性能分析器

Sun™ Studio 8

Sun Microsystems, Inc.
www.sun.com

部件号码 817-5804-10
2004 年 4 月 , 修订版 A

关于本文档的建议请发到：http://www.sun.com/hwdocs/feedback

版权所有 © 2004 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. 保留所有权利。

Sun Microsystems, Inc. 具有与本文档所描述的产品中所含技术相关的知识产权。需特别指出的是 （但不局限于此），这些知识产权中可能包括一项或多项在 `http://www.sun.com/patents` 上列出的美国专利，以及一项或多项在美国和其它国家 （地区）的其它专利或待批的专利申请。

本文档及其相关产品依据限制其使用、复制、分发和反编译的许可证进行分发。未经 Sun 及其许可方 （如果存在）的事先书面授权，不得以任何形式、任何手段复制本文档或产品的任何部分。

第三方软件 （包括字体技术）的版权归 Sun 供应商所有并由他们授权。

该产品的部分内容可能出自 Berkeley BSD 系统，由加州大学 (University of California) 授权。 UNIX 是在美国和其它国家 （地区）的注册商标，由 X/Open Company, Ltd. 独家授权。

Sun、 Sun Microsystems、 Sun 徽标、 Forte、 Java、 Solaris、 iPlanet、 NetBeans 以及 docs.sun.com 是 Sun Microsystems, Inc. 在美国和其它国家（地区）的商标或注册商标。

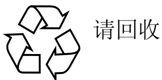
所有的 SPARC 商标均需获得授权才能使用，它们是 SPARC International, Inc. 在美国和其它国家 （地区）的商标或注册商标。带有 SPARC 商标的产品所基于的体系结构是由 Sun Microsystems, Inc 开发的。

Netscape 和 Netscape Navigator 是 Netscape Communications Corporation 在美国和其它国家 （地区）的商标或注册商标。

Sun `f90/f95` 的部分内容出自 Cray CF90™ （Cray Inc. 的产品）。

`libdwarf` 和 `lidredblack` 在 2000 年注册的版权归 Silicon Graphics Inc. 所有，您可以依据 GNU Lesser General Public License 从 `http://www.sgi.com` 获得。

本文档按 “原样”提供，对所有明示或默示的条件、陈述和担保，包括对适销性、特殊用途的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。



请回收

目录

开始之前 13

本书的结构 13

排版惯例 14

Shell 提示符 15

访问 Sun Studio 软件和手册页 15

- 访问编译器和工具 15
- 访问手册页 16
- 访问集成开发环境 17

访问编译器和工具文档 17

- 易读格式的文档 18
- 相关编译器和工具文档 18

访问相关 Solaris 文档 19

开发人员资源 19

与 Sun 技术支持联系 20

发送意见 20

1. 性能分析器概述 21

从集成开发环境启动性能分析器 21

性能分析的工具 21

- 收集器工具 22

性能分析器工具 22
er_print 命令 22
prof、gprof 和 tcov 工具 23
性能分析器窗口 23

2. 性能数据 25

使用联机帮助 25
收集器收集何种数据 26
时钟数据 27
硬件计数器溢出分析数据 28
同步等待跟踪数据 31
堆跟踪（内存分配）数据 31
MPI 跟踪数据 32
全局（样本）数据 34
度量如何分配到程序结构 34
函数级度量：排除、包括和属性 35
解释属性度量：示例 36
递归如何影响函数级度量 37

3. 收集性能数据 39

编译和链接程序 39
源码信息 39
静态链接 40
优化 40
编译 Java 程序 40
为数据收集和分析准备程序 41
使用动态分配的内存 41
使用系统库 42
使用信号处理程序 42

使用 setuid 43
数据收集的程序控制 43
C、C++、Fortran 和 Java API 函数 45
动态函数和模块 47
数据收集的局限 48
基于时钟的分析的局限 48
收集跟踪数据的局限 48
硬件计数器溢出分析的局限 49
硬件计数器溢出分析中的运行时失真和扩大 49
后续进程中数据收集的局限 50
Java 分析的局限 50
用 Java 编程语言所编写应用程序的运行时性能失真和扩大 51
数据存储的位置 51
实验名称 51
移动实验 52
估计存储需求 53
收集数据 54
使用 collect 命令收集数据 54
数据收集选项 55
实验控制选项 57
输出选项 59
其它选项 61
使用 dbx collector 子命令收集数据 61
数据收集子命令 62
实验控制子命令 64
输出子命令 65
信息子命令 66
收集运行进程中的数据 66

从 MPI 程序收集数据 69
存储 MPI 实验 69
在 MPI 下运行 collect 命令 71
通过在 MPI 下启动 dbx 来收集数据 71
与 ppgsz 一起使用 collect 72

4. er_print 命令行性能分析工具 73

er_print 语法 74
度量列表 74
控制函数列表的命令 77
控制调用方与被调用方列表的命令 78
控制泄漏和分配列表的命令 80
控制源码和反汇编列表的命令 80
控制数据空间列表的命令 84
列出实验、样本、线程和 LWP 的命令 84
控制选择的命令 86
控制负载对象选择的命令 87
列出度量的命令 88
控制输出的命令 88
打印其它显示的命令 89
设置缺省值的命令 90
设置仅用于性能分析器缺省的命令 91
杂项命令 92
示例 93

5. 理解性能分析器及其数据 95

数据收集如何工作 95
实验格式 96
记录实验 97

解释性能度量 98
基于时钟的分析 98
同步等待跟踪 101
硬件计数器溢出分析 101
堆跟踪 102
MPI 跟踪 102
调用栈和程序执行 103
单线程执行和函数调用 103
显式多线程 105
基于 Java 技术软件执行的概述 106
Java 处理表示 107
并行执行和编译器生成的主体函数 108
不完整的堆栈解除 112
将地址映射到程序结构 113
进程映像 113
负载对象和函数 113
有别名的函数 114
非唯一函数名称 114
源于剥离共享库的静态函数 114
Fortran 替代的入口点 115
克隆的函数 115
内联函数 115
编译器生成的主体函数 116
外联函数 116
动态编译的函数 117
<Unknown> 函数 117
<no Java callstack recorded> 函数 118
<Total> 函数 118

将数据地址映射到程序数据对象 119
Dataobject 描述符 119
注释代码列表 120
注释源代码 121
注释反汇编码 123

6. 操作实验和查看注释的代码列表 129
操作实验 129
用 er_src 查看带注释的代码列表 130
其他公用程序 132
er_archive 公用程序 132
er_export 公用程序 133

A. 使用 prof、gprof 和 tcov 来分析程序 135
使用 prof 生成程序分析 136
使用 gprof 生成调用图分析 138
将 tcov 用于语句级分析 140
创建 tcov 的分析共享库 143
锁定文件 144
tcov 运行时函数报告的错误 144
将 tcov 增强版用于语句级分析 145
为 tcov 增强版创建分析共享库 146
锁定文件 147
tcov 目录和环境变量 147

索引 149

图

图 2-1 描述排除、包括和属性度量的调用树 36

图 5-1 包含并行 Do 或并行 For 构造的多线程程序调用树示意图 110

图 5-2 带有共享任务 Do 或共享任务 For 构造的并行区域调用树示意图 111

表

表 P-1 字体惯例 14
表 P-2 代码惯例 14
表 2-1 定时度量 27
表 2-2 可用于 SPARC 和 IA 硬件的具有别名的硬件计数器 29
表 2-3 同步等待跟踪度量 31
表 2-4 内存分配（堆跟踪）度量 32
表 2-5 MPI 跟踪度量 33
表 2-6 MPI 函数的类别有发送、接收、发送和接收以及其它 33
表 3-1 `collector_func_load()` 的参数列表 47
表 3-2 预装库 `libcollector.so` 的环境变量设置 68
表 4-1 `er_print` 命令的选项 74
表 4-2 度量类型字符 75
表 4-3 度量可视性字符 75
表 4-4 度量名称字符串 76
表 4-5 编译器注释消息类 82
表 4-6 `dcc` 命令的附加选项 83
表 4-7 时间线显示模式选项 91
表 4-8 时间线显示数据类型 92
表 5-1 数据类型和相应的文件名称 96
表 5-2 内核微态如何贡献给基值 99
表 5-3 注释源代码度量 122
表 A-1 性能分析工具 135

开始之前

本手册描述了 Sun™ Studio 8 中的性能分析工具。

- 收集器和性能分析器这对工具可用于执行大范围性能数据的统计分析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。
- prof 和 gprof 是执行 CPU 使用率的统计分析和在函数级提供执行频率的工具。
- tcov 是在函数和源代码行级提供执行频率的工具。

本手册的适用对象是对 Fortran、 C、 C++ 或 Java™、 Solaris™ 操作系统和 UNIX® 操作系统命令熟悉的应用程序开发人员。掌握一些性能分析的知识有助于运用这些工具，不过这一点并不是必须的。

本书的结构

第 1 章介绍了性能分析工具，简要讨论了这些工具的用途和使用时机。

第 2 章描述了收集器收集的数据和将这些数据转换成性能度量的方式。

第 3 章描述了如何使用收集器从您的程序收集时间数据、同步延迟数据和硬件事件数据。

第 4 章描述了如何使用 er_print 命令行界面来分析 收集器收集的数据。

第 5 章描述了将收集器收集的数据转换成性能度量的过程，以及如何将这些度量关联到程序结构。

第 6 章介绍了关于公用程序的信息，无需运行实验，该程序就可以操作和转换性能实验并查看已注释的源代码和反汇编代码。

附录 A 描述了 UNIX 分析工具 prof、gprof 和 tcov。这些工具提供了时间信息和执行频率统计。

排版惯例

表 **P-1** 字体惯例

字体	含义	示例
`AaBbCc123`	命令、文件和目录的名称；计算机屏幕输出	编辑您的 `.login` 文件。 使用 `ls -a` 列出所有文件。 `% You have mail.`
`AaBbCc123`	键入的内容，以便与计算机屏幕输出相区别	`%` **`su`** `Password:`
AaBbCc123	书名、新词或术语以及要强调的词	请阅读 *《用户指南》*的第 6 章。 这些称作*类*选项。 您*必须*是超级用户才能执行此项操作。
AaBbCc123	命令行占位符文本；用实际的名称或值替换	要删除文件，请输入 **`rm`** *filename*。

表 **P-2** 代码惯例

代码符号	含义	表示法	代码示例
[ ]	方括号包含可选参数。	O[*n*]	-O4, -O
{ }	大括号包含一组可供选择的选项。	d{y|n}	-dy
|	分隔变量的 "|" 或 "-" 符号，只能选择其一。	B{dynamic|static}	-Bstatic
:	冒号，和逗号一样，有时用于分隔参数。	R*dir*[:*dir*]	-R/local/libs:/U/a
…	省略号，表示一系列省略。	-xinline=*f1*[,…*fn*]	-xinline=alpha,dos

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 的超级用户	#

访问 Sun Studio 软件和手册页

编译器和工具及其手册页并没有安装到标准的 /usr/bin/ 和 /usr/share/man 目录中。要访问编译器和工具，必须正确设置 PATH 环境变量（请参阅第 15 页的 "访问编译器和工具"）。要访问手册页，必须正确设置 PATH 环境变量（请参阅第 15 页的 "访问编译器和工具"）。

关于 PATH 变量的更多信息，请参阅 csh(1)、sh(1) 和 ksh(1) 手册页。有关 MANPATH 变量的详细信息，请参阅 man(1) 手册页。关于设置 PATH 变量和 MANPATH 变量以访问此发行版本的更多信息，请参阅安装指南或询问系统管理员。

注 – 本节中的信息假设您的 Sun Studio 编译器和工具被安装在 /opt 目录中。如果软件未安装在 /opt 目录下，请向系统管理员询问系统的等价路径。

访问编译器和工具

使用下列步骤来决定是否需要更改 PATH 变量以访问编译器和工具。

▼ 要决定是否需要设置 PATH 环境变量

1. **通过在命令提示符后输入下列内容以显示 PATH 变量的当前值。**

```
% echo $PATH
```

2. **查看输出中是否有包含** /opt/SUNWspro/bin/ **的路径字符串。**

   如果找到该路径，您的 PATH 变量已经设置好，可以访问编译器和工具了。如果没有找到该路径，按照下一步中的指示来设置 PATH 环境变量。

▼ 要设置 PATH 环境变量以访问编译器和工具

1. **如果使用的是 C shell，请编辑起始** .cshrc **文件。如果使用的是 Bourne shell 或 Korn shell，请编辑起始** .profile **文件。**

2. **将下列内容增加到** PATH **环境变量。如果已安装了 Sun ONE Studio 或 Forte Developer 软件，则将以下路径增加在这些安装的路径之前。**

   /opt/SUNWspro/bin

访问手册页

使用下列步骤来决定是否需要更改 MANPATH 变量以访问手册页面。

▼ 要决定是否需要设置 MANPATH 环境变量

1. **通过在命令提示符后输入下列内容以请求** dbx **手册页。**

```
% man dbx
```

2. **如果有输出的话，请查看输出。**

   如果 dbx(1) 手册页无法找到或者显示的手册页不是用于安装软件的当前版本，请按照下一步中的指示来设置 MANPATH 环境变量。

▼ 要设置 MANPATH 环境变量 以访问手册页

1. **如果使用的是 C shell，请编辑起始** .cshrc **文件。如果使用的是 Bourne shell 或 Korn shell，请编辑起始** .profile **文件。**

2. **将下列内容增加到** MANPATH **环境变量。**

   /opt/SUNWspro/man

访问集成开发环境

Sun Studio 8 集成开发环境 (IDE) 提供了各种模块，用于创建、编辑、生成、调试和分析 C、 C++ 或 Fortran 应用程序的性能。

IDE 需要 Sun Studio 8 的 Core Platform 组件。如果 Core Platform 组件没有安装在以下位置，则必须将 `SPRO_NETBEANS_HOME` 环境变量设置到安装 Core Platform 组件的位置 (*installation_directory*/`netbeans/3.5R`)：

- 缺省安装目录 `/opt/netbeans/3.5R`
- 与 Sun Studio 8 编译器和工具组件相同的位置 （例如，编译器和工具组件安装在 `/foo/SUNWspro` 而 Core Platform 组件安装在 `/foo/netbeans/3.5R`

启动 IDE 的命令是 `sunstudio`。关于该命令的详细信息，请参阅 `sunstudio` (1) 手册页。

访问编译器和工具文档

您可以通过下列位置访问该文档：

- 该文档可从与本地系统或网络上的软件一同安装于 `file:/opt/SUNWspro/docs/index.html` 的文档索引中找到。

  如果软件未安装在 `/opt` 目录下，请向系统管理员询问系统的等价路径。
- 多数手册可从 `docs.sun.com`sm web 站点中找到。下列书目只能从您所安装的软件中找到：
  - 《标准 C++ 类库参考》
  - 《标准 C++ 库用户指南》
  - 《Tools.h++ 类库参考》
  - 《Tools.h++ 用户指南》
- 发行版本说明可以从 `docs.sun.com` web 站点上获得。
- IDE 所有组件的联机帮助可以通过 [帮助] 菜单获得，也可以通过 IDE 中许多窗口和对话框中的 [帮助] 按钮获得。

`docs.sun.com` web 站点 (`http://docs.sun.com`) 使您可以通过因特网阅读、打印和购买 Sun Microsystems 的手册。如果找不到手册，请参见与本地系统或网络上的软件一同安装的文档索引。

注 –Sun 公司不对本文档所提及的第三方 web 站点的可用性负责，而且 Sun 公司不认可也不对以上站点或资源上的任何内容、广告、产品或其它资料承担责任。此外， Sun 公司也不会因您使用或依靠以上任何站点或资源上的 （或通过该站点或资源所获取的） 内容、货物或服务所产生的 （或所谓产生的）任何损失承担责任。

易读格式的文档

该文档以易读格式提供，以方便残障用户使用辅助技术进行阅读。您还可以按照下表所描述的信息找到文档的易读版本。如果软件未安装在 `/opt` 目录下，请向系统管理员询问系统的等价路径。

文档类型	易读版本的格式和位置
手册 （除第三方手册）	HTML，位于 `http://docs.sun.com`
第三方手册： • 《标准 C++ 类库参考》 • 《标准 C++ 库用户指南》 • 《Tools.h++ 类库参考》 • 《Tools.h++ 用户指南》	HTML，位于 `file:/opt/SUNWspro/docs/index.html` 文档索引的已安装软件中
自述文件和手册页	HTML，位于 `file:/opt/SUNWspro/docs/index.html` 文档索引的已安装软件中
联机帮助	HTML 可以通过 IDE 中的 [帮助] 菜单获得
发行说明	HTML，位于 `http://docs.sun.com`

相关编译器和工具文档

下表描述的相关文档可以在 `file:/opt/SUNWspro/docs/index.html` 和 `http://docs.sun.com` 上获得。如果软件未安装在 `/opt` 目录下，请向系统管理员询问系统的等价路径。

	描述
OpenMP API 用户指南	有关用于使程序并行化的编译器指令的信息。
Fortran 编程指南	讨论并行性、优化、创建共享库等编程技术。
使用 `dbx` 调试程序	调试器的使用参考手册，该手册提供了有关附加和拆离 Solaris 进程以及在受控环境中执行程序的信息。
语言用户的指南	描述编译和编译器选项。

访问相关 Solaris 文档

下表描述了可从 docs.sun.com web 站点上获得的相关文档。

文档集合	文档标题	描述
Solaris 参考手册集合	请参阅手册页部分的标题。	提供了关于 Solaris 操作系统的信息。
Solaris Software Developer Collection	*链接程序和库指南*	描述了 Solaris 链接编辑器和运行时链接程序的操作。
Solaris Software Developer Collection	*Multithreaded Programming Guide*	涵盖了 POSIX 和 Solaris 线程 API、使用同步对象进行程序设计、编译多线程程序和多线程程序的查找工具。
Solaris Software Developer Collection	*SPARC 汇编语言参考手册*	描述用于 SPARC® 处理器的汇编语言。
Solaris 9 更新集合	*Solaris 可调参数参考手册*	提供关于 Solaris 可调参数的参考信息。

开发人员资源

您可以访问 http://developers.sun.com/prodtech/cc 找到这些最新资源：

- 关于编程技术和最佳方法的文章
- 短小编程提示的知识库
- 编译器和工具组件的文档以及与软件同时安装的文档的更正
- 支持等级信息
- 用户论坛
- 可下载代码示例
- 新技术预览

您可以在 http://developers.sun.com 上找到开发人员的额外资源。

与 Sun 技术支持联系

如果您有关于本产品的技术问题而本文档未予以解答，请访问：

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

发送意见

Sun 致力于提高文档质量，并欢迎您提出宝贵的意见和建议。请通过电子邮件将您的意见发送至以下地址：

`docfeedback@sun.com`

请在电子邮件的主题行注明文档的部件号码 (817-5804-10) 。

第 1 章

性能分析器概述

开发高性能应用程序需要将编译器特性、已优化函数的库和性能分析的工具整合在一起。*性能分析器*手册描述的这些工具有助于评估代码的性能，标识潜在的性能问题并且定位出现这些问题的代码部分。

从集成开发环境启动性能分析器

关于从集成开发环境 (IDE) 启动性能分析器的信息，请参阅缺省安装目录下文档索引中的 “性能分析器自述文件”：`file:/opt/SUNWspro/docs/index.html`。如果 Sun Studio 8 编译器和工具未安装在 `/opt` 目录中，请询问系统管理员以了解系统中的等价路径。

性能分析的工具

本手册主要涉及收集器和性能分析器这一对工具，它们用于收集和分析应用程序的性能数据。从命令行或图形用户界面都可以使用这两个工具。

收集器和性能分析器的设计使用面向所有软件开发人员，即使他们的主要职责并不一定必须是性能调节。这些工具提供了比常用的分析工具 `prof` 和 `gprof` 更灵活、详细和准确的分析，并且不会受 `gprof` 中的属性错误影响。

这两个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负载对象？
- 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？
- 函数或负载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。这些数据可以包括调用栈、微态计算信息、线程同步延迟数据、硬件计数器溢出数据、消息传递接口 (MPI) 函数调用数据、内存分配数据和操作系统及进程的汇总信息。收集器可以收集 C、 C++ 和 Fortran 程序的各种数据，以及收集用 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。关于收集的数据信息请参阅第 2 章，而关于收集器的详细信息请参阅第 3 章。从性能分析器 GUI、 IDE、 dbx 命令行工具和使用 collect 命令都可以运行收集器。

性能分析器工具

性能分析器工具显示了收集器记录的数据，以便您分析这些信息。性能分析器处理数据并显示在程序、函数、源代码行和指令级别的各种性能度量。这些度量分为五类：

- 定时度量
- 硬件计数器度量
- 同步延迟度量
- 内存分配度量
- MPI 跟踪度量。

性能分析器也以图形格式显示了作为时间函数的原始数据。性能分析器可以创建映射文件，用于改善函数在程序地址空间中装入的顺序。

关于性能分析器的详细信息请参阅 IDE 或性能分析器 GUI 中的联机帮助，关于命令行分析工具 er_print 的信息请参阅第 4 章。

第 5 章讨论了有关理解性能分析器及其数据的主题，包括：数据收集如何工作，解释了性能度量、调用栈和程序执行，还有已注释的代码列表。包括编译器注释但不包括性能数据的已注释源代码列表和反汇编代码列表可以用 er_src 公用程序来查看 （更多信息请参阅第 6 章）。

er_print 命令

er_print 命令以纯文本显示了除 [时间线] 之外性能分析器显示的所有内容。

prof、 gprof 和 tcov 工具

本手册也包括关于以下性能工具的信息：

- prof 和 gprof

  prof 和 gprof 是生成分析数据的 UNIX® 工具， Solaris 7、 8 和 9 操作系统 (SPARC® *Platform Edition*) 都包括有这两个工具。 x86 平台也提供并支持这两个工具。

- tcov

  tcov 是代码覆盖工具，用于报告每个函数被调用和每个源代码行被执行的次数。

关于 prof、 gprof 和 tcov 的更多信息，请参阅附录 A。

性能分析器窗口

注 – 以下是性能分析器窗口的简要概述。关于下文讨论的标签的功能和特性的完整和详细讨论，请参阅联机帮助。

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程序的函数列表，该程序具有每个函数的排除和包括度量。该列表可以按负载对象、线程、轻量进程 (LWP) 和时间片过滤。

对于选中的函数，另一个标签会显示函数的调用方和被调用方。该标签可用于导航调用树，例如在其中寻找高度量值。

另外两个标签显示了用性能度量逐行注释和与编译器注释交叉的源代码，还显示了用每个指令的度量注释和与可用的源代码和编译器注释交叉的反汇编代码。

性能数据在另一个标签中显示为时间函数。

其他标签显示了实验和负载对象的详细信息，函数的汇总信息和进程的统计数据。

从键盘和鼠标都可以导航性能分析器。

第 2 章

性能数据

性能工具的工作方式是通过记录程序运行期间特定事件的数据，并将这些数据转换为对程序性能的测量（名为度量）。

本章描述了通过性能工具收集的数据，如何处理和显示这些数据，以及这些数据如何用于性能分析。因为收集性能数据的工具有很多，所以术语“收集器”用于表示这些工具中的任何一种。同样地，因为分析性能数据的工具也有很多，所以术语分析工具用于表示这些工具。

本章涵盖了以下主题。

- 使用联机帮助
- 收集器收集何种数据收集器收集何种数据
- 度量如何分配到程序结构

关于收集和存储性能数据的信息，请参阅第 3 章。

关于用 `er_print` 分析性能数据的信息，请参阅第 4 章。

使用联机帮助

关于分析性能数据的更多信息，请参阅 IDE 中的联机帮助或性能分析器 GUI 的联机帮助。

收集器收集何种数据

收集器收集三种不同类型的数据：分析数据、跟踪数据和全局数据。

- 通过在固定的间隔中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得的时间间隔，也可以是特定类型硬件事件的数目。间隔时间达到后会将一个信号传递到系统，数据将在下一个间隔被记录。
- 跟踪数据通过干预不同系统函数上的包装器函数来收集，因此可以截断对系统函数的调用并记录关于调用的数据。
- 全局数据的收集方式是通过调用不同的系统例程来获得信息。全局数据包称为样本。

分析数据和跟踪数据都包含特定事件的信息，这两种类型的数据都会转换成性能度量。全局数据不转换为度量，它的用途是提供将程序执行划分为很多时间段的标记。全局数据给出了时间段程序执行的概述。

在每个分析事件或跟踪事件收集的数据包中包括了以下信息：

- 标识数据的标题
- 高分辨率的时间标记
- 线程 ID
- 轻量进程 (LWP) ID
- 处理器 ID，操作系统中可用时
- 调用栈的副本

关于线程和轻量进程的更多信息，请参阅第 5 章。

除了公共数据外，每个特定事件的数据包还包含了特定数据类型的信息。收集器可以记录的五种数据类型是：

- 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

在以下五个子节中将描述产生度量的这五种数据类型，及如何使用这些数据类型。

时钟数据

在基于时钟的分析中，每个 LWP 的状态在定期的时间间隔存储。这种时间间隔称为分析间隔。这些信息被存储为整型数组：数组的每个元素用于内核维护的十个微态计算中的一个状态。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分辨率。缺省的分析间隔近似为 10 毫秒。收集器提供的高分辨率分析间隔近似为 1 毫秒，低分辨率分析间隔近似为 100 毫秒。此外，如果 OS 允许的话，可以使用任意的间隔。不带参数运行 `collect` 将打印运行该命令的系统可以允许的范围和分辨率。

在下表中定义了从基于时钟的数据计算的度量。

表 2-1 定时度量

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。
墙时间	在 LWP 1 中所用的 LWP 时间，即 “墙时钟时间”
全部 LWP 时间	全部 LWP 时间之和。
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。
其它等待时间	等待内核页所用的 LWP 时间，或休眠 \ 停止所用的时间。

对于多线程的实验，除了墙时钟时间之外的时间是所有 LWP 的总和。所定义的墙时间对于多程序多数据 (MPMD) 程序是无意义的。

定时度量用多种度量类型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 时间说明了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 时间说明了程序在对系统例程的调用中消耗了大量时间。
- 高等待 CPU 时间说明了准备运行的线程数比可用的 CPU 多，或其它进程正在使用 CPU。
- 高用户锁定时间说明线程无法获得请求的锁定。
- 高文本缺页时间意味着链接程序生成的代码在内存中组织，从而调用或分支导致装入新页。创建和使用映射文件（请参阅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了对数据的访问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该问题。

硬件计数器溢出分析数据

硬件计数器可跟踪诸如缓存丢失、缓存延迟循环、浮点运算、分支错误预测、CPU 循环和执行指令等事件。在硬件计数器溢出分析中，在运行 LWP 的 CPU 的指定硬件计数器溢出时收集器记录分析包。计数器被重置并继续计数。分析包中包括了溢出值和计数器类型。

UltraSPARC® III 处理器系列和 IA 处理器系列有两个可用于对事件计数的寄存器。收集器可从任一个也可从两个寄存器收集数据。对于每个寄存器，收集器允许选择计数器的类型来监视溢出，并设置计数器的溢出值。某些硬件计数器可以使用任一寄存器，而其它的计数器仅可以使用特定的寄存器。因此，不是所有的硬件计数器组合都可以在某一个实验中选择。

硬件计数器溢出分析数据由性能分析器转换成计数度量。对于在循环中计数的计数器，报告的度量被转换为时间；对于不在循环中计数的计数器，报告的度量是事件计数。在具有多个 CPU 的机器上，用于转换度量的时钟频率是单个 CPU 时钟频率的调和平均值。因为每种处理器自身都有一套硬件计数器，并且硬件计数器的数量庞大，所以此处没有列出硬件计数器度量。下一子节讲述如何找出可用的硬件计数器。

硬件计数器的一个用处就是诊断随着信息流入和流出 CPU 而产生的问题。例如，缓存丢失的高计数表明，重新组织程序的结构来改进数据或文本的位置或提高缓存的重用可以改善程序性能。

某些硬件计数器提供了类似或有关的信息。例如，分支错误预测和指令缓存丢失通常是相关的，因为分支错误预测使得错误指令被装入指令缓存中，而这些错误指令必须替换为正确指令。这种替换会引起指令缓存丢失或指令旁路转换缓冲 (ITLB) 丢失。

硬件计数器溢出通常在引起事件和相应事件计数器溢出的指令后传递一个或多个指令。这是指“刹车”，它会使计数器溢出分析难以解释。如果临时指令的精确标识缺少硬件支持，则可以对候选的临时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，硬件计数器分析包另外包括了适用于硬件计数器事件的候选内存引用指令的 PC（程序计数器）和 EA（有效地址）。（分析期间后续进程需要验证候选事件 PC 和 EA）。关于内存引用事件的附加信息有助于各种面向数据的分析。

硬件计数器列表

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正使用的处理器。性能工具为大量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中输入不带参数的 collect，您可以从收集器获得该系统上可用的硬件计数器列表。

以下示例显示了计数器列表中带别名的计数器条目。该例中的每一行输出都按打印格式显示。实际输出没有换行。

```
CPU Cycles (cycles = Cycle_cnt/*) 9999991 hi=1000003, lo=100000007
(CPU-cycles)

Instructions Executed (insts = Instr_cnt/*) 9999991 hi=1000003,
lo=100000007 (Events)

D$ Read Misses (dcrm = DC_rd_miss/1) 100003 hi=10007, lo=1000003
load (Events)
```

第一行的第一个字段 CPU Cycles 是度量名称。第二个字段 cycles 给出了可用于 -h 计数器中的别名和 er_print 的参数。第三个字段 Cycle_cnt/* 给出了 cputrack (1) 使用的内部名称和计数器可以使用的寄存器数目。寄存器数目不是 0 就是 1 或 *。本示例中，* 表示该计数器可以在任意寄存器上使用。下一个字段是溢出间隔，接下来的字段是高分辨率溢出间隔，而最后一个字段是低分辨率溢出间隔。(CPU-cycles) 表示计数器计数时以 CPU 循环为单位，并可转换为时间。

第二行以 (Events) 终止，表示对事件进行计数且不能转换为时间。

如示例的第三行所示，每行还可以在低分辨率值和计数器单位之间具有另外的字段，表示了计数器是否可以同时用装入和存储，或其中任意一个来触发，或表示计数器不是有关的程序。

