Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

文件号码 820–3242
2007 年 9 月

版权所有 2007 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 U.S.A. 保留所有权利。

对于本文档中介绍的产品，Sun Microsystems, Inc. 对其所涉及的技术拥有相关的知识产权。需特别指出的是（但不局限于此），这些知识产权可能包含一项或多项美国专利，以及在美国和其他国家/地区申请的待批专利。

美国政府权利－商业软件。政府用户应遵循 Sun Microsystems, Inc. 的标准许可协议，以及 FAR（Federal Acquisition Regulations，即“联邦政府采购法规”）的适用条款及其补充条款。

本发行版可能包含由第三方开发的内容。

本产品的某些部分可能是从 Berkeley BSD 系统衍生出来的，并获得了加利福尼亚大学的许可。UNIX 是 X/Open Company, Ltd. 在美国和其他国家/地区独家许可的注册商标。

Sun、Sun Microsystems、Sun 徽标、Solaris 徽标、Java 咖啡杯徽标、docs.sun.com、 Java 和 Solaris 是 Sun Microsystems, Inc. 在美国和其他国家/地区的商标或注册商标。所有 SPARC 商标的使用均已获得许可，它们是 SPARC International, Inc. 在美国和其他国家/地区的商标或注册商标。标有 SPARC 商标的产品均基于由 Sun Microsystems, Inc. 开发的体系结构。

OPEN LOOK 和 Sun 图形用户界面是 Sun Microsystems, Inc. 为其用户和许可证持有者开发的。Sun 感谢 Xerox 在研究和开发可视或图形用户界面的概念方面为计算机行业所做的开拓性贡献。Sun 已从 Xerox 获得了对 Xerox 图形用户界面的非独占性许可证，该许可证还适用于实现 OPEN LOOK GUI 和在其他方面遵守 Sun 书面许可协议的 Sun 许可证持有者。

本出版物所介绍的产品以及所包含的信息受美国出口控制法制约，并应遵守其他国家/地区的进出口法律。严禁将本产品直接或间接地用于核设施、导弹、生化武器或海上核设施，也不能直接或间接地出口给核设施、导弹、生化武器或海上核设施的最终用户。严禁出口或转口到美国禁运的国家/地区以及美国禁止出口清单中所包含的实体，包括但不限于被禁止的个人以及特别指定的国家/地区的公民。

本文档按“原样”提供，对于所有明示或默示的条件、陈述和担保，包括对适销性、适用性或非侵权性的默示保证，均不承担任何责任，除非此免责声明的适用范围在法律上无效。

071119@19028

目录

前言 13

1 性能分析器概述 19
从集成开发环境中启动性能分析器 19
性能分析工具 19
收集器工具 20
性能分析器工具 20
er_print 实用程序 21
tcov 实用程序 21
性能分析器窗口 21
传统的 UNIX 性能工具 22

2 性能数据 23
收集器收集何种数据 23
时钟数据 24
硬件计数器溢出分析数据 26
同步等待跟踪数据 28
堆跟踪（内存分配）数据 29
MPI 跟踪数据 29
全局（抽样）数据 31
如何将度量分配到程序结构 32
函数级度量：独占、包含和归属 32
解释归属度量：示例 33
递归如何影响函数级度量 35

3 收集性能数据 37
编译和链接程序 37

- 源代码信息 ... 37
- 静态链接 ... 38
- 编译时优化 ... 38
- 编译 Java 程序 ... 38

为数据收集和分析准备程序 ... 38

- 使用动态分配的内存 ... 39
- 使用系统库 ... 40
- 使用信号处理程序 ... 40
- 使用 `setuid` ... 41
- 数据收集的程序控制 ... 41
- C、C++、Fortran 和 Java API 函数 ... 43
- 动态函数和模块 ... 44

数据收集的限制 ... 45

- 基于时钟的分析的限制 ... 45
- 收集跟踪数据的限制 ... 46
- 硬件计数器溢出分析的限制 ... 46
- 硬件计数器溢出分析中的运行时失真和扩大 ... 47
- 后续进程中数据收集的限制 ... 47
- Java 分析的限制 ... 47
- 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 ... 48

数据的存储位置 ... 48

- 实验名称 ... 48
- 移动实验 ... 49

估计存储要求 ... 50

收集数据 ... 51

使用 `collect` 命令收集数据 ... 51

- 数据收集选项 ... 52
- 实验控制选项 ... 55
- 输出选项 ... 58
- 其他选项 ... 59
- 使用 `collect` 实用程序从正在运行的进程中收集数据 ... 60
- ▼ 使用 `collect` 实用程序从正在运行的进程中收集数据 ... 60

使用 `dbx collector` 子命令收集数据 ... 60

- ▼ 从 `dbx` 运行收集器： ... 60
- 数据收集子命令 ... 61
- 实验控制子命令 ... 64

输出子命令 64
信息子命令 65
使用 dbx 从正在运行的进程中收集数据 66
▼ 从正在运行且不受 dbx 控制的进程中收集数据： 66
从正在运行的程序中收集跟踪数据 67
从 MPI 程序收集数据 68
存储 MPI 实验 68
在 MPI 下运行 collect 命令 70
通过在 MPI 下启动 dbx 来收集数据 70
将 collect 和 ppgsz 一起使用 71

4 性能分析器工具 73
启动性能分析器 73
分析器选项 74
分析器缺省设置 75
性能分析器 GUI 75
菜单栏 75
工具栏 76
分析器数据显示 76
设置数据表示选项 84
保存数据表示选项 87
查找文本和数据 87
显示或隐藏函数 87
过滤数据 88
实验选择 88
样本选择 88
线程选择 88
LWP 选择 88
CPU 选择 89
记录实验 89
生成映射文件和函数重新排序 89
分析器缺省设置 90

5 内核分析 91
内核实验 91

为内核分析设置系统 91
运行 er_kernel 实用程序 92
▼ 分析内核 92
▼ 在有负载时的分析 93
▼ 一起分析内核和负载 93
分析特定的进程或内核线程 93
分析内核分析数据 94

6 er_print 命令行性能分析工具 95
er_print 语法 96
度量列表 96
控制函数列表的命令 99
functions 99
metrics *metric_spec* 100
sort *metric_spec* 101
fsummary 101
fsingle *function_name* [*N*] 101
控制调用者-被调用者列表的命令 102
callers-callees 102
cmetrics *metric_spec* 102
csingle *function_name* [*N*] 103
csort *metric_spec* 103
控制泄漏和分配列表的命令 104
leaks 104
allocs 104
控制源代码和反汇编代码列表的命令 104
pcs 104
psummary 104
lines 104
lsummary 104
source { *filename* | *function_name* } [*N*] 105
disasm { *filename* | *function_name* } [*N*] 105
scc *com_spec* 105
sthresh *value* 106
dcc *com_spec* 106

`dthresh` *value* ... 107
`cc` *com_spec* ... 107
`setpath` *path_list* ... 107
`addpath` *path_list* ... 107
`pathmap` *old-prefix new-prefix* ... 108
控制数据空间列表的命令 ... 108
`data_objects` ... 108
`data_single` *name* [*N*] ... 108
`data_layout` ... 108
`data_metrics` *metric_spec* ... 108
`data_sort` ... 109
控制内存对象列表的命令 ... 109
`memobj` *mobj_type* ... 109
`mobj_list` ... 109
`mobj_define` *mobj_type index_exp* ... 110
控制索引对象列表的命令 ... 110
`indxobj` *indxobj_type* ... 110
`indxobj_list` ... 110
`indxobj_define` *indxobj_type index_exp* ... 110
`indxobj_metrics` *metric_spec* ... 111
`indxobj_sort` *metric_spec* ... 111
支持线程分析器的命令 ... 111
`races` ... 111
`rdetail` *race_id* ... 111
`deadlocks` ... 111
`ddetail` *deadlock_id* ... 112
列出实验、抽样、线程和 LWP 的命令 ... 112
`experiment_list` ... 112
`sample_list` ... 112
`lwp_list` ... 112
`thread_list` ... 112
`cpu_list` ... 113
控制实验数据过滤的命令 ... 113
指定过滤表达式 ... 113
选择要进行过滤的抽样、线程、LWP 和 CPU ... 113
控制装入对象展开和折叠的命令 ... 114

object_list 114
object_select *object1,object2,...* 115
列出度量的命令 115
metric_list 115
cmetric_list 116
data_metric_list 116
indx_metric_list 116
控制输出的命令 116
outfile { *filename* | - } 116
appendfile *filename* 116
limit *n* 117
name { long | short } [:{ *shared_object_name* | *no_shared_object_name* }]" 117
viewmode { user | expert | machine } 117
列显其他信息的命令 118
header *exp_id* 118
ifreq 118
objects 118
overview *exp_id* 118
statistics *exp_id* 118
设置缺省值的命令 119
dmetrics *metric_spec* 119
dsort *metric_spec* 119
en_desc { on | off | =*regexp*} 120
仅为性能分析器设置缺省值的命令 120
tabs *tab_spec* 120
rtabs *tab_spec* 120
tlmode *tl_mode* 120
tldata *tl_data* 121
杂项命令 121
mapfile *load-object* { *mapfilename* | - } 121
procstats 122
script *file* 122
version 122
quit 122
help 122
表达式语法 122

er_print 命令示例 123

7 **了解性能分析器及其数据** 127
数据收集的工作原理 127
实验格式 127
记录实验 129
解释性能度量 130
基于时钟的分析 131
同步等待跟踪 133
硬件计数器溢出分析 133
堆跟踪 134
数据空间分析 134
MPI 跟踪 134
调用栈和程序执行 135
单线程执行和函数调用 135
显式多线程 138
基于 Java 技术的软件执行概述 138
Java 处理表示法 140
OpenMP 软件执行概述 141
不完全的堆栈展开 149
将地址映射到程序结构 150
进程映像 151
装入对象和函数 151
有别名的函数 151
非唯一函数名称 152
来自剥离共享库的静态函数 152
Fortran 备用入口点 152
克隆函数 153
内联函数 153
编译器生成的主体函数 154
外联函数 154
动态编译的函数 154
<Unknown> 函数 155
OpenMP 特殊函数 155
<JVM-System> 函数 156

<no Java callstack recorded> 函数 156
<Truncated-stack> 函数 156
<Total> 函数 156
与硬件计数器溢出分析相关的函数 156
将性能数据映射到索引对象 157
将数据地址映射到程序数据对象 157
数据对象描述符 158
将性能数据映射到内存对象 159

8 了解带注释的源代码和反汇编数据 161
带注释的源代码 161
性能分析器“源”标签布局 161
带注释的反汇编代码 167
解释带注释的反汇编代码 169
“源”、“反汇编”和 "PC" 标签中的特殊行 171
外联函数 171
编译器生成的主体函数 172
动态编译的函数 173
Java 本机函数 174
克隆函数 175
静态函数 175
包含度量 176
分支目标 176
在不运行实验的情况下查看源代码/反汇编代码 177
-func 177

9 处理实验 179
处理实验 179
使用 er_cp 实用程序复制实验 179
使用 er_mv 实用程序移动实验 180
使用 er_rm 实用程序删除实验 180
其他实用程序 180
er_archive 实用程序 180
er_export 实用程序 181

索引 183

前言

本手册介绍 Sun™ Studio 12 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执行大范围性能数据的统计分析以及跟踪各种系统调用，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。

本手册适用于具有 Fortran、C、C++ 或 Java™ 编程语言使用经验的应用程序开发者。使用性能工具的用户需要对 Solaris™ 操作系统 (Solaris Operating System, Solaris OS) 或 Linux 操作系统以及 UNIX® 操作系统命令有一定的了解。掌握一些性能分析知识有助于运用这些工具，但这并不是必须的。

本书的结构

第 1 章介绍性能分析工具，其中简要讨论了这些工具的用途以及何时使用这些工具。

第 2 章介绍收集器收集的数据以及如何将这些数据转换为性能度量。

第 3 章介绍如何使用收集器从程序中收集计时数据、同步延迟数据和硬件事件数据。

第 4 章介绍如何启动性能分析器和如何使用该工具来分析收集器收集的性能数据。

第 5 章介绍如何在 Solaris OS 运行负载时使用 Sun Studio 性能工具来分析内核。

第 6 章介绍如何使用 `er_print` 命令行界面分析收集器收集的数据。

第 7 章介绍将收集器收集的数据转换为性能度量的过程，以及如何将这些度量关联到程序结构。

第 8 章介绍如何使用和了解性能分析器的源代码和反汇编窗口中的信息。

第 9 章介绍关于实用程序的信息，无需运行实验，这些实用程序就可以处理和转换性能实验并查看带注释的源代码和反汇编代码。

印刷约定

下表介绍了本书中的印刷约定。

表 P–1 印刷约定

字体	含义	示例
`AaBbCc123`	命令、文件和目录的名称；计算机屏幕输出	编辑 `.login` 文件。 使用 `ls -a` 列出所有文件。 `machine_name% you have mail.`
`AaBbCc123`	用户键入的内容，与计算机屏幕输出的显示不同	`machine_name%` **`su`** `Password:`
aabbcc123	要使用实名或值替换的命令行占位符	删除文件的命令为 `rm` *filename*。
AaBbCc123	保留未译的新词或术语以及要强调的词	这些称为 *Class* 选项。 **注意**：有些强调的项目在联机时以粗体显示。
新词术语强调	新词或术语以及要强调的词	**高速缓存**是存储在本地的副本。 请**勿**保存文件。
《书名》	书名	阅读《用户指南》的第 6 章。

命令中的 shell 提示符示例

下表列出了 C shell、Bourne shell 和 Korn shell 的缺省 UNIX 系统提示符和超级用户提示符。

表 P–2 Shell 提示符

Shell	提示符
C shell	`machine_name%`
C shell 超级用户	`machine_name#`
Bourne shell 和 Korn shell	`$`
Bourne shell 和 Korn shell 超级用户	`#`

受支持的平台

此 Sun Studio 发行版支持使用 SPARC® 和 x86 系列处理器体系结构的系统：UltraSPARC®、SPARC64、AMD64、Pentium 和 Xeon EM64T。通过访问 http://www.sun.com/bigadmin/hcl 中的硬件兼容性列表，可以了解您在使用的 Solaris 操作系统版本所支持的系统。这些文档列举了在不同类型的平台上进行实现时的所有差别。

在本文档中，与 x86 相关的术语的含义如下：

- "x86" 泛指 64 位和 32 位的 x86 兼容产品系列。
- "x64" 表示有关 AMD64 或 EM64T 系统的特定 64 位信息。
- “32 位 x86”表示有关基于 x86 的系统的特定 32 位信息。

有关受支持的系统，请参阅硬件兼容性列表。

访问 Sun Studio 文档

可以访问以下位置的文档：

- 可以通过随软件一起安装在本地系统或网络中的文档索引获取文档，位置为 Solaris 平台上的 `file:/opt/SUNWspro/docs/index.html` 和 Linux 平台上的 `file:/opt/sun/sunstudio12/docs/index.html`。

  如果未将软件安装在 Solaris 平台上的 `/opt` 目录中或 Linux 平台上的 `/opt/sun` 目录中，请询问系统管理员以获取系统中的等效路径。
- 也可以通过 Sun Studio 门户网站联机获取文档索引，网址为 http://developers.sun.com/sunstudio/documentation/ss12。
- 可以通过 Sun Studio 门户网站获取最新的发行说明，网址为 http://developers.sun.com/sunstudio/documentation/ss12/release_notes.html。
- IDE 所有组件的联机帮助可通过 IDE 中的“帮助”菜单以及许多窗口和对话框上的“帮助”按钮获取。

您可以通过 Internet 访问 http://docs.sun.com/ Web 站点，以阅读、打印和购买 Sun Microsystems 的各种手册。如果找不到某手册，请参见随软件一起安装在本地系统或网络上的文档索引。

注 – Sun 对本文档中提到的第三方 Web 站点的可用性不承担任何责任。对于此类站点或资源中的（或通过它们获得的）任何内容、广告、产品或其他资料，Sun 并不表示认可，也不承担任何责任。对于因使用或依靠此类站点或资源中的（或通过它们获得的）任何内容、产品或服务而造成的或连带产生的实际或名义损坏或损失，Sun 概不负责，也不承担任何责任。

访问 Solaris 相关文档

下表介绍了可通过 docs.sun.com Web 站点获取的相关文档。

文档集合	文档标题	说明
Solaris 10 Reference Manual Collection	请参见手册页各章节的标题。	提供 Solaris 操作系统的有关信息。
Solaris 10 Software Developer Collection - Simplified Chinese	《Linker and Libraries Guide》	介绍了 Solaris 链接编辑器和运行时链接程序的操作。
Solaris 10 Software Developer Collection - Simplified Chinese	《Multithreaded Programming Guide》	涵盖 POSIX® 和 Solaris 线程 API、使用同步对象编程、编译多线程程序以及多线程程序的查找工具。
Solaris 10 Software Developer Collection - Simplified Chinese	《SPARC Assembly Language Reference Manual》	介绍用于 SPARC 处理器的汇编语言。
Solaris 10 System Administrator Collection - Simplified_Chinese	《Solaris Tunable Parameters Reference Manual》	提供有关 Solaris 可调参数的参考信息。

开发者资源

访问 `http://developers.sun.com/sunstudio` 以查找以下经常更新的资源：

- 有关编程技术和最佳做法的文章
- 有关编程小技巧的知识库
- 编译器和工具组件文档以及随软件一起安装的文档的更正信息
- 有关支持级别的信息
- 用户论坛
- 可下载的代码示例
- 新技术预览

您可以在以下位置找到性能分析器资源：
http://developers.sun.com/sunstudio/analyzer_index.html。

文档、支持和培训

Sun Web 站点提供有关下列附加资源的信息：

- 文档 (`http://www.sun.com/documentation/`)
- 支持 (`http://www.sun.com/support/`)
- 培训 (`http://www.sun.com/training/`)

联系 Sun 技术支持

如果您遇到通过本文档无法解决的技术问题，请访问以下 URL：

http://www.sun.com/service/contacting

Sun 欢迎您提出意见

Sun 致力于提高其文档的质量，并十分乐意收到您的意见和建议。您可以通过以下 URL 向 Sun 提交您的意见：

http://www.sun.com/documentation/feedback/feedback.jsp

请在您的反馈信息中注明文档的文件号码 (820-3242-10)。

第 1 章

性能分析器概述

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。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定位出现这些问题的代码部分。

从集成开发环境中启动性能分析器

有关从集成开发环境 (Integrated Development Environment, IDE) 中启动性能分析器的信息，请参见性能分析器自述文件，可通过位于 /*installation_directory*/docs/index.html 的文档索引获取该文件。Solaris 平台上的缺省安装目录为 /opt/SUNWspro。Linux 平台上的缺省安装目录为 /opt/sun/sunstudio12。如果 Sun Studio 12 编译器和工具未安装在 /opt 目录下，请询问系统管理员以获取系统中的等效路径。

性能分析工具

本手册介绍了收集器和性能分析器这一对 Sun Studio 工具，您可以使用它们来收集和分析应用程序的性能数据。既可通过命令行界面也可通过图形用户界面使用这两个工具。

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

收集器工具使用名为分析 (profiling) 的统计方法，并通过跟踪函数调用来收集性能数据。这些数据可能包括调用栈、微态记帐信息、线程同步延迟数据、硬件计数器溢出数据、消息传递接口 (Message Passing Interface, MPI) 函数调用数据、内存分配数据以及操作系统和进程的摘要信息。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后续进程的数据。有关收集的数据的信息，请参见第 2 章；有关收集器的详细信息，请参见第 3 章。可以通过性能分析器 GUI、IDE、`dbx` 命令行工具和使用 `collect` 命令运行收集器。

性能分析器工具

性能分析器工具显示收集器记录的数据，以便于您检查这些信息。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。这些度量分为五组：

- 时钟分析度量
- 硬件计数器度量
- 同步延迟度量
- 内存分配度量
- MPI 跟踪度量

性能分析器还可以按图形格式显示作为时间函数的原始数据。性能分析器可以创建映射文件，您可以使用该映射文件更改函数在程序地址空间的装入顺序，以提高性能。

有关性能分析器的详细信息，请参见第 4 章以及 IDE 或性能分析器 GUI 中的联机帮助。

第 5 章介绍如何在 Solaris™ 操作系统 (Solaris Operating System, Solaris OS) 运行负载时使用 Sun Studio 性能工具分析内核。

第 6 章介绍如何使用 `er_print` 命令行界面来分析收集器收集的数据。

第 7 章讨论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、调用栈和程序执行，以及已注释的代码列表。可以使用 `er_src` 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源代码列表和反汇编代码列表（有关更多信息，请参见第 9 章）。

第 8 章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。

第 9 章介绍如何复制、移动、删除、归档和导出实验。

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，但时间线显示除外。

tcov 实用程序

Sun Studio 软件还包含名为 tcov 的另外一款分析工具，该工具可生成程序中执行的每条语句的确切次数计数。有关此工具的更多信息，请参见 tcov(1) 手册页。

性能分析器窗口

注 – 以下是性能分析器窗口的简要概述。有关下文讨论的标签的功能和特性的完整详细论述，请参见第 4 章和联机帮助。

性能分析器窗口由带有菜单栏和工具栏的多标签化显示组成。性能分析器启动时显示的标签显示了该程序的函数列表，以及每个函数的独占和包含度量。可以按装入对象、线程、轻量级进程 (lightweight process, LWP)、CPU 和时间片来过滤该列表。

对于某个选定的函数，另一个标签会显示该函数的调用者和被调用者。可以使用此标签导航调用树，例如在搜索较大的度量值时。

另外两个标签显示用性能度量逐行注释的并与编译器注释交错的源代码，以及用每个指令的度量注释的并与可用的源代码和编译器注释交错的反汇编代码。

性能数据在另一个标签中显示为时间函数。

其他标签显示实验和装入对象的详细信息，函数和内存泄露的摘要信息，以及进程的统计数据。

其他标签显示索引对象、内存对象、数据对象、数据布局、行及 PC。有关各标签的更多信息，请参见第 76 页中的 “分析器数据显示” 。

对于记录了线程分析器数据的实验，数据争用和死锁标签也将可用。仅在装入的实验具有支持这些标签的数据时才会显示标签。

有关线程分析器的更多信息，请参见《Sun Studio 12：线程分析器用户指南》 。

使用键盘和鼠标都可以导航性能分析器。

传统的 UNIX 性能工具

Solaris OS 长期以来一直提供两个标准 UNIX® 分析工具，`prof` 和 `gprof`。`prof` 实用程序生成程序使用的 CPU 时间的统计分析数据，以及进入每个函数的次数的确切计数。`gprof` 实用程序生成程序使用的 CPU 时间的统计分析数据，以及进入每个函数的次数的确切计数和程序调用图中遍历各 arc（调用者-被调用者对）的次数。这些工具对于简单的程序很有用，但对于调节复杂程序并不胜任。有关这些标准工具的更多信息，请参见 `prof`(1) 和 `gprof`(1) 手册页。

第 2 章

性能数据

性能工具的工作方式是，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度量（称为度量）。

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。

- 第 23 页中的 “收集器收集何种数据”
- 第 32 页中的 “如何将度量分配到程序结构”

有关收集和存储性能数据的信息，请参见第 3 章。

有关使用性能分析器分析性能数据的信息，请参见第 4 章。

有关在 Solaris OS 运行负载时分析内核的信息，请参见第 5 章。

有关使用 er_print 实用程序分析性能数据的信息，请参见第 6 章。

收集器收集何种数据

收集器可收集三种不同类型的数据：分析数据、跟踪数据和全局数据。

- 可通过以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上插入包装函数来收集跟踪数据，以便拦截对系统函数的调用，并记录有关调用的数据。
- 可通过调用各种系统例程以获取信息来收集全局数据。全局数据包称为样本。

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。全局数据不会转换为度量，而是用于提供标记器，这些标记器可用于将程序执行划分为很多时间段。通过全局数据，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
- 调用栈的副本。对于 Java 程序，将记录两个调用栈：机器调用栈和 Java 调用栈。
- 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 7 章。

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。收集器可以记录的五种数据类型为：

- 时钟分析数据
- 硬件计数器溢出分析数据
- 同步等待跟踪数据
- 堆跟踪（内存分配）数据
- MPI 跟踪数据

以下几个小节将介绍这五种数据类型（度量即是根据这五种数据类型得出的），以及如何使用这些数据类型。第六种数据类型（全局抽样数据），无法转换为度量，因为它不包含调用栈信息。

时钟数据

进行基于时钟的分析时，收集的数据取决于操作系统所提供的度量。

Solaris OS 下基于时钟的分析

在 Solaris OS 下基于时钟的分析中，将按固定的时间间隔存储每个 LWP 的状态。这种时间间隔称为分析间隔。这些信息存储在一个整数数组中：数组的一个元素用于内核维护的十个微记帐状态中的每一个状态。收集的数据通过性能分析器转换为每个状态所用的时间和分析间隔的精度。缺省分析间隔约为 10 毫秒 (10 ms)。收集器提供的高精度分析间隔大约为 1 ms，低精度分析间隔大约为 100 ms，如果 OS 允许，则可使用任意的间隔。运行 `collect` 命令（不带任何参数）可列显运行该命令的系统所允许的范围和精度。

下表定义了从基于时钟的数据计算得来的度量。

表 2–1 Solaris 计时度量

度量	定义
用户 CPU 时间	在 CPU 中按用户模式运行所用的 LWP 时间。
挂钟时间	LWP 1 中所用的 LWP 时间。该时间通常称为“挂钟时间”。
全部 LWP 时间	全部 LWP 时间的总和。
系统 CPU 时间	在 CPU 中或陷阱状态下按内核模式运行所用的 LWP 时间。
等待 CPU 时间	等待 CPU 所用的 LWP 时间。
用户锁定时间	等待锁定所用的 LWP 时间。
文本缺页时间	等待文本页所用的 LWP 时间。
数据缺页时间	等待数据页所用的 LWP 时间。
其他等待时间	等待内核页所用的 LWP 时间，或休眠/ 停止所用的时间。

对于多线程实验，将计算所有 LWP 的时间（挂钟时间除外）的总和。所定义的挂钟时间对于多程序多数据 (multiple-program multiple-data, MPMD) 程序没有意义。

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，此外还可用于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生成的代码会在内存中进行组织，所以调用或分 支会导致新的页面被装入。创建和使用映射文件（请参见性能分析器联机帮助中的“生成和使用映射文件”）可以修复这种问题。
- 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 OS 下基于时钟的分析

在 Linux OS 下，唯一可用的度量是用户 CPU 时间。虽然报告的总 CPU 占用时间是准确的，但分析器不可能像在 Solaris OS 中那样准确地确定实际系统 CPU 时间的时间比例。虽然分析器显示的信息好像是轻量级进程 (lightweight process, LWP) 数据，但实际上 Linux OS 中没有 LWP 的数据；所显示的 LWP ID 实际上是线程 ID。

硬件计数器溢出分析数据

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。在硬件计数器溢出分析中，当运行 LWP 的 CPU 的指定硬件计数器溢出时，收集器会记录分析数据包。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 CPU 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，收集器都允许您选择计数器的类型来监视溢出，并设置计数器的溢出值。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。

硬件计数器溢出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 的机器上，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。下一小节讲述如何找出可用的硬件计数器。

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。

某些硬件计数器可提供类似或相关的信息。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，而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。

