Solaris 动态跟踪指南

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

文件号码 819–6959–10
2006 年 7 月

060714@14986

目录

图

表

示例

前言

DTrace 是一个用于 Solaris™ 操作系统的全面的动态跟踪框架。DTrace 提供一个强大的基础结构，使管理员、开发者和服务人员能简明地回答有关操作系统和用户程序行为的任意问题。《Solaris 动态跟踪指南》介绍如何使用 DTrace 观察、调试和调优系统行为。本书还包含一套完整的关于捆绑的 DTrace 观察工具和 D 编程语言的参考资料。

注 – 此 Solaris 发行版支持使用以下 SPARC® 和 x86 系列处理器体系结构的系统：UltraSPARC®、SPARC64、AMD64、Pentium 和 Xeon EM64T。支持的系统可以在 http://www.sun.com/bigadmin/hcl/ (http://www.sun.com/bigadmin/hcl/) 上的《Solaris 10 Hardware Compatibility List》中找到。本文档列举了在不同类型的平台上进行实现时的所有差别。

在本文档中，术语 "x86" 指使用与 AMD64 或 Intel Xeon/Pentium 产品系列兼容的处理器生产的 64 位和 32 位系统。若想了解本发行版支持哪些系统，请参见《Solaris 10 Hardware Compatibility List》。

目标读者

如果您想了解您的系统的行为，那么 DTrace 正是您所需要的工具。DTrace 是一个内置于 Solaris 中的全面的动态跟踪工具。DTrace 工具可用于检查用户程序的行为。DTrace 工具还可用于检查操作系统的行为。DTrace 可由系统管理员或应用程序开发者使用，它适用于实时生产系统。DTrace 允许您查看系统，以便了解其工作方式、在软件的多个层之间跟踪性能问题或找出导致异常行为的原因。如您所见，可以使用 DTrace 来创建自己的自定义程序，以便动态地检测系统，并对可用 DTrace D 编程语言阐明的任意问题作出快速简明的回答。

DTrace 允许所有 Solaris 用户执行以下操作：

- 动态地启用和管理数以千计的探测器
- 动态地将逻辑谓词和操作与探测器相关联
- 动态地管理跟踪缓冲区和缓冲区策略
- 显示和检查来自实时系统或崩溃转储的跟踪数据

DTrace 允许 Solaris 开发者和管理员执行以下操作：

- 实现使用 DTrace 工具的自定义脚本
- 实现使用 DTrace 检索跟踪数据的分层工具

本指南将讲授使用 DTrace 时需要了解的所有知识。了解编程语言（如 C）或脚本语言（如 awk(1) 或 perl(1)）的基本知识，有助于更快地学习 DTrace 和 D 编程语言，但您并不需要精通其中的任何领域。如果您以前从未使用任何语言编写过程序或脚本，第 23 页中的 “相关信息”中提供了一些其他文档，或许对您有所帮助。

本书的结构

第 1 章提供整个 DTrace 工具的简要说明并向读者介绍了 D 编程语言。第 2 章、第 3 章和第 4 章将更加详细地讨论 D 的基本知识，并说明了 D 程序如何转换为动态检测过程。所有读者都应首先阅读前几章的内容。

第 5 章、第 6 章、第 7 章和第 8 章讨论 D 语言的其他功能，其中的大部分内容已为 C、C++ 和 Java™ 程序员所熟悉。如果读者不熟悉上述语言中的任何一种语言，则应先阅读这几章，有经验的程序员可以直接学习后面的内容。

第 9 章和第 10 章讨论 DTrace 用于聚合数据的强大的元语功能，以及可用于生成跟踪实验脚本的内置操作集。所有读者都应认真阅读这几章。

第 11 章介绍 DTrace 缓冲数据的策略及如何配置这些策略。读者应在熟悉如何构造和运行 D 程序后阅读本章。

第 12 章介绍 D 输出格式化操作和用于格式化跟踪数据的缺省策略。熟悉 C `printf()` 函数的读者快速浏览一下本章即可。不了解 `printf()` 的读者应仔细阅读本章。

第 13 章讨论 DTrace 工具以推理方式向跟踪缓冲区提交数据。有时，必须先跟踪数据才能了解数据与目前的问题是否相关，需要在这种情况下使用 DTrace 的用户应阅读本章。

第 14 章提供 `dtrace` 命令行实用程序的完整参考（类似于相应的联机手册页）。用户可能会在本书中其他位置出现各种命令行选项时参考本章。接下来第 15 章讨论如何使用 `dtrace` 实用程序构造可执行 D 脚本并处理其命令行参数，第 16 章介绍可在命令行或在 D 程序（自身）内部调优的选项。

从第 17 章到第 32 章的这些章节讨论可用于检测 Solaris 系统各个方面的各种 DTrace 提供器。所有读者都应浏览这些章节，熟悉各种提供器，并在需要时再仔细阅读特定的章节。

第 33 章讨论使用 DTrace 检测用户进程的示例。第 34 章介绍应用程序程序员如何向用户应用程序添加自定义的 DTrace 提供器和探测器。希望使用 DTrace 查明用户进程行为的用户程序开发者或管理员应阅读这几章。

第 35 章和其余的章节讨论高级主题，如 DTrace 的安全性、版本管理和稳定性属性，及如何使用 DTrace 执行引导时和事后跟踪。这几章主要针对高级 DTrace 用户。

相关信息

建议您阅读以下与使用 DTrace 执行的任务有关的书籍和文章：

- 由 Kernighan, Brian W. 和 Ritchie, Dennis M. 合著的《The C Programming Language》。Prentice Hall 出版，1988。ISBN 0–13–110370–9
- 由 Vahalia, Uresh 编著的《UNIX Internals: The New Frontiers》。Prentice Hall 出版，1996。ISBN 0-13-101908-2
- 由 Mauro, Jim 和 McDougall, Richard 合著的《Solaris Internals: Core Kernel Components》。Sun Microsystems Press 出版，2001。ISBN 0-13-022496-0

您可以在 Web 站点（网址为 `http://www.sun.com/bigadmin/content/dtrace/`）上与其他 DTrace 社区人员共享您的 DTrace 经验和脚本。

联机访问 Sun 文档

可以通过 docs.sun.com℠ Web 站点联机访问 Sun 技术文档。您可以浏览 docs.sun.com 文档库或搜索某个特定的书名或主题。URL 为 `http://docs.sun.com`。

订购 Sun 文档

Sun Microsystems 提供了一些印刷的产品文档。有关文档列表以及订购方法，请参见 http://docs.sun.com 上的“购买印刷的文档”。

印刷约定

下表说明了本书中使用的印刷约定。

表 P–1 印刷约定

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

表 P–1 印刷约定 （续）

字体或符号	含义	示例
AaBbCc123	要使用实名或值替换的命令行占位符	要删除文件，请键入 **rm** *filename*。
AaBbCc123	保留未译的新词或术语以及要强调的词	这些称为 *class* 选项。
新词术语强调	新词或术语以及要强调的词	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。

命令中的 shell 提示符示例

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

表 P–2 Shell 提示符

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

◆ ◆ ◆ 第 1 章

1 简介

欢迎使用 Solaris 操作系统的动态跟踪工具！ 如果您想了解您的系统的行为，那么 DTrace 正是您所需要的工具。 DTrace 是一个内置于 Solaris 中的全面的动态跟踪工具。管理员和开发者可以使用该工具，在实时生产系统上检查用户程序及操作系统本身的行为。 DTrace 允许您查看系统，以便了解其工作方式、在软件的多个层之间跟踪性能问题或找出导致异常行为的原因。 如您所见，可以使用 DTrace 来创建自己的自定义程序，以便动态地检测系统，并对可用 DTrace D 编程语言阐明的任意问题作出快速简明的回答。 本章第一部分对 DTrace 进行了快速介绍，并说明如何编写第一个 D 程序。 本章其余部分则介绍了采用 D 语言编程的全套规则，以及用于对系统进行深入分析的技巧及方法。 您可以在 Web 站点（网址为 http://www.sun.com/bigadmin/content/dtrace/）上与其他 DTrace 社区人员共享您的 DTrace 经验和脚本。 本指南中提供的所有示例脚本均可在 Solaris 系统的 /usr/demo/dtrace 目录中找到。

入门

使用 DTrace，可以动态修改操作系统内核和用户进程，以记录您在所关注的位置（称为探测器）指定的其他数据，从而帮助您了解软件系统。 探测器是一个位置或活动，DTrace 可以将请求绑定到该位置或活动中，以便执行一组操作，如记录栈跟踪、时间标记或函数参数。 探测器类似于分散在整个 Solaris 系统中受关注位置的可编程传感器。 如果要确定系统的状态，可使用 DTrace 对相应的传感器进行编程，以便记录您关注的信息。 然后，当每个探测器触发时，DTrace 便会从探测器中收集数据并向您报告。 如果未为探测器指定任何操作，DTrace 将仅记录探测器的每次触发。

在 DTrace 中，每个探测器都有两个名称：一个唯一的整数 ID 以及一个人工可读的字符串名称。 接下来，我们将通过使用名为 BEGIN 的探测器生成一些非常简单的请求来开始学习 DTrace，该探测器在每次启动新的跟踪请求时触发一次。 您可以使用 dtrace(1M) 实用程序的 -n 选项，通过探测器的字符串名称来启用探测器。键入以下命令：

```
# dtrace -n BEGIN
```

在简短暂停后，DTrace 将会通知您已启用一个探测器，并且会显示一行输出，表明 BEGIN 探测器已被触发。 显示此输出后，dtrace 将保持暂停状态，以等待其他探测器触发。 由于尚未启用任何其他探测器，并且 BEGIN 仅触发一次，因此可在 shell 中按 Ctrl-C 组合键，退出 dtrace 并返回到 shell 提示符：

```
# dtrace -n BEGIN

dtrace: description 'BEGIN' matched 1 probe

CPU     ID                    FUNCTION:NAME

  0      1                      :BEGIN

^C

#
```

该输出表明，名为 BEGIN 的探测器已被触发一次，并且还会列显其名称和整数 ID 1。 请注意，缺省情况下会显示触发此探测器的 CPU 的整数名称。 在此示例中，CPU 列表明触发该探测器时，正在 CPU 0 上执行 dtrace 命令。

您可以使用任意数量的探测器和操作构造 DTrace 请求。 通过在上一示例命令中添加 END 探测器，我们来使用两个探测器创建一个简单的请求。 END 探测器在完成跟踪时触发一次。 请键入以下命令，然后在显示 BEGIN 探测器的输出行后，再次在 shell 中按 Ctrl-C 组合键：

```
# dtrace -n BEGIN -n END

dtrace: description 'BEGIN' matched 1 probe

dtrace: description 'END' matched 1 probe

CPU     ID                    FUNCTION:NAME

  0      1                      :BEGIN

^C

  0      2                        :END

#
```

如您所见，按 Ctrl-C 组合键退出 dtrace 时可触发 END 探测器。dtrace 在退出之前，会报告此探测器的触发情况。

至此，您已大致了解了如何命名和启用探测器，现在便可以尝试编写最简单的 DTrace 版程序 "Hello, World"。 除了可在命令行中构造 DTrace 实验脚本之外，还可使用 D 编程语言在文本文件中进行编写。 在文本编辑器中，创建一个称作 hello.d 的新文件，然后键入您的第一个 D 程序：

示例 1–1 hello.d: 采用 D 编程语言编写的 "Hello, World"

```
BEGIN

{

    trace("hello, world");

    exit(0);

}
```

保存程序后，可以使用 dtrace -s 选项运行该程序。键入以下命令：

```
# dtrace -s hello.d

dtrace: script 'hello.d' matched 1 probe

CPU     ID                    FUNCTION:NAME

  0        1                      :BEGIN   hello, world

#
```

如您所见，dtrace 显示的输出与前面相同，后跟文本 "hello, world"。 与上例不同的是，您不需要等待，也不需要按 Ctrl-C 组合键。 这些更改是在 hello.d 中为 BEGIN 探测器指定的操作的结果。 为了解所发生的情况，让我们来看看 D 程序的详细结构。

每个 D 程序均由一系列子句组成，每个子句用于描述一个或多个要启用的探测器，以及触发探测器时要执行的一组可选操作。 这些操作以一系列括在大括号 { } 中的语句形式列出，跟在探测器名称后面。 每条语句都以分号 (;) 结尾。 您的第一条语句使用函数 trace() 来指示 DTrace 应在触发 BEGIN 探测器时记录指定的参数（字符串 "hello, world"），然后将其显示出来。 第二条语句使用函数 exit() 来指示 DTrace 应停止跟踪并退出 dtrace 命令。 DTrace 提供了一组类似于 trace() 和 exit() 的有用函数，供您在 D 程序中调用。 要调用函数，请指定函数名称，并后跟用括号括起的参数列表。 有关全套 D 函数的描述，请参阅第 10 章。

如果您熟悉 C 编程语言，则此时您可能已从名称和示例了解到，DTrace 的 D 编程语言与 C 编程语言非常类似。实际上，D 源自一个大型的 C 子集和一组可简化跟踪的特定函数和变量。 在后续章节中，您将会进一步了解这些功能。 如果您以前编写过 C 程序，则可将大部分知识立即运用到 D 中生成跟踪程序。如果您以前从未编写过 C 程序，学习 D 也很容易。经过本章的学习后，您将了解所有语法。 不过，让我们首先从语言规则开始，进一步了解 DTrace 的工作方式，然后再返回来学习如何生成更有趣的 D 程序。

提供器和探测器

在前面的示例中，您学习了如何使用两个简单的、名为 BEGIN 和 END 的探测器。 但是，这些探测器来自何处呢？ DTrace 探测器来自一组称为提供器的内核模块，其中，每个模块都执行一种特定类型的检测过程来创建探测器。 使用 DTrace 时，每个提供器均有机会将可提供的探测器发布到 DTrace 框架。 然后，您可以启用并将跟踪操作绑定到已发布的任何探测器。 要列出系统中的所有可用探测器，请键入以下命令：

```
# dtrace -l

  ID   PROVIDER            MODULE          FUNCTION NAME

   1     dtrace                                     BEGIN

   2     dtrace                                     END

   3     dtrace                                     ERROR

   4   lockstat           genunix       mutex_enter adaptive-acquire

   5   lockstat           genunix       mutex_enter adaptive-block

   6   lockstat           genunix       mutex_enter adaptive-spin

   7   lockstat           genunix       mutex_exit  adaptive-release


   ... many lines of output omitted ...


#
```

显示所有输出可能需要一些时间。 要统计所有探测器的数量，可以键入以下命令：

```
# dtrace -l | wc -l

        30122
```

您可能会注意到您的计算机中的探测器总数有所不同，这是因为探测器的数量因操作平台及所安装的软件而异。 如您所见，有大量的探测器可供您使用，因此，您可以检查系统中以前未曾关注过的的每个位置。 事实上，如您随后将了解到的，即使此输出列表也是不完整的，因为某些提供器可基于跟踪请求动态创建新的探测器，从而导致 DTrace 探测器的实际数量是无限。

现在，我们返回来在终端窗口中查看 **dtrace -l** 的输出。 请注意先前已经提到过的，每个探测器都有两个名称，一个整数 ID 以及一个人工可读的名称。 人工可读的名称包括四个部分，如 dtrace 输出中的各列所示。 探测器名称的四个部分如下：

提供器	发布此探测器的 DTrace 提供器的名称。 提供器名称通常与执行检测过程以启用探测器的 DTrace 内核模块的名称相对应。
模块	此探测器对应于特定的程序位置时，为探测器所在模块的名称。 该名称为内核模块的名称或用户库的名称。
函数	此探测器对应于特定的程序位置时，为探测器所在程序函数的名称。
名称	探测器名称的最后组成部分是一个有助于您了解探测器语义的名称（如 BEGIN 或 END）。

写出完整的人工可读的探测器名称时，请用冒号分隔列出该名称的所有四个部分，如下所示：

提供器:模块:函数:名称

请注意，该列表中某些探测器没有模块和函数，如先前使用的 BEGIN 和 END 探测器。 某些探测器将这两个字段留为空白，这是因为这些探测器与任何特定的受检测程序函数或位置都不对应。 相反，这些探测器是指一个更抽象的概念，如跟踪请求结束。 将模块和函数作为其名称一部分的探测器称为固定探测器，名称中没有模板和函数的探测器则称为不固定探测器。

根据约定，如果没有指定探测器名称的所有字段，则只要探测器中的值与您指定的那部分名称相匹配，DTrace 就会将您的请求与所有此类探测器匹配。 换而言之，在先前使用探测器名称 BEGIN 时，您实际上是指示 DTrace 匹配名称字段为 BEGIN 的任何探测器，而不考虑提供器、模块和函数字段的具体值。 由于恰好只有一个探测器与该描述相符，因此才出现相同的结果。 但是，您现在知道 BEGIN 探测器的实际名称为 dtrace:::BEGIN，这表示该探测器由 DTrace 框架本身提供，并且不固定到任何函数。 因此，如果按以下所示方式编写 hello.d 程序，则也会生成相同结果。

```
dtrace:::BEGIN

{

    trace("hello, world");

    exit(0);

}
```

至此，您已了解探测器的来源和命名方式，接下来我们将进一步了解启用探测器并要求 DTrace 执行某些操作时发生的情况，然后我们会返回到 D 程序旋风之旅。

编译和检测过程

在 Solaris 中编写传统程序时，可使用编译器将程序从源代码转换为可执行的目标代码。使用 `dtrace` 命令时，将调用先前用于编写 `hello.d` 程序的 D 语言的编译器。程序编译完毕后，随即发送到操作系统内核中以供 DTrace 执行。在操作系统内核中，将启用程序中指定的探测器，并且对应的提供器会执行激活探测器所需的任何检测过程。

DTrace 中的所有检测过程都是完全动态的：仅在需要使用时分别启用所需的探测器。对于未激活的探测器，不存在任何检测代码，因此在不使用 DTrace 时，系统的性能丝毫不会降低。一旦完成实验脚本并退出 `dtrace` 命令，便会自动禁用使用的所有探测器，并取消探测器检测过程，从而将系统彻底返回至初始状态。在未激活 DTrace 的系统和未安装 DTrace 软件的系统之间，差异并不明显。

每个探测器的检测过程都在实时运行的操作系统或所选的用户进程上动态地执行。该系统不会以任何方式停顿或暂停，并且仅为启用的探测器添加检测代码。因此，使用 DTrace 所产生的探测影响被准确限制在要求 DTrace 执行的操作上：不会跟踪任何无关的数据，也不会在系统中打开一个大型“跟踪开关”，所有 DTrace 检测过程的设计原则都是尽可能提高效率。通过这些功能，可在生产中使用 DTrace 来实时解决实际问题。

DTrace 框架还提供对任意数量的虚拟客户机的支持。在系统内存容量允许的情况下，您可以同时运行任意数目的 DTrace 实验脚本和命令，并且所有命令都使用同一基本检测过程独立运行。这一功能还可使系统中任意数目的用户同时使用 DTrace：使用 DTrace，开发者、管理员和服务人员可以在同一系统中合作，也可单独处理同一系统中的不同问题，而不会相互干扰。

DTrace D 程序会被编译成一种安全的中间格式，用于在触发探测器时执行，这一点与 C 和 C++ 编写的程序不同，但与 Java™ 编程语言编写的程序类似。当 DTrace 内核软件对程序进行首次检查时，将会验证该中间格式是否安全。DTrace 执行环境还可处理在 D 程序执行期间可能发生的任何运行时错误，包括除数为零、取消引用无效内存等，并将这些错误向您报告。因此，您永远不会构造出可能使 DTrace 意外损坏 Solaris 内核或系统中运行的某个进程的不安全程序。通过这些安全功能，可在生产环境中使用 DTrace，而不用担心系统崩溃或损坏。如果出现编程错误，DTrace 将向您报告该错误并禁用检测过程。然后，您可以纠正相应错误并重试。有关 DTrace 错误报告和调试功能，请参见本书后续章节。

下图说明了 DTrace 体系结构的各个组件，包括提供器、探测器、DTrace 内核软件和 `dtrace` 命令。

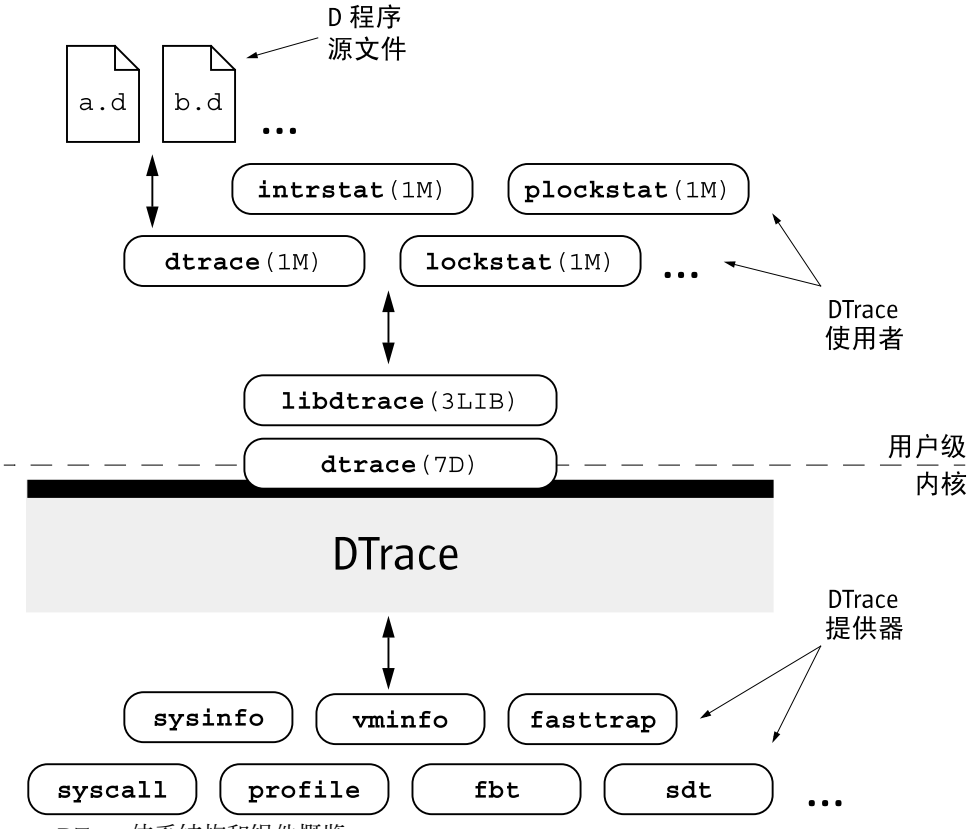

图 1–1 DTrace 体系结构和组件概览

至此，您已了解 DTrace 的工作方式，现在让我们返回来学习 D 编程语言，并开始编写一些更有趣的程序。

变量和算术表达式

接下来的示例程序使用 DTrace `profile` 提供器实现基于时间的简单计数器。 该 profile 提供器可以根据 D 程序中的描述创建新的探测器。 如果为某个整数 *n* 创建一个名为 `profile:::tick-`*n*`sec` 的探测器，该 profile 提供器将创建一个触发间隔为 *n* 秒的探测器。 键入以下源代码，并将其保存在名为 `counter.d` 的文件中。

```
/*
 * Count off and report the number of seconds elapsed
 */
```

```
dtrace:::BEGIN

{

    i = 0;

}


profile:::tick-1sec

{

    i = i + 1;

    trace(i);

}


dtrace:::END

{

    trace(i);

}
```

执行时，程序将统计已用的秒数，直到按下 Ctrl-C 组合键为止，然后在结尾显示总计：

```
# dtrace -s counter.d

dtrace: script 'counter.d' matched 3 probes

CPU     ID                    FUNCTION:NAME

  0  25499                       :tick-1sec         1

  0  25499                       :tick-1sec         2

  0  25499                       :tick-1sec         3

  0  25499                       :tick-1sec         4

  0  25499                       :tick-1sec         5
```

```
  0  25499                         :tick-1sec         6

^C

  0      2                             :END         6

#
```

程序的前三行为注释，用于解释程序将执行的操作。 与 C、C++、Java 编程语言类似，D 编译器会忽略 /* 和 */ 符号之间的任何字符。 可以在 D 程序中的任何位置使用注释，包括探测器子句的内部和外部。

BEGIN 探测器子句定义了一个名为 i 的新变量，并使用以下语句将整数值零赋给该变量：

```
i = 0;
```

与 C、C++ 和 Java 编程语言不同，只需在程序语句中使用便可创建 D 变量，而无需进行显式变量声明。 在程序中首次使用某个变量时，将会根据变量的第一个赋值的类型设置其类型。 在程序的生命周期中，每个变量仅有一个类型，因此后续引用的类型必须与初始赋值的类型相同。 在 counter.d 中，变量 i 首先被赋值为整型常数零，因此其类型设置为 int。 D 提供了与 C 相同的基本整数数据类型，包括：

char	字符或单字节整数
int	缺省整数
short	短整数
long	长整数
long long	扩展的长整数

这些类型的大小取决于操作系统内核的数据模型，如第 2 章中所述。 D 还为各种固定大小的带符号整数类型和无符号整数类型以及由操作系统定义的大量其他类型提供了内置的友好名称。

counter.d 的中心部分是使计数器 i 递增的探测器子句：

```
profile:::tick-1sec

{

    i = i + 1;

    trace(i);

}
```

此子句将探测器命名为 profile:::tick-1sec，以指示 profile 提供器创建一个在可用处理器中每秒触发一次的新探测器。 该子句包括两条语句，第一条语句指定 i 的值为前一个值加一，第二条语句跟踪 i 的新值。 所有常见的 C 算术运算符均可用于 D 程序；在第 2 章中可以找到完整的列表。 与 C 中一样，++ 运算符也可以用作以一为增量递增相应变量的简写。 trace() 函数接受任何 D 表达式作为其参数，因此可以更简明地编写 counter.d，如下所示：

```
profile:::tick-1sec

{

    trace(++i);

}
```

如果要显式控制变量 i 的类型，可以在为变量赋值时将所需类型括在括号中，以便将整数零强制转换为特定类型。 例如，如果要确定 D 中 char 的最大大小，可以按照如下所示更改 BEGIN 子句：

```
dtrace:::BEGIN

{

    i = (char)0;

}
```

在运行 counter.d 一段时间后，应看到跟踪的值增大，然后绕回为零。 如果您没有耐心等待值绕回，可尝试将 profile 探测器名称更改为 profile:::tick-100msec，使计数器每 100 毫秒递增一次或每秒递增 10 次。

谓词

D 语言和其他编程语言（如 C、C++ 和 Java 编程语言）之间的一个主要差别是没有控制流结构（如 if 语句和循环）。 D 程序子句以单个直线语句列表的形式编写，用于跟踪固定数量的可选数据。 D 使用称为谓词的逻辑表达式，提供了根据条件跟踪数据和修改控制流的功能，谓词可用于为程序子句添加前缀。 触发探测器时，在执行与相应的子句关联的任何语句之前，将计算谓词表达式。 如果谓词计算结果为 true（由任何非零值表示），则执行语句列表。 如果谓词计算结果为 false（由零值表示），则不执行任何语句并忽略探测器触发。

为下一个示例键入以下源代码，并将其保存在名为 countdown.d 的文件中：

```
dtrace:::BEGIN

{
```

```
    i = 10;
}

profile:::tick-1sec
/i > 0/
{
    trace(i--);
}

profile:::tick-1sec
/i == 0/
{
    trace("blastoff!");
    exit(0);
}
```

此 D 程序使用谓词实现一个 10 秒的递减计数计时器。 执行时，`countdown.d` 从 10 开始递减计数，然后显示一条消息并退出：

```
# dtrace -s countdown.d
dtrace: script 'countdown.d' matched 3 probes
CPU     ID                    FUNCTION:NAME
    0  25499                         :tick-1sec        10
    0  25499                         :tick-1sec         9
    0  25499                         :tick-1sec         8
    0  25499                         :tick-1sec         7
```

```
    0  25499                      :tick-1sec         6

    0  25499                      :tick-1sec         5

    0  25499                      :tick-1sec         4

    0  25499                      :tick-1sec         3

    0  25499                      :tick-1sec         2

    0  25499                      :tick-1sec         1

    0  25499                      :tick-1sec   blastoff!

#
```

此示例使用 `BEGIN` 探测器将整数 `i` 初始化为 10 以开始递减计数。 接下来，与前一个示例相同，程序使用 `tick-1sec` 探测器实现每秒触发一次的计时器。 请注意，在 `countdown.d` 中，`tick-1sec` 探测器描述用在两条不同的子句中，每条子句使用一个不同的谓词和操作列表。 谓词是置于封闭斜杠 `/ /` 之间的逻辑表达式，出现在探测器名称之后、括住子句语句列表的大括号 `{ }` 之前。

第一个谓词测试 `i` 是否大于零，以指示计时器是否仍在运行：

```
profile:::tick-1sec

/i > 0/

{

    trace(i--);

}
```

关系运算符 `>` 表示大于，如果为 false 则返回整数值零，如果为 true 则返回一。 D 中支持所有 C 关系运算符；在第 2 章中可以找到完整的列表。 如果 `i` 不为零，则脚本将跟踪 `i`，然后使用 `--` 运算符以一为减量递减。

如果 `i` 恰好等于零，则第二个谓词使用 `==` 运算符返回 true，指示递减计数已完成：

```
profile:::tick-1sec

/i == 0/

{

    trace("blastoff!");
```

```
    exit(0);

}
```

当递减计数完成时，`hello.d`、`countdown.d` 使用称为字符串常量、引在双引号中的一个字符序列来显示最终消息，这一点与第一个示例类似。 然后使用`exit()` 函数退出 `dtrace` 并返回到 shell 提示符。

如果返回查看 `countdown.d` 的结构，将会看到通过创建两个具有相同的探测器描述，但谓词和操作不同的子句，有效地创建了以下逻辑流：

```
i = 10

once per second,

  if i is greater than zero

    trace(i--);

  otherwise if i is equal to zero

    trace("blastoff!");

    exit(0);
```

如果您希望使用谓词编写复杂的程序，请首先尝试按照此方式将您的算法直观化，然后将条件结构的每个路径转换为独立的子句和谓词。

现在，我们将谓词与一个新的提供器（`syscall` 提供器）组合在一起，创建第一个真实的 D 跟踪程序。 `syscall` 提供器允许在进入任何 Solaris 系统调用或从该系统调用返回时启用探测器。 以下示例使用 DTrace 对 shell 执行的每次 `read`(2) 或 `write`(2) 系统调用进行观察。 首先打开两个终端窗口，一个用于 DTrace，另外一个包含您将查看的 shell 进程。 在第二个窗口中，键入以下命令以获取此 shell 的进程 ID：

```
# echo $$

12345
```

现在返回到第一个终端窗口并键入以下 D 程序，然后将其保存在名为 `rw.d` 的文件中。 键入程序时，用响应 `echo` 命令后列显的 shell 的进程 ID 替换整型常数 `12345`。

```
syscall::read:entry,

syscall::write:entry
```

```
/pid == 12345/

{



}
```

请注意，由于该程序仅用于跟踪探测器触发通知而不用于跟踪任何其他数据，因此将 rw.d 的探测器子句的主体保留为空。 完成 rw.d 键入后，使用 dtrace 开始编写实验脚本，然后转到第二个 shell 窗口键入一些命令，并在每个命令后按回车键。 键入时，您应在第一个窗口中看到 dtrace 报告所出现的探测器触发，与以下示例类似：

```
# dtrace -s rw.d

dtrace: script 'rw.d' matched 2 probes

CPU     ID                    FUNCTION:NAME

    0     34                      write:entry

    0     32                       read:entry

    0     34                      write:entry

    0     32                       read:entry

    0     34                      write:entry

    0     32                       read:entry

    0     34                      write:entry

    0     32                       read:entry

...
```

您现在会注意到，shell 执行 read(2) 和 write(2) 系统调用从终端窗口中读取字符并回显结果！ 此示例包括许多到现在为止已描述的概念和一些新概念。 首先，要采用相同方式检测 read(2) 和 write(2)，脚本应使用包含多个探测器描述的单条探测器子句，并用逗号分隔这些描述，如下所示：

```
syscall::read:entry,

syscall::write:entry
```

为了获得较好的可读性，每个探测器描述均独占一行。当然，并非一定按此方式编写，但这样做可使脚本可读性更好。接下来，脚本定义一个谓词，该谓词仅与 shell 进程执行的那些系统调用匹配：

```
/pid == 12345/
```

该谓词使用预定义的 DTrace 变量 `pid`，其计算结果始终为与触发相应探测器的线程关联的进程 ID。DTrace 为各种有用的项目（如进程 ID）提供了许多内置的变量定义。以下是一些 DTrace 变量的列表，您可以使用这些变量来编写第一个 D 程序：

变量名称	数据类型	含义
`errno`	`int`	系统调用的当前 `errno` 值
`execname`	`string`	当前进程的可执行文件的名称
`pid`	`pid_t`	当前进程的进程 ID
`tid`	`id_t`	当前线程的线程 ID
`probeprov`	`string`	当前探测器描述的提供器字段
`probemod`	`string`	当前探测器描述的模块字段
`probefunc`	`string`	当前探测器描述的函数字段
`probename`	`string`	当前探测器描述的名称字段

现在，您已经编写了一个实际的检测程序，可尝试通过更改要检测的进程 ID 和系统调用探测器，对系统上运行的不同进程试验该程序。然后，可以进行一次更简单的更改，将 `rw.d` 转变为一个非常简单的系统调用跟踪工具版本（如 `truss(1)`）。空的探测器描述字段充当通配符，可与任何探测器匹配，因此将程序更改为以下新源代码可跟踪 shell 执行的任何系统调用：

```
syscall:::entry

/pid == 12345/

{



}
```

尝试在 shell 中键入一些命令（如 `cd`、`ls` 和 `date`），然后查看 DTrace 程序报告的内容。

输出格式

系统调用跟踪是一种观察大多数用户进程行为的有效方法。 如果您以前以管理员或开发者身份使用过 Solaris truss(1) 实用程序，则可能已了解到，该实用程序是一种用于解决问题的有用工具。 如果以前从未使用过 truss，则可在某个 shell 中键入以下命令立即进行尝试：

```
$ truss date
```

您将会看到 date(1) 执行的所有系统调用的格式化跟踪，结尾紧跟一行输出。 以下示例对前面的 rw.d 程序进行了改进，将其输出格式设置为与 truss(1) 更相似，从而便于您理解输出。 键入以下程序，并将其保存在名为 trussrw.d 的文件中：

示例 1–2 trussrw.d: 使用 truss(1) 输出格式跟踪系统调用

```
syscall::read:entry,
syscall::write:entry
/pid == $1/
{
    printf("%s(%d, 0x%x, %4d)", probefunc, arg0, arg1, arg2);
}


syscall::read:return,
syscall::write:return
/pid == $1/
{
    printf("\t\t = %d\n", arg1);
}
```

在此示例中，将每个谓词中的常数 12345 替换为标签 $1。使用此标签可将所关注的进程指定为脚本参数：编译脚本时，将用第一个参数的值替换 $1。要执行 trussrw.d，使用 dtrace 选项 -q 和 -s，后跟 shell 的进程 ID 作为最后一个参数。 -q 选项指出，dtrace 应为静默模式，并取消前面示例中显示的标题行和 CPU 及 ID 列。 因此，您将仅会看到显式跟踪的数据的输出。 键入以下命令（将 12345 替换为 shell 进程的进程 ID），然后在指定的 shell 中多次按回车键。

```
# dtrace -q -s trussrw.d 12345

                    = 1

write(2, 0x8089e48,    1)               = 1

read(63, 0x8090a38, 1024)               = 0

read(63, 0x8090a38, 1024)               = 0

write(2, 0x8089e48,   52)               = 52

read(0, 0x8089878,    1)                = 1

write(2, 0x8089e48,    1)               = 1

read(63, 0x8090a38, 1024)               = 0

read(63, 0x8090a38, 1024)               = 0

write(2, 0x8089e48,   52)               = 52

read(0, 0x8089878,    1)                = 1

write(2, 0x8089e48,    1)               = 1

read(63, 0x8090a38, 1024)               = 0

read(63, 0x8090a38, 1024)               = 0

write(2, 0x8089e48,   52)               = 52

read(0, 0x8089878,    1)^C

#
```

现在我们来更详细地检查 D 程序及其输出。首先，与前面的程序类似的一条子句检测 shell 的每次 `read`(2) 和 `write`(2) 调用。但对于此示例，将使用一个新函数 `printf()` 来跟踪数据并以特定格式显示数据：

```
syscall::read:entry,

syscall::write:entry

/pid == $1/

{
```

```
    printf("%s(%d, 0x%x, %4d)", probefunc, arg0, arg1, arg2);

}
```

printf() 函数将跟踪数据的功能（就像先前使用的 trace() 函数一样）与以描述的特定格式输出数据和其他文本的功能组合在一起。 printf() 函数指示 DTrace 跟踪与第一个参数之后的每个参数关联的数据，然后使用第一个 printf() 参数描述的规则（称为格式字符串）格式化结果。

格式字符串是包含任意数量的格式转换的规则字符串，其中的每个转换都以% 字符开头，描述如何格式化相应的参数。 格式字符串中的第一个转换对应于第二个 printf() 参数，第二个转换对应于第三个参数，依此类推。 转换之间的所有文本将逐字显示。 % 转换字符之后的字符描述要用于相应参数的格式。以下是 trussrw.d 中使用的三种格式转换的含义：