在 UltraSPARC 和 IA 硬件上都可以使用的带别名的计数器在表 2-2 中列出。在 UltraSPARC 硬件上还有其它可用的别名。

表 2-2 可用于 SPARC 和 IA 硬件的具有别名的硬件计数器

带别名的计数器名称	度量名称	描述
cycles	CPU 循环	CPU 循环，在任意一个寄存器上计数
insts	执行的指令	执行的指令，在任意一个寄存器上计数

该例中的每行输出按打印格式化。实际输出不断行。不带别名的计数器的输出行如下所示：

```
Cycle_cnt Events (reg.0) 1000003 hi=100003, lo=9999991
(CPU-cycles)

Instr_cnt Events (reg.0) 1000003 hi=100003, lo=9999991 (Events)

DC_rd Events (reg.0) 1000003 hi=100003, lo=9999991 load (Events)
```

该行中的第一个字段 Cycle_cnt 给出了 cputrack(1) 使用的内部名称和计数器可以使用的寄存器数目。字符串 Cycle_cnt Events 是该计数器的度量名称。该行剩余部分的格式与带别名的计数器格式相同。

在循环中计数的计数器无论是否具有别名在行尾都用 (CPU-cycles) 表示，报告的度量缺省转换为包括和排除时间，但也可以像事件计数一样显示。事件中计数的计数器在行尾用 (Events) 表示，报告的度量是包括和排除事件计数。

对于与内存操作有关的硬件计数器，用后跟计数器名称的 load、store 或 load-store 表示，其中计数器的名称可能有前缀符号 +，目的是请求数据集合尝试查找在溢出的计数器上引起事件的精确指令和有效地址。

如果计数器与程序无关，将它用于分析会生成警告，且分析不会记录调用栈，而是会显示在人工函数 collector_not_program_related 中消耗的时间。线程和 LWP ID 所用的时间将被记录，不过是无意义的。字符串 not-program-related 将显示在对与运行程序无关的事件进行计数的任何计数器名称之后。使用这种计数器报告函数 collector_not_program_related 中的度量会在收集之前给出一个警告。

在计数器列表中，第一个显示的是带别名的计数器，然后是寄存器 0 上所有可用的计数器，最后是寄存器 1 上所有可用的计数器。带别名的计数器显示两次，一次带有别名，一次不带别名。在不带别名的列表中，这些计数器可以具有不同的溢出值。带别名的循环计数器的缺省溢出值已用于生成与时钟数据近似相同的数据收集率。其它计数器对应用程序的实际行为更敏感。

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使应用程序的执行延迟，例如，一个线程要访问已被其它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，并通过跟踪对线程库 `libthread.so` 中的函数调用来收集。收集和记录这些事件的过程称为同步等待跟踪。等待锁定消耗的时间称为等待时间。

只有等待时间超过阈值（单位为微秒）才记录事件。0 阈值表示所有的同步延迟事件被跟踪，没有等待时间。缺省阈值通过运行校准测试决定，测试中对线程库的调用不会出现任何同步延迟。阈值是这些调用的用任意因子（当前为 6）相乘的平均时间。该过程防止了事件的记录，因此等待时间仅取决于调用本身，而与真实的延迟无关。因此，数据量会大大减少，而同步事件的计数会被明显低估。

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，机器和 Java 调用栈都会被收集；但对于 JVM 中使用的内部锁定，不收集任何同步跟踪数据。在机器表示中，线程同步转换到对 `_lwp_mutex_lock` 的调用，且不显示同步数据，因为这些调用没有被跟踪。

同步等待跟踪数据被转换成以下度量：

表 2-3 同步等待跟踪度量

度量	定义
同步延迟事件。	对等待时间超过了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。

从该信息您可以决定对同步例程执行调用时，函数或负载对象是经常阻塞，还是经常会出现长的等待时间。高同步等待时间表示线程之间的争用。要减少争用，可以重新设计算法，尤其是重新组织锁定的结构，这样就可以仅包含每个需要锁定的线程的数据。

堆跟踪（内存分配）数据

对没有正确管理的内存分配和释放函数的调用是低效数据使用的来源，且会导致较差的程序性能。在堆跟踪中，收集器通过干预 C 标准库内存分配函数 `malloc`、`realloc`、`valloc` 和 `memalign` 以及释放函数 `free` 来跟踪内存分配和释放请求。因为 Fortran 函数 `allocate` 和 `deallocate` 调用 C 标准库函数，所以这些例程也被间接跟踪。

对于 Java 程序，堆跟踪数据记录所有对象分配事件（通过用户代码生成）和对象释放事件（通过垃圾收集器生成）。另外，任何 `malloc`、`free` 等的使用还生成了记录的事件。这些事件可能来源于本机代码或 JVM 本身。在机器表示中，内存通常以很大的块通过 JVM 分配和释放。Java 代码的内存分配完全由 JVM 及其垃圾收集器处理。堆跟踪不会显示 JVM 分配，因为这些分配由映射内存处理而不是调用正常的堆例程。堆跟踪也不显示任何关于 Java 内存分配和垃圾收集的信息。

堆跟踪数据被转换成以下度量：

表 2-4　内存分配 （堆跟踪）度量

度量	定义
分配	对内存分配函数的调用数
分配的字节	分配在对内存分配函数的每次调用中的字节的总数。
泄漏	对内存分配函数 （该函数对释放函数没有相应的调用）的调用数。
泄漏的字节	已分配但未释放的字节数。

收集堆跟踪数据有助于标识程序中的内存泄漏，或定位发生内存低效分配的位置。

通常使用内存泄漏的另外一个定义，例如在调试工具 dbx 中。这种替换的定义表示内存泄漏是内存的动态分配块，在程序数据空间的任何位置没有指针指向该块。这里所用的泄漏定义既包括了这种替换的定义，还包括了存在指针的内存。

MPI 跟踪数据

收集器可以收集对消息传递接口 (MPI) 库调用的数据。收集数据的函数在下面列出。

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 跟踪数据被转换成以下度量：

表 2-5　　MPI 跟踪度量

度量	定义
MPI 接收	接收数据的 MPI 函数中接收操作的数量
已接收 MPI 字节	MPI 函数中接收的字节数
MPI 发送	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 函数的调用所用的时间
其它 MPI 调用	对其它 MPI 函数的调用数

接收或发送时记录的字节数是调用中给定的缓冲大小。该缓冲大小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，假设直接处理器间通信和数据传送或数据的重新传输未优化，那么发送或接收的字节数是最大值。

被跟踪的 MPI 库的函数在表 2-6 中列出，分类为 MPI 发送函数、MPI 接收函数、MPI 发送和接收函数以及其它 MPI 函数。

表 2-6　　MPI 函数的类别有发送、接收、发送和接收以及其它

种类	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
其它 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 程序中因 MPI 调用而产生性能问题的位置。可能发生的性能问题有负载平衡、同步延迟和通信瓶颈。

全局（样本）数据

全局数据由收集器按名为样本包的包来记录。每个包中包含了一个标题、时间标记、诸如缺页和 I/O 数据内核的执行统计、上下文切换以及各种页面驻留（工作集和分页）的统计。记录在样本包中的数据对程序来说是全局的，且不转换为性能度量。记录样本包的过程称为抽样。

在以下情况中会记录样本包：

- 如果断点停止的选项被设置，而程序在 [调试] 窗口或 dbx 中因任何原因（如在断点处）停止时
- 在抽样间隔结束时且如果已选择了周期性抽样时。抽样间隔用一个整数以秒为单位指定。缺省值为 1 秒
- 选择 [调试]→ [性能工具箱] → [新样本]，或单击 [调试] 窗口的 [新样本] 按钮，或使用 dbx collector sample record 命令时
- 在对 collector_sample 调用且如果已将对该例程的调用放到代码中（请参阅第 43 页的“数据收集的程序控制”）时
- 指定信号已传递且如果已将 -l 选项和 collect 命令一起使用（请参阅 collect(1) 手册页）时
- 开始和终止收集时
- 后续进程创建前后

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。通过选择一组样本可以过滤特定事件的数据，以便您只看到特定时间周期的信息。此外也可以查看每个样本的全局数据。

性能工具消除了不同种类样本点之间的区别。要利用样本点进行分析，您应该只选择一种要记录的点。具体来讲，如果要记录与程序结构或执行序列有关的样本点，则您应该关闭周期性抽样并当 dbx 停止进程，或信号被传递到正使用 collect 命令记录数据的进程，或对收集器 API 函数调用时使用记录的样本。

度量如何分配到程序结构

度量使用与特定事件的数据一起记录的调用栈分配到程序指令。如果该信息可用，则每条指令被映射到一行源代码，而分配到该指令的度量也被分配到该行源代码。有关该过程的详细说明请参阅第 5 章。

除了源代码和指令，度量还分配到更高级别的对象：函数和负载对象。调用栈包含了关于函数调用序列的信息，目的是找到执行分析时记录的指令地址。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：排除、包括和属性

[性能分析器] 计算三种类型的函数级度量：排除度量、包括度量和属性度量。

- 函数的排除度量从函数本身内部发生的事件计算。这种度量排除了来自对其它函数的调用。
- 包括度量从函数本身和其调用的函数内部发生的事件计算。这种度量包括了来自对其它函数的调用。
- 属性度量说明了来自对（或从）另外一个函数的调用的包括度量数目：这种度量归属对另外一个函数的度量。

对于特定调用栈底部的函数（叶函数），排除度量和包括度量是相同的，因为这种函数没有对其它函数进行调用。

对于负载对象来说也要计算排除和包括度量。负载对象的排除度量通过累加负载对象中所有函数上函数级别的度量来计算。负载对象的包括度量与函数包括度量的计算方法相同。

函数的排除和包括度量给出了关于所有通过函数记录的路径信息。属性度量给出了关于通过函数记录的特定路径的信息。这些度量显示了来自特定函数的调用的度量数目。包含在调用中的两个函数描述为*调用方*和*被调用方*。对于调用树中的每个函数：

- 函数调用方的属性度量说明了取决于每个调用方的调用的函数包括度量数目。调用方的属性度量累计到函数的包括度量。
- 函数被调用方的属性度量说明了来自每个被调用方的调用的函数包括度量数目。它们的总和加上函数的排除度量等于函数的包括度量。

对调用方或被调用方的属性和包括度量进行比较，可以得到更多信息，如下所示：

- 调用方的属性度量和包括度量之间差额说明了来自对其它函数的调用和来自调用方本身的度量数目。
- 被调用方的属性度量和包括度量之间的差额说明了来自对其它函数调用的被调用方包括度量数目。

要定位能改善程序性能的位置，请执行以下操作：

- 使用排除度量定位具有高度量值的函数。
- 使用包括度量决定程序中哪个调用序列负责高度量值。
- 使用属性度量跟踪对该函数或负责高度量值的函数的特定调用序列。

解释属性度量：示例

排除、包括和属性度量在包含调用树碎片的图 2-1 中描述。焦点是中心的函数，函数 C。可能有对其它没有显示在该图中的函数的调用。

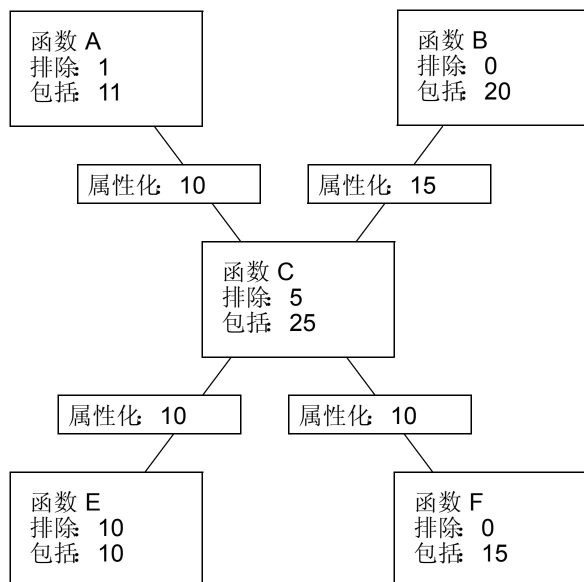


图 2-1 描述排除、包括和属性度量的调用树

函数 C 调用了两个函数，函数 E 和函数 F，并分别将 10 个单位的包括度量归属到函数 E 和函数 F。这些是被调用方的属性度量。它们的总和 (10+10) 加上函数 C 的排除度量 (5) 等于函数 C 的包括度量 (25)。

函数 E 的被调用方属性度量和被调用方包括度量是相同的，而函数 F 的这两种度量是不同的。这意味着函数 E 仅由函数 C 调用，而函数 F 由其它的一个或多个函数调用。函数 E 的排除度量和包括度量是相同的，而函数 F 的这两种度量是不同的。这意味着函数 F 可以调用其它的函数，而函数 E 不能。

函数 C 由以下两个函数调用：函数 A 和函数 B，并且将 10 个单位的包括度量归属到函数 A，将 15 个单位的包括度量归属到函数 B。这些是调用方的属性度量。它们的总和 (10+15) 等于函数 C 的包括度量。

调用方属性度量等于函数 A 包括度量和排除度量之间的差额，不等于函数 B 包括度量和排除度量之间的差额。这意味着函数 A 仅调用函数 C，而函数 B 除了调用函数 C 外还调用其它函数。（实际上，函数 A 也可以调用其它函数，只是时间很短在实验中不显示。）

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。性能分析器显示函数的整体度量，而不是函数的每个调用：因此，对一系列递归调用的度量必须压缩成单一度量。这不影响从调用栈（页函数）底部的函数计算的排除度量，但会影响包括和属性度量。

包括度量是通过将页函数的排除度量增加到调用栈中函数的包括度量来计算。为了确保在递归调用栈中不重复计算度量，叶函数的排除度量仅增加到每个唯一函数的包括度量。

属性度量从包括度量计算。在最简单的递归中，递归函数具有两个调用方：本身的函数和另一个函数（初始化函数）。如果在最后的调用中完成了所有处理，则递归函数的包括度量将被归属到本身，而不是初始化函数。这是因为递归函数所有更高调用的递归度量被视为零，避免了重复计算度量。不过，初始化函数做为被调用方（取决于递归调用的递归度量部分）正确归属到递归函数。

第 3 章

收集性能数据

数据收集是性能分析的第一阶段。本章描述了数据收集所需的内容，数据存储的位置，如何收集数据以及如何管理数据收集。关于数据本身的更多信息，请参阅第 2 章。

本章涵盖了以下主题。

- 编译和链接程序
- 为数据收集和分析准备程序
- 数据收集的局限
- 数据存储的位置
- 估计存储需求
- 收集数据
- 使用 `collect` 命令收集数据
- 使用 dbx `collector` 子命令收集数据
- 收集运行进程中的数据
- 从 MPI 程序收集数据
- 与 ppgsz 一起使用 `collect`

编译和链接程序

您可以为使用几乎任何选项编译的程序收集和分析数据，但部分选项会影响在性能分析器中收集或查看的内容。以下子节中描述了在编译和链接程序时应考虑的问题。

源码信息

要查看带注释的 [源码] 和 [反汇编] 中的源码和 [行] 分析中的源码行，就必须使用 `-g` 编译器选项（`-g0` 用于 C++ 启用前端内联）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 STABS 或 DWARF2，由 `-xdebugformat=(stabs|dwarf)` 指定。

要准备具有允许数据空间硬件计数器分析的调试信息的编译对象 （当前只为 SPARC® 的 C 编译器指定），就要编译指定 -xhwcprof -xdebugformat=dwarf 和任何级别的优化。(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，还要增加 -g 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对象文件调试符号的副本。如果 STABS 格式的调试符号与将 STABS 符号保留在各种对象文件及可执行文件中的 -xs 选项相链接，那么上述情况同样适用。这在您需要移动或删除对象文件时尤其重要。有了可执行文件和库自身的所有调试符号，就可以更容易地将试验和与程序相关的文件移至新位置。

静态链接

编译程序时，您不得禁止 -dn 和 -Bstatic 编译器选项所执行的动态链接。如果试图为完全静态链接的程序收集数据，那么收集器会打印出一条错误消息并且不收集数据。原因是在您运行收集器时该收集器库在其它库之间动态装入。

您不应该静态链接任何系统库。如果您执行了静态链接，就可能无法收集任何类型的跟踪数据。您也不应链接到收集器库 libcollector.so。

优化

如果使用某一级别的优化来编译程序，编译器就可以重新安排执行顺序，这样它就无需严格执行程序中行的顺序。性能分析器可以分析在优化后代码中收集的实验，但它在反汇编级别所显示的数据通常很难与初始源码行相关联。此外，如果编译器执行尾调用优化，则出现的调用序列可能与预想不同。优化可导致解除失败。更多信息请参阅第 105 页的 “尾调用优化”。

编译 Java 程序

用 javac 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，您无需为数据收集和分析作任何特殊准备。如果程序执行下列操作之一，您就应阅读一个或多个子章节：

- 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。

另外，如果您要控制程序中的数据收集，还应阅读相关子节。

使用动态分配的内存

很多程序依赖于动态分配的内存，使用如下特性：

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

必须确保程序不依赖于动态分配内存的初始内容，除非内存分配方法明确地说明设置初始值：例如，与 malloc(3C) 手册页中 calloc 和 malloc 的说明相比较。

在某些情况下，使用动态分配内存的程序可以独自正常运行，但是启用性能数据收集之后就会失败。失败症状可能包括不可预料的浮点行为、程序段失败或应用程序特定的错误消息。

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，则会发生这种行为，但应用程序与性能数据收集工具一起运行时，未初始化的内存设置为其它值。这些情况下，性能工具不会出现错误。依赖于动态分配内存内容的任何应用程序都具有潜在的错误：除非显式说明，否则操作系统随机提供动态分配内存中的任意内容。即使目前操作系统会始终将动态分配的内存设置为某一值，但是在操作系统的后续修订版中或将程序移植到以后不同的操作系统时，这些潜在的错误会引起意外的行为。

以下是一些工具，可以帮助您找到这些潜在的错误：

- f95 -xcheck=init_local

  有关更多信息请参阅 《Fortran 用户指南》或 f95(1) 手册页

- lint

  有关更多信息请参阅 《C 用户指南》或 lint(1) 手册页

- dbx 下的运行时检查

  有关更多信息请参阅 《使用 dbx 调试程序》手册或 dbx(1) 手册页。
- 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。下表描述了收集器插入对库函数调用的情况。

- 异步等待跟踪数据的收集。收集器插入线程库的函数 libthread.so。
- 堆跟踪数据的收集。收集器插入函数 malloc、realloc、memalign 和 free。这些函数的版本可在 C 标准库 libc.so，以及其它库，如 libmalloc.so 和 libmtmalloc.so 中找到。
- MPI 跟踪数据的收集。收集器插入 MPI 库的函数 libmpi.so。
- 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定时器。
- 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该计数器。执行从程序到库中函数的调用，其返回值是 -1。
- 启用后续进程中的数据收集。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) 及其变种。对 vfork 的调用被对 fork1 的调用内部替换。这些插入只适用于 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：

- 通过任何包含被插入函数的库来静态链接程序。
- 将 dbx 附加到运行中的应用程序 （该程序没有预装收集器）。
- 动态装入其中一个库，并通过只在该库中搜索来解决符号。

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

收集器使用两个信号 (SIGPROF 和 SIGEMT) 来收集分析数据。收集器为每个信号安装一个截取并处理信号的信号处理程序，但是会将不使用的信号传递到其它任何信号处理程序。如果程序将其自身的信号处理程序安装在这些信号上，那么收集器会将它的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。尽管用户处理程序安装时警告写入实验，但这些信号不受收集器保护。用户的责任是确保收集器对指定信号的使用和相同信号应用程序的任何使用之间没有冲突。

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标志。如果程序的信号处理程序将该标志设置为允许系统调用的中断，则该标志设置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，则取消信号延迟到达。

如果在未预装收集器库和启用性能数据收集的情况下将 dbx 附加到进程，并且程序随后安装自身信号处理程序，那么收集器不再重新安装自身信号处理程序。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收集器库时不同。

使用 setuid

由于动态加载器实施的限制，所以难以使用 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。

数据收集的程序控制

如果想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。

```
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);
```

在第 44 页的 “Java 接口” 中描述了由 CollectorAPI 类提供给 Java 程序的相似功能性。

C 和 C++ 接口

有两种方法访问 C 和 C++ 接口。

- 第一种方法是包括 `collectorAPI.h` 并用 `-lcollectorAPI`（包含用于检查基础 `libcollector.so` API 函数存在的实函数）相链接。

  这种方法要求与 API 库相链接，并可在所有情况下使用。如果没有激活的实验，则 API 调用被忽略。

- 第二种方法是包括 `libcollector.h`（包含检查基础 `libcollector.so` API 函数存在的宏。）

  当这种方法用于主可执行文件时，以及数据收集在启动程序的同时启动时，该方法有效。当 `dbx` 用于绑定进程，或用于进程 `dlopen` 的共享库时，该方法通常无效。第二种方法适用于向下兼容。

注意 – 不要将任何语言的程序用 `-lcollector` 链接。如果链接，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，应用程序必须用 `-lcollectorAPI` 链接。（该库的替代名称 `-lfcollector` 适用于向下兼容。）除动态函数、线程暂停和恢复调用的特性之外， Fortran API 提供了与 C 和 C++ API 相同的特性。

要使用 Fortran 的 API 函数，请插入下列语句：

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

注意 – 不要将任何语言的程序用 `-lcollector` 链接。如果链接，则收集器可能会出现不可预知的行为。

Java 接口

使用以下语句输入 CollectorAPI 类并访问 Java API。注意，无论如何都必须使用指向 `<installation-directory>/lib/collector.jar` 的类路径来调用应用程序，其中 `<installation-directory>` 是 Sun 编译器和工具所安装的目录。

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

Java CollectorAPI 方法定义为：

```
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 具有与 Fortran API 相同的函数。

未收集数据时，C 包含文件 `libcollector.h` 包括不使用对实 API 函数调用的宏。在这种情况下，不动态装入函数。不过，由于在某些情况下这些宏不能很好的运行，所以使用这些宏会有风险。使用 `collectorAPI.h` 就较为安全，因为它不使用宏。而且，它直接引用到函数。

如果正在收集性能数据，则 Fortran API 子例程调用 C API 函数，否则返回。检查的开销很低，不会对程序性能产生较大影响。

如本章后部分所述，要收集性能数据就必须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。

如果要在多线程程序中使用 API 函数，就应该确保它们只被一个线程调用。在异常为 `collector_thread_pause()` 和 `collector_thread_resume()` 时，API 函数执行适用于进程而不是单独线程的操作。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，如果一个线程在其它线程到达程序同一点之前调用了 `collector_pause()` 或 `collector_terminate_expt()`，则所有线程的集合会被暂停或终止，而对于在 API 调用前执行代码的线程来讲，可能会丢失其中的数据。要控制单独线程级别的数据收集，就要使用 `collector_thread_pause()` 和 `collector_thread_resume()` 函数。有两种方法使用以上函数：让一个主线程执行对全部线程 （包括其自身）的调用；或让每个线程只执行对其自身的调用。其它任何用法都可能产生不可预知的结果。

C、 C++、 Fortran 和 Java API 函数

API 函数的具体描述如下。

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

  Fortran：`collector_sample(string)`

  Java：`CollectorAPI.sample(String)`

  记录样本包并用指定的名称或字符串标记该样本。性能分析器将该标签显示在 [事件] 标签中。Fortran 变量 `string` 为 `character` 类型。

样本点包含进程而不是单独线程的数据。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调用，则该函数可确保只写入一个样本。所记录的样本数目可能会低于线程调用的次数。

性能分析器不对所记录的不同机制的样本进行区分。如果只想查看 API 调用所记录的样本，就应在记录性能数据时关闭所有其它样本模式。

- **C、C++、Fortran**：`collector_pause()`

  Java：`CollectorAPI.pause()`

  终止将事件特定的数据写入实验。实验保持打开状态，继续写入全局数据。如果没用激活的实验或数据记录已被停止，则该调用被忽略。该函数终止写入所有特定于事件的数据，即使它是由 `collector_thread_resume()` 函数为特定线程启用的。

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

  Java：`CollectorAPI.resume()`

  在调用 `collector_pause()` 之后恢复将事件特定的数据写入实验。如果没用激活的实验或数据记录已激活，则该调用被忽略。

- **只用于 C 和 C++**：`collector_thread_pause(unsigned int t)`

  Java：`CollectorAPI.threadPause(Thread)`

  终止将变量列表中特定线程的事件特定数据写入实验。变量 `t` 为 POSIX 线程标识符。如果实验已终止、没有激活的实验，或对该线程的数据写入已结束，则该调用被忽略。即使全局启用了数据写入，该函数还会终止写入特定线程的数据。缺省情况下，对单独线程的数据记录处于开启状态。

- **只用于 C 和 C++**：`collector_thread_resume(unsigned int t)`

  Java：`CollectorAPI.threadResume(Thread)`

  恢复将变量列表中特定线程的事件特定数据写入实验。变量 `t` 为 POSIX 线程标识符。如果实验已终止、没有激活的实验，或对该线程的数据写入已打开，则该调用被忽略。只有在数据写入被全局启用，以及线程的数据写入被启用的情况下，才可以将数据写入实验中。

- **C、C++、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。如果没有激活的实验，则该调用被忽略。

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，并且您想在性能分析器中查看动态函数或模块的数据，就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。

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

您无需将这些 API 函数用于由 Java HotSpot™ 虚拟机编译的 Java™ 方法中，原因是它使用了一个不同的接口。 Java 接口命名了已编译到收集器的方法。您可以查看 Java 编译方法的函数数据和注释反汇编列表，但不包括注释源码列表。

API 函数的具体描述如下。

- `collector_func_load()`

将动态编译函数的有关信息传递到收集器，以用于在实验中进行记录。下表对参数列表进行了具体描述。

表 3-1 `collector_func_load()` 的参数列表

参数	定义
`name`	性能工具所用动态编译函数的名称。该名称不必是函数的真实名称。虽然该名称不应包含嵌入的空格或引号字符，但是它无需遵循正常的函数命名规则。
`alias`	用于描述函数的任意字符串。它可以为 `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 [汇总] 标签中。alias 可用于指示函数的内容或动态构造函数的原因。
`sourcename`	到构造函数时所在源文件的路径。它可以为 `NULL`。该源文件用于注释的源码列表。
`vaddr`	函数的装入地址。
`size`	以字节为单位的函数尺寸。
`lntsize`	对行编号表中条目数量的计数。如果未提供行编号信息，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。而第一个整数是偏移，第二个条目是行编号。在一个条目的偏移和下一条目给定的偏移之间的所有指令都归属于第一个条目给定的行编号。偏移必须按照数值的增序，而行编号可为任意顺序。在 `lntable` 为 `NULL` 的情况下，虽然反汇编列表可用，但是也没有可用的函数源码列表。

- `collector_func_unload()`

通知收集器位于 `vaddr` 的动态函数已被卸载。

数据收集的局限

本节描述了数据收集的局限，这些局限是由硬件、操作环境、运行程序的方法或收集器自身造成的。

对同时收集不同的数据类型来讲，没有任何局限：您可以收集任何具有其它数据类型的数据类型。

基于时钟的分析的局限

用于分析的分析间隔最小值和时钟分辨率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时钟分辨率的倍数。时钟分辨率的最小值和最大值可以通过输入不带参数的 `collect` 命令来查找。

系统时钟用于在 Solaris 7 和 Solaris 8 操作系统的版本中进行分析。除非您选择了启用高分辨率系统时钟，否则它的分辨率为 10 毫秒。如果您有 root 特权，就可以通过增加以下行到文件 `/etc/system` 并重新启动来执行该操作。