通常会在导致事件和相应事件计数器溢出的指令之后，向硬件计数器溢出传送一条或多条指令：这称为“失控 (skid)”，它会使计数器溢出分析数据难以解释。如果缺少对精确识别因果指令的硬件支持，则可以对候选的因果指令尝试合适的回溯搜索。

收集期间支持和指定这种回溯时，硬件计数器分析数据包还包括适用于硬件计数器事件的候选内存引用指令的 PC（program counter，程序计数器）和 EA（effective address，有效地址）。（在分析期间需要进行后续处理来验证候选事件 PC 和 EA）。有关内存引用事件的此附加信息有助于进行各种面向数据的分析。

也可以为时钟分析指定候选事件 PC 和 EA 的回溯和记录。

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。通过在特定系统上的终端窗口键入不带参数的 `collect`，您可以从收集器获得该系统上可用的硬件计数器列表。如果处理器和系统支持硬件计数器分析，则 `collect` 命令会列显两个包含有关硬件计数器信息的列表。第一个列表包含“周知”（有别名的）硬件计数器；第二个列表包含原始硬件计数器。

以下示例显示了计数器列表中的条目。被认为是周知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。

```
Well known HW counters available for profiling:
cycles[/{0|1}],9999991 ('CPU Cycles', alias for Cycle_cnt; CPU-cycles)
insts[/{0|1}],9999991 ('Instructions Executed', alias for Instr_cnt; events)
dcrm[/1],100003 ('D$ Read Misses', alias for DC_rd_miss; load events)
...
Raw HW counters available for profiling:
Cycle_cnt[/{0|1}],1000003 (CPU-cycles)
Instr_cnt[/{0|1}],1000003 (events)
DC_rd[/0],1000003 (load events)
```

周知硬件计数器列表的格式

在周知硬件计数器列表中，第一个字段（例如 `cycles`）是可以在 `collect` 命令的 `-h` *counter...* 参数中使用的别名。该别名还是在 `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}]`。对于周知计数器，选择的缺省值可提供合理的抽样率。由于实际抽样率变化相当大，因此可能需要您指定缺省值以外的值。

第三个字段（例如 `9999991`）是计数器的缺省溢出值。

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 `Cycle_cnt`）以及计数单位类型（例如 `CPU-cycles`，其中最多可以包括两个单词）。

如果类型信息的第一个单词是：

- `load`、`store` 或 `load-store`，则表明计数器与内存相关。您可以在 `collect -h` 命令中的计数器名称前放置一个 + 号（例如 `+dcrm`），以请求搜索引发事件的准确指令和虚拟地址。+ 号还可以启用数据空间分析；有关详细信息，请参见第 81 页中的 "“数据对象”标签"、第 82 页中的 "“数据布局”标签"和第 83 页中的 "“内存对象”标签"。
- `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将生成警告，并且分析不记录调用栈。

如果类型信息的第二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包含时间，但是也可以显示为事件计数。
- `events`，则度量是包含和独占事件计数，且无法转换为时间。

在示例中的周知硬件计数器列表中，类型信息包含一个单词的，如第一个计数器的 `CPU-cycles` 和第二个计数器的 `events`。类型信息包括两个单词的，如第三个计数器的 `load 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周知硬件计数器列表中信息的子集。每行包括由 `cpu-track`(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，分析不会记录调用栈，而是显示人工函数 `collector_not_program_related` 中所用的时间。线程和 LWP ID 会被记录，但没有任何意义。

原始计数器的缺省溢出值是 1000003。由于该值对于大多数原始计数器来说是不理想的，因此应该在指定原始计数器时指定超时值。

同步等待跟踪数据

在多线程程序中，不同线程执行的任务同步会导致应用程序的执行延迟，例如，一个线程要访问已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。等待锁定花费的时间称为等待时间。目前，只能在运行 Solaris OS 的系统中进行同步等待跟踪。

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决定，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）相乘。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。

Java 程序的同步跟踪基于线程尝试获取 Java 监视器时生成的事件。对于这些事件，会同时收集机器调用栈和 Java 调用栈；但对于 Java™ 虚拟机 (Java Virtual Machine, JVM) 软件中使用的内部锁定，不会收集任何同步跟踪数据。在机器表示法中，线程同步被移交给对 `_lwp_mutex_lock` 的调用，且不显示任何同步数据，因为没有跟踪这些调用。

同步等待跟踪数据被转换为以下度量：

表 2–2 同步等待跟踪度量

度量	定义
同步延迟事件。	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解除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` 以及解除分配函数 `free` 跟踪内存分配和解除分配请求。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。由于 Fortran 函数 `allocate` 和 `deallocate` 调用 C 标准库函数，因此还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
分配数	对内存分配函数的调用数。
分配的字节	每次调用内存分配函数时分配的字节总数。
泄漏数	调用内存分配函数（未对解除分配函数进行相应的调用）的数量。
泄漏的字节	已分配但未解除分配的字节数。

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。

内存泄漏的另一个常用定义（例如在 `dbx` 调试工具中）为：内存泄漏是一个动态分配的内存块，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。

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。目前，MPI 跟踪仅在运行 Solaris OS 的系统中可用。下面列出了用于收集数据的函数。

`MPI_Allgather`	`MPI_Allgatherv`	`MPI_Allreduce`
`MPI_Alltoall`	`MPI_Alltoallv`	`MPI_Barrier`
`MPI_Bcast`	`MPI_Bsend`	`MPI_Gather`
`MPI_Gatherv`	`MPI_Irecv`	`MPI_Isend`
`MPI_Recv`	`MPI_Reduce`	`MPI_Reduce_scatter`

MPI_Rsend	MPI_Scan	MPI_Scatter
MPI_Scatterv	MPI_Send	MPI_Sendrecv
MPI_Sendrecv_replace	MPI_Ssend	MPI_Wait
MPI_Waitall	MPI_Waitany	MPI_Waitsome
MPI_Win_fence	MPI_Win_lock	

MPI 跟踪数据被转换为以下度量。

表 2–4 MPI 跟踪度量

度量	定义
MPI 接收数	接收数据的 MPI 函数中接收操作的数量
接收的 MPI 字节	MPI 函数中接收的字节数
MPI 发送数	发送数据的 MPI 函数中发送操作的数量
发送的 MPI 字节	MPI 函数中发送的字节数
MPI 时间	对 MPI 函数的所有调用所花费的时间
其他 MPI 调用	对其他 MPI 函数的调用数量

接收或发送时记录的字节数为调用中给定的缓冲区大小。此数字可能比接收或发送的实际字节数大。在全局通信函数和集合通信函数中，假定直接进行处理器间通信且未优化数据传输或重新传送数据，则发送或接收的字节数为最大数量。

表 2–5 中列出了被跟踪的 MPI 库的函数，分类为 MPI 发送函数、MPI 接收函数、MPI 发送和接收函数以及其他 MPI 函数。

表 2–5 MPI 函数分类为发送、接收、发送和接收以及其他

类别	函数
MPI 发送函数	MPI_Bsend、MPI_Isend、MPI_Rsend、MPI_Send、MPI_Ssend
MPI 接收函数	MPI_Irecv、MPI_Recv
MPI 发送和接收函数	MPI_Allgather、MPI_Allgatherv、MPI_Allreduce、MPI_Alltoall、MPI_Alltoallv、MPI_Bcast、MPI_Gather、 MPI_Gatherv、MPI_Reduce、MPI_Reduce_scatter、MPI_Scan、MPI_Scatter、MPI_Scatterv、MPI_Sendrecv、MPI_Sendrecv_replace
其他 MPI 函数	MPI_Barrier、MPI_Wait、MPI_Waitall 、MPI_Waitany、MPI_Waitsome、MPI_Win_fence、MPI_Win_lock

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。

全局（抽样）数据

全局数据由收集器按名为样本包的包来记录。每个包中都包含一个数据包头、时间戳、内核的执行统计数据（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）统计数据。记录在样本包中的数据对程序来说是全局的，且不会转换为性能度量。记录样本包的过程称为抽样。

在以下情况下，样本包会被记录下来：

- 如果设置了有关停止的选项，而程序在 IDE 的“调试”窗口或 `dbx` 中因任何原因（如在断点处）停止时
- 如果选择了周期抽样，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
- 选择了“调试”→“性能工具箱”→“启用收集器”，并选中“收集器”窗口中的“周期样本”复选框时
- 使用 `dbx collector sample record` 命令手动记录样本时
- 如果代码中包含对 `collector_sample` 的调用，则在调用该例程时（请参见第 41 页中的“数据收集的程序控制”）
- 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（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）
- 创建后续进程前后

性能工具使用记录在样本包中的数据，按时间周期将数据分组，这称为样本。您可以通过选择一组样本过滤特定事件的数据，以便只查看这些特定时间周期的信息。您也可以查看每个样本的全局数据。

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

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。有关此过程的更多详细说明，请参见第 7 章。

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、包含和归属

性能分析器可计算三种类型的函数级度量：独占度量、包含度量和归属度量。

- 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。
- 包含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。
- 归属度量说明了包含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。

对于只出现在调用栈底部的函数（叶函数），独占度量和包含度量是相同的。

对于装入对象，也要计算独占度量和包含度量。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。装入对象的包含度量与函数的包含度量的计算方法相同。

函数的独占度量和包含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分别为**调用者**和**被调用者**。对于调用树中的每个函数：

- 函数调用者的归属度量说明了函数的包含度量在多大程度上归因于来自每个调用者的调用。调用者的归属度量的总和等于函数的包含度量。
- 函数被调用者的归属度量说明了函数的包含性度量在多大程度上来自对每个被调用者的调用。它们的总和加上函数的独占度量等于函数的包含度量。

各度量间的关系可通过以下等式表示：

$$\sum_{\text{调用者}} \text{归属度量} = \text{包含度量} = \left(\sum_{\text{被调用者}} \text{归属度量} + \text{独占度量} \right)$$

通过比较调用者或被调用者的归属度量和包含度量，可以得到以下进一步的信息：

- 调用者的归属度量和包含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者本身的工作。

- 被调用者的归属度量和包含度量之间的差额说明了被调用者的包含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。
- 使用包含度量确定程序中哪个调用序列负责高度量值。
- 使用归属度量跟踪负责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 中说明了独占、包含和归属度量，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。

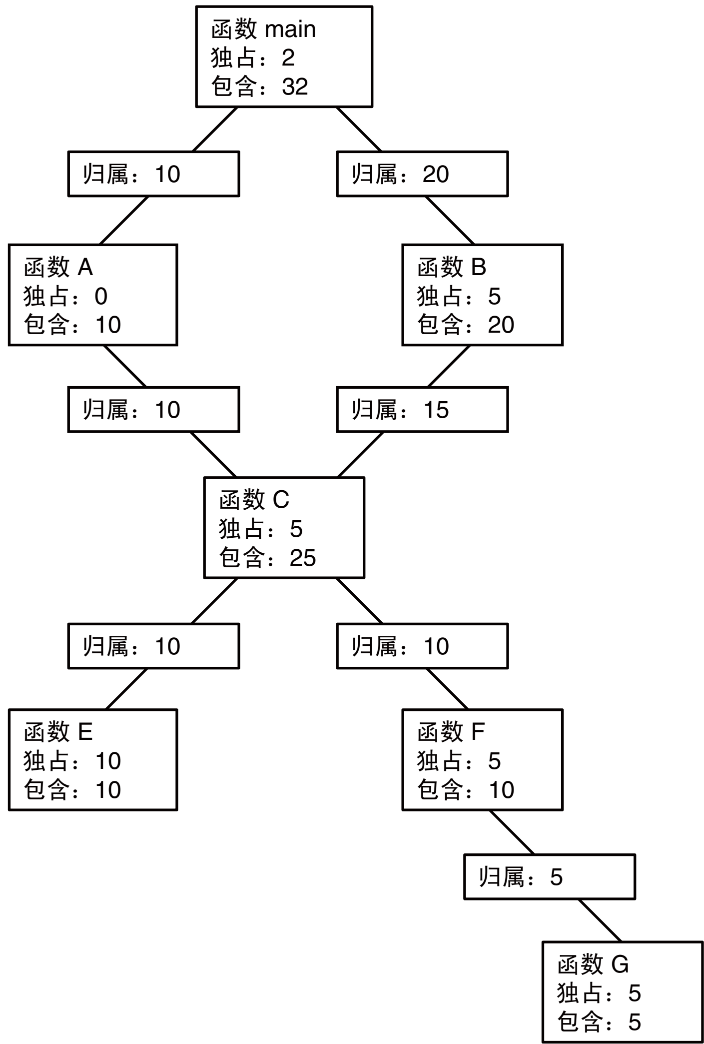

图 2–1 说明独占、包含和归属度量的调用树

Main 函数调用了函数 A 和函数 B，并将其 10 个单位的包含度量归属到函数 A，将 20 个单位的包含度量归属到函数 B。这些是函数 Main 的被调用者归属度量。它们的总和 (10+20) 加上函数 Main 的独占度量等于函数 main 的包含度量 (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 由以下两个函数调用：函数 A 和函数 B，并将其 10 个单位的包含度量归属到函数 A，将 15 个单位的包含度量归属到函数 B。这些是调用者归属度量。它们的总和 (10+15) 等于函数 C 的包含度量 (25)。

调用者归属度量等于函数 A 及 B 的包含度量和独占度量之间的差额，这意味着这两个函数只调用函数 C。（实际上，这两个函数可能会调用其他函数，但时间太短，在实验中不显示。）

函数 C 调用了两个函数，函数 E 和函数 F，并分别将其 10 个单位的包含度量归属到函数 E 和函数 F。这些是被调用者归属度量。它们的总和 (10+10) 加上函数 C 的独占度量 (5) 等于函数 C 的包含度量 (25)。

函数 E 及函数 F 的被调用者归属度量和被调用者包含度量是相同的。这意味着函数 E 及函数 F 仅由 C 调用。函数 E 的独占度量和包含度量是相同的，但函数 F 的这两种度量不同。这是因为函数 F 调用了其他函数（函数 G），但函数 E 没有调用。

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。这不会影响通过调用栈底部的函数（叶函数）计算得出的独占度量，但会影响包含度量和归属度量。

包含度量是通过将事件的度量添加到调用栈中函数的包含度量来计算的。为了确保在递归调用栈中不重复计算度量，事件的度量仅会添加到每个唯一函数的包含度量。

归属度量是通过包含度量计算得出的。在最简单的递归中，递归函数具有两个调用者：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，则会将递归函数的包含度量归属到它本身，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包含度量均被视为零，以避免重复计算度量。但是，初始化函数会由于递归调用而做为被调用者正确归属到递归函数的包含度量部分。

◆ ◆ ◆ 第 3 章

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。

本章包含以下主题。

- 第 37 页中的 “编译和链接程序”
- 第 38 页中的 “为数据收集和分析准备程序”
- 第 45 页中的 “数据收集的限制”
- 第 48 页中的 “数据的存储位置”
- 第 50 页中的 “估计存储要求”
- 第 51 页中的 “收集数据”
- 第 51 页中的 “使用 collect 命令收集数据”
- 第 60 页中的 “使用 dbx collector 子命令收集数据”
- 第 66 页中的 “使用 dbx 从正在运行的进程中收集数据”
- 第 68 页中的 “从 MPI 程序收集数据”
- 第 71 页中的 “将 collect 和 ppgsz 一起使用”

编译和链接程序

几乎可以为使用任何选项编译的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。

源代码信息

要查看带注释的 “源代码” 和 “反汇编” 分析中的源代码以及 “行” 分析中的源代码行，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`。

要使用允许使用数据空间分析的调试信息准备编译对象（当前仅适用于 SPARC® 处理器），请通过指定 `-xhwcprof -xdebugformat=dwarf` 和任何级别的优化来进行编译。（目前，这种功能在未经过优化的情况下无法使用。）要查看“数据对象”分析中的程序数据对象，也要添加 `-g`（对于 C++ 来说为 `-g0`）以获取全部符号信息。

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目标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与 `-xs` 选项（该选项将 stabs 符号保留在各个目标文件及可执行文件中）相链接，那么所生成的可执行文件和库中也会包括每个要素目标文件的调试符号。当您需要移动或删除目标文件时，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。

静态链接

编译程序时，必须使用 `-dn` 和 `-Bstatic` 编译器选项打开动态链接。如果试图收集完全静态链接的程序的数据，则收集器会列显一条错误消息并且不收集数据。出现此错误的原因在于，当您运行收集器时，该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。如果您执行了静态链接，则可能无法收集任何种类的跟踪数据。另外，请不要链接到收集器库 `libcollector.so`。

编译时优化

如果使用在某一级别打开的优化来编译程序，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。性能分析器可以分析在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，请参见第 137 页中的 “尾部调用优化” 。

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
- 显式动态装入系统库
- 动态编译函数

- 创建要分析的后续进程
- 使用异步 I/O 库
- 直接使用分析计时器或硬件计数器 API
- 调用 setuid(2) 或执行 setuid 文件

此外，如果要控制程序中的数据收集，还应当阅读相关小节。

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) 手册页中对 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错误：除非明确说明使用其他方法，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，这些潜在的错误会引起意外的行为。

下列工具可以帮助您找到这些潜在的错误：

- f95 -xcheck=init_local

  有关更多信息，请参见《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《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《使用 dbx 调试程序》手册或 dbx(1) 手册页。
- Rational Purify

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。收集器插入 Solaris 线程库 `libthread.so`（在 Solaris 9 OS 上）和 Solaris C 库 `libc.so`（在 Solaris 10 OS 上）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。收集器插入 Solaris MPI 库 `libmpi.so` 的函数。
- 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`。
- 针对后续进程启用数据收集。收集器插入函数 `fork`(2)、`fork1`(2)、`vfork`(2)、`fork`(3F)、`system`(3C)、`system`(3F)、`sh`(3F)、`popen`(3C) 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。这些插入仅适用于 `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。

在某些情况下，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。

使用信号处理程序

收集器使用以下两个信号来收集分析数据：`SIGPROF`（所有实验）和 `SIGEMT`（仅限硬件计数器实验）。收集器为其中的每个信号安装一个信号处理程序。该信号处理程序截获并处理它自己的信号，但是会将其他信号传递到所安装的其他信号处理程序。如果程序为这些信号安装其自己的信号处理程序，则收集器会将其信号处理程序作为主处理程序重新安装，以保证性能数据的完整性。

`collect` 命令还可以将用户指定的信号用于暂停和恢复数据收集以及记录样本。尽管在安装用户处理程序时向实验中写入警告，但这些信号不受收集器保护。确保收集器对指定信号的使用与应用程序对相同信号的使用之间没有冲突是您的责任。

由收集器安装的信号处理程序会设置一个确保系统调用不被信号传送中断的标志。如果程序的信号处理程序将该标志设置为允许中断系统调用，则设置该标志可以更改程序的行为。在异步 I/O 库 `libaio.so` 中就有一个行为更改的重要示例，它将 `SIGPROF` 用

于异步取消操作，并且中断系统调用。如果已安装收集器库 libcollector.so，则取消信号总是来得太迟，以至于无法取消异步 I/O 操作。

如果在未预装入收集器库和启用性能数据收集的情况下将 dbx 附加到进程，并且程序随后安装其自身的信号处理程序，则收集器不再重新安装其自身的信号处理程序。在这种情况下，程序的信号处理程序必须确保 SIGPROF 和 SIGEMT 信号被传递，以便性能数据不丢失。如果程序的信号处理程序中断系统调用，那么程序行为和分析行为都将与预装入收集器库时不同。

使用 setuid

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。

此问题可以通过以下方法来解决：确保 umask 的设置可将写入权限授予进程在运行时所使用的任何 UID 或 GID。针对实验，这些 ID 必须具有写入权限。

数据收集的程序控制

如果要控制程序中的数据收集，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 编写的，还提供了一个 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_thread_pause(unsigned int t);
void collector_thread_resume(unsigned int t);
void collector_terminate_expt(void);
```

CollectorAPI 类为 Java™ 程序提供了类似的功能，第 42 页中的 “Java 接口”中对其进行了介绍。

C 和 C++ 接口

可通过两种方法来访问 C 和 C++ 接口：

- 第一种方法是包括 collectorAPI.h 并使用 -lcollectorAPI（包含用于检查底层 libcollector.so API 函数是否存在的真正的函数）进行链接。

  这种方法要求与 API 库相链接，可在所有情况下使用。如果没有活动的实验，API 调用将被忽略。

- 第二种方法是包括 `libcollector.h`（包含用于检查基础 `libcollector.so` API 函数是否存在的宏）。

  当用于主可执行文件以及数据收集功能随程序一起启动时，该方法有效。当 `dbx` 用于附加到进程或者从进程针对其执行 `dlopen` 的共享库中使用时，该方法不总是有效。提供第二种方法的目的**仅**在于实现向后兼容，建议不要将它用于任何其他目的。

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。

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 Sun 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.threadPause(Thread thread)
CollectorAPI.threadResume(Thread thread)
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，则跳过对真正的 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。除 `collector_thread_pause()` 和 `collector_thread_resume()` 之外，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。要在单独的线程级别控制数据收集，请使用 `collector_thread_pause()` 和 `collector_thread_resume()` 函数。可通过两种方法来使用这些函数：一种是使用一个主线程为所有线程（包括其自身）执行调用；另一种是让每个线程只执行它自己的调用。其他任何用法都可能产生不可预知的结果。

C、C++、Fortran 和 Java API 函数

对 API 函数的描述如下所示。

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

  Fortran： `collector_sample(string name)`

  Java：`CollectorAPI.sample(String name)`

  记录样本包并用指定的名称或字符串标记该样本。该标签显示在性能分析器的“事件”选项卡中。Fortran 参数 `string` 的类型为 `character`。

  样本点包含进程（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 函数记录样本时发生另一个调用，则该函数可确保只写入一个样本。所记录的样本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样本进行区分。如果只想查看 API 调用所记录的样本，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。

- **仅限 C 和 C++**：collector_thread_pause(unsigned int t)

  Java：CollectorAPI.threadPause(Thread t)

  停止将特定于事件的数据从参数列表中所指定的线程写入实验。对于 C/C++ 程序，参数 t 是 POSIX 线程标识符；对于 Java 程序，该参数是 Java 线程。如果实验已经终止、没有活动的实验或已经关闭对该线程数据的写入功能，则该调用将被忽略。即使全局启用了数据写入功能，该函数也会停止写入指定线程的数据。缺省情况下，对单独线程的数据记录功能处于打开状态。

- **仅限 C 和 C++**：collector_thread_resume(unsigned int t)

  Java：CollectorAPI.threadResume(Thread t)

  恢复将参数列表中指定线程的特定于事件的数据写入实验。对于 C/C++ 程序，参数 t 是 POSIX 线程标识符；对于 Java 程序，该参数是 Java 线程。如果实验已经终止、没有活动的实验或对该线程数据的写入功能已经打开，则该调用将被忽略。只有在全局启用了数据写入功能，而且还针对该线程启用了数据写入功能时，才可以将数据写入实验中。

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

  Java：CollectorAPI.terminate

  终止其数据正在被收集的实验。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。

对 API 函数的描述如下所示。

collector_func_load()

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“摘要”选项卡中。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()

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：您可以在收集某种数据类型的同时收集任何其他数据类型。

基于时钟的分析的限制

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。可以通过键入不带参数的 collect 命令来查找最小值和最大值以及时钟精度。

时钟分析中的运行时失真和扩大

基于时钟的分析记录当 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栈时产生扩大。调用栈越深，信号越频繁，扩大越显著。在一定程度上，基于时钟的分析表现出一些失真，这是由程序中那些执行最深栈部分的显著扩大而导致的。

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 SPARC 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，请参见第 67 页中的 “从正在运行的程序中收集跟踪数据”。

跟踪过程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。如果完成了基于时钟的分析，则跟踪事件所引起的扩大将导致时钟数据失真。

硬件计数器溢出分析的限制

硬件计数器溢出分析存在多种限制：

- 只能在具有硬件计数器且支持溢出分析的处理器上收集硬件计数器溢出数据。在其他系统中，硬件计数器溢出分析功能处于禁用状态。UltraSPARC® III 处理器系列之前的 UltraSPARC 处理器不支持硬件计数器溢出分析。
- 在 cpustat(1) 运行过程中无法收集系统的硬件计数器溢出数据，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，则硬件计数器溢出分析将终止，而且会在实验中记录一条错误。
- 如果正在执行硬件计数器溢出分析，则无法将自己代码中的硬件计数器与 libcpc(3) API 同时使用。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。如果未进行这样的编码，或者如果根调用了同样使用计数器的系统范围的工具，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，请运行不带参数的 collect 命令。

硬件计数器溢出分析中的运行时失真和扩大

硬件计数器溢出分析功能记录当 SIGEMT 传递到目标时的数据。这将导致在处理该信号和展开调用栈时产生扩大。与基于时钟的分析不同的是，对于某些硬件计数器，程序的不同部分可能会比其他部分更快速地生成事件并显示在该部分代码中的扩大。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。

后续进程中数据收集的限制

可以在下列限制下在后续进程中收集数据。

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

- -F on 选项允许您针对 fork（及其变体）和 exec（及其变体）调用自动收集数据。
- -F all 选项导致收集器跟踪所有后续进程（包括那些因调用 system、popen 和 sh 而产生的进程）。
- -F '=*regexp*' 选项允许在其名称或沿袭与指定的正则表达式相匹配的所有后续进程上收集数据。

有关 -F 选项的更多信息，请参 第 55 页中的 “实验控制选项” 。

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 1.5.0_03 的 Java 2 Platform, Standard Edition (J2SE)。缺省情况下， collect 命令使用 Sun Studio 安装程序安装 J2SE 的路径（如果有）。可以通过设置 JDK_HOME 环境变量或 JAVA_PATH 环境变量来覆盖此缺省路径。收集器会验证它在这些环境变量中找到的 java 版本是否为 ELF 可执行文件，如果不是，则列显一条错误消息，指出所使用的环境变量以及已尝试使用的全路径名。
- 必须使用 collect 命令来收集数据，而不能使用 dbx collector 子命令或 IDE 的数据收集功能。
- 如果应用程序所创建的后续进程运行 JVM 软件，则不能对这些应用程序进行分析。
- 如果要使用 64 位 JVM 软件，则必须使用 -j on 标志并将 64 位 JVM 软件指定为目标。请不要使用 java -d64 在 64 位 JVM 软件中收集数据，否则收集不到任何数据。

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。

对于基于时钟的分析和硬件计数器溢出分析，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%。

对于同步跟踪，数据收集功能使用其他 JVMTI 事件，这将导致扩大（与应用程序中监视争用数量成比例）。

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。实验由存储在一个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目标文件和函数的地址、大小和名称以及装入对象的地址和上次修改的时间戳。

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。

后续进程的实验存储在创始进程的实验内部。

实验名称

新实验的缺省名称为 `test.1.er`。后缀 `.er` 是必需的：如果您赋予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，则收集器会列显一条错误消息。

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`。如果名称不以 `.erg` 结尾，则系统会显示一条错误并且不接受该名称。创建实验组后，您使用该组名运行的所有实验都会添加到该组中。

可以通过创建首行为

