性能分析器

Sun™ Studio 11

Sun Microsystems, Inc.
www.sun.com

文件号码 819-4763-10
2005 年 11 月，修订版 A

请将关于本文档的意见和建议提交至：http://www.sun.com/hwdocs/feedback

版权所有 © 2005 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. 保留所有权利。

美国政府权利 — 商业用途。政府用户应遵循 Sun Microsystems, Inc. 的标准许可协议，以及 FAR（Federal Acquisition Regulations，即 “联邦政府采购法规”）的适用条款及其补充条款。必须依据许可证条款使用。

本发行版可能包含由第三方开发的内容。

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

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

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

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

Adobe PostScript

目录

阅读本书之前　15

本书的结构　16

印刷约定　16

支持的平台　17

Shell 提示符　18

访问 Sun Studio 软件和手册页　19

访问 Sun Studio 文档　21

访问相关的 Solaris 文档　24

开发者资源　24

联系 Sun 技术支持　25

Sun 欢迎您提出意见　25

1.　性能分析器概述　27

从集成开发环境启动性能分析器　27

性能分析的工具　27

　收集器工具　28

　性能分析器工具　28

　`er_print` 实用程序　28

性能分析器窗口　29

2. 性能数据 31

收集器收集何种数据 32

时钟数据 32

硬件计数器溢出分析数据 34

同步等待跟踪数据 36

堆跟踪 （内存分配）数据 37

MPI 跟踪数据 38

全局 （抽样）数据 39

将度量分配到程序结构的方法 40

函数级度量：独占、非独占和属性 40

解释属性度量：示例 41

递归如何影响函数级度量 43

3. 收集性能数据 45

编译和链接程序 45

源代码信息 45

静态链接 46

优化 46

编译 Java 程序 46

为数据收集和分析准备程序 47

使用动态分配的内存 47

使用系统库 48

使用信号处理程序 49

使用 `setuid` 49

数据收集的程序控制 49

C、 C++、 Fortran 和 Java API 函数 52

动态函数和模块 53

数据收集的局限 54

基于时钟的分析的局限 54

收集跟踪数据的局限 55

硬件计数器溢出分析的局限 55

硬件计数器溢出分析中的运行时失真和扩大 56

后续进程中数据收集的局限 56

Java 分析的局限 56

用 Java 编程语言编写的应用程序的运行时性能失真和扩大 57

数据存储的位置 57

实验名称 58

移动实验 59

估计存储需求 59

收集数据 60

使用 `collect` 命令收集数据 61

数据收集选项 61

实验控制选项 64

输出选项 67

杂项选项 68

使用 `dbx collector` 子命令收集数据 69

数据收集子命令 69

实验控制子命令 72

输出子命令 73

信息子命令 74

收集运行进程中的数据 74

从 MPI 程序收集数据 77

存储 MPI 实验 77

在 MPI 下运行 `collect` 命令 78

通过在 MPI 下启动 `dbx` 来收集数据 79

与 `ppgsz` 一起使用 `collect` 80

4. 性能分析器工具 81

启动性能分析器 81

分析器选项 82

性能分析器 GUI 83

菜单栏 83

工具栏 84

分析器数据显示 84

设置数据表示选项 91

查找文本和数据 94

显示或隐藏函数 94

过滤数据 94

实验选择 95

样例选择 95

线程选择 95

LWP 选择 95

CPU 选择 95

记录实验 95

生成映射文件和函数顺序重排 96

默认 96

5. 内核分析 99

内核实验 99

为内核分析设置系统 99

运行 `er_kernel` 实用程序 100

分析内核 100

在有负载时分析 101

一起分析内核和负载 101

分析特定的进程或内核线程 102

分析内核性能分析数据 102

6. `er_print` **命令行性能分析工具 103**

`er_print` 语法 104

度量列表 105

控制函数列表的命令 108

控制调用方与被调用方列表的命令 110

控制泄漏和分配列表的命令 112

控制源代码和反汇编列表的命令 113

控制数据空间列表的命令 116

控制内存对象列表的命令 117

列出实验、样例、线程和 LWP 的命令 118

控制实验数据过滤的命令 120

指定过滤表达式 120

选择样例、线程、 LWP 和 CPU 以进行过滤 120

控制负载对象展开和折叠的命令 122

列出度量的命令 123

控制输出的命令 124

输出其他信息的命令 125

设置默认值的命令 126

设置仅用于性能分析器默认的命令 127

杂项命令 129

表达式语法 129

示例 131

7. 理解性能分析器及其数据 133

数据收集如何工作 133

实验格式 133

记录实验 135

解释性能度量 136

基于时钟的分析 136

同步等待跟踪 139
硬件计数器溢出分析 139
堆跟踪 140
数据空间分析 140
MPI 跟踪 140
调用栈和程序执行 141
单线程执行和函数调用 141
显式多线程 144
基于 Java 技术软件执行的概述 144
Java 处理表示 146
OpenMP 软件执行概述 147
不完整的堆栈解除 156
将地址映射到程序结构 157
进程映像 157
负载对象和函数 157
有别名的函数 158
非唯一函数名称 158
源于剥离共享库的静态函数 158
Fortran 替代的入口点 159
克隆的函数 159
内联函数 159
编译器生成的主体函数 160
外联函数 160
动态编译函数 161
<Unknown> 函数 161
新函数和 OpenMP 特殊函数 162
<JVM-System> 函数 162
<no Java callstack recorded> 函数 162

<Truncated-stack> 函数 162

<Total> 函数 163

与硬件计数器溢出分析有关的函数 163

将数据地址映射到程序数据对象 163

数据对象描述符 164

8. 了解带注释的源数据和反汇编数据 167

带注释的源代码 167

性能分析器 “源码” 标签的布局 168

注释反汇编代码 176

解释注释反汇编 177

在源代码、反汇编和 PC 标签中的特殊代码行 180

外联函数 180

编译器生成的主体函数 181

动态编译的函数 182

Java 本机函数 184

克隆的函数 184

静态函数 185

包括度量 186

分支目标 187

在不运行实验的情况下查看源代码 / 反汇编代码 187

9. 操作实验 189

操作实验 189

使用 er_cp 实用程序复制实验 189

使用 er_mv 实用程序移动实验 190

使用 er_rm 实用程序删除实验 190

其他实用程序 190

er_archive 实用程序 190

er_export 实用程序 192

A. 使用 prof、 gprof 和 tcov 来分析程序 193

使用 prof 生成程序分析 194

使用 gprof 生成调用图分析 196

将 tcov 用于语句级分析 198

创建 tcov 的分析共享库 201

锁定文件 202

tcov 运行时函数报告的错误 202

将 tcov 增强版用于语句级分析 203

为 tcov 增强版创建分析共享库 204

锁定文件 205

tcov 目录和环境变量 205

索引 207

图

图 2-1　描述独占、非独占和属性度量的调用树　42

表

表 2-1 Solaris 定时度量 33

表 2-2 同步等待跟踪度量 37

表 2-3 内存分配 （堆跟踪）度量 37

表 2-4 MPI 跟踪度量 38

表 2-5 MPI 函数的类别有发送、接收、发送和接收以及其他 39

表 3-1 `collector_func_load()` 的参数列表 53

表 3-2 预装库 `libcollector.so` 的环境变量设置 76

表 5-1 分析器中内核实验的字段标签含义 102

表 6-1 度量类型字符 105

表 6-2 度量可视性字符 106

表 6-3 度量名称字符串 107

表 6-4 编译器注释消息类 114

表 6-5 `dcc` 命令的附加选项 115

表 6-6 时间线显示模式选项 128

表 6-7 时间线显示数据类型 128

表 7-1 数据类型和相应的文件名称 134

表 7-2 内核微态如何贡献给基值 137

表 8-1 注释源代码度量 175

表 A-1 性能分析工具 193

阅读本书之前

本手册描述 Sun™ Studio 11 软件中的性能分析工具。

- 收集器和性能分析器这一对工具用于执行大范围性能数据的统计分析以及跟踪各种系统调用，并在函数、源代码行和指令级将这些数据与程序结构相关联。
- `prof` 和 `gprof` 是执行 CPU 使用率的统计分析并在函数级提供执行频率的工具。
- `tcov` 是在函数和源代码行级别提供执行频率的工具。

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

本书的结构

第 1 章 介绍了性能分析工具，其中简要介绍了这些工具的用途以及何时使用这些工具。

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 描述了如何启动性能分析器以及如何使用该工具来分析收集器收集的性能数据。

第 5 章描述了如何在 Solaris 操作系统运行负载的同时使用 Sun Studio 性能工具来分析内核。

第 6 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 7 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 8 章 描述了如何使用和了解性能分析器的源代码和反汇编窗口中的信息。

第 9 章 介绍了关于实用程序的信息，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。

印刷约定

表 P-1 字体约定

字体[1]	**含义**	**示例**
AaBbCc123	命令、文件和目录的名称；计算机屏幕输出。	编辑 .login 文件。 使用 ls -a 列出所有文件。 % You have mail.
AaBbCc123	用户键入的内容，与计算机屏幕输出的显示不同。	% **su** Password:

表 P-1 字体约定

字体[1]	含义	示例
AaBbCc123	保留未译的新词或术语以及要强调的词。要使用实名或值替换的命令行变量。	这些称为 *class* 选项。 要删除文件，请键入 **rm** *filename*。
新词术语强调	新词或术语以及要强调的词。	您**必须**成为超级用户才能执行此操作。
《书名》	书名	阅读 《用户指南》的第 6 章。

1 浏览器的设置可能会与这些设置不同。

表 P-2 代码约定

代码符号	含义	表示法	代码示例
[ ]	方括号中包含可选参数。	`O[`*n*`]`	`-O4， -O`
{ }	花括号中包含所需选项的选项集合。	`d{y|n}`	`-dy`
|	分隔变量的 “|” 或 “-” 符号，只能选择其一。	`B{dynamic|static}`	`-Bstatic`
:	与逗号一样，分号有时可用于分隔参数。	`R`*dir*`[:`*dir*`]`	`-R/local/libs:/U/a`
…	省略号表示一系列的省略。	`-xinline=`*f1*`[,`*…fn*`]`	`-xinline=alpha,dos`

支持的平台

此 Sun Studio 发行版本支持使用 SPARC® 和 x86 系列处理器体系结构 (UltraSPARC®、SPARC64、 AMD64、 Pentium 和 Xeon EM64T) 的系统。通过访问 `http://www.sun.com/bigadmin/hcl` 中的硬件兼容性列表，可以了解您在使用的 Solaris 操作系统版本的支持系统。这些文档列出了实现各个平台类型的所有差别。

在本文档中，这些与 x86 有关的术语具有以下含义：

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

有关所支持的系统，请参见硬件兼容性列表。

Shell 提示符

Shell	提示符
C shell	*machine-name*%
C shell 超级用户	*machine-name*#
Bourne shell、 Korn shell 和 GNU Bourne-Again shell	$
Bourne shell、 Korn shell 和 GNU Bourne-Again shell 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未安装到 /usr/bin/ 和 /usr/share/man 标准目录中。要访问编译器和工具，必须正确设置 PATH 环境变量（请参见第 19 页的 "访问软件"）。要访问手册页，必须正确设置 MANPATH 环境变量（请参见第 20 页的 "访问手册页"）。

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

注 – 本节中的信息假设 Sun Studio 编译器和工具安装在 Solaris 平台上的 /opt 目录中和 Linux 平台上的 /opt/sun 目录中。如果未将软件安装在 Solaris 平台的 /opt 目录和默认目录中，请询问系统管理员以获取系统中的相应路径。

访问软件

使用以下步骤决定是否需要更改 PATH 变量以访问该编译器和工具。

决定是否需要设置 PATH 环境变量

1. 通过在命令提示符后键入以下内容以显示 PATH 变量的当前值。

```
% echo $PATH
```

2. 在 **Solaris** 平台上，查看输出中是否包含有 /opt/SUNWspro/bin/ 的路径字符串。在 **Linux** 平台上，查看输出中是否包含有 /opt/sun/sunstudio11/bin 的路径字符串。

   如果找到该路径，则说明已设置了访问该编译器和工具的 PATH 变量。如果没有找到该路径，则需要按照下一步中的说明来设置 PATH 环境变量。

设置 PATH 环境变量以实现对编译器和工具的访问

1. 如果使用的是 **C shell**，请编辑 **Home** 目录的 .cshrc 文件。如果使用的是 **Bourne shell** 或 **Korn shel**，请编辑 **Home** 目录的 .profile 文件。

2. 在 **Solaris** 平台上，将以下路径添加到 PATH 环境变量中。如果以前安装了 **Forte Developer** 软件、**Sun ONE Studio** 软件，或其他发行版本的 **Sun Studio** 软件，则将以下路径添加到这些软件安装路径之前：

/opt/SUNWspro/bin

在 **Linux** 平台上，将以下路径添加到 PATH 环境变量中：

/opt/sun/sunstudio11/bin

访问手册页

使用以下步骤决定是否需要更改 MANPATH 变量以访问手册页。

决定是否需要设置 MANPATH 环境变量

1. 通过在命令提示符后键入以下内容以请求 collect **手册页。**

```
% man collect
```

2. 请查看输出 （如果有）。

如果找不到 collect(1) 手册页或者显示的手册页不是软件当前版本的手册页，请按照下一步的说明来设置 MANPATH 环境变量。

设置 MANPATH 环境变量以实现对手册页的访问

- **在 Solaris 平台上，将以下路径添加到** MANPATH **环境变量中：**

  /opt/SUNWspro/man

- **在 Linux 平台上，将以下路径添加到** MANPATH **环境变量中：**

  /opt/sun/sunstudio11/man

访问集成开发环境

Sun Studio 集成开发环境 (integrated development environment, IDE) 提供了创建、编辑、生成、调试 C、 C++、 Java 或 Fortran 应用程序并分析其性能的模块。

启动 IDE 的命令是 sunstudio。有关该命令的详细信息，请参见 sunstudio(1) 手册页。

IDE 是否可以正确操作取决于 IDE 能否找到核心平台。 sunstudio 命令会查找两个位置的核心平台：

- 该命令首先查找 Solaris 平台上的默认安装目录 /opt/netbeans/3.5V11 和 Linux 平台上的默认安装目录 /opt/sun/netbeans/3.5V11。

- 如果该命令在默认目录中找不到核心平台，则它会假设包含 IDE 的目录和包含核心平台的目录均安装在同一位置上。例如，在 Solaris 平台上，如果包含 IDE 的目录的路径是 /foo/SUNWspro，则该命令会在 /foo/netbeans/3.5V11 中查找核心平台。在 Linux 平台上，如果包含 IDE 的目录的路径是 /foo/sunstudio11，则该命令会在 /foo/netbeans/3.5V11 中查找核心平台。

如果核心平台未安装在 sunstudio 命令查找它的任一位置上，则客户端系统上的每个用户必须将环境变量 SPRO_NETBEANS_HOME 设置为安装核心平台的位置 (/*installation_directory*/netbeans/3.5V11)。

在 Solaris 平台上，IDE 的每个用户还必须将 /*installation_directory*/SUNWspro/bin 添加到其他任何 Forte Developer 软件、 Sun ONE Studio 软件或 Sun Studio 软件发行版本路径前面的 $PATH 中。

在 Linux 平台上，IDE 的每个用户还必须将 /*installation_directory*/sunstudio11/bin 添加到其他任何发行版本的 Sun Studio 软件路径前面的 $PATH 中。

路径 /*installation_directory*/netbeans/3.5V11/bin 不能添加到用户的 $PATH 中。

访问 Sun Studio 文档

您可以访问以下位置的文档：

- 可以通过随软件一起安装在本地系统或网络上的文档索引获取文档，位置为 Solaris 平台上的 file:/opt/SUNWspro/docs/zh/index.html 和 Linux 平台上的 file:/opt/sun/sunstudio11/docs/zh/index.html。

  如果未将软件安装在 Solaris 平台上的 /opt 目录中或 Linux 平台的 /opt/sun 目录中，请询问系统管理员以获取系统中的相应路径。
- 大多数的手册都可以从 docs.sun.com[sm] Web 站点获取。以下书目只能从所安装的软件中获取：
  - 《标准 C++ 库类参考》
  - 《标准 C++ 库用户指南》
  - 《Tools.h++ 类库参考》
  - 《Tools.h++ 用户指南》
- 发行说明可从 docs.sun.com Web 站点获取。
- 在 IDE 中通过“帮助”菜单以及许多窗口和对话框上的“帮助”按钮，可以访问 IDE 的所有组件的联机帮助。

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

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

使用易读格式的文档

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述，找到文档的易读版本。如果未将软件安装在 /opt 目录中，请询问系统管理员以获取系统中的相应路径。

文档类型	易读版本的格式和位置
手册 （第三方手册除外）	HTML，位于 http://docs.sun.com
第三方手册： • 《标准 C++ 库类参考》 • 《标准 C++ 库用户指南》 • 《Tools.h++ 类库参考》 • 《Tools.h++ 用户指南》	HTML，位于 Solaris 平台上所安装软件中的文档索引 file:/opt/SUNWspro/docs/zh/index.html
自述文件	HTML，位于开发者门户网站 http://developers.sun.com/prodtech/cc/documentation/ss11/docs/mr/READMEs
手册页	HTML，位于安装的软件上的文档索引，位置为 Solaris 平台上的 file:/opt/SUNWspro/docs/zh/index.html 和 Linux 平台上的 file:/opt/sun/sunstudio11/docs/zh/index.html。
联机帮助	HTML，可通过 IDE 或分析器 GUI 中的 “帮助”菜单和 “帮助”按钮访问
发行说明	HTML，位于 http://docs.sun.com

相关编译器和工具文档

对于 Solaris 平台，下表介绍的相关文档可以从 /opt/SUNWspro/docs/zh/index.html 和 http://docs.sun.com 上获取。如果未将软件安装在 /opt 目录中，请询问系统管理员以获取系统中的相应路径。

文档标题	描述
《C 用户指南》	描述了 Sun Studio 11 C 编程语言编译器和 ANSI C 编译器特定信息。
《C++ 用户指南》	指导您使用 Sun Studio 11 C++ 编译器，并提供有关命令行编译器选项的详细信息。
《Fortran 用户指南》	描述 Sun Studio 11 Fortran 编译器的编译时环境以及命令行选项。
《OpenMP API 用户指南》	有关实现程序并行化的编译器指令的信息。
《Fortran 编程指南》	介绍并行性、优化、创建共享库等编程技术。
《使用 dbx 调试程序》	描述了使用 dbx 命令行调试器的方法。提供有关附加和分离进程以及在受控环境中执行程序的信息。
性能分析器自述文件	列出性能分析器的新增功能、已知问题、限制和不兼容性。
analyzer(1)、collect(1)、collector(1)、er_kernel(1)、er_print(1)、er_src(1) 和 libcollector(3) 手册页	描述性能分析器命令行实用程序

对于 Linux 平台，下表描述的相关文档可以在 http://docs.sun.com 上的 file:/opt/sun/sunstudio11/docs/zh/index.html 上获取。如果未将软件安装在 /opt/sun 目录中，请询问系统管理员以获取系统中的相应路径。

文档标题	描述
性能分析器自述文件	列出性能分析器的新增功能、已知问题、限制和不兼容性。
analyzer(1)、collect(1)、collector(1)、er_print(1)、er_src(1) 和 libcollector(3) 手册页	描述了性能分析器命令行实用程序
《使用 dbx 调试程》序	描述了使用 dbx 命令行调试器的方法。提供有关附加和分离进程以及在受控环境中执行程序的信息。

访问相关的 Solaris 文档

下表描述了可从 docs.sun.com Web 站点上获取的相关文档。

文档集合	文档标题	描述
Solaris 参考手册集合	请参见手册页部分的标题。	提供有关 Solaris 操作系统的信息。
Solaris 软件开发者集合	《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 软件开发者集合	《多线程编程指南》	涵盖 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 软件开发者集合	《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》	提供有关 Solaris 可调参数的参考信息。

开发者资源

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

- 有关编程技术和最佳实例的文章
- 有关编程小技巧的知识库
- 有关编译器和工具组件的文档以及与软件安装在一起的文档的更正信息
- 有关支持级别的信息
- 用户论坛
- 可下载的代码样例
- 新技术预览

您可以通过访问 http://developers.sun.com 找到其他开发者资源。

联系 Sun 技术支持

如果您遇到通过本文档无法解决的技术问题，请访问以下网址：

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

Sun 欢迎您提出意见

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

`http://www.sun.com/hwdocs/feedback`

请在您的反馈信息中注明文档的文件号码 (819-4763-10)。

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

有关从集成开发环境 (IDE) 启动性能分析器的信息，请参见 “性能分析器自述文件” （可通过访问文档索引 /*installation_directory*/docs/zh/index.html 获取）。 Solaris 平台上的默认安装目录为 /opt/SUNWspro。 Linux 平台上的默认安装目录为 /opt/sun/sunstudio11。如果 Sun Studio 11 编译器和工具未安装在 /opt 目录下，请向系统管理员获取系统中的相应路径。

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者，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?
- 程序在执行过程中是如何出现这种问题的?
- 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见第 2 章，关于收集器的详细信息请参见第 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

性能分析器工具可以显示收集器记录的数据，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：

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

性能分析器还可以以图形格式显示作为时间函数的原始数据。性能分析器可以创建映射文件，以更改函数在程序地址空间中装入函数的顺序，从而提高性能。

关于性能分析器的详细信息，请参见第 4 章和 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，请参见第 6 章。

第 5 章描述如何在 Solaris™ 操作系统 (Solaris OS) 上运行负载的同时使用 Sun Studio 性能工具分析内核。

第 7 章介绍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` 实用程序来查看 （详细信息请参见第 9 章）。

第 8 章有助于了解已注释的源代码和反汇编代码，提供有关性能分析器显示的各种类型的索引行和编译器注释的解释。

第 9 章描述如何复制、移动、删除、归档和导出实验。

`er_print` 实用程序

`er_print` 实用程序以纯文本显示了除 “时间线”之外性能分析器显示的所有内容。

性能分析器窗口

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

性能分析器窗口由带有菜单栏和工具栏的多标签化显示组成。性能分析器启动时显示的标签显示了程序的函数列表，该程序具有每个函数的独占和非独占度量。该列表可以按负载对象、线程、轻量进程 (LWP)、 CPU 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。可以使用此标签导航调用树，例如搜索较大的度量值。

另外两个标签中，一个标签显示用性能度量逐行注释的并与编译器注释交叉的源代码，另一个标签显示用每个指令的度量注释的并与可用的源代码和编译器注释交叉的反汇编代码。

性能数据在另一个标签中显示为时间函数。

其他标签显示实验和负载对象的详细信息，函数和内存泄露的汇总信息，以及进程的统计数据。

使用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，将这些数据转换为程序性能的度量 （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种，所以术语 “收集器”指这些工具中的任一种。同样，因为分析性能数据的工具也有很多种，所以术语分析工具也指这些工具中的任一种。

本章涵盖了以下主题。

- 收集器收集何种数据
- 将度量分配到程序结构的方法

有关收集和存储性能数据的信息，请参见第 3 章。

有关用性能分析器分析性能数据的信息，请参见第 4 章。

有关在 Solaris OS 运行负载时分析内核的信息，请参见第 5 章。

有关使用 `er_print` 实用程序分析性能数据的信息，请参见第 6 章

收集器收集何种数据

收集器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可以收集分析数据。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。间隔时间结束后会将一个信号传递到系统，数据将在下一个间隔被记录。
- 跟踪数据是通过干预不同系统函数上的包装器函数收集的，因此可以截断对系统函数的调用并记录调用的数据。
- 通过调用不同的系统例程来获得信息可以收集全局数据。全局数据包称为样例。

分析数据和跟踪数据都包含特定事件的信息，这两种类型的数据会转换成性能度量。全局数据不会转换为度量，其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。

每个分析事件或跟踪事件收集的数据包都包括以下信息：

- 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 (CPU) ID，操作系统中可用时
- 调用栈的副本。对于 Java 程序，记录两个调用栈：机器调用栈和 Java 调用栈。

关于线程和轻量进程的详细信息，请参见第 7 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 （仅限 Solaris 操作系统）
- 同步等待跟踪数据 （仅限 Solaris 操作系统）
- 堆跟踪 （内存分配）数据
- MPI 跟踪数据 （仅限 Solaris 操作系统）

在以下子章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。第六种数据类型 （全局样例数据），无法转换为度量，因为它不包含调用栈信息。

时钟数据

进行基于时钟的分析时，收集的数据取决于操作系统提供的度量。

Solaris 操作系统下基于时钟的分析

在 Solaris 操作系统下基于时钟的分析中，每个 LWP 的状态在固定的时间间隔都会被存储。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。默认分析时间间隔约为 10 毫秒 (10 ms)。收集器提供的高分辨率分析时间间隔约为 1 毫秒，低分辨率分析时间间隔约为 100 毫秒，而操作系统允许设置任意的时间间隔。不带参数运行 `collect` 命令将输出在其上运行命令的系统所允许的范围和分辨率。

下表中定义了从基于时钟的数据计算得来的度量。

表 2-1 Solaris 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，通常称为 “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。
其他等待时间	等待内核页所用的 LWP 时间，或休眠 / 停止所用的时间。

对于多线程的实验，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。

- 高级用户 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。
- 高级系统 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。
- 高级等待 CPU 时间说明了准备运行的线程数比可用的 CPU 多，或其他进程正在使用 CPU。
- 高级用户锁定时间说明线程无法获得请求的锁定。
- 高级文本缺页时间意味着链接程序生成的代码在内存中会被组织起来，从而调用或分支生成要装入的新页。创建和使用映射文件（请参见性能分析器联机帮助中的 “生成和使用映射文件”）可以修复这种问题。
- 高级数据缺页时间表明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。

Linux 操作系统下基于时钟的分析

在 Linux 操作系统下，唯一可用的度量是 “用户 CPU” 时间。虽然报告的总 CPU 占用时间是准确的，但是对于 Solaris 操作系统，分析器不可能准确确定实际系统 CPU 时间的时间比例。虽然分析器显示的信息像是轻量进程 (LWP) 数据，但实际上在 Linux 操作系统中没有 LWP 的数据；显示的 LWP ID 实际是线程 ID。

硬件计数器溢出分析数据

硬件计数器可跟踪诸如缓存丢失、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，运行 LWP 的 CPU 的指定硬件计数器溢出时收集器将记录分析包。计数器被重置并继续计数。分析包中包括溢出值和计数器类型。

各种 CPU 系列支持的同步硬件计数器寄存器的数量从两个到十八个不等。收集器可以收集一个或多个寄存器上的数据。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。有些硬件计数器可以使用任意寄存器，而有些计数器仅可以使用特定的寄存器。因此，在一个实验中并非可以选择所有的硬件计数器组合。

硬件计数器溢出分析数据由性能分析器转换成计数度量。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。下一子章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，甚至缺页。

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 “失控”，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。如果处理器和系统支持硬件计数器分析，collect 命令会打印两个包含硬件计数器信息的列表。第一个列表包含 “已知”（有别名的）硬件计数器；第二个列表包含原始硬件计数器。

以下示例显示了计数器列表中的计数器条目。被认为是已知的计数器将首先显示在列表中，然后是原始硬件计数器列表。该例中的每一行输出都按打印格式显示。

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

已知硬件计数器列表的格式

在已知硬件计数器列表中，第一个字段 （例如 cycles）是可以在 collect 命令的 -h counter... 参数中使用的别名。该别名还是在 er_print 命令中使用的标识符。

第二个字段列出计数器的可用寄存器，例如 [/{0|1}]。对于已知计数器，选择的默认值可提供合理的抽样率。由于实际抽样率变化相当大，因此可能需要您指定默认值以外的值。

第三个字段 （例如 9999991）是计数器的默认溢出值。

第四个字段 （在圆括号中）包含类型信息。它提供简短描述 （例如 CPU Cycles）、原始硬件计数器名称 （例如 Cycle_cnt）和计数单位类型 （例如 CPU-cycles），该字段最多可以包括两个单词。

如果类型信息的第一个单词是：

- load、 store 或 load-store，则表明计数器与内存相关。您可以在 collect -h 命令中的计数器名称前放置 + 号 （例如 +dcrm），以请求搜索引发事件的准确指令和虚拟地址。 + 号还可以启用数据空间分析；有关详细信息，请参见第 87 页的 “DataObjects 标签”、第 88 页的“DataLayout 标签”和第 88 页的“MemoryObjects 标签”。
- not-program-related，计数器会捕获由其他某个程序启动的事件，如 CPU 到 CPU 的缓存嗅探。使用计数器进行分析将生成警告，而且分析不记录调用栈。