```
set hires_tick=1
```

在 Solaris 9 操作系统下和较新版本的 Solaris 8 操作系统中，没有必要启用用于高分辨率分析的高分辨率系统时钟。

时钟分析中的运行时失真和扩大

时钟分析记录 SIGPROF 信号传递到目标时的数据。这将导致处理该信号和解除调用栈的扩大。调用栈越深，则信号越频繁，扩大越显著。在有限的范围内，时钟分析将产生失真，这是由程序中这些执行最深栈部分的显著扩大导致。

收集跟踪数据的局限

只有在已预装收集器库 `libcollector.so` 的情况下，您才可以收集运行程序中任意种类的跟踪数据。更多信息请参阅第 66 页的 “收集运行进程中的数据”。

跟踪过程中的运行时失真和扩大

跟踪数据将与被跟踪事件的数量成比例地扩大运行。如果完成时钟分析，则跟踪事件引起的扩大将使时钟数据失真。

硬件计数器溢出分析的局限

对于硬件计数器溢出分析，存在多个局限：

- 您只能收集在具有硬件计数器的处理器和支持溢出分析的处理器上的硬件计数器溢出数据。在其它系统中，硬件计数器溢出分析被禁止。在 UltraSPARC® III 处理器系列之前的 UltraSPARC® 处理器不支持硬件计数器分析。
- 您无法收集早于 Solaris 8 发行版本操作系统版本下的硬件计数器溢出数据。
- 您可以为实验中至少两个硬件计数器记录数据。要为两个以上的硬件计数器或为使用相同寄存器的计数器记录数据，您必须运行多个独立的实验。
- 在 `cpustat(1)` 运行过程中，您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，且不让用户进程使用。如果在数据收集期间启动 `cpustat`，则硬件计数器溢出分析终止。
- 如果您正在执行硬件计数器分析，则无法同时使用自身代码形式的硬件计数器和 `libcpc(3)` API。如果调用不来自收集器，则收集器插入 `libcpc` 库函数并返回 `-1` 返回值。
- 如果您试图通过向进程附加 `dbx` 在使用硬件计数器库的运行程序上收集硬件计数器数据，则实验会被破坏。

注 – 要查看所有可用计数器的列表，请运行不带参数的 `collect`。

硬件计数器溢出分析中的运行时失真和扩大

硬件计数器分析记录 SIGEMT 传递到目标时的数据。这将导致处理该信号和解除调用栈的扩大。与时钟分析不同的是，对于某些硬件计数器来说，程序的不同部分可能会比其它部分更快速地生成事件，并显示出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被严重破坏。类似地，部分事件可能会在一个与其它线程不成比例的线程中生成。

后续进程中数据收集的局限

您可以根据以下局限在后续进程中收集数据：

- 如果您要在后跟收集器的所有后续进程中收集数据，就必须使用 collect 命令和 -F on 选项。
- 您可以为 fork（及其变异进程）和 exec（及其变异进程）调用自动收集数据。收集器不会跟在 system、popen 和 sh 调用之后。
- 如果要为单独的后续进程收集数据，就必须将 dbx 附加到该进程。更多信息请参阅附录第 66 页的 “收集运行进程中的数据”。
- 如果要为单独的或由 system、 popen、 sh 等创建的后续进程收集数据，就必须将一个独立的 dbx 附加到每个进程中并启用收集器。

Java 分析的局限

您可以根据下列局限在 Java 程序中收集数据：

- 您应该使用版本号不低于 1.4.2_02 的 Java™ 2 软件开发工具包。不要使用可能会崩溃的 1.4.2 或 1.4.2_01 版。应在以下四个环境变量之一中指定到 Java 虚拟机[1] 的路径：JDK_1_4_HOME、JDK_HOME, JAVA_PATH、PATH. 收集器会验证它在以上环境变量中找到的 java 版本为 ELF 可执行文件，如果不是，则打印错误信息，指出所用的环境变量和已尝试的全路径名称。
- 您必须使用 collect 命令来收集数据。不要使用 dbx collector 子命令或 IDE 的数据收集功能。
- 如果要使用 64 位 JVM™，则必须是在缺省情况下，或指定收集数据时的路径。不要使用 java -d64 来收集使用 64 位 JVM 的数据。如果使用，则不会收集到任何数据。

使用 1.4.2_02 之前的 JVM 版本会破坏数据，如下所示：

- **JVM 1.4.2_01**：数据收集期间该版本的 JVM 可能会崩溃。
- **JVM 1.4.2_02**：数据收集期间该版本的 JVM 可能会崩溃。
- **JVM 1.4.1**：正确记录和显示了 Java 表示，但是所有 JVM 内务处理被显示为 JVM 函数自身。在数据空间中执行 JVM 代码占用的部分时间由 JVM 所提供的代码区域的名称显示。由于某些 JVM 所创建的代码区没有被命名，所以 <Unknown> 函数中会显示大量时间。此外， JVM 1.4.1 中各种错误可能导致正在进行分析的程序崩溃。
- **JVM 1.4.0**：不支持 Java 表示，并且 <Unknown> 中显示了大量时间。 HotSpot 编译的函数显示为机器表示的名称。
- **JVM 1.4.0 之前的版本**：JVM 1.4.0 之前的版本不支持分析 Java 应用程序。

1. “Java 虚拟机” 和 “JVM” 术语表示 Java 平台的虚拟机。

用 Java 编程语言所编写应用程序的运行时性能失真和扩大

Java 分析使用的 JVMPI 接口可能会导致运行的失真和扩大。对于时钟和 hwc 分析，数据收集进程会对 JVM 进行各种调用，并处理单一处理程序中的分析事件。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。预计这种扩大不高于 10%。

此外，虽然缺省的垃圾收集器支持 JVMPI，但还存在其它不支持 JVMPI 的垃圾收集器。任何数据收集运行这种指定的垃圾收集器，都会出现致命的错误。

对于堆分析，数据收集进程使用描述内存分布和垃圾收集的 JVMPI 事件，可能会引起运行时的显著扩大。多数 Java 应用程序会生成许多上述事件，这将导致实验很大，以及处理数据时的可伸缩性问题。此外，如果需要这些事件，垃圾收集器就会禁止部分已内联的分配，将导致更多的 CPU 时间用于较长的分配路径。

对同步跟踪来讲，数据收集使用了其它 JVMPI 事件，这将导致应用程序中监视器争用数量成比例地扩大。

数据存储的位置

在应用程序的一次执行过程中所收集的数据称作实验。实验由存储在目录下的一系列文件组成。实验的名称即为目录的名称。

除了记录实验数据以外，收集器还为程序所使用的负载对象创建自己的归档文件。这些归档文件包含了负载对象中每个对象文件和函数的地址、大小和名称，以及负载对象的地址和最后一次修改的时间标记。

实验以缺省方式存储在当前目录中。如果该目录位于网络文件系统，则存储数据的时间比在本地文件系统中长，而且可能使性能数据失真。如有必要，您应该始终尝试在本地文件系统中记录实验。您可以在运行收集器时更改存储位置。

后续进程的实验存储在创建进程的实验内部。

实验名称

新实验的缺省名称为 `test.1.er`。后缀 `.er` 是强制的：如果命名不具有该后缀，则显示一条错误消息且不接受该名称。

如果您选择使用 *experiment*.*n*.er 格式的名称 （其中 *n* 为正整数），则收集器会将后续实验名称中的 *n* 自动增加一 - 例如， `mytest.1.er` 会后跟 `mytest.2.er`、`mytest.3.er` 等。如果该实验已存在，收集器还会增加 *n*，并持续到找到未使用实验名称时的 *n* 值。如果实验名称不含 *n* 且实验存在，则收集器会打印出一条错误消息。

实验可按组收集。组是在实验组文件中定义，并以缺省形式存储在当前目录下。实验组文件都是文本文件，具有特殊的标题行，并在下面的行显示实验名称。实验组文件的缺省名称为 test.erg。如果该名称不以 .erg 结尾，会显示出错且不接受该名称。一旦创建了实验组，那么您使用该组名运行的所有实验都会增加到这个组。

您可以通过创建文本文件来创建实验组文件，该文本文件的首行为

```
#analyzer experiment group
```

并将实验名称增加到下面的行。文件的名称必须以 .erg 结尾。

缺省实验名称与在 MPI 程序中收集的实验名称不同， MPI 程序为每个 MPI 进程都创建了一个实验。缺省实验名称为 test.*m*.er，其中 *m* 为进程的 MPI 级别。如果您指定了实验组 *group*.erg，则缺省实验名称为 *group*.*m*.er。如果指定了实验名称，则该名称覆盖缺省值。更多信息请参阅第 69 页的 "从 MPI 程序收集数据"。

后续进程的实验是使用其以下沿袭命名的。要组成后续进程的实验名称，可将下划线、代码字母和数字增加到它所创建的实验名称中。代码字母 f 表示 fork， x 表示 exec。数字是 fork 或 exec 的索引 （无论是否成功）。例如，如果创建进程的实验名称为 test.1.er，则由对 fork 的第三次调用所创建的子进程实验为 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

如果您要将实验移动到其它计算机并对其进行分析，就应该了解分析对记录实验所在操作环境的依存。

归档文件包含计算函数级度量所必需的所有信息，并显示出时间线。但是，如果要查看注释的源代码或注释的反汇编代码，就必须有权使用与记录实验时所用负载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，并在找到正确基名的文件时停止。

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

为确保您在程序中查看到正确的注释源代码和注释反汇编代码，可以在移动或复制实验之前将源代码、对象文件和可执行文件复制到该实验。如果您不想复制对象文件，可以将程序用 -xs 链接，以确保源代码行和文件位置上的信息插入可执行文件。您可以通过使用 collect 命令 -A 选项或 dbx collector archive 命令将负载对象自动复制到实验中。

估计存储需求

本节给出了一些关于估计记录实验所需磁盘空间容量的指导。实验的大小直接取决于数据包的尺寸、记录数据的速度、程序使用的 LWP 的数量和程序的执行时间。

数据包包含事件特定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总计约为 50 到 100 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 SPARC 体系结构中为 8 个字节）。数据包是为实验中每个 LWP 而记录。注意，对于 Java 程序，将会有两个感兴趣的调用栈：Java 调用栈和机器调用栈，因此将产生更多正写入磁盘的数据。

记录分析数据包的速度由时钟数据的分析间隔和硬件计数器数据的溢出值控制。但是，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。虽然这些参数的较小值会获得较好的统计数据，但也增加了开销。由于获取较好统计数据和降低开销之间的折衷，所以精心选择了分析间隔和溢出值的缺省值。较小值也意味着更多数据。

对于一个具有 10 毫秒分析间隔和小调用栈的基于时钟分析的实验，包的大小为 100 字节，每个 LWP 记录数据的速度为 10 千字节 / 秒。对于为 CPU 循环收集数据的硬件计数器溢出分析实验，和在溢出值为 1000000 而包的大小为 100 字节的 750MHz 处理器上执行的指令来讲，每个 LWP 记录数据的数度为 150 千字节 / 秒。调用栈深度为数百个调用的应用程序可以毫不费力地以十倍的速度记录数据。

在估计实验大小时，还应考虑归档文件所占用的磁盘空间，它通常是总磁盘空间需求的一小部分 （请参阅前一节）。如果您无法确定所需空间的大小，请在短时间内运行实验。从该测试中，您可以得到与数据收集时间无关的归档文件的大小，测量分析文件的大小以获得全长度实验的估计大小。

除分配磁盘空间之外，收集器还在内存中分配缓冲以便在将分析数据写入磁盘之前进行存储。当前还没有指定这些缓冲区大小的方法。如果收集器内存不足，则应尽量减少所收集数据的数量。

如果预计的存储实验所需空间大于可用空间，则可以考虑收集部分运行中的数据，而不是全部。要执行以上操作，可以使用 `collect` 命令、 `dbx collector` 子命令，或将程序中的调用插入收集器 API。您还可以限制分析和由 `collect` 命令或 `dbx collector` 子命令所收集跟踪数据的总量。

注 – 性能分析器无法读取大于 2 GB 的性能数据。

收集数据

您可以通过多种方法在独立性能分析器或分析器模块中收集性能数据：

- 在命令行使用 `collect` 命令（请参阅第 54 页的 “使用 `collect` 命令收集数据” 和 `collect`(1) 手册页）。收集命令行工具比 IDE [调试器] 中 `dbx` 或 [收集器] 对话框具有较小的数据收集开销，因此该方法比其它方法要好。
- 使用性能分析器的 [性能工具收集] 对话框（请参阅性能分析器联机帮助中 “通过性能工具收集器收集性能数据” 部分）
- 使用调试器的 [收集器] 对话框（请参阅性能分析器联机帮助中 “通过调试器收集性能数据” 部分）
- 使用 `dbx` 命令行中的 `collector` 命令（请参阅第 61 页的 “使用 `dbx collector` 子命令收集数据” 和 IDE 的 [调试] 联机帮助中 “收集器命令” 部分）

下列数据收集功能只适用于 [性能工具收集] 对话框和 `collect` 命令：

- 收集 Java™ 程序中的数据。如果使用 IDE 调试器中的 [收集器对话框] 或 `dbx` 中的 `collector` 命令来收集 Java 程序中的数据，则所收集的信息属于 [Java 虚拟机]，而不是 Java 程序。
- 自动收集后续进程中的数据。

使用 `collect` 命令收集数据

要使用 `collect` 命令从命令行运行收集器，请输入下列内容。

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

其中，*collect-options* 是 `collect` 命令选项，*program* 是要为其收集数据的程序的名称，*program-arguments* 是它的参数。

如果未给定命令参数，则缺省值开始分析间隔为 10 毫秒的基于时钟的分析。

要获取任意可用于分析的硬件计数器的选项列表和名称列表，请输入不带参数的 `collect` 命令。

```
% collect
```

对硬件计数器列表的描述，请参阅第 28 页的 “硬件计数器溢出分析数据”。另见第 49 页的 “硬件计数器溢出分析的局限”。

数据收集选项

这些选项控制收集数据的类型。关于数据类型的描述，请参阅第 26 页的 “收集器收集何种数据”。

如果给定了数据收集选项，则缺省值为 -p on，该缺省值可以启用缺省分析间隔为 10 毫秒的基于时钟的分析。该缺省值是由 -h 选项关闭，而不是由任何其它数据收集选项。

如果基于时钟的分析被明确禁止，也未启用任何类型的跟踪或硬件计数器溢出分析，则 collect 命令打印出一条警告消息，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 – 结束基于时钟的分析。
- on – 打开缺省分析间隔为 10 毫秒的基于时钟的分析。
- lo[w] – 打开低分辨率分析间隔为 100 毫秒的基于时钟的分析。
- hi[gh] – 打开高分辨率分析间隔为 1 毫秒的基于时钟的分析。在 Solaris 7 操作系统下和 Solaris 8 操作系统的早期版本中，必须显式启用高分辨率的分析。关于启用高分辨率分析的详细信息，请参阅第 48 页的 “基于时钟的分析的局限”
- *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。您可以将 *value* 指定为整数或浮点数。可在数值后加后缀 m 选择毫秒单位，或后加 u 选择微秒单位。这个值应该是时钟分辨率的倍数。如果数值较大且不是分辨率的倍数，则向下舍入。如果较小，就会打印出一条警告消息，并将其设置为时钟分辨率。

收集基于时钟的分析数据为 collect 命令的缺省操作。

-h *counter*[,*value*[,*counter2*[,*value2*]]]

收集硬件计数器溢出分析数据。计数器名称 *counter* 和 *counter2* 可为下列名称之一：

- 计数器别名
- 类似于 cputrack(1) 所用的内部名称。如果计数器可以使用任一事件寄存器，则可向内部名称附加 /0 或 /1 来指定将要使用的事件寄存器。

如果指定了两个计数器，则它们必须使用不同的寄存器。如果未使用不同的寄存器，则 collect 命令打印出一条错误消息，然后退出。部分计数器可在任何寄存器上计数。

要获取可用计数器列表，请在终端窗口中输入没有参数的 collect。在第 29 页的 “硬件计数器列表” 部分给出了对计数器列表的描述。

如果硬件计数器计算的事件与内存访问有关，则可在计数器名称前放置 + 号，开始搜索引起计数器溢出的指令的真实 PC。如果搜索成功，则引用的 PC 和有效地址存储在事件数据包中。

溢出值是在硬件计数器溢出和记录溢出事件的位置所计算的事件数目。该溢出值可通过 *value* 和 *value2* 指定，这两个变量可设置为下列值之一：

- `hi[gh]` – 使用所选计数器高分辨率值。缩写 `h` 还支持与先前软件发行版本的兼容。
- `lo[w]` – 使用所选计数器低分辨率值。
- *number* – 溢出值。必须为正整数。
- `on`，或空字符串 – 使用缺省溢出值。

缺省值是为每个计数器预定义的和出现在计数器列表的正常阈值。另见第 49 页的 “硬件计数器溢出分析的局限”。

如果使用未明确指定 `-p` 选项的 `-h` 选项，则关闭基于时钟的分析。要收集硬件计数器数据和基于时钟的数据，就必须指定 `-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` 等价。）
- `off` – 禁止同步等待跟踪。
- `on` – 用将在运行时通过校准设置阈值的缺省阈值启用同步等待跟踪。（与 `calibrate` 等价。）
- *value* – 将阈值设置为 *value*，给定值为以毫秒为单位的正整数。

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` – 打开跟踪堆分配和堆释放的请求。
- `off` – 关闭堆跟踪。

堆跟踪缺省值为关闭。

-m *option*

收集 MPI 跟踪数据。 *option* 的允许值包括：

- on – 打开跟踪 MPI 调用。
- off – 关闭跟踪 MPI 调用。

MPI 跟踪缺省值为关闭。

关于调用被跟踪的 MPI 函数以及从跟踪数据中所计算度量的更多信息，请参阅第 32 页的 "MPI 跟踪数据"

-S *option*

周期性地记录样本包。 *option* 的允许值包括：

- off – 关闭周期抽样。
- on – 打开缺省抽样间隔为 1 秒的周期抽样。
- *value* – 打开周期抽样并将抽样间隔设置为 *value*。间隔值必须为正并且以秒为单位。

缺省情况下，启用间隔为 1 秒的周期抽样。

实验控制选项

-F *option*

控制后续进程是否应该记录其数据。 *option* 的允许值包括：

- on – 为所有后跟收集器的后续进程记录实验。
- off – 不记录后续进程中的实验。

收集器跟随由函数 fork(2)、 fork1(2)、 fork(3F)、 vfork(2)、 exec（2）及其变种的调用所创建的进程。对 vfork 的调用被对 fork1 的调用内部替换。收集器不跟随由对 system(3C)、 system(3F)、 sh(3F) 和 popen(3C) 和类似函数调用创建的进程，和其它有关的后续进程。

如果指定 -F on 参数，则收集器为创建实验内的每个后续进程打开新的实验。这些新的实验按照以下方式命名。

要形成后续进程的实验名称，将下划线、字符和数字增加到实验后缀。字母 "f" 指示 fork 而字母 "x" 指示 exec。数字是 fork 或 exec 的索引（无论是否成功）。例如，如果初始进程的实验名称是 test.1.er，则其第三个派生创建的子进程的实验是 test.1.er/_f3.er。如果该子进程执行了新映像，则相应实验名称为 test.1.er/_f3_x1.er。

读取创建实验时分析器和 er_print 自动读取后续进程的实验，但不选择后续进程的实验用于数据显示。

要从命令行选择显示的数据，请显式对 er_print 或 analyzer 指定包括创建实验名称的路径名称和该目录内的后续实验。例如，要查看 test.1.er 实验的第三个派生数据，则需要指定以下内容：

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

```
analyzertest.1.er/_f3.er
```

此外，您可以使用感兴趣的后续实验显式名称来准备实验组文件。

要为分析器中的显示选择数据，装入 test.1.er 并从 [视图] 菜单选择 [筛选器] 数据。您将看到仅创建实验的实验列表 (test.1.er) 被选中。取消选择该实验和 check _f3.er。

-j *option*

启用非标准 Java 安装的 Java 分析，或选择是否收集由 Java HotSpot 虚拟机编译的方法上的数据。 *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 栈。
- off – 不尝试去识别由 Java HotSpot 虚拟机编译的方法。

如果您要收集 .class 或 .jar 文件中的数据，则无需该选项，所提供的到 java 可执行文件的路径在下列环境变量中：JDK_1_4_HOME、 JDK_HOME、 JAVA_PATH 或 PATH。然后您可以将 *program* 指定为具有或不具有扩展名的 .class 或 .jar 文件。

如果您无法在以上任何变量中定义到 java 的路径，或者想禁止由 Java HotSpot 虚拟机所编译方法的识别，就可以使用该选项。如果您使用该选项，则 *program* 必须是 Java 虚拟机，该虚拟机不是 1.4.2_02 之前的版本。 collect 命令不验证 *program* 是否为 JVM 机器，如果不是 JVM 机器则收集可能失败。但是它验证 *program* 为 ELF 可执行文件，如果不是，则 collect 命令打印出一条错误消息。

如果想用 64 位 JVM 机器收集数据，就不能将 -d64 选项用于 32 位 JVM 机器的 java。如果使用，则不会收集到任何数据。相反，您必须在 *program* 或本节给定的环境变量中指定到 64 位 JVM 机器的路径。

-l *signal*

当名为 *signal* 的信号传递到进程时，记录样本包。

该信号可通过全信号名、不带大写字母 SIG 的信号名或信号编码指定。不要使用程序所用信号或可终止执行的信号。推荐的信号为 SIGUSR1 和 SIGUSR2。可通过 kill(1) 命令将信号传递到进程。

如果您即使用 -l 选项也使用 -y 选项，就必须为各自使用不同的信号。

如果您使用该选项且程序具有自身的信号处理程序，您就应该确保用 -l 指定的信号会被传递到收集器的信号处理程序，而且不会被解释或忽略。

关于信号的更多信息，请参阅 signal(3HEAD) 手册页。

-x

将目标进程停止在 exec 系统调用的退出处，以便使调试器能够附加到其中。如果您将 dbx 附加到进程，请使用 dbx 命令 ignore PROF 和 ignore EMT，以确保收集信号被传递到 collect 命令。

-y *signal*[,r]

控制包含名为 *signal* 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（此期间不记录任何数据）和记录状态间转换（此期间记录数据）。样本点始终记录，而与切换的状态无关。

该信号可通过全信号名、不带大写字母 SIG 的信号名或信号编码指定。不要使用程序所用信号或可终止执行的信号。推荐的信号为 SIGUSR1 和 SIGUSR2。可通过 kill(1) 命令将信号传递到进程。

如果您即使用 -l 选项也使用 y 选项，就必须为各自使用不同的信号。

使用 -y 选项时，如果已给定可选的 r 参数，则收集器在记录状态下启动，否则在暂停状态下启动。如果未使用 -y 选项，则收集器在记录状态下启动。

如果您使用该选项且程序具有自身的信号处理程序，您就应该确保用 -y 指定的信号会被传递到收集器的信号处理程序，而且不会被解释或忽略。

关于信号的更多信息，请参阅 signal(3HEAD) 手册页。

输出选项

-d *directory-name*

替换目录 *directory-name* 中的实验。该选项只适用于单独的实验，而不适用于实验组。如果目录不存在，则 collect 命令打印出一条错误消息，然后退出。

-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 命令打印出一条错误消息并退出。如果该组存在，则实验增加到其中。如果 *group-name* 不是绝对路径并且用 -d 指定了目录，则实验组位于目录 *directory-name* 下，否则就放在当前目录下。

-o *experiment-name*

将 *experiment-name* 作为要记录实验的名称。如果 *experiment-name* 不以 .er 结尾，则 collect 命令打印出一条错误消息并退出。关于实验名称及收集器对其处理方式的更多信息，请参阅第 51 页的 "实验名称"

-A *option*

控制是否应将目标进程所使用的负载对象存档或复制到已记录的实验中。 option 的允许值包括：

- off – 不将负载对象存入实验。
- on – 将负载对象存入实验。
- copy – 复制并将负载对象存入实验。

如果您希望将实验从记录的位置复制到不同机器，或从不同机器读取实验，则应该指定 -A copy。使用该选项不会将任何源文件或对象文件复制到实验中。您应该确保在要放置复制实验的机器上可以访问这些文件。

-L *size*

将记录分析数据的数量限制为 *size* MB。该限制适用于基于时钟的分析数据、硬件计数器溢出分析数据和同步等待跟踪数据的数量之和，但不适用于样本点。该限制只是近似，可以被超出。

当达到限制时，不再记录分析数据，但实验会将打开状态保持到目标进程终止。如果启用了周期抽样，则继续写入样本点。

对记录数据的缺省限制为 2000 MB。选择该限制的原因是性能分析器无法处理所含数据大于 2 GB 的实验。要取消该限制，请将 *size* 设置为 unlimited 或 none。

其它选项

-n

不运行目标，但打印在目标运行情况下要生成实验的详细信息。这称为 dry run 选项。

-R

在终端窗口显示性能工具自述文件的文本格式文件。如果未找到自述文件，则打印出警告。不再检查任何参数，也不执行进一步的处理。

-V

打印 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

打印 collect 命令的当前版本和运行中实验的详细信息。

使用 dbx collector 子命令收集数据

从 dbx 运行收集器：

1. **通过输入以下命令将程序装入 dbx。**

```
% dbx program
```

2. **用 collector 命令启用数据收集，选择数据类型并设置可选参数。**

```
(dbx) collector subcommand
```

要获取可用 collector 子命令类型的列表，请输入：

```
(dbx) help collector
```

每个子命令中都必须使用一个 collector 命令。

3. **设置要使用的 dbx 选项并运行该程序。**

如果未直接给定子命令，则会打印出一条警告消息并忽略这条子命令。 collector 子命令的完整列表如下。

数据收集子命令

以下子命令控制收集器所收集数据的类型。如果实验处于激活状态，则会有警告表明以下子命令被忽略。

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 – 禁止基于时钟的分析。
- timer *interval* – 设置分析间隔。 *interval* 的允许值包括：
  - on – 使用 10 毫秒的缺省分析间隔。
  - lo[w] – 使用 100 毫秒的低分辨率分析间隔。
  - hi[gh] – 使用 1 毫秒的高分辨率分析间隔。在 Solaris 7 操作系统下和 Solaris 8 操作系统的早期版本中，必须显式启用高分辨率的分析。关于启用高分辨率分析的详细信息，请参阅第 48 页的 "基于时钟的分析的局限"
  - *value* - 将分析间隔设置为 *value*。*value* 的缺省单位为毫秒。您可以将 *value* 指定为整数或浮点数。可在数值后加后缀 m 选择毫秒单位，或后加 u 选择微秒单位。这个值应该是时钟分辨率的倍数。如果该值大于时钟分辨率但不是其倍数，则向下舍入。如果该值小于时钟分辨率，则将其设置为时钟分辨率。以上两种情况均打印出警告消息。

  缺省设置为 10 毫秒。

收集器在缺省情况下收集基于时钟的分析数据，除非打开用 hwprofile 子命令来收集硬件计数器溢出分析的数据。

hwprofile *option*

控制硬件计数器溢出分析数据的收集。如果您试图在不支持硬件计数器溢出分析的系统中启用它，则 dbx 返回一条警告消息，该命令被忽略。 *option* 的允许值包括：

- on – 打开硬件计数器溢出的分析。缺省操作是为具有正常溢出值的 cycles 计数器收集数据。
- off – 关闭硬件计数器溢出的分析。

- `list` – 返回可用计数器列表。关于该表的描述请参阅第 29 页的 “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 返回一条警告消息。
- `counter` *name value* [*name2 value2*] – 选择硬件计数器 *name* 并将其溢出值设置为 *value*；也可选择另一硬件计数器 *name2* 并将其溢出值设置为 *value2*。溢出值可能是下列值之一。
  - `hi[gh]` – 使用所选计数器高分辨率值。也支持缩写 `h`。
  - `lo[w]` – 使用所选计数器低分辨率值。
  - *number* – 溢出值。必须为正整数。
  - `on` – 使用缺省溢出值。

  两个计数器必须使用不同寄存器。如果使用了同一寄存器，则会打印出一条警告消息并忽略该命令。

  如果硬件计数器计算的事件与内存访问有关，则可在计数器名称前放置 + 号，开始搜索引起计数器溢出的指令的真实 PC。如果搜索成功，则引用的 PC 和有效地址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。如果启用了硬件计数器溢出分析且未给定 `profile` 命令，则关闭基于时钟的分析。

另见第 49 页的 “硬件计数器溢出分析的局限”。

`synctrace` *option*

控制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` – 禁止同步等待跟踪。
- `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` – 在运行时通过校准设置阈值。（与 `on` 等价。）
  - `off` – 关闭同步等待跟踪。
  - `on` – 用将在运行时通过校准设置阈值的缺省阈值。（与 `calibrate` 等价。）
  - *number* – 将阈值设置为 *number*，给定值为以毫秒为单位的正整数。如果 `value` 为 0，则跟踪全部事件。

缺省情况下，收集器不收集同步等待跟踪数据。

`heaptrace` *option*

控制堆跟踪数据的收集。 *option* 的允许值包括

- `on` – 启用堆跟踪。
- `off` – 禁止堆跟踪。

缺省情况下，收集器不收集堆跟踪数据。

mpitrace *option*

控制 MPI 跟踪数据的收集。 *option* 的允许值包括

- on – 启用 MPI 调用的跟踪。
- off – 禁止 MPI 调用的跟踪。

缺省情况下，收集器不收集 MPI 跟踪数据。

sample *option*

控制抽样模式。 *option* 的允许值包括：

- periodic – 启用周期抽样。
- manual – 禁止周期抽样。手动抽样保持启用状态。
- period *value* – 将抽样间隔设置为 *value*，以秒为单位。

缺省情况下，周期抽样启用，其抽样间隔 *value* 为 1 秒。

dbxsample { on | off }

控制 dbx 停止目标进程时对样本的记录。关键字的含义如下：

- 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

禁止数据收集。如果进程正在运行并收集数据，那么它将终止该实验并禁止数据收集。如果进程正在运行但数据收集被禁止，那么会有警告表明它被忽略。如果没有运行中的进程，那么它将禁止后续运行中的数据收集。

enable

启用数据收集。如果进程正在运行但数据收集被禁止，那么它将启用数据收集并启动新的实验。如果进程正在运行但数据收集被禁止，那么会有警告表明它被忽略。如果没有运行中的进程，那么它将启用后续运行中的数据收集。

您可以在进程执行期间任意次地启用和禁止数据收集。每次启用数据收集时，都会创建一个新的实验。

`pause`

暂停数据收集，但保持实验打开状态。仍然记录样本点。如果数据收集已经被暂停，则忽略该子命令。

`resume`

执行 `pause` 后恢复数据收集。如果正在收集数据，则忽略该子命令。

`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

下列子命令定义了实验的存储选项。如果实验处于激活状态，则会有警告表明以下子命令被忽略。

`archive` *mode*

设置用于归档实验的模式。 *mode* 的允许值包括

- `on` – 负载对象的正常归档。
- `off` – 不归档负载对象。
- `copy` – 除了正常归档外，将负载对象复制到实验中

如果您要将实验移动到不同机器，或从另一机器上读取，就应该启用对负载对象的复制。如果实验处于激活状态，会有警告表示该命令被忽略。该命令不会将任何源文件或对象文件复制到实验中。

`limit` *value*

将记录分析数据的数量限制为 *value* MB。该限制适用于基于时钟的分析数据、硬件计数器溢出分析数据和同步等待跟踪数据的数量之和，但不适用于样本点。该限制只是近似，可以被超出。

当达到限制时，不再记录分析数据，但实验会保持打开状态，且继续记录样本点。

对记录数据的缺省限制为 2000 MB。选择该限制的原因是性能分析器无法处理所含数据大于 2 GB 的实验。要取消该限制，请将 *value* 设置为 `unlimited` 或 `none`。

`store` *option*

控制实验的存储位置。如果实验处于激活状态，则会有警告表明该命令是活动的。*option* 的允许值包括：

- `directory` *directory-name* – 设置存储实验和实验组的目录。如果该目录不存在，则会有警告表明这个子命令被忽略。
- `experiment` *experiment-name* – 设置实验的名称。如果实验名称不以 `.er` 结尾，则会有警告表明该子命令被忽略。关于实验名称及收集器对其处理方式的更多信息，请参阅第 51 页的 "实验名称"
- `group` *group-name* – 设置实验组的名称。如果实验组的名称不以 `.erg` 结尾，则会有警告表明该子命令被忽略。

信息子命令

`show`

显示每个收集器控制的当前设置。

`status`

- 报告任意打开实验的状态。

收集运行进程中的数据

收集器使您能够收集运行进程中的数据。如果进程已受 `dbx` 的控制 （位于命令行版本或 IDE 中），您可以暂停该进程并使用在前几节所描述的方法来启用数据收集。

注 – 关于从 IDE 启动性能分析器的信息，请参阅 [性能分析器自述文件]，可通过 `file:/opt/SUNWspro/docs/index.html` 下的文档索引获取。如果在 `/opt` 目录下未安装 Sun Studio 8 软件，在您的系统中的等价路径请询问系统管理员。

如果进程不受 `dbx` 控制，您可以向其附加 `dbx`，收集性能数据，再从该进程分离并保持进程继续进行。如果要为所选后续进程收集性能数据，就必须将 `dbx` 附加到每个进程中。

要从一个不受 `dbx` 控制的运行进程中收集数据：

1. **决定程序的进程 ID (PID)**。

如果从命令行启动程序并将该程序放置在后台中，则其 PID 将由 shell 打印到标准输出。否则，您可以通过输入以下命令来决定程序的 PID。

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

2. **附加到进程**。

■ 从 IDE 的 [调试] 菜单，选择 [调试] → [附加到 Solaris 进程] 并使用对话框选择进程。相关指示请使用联机帮助。

■ 从 dbx，请输入 以下命令。

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

如果 dbx 还未运行，请输入以下命令。

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

关于附加到进程的详细信息，请参阅 《使用 dbx 调试程序》手册。附加到正在运行的进程会使该进程暂停。

3. **启动数据收集**。

■ 从 IDE 的 [调试] 菜单，选择 [性能工具箱] → [启用收集器] 并使用对话框设置数据收集参数。然后选择 [调试] → [继续]，恢复执行该进程。

■ 从 dbx 中，使用 collector 命令设置数据收集参数，并使用 cont 命令恢复执行该进程。

4. **从进程分离**。

收集数据完成后，暂停程序并从 dbx 分离该进程。

■ 在 IDE 中，右键单击 [调试器] 窗口 [会话] 视图中进程的会话，并从上下文菜单选择 [分离]。如果 [会话] 视图未显示，则单击 [调试器] 窗口顶部的 [会话] 按钮。

■ 从 dbx 中，输入以下命令。