%d	将相应的值显示为十进制整数
%s	将相应的值显示为字符串
%x	将相应的值显示为十六进制整数

DTrace printf() 的工作方式与 C printf(3C) 库例程或 shell printf(1) 实用程序类似。 如果您以前对 printf() 完全不了解，请参阅第 12 章中对格式和选项的详细说明。 即使您通过其他语言已经熟悉了 printf()，也应该认真阅读本章。在 D 中，将 printf() 作为内置功能提供，并且专为 DTrace 设计了一些新的格式转换供您使用。

为了帮助您编写正确的程序，D 编译器将会针对参数列表验证每个 printf() 格式字符串。请尝试将以上子句中的 probefunc 更改为整数 123。如果运行修改的程序，将会显示一条错误消息，通知您字符串格式转换 %s 不适用于整数参数：

```
# dtrace -q -s trussrw.d

dtrace: failed to compile script trussrw.d: line 4: printf( )

       argument #2 is incompatible with conversion #1 prototype:

            conversion: %s

             prototype: char [] or string (or use stringof)

              argument: int

#
```

要显示读或写系统调用及其参数的名称，请使用 printf() 语句：

```
printf("%s(%d, 0x%x, %4d)", probefunc, arg0, arg1, arg2);
```

以跟踪当前探测器函数和系统调用的前三个整数参数（可从 DTrace 变量 `arg0`、`arg1` 和 `arg2` 中获得）的名称。 有关探测器参数的更多信息，请参见第 3 章。`read(2)` 和 `write(2)` 的第一个参数为文件描述符，以十进制格式显示。 第二个参数为缓冲区地址，格式为十六进制值。 最后一个参数为缓冲区大小，格式为十进制值。 格式说明符 `%4d` 用于第三个参数，指示应使用 `%d` 格式转换显示该值，最小字段宽度为 4 个字符。 如果该整数的宽度小于 4 个字符，`printf()` 将插入额外的空白以对齐输出。

要显示系统调用的结果并完成每一行输出，使用以下子句：

```
syscall::read:return,

syscall::write:return

/pid == $1/

{

    printf("\t\t = %d\n", arg1);

}
```

请注意，除 `entry` 外，`syscall` 提供器还为每个系统调用发布了一个名为`return` 的探测器。 在 syscall `return` 探测器中，DTrace 变量 `arg1` 保存了系统调用的返回值。 返回值的格式为十进制整数。 格式字符串中以反斜杠开头的字符序列分别扩展为 tab 键（`\t`）和新行（`\n`）。 这些转义序列可帮助您显示或记录难以键入的字符。 D 支持与 C、C++ 和 Java 编程语言相同的一组转义序列。 完整的转义序列列表请参阅第 2 章。

数组

D 允许您定义整数和其他类型的变量，以便表示字符串和称为结构和联合的复合类型。 如果您熟悉 C 编程，那么应该很高兴，因为您可以在 D 中使用 C 中使用的任何类型。如果您不是 C 语言专家，也不要担心：第 2 章中描述了所有不同类型的数据类型。 D 还支持一种称为关联数组的特殊类型的变量。 关联数组与常规数组类似，因为它也将一组键与一组值相关联，但是在关联数组中，键不限于固定范围的整数。

D 关联数组可以由任何类型的一个或多个值的列表建立索引。 各个键值一起构成元组，用于对数组建立索引以及访问或修改与该键对应的值。与给定关联数组一起使用的每个元组必须符合相同类型的签名；也就是说，每个元组键的长度必须相同，且具有同样顺序的相同键类型。 整个数组中与给定关联数组的每个元素关联的值也是同一个固定类型。例如，以下 D 语句使用元组签名 `[ string, int ]` 定义一个 `int` 值类型的新关联数组 `a`，并将整数值 456 存储在数组中：

```
a["hello", 123] = 456;
```

在定义数组之后，可以像访问任何其他 D 变量一样访问数组的元素。 例如，以下 D 语句通过将值从 456 递增到 457 来修改 `a` 中先前存储的数组元素：

```
a["hello", 123]++;
```

尚未赋值的任何数组元素的值均设置为零。现在，我们学习在 D 程序中使用关联数组。 键入以下程序，并将其保存在名为 rwtime.d 的文件中：

示例 1–3 rwtime.d:时间 read(2) 和 write(2) 调用

```
syscall::read:entry,

syscall::write:entry

/pid == $1/

{

    ts[probefunc] = timestamp;

}



syscall::read:return,

syscall::write:return

/pid == $1 && ts[probefunc] != 0/

{

    printf("%d nsecs", timestamp - ts[probefunc]);

}
```

与 trussrw.d相同，执行 rwtime.d 时，指定 shell 进程的 ID。 如果键入一些 shell 命令，您将会看到每个系统调用过程中已用的时间。 在其他 shell 中键入以下命令，然后多次按回车键：

```
# dtrace -s rwtime.d ‘pgrep -n ksh‘

dtrace: script ’rwtime.d’ matched 4 probes

CPU     ID                    FUNCTION:NAME

  0     33                      read:return 22644 nsecs

  0     33                      read:return 3382 nsecs
```

```
  0     35                     write:return 25952 nsecs

  0     33                      read:return 916875239 nsecs

  0     35                     write:return 27320 nsecs

  0     33                      read:return 9022 nsecs

  0     33                      read:return 3776 nsecs

  0     35                     write:return 17164 nsecs

...

^C

#
```

要跟踪每个系统调用的已用时间，必须对进入 `read`(2) 和 `write`(2) 以及从它们返回的状态进行检测，并对每个点进行时间采样。 然后，在从给定的系统调用返回时，必须计算第一个时间标记和第二个时间标记之间的差异。 对于每个系统调用可以使用独立的变量，但这可能会使程序在扩展到其他系统调用时变得很乱。 相反，使用由探测器函数名称建立索引的关联数组则简单得多。 以下是第一个探测器子句：

```
syscall::read:entry,

syscall::write:entry

/pid == $1/

{

    ts[probefunc] = timestamp;

}
```

此子句定义一个名为 `ts` 的数组，并将 DTrace 变量 `timestamp` 的值赋给相应的成员。 此变量返回一个始终递增的纳秒计数器的值，与 Solaris 库例程 `gethrtime`(3) 类似。 保存进入时间标记后，对应的返回探测器将再次对 `timestamp` 采样，并报告当前时间与所保存值之间的差异：

```
syscall::read:return,

syscall::write:return

/pid == $1 && ts[probefunc] != 0/
```

```
{

    printf("%d nsecs", timestamp - ts[probefunc]);

}
```

返回探测器中的谓词要求，DTrace 正在跟踪相应的进程，且对应的 `entry` 探测器已触发，并为 `ts[probefunc]` 指定了一个非零值。 此技巧在首次启动 DTrace 时可以消除无效的输出。 如果在执行 `dtrace` 时，shell 已在 `read`(2) 系统调用中等待输入，将会触发 `read:return` 探测器，但前面的第一个 `read`(2) 的 `read:entry` 不会触发，且 `ts[probefunc]` 的计算结果将为零，因为尚未对其赋值。

外部符号和类型

DTrace 检测过程在 Solaris 操作系统内核内部执行，因此除了访问特殊 DTrace 变量和探测器参数外，您还可以访问内核数据结构、符号和类型。 通过这些功能，高级 DTrace 用户、管理员、服务人员和驱动程序开发者可以检查操作系统内核和设备驱动程序的低级行为。 在本书开头的阅读列表中，列出了可以帮助您进一步了解 Solaris 操作系统内核的书籍。

D 使用反引号字符 (‘) 作为访问操作系统（而非 D 程序）中所定义的符号的特殊作用域运算符。例如，Solaris 内核包含一个名为 `kmem_flags` 的系统可调参数的 C 声明，用于启用内存分配器调试功能。有关 `kmem_flags` 的更多信息，请参见《Solaris Tunable Parameters Reference Manual》 。 此可调参数在内核源代码中使用 C 语言声明，如下所示：

```
int kmem_flags;
```

要在 D 程序中跟踪此变量的值，可以编写以下 D 语句：

```
trace(‘kmem_flags);
```

DTrace 将每个内核符号与在对应操作系统 C 代码中用于该符号的类型关联，从而便于对本机操作系统数据结构进行基于源代码的访问。 内核符号名称存储在与 D 变量和函数标识符不同的名称空间中，因此您无需担心这些名称会与 D 变量发生冲突。

至此，您已完成了 DTrace 旋风之旅，并了解了许多对于生成更大、更复杂的 D 程序所必需的基本 DTrace 生成块。 在以下章节中，将描述用于 D 程序的整套规则，并演示 DTrace 如何使复杂的系统性能度量和功能分析变得更简单。 随后，您将会了解到如何使用 DTrace 将用户应用程序行为与系统行为相连，从而使您可以分析整个软件栈。

这才刚刚开始！

◆ ◆ ◆ 第 2 章

2 类型、运算符和表达式

D 提供了用于访问和处理多种数据对象的功能：可以创建和修改变量和数据结构，可以访问操作系统内核和用户进程中定义的数据对象，以及可以声明整数、浮点和字符串常量。D 提供了用于处理对象和创建复杂表达式的 ANSI-C 运算符的超集。 本章详细介绍了类型、运算符和表达式的规则聚合。

标识符名称和关键字

D 标识符名称由大小写字母、数字和下划线组成，其中第一个字符必须为字母或下划线。将保留以下划线 (_) 开头的所有标识符名称以供 D 系统库使用。 应避免在 D 程序中使用这类名称。 根据约定，D 程序员通常对变量使用混合大小写的名称，对常量使用全部大写的名称。

D 语言关键字是保留的特殊标识符，以供在编程语言语法本身中使用。 这些名称始终指定为小写，不能用作 D 变量的名称。

表 2–1 D 关键字

`auto`*	`goto`*	`sizeof`
`break`*	`if`*	`static`*
`case`*	`import`*+	`string`+
`char`	`inline`	`stringof`+
`const`	`int`	`struct`
`continue`*	`long`	`switch`*
`counter`*+	`offsetof`+	`this`+
`default`*	`probe`*+	`translator`+

表 2–1 D 关键字 （续）

do*	provider*+	typedef
double	register*	union
else*	restrict*	unsigned
enum	return*	void
extern	self+	volatile
float	short	while*
for*	signed	xlate+

D 将 ANSI-C 关键字的超集保留用作关键字。 为了供 D 语言将来使用而保留的关键字带有 "* " 标记。 如果尝试使用为了供将来使用而保留的关键字，则 D 编译器将生成语法错误。D 定义了但 ANSI-C 未定义的关键字带有 "+" 标记。 D 提供了 ANSI-C 中出现的类型和运算符的完整聚合。D 编程语言的主要差别是没有控制流结构。 将保留与 ANSI-C 中的控制流关联的关键字，以供将来在 D 中使用。

数据类型和大小

D 为整数和浮点常量提供了基础数据类型。D 程序中只可以对整数执行运算。 浮点常量可用于初始化数据结构，但在 D 中不允许执行浮点运算。D 提供了 32 位和 64 位数据模型供编写程序时使用。 执行程序时使用的数据模型是与活动操作系统内核关联的本机数据模型。可以使用 `isainfo -b` 确定系统的本机数据模型。

下表显示了两种数据模型中每一种的整数类型的名称及其大小。 整数始终按系统的本机字节编码顺序以二的补码形式表示。

表 2–2 D 整数数据类型

类型名称	32 位大小	64 位大小
char	1 字节	1 字节
short	2 字节	2 字节
int	4 字节	4 字节
long	4 字节	8 字节
long long	8 字节	8 字节

整数类型可以带有 `signed` 或 `unsigned` 限定符的前缀。 如果不存在任何符号限定符，则会将该类型假定为带符号。 D 编译器还提供了下表中列出的类型别名：

表 2–3 D 整数类型别名

类型名称	说明
int8_t	1 字节带符号整数
int16_t	2 字节带符号整数
int32_t	4 字节带符号整数
int64_t	8 字节带符号整数
intptr_t	大小等于指针的带符号整数
uint8_t	1 字节无符号整数
uint16_t	2 字节无符号整数
uint32_t	4 字节无符号整数
uint64_t	8 字节无符号整数
uintptr_t	大小等于指针的无符号整数

这些类型别名等效于使用上一个表中的对应基本类型的名称，并相应地为每一种数据模型进行了定义。例如，类型名称 uint8_t 是类型 unsigned char 的别名。有关如何定义自己的类型别名以供在 D 程序中使用的信息，请参见第 8 章。

D 提供了用于与 ANSI-C 声明和类型保持兼容性的浮点类型。 D 中不支持浮点运算符，但可以使用 printf() 函数跟踪浮点数据对象并设置其格式。 可以使用下表中列出的浮点类型：

表 2–4 D 浮点数据类型

类型名称	32 位大小	64 位大小
float	4 字节	4 字节
double	8 字节	8 字节
long double	16 字节	16 字节

D 还提供了特殊类型 string 用于表示 ASCII 字符串。 第 6 章中将更详细地讨论字符串。

常量

可以采用十进制 (12345)、八进制 (012345) 或十六进制 (0x12345) 表示整型常数。八进制（以 8 为基数）常量必须使用前导零作为前缀。 十六进制（以 16 为基数）常量必须使用 0x 或 0X 作为前缀。 在能够表示值的 int、long 和 long long 中，会为整型常数指定最小类型。 如果值为负数，则将使用该类型的带符号版本。 如果值为正数，并且太大以致不适合用带符号的类型表示，则将使用不带符号的类型表示。 您可以将以下后缀之一应用于任何整型常数，以显式指定整型常数的 D 类型：

u 或 U	编译器选择的类型的unsigned 版本
l 或 L	long
ul 或 UL	unsigned long
ll 或 LL	long long
ull 或 ULL	unsigned long long

浮点常量始终采用十进制表示，必须包含小数点 (12.345) 或指数 (123e45)，或者同时包含二者 (123.34e-5)。缺省情况下，对浮点常量指定的类型为 double。 您可以将以下后缀之一应用于任何浮点常量，以显式指定浮点常量的 D 类型：

f 或 F	float
l 或 L	long double

字符常量表示为单个字符或使用一对单引号 ('a') 引起来的转义序列。 对字符常量指定的类型为 int，字符常量等效于值由 ASCII 字符集中该字符的值确定的整型常数。 有关字符及其值的列表，可以参阅 ascii(5)。 也可以在字符常量中使用下表所示的任何特殊转义序列。 D 支持的转义序列与 ANSI-C 中相同。

表 2–5 D 字符转义序列

\a	警报	\\	反斜杠
\b	backspace 键	\?	问号
\f	换页	\'	单引号
\n	新行	\"	双引号
\r	回车	\0*oo*	八进制值 0*oo*
\t	水平制表符	\x*hh*	十六进制值 0x*hh*
\v	垂直制表符	\0	空字符

可以在单引号中包括多个字符说明符，以创建根据相应的字符说明符初始化各个字节的整数。 将从左向右读取字符常量中的字节，并按与操作环境的本机字节存储顺序对应的顺序将各个字节分配到产生的整数。 单个字符常量中最多可以包括八个字符说明符。

任何长度的字符串常量都可以通过引在一对双引号中 ("hello") 来构成。 字符串常量可能不含字面值换行符。要创建包含换行符的字符串，请使用 \n 转义序列来代替字面换行符。字符串常量可以包含上面所示字符常量的任何特殊字符转义序列。 与 ANSI-C 类似，字符串表示为由空字符 (\0) 结尾的字符数组，该空字符隐式添加到所声明的每个字符串常量中。 对字符串常量指定了特殊的 D 类型 string。 D 编译器提供了一组特殊功能，用于比较和跟踪被声明为字符串的字符数组，如第 6 章中所述。

算术运算符

D 提供了下表中所示的二元算术运算符，以供在程序中使用。对于整数，这些运算符含义与在 ANSI-C 中的完全一样。

表 2–6 D 二元算术运算符

+	整数相加
-	整数相减
*	整数相乘
/	整数相除
%	整数取模

D 中只可以对整数操作数或指针（如第 5 章中所讨论）执行运算。 D 程序中不可以对浮点操作数执行运算。 DTrace 执行环境不对整数溢出或下溢采取任何操作。在可能出现溢出和下溢的情况下，您必须自己检查这些情况。

DTrace 执行环境不会自动检查和报告由于不正确使用 / 和 % 运算符而导致的被零除错误。如果 D 程序执行无效的相除操作，则 DTrace 将自动禁用受影响的检测过程并报告错误。DTrace 检测到的错误不会对其他 DTrace 用户或操作系统内核产生影响，因此，不必担心如果 D 程序中包含其中某个未注意到的错误会导致任何破坏。

除了这些二元运算符，+ 和 - 运算符也可用作一元运算符；这些运算符比任何二元算术运算符具有更高的优先级。 表 2–11 中说明了所有 D 运算符的优先级顺序和关联属性。可以通过使用括号 ( ) 对表达式分组来控制优先级。

关系运算符

D 提供了下表中所示的二元关系运算符供在程序中使用。这些运算符都与在 ANSI-C 中具有相同的含义。

表 2–7 D 关系运算符

<	左边的操作数小于右边的操作数
<=	左边的操作数小于或等于右边的操作数
>	左边的操作数大于右边的操作数
>=	左边的操作数大于或等于右边的操作数
==	左边的操作数等于右边的操作数

表 2–7 D 关系运算符　　（续）

`!=`	左边的操作数不等于右边的操作数

关系运算符通常用于编写 D 谓词。每个运算符的计算结果都为 `int` 类型的值，如果条件为 true，则该值等于 1，如果条件为 false，则该值等于 0。

关系运算符可以应用于整数、指针或字符串对。如果比较指针，则结果等于解释为无符号整数的两个指针的整数比较。 如果比较字符串，则结果是通过对两个操作数执行 `strcmp`(3C) 来确定。以下是一些 D 字符串比较及其结果的示例：

```
"coffee" < "espresso"          ... 返回 1 (true)
"coffee" == "coffee"           ... 返回 1 (true)
"coffee" >= "mocha"            ... 返回 0 (false)
```

也可使用关系运算符比较与枚举类型关联的数据对象和由枚举定义的任何枚举器标记。枚举是用于创建命名的整型常数的工具，第 8 章中对其进行了详细介绍。

逻辑运算符

D 提供了以下二元逻辑运算符供在程序中使用。前两个运算符等效于相应的 ANSI-C 运算符。

表 2–8 D 逻辑运算符

`&&`	逻辑 *AND*：两个操作数都为 true 时，结果为 true
`||`	逻辑 *OR*：一个或两个操作数为 true 时，结果为 true
`^^`	逻辑 *XOR*：只有一个操作数为 true 时，结果为 true

逻辑运算符通常用于编写 D 谓词。逻辑运算符 AND 使用简化求值法：如果左边的操作数为 false，则不会计算右边的表达式。 逻辑运算符 OR 也使用简化求值法：如果左边的操作数为 true，则不会计算右边的表达式。 逻辑 XOR 运算符不使用简化求值法：始终会计算两个表达式操作数。

除了二元逻辑运算符，一元 `!` 运算符也可用于对单个操作数执行逻辑否定。它将零操作数转换为一，将非零操作数转换为零。 根据约定，D 程序员在处理要用于表示布尔值的整数时使用 `!`，而在处理非布尔整数时使用 `== 0`，尽管两种表达式在含义上相同。

逻辑运算符可以应用于整数或指针类型的操作数。逻辑运算符将指针操作数解释为无符号整数值。与 D 中的所有逻辑和关系运算符一样，如果操作数具有非零整数值，则将解释为 true，如果具有零整数值，则将解释为 false。

按位运算符

D 提供了以下二元运算符用于处理整数操作数中的各个位。 这些运算符都与在 ANSI-C 中具有相同的含义。

表 2–9 D 按位运算符

&	按位 AND
|	按位 OR
^	按位 XOR
<<	将左边的操作数向左移动右边的操作数指定的位数
>>	将左边的操作数向右移动右边的操作数指定的位数

二元 & 运算符用于从整数操作数中清除位。 二元 | 运算符用于在整数操作数中设置位。 二元 ^ 运算符在只设置了相应操作数位之一的每个位的位置返回 1。

移位运算符用于在指定的整数操作数中向左或向右移位。 向左移位将使用零填充结果右边空位的位置。 使用无符号整数操作数向右移位将使用零填充结果左边空位的位置。使用带符号的整数操作数向右移位将使用符号位的值填充左边空位的位置，也称为算术移位操作。

用负位数将整数值移位，或者移位的位数大于左边操作数本身的位数，将产生未定义的结果。 如果在编译 D 程序时，编译器可以检测到此条件，则 D 编译器将产生错误消息。

除了二元逻辑运算符，一元 ~ 运算符也可用于对单个操作数执行按位否定。它将操作数中的每个“零”位转换为“一”位，并将操作数中的每个“一”位转换为“零”位。

赋值运算符

D 提供了以下二元赋值运算符用于修改 D 变量。 您只可以修改 D 变量和数组。不可以使用 D 赋值运算符修改内核数据对象和常量。 赋值运算符与在 ANSI-C 中具有相同的含义。

表 2–10 D 赋值运算符

=	将左边的操作数设置为等于右边的表达式值
+=	将左边的操作数递增右边的表达式值
-=	将左边的操作数递减右边的表达式值
*=	将左边的操作数与右边的表达式值相乘
/=	将左边的操作数与右边的表达式值相除

表 2–10 D 赋值运算符 （续）

%=	将左边的操作数按右边的表达式值取模
|=	将左边的操作数与右边的表达式值进行按位 OR 运算
&=	将左边的操作数与右边的表达式值进行按位 AND 运算
^=	将左边的操作数与右边的表达式值进行按位 XOR 运算
<<=	将左边的操作数向左移动右边的表达式值指定的位数
>>=	将左边的操作数向右移动右边的表达式值指定的位数

除了赋值运算符 = 以外，所提供的其他赋值运算符是将 = 运算符与先前说明的某个其他运算符配合使用的简写形式。 例如，表达式 `x = x + 1` 等效于表达式 `x += 1`，不同的是表达式 `x` 计算一次。 这些赋值运算符遵守与前面说明的二元格式相同的操作类型规则。

任何赋值运算符的结果都是等于左边表达式的新值的表达式。 您可以将赋值运算符或到现在为止说明的任何运算符与任意复杂度的格式表达式配合使用。 也可以使用括号 ( ) 来对复杂表达式中的条件分组。

递增和递减运算符

D 提供了特殊的一元 ++ 和 -- 运算符用于递增和递减指针和整数。 这些运算符的含义与在 ANSI-C 中相同。这些运算符仅可应用于变量，可以在变量名称之前或之后应用。 如果运算符显示在变量名称之前，则将首先修改变量，于是产生的表达式等于变量的新值。 例如，以下两个表达式产生相同的结果：

```
x += 1;
y = x;
```

```
y = ++x;
```

如果运算符显示在变量名称之后，则在返回变量的当前值以供在表达式中使用之后，将修改该变量。 例如，以下两个表达式产生相同的结果：

```
y = x;
x -= 1;
```

```
y = x--;
```

您可以使用递增和递减运算符来创建新的变量，同时可以不声明这些变量。如果省略变量声明，并将递增或递减运算符应用于变量，则该变量将隐式声明为 `int64_t` 类型。

递增和递减运算符可以应用于整数或指针变量。 应用于整数变量时，运算符将对相应的值递增或递减一。 应用于指针变量时，运算符将对指针地址递增或递减指针所引用的数据类型的大小。 D 中的指针和指针运算将在第 5 章中讨论。

条件表达式

尽管 D 不支持 if-then-else 结构，但它支持使用 ? 和 : 运算符的简单条件表达式。这些运算符可以关联三元表达式（其中，第一个表达式用于根据条件计算另两个表达式中的一个）。例如，以下 D 语句可用于根据 i 的值将变量 x 设置为两个字符串之一。

```
x = i == 0 ? "zero" : "non-zero";
```

在此示例中，将首先计算表达式 i == 0 以确定该表达式为 true 还是 false。如果第一个表达式为 true，则将计算第二个表达式，并且 ?: 表达式将返回其值。如果第一个表达式为 false，则将计算第三个表达式，并且 ?: 表达式将返回其值。

与任何 D 运算符一样，可以在单个表达式中使用多个 ?: 运算符来创建更复杂的表达式。例如，以下表达式将采用一个 char 变量 c，该变量包含 0-9、a-z 或 A-Z 范围中的一个字符，然后在将此字符解释为十六进制（以 16 为基数）整数格式的数字时返回此字符的值。

```
hexval = (c >= '0' && c <= '9') ? c - '0' :

    (c >= 'a' && c <= 'z') ? c + 10 - 'a' : c + 10 - 'A';
```

与 ?: 配合使用的第一个表达式必须为指针或整数，以便计算表达式的实际值。第二个和第三个表达式可以为任何兼容类型。在一个路径返回字符串，另一个路径返回整数的情况下，不可以构造条件表达式。第二个和第三个表达式也不可以调用跟踪函数（如 trace() 或 printf()）。如果要根据条件跟踪数据，请改为使用第 1 章中讨论的谓词。

类型转换

使用不同但兼容的类型构造表达式时，将会执行类型转换以便确定所产生表达式的类型。类型转换的 D 规则与 ANSI-C 中整数的算术转换规则相同。这些规则有时又称为常用算术转换。

以下是说明转换规则的简单方法：每个整数类型按 char、short、int、long、long long 顺序排列，相应的无符号类型分配的级别高于其带符号类型，但低于下一个整数类型。使用两个整数操作数（如 x + y）构造表达式，且操作数为不同的整数类型时，具有最高级别的操作数类型将用作结果类型。

如果需要转换，则首先将低级别的操作数提升为高级别的类型。提升不会实际更改操作数的值：它仅根据符号将值扩展为更大的容器。如果提升无符号的操作数，则所产生整数的未用高序位将使用零填充。如果提升带符号操作数，则将通过执行符号扩展填充未用的高序位。如果将带符号类型转换为无符号类型，则将首先对带符号类型进行符号扩展，然后分配由转换确定的新的无符号类型。

也可以将整数和其他类型从一种类型显式地强制转换为另一种类型。在 D 中，可以将指针和整数强制转换为任何整数或指针类型，但不能转换为其他类型。强制转换和提升字符串和字符数组的规则将在第 6 章中讨论。整数和指针强制转换使用表达式构成，如下所示：

```
y = (int)x;
```

其中，目标类型括在括号中，置于源表达式的前面。 可以通过执行提升将整数强制转换为高级别的类型。 可以通过将整数多余的高顺序位填充零来强制将整数转换为低级别的类型。

因为 D 不允许浮点运算，所以不允许执行浮点操作数转换或强制转换，也没有定义隐式浮点转换的规则。

优先级

下表说明了运算符优先级和关联性的 D 规则。 这些规则有点复杂，但为了保持与 ANSI-C 运算符优先级规则的完全兼容性，这是必需的。表中项的顺序为从最高优先级到最低优先级。

表 2–11 D 运算符优先级和关联性

运算符	关联性
`() [] -> .`	从左到右
`! ~ ++ -- + - * &` (*type*) `sizeof stringof offsetof xlate`	从右到左
`* / %`	从左到右
`+ -`	从左到右
`<< >>`	从左到右
`< <= > >=`	从左到右
`== !=`	从左到右
`&`	从左到右
`^`	从左到右
`|`	从左到右
`&&`	从左到右
`^^`	从左到右
`||`	从左到右
`?:`	从右到左
`= += -= *= /= %= &= ^= |= <<= >>=`	从右到左
`,`	从左到右

表中还有多个运算符我们尚未讨论；后续章节中将对它们进行讨论：

`sizeof`	计算对象的大小（第 7 章）
`offsetof`	计算类型成员的偏移（第 7 章）
`stringof`	将相应操作数转换为字符串（第 6 章）
`xlate`	转换数据类型（第 40 章）
一元 `&`	计算对象的地址（第 5 章）
一元 `*`	取消引用指向对象的指针（第 5 章）
`->` 和 `.`	访问结构类型或联合类型的成员（第 7 章）

表中列出的逗号 (`,`) 运算符用于与 ANSI-C 逗号运算符兼容，可以使用该运算符按照从左到右的顺序计算一组表达式的值，然后返回最右边表达式的值。 提供此运算符是为了与 C 严格兼容，通常不应使用。

运算符优先级表中的 `()` 项表示函数调用；第 1 章 中说明了函数调用（如 `printf()` 和 `trace()`）的示例。D 中也使用逗号列出函数的参数以及用于构成关联数组键的列表。 此逗号与逗号运算符不同，它不一定从左向右计算。 D 编译器不提供对函数参数或关联数组键的计算顺序的保证。 在这些上下文中使用会产生负面影响的表达式（如表达式 `i` 和 `i++` 对）时务必小心。

运算符优先级表中的 `[]` 项表示数组或关联数组引用。 第 1 章说明了关联数组的示例。第 9 章中介绍了称为聚合的特殊类型关联数组。`[]` 运算符同样可用于对大小固定的 C 数组建立索引，如第 5 章中所述。

◆ ◆ ◆ 第 3 章

3 变量

D 提供了两种基本类型的变量供在跟踪程序中使用：标量变量和关联数组。在第一章的示例中简单介绍了这些变量的用法。本章将详细介绍 D 变量的规则以及如何将变量与不同的作用域关联。在第 9 章中将讨论称为聚合的特殊类型的数组变量。

标量变量

标量变量用于表示一个大小固定的数据对象（如整数和指针）。标量变量也可用于表示由一个或多个原始类型或复合类型组成的大小固定的对象。D 提供了用于创建对象数组和复合结构的功能。DTrace 还通过允许字符串扩大到预定义的最大长度，将字符串表示为大小固定的标量。第 6 章中将进一步讨论如何在 D 程序中控制字符串的长度。

在 D 程序中首次对先前未定义的标识符赋值时，会自动创建标量变量。例如，要创建一个名为 x 的 int 类型的标量变量，只需在任何探测子句中为该变量指定一个 int 类型的值：

```
BEGIN

{

    x = 123;

}
```

按照此方式创建的标量变量为全局变量：其名称和数据存储位置只需定义一次，并且在 D 程序的所有子句中都可见。每次引用标识符 x 时，就是在引用与此变量关联的单个存储位置。

与 ANSI-C 不同，D 编程语言不需要显式声明变量。如果确实要声明一个全局变量，以在使用该变量之前为其显式指定名称和类型，则可以在程序中探测子句外部进行声明，如下例所示。在大多数 D 程序中无需显式声明变量，但如果要谨慎控制变量类型，或要在程序的开头使用一组声明和注释来说明该程序的变量及其含意，显式声明变量会很有用。

```
int x; /* declare an integer x for later use */


BEGIN

{

    x = 123;

    ...

}
```

与 ANSI-C 声明不同，D 变量声明不能指定初始值。必须使用 `BEGIN` 探测子句指定所有初始值。 在首次引用变量之前，DTrace 将使用零填充所有全局变量存储空间。

D 语言定义对 D 变量的大小和数量不设限制，但是 DTrace 实现和系统中可用内存会定义一些限制。D 编译器在编译程序时将强制执行可应用的任何限制。 在第 16 章中可了解有关如何调整与程序限制相关的选项的更多信息。

关联数组

关联数组用于表示数据元素的集合，可通过指定一个称为键的名称来检索这些数据元素。D 关联数组的键由称为元组的标量表达式值的列表构成。 您可以将数组元组本身视为函数的假想参数列表，在引用该数组时系统将调用该函数来检索相应的数组值。 每个 D 关联数组都有一个固定的键签名，该签名由固定数量的元组元素组成，其中每个元素具有给定的固定类型。可以在 D 程序为每个数组定义不同的键签名。

关联数组与大小固定的常规数组不同，因为关联数组对元素的数量没有预定义的限制。关联数组的元素可以通过任何元组建立索引，而不是仅使用整数作为键，这些元素并不存储在预先分配的连续存储位置中。 在 C、C++ 或 Java™ 语言程序中要使用散列表或其他简单字典数据结构的情况下，使用关联数组会很有用。 使用关联数组可以为 D 程序中捕获的事件和状态创建动态历史记录，使用该历史记录可创建更复杂的控制流。

要定义关联数组，请编写以下形式的赋值表达式：

name [*key*] = *expression* ;

其中，*name* 是任何有效的 D 标识符，*key* 是逗号分隔的一个或多个表达式的列表。 例如，以下语句使用键签名 `[ int, string ]` 定义关联数组 `a`，并将整数值 456 存储在由元组 `[ 123, "hello" ]` 命名的位置中：

```
a[123, "hello"] = 456;
```

对于给定数组中的所有元素，数组中包含的每个对象的类型也是固定的。 由于已对 a 指定了整数 456，所以该数组中存储的每个后续值也将为 int 类型。 可以使用第 2 章中定义的任何赋值运算符，根据为每个运算符定义的操作数规则来修改关联数组元素。 如果尝试进行不兼容的赋值，D 编译器将产生相应的错误消息。您可以将对标量变量使用的任何类型用于关联数组键或值。 但不能将关联数组作为键或值嵌套在另一个关联数组中。

您可以使用与数组键签名兼容的任何元组引用关联数组。 元组兼容性规则与函数调用和变量赋值的规则相似：元组的长度必须相同，且实际参数列表中的每种类型必须与正式键签名中的相应类型兼容。例如，如果按照如下所示定义关联数组 x：

```
x[123ull] = 0;
```

则键签名为 unsigned long long 类型，值为 int 类型。 也可使用表达式 x['a'] 引用该数组，因为根据第 55 页中的 “类型转换”中说明的算术转换规则，由 int 类型的字符常量 'a' 组成的元组和长度 1 与键签名 unsigned long long 兼容。

如果需要在使用 D 关联数组前对其进行显式声明，可以在程序源代码中探测子句外部创建数组名称和键签名的声明。

```
int x[unsigned long long, char];


BEGIN

{

    x[123ull, 'a'] = 456;

}
```

定义关联数组后，将允许引用兼容键签名的任何元组，即使先前未对要引用的元组赋值。访问未赋值的关联数组元素定义为返回使用零填充的对象。此定义的后果是，在对关联数组元素赋非零值之前，将不会为该元素分配基础存储空间。相反，对关联数组元素赋值零将会导致 DTrace 解除分配基础存储空间。此行为很重要，因为可对关联数组元素分配的外部动态变量空间是有限的。如果在尝试分配时该空间已用完，分配将会失败，并生成一条错误消息，指示将进行动态变量删除。不再使用的关联数组元素将赋值零。有关消除动态变量删除的其他方法，请参见第 16 章。

线程局部变量

利用 Dtrace，可以声明只作用于每个操作系统线程的局部变量存储空间，这与本章前面介绍的全局变量空间相对。 在要启用探测器并使用一些标记或其他数据来标记将触发探测器的每个线程的情况下，可使用线程局部变量。 在 D 中创建解决此问题的程序很方便，因为线程局部变量在 D 代码中共享一个公用名称，但引用与各个线程关联的独立数据存储空间。通过将 -> 运算符应用于特殊标识符 `self` 可引用线程局部变量：

```
syscall::read:entry

{

    self->read = 1;

}
```

此 D 代码段在 read(2) 系统调用中启用探测器，并将名为 `read` 的线程局部变量与触发该探测器的每个线程关联。 与全局变量类似，线程局部变量在首次赋值时自动创建，并采用该赋值语句右边使用的类型（本例中为 `int`）。

每次在 D 程序中引用 `self->read` 变量时，引用的数据对象是与触发相应的 DTrace 探测器时执行的操作系统线程关联的对象。 可将线程局部变量视为关联数组，其隐式索引是系统中线程标识组成的元组。 线程的标识在系统的生命周期中是唯一的：如果该线程退出，并使用相同的操作系统数据结构创建一个新线程，则此新线程不会重用同一个 DTrace 线程局部存储标识。

定义线程局部变量后，即使先前未将要引用的的变量指定给该特定线程，也可以在系统中的任何线程中引用该局部变量。 如果尚未指定线程局部变量的线程副本，则该副本的数据存储空间将定义为使用零填充。 与关联数组元素一样，在对线程局部变量指定非零值之前，不会为该变量分配基础存储空间。另外还与关联数组元素一样，为线程局部变量赋值零将会导致 DTrace 解除分配基础存储空间。不再使用的线程局部变量将赋值零。有关微调动态变量空间（从该空间分配线程局部变量）的其他方法，请参见第 16 章。

可以在 D 程序中定义任何类型的线程局部变量，包括关联数组。 以下是一些线程局部变量定义的示例：

```
self->x = 123;              /* integer value */

self->s = "hello";              /* string value */

self->a[123, 'a'] = 456;    /* associative array */
```

与任何 D 变量一样，在使用线程局部变量之前，无需对其进行显式声明。 如果确实要创建声明，可以通过在程序子句外部在声明之前加上关键字 `self` 来进行声明：

```
self int x;    /* declare int x as a thread-local variable */


syscall::read:entry

{

    self->x = 123;

}
```

线程局部变量保存在与全局变量不同的名称空间中，因此可以重用这些名称。请记住，如果在程序中过载名称，x 和 self->x 是不同的变量。以下示例说明了如何使用线程局部变量。在文本编辑器中，键入以下程序并将其保存在名为 rtime.d 的文件中：

