运行时间极短的方法的过多混乱，可以在视图设置中设置最大时间和调用次数的下限。默认情况下，异常值统计中仅显示最大时间超过10毫秒且调用次数大于10的方法。

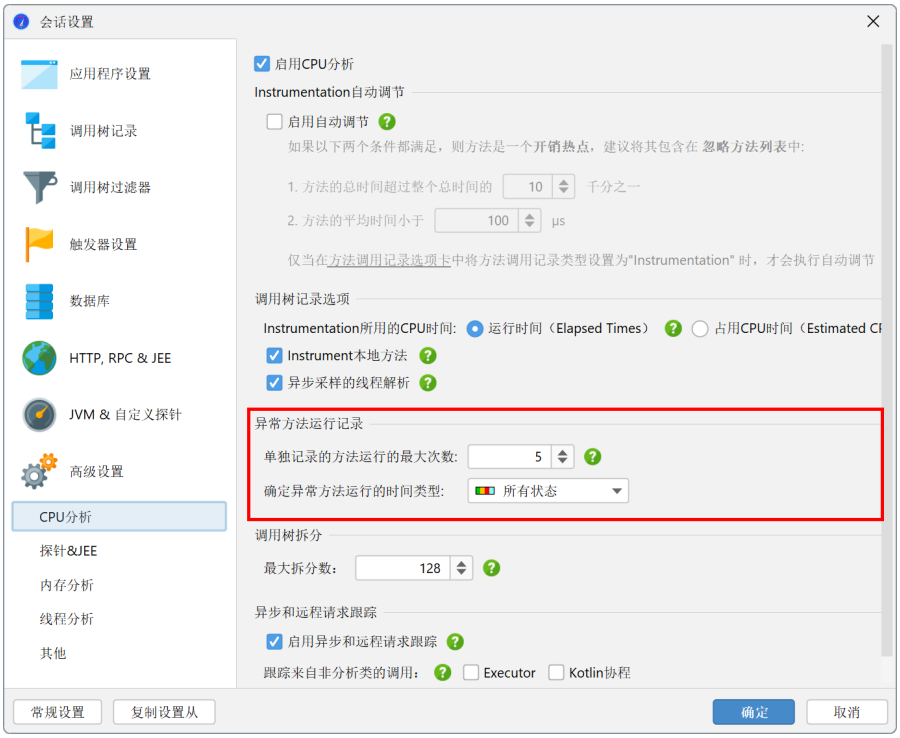
配置异常方法记录

一旦您确定了一个存在异常调用持续时间的问题方法，您可以在上下文菜单中将其添加为异常方法。同样的上下文菜单操作也可以在调用树视图中使用。

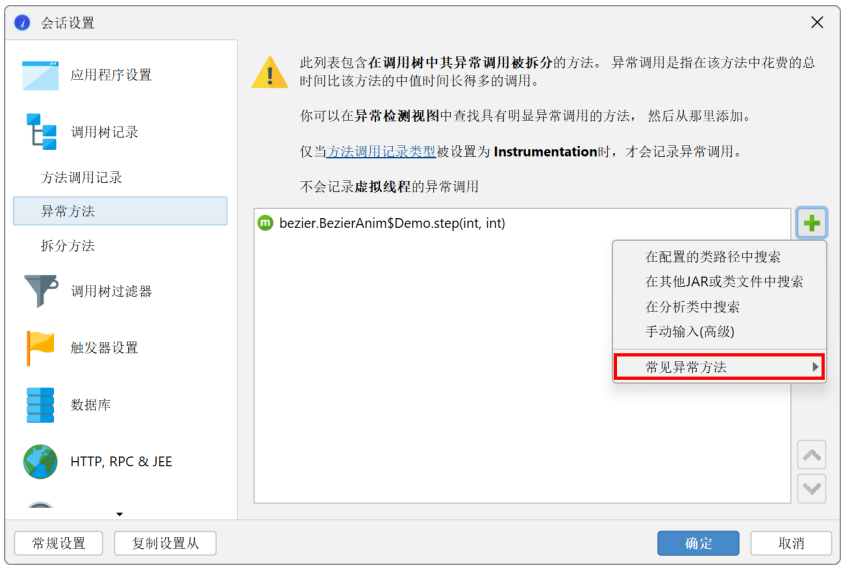


当您注册一个方法进行异常方法记录时，最慢的几个调用将单独保存在调用树中。其他调用将像往常一样合并到一个方法节点中。可以在配置文件设置中配置单独保留的调用次数。默认情况下，设置为5。

在区分慢方法调用时，必须使用某个线程状态进行时间测量。这不能是CPU视图中的线程状态选择，因为那只是一个显示选项，而不是记录选项。默认情况下，使用挂钟时间，但可以在配置文件设置中配置不同的线程状态。相同的线程状态也用于异常值检测视图。

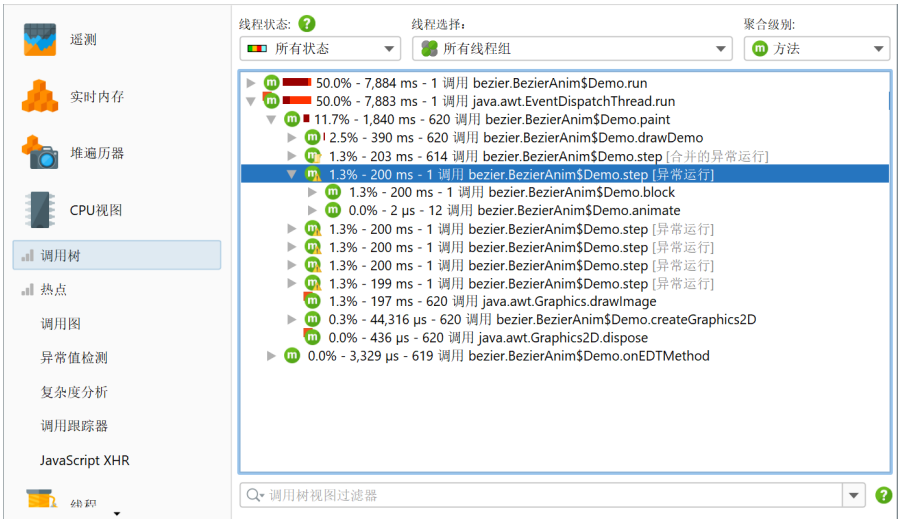


在会话设置中，您可以在没有调用树或异常值检测视图上下文的情况下删除异常方法或添加新方法。此外，异常方法配置提供了为知名系统添加异常方法定义的选项，例如AWT和JavaFX事件调度机制，其中异常长时间运行的事件是一个主要问题。



调用树中的异常方法

异常方法运行在调用树视图中以不同方式显示。



拆分的方法节点具有修改后的图标并显示附加文本：

- **[异常运行]**

  这样的节点包含一个异常慢的方法运行。根据定义，它的调用次数为1。如果许多其他方法运行在以后变得更慢，这个节点可能会消失并根据配置的单独记录方法运行的最大数量被添加到“合并异常运行”节点。

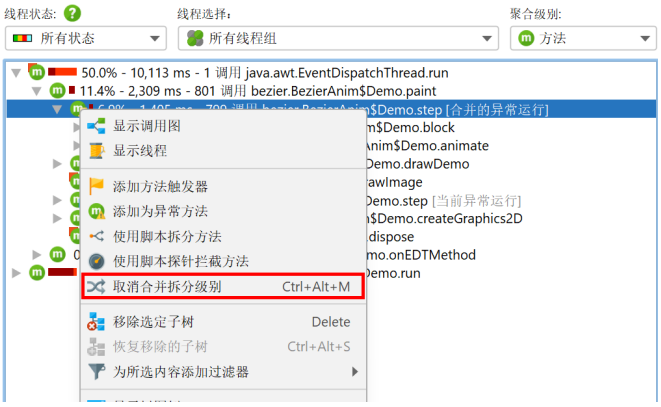
- **[合并异常运行]**

  不符合异常慢标准的方法调用将合并到此节点中。对于任何调用栈，每个异常方法只能有一个这样的节点。

- **[当前异常运行]**

如果在调用树视图传输到JProfiler GUI时调用正在进行中，尚不清楚调用是否异常慢。“当前异常运行”显示当前调用的单独维护的树。调用完成后，它将被维护为单独的“异常运行”节点或合并到“合并异常运行”节点。

与通过探针 [p. 96]和拆分方法 [p. 171]进行的调用树拆分一样，异常方法节点在上下文菜单中有一个合并拆分级别操作，允许您动态合并和取消合并所有调用。

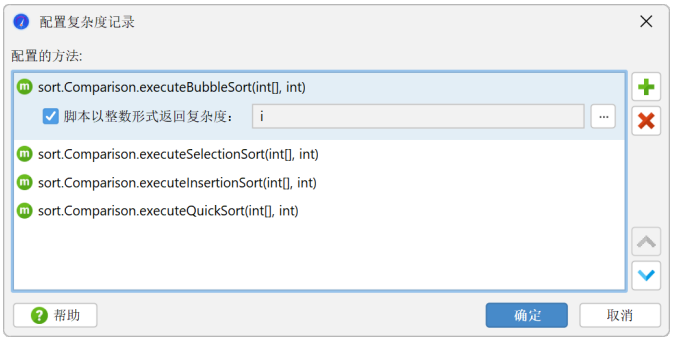


C.2 复杂性分析

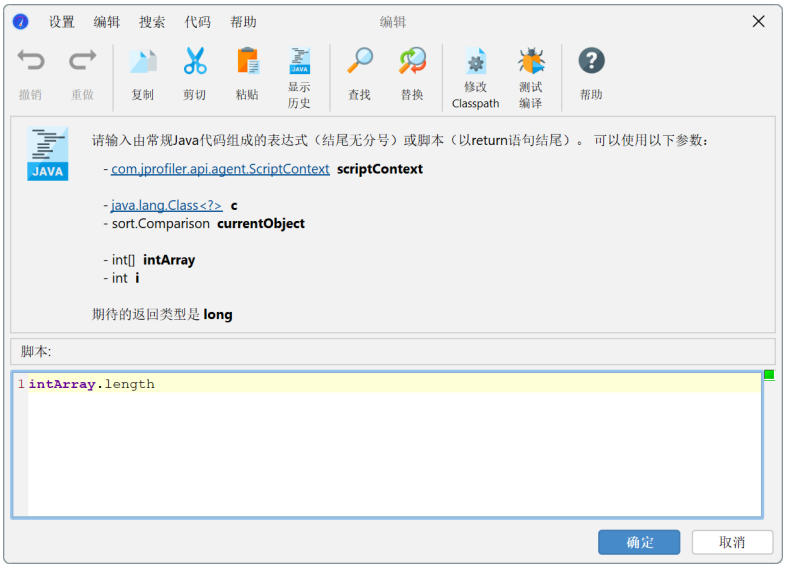
复杂性分析视图允许您根据方法参数调查所选方法的算法复杂性。

为了刷新对大O符号的详细了解，推荐阅读 算法复杂性入门 [1] 和 常见算法复杂性比较指南 [2]。

首先，您必须选择一个或多个需要监控的方法。



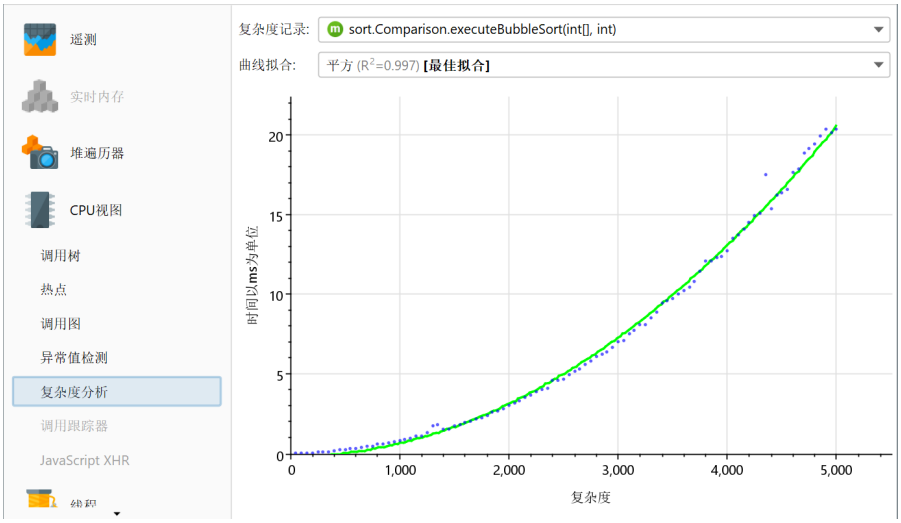
对于每个方法，您可以输入一个脚本，其返回值类型为 `long`，用于当前方法调用的复杂性。例如，如果某个方法参数的类型为`java.util.Collection`，并命名为`inputs`，则脚本可以是`inputs.size()`。



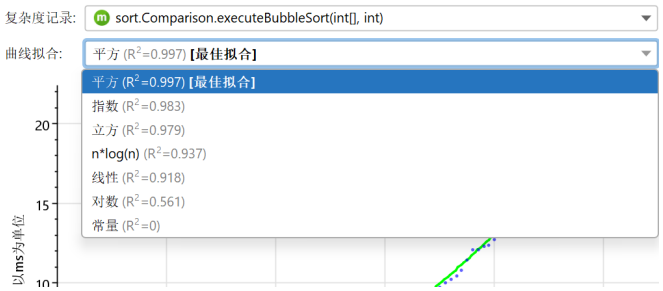
复杂性记录与CPU记录无关。您可以直接在复杂性分析视图中启动和停止复杂性记录，或者使用 记录配置文件或触发器动作 [p. 27]。 记录停止后，将显示一个图形，结果在x轴上绘制复杂性，在y轴上绘制执行时间。为了减少内存需求，JProfiler可以将不同的复杂性和执行时间合并到公共桶中。顶部的下拉菜单允许您在不同的配置方法之间切换。

图形是一个气泡图，其中每个数据点的大小与其中的测量次数成比例。如果所有测量值都是不同的，您将看到一个常规的散点图。在另一个极端情况下，如果所有方法调用具有相同的复杂性和执行时间，您将看到一个大的圆圈。

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



如果至少有3个数据点，将显示一个具有常见复杂性的曲线拟合。JProfiler尝试从几个常见复杂性中进行曲线拟合，并最初向您显示最佳拟合。曲线拟合的下拉菜单还允许您显示其他曲线拟合模型。嵌入在曲线拟合描述中的R^2值显示了拟合的好坏。下拉菜单中的模型按R^2降序排列，因此最佳模型始终是第一个项目。



请注意，R^2可以是负数，因为它只是一个符号，并不是真正的平方。负值表示拟合比常数线拟合更差。常数线拟合的R^2值始终为0，而完美拟合的值为1。

您可以通过在导出对话框中选择“属性”选项来导出当前显示的拟合参数。对于质量保证环境中的自动分析， 命令行导出 [p. 234]也支持属性格式。

C.3 调用跟踪器 (Call Tracer)

调用树中的方法调用记录累积具有相同调用栈的调用。保持精确的时间顺序信息通常不可行，因为内存需求巨大，并且记录的数据量使任何解释变得相当困难。

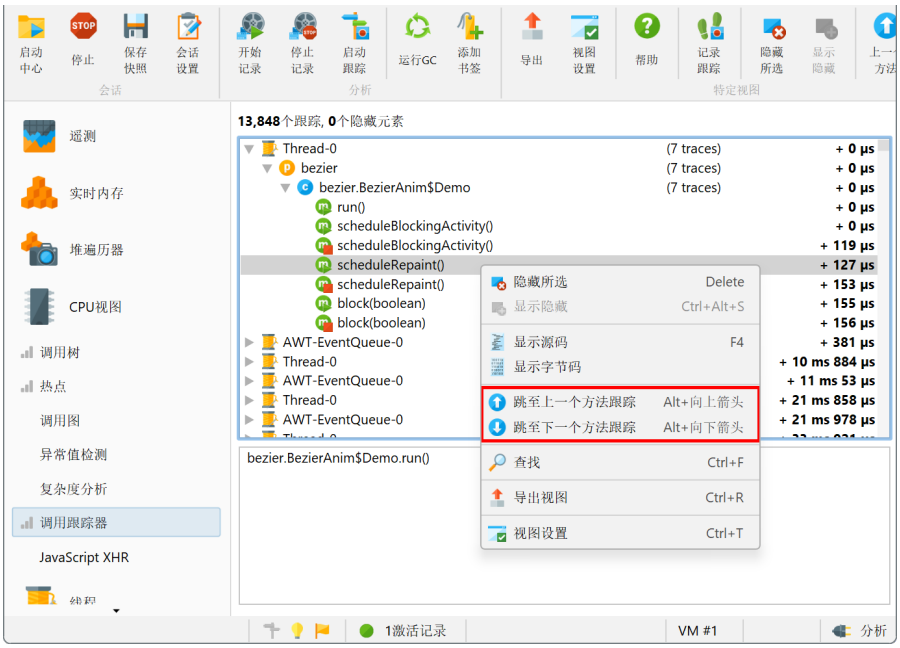
然而，在有限的情况下，跟踪调用并保持整个时间顺序是有意义的。例如，您可能希望分析多个协作线程的方法调用的精确交错。调试器无法逐步执行此类用例。或者，您希望分析一系列方法调用，但能够来回查看，而不仅仅像在调试器中那样只看一次。JProfiler通过调用跟踪器提供此功能。

调用跟踪器有一个单独的记录操作，可以在调用跟踪器视图中激活，使用触发器[p.27]或使用profiling API[p.119]。为了避免过度的内存消耗问题，对收集的调用跟踪的最大数量设置了上限。该上限可以在视图设置中配置。收集的跟踪率在很大程度上取决于您的过滤器设置。

只有在方法调用记录类型设置为插桩时，调用跟踪才有效。采样不跟踪单个方法调用，因此从技术上讲无法通过采样收集调用跟踪。对紧凑过滤类的调用在调用跟踪器中记录，就像在调用树中一样。如果您只想关注自己的类，可以在视图设置中排除这些调用。

跟踪的方法调用显示在一个具有三个级别的树中，这使得通过折叠它们更容易跳过相关调用。这三个组是 线程 (threads), 包 (packages) 和 类 (classes)。每次这些组中的任何一个的当前值更改时，都会创建一个新的分组节点。

在最低级别，有 方法入口 (method entry) 和 方法退出 (method exit)节点。在调用跟踪的表格下方，显示当前选定的方法跟踪的堆栈跟踪。如果从当前方法记录了对其他方法的调用跟踪，或者如果另一个线程中断了当前方法，则该方法的入口和退出节点将不相邻。您只能通过使用上一个方法 (Previous Method)和下一个方法 (Next Method)操作在方法级别上导航。



默认情况下，跟踪和所有分组节点上显示的时间指的是第一个跟踪，但可以更改为显示自上一个节点以来的相对时间。如果上一个节点是父节点，则差异为零。还可以选择显示相对于同类型上一个节点的相对时间。

即使使用适当的过滤器，也可以在很短的时间内收集大量的跟踪。为了消除不感兴趣的跟踪，调用跟踪器允许您快速修剪显示的数据。例如，某些线程可能不相关，或者某些包或类中的跟踪可能不感兴趣。此外，递归方法调用可能占用大量空间，您可能只想消除那些单个方法。

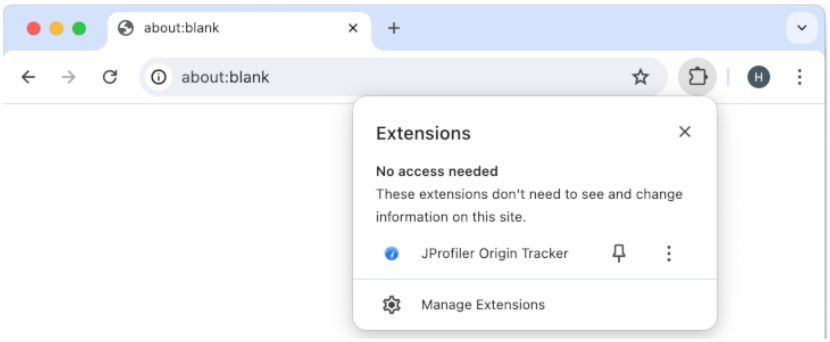
您可以通过选择节点并按 (delete)键来隐藏节点。所选节点的所有其他实例和所有关联的子节点也将被隐藏。在视图顶部，您可以看到所有记录的跟踪中仍显示了多少调用跟踪。要再次显示隐藏的节点，您可以单击显示隐藏 (Show Hidden)工具栏按钮。

C.4 JavaScript XHR 源跟踪

通过 JavaScript XHR 源跟踪，您可以在浏览器中为不同的堆栈跟踪拆分 servlet 调用，从而更好地将被分析的 JVM 中的活动与浏览器中的操作相关联。 在以下内容中，“XHR”指代 XMLHttpRequest 和 Fetch 机制。

浏览器插件

要使用此功能，您必须使用 Google Chrome [1] 作为浏览器并安装 JProfiler origin tracker 扩展 [2]。



Chrome 扩展会在工具栏中添加一个带有 JProfiler 图标的按钮以开始跟踪。当您开始跟踪时，扩展将拦截所有 XHR 调用并将其报告给本地运行的 JProfiler 实例。 只要跟踪尚未开始，JProfiler 将显示一页信息，告诉您如何设置 JavaScript XHR 源跟踪。