```
(dbx) detach
```

如要收集任何类型的跟踪数据，则必须在运行程序之前预装收集器库 libcollector.so，因为该库向包装器提供了能进行数据收集的实函数。此外，收集器将包装器函数增加到其它系统库调用，保证了性能数据的完整性。如果未预装收集器库，则这些包装器函数不能插入。关于收集器如何插入系统库函数的更多信息，请参阅第 42 页的 “使用系统库”。

要预装 libcollector.so，必须使用环境变量同时设置库的名称和到库的路径。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 和 / 或 LD_LIBRARY_PATH_64 设置到库的路径。(如果 _32 和 _64 变量未定义，则使用 LD_LIBRARY_PATH。) 如果已定义了这些环境变量，则对其增加新值。

表 3-2 预装库 libcollector.so 的环境变量设置

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

如果 Sun 编译器和工具未安装在 /opt/SUNWspro 中，那么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，不过当使用 SPARC V9 64 位体系结构时这样做会很复杂。

注 – 运行后删除 LD_PRELOAD 和 LD_LIBRARY_PATH 设置，这样变量对从相同 shell 启动的其它程序就不生效。

如要从已运行的 MPI 程序收集数据，则必须将 dbx 的独立实例附加到每个进程并为每个进程启用收集器。将 dbx 附加到 MPI 作业中的进程后，每个进程将被停止并在其他时间重新启动。时间差异可能更改 MPI 进程间的交互，并影响收集的性能数据。要尽量解决该问题，一个解决方案就是使用 pstop(1) 停止所有的进程。不过，将 dbx 附加到这些进程后，您必须从 dbx 重新启动这些进程，而且重新启动进程时会出现时间延迟，这会影响 MPI 进程的同步。另见第 69 页的 “从 MPI 程序收集数据”。

从 MPI 程序收集数据

收集器可以从使用 SUN 消息传递接口 (MPI) 库的多进程程序收集性能数据。 MPI 库包括在 Sun HPC ClusterTools™ 软件中。如果可能，您应该使用 ClusterTools 软件的最新版本 4.0，当然也可以使用 3.1 或兼容的版本。要启动并行作业，请使用 Sun 群运行时环境 (CRE) 命令 mprun。相关的更多信息请参阅 Sun HPC ClusterTools 文档。关于 MPI 和 MPI 标准的信息，请参阅 MPI web 站点 http://www.mcs.anl.gov/mpi。

因为 MPI 和 收集器实现的方式不同，所以每个 MPI 进程记录一个独立的实验。每个实验必须具有唯一的名称。实验被存储的位置和方式取决于对 MPI 作业可用的文件系统的类型。关于存储实验的问题在接下来的子节中讨论。

要从 MPI 作业收集数据，您既可以在 MPI 下运行 collect 命令，也可以在 MPI 下启动 dbx 并使用 dbx collector 子命令。在后续子节中讨论这些选项。

存储 MPI 实验

因为多进程环境比较复杂，所以从 MPI 程序收集性能数据时应该意识到关于存储 MPI 实验的某些问题。这些涉及的问题有数据收集和存储的效率以及实验的命名。关于命名实验 （包括 MPI 实验）的信息请参阅第 51 页的 “数据存储的位置”。

收集性能数据的每个 MPI 进程创建其自身的实验。MPI 进程创建实验时，该进程会锁定实验目录。所有其它 MPI 进程必须在可以使用该目录之前等待，一直到锁定被释放。因此，如果在文件系统上存储的实验可以访问所有的 MPI 进程，则依次创建这些实验；但是如果在文件系统上存储的实验是每个 MPI 进程的局部实验，则并行创建这些实验。

如果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，则这些实验具有唯一的名称。*n* 的值由 MPI 进程在实验目录上获得锁定的顺序决定，并且不能保证对应到进程的 MPI 级别。如果将 dbx 附加到运行中的 MPI 作业的 MPI 进程，则 *n* 将由附加的顺序决定。

如果将这些实验存储在本地文件系统上并以标准格式指定实验名称，则这些名称不是唯一的。例如，假设在具有 4 个单处理器节点的计算机上运行 MPI 作业，而这些节点标为 node0、 node1、 node2 和 node3。每个节点具有一个名为 /scratch 的本地磁盘，并将这些实验存储在该磁盘上的 *username* 目录中。 MPI 作业创建的实验具有以下全路径名称。

```
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有一个名为 test.1.er 的实验。如果 MPI 作业完成后将实验移动到公共位置，则必须确保这些名称是唯一的。例如，要将这些实验移动到假设可以从所有节点访问的初始目录，并重命名这些实验，请输入以下命令。

```
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 node0 'er_mv /scratch/username/test.1.er test.0.er'
```

对于大的 MPI 作业，可能要使用脚本将实验移动到公共位置。请勿使用 Unix 命令 cp 或 mv；关于如何复制和移动实验的信息请参阅第 129 页的 “操作实验”。

如果未指定实验名称，则收集器使用 MPI 级别以标准格式 *experiment.n*.er 构造实验名称，但这种情况下的 *n* 是 MPI 级别。如果指定了实验组，则主干 *experiment* 是实验组名称的主干，否则它为 test。不管使用公共文件系统还是本地文件系统，实验的名称都是唯一的。因此，如果使用本地文件系统记录实验并将这些实验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。

如果您不知道哪个本地文件系统可用，则请使用 df -lk 命令或咨询系统管理员。您应该始终确保实验被存储在现有的目录中，该目录是唯一定义的且不能用于任何其它实验。此外，还应该确保该文件系统对这些实验具有足够的空间。关于如何估计所需的空间请参阅第 53 页的 “估计存储需求”。

注 – 除非您已经具有用于运行实验的负载对象和源文件或带有相同路径和时间标记的副本，否则在计算机或节点间复制或移动实验时不能查看注释反汇编代码中的注释源代码或源代码行，。

在 MPI 下运行 collect 命令

要在 MPI 控制下使用 collect 命令收集数据，请使用以下语法。

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

此处的 *n* 是 MPI 要创建的进程数。该步骤创建 *n* 个 collect 的独立实例，每个实例记录一个实验。关于存储实验的位置和方法，请参阅第 51 页的 “数据存储的位置” 一节。

要确保从不同 MPI 运行的实验集合被分别存储，请为每个运行的 MPI 使用 -g 选项创建实验组。实验组应该存储在所有 MPI 进程可以访问的文件系统上。创建实验组也有助于将单一的 MPI 运行的实验集合装入性能分析器中。创建组的另一种方法是使用 -d 选项为每个 MPI 运行指定独立的目录。

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，请使用以下语法。

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

此处的 *n* 是要由 MPI 创建的进程数，而 *collection-script* 是包含设置和启动数据收集必需命令的 dbx 脚本。该步骤创建了 *n* 个 dbx 的独立实例，每个实例记录其中一个 MPI 进程上的实验。如果未定义实验名称，则该实验将用 MPI 级别标定。关于存储实验的位置和方法，请参阅第 69 页的 “存储 MPI 实验” 一节。

通过使用收集脚本和对 MPI_Comm_rank() 的调用，可以命名带有 MPI 级别的实验。例如，在 C 程序中会插入以下代码行。

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

在 Fortran 程序中插入以下代码行。

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

例如，如果该调用插入到第 17 行，则可以使用以下的脚本。

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

与 ppgsz 一起使用 collect

通过运行 ppgsz 命令上的 collect 并指定 -F on 标记，您可以与 ppgsz(1) 一起使用 collect。创建实验位于 ppgsz 可执行程序上并且对其没有兴趣。如果路径找到 32 位版本的 ppgsz，且实验在支持 64 位进程的系统上运行，则首先要做的是 exec 它的 64 位版本，创建 _x1.er。可执行文件的派生，创建 _x1_f1.er。

子进程尝试 exec 路径上第一个目录中的命名目标，然后对第二个目录的目标执行该操作，依此类推，直到其中一个 exec 成功。例如，如果第三个尝试成功，则前两个后续的实验命名为 _x1_f1_x1.er 和 _x1_f1_x2.er，且两个实验完全为空。目标上的实验是来自成功 exec 的其中一个目标 （示例中的第三个目标），且命名为 _x1_f1_x3.er，存储在创建实验下。通过在 test.1.er/_x1_f1_x3.er 上调用分析器或 er_print，可以直接处理该目标。

如果 64 位 ppgsz 是初始进程，或如果 32 位 ppgsz 在 32 位内核上调用， exec 真实目标的派生子将其数据置于 _f1.er 中，而真实目标的实验位于 _f1_x3.er，前提是按照上述示例假定相同的路径。

第 4 章

er_print 命令行性能分析工具

本章说明了如何使用 er_print 公用程序进行性能分析。 er_print 公用程序打印性能分析器支持的各种显示的 ASCII 版本。除非将其重定向到文件或打印机，这些信息将写入标准输出。必须将收集器生成的一个或多个实验或实验组的名称做为参数赋予 er_print 公用程序。可以使用 er_print 公用程序来显示函数、调用方与被调用方的性能度量，源代码和反汇编列表，样本信息，地址空间数据以及执行统计。

本章涵盖了以下主题。

- er_print 语法
- 度量列表
- 控制函数列表的命令
- 控制调用方与被调用方列表的命令
- 控制泄漏和分配列表的命令
- 控制源码和反汇编列表的命令
- 控制数据空间列表的命令
- 列出实验、样本、线程和 LWP 的命令
- 控制选择的命令
- 控制负载对象选择的命令
- 列出度量的命令
- 控制输出的命令
- 打印其它显示的命令
- 设置缺省值的命令
- 设置仅用于性能分析器缺省的命令
- 杂项命令
- 示例

关于收集器收集的数据描述，请参阅第 2 章。

关于如何使用性能分析器以图形格式显示信息的指示，请参阅 [联机帮助]。

er_print 语法

er_print 的命令行语法如下所示：

```
er_print [ -script script | -command | - | -V ] experiment-list
```

er_print 的选项在表 4-1 中列出。

表 4-1 er_print 命令的选项

选项	描述
-	读取从键盘输入的 er_print 命令。
-script *script*	从包含 er_print 命令列表的 *script* 文件 （每行一个命令）读取命令。如果 -script 选项不存在， er_print 从终端或命令行读取命令。
-*command* [*argument*]	处理给出的命令。
-V	显示版本信息并退出。

在 er_print 命令行上可以显示多个选项。这些选项以出现的顺序处理。可以用任何顺序混合脚本、连字符和显式命令。如果不支持任何命令或脚本，则缺省进入交互模式，交互模式中的命令从键盘输入。要退出交互模式，请输入 **quit** 或 Ctrl-D。

er_print 接受的命令在以下部分中列出。只要意思明确就可以用更短的字符串来缩写任何命令。

度量列表

很多 er_print 命令使用度量关键字的列表。该列表的语法如下所示：

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

除了 size、 address 和 name 关键字，度量关键字由三部分组成：度量类型字符串、度量可视性字符串和度量名称字符串。这三部分连在一起没有空格，如下所示：

```
<type><visibility><name>
```

度量类型和度量可视性字符串由类型和可视性字符组成。

允许的度量类型字符在表 4-2 中给出。包含多种类型字符的度量关键字扩展成一列度量关键字。例如， ie.user 扩展为 i.user:e.user。

表 4-2　度量类型字符

字符	描述
e	显示排除度量值
i	显示包括度量值
a	显示属性度量值 （仅为调用方与被调用方度量）

允许的度量可视性字符在表 4-3 中给出。改变可视性字符串中可视性字符的顺序不会影响相应度量的显示顺序。例如， i%.user 和 i.%user 都会被解释为 i.user:i%user。

只在可视性中不同的度量总是以标准顺序一起显示。如果两个只在可视性中有不同的度量关键字被某些其它关键字分开，则在两个度量中的第一个位置必须以标准顺序显示度量。

表 4-3　度量可视性字符

字符	描述
.	将度量显示为时间。应用到测量循环计数的定时度量和硬件计数器度量。其它度量解释为 "+"。
%	将度量显示为全部程序度量的百分比。对于调用方与被调用方列表中的属性度量，将度量显示为选定函数的包括度量百分比。
+	将度量显示为一个绝对值。对于硬件计数器，该值是事件计数。定时度量解释为 "."。
!	不显示任何度量值。不能用于与其它可视性字符的组合。

类型和可视性字符串具有多个字符时，首先扩展类型。因此， ie.%user 扩展到 i.%user:e.%user，然后解释为 i.user:i%user:e.user:e%user。

可视性字符句号 .、加号 + 和百分号 % 用于定义排序顺序的目的是一样的。因此， sort i%user、 sort i.user 和 sort i+user 都表示分析器只要以任何形式可见就应该通过包括用户 CPU 时间排序，而 sort i!user 则表示不管分析器是否可见都应该通过包括用户 CPU 时间排序。

表 4-4 列出了可用于定时度量、同步延迟度量、内存分配度量、 MPI 跟踪度量和两个通用硬件计数器度量的 er_print 度量名称字符串。对于其它硬件计数器度量，其度量名称字符串与计数器名称相同。可以通过使用不带参数的 collect 命令获得计数器名称的列表。关于硬件计数器的更多信息请参阅第 28 页的 “硬件计数器溢出分析数据”。

表 4-4 度量名称字符串

种类	字符串	描述
定时度量	user	用户 CPU 时间
	wall	墙时钟时间
	total	全部 LWP 时间
	系统	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	其它等待时间
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	MPI 发送操作中发送的字节数
	mpireceive	MPI 接收操作的数量
	mpibytesrecv	MPI 接收操作中接收的字节数
	mpiother	对其它 MPI 函数的调用数
内存分配度量	alloc	分配的数目
	balloc	分配的字节
	leak	泄漏的数量
	bleak	泄漏的字节
硬件计数器溢出度量	cycles	CPU 循环
	insts	发出的指令

除了在表 4-4 中列出的名称字符串之外，还有两个仅能用于缺省度量列表的名称字符串。它们是匹配任何硬件计数器名称的 hwc 和匹配任何度量名称字符串的 any。此外需要注意的是 cycles 和 insts 通用于 SPARC¨ 和 Intel，而其它特定的体系结构则喜欢用其它的名称字符串。要列出所有可用的计数器，请使用不带参数的 collect 命令。

控制函数列表的命令

以下命令控制如何显示函数信息。

functions

用当前选定的度量写入函数列表。函数列表包括了在选定用于显示函数的负载对象中的所有函数，还包括了任何负载对象，该对象的函数用 `object_select` 命令隐藏。

可以使用 `limit` 命令限制写入行的数目 （请参阅第 88 页的 “控制输出的命令”）。

打印的缺省度量是排除和包括用户 CPU 时间，该度量同时用两种形式显示：全部程序度量的秒数和百分比。可以用 `metrics` 命令更改当前显示的度量。要更改当前显示的度量，必须先执行 `functions` 命令。此外，也可以用 `dmetrics` 命令更改缺省设置。

对于用 Java 编程语言编写的应用程序，显示的函数信息取决于 Java 模式的设置，设置选项有 `on`、 `expert` 或 `off`。 Java 模式设置到 on 时，显示的函数信息包括与 Java 方法有冲突的度量和调用的任何本机方法。将 Java 模式设置到 expert，显示了独立于方法解释版本的 HotSpot 编译方法。将模式设置到 off 显示了源于 JVM 本身（而不是源于由 JVM 解释的 Java 应用程序）的函数和任何编译的方法以及本机方法。

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`，也可以是用冒号分隔的度量关键字列表。以下示例说明了度量列表。

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

该命令指示 `er_print` 显示以下度量：

- 用秒来表示的包括用户 CPU 时间
- 包括用户 CPU 时间百分比
- 用秒来表示的排除用户 CPU 时间
- 排除用户 CPU 时间百分比

`metrics` 命令完成后，打印显示当前度量选项的消息。上述示例打印出的消息如下所示：

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

关于度量列表语法的信息，请参阅第 74 页的 “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，则忽略并显示警告，且先前的设置仍然生效。

sort *metric_spec*

按指定的度量将函数列表排序。字符串 *metric_spec* 是第 74 页的 “度量列表”中所述的度量关键字之一，如本示例所示。

```
% sort i.user
```

该命令告知 er_print 用包括用户 CPU 时间来排序函数列表。如果度量不在装入的实验中，则打印警告并忽略该命令。命令完成后，打印排序度量。

fsummary

为函数列表中的每个函数写入汇总度量面板。可以使用 limit 命令限制写入面板的数目（请参阅第 88 页的 “控制输出的命令”）。

汇总度量面板包括了函数或负载对象的名称、地址和大小，（对于函数）源文件、对象文件和负载对象的名称，以及所有选定函数或负载对象记录的以值和百分比显示的 （排除和包括）度量。

fsingle *function_name* [*N*]

写入指定函数的汇总面板。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* 个函数写入汇总度量面板。命令在命令行上给出时，需要 *N*；如果不需要该参数，则忽略。当交互给出不带 *N* 的命令但又需要 *N* 时，打印带有对应 *N* 值的函数列表。

关于函数的汇总度量的描述，请参阅 fsummary 命令描述。

控制调用方与被调用方列表的命令

以下命令控制如何显示调用方与被调用方信息。

调用方与被调用方

按排序顺序打印每个函数的调用方与被调用方面板。可以使用 `limit` 命令限制写入面板的数目 （请参阅第 88 页的 “控制输出的命令”）。选定（中央）的函数用星号标记，如下例所示：

```
属性排除      包括       名称
用户 CPU  用户 CPU  用户 CPU
 秒        秒        秒
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
```

在本示例中， `gpf` 是选定的函数，被 `commandline` 调用，该函数调用 `gpf_a` 和 `gpf_b`。

`cmetrics` *metric_spec*

指定调用方与被调用方度量的选项。 *metric_spec* 是度量关键字的列表，用冒号分隔，如下例所示：

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

该命令指示 `er_print` 显示以下度量。

- 用秒来表示的包括用户 CPU 时间
- 包括用户 CPU 时间百分比
- 用秒来表示的属性用户 CPU 时间
- 属性用户 CPU 时间百分比

`cmetrics` 命令完成后，打印出显示当前度量选项的消息。上述示例打印出的消息如下所示：

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

关于度量列表语法的信息，请参阅第 74 页的 “度量列表”。要查看可用度量的列表，请使用 `cmetric_list` 命令。

csingle *function_name* [*N*]

写入命名函数的调用方与被调用方面板。当有多个函数具有相同的名称时，需要使用可选参数 *N*。为带有给定函数名称的第 *N* 个函数写入调用方与被调用方面板。命令在命令行上给出时，需要 *N*；如果不需要该参数，则忽略。当交互给出不带 *N* 的命令但又需要 *N* 时，打印带有对应 *N* 值的函数列表。

csort *metric_spec*

按指定的度量将调用方与被调用方显示排序。字符串 *metric_spec* 是第 74 页的 "度量列表"中所述的度量关键字之一，如本示例所示。

```
% csort a.user
```

该命令告知 `er_print` 将调用方与被调用方显示按属性用户 CPU 时间排序。命令完成后，打印排序度量。

控制泄漏和分配列表的命令

本节描述了与内存分配和释放有关的命令。

leaks

显示内存泄漏列表，由通用调用栈累计。每个条目显示了泄漏总数和给定调用栈的总泄漏字节数。该列表是按泄漏的字节数排序的。

allocs

显示内存分配列表，由通用调用栈累计。每个条目显示了分配数和给定调用栈的总分配字节数。该列表是按分配的字节数排序的。

控制源码和反汇编列表的命令

以下命令控制如何显示注释的源码和反汇编码。

pcs

写入程序计数器 (PC) 及其度量的列表，按当前排序度量排序。该列表包括显示每个负载对象的累计度量的行，其中对象的函数用 `object_select` 命令隐藏。

psummary

按当前排序度量指定的顺序为 PC 列表中每个 PC 写入汇总度量面板。

lines

写入源代码行及其度量的列表，按当前排序度量排序。该列表包括显示每个函数累计度量的行，其中的函数不具有行号信息或其源文件未知；还包括显示每个负载对象累计度量的行，其中对象的函数用 `object_select` 命令隐藏。

lsummary

按当前排序度量指定的顺序为行列表中每行写入汇总度量面板。

source { *filename* | *function_name* } [*N*]

为指定文件或包含指定函数的文件写出注释的源代码。任一种情况下的文件都必须位于路径中的目录。

只有在文件或函数名称模糊的情况下才使用可选参数 *N*（正整数），这种情况下使用第 *N* 个可能的选择。如果给出了不带数值说明符的模糊名称，则 `er_print` 打印可能的对象文件名称的列表；如果给出的名称是函数，则函数的名称追加到对象文件的名称，而代表该对象文件 *N* 的值的数字也会打印。

disasm { *filename* | *function_name* } [*N*]

为指定文件或包含指定函数的文件写出注释的反汇编码。任一种情况下的文件都必须位于路径中的目录。

可选参数 *N* 与 `source` 命令的可选参数的使用方法相同。

scc *com_spec*

指定显示在注释源码列表中的编译器注释类。类列表是用冒号分隔的类列表，包含了零个或多个以下消息类。

表 4-5 编译器注释消息类

类	含义
b[asic]	显示基本级别消息。
v[ersion]	显示版本消息（包括源文件名称和最后一次修改日期）、编译器组件的版本、编译日期和选项。
pa[rallel]	显示关于并行性的消息。
q[uery]	显示关于影响代码优化的代码问题。
l[oop]	显示关于循环优化和转换的消息。
pi[pe]	显示关于循环流水线的消息。
i[nline]	显示关于函数内联的消息。
m[emops]	显示关于诸如装入、存储、预取等内存操作的消息。
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 不能与其它类一起使用。

如果没有给出 scc 命令，则显示的缺省类为 basic。如果 scc 命令与空的 *class-list* 一起给出，则编译器注释被关闭。 scc 命令通常仅用于 .er.rc 文件。

sthresh *value*

指定注释源代码中突出显示度量的阈值百分比。如果任何度量的值等于或大于文件中任何源代码行最大度量值的 *value* %，则度量发生的行在行首插入 ##。

dcc *com_spec*

指定显示在注释反汇编码列表中的编译器注释的类。类列表是用冒号分隔的类列表。可用类的列表与列出注释源代码的类列表相同。以下选项可增加到类列表。

表 4-6 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]	在注释反汇编码列表中交叉列出源码。
nos[rc]	不在注释反汇编码列表中交叉列出源码。
as[rc]	在注释反汇编码列表中交叉列出注释源码。

dthresh *value*

指定注释反汇编码中突出显示图例的阈值百分比。如果任何度量的值等于或大于文件中任何指令最大度量值的 *value* %，则度量发生的行在行首插入 ##。

setpath *path_list*

设置用于查找源码、对象等文件的路径。 *path_list* 是以冒号分隔的目录列表。如果任何目录具有冒号字符，则目录应该用反斜杠来转义。特殊目录名 $expts 以装入实验的顺序引用当前实验集，也可以缩写为单一的 $ 字符。

缺省设置为：$expts:.. 如果搜索当前路径设置时找不到文件，则使用编译中的完整路径名。

不带参数的 setpath 打印当前路径。

addpath *path_list*

将 *path_list* 追加到当前的 setpath 设置。

控制数据空间列表的命令

data_objects

写入数据对象及其度量的列表。仅可用于指定了积极回溯的 HW 计数器实验和用 -xhwcprof 编译的文件中的对象。（仅可用于 SPARC 上的 C）。有关更多信息请参阅《C 用户指南》或 cc(1) 手册页。

data_osingle *name* [*N*]

写入命名数据对象的汇总度量面板。对象名称不明确时，需要使用可选参数 *N*。指令在命令行上时，需要 *N*；如果不需要该参数，则忽略。仅可用于指定了积极回溯的 HW 计数器实验和用 -xhwcprof 编译的文件中的对象。（仅可用于 SPARC 上的 C）。有关更多信息请参阅 《C 用户指南》或 cc(1) 手册页。

data_olayout

按在实验负载对象中定义的顺序，为具有数据派生度量数据的程序数据对象写入带注释的数据对象布局。每个累计数据对象显示带有合计度量属性，后跟按偏移顺序的所有元素，每个元素具有自己的度量和相对 32 字节块的大小位置指示符。

列出实验、样本、线程和 LWP 的命令

本节描述了列出实验、样本、线程和 LWP 的命令。

experiment_list

显示装入实验及其 ID 号的完整列表。每个实验用索引列出，其中索引用于选择样本、线程或 LWP。

以下示例是实验列表的示例。

```
(er_print) experiment_list
ID Experiment
== ==========
1 test.1.er
2 test.6.er
```

sample_list

显示当前选定用于分析的样本列表。

以下示例是样本列表的示例。

```
(er_print) sample_list
Exp Sel      Total
=== ======= =====
  1 1-6        31
  2 7-10,15    31
```

lwp_list

显示当前选定用于分析的 LWP 列表。

thread_list

显示当前选定用于分析的线程列表。

cpu_list

显示当前选定用于分析的 CPU 列表。

控制选择的命令

选择列表

选择的语法如下示例所示。该语法用于命令描述中。

```
[experiment-list:]selection-list[+[experiment-list:]selection-list … ]
```

每个选择列表都可以将实验列表做为前缀，之间用冒号 （不加空格）分隔。可以用 + 符号将多个选择列表连在一起产生多个选择。

实验列表和选择列表具有相同的语法，不是关键字 `all` 就是编号列表或数字范围 (*n-m*)，其中用逗号 （不加空格）分隔，如以下示例所示。

```
2,4,9-11,23-32,38,40
```

实验编号可以通过使用 `exp_list` 命令来决定。

选择的某些示例如下所示。

```
1:1-4+2:5,6
all:1,3-6
```

在第一个示例中，对象 1 到 4 从实验 1 选择，而对象 5 和 6 从实验 2 选择。在第二个示例中，对象 1， 3 到 6 从所有示例选择。对象可以是 LWP、线程或样本。

选择命令

选择 LWP、样本、 CPU 和线程的命令不是独立的。如果命令的实验列表与前一个命令的实验列表不同，则将最后一个命令的实验列表按以下方法全部应用到三个选择目标：LWP、样本和线程。

- 不在最后一个实验列表中实验的现有选择被关闭。
- 最后一个实验列表中实验的现有选项会保存。
- 为没有做出任何选择的目标，将选择设置到 `all`。

sample_select *sample_spec*

选择要显示信息的样本。命令完成后显示选定的样本列表。

lwp_select *lwp_spec*

选择要显示信息的 LWP。命令完成后显示选定的 LWP 列表。

thread_select *thread_spec*

选择要显示信息的线程。命令完成后显示选定的线程列表。

cpu_select *cpu_spec*

选择要显示信息的 CPU。命令完成后显示选定的 CPU 列表。

控制负载对象选择的命令

object_list

显示负载对象的列表。每个负载对象的名称以 yes 为前缀，表示该对象的函数显示在函数列表中，也可以用 no 为前缀，表示该对象的函数不显示在函数列表中。

以下是负载对象列表的示例。

```
(er_print) object_list
Sel Load Object
=== ==================================================
yes /tmp/var/synprog/synprog
yes /opt/SUNWspro/lib/libcollector.so
yes /usr/lib/libdl.so.1
yes /usr/lib/libc.so.1
```

object_select *object1,object2,...*

选择要显示关于负载对象中函数信息的负载对象。 *object-list* 是负载对象的列表，用逗号分隔但无空格。对于未选定的负载对象，显示整个负载对象的信息，而不显示负载对象中函数的信息。

负载对象的名称应该是全路径名称或基名。如果对象名称本身包含逗号，则必须为该名称加上双引号。

列出度量的命令

以下命令列出了当前选定的度量和所有可用的度量关键字。

metric_list

显示当前函数列表中选定的度量和可用于其它命令 （例如 metrics 和 sort）的度量关键字列表以引用函数列表中各种类型的度量。

cmetric_list

显示当前调用方与被调用方列表中选定的度量和可用于其它命令 （例如 cmetrics 和 csort）的度量关键字列表，以引用调用方与被调用方列表中各种类型的度量。

注 – 只有用 cmetrics 命令才能指定属性度量用于显示，而 metrics 命令则不行，并且只可与 callers-callees 命令一起显示，而 functions 命令则不行。

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，然后打开后续输出的 *filename*。如果指定破折号 (-) 而不是 *filename*，则输出写入标准输出。

`limit` *n*

限制对报告的前 *n* 个条目的输出； *n* 是无符号的正整数。

`name { long | short }`

指定要使用函数名称的长名形式还是短名形式 （仅 C++）。

`javamode { on | expert | off }`

将 Java 实验的模式设置为 on（显示 Java 模型）， expert（显示 Java 模型，但显示与解释的方法独立的 HotSpot 编译方法）或 off （显示机器模型）。

打印其它显示的命令

`header` *exp_id*

显示关于指定实验的描述性信息。 *exp_id* 可从 `exp_list` 命令获得。如果 *exp_id* 为 `all` 或未给定，则显示所有装入实验的信息。

除后跟的每个标题，还打印任何错误或警告。每个实验的标题用破折线分隔。

在命令行需要 *exp_id*，而脚本或交互模式中则不需要。

`objects`

列出带有任何错误或警告消息的负载对象，其中的消息因使用性能分析的负载对象而产生。通过使用 `limit` 命令可以限制列出的负载对象数目 （请参阅第 88 页的 “控制输出的命令”）。

`overview` *exp_id*

写入当前为指定实验选择的每个样本的样本数据。 *exp_id* 可从 `exp_list` 命令获得。如果 *exp_id* 为 `all` 或未给定，则显示所有实验的样本数据。在命令行上需要 *exp_id*，而在脚本或交互模式中则不需要。

statistics *exp_id*

写入执行统计，累计指定实验的当前样本集。关于出现的执行统计的定义和含义，请参阅 getrusage(3C) 和 proc(4) 手册页。执行统计包括了源于系统线程的统计，收集器不为该线程收集任何数据。 Solaris 操作系统版本 7 和 8 中的标准线程库创建未分析的系统线程。这些线程大部分时间休眠，休眠时间在统计显示中显示为 [其它等待] 时间。

exp_id 可从 experiment_list 命令获得。如果 *exp_id* 未给出，则显示所有实验数据的总和，累计每个实验的样本集。如果 *exp_id* 是 all，则显示每个实验的总和以及独立统计。