示例 3–1 rtime.d: 计算 read(2) 中花费的时间

```
syscall::read:entry

{

    self->t = timestamp;

}



syscall::read:return

/self->t != 0/

{

    printf("%d/%d spent %d nsecs in read(2)\n",

        pid, tid, timestamp - self->t);



    /*

     * We're done with this thread-local variable; assign zero to it to

     * allow the DTrace runtime to reclaim the underlying storage.

     */
```

示例 3–1 rtime.d: 计算 read(2) 中花费的时间　　（续）

```
    self->t = 0;

}
```

现在进入 shell 并开始运行程序。 等待几秒钟，您将会看到一些输出。 如果没有显示输出，请尝试运行一些命令。

```
# dtrace -q -s rtime.d

100480/1 spent 11898 nsecs in read(2)

100441/1 spent 6742 nsecs in read(2)

100480/1 spent 4619 nsecs in read(2)

100452/1 spent 19560 nsecs in read(2)

100452/1 spent 3648 nsecs in read(2)

100441/1 spent 6645 nsecs in read(2)

100452/1 spent 5168 nsecs in read(2)

100452/1 spent 20329 nsecs in read(2)

100452/1 spent 3596 nsecs in read(2)

...

^C

#
```

rtime.d 使用名为 t 的线程局部变量捕获任何线程进入 read(2) 的时间标记。 然后在返回子句中，程序通过从当前时间标记中减去 self->t，来列显 read(2) 中所花费的时间。内置 D 变量 pid 和 tid 报告执行 read(2) 的线程的进程 ID 和线程 ID。因为报告此信息后将不再需要 self->t，随后将会对其赋值 0，使 DTrace 可以在当前线程中重用与 t 关联的基础存储空间。

通常，在没有执行任何操作的情况下，也可以看到多行输出，这是因为即使您未执行任何操作，服务器进程和守护进程始终都在后台执行 read(2)。 尝试将 rtime.d 的第二条子句更改为使用 execname 变量，以列显执行 read(2) 的进程的名称，从而了解更多信息：

```
printf("%s/%d spent %d nsecs in read(2)\n",
```

```
    execname, tid, timestamp - self->t);
```

如果发现特别关注的进程，添加一个谓词以了解有关其 read(2) 行为的更多信息：

```
syscall::read:entry

/execname == "Xsun"/

{

    self->t = timestamp;

}
```

子句局部变量

也可以定义每个 D 程序子句将重用其存储空间的 D 变量。子句局部变量与 C、C++ 或 Java 语言程序中的自动变量类似，这些变量在每次调用函数期间处于活动状态。与所有 D 程序变量一样，子句局部变量会在首次赋值时自动创建。 通过将 -> 运算符应用于特殊标识符 this 可引用这些变量并对这些变量赋值：

```
BEGIN

{

    this->secs = timestamp / 1000000000;

    ...

}
```

如果要在使用子句局部变量前对其进行显式声明，可使用 this 关键字进行声明：

```
this int x;   /* an integer clause-local variable */

this char c;  /* a character clause-local variable */



BEGIN

{

    this->x = 123;
```

```
    this->c = 'D';

}
```

子句局部变量仅在给定探测子句的生命周期中处于活动状态。在 DTrace 执行与给定探测器的子句关联的操作后，将会回收所有子句局部变量的存储空间，并重用于下一条子句。因此，子句局部变量是唯一不使用零进行初始填充的 D 变量。请注意，如果程序的单个探测器包含多条子句，在执行这些子句时，所有子句局部变量将会保持不变，如下例所示：

示例 3–2 `clause.d`: 子句局部变量

```
int me;            /* an integer global variable */

this int foo;        /* an integer clause-local variable */



tick-1sec

{

    /*

     * Set foo to be 10 if and only if this is the first clause executed.

     */

    this->foo = (me % 3 == 0) ? 10 : this->foo;

    printf("Clause 1 is number %d; foo is %d\n", me++ % 3, this->foo++);

}



tick-1sec

{

    /*

     * Set foo to be 20 if and only if this is the first clause executed.

     */

    this->foo = (me % 3 == 0) ? 20 : this->foo;

    printf("Clause 2 is number %d; foo is %d\n", me++ % 3, this->foo++);
```

示例 3–2 clause.d: 子句局部变量　　（续）

```
}


tick-1sec

{

    /*

     * Set foo to be 30 if and only if this is the first clause executed.

     */

    this->foo = (me % 3 == 0) ? 30 : this->foo;

    printf("Clause 3 is number %d; foo is %d\n", me++ % 3, this->foo++);

}
```

因为子句始终按程序顺序执行，且子句局部变量在启用相同探测器的不同子句中具有持久性，所以运行上述程序将始终产生同样的输出：

```
# dtrace -q -s clause.d

Clause 1 is number 0; foo is 10

Clause 2 is number 1; foo is 11

Clause 3 is number 2; foo is 12

Clause 1 is number 0; foo is 10

Clause 2 is number 1; foo is 11

Clause 3 is number 2; foo is 12

Clause 1 is number 0; foo is 10

Clause 2 is number 1; foo is 11

Clause 3 is number 2; foo is 12

Clause 1 is number 0; foo is 10
```

```
Clause 2 is number 1; foo is 11

Clause 3 is number 2; foo is 12

^C
```

虽然子句局部变量在启用相同探测器的各条子句中具有持久性，但在为给定探测器执行的第一条子句中，未定义这些变量的值。在使用每条子句局部变量前，请务必为其指定相应的值，否则程序可能产生意外结果。

可使用任何标量变量类型定义子句局部变量，但不可以使用子句局部作用域定义关联数组。子句局部变量的作用域仅适用于相应的变量数据，而不适用于为变量定义的名称和类型标识。一旦定义子句局部变量，就可以在任何后续 D 程序子句中使用该名称和类型签名。存储位置在不同子句中不会一直保持不变。

可以使用子句局部变量积累计算的中间结果，或作为其他变量的临时副本。访问子句局部变量要比访问关联数组快得多。因此，如果需要在同一个 D 程序子句中多次引用关联数组值，首先将该值复制到一条子句局部变量中，然后重复引用该局部变量，将获得更高的效率。

内置变量

下表提供了 D 内置变量的完整列表。所有这些变量都是标量全局变量；D 语言中当前未定义线程局部变量、子句局部变量或内置关联数组。

表 3–1 DTrace 内置变量

类型和名称	说明
`int64_t arg0, ..., arg9`	探测器的前 10 个输入参数表示为原始的 64 位整数。如果传递到当前探测器的参数少于 10 个，则其余变量将返回 0。
`args[]`	为当前探测器键入的参数（如果存在）。可使用整数索引访问 `args[]` 数组，但将每个元素定义为与给定探测器参数对应的类型。例如，如果通过 `read(2)` 系统调用探测器来引用 `args[]`，则`args[0]` 为 `int` 类型，`args[1]` 为 `void *` 类型，`args[2]` 为 `size_t` 类型。
`uintptr_t caller`	进入当前探测器之前的当前线程的程序计数器位置。
`chipid_t chip`	当前物理芯片的 CPU 芯片标识符。有关更多信息，请参见第 26 章。

表 3–1 DTrace 内置变量 （续）

类型和名称	说明
`processorid_t cpu`	当前 CPU 的 CPU 标识符。有关更多信息，请参见第 26 章。
`cpuinfo_t *curcpu`	当前 CPU 的 CPU 信息。有关更多信息，请参见第 26 章。
`lwpsinfo_t *curlwpsinfo`	与当前线程关联的轻量进程 (lightweight process, LWP) 的 LWP 状态。此结构将在 `proc`(4) 手册页中详细说明。
`psinfo_t *curpsinfo`	与当前线程关联的进程的进程状态。此结构将在 `proc`(4) 手册页中详细说明。
`kthread_t *curthread`	当前线程 (`kthread_t`) 的操作系统内核的内部数据结构的地址。`kthread_t` 在 `<sys/thread.h>` 中定义。有关此变量和其他操作系统数据结构的更多信息，请参阅 Solaris Internals。
`string cwd`	与当前线程关联的进程的当前工作目录名称。
`uint_t epid`	当前探测器的已启用的探测器 ID (enabled probe ID, EPID)。该整数唯一标识使用特定谓词和操作集启用的特定探测器。
`int errno`	此线程最后一次执行的系统调用返回的错误值。
`string execname`	传递到 `exec`(2) 以执行当前进程的名称。
`gid_t gid`	当前进程的实际组 ID。
`uint_t id`	当前探测器的探测器 ID。此 ID 是当前探测器的系统范围内的唯一标识符，由 DTrace 发布，并列在 `dtrace -l` 的输出中。
`uint_t ipl`	触发探测器时当前 CPU 的中断优先级 (interrupt priority level, IPL)。有关 Solaris 操作系统内核中的中断级别和中断处理的更多信息，请参阅 Solaris Internals。
`lgrp_id_t lgrp`	当前 CPU 所属的延迟组的延迟组 ID。有关更多信息，请参见第 26 章。
`pid_t pid`	当前进程的进程 ID。
`pid_t ppid`	当前进程的父进程 ID。
`string probefunc`	当前探测器说明的函数名称部分。
`string probemod`	当前探测器说明的模块名称部分。
`string probename`	当前探测器说明的名称部分。

表 3–1 DTrace 内置变量 （续）

类型和名称	说明
`string probeprov`	当前探测器说明的提供器名称部分。
`psetid_t pset`	包含当前 CPU 的处理器集的处理器集 ID。有关更多信息，请参见第 26 章。
`string root`	与当前线程关联的进程的根目录名称。
`uint_t stackdepth`	触发探测器时当前线程的栈帧深度。
`id_t tid`	当前线程的线程 ID。对于与用户进程关联的线程，该值等于 `pthread_self`(3C) 调用的结果。
`uint64_t timestamp`	纳秒时间标记计数器的当前值。此计数器从过去的任意时间点递增，仅用于相对计算。
`uid_t uid`	当前进程的实际用户 ID。
`uint64_t uregs[]`	触发探测器时当前线程的已保存的用户模式寄存器值。`uregs[]` 数组的用法将在第 33 章中讨论。
`uint64_t vtimestamp`	纳秒时间标记的当前值，实际是当前线程在 CPU 中已运行的时间减去 DTrace 谓词和操作中所花费时间而得到的值。此计数器从过去的任意时间点递增，仅用于相对时间计算。
`uint64_t walltimestamp`	自 1970 年 1 月 1 日 00:00 世界标准时间以来的当前纳秒数。

D 语言中的内置函数（如 `trace()`）将在第 10 章中讨论。

外部变量

D 使用反引号字符 (‘) 作为访问操作系统（而非 D 程序）中所定义变量的特殊作用域运算符。例如，Solaris 内核包含一个名为 `kmem_flags` 的系统可调参数的 C 声明，用于启用内存分配器调试功能。有关`kmem_flags` 的更多信息，请参见《Solaris Tunable Parameters Reference Manual》。此可调参数在内核源代码中声明为 C 变量，如下所示：

```
int kmem_flags;
```

要在 D 程序中访问此变量的值，请使用 D 表示法：

```
‘kmem_flags
```

DTrace 将每个内核符号与在相应操作系统 C 代码中用于该符号的类型关联，从而使基于源代码访问本机操作系统数据结构更方便。要使用外部操作系统变量，需要访问相应的操作系统源代码。

从 D 程序访问外部变量时，访问的是其他程序的内部实现详细信息（如操作系统内核或其设备驱动程序）。这些实现详细信息不能构成可靠的稳定接口。在下次升级软件的相应部分时，所编写的依赖于这些详细信息的任何 D 程序可能会停止工作。因此，外部变量通常由内核和设备驱动程序开发者以及服务人员使用，以便使用 DTrace 调试性能或功能问题。要了解 D 程序稳定性的更多信息，请参见第 39 章。

内核符号名称存储在与 D 变量和函数标识符不同的名称空间中，因此您无需担心这些名称会与 D 变量发生冲突。在变量前加上一个反引号时，D 编译器为了找到匹配的变量定义，会使用已装入模块的列表按顺序搜索已知的内核符号。由于 Solaris 内核支持使用独立的名称空间动态装入的模块，所以在活动的操作系统内核中可多次使用相同的变量名称。可以通过在符号名称中指定应访问其变量的内核模块的名称并加上反引号，来解决这些名称冲突。例如，每个可装入内核模块通常提供一个 `_fini`(9E) 函数，以便引用名为 `foo` 的内核模块提供的 `_fini` 函数的地址，可写为：

```
foo`_fini
```

根据操作数类型的一般规则，可将任何 D 运算符应用于外部变量，但那些修改值的运算符除外。启动 DTrace 后，D 编译器将装入对应于活动内核模块的变量名称集，因此不需要声明这些变量。不能将修改变量值的任何运算符应用于外部变量，例如 = 或 += 。出于安全原因，DTrace 将会阻止损坏或破坏所观察软件的状态。

◆ ◆ ◆ 第 4 章

4 D 程序结构

D 程序由一组子句组成，这些子句说明要启用的探测器和要绑定到这些探测器的谓词和操作。D 程序也包含变量声明（如第 3 章中所介绍）和新类型定义（如第 8 章中所介绍）。本章正式介绍 D 程序的总体结构和用于构造与多个探测器匹配的探测器说明的功能。还将讨论如何结合 D 程序使用 C 预处理程序 cpp。

探测器子句和声明

如我们已提供过的示例所示，D 程序源文件由一条或多条探测器子句组成，这些子句说明 DTrace 启用的检测过程。每条探测器子句具有以下一般形式：

probe descriptions

/ predicate /

{

  action statements

}

谓词和操作语句列表可以省略。探测器子句外部的任何指令都称为声明。只可以在探测器子句外部使用声明。封闭的 { } 内部不允许使用声明，声明不能分布在如上所示的探测器子句的元素之间。可以使用空格分隔任何 D 程序元素，并缩进操作语句。

可以使用声明来声明 D 变量和外部 C 符号（如第 3 章中所讨论），或者定义要在 D 中使用的新类型（如第 8 章中所介绍）。称为 *pragma* 的特殊 D 编译器指令也可以出现在 D 程序中的任何位置（包括探测器子句的外部）。在以 # 字符开头的行上指定 D pragma。例如，D pragma 可用于设置运行时 DTrace 选项；有关详细信息，请参见第 16 章。

探测器说明

每条 D 程序子句都以一个或多个探测器说明的列表开头，每个说明都采用以下一般形式：

provider:*module*:*function*:*name*

如果省略探测器说明中的一个或多个字段，D 编译器将从右向左解释指定的字段。例如，探测器说明 `foo:bar` 将匹配包括函数 `foo` 和名称 `bar` 的探测器，而不考虑探测器的提供器和模块字段的具体值。所以，探测器说明更准确地说是一种模式，可用于根据名称匹配一个或多个探测器。

应编写指定所有四个字段分界符的 D 探测器说明，以便可以在左侧指定需要的提供器。如果未指定提供器，在多个提供器以相同的名称发布探测器时可能会得到意外的结果。类似地，将来版本的 DTrace 可能包括新的提供器，这些提供器的探测器可能会无意中与所指定的探测器说明部分匹配。可以指定提供器，但保留所有模块、函数和名称字段为空，以与提供器的任何探测器匹配。例如，说明 `syscall:::` 可用于匹配 DTrace `syscall` 提供器发布的每个探测器。

探测器说明还支持与 shell *globbing* 模式匹配语法（如 sh(1) 中所说明）类似的模式匹配语法。在将探测器与说明匹配之前，DTrace 将在每个说明字段中扫描字符 `*`、`?` 和 `[`。如果这些字符中的某一个出现在探测器说明字段中，并且前面没有 `\`，会将该字段视为一种模式。说明模式必须与给定探测器的整个相应字段匹配。要成功匹配和启用探测器，完整的探测器说明必须与每个字段匹配。非模式的探测器说明字段必须与该探测器的相应字段完全匹配。空的说明字段将与任何探测器匹配。

下表中是探测器名称模式中可识别的特殊字符：

表 4–1 与字符匹配的探测器名称模式

符号	说明
`*`	匹配任何字符串，包括空字符串。
`?`	匹配任何单个字符。
`[ ... ]`	匹配任何一个封闭字符。使用 - 分隔的一对字符将匹配这对字符之间的任何字符，包括这对字符。如果 `[` 之后的第一个字符是 `!`，则将与集合中未包括的任何字符匹配。
`\`	将下一个字符解释为其本身，不具有任何特殊含义。

可以在探测器说明的任何或所有四个字段中使用模式匹配字符。通过在命令行中将模式与 `dtrace -l` 一起使用，可以使用模式列出匹配的探测器。例如，命令 `dtrace -l -f kmem_*` 列出函数中名称以前缀 `kmem_` 开头的所有 DTrace 探测器。

如果要为多个探测器说明或说明模式指定相同的谓词和操作，可以将说明放置在逗号分隔的列表中。例如，以下 D 程序将跟踪每次触发探测器时的时间标记，这些探测器与包含单词 "`lwp`" 或 "`sock`" 的系统调用入口关联。

```
syscall::*lwp*:entry, syscall::*sock*:entry

{

    trace(timestamp);

}
```

探测器说明也可以使用整数探测器 ID 指定探测器。例如，子句：

```
12345

{

    trace(timestamp);

}
```

可用于启用由 `dtrace -l -i 12345` 报告的探测器 ID 12345。应始终使用人工可读的探测器说明编写 D 程序。装入和卸载 DTrace 提供器内核模块时或者在重新引导后，不能保证整数探测器 ID 仍然不变。

谓词

谓词是置于斜杠 `/ /` 中的表达式，在触发探测器时进行计算，以确定是否应执行关联的操作。 谓词是用于在 D 程序中生成更复杂的控制流的主要条件结构。 您可以完全省略任何探测器的探测器子句的谓词部分，在此情况下，触发探测器时将始终执行操作。

谓词表达式可以使用前面说明的任何 D 运算符，可以引用任何 D 数据对象（如变量和常数）。 谓词表达式的计算结果必须为整数或指针类型的值，以便可以将其视为 true 或 false。 与所有 D 表达式一样，零值将解释为 false，任何非零值将解释为 true。

操作

探测器操作由分号 (`;`) 分隔的语句列表说明，括在花括号 `{ }` 中。 如果仅想说明在特定 CPU 中触发特定探测器而不跟踪任何数据，或不执行任何其他操作，可以指定一个内部不包含任何语句的空花括号集合。

C 预处理程序的用法

用于定义 Solaris 系统接口的 C 编程语言包括在 C 程序编译中执行一组初始步骤的预处理程序。 C 预处理程序通常用于定义宏替换（在此情况下，C 程序中的某个标记将替换为另外一组预定义的标记），或者用于包括系统头文件的副本。 可以通过指定 `dtrace -C` 选项将 C 预处理程序与 D 程序配合使用。 此选项使 `dtrace` 首先执行程序源文件中的 `cpp`(1) 预处理程序，然后将结果传递到 D 编译器。 C 编程语言中详细介绍了 C 预处理程序。

D 编译器自动装入与操作系统实现关联的 C 类型说明集，但也可以使用预处理程序来包括其他类型的定义，例如您自己的 C 程序中使用的类型。还可以使用预处理程序执行其他任务（如创建将扩展为 D 代码块和其他程序元素的宏）。 如果将预处理程序与 D 程序一起使用，则只能使用包含有效 D 声明的文件。 一般的 C 头文件仅包括类型和符号的外部声明，这些声明可以由 D 编译器正确解释。 D 编译器无法解析包括其他程序元素（如 C 函数源代码）的 C 头文件，将生成相应的错误消息。

◆ ◆ ◆ 第 5 章

指针和数组

指针是操作系统内核或用户进程的地址空间中数据对象的内存地址。D 允许创建和处理指针，并将它们存储在变量和关联数组中。本章介绍用于指针的 D 语法、可用于创建或访问指针的运算符，以及指针和固定大小标量数组之间的关系。另外，还讨论了有关在不同地址空间中使用指针的问题。

注 – 如果您是有经验的 C 或 C++ 语言程序员，可略过本章的大部分内容，因为 D 指针语法与相应的 ANSI-C 语法相同。您应阅读第 84 页中的 “指向 DTrace 对象的指针”和第 84 页中的 “指针和地址空间”，因为这两章介绍了特定于 DTrace 的功能和问题。

指针和地址

Solaris 操作系统使用称为虚拟内存的技术，在您的系统中为每个用户进程提供独立的内存资源的虚拟视图。内存资源的虚拟视图又称为地址空间，它将一定范围的地址值（用于 32 位地址空间的 `[0 ... 0xffffffff]` 或者用于 64 位地址空间的 `[0 ... 0xffffffffffffffff]`）与一组转换相关联，操作系统和硬件使用这些转换将每个虚拟地址转换为相应的物理内存位置。D 中的指针是一些数据对象，用于存储整数虚拟地址值并将其与 D 类型（说明存储在相应内存位置中数据的格式）关联。

可通过以下方式将某个 D 变量声明为指针类型：首先指定被引用数据的类型，然后将星号 (*) 附加到类型名称以指示要声明指针类型。例如，以下声明：

```
int *p;
```

声明了名为 `p` 的 D 全局变量，它是指向整数的指针。此声明意味着 `p` 本身是大小为 32 位或 64 位的整数，它的值表示位于内存中某个位置的另一整数的地址。因为编译格式的 D 代码在操作系统内核本身中触发探测器时执行，所以 D 指针通常是与内核的地址空间关联的指针。可使用 `isainfo(1) -b` 命令来确定活动操作系统内核用于指针的位数。

如果要创建指向内核中某个数据对象的指针，可使用 & 运算符来计算该对象的地址。例如，操作系统内核源代码将声明 int kmem_flags 可调参数。可以通过在 D 中跟踪对该对象的名称应用 & 运算符的结果来跟踪此 int 的地址：

```
trace(&‘kmem_flags);
```

* 运算符可用于引用通过指针寻址的对象，并且充当 & 运算符的逆向运算符。例如，以下两个 D 代码段在含义上是等效的：

```
p = &‘kmem_flags;
trace(*p);
```

```
trace(‘kmem_flags);
```

左边代码段创建 D 全局变量指针 p。因为 kmem_flags 对象属于 int 类型，所以 &‘kmem_flags 的结果类型为 int *（即，指向 int 的指针）。左边代码段将跟踪 *p 的值，该值可以跟踪指向数据对象 kmem_flags 的指针。因此，此代码段与右侧代码段相同，后者直接使用名称来跟踪数据对象的值。

指针安全

如果您是 C 或 C++ 语言程序员，在阅读上节内容后可能会有些担心，因为您知道在程序中错误地使用指针会导致程序崩溃。DTrace 是一个强大、安全的环境，在其中执行 D 程序时，错误地使用指针不会导致程序崩溃。即使您确实编写了有很多错误的 D 程序，无效的 D 指针访问也绝对不会导致 DTrace 或操作系统内核故障或崩溃。相反，DTrace 软件会检测所有无效的指针访问，禁用检测过程并报告问题以便进行调试。

如果您使用过 Java 编程语言进行编程，则您可能知道，基于同样的安全原因，Java 语言不支持指针。因为指针在 C 语言中是操作系统实现的固有部分，所以它们在 D 中是必需的，但 DTrace 实现了与 Java 编程语言中相同的安全机制，可以避免包含错误的程序破坏自身或彼此破坏。DTrace 的错误报告类似于 Java 编程语言的运行时环境，它可以检测编程错误并向您报告异常。

要了解 DTrace 的错误处理和报告，请编写使用指针并且有意出错的 D 程序。在编辑器中，键入以下 D 程序并将其保存在名为 badptr.d 的文件中：

示例 5–1 badptr.d: DTrace 错误处理的演示

```
BEGIN
{
    x = (int *)NULL;
    y = *x;
```

示例 5–1 badptr.d: DTrace 错误处理的演示 （续）

```
    trace(y);

}
```

badptr.d 程序将创建名为 x 的 D 指针，它是指向 int 的指针。程序将特殊的无效指针值 NULL 赋给此指针，该值是地址 0 的内置别名。根据约定，地址 0 始终定义为无效，所以 NULL 可用作 C 程序和 D 程序中的标记值。程序使用强制转换表达式将 NULL 转换为指向整数的指针。然后程序使用表达式 *x 取消引用指针，并且将结果赋给另一个变量 y，然后尝试跟踪 y。在执行 D 程序时，DTrace 会在执行 y = *x 语句时检测无效的指针访问，并且报告错误：

```
# dtrace -s badptr.d

dtrace: script '/dev/stdin' matched 1 probe

CPU     ID                    FUNCTION:NAME

dtrace: error on enabled probe ID 1 (ID 1: dtrace:::BEGIN): invalid address

(0x0) in action #2 at DIF offset 4

dtrace: 1 error on CPU 0

^C

#
```

使用无效指针的程序可能产生的另一个问题就是对齐错误。按照体系结构约定，基础数据对象（如整数）将根据大小在内存中对齐。例如，2 字节整数在 2 的倍数的地址上对齐，4 字节整数在 4 的倍数的地址上对齐，依此类推。如果对指向 4 字节整数的指针取消引用，并且指针地址是无效值（不是 4 的倍数），则访问将失败，并且产生对齐错误。D 中的对齐错误几乎都是指示：指针因为 D 程序中的错误而包含无效的值或损坏的值。可通过将 badptr.d 的源代码更改为使用地址 (int *)2 而不是 NULL 来创建示例对齐错误。因为 int 是 4 字节，并且 2 不是 4 的倍数，所以表达式 *x 将导致 DTrace 对齐错误。

有关 DTrace 错误机制的详细信息，请参见第 207 页中的 “ERROR 探测器”。

数组声明和存储

除了第 3 章中说明的动态关联数组之外，D 还支持标量数组。标量数组是一组固定长度的连续内存位置，每个位置存储一个相同类型的值。可通过使用从零开始的整数引用每个位置来访问标量数组。标量数组直接对应于 C 和 C++ 中数组的概念和语法。虽然在 D 中使用标量数组不如使用关联数组和其高级对应项（聚合）那么频繁，但在访问 C 中声明的现有操作系统数组数据结构时需要标量数组。聚合将在第 9 章中介绍。

由 5 个整数组成的 D 标量数组应这样声明：使用类型 `int`，并在声明后加上用方括号括起来的元素数目，如下所示：

```
int a[5];
```

下图显示了数组存储的可视表示：

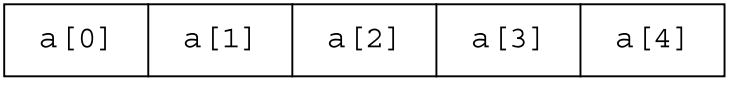

图 5–1 标量数组表示

D 表达式 `a[0]` 用于引用第一个数组元素，`a[1]` 引用第二个数组元素，依此类推。从语法的角度来看，标量数组和关联数组非常相似。可声明由 5 个整数（通过整数键引用）组成的关联数组，如下所示：

```
int a[int];
```

并且使用表达式 `a[0]` 来引用此数组。但从存储和实现角度来看，两个数组有很大的不同。静态数组 `a` 由 5 个连续内存位置（编号从 0 开始）组成，并且索引表示存储空间中为此数组分配的偏移。另一方面，关联数组没有预定义大小，并且不会将元素存储在连续内存位置中。此外，关联数组键与相应的值存储位置无关。可访问关联数组元素 `a[0]` 和 `a[-5]`，并且 DTrace 只分配两个词的存储空间，它们可能连续，也可能不连续。关联数组键是相应值的抽象名称，与值存储位置无关。

如果使用初始赋值创建数组，并将一个整数表达式用作数组索引（如 `a[0] = 2`），则 D 编译器将始终创建新的关联数组，即使在此表达式中 `a` 还可解释为对标量数组的赋值。在此情况下，必须预先声明标量数组，以便 D 编译器能够了解数组大小的定义并推断此数组为标量数组。

指针和数组关系

就像在 ANSI-C 中一样，指针和数组在 D 中有特殊关系。数组由与其第一个存储位置的地址关联的变量表示。指针也是具有所定义类型的存储位置的地址，所以 D 允许将 array [ ] 索引表示法与指针变量和数组变量配合使用。例如，以下两个 D 代码段的含义等价：

```
p = &a[0];
```

```
trace(a[2]);
```

```
trace(p[2]);
```

在左边代码段中，通过对表达式 `a[0]` 应用 `&` 运算符，指针 `p` 被赋给 `a` 中的第一个数组元素的地址。表达式 `p[2]` 将跟踪第三个数组元素（索引为 2）的值。因为 `p` 现在包含与 `a` 关联的同一地址，所以此表达式将生成与 `a[2]` 相同的值，如右侧代码段中所示。等价的一个后果就是 C 和 D 允许您访问任何指针或数组的任何索引。编译器或 DTrace 运行时环境不会执行数组界限检查。如果访问超出数组预定义值的内存，则会产生意外结果或者 DTrace 会报告无效地址错误，如先前示例中所示。与平常一样，不会破坏 DTrace 本身或操作系统，但您需要调试 D 程序。

指针与数组之间的区别在于：指针变量引用一个单独的存储块，该存储块包含另一存储空间的整数地址。数组变量会对数组存储空间本身命名，而不是对包含数组位置的整数位置命名。下图中显示了此区别：

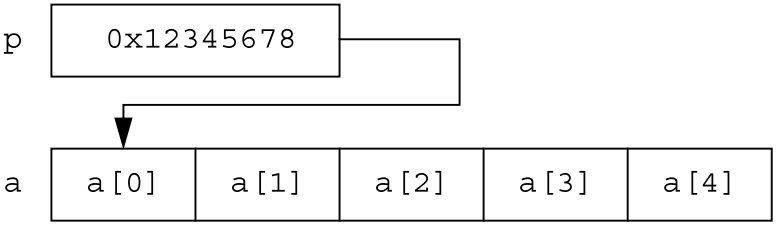

图 5–2 指针和数组存储

如果尝试对指针和标量数组赋值，则 D 语法中将显示此区别。如果 `x` 和 `y` 都是指针变量，则表达式 `x = y` 合法；表达式只将 `y` 中的指针地址复制到 `x` 命名的存储位置。如果 `x` 和 `y` 是标量数组变量，则表达式 `x = y` 不合法。在 D 中不能将数组作为一个整体赋值。但是，可在允许使用指针的任何上下文中使用数组变量或符号名。如果 `p` 是指针而 `a` 是数组，则允许使用语句 `p = a`，此语句等价于语句 `p = &a[0]`。

指针运算

因为指针只是用作内存中其他对象地址的整数，所以 D 提供了一组功能来对指针执行运算。但是，指针运算与整数运算并不完全相同。指针运算通过将指针引用的类型大小与操作数相乘或相除，来隐式调整基础地址。以下 D 代码段将说明此属性：

```
int *x;



BEGIN

{

    trace(x);

    trace(x + 1);

    trace(x + 2);

}
```

此代码段创建整数指针 x 并跟踪其值，首先将值递增 1，然后将值递增 2。如果创建并执行此程序，DTrace 会报告整数值 0，4 和 8。

因为 x 是指向整数的指针（大小为 4 字节），递增 x 将对基础指针值增加 4。在使用指针来引用连续存储位置（如数组）时，此属性很有用。例如，如果将数组 a 的地址赋给 x（如 图 5–2 中所示），则表达式 `x + 1` 将等价于表达式 `&a[1]`。类似地，表达式 `*(x + 1)` 将引用值 `a[1]`。每次使用 `+=`、`+` 或 `++` 运算符递增指针的值时，D 编译器将实现指针运算。

在从左边的指针中减去某个整数时、从一个指针中减去另一个指针时或者对指针应用 `--` 运算符时，也将应用指针运算。例如，以下 D 程序将跟踪结果 2：

```
int *x, *y;

int a[5];



BEGIN

{

    x = &a[0];

    y = &a[2];
```

```
    trace(y - x);

}
```

通用指针

有时在 D 程序中，在不指定指针引用的数据类型的情况下表示或处理通用指针地址会很有用。可使用类型 `void *`（其中关键字 `void` 表示缺少特定信息）或使用内置类型别名 `uintptr_t`（它表示对应当前数据模型中某个指针的大小的无符号整数类型别名）来指定通用指针。不能对 `void *` 类型的对象应用指针运算，并且只有在先将这些指针的类型强制转换为另一类型之后才能取消对它们的引用。在需要对指针值执行整数运算时，可将指针强制转换为 `uintptr_t` 类型。

可在需要指向另一数据类型的指针的任何上下文（如关联数组元组表达式或赋值语句的右侧）中使用指向 `void` 的指针。同样，可在需要指向 `void` 的指针的上下文中使用指向任何数据类型的指针。要用指向非 `void` 类型的指针来代替另一非 `void` 指针类型，需要使用显式强制类型转换。必须始终使用显式强制类型转换将指针转换为整数类型（如 `uintptr_t`），或者将这些整数转换回相应的指针类型。

多维数组

多维标量数组在 D 中使用较少，提供它们是为了与 ANSI-C 兼容，以及观察和访问在 C 中使用此功能创建的操作系统数据结构。通过基本类型和其后用方括号（`[ ]`）括起来的一系列连续的标量数组大小来声明多维数组。例如，要声明固定大小的二维矩形数组（12 行 x 34 列），应编写以下声明：

```
int a[12][34];
```

可使用类似表示法来访问多维标量数组。例如，要访问存储在第 0 行第 1 列中的值，应编写以下 D 表达式：

```
a[0][1]
```

多维标量数组值的存储位置是通过将行号与声明的总列数相乘并加上列号计算得出的。

应该注意的是，不要将多维数组语法与关联数组访问的 D 语法相混淆（即 `a[0][1]` 与 `a[0, 1]` 不同）。如果将不兼容的元组与关联数组配合使用，或者尝试通过关联数组来访问标量数组，则 D 编译器将报告相应的错误消息并拒绝编译程序。

指向 DTrace 对象的指针

D 编译器禁止使用 & 运算符来获取指向 DTrace 对象（如关联数组、内置函数和变量）的指针。禁止获取这些变量的地址，可以使 DTrace 运行时环境根据需要在探测器触发之间重新对变量定位，以便更有效地管理程序所需的内存。如果创建复合结构，则可以构造不检索 DTrace 对象存储的内核地址的表达式。应避免在 D 程序中创建这类表达式。如果需要使用这样的表达式，一定不要在探测器触发之间高速缓存地址。

在 ANSI-C 中，还可使用指针执行间接函数调用或赋值，如将使用一元 * 取消引用运算符的表达式放在赋值运算符的左边。在 D 中，不允许这些使用指针的表达式类型。您只能通过使用名称或对 D 标量数组或关联数组应用数组索引运算符 [] 来直接对 D 变量赋值。只能通过第 10 章中指定的名称来调用 DTrace 环境定义的函数。D 中不允许那些使用指针进行的间接函数调用。

指针和地址空间

指针是一个地址，用于将某些虚拟地址空间转换为物理内存块。DTrace 在操作系统内核本身的地址空间中执行 D 程序。整个 Solaris 系统将管理许多地址空间：操作系统内核使用一个地址空间，每个用户进程也使用一个地址空间。因为每个地址空间表现为它可以访问系统上的所有内存，所以同一虚拟地址指针值可在地址空间中重复使用，但会转换为不同的物理内存。因此，在编写使用指针的 D 程序时，必须注意对应于要使用的指针的地址空间。

例如，如果使用 `syscall` 提供器来检测将指向整数或整数数组的指针作为参数（如 `pipe(2)`）的系统调用入口，则使用 * 或 [] 运算符来对指针或数组取消引用是无效操作，因为要使用的地址是执行系统调用的用户进程地址空间中的地址。在 D 中，对此地址应用 * 或 [] 运算符将导致内核地址空间访问，这又将导致无效地址错误或对 D 程序返回意外数据（取决于地址是否与有效内核地址相匹配）。

要通过 DTrace 探测器访问用户进程内存，必须对用户地址空间指针应用第 10 章中介绍的 `copyin()`、`copyinstr()` 或 `copyinto()` 函数。要注意的是，为了避免混淆，在编写 D 程序时应对存储用户地址的变量进行相应的命名和注释。还可将用户地址存储为 `uintptr_t`，这样就不会无意中编译取消引用它们的 D 代码。第 33 章中介绍了在用户进程中使用 DTrace 的方法。

◆ ◆ ◆ 第 6 章

6 字符串

DTrace 可以对跟踪和处理字符串提供支持。本章介绍用于声明和处理字符串的完整的 D 语言功能集。 与 ANSI-C 不同，D 中的字符串具有自己的内置类型和运算符支持，以便您可以在跟踪程序中轻松而明确地使用这些字符串。

字符串表示

在 DTrace 中，字符串表示为由空字节（即，值为零的字节，通常写为 `'\0'`）结尾的字符数组。字符串的可见部分的长度是可变的，具体取决于空字节的位置，但是 DTrace 将以大小固定的数组存储每个字符串，以便每个探测器跟踪一致的数据量。 字符串的长度不能超过此预定义的字符串限制，但可以在 D 程序中或在 `dtrace` 命令行上通过调整 `strsize` 选项来修改该限制。 有关可调选项的更多信息，请参阅第 16 章。 缺省字符串限制为 256 字节。

D 语言提供了显式 `string` 类型，而不是使用 `char *` 类型来引用字符串。 `string` 类型与 `char *` 等效，因为它是字符序列的地址，但在将 D 编译器和 D 函数（如 `trace()`）应用于 `string` 类型的表达式时，可提供增强功能。 例如，在需要跟踪字符串的实际字节数时，字符串类型消除了 `char *` 类型的不确定性。 在 D 语句中：

```
trace(s);
```

