# Oracle® Solaris Studio 12.4：性能分析器

文件号码 E57219
2015 年 1 月

ORACLE

# 目录

# 使用本文档

- **概述** – 介绍 Oracle Solaris Studio 软件中的性能分析工具。收集器和性能分析器这一对工具用于收集各种性能数据，并在函数、源代码行和指令级将这些数据与程序结构相关联。收集的性能数据包括时钟分析统计信息、硬件计数器分析以及各种调用的跟踪信息。
- **目标读者** – 应用程序开发人员、开发者、架构师、支持工程师
- **必备知识** – 编程经验、程序/软件开发测试、生成和编译软件产品的能力

## 产品文档库

产品文档库位于 http://docs.oracle.com/cd/E37069_01。

系统要求和已知问题在《Oracle Solaris Studio 12.4：发行说明 》中介绍。

## 反馈

可以在 http://www.oracle.com/goto/docfeedback 上提供有关本文档的反馈。

♦ ♦ ♦ 第 1 章

# 1 性能分析器概览

开发高性能的应用程序需要将编译器特性、已优化的函数库和性能分析工具整合在一起。本性能分析器手册介绍了一些工具，这些工具有助于您评估代码的性能、识别潜在的性能问题并找到出现这些问题的代码部分。

本章包含以下主题：



## 性能分析工具

本手册介绍了收集器和性能分析器这一对工具，您可以使用它们来收集和分析应用程序的性能数据。该手册还介绍了 `er_print` 实用程序，一种用于以文本格式显示和分析所收集性能数据的命令行工具。性能分析器和 `er_print` 实用程序显示的数据大体相同，但使用不同的用户界面。

收集器和性能分析器设计旨在供任何软件开发者使用，即使性能调节并非开发者的主要职责。与常用的分析工具 `prof` 和 `gprof` 相比，这些工具提供了更加灵活、详细和准确的分析，并且不会产生 `gprof` 中的归属误差。

收集器和性能分析器工具有助于回答以下各种问题：

- 程序消耗的可用资源有多少？
- 最消耗资源的是哪些函数或装入对象？
- 消耗资源的是哪些源代码行和指令？
- 程序在执行过程中如何出现这种问题？
- 函数或装入对象消耗的是哪些资源？

### 收集器工具

收集器工具收集性能数据：

- 使用称为分析的统计方法，该方法可基于时钟触发器或硬件性能计数器的溢出

- 通过跟踪线程同步调用、内存分配和取消分配调用、IO 调用以及消息传递接口 (Message Passing Interface, MPI) 调用
- 作为系统和进程的汇总数据

在 Oracle Solaris 平台上，时钟分析数据包括微状态计数数据。所有记录的分析和跟踪事件包括调用堆栈以及线程和 CPU ID。

收集器可以收集 C、C++ 和 Fortran 程序的各种数据，也可以收集用 Java™ 编程语言编写的应用程序的分析数据。此外还可以收集动态生成的函数及子孙进程的数据。有关收集的数据的信息，请参见第 2 章 性能数据；有关收集器的详细信息，请参见第 3 章 收集性能数据。通过性能分析器、`collect` 命令和 `dbx collector` 命令分析应用程序时，将运行收集器。

# 性能分析器工具

性能分析器显示收集器记录的数据，以便您检查这些信息。性能分析器处理数据并显示程序、函数、源代码行和指令级别的各种性能度量。这些度量分为以下组：

- 时钟分析度量
- 硬件计数器分析度量
- 同步等待跟踪度量
- I/O 跟踪度量
- 堆跟踪度量
- MPI 跟踪度量
- 抽样点

性能分析器的 "Timeline"（时间线）视图可以按图形形式将原始数据显示为时间函数。"Timeline"（时间线）视图将记录的事件和抽样点的图表显示为时间函数。数据显示在水平栏中。

性能分析器还可以显示目标程序的数据空间中结构的性能度量，以及内存子系统的结构组件的性能度量。此数据是硬件计数器度量的扩展。

在任何受支持的体系结构上记录的实验都可以由在相同或任何其他受支持的体系结构上运行的性能分析器显示。例如，可以在应用程序在 Oracle Solaris SPARC 服务器上运行时对其进行分析，然后使用在 Linux 计算机上运行的性能分析器查看生成的实验。

可以在具有 Java 的任何系统上安装性能分析器的客户机版本，即远程性能分析器。您可以运行此远程性能分析器，连接到安装了完整 Oracle Solaris Studio 产品的服务器，并以远程方式查看实验。有关更多信息，请参见“远程使用性能分析器” [112]。

性能分析器由 Oracle Solaris Studio 分析套件中的其他工具使用：

- 线程分析器使用它检查线程分析实验。使用单独的命令 `tha` 启动性能分析器将显示一个专门视图，其中显示实验中的的数据争用和死锁，您可专门生成此视图来检查这些类型的数据。

  《Oracle Solaris Studio 12.4：线程分析器用户指南》介绍如何使用线程分析器。

- `uncover` 代码覆盖实用程序使用性能分析器显示 "Functions"（函数）、"Source"（源）、"Disassembly"（反汇编）和 "Inst-Freq"（指令频率）数据视图中的覆盖数据。有关更多信息，请参见《Oracle Solaris Studio 12.4：Discover 和 Uncover 用户指南》。

有关使用工具的详细信息，请参见第 4 章 性能分析器工具以及性能分析器中的 "Help"（帮助）菜单。

第 5 章 `er_print` 命令行性能分析工具介绍如何使用 `er_print` 命令行界面来分析收集器收集的数据。

第 6 章 了解性能分析器及其数据讨论了一些与了解性能分析器及其数据有关的主题，包括：数据收集的工作原理、解释性能度量、调用堆栈和程序执行。

第 7 章 了解带注释的源代码和反汇编数据介绍如何了解带注释的源代码和反汇编代码，提供了有关性能分析器显示的不同类型的索引行和编译器注释的解释。本章还介绍 `er_src` 命令行实用程序，可以使用该实用程序来查看包含编译器注释但不包含性能数据的带注释的源代码列表和反汇编代码列表。

第 8 章 操作实验介绍了如何复制、移动和删除实验；将标签添加到实验以及归档和导出实验。

第 9 章 内核分析介绍了如何在 Oracle Solaris 操作系统运行负载时使用 Oracle Solaris Studio 性能工具分析内核。

---

注 - 可以从 Oracle Solaris Studio 12.4 样例应用程序页面的样例应用程序 zip 文件中下载性能分析器的演示代码，网址为 http://www.oracle.com/technetwork/server-storage/solarisstudio/downloads/solaris-studio-12-4-samples-2333090.html。

接受许可并下载后，可以将 zip 文件提取到所选择的目录中。样例应用程序位于 `SolarisStudioSampleApplications` 目录的 `PerformanceAnalyzer` 子目录中。有关如何在性能分析器中使用样例代码的信息，请参见《Oracle Solaris Studio 12.4：性能分析器教程》。

---

## `er_print` 实用程序

`er_print` 实用程序以纯文本形式显示性能分析器提供的所有显示内容，但 "Timeline"（时间线）显示、"MPI Timeline"（MPI 时间线）显示和 "MPI Chart"（MPI 图表）显示除外。这些显示本身都是图形形式，无法将其表示为文本。

# 性能分析器窗口

本节简要说明性能分析器窗口布局。有关下面讨论的功能和特性的更多信息，请参见第 4 章 性能分析器工具和 "Help"（帮助）菜单。

启动性能分析器时，可以通过 "Welcome"（欢迎）页轻松地开始应用程序的分析（支持多种不同方式）、查看最近的实验、比较实验以及定位到文档。

当打开一个实验时，"Overview"（概述）视图显示所记录数据的要点以及可用的度量集。可以选择要检查哪些度量。

性能分析器围绕着数据视图进行组织，这些数据视图可以通过左侧导航栏中的按钮访问。每个视图都针对分析的应用程序从不同的角度显示性能度量。数据视图是相互连接的，在一个视图中选择了某个函数时，其他数据视图也会更新以重点显示有关所选函数的信息。

在大多数数据视图中，通过从上下文菜单中选择过滤器或单击过滤器按钮，可以使用性能分析器强大的过滤技术深入研究性能问题。

有关每个视图的更多信息，请参见“性能分析器视图” [88]。

可以使用键盘和鼠标在性能分析器中导航。

♦ ♦ ♦ 第 2 章

# 2 性能数据

性能工具在程序运行时记录有关特定事件的数据，然后将这些数据转换为程序性能的测量值，称为度量。度量可根据函数、源代码行和指令来显示。

本章介绍了通过性能工具收集的数据、如何处理和显示这些数据，以及如何使用这些数据进行性能分析。由于收集性能数据的工具有很多种，因此使用术语收集器来指代这些工具中的任何一种。同样，由于分析性能数据的工具也有很多种，因此使用术语分析工具来指代这些工具中的任何一种。

本章包含以下主题。



有关收集和存储性能数据的信息，请参见第 3 章 收集性能数据。

## 收集器收集的数据

收集器使用多种方法收集各种数据：

- 可通过按固定的间隔记录分析事件来收集**分析数据**。该间隔可以是使用系统时钟获取的时间间隔，也可以是特定类型硬件事件的数目。间隔时间结束时，会向系统传送一个信号，并在下一个间隔记录数据。
- 可通过在各种系统函数和库函数上插入包装函数来收集**跟踪数据**，以便拦截对函数的调用，并记录有关调用的数据。
- 可通过调用各种系统例程获取全局信息来收集**抽样数据**。
- 为可执行文件以及动态打开或静态链接到可执行文件并经过检测的任何共享对象收集**函数和指令计数数据**。记录函数和指令的执行次数。
- 收集**线程分析数据**以支持线程分析器。

分析数据和跟踪数据都包含有关特定事件的信息，并且这两种类型的数据都会转换为性能度量。抽样数据不会转换为度量，而是用于提供标记，这些标记可用于将程序执行划分为很多时间段。通过抽样数据，可以了解该时间段内程序执行的总体情况。

每个分析事件或跟踪事件收集的数据包都包含以下信息：

- 标识数据的数据包头。

- 高精度的时间戳。
- 线程 ID。
- 轻量级进程 (lightweight process, LWP) ID。
- 处理器 (CPU) ID，在操作系统中可用时。
- 调用堆栈的副本。对于 Java 程序，将记录两个调用堆栈：计算机调用堆栈和 Java 调用堆栈。
- 对于 OpenMP 程序，还会收集当前并行区域的标识符和 OpenMP 状态。

有关线程和轻量级进程的更多信息，请参见第 6 章 了解性能分析器及其数据。

除了通用数据外，每个事件特定的数据包还包含特定于数据类型的信息。

数据类型及其使用方法将在下列子节中介绍：


- “时钟分析数据” [20]
- “硬件计数器分析数据” [23]
- “同步等待跟踪数据” [26]
- “堆跟踪（内存分配）数据” [27]
- “I/O 跟踪数据” [28]
- “MPI 跟踪数据” [29]
- “抽样数据” [28]


# 时钟分析数据

进行时钟分析时，收集的数据取决于操作系统所提供的信息。

## Oracle Solaris 下的时钟分析

在 Oracle Solaris 下的时钟分析中，每个线程的状态都按固定的时间间隔存储。该时间间隔称为分析间隔。使用分析间隔的精度将收集的数据转换为每个状态所用的时间。

缺省分析间隔约为 10 毫秒 (10 ms)。可以指定高精度分析间隔（大约为 1 毫秒）和低精度分析间隔（大约为 100 毫秒）。如果操作系统允许，也可以指定定制间隔。不带任何其他参数运行 `collect -h` 可输出系统所允许的范围和精度。

下表显示当实验包含时钟分析数据时，性能分析器和 `er_print` 可显示的性能度量。请注意，来自所有线程的度量加在了一起。

表 2-1 Oracle Solaris 上时钟分析的计时度量

| 度量 | 定义 |
| --- | --- |
| Total thread time（总线程时间） | 线程在所有状态花费的时间总和。 |

| 度量 | 定义 |
| --- | --- |
| Total CPU time（CPU 总时间） | 在 CPU 中按用户、内核或陷阱模式运行所用的线程时间。 |
| User CPU time（用户 CPU 时间） | 以用户模式在 CPU 中运行所花费的线程时间。 |
| System CPU time（系统 CPU 时间） | 以内核模式在 CPU 中运行所花费的线程时间。 |
| Trap CPU time（自陷 CPU 时间） | 以陷阱模式在 CPU 中运行所花费的线程时间。 |
| User lock time（用户锁定时间） | 等待同步锁定所用的线程时间。 |
| Data page fault time（数据缺页时间） | 等待数据页所用的线程时间。 |
| Text page fault time（文本缺页时间） | 等待文本页所用的线程时间。 |
| Kernel page fault time（内核缺页时间） | 等待内核页所用的线程时间。 |
| Stopped time（停止时间） | 由于停止而花费的线程时间。 |
| Wait CPU time（等待 CPU 时间） | 等待 CPU 所用的线程时间。 |
| Sleep time（休眠时间） | 由于休眠而花费的线程时间 |

计时度量按多种类别说明程序消耗时间的位置，并且可用于改善程序的性能。

- 高用户 CPU 时间说明程序处理大部分工作的位置。可使用它来查找重新设计算法后可能受益最多的程序部分。
- 高系统 CPU 时间说明程序在对系统例程的调用中消耗了大量时间。
- 高等待 CPU 时间说明准备运行的线程数比可用的 CPU 多，或其他进程正在使用 CPU。
- 高用户锁定时间说明线程无法获得其请求的锁定。
- 高文本缺页时间意味着链接程序要求的代码会在内存中进行组织，所以很多调用或分支会导致装入新的页面。
- 高数据缺页时间表明对数据的访问会导致新的页面被装入。重新组织程序的数据结构或算法可以修复此问题。

## Linux 下的时钟分析

在 Linux 平台上，只能将时钟数据显示为 CPU 总时间。Linux CPU 时间是用户 CPU 时间和系统 CPU 时间的总和。

## 针对 OpenMP 程序的时钟分析

如果在 OpenMP 程序上执行时钟分析，还提供其他度量：Master Thread Time（主线程时间）、OpenMP Work Time（OpenMP 工作时间）和 OpenMP Wait Time（OpenMP 等待时间）。

- 在 Oracle Solaris 上，"Master Thread Time"（主线程时间）是主线程消耗的总时间，它与挂钟时间相对应。该度量在 Linux 上不可用。

- 在 Oracle Solaris 上，以串行或并行方式执行工作时，"OpenMP Work Time"（OpenMP 工作时间）会累计。在以下情况下，"OpenMP Wait Time"（OpenMP 等待时间）会累计：OpenMP 运行时正在等待进行同步时、该等待正在使用 CPU 时间或正在休眠时，以及正在以并行方式执行工作，但未在 CPU 上安排线程时。
- 在 Linux 操作系统上，仅当进程在用户模式或系统模式下处于活动状态时，"OpenMP Work Time"（OpenMP 工作时间）和 "OpenMP Wait Time"（OpenMP 等待时间）才会累计。除非您已指定 OpenMP 应执行忙等待，否则，Linux 上的 "OpenMP Wait Time"（OpenMP 等待时间）没有用处。

OpenMP 程序的数据可以在三种查看模式中的任一种模式下显示。在用户模式下，从线程按实际从主线程克隆的方式显示，并使其调用堆栈匹配主线程的调用堆栈。调用堆栈中来自 OpenMP 运行时代码 (`libmtsk.so`) 的帧会被禁止。在专家用户模式下，主从线程的显示方式不同，编译器生成的显式函数可见，来自 OpenMP 运行时代码 (`libmtsk.so`) 的框架将被禁止。对于计算机模式，将显示实际的本机堆栈。

## 针对 Oracle Solaris 内核的时钟分析

`er_kernel` 实用程序可以收集有关 Oracle Solaris 内核的基于时钟的分析数据。要分析内核，可从命令行直接运行 `er_kernel` 实用程序，或从性能分析器的 "File"（文件）菜单中选择 "Profile Kernel"（分析内核）。

`er_kernel` 实用程序捕获内核分析数据，并将数据记录为性能分析器实验，其格式与 `collect` 实用程序在用户程序上创建的实验格式相同。实验可以由 `er_print` 实用程序或性能分析器进行处理。内核实验可以显示函数数据、调用方-被调用方数据、指令级数据和时间线，但是不能显示源代码行数据（因为大多数 Oracle Solaris 模块不包含行号表）。

`er_kernel` 还可以对正在运行的用户有权限的任何进程记录用户级实验。此类实验类似于 `collect` 创建的实验，但它只包含 "User CPU Time"（用户 CPU 时间）和 "System CPU Time"（系统 CPU 时间）数据，不支持 Java 或 OpenMP 分析。

有关更多信息，请参见第 9 章 内核分析。

## 针对 MPI 程序的时钟分析

可以在用 Oracle Message Passing Toolkit（以前称为 Sun HPC ClusterTools）运行的 MPI 实验中收集时钟分析数据。Oracle Message Passing Toolkit 必须至少为版本 8.1。

Oracle Message Passing Toolkit 已集成在 Oracle Solaris 11 发行版中。如果系统中安装了此工具包，您可以在 `/usr/openmpi` 中找到它。如果您的 Oracle Solaris 11 系统中尚未安装此工具包，当您为系统配置了软件包系统信息库时，可以使用命令 `pkg search openmpi` 搜索该工具包。有关在 Oracle Solaris 11 中安装软件的更多信息，请参见《在 *Oracle Solaris 11* 中添加和更新软件》。

在 MPI 实验中收集时钟分析数据时，还可以查看两个其他度量：

- MPI Work Time（MPI 工作时间），该度量将在进程正在 MPI 运行时内执行工作（如处理请求或消息）时累计。
- MPI Wait Time（MPI 等待时间），该度量将在进程正在 MPI 运行时内等待事件、缓冲区或消息时累计

在 Oracle Solaris 上，以串行或并行方式执行工作时，MPI 工作时间会累计。在以下情况下，MPI 等待时间会累计：MPI 运行时正在等待进行同步时、该等待正在使用 CPU 时间或正在休眠时，以及正在以并行方式执行工作，但未在 CPU 上调度线程时。

在 Linux 上，仅当进程在用户模式或系统模式下处于活动状态时，MPI 工作时间和 MPI 等待时间才会累计。除非您已指定 MPI 应执行忙等待，否则，Linux 上的 MPI 等待时间没有用处。

---

注 - 如果要在 Linux 上使用 Oracle Message Passing Toolkit 8.2 或 8.2.1，可能需要解决方法。版本 8.1 或 8.2.1c 不需要解决方法，或者如果要使用 Oracle Solaris Studio 编译器，则任何版本都不需要解决方法。

Oracle Message Passing Toolkit 版本号由安装路径指定，例如 `/opt/SUNWhpc/HPC8.2.1`，或者，您可以按照如下所示键入 `mpirun –V` 查看输出，其中版本以斜体表示：

`mpirun (Open MPI) 1.3.4r22104-`*`ct8.2.1`*`-b09d-r70`

如果您的应用程序是使用 GNU 或 Intel 编译器编译的，并且要对 MPI 使用 Oracle Message Passing Toolkit 8.2 或 8.2.1，则要获取 MPI 状态数据，必须使用 `-WI` 和 `--enable-new-dtags` 选项和 Oracle Message Passing Toolkit `link` 命令。这些选项将使可执行文件定义 `RPATH` 及 `RUNPATH`，从而可使用 `LD_LIBRARY_PATH` 环境变量启用 MPI 状态库。

---

# 硬件计数器分析数据

硬件计数器可跟踪诸如高速缓存未命中次数、高速缓存停止周期、浮点运算、分支误预测、CPU 周期以及执行指令之类的事件。在硬件计数器分析中，当运行线程的 CPU 的指定硬件计数器发生溢出时，收集器将记录一个分析数据包。计数器将重置并继续进行计数。分析数据包中包括溢出值和计数器类型。

各种处理器芯片系列支持同时存在二到十八个硬件计数器寄存器。收集器可收集一个或多个寄存器上的数据。对于每个寄存器，可以选择监视溢出的计数器的类型，并为计数器设置溢出值。有些硬件计数器可以使用任意寄存器，而有些计数器仅可以使用特定的寄存器。因此，在一个实验中并非可以选择所有的硬件计数器组合。

硬件计数器分析还可以在内核上执行：通过性能分析器和 `er_kernel` 实用程序执行。有关更多信息，请参见第 9 章 内核分析。

硬件计数器分析数据由性能分析器转换为计数度量。对于以循环方式计数的计数器，所报告的度量会转换为次数；而对于不以循环方式计数的计数器，所报告的度量为事件计数。在具有多个 CPU 的计算机上，用于转换度量的时钟频率为各个 CPU 时钟频率的调和平均数。因为每种类型的处理器都有其自己的一组硬件计数器，并且硬件计数器的数目庞大，因此，此处未列出硬件计数器的度量。“硬件计数器列表” [24] 讲述如何找出可用的硬件计数器。

如果收集两个特定的计数器 "cycles" 和 "insts"，则可以使用两个额外的度量，即 "CPI" 和 "IPC"，分别表示每指令周期数和每周期指令数。它们始终以比率（而非时间、计数或百分比）的形式显示。高 CPI 值或低 IPC 值指示代码在计算机中运行效率低；相反，低 CPI 值或高 IPC 值指示代码在管道中运行效率高。

硬件计数器的一个用途是可诊断进出 CPU 的信息流问题。例如，高速缓存未命中次数计数较高表明，重新组织程序的结构来改进数据或文本的位置或提高高速缓存的重用率可以改善程序性能。

某些硬件计数器与其他计数器相互关联。例如，分支误预测和指令高速缓存未命中次数通常是相关的，因为分支误预测会导致将错误的指令装入到指令高速缓存中。这些指令必须替换为正确的指令。这种替换会导致指令高速缓存未命中，或指令转换后备缓冲器 (instruction translation lookaside buffer, ITLB) 未命中，或甚至缺页。

对于许多硬件计数器，经常会用导致溢出事件的指令之后的一条或多条指令报告溢出。这种情况称为“失控”(skid)，它会使计数器溢出分析数据难以解释。

在最新的 SPARC 处理器上，某些基于内存的计数器中断是精确的，并使用 PC（program counter，程序计数器）和触发事件的有效地址来报告。此类计数器的事件类型后面用字 `precise` 表示。缺省情况下将捕获这些计数器的内存空间和数据空间数据。有关更多信息，请参见“数据空间分析和内存空间分析” [165]。

## 硬件计数器列表

由于硬件计数器是特定于处理器的，因此可以选用的计数器取决于所使用的处理器。性能工具为许多可能常用的计数器提供了别名。您可以确定用于分析当前计算机的最大硬件计数器定义数量，并在当前计算机上运行不带任何其他参数的 `collect -h`，以查看可用硬件计数器及缺省计数器集的完整列表。

如果处理器和系统支持硬件计数器分析，则 `collect -h` 命令会输出两个包含有关硬件计数器信息的列表。第一个列表包含硬件计数器别名（这些别名是常用名称）。第二个列表包含原始硬件计数器。如果性能计数器子系统和 `collect` 命令都没有特定系统上的计数器的名称，则这些列表将为空。但是，在大多数情况下，可以用数值指定计数器。

以下示例显示计数器列表中的条目。有别名的计数器将首先显示在列表中，然后是原始硬件计数器列表。该示例中的每一行输出都按打印格式显示。

```
Aliased HW counters available for profiling:
    cycles[/{0|1|2|3}],<interval> (`CPU Cycles', alias for Cycles_user; CPU-cycles)
```

```
    insts[/{0|1|2|3}],<interval> (`Instructions Executed', alias for Instr_all; events)
    loads[/{0|1|2|3}],<interval>
     (`Load Instructions', alias for Instr_ld; precise load-store events)
    stores[/{0|1|2|3}],<interval>
     (`Store Instructions', alias for Instr_st; precise load-store events)
    dcm[/{0|1|2|3}],<interval>
     (`L1 D-cache Misses', alias for DC_miss_nospec; precise load-store events)
    l2l3dh[/{0|1|2|3}],<interval>
     (`L2 or L3 D-cache Hits', alias for DC_miss_L2_L3_hit_nospec; precise load-store events)
    l3m[/{0|1|2|3}],<interval>
     (`L3 D-cache Misses', alias for DC_miss_remote_L3_hit_nospec~emask=0x6; precise load-
store events)
    l3m_spec[/{0|1|2|3}],<interval>
     (`L3 D-cache Misses incl. Speculative', alias for DC_miss_remote_L3_hit~emask=0x6;
 events)
.
.
.
 Raw HW counters available for profiling:
    Sel_pipe_drain_cycles[/{0|1|2|3}],<interval> (CPU-cycles)
    Sel_0_wait[/{0|1|2|3}],<interval> (CPU-cycles)
    Sel_0_ready[/{0|1|2|3}],<interval> (CPU-cycles)
    Sel_1[/{0|1|2|3}],<interval> (CPU-cycles)
    Sel_2[/{0|1|2|3}],<interval> (CPU-cycles)
    Pick_0[/{0|1|2|3}],<interval> (CPU-cycles)
    Pick_1[/{0|1|2|3}],<interval> (CPU-cycles)
    Pick_2[/{0|1|2|3}],<interval> (CPU-cycles)
    Pick_3[/{0|1|2|3}],<interval> (CPU-cycles)
    Pick_any[/{0|1|2|3}],<interval> (CPU-cycles)
    Branches[/{0|1|2|3}],<interval> (events)
    Instr_FGU_crypto[/{0|1|2|3}],<interval> (events)
    Instr_ld[/{0|1|2|3}],<interval> (precise load-store events)
    Instr_st[/{0|1|2|3}],<interval> (precise load-store events)
```

## 有别名的硬件计数器列表的格式

在有别名的硬件计数器列表中，第一个字段（例如，cycles）提供可以在 collect 命令的 -h *counter...* 参数中使用的别名。此别名还是在 er_print 命令中使用的标识符。

第二个字段列出计数器的可用寄存器。例如，[/{0|1|2|3}]。

第三个字段 <interval> 可指定为 on、hi 或 low，或数值。如果指定为 on、hi 或 low，并且事件到达过快，速率将下降。

第四个字段（在圆括号中）包含类型信息。它提供简短描述（例如 CPU Cycles）、原始硬件计数器名称（例如 Cycles_user）以及计数单位类型（例如 CPU-cycles）。

类型信息字段中的可能条目包括下列项：

- precise – 当指令导致事件计数器溢出时，计数器发生精确中断。缺省情况下，精确计数器的 collect -h 命令收集内存空间和数据空间数据。有关详细信息，

请参见“"DataObjects"（数据对象）视图” [99]、“"DataLayout"（数据布局）视图” [99]和“"MemoryObjects"（内存对象）视图” [98]。

- `load`、`store` 或 `load-store`，表明计数器与内存相关。
- `not-program-related`，计数器会捕获由其他某个程序启动的事件，例如 CPU 到 CPU 的高速缓存嗅探。使用计数器进行分析时将生成警告，并且分析不记录调用堆栈。

如果类型信息的最后一个单词或仅有的单词是：

- `CPU-cycles`，则计数器可用于提供基于时间的度量。针对此类计数器报告的度量在缺省情况下会转换为独占时间和非独占时间，但是也可以显示为事件计数。
- `events`，则度量是非独占和独占事件计数，且无法转换为时间。

在示例中有别名的硬件计数器列表中，第一个计数器的类型信息包含单词 `CPU-cycles`，第二个计数器的类型信息包含单词 `events`。类型信息包括两个单词的，如第三个计数器的 `load-store events`。

### 原始硬件计数器列表的格式

原始硬件计数器列表中包含的信息是有别名的硬件计数器列表中信息的子集。原始硬件计数器列表中的每行包括由 `cputrack`(1) 使用的内部计数器名称、可以在其上使用计数器的寄存器编号、缺省溢出值、类型信息和计数器单位（可以是 `CPU-cycles` 或 `events`）。

如果计数器度量与运行的程序无关的事件，则类型信息的第一个单词是 `not-program-related`。对于这样的计数器，分析不会记录调用堆栈，而是显示人工函数 `collector_not_program_related` 中所用的时间。会记录线程 ID 和 LWP ID，但没有任何意义。

原始计数器的缺省溢出值为 1000003。对于大多数原始计数器来说，此值并非理想值，所以您应在指定原始计数器时指定溢出值。

## 同步等待跟踪数据

在多线程程序中，同步不同线程执行的任务会导致程序执行延迟。例如，一个线程要访问被其他线程锁定的数据时就不得不等待。这些事件称为同步延迟事件，并通过跟踪对 Solaris 或 pthread 线程函数的调用来收集这些事件。收集和记录这些事件的过程称为同步等待跟踪。等待锁花费的时间称为等待时间。

只有等待时间超过阈值（单位为微秒）时，才会记录事件。阈值为 0 表示跟踪所有的同步延迟事件，而不管等待时间为何。缺省阈值通过运行校准测试确定，在该测试中对线程库的调用不会出现任何同步延迟。阈值是这些调用的平均时间与某个因子（当前为

6）的乘积。该过程可防止对此类事件进行记录：即等待时间仅在于调用本身，而与实际的延迟无关。因此，数据量会大大减少，但同步事件的计数可能会被明显低估。

对于 Java 程序，同步跟踪包含所分析程序中的 Java 方法调用，但不会跟踪 JVM 中的任何内部同步调用。

同步等待跟踪数据被转换为下表中的度量。

表 2-2 同步等待跟踪度量

| 度量 | 定义 |
| --- | --- |
| 同步延迟事件计数。 | 对等待时间超过指定阈值的同步例程的调用数目。 |
| 同步等待时间。 | 超过指定阈值的等待时间的总和。 |

通过该信息，您可以确定函数或装入对象对同步例程进行调用时是会经常被阻塞还是会经历很长时间的等待。高同步等待时间表示线程间的争用。您可以通过重新设计算法，尤其是重新组织锁的结构，以便仅包含需要锁定的每个线程的数据来减少争用。

# 堆跟踪（内存分配）数据

对未正确管理的内存分配和取消分配函数进行调用可能会造成数据的使用效率降低，从而导致应用程序的性能降低。在堆跟踪中，收集器通过插入 C 标准库内存分配函数 `malloc`、`realloc`、`valloc` 和 `memalign` 以及取消分配函数 `free` 跟踪内存分配和取消分配请求。对 `mmap` 的调用被视为内存分配，它允许记录 Java 内存分配的堆跟踪事件。Fortran 函数 `allocate` 和 `deallocate` 调用 C 标准库函数，因此会间接跟踪这些例程。

不支持对 Java 程序的堆分析。

堆跟踪数据会被转换为以下度量。

表 2-3 内存分配（堆跟踪）度量

| 度量 | 定义 |
| --- | --- |
| Allocations（分配） | 对内存分配函数的调用数 |
| Bytes allocated（分配的字节数） | 每次调用内存分配函数时分配的字节总数 |
| Leaks（泄漏） | 调用内存分配函数（未对取消分配函数进行相应的调用）的数量 |
| Bytes leaked（泄漏的字节数） | 已分配但未取消分配的字节数 |

收集堆跟踪数据有助于识别程序中的内存泄漏，或定位内存分配效率不高的位置。

查看应用了过滤器的 "Leaks"（泄漏）视图时，显示的泄漏是在过滤标准下完成的内存分配，这些分配在任何时候都不会取消。泄漏不限于在过滤标准下未取消分配的分配。

有时（如在 dbx 调式工具中），内存泄漏定义为动态分配的内存块，在程序的数据空间中没有任何指向它的指针。此处所使用的泄漏定义包括这种替换的定义，但也包括存在指针的内存。

## I/O 跟踪数据

I/O 数据收集跟踪输入/输出系统调用，包括读取和写入。该收集测量调用的持续时间，跟踪文件和描述符以及传输的数据量。您可以使用 I/O 度量识别具有较高传输字节量和较长总线程时间的文件、文件句柄和调用堆栈。

表 2-4 I/O 跟踪度量

| 度量 | 定义 |
| --- | --- |
| Read Bytes（读取字节数） | 每次调用读取函数时读取的字节总数。 |
| Write Bytes（写入字节数） | 每次调用写入函数时写入的字节总数。 |
| Read Count（读取计数） | 执行读取调用的次数。 |
| Write Count（写入计数） | 执行写入调用的次数。 |
| Other I/O Count（其他 I/O 计数） | 执行其他 IO 调用的次数 |
| I/O Error Count（I/O 错误计数） | IO 调用期间发生的错误数 |
| Read Time（读取时间） | 读取数据花费的秒数 |
| Write Time（写入时间） | 写入数据花费的秒数 |
| Other I/O Time（其他 I/O 时间） | 执行其他 IO 调用花费的秒数 |
| I/O Error Time（I/O 错误时间） | 花费在 IO 错误上的秒数 |

## 抽样数据

全局数据由收集器中称为抽样包的数据包来记录。每个数据包中都包含一个包头、时间戳、内核的执行统计信息（如缺页和 I/O 数据）、上下文切换以及各种页面驻留（工作集和分页）统计信息。记录在抽样包中的数据对程序来说是全局的，且不会转换为性能度量。记录抽样包的过程称为抽样。

在以下情况下，抽样包会被记录下来：

- 当程序在 dbx 中调试期间因任何原因停止时（如在断点处，前提是设置了有关在断点处停止的选项）。

- 如果选择了定期抽样，则在抽样间隔时间结束时。抽样间隔被指定为以秒为单位的整数。缺省值为 1 秒。
- 使用 `dbx collector sample record` 命令手动记录抽样时。
- 如果代码中包含对 `collector_sample` 的调用，则在调用该例程时（请参见 “使用 libcollector 库从程序控制数据收集” [43]）。
- 如果将 `-l` 选项与 `collect` 命令一起使用，则在传送指定信号时（请参见 `collect`(1) 手册页）。
- 开始和终止收集时。
- 使用 `dbx collector pause` 命令暂停收集时（就在暂停之前）和使用 `dbx collector resume` 命令恢复收集时（就在恢复之后）。
- 创建子孙进程前后。

性能工具使用记录在抽样包中的数据按时间段将数据分组，这些称为抽样。您可以通过选择一组抽样过滤特定于事件的数据，以便只查看这些特定时间段的信息。您也可以查看每个抽样的全局数据。

性能工具不对不同种类的抽样点进行区分。要利用抽样点进行分析，您应只选择一种类型的点进行记录。具体地说，如果要记录与程序结构或执行序列有关的抽样点，则应关闭定期抽样，并使用在 `dbx` 停止进程时，或将信号传送到正使用 `collect` 命令记录数据的进程时，或调用收集器 API 函数时记录的抽样。

## MPI 跟踪数据

收集器可以收集有关对消息传递接口 (Message Passing Interface, MPI) 库的调用的数据。

使用开源 VampirTrace 5.5.3 发行版来实现 MPI 跟踪。该跟踪可识别以下 VampirTrace 环境变量：

| | |
|---|---|
| `VT_STACKS` | 控制是否在数据中记录调用堆栈。缺省设置为 `1`。将 `VT_STACKS` 设置为 `0` 将禁用调用堆栈。 |
| `VT_BUFFER_SIZE` | 控制 MPI API 跟踪收集器的内部缓冲区的大小。缺省值为 `64M`（64 兆字节）。 |
| `VT_MAX_FLUSHES` | 控制在终止 MPI 跟踪前刷新缓冲区的次数。缺省值是 `0`，用于设置只要缓冲区满了就允许刷新到磁盘。将 `VT_MAX_FLUSHES` 设置为正数将为刷新缓冲区的次数设置限制。 |
| `VT_VERBOSE` | 启用各种错误和状态消息显示。缺省值为 `1`，在此设置下会启用紧急错误和状态消息。如果出现问题，请将此变量设置为 `2`。 |

有关这些变量的更多信息，请参见 Technische Universität Dresden Web 站点上的《Vampirtrace User Manual》（《Vampirtrace 用户手册》）。

在达到缓冲区限制之后发生的 MPI 事件将不会写入跟踪文件，这将导致跟踪不完整。

要去掉限制并获取应用程序的完整跟踪，请将 VT_MAX_FLUSHES 环境变量设置为 0。该设置将导致 MPI API 跟踪收集器在缓冲区已满时刷新磁盘的缓冲区。

要更改缓冲区大小，请设置 VT_BUFFER_SIZE 环境变量。该变量的最佳值取决于要跟踪的应用程序。设置较小的值将增加应用程序可以使用的内存，但是将触发 MPI API 跟踪收集器频繁进行缓冲区刷新。这些缓冲区刷新可能会显著改变应用程序的行为。另一方面，设置较大的值（如 2 G）可以使 MPI API 跟踪收集器刷新缓冲区的次数降至最低，但是将减少应用程序可以使用的内存。如果没有足够的内存可用来容纳缓冲区和应用程序数据，应用程序的某些部分可能会交换至磁盘，从而导致应用程序的行为发生显著改变。

以下列表显示会收集数据的函数。

| | | |
|---|---|---|
| MPI_Abort | MPI_Accumulate | MPI_Address |
| MPI_Allgather | MPI_Allgatherv | MPI_Allreduce |
| MPI_Alltoall | MPI_Alltoallv | MPI_Alltoallw |
| MPI_Attr_delete | MPI_Attr_get | MPI_Attr_put |
| MPI_Barrier | MPI_Bcast | MPI_Bsend |
| MPI_Bsend-init | MPI_Buffer_attach | MPI_Buffer_detach |
| MPI_Cancel | MPI_Cart_coords | MPI_Cart_create |
| MPI_Cart_get | MPI_Cart_map | MPI_Cart_rank |
| MPI_Cart_shift | MPI_Cart_sub | MPI_Cartdim_get |
| MPI_Comm_compare | MPI_Comm_create | MPI_Comm_dup |
| MPI_Comm_free | MPI_Comm_group | MPI_Comm_rank |
| MPI_Comm_remote_group | MPI_Comm_remote_size | MPI_Comm_size |
| MPI_Comm_split | MPI_Comm_test_inter | MPI_Dims_create |
| MPI_Errhandler_create | MPI_Errhandler_free | MPI_Errhandler_get |
| MPI_Errhandler_set | MPI_Error_class | MPI_Error_string |
| MPI_File_close | MPI_File_delete | MPI_File_get_amode |
| MPI_File_get_atomicity | MPI_File_get_byte_offset | MPI_File_get_group |
| MPI_File_get_info | MPI_File_get_position | MPI_File_get_position_shared |
| MPI_File_get_size | MPI_File_get_type_extent | MPI_File_get_view |
| MPI_File_iread | MPI_File_iread_at | MPI_File_iread_shared |
| MPI_File_iwrite | MPI_File_iwrite_at | MPI_File_iwrite_shared |
| MPI_File_open | MPI_File_preallocate | MPI_File_read |
| MPI_File_read_all | MPI_File_read_all_begin | MPI_File_read_all_end |
| MPI_File_read_at | MPI_File_read_at_all | MPI_File_read_at_all_begin |
| MPI_File_read_at_all_end | MPI_File_read_ordered | MPI_File_read_ordered_begin |
| MPI_File_read_ordered_end | MPI_File_read_shared | MPI_File_seek |

| | | |
|---|---|---|
| MPI_File_seek_shared | MPI_File_set_atomicity | MPI_File_set_info |
| MPI_File_set_size | MPI_File_set_view | MPI_File_sync |
| MPI_File_write | MPI_File_write_all | MPI_File_write_all_begin |
| MPI_File_write_all_end | MPI_File_write_at | MPI_File_write_at_all |
| MPI_File_write_at_all_begin | MPI_File_write_at_all_end | MPI_File_write_ordered |
| MPI_File_write_ordered_begin | MPI_File_write_ordered_end | MPI_File_write_shared |
| MPI_Finalize | MPI_Gather | MPI_Gatherv |
| MPI_Get | MPI_Get_count | MPI_Get_elements |
| MPI_Get_processor_name | MPI_Get_version | MPI_Graph_create |
| MPI_Graph_get | MPI_Graph_map | MPI_Graph_neighbors |
| MPI_Graph_neighbors_count | MPI_Graphdims_get | MPI_Group_compare |
| MPI_Group_difference | MPI_Group_excl | MPI_Group_free |
| MPI_Group_incl | MPI_Group_intersection | MPI_Group_rank |
| MPI_Group_size | MPI_Group_translate_ranks | MPI_Group_union |
| MPI_Ibsend | MPI_Init | MPI_Init_thread |
| MPI_Intercomm_create | MPI_Intercomm_merge | MPI_Irecv |
| MPI_Irsend | MPI_Isend | MPI_Issend |
| MPI_Keyval_create | MPI_Keyval_free | MPI_Op_create |
| MPI_Op_free | MPI_Pack | MPI_Pack_size |
| MPI_Probe | MPI_Put | MPI_Recv |
| MPI_Recv_init | MPI_Reduce | MPI_Reduce_scatter |
| MPI_Request_free | MPI_Rsend | MPI_rsend_init |
| MPI_Scan | MPI_Scatter | MPI_Scatterv |
| MPI_Send | MPI_Send_init | MPI_Sendrecv |
| MPI_Sendrecv_replace | MPI_Ssend | MPI_Ssend_init |
| MPI_Start | MPI_Startall | MPI_Test |
| MPI_Test_cancelled | MPI_Testall | MPI_Testany |
| MPI_Testsome | MPI_Topo_test | MPI_Type_commit |
| MPI_Type_contiguous | MPI_Type_extent | MPI_Type_free |
| MPI_Type_hindexed | MPI_Type_hvector | MPI_Type_indexed |
| MPI_Type_lb | MPI_Type_size | MPI_Type_struct |
| MPI_Type_ub | MPI_Type_vector | MPI_Unpack |
| MPI_Wait | MPI_Waitall | MPI_Waitany |
| MPI_Waitsome | MPI_Win_complete | MPI_Win_create |
| MPI_Win_fence | MPI_Win_free | MPI_Win_lock |
| MPI_Win_post | MPI_Win_start | MPI_Win_test |
| MPI_Win_unlock | | |

MPI 跟踪数据会被转换为以下度量。

表 2-5 MPI 跟踪度量

| 度量 | 定义 |
| --- | --- |
| MPI Sends（MPI 发送次数） | 已启动 MPI 点对点发送数 |
| MPI Bytes Sent（发送的 MPI 字节数） | "MPI Sends"（MPI 发送次数）中的字节数 |
| MPI Receives（MPI 接收次数） | 已完成 MPI 点对点接收数 |
| MPI Bytes Received（接收的 MPI 字节数） | "MPI Receives"（MPI 接收次数）中的字节数 |
| MPI Time（MPI 时间） | 对 MPI 函数的所有调用所花费的时间 |
| Other MPI Events（其他 MPI 事件） | 对既没有发送也没有接收点对点消息的 MPI 函数的调用数 |

"MPI Time"（MPI 时间）是 MPI 函数中所用的总线程时间。如果还收集了 MPI 状态时间，则除 MPI_Init 和 MPI_Finalize 之外的所有 MPI 函数的 MPI 工作时间加上 MPI 等待时间应大约等于 MPI 工作时间。在 Linux 上，MPI 等待和工作时间基于用户 CPU 时间加系统 CPU 时间，而 MPI 时间基于实际时间，所以这些数值将不匹配。

当前，仅针对点对点消息收集 MPI 字节和消息计数。不针对集合通信函数记录 MPI 字节和消息计数。"MPI Bytes Received"（接收的 MPI 字节数）度量会计算所有消息中接收的实际字节数。"MPI Bytes Sent"（发送的 MPI 字节数）会计算所有消息中发送的实际字节数。"MPI Sends"（MPI 发送次数）会计算发送的消息数，"MPI Receives"（MPI 接收次数）会计算接收的消息数。

收集 MPI 跟踪数据有助于标识 MPI 程序中可能因 MPI 调用而产生性能问题的位置。可能发生的性能问题的例子有负载平衡、同步延迟和通信瓶颈。

# 如何将度量分配到程序结构

使用与特定事件的数据一起记录的调用堆栈将度量分配到程序指令。如果该信息可用，则会将每条指令都映射到一行源代码，而分配到该指令的度量也被分配到该行源代码。有关如何完成此过程的更多详细说明，请参见第 6 章 了解性能分析器及其数据。

除了源代码和指令，还会将度量分配到更高级别的对象：函数和装入对象。调用堆栈包含在执行分析时记录的有关函数调用到达指令地址的序列的信息。性能分析器使用调用堆栈来计算程序中每个函数的度量。这些度量称为函数级度量。

## 函数级度量：独占、非独占和归属

性能分析器可计算三种类型的函数级度量：独占度量、非独占度量和归属度量。

- 函数的**独占度量**通过函数本身内部发生的事件计算得出：这种度量不包括来自对其他函数调用的度量。
- **非独占度量**通过函数本身和其调用的函数内部发生的事件计算得出：这种度量包括来自对其他函数调用的度量。
- **归属度量**说明了非独占度量在多大程度上来自对（或从）其他函数的调用：这种度量归属到其他函数的度量。

对于只出现在调用堆栈底部的函数（叶函数），独占度量和非独占度量是相同的。

对于装入对象，也要计算独占度量和非独占度量。装入对象的独占度量通过累加装入对象中所有函数上函数级别的度量计算得出。装入对象的非独占度量与函数的非独占度量的计算方法相同。

函数的独占度量和非独占度量给出了有关所有通过函数记录的路径信息。归属度量给出了有关通过函数记录的特定路径的信息。这些度量显示了度量在多大程度上来自特定函数调用。调用中所涉及的两个函数称为调用方和被调用方。对于调用树中的每个函数：

- 函数调用方的归属度量说明了函数的非独占度量在多大程度上归因于来自每个调用方的调用。调用方的归属度量的总和等于函数的非独占度量。
- 函数的被调用方的归属度量说明了函数的非独占度量在多大程度上来自对每个被调用方的调用。它们的和加上函数的独占度量等于函数的非独占度量。

各度量间的关系可通过以下等式表示：

$$\sum_{\text{调用方}} \text{归属度量} = \text{非独占度量} = \left( \sum_{\text{被调用方}} \text{归属度量} + \text{独占度量} \right)$$

通过比较调用方或被调用方的归属度量和非独占度量，可以得到以下进一步的信息：

- 调用方的归属度量和非独占度量之间差额说明了度量在多大程度上来自对其他函数的调用以及调用方本身的工作。
- 被调用方的归属度量和非独占度量之间的差额说明了被调用方的非独占度量在多大程度上来自从其他函数对它的调用。

要定位可改善程序性能的位置，请执行以下操作：

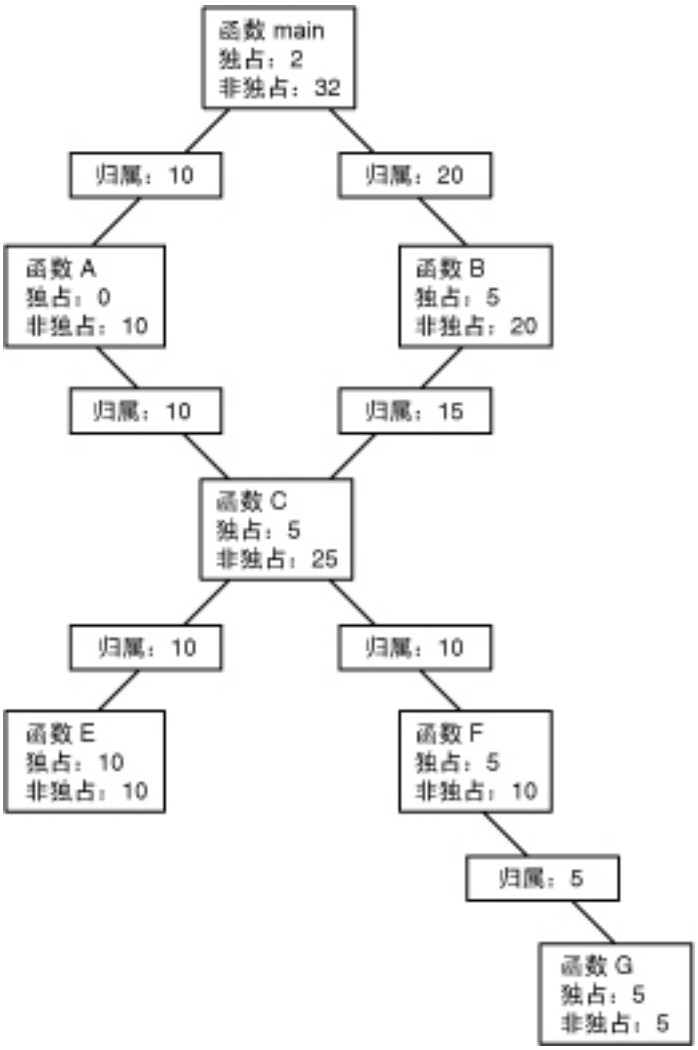
- 使用独占度量定位具有高度量值的函数。
- 使用非独占度量确定程序中哪个调用序列导致高度量值。
- 使用归属度量跟踪导致高度量值的函数的特定调用序列。

# 解释归属度量：示例

图 2-1 “说明独占、非独占和归属度量的调用树” 中说明了独占、非独占和归属度量，该图包含完整的调用树。其中的焦点是中心函数，即函数 C。

图的后面显示了该程序的伪代码。

图 2-1 说明独占、非独占和归属度量的调用树




Main 函数调用函数 A 和函数 B，将 10 个单位的非独占度量归属函数 A 并将 20 个单位归属函数 B。这些是函数 Main 的被调用方归属度量。它们的总和 (10+20) 加上函数 Main 的独占度量等于函数 main 的非独占度量 (32)。

由于函数 A 将其所有时间都花费在对函数 C 的调用上，因此它的独占度量为 0 个单位。

函数 C 由函数 A 和函数 B 这两个函数调用，将 10 个单位的非独占度量归属函数 A 并将 15 个单位归属函数 B。这些是调用方归属度量。它们的总和 (10+15) 等于函数 C 的非独占度量 (25)。

调用方归属度量相当于函数 A 和 B 的非独占度量与独占度量之间的差异，这表示它们各自仅调用函数 C。（事实上，这些函数可能调用其他函数，但时间很短，不会显示在实验中。）

函数 C 调用函数 E 和函数 F 这两个函数，将 10 个单位的非独占度量归属函数 E 并将 10 个单位归属函数 F。这些是被调用方归属度量。它们的总和 (10+10) 加上函数 C 的独占度量 (5) 等于函数 C 的非独占度量 (25)。

对于函数 E 和函数 F 来说，被调用方归属度量和被调用方非独占度量是相同的。这表示函数 E 和函数 F 都仅由函数 C 调用。对于函数 E 来说，独占度量和非独占度量是相同的，但对于函数 F 来说则不同。这是由于函数 F 调用另一个函数（函数 G），而函数 E 没有。

下面显示了该程序的伪代码。

```
main() {
   A();
   /Do 2 units of work;/
   B();
}

A() {
   C(10);
}

B() {
   C(7.5);
   /Do 5 units of work;/
   C(7.5);
}

C(arg) {
      /Do a total of "arg" units of work, with 20% done in C itself,
      40% done by calling E, and 40% done by calling F./
}
```

# 递归如何影响函数级度量

递归函数直接或间接的调用使得度量的计算复杂化。性能分析器将函数的度量作为一个整体显示，而不是显示函数的每个调用的度量：因此，必须将一系列递归调用的度量压缩为单一度量。此行为不会影响通过调用堆栈底部的函数（叶函数）计算得出的独占度量，但会影响非独占度量和归属度量。

非独占度量是通过将事件的度量添加到调用堆栈中函数的非独占度量来计算的。为了确保在递归调用堆栈中不重复计算度量，事件的度量仅能向每个唯一函数的非独占度量添加一次。

归属度量是通过非独占度量来计算的。在最简单的递归中，递归函数具有两个调用方：它本身和另一个函数（初始化函数）。如果在最后的调用中完成了所有工作，会将递归函数的非独占度量归属到它本身，而不是初始化函数。之所以发生此归属，是因为递归函数的所有更高调用的非独占度量均被视为零，以避免重复计算度量。但是，初始化函数会由于递归调用而作为被调用方正确归属到递归函数的非独占度量部分。

♦ ♦ ♦ 第 3 章

# 3 收集性能数据

性能分析的第一个阶段是数据收集。本章介绍了进行数据收集的要求、数据的存储位置、如何收集数据以及如何管理数据收集。有关数据本身的更多信息，请参见第 2 章 性能数据。

可通过命令行或性能分析器工具执行数据收集。

通过命令行从内核收集数据需要单独的工具 `er_kernel`。有关更多信息，请参见第 9 章 内核分析。

本章包含以下主题。



## 编译和链接程序

无论程序使用何种编译器选项进行编译，您都可以为该程序收集和分析数据，但有些选项会影响能够在性能分析器中收集或查看的内容。以下几个小节介绍了在编译和链接程序时应考虑的问题。

# 针对源代码分析进行编译

要在带注释的 "Source"（源）和 "Disassembly"（反汇编）视图中查看源代码以及在 "Lines"（行）视图中查看源代码行，就必须使用 `-g` 编译器选项（对于 C++ 来说为用于启用前端内联的 `-g0`）编译感兴趣的源文件，以生成调试符号信息。调试符号信息的格式可以是 DWARF2 或 `stabs`，由 `-xdebugformat=(dwarf|stabs)` 指定。缺省的调试格式是 `dwarf`。

用 DWARF 格式的调试符号生成的可执行文件和库会自动包括每个要素对象文件调试符号的副本。如果用 `stabs` 格式的调试符号生成的可执行文件和库是通过 `-xs` 选项（该选项将 `stabs` 符号保留在各个对象文件及可执行文件中）进行链接的，那么所生成的可执行文件和库中也会包括每个要素对象文件调试符号的副本。当您需要移动或删除对象文件时，包括这些信息尤为重要。使用可执行文件和库本身中的所有调试符号，可以更容易地将实验和与程序相关的文件移至新位置。