如果类型信息的第二个单词或仅有的单词是：

- CPU-cycles，则可以使用计数器提供基于时间的度量。为这样的计数器报告的度量在默认情况下会转换为非独占和独占时间，但是可以有选择地显示为事件计数。
- events，则度量是非独占和独占事件计数，且无法转换为时间。

在示例的已知硬件计数器列表中，类型信息包含一个单词的，如第一个计数器的 CPU-cycles ；第二个计数器的 events。类型信息包含两个单词的，如第三个计数器的 load events。

原始硬件计数器列表的格式

原始硬件计数器列表中包括的信息是已知硬件计数器列表中信息的子集。每行包括由 cpu-track(1) 使用的内部计数器名称、可以在其上使用计数器的寄存器编号、默认溢出值和计数器单位 （它可以是 CPU-cycles 或 Events）。

如果计数器度量与运行的程序无关的事件，则类型信息的第一个单词是 not-program-related。对于这样的计数器，分析不会记录调用栈，而显示人工函数 collector_not_program_related 中所用的时间。线程和 LWP ID 会被记录，但不具有任何意义。

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

同步等待跟踪数据

在多线程程序中，不同线程执行的任务同步会使应用程序的执行延迟，例如，一个线程要访问已被其他线程锁定的数据时就不得不等待。这些事件称为同步延迟事件，并通过跟踪对 Solaris 或 pthread 线程函数的调用来收集。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。目前，只能在运行 Solaris 操作系统的系统中进行同步等待跟踪。

只有等待时间超过阈值 （单位为微秒）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。默认阈值由运行校准测试决定，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，机器调用栈和 Java 调用栈都会被收集；但对于 Java™ 虚拟机 (JVM)[1] 软件中使用的内部锁定，不会收集任何同步跟踪数据。在机器表示中，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。

1. （术语 “Java 虚拟机”和 "JVM" 是指用于 Java(TM) 平台的虚拟机。）

同步等待跟踪数据被转换成以下度量：

表 2-2 同步等待跟踪度量

度量	定义
同步延迟事件。	对等待时间超过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。

从该信息您可以确定函数或负载对象调用同步例程时是会经常被阻塞还是会长时间等待。高级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

如果调用未正确管理的内存分配和释放函数可能会导致数据的使用效率降低，并导致应用程序的性能降低。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` 跟踪内存分配和释放请求。对 `mmap` 的调用视为内存分配，它允许记录有关 Java 内存分配的堆跟踪事件。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。

性能分析器不支持对 Java 程序的堆分析。

堆跟踪数据被转换成以下度量：

表 2-3 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量
分配的字节	每次调用内存分配函数时分配的字节总数。
泄漏	调用内存分配函数 （该函数对释放函数不作相应的调用）的数量。
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。

内存泄漏的另一个常用定义 （如在 `dbx` 调试工具中）是：内存泄漏是一个动态分配的内存块，在程序的数据空间中没有任何指向它的指针。这里所用的泄漏定义既包括这种替换的定义，还包括存在指针的内存的定义。

MPI 跟踪数据

收集器可以收集调用消息传递接口 (MPI) 库的数据。目前，只能在运行 Solaris 操作系统的系统中使用 MPI 跟踪。下面列出了用于收集数据的函数。

MPI_Allgather	MPI_Allgatherv	MPI_Allreduce
MPI_Alltoall	MPI_Alltoallv	MPI_Barrier
MPI_Bcast	MPI_Bsend	MPI_Gather
MPI_Gatherv	MPI_Irecv	MPI_Isend
MPI_Recv	MPI_Reduce	MPI_Reduce_scatter
MPI_Rsend	MPI_Scan	MPI_Scatter
MPI_Scatterv	MPI_Send	MPI_Sendrecv
MPI_Sendrecv_replace	MPI_Ssend	MPI_Wait
MPI_Waitall	MPI_Waitany	MPI_Waitsome
MPI_Win_fence	MPI_Win_lock	

MPI 跟踪数据被转换成以下度量：

表 2-4 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
其他 MPI 调用	对其他 MPI 函数的调用数量

接收或发送时记录的字节数为调用中给定的缓冲大小。此数字可能比接收或发送的实际字节数大。在全局通信函数和集合通信函数中，假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，则发送或接收的字节数是最大值。

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

表 2-5 MPI 函数的类别有发送、接收、发送和接收以及其他

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
其他 MPI 函数	MPI_Barrier、MPI_Wait、MPI_Waitall、MPI_Waitany、 MPI_Waitsome、 MPI_Win_fence、 MPI_Win_lock

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

全局 （抽样）数据

全局数据由收集器按名为样例包的包来记录。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。记录在样例包中的数据对程序来说是全局的，且不转换为性能度量。记录样例包的过程称为抽样。

当发生以下情况时，样例包会被记录下来：

- 如果断点停止的选项被设置，而程序在 IDE 的“调试”窗口或 dbx 中因任何原因（如在断点处）停止时
- 如果在抽样间隔结束时已选择了周期性抽样。抽样间隔被指定为一个以秒为单位的整数。默认值为 1 秒
- 选择了 “调试” → “性能工具箱” → “启用收集器”，并在 “收集器” 窗口选中 “周期性抽样” 复选框时
- 使用 dbx collector sample record 命令手动记录样例时
- 如果在调用 collector_sample 前已将对该例程的调用放到代码中 （请参见第 49 页的 “数据收集的程序控制”）时
- 如果在指定传递信号前已将 -l 选项和 collect 命令一起使用（请参见 collect(1) 手册页）时
- 开始和终止收集时

- 使用 dbx collector pause 命令暂停收集 （就在暂停之前）和使用 dbx collector resume 命令恢复收集时 （就在恢复之后）
- 后续进程创建前后

性能工具使用记录在样例包中的数据，按时间周期将数据分组，这称为样例。通过选择一组样例可以过滤特定事件的数据，使您只看到特定时间周期的信息。此外您还可以查看每个样例的全局数据。

性能工具消除了不同种类样例点之间的区别。要利用样例点进行分析，您应该只选择一种要记录的点。具体说就是，如果要记录与程序结构或执行序列有关的样例点，您则应该关闭周期性抽样，并且在 dbx 停止进程，或将信号传递到正使用 collect 命令记录数据的进程，或调用收集器 API 函数时使用记录的样例。

将度量分配到程序结构的方法

使用与特定事件的数据一起记录的调用栈将度量分配到程序指令。如果该信息可用，则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见第 7 章。

除了源代码和指令，度量还被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：独占、非独占和属性

"性能分析器"计算三种类型的函数级度量：独占度量、非独占度量和属性度量。

- 函数的独占度量由函数本身内部发生的事件计算得来。这种度量不包括来自其他函数调用的度量。
- 非独占度量由函数本身和其调用的函数内部发生的事件计算得来。这种度量包括来自对其他函数调用的度量。
- 属性度量说明了来自对 （或从）另外一个函数的调用的非独占度量数目：这种度量归属对另外一个函数的度量。

对于只出现在调用栈底部的函数 （叶函数），独占度量和非独占度量是相同的。

对于负载对象来说也要计算独占和非独占度量。负载对象的独占度量通过累加负载对象中所有函数上函数级别的度量计算得来。负载对象的非独占度量与函数非独占度量的计算方法相同。

函数的独占和非独占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函数非独占度量数目取决于每个调用方的调用。调用方的属性度量将累计到函数的非独占度量。
- 函数被调用方的属性度量说明函数非独占度量数目取决于每个被调用方的调用。它们的总和加上函数的独占度量等于函数的非独占度量。

对调用方或被调用方的属性和非独占度量进行比较，可以得到详细信息，如下所示：

- 调用方的属性度量和非独占度量之间差额说明了来自对其他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非独占度量之间的差额说明了来自对其他函数调用的被调用方非独占度量数目。

要定位能改善程序性能的位置，请执行以下操作：

- 使用独占度量定位具有高度量值的函数。
- 使用非独占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

将在包含调用树碎片的图 2-1 中描述独占、非独占和属性度量。其中焦点是中心函数和函数 C。示例中可能存在对其他未在该图中显示的函数的调用。

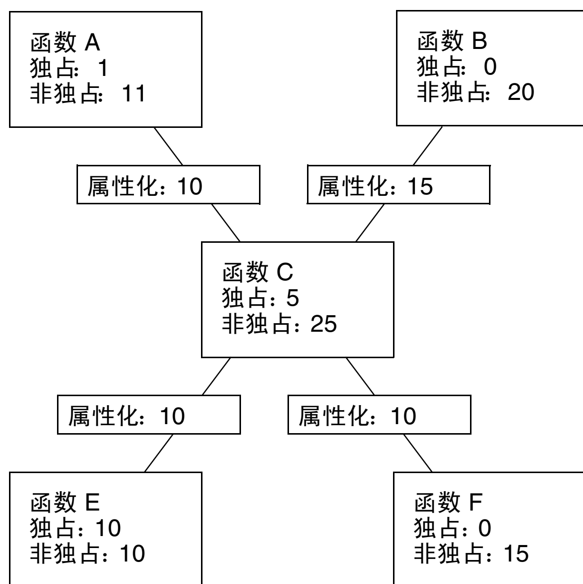

图 2-1 描述独占、非独占和属性度量的调用树

函数 C 调用了两个函数，函数 E 和函数 F，并分别将 10 个单位的非独占度量归属到函数 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 的独占度量 (5) 等于函数 C 的非独占度量 (25)。

函数 E 的被调用方属性度量和被调用方非独占度量是相同的，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 由其他的一个或多个函数调用。函数 E 的独占度量和非独占度量是相同的，而函数 F 的这两种度量是不同的。这意味着函数 F 可以调用其他的函数，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 个单位的非独占度量归属到函数 A，将 15 个单位的非独占度量归属到函数 B。这些是调用方的属性度量。它们的总和 (10+15) 等于函数 C 的非独占度量。

调用方属性度量等于函数 A 非独占度量和独占度量之间的差额，不等于函数 B 非独占度量和独占度量之间的差额。这意味着函数 A 仅调用函数 C，而函数 B 除了调用函数 C 外还调用其他函数。（实际上，函数 A 也可以调用其他函数，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 （叶函数）底部的函数计算的独占度量，但会影响非独占和属性度量。

非独占度量是通过将事件的独占度量增加到调用栈中函数的非独占度量来计算。为了确保在递归调用栈中不重复计算度量，事件的度量仅增加到每个唯一函数的非独占度量。

属性度量从非独占度量计算。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，则递归函数的非独占度量会被归属到本身，而不是初始化函数。之所以发生此归属，是因为递归函数的所有更高调用的递归度量被视为零，避免了重复计算度量。但是，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）会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。有关数据本身的详细信息，请参见第 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

您可以为几乎使用任何选项编译的程序收集和分析数据，但部分选项会影响性能分析器中收集或查看的内容。以下子章节中将描述在编译和链接程序时应考虑的问题。

源代码信息

要查看带注释的“源码”和“反汇编”分析中的源代码及“行”分析中的源代码行，则必须使用 `-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® 处理器的 C 编译器和 C++ 编译器指定），则要通过指定 -xhwcprof -xdebugformat=dwarf 和任何级别的优化进行编译。（目前，这种功能在未经过优化的情况下是无法使用的。）要查看 “数据对象” 分析中的程序数据对象，还要增加 -g（或者，对于 C++，增加 -g0）获取全部符号信息。

用 DWARF 格式的调试符号生成的可执行文件和库，会自动包括每个要素目标文件调试符号的副本。用 stabs 格式的调试符号生成的可执行文件和库，也包括每个要素目标文件调试符号的副本（如果 stabs 格式的调试符号与将 stabs 符号保留在各种目标文件及可执行文件中的 -xs 选项相链接）。在您需要移动或删除目标文件时，此信息的包括尤为重要。有了可执行文件和库自身的所有调试符号，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，不得禁止 -dn 和 -Bstatic 编译器选项执行的动态链接。如果试图为完全静态链接的程序收集数据，收集器会输出一条错误消息并且不收集数据。出现该错误的原因是在运行收集器时该收集器库在其他库之间已动态装入。

请勿静态链接任何系统库。如果您执行了静态链接，可能将无法收集任何类型的跟踪数据。另外，请勿链接收集器库 libcollector.so。

优化

如果使用某一级别的优化编译程序，编译器可能会重新安排执行顺序，这样它将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代码行相关联。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。有关详细信息，请参见第 143 页的 “尾调用优化”。

编译 Java 程序

用 javac 命令编译 Java 程序无需任何特殊操作。

为数据收集和分析准备程序

对大多数程序来讲，您无需为数据收集和分析做特殊准备。如果程序执行以下操作之一，您则应该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，还应阅读相关子章节。

使用动态分配的内存

许多程序依赖动态分配的内存，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他值。在这些情况下，性能工具不会出现错误。依赖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，这些潜在的错误会导致发生意外行为。

以下工具可以帮助您找到这些潜在的错误：

- `f95 -xcheck=init_local`

  有关详细信息，请参见 《Fortran 用户指南》或 `f95`(1) 手册页

- lint

  有关详细信息，请参见 《C 用户指南》或 `lint`(1) 手册页

- dbx 下的运行时检查

  有关详细信息，请参见 《使用 dbx 调试程序》手册或 dbx(1) 手册页。
- Purify

使用系统库

收集器可以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。下表将描述收集器插入对库函数调用的情况。

- 收集同步等待跟踪数据。收集器插入 Solaris 线程库 libthread.so[1]（在 Solaris 8 操作系统和 Solaris 9 操作系统上）和 Solaris C 库 libc.so（在 Solaris 10 操作系统上）的函数。同步等待跟踪在 Linux 操作系统上不可用。
- 收集堆跟踪数据。收集器插入函数 malloc、 realloc、 memalign 和 free。这些函数的版本可在 C 标准库 libc.so，以及其他库，如 libmalloc.so 和 libmtmalloc.so 中找到。
- 收集 MPI 跟踪数据。收集器插入 Solaris MPI 库 libmpi.so 的函数中。在 Linux 下无法使用 MPI 跟踪。
- 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) 及其变体中。对 fork1 的调用内部替换了对 vfork 的调用。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。

插入在以下情况中会失败：

- 通过任何包含被插入函数的库静态链接程序。
- 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。

1. 进行分析时， Solaris 操作系统 8 上的默认线程库 /usr/lib/libthread.so （称为 T1）存在一些问题。当 LWP 上没有调度任何线程时，它可能丢弃分析中断；这种情况下，报告的“全部 LWP 时间”将严重低估真正的 LWP 时间。某些情况下，在访问内部库互斥时也可能出现段故障，导致应用程序崩溃。解决方法是通过将 /usr/lib/lwp 置于 LD_LIBRARY_PATH 设置的前面，利用可替换线程库 （/usr/lib/lwp/libthread.so，称为 T2） 。在 Solaris 9 上，默认库为 T2，该库已归入 libc 库中。

使用信号处理程序

收集器使用两种信号来收集分析数据：SIGPROF（适用于所有实验）和 SIGEMT（只适用于硬件计数实验）。对于每一个信号，收集器都安装一个相应的信号处理程序。信号处理程序截取并处理它自己的信号，但是将其他信号传递给任何其他已安装的信号处理程序。如果程序将其自身的信号处理程序安装在这些信号上，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，以及记录样例。尽管用户处理程序安装时将警告写入实验，但这些信号不受收集器保护。确保收集器对指定信号的使用和相同信号应用程序的任何使用之间没有冲突是用户的责任。

由收集器安装的信号处理程序会设置一个确保系统调用不被信号传送中断的标志。如果程序的信号处理程序将该标志设置为允许中断系统调用，则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。如果已经安装收集器库 libcollector.so，取消信号将总是因到达太晚而不能取消异步 I/O 操作。

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，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了系统调用，程序行为和分析行为都将与预装收集器库时不同。

使用 setuid

动态加载器实施的限制使得难以使用 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 还包含一些可以使用的 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 51 页的 “Java 接口” 中将描述由 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，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 还提供了与 C 和 C++ API 相同的特性。

要使用 Fortran 的 API 函数，请插入以下语句：

```
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* 是安装 Sun Studio 软件的目录。

```
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 C 和 C++ API 相同的函数。

未收集数据时， C 包括含不使用对实 API 函数调用的宏的 libcollector.h 文件。在这种情况下，不要动态装入函数。但是，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 会较为安全，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，要收集性能数据则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，则应该确保它们只被一个线程调用。除 collector_thread_pause() 和 collector_thread_resume() 外，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，如果一个线程在其他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。有两种可行方法使用以上函数：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。除此之外的其他任何用法都可能产生不可预知的结果。

C、C++、Fortran 和 Java API 函数

API 函数的具体描述如下。

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

  Fortran：`collector_sample(string name)`

  Java：`CollectorAPI.sample(String name)`

  记录样例包并用指定的名称或字符串标记该样例。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。

  样例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 函数记录样例时发生另一调用，则该函数可确保只写入一个样例。所记录的样例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例进行区分。如果只想查看 API 调用所记录的样例，就应在记录性能数据时关闭所有其他样例模式。

- **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，且可以继续写入全局数据。如果没有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。如果没有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 t)`

  终止将变量列表中特定线程的事件特定数据写入实验。参数 t 对于 C/C++ 程序来说是 POSIX 线程标识符，对于 Java 程序来说是 Java 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。默认情况下，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 t)`

  恢复将变量列表中特定线程的事件特定数据写入实验。参数 `t` 对于 C/C++ 程序来说是 POSIX 线程标识符，对于 Java 程序来说是 Java 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，则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 Java HotSpot™ 虚拟机编译的 Java 方法中，因为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，但不包括注释源代码列表。

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。可以为 `NULL`。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`	到构造函数所在源文件的路径。可以为 `NULL`。该源文件用于注释的源代码列表。
`vaddr`	函数的装入地址。

表 3-1 collector_func_load() 的参数列表 *（续）*

参数	定义
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。偏移必须按照数值递增的顺序，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，但是没有可用的函数源代码列表。

- 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性，这些局限性是由硬件、操作系统、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他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键入不带参数的 collect 命令来查找。

在 Solaris 8 操作系统的早期版本中，会使用系统时钟进行分析。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，可以通过将以下行添加到文件 /etc/system 并重新启动来执行该操作。

```
set hires_tick=1
```

在 Solaris 9 操作系统和 Solaris 10 操作系统以及较早版本的 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。

基于时钟分析过程中的运行时失真和扩大

基于时钟的分析会记录 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，信号越频繁，扩大越显著。在有限的范围内，基于时钟的分析会产生失真，这是由程序中这些执行最深栈部分的显著扩大引起的。

如有可能，请不要将默认值设置为精确的毫秒数，而是设置为比精确值稍大或稍小 （例如 10.007 ms 或 0.997 ms）的值，以避免与系统时钟关联 （这也可能使数据失真）。在 SPARC 平台上以相同的方法设置自定义值 （而在 Linux 平台上是不可能的）。

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。有关详细信息，请参见第 74 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

跟踪数据以被跟踪事件的数量成比例地扩大运行。如果完成基于时钟的分析，则跟踪事件引起的扩大会使时钟数据失真。

硬件计数器溢出分析的局限

硬件计数器溢出分析具有以下几个限制条件：

- 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。在其他系统中，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溢出分析。
- 使用早于 Solaris 8 发行版本的 Solaris 操作系统版本无法收集硬件计数器溢出数据。
- 在 `cpustat`(1) 运行过程中，无法收集系统的硬件计数器溢出数据，原因是 `cpustat` 控制了这些计数器，且不允许用户进程使用。如果在数据收集期间启动 `cpustat`，则硬件计数器溢出分析将终止，并在实验中记录错误。
- 如果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。如果调用不是来自收集器，则收集器插入 `libcpc` 库函数并返回为 `-1` 的返回值。
- 如果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` 命令。

硬件计数器溢出分析中的运行时失真和扩大

硬件计数器溢出分析会记录 SIGEMT 传递到目标时的数据。这将导致处理该信号和调用栈回溯的扩大。与基于时钟的分析不同的是，对于某些硬件计数器来说，程序的不同部分可能会比其他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他线程不成比例的线程中生成。

后续进程中数据收集的局限

您必须遵循以下条件限制才能搜集有关后续进程中的数据：

- 如果您要在后跟收集器的所有后续进程中收集数据，则必须用含 `-F on` 或 `-F` 所有选项的 `collect` 命令。`-F` 选项设置为 `on` 时，您可以自动收集对 `fork` 及其变体的调用数据以及对 `exec` 及其变体的调用数据。在 `-F` 选项设置为 `all` 时，收集器会跟在所有后续进程的后面，包括那些由于对 `system`、 `popen` 和 `sh` 进行调用而产生的进程。
- 如果要为单独的后续进程收集数据，则必须将 `dbx` 附加到该进程。有关详细信息，请参见第 74 页的 "收集运行进程中的数据"。
- 如果要为单独的后续进程收集数据，必须将单独的 `dbx` 附加到每个进程，并启用收集器。但是请注意，在 `dbx` 下目前不支持 Java 程序的数据收集。

Java 分析的局限

您必须遵循以下条件限制才能搜集有关 Java 程序的数据：

- 您应该使用版本不低于 1.4.2_02 的 Java™ 2 Platform 或 Standard Edition (J2SE)。不要使用可能会崩溃的 1.4.2 或 1.4.2_01 版本。默认情况下， `collect` 命令使用 Sun Studio 安装程序安装 J2SE 的路径 （如果有的话）。您可以通过设置 `JDK_HOME` 环境变量或 `JAVA_PATH` 环境变量覆盖此默认路径。收集器会验证它在这些环境变量中找到的 `java` 版本是否为 ELF 可执行文件，如果不是，则输出一条错误消息，指出已使用的环境变量和尝试的完整路径名。
- 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。
- 不能对创建运行 JVM 软件的后续进程的应用程序进行分析。
- 如果要使用 64 位 JVM 软件，必须使用 `-j on` 标志，并将 64 位 JVM 软件指定为目标。不要使用 `java -d64` 来收集使用 64 位 JVM 软件的数据。如果您这样做，将不会收集到任何数据。
- 非 Java 应用程序可以动态打开 `libjvm.so` 并将其传递给 Java 类。支持分析这种应用程序的解决方法是，在最初收集调用时不要将 `-j` 设置为 `on`，并且确保将 `-Xruncollector` 选项传递给调用的 JVM 软件。

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 软件可能会崩溃。
- **JVM 1.4.2**：数据收集期间该版本的 JVM 软件可能会崩溃。
- **JVM 1.4.1**：会正确纪录并显示 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 软件所提供的代码区域的名称显示。由于某些 JVM 软件所创建的代码区没有命名，所以 `<Unknown>` 函数中会显示大量时间。此外，JVM 1.4.1 软件中的各种错误可能会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。
- **JVM 1.4.0 之前的版本**：早于 1.4.0 版本的 JVM 软件不支持分析 Java 应用程序。

用 Java 编程语言编写的应用程序的运行时性能失真和扩大

如果运行的是 J2SE 1.4.2，则 Java 分析使用 Java™ 虚拟机分析接口 (JVMPI)；如果运行的是 J2SE 5.0，则使用 Java™ 虚拟机工具接口 (JVMTI) （这两种情况可能会导致运行的失真和扩大）。

对于基于时钟的分析和硬件计数器溢出分析，数据收集进程会对 JVM 软件进行各种调用，并处理信号处理程序中的分析事件。这些例程的系统开销和将实验写入磁盘所用的时间将扩大 Java 程序的运行时。这样的扩大通常小于 10%。

虽然默认的垃圾收集器支持 JVMPI，但还存在其他不支持 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时，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。大多数 Java 应用程序都会生成许多上述事件，这将扩大实验，并在处理数据时出现可伸缩性问题。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，这将导致更多的 CPU 时间用于较长的分配路径。

对同步跟踪来讲，数据收集使用了其他 JVMTI 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

应用程序在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。这些归档文件包括负载对象中每个目标文件和函数的地址、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。

实验以默认方式存储在当前目录中。如果该目录位于网络文件系统，则存储数据所用的时间比在本地文件系统中所用的时间要长，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指定存储位置。

后续进程的实验存储在创建进程的实验的内部。

实验名称

新实验的默认名称为 `test.1.er`。后缀 `.er` 是强制的：如果所付给的名称不带有该后缀，则显示一条错误消息且不接受该名称。

如果选择使用 *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 1 — 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*，直到找到未使用 *n* 值的实验名称为止。如果实验名称不含 *n* 且实验存在，则收集器会输出一条错误消息。

实验可按组收集。组是在实验组文件中定义，并以默认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，并在下面的行中显示实验名称。实验组文件的默认名称为 `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，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中。文件的名称必须以 `.erg` 结尾。

默认实验名称与在 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。默认实验名称为 `test.`*m*`.er`，其中 *m* 为进程的 MPI 级别。如果指定了实验组 *group*`.erg`，则默认实验名称为 *group*`.`*m*`.er`。如果指定了实验名称，则该名称覆盖默认值。有关详细信息，请参见第 77 页的 “从 MPI 程序收集数据”。

后续进程的实验是其沿袭命名的，如后面的示例。要组成后续进程的实验名称，可将下划线、代码字母和数字添加到它所创建的实验名称中。代码字母 `f` 表示 fork，x 表示 exec，c 表示 combination。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`，则由第三次调用 `fork` 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

如果要将实验移动到其他计算机并对其进行分析，则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，如果要查看带注释的源代码或带注释的反汇编代码，则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。

性能分析器在以下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搜索。

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

使用分析器 GUI 或使用 `setpath` 和 `addpath` 指令，可以更改搜索顺序或增加其他搜索目录。

为确保您在程序中查看到正确的带注释源代码和带注释反汇编代码，可以在移动或复制实验之前将源代码、目标文件和可执行文件复制到该实验。如果您不想复制目标文件，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，会有两个相关的调用栈：Java 调用栈和机器调用栈，因此将导致更多的数据写入磁盘。

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。为了获得统计数据和降低开销之间较好的折衷办法，所以精心选择了分析间隔和溢出值的默认值。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 记录数据的速度为 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，它通常占总磁盘空间需求的一小部分（请参见前一节）。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和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，请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，可以考虑收集部分运行中的数据，而不是全部。要收集部分运行中的数据，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

可以用多种方式在独立性能分析器或 IDE 的分析器窗口中收集性能数据：

- 在命令行使用 `collect` 命令（请参见第 61 页的“使用 `collect` 命令收集数据”和 `collect(1)` 手册页）。`collect` 命令行工具比 `dbx` 或 IDE 调试器中的“收集器”对话框具有更小的数据收集开销，所以此方法比其他方法要好。
- 使用性能分析器的“性能工具收集”对话框（请参见性能分析器联机帮助中“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（请参见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（请参见第 69 页的“使用 `dbx collector` 子命令收集数据”和 IDE 的“调试”联机帮助中的“收集器命令”部分）

以下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，则收集的信息属于 JVM 软件，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，请键入以下内容。

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

其中，*collect-options* 是 collect 命令选项，*program* 是要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，则以分析间隔约为 10 毫秒的基于时钟的分析为默认值。

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，请键入不带参数的 collect 命令。

```
% collect
```

对硬件计数器列表的描述，请参见第 34 页的 “硬件计数器溢出分析数据”。另请参见第 55 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，请参见第 32 页的“收集器收集何种数据”。

如果不指定数据收集选项，则默认值为 -p on，该默认值启用默认分析间隔约为 10 毫秒的基于时钟的分析。 -h 选项可以关闭该默认值，但任何其他数据收集选项不能关闭该默认值。

如果基于时钟的分析被明确禁用，也未启用任何类型的跟踪或硬件计数器溢出分析，则 collect 命令将输出一条警告消息，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 – 打开默认分析间隔约为 10 毫秒的基于时钟的分析。
- lo[w] – 打开低分辨率分析间隔约为 100 毫秒的基于时钟的分析。

- hi[gh] – 打开高分辨率分析间隔约为 1 毫秒的基于时钟的分析。在 Solaris 8 操作系统中，必须明确启用高分辨率分析。关于启用高分辨率分析的详细信息，请参见第 54 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* 的默认单位为毫秒。您可以将 *value* 指定为整数或浮点数。可在数值后加后缀 m 选择以毫秒为单位，或后加 u 选择以微秒为单位。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，则会输出一条警告消息，并将其设置为时钟分辨率。

收集基于时钟的分析数据是 collect 命令的默认操作。

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

收集硬件计数器溢出分析数据。计数器定义的数值与处理器相关。如果安装了 perfctr 修补程序（可以从 http://user.it.uu.se/~mikpe/linux/perfctr/2.6/perfctr-2.6.15.tar.gz 下载），则在运行 Linux 操作系统的系统上，此选项是可用的。