如果 `s` 属于类型 `char *`，则 DTrace 将跟踪指针 `s` 的值（即，跟踪一个整数地址值）。 在 D 语句中：

```
trace(*s);
```

根据 * 运算符的定义，D 编译器将取消对指针 `s` 的引用并跟踪该位置的单个字符。 对于处理设计用于引用单个字符或由字节大小整数（非字符串且不以空字节结尾）构成的数组的字符指针，这些行为很重要。 在 D 语句中：

```
trace(s);
```

如果 s 属于 string 类型，则该 string 类型指示 D 编译器，需要 DTrace 跟踪字符地址存储在变量 s 中的以空字符结尾的字符串。 也可以对 string 类型的表达式执行词法比较，如第 87 页中的 “字符串比较”中所说明。

字符串常量

字符串常量引在双引号 (") 中，D 编译器自动为其分配 string 类型。 您可以定义任意长度的字符串常量，该长度仅受系统中允许 DTrace 使用的内存数量的限制。 结尾的空字节 (\0) 由 D 编译器自动添加到所声明的任何字符串常量中。字符串常量对象的大小是与字符串关联的字节数加上表示结尾空字节的一个附加字节。

字符串常量可能不含字面值换行符。要创建包含换行符的字符串，请使用 \n 转义序列代替字面值换行符。 字符串常量也可包含任何为表 2–5中的字符常量定义的特殊字符转义序列。

字符串赋值

与 char * 变量的赋值不同，字符串是按值而不是按引用复制。 字符串赋值使用 = 运算符进行，从源操作数中复制实际的字符串字节，直到包含变量左侧的空字节为止，该变量必须属于 string 类型。可以通过对变量分配 string 类型的表达式，来新建 string 类型的变量。例如，以下 D 语句：

```
s = "hello";
```

将新建一个 string 类型的变量 s，并将六个字节的字符串 "hello" 复制到该变量中（五个可列显字符和一个空字节）。 字符串赋值与 C 库函数 strcpy(3C) 类似，不同之处在于，如果源字符串超出目标字符串的存储空间限制，则结果字符串将自动按此限制截断。

也可以为字符串变量分配与字符串类型兼容的表达式。 在此情况下，D 编译器将自动提升源表达式为字符串类型，然后执行字符串赋值。D 编译器允许将 char * 类型或 char[*n*] 类型（即，任意大小的 char 类型的标量数组）的任意表达式提升为string 类型。

字符串转换

使用强制类型转换表达式或应用 stringof 特殊运算符，都可以将其他类型的表达式显式转换为 string 类型，其含义等价：

```
s = (string) expression                s = stringof ( expression )
```

stringof 运算符紧密绑定到右侧的操作数。为了表达清晰，通常使用括号将表达式括起来，但并非严格要求这样做。

任何标量类型的表达式（如指针、整数、或标量数组地址）都可以转换为字符串。其他类型的表达式（如 `void`）不能转换为 `string`。如果将无效地址错误地转换为字符串，DTrace 安全功能将会阻止破坏系统或 DTrace，但您可能最后会跟踪一系列不可识别的字符。

字符串比较

D 过载二元关系运算符，并允许使用这些运算符进行字符串比较和整数比较。只要两个操作数均属于 `string` 类型，或一个操作数属于 `string` 类型而另一个操作数可以提升为 `string` 类型，则关系运算符就可以进行字符串比较，如第 86 页中的 “字符串赋值”中所述。所有关系运算符均可用于比较字符串：

表 6–1 用于字符串的 D 关系运算符

`<`	左边的操作数小于右边的操作数
`<=`	左边的操作数小于或等于右边的操作数
`>`	左边的操作数大于右边的操作数
`>=`	左边的操作数大于或等于右边的操作数
`==`	左边的操作数等于右边的操作数
`!=`	左边的操作数不等于右边的操作数

与整数一样，每个运算符计算为 `int` 类型的值，如果条件为 true，则该值等于 1，如果条件为 false，则等于 0。

关系运算符逐字节比较两个输入字符串，这与 C 库例程 `strcmp`(3C) 类似。每个字节使用 ASCII 字符集中的相应整数值进行比较，如 `ascii`(5) 中所示，直到读取空字节或达到最大字符串长度为止。以下是一些 D 字符串比较及其结果的示例：

```
"coffee" < "espresso"        ... 返回 1 (true)
"coffee" == "coffee"         ... 返回 1 (true)
"coffee" >= "mocha"          ... 返回 0 (false)
```

◆ ◆ ◆ 第 7 章

7 结构和联合

相关变量的集合可以组合成称为结构和联合的复合数据对象。 通过为这些对象创建新的类型定义，可在 D 中对其进行定义。 您可以将新类型用于任何 D 变量，包括关联的数组值。 本章介绍了用于创建和处理这些复合类型及与其交互的 D 运算符的语法和语义， 并使用多个演示 DTrace `fbt` 和 `pid` 提供器用法的示例程序，对结构和联合的语法进行说明。

结构

D 关键字 `struct` 为 *structure* 的缩写，用于引入由一组其他类型组成的新类型。 新的结构类型可用作 D 变量和数组的类型，从而使您可以采用单一名称定义相关变量组。 D 结构与 C 和 C++ 中对应的构造相同。 如果您采用 Java 编程语言编写程序，可将 D 结构视为只包含数据成员但不包含方法的类。

我们假定您希望在 D 中创建一个较为复杂的系统调用跟踪程序，用于记录与 shell 执行的每个 `read`(2) 和 `write`(2) 系统调用有关的多种因素，如已用时间、调用数以及作为参数传递的最大字节计数。 您可以编写一条 D 子句，在三个单独的关联数组中记录这些属性，如下例所示：

```
syscall::read:entry, syscall::write:entry

/pid == 12345/

{

    ts[probefunc] = timestamp;

    calls[probefunc]++;

    maxbytes[probefunc] = arg2 > maxbytes[probefunc] ?

        arg2 : maxbytes[probefunc];
```

```
}
```

但是，该子句效率很低，因为 DTrace 必须创建三个单独的关联数组，并存储与每个数组的 probefunc 对应的同一元组值的各个副本。 相反，可以使用结构来节省空间并使程序易于阅读和维护。 首先，在程序源文件顶部声明一个新的结构类型：

```
struct callinfo {

    uint64_t ts;      /* timestamp of last syscall entry */

    uint64_t elapsed; /* total elapsed time in nanoseconds */

    uint64_t calls;   /* number of calls made */

    size_t maxbytes;  /* maximum byte count argument */

};
```

struct 关键字后跟一个可选标识符，用于引用现在称为 struct callinfo 的新类型。 然后，将结构成员括在一组大括号 { } 中，并使用分号 (;) 结束整个声明。 定义每个结构成员使用的语法与 D 变量声明相同，即首先列出成员类型，然后是命名成员的标识符，以及另一个分号 (;)。

结构声明本身仅定义新类型，而不会创建任何变量或在 DTrace 中分配存储空间。 完成声明之后，即可在整个 D 程序的其余部分将 struct callinfo 用作类型，并且每个类型为 struct callinfo 的变量都将存储结构模板描述的四个变量副本。 这些成员将根据成员列表按顺序分配内存，并在成员间引入数据对象对齐所必需的填充空间。

您可以通过 "." 运算符编写以下格式的表达式，使用成员标识符名称来访问各成员值：

variable-name.*member-name*

以下示例为使用新结构类型的改进程序。 请转到编辑器并键入以下 D 程序，然后将其保存在名为 rwinfo.d 的文件中：

示例 7–1 rwinfo.d: 收集 read(2) 和 write(2) 统计信息

```
struct callinfo {

    uint64_t ts;      /* timestamp of last syscall entry */

    uint64_t elapsed; /* total elapsed time in nanoseconds */

    uint64_t calls;   /* number of calls made */

    size_t maxbytes;  /* maximum byte count argument */
```

示例 7–1 rwinfo.d:收集 read(2) 和 write(2) 统计信息　　（续）

```
};

struct callinfo i[string];    /* declare i as an associative array */

syscall::read:entry, syscall::write:entry
/pid == $1/
{
    i[probefunc].ts = timestamp;
    i[probefunc].calls++;
    i[probefunc].maxbytes = arg2 > i[probefunc].maxbytes ?
        arg2 : i[probefunc].maxbytes;
}

syscall::read:return, syscall::write:return
/i[probefunc].ts != 0 && pid == $1/
{
    i[probefunc].elapsed += timestamp - i[probefunc].ts;
}

END
{
    printf("        calls  max bytes  elapsed nsecs\n");
    printf("------  -----  ---------  -------------\n");
```

示例 7–1 rwinfo.d: 收集 read(2) 和 write(2) 统计信息　　（续）

```
    printf("  read  %5d  %9d  %d\n",
        i["read"].calls, i["read"].maxbytes, i["read"].elapsed);
    printf(" write  %5d  %9d  %d\n",
        i["write"].calls, i["write"].maxbytes, i["write"].elapsed);
}
```

键入该程序后，请运行 dtrace -q -s rwinfo.d，并指定其中一个 shell 进程。 然后在 shell 中键入数条命令，并在完成 shell 命令输入后，在 dtrace 终端按 Ctrl-C 组合键，以触发 END 探测器并列显结果：

```
# dtrace -q -s rwinfo.d ‘pgrep -n ksh‘
^C
        calls  max bytes  elapsed nsecs
------  -----  ---------  -------------
  read     36       1024  3588283144
 write     35         59  14945541
#
```

结构指针

在 C 和 D 中，通过指针引用结构很常见。您可以使用运算符 -> 来通过指针访问结构成员。如果 struct s 包含成员 m，并且您具有指向名为 sp 的结构的指针（即 sp 为 struct s * 类型的变量），则可以使用 * 运算符，首先取消对 sp 指针的引用，以便访问该成员：

```
struct s *sp;

(*sp).m
```

或者，可将 -> 运算符用作该表示法的简写。 如果 sp 为指向结构的指针，则以下两个 D 代码段的含义完全相同。

```
(*sp).m                 sp->m
```

DTrace 提供了多个属于结构指针的内置变量，包括 curpsinfo 和 curlwpsinfo。 这些指针分别引用结构 psinfo 和 lwpsinfo，并且其内容提供了有关当前进程及轻量进程 (lightweight process, LWP)（与触发当前探测器的线程关联）状态的信息快照。 Solaris LWP 是用户线程的内核表示，Solaris 线程和 POSIX 线程接口是基于该进程建立的。 为方便起见，DTrace 采用与 /proc 文件系统文件 /proc/ *pid*/psinfo 和 /proc/*pid*/lwps/ *lwpid*/lwpsinfo 相同的格式导出此信息。 /proc 结构可供观察工具和调试工具（如 ps(1)、pgrep(1) 和 truss(1)）使用；系统头文件 <sys/procfs.h> 中定义了该结构，proc(4) 手册页中也有其说明。 下面列出了几个使用 curpsinfo 的示例表达式，及其类型和含义：

curpsinfo->pr_pid	pid_t	当前进程 ID
curpsinfo->pr_fname	char []	可执行文件名
curpsinfo->pr_psargs	char []	初始命令行参数

您应在随后通过检查 <sys/procfs.h> 头文件以及 proc(4) 中的相应说明，查看完整的结构定义。 下例将通过匹配命令行参数，使用 pr_psargs 成员来确定所关注的进程。

在 C 程序中，经常使用结构来创建复杂的数据结构，因此，描述和引用来自 D 的结构的能力也可提供强大的功能，以便观察 Solaris 操作系统内核及其系统接口的内部工作方式。 除使用前述的 curpsinfo 结构外，下例还通过观察 ksyms(7D) 驱动程序和 read(2) 请求之间的关系，检查某些内核结构。 该驱动程序使用两个称作 uio(9S) 和 iovec(9S) 的常见结构，来响应从字符设备文件 /dev/ksyms 读取的请求。

uio(9S) 手册页介绍了可通过名称 struct uio 或类型别名 uio_t 访问的 uio 结构。该结构用于说明与在内核和用户进程之间复制数据有关的 I/O 请求。 uio 又包含由一个或多个 iovec(9S) 结构组成的数组，如果使用 readv(2) 或 writev(2) 系统调用请求多个块，则每个结构都说明一部分请求的 I/O。 在 struct uio 上运行的内核设备驱动程序接口 (device driver interface, DDI) 例程中有一个是函数 uiomove(9F)，该函数是用于响应用户进程 read(2) 请求并将数据复制到用户进程的内核驱动程序函数系列之一。

ksyms 驱动程序用于管理名为 /dev/ksyms 的字符设备文件，该文件似乎为包含内核符号表相关信息的 ELF 文件，但实际上是驱动程序使用当前装入内核的模块集创建的一种假象。 该驱动程序使用 uiomove(9F) 例程来响应 read(2) 请求。 下例说明，/dev/ksyms 的 read(2) 的参数和调用将调用通过驱动程序与 uiomove(9F) 匹配，以将结果复制到 read(2) 中指定位置的用户地址空间。

我们可以使用 strings(1) 实用程序和 -a 选项，强制从 /dev/ksyms 进行串读取。 请在 shell 中尝试运行 strings -a /dev/ksyms，然后查看生成的输出结果。 在编辑器中，键入示例脚本的第一条子句，然后将其保存在名为 ksyms.d 的文件中。

```
syscall::read:entry

/curpsinfo->pr_psargs == "strings -a /dev/ksyms"/
```

```
{

    printf("read %u bytes to user address %x\n", arg2, arg1);

}
```

第一条子句使用表达式 `curpsinfo->pr_psargs` 来访问和匹配 `strings`(1) 命令的命令行参数，以便脚本在跟踪参数之前选择正确的 `read`(2) 请求。 请注意，通过使用运算符 `==`（左侧参数为 `char` 类型的数组，右侧参数为字符串），D 编译器将指示左侧参数应提升为字符串，并执行字符串比较。 在某个 shell 中键入并执行命令 `dtrace -q -s ksyms.d`，然后在另一个 shell 中键入命令 `strings -a /dev/ksyms`。 执行 `strings`(1) 时，将会显示与以下示例类似的 DTrace 输出：

```
# dtrace -q -s ksyms.d

read 8192 bytes to user address 80639fc

read 8192 bytes to user address 80639fc

read 8192 bytes to user address 80639fc

read 8192 bytes to user address 80639fc

...

^C

#
```

此示例可使用常用的 D 编程方法进行扩展，以便跟踪该初始 `read`(2) 请求的线程深入内核。 进入 `syscall::read:entry` 中的内核后，接下来的脚本将设置线程局部标志变量，用于指示此线程为关注线程，并在 `syscall::read:return` 中清除该标志。 设置该标志后，可在其他探测器上将其用作谓词，以检测 `uiomove`(9F) 等内核函数。 DTrace 函数边界跟踪 (function boundary tracing,`fbt`) 提供器发布了用于进入和返回该内核中定义的函数（包括 DDI 中的那些函数）的探测器。 请键入下列使用 `fbt` 提供器检测 `uiomove`(9F) 的源代码，然后再次将其保存到文件 `ksyms.d` 中：

示例 7–2 `ksyms.d`: 跟踪 `read`(2) 和 `uiomove`(9F) 的关系

```
/*

 * When our strings(1) invocation starts a read(2), set a watched flag on

 * the current thread.  When the read(2) finishes, clear the watched flag.
```

示例 7–2 ksyms.d: 跟踪 read(2) 和 uiomove(9F) 的关系 （续）

```
 */
syscall::read:entry
/curpsinfo->pr_psargs == "strings -a /dev/ksyms"/
{
    printf("read %u bytes to user address %x\n", arg2, arg1);
    self->watched = 1;
}

syscall::read:return
/self->watched/
{
    self->watched = 0;
}

/*
 * Instrument uiomove(9F).  The prototype for this function is as follows:
 * int uiomove(caddr_t addr, size_t nbytes, enum uio_rw rwflag, uio_t *uio);
 */
fbt::uiomove:entry
/self->watched/
{
    this->iov = args[3]->uio_iov;
```

示例 7–2 ksyms.d: 跟踪 read(2) 和 uiomove(9F) 的关系 （续）

```
    printf("uiomove %u bytes to %p in pid %d\n",

        this->iov->iov_len, this->iov->iov_base, pid);

}
```

该示例的最后一条子句使用线程本地变量 self->watched，来确定所关注的内核线程进入 DDI 例程 uiomove(9F) 的时间。 在此之后，脚本使用内置的 args 数组访问 uiomove() 的第四个参数 (args[3])，该参数是一个指向代表该请求的 struct uio 的指针。 D 编译器会自动将 args 数组的每个成员，与受检测的内核例程的 C 函数原型所对应的类型进行关联。 uio_iov 成员包含一个指向该请求的 struct iovec 的指针。 为供子句中的子句局部变量 this->iov 使用，保存了该指针副本。 在最后一条语句中，脚本取消了对 this->iov 的引用，以访问 iovec 成员 iov_len 和 iov_base，二者分别表示 uiomove(9F) 的长度（以字节为单位）以及目标基本地址。 这些值应与驱动程序中发出的 read(2) 系统调用的输入参数匹配。 请转到您的 shell 并运行 dtrace -q -s ksyms.d，然后在另一个 shell 中再次输入命令 strings -a /dev/ksyms。 显示的输出将与以下示例类似：

```
# dtrace -q -s ksyms.d

read 8192 bytes at user address 80639fc

uiomove 8192 bytes to 80639fc in pid 101038

read 8192 bytes at user address 80639fc

uiomove 8192 bytes to 80639fc in pid 101038

read 8192 bytes at user address 80639fc

uiomove 8192 bytes to 80639fc in pid 101038

read 8192 bytes at user address 80639fc

uiomove 8192 bytes to 80639fc in pid 101038

...

^C

#
```

在您的输出中，地址和进程 ID 将有所不同，但您应看到 read(2) 的输入参数与通过 ksyms 驱动程序传递至 uiomove(9F) 的参数匹配。

联合

联合是 ANSI-C 和 D 支持的另一种复合类型，它与结构密切相关。 联合是一种复合类型，其中，定义了一组不同类型的成员，并且所有成员对象都占用同一存储区域。 因此，根据联合的指定方式，联合是一个变体类型的对象，在任何给定时间内仅有一个成员有效。 通常，使用其他变量或状态部分来指示当前处于有效状态的联合成员。 联合的大小为其最大成员的大小，用于联合的内存对齐为联合成员所需的最大对齐值。

Solaris `kstat` 框架定义了一个包含联合的结构，用于在以下示例中说明和观察 C 和 D 联合。 `kstat` 框架用于导出一组表示内核统计信息（如内存使用情况和 I/O 吞吐量）的指定计数器。 使用该框架可以实现 `mpstat`(1M) 和 `iostat`(1M) 等实用程序。 此框架使用 `struct kstat_named` 来描述指定的计数器及其值，定义如下：

```
struct kstat_named {

    char name[KSTAT_STRLEN]; /* name of counter */

    uchar_t data_type;    /* data type */

    union {

        char c[16];

        int32_t i32;

        uint32_t ui32;

        long l;

        ulong_t ul;

        ...

    } value;    /* value of counter */

};
```

为了直观起见，缩短了检查声明。 完整的结构定义可在 `<sys/kstat.h>` 头文件中找到。有关其说明，请参阅 `kstat_named`(9S)。 以上声明在 ANSI-C 和 D 中均有效，该声明定义了一个结构，其中包含一个作为其成员之一的联合值，该值具有各种类型的成员，具体取决于计数器的类型。 请注意，由于该联合本身是在另一种类型 `struct kstat_named` 中声明的，因此省略了该联合类型的正式名称。 此声明样式称作匿名联合。 名为 `value` 的成员具有上述声明描述的联合类型，但是此联合类型本身没有名称，因为无需在任何其他位置使用该类型。 为结构成员 `data_type` 分配的值指示对类型为 `struct kstat_named` 的所有对象都有效的联合成员。 对于 `data_type` 的值，定义了一组 C 预处理器标记。 例如，标记 `KSTAT_DATA_CHAR` 等于零，指示成员 `value.c` 为该值的当前存储位置。

示例 7–3 演示了如何通过跟踪用户进程访问 kstat_named.value 联合。 通过 kstat_data_lookup(3KSTAT) 函数，可从用户进程对 kstat 计数器进行采样，该函数将返回指向 struct kstat_named 的指针。 为获取最新的计数器采样值，mpstat(1M) 实用程序将在执行时重复调用该函数。 请转到您的 shell 并尝试运行 mpstat 1，然后观察输出。 几秒钟之后，在 shell 中按 Ctrl-C 组合键以中止 mpstat。 为观察计数器采样情况，我们要启用一个在 mpstat 命令每次调用 libkstat 中的 kstat_data_lookup(3KSTAT) 函数时都会触发的探测器。 为此，我们将使用一个新的 DTrace 提供器： pid。 pid 提供器允许您在用户进程的 C 符号位置（如函数入口点）动态创建探测器。 您可以通过编写以下格式的探测器说明，要求 pid 提供器在用户函数进入和返回位置创建探测器：

pid*process-ID*:*object-name*:*function-name*:entry

pid*process-ID*:*object-name*:*function-name*:return

例如，如果要在进程 ID 12345 中创建在进入 kstat_data_lookup(3KSTAT) 时触发的探测器，可编写以下探测器说明：

```
pid12345:libkstat:kstat_data_lookup:entry
```

pid 提供器将在指定用户进程与探测器说明对应的程序位置插入动态检测过程。该探测器实现强制到达受检测的程序位置的每个用户线程陷入操作系统内核并进入 DTrace，同时触发对应的探测器。因此，尽管实现位置与用户进程关联，但指定的 DTrace 谓词和操作仍在操作系统内核环境中执行。有关 pid 提供器的详细信息，请参阅第 30 章。

您可以在程序中插入称为宏变量的标识符，在编译程序时将计算该标识符，并将其替换为其他 dtrace 命令行参数，从而无需在每次要将程序应用于不同进程时，都对 D 程序源代码进行编辑。宏变量使用美元符号 $ 并后跟标识符或数字进行指定。如果您执行命令 dtrace -s *script* foo bar baz，D 编译器将自动定义宏变量 $1、$2 和 $3，分别作为 foo、bar 和 baz 的标记。您可以在 D 程序表达式或探测器说明中使用宏变量。例如，以下探测器说明检测指定为 dtrace 附加参数的任何进程 ID：

```
pid$1:libkstat:kstat_data_lookup:entry

{

    self->ksname = arg1;

}



pid$1:libkstat:kstat_data_lookup:return

/self->ksname != NULL && arg1 != NULL/

{
```

```
    this->ksp = (kstat_named_t *)copyin(arg1, sizeof (kstat_named_t));

    printf("%s has ui64 value %u\n", copyinstr(self->ksname),

        this->ksp->value.ui64);

}



pid$1:libkstat:kstat_data_lookup:return

/self->ksname != NULL && arg1 == NULL/

{

    self->ksname = NULL;

}
```

有关宏变量和可重用脚本的详细信息，请参阅第 15 章。至此，我们已了解如何使用进程 ID 检测用户进程，现在让我们返回到采样联合。请转到编辑器并键入完整示例的源代码，然后将其保存在名为 `kstat.d` 的文件中：

示例 7–3 `kstat.d`: 跟踪对 `kstat_data_lookup`(3KSTAT) 的调用

```
pid$1:libkstat:kstat_data_lookup:entry

{

    self->ksname = arg1;

}



pid$1:libkstat:kstat_data_lookup:return

/self->ksname != NULL && arg1 != NULL/

{

    this->ksp = (kstat_named_t *) copyin(arg1, sizeof (kstat_named_t));

    printf("%s has ui64 value %u\n",

        copyinstr(self->ksname), this->ksp->value.ui64);
```

示例 7–3 kstat.d: 跟踪对 kstat_data_lookup(3KSTAT) 的调用 （续）

```
}


pid$1:libkstat:kstat_data_lookup:return
/self->ksname != NULL && arg1 == NULL/
{
    self->ksname = NULL;
}
```

现在，转到其中一个 shell 并执行命令 `mpstat 1`，以启动在进行统计信息采样并每秒报告一次的模式下运行的 mpstat(1M)。开始运行 `mpstat` 后，在其他 shell 中执行命令 `dtrace -q -s kstat.d ‘pgrep mpstat‘`。您将会看到与正在访问的统计信息对应的输出。按 Ctrl-C 组合键中止 `dtrace`，然后返回到 shell 提示符。

```
# dtrace -q -s kstat.d ‘pgrep mpstat‘
cpu_ticks_idle has ui64 value 41154176
cpu_ticks_user has ui64 value 1137
cpu_ticks_kernel has ui64 value 12310
cpu_ticks_wait has ui64 value 903
hat_fault has ui64 value 0
as_fault has ui64 value 48053
maj_fault has ui64 value 1144
xcalls has ui64 value 123832170
intr has ui64 value 165264090
intrthread has ui64 value 124094974
pswitch has ui64 value 840625
inv_swtch has ui64 value 1484
```

```
cpumigrate has ui64 value 36284

mutex_adenters has ui64 value 35574

rw_rdfails has ui64 value 2

rw_wrfails has ui64 value 2

...

^C

#
```

如果您在每个终端窗口中捕获输出，并通过统计信息从前一次迭代所报告的值中减去每个值，则应可将 dtrace 输出与 mpstat 输出关联。该示例程序在进入查找函数时记录计数器名称指针，然后在从 kstat_data_lookup(3KSTAT) 返回时执行大多数跟踪操作。如果 arg1（返回值）不为 NULL，D 内置函数 copyinstr() 和 copyin() 则会将函数结果从用户进程复制到 DTrace 中。完成 kstat 数据复制后，该示例将报告联合的 ui64 计数器值。此简化示例假定 mpstat 对使用 value.ui64 成员的计数器进行采样。作为一种练习，请尝试记录 kstat.d，以使用多个谓词并列显示与 data_type 成员对应的联合成员。此外，还可以尝试创建用于计算连续数据值之差，以及实际生成与 mpstat 类似的输出的 kstat.d 版本。

成员大小和偏移

通过 sizeof 运算符，可以确定包括结构或联合在内的任何 D 类型或表达式的大小（以字节为单位）。 sizeof 运算符可应用于表达式或由括号括住的类型名称，如以下两个示例所示：

```
sizeof expression                    sizeof (type-name)
```

例如，如果将表达式 sizeof (uint64_t) 和 sizeof (callinfo.ts) 插入以上示例程序的源代码中，二者都将返回值 8。 sizeof 运算符的正式返回类型为类型别名 size_t，后者被定义为一个大小与当前数据模型中的指针相同的无符号整数，用于表示字节计数。 如果将 sizeof 运算符应用于表达式，D 编译器将会验证该表达式，但是，生成的对象大小在编译时进行计算，并且不会生成表达式代码。 您可以在需要整型常数的任何位置使用 sizeof。

您可以使用配套运算符 offsetof，来确定结构或联合成员相对于与结构或联合类型的任何变量关联的存储区起始位置的偏移量（以字节为单位）。 offsetof 运算符用于以下格式的运算符：

```
offsetof (type-name, member-name)
```

其中，*type-name* 是任何结构或联合类型的名称或类型别名，*member-name* 是命名该结构或联合的成员的标识符。`offsetof` 与 `sizeof` 类似，也返回 `size_t`，并且可在 D 程序中可使用整型常数的任何位置使用。

位字段

D 还允许定义任意位数的整型结构和联合成员（称作位字段）。通过指定带符号整数或无符号整数的基本类型、成员名以及指示要为字段分配的位数的后缀，可以声明位字段，如以下示例所示：

```
struct s {

    int a : 1;

    int b : 3;

    int c : 12;

};
```

位字段宽度是一个整型常数，通过结尾冒号与成员名分隔。位字段宽度必须为正数，并且其位数不得超过对应的整数基本类型的宽度。在 D 中，无法声明 64 位以上的位字段。D 位字段与对应的 ANSI-C 功能兼容，并提供对相应 ANSI-C 功能的访问。位字段通常用于非常需要内存存储空间，或结构布局必须与硬件寄存器布局匹配的情况。

位字段是一种编译器构造，该构造可自动设置整数布局和一组掩码，以提取成员值。您只需自己定义掩码并使用 & 运算符，即可获取同样的结果。C 编译器和 D 编译器将尽可能对位进行有效打包，但是，由于它们会按照所需顺序或方式随意打包，因此，无法保证位字段在不同编译器或体系结构间生成相同的位布局。如果您需要稳定的位布局，则应使用 & 运算符，亲自构造位掩码并提取值。

与任何其他结构或结构成员一样，只需指定位字段成员名并结合 "." 或 -> 运算符，便可访问位字段成员。位字段将自动提升为下一个最大的整数类型，以供所有表达式使用。由于位字段存储不能在字节边界上对齐，或大小不能为字节约数，因此，无法将 `sizeof` 或 `offsetof` 运算符应用于位字段成员。另外，D 编译器还禁止您使用 & 运算符获取位字段成员的地址。

◆ ◆ ◆ 第 8 章

8 类型和常量定义

本章介绍如何在 D 中声明类型别名和命名的常量。本章还将讨论程序和操作系统类型及标识符的 D 类型和名称空间管理。

Typedef

`typedef` 关键字用于将一个标识符声明为现有类型的别名。 与所有 D 类型声明一样，`typedef` 关键字在探测器子句外部的以下格式的声明中使用：

```
typedef existing-type new-type ;
```

其中，*existing-type* 是任何类型的声明，*new-type* 是要用作此类型别名的标识符。 例如，D 编译器在内部使用以下声明：

```
typedef unsigned char uint8_t;
```

创建 `uint8_t` 类型别名。 可以使用常规类型（例如变量类型、关联数组值类型或元组成员类型）的位置都可以使用类型别名。 您也可以将 `typedef` 与更详细的声明（如新的 `struct` 定义）组合在一起。

```
typedef struct foo {
    int x;
    int y;
} foo_t;
```

在此示例中，`struct foo` 定义为与其别名 `foo_t` 相同的类型。Solaris C 系统头通常使用后缀 `_t` 表示 `typedef` 别名。

枚举

在程序中为常量定义符号名称，会使将来对程序的维护变得简单明了。一种方法是定义枚举，即，将一组整数与一组称为枚举器的标识符进行关联，编译器可以识别这些枚举器并将其替换为相应的整数值。 使用如下所示的声明定义枚举：

```
enum colors {

    RED,

    GREEN,

    BLUE

};
```

枚举中的第一个枚举器 RED 被赋予的值为零，每个后续标识符被赋予下一个整数值。 您也可以通过在任何枚举器的后面加上等号和整形常数，来为该枚举器指定显式的整数值。

```
enum colors {

    RED = 7,

    GREEN = 9,

    BLUE

};
```

由于枚举器 BLUE 没有指定值，且前一个枚举器设置为 9，所以编译器将为其赋予值 10。定义枚举之后，D 程序中可以使用整形常数的任何位置都可以使用该枚举器。 此外，enum colors 还定义为与 int 等效的类型。 D 编译器允许在可以使用 int 的任何位置使用 enum 类型的变量，并允许将任何整数值赋给 enum 类型的变量。 如果不需要类型名称，也可以在声明中省略 enum 名称。

枚举器在程序的所有后续子句和声明中都可见，因此不能在多个枚举中定义相同的枚举器标识符。 但是，您可以在相同或不同的枚举中定义具有相同值的多个枚举器。您也可以将没有相应枚举器的整数赋给枚举类型的变量。

D 枚举语法与 ANSI-C 中相应的语法相同。D 还可以访问操作系统内核及其可装入的模块中定义的枚举，但这些枚举器在 D 程序中并不全局可见。 内核枚举器仅当与相应枚举类型的某个对象比较、并用作某个二元比较运算符的参数时可见。 例如，函数 uiomove(9F) 具有如下所定义的 enum uio_rw 类型的参数：

```
enum uio_rw { UIO_READ, UIO_WRITE };
```

枚举器 UIO_READ 和 UIO_WRITE 通常在 D 程序中不可见，但可以通过将一个枚举器与 enum uio_rw 类型的值进行比较，来将两种枚举器提升为全局可见，如以下示例子句中所示：

```
fbt::uiomove:entry

/args[2] == UIO_WRITE/

{

    ...

}
```

此示例通过将 args[2]（enum uio_rw 类型的变量）与枚举器 UIO_WRITE 进行比较，来跟踪写入请求对 uiomove(9F) 函数的调用。因为左边的参数为枚举类型，D 编译器将在尝试解析右边的标识符时，在该枚举中进行搜索。此功能可以避免 D 程序无意中与操作系统内核中定义的大枚举集合产生标识符名称冲突。

内置

也可以使用 inline 指令定义 D 命名的常量，该指令提供了一种更常规的方法来创建编译期间将替换为预定义的值或表达式的标识符。与 C 预处理程序提供的 #define 指令相比，内置指令是一种功能更强大的词法替换。内置指令使用以下格式的声明指定：

```
inline type name = expression ;
```

其中，*type* 是现有类型的类型声明，*name* 是先前未定义为内置或全局变量的任何有效 D 标识符，*expression* 是任何有效 D 表达式。在处理内置指令之后，D 编译器将使用程序源代码中已编译格式的 *expression* 替换 *name* 的每个后续实例。例如，以下 D 程序将跟踪字符串 "hello" 和整数值 123：

```
inline string hello = "hello";

inline int number = 100 + 23;



BEGIN

{

    trace(hello);

    trace(number);
```

```
}
```

在可以使用相应类型的全局变量的任何位置都可以使用内置名称。 如果在编译时内置表达式可以计算为整数或字符串常量，则还可以在要求常量表达式（如标量数组维度）的上下文中使用内置名称。

对内置表达式进行语法错误验证是计算指令过程的一部分。根据用于 D 赋值运算符 (=) 的相同规则，表达式结果类型必须与内置指令所定义的类型兼容。 内置表达式不可以引用内置标识符本身：不允许使用递归定义。

DTrace 软件包在系统目录 `/usr/lib/dtrace` 中安装大量的 D 源文件，这些源文件中包含可以在 D 程序中使用的内置指令。 例如，`signal.d` 库包括以下格式的指令：

```
inline int SIGHUP = 1;

inline int SIGINT = 2;

inline int SIGQUIT = 3;

...
```

这些内置定义使您可以访问当前 Solaris 信号名称集，如 `signal`(3HEAD) 中所述。 类似地，`errno.d` 库包含 C `errno` 常量的内置指令，如 `Intro`(2) 中所述。

缺省情况下，D 编辑器自动包括所提供的所有 D 库文件，以便可以在任何 D 程序中使用这些定义。

类型名称空间

本节讨论 D 名称空间以及与各种类型有关的名称空间问题。 在传统语言（如 ANSI-C）中，类型可见性由类型是嵌入在函数中还是嵌入在其他声明中来确定。 在 C 程序的外部范围中声明的类型与单个全局名称空间关联，这些类型在整个程序中可见。C 头文件中定义的类型通常包括在此外部范围中。与这些语言不同，D 可以访问多个外部范围中的类型。

D 是一种可以很方便地在软件栈的多个层（包括操作系统内核、关联的可装入内核模块集和系统上正在运行的用户进程）中进行动态观察的语言。单个 D 程序可以实例化探测器，以便从编译到独立二进制对象的多个内核模块或其他软件实体中收集数据。因此，在 DTrace 和 D 编译器可以使用的总体类型中，可能存在名称相同但定义不同的多种数据类型。 为了解决这种情况，D 编译器将每种类型与所包含程序对象标识的名称空间关联。 通过在任何类型名称中指定对象名称和反引号 (‘) 作用域运算符，可以访问特定程序对象的类型。

例如，如果名为 `foo` 的内核模块包含以下 C 类型声明：

```
typedef struct bar {

    int x;
```

```
} bar_t;
```

则可以在 D 中使用以下类型名称来访问 `struct bar` 和 `bar_t`：

```
struct foo‘bar                foo‘bar_t
```

在类型名称合适的任何上下文中（包括当为 D 探测子句中的 D 变量声明或强制类型转换表达式指定类型时）都可以使用反引号运算符。

D 编译器还提供了两种特殊的内置类型名称空间（分别使用 `C` 和 `D`）。C 类型名称空间最初使用标准 ANSI-C 内部类型（如 `int`）填充。此外，使用 C 预处理程序 `cpp(1)` 并使用 `dtrace -C` 选项获取的类型定义将由 C 范围处理并添加到其范围中。因此，可以将已可见的类型指令所在的 C 头文件包含到另一类型名称空间中，而不会引起编译错误。

D 类型名称空间初始使用 D 内部类型（如 `int` 和 `string`）及内置 D 类型别名（如 `uint32_t`）填充。出现在 D 程序源代码中的任何新类型声明都将自动添加到 D 类型名称空间中。如果在 D 程序中创建由来自其他名称空间的成员类型组成的复杂类型（如 `struct`），这些成员类型将会根据声明复制到 D 名称空间中。

当 D 编译器遇到未使用反引号运算符指定显式名称空间的类型声明时，编译器将会使用指定的类型名称搜索活动的类型名称空间集以找到匹配项。将始终首先搜索 C 名称空间，然后搜索 D 名称空间。如果在 C 或 D 名称空间中未找到类型名称，则将会按内核模块 ID 的升序搜索活动的内核模块的类型名称空间。此顺序保证，将首先搜索构成核心内核的二进制对象，然后搜索任何可装入的内核模块，但不能保证可装入模块间的任何顺序属性。访问可装入内核模块中定义的类型时，应使用作用域运算符以避免与其他内核模块发生类型名称冲突。