可以通过 Oracle Solaris Studio 编译器或 GNU 编译器来编译程序。但是，GNU 编译器无法支持某些功能，如使用 OpenMP 重构的调用堆栈。

在 Studio 编译器中，使用 `-g` 编译不会更改优化，`O2` 和 `O3` 优化级别的尾部调用优化除外。

支持 Java 代码的源代码级信息。与本机语言不同，在实验中不会记录 Java 源代码的位置。您可能需要使用路径映射或者设置搜索路径来指向源代码。有关更多信息，请参见“工具如何查找源代码” [189]。

## 针对数据空间和内存空间分析进行编译

数据空间分析将内存访问归属到数据结构元素。要启用数据空间分析，必须使用 Oracle Solaris Studio 编译器和 `-xhwcprof` 选项编译 C、C++ 和 Fortran 可执行文件。如果不使用该选项进行编译，则 "DataObjects"（数据对象）和 "DataLayout"（数据布局）视图不会显示二进制文件的数据。

内存空间分析允许您查看哪些内存地址消耗的性能最多。在针对内存空间分析准备程序时不需要使用特殊编译器选项，但只能在运行 Oracle Solaris 10 1/13 的 SPARC 平台以及运行 Oracle Solaris 11.2 的 Intel 平台上使用此功能。有关更多信息，请参见“数据空间分析和内存空间分析” [165]。

# 静态链接

对于某些类型的性能数据，如堆跟踪和 I/O 跟踪，数据收集依赖于动态链接的 `libc`。进行静态链接时将失去此功能，因此不应将 `-dn` 和 `-Bstatic` 等选项用于 Oracle Solaris Studio 编译器。

如果试图收集完全静态链接的程序的数据，则收集器会输出一条错误消息且不会收集数据。出现此错误的原因在于，当您运行收集器时，收集器库也会像其他库一样动态装入。

请不要静态链接任何系统库或收集器库 `libcollector.so`。

## 共享对象处理

通常，`collect` 命令会为目标地址空间中的所有共享对象收集数据，而不管这些对象是在初始库列表中，还是使用 `dlopen`（） 显式装入。但是，在某些情况下不会分析某些共享对象：

- 利用延迟装入调用目标程序时。在这种情况下，库不是在启动时装入的，并且不是通过显式调用 `dlopen`（） 装入的，因此共享对象不包括在实验中，其中的所有 PC 将映射到 `<Unknown>` 函数。解决方法是设置 `LD_BIND_NOW` 环境变量，该设置可强制在启动时装入库。
- 使用 `-B direct` 选项生成可执行文件时。在这种情况下，通过专门针对 `dlopen`（） 动态链接程序入口点的调用来动态装入对象，并且忽略 `libcollector` 插入。共享对象名称不包括在实验中，其中的所有 PC 将映射到 `<Unknown>`（） 函数。解决方法是不使用 `-B direct` 选项。

## 编译时优化

如果在某一级别启用优化的情况下编译程序，编译器就可以重新安排执行顺序，这样就无须严格按照程序中行的顺序来执行程序。性能分析器会分析从优化后的代码中收集的实验，但它在反汇编级别所显示的数据通常很难与初始源代码行相关联。此外，如果编译器执行尾部调用优化，则调用序列可能与预期的序列不同。有关更多信息，请参见“尾部调用优化” [169]。

## 编译 Java 程序

用 `javac` 命令编译 Java 程序无需任何特殊操作。

# 为数据收集和分析准备程序

对于大多数程序来说，您不必为数据收集和分析做任何特殊的准备。如果程序执行下列任一操作，则应当阅读下面的一个或多个小节：

- 安装信号处理程序。请参见“数据收集和信号” [42]。

- 显式动态装入系统库。请参见“使用系统库” [41]。
- 动态编译函数。请参见“动态函数和模块” [46]。
- 创建要分析的子孙进程。请参见“使用系统库” [41]。
- 直接使用分析计时器或硬件计数器 API。请参见“使用系统库” [41]。
- 调用 `setuid`(2) 或执行 `setuid` 文件。请参见“数据收集和信号” [42]和“使用 `setuid` 和 `setgid`” [43]。

此外，如果要在运行时从程序控制数据收集，请参阅“使用 libcollector 库从程序控制数据收集” [43]。

# 使用动态分配的内存

许多程序依赖于动态分配的内存，它们使用诸如以下各项的功能：

- `malloc`、`valloc` 和 `alloca` (C/C++)
- `new` (C++)
- 堆栈局部变量 (Fortran)
- `MALLOC` 和 `MALLOC64` (Fortran)

必须小心确保程序不依赖于动态分配的内存的初始内容，除非内存分配方法明确地说明要设置初始值。例如，比较 `malloc`(3C) 手册页中的 `calloc` 和 `malloc` 的描述。

偶尔，使用动态分配的内存的程序似乎可以单独地正常运行，但是启用性能数据收集之后就会失败。症状可能包括意外的浮点行为、段故障或特定于应用程序的错误消息。

如果应用程序单独运行时未初始化的内存偶然设置为良性值，但应用程序与性能数据收集工具一起运行时未初始化的内存被设置为其他值，则会出现这种行为。发生这种情况时，问题不出在性能工具上。依赖于动态分配的内存内容的任何应用程序都具有潜在的已知问题：除非明确说明使用其他方式，否则操作系统将为动态分配的内存随机提供任意内容。即使目前操作系统会始终将动态分配的内存设置为某个值，但是将来在使用操作系统的后续修订版或将程序移植到其他操作系统时，这些潜在的已知问题会引起意外的行为。

下列工具可以帮助您找到这些潜在的已知问题：

- 代码分析器，该 Oracle Solaris Studio 工具在与编译器和其他工具一起使用时可显示以下信息：

  静态代码检查 — 代码分析器可显示静态代码检查的结果，当您使用 Oracle Solaris Studio C 或 C++ 编译器并指定 `–xanalyze=code` 选项来编译应用程序时，执行该检查。

  动态内存访问检查 — 代码分析器可显示动态内存访问检查的结果，当您使用带有 `–a` 选项的 `discover` 检测二进制文件，然后运行检测后的二进制文件以生成数据时，执行该检查。

有关更多信息，请参见《Oracle Solaris Studio 12.4：代码分析器用户指南》

- `f95 -xcheck=init_local`

  有关更多信息，请参见《Oracle Solaris Studio 12.4：Fortran 用户指南》或 `f95`(1) 手册页。
- `lint` 实用程序

  有关更多信息，请参见《Oracle Solaris Studio 12.4：C 用户指南》或 `lint`(1) 手册页。
- `dbx` 下的运行时检查

  有关更多信息，请参见《Oracle Solaris Studio 12.4：使用 dbx 调试程序》手册或 `dbx`(1) 手册页。

# 使用系统库

收集器插入各种系统库的函数，以收集跟踪数据并确保数据收集的完整性。下面的列表描述了收集器插入库函数调用的情况。

- 收集同步等待跟踪数据。在 Oracle Solaris 上，收集器插入 Oracle Solaris C 库 `libc.so` 中的函数。
- 收集堆跟踪数据。收集器插入函数 `malloc`、`realloc`、`memalign` 和 `free`。这些函数的版本可以在 C 标准库 `libc.so` 和其他库（如 `libmalloc.so` 和 `libmtmalloc.so`）中找到。
- 收集 MPI 跟踪数据。收集器从指定的 MPI 库插入函数。
- 确保时钟数据的完整性。收集器插入 `setitimer` 并阻止程序使用分析计时器。
- 确保硬件计数器数据的完整性。收集器插入硬件计数器库 `libcpc.so` 中的函数并阻止程序使用计数器。程序对该库中函数的调用的返回值是 `-1`。
- 针对子孙进程启用数据收集。收集器插入函数 `fork`(2)、`fork1`(2)、`vfork`(2)、`fork`(3F)、`posix_spawn`(3p)、`posix_spawnp`(3p)、`system`(3C)、`system`(3F) 和 `exec`(2) 及其变体。对 `vfork` 的调用已在内部被替换为对 `fork1` 的调用。这些插入适用于 `collect` 命令。
- 保证由收集器处理 `SIGPROF` 和 `SIGEMT` 信号。收集器插入 `sigaction` 以确保其信号处理程序是这些信号的主信号处理程序。

在下列情况下，插入不会成功：

- 将程序与任何包含被插入的函数的库进行静态链接。
- 将 `dbx` 附加到运行中的未预装入收集器库的应用程序。
- 动态装入其中一个库并通过只在该库中搜索来解析符号。

收集器插入失败可能会导致性能数据丢失或无效。

`er_sync.so`、`er_heap.so` 和 `er_mpview`*n*`.so`（其中 *n* 表示 MPI 版本）库仅在分别请求同步等待跟踪数据、堆跟踪数据或 MPI 跟踪数据时装入。

# 数据收集和信号

信号同时用于时钟分析和硬件计数器分析。SIGPROF 用于所有实验的数据收集。生成信号的周期取决于收集的数据。SIGEMT（在 Solaris 上）或 SIGIO（在 Linux 上）用于硬件计数器分析。溢出间隔取决于要分析的用户参数。任何使用或处理分析信号的用户代码都可能会干扰数据收集。当收集器安装用于分析信号的信号处理程序时，它会设置一个标志来确保系统调用不被中断以传送信号。此设置可能会更改将分析信号用于其他用途的目标程序的行为。

当收集器为分析信号安装其信号处理程序时，它会记住目标是否安装了自己的信号处理程序。收集器还会插入到某些信号处理例程且不允许用户为这些信号安装信号处理程序；它会保存用户的处理程序，就像收集器在启动实验时替换用户处理程序一样。

分析信号由内核中的分析计时器或硬件计数器溢出处理代码提供，或在对 kill(2)、sigsend(2) tkill(2)、tgkill(2) 或 _lwp_kill(2) 系统调用、raise(3C) 和 sigqueue(3C) 库调用或 kill 命令的响应中提供。会随信号提供信号代码，以便收集器能区分源。如果针对分析提供，收集器会处理该代码；如果不是针对分析提供，则会将其提供给目标信号处理程序。

在 dbx 下运行收集器时，提供的分析信号的信号代码有时会被损坏，此时会当作分析信号是从系统或库调用或某个命令生成的。在这种情况下，会错误地将其提供给用户的处理程序。如果用户处理程序设置为 SIG_DFL，将导致进程的信息转储失败。

在连接到目标进程后调用收集器时，收集器会安装其信号处理程序，但它无法插入到信号处理例程。如果用户代码在连接后安装信号处理程序，则该处理程序会覆盖收集器的信号处理程序，且数据将会丢失。

请注意，包含任一分析信号的信号都可能会使系统调用过早终止。程序必须准备好处理此行为。当 libcollector 安装信号处理程序以进行数据收集时，它会指定重新启动可重新启动的系统调用。但是，类似 sleep(3C) 的某些系统调用会过早返回而不报告错误。

## 抽样信号和暂停-恢复信号

用户可以将信号指定为抽样信号 (-l) 或暂停-恢复信号 (-y)。建议将 SIGUSR1 或 SIGUSR2 用于此用途，但也可以使用目标未使用的任何信号。

如果进程未将分析信号用于其他用途，则可以使用这些信号，但仅在没有其他信号可用时才能使用这些信号。收集器会插入到某些信号处理例程且不允许用户为这些信号安装信号处理程序；它会保存用户的处理程序，就像收集器在启动实验时替换用户处理程序一样。

如果在连接到目标进程后调用收集器，且用户代码为抽样信号或暂停-恢复信号安装了一个信号处理程序，则这些信号将无法再按指定完成操作。

## 使用 setuid 和 setgid

由于动态装入器实施了一定的限制，因此将难以使用 setuid(2) 和收集性能数据。如果您的程序调用 setuid 或执行 setuid 文件，则收集器可能无法写入实验文件，原因是它缺少新用户 ID 的必需权限。

collect 命令通过将共享库 libcollector.so 插入目标的地址空间 (LD_PRELOAD) 来运行。如果对调用 setuid 或 setgid 或创建调用 setuid 或 setgid 的子孙进程的可执行文件调用过的 collect 命令进行调用，可能会出现多个问题。如果您不是 root 用户，在运行实验时，收集会因为共享库未安装在可信目录中而失败。解决方法是以 root 用户身份运行实验，或使用 crle(1) 授予权限。应对安全障碍时请格外小心，操作风险需自行承担。

运行 collect 命令时，必须为您、由使用 exec（） 执行的程序的 setuid 属性和 setgid 属性设置的任何用户或组以及该程序自身设置的任何用户或组，将 umask 设置为允许写权限。如果未正确设置掩码，某些文件可能无法写入实验，并且可能无法处理实验。如果可以写入日志文件，尝试处理实验时将显示错误。

如果目标本身发出了设置 UID 或 GID 的任何系统调用，或者如果目标更改其 umask，然后对其他某个可执行文件派生或运行 exec（） ，或者 crle 用于配置运行时链接程序如何搜索共享对象，则可能会出现其他问题。

如果在更改其有效 GID 的目标上以 root 用户身份启动实验，实验终止时自动运行的 er_archive 进程将失败，原因是它需要未标记为可信的共享库。在这种情况下，您可以在实验终止后立即在记录实验的计算机上运行 er_archive 实用程序（或 er_print 实用程序或 analyzer 命令）。

## 使用 libcollector 库从程序控制数据收集

如果要控制程序的数据收集，收集器共享库 libcollector.so 包含了一些可以使用的 API 函数。这些函数是用 C 语言编写的。另外，也提供 Fortran 接口。C 接口和 Fortran 接口都是在由库所提供的头文件中定义的。

API 函数定义如下所示。

```
void collector_sample(char *name);
void collector_pause(void);
void collector_resume(void);
void collector_terminate_expt(void);
```

CollectorAPI 类为 Java™ 程序提供了类似的功能，“Java 接口” [44]中对其进行了介绍。

## C 和 C++ 接口

可以通过包括 `collectorAPI.h` 并与 `-lcollectorAPI`（包含用于检查底层 `libcollector.so` API 函数是否存在的实际函数）相链接来访问收集器 API 的 C 和 C++ 接口。

如果没有活动的实验，API 调用将被忽略。

## Fortran 接口

Fortran API `libfcollector.h` 文件定义了库的 Fortran 接口。要使用该库，必须使用 `-lcollectorAPI` 链接应用程序。（还提供了该库的替代名称 `-lfcollector`，目的在于实现向后兼容性。除动态函数、线程暂停和恢复调用等功能外，Fortran API 提供了与 C 和 C++ API 相同的功能。

要使用 Fortran 的 API 函数，请插入下面的语句：

```
include "libfcollector.h"
```

---

注 - 请勿使用 `-lcollector` 链接任何语言的程序。否则，收集器可能会出现不可预知的行为。

---

## Java 接口

使用以下语句可以导入 `CollectorAPI` 类并访问 Java API。但是请注意，必须使用指向 */installation_directory*`/lib/collector.jar` 的类路径来调用应用程序，其中 *installation-directory* 是 Oracle Solaris Studio 软件的安装目录。

```
import com.sun.forte.st.collector.CollectorAPI;
```

Java `CollectorAPI` 方法的定义如下所示：

```
CollectorAPI.sample(String name)
CollectorAPI.pause()
CollectorAPI.resume()
CollectorAPI.terminate()
```

除动态函数 API 之外，Java API 包含与 C 和 C++ API 相同的函数。

C 头文件 `libcollector.h` 包含一些宏，这些宏的作用是如果当时未在收集数据，则跳过对实际 API 函数的调用。在这种情况下，不动态装入函数。但是，由于在某些情况下这些宏不能很好地运行，所以使用这些宏会有风险。使用 `collectorAPI.h` 较为安全，因为它不使用宏。而是直接引用函数。

如果正在收集性能数据，则 Fortran API 子例程会调用 C API 函数，否则这些子例程将返回。检查的开销很低，不会对程序性能产生太大的影响。

如本章稍后所述，要收集性能数据就必须使用收集器运行您的程序。插入对 API 函数的调用不会启用数据收集功能。

如果要在多线程程序中使用 API 函数，应当确保它们只由一个线程调用。API 函数执行适用于进程（而不是单独的线程）的操作。如果每个线程都调用 API 函数，则记录的数据可能会与预期不同。例如，如果一个线程在其他线程到达程序中的同一点之前调用了 `collector_pause（）` 或 `collector_terminate_expt（）`，则会针对所有线程暂停或终止收集，从而丢失那些正在执行 API 调用之前代码的线程的数据。

# C、C++、Fortran 和 Java API 函数

本节介绍 API 函数。

- C 和 C++：`collector_sample(char *name)`

  Fortran: `collector_sample(string name)`

  Java：`CollectorAPI.sample(String name)`

  记录抽样包并用指定的名称或字符串标记该抽样。当在 "Timeline"（时间线）视图中选择一个抽样时，性能分析器将在 "Selection Details"（选择详细信息）窗口中显示此标签。Fortran 参数 `string` 的类型为 `character`。

  抽样点包含进程（而不是单独的线程）的数据。在多线程应用程序中，如果在 `collector_sample()` API 函数记录抽样时发生另一个调用，则该函数可确保只写入一个抽样。所记录的抽样数可能会少于发出该调用的线程数。

  性能分析器不对由不同机制记录的抽样进行区分。如果只想查看 API 调用所记录的抽样，则应当在记录性能数据时关闭所有其他抽样模式。

- C、C++ 和 Fortran：`collector_pause()`

  Java: `CollectorAPI.pause()`

  停止将特定于事件的数据写入实验。实验将保持打开状态，并将继续写入全局数据。如果没有活动的实验或者已经停止记录数据，则该调用将被忽略。该函数停止写入所有特定于事件的数据，即使它是由 `collector_thread_resume()` 函数针对特定线程启用的也是如此。

- C、C++ 和 Fortran：`collector_resume()`

  Java：`CollectorAPI.resume()`

  在调用 `collector_pause()` 之后恢复将特定于事件的数据写入实验。如果没用活动的实验或数据记录功能处于活动状态，则该调用将被忽略。

- C、C++ 和 Fortran：`collector_terminate_expt()`

  Java：`CollectorAPI.terminate`

  终止正在收集其数据的实验。不再收集数据，但程序继续正常运行。如果没有活动的实验，则该调用将被忽略。

# 动态函数和模块

如果 C 或 C++ 程序向程序的数据空间动态编译函数，而且您希望在性能分析器中查看动态函数或模块的数据，那么，您必须向收集器提供信息。该信息由对收集器 API 函数的调用传递。API 函数的定义如下所示。

```
void collector_func_load(char *name, char *alias,
    char *sourcename, void *vaddr, int size, int lntsize,
    Lineno *lntable);
void collector_func_unload(void *vaddr);
```

您不必将这些 API 函数用于由 Java HotSpot™ 虚拟机编译的 Java 方法，该虚拟机使用的是另一个接口。Java 接口提供已编译到收集器的方法的名称。您可以查看 Java 编译方法的函数数据和带注释的反汇编列表，但不能查看带注释的源代码列表。

本节介绍 API 函数。

## `collector_func_load()` 函数

将有关动态编译的函数的信息传递到收集器，以便在实验中进行记录。下表对参数列表进行了描述。

表 3-1 `collector_func_load()` 的参数列表

| 参数 | 定义 |
|---|---|
| `name` | 性能工具所使用的动态编译函数的名称。该名称不必是函数的实际名称。虽然该名称不应包含嵌入的空格或引号字符，但无须遵循通常的函数命名约定。 |
| `alias` | 用于描述函数的任意字符串。它可以是 `NULL`。它不经过任何方式的解释，可以包含嵌入的空格。它显示在分析器的 "Summary"（摘要）标签中。它可用于指示函数的内容或动态构造函数的原因。 |
| `sourcename` | 构造函数时所在源文件的路径。它可以是 `NULL`。该源文件用于带注释的源代码列表。 |
| `vaddr` | 函数的装入地址。 |
| `size` | 以字节为单位的函数大小。 |
| `lntsize` | 对行号表中条目数量的计数。如果未提供行号信息，则计数应为零。 |
| `lntable` | 包含 `lntsize` 条目的表，其中每个条目都是一对整数。第一个整数是偏移量，第二个整数是行号。在一个条目的偏移量和下一个条目中所给出的偏移量之间的所有指令都归属于在第一个条目中提供的行号。偏移量必须按数字升序列出，但行号的顺序可以是任意的。如果 `lntable` 为 `NULL`，则没有可用的函数源代码列表，不过反汇编列表是可用的。 |

## `collector_func_unload()` 函数

通知收集器位于地址 `vaddr` 的动态函数已卸载。

# 数据收集的限制

本节描述了数据收集的限制，这些限制是由硬件、操作系统、程序的运行方式或收集器本身造成的。

对同时收集不同类型的数据来说，没有任何限制。您可以收集除计数数据以外的任何其他数据类型的数据。

收集器最多可支持 32,000 个用户线程。其他线程中的数据将被放弃，并生成收集器错误。要支持更多线程，请将 `SP_COLLECTOR_NUMTHREADS` 环境变量设置为更大的数字。

缺省情况下，收集器收集的堆栈深度最多为 256 帧。如果堆栈较深，则可能在 `er_print` 和性能分析器中看到 `<Truncated-stack>` 函数。有关更多信息，请参见“<Truncated-stack> 函数” [184]。要支持较深的堆栈，请将 `SP_COLLECTOR_STACKBUFSZ` 环境变量设置为更大的数字。

## 时钟分析的限制

用于分析的分析间隔最小值和时钟精度取决于特定的操作环境。最大值设置为 1 秒。分析间隔值将向下舍入到最接近的时钟精度的整数倍。要显示可能发现的最小值和最大值以及时钟精度，请键入不带其他参数的 `collect` 命令。

不能对使用分析计时器的程序执行时钟分析。收集器拦截对用于设置分析时钟参数的 `setitimer`(3) 的调用并阻止其他程序对其进行使用。

在 Linux 平台上，只能将时钟数据显示为 CPU 总时间。Linux CPU 时间是用户 CPU 时间和系统 CPU 时间的总和。

在 Linux 系统上，多线程应用程序的时钟分析所报告的线程数据可能不准确。内核并不总是按指定的间隔将分析信号传送到每个线程；有时，信号传送到了错误的线程。如果可用，使用周期计数器进行的硬件计数器分析通常提供更准确的线程数据。

### 时钟分析中的运行时失真和扩大

时钟分析记录当 `SIGPROF` 信号传递到目标时的数据。这将导致在处理该信号和展开调用堆栈时产生扩大。调用堆栈越深，信号越频繁，扩大越显著。在一定程度上，时钟分析表现出一些失真，这是由程序中那些执行最深堆栈的部分存在显著扩大而导致的。

请尽可能不要将缺省值设置为一个精确的毫秒数，而是将其设置为稍大于或稍小于某个精确数（例如，10.007 毫秒或 0.997 毫秒），以免与系统时钟关联，从而避免数据失真。在 Oracle Solaris 平台上，可以按照同样的方式来设置定制值（在 Linux 平台上不能设置定制值）。

# 收集跟踪数据的限制

只有在已预装入收集器库 `libcollector.so` 的情况下，才可以从已在运行的程序中收集任何种类的跟踪数据。有关更多信息，请参见“从正在运行的程序中收集跟踪数据” [77]。

## 跟踪中的运行时失真和扩大

跟踪数据使运行与被跟踪事件的数量成比例地扩大。如果完成了时钟分析，则跟踪事件所引起的扩大将导致时钟数据失真。

# 硬件计数器分析的限制

硬件计数器分析存在多种限制：

- 只能在具有硬件计数器且支持硬件计数器分析的处理器上收集硬件计数器数据。在其他系统中，硬件计数器分析功能处于禁用状态。具有 Unbreakable Enterprise Kernel 或 Red Hat 兼容内核 6.0 和更新版本的 Oracle Solaris 和 Oracle Linux 不支持硬件计数器。
- 在 `cpustat`(1) 命令运行过程中无法在运行 Oracle Solaris 的系统上收集硬件计数器数据，原因是 `cpustat` 控制了这些计数器，不允许用户进程使用它们。如果 `cpustat` 是在数据收集过程中启动的，则硬件计数器分析将终止，而且会在实验中记录一条错误。如果 `root` 使用硬件计数器启动 `er_kernel` 实验，也是如此。
- 如果正在执行硬件计数器分析，则无法使用自己代码中的硬件计数器。如果调用不是来自收集器，则收集器将插入 `libcpc` 库函数并返回一个返回值 `-1`。您应当对程序进行适当编码，使其在无法访问硬件计数器时能够正常工作。如果没有进行编码来对此加以处理，或者如果超级用户调用了同样使用计数器的系统范围的工具，或者如果该系统不支持计数器，程序将在硬件计数器分析过程中失败。
- 如果您试图通过向进程附加 `dbx` 来在使用硬件计数器库的正在运行的程序上收集硬件计数器数据，则实验可能会被破坏。

---

注 - 要查看所有可用计数器的列表，请运行不带其他参数的 `collect -h` 命令。

---

# 硬件计数器分析中的运行时失真和扩大

硬件计数器分析记录当 `SIGEMT` 信号（在 Solaris 平台上）或 `SIGIO` 信号（在 Linux 平台上）传递到目标时的数据。这将导致在处理该信号和展开调用堆栈时产生扩大。与时钟分析不同的是，对于某些硬件计数器，程序的某些部分可能会比其他部分更快地生成

事件并在该部分代码中显示扩大。程序中快速生成这类事件的任何部分都可能会显著失真。类似地，某些事件可能会在一个线程中与其他线程不成比例地生成。

## 子孙进程的数据收集限制

可以在某些限制下在子孙进程中收集数据。

如果要在收集器所跟踪的所有子孙进程中收集数据，就必须使用带有下列选项之一的 `collect` 命令：

- `-F on` 选项，允许对 `fork` 及其变体和 `exec` 及其变体调用以及所有其他子孙进程（包括那些因调用 `system`、`popen`、`posix_spawn`(3p)、`posix_spawnp`(3p) 和 `sh` 而产生的进程）自动收集数据。
- `-F all` 与 `-F on` 相同。
- `-F '=`*regexp*`'` 选项，允许对其名称与指定的正则表达式相匹配的所有子孙进程收集数据。

有关 `-F` 选项的更多信息，请参见“实验控制选项” [63]。

## OpenMP 分析的限制

在执行程序期间收集 OpenMP 数据开销可能很大。可以通过设置 `SP_COLLECTOR_NO_OMP` 环境变量来压低开销。如果这样做，程序执行时间将显著缩短，但是将看不到自从属线程向上传播到调用方并最终传播到 `main（）` 的数据，而在未设置该变量时通常情况下将能看到这些数据。

OpenMP 分析功能仅对使用 Oracle Solaris Studio 编译器编译的应用程序可用，因为它取决于 Oracle Solaris Studio 编译器运行时。对于使用 GNU 编译器编译的应用程序，仅显示计算机级别的调用堆栈。

## Java 分析的限制

可以在下列限制下在 Java 程序中收集数据：

- 应当使用版本不低于 JDK 7 Update 25 (JDK 1.7.0_25) 的 Java2 Software Development Kit (JDK)。收集器先在 `JDK_HOME` 环境变量或 `JAVA_PATH` 环境变量中设置的路径中查找 JDK。如果未设置这些变量，它将在 `PATH` 中查找 JDK。如果 `PATH` 中没有 JDK，它将在 `/usr/java/bin/java` 中查找 `java` 可执行文件。

  收集器将验证它找到的 `java` 可执行文件的版本是 ELF 可执行文件。如果不是，则将输出错误消息，指示所使用的环境变量或路径，以及所尝试的全路径名。
- 必须使用 `collect` 命令来收集数据。无法使用 `dbx collector` 子命令。

- 如果应用程序所创建的子孙进程运行 JVM 软件，则不能对这些应用程序进行分析。
- 某些应用程序不是纯 Java，而是 C 或 C++ 应用程序，它们调用 `dlopen（）` 以装入 `libjvm.so`，然后通过调用 JVM 软件来启动 JVM 软件。要分析此类应用程序，请设置 `SP_COLLECTOR_USE_JAVA_OPTIONS` 环境变量，并将 `-j on` 选项添加到 `collect` 命令行。对于这种情况，请不要设置 `LD_LIBRARY_PATH` 环境变量。
- 如果目标是 JVM 计算机，必须使用 `-j on` 才能获取分析数据。如果目标是类文件或 jar 文件，则不需要 `-j on` 选项。如果使用 64 位 JVM 计算机，您必须将其路径显式指定为目标；对于 32 位 JVM 计算机，请勿使用 `-d64` 选项。如果指定了 `-j on` 选项，但目标不是 JVM 计算机，可能会向目标传递一个无效参数，且不会记录任何数据。`collect` 命令将验证为 Java 分析指定的 JVM 计算机的版本。

## 用 Java 编程语言所编写的应用程序的运行时性能失真和扩大

Java 分析功能使用的Java虚拟机工具接口 (Java Virtual Machine Tool Interface, JVMTI) 可能会导致运行的失真和扩大。

对于时钟分析和硬件计数器分析，数据收集进程会对 JVM 软件进行各种调用，并使用信号处理程序处理分析事件。这些例程的开销和将实验写入磁盘的代价将扩大 Java 程序的运行时。这种扩大通常小于 10%。

# 数据的存储位置

在应用程序的一次执行过程中所收集的数据称作实验。实验由存储在目录下的一组文件组成。实验的名称即目录的名称。

除记录实验数据以外，收集器还为程序所使用的装入对象创建自己的归档文件。这些归档文件包含装入对象中每个对象文件和函数的地址、大小和名称以及装入对象的地址和上次修改的时间戳。归档还可能包含所有共享对象及某些或所有源文件的副本。有关更多信息，请参见“`er_archive` 实用程序” [214]。

缺省情况下，实验存储在当前目录中。如果该目录位于网络文件系统上，则存储数据所需的时间比在本地文件系统中长，而且可能会导致性能数据失真。如有可能，应始终尝试在本地文件系统中记录实验。可以在运行收集器时指定存储位置。

子孙进程的实验存储在创建者进程的实验内部。

## 实验名称

新实验的缺省名称为 `test.1.er`。后缀 `.er` 是必需的：如果您提供的名称不具有该后缀，则系统会显示一条错误消息而且不接受该名称。

如果您选择使用格式为 *experiment*.*n*.er 的名称，（其中 *n* 是正整数），则收集器会将后续实验名称中的 *n* 自动递增 1。例如，mytest.1.er 的后面是 mytest.2.er、mytest.3.er 等。如果实验已经存在，收集器也会递增 *n*，直到找到未使用的实验名称才停止递增 *n*。如果实验名称不含 *n* 且实验存在，则收集器会输出一条错误消息。

子实验遵循相同的命名规则。有关更多信息，请参见“子实验” [160]。

## 实验组

实验可按组收集。组在实验组文件中定义，缺省情况下该文件存储在当前目录中。实验组文件是纯文本文件，它具有特殊的标题行，并在随后的每一行中显示实验名称。实验组文件的缺省名称为 test.erg。如果该名称不以 .erg 结尾，则将显示错误而且不接受该名称。创建实验组后，使用该组名称运行的所有实验都会添加到该组。

要手动创建实验组文件，请创建首行为以下内容的纯文本文件：

```
#analyzer experiment group
```

然后将实验名称添加到随后的行中。文件的扩展名必须为 .erg。

还可以通过使用带有 -g 参数的 collect 命令来创建实验组。

## 子孙进程的实验

子孙进程的实验是按世系命名的，如下所示。在形成子孙进程的实验名称时，将下划线、代码字母和数字添加到其创建者实验名称的主干中。代码字母 f 表示派生，x 表示执行，c 表示组合。数字是派生或执行的索引（无论是否成功）。例如，如果创建者进程的实验名称为 test.1.er，则在第三次调用 fork 时为子进程创建的实验为 test.1.er/_f3.er。如果子进程成功调用 exec，则新子孙进程的实验名称为 test.1.er/_f3_x1.er。

## MPI 程序的实验

缺省情况下，MPI 程序的数据收集到 test.1.er 中，而 MPI 进程的所有数据都收集到子实验中，每个等级一个子实验。收集器使用 MPI 等级以格式 M_r*m*.er 构造子实验名称，其中 *m* 是 MPI 等级。例如，MPI 等级 1 的子实验数据将记录在 test.1.er/M_r1.er 目录中。

## 内核和用户进程上的实验

缺省情况下，内核上的实验命名为 ktest.1.er 而不是 test.1.er。当同时收集用户进程的数据时，内核实验将包含每个跟随的用户进程所对应的子实验。

子实验使用 _*process-name*_PID_*process-id*.1.er 格式命名。例如，在进程 ID 1264 下运行的 sshd 进程上所运行的实验将命名为 ktest.1.er/_sshd_PID_1264.1.er。

## 移动实验

如果要将实验移到其他计算机以便对其进行分析，则应了解分析与在其中记录实验的操作环境的相关性。

实验包含计算函数级度量和显示时间线所必需的全部信息。但是，如果要查看带注释的源代码或带注释的反汇编代码，则必须能够访问与用于生成目标或实验的装入对象或源文件相同的版本。

有关用于查找实验源代码的过程的说明，请参见“工具如何查找源代码” [189]。

要确保看到程序正确的带注释的源代码和带注释的反汇编代码，可以在移动或复制实验之前使用 er_archive 命令将源代码、对象文件和可执行文件复制到实验中。

有关更多信息，请参见 er_archive(1) 手册页。

# 估计存储要求

本节就如何对记录实验所需的磁盘空间量进行估计提供了一些指导。实验的大小直接取决于数据包的大小、记录数据包的速率、程序使用的 LWP 的数量和程序的执行时间。

数据包中包含特定于事件的数据和取决于程序结构（调用堆栈）的数据。取决于数据类型的数据量约为 50 到 100 个字节。调用堆栈数据由每个调用的返回地址组成，每个地址包含 4 个字节（在 64 位可执行文件中为 8 个字节）。记录了实验中的每个线程的数据包。请注意，对于 Java 程序，有两个相关的调用堆栈：Java 调用堆栈和计算机调用堆栈，因此将导致向磁盘中写入更多的数据。

记录分析数据包的速率由时钟数据的分析间隔和硬件计数器数据的溢出值控制，对于跟踪函数，是跟踪函数出现的速率。对于分析间隔参数的选择也会因数据收集的开销而影响数据质量和程序性能的失真。这些参数的值越小，提供的统计信息越好，但开销也会越大。为了在获得较好统计信息和尽可能降低开销之间实现折衷，我们已经为分析间隔和溢出值选择了缺省值。值越小，数据越多。

对于分析间隔大约为每秒 100 个抽样、数据包大小范围从 80 字节（对于小调用堆栈）到最多 120 字节（对于大调用堆栈）的时钟分析实验或硬件计数器分析实验，将以每个线程每秒 10 千字节的速率记录数据。调用堆栈深度为数百个调用的应用程序可以很轻松地以十倍的速率记录数据。

对于 MPI 跟踪实验，数据量为每个跟踪 MPI 调用 100-150 字节，具体取决于发送的消息数和调用堆栈的深度。此外，在使用 collect 命令的 -M 选项时，缺省情况下会启用时

钟分析，因此，请为时钟分析实验添加估计的数字。可以通过使用 `-p off` 选项禁用时钟分析来减少 MPI 跟踪的数据量。

---

注 - 收集器以自己的格式 (`mpview.dat3`) 存储 MPI 跟踪数据，还以 VampirTrace OTF 格式 (`a.otf`, `a.*.z`) 存储 MPI 跟踪数据。可以删除 OTF 格式的文件而不影响性能分析器。

---

在估计实验大小时，还应当考虑归档文件所占用的磁盘空间（请参见“数据的存储位置” [50]）。如果您无法确定所需空间的大小，请尝试运行实验一小会儿。通过该测试，可以得到与数据收集时间无关的归档文件大小，然后对分析文件的大小进行放大以获得全长实验的估计大小。

归档还可能包含所有共享对象及某些或所有源文件的副本。有关更多信息，请参见“er_archive 实用程序” [214]。

除了分配磁盘空间之外，收集器还在内存中分配缓冲区，以便在将分析数据写入磁盘之前对其进行存储。目前无法指定这些缓冲区的大小。如果收集器用完了内存，请尝试减少所收集的数据量。

如果存储实验所需的估计空间大于可用空间，请考虑收集部分运行（而不是全部运行）的数据。要收集部分运行的数据，可以使用带有 `-y` 或 `-t` 选项的 `collect` 命令或 `dbx collector` 子命令，也可以在程序中插入对收集器 API 的调用。还可以对由带有 `-L` 选项的 `collect` 命令或 `dbx collector` 子命令所收集的分析和跟踪数据总量进行限制。

请参见文章 Data Selectivity and the Oracle Solaris Studio Performance Analyzer（数据选择性和 Oracle Solaris Studio 性能分析器），以了解有关选择性数据收集和分析的信息。

# 收集数据

您可以使用以下多种方式收集用户模式目标的性能数据：

- 在命令行上使用 `collect` 命令（请参见“使用 collect 命令收集数据” [54]和 `collect`(1) 手册页）。`collect` 命令行工具的数据收集开销比 `dbx` 的开销小，因此该方法比其他方法要好。
- 使用性能分析器中的 "Profile Application"（分析应用程序）对话框。（请参见性能分析器帮助中的 "Profiling an Application"（分析应用程序）。）
- 在 `dbx` 命令行上使用 `collector`。（请参见“使用 dbx collector 子命令收集数据” [70]）。

只能从 "Profile Application"（分析应用程序）对话框和 `collect` 命令自动收集子孙进程的数据。

您可以使用 `er_kernel` 实用程序或 "Profile Kernel"（分析内核）对话框收集 Oracle Solaris 内核的性能数据。有关更多信息，请参见第 9 章 内核分析。

# 使用 collect 命令收集数据

要从命令行使用 collect 命令运行收集器，请键入以下内容。

```
% collect collect-options program program-arguments
```

*collect-options* 是 collect 命令选项，*program* 是要收集其数据的程序的名称，*program-arguments* 是该程序的参数。目标程序通常是二进制可执行文件或脚本。

如果执行无参数调用，collect 将显示用法摘要，其中包含实验的缺省配置。

要获取选项列表和任何可用于分析的硬件计数器名称列表，请不带其他参数键入 collect -h 命令。

```
% collect -h
```

有关对硬件计数器列表的描述，请参见“硬件计数器分析数据” [23]。另请参见“硬件计数器分析的限制” [48]。

## 数据收集选项

这些选项控制所收集数据的类型。有关对数据类型的介绍，请参见“收集器收集的数据” [19]。

如果未指定数据收集选项，则缺省值为 -p on，这会启用缺省分析间隔大约为 10 毫秒的时钟分析。

如果您使用 -p off 明确禁用了时钟分析，而且未启用跟踪或硬件计数器分析，则 collect 命令会输出一条警告消息，并且只收集全局抽样数据。

### 使用 -p *option* 选项收集时钟分析数据

使用 -p 选项可以收集时钟分析数据。*option* 的允许值包括：

- off – 关闭时钟分析。
- on – 打开缺省分析间隔大约为 10 毫秒的时钟分析。
- lo[w] – 打开分析间隔大约为 100 毫秒（低精度）的时钟分析。
- hi[gh] – 打开分析间隔大约为 1 毫秒（高精度）的时钟分析。有关启用高精度分析的信息，请参见“时钟分析的限制” [47]。
- [+]*value* – 打开时钟分析并将其分析间隔设置为 *value*。*value* 的缺省单位为毫秒。可以将 *value* 指定为整数或浮点数。可以选择在数值后加后缀 m 来选择毫秒单位或者加 u 来选择微秒单位。该值应当是时钟精度的倍数。如果该值较大但不是精度的倍数，则会向下舍入。如果较小，则会输出一条警告消息并将其设置为时钟精度。

collect 命令的缺省操作是收集时钟分析数据。如果不收集计数数据 (-c) 或数据争用和死锁数据 (-r)，即使未指定 -p 选项，也会收集时钟分析数据。

如果指定 -h high 或 -h low 以高频率或低频率请求为该处理器设置的缺省硬件计数器，则缺省的时钟分析也将设置为高频率或低频率。您可以为时钟分析设置其他频率，方法是使用 -p hi、-p low 或 -p *n* 选项明确设置它。

有关 Linux 上多线程应用程序的时钟分析的说明，请参见“时钟分析的限制” [47]。

## 使用 collect -h 收集硬件计数器分析数据

收集硬件计数器分析数据。对于某些硬件，collect 命令定义了一个缺省计数器集，您可以使用不带任何参数的 collect -h 显示它。还可以指定特定计数器而不使用缺省计数器集。可以使用多个 -h 参数指定计数器。

有关使用 collect -h 显示的计数器格式的信息，请参见“硬件计数器列表” [24]。

允许和 collect -h 结合使用的 *option* 值包括：

| | |
|---|---|
| off | 关闭硬件计数器分析。任何其他选项不能与 -h off 一同指定。 |
| on | 对于特定硬件系统，打开使用缺省计数器集的硬件计数器分析。如果系统没有缺省计数器集，指定 -h on 时会生成错误。 |
| hi \| high | 为系统打开使用缺省计数器集的硬件计数器分析，并以高速率进行分析。如果系统没有缺省计数器集，指定 -h hi 时会生成错误。 |
| lo \| low | 为系统打开使用缺省计数器集的硬件计数器分析，并以低速率进行分析。如果系统没有缺省计数器集，指定 -h lo 时会生成错误。 |
| *ctr_def*... [,*ctr_n_def*] | 使用一个或多个指定的计数器收集硬件计数器分析。支持的计数器最大数量（*ctr_def* 到 *ctr_n_def*）取决于处理器。在当前计算机上运行不带任何参数的 collect -h 可确定用于分析的硬件计数器定义的最大数量，并显示可用的硬件计数器和缺省计数器集的完整列表。<br>内存相关计数器是指类型为 load、store 或 load-store 的计数器；在运行不带任何其他命令行参数的 collect -h 命令时计数器列表中会显示这些计数器。某些这类计数器还带有 precise 标签。对于 SPARC 或 x86 上的 precise 计数器，缺省情况下将记录数据空间和内存空间。<br>每个 *ctr_def* 计数器定义都采用以下格式：<br>*ctr*[~*attr*=*val*]...[~*attrN*=*valN*][/*reg#*],[*interval*]<br>计数器定义选项的含义如下所示： |

*ctr*

通过运行不带任何其他命令行参数的 `collect -h` 命令显示的处理器特定的计数器名称。在大多数系统上，即使未列出某计数器，仍可使用数值指定该计数器；指定的数值为十六进制 (0x1234) 或十进制。早期芯片的驱动程序不支持数值，但近期芯片的驱动程序支持。以数字方式指定计数器时，也应该指定寄存器号。要使用的数值可在芯片特定制造商的手册中找到。手册名称在 `collect -h` 输出中指定。有些计数器仅在专有供应商手册中进行介绍。

*~attr=val*

可选的一个或多个属性选项。在某些处理器上，可以将多个属性选项与一个硬件计数器关联。如果处理器支持多个属性选项，则运行不带任何其他命令行参数的 `collect -h` 也会列出用于 *~attr* 的属性名称。值 *val* 可以采用十进制或十六进制格式。十六进制格式的数字使用 C 程序格式，其中的数字前置了零和小写 x (0x*hex_number*)。大多数属性都与计数器名称串联。每个属性名称前面需要 ~。

*/reg#*

要用于计数器的硬件寄存器。如果未指定寄存器，则 `collect` 会尝试将计数器放入第一个可用的寄存器，因此，可能由于寄存器冲突而无法放置后续计数器。如果指定了多个计数器，则这些计数器必须使用不同的寄存器。通过运行不带任何其他命令行参数的 `collect -h` 命令，您可以查看允许的寄存器号列表。如果指定了寄存器，则必须使用 / 字符。

*interval*

间隔是抽样频率，可以设置为下列值之一：

- `on` – 缺省间隔，可以通过键入不带其他参数的 `collect -h` 来确定。请注意，所有原始计数器的缺省值都是相同的，并且可能不是最合适特定计数器的值。
- `hi` – 所选计数器的高精度值，大约是缺省值的十分之一。
- `lo` – 所选计数器的低精度值，大约是缺省值的十倍。
- *value* – 特定的值，必须是正整数，可以采用十进制格式，也可以采用十六进制格式。

如果省略间隔，则使用 `-h on` 的值。但是，如果省略间隔，必须仍在省略的间隔指定部分前

提供逗号，除非是在 -h 参数中指定最后一个计数器。

对于原始计数器，hi、lo 和 on 的值是猜测值，但是对于任何特定程序，很难猜测适当的间隔。如果为任何原始计数器指定 on/hi/lo，并且事件分别快于每线程每秒 100/1000/10，则会将间隔向下调节为 Oracle Solaris 系统上更合理的最大值。

例 3-1 -h 用法的有效示例

```
-h on
-h lo
-h hi
      Enable the default counters with default, low, or
  high rates, respectively

-h cycles,,insts,,dcm
-h cycles -h insts -h dcm
     Both have the same meaning: three counters: cycles, insts
 and dataspace-profiling of D-cache misses (SPARC only)

-h cycles~system=1
     Count cycles in both user and system modes

-h 0xc0/0,10000003
    On Nehalem, that is the equivalent to
-h inst_retired.any_p/0,10000003
```

例 3-2 -h 用法的无效示例

```
-h cycles -h off
      Can't use off with any other -h arguments
-h cycles,insts
     Missing comma, and "insts" does not parse as a number for
  <interval>
```

如果 -h 参数指定使用硬件计数器，但硬件计数器在命令发出时正由根使用，则 collect 命令将报告错误，且不会运行实验。

如果未指定 -h 参数，则不收集硬件计数器分析数据。一个实验可以同时指定硬件计数器溢出分析和基于时钟的分析。指定硬件计数器溢出分析不会禁用基于时钟的分析（即使已缺省启用）。

## 使用 **-s** *option* 收集同步等待跟踪数据

收集同步等待跟踪数据。*option* 的允许值包括：

`on`
: 打开同步延迟跟踪并在运行时通过校准来设置阈值

`calibrate`
: 与 `on` 相同。

`off`
: 关闭同步延迟跟踪

*n*
: 打开同步延迟跟踪并将阈值设置为 *n* 微秒；如果 *n* 为零，则跟踪所有事件

`all`
: 打开同步延迟跟踪并跟踪所有同步事件
缺省情况下，会关闭同步延迟跟踪。

对于 Java 程序，将记录用户代码中 Java 监视器的同步事件，而不会记录 JVM 计算机内的本机同步调用。

在 Oracle Solaris 上，跟踪以下函数：

```
mutex_lock()
rw_rdlock()
rw_wrlock()
cond_wait()
cond_timedwait()
cond_reltimedwait()
thr_join()
sema_wait()
pthread_mutex_lock()
pthread_rwlock_rdlock()
pthread_rwlock_wrlock()
pthread_cond_wait()
pthread_cond_timedwait()
pthread_cond_reltimedwait_np()
pthread_join()
sem_wait()
```

在 Linux 上，跟踪以下函数：

```
pthread_mutex_lock ()
pthread_cond_wait ()
pthread_cond_timedwait ()
pthread_join ()
sem_wait ()
```

## 使用 **-H** *option* 收集堆跟踪数据

收集堆跟踪数据。*option* 的允许值包括：

`on` 打开对内存分配请求的跟踪

`off` 打开对内存分配请求的跟踪

缺省情况下，堆跟踪功能处于关闭状态。

将记录任何本机调用的堆跟踪事件，对 `mmap` 的调用被视为内存分配。

对于 Java 程序，不支持堆分析。指定将其视为一个错误。

请注意，堆跟踪可能会生成非常大的实验。此类实验的装入和浏览速度都非常慢。

## 使用 **-i** *option* 收集 I/O 跟踪数据

收集 I/O 跟踪数据。允许的选项值包括：

`on` 打开对 I/O 操作的跟踪

`off` 关闭对 I/O 操作的跟踪

缺省情况下不执行 I/O 跟踪。I/O 跟踪会产生非常大的实验，此类实验的装入和浏览速度非常慢。

## 使用 **-c** *option* 对数据计数

记录计数数据，仅针对 Oracle Solaris 系统。

*option* 的允许值包括：

`on` 打开计数数据。

`static` 在假定每个指令执行了一次的情况下，打开模拟计数数据。

`off` 关闭计数数据。

缺省情况下，关闭对计数数据的收集。计数数据不能和任何其他类型的数据一起收集。

对于计数数据和模拟计数数据，将对可执行文件以及检测过的和静态链接的任何共享对象进行计数。对于计数数据而非模拟计数数据，还将检测和计数动态装入的共享对象。

在 Oracle Solaris 上，不需要特殊的编译，但计数选项与编译标志 `-p`、`-pg`、`-qp`、`-xpg` 和 `--xlinkopt` 不兼容。

在 Linux 上，可执行文件必须使用 `-annotate=yes` 标志进行编译，才能收集计数数据。

在 Oracle Linux 5 上，运行时链接程序审计接口（也称为 rtld-audit 或 LD_AUDIT）的不稳定性可能会阻止计数数据的收集。

## 使用 **-I** *directory* 指定计数数据检测目录

为计数数据检测指定目录。该选项仅在 Oracle Solaris 系统上可用，且仅当指定了 `-c` 选项时才有意义。

## 使用 **-N** *library-name* 指定排除的库

指定要从计数数据检测中排除的库，不管该库是链接到可执行文件还是使用 `dlopen`（）装入。该选项仅在 Oracle Solaris 系统上可用，且仅当指定了 `-c` 选项时才有意义。可以指定多个 `-N` 选项。

## 使用 **-S** *option* 对数据抽样

定期记录抽样包。*option* 的允许值包括：

`off` 关闭定期抽样功能。

`on` 打开使用缺省抽样间隔（1 秒）的定期抽样功能。

*n* 打开抽样间隔为 *n* 秒的定期抽样功能；*n* 必须是正数。

缺省情况下，启用间隔为 1 秒的定期抽样功能。

如果未提供数据指定参数，将使用缺省精度执行时钟分析。

如果已显式禁用时钟分析，且未启用硬件计数器溢出分析或任何类型的跟踪，则 `collect` 会显示警告说明未收集函数级别的数据，然后执行目标并记录全局数据。

## 使用 -r *option* 收集数据争用和死锁检测数据

为线程分析器收集数据争用检测或死锁检测数据。

*option* 的允许值包括：

`race`
: 收集数据以检测数据争用。

`deadlock`
: 收集数据以检测死锁和可能的死锁。

`all`
: 收集数据以检测数据争用、死锁和可能的死锁。也可以指定为 `race,deadlock`。

`off`
: 关闭对数据争用、死锁和可能的死锁的数据收集。

`on`
: 收集数据以检测数据争用（与 `race` 相同）。

`terminate`
: 如果检测到不可修复的错误，将终止目标进程。

`abort`
: 如果检测到不可修复的错误，则终止带有信息转储的目标进程。

`continue`
: 如果检测到不可修复的错误，将允许进程继续。

缺省情况下，关闭对所有线程分析器数据的收集。`terminate`、`abort` 和 `continue` 选项可添加到任何数据收集选项，并控制发生不可修复的错误（例如，实际（而不是潜在）死锁）时的行为。缺省行为是 `terminate`。

线程分析器数据不能与任何跟踪数据一起收集，但可以与时钟或硬件计数器分析数据一起收集。线程分析器数据会明显减缓目标执行速度，且分析数据可能对用户代码没有意义。

可以使用 `analyzer` 或 `tha` 对线程分析器实验进行检查。线程分析器 (`tha`) 显示一组简化的缺省数据视图，但其实是一样的。

在启用数据争用检测之前，必须在编译时或通过调用后处理程序来检测可执行文件。如果未检测目标，也未检测其库列表中的任何共享对象，则会显示一个警告，但会运行实验。其他线程分析器数据不需要检测。

有关 collect -r 命令和线程分析器的更多信息，请参见《Oracle Solaris Studio 12.4：线程分析器用户指南 》和 tha(1) 手册页。

## 使用 -M *option* 进行 MPI 分析

指定对 MPI 实验的收集。collect 命令的目标必须为 mpirun 命令，必须使用 –选项将 mpirun 的选项与要使用 mpirun 命令运行的目标程序分开。（始终将 -- 选项和 mpirun 命令一起使用，以便可以通过将 collect 命令及其选项前置于 mpirun 命令行来收集实验。）实验按通常方式命名且称为创建者实验，其目录包含每个 MPI 进程的子实验，按等级进行命名。

*option* 的允许值包括：

*MPI-version*

打开 MPI 实验的收集功能并假定指定的 MPI 版本。当您键入不带任何参数的 collect 时或响应使用 -M 指定的无法识别的版本时，将输出识别的 MPI 版本。

off

关闭对 MPI 实验的收集。

缺省情况下，MPI 实验的收集功能是关闭的。打开 MPI 实验的收集功能时，-m 选项的缺省设置更改为 on。

键入不带其他选项的 collect -h 命令时，或者如果使用 -M 选项指定无法识别的版本，将会输出受支持的 MPI 版本。

## 使用 -m *option* 收集 MPI 跟踪数据

收集 MPI 跟踪数据。

*option* 的允许值包括：

on　　打开 MPI 跟踪信息。

off　　关闭 MPI 跟踪信息。

缺省情况下关闭 MPI 跟踪，除非启用了 -M 选项，在这种情况下会缺省打开 MPI 跟踪。通常使用 -M 选项收集 MPI 实验，无需用户对 MPI 跟踪进行控制。如果要收集 MPI 实验，但不收集 MPI 跟踪数据，请使用显式选项 -M *MPI-version* -m off。