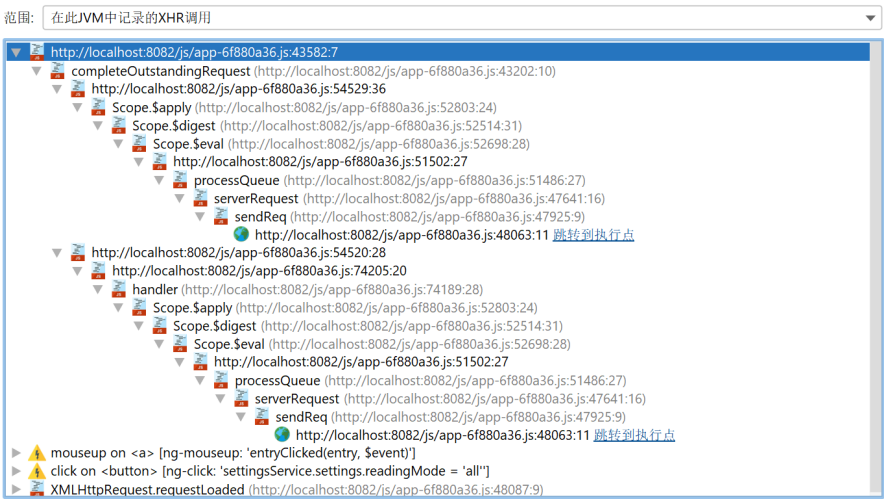
当跟踪被激活时，JProfiler扩展将要求您重新加载页面。这是为了添加检测。如果您选择不重新加载页面，事件检测可能无法正常工作。

跟踪状态在每个域名基础上是持久的。如果您在跟踪活动时重新启动浏览器并访问相同的URL，跟踪将自动启用，无需重新加载页面。

JavaScript XHR 树

如果 XHR 调用由 JProfiler 中活动分析会话分析的 JVM 处理，JavaScript XHR 视图将显示这些调用的累积调用树。 如果视图保持为空，您可以将视图顶部的“范围”切换为“所有 XHR 调用”以检查是否有任何 XHR 调用。

[1] http://www.google.com/chrome/
[2] https://chrome.google.com/webstore/detail/jprofiler-origin-tracker/mnicmpklpjkhohdbcdkflhochdfnmmbm



Javascript 调用栈节点包括源文件和行号的信息。 进行 XHR 调用的函数有一个 特殊图标和相邻的超链接， 如果 XHR 调用由被分析的 JVM 处理。超链接将带您到 Javascript 拆分节点在 调用树视图 [p. 51]，您可以在其中查看负责处理此类请求的服务器端调用树。

在树的顶部，您会发现 浏览器事件 节点，显示事件名称和元素名称以及有助于您确定事件来源的重要属性。并非所有请求都有相关事件。

扩展了解几个流行的 JavaScript 框架，并在事件的目标节点和事件监听器所在的节点之间遍历祖先层次结构，寻找适合显示和拆分调用树的属性。如果找不到特定于框架的属性，它会在 `id` 属性处停止。如果没有 ID，它会搜索“控制元素”，如 `a`、`button` 或 `input`。如果都失败，将显示注册事件监听器的元素。

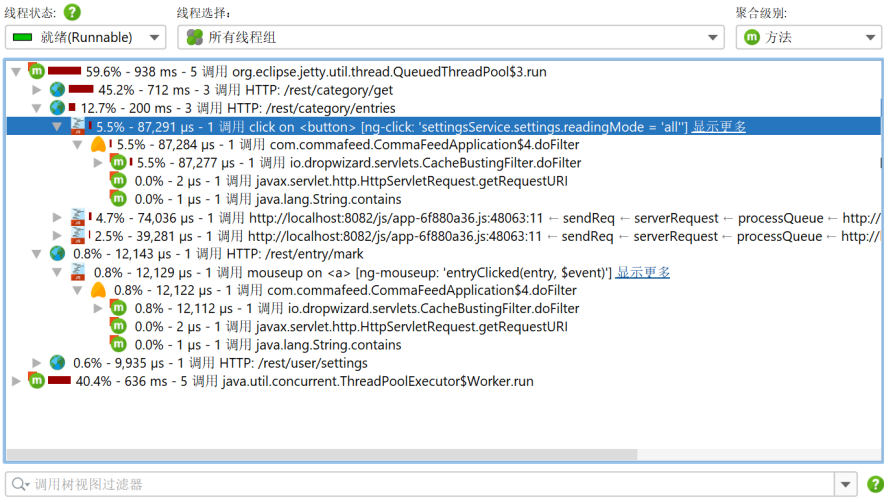
在某些情况下，自动检测有趣属性可能不合适，您可能更喜欢不同的调用树拆分。例如，一些框架分配自动 ID，但将所有元素与动作的语义描述分组在一起会更具可读性。要实现不同的调用树拆分，请添加 HTML 属性

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

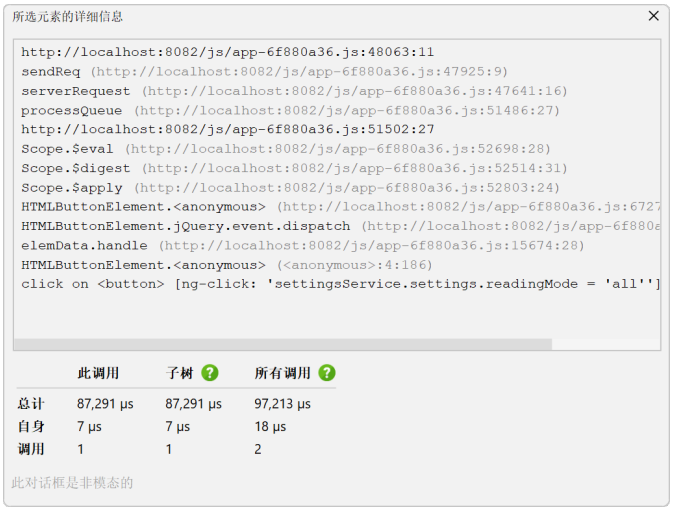
到目标元素或目标与事件监听器位置之间的元素。该属性中的文本将用于拆分，其他属性将被忽略。

调用树拆分

在调用树视图中，XHR 调用将为每个不同的浏览器事件和调用栈组合拆分调用树。 拆分节点显示有关浏览器事件的信息。 如果没有事件正在进行，例如在调用 `setTimeout()` 时，最后几个堆栈帧将内联显示。



这些节点上的“显示更多”超链接打开与 视图->显示节点详细信息 操作打开的相同详细对话框。 对于 JavaScript 拆分节点，详细对话框不显示节点的文本，而是显示整个浏览器调用栈。 要检查其他JavaScript拆分节点的调用栈，请保持非模态详细对话框打开并单击这些节点。 详细对话框将自动更新其内容。



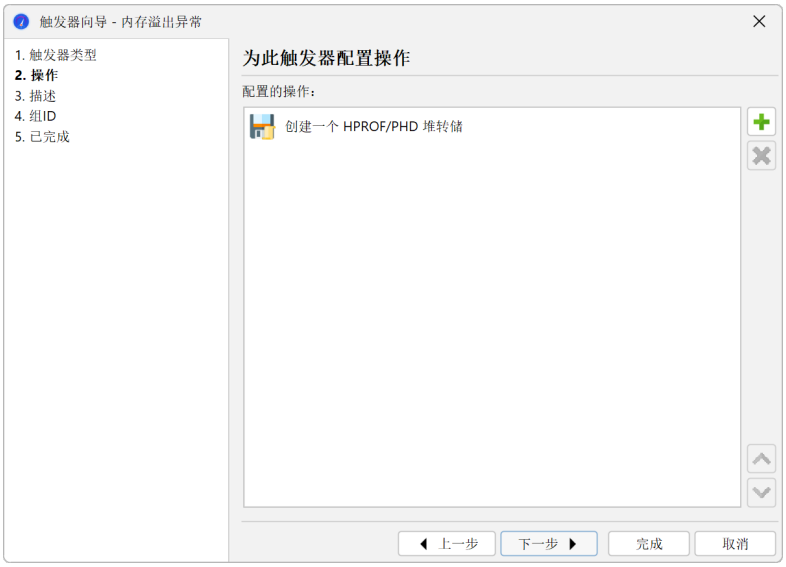
D 堆遍历器功能详解

D.1 HPROF 和 PHD 堆快照

HotSpot JVM 和 Android Runtime 都支持 HPROF 格式的堆快照，IBM J9 JVM 以 PHD 格式写入此类快照。PHD 文件不包含垃圾收集器根，因此 JProfiler 将类模拟为根。使用 PHD 文件查找类加载器内存泄漏可能会很困难。

本机堆快照可以在没有分析代理的情况下保存，并且比 JProfiler 堆快照的开销更低，因为它们是在没有通用 API 约束的情况下保存的。另一方面，本机堆快照支持的功能比 JProfiler 堆快照少。例如，分配记录信息不可用，因此您无法查看对象的分配位置。HPROF 和 PHD 快照可以在 JProfiler 中通过会话->打开快照打开，就像打开 JProfiler 快照一样。只有堆遍历器可用，所有其他部分将被灰显。

在实时会话中，您可以通过调用分析->保存 HPROF/PHD 堆快照创建并打开 HPROF/PHD 堆快照。对于离线分析[p. 119]，有一个“创建 HPROF 堆转储”触发器动作。通常与“内存不足异常”触发器一起使用，以便在抛出`OutOfMemoryError`时保存 HPROF 快照。



这对应于 VM 参数[(1)]

```
-XX:+HeapDumpOnOutOfMemoryError
```

由 HotSpot JVM 支持。

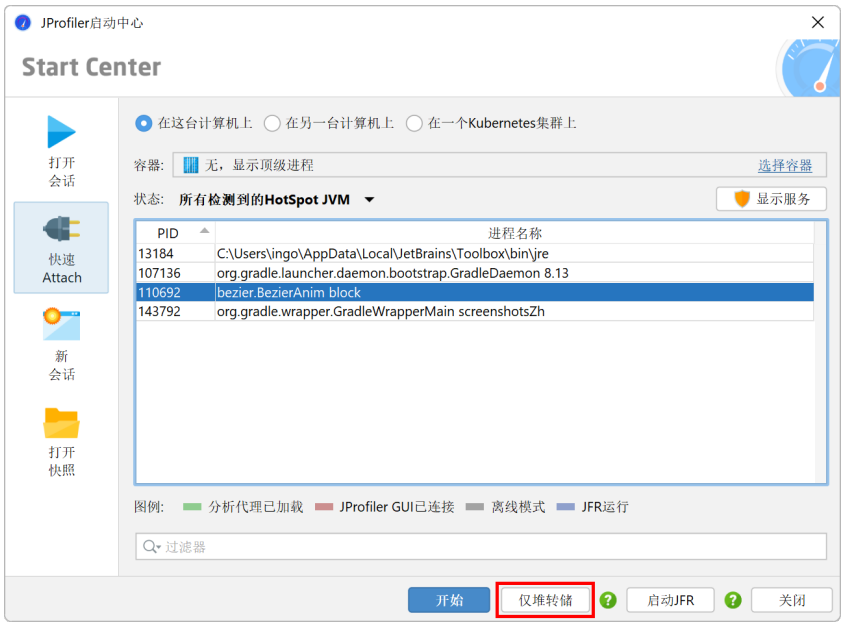
从正在运行的系统中提取 HPROF 堆转储的另一种方法是通过命令行工具`jmap`，它是 JRE 的一部分。其调用语法

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

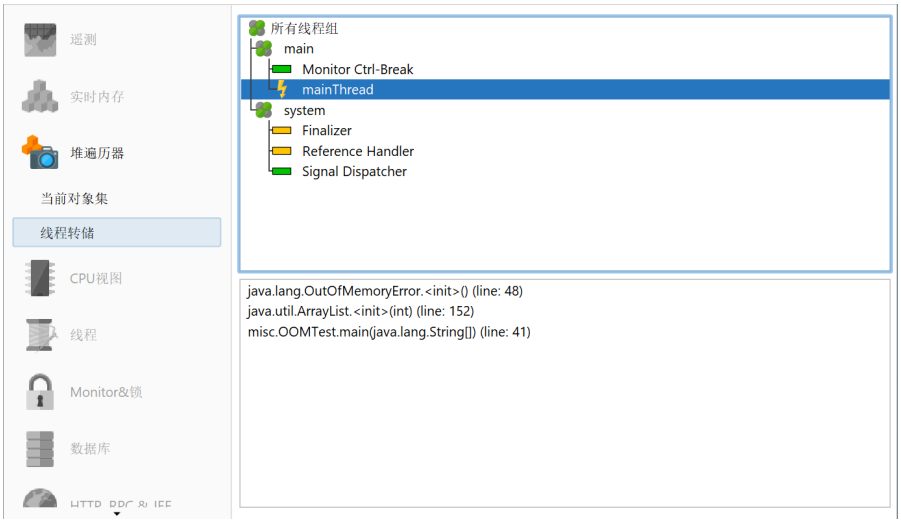
很难记住，并且需要您首先使用`jps`可执行文件来找出 PID。JProfiler 附带一个交互式命令行可执行文件`bin/jpdump`，它更加方便。它允许您选择一个进程，可以连接到在 Windows 上作为服务运行的进程，对混合 32 位/64 位 JVM 没有问题，并且自动编号 HPROF 快照文件。使用`-help`选项执行它以获取更多信息。

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

在不加载分析代理的情况下拍摄 HPROF 堆快照也在 JProfiler GUI 中得到支持。当附加到进程时，无论是本地还是远程，您始终可以选择仅拍摄 HPROF 堆快照。



HPROF 快照可以包含线程转储。当 HPROF 快照因`OutOfMemoryError`而保存时，线程转储可能能够传达应用程序在错误发生时处于活动状态的部分。触发错误的线程用一个特殊的图标标记。



D.2 最小化堆遍历器中的开销

对于小型堆，获取堆快照只需几秒钟，但对于非常大的堆，这可能是一个漫长的过程。物理内存不足会使计算变得更慢。例如，如果JVM有50 GB的堆，而您在本地机器上分析堆转储时只有5 GB的可用物理内存，JProfiler无法在内存中保存某些索引，处理时间会不成比例地增加。

因为JProfiler主要使用本机内存进行堆分析，所以不建议增加`-Xmx`值在`bin/jprofiler.vmoptions`文件中，除非您遇到了`OutOfMemoryError`并且JProfiler指示您进行这样的修改。如果本机内存可用，它将自动使用。分析完成并构建内部数据库后，本机内存将被释放。

对于实时快照，分析在获取堆转储后立即计算。当您保存快照时，分析将保存到快照文件旁边带有后缀`.analysis`的目录中。当您打开快照文件时，堆遍历器将非常快速地可用。如果您删除`.analysis`目录，打开快照时将重新进行计算，因此如果您将快照发送给其他人，您不必将分析目录一起发送。

如果您想在磁盘上节省内存或生成的`.analysis`目录不方便，您可以在常规设置中禁用它们的创建。

HPROF快照和使用离线分析 [p. 119]保存的JProfiler快照旁边没有`.analysis`目录，因为分析是由JProfiler UI执行的，而不是由分析代理执行的。如果您不想在打开此类快照时等待计算，可以使用`jpanalyze`命令行可执行文件来预分析 [p. 234]快照。

建议从可写目录中打开快照。当您打开没有分析的快照且其目录不可写时，将使用临时位置进行分析。然后每次打开快照时都必须重复计算。

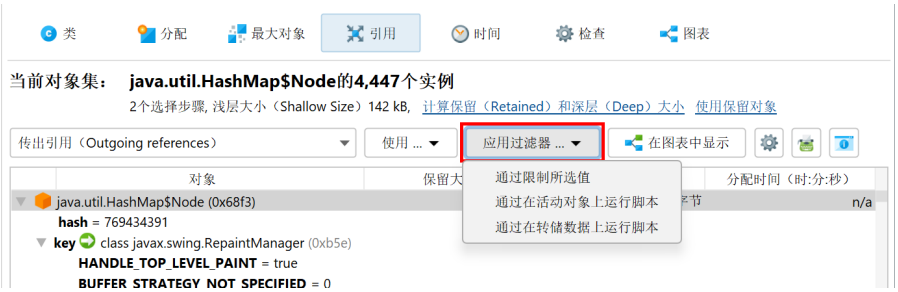
分析的一个重要部分是保留大小的计算。如果处理时间过长且您不需要保留大小，您可以在堆遍历器选项对话框的开销选项中禁用它们的计算。在这种情况下，“最大对象”视图也将不可用。不记录原始数据会使堆快照更小，但您将无法在引用视图中看到它们。如果您在文件选择对话框中选择自定义分析，打开快照时会显示相同的选项。


堆快照选项
选择记录的对象
最初，堆遍历器将只显示在动态内存视图部分中记录的对象。
在堆快照中执行完整的GC
保留由类型为
软
的引用持有的对象
隐藏开销选项
计算保留大小(retained sizes)
记录原始数据
确定
取消

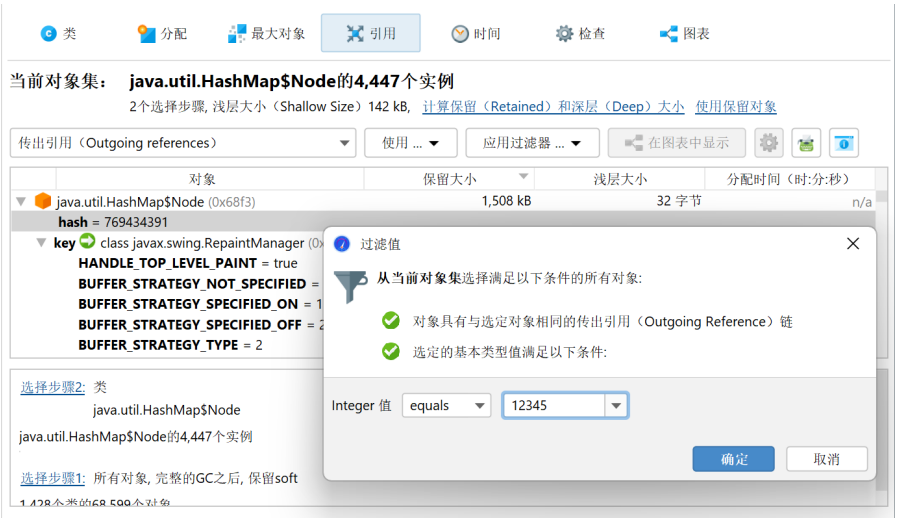
D.3 过滤器和实时交互

在堆遍历器中寻找感兴趣的对象时，您通常会到达一个包含太多相同类实例的对象集。为了根据您的特定关注点进一步修剪对象集，选择标准可以涉及它们的属性或引用。例如，您可能对包含特定属性的HTTP会话对象感兴趣。在堆遍历器的合并传出引用视图中， 您可以执行涉及整个对象集引用链的选择步骤。

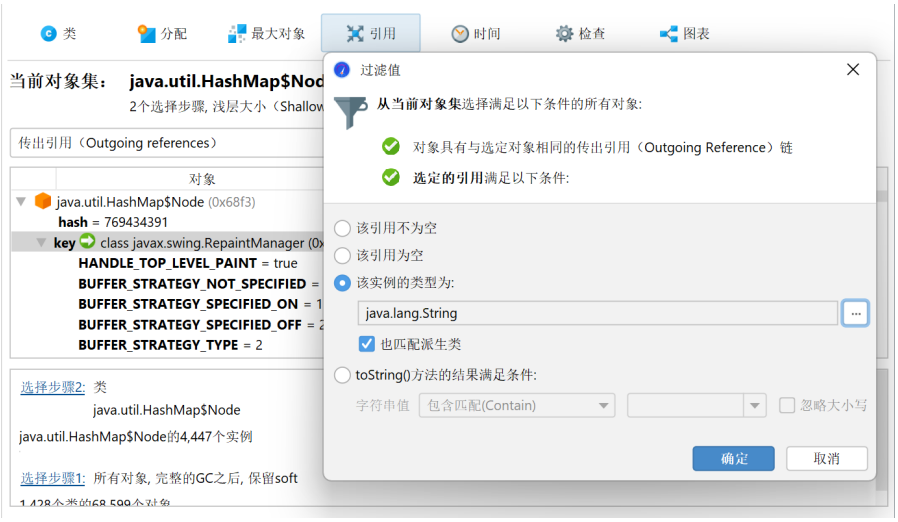
然而，在您看到单个对象的传出引用视图中，提供了更强大的功能来进行约束引用和原始字段的选择步骤。



当您在传出引用视图中选择顶级对象、原始值或引用时，应用过滤器->通过限制所选值操作将被启用。根据选择，过滤器值对话框提供不同的选项。无论您配置了什么选项，您总是隐式地添加约束，即新对象集中的对象必须具有类似于所选对象的传出引用链。过滤器始终通过将当前对象集限制为可能更小的集合来作用于顶级对象。

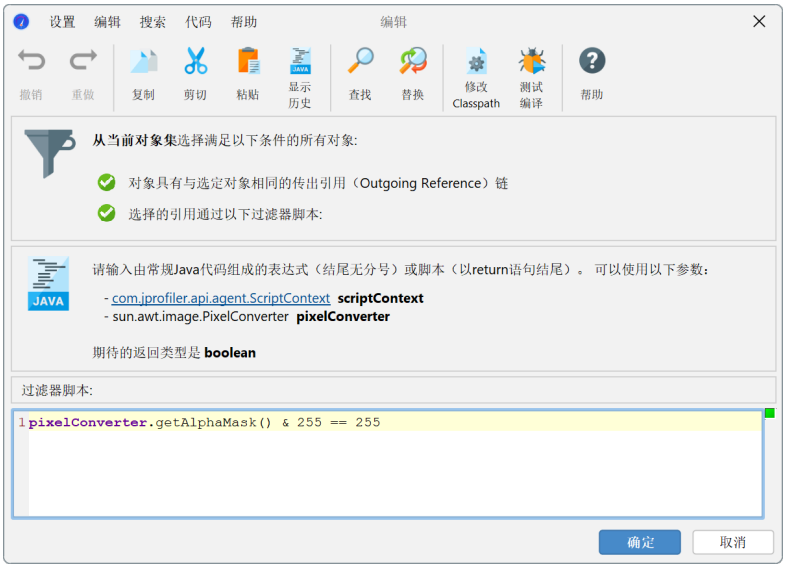


在HPROF和JProfiler堆快照中，约束原始值均有效。对于引用类型，您可以要求JProfiler过滤非空值、空值和选定类的值。通过`toString()`方法的结果进行过滤仅在实时会话中可用，`java.lang.String`和`java.lang.Class`对象除外， JProfiler可以自行解决。