计数器定义可以采用以下形式之一，具体采用哪一种形式取决于处理器是否支持硬件计数器的属性。

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

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

处理器特定的 *counter_name* 可以是以下任一项：

- 已知的（有别名的）计数器名称
- 类似 cputrack(1) 使用的原始（内部）计数器名称。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 来指定要使用的事件寄存器。

如果指定多个计数器，则它们必须使用不同的寄存器。如果未使用不同的寄存器，collect 命令会输出一条错误消息，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，请在终端窗口中键入没有参数的 collect。在第 35 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，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。此回溯适用于 SPARC 处理器，且只能与类型为 load、store 或 load-store 的计数器一起使用。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。

在一些处理器上，属性选项可以与硬件计数器关联。如果处理器支持属性选项，则不带参数运行 collect 命令将列出包含属性名称的计数器定义。可以按十进制或十六进制格式指定属性值。

间隔 （溢出值）是在硬件计数器溢出和记录溢出事件的位置计数的事件数目。可以将间隔设置为以下值之一：

- on 或空字符串 – 默认的溢出值，可以通过键入不带参数的 collect 来确定。
- hi[gh] – 所选计数器的高分辨率值，约为默认溢出值的十分之一。缩写 h 还支持与先前软件发行版本的兼容。
- lo[w] – 所选计数器的低分辨率值，约为默认溢出值的十倍。
- *interval* – 特定溢出值，必须为正整数，可以为十进制或十六进制格式。

默认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。另请参见第 55 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，则必须指定 -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默认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。

堆跟踪的默认值为关闭。 Java 程序不支持堆跟踪；如果指定它，则将被视为错误。

-m *option*

收集 MPI 跟踪数据。 *option* 的允许值包括：

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

MPI 跟踪的默认值为关闭。

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详细信息，请参见第 38 页的 “MPI 跟踪数据”。

`-S` *option*

周期性地记录样例包。 *option* 的允许值包括：

- `off` – 关闭周期性抽样。
- `on` – 打开默认抽样间隔为 1 秒的周期性抽样。
- *value* – 打开周期性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。

默认情况下，启用间隔为 1 秒的周期性抽样。

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` – 只记录由函数 `fork`、 `exec` 及其变体创建的后续进程中的实验。
- `all` – 记录所有后续进程中的实验。
- `off` – 不记录后续进程中的实验。

如果指定 `-F on` 选项，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 `fork1` 的调用内部替换了对 `vfork` 的调用。

如果指定 `-F all` 选项，收集器跟随所有后续进程，包括那些由 `system` （3C）、 `system`(3F)、 `sh`(3F) 和 `popen` （3C）以及类似函数的调用创建的后续进程，以及与它们关联的后续进程。

如果指定 `-F on` 或 `-F all` 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。可通过将下划线、字母和数字添加到实验后缀来命名这些新的实验，如下所示：

- 其中，字母 "f" 表示 fork， "x" 表示 exec， "c" 则表示任何其他后续进程。
- 数字是 fork 或 exec （无论是否成功）或其他调用的索引。

例如，如果初始进程的实验名称是 `test.1.er`，则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`。如果该子进程使用 `popen` 调用创建了另一个进程，则相应实验名称为 `test.1.er/_f3_x1_c1.er`。

读取创建实验时分析器和 `er_print` 实用程序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。

要通过命令行选择要显示的数据，请将路径名明确指定为 er_print 或 analyzer。指定的路径必须包含创建实验名称以及该创建目录中的后续实验名称。

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。

要在分析器中检查后续进程，请装入创建实验并从 “视图”菜单中选择过滤 数据。只有创建实验处于选中状态才能显示实验列表。取消选中它，并选中相关的后续实验。

-j *option*

当目标为 JVM 机器时，启用 Java 分析。 *option* 的允许值包括：

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

假如 java 可执行文件的路径在 JDK_HOME 环境变量或 JAVA_PATH 环境变量中，如果要收集 .class 文件或 .jar 文件的数据，则不需要 -j 选项。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达 java 的路径，或想禁止由 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，该虚拟机的系统版本不能是 1.4.2_02 之前的版本。 collect 命令会验证 *program* 是否为 JVM 机器，以及是否为 ELF 可执行文件；如果不是， collect 命令会输出错误消息。

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。如果您这样做，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。

-J *java_arguments*

指定要传递到用于分析的 JVM 机器的参数。如果指定 -J 选项，而不指定 Java 分析，则会生成错误，且不运行实验。

`-l` *signal*

当名为 *signal* 的信号传递到进程时，记录样例包。

可以通过信号全名、不带大写字母 `SIG` 的信号名或信号编号来指定信号。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，则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，则应该确保用 `-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。

有关信号的详细信息，请参见 `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）。样例点始终记录，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，则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，收集器会在记录状态下启动，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，请确保用 `-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。

有关信号的详细信息，请参见 `signal`(3HEAD) 手册页。

输出选项

-o *experiment_name*

将 *experiment_name* 作为要记录实验的名称。*experiment_name* 字符串必须以字符串 ".er" 结尾；否则， collect 实用程序会输出错误消息并退出。

-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， collect 实用程序会输出一条错误消息，然后退出。如果通过 -g 选项指定组，还会将组文件写入 *directory-name*。

-g *group-name*

使实验成为实验组 *group-name* 的一部分。如果 *group-name* 不以 .erg 结尾，则 collect 实用程序输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，实验组位于目录 *directory-name* 下，否则就放在当前目录下。

-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，请指定 -A copy。使用该选项不会将任何源文件或目标文件复制到实验中。您应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，但不适用于样例点。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。如果启用了周期性抽样，可继续写入样例点。

对记录数据的默认限制为 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`。

`-O` *file*

将 `collect` 自身的输出附加到名称 *file* 结尾，但是不重定向产生的目标的输出。如果文件设为 `/dev/null`，则禁止所有 `collect` 的输出，包括所有错误消息。

杂项选项

`-C` *comment*

将 *comment* 放入实验的 `notes` 文件。最多可以提供十个 -C 选项。该 notes 文件的内容将置于实验标题的前面。

`-n`

不运行目标，但输出在目标运行情况下要生成实验的详细信息。该选项是一个 dry run 选项。

`-R`

在终端窗口显示 “性能分析器自述文件”的文本格式文件。如果未找到自述文件，则输出警告。不再检查任何参数，也不执行进一步的处理。

`-V`

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

`-v`

输出 `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

注 – 只有在运行 Solaris 操作系统的系统上，才能使用 dbx collector 子命令收集数据。

从 dbx 运行收集器：

1. **通过键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
```

要获取有效的 collector 子命令类型的列表，请键入：

```
(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，则会输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 – 启用默认分析间隔为 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 – 使用约为 10 毫秒的默认分析间隔。
  - lo[w] – 使用约为 100 毫秒的低分辨率分析间隔。
  - hi[gh] – 使用约为 1 毫秒的高分辨率分析间隔。在 Solaris 8 操作系统的早期版本中，必须明确指定启用高分辨率分析。关于启用高分辨率分析的详细信息，请参见第 54 页的 “基于时钟的分析的局限”。
  - *value* – 将分析间隔设置为 *value*。*value* 的默认单位为毫秒。您可以将 *value* 指定为整数或浮点数。可在数值后加后缀 m 选择以毫秒为单位，或后加 u 选择以微秒为单位。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。以上两种情况均输出警告消息。

  默认设置约为 10 毫秒。

收集器在默认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。默认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见第 35 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 *counter_definition*...[, *counter_definition*] - 计数器定义可以采用以下形式之一，具体采用哪种形式取决于处理器是否支持硬件计数器的属性。

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

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

  选择该硬件计数器*名称*，并将其溢出值设置为 *interval*； （可选）选择其他硬件计数器名称并将其溢出值设置为指定的间隔。溢出值可能是以下值之一。
  - on 或空字符串 – 默认的溢出值，可以通过键入不带参数的 collect 确定。
  - hi[gh] – 所选计数器的高分辨率值，约为默认溢出值的十分之一。缩写 h 还支持与先前软件发行版本的兼容。
  - lo[w] – 所选计数器的低分辨率值，约为默认溢出值的十倍。

- *interval* – 特定溢出值，必须是正整数，可以为十进制或十六进制格式。

  如果指定多个计数器，则它们必须使用不同的寄存器。如果使用了同一寄存器，则会输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。

默认情况下，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。

另请参见第 55 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` – 启用具有默认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默认阈值。（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。

默认情况下，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。

默认情况下，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。

默认情况下，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` – 启用周期性抽样。
- `manual` – 禁止周期性抽样。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。

默认情况下，周期性抽样启用，其抽样间隔 *value* 为 1 秒。

`dbxsample` { `on` | `off` }

控制 `dbx` 停止目标进程时对样例的记录。关键字的含义如下：

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

默认情况下，在 `dbx` 停止目标进程时记录样例。

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，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，则会有警告表明它被忽略。如果没有运行中的进程，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，它将启用数据收集并启动新的实验。如果进程正在运行而数据收集被启用，则会有警告表明它被忽略。如果没有运行中的进程，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，但保持实验处于打开状态。收集器暂停时，不记录样例点。暂停数据收集之前会生成一个样例，恢复数据收集之后则会立即生成另一个样例。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* 的样例包。该标签显示在性能分析器的 “事件”标签中。

输出子命令

以下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目标文件复制到实验中。

`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，但不适用于样例点。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，且继续记录样例点。

对记录数据的默认限制为 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被忽略。*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详细信息，请参见第 57 页的 “数据存储的位置”
- `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。如果该组已存在，则实验会增加到其中。如果已使用 `store directory` 子命令设置了目录名称，并且组名称不是绝对路径，则该组名称使用目录名作为前缀。

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 – 有关从 IDE 启动性能分析器的信息，请参见 “性能分析器自述文件”，该文件可以从 SDN Sun Studio 门户 （其网址为 `http://developers.sun.com/prodtech/cc/documentation/ss11/docs/mr/READMEs/analyzer.html`）获取。

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，则必须将 `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，请执行以下操作：

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，您可以通过键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”并使用对话框选择进程。相关指示请使用联机帮助。
- 从 dbx，请键入 以下命令。

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

如果 dbx 还未运行，请键入以下命令。

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

关于附加到进程的详细信息，请参见 《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“收集”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，键入以下命令。

```
(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他系统库调用，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。有关收集器如何插入系统库函数的详细信息，请参见第 48 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/prod/lib/dbxruntime
LD_LIBRARY_PATH_32	/opt/SUNWspro/prod/lib/dbxruntime
LD_LIBRARY_PATH_64	/opt/SUNWspro/prod/lib/v9/dbxruntime
LD_LIBRARY_PATH_64	/opt/SUNWspro/prod/lib/amd64/dbxruntime

如果 Sun Studio 软件未安装在 /opt/SUNWspro 中，请向系统管理员咨询正确的路径。可以在 LD_PRELOAD 中设置全路径，但在使用 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 启动的其他程序就不生效。

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。但是将 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，这将影响 MPI 进程的同步。另请参见第 77 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 (5.0)，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 群运行环境 (CRE) 命令 `mprun`。有关详细信息，请参见 Sun HPC ClusterTools 文档。关于 MPI 和 MPI 标准的信息，参见 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章节中介绍。

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。后续子章节中将介绍这些选项。

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注意有关存储 MPI 实验的一些问题。这些涉及的问题包括数据收集、存储的效率以及实验的命名。关于命名实验（包括 MPI 实验）的信息请参见第 57 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。在可以使用该目录之前，所有其他 MPI 进程必须等待，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* 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，假设在具有四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，请键入以下命令。

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

对于大量的 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见第 189 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，请使用 df -lk 命令或咨询系统管理员。请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他实验。另请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见第 59 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，请参见第 57 页的 “数据存储的位置” 一节。

要确保不同 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，请参见第 77 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 或 -F all 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不关注。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。可以通过在 test.1.er/_x1_f1_x3.er 上调用分析器或 er_print 实用程序，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

性能分析器工具

性能分析器是一种图形数据分析工具，用于分析收集器使用 `collect` 命令、IDE 或 `dbx` 中的 `collector` 命令收集的性能数据。收集器利用收集的性能信息来创建进程执行的实验，如第 3 章中所述。性能服务器读取此类实验、分析数据，并在表格和图形显示中显示数据。以 `er_print` 实用程序的形式提供了分析器的命令行版本，如第 6 章中所述。

启动性能分析器

要启动性能分析器，请在命令行上键入以下内容：

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

也可以在 IDE 中使用浏览器导航到实验并打开该实验。*experiment-list* 命令参数是以空格分隔的实验名称、实验组名称或两者的列表。

您可以在命令行上指定多个实验或实验组。如果指定其中包含后续实验的实验，会自动加载所有这些后续实验，但会禁止显示后续实验的数据。要加载单个后续实验，必须显式指定每个实验或创建实验组。要创建实验组，请创建一个纯文本文件，该文件的首行为：

```
#analyzer experiment group
```

然后将实验名称添加到下面的行。文件扩展名必须为 `erg`。

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

分析器显示多个实验时，无论这些实验是如何加载的，都会聚集所有实验中的数据。

只需在“打开实验”对话框或“添加实验”对话框中单击要加载的实验或实验组的名称，即可预览该实验或实验组。

您还需要从命令行启动性能分析器以记录实验，如下所示：

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

分析器启动时会出现“性能工具收集”窗口，该窗口显示命名的目标、其参数及收集实验的设置。详细信息，请参见第 95 页的“记录实验”。

分析器选项

这些选项控制分析器的行为，它们划分为三个组：

- Java 选项
- 控制选项
- 信息选项

Java 选项

-j | --jdkhome *jvm-path*

指定运行分析器的 JVM 软件的路径。最先检查 JVM 路径的环境变量首先采用默认路径，按照先 JDK_HOME 再 JAVA_PATH 的顺序。如果这两个环境变量均未设置，则默认路径为 Sun Studio 安装程序安装 Java™2 软件开发工具包的位置；如果未安装软件开发工具包，则按用户的 PATH 设置。

-J *jvm-options*

指定 JVM 选项。

控制选项

-f | --fontsize *size*

指定在分析器 GUI 中使用的字体大小。

-v | --verbose

在开始之前打印版本信息和 Java 运行时参数。

信息选项

这些选项不调用性能分析器 GUI，但会将分析器的有关信息打印到标准输出。以下各个选项都是独立选项；它们既不能与其他分析器选项结合使用，也不能与目标或实验列表参数结合使用。

-V | --version

在开始之前打印版本信息和 Java 运行时参数。

-? | --h | --help

打印用法信息并退出。

性能分析器 GUI

分析器窗口具有菜单栏、工具栏，以及包含用于各种数据显示的标签的拆分窗格。

菜单栏

菜单栏包含 “文件”菜单、“视图”菜单、“时间线”菜单和 “帮助”菜单。

使用 “文件”菜单可以打开、添加和删除实验及实验组。利用 “文件”菜单可以使用性能分析器 GUI 收集实验的数据。有关使用性能分析器收集数据的详细信息，参见第 95 页的 “记录实验”。在 “文件”菜单中，还可以创建映射文件。映射文件可用于优化可执行文件的大小或优化其有效的高速缓存行为。有关映射文件的详细信息，参见第 96 页的 “生成映射文件和函数顺序重排”。

使用 “视图”菜单可以配置如何显示实验数据。

如名称所示，“时间线”菜单可帮助您浏览时间线显示，第 84 页的 “分析器数据显示”中有相关描述。

“帮助”菜单提供性能分析器的联机帮助、新功能概述、快速参考和快捷方式部分，以及疑难解答部分。

工具栏

工具栏提供了各组图标作为菜单快捷方式，并且包含“查找”功能，以帮助您在标签中查找文本或突出显示的行。有关“查找”功能的详细信息，参见第 94 页的“查找文本和数据”

分析器数据显示

性能分析器使用拆分窗口将数据表示拆分为两个窗格。每个窗格都有标签，以允许您为同一实验或实验组选择不同的数据显示。

数据显示，左窗格

左窗格包含主要分析器显示的标签：

- “函数”标签
- “调用方与被调用方”标签
- “源码”标签
- “行”标签
- “反汇编”标签
- PC 标签
- DataLayout 标签
- DataObjects 标签
- 各种 MemoryObjects 标签
- “时间线”标签
- “泄漏列表”标签
- “统计”标签
- “实验”标签

调用分析器时如果没有目标，会提示您打开实验。

默认情况下，第一个可见标签处于选中状态。仅显示适用于已加载实验中数据的标签。

打开实验时是否在分析器的左窗格中显示某个标签，取决于启动分析器时读取的 `.er.rc` 文件中的 tabs 指令以及该标签是否适合实验中的数据。您可以使用“设置数据表示”对话框中的“标签”标签（请参见第 93 页的“标签标签”）选择要为实验显示的标签。

函数标签

“函数”标签显示包含功能及其度量的列表。度量派生于实验中收集的数据。度量可以是独占度量或非独占度量。独占度量表示函数本身中的用法。非独占度量表示函数内部及其调用的所有函数的用法。

collect(1) 手册页中列出了每一种收集的数据的可用度量。但列出的只是具有非零度量的函数。

时间度量以秒为单位，可以精确到毫秒。百分比精确到 0.01%。如果度量值正好是 0，则其时间和百分比显示为 "0."。如果值不是 0，但小于精度，则其值显示为 "0.000"，其百分比显示为 "0.00"。由于进行了舍入，可能百分比的总和不会正好等于 100%。计数度量显示为整数计数。

度量最初基于收集的数据以及从不同的 .er.rc 文件读取的默认设置进行显示。最初安装性能分析器时，默认值显示如下：

- 对于基于时钟的分析，默认集合包含非独占和独占用户 CPU 时间。
- 对于同步延迟跟踪，默认集合包含非独占同步等待计数和非独占同步时间。
- 对于硬件计数器溢出分析，默认集合包含非独占及独占时间 （对于在循环中计数的计数器）或事件计数 （对于其他计数器）。
- 对于堆跟踪，默认集合包含堆泄漏和泄漏的字节。

如果已经收集了多种数据类型，将显示每一种类型的默认度量。

可以更改或重新组织显示的度量。有关详细信息，请参见联机帮助。

要搜索函数，请使用查找工具。有关 "查找" 工具的详细信息，参见第 94 页的 "查找文本和数据"。

要选择单个函数，请单击该函数。

要选择标签中连续显示的几个函数，请选择组中的第一个函数，通过按住 Shift 键单击组中的最后一个函数。

要选择标签中未连续显示的几个函数，请选择组中的第一个函数，通过按住 Ctrl 键单击每个函数选择其他函数。

单击工具栏上的 "编写过滤子句" 按钮时，将打开 "高级" 标签处于选中状态的 "过滤" 对话框和加载了反映 "函数" 标签中选择的过滤子句的 "过滤子句" 文本框。

调用方与被调用方标签

"调用方与被调用方" 标签在中央的窗格中显示所选函数，在上面的窗格中显示该函数的调用方，并在下面的窗格中显示该函数的被调用方。

除了显示每个函数的独占和非独占度量值，此标签还显示属性度量。对于所选函数，属性度量表示该函数的独占度量。对于被调用方，属性度量代表被调用方的非独占度量中从中央函数调用的那一部分。被调用方和所选函数的属性度量之和将合计为所选函数的非独占度量。

对于调用方，属性度量代表函数的非独占度量中从调用方调用的那一部分。所有调用方的属性度量之和也将合计为所选函数的非独占度量。

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

再次单击调用方或被调用方窗格中的某个函数将选中该函数，窗口的内容将被刷新，以使所选函数显示在中央窗格中。

源代码标签

如果可用，“源码”标签显示包含所选函数的源代码的文件，以及为每个源代码行注释的性能度量。源代码的列标题中显示了源文件的全名、相应的目标文件和负载对象。在极少情况下，会使用同一源文件来编译多个目标文件，此时，“源码”标签显示包含所选函数的目标文件的性能数据。

分析器在可执行文件中记录的绝对路径名下搜索包含所选函数的文件。如果该文件不在该路径下，分析器则尝试在当前的工作目录下查找相同基名的文件。如果已经移动了源代码，或者在不同的文件系统中记录了实验，则可以放置一个从当前目录指向实际源代码位置的符号链接，以查看带注释的源代码。

当您在“函数”标签中选择函数，而且“源码”标签已打开时，显示的源文件是该函数的默认源上下文。函数的默认源上下文是包含函数第一个指令（对于 C 代码，是函数的左大括号）的文件。紧跟第一个指令之后，带注释的源文件将添加该函数的索引行。源代码窗口以红色斜体文本显示索引行，并用尖括号将文本括起来，格式如下：

<Function:f_name>

函数可能具有替代源上下文，该上下文是包含归属到该函数的指令的另一个文件。这样的指令可能来自包含文件或内联到所选函数中的其他函数。如果存在任何替代源上下文，则默认源上下文的开头包括指示定位替代源上下文位置的扩展索引行的列表。

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

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

为了便于导航，替代源上下文也以一个索引行列表开头，通过这些索引行可以返回到默认源上下文和其他替代源上下文中定义的函数。

源代码与已选定的任何编译器注释穿插在一起显示。可以在“设置数据表示”对话框中设置显示的注释类。在默认文件中可以设值默认类。

可以更改或重新组织“源码”标签中显示的度量。有关详细信息，请参见联机帮助。

源文件中等于或超过任何行的最大度量阈值百分比的度量的行被突出显示，使查找重要的行更容易。可以在“设置数据表示”对话框中设置阈值。在默认文件中可以设置默认阈值。滚动条的旁边显示刻度线，对应于源文件中超过阈值的行的位置。例如，如果靠近源文件的末尾有两个超过阈值的行，则在滚动条靠近源代码窗口底部的旁边将显示两个刻度线。调整刻度线旁边的滚动条可以调整显示在源代码窗口中的刻度线的位置，以显示相应的超过阈值的行。

行标签

“行”标签显示包含源代码行及其度量的列表。源代码行中标有包含这些源代码行的函数、行号以及源文件名。如果没有函数的行号信息，或者不知道函数的源文件，则所有函数的 PC 显示时聚集在一个条目中，即行显示中的函数的条目。来自函数（这些函数来自其函数被隐藏的负载对象）的 PC 显示时聚集在行显示中的负载对象的单个条目中。选中“行”标签中的行会在“摘要”标签中显示该行的所有度量。从“行”标签中选中行后，选择“源码”或“反汇编”标签会将显示调整在适当的行上。

反汇编标签

“反汇编”标签显示包含所选函数的目标文件的反汇编列表，以及为每个指令注释的性能度量。

穿插在反汇编列表中的是源代码（如果有），以及为显示选定的任何编译器注释。用于在“反汇编”标签中查找源文件的算法与“源码”标签中使用的算法相同。

与“源码”标签一样，索引行显示在“反汇编”标签中。但是与“源码”标签中不同的是，无法将替代源上下文的索引行直接用于导航目的。此外，替代源上下文的索引行显示在 `#included` 或内联代码的插入位置的开头，而不仅在“反汇编”视图的开头列出。为 `#included` 或从其他文件内联的代码显示为不与源代码交叉的原始反汇编指令。但是通过将光标放在这些原始反汇编指令的其中一条上，然后选择“源码”标签，将打开包含 `#included` 或内联代码的文件。在显示此文件时选择“反汇编”标签，将在新的上下文中打开“反汇编”视图，从而显示与源代码交叉的反汇编代码。

可以在“设置数据表示”对话框中设置显示的注释类。在默认文件中可以设置默认类。

分析器会突出显示带有等于或超过度量特定阈值的度量的行，使查找重要的行更容易。可以在“设置数据表示”对话框中设置阈值。也可以在默认文件中设置默认阈值。与“源码”标签一样，滚动条的旁边显示刻度线，对应于反汇编代码中超过阈值的行的位置。

PC 标签

PC 标签显示包含 PC 及其度量的列表。PC 所带的标签包含它们来自于其中的函数以及该函数中的偏移。来自函数（这些函数来自其函数被隐藏的负载对象）的 PC 显示时聚集在 PC 显示中的负载对象的单个条目中。选中 PC 标签中的行会在“摘要”标签中显示该 PC 的所有度量。从 PC 标签中选中行后，选择“源码”标签或“反汇编”标签会将显示调整在适当的行上。

DataObjects 标签

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

可以在“设置数据表示”对话框的“标签”标签中选中该标签来显示它（请参见第 93 页的“标签标签”）。仅当一个或多个已加载实验包含数据空间性能分析时，您才可以使 DataObjects 标签可见。

该标签显示针对程序中的各种数据结构和变量的硬件计数器内存操作度量。

要选择单个数据对象，请单击该对象。

要选择在标签中连续显示的几个对象，请选择第一个对象，通过按住 Shift 键单击最后一个对象。

要选择标签中未连续显示的几个对象，请选择第一个对象，通过按住 Ctrl 键单击每个对象来选择其他对象。

单击工具栏上的“编写过滤子句”按钮时，将打开“高级”标签处于选中状态的“过滤”对话框和加载了反映 DataObjects 标签中选择的过滤子句的“过滤子句”文本框。

DataLayout 标签

DataLayout 标签显示所有程序数据对象的注释数据对象布局，以及数据派生的度量数据。该标签中显示的布局按整个结构的数据排序度量值进行排序。此标签显示每个聚集数据对象，以及它所属的总度量，后面按偏移顺序跟随其所有元素。每个元素依次具有自己的度量，以及大小指示器和 32 字节块中的位置。

可以在“设置数据表示”对话框的“标签”标签中选中 DataLayout 标签来显示它（请参见第 93 页的“标签标签”）。与 DataObjects 标签一样，仅当一个或多个已加载实验包含数据空间性能分析时，您才可以使 DataLayout 标签可见。

要选择单个数据对象，请单击该对象。

要选择在标签中连续显示的几个对象，请选择第一个对象，通过按住 Shift 键单击最后一个对象。

要选择标签中未连续显示的几个对象，请选择第一个对象，通过通过按住 Ctrl 键单击每个对象来选择其他对象。

单击工具栏上的“编写过滤子句”按钮时，将打开“高级”标签处于选中状态的“过滤”对话框和加载了反映 DataLayout 标签中选择的过滤子句的“过滤子句”文本框。

MemoryObjects 标签

每个 MemoryObjects 标签都显示归属到各种内存对象（如页面）的数据空间度量的度量值。如果一个或多个已加载实验包含数据空间性能分析，则可以选择要在“设置数据表示”对话框的“标签”标签中为其显示标签的内存对象。可以显示任意数目的 MemoryObjects 标签。

各种 MemoryObject 标签都可以预定义。可以通过单击“设置数据表示”对话框中的“添加定制对象”按钮打开“添加内存对象”对话框来定义自定义内存对象。

通过每个 MemoryObjects 标签上的单选按钮，您可以选择文本显示或图形显示。文本显示类似于 DataObjects 标签中的显示，且使用相同的度量设置。图形显示表示每个内存对象相对值的图形化说明，按数据排序度量排序的每个度量有一个单独的直方图。

单击工具栏上的“编写过滤子句”按钮时，将打开“高级”标签处于选中状态的“过滤”对话框和加载了反映 MemoryObjects 标签中选项的过滤子句的 “过滤子句”文本框。

时间线标签

“时间线”标签显示收集器记录为时间函数的事件和样例点的图表。数据显示在水平条上。对于每个实验，都有一个条用于显示样例数据，以及一组条用于每个 LWP。用于 LWP 的那组条是由用于记录以下各数据类型的条组成的：基于时钟的分析、硬件计数器溢出分析、同步跟踪、堆跟踪和 MPI 跟踪。

包含样例数据的条以颜色编码的表现形式显示每个样例的每个微态中花费的时间。样例显示为时间段，因为样例点中的数据表示在该点和上一个点之间花费的时间。单击样例可以在 “事件”标签中显示该样例的数据。

分析数据或跟踪数据条显示每个记录的事件标记。事件标记包含随事件记录的调用栈的颜色编码表现形式，如彩色矩形堆栈。单击事件标记中的彩色矩形可以选择相应的函数和 PC，并在 “事件”标签中显示该事件以及该函数的数据。“事件”标签和 “图例”标签中的选定内容会突出显示，并且选择 “源码”标签或 “反汇编”标签可以将标签显示定位在相对于调用栈中的该帧的行上。

对于某些类型的数据，事件可能会重叠而看不到。如果在同一个位置有两个或更多事件，则仅绘制一个事件；如果在一个或两个像素内有两个或更多事件，将绘制所有事件，即使可能无法从视觉上区分它们。在任何一种情况下，绘制的事件下将显示一个灰色的小刻度以指示重叠情况。

使用 “设置数据表示”对话框中的 “时间线”标签，可以更改所显示的事件特定数据的类型；选择线程、 LWP 或 CPU 的事件特定数据的显示；选择位于根或叶的调用栈表示的对齐方式；以及选择显示的调用栈的级别数。

您可以更改在 “时间线”标签中显示的事件特定数据的类型，以及映射到所选函数的颜色。有关使用 “时间线”标签的详细信息，参见联机帮助。

泄漏列表标签

“泄漏列表”标签显示两行，上面一行表示泄漏，下面一行表示分配。每一行包含一个调用栈，与 “时间线”标签中显示的调用栈类似，在中央，上方有 1 个条，与泄漏或分配的字节数相对应，下面也有 1 个条，与泄漏或分配数相对应。

选择泄漏或分配后，会在 “泄漏”标签中显示所选泄漏或分配的数据，并在调用栈中选择一个帧，如同在 “时间线”标签中的行为。

可以在“设置数据表示”对话框的“标签”标签中选中 LeakList 标签来显示它（请参见第 93 页的“标签标签”）。仅当一个或多个已加载实验包含堆跟踪数据时，您才可以使 LeakList 标签可见。

统计标签

“统计”标签显示针对所选实验和样例合计的不同系统资料的总计。总计后面跟随每个实验所选样例的统计信息。有关表示的统计内容的信息，请参见 `getrusage`(3C) 和 `proc`(4) 手册页。

实验标签

“实验”标签分为两个面板。上面板包含一个树，其中包括所有已加载实验中负载对象的节点以及每个实验负载的节点。展开“负载对象”节点时，将显示所有负载对象的列表，以及有关其处理情况的各种消息。展开一个实验的节点时，将显示以下两个区域：“说明”区域和“信息”区域。

“说明”区域显示实验中任意说明文件的内容。您可以通过直接在“说明”区域中键入内容编辑说明。“说明”区域包括它自己的工具栏，该工具栏上的按钮可用于保存或丢弃说明以及撤消或重做自上次保存以来进行的任何编辑。

“信息”区域包含有关收集的实验和集合目标访问的负载对象的信息，其中包括在处理实验或负载对象期间生成的任何错误消息或警告消息。

下面板列出了分析器会话中生成的错误和警告消息。

数据显示，右窗格

右窗格包含“摘要”标签、“事件”标签和“图例”标签。默认情况下，显示“摘要”标签。另两个标签灰显，除非选择了“时间线”标签。

摘要标签

“摘要”标签不但显示所选函数或负载对象的所有记录的度量（所有这些度量都以值和百分比的形式显示），还显示所选函数或负载对象的有关信息。只要在任何标签中选择了新函数或负载对象，就会更新“摘要”标签。

事件标签

“事件”标签显示在“时间线”标签中选择的事件的详细数据，其中包括事件类型、叶函数、LWP ID、线程 ID 和 CPU ID。在数据面板下面显示的是调用栈，堆栈中的每个函数带有颜色编码。单击调用栈中的某个函数可选定该函数。

在“时间线”标签中选择了样例后，“事件”标签会显示样例编号、样例的开始和结束时间，以及带有每个微态和颜色编码中花费的时间量的微态。

图例标签

“图例”标签显示颜色到“时间线”标签中的函数和微态的映射的图例。仅当在左窗格中选择了 “时间线”标签后，此标签才可用。您可以更改映射到某一项的颜色，方法是在图例中选中该项并从 “时间线”菜单中选择颜色选择器，或者双击颜色框。

泄漏标签

“泄漏”标签显示“泄漏列表”标签中选择的泄漏或分配的详细数据。在数据面板下方，“泄漏”列表显示检测到所选泄漏或分配时的调用栈。单击调用栈中的某个函数可选定该函数。

设置数据表示选项

您可以从 “设置数据表示”对话框中控制数据的表示形式。要打开此对话框，请单击工具栏中的 “设置数据表示”按钮，或选择 “视图” → “设置数据表示”。

“设置数据表示”对话框具有包含七个标签的标签窗格：

- 度量
- 排序
- 源代码 / 反汇编
- 格式
- 时间线
- 搜索路径
- 标签

该对话框具有一个 “保存”按钮，您可以使用它存储当前设置 （包括任何自定义的内存对象）。

注 – 由于分析器、 er_print 实用程序和 .er_src 实用程序的默认值是由一般的 .er.rc 文件设置的，因此在 “设置数据首选项”对话框中保存更改会影响 .er_print 实用程序和 .er_src 实用程序的输出。

度量标签

“度量”标签显示所有可用度量。每个度量在一个或多个标有时间、值 和 % 的列中具有复选框，具体取决于度量的类型。或者，您可以一次设置所有度量而不是单个度量，方法是选中或取消选中对话框中最下面一行中的复选框，然后单击“应用于所有度量”按钮。

排序标签

“排序”标签显示表示度量的顺序，以及度量的排序依据选项。

源代码 / 反汇编标签

“源代码 / 反汇编”标签表示复选框列表，可用于选择所表示的信息，如下所示：

- 显示在源代码列表和反汇编列表中的编译器注释
- 源代码列表和反汇编列表中的突出显示的重要行的阈值
- 反汇编列表中的交叉源代码
- 反汇编列表中的源代码行上的度量
- 反汇编列表中以十六进制显示的指令

格式标签

“格式”标签显示 C++ 函数名和 Java 方法名的长名形式、短名形式或修整名称形式。如果选中“在函数名后追加 SO 名称”复选框，则将函数或方法所在的共享对象的名称追加到函数名或方法名。

“格式”标签还可以显示用户、专家或机器的视图模式。“视图模式”设置控制 Java 实验和 OpenMP 实验的处理方式。

对于 Java 实验：

- 用户模式显示 Java 线程的 Java 调用栈，但不显示内务处理线程。
- 专家模式显示执行用户的 Java 代码时 Java 线程的 Java 调用栈，及执行 JVM 代码时或 JVM 软件未报告 Java 调用栈时的机器调用栈。它显示内务处理线程的调用栈。
- 机器模式显示所有线程的机器调用栈。

对于 OpenMP 实验：

- 当 OpenMP 运行环境正在执行某些操作时，用户模式和专家模式会显示调和的主线程调用栈和从属线程调用栈，并添加名称形式为 <OMP-*> 的特殊函数。
- 机器模式显示所有线程的机器调用栈。

对于所有其他实验，这三种模式都显示相同的数据。

时间线标签

“时间线”标签包含所显示的事件特定数据的类型的选项、线程的事件特定数据的显示、LWP 或 CPU、位于根或叶的调用栈表示的对齐方式，以及显示的调用栈的级别数。

搜索路径标签

用户可以使用 “搜索路径”标签管理用于搜索源代码和目标文件的目录列表。特殊名称 `$expts` 表示加载的实验；所有其他名称应该是文件系统中的路径。

“设置数据表示”对话框具有用于存储当前设置的 “保存”按钮。

注 – 由于分析器、 `er_print` 实用程序和 `er_src` 实用程序的默认值是由一般的 `.er.rc` 文件设置的，因此在分析器的 “设置数据首选项”对话框中保存更改会影响 `er_print` 实用程序和 `er_src` 实用程序的输出。

标签标签

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

“标签”标签列出适用于当前实验的标签。常规标签位于左列。定义的 MemoryObject 标签位于右列。

单击左列的复选框可选中或取消显示的常规标签。要显示所有适用的标签，请选中 “应用于所有标签”复选框，然后单击 “应用于所有标签”按钮。要取消选择当前显示的所有常规标签，请取消选中 “应用于所有标签”复选框，然后单击 “应用于所有标签”按钮以取消选择当前显示的所有标签。

单击右列的复选框可选择或取消显示的 “内存对象”标签。要添加自定义对象，请单击 “添加定制对象”复选框打开 “添加内存对象”对话框。在 “对象名”文本框中，键入新的自定义内存对象的名称。在 “公式”文本框中，键入用于将记录的物理地址或虚拟地址映射到对象索引的索引表达式。有关索引表达式规则的信息，请参见第 118 页的 ″mobj_define mobj_type index_exp″。

使用 “添加内存对象”对话框添加自定义内存对象后，该对象的复选框就会添加到 “标签”标签的右列中，并且在默认情况下处于选中状态。

查找文本和数据

分析器具有一个可通过工具栏访问的 “查找”工具，其中带有用于搜索组合框中给定目标的两个选项。您可以在 “函数”标签或 “调用方与被调用方”标签的 “名称”列，以及 “源码”标签和 “反汇编”标签的代码列中搜索文本。可以在 “源码”标签和 “反汇编”标签中搜索高度量项。包含该度量项的行上的度量值以绿色突出显示。使用 “查找”字段旁边的箭头键可以向下或向上搜索。

显示或隐藏函数

默认情况下，每个负载对象中的所有函数显示在 “函数”标签和 “调用方与被调用方”标签中。您可以使用 “显示 / 隐藏函数”对话框隐藏负载对象中的所有函数。有关详细信息，请参见联机帮助。

负载对象中的函数被隐藏时，“函数”标签和 “调用方与被调用方”标签显示单个条目，表示负载对象中的所有函数的聚集。同样，“行”标签和 “PC” 标签显示单个条目，表示对象中的所有函数中的所有 PC 的聚集。

与过滤相对比，与隐藏的函数对应的度量仍在所有显示中以某种形式表示。

过滤数据

默认情况下，每个标签中显示所有实验、所有样例、所有线程、所有 LWP 和所有 CPU 的数据。可以使用 “过滤数据”对话框选择数据的子集。

“过滤数据”对话框包含 “简单”标签和 “高级”标签。在 “简单”标签中，您可以选择要为其过滤数据的实验。然后，可以指定要为其显示度量的样例、线程、LWP 和 CPU。在 “高级”标签中，您可以指定一个过滤表达式，该表达式使您要在显示中包括的任何数据记录的求值为真。有关在过滤表达式中使用的语法信息，请参见第 129 页的 “表达式语法”。

在分析器的 “函数”标签、DataLayout 标签、DataObjects 标签或 MemoryObject 标签中进行选择后，单击工具栏上的 “编写过滤子句”按钮将打开 “过滤数据”对话框的 “高级”标签，并用反映选择的子句加载 “过滤子句”文本框。

有关使用 “过滤数据”对话框的详细信息，参见联机帮助。

实验选择

如果加载了多个实验，分析器还允许过滤实验。可以单独加载实验，也可以通过指定实验组来加载实验。

样例选择

样例的编号方式是从 1 到 *N*，并且可以选择样例的任意集合。选择包括样例编号或范围（例如 15）的逗号分隔列表。

线程选择

线程的编号方式是从 1 到 *N*，并且可以选择线程的任意集合。选择包括线程编号或范围的逗号分隔列表。线程的分析数据仅包括运行的某一部分，在该部分中，针对 LWP 实际调度线程。

LWP 选择

LWP 的编号方式是从 1 到 *N*，并且可以选择 LWP 的任意集合。选择包括 LWP 编号或范围的逗号分隔列表。如果记录了同步数据，则报告的 LWP 是位于同步事件的进入点的 LWP，这可能与从同步事件退出时的 LWP 不同。

在 Linux 系统中，线程和 LWP 是同义词。

CPU 选择

在记录 CPU 信息处（Solaris 9 操作系统），可以选择 CPU 的任意集合。选择包括 CPU 编号或范围的逗号分隔列表。

记录实验

使用目标名称和目标参数调用分析器时，分析器将启动，并打开“性能工具收集”窗口，这样您就可以在指定目标上记录实验。如果不使用参数或使用实验列表调用分析器，则可以记录新实验，方法是选择 “文件” → “收集实验” 打开 “性能工具收集” 窗口。

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

紧靠面板下方是“预览命令”按钮以及一个文本字段。单击该按钮时，将用 `collect` 命令（单击“运行”按钮时可使用它）填充文本字段。

在“要收集的数据”标签中，您可以选择要收集的数据类型。

“输入 / 输出”标签有两个面板：一个用于从收集器本身接收输出，另一个用于从进程接收输出。

一组按钮，允许以下操作：

- 运行实验
- 终止运行
- 在运行过程中将暂停、恢复和样例信号发送给进程（在指定了相应的信号时启用）
- 关闭窗口。

如果在关闭窗口时仍有实验正在进行中，该实验将继续。如果重新打开窗口，它会显示正在进行中的实验，就像该实验在运行过程中保持打开那样。如果在某个实验正在进行的时候试图退出分析器，将打开一个对话框，询问您想要终止运行还是允许继续运行。

生成映射文件和函数顺序重排

除了分析数据外，分析器还提供函数顺序重排功能。基于实验中的数据，分析器可以生成映射文件。与静态链接程序 (ld) 一起使用以重新链接应用程序时，映射文件会创建一个可执行文件，该文件具有较小的工作集合大小或更佳的 I 高速缓存行为或两者。

映射文件中记录的用于重组可执行文件中的函数的函数顺序是由用于排序函数列表的度量确定的。独占用户 CPU 时间或独占 CPU 循环时间通常用于生成映射文件。某些度量（例如来自同步延迟或堆跟踪中的度量）或者名称或地址不会生成映射文件的有意义的排序。

默认

如果当前目录中有 `.er.rc` 文件，分析器将处理该文件中的指令；如果用户的 Home 目录中有 `.er.rc` 文件，分析器将处理该文件中的指令；分析器还处理系统范围的 `.er.rc` 文件中的指令。这些文件可以包含将实验加载到分析器时其标签可见的默认设置。标签由相应报告的 `er_print` 命令命名，但“实验”标签和“时间线”标签除外。

.er.rc 文件可以包含用于度量、排序、指定编译器注释选项，以及突出显示源代码和反汇编输出的阈值的默认设置。可以指定 “时间线” 标签和名称格式的默认设置，以及视图模式的设置。可以包含用于控制源代码文件和目标文件的搜索路径的指令。

.er.rc 文件还可以包含打开或关闭 en_desc 模式的设置，以控制在读取创建实验时是否选择后续实验并进行读取。

在分析器 GUI 中，单击 “设置数据表示” 对话框 （可以从 “视图” 菜单打开它）中的 “保存” 按钮，可以保存 .er.rc 文件。从 “设置数据表示” 对话框保存 .er.rc 文件不但会影响对分析器的后续调用，而且还会影响 er_print 实用程序和 er_src 实用程序。

分析器将消息置于其 “错误 / 警告日志” 区域中，并命名它处理的用户 .er.rc 文件。

第5章

内核分析

本章描述如何在 Solaris 操作系统运行负载的同时使用 Sun Studio 性能工具分析内核。如果在 Solaris 10 操作系统上运行 Sun Studio 软件，则内核分析是可用的。

内核实验

可以使用 er_kernel 实用程序记录内核性能分析数据。

er_kernel 实用程序使用 DTrace 驱动程序、 Solaris 10 操作系统中内置的综合动态跟踪工具。

er_kernel 实用程序捕获内核性能分析数据，并以与用户性能分析数据相同的格式将数据记录为分析器实验。实验可以由 er_print 实用程序或性能分析器进行处理。内核实验可以显示功能数据、调用方与被调用方数据、指令级数据和时间线，但是不能显示源代码行数据 （因为大多数 Solaris 操作系统模块不包含行号表）。

为内核分析设置系统

您需要先设置对 DTrace 驱动程序的访问，才能使用 er_kernel 实用程序进行内核分析。

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