D 编译器使用为核心 Solaris 内核模块提供的压缩 ANSI-C 调试信息，以便自动访问与操作系统源代码关联的类型，而无需访问相应的 C 包含文件。此符号调试信息可能并不对系统上的所有内核模块都可用。如果尝试访问模块的名称空间内的类型，而该模块缺少要用于 DTrace 的压缩 C 调试信息，则 D 将会报告错误。

◆ ◆ ◆ 第 9 章

9 聚合

当检测系统以回答有关性能的问题时，可以考虑如何聚合数据以回答特定问题，而不是考虑根据单个探测器收集的数据获得答案。例如，如果要知道某用户 ID 进行的系统调用的数目，您不必关心每次系统调用收集的数据，只需要查看包含用户 ID 和系统调用的表即可。以前，您需要通过在每次系统调用时收集数据，并使用工具（如 awk(1) 或 perl(1)）对数据进行后期处理来回答此问题。但是在 DTrace 中，聚合数据非常容易。本章介绍用于处理聚合的 DTrace 工具。

聚合函数

聚合函数具有以下属性：

$$f(f(x_0)\ U\ f(x_1)\ U\ \ldots\ U\ f(x_n)) = f(x_0\ U\ x_1\ U\ \ldots\ U\ x_n)$$

其中，x_n 是一个包含任意数据的数据集。即，对整个数据集合的子集应用聚合函数，然后再次对结果应用该聚合函数，得出的结果与对整个数据集合本身应用该函数相同。例如，生成给定数据集之和的 SUM 函数。如果原始数据由 {2, 1, 2, 5, 4, 3, 6, 4, 2} 组成，则对整个集合应用 SUM 的结果将是 {29}。同样，对由前三个元素组成的子集应用 SUM 的结果将是 {5}，对由接下来的三个元素组成的子集应用 SUM 的结果将是 {12}，而对余下三个元素应用 SUM 的结果也是 {12}。SUM 是一个聚合函数，因为将其应用于这些结果的集合 {5, 12, 12} 与将 SUM 应用于原始数据生成的结果相同，即 {29}。

并非所有函数都为聚合函数。用于确定集合的中间元素的 MEDIAN 函数就是一个非聚合函数。（中间元素的定义是：集合中大于它的元素数目与小于它的元素数目相同。）MEDIAN 是通过对集合进行排序，然后选择中间元素获取的。现在返回到最初的原始数据，如果对由前三个元素组成的集合应用 MEDIAN，则结果为 {2}。（经过排序的集合为 {1, 2, 2}；{2} 是由中间元素组成的集合。）同样，对接下来的三个元素应用 MEDIAN 将生成 {4}，而对最后三个元素应用 MEDIAN 将生成 {4}。因此，对每个子集应用 MEDIAN 将生成集合 {2, 4, 4}。对此集合应用 MEDIAN 将生成结果 {4}。但是，对原始集合进行排序将生成 {1, 2, 2, 2, 3, 4, 4, 5, 6}。对此集合应用 MEDIAN 将生成 {3}。因为这些结果不匹配，所以 MEDIAN 不是聚合函数。

许多用于了解数据集的常见函数都是聚合函数。这些函数包括：用于计算集合中元素数目的函数、用于计算集合的最小值和最大值的函数以及用于对集合中的所有元素求和的函数。可通过用于计算集合中元素数目的函数和用于对集中的元素进行求和的函数，来确定集合的运算方法。

但是，一些有用的函数并非聚合函数。这些函数包括：用于计算集合的模（最常见元素）的函数、用于计算集合的中间元素值的函数以及用于计算集合的标准差的函数。

在跟踪数据时对数据应用聚合函数有许多优点：

- 无需存储整个数据集。每次将新元素添加到集合中时，只要该集合由当前中间结果和新元素组成，就将计算聚合函数。在计算新结果后，新元素可能会被废弃。此过程将成倍降低数据点需要的存储空间（通常非常大）。
- 数据收集不会导致异常可伸缩性问题。聚合函数允许在每个 *CPU* 中保存中间结果，而不是将中间结果保存在共享数据结构中。然后，DTrace 将对由每个 CPU 中的中间结果组成的集合应用聚合函数，以生成最终系统范围的结果。

聚合

DTrace 将聚合函数的结果存储在称为聚合的对象中。聚合结果使用类似于关联数组所用的表达式元组来建立索引。在 D 中，聚合的语法如下所示：

```
@name[ keys ] = aggfunc ( args );
```

其中，*name* 是聚合的名称，*keys* 是用逗号分隔的 D 表达式列表，*aggfunc* 是其中一个 DTrace 聚合函数，而 *args* 是用逗号分隔的适用于聚合函数的参数列表。聚合 *name* 是使用特殊字符 `@` 作为前缀的 D 标识符。D 程序中指定的所有聚合都为全局变量；没有线程局部聚合或子句局部聚合。聚合名称保存在与其他 D 全局变量不同的标识符名称空间中。请记住，如果重用名称，则 `a` 和 `@a` 不是同一变量。特殊聚合名称 `@` 可用于在简单 D 程序中命名匿名聚合。D 编译器将此名称视为聚合名称 `@_` 的别名。

下表中显示了 DTrace 聚合函数。大多数聚合函数仅接受表示新数据的单个参数。

表 9–1 DTrace 聚合函数

函数名	参数	结果
`count`	无	调用次数。
`sum`	标量表达式	所指定表达式的总计值。
`avg`	标量表达式	所指定表达式的算术平均值。
`min`	标量表达式	所指定表达式中的最小值。
`max`	标量表达式	所指定表达式中的最大值。

表 9–1 DTrace 聚合函数 （续）

函数名	参数	结果
lquantize	标量表达式、下限、上限、步长值	所指定表达式的值的线性频数分布（按指定范围确定大小）。递增最高存储桶中小于指定表达式的值。
quantize	标量表达式	所指定表达式的值的二次方频数分布。递增最高二次方存储桶中小于所指定表达式的值。

例如，要计算系统中 write(2) 系统调用数，可以使用 count() 聚合函数，并以提示性字符串作为关键字：

```
syscall::write:entry

{

    @counts["write system calls"] = count();

}
```

缺省情况下，当因为显式 END 操作或用户按 Ctrl-C 组合键而使进程终止时，dtrace 命令将列显聚合结果。以下示例输出显示了运行此命令、等待几秒钟，然后按 Ctrl-C 组合键的结果：

```
# dtrace -s writes.d

dtrace: script ’./writes.d’ matched 1 probe

^C



  write system calls                                              179

#
```

可以通过将 execname 变量用作聚合的关键字来按每个进程名称计算系统调用的数目：

```
syscall::write:entry

{

    @counts[execname] = count();

}
```

以下示例输出显示了运行此命令、等待几秒钟，然后按 Ctrl-C 组合键的结果：

```
# dtrace -s writesbycmd.d

dtrace: script ’./writesbycmd.d’ matched 1 probe

^C


  dtrace                                                            1

  cat                                                               4

  sed                                                               9

  head                                                              9

  grep                                                             14

  find                                                             15

  tail                                                             25

  mountd                                                           28

  expr                                                             72

  sh                                                              291

  tee                                                             814

  def.dir.flp                                                    1996

  make.bin                                                       2010

#
```

或者，您可能想要进一步查看按可执行文件的名称和文件描述符组织的输出内容。文件描述符是 `write`(2) 的第一个参数，所以下面示例使用由 `execname` 和 `arg0` 组成的关键字：

```
syscall::write:entry

{

    @counts[execname, arg0] = count();

}
```

运行此命令将会生成包含可执行文件的名称和文件描述符的表，如下例所示：

```
# dtrace -s writesbycmdfd.d

dtrace: script ’./writesbycmdfd.d’ matched 1 probe

^C


  cat                                                       1      58

  sed                                                       1      60

  grep                                                      1      89

  tee                                                       1     156

  tee                                                       3     156

  make.bin                                                  5     164

  acomp                                                     1     263

  macrogen                                                  4     286

  cg                                                        1     397

  acomp                                                     3     736

  make.bin                                                  1     880

  iropt                                                     4    1731

#
```

以下示例显示按进程名组织的用于写入的系统调用中花费的平均时间。此示例使用 `avg()` 聚合函数，并将用于求平均值的表达式指定为参数。该示例对系统调用中花费的挂钟时间求平均值。

```
syscall::write:entry

{

    self->ts = timestamp;

}
```

```
syscall::write:return
/self->ts/
{
    @time[execname] = avg(timestamp - self->ts);
    self->ts = 0;
}
```

以下示例输出显示了运行此命令、等待几秒钟，然后按 Ctrl-C 组合键的结果：

```
# dtrace -s writetime.d
dtrace: script ’./writetime.d’ matched 2 probes
^C

  iropt                                                31315
  acomp                                                37037
  make.bin                                             63736
  tee                                                  68702
  date                                                 84020
  sh                                                   91632
  dtrace                                              159200
  ctfmerge                                            321560
  install                                             343300
  mcs                                                 394400
  get                                                 413695
  ctfconvert                                          594400
  bringover                                          1332465
```

```
  tail                                                      1335260

#
```

平均值非常有用，但通常不能提供足够的详细信息来帮助您了解数据点的分布。 要更详细地了解分布情况，请使用 quantize() 聚合函数，如下例所示：

```
syscall::write:entry

{

    self->ts = timestamp;

}



syscall::write:return

/self->ts/

{

    @time[execname] = quantize(timestamp - self->ts);

    self->ts = 0;

}
```

因为每一行输出都会产生频数分布图，所以此脚本的输出实际上比之前的输出要长一些。以下示例显示了选择的样本输出：

```
  lint

           value  ------------- Distribution ------------- count

            8192 |                                         0

           16384 |                                         2

           32768 |                                         0

           65536 |@@@@@@@@@@@@@@@@@@@                      74

          131072 |@@@@@@@@@@@@@@                           59

          262144 |@@@                                      14
```

```
        524288 |                                         0


acomp

         value  ------------- Distribution ------------- count
          4096 |                                         0
          8192 |@@@@@@@@@@@@                             840
         16384 |@@@@@@@@@@@@                             750
         32768 |@@                                       165
         65536 |@@@@@@                                   460
        131072 |@@@@@@                                   446
        262144 |                                         16
        524288 |                                         0
       1048576 |                                         1
       2097152 |                                         0


iropt

         value  ------------- Distribution ------------- count
          4096 |                                         0
          8192 |@@@@@@@@@@@@@@@@@@@@@@@@@                4149
         16384 |@@@@@@@@@@                               1798
         32768 |@                                        332
         65536 |@                                        325
        131072 |@@                                       431
        262144 |                                         3
```

```
          524288 |                                         2
         1048576 |                                         1
         2097152 |                                         0
```

请注意，频数分布的行数始终是二次方值。每一行表示大于或等于对应值，但小于下一个更大行值的元素数目。例如，以上输出说明 `iropt` 在 8,192 纳秒和 16,383 纳秒之间（含 8,192 纳秒和 16,383 纳秒）进行了 4149 次写入操作。

虽然 `quantize()` 可用于快速了解数据，但您可能想要检查线性值的分布情况。要显示线性值的分布，请使用 `lquantize()` 聚合函数。除 D 表达式外，`lquantize()` 函数还接受三个参数：下限、上限和步长。例如，如果想要按文件描述符查看写入分布，则使用二次方量化可能不再有效。应改为使用范围较小的线性量化，如下例所示：

```
syscall::write:entry
{
    @fds[execname] = lquantize(arg0, 0, 100, 1);
}
```

运行此脚本几秒钟后将会生成大量信息。以下示例显示了一组典型输出：

```
  mountd

           value  ------------- Distribution ------------- count
              11 |                                         0
              12 |@                                        4
              13 |                                         0
              14 |@@@@@@@@@@@@@@@@@@@@@@@@@@@              70
              15 |                                         0
              16 |@@@@@@@@@@@@                             34
              17 |                                         0


  xemacs-20.4
```

```
           value  ------------- Distribution ------------- count
               6 |                                         0
               7 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  521
               8 |                                         0
               9 |                                         1
              10 |                                         0


make.bin
           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  3596
               2 |                                         0
               3 |                                         0
               4 |                                         42
               5 |                                         50
               6 |                                         0


acomp
           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |@@@@@                                    1156
               2 |                                         0
               3 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@    6635
               4 |@                                        297
```

```
              5 |                                         0


  iropt

           value  ------------- Distribution ------------- count

              2 |                                         0

              3 |                                         299

              4 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  20144

              5 |                                         0
```

还可使用 `lquantize()` 聚合函数来聚合自过去某个时间点以来的时间。 通过此方法，可以观察一段时间内行为的变化。 以下示例显示了执行 `date(1)` 命令的进程的生命周期中系统调用行为的变化：

```
syscall::exec:return,

syscall::exece:return

/execname == "date"/

{

    self->start = vtimestamp;

}



syscall:::entry

/self->start/

{

    /*

     * We linearly quantize on the current virtual time minus our

     * process's start time.  We divide by 1000 to yield microseconds

     * rather than nanoseconds.  The range runs from 0 to 10 milliseconds
```

```
     * in steps of 100 microseconds; we expect that no date(1) process
     * will take longer than 10 milliseconds to complete.
     */
    @a["system calls over time"] =
        lquantize((vtimestamp - self->start) / 1000, 0, 10000, 100);
}

syscall::rexit:entry
/self->start/
{
    self->start = 0;
}
```

执行许多 date(1) 进程时，上面的脚本有助于更好地了解系统调用行为。要查看此结果，请在一个窗口中运行 `sh -c 'while true; do date >/dev/null; done'`，同时在另一个窗口中执行 D 脚本。该脚本生成 date(1) 命令的系统调用行为的配置文件：

```
# dtrace -s dateprof.d
dtrace: script './dateprof.d' matched 218 probes
^C

  system calls over time
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@                                       20530
             100 |@@@@@@                                   48814
             200 |@@@                                      28119
```

```
 300 |@                                        14646
 400 |@@@@@                                    41237
 500 |                                         1259
 600 |                                         218
 700 |                                         116
 800 |@                                        12783
 900 |@@@                                      28133
1000 |                                         7897
1100 |@                                        14065
1200 |@@@                                      27549
1300 |@@@                                      25715
1400 |@@@@                                     35011
1500 |@@                                       16734
1600 |                                         498
1700 |                                         256
1800 |                                         369
1900 |                                         404
2000 |                                         320
2100 |                                         555
2200 |                                         54
2300 |                                         17
2400 |                                         5
2500 |                                         1
2600 |                                         7
```

```
            2700 |                                         0
```

通过此输出，可以大致了解与内核的必需服务相关的 date(1) 命令的不同阶段。为了更好地了解这些阶段，您可能需要了解何时进行哪些系统调用。 如果这样，可以更改 D 脚本来聚合变量 probefunc 而不是聚合常量字符串。

列显聚合

缺省情况下，多个聚合按照它们在 D 程序中的引入顺序显示。 可使用 printa() 函数列显聚合来覆盖此行为。printa() 函数还允许您使用格式字符串精确地设置聚合数据的格式，如第 12 章中所述。

如果在 D 程序中未使用 printa() 语句设置聚合的格式，则 dtrace 命令将对聚合数据进行快照，并在跟踪完成后使用缺省聚合格式立即列显结果。如果使用 printa() 语句设置指定聚合的格式，则将禁用缺省行为。通过将语句 printa(@*aggregation-name*) 添加到程序的 dtrace:::END 探测器子句中，可获取相同的结果。avg()、count()、min()、max() 和 sum() 聚合函数的缺省输出格式显示与每个元组的聚合值对应的十进制整数值。lquantize() 和 quantize() 聚合函数的缺省输出格式将显示结果的 ASCII 表。列显聚合元组时，就像已经对每个元组元素应用了 trace() 一样。

数据标准化

在聚合数据一段时间后，您可能需要将与某个常数因子有关的数据标准化。通过此方法，可以更容易地比较不相交的数据。 例如，在聚合系统调用时，您可能需要按每秒的速率而不是基于运行过程的绝对值来输出系统调用。DTrace normalize() 操作使您可以按此方式将数据标准化。normalize() 的参数包括聚合和标准化因子。聚合的输出显示除以标准化因子之后的每个值。

以下示例显示如何按系统调用聚合数据：

```
#pragma D option quiet



BEGIN

{

    /*

     * Get the start time, in nanoseconds.

     */
```

```
    start = timestamp;
}

syscall:::entry
{
    @func[execname] = count();
}

END
{
    /*
     * Normalize the aggregation based on the number of seconds we have
     * been running.  (There are 1,000,000,000 nanoseconds in one second.)
     */
    normalize(@func, (timestamp - start) / 1000000000);
}
```

运行以上脚本一小段时间将会在台式计算机上生成以下输出：

```
# dtrace -s ./normalize.d
 ^C
  syslogd                                                           0
  rpc.rusersd                                                       0
  utmpd                                                             0
  xbiff                                                             0
  in.routed                                                         1
```

```
sendmail                                                  2
echo                                                      2
FvwmAuto                                                  2
stty                                                      2
cut                                                       2
init                                                      2
pt_chmod                                                  3
picld                                                     3
utmp_update                                               3
httpd                                                     4
xclock                                                    5
basename                                                  6
tput                                                      6
sh                                                        7
tr                                                        7
arch                                                      9
expr                                                     10
uname                                                    11
mibiisa                                                  15
dirname                                                  18
dtrace                                                   40
ksh                                                      48
java                                                     58
xterm                                                   100
```

```
  nscd                                                              120

  fvwm2                                                             154

  prstat                                                            180

  perfbar                                                           188

  Xsun                                                             1309

  .netscape.bin                                                    3005
```

normalize() 对指定的聚合设置标准化因子，但此操作不会修改基础数据。这会导致使用 denormalize() 函数将数据取消标准化。denormalize() 仅接受聚合。 将取消标准化操作添加到前面的示例中会同时返回原始系统调用的计数和每秒的速率：

```
#pragma D option quiet



BEGIN

{

    start = timestamp;

}



syscall:::entry

{

    @func[execname] = count();

}



END

{

    this->seconds = (timestamp - start) / 1000000000;

    printf("Ran for %d seconds.\n", this->seconds);
```

```
    printf("Per-second rate:\n");
    normalize(@func, this->seconds);
    printa(@func);

    printf("\nRaw counts:\n");
    denormalize(@func);
    printa(@func);
}
```

运行以上脚本一小段时间将生成与以下示例类似的输出：

```
# dtrace -s ./denorm.d
^C
Ran for 14 seconds.
Per-second rate:

  syslogd                                                   0
  in.routed                                                 0
  xbiff                                                     1
  sendmail                                                  2
  elm                                                       2
  picld                                                     3
  httpd                                                     4
  xclock                                                    6
  FvwmAuto                                                  7
```

```
  mibiisa                                                  22
  dtrace                                                   42
  java                                                     55
  xterm                                                    75
  adeptedit                                               118
  nscd                                                    127
  prstat                                                  179
  perfbar                                                 184
  fvwm2                                                   296
  Xsun                                                    829

Raw counts:

  syslogd                                                   1
  in.routed                                                 4
  xbiff                                                    21
  sendmail                                                 30
  elm                                                      36
  picld                                                    43
  httpd                                                    56
  xclock                                                   91
  FvwmAuto                                                104
  mibiisa                                                 314
  dtrace                                                  592
```

```
  java                                                            774

  xterm                                                          1062

  adeptedit                                                      1665

  nscd                                                           1781

  prstat                                                         2506

  perfbar                                                        2581

  fvwm2                                                          4156

  Xsun                                                          11616
```

也可以重新标准化聚合。如果对同一聚合多次调用 `normalize()`，则标准化因子将成为最新调用中指定的因子。 以下示例列显了一段时间内每秒的速率 :

示例 9–1 `renormalize.d`: 重新标准化聚合

```
#pragma D option quiet



BEGIN

{

    start = timestamp;

}



syscall:::entry

{

    @func[execname] = count();

}



tick-10sec
```

示例 9–1 renormalize.d: 重新标准化聚合　　（续）

```
{
    normalize(@func, (timestamp - start) / 1000000000);
    printa(@func);
}
```

清除聚合

使用 DTrace 生成简单监视脚本时，可使用 clear() 函数定期清除聚合中的值。此函数仅接受聚合作为其参数。clear() 函数仅清除聚合的值；聚合的关键字将保留。因此，如果聚合中的某个关键字的关联值为零，则表示该关键字具有非零值，但后来作为 clear() 的一部分被设置为零。要同时废弃聚合的值和关键字，请使用 trunc()。有关详细信息，请参见第 130 页中的 “截断聚合”。

以下示例将 clear() 添加到示例 9–1 中：

```
#pragma D option quiet

BEGIN
{
    last = timestamp;
}

syscall:::entry
{
    @func[execname] = count();
}

tick-10sec
```

```
{

    normalize(@func, (timestamp - last) / 1000000000);

    printa(@func);

    clear(@func);

    last = timestamp;

}
```

虽然示例 9–1 显示了 dtrace 调用的生命周期中系统调用的速率，但上面的示例仅显示了最近 10 秒内的系统调用速率。

截断聚合

查看聚合结果时，您通常只需关心最前面的几个结果。无需关注与最高值之外的任何其他对象关联的关键字和值。您可能还希望废弃整个聚合结果，从而删除关键字和值。 DTrace trunc() 函数适用于这两种情况。

trunc() 的参数包括聚合和可选截断值。 如果没有截断值，trunc() 将同时废弃整个聚合的聚合值和聚合关键字。 如果存在截断值 *n*，则 trunc() 将废弃聚合值和聚合关键字，但与最高 *n* 个值关联的值和关键字除外。即，trunc(@foo, 10)将截断前 10 个值之后的名为 foo 的聚合，其中 trunc(@foo) 将废弃整个聚合。如果将 0 指定为截断值，将会废弃整个聚合。

要查看后 *n* 个值（而不是前 *n* 个值），请为 trunc() 指定负的截断值。 例如，trunc(@foo, -10) 将截断后 10 个值之前的名为 foo 的聚合。

以下示例增加了系统调用示例，以便仅显示 10 秒内最前面 10 个系统调用应用程序的每秒系统调用速率。

```
#pragma D option quiet



BEGIN

{

    last = timestamp;

}
```

```
syscall:::entry
{
    @func[execname] = count();
}


tick-10sec
{
    trunc(@func, 10);
    normalize(@func, (timestamp - last) / 1000000000);
    printa(@func);
    clear(@func);
    last = timestamp;
}
```

以下示例显示在轻负荷膝上型计算机上运行以上脚本的输出：

```
  FvwmAuto                                                        7
  telnet                                                         13
  ping                                                           14
  dtrace                                                         27
  xclock                                                         34
  MozillaFirebird-                                               63
  xterm                                                         133
  fvwm2                                                         146
  acroread                                                      168
  Xsun                                                          616
```

```
telnet                                                    4

FvwmAuto                                                  5

ping                                                     14

dtrace                                                   27

xclock                                                   35

fvwm2                                                    69

xterm                                                    70

acroread                                                164

MozillaFirebird-                                        491

Xsun                                                   1287
```

最小化删除

因为 DTrace 将一些聚合数据缓存在内核中，所以向聚合中添加新关键字时可能会出现空间不足的情况。在此情况下，数据将被删除并且计数器会递增，而 `dtrace` 将生成一条消息，指示进行了聚合删除操作。因为 DTrace 在用户级（可以动态增加空间）保持长期运行状态（由聚合的关键字和中间结果组成），所以这种情况很少发生。 如果在极少数情况下发生了聚合删除，则可以使用 `aggsize` 选项增加聚合缓冲区大小，以减少发生删除的可能性。也可以使用此选项将 DTrace 的内存使用量减至最少。 与任何大小选项一样，可以使用任何大小后缀指定 `aggsize`。 此缓冲区的调整大小策略由 `bufresize` 选项指示。有关缓冲的更多详细信息，请参见第 11 章。有关选项的更多详细信息，请参见第 16 章。

避免聚合删除的另一个方法是提高在用户级使用聚合数据的速率。 此速率缺省为每秒一次，并且可使用 `aggrate` 选项显式调整。 与任何速率选项一样，可以使用任何时间后缀指定 `aggrate`，但缺省为每秒的速率。 有关 `aggsize` 选项的更多详细信息，请参见第 16 章。

◆ ◆ ◆ 第 10 章

10 操作和子例程

您可以使用 D 函数调用（如 `trace()` 和 `printf()`）来调用由 DTrace 提供的两种不同类型的服务：操作，用于跟踪数据或修改 DTrace 外部状态；子例程，仅影响内部 DTrace 状态。本章定义操作和子例程，并说明它们的语法和语义。

操作

通过操作可将 DTrace 程序与 DTrace 外部的系统交互。大多数常见操作将数据记录到 DTrace 缓冲区。也有其他可用操作，例如停止当前进程，提高当前进程的特定信号或停止全部跟踪。这些操作中的一些具有破坏性，因为它们会更改系统，虽然这些操作具有可靠的定义。只有显式启用了破坏性的操作，才可以使用这些操作。缺省情况下，数据记录操作将数据记录到主体缓冲区。有关主体缓冲区和缓冲区策略的详细信息，请参见第 11 章。

缺省操作

子句可以包含任意数量的操作和变量处理。如果将子句保留为空，则会采用缺省操作。缺省操作将跟踪主体缓冲区已启用的探测器标识符 (enabled probe identifier, EPID)。EPID 使用特定谓词和操作标识特定探测器的特定启用。通过 EPID，DTrace 使用者可以确定引起该操作的探测器。实际上，只要跟踪数据，都必须为该数据附加 EPID，以便于使用者理解数据。这样，缺省操作将跟踪 EPID，而不会执行任何其他任务。

使用缺省操作可以使 `dtrace`(1M) 的用法很简单。例如，以下示例命令使用缺省操作启用 `TS` 分时调度模块中的所有探测器：

```
# dtrace -m TS
```

上面的命令可能生成与以下示例类似的输出：

```
# dtrace -m TS

dtrace: description 'TS' matched 80 probes
```

```
CPU     ID                    FUNCTION:NAME
  0  12077                 ts_trapret:entry
  0  12078                ts_trapret:return
  0  12069                   ts_sleep:entry
  0  12070                  ts_sleep:return
  0  12033                  ts_setrun:entry
  0  12034                 ts_setrun:return
  0  12081                  ts_wakeup:entry
  0  12082                 ts_wakeup:return
  0  12069                   ts_sleep:entry
  0  12070                  ts_sleep:return
  0  12033                  ts_setrun:entry
  0  12034                 ts_setrun:return
  0  12069                   ts_sleep:entry
  0  12070                  ts_sleep:return
  0  12033                  ts_setrun:entry
  0  12034                 ts_setrun:return
  0  12069                   ts_sleep:entry
  0  12070                  ts_sleep:return
  0  12023                  ts_update:entry
  0  12079              ts_update_list:entry
  0  12080             ts_update_list:return
  0  12079              ts_update_list:entry
...
```

数据记录操作

数据记录操作包含核心 DTrace 操作。缺省情况下，这些操作中的每一个都会将数据记录到主体缓冲区，但每个操作也可以用于将数据记录到随机缓冲区。有关主体缓冲区的更多详细信息，请参见第 11 章。有关随机缓冲区的更多详细信息，请参见第 13 章。本节中的说明仅是指定向缓冲区，表示如果操作执行 `speculate()`，数据将记录到主体缓冲区或随机缓冲区。

trace()

`void trace(`*expression*`)`

最基本的操作是 `trace()` 操作，此操作将 D 表达式用作其参数并跟踪结果到定向缓冲区。以下语句为 `trace()` 操作的示例：

```
trace(execname);

trace(curlwpsinfo->pr_pri);

trace(timestamp / 1000);

trace(‘lbolt);

trace("somehow managed to get here");
```

tracemem()

`void tracemem(`*address*`, size_t` *nbytes*`)`

`tracemem()` 操作将 D 表达式用作其第一个参数 *address*，将常量用作其第二个参数 *nbytes*。`tracemem()` 从 *addr* 指定的地址复制 *nbytes* 指定长度的内存到定向缓冲区中。

printf()

`void printf(string` *format*`, ...)`

与 `trace()` 一样，`printf()` 操作将跟踪 D 表达式。但是，`printf()` 允许详细设置 `printf`(3C) 样式的格式。与 `printf`(3C) 一样，参数包含 *format* 字符串，后跟任意数量的参数。缺省情况下，将跟踪参数到定向缓冲区。然后，`dtrace`(1M) 将根据指定的格式字符串对参数进行格式化，以便输出。例如，可以将第 135 页中的 “trace()”中的前两个 `trace()` 示例组合到一个 `printf()` 中：

```
printf("execname is %s; priority is %d", execname, curlwpsinfo->pr_pri);
```

有关 printf() 的更多信息，请参见第 12 章。

printa()

void printa(*aggregation*)

void printa(string *format*, *aggregation*)

使用 printa() 操作可以显示和格式化聚合。有关聚合的更多详细信息，请参见第 9 章。如果未提供 *format*，printa() 将仅跟踪指示使用缺省格式处理和显示指定聚合的 DTrace 使用者指令。如果提供了 *format*，将按照指定的格式对聚合进行格式设置。有关 printa() 格式字符串的更详细说明，请参见第 12 章。

printa() 仅跟踪指示应由 DTrace 使用者处理聚合的指令。它不会处理内核中的聚合。因此，跟踪 printa() 指令和实际处理指令之间的时间取决于影响缓冲区处理的因素。这些因素包括聚合速率、缓冲策略，以及切换缓冲区的速率（如果缓冲策略为 switching）。有关这些因素的详细说明，请参见第 9 章和第 11 章。

stack()

void stack(int *nframes*)

void stack(void)

stack() 操作会将内核栈跟踪记录到定向缓冲区。内核栈的深度将为 *nframes*。如果未提供 *nframes*，则记录的栈帧的数目即为 stackframes 选项指定的数目。例如：

```
# dtrace -n uiomove:entry'{stack()}'

  CPU     ID                    FUNCTION:NAME

    0   9153                    uiomove:entry

                genunix‘fop_write+0x1b

                namefs‘nm_write+0x1d

                genunix‘fop_write+0x1b

                genunix‘write+0x1f7



    0   9153                    uiomove:entry
```

```
              genunix‘fop_read+0x1b
              genunix‘read+0x1d4


    0   9153                    uiomove:entry
              genunix‘strread+0x394
              specfs‘spec_read+0x65
              genunix‘fop_read+0x1b
              genunix‘read+0x1d4
   ...
```

stack() 操作与其他操作有些不同，该操作还可以用作聚合的关键字：

```
# dtrace -n kmem_alloc:entry’{@[stack()] = count()}’
dtrace: description ’kmem_alloc:entry’ matched 1 probe
^C


              rpcmod‘endpnt_get+0x47c
              rpcmod‘clnt_clts_kcallit_addr+0x26f
              rpcmod‘clnt_clts_kcallit+0x22
              nfs‘rfscall+0x350
              nfs‘rfs2call+0x60
              nfs‘nfs_getattr_otw+0x9e
              nfs‘nfsgetattr+0x26
              nfs‘nfs_getattr+0xb8
              genunix‘fop_getattr+0x18
              genunix‘cstat64+0x30
```

```
genunix‘cstatat64+0x4a

genunix‘lstat64+0x1c

  1


genunix‘vfs_rlock_wait+0xc

genunix‘lookuppnvp+0x19d

genunix‘lookuppnat+0xe7

genunix‘lookupnameat+0x87

genunix‘lookupname+0x19

genunix‘chdir+0x18

  1


rpcmod‘endpnt_get+0x6b1

rpcmod‘clnt_clts_kcallit_addr+0x26f

rpcmod‘clnt_clts_kcallit+0x22

nfs‘rfscall+0x350

nfs‘rfs2call+0x60

nfs‘nfs_getattr_otw+0x9e

nfs‘nfsgetattr+0x26

nfs‘nfs_getattr+0xb8

genunix‘fop_getattr+0x18

genunix‘cstat64+0x30

genunix‘cstatat64+0x4a

genunix‘lstat64+0x1c
```

```
                1

    ...
```

ustack()

```
void ustack(int nframes, int strsize)

void ustack(int nframes)

void ustack(void)
```

ustack() 操作将用户栈跟踪记录到定向缓冲区。用户栈的深度将为 *nframes*。 如果未提供 *nframes*，则记录的栈帧的数目即为 ustackframes 选项指定的数目。 虽然触发探测器时，ustack() 能够确定调用帧的地址，但 DTrace 使用者在用户级处理 ustack() 操作之前，栈帧将不会转换为符号。如果指定了 *strsize* 且它不为零，则 ustack() 将分配指定的字符串空间量，并使用该空间直接从内核中执行地址到符号的转换。这种直接的用户符号转换当前仅在版本 1.5 和更高版本的 Java 虚拟机中可用。 Java 地址到符号转换使用 Java 类和方法名注释包含 Java 帧的用户栈。如果无法转换这类帧，帧将仅显示为十六进制地址。

以下示例跟踪不包含字符串空间的栈，因此将不会进行 Java 地址到符号的转换：

```
# dtrace -n syscall::write:entry'/pid == $target/{ustack(50, 0);

    exit(0)}' -c "java -version"

dtrace: description 'syscall::write:entry' matched 1 probe

java version "1.5.0-beta3"

Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0-beta3-b58)

Java HotSpot(TM) Client VM (build 1.5.0-beta3-b58, mixed mode)

dtrace: pid 5312 has exited

CPU     ID                    FUNCTION:NAME

  0     35                      write:entry

              libc.so.1`_write+0x15

              libjvm.so`__1cDhpiFwrite6FipkvI_I_+0xa8

              libjvm.so`JVM_Write+0x2f

              d0c5c946
```

```
libjava.so‘Java_java_io_FileOutputStream_writeBytes+0x2c
cb007fcd
cb002a7b
cb002a7b
cb002a7b
cb002a7b
cb002a7b
cb002a7b
cb002a7b
cb002a7b
cb002a7b
cb002a7b
cb002a7b
cb002a7b
cb002a7b
cb002a7b
cb000152
libjvm.so‘__1cJJavaCallsLcall_helper6FpnJJavaValue_
            pnMmethodHandle_pnRJavaCallArguments_
            pnGThread__v_+0x187
libjvm.so‘__1cCosUos_exception_wrapper6FpFpnJJavaValue_
            pnMmethodHandle_pnRJavaCallArguments_
            pnGThread__v2468_v_+0x14
libjvm.so‘__1cJJavaCallsEcall6FpnJJavaValue_nMmethodHandle_
            pnRJavaCallArguments_pnGThread __v_+0x28
```

```
              libjvm.so‘__1cRjni_invoke_static6FpnHJNIEnv__pnJJavaValue_

                          pnI_jobject_nLJNICallType_pnK_jmethodID_pnSJNI_

                          ArgumentPusher_pnGThread__v_+0x180

              libjvm.so‘jni_CallStaticVoidMethod+0x10f

              java‘main+0x53d
```

请注意，Java 虚拟机中的 C 和 C++ 栈帧使用 C++ 的“损坏”符号名称以符号形式显示，Java 栈帧仅显示为十六进制地址。以下示例说明了对包含非零字符串空间的 `ustack()` 的调用：

```
# dtrace -n syscall::write:entry’/pid == $target/{ustack(50, 500); exit(0)}’

      -c "java -version"

dtrace: description ’syscall::write:entry’ matched 1 probe

java version "1.5.0-beta3"

Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0-beta3-b58)

Java HotSpot(TM) Client VM (build 1.5.0-beta3-b58, mixed mode)

dtrace: pid 5308 has exited

CPU     ID                    FUNCTION:NAME

  0     35                      write:entry

              libc.so.1‘_write+0x15

              libjvm.so‘__1cDhpiFwrite6FipkvI_I_+0xa8

              libjvm.so‘JVM_Write+0x2f

              d0c5c946

              libjava.so‘Java_java_io_FileOutputStream_writeBytes+0x2c

              java/io/FileOutputStream.writeBytes

              java/io/FileOutputStream.write

              java/io/BufferedOutputStream.flushBuffer
```

```
java/io/BufferedOutputStream.flush

java/io/PrintStream.write

sun/nio/cs/StreamEncoder$CharsetSE.writeBytes

sun/nio/cs/StreamEncoder$CharsetSE.implFlushBuffer

sun/nio/cs/StreamEncoder.flushBuffer

java/io/OutputStreamWriter.flushBuffer

java/io/PrintStream.write

java/io/PrintStream.print

java/io/PrintStream.println

sun/misc/Version.print

sun/misc/Version.print

StubRoutines (1)

libjvm.so‘__1cJJavaCallsLcall_helper6FpnJJavaValue_

            pnMmethodHandle_pnRJavaCallArguments_pnGThread

            __v_+0x187

libjvm.so‘__1cCosUos_exception_wrapper6FpFpnJJavaValue_

            pnMmethodHandle_pnRJavaCallArguments_pnGThread

            __v2468_v_+0x14

libjvm.so‘__1cJJavaCallsEcall6FpnJJavaValue_nMmethodHandle

            _pnRJavaCallArguments_pnGThread__v_+0x28

libjvm.so‘__1cRjni_invoke_static6FpnHJNIEnv__pnJJavaValue_pnI

            _jobject_nLJNICallType_pnK_jmethodID_pnSJNI

            _ArgumentPusher_pnGThread__v_+0x180

libjvm.so‘jni_CallStaticVoidMethod+0x10f
```

```
              java‘main+0x53d

              8051b9a
```