设置缺省值的命令

可以使用以下命令设置 er_print 和性能分析器的缺省值。可以使用以下命令设置缺省值：它们不能用于 er_print 的输入。但可以包括在名为 .er.rc 的缺省文件中。某些命令仅适用于性能分析器。

要设置所有实验的缺省，则缺省文件可以包括在起始目录中；而要设置本地缺省，则缺省文件可以包括在任何其它目录中。 er_print、 er_src 或性能分析器启动时，在当前目录和起始目录扫描缺省文件，如果存在则读取该缺省文件，同时还读取系统缺省文件。起始目录中 .er.rc 文件的缺省覆盖系统的缺省，而当前目录中 .er.rc 文件的缺省覆盖起始和系统的缺省。

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

缺省文件也可以包括 scc、sthresh、dcc 和 dthresh 命令。在缺省文件中给出了多个 dmetrics 和 dsort 命令，而且文件内的命令被连接。

dmetrics *metric_spec*

指定在函数列表中显示或打印的缺省度量。度量列表的语法和使用在 第 74 页的 “度量列表” 中描述。列表中度量关键字的顺序决定了性能分析器中 [度量] 选择器中度量出现和显示的顺序。

通过将相应的属性度量增加在列表中首次出现每个度量名称的前面，可以从函数列表缺省度量派生 [调用方与被调用方] 列表的缺省度量。

dsort *metric_spec*

按函数列表排序的方式指定缺省度量。排序度量是该列表中的第一个度量，与任何已装入实验中的度量匹配且受到以下条件的限制：

- 如果 *metric_spec* 中的条目具有可视性的感叹号字符串 !，则使用与名称匹配的第一个度量，不管该度量是否可见。
- 如果 *metric_spec* 中的条目具有任何其它可见性字符串，则使用与名称匹配的第一个可见度量。

度量列表的语法和使用在第 74 页的 “度量列表”中描述。

[调用方与被调用方] 列表的缺省排序度量是与函数列表缺省排序度量对应的属性度量。

gdemangle *library.so*

将路径设置到支持 API 裁减 C++ 函数名称的共享对象。共享对象必须按照 GNU 标准 `libiberty.so` 接口输出 C 函数 `cplus_demangle()`。

设置仅用于性能分析器缺省的命令

tlmode *tl_mode*

设置性能分析器 [时间线] 标签的显示模式选项。选项的列表是用冒号分隔的列表。下表具体描述了允许的选项。

表 4-7 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]`	在根对齐调用栈
`le[af]`	在叶对齐调用栈
`d[epth]` *nn*	设置可以显示的调用栈的最大深度

选项 `lwp`、`thread` 和 `cpu` 如同 `root` 和 `leaf`，是互斥的。如果列表中包括了多个互斥的选项集，则仅使用最后一个选项。

tldata *tl_data*

选择显示在性能分析器 [时间线] 标签中的缺省数据类型。类型列表中的类型用冒号分隔。允许的类型在下表中列出。

表 4-8 时间线显示数据类型

类型	含义
sa[mple]	显示样本数据
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

datamode { *on*| *off* }

将显示与数据空间有关的屏幕设置为 on（标签可见）或 off（使其不可见）。

杂项命令

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

将指定负载对象的映射文件写入 *mapfilename* 文件。如果指定破折号 (-) 而不是 *mapfilename*，则 er_print 将映射文件写入标准输出。

script *file*

处理脚本文件 *file* 的附加命令

version

打印 er_print 的当前发行版本编号。

quit

终止当前脚本的处理或退出交互模式。

help

打印 er_print 命令的列表。

示例

- 以下示例从实验生成了一个类似 gprof 的列表。输出是一个名为 er_print.out 的文件，该文件列出了最前面的 100 个函数，后跟调用方与被调用方数据，按每个函数的属性用户时间排序。

  ```
  er_print -outfile er_print.out -metrics e.user:e%user\
  -sort e.user -limit 100 -functions -cmetrics a.user:a%user\
  -csort a.user -callers-callees test.1.er
  ```

  此外，也可以将该示例简化为以下独立的命令。不过要记住，在大的实验或应用程序中每个对 er_print 的调用是时间密集计算：

  - ```
    er_print -metrics e.user:e%user -sort e.user \
    -limit 100 -functions test.1.er
    ```
  - ```
    er_print -cmetrics a.user:a%user -csort a.user \
    -callers-callees test.1.er
    ```

- 该示例总结了在函数中是如何花费时间的。

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

- 该示例显示了调用方和被调用方的关系。

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

- 该示例显示了哪些行比较重要。源码行信息假设代码用 -g 编译和链接。在 Fortran 函数和例程的函数名称后面要附加下划线。函数名称后加 1 用于区分 *myfunction* 的多个实例。

  er_print -source *myfunction* 1 test.*.er

- 该示例仅显示了编译器注释。不需要运行您的程序来使用该命令。

  er_src -*myfile*.o

- 这些示例使用了墙时钟分析以列出函数和调用方 / 被调用方。

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

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

- 该示例显示了高层次的 MPI 函数。 MPI 中有很多内部软件层，但这只是查看入口点的一种方法。其中可能会有很多重复符号，不过可以忽略它们。

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

第 5 章

理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。度量为目标程序结构中的各种元素而计算，例如指令、源代码行、函数和负载对象。除标题外，为每个收集的事件记录的数据还包括两部分：

- 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

将度量与程序结构关联的过程并不简单，取决于编译器所做的插入、转换和优化。本章描述了该关联过程并讨论了对性能分析器显示的影响。

本章涵盖了以下主题：

- 数据收集如何工作
- 解释性能度量
- 调用栈和程序执行
- 将地址映射到程序结构
- 将数据地址映射到程序数据对象
- 注释代码列表

数据收集如何工作

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

所有的实验必须具有三个文件：

- 日志文件是 ASCII 文件，包含的信息有收集的数据内容，各种组件的版本，以及目标生命周期中各种事件的记录。
- 负载对象文件是用于记录与时间有关的信息的 ASCII 文件，这些信息包括什么负载对象被装入目标的地址空间，以及装入或卸载的时间。
- 概述文件是包含实验中每个样本点记录的使用信息的二进制文件。

此外，实验包含代表处理期间文件配置事件的二进制数据文件。每个数据文件具有一系列事件，如下面的第 98 页的 “解释性能度量”所述。每种类型的数据使用单独的文件，但每个文件由目标中所有的 LWP 共享。数据文件按以下方式命名：

表 5-1 数据类型和相应的文件名称

数据类型	文件名
时钟分析	profile
HWC 分析	hwcounters
同步跟踪	synctrace
堆跟踪	heaptrace
MPI 跟踪	mpitrace

对于时钟分析或 HW 计数器溢出分析，将数据写入时钟周期或时钟溢出调用的信号处理程序中。对于同步跟踪、堆跟踪或 MPI 跟踪，从正常用户调用例程上的 `LD_PRELOAD` 干预的 `libcollector.so` 例程写入数据。每个这种干预例程部分填充数据记录，然后调用正常用户调用的例程，并当例程返回时填充剩下的数据记录，最后将记录写入数据文件。

所有数据文件都是内存映射的并以块来填充。记录总是以具有有效记录结构的方式填充，因此实验可以像写入时那样读取。缓冲管理策略设计用于最小化 LWP 间的争用和序列化。

归档子目录

每个实验具有一个归档子目录，其中包含了描述负载对象文件中引用的每个负载对象的二进制文件。这些文件是由运行在数据收集结尾的 `er_archive` 生成的。如果该进程异常终止，则 `er_archive` 不会被调用，这种情况下，首先在实验上调用时通过 `er_print` 或分析器将归档文件写入。

后续进程

后续进程将其实验写入创建进程的实验内的子目录。这些子目录用下划线、代码字母（f 代表派生而 x 代表执行）来命名，并且在刚刚创建的实验名称之后增加一个数字，指定后续的层次关系。例如，如果创建进程的实验名称是 test.1.er，则其第三个派生创建的子进程的实验是 test.1.er/_f3.er。如果该子进程执行了新映像，则相应实验名称为 test.1.er/_f3_x1.er。后续实验由与父实验相同的文件组成，不过这些文件不具有后续实验（所有后续用创建实验中的子目录表示），也不具有归档子目录（所有存档在创建实验时生成）。

动态函数

目标创建动态函数的实验在描述这些函数的负载对象文件中具有附加的记录，此外还具有一个附加文件 dyntext，该文件包含了动态函数的实际指令的副本。生成动态函数的注释反汇编码时需要该副本。

Java 实验

Java 实验在负载对象文件中也具有既是 JVM 因其内部目的而创建的动态函数也是目标 Java 方法的动态编译 (HotSpot) 版本的附加记录。

Java 实验在负载对象文件中也具有既是 JVM 因其内部目的而创建的动态函数也是目标 Java 方法的动态编译 (HotSpot) 版本的附加记录。

此外，Java 实验具有 JAVA_CLASSES 文件，包含了关于所有调用用户 Java 类的信息。

使用 JVMPI 代理记录 Java 堆跟踪数据和同步跟踪数据，该代理是 libcollector.so 的一部分，它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）的事件以及负载对象文件中 Java 编译的方法记录。

记录实验

记录实验有三种不同的方法：使用 collect 命令，使用 dbx 创建进程和使用 dbx 从运行的进程创建实验。

collect 实验

collect 用于记录实验时， collect 程序本身创建实验目录并设置 LD_PRELOAD 以确保 libcollector.so 被预装入目标的地址空间。然后该程序设置环境变量，通知 libcollector 实验名称、数据收集选项和最先执行的目标。

libcollector.so 负责写入所有实验文件。

dbx 实验，创建进程

当 dbx 用于在启用数据收集时启动进程，还创建了实验目录，并且确保了 libcollector.so 的预装入。该命令使进程在其第一个指令前的断点处中止，然后调用 libcollector 中的初始化例程启动数据收集。

Java 实验不能用 dbx 收集，因为 dbx 使用 JVMDI 代理进行调试，而该代理不能与数据收集所需的 JVMPI 代理共存。

dbx 实验，在运行的进程上

dbx 用于在运行的进程上启动实验时会创建实验目录，但不能使用 LD_PRELOAD。dbx 调用到 dlopen libcollector.so 目标的交互式函数，然后调用 libcollector.so 的初始化例程 （与创建进程时相同）。数据像收集实验中一样由 libcollector.so 写入。

因为进程启动时 libcollector.so 不在目标地址空间，所以取决于用户可调用函数（同步跟踪、堆跟踪、 MPI 跟踪）的任何数据收集都无法正常工作。通常，对基础函数的符号已经解决，因此插入不会发生。此外，以下后续进程也依赖于插入，并且对于 dbx 在运行的进程上创建的实验将无法正常工作。

如果 dbx 启动进程或使用 dbx 附加到运行的进程之前用户已显式 LD_PRELOAD' libcollector.so，则跟踪数据可以被收集。

解释性能度量

每个事件的数据包含了高分辨率的时间标记、线程 ID、 LWP ID 和处理器 ID。前三个可按照时间、线程或 LWP 过滤性能分析器中的度量。关于处理器 ID 的信息请参阅 getcpuid(2) 手册页。在 getcpuid 不可用的系统上，处理器 ID 是映射到 [未知] 的 -1。

除了通常的数据外，每个事件生成特定的原始数据，如以下几节所述。每节还讨论了从原始数据派生的度量准确性和数据收集对度量的影响。

基于时钟的分析

基于时钟的分析的特定事件数据由十个微态中任一分析间隔计数的数组组成，其中微态由每个 LWP 的内核维护。分析间隔结束时，每个 LWP 的微态计数按 1 递增并调度分析信号。数组仅在 CPU 中 LWP 处于用户模式时才被记录和复位。如果调度分析信号时

LWP 处于用户模式，则用户 CPU 状态的数组元素为 1，而所有其它状态的数组元素为 0。如果 LWP 不处于用户模式，则 LWP 下一次进入用户模式时该数据被记录，而且数组可以包含不同状态计数的累积。

调用栈与数据同时记录。如果 LWP 在分析间隔结束时未处于用户模式，那么只有 LWP 再次进入用户模式调用栈才会更改。因此调用栈总是在每个分析间隔结束时精确记录程序计数器的位置。

每个微态贡献的度量如表 5-2 中所示。

表 5-2 内核微态如何贡献给基值

内核微态	描述	度量名称
LMS_USER	在用户模式中运行	用户 CPU 时间
LMS_SYSTEM	在系统调用或缺页中运行	系统 CPU 时间
LMS_TRAP	在任何其它陷阱中运行	系统 CPU 时间
LMS_TFAULT	在用户文本缺页中休眠	文本缺页时间
LMS_DFAULT	在用户数据缺页中休眠	数据缺页时间
LMS_KFAULT	在内核缺页中休眠	其它等待时间
LMS_USER_LOCK	等待用户模式锁定的休眠	用户锁定时间
LMS_SLEEP	任何其它原因的休眠	其它等待时间
LMS_STOPPED	停止（/proc、作业控制或 lwp_stop）	其它等待时间
LMS_WAIT_CPU	等待 CPU	等待 CPU 时间

定时度量的准确性

定时数据是基于统计收集的，也因此会出现所有统计抽样方法的错误。在非常短暂的运行期间，只有少数分析包被记录，调用栈不能代表消耗大部分资源的程序部分。应该在足够长的时间运行足够次数的程序，累计感兴趣的函数或源代码行的数百个分析包。

除了统计抽样错误外，收集和归属数据的方法和程序通过系统进行的方法都会引起特定的错误。在定时度量中会出现不准确或失真的某些情况在下文中描述。

- LWP 创建时，记录第一个分析包之前所用时间比分析间隔短，但是整个分析间隔取决于第一个分析包中记录的微态。如果有多个创建的 LWP，则其中的错误会是分析间隔的几倍。
- LWP 被销毁时，记录最后的分析包会消耗一些时间。如果有多个销毁的 LWP，则其中的错误会是分析间隔的几倍。
- LWP 重新调度可在分析间隔期间发生。因此， LWP 的记录状态不可以代表消耗大部分分析间隔的微态。有多个 LWP 而不是多个处理器运行它们时，错误可能更大。

- 可以让程序以与系统时钟关联的方式运行。这种情况下， LWP 处于可能表示一小部分所用时间的状态时分析间隔始终会过期，而且为程序特定部分记录的调用栈有过多的表示。在多处理器系统上，分析信号引入关联是可能的：记录微态时，处理器运行程序的 LWP 时分析信号中断的处理器可能处于陷阱 CPU 微态。
- 分析间隔过期时内核记录微态值。在系统负载大的时候，该值可能无法表示进程的真实状态。这种情况可能会导致对陷阱 CPU 或等待 CPU 微态的过多计数。
- 线程库处于临界段时有时会丢弃分析信号，从而导致对定时度量的低估。
- 系统时钟与外部源同步时，记录在分析包中的时间标记不反映分析间隔但包括了对时钟所作的任何调整。时钟调整可以使分析包丢失。包含的时间周期通常是几秒，并且以某个增量进行调整。

除刚刚描述的不准确外，时间度量会因收集数据的进程而失真。从不显示记录分析包所用的时间，因为该记录通过分析信号来初始化。（这是关联的另一个实例）。不管微态如何记录，记录进程中所用的用户 CPU 时间都会被发布。结果是对 [用户 CPU 时间] 度量的过少计数和对其它度量的过多计数。记录数据所用的时间总和通常少于缺省分析间隔 CPU 时间的百分之一。

定时度量的比较

如果将从基于时钟实验中完成的分析获得的定时度量与用其它方法获得的时间相比较，则应该注意以下问题。

对于单线程应用程序，如果与相同进程的 gethrtime(3C) 返回的值比较，进程记录的 LWP 时间总和通常精确到千分之几。该 CPU 时间与相同进程 gethrvtime(3C) 返回的值相差几个百分点。在重负载的情况下，这种差异可能更为明显。不过，该 CPU 时间差异不表示系统失真，并且从不同函数、源代码行等报告的相对时间也不会有大的失真。

对于使用无界线程的多线程应用程序，gethrvtime() 返回值中的差异可能毫无意义。这是因为 gethrvtime() 返回 LWP 的值，并且线程可以因不同的 LWP 而更改。

性能分析器中报告的 LWP 时间可以与 vmstat 报告的时间有很大区别，因为 vmstat 报告所有 CPU 总计的时间。如果目标进程比运行该进程的多 CPU 系统具有更多的 LWP 时，则性能分析器显示出比 vmstat 报告更多的等待时间。

出现在性能分析器 [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使用报告，因此微态中所用时间的记录具有很高的准确性。更多信息请参阅 proc(4) 手册页。可以把这些计时与将程序表示为一个整体的 <Total> 函数的度量做比较，目的是获取累计定时度量的准确性指示。不过，显示在 [统计] 标签中的值可以包括在 <Total> 的定时度量值中不包含的其它基值。这些基值来自以下源码：

- 由未分析的系统创建的线程。 Solaris 操作系统版本 7 和 8 中的标准线程库创建未分析的系统线程。这些线程大部分时间休眠，休眠时间在 [统计数据] 标签中显示为 [其它等待] 时间。
- 暂停数据收集所在的时间周期。

同步等待跟踪

通过跟踪对线程库 `libthread.so` 中的函数或实时扩展库 `librt.so` 的调用，收集器收集同步延迟事件。事件特定的数据由请求和授权的高分辨率时间标记 （跟踪的调用开头和结尾）和同步对象 （例如，请求的互斥锁）的地址组成。线程和 LWP ID 是记录数据时的 ID。等待时间就是请求时间和授权时间的差。只有等待时间超过指定域值的事件才被记录。同步等待跟踪数据记录在授权时的实验中。

如果程序使用有界线程，则引起延迟的事件完成之前 LWP（在其上调度等待线程）不能执行任何其它工作。等待所用的时间同时显示为 [同步等待时间] 和 [用户锁定时间]。[用户锁定时间] 可以大于 [同步等待时间]，因为同步延迟阈值限制了短期延迟。

如果程序使用无界线程，则 LWP （在该 LWP 上调度等待线程）可以在其上具有其它调度的线程调度并继续执行用户工作。如果某些线程正等待同步事件时所有 LWP 都保持忙，则 [用户锁定时间] 为零。不过， [同步等待时间] 不为零，因其与特定的线程关联，而不是与正运行线程的 LWP 关联。

因数据收集的开销使等待时间失真。该开销与收集的事件数量成比例。通过增加记录事件的阈值，开销中所用的等待时间部分可以被最小化。

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。该值可以比计数器设置的溢出值大，因为处理器在溢出和事件的记录之间执行某些指令。对循环和指令计数器来说尤其如此，该计数器的递增速度比诸如浮点运算或缓冲缺少的计数器快。记录事件中的延迟也意味着用调用栈记录的程序计数器地址没有精确对应到溢出事件。

更多信息请参阅第 127 页的 “硬件计数器溢出的属性”。另请参阅第 104 页的 “陷阱”的讨论。陷阱和陷阱处理程序会使得报告的 [用户 CPU] 时间与循环计数器报告的时间显著不同。

收集的数据总和取决于溢出值。选择过小的值可以导致以下结果。

- 收集数据所用的时间总和可以是程序执行时间的主要部分。运行的收集可能消耗大部分时间来处理溢出和写数据，而不是消耗大部分时间运行程序。
- 计数的主要部分可以来自收集进程。这些计数被归属到收集器函数 `collector_record_counters`。如果看到该函数的高计数，则溢出值过小。
- 数据的收集会改变程序的行为。例如，如果正收集关于缓冲缺少的数据，则大多数缺少可能来自刷新收集器指令，而分析数据来自缓冲，并将其替换为程序指令和数据。程序看上去具有许多缓冲缺少，但没有数据收集时实际上只有很少的缓冲缺少。

选择过大的值会导致良好统计过少溢出。最后溢出之后产生的计数被归属到收集器函数 `collector_final_counters`。如果在该函数中看到主要部分的计数，则溢出值过大。

堆跟踪

收集器通过干预内存分配和释放函数 malloc、realloc、memalign 和 free 从而记录对这些函数调用的跟踪数据。如果程序分配内存时忽视这些函数，则跟踪数据不被记录。不记录 Java 内存管理的跟踪数据，该管理使用了不同的机制。

被跟踪的函数可以从许多库中的任意一个装入。性能分析器中看到的数据可能取决于装入给定函数的库。

如果程序在短时间内生成了对跟踪函数的大量调用，则执行程序的时间会显著延长。额外的时间用于记录跟踪数据。

MPI 跟踪

MPI 跟踪记录了关于对 MPI 库函数调用的信息。事件特定的数据由请求和授权的高分辨率时间标记 （跟踪的调用的开头和结尾）、发送和接收的操作数以及发送或接收的字节数。跟踪通过干预对 MPI 库的调用来执行。干预的函数不具有关于数据传送优化和传送错误的详细信息，因此出现的信息表示了数据传送的简单模型，如以下几段所述。

接收的字节数是对 MPI 函数调用中定义的缓冲长度。实际接收的字节数对干预的函数是不可用的。

某些 [全局通信] 函数具有单一起始点或单一的接收进程 （称为根）。这些函数的计算按以下步骤执行：

- 根将数据发送到所有进程，包括本身。
- 根从所有进程接收数据，包括本身。
- 每个进程与其它进程通信，包括本身。

以下示例说明了计算的步骤。在这些示例中， G 是组的大小。

对于到 MPI_Bcast() 的调用，

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

对于到 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

对于到 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

调用栈是表示程序内指令的一系列程序计数器地址 (PC)。首个 PC 名为分支 PC，位于堆栈的底部，是下一条要执行的指令的地址。下一个 PC 是对包含分支 PC 的函数调用的地址；到达堆栈的顶部之前，下一个 PC 是对该函数调用的地址。每个这种地址被称为返回地址。记录调用栈的进程包含了从程序堆栈获取返回地址，这指得是解除堆栈。关于解除失败的信息，请参阅第 112 页的 “不完整的堆栈解除”。

调用栈中的分支 PC 用于将排除度量从性能数据分配到该 PC 所在的函数。堆栈上的每个 PC （包括分支 PC）用于将包括度量分配到该度量所在的函数。

大部分时间，记录调用栈中的 PC 以一种自然的方式与程序的源代码中显示的函数对应，而报告度量的性能分析器直接与这些函数对应。不过，有时程序的实际执行并不与程序如何执行的简单初始化模型对应，而且性能分析器的报告度量可能会引起误解。与这种情况相关的信息请参阅第 113 页的 “将地址映射到程序结构”。

单线程执行和函数调用

程序执行最简单的情况是执行调用自身负载对象内函数的单线程程序。

程序装入内存并开始执行时，会建立包括要执行的初始地址、初始寄存器集和堆栈 （用于临时数据和跟踪函数如何彼此调用的内存区域）的上下文。初始地址始终位于每个可执行文件中建立的函数 `_start()` 的开头。

程序运行后，遇到分支指令之前指令按顺序执行，在其它情况中该分支语句可能表示函数调用或条件语句。在分支点上，控制被传输到分支目标给出的地址，然后从该地址继续执行。（通常分支后的下一条指令已提交准备执行：该指令称为分支延迟槽指令。不过，某些分支指令取消了对分支延迟槽指令的执行。）

表示调用的指令序列被执行时，返回地址被放置到寄存器中，并且继续执行正被调用的第一条指令。

在大多数情况下，在调用函数的前几个指令中的某些地方，新的帧 （用于存储关于函数信息的内存区域）会放入堆栈，而返回地址被放置到该帧中。调用的函数本身调用另一个函数时，可以使用返回地址所用的寄存器。函数准备返回时，从堆栈弹出它的帧，并控制返回到调用函数的地址。

共享对象间的函数调用

一个共享对象中的函数调用另一个共享对象中的函数时，该执行比对程序内函数简单调用中的执行复杂。每个共享对象包含程序链接表 (PLT)，该表包含的条目有从该表引用的共享对象外部的每个函数。 PLT 中每个外部函数的初始地址实际上是动态链接程序 `ld.so` 内的地址。第一次调用这个函数时，控制被传输到动态链接程序，该程序解决了对真实外部函数的调用并为后续调用的 PLT 地址打补丁。

如果在执行三个 PLT 指令的任一个期间发生分析事件， PLT PC 会被删除，并且排除时间归属到调用指令。如果首次通过 PLT 条目调用期间发生分析事件，但是分支 PC 不是其中一个 PLT 指令，则从 PLT 引起的任何 PC 和 ld.so 中的代码通过对聚集包括时间的人造函数 `@plt` 的调用来替换。每个共享对象有一个这种人造函数。如果程序使用 `LD_AUDIT` 接口，则 PLT 条目可能永远不会打补丁，且来自 `@plt` 的非分支 PC 会更频繁的发生。

信号

信号发送到进程时，会发生多种寄存器和堆栈操作，使得它看上去好像发送信号时的分支 PC 是对系统函数 `sigacthandler()` 调用的返回地址。 `sigacthandler()` 像任何函数调用另一个函数一样调用用户指定的信号处理程序。

性能分析器像处理普通帧一样处理从信号传递产生的帧。信号传递时的用户代码与调用系统函数 `sigacthandler()` 时显示的相同，而且它与调用用户信号处理程序一样依次显示。 `sigacthandler()` 和任何用户信号处理程序的包括度量和调用的任何其它函数显示为中断函数的包括度量。

收集器在 `sigaction()` 上插入以确保它的处理程序是主要的处理程序，收集时钟数据时是 `SIGPROF` 信号的处理程序，而收集硬件计数器数据时是 `SIGEMT` 信号的处理程序。

陷阱

可以通过指令或硬件发出陷阱，并且通过陷阱处理程序捕获。系统陷阱是从指令初始化的陷阱并会陷入内核。例如，所有系统调用均使用陷阱指令实现。硬件陷阱的某些示例来源于不能完成指令 （例如 UltraSPARC® III 平台上的 `fitos` 指令）时或指令在硬件中没有实现时的浮点单元。

陷阱发生时 LWP 进入系统模式。微态通常从 [用户 CPU] 状态切换到 [陷阱] 状态再到 [系统] 状态。取决于微态切换的位置，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。该时间被归属到初始化陷阱的用户代码中的指令 （或系统调用）。

对于某些系统调用，关键要考虑的是提供尽可能有效的调用处理。这些调用生成的陷阱称为 *fast 陷阱*。生成 fast 陷阱的系统函数有 `gethrtime` 和 `gethrvtime`。在这些函数中，由于包含的开销所以微态不会切换。

在其它情况下，也需要重点考虑提供尽可能有效的陷阱处理。这些示例中的某些是 TLB（旁路转换缓冲）缺少和寄存器窗口溢出和填充，其中不切换微态。

在这两种情况下，所用的时间记录为 [用户 CPU] 时间。不过，因为模式已切换到系统模式所以硬件计数器被关闭。通过掌握 [用户 CPU] 时间和 [循环] 时间之间的差异，可以估算处理这些陷阱所用的时间，尤其是记录在相同实验中的时间。

有一种陷阱处理程序切换回用户模式的情况，这是 8 字节整数未对齐内存引用陷阱，与 Fortran 中的 4 字节边界对齐。陷阱处理程序的帧显示在堆栈上，而对处理程序的调用可以显示在性能分析器中，归属到整数装入或存储指令。

指令陷入内核后，陷阱指令后的指令要消耗长时间来显示，因为内核执行陷阱指令完成之前该指令不能启动。

尾调用优化

无论何时特定函数最后调用另一个函数，编译器都可以执行一个特定的优化。除了生成新的帧外，被调用方重用来自调用方的帧，并且被调用方的返回地址从调用方复制。该优化的动机是减少堆栈的大小，（在 SPARC® 平台上时）减少使用寄存器窗口。

假设程序源码中的调用序列如下所示：

```
A -> B -> C -> D
```

B 和 C 经过尾调用优化后，调用栈看上去如同函数 A 直接调用函数 B、 C 和 D。

```
A -> B
A -> C
A -> D
```

如此，调用树被展开。代码用 -g 选项编译时，尾调用优化仅发生在编译器优化级别为 4 或更高。代码不用 -g 选项编译时，尾调用优化发生在编译器优化级别为 2 或更高。

显式多线程

简单程序在单一 LWP （轻量进程）上的单线程中执行。多线程的可执行文件生成对线程创建函数的调用，执行的目标函数被传递到该函数。目标存在时，线程由线程库销毁。新创建的线程在名为 _thread_start() 的函数开始执行，该函数调用线程创建调用中传递的函数。对于该线程执行时包含目标的任何调用栈，堆栈的顶部是 _thread_start()，对线程创建函数的调用方没有任何连接。因此与创建的线程关联的包括度量仅传送到 _thread_start() 和 <Total> 函数。

除创建线程外，线程库还创建 LWP 来执行线程。线程可以用有界线程 （其中每个线程都以特定的 LWP 为边界）或无界线程 （其中每个线程可以在不同时间不同的 LWP 上调度）执行。

- 如果使用了有界线程，则线程库为每个线程创建一个 LWP。

- 如果使用了无界线程，则线程库决定创建多少 LWP 使运行有效，以及哪些 LWP 用于调度线程。以后如果需要，线程库可以创建更多 LWP。无界线程既不是 Solaris 9 操作系统的一部分，也不是 Solaris 8 操作系统中可选线程库的一部分。

  做为调度无界线程的示例，线程位于诸如 `mutex_lock` 的同步屏障时，线程库可以在执行第一个线程的 LWP 上调度不同的线程。等待通过屏障处的线程锁定所用的时间显示在 [同步等待时间] 度量中，但是因为 LWP 不是空闲的，所以该时间不增加到 [用户锁定时间] 度量中。

除用户线程外， Solaris 操作系统版本 7 和 8 中的标准线程库创建了一些线程用于执行信号处理和其它任务。如果程序使用有界线程，则也为这些线程创建附加的 LWP。不收集或显示这些线程的性能数据，它们的大部分时间用于休眠。不过，这些线程中所用的时间包括在进程统计中和样本数据中记录的时间中。Solaris 9 操作系统中的线程库和 Solaris 8 操作系统的替换线程库不创建这些额外的线程。

基于 Java 技术软件执行的概述

对于典型的开发人员，基于 Java™ 技术的应用程序运行时类似于任何其它程序。该应用程序开始时有一个主入口点，通常名为 *class*.main，就像 C 或 C++ 应用程序一样可以调用其它方法。