```
#analyzer experiment group
```

的纯文本文件并将实验名称添加到后续行来手动创建实验组文件。文件的名称必须以 .erg 结尾。

还可以通过使用带有 -g 参数的 collect 命令来创建实验组。

缺省实验名称与从 MPI 程序中收集的实验名称不同，MPI 程序会为每个 MPI 进程创建一个实验。缺省实验名称为 test.*m*.er，其中 *m* 是进程的 MPI 等级。如果指定了实验组 *group*.erg，则缺省实验名称为 *group.m*.er。如果指定了实验名称，则该名称会覆盖缺省值。有关更多信息，请参见第 68 页中的 “从 MPI 程序收集数据” 。

后续进程的实验是沿袭命名的，如下所示。要形成后续进程的实验名称，可将下划线、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，如果创始进程的实验名称为 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，则新后续进程的实验名称为 test.1.er/_f3_x1.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，则应了解分析对在其中记录实验的操作环境的依赖性。

归档文件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，则必须能够访问与记录实验时所用装入对象或源文件相同的版本。

性能分析器在下列位置依次搜索源代码、对象和可执行文件，并在找到具有正确基本名称的文件时停止：

- 实验的归档目录。
- 当前的工作目录。
- 记录在可执行文件或编译对象中的绝对路径名。

可以从分析器 GUI 或通过使用 setpath 和 addpath 指令来更改搜索顺序或添加其他搜索目录。

为了确保能够看到程序的正确的带注释源代码和带注释反汇编代码，可以在移动或复制实验之前将源代码、目标文件和可执行文件复制到该实验中。如果您不想复制目标文件，则可以使用 -xs 来链接程序，以确保源代码行和文件位置上的信息插入可执行文件中。可以通过使用 collect 命令的 -A 选项或 dbx collector archive 命令将装入对象自动复制到实验中。

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 LWP 的数据包。请注意，对于 Java 程序，有两个相关的调用栈：Java 调用栈和机器调用栈，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来控制。但是，对于这些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精心选择了缺省值。值越小，数据越多。

对于一个具有 10 毫秒分析间隔和较小调用栈的基于时钟的分析实验，包的大小为 100 个字节，对于每个 LWP，记录数据的速率为 10 千字节/ 秒。对于溢出值为 1000000 以及在 750 MHz 处理器上收集有关 CPU 循环和执行指令数据的硬件计数器溢出分析实验，包的大小为 100 个字节，对于每个 LWP，记录数据的速率为 150 千字节/ 秒。调用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。

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，它通常是所要求的总磁盘空间的一小部分（请参见上一节）。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。

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 `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 `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。

注 – 性能分析器无法读取大于 2 GB 的性能数据。

收集数据

可以通过多种方法在独立性能分析器或 IDE 的分析器窗口中收集性能数据：

- 在命令行上使用 collect 命令（请参见第 51 页中的 “使用 collect 命令收集数据”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。
- 使用性能分析器的“性能工具收集”对话框（请参见性能分析器联机帮助中的“使用性能工具收集器收集性能数据”）。
- 使用调试器中的“收集器”对话框（请参见性能分析器联机帮助中的“使用调试器收集性能数据”）。
- 在 dbx 命令行上使用 collector 命令（请参见第 60 页中的 “使用 dbx collector 子命令收集数据”以及 IDE 中“调试”联机帮助中的“collector 命令”）。

只有“性能工具收集”对话框和 collect 命令提供下列数据收集功能：

- 收集 Java 程序中的数据。如果尝试使用 IDE 调试器中的“收集器”对话框或者 dbx 中的 collector 命令收集 Java 程序中的数据，则收集的是 JVM 软件（而不是 Java 程序）的信息。
- 自动收集后续进程中的数据。

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
```

其中，*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。

如果未提供 *collect-options*，则缺省情况下会打开分析间隔大约为 10 毫秒的基于时钟的分析。

要获取可用于分析的选项列表和所有硬件计数器的名称列表，请键入不带参数的 collect 命令。

```
% collect
```

有关对硬件计数器列表的描述，请参见第 26 页中的 “硬件计数器溢出分析数据”。另请参见第 46 页中的 “硬件计数器溢出分析的限制”。

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，请参见第 23 页中的 “收集器收集何种数据” 。

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 毫秒的基于时钟的分析。该缺省值由 -h 选项关闭，而不是由任何其他数据收集选项关闭。

如果您明确禁用了基于时钟的分析，而且未启用跟踪或硬件计数器溢出分析，则 collect 命令会列显一条警告消息，并且只收集全局数据。

-p *option*

收集基于时钟的分析数据。*option* 的允许值包括：

- off－关闭基于时钟的分析。
- on－打开缺省分析间隔大约为 10 毫秒的基于时钟的分析。
- lo[w]－打开分析间隔大约为 100 毫秒（低精度）的基于时钟的分析。
- hi[gh]－打开分析间隔大约为 1 毫秒（高精度）的基于时钟的分析。有关启用高精度分析的信息，请参见第 45 页中的 “基于时钟的分析的限制” 。
- [+]*value*－打开基于时钟的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，则会列显一条警告消息并将其设置为时钟精度。

  在 SPARC 平台上，可以像对硬件计数器分析那样在任何值前面添加 + 符号来启用基于时钟的数据空间分析。

collect 命令的缺省操作是收集基于时钟的分析数据。

-h *counter_definition_1* ...[,*counter_definition_n*]

收集硬件计数器溢出分析数据。计数器定义的数量与处理器有关。如果安装了 perfctr 修补程序（可以从 http://user.it.uu.se/~mikpe/linux/perfctr/2.6/perfctr-2.6.15.tar.gz 下载），则该选项当前在运行 Linux 操作系统的系统上是可用的。

计数器定义可以采用下列形式之一，具体取决于处理器是否支持硬件计数器的属性。

[+]*counter_name*[/*register_number*][,*interval*]

[+]*counter_name*[~ *attribute_1=value_1*]...[~*attribute_n=value_n*][/*register_number*][,*interval*]

特定于处理器的 *counter_name* 可以为下列名称之一：

- 周知的（有别名的）计数器名称
- 由 cputrack(1) 使用的原始（内部）名称。如果计数器可以使用任一事件寄存器，则可以通过向内部名称附加 /0 或 /1 来指定将要使用的事件寄存器。

如果指定了多个计数器，则它们必须使用不同的寄存器。如果它们未使用不同的寄存器，则 collect 命令会列显一条错误消息并退出。某些计数器可在任一寄存器上计数。

要获取可用计数器的列表，请在终端窗口中键入不带参数的 collect。第 26 页中的 “硬件计数器列表”一节提供了对计数器列表的介绍。

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添加 + 符号，以针对引起计数器溢出的指令打开对其真实程序计数器地址 (PC) 的搜索。这种回溯功能适用于 SPARC 处理器，并且仅适用于类型为 load、store 或 load-store 的计数器。如果搜索成功，则所引用的虚拟 PC、物理 PC 和有效地址将存储在事件数据包中。

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。如果某个处理器支持多个属性选项，则运行不带参数的 collect 命令会列出计数器定义（包括属性名）。可以使用十进制或十六进制格式来指定属性值。

间隔（溢出值）是事件计数的数量，在达到该数量时，硬件计数器将溢出并且将记录溢出事件。间隔可以设置为下列值之一：

- on 或空字符串－缺省溢出值，可以通过键入不带参数的 collect 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。

缺省值是为每个计数器预定义的正常阈值，它出现在计数器列表中。另请参见第 46 页中的 “硬件计数器溢出分析的限制”。

如果在使用 -h 选项时未明确指定 -p 选项，则基于时钟的分析功能将处于关闭状态。要同时收集硬件计数器数据和基于时钟的数据，必须同时指定 -h 选项和 -p 选项。

-s *option*

收集同步等待跟踪数据。*option* 的允许值包括：

- all－启用具有零阈值的同步等待跟踪。该选项强制记录所有同步事件。
- calibrate－启用同步等待跟踪并在运行时通过校准来设置阈值。（与 on 等效。）
- off－禁用同步等待跟踪。
- on－启用具有缺省阈值的同步等待跟踪，这将在运行时通过校准来设置缺省阈值。（与 calibrate 等效。）

- *value*－将阈值设置为 *value*，该值以正整数形式提供，单位为微秒。

不记录 Java 监视器的同步等待跟踪数据。

-H *option*

收集堆跟踪数据。*option* 的允许值包括：

- on－打开对堆分配和堆释放请求的跟踪。
- off－关闭堆跟踪。

缺省情况下，堆跟踪功能处于关闭状态。对于 Java 程序，不支持堆跟踪；如果指定堆跟踪，将被视为错误。

-m *option*

收集 MPI 跟踪数据。*option* 的允许值包括：

- on－打开对 MPI 调用的跟踪。
- off－关闭对 MPI 调用的跟踪。

缺省情况下，MPI 跟踪功能处于关闭状态。

有关其调用被跟踪的 MPI 函数以及从跟踪数据中所计算的度量的更多信息，请参见第 29 页中的 “MPI 跟踪数据”。

-S *option*

定期记录样本包。*option* 的允许值包括：

- off－关闭定期抽样功能。
- on－打开缺省抽样间隔为 1 秒的定期抽样功能。
- *value*－打开定期抽样功能并将抽样间隔设置为 *value*。间隔值必须为正数并且以秒为单位。

缺省情况下，启用间隔为 1 秒的定期抽样功能。

-c *option*

记录计数数据（仅用于 SPARC 处理器）。

注 – 此功能要求您安装 Binary Interface Tool (BIT)，它是 Sun Studio 12（可从 http://cooltools.sunsource.net/ 获取）中一个很不错的附加工具。BIT 是用来度量 SPARC 二进制代码的性能或测试套件适用范围的工具。

---

option 的允许值包括：

- on－打开对函数和指令计数数据的收集。如果可执行文件及其静态链接的任何共享对象是使用 -xbinopt=prepare 标志编译的，则会记录这些可执行文件和共享对象的数据。虽然是静态链接但是未使用 -xbinopt=prepare 标志编译的任何其他共享对象将不包含在数据中。同样，动态打开的任何共享对象将不包含在数据中。可以在性能分析器的“指令-频率”选项卡中或使用 er_print ifreq 命令查看数据。
- static－在目标可执行文件中的每个指令以及所有静态链接的共享对象都刚好执行一次的情况下，将生成一个实验。和 -c on 选项一样，-c static 选项也要求使用 -xbinopt=prepare 标志编译可执行文件和共享对象。

-r *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：

- on－打开线程分析器的数据争用检测数据
- off－关闭线程分析器数据
- all－打开所有线程分析器数据
- race－打开线程分析器的数据争用检测数据
- deadlock－收集死锁和潜在死锁数据
- dt*N*－打开特定的线程分析器数据类型，这些数据类型由 dt* 参数指定。

有关 collect -r 命令和线程分析器的更多信息，请参见《Sun Studio 12：线程分析器用户指南》和 tha.1 手册页。

实验控制选项

-F *option*

控制是否应当记录后续进程的数据。*option* 的允许值包括：

- on－仅针对由函数 fork（及其变体）和 exec（及其变体）创建的后续进程记录实验。
- all－针对所有后续进程记录实验。
- off－不针对后续进程记录实验。
- =*regexp*－针对其名称或沿袭与指定的正则表达式匹配的所有后续进程记录实验。

如果您指定 -F on 选项，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。

如果您指定 -F all 选项，收集器将跟踪所有后续进程，其中包括那些通过调用 system(3C)、system(3F)、sh(3F) 和 popen(3C) 以及类似函数而创建的后续进程以及与它们相关的后续进程。

如果您指定 -F '= *regexp*' 选项，收集器将跟踪其名称或沿袭与指定的正则表达式相匹配的所有后续进程。有关正则表达式的信息，请参见 regexp(5) 手册页。

当您在后续进程上收集数据时，收集器会针对创始实验中的每个后续进程打开一个新实验。这些新实验是通过向实验后缀添加一个下划线、一个字母和一个数字来命名的，如下所示：

- 字母可以是 "f"（表示派生）、"x"（表示执行）或 "c"（表示任何其他后续进程）。
- 数字是派生或执行（无论是否成功）或其他调用的索引。

例如，如果初始进程的实验名称是 test.1.er，则由它的第三个派生创建的子进程的实验是 test.1.er/_f3.er。如果该子进程针对新映像执行 exec 操作，则相应的实验名称为 test.1.er/_f3_x1.er。如果该子进程使用 popen 调用创建另一个进程，则实验名称为 test.1.er/_f3_x1_c1.er。

分析器和 er_print 实用程序在读取创始实验后会自动读取后续进程的实验，但不会选择将后续进程的实验用于数据显示。

要从命令行中选择要显示的数据，请明确指定 er_print 或 analyzer 的路径名。所指定的路径必须包含创始实验的名称以及创始目录中后续实验的名称。

例如，要查看 test.1.er 实验的第三个派生的数据，需要指定以下内容：

er_print test.1.er/_f3.er

analyzer test.1.er/_f3.er

或者，可以使用感兴趣的后续实验的显式名称来准备实验组文件。

要在分析器中检查后续进程，请装入创始实验并从“视图”菜单中选择“过滤数据”。此时将显示一个实验列表，其中只有创始实验处于选中状态。取消选中初始实验并选中感兴趣的后续实验。

注 – 如果在后续进程正在被跟踪时创始进程退出，则可能会继续从后续进程中收集数据。创始实验目录会相应地继续变大。

-j *option*

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

- on－识别由 Java HotSpot 虚拟机编译的方法并尝试记录 Java 调用栈。
- off－不尝试识别由 Java HotSpot 虚拟机编译的方法。
- *path*－记录安装在指定 *path* 中的 JVM 的分析数据。

如果要收集 .class 文件或 .jar 文件中的数据，则不需要 -j 选项，但前提是 java 可执行文件的路径在 JDK_HOME 环境变量或 JAVA_PATH 环境变量中。随后可以在 collect 命令行上将目标 *program* 指定为具有或不具有扩展名的 .class 文件或 .jar 文件。

如果无法在 JDK_HOME 或 JAVA_PATH 环境变量中定义java 的路径，或者要禁用对 Java HotSpot 虚拟机所编译的方法的识别，则可以使用 -j 选项。如果使用该选项，则在

collect 命令行上指定的 *program* 必须是版本不低于 1.5_03 的 Java 虚拟机。collect 命令验证 *program* 是否为 JVM 以及是否为 ELF 可执行文件；如果不是，collect 命令会列显一条错误消息。

如果要使用 64 位 JVM 收集数据，则不能将 java 的 -d64 选项用于 32 位 JVM，否则将收集不到任何数据；相反，您必须在 collect 命令的 *program* 参数中或本节提供的两个环境变量之一中指定 64 位 JVM 的路径。

-J *java_argument*

指定要传递到用于分析的 JVM 的单个参数。如果您指定了 -J 选项但未指定 Java 分析，则会生成一个错误并且不运行实验。该参数作为单个参数传递到 JVM。如果需要多个参数，请不要使用 -J 选项，而是明确指定 JVM 的路径，使用 -j on，并在 collect 命令行上在 JVM 路径的后面添加 JVM 的参数。

-l *signal*

当名为 *signal* 的信号传递到进程时，记录样本包。

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。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。

如果您使用该选项而程序具有其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，而不是被截获或忽略。

有关信号的更多信息，请参见 signal(3HEAD) 手册页。

-t *duration*

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。

-x

在从 exec 系统调用退出时使目标进程停止，以便允许调试器附加到目标进程。如果将 dbx 附加到目标进程，请使用 dbx 命令 ignore PROF 和 ignore EMT 来确保收集信号传递到 collect 命令。

-y *signal*[,r]

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。无论切换状态如何，都将始终记录样本点。

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。可通过 kill(1) 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其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，而不是被截获或忽略。

有关信号的更多信息，请参见 signal(3HEAD) 手册页。

输出选项

-o *experiment_name*

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 实用程序会列显一条错误消息并退出。

-d *directory-name*

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 实用程序会列显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。

-g *group-name*

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 实用程序会列显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。

-A *option*

控制是否应将目标进程所使用的装入对象归档或复制到已记录的实验中。option 的允许值包括：

- off－不将装入对象归档到实验中。

- on－将装入对象归档到实验中。
- copy－将装入对象复制和归档到实验中。

如果您希望将实验从记录的位置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 copy。使用该选项不会将任何源文件或目标文件复制到实验中。您应当确保在要将实验复制到的计算机上可以访问这些文件。

-L *size*

将所记录的分析数据量限制在 *size* 兆字节。该限制适用于基于时钟的分析数据量、硬件计数器溢出分析数据量和同步等待跟踪数据量之和，但不适用于样本点。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，直到目标进程终止。如果启用了定期抽样，则会继续写入样本点。

记录的数据量的缺省限制为 2000 MB。选择该限制的原因是性能分析器无法处理所含数据量大于 2 GB 的实验。要去掉该限制，请将 *size* 设置为 unlimited 或 none。

-O *file*

将 collect 本身的所有输出附加到名称 *file*，但是不重定向所产生的目标的输出。如果该文件设置为 /dev/null，则禁止 collect 的所有输出（包括任何错误消息）。

其他选项

-C comment

将注释放在实验的 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。

-n

不运行目标，但列显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。

-R

在终端窗口中显示性能分析器自述文件的文本版本。如果未找到自述文件，则列显一条警告。不再检查任何参数，也不执行进一步的处理。

-V

列显 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。

`-v`

列显 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

可以将 -P *pid* 选项与 collect 实用程序一起使用来连接具有指定 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。只能收集基于时钟的分析数据（-p 选项）和硬件计数器溢出分析数据（-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，则基于时钟的分析将处于关闭状态。要同时收集硬件计数器数据和基于时钟的数据，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)。**

   如果从命令行启动程序并将其放置在后台中，则其 PID 将由 shell 列显到标准输出中；否则，您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 命令启用对该进程的数据收集功能并设置任何可选参数。**

   % **collect -P** *pid collect-options*

   第 52 页中的“数据收集选项”中对收集器选项进行了说明。有关基于时钟的分析的信息，请参见第 52 页中的“-p *option*”。有关硬件时钟分析的信息，请参见-h option。

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
```

要获取可用 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 设置要使用的任何 dbx选项并运行该程序。

如果提供的子命令不正确，则会列显一条警告消息并忽略该子命令。下面是 collector 子命令的完整列表。

数据收集子命令

下列子命令控制收集器所收集的数据的类型。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*

控制对基于时钟的分析数据的收集。*option* 的允许值包括：

- on－启用缺省分析间隔为 10 毫秒的基于时钟的分析。
- off－禁用基于时钟的分析。
- 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，请参见第 45 页中的 "基于时钟的分析的限制" 。
  - *value*－将分析间隔设置为 *value*。*value* 的缺省单位是毫秒。可以将 *value* 指定为整数或浮点数。可以选择在数值后添加后缀 m 来选择毫秒单位，也可以添加 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列显警告消息。

    缺省设置大约为 10 毫秒。

    缺省情况下收集器收集基于时钟的分析数据，除非使用 hwprofile 子命令打开对硬件计数器溢出分析数据的收集。

hwprofile *option*

控制对硬件计数器溢出分析数据的收集。如果您尝试在不支持硬件计数器溢出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－打开硬件计数器溢出分析。缺省操作是收集具有正常溢出值的 cycles 计数器的数据。
- off－关闭硬件计数器溢出分析。
- list－返回可用计数器的列表。有关对该列表的描述，请参见第 26 页中的 “硬件计数器列表” 。如果系统不支持硬件计数器溢出分析，则 dbx 会返回一条警告消息。
- counter *counter_definition*... [, *counter_definition*]－计数器定义可以采用下列形式之一，具体取决于处理器是否支持硬件计数器的属性。

  [+] *counter_name*[/ *register_number*][,*interval*]

  [+]*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，可以通过键入不带参数的 collect 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，则它们必须使用不同的寄存器。否则，将列显一条警告消息，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溢出分析数据。如果硬件计数器溢出分析处于启用状态，而且尚未提供 profile 命令，则基于时钟的分析将处于关闭状态。

    另请参见第 46 页中的 “硬件计数器溢出分析的限制” 。

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。

mpitrace *option*

控制对 MPI 跟踪数据的收集。*option* 的允许值包括：

- on－启用对 MPI 调用的跟踪。
- off－禁用对 MPI 调用的跟踪。

缺省情况下，收集器不收集 MPI 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：

- on－打开线程分析器的数据争用检测数据
- off－关闭线程分析器数据
- all－打开所有线程分析器数据
- race－打开线程分析器的数据争用检测数据
- deadlock－收集死锁和潜在死锁数据
- dt*N*－打开特定的线程分析器数据类型，这些数据类型由 dt* 参数指定。

有关线程分析器的更多信息，请参见《Sun Studio 12：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 停止目标进程时对样本的记录。关键字的含义如下所示：

- on－在 dbx 每次停止目标进程时记录样本。
- off－在 dbx 停止目标进程时不记录样本。

缺省情况下，在 dbx 停止目标进程时记录样本。

实验控制子命令

disable

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

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

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样本点。在暂停之前会生成一个样本，在恢复之后会立即生成另一个样本。如果已暂停数据收集功能，则该子命令将被忽略。

resume

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*

记录具有标签 *name* 的样本包。该标签显示在性能分析器的“事件”标签中。

输出子命令

下列子命令定义实验的存储选项。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。

archive *mode*

设置归档实验的模式。*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目标文件复制到实验中。

limit *value*

将所记录的分析数据量限制在 *value* 兆字节。该限制适用于基于时钟的分析数据量、硬件计数器溢出分析数据量和同步等待跟踪数据量之和，但不适用于样本点。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，而且会继续记录样本点。

记录的数据量的缺省限制为 2000 MB。选择该限制的原因是性能分析器无法处理所含数据量大于 2 GB 的实验。要去掉该限制，请将 *value* 设置为 unlimited 或 none。

store *option*

控制实验的存储位置。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。*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，请参见第 48 页中的 “数据的存储位置” 。
- 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

show

显示每个收集器控件的当前设置。

`status`

报告任何打开的实验的状态。

使用 dbx 从正在运行的进程中收集数据

使用收集器可以从正在运行的进程中收集数据。如果进程已受 dbx 的控制，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。

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，如第 60 页中的 “使用 collect 实用程序从正在运行的进程中收集数据”中所述。您还可以向其附加 dbx，收集性能数据，然后从该进程分离并使进程继续运行。如果要为选定的后续进程收集性能数据，则必须将 dbx 附加到每个进程。

▼ 从正在运行且不受 dbx 控制的进程中收集数据：

1 确定程序的进程 ID (process ID, PID)。

如果从命令行启动程序并将其放置在后台中，则其 PID 将由 shell 列显到标准输出中；否则，您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 键入以下内容。

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

如果 dbx 尚未运行，请键入以下内容。

```
% 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《Sun Studio 12：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。

在 dbx 中键入以下内容。

```
(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，因为该库提供了能使数据收集发生的真正函数的包装。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。如果未预装入收集器库，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，请参见第 40 页中的 “使用系统库”。

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 库的环境变量设置

环境变量	值
LD_PRELOAD	libcollector.so
LD_LIBRARY_PATH	/opt/SUNWspro/prod/lib/dbxruntime
LD_LIBRARY_PATH_32	/opt/SUNWspro/prod/lib/dbxruntime
LD_LIBRARY_PATH_64	/opt/SUNWspro/prod/lib/v9/dbxruntime
LD_LIBRARY_PATH_64	/opt/SUNWspro/prod/lib/amd64/dbxruntime

如果 Sun Studio 软件未安装在 /opt/SUNWspro 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。

如果要从已在运行的 MPI 程序收集数据，则必须将一个单独的 dbx 实例附加到每个进程并针对每个进程启用收集器。在将 dbx 附加到 MPI 作业中的多个进程时，每个进程将被停止并在不同的时间重新启动。时间差异可能会更改 MPI 进程间的交互并影响所收集的性能数据。为了使该问题带来的影响最小，一种解决方案是使用 pstop(1) 命令停止所有进程。但是，在将 dbx 附加到这些进程后，必须从 dbx 重新启动这些进程，重新启动进程时会发生时间延迟，这会影响 MPI 进程的同步。另请参见第 68 页中的 “从 MPI 程序收集数据”。

从 MPI 程序收集数据

收集器可以从那些使用消息传递接口 (Message Passing Interface, MPI) 库的多进程程序中收集性能数据。Open MPI 库包含在 Sun HPC ClusterTools™ 7 软件中，Sun MPI 库包含在 ClusterTools 5 和 ClusterTools 6 软件中。可从 `http://www.sun.com/software/products/clustertools/` 获取 ClusterTools 软件。

要在使用 ClusterTools 7 时启动并行作业，请使用 Open Run-Time Environment (ORTE) 命令 `mpirun`。

要在使用 ClusterTools 5 或 ClusterTools 6 时启动并行作业，请使用 Sun Cluster Runtime Environment (CRE) 命令 `mprun`。

有关更多信息，请参见 Sun HPC ClusterTools 文档。

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

由于 MPI 和收集器的实现方式，每个 MPI 进程记录一个单独的实验。每个实验必须具有唯一的名称。实验的存储位置和方式取决于 MPI 作业可用的文件系统类型。有关存储实验的问题将在下一个小节第 68 页中的 “存储 MPI 实验”中讨论。

要从 MPI 作业收集数据，可以在 MPI 下运行 `collect` 命令，也可以在 MPI 下启动 `dbx` 并使用 `dbx collector` 子命令。这些任务将在第 70 页中的 “在 MPI 下运行 collect 命令”和第 70 页中的 “通过在 MPI 下启动 dbx 来收集数据”中讨论。

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，请参见第 48 页中的 “数据的存储位置”。

用来收集性能数据的每个 MPI 进程都创建其自己的实验。当某个 MPI 进程创建实验时，它会锁定实验目录。所有其他 MPI 进程都必须等到该锁定被释放之后才能使用此目录。因此，如果将实验存储到所有 MPI 进程都可以访问的文件系统，则会按顺序创建这些实验；但是如果将实验存储到每个 MPI 进程的本地文件系统，则会并行创建这些实验。

如果允许收集器创建实验名称，则可以避免与实验名称和存储位置有关的问题。请参见下一节第 68 页中的 “缺省的 MPI 实验名称”。

缺省的 MPI 实验名称

如果未指定实验名称，则使用缺省的实验名称。收集器使用 MPI 等级以标准格式 *experiment.m*`.er` 构造实验名称，其中 *m* 是 MPI 等级。如果指定了实验组，则主干 *experiment* 为实验组名称的主干，否则主干为 `test`。无论使用的是公共文件系统还是本

地文件系统，实验名称都是唯一的。因此，如果使用本地文件系统记录实验并将其复制到公共文件系统，则复制这些实验并重新构造任何实验组文件时不必重命名实验。在多数情况下，您应当允许收集器创建实验名称，以确保名称在所有文件系统的范围内是唯一的。

指定非缺省的 MPI 实验名称

如果您将实验存储在公共文件系统上，并以标准格式 *experiment*.*n*.er 指定实验名称，那么当每个实验的 *n* 值递增时，每个实验将被赋予一个唯一的名称。实验是按照 MPI 进程获取实验目录锁的顺序来编号的，因而无法保证与进程的 MPI 等级相对应。如果您将 dbx 附加到正在运行的 MPI 作业中的 MPI 进程，则实验编号由附加的顺序来确定。

如果您将每个实验都存储在其自己的本地文件系统上并指定一个显式的实验名称，则每个实验都可能会获得这个名称。例如，假设您在具有四个单处理器节点的群集中运行 MPI 作业，这四个节点分别标记为 node0、node1、node2 和 node3。每个节点具有一个名为 /scratch 的本地磁盘，您将实验存储在该磁盘的 *username* 目录中。由 MPI 作业创建的实验具有下面的全路径名称。