以上示例的输出演示了 Java 栈帧的符号栈帧信息。该输出中仍然有一些十六进制帧，这是因为一些函数为静态函数，并在应用程序符号表中没有条目。无法对这些帧进行转换。

非 Java 帧的 `ustack()` 符号转换发生在记录栈数据之后。因此在执行符号转换之前，对应的用户进程可能会退出，这将使栈帧转换无法进行。如果用户进程在执行符号转换之前退出，则 `dtrace` 将发出一条警告消息，后跟十六进制栈帧，如下例所示：

```
  dtrace: failed to grab process 100941: no such process

                c7b834d4

                c7bca85d

                c7bca1a4

                c7bd4374

                c7bc2628

                8047efc
```

第 33 章中介绍了减小此问题影响的方法。

最后，由于事后 DTrace 调试器命令无法执行帧转换，因此将 `ustack()` 与 `ring` 缓冲区策略同时使用总是会生成原始 `ustack()` 数据。

以下 D 程序显示了未指定 *strsize* 的 `ustack()` 示例：

```
syscall::brk:entry

/execname == $$1/

{

    @[ustack(40)] = count();

}
```

要在 Netscape Web 浏览器 `.netscape.bin`（缺省 Solaris 安装）中运行此示例，请使用以下命令：

```
# dtrace -s brk.d .netscape.bin
dtrace: description 'syscall::brk:entry' matched 1 probe
^C

                libc.so.1`_brk_unlocked+0xc
                88143f6
                88146cd
                .netscape.bin`unlocked_malloc+0x3e
                .netscape.bin`unlocked_calloc+0x22
                .netscape.bin`calloc+0x26
                .netscape.bin`_IMGCB_NewPixmap+0x149
                .netscape.bin`il_size+0x2f7
                .netscape.bin`il_jpeg_write+0xde
                8440c19
                .netscape.bin`il_first_write+0x16b
                8394670
                83928e5
                .netscape.bin`NET_ProcessHTTP+0xa6
                .netscape.bin`NET_ProcessNet+0x49a
                827b323
                libXt.so.4`XtAppProcessEvent+0x38f
                .netscape.bin`fe_EventLoop+0x190
                .netscape.bin`main+0x1875
                   1
```

```
                libc.so.1‘_brk_unlocked+0xc
                libc.so.1‘sbrk+0x29
                88143df
                88146cd
                .netscape.bin‘unlocked_malloc+0x3e
                .netscape.bin‘unlocked_calloc+0x22
                .netscape.bin‘calloc+0x26
                .netscape.bin‘_IMGCB_NewPixmap+0x149
                .netscape.bin‘il_size+0x2f7
                .netscape.bin‘il_jpeg_write+0xde
                8440c19
                .netscape.bin‘il_first_write+0x16b
                8394670
                83928e5
                .netscape.bin‘NET_ProcessHTTP+0xa6
                .netscape.bin‘NET_ProcessNet+0x49a
                827b323
                libXt.so.4‘XtAppProcessEvent+0x38f
                .netscape.bin‘fe_EventLoop+0x190
                .netscape.bin‘main+0x1875
                  1
...
```

jstack()

```
void jstack(int nframes, int strsize)

void jstack(int nframes)

void jstack(void)
```

jstack() 是 ustack() 的别名，它为栈帧的数量使用 jstackframes 选项指定的值，为字符串空间大小使用 jstackstrsize 选项指定的值。缺省情况下，jstacksize 缺省为非零值。因此，使用 jstack() 将导致栈在原位置进行 Java 帧转换。

破坏性操作

一些 DTrace 操作具有破坏性，因为它们会以某种明确定义的方式更改系统的状态。如果未显式启用，则无法使用这些破坏性操作。使用 dtrace(1M) 时，可以使用 -w 选项启用破坏性操作。如果尝试在 dtrace(1M) 中启用破坏性操作，但不显式启用这些操作，则 dtrace 将失败，并显示一条与以下示例类似的消息：

```
dtrace: failed to enable 'syscall': destructive actions not allowed
```

处理破坏性操作

一些破坏性操作仅对特定进程具有破坏性。具有 dtrace_proc 或 dtrace_user 权限的用户可以使用这些操作。有关 DTrace 安全权限的详细信息，请参见第 35 章。

stop()

```
void stop(void)
```

当触发已启用的探测器的进程接下来退出内核时，stop() 操作将强制停止该进程，就像由 proc(4) 操作停止一样。prun(1) 实用程序可用于恢复由 stop() 操作停止的进程。stop() 操作可用于在任何 DTrace 探测器位置停止进程。此操作可用于在特定状态下，捕获难以使用简单断点获取的程序，然后将一个传统调试器（如 mdb(1)）附加至该进程。您也可以使用 gcore(1) 实用程序将已停止进程的状态保存在一个核心转储文件中，供以后进行分析。

raise()

```
void raise(int signal)
```

raise() 操作将指定的信号发送至当前正在运行的进程。此操作与使用 kill(1) 命令向进程发送信号类似。raise() 操作可用于在进程执行过程中的一个准确时间点发送信号。

copyout()

```
void copyout(void *buf, uintptr_t addr, size_t nbytes)
```

copyout() 操作在与当前线程关联的进程的地址空间中，从 *buf* 指定的缓冲区中复制 *nbytes* 到 *addr* 指定的地址。 如果用户空间地址与当前地址空间中有效的错误页面不对应，则将生成错误。

copyoutstr()

```
void copyoutstr(string str, uintptr_t addr, size_t maxlen)
```

copyoutstr() 操作在与当前线程关联的进程的地址空间中，将 *str* 指定的字符串复制到 *addr* 指定的地址。 如果用户空间地址与当前地址空间中有效的错误页面不对应，则将生成错误。 字符串长度限于 strsize 选项指定的值。有关详细信息，请参见第 16 章。

system()

```
void system(string program, ...)
```

system() 操作将导致执行 *program* 指定的程序，就像将该程序作为输入提供给 shell 一样。 *program* 字符串可以包含任何 printf()/printa() 格式转换。指定的参数必须与格式转换匹配。 有关有效的格式转换的详细信息，请参阅第 12 章。

以下示例每秒运行 date(1) 命令一次：

```
# dtrace -wqn tick-1sec'{system("date")}'

 Tue Jul 20 11:56:26 CDT 2004

 Tue Jul 20 11:56:27 CDT 2004

 Tue Jul 20 11:56:28 CDT 2004

 Tue Jul 20 11:56:29 CDT 2004

 Tue Jul 20 11:56:30 CDT 2004
```

以下示例通过在 *program* 字符串中使用 printf() 转换，同时使用传统过滤工具（如管道），更详细地说明了该操作的用法。

```
#pragma D option destructive

#pragma D option quiet



proc:::signal-send

/args[2] == SIGINT/
```

```
{

    printf("SIGINT sent to %s by ", args[1]->pr_fname);

    system("getent passwd %d | cut -d: -f5", uid);

}
```

运行以上脚本将会生成与以下示例类似的输出：

```
# ./whosend.d

SIGINT sent to MozillaFirebird- by Bryan Cantrill

SIGINT sent to run-mozilla.sh by Bryan Cantrill

^C

SIGINT sent to dtrace by Bryan Cantrill
```

在触发探测器的上下文中不会执行指定的命令，此命令是在用户级处理包含 `system()` 操作的详细信息的缓冲区时执行的。 进行此处理的方式和时间取决于第 11 章中介绍的缓冲策略。 使用缺省缓冲策略时，缓冲区处理速率由 `switchrate` 选项指定。 如果将 `switchrate` 显式地调整到高于其缺省值（一秒），则将会看到 `system()` 中的固有延迟，如下例所示：

```
#pragma D option quiet

#pragma D option destructive

#pragma D option switchrate=5sec



tick-1sec

/n++ < 5/

{

    printf("walltime  : %Y\n", walltimestamp);

    printf("date      : ");

    system("date");

    printf("\n");
```

```
}


tick-1sec

/n == 5/

{

    exit(0);

}
```

运行以上脚本将会生成与以下示例类似的输出：

```
# dtrace -s ./time.d

 walltime  : 2004 Jul 20 13:26:30

date      : Tue Jul 20 13:26:35 CDT 2004


walltime  : 2004 Jul 20 13:26:31

date      : Tue Jul 20 13:26:35 CDT 2004


walltime  : 2004 Jul 20 13:26:32

date      : Tue Jul 20 13:26:35 CDT 2004


walltime  : 2004 Jul 20 13:26:33

date      : Tue Jul 20 13:26:35 CDT 2004


walltime  : 2004 Jul 20 13:26:34

date      : Tue Jul 20 13:26:35 CDT 2004
```

请注意，`walltime` 值会不同，但 `date` 值相同。此结果反映了一个事实，即仅当处理缓冲区而不是记录 `system()` 操作时，才会执行 `date(1)` 命令。

内核破坏性操作

一些破坏性操作对整个系统都具有破坏性。由于这些操作将会影响系统中的每个进程，以及任何其他隐式或显式依赖于受影响系统的网络服务的系统，因此使用这些操作时必须非常小心。

breakpoint()

```
void breakpoint(void)
```

`breakpoint()` 操作会引起内核断点，从而使系统停止并将控制转移到内核调试器。内核调试器将发出一个字符串，表示触发该操作的 DTrace 探测器。例如，如果要执行以下操作：

```
# dtrace -w -n clock:entry'{breakpoint()}'

dtrace: allowing destructive actions

dtrace: description 'clock:entry' matched 1 probe
```

在 SPARC 上运行的 Solaris 中，可能会在控制台上显示以下消息：

```
dtrace: breakpoint action at probe fbt:genunix:clock:entry (ecb 30002765700)

Type  'go' to resume

ok
```

在 x86 上运行的 Solaris 中，可能会在控制台上显示以下消息：

```
dtrace: breakpoint action at probe fbt:genunix:clock:entry (ecb d2b97060)

stopped at      int20+0xb:      ret

kmdb[0]:
```

探测器说明后面的地址是 DTrace 中启用控制块 (enabling control block, ECB) 的地址。您可以使用此地址确定有关引起断点操作的探测器启用的更多详细信息。

`breakpoint()` 操作中的错误可能会导致调用此操作的次数远远多于预期情况。此行为可能反过来阻止您终止正在触发断点操作的 DTrace 使用者。在此情况下，请将内核整数变量 `dtrace_destructive_disallow` 设置为 1。此设置将禁止计算机上的所有破坏性操作。仅在此特定情况下才应用此设置。

设置 `dtrace_destructive_disallow` 的准确方法将取决于您所使用的内核调试器。如果在 SPARC 系统上使用 OpenBoot PROM，请使用 `w!`：

```
ok 1 dtrace_destructive_disallow w!

ok
```

确认是否已使用 w? 设置了该变量：

```
ok dtrace_destructive_disallow w?

1

ok
```

键入 go 以继续：

```
ok go
```

如果在 x86 或 SPARC 系统上使用 kmdb(1)，请将 4 字节的写入修饰符 (W) 与 / 格式设置 dcmd 一起使用：

```
kmdb[0]: dtrace_destructive_disallow/W 1

dtrace_destructive_disallow:    0x0             =       0x1

kmdb[0]:
```

继续使用 :c：

```
kadb[0]: :c
```

要在继续之后重新启用破坏性操作，需要使用 mdb(1) 将 dtrace_destructive_disallow 显式重置为 0：

```
# echo "dtrace_destructive_disallow/W 0" | mdb -kw

dtrace_destructive_disallow:    0x1             =       0x0

#
```

panic()

```
void panic(void)
```

触发panic() 操作将会导致内核出现紧急情况。 此操作应该用于在重要时间强制进行系统崩溃转储。您可以将此操作与环形缓冲和事后分析结合使用来理解问题。有关更多信息，请分别参见第 11 章和第 37 章。使用紧急情况操作时，将会显示紧急情况消息，表示导致该紧急情况的探测器。例如：

```
panic[cpu0]/thread=30001830b80: dtrace: panic action at probe
syscall::mmap:entry (ecb 300000acfc8)

000002a10050b840 dtrace:dtrace_probe+518 (fffe, 0, 1830f88, 1830f88,
  30002fb8040, 300000acfc8)
  %l0-3: 0000000000000000 00000300030e4d80 0000030003418000 00000300018c0800
  %l4-7: 000002a10050b980 0000000000000500 0000000000000000 0000000000000502
000002a10050ba30 genunix:dtrace_systrace_syscall32+44 (0, 2000, 5,
  80000002, 3, 1898400)
  %l0-3: 00000300030de730 0000000002200008 00000000000000e0 000000000184d928
  %l4-7: 00000300030de000 0000000000000730 0000000000000073 0000000000000010

syncing file systems... 2 done
dumping to /dev/dsk/c0t0d0s1, offset 214827008, content: kernel
100% done: 11837 pages dumped, compression ratio 4.66, dump
succeeded
rebooting...
```

`syslogd`(1M) 还将在重新引导时发出一条消息：

```
Jun 10 16:56:31 machine1 savecore: [ID 570001 auth.error] reboot after panic:
dtrace: panic action at probe syscall::mmap:entry (ecb 300000acfc8)
```

崩溃转储的消息缓冲区还包含负责 `panic()` 操作的探测器和 ECB。

chill()

`void chill(int` *nanoseconds*`)`

chill() 操作将导致 DTrace 显示指定的纳秒数。chill() 主要用于查找可能与计时相关的问题。例如，您可以使用此操作打开竞争情况窗口，或者使定期事件相互之间同步或异步。由于处于 DTrace 探测器上下文中时禁止中断，因此使用任何 chill() 都将引起中断延迟、调度延迟和分发延迟。因此，chill() 可能会导致意外的系统影响，不应毫无限制地使用。由于系统活动依赖于定期的中断处理，如果在任何给定 CPU 的每一秒间隔内，chill() 操作需要的时间超过 500 毫秒，DTrace 将拒绝执行。如果超过最大 chill() 间隔，DTrace 将报告非法的操作错误，如下例所示：

```
# dtrace -w -n syscall::open:entry'{chill(500000001)}'

dtrace: allowing destructive actions

dtrace: description 'syscall::open:entry' matched 1 probe

dtrace: 57 errors

CPU     ID                    FUNCTION:NAME

dtrace: error on enabled probe ID 1 (ID 14: syscall::open:entry): \

  illegal operation in action #1
```

即使时间跨越多个 chill() 调用或者单个探测器的多个 DTrace 使用者，仍将会强制执行此限制。例如，以下命令将会生成相同的错误：

```
# dtrace -w -n syscall::open:entry'{chill(250000000); chill(250000001);}'
```

特殊操作

本节说明的操作既不是数据记录操作，也不是破坏性操作。

随机操作

与随机跟踪关联的操作为 speculate()、commit() 和 discard()。这些操作将在第 13 章中讨论。

exit()

void exit(int *status*)

exit() 操作用于立即停止跟踪，以及通知 DTrace 使用者应停止跟踪、执行任何最终处理并使用指定的状态调用 exit(3C)。因为 exit() 将状态返回至用户级，所以它是数据记录操作，但与其他数据存储操作不同，不能随机跟踪 exit()。无论缓冲策略是什么，exit() 都将导致 DTrace 使用者退出。由于 exit() 是数据记录操作，因此可以删除该操作。

调用 exit() 时，仅会完成其他 CPU 中已在进行中的 DTrace 操作。在任何 CPU 中都不会进行新的操作。对于此规则，唯一的例外是 END 探测器的处理，该探测器在 DTrace 使用者已处理了 exit() 操作，并指示应停止跟踪时调用。

子例程

子例程与操作不同，因为子例程通常仅会影响 DTrace 的内部状态。因此，没有破坏性的子例程，并且子例程从不会跟踪数据到缓冲区中。在 9F 或 3C 一节的接口中，许多子例程都有相应的模拟程序。有关相应子例程的更多信息，请参见 Intro(9F) 和 Intro(3)。

alloca()

```
void *alloca(size_t size)
```

alloca() 在临时空间外部分配 *size* 个字节，并返回指向已分配内存的指针。返回的指针一定是每 8 个字节对齐的。临时空间仅在子句的持续时间中有效。使用 alloca() 分配的内存在子句完成时将解除分配。如果没有可用的足够临时空间，将不会分配内存，并且将生成错误。

basename()

```
string basename(char *str)
```

basename() 是 basename(1) 的 D 类似程序。此子例程创建一个包含所指定字符串的副本的字符串，但该字符串不包含以 / 结尾的任何前缀。返回的字符串分配到临时内存外部，因此它们仅在子句的持续时间中有效。如果没有可用的足够临时空间，则不会执行 basename，并且将生成错误。

bcopy()

```
void bcopy(void *src, void *dest, size_t size)
```

bcopy() 从 *src* 指向的内存复制 *size* 字节到 *dest* 指向的内存。所有源内存都必须位于临时内存外部，所有目标内存都必须位于临时内存中。如果不满足这些条件，则不会进行复制操作，并且将生成错误。

cleanpath()

```
string cleanpath(char *str)
```

cleanpath() 创建包含由 *str* 表示的路径副本的字符串，但该字符串删除了某些多余元素。需要特别指出的是，路径中的 "/./" 元素已删除，并且 "/../" 元素已折叠。折叠路径中的 /../ 元素时将不考虑符号链接。因此，cleanpath() 可能会采用有效路径，但返回更短的无效路径。

例如，如果 *str* 为 "/foo/../bar" 且 /foo 为指向 /net/foo/export 的符号链接，则 cleanpath() 将返回字符串 "/bar"，即使 bar 可能仅位于 /net/foo 中而不是 / 中。作出此限制是因为，cleanpath() 在触发探测器的上下文中调用，在此情况下不能进行完整的符号链接解析或使用任意名称。返回的字符串分配到临时内存外部，因此它们仅在子句的持续时间中有效。如果没有可用的足够临时空间，则不会执行 cleanpath，并且将生成错误。

copyin()

void *copyin(uintptr_t *addr*, size_t *size*)

copyin() 从指定的用户地址中复制指定的字节大小到 DTrace 临时缓冲区中，并返回此缓冲区的地址。用户地址解释为与当前线程关联的进程空间中的地址。生成的缓冲区指针一定是每 8 个字节对齐的。有问题的地址必须与当前进程中的错误页面对应。如果地址与错误页面不对应，或者没有可用的足够临时空间，则会返回 NULL，并且将生成错误。有关降低出现 copyin 错误可能性的方法，请参见第 33 章。

copyinstr()

string copyinstr(uintptr_t *addr*)

copyinstr() 从指定的用户地址中复制以 null 结尾的 C 字符串到 DTrace 临时缓冲区中，并返回此缓冲区的地址。用户地址解释为与当前线程关联的进程空间中的地址。字符串长度限于由 strsize 选项指定的值；有关详细信息，请参见第 16 章。与 copyin 一样，指定的地址必须与当前进程中的错误页面对应。如果地址与错误页面不对应，或者没有可用的足够临时空间，则会返回 NULL，并且将生成错误。有关降低出现 copyinstr 错误可能性的方法，请参见第 33 章。

copyinto()

void copyinto(uintptr_t *addr*, size_t *size*, void **dest*)

copyinto() 从指定的用户地址中复制指定的字节大小到 *dest* 指定的 DTrace 临时缓冲区中。用户地址解释为与当前线程关联的进程空间中的地址。有问题的地址必须与当前进程中的错误页面对应。如果地址与错误页面不对应，或者任何目标内存位于临时空间外部，则不会进行复制操作，并且将生成错误。有关降低出现 copyinto 错误可能性的方法，请参见第 33 章。

dirname()

```
string dirname(char *str)
```

dirname() 是 dirname(1) 的 D 类似程序。此子例程创建一个字符串，该字符串包含除 *str* 指定的最后一级路径名之外的所有内容。返回的字符串分配到临时内存外部，因此它们仅在子句的持续时间中有效。如果没有可用的足够临时内存，则不会执行 dirname，并且将生成错误。

msgdsize()

```
size_t msgdsize(mblk_t *mp)
```

msgdsize() 返回由 *mp* 指向的数据消息中的字节数。有关详细信息，请参见 msgdsize(9F)。msgdsize() 在计数中仅包括 M_DATA 类型的数据块。

msgsize()

```
size_t msgsize(mblk_t *mp)
```

msgsize() 返回由 *mp* 指向的消息中的字节数。与 msgdsize()（仅返回数据字节数）不同，msgsize() 返回消息中字节的总数。

mutex_owned()

```
int mutex_owned(kmutex_t *mutex)
```

mutex_owned() 是 mutex_owned(9F) 的实现。如果调用线程当前拥有指定的内核互斥，则 mutex_owned() 将返回非零值；如果调用线程当前不拥有指定的自适应互斥，则将返回零。

mutex_owner()

```
kthread_t *mutex_owner(kmutex_t *mutex)
```

mutex_owner() 返回所指定自适应内核互斥的当前属主的线程指针。如果当前不拥有指定的自适应互斥，或者指定的互斥为自旋互斥，则 mutex_owner() 将返回 NULL。请参见 mutex_owned(9F)。

mutex_type_adaptive()

```
int mutex_type_adaptive(kmutex_t *mutex)
```

如果指定的内核互斥为 MUTEX_ADAPTIVE 类型，则mutex_type_adaptive() 将返回非零值，如果不是该类型，后者将返回零。 如果互斥满足以下一个或多个条件，则它们是自适应的：

- 互斥以静态方式声明
- 互斥使用 NULL 的中断块 cookie 创建
- 互斥使用与高级中断不对应的中断块 cookie 创建

有关互斥的更多详细信息，请参见 mutex_init(9F)。 Solaris 内核中的大多数互斥都是自适应的。

progenyof()

```
int progenyof(pid_t pid)
```

如果调用进程（与当前正在触发匹配探测器的线程关联的进程）是指定进程 ID 的子孙，则progenyof() 将返回非零值。

rand()

```
int rand(void)
```

rand() 返回伪随机整数。 返回的数字为脆弱的伪随机数，不应用于任何加密应用程序。

rw_iswriter()

```
int rw_iswriter(krwlock_t *rwlock)
```

如果指定的读写锁由写入者拥有或者写入者希望获取该锁，则 rw_iswriter() 将返回非零值。 如果仅读取者拥有该锁，并且未阻止任何写入者；或者根本不拥有该锁，则 rw_iswriter() 将返回零。请参见 rw_init(9F)。

rw_write_held()

```
int rw_write_held(krwlock_t *rwlock)
```

r**如果指定的读取器-写入器锁定当前由写入者拥有，则** w_write_held() 将返回非零值。如果仅读取者拥有该锁，或者根本不拥有该锁，则 rw_write_held() 将返回零。 请参见 rw_init(9F)。

speculation()

```
int speculation(void)
```

speculation() 将随机跟踪缓冲区保留用于 speculate()，并返回此缓冲区的标识符。 有关详细信息，请参见第 13 章。

strjoin()

```
string strjoin(char *str1, char *str2)
```

strjoin() 创建一个字符串，该字符串包含 *str1* 与 *str2* 的串连。返回的字符串分配到临时内存外部，因此它们仅在子句的持续时间中有效。如果没有可用的足够临时空间，则不会执行 strjoin，并且将生成错误。

strlen()

```
size_t strlen(string str)
```

strlen() 返回指定的字符串的长度（以字节为单位），不包括结尾的空字节。

◆ ◆ ◆ 第 11 章

11 缓冲区和缓冲

数据缓冲和管理是 DTrace 框架为其客户机（如 dtrace(1M)）提供的基本服务。本章详细介绍了数据缓冲，以及可用于更改 DTrace 的缓冲区管理策略的选项。

主体缓冲区

在每个 DTrace 调用中都会使用主体缓冲区，它是缺省情况下跟踪操作记录其数据的缓冲区。 这些操作包括：

exit()	printf()	trace()	ustack()
printa()	stack()	tracemem()	

主体缓冲区始终基于每个 CPU 进行分配。 此策略不可以调整，但可以使用 cpu 选项将跟踪和缓冲区分配限制为单个 CPU。

主体缓冲区策略

DTrace 允许在高约束的内核上下文中使用跟踪。需要特别指出的是，DTrace 允许在内核软件不能可靠分配内存的上下文中使用跟踪。 这种灵活上下文的后果是：始终存在这样一种可能性，即 DTrace 会在没有可用空间时尝试跟踪数据。 出现这样的情况时，DTrace 必须具有处理此类情况的策略，但是您可能希望根据给定实验脚本的需要来调整策略。有时，相应的策略可能是放弃新数据。在另外一些时候，可能需要重用最早记录的数据所在的空间以跟踪新数据。 在大多数情况下，所需策略是尽可能避免用尽第一个位置中的可用空间。为了适应这些不同的需要，DTrace 支持多种不同的缓冲策略。 此支持使用 bufpolicy 选项实现，可以基于每个使用者进行设置。 有关设置选项的更多详细信息，请参见第 16 章。

switch 策略

缺省情况下，主体缓冲区具有 switch 缓冲区策略。 在此策略下，每个 CPU 的缓冲区都成对分配： 一个缓冲区处于活动状态，另一个缓冲区处于非活动状态。 在 DTrace 使用者尝试读取缓冲区时，内核首先切换非活动缓冲区和活动缓冲区。 缓冲区的切换方式应保证任何窗口中的跟踪数据都不会丢失。 切换缓冲区后，新的非活动缓冲区将复制给 DTrace 使用者。 此策略确保使用者始终看到前后一致的缓冲区： 不能同时跟踪和复制缓冲区。此方法还避免了引入跟踪被暂停或被阻止的窗口。 切换和读出缓冲区的速率由使用者通过 switchrate 选项控制。 与任何速率选项一样，可以使用任何时间后缀指定 switchrate，但缺省为每秒的速率。 有关 switchrate 和其他选项的更多详细信息，请参见第 16 章。

在 switch 策略下，如果给定的已启用探测器跟踪的数据超出了活动主体缓冲区中的可用空间，则会删除多余数据，并递增每个 CPU 的删除计数。 在发生一次或多次删除时，dtrace(1M) 显示一条与以下示例类似的消息：

```
dtrace: 11 drops on CPU 0
```

如果给定的记录大于总计缓冲区大小，则不管缓冲区策略是什么，都将删除记录。您可以通过使用 bufsize 选项增加主体缓冲区的大小或者使用 switchrate 选项增加切换速率，来降低或消除删除次数。

在 switch 策略下，在活动缓冲区外部分配 copyin()、copyinstr() 和 alloca() 的临时空间。

fill 策略

对于某些问题，您可能希望使用一个内核内部的缓冲区。虽然此方法可以使用 switch 策略和相应的 D 结构（通过在 D 中递增变量并相应地预测 exit() 操作）实现，但这样的实现不能消除发生删除的可能性。 要请求内核中的单个大缓冲区，并继续跟踪，直到一个或多个每 CPU 缓冲区已填充，请使用 fill 缓冲区策略。在此策略下，跟踪将继续，直到已启用的探测器尝试跟踪的数据多于剩余主体缓冲区空间中可以容纳的数据。在剩余空间不足时，缓冲区将标记为已填充，并且将通知使用者，至少有一个每 CPU 缓冲区已填充。 在 dtrace(1M) 检测到一个已填充的缓冲区后，将停止跟踪，并处理所有缓冲区，然后 dtrace 退出。 即使缓冲区中还可容纳一些数据，也不会在已填充缓冲区中跟踪更多的数据。

要使用 fill 策略，请将 bufpolicy 选项设置为 fill。例如，以下命令使用设置为 fill 的缓冲策略，在每 CPU 2K 的缓冲区中跟踪每个系统调用项：

```
# dtrace -n syscall:::entry -b 2k -x bufpolicy=fill
```

fill 策略和 END 探测器

在 DTrace 使用者显式停止跟踪之前，END 探测器通常不会触发。 END 探测器一定仅在一个 CPU 中触发，而且是将触发探测器的未定义的 CPU。使用 fill 缓冲区时，如果至少有一个每 CPU 主体缓冲区标记为已填充，便会显式停止跟踪。 如果选择了 fill 策略，则 END 探测

器可能在有已填充缓冲区的 CPU 中触发。为了适应 `fill` 缓冲区中的 `END` 跟踪，DTrace 将计算 `END` 探测器可能使用的空间的数量，然后从主体缓冲区的大小中减去此空间。 如果净大小为负数，DTrace 将拒绝启动，`dtrace`(1M) 将输出相应的错误消息：

```
dtrace: END enablings exceed size of principal buffer
```

预留空间机制可确保，对于任何 `END` 探测器，整个缓冲区始终有足够的空间。

`ring` 策略

使用 DTrace `ring` 缓冲策略，可以跟踪事件直到失败为止。如果重新生成失败需要几个小时或几天时间，您可能希望仅保留最新的数据。 在填充某个主体缓冲区后，跟踪将会绕回到第一项，从而覆盖旧的跟踪数据。 可以通过将 `bufpolicy` 选项设置为字符串 `ring` 来建立环形缓冲区：

```
# dtrace -s foo.d -x bufpolicy=ring
```

用于创建环形缓冲区时，`dtrace`(1M) 将不会显示任何输出，直到终止进程为止。此时，将会使用并处理环形缓冲区。`dtrace` 按照 CPU 顺序处理每个环形缓冲区。在 CPU 的缓冲区中，跟踪记录按照从旧到新的顺序显示。 与使用 `switch` 缓冲策略一样，不会对不同 CPU 之间的记录进行排序。如果需要这样的排序，应将 `timestamp` 变量作为跟踪请求的一部分进行跟踪。

以下示例演示了如何使用 `#pragma option` 指令启用环形缓冲：

```
#pragma D option bufpolicy=ring

#pragma D option bufsize=16k



syscall:::entry

/execname == $1/

{

    trace(timestamp);

}



syscall::rexit:entry
```

```
{

    exit(0);

}
```

其他缓冲区

每个 DTrace 启用中都存在主体缓冲区。除了主体缓冲区，一些 DTrace 使用者可能有其他内核中的数据缓冲区：聚合缓冲区（在第 9 章中讨论）和一个或多个随机缓冲区（在第 13 章中讨论）。

缓冲区大小

可以基于每个使用者调整每个缓冲区的大小。为每个缓冲区提供了不同的用于调整缓冲区大小的选项，如下表中所示：

缓冲区	大小选项
主体	`bufsize`
随机	`specsize`
聚合	`aggsize`

这些选项的每一个都使用表示大小的值设置。与任何大小选项一样，该值可以具有可选的大小后缀。有关更多详细信息，请参见第 16 章。例如，要在 `dtrace` 的命令行上将缓冲区大小设置为 1MB，可以使用 `-x` 设置该选项：

```
# dtrace -P syscall -x bufsize=1m
```

或者，可以将 `-b` 选项用于 `dtrace`：

```
# dtrace -P syscall -b 1m
```

最后，可以使用 `#pragma D option` 设置 `bufsize`：

```
#pragma D option bufsize=1m
```

所选择的缓冲区大小表示每个 CPU 中的缓冲区的大小。而且，对于 `switch` 缓冲区策略，`bufsize` 表示每个 CPU 中每个缓冲区的大小。缓冲区大小缺省为 4 MB。

缓冲区调整大小策略

偶尔，系统可能会因为没有足够的可用内存，或者 DTrace 使用者超出第 16 章中说明的某项可调整限制，而没有足够可用的内核内存来分配所需大小的缓冲区。 可以使用 bufresize 选项（缺省为 auto）配置缓冲区分配失败的策略。 在 auto 缓冲区调整大小策略下，将平均分配缓冲区大小，直到成功进行分配。如果分配的缓冲区小于请求的大小，dtrace(1M) 将生成一条消息：

```
# dtrace -P syscall -b 4g

dtrace: description 'syscall' matched 430 probes

dtrace: buffer size lowered to 128m

...
```

或：

```
# dtrace -P syscall'{@a[probefunc] = count()}' -x aggsize=1g

dtrace: description 'syscall' matched 430 probes

dtrace: aggregation size lowered to 128m

...
```

或者，可以通过将 bufresize 设置为 manual，要求在缓冲区分配失败之后进行手动干预。在此策略下，分配失败将会导致 DTrace 启动失败：

```
# dtrace -P syscall -x bufsize=1g -x bufresize=manual

dtrace: description 'syscall' matched 430 probes

dtrace: could not enable tracing: Not enough space

#
```

所有缓冲区（主体、随机和聚合）的缓冲区调整大小策略都由 bufresize 选项指定。

◆ ◆ ◆ 第 12 章

12 输出格式化

DTrace 提供了内置格式化函数 `printf()` 和 `printa()`，可在 D 程序中使用这些函数来设置输出的格式。D 编译器提供了 `printf`(3C) 库例程中没有的功能，所以，即使您已经熟悉 `printf()`，也应该阅读本章内容。本章还将讨论 `trace()` 函数的格式化行为以及 `dtrace`(1M) 用于显示聚合的缺省输出格式。

printf()

`printf()` 函数可跟踪数据，这部分功能类似于`trace()`函数，另外还可按指定的特定格式输出数据和其他文本。 `printf()` 函数会指示 DTrace 跟踪与第一个参数之后的每个参数关联的数据，然后使用第一个 `printf()` 参数说明的规则（称为格式字符串）来格式化结果。

格式字符串是包含任意数量的格式转换的规则字符串，其中的每个转换都以 `%` 字符开头，该字符说明如何设置对应的参数格式。 格式字符串中的第一个转换对应于第二个 `printf()` 参数，第二个转换对应于第三个参数，依此类推。 转换之间的所有文本将逐字列显。 `%` 转换字符之后的字符说明要用于对应参数的格式。

与 `printf`(3C) 不同，DTrace `printf()` 是 D 编译器可识别的内置函数。D 编译器为 DTrace `printf()` 提供了几项在 C 库 `printf()` 中未提供的有用服务：

- D 编译器可将参数与格式字符串中的转换进行比较。如果参数类型与格式转换不兼容，则 D 编译器会提供错误消息来说明该问题。
- D 编译器在 `printf()` 格式转换中不需要使用大小前缀。C `printf()` 例程要求通过添加前缀（如用 `%ld` 表示 `long`，或者用 `%lld` 表示 `long long`）来指示参数的大小。由于 D 编译器已知参数的大小和类型，因此 D `printf()` 语句中不需要这些前缀。
- DTrace 为调试和观察功能提供了其他有用的格式化字符。例如，`%a` 格式转换可用于将指针列显为符号名和偏移。

要实现这些功能，必须在 D 程序中将 DTrace `printf()` 函数中的格式字符串指定为字符串常量。格式字符串不能是 `string` 类型的动态变量。

转换规范

格式字符串中的每种转换规范都由 % 字符引入，在该字符后按顺序显示以下信息：

- 零个或多个标志（以任意顺序显示），用于修改转换规范的含义，如下节所述。
- 可选最小字段宽度。如果转换的值的字节数小于字段宽度，则缺省情况下将在该值的左侧填充空格，如果指定了左对齐标志 (-)，则将在该值的右侧填充空格。还可将字段宽度指定为星号 (*)，在此情况下，将根据 int 类型的另一参数的值动态设置字段宽度。
- 精度（可选），用于指示 d、i、o、u、x 和 X 转换要显示的最小位数（将对字段填充前导零）；e、E 和 f 转换的基数字符之后要显示的位数，g 和 G 转换要显示的最大有效位数；或者进行 s 转换后字符串中要列显的最大字节数。精度采用句点 (.) 后跟星号 (*) 的形式（如下所述），或十进制数字字符串的形式。
- 大小前缀序列（可选），用于指示对应参数的大小，如第 167 页中的 “大小前缀”中所述。大小前缀在 D 中不是必需的，提供它是为了与 C printf() 函数兼容。
- 转换说明符，用于指示要应用于参数的转换类型。

printf(3C) 函数还支持 %*n*$ 形式的转换规范，其中 *n* 是十进制整数；DTrace printf() 不支持此类型的转换规范。

标志说明符

可通过指定下列其中一个或多个字符（可以按任何顺序显示）来启用 printf() 转换标志：

' 十进制转换结果的整数部分（%i、%d、%u、%f、%g 或 %G）的格式是通过使用非货币分组字符的千位分组字符设置的。某些语言环境（包括 POSIX C 语言环境）未提供用于此标志的非货币分组字符。

- 转换结果将在字段中左对齐。如果未指定此标志，则转换将右对齐。

+ 带符号的转换结果始终以符号（+ 或 -）开头。如果未指定此标志，则仅当转换负值时，转换才以符号开头。

空格 如果带符号的转换的第一个字符不是符号或者未产生任何字符，则会在结果前放置空格。如果同时出现**空格**和 + 标志，则将忽略空格标志。

如果为所选转换定义了替换形式，则会将该值转换为替换形式。转换的替换格式将与对应转换一起说明。

0 对于 d、i、o、u、x、X、e、E、f、g 和 G 转换，前导零（跟在任何符号或基数表示之后）用于填充字段宽度。不会执行任何空格填充。如果同时出现 **0** 和 - 标志，则将忽略 **0** 标志。对于 d、i、o、u、x 和 X 转换，如果指定了精度，则将忽略 **0** 标志。如果同时出现 **0** 和 ' 标志，则会在进行零填充之前插入分组字符。

宽度和精度说明符

可将最小字段宽度指定为任意标志说明符后跟十进制数字字符串，在此情况下，字段宽度将设置为指定的列数。字段宽度还可以指定为星号 (*)，在此情况下，将访问 int 类型的另一参数来确定字段宽度。例如，要在字段中列显整数 x（由 int 变量 w 的值确定），应编写如下 D 语句：

```
printf("%*d", w, x);
```