最强大的过滤器类型是使用代码片段的过滤器。有两种根本不同的方法来过滤对象：

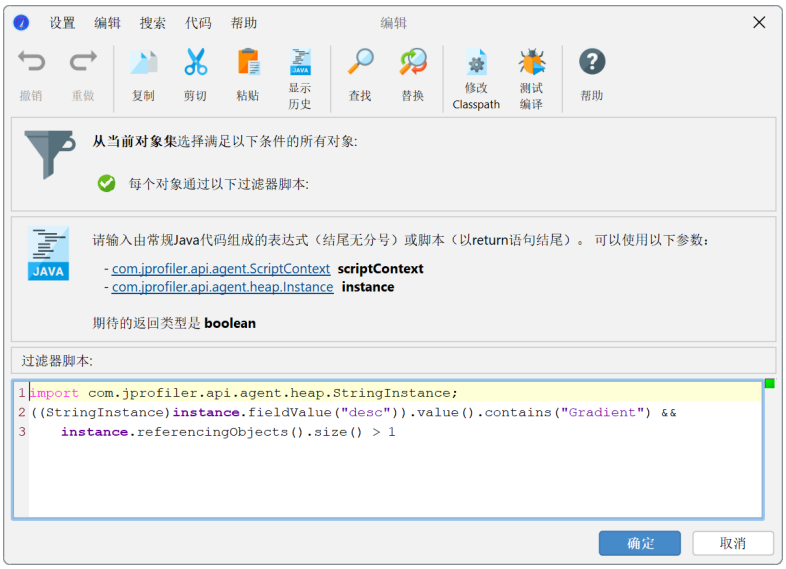
在实时会话中，JProfiler可以在被分析的JVM中运行过滤脚本并将实际实例传递给您的脚本。在由通过在实时对象上运行脚本应用过滤器显示的脚本编辑器中，您可以编写一个直接访问属性的表达式或脚本，其布尔返回值决定实例是否应保留在当前对象集中。



当然，此功能只能在实时会话中工作，因为JProfiler需要访问实时对象。另一个需要考虑的因素是，在堆快照拍摄后，对象可能已被垃圾回收。在这种情况下，当执行代码片段过滤器时，该对象将不会包含在新对象集中。

第二个选项也适用于快照，包括HPROF和PDH快照，是通过在转储数据上运行脚本应用过滤器操作。每个实例将作为 `com.jprofiler.api.agent.heap.HeapObject`的实例传递给您的脚本。如果适用，您可以将参数向下转换为多个子接口。请参阅Javadoc以获取更多信息。例如，如果对象是对象实例并提供对字段值的访问，则`com.jprofiler.api.agent.heap.Instance`接口可用。如果脚本在顶级对象上运行并且当前对象集中的所有对象都是同一类型，则脚本参数将自动具有合适的子类型。

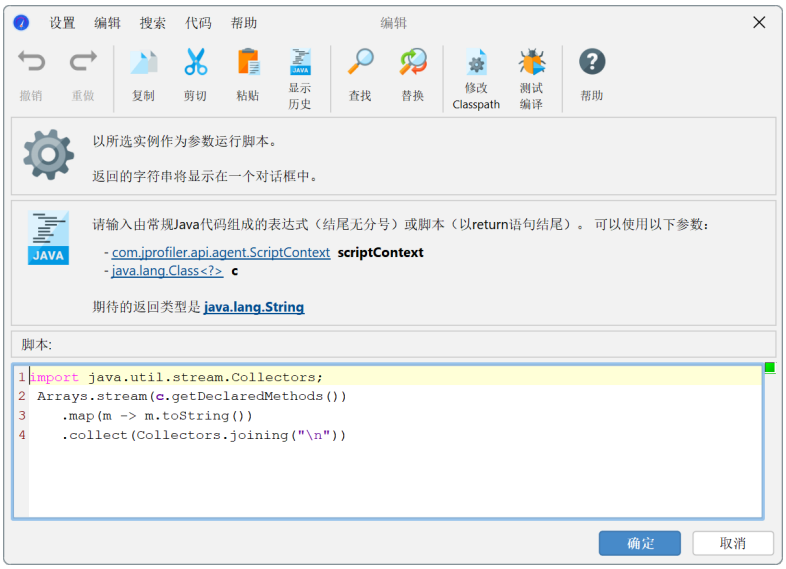
在这些过滤器脚本中，您还可以通过`HeapObject`参数的方法访问所有传入和传出的引用。



请注意，PHD快照不包含字段信息，因此对于此类快照，所有实例都作为`com.jprofiler.api.agent.heap.HeapObject`或 `com.jprofiler.api.agent.heap.ClassObject`传递，字段值只能通过`referencedObjects()`方法访问。

除了过滤器之外，传出引用视图中还有两个其他功能用于与单个对象交互：显示toString()值操作调用当前在视图中可见的所有对象的`toString()`方法，并直接在引用节点中显示它们。节点可能会变得很长，文本可能会被截断。使用上下文菜单中的显示节点详细信息操作可以帮助您查看整个文本。

获取对象信息的一种更通用的方法是运行返回字符串的任意脚本。运行脚本操作位于显示toString()值操作旁边，允许您在选择顶级对象或引用时执行此操作。脚本执行的结果显示在一个单独的对话框中。

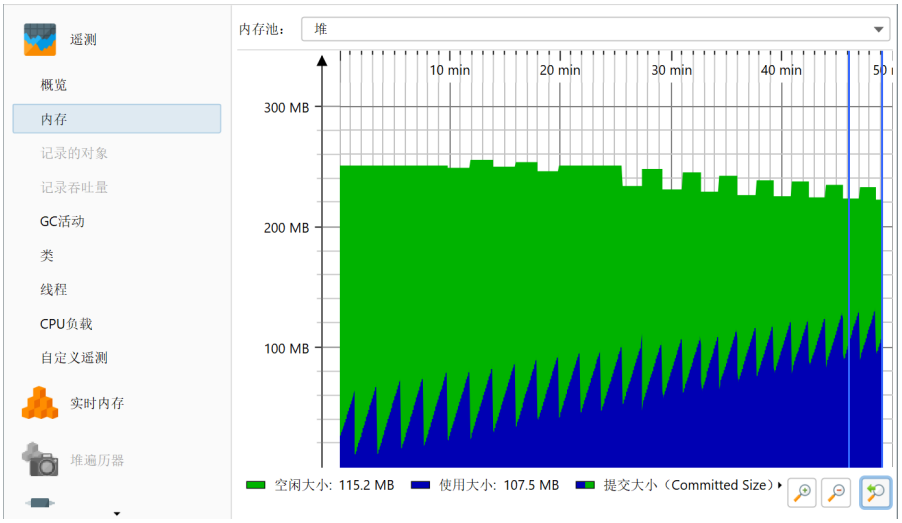

设置 编辑 搜索 代码 帮助
编辑
撤销
重做
复制
剪切
粘贴
显示历史
查找
替换
修改Classpath
测试编译
帮助
以所选实例作为参数运行脚本。
返回的字符串将显示在一个对话框中。
JAVA
请输入由常规Java代码组成的表达式（结尾无分号）或脚本（以return语句结尾）。 可以使用以下参数：
- com.jprofiler.api.agent.ScriptContext scriptContext
- java.lang.Class<?> c
期待的返回类型是 java.lang.String
脚本:
1 import java.util.stream.Collectors;
2 Arrays.stream(c.getDeclaredMethods())
3 .map(m -> m.toString())
4 .collect(Collectors.joining("\n"))
确定
取消

D.4 查找内存泄漏

区分常规内存使用和内存泄漏通常并不简单。然而，过多的内存使用和内存泄漏具有相同的症状，因此可以用相同的方式进行分析。分析分为两个步骤：定位可疑对象并找出这些对象为何仍在堆中。

查找新对象

当具有内存泄漏的应用程序运行时，它会随着时间的推移消耗越来越多的内存。检测内存使用增长的最佳方法是使用VM遥测和"所有对象"和"记录对象"视图中的 差异功能 [p. 68]。通过这些视图，您可以确定是否存在问题以及问题的严重程度。有时，调用直方图表中的差异列已经可以让您了解问题所在。



对内存泄漏的任何深入分析都需要在堆漫游器中进行。要详细调查特定用例周围的内存泄漏，"标记堆"功能[p. 75]最为合适。它允许您识别自特定先前时间点以来仍留在堆中的新对象。对于这些对象，您必须检查它们是否仍然合法地留在堆中，或者是否有错误的引用使它们保持活动状态，即使对象不再有任何用途。

隔离您感兴趣的对象集的另一种方法是通过分配记录。当拍摄堆快照时，您可以选择显示所有记录的对象。然而，您可能不希望将分配记录仅限于特定用例。此外，分配记录的开销很高，因此标记堆操作的影响相对较小。最后，堆漫游器允许您在任何选择步骤中使用使用新和使用旧超链接选择旧对象和新对象。

分析最大对象

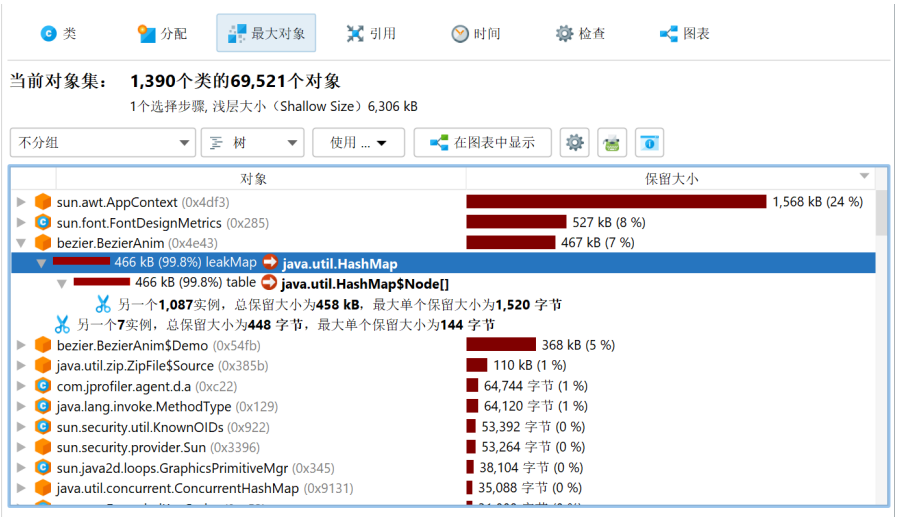
如果内存泄漏填满了可用堆，它将使被分析应用程序中的其他类型内存使用显得微不足道。在这种情况下，您无需检查新对象，只需分析哪些对象最为重要。

内存泄漏可能具有非常缓慢的速率，并且可能在很长时间内不会变得显著。在内存泄漏变得可见之前进行分析可能不切实际。通过JVM内置的功能，在抛出`OutOfMemoryError`时自动 保存HPROF快照 [p. 191]，您可以获得一个快照，其中内存泄漏比常规内存消耗更为重要。事实上，始终添加

```
-XX:+HeapDumpOnOutOfMemoryError
```

到VM参数或生产系统中是个好主意，这样您就有办法分析在开发环境中难以重现的内存泄漏。

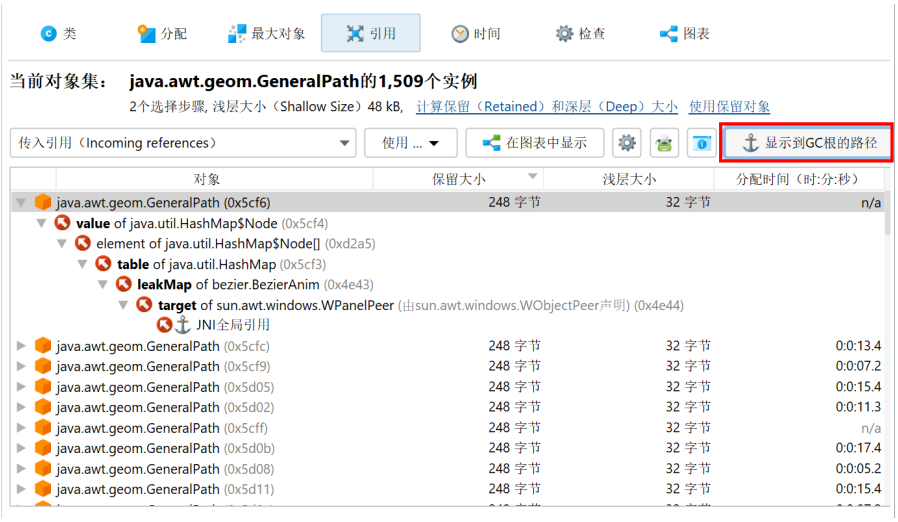
如果内存泄漏占主导地位，堆漫游器的"最大对象"视图中的顶级对象将包含错误保留的内存。虽然最大对象本身可能是合法对象，但打开它们的支配树将导致泄漏对象。在简单情况下，可能有一个对象将包含大部分堆。例如，如果使用映射来缓存对象并且该缓存从未清除，那么该映射将在最大对象的支配树中显示。



查找从垃圾收集器根到强引用链

只有当对象被强引用时，它才可能成为问题。"强引用"意味着至少有一条从垃圾收集器根到对象的引用链。"垃圾收集器"根（简称GC根）是JVM中垃圾收集器知道的特殊引用。

要查找从GC根到引用链，您可以使用"传入引用"视图或堆漫游器图中的显示到GC根的路径操作。这样的引用链在实践中可能非常长，因此通常可以在"传入引用"视图中更容易地解释。引用从底部指向顶层的对象。只有搜索结果的引用链被展开，同一层次上的其他引用在节点关闭并重新打开或在上下文菜单中调用显示所有传入引用操作之前不可见。



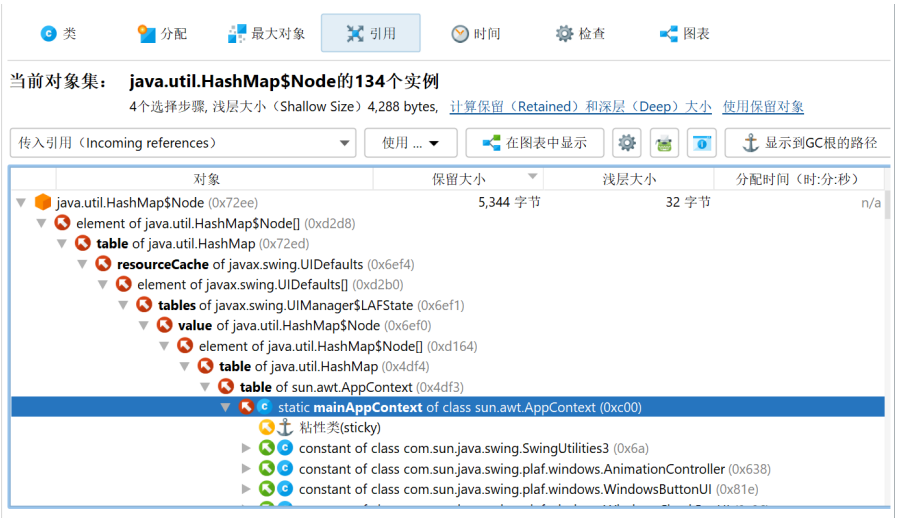
要获取GC根类型和引用节点中使用的其他术语的解释，请使用树图例。

当您在树中选择节点时，非模态树图例会突出显示所选节点中使用的所有图标和术语。单击对话框中的一行将在底部显示解释。

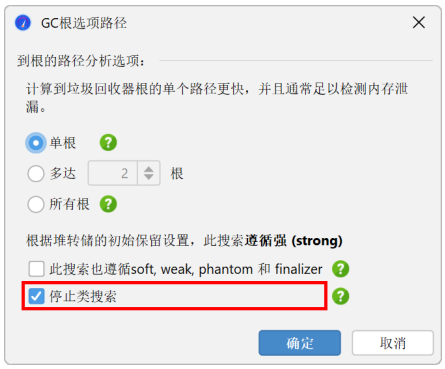
垃圾收集器根的重要类型是来自堆栈的引用，通过JNI创建的本机代码引用以及当前正在使用的资源，如活动线程和对象监视器。此外，JVM还添加了一些"粘性"引用以保持重要系统的稳定。

类和类加载器具有特殊的循环引用方案。当

- 该类加载器加载的类没有任何活动实例
- 类加载器本身除了其类之外没有被引用
- 除了在类加载器的上下文中，没有引用任何`java.lang.Class`对象

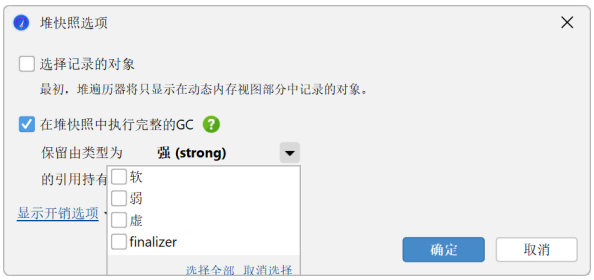


在大多数情况下，类是您感兴趣的GC根路径上的最后一步。类本身不是GC根。然而，在不使用自定义类加载器的所有情况下，将它们视为GC根是合适的。这是JProfiler在搜索垃圾收集器根时的默认模式，但您可以在路径到根选项对话框中更改它。



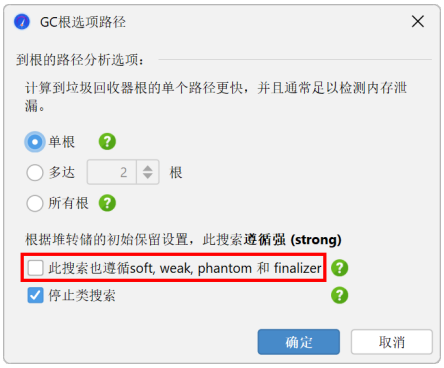
如果您在解释到GC根的最短路径时遇到问题，可以搜索其他路径。一般来说，不建议搜索到GC根的所有路径，因为它可能会产生大量路径。

与实时内存视图不同，堆漫游器从不显示未引用的对象。然而，堆漫游器可能不仅显示强引用的对象。默认情况下，堆漫游器还保留仅由软引用引用的对象，但消除仅由弱引用、幻影引用或终结器引用的对象。因为软引用在堆耗尽之前不会被垃圾收集，所以它们被包括在内，因为否则您可能无法解释大堆使用。在拍摄堆快照时显示的选项对话框中，您可以调整此行为。



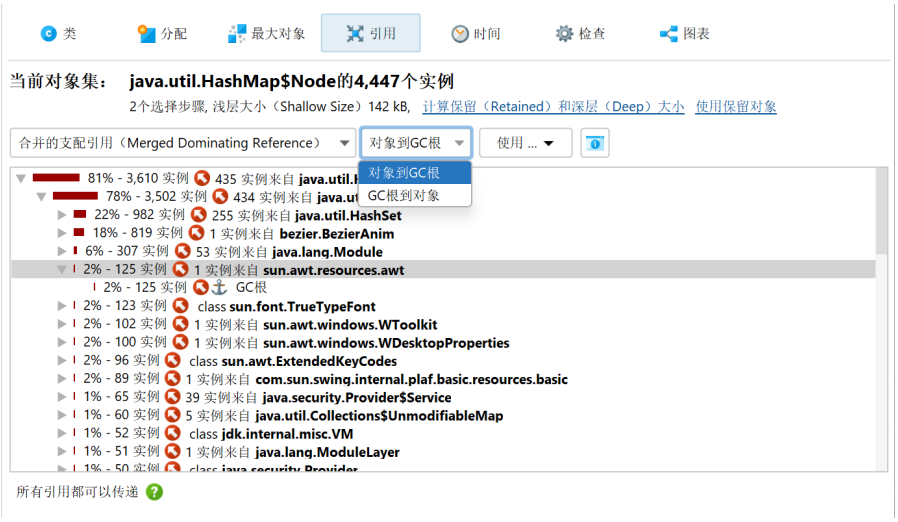
在堆漫游器中拥有弱引用的对象可能对调试目的很有趣。如果您想稍后删除弱引用的对象，可以使用"移除由弱引用保留的对象"检查。

在搜索到GC根的路径时，堆漫游器选项对话框中选择的保留对象的引用类型会被考虑在内。这样，路径到GC根搜索始终可以解释为什么对象在堆漫游器中被保留。在路径到GC根搜索的选项对话框中，您可以将可接受的引用类型扩展到所有弱引用。



消除整个对象集

到目前为止，我们只看了单个对象。通常，您会有许多相同类型的对象，它们是内存泄漏的一部分。在许多情况下，单个对象的分析也适用于当前对象集中的其他对象。对于更一般的情况，其中感兴趣的对象以不同方式被引用，"合并支配引用"视图将帮助您找出哪些引用负责将当前对象集保留在堆中。



支配引用树中的每个节点都会告诉您，如果消除该引用，当前对象集中有多少对象将有资格进行垃圾收集。由多个垃圾收集器根引用的对象可能没有任何支配的传入引用，因此该视图可能只对一部分对象有帮助，或者甚至可能为空。在这种情况下，您必须使用合并传入引用视图并逐个消除垃圾收集器根。

E JDK Flight Recorder (JFR)

E.1 支持JDK Flight Recorder (JFR)

JDK Flight Recorder (JFR)[1] 是一个结构化的日志记录工具，记录广泛的系统级事件。类似于飞机的黑匣子，持续记录飞行数据以用于事件调查，JFR持续记录JVM中的事件流以用于诊断问题。这种方法的优势在于，它可以按时间顺序捕获系统在事件发生前的详细信息。JFR设计为对性能影响最小，并且可以安全地在生产环境中长时间运行。