```
username::::defaultpriv=basic,dtrace_kernel,dtrace_proc
```

要将您自己添加到组 sys，请将您的用户名添加到文件 /etc/group 中的 sys 行。

运行 er_kernel 实用程序

您可以运行 er_kernel 实用程序，以便仅分析内核或同时分析内核和正在运行的负载。有关 er_kernel 命令的完整描述，参见 er_kernel(1) 手册页。

分析内核

1. **通过键入以下内容来收集实验：**

```
% er_kernel -p on
```

2. **在单独的 shell 中运行所需的任意负载。**
3. **负载完成后，按 ctrl-C 来终止 er_kernel 实用程序。**
4. **将生成的实验（默认情况下名为 ktest.1.er）加载到性能分析器或 er_print 实用程序中。**

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

   您可以将 er_kernel 实用程序的 -p on 参数替换为 -p high（用于高分辨率分析）或 -p low（用于低分辨率分析）。如果期望用 2 到 20 分钟来运行负载，则默认时钟分析是合适的。如果期望用不到 2 分钟的时间来运行，请使用 -p high；如果期望用 20 分钟以上的时间来运行，请使用 -p low。

   您可以添加 -t *n* 参数，该参数将导致 er_kernel 实用程序在 *n* 秒后自行终止。

   如果希望在屏幕上输出有关运行的详细信息，则可以添加 -v 参数。通过 -n 参数，您可以预览将被记录的实验，而无须实际记录任何内容。

   默认情况下，er_kernel 实用程序生成的实验被命名为 ktest.1.er；对于相继的运行，该数字将递增

在有负载时分析

如果有要用作负载的单个命令（程序或脚本）：

1. **通过键入以下内容来收集实验：**

```
% er_kernel -p on load
```

2. **通过键入以下内容来分析实验：**

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

er_kernel 实用程序派生一个子进程，并暂停一个静止期，然后子进程会运行指定的负载。在负载终止时，er_kernel 实用程序再次暂停一个静止期，然后退出。实验显示运行负载期间以及之前和之后的静止期内 Solaris 操作系统的行为。您可以使用 er_kernel 命令的 -q 参数，以秒为单位指定静止期的持续时间。

一起分析内核和负载

如果有要用作负载的单个程序，并希望同时看到该负载的性能分析数据和内核性能分析数据：

1. **通过键入 er_kernel 命令和 collect 命令，同时收集内核性能分析数据和用户性能分析数据：**

```
% er_kernel collect load
```

2. **通过键入以下内容一起分析这两个性能分析数据：**

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

分析器显示的数据同时显示 ktest.1.er 中的内核性能分析数据和 test.1.er 中的用户性能分析数据。时间线允许您查看两个实验之间的关联。

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

分析特定的进程或内核线程

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

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

在为目标线程调用 er_kernel 实用程序之前，必须已经创建了它们。

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

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

分析内核性能分析数据

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

表 5-1 分析器中内核实验的字段标签含义

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

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

第6章

er_print 命令行性能分析工具

本章说明了如何使用 er_print 实用程序进行性能分析。er_print 实用程序输出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，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 实用程序。可以使用 er_print 实用程序来显示函数、调用方与被调用方的性能度量、源代码列表和反汇编列表、抽样信息、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
- 控制源代码和反汇编列表的命令
- 控制数据空间列表的命令
- 控制内存对象列表的命令
- 列出实验、样例、线程和 LWP 的命令
- 控制实验数据过滤的命令
- 控制负载对象展开和折叠的命令
- 列出度量的命令
- 控制输出的命令
- 输出其他信息的命令
- 设置默认值的命令
- 设置仅用于性能分析器默认的命令
- 杂项命令
- 示例

关于收集器收集的数据描述，请参见第 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，请参见第 4 章和联机帮助。

er_print 语法

er_print 实用程序的命令行语法如下所示：

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

er_print 实用程序的选项如下：

-	读取从键盘输入的 er_print 命令。
-script *script*	从包含 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。按出现的顺序处理这些选项。可以用任何顺序的混合脚本、连字符和显式命令。如果不支持任何命令或脚本，则默认进入交互模式，交互模式中的命令从键盘输入。要退出交互模式，请键入 quit 或 Ctrl-D。

处理完每个命令后，会输出因处理而引起的任何错误消息或警告消息。您可以使用 procstats 命令输出有关处理的摘要统计信息。

er_print 实用程序接受的命令会在以下章节中列出。

只要意思明确就可以用更短的字符串来缩写任何命令。通过以 \ 结束行的方式可以将命令拆分为多行。以 \ 结尾的任何行在被分析之前都会将 \ 字符删除，并追加下一行的内容。除可用内存外，对可用于命令的行数没有限制。

必须将包含嵌入空格的参数用双引号括起来。可以将引号内的文本拆分为多行。

度量列表

许多 er_print 命令使用度量关键字的列表。列表的语法为：

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

对于动态度量（它们基于已度量的数据），度量关键字包括三部分：度量类型字符串、度量可视性字符串和度量名称字符串。这三部分连在一起中间没有空格，如下所示：

```
flavorvisibilityname
```

对于静态度量 — 它们基于实验中负载对象的静态属性 （名称、地址和大小），度量关键字包括度量名称以及前置的度量可视性字符串 （可选），它们连在一起中间没有空格：

```
[visibility]name
```

度量 *flavor* 和度量 *visibility* 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 6-1 中给出。包含多个类型字符的度量关键字扩展成一系列度量关键字。例如， ie.user 扩展为 i.user:e.user。

表 6-1 度量类型字符

字符	描述
e	显示独占度量值
i	显示非独占度量值
a	显示属性度量值 （仅适用于调用方与被调用方度量）
d	显示动态度量值 （仅适用于数据派生的度量）

允许的度量可视性字符在表 6-2 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不同的度量的关键字被某些其他关键字分开，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 6-2 度量可视性字符

字符	描述
.	将度量显示为时间。应用于测量循环计数的定时度量和硬件计数器度量。其他度量解释为"+"。
%	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非独占度量百分比。
+	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。不能用于与其他可视性字符的组合。

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。

对于静态度量，可视性字符句点 (.)、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 均表示分析器只要以任一形式可见都应该按非独占用户 CPU 时间排序，而 sort i!user 则表示不管分析器是否可见都应该按非独占用户 CPU 时间排序。

您可以使用可视性字符感叹号 (!) 覆盖每种度量类型的内置可视性默认值。

如果同一度量在度量列表中出现多次，则仅对第一次出现的度量进行处理，而忽略后续出现的度量。如果指定的度量不在列表上，则将它追加到列表。

表 6-3 列出了可用于定时度量、同步延迟度量、内存分配度量、MPI 跟踪度量和两个一般硬件计数器度量的 er_print 度量名称字符串。对于其他硬件计数器度量，其度量名称字符串与计数器名称相同。您可以使用 metric_list 命令获得已加载实验的所有可用度量名称字符串的列表。还可以通过使用不带参数的 collect 命令获得计数器名称的列表。关于硬件计数器的详细信息，请参见第 34 页的 “硬件计数器溢出分析数据”。

表 6-3 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
	system	系统 CPU 时间
	wait	CPU 等待时间
	ulock	用户锁定时间
	text	文本缺页时间
	数据	数据缺页时间
	owait	其他等待时间
同步延迟度量	sync	同步等待时间
	syncn	同步等待计数
MPI 跟踪度量	mpitime	MPI 调用所用的时间
	mpisend	MPI 发送操作的数量
	mpibytessent	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	对其他 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 6-3 中列出的名称字符串之外，还有两个仅能用于默认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。另请注意，cycles 和 insts 通常用于 SPARC® 平台和 x86 平台，而其他特定的体系结构则喜欢用其他的名称字符串。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 （请参见第 124 页的 “控制输出的命令”）。

输出的默认度量是独占和非独占用户 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。您可以使用 metrics 命令更改当前显示的度量，该命令必须在发出 functions 命令之前发出。此外，也可以用 .er.rc 文件中的 dmetrics 命令更改默认设置。

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视图模式的设置，设置选项有 “用户”、“专家” 或 “机器”。

- 用户模式按名称显示每种方法，将解释的方法与 HotSpot 编译的方法的数据聚集在一起；它禁止非用户 Java 线程的数据。
- 专家模式将 HotSpot 编译的方法与解释的方法分开，并且不禁止非用户 Java 线程。
- 机器模式在进行解释时对照 Java 虚拟机 (JVM) 软件显示解释的 Java 方法的数据，而对指定的方法报告使用 Java HotSpot 虚拟机编译的方法数据。显示所有线程。

在所有这三种模式中，对于由 Java 目标调用的任何 C、C++ 或 Fortran 代码，都以通用的方法报告数据。

metrics *metric_spec*

指定函数列表度量的选项。字符串 *metric_spec* 可以是恢复默认度量选项的关键字 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 实用程序显示以下度量：

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

默认情况下，使用基于 dmetrics 命令且在 .er.rc 文件中处理的度量设置，如第 126 页的“设置默认值的命令”所述。如果 metrics 命令将 *metric_spec* 显式设置为 default，则恢复适合于所记录数据的默认设置。

重置度量时，默认的排序度量会在新列表中设置。

如果省略 *metric_spec*，则输出当前的度量设置。

除了设置函数列表的度量，metrics 命令还将调用方与被调用方的度量与数据派生输出的度量设为相同设置。有关更多的详细信息，请参见第 111 页的 ”cmetrics metric_spec” 和第 117 页的 ”data_metrics metric_spec”。

metrics 命令处理完成后，输出显示当前度量选项的信息。上述示例输出的信息如下所示：

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

关于度量列表语法的信息，请参见第 105 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，但先前的设置仍然有效。

sort *metric_spec*

用 *metric_spec* 对函数列表进行排序。度量名称中的 *visibility* 不会影响排序顺序。如果在 *metric_spec* 中指定了多个度量，请使用第一个可见度量。如果指定的度量均不可见，请忽略该命令。您可以在 *metric_spec* 前面加上负号 (-) 以指定反向排序。

默认情况下，使用基于 dsort 命令且在 .er.rc 文件中处理的度量排序设置，如第 126 页的“设置默认值的命令”所述。如果 sort 命令将 *metric_spec* 显式设置为 default，则使用默认设置。

字符串 *metric_spec* 是第 105 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
```

此命令说明 er_print 实用程序按非独占用户 CPU 时间对函数列表进行排序。如果度量不在装入的实验中，则输出警告并忽略该命令。命令完成后，输出排序度量。

fsummary

为函数列表中的每个函数写入汇总面板。可以使用 limit 命令限制写入面板的数目（请参见第 124 页的 “控制输出的命令”）。

汇总度量面板包括函数或负载对象的名称、地址和大小，（函数的）源文件、目标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（独占和非独占）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，输出带有对应 *N* 值的函数列表。

关于函数的汇总度量的描述，请参见对 fsummary 命令的描述。

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。

callers-callees

按函数排序度量指定的顺序，输出每个函数的调用方与被调用方面板 (sort)。

在每个调用方与被调用方报告内，调用方与被调用方按调用方与被调用方排序度量进行排序 (csort)。可以使用 limit 命令限制写入面板的数目（请参见第 124 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      独占      非独占      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。默认情况下，只要更改了函数列表度量，就将调用方与被调用方度量设置为与函数列表度量匹配。如果省略 *metric_spec*，则输出当前的调用方与被调用方度量设置。

字符串 *metric_spec* 是第 105 页的"度量列表"中所述的度量关键字之一，如本示例所示。

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

该命令指示 er_print 显示以下度量。

- 用秒表示的非独占用户 CPU 时间
- 非独占用户 CPU 时间百分比
- 用秒表示的属性用户 CPU 时间
- 属性用户 CPU 时间百分比

cmetrics 命令完成后，输出显示当前度量选项的信息。上述示例输出的信息如下所示：

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

默认情况下，只要更改了函数列表度量，就将调用方与被调用方度量设置为与函数列表度量匹配。

调用方与被调用方属性度量被插入到对应的独占度量和非独占度量前面， *visibility* 对应于这两个度量的 *visibility* 设置的逻辑 "或"。静态度量设置被复制到调用方与被调用方度量。如果 *metric-name* 不在列表中，请将其追加到列表。

可以使用 cmetric_list 命令获得已加载实验的所有可用 *metric-name* 值的列表。

如果 cmetrics 命令中包含错误，则忽略并显示警告，但先前的设置仍然有效。

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，输出带有对应 *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 105 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
```

如果省略 *metric-spec*，则输出当前的调用方与被调用方排序度量。

csort 度量必须是属性度量或静态度量。如果指定多个度量，则按匹配的第一个可见度量进行排序。

只要设置了度量（显式设置或默认设置），就会基于以下函数度量设置调用方与被调用方排序度量：

- 如果依据动态度量排序 （独占或非独占），则按对应的属性度量进行排序。
- 如果依据静态度量排序，则按它进行排序。

该命令告知 er_print 实用程序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，输出排序度量。

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，由一般调用栈聚集。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，由一般调用栈聚集。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。

控制源代码和反汇编列表的命令

以下命令控制如何显示注释的源代码和反汇编代码。

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聚集度量的行，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聚集度量的行，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聚集度量的行，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` 实用程序输出可能的目标文件名称的列表；如果给出的名称是函数，则函数的名称追加到目标文件的名称，而代表该目标文件 *N* 的值的数字也会输出。

函数名称还可以指定为 *function`file`*，其中 *file* 用于指定函数的替代源上下文。紧邻第一个指令之后，添加函数的索引行。索引行显示为尖括号内的文本，格式如下：