```
node0:/scratch/username/test.1.er
node1:/scratch/username/test.1.er
node2:/scratch/username/test.1.er
node3:/scratch/username/test.1.er
```

包括节点名称的全名是唯一的，但在每个实验目录中都有一个名为 test.1.er 的实验。如果在 MPI 作业完成后将实验移到公共位置，则必须确保这些名称仍然是唯一的。例如，要将这些实验移到假设可以从所有节点访问的起始目录并重命名这些实验，请键入以下命令。

```
rsh node0 'er_mv /scratch/username/test.1.er test.0.er'
rsh node1 'er_mv /scratch/username/test.1.er test.1.er'
rsh node2 'er_mv /scratch/username/test.1.er test.2.er'
rsh node3 'er_mv /scratch/username/test.1.er test.3.er'
```

对于大型 MPI 作业，可能需要使用脚本将实验移到公共位置。请不要使用 UNIX® 命令 cp 或 mv；应当使用上例中所示的 er_cp 或 er_mv，如第 179 页中的 “处理实验” 中所述。

如果不知道哪些本地文件系统可用，请使用 df -lk 命令或咨询系统管理员。请始终确保用来存储实验的目录已经存在、其定义是唯一的以及不将其用于任何其他实验，还要确保文件系统具有足够的空间来容纳这些实验。有关如何估计所需空间的信息，请参见第 50 页中的 “估计存储要求”。

注 – 除非您能够访问用于运行实验的装入对象和源文件或者拥有具备相同路径和时间戳的副本，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。

在 MPI 下运行 collect 命令

要在 MPI 的控制下使用 collect 命令收集数据，，请使用与 ClusterTools 版本相对应的语法，如下所示。

在 Sun HPC ClusterTools 7 上：

```
% mpirun -np n
collect [collect-arguments] program-name
 [program-arguments]
```

在 Sun HPC ClusterTools 6 和更低版本上：

```
% mprun -np n
collect [collect-arguments] program-name
 [program-arguments]
```

在以上两种情况下，*n* 都表示要由 MPI 创建的进程数。此过程会创建 *n* 个单独的 collect 实例，每个实例都记录一个实验。有关实验的存储位置和方法的信息，请阅读第 48 页中的 “数据的存储位置”一节。

为了确保将来自不同 MPI 运行的实验集存储在不同的位置，可以使用 -g 选项为每个 MPI 运行创建一个实验组。实验组应当存储在所有 MPI 进程都可以访问的文件系统上。创建实验组还有助于将单个 MPI 运行的实验集装入性能分析器。如果不创建组，则可以使用 -d 选项为每个 MPI 运行指定一个单独的目录。

通过在 MPI 下启动 dbx 来收集数据

要在 MPI 的控制下启动 dbx 并收集数据，请使用下面的语法。

在 Sun HPC ClusterTools 7 上：

```
% mpirun -np n dbx program-name < collection-script
```

在 Sun HPC ClusterTools 6 或更低版本上：

```
% mprun -np n dbx program-name < collection-script
```

在以上两种情况下，*n* 都表示要由 MPI 创建的进程数，*collection-script* 都表示包含设置和启动数据收集所必需的命令的 dbx 脚本。此过程会创建 *n* 个单独的 dbx 实例，每个实例都记录其中一个 MPI 进程上的实验。如果未定义实验名称，则使用 MPI 等级对实验进行标记。有关实验的存储位置和方法的信息，请阅读第 68 页中的 “存储 MPI 实验”一节。

通过在程序中使用收集脚本以及对 MPI_Comm_rank() 的调用，可以使用 MPI 等级来对实验进行命名。例如，在 C 程序中插入以下行。

```
ier = MPI_Comm_rank(MPI_COMM_WORLD,&me);
```

在 Fortran 程序中插入以下行。

```
call MPI_Comm_rank(MPI_COMM_WORLD, me, ier)
```

例如，如果将该调用插入到第 17 行，则可以使用类似以下内容的脚本。

```
stop at 18
run program-arguments
rank=$[me]
collector enable
collector store filename experiment.$rank.er
cont
quit
```

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。创始实验位于 ppgsz 可执行文件上，我们不需要关注它。如果您的路径找到 32 位版本的 ppgsz，并且实验在支持 64 位进程的系统上运行，则首先要做的是针对它的 64 位版本执行 exec，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，则前两个后续实验分别命名为 _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 并存储在创始实验之下。通过针对 test.1.er/_x1_f1_x3.er 调用分析器或 er_print 实用程序，可以直接处理该目标。

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。

第 4 章

性能分析器工具

性能分析器是一种图形数据分析工具，可以分析收集器使用 `collect` 命令、IDE 或 `dbx` 中的 `collector` 命令收集的性能数据。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。分析器具有命令行版本，即 `er_print` 实用程序，将在第 6 章中对其进行介绍。

启动性能分析器

要启动性能分析器，请在命令行中键入以下内容：

% **analyzer** [*control-options*] [*experiment-list*]

或者，使用 IDE 中的资源管理器导航到某个实验并将其打开。*experiment-list* 命令参数是空格分隔的实验名称、实验组名称或这两者的列表。

您可以在命令行中指定多个实验或实验组。如果指定的实验中具有后续实验，将自动装入所有后续实验，但是禁用后续实验的数据显示。要装入个别后续实验，必须明确指定每个实验或创建实验组。

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`。

也可以使用分析器窗口中的“文件”菜单来添加实验或实验组。要打开所记录的后续进程上的实验，必须在“打开实验”对话框（或“添加实验”对话框）中键入文件名称，因为文件选择器不允许将实验作为目录打开。

分析器显示多个实验时，无论这些实验是如何装入的，都会聚集来自所有实验的数据。

可以在“打开实验”对话框或“添加实验”对话框中单击要装入的实验或实验组的名称来预览该实验或实验组。

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*]

分析器启动并显示“性能工具收集”窗口，显示指定的目标及其参数以及用于收集实验的设置。有关详细信息，请参见第 89 页中的 “记录实验”。

分析器选项

这些选项控制分析器的行为，可分为三组：

- Java 选项
- 控制选项
- 信息选项

Java 选项

-j | --jdkhome *jvm-path*

指定运行分析器的 JVM 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）。如果上述两个变量均未设置，则缺省路径为 Sun Studio 安装程序安装 Java™ 2 软件开发工具包的位置。如果未安装 SDK，将使用在用户路径中找到的 JVM。使用 -j 选项可覆盖所有缺省路径。

-J *jvm-options*

指定 JVM 选项。

控制选项

-f | --fontsize *size*

指定要在分析器 GUI 中使用的字体大小。

-v | --verbose

启动之前列显版本信息和 Java 运行时参数。

信息选项

这些选项不调用性能分析器 GUI，但将有关 analyzer 的信息列显至标准输出。以下每个选项都是独立选项；它们不能与其他 analyzer 选项、目标或实验列表参数结合使用。

-V | --version

列显版本信息并退出。

-? | --h | --help

列显用法信息并退出。

分析器缺省设置

启动时，分析器使用名为 .er.rc 的资源文件来确定各种设置的缺省值。首先读取系统范围的 er.rc 缺省值文件，然后读取用户的起始目录中的 .er.rc 文件（如果有），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 中的缺省值覆盖起始目录缺省值和系统缺省值。.er.rc 文件由分析器和 er_print 实用程序使用。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。

有关 .er.rc 文件的更多信息，请参见第 90 页中的 “分析器缺省设置” 一节。有关使用 er_print 命令设置缺省值的信息，请参见第 119 页中的 “设置缺省值的命令”和第 120 页中的 “仅为性能分析器设置缺省值的命令”。

性能分析器 GUI

分析器窗口具有菜单栏、工具栏以及一个包含用于显示各种数据的标签的拆分窗格。

菜单栏

菜单栏包含“文件”菜单、“视图”菜单、“时间线”菜单和“帮助”菜单。

“文件”菜单用来打开、添加和删除实验和实验组。利用“文件”菜单可以通过性能分析器 GUI 来收集实验数据。有关使用性能分析器收集数据的详细信息，请参阅第 89 页中的 “记录实验”。您也可以通过“文件”菜单创建映射文件，用以优化可执行文件的大小或优化其有效的高速缓存行为。有关映射文件的更多详细信息，请参阅第 89 页中的 “生成映射文件和函数重新排序”。

利用“视图”菜单可以配置实验数据的显示方式。

正如其名称所表示的那样，“时间线”菜单可以帮您浏览时间线显示，第 76 页中的 “分析器数据显示”对其进行了介绍。

“帮助”菜单提供性能分析器的联机帮助、新功能摘要、快速参考和快捷键以及故障排除。

工具栏

工具栏提供了几组图标作为快捷方式，此外包含“查找”功能以帮助您在标签中定位文本或突出显示的行。有关“查找”功能的更多详细信息，请参阅第 87 页中的 “查找文本和数据”。

分析器数据显示

性能分析器使用拆分窗口将数据显示在两个窗格中。每个窗格都含有标签，以便您可以为同一实验或实验组选择不同的数据显示。

数据显示，左窗格

左窗格按标签出现的顺序显示主要分析器显示中的标签：

- “争用”标签
- “死锁”标签
- “函数”标签
- “调用者与被调用者”标签
- “双重数据源”标签
- "Source-Disassembly"（源-反汇编）标签
- “源”标签
- “行”标签
- “反汇编”标签
- "PC" 标签
- “时间线”标签
- “泄漏列表”标签
- “数据对象”标签
- “数据布局”标签
- "Inst-Freq" 标签
- “统计数据”标签
- “实验”标签
- 各种“内存对象”标签
- 各种“索引对象”标签

如果调用分析器时没有指定目标，系统将提示您打开一个实验。

缺省情况下，选中第一个可见的标签。仅显示适用于装入实验中的数据的标签。

打开实验时某个标签是否显示在分析器窗口的左窗格中取决于启动分析器时读取的 .er.rc 文件中的标签指令以及标签是否适用于实验中的数据。您可以使用“设置数据表示”对话框中的“标签”标签（请参见第 86 页中的““标签”标签”）来选择要为实验显示的标签。

“ 争用 ”标签

“争用”标签显示在数据争用实验中检测到的所有数据争用列表。有关更多信息，请参见《Sun Studio 12：线程分析器用户指南》。

“ 死锁 ”标签

“死锁”标签显示在死锁实验中检测到的所有死锁列表。有关更多信息，请参见《Sun Studio 12：线程分析器用户指南》。

“ 函数 ”标签

“函数”标签显示包含函数及其度量的列表。度量是根据在实验中收集的数据得出的。度量可以是独占的，也可以是包含的。独占的度量在函数本身内部使用。包含的度量在函数及其所调用的所有函数中使用。

collect(1) 手册页中提供了每种收集的数据的可用度量列表。仅列出具有非零度量的函数。

时间度量显示为秒，精度可达到毫秒。百分比精确到 0.01%。如果某个度量值正好为零，则其时间和百分比显示为 "0"。如果值不是恰好为零，但小于精度，则其值将显示为 "0.000"，其百分比为 "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以及从各种 .er.rc 文件中读取的缺省设置。初始安装性能分析器时，缺省值如下所示：

- 对于基于时钟的分析，缺省设置包含包含的和独占的用户 CPU 时间。
- 对于同步延迟跟踪，缺省设置包含包含的同步等待计数和包含的同步时间。
- 对于硬件计数器溢出分析，缺省设置包含包含的和独占的次数（对于周期计数的计数器）或事件计数（对于其他计数器）。
- 对于堆跟踪，缺省设置包含堆泄漏和泄漏的字节。

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。

可以更改或重新组织显示的度量；有关详细信息，请参见联机帮助。

要搜索函数，请使用**查找**工具。有关“查找”工具的更多详细信息，请参阅第 87 页中的“查找文本和数据”。

要选择单个函数，请单击该函数。

要选择在标签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。

要选择标签中不是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。

单击工具栏中的“编写过滤子句”按钮时，将打开“过滤”对话框，同时已选中其中的“高级”标签并且“过滤子句”文本框装入反映“函数”标签中的选择的过滤子句。

“调用者与被调用者”标签

“调用者与被调用者”标签在中间的窗格中显示所选择的函数，该函数的调用者显示在上面的窗格中，该函数的被调用者显示在下面的窗格中。

除了显示每个函数独占的和包含的度量值之外，该标签还显示归属度量。对于所选择的函数，归属度量表示该函数的独占度量。对于被调用者，归属度量表示被调用者的包含度量中归属于中心函数调用的那一部份。被调用者和所选函数的归属度量之和等于所选函数的包含度量。

对于调用者，归属度量表示所选函数的包含度量中归属于调用者的调用的那一部分。所有调用者的归属度量之和同样应等于所选函数的包含度量。

可以更改或重新组织“调用者与被调用者”标签中显示的度量；有关详细信息，请参见联机帮助。

单击调用者或被调用者窗格中的函数将选择该函数，窗口的内容将被刷新，从而使所选函数显示在中间的窗格中。

“双重数据源”标签

“双重数据源”标签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标签。

"Source-Disassembly"（源-反汇编）标签

"Source-Disassembly"（源-反汇编）标签在上面的窗格中显示带注释的源，在下面的窗格中显示带注释的反汇编。缺省情况下该标签不可见。可以使用“视图”菜单中的“设置数据表示”选项来添加 "Source-Disassembly"（源-反汇编）标签。

“源”标签

如果可用，“源”标签显示包含所选函数源代码的文件，每个源代码行都带有性能度量注释。源代码列标题中显示源文件、对应的目标文件以及装入对象的全名。少数情况下，会使用同一源文件编译多个目标文件，此时“源”标签显示包含所选函数的目标文件的性能数据。

分析器在可执行文件中记录的绝对路径名下查找包含所选函数的文件。如果文件不在此处，分析器尝试在当前工作目录中查找具有相同基本名称的文件。如果移动了源，或者实验记录在其他文件系统中，则可以设置从当前目录指向实际源位置的符号链接，以便查看带注释的源。

在“函数”标签中选择一个函数并且“源”标签打开时，显示的源文件为该函数的缺省源上下文。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>

函数可能具有其他源上下文，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。如果存在其他源上下文，则缺省源上下文的开头将包含指示其他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，其他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其他源上下文中定义的函数。

源代码中插入了所有选择显示的编译器注释。可以在“设置数据表示”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“源”标签中显示的度量，有关详细信息，请参见联机帮助。

度量等于或大于该度量针对源文件中任意行设置的最大阈值百分比的行将突出显示，以便于查找重要的行。可以在“设置数据表示”对话框中设置阈值。可以在 `.er.rc` 缺省值文件中设置缺省阈值。滚动条的旁边将显示勾号（对应于源文件中超过阈值的行的位置）。例如，如果源文件的末尾附近有两个超过阈值的行，源窗口的底部附近的滚动条附近将显示两个勾号。将滚动条定位到勾号的旁边将使源行显示在源窗口中，从而显示对应的超过阈值的行。

“行”标签

“行”标签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“行”标签中选择一个行，则会在“摘要”标签中显示该行的所有度量。如果从“行”标签中选择一个行后选择“源”或“反汇编”标签，则会将显示定位到适当的行。

“反汇编”标签

“反汇编”标签显示包含所选函数的目标文件的反汇编列表，带有针对每条指令的性能度量注释。

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“反汇编”标签中查找源文件的算法与在“源”标签中使用的算法相同。

与“源”标签相同，“反汇编”标签中显示索引行。但与“源”标签不同的是，其他源上下文的索引行不能直接用于导航。此外，其他源上下文的索引行显示在 `#included` 或内联代码插入位置的开头，而不是仅在“反汇编”视图的开头列出。来自其他文件的 `#included` 或内联代码显示为原始反汇编指令，不插入源代码。但是，将光标置于这些指令之一上并选择“源”标签可以打开包含 `#included` 或内联代码的源文件。显示此文件时选择“反汇编”标签将在新上下文中打开“反汇编”视图，从而显示插入了源代码的反汇编代码。

可以在“设置数据表示”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。

分析器突出显示度量等于或大于特定于度量的阈值的行，以便于查找重要的行。可以在“设置数据表示”对话框中设置阈值。可以在 `.er.rc` 缺省值文件中设置缺省阈值。与“源”标签相同，滚动条旁边显示有勾号（对应于反汇编代码中超过阈值的行的位置）。

"PC" 标签

"PC" 标签显示包含程序计数器 (program counter, PC) 及其度量的列表。PC 标有其所来自的函数及在该函数中的偏移。在 "PC" 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，显示为装入对象的单个条目。在 "PC" 标签中选择一个行将在“摘要”标签中显示该 PC 的所有度量。从 "PC" 标签中选择一个行后选择“源”或“反汇编”标签会将显示定位到适当的行。

有关 PC 的更多信息，请参见第 135 页中的 “调用栈和程序执行” 一节。

“时间线”标签

“时间线”标签以时间的函数的形式显示收集器记录的事件和样本点图。数据显示在水平栏中。对于每个实验，都有一个样本数据栏和一组用于每个 LWP 的栏。LWP 栏中包含一个用于每个记录的数据类型的栏：基于时钟的分析、硬件计数器溢出分析、同步跟踪、堆跟踪以及 MPI 跟踪。

包含样本数据的栏显示在每个样本的每个微状态中花费时间的彩色编码表示。样本显示为时间段，原因是样本点上的数据表示该样本点和上一个样本点之间所花费的时间。单击样本将在“事件”标签中显示该样本的数据。

分析数据栏或跟踪数据栏显示每个记录的事件的事件标记。事件标记包含随事件记录的调用栈的彩色编码表示，如彩色矩形栈。单击事件标记中的彩色矩形将选择对应的

函数和 PC 并在“事件”标签中显示该事件和该函数的数据。“事件”标签和“图例”标签中的选定内容会突出显示，并且选择“源”标签或“反汇编”标签可以将标签显示定位在调用栈中的该帧对应的行上。

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。在以上两种情况下，绘制的事件下将显示一个小的灰色勾号，以表示重叠。

利用“设置数据表示”对话框的“时间线”标签可以执行以下操作：更改显示的特定于事件的数据的类型；选择线程、LWP 或 CPU 特定于事件的数据的显示；选择在根或叶处对齐调用栈表示；选择显示的调用栈的级别数。

可以更改“时间线”标签中显示的特定于事件的数据的类型，也可以更改映射到所选函数的颜色。有关使用“时间线”标签的详细信息，请参阅联机帮助。

“泄漏列表”标签

“泄漏列表”标签显示两行，上面一行表示泄漏，下面一行表示分配。每一行包含一个调用栈，与“时间线”标签中显示的调用栈类似；在中间，上面有一个条，其长度与泄漏或分配的字节数成比例，下面也有一个条，其长度与泄漏或分配数成比例。

选择泄漏或分配将在“泄漏”标签中显示所选的泄漏或分配的数据，并且在调用栈中选择一个帧（与在“时间线”标签中相同）。

可以通过在“设置数据表示”对话框的“标签”标签中选择“泄漏列表”标签来显示该标签（请参见第 86 页中的““标签”标签”）。仅在一个或多个装入的实验包含堆跟踪数据时才能使“泄漏列表”标签可见。

“数据对象”标签

“数据对象”标签显示数据对象及其度量的列表。该标签仅适用于已启用主动回溯选项的硬件计数器溢出实验以及在 C 编译器中使用 `-xhwcprof` 选项编译的源文件。

可以通过在“设置数据表示”对话框的“标签”标签中选中该标签来显示它（请参见第 86 页中的““标签”标签”）。仅在一个或多个装入的实验包含数据空间分析时才能使“数据对象”标签可见。

该标签显示针对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。

要选择在标签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。

要选择在标签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。

单击工具栏上的“编写过滤子句”按钮时，将打开“过滤”对话框，此时已选中对话框的“高级”标签并且在“过滤子句”文本框中装入了反映“数据对象”标签中的选择的过滤子句。

“ 数据布局 ”标签

“数据布局”标签显示所有程序数据对象（带有从数据获取的度量数据）的带注释的数据对象布局。总的来说，标签中显示的布局按照结构的数据排序度量值进行排序。标签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。

可以通过在“设置数据表示”对话框的“标签”标签中选择“数据布局”标签来显示它（请参见第 86 页中的 ““标签”标签”）。与“数据对象”标签相同，仅在一个或多个装入的实验包含数据空间分析时才可使“数据布局”标签可见。

要选择单个数据对象，请单击该对象。

要选择在标签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。

要选择在标签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。

单击工具栏上的“编写过滤子句”按钮时，将打开“过滤”对话框，此时已选中对话框的“高级”标签，并且在“过滤子句”文本框中装入了反映“数据布局”标签中的选择的过滤子句。

"Inst-Freq" 标签

"Inst-Freq"（即“指令-频率）标签显示计数数据实验中各类指令执行频率的摘要。该标签还显示有关装入、存储和浮点指令的执行频率的数据。此外，该标签还包含有关取消的指令和分支延迟槽中的指令的信息。

“ 统计数据 ”标签

“统计数据”标签显示所选的实验和样本的各种系统度量的总和。总和后面是每个实验所选样本的统计信息。有关显示的统计信息的信息，请参见 `getrusage`(3C) 和 `proc`(4) 手册页。

“ 实验 ”标签

“实验”标签分为两个面板。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“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分析器会话的错误消息和警告消息。

"Index"（索引）标签

每个 "Index"（索引）标签显示归属于各种索引对象（如线程、Cpu 和 秒）的数据的度量值。由于 "Index"（索引）对象不是分层结构，因此不显示包含的和独占的度量。对于每一种类型，只显示一个度量。

预定义了多个 "Index"（索引）标签：线程、Cpu、样本和秒。您可以定义定制的索引对象，方法是在“设置数据表示”对话框中单击“添加定制索引标签”按钮，以打开“添加索引对象”对话框。

使用每个“索引”标签顶部的单选按钮可以选择“文本”显示或“图形”显示。“文本”显示类似于“数据对象”标签中的显示，并且使用同样的度量设置。“图形”显示以图形的形式显示每个索引对象的相关值，每个度量用一个单独的直方图表示，按数据排序度量排序。

单击工具栏中的“过滤数据”按钮时，将打开“过滤数据”对话框。单击“高级”标签，“过滤子句”文本框将装入反映“索引对象”标签中的选择的过滤子句。

“内存对象”标签

每个“内存对象”标签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“设置数据表示”对话框的“标签”标签中选择要显示其标签的内存对象。可以显示任意数量的“内存对象”标签。

预定义了各种“内存对象”标签。您可以定义定制的内存对象，方法是在“设置数据表示”对话框中单击“添加定制对象”按钮，以打开“添加内存对象”对话框。

使用每个“内存对象”标签上的单选按钮，可以选择“文本”显示或“图形”显示。“文本”显示类似于“数据对象”标签中的显示，并且使用同样的度量设置。“图形”显示以图形的形式显示每个内存对象的相关值，每个度量用一个单独的直方图表示，按数据排序度量排序。

单击工具栏上的“编写过滤子句”按钮时，将打开“过滤”对话框，此时已选中对话框的“高级”标签，并且在“过滤子句”文本框中装入了反映“内存对象”标签中的选择的过滤子句。

数据显示，右窗格

右窗格包含“摘要”标签、“事件”标签、“争用详细信息”标签、“死锁详细信息”标签和“泄漏”标签。缺省情况下显示“摘要”标签。

“ 摘要 ”标签

“摘要”标签同时以值和百分比的形式显示所选函数或装入对象的所有记录的度量，此外还显示所选函数或装入对象的信息。无论何时在任何标签中选择了新的函数或装入对象，都将更新“摘要”标签。

“ 事件 ”标签

“事件”标签显示在“时间线”标签中选择的事件的详细数据，包括事件类型、叶函数、LWP ID、线程 ID 和 CPU ID。数据面板的下方显示调用栈，栈中的每个函数都使用彩色编码。单击调用栈中的某个函数将选中该函数。

在“时间线”标签中选中某个样本时，“事件”标签将显示样本编号、样本的开始时间和结束时间、微状态及在每个微状态中所花费的时间和彩色编码。

“ 泄漏 ”标签

“泄漏”标签显示“泄漏列表”标签中所选的泄漏或分配的详细数据。在数据面板的下方，“泄漏”标签显示检测到所选的泄漏或分配时的调用栈。单击调用栈中的函数将选中该函数。

“ 争用详细信息 ”标签

“争用详细信息”标签显示“争用”标签中所选的数据争用的详细数据。有关更多信息，请参见《Sun Studio 12：线程分析器用户指南》。

“ 死锁详细信息 ”标签

“死锁详细信息”标签显示“死锁”标签中所选的死锁的详细数据。有关更多信息，请参见《Sun Studio 12：线程分析器用户指南》。

设置数据表示选项

您可以通过“设置数据表示”对话框控制数据的表示。要打开此对话框，请单击工具栏中的“设置数据表示”按钮或选择“视图”→“设置数据表示”。

“设置数据表示”对话框具有一个包含多个标签的窗格，这些标签如下所示：

- 度量
- 排序

- 源/反汇编
- 格式
- 时间线
- 搜索路径
- 路径映射
- 标签

对话框中有“保存”按钮，使用该按钮可以存储当前设置，包括任何自定义的内存对象。

注 – 由于分析器、`er_print` 实用程序和 `er_src` 实用程序的缺省值是通过通用的 `.er.rc` 文件设置的，因此在“设置数据表示”对话框中保存更改将会影响 `er_print` 实用程序和 `er_src` 实用程序的输出。

“ 度量 ”标签

“度量”标签显示所有可用的度量。每个度量具有多个复选框（在一个或多个列中），复选框标有**时间**、**值** 和 `%`，具体取决于度量的类型。或者，也可以不设置个别度量，而是通过选中或取消选中对话框最底下一行中的复选框，然后单击“应用于所有度量”按钮，来一次设置所有度量。

“ 排序 ”标签

“排序”标签显示所显示的度量的顺序以及度量的排序依据选项。

“ 源/反汇编 ”标签

“源/反汇编”标签显示一个复选框列表，您可以使用该列表来选择显示的信息，如下所示：

- 源列表和反汇编列表中显示的编译器注释
- 在源列表和反汇编列表中突出显示重要行的阈值
- 在反汇编列表中插入的源代码
- 反汇编列表中的源行上的度量
- 反汇编列表中十六进制形式的指令显示。

“ 格式 ”标签

“格式”标签提供 C++ 函数名称和 Java 方法名称的长名形式、短名形式或修整名称形式选项。如果选择“将 SO 名称附加到函数名称” 复选框，则函数或方法所在的共享对象的名称将会附加到函数名称或方法名称。

此外，“格式”标签还显示**用户**、**专家**或**机器**“视图模式”选项。“视图模式”设置控制对 Java 实验和 OpenMP 实验的处理。

对于 Java 实验：

- 用户模式显示 Java 线程的 Java 调用栈，不显示内务处理线程。
- 专家模式在执行用户的 Java 代码时显示 Java 线程的 Java 调用栈，在执行 JVM 代码或 JVM 软件未报告 Java 调用栈时显示机器调用栈。它显示内务处理线程的机器调用栈。
- 机器模式显示所有线程的机器调用栈。

对于 OpenMP 实验：

- 用户模式和专家模式显示协调的主线程调用栈和从属线程调用栈，并在 OpenMP 运行时执行特定操作时添加名称格式为 `<OMP-*>` 的特殊函数。
- 机器模式显示所有线程的机器调用栈。

对于所有其他实验，所有三种模式显示同样的数据。

“时间线”标签

“时间线”标签显示以下内容：显示的事件特定数据的类型选项，线程、LWP 或 CPU 事件特定数据的显示，调用栈表示在根或叶的对齐，显示的调用栈的级别数。

“搜索路径”标签

利用“搜索路径”标签可以管理用来搜索源文件和目标文件的目录列表。特殊名称 `$expts` 表示装入的实验；所有其他名称应为文件系统中的路径。

“路径映射”标签

使用“路径映射”标签可以将文件路径的前面部分从一个位置映射到另一个位置。您可以指定一组前缀对：原始前缀和新前缀。这样，对于给定的路径，可将路径从原始前缀映射到新前缀。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。

“标签”标签

您可以使用“设置数据表示”对话框的“标签”标签来选择要在分析器窗口中显示的标签。

“标签”标签列出适用于当前实验的标签。标准标签列在左边的列中。“索引”标签列在中间的列中，定义的“内存”标签列在右边的列中。

在左边的列中，单击复选框可以选择或取消选择显示标准标签。

在中间的列中，单击复选框可以选择或取消选择显示“索引”标签。预定义的“索引”标签为“线程”、"CPU"、“样本”和“秒”。要添加其他索引对象的标签，请单击“添加定制索引标签”按钮以打开“添加索引对象”对话框。在“对象名称”文本框中，键入新对象的名称。在“公式”文本框中，键入用来将记录的物理地址或虚拟地址映射到对象索引的索引表达式。有关索引表达式规则的信息，请参见第 110 页中的“`indxobj_define` *indxobj_type index_exp*”。

在右边的列中，单击复选框可以选择或取消选择显示“内存对象”标签。要添加定制对象，请单击“添加定制对象”按钮以打开“添加内存对象”对话框。在“对象名称”文本框中，键入新定制内存对象的名称。在“公式”文本框中，键入用来将记录的物理地址或虚拟地址映射到对象索引的索引表达式。有关索引表达式规则的信息，请参见第 110 页中的“mobj_define *mobj_type index_exp*”。

添加定制索引对象或内存对象后，将会向“标签”标签中添加该对象的复选框，缺省情况下，该复选框是选中的。

保存数据表示选项

“设置数据表示”对话框中有“保存”按钮，用以存储当前设置。

注 – 由于分析器、er_print 实用程序和 er_src 实用程序的缺省值是通过通用的 .er.rc 文件设置的，因此在分析器的“设置数据表示”对话框中保存更改会影响 er_print 实用程序和 er_src 实用程序的输出。

查找文本和数据

分析器具有可通过工具栏访问的“查找”工具，该工具在组合框中提供了两个用于搜索给定目标的选项。您可以在“函数”标签或“调用者与被调用者”标签的“名称”列以及“源”标签和“反汇编”标签的“代码”列中搜索文本。您可以在“源”标签和“反汇编”标签中搜索高度量项。包含高度量项的行上的度量值以绿色突出显示。使用“查找”字段旁边的箭头按钮可以向上或向下搜索。

显示或隐藏函数

缺省情况下，“函数”标签和“调用者与被调用者”标签中显示每个装入对象中的所有函数。您可以使用“显示/隐藏函数”对话框隐藏装入对象中的所有函数；有关详细信息，请参见联机帮助。

隐藏装入对象中的函数时，“函数”标签和“调用者与被调用者”标签显示表示来自装入对象中的所有函数的聚集的单个条目。同样，“行”标签和 "PC" 标签也显示聚集了装入对象所有函数中 PC 的单个条目。

同过滤相比，与隐藏的函数对应的度量仍在所有显示中以某种形式表示。

过滤数据

缺省情况下，所有实验、所有样本、所有线程、所有 LWP 和所有 CPU 的数据全部显示在每个标签中。可以使用“过滤数据”对话框选择数据的某个子集。

“过滤数据”对话框中有一个“简单”标签和一个“高级”标签。在“简单”标签中，您可以选择您要过滤其数据的实验。然后，您可指定您要显示其度量的样本、线程、LWP 和 CPU。在“高级”标签中，您可以指定针对您要在显示中包括的任何数据记录值为真的过滤表达式。有关在过滤表达式中使用的语法的信息，请参见第 122 页中的“表达式语法”。

当在分析器的“函数”标签、“数据布局”标签、“数据对象”标签或“内存对象”标签中作出选择后，单击工具栏上的“编写过滤子句”按钮将打开“过滤数据”对话框的“高级”标签并在“过滤子句”文本框中装入反映选择的子句。

有关使用“过滤数据”对话框的详细信息，请参阅联机帮助。

实验选择

装入多个实验时，分析器允许按实验过滤。实验可以单个装入，也可以通过指定实验组来装入。

样本选择

样本从 1 到 *N* 进行编号，您可以选择样本的任意集合。选择包含逗号分隔的样本编号或范围（例如 1–5）列表。

线程选择

线程从 1 到 *N* 进行编号，您可以选择线程的任意集合。选择包含逗号分隔的线程编号或范围列表。线程的分析数据仅涵盖运行中 LWP 实际调度线程的部分。

LWP 选择

LWP 从 1 到 *N* 进行编号，您可以选择 LWP 的任意集合。选择包含逗号分隔的 LWP 编号或范围列表。如果记录了同步数据，则报告的 LWP 是同步事件入口处的 LWP，可能与同步事件出口处的 LWP 不同。

在 Linux 系统中，线程和 LWP 同义。

CPU 选择

如果记录 CPU 信息 (Solaris OS)，可以选择 CPU 的任意集合。选择包含逗号分隔的 CPU 编号或范围列表。

记录实验

使用目标名称和目标参数调用分析器时，分析器将启动并打开“性能工具收集”窗口，利用该窗口，可以在指定的目标上记录实验。如果调用分析器时没有指定参数或指定了实验列表，您可以通过选择“文件”→“收集实验”打开“性能工具收集”窗口以记录新的实验。

“性能工具收集”窗口的“收集实验”标签有一个面板，您可以使用该面板指定目标、其参数以及用于运行实验的各种参数。它们对应于 `collect` 命令中的可用选项，如第 3 章中所述。

紧靠面板下方是一个“预览命令”按钮和一个文本框。单击该按钮时，将在文本框中填入单击“运行”按钮时使用的 `collect` 命令。

在“要收集的数据”标签中，您可以选择要收集的数据的类型。

“输入/输出”标签有两个面板：一个接收收集器自身的输出，另一个用于接收进程的输出。

利用一组按钮，可以执行以下操作：

- 运行实验
- 终止运行
- 在运行过程中向进程发送“暂停”、“恢复”和“样本”信号（在指定了相应的信号时启用）。
- 关闭窗口。

如果关闭窗口时有实验正在运行，该实验将继续运行。重新打开窗口时，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。

生成映射文件和函数重新排序

除分析数据之外，分析器还提供了函数重新排序功能。分析器可以根据实验中的数据生成映射文件，当将该映射文件与静态链接程序 (`ld`) 一起使用以重新链接应用程序时，可以创建工作集更小或指令高速缓存性能更高或两者兼有的可执行文件。

映射文件中记录的用来重新排序可执行文件中函数的函数顺序由用于对函数列表进行排序的度量决定。通常使用“独占用户 CPU 时间”或“独占 CPU 周期时间”生成映射文件。某些度量（例如来自同步延迟或堆跟踪、名称或地址的度量）不会生成映射文件的有意义的排序。

分析器缺省设置

分析器处理当前目录中的 `.er.rc` 文件（如果有）中的指令、起始目录中的 `.er.rc` 文件（如果有）中的指令以及系统范围内的 `.er.rc` 文件中的指令。这些文件可以包含将实验装入分析器时其标签可见的默认设置。由 `er_print` 命令针对相应的报告来命名各个选项（“实验”标签和“时间线”标签除外）。

`.er.rc` 文件还可以包含用于度量、排序、指定编译器注释选项以及突出显示源和反汇编输出阈值的缺省设置。它们还指定“时间线”标签、名称格式以及“视图”模式的缺省设置。这些文件还可以包含用于控制源文件和目标文件搜索路径或路径映射的指令。

`.er.rc` 文件还可以包含定制内存对象和索引对象的定义。

`.er.rc` 文件还可以包含 `en_desc` 模式设置，用来控制读取创始实验时是否选择并读取后续实验。`en_desc` 设置可以为 `on`、`off` 或 =*regexp*，分别用来指定读取并装入所有后续实验、不读取或装入任何后续实验或读取并装入其沿袭或可执行文件名与给定的正则表达式匹配的后续实验。

在分析器 GUI 中，可以通过单击“设置数据表示”对话框（可通过“视图”菜单打开）中的“保存”按钮来保存 `.er.rc` 文件。通过“设置数据表示”对话框保存 `.er.rc` 文件不仅影响后续的分析器调用，也会影响 `er_print` 实用程序和 `er_src` 实用程序。

第 5 章

内核分析

本章介绍如何在 Solaris OS 运行负载时使用 Sun Studio 性能工具分析内核。如果在 Solaris 10 OS 上运行 Sun Studio 软件，则可以进行内核分析。在 Solaris 9 OS 和 Linux 系统上不可进行内核分析。

内核实验

可以使用 er_kernel 实用程序记录内核分析数据。

er_kernel 实用程序使用 DTrace 驱动程序，该驱动程序是 Solaris OS 中内置的综合动态跟踪工具。

er_kernel 实用程序捕获内核分析数据，并以与用户分析数据相同的格式将数据记录为分析器实验。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、调用者-被调用者数据、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 Solaris OS 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 驱动程序的访问，才能使用 er_kernel 实用程序进行内核分析。

通常，DTrace 驱动器仅限于用户 root 使用。要以 root 之外的用户身份运行 er_kernel 实用程序，必须具有分配的特定权限，而且是组 sys 的成员。要分配必要的权限，请将以下行添加到文件 /etc/user_attr 中：