从Java 17开始，JFR也是JProfiler的数据源之一。除了使用JVM分析接口的本机代理外，JVM中还有一些在分析上下文中感兴趣的高级系统。一个是MBean系统，它为JProfiler中的一些遥测提供数据，另一个是用于垃圾收集器探针 [p. 109]的JFR。为此，您无需与JFR交互，JProfiler会透明地处理JFR事件流。

JProfiler中的JFR集成

JProfiler完全集成了JFR记录 [p. 206]，因此您可以轻松捕获本地机器或未配置JFR记录的远程机器上运行的JVM的数据。

当您在JProfiler UI中打开JFR快照时，可用的视图和部分与常规分析会话不同。UI的核心是 事件浏览器 [p. 210]。所有其他可用于JFR视图的视图都在 单独的章节中解释 [p. 216]。

当您处理事件类型、设置过滤器和查看分析时，JProfiler有时需要重新扫描JFR快照文件。JFR快照文件可能非常大，无法将所有数据保存在内存中或提前计算所有分析。因此，不建议从网络驱动器打开JFR快照。

在打开非常大的JFR快照时，您可以通过在文件选择器中单击“自定义分析”复选框并排除不需要的事件类别来加快快照处理速度并减少内存使用。可用的事件类别涵盖单个探针和视图部分。CPU视图、内存视图和遥测视图的事件类型是必需的，必须加载。

例如，如果您只对CPU数据感兴趣，可以排除所有探针和事件浏览器。JProfiler旨在成为最快的JFR查看器，并快速打开典型的JFR快照，但JFR记录可能是无限的，您可能会遇到大小为几十GB的快照，此时打开速度可能会成为问题。

JFR快照中的堆栈跟踪

JFR的一个重要功能是能够以高效的方式记录某些事件类型的整个堆栈跟踪。对于此类事件类型，您可以在JFR设置中切换堆栈跟踪记录。许多JVM应用程序事件类型，尤其是与线程相关的事件类型，默认启用堆栈跟踪记录。

JFR仅收集固定深度的堆栈跟踪，因此长堆栈跟踪会被截断。截断的跟踪不适合构建可理解的调用树，因此这些跟踪显示在特别标记的节点下。 使用

```
-XX:FlightRecorderOptions=stackdepth=<nnnn>
```

VM参数，您可以增加JFR中收集的跟踪大小，并消除应用程序中截断的跟踪。

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

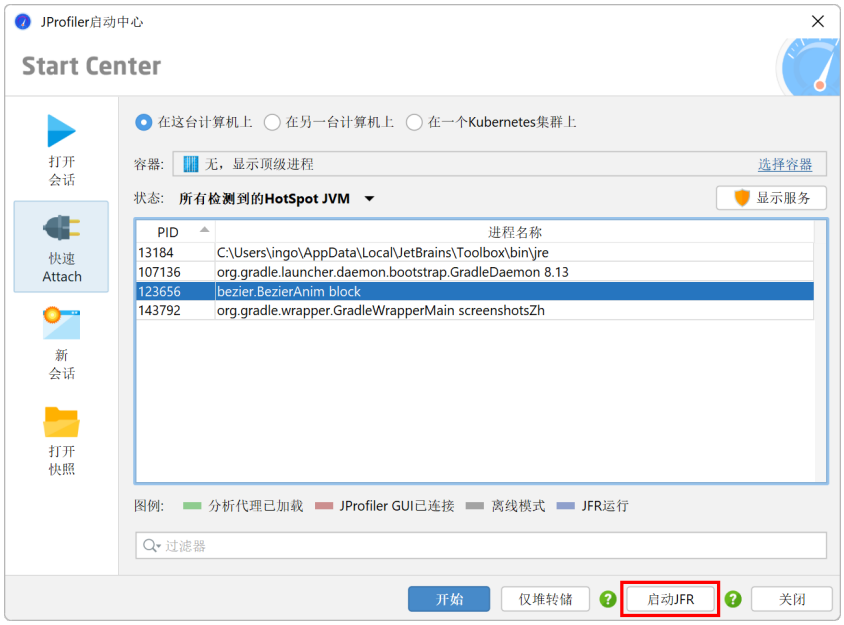
E.2 使用JProfiler记录JFR快照

由于在生产JVM中运行JFR的好处是开销最小且无需启用分析接口，JProfiler在UI中直接支持JFR记录。 虽然您可以通过编程方式启动JFR或在命令行中添加`-XX:StartFlightRecording` VM参数，但JProfiler可以帮助您启动和停止已运行JVM的记录。

当您使用JProfiler附加到JVM时，您可以选择启动和停止JFR记录，而不是加载本机分析代理。 借助JProfiler广泛的远程连接功能，您可以在不修改容器的情况下，例如在Docker或Kubernetes容器中运行的JVM中启动JFR记录。

启动和停止JFR记录

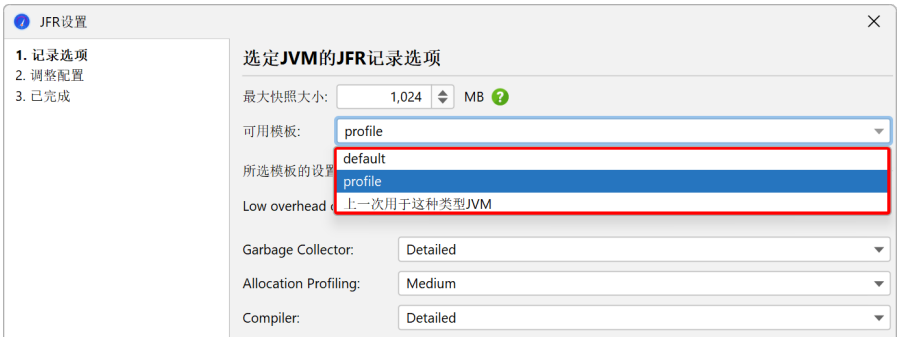
在启动中心的"Quick attach"选项卡上，选择一个JVM并点击对话框底部的Start JFR按钮。 屏幕截图中显示的是本地运行的JVM，但当您附加到远程JVM时，同样的按钮也可用。



在JFR设置向导中，您可以选择从所选进程使用的JRE的`lib/jfr`目录中传输的**事件设置模板**之一。默认情况下，有两个这样的模板，"default"和"profile"，其中"profile"记录更多数据并增加更多开销。 如果您在该目录中创建其他文件，您将能够在向导中选择相应的模板。

这些模板文件包含可用事件以及重要**高级设置**的配置指令。每个高级设置可以与多个不同的事件关联。此UI是根据模板文件的内容动态生成的。在不同的配置文件之间切换将向您显示不同的默认值。还有许多事件类型未包含在此UI中，仅在下一步中可配置。

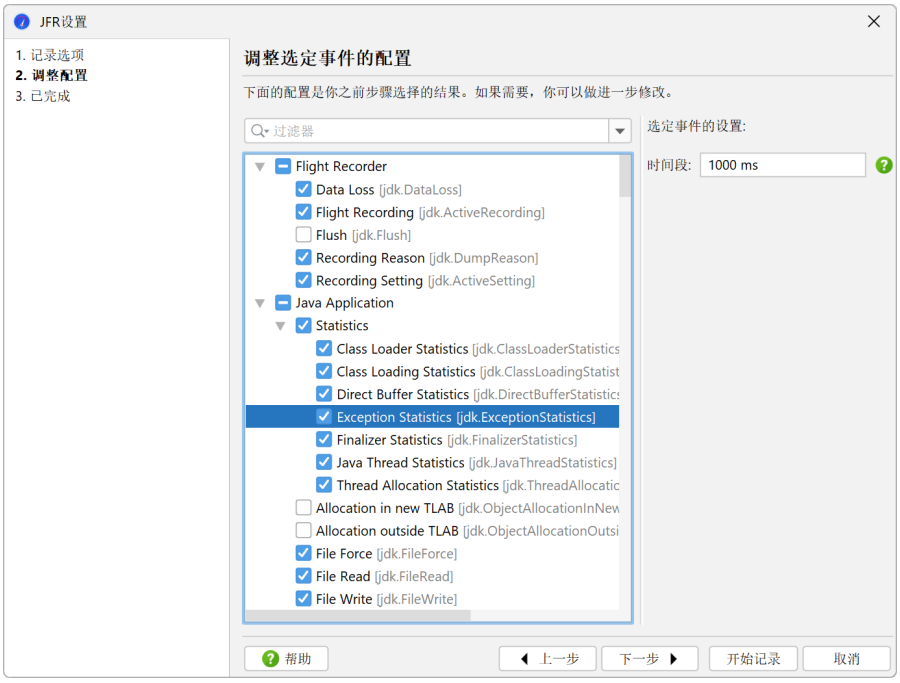
如果您已经为具有相同事件类型集的JVM启动了JFR记录，JProfiler将为您提供使用**上次设置**的选项。



如果您选择该选项，高级记录设置将不可用，您可以继续下一步查看整个配置并进行进一步更改。

向导此步骤中的另一个重要设置是**最大快照大小**。由于JFR记录的性质，快照的大小可能会迅速增加，可能会填满您的整个硬盘。 为了避免这种情况，最大快照大小限制可以防止过度的存储利用。当达到最大大小时，旧事件将被丢弃，而新事件将继续被记录。 此过程是JFR的自动机制。

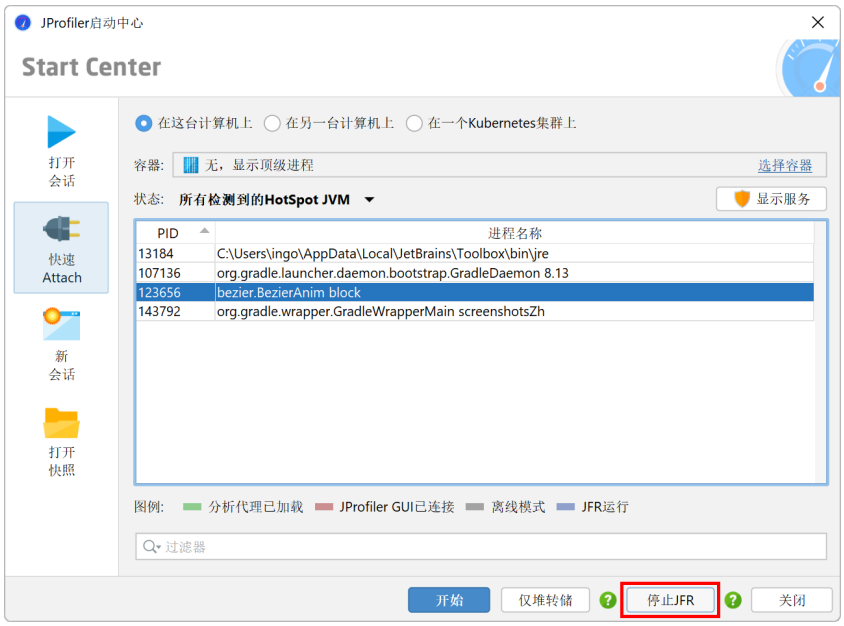
在向导的下一步中，您可以看到所有事件类型的分类树，并在右侧进行进一步的配置。



事件可能有一个**周期**、一个**阈值**和一个标志，用于指示是否为每个事件记录**堆栈跟踪**。周期和阈值都是带有时间单位的设置，您可以按down键获取可用单位的自动完成弹出窗口。 周期还支持特殊值"everyChunk"、"beginChunk"和"endChunk"，这些值也可以从自动完成弹出窗口中获得。"chunk"指的是JFR记录的一部分，其中包含一组连续的事件数据和元数据，并作为记录中的基本存储和数据传输单元。

树中选择的事件越多，记录的数据就越多。有些事件类型会生成大量数据，而有些则只生成少量事件。

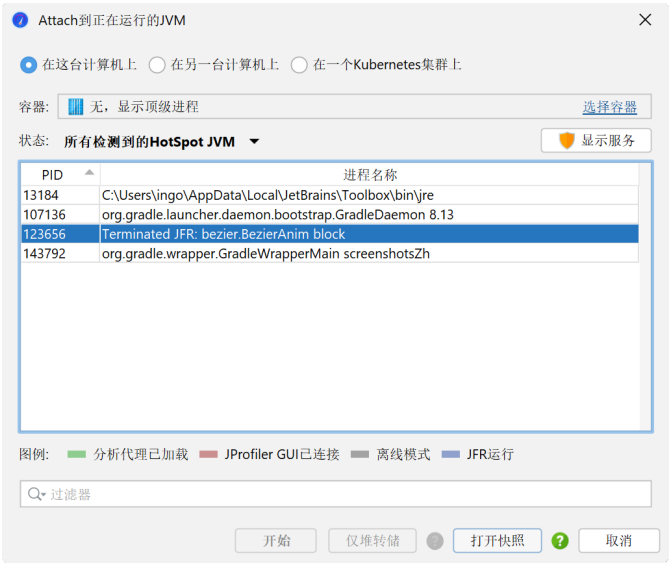
与完整分析模式或"Heap dump only"模式不同，在这些模式下，您可以立即在UI中看到一些数据，启动JFR快照仅会在未选择时修改JVM在表中的背景颜色， 以便您可以看到JProfiler已开始记录。当选择JVM时，底部的JFR按钮文本现在会显示您将停止记录。



当您停止由JProfiler启动的JFR记录时，JFR快照将被传输并在JProfiler中打开。**快照是临时的**，关闭窗口时将被删除。 要将快照保存到永久位置，请使用工具栏中的"Save snapshot"操作。

具有JFR记录的终止JVM

JFR的一个提到的用途是调查崩溃前的时刻。在这种情况下，JVM将不再在JVM表中可用以停止JFR记录并打开JFR快照。 如果在JProfiler中启动了JFR记录，并且在您停止记录之前JVM终止，则将在JVM表中添加一个以"Terminated JFR:"为前缀的特殊条目。 通过双击该条目或使用"JFR"按钮，您可以打开JFR快照。



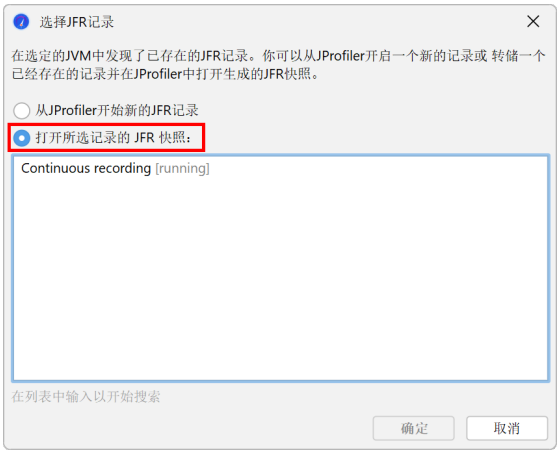
一旦您打开这样的条目，它将从列表中删除。与手动停止的记录一样，打开的JFR快照将是临时的，如果您想保留以供以后分析，您必须保存它。

显示外部启动的JFR记录

在上面的示例中，JFR记录是在JProfiler中启动和停止的。外部启动的JFR记录也可以显示。 可以通过类似的VM参数轻松启动连续的JFR记录

```
"-XX:StartFlightRecording=maxsize=500m=filename=$TEMP/myapp.jfr,name=Continuous
recording"
```

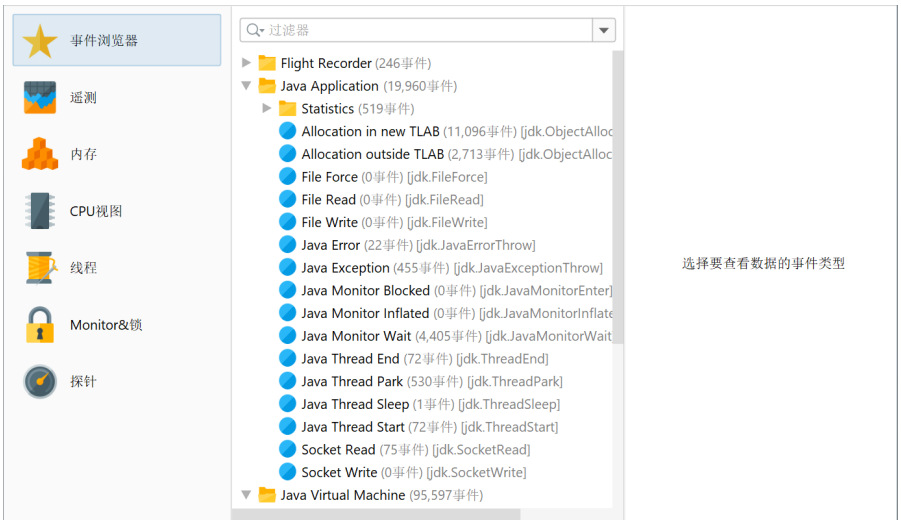
JVM表中的特殊背景颜色指示JFR记录正在运行仅指的是在JProfiler中启动的JFR记录。 如果您连接到通过其他方式启动JFR记录的JVM，将显示另一个对话框。



您现在可以选择在JProfiler中启动新记录或转储现有记录并在JProfiler中显示生成的JFR快照。 外部启动的JFR记录具有独立的生命周期，不会被JProfiler停止。

E.3 JFR事件浏览器

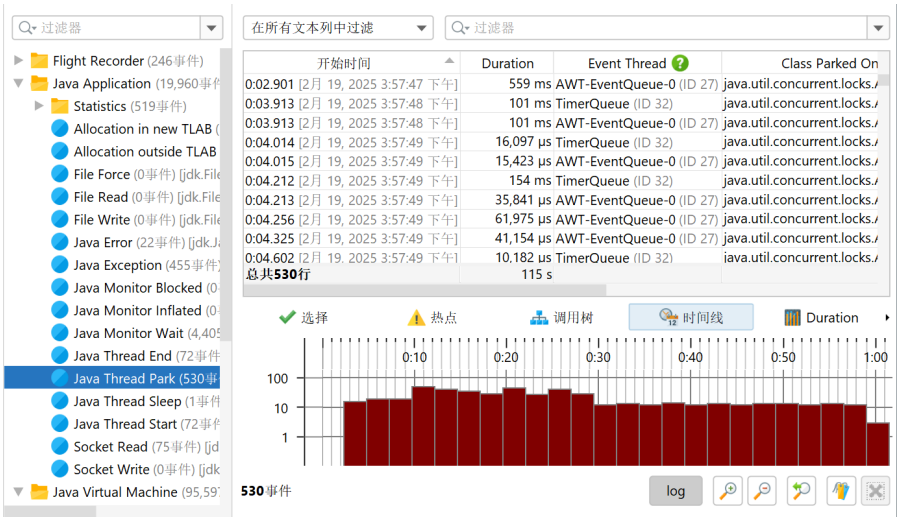
事件浏览器显示在JFR快照中记录的所有数据。



JFR将事件类型组织成层次结构类别，构成事件浏览器左侧的树。您可以选择单个事件类型来显示记录的事件。默认情况下，JProfiler显示所有注册的事件类型，即使没有为它们记录事件。或者，您可以选择在视图设置对话框中隐藏空事件类别。

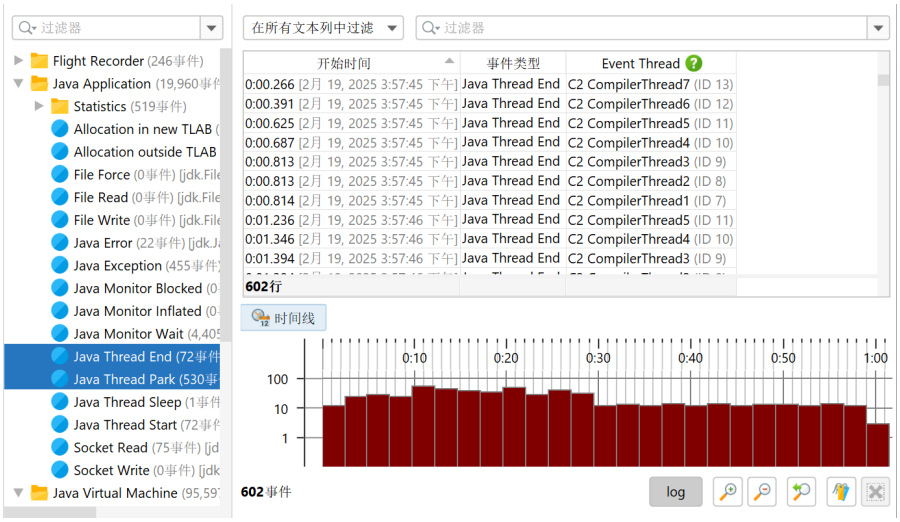
JFR事件

事件显示为主表中的行，列取决于事件类型树中的选择。



表中的事件默认按时间顺序排序。为了避免UI过载，表中仅显示前10000个事件。**底部的分析**始终从所有事件中计算。如果您设置了过滤器，它也会检查所有事件，而不仅仅是前10000个。这意味着在设置过滤器时，可能会在表中显示先前未显示的事件。

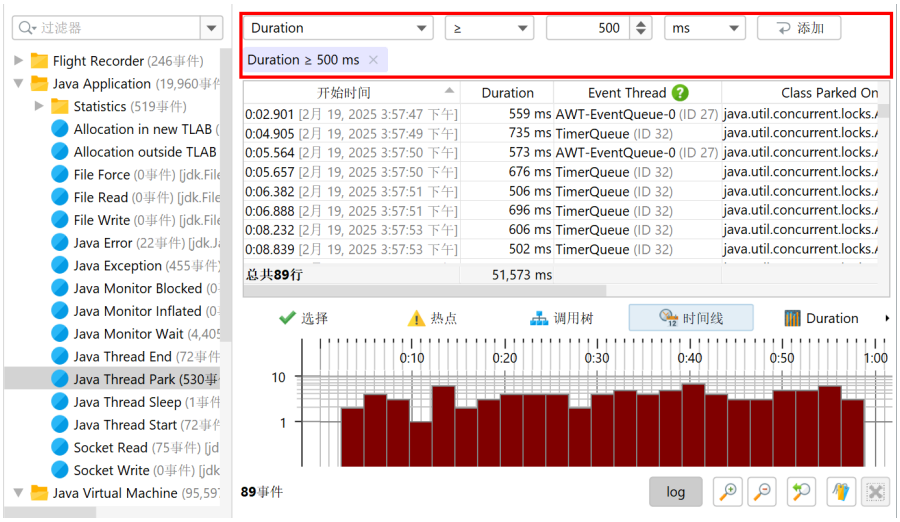
您还可以**选择多个事件类型**或整个类别。在这种情况下，表中显示所有选定事件的并集。由于每种事件类型都有自己的列集，因此仅包含所有选定事件类型共有的列。