```
<Function:f_name>
```

任何函数的默认源上下文都被定义为该函数的第一条指令所属的源文件。该文件通常是经过编译而生成包含函数的对象模块的源文件。替换源上下文由包含该函数指令的其他文件组成。这样的上下文包括源自包含文件的指令及源自内联为已命名函数的函数的指令。如果存在任何替代源上下文，请在默认源上下文的开头包括扩展索引行的列表，按以下格式指示替代源上下文所在的位置：

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

注 – 如果在命令行上调用 `er_print` 实用程序时使用 `-source` 参数，则必须在文件引号的前面放置反斜杠转义符。换句话说，函数名称的格式应为 `function\'file\'`。当 `er_print` 实用程序处于交互模式时不需要而且也不应该使用反斜杠。

通常，在使用默认源上下文时，显示该文件中所有函数的度量。如果显式引用该文件，则仅为指定的函数显示度量。

`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代码。文件必须位于路径的目录中。

可选参数 *N* 与 `source` 命令的可选参数的使用方法相同。

`scc` *com_spec*

指定显示在注释源代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 6-4 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。

表 6-4 编译器注释消息类 *（续）*

类	含义
f[e]	显示前端消息。
all	显示所有消息。
none	不显示任何消息。

类 all 和 none 不能与其他类一起使用。

如果没有给出 scc 命令，则显示的默认类为 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代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。您可以将以下选项添加到类列表。

表 6-5 dcc 命令的附加选项

选项	含义
h[ex]	显示指令的十六进制值。
noh[ex]	不显示指令的十六进制值。
s[rc]	在注释反汇编代码列表中交叉列出源代码。
nos[rc]	不在注释反汇编代码列表中交叉列出源代码。
as[rc]	在注释反汇编代码列表中交叉列出注释源代码。

dthresh *value*

指定注释反汇编代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*

设置用于查找源代码、对象等文件的路径。*path_list* 是以冒号分隔的目录列表。如果任何目录包含冒号字符，请用反斜杠转义该目录。特殊目录名 `$expts`，按装入实验的顺序引用当前实验集；可以将它缩写为单个 `$` 字符。

默认设置为：`$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` 输出当前路径。

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

数据空间命令仅适用于指定了积极回溯的硬件计数器实验以及使用 `-xhwcprof` 选项（仅可用于 C 和 C++ 的 SPARC® 平台上）编译的文件中的对象。有关更多的详细信息，请参见 《C 用户指南》或 《C++ 用户指南》。

data_objects

写入数据对象及其度量的列表。仅适用于指定了积极回溯的硬件计数器溢出实验和用 `-xhwcprof` 编译的文件中的对象。（仅可用于基于 SPARC 的系统上的 C）。有关更多的详细信息，请参见 《C 用户指南》或 cc(1) 手册页。

data_single *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。仅适用于指定了积极回溯的硬件计数器溢出实验和用 `-xhwcprof` 编译的文件中的对象。（仅可用于基于 SPARC 的系统上的 C）。有关更多的详细信息，请参见 《C 用户指南》或 cc(1) 手册页。

data_layout

为具有数据派生度量数据的所有程序数据对象写入带注释的数据对象布局，按整个结构的当前数据排序度量值排序。每个聚集数据对象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。

data_metrics *metric_spec*

设置数据派生的度量。 *metric_spec* 已在第 105 页的 “度量列表”中定义。

默认情况下，只要更改了函数列表度量，就会将数据派生度量设置为与函数列表度量匹配。设置对应于具有数据派生类型的任何可见独占或非独占度量的数据派生度量，*visibility* 对应于那两个度量的 *visibility* 设置的逻辑 “或”。

将静态度量设置复制到数据派生度量。如果度量名称不在列表中，请将其追加到列表。

如果省略 *metric_spec*，则输出当前的数据派生度量设置。

可以使用 `data_metric_list` 命令获得已加载实验的所有可用 *metric-name* 值的列表。

如果 *metric_spec* 有任何错误，则忽略它，并将数据派生度量保持不变。

data_sort

设置数据对象的排序度量。动态度量需要有前缀 `d`，但是静态度量可以省略它。`data_sort` 度量必须是数据派生度量或静态度量。

如果指定多个度量，则按匹配的第一个可见度量进行排序。只要设置了度量 （显式设置或默认设置），就会基于以下函数度量设置数据派生的排序度量：

- 如果排序依据为具有对应的数据派生类型的动态度量 （独占或非独占），则按对应的数据派生度量进行排序。
- 如果排序依据为没有数据派生类型的非独占或独占度量，则按第一个可见的数据派生度量进行排序。
- 如果排序依据为静态度量，则按它进行排序。

控制内存对象列表的命令

内存对象命令仅适用于指定了积极回溯的硬件计数器实验以及使用 -xhwcprof 选项 （仅可用于 C 和 C++ 的 SPARC® 平台上）编译的文件中的对象。有关更多的详细信息，请参见 《C 用户指南》或 《C++ 用户指南》。

内存对象是内存子系统中的组件，如缓存行、页面和内存库。对象是通过从记录的虚拟或物理地址计算的索引确定的。预定义的虚拟页面和物理页面的内存对象的大小可为 8KB、64KB、 512KB 和 4 MB。您可以使用 `mobj_define` 命令定义其他对象。

以下命令控制内存对象列表。

memobj *mobj_type*

用当前度量写入给定类型内存对象的列表。可以用于数据空间列表的度量及排序。也可以直接将名称 *mobj_type* 作为命令。

mobj_list

写入已知内存对象类型的列表，例如用于 memobj 命令中的 *mobj_type* 。

mobj_define *mobj_type index_exp*

通过将 VA/PA 映射到 *index_exp* 给出的对象，定义新的内存对象类型。表达式的语法将在第 129 页的 “表达式语法” 中介绍。

mobj_type 不必已经定义。但其名称必须完全由字母数字字符或 '_' 组成，并以字母字符开头。

index_exp 必须在语法上是正确的。如果它在语法上不正确，则返回一个错误，并忽略定义。

<Unknown> 内存对象的索引为 -1，用于定义新内存对象的表达式应该支持识别 <Unknown>。例如，基于 VADDR 的对象，表达式应该采用以下格式：

VADDR>255?*expression*:-1

基于 PADDR 的对象，表达式应该采用以下格式：

PADDR>0?*expression*:-1

列出实验、样例、线程和 LWP 的命令

本节描述了列出实验、样例、线程和 LWP 的命令。

experiment_list

显示装入实验及其 ID 号的完整列表。每个实验都与索引一起列出，在选择样例、线程或 LWP 以及 PID （可以用于高级过滤）时可使用索引。

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

显示当前选定的分析样例列表。

以下示例是样例列表的示例。

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

lwp_list

显示当前选定的分析 LWP 列表。

thread_list

显示当前选定的分析线程列表。

cpu_list

显示当前选定的分析 CPU 列表。

控制实验数据过滤的命令

您可以使用以下两种方法指定实验数据的过滤：

- 指定过滤表达式，为每个数据记录计算该表达式以确定是否应该包括该记录
- 选择实验、样例、线程、 CPU 和 LWP 以进行过滤

指定过滤表达式

您可以使用 `filters` 命令指定过滤表达式。

`filters` *filter_exp*

filter_exp 是一个表达式，对于应该包括的任何数据记录，其计算结果为真；对于不应该包括的记录，其计算结果为假。该表达式的语法将在第 129 页的“表达式语法”中描述。

选择样例、线程、 LWP 和 CPU 以进行过滤

选择列表

选择的语法如以下示例所示。该语法用于命令描述。

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

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生成多个选择。

实验列表和选择列表具有相同的语法，可以是关键字 `all`，也可以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，对象 5 和 6 从实验 2 选择。在第二个示例中，对象 1、 3 到 6 从所有示例中选择。对象可以是 LWP、线程或样例。

选择命令

选择 LWP、样例、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、样例和线程。

- 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。
- 对于没有做出任何选择的目标，将选择设置为 `all`。

`sample_select` *sample_spec*

选择要显示信息的样例。命令完成后显示选定的样例列表。

`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。

控制负载对象展开和折叠的命令

object_list

显示一个两列列表，其中包含可用负载对象的状态和名称。每个负载对象的展开状态显示在第一列，对象的名称显示在第二列。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（展开）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（折叠）中。已折叠负载对象的所有函数都映射到函数列表中，表示整个负载对象的单个条目。

以下是负载对象列表的示例。

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

object_select *object1,object2,...*

选择要显示的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。如果选定某个负载对象，则会展开其函数，并且在函数列表中显示具有非零度量的所有函数。如果未选定某个负载对象，则会折叠其函数，并且仅显示包含整个负载对象的度量的单个行，而不是其各个函数。

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，则该名称必须加双引号。

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他命令（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他命令（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 – 可以将属性度量指定为只有在使用 cmetrics 命令（而不是 metrics 命令或 data_metrics 命令）时才显示，并且可以将其指定为只有在使用 callers-callees 命令（而不是 functions 命令或 data_objects 命令）时才显示。

---

data_metric_list

显示当前选择的数据派生度量以及所有数据派生报告的度量和关键字名称的列表。显示列表的方式与 metric_list 命令的输出相同，但是仅包括具有数据派生类型的那些度量和静态度量。

注 – 可以将数据派生的度量指定为只有在使用 data_metrics 命令（而不是 metrics 命令或 cmetrics 命令）时才显示，并且可以将其指定为只有在使用 data_objects 命令（而不是 functions 命令或 callers-callees 命令）时才显示。

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*。打开 *filename* 时，将清除预先存在的任何内容。如果指定破折号 (-) 而不是 *filename*，则输出写入标准输出。如果指定两个破折号 (--) 而不是 *filename*，则输出写入标准错误。

appendfile *filename*

关闭任何打开的输出文件并打开 *filename*，保留任何预先存在的内容，以便将后续输出追加到文件的结尾。如果 *filename* 不存在，则 appendfile 命令的功能与 outfile 命令的功能相同。

limit *n*

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 } [:{*shared_object_name* | *no_shared_object_name* }]"

指定要使用函数名称的长名形式还是短名形式（仅 C++ 和 Java）。如果指定了 *shared_object_name* ，则将共享对象名称追加到函数名称。

viewmode { user | expert | machine }

将模式设置为以下模式之一：

user	对于 Java 实验，显示 Java 线程的 Java 调用栈，但不显示内务处理线程。函数列表包括函数 <JVM-System>，该函数表示来自非 Java 线程的聚集时间。当 JVM 软件不报告 Java 调用栈时，将根据函数 <no Java callstack recorded> 报告时间。 对于 OpenMP 实验，在 OpenMP 运行时执行某些操作时，将显示调和的主线程调用栈和从属线程调用栈，并添加名称格式为 <OMP-*> 的特殊函数。
expert	对于 Java 实验，在执行用户的 Java 代码时显示 Java 线程的 Java 调用栈，在执行 JVM 代码或 JVM 软件未报告 Java 调用栈时显示机器调用栈。显示内务处理线程的机器调用栈。 对于 OpenMP 实验，在 OpenMP 运行环境执行某些操作时，将显示调和的主线程调用栈和从属线程调用栈，并添加名称格式为 <OMP-*> 的特殊函数。
machine	对于 Java 实验和 OpenMP 实验，显示所有线程的机器调用栈。

对于除 Java 实验和 OpenMP 实验外的所有实验，这三种模式将显示相同的数据。

输出其他信息的命令

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，还输出任何错误或警告。每个实验的标题用破折线分隔。

如果实验目录包含名为 notes 的文件，则该文件的内容将置于标题信息的前面。可以编辑或用 -C“*comment* ” 参数来指定 notes 文件，或手动将其添加到 collect 命令中。

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（请参见第 124 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例的样例数据。*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例数据。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，聚集指定实验的当前样例集。关于出现的执行统计的定义和含义，请参见 getrusage(3C) 和 proc(4) 手册页。执行统计包括源于系统线程的统计，收集器不为该线程收集任何数据。Solaris 7 操作系统和 Solaris 8 操作系统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处于休眠状态，休眠时间在统计显示中显示为 “其他等待” 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，聚集每个实验的样例集。如果 *exp_id* 是 all，则显示每个实验的总和以及独立统计。

设置默认值的命令

可以使用以下命令设置 er_print 和性能分析器的默认值。只能使用这些命令设置默认值：在 er_print 实用程序的输入中不能使用它们。它们可以包括在名为 .er.rc 的默认文件中。仅适用于性能分析器默认值的命令在第 127 页的 “设置仅用于性能分析器默认的命令” 中有描述。

可以将默认文件包括在 Home 目录中以设置所有实验的默认值，或将默认文件包括在任何其他目录中，并在相应的目录中设置默认值。er_print 实用程序、er_src 实用程序或性能分析器启动时，会在当前目录和 Home 目录扫描默认文件，如果存在默认文件，则读取该默认文件，同时还读取系统默认文件。 Home 目录中 .er.rc 文件的默认值将覆盖系统的默认值，而当前目录中 .er.rc 文件的默认值将覆盖 Home 目录和系统的默认值。

注 – 要确保从存储实验的目录读取默认文件，必须在该目录中启动性能分析器或 er_print 实用程序。

默认文件可以包括 scc、sthresh、dcc、dthresh、setpath、addpath、name、mobj_define、 tabs 和 viewmode 命令。还可以包括多个 dmetrics、 dsort 和 addpath 命令，而且将并置所有 .er.rc 文件的命令。对于所有其他命令，将使用该命令的第一个实例，而忽略其后续实例。

dmetrics *metric_spec*

指定在函数列表中显示或输出的默认度量。度量列表的语法和使用将在第 105 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，可以从函数列表默认度量派生 “调用方与被调用方” 列表的默认度量。

dsort *metric_spec*

按函数列表排序的方式指定默认度量。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他可见性字符串，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将在第 105 页的 “度量列表” 中描述。

“调用方与被调用方” 列表的默认排序度量是与函数列表默认排序度量对应的属性度量。

en_desc { on | off }

将读取后续实验的模式设置为 *on* （启用所有后续实验）或 off （禁用所有后续实验）。

设置仅用于性能分析器默认的命令

tabs *tab_spec*

设置在分析器中可见的默认标签集合。标签由生成对应报告的 er_print 命令命名，其中包括内存对象标签的 *mobj_type*。 *timeline* 指定 “时间线” 标签，而 *headers* 指定 “实验” 标签。

仅显示已加载实验中的数据支持的那些标签。

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 6-6 时间线显示模式选项

选项	含义
lw[p]	显示 LWP 的事件
t[hread]	显示线程的事件
c[pu]	显示 CPU 的事件
r[oot]	在根对齐调用栈
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 一样，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”标签中的默认数据类型。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 6-7 时间线显示数据类型

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

杂项命令

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

将指定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。

procstats

输出处理数据的累积统计信息。

script *file*

处理脚本文件 *file* 的附加命令

version

输出 er_print 实用程序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

输出 er_print 命令的列表。

表达式语法

定义过滤器的表达式和用于计算内存对象索引的表达式使用相同的语法。

语法将表达式指定为运算符和操作数的组合。对于过滤器，如果表达式的求值为真，则包括数据包；如果表达式的求值为假，则不包括数据包。对于内存对象，表达式的值为索引，该索引定义了数据包中引用的特定内存对象。

表达式中的操作数可以是常量、数据记录中的字段（包括 THRID、 LWPID、 CPUID、STACK、LEAF、VADDR、PADDR、DOBJ、TSTAMP、SAMPLE、EXPID、PID）或内存对象的名称。运算符包括常见的逻辑运算符和算术运算符（包括移位运算符），在 C 表示法中，使用 C 优先规则，并且使用运算符来确定某一元素在集合 (IN) 中，还是任何或者所有集合元素均包含在集合 （分别为 SOME IN 或 IN）中。如果在 C 中指定了 If-then-else 构造，则使用 ? 和：运算符。用圆括号可确保正确分析所有表达式。在 er_print 命令行上，不能将表达式拆分为多行。在脚本中或命令行上，如果表达式包括空格，则必须将它括在双引号内。

过滤表达式的求值为布尔值时，如果应该包括数据包，则为求值为真；如果不应该包括数据包，则求值为假。线程、 LWP、 CPU、实验 id、进程 pid 和样例过滤基于相应关键字和整数之间的关系表达式，或使用 IN 运算符和逗号分隔的整数列表。

通过在 TSTAMP 和时间 （指定为整数纳秒，从处理其数据包的实验启动算起）之间指定一个或多个关系表达式来使用时间过滤。使用 overview 命令可以获得样例时间。overview 命令中的时间被指定为以秒为单位的时间，必须将其转换为纳秒以用于时间过滤。也可以从分析器中的 "时间线" 显示获得时间。

函数过滤可以基于叶函数或堆栈中的任何函数。按叶函数进行过滤是通过 LEAF 关键字和整型函数 id 之间的关系表达式指定的，或者使用 IN 运算符和构造 FNAME("*regex*") 指定的，其中 *regex* 是 regex(5) 手册页上指定的正则表达式。函数的整个名称（如 *name* 的当前设置所指定）必须匹配。

基于调用栈中任意函数的过滤，是通过确定构造 FNAME("*regex*") 中的任意函数是否在关键字 STACK:(FNAME("myfunc") SOME IN STACK) 表示的函数数组中指定的。

数据对象过滤与堆栈函数过滤类似，使用 DOBJ 关键字和括在圆括号中的构造 DNAME("*regex*")。

内存对象过滤是使用内存对象的名称 （如 mobj_list 命令中所示）以及对象的整型索引或对象集的索引指定的。（<Unknown> 内存对象的索引是 -1。）

数据对象过滤和内存对象过滤仅对具有数据空间数据的硬件计数器数据包有意义；这样的过滤不包括所有其他数据包。

虚拟地址或物理地址的直接过滤由 VADDR 或 PADDR 与地址之间的关系表达式指定。

内存对象定义 （请参见第 118 页的 "mobj_define mobj_type index_exp"）使用其求值为整型索引的表达式 （求值时使用 VADDR 关键字或 PADDR 关键字）。定义仅适用于内存计数器和数据空间数据的硬件计数器数据包。对于 <Unknown> 内存对象，表达式应该返回一个整数或 -1。

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。但请记住，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。源代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 具有许多内部软件层，但是该示例显示了仅查看入口点的一种方法。可能存在一些重复符号，可以忽略。

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

第 7 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介绍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

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、目标生命周期中各种事件的记录，以及目标的字大小。
- 映射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例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 136 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 7-1 数据类型和相应的文件名称

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

对于基于时钟的分析或硬件计数器溢出分析，将数据写入时钟周期或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 环境变量干预的 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。

实验可以选择性地包含文件名为 notes 的 ASCII 文件。使用 collect 命令的 -C comment 参数收集时，会自动创建此文件。创建实验后，可以手动编辑或创建此文件。文件的内容会置于实验标题之前。

archives 目录

每个实验都有一个 archives 目录，其中包含了描述 loadobjects 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 实用程序生成的。如果该进程异常终止，则 er_archive 实用程序不会被调用，这种情况下，首先在实验上调用时通过 er_print 实用程序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，x 代表执行，c 代表组合）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，但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 loadobjects 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代码时需要该副本。

Java 实验

Java 实验在 loadobjects 文件中就有附加记录，这些记录既是 JVM 软件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。

此外， Java 实验还具有一个 JAVA_CLASSES 文件，该文件包含有关所有调用用户 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 loadobjects 文件中 Java 编译的方法记录。

记录实验

您可以用三种不同方法记录实验：

- 用 collect 命令
- 用 dbx 创建进程
- 用 dbx 从正在运行的进程创建实验

可以在分析器 GUI 中的 “性能工具收集”窗口运行 collect 实验； IDE 中的 “收集器”对话框运行 dbx 实验。

collect 实验

使用 collect 命令记录实验时，collect 实用程序会创建实验目录并设置 LD_PRELOAD 环境变量，以确保将 libcollector.so 预加载到目标的地址空间。然后该程序设置环境变量，通知 libcollector 实验名称、数据收集选项，并执行自己顶部的目标。

libcollector.so 负责写入所有实验文件。

创建进程的 dbx 实验

在启用数据收集的情况下用 dbx 启动进程时， dbx 还会创建实验目录，并确保 libcollector.so 的预加载。 dbx 在其第一个指令之前的断点处停止进程，然后调用 libcollector.so 中的初始化例程以启动数据收集。

Java 实验不能用 dbx 收集，因为 dbx 使用 Java™ 虚拟机调试接口 (JVMDI) 代理进行调试，而该代理无法与数据收集所需的 Java™ 虚拟机分析接口 (JVMPI) 代理共存。

dbx 实验，在运行的进程上

在正运行的进程上用 dbx 启动实验时，它会创建实验目录，但不能使用 LD_PRELOAD 环境变量。 dbx 对 tolopen libcollector.so 目标发出交互式函数调用，然后调用 libcollector.so 初始化例程 （与创建进程时相同）。与收集实验中一样，数据由 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。

如果在使用 dbx 启动进程之前或在使用 dbx 附加到运行的进程之前，已经显式预加载了 libcollector.so，则可以收集跟踪数据。

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 的信息请参见 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。每节还将介绍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

基于时钟的分析的事件特定数据由分析间隔计数的数组组成。在 Solaris 操作系统上，间隔计数器是在分析间隔结束时提供的，正确的间隔计数器按 1 递增，然后调度另一个分析信号。仅当 Solaris LWP 线程进入 CPU 用户模式时，才记录并重置数组。重置数组包括将用户 CPU 状态的数组元素设置为 1，并将所有其他状态的数组元素设置为 0。在重置数组之前进入用户模式时记录数组数据。因此，数组将包含自上一次进入用户模式以来

进入的每个微态的计数累计，以及内核为每个 Solaris LWP 维护的 10 个微态中的每个微态的计数累计。在 Linux 操作系统上，微态不存在；仅有的间隔计数器是“用户 CPU 时间”。

调用栈与数据同时记录。如果 Solaris LWP 在分析间隔结束时未处于用户模式，只有 LWP 或线程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。

Solaris 操作系统上每个微态贡献的度量如表 7-2 中所示。

表 7-2 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	在任何其他陷阱中运行	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	其他等待时间
LMS_USER_LOCK	等待用户模式锁定的休眠	用户锁定时间
LMS_SLEEP	任何其他原因的休眠	其他等待时间
LMS_STOPPED	停止（/proc、作业控制或 lwp_stop）	其他等待时间
LMS_WAIT_CPU	等待 CPU	等待 CPU 时间

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。因此应该在足够长的时间内尽量多次地运行程序，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以及程序通过系统进行的方法都会引起特定的错误。下面是定时度量可能不准确或失真的一些情况：

- Solaris LWP 或 Linux 线程创建时，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。如果创建了多个 LWP 或线程，则其中的错误会是分析间隔的几倍。
- Solaris LWP 或 Linux 线程被销毁时，记录最后的分析包会消耗一些时间。如果销毁了多个 LWP 或线程，则其中的错误会是分析间隔的几倍。
- LWP 或线程的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。在多个 Solaris LWPs 或 Linux 线程（而不是多个处理器）运行它们时，错误可能更大。

- 程序可以按与系统时钟关联的方式运行。这种情况下，在 Solaris LWP 或 Linux 线程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，分析信号可以引入关联：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。在 Solaris 操作系统上，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。此问题只会出现在 Solaris 8 操作系统上的默认线程库中。
- 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。时钟调整会使分析包丢失。包含的时间周期通常是几秒，并且以某个增量进行调整。

除刚刚介绍的不准确外，时间度量会因收集数据的进程而失真。程序的度量中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初始化。（这是有关的另一个实例）。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他度量的过多计数。记录数据所用的时间总和通常少于默认分析间隔 CPU 时间的百分之几。

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他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 Solaris LWP 或 Linux 线程时间总和通常被精确到百分之几。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。但是，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 Solaris 操作系统上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，因为 `gethrvtime()` 返回的是 LWP 的值，而线程可能随 LWP 的不同而不同。

性能分析器中报告的 LWP 时间可以与 `vmstat` 报告的时间有很大差别，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是以进程文件系统 `/proc` 使用报告为基础的，因此微态中所用时间的记录具有很高的准确性。有关详细信息，请参见 `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。但是，显示在“统计”标签中的值可以包括其他基值，且这些基值不包含在 `<Total>` 的定时度量值中。这些基值来自以下源代码：

- 由系统创建的未分析的线程。Solaris 8 操作系统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处于休眠状态，休眠时间在 “统计数据” 标签中显示为 “其他等待” 时间。
- 暂停数据收集所在的时间周期。

用户 CPU 时间与硬件计数器循环时间不同，因为在将 CPU 模式切换到系统模式时硬件计数器会关闭。有关更多的详细信息，请参见第 143 页的 “陷阱”。

同步等待跟踪

同步等待跟踪仅在 Solaris 平台上可用。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）不能执行任何其他工作。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）可以在其上具有其他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 都处于繁忙状态，则 “用户锁定时间” 为零。但是，“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，可以将开销中所用的等待时间部分减到最少。

硬件计数器溢出分析

硬件计数器溢出分析仅在 Solaris 平台上可用。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。尤其对循环和指令计数器来说，该值可能会更大，这些计数器的递增频率比诸如浮点运算或缓存未命中的计数器快。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。有关更多的详细信息，请参见第 180 页的 “硬件计数器溢出的属性”。另请参见第 143 页的 “陷阱” 的介绍。陷阱和陷阱处理程序会使报告的 “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，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。不记录使用不同机制的 Java 内存管理的跟踪数据。

被跟踪的函数可以从许多库中的任意一个库装入。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。额外的时间将用于记录跟踪数据。

数据空间分析

数据空间分析是一个数据集合，在其中根据导致事件（而不仅仅是之中发生与内存相关的事件的指令）的数据对象引用报告与内存相关的事件，例如缓存缺失。数据空间分析在 Linux 系统中不可用。

要进行数据空间分析，目标必须是使用 -xhwcprof 标记和 -xdebugformat=dwarf -g 标记为 SPARC 体系结构编译的 C 程序。而且，收集的数据必须是与内存相关的计数器的硬件计数器分析，并且可选的 + 号必须放在计数器名称之前。目前，性能分析器包括两个与数据空间分析相关的标签（DataObject 标签和 DataLayout 标签）和用于内存对象的各种标签。

MPI 跟踪

MPI 跟踪仅在 Solaris 平台上可用。 MPI 跟踪记录了对 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。通过干预调用 MPI 库来执行跟踪。干预的函数不含有关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。
- 每个进程与其他进程通信，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。

对于调用 `MPI_Bcast()`

- 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

对于调用 `MPI_Allreduce()`

- 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

对于调用 `MPI_Reduce_scatter()`

- 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，请参见第 156 页的 “不完整的堆栈解除”。

调用栈中的分支 PC 用于将独占度量从性能数据分配到该 PC 所在的函数。堆栈上的每个 PC （包括分支 PC）用于将非独占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。但是，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。与这种情况相关的详细信息请参见第 157 页的 “将地址映射到程序结构”。

单线程执行和函数调用

最简单的程序执行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() 的前面。

程序运行后，遇到分支指令之前指令按顺序执行，在其他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，控制权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。但是，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，且控制权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，控制权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修补后续调用的 PLT 地址。

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，并且独占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，则由 PLT 生成的任何 PC 和 ld.so 中的代码通过对聚集包括时间的人工函数 @plt 的调用来替换。每个共享对象有一个这种人工函数。如果程序使用 LD_AUDIT 接口，则 PLT 条目可能永远不会得到修补，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显示为调用系统函数 sigacthandler()，而 sigacthandler() 又显示为调用用户的信号处理程序。 sigacthandler() 和任何用户信号处理程序的非独占度量和调用的任何其他函数显示为中断函数的非独占度量。

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，而收集硬件计数器溢出数据时是 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 平台上用于某些寄存器内容值的 `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， Solaris LWP 或 Linux 内核进入系统模式。在 Solaris 操作系统上，微态通常从 “用户 CPU”状态切换到 “陷阱”状态再到 “系统”状态。根据微态切换的位置，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。

在其他情况下，也需要重点考虑提供尽可能有效的陷阱处理。这些示例中的一些是 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。但是，因为 CPU 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。该优化的目的是减少堆栈的大小，（在 SPARC 平台上时）减少使用寄存器窗口。

假设程序源代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 或更高级别上。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 或更高级别上。

显式多线程

在 Solaris 造作系统中，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()`，与线程创建函数的调用方没有任何联系。因此与创建的线程关联的非独占度量仅传送到 `_thread_start()` 和 `<Total>` 函数。

除创建线程外，线程库还在 Solaris 上创建 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。

- 如果使用有界线程，则线程库为每个线程创建一个 LWP。
- 如果使用无界线程，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 8 操作系统中的标准线程库创建了一些线程用来执行信号处理和其他任务。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。但是这些线程中所用的时间包括在进程统计中和样例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。

Linux 操作系统为显式多线程提供了 P 线程（POSIX 线程）。数据类型 `pthread_attr_t` 可以控制线程的行为属性。要创建绑定线程，必须使用 `pthread_attr_setscope()` 函数将属性的作用域设置为 `PTHREAD_SCOPE_SYSTEM`。在默认情况下，或者将属性的作用域设置为 `PTHREAD_SCOPE_PROCESS` 时，线程是无界的。为创建新线程，应用程序调用 P 线程 API 函数 `pthread_create()`，将指针作为函数参数之一传递到应用程序定义的启动例程。新线程开始执行时，在 Linux 特定系统函数 `clone()` 中运行，该函数会调用另一个内部初始化函数 `pthread_start_thread()`，然后，该内存初始化函数又调用最初传递到 `pthread_create()` 的用户定义的启动例程。可用于收集器的 Linux 度量收集函数是特定于线程的，而与线程是否绑定到 LWP 无关。因此，`collect` 实用程序运行时，会在 `pthread_start_thread()` 和应用程序定义的线程启动例程之间插入一个度量收集函数 `collector_root()`。

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他程序。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他方法。

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 JVM 软件的进程。JVM 软件从 C++ 源代码编译，并在调用 `main` 之类的函数 `_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。可以指定的操作是本机共享对象的动态装入，而对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*。其他方法可以通过 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，有一个解释的版本，还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以 C、 C++ 或 Fortran 编写的本机编译代码；此类调用的目标称为*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。其中一个区域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，数据空间中生成其他代码来执行解释和编译代码之间的转换。

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都具有一个 JVM 软件线程。 Java 应用程序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。对于 Solaris 8 操作系统上的默认线程库 `libthread.so`，可能随时取消调度某个线程，或者将某个线程调度到 LWP。线程未调度到 LWP 上时不收集线程的数据。使用 Solaris 8 操作系统上的可替换的 `libthread.so` 库或使用 Solaris 9 操作系统线程时，不会取消调度线程。

在 J2SE 1.4.2 中的 JVMPI 和 J2SE 5.0 中的 JVMTI 上，可以使用各种方法实现数据收集。

Java 调用栈和机器调用栈

通过记录进程的每个 Solaris LWP 或 Linux 线程生命期间的事件，性能工具将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的任何时刻，调用栈表示程序处于执行中的哪个位置，以及如何到达该位置。区别混合模型 Java 应用程序与传统 C、 C++ 和 Fortran 应用程序的一个重要方法是，混合模型 Java 应用程序在目标运行的任何时刻都有两个有意义的调用：Java 调用栈和机器调用栈。这两个调用栈都是在分析期间进行记录和调和的。

基于时钟的分析和硬件计数器溢出分析

用于 Java 程序的基于时钟的分析和硬件计数器溢出分析除了都收集 Java 调用栈和机器调用栈外，与用于 C、 C++ 和 Fortran 程序的基本相同。

同步跟踪

Java 程序的同步跟踪基于线程尝试获取 Java 监视器时生成的事件。对于这些事件，机器调用栈和 Java 调用栈都会被收集；但对于 JVM 软件中使用的内部锁定，不收集任何同步跟踪数据。

堆跟踪

堆跟踪数据记录对象分配事件 （通过用户代码生成）和对象释放事件 （通过垃圾收集器生成）。此外，所有对 C++ 内存管理函数 （如 `malloc` 和 `free`）的使用也生成记录的事件。这些事件可能来源于本机代码或 JVM 软件本身。

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。默认情况下， Java 表示显示在数据支持它的位置。以下部分总结了这三种表示之间的主要差异。

用户表示

用户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，可能有许多特定 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 表示中所有方法均被累计显示为单一的方法。在分析器中该表示被默认选定。

用于 Java 表示中的 Java 方法的 PC 对应于进入该方法的方法 ID 和字节码索引；用于本机函数的 PC 对应于机器 PC。用于 Java 线程的调用栈混合使用 Java PC 和机器 PC。它没有任何对应于 Java 内务处理代码（没有 Java 表示）的帧。在某些情况下， JVM 软件无法解除 Java 栈，而且会返回带有特殊函数 `<no Java callstack recorded>` 的单个帧。通常，它占总时间的比例不会超过 5-10%。

Java 表示中的函数列表显示针对调用的 Java 方法以及任何本机方法的度量。“调用方与被调用方”面板显示 Java 表示中的调用关系。

Java 方法的源代码对应于从中进行编译的 `.java` 文件（在每个源代码行上带有度量）中的源代码。任何 Java 方法的反汇编显示为它生成的字节码、每个字节码的度量，以及交错的 Java 源代码 （如果可用）。

Java 表示中的时间线仅表示 Java 线程。每个线程的调用栈与其 Java 方法一起显示。

所有 Java 程序可能具有显示同步，而显示同步通常由调用 `monitor-enter` 例程来执行。

Java 表示中的同步延迟跟踪基于 JVMPI 同步事件。来自正常同步跟踪的数据不显示在 Java 表示中。

当前不支持 Java 表示中的数据空间分析。

专家用户表示

“专家 Java 表示”类似于“Java 表示”，不同之处在于“Java 表示”中禁用的一些 JVM 内部详细信息显示在“专家 Java 表示”中。使用专家 Java 表示时，时间线显示所有线程；内部处理线程的调用栈是本机调用栈。

机器表示

“机器表示”显示的函数来自 JVM 软件本身，而不是来自正被 JVM 软件解释的应用程序。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。一些 JVM 帧表示在解释的 Java、编译的 Java 和本机代码之间的转换代码。

针对 Java 源代码显示已编译方法中的源代码；数据表示所选的已编译方法的特定实例。已编译方法的反汇编显示生成的机器反汇编代码，而不是 Java 字节码。调用方与被调用方关系显示所有开销帧，以及所有表示在已解释、已编译方法及本机方法之间的转换的所有帧。

机器表示中的时间线显示所有线程、LWP 或 CPU，并且每个时间线中的调用栈是机器表示调用栈。

在机器表示中，线程同步转换为对 `_lwp_mutex_lock` 的调用。由于不跟踪这些调用，因此不显示同步数据。

OpenMP 软件执行概述

OpenMP 应用程序的实际执行模型在 OpenMP 规范中有介绍（例如，请参见 OpenMP 应用程序接口 2.5 版 1.3 节）。但是该规范并没有介绍那些对用户来说可能很重要的实现详细信息，而且在 Sun Microsystems 的实际实现中，用户也很难从直接记录的分析信息中了解线程是如何交互的。

当任意单线程程序运行时，其调用栈会显示该程序当前位置，以及如何到达那里的跟踪，跟踪从名为 `_start` 的例程（该例程调用 `main`，后者又继续调用程序内的各种子例程）中的起始指令开始。如果子例程包含循环，则程序会重复执行循环内的代码，直到符合循环退出条件为止。然后继续执行下一序列代码，依此类推。

通过 OpenMP（或自动并行化）并行化程序时，行为是不同的。该行为的直观模型具有像单线程程序那样执行的主线程。当它到达并行循环或并行区域时，将出现附加从属线程（其中每个从属线程都是主线程的克隆），它们都并行执行循环或并行区域的内容，各自完成工作的不同部分。在完成工作的所有部分后，所有线程会被同步，从属线程消失，主线程继续运行。

当编译器为并行区域或循环（或任何其他 OpenMP 构造）生成代码时，将提取它内部的代码，并使之成为一个独立的函数（称为 m 函数）。（也可以将它称为外联函数或循环体函数。）函数的名称对 OpenMP 构造类型、从中提取它的函数名称以及该构造所在源代码行的行号进行编码。这些函数的名称在分析器中按以下形式显示（方括号中的名称是函数的实际符号表名称）：