有关其调用被跟踪的 MPI 函数以及根据跟踪数据计算的度量的更多信息，请参见“MPI 跟踪数据” [29]。

# 实验控制选项

这些选项控制如何收集实验数据。

## 使用 **-L** *size* 限制实验大小

将所记录的分析数据量限制在 *size* 兆字节。该限制适用于所有分析数据量和跟踪数据量之和，但不适用于抽样点。该限制只是近似值，可以被超出。

当达到该限制时，不再记录分析和跟踪数据，但实验会一直保持打开状态，直到目标进程终止。如果启用了定期抽样，则会继续写入抽样点。

*size* 的允许值包括：

`unlimited` 或 `none`
: 不对实验强加大小限制。

*n*
: 强加 *n* MB 限制。*n* 的值必须为正数（大于零）。

缺省情况下，记录的数据量不存在限制。

例如，要将限制确定为约 2 GB，请指定 `-L 2000`。

## 使用 **-F** *option* 跟踪进程

控制子孙进程是否应记录数据。始终会收集创建者进程的数据，这和 `-F` 设置无关。*option* 的允许值包括：

`on`
: 针对所有子孙进程记录实验。

`all`
: 与 `on` 相同。

`off`
: 不针对任何子孙进程记录实验。

=*regex*
: 针对可执行文件名称与正则表达式匹配的子孙进程记录实验。仅使用可执行文件的基名而不是完整路径。如果您使用的 *regex* 包含空白或 shell 解释的字符，请务必将整个 *regex* 参数括在单引号内。

缺省情况下设置 -F on 选项，这样收集器将跟踪通过调用函数 fork(2)、fork1(2)、fork(3F)、vfork(2) 和 exec(2) 及其变体而创建的进程。对 vfork 的调用已在内部被替换为对 fork1 的调用。

还将跟踪通过调用 system(3C)、system(3F)、sh(3F)、posix_spawn(3p)、posix_spawnp(3p) 和 popen(3C) 以及类似函数而创建的子孙进程以及与其相关的子孙进程。

在 Linux 上，缺省情况下将跟踪由不带 CLONE_VM 标志的 clone(2) 创建的子孙进程。使用 CLONE_VM 标志创建的子孙进程被视为例程而不是进程，会始终跟踪，与 -F 设置无关。

如果指定 -F '=*regexp* ' 选项，收集器将跟踪所有子孙进程。当子孙进程的名称或子实验的名称与指定的正则表达式匹配时，收集器将创建子实验。有关正则表达式的信息，请参见 regexp(5) 手册页。

使用正则表达式的示例：

- 要针对创建者进程中第一个系统调用的第一个 fork 的第一个 exec 的子孙进程捕获数据，请使用： collect -F '=_x1_f1_x1'
- 要捕获 exec 而不是 fork 的所有变体的数据，请使用：collect -F '=.*_x[0-9]/*'
- 要捕获系统 ("echo hello") 而非系统 ("goodbye") 调用的数据，请使用：collect -F '=echo hello'

有关如何创建和命名子孙进程的实验的更多信息，请参见“子实验” [160]。

对于 MPI 实验，缺省情况下还跟踪子孙进程。

读取创建者实验时，性能分析器和 er_print 实用程序将自动读取子孙进程的实验，并在数据显示中显示子孙进程。

要从命令行选择用于显示的特定子实验的数据，请将子实验路径名显式指定为 er_print 或 analyzer 命令的参数。所指定的路径必须包含创建者实验的名称以及创建者目录中子孙实验的名称。

例如，要查看 test.1.er 实验的第三个 fork 的数据：

```
er_print test.1.er/_f3.er

analyzer test.1.er/_f3.er
```

或者，可以使用感兴趣的子孙实验的显式名称来准备实验组文件。有关更多信息，请参见“实验组” [51]。

---

注 - 如果正在跟踪子孙进程时创建者进程退出，将继续从仍然在运行的子孙进程中收集数据。创建者实验目录会相应地继续变大。

---

您还可以收集脚本上的数据并跟踪脚本的子孙进程。有关更多信息，请参见“从脚本收集数据” [78]。

## 使用 -j *option* 分析 Java

当目标程序是 JVM 时，启用 Java 分析。*option* 的允许值包括：

on
: 记录 JVM 计算机的分析数据，并识别 Java HotSpot 虚拟机编译的方法以及记录 Java 调用堆栈。

off
: 不记录 Java 分析数据。

*path*
: 记录 JVM 的分析数据并使用安装在指定 *path* 中的 JVM。

如果目标是 JVM 计算机，必须使用 -j on 才能获取分析数据。

如果您要收集 .class 文件或 .jar 文件中的数据，则不需要使用 -j on 选项。

如果使用 64 位 JVM 计算机，您必须将其路径显式指定为目标；对于 32 位 JVM 计算机，请勿使用 -d64 选项。如果指定了 -j on 选项，但目标不是 JVM 计算机，可能会向目标传递一个无效参数，且不会记录任何数据。collect 命令将验证为 Java 分析指定的 JVM 计算机的版本。

## 使用 -J *java-argument* 传递 Java 选项

指定要传递到用于分析的 JVM 的其他参数。如果指定了 -J 选项，将启用 Java 分析 (-j on)。如果 *java-argument* 包含多个参数，则必须使用引号将其括起来。它必须包括一组由空格或制表符分隔的令牌。每个令牌作为单独的参数传递给 JVM。JVM 的大多数参数必须以 (-) 短划线字符开头。

## 使用 -l *signal* 指定抽样的信号

当名为 *signal* 的信号传递到进程时，记录抽样包。

可以通过全信号名、不带开始的几个字母 SIG 的信号名或信号编号来指定信号。请不要使用程序所使用的信号或会终止执行的信号。建议的信号为 SIGUSR1 和 SIGUSR2。即使指定时钟分析，也可以使用 SIGPROF。可通过 kill 命令将信号传递到进程。

如果同时使用 -l （小写 L）和 -y 选项，则各选项必须使用不同的信号。

如果您使用该选项而程序具有自己的信号处理程序，则应当确保使用 -l 指定的信号会传递到收集器的信号处理程序，而不是被拦截或忽略。

有关信号的更多信息，请参见 signal(3HEAD) 手册页。

## 使用 -t *duration* 设置时间范围

指定数据收集的时间范围。

可以将 *duration* 指定为单个数字（可以选择添加 m 或 s 后缀）以指示实验在该时间（单位为分钟或秒）终止。缺省情况下，持续时间以秒为单位。也可以将 *duration* 指定为用连字符分隔的两个数字，这会导致数据收集暂停，直到第一个时间过去。到那时，将开始收集数据。当到达第二个时间时，数据收集终止。如果第二个数字为零，则在初次暂停之后收集数据，直到该程序运行结束。即使该实验已经终止，也允许目标进程运行至结束。

## 停止分析的目标以允许使用 -x 连接 dbx

从 exec 系统调用退出时使目标进程保持停止状态，以允许调试器连接到该进程。collect 命令将输出一条含有进程 ID 的消息。

要在 collect 停止调试器之后将其添加到目标，您可以按照下文中的过程进行操作。

1. 从collect -x命令输出的消息获取进程的 PID。
2. 启动调试器
3. 将调试器配置为忽略 SIGPROF；如果您选择收集硬件计数器数据，则配置为忽略 Solaris 上的 SIGEMT 或 Linux 上的 SIGIO
4. 使用 dbx attach 命令连接到进程。
5. 为您要收集的实验设置收集器参数
6. 发出 collector enable 命令。
7. 发出 cont 命令以允许目标进程运行

由于进程在调试器的控制之下运行，因此收集器会记录一个实验。

或者，您可以连接到进程并使用 collect -P *PID* 命令收集实验。

## 使用 -y *signal* [,r] 指定信号暂停和恢复状态

控制对包含名为 *signal* 的信号的数据的记录。无论何时将信号传递到进程，它都在暂停状态（在此期间不记录任何数据）和记录状态（在此期间记录数据）之间切换。

缺省情况下，数据收集将在暂停状态下开始。如果指定了可选的 ,r 标志，数据收集将在恢复状态下开始，这表示将立即进行分析。无论 -y 选项的状态如何，都将始终记录抽样点。

暂停-恢复信号的一个用途是在不收集数据的情况下启动一个目标，让其达到稳定状态，然后再启用数据收集。

可以通过全信号名、不带开始的几个字母 SIG 的信号名或信号编号来指定信号。请不要使用程序所使用的信号或会终止执行的信号。建议的信号为 SIGUSR1 和 SIGUSR2。即使指定时钟分析，也可以使用 SIGPROF。可通过 kill 命令将信号传递到进程。

如果同时使用 -l 和 -y 选项，则必须针对每个选项使用不同的信号。

使用 -y 选项时，如果已提供可选的 r 参数，则收集器将在记录状态下启动，否则将在暂停状态下启动。如果未使用 -y 选项，则收集器将在记录状态下启动。

如果您使用该选项而程序具有自己的信号处理程序，则应当确保使用 -y 指定的信号会传递到收集器的信号处理程序，而不是被拦截或忽略。

有关信号的更多信息，请参见 signal(3HEAD) 手册页。

# 输出选项

这些选项控制收集器生成实验的各个方面。

## 使用 **-o** *experiment-name* 设置实验名称

使用 *experiment-name* 作为要记录的实验的名称。*experiment-name* 字符串必须以字符串 ".er" 结尾；否则 collect 实用程序会输出一条错误消息并退出。

如果不指定 -o 选项，则为实验提供一个 *stem*.*n*.er 格式的名称，其中 *stem* 是字符串，*n* 是数字。如果使用 -g 选项指定了组名称，则将 *stem* 设置为不带 .erg 后缀的组名称。如果未指定组名称，则将 *stem* 设置为字符串 test。

如果您要从用于运行 MPI 作业的命令之一（例如 mpirun，但不带 -M *MPI-version* 选项和 -o 选项）调用 collect 命令，则采用用于定义该进程的 MPI 等级的环境变量的名称中使用的值 *n*。否则，将 *n* 设置为比当前使用的最大整数还要大的值。

如果没有使用 *stem*.*n*.*er* 格式指定名称，而且给定名称已在使用，将显示一条错误消息且实验不会运行。如果名称采用 *stem*.*n*.*er* 格式且提供的名称已在使用，将在比当前使用的最大值 *n* 还要大的值所对应的名称下记录实验。如果名称已更改，则会显示一条警告。

## 使用 **-d** *directory-name* 设置实验目录

将实验置于 *directory-name* 目录中。此选项仅适用于个别实验，而不适用于实验组。如果该目录不存在，则 collect 实用程序会输出一则错误消息并退出。如果使用 -g 选项指定了某个组，则该组文件也将写入 *directory-name* 中。

对于最轻量的数据收集，最好使用 -d 选项指定存放数据的目录，以便将数据记录到本地文件。但是，对于群集上的 MPI 实验，创建者实验必须在相同的路径下可用，以便所有进程将所有数据记录到创建者实验中。

如果将实验写入延迟长的文件系统，问题尤为突出，可能进展非常缓慢，特别是在收集抽样数据时（缺省采用 -S on 选项）。如果必须通过延迟长的连接来进行记录，请禁用抽样数据。

## 使用 **-g** *group-name* 在组中创建实验

使实验成为实验组 *group-name* 的一部分。如果 *group-name* 不以 .erg 结尾，则 collect 实用程序会输出一条错误消息并退出。如果该组存在，则会将实验添加到该组中。如果 *group-name* 不是绝对路径并且使用 -d 指定了一个目录，则实验组将被置于 *directory-name* 目录中，否则，将被置于当前目录中。

## 使用 **-A** *option* 归档实验中的装入对象

控制是否应将目标进程所使用的装入对象归档或复制到记录的实验中。允许的选项值包括：

- on – 将装入对象（目标以及其使用的任何共享对象）复制到实验中。也会复制任何 .anc 文件和 .o 文件，这些文件的 Stabs 或 DWARF 调试信息不在装入对象中。这是缺省值。
- src – 除了像 -A on 中一样复制装入对象，还将可找到的所有源文件和 .anc 文件复制到实验中。
- used[src] – 除了像 -A on 中一样复制装入对象，还将记录的数据中引用的和可找到的所有源文件和 .anc 文件复制到实验中。
- off – 不将装入对象归档复制或归档到实验中。

如果希望将实验复制到另一台计算机，或者从另一台计算机读取实验，请指定 -A on。实验将占用更多磁盘空间，但允许在其他计算机上读取实验。

-A on 不会将任何源文件或对象 (.o) 文件复制到实验中。必须确保可从用于检查实验的计算机访问这些文件。在记录实验后，不应更改或重新生成这些文件。

在收集时归档实验（尤其是包含许多子孙进程的实验）可能代价很高。对于此类实验，较好的策略是使用 -A off 收集数据，运行终止后使用 -A 标志运行 er_archive。

-A 的缺省设置为打开。

### 使用 -o *file* 将命令输出保存到文件

将 collect 本身的所有输出附加到指定的 *file*，但不会重定向以下各项的的输出：产生的目标、dbx（使用 -P 选项调用）或记录计数数据中涉及的进程（使用 -c 参数调用）。如果 *file* 设置为 /dev/null，则禁止 collect 的所有输出（包括任何错误消息）。

## 其他选项

这些 collect 命令选项用于其他目的。

### 使用 -P *process-id* 连接到进程

编写 dbx 脚本以附加到具有给定 *process-id* 的进程，从中收集数据，然后使用该脚本调用 dbx。

可以指定时钟或硬件计数器分析数据，但是不支持跟踪或计数数据。有关更多信息，请参见 collector(1) 手册页。

当连接到进程时，会使用运行 collect -P 的用户的 umask 创建目录，但是会以运行 dbx 所连接进程的用户身份写入实验。如果执行连接的用户是 root 用户，但 umask 不为零，则实验将失败。

### 使用 -C *comment* 将注释添加到实验

将注释置于实验的 notes 文件中。最多可以提供十个 -C 选项。该 notes 文件的内容会置于实验标题的前面。

### 使用 -n 试用命令

不运行目标，但输出在运行目标时要生成的实验的详细信息。此选项是模拟运行选项。

### 使用 -V 显示 collect 版本

输出 collect 命令的当前版本。不再检查任何参数，也不执行进一步的处理。

### 使用 -v 显示详细输出

输出 collect 命令的当前版本和正在运行的实验的详细信息。

## 使用 collect 实用程序从正在运行的进程中收集数据

仅在 Oracle Solaris 平台中，可以将 -P *pid* 选项与 collect 实用程序一起使用来附加到具有指定 PID 的进程并从该进程收集数据。collect 命令的其他选项被转换为 dbx 脚本，系统会调用该脚本来收集数据。只能收集时钟分析数据（-p 选项）和硬件计数器分析数据（-h 选项）。不支持跟踪数据。

如果在使用 -h 选项时未明确指定 -p 选项，则时钟分析功能将处于关闭状态。要同时收集硬件计数器数据和时钟数据，必须同时指定 -h 选项和 -p 选项。

### ▼ 使用 collect 实用程序从正在运行的进程中收集数据

1. 确定程序的进程 ID (process ID, PID)。
   如果从命令行启动了程序并将其置于后台，shell 将在标准输出中输出其 PID。否则，可以通过键入以下内容来确定程序的 PID。

   ```
   % ps -ef | grep program-name
   ```

2. 使用 collect 命令对该进程启用数据收集，并设置任何可选参数。

   ```
   % collect -P pid collect-options
   ```

   “数据收集选项” [54]中对收集器选项进行了说明。有关时钟分析的信息，请参见“使用 -p *option* 选项收集时钟分析数据” [54]。有关硬件时钟分析的信息，请参见“使用 collect -h 收集硬件计数器分析数据” [55]。

# 使用 dbx collector 子命令收集数据

本节介绍如何从 dbx 运行收集器，然后介绍可以与 dbx 中的 collector 命令一起使用的每个子命令。

# ▼ 从 dbx 运行收集器

1. **通过键入以下命令将程序装入 dbx 中。**

   % **dbx** *program*

2. **使用 collector 命令启用数据收集，选择数据类型并设置任何可选参数。**

   (dbx) **collector** *subcommand*

   要查看可用 collector 子命令的列表，请键入：

   (dbx) **help collector**

   必须针对每个子命令都使用一个 collector 命令。

3. **设置任何 dbx 选项并运行该程序。**
   如果提供的子命令不正确，则会输出一条警告消息并忽略该子命令。下节提供了 collector 子命令的完整列表。

## 数据收集子命令

以下子命令可与 dbx 中的 collector 命令一起使用，对收集器收集的数据的类型进行控制。如果实验处于活动状态，则这些子命令将被忽略并显示一条警告。

### **profile** *option*

控制对时钟分析数据的收集。*option* 的允许值包括：

- on – 启用缺省分析间隔为 10 毫秒的时钟分析。
- off – 禁用时钟分析。
- timer *interval* – 设置分析间隔。*interval* 的允许值包括：
  - on – 使用大约为 10 毫秒的缺省分析间隔。
  - lo[w] – 使用大约为 100 毫秒的低精度分析间隔。
  - hi[gh] – 使用大约为 1 毫秒的高精度分析间隔。有关启用高精度分析的信息，请参见“时钟分析的限制” [47]。
  - *value* – 将分析间隔设置为 *value*。*value* 的缺省单位为毫秒。可以将 *value* 指定为整数或浮点数。可以选择在数值后加后缀 m 来选择毫秒单位或者加 u 来选择微秒单位。该值应当是时钟精度的倍数。如果该值大于时钟精度但不是其倍数，则向下舍入。如果该值小于时钟精度，则将其设置为时钟精度。在以上两种情况下均输出警告消息。

    缺省设置大约为 10 毫秒。

缺省情况下收集器收集时钟分析数据，除非使用 hwprofile 子命令打开对硬件计数器分析数据的收集。

### **hwprofile** *option*

控制对硬件计数器分析数据的收集。如果您尝试在不支持硬件计数器分析的系统中启用它，则 dbx 会返回一条警告消息，而且该命令将被忽略。*option* 的允许值包括：

- on – 打开硬件计数器分析。缺省操作是收集具有正常溢出值的 cycles 计数器的数据。
- off – 关闭硬件计数器分析。
- list – 返回可用计数器的列表。有关对该列表的描述，请参见“硬件计数器列表” [24]。如果系统不支持硬件计数器分析，则 dbx 会返回一条警告消息。
- counter *counter-definition*... [, *counter-definition* ] – 计数器定义采用以下格式。

  [+]*counter-name*[~ *attribute_1*=*value_1*]...[~*attribute_n* =*value_n*][/ *register-number*][,*interval* ]

  选择硬件计数器 *name*，并将其溢出值设置为 *interval*；也可以选择其他硬件计数器名称并将其溢出值设置为指定的间隔。溢出值可以为下列值之一：

  - on 或空字符串 – 缺省溢出值，可以通过键入不带其他参数的 collect -h 来确定。
  - hi[gh] – 所选计数器的高精度值，大约比缺省溢出值短十倍。之所以还支持缩写 h，是为了与以前的软件发行版兼容。
  - lo[w] – 所选计数器的低精度值，大约比缺省溢出值长十倍。
  - *interval* – 特定的溢出值，必须是正整数，可以采用十进制格式，也可以采用十六进制格式。

    如果指定了多个计数器，则各计数器必须使用不同的寄存器。否则，将输出一条警告消息，而且该命令将被忽略。

    如果硬件计数器对其计数的事件与内存访问有关，则可以在计数器名称前放置 + 符号，以便对导致计数器溢出的指令的真实 PC 进行搜索。如果搜索成功，则 PC 和所引用的有效地址将被存储在事件数据包中。

    缺省情况下，收集器不收集硬件计数器分析数据。如果硬件计数器分析处于启用状态，而且尚未发出 profile 命令，则时钟分析将处于关闭状态。

    另请参见“硬件计数器分析的限制” [48]。

### **synctrace** *option*

控制对同步等待跟踪数据的收集。*option* 的允许值包括：

- on – 启用具有缺省阈值的同步等待跟踪。

- `off` – 禁用同步等待跟踪。
- `threshold` *value* – 为最小同步延迟设置阈值。*value* 的允许值包括：
  - `all` – 使用零阈值。该选项强制记录所有同步事件。
  - `calibrate` – 在运行时通过校准来设置阈值。（与 `on` 等效。）
  - `off` – 关闭同步等待跟踪。
  - `on` – 使用缺省阈值，这将在运行时通过校准来设置阈值。（与 `calibrate` 等效。）
  - *number* – 将阈值设置为 *number*，该值以正整数形式提供，单位为微妙。如果 `value` 为 0，则跟踪所有事件。

    缺省情况下，收集器不收集同步等待跟踪数据。

### **heaptrace** *option*

控制对堆跟踪数据的收集。*option* 的允许值包括：

- `on` – 启用堆跟踪。
- `off` – 禁用堆跟踪。

缺省情况下，收集器不收集堆跟踪数据。

### **tha** *option*

为线程分析器收集数据争用检测或死锁检测数据。允许的值包括：

- `off` – 关闭线程分析器数据收集。
- `all` – 收集所有线程分析器数据。
- `race` – 收集数据争用检测数据。
- `deadlock` – 收集死锁和潜在死锁数据。

有关线程分析器的更多信息，请参见《Oracle Solaris Studio 12.4：线程分析器用户指南》和 `tha`.1 手册页。

### **sample** *option*

控制抽样模式。*option* 的允许值包括：

- `periodic` – 启用定期抽样。
- `manual` – 禁用定期抽样。手动抽样仍保持启用状态。
- `period` *value* – 将抽样间隔设置为 *value*，以秒为单位。

缺省情况下，启用抽样间隔 *value* 为 1 秒的定期抽样。

### **dbxsample** { **on** | **off** }

控制当 dbx 停止目标进程时对抽样的记录。关键字的含义如下所示：

- on – 在 dbx 每次停止目标进程时记录抽样。
- off – 在 dbx 停止目标进程时不记录抽样。

缺省情况下，在 dbx 停止目标进程时记录抽样。

## 实验控制子命令

以下子命令可与 dbx 中的 collector 命令一起使用，对收集器收集的实验数据进行控制。如果实验处于活动状态，将忽略这些子命令并显示一条警告。

### **disable** 子命令

禁用数据收集功能。如果进程正在运行而且正在收集数据，该子命令将终止实验并禁用数据收集功能。如果进程正在运行而且数据收集功能处于禁用状态，则该子命令将被忽略并显示一条警告。如果没有任何进程在运行，则该子命令将针对后续运行禁用数据收集功能。

### **enable** 子命令

启用数据收集功能。如果进程正在运行，但数据收集功能处于禁用状态，则该子命令将启用数据收集功能并启动新的实验。如果进程正在运行，而且数据收集功能处于启用状态，则该子命令将被忽略并显示一条警告。如果没有任何进程在运行，则该子命令将针对后续运行启用数据收集功能。

您可以在任何进程执行期间根据需要启用和禁用数据收集功能任意次数。每次启用数据收集功能时，都会创建一个新实验。

### **pause** 子命令

暂停数据收集，但使实验保持打开状态。收集器暂停时不记录抽样点。在暂停之前会生成一个抽样，在恢复之后会立即生成另一个抽样。如果已暂停数据收集功能，则该子命令将被忽略。

### **resume** 子命令

在发出 pause 之后恢复数据收集。如果正在收集数据，则该子命令将被忽略。

### **sample record** *name* 子命令

记录具有标签 *name* 的抽样包。该标签显示在性能分析器的 "Selection Details"（选择详细信息）窗口中。

## 输出子命令

以下子命令可与 dbx 中的 collector 命令一起使用，对实验的存储选项进行定义。如果实验处于活动状态，将忽略这些子命令并显示一条警告。

### **archive** *mode* 子命令

设置用于归档实验的模式。*mode* 的允许值包括：

- on – 对装入对象进行正常归档
- off – 不对装入对象进行归档
- copy – 除正常归档之外，还将装入对象复制到实验中

如果打算将实验移到另一台计算机上，或者从另一台计算机上读取实验，则应当启用对装入对象的复制。如果实验处于活动状态，则该命令将被忽略并显示一条警告。该命令不会将源文件或对象文件复制到实验中。

### **limit** *value* 子命令

将所记录的分析数据量限制在 *value* 兆字节。该限制适用于时钟分析数据量、硬件计数器分析数据量和同步等待跟踪数据量之和，但不适用于抽样点。该限制只是近似值，可以被超出。

当达到该限制时，不再记录分析数据，但实验会一直保持打开状态，而且会继续记录抽样点。

缺省情况下，记录的数据量是不受限制的。

### `store` *option* 子命令

控制实验的存储位置。如果实验处于活动状态，则此命令将被忽略并显示一条警告。*option* 的允许值包括：

- `directory` *directory-name* – 设置用来存储实验和实验组的目录。如果该目录不存在，则该子命令将被忽略并显示一条警告。
- `experiment` *experiment-name* – 设置实验的名称。如果实验名称不以 `.er` 结尾，则该子命令将被忽略并显示一条警告。有关实验名称以及收集器如何对其进行处理的更多信息，请参见“数据的存储位置” [50]。
- `group` *group-name* – 设置实验组的名称。如果实验组名称不以 `.erg` 结尾，则该子命令将被忽略并显示一条警告。如果该组已经存在，则将实验添加到该组中。如果目录名是用 `store directory` 子命令设置的，而且组名不是绝对路径，则组名以目录名为前缀。

## 信息子命令

以下子命令可与 `dbx` 中的 `collector` 命令一起使用，从而获取关于收集器设置和实验状态的报告。

### `show` 子命令

显示每个收集器控件的当前设置。

### `status` 子命令

报告任何打开的实验的状态。

# 在 Oracle Solaris 平台上使用 `dbx` 从正在运行的进程中收集数据

在 Oracle Solaris 平台上，使用收集器可以从正在运行的进程中收集数据。如果进程已在 `dbx` 的控制下，则可以暂停该进程并使用前几节中所描述的方法来启用数据收集。Linux 平台不支持在正在运行的进程上启动数据收集。

如果进程不受 `dbx` 的控制，则可以使用 `collect –P` *pid* 命令从正在运行的进程中收集数据，如“使用 `collect` 实用程序从正在运行的进程中收集数据” [70]中所述。您还可以向进程附加 `dbx`，收集性能数据，然后从该进程分离并使进程继续运行。如果要为选定的子孙进程收集性能数据，则必须将 `dbx` 附加到每个进程。

## ▼ 从不受 dbx 控制的正在运行的进程中收集数据

1. 确定程序的进程 ID (process ID, PID)。
   如果从命令行启动了程序并将其置于后台，shell 将在标准输出中输出其 PID。否则，可以通过键入以下内容来确定程序的 PID：

   ```
   % ps -ef | grep program-name
   ```

2. 附加到该进程。
   从 dbx 中，键入以下命令：

   ```
   (dbx) attach program-name pid
   ```

   如果 dbx 尚未运行，请键入以下命令：

   ```
   % dbx program-name pid
   ```

   附加到正在运行的进程会使该进程暂停。
   有关附加到进程的更多信息，请参见手册《Oracle Solaris Studio 12.4：使用 dbx 调试程序》。

3. 启动数据收集功能。
   在 dbx 中，使用 `collector` 命令来设置数据收集参数，使用 `cont` 命令来恢复执行进程。

4. 从进程中分离。
   在完成对数据的收集之后，暂停该程序并从 dbx 中分离该进程。
   从 dbx 中，键入以下命令：

   ```
   (dbx) detach
   ```

# 从正在运行的程序中收集跟踪数据

如果要收集任何种类的跟踪数据，则必须在运行程序之前预装入收集器库 `libcollector.so`。要收集堆跟踪数据或同步等待跟踪数据，还必须分别预装入 `er_heap.so` 和 `er_sync.so`。这些库提供了能使数据收集发生的实际函数的包装。此外，收集器还将包装函数添加到其他系统库调用中，以保证性能数据的完整性。如果未预装入库，则不能插入这些包装函数。有关收集器如何插入系统库函数的更多信息，请参见“使用系统库” [41]。

要预装入 `libcollector.so`，必须使用环境变量同时设置库的名称和库的路径，如下表中所示。使用环境变量 `LD_PRELOAD` 设置库的名称。使用环境变量 `LD_LIBRARY_PATH`、`LD_LIBRARY_PATH_32` 或 `LD_LIBRARY_PATH_64` 设置库的路径。如果未定义 `_32` 和 `_64` 变量，则使用 `LD_LIBRARY_PATH`。如果已经定义了这些环境变量，则向其中添加新值。

表 3-2 用来预装入 libcollector.so、er_sync.so 和 er_heap.so 的环境变量设置

| 环境变量 | 值 |
| --- | --- |
| LD_PRELOAD | libcollector.so |
| LD_PRELOAD | er_heap.so |
| LD_PRELOAD | er_sync.so |
| LD_LIBRARY_PATH | /opt/solarisstudio12.4/lib/analyzer/runtime |
| LD_LIBRARY_PATH_32 | /opt/solarisstudio12.4/lib/analyzer/runtime |
| LD_LIBRARY_PATH_64 | /opt/solarisstudio12.4/lib/analyzer/v9/runtime |
| LD_LIBRARY_PATH_64 | /opt/solarisstudio12.4/lib/analyzer/amd64/runtime |

如果 Oracle Solaris Studio 软件未安装在 /opt/solarisstudio12.4 中，请向系统管理员咨询正确的路径。可以在 LD_PRELOAD 中设置全路径，但是，当使用 SPARC® V9 64 位体系结构时，这样做会使问题复杂化。

注 - 运行后删除 LD_PRELOAD 和 LD_LIBRARY_PATH 设置，以便它们对于从同一个 shell 启动的其他程序无效。

# 从脚本收集数据

您可以指定一个脚本作为 collect 命令的目标。当目标是脚本时，在缺省情况下，collect 将收集为执行此脚本而启动的程序的数据，以及所有子孙进程的数据。

要仅针对特定进程收集数据，请使用 -F 选项指定要跟踪的可执行文件的名称。

例如，要分析脚本 start.sh，但是首先从可执行文件 myprogram 收集数据，需要使用以下命令。

```
$ collect -F =myprogram start.sh
```

针对为执行 start.sh 脚本而启动的创建者进程和从脚本产生的所有 myprogram 进程收集数据，但不为其他进程收集数据。

# 将 collect 和 ppgsz 一起使用

通过在 ppgsz 命令上运行 collect 并指定 -F on 或 -F all 标志，可以将 collect 与 ppgsz(1) 一起使用。创建者实验位于 ppgsz 可执行文件上，我们不需要关注它。如果在您的路径中找到 32 位版本的 ppgsz，而实验在支持 64 位进程的系统上运行，则将执行 (exec) 它的 64 位版本，创建 _x1.er。该可执行文件将进行派生，创建 _x1_f1.er。

子进程尝试针对路径上的第一个目录中指定的目标执行 exec，然后针对第二个目录中的目标执行 exec，依此类推，直到其中一个 exec 尝试成功。例如，如果第三个尝试成功，则前两个子孙实验分别命名为 _x1_f1_x1.er 和 _x1_f1_x2.er，并且这两个实验都完全为空。目标上的实验是来自成功的 exec 的某个实验，在本例中为第三个实验，命名为 _x1_f1_x3.er 并存储在创建者实验之下。通过针对 test.1.er/_x1_f1_x3.er 调用分析器或 er_print 实用程序，可以直接处理该实验。

如果 64 位 ppgsz 是初始进程，或者如果在 32 位内核上调用 32 位 ppgsz，那么，针对实际目标执行 exec 操作的派生子进程的数据位于 _f1.er 中，而实际目标的实验位于 _f1_x3.er 中，前提是采用与上例相同的路径属性。

# 从 MPI 程序收集数据

收集器可以从使用消息传递接口 (Message Passing Interface, MPI) 的多进程程序收集性能数据。

收集器支持 Oracle Message Passing Toolkit 8（以前称为 Sun HPC ClusterTools 8）及其更新。收集器可以识别其他版本的 MPI；当您运行不带其他参数的 collect -h 时，将显示有效的 MPI 版本列表。

在 http://www.oracle.com/us/products/tools/message-passing-toolkit-070499.html 上可以找到 Oracle Message Passing Toolkit MPI 软件，它用于安装 Oracle Solaris 10 和 Linux 系统。

Oracle Message Passing Toolkit 已集成在 Oracle Solaris 11 发行版中。如果系统中安装了此工具包，您可以在 /usr/openmpi 中找到它。如果您的 Oracle Solaris 11 系统中尚未安装此工具包，当您为系统配置了软件包系统信息库时，可以使用命令 pkg search openmpi 搜索该工具包。有关在 Oracle Solaris 11 中安装软件的更多信息，请参见 Oracle Solaris 11 文档库中的《*Adding and Updating Oracle Solaris 11 Software Packages*》手册。

有关 MPI 和 MPI 标准的信息，请访问 MPI Web 站点 http://www.mcs.anl.gov/mpi/。有关 Open MPI 的更多信息，请访问 Web 站点 http://www.open-mpi.org/。

要从 MPI 作业收集数据，必须使用 collect 命令；dbx collector 子命令不能用于启动 MPI 数据收集。“对 MPI 运行 collect 命令” [79]中提供了详细信息。

## 对 MPI 运行 collect 命令

collect 命令可用于跟踪和分析 MPI 应用程序。

要收集数据，请使用以下语法：

**collect [***collect-arguments***] mpirun [***mpirun-arguments***] -- *program-name* [***program-arguments***]**

例如，以下命令对 16 个 MPI 进程的每个进程运行 MPI 跟踪和分析，在单个 MPI 实验中存储数据：

```
collect -M OMPT mpirun -np 16 -- a.out 3 5
```

`-M OMPT` 选项指明将进行 MPI 分析，Oracle Message Passing Toolkit 为 MPI 版本。

初始收集进程重设 `mpirun` 命令的格式来指定对各个 MPI 进程运行 `collect` 命令（带有适当的参数）。

紧接在 *program_name* 前面的 `–`参数是 MPI 分析所必需的。如果不包括 `--` 参数，`collect` 命令将显示一条错误消息且不会收集实验。

---

注 - 不再支持使用 `mpirun` 命令在 MPI 进程上产生显式 `collect` 命令的技术来收集 MPI 跟踪数据。您仍可以使用这种技术来收集其他类型的数据。

---

# 存储 MPI 实验

由于多进程环境比较复杂，因此从 MPI 程序收集性能数据时应当注意一些有关存储 MPI 实验的问题。这些问题涉及到数据收集和存储的效率以及实验的命名。有关命名实验（包括 MPI 实验）的信息，请参见“数据的存储位置” [50]。

用来收集性能数据的每个 MPI 进程都创建其自己的子实验。当 MPI 进程创建实验时，将锁定实验目录；所有其他 MPI 进程必须一直等待，直到解除锁定，这些进程才可以使用该目录。将试验存储在所有 MPI 进程可以访问的文件系统上。

如果未指定实验名称，则使用缺省的实验名称。在实验中，收集器将为每个 MPI 等级创建一个子实验。收集器使用 MPI 等级以格式 `M_r`*m*`.er` 构造子实验名称，其中 *m* 是 MPI 等级。

如果打算在实验完成后将其移动到其他位置，则在 `collect` 命令中指定 `-A copy` 选项。要复制或移动实验，请勿使用 UNIX `cp` 或 `mv` 命令。而应使用 `er_cp` 或 `er_mv` 命令，如第 8 章 操作实验中所述。

MPI 跟踪在每个节点的 `/tmp/a.*.z` 中创建临时文件。`MPI_finalize`（） 函数调用期间将删除这些文件。确保文件系统有足够的空间用于实验。对长时间运行的 MPI 应用程序收集数据之前，请执行短期试运行以确认文件大小。有关如何估计所需空间的信息，另请参见“估计存储要求” [52]。

MPI 分析以开源 VampirTrace 5.5.3 发行版为基础。它可识别多个受支持的 VampirTrace 环境变量以及新的环境变量 `VT_STACKS`，该环境变量控制调用堆栈是否记录在数据中。有关这些变量的含义的详细信息，请参见 VampirTrace 5.5.3 文档。

环境变量 VT_BUFFER_SIZE 的缺省值将 MPI API 跟踪收集器的内部缓冲区限制为 64 MB。某一 MPI 进程达到限制的大小时，缓冲区将刷新到磁盘，前提是尚未达到 VT_MAX_FLUSHES 限制。缺省情况下，VT_MAX_FLUSHES 被设置为 0。该设置将导致 MPI API 跟踪收集器在缓冲区已满时刷新磁盘的缓冲区。如果将 VT_MAX_FLUSHES 设置为正数，就是对允许刷新的次数进行了限制。如果缓冲区已满且无法刷新，不再将该进程的事件写入跟踪文件。结果是试验可能不完整，在某些情况下，实验可能不可读。

要更改缓冲区大小，请使用环境变量 VT_BUFFER_SIZE。该变量的最佳值取决于要跟踪的应用程序。设置较小的值将增加应用程序可以使用的内存，但是将触发 MPI API 跟踪收集器频繁进行缓冲区刷新。这些缓冲区刷新可能会显著改变应用程序的行为。另一方面，设置较大的值（如 2 GB）可以使 MPI API 跟踪收集器刷新缓冲区的次数降至最低，但是将减少应用程序可以使用的内存。如果没有足够的内存可用来容纳缓冲区和应用程序数据，应用程序的某些部分可能会交换至磁盘，从而还会导致应用程序的行为发生显著改变。

另一个重要的变量是 VT_VERBOSE，该变量启用各种错误和状态消息显示。如果出现问题，请将此变量设置为 2 或更大的值。

通常，当 mpirun 目标存在时，MPI 跟踪输出数据是后处理的；将处理的数据文件写入实验，而将关于后处理时间的信息写入实验标题。如果利用 -m off 显式禁用 MPI 跟踪，就不执行 MPI 后处理。后处理失败时，报告错误，且任何 MPI 标签或 MPI 跟踪度量都不可用。

即使 mpirun 目标没有实际调用 MPI，仍然会记录实验，但不会生成任何 MPI 跟踪数据。该实验报告 MPI 后处理错误，任何 MPI 标签或 MPI 跟踪度量都将不可用。

如果将环境变量 VT_UNIFY 设置为 0，collect 不会运行后处理例程。首次在实验上调用 er_print 或 analyzer 时，将运行这些后处理例程。

---

注 - 除非您能够访问源文件或者拥有具备相同时间戳的副本，否则在计算机或节点间复制或移动实验时不能在带注释的反汇编代码中查看带注释的源代码或源代码行。可以在当前目录中放置指向原始源文件的符号链接，以便查看带注释的源代码。也可以使用 "Settings"（设置）对话框中的设置。使用 "Search Path"（搜索路径）标签（请参见“搜索路径设置” [118]）管理要用来搜索源文件的目录列表。使用 "Pathmaps"（路径映射）标签（请参见“路径映射设置” [119]）将文件路径的前面部分从一个位置映射到另一个位置。

---

♦ ♦ ♦ 第 4 章

# 性能分析器工具

本章介绍性能分析器图形化数据分析工具，其中包含以下主题：



## 关于性能分析器

性能分析器是一款用于分析收集器所收集性能数据的图形化数据分析工具。从性能分析器分析应用程序、使用 `collect` 命令或在 `dbx` 中使用 `collector` 命令时，收集器将启动。收集器在进程执行过程中收集性能信息以创建实验，如第 3 章 收集性能数据中所述。

性能分析器读入这些实验，分析数据，然后以表格和图形显示这些数据。

---

注 - 可以从 Oracle Solaris Studio 12.4 样例应用程序页面的样例应用程序 zip 文件中下载性能分析器的演示代码，网址为 http://www.oracle.com/technetwork/server-storage/solarisstudio/downloads/solaris-studio-12-4-samples-2333090.html。

接受许可并下载后，可以将 zip 文件提取到所选择的目录中。样例应用程序位于 `SolarisStudioSampleApplications` 目录的 `PerformanceAnalyzer` 子目录中。有关如何在性能分析器中使用样例代码的信息，请参见《Oracle Solaris Studio 12.4：性能分析器教程》。

---

# 启动性能分析器

要启动性能分析器，请在终端窗口中键入以下命令：

```
% analyzer [control-options] [experiment | experiment-list]
```

可以指定实验名称或列表。*experiment-list* 命令参数是实验名称、实验组名称或两者的空格分隔列表。如果未提供实验列表，性能分析器将启动并打开 "Welcome"（欢迎）页。

您可以在命令行中指定多个实验或实验组。如果指定的实验中包含子孙实验，将会自动装入所有子孙实验并聚集数据。要装入个别子孙实验，必须显式指定每个实验或者创建实验组。也可以在 `.er.rc` 文件中放入 `en_desc` 指令（请参见“en_desc { on | off | =*regexp*}” [152]）。

读取包含子孙的实验时，性能分析器和 `er_print` 将忽略包含少量性能数据或不包含性能数据的任何子实验。

要创建实验组，可以在 `collect` 实用程序中使用 `-g` 参数。要手动创建实验组，请创建首行为以下内容的纯文本文件：

```
#analyzer experiment group
```

然后将实验名称添加到随后的行中。文件的扩展名必须为 `erg`。

性能分析器显示多个实验时，缺省情况下聚集所有实验的数据。组合和查看数据时，就好像数据来自一个实验一样。但是，如果指定 `-c` 选项，还可以选择比较实验而不聚集数据。请参见“比较实验” [111]。

可以在 "Open Experiment"（打开实验）对话框中单击实验或实验组的名称来预览它们。

您还可以按以下所示通过命令行启动性能分析器来记录实验：

```
% analyzer [Java-options] [control-options] target [target-arguments]
```

性能分析器启动后显示 "Profile Application"（分析应用程序）窗口，其中显示指定的目标及其参数以及用于分析应用程序和收集实验的设置。有关详细信息，请参见“从性能分析器分析应用程序” [110]。

您也可以打开“实时”实验，也就是仍在进行收集的实验。在打开实时实验时，您将只看到在实验打开时已经收集到的数据。在新数据传入时，实验并不自动更新。要进行更新，可以重新打开实验。

## `analyzer` 命令选项

这些 `analyzer` 命令选项控制性能分析器的行为，分为以下几组：

- 实验选项
- Java 选项
- 控制选项
- 信息选项

## 实验选项

这些选项指定如何处理在命令行上指定的实验。

### **-c** *base-group compare-group*

启动性能分析器并比较指定的实验。

*base-group* 要么是单个实验，要么是指定多个实验的 *groupname*`.erg` 文件。*compare-group* 是要与基本组比较的一个或多个实验。

要指定比较组中的多个实验，请使用空格来分隔实验名称。也可以指定 *groupname*`.erg` 文件，该文件指定比较组中的多个实验。

**例 4-1** 在比较模式下打开实验的样例命令

打开实验 `test.1.er`，将其与 `test.4.er` 进行比较：

```
% analyzer -c test.1.er test.4.er
```

打开实验组 `demotest.erg`，将其与 `test.4.er` 进行比较：

```
% analyzer -c demotest.erg test.4.er
```

## **analyzer** 命令的 Java 选项

这些选项为运行性能分析器的 JVM 指定设置。

### **-j | --jdkhome** *jvm-path*

指定运行性能分析器的 Java 软件的路径。如果未指定 `-j` 选项，则首先采用缺省路径，方法是在环境变量中检查 JVM 的路径（先检查 `JDK_HOME`，再检查 `JAVA_PATH`）。如果这两个环境变量均未设置，将使用 PATH 上找到的 JVM。使用 `-j` 选项可覆盖所有缺省路径。请参见

**-J***jvm-option*

指定 JVM 选项。可以指定多个选项。例如：

- 要运行 64 位性能分析器，请键入：

  ```
  analyzer -J-d64
  ```
- 要运行最大 JVM 内存为 2 GB 的性能分析器，请键入：

  ```
  analyzer -J-Xmx2G
  ```
- 要运行最大 JVM 内存为 8 GB 的 64 位性能分析器，请键入：

  ```
  analyzer -J-d64 -J-Xmx8G
  ```

## analyzer 命令的控制选项

这些 analyzer 命令选项控制存储设置的用户目录的位置，设置用户界面的字体大小，并在启动性能分析器之前显示版本和运行时信息。

**-f | --fontsize** *size*

指定要在性能分析器用户界面中使用的字体大小。

要启动性能分析器并以 14 磅字体显示菜单，请键入以下命令：

```
analyzer -f 14
```

**-v | --verbose**

显示版本信息和 Java 运行时参数，然后启动性能分析器。

## analyzer 命令的信息选项

这些选项将有关 analyzer 的信息输出至标准输出。以下每个选项都是独立选项；它们不能与其他 analyzer 选项、目标或 experiment-list 参数结合使用。

**-V | --version**

仅显示版本信息，不启动性能分析器。

**-? | -h | --help**

输出用法信息并退出。

# 性能分析器用户界面

性能分析器窗口包含菜单栏、工具栏和导航面板。可使用其中的每个工具与性能分析器以及数据进行交互。

## 菜单栏

菜单栏包含以下命令菜单。

- "File"（文件）提供用于分析应用程序、打开实验以及比较或聚集多个实验的数据的多个选项。其他选项的功能包括将数据视图的内容导出为多个不同格式，连接到远程主机，在性能分析器中使用此处的文件和应用程序。
- "Views"（视图）用于选择性能分析器在导航栏中显示的数据视图。还可以单击导航栏中的 "Views +"（视图 +）按钮或 "More Views"（更多视图）添加更多数据视图。
- "Tools"（工具）提供用于过滤数据、在数据视图中隐藏库函数的选项以及其他设置。
- "Help"（帮助）提供性能分析器的集成文档链接，包括新功能、快速参考、键盘快捷键以及故障排除信息的相关链接。也可以按键盘上的 F1 键，显示有关性能分析器中当前视图的信息。

## 工具栏

工具栏提供了可用作命令快捷方式的按钮，一个查看模式选择器（用于更改 Java 和 OpenMP 实验数据的显示方式），以及一个查找工具（用于在数据视图中查找文本）。

图 4-1　性能分析器的工具栏

## 导航面板

使用左侧的垂直导航面板，可选择性能分析器中称为数据视图 或一般视图 的各个页面。大多数视图都从某个角度显示分析的应用程序的性能度量。这些视图是相关的，因此在一个视图中进行的选择和应用的过滤器也会应用于其他视图。

导航面板中的某些按钮可打开页面（如 "Welcome"（欢迎）页），便于使用工具和查找信息；"Overview"（概述）页则提供有关打开的实验中数据的信息。

通过单击导航面板顶部的 + 按钮或导航面板底部的 "More Views"（更多视图）按钮，可在导航面板中添加更多视图。也可以使用 "Views"（视图）菜单选择要添加的其他视图。

在大多数数据视图中，单击项目可在 "Selection Details"（选择详细信息）窗口中查看有关这些项目的更详细信息，该窗口在下一节“"Selection Details"（选择详细信息）窗口” [88]中进行了说明。

数据视图在“性能分析器视图” [88]中进行了说明。

## "Selection Details"（选择详细信息）窗口

右侧的 "Selection Details"（选择详细信息）窗口以值和百分比的形式显示为所选项目记录的所有度量，以及有关所选函数或装入对象的信息。只要在任何数据视图中选择了新项目，"Selection Details"（选择详细信息）窗口就会更新。

在 "Timeline"（时间线）视图的抽样栏中选中某个抽样时，"Selection Details"（选择详细信息）窗口将显示抽样编号、抽样的开始时间和结束时间、微状态以及在每个微状态中所花费的时间和彩色编码。

在 "Timeline"（时间线）视图的数据栏中选择某个事件时，"Selection Details"（选择详细信息）窗口将显示事件详细信息和调用堆栈。

## "Called-By/Calls"（调用方/调用）面板

某些数据视图底部提供了 "Called-By/Calls"（调用方/调用）面板，可用于浏览调用路径。选择视图中的一个函数，然后使用 "Called-by/Calls"（调用方/调用）面板导航到调用它的函数或它执行的函数调用。当单击 "Called-by/Calls"（调用方/调用）中的某个函数时，会在数据视图中选择该函数。

# 性能分析器视图

以下因素决定了在打开实验时，是否在导航栏中显示数据视图：

- 实验中数据的类型决定了显示什么数据视图。例如，如果实验包含 OpenMP 数据，则将自动打开 OpenMP 的视图来显示数据。
- 启动性能分析器时读取的配置文件指定要显示的缺省数据视图。

可以使用 "Views"（视图）菜单或按钮打开 "Settings"（设置）对话框（请参见“视图设置” [114]），以选择要在当前性能分析器会话中显示的视图。

大多数视图包含上下文菜单，在右键单击视图中的某个项目时将打开上下文菜单。您可以使用上下文菜单来添加过滤器，或者执行与该数据视图相关的其他活动。在一个视图中应用过滤器时，将在可以过滤的所有视图中过滤数据。

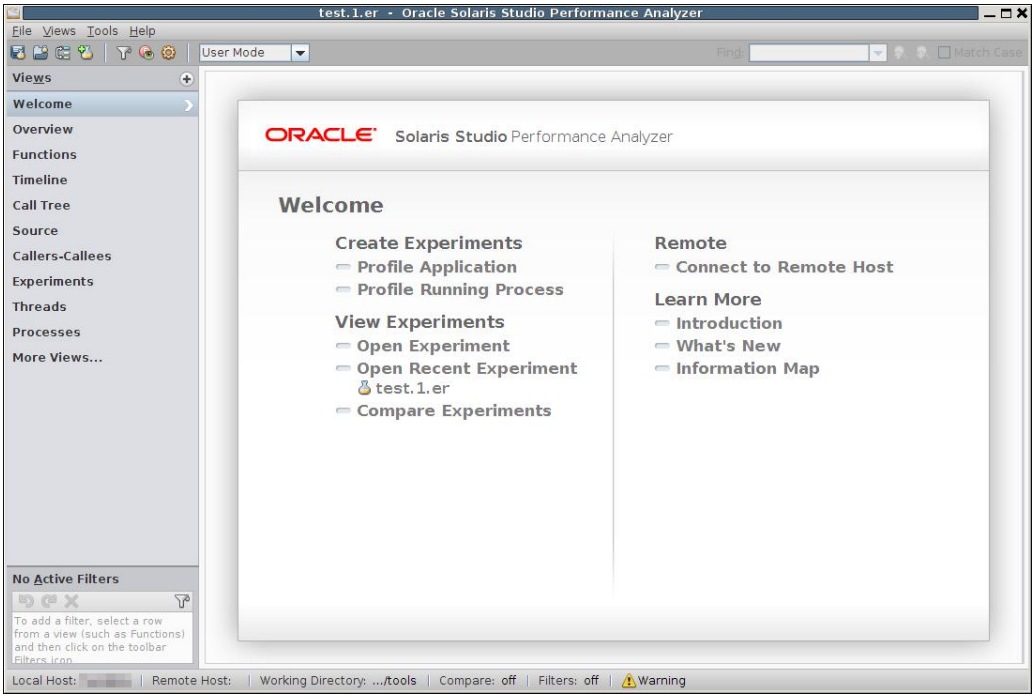
导航栏显示以下项：

# "Welcome"（欢迎）页

如果启动性能分析器时未在命令行上指定要打开的实验，显示的第一个页面是 "Welcome"（欢迎）页。可以通过 "Welcome"（欢迎）页轻松地开始应用程序的分析、查看最近的实验、比较实验以及查看文档。打开实验后，在性能分析器会话期间仍可以随时从导航面板选择 "Welcome"（欢迎）页。

下图显示 "Welcome"（欢迎）页。

图 4-2 性能分析器 "Welcome"（欢迎）页

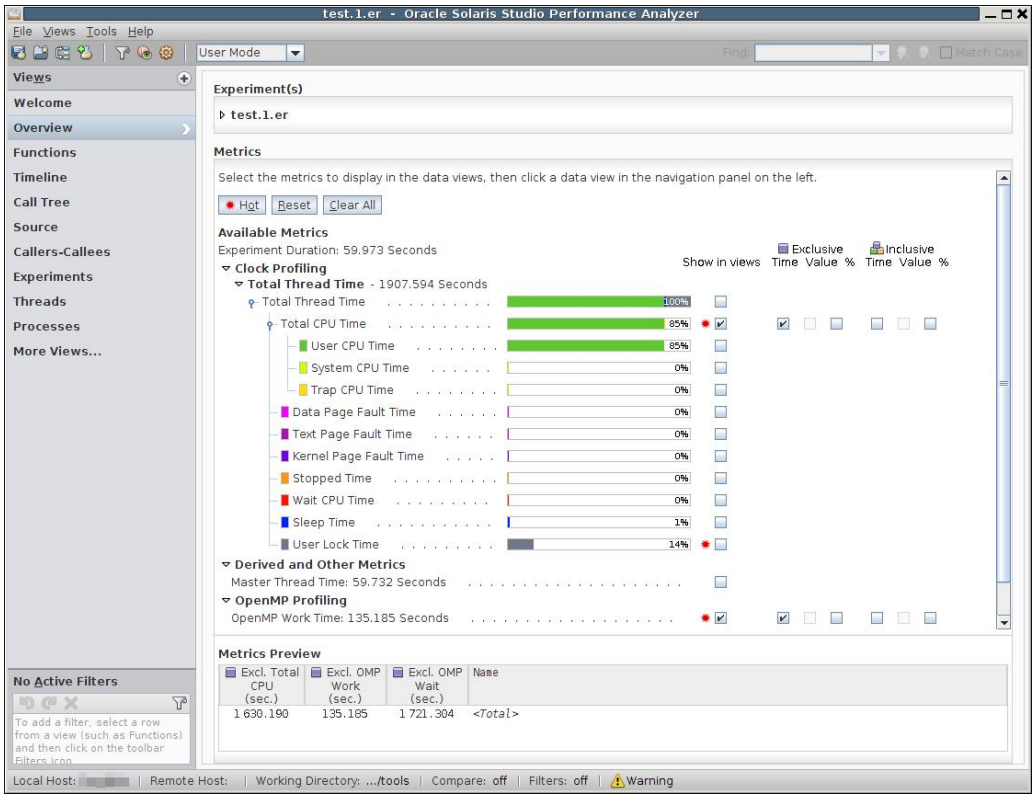


# "Overview"（概述）屏幕

打开实验时，性能分析器将显示 "Overview"（概述），其中显示已装入实验的性能度量，您可以快速了解度量值高的位置。可以使用 "Overview"（概述）选择要在其他视图中了解的度量。查看 "Overview"（概述）时按 F1 可显示有关 "Overview"（概述）的详细信息。