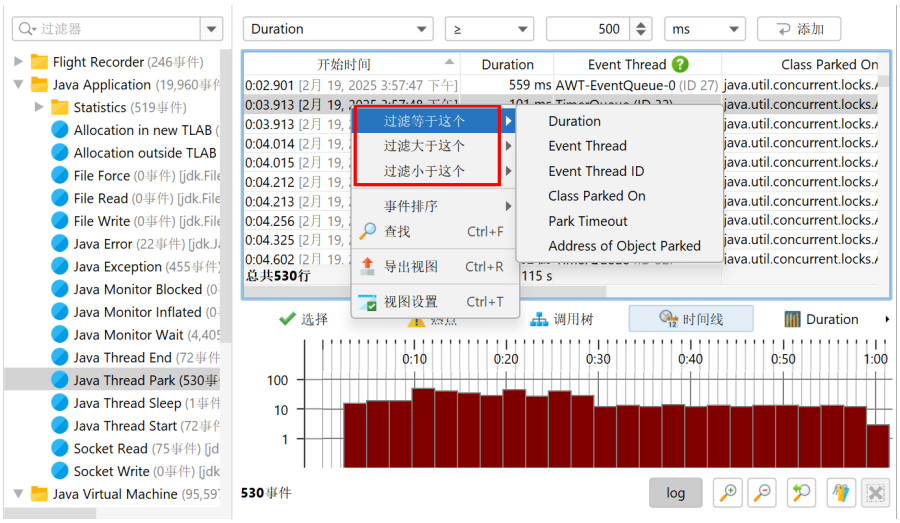
可用分析的数量也可能减少，因为分析视图是基于可用列添加的。

列宽根据实际内容自动调整，直到您调整列的大小。然后，具有相同内容类型的列宽将固定为您的选择，并且不再自动更改，直到您在视图设置对话框中**清除列宽**。具有时间或内存等单位的列中的比例也会为每个单元格自动计算。如果您希望固定列的比例以便更好地进行比较，视图设置对话框为每个这样的列提供了一个选项。在这种情况下，设置会为每个选定的事件类型单独保存。

有几种方法可以过滤事件。在表的顶部，有一个过滤器选择器，允许您在所有文本列中进行过滤，或者选择单个列并配置与列类型匹配的过滤器。

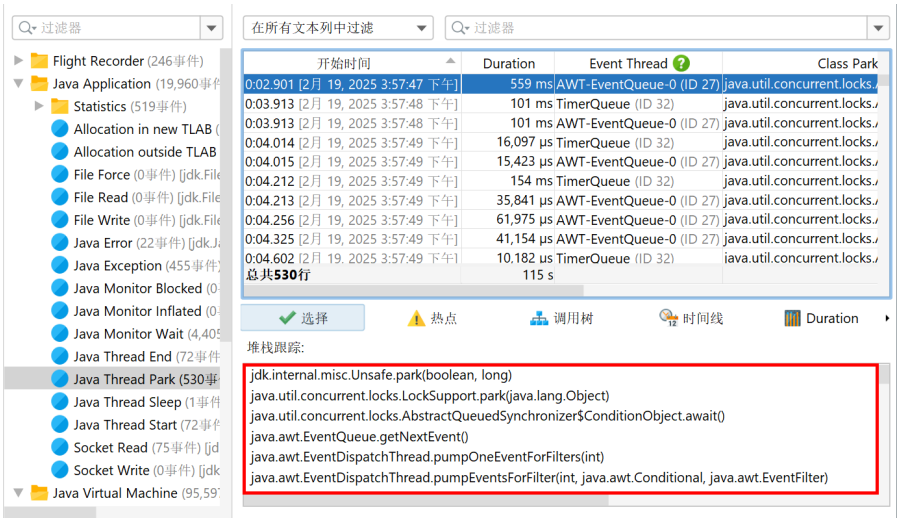


另一种过滤方法是选择感兴趣的行，并使用上下文菜单选择基于所选行中值的特定过滤器。顶部的过滤器选择器将进行调整，以显示您的选择。您现在可以选择另一个值并再次添加过滤器，它将替换同一列的先前过滤器。一般来说，**每种过滤器类型只能存在一次**，再次设置相同的过滤器将替换先前的过滤器。

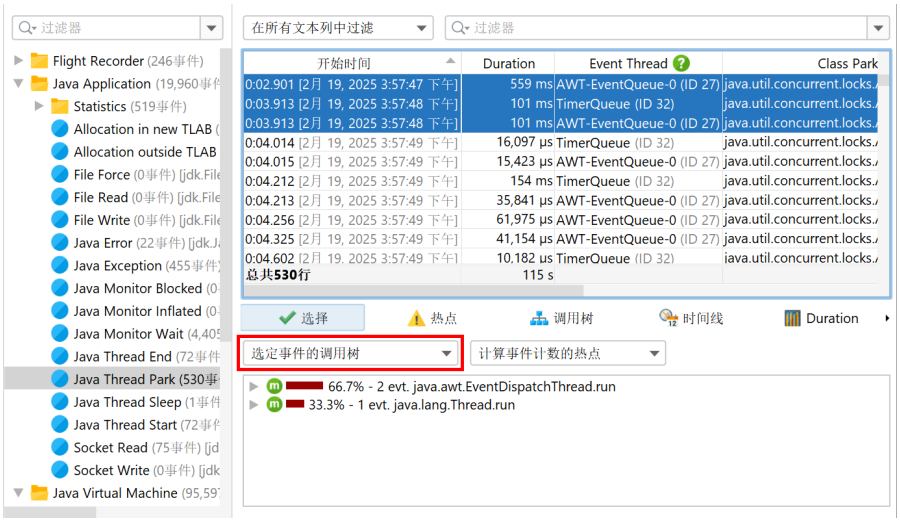


调用栈

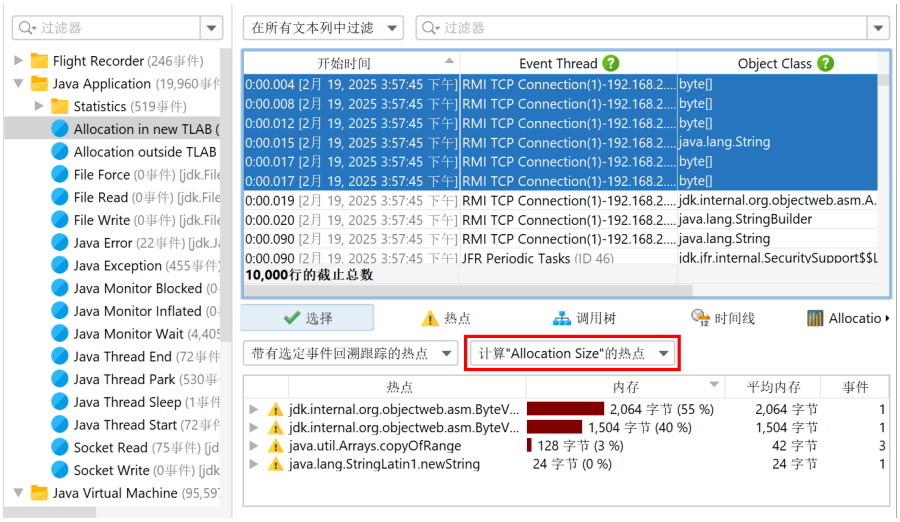
在JProfiler中，所选事件的调用栈在事件表下方的“选择”选项卡中可见。



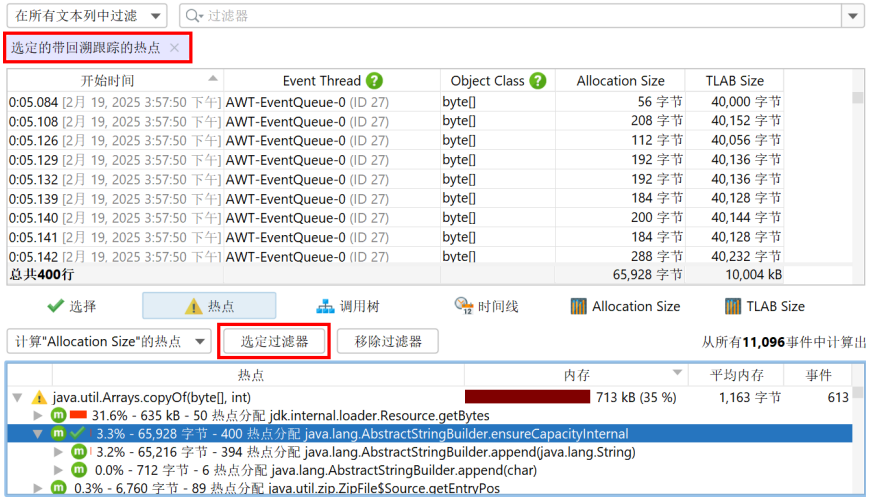
如果选择多个事件，选择选项卡会更改为一个视图，显示您选择的事件的调用栈计算出的热点或累积调用树。



默认情况下，事件计数决定了调用树和热点视图中节点的百分比。一些事件类型包括其他适合此目的的测量，例如持续时间或分配的内存。如果有这样的测量，您可以从选择选项卡中的第二个下拉菜单中选择它们作为**热点类型**。



下方分割窗格中的“热点”和“调用树”视图包含相同的视图，但它们是**为快照中的所有事件计算的**。与选择选项卡类似，它们也有一个“热点类型”下拉菜单。除了显示所有事件，您还可以从这些视图中**选择一个过滤器**。在调用树视图中，选择特定的调用栈并单击过滤选定项按钮将仅在上表中显示具有该调用栈的事件。对于热点视图，您可以选择顶级热点或回溯中的任何节点，以便仅显示调用栈以选定节点为结束的事件。

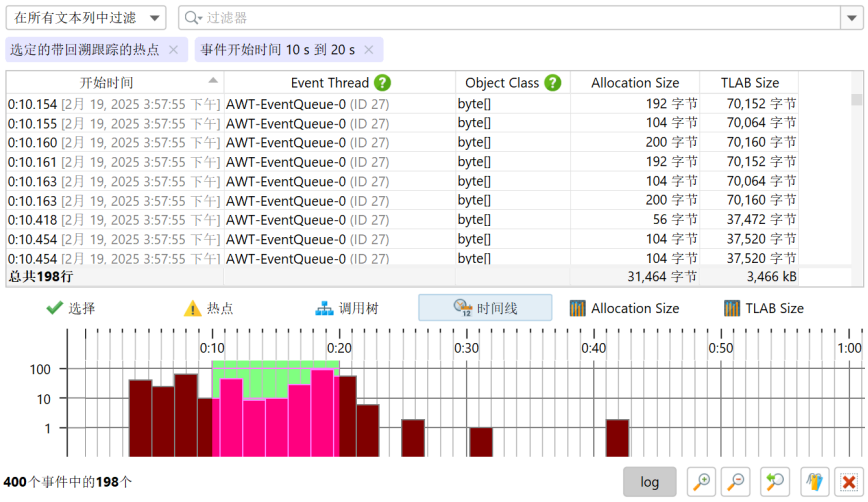


在上面的截图中，您可以看到回溯中的一个节点被选为过滤节点。除了常规的调用树图标外，它还包括一个复选标记。您可以通过顶部的标签或移除过滤器按钮移除过滤器。表中的事件计数等于所选节点上的数字。热点树仍然显示所有事件，而不包括在热点视图中设置的过滤器。

这是从分析视图中设置的过滤器的一般特性：分析视图本身是从所有过滤事件中计算的，**但不包括在分析视图中设置的过滤器**。这使得分析视图更有用，因为您可以看到您在此处选择的总事件集的哪一部分。

时间线视图

所有JFR事件都有相关的时间，因此每种事件类型或事件类型集都有一个时间线视图，显示事件的时间分布。

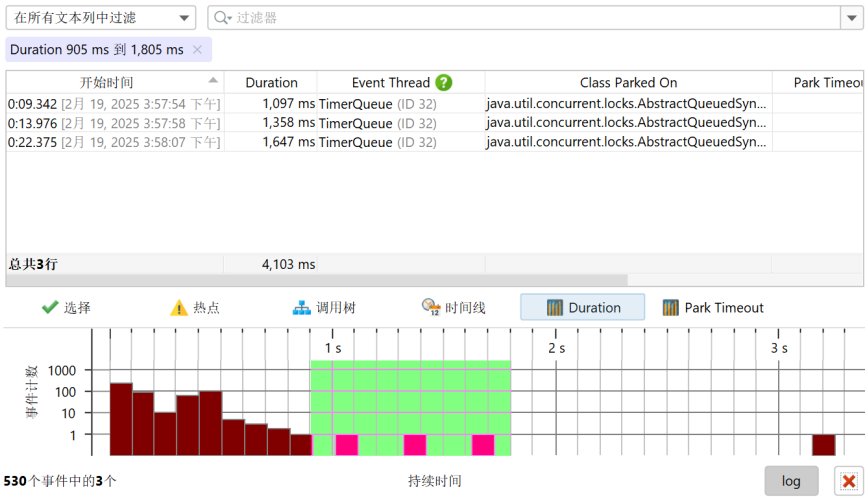


要专注于特定的时间范围，您可以**沿时间轴拖动**。在上面的示例中，我们现在有两个过滤器：一个来自热点回溯的过滤器和一个来自时间线视图的过滤器。同样，时间线视图继续显示整个时间范围，而其他分析视图现在只显示选定时间范围内的事件。

默认显示模式是**对数**，因此低事件计数的区域仍然可见于高事件计数的区域。您可以通过取消选择时间线下方的log按钮切换到线性模式。默认情况下，整个时间范围显示在可用宽度中，但您可以切换到可变时间范围，并像在JProfiler中的其他遥测中一样进行缩放和滚动。还提供了**书签**，您可以在选定的时间范围内添加垂直标记。这样，您可以跨不同的事件类型比较时间点。

直方图视图

所有可以为多个事件求和的测量，例如持续时间和分配大小，都以特殊方式处理：首先，事件表中这些测量的列在底部有一个总值。其次，调用树和热点分析视图提供一个“热点类型”下拉菜单，以这些测量而不是事件计数来计算它们的树。最后，对于每个这样的测量，直方图分析被添加到下方的分割面板中。



直方图在其垂直轴上显示事件计数，而水平轴显示选定的测量，并分为若干个箱，以便可以计算分布。箱的大小和事件计数可以从工具提示中获得。

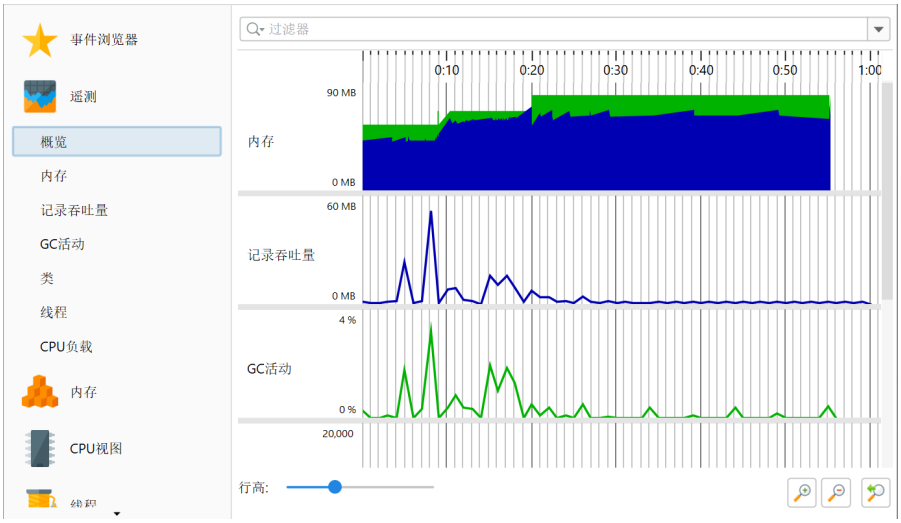
上面的截图显示了如何在直方图中设置过滤器。就像其他分析视图一样，过滤器仅适用于其他分析视图，而整个直方图仍然显示。对于时间线视图，直方图默认具有对数垂直轴。在这里，截图中选定的事件在使用线性轴时将不可见。

E.4 JFR 快照中的视图

除了 JFR 事件浏览器 [p. 210]，JProfiler 使用了一些可用于完整分析会话的视图，并用 JFR 数据填充它们。这是可能的，因为 JFR 收集了内存分配和方法执行的数据。主要的限制是记录速率较低，因此获取足够的数据以查看问题热点可能需要很长时间。

遥测

除了“记录对象遥测”外，完整分析会话中的所有遥测也可以在 JFR 快照中使用，但显示的数据有一些限制。内存遥测不显示 GC 特定的池，线程遥测不显示线程状态的线程计数，记录的吞吐量遥测显示大小而不是对象计数，并且不显示正在释放的对象。



下表显示了各种遥测使用的事件类型，以及它们是否在“默认”和“配置文件”模板中启用。

遥测	事件类型	在配置文件中启用
内存	jdk.GCHeapSummary, jdk.MetaspaceSummary	全部
记录的吞吐量	jdk.ObjectAllocationSample, jdk.ObjectAllocationInNewTLAB, jdk.ObjectAllocationOutsideTLAB	仅配置文件
GC 活动	jdk.GarbageCollection	全部
类	jdk.ClassLoadingStatistics	全部
线程	jdk.JavaThreadStatistics	全部
CPU 负载	jdk.CPULoad	全部

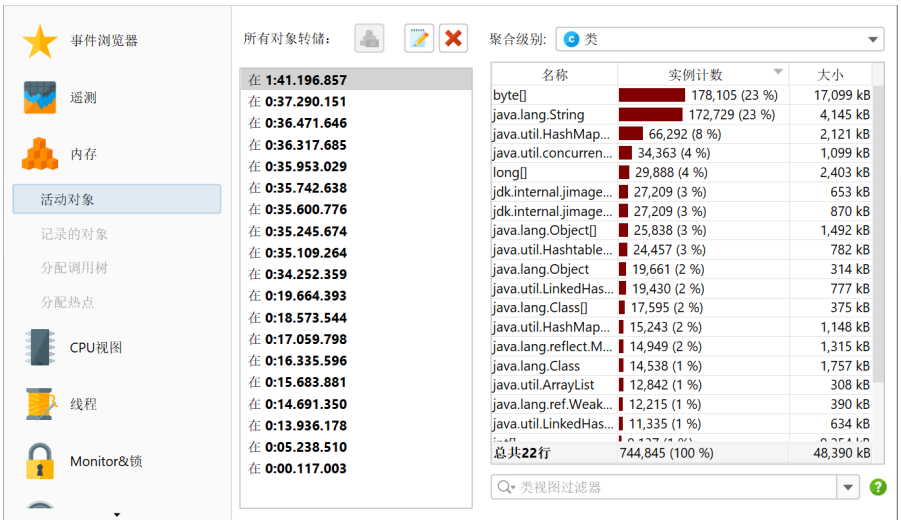
内存视图

在“内存”部分，使用了两种不同的事件类型来填充视图数据。“活动对象”视图向您展示了在完整垃圾回收后仍然保留在堆上的所有类和实例计数的统计表示。此数据仅在启用了 `jdk.ObjectCount` 事件时可用，而默认的 JFR 模板中没有启用该事件，因为它带来了显著的开销。您还可以在高级 JFR 配置中通过“垃圾收集器”下拉菜单切换此设置。在 Java 17 之前，此下拉菜单标记为“内存分析”。

如果在快照中多次记录了 `jdk.ObjectCount` 事件，视图将向您显示 **第一次和最后一次出现** 的 `jdk.ObjectCount` 事件之间的差异。通过这种方式，您可以了解在记录时间内数字如何变化，并可能提

供一些内存泄漏的指示。如果这些时间与快照记录的开始和结束点不一致，则在遥测视图中添加相应的书签。仅包括总对象大小超过固定阈值（通常为堆的 1%）的类。

对于任何严肃的调查，请考虑使用 完整分析会话 [p. 68] 或拍摄 HPROF 快照 [p. 191]。

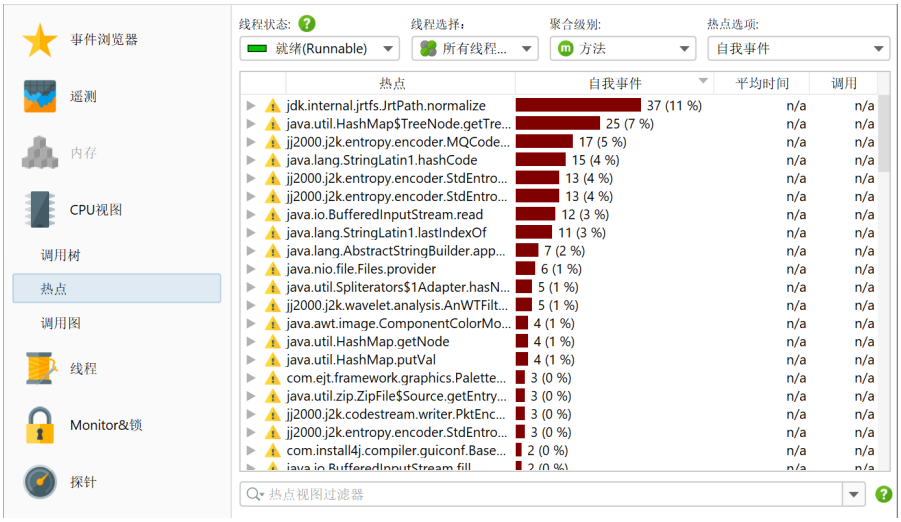


“记录对象”视图以及分配视图向您展示了自 Java 16 以来的 `jdk.ObjectAllocationSample` 事件的数据，以及早期 Java 版本中的 `jdk.ObjectAllocationInNewTLAB` 和 `jdk.ObjectAllocationOutsideTLAB` 事件。高级 UI 中的“分配分析”下拉菜单还提供了一种启用这些事件类型的方法。