字段宽度也可使用 ? 字符来指定，以指定应根据设置地址格式（该地址在操作系统内核的数据模型中使用十六进制）所需的字符数来设置字段宽度。如果内核使用 32 位数据模型，则宽度设置为 8，如果内核使用 64 位数据模型，则宽度设置为 16。

转换的精度可指定为跟在句点 (.) 之后的十进制数字字符串，或者跟在句点之后的星号 (*)。如果使用星号来指定精度，则会访问转换参数之前的类型为 int 的另一参数来确定精度。如果同时将宽度和精度指定为星号，则对应转换的 printf() 参数的顺序应按以下顺序显示：宽度、精度、值

大小前缀

在使用 printf(3C) 指示转换参数的大小和类型的 ANSI-C 程序中，需要使用大小前缀。因为 D 编译器会自动对 printf() 调用执行此处理，所以不需要大小前缀。尽管提供大小前缀是为了与 C 兼容，但因为在使用派生类型时它们会将代码绑定到特定数据模型，所以明确建议不要在 D 程序中使用它们。例如，如果根据数据模型将 typedef 重新定义为其他整数基本类型，则在未明确知道两种基础类型并且包括强制转换表达式（或定义多个格式字符串）的情况下，不能使用单个 C 转换来同时处理两种数据模型。D 编译器通过允许省略大小前缀并自动确定参数大小来自动解决此问题。

大小前缀只能放在格式转换名称之前和所有标志、宽度以及精度说明符之后。大小前缀如下所示：

- h（可选），用于指定后续的 d、i、o、u、x 或 X 转换将应用于 short 或 unsigned short。
- l（可选），用于指定后续的 d、i、o、u、x 或 X 转换将应用于 long 或 unsigned long。
- Il（可选），用于指定后续的 d、i、o、u、x 或 X 转换将应用于 long long 或 unsigned long long。
- L（可选），用于指定后续的 e、E、f、g 或 G 转换将应用于 long double。
- l（可选），用于指定后续的 c 转换将应用于 wint_t 参数，并且后续的 s 转换字符将应用于指向 wchar_t 参数的指针。

转换格式

每个转换字符序列都将导致提取零个或多个参数。如果没有为格式字符串提供足够的参数，或者格式字符串已用完但仍有参数剩余，则 D 编译器将发出相应的错误消息。如果指定了未定义的转换格式，则 D 编译器将发出相应的错误消息。转换字符序列为：

a　　指针或 `uintptr_t` 参数将列显为如下形式的内核符号名：*module*`‘`*symbol-name* 加上十六进制字节偏移（可选）。如果该值不在已知内核符号定义的范围内，则该值将列显为十六进制整数。

c　　`char`、`short` 或 `int` 参数将列显为 ASCII 字符。

C　　如果字符为可列显的 ASCII 字符，则 `char`、`short` 或 `int` 参数将列显为 ASCII 字符。如果该字符不是可列显字符，则将使用 表 2–5 中说明的对应转义序列将其列显。

d　　`char`、`short`、`int`、`long` 或 `long long` 参数将列显为十进制（以 10 为基数）整数。如果参数类型为 `signed`，则将列显为带符号的值。如果参数类型为 `unsigned`，则将列显为无符号值。此转换与 `i` 具有相同的含义。

e, E　　`float`、`double` 或 `long double` 参数将转换为 `[-]`*d*`.`*ddd*`e±`*dd* 样式，其中基数字符前有一位，基数字符后的位数等于精度。如果参数不为零，则基数字符也不为零。如果未指定精度，则缺省精度值为 6。如果精度为 0 并且未指定 `#` 标志，则不会显示任何基数字符。`E` 转换格式将生成一个数字，该数字将使用 `E` 而不是 `e` 来引入指数。指数总是至少包含两位数。该值将向上舍入为相应的位数。

f　　`float`、`double` 或 `long double` 参数将转换为 `[-]`*ddd*`.`*ddd* 样式，其中基数字符之后的位数等于指定的精度。如果未指定精度，则缺省精度值为 6。如果精度为 0 并且未指定 `#` 标志，则不会显示任何基数字符。如果显示基数字符，则它之前至少会显示一位数字。该值将向上舍入为相应的位数。

g, G　　`float`、`double` 或 `long double` 参数将列显为 `f` 或 `e` 样式（如果使用 `G` 转换字符则为 `E` 样式），并且使用指定有效数字位数的精度。如果显式精度为 0，则会将其视为 1。使用的样式取决于转换的值：仅当转换生成的指数小于 -4 或大于等于精度时，才会使用 `e`（或 `E`）样式。结果的小数部分中的结尾零将删除。仅当基数字符后跟数字时，才会显示基数字符。如果指定了 `#` 标志，则不会从结果中删除结尾零。

i　　`char`、`short`、`int`、`long` 或 `long long` 参数将列显为十进制（以 10 为基数）整数。如果参数类型为 `signed`，则将列显为带符号的值。如果参数类型为 `unsigned`，则将列显为无符号值。此转换与 `d` 具有相同的含义。

o　　`char`、`short`、`int`、`long` 或 `long long` 参数将列显为无符号的八进制（以 8 为基数）整数。`signed` 类型或 `unsigned` 类型的参数可用于此转换。如果指定了 `#` 标志，则在需要将结果的第一位强制为零时，结果的精度将增加。

p　　指针或 `uintptr_t` 参数将列显为十六进制（以 16 为基数）整数。D 接受任何类型的指针参数。如果指定了 `#` 标志，则将在非零结果的开头加上 `0x`。

s　　参数必须为 `char` 数组或 `string`。将一直读取数组或 `string` 中的字节，直到达到终止空字符或数据的结尾，并将其解释和列显为 ASCII 字符。如果未指定精度，则会

将其视为无穷的，所以将列显所有字符，直到达到第一个空字符。如果指定了精度，则仅会列显在对应屏幕列数中显示的那一部分字符数组。如果要格式化 `char *` 类型的参数，则应将该参数强制转换为 `string`，或者使用 D `stringof` 运算符为前缀，以指示 DTrace 应跟踪的字符串字节数并设置这些字节的格式。

`S` 参数必须为 `char` 数组或 `string`。参数将按类似 `%s` 转换的方式进行处理，但不可列显的所有 ASCII 字符都将替换为表 2–5 中说明的对应转义序列。

`u` `char`、`short`、`int`、`long` 或 `long long` 参数将列显为无符号十进制（以 10 为基数）整数。`signed` 类型或 `unsigned` 类型的参数可用于此转换，并且结果都将格式化为 `unsigned`。

`wc` `int` 参数将转换为宽字符 (`wchar_t`) 并且将列显生成的宽字符。

`ws` 参数必须为 `wchar_t` 数组。会一直读取数组中的字节，直到达到终止空字符或数据结尾，并将其解释和列显为宽字符。如果未指定精度，则会将其视为是无穷的，所以将列显所有宽字符，直到达到第一个空字符。如果指定了精度，则仅会列显在对应屏幕列数中显示的那一部分宽字符数组。

`x, X` `char`、`short`、`int`、`long` 或 `long long` 参数将列显为无符号十六进制（以 16 为基数）整数。`signed` 类型或 `unsigned` 类型的参数可用于与此转换。如果使用 `x` 形式的转换，则将使用字母数字 `abcdef`。如果使用 `X` 形式的转换，则将使用字母数字 `ABCDEF`。如果指定了 `#` 标志，则将在非零结果的开头加上 `0x`（对于 `%x`）或 `0X`（对于 `%X`）。

`Y` `uint64_t` 参数将解释为自 1970 年 1 月 1 日 00:00 世界标准时间以来的纳秒数，并且按以下 `cftime`(3C) 形式列显：“`%Y %a %b %e %T %Z`”。自 1970 年 1 月 1 日 00:00 世界标准时间以来的当前纳秒数将通过 `walltimestamp` 变量提供。

`%` 列显文字 `%` 字符。不会转换任何参数。完整的转换规范必须为 `%%`。

printa()

`printa()` 函数用于格式化 D 程序中聚合的结果。可通过下列两种形式之一来调用该函数：

```
printa(@aggregation-name);

printa(format-string, @aggregation-name);
```

如果使用函数的第一种形式，则 `dtrace`(1M) 命令将提取聚合数据的一致快照并生成输出，该输出与用于聚合的缺省输出格式等效，如第 9 章中所述。

如果使用函数的第二种形式，则 `dtrace`(1M) 命令将提取聚合数据的一致快照，并根据 *format string* 中指定的转换以及下列规则来生成输出：

- 格式转换必须与用于创建聚合的元组签名相匹配。每个元组元素仅能显示一次。例如，如果使用下列 D 语句来聚合计数：

```
@a["hello", 123] = count();

@a["goodbye", 456] = count();
```

然后将 D 语句 `printa(`*format-string*`, @a)` 添加到探测器子句，则 `dtrace` 将提取聚合数据的快照并生成输出，就像输入了下列语句：

```
printf(format-string, "hello", 123);

printf(format-string, "goodbye", 456);
```

及类似语句一样（对于聚合中定义的每个元组）。

- 与 `printf()` 不同，用于 `printa()` 的格式字符串无需包括元组的所有元素。即，元组的长度可以为 3，并且仅进行一次格式转换。因此，可以通过更改聚合声明将要在 `printa()` 输出中省略的任何元组键移到元组的结尾来将其省略，然后在 `printa()` 格式字符串中省略其对应的转换说明符。
- 可通过使用附加的 `@` 格式标志字符（仅在用于 `printa()` 时有效）将聚合结果包括在输出中。`@` 标志可与任何相应的格式转换说明符组合，并且可在一个格式字符串中多次显示，以便元组结果可在输出中的任何位置多次显示。可用于聚合函数的一组转换说明符隐含在聚合函数的结果类型中。聚合结果类型包括：

`avg()`	`uint64_t`
`count()`	`uint64_t`
`lquantize()`	`int64_t`
`max()`	`uint64_t`
`min()`	`uint64_t`
`quantize()`	`int64_t`
`sum()`	`uint64_t`

例如，要格式化 `avg()` 的结果，可应用 `%d`、`%i`、`%o`、`%u` 或 `%x` 格式转换。`quantize()` 和 `lquantize()` 函数会将其结果格式化为 ASCII 表而不是单个值。

以下 D 程序显示 `printa()` 的完整示例，通过使用`profile` 提供器来对 `caller` 的值进行采样，然后将结果格式化为简单的表：

```
profile:::profile-997

{

    @a[caller] = count();
```

```
}


END

{

    printa("%@8u %a\n", @a);

}
```

如果使用 dtrace 来执行此程序，请等待几秒钟，然后按 Ctrl-C 组合键，将会看到与以下示例类似的输出：

```
# dtrace -s printa.d

^C

CPU     ID                    FUNCTION:NAME

  1      2                             :END        1 0x1

       1 ohci‘ohci_handle_root_hub_status_change+0x148

       1 specfs‘spec_write+0xe0

       1 0xff14f950

       1 genunix‘cyclic_softint+0x588

       1 0xfef2280c

       1 genunix‘getf+0xdc

       1 ufs‘ufs_icheck+0x50

       1 genunix‘infpollinfo+0x80

       1 genunix‘kmem_log_enter+0x1e8

       ...
```

trace() 缺省格式

如果使用 trace() 函数而不是 printf() 函数来捕获数据，则 dtrace 命令将使用缺省输出格式来设置结果的格式。如果数据大小为 1、2、4 个字节或 8 个字节，则结果格式将设置为十进制整数值。如果数据为任何其他大小并且是可列显字符序列（如果解释为字节序列），则它将列显为 ASCII 字符串。如果数据为任何其他大小并且不是可列显字符序列，则它将列显为一系列十六进制整数格式的字节值。

◆ ◆ ◆ 第 13 章

13 推理跟踪

本章介绍用于推理跟踪的 DTrace 工具，它可用于试探性地跟踪数据，并在以后决定是将这些数据提交到跟踪缓冲区还是放弃这些数据。在 DTrace 中，过滤出非关注事件的主要机制是谓词机制（已在第 4 章中介绍）。如果在触发探测器时，要知道该探测事件是否是所关注的事件，此时谓词很有用。例如，如果您只关注与某个进程或某个文件描述符关联的活动，则在触发探测器时，您将知道该活动是否与所关注的进程或文件描述符关联。但是，在其他情况下，可能需要在触发探测器之后的某个时间，才可知道指定的探测事件是否是所关注的事件。

例如，如果系统调用偶尔失败，并生成常见的错误代码（例如，`EIO` 或 `EINVAL`），则可能需要检查导致该错误的代码路径。要捕获代码路径，可以启用所有探测器－但只有隔离失败的调用，才可以构造有意义的谓词。如果故障具有偶发性或不确定性，则必须跟踪可能需要关注的所有事件，然后对数据进行后期处理以过滤出与失败的代码路径无关的数据。在此情况下，即使需要关注的事件数量可能相当少，但必须跟踪的事件数量还是会非常大，这将使得后期处理很困难。

在这些情况下，您可以使用推理跟踪工具在一个或多个探测位置试探性地跟踪数据，然后在另一个探测位置决定是否将这些数据提交到主体缓冲区。因此，跟踪数据将只包含所关注的输出，无需进行后期处理，从而 DTrace 开销也降至最低。

推理接口

下表说明了 DTrace 推理函数：

表 13–1 DTrace 推理函数

函数名	参数	说明
`speculation`	无	返回新推理缓冲区的标识符
`speculate`	ID	表示子句的其余部分应跟踪到由 ID 指定的推理缓冲区

表 13–1 DTrace 推理函数 （续）

函数名	参数	说明
commit	ID	提交与 ID 关联的推理缓冲区
discard	ID	放弃与 ID 关联的推理缓冲区

创建推理

speculation() 函数分配推理缓冲区并返回推理标识符。 在对 speculate() 函数的后续调用中应使用推理标识符。 推理缓冲区是一种有限的资源：如果在调用 speculation() 时无推理缓冲区可用，则返回的 ID 为零，并将递增相应的 DTrace 错误计数器。 值为零的 ID 始终无效，但可以传递到 speculate()、commit() 或 discard()。 如果调用 speculation() 失败，则将生成与以下示例类似的 dtrace 消息：

```
dtrace: 2 failed speculations (no speculative buffer space available)
```

推理缓冲区的数量缺省值为 1，但是也可以调高。有关更多信息，请参见第 184 页中的 “推理选项和调整”。

使用推理

要使用推理，必须在执行任何数据记录操作之前，将从 speculation() 返回的标识符传递到子句中的 speculate() 函数。包含 speculate() 的子句中所有后续数据记录操作将被推理跟踪。 如果在 D 探测子句中对 speculate() 的调用紧跟在数据记录操作之后，D 编译器将会生成编译时错误。 因此，子句可以包含推理跟踪请求或非推理跟踪请求，但不能同时包含二者。

不能对聚集操作、破坏性操作和 exit 操作进行推理跟踪。在包含 speculate() 的子句中采用这其中某个操作的任何尝试，都将导致编译时错误。一个speculate() 不能跟在另一个 speculate() 之后：每条子句中只允许使用一个推理。只包含一个 speculate() 的子句将推理性地跟踪缺省操作，该缺省操作被定义为仅跟踪已经启用的探测器 ID。有关缺省操作的说明，请参见第 10 章。

通常，将 speculation() 的结果赋给线程局部变量，然后使用该变量作为其他探测器的后续谓词以及 speculate() 的参数。例如：

```
syscall::open:entry

{

    self->spec = speculation();

}
```

```
syscall:::

/self->spec/

{

    speculate(self->spec);

    printf("this is speculative");

}
```

提交推理

请使用 commit() 函数提交推理。 提交推理缓冲区时，其中的数据将被复制到主体缓冲区。 如果指定的推理缓冲区中包含的数据多于主体缓冲区中的可用空间，则不会复制该数据，并将递增缓冲区的删除计数。 如果已推理跟踪多个 CPU 中的缓冲区，则会立即复制要提交的 CPU 中的推理数据，而其他 CPU 中的推理数据将在 commit() 之后的某个时间复制。 因此，从 commit() 开始在一个 CPU 中运行到数据从推理缓冲区复制到所有 CPU 中的主体缓冲区之间可能会经历一段时间。 此时间肯定不长于清除速率指定的时间。有关更多详细信息，请参见第 184 页中的 “推理选项和调整” 。

在将每个 CPU 的推理缓冲区完全复制到相应的每个 CPU 的主体缓冲区之前，后续 speculation() 调用将不能使用要提交的推理缓冲区。 类似地，提交缓冲区对 speculate() 的后续调用将默认被放弃，对 commit() 或 discard() 的后续调用将默认失败。 最后，包含 commit() 的子句不能包含数据记录操作，但是子句可能包含多个 commit() 调用来提交不相交的缓冲区。

放弃推理

可使用 discard() 函数放弃推理。 放弃推理缓冲区时，其中的内容将被丢弃。 如果推理仅在调用 discard() 的 CPU 中处于活动状态，则后续对 speculation() 的调用将可以立即使用该缓冲区。 如果推理在多个 CPU 中处于活动状态，则在调用 discard() 之后的某个时间，后续 speculation() 将可以使用放弃的缓冲区。在一个 CPU 中执行 discard() 之后到后续推理操作可以使用缓冲区之间的时间肯定不长于清除速率指定的时间。 如果在调用 speculation() 时，因为所有推理缓冲区当前正在被放弃或提交，而没有可用的缓冲区，将会生成与以下示例类似的 dtrace 消息：

```
dtrace: 905 failed speculations (available buffer(s) still busy)
```

可以通过调整推理缓冲区的数量或清除速率，来降低所有缓冲区都不可用的可能性。有关详细信息，请参见第 184 页中的 “推理选项和调整” 。

推理示例

推理的一个可能用法是突出显示特定的代码路径。只有当 `open()` 失败时，以下示例才在 `open(2)` 系统调用下显示整个代码路径：

示例 13–1 `specopen.d`: 失败的 `open(2)` 代码流

```
#!/usr/sbin/dtrace -Fs


syscall::open:entry,
syscall::open64:entry
{
    /*
     * The call to speculation() creates a new speculation.  If this fails,
     * dtrace(1M) will generate an error message indicating the reason for
     * the failed speculation(), but subsequent speculative tracing will be
     * silently discarded.
     */
    self->spec = speculation();
    speculate(self->spec);

    /*
     * Because this printf() follows the speculate(), it is being
     * speculatively traced; it will only appear in the data buffer if the
     * speculation is subsequently commited.
     */
    printf("%s", stringof(copyinstr(arg0)));
}
```

示例 13–1 specopen.d: 失败的 open(2) 代码流 （续）

```
fbt:::
/self->spec/
{
    /*
     * A speculate() with no other actions speculates the default action:
     * tracing the EPID.
     */
    speculate(self->spec);
}

syscall::open:return,
syscall::open64:return
/self->spec/
{
    /*
     * To balance the output with the -F option, we want to be sure that
     * every entry has a matching return.  Because we speculated the
     * open entry above, we want to also speculate the open return.
     * This is also a convenient time to trace the errno value.
     */
    speculate(self->spec);
    trace(errno);
```

示例 13–1 specopen.d: 失败的 open(2) 代码流 （续）

```
}


syscall::open:return,
syscall::open64:return
/self->spec && errno != 0/
{
    /*
     * If errno is non-zero, we want to commit the speculation.
     */
    commit(self->spec);
    self->spec = 0;
}


syscall::open:return,
syscall::open64:return
/self->spec && errno == 0/
{
    /*
     * If errno is not set, we discard the speculation.
     */
    discard(self->spec);
    self->spec = 0;
}
```

运行上面的脚本将会生成与以下示例类似的输出：

```
# ./specopen.d

dtrace: script ’./specopen.d’ matched 24282 probes

CPU FUNCTION

  1  => open                                   /var/ld/ld.config

  1    -> open

  1      -> copen

  1        -> falloc

  1          -> ufalloc

  1            -> fd_find

  1              -> mutex_owned

  1              <- mutex_owned

  1            <- fd_find

  1            -> fd_reserve

  1              -> mutex_owned

  1              <- mutex_owned

  1              -> mutex_owned

  1              <- mutex_owned

  1            <- fd_reserve

  1          <- ufalloc

  1          -> kmem_cache_alloc

  1            -> kmem_cache_alloc_debug

  1              -> verify_and_copy_pattern

  1              <- verify_and_copy_pattern
```

```
1              -> file_cache_constructor
1                -> mutex_init
1                <- mutex_init
1              <- file_cache_constructor
1              -> tsc_gethrtime
1              <- tsc_gethrtime
1              -> getpcstack
1              <- getpcstack
1              -> kmem_log_enter
1              <- kmem_log_enter
1            <- kmem_cache_alloc_debug
1          <- kmem_cache_alloc
1          -> crhold
1          <- crhold
1        <- falloc
1        -> vn_openat
1          -> lookupnameat
1            -> copyinstr
1            <- copyinstr
1            -> lookuppnat
1              -> lookuppnvp
1                -> pn_fixslash
1                <- pn_fixslash
1                -> pn_getcomponent
```

```
1                <- pn_getcomponent
1                -> ufs_lookup
1                  -> dnlc_lookup
1                    -> bcmp
1                    <- bcmp
1                  <- dnlc_lookup
1                  -> ufs_iaccess
1                    -> crgetuid
1                    <- crgetuid
1                    -> groupmember
1                      -> supgroupmember
1                      <- supgroupmember
1                    <- groupmember
1                  <- ufs_iaccess
1                <- ufs_lookup
1                -> vn_rele
1                <- vn_rele
1                -> pn_getcomponent
1                <- pn_getcomponent
1                -> ufs_lookup
1                  -> dnlc_lookup
1                    -> bcmp
1                    <- bcmp
1                  <- dnlc_lookup
```

```
1                  -> ufs_iaccess
1                    -> crgetuid
1                    <- crgetuid
1                  <- ufs_iaccess
1                <- ufs_lookup
1                -> vn_rele
1                <- vn_rele
1                -> pn_getcomponent
1                <- pn_getcomponent
1                -> ufs_lookup
1                  -> dnlc_lookup
1                    -> bcmp
1                    <- bcmp
1                  <- dnlc_lookup
1                  -> ufs_iaccess
1                    -> crgetuid
1                    <- crgetuid
1                  <- ufs_iaccess
1                  -> vn_rele
1                  <- vn_rele
1                <- ufs_lookup
1                -> vn_rele
1                <- vn_rele
1              <- lookuppnvp
```

```
1            <- lookuppnat
1          <- lookupnameat
1        <- vn_openat
1        -> setf
1          -> fd_reserve
1            -> mutex_owned
1            <- mutex_owned
1            -> mutex_owned
1            <- mutex_owned
1          <- fd_reserve
1          -> cv_broadcast
1          <- cv_broadcast
1        <- setf
1        -> unfalloc
1          -> mutex_owned
1          <- mutex_owned
1          -> crfree
1          <- crfree
1          -> kmem_cache_free
1            -> kmem_cache_free_debug
1              -> kmem_log_enter
1              <- kmem_log_enter
1              -> tsc_gethrtime
1              <- tsc_gethrtime
```

```
  1              -> getpcstack
  1              <- getpcstack
  1              -> kmem_log_enter
  1              <- kmem_log_enter
  1              -> file_cache_destructor
  1                -> mutex_destroy
  1                <- mutex_destroy
  1              <- file_cache_destructor
  1              -> copy_pattern
  1              <- copy_pattern
  1            <- kmem_cache_free_debug
  1          <- kmem_cache_free
  1        <- unfalloc
  1        -> set_errno
  1        <- set_errno
  1      <- copen
  1    <- open
  1  <= open                                          2
```

推理选项和调整

如果在尝试推理跟踪操作时推理缓冲区已满，则不会在缓冲区中存储任何数据，并将递增删除计数。 在此情况下，将会生成与以下示例类似的 `dtrace` 消息：

```
dtrace: 38 speculative drops
```

在提交缓冲区后，推理删除不会阻止将整个推理缓冲区复制到主体缓冲区。类似地，即使在最终放弃的推理缓冲区中执行过删除，也可能会发生推理删除。 通过增加推理缓冲区大小（使用 `specsize` 选项调整），可减少推理删除。 可以使用任何大小后缀指定 `specsize` 选项。 调整此缓冲区大小的策略由 `bufresize` 选项指示。

调用 speculation() 时，推理缓冲区可能不可用。 如果存在尚未提交或放弃的缓冲区，则会生成与以下示例类似的 dtrace 消息：

```
dtrace: 1 failed speculation (no speculative buffer available)
```

通过使用 nspec 选项增加推理缓冲区的数量，可以降低此特性的推理失败的可能性。 nspec 的缺省值为 1。

此外，speculation() 可能会因为所有推理缓冲区正忙而失败。 在此情况下，将会生成与以下示例类似的 dtrace 消息：

```
dtrace: 1 failed speculation (available buffer(s) still busy)
```

此消息说明，在对推理缓冲区调用 commit() 之后，但在所有 CPU 中实际提交该缓冲区之前，调用了 speculation()。通过使用 cleanrate 选项增加清除 CPU 的速率，可以降低此特性的推理操作失败的可能性。 cleanrate 的缺省值为 101。

◆ ◆ ◆ 第 14 章

dtrace(1M) 实用程序

dtrace(1M) 命令是 DTrace 工具的通用前端。 该命令实现了用于调用 D 语言编译器的简单界面、通过 DTrace 内核工具检索缓存的跟踪数据的功能以及一组用于格式化和列显跟踪数据的基本例程。 本章提供 dtrace 命令的完整参考。

说明

dtrace 命令提供了一个可访问 DTrace 工具提供的所有基本服务的通用界面，包括：

- 用于列出 DTrace 当前发布的探测器和提供器集的选项
- 允许使用任何探测器说明符（提供器、模块、函数和名称）直接启用探测器的选项
- 用于运行 D 编辑器并编译一个或多个直接在命令行上编写的 D 程序文件或程序的选项
- 用于生成匿名跟踪程序的选项（请参见第 36 章）
- 用于生成程序稳定性报告的选项（请参见第 39 章）
- 用于修改 DTrace 跟踪和缓冲行为并启用其他 D 编译器功能的选项（请参见第 16 章）

还可通过在 #! 声明中使用 dtrace创建解释程序文件，来创建 D 脚本（请参见第 15 章）。 最后，可在不使用 -e 选项启用任何跟踪的情况下，使用 dtrace 尝试编译 D 程序并确定它们的属性，如下文所述。

选项

dtrace 命令接受下列选项：

dtrace [-32 | -64] [-aACeFGHlqSvVwZ] [-b *bufsz*] [-c *cmd*] [-D *name* [=*def*]] [-I *path*] [-L *path*] [-o *output*] [-p *pid*] [-s *script*] [-U *name*] [-x *arg* [=*val*]] [-Xa | c | s | t] [-P *provider* [[*predicate*]*action*]] [-m [[*provider:*]*module* [[*predicate*]*action*]]] [-f [[*provider:*]*module:*]*func* [[*predicate*]*action*]] [-n [[[*provider:*]*module:*]*func:*]*name* [[*predicate*]*action*]] [-i *probe-id* [[*predicate*]*action*]]

其中，*predicate* 是置于斜杠 / / 中的任何 D 谓词，而 *action* 是用括号 { } 括起来的任何 D 语句列表（根据先前说明的 D 语言语法）。如果将 D 程序代码提供为 -P、-m, -f、-n 或 -i 选项的参数，则必须相应地用引号将此文本引起来以避免 shell 对其进行解释。这些选项如下所示：

-32, -64　D 编译器使用操作系统内核的本机数据模型生成程序。可使用 isainfo(1) -b 命令确定当前操作系统的数据模型。如果指定了 -32 选项，则 dtrace 将强制 D 编译器使用 32 位数据模型编译 D 程序。如果指定了 -64 选项，则 dtrace 将强制 D 编译器使用 64 位数据模型编译 D 程序。通常这些选项不是必需的，因为 dtrace 会选择本机数据模型作为缺省数据模型。数据模型会影响整数类型的大小和其他语言属性。对任一数据模型编译的 D 程序可在 32 位和64 位内核上执行。-32 和 -64 选项还可确定 -G 选项生成的 ELF 文件格式（ELF32 或 ELF64）。

-a　声明匿名跟踪状态并显示跟踪数据。可将 -a 选项与 -e 选项组合在一起，以强制 dtrace 在使用匿名跟踪状态后立即退出，而不是继续等待新数据。有关匿名跟踪的更多信息，请参见第 36 章。

-A　生成 driver.conf(4) 指令以便进行匿名跟踪。如果指定了 -A 选项，则 dtrace 将编译使用 -s 选项或在命令行上指定的任何 D 程序，并构造 dtrace(7D) 配置文件指令集，以允许指定探测器进行匿名跟踪（请参见第 36 章），然后退出。缺省情况下，dtrace 尝试将指令存储到文件 /kernel/drv/dtrace.conf 中。可通过使用 -o 选项指定备用输出文件来修改此行为。

-b　设置主体跟踪缓冲区大小。跟踪缓冲区大小可以使用任何大小后缀（k、m、g 或 t），如第 36 章中所述。如果不能分配缓冲区空间，则 dtrace 将根据 bufresize 属性的设置减小缓冲区大小或退出。

-c　运行指定的命令 *cmd*，并在完成时退出。如果命令行上有多个 -c 选项，则 dtrace 将在所有命令退出后退出，并报告每个子进程终止时的退出状态。第一个命令的进程 ID 可供在命令行上（或者使用 -s 选项通过 $target 宏变量）指定的任何 D 程序使用。有关宏变量的更多信息，请参阅第 15 章。

-C　在编译 D 程序之前对它们运行 C 预处理程序 cpp(1)。可使用 -D、-U、-I 和 -H 选项将这些选项传递到 C 预处理程序。可使用 -X 选项选择符合 C 标准的程度。有关在调用 C 预处理程序时由 D 编译器定义的标记集的说明，请参阅 -X 选项的说明。

-D　在调用 cpp(1)（使用 -C 选项启用）时定义指定的 *name*。如果指定了等号 (=) 以及 *value*，则会将该值赋给名称。此选项将 -D 选项传递到每个 cpp 调用。

-e　在编译所有请求并使用匿名跟踪状态（-a 选项）之后、启用任何探测器之前退出。此选项可与 -a 选项组合在一起，用于列显匿名跟踪数据并退出，它也可以与 D 编译器选项一起使用，用于验证在不实际执行程序并启用相应检测过程的情况下，程序是否会进行编译。

-f　指定要跟踪或列出的函数名称（-l 选项）。相应参数可以包括下列任何探测器说明形式：*provider:module:function*、*module:function* 或 *function*。未指定的探测器说明字段将留为空白，可与任何探测器相匹配，无论这些字段中的值是什

么。如果说明中仅指定了 *function* 限定符，则会匹配所有包含相应 *function* 的探测器。可将可选 D 探测器子句用作 `-f` 参数的后缀。 在命令行上，一次可以指定多个 `-f` 选项。

`-F` 通过标识函数的入口和返回位置来合并跟踪输出。对函数入口的探测报告将进行缩进，并在其输出内容前附带 `->` 前缀。对函数返回的探测报告将不进行缩进，并在其输出内容前附带 `<-` 前缀。

`-G` 生成包含嵌入式 DTrace 程序的 ELF 文件。在程序中指定的 DTrace 探测器将保存在可与其他程序链接的可重定位 ELF 对象中。如果 `-o` 选项存在，则会使用指定为此操作数参数的路径名来保存 ELF 文件。如果 `-o` 选项不存在，并且 DTrace 程序与名为 *filename*`.s` 的文件一起包含在 ELF 文件中，则将使用名称 *file*`.o` 来保存 ELF 文件；否则将使用名称 `d.out` 来保存 ELF 文件。

`-H` 在调用 `cpp`(1)（使用 `-C` 选项启用）时列显所包含文件的路径名。此选项会将 `-H` 选项传递到每个 `cpp` 调用，从而使调用将路径名的列表显示到 `stderr`，每个路径名显示为一行。

`-i` 指定要跟踪或列出的探测器标识符（`-l` 选项）。 如 `dtrace -l` 所示，探测器 ID 指定为十进制整数。 可将可选 D 探测器子句用作 `-i` 参数的后缀。 在命令行上，一次可以指定多个 `-i` 选项。

`-I` 在调用 `cpp`(1)（使用 `-C` 选项启用）时，将指定的目录 *path* 添加到 `#include` 文件的搜索路径。此选项将 `-I` 选项传递到每个 `cpp` 调用。指定的目录将插入到缺省目录列表之前的搜索路径中。

`-l` 列出探测器而不是启用它们。 如果指定了 `-l` 选项，则 `dtrace` 将生成与使用 `-P`、`-m`、`-f`、`-n`、`-i` 和 `-s` 选项指定的说明相匹配的探测器报告。 如果未指定其中任何选项，则将列出所有探测器。

`-L` 将指定的目录 *path* 添加到 DTrace 库的搜索路径。DTrace 库用于包含可在编写 D 程序时使用的常见定义。指定的 *path* 将添加到缺省库搜索路径后面。

`-m` 指定要跟踪或列出的模块名称（`-l` 选项）。相应参数可以包括下列任何探测器说明形式：*provider*:*module* 或 *module*。未指定的探测器说明字段将留为空白，并且与任何探测器相匹配，无论这些字段中的值是什么。 如果说明中仅指定了 *module* 限定符，则会匹配所有包含相应 *module* 的探测器。可将可选 D 探测器子句用作 `-m` 参数的后缀。 在命令行上，一次可以指定多个 `-m` 选项。

`-n` 指定要跟踪或列出的探测器名称（`-l` 选项）。相应参数可以包括下列任何探测器说明形式：*provider*:*module*:*function*:*name*、*module*:*function*:*name*、*function*:*name* 或 *name*。 未指定的探测器说明字段将留为空白，并且与任何探测器相匹配，无论这些字段中的值是什么。如果说明中仅指定了 *name* 限定符，则会匹配所有包含相应 *name* 的探测器。可将可选 D 探测器子句用作 `-n` 参数的后缀。在命令行上，一次可以指定多个 `-n` 选项。

`-o` 对 `-A`、`-G` 和 `-l` 选项或跟踪数据指定 *output* 选项。如果 `-A` 选项存在而 `-o` 不存在，则缺省输出文件为 `/kernel/drv/dtrace.conf`。如果 `-G` 选项存在并且 `-s` 选项的参数为 *filename*`.d` 形式，而 `-o` 不存在，则缺省输出文件为 *filename*`.o`；否则缺省输出文件为 `d.out`。

`-p` 获取指定的进程 ID *pid*，高速缓存其符号表，然后在完成时退出。如果命令行上有多个 `-p` 选项，则 `dtrace` 将在所有命令退出后退出，并报告每个进程终止时的退出状态。第一个进程 ID 可供在命令行上（或使用 `-s` 选项通过 `$target` 宏变量）指定的任何 D 程序使用。有关宏变量的更多信息，请参阅第 15 章。

`-P` 指定要跟踪或列出的提供器名称（`-l` 选项）。其余的探测器说明字段（模块、函数和名称）将留为空白，并且与任何探测器相匹配，无论这些字段中的值是什么。可将可选 D 探测器子句用作 `-P` 参数的后缀。在命令行上，一次可指定多个 `-P` 选项。

`-q` 设置静默模式。`dtrace` 将取消显示消息（如按指定选项和 D 程序相匹配的探测器数目），并且将不列显列标题、CPU ID、探测器 ID，也不会在输出中插入新行。只有 D 程序语句（如 `trace()` 和 `printf()`）跟踪并格式化的数据才会显示在 `stdout` 中。

`-s` 编译指定的 D 程序源文件。如果 `-e` 选项存在，则将编译该程序，但不会启用任何检测过程。如果 `-l` 选项存在，则将编译该程序，并且会列出与其相匹配的一组探测器，但不会启用任何检测过程。如果 `-e` 和 `-l` 都不存在，则将启用 D 程序指定的检测过程并开始跟踪。

`-S` 显示 D 编译器中间代码。对于为每个 D 程序生成的中间代码，D 编译器会将它的报告生成到 `stderr`。

`-U` 在调用 `cpp`(1)（使用 `-C` 选项启用）时取消定义指定的 *name*。此选项将 `-U` 选项传递到每个 `cpp` 调用。

`-v` 设置详细模式。如果指定了 `-v` 选项，则 `dtrace` 将生成程序稳定性报告，说明所指定 D 程序的最低接口稳定性和相关性级别。DTrace 稳定性级别将在第 39 章中进一步地详细说明。

`-V` 报告 `dtrace` 支持的最高 D 编程接口版本。版本信息将列显到 `stdout`，然后 `dtrace` 命令将退出。有关 DTrace 版本控制功能的更多信息，请参见第 41 章。

`-w` 允许使用 `-s`、`-P`、`-m`、`-f`、`-n` 或 `-i` 选项在指定的 D 程序中执行破坏性操作。如果未指定 `-w` 选项，则 `dtrace` 将不允许编译或启用包含破坏性操作的 D 程序。破坏性操作在第 10 章中进行详细说明。

`-x` 启用或修改 DTrace 运行时选项或 D 编译器选项。第 16 章中列出了这些选项。将通过指定名称来启用布尔选项。将通过使用等号（`=`）分隔选项名称和值来设置包含值的选项。

`-X` 指定符合在调用 `cpp`(1)（使用 `-C` 选项启用）选项时应选择的 ISO C 标准的程度。根据参数字母的值，`-X` 选项参数会影响 `__STDC__` 宏的值和存在性：