对于操作系统，使用 Java 编程语言 （纯 Java 代码或与 C/C++ 混合）编写的应用程序运行时如同实例化 [1]Java 虚拟机 (JVM) 的进程。JVM™ 软件从 C++ 源码编译，并在调用 main 等的 at _start 处开始执行。该软件从 .class 和 / 或 .jar 文件读取字节码并执行在该程序中指定的操作。可以指定的操作是共享对象的动态装入，而到不同函数或方法的调用包含在该对象内。

执行基于 Java 技术的应用程序期间，大多数方法用 JVM 软件解释；这些方法在本文档中是指*解释的方法*。其它方法可以通过 Java HotSpot™ 虚拟机动态编译，是指*编译的方法*。动态编译的方法被装入到应用程序的数据空间，并且可以在以后的某个时刻及时卸载。对于任何特定的方法，会有一个解释的版本，也可能有一个或多个编译的版本。用 Java 编程语言编写的代码也可以直接调用 C、C++ 或本机编译的 (SBA SPARC Bytecode Accelerator) Java 编写的本机编译的代码中；这种调用的目标指得是*本机方法*。

JVM 软件可以处理传统语言编写的应用程序不能处理的大量工作。启动时，该软件在其数据空间创建了大量动态生成代码的区域。其中一个是用于处理应用程序字节码方法的实际解释程序代码。

解释执行期间， Java HotSpot 虚拟机监控性能，并可决定使用已解释的一个或多个方法生成它们的机器码，并执行更有效的机器码版本而不是解释原始代码。生成的机器码也位于进程的数据空间。此外，数据空间中生成其它代码来执行解释和编译代码之间的转换。

1. “Java 虚拟机”和 “JVM”术语表示 Java™ 平台的虚拟机。

用 Java 编程语言编写的应用程序本来就是多线程的，并且用户程序中的每个线程具有一个 JVM 软件线程，此外还具有多个用于信号处理、内存管理和 Java HotSpot 虚拟机编译的内务处理线程。根据所用的 libthread.so 版本，线程和 LWP 间可能一一对应，或存在更为复杂的关系。但是对于任何版本的库，线程会被随时取消调度或调度到 LWP 上。线程未调度到 LWP 上时不收集线程的数据。

通过记录进程的每个 LWP 生命期间的事件，性能工具收集它们的数据和事件期间的调用栈。在执行任何应用程序 （基于 Java 技术或其它）的任何时刻，调用栈表示了程序处于执行中的哪个位置，以及如何到达该位置。区别基于 Java 技术的应用程序混合模型与传统 C、 C++ 和 Fortran 应用程序的一个重要方法是目标运行的任何时刻有两个有意义的调用栈 £½Java 调用栈和机器调用栈。这两个堆栈会被收集并彼此关联。

Java 处理表示

对于用 Java 编程语言编写的应用程序，显示性能数据有三种表示：Java 表示、专家 Java 表示和机器表示。Java 表示缺省显示在数据支持该表示的位置。以下部分总结了这三种表示之间的主要差异。

Java 表示

Java 表示按名称显示编译和解释的 Java 方法，并以自然的形式显示本机方法。执行期间，可能有很多特定 Java 方法执行的实例：解释版本和 （可能）一个或多个编译版本。在 Java 表示中所有方法被累计显示为单一的方法。在分析器中该模式被缺省选定。

专家 Java 表示

[专家 Java 表示] 类似于 [Java 表示]，不同之处为 [Java 表示] 中禁用的某些 JVM 内部详细信息显示在 [专家 Java 表示] 中。

机器表示

[机器表示] 显示的函数来自 JVM 本身，不是来自正被 JVM 解释的应用程序。该表示还显示了所有编译和本机的方法。该机器表示看上去与用传统语言编写的应用程序相同。调用栈显示 JVM 帧、本机帧和编译方法的帧。某些 JVM 帧表示了在解释的 Java、编译的 Java 和本机代码之间的转换代码。

并行执行和编译器生成的主体函数

如果代码包含 Sun、Cray 或 OpenMP 并行指令，则可以为并行执行来编译。OpenMP 是 Sun 编译器和工具可用的特性。请参阅 《OpenMP API 用户指南》和 《Fortran 编程指南》及 《C 用户指南》的相关节，或访问定义 OpenMP 标准的 web 站点 http://www.openmp.org。

为并行执行编译循环或其它并行构造时，编译器生成的代码通过多线程来执行，通过微任务化库调整。 Sun 编译器的并行性请遵循下述步骤。

主体函数的生成

编译器遇到并行构造时，通过将该构造的主体放入独立的 *body function* 中并将该构造替换为对微任务化库函数的调用来设置并行执行的代码。微任务化库函数负责分发线程来执行主体函数。主体函数的地址做为参数传递到微任务化库函数。

如果并行构造用以下列表中的其中一个指令限定，则该构造被替换为对微任务化库函数 `__mt_MasterFunction_()` 的调用。

- Sun Fortran 指令 `!$par doall`
- Cray Fortran 指令 `c$mic doall`
- Fortran OpenMP `!$omp PARALLEL`, `!$omp PARALLEL DO` 或 `!$omp PARALLEL SECTIONS` 指令
- C 或 C++ OpenMP `#pragma omp parallel`， `#pragma omp parallel for` 或 `#pragma omp parallel sections` 指令

编译器自动并行的循环也被对 `__mt_MasterFunction_()` 的调用替换。

如果 OpenMP 并行构造包含一个或多个共享任务的 do、for 或段指令，则每个共享任务构造被替换为对微任务化库函数 `__mt_Worksharing_()` 的调用，并且为每个构造创建一个新的主体函数。

编译器将名称分配到主体函数，这些函数为并行构造的类型、提取构造的函数名称、初始源码中构造的起始行号和并行构造的序列号编码。这些粉碎名称因微任务化库的发行版本而不同。

并行执行序列

程序开始执行时只有一个线程，即主线程。程序首次调用 `__mt_MasterFunction_()` 时，该函数调用 Solaris 线程库函数 `thr_create()` 来创建工作线程。每个工作线程执行微任务化库函数 `__mt_SlaveFunction_()`，该函数做为参数传递到 `thr_create()`。

除工作线程外， Solaris 操作系统版本 7 和 8 中的标准线程库创建了一些线程用于执行信号处理和其它任务。不收集这些线程的性能数据，线程的大部分时间用于休眠。不过，这些线程中所用的时间包括在进程统计中和样本数据中记录的时间。 Solaris 9 操作系统中的线程库和 Solaris 8 操作系统的替换线程库不创建这些额外的线程。

线程创建之后， `__mt_MasterFunction_()` 管理主线程和工作线程中可用工作的分配。如果工作不可用，则 `__mt_SlaveFunction_()` 调用工作线程等待可用工作的 `__mt_WaitForWork_()`。工作一成为可用，该线程就返回到 `__mt_SlaveFunction_()`。

工作可用后，每个线程执行对 `__mt_run_my_job_()` 的调用，关于主体函数的信息被传递到该函数。此处的执行序列取决于主体函数是从并行段指令、并行 do （或并行）指令、并行共享任务指令还是并行指令生成。

- 在并行段的情况中， `__mt_run_my_job_()` 直接调用主体函数。
- 在并行 do 或 for 的情况下，`__mt_run_my_job_()` 调用取决于循环性质的其它函数，并且其它函数调用主体函数。
- 在并行情况下， `__mt_run_my_job_()` 直接调用主体函数，并且所有线程遇到对 `__mt_WorkSharing_()` 的调用之前会执行主体函数中的代码。本函数中有另外一个对 `__mt_run_my_job_()` 的调用，该函数直接在共享任务段的情况下调用共享任务主体函数，或在共享任务 do 或 for 的情况下调用其它库函数。如果 `nowait` 在共享任务中指定，则每个线程返回到并行主体函数并继续执行。否则，这些线程返回到 `__mt_WorkSharing_()`，该函数在继续之前调用 `__mt_EndOfTaskBarrier_()` 使线程同步。

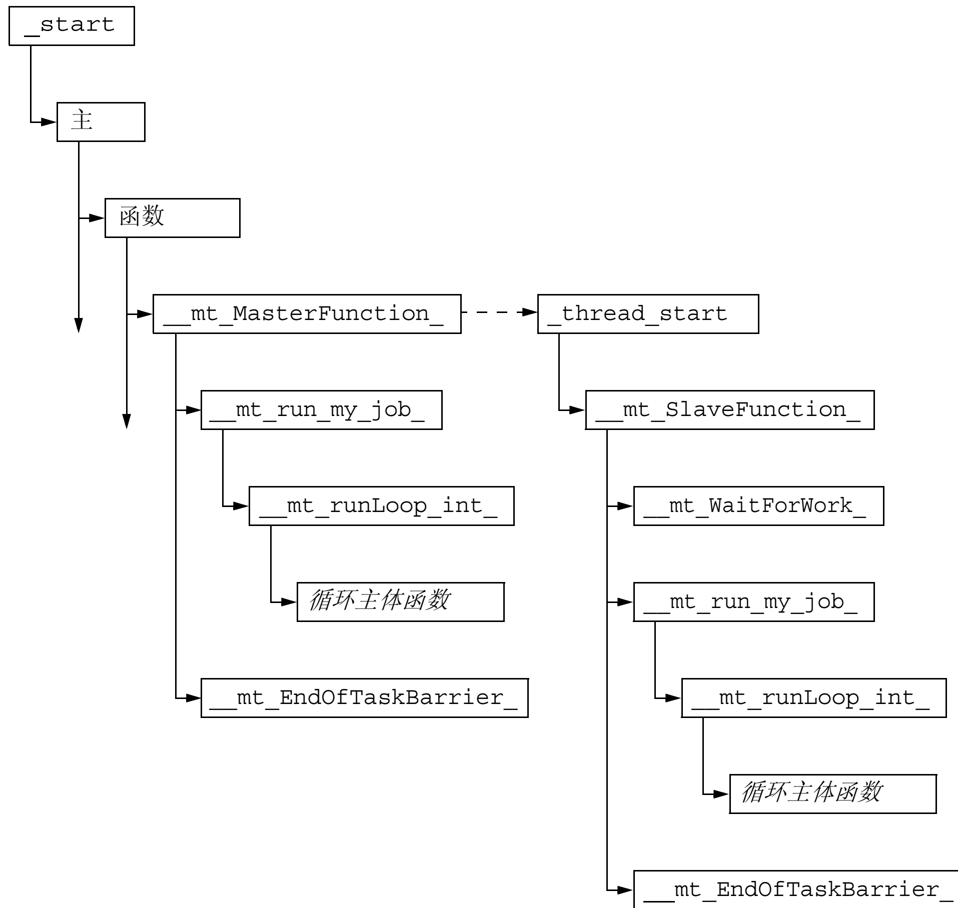


图 5-1 包含并行 Do 或并行 For 构造的多线程程序调用树示意图

所有并行工作完成后，线程返回到 `__mt_MasterFunction_()` 或 `__mt_SlaveFunction_()` 并调用 `__mt_EndOfTaskBarrier_()` 来执行包含在并行构造终端中的任何同步工作。工作线程然后再调用 `__mt_WaitForWork_()`，同时主线程继续在串行区域执行。

这里描述的调用序列不仅适用于并行运行的程序，也适用于为并行性编译的程序但运行在单 CPU 机器上或仅使用一个 LWP 的多处理器机器上。

简单并行 do 构造的调用序列在图 5-1 中描述。工作线程的调用栈以线程库函数 `_thread_start()` 开始，该函数实际上调用 `__mt_SlaveFunction_()`。虚线箭头表示因从 `__mt_MasterFunction_()` 到 `thr_create()` 的调用而产生的线程开始。连续的箭头表示有一些不在这里表示的其它函数调用。

有一个共享任务 do 构造的并行区域的调用序列在图 5-2 中描述。__mt_run_my_job_() 的调用方不是 __mt_MasterFunction_() 就是 __mt_SlaveFunction_()。整个图表可以替换图 5-1 中对 __mt_run_my_job_() 的调用。

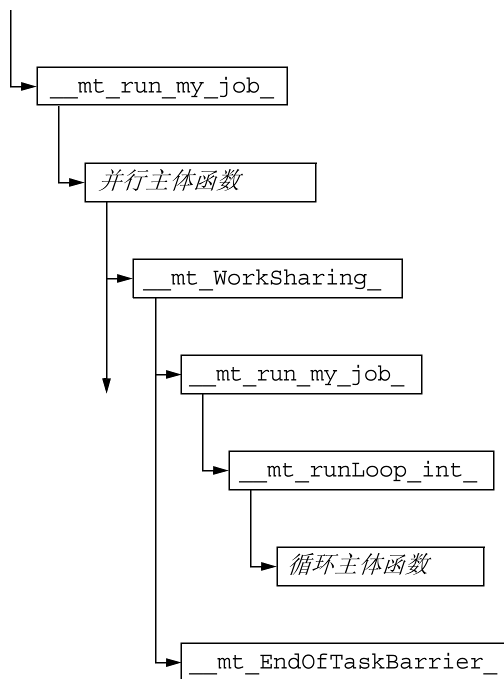


图 5-2 带有共享任务 Do 或共享任务 For 构造的并行区域调用树示意图

在这些调用序列中，所有编译器生成的主体函数从微任务化库中的相同函数调用，这使得将主体函数的度量与初始用户函数关联起来有些困难。性能分析器从初始用户函数插入一个到主体函数的输入调用，而微任务化库将输入调用从主体函数插入到屏障函数 __mt_EndOfTaskBarrier_()。取决于同步的度量因此被归属到主体函数，而主体函数的度量被归属到最初的函数。使用这些插入，来自主体函数的包括度量直接传送到初始函数，而不经过微任务化库函数。这些输入调用的副作用是主体函数同时显示为初始用户函数和微任务化函数的被调用方。此外，用户函数像其调用方一样显示具有微任务化库函数，并且可以在调用自身时显示。通过递归函数调用所用的机制可以避免对包括度量的重复计数 （请参阅第 37 页的 “递归如何影响函数级度量”）。

为了在新工作到达 （即主线程到达新的并行构造）时减少延迟，工作线程通常使用处于 __mt_WaitForWork_() 时的 CPU 时间。这称为忙等待。不过，可以设置环境变量来指定休眠等待，在性能分析器中显示为 [其它等待] 时间而不是 [用户 CPU] 时间。工作线程消耗时间用于等待工作和想要重新设计程序来减少等待的位置通常有两种情况：

- 主线程在串行区域中执行时且工作线程没有任何任务可做时
- 工作负载不对称时和其它线程仍然执行时某些线程已经完成且正在等待时

缺省情况下，微任务化库使用对 LWP 有界的线程。通过将环境变量 `MT_BIND_LWP` 设置到 `FALSE` 可以覆盖 Solaris 操作系统版本 7 和 8 中的该缺省设置。

注 – 多进程的分配进程是独立实现的并且可能因不同的发行版本而更改。

不完整的堆栈解除

堆栈解除会由于以下原因而失败：

- 如果堆栈已被用户代码破坏，则可能在程序或数据收集代码中发生信息转储，这取决于堆栈如何被破坏。
- 如果用户代码不遵循函数调用的标准 ABI 惯例。具体来说，在 SPARC 平台上，如果执行保存指令之前返回寄存器 %o7 改变。

  在任何平台上，手写汇编程序代码可能违反惯例。
- 在 Intel 平台上，如果 C 或 Fortran 代码以高优化级别编译，这意味着它不需要帧指针来帮助解除。
- 在 Intel 平台上，如果 C++ 代码用带有 `-noex` 或 `-features=no%except` 选项的任何优化级别来编译。
- 如果被调用方的帧从堆栈弹出之后但在函数返回之前，分支 PC 位于函数中。
- 如果调用栈包含的帧超过 250，则收集器没有足够的空间来完全解除调用栈。这种情况下，从 `_start` 到调用栈某些点的函数的 PC 不在实验中记录，并且 `<Total>` 显示为记录 PC 的最后一个函数的调用方。

中间文件

如果使用 `-E` 或 `-P` 编译器选项生成了中间文件，则性能分析器使用注释源代码的中间文件，而不使用初始源文件。用 `-E` 生成的 `#line` 指令会在将度量分配到源代码行时产生问题。

如果从不具有引用行号的函数到编译生成该函数的源文件存在指令，则在注释源码中显示以下代码行：

function_name -- < 没有行号的指令 >

在以下几种情况中可以没有行号：

- 编译时未指定 `-g` 选项。
- 编译后调试信息被剥离，或者包含该信息的可执行文件或对象文件被移动、删除或之后被修改。
- 该函数包含从 #include 文件生成的代码，不包含从初始源文件生成的代码。
- 在高优化级别上，如果代码从不同文件中的函数内联。

- 源文件具有引用到某些其它文件的 #line 指令；用 `-E` 选项编译的情况下仅编译生成的 `.i` 文件。用 `-P` 标志编译时，也会出现类似的结果。
- 找不到读取行号信息的目标文件。
- 该文件编译时没有使用 -g 标志，或者该文件已被剥离。
- 使用的编译器生成了不完整的行号表。

将地址映射到程序结构

调用栈被处理成 PC 值后，性能分析器将这些 PC 映射到程序中的共享对象、函数、源代码行和反汇编代码行 （指令）。本节描述了这些映射。

进程映像

程序运行时，从该程序的可执行文件实例化进程。该进程在其地址空间中具有许多区域，某些区域是文本，表示可执行文件的指令；而某些区域是非正常执行的数据。与记录在调用栈中的相同， PC 通常对应于其中一个程序文本段内的地址。

进程中第一个文本部分从可执行文件本身派生。其它文本部分对应于进程启动时或进程动态装入时与可执行文件一起装入的共享对象。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而解决。可执行文件和共享对象非常类似，并且可以统称为负载对象。

因为程序执行过程中共享对象可以被装入和卸载，所以给定的 PC 可能对应于运行期间不同时间的不同函数。此外，如果共享对象被卸载并在不同的地址重载，则不同的 PC 可能对应于相同的函数。

负载对象和函数

每个负载对象，无论是可执行文件还是共享对象都包含带有编译器生成指令的文本段，数据的数据段和各种符号表。所有负载对象必须包含 ELF 符号表，该表给出了这个对象中所有已知函数的名称和地址。用 `-g` 选项编译的负载对象包含了附加的符号信息，该信息可以增加 ELF 符号表并提供关于非全局函数的信息，关于函数起源的对象模块附加信息，以及将地址相关到源代码行的行号信息。

术语 *function* 用于描述代表源代码中所述高级操作的指令集。该术语涵盖了 Fortran 中所用的子例程， C++ 和 Java 中所用的方法等等。函数在源代码中描述地很清晰，并且通常函数的名称显示在表示一组地址的符号表中；如果程序计数器位于该组内，则程序在该函数内执行。

原则上，负载对象文本段内的任何地址都可以映射到函数。调用栈上的分支 PC 和所有其它 PC 都使用完全相同的映射。大多数函数直接对应到程序的源模型。其它未对应到源模型的函数在以下部分描述。

有别名的函数

一般来说，函数被定义为全局函数，意味着从程序中的每个地方都可以知道这些函数名称。在可执行文件内全局函数的名称必须是唯一的。如果地址空间内多个全局函数只有一个给定的名称，则运行时链接程序解决对这些函数之一的所有引用。其它的函数从不执行，并且不显示在函数列表中。在 [汇总] 标签中，您可以看到包含选定函数的共享对象和对象模块。

在不同的情况下，函数可以有多个不同的名称。这种情况最常见的示例是使用相同代码段的所谓弱符号和强符号。除前导下划线外，强名称通常与对应的弱名称相同。线程库中的许多函数也具有 pthread 和 Solaris 线程的替代名称，以及强弱名称和替代的内部符号。在这种情况下，只有一个名称用于性能分析器的函数列表。选择的名称是以字母顺序在给定地址上的最后符号。该选择最有可能对应于用户使用的名称。在 [汇总] 标签中显示选定函数的所有别名。

非唯一函数名称

有别名的函数反映相同代码段的多个名称时，会有多个代码段具有相同名称的情况：

- 有时，由于模块性的原因，函数被定义为静态，这意味着只有在程序的某些部分（通常是某一编译的对象模块）中才能知道这些函数的名称。在这些情况下，引用到程序完全不同部分相同名称的多个函数显示在性能分析器中。在 [汇总] 标签中，给出每个这些函数的对象模块名称用以区分这些函数。此外，选择这些函数中的任一个可以用于显示源码、反汇编码和特定函数的调用方和被调用方。
- 有时程序使用库中具有函数的弱名称的包装器或插入函数，并代替对该库函数的调用。某些包装器函数调用库中的初始函数，这种情况下名称的两种实例都显示在性能分析器函数列表中。这些函数来自不同的共享对象和不同的对象模块，并且可以用这种方法互相区分这些函数。收集器包装某些库函数，而且包装器函数和实函数都可以显示在性能分析器中。

源于剥离共享库的静态函数

静态函数通常在库内使用，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，静态函数的名称从符号表删除。这种情况下，性能分析器在包含剥离静态函数的库中生成每个文本区域的人工名称。该名称的形式是 `<static>@0x12345`，其中后跟 `@` 符号的字符串是库内文本区域的偏移。性能分析器不能区分连续的剥离静态函数和单一的这种函数，因此会有两个或两个以上的这种函数与其接合的度量一起显示。

剥离静态函数显示为从正确的调用方调用，只有静态函数的 PC 是显示在静态函数中保存指令后的分支 PC 时例外。如果没有符号信息，性能分析器就不会知道保存地址，也不能告知是否像调用方一样使用返回寄存器，它会一直忽略返回寄存器。因为多个函数可以被接合成单一的 `<static>@0x12345` 函数，所以真实调用方或被调用方不可能与相邻的函数区分开来。

Fortran 替代的入口点

Fortran 为单一的代码提供了一种具有多个入口点的方法，允许调用方调用到函数的中间。这种代码被编译时，由主入口点的序言、替代入口点的前导部分和函数的代码主体组成。每个前导部分设置函数最后返回的堆栈，并转移或转向到代码的主体。

每个入口点的前导部分代码始终与具有入口点名称的文本区域对应，但是子例程主体的代码仅接收其中一个可能的入口点名称。不同的编译器接收不同的名称。

这些前导部分很少占用任何大量时间，与入口点对应而与子例程主体关联的函数不对应的这些函数很少显示在性能分析器中。在带有替代入口点的 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导部分中具有 PC，而且只有与主体关联的名称才做为被调用方显示。同样，子例程的所有调用显示为从与子例程主体关联的名称生成。

克隆的函数

编译器有能力识别对可以执行额外优化的函数的调用。这些调用的其中一个示例是对某些参数是常量的函数的调用。编译器标识其可以优化的特定调用时，会创建名为克隆的函数副本，并生成优化的代码。克隆函数名称是标识特定调用的粉碎名称。分析器不裁减该名称，并在函数列表中单独显示克隆函数的每个实例。因为每个克隆的函数具有不同的指令集合，所以注释反汇编码列表单独显示克隆的函数。因为每个克隆的函数具有相同的源代码，所以注释源码列表汇总函数所有副本的数据。

内联函数

内联函数是编译器生成的指令被插入该函数的调用点而不是实际调用的函数。有两种类型的内联，都可以改善性能并影响性能分析器。

- C++ 内联函数定义。这种情况下内联的理论基础是调用函数所用的代价比完成内联函数的代价要大的多，因此最好只是将函数的代码插入调用点，而不是设置调用。一般来说，访问函数被定义为要被内联，因为这些函数通常仅需要一条指令。使用 `-g` 选项编译时，函数的内联被禁用；而使用 `-g0` 编译则允许函数的内联。
- 显式或自动的内联由高优化级别（4 和 5）的编译器执行。在打开 `-g` 时执行显式和自动的内联。这种类型内联的理论基础可以节省函数调用的代价，但是更为常见是提供可以优化寄存器使用和指令调度的更多指令。

两种类型的内联对度量的显示具有相同的影响。显示在源代码中但已内联的函数不显示在函数列表中，也不做为已内联的函数的被调用方显示。否则在内联函数的调用点上显示为包括度量（表示被调用的函数中所用的时间）的度量实际显示为归属到调用点的排除度量，表示内联函数的指令。

注 – 内联会使数据难以解释，因此编译程序进行性能分析时可能要禁用内联。

某些情况下，甚至在函数被内联时，会留下所谓的线外函数。某些调用点调用线外函数，但其它站点具有内联的指令。这些情况下，函数显示在函数列表中但归属到函数的度量仅表示线外调用。

编译器生成的主体函数

编译器并行执行函数中的循环或具有并行指令的区域时，编译器创建初始源代码中不存在的新的主体函数。这些函数在第 108 页的 “并行执行和编译器生成的主体函数”中描述。

性能分析器将这些函数做为普通函数显示，并将名称分配到这些函数（基于它们被提取的函数），其中编译器生成的名称例外。它们的排除和包括度量表示主主体函数中所用的时间。此外，提取结构的函数显示了源于每个主体函数的包括度量。产生这种情况的方法在第 108 页的 “并行执行序列”中描述。

包含并行循环的函数被内联时，该函数编译器生成的主体函数名称将函数反映到它被内联的函数中，而不是原来的函数。

注 – 只有用 -g 编译的模块才能裁减编译器生成主体函数的名称

外联函数

外联函数可以在反馈优化期间创建。这些函数表示非正常执行的代码。具体来讲，指得是用于生成反馈的训练运行期间未被执行的代码。要改善页面和指令缓冲行为，这种代码被移动到地址空间的其它位置，并放置到独立的函数中。外联函数的名称将关于外联的代码段信息编码，这些信息包括代码被提取的函数名称和源代码中开始的行号。不同的发行版本可以具有不同的粉碎名称。性能分析器提供了函数名称的可读版本。

外联函数不会被真正的调用，只是被跳转；类似的，这些函数不是返回，而是跳回。为了使该行为更紧密地匹配用户的源代码模型，性能分析器将人工调用从主函数输入到它的外联部分。

带有合适包括和排除度量的外联函数做为普通函数显示。此外，外联函数的度量如同在外联化代码函数中的包括度量一样增加。

动态编译的函数

动态编译的函数是程序执行时编译和链接的函数。收集器不具有关于用 C 或 C++ 编写的动态编译函数的信息，除非用户使用 [收集器 API] 函数提供了所需的信息。关于 API 函数的信息请参阅第 47 页的 “动态函数和模块”。如果未提供该信息，则该函数在性能分析中显示为 `<Unknown>`。

对于 Java 程序，收集器获得关于 Java HotSpot™ 虚拟机编译的方法的信息，并且不需要使用 API 函数提供信息。对于其它的方法，性能工具显示执行这些方法的 Java™ 虚拟机的信息。在 Java 模式中，所有的方法与解释版本合并。在专家 Java 模式中，每个方法的 HotSpot 编译和解释的版本同时单独地显示。在机器模式中，每个 HotSpot 编译版本单独显示，并且显示每个解释方法的 JVM 函数。

<Unknown> 函数

在某些情况下， PC 不映射到已知的函数。在这些情况下， PC 被映射到名为 `<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++ 编写的函数时，不使用 [收集器 API] 函数将关于函数的信息提供到收集器时。关于 [收集器 API] 函数的更多信息请参阅第 47 页的 “动态函数和模块”。
- Java 方法被动态编译但 Java 分析被禁用时。
- PC 对应到可执行文件或共享对象的数据段中的地址时。一种情况是 SPARC V7 版本的 `libc.so`，该版本在其数据段中具有多个函数 （例如 `.mul` 和 `.div`）。该代码位于数据段中，以便该库检测到该代码正在 SPARC V8 或 V9 平台上执行时可以动态重写该代码以使用机器指令。
- PC 对应到可执行文件 （未在实验中记录）的地址空间中的共享对象时。
- PC 不在任何已知的负载对象内时。这种情况最可能的原因是解除失败，做为 PC 记录的值根本不是 PC，而是某些其它数据。如果 PC 是返回寄存器，并且看上去不在任何已知的负载对象内，则该 PC 被忽略，而不是归属到 `<Unknown>` 函数。
- PC 映射到收集器不具有符号信息的 Java™ 虚拟机内部时。

`<Unknown>` 函数的调用方与被调用方表示调用栈中的前一个和下一个 PC，并且被正常对待。

<no Java callstack recorded> 函数

<no Java callstack recorded> 函数类似于 <Unknown> 函数，但对于 Java 线程，只有在 Java 表示中才与后者类似。收集器从 Java 线程接收事件时，收集器会解除本机堆栈并调用到 Java 虚拟机来获取相应的 Java 堆栈。如果因任何原因该调用失败，则该事件显示在带有人工函数 <no Java callstack recorded> 的分析器中。JVM 为了避免死锁或解除 Java 堆栈时引起过多同步，可能拒绝报告调用栈。

<Total> 函数

<Total> 函数是人工的结构，用于将程序表示为一个整体。除正被归属到调用栈上函数的性能度量外，其它性能度量都被归属到特殊函数 <Total>。该函数显示在函数列表的顶部，其数据可用于对其它函数的数据给出透视。在 [调用方与被调用方] 列表中，该函数显示为任何程序执行主线程中的 _start() 名义调用方，还显示为已创建线程的 _thread_start() 的名义调用。如果堆栈解除是不完整的，则 <Total> 函数可以做为其它函数的调用方显示。

与 HW 计数器分析有关的函数

以下函数与 HW 计数器分析相关：

- collector_not_program_related：计数器与该程序不相关。
- collector_lost_hwc_overflow：计数器显示为已超出不生成溢出符号的溢出值。该值被记录而计数器重置。
- collector_lost_sigemt：计数器显示为已超出溢出值并且被停止，但溢出符号显示为已丢失。该值被记录而计数器重置。
- collector_hwc_ABORT：读取硬件计数器已经失败，一般当优先进程已控制计数器时会导致硬件计数器收集的终止。
- collector_final_counters：挂起或终止之前一刻获得的计数器值，以及因前一个溢出而产生的计数。如果这对应到 <Total> 计数的主要部分，则推荐使用更小的溢出间隔 （即，更高的分辨率配置）。
- collector_record_counters：处理和记录硬件计数器事件时累计的计数，还有一部分是对硬件计数器分析的开销的计数。如果这对应到 <Total> 计数的主要部分，则推荐使用更大的溢出间隔 （即，更低的分辨率配置）。

将数据地址映射到程序数据对象

一旦源于与内存操作对应的硬件计数器事件的 PC 已被处理，并成功回溯到可能的临时内存引用指令，则性能分析器使用硬件分析支持信息中编译器提供的指令标识符和描述符来派生有关的程序数据对象。