与“活动对象”视图相反，它们仅显示在记录活动时分配的对象。分配由JFR采样，但大小报告为**总分配大小的估计值**。由于这种差异，这些视图报告的大小与样本计数乘以平均实例大小不对应。否则，这些视图具有与 完整分析会话中的内存视图 [p. 68] 类似的功能。

CPU 视图

“CPU视图”包括调用树、热点视图以及调用图。数据基于默认情况下在标准JFR模板中记录的`jdk.ExecutionSample`事件中的“可运行”线程状态。然而，采样率默认设置为20毫秒，这对应于JFR高级 UI 中“方法采样”设置的“正常”选项。考虑到 JFR 仅采样非常少量的随机线程，因此获取足够的数据以使热点足够突出可能需要很长时间。如有必要，请考虑降低 `jdk.ExecutionSample` 的周期。请记住，这可能导致非常大的快照大小，因为 JFR 不累积数据。



由于线程是偶尔采样的，因此不可能像在完整分析会话中那样估计实际执行时间。调用树和热点视图中显示的是 **事件计数**，而不是时间。这类似于 异步采样 [p. 63]，它具有相同的缺点。其他 JFR 线程状态为“等待”、“阻塞”和“Socket 和文件 I/O”，仍然测量时间。由于这种差异，线程状态选择器中不可用“所有线程状态”模式。

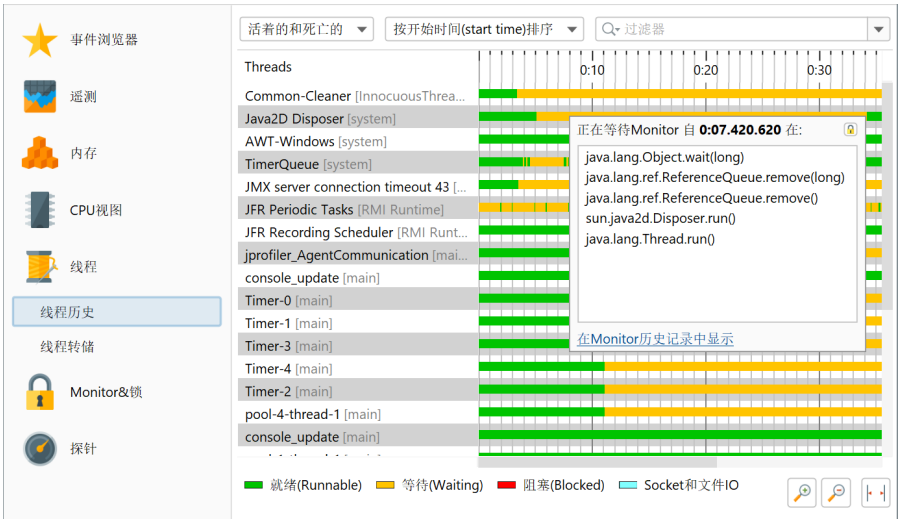
另一个考虑因素是，非可运行线程状态是从具有可配置最小持续时间阈值的事件计算得出的，这些阈值显示在线程状态选择器旁边的工具提示中。这些线程状态的实际总时间可能大得多。用于组装线程状态的事件类型表如下所示：

线程状态	事件类型
可运行	jdk.ExecutionSample
等待	jdk.JavaMonitorWait, jdk.ThreadSleep, jdk.ThreadPark
阻塞	jdk.JavaMonitorEnter
Socket 和文件 I/O	jdk.SocketRead, jdk.SocketWrite, jdk.FileRead, jdk.FileWrite

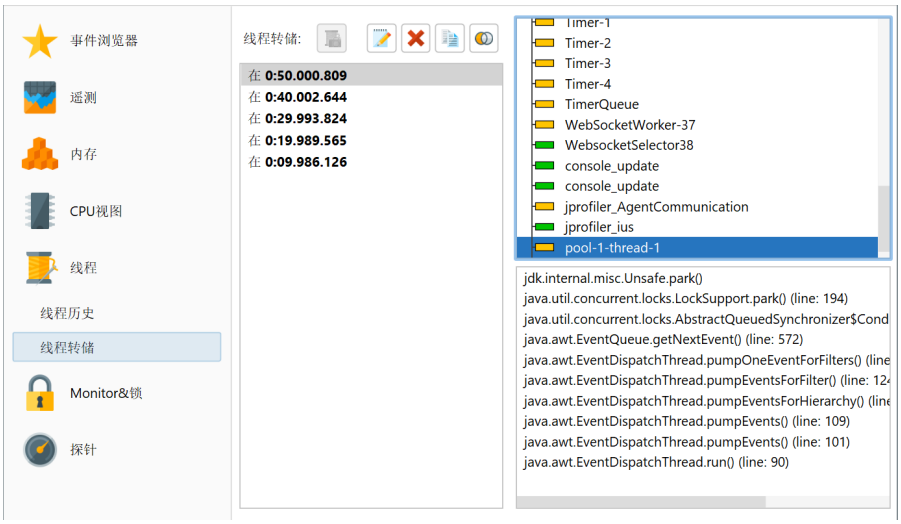
视图的功能在 CPU 视图帮助主题 [p. 51] 中进行了说明。请注意，完整分析会话的许多功能在 JFR 上下文中不可用。

线程和监视器视图

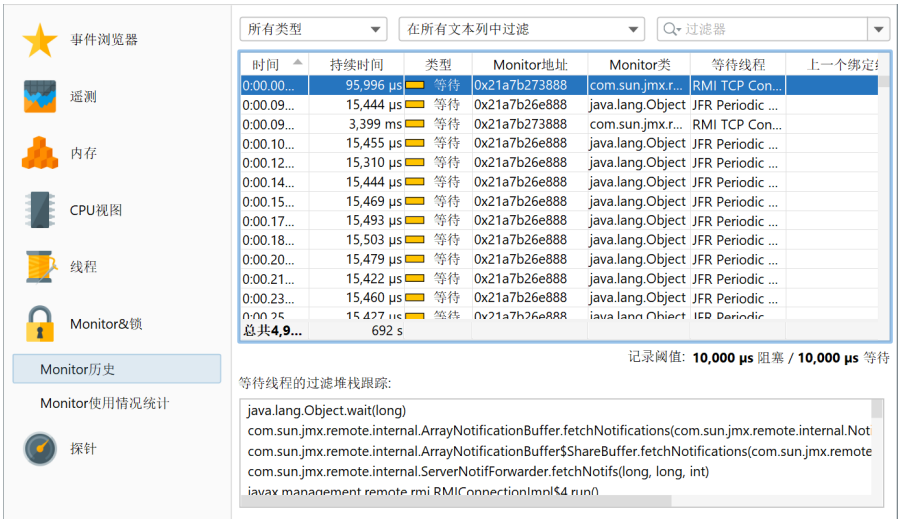
从时间顺序方法采样数据中，可以计算线程历史视图，包括显示等待和阻塞时间的堆栈跟踪的工具提示。



线程转储是 JFR 和 JProfiler 中的一个功能，并在同一视图中显示。在这种情况下，事件浏览器不是替代品，因为它无法显示 `jdk.ThreadDump` 事件的线程转储列的结构化内容。在线程转储视图中，您还可以 比较不同的线程转储 [p. 90]。



从 `jdk.JavaMonitorWait`、`jdk.ThreadSleep` 和 `jdk.ThreadPark` 事件中，JProfiler 计算出类似于 完整分析会话 [p. 90] 的监视器历史，只是没有阻塞线程的信息。如果您需要这些信息来解决问题，请切换到完整分析会话。这也意味着完整分析会话中的锁定图形在JFR快照中不可用。显示等待事件汇总信息的监视器使用统计信息存在，并且仅显示等待时间。



探针

完整分析会话中的一些 JVM 探针在 JFR 快照中具有等效的数据源。与事件浏览器相比，它们的主要优势在于它们结合了多个相关的事件类型。下表显示了可用探针及其用作数据源的事件类型。

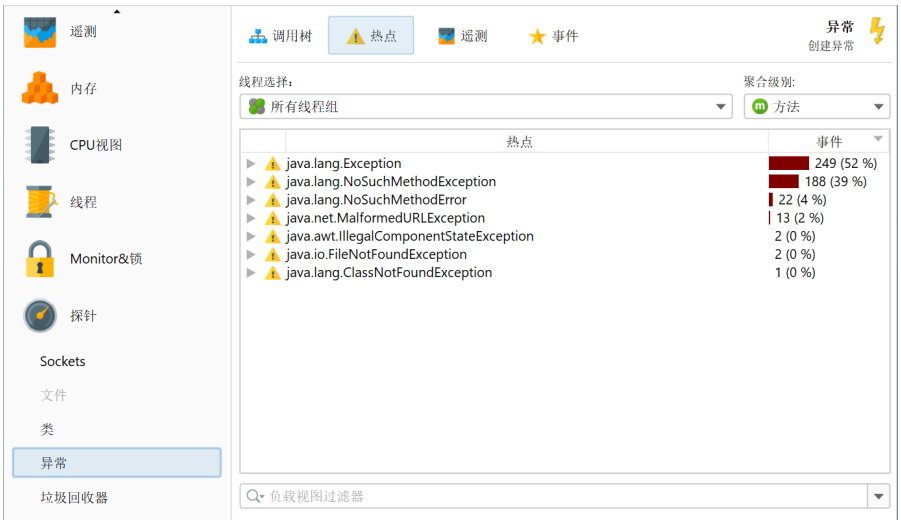
探针	事件类型	在配置文件中启用
Sockets	jdk.SocketRead, jdk.SocketWrite	全部
文件	jdk.FileRead, jdk.FileWrite	全部
类	jdk.ClassLoad, jdk.ClassUnload, jdk.ClassDefine	无
异常	jdk.JavaErrorThrow, jdk.JavaExceptionThrow	错误在两者中，异常在无

探针	事件类型	在配置文件中启用
垃圾收集器	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	全部

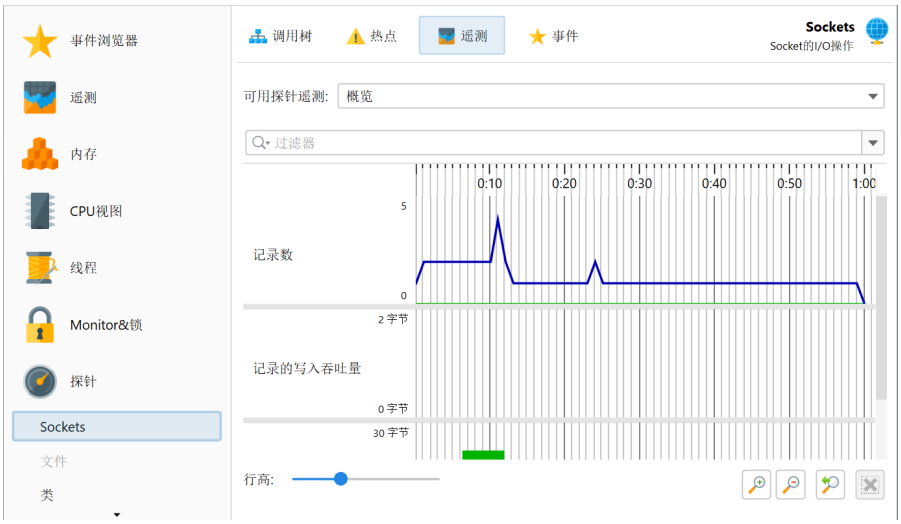
类加载在高级 JFR UI 中有一个单独的复选框，可以打开所有三个类加载事件。

每个探针显示多个视图。与事件浏览器相比，重点是聚合数据而不是单个事件。这也是JProfiler中的探针在概念上与 JFR 数据收集的不同之处。

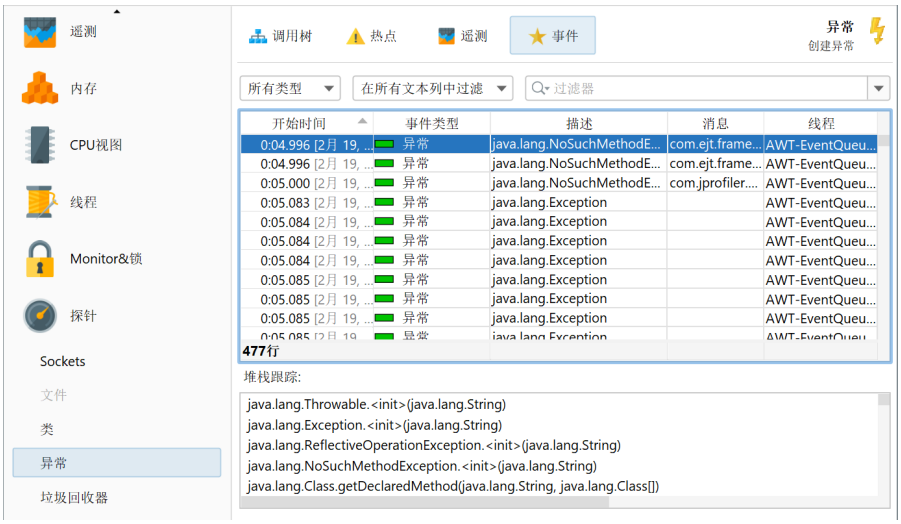
除了垃圾收集器探针外，所有探针都有以下视图：调用树和热点视图允许您选择单个线程或线程组以及聚合级别。默认情况下，显示所有线程，并且聚合级别设置为“方法”。



遥测视图显示来自记录数据的一种或多种遥测，并提供一个概览页面，显示所有遥测。可以通过单击遥测名称打开完整的遥测。通过沿时间轴拖动，您可以在事件视图中选择相应的事件。



事件视图类似于 JFR 浏览器中的视图。然而，它显示了与底层 JFR 事件相对应的多种事件类型，并提供了类型选择器。单选和多选的过滤和堆栈跟踪显示与事件浏览器中的处理方式相同。此外，还有时间和内存测量的直方图视图，您可以通过沿水平轴拖动来选择范围。



垃圾收集器视图是特殊的，因为完整分析会话可以在 Java 17 或更高版本的分析会话中显示完全相同的信息。当 JVM 探针类别中的垃圾收集器探针被记录时，使用 JFR 流来获取必要的数据。有关更多信息，请参阅 垃圾收集器分析 [p. 109] 章节。

F 详细配置

F.1 连接问题故障排除

当无法建立分析会话时，首先要查看被分析应用程序或应用服务器的终端输出。对于应用服务器，stderr流通常会写入日志文件。这可能是一个单独的日志文件，而不是应用服务器的主日志文件。例如，Websphere应用服务器会写入一个`native_stderr.log`文件，其中仅包含stderr输出。根据stderr输出的内容，问题的搜索方向会有所不同：

连接问题

如果stderr包含`"Waiting for connection ..."`，则被分析应用程序的配置是正确的。问题可能与以下问题有关：

- 您是否忘记在本地机器上的JProfiler GUI中启动"Attach to remote JVM"会话？除非分析代理被配置为立即使用"nowait"选项启动，否则它将等待JProfiler GUI连接后才让VM继续启动。
- 会话设置中是否正确配置了主机名或IP地址？
- 您是否配置了错误的通信端口？通信端口与HTTP或其他标准端口号无关，且不得与任何已在使用的端口相同。对于被分析应用程序，通信端口被定义为分析VM参数的一个选项。使用VM参数`-agentpath:<path to jprofilerti library>=port=25000`，将使用25000端口。
- 您是否尝试通过仅监听环回接口的直接连接来连接到代理？默认情况下，代理仅监听环回接口。您可以配置JProfiler以设置SSH隧道到远程机器。如果不需要加密，也可以使用`address=[IP address]`选项用于`-agentpath`参数。
- 本地机器和远程机器之间是否有防火墙？可能存在用于传入和传出连接的防火墙，甚至在中间网关机器上也有防火墙。

端口绑定问题

如果stderr包含关于无法绑定socket的错误消息，则端口已在使用中。在这种情况下，请检查以下问题：

- 您是否多次启动了被分析应用程序？每个被分析的应用程序需要一个单独的通信端口。
- 是否有以前分析运行的僵尸Java进程阻塞了端口？
- 是否有其他应用程序正在使用通信端口？

如果stderr中没有以`JProfiler>`为前缀的行，并且您的应用程序或应用服务器正常启动，则`-agentpath:[path to jprofilerti library]` VM参数未包含在Java调用中。您应该找出启动脚本中实际执行的Java调用，并在那里添加VM参数。

附加问题

当附加到正在运行的JVM时，有时可能在所有JVM列表中看不到感兴趣的JVM。要找出此问题的原因，了解附加机制的工作原理很重要。当JVM启动时，它会将PID文件写入临时目录中的`hsperfdata_$USER`目录，从而被发现。只有相同用户或管理员用户才能附加到JVM。JProfiler可以帮助您以管理员用户身份连接到JVM。

在Windows上，使用Show Services按钮显示所有JVM服务进程。JProfiler安装了一个帮助服务，该服务将以系统帐户运行，可以连接到以系统帐户运行的服务以及配置的用户帐户。该服务的名称为"JProfiler helper"，当您点击该按钮时安装。您必须确认UAC提示以允许安装服务。当JProfiler退出时，服务将再次卸载。

在Linux上，您可以在附加对话框中使用用户切换器以root帐户附加。此用户切换器在分析本地JVM时以及附加到远程Linux或macOS机器时显示。对于远程附加情况，您还可以切换到不同的非root用户。如果您有root密码，请始终切换到root，而不是运行服务的实际用户。

如果在Linux上即使您认为应该可见的JVM不可见，问题通常与临时目录有关。一种可能性是`/tmp/hsperfdata_$USER`目录的访问权限错误。在这种情况下，删除目录并重新启动JVM。要附加的进程必须具有对/tmp的写访问权限，否则不支持附加。

如果您使用**systemd**，您感兴趣的进程可能在其systemd服务文件中设置了`PrivateTmp=yes`。然后pid文件被写入不同的位置。如果您在附加对话框中使用用户切换器切换到root用户，或者以root身份使用CLI工具，JProfiler将处理此问题。

远程附加的自动代理下载

对于远程附加，JProfiler需要远程目标平台的代理库。如果它们在本地不可用，它将尝试下载它们。如果有防火墙阻止HTTPS连接到`https://download.ej-technologies.com`或使用中间人方案检查SSL连接以解密流量，则此操作可能会失败。在后一种情况下，HTTPS连接将失败，因为JProfiler没有防火墙的证书。

当发生代理下载错误时，JProfiler提供手动解决方案。会显示一个对话框，显示从网站手动下载代理档案的说明，并在继续远程附加操作之前定位下载的档案。

由于代理文件被缓存，因此对于每个远程平台，这是一项一次性操作。当JProfiler更新时，代理会更改，下载将需要重复。

F.2 脚本在JProfiler中的使用

JProfiler内置的脚本编辑器允许你在JProfiler GUI的各个地方输入自定义逻辑， 包括自定义探针配置、拆分方法、堆遍历器过滤器等等。

编辑区上方的方框显示了脚本的可用参数以及它的返回类型。通过调用菜单中的帮助->显示Javadoc概述，你可以获得 `com.jprofiler.api.*` 包中类的更多详情。

一些包可以在不使用完全限定的类名的情况下使用。这些包是:

- java.util.*
- java.io.*

你可以把一些导入语句作为文本区域的第一行，以避免使用完全限定的类名。

所有的脚本都会被传递一个`com.jprofiler.api.agent.ScriptContext` 的实例， 允许你保存连续调用脚本之间的状态。

为了获得最完整的编辑器功能，建议在常规设置中配置一个JDK。默认情况下，使用JProfiler运行使用的JRE。 在这种情况下，代码补全不提供参数名和JRE中类的Javadoc。

脚本类型

脚本可以是表达式。表达式没有尾部的分号，并且评估为所需的返回类型。例如

```
object.toString().contains("test")
```

将作为堆遍历器的传出引用视图中的过滤脚本。

也可以，一个脚本由一系列Java语句组成，最后一条语句是所需返回类型的返回语句:

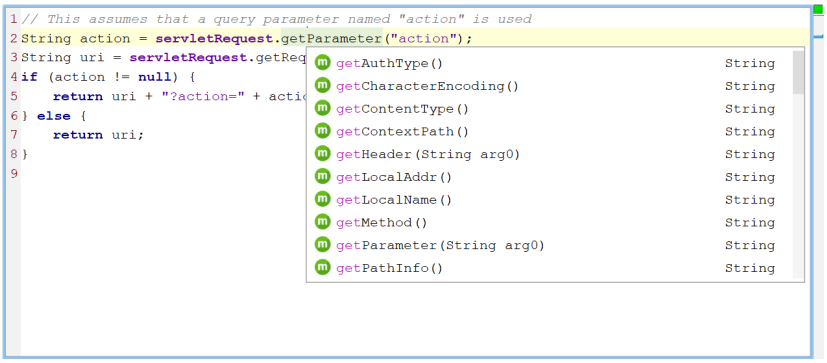
```
import java.lang.management.ManagementFactory;
return ManagementFactory.getRuntimeMXBean().getUptime();
```