```
bardo_ -- OMP parallel region from line 9 [_$p1C9.bardo_]
atomsum_ -- MP doall from line 7 [_$d1A7.atomsum_]
```

还有其他形式的此类函数，它们是从其他源代码构造派生的，名称中的 `OMP parallel region` 被替换为 `MP construct`、`MP doall` 或 `OMP sections`。在下面的介绍中，所有这些都统称为“并行区域”。

执行并行循环内代码的每个线程都可以多次调用其 m 函数，每次调用完成循环内的工作部分。完成工作的所有部分后，每个线程调用库中的同步或约简例程；然后主线程继续，而从属线程变为空闲状态，等待主线程进入下一并行区域。所有调度和同步都是通过调用 OpenMP 运行环境处理的。

在其执行过程中，并行区域内的代码可能完成部分工作，或者它可能与其他线程同步或选择要完成的其他工作部分。它还可能调用其他函数，而这些函数又会调用其他的函数。在并行区域内执行的从属线程（或主线程）可能会将其自身充当主线程，或者通过其调用的一个函数来充当主线程，接着进入它自己的并行区域，从而导致嵌套并行操作。

分析器基于调用栈的统计抽样收集数据，跨所有线程聚集其数据，并对照函数、调用方与被调用方、源代码行和指令基于收集的数据类型来显示性能度量。它提供有关用户模式或机器模式下 OpenMP 程序性能的信息。（支持第三种模式“专家模式”，但该模式与用户模式相同。）

OpenMP 分析数据的用户模式显示

分析数据的用户模式表示试图提供信息，如第 147 页的“OpenMP 软件执行概述”中所述的模型真正执行程序。实际的数据捕获运行时库 `libmtsk.so`（与模型不对应）的实现详细信息。在用户模式下，分析数据的表示被更改为更好地匹配模型，且在以下三个方面与记录的数据和机器模式表示不同：

- 从 OpenMP 运行时库的角度来看，构造的人工函数表示每个线程的状态。
- 操作调用栈以报告对应于代码运行方式模型的数据，如上所述。
- 为基于时钟的分析实验构造另外两个性能度量，对应于完成有用工作所用的时间和在 OpenMP 运行环境中等待所用的时间。

人工函数

构造人工函数，并将其放置到用户模式调用栈上，反映线程在 OpenMP 运行时库内处于某个状态的事件。

以下是几个人工函数的定义，每个函数后面是其功能描述：

- *<OMP-overhead>* — 在 OpenMP 库中执行
- *<OMP-idle>* — 从属线程，等待工作
- *<OMP-reduction>* — 执行约简操作的线程
- *<OMP-implicit_barrier>* — 在隐式屏障处等待的线程
- *<OMP-explicit_barrier>* — 在显式屏障处等待的线程
- *<OMP-lock_wait>* — 等待锁定的线程
- *<OMP-critical_section_wait>* — 等待进入临界段的线程
- *<OMP-ordered_section_wait>* — 等待轮流进入有序段的线程

当线程所在的 OpenMP 运行时状态与其中一个函数对应时，会将对应函数作为堆栈上的叶函数添加。当线程的叶函数处于 OpenMP 运行环境中的任意位置时， *<OMP-overhead>* 将作为叶函数替换它。否则，在用户模式堆栈中将忽略来自 OpenMP 运行环境的所有 PC。

用户模式调用栈

了解该模型的最简单方法是查看 OpenMP 程序在其执行过程中各个时刻的调用栈。本节以一个简单程序为例，该程序包括调用一个子例程 foo 的主程序。该子例程具有单个并行循环，线程在其中完成工作，争用、获得和释放锁定，以及进入和离开临界段。显示另一组调用栈，反映从属线程调用了另一函数 bar（它进入嵌套的并行区域）时的状态。

在此演示中，并行区域中所用的所有非独占时间都包括在从中提取它的函数的非独占时间中，其中包括在 OpenMP 运行环境中所用的时间，而且该非独占时间一直传递到 main 和 _start

表示此模型中行为的调用栈如后面的子章节所示。并行区域函数的实际名称为以下形式，如上所述：

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

为清晰起见，在描述中将使用以下简化形式：

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

在描述中，来自所有线程的调用栈都在执行程序期间的某个时刻显示。每个线程的调用栈都显示为帧栈，将分析器 “时间线”标签中为单个线程选择单独分析事件的数据与顶部的叶 PC 匹配。在 “时间线”标签中，每个帧都显示有 PC 偏移（下面将省略）。来自所有线程的堆栈会在水平数组中显示，而在分析器 “时间线”标签中，其他线程的堆栈将出现在垂直堆叠的分析栏中。此外，在提供的表示中，将显示所有线程的堆栈，它们看上去像是在同一时刻捕获的，而在实际的实验中，堆栈是在每个线程中独立捕获的，并且彼此可能是相对偏离的。

显示的调用栈表示该数据是通过分析器或 er_print 实用程序中的用户视图模式呈现的。

1. 在第一个并行区域之前

在进入第一个并行区域之前，只有一个线程（即主线程）。

主线程
foo
main
_start

2. 在进入第一个并行区域时

此时，库已经创建了从属线程，而且所有线程（主线程和从属线程）即将开始处理各自的工作部分。从构造的 OpenMP 指令所在行上的 foo，或者从包含自动并行化的循环语句的行上，所有线程都显示为调用到并行区域 foo-OMP... 的代码中。每个线程中的并行区域代码将从并行区域中的第一个指令调用到 OpenMP 支持库（显示为 *<OMP-overhead>* 函数）中。

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

显示的 <OMP-overhead> 窗口可能很小，所以该函数可能不出现在任何特定实验中。

3. 在并行区域内执行时

所有四个线程都在并行区域中完成有用的工作。

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

4. 在并行区域内执行，且在工作部分之间时

所有四个线程都在做有用的工作，但是其中一个线程已完成一部分工作，并正在获取其下一部分工作。

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

<OMP-overhead> 不大可能出现在实际的实验中。

5. 在并行区域内的临界段中执行时

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

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

收集的数据不能区分正在临界段中执行的线程的调用栈和尚未到达或已经通过临界段的线程的调用栈。

6. 在并行区域内的锁定周围执行时

锁定周围的代码段与临界段完全类似。所有四个线程都正在并行区域内执行。一个线程在持有锁定的同时执行，一个线程在获得锁定之前 （或者在获得再释放锁定之后）执行，而其他两个线程正在等待锁定。

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

主线程	从属线程 1	从属线程 2	从属线程 3
foo	*foo*	*foo*	*foo*
main	*main*	*main*	*main*
_start	*_start*	*_start*	*_start*

与临界段示例中一样，收集的数据不能区分持有锁定并执行的线程的调用栈和在获得锁定之前或在释放锁定之后执行的线程的调用栈。

7. 在并行区域末端附近

此时，其中三个线程已完成其工作的所有部分，而其中一个线程仍在工作。这种情况下的 OpenMP 构造隐式指定了一个屏障；如果用户代码显式指定了屏障，则 `<OMP-implicit_barrier>` 函数将由 `<OMP-explicit_barrier>` 替换。

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

8. 在并行区域末端附近，具有一个或多个约简变量

此时，其中两个线程已完成其工作的所有部分，正在执行约简计算，但其中一个线程仍在工作，第四个线程已完成其约简部分，正在屏障处等待。

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

尽管在 `<OMP-reduction>` 函数中显示了一个线程，但是执行约简所用的实际时间通常很短，因此在调用栈样例中很少能捕获到。

9. 在并行区域的末端

此时，所有线程都完成了并行区域内工作的所有部分，并且已到达屏障。

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

由于所有线程都已到达屏障，它们可能都会继续，所以实验不大可能找到处于此状态的所有线程。

10. 离开并行区域之后

此时，所有从属线程都在等待进入下一并行区域，它们处于空转或休眠状态，具体状态取决于用户设置的各种环境变量。以下程序是以串行方式执行的。

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

11. 在嵌套的并行区域中执行时

所有四个线程都在工作，每个线程都在外部并行区域中。其中一个从属线程已调用另一函数 bar，并已创建了一个嵌套的并行区域，而且会创建一个附加从属线程以便与它一起工作。

主线程	从属线程 1	从属线程 2	从属线程 3	从属线程 4
	bar-OMP...			bar-OMP...
	bar			bar
foo-OMP...	foo-OMP...	foo-OMP...	foo-OMP...	foo-OMP...
foo	foo	foo	foo	foo
main	main	main	main	main
_start	_start	_start	_start	_start

OpenMP 度量

在处理 OpenMP 程序的时钟分析事件时，将显示两种度量，它们分别对应于 OpenMP 系统中两种状态所用的时间。这两种状态是 “OMP 工作”和 “OMP 等待”。

每当从用户代码执行线程时（不管串行执行还是并行执行），都会按“OMP 工作”累积时间。每当线程在等待某项之后才能继续时，不管等待是忙等待（空转等待）还是休眠，都会按 “OMP 等待”累积时间。这两种度量之和与时钟分析中的 “LWP 时间总计”度量相匹配。

OpenMP 分析数据的机器表示

在执行的各个不同阶段中程序的实际调用栈与上面在直观模型中描述的不大相同。演示的机器模式将调用栈显示为已度量，没有执行转换，且没有构造人工函数。但是仍显示时钟分析度量。

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

1. 在第一个并行区域之前

在进入第一个并行区域之前，只有一个线程 （即主线程）。调用栈与在用户模式下相同。

主线程
`foo`
`main`
`_start`

2. 在并行区域中执行期间

主线程	**从属线程 1**	**从属线程 2**	**从属线程 3**
`foo-OMP...`			
`libmtsk`			
`foo`	`foo-OMP...`	`foo-OMP...`	`foo-OMP...`
`main`	*`libmtsk`*	*`libmtsk`*	*`libmtsk`*
`_start`	`_lwp_start`	`_lwp_start`	`_lwp_start`

在机器模式下，从属线程显示为在 `_lwp_start` 中启动，而不是在 `_start`（主线程在其中启动）中启动。（在线程库的一些版本中，该函数可能显示为 `_thread_start`。）

3. 所有线程都在屏障处时

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

与线程在并行区域中执行时不同，当线程在屏障处等待时，在 foo 和并行区域代码 foo-OMP... 之间没有来自 OpenMP 运行环境的帧。原因是在实际的执行中不包括 OMP 并行区域函数，但是 OpenMP 运行环境操作寄存器，以便堆栈解除显示从最后执行的并行区域函数到运行时屏障代码的调用。如果没有它，在机器模式下将无法确定哪个并行区域与屏障调用有关。

4. 离开并行区域之后

主线程	从属线程 1	从属线程 2	从属线程 3
foo			
main	*libmtsk*	*libmtsk*	*libmtsk*
_start	_lwp_start	_lwp_start	_lwp_start

在从属线程中，用户帧都不在调用栈上。

5. 在嵌套的并行区域中时

主线程	从属线程 1	从属线程 2	从属线程 3	从属线程 4
	bar-OMP...			
foo-OMP...	*libmtsk*			
libmtsk	bar			
foo	foo-OMP...	foo-OMP...	foo-OMP...	bar-OMP...
main	*libmtsk*	*libmtsk*	*libmtsk*	*libmtsk*
_start	_lwp_start	_lwp_start	_lwp_start	_lwp_start

不完整的堆栈解除

堆栈解除可能由于以下原因而失败：

- 堆栈已被用户代码破坏；如果是这样，可能在程序或数据收集代码中发生核心转储，这取决于堆栈如何被破坏。
- 用户代码不遵循函数调用的标准 ABI 惯例。尤其在 SPARC 平台上，执行保存指令之前返回寄存器 %o7 改变时。

  在任何平台上，手写汇编程序代码可能违反惯例。
- 在 x86 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 x86 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栈弹出之后且在函数返回之前，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。在这种情况下，实验中不会记录调用栈中一些来自 `_start` 的函数的 PC。人工函数 `<Truncated-stack>` 显示为从 `<Total>` 调用，以清点记录的最顶端帧。
- 在 x86 和 x64 平台上，如果代码不保留帧指针，则堆栈解除可能难以完成。要保留帧指针，请用 `-xreg=no%frameptr` 选项进行编译。

中间文件

如果使用 `-E` 或 `-P` 编译器选项生成了中间文件，则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。

如果没有从不含引用行号的函数到编译生成该函数的源文件的指令，则在注释源代码中显示以下代码行：

function_name -- < 没有行号的指令 >

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目标文件被移动、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他文件的 #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，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）。本节介绍了这些映射。

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，一些区域是文本，表示可执行文件的指令；而一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。其他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新装入，则不同时间的不同 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、函数起源的对象模块附加信息，以及将地址关联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 编程语言中所用的方法等等。函数在源代码中描述得很清晰，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 和所有其他 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。其他未对应到源模型的函数在以下部分介绍。

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。其他的函数从不执行，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

尽管有别名的函数反映相同代码段的多个名称，但是在某些情况下，多个代码段具有相同的名称：

- 有时，由于模块性的原因，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分析器中。在“汇总”标签中，给出这些函数的每个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代码、反汇编代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您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，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。分析器不能区分连续的剥离静态函数和单一的剥离静态函数，因此会有两个或两个以上的剥离静态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，分析器不会知道保存地址，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显示为被调用方。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常量参数函数的调用。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修整名称。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，所以注释反汇编代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，所以注释源代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。建议使用后者。
- 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，但是更为常见的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非独占度量 （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独占度量，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，但其他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 147 页的 “OpenMP 软件执行概述”中已介绍过。

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。它们的独占度量和非独占度量表示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非独占度量。产生这种情况的方法在第 147 页的 “OpenMP 软件执行概述”中已介绍过。

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编译期间创建。它们表示非正常执行的代码。具体来讲，指的是用于生成最终优化编译反馈的训练运行期间未被执行的代码。典型示例是针对从库函数返回的值执行错误检查的代码；错误处理代码一般情况下是不会运行的。要改善页面和指令缓冲行为，就要将这种代码移动到地址空间的其他位置，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修整名称。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，分析器将人工调用从主函数输入到它的外联部分。

带有合适非独占和独占度量的外联函数做为普通函数显示。此外，外联函数的度量同外联化代码函数中的非独占度量一样被增加。

有关反馈优化编译的详细信息，请参见 《C 用户指南》的附录 B、《C++ 用户指南》的附录 A 或 《Fortran 用户指南》的第 3 章中对 `-xprofile` 编译器选项的描述。

动态编译函数

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 函数的信息请参见第 53 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。对于其他的方法，性能工具显示执行这些方法的 JVM 软件的信息。在 Java 表示中，所有的方法与解释版本合并。在机器表示中，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”函数的详细信息，请参见第 53 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 SPARC 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。这种情况最有可能的原因是解除失败，做为 PC 记录的值根本不是 PC，而是某些其他数据。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 JVM 软件内部时。

`<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。

新函数和 OpenMP 特殊函数

构造人工函数，并将其放置到用户模式调用栈上，反映线程在 OpenMP 运行时库内处于某个状态的事件。以下是几个人工函数的定义，每个函数的后面是其功能描述：

- *<OMP-overhead>* — 在 OpenMP 库中执行
- *<OMP-idle>* — 从属线程，等待工作
- *<OMP-reduction>* — 执行约简操作的线程
- *<OMP-implicit_barrier>* — 在隐式屏障处等待的线程
- *<OMP-explicit_barrier>* — 在显式屏障处等待的线程
- *<OMP-lock_wait>* — 等待锁定的线程
- *<OMP-critical_section_wait>* — 等待进入临界段的线程
- *<OMP-ordered_section_wait>* — 等待轮流进入有序段的线程

<JVM-System> 函数

在用户表示中， `<JVM-System>` 函数表示 JVM 软件执行操作所花费的时间而不是运行 Java 程序的时间。在该时间间隔中，JVM 软件执行诸如垃圾收集和 HotSpot 编译之类的任务。默认情况下，可以在函数列表中看到 `<JVM-System>`。

<no Java callstack recorded> 函数

`<no Java callstack recorded>` 函数类似于 `<Unknown>` 函数，但是对于 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 JVM 软件以获取相应的 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 软件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。

<Truncated-stack> 函数

分析器用于记录调用堆栈中的单个函数的度量所使用缓冲区大小是有限的。如果调用堆栈太大，使得缓冲区已满，则调用堆栈大小的任何增加将强制分析器删除函数分析信息。由于在大多数程序中，大量独占 CPU 时间花费在叶函数中，因此分析器将删除堆栈底部那些不太重要的函数（从入口函数 `_start()` 和 `main()` 开始）的度量。被删除函数的度量合并为一个人工函数 `<Truncated-stack>` 。`<Truncated-stack>` 函数也会显示在 Java 程序中。

<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，其他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，其数据可用于对其他函数的数据给出透视。在 "调用方与被调用方" 列表中，该函数不但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() 的名义调用方。如果堆栈解除是不完整的，则 <Total> 函数可以做为 <Truncated-stack> 的调用方显示。

与硬件计数器溢出分析有关的函数

以下函数与硬件计数器溢出分析有关：

- 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。该值被记录下来，并且计数器将重新设置。
- 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。该值被记录下来，并且计数器将重新设置。
- 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溢出分析的开销的计数。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*数据对象*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。数据对象的类型及其大小随源语言的不同而不同。许多数据对象在源程序中显式命名，而其他可能是未命名的。某些数据对象从其他 （简单的）数据对象派生或累计，产生丰富且通常会是复杂的数据对象集合。

每个数据对象都包含关联的范围，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（目标文件）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数据对象类型，并传送到包含数据对象和人工 <Total> 的任何累计，该人工函数被认为包含 <Unknown> 和 <Scalars> 在内的所有数据对象。<Unknown> 的不同子类型会传播到 <Unknown> 累计。下一节将介绍 <Total>、<Scalars> 和 <Unknown> 数据对象。

数据对象描述符

可以通过声明类型和名称的组合来完整描述数据对象。简单的标量数据对象 {int i} 描述了类型为 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。类型名称中的空格将替换为下划线（_），而未命名的数据对象用破折号（-）的名称表示，例如：{double_precision_complex -}。

完整累计被类似地表示为 foo_t 类型的结构 \{structure:foo_t foo\}。累计的元素需要其容器的附加规范，例如 {structure:foo_t}.{int i} 中成员为 i，类型为 int，使用的是类型为 foo_t 的前一个结构。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。

在完全限定的描述符可能不总需要消除 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> 数据对象

<Total> 数据对象是人工的结构，用于将程序的数据对象表示为一个整体。除正被归属到不同数据对象（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数据对象 <Total>。该函数显示在数据对象列表的顶部，其数据可用于对其他数据对象的数据给出透视。

<Scalars> 数据对象

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> 数据对象的度量值中。

<Unknown> 数据对象及其元素

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

- 具有触发器 PC 且未使用 -xhwcprof 编译的模块

无法识别事件引发的指令或数据对象，因为对象代码不是通过硬件计数器分析支持编译的。

- 回溯找不到有效的分支目标

  无法识别事件引发的指令，因为在编译对象中提供的硬件分析支持信息不足以验证回溯的有效性。

- 回溯已遍历分支目标

  无法识别事件引发的指令或数据对象，因为回溯在指令流中遇到了控制转移目标。

- 编译器未提供标识描述符

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

- 无类型信息

  回溯决定了事件可能导致的指令，但是编译器未将该指令识别为内存引用指令。

- 通过从编译器提供的符号信息未决定

  回溯决定了可能的临时内存引用指令，但编译器不标识该回溯，因此不可能确定有关的数据对象。编译器临时文件通常是未标识的。

- 跳转或调用指令阻止回溯

  无法识别事件引发的指令，因为回溯在指令流中遇到了分支或调用指令。

- 回溯找不到触发器 PC

  在最大回溯范围内找不到事件导致的任何指令。

- 无法确定 VA，因为寄存器在触发器指令后发生改变

  无法确定数据对象的虚拟地址，因为在硬件计数器暂停期间覆盖了寄存器。

- 内存引用指令未指定有效的 VA

  数据对象的虚拟地址看起来无效。

内存对象

内存对象是内存子系统中的组件，如缓存行、页面和内存库。对象是通过从记录的虚拟和 / 或物理地址计算的索引确定的。预定义的虚拟页面和物理页面的内存对象的大小可以为 8KB、64KB、512KB 和 4 MB。您可以用 `er_print` 实用程序中的 `mobj_define` 命令定义其他内存对象。也可以用分析器中的 “添加内存对象”对话框 （打开方法是单击 “设置数据表示”对话框中的 “添加定制对象”按钮）定义自定义内存对象。

第 8 章

了解带注释的源数据和反汇编数据

注释源代码和注释反汇编代码可用于确定函数中哪些源代码或指令会使性能降低，同时也可用于查看编译器如何对代码执行转换的注释。本节介绍了注释过程和与解释注释代码有关的问题。

带注释的源代码

实验的注释源代码可以在性能分析器中通过选择分析器窗口左窗格中的 “源码” 标签查看。或者，可以使用 er_src 实用程序，在不运行实验的情况下查看注释源代码。本手册的这部分介绍了源代码如何在性能分析器中显示。有关使用 er_src 实用程序查看注释源代码的详细信息，请参见第 187 页的 “在不运行实验的情况下查看源代码 / 反汇编代码”。

分析器中的注释源代码包含以下信息：

- 初始源文件中的内容
- 每行可执行源代码的性能度量
- 由于其度量超过特定阈值而突出显示的代码行
- 索引行
- 编译器注释

性能分析器“源码”标签的布局

“源码”标签分为若干列，其中左侧固定宽度的几列显示各个度量，右侧的其余部分显示注释源代码。

标识初始源代码行

在注释源代码中以黑色显示的所有代码行都来自初始源文件。注释源代码列中每行开始处的数字与初始源文件中的行号对应。以不同颜色显示其中字符的行可能是索引行，也可能是编译器注释行。

“源码”标签中的索引行

源文件是可编译生成目标文件或可解释为字节代码的任何文件。目标文件通常包含与源代码中的函数、子例程或方法对应的一个或多个可执行代码区域。分析器分析目标文件，标识每个作为函数的可执行区域，并尝试将它在对象代码中找到的函数映射到该对象代码关联的源文件中的函数、例程、子例程或方法。分析器成功执行分析后，它会在注释源代码文件中与对应对象代码中找到的函数的第一条指令处增加一条索引行。

注释源代码会为每个函数（包括内联函数）显示索引行，即使内联函数不显示在“函数”标签所显示的列表中。“源码”标签以红色斜体显示索引行，并将索引行中的文本用尖括号括起来。类型最简单的索引行与函数的默认上下文对应。任何函数的默认源上下文都被定义为该函数的第一条指令所属的源文件。以下示例显示了 C 函数 `icputime` 的索引行。

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

从以上示例可以看出，索引行紧跟在第一条指令行的后面。对于 C 源代码，第一条指令对应函数体开始处的左括号。在 Fortran 源代码中，每个子例程的索引行紧跟在包含 subroutine 关键字的代码行后面。同时，main 函数的索引行紧跟在应用程序启动时执行的第一条 Fortran 源代码指令的后面，如以下示例所示：

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

有时，分析器可能无法将它在对象代码中找到的函数与该对象代码关联的源文件中的任何编程指令映射到一起；例如，代码可能通过另一个文件（如头文件）被执行 #included 操作或内联操作。

如果对象代码的源代码不是该对象代码包含的函数的默认源上下文，与该对象代码对应的注释源代码将包含一条特殊索引行，该索引行交叉引用该函数的默认源上下文。例如，编译 synprog 演示程序会创建一个对象模块 endcases.o，该模块与源文件 endcases.c 对应。endcases.c 文件中的源代码声明函数 inc_func，该函数是在头文件 inc_func.h 中定义的。头文件通常采用这种方式包含内联函数定义。由于 endcases.c 声明函数 inc_func，但 endcases.c 中没有任何源代码行与 inc_func 的指令对应，所以 endcases.c 的注释源代码文件的顶部会显示一条特殊索引行，如下所示：