```
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 er_kernel 命令的完整描述，请参见 er_kernel (1) 手册页。

▼ 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 中运行所需的任意负载。

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）加载到性能分析器或 er_print 实用程序中。

内核时钟分析将生成一个标有“KCPU 周期”的性能度量。在性能分析器中，对于“函数”标签中的内核函数、“调用者-被调用者”标签中的调用者与被调用者以及“反汇编”标签中的指令，将显示该性能度量。“源”标签不显示数据，因为附带的内核模块通常不包含文件和行符号表信息 (stab)。

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。

如果您希望在屏幕上列显有关运行的更多信息，则可以添加 -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。

▼ 在有负载时的分析

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
```

2 **通过键入以下内容来分析实验：**

```
% analyzer ktest.1.er
```

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。实验显示运行负载期间以及之前和之后的静止期内Solaris OS的行为。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。

▼ 一起分析内核和负载

如果有要用作负载的单个程序，并希望同时看到该负载的分析数据和内核分析数据：

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
```

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。时间线允许您查看两个实验之间的相关性。

注–要将脚本用作负载，并分析其各个部分，请使用相应的参数在脚本内的不同命令前放置collect命令。

分析特定的进程或内核线程

您可以通过一个或多个-T参数来调用er_kernel实用程序，以指定分析特定的进程或线程：

- -T *pid*/ *tid*，用于特定的进程和内核线程
- -T 0/ *did*，用于特定的纯内核线程

在为目标线程调用er_kernel实用程序之前，必须已经创建了目标线程。

如果指定一个或多个-T参数，将生成标有Kthr Time的附加度量。并为分析的所有线程捕获数据，而不管是否在CPU上运行。特殊的单帧调用栈用于指示进程是已挂起（函数<SLEEPING>）还是正在等待CPU（函数<STALLED>）。

Kthr Time 度量高、但 KCPU 周期度量低的函数，是为已分析的线程花费很长时间等待某些其他事件的函数。

分析内核分析数据

内核实验中记录的几个字段与用户模式实验中相同字段的含义不同。用户模式实验仅包含单个进程 ID 的数据；而内核实验的数据可能适用于许多不同进程 ID。分析器中一些字段标签在这两种类型的实验中所具有的不同含义，可以在下表中更好地表示出来：

表 5–1 分析器中内核实验的字段标签含义

分析器标签	用户模式实验中的含义	内核实验中的含义
LWP	用户进程 LWP ID	进程 PID；0 表示内核线程
线程	进程内的线程 ID	内核 TID；内核 DID 表示内核线程

例如，在内核实验中，如果希望仅过滤几个进程 ID，请在“过滤数据”对话框的“LWP 过滤器”字段中输入感兴趣的 PID。

第 6 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。er_print 实用程序列显性能分析器支持的各种显示的 ASCII 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。可以使用 er_print 实用程序显示函数、调用者和被调用者的性能度量；源代码列表和反汇编代码列表；抽样信息；数据空间数据；以及执行统计数据。

本章包含以下主题。

- 第 96 页中的 “er_print 语法”
- 第 96 页中的 “度量列表”
- 第 99 页中的 “控制函数列表的命令”
- 第 102 页中的 “控制调用者-被调用者列表的命令”
- 第 104 页中的 “控制泄漏和分配列表的命令”
- 第 104 页中的 “控制源代码和反汇编代码列表的命令”
- 第 108 页中的 “控制数据空间列表的命令”
- 第 109 页中的 “控制内存对象列表的命令”
- 第 110 页中的 “控制索引对象列表的命令”
- 第 111 页中的 “支持线程分析器的命令”
- 第 112 页中的 “列出实验、抽样、线程和 LWP 的命令”
- 第 113 页中的 “控制实验数据过滤的命令”
- 第 114 页中的 “控制装入对象展开和折叠的命令”
- 第 115 页中的 “列出度量的命令”
- 第 116 页中的 “控制输出的命令”
- 第 118 页中的 “列显其他信息的命令”
- 第 119 页中的 “设置缺省值的命令”
- 第 120 页中的 “仅为性能分析器设置缺省值的命令”
- 第 121 页中的 “杂项命令”
- 第 122 页中的 “表达式语法”
- 第 123 页中的 “er_print 命令示例”

有关收集器所收集的数据的说明，请参见第 2 章。

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 章和联机帮助。

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。

多个选项可以出现在 er_print 命令行上。它们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，会列显处理时出现的任何错误消息或警告消息。可以使用 procstats 命令列显有关处理的摘要统计数据。

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的列表。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 6–1 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 6–1 度量类型字符

字符	说明
e	显示独占度量值
i	显示包含度量值
a	显示归属度量值（仅适用于调用者-被调用者度量）
d	显示动态度量值（仅适用于数据派生的度量）

表 6–2 中列出了允许的度量可见性字符。可见性字符串中可见性字符的顺序无关紧要：它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 6–2 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。对于调用者-被调用者列表中的归属度量，将度量显示为所选函数的包含度量的百分比。
+	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` 均表示只要包含用户 CPU 时间以任一形式可见，分析器就应该按它排序；而 `sort i!user` 则表示不管包含用户 CPU 时间是否可见，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 6–3 中列出了计时度量、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，可以获取已装入实验的所有可用度量名称字符串的列表。通过使用不带参数的 `collect` 命令，可以获取计数器名称的列表。有关硬件计数器的更多信息，请参见第 26 页中的 "硬件计数器溢出分析数据"。

表 6–3 度量名称字符串

类别	字符串	说明
计时度量	`user`	用户 CPU 时间
	`wall`	挂钟时间
	`total`	总 LWP 时间
	`system`	系统 CPU 时间
	`wait`	CPU 等待时间
	`ulock`	用户锁定时间
	`text`	文本缺页时间
	`data`	数据缺页时间
	`owait`	其他等待时间
同步延迟度量	`sync`	同步等待时间
	`syncn`	同步等待计数
MPI 跟踪度量	`mpitime`	MPI 调用所用的时间
	`mpisend`	MPI 发送操作的数量
	`mpibytessent`	在 MPI 发送操作中发送的字节数
	`mpireceive`	MPI 接收操作的数量
	`mpibytesrecv`	在 MPI 接收操作中接收的字节数
	`mpiother`	对其他 MPI 函数的调用数
内存分配度量	`alloc`	分配的数量
	`balloc`	分配的字节

表 6–3 度量名称字符串 （续）

类别	字符串	说明
	leak	泄漏的数量
	bleak	泄漏的字节
硬件计数器溢出度量	cycles	CPU 周期
	insts	发出的指令
线程分析器度量	raccesses	数据争用访问
	deadlocks	死锁

除了表 6–3 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。要列出所有可用的计数器，请使用不带参数的 collect 命令。

要查看在已装入的实验中可用的度量，请使用 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。

可以使用 limit 命令限制写入的行数（请参见第 116 页中的 “控制输出的命令”）。

列显的缺省度量是独占和包含用户 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。

对于用 Java 编程语言编写的应用程序，显示的函数信息因视图模式的设置而异，视图模式可设置为用户、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；它还抑制非用户 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，且不抑制非用户 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：

- 用秒表示的包含用户 CPU 时间
- 包含用户 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，如第 119 页中的 “设置缺省值的命令”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。

除了设置函数列表的度量外，metrics 命令还将调用者-被调用者的度量以及数据派生输出的度量设为相同的设置。有关详细信息，请参见第 102 页中的 “cmetrics *metric_spec*”和第 108 页中的 “data_metrics *metric_spec*”。

处理 metrics 命令时，将列显一条消息，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，请参见第 96 页中的 “度量列表”。要查看可用度量的列表，请使用 metric_list 命令。

如果 metrics 命令中包含错误，则忽略它并显示一条警告，但先前的设置仍然有效。

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，如第 119 页中的 “设置缺省值的命令”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* 为第 96 页中的 “度量列表”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` 实用程序按包含用户 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，则列显一条警告并忽略该命令。完成命令时，将列显排序度量。

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（请参见第 116 页中的 “控制输出的命令”）。

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，还包括源文件、目标文件和装入对象的名称），以及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包含度量）。

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，则会列显具有对应 *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。

控制调用者-被调用者列表的命令

以下命令控制显示调用者和被调用者信息的方式。

callers-callees

按函数排序度量 (sort) 指定的顺序，列显每个函数的调用者-被调用者面板。

在每个调用者-被调用者报告中，调用者和被调用者按调用者-被调用者排序度量 (csort) 进行排序。可以使用 limit 命令限制写入的面板数（请参见第 116 页中的 “控制输出的命令”）。选定的（中央）函数以星号标记，如以下示例所示。

```
Attr.     Excl.     Incl.      Name
User CPU  User CPU  User CPU
 sec.      sec.      sec.
4.440     0.        42.910     commandline
0.        0.         4.440    *gpf
4.080     0.         4.080     gpf_b
0.360     0.         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。

cmetrics *metric_spec*

指定调用者-被调用者度量的选项。缺省情况下，只要更改了函数列表度量，就会将调用者-被调用者度量设置为与函数列表度量匹配。如果省略 *metric_spec*，则显示当前的调用者-被调用者度量设置。

字符串 *metric_spec* 为第 96 页中的 “度量列表”中所述的度量关键字之一，如以下示例所示。

```
% cmetrics i.%user:a.%user
```

此命令指示 er_print 显示以下度量。

- 用秒表示的包含用户 CPU 时间
- 包含用户 CPU 时间百分比
- 用秒表示的归属用户 CPU 时间
- 归属用户 CPU 时间百分比

完成 cmetrics 命令时，将列显一条消息，指明当前的度量选项。对于前面的示例，消息如下。

```
current: i.%user:a.%user:name
```

缺省情况下，只要更改了函数列表度量，就会将调用者-被调用者度量设置为与函数列表度量匹配。

调用者-被调用者归属度量被插入到对应的独占度量和包含度量前面，*visibility* 对应于这两个度量的 *visibility* 设置的逻辑“或”。将静态度量设置复制到调用者-被调用者度量。如果 *metric-name* 不在列表中，则将它附加到列表中。

可以使用 `cmetric_list` 命令获取已装入实验的所有可用 *metric-name* 值的列表。

如果 `cmetrics` 命令中包含错误，则忽略它并显示一条警告，但先前的设置仍然有效。

csingle *function_name* [*N*]

为指定的函数写入调用者-被调用者面板。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* 个函数写入调用者-被调用者面板。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，则会列显具有对应 *N* 值的函数列表。

csort *metric_spec*

按指定的度量对调用者-被调用者显示进行排序。字符串 *metric_spec* 为第 96 页中的 “度量列表”中所述的度量关键字之一，如以下示例所示。

```
% csort a.user
```

如果省略 *metric-spec*，则显示当前的调用者-被调用者排序度量。

`csort` 度量必须是归属度量或静态度量。如果指定了多个度量，则按匹配的第一个可见度量进行排序。

只要设置了度量（显式设置或缺省设置），就会基于函数度量设置调用者-被调用者排序度量，如下所示：

- 如果排序依据是动态度量（包含或独占），则按对应的归属度量进行排序。
- 如果排序依据是静态度量，则按它进行排序。

此命令指示 `er_print` 实用程序按归属用户 CPU 时间对调用者-被调用者显示进行排序。命令完成时，将列显排序度量。

控制泄漏和分配列表的命令

本节介绍与内存分配和解除分配相关的命令。

leaks

显示由通用调用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栈的总分配字节数。该列表按分配的字节数进行排序。

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。

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。

source { *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。在任一种情况下，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` 实用程序将列显可能的目标文件名称的列表；如果提供的名称是函数，则将函数名称附加到目标文件名称，还将列显表示该目标文件的 *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目标模块的源文件。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 { *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 6–4 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、预取）的消息。
f[e]	显示前端消息。
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 6–4 中所示的带注释的源代码列表的类列表相同。可以在类列表中添加以下选项。

表 6–5 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 6–4 中所示的带注释的源代码列表的类列表相同。

setpath *path_list*

设置用于查找源文件、目标文件等文件的路径。*path_list* 是目录的冒号分隔列表。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 引用当前实验集（按装入实验的顺序）；可以将其缩写为单个 $ 字符。

缺省设置为：$expts:.。如果在搜索当前路径设置时未找到文件，则使用编译时的全路径名。

不带参数的 setpath 列显当前路径。

addpath *path_list*

将 *path_list* 附加到当前 setpath 设置。

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、目标文件或共享对象的任何路径名中，旧的前缀将由 *new-prefix* 所指定的新前缀替代。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

数据空间命令仅适用于在其中指定了主动回溯的硬件计数器实验，以及使用 -xhwcprof 选项（该选项在 SPARC 平台上可用）编译的文件中的对象。有关详细信息，请参见《Sun Studio 12：Fortran 用户指南》、《Sun Studio 12：C 用户指南》或《Sun Studio 12：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。每个聚集数据对象会显示归属于该对象的总度量，后跟按偏移顺序显示的所有元素，每个元素具有自己的度量和相对于 32 字节块的大小和位置指示符。

data_metrics *metric_spec*

设置数据派生的度量。*metric_spec* 是在第 96 页中的 “度量列表”中定义的。

缺省情况下，只要更改了函数列表度量，就会将数据派生的度量设置为与函数列表度量匹配。将对应于任何具有数据派生类型的可见独占度量或包含度量的数据派生度量设置为 *visibility* 对应于这两个度量的 *visibility* 设置的逻辑“或”。

将静态度量设置复制到数据派生的度量。如果度量名称不在列表中，则将度量名称附加到列表中。

如果省略 *metric_spec*，则显示当前的数据派生度量设置。

可以使用 `data_metric_list` 命令获取已装入实验的所有可用 *metric-name* 值的列表。

如果 *metric_spec* 存在任何错误，则忽略它，而数据派生度量保持不变。

data_sort

设置数据对象的排序度量。动态度量需要前缀 `d`，而静态度量可以省略它。`data_sort` 度量必须是数据派生的度量或静态度量。

如果指定了多个度量，则按匹配的第一个可见度量进行排序。只要设置了度量（显式设置或缺省设置），就会基于函数度量设置数据派生排序度量：

- 如果排序依据是具有对应数据派生类型的动态度量（包含或独占），则按对应的数据派生度量进行排序。
- 如果排序依据是不具有数据派生类型的包含或独占度量，则按第一个可见的数据派生度量进行排序。
- 如果排序依据是静态度量，则按它进行排序。

控制内存对象列表的命令

内存对象命令仅适用于在其中指定了主动回溯的硬件计数器实验，以及使用 `-xhwcprof` 选项（该选项在 SPARC 平台上可用）编译的文件中的对象。有关详细信息，请参见《Sun Studio 12：Fortran 用户指南》、《Sun Studio 12：C 用户指南》或《Sun Studio 12：C++ 用户指南》。

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，其大小为 8KB、64KB、512KB 和 4 MB。可以使用 `mobj_define` 命令定义其他内存对象。

以下命令控制内存对象列表。

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* 提供的对象，定义新的内存对象类型。表达式的语法在第 122 页中的 “表达式语法”中介绍。

不应先定义 *mobj_type*。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

而对于基于 `PADDR` 的对象，表达式应该采用以下格式：

PADDR>0?*expression*:-1

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。为线程、Cpu、抽样和秒预定义了索引对象。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。

indxobj_define *indxobj_type index_exp*

通过将包映射到由 *index_exp* 提供的对象，定义新的索引对象类型。表达式的语法在第 122 页中的 “表达式语法”中介绍。

不应先定义 *indxobj_type*。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
```

`indxobj_metrics` *metric_spec*

指定索引对象的度量选项。*metric_spec* 只能包含独占度量和静态度量，因为索引对象是不分层的。

有关度量列表的语法的信息，请参见第 96 页中的 “度量列表”。要查看可用度量的列表，请使用 `metric_list` 命令。

`indxobj_sort` *metric_spec*

按指定的度量对索引对象列表进行排序。`indxobj_sort` 度量必须是独占度量或静态度量。如果指定了多个度量，则按匹配的第一个可见度量进行排序。

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《Sun Studio 12：线程分析器用户指南》。

`races`

写入实验中所有数据争用的列表。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`，则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 Experiment
== ==========
1 test.1.er
2 test.6.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：

- 指定过滤表达式，为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

指定过滤表达式

可以使用 `filters` 命令指定过滤表达式。

`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。表达式的语法在第 122 页中的 "表达式语法"中介绍。

选择要进行过滤的抽样、线程、LWP 和 CPU

选择列表

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* ...]