术语 *dataobject* 用于表示程序常量、变量，数组和累计 （如结构和联合以及具有明确累计的元素），如源代码中所述。取决于源语言， dataobject 类型和它们大小是会变化的。许多 dataobject 在源程序中显式命名，而其它可能是未命名的。某些 dataobject 从其它（简单的） dataobject 派生或累计，产生丰富且通常会是复杂的 dataobject 集合。

每个 dataobject 包含关联的范围，定义和可以引用源程序的可能是全局的区域 （例如负载对象），特定的编译单元 （对象文件）或函数。相同的数据对象可能定义为具有不同的范围，或在不同的范围以不同的方式引用的特定数据对象。

从使用启用回溯收集内存操作的硬件计数器事件派生的数据被归属到关联的程序 dataobject，并传送到包含 dataobject 和人工 `<Total>` 的任何累计，该人工函数被认为包含 `<Unknown>` 和 `<Scalars>` 在内的所有 dataobject。 `<Unknown>` 的不同子类型会传播到 `<Unknown>` 累计。接下来的一节描述 `<Total>`、 `<Scalars>` 和 `<Unknown>` dataobject。

Dataobject 描述符

可以通过声明类型和名称的组合来完整描述数据对象。简单的标量 dataobject {`int i`} 描述了类型为 `int` 名为 `i` 的变量，而 {`const+pointer+int p`} 描述了指向类型为 `int` 名为 `p` 的常量指针。包含空格的类型将它们替换为下划线 (_)，而未命名的数据对象用破折号 (-) 的名称表示，例如：{`double_precision_complex -`}。

完整累计被类似表示为类型为 `foo_t` 名为 `foo` 的结构 {`structure:foo_t foo`}。累计的 `single` 元素需要其容器的附加规范，例如 {`structure:foo_t foo`}.{`int i`} 中成员为 `i`，类型为 `int`，使用的是前一个结构 `foo`。累计本身也可以是（更大的）累计的元素，同时与累计对应的描述符被构造为累计描述符和最终标量描述符的拼接。

完全限定的描述符不可能总需要消除 dataobject 的歧义时，该描述符提供了一般的完整规范来协助标识 dataobject。

`<Total>` Dataobject

`<Total>` dataobject 是人工的结构，用于将程序的数据对象表示为一个整体。除正被归属到不同 dataobject （和任何它属于的累计）的性能度量外，所有性能度量都被归属到特殊的 dataobject `<Total>`。该函数显示在 dataobject 列表的顶部，其数据可用于对其它数据对象的数据给出透视。

<Scalars> Dataobject

当累计元素包含它们的性能度量，并将这些度量另外归属到它们关联累计的度量值中时，所有标量常量和变量包含它们的性能度量，并将这些度量另外归属到人工 <Scalars> dataobject 的度量值中。

<Unknown> Dataobject 及其元素

在不同的情况下，事件数据不能被映射到特定的 dataobject。这种情况下，数据被映射到名为 <Unknown> 的特殊 dataobject 及其元素之一，如下文所述。

- 不确定

  无需硬件分析支持，一个或多个编译对象被编译，因此确定与内存引用指令有关的数据对象或验证回溯是不可能的。

- 无法验证

  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。

- 无法解析

  事件回溯遇到了控制传输目标，并且不管控制传输是否发生都不能解决，所以事件回溯不能决定可能的临时内存引用指令 （与其关联的 dataobject）。

- 未指定

  回溯决定了可能的临时内存引用指令，但与回溯关联的 dataobject 不由编译器指定。

- 未标识

  回溯决定了可能的临时内存引用指令，但回溯不由编译器标识，并且因此确定有关的 dataobject 是不可能的。编译器临时文件通常是未标识的。

注释代码列表

注释源代码行和注释反汇编代码可用于决定函数中哪些源代码或指令会使性能降低。本节描述了注释过程和与解释注释代码有关的问题。

注释源代码

注释源代码在源代码行级别显示了应用程序消耗的资源。要显示资源使用状况，可以通过获得记录在应用程序调用栈中的 PC，并将每个 PC 映射到源代码行。要生成注释源文件，性能分析器首先决定在特定对象模块 （.o 文件）或负载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。为了生成注释源码，性能分析器必须能够找到并读取对象模块或负载对象来决定从 PC 到源代码行的映射，而且性能分析器必须能够读取源文件来生成显示的注释副本。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，并在找到正确基名的文件时停止。

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

编译过程要经历很多阶段，这取决于请求的优化级别，并且会发生转换，该转换会混淆从指令到源代码行的映射。对于某些优化，源代码行信息可能完全丢失；而对于其它的优化，源代码行信息则可能发生混淆。编译器依赖各种试探来跟踪指令的源代码行，而这些试探不是绝对无误的。

解释源代码行度量

等待指令被执行时指令的度量必须作为增加的度量解释。如果记录事件时执行的指令来自与分支 PC 相同的源代码行，那么度量可被解释为原因是执行了该代码行。不过，如果分支 PC 与执行的指令来自不同的源代码行，那么分支 PC 所属源代码行的度量至少有一些必须解释为 “该行等待执行时累积的度量”。如示例：当一条源代码行上计算的值被用在下一条源代码行上时。

执行中有严重的延迟时 （例如缓存缺少或资源队列延迟），或指令等待前一个指令返回结果时，如何解释度量问题是最重要的。在这些情况下，源代码行的度量看上去高的不合理，您可以在代码中的其它行上查找负责高度量值的行。

度量格式

可以出现在注释源代码行上度量的四种可能格式在表 5-3 中说明。

表 5-3 注释源代码度量

度量	有效
（空白）	程序中没有 PC 对应于该行代码。这种情况应该始终适用于注释行，也适用于以下环境中出现的源代码行： • 出现的代码段的所有指令已在优化期间删除。 • 代码在其它地方重复，并且编译器执行公用子表达式识别并标记带有其它副本代码行的所有指令。 • 编译器用不正确的行号来标记指令。
`0.`	程序中的某些 PC 被标记为从该代码行派生，但没有引用到这些 PC 的数据：它们从不会在统计抽样或跟踪线程同步数据的调用栈中。 `0` 度量不表示该行不执行，只是表示该行不统计地显示在分析数据包或跟踪数据包中。
`0.000`	至少该行的一个 PC 出现在数据中，但是计算的度量值取整到零。
`1.234`	到该行的所有 PC 属性的度量总计达到了显示的非零数值。

编译器注释

编译器的不同部分可以将注释收编到可执行文件中：每个注释与特定的源代码行关联。注释源码被写入后，任何源代码行的编译器注释立即显示在源代码行的前面。

编译器注释描述了优化源代码已经做的大量转换。这些转换包括循环优化、并行、内联和流水线作业。

公用子表达式排除

一个很普通的优化就可以识别到相同的表达式会出现在多个位置，并且通过为某个位置中的表达式生成代码可以改善性能。例如，如果在代码块的 `if` 和 `else` 分支同时出现相同的操作，则编译器可以将该操作刚好移动到 `if` 语句的前面。然后，编译器将行号分配到基于前一个出现的表达式的指令。如果分配到公共代码的行号对应到 `if` 结构的一个分支，并且该代码实际上始终包含另外的分支，则注释源码在不包含的分支内的行上显示度量。

并行指令

编译器从包含并行指令的代码生成主体函数时，并行循环或段的包括度量被归属到该并行指令，因为该代码行是编译器生成的主体函数的调用点。包括或排除度量也显示在循环或段中的代码上。这些度量累加到并行指令上的包括度量。

源码中的特殊代码行

无论何时 PC 的源代码行都不能被决定，该 PC 的度量被归属到插入注释源文件顶部的特殊源代码行。该行上的高度量表明源于给定对象模块的代码部分不具有行映射。注释反汇编码可以帮助您决定不具有映射的指令。特殊的代码行有：

- *Function* < 没有行号的指令 >
  其中 *function* 是产生指令的函数名称。
- *Function* < 源文件名未记录 >
  其中 *function* 是产生指令的函数名称。

注释反汇编码

注释反汇编提供了函数或对象模块指令的汇编码列表，以及与每个指令关联的性能度量。注释反汇编有多种显示方法，这取决于行号映射和源文件是否可用，以及正在请求注释反汇编码的函数的对象模块是否已知：

- 如果对象模块是未知的，则性能分析器反汇编刚刚指定函数的指令，并且不显示任何反汇编中的源代码。
- 如果对象模块是已知的，则反汇编涵盖了对象模块内所有的函数。
- 如果源文件可用，并且行号数据记录，则性能分析器可以取决于显示首选项与反汇编交叉存取源码。
- 如果编译器已将任何注释插入对象代码中，则如果对应的首选项被设置后该代码也可以在反汇编中交叉存取。

反汇编码中的每个指令都用以下信息来注释。

- 源代码行号，如编译器所报告
- 它的相对地址
- 请求后指令的十六进制表示
- 指令的汇编程序 ASCII 表示

如果可能，调用地址被解决到符号 （例如函数名称）。度量被显示在指令的代码行上，并且如果对应的首选项被设置后度量可以显示在任何交叉存取的源代码上。可能的度量值像源代码注释的度量值一样描述，如表 5-3 所示。

代码未被优化时，每个指令的行号按顺序显示，源代码行与反汇编指令的交叉存取以意料中的方式出现。进行优化时，后面代码行的指令有时会显示在早期代码行的指令前面。性能分析器的交叉存取算法是指令被随时显示为来源于第 *N* 行，该行前所有的源代码行（包括第 *N* 行）都写在该指令之前。优化的一个效果是源代码可以显示在控制传输指令与其延迟槽指令之间。与源码的第 *N* 行关联的编译器注释刚好写在该行的前面。

解释注释反汇编并不简单。分支 PC 是下一条要执行的指令地址，因此归属到指令的度量应该被视为等待执行指令所用的时间。不过，指令的执行不总是按顺序发生，而且在记录调用栈时可能有延迟。要利用注释反汇编，您应该熟悉记录实验的硬件和硬件装入和执行指令的方式。

接下来的几个子章节讨论解释注释反汇编的问题。

指令发布分组

指令按组装入和发布就叫做指令发布分组。哪些指令包括在组中取决于硬件、指令类型、已执行的指令和与对其它指令或寄存器的任何依存。这意味着并没有充分表示某些指令，因为这些指令总是在与前一条指令相同的时钟循环中执行，因此这些指令根本未表示下一条要执行的指令。这也意味着记录调用栈时，可能有多条指令被视为是下一条要执行的指令。

指令发布规则因处理器类型而不同，且取决于缓存代码行内的指令对齐。因为链接程序以更精细的粒度强制指令对齐而不是缓冲代码行，在看上去不相关的函数中的更改会导致指令的不同对齐。不同的对齐会引起性能的改善或降级。

以下人工情况显示了在稍微不同的情况下编译和链接的相同函数。以下显示的两个输出示例是 er_print 的注释反汇编列表。两个示例的指令是相同的，但是指令的对齐不同。

在该示例中，指令对齐将这两个指令 cmp 和 bl,a 映射到不同的缓冲代码行，并且大量时间用于等待执行这两条指令。

```
   排除       包括
用户 CPU   用户 CPU
    秒        秒
                                    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. }
```

在该示例中，指令对齐将这两个指令 cmp 和 bl,a 映射到相同的缓冲代码行，并且大量时间用于等待执行这些指令中的其中一条。

```
   排除      包括
用户 CPU   用户 CPU
    秒        秒
                                  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. }
```

指令发布延迟

有时，会更频繁地显示特定的分支 PC，因为这些 PC 表示的指令在发布前被延迟。发生这种情况的原因有很多，以下列出了其中的一些：

- 执行前一条指令用了很长的时间，且执行不能间断，例如指令陷入内核中时。
- 运算指令需要寄存器，该寄存器内容因使用还未完成的早期指令设置而使其不可用。具有数据缓冲缺少的装入指令就是这种延迟的示例。
- 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。
- 该指令缓冲不包括包含指令 （I-cache 缺少）的内存字。
- 在 UltraSPARC III 处理器上，装入指令上的缓冲缺少在缺少被解决之前会阻塞后续的所有指令，这与这些指令是否使用正被装入的数据项无关。UltraSPARC® II 处理器仅阻塞使用正被装入的数据项。

硬件计数器溢出的属性

除 TLB 缺少外，硬件计数器溢出事件的调用栈按指令的顺序被记录在后续的某些点上，而不是记录在发生溢出的点，其中的一个原因是处理溢出生成的中断所用的时间。对于某些计数器（诸如发布的循环或指令），不会发生这种情况。对于诸如这些计数缓冲缺少或浮点运算的其它计数器，度量被归属到负责溢出的不同指令。引起事件的 PC 通常只是记录的 PC 前面的几条指令，而这些指令可以在反汇编列表中正确定位。不过，如果该指令范围内有分支目标，则要告知哪条指令对应到引起事件的 PC 是很困难或不可能的。对于计数内存访问事件的硬件计数器，如果该计数器名称的前缀为加号 +，则收集器将搜索引起事件的 PC。

在 [反汇编] 和 [PC] 标签中的特殊代码行

在 [反汇编] 和 [PC] 标签中可能显示很多特殊代码行。本节使用这些代码行在分析器中的可能显示来描述这些代码行。

- <static>@0x1a4400 + 0x000032B8
  表示静态函数及其偏移
- <*library*.so> -- 未找到函数 + 0x0000F870
  与上一个相同，也包括偏移
- *Method_Name* <HotSpot 编译分支指令 >
  表示已命名 Java 方法的 HotSpot 编译版本
- *Method_Name* <Java 本机方法 >
  表示 Java 本机方法生成的指令
- < 未记录 Java 调用栈 > + 0x00000000
  偏移是来自 JVM 的错误代码，通常 0 表示什么也没有，但没有解除
- < 分支目标 >
  在反汇编中，事件 PC 的回溯及其有效的地址因运行到分支目标而失败，并因此在其后不能复原。

程序链接表 (PLT) 指令

一个负载对象中的函数调用不同共享对象中的函数时，实际调用首先传输到 PLT 中的三指令序列，然后再传输到真实目标。分析器删除对应到 PLT 的 PC，并将这些 PC 的度量分配到调用指令。因此，如果调用指令具有意外的高度量，则这可能取决于 PLT 指令而不是调用指令。另见第 104 页的 “共享对象间的函数调用”。

第 6 章

操作实验和查看注释的代码列表

本章描述了可以与收集器和性能分析器一起使用的公用程序。

本章涵盖了以下主题：

- 操作实验
- 用 er_src 查看带注释的代码列表
- 其他公用程序

操作实验

实验存储在收集器创建的目录中。要操作实验，您可以使用正常的 Unix 命令 cp、 mv 和 rm 并将它们应用到该目录。比 Forte Developer 7 （Solaris™ 操作系统的 Sun™ ONE Studio 7 企业版）早的发行版本的实验不具有这种功能。三个公用程序的功能类似于 Unix 命令，用来复制、移动和删除实验。 er_cp(1)、 er_mv(1) 和 er_rm(1) 这三个公用程序在下文中描述。

实验中的数据包括了程序所用每个负载对象的归档文件。这些归档文件包含了负载对象的绝对路径和最后一次修改的日期。移动或复制实验时该信息不会更改。

er_cp [-V] *experiment1* *experiment2*

er_cp [-V] *experiment-list* *directory*

er_cp 命令的第一种形式将 *experiment1* 复制到 *experiment2*。如果 *experiment2* 存在，则 er_cp 退出并显示错误消息。第二种形式是将空格分隔的实验列表复制到一个目录。如果目录已经包含的实验名称与正被复制的实验名称相同，则 er_mv 退出并显示错误消息。 -V 选项打印 er_cp 的版本。该命令不能复制早于 Forte Developer 7 的软件发行版本创建的实验。

er_mv [-V] *experiment1 experiment2*

er_mv [-V] *experiment-list directory*

er_mv 命令的第一种形式将 *experiment1* 移动到 *experiment2*。如果 *experiment2* 存在，则 er_mv 退出并显示错误消息。第二种形式是将空格分隔的实验列表移动到一个目录。如果目录已经包含的实验名称与正被移动的实验名称相同，则 er_mv 退出并显示错误消息。-V 选项打印 er_mv 的版本。该命令不能移动早于 Forte Developer 7 的软件发行版本创建的实验。

er_rm [-f] [-V] *experiment-list*

删除实验列表或实验组。实验组删除后，组中的每个实验以及组文件都被删除。无论是否找到实验， -f 选项都会禁止错误消息并确保成功完成。 -V 选项打印 er_rm 的版本。该命令删除早于 Forte Developer 7 的软件发行版本创建的实验。

用 er_src 查看带注释的代码列表

无需运行实验，使用 er_src 公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。显示的生成方式与在性能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 [ options ] object item tag
```

object 是可执行文件、共享对象或对象文件 （.o 文件）的名称。

item 是用于生成可执行文件或共享对象的函数、源代码或对象文件的名称，指定对象文件时该名称可以省略。

tag 是索引，用于决定多个函数具有相同的名称时引用到哪个 *item*。如果不需要，则可以省略。如果需要但已被省略，则打印消息，列出可能的选择。

以下几节描述了 er_src 公用程序可以使用的选项。

`-c` *commentary-classes*

定义要显示的编译器注释类。 *commentary-classes* 是用冒号分隔的类的列表。关于这些类的描述请参阅第 80 页的 “控制源码和反汇编列表的命令”。

注释类可以在缺省文件中指定。首先读取系统范围的缺省文件 `er.rc`，然后读取用户起始目录中的 `.er.rc` 文件，最后读取当前目录中的 `.er.rc` 文件（如果存在）。起始目录中 `.er.rc` 文件的缺省覆盖系统的缺省，而当前目录中 `.er.rc` 文件的缺省覆盖起始和系统的缺省。这些文件也可以由性能分析器和 `er_print` 使用，但只有源码和反汇编编译注释由 `er_src` 使用。

关于缺省文件的描述请参阅第 90 页的“设置缺省值的命令”。在缺省文件中除 `scc` 和 `dcc` 之外的命令都被 `er_src` 忽略。

`-d`

在列表中包括反汇编。缺省列表不包括反汇编。如果没有可用的源码，则产生一个不带编译器注释的反汇编列表。

`-o` *filename*

打开文件 *filename* 用于列表的输出。缺省情况下，输出写入 `stdout`。

`-V`

打印当前发行版本。

其他公用程序

有一些其他公用程序不必在正常情况下使用。在这里说明这些公用程序是为了手册的完整，同时还描述了可能需要使用这些程序的情况。

er_archive 公用程序

er_archive 命令的语法如下所示。

```
er_archive [-qAF] 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 时， er_archive 公用程序自动运行。公用程序读取在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，而且包含了共享对象的函数和模块映射。

如果目标程序异常终止，则 er_archive 不能由收集器运行。如果想要分析在与记录数据不同的机器上运行异常终止的实验，则必须在记录数据的机器上运行实验的 er_archive。要确保在将实验复制到的机器上可以使用负载对象，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。这些归档文件包含了负载对象中每个对象文件和每个函数的地址、大小和名称，以及负载对象的绝对路径和最后一次修改的时间标记。

如果运行 er_archive 时找不到共享对象，或如果时间标记与实验中记录的不同，或如果在与记录实验不同的机器上运行 er_archive，则归档文件包含警告。无论何时手动（没有 -q 标志）运行 er_archive，警告也会写到 stderr 中。

以下几节描述了 er_archive 公用程序可以使用的几个选项。

-q

不将任何警告写入 stderr。警告被收入归档文件中，且显示在性能分析器或 er_print 输出中。

-A

请求将所有负载对象写入实验中。该参数用于生成实验，该实验很可能被复制到不是记录实验的机器上。

-F

强制写入或重写归档文件。该参数可用于手动运行 er_archive，和重写具有警告的文件。

-V

写入 er_archive 的版本号信息并退出。

er_export 公用程序

er_export 命令的语法如下所示。

```
er_export [-V] experiment
```

er_export 公用程序将实验中的原始数据转换成 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。只有性能分析器不能读取实验时才使用该公用程序；输出允许工具的开发人员理解原始数据并分析故障。 -V 选项用于打印版本号信息。

附录 A

使用 prof、 gprof 和 tcov 来分析程序

本附录中所讨论的工具是为程序计时和获取分析性能数据的标准公用程序，通常被称作传统分析工具。分析工具 prof 和 gprof 由 Solaris™ 操作系统提供。tcov 是由 Sun™ 编译器和工具提供的代码覆盖工具。

注 – 如果您要跟踪了解一个函数被调用的次数或一行源代码的执行频率，就请使用这些传统分析工具。如果您要了解对程序花费时间所在位置的详细分析，那么可以通过 [收集器] 和 [性能分析器] 获取更准确的信息。关于使用这些工具的更多信息，请参阅第 3 章和 [联机帮助]。

表 A-1 描述了由这些标准性能分析工具生成的信息。

表 A-1 性能分析工具

命令	输出
prof	生成程序所用 CPU 时间的统计分析和每个函数的精确进入次数。
gprof	生成程序所用 CPU 时间的统计分析，每个函数的精确进入次数，以及遍历程序调用图中每个 arc （调用方与被调用方对）的次数。
tcov	生成执行程序中每个语句的精确次数。

并非所有传统分析工具都在使用除 C 以外的编程语言编写的模块中工作。关于各个语言的更多信息请参阅每个工具中的章节。

本附录涵盖了以下主题：

- 使用 prof 生成程序分析
- 使用 gprof 生成调用图分析
- 将 tcov 用于语句级分析
- 将 tcov 增强版用于语句级分析

使用 prof 生成程序分析

prof 生成程序所用 CPU 时间的统计分析并计算程序中每个函数的进入次数。不同或更详细的数据由 gprof 调用图分析和 tcov 代码覆盖工具提供。

要使用 prof 生成分析报告：

1. **用 -p 编译器选项编译您的程序。**
2. **运行您的程序。**

   分析数据被发送到名为 mon.out 的分析文件。每次运行程序时都会覆盖该文件。
3. **运行 prof 生成分析报告。**

   prof 命令的语法如下所示。

```
% prof program-name
```

其中，*program-name* 是可执行文件的名称。该分析报告被写入 stdout。它表示为以下列标题下每个函数的一系列行：

- %Time - 函数占用总 CPU 时间的百分比。
- Seconds - 函数计算的总 CPU 时间。
- Cumsecs - 函数计算的运行秒数，以及先前所列秒数之和。
- #Calls - 函数的调用次数。
- msecs/call - 每次调用函数所用的平均毫秒数。
- Name - 函数的名称。

prof 的使用在下例中进行了描述。

```
% cc -p -o index.assist index.assist.c
% index.assist
% prof index.assist
```

prof 的分析报告如下表所示：

%Time	Seconds	Cumsecs	#Calls	msecs/call	Name
19.4	3.28	3.28	11962	0.27	compare_strings
15.6	2.64	5.92	32731	0.08	_strlen
12.6	2.14	8.06	4579	0.47	__doprnt
10.5	1.78	9.84			mcount
9.9	1.68	11.52	6849	0.25	_get_field
5.3	0.90	12.42	762	1.18	_fgets
4.7	0.80	13.22	19715	0.04	_strcmp
4.0	0.67	13.89	5329	0.13	_malloc
3.4	0.57	14.46	11152	0.05	_insert_index_entry
3.1	0.53	14.99	11152	0.05	_compare_entry
2.5	0.42	15.41	1289	0.33	lmodt
0.9	0.16	15.57	761	0.21	_get_index_terms
0.9	0.16	15.73	3805	0.04	_strcpy
0.8	0.14	15.87	6849	0.02	_skip_space
0.7	0.12	15.99	13	9.23	_read
0.7	0.12	16.11	1289	0.09	ldivt
0.6	0.10	16.21	1405	0.07	_print_index

.
.
.
（剩余输出可忽略不计）

该分析报告显示多数程序执行时间用于 compare_strings() 函数；此外多数 CPU 时间用于 _strlen() 库函数。为了使该程序更有效率，用户会关注几乎占用总 CPU 时间 20% 的 compare_strings() 函数，改进算法或减少调用的次数。

从 prof 分析报告来看， compare_strings() 的大量递归并不明显，但是您可以通过第 138 页的 “使用 gprof 生成调用图分析” 所述的调用图分析推出递归程度。在这个特殊条件下，改进算法也可以减少调用次数。

注 – 对 Solaris 操作系统 7 和 8 而言， CPU 时间的分析对使用多个 CPU 的程序是精确的，但实际上，未锁定的计数可能会影响函数计数的准确性。

使用 gprof 生成调用图分析

当 prof 的平直分析可以提供有价值的性能改善数据时，通过调用图分析显示一个标识了哪些模块被其它模块调用，哪些模块调用其它模块的列表，就可以获得更详细的分析。有时，完全删除调用可带来性能改善。

注 – gprof 规定在对调用方的函数中所花的时间与每个 arc 被遍历的次数成比例。因为并非所有的调用都在性能上等效，这就可能产生错误的假定。示例请见 developers.sun.com 网站上的性能分析器教程。

与 prof 类似， gprof 生成程序所用 CPU 时间的统计分析并计算每个函数进入的次数。 gprof 还计算程序调用图中每个 arc 被遍历的次数。 *arc* 是调用方与被调用方对。

注 – 对 Solaris 操作系统 7 和 8 而言， CPU 时间的分析对使用多个 CPU 的程序是精确的，但实际上，未锁定的计数可能会影响函数计数的准确性。

要使用 gprof 生成分析报告：

1. **用适当的编译器选项编译您的程序。**
   - 对 C 程序，请使用 -xpg 选项。
   - 对 Fortran 程序，请使用 -pg 选项。

2. **运行您的程序。**

   分析数据被发送到名为 gmon.out 的分析文件。每次运行程序时都会覆盖该文件。

3. **运行 gprof 生成分析报告。**

   prof 命令的语法如下所示。

```
% gprof program-name
```

其中， *program-name* 是可执行文件的名称。该分析报告被写入 stdout，可能会非常大。该报告由两个主要项目组成：

- 完整的调用图分析，它显示了关于程序中每个函数调用方和被调用方的信息。其格式在下例中进行了描述：
- 平直分析，它与 prof 命令所提供的汇总相似。

gprof 的分析报告包含对汇总各部分含义的解释，并标识了抽样的粒度，具体情况如下例所示。

```
granularity:each sample hit covers 4 byte(s) for 0.07% of 14.74
seconds
```

4 bytes 意味着一条指令的分辨率。 0.07% of 14.74 seconds 意味着表示十毫秒 CPU 时间的每个抽样占运行的 0.07%。

gprof 的使用在下例中进行了描述。

```
% cc -xpg -o index.assist index.assist.c
% index.assist
% gprof index.assist > g.output
```

下表是部分调用图分析。

index	%time	self	descendants	called/total parents called+self called/total children	name	index
--						
		0.00	14.47	1/1	start	[1]
[2]	98.2	0.00	14.47	1	_main	[2]
		0.59	5.70	760/760	_insert_index_entry	[3]
		0.02	3.16	1/1	_print_index	[6]
		0.20	1.91	761/761	_get_index_terms	[11]
		0.94	0.06	762/762	_fgets	[13]
		0.06	0.62	761/761	_get_page_number	[18]
		0.10	0.46	761/761	_get_page_type	[22]
		0.09	0.23	761/761	_skip_start	[24]
		0.04	0.23	761/761	_get_index_type	[26]
		0.07	0.00	761/820	_insert_page_entry	[34]
--						
				10392	_insert_index_entry	[3]
		0.59	5.70	760/760	_main	[2]

```
[3]      42.6    0.59    5.70       760+10392          _insert_index_entry    [3]
                 0.53    5.13       11152/11152        _compare_entry         [4]
                 0.02    0.01       59/112             _free                  [38]
                 0.00    0.00       59/820             _insert_page_entry     [34]
                                    10392              _insert_index_entry    [3]
-----------------------------------------------
```

在该示例中， index.assist 程序的输入文件中有 761 行数据。可得出以下结论：

- fgets() 被调用 762 次。对 fgets() 的最后一次调用返回文件结尾。
- insert_index_entry() 函数在 main() 中被调出了 760 次。
- 除了从 main() 调用 insert_index_entry() 760 次之外，insert_index_entry() 还自身调用 10,392 次。 insert_index_entry() 是一个大量递归。
- compare_entry() 从 insert_index_entry() 调出 11,152 次，等于 760+10,392 次。每次调用 insert_index_entry() 时都会调用一次 compare_entry()。这是正确的。如果在调用次数上有差异，您就会怀疑在程序逻辑上出现了某些问题。
- insert_page_entry() 总计被调用 820 次：其中， 761 次是在程序生成索引结点时从 main() 调用，59 次是从 insert_index_entry() 调用的。这个频率表示有 59 个重复的索引条目，所以它们的页面编号条目被链接到一个具有索引结点的链中。然后这 59 个重复的索引条目被释放并调用 free()。

将 tcov 用于语句级分析

tcov 公用程序提供了关于程序执行代码段频率的信息。它生成具有执行频率注释的源文件副本。可在基本块级别或源行级别注释代码。基本块是没有分支的源代码线性段。基本块中的语句会被执行相同的次数，因此基本块执行的计数还可以告诉您块中每个语句被执行的次数。 tcov 公用程序不生成任何基于时间的数据。

注 – 虽然 tcov 可用于 C 和 C++ 程序，但它不支持包含 #line 或 #file 指令的文件。 tcov 不启用 #include 头文件中代码的测试覆盖分析。

要使用 tcov 生成带注释的源代码：

1. 用适当的编译器选项编译您的程序。

- 对 C 程序，请使用 -xa 选项。
- 对 Fortran 程序，请使用 -a 选项。

如果您使用 -a 或 -xa 选项进行编译，还必须与其相链接。编译器为每个对象文件创建一个具有 .d 后缀的覆盖数据文件。该覆盖数据文件创建在由环境变量 TCOVDIR 指定的目录下。如果未设置 TCOVDIR，则覆盖数据文件创建在当前目录下。

注 – 用 -xa (C) 或 -a （其它编译器）编译的程序比正常情况要运行缓慢，因为更新每个执行的 .d 文件占用了可观的时间。

---

2. 运行您的程序。

当程序完成时，覆盖数据文件已被更新。

3. 运行 tcov 生成带注释的源代码。

tcov 命令的语法如下所示。

```
% tcov options source-file-list
```

其中， *source-file-list* 为源代码文件名列表。关于选项列表，请参阅 tcov(1) 手册页。tcov 的缺省输出为一系列文件，每个文件都带有可使用 -o *filename* 选项进行修改的后缀 .tcov。

为代码覆盖分析编译的程序可多次运行 （有潜在的可变输入时）；每次运行比较行为之后， tcov 可用于该程序。

下面示例说明了 tcov 的使用。

```
% cc -xa -o index.assist index.assist.c
% index.assist
% tcov index.assist.c
```