```
0.650 .650          <Function:inc_func, instructions from source file inc_func.h>
```

该索引行上的度量指示来自 endcases.o 对象模块的这部分代码（在源文件 endcases.c 内）没有行映射。

分析器还在定义了 inc_func 函数的 inc_func.h 中的注释源代码内增加了一条标准索引行。

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

类似地，如果函数有**可替换的**源上下文[1]，交叉引用该上下文的索引行会显示在**默认**源上下文的注释源代码中。

```
              142. inc_body(int n)
0.650 .650         <Function:inc_body, instructions from source file inc_body.h>
0.     0.     143. {
                    <Function:inc_body>
              144. #include "inc_body.h"
0.     0.     145. }
```

双击引用另一个源上下文的索引行，将在源代码窗口中显示该源文件的内容。

以红色文本显示的行还有特殊索引行，以及不是编译器注释的其他特殊代码行。例如，作为编译器优化的结果，可能会为对象代码中的某个函数创建一条特殊索引行，该索引行并不与任何源文件中编写的代码对应。有关详细信息，参见第 180 页的 “在源代码、反汇编和 PC 标签中的特殊代码行”。

编译器注释

编译器注释指示如何生成编译器优化代码。编译器注释行以蓝色文本显示，以便与索引行和初始源代码行区别开来。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。注释源代码被写入后，任何源代码行的编译器注释立即显示在源代码行的前面。

1. 替换源上下文由包含该函数指令的其他文件组成。这类上下文包括来自 include 文件的指令（如以上示例所示）以及来自内联成命名函数的函数的指令。

编译器注释描述了为优化源代码而对其进行的许多转换。这些转换包括循环优化、并行、内联和流水线作业。以下是一个编译器注释的示例。

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

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

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

请注意，第 51 行的注释包括循环注释，因为函数 setvalsmod() 包含循环代码，并且该函数已经内联。

以上摘录的代码显示编译器注释会在每行的结尾换行显示，这种情况不会出现在分析器的 “源码” 标签中，因为该标签不受宽度限制。

显示在 “源码” 标签中的编译器注释的类型可使用 “设置数据表示” 对话框中的 “源代码 / 反汇编” 标签进行设置；有关详细信息，请参见第 91 页的 “设置数据表示选项”。

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的表达式生成代码来改善性能。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，则注释源代码在不包含的分支内的行上显示度量。

循环优化

编译器可以执行多种类型的循环优化。其中一些比较常见的优化如下所示：

- 循环展开
- 循环剥离
- 循环交换
- 循环分裂
- 循环合并

循环展开是指在循环体内重复多次循环迭代，并相应调整循环索引的过程。随着循环体逐渐变大，编译器可以更有效地调度指令。同时减少了由于循环索引递增和条件检查操作所导致的开销。循环的其余部分可通过使用循环剥离进行处理。

循环剥离是指从循环删除一系列循环迭代，并将它们相应移动到循环的前面或后面的过程。

循环交换更改嵌套循环的顺序，以便最大程度地降低内存跨距，最大程度地提高缓存行命中率。

循环合并是指将相邻或位置接近的循环合并为一个循环的过程。循环合并的好处与循环展开的好处相似。此外，只要编译器有很多利用指令级并行操作的机会，如果在两个预先优化的循环中访问公共数据，循环合并可改进缓存局域性。

循环分裂与循环合并正相反：它将一个循环分成两个或更多的循环。如果循环中计算的数量过多，导致寄存器溢出，使性能下降，则有必要采用这种优化。如果循环包含条件语句，也可以使用循环分裂。有时可以将循环分为两种类型：一种带条件语句，另一种不带条件语句。这样在不带条件语句的循环中，可增大执行软件流水线作业的机会。

有时，对于嵌套的循环，编译器会先使用循环分裂来拆分循环，然后执行循环合并，用其他方式重新组合该循环，以达到提高性能的目的。这种情况下，您将看到如下所示的编译器注释：

```
    Loop below fissioned into 2 loops
    Loop below fused with loop on line 116
    [116]    for (i=0;i<nvtxs;i++) {
```

内联

利用内联函数，编译器将函数指令直接插入调用该函数的位置，而不必执行真正的函数调用。因此，与 C/C++ 宏类似，内联函数的指令会在每个调用位置重复。编译器使用较高的优化级别（4 和 5）执行显式内联或自动内联。内联以内存中的代码覆盖区变大为代价，节约了函数调用的代价，提供了可以优化寄存器使用和指令调度的更多指令以下是一个内联编译器注释的示例。

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

在分析器的“源码”标签中，编译器注释没有通过换行显示在两行上。

并行化

如果代码包含 Sun、 Cray 或 OpenMP 并行指令，则可以将其编译为在多个处理器上并行执行。编译器注释会指示执行过并行操作和尚未执行并行操作的位置，以及相应的原因。以下是一个执行并行化操作的计算机注释示例。

```
0.    6.324  9. c$omp  parallel do shared(a,b,c,n) private(i,j,k)

           Loop below parallelized by explicit user directive
           Loop below interchanged with loop on line 12
0.010 0.010[10]           do i = 2, n-1

           Loop below not parallelized because it was nested in a parallel loop
           Loop below interchanged with loop on line 12
0.170 0.170 11.              do j = 2, i
```

有关并行执行和编译器生成的主体函数的详细信息，参见第 147 页的“OpenMP 软件执行概述”。

注释源代码中的特殊代码行

在“源码”标签中还可以看到有关特殊情况的多种其他注释，它们或者显示为编译器注释，或者显示为和索引行相同颜色的特殊代码行。有关详细信息，参见第 180 页的“在源代码、反汇编和 PC 标签中的特殊代码行”。

源代码行度量

每行可执行代码的源代码度量都会在固定宽度的几列中显示出来。这些度量与在函数列表中的度量相同。您可以使用 .er.rc 文件更改实验的默认值；有关详细信息，请参见第 126 页的 “设置默认值的命令”。还可以在分析器中使用 “设置数据表示”对话框更改显示的度量和突出显示的阈值；有关详细信息，请参见第 91 页的 “设置数据表示选项”。

注释源代码在源代码行级别显示了应用程序的度量。该度量是通过使用记录在应用程序调用栈中的 PC （程序记数），并将每个 PC 映射到源代码行的方式生成的。要生成注释源文件，分析器首先确定在特定对象模块（.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。为了生成注释源代码，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，而且必须能够读取源文件来生成显示的注释副本。分析器在以下默认位置依次搜索源文件、目标文件和可执行文件，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
- 可执行文件或编译对象中记录的绝对路径名

默认位置可以使用 addpath 或 resetpath 指令更改，也可以使用分析器 GUI 更改。

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；而对于其他的优化，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，度量可被解释为原因是执行了该代码行。但是如果分支 PC 与执行的指令来自不同的源代码行，分支 PC 所属源代码行的度量至少有一些必须解释为在此行代码等待执行性，度量进行了累积。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他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 8-1 中说明。

表 8-1 注释源代码度量

度量	含义
（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。 • 代码在其他地方重复，并且编译器执行公用子表达式识别并标记带有其他副本代码行的所有指令。 • 编译器用不正确的行号来标记该行的指令。
`0.`	程序中的某些 PC 被标记为从该代码行派生，但没有数据引用这些 PC：它们从不会在被统计抽样或被跟踪的调用栈中。`0.` 度量不表示该行不执行，只是表示该行不以统计方式显示在分析数据包或跟踪数据包中。
`0.000`	至少该行的一个 PC 出现在数据中，但是计算的度量值舍入为零。
`1.234`	归属该行的所有 PC 的度量总计达到了显示的非零数值。

注释反汇编代码

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代码的函数的对象模块是否已知：

- 如果对象模块是未知的，则分析器只对指定函数的指令进行反汇编，并且不会在反汇编代码中显示所有源代码行。
- 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，并且记录了行号数据，则分析器可以根据显示首选项，交叉存取源代码和反汇编代码。
- 如果编译器已将任何注释插入对象代码中，并且如果设置了相应的首选项，则该代码也可以在反汇编代码中交叉存取。

反汇编代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
- 指令的十六进制表示 （如果要求）
- 指令的汇编程序 ASCII 表示

调用地址被尽可能解析为符号（如函数名称）。度量被显示在指令的代码行上，并且如果设置了相应的首选项，度量可以显示在任何交叉存取的源代码上。可能的度量值与描述的源代码注释的度量值是一样的，如表 8-1 所示。

通过 #included 命令包括在多个位置中的代码的反汇编代码列表将在每次代码被包含后重复执行一次反汇编指令。只有重复的反汇编代码块首次显示在某个文件中时，源代码才会交叉出现。例如，如果在名为 inc_body.h 的头文件中定义的代码块分别由以下四个函数执行 #included 操作：inc_body、inc_entry、inc_middle 和 inc_exit，则反汇编指令块将在 inc_body.h 的反汇编代码列表中出现四次，但是源代码只在四块反汇编指令的第一块中进行交错。切换到 “源码” 标签，可看到与每次重复的反汇编代码对应的索引行。

索引行可以显示在 “反汇编” 标签中。与 “源码” 标签不同的是，不能直接使用这些索引行进行导航。但是通过将光标放在紧跟在索引行下方的其中一条指令上，然后选择“源码”标签，可以转到该索引行中引用的文件。

对其他文件的代码执行了 #include 操作的文件，将把包含的代码显示为不与源代码交叉存取的原始反汇编指令。但是通过将光标放在这些原始反汇编指令的其中一条上，然后选择 “源码” 标签，将打开包含 #included 代码的文件。在显示此文件时选择 “反汇编” 标签将显示与源代码交叉存取的反汇编代码。

内联函数的源代码可以与反汇编代码交叉存取，但宏的源代码不能与反汇编代码交叉存取。

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。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。与源代码的第 *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

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。但是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。

接下来的几个小节将介绍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他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，所以这些指令根本未表示下一条要执行的指令。这还意味着在记录调用栈时，可能有多条指令被视为是下一条要执行的指令。

指令发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。因为链接程序以比缓冲代码行更精细的粒度强制指令对齐，所以在看上去不相关的函数中的更改可能会导致指令的不同排列。不同的对齐会引起性能的改善或降级。

下面列举了一种假设情况，即相同的函数在稍有不同的环境下进行编译和链接的情况。两个输出示例均为 `er_print` 实用程序的注释反汇编代码列表。其中示例的指令是相同的，但指令的对齐不同。

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。

```
   Excl.     Incl.
User CPU  User CPU
    sec.      sec.
                             1. static int
                             2. ifunc()
                             3. {
                             4.     int i;
                             5.
                             6.     for (i=0; i<10000; i++)
                                <function:ifunc>
   0.010     0.010              [ 6]    1066c:clr         %o0
   0.        0.                 [ 6]    10670:  sethi       %hi(0x2400), %o5
   0.        0.                 [ 6]    10674:  inc         784, %o5
                             7.         i++;
   0.        0.                 [ 7]    10678:  inc         2, %o0
## 1.360     1.360              [ 7]    1067c:cmp         %o0, %o5
## 1.510     1.510              [ 7]    10680:  bl,a        0x1067c
   0.        0.                 [ 7]    10684:  inc         2, %o0
   0.        0.                 [ 7]    10688:  retl
   0.        0.                 [ 7]    1068c:nop
                             8.     return i;
                             9. }
```

在该示例中，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。

```
   Excl.     Incl.
User CPU  User CPU
    sec.      sec.
                             1. static int
                             2. ifunc()
                             3. {
                             4.     int i;
                             5.
                             6.     for (i=0; i<10000; i++)
                                <function:ifunc>
   0.        0.                 [ 6]    10684:  clr         %o0
   0.        0.                 [ 6]    10688:  sethi       %hi(0x2400), %o5
   0.        0.                 [ 6]    1068c:inc         784, %o5
                             7.         i++;
   0.        0.                 [ 7]    10690:  inc         2, %o0
## 1.440     1.440              [ 7]    10694:  cmp         %o0, %o5
   0.        0.                 [ 7]    10698:  bl,a        0x10694
   0.        0.                 [ 7]    1069c:inc         2, %o0
   0.        0.                 [ 7]    106a0:retl
   0.        0.                 [ 7]    106a4:nop
                             8.     return i;
                             9. }
```

指令发布延迟

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其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的指令。

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（如发布的循环或指令），这种延迟无关紧要。对于诸如这些计数缓冲缺少或浮点运算的其他计数器，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。但是，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。

在源代码、反汇编和 PC 标签中的特殊代码行

外联函数

外联函数可以在反馈优化编译期间创建。在 “源码” 标签和 “反汇编” 标签中，外联函数显示为特殊的索引行。在 “源码” 标签中，在已转换为外联函数的代码块中会显示一条注释。

```
             Function binsearchmod inlined from source file ptralias2.c into the
            58.         if( binsearchmod( asize, &element ) ) {
0.240 0.240   59.               if( key != (element << 1) ) {
0.    0.      60.                   error |= BINSEARCHMODPOSTESTFAILED;
                  <Function:main -- outline code from line 60 [_$o1B60.main]>
0.040 0.040[ 61]                    break;
              62.               }
              63.           }
```

在“反汇编”标签中，外联函数通常显示在文件结尾。

```
                  <Function:main -- outline code from line 85 [_$o1D85.main]>
0.    0.            [ 85] 100001034:  sethi       %hi(0x100000), %i5
0.    0.            [ 86] 100001038:  bset        4, %i3
0.    0.            [ 85] 10000103c:or          %i5, 1, %l7
0.    0.            [ 85] 100001040:  sllx        %l7, 12, %l5
0.    0.            [ 85] 100001044:  call        printf !0x100101300
0.    0.            [ 85] 100001048:  add         %l5, 336, %o0
0.    0.            [ 90] 10000104c:cmp         %i3, 0
0.    0.            [ 20] 100001050:  ba,a        0x1000010b4
                  <Function:main -- outline code from line 46 [_$o1A46.main]>
0.    0.            [ 46] 100001054:  mov         1, %i3
0.    0.            [ 47] 100001058:  ba          0x100001090
0.    0.            [ 56] 10000105c:clr         [%i2]
                  <Function:main -- outline code from line 60 [_$o1B60.main]>
0.    0.            [ 60] 100001060:  bset        2, %i3
0.    0.            [ 61] 100001064:  ba          0x10000109c
0.    0.            [ 74] 100001068:  mov         1, %o3
```

外联函数的名称显示在方括号中，该函数对与外联的代码段有关的信息进行编码，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修整名称。分析器提供了函数名称的可读版本。有关详细信息，参见第 160 页的“外联函数”。

如果在收集有关应用程序的性能数据时调用了外联函数，则分析器将在注释反汇编中显示一条特殊代码行，以显示该函数的非独占度量。有关详细信息，请参见第 186 页的“包括度量”。

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。这些函数已在第 147 页的“OpenMP 软件执行概述”中介绍了。

编译器将修整名称分配到主体函数，这些函数对并行构造的类型、提取构造的函数名称、初始源代码中构造的起始行号和并行构造的序列号编码。这些修整名称因微任务化库的发行版本而异，但均会还原显示为更易于理解的名称。

以下显示的是函数列表中显示的一个典型的编译器生成的主体函数。

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

从上面的示例可以看出，最先显示的是提取构造的函数名称，接着是并行构造的类型，然后是并行构造的行号，最后以方括号显示编译器生成的主体函数的修整名称。同样，在反汇编代码中，也会生成一条特殊索引行。

```
                   <Function:psec_ -- OMP sections from line 9 [_$s1A9.psec_]>
0.    7.445       [24]     1d8cc:save         %sp, -168, %sp
0.    0.          [24]     1d8d0:ld           [%i0], %g1
0.    0.          [24]     1d8d4:tst          %i1
```

```
                   <Function:craydo_ -- MP doall from line 10 [_$d1A10.craydo_]>
0.    0.030       [ ?]     197e8:save         %sp, -128, %sp
0.    0.          [ ?]     197ec:ld           [%i0 + 20], %i5
0.    0.          [ ?]     197f0:st           %i1, [%sp + 112]
0.    0.          [ ?]     197f4:ld           [%i5], %i3
```

如果使用 Cray 指令，这种函数可能不与源代码行号关联。这种情况下，会在行号的位置显示 `[ ?]`。如果索引行显示在注释源代码中，则该索引行表示不带行号的指令，如下所示。

```
              9. c$mic  doall shared(a,b,c,n) private(i,j,k)

            Loop below fused with loop on line 23
            Loop below not parallelized because autoparallelization is not enabled
            Loop below autoparallelized
            Loop below interchanged with loop on line 12
            Loop below interchanged with loop on line 12
3.873 3.903       <Function:craydo_ -- MP doall from line 10 [_$d1A10.craydo_],
instructions without line numbers>
0.     3.903 10.            do i = 2, n-1
```

注 – 在实际显示时，索引行和编译器注释行并不折行。

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。收集器没有有关用 C 或 C++ 编写的动态编译函数的信息，除非用户使用收集器 API 函数 `collector_func_load()` 提供了所需的信息。“函数”标签、“源码”标签和“反汇编”标签中显示的信息取决于传递给 `collector_func_load()` 的信息，具体情况如下：

- 如果未提供信息（未调用 collector_func_load()），在函数列表中，动态编译和装入的函数会显示为 <Unknown>。分析器中既看不到函数源代码，也看不到反汇编代码。
- 如果未提供源文件名，也未提供行号表，但提供了函数名称、函数大小及其地址，函数列表中将显示动态编译和装入的函数名称及其度量。注释的源代码是可用的，而反汇编指令是可视的，只是其中的行号将指定为 [?]，以指示行号未知。
- 如果提供了源文件名，但未提供行号表，分析器显示的信息将与不提供源文件名所显示的信息类似，只是注释源代码的前面将显示一条特殊索引行，以指示该函数的指令不带行号。例如：

```
1.121 1.121       <Function func0, instructions without line numbers>
                1. #include         <stdio.h>
```

- 如果提供了源文件名和行号表，将按照显示编译函数的惯例在“函数”标签、“源码”标签和“反汇编”标签中显示动态编译和装入的函数及其度量。

有关收集器 API 函数的详细信息，请参见第 53 页的“动态函数和模块”。

对于 Java 程序，大部分方法由 JVM 软件解释。解释执行期间运行在单独线程上的 Java HotSpot 虚拟机监控性能。监控过程中，虚拟机可以决定使用已解释的一个或多个方法，生成它们的机器码，并执行更有效的机器码版本而不是解释原始代码。

对于 Java 程序，无需使用收集器 API 函数；分析器通过在该方法的索引行的下方使用一行特殊代码，在注释反汇编代码列表中表示 HotSpot 编译的代码的存在，如以下示例所示。

```
              11.    public int add_int () {
              12.        int        x = 0;
                  <Function:Routine.add_int()>
2.832 2.832        Routine.add_int() <HotSpot-compiled leaf instructions>
0.    0.           [ 12] 00000000: iconst_0
0.    0.           [ 12] 00000001: istore_1
```

反汇编列表只显示已解释的字节代码，而不显示编译的指令。默认情况下，在该特殊代码行的旁边显示已编译代码的度量。每行已解释字节代码的独占和非独占的 CPU 时间与所有独占和非独占的 CPU 时间总和不同。通常情况下，如果在多种情况下调用了该方法，已编译指令的 CPU 时间将大于已解释字节代码的 CPU 时间总和，原因是已解释的代码只在初次调用该方法时执行一次，而已编译的代码将在其后执行。

注释源代码不显示 Java HotSpot 编译的函数。而是显示一条特殊指令行，指示这些指令没有行号。例如，下面显示了与上例显示的反汇编代码片段对应的注释源代码：

```
                11.    public int add_int () {
2.832 2.832           <Function:Routine.add_int(), instructions without line numbers>
0.     0.       12.       int        x = 0;
                      <Function:Routine.add_int()>
```

Java 本机函数

本机代码是最初用 C、C++ 或 Fortran 编写，由 Java 代码通过 Java 本地接口 (JNI) 调用的已编译代码。以下示例是演示程序 jsynprog 关联的文件 jsynprog.java 的注释反汇编代码片段。

```
                5. class jsynprog
                  <Function:jsynprog.<init>()>
0.     5.504         jsynprog.JavaCC() <Java native method>
0.     1.431         jsynprog.JavaCJava(int) <Java native method>
0.     5.684         jsynprog.JavaJavaC(int) <Java native method>
0.     0.            [  5] 00000000: aload_0
0.     0.            [  5] 00000001: invokespecial <init>()
0.     0.            [  5] 00000004: return
```

由于本机方法不包含在 Java 源代码中，jsynprog.java 的注释源代码的前面使用一条特殊索引行显示了所有的 Java 本机方法，该索引行指出指令没有行号。

```
0.     5.504        <Function:jsynprog.JavaCC(), instructions without line numbers>
0.     1.431         <Function:jsynprog.JavaCJava(int), instructions without line
numbers>
0.     5.684         <Function:jsynprog.JavaJavaC(int), instructions without line
numbers>
```

注 – 在实际的注释源代码显示中，索引行并不折行。

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。这类调用的其中一个示例是对传递的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。

在注释源代码中，编译器注释将指示是否创建了克隆的函数：

```
          Function foo from source file clone.c cloned, creating cloned function
_$c1A.foo; constant parameters propagated to clone
0.   0.570 27.    foo(100, 50, a, a+50, b);
```

注 – 在实际的注释源代码显示中，编译器注释行并不折行。

克隆函数名称是标识特定调用的修整名称。在上面的示例中，编译器注释指出克隆的函数的名称为 _$c1A.foo。在函数列表中，该函数显示为：

```
0.350 0.550 foo
0.340 0.570 _$c1A.foo
```

因为每个克隆的函数具有不同的指令集合，所以注释反汇编代码列表单独显示克隆的函数。这些函数与任何源文件都没有关联，因此其指令也与任何源代码行号无关。以下显示了某个克隆的函数的注释反汇编代码的前几行。

```
                    <Function:_$c1A.foo>
0.   0.          [?]     10e98:save          %sp, -120, %sp
0.   0.          [?]     10e9c:sethi         %hi(0x10c00), %i4
0.   0.          [?]     10ea0:mov           100, %i3
0.   0.          [?]     10ea4:st            %i3, [%i0]
0.   0.          [?]     10ea8:ldd           [%i4 + 640], %f8
```

静态函数

静态函数通常在库内使用，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，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。分析器不能区分连续的剥离静态函数和单一的剥离静态函数，因此会有两个或两个以上的剥离静态函数与其接合的度量一起显示。静态函数的示例可在 jsynprog 演示程序的函数列表中找到，如下所示。

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

在 “PC” 标签中，上述函数使用如下所示的偏移量表示：

```
0.    0.    <static>@0x18780 + 0x00000818
0.    0.    <static>@0x20cc + 0x0000032C
0.    0.    <static>@0xc9f0 + 0x00000060
0.    0.    <static>@0xd1d8 + 0x00000040
0.    0.    <static>@0xe204 + 0x00000170
```

在 “PC” 标签中，在剥离库内调用的函数的替代表示方法为：<*library*.so> -- 未找到函数 + 0x0000F870

包括度量

在注释反汇编代码中，有一些特殊代码行标记从属线程占用的时间以及外联函数占用的时间。

以下是演示程序 omptest 的 < *从属线程的包括度量* > 示例。

```
            3.          subroutine pardo(n,m,a,b,c)
               <Function:pardo_>
0.    0.       [ 3]     1d200:save        %sp, -240, %sp
0.    0.       [ 3]     1d204:ld          [%i0], %i5
0.    0.       [ 3]     1d208:st          %i5, [%fp - 24]
            4.
            5.          real*8 a(n,n), b(n,n), c(n,n)
            6.
            7.          call initarray(n,a,b,c)
            8.
0.    12.679    <inclusive metrics for slave threads>
            9. c$omp  parallel shared(a,b,c,n) private(i,j,k)
0.    3.653    <inclusive metrics for slave threads>
           10. c$omp  do
```

以下所示为一个调用外联函数时的注释反汇编代码示例：

```
           43.          else
           44.          {
           45.                  printf("else reached\n");
0.    2.522    <inclusive metrics for outlined functions>
```

分支目标

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

在不运行实验的情况下查看源代码/反汇编代码

无需运行实验，使用 `er_src` 实用程序就可以查看注释源代码和注释反汇编代码。显示的生成方式与在分析器中的生成方式相同，只是不显示任何度量。`er_src` 命令的语法如下所示：

```
er_src [ -func | -{source,src} item tag | -disasm item tag |
-{cc,scc,dcc} com_spec | -outfile filename | -V ] object
```

object 是可执行文件、共享对象或目标文件 （`.o` 文件）的名称。

item 是用于生成可执行文件或共享对象的函数、源文件或目标文件的名称。 *item* 还可以采用 *functon'file'* 形式指定，在这种情况下 `er_src` 将在该命名文件的源上下文中显示该命名函数的源代码或反汇编代码。

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*。该选项是必需选项，但如果没必要解析函数，则可以忽略该选项。

特殊的项和标记 `all -1` 指示 `er_src` 为该对象中的所有函数生成带注释的源代码或反汇编代码。

注 – 在可执行文件和共享对象上使用 `all -1` 生成的输出可能非常大。

以下几节将介绍 `er_src` 实用程序可以使用的选项。

`-func`

列出给定对象的所有函数。

`-{source,src}` 项标记

显示列出的*项*的注释源代码。

-disasm 项标记

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

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

指定要显示那些编译器注释类。*com-spec* 是用冒号隔开的类列表。如果使用的是 -scc 选项，则对源代码编译器注释应用类列表 *com-spec* ；如果使用的是 -dcc 选项，则对反汇编代码编译器注释应用类列表；如果使用的是 -c 选项，则既对源代码注释应用类列表，也对反汇编代码注释应用类列表。关于这些类的描述，请参见第 113 页的 “控制源代码和反汇编列表的命令”。

注释类可以在默认文件中指定。首先读取系统范围的默认文件 er.rc，然后读取用户 Home 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） 。Home 目录中 .er.rc 文件的默认值覆盖系统的默认值，而当前目录中 .er.rc 文件的默认值覆盖 Home 目录和系统的默认值。这些文件也由分析器和 er_print 实用程序使用，但只有源代码和反汇编代码编译器注释的设置由 er_src 实用程序使用。关于默认文件的描述，请参见第 126 页的“设置默认值的命令”。默认文件中除 scc 和 dcc 之外的命令均被 er_src 实用程序忽略。

-outfile *filename*

打开显示列表输出的文件 *filename*。默认情况下，如果文件名为破折号 (-)，则列表输出会写入 stdout。

-V

打印当前发行版本。

第9章

操作实验

本章介绍了可以与收集器和性能分析器一起使用的实用程序。

本章涵盖了以下主题：

- 操作实验
- 其他实用程序

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。对于早于 Forte Developer 7 （Solaris 的 Sun™ ONE Studio 7 企业版）的发行版本的实验，则不能这样做。有三个实用程序可用来复制、移动和删除实验，功能类似于 UNIX 命令。这三个实用程序是 er_cp(1)、 er_mv(1) 和 er_rm(1)，将在下文中描述。

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。

使用 er_cp 实用程序复制实验

存在两种形式的 er_cp 命令：