下图为显示 "Overview"（概述）的性能分析器窗口。

图 4-3 性能分析器的 "Overview"（概述）屏幕




选择度量时，窗口底部的 "Metrics Preview"（度量预览）面板将显示度量在数据视图中的形态。

# "Functions"（函数）视图

"Functions"（函数）视图显示目标程序的函数及其度量的列表，这些度量是根据在实验中收集的数据得出的。度量要么是独占度量，要么是非独占度量。独占度量表示仅由函数本身使用。非独占度量表示可由函数及其所调用的所有函数使用。有关度量的更多详细信息，请参见“函数级度量：独占、非独占和归属” [32]。

`collect`(1) 手册页和帮助以及第 2 章 性能数据中提供了收集的各种数据类型的可用度量列表。

时间度量显示为秒，精度可达到毫秒。百分比精确到 0.01%。如果度量值恰好为 0，则时间和百分比显示为 "0"。如果值并非恰好为 0，但是小于精确度，则其值显示为 "0.000"，百分比显示为 "0.00"。由于进行了舍入，百分比总和可能不会恰好是 100%。计数度量显示为整数计数。

初始显示的度量基于收集的数据。如果收集的数据类型多于一种，则显示每种类型的缺省度量。可以选择要在 "Overview"（概述）页面中显示的度量。有关更多信息，请参见“"Overview"（概述）屏幕” [91]。

要搜索函数，请使用工具栏中的查找工具。

要查看函数的源代码，可双击函数以打开 "Source"（源）视图，其中显示源代码中代表该函数的行。

要选择单个函数，请单击该函数，在右侧的 "Selection Details"（选择详细信息）窗口中将显示有关它的更多信息。

要选择在视图中连续显示的多个函数，请选择组中的第一个函数，然后按住 Shift 键并单击组中的最后一个函数。

要选择视图中不连续显示的多个函数，请选择组中的第一个函数，然后通过按住 Ctrl 键并单击每个函数来选择其他函数。

您也可以在 "Function"（函数）视图中右键单击打开上下文菜单，然后选择所选函数的预定义过滤器。有关过滤的详细信息，请参见性能分析器帮助。

# "Timeline"（时间线）视图

"Timeline"（时间线）视图以时间函数形式显示所记录的事件和抽样点的图表。数据显示在水平栏中。缺省情况下，对于每个实验，顶部都有一栏用于表示 CPU 利用率抽样，而且还有一组分析数据栏用于表示每个线程。针对每个线程显示的数据由分析应用程序时收集的数据确定。

您可能会看到以下数据栏：

CPU Utilization Samples（CPU 利用率抽样）

实验包含抽样数据时，显示的顶部栏为 CPU 利用率抽样。抽样点中的数据表示该点和前面点之间所用的 CPU 时间。抽样数据包括微观状态信息，在 Oracle Solaris 系统上可以获得该信息。

Oracle Solaris 操作系统使用称为微观状态计数的技术收集有关每个事件的执行状态的统计信息。性能分析器显示的事件计时度量对应于在各状态中花费的相对时间长度。CPU 利用率抽样显示实验中所有线程的计时度量的摘要。单击某个抽样可在右侧的"Selection Details"（选择详细信息）面板中显示该抽样的计时度量。

分析和跟踪数据栏

时钟分析、硬件计数器分析和跟踪数据的数据栏针对每个记录的事件显示一个事件标记。事件标记包含随事件一起记录的调用堆栈的颜色编码表示形式。

单击某个事件标记可以在"Selection Details"（选择详细信息）面板中查看有关该事件的信息，并在 "Call Stack"（调用堆栈）面板中查看调用堆栈函数。双击 "Call Stack"（调用堆栈）面板中的函数可以转至 "Source"（源）视图并查看该函数的源代码以及度量。

对于某些类型的数据，事件可能会因重叠而不可见。如果两个或多个事件恰好出现在同一位置，则只能绘制一个事件；如果一个或两个像素内有两个或多个事件，将绘制全部事件，但是可能无法从视觉上区分它们。不管哪种情况，都会在事件下方显示一个小型的灰色勾选标记，指示该事件的边界。可以进行放大来查看事件。可以使用左右方向键沿任意方向在事件之间切换，以及显示隐藏的事件。可以通过显示事件密度查看有关事件的更多信息。

事件状态

事件状态显示在条形图中，该条形图将应用程序处于各种状态的时间的分布显示为时间函数。

对于在 Oracle Solaris 上记录的时钟分析数据，事件状态图显示 Oracle Solaris 微状态。事件状态的颜色编码与 CPU 利用率抽样栏相同。

缺省情况下显示事件状态。可以通过单击 "Timeline"（时间线）工具栏中的 "Timeline Settings"（时间线设置）按钮或 "Timeline settings"（时间线设置）图标，然后在 "Settings"（设置）对话框的 "Timeline"（时间线）区域中取消选择 "Event States"（事件状态）来隐藏事件状态。

事件密度

事件密度由一条蓝色的线表示，这条线将事件频率显示为时间函数。

要显示事件密度，请单击 "Timeline"（时间线）工具栏中的 "Timeline Settings"（时间线设置）按钮或 "Timeline settings"（时间线设置）图标，然后在 "Settings"（设置）对话框的 "Timeline"（时间线）区域中选择 "Event Density"（事件密度）。

然后事件密度会立即显示在每种数据类型的时间线数据栏的下方。事件密度显示每个水平时间段内发生的事件计数。折线图垂直轴的刻度为 0 到该特定数据栏在可见时间范围内的最大事件计数。

根据时间线的缩放设置每个可见时间段内有许多事件时，事件密度可用于识别具有高事件频率的时间段。要找出这样的时间段，您可以放大。然后可以右键单击并选择一个上下文过滤器以仅包括可见时间范围内的数据，并使用其他性能分析器数据视图分析该特定时间段的数据。

## "Source"（源）视图

如果有源代码可用，"Source"（源）视图显示包含所选函数源代码的文件，在每个源代码行左侧的列中，都提供有性能度量注释。

高度量值用黄色高亮显示，指明这些源代码行位于资源使用的热点区域。对于每个热点源代码行，还在滚动条旁边的右边界中显示一个黄色的导航标记。低于热点阈值的非零度量不高亮显示，但用黄色导航标记进行标记。

要快速导航到具有度量的源代码行，请单击右边界中的黄色标记以跳到具有度量的行。要跳到下一个具有度量的代码行，可以右键单击度量本身并选择诸如 "Next Hot Line"（下一个热点行）或 "Next Non-Zero Metric Line"（下一个非零度量行）之类的选项。

可以在 "Settings"（设置）对话框的 "Source/Disassembly"（源/反汇编）标签中设置突出显示度量的阈值。

"Source"（源）视图在源代码的列标题中显示源文件和相应对象文件的完整路径以及装入对象的名称。少数情况下，会使用同一源文件编译多个对象文件，此时 "Source"（源）视图显示包含所选函数的对象文件的性能数据。

如果性能分析器找不到源文件，您可以单击 "Source"（源）视图中的 "Resolve"（解析）按钮，浏览到源文件，或键入指向源文件的路径或浏览到该路径，源代码随后将从新位置显示出来。在执行此操作时，将在实验中创建路径映射，这样下次打开实验时就可以找到源文件了。

有关用于查找实验源代码的过程的说明，请参见“工具如何查找源代码” [189]。

双击 "Function"（函数）视图中的函数打开 "Source"（源）视图时，显示的源文件是该函数的缺省源代码上下文。函数的缺省源上下文是包含函数的第一条指令（对于 C 代码，为函数的左花括号）的文件。紧接着第一条指令，带注释的源文件为函数添加索引行。源窗口用尖括号中的红色斜体文本显示索引行，格式如下：

*<Function: f_name>*

函数可能具有替代源上下文，即包含归属于该函数的指令的其他文件。这些指令可能来自头文件或内联到所选函数中的其他函数。如果存在替代源上下文，则缺省源上下文的开头将包含指示替代源上下文所在位置的扩展索引行列表。

*<Function: f, instructions from source file src.h>*

双击引用其他源上下文的索引行，将在与索引函数关联的位置打开包含该源上下文的文件。

为了便于导航，替代源上下文也以一个索引行列表开头，通过这些索引行可以返回到缺省源上下文和替代源上下文中定义的函数。

源代码与选择显示的编译器注释交错显示。可以在 "Settings"（设置）对话框中设置显示的注释类别。可以在 `.er.rc` 缺省值文件中设置缺省类别。

可以更改或重新组织 "Source"（源）视图中显示的度量。有关详细信息，请参见 "Help"（帮助）菜单。

有关 "Source"（源）视图内容的详细信息，请参见“性能分析器 "Source"（源）视图布局” [190]。

## "Call Tree"（调用树）视图

"Call Tree"（调用树）视图将程序的动态调用图显示为树，其中每个函数调用显示为可以展开和折叠的节点。展开的函数节点显示由该函数生成的所有函数调用，以及这些函数调用的性能度量。

当您选择某个节点时，"Selection Details"（选择详细信息）窗口将显示该函数调用及其被调用方的度量。归属度量的百分比是总程序度量的百分比。树的缺省根目录是 `<Total>`，这不是一个函数，而是表示程序的所有函数的 100% 的性能度量。

使用 "Call Tree"（调用树）视图可以查看特定调用跟踪的详细信息并分析哪些跟踪具有最大的性能影响。可以通过程序结构进行导航，搜索高度量值。

---

提示 - 要轻松查找花费时间最多的分支，请右键单击任一节点，然后选择 "Expand Hottest Branch"（展开最热分支）。

---

要为所选分支或所选函数设置预定义过滤器，可以在 "Call Tree"（调用树）视图中右键单击打开上下文菜单。采用这种方法过滤，您可以筛选掉所有分析器视图中您不感兴趣的区域的数据。

## "Callers-Callees"（调用方-被调用方）视图

"Callers-Callees"（调用方-被调用方）视图显示代码中函数之间的调用关系，同时显示性能度量。"Callers-Callees"（调用方-被调用方）视图允许通过一次为一个调用构建调用堆栈片段详细检查代码分支的度量。

该视图显示三个独立的面板："Callers"（调用方）面板位于顶部，"Stack Fragment"（堆栈片段）面板位于中部，"Callees"（被调用方）面板位于下部。首次打开 "Callers-Callees"（调用方-被调用方）视图时，"Stack Fragment"（堆栈片段）面板中的函数即为在其他分析器视图之一中选择的函数，例如，"Function"（函数）视图或 "Source"（源）视图。"Callers"（调用方）面板列出用于调用 "Stack Fragment"（堆栈片段）面板中函数的函数，"Callees"（被调用方）面板列出由 "Stack Fragment"（堆栈片段）面板中函数调用的函数。

通过将调用方或被调用方添加到调用堆栈，可以以一次一个调用的方式围绕中心函数构造调用堆栈片段。

要添加对堆栈片段的调用，可在 "Callers"（调用方）窗格或 "Callees"（被调用方）窗格中双击一个函数，也可以选择一个函数并单击 "Add"（添加）按钮。

要删除函数调用，可双击位于调用堆栈片段顶部或底部的函数，或者选择顶部或底部函数并单击 "Remove"（删除）。

---

提示 - 要通过上下文菜单执行 "Add"（添加）和 "Remove"（删除）任务，可右键单击一个函数并选择适当的命令。

---

要将函数设置为调用堆栈片段的头部（顶部）、中心或尾部（底部），请选择函数并单击 "Set Head"（设置头部）、"Set Center"（设置中心）或 "Set Tail"（设置尾部）。这种新的排序可使当前位于调用堆栈片段中的其他函数，相对于所选函数在堆栈片段中的新位置，移动到 "Callers"（调用方）或 "Callees"（被调用方）区域中的相应位置。

使用位于 "Stack Fragment"（堆栈片段）面板上方的 "Back"（后退）和 "Forward"（前进）按钮查看您对调用堆栈片段的更改历史记录。

当您在堆栈片段中添加和删除函数时，将为整个片段计算度量，并将结果显示在片段中的最后一个函数旁边。

可以在 "Callers-Callees"（调用方-被调用方）视图的任意面板中选择一个函数，然后右键单击打开上下文菜单并选择过滤器。将根据您在此视图以及所有分析器数据视图中所做的选择来过滤数据。有关使用上下文过滤器的详细信息，请参见联机帮助。

"Callers-Callees"（调用方-被调用方）视图显示归属度量：

- 对于 "Stack Fragment"（堆栈片段）面板中的调用堆栈片段，归属度量表示该调用堆栈片段的独占度量。
- 对于被调用方，归属度量表示被调用方度量中归属于调用堆栈片段调用的那一部分。被调用方的归属度量与调用堆栈片段之和应该等于调用堆栈片段的度量。
- 对于调用方，归属度量表示调用堆栈片段的度量中归属于调用方的调用的那一部分。所有调用方的归属度量之和同样应等于调用堆栈片段的度量。

有关度量的更多信息，请参见“函数级度量：独占、非独占和归属” [32]。

# Index Objects（索引对象）视图

每个 "Index Objects"（索引对象）视图显示归属于各种索引对象的数据的度量值，例如，"Threads"（线程）、"CPUs" (CPU) 和 "Seconds"（秒）。由于索引对象不是分层结构，因此不显示非独占度量和独占度量。对于每一种类型，只显示一个度量。

有多个预定义的 "Index Objects"（索引对象）视图："Threads"（线程）、"CPUs" (CPU)、"Samples"（抽样）、"Seconds"（秒）、"Processes"（进程）和 "Experiment IDs"（实验 ID）。这些视图将在下文中分别予以说明。

还可以定义定制索引对象。单击 "Settings"（设置）对话框中的 "Add Custom Index View"（添加定制索引视图）按钮，为 "Add Index Objects"（添加索引对象）对话框中的对象设置值。还可以在 `.er.rc` 文件中使用 `indxobj_define` 指令定义索引对象。（请参见“`indxobj_define` *indxobj-type index-exp*” [138]。）

## "Threads"（线程）视图

"Threads"（线程）视图显示线程及相应度量的列表。线程使用进程和线程对表示，并且缺省情况下显示 CPU 总时间。如果装入的实验中存在其他度量，则缺省情况下还会显示这些度量。缺省情况下，不显示 "Threads"（线程）视图。您可以从 "Views"（视图）菜单选择它。

您可以使用过滤器按钮来过滤在此视图和性能分析器视图中显示的数据。

## "CPUs" (CPU) 视图

"CPUs" (CPU) 视图显示处理目标应用程序运行的 CPU 及其度量的列表。CPU 使用 CPU 编号表示，并且缺省情况下显示 CPU 总时间。如果装入的实验中存在其他度量，则缺省情况下还会显示这些度量。如果 "CPUs" (CPU) 视图不可见，可以从 "Views"（视图）菜单中选择。

您可以使用过滤器按钮来过滤在此视图和性能分析器视图中显示的数据。

## "Samples"（抽样）视图

"Samples"（抽样）视图显示抽样点及其度量的列表，这些度量反映在已装入实验的每个抽样点处记录的微观状态。抽样使用抽样编号进行表示，并在缺省情况下显示 CPU 总时间。如果在 "Overview"（概述）面板或 "Settings"（设置）对话框中选择了其他度量，则这些度量也会显示。如果 "Samples"（抽样）视图不可见，可以从 "Views"（视图）菜单中选择。

您可以使用过滤器按钮来过滤在此视图和性能分析器视图中显示的数据。

### "Seconds"（秒）视图

"Seconds"（秒）视图显示在实验中捕获的每秒程序运行情况，以及在那一秒内收集的度量。"Seconds"（秒）视图与 "Samples"（抽样）视图的不同之处在于，前者显示定期抽样，从 0 开始每一秒都进行抽样，而且无法更改间隔。"Seconds"（秒）视图缺省情况下列出执行秒数和 CPU 总时间。如果装入的实验中存在其他度量，则还会显示这些度量。如果在 "Overview"（概述）中或使用 "Settings"（设置）对话框选择了其他度量，则那些度量也会显示。

您可以使用过滤器按钮来过滤在此视图和性能分析器视图中显示的数据。

### "Processes"（进程）视图

"Processes"（进程）视图显示应用程序创建的进程及其度量的列表。进程使用进程 ID (process ID, PID) 编号表示，缺省情况下显示"Total CPU time"（CPU 总时间）度量。如果装入的实验中存在其他度量，则还会显示这些度量。如果在 "Overview"（概述）中或使用 "Settings"（设置）对话框选择了其他度量，则那些度量也会显示。

使用 "Processes"（进程）视图可以查找使用最多资源的进程。如果存在您希望使用其他视图隔离或浏览的特定进程集，则可以使用上下文菜单中提供的过滤器来过滤出其他进程。

### "Experiment IDs"（实验 ID）视图

"Experiment IDs"（实验 ID）视图显示应用程序创建的进程及其度量的列表。实验 ID 使用进程 ID (process ID, PID) 编号表示，缺省情况下显示 CPU 总时间度量。如果装入的实验中存在其他度量，则还会显示这些度量。如果在 "Overview"（概述）中或使用 "Settings"（设置）对话框选择了其他度量，则那些度量也会显示。度量值反映了装入的实验中在每个采样点记录的微状态。这些值反映了所装入实验的每个实验中记录的度量的值或百分比。

## "MemoryObjects"（内存对象）视图

每个 "MemoryObjects"（内存对象）视图显示归属于各种内存对象（例如页）的数据空间度量的度量值。如果一个或多个装入的实验包含数据空间分析，您可以在 "Settings"（设置）对话框的 "Views"（视图）标签中选择要显示其视图的内存对象。可以显示任意数量的 "MemoryObjects"（内存对象）视图。

预定义了多种 "MemoryObjects"（内存对象）视图。为虚拟页面和物理页面预定义了内存对象，名称包括 Vpage_8K、Ppage_8K、Vpage_64K 等等。还可以定义定制内存对象。单击 "Settings"（设置）对话框中的 "Add Custom Object"（添加定制对象）按钮，然后为 "Add Memory Objects"（添加内存对象）对话框中的对象设置值。还可以在 `.er.rc` 文件中使用 `mobj_define` 指令定义内存对象。（请参见“`mobj_define` *mobj-type index-exp*” [139]。）

## "DataLayout"（数据布局）视图

"DataLayout"（数据布局）视图显示所有程序数据对象的带注释的数据对象布局，其中还包括这些对象的数据派生度量数据。该视图仅适用于包括数据空间分析的实验，包括数据空间分析的实验是对硬件计数器溢出分析的扩展。有关更多信息，请参见“数据空间分析和内存空间分析” [165]。

总的来说，视图中显示的布局按照结构的数据排序度量值进行排序。视图显示每个聚集的数据对象及归属于该对象的总度量，后跟数据对象中的所有元素（按偏移量顺序）。每个元素相应地具有其自己的度量，并且在一个 32 字节的块中指示其大小和位置。

要显示 "DataLayout"（数据布局）视图，可在 "Settings"（设置）对话框的 "Views"（视图）标签中选择它（请参见“视图设置” [114]）。与 "DataObjects"（数据对象）视图相同，仅在一个或多个装入的实验包含数据空间分析时 "DataLayout"（数据布局）视图才可见。

要选择单个数据对象，请单击该对象。

要选择在视图中连续显示的多个对象，请选择第一个对象，然后按住 Shift 键并单击最后一个对象。

要选择在视图中不连续显示的多个对象，请选择第一个对象，然后通过按住 Ctrl 键并单击每个对象来选择其他对象。

## "DataObjects"（数据对象）视图

"DataObjects"（数据对象）视图显示数据对象及其度量的列表。该视图仅适用于包括数据空间分析的实验，包括数据空间分析的实验是对硬件计数器溢出分析的扩展。有关更多信息，请参见“数据空间分析和内存空间分析” [165]。

要显示该视图，可在 "Settings"（设置）对话框的 "Views"（视图）标签中选择它（请参见“视图设置” [114]）。仅在一个或多个装入的实验包含数据空间分析时 "DataObjects"（数据对象）视图才可见。

该视图显示程序中的各种数据结构和变量的硬件计数器内存操作度量。

要选择单个数据对象，请单击该对象。

要选择在视图中连续显示的多个对象，请选择第一个对象，然后按住 Shift 键并单击最后一个对象。

要选择在视图中不连续显示的多个对象，请选择第一个对象，然后通过按住 Ctrl 键并单击每个对象来选择其他对象。

## I/O 视图

使用 I/O 视图可以识别应用程序的 I/O 模式并确定影响其性能的 I/O 瓶颈。如果针对 I/O 跟踪数据分析了应用程序，则 I/O 视图可用。

可以根据以下选项之一聚集 I/O 数据：

| | |
|---|---|
| 文件名 | 以表显示程序访问的文件。每行代表一个文件。每行的度量代表针对该文件的所有访问聚集的 I/O 统计信息。 |
| 文件描述符 | 以表显示程序访问的文件的文件描述符。每行代表单个打开的文件实例。如果同一文件打开了多次，则该表有多行对应同一文件。每行的度量适用于单个打开的文件实例。 |
| 调用堆栈 | 以表显示列有任意堆栈编号的调用堆栈。单击某个堆栈时，该堆栈中的函数调用将显示在 "Call Stack"（调用堆栈）面板中。度量适用于所选的调用堆栈。 |

## "Heap"（堆）视图

"Heap"（堆）视图显示具有指示可能存在内存泄漏的内存分配度量的调用堆栈列表。调用堆栈使用任意的堆栈编号进行标识。一个调用堆栈用于指示内存使用量峰值。

单击某个调用堆栈将在 "Selection Details"（选择详细信息）面板中显示度量详细信息，并在 "Call Stack"（调用堆栈）面板中显示调用堆栈的函数调用。可以在 "Call Stack"（调用堆栈）面板中双击某个函数以查看源代码。还可以设置过滤器以过滤掉选择的调用堆栈，或者过滤掉未选择的调用堆栈。仅当一个或多个装入的实验包含堆跟踪数据时，才可以在 "Settings"（设置）对话框的 "Views"（视图）标签中选择 "Heap"（堆）视图。"Heap"（堆）视图的底部面板显示代表完整目标应用程序的 `<Total>` pseudo 函数的详细数据。

## "Data Size"（数据大小）视图

针对包含的数据具有大小元素（例如字节数）的实验提供了 "Data Size"（数据大小）视图。包括堆跟踪、I/O 跟踪或 MPI 跟踪的实验具有 "Data Size"（数据大小）视图。

"Data Size"（数据大小）视图将数据组织为数据大小范围，并为其数据落在某个给定范围内的事件计算度量。没有大小元素的数据归属到数据大小 0。

您可以使用 "Data Size"（数据大小）视图来过滤数据。例如，在包含堆跟踪数据的实验中，您可以选择对于 "Bytes Leaked"（泄漏的字节数）具有较高度量的一个大小范围行并添加过滤器 "Include only events with selected items"（仅包括含有所选项的事件）。当转到其他数据视图时，将对数据进行过滤以仅显示产生的内存泄漏处于您选择的大小范围内的事件。

## "Duration"（持续时间）视图

针对包含的数据具有持续时间的实验提供了 "Duration"（持续时间）视图。包括 I/O 跟踪数据、MPI 跟踪、堆跟踪和同步跟踪数据的实验具有 "Duration"（持续时间）视图。

"Duration"（持续时间）视图将数据组织为持续时间范围，并为其数据落在各个持续时间范围内的事件计算度量。对于 I/O 跟踪、MPI 跟踪和同步跟踪，都会记录函数调用的持续时间。对于堆跟踪，持续时间是指分配内存与释放内存之间的时间。没有持续时间元素的数据归属到持续时间 0。

您可以使用 "Duration"（持续时间）视图来过滤数据。例如，在包含堆跟踪数据的实验中，您可以选择对于 "Bytes Allocated"（分配的字节数）具有较高度量的一个持续时间范围行并添加过滤器 "Include only events with selected items"（仅包括含有所选项的事件）。当转到其他数据视图时，将对数据进行过滤以仅显示其持续时间与您选择的范围匹配的事件，这些事件可能表明内存分配的持续时间比预期时间要长。

## "OpenMP Parallel Region"（OpenMP 并行区域）视图

对于使用通过 Oracle Solaris Studio 编译器编译的 OpenMP 任务的程序，"OpenMP Parallel Region"（OpenMP 并行区域）视图仅适用于使用 OpenMP 3.0 收集器记录的实验。有关更多信息，请参见“OpenMP 分析的限制” [49]。

该视图列出程序执行期间遇到的所有并行区域和从相同分析数据计算得出的度量值。针对当前并行区域计算独占度量。非独占度量反映嵌套并行性。它们归属于当前并行区域以及从中创建该区域的父代并行区域。归属进一步递归追溯，最终将追溯到最顶层的隐式 OpenMP 并行区域，该区域表示程序的串行执行（在任何并行区域之外）。如果程序中不存在嵌套并行区域，独占度量和非独占度量具有相同的值。

如果多次调用包含并行区域的函数，并行区域的所有实例将聚集在一起，呈现为相应视图中的一个行项目。

该视图对导航很有用。您可以选择感兴趣的项，例如具有最长 OpenMP 等待时间的并行区域，然后分析其源代码或者选择一个上下文过滤器以便仅包括与选定的项相关

的数据。然后，您可以通过使用其他视图（"Functions"（函数）、"Timeline"（时间线）、"Threads"（线程）等）来分析其他程序对象是如何表示数据的。

## "OpenMP Task"（OpenMP 任务）视图

"OpenMP Task"（OpenMP 任务）视图显示 OpenMP 任务及其度量的列表。对于使用通过 Oracle Solaris Studio 编译器编译的 OpenMP 任务的程序，此视图中的选项仅适用于使用 OpenMP 3.0 收集器记录的实验。有关更多信息，请参见“OpenMP 分析的限制” [49]。

该视图列出在程序执行期间遇到的任务，以及从分析数据计算的度量值。独占度量仅应用于当前任务。非独占度量包括 OpenMP 任务的度量及其子任务的度量，父子关系是在任务创建时间建立的。来自隐式并行区域的 OpenMP 任务表示以串行方式执行程序。

如果多次调用包含任务的函数，并行区域的所有实例将聚集在一起，呈现为相应视图中的一个行项目。

该视图对导航很有用。可以选择感兴趣的项目（如具有最高 OpenMP 等待时间的任务），通过单击 "Source"（源）视图分析其源代码。您还可以右键单击选择上下文过滤器，从而只包含与所选项相关的数据，然后使用以下的其他视图来分析其他程序对象如何表示该项：："Functions"（函数）、"Timeline"（时间线）、"Threads"（线程）等等。

## "Lines"（行）视图

"Lines"（行）视图显示包含源行及其度量的列表。

源行标有行所在的函数以及行号和源文件名称。如果函数无行号信息或函数的源文件未知，则在行显示中，函数的所有程序计数器 (program counter, PC) 聚集在一起，显示为该函数的单个条目。在行显示中，来自隐藏了其函数的装入对象中的函数中的 PC 聚集在一起，显示为该装入对象的单个条目。如果从 "Lines"（行）视图中选择一个行，则会在 "Selection Details"（选择详细信息）窗口中显示指定行的所有度量。如果从 "Lines"（行）视图中选择一个行后选择 "Source"（源）或 "Disassembly"（反汇编）视图，则会将显示定位到相应的行。

## "PCs" (PC) 视图

"PCs" (PC) 视图列出程序计数器 (program counter, PC)及相应度量。PC 标有其所来自的函数及在该函数中的偏移。在 PC 显示中，隐藏了函数的装入对象中的函数中的 PC 聚集在一起，显示为装入对象的单个条目。在 "PCs" (PC) 视图中选择一行，会在 "Summary"（摘要）标签中显示该 PC 的所有度量。如果从 "PCs" (PC) 视图中选择一个行后选择 "Source"（源）视图或 "Disassembly"（反汇编）视图，则会将显示定位到相应的行。

有关 PC 的更多信息，请参见“调用堆栈和程序执行” [167]一节。

## "Disassembly"（反汇编）视图

"Disassembly"（反汇编）视图显示包含所选函数的对象文件的反汇编列表，且每条指令带有性能度量注释。要查看反汇编代码列表，需要在工具栏的 "View Mode"（查看模式）列表中选择 "Machine"（计算机）。

反汇编列表中插入了源代码（如果有）以及所有选择显示的编译器注释。在 "Disassembly"（反汇编）视图中查找源文件的算法与在 "Source"（源）视图中使用的算法相同。

与 "Source"（源）视图一样，"Disassembly"（反汇编）视图中也显示索引行。但与 "Source"（源）视图不同的是，替代源上下文的索引行不能直接用于导航。此外，替代源上下文的索引行显示在 `#include`d 或内联代码插入位置的开头，而不是仅在 "Disassembly"（反汇编）视图的开头列出。

`#include`d 代码或来自其他文件的内联代码显示为原始反汇编指令，不与源代码交叉在一起显示。但是，将光标置于这些指令之一上并选择 "Source"（源）视图可以打开包含 `#include`d 代码或内联代码的源文件。显示此文件时选择 "Disassembly"（反汇编）视图将在新上下文中打开 "Disassembly"（反汇编）视图，从而显示插入了源代码的反汇编代码。

可以在 "Settings"（设置）对话框中设置显示的注释类别。通过单击对话框中的 "Save"（保存）按钮，可在 `.er.rc` 缺省值文件中设置缺省类。

性能分析器突出显示度量等于或大于特定于度量的阈值的热点行，以便于查找重要的行。可以在 "Settings"（设置）对话框中设置阈值。

在 "Source"（源）视图中，对于每个带有度量的源代码行，在滚动条旁边的右边界中将显示黄色的导航标记。低于热点阈值的非零度量不高亮显示，但用黄色导航标记进行标记。要快速导航到具有度量的源代码行，可以单击右边界中的黄色标记以跳到具有度量的行。还可以右键单击度量本身并选择诸如 "Next Hot Line"（下一个热点行）或 "Next Non-Zero Metric Line"（下一个非零度量行）之类的选项以跳到下一个具有度量的代码行。

有关 "Disassembly"（反汇编）视图内容的详细信息，请参见“带注释的反汇编代码” [196]。

## "Source/Disassembly"（源/反汇编）视图

"Source/Disassembly"（源/反汇编）视图在上面的窗格中显示带注释的源代码，在下面的窗格中显示带注释的反汇编代码。这些窗格是协同工作的，因此在一个窗格中选择行时，在另一个窗格中也会选中相关行。缺省情况下，该视图不显示。

## "Races"（争用）视图

"Races"（争用）视图显示在数据争用实验中检测到的所有数据争用列表。可以单击数据争用，然后在右侧面板的 "Race Details"（争用详细信息）窗口中查看有关它的详细信息。有关更多信息，请参见《Oracle Solaris Studio 12.4：线程分析器用户指南》。

## "Deadlocks"（死锁）视图

"Deadlocks"（死锁）视图显示在死锁实验中检测到的所有死锁列表。可以单击死锁，然后在右侧面板的 "Deadlock Details"（死锁详细信息）窗口中查看有关它的详细信息。有关更多信息，可按 F1 以查看帮助并参见《Oracle Solaris Studio 12.4：线程分析器用户指南》。

## "Dual Source"（双源）视图

"Dual Source"（双源）视图显示所选数据争用或死锁涉及到的两个源上下文。仅当装入了数据争用检测或死锁实验时才会显示此视图。有关更多信息，请参见《Oracle Solaris Studio 12.4：线程分析器用户指南》。

## "Statistics"（统计信息）视图

"Statistics"（统计信息）视图显示所选的实验和抽样的各种系统统计信息的总和。总和后面是每个实验的选定抽样的统计信息。有关显示的统计信息的信息，请参见 `getrusage`(3C) 和 `proc`(4) 手册页。

## "Experiments"（实验）视图

"Experiments"（实验）视图分为两个面板。上面的面板包含一个树，树中包含所有装入实验中的装入对象和每个装入实验的节点。展开 "Load Objects"（装入对象）节点时，将显示所有装入对象的列表（附带有关装入对象的处理情况的各种消息）。展开实验节点时，将显示两个区域："Notes"（说明）区域和 "Info"（信息）区域。

"Notes"（说明）区域显示实验中的所有说明文件的内容。可以通过直接在 "Notes"（说明）区域中键入内容来编辑说明。"Notes"（说明）区域包含其自己的工具栏，其中有用来保存或丢弃说明以及撤消或重做自上次保存以后的所有编辑的按钮。

"Info"（信息）区域包含有关收集的实验以及收集目标访问的装入对象的信息，包括在处理实验或装入对象过程中生成的所有错误消息或警告消息。

底部的面板列出来自性能分析器会话的错误消息和警告消息。

# "Inst-Freq"（指令频率）视图

Inst-Freq（指令频率）视图显示计数数据实验中各类指令执行频率的摘要，该数据在 `collect -c` 中收集。该视图还显示有关装入、存储和浮点指令的执行频率的数据。此外，该视图还包含有关取消的指令和分支延迟槽中的指令的信息。

# "MPI Timeline"（MPI 时间线）视图

"MPI Timeline"（MPI 时间线）视图显示一组水平栏，这些水平栏与 MPI 实验中的进程一一对应，其间以指示消息的对角线连接。每个栏中的区域都根据所在的 MPI 函数着色，或者表明该进程不在 MPI 内（即，位于应用程序代码中的其他位置）。

选择栏中的某个区域或消息线，可在 "MPI Timeline Controls"（MPI 时间线控件）窗口中显示有关该选定区域的详细信息。

拖动鼠标可使 "MPI Timeline"（MPI 时间线）视图沿水平（时间）轴或垂直（进程）轴放大，具体取决于拖动的主方向。

可以将 "MPI Timeline"（MPI 时间线）的图像输出到 `.jpg` 文件。选择 "File"（文件）-> "Export"（导出），然后选择 "Export as JPEG"（导出为 JPEG）。

## MPI Timeline Controls（MPI 时间线控件）

"MPI Timeline Controls"（MPI 时间线控件）窗口支持对 "MPI Timeline"（MPI 时间线）视图进行缩放、平移、事件移步和过滤。它包括一个可调整 "MPI Timeline"（MPI 时间线）上显示的 MPI 消息百分比的控件。

通过过滤，可使得当前视图字段以外的数据不再出现在 "MPI Timeline"（MPI 时间线）视图和 "MPI Chart"（MPI 图表）视图中显示的数据集中。单击 "Filter"（过滤器）按钮可应用过滤器。使用反向过滤器按钮撤消上一个过滤器；使用正向过滤器按钮重新应用一个过滤器。过滤器在 "MPI Timeline"（MPI 时间线）视图和 "MPI Chart"（MPI 图表）视图之间共享，但不适用于其他数据视图。

可以调整消息滑块来控制显示消息的百分比。选择小于 100% 时，成本最高的消息具有优先权。"Cost"（成本）定义为消息的发送和接收功能所用的时间。

"MPI Timeline Controls"（MPI 时间线控件）窗口还显示在 "MPI Timeline"（MPI 时间线）视图中选择的函数或消息的详细信息。

## "MPI Chart"（MPI 图表）视图

"MPI Chart"（MPI 图表）视图显示 "MPI Timeline"（MPI 时间线）视图中显示的 MPI 跟踪数据的图表。它可显示与 MPI 执行有关的数据图。更改 "MPI Chart"（MPI 图表）视图中的控件然后单击 "Redraw"（重画），将显示一个新的图表。选择图表中的某个元素可在 "MPI Chart Controls"（MPI 图表控件）视图中显示有关该元素的更多详细信息。

拖动鼠标可使 "MPI Chart"（MPI 图表）视图放大拖动鼠标形成的矩形区域。

可以将 "MPI Chart"（MPI 图表）的图像输出到 `.jpg` 文件。选择 "File"（文件）-> "Export"（导出），然后选择 "Export as JPEG"（导出为 JPEG）。

### MPI Chart Controls（MPI 图表控件）

"MPI Chart"（MPI 图表）视图具有一组下拉式列表，可用于控制图表的类型、X 和 Y 轴参数以及用于聚集数据的度量和运算符。单击 "Redraw"（重画）将可绘制一个新图形。

通过过滤，可使得当前视图字段以外的数据不再出现在 "MPI Timeline"（MPI 时间线）视图和 "MPI Chart"（MPI 图表）视图中显示的数据集中。要应用过滤器，请单击 "Filter"（过滤器）按钮。单击反向过滤器按钮可撤消上一个过滤器；单击正向过滤器按钮可重新应用一个过滤器。

"MPI Chart Controls"（MPI 图表控件）窗口也可用于显示 "MPI Chart"（MPI 图表）视图中选项的详细信息。

# 设置库和类可见性

缺省情况下，性能分析器显示目标程序和该程序使用的所有共享库和类的函数数据。可以使用 "Library and Class Visibility"（库和类可见性）对话框隐藏任何库或类的函数数据。

要打开此对话框，请选择 "Tools"（工具）> "Library Visibility"（库可见性）(Alt-T, H) 或单击 "Library Visibility"（库可见性）工具栏按钮。

"Library and Class Visibility"（库和类可见性）对话框将列出实验中的所有共享库和类。对于每个列出的项目，可以选择以下一个可见性级别：

| | |
|---|---|
| 函数 | 库或类的所有函数都可见。如果适用，将显示每个函数的度量。 |
| API | 仅显示表示对库或类的调用的函数。这些函数之下的调用，无论是在该库中还是在其他库中（包括回调），都不会显示。 |
| 库 | 仅库或类的名称可见；隐藏所有内部函数。库的度量将反映内部函数引起的度量聚集。为列表中的所有条目选中 "Library"（库）复选框将允许您按库或类执行分析。 |

通过 "Filters"（过滤器）字段，可以更新部分库和类的可见性设置。对于装入大量库的程序，该列表会非常大。这些过滤器仅在对话框中起作用，与数据视图过滤器无关。

设置库可见性时，与隐藏的函数对应的度量在所有显示中仍以某种形式表示。这与数据视图过滤器不同，该过滤器将从显示中删除数据。

# 过滤数据

打开实验时，可以看到程序的所有分析数据。利用过滤，可以从视图中临时删除不感兴趣的数据，从而可以聚焦于程序的特定区域或特性。

在一个视图中应用的过滤器会影响所有视图。例如，可以在 "Timeline"（时间线）视图中指定时间段过滤器，从而使 "Functions"（函数）等其他视图仅显示与过滤的时间段相关的度量。在视图中选择一个或多个项目，然后以每次一个的方式选择过滤器，以指定要包括在视图中的数据。

可以通过多种方法进行过滤：

- 单击 "Filter"（过滤器） 按钮打开可以为当前数据视图中的所选项目应用的过滤器列表。
- 右键单击数据视图中的一个项目，或在选中该项目时按 Shift-F10，然后选择要应用的过滤器。
- 使用性能分析器左下角的 "Active Filters"（活动过滤器）面板查看已经应用的过滤器并进行添加或删除。

可以将各个过滤器结合使用，显示非常具有针对性的程序运行区域的度量。例如，可以在 "Functions"（函数）视图中应用一个过滤器并在 "Timeline"（时间线）视图中应用一个过滤器，以便聚焦于包括程序运行的特定时间段中特定函数的调用堆栈。

使用过滤器时，会在所有性能分析器视图中过滤数据，但 "MPI Timeline"（MPI 时间线）视图除外，该视图具有不与其他数据视图交互的单独过滤机制。

对性能分析器具有丰富经验的用户还可以使用 "Advanced Custom Filters"（高级定制过滤器）对话框编辑过滤器表达式创建定制过滤器，从而精确定义要显示的数据。

注 - 此处所述的过滤器不同于“MPI Timeline Controls（MPI 时间线控件）” [105]和“MPI Chart Controls（MPI 图表控件）” [106]中所述的 MPI 过滤。这些过滤器不会影响 "MPI Timeline"（MPI 时间线）视图和 "MPI Chart"（MPI 图表）视图。

# 使用过滤器

在性能分析器的大多数数据视图中都可以使用过滤器。使用工具栏或 "Active Filters"（活动过滤器）面板上的 "Filters"（过滤器）按钮，或单击鼠标右键或者在键盘上按 Shift-F10 可访问过滤器。添加过滤器后，将立即过滤数据。

使用过滤器时，通常在视图中选择一项或多项您希望重点关注的项目，然后选择合适的过滤器。对于大多数视图，使用过滤器可以包括或排除满足在过滤器中指定的标准的数据。这样，您便可以使用过滤器，将重点放在程序特定区域中的数据，或者排除特定区域的数据。

一些使用过滤器的方法包括：

- 添加多个过滤器来缩小数据的范围。过滤器按照逻辑 AND 的关系进行组合，这种逻辑关系要求数据与所有过滤器匹配。
- 在一个视图上添加过滤器，然后在另一个视图上检查过滤后的数据。例如，在 "Call Tree"（调用树）视图中，可以查找最热的分支，并选择 "Add Filter: Include only stacks containing the selected branch"（添加过滤器：仅包括含有所选分支的堆栈），然后转到 "Function"（函数）视图即可查看在该代码分支中调用的函数的度量。
- 在多个视图中添加多个过滤器可以创建非常具有针对性的数据集。
- 使用过滤器作为基础来创建高级定制过滤器。请参见“使用高级定制过滤器” [108]。

# 使用高级定制过滤器

在性能分析器数据视图中添加过滤器时，将生成过滤器表达式，并且会立即应用来过滤数据。生成的过滤器表达式在 "Advanced Custom Filter"（高级定制过滤器）对话框中可见。经验丰富的用户可以将这些生成的过滤器表达式用作创建定制过滤器的起点。

创建定制过滤器：

1. 通过执行以下操作之一打开 "Advanced Custom Filter"（高级定制过滤器）对话框：
   - 单击 "Filter"（过滤器）按钮并选择 "Add Filter: Advanced Custom Filter"（添加过滤器：高级定制过滤器）
   - 选择 "Tools"（工具）⇒ "Filters"（过滤器）⇒ "Add Filter: Advanced Custom Filter"（添加过滤器：高级定制过滤器）
2. 在 "Filter Specification"（过滤器规范）文本框中单击并编辑过滤器。有关过滤器的更多信息，请参见下文。

3. 如有必要，使用方向键撤消或重做编辑。
4. 单击 "OK"（确定）根据过滤器表达式过滤数据并关闭对话框。

"Filter Specification"（过滤器规范）面板显示以前应用的过滤器（通过在性能分析器数据视图中选择相应的过滤器）的过滤器表达式。可以编辑这些过滤器，并使用顶部的箭头按钮撤消和重做所进行的编辑。您也可以采用与文本编辑器中相同的方法，按 Ctrl-Z 组合键进行撤消，按 Shift-Ctrl-Z 组合键进行重做。单击 "OK"（确定）时，这些过滤器仅影响数据视图。

将每个新过滤器放置在新行中，以逻辑 AND 运算符 `&&` 开头。要显示实验数据，这些数据必须与第一个过滤器匹配、并且与第二个过滤器匹配、并且与第三个过滤器匹配，等等。

如果希望数据匹配第一个过滤器 OR 第二个过滤器，则可以将 `&&` 更改为 `||`。

过滤器表达式使用标准的 C 关系运算符（`==`、`>=`、`&&`、`||` 等等）以及实验特定的关键字。"Advanced Custom Filters"（高级定制过滤器）对话框的 "Keywords"（关键字）面板中显示了可以在实验中使用的关键字。

有关过滤器关键字和过滤器表达式的更多信息，请搜索性能分析器帮助。

过滤器表达式语法与 `er_print` 中使用的过滤语法相同。有关过滤器表达式的信息，请参见“表达式语法” [153]。

# 使用标签进行过滤

标签是分配给实验的某一部分的名称。使用 `er_label` 命令，您可以将标签名分配给实验中的一段时间，而标签将保存在实验中。通过 `er_print` 命令或性能分析器，您可以使用标签来过滤实验数据，以便包含或排除在所标记时段内收集的数据。

有关如何使用 “为实验加标签” [210] 实用程序创建标签的信息，请参见`Labeling Experiments`。

在性能分析器中，可以在 "Advanced Custom Filter"（高级定制过滤器）对话框中过滤所标记的时间段的数据。在 "Filter Specification"（过滤器规范）面板中键入标签名称，然后单击 "Apply"（应用）来过滤标签所指定的数据。由于标签用作通过 `TSTAMP` 关键字使用数值进行比较的过滤器表达式的昵称，您无需进行任何数值比较。在 "Filter Specification"（过滤器规范）面板中，将标签添加到单独的行中并且在前面放置 `&&`，可以将标签与其他过滤器结合使用。

可以在 "Advanced Custom Filter"（高级定制过滤器）对话框的 "Keywords"（关键字）面板中查看在性能分析器中打开的实验是否分配了标签。您也可以使用 `er_print -describe` 命令查看相同的信息。标签在显示内容中将首先列出，并且包含实际过滤器表达式，该表达式带有由标签实施的 `TSTAMP` 关键字。

应用标签过滤器之后，可以单击 "Timeline"（时间线）视图以便查看按标签定义的间隔删除的数据。其他支持过滤的数据视图上也会对数据进行过滤。

# 从性能分析器分析应用程序

在终端窗口使用 `collect` 命令或使用性能分析器，可以分析应用程序。

要在性能分析器中分析应用程序，请执行以下操作之一：

- 单击 "Welcome"（欢迎）屏幕中的 "Profile Application"（分析应用程序）。
- 单击 "Profile Application"（分析应用程序）工具栏按钮。
- 选择 "File"（文件）-> "Profile Application"（分析应用程序）(Alt-F, E)。
- 运行 `analyzer` 命令，并在命令行上指定目标程序及其参数。

每个方法都将打开 "Profile Application"（分析应用程序）对话框。按 F1 可查看有关该对话框的帮助信息。

"Profile Application"（分析应用程序）对话框中的选项对应于 `collect` 命令中提供的选项，如第 3 章 收集性能数据中所述。

如果要使用缺省分析选项，并且只收集时钟分析数据，只需指定 "Target Program"（目标程序）项。否则，您可以在 "General"（常规）标签中指定实验选项，然后在 "Data to Collect"（要收集的数据）标签上选择要收集的数据的类型。

如果单击 "Preview Command"（预览命令）按钮，则可以查看在单击 "Run"（运行）按钮时将使用的 `collect` 命令。然后单击 "Run"（运行）开始分析、收集数据和创建实验。

目标程序的输出缺省显示在由性能分析器打开的单独终端窗口中。如果关闭 "Profile Application"（分析应用程序）对话框时有实验正在运行，该实验将继续运行。重新打开对话框时，将显示运行中的实验，就像在运行过程中它一直保持打开一样。如果尝试退出性能分析器时有实验正在运行，则会打开一个对话框，询问是终止运行还是允许继续运行。

要停止实验，请单击 "Terminate"（终止）。必须确认您想要停止实验。

还可以分析正在运行的进程（如下一节中所述），以及分析内核（如第 9 章 内核分析中所述）。

# 分析正在运行的进程

在 Oracle Solaris 中，可以在性能分析器中或通过命令行对任何正在运行的进程收集数据。在 Linux 中，正在运行的进程分析仅对单线程应用程序可靠起作用。由于 JVM 是多线程的，因此无法在 Linux 中分析 Java 应用程序。

如果要通过命令行分析正在运行的进程，请参见collect(1)或dbx(1)手册页。

要在性能分析器中分析正在运行的进程，请执行以下操作之一：

- 单击 "Welcome"（欢迎）屏幕中的 "Profile Running Process"（分析运行的进程）。
- 选择 "File"（文件）-> "Profile Running Process"（分析运行的进程）(Alt-F, R)。

在 "Profile Running Process"（分析运行的进程）对话框中选择您要分析的进程，如果要使用缺省选项，可单击 "Run"（运行）。否则，您可以在 "General"（常规）标签中指定实验选项，然后在 "Data to Collect"（要收集的数据）标签上选择要收集的数据的类型。

在对话框处于打开状态时按 F1 可查看该对话框的帮助。

"Output"（输出）标签显示收集器的输出和进程的任何输出。如果要停止分析，请单击 "Terminate"（终止）。性能分析器会提示您打开实验。

# 比较实验

可以在性能分析器中同时装入多个实验或实验组。缺省情况下，在装入同一个可执行文件上的多个实验时，数据将聚集并呈现为好像其为一个实验。同时还可以分别查看数据，从而比较实验中的数据。

要比较性能分析器中的两个实验，可以用常规方式打开第一个实验，然后选择 "File"（文件）> "Add Experiment"（添加实验）装入第二个实验。要比较它们，请在支持比较的视图中右键单击并选择 "Enable Experiment Comparison"（启用实验比较）。

支持比较试验功能的标签有 "Functions"（函数）、"Callers-Callees"（调用方-被调用方）、"Source"（源）、"Disassembly"（反汇编）、"Lines"（行）和 "PCs" (PC)。在比较模式下，实验或组中的数据在这些标签上以相邻列显示。这些列按装入实验或组的顺序显示，同时带有一个给出实验或组名称的附加标题行。

## 设置比较样式

对于比较样式，可以指定以下样式之一：

| | |
|---|---|
| 绝对值 | 显示所有装入的实验的度量值。 |
| 差值 | 显示基线实验和其他装入的实验度量之间的 +/- 差异。 |
| 比率 | 将基线实验和其他装入的实验度量之间的差异显示为比率。例如，比较实验度量可能显示为 `x0.994`，这表示其相对于基本实验的值。 |

单击 "Settings"（设置）对话框中的 "OK"（确定）时，数据视图将更新，显示新比较样式。

# 远程使用性能分析器

可以在都安装了 Oracle Solaris Studio 工具的服务器之间以远程方式使用性能分析器，甚至在无法安装 Oracle Solaris Studio 的桌面客户机上使用它。

## 在桌面客户机上使用性能分析器

您可以在桌面客户机系统上安装特殊版本的性能分析器，然后连接到安装了这些工具的远程服务器。

客户机系统要求：

- 操作系统必须为 Mac OS X、Windows、Linux 或 Oracle Solaris。
- 性能分析器的版本必须与安装在远程系统上的 Oracle Solaris Studio 工具的版本相匹配。
- Java 1.7 或 1.8 必须位于用户的路径。

远程系统的要求：

- 操作系统必须是 Oracle Solaris 或 Linux。
- Secure Shell (SSH) 守护进程 sshd 必须处于运行状态
- 必须可以在远程主机上访问 Oracle Solaris Studio 软件，并且您需要知道该软件的路径。
- 您必须拥有主机上的用户帐户。

在客户机系统上使用的性能分析器版本以 tar 文件的形式提供，可以将该文件复制到任何系统（包括 Mac OS X 或 Windows 计算机）。该版本称为远程性能分析器客户机，利用该版本，可以在任何具有 Java 1.7 或 Java 1.8 的系统上使用性能分析器。

远程性能分析器客户机位于 Oracle Solaris Studio 安装的 */Oracle-Solaris-Studio-install-dir*`/lib/analyzer/RemoteAnalyzer.tar` 中。

在桌面客户机上使用性能分析器之前，必须在客户机上安装特殊远程版本，如以下任务所述。

### ▼ 如何在客户机上使用远程性能分析器

1. **将 `RemoteAnalyzer.tar` 文件复制到要运行它的桌面客户机系统。**
2. **使用文件提取实用程序或命令提取 `RemoteAnalyzer.tar`。**

请注意，在 Windows 上，可能需要安装诸如 WinZip 或 7-Zip 之类的应用程序才能提取该文件。

提取的 `RemoteAnalyzer` 目录包含适用于 Windows、Mac OS X、Linux 和 Solaris 的脚本以及 `README.txt` 文件。

3. **为所使用的系统执行该脚本。**

| | |
|---|---|
| Windows | 双击 `RemoteAnalyzer` 目录中的 `AnalyzerWindows.bat` 文件。 |
| Mac | 双击 `RemoteAnalyzer` 目录中的 `AnalyzerMacOS.command` 文件。 |
| Linux | 在终端窗口中运行 `AnalyzerLinux.sh` 脚本。 |
| Oracle Solaris | 在终端窗口中运行 `AnalyzerSolaris.sh` 脚本。 |

此时将打开性能分析器窗口，其中显示 "Welcome"（欢迎）屏幕，该屏幕仅包含可以远程运行的功能。有关更多信息，请参见 `RemoteAnalyzer/README.txt` 文件。

## 在性能分析器中连接到远程主机

可使用以下方法连接到远程主机：

- 在 "Welcome"（欢迎）视图中单击 "Connect To Remote Host"（连接到远程主机）。
- 选择 "File"（文件）> "Connect To Remote Host"（连接到远程主机）。
- 在 "Performance Analyzer"（性能分析器）窗口的底部单击 "Remote Host"（远程主机）状态消息。

在 "Connect To Remote Host"（连接到远程主机）对话框中按 F1 可了解有关回答提示的信息。

当连接完成时，"Performance Analyzer"（性能分析器）主窗口底部的状态区域显示已连接到主机。请注意，当连接到远程主机时，性能分析器的文件浏览器会自动访问远程主机的文件系统以打开实验。