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。可以通过用 + 符号连接多个选择列表来进行多个选择。

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 `exp_list` 命令确定实验编号。

一些选择示例如下所示。

```
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

`object_list`

显示一个由两列组成的列表，包含可用装入对象的状态和名称。在第一列中显示每个装入对象的展开状态，在第二列中显示对象的名称。每个装入对象的名称前有 `yes`（指示在函数列表中显示该对象的函数（已展开））或 `no`（指示在函数列表中不显示该对象的函数（已折叠））。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
=== ==================
```

```
no  <Unknown>
yes <Freeway>
yes <libCstd_isa.so.1>
yes <libnsl.so.1>
yes <libmp.so.2>
yes <libc.so.1>
yes <libICE.so.6>
yes <libSM.so.6>
yes <libm.so.1>
yes <libCstd.so.1>
yes <libX11.so.4>
yes <libXext.so.0>
yes <libCrun.so.1>
yes <libXt.so.4>
yes <libXm.so.4>
yes <libsocket.so.1>
yes <libgen.so.1>
yes <libcollector.so>
yes <libc_psr.so.1>
yes <ld.so.1>
yes <liblayout.so.1>
```

`object_select` *object1,object2,...*

选择要显示其中函数信息的装入对象。*object-list* 是装入对象的列表，用逗号分隔且不加空格。如果选择某个装入对象，则会展开其函数，并且在函数列表中显示具有非零度量的所有函数。如果未选择某个装入对象，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基本名称。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

显示调用者-被调用者列表中当前选定的度量和可以在其他命令（例如 cmetrics 和 csort）中使用的度量关键字列表，以在调用者-被调用者列表中引用各种类型的度量。

注 – 可以将归属度量指定为仅与 cmetrics 命令一起显示，而不是与 metrics 命令或 data_metrics 命令一起显示，而且仅与 callers-callees 命令一起显示，而不是与 functions 命令或 data_objects 命令一起显示。

---

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。

注 – 可以将数据派生度量指定为仅与 data_metrics 命令一起显示，而不是与 metrics 命令或 cmetrics 命令一起显示，而且仅与 data_objects 命令一起显示，而不是与 functions 命令或 callers-callees 命令一起显示。

---

indx_metric_list

显示当前选定的索引对象度量。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*

将输出限制为报告中的前 *n* 个条目；*n* 是一个无符号正整数。

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（**用户**）　对于 Java 实验，显示 Java 线程的 Java 调用栈，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 调用栈时，将根据函数 <no Java callstack recorded> 报告时间。

对于 OpenMP 实验，在 OpenMP 运行时执行某些操作时，将显示已协调的主线程调用栈和从属线程调用栈，并添加名称格式为 <OMP-*> 的特殊函数。

expert（**专家**）　对于 Java 实验，在执行用户的 Java 代码时，将显示 Java 线程的 Java 调用栈；而在执行 JVM 代码或当 JVM 软件不报告 Java 调用栈时，则显示计算机调用栈。显示内务处理线程的计算机调用栈。

对于 OpenMP 实验，显示与 user 模式相同的信息。

machine（**计算机**）　对于 Java 实验和 OpenMP 实验，显示所有线程的计算机调用栈。

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。

列显其他信息的命令

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，将列显任何错误或警告。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。此命令仅在运行 Solaris OS 的 SPARC 处理器上适用。

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（请参见第 116 页中的 “控制输出的命令”）。

overview *exp_id*

写出当前为指定实验选择的每个抽样的抽样数据。*exp_id* 可以通过 exp_list 命令获得。如果 *exp_id* 为 all 或未提供，则显示所有实验的抽样数据。*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。

statistics *exp_id*

写出在指定实验的当前抽样集上聚集的执行统计数据。有关显示的执行统计数据的定义和含义，请参见 getrusage(3C) 和 proc(4) 手册页。执行统计数据包括来自收集器未收集其任何数据的系统线程的统计数据。

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。

设置缺省值的命令

可以使用以下命令设置 er_print 与性能分析器的缺省值。这些命令只能用于设置缺省值：它们不能用作 er_print 实用程序的输入。可以将它们包括在名为 .er.rc 的缺省文件中。仅适用于性能分析器缺省值的命令在第 120 页中的 "仅为性能分析器设置缺省值的命令"中介绍。

可将缺省文件包含在起始目录中以为所有实验设置缺省值，或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，将扫描当前目录和起始目录以查找缺省文件。如果存在缺省文件，则读取它们，同时也读取系统缺省文件。起始目录中的 .er.rc 文件的缺省值覆盖系统的缺省值，而当前目录中 .er.rc 文件的缺省值覆盖起始目录和系统的缺省值。

注 – 要确保从存储实验的目录读取缺省文件，必须从该目录启动性能分析器或 er_print 实用程序。

缺省文件还可以包括 scc、sthresh、dcc、dthresh、cc、setpath、addpath、pathmap、name、mobj_define、indxobj_define、tabs、rtabs 和 viewmode 命令。可以在缺省文件中包括多个 dmetrics、dsort、addpath、pathmap、mobj_define 和 indxobj_define 命令，并串联来自所有 .er.rc 文件的命令。对于所有其他命令，如果出现多次，则使用第一次出现的该命令，而忽略随后出现的该命令。

dmetrics *metric_spec*

指定要在函数列表中显示或列显的缺省度量。度量列表的语法和用法在第 96 页中的 "度量列表"一节中介绍。列表中度量关键字的顺序确定显示度量的顺序和它们在性能分析器的度量选择器中出现的顺序。

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者-被调用者列表的缺省度量。

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第 96 页中的 “度量列表”一节中介绍。

调用者-被调用者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*`}`

将读取后续实验的模式设置为 `on`（启用所有后续实验）或 `off`（禁用所有后续实验）。如果使用 =*regexp*，则启用其后代或可执行文件名称与正则表达式匹配的那些实验的数据。

仅为性能分析器设置缺省值的命令

`tabs` *tab_spec*

设置在分析器中可见的一组缺省标签。标签由生成对应报告的 `er_print` 命令命名（包括用于内存对象标签的 *mobj_type* 或用于索引对象标签的 *indxobj_type*）。*timeline* 指定 “时间线” 标签，而 *headers* 指定 “实验” 标签。

仅显示已装入实验中的数据所支持的那些标签。

`rtabs` *tab_spec*

设置使用 `tha` 命令调用分析器时可见的一组缺省标签，以便检查线程分析器实验。仅显示已装入实验中的数据所支持的那些标签。

`tlmode` *tl_mode*

设置性能分析器的 “时间线” 标签的显示模式选项。选项列表是一个冒号分隔列表。下表介绍了允许的选项。

表 6–6 时间线显示模式选项

选项	含义
lw[p]	显示 LWP 的事件
t[hread]	显示线程的事件
c[pu]	显示 CPU 的事件
r[oot]	在根上对齐调用栈
le[af]	在叶上对齐调用栈
d[epth] *nn*	设置可以显示的调用栈的最大深度

选项 `lwp`、`thread` 和 `cpu` 是互斥的，`root` 和 `leaf` 也是互斥的。如果在列表中包括一组互斥选项中的多个，则仅使用最后一个选项。

`tldata` *tl_data*

选择在性能分析器的"时间线"标签中显示的缺省数据类型。类型列表中的类型由冒号分隔。下表列出了允许的类型。

表 6–7 时间线显示数据类型

类型	含义
sa[mple]	显示抽样数据
c[lock]	显示时钟分析数据
hw[c]	显示硬件计数器分析数据
sy[nctrace]	显示线程同步跟踪数据
mp[itrace]	显示 MPI 跟踪数据
he[aptrace]	显示堆跟踪数据

杂项命令

`mapfile` *load-object* { *mapfilename* | - }

将指定装入对象的映射文件写入文件 *mapfilename*。如果指定一个短划线 (-) 而不是 *mapfilename*，则 `er_print` 将映射文件写入标准输出。

`procstats`

列显处理数据的累积统计数据。

`script` *file*

处理脚本文件 *file* 中的附加命令。

`version`

列显 `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。

`help`

列显 `er_print` 命令的列表。

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的组合。对于过滤器，如果表达式的计算结果为真，则包括包；如果表达式的计算结果为假，则排除包。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常量、数据记录中的字段（包括 `THRID`、`LWPID`、`CPUID`、`STACK`、`LEAF`、`VIRTPC`、`PHYSPC`、`VADDR`、`PADDR`、`DOBJ`、`TSTAMP`、`SAMPLE`、`EXPID`、`PID`）或内存对象的名称。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`）。如在 C 中那样指定 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。

过滤表达式的计算结果为布尔值。如果应该包括包，则计算结果为真，如果不应该包括包，则计算结果为假。线程、LWP、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（提供整数纳秒，从将要处理其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 命令中的时间以秒提供，必须转换为纳秒以用于时间过滤。时间还可以从分析器中的“时间线”显示获得。

函数过滤可以基于叶函数或栈中的任何函数。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*")，其中 *regexp* 是正则表达式，如 regexp(5) 手册页中所指定的那样。函数的整个名称（如 *name* 的当前设置所指定）必须匹配。

可以通过确定结构 FNAME("*regexp*") 中的任何函数是否位于由关键字 STACK:(FNAME("myfunc") SOME IN STACK) 表示的函数数组中来指定基于调用栈中的任何函数的过滤。

数据对象过滤类似于栈函数过滤，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包有意义；此类过滤将排除所有其他包。

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（请参见第 110 页中的“mobj_define *mobj_type index_exp*”）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（请参见第 110 页中的“indxobj_define *indxobj_type index_exp*”）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。

er_print 命令示例

- 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，后跟调用者-被调用者数据（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  -functions -cmetrics a.user \
-csort a.user -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 -sort e.user  \
-limit 100  -functions test.1.er

er_print -cmetrics  a.%user -csort a.user  \
-callers-callees test.1.er
```

- 此示例总结了在函数中是如何使用时间的。

```
er_print -functions test.*.er
```

- 此示例显示调用者-被调用者关系。

```
er_print -callers-callees test.*.er
```

- 此示例显示哪些源代码行是常用的。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.*.er
```

- 此示例仅显示编译器注释。无需运行程序即可使用此命令。

```
er_src -myfile.o
```

- 这些示例使用挂钟分析来列出函数和调用者-被调用者。

```
er_print -metrics  ei.%wall -functions test.*.er

er_print -cmetrics aei.%wall  -callers-callees test.*.er
```

- 此示例显示如何运行包含 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
```

- 此示例显示高级 MPI 函数。MPI 具有许多内部软件层，但是此示例显示一种只查看入口点的方法。可能存在一些重复符号，可以忽略它们。

```
er_print -functions test.*.er  | grep PMPI_
```

第 7 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。除了数据包头外，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：

- 第 127 页中的 "数据收集的工作原理"
- 第 130 页中的 "解释性能度量"
- 第 135 页中的 "调用栈和程序执行"
- 第 150 页中的 "将地址映射到程序结构"
- 第 157 页中的 "将性能数据映射到索引对象"
- 第 157 页中的 "将数据地址映射到程序数据对象"

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，记录有关将哪些装入对象装入目标的地址空间以及装入或卸载它们的时间的时间相关信息。
- 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，如下面第 130 页中的 “解释性能度量” 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 LWP 共享。数据文件的命名如下：

表 7–1 数据类型和对应的文件名

数据类型	文件名
基于时钟的分析	profile
硬件计数器溢出分析	hwcounters
同步跟踪	synctrace
堆跟踪	heaptrace
MPI 跟踪	mpitrace
开放式 MP 跟踪	omptrace

对于基于时钟的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、MPI 跟踪或开放式 MP 跟踪，从 LD_PRELOAD 环境变量在用户调用的常规例程上插入的 libcollector.so 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 LWP 之间的争用和序列化。

实验可以选择性地包含文件名为 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 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 实用程序或分析器在实验上首次调用时写入。

后续进程

后续进程将其实验写入创始进程的实验目录内的子目录。

对这些新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 代表其他后续进程。
- 在代码字母后添加一个表示派生或执行的索引的数字。不管是否成功启动了进程，都会应用此数字。
- 附加实验后缀 .er 以构成完整的实验名称。

例如，如果创始进程的实验名称为 test.1.er，则由其第三个派生创建的子进程的实验为 test.1.er/_f3.er。如果该子进程执行新映像，则对应的实验名称为 test.1.er/_f3_x1.er。后续实验由与父实验相同的文件组成，但是它们没有后续实验（所有后续实验都由创始实验中的子目录表示），而且它们没有归档子目录（所有归档都在创始实验中进行）。

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，还具有一个附加文件 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 实验具有一个 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

可以使用以下三种不同方法记录实验：

- 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

分析器 GUI 中的“性能工具收集”窗口运行 collect 实验；IDE 中的“收集器”对话框运行 dbx 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` 预装入到目标的地址空间。然后，该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` 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。

dbx 实验，在正在运行的进程上

在正在运行的进程上使用 `dbx` 启动实验时，它会创建实验目录，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` 写入。

由于进程启动时 `libcollector.so` 不在目标地址空间中，因此取决于插入用户可调用函数（同步跟踪、堆跟踪、MPI 跟踪）的任何数据收集可能都不起作用。通常，符号已经解析为基础函数，因此无法发生插入。此外，以下后续进程也取决于插入，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用 dbx 附加到正在运行的进程之前已显式预装入 `libcollector.so`，则可以收集跟踪数据。

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、LWP ID 和处理器 ID。其中的前三项可以用于在性能分析器中按时间、线程或 LWP 过滤度量。有关处理器 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“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。

基于时钟的分析

基于时钟的分析的事件特定数据由分析间隔计数的数组组成。在 Solaris OS 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 LWP 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，数组包含在自上次进入用户模式以来进入的每个微态的计数累积（内核为每个 Solaris LWP 维护十个微态）。在 Linux OS 上，不存在微态；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栈。如果在分析间隔结束时 Solaris LWP 未处于用户模式，则在 LWP 或线程再次进入用户模式之前调用栈无法更改。因此，调用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。

Solaris OS 上每个微态所服务于的度量如表 7–2 所示。

表 7–2 内核微态如何服务于度量

内核微态	说明	度量名称
LMS_USER	在用户模式下运行	用户 CPU 时间
LMS_SYSTEM	在系统调用或缺页时运行	系统 CPU 时间
LMS_TRAP	在出现任何其他陷阱时运行	系统 CPU 时间
LMS_TFAULT	在用户文本缺页时休眠	文本缺页时间
LMS_DFAULT	在用户数据缺页时休眠	数据缺页时间
LMS_KFAULT	在内核缺页时休眠	其他等待时间
LMS_USER_LOCK	等待用户模式锁定时休眠	用户锁定时间
LMS_SLEEP	由于任何其他原因而休眠	其他等待时间
LMS_STOPPED	已停止（/proc、作业控制或 lwp_stop）	其他等待时间
LMS_WAIT_CPU	等待 CPU	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 Solaris LWP 或 Linux 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态。如果创建了许多 LWP 或线程，则误差可能是分析间隔的许多倍。
- 销毁 Solaris LWP 或 Linux 线程时，在记录最后一个分析数据包后会消耗一些时间。如果销毁许多 LWP 或线程，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 LWP 或线程的重新调度。因此，记录的 LWP 状态可能不能表示消耗大部分分析间隔的微态。如果要运行的 Solaris LWP 或 Linux 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 Solaris LWP 或 Linux 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：记录微态时，在运行程序的 LWP 时由分析信号中断的处理器很可能处于陷阱 CPU 微态。
- 分析间隔过期时，内核记录微态值。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 Solaris OS 上，此情况很可能会导致对陷阱 CPU 或等待 CPU 微态的过多记帐。
- 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。

除了刚刚介绍的不准确外，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 Solaris LWP 或 Linux 线程时间总计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，并且为不同函数、源代码行等报告的相对时间也不会显著失真。

对于 Solaris OS 上使用未绑定线程的多线程应用程序，`gethrvtime()` 所返回的值的差异可能没有意义，因为 `gethrvtime()` 返回 LWP 的值，而线程可能随 LWP 的不同而不同。

性能分析器中报告的 LWP 时间可能与 `vmstat` 报告的时间有很大差异，因为 `vmstat` 报告 CPU 的汇总时间。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"统计"标签和 er_print 统计显示中的微态计时基于进程文件系统 /proc 使用报告，因此微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，"统计"标签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，请参见第 137 页中的 "陷阱"。

同步等待跟踪

同步等待跟踪仅在 Solaris 平台上可用。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。

在完成导致延迟的事件之前，在其上安排有等待线程的 LWP 无法执行任何其他工作。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所用的等待时间部分。

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，请参见第 171 页中的 "硬件计数器溢出的归属"。另请参见第 137 页中的 "陷阱"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。

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。这些计数被归属到收集器函数 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。

选择过大的值可能导致溢出过少，不利于统计。最后一次溢出后产生的计数被归属到收集器函数 `collector_final_counters`。如果在此函数中看到一大部分计数，则表明溢出值过大。

堆跟踪

收集器通过插入内存分配和解除分配函数 `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从其装入给定函数的库。

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。

数据空间分析

数据空间分析是一个数据集合，在其中根据导致事件的数据对象引用（而不仅仅是发生与内存相关事件的指令）报告与内存相关的事件（如高速缓存未命中）。数据空间分析在 Linux 系统上不可用。

要允许进行数据空间分析，目标必须是使用 `-xhwcprof` 标志和 `-xdebugformat=dwarf -g` 标志为 SPARC 体系结构编译的 C 程序。此外，收集的数据必须是与内存相关的计数器的硬件计数器分析数据，且必须在计数器名称之前放置可选的 + 号。性能分析器包括两个与数据空间分析相关的标签（即 DataObject 标签和 DataLayout 标签），以及用于内存对象的各种标签。

也可以通过在分析间隔之前放置加号 (+)，使用时钟分析进行数据空间分析。

MPI 跟踪

MPI 跟踪仅在 Solaris 平台上可用。MPI 跟踪记录有关对 MPI 库函数的调用的信息。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）、发送和接收的操作数以及发送或接收的字节数组成。跟踪是通过插入对 MPI 库的调用进行的。插入函数既不具有有关数据传输优化的详细信息，也不具有有关传输误差的详细信息，因此提供的信息表示数据传输的简单模型，将在下面段落中对此进行说明。

接收的字节数是指对 MPI 函数的调用中定义的缓冲区长度。接收的实际字节数对于插入函数是不可用的。

有些全局通信函数具有单一起始点或单一接收进程（称为根）。对这样的函数进行记帐的过程如下：

- 根将数据发送到所有进程，包括它本身。
- 根从所有进程接收数据，包括它本身。
- 各个进程相互通信，包括它本身

以下示例说明了记帐过程。在这些示例中，G 是组的大小。

对于对 `MPI_Bcast()` 的调用，

- 根发送 N 个字节的 G 包，每个进程一个包，包括它本身
- 组中的所有 G 进程（包括根）接收 N 个字节

对于对 `MPI_Allreduce()` 的调用，

- 每个进程发送 N 个字节的 G 包
- 每个进程接收 N 个字节的 G 包

对于对 `MPI_Reduce_scatter()` 的调用，

- 每个进程发送 N/G 个字节的 G 包
- 每个进程接收 N/G 个字节的 G 包

调用栈和程序执行

调用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 是对包含叶 PC 的函数的调用的地址；下一个 PC 是对该函数的调用的地址，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。记录调用栈的过程涉及从程序栈获取返回地址，这称为**展开堆栈**。有关展开失败的信息，请参见第 149 页中的 “不完全的堆栈展开”。

调用栈中的叶 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）用于将包含度量分配到它所在的函数。

大多数时候，已记录调用栈中的 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有关程序如何执行的简单直观模型对应，而且性能分析器的已报告度量可能会引起混淆。有关此类情况的更多信息，请参见第 150 页中的 “将地址映射到程序结构”。

单线程执行和函数调用

程序执行的最简单案例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。（通常，已提交分支后的下一条指令以供执行：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更复杂。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。如果在首次通过 PLT 条目调用过程中发生分析事件，但是叶 PC 不是 PLT 指令之一，则 PLT 和 `ld.so` 中的代码引起的任何 PC 都将由对人工函数 `@plt` 的调用替换，该函数将累计包含时间。每个共享对象都有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 看起来好像是对系统函数 `sigacthandler()` 的调用的返回地址。`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包含度量，都显示为中断函数的包含度量。

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，而且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。硬件陷阱的一些示例是浮点单元无法完成某个指令（例如，用于 UltraSPARC® III 平台上的某些寄存器内容值的 `fitos` 指令）时发出的陷阱，或者指令没有在硬件中实现时发出的陷阱。

发出陷阱时，Solaris LWP 或 Linux 内核进入系统模式。在 Solaris OS 上，微态通常从用户 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，其中不切换微态。

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。被调用者可重用来自调用者的帧，而不是生成新的帧，而且可从调用者复制被调用者的返回地址。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
```

也就是说，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

在 Solaris OS 中，简单程序在单个 LWP（lightweight process，轻量级进程）上的单个线程中执行。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。

Solaris OS 支持两种线程实现：Solaris 线程和 POSIX 线程 (Pthread)。从 Solaris 10 OS 开始，这两种线程实现都包括在 libc.so 中。在 Solaris 9 OS 中，线程实现包含在单独的库（即 libthread.so 和 libpthread.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。对于涉及目标由此线程执行的任何调用栈，堆栈的顶部是 _thread_start()，与线程创建函数的调用者没有任何联系。因此，与所创建的线程关联的包含度量仅传播至 _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。

Pthread 在 Solaris 10 OS 中以及 Linux OS 中可用于显式多线程。

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 Solaris OS 上，新的 pthread 开始执行时，将会调用 _lwp_start() 函数。在 Solaris 10 OS 上，_lwp_start() 调用中间函数 _thr_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 Solaris 9 OS 上，_lwp_start() 直接调用应用程序的启动例程。

在 Linux OS 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，JVM 软件解释大多数方法；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，以生成这些方法的机器码。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 5.0 中的 JVMTI 上，可通过各种方法实现数据收集。

Java 调用栈和机器调用栈

性能工具通过记录每个 Solaris LWP 或 Linux 线程生存期中的事件，以及发生事件时的调用栈来收集其数据。在执行任何应用程序的任意点上，调用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都存在两个有意义的调用栈：Java 调用栈和机器调用栈。这两个调用栈都在配置期间进行记录，并在分析期间进行协调。

基于时钟的分析和硬件计数器溢出分析

用于 Java 程序的基于时钟的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 调用栈和机器调用栈。

同步跟踪

Java 程序的同步跟踪基于线程尝试获取 Java 监视器时生成的事件。将会为这些事件同时收集机器调用栈和 Java 调用栈，但不为在 JVM 软件中使用的内部锁收集同步跟踪数据。

堆跟踪

堆跟踪数据记录由用户代码生成的对象分配事件以及由垃圾收集器生成的对象解除分配事件。此外，对 C/C++ 内存管理函数（如 `malloc` 和 `free`）的任何使用也将生成记录的事件。

Java 处理表示法

对于用 Java 编程语言编写的应用程序，有以下三种显示性能数据的表示法：Java 表示法、专家 Java 表示法和机器表示法。缺省情况下，将显示 Java 表示法（前提是数据支持它）。下一节汇总了这三种表示法的主要差异。

用户表示法

用户表示法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，可能存在已执行的特定 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 Java 表示法中，所有方法会被聚集显示为一个方法。缺省情况下，在分析器中选定此表示法。

Java 表示法中 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 与机器 PC 相对应。Java 线程的调用栈可能同时具有 Java PC 和机器 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%。

Java 表示法中的函数列表针对所调用的 Java 方法和任何本机方法显示度量。调用者-被调用者面板显示 Java 表示法中的调用关系。

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它生成的字节码，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。每个线程的调用栈与其 Java 方法一起显示。

所有 Java 程序都可能具有显式同步，通常是通过调用 `monitor-enter` 例程执行的。

Java 表示法中的同步延迟跟踪基于 JVMTI 同步事件。Java 表示法中不显示来自常规同步跟踪的数据。

当前不支持 Java 表示法中的数据空间分析。

专家用户表示法

专家 Java 表示法与 Java 表示法类似，不同之处是在专家 Java 表示法中公开了在 Java 表示法中抑制的一些 JVM 内部详细信息。对于专家 Java 表示法，时间线显示所有线程；内务处理线程的调用栈是本机调用栈。

机器表示法

机器表示法显示来自 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。机器表示法看起来与用传统语言编写的应用程序的表示法相同。调用栈显示 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机器汇编程序代码，而不是 Java 字节码。调用者-被调用者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。

机器表示法中的时间线以条形图显示所有线程、LWP 或 CPU，而其中每项的调用栈都是机器表示法调用栈。

在机器表示法中，线程同步被移交给对 `_lwp_mutex_lock` 的调用。不显示同步数据，因为未跟踪这些调用。

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 2.5》 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，Sun Microsystems 的实际实现是这样的：通过直接记录的分析信息，用户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（或通过自动并行化）并行化程序时，行为是不同的。该行为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。

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。）函数的名称对 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。这些函数的名称在分析器中按以下形式显示，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` 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。

分析器基于调用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、调用者和被调用者、源代码行和指令，基于所收集数据的类型显示性能度量。它以两种模式（用户模式和机器模式）之一提供有关 OpenMP 程序性能的信息。（支持第三种模式，即专家模式，但该模式与用户模式完全相同。）

有关更多详细信息，请参见 OpenMP 用户社区 Web 站点上的白皮书《An OpenMP Runtime API for Profiling》。

OpenMP 分析数据的用户模式显示

分析数据的用户模式显示尝试提供信息，好像程序按照第 141 页中的 “OpenMP 软件执行概述”中所述的模型实际执行一样。实际的数据捕获运行时库 `libmtsk.so`（它不对应于模型）的实现详细信息。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，在以下三个方面不同于记录的数据和机器模式显示：

- 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。
- 处理调用栈以报告对应于代码运行方式模型的数据，如上所述。
- 为基于时钟的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。

人工函数

构造人工函数，并将其放置在用户模式调用栈上，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；每个人工函数后跟其功能说明：

- `<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

当线程处于对应于其中一个函数的 OpenMP 运行时状态时，会将对应函数作为堆栈上的叶函数添加。当线程的叶函数处于 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。

用户模式调用栈

了解此模型的最简单方法是查看 OpenMP 程序在其执行过程中各个时刻的调用栈。本节介绍一个简单的程序，该程序具有调用一个子例程 foo 的主程序。该子例程具有单个并行循环，线程在其中完成工作、争用、获取和释放锁，以及进入和离开临界段。显示另一组调用栈，反映一个从属线程调用了另一函数 bar（它进入嵌套并行区域）时的状态。

在此显示中，并行区域中所用的所有包含时间都包括在从中提取它的函数的包含时间中，其中包括在 OpenMP 运行时中所用的时间，而且该包含时间一直向上传播到 main 和 _start。

表示此模型中行为的调用栈如后续小节中所示。并行区域函数的实际名称具有以下形式，如上所述：

```
foo -- OMP parallel region from line 9[ [_$p1C9.foo]
bar -- OMP parallel region from line 5[ [_$p1C5.bar]
```

为了清晰起见，在说明中使用以下简化形式：

```
foo -- OMP...
bar -- OMP...
```

在说明中，所有线程的调用栈都在执行程序期间的某个时刻显示。每个线程的调用栈均显示为帧堆栈，将在分析器时间线标签中为单个线程选择单独分析事件得到的数据与顶部的叶 PC 匹配。在时间线标签中，每个帧都显示有 PC 偏移（在下面省略了此偏移）。所有线程的堆栈都在水平数组中显示，而在分析器时间线标签中，其他线程的堆栈将出现在垂直堆叠的分析数据栏中。此外，在提供的表示法中，将显示所有线程的堆栈，就像它们是在同一时刻捕获的那样，而在实际实验中，堆栈在每个线程中独立地捕获，并且彼此之间可能存在相对偏离。

所示的调用栈表示的数据如在分析器中或在 er_print 实用程序中通过用户视图模式显示的一样。

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。

主线程
foo
main
_start

2. 进入第一个并行区域时

   此时，库已创建从属线程，而且所有线程（主线程和从属线程）即将开始处理其工作块。所有线程都显示为从构造的 OpenMP 指令所在行上的 `foo`，或者从包含已自动并行化的循环语句的行，调用到并行区域 `foo-OMP...` 的代码中。每个线程中并行区域的代码从并行区域中的第一个指令调用到 OpenMP 支持库（显示为 `<OMP-overhead>` 函数）中。

主线程	从属线程 1	从属线程 2	从属线程 3
<OMP-overhead>	<OMP-overhead>	<OMP-overhead>	<OMP-overhead>
foo-OMP...	foo-OMP...	foo-OMP...	foo-OMP...
foo	foo	foo	foo
main	main	main	main
_start	_start	_start	_start