这个小 C 代码片断 （来自 index.assist 模块之一）显示了被递归调用的 insert_index_entry() 函数。位于 C 代码左侧的数字显示了每个基本块的执行次数。 insert_index_entry() 函数被调用了 11,152 次。

```
               struct index_entry *
11152     ->   insert_index_entry(node, entry)
               struct index_entry *node;
               struct index_entry *entry;
               {
                   int result;
                   int level;

                   result = compare_entry(node, entry);
                   if (result == 0) {     /* exact match */
                                          /* Place the page entry for the
duplicate */
                                          /* into the list of pages for this node
*/
59        ->               insert_page_entry(node, entry->page_entry);
                           free(entry);
                           return(node);
                   }

11093     ->       if (result > 0)        /* node greater than new entry -- */
                                              /* move to lesser nodes */
3956      ->               if (node->lesser != NULL)
3626      ->                       insert_index_entry(node->lesser, entry);
                           else {
330       ->                       node->lesser = entry;
                                   return (node->lesser);
                           }
                       else               /* node less than new entry -- */
                                          /* move to greater nodes */
7137      ->               if (node->greater != NULL)
6766      ->                       insert_index_entry(node->greater, entry);
                           else {
371       ->                       node->greater = entry;
                                   return (node->greater);
                           }
               }
```

tcov 公用程序会在带注释的程序列表的末尾放置一个如下所示的汇总。对最频繁执行基本块的统计是按照执行频率的顺序列出的。行号是块中第一行的行号。

以下是对 index.assist 程序的汇总：

```
              Top 10 Blocks

              Line          Count
              240           21563
              241           21563
              245           21563
              251           21563
              250           21400
              244           21299
              255           20612
              257           16805
              123           12021
              124           11962

77            Basic blocks in this file
55            Basic blocks executed
71.43         Percent of the file executed

              439144      Total basic block executions
              5703.17     Average executions per basic block
```

创建 tcov 的分析共享库

可以创建一个 tcov 分析可共享库，用其代替已链接的相应二进制库。如本示例所示，在创建可共享库时，请包括 -xa (C) 或 -a （其它编译器）选项。

```
% cc -G -xa -o foo.so.1 foo.o
```

此命令包括可共享库 tcov 分析函数的副本，因此，无需重新链接该库的客户端。如果库的客户端已链接用于分析，那么该客户端所使用的 tcov 函数版本将用于分析可共享库。

锁定文件

tcov 使用一个简单的文件锁定机制来更新 .d 文件中的块覆盖数据库。为达到这个目的，它使用了单一文件 tcov.lock。因此，同一时间应该只有一个用 -xa (C) 或 -a（其它编译器）编译的可执行文件在系统中运行。如果手动终止执行用 -xa （或 -a）选项编译的程序，那么也必须手动删除 tcov.lock 文件。

当程序被链接用于 tcov 分析时，用 -xa 或 -a 选项编译的文件会自动调用分析工具函数。程序退出时，这些函数将在文件 xyz.f （例如）运行时收集的信息与存储在文件 xyz.d 中现有的分析信息混合。为了确保该信息不被同时运行分析二进制的多个人破坏，创建了在 xyz.d 更新期间使用的 xyz.d.lock 锁定文件。如果在打开或读取 xyz.d 或其锁定文件时发生错误，或者在运行时信息和已存储信息之间存在不一致，则不更改存储在 xyz.d 的信息。

如果您编辑并重新编译 xyz.f，那么 xyz.d 中计数器的数目可能会更改。如果运行的是先前分析的二进制文件，则可以检测到这一点。

如果过多人在运行分析后的二进制文件，则部分人将无法获得锁定。数秒延迟后，会显示一条错误消息。已存储的信息不会更新。该锁定在网络中是安全的。因为锁定的执行是基于文件，所以其它文件也可正确更新。

分析函数试图处理无法访问的自动安装文件系统。如果包含覆盖数据文件的文件系统以不同名称安装在不同计算机上，或者运行分析二进制的用户没有写入覆盖数据文件或写入包含该文件目录的权限，那么以上处理失败。确保所有目录统一命名，并且任何想要运行二进制文件的人都可写入。

tcov 运行时函数报告的错误

tcov 运行时函数可能会报告下列错误消息：

- 运行二进制文件的用户缺少读取或写入覆盖数据文件的权限。如果覆盖数据文件已被删除，也会发生这个问题。

```
tcov_exit:Could not open coverage data file 'coverage-data-file-name'
because 'system-error-message-string'.
```

- 运行二进制的用户缺少写入包含覆盖数据文件的目录的权限。如果包含覆盖数据文件的目录未安装在运行二进制文件的计算机上，也会发生上述问题。

```
tcov_exit:Could not write coverage data file 'coverage-data-file-name'
because 'system-error-message-string'.
```

- 过多用户试图在同一时间更新覆盖数据文件。如果计算机在更新覆盖数据文件时已崩溃，也会发生上述问题并留下锁定文件。在崩溃时，两个文件中较长的应作为后崩溃覆盖数据文件使用。手动删除锁定文件。

```
tcov_exit:Failed to create lock file 'lock-file-name' for coverage
data file 'coverage-data-file-name' after 5 tries.Is someone else
running this executable?
```

- 没有可用内存，标准 I/O 软件包将无法工作。在该点，您无法更新覆盖数据文件。

```
tcov_exit:Stdio failure, probably no memory left.
```

- 锁定文件名称比覆盖数据文件名称长 6 个字符。因此，派生的锁定文件名称可能不合法。

```
tcov_exit:Coverage data file path name too long (length
characters) 'coverage-data-file-name'.
```

- 启用 tcov 文件分析的库或二进制文件同时运行、编辑和重新编译。旧版本二进制文件想要某一大小的覆盖数据文件，但编辑操作通常会更改它的大小。如果编译器在旧版本二进制文件试图更新先前覆盖数据文件的同时，创建了一个新的覆盖数据文件，那么该二进制文件可能会看到一个为空或已被破坏的覆盖文件。

```
tcov_exit:Coverage data file 'coverage-data-file-name' is too short.Is
it out of date?
```

将 tcov 增强版用于语句级分析

与原始的 tcov 一样， tcov 增强版提供了关于如何执行程序的逐行信息。它生成一个注释的源文件副本，显示使用了哪些行及其使用频率。它还提供了对基本块信息的汇总。 tcov 增强版可以与 C 和 C++ 源文件一起使用。

tcov 增强版克服了原始的 tcov 的部分缺点。 tcov 增强版所改善的特性有：

- 提供了对 C++ 更完整的支持。
- 它支持在 #include 头文件中找到的代码，并纠正模板类和函数的覆盖数量的缺点。
- 它的运行时比原始 tcov 运行时更有效率。
- 编译器支持的所有平台都支持它。

要使用 tcov 增强版生成带注释的源码：

1. **用 -xprofile=tcov 编译器选项编译您的程序。**

   与 tcov 不同， tcov 增强版在编译期间不生成任何文件。

2. **运行您的程序。**

   然后会创建一个用于存储分析数据的目录，并在该目录中创建名为 tcovd 的单一覆盖数据文件。缺省情况下，该目录会创建在运行程序 *program-name* 的位置，并命名为 *program-name*.profile。该目录也被称为*分析存储桶*。通过环境变量可以更改缺省值（请参阅第 147 页的 “tcov 目录和环境变量”

3. **运行 tcov 生成带注释的源代码。**

   tcov 命令的语法如下所示。

```
% tcov option-list source-file-list
```

其中， *source-file-list* 是源代码文件名列表， *option-list* 是可从 tcov(1) 手册页获取的选项列表。您必须包括 -x 选项以启用 tcov 增强版处理。

tcov 增强版的缺省输出是一组带注释的源文件，其名称是通过将 .tcov 附加到相应的源文件名而派生的。

下面示例说明了 tcov 增强版的语法。

```
% cc -xprofile=tcov -o index.assist index.assist.c
% index.assist
% tcov -x index.assist.profile index.assist.c
```

tcov 增强版的输出与原始 tcov 的输出一致。

为 tcov 增强版创建分析共享库

如下例所示，您可以通过包括 -xprofile=tcov 编译器选项来为 tcov 增强版的使用创建分析共享库。

```
% cc -G -xprofile=tcov -o foo.so.1 foo.o
```

锁定文件

tcov 增强版使用一个简单的文件锁定机制来更新块覆盖数据文件。它将同一目录下创建的单一文件作为 tcovd 文件。该文件的名称为 tcovd.temp.lock。如果手动终止执行为覆盖分析编译的程序，那么也必须手动删除锁定文件。

如果锁定有争用，则该锁定方案执行指数后退。如果 tcov 运行时在五次尝试之后仍然无法获取锁定，则退出。此次运行的数据丢失。在这种情况下，显示下列消息。

```
tcov_exit:temp file exists, is someone else running this
executable?
```

tcov 目录和环境变量

在您为 tcov 编译程序并运行该程序时，运行的程序生成分析存储桶。如果还存在前一分析存储桶，则该程序使用这个分析存储桶。如果不存在分析存储桶，它会创建新的分析存储桶。

分析存储桶指定生成分析输出的所在目录。分析输出的名称和位置由缺省值控制，您可以通过环境变量修改这些缺省值。

注 – tcov 所使用的缺省值和环境变量与用于搜集分析反馈的编译器选项相同：-xprofile=collect 和 -xprofile=use。关于这些编译器选项的更多信息，请参阅相关编译器文档。

缺省分析存储桶以具有 .profile 扩展名的可执行文件命名，并创建在运行该可执行文件的目录下。因此，如果从 /home/userdir 运行名为 /usr/bin/xyz 的程序，则缺省行为是在 /home/userdir 创建一个名为 xyz.profile 的分析存储桶。

UNIX 进程可在程序执行期间更改其当前的工作目录。用于生成分析存储桶的当前工作目录就是程序退出时的当前工作目录。在极少数程序在执行期间确实更改了其当前工作目录的情况下，您可以使用环境变量来控制生成分析存储桶的位置。

您可以设置下列环境变量来修改缺省值：

- SUN_PROFDATA

  可在运行时指定分析存储桶的名称。如果两个变量均被设置，则该变量的值通常被添加到 SUN_PROFDATA_DIR 的值。如果可执行文件的名称与 argv[0] 中的值不同，执行此操作可能有效 （例如，可执行文件的调用是通过一个具有不同名称的符号链接）。

- SUN_PROFDATA_DIR

  可用于指定包含分析存储桶的目录的名称。它由 tcov 命令在运行时使用。

- TCOVDIR

  TCOVDIR 支持为 SUN_PROFDATA_DIR 的同义字，用于维护向下兼容。SUN_PROFDATA_DIR 的任何设置都会导致 TCOVDIR 被忽略。如果 SUN_PROFDATA_DIR 和 TCOVDIR 均已设置，生成分析存储桶时会显示一个警告。

  TCOVDIR 由 tcov 命令在运行时使用。

索引

A

addpath 命令 83
API，收集器 43
arc，调用图，定义 138

B

版本信息
  collect 61
  er_cp 129
  er_mv 130
  er_print 92
  er_rm 130
  er_src 131
包括度量
  递归效果 37
  定义 35
  PLT 指令 104
  如何计算 103
  使用 35
  图示 36
包装器函数 114
编译
  程序分析优化影响 40
  代码行分析 39
  带注释的 [源码] 和 [反汇编] 中的源码 39
  调试符号信息格式 39
  gprof 138
  Java 编程语言 40
  库的静态链接 40
  prof 136
  数据收集的链接 40
  tcov 141
  tcov 增强版 146
  影响数据收集静态链接 40
编译器生成的主体函数
  包括度量的传播 111
  定义 108
  names 108
  由性能分析器显示 116
编译器注释
  定义的类 82
  描述 122
  为 er_print 中列出的注释反汇编码选择 83
  为 er_print 中列出的注释源码选择 82
编译器，访问 15
<Scalar> dataobject 描述符 120
并行执行
  调用序列 109
  指令 108

C

C++ 名称裁减，设置 .er.rc 文件中的缺省库 91
collect 命令
  版本 (-V) 选项 61
  存档 (-A) 选项 60
  dry run (-n) 选项 61
  堆跟踪 (-H) 选项 56
  后跟后续进程 (-F) 选项 57

Java 版本 (-j) 选项 58
记录样本点 (-l) 选项 58
基于时钟的分析 (-p) 选项 55
MPI 跟踪 (-m) 选项 57
冗余 (-v) 选项 61
实验控制选项 57
实验名称 (-o) 选项 60
实验目录 (-d) 选项 59
实验组 (-g) 选项 60
使用 ppgsz 命令 72
输出选项 59
数据收集选项 55
数据限制 (-L) 选项 60
停止 exec (-x) 选项后的目标 59
同步等待跟踪 (-s) 选项 56
选项列表 54
硬件计数器溢出分析 (-h) 选项 55
用其收集数据 54
语法 54
杂项选项 61
暂停和恢复数据记录 (-y) 选项 59
周期抽样 (-S) 选项 57
自述文件显示 (-R) 选项 61
collectorAPI.h 45
CPU
选定的列表，在 er_print 中 85
在 er_print 中选择 87
程序计数器 (PC)，定义 103
程序结构，将调用栈地址映射到 113
程序链接表 (PLT) 104, 127
程序执行
单线程 103
共享对象和函数调用 104
OpenMP 并行 108
所述的调用栈 103
尾调用优化 105
陷阱 104
显式多线程 105
信号处理 104
抽样间隔
定义 34
通过 collect 命令设置 57
在 dbx 设置 64
磁盘空间，实验估计 53
存储需求，实验估计 53

D

dataobject
定义 119
范围 119
在硬件计数器实验中 84
<Scalar> 描述符 120
<Total> 描述符 119
data_objects 命令 84
data_olayout 命令 84
data_osingle 命令 84
dbx
在 MPI 下收集数据 71
在其中运行收集器 61
dbx collector 子命令
archive 65
dbxsample 64
disable 64
enable 64
enable_once （已废弃）66
hwprofile 62
limit 65
pause 65
profile 62
quit （已废弃）66
resume 65
sample 64
sample record 65
show 66
status 66
store 66
store filename （已废弃）66
synctrace 63, 64
打印当前路径 83
单线程程序执行 103
等待时间，*请参阅*同步等待时间
递归函数调用
度量分配到 37
外观，在 OpenMP 程序中 111
地址空间，文本和数据区域 113
调用方与被调用方度量

打印 er_print 中的单一函数 80
属性，定义 35
显示 er_print 中的列表 88
在 er_print 中打印 79
在 er_print 中的排序顺序 80
在 er_print 中选择 79
调用栈
[时间线] 标签中的缺省对齐和深度 91
不完整的解除 112
定义 103
将地址映射到程序结构 113
解除 103
尾调用优化的影响 105
动态编译函数
定义 117
收集器 API 47
度量
包括，*请参阅*包括度量
定时 27
定义 25
堆跟踪 32
关联的影响 100
函数列表，*请参阅*函数列表度量
基于时钟的分析 27, 99
MPI 跟踪 33
内存分配 32
排除，*请参阅*排除度量
属性，*请参阅*属性度量
同步等待跟踪 31
硬件计数器，归属到指令 127
源代码行的解释 121
指令的解释 124
堆跟踪
度量 32
通过 collect 收集数据 56
预装收集器库 68
在 dbx 中收集数据 63
堆栈帧
从陷阱处理程序 105
定义 103
尾调用优化中的重用 105
多线程
并行性指令 108
explicit 105
多线程的应用程序
将收集器附加到 66
执行序列 108

E

er_archive 公用程序 132
er_cp 公用程序 129
er_export 公用程序 133
er_mv 公用程序 130
er_print 公用程序
度量关键字 76
度量列表 74
命令行选项 74
命令，*请参阅* er_print 命令
目的 73
语法 74
er_print 命令
addpath 83
allocs 80
版本 92
cmetric_list 88
cmetrics 79
cpu_list 85
cpu_select 87
csingle 80
csort 80
data_objects 84
data_olayout 84
data_osingle 84
dcc 83
disasm 81
dmetrics 90
dsort 91
调用方与被调用方 79
度量 77
对象 89
exp_list 84
fsingle 78
fsummary 78
gdemangle 91
header 89
help 93
函数 77
javamode 89

leaks 80
limit 89
lines 81
lsummary 81
lwp_list 85
lwp_select 87
mapfile 92
metric_list 88
name 89
object_list 87
object_select 88
overview 89
outfile 88
pcs 81
psummary 81
quit 93
sample_list 85
sample_select 87
scc 82
script 92
setpath 83
sort 78
src 81
statistics 90
sthresh 82, 83
thread_list 85
thread_select 87
tldata 92
tlmode 91
源文件 81
er_rm 公用程序 130
er_src 公用程序 130

F

fast 陷阱 104
Fortran
收集器 API 43
替代的入口点 115
子例程 113
Fortran 函数中替代的入口点 115
反汇编码，带注释
度量格式 122
解释 124
克隆的函数 115
可执行文件的位置 52
描述 123
设置 er_print 中的首选项 83
设置 er_print 中的突出显示阈值 83
硬件计数器度量属性 127
用 er_src 查看 130
在 er_print 中打印 81
指令执行依存 124
方法，*请参阅*函数
非唯一函数名称 114
分析存储桶， tcov 增强版 146, 147
分析共享库，创建
tcov 143
tcov 增强版 146
分析间隔
定义 27
实验大小，影响 53
通过 collect 命令设置 55, 62
通过 dbx collector 设置 62
值的限制 48
分析器，*请参阅*性能分析器
分析文件包
大小 53
基于时钟的数据 98
MPI 跟踪数据 102
同步等待跟踪数据 101
硬件计数器溢出数据 101
分析，定义 26
分支 PC，定义 103
符号表，负载对象 113
负载对象
打印 er_print 中的列表 89
定义 113
符号表 113
函数的地址 114
内容 113
写入布局 84
选定的列表，在 er_print 中 87
在 er_print 中选择 88
复制实验 129

G

gprof
汇总 135
使用 138
输出，解释 138
限制 138
概述数据，在 er_print 中打印 89
高度量值
在注释的反汇编码中 83
在注释的源代码中 82
共享对象，之间的函数调用 104
公用子表达式排除 122
关键字，度量， er_print 公用程序 76
关联，对度量的影响 100

H

函数
@plt 104
包装器 114
地址变化 113
定义 113
动态编译 47, 117
非唯一，名称 114
负载对象内的地址 114
静态，带有重复的名称 114
静态，在剥离共享库中 114
克隆 115
MPI，跟踪 32
内联 115
全局 114
收集器 API 43, 47
<Total> 118
替代的入口点 (Fortran) 115
外联 116
< 未知 > 117
系统库，由收集器插入 42
有别名的 114
主体，编译器生成，*请参阅*主体函数，编译器生成
函数调用
递归，度量分配到 37
共享对象之间 104
输入的，在 OpenMP 程序中 111
在单线程程序中 103
函数列表
排序顺序，在 er_print 中指定 78
在 er_print 中打印 77
函数列表度量
设置 .er.rc 文件中的缺省排序顺序。91
显示 er_print 中的列表 88
选择 .er.rc 文件中的缺省值 90
在 er_print 中选择 77
函数名称， C++
设置 .er.rc 文件中的缺省裁减库。91
选择 er_print 中的长名形式或短名形式 89
后续进程
后跟收集器 50
其数据收集的局限 50
实验名称 52
实验位置 51
为所选后续进程收集数据 66
为所有后跟内容收集数据 57
环境变量
JAVA_PATH 50
JDK_1_4_HOME 50
JDK_HOME 50
LD_LIBRARY_PATH 68
LD_PRELOAD 68
PATH 50
SUN_PROFDATA 147
SUN_PROFDATA_DIR 148
TCOVDIR 141, 148
恢复数据收集
collect 59
从程序 46
在 dbx 65
汇总度量
对于单一函数，在 er_print 中打印 78
对于所有的函数，在 er_print 中打印 78

J

Java
动态编译的方法 47, 117
分析限制 50
内存分配 31

设置 er_print 显示输出 89
显示器 31
javamode 命令 89
JAVA_PATH 环境变量 50
JDK_1_4_HOME 环境变量 50
JJDK_HOME 环境变量 50
JVM 版本 50
基于时钟的分析
定义 27
度量 27, 99
度量的准确性 100
分析包中的数据 98
gethrtime 和 gethrvtime 的比较 100
间隔，*请参阅*分析间隔
开销引起的失真 100
通过 collect 收集数据 55
在 dbx 中收集数据 62
间隔，抽样，*请参阅*抽样间隔
间隔，分析，*请参阅*分析间隔
将负载对象归档到实验 60, 65
将路径附加到文件 83
将收集器附加到正在运行的进程 66
解除调用栈 103
进程地址空间文本和数据区域 113
静态函数
在剥离共享库中 114
重复的名称 114
静态链接，影响数据收集 40

K

克隆的函数 115
库
剥离的共享，和静态函数 114
collectorAPI.h 45
插入 42
静态链接 40
libaio.so 43
libcollector.so 43, 68
libcpc.so 42, 49
libthread.so 42, 105, 106, 109
MPI 42, 69
系统 42

L

LD_LIBRARY_PATH 环境变量 68
LD_PRELOAD 环境变量 68
libaio.so，与数据收集交互 43
libcollector.h 44
做为到收集器的 C 和 C++ 接口 44
做为收集器 Java 编程语言接口的一部分 45
libcollector.so 共享库
预装 68
在程序中使用 43
libcpc.so，使用 49
libfcollector.h 44
LWP
选定的列表，在 er_print 中 85
由线程库创建 105
在 er_print 中选择 87

M

MANPATH 环境变量，设置 16
MPI 程序
附加到 69
实验存储问题 69
实验名称 52, 69, 70
收集数据 69
通过 collect 收集数据 71
通过 dbx 收集数据 71
MPI 跟踪
度量 33
度量的解释 102
分析包中的数据 102
跟踪的函数 32
通过 collect 收集数据 57
预装收集器库 68
在 dbx 中收集数据 64
MPI 实验
存储问题 69
缺省名称 52
移动 70

N

nfs 51
内存分配 32
内存泄漏，定义 32
内联函数 115

O

OpenMP 并行性 108

P

PATH 环境变量 50
PATH 环境变量，设置 16
PC
  从 PLT 104
  定义 103
  在 er_print 中的排序列表 81
@plt 函数 104
PLT （程序链接表）104, 127
ppgsz 命令 72
prof
  汇总 135
  使用 136
  输出 137
  限制 137
排版惯例 14
排除度量
  定义 35
  PLT 指令 104
  如何计算 103
  使用 35
  图示 36
排序顺序
  调用方与被调用方度量，在 er_print 中 80
  函数列表，在 er_print 中指定 78

Q

缺省
  缺省文件中的设置 90

R

入口点，替代的，在 Fortran 函数中 115

S

setpath 命令 83
setuid，使用 43
shell 提示符 15
SUN_PROFDATA_DIR，环境变量 148
SUN_PROFDATA，环境变量 147
删除实验或实验组 130
事件
  [时间线] 标签中的缺省显示类型 91
实验
  从程序中终止 46
  存储位置 59, 66
  存储需求，估计 53
  定义 51
  er_print 中的标题信息 89
  附加当前路径 83
  复制 129
  Java 的设置模式 89
  将负载对象归档到 60, 65
  *另见*实验目录
  MPI 存储问题 69
  命名 51
  缺省名称 51
  删除 130
  设置路径来查找文件 83
  位置 51
  限制其大小 60, 65
  移动 52, 130
  移动 MPI 70
  在 er_print 中列出 84
  组 52
实验名称
  MPI 缺省 52, 70
  MPI，使用 MPI_comm_rank 和脚本 71
  缺省 51
  通过 collect 指定 60
  约束 51
  在 dbx 中指定 66

实验目录
- 缺省 51
- 通过 collect 指定 59
- 在 dbx 中指定 66

实验组
- 定义 52
- 名称约束 52
- 缺省名称 52
- 删除 130
- 通过 collect 指定名称 60
- 在 dbx 中指定名称 66

手册页，访问 15

收集器
- API，在程序中使用 43, 44
- 定义 22, 26
- 附加到正在运行的进程 66
- 使用 collect 运行 54
- 在 dbx 中禁止 64
- 在 dbx 中启用 64
- 在 dbx 中运行 61

输出文件，在 er_print 中 88

数据类型 26
- 堆跟踪 32
- 基于时钟的分析 27
- MPI 跟踪 32
- 缺省，在 [时间线] 标签中 92
- 同步等待跟踪 31
- 硬件计数器溢出分析 28

数据收集
- 程序控制 43
- 从 MPI 程序 69
- 从程序禁止 46
- 从程序中恢复 46
- 从程序中控制 43
- 动态内存分配影响 41
- 段失败 41
- 恢复 collect 59
- 链接 40
- MPI 程序，使用 collect 71
- MPI 程序，使用 dbx 71
- 使用 collect 54
- 使用 dbx 61
- 速度 53
- 在 dbx 中恢复 65
- 在 dbx 中禁止 64
- 在 dbx 中启用 64
- 在 dbx 中暂停 65
- 在程序中暂停 46
- 暂停 collect 59
- 准备程序 41

数据收集内存分配影响 41

数据收集期间段失败 41

输入文件
- 到 er_print 92
- 在 er_print 中终止 93

属性度量
- 递归效果 37
- 定义 35
- 使用 35
- 图示 36

锁定文件管理
- tcov 144
- tcov 增强版 147

T

tcov
- 报告的错误 144
- 带注释的源代码 142
- 分析共享库，创建 143
- 汇总 135
- 使用 141
- 输出，解释 142
- 锁定文件管理 144
- 为其编译程序 141
- 限制 140

tcov 增强版
- 分析存储桶 146, 147
- 分析共享库，创建 146
- 使用 145
- 锁定文件管理 147
- 为其编译程序 146
- 优点 145

TCOVDIR 环境变量 141, 148

TLB （旁路转换缓冲）缺少 105, 127

<Total> dataobject 描述符 119

<Total> 函数
与执行统计比较时间 100
描述 118
同步等待跟踪
等待时间 31, 101
定义 31
度量 31
分析包中的数据 101
通过 collect 收集数据 56
阈值，*请参阅*阈值，同步等待跟踪
预装收集器库 68
在 dbx 中收集数据 63
同步等待时间
带有无界线程 101
定义 31, 101
度量，定义 31
同步延迟事件
定义 31
定义度量 31
分析包中的数据 101

W

<Unknown> 代码行，在注释源代码中 123
<Unknown> 函数
调用方与被调用方 117
PC 的映射 117
外联函数 116
网络磁盘 51
尾调用优化 105
微任务化库例程 108
为实验命名 51
微态
对度量的贡献 99
切换 104
<Unknown> dataobject 描述符
不确定元素 120
未标识的元素 120
未指定的元素 120
无法解析的元素 120
无法验证的元素 120
文档索引 17
文档，访问 17, 18
文件路径 83

X

-xdebugformat
设置调试符号信息格式 39
线程
创建 105
等待模式 111
调度 105, 108
工作线程 105, 108
库 42, 105, 106, 109
系统 100, 109
选定的列表，在 er_print 中 85
有界和无界 105, 112
在 er_print 中选择 87
主 108
陷阱 104
显式多线程 105
限制
分析间隔值 48
后续进程数据收集 50
Java 分析 50
实验名称 51
实验组名称 52
tcov 140
硬件计数器溢出分析 49
限制 er_print 中的输出 89
限制实验大小 60, 65
限制，*请参阅*局限
泄漏，内存
定义 32
信号
对处理程序的调用 104
分析 42
分析，从 dbx 传送到 collect 59
用于 collect 的暂停和恢复 59
用于通过 collect 的手动抽样 58
信号处理程序
用户程序 42
由收集器安装 42, 104

性能度量，*请参阅*度量
性能分析器
  定义 22
性能数据，转换为度量 25
选项，命令行， er_print 公用程序 74

Y

样本
  从程序中记录 45
  定义 34
  记录的条件 34
  间隔，*请参阅*抽样间隔
  通过 collect 进行周期记录 57
  通过 collect 手动记录 58
  选定的列表，在 er_print 中 85
  样本包中包含的信息 34
  在 dbx 停止进程时进行记录 64
  在 dbx 中进行周期记录 64
  在 dbx 中手动记录 65
  在 er_print 中选择 87
样本收集器，*请参阅*收集器
异步 I/O 库，与数据收集交互 43
溢出值，硬件计数器，*请参阅*硬件计数器溢出值
移动实验 52, 130
易读文档 18
硬件计数器
  获取其列表 54, 63
  数据对象和度量 84
  所述的列表 29
  通过 collect 选择 55
  通过 dbx collector 选择 63
  溢出值 28
硬件计数器库， libcpc.so 49
硬件计数器列表
  通过 collect 获取 54
  通过 dbx collector 获取 63
  字段的描述 29
硬件计数器溢出分析
  定义 28
  分析包中的数据 101
  通过 collect 收集数据 55
  通过 dbx 收集数据 62
  限制 49
硬件计数器溢出值
  定义 28
  过小或过大的结果 101
  实验大小，影响 53
  通过 collect 设置 56
  在 dbx 设置 63
映射文件
  用 er_print 生成 92
由 tcov 报告的错误 144
有别名的函数 114
优化
  程序分析影响 40
  公用子表达式排除 122
  尾调用 105
由收集器插入系统库函数 42
语法
  er_archive 公用程序 132
  er_export 公用程序 133
  er_print 公用程序 74
  er_src 公用程序 130
阈值，同步等待跟踪
  定义 31
  对收集开销的影响 101
  通过 collect 命令设置 56, 63
  通过 dbx collector 设置 63
  校准 31
阈值，突出显示
  在注释的反汇编码中， er_print 83
  在注释的源代码中， er_print 82
预装 libcollector.so 68
源代码行
  在 er_print 中的排序列表 81
源代码，带注释
  编译器注释 122
  并行性指令 122
  度量格式 122
  解释 121
  克隆的函数 115
  描述 121
  设置 er_print 中的编译器注释类 82
  设置 er_print 中的突出显示阈值 82
  tcov 142

<Unknown> 代码行 123
用 er_src 查看 130
源代码文件的位置 52
在 er_print 中打印 81
中间文件的使用 112

Z

暂停数据收集
collect 59
从程序 46
在 dbx 65
帧，堆栈，*请参阅*堆栈帧
指令发布
分组，对注释反汇编的影响 124
延迟 126
指令，并行性
度量的属性 122
微任务化库调用 108
执行统计
与 <Total> 函数时间的比较 100
在 er_print 中打印 90
中间文件，注释源码列表的使用 112
注释反汇编码，*请参阅*反汇编码，注释
注释源代码，*请参阅*源代码，注释
主体函数，编译器生成
包括度量的传播 111
定义 108
names 108
由性能分析器显示 116
子例程，*请参阅*函数