```
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 实用程序退出并显示错误消息。-V 选项打印 er_cp 实用程序的版本。该命令不能复制早于 Forte Developer 7 的软件发行版本创建的实验。

使用 er_mv 实用程序移动实验

存在两种形式的 er_mv 命令：

```
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 实用程序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 实用程序退出并显示错误消息。 -V 选项打印 er_mv 实用程序的版本。该命令不能移动早于 Forte Developer 7 的软件发行版本创建的实验。

使用 er_rm 实用程序删除实验

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。

er_rm 命令的语法如下所示：

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

无论是否找到实验，-f 选项都会禁止错误消息并确保成功完成。-V 选项打印 er_rm 实用程序的版本。该命令删除早于 Forte Developer 7 的软件发行版本创建的实验。

其他实用程序

其他一些实用程序不必在正常情况下使用。在这里说明这些实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 实用程序

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 实用程序时，er_archive 实用程序自动运行。实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 实用程序不能由收集器运行。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 实用程序。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目标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 实用程序时找不到共享对象，或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 实用程序，则归档文件包含警告。无论何时手动（没有 -q 标志）运行 er_archive 实用程序，也都会将警告写入 stderr。

以下几节将介绍 er_archive 实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中或从 er_print 实用程序输出。

-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 实用程序的版本号信息，并退出。

er_export 实用程序

er_export 命令的语法如下所示。

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

er_export 实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实用程序；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

本附录中所介绍的工具是程序计时和获取分析性能数据的标准实用程序，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` 是随 Sun Studio 软件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，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，请参见第 3 章和联机帮助。

表 A-1 介绍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。关于各个语言的详细信息，请参见每个工具中的章节。

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 196 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 7 操作系统和 Solaris 8 操作系统而言，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他模块调用，哪些模块调用其他模块的列表，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 Web 站点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 7 操作系统和 Solaris 8 操作系统而言，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 实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 实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。编译器为每个目标文件创建一个具有 .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 （其他编译器）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，请参见 tcov(1) 手册页。tcov 的默认输出为一系列文件，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 实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 （其他编译器）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，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（其他编译器）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，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，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，所以其他文件也可正确更新。

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，以上处理将失败。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以下错误消息：

- 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，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代码：

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。默认情况下，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默认值（请参见第 205 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 增强版的默认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，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，显示以下消息。

```
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默认值控制，您可以通过环境变量修改这些默认值。

注 – tcov 所使用的默认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详细信息，请参见相关编译器文档。

默认分析存储桶以具有 .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，则默认行为是在 /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。

您可以设置以下环境变量来修改默认值：

- 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，116
analyzer 命令
  版本 (-V) 选项，83
  帮助 (-h) 选项，83
  JVM 路径 (-j) 选项，82
  JVM 选项 (-j) 选项，82
  冗余 (-v) 选项，82
  字体大小 (-f) 选项，82
API, 收集器，49
arc, 调用图，定义，196

B

版本信息
  er_cp 实用程序，189
  er_mv 实用程序，190
  er_print 实用程序，129
  er_rm 实用程序，190
  er_src 实用程序，188
  为 collect 命令，68
包括度量
  从属线程，186
  递归效果，43
  定义，40
  PLT 指令，142
  如何计算，141
  使用，41
  图示，42
  外联函数，186
包装器函数，158
编译
  程序分析优化影响，46
  代码行分析，45
  带注释的 “源码” 和 “反汇编” 中的源代码，45
  调试符号信息格式，45
  gprof, 196
  Java 编程语言，46
  库的静态链接，46
  prof, 194
  数据收集的链接，46
  tcov, 198
  tcov 增强版，204
  影响数据收集静态链接，46
编译器，访问，19
编译器生成的主体函数
  名称，181
  由性能分析器显示，160, 181
编译器优化
  并行化，173
  内联，173
编译器注释，86
  并行化，173
  定义的类，114
  公用子表达式排除，171
  过滤显示的类型，171
  克隆的函数，185
  描述，170
  内联函数，173

为 er_print 实用程序中列出的注释反汇编代码选择，115
为 er_print 实用程序中列出的注释源代码选择，114
循环优化，172
在 er_src 实用程序中过滤，188
标签
设置默认可见集合，在 er_print 实用程序中，127
选择以显示，93
并行执行
指令，173

C

C 编译器选项，xhwcprof, 87
collect 命令
版本 (-V) 选项，68
存档 (-A) 选项，67
dry run (-n) 选项，68
堆跟踪 (-H) 选项，63
后跟后续进程 (-F) 选项，64
Java 版本 (-j) 选项，65
记录样例点 (-l) 选项，66
基于时钟的分析 (-p) 选项，61
MPI 跟踪 (-m) 选项，63
冗余 (-v) 选项，68
实验控制选项，64
实验名称 (-o) 选项，68
实验目录 (-d) 选项，67
实验组 (-g) 选项，67
使用 ppgsz 命令，80
输出选项，67
数据收集选项，61, 82
数据限制 (-L) 选项，67
停止 exec (-x) 选项后的目标，66
同步等待跟踪 (-s) 选项，63
选项列表，61
硬件计数器溢出分析 (-h) 选项，62
用其收集数据，61
语法，61
杂项选项，68
暂停和恢复数据记录 (-y) 选项，66
周期性抽样 (-S) 选项，64
自述文件显示 (-R) 选项，68
collectorAPI.h, 51
CPU
选定的列表，在 er_print 实用程序中，119
在 er_print 实用程序中选择，121
CPU 过滤，95
“查找”工具，94
程序计数器 (PC), 定义，141
程序结构，将调用栈地址映射到，157
程序链接表 (PLT), 142
程序执行
单线程，141
共享对象和函数调用，142
所述的调用栈，141
尾调用优化，143
陷阱，143
显式多线程，144
信号处理，142
抽样间隔
定义，39
通过 collect 命令设置，64
在 dbx 设置，72
磁盘空间，实验估计，59
从 er_print 实用程序输出累积统计信息，129
存储需求，实验估计，59

D

data_layout 命令，116
data_objects 命令，116
data_single 命令，116
data_sort 命令，117
DataLayout 标签，88
DataObjects 标签，87
dbx
在 MPI 下收集数据，79
在其中运行收集器，69
dbx collector 子命令
archive, 73
dbxsample, 72
disable, 72

enable, 72
enable_once （已废弃）, 74
hwprofile, 70
limit, 73
pause, 73
profile, 69
quit （已废弃）, 74
resume, 73
sample, 72
sample record, 73
show, 74
status, 74
store, 74
store filename （已废弃）, 74
synctrace, 71
DTrace 驱动程序
描述, 99
设置访问, 99
单线程程序执行, 141
等待时间, 请参见同步等待时间
递归函数调用
度量分配到, 43
地址空间, 文本和数据区域, 157
调用方与被调用方度量
属性, 定义, 41
在 er_print 实用程序中的排序顺序, 112
在 er_print 实用程序中输出, 110
在 er_print 实用程序中为单个函数输出, 111
在 er_print 实用程序中显示其列表, 123
在 er_print 实用程序中选择, 111
“调用方与被调用方”标签, 85, 94
调用栈, 89
不完整的解除, 156
定义, 141
将地址映射到程序结构, 157
解除, 141
“时间线”标签中的默认对齐和深度, 128
尾调用优化的影响, 143
在 “时间线”标签中, 89
在 “事件”标签中, 90
动态编译函数
定义, 161, 182
收集器 API, 53
度量
定时, 33
定义, 31
独占, 请参见独占度量
堆跟踪, 37
非独占, 请参见非独占度量
非独占和独占, 84, 85
关联的影响, 138
函数列表, 请参见函数列表度量
基于时钟的分析, 33, 137
MPI 跟踪, 38
默认, 85
内存分配, 37
时间精度, 85
属性, 85
属性, 请参见属性度量
同步等待跟踪, 37
硬件计数器, 归属到指令, 180
阈值, 87
阈值, 设置, 86
源代码行的解释, 174
指令的解释, 177
独占度量
定义, 40
PLT 指令, 142
如何计算, 141
使用, 41
图示, 42
堆跟踪
度量, 37
默认度量, 85
使用 collect 命令收集数据, 63
预装收集器库, 76
在 dbx 中收集数据, 71
堆栈帧
从陷阱处理程序, 143
定义, 142
尾调用优化中的重用, 143
多线程
显式, 144
多线程的应用程序
将收集器附加到, 74

E

.er.rc 文件 , 85, 96, 188

er_archive 实用程序 , 191

er_cp 实用程序 , 189

er_export 实用程序 , 192

er_kernel 实用程序 , 99

er_mv 实用程序 , 190

er_print 命令
- addpath, 116
- allocs, 112
- appendtfile, 124
- callers-callees, 110
- cmetric_list, 123
- cmetrics, 111
- cpu_list, 119
- cpu_select, 121
- csingle, 111
- csort, 112
- data_layout, 116
- data_metric_list, 123
- data_metrics, 117
- data_objects, 116
- data_single, 116
- data_sort, 117
- dcc, 115
- disasm, 114
- dmetrics, 127
- dsort, 127
- en_desc, 127
- exp_list, 118
- fsingle, 110
- fsummary, 109
- functions, 108
- header, 125
- help, 129
- leaks, 112
- limit, 124
- lines, 113
- lsummary, 113
- lwp_list, 119
- lwp_select, 121
- mapfile, 129
- metric_list, 123
- metrics, 108
- name, 124
- object_list, 122
- object_select, 123
- objects, 125
- overview, 126
- outfile, 124
- pcs, 113
- procstats, 129
- psummary, 113
- quit, 129
- sample_list, 119
- sample_select, 121
- scc, 114
- script, 129
- setpath, 116
- sort, 109
- source, 113
- src, 113
- statistics, 126
- sthresh, 115
- tabs, 127
- thread_list, 119
- thread_select, 121
- tldata, 128
- tlmode, 128
- version, 129
- viewmode, 125

er_print 实用程序
- 度量关键字 , 107
- 度量列表 , 105
- 命令 , 请参见 er_print 命令
- 命令行选项 , 104
- 目的 , 103
- 语法 , 104

er_rm 实用程序 , 190

er_src 实用程序 , 187

F

fast 陷阱 , 143

Fortran
- 收集器 API, 49
- 替代的入口点 , 159
- 子例程 , 157

Fortran 函数中替代的入口点 , 159

反汇编代码 , 带注释
- 包括度量 , 186
- 度量格式 , 175

分支目标，187
HotSpot 编译的指令，183
Java 本机方法，184
解释，177
克隆的函数，159, 185
可执行文件的位置，59
描述，176
设置在 er_print 实用程序中的选项，115
硬件计数器度量属性，180
用 er_src 实用程序查看，187
在 er_print 实用程序中设置突出显示的阈值，115
在 er_print 实用程序中输出，114
指令执行依存，177
“反汇编”标签，87
方法，请参见函数
非唯一函数名称，158
分析，定义，32
分析存储桶，tcov 增强版，204, 205
分析共享库，创建
tcov, 201
tcov 增强版，204
分析间隔
定义，33
实验大小，影响，60
使用 dbx collector 命令设置，70
通过 collect 命令设置，62, 70
值的限制，54
分析器，请参见性能分析器
分析文件包
大小，59
基于时钟的数据，136
MPI 跟踪数据，140
同步等待跟踪数据，139
硬件计数器溢出数据，139
分支 PC, 定义，141
分支目标，187
符号表，负载对象，157
负载对象
定义，157
符号表，157
函数的地址，157
内容，157
写入布局，116
选定的列表，在 er_print 实用程序中，122
在 er_print 实用程序中输出列表，125
在 er_print 实用程序中选择，123
复制实验，189

G

gprof
使用，196
输出，解释，196
限制，196
小结，193
概述数据，在 er_print 实用程序中输出，126
高度量值
在注释的反汇编代码中，115
在注释的源代码中，115
共享对象，之间的函数调用，142
公用子表达式排除，171
关键字，度量，er_print 实用程序，107
关联，对度量的影响，138
过滤 CPU, 95
过滤 LWP, 95
过滤实验，95
“过滤数据”对话框，94
过滤线程，95
过滤样例，95

H

函数
@plt, 142
包装器，158
地址变化，157
定义，157
动态编译，53, 161, 182
非唯一，名称，158
负载对象内的地址，157
静态，带有重复的名称，158
静态，在剥离共享库中，158, 185
克隆，159, 184
MPI, 跟踪，38

内联，159
全局，158
收集器 API, 49, 53
<Total>, 163
替代的入口点 (Fortran), 159
外联，160, 180
<Unknow>, 161
系统库，由收集器插入，48
有别名的，158
函数 PC, 聚集，87, 94
函数调用
递归，度量分配到，43
共享对象之间，142
在单线程程序中，141
函数列表
编译器生成的主体函数，181
排序顺序，在 er_print 实用程序中指定，109
在 er_print 实用程序中输出，108
函数列表度量
设置 .er.rc 文件中的默认排序顺序，127
选择 .er.rc 文件中的默认值，127
在 er_print 实用程序中显示其列表，123
在 er_print 实用程序中选择，108
函数名称，C++, 在 er_print 实用程序中选择长名形式或短名形式，124
函数重组，96
“函数”标签，84, 94
后续进程
后跟收集器，56
其数据收集的局限，56
实验名称，58
实验位置，58
为所选后续进程收集数据，74
为所有后跟内容收集数据，64
后续实验
加载，81
设置用于读取的模式，在 er_print 实用程序中，127
环境变量
JAVA_PATH, 56
JDK_HOME, 56
LD_LIBRARY_PATH, 76
LD_PRELOAD, 76
PATH, 56
SUN_PROFDATA, 205
SUN_PROFDATA_DIR, 206
TCOVDIR, 199, 206
恢复数据收集
从程序，52
为 collect 命令，66
在 dbx, 73
汇总度量
为单个函数，在 er_print 实用程序中输出，110
为所有函数，在 er_print 实用程序中输出，109

J

Java
动态编译的方法，53, 161
分析限制，56
设置 er_print 显示输出，125
显示器，36
Java 虚拟机路径，analyzer 命令选项，82
JAVA_PATH 环境变量，56
--jdkhome analyzer 命令选项，82
JJDK_HOME 环境变量，56
JVM 版本，57
积极回溯，87
基于时钟的分析
定义，33
度量，33, 137
度量的准确性，138
分析包中的数据，136
gethrtime 和 gethrvtime 的比较，138
间隔，请参见分析间隔
开销引起的失真，138
默认度量，85
使用 collect 命令收集数据，61
在 dbx 中收集数据，69
间隔，抽样，请参见抽样间隔
间隔，分析，请参见分析间隔
将负载对象归档到实验，67, 73
将路径附加到文件，116
将收集器附加到正在运行的进程，74
解除调用栈，141
进程地址空间文本和数据区域，157

静态函数
在剥离共享库中，158, 185
重复的名称，158
静态链接，影响数据收集，46

K

克隆的函数，159, 184, 185
库
剥离的共享，和静态函数，158, 185
collectorAPI.h, 51
插入，48
静态链接，46
libaio.so, 49
libcollector.so, 49, 76
libcpc.so, 48, 55
libthread.so, 48, 144
MPI, 48, 77
system, 48

L

LD_LIBRARY_PATH 环境变量，76
LD_PRELOAD 环境变量，76
libaio.so, 与数据收集交互，49
libcollector.h, 50
做为到收集器的 C 和 C++ 接口，50
做为收集器 Java 编程语言接口的一部分，51
libcollector.so 共享库
预装，76
在程序中使用，49
libcpc.so, 使用，55
libfcollector.h, 50
LWP
过滤，95
选定的列表，在 er_print 实用程序中，119
由线程库创建，144
在 er_print 实用程序中选择，121

M

MANPATH 环境变量，设置，20
mapfile
生成，96
用 er_print 实用程序生成，129
MPI 程序
附加到，76
实验存储问题，77
实验名称，58, 77, 78
使用 collect 命令收集数据，78
收集数据，77
通过 dbx 收集数据，79
MPI 跟踪
度量，38
度量的解释，140
分析包中的数据，140
跟踪的函数，38
使用 collect 命令收集数据，63
预装收集器库，76
在 dbx 中收集数据，71
MPI 实验
存储问题，77
默认名称，58
移动，78
默认度量，85
默认值，在默认文件中设置，126

N

nfs, 57
内存分配，37
对数据收集的影响，47
和泄漏，89
内存泄漏，定义，37
内核分析
分析特定的进程或内核线程，102
设置系统，99
一起分析内核和负载，101
在有负载时分析，101
内核实验
数据类型，99
字段标签含义，102
内核时钟分析，100
内核性能分析数据，分析，102

内联函数 , 159

O

OpenMP
度量 , 154
分析数据 , 机器表示 , 154
分析数据的用户模式显示 , 148
设置 er_print 显示输出 , 125
用户模式调用栈 , 149
执行概述 , 147
OpenMP 并行性 , 173
OpenMP 应用程序中的用户模式调用栈 , 149

P

PATH 环境变量 , 56
设置 , 19
PC
从 PLT, 142
定义 , 141
er_print 实用程序中的排序列表 , 113
PC 标签 , 87, 94
@plt 函数 , 142
PLT （程序链接表） , 142
ppgsz 命令 , 80
prof
使用 , 194
输出 , 195
限制 , 195
小结 , 193
排序顺序
调用方与被调用方度量 , 在 er_print 实用程序中 , 112
函数列表 , 在 er_print 实用程序中指定 , 109

R

人工函数 , 在用户模式调用栈中 , 148
入口点 , 替代的 , 在 Fortran 函数中 , 159

S

<Scalar> 数据对象描述符 , 164
setpath 命令 , 116
setuid, 使用 , 49
Shell 提示符 , 18
SUN_PROFDATA, 环境变量 , 205
SUN_PROFDATA_DIR, 环境变量 , 206
删除实验或实验组 , 190
事件
“时间线”标签中的默认显示类型 , 128
在 “时间线”标签中显示的 , 89
事件标记 , 89
时间度量 , 精度 , 85
“时间线”标签 , 89, 91
“时间线”菜单 , 84
“事件”标签 , 89, 90
实验
从程序中终止 , 53
存储位置 , 67, 74
存储需求 , 估计 , 59
打开 , 81
定义 , 57
多个 , 81
er_print 实用程序中的标题信息 , 125
附加当前路径 , 116
复制 , 189
后续 , 加载 , 81
将负载对象归档到 , 67, 73
另请参见实验目录
MPI 存储问题 , 77
命名 , 58
默认名称 , 58
删除 , 190
设置路径来查找文件 , 116
数据聚集 , 81
添加 , 81
为 Java 和 OpenMP 设置模式 , 125
位置 , 58
限制其大小 , 67, 73
移动 , 59, 190
移动 MPI, 78
预览 , 81

在 er_print 实用程序中列出，118
组，58
实验名称
实验，后续
设置读取的模式，在 er_print 实用程序中，127
实验过滤，95
实验名称
MPI 默认，58, 78
MPI, 使用 MPI_comm_rank 和脚本，79
默认，58
约束，58
在 dbx 中指定，74
实验目录
默认，58
使用 collect 命令指定，67
在 dbx 中指定，74
实验组
创建，81
定义，58
多个，81
名称约束，58
默认名称，58
删除，190
使用 collect 命令指定名称，67
添加，81
预览，81
在 dbx 中指定名称，74
"实验"标签，90
手册页，访问，19
收集器
API, 在程序中使用，49, 51
定义，28, 32
附加到正在运行的进程，74
使用 collect 命令运行，61
在 dbx 中禁止，72
在 dbx 中启用，72
在 dbx 中运行，69
收集实验
预览命令，96
输出当前路径，116
输出文件
关闭，在 er_print 实用程序中，124
关闭并打开新的，在 er_print 实用程序中，124
数据对象
布局，88
定义，163
范围，164
设置排序度量，117
在硬件计数器溢出实验中，116
<Scalar> 描述符，164
<Total> 描述符，164
数据类型，32
堆跟踪，37
基于时钟的分析，33
MPI 跟踪，38
默认，在"时间线"标签中，128
同步等待跟踪，36
硬件计数器溢出分析，34
数据派生的度量
在 er_print 实用程序中设置，117
在 er_print 实用程序中显示其列表，123
数据收集
程序控制，49
从 MPI 程序，77
从程序禁止，53
从程序中恢复，52
从程序中控制，49
动态内存分配影响，47
段失败，47
链接，46
MPI 程序，使用 collect 命令，78
MPI 程序，使用 dbx, 79
使用 collect 命令，61
使用 dbx, 69
速度，60
为 collect 命令恢复，66
为 collect 命令暂停，66
在 dbx 中恢复，73
在 dbx 中禁止，72
在 dbx 中启用，72
在 dbx 中暂停，73
在程序中暂停，52
准备程序，47
数据收集期间段失败，47
输入文件
到 er_print 实用程序，129
在 er_print 实用程序中终止，129

属性度量
递归效果，43
定义，40
使用，41
图示，42
锁定文件管理
tcov, 202
tcov 增强版，205
索引行，168
er_print 实用程序中的，113, 114
在 “反汇编”标签中，87, 176
在 “源码”标签中，86, 168, 176
索引行，特殊
编译器生成的主体函数，182
HotSpot 编译的指令，183
Java 本机方法，184
没有行号的指令，183
外联函数，180

T

tcov
报告的错误，202
带注释的源代码，200
分析共享库，创建，201
使用，198
输出，解释，200
锁定文件管理，202
为其编译程序，198
限制，198
小结，193
tcov 增强版
分析存储桶，204, 205
分析共享库，创建，204
使用，203
锁定文件管理，205
为其编译程序，204
优点，203
TCOVDIR 环境变量，199, 206
TLB （旁路转换缓冲）缺少，143, 180
<Total> 数据对象描述符，164
<Total> 函数
与执行统计比较时间，138
描述，163
替代源上下文，113
同步等待跟踪
等待时间，36, 139
定义，36
度量，37
分析包中的数据，139
使用 collect 命令收集数据，63
阈值，请参见阈值、同步等待跟踪
预装收集器库，76
在 dbx 中收集数据，71
同步等待时间
带有无界线程，139
定义，36, 139
度量，定义，37
同步延迟跟踪
默认度量，85
同步延迟事件
定义，36
定义度量，37
分析包中的数据，139
“统计”标签，90
“图例”标签，89, 91

U

<Unknown> 函数
调用方与被调用方，161
PC 的映射，161

V

viewmode 命令，125

W

外联函数，160, 180
网络磁盘，57
尾调用优化，143
为实验命名，58
微态，91

对度量的贡献，137
切换，143
文档，访问，21 - 23
文档索引，21
文件路径，116

X

-xdebugformat, 设置调试符号信息格式，45
xhwcprof C 编译器选项，87
线程
创建，144
调度，144
工作线程，144
库，48, 144
system, 139
选定的列表，在 er_print 实用程序中，119
有界和无界，144
在 er_print 实用程序中选择，121
线程过滤，95
陷阱，143
"显示 / 隐藏函数"对话框，94
显式多线程，144
限制
分析间隔值，54
后续进程数据收集，56
Java 分析，56
实验名称，58
实验组名称，58
tcov, 198
限制，参见局限
限制实验大小，67, 73
泄漏，内存，定义，37
"泄漏列表"标签，89
"泄漏"标签，91
信号
对处理程序的调用，142
分析，从 dbx 传送到 collect 命令，66
文件配置，49
用于使用 collect 命令的手动抽样，66
用于使用 collect 命令的暂停和恢复，66
信号处理程序
用户程序，49
由收集器安装，49, 142
性能度量，请参见度量
性能分析器
"帮助"菜单，83
"查找"工具，94
DataLayout 标签，88
DataObjects 标签，87
"调用方与被调用方"标签，85, 94
定义，28, 81
"反汇编"标签，87
"过滤数据"对话框，94
"函数"标签，84, 94
记录实验，82
命令行选项，82
默认，96
PC 标签，87, 94
启动，81
"时间线"标签，89, 91
"时间线"菜单，83, 84
"事件"标签，89, 90
"视图"菜单，83
"实验"标签，90
"统计"标签，90
"图例"标签，89, 91
"文件"菜单，83
显示 / 隐藏函数，94
"泄漏列表"标签，89
"泄漏"标签，91
"行"标签，87, 94
"源码"标签，86
"摘要"标签，87, 90
性能数据，转换为度量，31
"行"标签，87, 94
选项，命令行，er_print 实用程序，104
"选择标签"对话框，88, 93
循环优化，172

Y

样本收集器，请参见收集器
样例
从程序中记录，52

定义，40
记录的条件，39
间隔，请参见抽样间隔
使用 collect 命令进行周期记录，64
通过 collect 手动记录，66
选定的列表，在 er_print 实用程序中，119
样例包中包含的信息，39
在 dbx 停止进程时进行记录，72
在 dbx 中进行周期记录，72
在 dbx 中手动记录，73
在 er_print 实用程序中选择，121
样例点，在 “时间线”标签中显示的，89
样例过滤，95
异步 I/O 库，与数据收集交互，49
溢出值，硬件计数器，请参见硬件计数器溢出值
移动实验，59, 190
易读文档，22
已知硬件计数器，35
印刷约定，16
硬件计数器
获取其列表，61, 70
计数器名称，62
使用 collect 命令选择，62
使用 dbx collector 命令选择，70
数据对象和度量，116
所述的列表，35
溢出值，34
硬件计数器度量，在 DataObjects 标签中显示的，88
硬件计数器库，libcpc.so, 55
硬件计数器列表
使用 collect 命令获取，61
使用 dbx collector 命令获取，70
已知计数器，35
原始计数器，36
字段的描述，35
硬件计数器属性选项，62
硬件计数器溢出分析
定义，34
分析包中的数据，139
默认度量，85
使用 collect 命令收集数据，62
通过 dbx 收集数据，70
硬件计数器溢出值
定义，34
过小或过大的结果，139
实验大小，影响，60
通过 collect 设置，63
在 dbx 设置，70
由 tcov 报告的错误，202
有别名的函数，158
优化
程序分析影响，46
公用子表达式排除，171
尾调用，143
由收集器插入系统库函数，48
语法
er_archive 实用程序，191
er_export 实用程序，192
er_print 实用程序，104
er_src 实用程序，187
阈值，同步等待跟踪
定义，36
对收集开销的影响，139
通过 collect 命令设置，63, 71
通过 dbx collector 设置，71
校准，36
阈值，突出显示
在注释反汇编代码中，er_print 实用程序，115
在注释源代码中，er_print 实用程序，115
预装 libcollector.so, 76
源代码，编译器注释，86
源代码，带注释
编译器生成的主体函数，182
编译器注释，170
从源代码中辨别注释，168
度量格式，175
解释，174
克隆的函数，159, 185
没有行号的指令，183
描述，167, 174
设置 er_print 实用程序中的译器注释类，114
设置在 er_print 实用程序中的出显示的阈值，115
索引行，168
tcov, 200

外联函数，180
用 er_src 实用程序查看，187
源代码文件的位置，59
在 er_print 实用程序中输出，113
在性能分析器中查看，167
中间文件的使用，156
源代码行，er_print 实用程序中的排序列表，113
“源码”标签，86
原始硬件计数器，35, 36

Z

在 er_print 实用程序中设置读取后续实验的模式，127
在 er_print 实用程序中限制输出，124
暂停数据收集
从程序，52
为 collect 命令，66
在 dbx, 73
“摘要”标签，87, 90
帧，堆栈，请参见堆栈帧
指令发布
分组，对注释反汇编的影响，177
延迟，179
执行统计
与 <Total> 函数时间的比较，138
在 er_print 实用程序中输出，126
中间文件，注释源代码列表的使用，156
注释反汇编代码，请参见反汇编代码，注释
注释源代码，请参见源代码，注释
主体函数，编译器生成
名称，181
由性能分析器显示，160, 181
子例程，请参见函数