   其中可能出现 `<OMP-overhead>` 的窗口相当小，所以该函数可能不出现在任何特定实验中。

3. 在并行区域中执行时

   所有四个线程都在并行区域中执行有用的工作。

主线程	从属线程 1	从属线程 2	从属线程 3
foo-OMP...	foo-OMP...	foo-OMP...	foo-OMP...
foo	foo	foo	foo
main	main	main	main
_start	_start	_start	_start

4. 在并行区域中工作块之间执行时

   所有四个线程都在执行有用的工作，但是一个线程已完成一个工作块，并正在获取其下一个工作块。

主线程	从属线程 1	从属线程 2	从属线程 3
	<OMP-overhead>		
foo-OMP...	foo-OMP...	foo-OMP...	foo-OMP...
foo	foo	foo	foo
main	main	main	main
_start	_start	_start	_start

5. 在并行区域内的临界段中执行时

   所有四个线程都在执行，每个线程都在并行区域内。其中一个线程在临界段中，而其他线程之一在到达临界段之前（或完成它之后）正在运行。剩余的两个线程正在等待，以便它们自己进入临界段。

主线程	从属线程 1	从属线程 2	从属线程 3
<OMP-critical_section_wait>			<OMP-critical_section_wait>
foo-OMP...	foo-OMP...	foo-OMP...	foo-OMP...
foo	foo	foo	foo
main	main	main	main
_start	_start	_start	_start

   收集的数据不区分正在临界段中执行的线程的调用栈与尚未到达或已通过临界段的线程的调用栈。

6. 在并行区域内的锁定周围执行时

   锁定周围的代码段完全类似于临界段。所有四个线程都在并行区域内执行。一个线程在持有锁定时执行，一个在获取锁定之前（或获取并释放它之后）执行，另外两个线程正在等待锁定。

主线程	从属线程 1	从属线程 2	从属线程 3
<OMP-lock_wait>			<OMP-lock_wait>
foo-OMP...	foo-OMP...	foo-OMP...	foo-OMP...
foo	foo	foo	foo
main	main	main	main
_start	_start	_start	_start

与临界段示例中一样，收集的数据不区分持有锁定并执行的线程的调用栈与在获取锁定之前或释放锁定之后执行的线程的调用栈。

7. 在并行区域结尾附近

   此时，其中三个线程已完成其所有工作块，但是还有一个线程仍在工作中。在这种情况下，OpenMP 构造隐式指定了屏障；如果用户代码已显式指定屏障，则 <OMP-implicit_barrier> 函数将由 <OMP-explicit_barrier> 替换。

主线程	从属线程 1	从属线程 2	从属线程 3
<OMP-implicit_barrier>	<OMP-implicit_barrier>		<OMP-implicit_barrier>
foo-OMP...	foo-OMP...	foo-OMP...	foo-OMP...
foo	foo	foo	foo
main	main	main	main
_start	_start	_start	_start

8. 在并行区域结尾附近，具有一个或多个归约变量

   此时，其中两个线程已完成其所有工作块，而且正在执行归约计算，但是其中一个线程仍在工作，第四个线程已完成其归约部分，正在屏障处等待。

主线程	从属线程 1	从属线程 2	从属线程 3
<OMP-reduction>	<OMP-implicit_barrier>		<OMP-implicit_barrier>
foo-OMP...	foo-OMP...	foo-OMP...	foo-OMP...
foo	foo	foo	foo
main	main	main	main
_start	_start	_start	_start

   尽管在 <OMP-reduction> 函数中显示了一个线程，但是执行归约所用的实际时间通常相当少，且在调用栈样本中很少捕获到。

9. 在并行区域的结尾

   此时，所有线程均已完成并行区域内的所有工作块，且已到达屏障。

主线程	从属线程 1	从属线程 2	从属线程 3
<OMP-implicit_barrier>	<OMP-implicit_barrier>	<OMP-implicit_barrier>	<OMP-implicit_barrier>

主线程	从属线程 1	从属线程 2	从属线程 3
foo-OMP...	foo-OMP...	foo-OMP...	foo-OMP...
foo	foo	foo	foo
main	main	main	main
_start	_start	_start	_start

由于所有线程均已到达屏障，因此它们都可能会继续，实验不大可能找到处于此状态的所有线程。

10. 离开并行区域之后

 此时，所有从属线程都在等待进入下一个并行区域，它们处于自旋或休眠状态，具体状态取决于用户设置的各种环境变量。程序以串行方式执行。

主线程	从属线程 1	从属线程 2	从属线程 3
foo			
main			
_start	<OMP-idle>	<OMP-idle>	<OMP-idle>

11. 在嵌套并行区域中执行时

 所有四个线程都在工作，每个线程都在外部并行区域内。其中一个从属线程已调用另一函数 `bar`，并已创建嵌套并行区域，而且会创建一个附加从属线程以便与它一起工作。

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...			bar-OMP...
	bar			bar
foo-OMP...	foo-OMP...	foo-OMP...	foo-OMP...	foo-OMP...
foo	foo	foo	foo	foo
main	main	main	main	main
_start	_start	_start	_start	_start

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。它们是“OMP 工作”和“OMP 等待”。

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“OMP 工作”中累计时间。只要线程正在等待某项，之后才能继续，就会在“OMP 等待”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“总LWP 时间”度量相匹配。

OpenMP 分析数据的机器表示

在执行的各个阶段中，程序的实际调用栈与上面在直观模型中描述的有很大差异。机器表示模式将调用栈显示为已度量，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。

在下面的每个调用栈中，libmtsk 表示 OpenMP 运行时库内调用栈中的一个或多个帧。出现哪些函数以及出现顺序的详细信息随发行版的不同而不同，屏障代码的内部实现或执行归约也是如此。

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。调用栈与用户模式下的完全相同。

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

   在机器模式下，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。）

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，在机器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，则程序可能进行核心转储，或者数据收集代码可能进行核心转储，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者的帧之后，但在函数返回之前，叶 PC 位于函数中。
- 如果调用栈包含的帧超过 250 个，则收集器没有用于完全展开调用栈的空间。在这种情况下，调用栈中从 `_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。

中间文件

如果使用 `-E` 或 `-P` 编译器选项生成中间文件，则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目标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；使用 `-E` 选项进行编译，然后再编译生成的 `.i` 文件是出现此情况的一种方式。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目标文件。
- 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。在调用栈中记录的 PC 通常对应于程序文本段之一中的地址。

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。调用栈中的 PC 基于记录调用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。有些函数却不是这样；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是唯一的。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“摘要”标签中，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。此情况的一个非常常见的示例是，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，分析器的函数列表中仅使用一个名称。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“摘要”标签中，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目标模块）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分析器中。在“摘要”标签中，提供了其中每个函数的目标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、反汇编以及调用者和被调用者。
- 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目标模块，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，以便在库中内部使用的名称不会与您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，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移量。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者进行调用，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者。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，因此可能不将实际调用者或被调用者与相邻函数区分开。

Fortran 备用入口点

Fortran 提供了一种具有单个代码段的多个入口点的方法，允许调用者调用到函数的中间。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者。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重整名称 (mangled name)。分析器取消重整 (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，并且它们都影响分析器。

- 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，因此最好只是在调用点上插入函数的代码，而不是设置调用。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者。否则在内联函数的调用点上显示为包含度量的度量（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。

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。第 141 页中的 “OpenMP 软件执行概述”中介绍了这些函数。

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，分析器还基于从其提取这些函数的函数为其分配名称。其独占度量和包含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包含度量。第 141 页中的 “OpenMP 软件执行概述”中介绍了实现这一点的方法。

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重整 (demangle) 编译器生成主体函数的名称。

外联函数

可以在反馈优化编译期间创建外联函数。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。这些重整名称可能随发行版的不同而不同。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，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，具有相应的包含度量和独占度量。此外，外联函数的度量作为代码从中进行外联的函数的包含度量添加。

有关反馈优化编译的进一步详细信息，请参阅《C 用户指南》的附录 B、《C++ 用户指南》的附录 A 或《Fortran 用户指南》的第 3 章中对 `-xprofile` 编译器选项的说明。

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。除非用户使用收集器 API 函数提供所需的信息，否则收集器没有有关用 C 或 C++ 编写的动态编译函数的信息。有关 API 函数的信息，请参见第 44 页中的 “动态函数和模块”。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。在机器表示法中，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 不映射到已知函数。在这样的情况下，PC 映射到名为 <Unknown> 的特殊函数。

以下情况显示映射到 <Unknown> 的 PC：

- 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，请参见第 44 页中的 “动态函数和模块”。
- 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机器指令。
- 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> 函数的调用者和被调用者表示调用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栈（这些用户模式调用栈反映线程在 OpenMP 运行时库内处于某个状态的事件）。定义了以下人工函数；每个人工函数后跟其功能的说明：

- <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>。

<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，收集器会展开本机栈并调用到 JVM 软件中，以获取对应的 Java 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> 一起显示在分析器中。为了避免死锁，或展开 Java 栈时导致过度同步，JVM 软件可能会拒绝报告调用栈。

<Truncated-stack> 函数

分析器为记录调用栈中各个函数的度量所用的缓冲区大小是有限的。如果调用栈大小变得如此大而导致缓冲区变满，则对调用栈大小的任何进一步增加都将强制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。

<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“调用者-被调用者”列表中，它显示为执行任何程序的主线程中 _start() 的名义调用者，还显示为已创建线程的 _thread_start() 的名义调用者。如果堆栈展开是不完整的，则 <Total> 函数可能显示为 <Truncated-stack> 的调用者。

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_lost_hwc_overflow：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_lost_sigemt：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_final_counters：暂停或终止收集前一刻采用的计数器值，具有自上次溢出以来的计数。如果这对应于 <Total> 计数的重要部分，则建议使用较小的溢出间隔（即，较高的精度配置）。
- collector_record_counters：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。如果这对应于 <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、样本和秒。其他索引对象可能是通过直接发出的或 .er.rc 文件中的 er_print indxobj_define 命令定义的。在 IDE 中，可以通过从“视图”菜单中选择“设置数据表示”，选择“标签”标签，然后单击“添加定制索引对象”按钮来定义索引对象。

对于每个包，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，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范围，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，该区域可能是全局性的（如装入对象），也可能是特定的编译单元（目标文件）或函数。相同的数据对象可能是使用不同的范围定义的，或者是在不同的范围内以不同的方式引用的特定数据对象。

通过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，并传播到包含数据对象和人工 <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}。聚集元素需要其容器的其他规范，例如，{structure:foo_t}.{int i} 表示 foo_t 类型的上一结构的 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规范以协助标识数据对象。

<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）的性能度量外，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，其数据可以用于为其他数据对象的数据提供透视。

<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，如下面所述。

- 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

  回溯已确定很可能引发事件的内存引用指令，但是编译器未指定其关联的数据对象。
- 无类型信息

  回溯已确定很可能引发事件的指令，但是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

  回溯已确定很可能引发事件的内存引用指令，但是编译器未识别它，因此不可能确定关联的数据对象。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写，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。

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和/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。您也可以使用分析器中的“添加内存对象”对话框（通过单击“设置数据表示”对话框中的“添加定制内存对象”按钮，可以打开该对话框）定义定制内存对象。

◆ ◆ ◆ 第 8 章

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。

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，方法是选择分析器窗口左窗格中的“源”标签。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，请参见第 177 页中的 “在不运行实验的情况下查看源代码/反汇编代码”。

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“源”标签布局

“源”标签分为若干列，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。

“源”标签中的索引行

源文件是指可编译生成目标文件或解释为字节代码的任何文件。目标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。分析器分析目标文件，标识每个作为函数的可执行区域，并尝试将其在目标代码中找到的函数映射至与目标代码相关联的源文件中的函数、例程、子例程或方法。当分析器操作成功后，它将在带注释的源文件中对应于在目标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“函数”标签中的列表中显示）显示一个索引行。在“源”标签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。

```
0.       0.         600. int
0.       0.         601. icputime(int k)
0.       0.         602. {
0.       0.                   <Function: icputime>
```

从上面的示例能够看出，索引行紧跟在第一条指令行的后面。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：

```
0.       0.       1. ! Copyright 02/04/2000 Sun Microsystems, Inc. All Rights Reserved
0.       0.       2. !
0.       0.       3. ! Synthetic f90 program, used for testing openmp directives and
0.       0.       4. ! the analyzer
0.       0.       5.
0.       0.       6. !$PRAGMA C (gethrtime, gethrvtime)
0.       0.       7.
0.       81.497    [  8] 9000    format('X ', f7.3, 7x, f7.3, 4x, a)
0.       0.                 <Function: main>
```

有时，分析器可能无法将它在目标代码中找到的函数映射至与该目标代码相关联的源文件中的任何编程指令；例如，可能从另一个文件（如某个头文件）执行代码 `#included` 操作或内联操作。

如果目标代码的源代码不是目标代码中包含的函数的缺省源上下文，那么对应于目标代码的带注释的源代码将包含一个特殊索引行，该索引行交叉引用该函数的缺省源上下文。例如，编译 `synprog` 演示程序将创建对应于源文件 `endcases.c` 的目标模块 `endcases.o`。`endcases.c` 中的源代码声明函数 `inc_func`，该函数在头文件 `inc_func.h` 中定义。头文件通常以这种方式包含内联函数定义。由于 `endcases.c` 中声明函数 `inc_func`，但 `endcases.c` 中没有源代码行与 `inc_func` 的指令对应，因此 `endcases.c` 的带注释的源文件的顶部会显示一个特殊的索引行，如下所示：

```
0.650     0.650       <Function: inc_func, instructions from source file inc_func.h>
```

该索引行上的度量指示来自 endcases.o 目标模块的部分代码（在源文件 endcases.c 中）不具有行映射。

此外，分析器还会在带注释的源代码中为定义了 inc_func 函数的 inc_func.h 添加一个标准索引行。

```
.

0.      0.         2.
0.      0.         3. void
0.      0.         4. inc_func(int n)
0.      0.         5. {
0.      0.            <Function: inc_func>
```

类似地，如果函数具有**替代**源上下文，则会在**缺省**源上下文的带注释的源代码中显示交叉引用该上下文的索引行。

```
0.           0.       142. inc_body(int n)
0.650     0.650      <Function: inc_body, instructions from source file inc_body.h>
0.           0.       143. {
0.           0.                 <Function: inc_body>
0.           0.       144. #include "inc_body.h"
0.           0.       145. }
```

双击引用另一个源上下文的索引行，将在源代码窗口中显示该源文件的内容。

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，可能会为目标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，请参阅第 171 页中的 " "源" 、 "反汇编" 和 "PC" 标签中的特殊行" 。

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。以下显示了一个编译器注释的示例。

```
                    Function freegraph inlined from source file ptraliasstr.c into
                    the code for the following line
0.      0.          47.       freegraph();
                    48.    }
0.      0.          49.    for (j=0;j<ITER;j++) {
```

```
                    Function initgraph inlined from source file ptraliasstr.c into
                    the code for the following line
0.       0.         50.       initgraph(rows);

                    Function setvalsmod inlined from source file ptraliasstr.c into
                    the code for the following line
                    Loop below fissioned into 2 loops
                    Loop below fused with loop on line 51
                    Loop below had iterations peeled off for better unrolling and/or
                    parallelization
                    Loop below scheduled with steady-state cycle count = 3
                    Loop below unrolled 8 times
                    Loop below has 0 loads, 3 stores, 3 prefetches, 0 FPadds,
                    0 FPmuls, and FPdivs per iteration
                    51.       setvalsmod();
```

请注意，第 51 行的注释包括循环注释，因为函数 `setvalsmod()` 包含循环代码，并且函数已内联。

可以使用“设置数据表示”对话框中的“源码/反汇编”标签来设置在“源”标签中显示的编译器注释类型；有关详细信息，请参见第 84 页中的 “设置数据表示选项” 。

通用子表达式删除

一种最常见的优化是可以识别在多个位置出现的同一个表达式，并可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。这可以提高不带条件语句的循环中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，您可以看到与以下注释类似的编译器注释：

```
    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。以下为内联编译器注释的一个示例。

```
                Function initgraph inlined from source file ptralias.c
                    into the code for the following line
0.       0.         44.       initgraph(rows);
```

注 – 在分析器的“源”标签中，编译器注释不会换行，即不会显示在两行上。

并行化

如果您的代码包含 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。以下是一个有关并行化操作的计算机注释示例。

```
0.       6.324       9. c$omp  parallel do shared(a,b,c,n) private(i,j,k)
0.       0.
                   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，请参阅第 141 页中的“OpenMP 软件执行概述”。

带注释的源代码中的特殊行

在“源”标签中，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，请参阅第 171 页中的““源”、“反汇编”和 "PC" 标签中的特殊行”。

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，请参见第 119 页中的“设置缺省值的命令”。您还可以在分析器中使用“设置数据表示”对话框更改显示的度量和突出显示的阈值；有关详细信息，请参见第 84 页中的“设置数据表示选项”。

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，分析器首先确定在特定目标模块（`.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，分析器必须能够找到并读取目标模块或装入对象来确定从 PC 至源代码行的映射，此外，它还必须能够读取源文件来生成要显示的带注释的副本。分析器依次在以下缺省位置搜索源文件、目标文件和可执行文件，并在找到具有正确基本名称的文件时停止：

- 实验的归档目录
- 当前工作目录
- 可执行文件或编译对象中记录的绝对路径名

缺省位置可以使用 `addpath` 或 `setpath` 指令更改，也可以使用分析器 GUI 更改。

如果使用由 `addpath` 或 `setpath` 设置的路径列表找不到文件，则可以使用 `pathmap` 命令指定一个或多个路径重映射。使用 `pathmap` 命令，可以指定 *old-prefix* 和 *new-prefix*。在以 *old-prefix* 所指定的前缀开头的源文件、目标文件或共享对象的任何路径名中，旧的前缀将由 *new-prefix* 所指定的新前缀替代。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。

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指令时累积的度量。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。

当执行过程中存在严重的延迟（如高速缓存未命中或资源队列停止）或指令等待前一个指令返回结果时，如何解释度量的问题是最重要的问题。在这些情况下，源代码行的度量看上去很高（高得不太合理），应该查看代码中的其他行以找出导致高度量值的行。

度量格式

表 8–1 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 8–1 带注释的源代码度量

度量	含义
（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： ■ 所出现的代码段的所有指令已在优化期间删除。 ■ 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。 ■ 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栈中。`0.` 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目标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，以及请求带注释的反汇编代码的函数的目标模块是否已知：

- 如果目标模块未知，则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。
- 如果目标模块已知，则针对目标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。
- 如果编译器已向目标代码中插入注释，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。可能的度量值与前面介绍的源代码注释的度量值一样（请参见表 8–1）。

在多个位置执行 `#included` 的代码的反汇编代码列表将在每次代码执行 `#included` 时重复一次反汇编指令。只有重复的反汇编代码块首次显示在某个文件中时，源代码才会交错显示。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“源”标签，可以看到与每次重复的反汇编代码对应的索引行。

索引行可以显示在“反汇编”标签中。与“源”标签不同的是，不能直接使用这些索引行进行导航。不过，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“源”标签，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。不过，将光标放在这些指令中的其中一条指令上，然后选择“源”标签，将打开包含 `#included` 代码的文件。在显示此文件时选择“反汇编”标签，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件简单的工作。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，而且在记录调用栈时可能存在延迟。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，可能不会单独发送某些指令，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。而且，在记录调用栈时，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，并且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. }
```

在以下示例中，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，并且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，以下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）。
- 在 UltraSPARC® III 处理器上，装入指令上的高速缓存未命中会阻塞所有后续指令，直到未命中问题得到解决，不管这些指令是否使用正在装入的数据项。在 UltraSPARC II 处理器上，仅阻塞使用正在装入的数据项的指令。

硬件计数器溢出的归属

除 TLB 未命中外，硬件计数器溢出事件的调用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，原因之一是处理由溢出产生的中断所用的时间。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，那么要分辨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。对于计数内存访问事件的硬件计数器，如果该计数器名称带有加号 + 前缀，则收集器会搜索引起事件的 PC。

“ 源 ”、“ 反汇编 ” 和 "PC" 标签中的特殊行

外联函数

外联函数可以在反馈优化编译期间创建。在“源”标签和“反汇编”标签中，外联函数显示为特殊的索引行。在“源”标签中，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“反汇编”标签中，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。不同的发行版可以具有不同的重整名称 (mangled name)。分析器提供了函数名称的可读版本。有关详细信息，请参阅第 154 页中的“外联函数”。

如果在收集应用程序的性能数据时调用了外联函数，则分析器会在带注释的反汇编代码中显示一个特殊行，以显示该函数的包含度量。有关详细信息，请参见第 176 页中的“包含度量”。

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。第 141 页中的“OpenMP 软件执行概述”中介绍了这些函数。

编译器将重整名称 (mangled name) 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。这些重整名称 (mangled name) 因微任务化库的发行版而异，但均会取消重整 (demangle) 为更易于理解的名称。

以下显示的是函数列表中显示的一个典型的由编译器生成的主体函数。

```
7.415      14.860      psec_ -- OMP sections from line 9 [_$s1A9.psec_]
3.873       3.903      craydo_ -- MP doall from line 10 [_$d1A10.craydo_]
```

从上面的示例可以看出，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重整名称 (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，如下所示。

```
                    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采用 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。“函数”标签、“源”标签和“反汇编”标签中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（未调用 `collector_func_load()`），则在函数列表中，动态编译和装入的函数会显示为 `<Unknown>`。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，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“函数”标签、“源”标签和“反汇编”标签中显示函数及其度量。

有关收集器 API 函数的更多信息，请参见第 44 页中的 “动态函数和模块”。

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，虚拟机可以决定使用已解释的一个或多个方法，生成相应的机器码，并执行更有效的机器码版本，而不是解释原始代码。

对于 Java 程序，无需使用收集器 API 函数；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 Java HotSpot 编译的代码，如以下示例所示。

```
                     11.    public int add_int () {
                     12.       int       x = 0;
                         <Function: Routine.add_int()>
2.832     2.832          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。该独占和包含 CPU 时间与各行已解释字节代码的所有独占和包含 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，原因是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。所传递的部分参数是常量的函数调用即是这类调用的其中一个示例。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重整名称 (mangled name)。在上面的示例中，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
```

因为每个克隆函数都具有不同的指令集，所以带注释的反汇编代码列表会分别显示这些克隆函数。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，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移量。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。可在 jsynprog 演示程序的函数列表中找到静态函数的示例，如下所示。

```
0.       0.       <static>@0x18780
0.       0.       <static>@0x20cc
0.       0.       <static>@0xc9f0
0.       0.       <static>@0xd1d8
0.       0.       <static>@0xe204
```

在 "PC" 标签中，上述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 "PC" 标签中，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
```

包含度量

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。

以下示例显示了当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>
```

分支目标

带注释的反汇编代码列表中显示的一行人工行 <branch target> 与指令的某个 PC 对应，在该指令中使用回溯查找它的有效地址失败，原因是该回溯算法遇到了分支目标。

在不运行实验的情况下查看源代码/反汇编代码

无需运行实验，使用 er_src 实用程序就可以查看带注释的源代码和带注释的反汇编代码。输出的生成方式与在分析器中的生成方式相同，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 item tag |
-{cc,scc,dcc} com_spec | -outfile filename | -V ] object
```

object 是可执行文件、共享对象或目标文件（.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、源文件或目标文件的名称。*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。

以下几节将介绍 er_src 实用程序接受的选项。

-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 *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。如果没有可用的源代码，则产生一个不带编译器注释的反汇编代码列表。

-{c,scc,dcc} *com-spec*

指定要显示哪些编译器注释类。*com-spec* 是用冒号隔开的类列表。如果使用的是 -scc 选项，则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，则对反汇编代码注释应用类列表；如果使用的是 -c 选项，则既对源代码注释，也对反汇编代码注释应用类列表。有关这些类的描述，请参见第 104 页中的 “控制源代码和反汇编代码列表的命令”。

注释类可以在缺省文件中指定。首先读取系统范围的 er.rc 缺省文件，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。起始目录中的 .er.rc 文件的缺省值覆盖系统的缺省值，而当前目录中 .er.rc 文件的缺省值覆盖起始目录和系统的缺省值。这些文件也由分析器和 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。有关这些缺省文件的描述，请参见第 119 页中的 “设置缺省值的命令”。缺省文件中除 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，如果文件名为短划线 (-)，则输出会写入 stdout。

-V

打印当前发行版本。

第 9 章

处理实验

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：

- 第 179 页中的 “处理实验”
- 第 180 页中的 “其他实用程序”

处理实验

实验存储在收集器创建的目录中。要处理实验，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被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` 选项可列显 `er_cp` 实用程序的版本。此命令不能复制早于 Forte Developer 7 的软件发行版所创建的实验。

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 选项可列显 er_mv 实用程序的版本。此命令不能移动早于 Forte Developer 7 的软件发行版所创建的实验。

使用 er_rm 实用程序删除实验

删除实验列表或实验组。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 选项都会抑制错误消息并确保成功完成。-V 选项可列显 er_rm 实用程序的版本。此命令可删除早于 Forte Developer 7 的软件发行版所创建的实验。

其他实用程序

在正常情况下，应不必使用其他一些实用程序。在此记录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 [–nqAF] experiment
er_archive -V
```

实验正常完成或在实验上启动性能分析器或 er_print 实用程序时，er_archive 实用程序将自动运行。该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。每个输出文件都以 .archive 后缀命名，并且包含共享对象的函数和模块映射。

如果目标程序异常终止，收集器可能不会运行 er_archive 实用程序。如果要在与记录实验的计算机不同的其他计算机上检查运行时异常终止的实验，则必须在记录数据的计算机的实验上运行 er_archive 实用程序。要确保在将实验复制到的计算机上可以使用装入对象，请使用 -A 选项。

在实验中为所有引用到的共享对象生成了归档文件。这些归档文件包含装入对象中的每个对象文件和函数的地址、大小和名称，以及装入对象的绝对路径和其最后一次修改的时间戳。

如果运行 er_archive 实用程序时找不到共享对象，或者共享对象的时间戳与实验中记录的不同，或 er_archive 实用程序在与记录实验的计算机不同的其他计算机上运行，则归档文件中将包含一则警告。只要手动运行 er_archive 实用程序（不带 -q 标志），警告也同样会写入 stderr。

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，不包括其后代。

–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。

–A

请求将所有装入对象写入实验。该参数可用于生成实验，这些实验很可能被复制到不是记录实验的计算机上。

–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。

–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 选项用于列显版本号信息。

索引

数字和符号

"Index"（索引）标签，83
"Inst–Freq" 标签，82
"PC" 标签，80, 87
@plt 函数，136
<Scalars> 数据对象描述符，158
<Total> 函数
  将时间与执行统计进行比较，133
  说明，156
<Total> 数据对象描述符，158
<Unknown> **函数**
  调用者和被调用者，155
  将 PC 映射到，155

A

addpath 命令，107
analyzer 命令
  JVM 路径 (-j) 选项，74
  JVM 选项 (-J) 选项，74
  版本 (-V) 选项，75
  帮助 (-h) 选项，75
  详细 (-v) 选项，74
  字体大小 (-f) 选项，74
API, 收集器，41