# 配置设置

可以使用 "Settings"（设置）对话框控制数据和其他配置设置的表示形式。要打开该对话框，请单击工具栏中的 "Setttings"（设置）按钮或选择 "Tools"（工具）-> "Setttings"（设置）。

"Setttings"（设置）分为以下类别：

"OK"（确定）按钮应用对当前会话所做的更改并关闭对话框。"Apply"（应用）按钮应用对当前会话的更改，但保持对话框的开启状态，以便进行其他更改。"Close"（关闭）按钮关闭对话框，不保存或应用任何更改。

"Export"（导出）按钮打开 "Export Settings"（导出设置）对话框，您可以使用该对话框选择要导出的设置以及保存它们的位置。当您在将来的性能分析器会话或当前会话中再次打开该实验时，可以将导出的配置设置应用于该实验。您还可以将配置用于其他实验。有关更多信息，请参见“性能分析器配置文件” [119]。

# 视图设置

视图设置面板列出当前实验适用的数据视图。

标准视图

单击复选框可选择或取消选择要显示在性能分析器中的标准数据视图。

索引对象视图

单击复选框可选择或取消选择要显示的索引对象视图。预定义的索引对象视图包括 "Threads"（线程）、"Cpus" (Cpu)、"Samples"（抽样）、"Seconds"（秒）、"Processes"（进程）。

要添加定制索引对象的视图，请单击 "Add Custom Index View"（添加定制索引视图）按钮以打开 "Add Index Object"（添加索引对象）对话框。您指定的索引对象名称必须尚未定义，并且不能与任何现有命令或任何内存对象类型相同。该名称不区分大小写，必须完全由字母数字字符或 '_' 字符组成，且以字母字符开头。公式必须遵循“表达式语法” [153]中说明的语法。

也可以使用 `er_print` 命令创建索引对象。请参见“控制索引对象列表的命令” [137]

内存对象视图

单击复选框可选择或取消选择要显示的预定义内存对象视图。当实验包含硬件计数器分析数据时，这些视图可用。

内存对象表示内存子系统中的组件，如高速缓存行、页面和内存区。为虚拟页面和物理页面预定义了内存对象，其大小为 8KB、64KB、512KB 和 4 MB。

要添加定制内存对象的视图，请单击 "Add Custom Memory Object View"（添加定制内存对象视图）按钮以打开 "Add Memory Object"（添加内存对象）对话框。您指定的内存对象名称必须尚未定义，并且不能与任何现有命令或任何索引对象类型相同。该名称不区分大小写，必须完全由字母数字字符或 '_' 字符组成，且以字母字符开头。公式必须遵循“表达式语法” [153]中说明的语法。

计算机模型文件

可以针对特定的 SPARC 系统体系结构装入用于定义内存对象的文件。单击 "Load Machine Model"（装入计算机模型）按钮并选择感兴趣的系统体系结构。单击 "Apply"（应用）或 "OK"（确定），此时新的对象列表将显示在 "Memory Objects Views"（内存对象视图）列中。可以从这些视图中进行选择以显示关联的数据。在帮助中搜索“计算机模型”以了解更多信息。

缺省情况下，性能分析器将装入对记录实验所在的计算机适用的计算机模型文件。计算机模型文件可定义内存对象和索引对象。

# 度量设置

使用度量设置可以选择在大多数分析器标签（包括 "Function"（函数）、"Callers-Callees"（调用方-被调用方）、"Source"（源）、"Disassembly"（反汇编）等等）中显示的度量。一些度量可以根据您的选择以时间或百分比显示，而另一些则显示为值。度量列表包括装入的任意实验中可用的所有度量。

每种度量都提供了 "Time"（时间）和 "Percentage"（百分比）复选框，或者 "Value"（值）复选框。选中希望性能分析器显示的度量类型的复选框。单击 "Apply"（应用）可使用新度量更新视图。还可以在 "Overview"（概述）中选择度量，其显示将与此处建立的设置同步。

---

注 - 只能选择显示独占度量和非独占度量。如果显示了独占度量或非独占度量，则归属度量始终显示在 "Call Tree"（调用树）视图中。

---

# 时间线设置

使用时间线设置可以指定显示在“"Timeline"（时间线）视图” [92]中的信息。

Data Types（数据类型）

选择要显示的数据的种类。选定项将应用至所有实验和所有显示类型。如果某个数据类型未包含在实验中，则该数据类型不会作为可选的数据类型显示在设置中。

CPU Utilization Samples（CPU 利用率抽样）

选择该项可为每个进程显示 CPU 利用率抽样栏。该抽样栏显示了一个图形，其中汇总了每个定期抽样的微观状态信息。

Clock Profiling（时钟分析）

选择该项可为每个 LWP、线程、CPU 或实验捕获的时钟分析数据显示时间线栏。每项的时间线栏显示在每个抽样事件处执行的函数的有色调用堆栈。

HW Counter Profiling (keyword)（HW 计数器分析（关键字））

选择该项可显示硬件计数器分析数据的时间线栏。

I/O Tracing（I/O 跟踪）

选择该项可显示 I/O 跟踪数据的时间线栏。

Heap Tracing（堆跟踪）

选择该项可显示堆跟踪数据的时间线栏。

Synchronization Tracing（同步跟踪）

选择该项可显示同步跟踪调用堆栈的时间线栏。

Event States（事件状态）

选择该项可将用于显示每个事件的微观状态的图添加到每个时间线栏。Event Density（事件密度） – 选择该项可将用于显示事件何时发生的图添加到每个时间线栏。

Group Data By（数据分组方式）

指定如何为每个进程（按 LWP、线程、CPU）或总体进程划分时间线栏。还可以使用 "Timeline"（时间线）工具栏中的 "Group Data"（数据分组）列表设置分组。

Call Stack Alignment（调用堆栈对齐）

指定显示在时间线事件标记中的调用堆栈是在叶函数还是根函数处对齐。如果希望最后一个调用的函数显示在堆栈的底部，则选择叶。该设置不影响显示在 "Selection Details"（选择详细信息）面板中的数据，该面板始终将叶函数显示在顶部。

调用堆栈放大（Call Stack Magnification）

指定在调用堆栈中显示每个函数时应当使用多少像素。缺省值为三。此设置（以及控制每行可用空间的时间线垂直缩放）决定了是将截断还是完全显示较深的调用堆栈。

# 源/反汇编设置

使用源/反汇编设置可以选择显示在 "Source"（源）视图、"Disassembly"（反汇编）视图和 "Source/Disassembly"（源/反汇编）视图中的信息。

Compiler Commentary（编译器注释）

选择显示在 "Source"（源）视图和 "Disassembly"（反汇编）视图中的编译器注释的类。

Highlighting Threshold（突出显示阈值）

在 "Source"（源）视图和 "Disassembly"（反汇编）视图中突出显示高度量行的阈值。阈值是归属于显示源代码或反汇编代码的文件中任意行的最大度量值百分比。阈值独立应用到各个度量。

Source Code（源代码）

在 "Disassembly"（反汇编）视图中显示源代码。如果在 "Disassembly"（反汇编）视图中显示源代码，则还将显示启用的类的编译器注释。

Metrics for Source Lines（源代码行的度量）

在 "Disassembly"（反汇编）视图中显示源代码的度量。

Hexadecimal Instructions（十六进制指令）

在 "Disassembly"（反汇编）视图中以十六进制格式显示指令。

Only Show Data of Current Function（仅显示当前函数的数据）

仅对在其他视图中选定的当前函数的指令显示度量。如果选择此选项，则对所有其他指令隐藏度量。

Show Command-line Flags（显示命令行标志）

显示用于编译目标程序的编译器命令和选项。滚动到 "Source"（源）视图的最后一行以查看命令行。

# 调用树设置

调用树设置 "Expand branches when percent of metric exceeds this threshold"（当度量百分比超出此阈值时展开分支）可设置在 "Call Tree"（调用树）视图中展开分支的触发器。如果调用树的某个分支使用度量的百分比达到或低于指定值，则在选择诸如 "Expand Branch"（展开分支）或 "Expand All Branches"（展开所有分支）之类的展开操作时，它不会自动展开。如果它超出该百分比，则会展开。

# 格式设置

使用格式设置可指定其他数据视图格式。

Function Name Style（函数名样式）

指定是以长格式、短格式还是 C++ 函数名称和 Java 方法名称的改编格式显示函数名称。

Append SO name to Function name（将 SO 名称附加到函数名）

选中该复选框可以将函数或方法所在的共享对象的名称附加到该函数或方法名称。

View Mode（查看模式）

设置查看模式工具栏的初始设置，仅对 Java 实验和 OpenMP 实验启用该设置。查看模式 "User"（用户）、"Expert"（专家）和 "Machine"（计算机）设置用于查看实验的缺省模式。使用工具栏中的查看模式列表可以切换当前视图。

对于 Java 实验：

- "User"（用户）模式显示解释的方法和调用的任何本机方法的度量。特殊函数 `<no Java call stack recorded >` 指示 Java 虚拟机 (Java Virtual Machine, JVM) 软件不报告 Java 调用堆栈，即使 Java 程序当时正在运行也是如此。
- "Expert"（专家）模式显示解释的方法和调用的任何本机方法的度量，并列出由 JVM 动态编译的方法。
- "Machine"（计算机）模式将多个 JVM 编译显示为完全独立的函数，但这些函数将具有相同的名称。在该模式中，JVM 软件中的所有函数都按照此方式显示。

有关 Java 实验查看模式的详细说明，请参见“Java 分析查看模式” [171]。

对于 OpenMP 实验：

- "User"（用户）模式显示重构的调用堆栈，这些重构的调用堆栈类似于在不使用 OpenMP 的情况下编译程序时获取的调用堆栈。在 OpenMP 运行时执行某些操作时，显示名称格式为 `<OMP-*>` 的特殊函数。
- "Expert"（专家）模式显示编译器生成的且代表并行化循环、任务等的函数，这些函数会与 "User"（用户）模式中的用户函数聚集。在 OpenMP 运行时执行某些操作时，显示名称格式为 `<OMP-*>` 的特殊函数。
- "Machine"（计算机）模式显示所有不具有任何特殊 `<OMP-*>` 函数的线程的计算机调用堆栈。

有关 OpenMP 实验查看模式的详细说明，请参见“OpenMP 软件执行概述” [173]。

对于所有其他实验，所有三种模式显示同样的数据。

Comparison Style（比较样式）

指定在比较实验时要如何显示数据。例如，比较实验度量可能显示为 x0.994 以指示其相对于基本实验的值。

| | |
|---|---|
| 绝对值 | 显示所有装入的实验的度量值。 |
| 差值 | 显示基线实验和其他装入的实验度量之间的 +/- 差异。 |
| 比率 | 将基线实验和其他装入的实验度量之间的差异显示为比率。 |

# 搜索路径设置

搜索路径设置指定查找所装入实验的关联源代码文件和对象文件的路径，以便在 "Source"（源）和 "Disassembly"（反汇编）视图中显示带注释的源代码数据。搜索路径还可用于在系统上查找 Java 运行时环境的 `.jar` 文件。特殊目录名称 `$expts` 按照装

入实验的顺序引用当前实验集。搜索 `$expts` 时，只查找创建者实验，不检查任何子孙实验。

缺省情况下，搜索路径设置为 `$expts` 和 .（当前目录）。可以通过键入路径或浏览路径，然后单击 "Append"（附加）来添加其他路径。要编辑列表中的路径，请选择一个路径，在 "Paths"（路径）字段中进行编辑，然后单击 "Update"（更新）。要更改搜索顺序，请在列表中选择路径，然后单击 "Move Up/Move Down"（上移/下移）按钮。

有关搜索路径使用方法的更多信息，请参见“工具如何查找源代码” [189]。

## 路径映射设置

使用路径映射设置，可以将文件路径的开头部分从一个位置映射到另一个位置，以帮助性能分析器定位源文件。对于已经从记录的原始位置移动的实验，路径映射很有用。在找到源文件时，性能分析器会在 "Source"（源）和 "Disassembly"（反汇编）视图中显示带注释的源数据。

源路径
: 键入实验中使用的源的路径开头部分。可以通过在性能分析器中打开实验时查看 "Selection Details"（选择详细信息）面板找到该路径。

目标路径
: 键入或浏览至从运行性能分析器的当前位置到源的路径的开头部分。

  例如，如果实验包含指定为 `/a/b/c/d/sourcefile` 的路径，而 `soucefile` 现在位于 `/x`，则可以使用 "Pathmaps"（路径映射）设置将 `/a/b/c/d/` 映射到 `/x/`。可以指定多个路径映射，这样将依次尝试每个路径映射以查找文件。

有关路径映射使用方法的更多信息，请参见“工具如何查找源代码” [189]。

## 性能分析器配置文件

退出工具时，性能分析器会自动保存您的配置设置。您启用的度量和数据视图之类的实验设置将存储在实验中。再次打开同一实验时，其配置与以前关闭实验时一样。

您可以将某些设置保存在名称以 `config.xml` 结尾的配置文件中，当您从 "Open Experiment"（打开实验）对话框打开任何实验时，可将该配置文件应用于实验。可以将配置保存在仅供您使用的位置，也可以保存到供其他用户使用的共享位置。

打开实验时，性能分析器将搜索缺省位置以查找可用的配置文件，并允许您选择适用于所打开的实验的配置。

也可以使用 "Tools"（工具）> "Export Settings as er.rc"（将设置导出为 er.rc）将设置导出到 `.er.rc` 文件，该文件可由 `er_print` 命令读取。这样就可在 `er_print` 和性能分析器中启用相同的度量。

♦ ♦ ♦ 第 5 章

# er_print 命令行性能分析工具

本章介绍如何使用 er_print 实用程序进行性能分析。

本章包含以下主题。

# 关于 er_print

er_print 实用程序输出性能分析器支持的各种数据视图的文本版本。除非将信息重定向到某个文件，否则这些信息将写入标准输出。必须为 er_print 实用程序提供由收集器生成的一个或多个实验或实验组的名称作为参数。

er_print 实用程序仅适用于使用 Oracle Solaris Studio 12.3 和此发行版 (Oracle Solaris Studio 12.4) 记录的实验。如果使用通过任何其他版本记录的实验，将会报告错误。如果具有较旧版本的实验，则必须使用记录实验时所用发行版中的 er_print 版本。

可以使用 er_print 实用程序显示函数以及调用方和被调用方的性能度量、源代码列表和反汇编代码列表、抽样信息、数据空间数据、线程分析数据以及执行统计信息。

在多个实验或实验组上调用时，缺省情况下，er_print 将聚集实验数据，也可用于对实验进行比较。有关更多信息，请参见“compare { on | off | delta | ratio }” [148]。

有关收集器所收集的数据的说明，请参见第 2 章 性能数据。

有关如何使用性能分析器以图形格式显示信息的说明，请参见第 4 章 性能分析器工具和性能分析器的 "Help"（帮助）菜单。

# er_print 语法

er_print 实用程序的命令行语法如下：

er_print [ -script *script* | –*command* | - | -V ] *experiment-list*

er_print 实用程序的选项如下：

–

读取从键盘输入的 er_print 命令。

-script *script*

从文件 *script* 读取命令（该文件包含 er_print 命令的列表，每行一个命令）。如果 -script 选项不存在，则 er_print 从终端或从命令行读取命令。

-*command* [*argument*]

处理给定的命令。

-V

显示版本信息并退出。

er_print 命令行上的多个选项按其出现顺序进行处理。可以按任何顺序混合脚本、连字符和显式命令。未提供任何命令或脚本时的缺省操作是进入交互模式，在该模式下命令是从键盘输入的。要退出交互模式，请键入 quit 或 Ctrl-D。

处理每个命令后，会输出处理时出现的任何错误消息或警告消息。可以使用 procstats 命令输出有关处理的摘要统计信息。

在以下各节中列出了 er_print 实用程序接受的命令。

只要命令是明确的，就可以将其缩写为更短的字符串。通过以反斜杠 \ 结束行的方式可以将一个命令拆分为多行。以 \ 结尾的任何行在解析之前都会将 \ 字符删除，并追加下一行的内容。除可用内存外，对命令可使用的行数并没有限制。

必须将包含嵌入空白的参数用双引号引起来。可以将引号内的文本拆分为多行。

# 度量列表

许多 er_print 命令都使用度量关键字列表。列表的语法如下：

*metric-keyword-1*[:*metric-keyword2*…]

对于动态度量（它们基于度量的数据），度量关键字包含以下三部分：度量类型字符串、度量可见性字符串和度量名称字符串。这些字符串连接在一起且没有空格，如下所示。

*flavorvisibilityname*

对于静态度量 – 它们基于实验中装入对象的静态属性（名称、地址和大小），度量关键字包含度量名称和可选的前置度量可见性字符串，两者连接在一起且没有空格：

[*visibility*]*name*

度量 *flavor* 和度量 *visibility* 字符串由类型和可见性字符组成。

表 5-1 “度量类型字符” 中列出了允许的度量类型字符。包含多个类型字符的度量关键字可扩展为一系列度量关键字。例如，ie.user 可扩展为 i.user:e.user。

**表 5-1** 度量类型字符

| 字符 | 说明 |
|---|---|
| e | 显示独占度量值 |
| i | 显示非独占度量值 |
| a | 显示归属度量值（仅适用于调用方-被调用方度量） |
| d | 显示数据空间度量值（仅适用于数据派生的度量） |

表 5-2 “度量可见性字符” 中列出了允许的度量可见性字符。可见性字符在可见性字符串中的顺序不影响对应度量的显示顺序。例如，`i%.user` 和 `i.%user` 都解释为 `i.user:i%user`。

仅可见性不同的度量始终按标准顺序一起显示。如果仅可见性不同的两个度量关键字由某些其他关键字分隔，则度量按标准顺序出现在两个度量中第一个度量的位置。

表 5-2 度量可见性字符

| 字符 | 说明 |
|---|---|
| . | 将度量显示为时间。适用于计时度量以及度量循环计数的硬件计数器度量。其他度量解释为 "+"。 |
| % | 将度量显示为总程序度量的百分比。对于调用方-被调用方列表中的归属度量，将度量显示为所选函数的非独占度量的百分比。 |
| + | 将度量显示为绝对值。对于硬件计数器，该值是事件计数。计时度量解释为 "."。 |
| ! | 不显示任何度量值。不能与其他可见性字符组合使用。 |

当类型字符串和可见性字符串都具有多个字符时，首先扩展类型。因此，将 `ie.%user` 扩展为 `i.%user:e.%user`，然后将其解释为 `i.user:i%user:e.user:e%user`。

对于静态度量，句点 (.)、加号 (+) 和百分号 (%) 这三种可见性字符在用于定义排序顺序时是等效的。因此，`sort i%user`、`sort i.user` 和 `sort i+user` 均表示只要非独占用户 CPU 时间以任一形式可见，性能分析器就应该按它排序；`sort i!user` 则表示不管非独占用户 CPU 时间是否可见，性能分析器都应该按它排序。

对于每种度量类型，可以使用叹号 (!) 可见性字符覆盖内置可见性缺省值。

如果同一度量在度量列表中出现多次，则仅处理第一次出现的该度量，而忽略随后出现的该度量。如果指定的度量不在列表中，则将它附加到列表中。

表 5-3 “度量名称字符串” 列出了计时度量、同步延迟度量、内存分配度量、MPI 跟踪度量以及两个常见的硬件计数器度量的可用 `er_print` 度量名称字符串。对于其他硬件计数器度量，度量名称字符串与计数器名称相同。通过 `metric_list` 命令，可以显示所装入实验的所有可用度量名称字符串的列表。要列出计数器名称，可发出不带其他参数的 `collect -h` 命令。有关硬件计数器的更多信息，请参见“硬件计数器分析数据” [23]。

表 5-3 度量名称字符串

| 类别 | 字符串 | 说明 |
|---|---|---|
| 时钟分析度量 | `total` | 总线程时间 |
| | `totalcpu` | CPU 总时间 |
| | `user` | 用户 CPU 时间 |
| | `system` | 系统 CPU 时间 |
| | `trap` | 自陷 CPU 时间 |
| | `lock` | 用户锁定时间 |
| | `datapfault` | 数据缺页时间 |

| 类别 | 字符串 | 说明 |
| --- | --- | --- |
| | textpfault | 文本缺页时间 |
| | kernelpfault | 内核缺页时间 |
| | stop | 停止时间 |
| | wait | CPU 等待时间 |
| | sleep | 休眠时间 |
| 硬件计数器度量 | insts | 发出的指令数<br>在所有支持的系统上可用。 |
| | cycles | CPU 周期<br>在大多数受支持的系统上可用。此外，每个处理器还有自己的一组计数器。可使用 `collect -h` 查看系统计数器的完整列表。 |
| | CPI | 每指令周期数, 从 `cycles` 和 `insts` 度量计算得到只有记录了两个计数器时才可用。 |
| | IPC | 每周期指令数, 从 `cycles` 和 `insts` 度量计算得到只有记录了两个计数器时才可用。 |
| OpenMP 分析度量 | ompwork | 以串行或并行方式执行工作所用的时间 |
| | ompwait | OpenMP 运行时等待同步时所用的时间 |
| | masterthread | "Master Thread Time"（主线程时间）是主线程消耗的总时间。只能在 Solaris 实验中查看该时间。该时间与挂钟时间相对应。 |
| 同步延迟度量 | sync | 同步等待时间 |
| | syncn | 同步等待计数 |
| 堆跟踪度量 | heapalloccnt | 分配的数量 |
| | heapallocbytes | 分配的字节数 |
| | heapleakcnt | 泄漏的数量 |
| | heapleakbytes | 泄漏的字节数 |
| I/O 跟踪度量 | ioreadbytes | 读取的字节数 |
| | ioreadcnt | 读取计数 |
| | ioreadtime | 读取时间 |
| | iowritebytes | 写入的字节数 |
| | iowritecnt | 写入计数 |
| | iowritetime | 写入时间 |
| | ioothrcnt | 其他 IO 计数 |
| | iootherttime | 其他 IO 时间 |
| | ioerrornt | IO 错误计数 |
| | ioerrortime | IO 错误时间 |
| 线程分析器度量 | raccesses | 数据争用访问 |
| | deadlocks | 死锁 |
| MPI 跟踪度量 | mpitime | MPI 调用所用的时间 |
| | mpisend | 已启动 MPI 点对点发送数 |

| 类别 | 字符串 | 说明 |
|---|---|---|
| MPI 分析度量 | mpibytessent | "MPI Sends"（MPI 发送次数）中的字节数 |
| | mpireceive | 已完成 MPI 点对点接收数 |
| | mpibytesrecv | "MPI Receives"（MPI 接收次数）中的字节数 |
| | mpiother | 对其他 MPI 函数的调用数 |
| | mpiwork | 在 MPI 运行时内执行工作（如处理请求或消息）所用的时间 |
| | mpiwait | 在 MPI 运行时内等待事件、缓冲区或消息所用的时间 |

除了表 5-3 “度量名称字符串” 中列出的名称字符串外，还有两个只能在缺省度量列表中使用的名称字符串。这两个名称字符串是 `hwc`（它与任何硬件计数器名称匹配）和 `any`（它与任何度量名称字符串匹配）。另请注意，`cycles` 和 `insts` 对于 SPARC® 平台和 x86 平台是通用的，但是还存在特定于体系结构的其他类型的名称字符串。

要查看在装入的实验中可用的度量，请发出 `metric_list` 命令。

# 控制函数列表的命令

以下命令控制显示函数信息的方式。

## functions

使用当前选定的度量写入函数列表。函数列表包括选定进行函数显示的装入对象中的所有函数，以及使用 `object_select` 命令隐藏其函数的任何装入对象。

输出的缺省度量是独占和非独占用户 CPU 时间，以秒和总程序度量百分比表示。可以使用 `metrics` 命令更改当前显示的度量，该命令必须在发出 `functions` 命令之前发出。也可以在 `.er.rc` 文件中使用 `dmetrics` 命令更改缺省值。

可以使用 `limit` 命令限制写入的行数（请参见“控制输出的命令” [147]）。

对于用 Java 编程语言编写的应用程序，显示的函数信息因查看模式的设置而异，查看模式可设置为用户、专家或计算机。

- 用户模式按名称显示每个方法，将已解释的方法和 HotSpot 编译的方法的数据聚集在一起。它还禁止显示非用户 Java 线程的数据。
- 专家模式将 HotSpot 编译的方法与已解释的方法分开。它不禁止显示非用户 Java 线程。

- 计算机模式在进行解释时根据 Java 虚拟机 (Java Virtual Machine, JVM) 软件显示已解释的 Java 方法的数据，并对指定的方法报告使用 Java HotSpot 虚拟机编译的方法的数据。显示所有线程。

在所有三种模式下，对于 Java 目标调用的任何 C、C++ 或 Fortran 代码，都以通用的方法报告数据。

## **metrics** *metric-spec*

指定函数列表度量的选项。字符串 *metric-spec* 可以是恢复缺省度量选项的关键字 `default`，也可以是由冒号分隔的一系列度量关键字。以下示例说明一个度量列表。

```
% metrics i.user:i%user:e.user:e%user
```

此命令指示 `er_print` 实用程序显示以下度量：

- 用秒表示的非独占用户 CPU 时间
- 非独占用户 CPU 时间百分比
- 用秒表示的独占用户 CPU 时间
- 独占用户 CPU 时间百分比

缺省情况下，所用的度量设置基于从 `.er.rc` 文件处理的 `dmetrics` 命令，如“在 .er.rc 文件中设置缺省值” [151]中所述。如果 `metrics` 命令将 *metric-spec* 显式设置为 `default`，则根据所记录的数据恢复缺省设置。

重置度量时，会在新列表中设置缺省排序度量。

如果省略 *metric-spec*，则显示当前的度量设置。

除了为函数列表设置度量外，`metrics` 命令还可以为调用方-被调用方、派生的数据输出和索引对象设置度量。调用方-被调用方度量显示以下归属度量：它们对应于显示非独占度量或独占度量的函数列表中的那些度量；同时还显示静态度量。

数据空间度量显示以下数据空间度量：这些度量有可用的数据并且对应于显示非独占度量或独占度量的函数列表中的那些度量；同时还显示静态度量。

索引对象度量显示以下索引对象度量：这些度量对应于显示非独占度量或独占度量的函数列表中的那些度量；同时还显示静态度量。

处理 `metrics` 命令时，将输出一条消息，指明当前的度量选项。对于前面的示例，消息如下。

```
current: i.user:i%user:e.user:e%user:name
```

有关度量列表的语法的信息，请参见“度量列表” [123]。要查看可用度量的列表，请使用 `metric_list` 命令。

如果 `metrics` 命令出现错误，则将忽略该命令并发出警告，同时仍将使用以前的设置。

## **sort** *metric_spec*

按 *metric-spec* 对函数列表进行排序。度量名称中的 *visibility* 不影响排序顺序。如果在 *metric-spec* 中指定了多个度量，则使用第一个可见度量。如果指定的度量都不可见，则忽略该命令。可以在 *metric-spec* 前加上减号 (-) 以指定反向排序。

缺省情况下，度量排序设置基于从 `.er.rc` 文件处理的 `dsort` 命令，如“在 .er.rc 文件中设置缺省值” [151]中所述。如果 `sort` 命令将 *metric_spec* 显式设置为 `default`，则使用缺省设置。

字符串 *metric-spec* 为“度量列表” [123]中所述的度量关键字之一，如以下示例所示。

```
sort i.user
```

此命令指示 `er_print` 实用程序按非独占用户 CPU 时间对函数列表进行排序。如果度量不在已装入的实验中，则输出一条警告并忽略该命令。完成命令时，将输出排序度量。

## **fsummary**

为函数列表中的每个函数写入摘要面板。可以使用 `limit` 命令限制写入的面板数（请参见“控制输出的命令” [147]）。

摘要度量面板包括函数或装入对象的名称、地址和大小。对于函数，它包括源文件、对象文件和装入对象的名称。该面板显示选定函数或装入对象的所有记录的度量，包括以值和百分比形式表示的独占和非独占度量。

## **fsingle** *function-name* **[***N***]**

为指定的函数写入摘要面板。当有多个函数具有相同的名称时，需要使用可选参数 *N*。为具有给定函数名称的第 *N* 个函数写入摘要度量面板。在命令行上提供命令时，*N* 是必需的；如果不需要它，则将其忽略。当以交互方式提供不带 *N* 的命令但又需要 *N* 时，则会输出具有对应 *N* 值的函数列表。

有关函数的摘要度量的说明，请参见对 `fsummary` 命令的描述。

# 控制调用方-被调用方列表的命令

以下命令控制显示调用方和被调用方信息的方式。

## callers-callees

按函数排序度量 (sort) 指定的顺序，输出每个函数的调用方-被调用方面板。

在每个调用方-被调用方报告中，调用方和被调用方按调用方-被调用方排序度量 (csort) 进行排序。可以使用 limit 命令限制写入的面板数（请参见“控制输出的命令” [147]）。选定的（中央）函数以星号标记，如以下示例中所示。

```
Attr.       Name
User CPU
 sec.
4.440          commandline
0.            *gpf
4.080          gpf_b
0.360          gpf_a
```

在此示例中，gpf 是选定的函数；它由 commandline 调用，而它调用 gpf_a 和 gpf_b。

## csingle *function-name* [*N*]

为指定的函数写入调用方-被调用方面板。当有多个函数具有相同的名称时，需要使用可选参数 *N*。为具有给定函数名称的第 *N* 个函数写入调用方-被调用方面板。在命令行上提供命令时，*N* 是必需的；如果不需要它，则将其忽略。当以交互方式提供不带 *N* 的命令但又需要 *N* 时，则会输出具有对应 *N* 值的函数列表。

## cprepend *function-name [N | ADDR]*

构建调用堆栈时，将指定的函数置于当前调用堆栈片段之前。如果函数名称不明确，可选参数是必需的；有关指定该参数的更多信息，请参见“source|src { *filename* | *function-name* } [ *N*]” [132]。

## cappend *function-name [N | ADDR]*

构建调用堆栈时，将指定函数附加到当前调用堆栈片段。如果函数名称不明确，可选参数是必需的；有关指定该参数的更多信息，请参见“source|src { *filename* | *function-name* } [ *N*]” [132]。

### crmfirst

构建调用堆栈时，从调用堆栈片段删除顶部帧。

### crmlast

构建调用堆栈时，从调用堆栈片段删除底部帧。

## 控制调用树列表的命令

本节介绍用于调用树的命令。

### calltree

显示来自实验的动态调用图，显示每个级别的分层度量。

## 跟踪数据常用的命令

本节介绍可对包含跟踪数据的实验使用的命令。

### datasize

在对数刻度上写入跟踪数据中引用的数据大小的分布。对于堆跟踪，大小是分配或泄漏大小。对于 I/O 跟踪，大小是已传输的字节数。

### duration

在对数刻度上写入跟踪数据中事件持续时间的分布。对于同步跟踪，持续时间是同步延迟。对于 I/O 跟踪，持续时间是 I/O 操作所用的时间。

## 控制泄漏和分配列表的命令

本节介绍与内存分配和取消分配相关的命令。

### leaks

显示由通用调用堆栈聚集的内存泄漏列表。每个条目显示了总泄漏数和给定调用堆栈的总泄漏字节数。该列表按泄漏的字节数进行排序。

### allocs

显示由通用调用堆栈聚集的内存分配列表。每个条目显示了分配数和给定调用堆栈的总分配字节数。该列表按分配的字节数进行排序。

### heap

写入按通用调用堆栈聚集的分配和泄漏的列表。

### heapstat

写入堆使用情况的整体统计信息，包括应用程序的内存使用量峰值。

## 控制 I/O 活动报告的命令

本节介绍与 I/O 活动相关的命令。

### ioactivity

写入所有 I/O 活动的报告，按文件排序。

### iodetail

写入所有 I/O 活动的报告，按虚拟文件描述符排序。每次打开文件，系统都会生成一个不同的虚拟文件描述符，即使从打开的文件返回相同的文件描述符也是如此。

### iocallstack

写入所有 I/O 活动的报告，按调用堆栈排序，并对调用堆栈相同的所有事件进行聚集。对于每个聚集的调用堆栈，包括调用堆栈跟踪。

### iostat

写入所有 I/O 活动的摘要统计信息。

## 控制源代码和反汇编代码列表的命令

以下命令控制显示带注释的源代码和反汇编代码的方式。

### source|src { *filename* | *function-name* } [ *N*]

为指定文件或包含指定函数的文件写出带注释的源代码。在任一情况下该文件都必须位于您所指定的路径中的目录下。如果使用 GNU Fortran 编译器编译了源代码，则函数名称出现在源代码中时，必须在其后添加两个下划线字符。

仅当文件或函数的名称不明确时，才使用可选参数 *N*（正整数）；在这种情况下，使用第 *N* 个可能的选项。如果提供不带数字说明符的不明确名称，则 `er_print` 实用程序将输出可能的对象文件名称的列表。如果提供的名称是函数，则将函数名称附加到对象文件名称，还将输出表示该对象文件的 *N* 值的数字。

也可以将函数名称指定为 *function"file"*，其中 *file* 用于指定函数的替代源上下文。紧邻第一个指令之后，将添加函数的索引行。索引行显示为尖括号内的文本，其格式如下：

```
<Function: f_name>
```

任何函数的缺省源上下文都被定义为该函数的第一条指令所归属的源文件。它通常是经过编译而生成包含该函数的对象模块的源文件。替代源上下文由包含归属于该函数的指令的其他文件组成。此类上下文包括来自头文件的指令和来自内联到指定函数中的函数的指令。如果存在任何替代源上下文，则在缺省源上下文的开头包括扩展索引行的列表以指示替代源上下文所在的位置，格式如下：

```
<Function: f, instructions from source file src.h>
```

---

注 - 如果在命令行上调用 `er_print` 实用程序时使用了 `-source` 参数，则必须在 file 引号前加上反斜杠转义符。换句话说，函数名称的格式为 `function\"file\"`。当 `er_print` 实用程序处于交互模式时，反斜杠不是必需的，也不应使用它。

---

通常，在使用缺省源上下文时，会显示该文件中所有函数的度量。如果显式引用该文件，则仅显示指定函数的度量。

## **disasm|dis {** *filename* **|** *function-name* **} [** *N***]**

为指定文件或包含指定函数的文件写出带注释的反汇编代码。该文件必须位于您所指定的路径中的目录下。

可选参数 *N* 与 `source` 命令的可选参数的使用方法相同。

## **scc** *com-spec*

指定在带注释的源代码列表中显示的编译器注释的类。类列表是类的冒号分隔列表，包含零个或多个以下消息类。

**表 5-4** 编译器注释消息类

| 类 | 含义 |
|---|---|
| `b[asic]` | 显示基本级别的消息。 |
| `v[ersion]` | 显示版本消息，包括源文件名称和上次修改日期、编译器组件的版本、编译日期和选项。 |
| `pa[rallel]` | 显示有关并行化的消息。 |
| `q[uery]` | 显示有关影响代码优化的代码问题。 |
| `l[oop]` | 显示有关循环优化和转换的消息。 |
| `pi[pe]` | 显示有关循环的流水线作业的消息。 |
| `i[nline]` | 显示有关函数内联的消息。 |
| `m[emops]` | 显示有关内存操作（如装入、存储和预取）的消息。 |
| `f[e]` | 显示前端消息。 |
| `co[degen]` | 显示代码生成器消息。 |
| `cf` | 在源代码底部显示编译器标志。 |
| `all` | 显示所有消息。 |
| `none` | 不显示任何消息。 |

类 `all` 和 `none` 不能与其他类一起使用。

如果未提供 `scc` 命令，则显示的缺省类为 `basic`。如果提供了 `scc` 命令，但 *class-list* 为空，则关闭编译器注释。`scc` 命令通常仅在 `.er.rc` 文件中使用。

### **sthresh** *value*

指定带注释的源代码中突出显示度量的阈值百分比。对于文件中的任何源代码行，如果任何度量的值等于或大于该度量最大值的 *value* %，则在该度量所在行的开头插入 *##*。

### **dcc** *com-spec*

指定在带注释的反汇编代码列表中显示的编译器注释的类。类列表是类的冒号分隔列表。可用类的列表与表 5-4 “编译器注释消息类” 中所示的带注释的源代码列表的类列表相同。可将以下表中的选项添加到类列表中。

表 5-5 dcc 命令的附加选项

| 选项 | 含义 |
| --- | --- |
| h[ex] | 显示指令的十六进制值。 |
| noh[ex] | 不显示指令的十六进制值。 |
| s[rc] | 在带注释的反汇编代码列表中交错显示源代码列表。 |
| nos[rc] | 不在带注释的反汇编代码列表中交错显示源代码列表。 |
| as[rc] | 在带注释的反汇编代码列表中交错显示带注释的源代码。 |

### **dthresh** *value*

指定带注释的反汇编代码中突出显示度量的阈值百分比。对于文件中的任何指令行，如果任何度量的值等于或大于该度量最大值的 *value* %，则在该度量所在行的开头插入 *##*。

### **cc** *com-spec*

指定在带注释的源代码和反汇编代码列表中显示的编译器注释的类。类列表是类的冒号分隔列表。可用类的列表与表 5-4 “编译器注释消息类” 中所示的带注释的源代码列表的类列表相同。

## 控制 PC 和行的命令

以下命令控制程序计数器和行信息的显示方式。

### pcs

写入程序计数器 (program counter, PC) 及其度量的列表（按当前排序度量排序）。该列表包括为使用 `object_select` 命令隐藏其函数的每个装入对象显示聚集度量的行。

### psummary

按当前排序度量指定的顺序，为 PC 列表中的每个 PC 写入摘要度量面板。

### lines

写入源代码行及其度量的列表（按当前排序度量排序）。该列表包括为没有行号信息或其源文件未知的每个函数显示聚集度量的行，以及为使用 `object_select` 命令隐藏其函数的每个装入对象显示聚集度量的行。

### lsummary

按当前排序度量指定的顺序，为行列表中的每个行写入摘要度量面板。

## 控制源文件搜索的命令

`er_print` 实用程序查找实验中引用的源文件和装入对象文件。可以使用此部分中所述的指令来帮助 `er_print` 查找实验中引用的文件。

有关用于查找实验源代码的过程的说明，包括如何使用这些指令，请参见“工具如何查找源代码” [189]。

### setpath *path-list*

设置用于查找源文件和对象文件的路径。*path-list* 是目录、jar 文件或 zip 文件的冒号分隔列表。如果任何目录中包含冒号字符，请用反斜杠将它转义。特殊目录名称 `$expts` 按照装入实验的顺序引用当前实验集。可以将其缩写为单个 `$` 字符。

缺省路径为：`$expts:.`.，这是已装入实验的目录和当前工作目录。

使用不带参数的 `setpath` 可以显示当前路径。

`setpath` 命令不能在 `.er.rc` 文件中使用。

### **addpath** *path-list*

将 *path-list* 附加到当前 `setpath` 设置。

`addpath` 命令可在 `.er.rc` 文件中使用，并且将会串联。

### **pathmap** *old-prefix new-prefix*

如果使用由 `addpath` 或 `setpath` 设置的 *path-list* 找不到文件，则可以使用 `pathmap` 命令指定一个或多个路径重映射。在以 *old-prefix* 所指定的前缀开头的源文件、对象文件或共享对象的所有路径名中，旧的前缀将由 *new-prefix* 所指定的前缀替代。然后，使用所得到的路径查找文件。可采用多个 `pathmap` 命令，并逐一尝试，直到找到文件为止。

## 控制数据空间列表的命令

对于 Solaris x86 或 SPARC 系统上的精确计数器，数据空间命令仅适用于记录内存空间/数据空间数据的硬件计数器实验（缺省或显式）。有关更多信息，请参见 `collect(1)` 手册页。

数据空间数据仅可用于使用 `-xhwcprof` 标志编译的函数中发生的分析命中。`-xhwcprof` 标志适用于使用 C、C++ 和 Fortran 编译器进行编译，仅在 SPARC 平台上有意义；在其他平台上将被忽略。

有关这些类型数据的更多信息，请参见“硬件计数器分析数据” [23]。有关用于执行硬件计数器溢出分析的命令的信息，请参见“使用 `collect -h` 收集硬件计数器分析数据” [55]。

有关 `-xhwcprof` 编译器选项的信息，请参见《Oracle Solaris Studio 12.4：Fortran 用户指南》、《Oracle Solaris Studio 12.4：C 用户指南》或《Oracle Solaris Studio 12.4：C++ 用户指南》。

### **data_objects**

写入数据对象及其度量的列表。

### **data_single** *name* **[***N***]**

写入指定数据对象的摘要度量面板。在对象名称不明确的情况下，需要使用可选参数 N。当指令在命令行上时，N 是必需的；如果不需要它，则将其忽略。

### **data_layout**

为具有数据派生度量数据的所有程序数据对象写入带注释的数据对象布局，按整个结构的当前数据排序度量值排序。显示每个聚集的数据对象及归属于该对象的总度量，后跟按偏移顺序显示的所有元素，每个元素都有自己的度量和相对于 32 字节块的大小和位置指示符，其中：

| | |
|---|---|
| < | 元素与块完全契合。 |
| / | 元素启动一个块。 |
| \| | 元素位于块内部。 |
| \ | 元素表示块结束。 |
| # | 元素大小需要多个块。 |
| X | 元素跨多个块，但也可以放入一个块中。 |
| ? | 未定义。 |

## 控制索引对象列表的命令

索引对象命令适用于所有实验。索引对象列表是可以从所记录的数据计算其索引的对象的列表。可以为 "Threads"（线程）、"CPUs" (CPU)、"Samples"（抽样）和 "Seconds"（秒）等预定义索引对象。您可以使用 `indxobj_list` 命令获取完整列表。可以使用 `indxobj_define` 命令定义其他索引对象。

以下命令控制索引对象列表。

### **indxobj** *indxobj-type*

写入与给定类型匹配的索引对象及其度量的列表。索引对象的度量和排序方式与函数列表相同，只不过仅包含独占度量。也可以将名称 *indxobj-type* 直接用作命令。

### **indxobj_list**

写入已知类型的索引对象的列表，用法与 `indxobj` 命令中的 *indxobj-type* 相同。可以为 "Threads"（线程）、"CPUs" (CPU)、"Samples"（抽样）和 "Seconds"（秒）等预定义索引对象。

### **indxobj_define** *indxobj-type index-exp*

通过将数据包映射到由 *index-exp* 提供的对象，定义新的索引对象类型。表达式的语法在“表达式语法” [153]中介绍。

*indxobj-type* 必须尚未定义。其名称不区分大小写，必须完全由字母数字字符或 "_" 字符组成，且以字母字符开头。

*index-exp* 必须在语法上是正确的，否则将返回错误并忽略定义。如果 *index-exp* 包含任何空格，则必须用双引号 (") 将其引起来。

`<Unknown>` 索引对象的索引是 -1，而且用于定义新索引对象的表达式应该支持识别 `<Unknown>`。

例如，对于基于虚拟或物理 PC 的索引对象，表达式应该采用以下格式：

```
VIRTPC>0?VIRTPC:-1
```

## 控制内存对象列表的命令

对于 Solaris x86 或 SPARC 系统上的精确计数器，内存对象命令仅适用于记录内存空间数据的硬件计数器实验（缺省或显式）。有关更多信息，请参见 `collect`(1) 手册页。

内存对象是内存子系统中的组件，例如高速缓存行、页面、内存区等。对象根据从记录的虚拟和/或物理地址计算的索引确定。为虚拟页面和物理页面预定义了内存对象，其大小为 8KB、64KB、512KB 和 4 MB。您可以使用 `memobj_define` 命令定义其他索引对象。

### **memobj** *mobj-type*

使用当前度量写入给定类型的内存对象的列表。所用度量和排序方式与数据空间列表相同。还可以将名称 `mobj_type` 直接用作命令。

## **mobj_list**

写入已知类型的内存对象的列表，用法与 `memobj` 命令中的 *mobj-type* 相同。

## **mobj_define** *mobj-type index-exp*

通过将 VA/PA 映射到由 *index-exp* 指定的对象，定义新的内存对象类型。表达式的语法在“表达式语法” [153]中介绍。

*mobj-type* 必须尚未定义，并且不能与任何现有命令或任何索引对象类型（请参见下文）匹配。其名称必须完全由字母数字字符或 "_" 字符组成，且以字母字符开头。

*index-exp* 必须在语法上是正确的。如果它在语法上不正确，则将返回错误并忽略定义。

`<Unknown>` 内存对象的索引是 -1，而且用于定义新内存对象的表达式应该支持识别 `<Unknown>`。例如，对于基于 VADDR 的对象，表达式应该采用以下格式：

`VADDR>255?`*expression*`:-1`

对于基于 PADDR 的对象，表达式应该采用以下格式：

`PADDR>0?`*expression*`:-1`

## **memobj_drop** *mobj_type*

丢弃给定类型的内存对象。

## **machinemodel** *model_name*

按照指定计算机模型中的定义创建内存对象。*model_name* 是用户当前目录或用户主目录中的一个文件名，或者是在发行版中定义的计算机模型的名称。存储的计算机模型文件带有后缀 `.ermm`。如果 `machinemodel` 命令中的 *model_name* 不是以该后缀结尾，则将使用附加了 `.ermm` 的 *model_name*。如果 *model_name* 以 `/` 开头，则假定它是绝对路径，并且将仅尝试该路径（如果需要，还会附加 `.ermm`）。如果 *model_name* 包含 `/`，将仅尝试相对于当前目录或用户主目录的路径名。

计算机模型文件可以包含注释和 `mobj_define` 命令。将忽略任何其他命令。`machinemodel` 命令可以出现在 `.er.rc` 文件中。如果已装入了计算机模型（通过命令或通过读取记录了计算机模型的实验），则后续 `machinemodel` 命令将删除来自以前计算机模型的所有定义。

如果缺少 *model_name*，则输出所有已知计算机模型的列表。如果 *model_name* 是零长度字符串，则卸载装入的任何计算机模型。

# 用于 OpenMP 索引对象的命令

使用以下命令输出有关 OpenMP 索引对象的信息。

### `OMP_preg`

输出在实验中执行的 OpenMP 并行区域及其度量的列表。此命令仅可用于含有 OpenMP 3.0 或 3.1 性能数据的实验。

### `OMP_task`

输出在实验中执行的 OpenMP 任务及其度量的列表。此命令仅可用于含有 OpenMP 3.0 或 3.1 性能数据的实验。

# 支持线程分析器的命令

以下命令支持线程分析器。有关捕获和显示的数据的更多信息，请参见《Oracle Solaris Studio 12.4：线程分析器用户指南》。

### `races`

将所有数据争用的列表写入实验中。数据争用报告只能从具有数据争用检测数据的实验获得。

### `rdetail` *race-id*

写入给定 *race-id* 的详细信息。如果 *race-id* 设置为 `all`，显示所有数据争用的详细信息。数据争用报告只能从具有数据争用检测数据的实验获得。

### deadlocks

写入实验中检测到的所有实际死锁和潜在死锁的列表。死锁报告只能从具有死锁检测数据的实验获得。

### ddetail *deadlock-id*

写入给定 *deadlock-id* 的详细信息。如果 *deadlock-id* 设置为 `all`，则显示所有死锁的详细信息。死锁报告只能从具有死锁检测数据的实验获得。

## 列出实验、抽样、线程和 LWP 的命令

本节介绍列出实验、抽样、线程和 LWP 的命令。

### experiment_list

显示装入的实验及其 ID 号的完整列表。列出的每个实验都有一个索引（在选择抽样、线程或 LWP 时使用）和一个 PID（可在高级过滤时使用）。

以下示例显示一个实验列表。

```
(er_print) experiment_list
 ID Sel   PID Experiment
=== === ===== ================
  1 yes 13493 test.1.er
  2 yes 24994 test.2.er
  3 yes 25653 test.2.er/_f8.er
  4 yes 25021 test.2.er/_x5.er
```

### sample_list

显示当前选定的要进行分析的抽样的列表。

以下示例显示一个抽样列表。

```
(er_print) sample_list
```

```
Exp Sel     Total
=== ======= =====
  1 1-6        31
  2 7-10,15    31
```

### lwp_list

显示当前选定的要进行分析的 LWP 的列表。

### thread_list

显示当前选定的要进行分析的线程的列表。

### cpu_list

显示当前选定的要进行分析的 CPU 的列表。

## 控制实验数据过滤的命令

可以指定按以下两种方式过滤实验数据：

- 指定过滤器表达式，对每个数据记录计算该表达式以确定是否应该包括该记录
- 选择要进行过滤的实验、抽样、线程、CPU 和 LWP

## 指定过滤器表达式

可以使用 `filters` 命令指定过滤器表达式。

### filters *filter-exp*

*filter-exp* 是一个表达式，对于应该包括的任何数据记录，其计算结果为真；对于不应包括的记录，其计算结果为假。表达式的语法在“表达式语法” [153]中介绍。

# 列出过滤器表达式的关键字

可以查看可在实验的过滤器表达式中使用的操作数或关键字的列表。

### describe

打印可用于构建过滤器表达式的关键字列表。“表达式语法” [153]中介绍了过滤器表达式的一些关键字和语法。

# 选择要进行过滤的抽样、线程、LWP 和 CPU

选择语法如以下示例所示。该语法用于命令描述。

[*experiment-list*:]*selection-list*[+[
*experiment-list*:]*selection-list* … ]

## 选择列表

可以在每个选择列表前面加上实验列表，用冒号与其隔开且不加空格。要进行多个选择，请用 + 符号连接多个选择列表。

实验列表和选择列表具有相同的语法，可以使用关键字 `all`，也可以使用编号或编号范围 (*n*-*m*) 的列表，其中用逗号分隔且不加空格，如以下示例所示。

```
2,4,9-11,23-32,38,40
```

可以使用 `experiment_list` 命令确定实验编号。

一些选择示例：

```
1:1-4+2:5,6
all:1,3-6
```

在第一个示例中，从实验 1 选择了对象 1 到 4，从实验 2 选择了对象 5 和 6。在第二个示例中，从所有实验选择了对象 1 以及 3 到 6。对象可以是 LWP、线程或抽样。

## 选择命令

用于选择 LWP、抽样、CPU 和线程的命令不是独立的。如果某个命令的实验列表与前一个命令的实验列表不同，则按以下方式将来自最后一个命令的实验列表应用于所有三个选择目标 – LWP、抽样和线程。

- 关闭不在最后一个实验列表中的实验的现有选择。
- 保留最后一个实验列表中的实验的现有选择。
- 对于没有做出任何选择的目标，将选择设置为 `all`。

**`sample_select`** *sample-spec*

选择要显示其信息的抽样。命令完成时，会显示所选抽样的列表。

**`lwp_select`** *lwp-spec*

选择要显示其信息的 LWP。命令完成时，会显示所选 LWP 的列表。

**`thread_select`** *thread-spec*

选择要显示其信息的线程。命令完成时，会显示所选线程的列表。

**`cpu_select`** *cpu-spec*

选择要显示其信息的 CPU。命令完成时，会显示所选 CPU 的列表。

# 控制装入对象展开和折叠的命令

这些命令确定 `er_print` 实用程序显示装入对象的方式。

## `object_list`

显示一个由两列组成的列表，包含全部装入对象的状态和名称。在第一列中显示每个装入对象的显示/隐藏/API 状态，在第二列中显示对象的名称。每个装入对象名称前有 `show`（指示在函数列表中显示该对象的函数（已展开））、`hide`（指示在函数列表中不显示该对象的函数（已折叠））或 `API-only`（如果仅显示那些代表装入对象入口点的函数）。已折叠装入对象的所有函数都映射到函数列表中表示整个装入对象的单个条目。

以下是装入对象列表的示例。

```
(er_print) object_list
Sel  Load Object
==== ==================
hide <Unknown>
show <Freeway>
show <libCstd_isa.so.1>
show <libnsl.so.1>
show <libmp.so.2>
show <libc.so.1>
show <libICE.so.6>
show <libSM.so.6>
show <libm.so.1>
show <libCstd.so.1>
show <libX11.so.4>
show <libXext.so.0>
show <libCrun.so.1>
show <libXt.so.4>
show <libXm.so.4>
show <libsocket.so.1>
show <libgen.so.1>
show <libcollector.so>
show <libc_psr.so.1>
show <ld.so.1>
show <liblayout.so.1>
```

## **object_show** *object1,object2,...*

设置所有已命名的装入对象以显示其所有函数。对象的名称可以为全路径名或根基名称。如果名称包含逗号字符，则必须用双引号将名称引起来。如果使用字符串 "all" 命名装入对象，则显示所有装入对象的函数。

## **object_hide** *object1,object2,...*

设置所有已命名的装入对象以隐藏其所有函数。对象的名称可以为全路径名或根基名称。如果名称包含逗号字符，则必须用双引号将名称引起来。如果使用字符串 "all" 命名装入对象，则显示所有装入对象的函数。

## **object_api** *object1,object2,...*

设置所有已命名的装入对象以仅显示所有代表库入口点的函数。对象的名称可以为全路径名或根基名称。如果名称包含逗号字符，则必须用双引号将名称引起来。如果使用字符串 "all" 命名装入对象，则显示所有装入对象的函数。

### objects_default

按照 .er.rc 文件处理中的初始缺省值设置所有装入对象。

### object_select *object1,object2,...*

选择要显示有关其中函数信息的装入对象。显示所有指定的装入对象的函数，隐藏所有其他装入对象的函数。*object-list* 是装入对象的列表，用逗号分隔且不加空格。如果显示某个装入对象的函数，则在函数列表中显示具有非零度量的所有函数。如果隐藏某个装入对象的函数，则会折叠其函数，并且仅显示包含整个装入对象的度量的单个行，而不是显示其各个函数。