上面的例子可以用于脚本遥测。JProfiler会自动检测你输入的是表达式还是脚本。

如果你想重用之前输入的脚本，你可以从脚本历史中选择它。 如果你点击 显示历史 工具栏按钮，所有以前使用过的脚本都会显示出来。脚本按脚本签名组织，默认选择当前脚本签名。

代码补全

按CTRL-Space，会弹出一个代码补全提示弹窗。另外，输入一个点(".")并且延迟一会后如果没有输入其他字符，也会显示该弹窗。该延迟时间可在编辑器设置中配置。在弹出窗口显示时，你可以继续输入或使用Backspace删除字符， 弹出窗口将相应更新。支持"驼峰"补全。例如，输入NPE ，点击CTRL-Space ， 推荐的类中将包含java.lang.NullPointerException。 如果你接受一个没有自动导入的类，将插入完全限定名。



补全弹窗可以提示:

- 变量和脚本参数。脚本参数以粗体字显示。
- 包，当键入import语句时
- 类
- 字段，当上下文是一个类
- 方法，当上下文是一个类或一个方法的参数列表时。

参数的类，如果既不包含在配置的会话类路径中，也不包含在配置的JDK中，就会被标记为 `[unresolved]` ， 并改为通用的`java.lang.Object` 类型。为了能够对这样的参数调用方法，并为其获取代码补全， 请在应用程序设置中的类路径中添加缺少的JAR文件。

问题分析

你输入的代码会被分析并检查是否有错误和警告条件。错误在编辑器中显示为红色下划线，在右边的边列中显示为红色条纹。警告，如未使用的变量声明，在编辑器中显示为黄色背景，在边列中显示为黄色条纹。将鼠标悬停在编辑器中的错误或警告上，或将鼠标悬停在边列的条纹上，都会显示错误或警告信息。

如果代码中没有警告或错误，右边列顶部的状态指示为绿色，如果有警告则为黄色，如果发现错误则为红色。 你可以在编辑器设置中配置问题分析的阈值。

如果对话框右上角的边列图标是绿色的，那么你的脚本将被编译，除非你在编辑器设置中禁用了错误分析。在某些情况下，你可能想尝试真正的编译。选择菜单中的代码->测试编译将编译该脚本并在一个单独的对话框中显示任何错误。在保存脚本时使用 好的 按钮不会测试脚本的语法正确性，除非马上使用该脚本。

键绑定

按SHIFT-F1，会在浏览器中打开Javadoc页面，描述光标位置的元素。只有在常规设置中为代码编辑器配置了具有有效Javadoc位置的JDK，才能显示Java运行时库的Javadoc。

Java代码编辑器中的所有键绑定都是可配置的。从窗口菜单中选择 设置->键映射 以显示键映射编辑器。键绑定保存在文件`$HOME/.jprofiler15/editor_keymap.xml`。这个文件只有在复制了默认的键映射时才存在。当把JProfiler安装迁移到不同的计算机时，可以复制这个文件来保存键绑定。

F.3 Custom Help

如果你有一个内部网站为用户提供额外指导，你可以在工具栏和“帮助”菜单上添加一个额外的帮助按钮。 通过向.vmoptions文件添加下列属性:

```
-Dcustom.help.url=https://www.internal.website.com
-Dcustom.help.toolBarText=内部#帮助
-Dcustom.help.actionName=显示内部帮助
```

必须定义所有这三个属性才能在UI中看到这个操作。属性custom.help.toolBarText是要在工具栏中显示的文本。它应该尽可能简明扼要， 可以通过 # 分隔符添加第二行就像上面示例中那样。

文件.vmoptions，在Windows和Linux上位于<JProfiler >/bin/jprofiler.vmoptions，在macOS上位于/Applications/JProfiler.app/Contents/vmoptions.txt。 另外，还有用户可写的位置，在Windows上位于%USERPROFILE%\.jprofiler15\jprofiler.vmoptions下， Linux上位于$HOME/.jprofiler15/jprofiler.vmoptions， macOS上位于$HOME/Library/Preferences/jprofiler.vmoptions。

F.4 在启动时设置配置文件设置

在配置文件代理可以开始任何记录之前，必须设置配置文件设置。当您使用 JProfiler UI 连接时会发生这种情况。在某些情况下，要求配置文件代理在启动时知道配置文件设置。主要用例是：

- **离线分析**

  使用触发器或API记录数据并保存快照。在此模式下，JProfiler GUI无法连接。有关更多信息，请参阅 离线分析帮助主题 [p. 119]。

- **在无头机器上使用 jpcontroller 进行分析**

  命令行实用程序 jpcontroller [p. 231] 可以代替 JProfiler GUI 以交互方式或使用非交互命令文件记录数据和保存快照。然而，jpcontroller 没有配置配置文件设置的功能，因此必须提前设置。

- **远程附加到旧版 OpenJ9 和 IBM JVM**

  旧版 OpenJ9 和 IBM JVM 在 8u281、11.0.11 和 Java 17 之前无法在不危及被分析进程稳定性的情况下重新定义类，因此必须在启动时设置配置文件设置。JProfiler 中远程集成向导的“被分析 JVM”步骤会询问您 JVM 的类型，如果您在那里选择 旧版 OpenJ9 和 IBM JVM，向导将添加下面讨论的选项。

一般来说，在启动时设置配置文件设置是最有效的操作模式，因为需要执行的类重新定义次数最少。如果减少的便利性不是问题，它可以用于任何类型的配置文件会话。

在启动时设置配置文件设置

如果您使用集成向导，请在“本地或远程”步骤中选择 在远程计算机上 选项，然后在“配置同步”步骤中选择 在启动时应用配置 选项。然后，向导将添加与以下段落中讨论的相同选项。

如果您已在启动脚本中添加了 `-agentpath` VM 参数以加载配置文件代理，则可以通过添加以下内容来设置配置文件设置

```
,config=<配置文件路径>,id=<会话 ID>
```

到 `-agentpath` 参数。完整的参数将如下所示：

```
-agentpath:/path/to/libjprofilerti.so=port=8849,nowait,config=/path/to/config,id=123
```

如果您使用 `jpenable` 在进程启动后加载配置文件代理，您可以在交互执行中选择 离线模式 并在那里指定配置和 ID。或者，传递 `--offline`、`--config` 和 `--id` 参数以进行非交互执行。

准备配置文件

引用的配置文件可以是当前机器上JProfiler安装的配置文件，在这种情况下，根本不需要指定配置参数。JProfiler配置文件位于`$HOME/.jprofiler15/jprofiler_config.xml`或`%USERPROFILE%\.jprofiler15\jprofiler_config.xml`，是 `-agentlib` VM 参数的 `config` 选项的默认值。

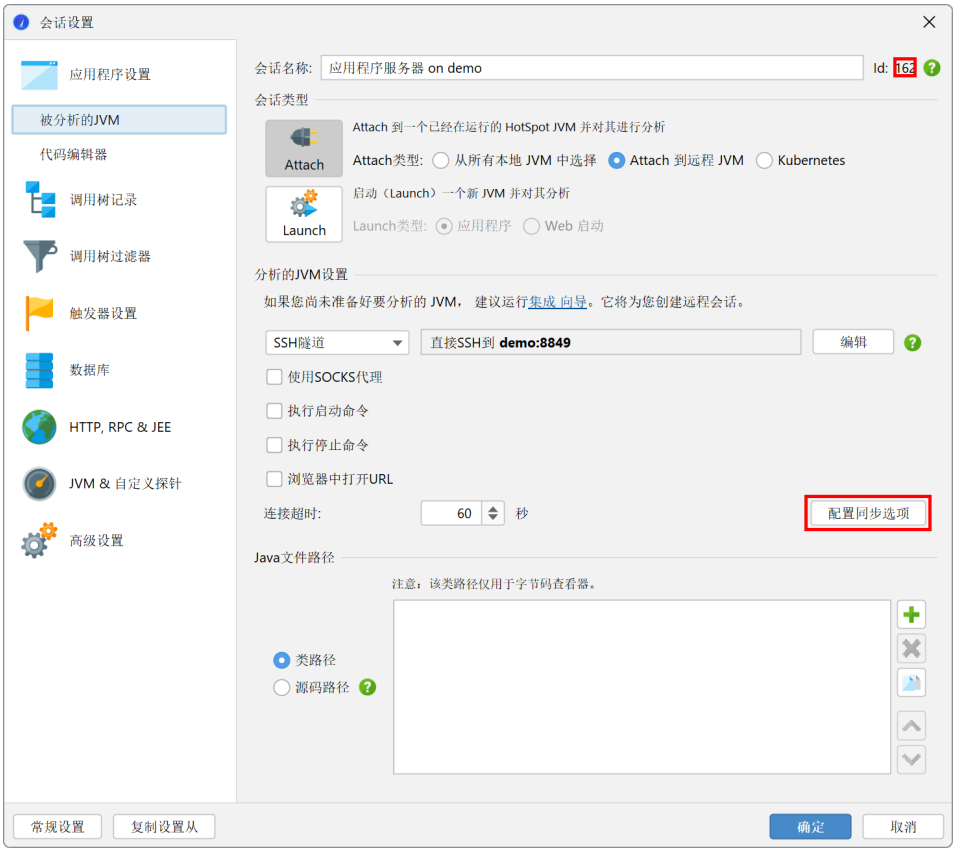
通常，自动化分析应在不同的机器上执行，无法引用本地JProfiler配置文件。然后，您可以在本地机器上的 JProfiler UI 中准备一个包含配置文件设置的会话，通过 会话->导出会话设置 导出并传输到运行 JProfiler 的机器上。

会话ID可以在会话设置对话框的“应用程序设置”选项卡的右上角看到（请参见下面的截图）。如果导出的文件仅包含一个会话，则不需要指定 `id` 参数。

同步配置文件

完成初始设置后，您可能希望调整未来分析运行的配置文件设置。这需要每次进行修改时将配置文件复制到远程机器上。

JProfiler 中的远程会话具有“配置同步”功能，可以为您自动化此过程。



如果会话是通过 SSH 启动的，您可以通过 SSH 直接将配置文件复制到远程机器上。否则，您仍然可以将配置文件复制到可能挂载在远程机器上的本地目录中。最后，您可以执行任意命令，通过其他方式复制配置文件。

G 命令行参考

G.1 用于分析的命令行可执行文件

JProfiler 包含许多命令行工具，用于设置分析代理和从命令行控制分析操作。

将分析代理加载到正在运行的 JVM 中

使用命令行工具 bin/jpenable，您可以将分析代理加载到任何版本为 6 或更高的正在运行的 JVM 中。通过命令行参数，您可以自动化该过程，使其无需用户输入。支持的参数有：

```
用法: jpenable [选项]

jpenable 在选定的本地 JVM 中启动分析代理，以便您可以从不同的计算机连接
到它。如果 JProfiler GUI 在本地运行，您可以直接从 JProfiler GUI 附加，而无需运行此可执行文件。

* 如果没有给出参数，jpenable 会尝试发现尚未被分析的本地 JVM，并在命令行上
  请求所有必需的输入。
* 使用以下参数，您可以在命令行上部分或完全提供整个用户输入：

  -d  --pid=<PID>     应该被分析的 JVM 的 PID
  -n  --noinput       在任何情况下都不要求用户输入
  -h  --help          显示此帮助
      --options=<OPT> 传递给代理的调试选项

  GUI 模式: (默认)
  -g  --gui           将使用 JProfiler GUI 附加到 JVM
  -p  --port=<nnnnn>  分析代理应监听 JProfiler GUI 连接的端口
  -a  --address=<IP>  分析代理应监听的地址。没有此参数，附加仅可能从 localhost。使用
                      0.0.0.0 监听所有地址

  离线模式:
  -o  --offline       JVM 将在离线模式下被分析
  -c  --config=<PATH> 配置文件的路径，其中包含分析设置
  -i  --id=<ID>       配置文件中的会话 ID。如果配置文件仅包含一个会话，则不需要。

请注意，JVM 必须以与 jpenable 相同的用户身份运行，否则 JProfiler 无法连接到它。
一个例外是 Windows 服务将运行在本地系统账号下，如果你
与 jpenable 交互的方式列出它们。
```

保存 HPROF 快照

如果您只需要堆快照，可以考虑使用 bin/jpdump 命令行工具，该工具可以在不将分析代理加载到 VM 中的情况下保存 HPROF 快照 [p. 191]：

```
用法: jpdump [options]

jpdump 将一个本地运行JVM中的堆,转储到一个文件。
Hotspot 虚拟机生成HPROF文件, OpenJ9 虚拟机生成PHD文件。
然后HPROF 和 PHD 文件可以在JProfiler GUI中打开。

* 如果没有提供参数，jpdump 将列出所有本地运行的 JVM。
* 你可以提供以下部分或全部参数
  用户在命令行输入：

  -p  --pid=<PID>     指定要做堆转储的 JVM
                      的 PID, 不会再被问及其他问题
  -a  --all           保存所有对象。如果未指定，则只有活性对象会被
                      转储
  -f  --file=<PATH>   转储文件路径。如果未指定，转储文件
```

```
                        <VM name>.hprof 会被写到当前路径下。
                        如果文件已经存在，会追加一个数字。
  -h  --help            显示该帮助信息

注意，运行JVM的用户必须和运行jpdump的用户相同，否则
JProfiler 无法连接JVM。
一个例外是 Windows 服务将运行在本地系统账号下，如果你
与 jpdump 交互的方式列出它们。
```

这比加载分析代理并保存JProfiler堆快照的开销更低。此外，由于分析代理永远无法卸载，因此此方法适用于在生产中运行的 JVM。

控制分析代理

当您在没有参数的情况下启动 `bin/jpcontroller` 可执行文件时，它会尝试连接到本地机器上的被分析 JVM。如果发现多个被分析的 JVM，您可以从列表中选择一个。

`jpcontroller` 只能连接到已设置分析设置的 JVM，因此如果 JVM 是使用 `-agentpath` VM 参数的 "nowait" 选项启动的，它将无法工作。该选项是在集成向导的 "启动模式" 屏幕上选择 立即启动，稍后使用 JProfiler GUI 连接 单选按钮时设置的，并且尚未有 JProfiler GUI 连接到代理。使用 `jpenable` 而不带 `--offline` 参数也需要 JProfiler GUI 的连接，才能让 `jpcontroller` 连接到被分析的进程。

如果您想连接到远程计算机上的进程，或者 JVM 不是版本为 6 或更高的 HotSpot JVM，您必须将 VM 参数 `-Djprofiler.jmxServerPort=[port]` 传递给被分析的 JVM。MBean 服务器将在该端口上发布，您可以将选择的端口作为参数传递给 `jpcontroller`。使用附加的 VM 参数 `-Djprofiler.jmxPasswordFile=[file]`，您可以指定一个包含键值对形式的属性文件 `user password` 来设置身份验证（用空格或制表符分隔）。注意，这些VM参数会被`com.sun.management.jmxremote.port` VM 参数覆盖。

通过显式设置 JMX 服务器，您可以使用命令行控制器通过调用 `jpcontroller host:port` 连接到远程服务器。如果远程计算机只能通过 IP 地址访问，您必须将 `-Djava.rmi.server.hostname=[IP address]` 作为 VM 参数添加到远程 VM。

默认情况下，`jpcontroller`是一个交互式命令行工具，但您也可以使用它自动化分析会话，而无需手动输入。自动调用将传递 `[pid | host:port]` 以选择被分析的 JVM 以及 `--non-interactive` 参数。此外，还会从 stdin 或通过 `--command-file` 参数指定的命令文件中读取命令列表。每个命令都从新行开始，空行或以 "#" 注释字符开头的行将被忽略。

此非交互模式的命令与 JProfiler MBean [(1)] 中的方法名称相同。它们需要相同数量的参数，用空格分隔。如果字符串包含空格，则必须用双引号括起来。此外，还提供了一个`sleep <seconds>`命令，用于暂停指定的秒数。这允许您开始记录，等待一段时间，然后将快照保存到磁盘。

请注意，分析设置必须在分析代理中设置。这会在您使用 JProfiler UI 连接时发生。如果您从未使用 JProfiler UI 连接，则必须在启动命令中手动设置它们或使用 jpenable，请参阅 有关在启动时设置分析设置的帮助主题 [p. 229] 以获取更多信息。

jpcontroller 的支持参数如下所示：

```
用法: jpcontroller [options] [host:port | pid]

* 如果没有提供任何参数， jpcontroller 将尝试发现
    正在被分析的本地 JVM
* 如果指定了一个数字，则 jpcontroller 将尝试连接到
    进程 ID 为 [pid] 的 JVM。 如果该 JVM 未被分析， jpcontroller 将无法
    连接。 在这种情况下， 首先使用 jpenable 实用程序。
```

[(1)] https://www.ej-technologies.com/resources/jprofiler/help/api/javadoc/com/jprofiler/api/agent/mbean/RemoteControllerMBean.html

```
* 除此之外, jpcontroller 将连接到 "host:port", port 已经
   在被分析 JVM 的 VM 参数 -Djprofiler.jmxServerPort=[port]
   中指定过。

可以使用以下参数选项(options):
  -n --non-interactive      运行一个从标准输入(stdin)
                            读取命令列表的自动会话
  -f --command-file=<PATH>  从一个文件而不是标准输入(stdin)读取命令， 仅当和
                            --non-interactive 一块使用。

非交互命令的语法:
  请参见 RemoteControllerMBean (https://bit.ly/2DimDN5) 的 javadoc
  操作列表。 各参数以空格分割，如果某个参数含有空格
  必须使用引号引起来。 例如:

    addBookmark "Hello world"
    startCPURecording true
    startProbeRecording builtin.JdbcProbe true true
    sleep 10
    stopCPURecording
    stopProbeRecording builtin.JdbcProbe
    saveSnapshot /path/to/snapshot.jps

  sleep <seconds> 命令暂停指定的秒数。
  空行和以 # 开头的行将被忽略。
```

G.2 用于处理快照的命令行可执行文件

当使用离线分析 [p. 119]以编程方式保存快照时，可能还需要以编程方式从这些快照中提取数据或报告。 JProfiler 提供了两个独立的命令行可执行文件，一个用于从快照中导出视图，一个用于比较快照。

从快照导出视图

可执行文件bin/jpexport将视图数据导出为多种格式。如果您使用-help选项执行它，您将获得可用视图名称和视图选项的信息。 出于简洁的原因，下面的输出中省略了重复的帮助文本。

```
用法: jpexport "快照文件" [全局选项]
              "视图名称" [选项] "输出文件"
              "视图名称" [选项] "输出文件" ...

其中“快照文件”是具有以下扩展名之一的快照文件:
          .jps, .hprof, .hpz, .phd, .jfr
      "视图名称"是下边列出的视图名称之一
      [选项]是一个格式为“-选项=值”的一个选项列表
      "输出文件" 是该导出的输出文件

全局选项:
    -obfuscator=none|proguard|yguard
        对所选混淆器去混淆。默认值为"无"，对于其他值，必须指定mappingFile选项。
    -mappingfile=<file>
        所选混淆器的映射文件。
    -outputdir=<输出目录>
        当视图的输出文件是一个相对文件时，将使用基目录
    -ignoreerrors=true|false
       忽略当无法设置视图选项时发生的错误，并继续下一个视图。默认值为"false"，即，在第一个错误发生
时终止导出。
    -csvseparator=<分隔符>
        CSV导出的字段分隔符。默认为','。
    -bitmap=false|true
        在适当的情况下，代替SVG，为主要内容导出位图(bitmap)图片

可用的视图名称和选项:
* TelemetryHeap, TelemetryObjects, TelemetryThroughput, TelemetryGC,
  TelemetryClasses, TelemetryThreads, TelemetryCPU
  ??:
    -format=html|csv
        确定导出文件的输出格式。如果不存在，则将根据输出文件的扩展名确定导出格式。
    -minwidth=<像素值>
        graph的最小宽度（以像素为单位）。默认值为800。
    -minheight=<像素值>
        graph的最小高度（以像素为单位）。默认值为600。

* Bookmarks, ThreadMonitor, CurrentMonitorUsage, MonitorUsageHistory
  ??:
    -format=html|csv

* AllObjects
  ??:
    -format=html|csv
    -viewfilters=<以逗号分隔的列表>
        设置导出的视图过滤器。如果设置视图过滤器，则导出的视图将仅显示指定的包及其子包。
    -viewfiltermode=startswith|endswith|contains|equals
        设置视图过滤器模式。默认值为contains。
    -viewfilteroptions=casesensitive
        视图过滤器的布尔选项。默认情况下，没有设置任何选项。
    -aggregation=class|package|component
        选择导出的聚合级别。默认值为类。
    -expandpackages=true|false
```

```
        展开包聚合级别中的包节点以显示包含的类。默认值为“false”。对其他聚合级别和csv输出格式没有影
响。

* RecordedObjects
  ?AllObjects???????????:
    -liveness=live|gc|all
        选择导出的活性（Liveness）模式，即，显示活动对象，垃圾回收对象或所有。默认值是活动对象。

* AllocationTree
  ??:
    -format=html|xml
    -viewfilters=<以逗号分隔的列表>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive
    -aggregation=method|class|package|component
        选择导出的聚合级别。默认值为方法。
    -class=<fully qualified class name>
        指定应为其计算分配数据的类。如果为空，则显示所有类的分配。不能与package选项一起使用。
    -package=<fully qualified package name>
        指定要计算分配数据的包。如果为空，将显示所有包的分配。将 .** 附加到包名将递归选择包。不能与
 class 选项一起使用。
    -liveness=live|gc|all

* AllocationHotSpots
  ??:
    -format=html|csv|xml
    -viewfilters=<以逗号分隔的列表>
    -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
        选择是将非分析类单独显示还是将其添加到调用类。 默认值为单独显示非分析类。
    -valuesummation=self|total
        确定如何计算热点的时间。默认为“自身”。
    -expandbacktraces=true|false
        以HTML或XML格式展开回溯跟踪。默认值为“false”。

* ClassTracker
  ?TelemetryHeap???????????:
    -class
        被跟踪的类。 如果缺少，则导出第一个跟踪的类。

* CallTree
  ??:
    -format=html|xml
    -viewfilters=<以逗号分隔的列表>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive
    -aggregation=method|class|package|component
    -threadgroup=<线程组名称>
        选择要导出的线程组。如果同时指定thread，则仅在此线程组中搜索线程，否则将显示整个线程组。
    -thread=<线程名称>
        选择要导出的线程。默认情况下，将合并所有线程的调用树。
    -threadstatus=all|running|waiting|blocking|netio
        选择导出的线程状态。默认值是“running”。

* HotSpots
  ??:
    -format=html|csv|xml
    -viewfilters=<以逗号分隔的列表>
    -viewfiltermode=startswith|endswith|contains|equals
```

```
    -viewfilteroptions=casesensitive
    -aggregation=method|class|package|component
    -threadgroup=<线程组名称>
    -thread=<线程名称>
    -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=<以逗号分隔的列表>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive

* Complexity
  ??:
    -format=html|csv|properties
    -fit=best|constant|linear|quadratic|cubic|exponential|logarithmic|n_log_n
        应该导出的拟合。 默认值为“最佳”。 没有曲线拟合数据导出到CSV。
    -method=<method name>
        应该为其导出复杂度图的方法名称。 如果未给出，则将导出第一个方法。 否则，将导出以给定文本开头
的第一个方法。
    -width=<像素值>
    -height=<像素值>

* ThreadHistory
  ?TelemetryHeap??? ???????:
    -format=html

* MonitorUsageStatistics
  ??:
    -format=html|csv
    -type=monitors|threads|classes
        选择应为其计算Monitor统计信息的实体。默认值为“Monitor”。

* ProbeTimeLine
  ?ThreadHistory???????????:
    -probeid=<id>
        应该导出的探针的内部ID。 运行“ jpexport --listProbes”以列出所有可用的内置探针以及自定义
探针名称的说明。

* ProbeControlObjects
  ??:
    -probeid=<id>
    -format=html|csv

* ProbeCallTree
  ??:
    -probeid=<id>
    -format=html|xml
    -viewfilters=<以逗号分隔的列表>
    -viewfiltermode=startswith|endswith|contains|equals|wildcard|regex
    -viewfilteroptions=exclude,casesensitive
    -aggregation=method|class|package|component
    -threadgroup=<线程组名称>
    -thread=<线程名称>
    -threadstatus=all|running|waiting|blocking|netio
        选择导出的线程状态。默认值是“all”。

* ProbeHotSpots
  ?ProbeCallTree?????????????:
```

```
    -format=html|csv|xml
    -expandbacktraces=true|false

* ProbeTelemetry
  ?TelemetryHeap???????????:
    -probeid=<id>
    -telemetrygroup
        设置应该被导出的遥测组的基于一的索引。这是指您在探针遥测视图上方的下拉列表中看到的条目。 默
认值为“1”。

* ProbeEvents
  ??:
    -probeid=<id>
    -format=html|csv|xml

* ProbeTracker
  ?TelemetryHeap???????????:
    -probeid=<id>
    -index=<number>
        设置包含跟踪元素的下拉列表的从零开始的索引。默认值是0。
```