C

C 编译器选项, xhwcprof，81
collect 命令
  Java 版本 (-j) 选项，56-57
  MPI 跟踪 (-m) 选项，54
  版本 (-V) 选项，59
  定期抽样 (-S) 选项，54
  堆跟踪 (-H) 选项，54
  跟踪后续进程 (-F) 选项，55-56
  归档 (-A) 选项，58-59
  基于时钟的分析 (-p) 选项，52
  记录计数数据 (-c) 选项，54-55
  记录样本点 (-l) 选项，57
  模拟运行 (-n) 选项，59
  实验控制选项，55-58
  实验名称 (-o) 选项，59
  实验目录 (-d) 选项，58
  实验组 (-g) 选项，58
  收集数据，51
  输出选项，58-59
collect 命令, 数据收集的时间范围 (-t) 选项，57
collect 命令
  数据收集选项，52, 74
  数据限制 (-L) 选项，59
  数据争用检测 (-r) 选项，55
  死锁检测 (-r) 选项，55
  同步等待跟踪 (-s) 选项，53-54
  详细 (-v) 选项，60
  选项列表，51
  硬件计数器溢出分析 (-h) 选项，52-53
  与 ppgsz 命令一起使用，71
  语法，51
  杂项选项，59-60
  在 exec (-x) 选项之后停止目标，57

collect 命令（续）
  暂停和恢复数据记录 (-y) 选项，58
  自述文件显示 (-R) 选项，59
Collector, 使用 collect 命令运行，51
collectorAPI.h，43
  作为收集器 C 和 C++ 接口的一部分，41
CPU
  选择的列表, 在 er_print 实用程序中，113
  在 er_print 实用程序中选择，114
CPU 过滤，89

D

data_layout 命令，108
data_objects 命令，108
data_single 命令，108
data_sort 命令，109
dbx
  运行收集器，60
  在 MPI 下收集数据，70
dbx collector 子命令
  archive，65
  dbxsample，64
  disable，64
  enable，64
  enable_once（已废弃），66
  hwprofile，61-62
  limit，65
  pause，64
  profile，61
  quit（已废弃），66
  resume，64
  sample，63
  sample record，64
  show，65
  status，66
  store，65
  store filename（已废弃），66
  synctrace，62-63, 63
  tha，63
ddetail 命令，112
deadlocks 命令，111
DTrace 驱动程序
  描述，91
DTrace 驱动程序（续）
  设置访问，91

E

er_archive 实用程序，180
er_cp 实用程序，179
er_export 实用程序，182
er_kernel 实用程序，91
er_mv 实用程序，180
er_print 命令
  addpath，107
  allocs，104
  appendtfile，116
  callers-callees，102
  cc，107
  cmetric_list，116
  cmetrics，102-103
  cpu_list，113
  cpu_select，114
  csingle，103
  csort，103
  data_layout，108
  data_metric_list，116
  data_metrics，108-109
  data_objects，108
  data_single，108
  data_sort，109
  dcc，106-107
  ddetail，112
  deadlocks，111
  disasm，105
  dmetrics，119
  dsort，119-120
  en_desc，120
  exp_list，112
  fsingle，101
  fsummary，101
  functions，99-100
  header，118
  help，122
  ifreq，118
  indx_metric_list，116
  indxobj，110

er_print 命令（续）
- indxobj_define，110-111
- indxobj_list，110
- indxobj_metrics，111
- indxobj_sort，111
- leaks，104
- limit，117
- lines，104
- lsummary，104
- lwp_list，112
- lwp_select，114
- mapfile，121
- metric_list，115
- metrics，100
- name，117
- object_list，114-115
- object_select，115
- objects，118
- outfile，116
- overview，118
- pathmap，108
- pcs，104
- procstats，122
- psummary，104
- quit，122
- races，111
- rdetail，111
- rtabs，120
- sample_list，112
- sample_select，114
- scc，105-106
- script，122
- setpath，107
- sort，101
- source，105
- statistics，118-119
- sthresh，106, 107
- tabs，120
- thread_list，112
- thread_select，114
- tldata，121
- tlmode，120-121
- version，122
- viewmode，117

er_print 实用程序
- 度量关键字，98
- 度量列表，96
- 命令
  - **请参见**er_print 命令
- 命令行选项，96
- 用途，95
- 语法，96

.er.rc 文件，75, 77, 87, 90, 178
er_rm 实用程序，180
er_src 实用程序，177

F

Fortran
- 备用入口点，152
- 收集器 API，41
- 子例程，151

Fortran 函数中的备用入口点，152

G

gprof 实用程序，22

I

indxobj_define 命令，110-111
indxobj_list 命令，110
indxobj_metrics 命令，111
indxobj_sort 命令，111
indxobj 命令，110

J

Java
- 动态编译的方法，44, 155
- 分析限制，47
- 监视器，28
- 设置 er_print 显示输出，117

JAVA_PATH 环境变量，47
Java 虚拟机路径, analyzer 命令选项，74

JDK_HOME 环境变量，47
--jdkhome analyzer 命令选项，74

L

LD_LIBRARY_PATH 环境变量，67
LD_PRELOAD 环境变量，67
libaio.so, 与数据收集交互，40
libcollector.h，42
  作为收集器 Java 编程语言接口的一部分，42
libcollector.so 共享库
  预装入，67
  在程序中使用，41
libcpc.so, 使用，46
libfcollector.h，42
LWP
  过滤，88
  选择的列表, 在 er_print 实用程序中，112
  由 Solaris 线程创建，138
  在 er_print 实用程序中选择，114

M

MPI 程序
  附加到，67
  实验存储问题，69
  实验名称，49, 68, 69
  使用 collect 命令收集数据，70
  使用 dbx 收集数据，70
  收集数据，68
MPI 跟踪
  度量，30
  度量的解释，135
  分析数据包中的数据，134
  跟踪的函数，29
  使用 collect 命令收集数据，54
  预装入收集器库，67
  在 dbx 中收集数据，63
MPI 实验
  存储问题，69
  缺省名称，49
  移动，69

N

NFS，48

O

OpenMP
  度量，147-148
  分析数据, 机器表示，148-149
  分析数据的用户模式显示，142-147
  设置 er_print 显示输出，117
  用户模式调用栈，143
  执行概述，141-149
OpenMP 并行化，165
OpenMP 应用程序中的用户模式调用栈，143

P

PATH 环境变量，47
pathmap 命令，108
PC
  er_print 实用程序中的排序列表，104
  从 PLT，136
  已定义，135
PLT（Program Linkage Table，程序链接表），136
ppgsz 命令，71
prof 实用程序，22

R

races 命令，111
rdetail 命令，111

S

setpath 命令，107
setuid, 使用，41

T

TLB（translation lookaside buffer，转换后备缓冲器）未命中，137, 171

V

viewmode 命令，117

X

-xdebugformat, 设置调试符号信息的格式，37
xhwcprof C 编译器选项，81

“

“查找”工具，87
“调用者与被调用者”标签，78, 87
“度量”标签，85
“反汇编”标签，80
“格式”标签，85
“过滤数据”对话框，88
“函数”标签，77
“函数标签”，87
“路径映射”标签，86
“内存对象”标签，83
“排序”标签，85
“实验”标签，82
“时间线”标签，80, 84, 86
“时间线”菜单，76
“事件”标签，81, 84
“数据布局”标签，82
“数据对象”标签，81
“双重数据源”标签，78
“死锁”标签，77
“死锁详细信息”标签，84
“搜索路径”标签，86
“统计数据”标签，82
“图例”标签，81
“显示/隐藏函数”对话框，87
“泄漏”标签，84
“泄漏列表”标签，81
“行”标签，79, 87
“选择标签”对话框，82, 86-87
“源/反汇编”标签，85
“源”标签，78
“摘要”标签，80, 84
“争用”标签，77
“争用详细信息”标签，84

版

版本信息
  er_cp 实用程序，179
  er_mv 实用程序，180
  er_print 实用程序，122
  er_rm 实用程序，180
  er_src 实用程序，178
  针对 collect 命令，59

包

包含度量
  PLT 指令，136
  递归的影响，36
  对于外联函数，176
  如何计算，135
  使用，33
  说明，34
  已定义，32
包装函数，152

编

编译
  Java 编程语言，38
  “行”分析，37
  带注释的“源代码”和“反汇编”的源代码，37
  调试符号信息的格式，37
  静态链接对数据收集的影响，38
  库的静态链接，38
  用于数据收集的链接，38
  优化对程序分析的影响，38
编译器生成的主体函数
  名称，172

编译器生成的主体函数（续）
  由性能分析器显示，154, 172
编译器优化
  并行化，165
  内联，165
编译器注释，79
  并行化，165
  克隆的函数，175
  描述，163
  内联的函数，165
  通用子表达式删除，164
  显示的过滤类型，164
  循环优化，164
  已定义的类，105-106
  在 er_print 实用程序中为带注释的反汇编代码列表选择，106
  在 er_print 实用程序中为带注释的源代码和反汇编代码列表选择，107
  在 er_print 实用程序中为带注释的源代码列表选择，105
  在 er_src 实用程序中过滤，177

标

标签
  设置一组缺省可见的, 在 er_print 实用程序中，120
  为线程分析器设置一组缺省可见的, 在 er_print 实用程序中，120
  选择以显示，86-87

并

并行执行, 指令，165

程

程序计数器 (program counter, PC), 已定义，135
程序结构, 将调用栈地址映射到，150
程序链接表 (Program Linkage Table, PLT)，136
程序执行
  单线程，135
程序执行（续）
  共享对象和函数调用，136
  描述的调用栈，135
  尾部调用优化，137
  显式多线程，138
  陷阱，137
  信号处理，136

抽

抽样
  选择的列表, 在 er_print 实用程序中，112
  在 er_print 实用程序中选择，114
抽样间隔
  使用 collect 命令设置，54
  已定义，31
  在 dbx 中设置，63
抽样收集器, **请参见**收集器

磁

磁盘空间, 估计实验，50

从

从 er_print 实用程序列显累积统计数据，122

存

存储要求, 估计实验，50

带

带注释的反汇编代码, **请参见**反汇编代码, 带注释的
带注释的源代码, **请参见**源代码, 带注释的

单

单线程程序执行，135

等

等待时间, **请参见**同步等待时间

地

地址空间, 文本和数据区域，151

递

递归函数调用, 度量分配，35

调

调用栈，81
  “时间线”标签中的缺省对齐方式和深度，120
  “事件”标签中，84
  不完全的展开，150
  将地址映射到程序结构，150
  尾部调用优化产生的影响，138
  已定义，135
  在“时间线”标签中，80
  展开，135
调用者-被调用者度量
  er_print 实用程序中的排序顺序，103
  归属, 已定义，32
  在 er_print 实用程序中列显，102
  在 er_print 实用程序中为单个函数列显，103
  在 er_print 实用程序中显示其列表，116
  在 er_print 实用程序中选择，102-103

动

动态编译的函数
  定义，154, 173
  收集器 API，44

独

独占度量
  PLT 指令，136

独占度量（续）
  如何计算，135
  使用，33
  说明，34
  已定义，32

度

度量
  MPI 跟踪，30
  包含
    请参见包含度量
  包含的和独占的，77, 78
  独占
    请参见独占度量
  堆跟踪，29
  归属，78
    请参见归属度量
  函数列表
    请参见函数列表度量
  基于时钟的分析，24, 131
  计时，24
  解释源代码行，167
  解释指令，169
  内存分配，29
  缺省，77
  时间精度，77
  同步等待跟踪，28
  相关性的影响，132
  已定义，23
  硬件计数器, 归属到指令，171
  阈值，80
  阈值, 设置，79

堆

堆跟踪
  度量，29
  缺省度量，77
  使用 collect 命令收集数据，54
  预装入收集器库，67
  在 dbx 中收集数据，63

堆栈帧
来自陷阱处理程序，137
已定义，136
在尾部调用优化中重用，137

多

多线程, 显式，138
多线程应用程序, 将收集器附加到，66

反

反汇编代码, 带注释的
HotSpot 编译的指令，174
Java 本机方法，174
包含度量，176
度量格式，167
对于克隆的函数，175
对于克隆函数，153
分支目标，176
解释，169
可执行文件的位置，49
克隆的函数，175
描述，167
使用 `er_src` 实用程序查看，177
硬件计数器度量归属，171
在 `er_print` 实用程序中列显，105
在 `er_print` 实用程序中设置首选项，106-107
在 `er_print` 实用程序中设置突出显示阈值，107
指令发送相关性，169

方

方法, **请参见**函数

非

非唯一函数名称，152

分

分析, 已定义，23
分析间隔
实验大小, 影响，50
使用 `collect` 命令设置，52, 61
使用 `dbx collector` 命令设置，61
已定义，24
值的限制，45
分析器, **请参见**性能分析器
分析数据包
MPI 跟踪数据，134
大小，50
基于时钟的数据，131
同步等待跟踪数据，133
硬件计数器溢出数据，133
分支目标，176

符

符号表, 装入对象，151

复

复制实验，179

概

概述数据, 在 `er_print` 实用程序中列显，118

高

高度量值
在带注释的反汇编代码中，107
在带注释的源代码中，106

共

共享对象, 之间的函数调用，136

关

关键字, 度量, er_print 实用程序，98

归

归属度量
- 递归的影响，36
- 使用，33
- 说明，34
- 已定义，32

过

过滤 CPU，89
过滤 LWP，88
过滤实验，88
过滤线程，88

函

函数
- @plt，136
- <Total>，156
- <Unknown>，155
- MPI, 跟踪，29
- 包装，152
- 备用入口点 (Fortran)，152
- 地址变化，151
- 定义，151
- 动态编译，44
- 动态编译的，154, 173
- 非唯一, 名称，152
- 静态, 具有重复名称，152
- 静态, 在剥离的共享库中，175
- 静态, 在剥离共享库中，152
- 克隆的，153, 175
- 内联，153
- 全局，151
- 收集器 API，41, 44
- 外联，154, 171
- 系统库, 由收集器插入，40
- 有别名的，151

函数（续）
- 装入对象中的地址，151

函数 PC, 聚集，79, 80, 87
函数调用
- 递归, 度量分配，35
- 共享对象之间，136
- 在单线程程序中，135

函数列表
- 编译器生成的主体函数，172
- 排序顺序, 在 er_print 实用程序中指定，101
- 在 er_print 实用程序中列显，99-100

函数列表度量
- 在 .er.rc 文件中设置缺省排序顺序，119-120
- 在 .er.rc 文件中选择缺省值，119
- 在 er_print 实用程序中显示其列表，115
- 在 er_print 实用程序中选择，100

函数名称, C++, 在 er_print 实用程序中选择长形式或短形式，117
函数重新排序，90

后

后续进程
- 实验名称，49
- 实验位置，48
- 收集器所跟踪的，47
- 收集所有后续进程的数据，55
- 数据收集的限制，47
- 为选定的进程收集数据，66

后续实验
- 设置读取模式, 在 er_print 实用程序中，120
- 装入，73

环

环境变量
- JAVA_PATH，47
- JDK_HOME，47
- LD_LIBRARY_PATH，67
- LD_PRELOAD，67
- PATH，47

恢

恢复收集数据, 从程序中，44
恢复数据收集
  在 dbx 中，64
  针对 collect 命令，58

回

回溯，26

基

基于时钟的分析
  比较gethrtime 和 gethrvtime，132
  度量，24, 131
  度量的准确性，133
  分析数据包中的数据，131
  间隔
    请参见分析间隔
  缺省度量，77
  使用 collect 命令收集数据，52
  已定义，24
  因开销而失真，132
  在 dbx 中收集数据，61

间

间隔, 抽样, **请参见**抽样间隔
间隔, 分析, **请参见**分析间隔

将

将路径附加到文件，107
将收集器附加到正在运行的进程，66
将装入对象归档到实验中，58, 65

进

进程地址空间文本和数据区域，151

静

静态函数
  在剥离的共享库中，175
  在剥离共享库中，152
  重复名称，152
静态链接, 对数据收集的影响，38

克

克隆的函数，153, 175

库

库
  collectorAPI.h，43
  libaio.so，40
  libcollector.so，41, 67
  libcpc.so，40, 46
  libthread.so，40
  MPI，40, 68
  剥离的共享, 和静态函数，175
  剥离共享, 和静态函数，152
  插入，40
  静态链接，38
  系统，40

快

快速陷阱，137

列

列显当前路径，107

路

路径前缀映射，86

命

命名实验，48

内

内存对象, 已定义，109
内存分配，29
  对数据收集的影响，39
  和泄漏，81
内存泄漏, 定义，29
内核分析
  分析特定的进程或内核线程，93-94
  设置系统，91
内核分析数据, 分析，94
内核实验
  数据类型，91
  字段标签含义，94
内核时钟分析，92
内联函数，153

排

排序顺序
  调用者-被调用者度量, 在 er_print 实用程序中，103
  函数列表, 在 er_print 实用程序中指定，101

缺

缺省度量，77
缺省值, 在缺省文件中设置，119

人

人工函数, 在用户模式调用栈中，142

入

入口点, 备用, 在 Fortran 函数中，152

删

删除实验或实验组，180

实

实验
  另请参见实验目录
  er_print 实用程序中的标头信息，118
  MPI 存储问题，69
  从程序中终止，44
  存储位置，58, 65
  存储要求, 估计，50
  打开，73
  定义的，48
  多个，73
  复制，179
  附加当前路径，107
  归档装入对象，58, 65
  后续, 装入，73
  命名，48
  缺省名称，48
  删除，180
  设置路径以查找文件，107
  数据聚集，73
  添加，73
  为 Java 和 OpenMP 设置模式，117
  位置，48
  限制大小，59, 65
  移动，49, 180
  移动 MPI，69
  预览，74
  在 er_print 实用程序中列出，112
  重映射路径前缀，108
  组，48
实验, 后续, 设置读取模式, 在 er_print 实用程序中，120
实验过滤，88
实验名称，48
  MPI, 使用 MPI_comm_rank 和脚本，71
  MPI 缺省，49
  MPI 缺省的，68
  局限，48
  缺省，48
  在 dbx 中指定，65

实验目录
缺省，48
使用 collect 命令指定，58
在 dbx 中指定，65
实验组，48
创建，73
定义的，48
多个，73
名称限制，48
缺省名称，48
删除，180
使用 collect 命令指定名称，58
添加，73
预览，74
在 dbx 中指定名称，65

时

时间度量, 精度，77

事

事件
“时间线”标签中的缺省显示类型，120
显示在“时间线”标签中，80
事件标记，80

视

视图模式, 说明，85

收

收集器
API, 在程序中使用，41, 42
附加到正在运行的进程，66
已定义，20, 23
在 dbx 中禁用，64
在 dbx 中启用，64
在 dbx 中运行，60
收集实验, 预览命令，89

输

输出文件
关闭, 在er_print 实用程序中，116
关闭并打开新的, 在 er_print 实用程序中，116
输入文件
er_print 实用程序，122
在 er_print 实用程序中终止，122

数

数据表示
保存设置，87
设置选项，84
数据对象
<Scalars> 描述符，158
<Total>描述符，158
布局，82
范围，157
描述符，158
设置排序度量，109
已定义，157
在硬件计数器溢出实验中，108
数据空间分析, 数据对象，157
数据类型，24
MPI 跟踪，29
堆跟踪，29
基于时钟的分析，24
缺省, 在“时间线”标签中，121
同步等待跟踪，28
硬件计数器溢出分析，26
数据派生的度量, 在 er_print 实用程序中设置，108-109
数据派生度量, 在 er_print 实用程序中显示其列表，116
数据收集
MPI 程序, 使用 collect 命令，70
MPI 程序, 使用 dbx，70
程序控制，41
从 MPI 程序中，68
从程序中恢复，44
从程序中禁用，44
从程序中控制，41
从程序中暂停，43
动态内存分配的影响，39

数据收集（续）
段故障，39
链接，38
使用 collect 命令，51
使用 dbx，60
速率，50
在 dbx 中恢复，64
在 dbx 中禁用，64
在 dbx 中启用，64
在 dbx 中暂停，64
针对 collect 命令恢复，58
针对 collect 命令暂停，58
准备程序，38
数据收集过程中出现的段故障，39
数据争用
列表，111
详细信息，111

死
死锁
列表，111
详细信息，112

搜
搜索源文件和目标文件，86

索
索引对象，110
定义，110-111
度量，111
列表，110
排序，111
索引对象度量, 在 er_print 实用程序中显示其列表，116
索引行，162
“源”标签中，162
在 er_print 实用程序中，105
在“反汇编”标签中，80, 168
在“源”标签中，79, 168
索引行, 特殊
HotSpot 编译的指令，174
Java 本机方法，174
编译器生成的主体函数，172
不带行号的指令，173
外联函数，171

替
替代源上下文，105

通
通用子表达式删除，164

同
同步等待跟踪
等待时间，28, 133
度量，28
分析数据包中的数据，133
使用 collect 命令收集数据，53
已定义，28
阈值
请参见阈值, 同步等待跟踪
预装入收集器库，67
在 dbx 中收集数据，62
同步等待时间
度量, 已定义，28
已定义，28, 133
同步延迟跟踪, 缺省度量，77
同步延迟事件
定义的度量，28
分析数据包中的数据，133
已定义，28

外
外联函数，154, 171

网

网络磁盘，48

微

微态
  服务于度量，131
  切换，137
微状态，84

尾

尾部调用优化，137

文

文档, 访问，15-16
文档索引，15
文件路径，107

显

显式多线程，138

线

线程
  创建，138
  工作，138
  库，40
  选择的列表, 在 er_print 实用程序中，112
  在 er_print 实用程序中选择，114
线程分析器, 设置一组缺省可见的, 在 er_print 实用程序中，120
线程过滤，88

限

限制
  Java 分析，47
  分析间隔值，45
  后续进程的数据收集，47
  实验名称，48
  实验组名称，48
限制实验大小，59, 65

陷

陷阱，137

相

相关性, 对度量的影响，132

泄

泄漏, 内存, 定义，29

信

信号
  对处理程序的调用，136
  分析，40
  分析, 从 dbx 传递到 collect 命令，57
  用于通过 collect 命令手动抽样，57
  用于通过 collect 命令暂停和恢复，58
信号处理程序
  用户程序，40
  由收集器安装，40, 136

性

性能度量, **请参见**度量
性能分析器
  "Index"（索引）标签，83
  "Inst–Freq" 标签，82
  "PC" 标签，80, 87

性能分析器（续）
“帮助”菜单，76
“查找”工具，87
“调用者与被调用者”标签，78, 87
“度量”标签，85
“反汇编”标签，80
“格式”标签，85
“过滤数据”对话框，88
“函数”标签，77, 87
“路径映射”标签，86
“内存对象”标签，83
“排序”标签，85
“实验”标签，82
“时间线”标签，80, 84, 86
“时间线”菜单，76
“事件”标签，81, 84
“视图”菜单，75
“数据布局”标签，82
“数据对象”标签，81
“双重数据源”标签，78
“死锁”标签，77
“死锁详细信息”标签，84
“搜索路径”标签，86
“统计数据”标签，82
“图例”标签，81
“文件”菜单，75
“显示/隐藏函数”，87
“泄漏”标签，84
“泄漏列表”标签，81
“行”标签，79, 87
“源/反汇编”标签，85
“源”标签，78
“摘要”标签，80, 84
“争用”标签，77
“争用详细信息”标签，84
定义，73
记录实验，74
命令行选项，74
启动，73
缺省值，90
要显示的标签，84
已定义，20
性能数据, 转换为度量，23

选
选项, 命令行, er_print 实用程序，96

循
循环优化，164

样
样本
包中包含的信息，31
从程序中记录，43
当 dbx 停止进程时记录，64
记录情况，31
间隔
请参见抽样间隔
使用 collect 命令定期记录，54
使用 collect 手动记录，57
已定义，31
在 dbx 中定期记录，63
在 dbx 中手动记录，64
样本点, 显示在“时间线”标签中，80
样本过滤，88

叶
叶 PC, 已定义，135

移
移动实验，49, 180

异
异步 I/O 库, 与数据收集交互，40

溢
溢出值, 硬件计数器, **请参见**硬件计数器溢出值

映

映射文件
  生成，89
  使用 er_print 实用程序生成，121

硬

硬件计数器
  获取列表，51, 62
  计数器名称，52
  列表说明，27
  使用 collect 命令选择，52
  使用 dbx collector 命令选择，62
  数据对象和度量，108
  已定义，26
  溢出值，26
硬件计数器度量, 显示在“数据对象”标签中，81
硬件计数器库, libcpc.so，46
硬件计数器列表
  使用 collect 命令获取，51
  使用 dbx collector 命令获取，62
  原始计数器，28
  周知计数器，27
  字段说明，27
硬件计数器属性选项，53
硬件计数器溢出分析
  分析数据包中的数据，133
  缺省度量，77
  使用 collect 命令收集数据，52
  使用 dbx 收集数据，61
  已定义，26
硬件计数器溢出值
  过小或过大产生的结果，133
  实验大小, 效果，50
  使用 collect 设置，53
  已定义，26
  在 dbx 中设置，62

优

优化
  程序分析的影响，38
  通用子表达式删除，164
优化（续）
  尾部调用，137

由

由收集器插入系统库函数，40

有

有别名的函数，151

语

语法
  er_archive实用程序，180
  er_export实用程序，182
  er_print 实用程序，96
  er_src 实用程序，177

阈

阈值, 同步等待跟踪
  对收集开销的影响，133
  使用 collect 命令设置，54, 63
  使用 dbx collector 设置，62
  校准，28
  已定义，28
阈值, 突出显示
  在带注释的反汇编代码中, er_print 实用程序，107
  在带注释的源代码中, er_print 实用程序，106

预

预装入 libcollector.so，67

原

原始硬件计数器，27, 28

源
源代码, 编译器注释，79
源代码, 带注释的
　编译器生成的主体函数，172
　编译器注释，163
　不带行号的指令，173
　从源代码中分辨注释，161
　度量格式，167
　对于克隆函数，153
　解释，167
　克隆的函数，175
　描述，161, 166
　使用 er_src 实用程序查看，177
　使用中间文件，150
　索引行，162
　外联函数，171
　源文件的位置，49
　在 er_print 实用程序中列显，105
　在 er_print 实用程序中设置编译器注释类，105
　在 er_print 实用程序中设置突出显示阈值，106
　在性能分析器中查看，161
源代码和反汇编代码, 带注释的, 在 er_print 实用程序中设置首选项，107
源代码行, er_print 实用程序中的排序列表，104

约
约束, **请参见**限制

在
在 er_print 实用程序中设置读取后续实验的模式，120
在 er_print 实用程序中限制输出，117

暂
暂停数据收集
　从程序中，43
　在 dbx 中，64
　针对 collect 命令，58

摘
摘要度量
　对于单个函数，在 er_print 实用程序中列显，101
　对于所有函数, 在 er_print 实用程序中列显，101

展
展开调用栈，135

帧
帧, 堆栈, **请参见**堆栈帧

执
执行统计, 将时间与函数进行比较，133
执行统计数据, 在 er_print 实用程序中列显，118-119

指
指令发送
　分组, 对带注释的反汇编代码的影响，169
　延迟，170
指令频率, 在 er_print 实用程序中列显列表，118

中
中间文件, 用于带注释的源代码列表，150

重
重新映射路径前缀，86
重映射路径前缀，108

周

周知硬件计数器，27

主

主动回溯，81
主体函数, 编译器生成的
    名称，172
    由性能分析器显示，154, 172

装

装入对象
    符号表，151
    函数的地址，151
    内容，151
    写入布局，108
    选择的列表, 在 er_print 实用程序中，114-115
    已定义，151
    在 er_print 实用程序中列显列表，118
    在 er_print 实用程序中选择，115

子

子例程, **请参见**函数