装入对象的名称应该为全路径名或根基名称。如果对象名称本身包含逗号，则必须用双引号将名称引起来。

## 列出度量的命令

以下命令列出当前选定的度量以及所有可用的度量关键字。

### metric_list

显示函数列表中当前选定的度量和可以在其他命令（例如 metrics 和 sort）中使用的度量关键字列表，以在函数列表中引用各种类型的度量。

### cmetric_list

显示当前选择的调用方-被调用方归属的度量以及当前用于排序的度量。

### data_metric_list

显示当前选定的数据派生度量以及所有数据派生报告的度量和关键字名称的列表。列表的显示方式与 metric_list 命令的输出方式相同，但是仅包括那些具有数据派生类型的度量和静态度量。

### indx_metric_list

显示当前选定的索引对象度量以及所有索引对象报告的度量和关键字名称的列表。以与 metric_list 命令相同的方式显示列表，但是仅包括那些具有独占类型的度量和静态度量。

## 控制输出的命令

以下命令控制 er_print 显示输出。

### outfile {*filename*|-|--}

关闭任何打开的输出文件，然后为后续输出打开 *filename*。打开 *filename* 时，将清除任何先前存在的内容。如果指定一个短划线 (-) 而不是 *filename*，则将输出写入标准输出。如果指定两个短划线 (--) 而不是 *filename*，则将输出写入标准错误。

### appendfile *filename*

关闭任何打开的输出文件并打开 *filename*，保留任何先前存在的内容，以便将后续输出附加到文件的结尾。如果 *filename* 不存在，则 appendfile 命令的功能与 outfile 命令的功能相同。

### limit *n*

将任何输出限制为报告中的前 *n* 个条目，其中 *n* 是一个无符号整数。如果 *n* 为零，则删除所有限制。如果省略了 *n*，则会输出当前限制。

### name { long | short } [ :{ *shared-object-name* | *no-shared-object-name* } ]

指定是使用长形式还是短形式的函数名称（仅限 C++ 和 Java）。如果指定了 *shared-object-name*，则将共享对象名称附加到函数名称。

## viewmode { user| expert | machine }

将模式设置为以下模式之一：

`user`
对于 Java 实验，显示 Java 线程的 Java 调用堆栈，而不显示内务处理线程。函数列表包括函数 `<JVM-System>`，该函数表示来自非 Java 线程的聚集时间。当 JVM 软件不报告 Java 调用堆栈时，将根据函数 `<no Java callstack recorded>` 报告时间。

对于 OpenMP 实验，显示重构的调用堆栈，这些重构的调用堆栈类似于在不使用 OpenMP 的情况下编译程序时获取的调用堆栈。在 OpenMP 运行时执行某些操作时，添加名称格式为 `<OMP-*>` 的特殊函数。

`expert`
对于 Java 实验，在执行用户的 Java 代码时，将显示 Java 线程的 Java 调用堆栈，而在执行 JVM 代码或当 JVM 软件不报告 Java 调用堆栈时，则显示计算机调用堆栈。显示内务处理线程的计算机调用堆栈。

对于 OpenMP 实验，显示编译器生成的、代表并行化循环、任务等的函数，这些函数会与用户模式中的用户函数聚集。在 OpenMP 运行时执行某些操作时，添加名称格式为 `<OMP-*>` 的特殊函数。禁止 OpenMP 运行时代码 `libmtsk.so` 中的函数。

`machine`
对于 Java 实验和 OpenMP 实验，显示所有线程的实际本机调用堆栈。

对于除 Java 实验和 OpenMP 实验之外的所有实验，所有三种模式都显示相同的数据。

## compare { on | off | delta | ratio }

将比较模式设置为关闭（`compare off`，缺省值）、开启 (`compare on`)、增量 (`compare delta`) 或比率 (`compare ratio`)。如果比较模式为关闭，则读取多个实验时将聚集数据。如果启用比较，则装入多个实验时，将为每个实验的数据显示单独的度量列。如果比较模式为增量，则基实验显示绝对度量，但是比较实验显示其与基实验之间的差异。如果比较模式为比率，则比较实验显示其与基实验之间的比率。

比较模式将每个实验或实验组视为单独的比较组。第一个实验或实验组参数是基本组。如果您希望在一个比较组中包括多个实验，则必须创建实验组文件以用作 `er_print` 的单个参数。

### **printmode** *string*

基于 *string* 设置输出模式。如果 *string* 为 `text`，输出为表格形式。如果 *string* 是一个字符，输出将作为分隔符分隔的列表执行，这一个字符作为分隔符。如果 *string* 为 `html`，输出将采用 HTML 表的格式。其他任何 *string* 均无效，将忽略该命令。

`printmode` 设置只能用于生成表的命令，如 `functions`、`memobj`、`indxobj`。对于其他输出命令，如 `source` 和 `disassembly`，将忽略该设置。

## 输出其他信息的命令

以下 `er_print` 子命令显示关于该实验的其他信息。

### **header** *exp-id*

显示有关指定实验的描述性信息。*exp-id* 可以通过 `exp_list` 命令获得。如果 *exp-id* 为 `all` 或未提供，则显示所有已装入实验的信息。

在每个标头后，将输出任何错误或警告。每个实验的标头由一行短划线分隔。

如果实验目录包含名为 `notes` 的文件，则将该文件的内容添加到标头信息之前。可以通过 `collect` 命令的 `-C` "*comment*" 参数，手动添加、编辑或指定 notes 文件。

*exp-id* 在命令行上是必需的，但是在脚本中或交互模式下不是必需的。

### **ifreq**

从度量的计数数据写入指令频率列表。指令频率报告只能从计数数据生成。此命令仅适用于 Oracle Solaris。

### **objects**

列出装入对象以及由于使用装入对象而产生的任何错误或警告消息以用于性能分析。可以使用 `limit` 命令限制列出的装入对象数（请参见“控制输出的命令” [147]）。

### **overview** *exp_id*

写入对所有实验汇总的所有数据的概述。使用 [X] 指示函数列表度量，而热点度量具有星号来突出显示其值。

### **sample_detail** [ *exp_id* ]

写入有关指定实验的详细样例信息。*exp_id* 是 `experiment_list` 命令指定的实验的数字标识符。如果 *exp_id* 被省略或为 `all`，则写入所有实验中所有样例的总和与统计信息。

在以前的发行版中使用 `overview` 命令输出的报告现在由 `sample_detail` 生成。

### **statistics** *exp_id*

写出在指定实验的当前抽样集上聚集的执行统计信息。有关显示的执行统计信息的定义和含义的信息，请参见 `getrusage`(3C) 和 `proc`(4) 手册页。执行统计信息包括来自收集器未收集其任何数据的系统线程的统计信息。

*exp_id* 可以通过 `experiment_list` 命令获得。如果未提供 *exp_id*，则显示在每个实验的抽样集上聚集的所有实验的数据总和。如果 *exp_id* 为 `all`，则显示每个实验的总和统计和单独统计。

## 用于实验的命令

以下命令仅在脚本中和交互模式下使用，而不允许在命令行上使用。

### **add_exp** *exp_name*

将指定的实验或实验组添加到当前会话。

### **drop_exp** *exp_name*

从当前会话中丢弃指定的实验。

### **open_exp** *exp_name*

从会话中丢弃装入的所有实验，然后装入指定的实验或实验组。

## 在 .er.rc 文件中设置缺省值

er_print 实用程序从多个位置中读取名为 .er.rc 的资源文件中的设置，以确定缺省值。按以下顺序读取文件：

1. 系统资源文件 /*Studio-installation-dir*/lib/analyzer/lib/er.rc
2. 用户的 .er.rc 资源文件（如果该文件存在于用户的起始目录中）。
3. .er.rc 资源文件（如果该文件存在于当前执行 er_print 命令的目录中）。

每个文件的设置将覆盖在它之前读取的文件的设置。您的起始目录中的 .er.rc 文件的缺省值覆盖系统缺省值，当前目录中的 .er.rc 文件的缺省值覆盖起始目录缺省值和系统缺省值。

er_src 实用程序也使用 .er.rc 中应用至源文件和反汇编编译器注释的所有设置。

可以在 .er.rc 文件中使用以下命令为 er_print 和 er_src 设置缺省值。只能使用这些命令设置缺省值，不能将其用作 er_print 实用程序的输入。

可将 .er.rc 缺省值文件包含在起始目录中以为所有实验设置缺省值，或者将其包含在任何其他目录中以在本地设置缺省值。启动 er_print 实用程序、er_src 实用程序或性能分析器时，将扫描当前目录和您的起始目录以查找 .er.rc 文件。如果存在这些文件，将读取它们，同时还将读取系统缺省值文件。您的起始目录中的 .er.rc 文件的缺省值覆盖系统缺省值，当前目录中的 .er.rc 文件的缺省值覆盖起始目录缺省值和系统缺省值。

---

注 - 要确保从存储实验的目录读取缺省值文件，必须从该目录启动 er_print 实用程序。

---

这类文件可包含
scc、sthresh、dcc、dthresh、addpath、pathmap、name、mobj_define、indxobj_define、object_show、obj
和 viewmode 命令，如本章前面所述。它们也可包含以下命令，这些命令不能用在命令行上或脚本中：

### **dmetrics** *metric-spec*

指定要在函数列表中显示或输出的缺省度量。度量列表的语法和用法在“度量列表” [123]一节中介绍。列表中度量关键字的顺序决定了显示度量的顺序。

通过在列表中每个度量名称第一次出现的地方之前添加对应的归属度量，可从函数列表缺省度量派生调用方-被调用方列表的缺省度量。

### **dsort** *metric-spec*

指定函数列表将按其排序的缺省度量。排序度量是该列表中与任何已装入实验中的度量匹配的第一个度量，且受到以下条件的限制：

- 如果 *metric-spec* 中的条目具有叹号 ! 可见性字符串，则使用其名称匹配的第一个度量，而不管它是否可见。
- 如果 *metric-spec* 中的条目具有任何其他可见性字符串，则使用其名称匹配的第一个可见度量。

度量列表的语法和用法在“度量列表” [123]一节中介绍。

调用方-被调用方列表的缺省排序度量是对应于函数列表的缺省排序度量的归属度量。

### **en_desc { on | off |** *=regexp***}**

将读取子孙实验的模式设置为 `on`（启用所有子孙实验）或 `off`（禁用所有子孙实验）。如果使用 *=regexp*，则启用可执行文件名称与正则表达式匹配的那些实验的数据。缺省设置是 `on`，即跟踪所有子孙进程。

读取包含子孙的实验时，性能分析器和 `er_print` 将忽略包含少量性能数据或不包含性能数据的任何子实验。

## 其他命令

还可在 `er_print` 实用程序中使用以下命令执行其他任务。

### **procstats**

输出处理数据的累计统计信息。

### **script** *filename*

处理脚本文件 *filename* 中的附加命令。

### version

输出 er_print 实用程序的当前版本号。

### quit

终止当前脚本的处理，或者退出交互模式。

### exit

quit 的别名。

### help

输出 er_print 命令的列表。

### # ...

注释行；在脚本或 .er.rc 文件中使用。

## 表达式语法

定义过滤器的表达式和用于计算内存对象索引的表达式使用通用语法。

语法将表达式指定为运算符和操作数或关键字的组合。对于过滤器，如果表达式的计算结果为真，则包括数据包；如果表达式的计算结果为假，则排除数据包。对于内存对象或索引对象，表达式的计算结果为一个索引，该索引定义数据包中引用的特定内存对象或索引对象。

表达式中的操作数可以是标签、常量或者是数据记录中的字段，如使用 describe 命令的执行结果所示。这些操作数包括 THRID、LWPID、CPUID、USTACK、XSTACK、MSTACK、LEAF、VIRTPC、PHYSPC、VADDR、PADDR、DOBJ、TSTAMP、SA 或内存对象的名称。操作数名称不区分大小写。

USTACK、XSTACK 和 MSTACK 分别表示用户视图、专家视图和计算机视图中的函数调用堆栈。

仅当为硬件计数器分析或时钟分析指定 "+" 时，VIRTPC、PHYSPC、VADDR 和 PADDR 都不为零。此外，当无法确定实际虚拟地址时，VADDR 小于 256。如果无法确定 VADDR 或者无法将虚拟地址映射到物理地址，PADDR 为零。同样，如果回溯失败或者未请求回溯，则 VIRTPC 为零；如果 VIRTPC 为零或者无法将 VIRTPC 映射到物理地址，则 PHYSPC 为零。

运算符包括常见的逻辑运算符和算术（包括移位）运算符（在 C 表示法中，具有 C 优先级规则），以及用于确定某个元素是否位于集中的运算符 (IN) 或用于确定某组元素中的任一元素或全部元素是否包含在集中的运算符（分别对应 SOME IN 或 IN）。附加运算符 ORDERED IN 确定左侧操作数中的所有元素是否以相同的顺序出现在右侧操作数中。注意，IN 运算符要求左侧操作数中的所有元素出现在右侧操作数中，但对顺序未作要求。

按照 C 语言中的方式指定 If-then-else 结构（使用 ? 和 : 运算符）。使用圆括号以确保正确解析所有表达式。在 er_print 命令行上，不能将表达式拆分为多行。在脚本中或命令行上，如果表达式包含空格，则必须用双引号将其引起来。

过滤器表达式的计算结果为布尔值。如果应该包括数据包，则计算结果为 True，如果不应该包括数据包，则计算结果为 False。线程、CPU、实验 id、进程 pid 和抽样过滤基于相应关键字和整数之间的关系表达式，或者使用 IN 运算符和逗号分隔的整数列表。

可通过在 TSTAMP 和时间（以整数纳秒为单位，从将要处理其数据包的实验开始时算起）之间指定一个或多个关系表达式来使用时间过滤。抽样的时间可以使用 overview 命令获得。overview 命令中的时间以秒为单位，必须转换为纳秒以用于时间过滤。时间也可以从性能分析器中的 "Timeline"（时间线）显示中获得。

函数过滤可以基于叶函数或堆栈中的任何函数。按叶函数进行过滤是通过 LEAF 关键字和整型函数 ID 之间的关系表达式指定的，或者使用 IN 运算符和结构 FNAME("*regexp*") 指定，其中 *regexp* 是正则表达式，如 regexp(5) 手册页中所规定的那样。函数的整个名称（如 *name* 的当前设置所指定）必须匹配。

可以通过使用表达式 (FNAME("myfunc") SOME IN USTACK) 确定结构 FNAME("*regexp*") 中的任何函数是否位于由关键字 USTACK 表示的函数数组中，来指定基于调用堆栈中的任何函数的过滤。FNAME 也可用于按照相同的方式过滤堆栈的计算机视图 (MSTACK) 和专家视图 (XSTACK)。

数据对象过滤类似于堆栈函数过滤，使用 DOBJ 关键字和结构 DNAME("*regexp*")（括在圆括号中）。

内存对象过滤是使用内存对象的名称（如 mobj_list 命令中所示）和对象的整型索引或对象集的索引来指定的。（<Unknown> 内存对象的索引是 -1。）

索引对象过滤是使用索引对象的名称（如 indxobj_list 命令中所示）和对象的整型索引或对象集的索引来指定的。（<Unknown> 索引对象的索引是 -1。）

数据对象过滤和内存对象过滤仅对具有数据空间数据的硬件计数器数据包有意义；此类过滤将排除所有其他数据包。

虚拟地址或物理地址的直接过滤是通过 VADDR 或 PADDR 与地址之间的关系表达式指定的。

内存对象定义（请参见“mobj_define *mobj-type index-exp*” [139]）使用其计算结果为整型索引的表达式（使用 VADDR 关键字或 PADDR 关键字）。该定义仅适用于内存计数器和数据空间数据的硬件计数器数据包。该表达式应该返回整数，对于 <Unknown> 内存对象，则返回 -1。

索引对象定义（请参见“indxobj_define *indxobj-type index-exp*” [138]）使用其计算结果为整型索引的表达式。该表达式应该返回整数，对于 <Unknown> 索引对象，则返回 -1。

# 示例过滤器表达式

本节显示可与 er_print -filters 命令一起使用以及可在 "Advanced Custom Filter"（高级定制过滤器）对话框中使用的过滤器表达式的示例。

对于 er_print -filters 命令，过滤器表达式用单引号括起来，类似以下示例：

```
er_print -filters 'FNAME("myfunc") SOME IN USTACK' -functions test.1.er
```

例 5-1 按名称和堆栈过滤函数

从用户函数堆栈过滤名为 myfunc 的函数：

```
FNAME("myfunc") SOME IN USTACK
```

例 5-2 按线程和 CPU 过滤事件

当线程 1 仅在 CPU 2 上运行时，要查看线程 1 中的事件，请使用：

```
THRID == 1 && CPUID == 2
```

例 5-3 按索引对象过滤事件

如果将索引对象 THRCPU 定义为 "CPUID<<16|THRID "，下面的过滤器与上述当线程 1 仅在 CPU 2 上运行时查看线程 1 中事件的过滤器等效：

```
THRCPU == 0x10002
```

例 5-4 过滤指定时间段发生的事件

过滤介于第 5 秒和第 9 秒之间的时间段中发生的实验 2 的事件：

```
EXPID==2 && TSTAMP >= 5000000000 && TSTAMP < 9000000000
```

例 5-5 过滤来自特定 Java 类的事件

过滤堆栈中具有特定 Java 类的任何方法的事件（在用户查看模式下）：

```
FNAME("myClass.*") SOME IN USTACK
```

例 5-6 按内部函数 ID 和调用序列过滤事件

已知函数 ID（如性能分析器中所示）时，过滤包含计算机调用堆栈中特定调用序列的事件：

```
(314,272) ORDERED IN MSTACK
```

例 5-7 按状态或持续时间过滤事件

如果 `describe` 命令列出时钟分析实验的以下属性：

```
MSTATE    UINT32  Thread state
NTICK     UINT32  Duration
```

可以使用以下过滤器选择处于特定状态的事件：

```
MSTATE == 1
```

或者，可以使用以下过滤器选择处于特定状态且其持续时间比 1 个时钟周期长的事件：

```
MSTATE == 1 && NTICK > 1
```

# er_print 命令示例

本节提供一些使用 `er_print` 命令的示例。

例 5-8 显示函数中时间花费情况的摘要

```
er_print -functions test.1.er
```

例 5-9 显示调用方-被调用方关系

```
er_print -callers-callees test.1.er
```

例 5-10 显示热点源代码行

源代码行信息假设代码已使用 `-g` 进行编译和链接。将尾随下划线附加到 Fortran 函数和例程的函数名称。函数名称后的 `1` 用于区分 `myfunction` 的多个实例。

```
er_print -source  myfunction 1 test.1.er
```

例 5-11　从用户函数堆栈过滤名为 myfunc 的函数：

```
er_print -filters 'FNAME("myfunc") SOME IN USTACK' -functions test.1.er
```

例 5-12　生成类似于 gprof 的输出

以下示例从实验生成一个类似 gprof 的列表。输出是一个名为 er_print.out 的文件，该文件列出前 100 个函数，后跟调用方-被调用方数据（按每个函数的归属用户时间排序）。

```
er_print -outfile  er_print.out -metrics e.%user -sort e.user \
-limit 100 -func -callers-callees test.1.er
```

也可以将此示例简化为以下独立的命令。但是请记住，在大型实验或应用程序中对 er_print 的每次调用可能需要花费很长时间。

```
er_print -metrics  e.%user -limit 100  -functions test.1.er

er_print -metrics  e.%user -callers-callees test.1.er
```

例 5-13　仅显示编译器注释

无需运行程序即可使用此命令。

```
er_src -myfile.o
```

例 5-14　使用挂钟分析来列出函数和调用方-被调用方

```
er_print -metrics  ei.%wall -functions test.1.er

er_print -metrics aei.%wall  -callers-callees test.1.er
```

例 5-15　运行包含 er_print 命令的脚本

```
er_print -script myscriptfile test.1.er
```

myscriptfile 脚本包含 er_print 命令。脚本文件内容的样例如下：

```
## myscriptfile

## Send script output to standard output
outfile -

## Display descriptive information about the experiments
header

## Write out the sample data for all experiments
```

```
overview

## Write out execution statistics, aggregated over
## the current sample set for all experiments
statistics

## List functions
functions

## Display status and names of available load objects
object_list


## Write out annotated disassembly code for systime,
## to file disasm.out
outfile disasm.out
disasm systime


## Write out annotated source code for synprog.c
## to file source.out
outfile source.out
source synprog.c

## Terminate processing of the script
quit
```

♦ ♦ ♦ 第 6 章

# 了解性能分析器及其数据

性能分析器读取收集器收集的事件数据，并将其转换为性能度量。将会针对目标程序结构中的各种元素（如指令、源代码行、函数和装入对象）计算度量。除了包含时间戳、线程 ID、LWP ID 和 CPU ID 的标头外，为收集的每个事件记录的数据还包含以下两部分：

- 用于计算度量的某些事件特定的数据
- 用于将这些度量与程序结构关联的应用程序调用堆栈

由于编译器所进行的插入、转换和优化，将度量与程序结构关联的过程并不总是简单易懂。本章介绍该过程，并讨论对性能分析器显示内容的影响。

本章包含以下主题：



## 数据收集的工作原理

运行数据收集的输出是实验，该实验在文件系统中存储为带有各种内部文件和子目录的目录。

### 实验格式

所有实验都必须具有以下三个文件：

- 日志文件 (`log.xml`)，它是一个 XML 文件，包含有关所收集数据、各种组件的版本、目标生存期内各种事件的记录和目标字大小的信息

- 映射文件 (`map.xml`)，它是一个 XML 文件，所记录的信息包括将哪些装入对象装入目标的地址空间以及装入或卸载这些对象的具体时间
- 概述文件，它是一个二进制文件，包含在实验中的每个抽样点记录的用法信息

此外，实验还具有表示整个处理过程中的分析事件的二进制数据文件。每个数据文件都具有一系列事件，如“解释性能度量” [162]中所述。对于每种类型的数据，都将使用单独的文件，但每个文件都由目标中的所有线程共享。

对于时钟分析或硬件计数器溢出分析，将数据写入时钟周期或计数器溢出调用的信号处理程序中。对于同步跟踪、堆跟踪、I/O 跟踪、MPI 跟踪或 OpenMP 跟踪，从 `LD_PRELOAD` 环境变量在用户调用的常规例程上插入的 `libcollector` 例程写入数据。每个这样的插入例程都部分填充数据记录，然后调用用户调用的常规例程，在该例程返回时填充数据记录的其余部分，并将记录写入数据文件。

所有数据文件都按块进行内存映射和写入。以这样的方式填充记录以便始终具有有效的记录结构，这样就可以在写入时读取实验。缓冲区管理策略设计用于最大限度地减少线程之间的争用和序列化。

实验可以选择性地包含名称为 `notes` 的 ASCII 文件。使用 `collect` 命令的 `-C comment` 参数时会自动创建此文件。创建实验后，可以手动创建或编辑该文件。该文件的内容会置于实验标题之前。

### `archives` 目录

每个实验都具有一个 `archives` 目录，该目录包含描述 `map.xml` 文件中引用的每个装入对象的二进制文件。这些文件由 `er_archive` 实用程序（它在数据收集结束时运行）生成。如果进程异常终止，则可能无法调用 `er_archive` 实用程序，在这种情况下，归档文件由 `er_print` 实用程序或性能分析器在实验上首次调用时写入。

归档目录还包括共享对象文件或源文件的副本，具体取决于用于归档实验的选项。

## 子实验

当分析多个进程（例如，跟踪子孙进程、收集 MPI 实验或者使用用户进程分析内核）时，将创建子实验。

子孙进程将其实验写入创建者实验目录内的子目录。对这些新子实验的命名可指示其衍生情况，如下所示：

- 将下划线附加到创建者的实验名称。
- 添加以下代码字母之一：`f` 代表派生，`x` 代表执行，`c` 代表其他子孙进程。在 Linux 上，使用 `C` 表示 `clone`(2) 生成的子孙。
- 在代码字母后添加一个表示派生或执行的索引的数字。
- 附加实验后缀 `.er` 以构成完整的实验名称。

对于用户进程，如果创建者进程的实验名称为 test.1.er，则其第三个派生创建的子孙进程的实验为 test.1.er/_f3.er。如果该子孙进程执行新映像，则对应的实验名称为 test.1.er/_f3_x1.er。子孙实验由与父实验相同的文件组成，但是它们没有子孙实验（所有子孙实验都由创建者实验中的子目录表示），而且它们没有归档子目录（所有归档都在创建者实验中进行）。

缺省情况下，内核上的实验命名为 ktest.1.er 而不是 test.1.er。当同时收集用户进程的数据时，内核实验将包含每个跟随的用户进程所对应的子实验。内核子实验使用 _*process-name*_PID_*process-id*.1.er 格式命名。例如，在进程 ID 1264 下运行的 sshd 进程上所运行的实验将命名为 ktest.1.er/_sshd_PID_1264.1.er。

缺省情况下，MPI 程序的数据收集到 test.1.er 中，而 MPI 进程的所有数据都收集到子实验中，每个等级一个子实验。收集器使用 MPI 等级以格式 M_r*m*.er 构造子实验名称，其中 *m* 是 MPI 等级。例如，MPI 等级 1 的子实验数据将记录在 test.1.er/M_r1.er 目录中。

### 动态函数

目标创建动态函数的实验在 map.xml 文件中具有描述这些函数的附加记录，还有一个附加文件 dyntext，该文件包含动态函数的实际指令的副本。生成动态函数的带注释反汇编时需要该副本。

### Java 实验

Java 实验在 map.xml 文件中具有附加记录，这两个记录用于 JVM 软件因其内部目的而创建的动态函数和目标 Java 方法的动态编译 (HotSpot) 版本。

此外，Java 实验包括一个 JAVA_CLASSES 文件，该文件包含有关调用的所有用户 Java 类的信息。

使用 JVMTI 代理记录 Java 跟踪数据，该代理是 libcollector.so 的一部分。该代理接收映射到记录的跟踪事件中的事件。该代理还接收类装入和 HotSpot 编译（用于写入 JAVA_CLASSES 文件）的事件以及 map.xml 文件中 Java 编译的方法记录。

## 记录实验

您可以按三种不同的方式记录用户模式目标上的实验：

- 使用 collect 命令
- 使用 dbx 创建进程
- 使用 dbx 从正在运行的进程创建实验

性能分析器中的 "Profile Application"（分析应用程序）对话框可运行 collect 实验。

### collect 实验

使用 collect 命令记录实验时，collect 实用程序会创建实验目录，并设置 LD_PRELOAD 环境变量，以确保将 libcollector.so 及其他 libcollector 模块预装入到目标的地址空间中。然后，该 collect 实用程序设置环境变量，将实验名称和数据收集选项通知 libcollector.so，并执行自己顶部的目标。

libcollector.so 及其相关模块负责写入所有实验文件。

### 创建进程的 dbx 实验

在启用数据收集的情况下使用 dbx 启动进程时，dbx 还会创建实验目录，并确保预装入 libcollector.so。然后 dbx 在其第一个指令前的断点处停止进程，并调用 libcollector.so 中的初始化例程以启动数据收集。

Java 实验不能由 dbx 收集，因为 dbx 使用 Java 虚拟机调试接口 (Java Virtual Machine Debug Interface, JVMDI) 代理进行调试，而该代理无法与数据收集所需的 Java 虚拟机工具接口 (Java Virtual Machine Tools Interface, JVMTI) 代理共存。

### 正在运行的进程上的 dbx 实验

在正在运行的进程上使用 dbx 启动实验时，dbx 会创建实验目录，但不能使用 LD_PRELOAD 环境变量。dbx 对目标进行交互式函数调用以打开 libcollector.so，然后调用 libcollector.so 初始化例程，就像它创建进程时那样。与 collect 实验中一样，数据由 libcollector.so 及其模块写入。

由于进程启动时 libcollector.so 不在目标地址空间中，因此依赖于插入用户可调用函数（同步跟踪、堆跟踪、MPI 跟踪）的所有数据收集可能都不起作用。通常，符号已经解析为底层函数，因此无法发生插入。此外，以下子孙进程也取决于插入，对于 dbx 在正在运行的进程上创建的实验无法正常工作。

如果在使用 dbx 启动进程之前或者在使 dbx 附加到正在运行的进程之前已显式预装入 libcollector.so，可以收集跟踪数据。

## 解释性能度量

每个事件的数据都包含高精度时间戳、线程 ID 和 CPU ID。这些项可用于在性能分析器中按时间、线程或 CPU 过滤度量。有关 CPU ID 的信息，请参见 getcpuid(2) 手册页。在 getcpuid 不可用的系统上，处理器 ID 为 -1（它映射为 "Unknown"（未知））。

除了通用数据外，每个事件还生成特定的原始数据，将在以下各节中对此进行描述。每节还将介绍从原始数据派生的度量的准确性，以及数据收集对度量的影响。

# 时钟分析

时钟分析的事件特定数据由分析间隔计数的数组组成。在 Oracle Solaris 上，提供了间隔计数器。在分析间隔结束时，相应的间隔计数器加 1，并安排另一个分析信号。仅当 Solaris 线程进入 CPU 用户模式时，才记录和重置数组。重置数组包括将用户 CPU 状态的数组元素设置为 1，将所有其他状态的数组元素设置为 0。在重置数组之前，进入用户模式时会记录数组数据。因此，该数组包含自上次进入用户模式以来进入的每个微状态的计数累积（内核为每个 Solaris 线程维护十个微状态）。在 Linux 操作系统上，不存在微状态；唯一的间隔计数器是用户 CPU 时间。

在记录数据的同时记录调用堆栈。如果在分析间隔结束时 Solaris 线程未处于用户模式，则在线程再次进入用户模式之前调用堆栈无法更改。因此，调用堆栈总是会在每个分析间隔结束时准确记录程序计数器的位置。

Oracle Solaris 上每个微状态提供的度量显示在 表 6-1 “内核微状态如何影响度量” 中。

表 6-1 内核微状态如何影响度量

| 内核微状态 | 说明 | 度量名称 |
|---|---|---|
| LMS_USER | 在用户模式下运行 | User CPU Time（用户 CPU 时间） |
| LMS_SYSTEM | 在系统调用或缺页时运行 | System CPU Time（系统 CPU 时间） |
| LMS_TRAP | 在出现任何其他陷阱时运行 | System CPU Time（系统 CPU 时间） |
| LMS_TFAULT | 在用户文本缺页时休眠 | Text Page Fault Time（文本缺页时间） |
| LMS_DFAULT | 在用户数据缺页时休眠 | Data Page Fault Time（数据缺页时间） |
| LMS_KFAULT | 在内核缺页时休眠 | Other Wait Time（其他等待时间） |
| LMS_USER_LOCK | 等待用户模式锁定时休眠 | User Lock Time（用户锁定时间） |
| LMS_SLEEP | 由于任何其他原因而休眠 | Other Wait Time（其他等待时间） |
| LMS_STOPPED | 已停止（/proc、作业控制或 lwp_stop） | Other Wait Time（其他等待时间） |
| LMS_WAIT_CPU | 等待 CPU | Wait CPU Time（等待 CPU 时间） |

## 计时度量的准确性

计时数据是基于统计收集的，因此易于出现任何统计抽样方法的所有误差。对于时间非常短的运行（仅记录少量分析数据包），调用堆栈可能不能表示程序中使用大多数资源的各部分。因此应以足够长的时间或足够多的次数运行程序，以累积感兴趣的函数或源代码行的数百个分析数据包。

除了统计抽样误差外，收集和归属数据的方式以及程序在系统中前进方式也会引起特定的误差。以下是计时度量可能出现不准确或失真的一些情况：

- 创建线程时，记录第一个分析数据包之前所用的时间少于分析间隔，但是整个分析间隔取决于第一个分析数据包中记录的微状态。如果创建了许多线程，则误差可能是分析间隔的许多倍。
- 销毁线程时，在记录最后一个分析数据包后会消耗一些时间。如果销毁了许多线程，则误差可能是分析间隔的许多倍。
- 在分析间隔期间可能发生线程的重新调度。因此，记录的线程状态可能不能表示消耗大部分分析间隔的微状态。如果要运行的线程多于要运行它们的处理器，则误差很可能更大。
- 程序可以与系统时钟相关联的方式运行。在这种情况下，如果线程处于可能表示所用的一小部分时间的状态，而为特定程序部分记录的调用堆栈被过度表示，则分析间隔始终会过期。在多处理器系统上，分析信号可以引入相关性：记录微状态时，在运行程序的线程时由分析信号中断的处理器很可能处于陷阱 CPU 微状态。
- 分析间隔过期时，内核记录微状态值。如果系统负载过重，则该值可能无法表示进程的真实状态。在 Oracle Solaris 上，此情况很可能会导致对陷阱 CPU 或等待 CPU 微状态的过多记帐。
- 系统时钟与外部源同步时，在分析数据包中记录的时间戳不反映分析间隔，但包括对时钟进行的任何调整。时钟调整可能使分析数据包看起来已丢失。涉及的时间段通常为几秒，并且调整是以增量方式进行的。
- 在动态更改其操作时钟频率的计算机上所记录的实验可能在分析中显示不准确之处。

除刚刚介绍的不准确之处外，计时度量还会因收集数据的过程而失真。记录分析数据包所用的时间从不出现在程序的度量中，因为记录是由分析信号启动的。（这是相关性的另一个实例。记录过程中所用的用户 CPU 时间在所记录的任何微状态之间分配。结果是对用户 CPU 时间度量过少记帐，而对其他度量过多记帐。记录数据所用的时间量通常不到缺省分析间隔的 CPU 时间的百分之几。

## 计时度量的比较

如果将通过在基于时钟的实验中进行分析所获得的计时度量与通过其他方式获得的时间进行比较，则应该注意以下问题。

对于单线程应用程序，为进程记录的总线程时间通常精确到千分之几（与同一进程的 `gethrtime`(3C) 返回的值相比）。CPU 时间可能与由同一进程的 `gethrvtime`(3C) 返回的值相差几个百分点。如果负载过重，则差异可能更加明显。但是，CPU 时间差异并不表示系统失真，为不同函数、源代码行等报告的相对时间不会显著失真。

性能分析器中报告的线程时间可能与 `vmstat` 报告的时间有很大差异，因为 `vmstat` 报告的是各 CPU 的线程时间总和。如果目标进程具有的 LWP 比它所运行的系统具有的 CPU 多，则性能分析器显示的等待时间比 `vmstat` 报告的长。

出现在性能分析器的 "Statistics"（统计信息）视图和 `er_print` 统计信息显示中的微状态计时基于进程文件系统 `/proc` 使用报告，因此微状态中所用时间的记录具有很高的

准确性。有关更多信息，请参见 proc (4) 手册页。可以将这些计时与 <Total> 函数（它将程序作为一个整体表示）的度量进行比较，以获取聚集计时度量的准确性指示。但是，"Statistics"（统计信息）视图中显示的值可能包含其他基值，而 <Total> 的计时度量值中不包含这些基值。这些基值来自暂停数据收集的时间段。

用户 CPU 时间和硬件计数器循环时间是不同的，因为在将 CPU 模式切换到系统模式时会关闭硬件计数器。有关更多信息，请参见“陷阱” [168]。

## 硬件计数器溢出分析

硬件计数器溢出分析数据包括计数器 ID 和溢出值。该值可能大于计数器的溢出设置值，因为处理器在事件的溢出和记录之间会执行某些指令。尤其对循环和指令计数器来说，该值可能会更大，这些计数器的递增频率比诸如浮点运算或高速缓存未命中次数的计数器更快。记录事件中的延迟还意味着，通过调用堆栈记录的程序计数器地址并不精确对应于溢出事件。有关更多信息，请参见“硬件计数器溢出的归属” [199]。另请参见“陷阱” [168]的讨论。陷阱和陷阱处理程序可以导致报告的用户 CPU 时间和循环计数器报告的时间有很大差别。

在动态更改其操作时钟频率的计算机上所记录的实验在基于周期的时间计数转换时可能显示不准确之处。

收集的数据量取决于溢出值。选择过小的值可能会产生以下结果：

- 收集数据所用的时间可能是程序执行时间的很大一部分。收集运行可能要用大部分时间来处理溢出和写入数据而不是运行程序。
- 计数的很大一部分可能来自收集过程。这些计数归属到收集器函数 `collector_record_counters`。如果看到此函数的计数很大，则表明溢出值过小。
- 数据的收集可以更改程序的行为。例如，如果要收集有关高速缓存未命中次数的数据，则大多数未命中可能是由刷新收集器指令以及分析高速缓存中的数据并将其替换为程序指令和数据所导致的。程序的高速缓存未命中次数看上去似乎很大，但是如果没有数据收集，则实际上高速缓存未命中次数可能非常小。

## 数据空间分析和内存空间分析

在内存空间分析中，将针对计算机的物理结构（例如，高速缓存行、内存区或页面）报告内存相关事件（例如，高速缓存未命中次数）。

在数据空间分析中，将针对导致事件的数据结构引用而非仅针对发生内存相关事件的指令报告内存相关事件。数据空间分析仅在运行 Oracle Solaris 的 SPARC 系统上可用。它在运行 Oracle Solaris 或 Linux 的 x86 系统上尚不可用。

对于内存空间或数据空间分析，收集的数据必须是使用基于内存的计数器的硬件计数器分析。对于 SPARC 平台或 x86 Oracle Solaris 平台上的精确计数器，缺省情况下将收集内存空间和数据空间数据。

为了支持数据库空间分析，应使用 `-xhwcprof` 标志编译可执行文件。此标志适用于使用 C、C++ 和 Fortran 编译器进行编译，但仅在 SPARC 平台上有意义。在其他平台上将忽略该标志。如果不使用 `-xhwcprof` 编译可执行文件，`er_print` 中的 `data_layout`、`data_single` 和 `data_objects` 命令将不显示数据。数据空间分析无需针对精确计数器使用 `-xhwcprof`。

当实验包含数据空间或内存空间分析时，`er_print` 实用程序允许执行三条附加命令：`data_objects`、`data_single` 和 `data_layout` 以及与内存对象相关的各种命令。有关更多信息，请参见“控制数据空间列表的命令” [136]。

此外，性能分析器包括两个与数据空间分析相关的视图，以及用于内存对象的各种标签。请参见“"DataObjects"（数据对象）视图” [99]、“"DataLayout"（数据布局）视图” [99] 和 “"MemoryObjects"（内存对象）视图” [98]。

运行不带其他参数的 `collect -h` 将列出硬件计数器，并指明这些计数器是否与装入、存储或装入存储相关以及它们是否精确。请参见“硬件计数器分析数据” [23]。

## 同步等待跟踪

收集器通过跟踪对线程库 `libthread.so` 中函数的调用或对实时扩展库 `librt.so` 的调用来收集同步延迟事件。事件特定的数据由请求和授权的高精度时间戳（跟踪的调用的开始和结束）以及同步对象（例如，请求的互斥锁）的地址组成。线程 ID 和 LWP ID 是记录数据时的 ID。等待时间是请求时间和授权时间之间的差值。仅记录其等待时间超过指定阈值的事件。同步等待跟踪数据在授权时记录在实验中。

等待线程不能执行任何其他工作，直到导致延迟的事件完成为止。等待所用时间同时显示为同步等待时间和用户锁定时间。用户锁定时间可能比同步等待时间长，因为同步延迟阈值筛去了短期延迟。

数据收集的开销使等待时间失真。该开销与收集的事件数成比例。通过增加用于记录事件的阈值，可以最大限度地减少开销中的等待时间部分。

## 堆跟踪

收集器通过插入内存分配和取消分配函数 `malloc`、`realloc`、`memalign` 和 `free` 来记录对这些函数的调用的跟踪数据。如果程序分配内存时忽视这些函数，则不记录跟踪数据。不记录 Java 内存管理（它使用不同的机制）的跟踪数据。

跟踪的函数可能从许多库中的任一个库装入。在性能分析器中看到的数据可能取决于给定函数从哪个库装入。

如果程序在很短的时间段内发出对被跟踪函数的大量调用，则执行程序所用的时间可能会大大延长。额外的时间将用于记录跟踪数据。

## I/O 跟踪

收集器可记录对标准 I/O 例程的调用和所有 I/O 系统调用的跟踪数据。

## MPI 跟踪

MPI 跟踪基于修改的 VampirTrace 数据收集器。有关更多信息，请在 Technische Universität Dresden Web 站点上搜索《VampirTrace User Manual》（《VampirTrace 用户手册》）。

# 调用堆栈和程序执行

调用堆栈是一系列程序计数器 (program counter, PC) 地址，表示来自程序内的指令。第一个 PC 称为 叶 *PC*，它位于堆栈的底部，是要执行的下一条指令的地址。下一个 PC 是调用包含叶 PC 的函数的地址；再下一个 PC 是调用该函数的地址，依此类推，直至到达堆栈的顶部。每个这样的地址称为返回地址。记录调用堆栈的过程涉及从程序堆栈获取返回地址，这称为展开堆栈。有关展开失败的信息，请参见“不完全的堆栈展开” [177]。

调用堆栈中的叶 PC 用于将独占度量从性能数据分配到该 PC 所在的函数。堆栈中的每个 PC（包括叶 PC）用于将非独占度量分配到它所在的函数。

大多数时候，已记录调用堆栈中的 PC 自然地对应于出现在程序源代码中的函数，而且性能分析器的已报告度量直接对应于这些函数。但是，有时程序的实际执行并不与程序执行方式的简单直观模型相对应，而且性能分析器的报告度量可能会引起混淆。有关此类情况的更多信息，请参见“将地址映射到程序结构” [178]。

## 单线程执行和函数调用

程序执行的最简单情形是单线程程序调用它自己的装入对象内的函数。

将程序装入到内存中开始执行时，会为其建立上下文，包括要执行的初始地址、初始寄存器集和堆栈（用于存储临时数据和用于跟踪函数如何相互调用的内存区域）。初始地址始终位于函数 `_start()`（它内置于每个可执行程序中）的开头。

程序运行时，将按顺序执行指令，直至遇到分支指令，该指令以及其他指令可能表示函数调用或条件语句。在分支点上，控制权转移到分支目标指定的地址，然后从该地址继续执行。（在 SPARC 上，通常已提交分支后的下一条指令供执行：此指令称为分支延迟槽指令。但是，有些分支指令会取消分支延迟槽指令的执行。

当执行表示调用的指令序列时，返回地址被放入寄存器，且在被调用函数的第一条指令处继续执行。

在大多数情况下，在被调用函数的前几个指令中的某个位置，一个新帧（用于存储有关函数的信息的内存区域）会被推到堆栈上，而返回地址被放入该帧。然后，在被调用函数本身调用其他函数时可以使用用于返回地址的寄存器。函数即将返回时，将其帧从堆栈中弹出，而且控制权返回到从其调用该函数的地址。

## 共享对象之间的函数调用

一个共享对象中的函数调用另一个共享对象中的函数时，其执行情况比在程序内对函数的简单调用要复杂。每个共享对象都包含一个程序链接表 (Program Linkage Table, PLT)，该表包含位于该共享对象外部并从该共享对象引用的每个函数的条目。最初，PLT 中每个外部函数的地址实际上是 `ld.so`（即动态链接程序）内的地址。第一次调用这样的函数时，控制权将转移到动态链接程序，该动态链接程序会解析对实际外部函数的调用并为后续调用修补 PLT 地址。

如果在执行三个 PLT 指令之一的过程中发生分析事件，则 PLT PC 会被删除，并将独占时间归属到调用指令。如果在通过 PLT 条目首次调用期间发生分析事件，但是叶 PC 不是 PLT 指令之一，PLT 和 `ld.so` 中的代码引起的任何 PC 都将归属到人工函数 `@plt` 中，该函数将累计非独占时间。每个共享对象都存在一个这样的人工函数。如果程序使用 `LD_AUDIT` 接口，则可能从不修补 PLT 条目，而且来自 `@plt` 的非叶 PC 可能发生得更频繁。

## 信号

将信号发送到进程时，会发生各种寄存器和堆栈操作，使得发送信号时的叶 PC 看起来好像是调用系统函数 `sigacthandler()` 的返回地址。`sigacthandler()` 调用用户指定的信号处理程序，就像任何函数调用另一个函数一样。

性能分析器将信号传送产生的帧视为普通帧。传送信号时的用户代码显示为调用系统函数 `sigacthandler()`，而 `sigacthandler()` 又显示为调用用户的信号处理程序。来自 `sigacthandler()` 和任何用户信号处理程序以及它们调用的任何其他函数的非独占度量，都显示为中断函数的非独占度量。

收集器通过插入 `sigaction()`，以确保其处理程序在收集时钟数据时是 `SIGPROF` 信号的主处理程序，而在收集硬件计数器溢出数据时是 `SIGEMT` 信号的主处理程序。

## 陷阱

陷阱可以由指令或硬件发出，而由陷阱处理程序捕获。系统陷阱是指通过指令启动的陷阱，它们会陷入内核。所有系统调用均使用陷阱指令实现。一些硬件陷阱示例包括：当浮点单元无法完成指令或者当指令无法在硬件中实现时，浮点单元会发出硬件陷阱。

当发出陷阱时，内核将进入系统模式。在 Oracle Solaris 上，微状态通常从用户 CPU 状态切换到陷阱状态，再切换到系统状态。处理陷阱所用的时间可以显示为系统 CPU 时间和用户 CPU 时间的组合，具体取决于切换微状态的时间点。该时间被归属到用户代码中从其启动陷阱的指令（或归属到系统调用）。

对于某些系统调用，提供尽可能高效的调用处理被认为是很关键的。由这些调用生成的陷阱称为快速陷阱。生成快速陷阱的系统函数包括 `gethrtime` 和 `gethrvtime`。在这些函数中，由于涉及到的开销，所以不会切换微状态。

在其他情况下，提供尽可能高效的陷阱处理也被认为是很关键的。其中的一些示例是 TLB（translation lookaside buffer，转换后备缓冲器）未命中以及寄存器窗口溢出和填充，这类情况下不切换微状态。

在这两种情况下，所用的时间都记录为用户 CPU 时间。但是，由于 CPU 模式已切换为系统模式，所以将关闭硬件计数器。因此，通过求出用户 CPU 时间和周期时间（最好在同一实验中记录）之间的差值，可以估算处理这些陷阱所用的时间。

有一种陷阱处理程序切换回用户模式的情况，那是 Fortran 中在 4 字节边界上对齐的 8 字节整数的未对齐内存引用陷阱。陷阱处理程序的帧出现在堆栈上，而对处理程序的调用可以出现在性能分析器中，归属到整数装入或存储指令。

指令陷入内核后，陷阱指令后的指令看起来要使用很长时间，这是因为它在内核完成陷阱指令的执行之前无法启动。

## 尾部调用优化

只要特定函数执行的最后一个操作是调用另一个函数，编译器就可以执行一种特定的优化。被调用方可重用来自调用方的帧，而不是生成新的帧，而且可从调用方复制被调用方的返回地址。此优化的动机是减小堆栈的大小，以及（在 SPARC 平台上）减少对寄存器窗口的使用。

假定程序源代码中的调用序列与如下所示类似：

```
A -> B -> C -> D
```

对 `B` 和 `C` 进行尾部调用优化后，调用堆栈看起来好像是函数 `A` 直接调用函数 `B`、`C` 和 `D`。

```
A -> B
A -> C
A -> D
```

调用树被展平。使用 `-g` 选项编译代码时，尾部调用优化仅发生在编译器优化级别 4 或更高级别上。在不使用 `-g` 选项的情况下编译代码时，尾部调用优化发生在编译器优化级别 2 或更高级别上。

## 显式多线程

简单的程序在单线程中执行。多线程可执行程序调用线程创建函数（执行的目标函数会传递到该函数）。目标退出时，会销毁线程。

Oracle Solaris 支持两种线程实现：Solaris 线程和 POSIX 线程 (Pthread)。从 Oracle Solaris 10 开始，这两种线程实现都包括在 `libc.so` 中。

对于 Solaris 线程，新创建的线程从名为 `_thread_start()` 的函数开始执行，该函数调用在线程创建调用中传递的函数。对于涉及此线程执行的目标的任何调用堆栈，堆栈的顶部是 `_thread_start()`，与线程创建函数的调用方没有任何联系。因此，与所创建的线程关联的非独占度量最多仅向上传播至 `_thread_start()` 和 `<Total>` 函数。除了创建线程外，Solaris 线程实现还在 Solaris 上创建 LWP 以执行线程。每个线程都绑定到特定的 LWP。

Oracle Solaris 和 Linux 中都提供了用于显式多线程的 Pthread。

在这两种环境中，为创建新线程，应用程序会调用 Pthread API 函数 `pthread_create（）`，将指针作为函数参数之一传递到应用程序定义的启动例程。

在早于 Oracle Solaris 10 的 Solaris 版本上，新的 pthread 开始执行时，将会调用 `_lwp_start（）` 函数。从 Oracle Solaris 10 开始，`_lwp_start（）` 调用中间函数 `_thrp_setup（）`，该中间函数随后调用在 `pthread_create（）` 中指定的应用程序定义的启动例程。

在 Linux 操作系统上，新的 pthread 开始执行时，将会运行 Linux 特定的系统函数 `clone()`，该系统函数调用另一个内部初始化函数 `pthread_start_thread()`，该初始化函数又调用在 `pthread_create（）` 中指定的应用程序定义的启动例程。可用于收集器的 Linux 度量收集函数是线程特定的。因此，`collect` 实用程序运行时，会在 `pthread_start_thread()` 和应用程序定义的线程启动例程之间插入一个名为 `collector_root()` 的度量收集函数。

## 基于 Java 技术的软件执行概述

对于典型的开发者，基于 Java 技术的应用程序就像任何其他程序那样运行。此类应用程序从主入口点（通常名为 `class.main`，可以调用其他方法）开始，就像 C 或 C++ 应用程序那样。

对于操作系统，使用 Java 编程语言（纯 Java 或与 C/C++ 混合）编写的应用程序作为实例化 JVM 软件的进程运行。JVM 软件是从 C++ 源代码编译的，从 `_start`（它会调用 `main` 等）开始执行。它从 `.class` 和/或 `.jar` 文件读取字节码，并执行在该程序中指定的操作。可以指定的操作包括动态装入本机共享对象以及调用该对象内包含的各种函数或方法。

JVM 软件可以执行许多用传统语言编写的应用程序通常不能执行的操作。启动时，该软件会在其数据空间中创建许多动态生成的代码的区域。其中一个区域是用于处理应用程序的字节码方法的实际解释器代码。

在执行基于 Java 技术的应用程序期间，大部分方法由 JVM 软件解释。这些方法称为已解释的方法。Java HotSpot 虚拟机会在解释字节码以检测频繁执行的方法时监视性能。然后，Java HotSpot 虚拟机可能编译重复执行的方法，以生成这些方法的计算机代码。生成的方法称为已编译的方法。之后，虚拟机执行更高效的已编译方法，而不是解释方法的原始字节码。已编译的方法会被装入应用程序的数据空间，并且可能会在之后的某个时间点卸载它们。此外，还会在数据空间中生成其他代码以执行已解释代码和已编译代码之间的转换。

用 Java 编程语言编写的代码还可以直接调用本机编译的代码（C、C++ 或 Fortran）；此类调用的目标称为本机方法。

用 Java 编程语言编写的应用程序本身就是多线程的，对于用户程序中的每个线程，都具有一个 JVM 软件线程。Java 应用程序还具有若干个内务处理线程，用于信号处理、内存管理和 Java HotSpot 虚拟机编译。

在 J2SE 中的 JVMTI 上，可通过各种方法实现数据收集。

### Java 调用堆栈和计算机调用堆栈

性能工具通过记录每个线程生存期中的事件，以及发生事件时的调用堆栈来收集其数据。在执行任何应用程序的任意点上，调用堆栈表示程序在其执行中所处的位置以及它如何到达该位置。混合模型 Java 应用程序区别于传统 C、C++ 和 Fortran 应用程序的一个重要方面是，在运行目标的过程中的任何瞬间，Java 调用堆栈和计算机调用堆栈这两个调用堆栈都有意义。这两个调用堆栈都在配置期间进行记录，并在分析期间进行协调。

### 时钟分析和硬件计数器溢出分析

用于 Java 程序的时钟分析和硬件计数器溢出分析的工作方式与用于 C、C++ 和 Fortran 程序的情况基本相同，不同之处在于前者会同时收集 Java 调用堆栈和计算机调用堆栈。

## Java 分析查看模式

性能分析器为用 Java 编程语言编写的应用程序提供了三种显示性能数据的查看模式：用户模式、专家模式和计算机模式。缺省情况下，将显示用户模式（前提是数据支持它）。下一节汇总了这三种查看模式的主要差异。

## 用户查看模式下的 Java 分析数据

用户模式按名称显示已编译的和已解释的 Java 方法，并以其自然形式显示本机方法。在执行过程中，可能会执行特定 Java 方法的许多实例：已解释的版本，也许还有一个或多个已编译的版本。在用户模式中，所有方法会被聚集显示为一个方法。在性能分析器中缺省选择此查看模式。

用户查看模式中 Java 方法的 PC 与该方法中的方法 ID 和字节码索引相对应；本机函数的 PC 与计算机 PC 相对应。Java 线程的调用堆栈可能同时具有 Java PC 和计算机 PC。它没有对应于 Java 内务处理代码（无 Java 表示法）的任何帧。在某些情况下，JVM 软件无法展开 Java 堆栈，将返回单个帧及特殊函数 `<no Java callstack recorded>`。通常，它占总时间的比例不会超过 5-10%。

在用户模式下的 "Functions"（函数）视图中，会针对所调用的 Java 方法和任何本机方法显示度量。"Callers-Callees"（调用方-被调用方）视图显示用户模式中的调用关系。