比较快照

可执行文件bin/jpcompare比较不同的快照[p. 123]并将其导出为HTML或机器可读格式。其-help输出如下所示， 同样省略了任何重复的解释。

```
用法: jpcompare "快照文件"[,"快照文件",...] [全局选项]
              "比较名称" [选项] "输出文件"
              "比较名称" [选项] "输出文件" ...

其中“快照文件”是具有以下扩展名之一的快照文件:
           .jps, .hprof, .hpz, .phd, .jfr
      "比较名称"是下边列出的比较名称之一
      [选项]是一个格式为“-选项=值”的一个选项列表
      "输出文件" 是该导出的输出文件

全局选项:
    -outputdir=<输出目录>
        当比较的输出文件是一个相对文件时，将使用基目录
    -ignoreerrors=true|false
        忽略当无法设置比较选项时发生的错误，并继续下一个比较。默认值为"false"，即，在第一个错误发生
时终止导出。
    -csvseparator=<分隔符>
        CSV导出的字段分隔符。默认为','。
    -bitmap=false|true
        在适当的情况下，代替SVG，为主要内容导出位图(bitmap)图片
    -sortbytime=false|true
        按修改时间对指定的快照文件进行排序。默认值为false。
    -listfile=<filename>
        读取包含快照文件路径的文件，每行一个快照文件。

可用的比较名称和选项:
* Objects
  ??:
    -format=html|csv
        确定导出文件的输出格式。如果不存在，则将根据输出文件的扩展名确定导出格式。
    -viewfilters=<以逗号分隔的列表>
        设置导出的视图过滤器。如果设置视图过滤器，则导出的视图将仅显示指定的包及其子包。
    -viewfiltermode=startswith|endswith|contains|equals
        设置视图过滤器模式。默认值为contains。
    -viewfilteroptions=casesensitive
        视图过滤器的布尔选项。默认情况下，没有设置任何选项。
```

```
    -aggregation=class|package|component
        选择导出的聚合级别。默认值为类。
    -liveness=live|gc|all
        选择导出的活性（Liveness）模式，即，显示活动对象，垃圾回收对象或所有。默认值是活动对象。
    -objects=recorded|heapwalker|all
        記録されたオブジェクト、ヒープウォーカー（heap walker）内のオブジェクト、またはすべてのオブジェクトダンプ（object
       dumps）からのオブジェクト数（object counts）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、HP
        ROF/PHDファイルの場合はヒープウォーカーです。
    -dumpselection=first|last|label
        用于计算比较的所有对象转储。默认值为最后一个值。
    -label
        如果将dumpselection设置为'label'，则应计算比较的标签名称。

* AllocationHotSpots
  ??:
    -format=html|csv
    -viewfilters=<以逗号分隔的列表>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive
    -aggregation=method|class|package|component
        选择导出的聚合级别。默认值为方法。
    -liveness=live|gc|all
    -unprofiledclasses=separately|addtocalling
        选择是将非分析类单独显示还是将其添加到调用类。 默认值为单独显示非分析类。
    -valuesummation=self|total
        确定如何计算热点的时间。默认为“自身”。
    -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 '.*' を、再帰的なパッケージには '.**'
        を追加して指定します。 指定一个带有通配符的软件包， 例如 'java.awt.*'。

* AllocationTree
  ??:
    -format=html|xml
    -viewfilters=<以逗号分隔的列表>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive
    -aggregation=method|class|package|component
    -liveness=live|gc|all
    -classselection

* HotSpots
  ??:
    -format=html|csv
    -viewfilters=<以逗号分隔的列表>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive
    -firstthreadselection
        计算特定线程或线程组的比较。 指定线程组，例如'group.*' 和特定线程组中的线程，如'group.thread'。反斜杠转义线程名称中
        的点。
    -secondthreadselection
        计算特定线程或线程组的比较。 只有在设置了"firstthreadselection"时才可用。如果为空，则使用与"firstthreads
       election"相同的值。 指定线程组，例如'group.*' 和特定线程组中的线程，如'group.thread'。反斜杠转义线程名称中
        的点。
    -threadstatus=all|running|waiting|blocking|netio
        选择导出的线程状态。默认值是“running”。
    -aggregation=method|class|package|component
    -differencecalculation=total|average
        为调用次数选择不同的计算方法。默认值是总次数。
```

```
    -unprofiledclasses=separately|addtocalling
    -valuesummation=self|total

* CallTree
  ??:
    -format=html|xml
    -viewfilters=<以逗号分隔的列表>
    -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=<像素值>
        graph的最小宽度（以像素为单位）。默认值为800。
    -minheight=<像素值>
        graph的最小高度（以像素为单位）。默认值为600。
    -valuetype=current|maximum|bookmark
        为每个快照计算的值的类型。 默认值为当前值。
    -bookmarkname
        如果 valuetype 被设置为 'bookmark'， 应当为其计算值的书签的名称。
    -measurements=maximum,free,used
        测量结果显示在比较图中。用逗号连接多个值。默认值是“used”。
    -memorytype=heap|nonheap
        应该分析的内存类型。 默认为“堆”。
    -memorypool
        如果需要分析一个指定内存池，可以用这个参数指定它的名称。默认为空，即没有指定任何内存池。

* TelemetryObjects
  ??:
    -format=html|csv
    -minwidth=<像素值>
    -minheight=<像素值>
    -valuetype=current|maximum|bookmark
    -bookmarkname
    -measurements=total,nonarrays,arrays
        测量结果显示在比较图中。用逗号连接多个值。默认值是“total”。

* TelemetryClasses
  ?TelemetryObjects??? ???????:
    -measurements=total,filtered,unfiltered

* TelemetryThreads
  ?TelemetryObjects??? ???????:
    -measurements=total,runnable,blocked,netio,waiting

* ProbeHotSpots
  ??:
    -format=html|csv
    -viewfilters=<以逗号分隔的列表>
    -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>
       应该导出的探针的内部ID。 运行“ jpexport --listProbes”以列出所有可用的内置探针以及自定义
```

```
探针名称的说明。

* ProbeCallTree
  ?ProbeHotSpots??? ???????:
    -format=html|xml

* ProbeTelemetry
  ?TelemetryObjects?????????????:
    -measurements
       比较图中显示的遥测组测量值以一为索引基数。用逗号连接多个值，例如 "1,2"。默认显示所有测量值。

    -probeid=<id>
    -telemetrygroup
        设置应该被导出的遥测组的基于一的索引。这是指您在探针遥测视图上方的下拉列表中看到的条目。 默
认值为“1”。
```

自动输出格式

大多数视图和比较支持多种输出格式。默认情况下，输出格式是从输出文件的扩展名推导出来的：

- **.html**
  视图或比较被导出为HTML文件。将创建一个名为jprofiler_images的目录，其中包含HTML页面中使用的图像。
- **.csv**
  数据被导出为CSV数据，其中第一行包含列名。
  使用Microsoft Excel时，CSV不是一种稳定的格式。Windows上的Microsoft Excel从区域设置中获取分隔符字符。JProfiler在使用逗号作为小数分隔符的区域中使用分号作为分隔符，在使用点作为小数分隔符的区域中使用逗号。如果您需要覆盖CSV分隔符字符，可以通过设置全局csvseparator选项来实现。
- **.xml**
  数据被导出为XML。数据格式是自描述的。

如果您想使用不同的扩展名，可以使用format选项来覆盖输出格式的选择。

分析快照

如果生成的快照中包含堆转储，您可以使用bin/jpanalyze可执行文件来提前准备堆转储分析[p. 75]。然后在JProfiler GUI中打开快照将非常快速。工具的使用信息如下所示：

```
用法: jpanalyze [选项] "快照文件" ["快照文件" ...]

其中“快照文件”是具有以下扩展名之一的快照文件:
           .jps, .hprof, .hpz, .phd, .jfr
      [选项]是一个格式为“-选项=值”的一个选项列表

选项:
    -obfuscator=none|proguard|yguard
        对所选混淆器去混淆。默认值为"无"，对于其他值，必须指定mappingFile选项。
    -mappingfile=<file>
        所选混淆器的映射文件。
    -removeunreferenced=true|false
        如果未引用或弱引用的对象应被移除。
    -retained=true|false
        计算保留大小（最大对象）。删除未引用将设置为true。
```

```
    -retainsoft=true|false
        如果删除了未引用的对象，则指定是否应保留软引用。
    -retainweak=true|false
        如果删除了未引用的对象，则指定是否应保留弱引用。
    -retainphantom=true|false
        如果删除了未引用的对象，则指定是否应保留虚引用。
    -retainfinalizer=true|false
        如果删除了未引用的对象，则指定是否应保留Finalizer引用。
```

removeUnreferenced、retained以及所有retain*命令行选项对应于堆行走器选项对话框中的选项。

G.3 Gradle Tasks（Gradle任务）

JProfiler支持通过Gradle的特殊任务进行分析。此外，JProfiler提供了一些 用于处理快照的命令行可执行文件 [p. 234] ，这些文件有相应的Gradle任务。

使用Gradle任务

要在Gradle构建文件中使用JProfiler Gradle任务，可以使用`plugins`块

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

如果您不想为此目的使用Gradle插件库，Gradle插件分发在文件`bin/gradle.jar`中。

接下来，您需要告诉JProfiler Gradle插件JProfiler的安装位置。

```
jprofiler {
    installDir = file('/path/to/jprofiler/home')
}
```

从Gradle进行分析

使用类型为`com.jprofiler.gradle.JavaProfile`的任务，您可以分析任何Java进程。 该类扩展了Gradle内置的`JavaExec`，因此您可以使用相同的参数来配置进程。对于分析测试，使用类型为`com.jprofiler.gradle.TestProfile`的任务，该任务扩展了Gradle的`Test`任务。

在没有任何进一步配置的情况下，这两个任务都会启动一个交互式分析会话，其中分析代理在默认端口8849上等待来自JProfiler GUI的连接。 对于离线分析，您需要添加一些在下表中显示的属性。

Attribute（属性）	Description（描述）	Required（必需）
offline	分析运行是否应在离线模式下进行。	否，`offline`和`nowait`不能同时为`true`。
nowait	分析是否应立即开始，或者被分析的JVM是否应等待来自JProfiler GUI的连接。	
sessionId	定义应从中获取分析设置的会话ID。如果既未设置`nowait`也未设置offline，则无效， 因为在这种情况下，分析会话是在GUI中选择的。	如果以下条件之一成立，则必需： • `offline`已设置 • 对于1.5 JVM，`nowait`已设置
configFile	定义应从中读取分析设置的配置文件。	否
port	定义分析代理应监听来自JProfilerGUI连接的端口号。这必须与远程会话配置中配置的端口相同。如果未设置或为零，将使用默认端口（8849）。如果设置了`offline`，则无效， 因为在这种情况下没有来自GUI的连接。	否

Attribute（属性）	Description（描述）	Required（必需）
debugOptions	如果您想传递任何额外的库参数以进行调优或调试，可以使用此属性。	否

下面给出了一个使用包含项目编译的主方法分析Java类的示例：

```
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')
}
```

您可以在api/samples/offline示例项目中看到此任务的可运行示例。与标准JavaExec任务不同， JavaProfile任务还可以通过调用createProcess()在后台启动。有关此功能的演示， 请参见api/samples/mbean示例项目。

如果您需要分析所需的VM参数，com.jprofiler.gradle.SetAgentpathProperty任务会将其分配给一个属性，该属性的名称由propertyName属性配置。应用JProfiler插件会自动向您的项目添加一个名为 setAgentPathProperty的此类型任务。要获取在上一个示例中使用的VM参数，您只需添加

```
setAgentPathProperty {
    propertyName = 'profilingVmParameter'
    offline = true
    sessionId = 80
    configFile = file('path/to/jprofiler_config.xml')
}
```

到您的项目中，并将setAgentPathProperty作为依赖项添加到其他任务中。然后，您可以在该任务的执行阶段使用 项目属性profilingVmParameter。在将属性分配给其他任务属性时，请将其用doFirst {...} 代码块包围，以确保您处于Gradle执行阶段而不是配置阶段。

从快照导出数据

com.jprofiler.gradle.Export任务可用于从保存的快照中导出视图，并复制bin/jpexport命令行工具 [p. 234]的参数。 它支持以下属性：

Attribute（属性）	Description（描述）	Required（必需）
snapshotFile	快照文件的路径。这必须是一个具有.jps扩展名的文件。	是
ignoreErrors	忽略在无法设置视图选项时发生的错误，并继续下一个视图。默认值为false，表示在第一个错误发生时终止导出。	否
csvSeparator	CSV导出的字段分隔符字符。默认为","。	否

Attribute（属性）	Description（描述）	Required（必需）
obfuscator	为所选混淆器反混淆类和方法名称。默认为"none"，对于其他值，必须指定mappingFile选项。 其中之一none、proguard或yguard。	否
mappingFile	所选混淆器的映射文件。只有在指定obfuscator属性时才能设置。	仅当指定obfuscator时

在导出任务中，您调用views方法，并传递一个闭包，在其中可以多次调用view(name, file[, options])。每次调用view都会生成一个输出文件。name参数是视图名称。有关可用视图名称的列表，请参见jpexport命令行可执行文件[p. 234]的帮助页面。参数file是输出文件，可以是绝对文件或相对于项目的文件。最后，可选的options参数是 一个包含所选视图导出选项的映射。

使用导出任务的示例如下：

```
task export(type: com.jprofiler.gradle.Export) {
    snapshotFile = file('snapshot.jps')
    views {
        view('CallTree', 'callTree.html')
        view('HotSpots', 'hotSpots.html',
            [threadStatus: 'all', expandBacktraces: 'true'])
    }
}
```

比较快照

像bin/jpcompare命令行工具[p. 234]一样，com.jprofiler.gradle.Compare任务可以比较两个或多个快照。 它的属性有：

Attribute（属性）	Description（描述）	Required（必需）
snapshotFiles	应该比较的快照文件。您可以传递任何Iterable，其中包含Gradle解析为文件集合的对象。	是
sortByTime	如果设置为true，则所有提供的快照文件将按其文件修改时间排序，否则它们将按照在snapshotFiles属性中指定的顺序进行比较。	否
ignoreErrors	忽略在无法设置比较选项时发生的错误，并继续下一个比较。默认值为false，表示在第一个错误发生时终止导出。	否

就像为Export任务定义导出的视图一样，Compare任务有一个comparisons方法， 其中嵌套调用comparison(name, file[, options])定义要执行的比较。可用比较名称的列表可在jpcompare命令行可执行文件 [p. 234]的帮助页面上找到。

使用比较任务的示例如下：

```
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'])
    }
}
```

或者，如果您想为多个快照创建遥测比较：

```
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'])
    }
}
```

分析堆快照

Gradle任务`com.jprofiler.gradle.Analyze`具有与`bin/jpanalyze`命令行工具[p. 234]相同的功能。

该任务具有与`Compare`任务类似的`snapshotFiles`属性，用于指定处理的快照， 以及与`Export`任务类似的`obfuscator`和`mappingfile`属性用于反混淆。 属性`removeUnreferenced`、`retainSoft`、`retainWeak`、 `retainPhantom`、`retainFinalizer`和`retained`对应于命令行工具的参数。

使用`Analyze`任务的示例如下：

```
task analyze(type: com.jprofiler.gradle.Analyze) {
    snapshotFiles = fileTree(dir: 'snapshots', include: '*.jps')
    retainWeak = true
    obfuscator = 'proguard'
    mappingFile = file('obfuscation.txt')
}
```

G.4 Ant Tasks

JProfiler 提供的 Ant[1] 任务与 Gradle 任务非常相似。本章重点介绍与 Gradle 任务的区别，并展示每个 Ant 任务的示例。

所有 Ant 任务都包含在 `bin/ant.jar` 存档中。为了使任务可用于 Ant，您必须首先插入一个 `taskdef` 元素，告诉 Ant 在哪里可以找到任务定义。下面的所有示例都包含该 taskdef。它在每个构建文件中只能出现一次，并且可以出现在项目元素下的任何级别。

无法将 `ant.jar` 存档复制到 Ant 分发版的 `lib` 文件夹中，您必须在任务定义中引用 JProfiler 的完整安装。

Profiling from Ant

`com.jprofiler.ant.ProfileTask` 派生自内置的 `Java` 任务，并支持其所有属性和嵌套元素。附加属性与 `ProfileJava` Gradle 任务 [p. 242] 相同。Ant 属性不区分大小写，通常以小写书写。

```
<taskdef name="profile"
         classname="com.jprofiler.ant.ProfileTask"
         classpath="<path to JProfiler installation>/bin/ant.jar"/>

<target name="profile">
  <profile classname="MyMainClass" offline="true" sessionid="80">
    <classpath>
      <fileset dir="lib" includes="*.jar" />
    </classpath>
  </profile>
</target>
```

Exporting data from snapshots

使用 `com.jprofiler.ant.ExportTask`，您可以从快照中导出视图，就像使用 `Export` Gradle 任务 [p. 242] 一样。视图的指定方式与 Gradle 任务不同：它们直接嵌套在任务元素下，选项通过嵌套的 `option` 元素指定。

```
<taskdef name="export"
         classname="com.jprofiler.ant.ExportTask"
         classpath="<path to JProfiler installation>/bin/ant.jar"/>

<target name="export">
  <export snapshotfile="snapshots/test.jps">
    <view name="CallTree" file="calltree.html"/>
    <view name="HotSpots" file="hotspots.html">
      <option name="expandbacktraces" value="true"/>
      <option name="aggregation" value="class"/>
    </view>
  </export>
</target>
```

Comparing snapshots

`com.jprofiler.ant.CompareTask` 对应于 `Compare` Gradle 任务，并在两个或多个快照之间进行比较。与 `com.jprofiler.ant.ExportTask` 一样，比较直接嵌套在元素下，选项为每个 `comparison` 元素嵌套。快照文件通过嵌套文件集指定。

[1] http://ant.apache.org

```
<taskdef name="compare"
         classname="com.jprofiler.ant.CompareTask"
         classpath="<path to JProfiler installation>/bin/ant.jar"/>

<target name="compare">
  <compare sortbytime="true">
    <fileset dir="snapshots">
      <include name="*.jps" />
    </fileset>
    <comparison name="TelemetryHeap" file="heap.html"/>
    <comparison name="TelemetryThreads" file="threads.html">
      <option name="measurements" value="inactive,active"/>
      <option name="valuetype" value="bookmark"/>
      <option name="bookmarkname" value="test"/>
    </comparison>
  </compare>
</target>
```

Analyzing heap snapshots

与 Analyze Gradle 任务类似，Ant 的等效任务 com.jprofiler.ant.AnalyzeTask 准备在已通过离线分析保存的快照中进行堆快照分析，以便在GUI中更快地访问。要处理的快照通过嵌套文件集指定。

```
<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>
```