Java 方法的源代码对应于 `.java` 文件（从中编译源代码，每个源代码行上都有度量）中的源代码。任何 Java 方法的反汇编显示为其生成的字节码，以及针对每个字节码的度量和交错的 Java 源代码（如果可用）。

Java 表示法中的时间线仅显示 Java 线程。每个线程的调用堆栈与其 Java 方法一起显示。

当前不支持 Java 表示法中的数据空间分析。

## 专家查看模式下的 Java 分析数据

专家模式类似于用户模式，只不过在用户模式下禁止的一些 JVM 内部详细信息会在专家模式中公开。在专家模式下，"Timeline"（时间线）显示所有线程。内务处理线程的调用堆栈是本机调用堆栈。

## 计算机查看模式下的 Java 分析数据

计算机模式显示来自 JVM 软件本身而不是来自 JVM 软件解释的应用程序的函数。该表示法还显示所有已编译方法和本机方法。计算机模式看起来与用传统语言编写的应用程序的计算机模式相同。调用堆栈显示 JVM 帧、本地帧和编译方法帧。一些 JVM 帧表示已解释的 Java、已编译的 Java 和本机代码之间的转换代码。

针对 Java 源代码显示已编译方法的源代码；数据表示所选已编译方法的特定实例。已编译方法的反汇编显示生成的计算机汇编程序代码，而不是 Java 字节码。调用方-被调用方关系显示所有开销帧，以及表示已解释方法、已编译方法和本机方法之间的转换的所有帧。

在计算机查看模式下的 "Timeline"（时间线）中，所有线程、LWP 或 CPU 都显示时间线栏，而其中每项的调用堆栈都是计算机模式的调用堆栈。

# OpenMP 软件执行概述

OpenMP 应用程序的实际执行模型在 OpenMP 规范中进行了描述（例如，请参见《OpenMP Application Program Interface, Version 3.0》（《OpenMP 应用程序接口 3.0 版》）中的 1.3 节）。但是，该规范未描述对用户可能很重要的一些实现详细信息，在 Oracle 的实际实现中，用户通过直接记录的分析信息并不能轻松了解线程是如何交互的。

在任何单线程程序运行时，其调用堆栈会显示其当前位置，以及如何到达那里的跟踪，跟踪从名为 `_start` 的例程（该例程调用 `main`，后者又继续调用程序内的各种子例程）中的起始指令开始。如果子例程包含循环，则程序会重复执行循环内的代码，直至达到循环退出条件。然后继续执行下一个代码序列，依此类推。

通过 OpenMP 并行化程序（因为是通过自动并行化）时，行为是不同的。该并行化程序的直观模型具有像单线程程序那样执行的主线程。当它到达并行循环或并行区域时，将出现其他从属线程（每个都是主线程的克隆），它们都并行执行循环或并行区域的内容，每个从属线程执行不同的工作块。在完成所有工作块时，所有线程都是同步的，从属线程将消失，而主线程继续运行。

并行化程序的实际行为并非如此简单。在 Oracle 实现中，当编译器为并行区域或循环（或任何其他 OpenMP 构造）生成代码时，将提取其内部的代码，并使之成为一个名为 *mfunction* 的独立函数。（也可以将它称为外联函数或循环体函数。）mfunction 的名称是对 OpenMP 构造类型、从中提取它的函数的名称以及该构造所在源代码行的行号进行编码的结果。这些函数的名称在性能分析器的专家模式和计算机模式下按以下形式显示，其中方括号中的名称是函数的实际符号表名称：

```
bardo_ -- OMP parallel region from line 9 [_$p1C9.bardo_]
atomsum_ -- MP doall from line 7 [_$d1A7.atomsum_]
```

还有其他形式的此类函数，它们是从其他源代码构造派生的，名称中的 `OMP parallel region` 会被替换为 `MP construct`、`MP doall` 或 `OMP sections`。在下面的讨论中，所有这些都统称为并行区域。

执行并行循环内代码的每个线程都可以多次调用其 mfunction，每次调用执行循环内的一个工作块。在所有工作块完成时，每个线程调用库中的同步或归约例程；然后主线程继续，而从属线程变为空闲状态，等待主线程进入下一个并行区域。所有调度和同步都是通过对 OpenMP 运行时的调用处理的。

在其执行过程中，并行区域内的代码可能执行一个工作块，或者它可能与其他线程同步，或者选择要执行的其他工作块。它还可能调用其他函数，而这些函数又可能会再调用其他函数。在并行区域内执行的从属线程（或主线程）可能本身（或者通过它调用的函数）充当主线程，并进入它自己的并行区域，从而导致嵌套并行操作。

性能分析器基于调用堆栈的统计抽样收集数据，跨所有线程聚集其数据，并针对函数、调用方和被调用方、源代码行和指令，基于所收集数据的类型显示性能度量。性能分析器按以下三种查看模式之一提供有关 OpenMP 程序性能的信息：用户模式、专家模式和计算机模式。

有关 OpenMP 程序的数据收集的更多详细信息，请参见 OpenMP 用户社区 Web 站点上的《An OpenMP Runtime API for Profiling》 (http://www.compunity.org/futures/omp-api.html)（《用于分析的 OpenMP 运行时 API》）。

## 用户查看模式下的 OpenMP 分析数据

以用户模式显示分析数据时，好像程序按照“OpenMP 软件执行概述” [173]中所述的直观模型实际执行一样。在计算机模式下显示的实际数据捕获运行时库 `libmtsk.so`（它不对应于模型）的实现详细信息。专家模式显示为匹配模型而改变的混合数据和实际数据。

在用户模式下，更改了分析数据的显示以便更好地匹配模型，在以下方面不同于记录的数据和计算机模式显示：

- 从 OpenMP 运行时库的角度来看，构造的人工函数表示每个线程的状态。
- 处理调用堆栈以报告对应于代码运行方式模型的数据，如上所述。
- 为时钟分析实验构造另外两个性能度量，它们分别对应于执行有用工作所用的时间和在 OpenMP 运行时中等待所用的时间。度量为 "OpenMP Work Time"（OpenMP 工作时间）和 "OpenMP Wait Time"（OpenMP 等待时间）。
- 对于 OpenMP 3.0 和 4.0 程序，又构造了一个度量 "OpenMP Overhead Time"（OpenMP 开销时间）。

### 人工函数

构造人工函数，并将其放置在用户模式和专家模式调用堆栈上，以反映线程在 OpenMP 运行时库中处于某个状态的事件。

定义了以下人工函数：

| | |
|---|---|
| `<OMP-overhead>` | 在 OpenMP 库中执行 |
| `<OMP-idle>` | 从属线程，等待工作 |
| `<OMP-reduction>` | 执行归约操作的线程 |
| `<OMP-implicit_barrier>` | 在隐式屏障处等待的线程 |
| `<OMP-explicit_barrier>` | 在显式屏障处等待的线程 |
| `<OMP-lock_wait>` | 等待锁定的线程 |
| `<OMP-critical_section_wait>` | 等待进入临界段的线程 |
| `<OMP-ordered_section_wait>` | 等待轮流进入排序段的线程 |
| `<OMP-atomic_wait>` | 等待 OpenMP 原子构造的线程。 |

当线程处于对应于其中一个人工函数的 OpenMP 运行时状态时，会将该人工函数作为堆栈上的叶函数添加。当线程的实际叶函数处于 OpenMP 运行时中的任意位置时，`<OMP-`

overhead> 将作为叶函数替换它。否则，从用户模式堆栈中忽略 OpenMP 运行时中的所有 PC。

对于 OpenMP 3.0 和 4.0 程序，不使用 <OMP-overhead> 人工函数。由 "OpenMP Overhead"（OpenMP 开销）度量替换人工函数。

### 用户模式调用堆栈

对于 OpenMP 实验，用户模式显示重构的调用堆栈，这些重构的调用堆栈类似于在不使用 OpenMP 的情况下编译程序时获取的调用堆栈。目的在于以与程序的直观了解相匹配的方式提供分析数据，而不是显示实际处理的所有详细信息。当 OpenMP 运行时库执行特定操作时，将协调主线程与从线程的调用堆栈并将人工 <OMP-*> 函数添加到此调用堆栈。

## OpenMP 度量

处理 OpenMP 程序的时钟分析事件时，将显示两个度量（它们分别对应于 OpenMP 系统中的两种状态所用的时间）："OpenMP Work Time"（OpenMP 工作时间）和 "OpenMP Wait Time"（OpenMP 等待时间）。

只要从用户代码执行线程（不管串行执行还是并行执行），就会在 "OpenMP Work Time"（OpenMP 工作时间）中累计时间。只要线程正在等待某项，之后才能继续，就会在 "OpenMP Wait Time"（OpenMP 等待时间）中累计时间，而不管等待是忙等待（自旋等待）还是休眠。这两个度量的总和与时钟分析中的 "Total Thread Time"（总线程时间）度量相匹配。

在用户模式、专家模式和计算机模式下显示 "OpenMP Wait Time"（OpenMP 等待时间）和 "OpenMP Work Time"（OpenMP 工作时间）度量。

## 专家查看模式下的 OpenMP 分析数据

在专家查看模式下查看 OpenMP 实验时，如果 OpenMP 运行时正在执行某些特定操作，您将看到格式为 <OMP-*> 的人工函数，这一点类似于用户查看模式。但是，专家查看模式单独显示表示并行化循环、任务等的编译器生成的 mfunction。用户模式会将这些编译器生成的 mfunction 与用户函数聚集在一起。

## 计算机查看模式下的 OpenMP 分析数据

计算机模式显示所有线程和编译器生成的外联函数的本机调用堆栈。

在执行的各个阶段中，程序的实际调用堆栈与用户模型中提及的堆栈有很大差异。计算机模式将调用堆栈显示为已度量，没有进行转换，且没有构造人工函数。但是，仍显示时钟分析度量。

在下面的每个调用堆栈中，`libmtsk` 表示 OpenMP 运行时库内调用堆栈中的一个或多个帧。就像屏障代码的内部实现或执行归约一样，出现哪些函数以及出现顺序的详细信息随 OpenMP 的发行版的不同而不同。

1. 在第一个并行区域之前

   在进入第一个并行区域之前，只有一个线程，即主线程。调用堆栈与用户模式和专家模式下的完全相同。

| 主线程 |
| --- |
| foo |
| main |
| _start |

2. 在并行区域中执行时

| 主线程 | 从属线程 1 | 从属线程 2 | 从属线程 3 |
| --- | --- | --- | --- |
| foo-OMP... | | | |
| libmtsk | | | |
| foo | foo-OMP... | foo-OMP... | foo-OMP... |
| main | libmtsk | libmtsk | libmtsk |
| _start | _lwp_start | _lwp_start | _lwp_start |

   在计算机模式下，从属线程显示为在 `_lwp_start` 中启动，而不是在 `_start`（主线程在其中启动）中启动。（在线程库的某些版本中，该函数可能显示为 `_thread_start`。对 `foo-OMP...` 的调用表示为并行化区域生成的 mfunction。

3. 所有线程都在屏障处时

| 主线程 | 从属线程 1 | 从属线程 2 | 从属线程 3 |
| --- | --- | --- | --- |
| libmtsk | | | |
| foo-OMP... | | | |
| foo | libmtsk | libmtsk | libmtsk |
| main | foo-OMP... | foo-OMP... | foo-OMP... |
| _start | _lwp_start | _lwp_start | _lwp_start |

与线程在并行区域中执行时不同，当线程在屏障处等待时，在 foo 和并行区域代码 foo-OMP... 之间没有来自 OpenMP 运行时的帧。原因是实际执行中不包括 OMP 并行区域函数，但 OpenMP 运行时处理寄存器，以便堆栈展开显示从最后执行的并行区域函数到运行时屏障代码的调用。如果没有它，在计算机模式下将无法确定哪个并行区域与屏障调用相关。

4. 离开并行区域之后

| 主线程 | 从属线程 1 | 从属线程 2 | 从属线程 3 |
|---|---|---|---|
| foo | | | |
| main | libmtsk | libmtsk | libmtsk |
| _start | _lwp_start | _lwp_start | _lwp_start |

在从属线程中，没有用户帧位于调用堆栈上。

5. 在嵌套并行区域中时

| 主线程 | 从属线程 1 | 从属线程 2 | 从属线程 3 | 从属线程 4 |
|---|---|---|---|---|
| | bar-OMP... | | | |
| foo-OMP... | libmtsk | | | |
| libmtsk | bar | | | |
| foo | foo-OMP... | foo-OMP... | foo-OMP... | bar-OMP... |
| main | libmtsk | libmtsk | libmtsk | libmtsk |
| _start | _lwp_start | _lwp_start | _lwp_start | _lwp_start |

# 不完全的堆栈展开

堆栈展开在“调用堆栈和程序执行” [167]中定义。

堆栈展开可能由于多种原因而失败：

- 如果堆栈已被用户代码破坏；如果是这样，则程序可能进行信息转储，或者数据收集代码可能进行信息转储，具体取决于堆栈被破坏的确切方式。
- 如果用户代码不遵循函数调用的标准 ABI 约定。特别是，在 SPARC 平台上，如果在执行保存指令之前更改了返回寄存器 %o7。

  在任何平台上，手工编写的汇编程序代码都可能违反约定。
- 如果在从堆栈中弹出被调用方的帧之后，但在函数返回之前，叶 PC 位于函数中。
- 如果调用堆栈包含的帧超过 250 个，则收集器没有用于完全展开调用堆栈的空间。在这种情况下，调用堆栈中从 _start 到某个点的函数的 PC 不会记录在实验

中。人工函数 <Truncated-stack> 显示为从 <Total> 调用，以清点所记录的最上面的帧。

- 如果收集器无法在 x86 平台上展开优化的函数的帧。

### 中间文件

如果使用 -E 或 -P 编译器选项生成中间文件，则性能分析器将中间文件用于带注释的源代码，而不是原始源文件。使用 -E 生成的 #line 指令可能会导致为源代码行分配度量时出现问题。

如果函数中的指令没有行号（这些行号引用为生成该函数而编译的源文件），则带注释的源代码中会出现以下行：

*function_name* -- <instructions without line numbers>

在以下情况下可能缺少行号：

- 编译时未指定 -g 选项。
- 调试信息在编译后被剥离，或者包含该信息的可执行文件或对象文件被移动或删除或者随后被修改。
- 函数包含从 #include 文件而不是从原始源文件生成的代码。
- 在进行较高级别优化时，如果代码从不同文件中的函数内联。
- 源文件具有引用某个其他文件的 #line 指令；如果使用 -E 选项进行编译，然后再编译生成的 .i 文件，就会出现此情况。使用 -P 标志编译时也可能会出现此情况。
- 找不到读取行号信息的对象文件。
- 所用的编译器生成不完整的行号表。

## 将地址映射到程序结构

将调用堆栈处理为 PC 值后，分析器会将这些 PC 映射到程序中的共享对象、函数、源代码行和反汇编行（指令）。本节将介绍这些映射。

### 进程映像

运行程序时，会从该程序的可执行文件对进程进行实例化。该进程在其地址空间中具有许多区域，其中一些区域是文本，表示可执行指令，而另一些区域是通常不执行的数据。在调用堆栈中记录的 PC 通常对应于程序的某个文本段中的地址。

进程中的第一个文本段从可执行文件本身派生。其他文本段对应于与可执行文件一起装入（在启动进程时，或由进程动态装入）的共享对象。调用堆栈中的 PC 基于记录调用堆栈时装入的可执行文件和共享对象进行解析。可执行文件和共享对象非常类似，它们统称为装入对象。

由于可以在程序执行过程中装入和卸载共享对象，因此在运行期间的不同时间，任何给定的 PC 可能对应于不同的函数。此外，当卸载共享对象，然后在不同地址上重新装入它时，不同时间的不同 PC 可能对应于同一函数。

## 装入对象和函数

每个装入对象，不管是可执行文件还是共享对象，都包含一个文本段（含有编译器生成的指令）、一个存储数据的数据段以及各种符号表。所有装入对象都必须包含 ELF 符号表，该符号表提供该对象中所有全局已知函数的名称和地址。使用 `-g` 选项编译的装入对象包含附加的符号信息，该信息可以扩充 ELF 符号表，并提供有关非全局函数的信息、有关函数来自的对象模块的附加信息以及使地址与源代码行相关联的行号信息。

术语函数 用于描述一组表示源代码中所述的高级别操作的指令。该术语涵盖 Fortran 中所用的子例程，C++ 和 Java 编程语言中所用的方法等等。函数在源代码中进行了清晰的描述，通常其名称出现在表示一组地址的符号表中；如果程序计数器位于该组中，则程序正在该函数内执行。

原则上，装入对象文本段中的任何地址都可以映射到函数。调用堆栈上的叶 PC 和所有其他 PC 都使用完全相同的映射。大多数函数直接对应于程序的源模型。一些函数却不是这样，将在以下各节中介绍这些函数。

## 有别名的函数

通常，函数被定义为全局函数，这意味着其名称在程序中的所有位置都是已知的。全局函数的名称在可执行文件中必须唯一。如果在地址空间中存在多个具有同一给定名称的全局函数，则运行时链接程序将解析对其中之一的所有引用。从不执行其他全局函数，因此它们不会出现在函数列表中。在 "Selection Details"（选择详细信息）窗口中，可以看到包含所选函数的共享对象和对象模块。

在不同情况下，一个函数可以具有若干个不同的名称。非常常见的示例是，将所谓的弱符号和强符号用于同一代码段。强名称通常与对应的弱名称相同，不同之处是它具有一个前导下划线。线程库中的许多函数还具有 pthread 和 Solaris 线程的备用名称，以及强名称、弱名称和备用内部符号。在所有此类情况下，性能分析器的 "Functions"（函数）视图中仅使用一个名称。所选名称是给定地址处按字母顺序排序的最后一个符号。此选择通常对应于用户将使用的名称。在 "Selection Details"（选择详细信息）窗口中，将显示所选函数的所有别名。

## 非唯一函数名称

尽管有别名的函数反映同一代码段的多个名称，但是在某些情况下，多个代码段具有相同的名称：

- 有时，由于模块化原因，函数被定义为静态的，这意味着其名称仅在程序的某些部分（通常为单个已编译的对象模块）中是已知的。在这样的情况下，引用程序完全不同部分的若干个同名函数将出现在性能分析器中。在 "Selection Details"（选择详细信息）窗口中，显示其中每个函数的对象模块名称以便将它们区分开。此外，选择这些函数中的任何一个都可以用于显示该特定函数的源代码、反汇编以及调用方和被调用方。
- 有时，程序使用库中具有函数弱名称的包装函数或插入函数，并取代对该库函数的调用。有些包装函数调用库中的原始函数，在这种情况下，名称的两个实例都出现在性能分析器函数列表中。这样的函数来自不同的共享对象和不同的对象模块，这样它们彼此可以区分开。收集器包装某些库函数，而且包装函数和实际函数都可以出现在性能分析器中。

## 来自剥离共享库的静态函数

静态函数通常在库中使用，因此库内部使用的名称不会与用户可能使用的名称发生冲突。库被剥离后，静态函数的名称将从符号表中删除。在这种情况下，性能分析器会为包含剥离静态函数的库中的每个文本区域生成人工名称。该名称的格式为 `<static>@0x12345`，其中 `@` 符号后的字符串是库中文本区域的偏离量。性能分析器无法区分连续的剥离静态函数和单个这样的函数，因此可能出现两个或多个这样的函数，且其度量合并在一起。

剥离静态函数显示为从正确的调用方进行调用，例外情况为静态函数中的 PC 是出现在静态函数中保存指令后的叶 PC 时。如果没有符号信息，则性能分析器不知道保存地址，无法断定是否将返回寄存器用作调用方。它会始终忽略返回寄存器。由于可以将若干个函数合并为一个 `<static>@0x12345` 函数，因此可能不能将实际调用方或被调用方与相邻函数区分开。

## Fortran 备用入口点

Fortran 可使单个代码段具有多个入口点，使调用方调用到函数的中间。在编译这样的代码时，它包含主入口点的序言 (prologue)、备用入口点的序言和函数的代码主体。每个序言为函数的最终返回设置堆栈，然后转移或下行到代码的主体。

每个入口点的序言代码始终对应于具有该入口点名称的文本区域，但是子例程主体的代码仅接收可能的入口点名称之一。接收的名称随编译器的不同而不同。

序言很少占用大量时间，而且对应于除了与子例程的主体关联的入口点之外的入口点的函数很少出现在性能分析器中。在具有备用入口点的 Fortran 子例程中表示时间的调用堆栈通常在子例程的主体而不是前言中具有 PC，而且只有与主体关联的名称才显示为被调用方。同样，来自子例程的所有调用都显示为从与子例程主体关联的名称进行。

## 克隆函数

编译器能够识别可以对其执行额外优化的函数调用。此类调用的一个示例是对其某些参数为常量的函数的调用。当编译器识别出它可以优化的特定调用时，会创建该函数的副本（称为克隆）并生成优化代码。克隆函数名称是标识特定调用的改编名称 (mangled name)。性能分析器取消改编 (demangle) 该名称，并在函数列表中单独显示克隆函数的每个实例。每个克隆函数都具有不同的指令集，因此带注释的反汇编列表单独显示克隆函数。每个克隆函数都具有相同的源代码，因此带注释的源代码列表汇总了函数的所有副本的数据。

## 内联函数

内联函数是这样的函数：在函数的调用点上（而不是实际调用上）为其插入由编译器生成的指令。有两种类型的内联，执行它们都可提高性能，并且它们都影响性能分析器。这两种类型是 C++ 内联函数定义和显式或自动内联。

- C++ 内联函数定义。此情况下内联的理论基础是，调用函数的成本比内联函数完成的工作要大得多，因此在调用点上插入函数的代码比设置函数调用好得多。通常，访问函数被定义为进行内联，因为它们通常仅需要一条指令。使用 `-g` 选项进行编译时，会禁用函数内联；使用 `-g0` 编译时，允许函数内联，这是建议的做法。
- 在高优化级别（4 和 5）上，编译器执行显式或自动内联。甚至在打开 `-g` 时，也会执行显式和自动内联。这种类型内联的理论基础可能在于，节省了函数调用的成本，但更为常见的目的是提供可以优化寄存器使用和指令调度的更多指令。

这两种类型的内联对于度量的显示具有相同的效果。出现在源代码中但已被内联的函数不出现在函数列表中，也不显示为它们内联到的函数的被调用方。否则在内联函数的调用点上显示为非独占度量的度量（表示被调用函数中所用的时间）实际上将显示为归属到调用点的独占度量（表示内联函数的指令）。

---

注 - 内联可能会使数据难以解释，因此在编译程序以进行性能分析时，可能希望禁用内联。不应禁用 C++ 访问函数的内联，因为它会导致高性能成本。

---

在某些情况下，甚至在函数被内联时，会留下所谓的外部函数 (out-of-line function)。某些调用点调用外部函数 (out-of-line function)，而其他调用点使指令内联。在这样的情况下，函数出现在函数列表中，但归属到该函数的度量仅表示外部调用 (out-of-line call)。

## 编译器生成的主体函数

编译器并行化函数中的循环或具有并行化指令的区域时，将会创建初始源代码中不存在的新主体函数。“OpenMP 软件执行概述” [173]中介绍了这些函数。

在用户模式下，性能分析器不显示这些函数。在专家模式和计算机模式下，性能分析器将这些函数显示为常规函数，除了编译器生成的名称外，性能分析器还基于从其提取

这些函数的函数为其分配名称。其独占度量和非独占度量表示在主体函数中所用的时间。此外，从其提取构造的函数显示每个主体函数的非独占度量。“OpenMP 软件执行概述” [173]中介绍了实现这一点的过程。

内联一个包含并行循环的函数时，其编译器生成的主体函数的名称反映它所内联到的函数，而不是初始函数。

---

注 - 只有使用 -g 编译的模块才能取消改编 (demangle) 编译器生成主体函数的名称。

---

# 外联函数

外联函数可以在反馈优化编译期间创建。它们表示正常情况下不执行的代码，特别是在用于生成最终优化编译反馈的训练运行期间不执行的代码。一个典型的示例是，对来自库函数的返回值执行错误检查的代码；正常情况下从不运行错误处理代码。为改进分页和指令高速缓存行为，将这样的代码移动到地址空间的其他位置，并使其成为单独的函数。外联函数的名称对有关外联代码段的信息进行编码，包括从其提取代码的函数的名称和源代码中该段开头的行号。这些改编名称 (mangled name) 可能随发行版的不同而不同。性能分析器提供了函数名称的可读版本。

外联函数不会被真正调用，而是跳转到它们；同样，它们不返回，而是跳回。为了使该行为与用户的源代码模型更紧密地匹配，性能分析器将来自主函数的人工调用转嫁于其外联部分。

外联函数显示为常规函数，具有相应的非独占度量和独占度量。此外，外联函数的度量作为代码从中进行外联的函数的非独占度量添加。

有关反馈优化编译的更多详细信息，请参见下列手册之一中的 `-xprofile` 编译器选项说明：

- 《Oracle Solaris Studio 12.4：C 用户指南》中的附录 B “C 编译器选项参考”
- 《Oracle Solaris Studio 12.4：C++ 用户指南》中的附录 A “C++ 编译器选项”
- 《Oracle Solaris Studio 12.4：Fortran 用户指南》中的第 3 章 “Fortran 编译器选项”

# 动态编译的函数

动态编译的函数是指程序执行时编译和链接的函数。收集器中并没有有关用 C 或 C++ 编写的动态编译函数的信息，除非用户使用收集器 API 函数提供所需的信息。有关 API 函数的信息，请参见“动态函数和模块” [46]。如果未提供信息，则函数在性能分析工具中显示为 `<Unknown>`。

对于 Java 程序，收集器获取有关由 Java HotSpot 虚拟机编译的方法的信息，无需使用 API 函数提供信息。对于其他方法，性能工具显示有关执行方法的 JVM 软件的信息。在

Java 表示法中，所有方法都与已解释版本合并在一起。在计算机表示法中，单独显示每个 HotSpot 编译的版本，且为每个已解释的方法显示 JVM 函数。

## <Unknown> 函数

在某些情况下，PC 不会映射到已知函数。在这样的情况下，PC 会映射到名为 <Unknown> 的特殊函数。

以下是 PC 映射到 <Unknown> 的情形：

- 动态生成用 C 或 C++ 编写的函数，且未使用收集器 API 函数为收集器提供有关函数的信息时。有关收集器 API 函数的更多信息，请参见“动态函数和模块” [46]。
- 动态编译 Java 方法但禁用 Java 程序分析时。
- PC 对应于可执行文件或共享对象的数据段中的地址时。一种情况是 `libc.so` 的 SPARC V7 版本，在其数据段中有多个函数（例如，`.mul` 和 `.div`）。代码位于数据段中，以便库检测到该代码正在 SPARC V8 或 SPARC V9 平台上执行时可以动态重新编写该代码以使用计算机指令。
- PC 对应于在实验中未记录的可执行文件的地址空间中的共享对象时。
- PC 不在任何已知的装入对象中时。最有可能的原因是展开失败，其中记录为 PC 的值根本不是 PC，而是某个其他字。如果 PC 是返回寄存器，并且看上去不在任何已知的装入对象中，则该 PC 会被忽略，而不是归属到 <Unknown>> 函数。
- PC 映射到收集器没有其符号信息的 JVM 软件的内部部分时。

<Unknown> 函数的调用方和被调用方表示调用堆栈中的上一个和下一个 PC，并以常规方式处理。

# OpenMP 特殊函数

构造人工函数，并将其放置到用户模式调用堆栈（这些用户模式调用堆栈反映线程在 OpenMP 运行时库内处于某个状态的事件）。定义了以下人工函数：

| | |
|---|---|
| <OMP-overhead> | 在 OpenMP 库中执行 |
| <OMP-idle> | 从属线程，等待工作 |
| <OMP-reduction> | 执行归约操作的线程 |
| <OMP-implicit_barrier> | 在隐式屏障处等待的线程 |
| <OMP-explicit_barrier> | 在显式屏障处等待的线程 |
| <OMP-lock_wait> | 等待锁定的线程 |
| <OMP-critical_section_wait> | 等待进入临界段的线程 |
| <OMP-ordered_section_wait> | 等待轮流进入排序段的线程 |

### <JVM-System> 函数

在用户表示法中，<JVM-System> 函数表示 JVM 软件执行操作而不是运行 Java 程序所用的时间。在此时间间隔中，JVM 软件执行诸如垃圾收集和 HotSpot 编译之类的任务。缺省情况下，可在函数列表中看到 <JVM-System>。 “<JVM-System> 函数” [184]

### <no Java callstack recorded> 函数

<no Java callstack recorded> 函数类似于 <Unknown> 函数，但它仅用于 Java 表示法中的 Java 线程。当收集器从 Java 线程收到事件时，收集器会展开本机堆栈并调用到 JVM 软件中，以获取对应的 Java 堆栈。如果该调用由于任何原因而失败，则该事件会与人工函数 <no Java callstack recorded> 一起显示在性能分析器中。为了避免死锁，或展开 Java 堆栈时导致过度同步，JVM 软件可能会拒绝报告调用堆栈。

### <Truncated-stack> 函数

性能分析器为记录调用堆栈中各个函数的度量所用的缓冲区大小是有限的。如果调用堆栈大小变得如此大而导致缓冲区变满，则对调用堆栈大小的任何进一步增加都将强制性能分析器删除函数分析信息。由于在大多数程序中，大部分独占 CPU 时间用在叶函数中，因此性能分析器删除堆栈底部不太重要的函数（从入口函数 _start() 和 main() 开始）的度量。已删除函数的度量将合并到单个人工 <Truncated-stack> 函数中。<Truncated-stack> 函数也可能出现在 Java 程序中。

要支持较深的堆栈，请将 SP_COLLECTOR_STACKBUFSZ 环境变量设置为较大的数字。

### <Total> 函数

<Total> 函数是一个人工结构，用于将程序作为一个整体表示。除了归属到调用堆栈上的函数外，所有性能度量都归属到特殊函数 <Total>。该函数出现在函数列表的顶部，其数据可以用于为其他函数的数据提供透视。在 "Callers-Callees"（调用方-被调用方）列表中，此函数显示为所执行的任何程序的主线程中 _start() 的名义调用方，还显示为已创建线程的 _thread_start() 的名义调用方。如果堆栈展开是不完整的，则 <Total> 函数可能显示为 <Truncated-stack> 的调用方。

## 与硬件计数器溢出分析相关的函数

以下函数与硬件计数器溢出分析相关：

- collector_not_program_related – 计数器与程序不相关。
- collector_hwcs_out_of_range – 计数器似乎已超过溢出值，但未生成溢出信号。将会记录该值，并重置计数器。
- collector_hwcs_frozen – 计数器似乎已超过溢出值并被停止，但溢出信号似乎已丢失。将会记录该值，并重置计数器。
- collector_hwc_ABORT – 读取硬件计数器已失败（通常在特权进程已控制计数器时），导致硬件计数器收集终止。
- collector_record_counter – 处理和记录硬件计数器事件时累积的计数，一部分用于说明硬件计数器溢出分析开销。如果此值对应于 <Total> 计数的重要部分，则建议使用较大的溢出间隔（即，较低的精度配置）。

# 将性能数据映射到索引对象

索引对象表示可以通过每个数据包中记录的数据计算其索引的对象集。预定义的索引对象集包括：线程、CPU、抽样和秒。其他索引对象可通过直接发出的或 .er.rc 文件中的 er_print indxobj_define 命令定义。在性能分析器中，可以通过从 "Tools"（工具）菜单中选择 "Settings"（设置），再选择 "Views"（视图）标签，然后单击 "Add Custom Index Object View"（添加定制索引对象视图）按钮来定义索引对象。

对于每个数据包，将会计算索引，并将与包关联的度量添加到该索引处的索引对象。索引 -1 映射到 <Unknown> 索引对象。索引对象的所有度量都是独占度量，因为索引对象的分层表示都是没有意义的。

# 将性能数据映射到内存对象

内存对象是内存子系统中的组件，如高速缓存行、页面和内存区。对象是通过从所记录的虚拟地址或物理地址计算的索引确定的。为虚拟页面和物理页面预定义了内存对象，其大小可以为 8 KB、64 KB、512 KB 和 4 MB。您可以在 er_print 实用程序中使用 mobj_define 命令定义其他内存对象。您也可以使用性能分析器的 "Settings"（设置）对话框中的 "Add Memory Objects View"（添加内存对象视图）按钮定义定制内存对象。有关更多信息，请参见“配置设置” [113]。

可以为特定的 SPARC 系统体系结构装入用于定义 "Memory Objects"（内存对象）的文件。单击 "Load Machine Model"（装入计算机模型）按钮并选择感兴趣的系统体系结构。单击 "Apply"（应用）或 "OK"（确定），此时新的对象列表将显示在 "Memory Objects Views"（内存对象视图）列中。可以从这些视图中进行选择以显示关联的数据。在帮助中搜索“计算机模型”以了解更多信息。

缺省情况下，性能分析器将装入对记录实验所在的计算机适用的计算机模型文件。计算机模型文件可定义内存对象和索引对象。

# 将数据地址映射到程序数据对象

当对应于内存操作的硬件计数器事件的 PC 已被处理，可以成功回溯到很可能引发事件的内存引用指令时，性能分析器将使用编译器在其硬件分析支持信息中提供的指令标识符和描述符派生关联的程序数据对象。

术语数据对象用于表示程序常量、变量、数组和聚集（如结构和联合），以及源代码中所述的各种聚集元素。数据对象的类型及其大小随源语言的不同而不同。许多数据对象是在源程序中显式命名的，而其他数据对象可能是未命名的。有些数据对象是从其他（更简单的）数据对象派生或聚集的，从而产生了一组丰富的、通常很复杂的数据对象。

每个数据对象都具有关联的作用域，即在其中定义数据对象并可以引用数据对象的源程序区域。该作用域可以是全局性的（如装入对象），也可以是特定的编译单元（对象文件）或函数。可以使用不同的作用域定义相同的数据对象，也可以在不同的作用域内以不同的方式引用特定数据对象。

在启用回溯的情况下为内存操作的硬件计数器事件收集的数据派生度量被归属到关联的程序数据对象类型。这些度量将传播到包含数据对象和人工结构 `<Total>`（它被视为包含所有数据对象，其中包括 `<Unknown>` 和 `<Scalars>`）的任何聚集。`<Unknown>` 的不同子类型向上传播到 `<Unknown>` 聚集。下一节将介绍 `<Total>`、`<Scalars>` 和 `<Unknown>` 数据对象。

## 数据对象描述符

可以通过数据对象的声明类型和名称的组合来完整描述数据对象。简单的标量数据对象 `{int i}` 描述名为 `i`、类型为 `int` 的变量，而 `{const+pointer+int p}` 描述类型为 `int`、名为 `p` 的常量指针。类型名称中的空格将替换为下划线 (_)，未命名的数据对象用短划线 (-) 名称表示，例如，`{double_precision_complex -}`。

同样，对于 `foo_t` 类型的结构，将整个聚集表示为 `{structure:foo_t}`。聚集元素需要进一步指定其容器，例如，`{structure:foo_t}.{int i}` 表示上一结构为 `foo_t` 类型的 `int` 类型的成员 `i`。聚集本身也可以是（更大）聚集的元素，其对应描述符构造为聚集描述符的串联，并最终成为标量描述符。

虽然并不总是需要使用全限定描述符来消除数据对象的歧义，但是该描述符提供了完整的通用指定方式来协助标识数据对象。

### `<Total>` 数据对象

`<Total>` 数据对象是一个人工结构，用于将程序的数据对象作为一个整体表示。除了归属到不同数据对象（以及它所属的任何聚集）外，所有性能度量都归属到特殊的数据对

象 <Total>。该数据对象出现在数据对象列表的顶部，其数据可以用于了解其他数据对象的数据。

## <Scalars> 数据对象

聚集元素将其性能度量另外归属到其关联聚集的度量值，而所有标量常量和变量都将其性能度量另外归属到人工 <Scalars> 数据对象的度量值。

## <Unknown> 数据对象及其元素

在许多情况下，不能将事件数据映射到特定的数据对象。在这样的情况下，将数据映射到名为 <Unknown> 的特殊数据对象及其元素之一，如下所示：

- 带触发 PC 的模块未使用 -xhwcprof 编译

  由于对象代码未通过硬件计数器分析支持进行编译，因此未识别任何引发事件的指令或数据对象。
- 查找有效分支目标的回溯失败

  由于在编译对象中提供的硬件分析支持信息不足以验证回溯的有效性，因此未识别任何引发事件的指令。
- 回溯遍历到一个分支目标

  由于回溯遇到了指令流中的控制转移目标，因此未识别任何引发事件的指令或数据对象。
- 编译器未提供任何标识描述符

  编译器未提供内存引用指令的数据对象信息。
- 无类型信息

  编译器未将该指令识别为内存引用指令。
- 无法根据编译器提供的符号信息来确定

  编译器没有指令的符号信息。编译器临时文件通常是不可识别的。
- 跳转或调用指令阻止回溯

  由于回溯遇到了指令流中的分支或调用指令，因此未识别任何引发事件的指令。
- 回溯找不到触发 PC

  在最大回溯范围内未找到任何引发事件的指令。
- 无法确定 VA，因为寄存器在触发器指令后发生更改

  由于在硬件计数器失控期间寄存器被覆盖，因此未确定数据对象的虚拟地址。
- 内存引用指令未指定有效的 VA

  数据对象的虚拟地址看起来无效。

♦ ♦ ♦ 第 7 章

# 7 了解带注释的源代码和反汇编数据

带注释的源代码和带注释的反汇编代码可用于确定函数中哪些源代码行或指令造成性能低下，同时也可用于查看有关编译器如何在代码上执行转换的注释。本节介绍了注释过程，以及在解释带注释的代码时涉及的一些问题。

本章包含以下主题：



## 工具如何查找源代码

要显示带注释的源代码和带注释的反汇编代码，性能分析器和 `er_print` 实用程序对于运行实验的程序所使用的源代码文件和装入对象文件必须具有访问权限。

首先在实验的 `archives` 目录中查找装入对象文件。如果在该目录中未找到，则将使用与下面所述的源文件和对象文件相同的算法查找这些文件。

在大多数实验中，源文件和对象文件按照完整路径的格式记录。Java 源文件还具有一个软件包名称，其中列出文件的目录结构。如果在记录实验的同一系统上查看实验，则可以使用完整路径找到源文件和装入对象。当实验移到其他计算机或者在其他计算机上查看实验时，这些完整路径可能无法访问。

可以使用两个补充方法来查找源文件和对象文件：路径映射和搜索路径。如果在 `archives` 子目录中没有找到装入对象文件，可以使用相同的方法来查找这些文件。

可以设置路径映射和搜索路径，帮助工具查找实验中引用的文件。在性能分析器中，使用 "Settings"（设置）对话框的 "Pathmaps"（路径映射）标签设置路径映射，并使用 "Search Path"（搜索路径）标签设置搜索路径，如“配置设置” [113]中所述。对于 `er_print` 实用程序，使用 `pathmap` 和 `setpath` 指令，如“控制源文件搜索的命令” [135]中所述。

首先将应用路径映射，并指定如何将完整文件路径的开头替换为其他路径。例如，如果将文件指定为 `/a/b/c/sourcefile`，并且 `pathmap` 指令指定将 `/a/` 映射到 `/x/y/`，则可以在 `/x/y/b/c/sourcefile` 中找到文件。如果 `pathmap` 指令将 `/a/b/c/` 映射到 `/x/`，则可以在 `/x/sourcefile` 中找到文件。

如果通过路径映射找不到文件，则将使用搜索路径。搜索路径提供了一个要为具有指定基名的文件搜索的目录列表，在上面的示例中，指定的基名为 `sourcefile`。可以使用 `setpath` 命令设置搜索路径，并使用 `addpath` 命令向搜索路径附加一个目录。对于 Java 文件，将尝试软件包名称，然后再尝试基名。

使用搜索路径中的每个目录来构造尝试搜索的完整路径。对于 Java 源文件，将构造两个完整路径，一个用于基名，另一个用于软件包名称。工具会将路径映射应用于每个完整路径，如果没有映射路径指向文件，则将尝试下一个搜索路径目录。

如果在搜索路径中没有找到文件，并且没有映射前缀与原始完整路径匹配，则将尝试原始完整路径。如果有任何路径映射前缀与原始完整路径匹配，但没有找到文件，则不会尝试原始完整路径。

请注意，缺省搜索路径包含当前目录和实验目录，因此一个使源文件可访问的方法是将源文件复制到这些位置之一，或者在这些位置中放置指向源文件当前位置的符号链接。

# 带注释的源代码

可在性能分析器中查看实验的带注释的源代码，方法是在性能分析器窗口的左窗格中选择 "Source"（源）视图。另外，可以使用 `er_src` 实用程序，在不运行实验的情况下查看带注释的源代码。本手册的此部分介绍了源代码如何在性能分析器中显示。有关使用 `er_src` 实用程序查看带注释的源代码的详细信息，请参见“在不运行实验的情况下查看源代码/反汇编代码” [205]。

性能分析器中的带注释的源代码包含以下信息：

- 初始源文件的内容
- 每行可执行源代码的性能度量
- 由于度量超过特定阈值而突出显示的代码行
- 索引行
- 编译器注释

## 性能分析器 "Source"（源）视图布局

"Source"（源）视图分为若干列，其中左侧固定宽度的几列显示各个度量，右侧的其余部分显示带注释的源代码。

## 标识初始源代码行

在带注释的源代码中，所有以黑色显示的行都来自于初始源文件。在带注释的源代码列中，每行开头的数字与初始源文件中的行号对应。以不同颜色显示其中字符的行可能是索引行，也可能是编译器注释行。

## "Source"（源）视图中的索引行

源文件是指可编译生成对象文件或解释为字节代码的任何文件。对象文件通常包含与源代码中的函数、子例程或方法对应的一个或多个可执行代码区域。性能分析器分析对象文件，标识每个作为函数的可执行区域，并尝试将其在对象代码中找到的函数映射至与对象代码相关联的源文件中的函数、例程、子例程或方法。当性能分析器操作成功后，它将在带注释的源文件中对应于在对象代码中找到的函数的第一条指令处添加一个索引行。

带注释的源代码会为每个函数（包括内联函数，即使内联函数没有在 "Functions"（函数）视图中的列表中显示）显示一个索引行。在 "Source"（源）视图中以红色斜体显示索引行，且索引行中的文本括在尖括号中。类型最简单的索引行与函数的缺省上下文对应。任何函数的缺省源上下文都被定义为该函数的第一条指令所归属的源文件。以下示例显示了 C 函数 `icputime` 的索引行。

```
                  578. int
                  579. icputime(int k)
0.       0.       580. {
                       <Function: icputime>
```

从上面的示例能够看出，索引行显示在第一条指令后面的行中。对于 C 源代码，第一条指令对应于函数体开头的开型花括号处。在 Fortran 源代码中，各个子例程的索引行跟在包含 `subroutine` 关键字的行后面。同样，`main` 函数索引行跟在应用程序启动时执行的第一个 Fortran 源代码指令的后面，如以下示例所示：

```
                     1. ! Copyright (c) 2006, 2010, Oracle and/or its affiliates. All Rights
 Reserved.
                     2. ! @(#)omptest.f 1.11 10/03/24 SMI
                     3. ! Synthetic f90 program, used for testing openmp directives and the
                     4. !       analyzer
                     5.
0.    0.    0.    0.   6.        program omptest
                                      <Function: MAIN>
                                   7.
                                   8. !$PRAGMA C (gethrtime, gethrvtime)
```

有时，性能分析器可能无法将它在对象代码中找到的函数映射至与该对象代码相关联的源文件中的任何编程指令；例如，`#include`d 代码或从另一个文件（如某个头文件）内联的代码。

此外，以红色显示的行还有一些特殊索引行，以及其他一些不是编译器注释的特殊行。例如，由于编译器优化的原因，可能会为对象代码中的某个函数创建一个特

殊索引行，但该索引行并不对应在任何源文件中编写的代码。有关详细信息，请参阅“"Source"（源）、"Disassembly"（反汇编）和 "PCs"(PC) 标签中的特殊行” [200]。

## 编译器注释

编译器注释指示如何生成编译器优化代码。为了将编译器注释行同索引行及初始源代码行区分开，以蓝色显示编译器注释行。编译器的各个部分可将注释合并到可执行文件中。每个注释都与源代码的特定行相关联。当写入带注释的源代码后，任何源代码行的编译器注释会直接显示在源代码行的前面。

编译器注释描述为了优化源代码而对其进行的大量转换。这些转换包括循环优化、并行化、内联和流水线作业。以下示例显示了编译器注释。

```
0.    0.    0.    0.     28.       SUBROUTINE dgemv_g2 (transa, m, n, alpha, b, ldb,   &
                         29.      &                   c, incc, beta, a, inca)
                         30.       CHARACTER (KIND=1) :: transa
                                   31.       INTEGER   (KIND=4) :: m, n, incc, inca, ldb
                                   32.       REAL      (KIND=8) :: alpha, beta
                                   33.       REAL      (KIND=8) :: a(1:m), b(1:ldb,1:n), c(1:n)
                                   34.       INTEGER            :: i, j
                                   35.       REAL      (KIND=8) :: tmr, wtime, tmrend
                                   36.       COMMON/timer/ tmr
                                   37.
                             Function wtime_ not inlined because the compiler has not seen
                             the body of the routine
0.    0.    0.    0.     38.       tmrend = tmr + wtime()


                             Function wtime_ not inlined because the compiler has not seen
                             the body of the routine
                             Discovered loop below has tag L16
0.    0.    0.    0.     39.       DO WHILE(wtime() < tmrend)

                            Array statement below generated loop L4
0.    0.    0.    0.     40.       a(1:m) = 0.0
                                  41.

                            Source loop below has tag L6
0.    0.    0.    0.     42.       DO j = 1, n       ! <=-----\ swapped loop indices

                            Source loop below has tag L5
                               L5 cloned for unrolling-epilog.  Clone is L19
                               All 8 copies of L19 are fused together as part of unroll and jam
                               L19 scheduled with steady-state cycle count = 9
                               L19 unrolled 4 times
                               L19 has 9 loads, 1 stores, 8 prefetches, 8 FPadds,
                               8 FPmuls, and 0 FPdivs per iteration
                               L19 has 0 int-loads, 0 int-stores, 11 alu-ops, 0 muls,
                               0 int-divs and 0 shifts per iteration
                               L5 scheduled with steady-state cycle count = 2
                               L5 unrolled 4 times
```

```
                                L5 has 2 loads, 1 stores, 1 prefetches, 1 FPadds, 1 FPmuls,
                                and 0 FPdivs per iteration
                                L5 has 0 int-loads, 0 int-stores, 4 alu-ops, 0 muls,
                                0 int-divs and 0 shifts per iteration
0.210 0.210 0.210 0.     43.          DO i = 1, m
4.003 4.003 4.003 0.050  44.             a(i) = a(i) + b(i,j) * c(j)
0.240 0.240 0.240 0.     45.          END DO
0.    0.    0.    0.     46.       END DO
                                47.       END DO
                                48.
0.    0.    0.    0.     49.       RETURN
0.    0.    0.    0.     50.       END
```

可以使用 "Settings"（设置）对话框中的 "Source/Disassembly"（源/反汇编）标签来设置在 "Source"（源）视图中显示的编译器注释类型；有关详细信息，请参见“配置设置” [113]。

## 通用子表达式删除

一种很常见的优化是识别在多个位置出现的同一个表达式，并通过在一个位置生成该表达式的代码来提高性能。例如，如果在一个代码块的 `if` 和 `else` 分支同时出现了相同的操作，则编译器可以仅将该操作移到 `if` 语句前。同时，编译器将基于前面某次出现的该表达式为指令分配行号。如果分配给通用代码的行号与 `if` 结构的一个分支对应，并且该代码实际上始终包含另一个分支，则带注释的源代码会在不包含的分支内的行上显示度量。

## 循环优化

编译器可以执行多种类型的循环优化。部分比较常用的优化如下：

- 循环展开
- 循环剥离
- 循环交换
- 循环分裂
- 循环合并

循环展开是指在循环体中重复多次循环迭代，并相应调整循环索引的过程。随着循环体的增大，编译器能够更加有效地调度指令。同时降低由于循环索引递增和条件检查操作而引起的开销。循环的剩余部分则使用循环剥离进行处理。

循环剥离是指从循环中删除一定数量的循环迭代，并适当将它们移动至循环的前面或后面的过程。

循环交换可更改嵌套循环的顺序，以便最大程度地降低内存跨距 (memory stride)，最大程度地提高高速缓存行命中率。

循环合并是指将相邻或位置接近的循环合并为一个循环的过程。循环合并的优点与循环展开类似。此外，如果在两个预优化的循环中访问通用数据，则循环合并可以改进高速缓存定位 (cache locality)，从而为编译器提供更多利用指令级并行性的机会。

循环分裂与循环合并正好相反：它将一个循环分裂为两个或更多的循环。如果循环中的计算量过于庞大，导致寄存器溢出进而造成性能降级，则有必要采用这种优化。如果一个循环中包含多个条件语句，也可以使用循环分裂。有时可以将循环分成两类：带条件语句的和不带条件语句的。此方法可以在不带条件语句的循环中提高软件流水线作业的机会。

有时，对于嵌套式循环，编译器会先使用循环分裂来拆分循环，然后执行循环合并，以不同的方式将循环重新组合，以达到提高性能的目的。在这种情况下，您可以看到与以下示例类似的编译器注释：

```
Loop below fissioned into 2 loops
Loop below fused with loop on line 116
[116]    for (i=0;i<nvtxs;i++) {
```

## 函数内联

利用内联函数，编译器可将函数指令直接插在该函数的调用位置处，而不必执行真正的函数调用。这样，与 C/C++ 宏类似，内联函数的指令会在每个调用位置进行复制。编译器在高优化级别（4 和 5）执行显式内联或自动内联。

内联可以节约函数调用方面的开销，并提供可以优化寄存器使用和指令调度的更多指令，但代价是代码会占用内存中较多的资源。以下示例显示了内联编译器注释。

```
                Function initgraph inlined from source file ptralias.c
                    into the code for the following line
0.       0.         44.       initgraph(rows);
```

注 - 在性能分析器的 "Source"（源）视图中，编译器注释不会换行，即不会显示在两行上。

## 并行化

如果代码包含 Sun、Cray 或 OpenMP 并行化指令，则可经过编译在多个处理器上并行执行。编译器注释会指示执行过并行化操作和尚未执行并行化操作的位置，以及相应的原因。以下示例显示了并行化计算机注释。

```
0.       6.324       9. c$omp  parallel do shared(a,b,c,n) private(i,j,k)
                     Loop below parallelized by explicit user directive
                     Loop below interchanged with loop on line 12
0.010    0.010      [10]            do i = 2, n-1

                     Loop below not parallelized because it was nested in a parallel loop
                     Loop below interchanged with loop on line 12
```

```
0.170    0.170      11.              do j = 2, i
```

有关并行执行和编译器生成的主体函数的更多详细信息，请参阅“OpenMP 软件执行概述” [173]。

## 带注释的源代码中的特殊行

在 "Source"（源）视图中，还可以看到有关某些特殊情况的另外几种注释，它们或者显示为编译器注释，或者显示为与索引行颜色相同的特殊行。有关详细信息，请参阅“"Source"（源）、"Disassembly"（反汇编）和 "PCs"(PC) 标签中的特殊行” [200]。

## 源代码行度量

每行可执行代码的源代码度量都会在固定宽度的几列中显示出来。这些度量与函数列表中的度量相同。您可以使用 `.er.rc` 文件更改实验的缺省值。有关详细信息，请参见“在 `.er.rc` 文件中设置缺省值” [151]。还可以使用 "Settings"（设置）对话框更改性能分析器中显示的度量以及突出显示的阈值。有关详细信息，请参见“配置设置” [113]。

带注释的源代码在源代码行级别显示应用程序的度量。该度量是通过使用记录在应用程序调用堆栈中的 PC（program count，程序计数），并将每个 PC 映射至源代码行的方式生成的。要生成带注释的源文件，性能分析器首先确定在特定对象模块（`.o` 文件）或装入对象中生成的所有函数，然后从每个函数扫描所有 PC 的数据。

为了生成带注释的源代码，性能分析器必须能够找到并读取对象模块或装入对象来确定从 PC 至源代码行的映射。它必须能够读取源文件来生成要显示的带注释的副本。有关用于查找实验源代码的过程的说明，请参见“工具如何查找源代码” [189]。

编译过程要经过很多阶段，这取决于请求的优化级别，并且会发生转换，该转换会使指令到源代码行的映射变得混乱。对于某些优化，源代码行信息可能完全丢失，而对于其他优化，源代码行信息则可能变得混乱。编译器依赖各种试探操作来跟踪指令的源代码行，而这些试探操作不是绝对无误的。

### 解释源代码行度量

指令的度量必须解释为在等待执行该指令时产生的度量。如果记录事件时执行的指令来自与叶 PC 相同的源代码行，则度量可以解释为由于执行了该源代码行的缘故。但是，如果叶 PC 与所执行的指令来自不同的源代码行，那么叶 PC 所属源代码行的度量中至少有一些度量必须解释为在等待执行此代码行时累积的度量。例如，当一个源代码行上计算的值被用在下一个源代码行上时。

对于使用精确硬件计数器的硬件计数器溢出分析（如在 `collect -h` 的输出中所指示的那样），叶 PC 是导致计数器溢出的指令而不是要执行的下一个指令的 PC。对于非精确硬件计数器，报告的叶 PC 可能是导致溢出的指令之后的多个指令。这是因为用于识别溢出何时发生的内核机制有幅度可变的失控 (skid)。

当执行中存在长时间的延迟时（如遇到高速缓存未命中或资源队列停滞，或指令等待上一条指令的结果），如何解释度量变得至关重要。在这些情况下，源代码行的度量看上去高得不太合理。应该查看代码中的其他邻近行，以找出导致高度量值的行。

### 度量格式

表 7-1 “带注释的源代码度量” 中说明了可能出现在带注释的源代码行上的四种度量格式。

表 7-1 带注释的源代码度量

| 度量 | 显著性 |
| --- | --- |
| （空白） | 程序中没有 PC 对应于该代码行。此情形应该始终适用于注释行，并且在下列情况下适用于出现的代码行：<br>■ 所出现的代码段的所有指令已在优化期间删除。<br>■ 代码在其他地方重复，并且编译器执行通用子表达式识别并标记带有其他副本代码行的所有指令。<br>■ 编译器用不正确的行号来标记该行的指令。 |
| `0.` | 程序中的某些 PC 被标记为从该代码行派生，但没有数据引用这些 PC：它们从不会位于被抽样统计或被跟踪的调用堆栈中。度量不表示该行不执行，只是表示该行不以统计方式显示在分析数据包或记录的跟踪数据包中。 |
| `0.000` | 该行至少有一个 PC 出现在数据中，但是计算的度量值舍入为零。 |
| `1.234` | 归属于该行的所有 PC 的度量总计达到了显示的非零数值。 |

# 带注释的反汇编代码

带注释的反汇编代码提供了函数或对象模块指令的汇编代码列表，以及与每个指令相关联的性能度量。带注释的反汇编代码有多种显示方法，具体取决于行号映射和源文件是否可用，以及请求其带注释的反汇编代码的函数的对象模块是否已知。

- 如果对象模块未知，则性能分析器只对指定函数的指令进行反汇编，并且不会在反汇编代码中显示任何源代码行。
- 如果对象模块已知，则针对对象模块内的所有函数进行反汇编。
- 如果源文件可用，并且记录了行号数据，则性能分析器可以根据显示首选项交错显示源代码和反汇编代码。
- 如果编译器已向对象代码中插入注释，则它也可交错显示在反汇编代码中（如果已经设置了相应的首选项）。

反汇编代码中的每个指令都用以下信息进行注释：

- 由编译器报告的源代码行号
- 它的相对地址
- 指令的十六进制表示（如果要求）

- 指令的汇编程序 ASCII 表示

调用地址尽可能解析为符号（如函数名称）。度量显示在指令行上。如果设置了相应的首选项，度量还可显示在任何交错显示的源代码中。源代码注释的可能度量值如表 7-1 “带注释的源代码度量” 所述。

对于在多个位置包含在 `#include` 中的代码，每次当代码包含在 `#include` 中时，代码的反汇编代码列表都会重复执行一次反汇编指令。只有在文件中首次显示反汇编代码的重复块时，源代码才会交叉。例如，如果在名为 `inc_body.h` 的头文件中定义的代码块分别由 `inc_body`、`inc_entry`、`inc_middle` 和 `inc_exit` 四个函数 `#included`，则反汇编指令块将在 `inc_body.h` 的反汇编代码列表中出现四次，但是源代码仅在四个反汇编指令块的第一个块中交错显示。切换到 "Source"（源）视图，可以看到与每次重复的反汇编代码对应的索引行。

可以在 "Disassembly"（反汇编）视图中显示索引行。与 "Source"（源）视图不同的是，不能直接使用这些索引行进行导航。将光标放在紧挨该索引行下方的其中一条指令上，然后选择 "Source"（源）视图，可以导航至该索引行中引用的文件。

对其他文件中的代码执行了 `#include` 操作的文件会将通过该操作包含的代码显示为不与源代码交错显示的原始反汇编指令。将光标放在这些指令中的其中一条指令上，然后选择 "Source"（源）视图，将打开包含 `#included` 代码的文件。在显示此文件时选择 "Disassembly"（反汇编）视图，将显示与源代码交错显示的反汇编代码。

内联函数的源代码可以与反汇编代码交错显示，但宏的源代码则不能与反汇编代码交错显示。

代码未优化时，每个指令的行号按顺序显示，源代码行和反汇编指令的交错显示以预期的方式进行。优化后，后面行的指令有时会显示在前面行的指令前面。分析器的交错显示算法为：只要指令显示为来自第 *N* 行，该行前所有源代码行（包括第 *N* 行）都会写入该指令前。优化的一个作用是源代码可以在控制转移指令与其延迟槽指令之间显示。与源代码的第 *N* 行关联的编译器注释就写在该行之前。

# 解释带注释的反汇编代码

解释带注释的反汇编代码并不是一项简单的工作。叶 PC 是要执行的下一条指令的地址，因此归属到某个指令的度量应该视为等待执行该指令所用的时间。不过，指令的执行并不总是按顺序发生，而且在记录调用堆栈时可能发生延迟。要利用带注释的反汇编代码，应该熟悉记录实验的硬件以及装入和执行指令的方式。

接下来的几个小节将讨论有关解释带注释的反汇编代码的一些问题。

## 指令发送分组

指令按组装入和发送（称为指令发送组）。组中包括哪些指令取决于硬件、指令类型、已执行的指令以及与其他指令或寄存器的各种相关性。因此，某些指令可能会被误解，

因为它们始终与前一条指令在同一个时钟周期内发送，所以它们从不单独表示下一条要执行的指令。在记录调用堆栈时，可能有多条指令被视为是下一条要执行的指令。

指令发送规则因处理器类型的不同而不同，并且还取决于高速缓存行内的指令对齐。由于链接程序比高速缓存行要求执行更精确的指令对齐，因此在看上去不相关的函数中进行的更改可能会导致指令的对齐方式不同。不同的对齐方式会引起性能的改进或降级。

下面列举了一种假设情况，即同一个函数在略微不同的环境下进行编译和链接的情况。以下两个输出示例均为 `er_print` 实用程序的带注释的反汇编代码列表。两个示例中的指令是相同的，但指令的对齐方式不同。

在以下输出示例中，指令的对齐方式是将 `cmp` 和 `bl,a` 两条指令映射到不同的高速缓存行。大量的时间都用于等待执行这两条指令。

```
   Excl.     Incl.
User CPU  User CPU
    sec.      sec.
                            1. static int
                            2. ifunc()
                            3. {
                            4.     int i;
                            5.
                            6.     for (i=0; i<10000; i++)
                               <function: ifunc>
   0.010     0.010              [ 6]    1066c:  clr         %o0
   0.        0.                 [ 6]    10670:  sethi       %hi(0x2400), %o5
   0.        0.                 [ 6]    10674:  inc         784, %o5
                            7.         i++;
   0.        0.                 [ 7]    10678:  inc         2, %o0
## 1.360     1.360              [ 7]    1067c:  cmp         %o0, %o5
## 1.510     1.510              [ 7]    10680:  bl,a        0x1067c
   0.        0.                 [ 7]    10684:  inc         2, %o0
   0.        0.                 [ 7]    10688:  retl
   0.        0.                 [ 7]    1068c:  nop
                            8.     return i;
                            9. }
```

在下一个输出示例中，指令的对齐方式是将 `cmp` 和 `bl,a` 两条指令映射到相同的高速缓存行。大量的时间都用于等待执行其中一条指令。

```
   Excl.     Incl.
User CPU  User CPU
    sec.      sec.
                            1. static int
                            2. ifunc()
                            3. {
                            4.     int i;
                            5.
                            6.     for (i=0; i<10000; i++)
                               <function: ifunc>
   0.        0.                 [ 6]    10684:  clr         %o0
   0.        0.                 [ 6]    10688:  sethi       %hi(0x2400), %o5
   0.        0.                 [ 6]    1068c:  inc         784, %o5
```

```
                              7.          i++;
   0.        0.                  [ 7]    10690:  inc         2, %o0
## 1.440     1.440               [ 7]    10694:  cmp         %o0, %o5
   0.        0.                  [ 7]    10698:  bl,a        0x10694
   0.        0.                  [ 7]    1069c:  inc         2, %o0
   0.        0.                  [ 7]    106a0:  retl
   0.        0.                  [ 7]    106a4:  nop
                              8.      return i;
                              9. }
```

## 指令发送延迟

有时，会更频繁地出现特定的叶 PC，因为这些 PC 表示的指令在发送前被延迟。导致出现这种情况的原因有多种，下面列出了其中的一些原因：

- 执行前一条指令用了很长的时间，并且执行不能中断，例如指令陷入内核中时。
- 运算指令需要的寄存器不可用，因为该寄存器的内容是由前面尚未完成的指令设置的。具有数据高速缓存未命中的装入指令就是这种延迟的一个示例。
- 浮点运算指令正在等待另一个浮点指令完成。指令不能流水线作业时会出现这种情况，例如平方根和浮点除法。
- 指令高速缓存不存在包含指令的内存字（指令高速缓存未命中）。

## 硬件计数器溢出的归属

除某些平台上的 TLB 未命中和 `precise` 计数器外，硬件计数器溢出事件的调用堆栈按指令的顺序记录在后续的某些点上，而不是记录在发生溢出的点上这种延迟出于多种原因，其中一个是处理由溢出产生的中断需要一定时间。对于某些计数器（如发送的周期或指令），这种延迟无关紧要。对于其他计数器（例如，那些计数高速缓存未命中或浮点运算的计数器），度量被归属到导致溢出的指令之外的其他指令。

通常，引起事件的 PC 只是记录的 PC 前的几条指令，而这些指令可以在反汇编代码列表中正确定位。但是，如果该指令范围内存在分支目标，那么要确定哪条指令对应于引起事件的 PC 是很困难的，甚至是不可能的。

如果系统中的处理器具有使用 `precise` 关键字标记的计数器，则允许进行内存空间分析，而不需要任何特殊的二进制文件编译。例如，SPARC T2、SPARC T3 和 SPARC T4 处理器提供多个 `precise` 计数器。运行 `collect -h` 命令并查找 `precise` 关键字，可确定系统是否允许内存空间分析。

例如，在配备 SPARC T4 处理器的系统上运行以下命令时，将会显示可用的精确原始计数器：

```
% collect -h | & grep -i precise | grep -v alias
    Instr_ld[/{0|1|2|3}],1000003 (precise load-store events)
    Instr_st[/{0|1|2|3}],1000003 (precise load-store events)
    SW_prefetch[/{0|1|2|3}],1000003 (precise load-store events)
```

```
Block_ld_st[/{0|1|2|3}],1000003 (precise load-store events)
DC_miss_L2_L3_hit_nospec[/{0|1|2|3}],1000003 (precise load-store events)
DC_miss_local_hit_nospec[/{0|1|2|3}],1000003 (precise load-store events)
DC_miss_remote_L3_hit_nospec[/{0|1|2|3}],1000003 (precise load-store events)
DC_miss_nospec[/{0|1|2|3}],1000003 (precise load-store events)
```

# "Source"（源）、"Disassembly"（反汇编）和 "PCs"(PC) 标签中的特殊行

性能分析器在 "Source"（源）、"Disassembly"（反编译）和 "PCs" (PC) 视图中显示的某些行并不直接对应于代码行、指令行或程序计数器行。以下部分介绍了这些特殊的行。

## 外联函数

外联函数可以在反馈优化编译期间创建。在 "Source"（源）视图和 "Disassembly"（反编译）视图中，外联函数显示为特殊的索引行。在 "Source"（源）视图中，在已转换为外联函数的代码块中会显示一条注释。

```
                  Function binsearchmod inlined from source file ptralias2.c into the
0.       0 .         58.         if( binsearchmod( asize, &element ) ) {
0.240    0.240       59.             if( key != (element << 1) ) {
0.       0.          60.                 error |= BINSEARCHMODPOSTESTFAILED;
                        <Function: main -- outline code from line 60 [_$o1B60.main]>
0.040    0.040    [ 61]                 break;
0.       0.          62.             }
0.       0.          63.         }
```

在 "Disassembly"（反编译）视图中，外联函数通常在文件结尾处显示。

```
                        <Function: main -- outline code from line 85 [_$o1D85.main]>
0.       0.          [ 85] 100001034:  sethi       %hi(0x100000), %i5
0.       0.          [ 86] 100001038:  bset        4, %i3
0.       0.          [ 85] 10000103c:  or          %i5, 1, %l7
0.       0.          [ 85] 100001040:  sllx        %l7, 12, %l5
0.       0.          [ 85] 100001044:  call        printf ! 0x100101300
0.       0.          [ 85] 100001048:  add         %l5, 336, %o0
0.       0.          [ 90] 10000104c:  cmp         %i3, 0
0.       0.          [ 20] 100001050:  ba,a        0x1000010b4
                        <Function: main -- outline code from line 46 [_$o1A46.main]>
0.       0.          [ 46] 100001054:  mov         1, %i3
0.       0.          [ 47] 100001058:  ba          0x100001090
0.       0.          [ 56] 10000105c:  clr         [%i2]
                        <Function: main -- outline code from line 60 [_$o1B60.main]>
0.       0.          [ 60] 100001060:  bset        2, %i3
0.       0.          [ 61] 100001064:  ba          0x10000109c
0.       0.          [ 74] 100001068:  mov         1, %o3
```

外联函数的名称显示在方括号中，并对外联代码段的有关信息进行编码，这些信息包括从中提取代码的函数名称以及源代码中该段的起始行号。这些改编名称 (mangled name) 可能随发行版的不同而不同。性能分析器提供了函数名称的可读版本。有关更多详细信息，请参阅“外联函数” [182]。

如果在收集应用程序的性能数据时调用了外联函数，则性能分析器会在带注释的反汇编代码中显示一个特殊行，以显示该函数的非独占度量。有关更多详细信息，请参见“非独占度量” [205]。

# 编译器生成的主体函数

编译器并行化函数中的循环或具有并行化指令的区域时，将会创建初始源代码中不存在的新主体函数。“OpenMP 软件执行概述” [173]中介绍了这些函数。

编译器将改编名称分配给主体函数，这些主体函数对并行结构的类型、从中提取结构的函数名称、初始源代码中该结构的起始行号以及并行结构的序列号进行了编码。这些改编名称 (mangled name) 因微任务化库的发行版而异，但均会取消改编 (demangle) 为更易于理解的名称。

以下示例显示的是一个由编译器生成的典型主体函数，如计算机模式下的函数列表中所示。

```
7.415      14.860      psec_ -- OMP sections from line 9 [_$s1A9.psec_]
3.873       3.903      craydo_ -- MP doall from line 10 [_$d1A10.craydo_]
```

在示例中，最先显示的是从中提取结构的函数名称，接着是并行结构的类型，然后是并行结构的行号，最后在方括号中显示的是由编译器生成的主体函数的改编名称 (mangled name)。类似地，在反汇编代码中，也会生成一个特殊索引行。

```
0.       0.            <Function: psec_ -- OMP sections from line 9 [_$s1A9.psec_]>
0.       7.445         [24]    1d8cc:  save        %sp, -168, %sp
0.       0.            [24]    1d8d0:  ld          [%i0], %g1
0.       0.            [24]    1d8d4:  tst         %i1

0.       0.            <Function: craydo_ -- MP doall from line 10 [_$d1A10.craydo_]>
0.       0.030         [ ?]    197e8:  save        %sp, -128, %sp
0.       0.            [ ?]    197ec:  ld          [%i0 + 20], %i5
0.       0.            [ ?]    197f0:  st          %i1, [%sp + 112]
0.       0.            [ ?]    197f4:  ld          [%i5], %i3
```

对于 Cray 指令，函数可能与源代码行号不相关。在这种情况下，会在行号的位置显示 `[ ?]`。如果索引行显示在带注释的源代码中，则该索引行指示不带行号的指令，如以下示例中所示。

```
                  9. c$mic  doall shared(a,b,c,n) private(i,j,k)

                Loop below fused with loop on line 23
                Loop below not parallelized because autoparallelization
                  is not enabled
                Loop below autoparallelized
```

```
                  Loop below interchanged with loop on line 12
                  Loop below interchanged with loop on line 12
3.873     3.903         <Function: craydo_ -- MP doall from line 10 [_$d1A10.craydo_],
                     instructions without line numbers>
0.        3.903     10.            do i = 2, n-1
```

注 - 在实际显示中，索引行和编译器注释行并不换行。

# 动态编译的函数

动态编译的函数是指程序执行时编译和链接的函数。收集器中并没有有关用 C 或 C++ 编写的动态编译函数的信息，除非用户使用收集器 API 函数 `collector_func_load()` 提供所需的信息。由 "Function"（函数）视图、"Source"（源）视图和 "Disassembly"（反汇编）视图显示的信息取决于传递给 `collector_func_load()` 的信息，如下所示：

- 如果未提供信息，则不会调用 `collector_func_load()`。在函数列表中，动态编译和装入的函数会显示为 `<Unknown>`。性能分析器中既看不到函数源代码，也看不到反汇编代码。
- 如果没有提供源文件名和行号表，但提供了函数的名称、大小以及地址，则函数列表中将显示动态编译和装入的函数的名称及度量。带注释的源代码是可用的，并且反汇编指令是可见的，只是行号被指定为 `[?]`，这表示行号未知。
- 如果提供了源文件名，但未提供行号表，性能分析器显示的信息将与不提供源文件名所显示的信息类似，只是带注释的源代码开头将显示一个特殊索引行，以指示该函数由不带行号的指令组成。例如：

  ```
  1.121     1.121          <Function func0, instructions without line numbers>
                        1. #include        <stdio.h>
  ```

- 如果提供了源文件名和行号表，将按照与常规编译函数相同的方式在 "Function"（函数）视图、"Source"（源）视图和 "Disassembly"（反汇编）视图中显示函数及其度量。

有关收集器 API 函数的更多信息，请参见“动态函数和模块” [46]。

对于 Java 程序，大部分方法由 JVM 软件解释。在解释执行期间，在单独的线程上运行的 Java HotSpot 虚拟机会监视性能。在监视过程中，虚拟机可能决定使用已解释的一个或多个方法，生成相应的计算机代码，并执行更有效的计算机代码版本，而不是解释原始代码。

对于 Java 程序，无需使用收集器 API 函数。性能分析器通过在该方法的索引行下方使用一个特殊行，表明在带注释的反汇编代码列表中存在 JavaHotSpot 编译的代码，如以下示例所示。

```
                  11.    public int add_int () {
                  12.       int       x = 0;
2.832     2.832      <Function: Routine.add_int: HotSpot-compiled leaf instructions>
0.        0.         [ 12] 00000000: iconst_0
0.        0.         [ 12] 00000001: istore_1
```

反汇编代码列表仅显示已解释的字节代码，而不显示编译的指令。缺省情况下，在该特殊行的旁边显示已编译代码的度量。该独占和非独占 CPU 时间与各行已解释字节代码的所有独占和非独占 CPU 时间总和不同。通常，如果多次调用该方法，已编译指令的 CPU 时间将大于已解释字节代码的 CPU 时间总和，这种差异之所以出现，是因为已解释代码只是在最初调用该方法时执行一次，而已编译代码则在其后执行。

带注释的源代码不显示 Java HotSpot 编译的函数。而是显示一个特殊索引行，以指示不带行号的指令。例如，下面显示了与上例显示的反汇编提取对应的带注释的源代码：

```
                    11.    public int add_int () {
2.832     2.832       <Function: Routine.add_int(), instructions without line numbers>
0.        0.        12.       int       x = 0;
                      <Function: Routine.add_int()>
```

# Java 本机函数

本机代码是最初用 C、C++ 或 Fortran 编写，由 Java 代码通过 Java 本地接口 (Java Native Interface, JNI) 调用的已编译代码。以下示例来自与演示程序 `jsynprog` 关联的文件 `jsynprog.java` 的带注释的反汇编代码。

```
                    5. class jsynprog
                       <Function: jsynprog.<init>()>
0.       5.504          jsynprog.JavaCC() <Java native method>
0.       1.431          jsynprog.JavaCJava(int) <Java native method>
0.       5.684          jsynprog.JavaJavaC(int) <Java native method>
0.       0.             [  5] 00000000: aload_0
0.       0.             [  5] 00000001: invokespecial <init>()
0.       0.             [  5] 00000004: return
```

由于本机方法不包含在 Java 源代码中，`jsynprog.java` 的带注释的源代码的开头会显示每个 Java 本机方法，并使用一个特殊的索引行来指示不带行号的指令。

```
0.       5.504          <Function: jsynprog.JavaCC(), instructions without line
                           numbers>
0.       1.431          <Function: jsynprog.JavaCJava(int), instructions without line
                           numbers>
0.       5.684          <Function: jsynprog.JavaJavaC(int), instructions without line
                           numbers>
```

注 - 在实际的带注释的源代码显示中，索引行并不换行。

# 克隆函数

编译器能够识别可以对其执行额外优化的函数调用，例如所传递的部分参数是常量的函数调用。当编译器识别出它可以优化的特定调用时，会创建该函数的副本（称为克隆）并生成优化代码。

在带注释的源代码中，编译器注释将指示是否创建了克隆函数：

```
0.       0.       Function foo from source file clone.c cloned,
                   creating cloned function _$c1A.foo;
                   constant parameters propagated to clone
0.       0.570     27.    foo(100, 50, a, a+50, b);
```

注 - 在实际的带注释的源代码显示中，编译器注释行并不换行。

克隆函数名称是标识特定调用的改编名称 (mangled name)。在前面的示例中，编译器注释指示克隆的函数名称为 `_$c1A.foo`。在函数列表中，该函数显示为：

```
0.350     0.550     foo
0.340     0.570     _$c1A.foo
```

每个克隆函数都具有不同的指令集，因此带注释的反汇编列表单独显示克隆函数。这些函数与任何源文件都没有关联，因此其指令也与任何源代码行号无关。以下示例显示了某个克隆函数的带注释的反汇编代码的前几行。

```
0.       0.           <Function: _$c1A.foo>
0.       0.           [?]    10e98:  save        %sp, -120, %sp
0.       0.           [?]    10e9c:  sethi       %hi(0x10c00), %i4
0.       0.           [?]    10ea0:  mov         100, %i3
0.       0.           [?]    10ea4:  st          %i3, [%i0]
0.       0.           [?]    10ea8:  ldd         [%i4 + 640], %f8
```

# 静态函数

静态函数通常在库中使用，因此库内部使用的名称不会与用户可能使用的名称发生冲突。库被剥离后，静态函数的名称将从符号表中删除。在这种情况下，性能分析器会为包含剥离静态函数的库中的每个文本区域生成人工名称。该名称的格式为 `<static>@0x12345`，其中 `@` 符号后的字符串是库中文本区域的偏离量。性能分析器无法区分连续的剥离静态函数和单个这样的函数，因此可能出现两个或多个这样的函数，且其度量合并在一起。以下示例显示 `jsynprog` 样例演示的函数列表中的静态函数。

```
0.       0.       <static>@0x18780
0.       0.       <static>@0x20cc
0.       0.       <static>@0xc9f0
0.       0.       <static>@0xd1d8
0.       0.       <static>@0xe204
```

在 "PCs" (PC) 视图中，这些函数使用如下所示的偏移量表示：

```
0.       0.       <static>@0x18780 + 0x00000818
0.       0.       <static>@0x20cc + 0x0000032C
0.       0.       <static>@0xc9f0 + 0x00000060
0.       0.       <static>@0xd1d8 + 0x00000040
0.       0.       <static>@0xe204 + 0x00000170
```

在 "PCs" (PC) 视图中，在被剥离的库内调用的函数的替代表示方法为：

```
<library.so> -- no functions found + 0x0000F870
```

## 非独占度量

在带注释的反汇编代码中，有一些特殊的行标记外联函数占用的时间。

以下示例显示了调用外联函数时显示的带注释的反汇编代码：

```
0.       0.        43.         else
0.       0.        44.         {
0.       0.        45.                 printf("else reached\n");
0.       2.522         <inclusive metrics for outlined functions>
```

## 存储和装入指令的注释

利用 -xhwcprof 选项进行编译时，编译器将为存储 (st) 和装入 (ld) 指令生成附加信息。可以在反汇编代码列表中查看带注释的 st 和 ld 指令。

## 分支目标

使用 -xhwcprof 选项进行编译时，带注释的反汇编代码列表中显示的人工行 <branch target> 与指令的某个 PC 对应，可从多个代码路径到达该指令。

# 在不运行实验的情况下查看源代码/反汇编代码

使用 er_src 实用程序就可以在不运行实验的情况下查看带注释的源代码和带注释的反汇编代码。显示的生成方式与在性能分析器中的显示生成方式相同，只是不显示任何度量。er_src 命令的语法如下所示：

er_src [ -func | -{source,src} *item tag* | -{disasm,dis} *item tag* | -{cc,scc,dcc} *com-spec* | -outfile *filename* | -V ] *object*

*object* 是可执行文件、共享对象或对象文件（.o 文件）的名称。

*item* 是用于生成可执行文件或共享对象的函数、源文件或对象文件的名称。*item* 还可以采用 *function*' *file*'的格式指定，在这种情况下，er_src 将在指定文件的源上下文中显示指定函数的源代码或反汇编代码。

*tag* 是索引，用于决定当多个函数具有相同的名称时引用哪个 *item*。该选项是必需选项，但如果没有必要解析函数，则可以忽略该选项。

特殊的项和标记 `all -1` 指示 `er_src` 为该对象中的所有函数生成带注释的源代码或反汇编代码。

---

注 - 在可执行文件和共享对象上使用 `all -1` 生成的输出可能非常大。

---

以下几节介绍 `er_src` 实用程序可以接受的选项。

## -func

列出给定对象的所有函数。

### -{source,src} *item tag*

显示列出的项的带注释的源代码。

### -{disasm,dis} *item tag*

在列表中包括反汇编代码。缺省列表不包括反汇编代码。如果没有源代码可用，则产生一个不带编译器注释的反汇编代码列表。

### -{cc,scc,dcc} *com-spec*

指定要显示哪些编译器注释类。*com-spec* 是用冒号分隔的类列表。如果使用的是 *-scc* 选项，对源代码编译器注释应用 `com-spec` 类列表；如果使用的是 `-dcc` 选项，对反汇编代码注释应用类列表；如果使用的是 `-cc` 选项，既对源代码注释，也对反汇编代码注释应用类列表。有关这些类的描述，请参见“控制源代码和反汇编代码列表的命令” [132]。

注释类可以在缺省值文件中指定。首先读取系统范围的 `er.rc` 缺省值文件，然后读取用户起始目录中的 `.er.rc` 文件（如果存在），最后读取当前目录中的 `.er.rc` 文件。您的起始目录中的 `.er.rc` 文件的缺省值覆盖系统缺省值，当前目录中的 `.er.rc` 文件的缺省值覆盖起始目录缺省值和系统缺省值。这些文件也由性能分析器和 `er_print` 实用程序使用，但只有源代码和反汇编代码编译器注释的设置由 `er_src` 实用程序使用。有关对缺省值文件的描述，请参见“在 .er.rc 文件中设置缺省值” [151]。缺省值文件中除 `scc` 和 `dcc` 之外的命令均被 `er_src` 实用程序忽略。

### -outfile *filename*

打开显示列表输出的文件 *filename*。缺省情况下，或者如果文件名为短划线 (-)，则输出会写入 `stdout`。

**-V**

显示当前发行版本。

♦ ♦ ♦ 第 8 章

# 操作实验

本章介绍可以与收集器和性能分析器一起使用的实用程序。
本章包含以下主题：



# 操作实验

实验存储在收集器创建的目录中。要操作实验，您可以使用常用的 UNIX® 命令 `cp`、`mv` 和 `rm`，并将它们应用到该目录。对于早于 Forte Developer 7 (Sun™ ONE Studio 7, Enterprise Edition for Solaris) 发行版的实验，则不能这样做。提供了三个功能类似于 UNIX 命令的实用程序，以复制、移动和删除实验。这些实用程序为 `er_cp`(1)、`er_mv`(1) 和 `er_rm`(1)，将在下文中介绍这些实用程序。

实验中的数据包括程序使用的每个装入对象的归档文件。这些归档文件包含装入对象的绝对路径及其最后一次修改的日期。移动或复制实验时，该信息不会改变。

## 使用 er_cp 实用程序复制实验

存在两种格式的 `er_cp` 命令：

`er_cp [-V]` *experiment1 experiment2*
`er_cp [-V]` *experiment-list directory*

第一种格式的 `er_cp` 命令将 *experiment1* 复制到 *experiment2*。如果 *experiment2* 已存在，则 `er_cp` 将退出，并显示一条错误消息。第二种格式将空格分隔的实验列表复制到一个目录。如果该目录中已经包含与正在复制的实验之一同名的实验，则 `er_cp` 实用程序将退出，并显示一条错误消息。`-V` 选项可输出 `er_cp` 实用程序的版本。此命令不能复制使用早期版本的工具创建的实验。

## 使用 er_mv 实用程序移动实验

存在两种格式的 er_mv 命令：

er_mv [-V] *experiment1 experiment2*
er_mv [-V] *experiment-list directory*

第一种格式的 er_mv 命令将 *experiment1* 移动到 *experiment2*。如果 *experiment2* 已存在，则 er_mv 实用程序将退出，并显示一条错误消息。第二种格式将空格分隔的实验列表移动到一个目录。如果该目录中已经包含与正被移动的实验之一同名的实验，则 er_mv 实用程序将退出，并显示一条错误消息。-V 选项可输出 er_mv 实用程序的版本。此命令不能移动使用早期版本的工具创建的实验。

## 使用 er_rm 实用程序删除实验

er_rm 实用程序删除实验或实验组的列表。删除实验组后，组中的每个实验以及组文件都被删除。

er_rm 命令的语法如下：

er_rm [-f] [-V] *experiment-list*

无论是否找到实验，-f 选项都会禁止错误消息并确保成功完成。-V 选项可输出 er_rm 实用程序的版本。此命令可删除使用早期发行版的工具创建的实验。

# 为实验加标签

er_label 命令可以定义实验的一部分并为其分配名称或添加标签。标签捕获在实验中以开始时间和停止时间标记定义的一个或多个时间段内发生的分析事件。

可以将时间标记指定为当前时间、当前时间加上或减去某个时间偏移，或者指定为相对于实验开始时间的某个偏移。可以在标签中指定任何数量的时间间隔，并且在创建标签后，可以向标签添加其他间隔。

对于 er_label 实用程序，应该使用成对标记指定间隔：开始时间后面跟随停止时间。实用程序忽略出现顺序错误的标记，例如在任何开始标记前指定的停止标记，在上一个开始标记后面没有跟随任何停止标记而直接跟随的开始标记，或者在上一个停止标记后面没有跟随任何开始标记而直接跟随的停止标记。

您可通过在命令行运行 er_label 命令或在脚本中执行该命令来为实验分配标签。向实验添加标签后，便可以使用标签进行过滤。例如，您可能会对实验进行过滤，包含或排除在标签定义的时间段中的分析事件，如“使用标签进行过滤” [109]中所述。

注 - 不应创建与可在过滤中使用的任何其他关键字相同的标签名称，因为这样会导致冲突和意外的结果。可以使用 `er_print -describe` 命令查看实验的关键字。

## `er_label` 命令语法

`er_label` 命令的语法为：

`er_label -o` *experiment-name* `-n` *label-name* `-t {start|stop}[=`*time-specification*`] [-C` *comment*

选项定义如下：

`-o` 实验名称。这是一个必需的选项，用于指定希望标记的实验的名称。只能指定一个实验名称，不支持实验组。`-o` 选项可以出现在命令行中的任意位置。

`-n` 标签名称。这是一个必需的选项，用于指定标签名称。

标签名称可以是任意长度，但必须是字母数字，以字母开头且没有嵌入的空格，即使用引号将字符串括起也是如此。如果 *label-name* 存在，会向它添加新条件；如果不存在，将创建它。需要一个 `-n` 参数，但是该参数可出现在命令行上的任意位置。标签名不区分大小写。标签名称不得与可以出现在过滤条件中的其他名称（包括实验中的属性或者内存对象或索引对象的名称）冲突。已装入实验中的属性是使用 `er_print describe` 命令列出的。内存对象是使用 `er_print mobj_list` 命令列出的。索引对象是使用 `er_print indxobj_list` 命令列出的。

`-C` 注释。这是标签的注释，可选。可以对单个标签使用多个 `-C` 选项，在显示标签时，这些注释将串联起来，相互之间使用分号和空格分隔。可以使用多个注释，例如，在标签中提供每个时间间隔的信息。

`-t start|stop` *=time-specification* 是为定义实验中的时间范围所指定的开始点或停止点。如果省略 *=time-specification*，则创建当前时间的标记。

可以按以下格式之一指定 *time-specification*：

*hh:mm:ss.uuu*
: 指定相对于实验开始的时间，在此处应该放置开始或停止标记。必须至少指定秒数，还可以选择指定小时数、分钟数和零点几秒。

  指定的时间值将按以下方式解释：

  nn
  : 如果指定不带冒号的整数，则将解释为秒数。如果值大于 60，则秒数在标签中将转换为 mm:ss。例如，`-t start=120` 会在实验开始后的 02:00 处放置一个开始标记。

  nn.nn
  : 如果包含任意精度的小数，则该值将解释为零点几秒，并且以纳秒精度进行保存。例如，`-t`

`start=120.3` 会在实验开始后的 02:00.300（即 2 分 300 纳秒）处放置一个开始标记。

nn:nn
: 如果使用 nn:nn 格式指定时间，则将解释为 mm:ss；如果 mm 的值大于 60，则时间将转换为 hh:mm:ss。为 ss 指定的数字必须介于 0 和 59 之间，否则会出现错误。例如，`-t start=90:30` 会在实验开始后的 01:30:30（即 1 小时 30 分 30 秒）处放置一个开始标记。

nn:nn:nn
: 如果使用 nn:nn:nn 格式指定时间，则将解释为 hh:mm:ss。为分钟和秒指定的数字必须介于 0 和 59 之间，否则会出现错误。例如，`-t stop=01:45:10` 会在实验开始后的 1 小时 45 分 10 秒处放置一个结束标记。

`@`
: 指定当前时间，以便在实验中执行 `er_label` 命令的那一刻放置一个标记。当前时间在命令的单个调用中设置一次，因此使用 `@` 的任何其他标记将相对于该原始时间戳值设置。

`@+`*offset*
: 指定当前时间戳之后的某个时间，其中 *offset* 是一个使用与上面所述的相同 *hh:mm:ss.uuu* 规则的时间。此时间格式在原始时间戳之后的指定时间处放置一个标记。例如，`-t stop=@+180` 在当前时间之后的 3 分钟处放置一个停止标记。

`@-`*offset*
: 指定当前时间戳之前的某个时间，其中 *offset* 是一个使用与上面所述的相同 *hh:mm:ss.uuu* 规则的时间。此时间格式在原始时间戳之前的指定时间处放置一个标记。例如，`-t start=@-20:00` 在当前时间之前的 20 分钟处放置一个开始标记。如果实验的运行时间尚未达到 20 分钟，则将忽略该标记。

可以在单个 `er_label` 命令中指定多个 `-t`，也可以在相同标签名称的单独命令中指定多个 `-t`，但它们应该以成对的 `-t start` 和 `-t stop` 标记出现。

如果 `-t start` 或 `-t stop` 选项后面未跟随任何时间指定，则将在指定部分中采用 `=@`。必须为其中一个标记包含时间指定。

## `er_label` 示例

**例 8-1** 使用相对于实验开始的时间标记定义标签

要在实验 `test.1.er` 中定义名为 `snap` 的标签，其中涵盖从实验开始之后的 15 秒起持续时间为 10 分钟的运行部分，请使用以下命令：

```
% er_label -o test.1.er -n snap -t start=15 -t stop=10:15
```

此外，可以在单独的命令中为间隔指定标记：

```
% er_label -o test.1.er -n snap -t start=15
% er_label -o test.1.er -n snap -t stop=10:15
```

例 8-2 使用相对于当前时间的时间标记定义标签

要在实验 `test.1.er` 中定义名为 `last5mins` 的标签，其中涵盖当前时间之前的 5 分钟起的运行部分，请执行以下操作：

```
% er_label -o test.1.er -n last5mins -t start=@-05:00 -t stop
```

# 在脚本中使用 er_label

`er_label` 的一个用途是支持将由客户端驱动的服务器程序作为一个独立的进程或多个进程进行分析。在这种使用模型中，使用 `collect` 命令启动服务器，以便开始在服务器上创建实验。服务器启动并准备好接受客户端请求后，您便可运行客户端脚本，以请求驱动服务器并运行 `er_label` 来标记发生客户端请求的实验部分。

以下样例客户机脚本在 `test.1.er` 实验中为针对服务器的每个请求运行生成一个时间标签。创建的五个标签中的每个标签都划分出处理指定请求所花费的时间。

```
for REQ in req1 req2 req3 req4 req5
          do

          echo "============================================================"
          echo " $REQ started at `date`"

          er_label -o test.1.er -n $REQ -t start=@
          run_request $REQ
          er_label -o test.1.er -n $REQ -t stop=@
          done
```

以下样例脚本显示了一个备选用法，该脚本生成名为 `all` 的包含所有请求的单个标签。

```
for REQ in req1 req2 req3 req4 req5
          do

          echo "============================================================"
          echo " $REQ started at `date`"

          er_label -o test.1.er -n all -t start=@
          run_request $REQ
          er_label -o test.1.er -n all -t stop
          done
```

请注意，在第二个 er_label 调用中的 -t stop 后面未跟随时间指定，因此缺省采用 stop=@。

可以创建更复杂的脚本，并且可以在相同节点或不同节点上同时运行多个脚本。如果实验位于所有节点都可访问的共享目录中，则脚本可以在相同的实验中标记间隔。各个脚本中的标签可以是相同的，也可以是不同的。

# 其他实用程序

在正常情况下，不必使用其他一些实用程序。在此说明这些程序是为了完整性，并对可能需要使用这些实用程序的情况进行了描述。

## er_archive 实用程序

er_archive 命令的语法如下。

```
er_archive [ -nqF ] [-s option [-m regexp ] ] experiment er_archive -V
```

当实验正常完成时，er_archive 将自动运行（除非在运行实验时关闭了归档功能）。它会读取实验中引用的共享对象的列表，并将实验中引用的所有共享对象复制到创建者实验中的目录 *archives* 中。副本以改编名称存储，所以创建者实验及其子实验可能有同名对象的不同版本。对于 Java 实验，引用的所有 .jar 文件也会复制到实验中。（er_archive 不再生成 *.archive* 文件。）如果目标程序异常终止，er_archive 可能无法运行；在这种情况下，应该手动运行它。

如果用户希望在与记录实验所在的计算机不同的其他计算机上检查实验，则 er_archive 必须在实验完成时已运行，或者必须对记录数据所在的计算机上的实验手动运行，才能获得所有共享对象的正确版本。

如果无法找到共享对象，或者如果其时间戳与实验中记录的不同，又或者如果 er_archive 运行所在的计算机与记录实验所在的计算机不同，则不会复制对象。如果手动运行 er_archive（不带 -q 标志），则会将警告写入 stderr。当 er_print 或分析器读取引用此类共享对象的实验时，也会生成警告。

通过在手动运行时指定 -s 参数，er_archive 也可归档源代码（包括任何必要的对象文件与所需的符号表）。通过设置环境变量 SP_ARCHIVE_ARGS，可将源代码的归档指定为在实验完成时自动执行。该环境变量可能包含 -s 和 -m 参数，形式为成对的参数和由一个或多个空格分隔的选项。如果命令行上出现多个 -s 参数，则优先使用最后一个。如果 -s 既在命令行上传递，又由环境变量设置，则优先使用环境变量的选项。

以下几节介绍 er_archive 实用程序可以接受的选项。

**–n**

仅归档指定的实验，不包括其子孙。

**–q**

不将任何警告写入 stderr。警告将并入归档文件，并显示在性能分析器或 er_print 实用程序的输出中。

**–F**

强制写入或重新写入归档文件。该参数可用于手动运行 er_archive，以重新写入带有警告的文件。

**–s** *option*

指定源文件的归档。*option* 的允许值包括：

| | |
|---|---|
| no | 不归档任何源文件 |
| all | 归档可找到的所有源代码、对象和 .anc 文件。 |
| used[src] | 归档在实验中记录数据所针对的函数的且可找到的源代码、对象和 .anc 文件。<br>如果在命令行上提供了或者在环境变量中指定了多个 -s 参数，为所有参数指定的选项必须相同。如果不同，er_archive 将退出，并显示一个错误。 |

**–m** *regex*

仅归档由 -s 标志指定且在可执行文件或共享对象中记录的其完整路径名与指定的 *regex* 匹配的源代码、对象和 .anc 文件。有关更多信息，请参见 regex(5) 手册页。

可以在命令行上或环境变量中提供多个 -m 参数。如果源文件与任一参数的表达式匹配，则会归档源文件。

**–V**

写入 er_archive 实用程序的版本号信息，并退出。

# er_export 实用程序

er_export 命令的语法如下。

er_export [-V] *experiment*

er_export 实用程序将实验中的原始数据转换为 ASCII 文本。文件的格式和内容可以更改，任何使用都不应该依赖这种格式和内容。仅当性能分析器无法读取实验时才使用该实用程序。工具开发者可利用输出内容了解原始数据并分析故障。–V 选项用于输出版本号信息。

♦ ♦ ♦ 第 9 章

# 9 内核分析

本章介绍如何在 Oracle Solaris 运行负载时使用 Oracle Solaris Studio 性能工具分析内核。如果在 Oracle Solaris 10 或 Oracle Solaris 11 上运行 Oracle Solaris Studio 软件，可以进行内核分析。在 Linux 系统上不能进行内核分析。

本章包含以下主题：



## 内核实验

可以使用性能分析器或使用 `er_kernel` 实用程序记录内核分析数据。在性能分析器中分析内核时，还可以在后台运行 `er_kernel` 实用程序。

`er_kernel` 实用程序使用的 DTrace 是 Oracle Solaris 操作系统内置的一款综合动态跟踪工具。

`er_kernel` 实用程序捕获内核分析数据，并将数据记录为性能分析器实验，其格式与使用 `collect` 记录的用户实验相同。实验可以由 `er_print` 实用程序或性能分析器进行处理。内核实验可以显示函数数据、调用方-被调用方数据、指令级数据和时间线，但是不能显示源代码行数据，因为大多数 Oracle Solaris 模块不包含行号表。

## 为内核分析设置系统

您需要先设置对 DTrace 的访问，才能使用 `er_kernel` 实用程序进行内核分析。

通常，DTrace 仅限用户 `root` 使用。要以 `root` 之外的用户身份运行 `er_kernel` 实用程序，必须分配有特定特权，而且是组 `sys` 的成员。要分配必要的特权，请在文件 `/etc/user_attr` 中按以下方式编辑包含用户名的行：

*username*`::::defaultpriv=basic,dtrace_kernel,dtrace_proc`

要将您自己添加到组 sys，请将您的用户名添加到文件 /etc/group 中的 sys 行。

# 运行 er_kernel 实用程序

您可以运行 er_kernel 实用程序，以便仅分析内核或同时分析内核和正在运行的负载。有关完整说明，请参见 er_kernel(1) 手册页。

要显示用法消息，请运行不带参数的 er_kernel 命令。

缺省情况下将使用 er_kernel 选项 -p on，因此不需要明确指定它。

您可以将 er_kernel 实用程序的 -p on 参数替换为 -p high（用于高精度分析）或 -p low（用于低精度分析）。如果期望用 2 到 20 分钟来运行负载，则缺省时钟分析是合适的。如果期望用不到 2 分钟的时间来运行，请使用 -p high；如果期望用 20 分钟以上的时间来运行，请使用 -p low。

您可以添加 -t *duration* 参数，该参数将导致 er_kernel 实用程序按 *duration* 所指定的时间自行终止。

可以将 -t *duration* 指定为一个具有可选的 m 或 s 后缀的数字，以指示实验应终止的时间（以分钟或秒为单位）。缺省情况下，持续时间以秒为单位。也可以将 *duration* 指定为用连字符分隔的两个这样的数字，这会导致数据收集暂停，直到经过第一个时间之后才开始收集数据。当到达第二个时间时，数据收集终止。如果第二个数字为零，则在初次暂停之后收集数据，直到该程序运行结束。即使该实验已经终止，也允许目标进程运行至结束。

如果未指定任何持续时间或时间间隔，在终止之前，er_kernel 将一直运行。可以通过按 Ctrl-C (SIGINT)，或者使用 kill 命令并将 SIGINT、SIGQUIT 或 SIGTERM 发送到 er_kernel 进程终止该实用程序。将其中任何一个信号发送到 er_kernel 时，该进程将终止实验并运行 er_archive（除非指定了 -A off）。er_archive 实用程序会读取实验中引用的共享对象列表，并为每个共享对象构造一个归档文件。

可使用 -x 选项排除空闲 CPU 的分析事件，该选项缺省设置为 on，因此不会记录这类事件。可以设置 -x off 以记录空闲 CPU 的分析事件，以便完整地包括所有 CPU 时间。

如果您希望在屏幕上输出有关运行的更多信息，可以添加 -v 参数。通过 -n 参数，您可以预览将被记录的实验，而无需实际记录任何内容。

缺省情况下，由 er_kernel 实用程序生成的实验被命名为 ktest.1.er；对于相继的运行，该数字将递增。

## ▼ 使用 er_kernel 分析内核

1. **通过键入以下内容来收集实验：**

```
% er_kernel -p on
```

2. 在单独的 shell 中运行负载。

3. 负载完成后，按 Ctrl-C 来终止 **er_kernel** 实用程序。

4. 将生成的实验（缺省情况下名为 **ktest.1.er**）装入到性能分析器或 **er_print** 实用程序中。

   内核时钟分析将生成两个度量：一个是 KCPU 周期（度量名称为 kcycles），该度量用于在内核创建者实验中记录的时钟分析事件；另一个是 KUCPU 周期（度量名称为 kucycles），该度量用于当 CPU 处于用户模式时在用户进程子实验中记录的时钟分析事件。

   在性能分析器中，对于 "Functions"（函数）视图中的内核函数、"Callers-Callees"（调用方-被调用方）视图中的调用方与被调用方以及 "Disassembly"（反汇编）视图中的指令，将显示这些度量。"Source"（源）视图不显示数据，因为附带的内核模块通常不包含文件和行符号表信息 (stabs)。

## ▼ 使用 er_kernel 分析负载不足

如果有要用作负载的单个命令（程序或脚本）：

1. 通过键入以下内容来收集实验：

   ```
   % er_kernel -p on load
   ```

   如果负载是一个脚本，则该脚本在退出之前，应等待它所产生的任何命令终止，否则实验可能会过早终止。

   er_kernel 实用程序派生一个子进程，并暂停一个静止期。然后子进程会运行指定的负载。在负载终止时，er_kernel 实用程序再次暂停一个静止期，然后退出。

   您可以使用 er_kernel 命令的 -q 参数，以秒为单位指定静止期的持续时间。

2. 通过键入以下内容来分析实验：

   ```
   % analyzer ktest.1.er
   ```

   实验显示运行负载期间以及之前和之后的静止期内 Oracle Solaris 内核的行为。有关可在内核分析中查看的内容的更多信息，请参见“分析内核分析数据” [222]。

## 硬件计数器溢出的内核分析

er_kernel 实用程序可以使用 DTrace cpc 提供者（仅在运行 Oracle Solaris 11 的系统上可用）为内核收集硬件计数器溢出分析。

就如使用 collect 命令一样，可以使用 er_kernel 命令的 -h 选项来执行内核的硬件计数器溢出分析。

与 collect 一样，如果未显式指定任何 -p off 参数，缺省情况下将启用基于时钟的分析。如果指定 -h high 或 -h low 以高频率或低频率请求该芯片的缺省计数器集，则缺省的时钟分析也将设置为高频率或低频率；将遵从显式 -p 参数。

er_kernel -h 命令使用 DTrace cpc 提供程序收集硬件计数器溢出分析。硬件计数器分析在 Oracle Solaris 11 之前的系统上不可用。如果芯片上的溢出机制使内核能够指明哪个计数器发生溢出，则可以使用芯片提供的任意多个计数器；否则，只能指定一个计数器。

数据空间分析在运行 DTrace 1.8 或更高版本的 SPARC 系统上受支持，且仅适用于精确计数器。如果在不支持数据空间分析的系统上请求，将忽略数据空间标志，但是实验仍会运行。

系统硬件计数器机制可以由多个进程用于执行用户分析，但如果任何用户进程、cputrack 或其他 er_kernel 正在使用该机制，则该机制不能用于内核分析。在这种情况下，er_kernel 将报告 "HW counters are temporarily unavailable; they may be in use for system profiling"（硬件计数器暂时不可用，它们可能正用于进行系统分析）。

要显示处理器支持硬件计数器溢出分析的计算机上的硬件计数器，请运行不带任何其他参数的 er_kernel —h 命令。

如果芯片上的溢出机制使内核能够指明哪个计数器发生溢出，则可以分析芯片提供的任意多个计数器；否则只能指定一个计数器。er_kernel -h 输出通过显示消息来指定您是否可以使用多个计数器，例如，"specify HW counter profiling for up to 4 HW counters"（为多达 4 个 HW 计数器指定 HW 计数器分析）。

有关硬件计数器分析的更多信息，请参见“硬件计数器分析数据” [23]和“使用 collect -h 收集硬件计数器分析数据” [55]。

另请参见 er_print 手册页，了解硬件计数器溢出分析的更多信息。

## 分析内核和用户进程

使用 er_kernel 实用程序可以执行内核和应用程序的分析。可以使用 -F 选项控制是否应跟踪应用程序进程并记录这些进程的数据。

使用 -F on 或 -F all 选项时，er_kernel 将记录所有应用程序进程以及内核上的实验。将跟踪在收集 er_kernel 实验时检测到的用户进程，并且为每个跟踪的进程创建一个子实验。

如果您以非 root 用户的身份运行 er_kernel，则可能不会记录许多子实验，因为不享有权限的用户通常无法读取有关其他用户进程的任何内容。

假定有足够的特权，那么用户进程数据也只在进程处于用户模式时才记录，并且只记录用户调用堆栈。每个跟踪的进程的子实验包含 kucycles 度量的数据。子实验使用 *_process-name*_PID_*process-pid*.1.er 格式命名。例如，某个在 sshd 进程上运行的实验的名称可能为 _sshd_PID_1264.1.er。

要只跟踪某些用户进程，可以使用 -F =*regexp* 指定正则表达式，以便记录名称或 PID 与正则表达式匹配的进程上的实验。

例如，er_kernel -F =synprog 将跟踪名为 synprog 的程序的进程。

请注意，对于 er_kernel 从 /proc 文件系统读取的进程名称，操作系统会将其截断，最多留下 15 个字符（外加零字节）。应该指定模式与这样截断的进程名称匹配。

有关正则表达式的信息，请参见 regexp(5) 手册页。

缺省情况下设置 -F off 选项，这样 er_kernel 不会执行用户进程分析。

---

注 - er_kernel 的 -F 选项不同于 collect 的 -F 选项。collect -F 命令用于只跟踪那些由命令行中指定的目标创建的进程；而 er_kernel -F 用于跟踪当前在系统上运行的所有进程。

---

# 一起分析内核和负载的替代方法

除了使用 er_kernel -F=*regexp* 以外，如果使用 collect *load* 而非 *load* 的目标运行 er_kernel，则可以一起分析内核和负载。在指定 collect 和 er_kernel 时只有其中一个可包括硬件计数器。如果 er_kernel 正在使用硬件计数器，则 collect 命令无法使用。

这种技术的优点是，它可以在用户进程未在 CPU 上运行时收集其中的数据，由 er_kernel 收集的用户实验只包括用户 CPU 时间和系统 CPU 时间。另外，在使用 collect 时，可以在用户模式下获得 OpenMP 和 Java 分析的数据。使用 er_kernel 时，上述任一分析只能获得计算机模式的分析结果，并且没有关于 Java 环境中 HotSpot 编译的任何信息。

## ▼ 一起分析内核和负载

1. **通过键入 er_kernel 命令和 collect 命令，同时收集内核分析数据和用户分析数据：**

```
% er_kernel collect load
```

2. 通过键入以下内容一起分析这两个分析数据：

```
% analyzer ktest.1.er test.1.er
```

性能分析器显示的数据同时显示 ktest.1.er 中的内核分析数据和 test.1.er 中的用户分析数据。使用 "Timeline"（时间线）视图可以查看两个实验之间的相关性。

---

注 - 要将脚本用作负载，并单独分析脚本的各个部分，请在脚本内的各个命令前放置 collect 命令（带有适当的参数）。

---

# 分析内核分析数据

内核创建者实验包含 kcycles 度量的数据。当 CPU 处于系统模式时，将记录内核调用堆栈。当 CPU 空闲时，将记录人工函数 <IDLE> 的单帧调用堆栈。当 CPU 处于用户模式时，将记录归属于人工函数 *<process-name*_PID_*process-pid>* 的单帧调用堆栈。在内核实验中，不记录用户进程的任何调用堆栈信息。

内核创建者实验中的人工函数 <INCONSISTENT_PID> 指示由于未知原因而具有不一致进程 ID 的 DTrace 事件传送到的位置。

如果使用 -F 来指定跟踪用户进程，则每个跟踪的进程的子实验将包含 kucycles 度量的数据。当进程在用户模式下运行时，将为所有时钟分析事件记录用户级别调用堆栈。

可以在 "Processes"（进程）视图和 "Timeline"（时间线）视图中使用过滤器，以便限定您关注的那些 PID。

# 索引

## 数字和符号

## D

## E

## I



## J



## K

## Y

## Z