- `a`（缺省值） ISO C 及 K&R 兼容性扩展，包含 ISO C 所需的语义更改。如果未指定 `-X`，则此模式为缺省模式。在将 `cpp` 与 `-Xa` 选项一起调用时，预定义宏 `__STDC__` 具有值 0。
- `c`（符合） 严格符合 ISO C（不包含 K&R C 兼容性扩展）。将 `cpp` 与 `-Xc` 选项一起调用时，预定义宏 `__STDC__` 具有值 1。

s (K&R C)　仅 K&R C。将 cpp 与 -Xs 选项一起调用时，不会定义宏 __STDC__。

t（转换）　ISO C 及 K&R C 兼容性扩展，不包含 ISO C 所需的语义更改。将 cpp 与 -Xt 选项一起调用时，__STDC__ 具有值 0。

因为 -X 选项仅影响 D 编译器调用 C 预处理程序的方式，所以从 D 的角度看来，-Xa 和 -Xt 选项是等价的。提供这两个选项都是为了简化 C 生成环境中设置的重复使用。

无论 -X 模式如何，始终会指定下列附加 C 预处理程序定义，并且它们在所有模式中有效：

- __sun
- __unix
- __SVR4
- __sparc（仅在 SPARC® 系统上）
- __sparcv9（仅当在 SPARC® 系统上编译 64 位程序时）
- __i386（仅当在 x86 系统上编译 32 位程序时）
- __amd64（仅当在 x86 系统上编译 64 位程序时）
- __*‘uname -s‘*_*‘uname -r‘*（例如 __SunOS_5_10）
- __SUNW_D=1
- __SUNW_D_VERSION=0x*MMmmmuuu*（其中，*MM* 是十六进制的主发行版本号，*mmm* 是十六进制的次发行版本号，而 *uuu* 是十六进制的微发行版本号；有关 DTrace 版本控制的更多信息，请参见第 41 章）

-Z　允许与任何探测器都不匹配的探测器说明。在未指定 -Z 选项时，如果在 D程序文件中（使用 -s 选项）或在命令行上（使用 -P、-m, -f、-n 或 -i 选项）指定了任何探测器说明，则 dtrace 会报告错误并退出。

操作数

可在 dtrace 命令行上指定零个或多个其他参数，来定义一组宏变量（$1、$2 等），以在使用 -s 选项指定的或在命令行上指定的任何 D 程序中使用。宏变量的用法将在第 15 章进一步地说明。

退出状态

dtrace 实用程序将返回下列退出值：

0 成功完成了指定的请求。对于 D 程序请求，退出状态 0 指示已成功编译程序，并且已成功启用探测器，或者已成功检索匿名状态。即使指定的跟踪请求遇到错误或删除，dtrace 也会返回 0。

1 发生了致命错误。对于 D 程序请求，退出状态 1 指示程序编译失败或者无法满足指定的请求。

2 指定的命令行选项或参数无效。

◆ ◆ ◆ 第 15 章

15 脚本

您可以使用 dtrace(1M) 实用程序在 D 程序之外创建与 shell 脚本类似的解释程序文件，这些文件可安装为可重用的交互式 DTrace 工具。D 编译器和 dtrace 命令提供一组由 D 编译器扩展的宏变量，使得创建 DTrace 脚本更容易。本章介绍对宏变量工具的引用和创建持久性脚本的技巧。

解释程序文件

与 shell 和实用程序（如 awk(1) 和 perl(1)）类似，可使用 dtrace(1M) 创建可执行的解释程序文件。解释程序文件的起始行格式如下所示：

#! *pathname* [*arg*]

其中，*pathname* 是解释程序的路径，*arg* 是单个可选参数。执行解释程序文件时，系统将调用指定的解释程序。如果解释程序文件中指定了 *arg*，则会将其作为参数传递到解释程序。解释程序文件本身的路径和执行该文件时指定的任何其他参数，随后将附加到解释程序参数列表。因此，将始终需要创建至少带有以下参数的 DTrace 解释程序文件：

```
#!/usr/sbin/dtrace -s
```

执行解释程序文件时，-s 选项的参数将成为解释程序文件本身的路径名。然后 dtrace 将读取、编译和执行此文件，作用与在 shell 中键入以下命令类似：

```
# dtrace -s interpreter-file
```

下列示例说明如何创建和执行 dtrace 解释程序文件。键入以下 D 源代码，并将其保存在名为 interp.d 的文件中：

```
#!/usr/sbin/dtrace -s

BEGIN
```

```
{

    trace("hello");

    exit(0);

}
```

将 interp.d 文件标记为可执行文件，并按以下方式执行该文件：

```
# chmod a+rx interp.d

# ./interp.d

dtrace: script ’./interp.d’ matched 1 probe

CPU     ID                    FUNCTION:NAME

  1      1                           :BEGIN   hello

#
```

请记住，#! 指令必须是文件的前两个字符，中间或前面都不得有空格。D 编译器在处理解释程序文件时会自动忽略此行。

dtrace 使用 getopt(3) 处理命令行选项，以便于将多个选项合并为单个解释程序参数。例如，要将 -q 选项添加到前面的示例中，可以将解释程序指令更改为：

```
#!/usr/sbin/dtrace -qs
```

如果指定多个选项字母，则必须始终用 -s 选项结束布尔选项列表，以便将下一个参数（解释程序文件名）作为与 -s 选项对应的参数进行处理。

如果需要在解释程序文件中指定多个要求参数的选项，则不能将所有选项和参数放入单个解释程序参数中。 请改为使用 #pragma D option 指令语法设置这些选项。 所有 dtrace 命令行选项都有可以使用的 #pragma 等效选项，如第 16 章中所示。

宏变量

D 编译器定义了一组内置的宏变量，可以在编写 D 程序或解释程序文件时使用这些变量。宏变量是以美元符号 ($) 作为前缀的标识符，并在处理输入文件时由 D 编译器扩展一次。D 编译器提供了以下宏变量：

表 15–1 D 宏变量

名称	说明	参考
`$[0-9]+`	宏参数	请参见第 196 页中的 “宏参数”
`$egid`	有效组 ID	`getegid`(2)
`$euid`	有效用户 ID	`geteuid`(2)
`$gid`	实际组 ID	`getgid`(2)
`$pid`	进程 ID	`getpid`(2)
`$pgid`	进程组 ID	`getpgid`(2)
`$ppid`	父进程 ID	`getppid`(2)
`$projid`	项目 ID	`getprojid`(2)
`$sid`	会话 ID	`getsid`(2)
`$target`	目标进程 ID	请参见第 198 页中的 “目标进程 ID”
`$taskid`	任务 ID	`gettaskid`(2)
`$uid`	实际用户 ID	`getuid`(2)

除了 `$[0-9]+` 宏参数和 `$target` 宏变量以外，所有宏变量都扩展为与系统属性（如进程 ID 和用户 ID）对应的整数。变量扩展为与当前 `dtrace` 进程本身或正在运行 D 编译器的任何进程关联的属性值。

通过在解释程序文件中使用宏变量，可以创建持久性的 D 程序，每次使用这些程序时无需对其进行编辑。例如，要统计除 `dtrace` 命令执行的系统调用以外的所有系统调用，可以使用包含 `$pid` 的以下 D 程序子句：

```
syscall:::entry

/pid != $pid/

{

    @calls = count();

}
```

即使每次调用 dtrace 命令都将得到不同的进程 ID，此子句也始终会生成需要的结果。

D 程序中可使用整数、标识符或字符串的任何位置均可使用宏变量。在解析输入文件时，将只扩展宏变量一次（即，不递归扩展）。每个扩展的宏变量形成一个单独的输入标记，不能与其他文本串联生成单个标记。例如，如果 $pid 扩展为值 456，则 D 代码：

```
123$pid
```

将扩展为两个相邻的标记 123 和 456，这将导致语法错误，而不会生成单个整数标记 123456。

宏变量在程序子句的开始位置扩展，并与 D 探测器说明内部的相邻文本串联。例如，以下子句使用 DTrace pid 提供器检测 dtrace 命令：

```
pid$pid:libc.so:printf:entry

{

    ...

}
```

宏变量在每个探测器说明字段中仅扩展一次；它们不得包含探测器说明分界符 (:)。

宏参数

D 编译器还提供了一组与任何附加参数操作数（指定为 dtrace 命令调用一部分）对应的宏变量。可使用内置名称 $0 表示 D 程序文件的名称或 dtrace 命令、$1 表示第一个附加操作数、$2 表示第二个操作数，依此类推，来访问这些宏参数。如果使用了 dtrace -s 选项，则 $0 将扩展为与此选项配合使用的输入文件的名称的值。对于在命令行上指定的 D 程序，$0 扩展为用于执行 dtrace 本身的 argv[0] 的值。

宏参数可以扩展为整数、标识符或字符串，具体取决于相应文本的格式。与所有宏变量一样，D 程序中可使用整数、标识符和字符串标记的任何位置均可使用宏参数。以下所有示例都可以构成带有相应宏参数值的有效 D 表达式：

```
execname == $1    /* with a string macro argument */

x += $1           /* with an integer macro argument */

trace(x->$1)      /* with an identifier macro argument */
```

宏参数可用于创建 dtrace 解释程序文件，这些文件以类似于实际 Solaris 命令的方式操作，并使用用户或其他工具指定的信息来修改其行为。例如，以下 D 解释程序文件跟踪按特定进程 ID 执行的 write(2) 系统调用：

```
#!/usr/sbin/dtrace -s


syscall::write:entry

/pid == $1/

{

}
```

如果将此解释程序文件设置为可执行文件，则可以使用解释程序文件的附加命令行参数来指定 `$1` 的值：

```
# chmod a+rx ./tracewrite

# ./tracewrite 12345
```

产生的命令调用统计由进程 ID 12345 执行的每个 `write`(2) 系统调用。

如果 D 程序引用的宏参数不是在命令行上提供的，则会列显相应的错误消息，且程序将编译失败：

```
# ./tracewrite

dtrace: failed to compile script ./tracewrite: line 4:

  macro argument $1 is not defined
```

如果设置了 `defaultargs` 选项，则 D 程序可以引用未指定的宏参数。如果设置 `defaultargs`，则未指定的参数的值将为 `0`。有关 D 编译器选项的更多信息，请参见第 16 章。如果在命令行上指定了 D 程序未引用的附加参数，则 D 编译器也将生成错误消息。

宏参数值必须与整数、标识符或字符串的格式匹配。如果参数与这些格式中的任何一种都不匹配，则 D 编译器将报告相应的错误消息。将字符串宏参数指定到 DTrace 解释程序文件时，使用一对额外的单引号将该参数引起来可避免 shell 解释双引号和字符串内容：

```
# ./foo '"a string argument"'
```

如果要将 D 宏参数解释为字符串标记（即使这些参数与整数或标识符的格式匹配），请使用两个前导美元符号（如 `$$1`）作为宏变量或参数名称的前缀，以强制 D 编译器解释参数值（就像参数值是由双引号引起来的字符串一样）。无论是使用 `$`*arg* 还是 `$$`*arg* 格式的宏来引用 D 字符串转义序列，任何字符串宏参数中的所有常用 D 字符串转义序列（参见表 2–5）都将被扩展。如果已设置 `defaultargs` 选项，则使用 `$$`*arg* 格式引用的未指定参数的值为空字符串 (`""`)。

目标进程 ID

使用 $target 宏变量可创建脚本，这些脚本可应用于所关注的特定用户进程（在 dtrace 命令行上使用 -p 选项选择的进程或使用 -c 选项创建的进程）。在创建或抓取进程且 $target 变量扩展为第一个此类进程的整数进程 ID 后，将编译在命令行上指定或使用 -s 选项指定的 D 程序。例如，以下 D 脚本可用于确定由特定主题进程执行的系统调用的分布：

```
syscall:::entry

/pid == $target/

{

    @[probefunc] = count();

}
```

要确定由 date(1) 命令执行的系统调用的数量，请将脚本保存在文件 syscall.d 中并执行以下命令：

```
# dtrace -s syscall.d -c date

dtrace: script 'syscall.d' matched 227 probes

Fri Jul 30 13:46:06 PDT 2004

dtrace: pid 109058 has exited



  gtime                                                             1

  getpid                                                            1

  getrlimit                                                         1

  rexit                                                             1

  ioctl                                                             1

  resolvepath                                                       1

  read                                                              1

  stat                                                              1

  write                                                             1
```

```
munmap                                                    1
close                                                     2
fstat64                                                   2
setcontext                                                2
mmap                                                      2
open                                                      2
brk                                                       4
```

◆ ◆ ◆ 第 16 章

选项和可调参数

为了允许进行自定义，DTrace 为其使用者提供了多种重要的自由度。为最大限度地降低需要特定调整的可能性，DTrace 将通过使用合理的缺省值和灵活的缺省策略来实现。但是，可能会出现要求基于各个使用者调整 DTrace 行为的情况。本章介绍 DTrace 选项和可调参数，以及可用于修改它们的接口。

使用者选项

通过设置或启用这些选项，可对 DTrace 进行调整。下表中说明了可用的选项。对于一些选项，`dtrace`(1M) 提供了对应的命令行选项。

表 16–1 DTrace 使用者选项

选项名称	值	`dtrace`(1M) 别名	说明	参见章节
`aggrate`	*time*		聚合读取的速率	第 9 章
`aggsize`	*size*		聚合缓冲区大小	第 9 章
`bufresize`	`auto` 或 `manual`		缓冲区调整大小策略	第 11 章
`bufsize`	*size*	`-b`	主体缓冲区大小	第 11 章
`cleanrate`	*time*		清除速率	第 13 章
`cpu`	*scalar*	`-c`	启用跟踪的 CPU	第 11 章
`defaultargs`	—		允许引用未指定的宏参数	第 15 章
`destructive`	—	`-w`	允许破坏性操作	第 10 章
`dynvarsize`	*size*		动态变量空间大小	第 3 章

表 16–1 DTrace 使用者选项 （续）

选项名称	值	dtrace(1M) 别名	说明	参见章节
flowindent	—	-F	缩进函数输入并加前缀 -> ；取消缩进函数返回并加前缀 <- 。	第 14 章
grabanon	—	-a	声明匿名状态	第 36 章
jstackframes	*scalar*		jstack() 的缺省栈帧数	第 10 章
jstackstrsize	*scalar*		jstack() 的缺省字符串空间大小	第 10 章
nspec	*scalar*		随机数	第 13 章
quiet	—	-q	仅输出显式跟踪的数据	第 14 章
specsize	*size*		随机缓冲区大小	第 13 章
strsize	*size*		字符串大小	第 6 章
stackframes	*scalar*		栈帧数	第 10 章
stackindent	*scalar*		缩进 stack() 和 ustack() 输出时要使用的空格字符数	第 10 章
statusrate	*time*		状态检查的速率	
switchrate	*time*		缓冲区切换的速率	第 11 章
ustackframes	*scalar*		用户栈帧数	第 10 章

对于表示大小的值，可以根据需要指定 k、m、g 或 t 作为后缀，以分别表示千字节、兆字节、千兆字节和兆兆字节。对于表示时间的值，可以根据需要指定 ns、us、ms、s 或 hz 作为后缀，以分别表示纳秒、微秒、毫秒、秒和每秒次数。

修改选项

在 D 脚本中，可以使用 #pragma D，并后跟字符串 option 和选项名称来设置选项。如果选项接受一个值，则该选项名称后应跟等号 (=) 和选项值。以下示例都是有效的选项设置：

```
#pragma D option nspec=4

#pragma D option grabanon

#pragma D option bufsize=2g
```

```
#pragma D option switchrate=10hz

#pragma D option aggrate=100us

#pragma D option bufresize=manual
```

dtrace(1M) 命令还可在命令行中接受选项设置，作为 -x 选项的参数。例如：

```
# dtrace -x nspec=4 -x grabanon -x bufsize=2g \

    -x switchrate=10hz -x aggrate=100us -x bufresize=manual
```

如果指定的选项无效，dtrace 将指示该选项名称无效并退出：

```
# dtrace -x wombats=25

dtrace: failed to set option -x wombats: Invalid option name

#
```

同样，如果给定选项的选项值无效，dtrace 将指示该值无效：

```
# dtrace -x bufsize=100wombats

dtrace: failed to set option -x bufsize: Invalid value for specified option

#
```

如果多次设置某选项，则后续设置将覆写先前的设置。某些选项（如 grabanon）只能设置。此类选项表现为，您可以对其进行设置，但以后将无法取消该设置。

为启用匿名而设置的选项将由声明匿名状态的 DTrace 使用者接受。有关启用匿名跟踪的信息，请参见第 36 章。

◆ ◆ ◆ 第 17 章

17 dtrace 提供器

dtrace 提供器提供了多个与 DTrace 本身相关的探测器。您可以使用这些探测器在开始跟踪前初始化状态，在完成跟踪后处理状态，并在其他探测器中处理意外的执行错误。

BEGIN 探测器

BEGIN 探测器在任何其他探测器之前触发。在所有 BEGIN 子句完成之前，将不会再触发任何其他探测器。此探测器可用于初始化其他探测器中需要的任何状态。以下示例说明如何使用 BEGIN 探测器初始化用于在 mmap(2) 保护位和文本说明之间进行映射的关联数组：

```
BEGIN

{

    prot[0] = "---";

    prot[1] = "r--";

    prot[2] = "-w-";

    prot[3] = "rw-";

    prot[4] = "--x";

    prot[5] = "r-x";

    prot[6] = "-wx";

    prot[7] = "rwx";

}
```

```
syscall::mmap:entry

{

    printf("mmap with prot = %s", prot[arg2 & 0x7]);

}
```

BEGIN 探测器在未指定的上下文中触发。这表示，stack() 或 ustack() 的输出以及特定于上下文的变量（例如，execname）的值都是任意的。不应依赖于这些值，或者解释这些值来推断任何有意义的信息。 BEGIN 探测器未定义任何参数。

END 探测器

END 探测器在所有其他探测器之后触发。在所有其他探测器子句完成之前，将不会触发此探测器。此探测器可用于处理已收集的状态，或者格式化输出。因此printa() 操作通常在 END 探测器中使用。可以同时使用 BEGIN 和 END 探测器来度量跟踪花费的总时间：

```
BEGIN

{

    start = timestamp;

}



/*

 * ... other tracing actions...

 */



END

{

    printf("total time: %d secs", (timestamp - start) / 1000000000);

}
```

有关 END 探测器的其他常见用法，请参见第 122 页中的 “数据标准化” 和第 169 页中的 “printa()” 。

与 BEGIN 探测器一样，END 探测器没有定义任何参数。触发 END 探测器的上下文是任意的，不应依赖于该上下文。

跟踪时，如果 bufpolicy 选项已设置为 fill，则会保留足够的内存，以便容纳 END 探测器中跟踪的任何记录。有关详细信息，请参见第 160 页中的 “fill 策略和 END 探测器” 。

注 – exit() 操作将导致跟踪停止并触发 END 探测器。但是，调用 exit() 操作和触发 END 探测器之间会有一定延迟。在此延迟期间，将不会触发任何探测器。 在探测器调用 exit() 操作之后，在 DTrace 使用者确定已调用 exit() 并停止跟踪之前，将不会触发 END 探测器。可以使用 statusrate 选项设置检查退出状态的速率。有关更多信息，请参见第 16 章。

ERROR 探测器

如果在执行 DTrace 探测器的子句时发生运行时错误，将触发 ERROR 探测器。例如，如果子句尝试废除引用 NULL 指针，则将触发 ERROR 探测器，如下例所示。

示例 17–1 error.d: 记录错误

```
BEGIN

{

    *(char *)NULL;

}



ERROR

{

    printf("Hit an error!");

}
```

运行此程序时，将会看到与以下示例类似的输出。

```
# dtrace -s ./error.d

dtrace: script ’./error.d’ matched 2 probes
```

```
CPU     ID                    FUNCTION:NAME

  2      3                           :ERROR Hit an error!

dtrace: error on enabled probe ID 1 (ID 1: dtrace:::BEGIN): invalid address

(0x0) in action #1 at DIF offset 12

dtrace: 1 error on CPU 2
```

该输出说明已触发 ERROR 探测器，还说明 dtrace(1M) 报告了错误。dtrace 可以独立启用 ERROR 探测器，从而允许该探测器报告错误。使用 ERROR 探测器时，可以创建您自己的自定义错误处理。

ERROR 探测器的参数如下所示：

arg1	引发错误的探测器的已启用探测器标识符 (enabled probe identifier, EPID)
arg2	导致故障的操作的索引
arg3	该操作的 DIF 偏移，如果不适用则为 -1
arg4	故障类型
arg5	特定于故障类型的值

下表说明了各种故障类型以及 arg5 针对每种故障类型的值：

arg4 值	说明	arg5 含义
DTRACEFLT_UNKNOWN	未知故障类型	无
DTRACEFLT_BADADDR	访问已取消映射或无效的地址	已访问的地址
DTRACEFLT_BADALIGN	未正确配置的内存访问	已访问的地址
DTRACEFLT_ILLOP	非法或无效操作	无
DTRACEFLT_DIVZERO	整数被零除	无
DTRACEFLT_NOSCRATCH	没有满足临时分配的足够临时空间	无
DTRACEFLT_KPRIV	尝试使用不充分的权限访问内核地址或属性	已访问的地址，如果不适用则为 0

arg4 值	说明	arg5 含义
DTRACEFLT_UPRIV	尝试使用不充分的权限访问用户地址或属性	已访问的地址，如果不适用则为 0
DTRACEFLT_TUPOFLOW	DTrace 内部参数栈溢出	无

如果 ERROR 探测器本身采用的操作导致错误，则将默认删除该错误－不会递归调用 ERROR 探测器。

稳定性

dtrace 提供器使用 DTrace 的稳定性机制说明其稳定性，如下表所示。有关稳定性机制的更多信息，请参见第 39 章。

元素	名称稳定性	数据稳定性	相关性类
提供器	稳定	稳定	公用
模块	专用	专用	未知
函数	专用	专用	未知
名称	稳定	稳定	公用
参数	稳定	稳定	公用

◆ ◆ ◆ 第 18 章

18 lockstat 提供器

lockstat 提供器提供了可用于理解锁定争用统计信息，或者实际了解锁定行为任何方面的探测器。 lockstat(1M) 命令实际上是 DTrace 使用者，它使用 lockstat 提供器收集其原始数据。

概述

lockstat 提供器提供了两种类型的探测器：争用事件探测器和暂挂事件探测器。

争用事件探测器对应于同步元语的争用，并在强制线程等待资源可用时触发。 通常会将 Solaris 优化为不出现争用情况，从而不会发生延迟的争用。应使用这些探测器来了解发生争用的情况。 因为争用相对较少，所以启用争用事件探测器通常不会实际影响性能。

暂挂事件探测器对应于获取、释放或在其他方面处理同步元语。这些探测器可用于回答有关处理同步元语的方式的任意问题。因为 Solaris 频繁（在一个繁忙系统中，每个 CPU 每秒几百万次）获取和释放同步元语，所以启用暂挂事件探测器产生的探测影响要远远大于启用争用事件探测器。 虽然启用这些探测器会产生很强的探测效果，但这些效果并不是负面的；因此仍然可以在产品化的系统上放心启用这些探测器。

lockstat 提供器提供了对应于 Solaris 中的各种同步元语的探测器；这些元语和与其对应的探测器将在本章的剩余部分中讨论。

自适应锁定探测器

自适应锁定强制临界段互斥，在内核的大多数上下文中都可以获取该锁定。 因为自适应锁定的上下文限制极少，所以它们组成了 Solaris 内核中绝大多数的同步元语。 这些锁定的与争用相关的行为是自适应的：当线程尝试获取暂挂的自适应锁定时，它将确定拥有的线程当前是否正在 CPU 中运行。如果属主正在另一个 CPU 中运行，则获取线程将旋转。如果属主未运行，则获取线程将阻塞。

表 18–1 中介绍了与自适应锁定有关的四种 lockstat 探测器。对于每种探测器，arg0 都包含指向代表自适应锁定的 kmutex_t 结构的探测器。

表 18–1 自适应锁定探测器

adaptive-acquire	获取自适应锁定后立即触发的暂挂事件探测器。
adaptive-block	在暂挂的自适应锁定上阻塞的线程重新唤醒并获取互斥之后，将触发的争用事件探测器。 如果两种探测器都已启用，则 adaptive-block 将在 adaptive-acquire 之前触发。 对于单个锁定获取，最多将触发 adaptive-block 和 adaptive-spin 之一。 adaptive-block 的arg1 包含休眠时间（以纳秒为单位）。
adaptive-spin	在暂挂的自适应互斥上旋转的线程成功获取互斥后将触发的争用事件探测器。 如果两种探测器都已启用，则 adaptive-spin 将在 adaptive-acquire 之前触发。 对于单个锁定获取，最多将触发 adaptive-spin 和 adaptive-block 之一。 adaptive-spin 的arg1 包含旋转计数： 获取锁定之前，通过旋转循环重复的次数。 独立的旋转计数本身没有什么意义，但可用于比较旋转次数。
adaptive-release	释放自适应锁定之后将立即触发的暂挂事件探测器。

旋转锁定探测器

在内核的某些上下文中不能阻塞线程，如高级中断上下文和任何处理分发程序状态的上下文。在这些上下文中，此限制会禁止使用自适应锁定。 在这些上下文中将改用旋转锁定来影响临界段的互斥。 顾名思义，在存在争用的情况下，这些锁定的行为是旋转，直到拥有线程释放锁定。表 18–2 中介绍了与旋转锁定有关的三种探测器。

表 18–2 旋转锁定探测器

spin-acquire	获取旋转锁定之后将立即触发的暂挂事件探测器。
spin-spin	在暂挂的旋转锁定上旋转的线程成功获取旋转锁定后将触发的争用事件探测器。 如果两种探测器都已启用，则 spin-spin 将在 spin-acquire 之前触发。 spin-spin 的 arg1 包含旋转计数： 获取锁定之前，通过旋转循环重复的次数。 独立的旋转计数本身没有什么意义，但可用于比较旋转次数。
spin-release	释放旋转锁定之后将立即触发的暂挂事件探测器。

自适应锁定比旋转锁定更常见。以下脚本显示了两种锁定类型的总计，以提供支持此观察的数据。

```
lockstat:::adaptive-acquire

/execname == "date"/

{

    @locks["adaptive"] = count();
```

```
}


lockstat:::spin-acquire

/execname == "date"/

{

    @locks["spin"] = count();

}
```

在一个窗口中运行此脚本，在另一个窗口中运行 date(1) 命令。 终止 DTrace 脚本时，将会看到与以下示例类似的输出：

```
# dtrace -s ./whatlock.d

dtrace: script ’./whatlock.d’ matched 5 probes

^C

spin                                                    26

adaptive                                              2981
```

如此输出所指示的那样，在运行 date 命令时获取的锁定，超过 99% 都为自适应锁定。 令人吃惊的是，在执行像 date 这样简单的操作时获取了如此多的锁定。 大量的锁定是具有极大可伸缩性的系统（如 Solaris 内核）所需的细分锁定的自然产物。

线程锁定

线程锁定是一种特殊类型的旋转锁定，用于锁定线程，以更改线程的状态。 线程锁定暂挂事件可用作旋转锁定暂挂事件探测器（即，spin-acquire 和 spin-release），但争用事件具有各自特定于线程锁定的探测器。表 18–3 中介绍了线程锁定暂挂事件探测器。

表 18–3 线程锁定探测器

thread-spin	线程在线程锁定上旋转之后将触发的争用事件探测器。与其他争用事件探测器一样，如果争用事件探测器和暂挂事件探测器都已启用，则 thread-spin 将在 spin-acquire 之前触发。 但与其他争用事件探测器不一样，thread-spin 在实际获取锁定之前触发。 因此，多个 thread-spin 探测器触发可能与单个 spin-acquire 探测器触发对应。

读取器/写入器锁定探测器

读取器/写入器锁定强制执行允许临界段中有多个读取器或单个写入器的策略，但不允许同时存在二者。这些锁定通常用于搜索操作比修改操作更频繁且临界段中具有足够的时间的结构。如果临界段时间比较短，则读取器/写入器锁定将在用于实现锁定的共享内存中隐式串行化，从而使它们无法比自适应锁定提供更大的优势。有关读取器/写入器锁定的更多详细信息，请参见 rwlock(9F)。

表 18–4 中介绍了与读取器/写入器锁定有关的探测器。对于每种探测器，arg0 都包含指向代表自适应锁定的 krwlock_t 结构的指针。

表 18–4 读取器/写入器锁定探测器

rw-acquire	获取读取器/写入器锁定之后将立即触发的暂挂事件探测器。如果作为读取器获取锁定，则 arg1 将包含常量 RW_READER，如果作为写入器获取锁定，则将包含常量 RW_WRITER。
rw-block	在暂挂的读取器/写入器锁定上阻塞的线程重新唤醒并获取锁定之后，将触发的争用事件探测器。arg1 包含当前线程要获取锁定必须休眠的时间长度（以纳秒为单位）。如果作为读取器获取锁定，则 arg2 将包含常量 RW_READER，如果作为写入器获取锁定，将包含常量 RW_WRITER。arg3 和 arg4 包含有关阻塞原因的更多信息。在当前线程阻塞的情况下，只有在作为写入器暂挂锁定时，arg3 才不为零。arg4 包含当前线程阻塞时的读取器计数。如果 rw-block 和 rw-acquire 探测器都已启用，则 rw-block 将在 rw-acquire 之前触发。
rw-upgrade	在线程已成功将读取器/写入器锁定从读取器升级到写入器之后，将触发的暂挂事件探测器。升级没有关联的争用事件，因为仅可以通过非阻塞接口 rw_tryupgrade(TRYUPGRADE.9F) 进行升级。
rw-downgrade	在线程将其读取器/写入器锁定的拥有权从写入器降级到读取器之后，将触发的暂挂事件探测器。降级没有关联的争用事件，因为降级在没有争用的情况下始终会成功。
rw-release	释放读取器/写入器锁定之后，将立即触发的暂挂事件探测器。如果作为读取器暂挂释放的锁定，则 arg1 将包含常量 RW_READER，如果作为写入器暂挂释放的锁定，则将包含常量 RW_WRITER。由于升级和降级，获取锁定时，锁定可能并没有释放。

稳定性

lockstat 提供器使用 DTrace 的稳定性机制* 说明其稳定性，如下表所示。有关稳定性机制的更多信息，请参见第 39 章。

元素	名称稳定性	数据稳定性	相关性类
提供器	发展中	发展中	公用

元素	名称稳定性	数据稳定性	相关性类
模块	专用	专用	未知
函数	专用	专用	未知
名称	发展中	发展中	公用
参数	发展中	发展中	公用

◆ ◆ ◆ 第 19 章

19 profile 提供器

profile 提供器提供了与按所指定固定时间间隔触发、基于时间的中断关联的探测器。这些不固定探测器不与任何特定执行点关联，而是与异步中断事件关联。这些探测器可用于按单位时间对系统状态的某些方面进行采样，然后使用这些样本来推断系统行为。如果采样速率较高或采样时间较长，则推断可能更准确。使用 DTrace 操作时，可以使用 profile 提供器对系统中的任何内容进行实际采样。例如，您可对当前线程的状态、CPU 的状态或当前的计算机指令进行采样。

profile-*n* 探测器

profile-*n* 探测器在每个 CPU 中以高中断级别按固定的时间间隔触发。探测器的触发时间间隔以 *n* 值表示：中断源将每秒触发 *n* 次。*n* 也可带可选的时间前缀，在此情况下，*n* 将解释为由后缀表示的单位。表 19–1 中列出了有效的后缀和这些后缀表示的单位。

表 19–1 有效时间后缀

后缀	时间单位
nsec 或 ns	纳秒
usec 或 us	微秒
msec 或 ms	毫秒
sec 或 s	秒
min 或 m	分钟
hour 或 h	小时
day 或 d	天
hz	赫兹（每秒频率）

以下示例创建一个按 97 赫兹触发，以便对当前正在运行的进程进行采样的探测器：

```
#pragma D option quiet


profile-97

/pid != 0/

{

    @proc[pid, execname] = count();

}


END

{

    printf("%-8s %-40s %s\n", "PID", "CMD", "COUNT");

    printa("%-8d %-40s %@d\n", @proc);

}
```

运行上例一小段时间将产生与以下示例类似的输出：

```
# dtrace -s ./prof.d

^C

PID      CMD                                      COUNT

223887   sh                                       1

100360   httpd                                    1

100409   mibiisa                                  1

223887   uname                                    1

218848   sh                                       2

218984   adeptedit                                2

100224   nscd                                     3
```

```
3        fsflush                                 4

2        pageout                                 6

100372   java                                    7

115279   xterm                                   7

100460   Xsun                                    7

100475   perfbar                                 9

223888   prstat                                  15
```

也可使用 `profile-`*n* 提供器对有关正在运行进程的信息进行采样。以下示例 D 脚本使用一个 1,001 赫兹的 profile 探测器对所指定进程的当前优先级进行采样：

```
profile-1001

/pid == $1/

{

    @proc[execname] = lquantize(curlwpsinfo->pr_pri, 0, 100, 10);

}
```

要查看运行中的此示例脚本，请在一个窗口中键入下列命令：

```
$ echo $$

494621

$ while true ; do let i=0 ; done
```

在另一个窗口中运行 D 脚本一小段时间：

```
# dtrace -s ./profpri.d 494621

?? dtrace: script ’./profpri.d’ matched 1 probe

^C

ksh

           value  ------------- Distribution ------------- count
```

```
               < 0 |                                       0
                 0 |@@@@@@@@@@@@@@@@@@@@                  7443
                10 |@@@@@@                                2235
                20 |@@@@                                  1679
                30 |@@@                                   1119
                40 |@                                     560
??                50 |@                                    554
                60 |                                       0
```

此输出显示了分时调度类的偏差。由于 shell 进程在 CPU 上旋转，因此其优先级经常会被系统降低。如果该 shell 进程运行不频繁，则其优先级将更高。要查看此结果，请在此旋转的 shell 中键入 Ctrl-C 组合键，然后再次运行脚本：

```
# dtrace -s ./profpri.d 494621

 dtrace: script ’./profpri.d’ matched 1 probe
```

现在于 shell 中键入一些字符。终止 DTrace 脚本时，将会显示与以下示例类似的输出：

```
ksh

           value  ------------- Distribution ------------- count
              40 |                                         0
              50 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 14
              60 |                                         0
```

由于等待用户输入时 shell 进程进入休眠状态而不是在 CPU 中旋转，所以当真正运行时，它将以较高的优先级运行。

tick-*n* 探测器

与 profile-*n* 探测器一样，tick-*n* 探测器以高中断级别按固定时间间隔触发。但是，与在每个 CPU 中都触发的 profile-*n* 探测器不同，tick-*n* 探测器仅按时间间隔在一个 CPU 中触发。实际的 CPU 可能随时间变化。与 profile-*n* 探测器一样，*n* 缺省为每秒的速率，但是也可以带有可选的时间后缀。tick-*n* 探测器具有多种用途（例如提供某些定期输出或执行定期操作）。

参数

profile 探测器的参数如下所示：

arg0	触发探测器时内核中的程序计数器 (program counter, PC)，如果在触发探测器时内核中未执行当前进程，则为 0
arg1	触发探测器时用户级进程中的程序计数器，如果在触发探测器时内核中正在执行当前进程，则为 0

从说明可以判断出，如果 arg0 为非零值，则 arg1 为零；如果 arg0 为零，则 arg1 为非零值。因此，可以使用 arg0 和 arg1 来区分用户级和内核级，如以下简单示例所示：

```
profile-1ms

{

    @ticks[arg0 ? "kernel" : "user"] = count();

}
```

计时器分辨率

profile 提供器使用操作系统中的任意分辨率间隔计时器。在不支持真正基于任意分辨率时间中断的体系结构中，此频率受系统时钟频率的限制，由 hz 内核变量指定。此类体系结构中频率高于 hz 的探测器将每隔 1/hz 秒触发若干次。例如，这种体系结构中一个 1000 赫兹的 profile 探测器（hz 设置为 100）将每隔 10 毫秒快速连续触发 10 次。在支持任意分辨率的平台上，1000 赫兹的 profile 探测器将准确地每隔 1 毫秒触发一次。

以下示例测试给定体系结构的分辨率：

```
profile-5000

{
```

```
    /*
     * We divide by 1,000,000 to convert nanoseconds to milliseconds, and
     * then we take the value mod 10 to get the current millisecond within
     * a 10 millisecond window.  On platforms that do not support truly
     * arbitrary resolution profile probes, all of the profile-5000 probes
     * will fire on roughly the same millisecond.  On platforms that
     * support a truly arbitrary resolution, the probe firings will be
     * evenly distributed across the milliseconds.
     */
    @ms = lquantize((timestamp / 1000000) % 10, 0, 10, 1);
}


tick-1sec
/i++ >= 10/
{
    exit(0);
}
```

在支持任意分辨率 `profile` 探测器的体系结构中，运行此示例脚本将产生均匀分布：

```
# dtrace -s ./restest.d
 dtrace: script ’./restest.d’ matched 2 probes
CPU     ID                    FUNCTION:NAME
  0  33631                        :tick-1sec
```

```
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@                                      10760
               1 |@@@@                                     10842
               2 |@@@@                                     10861
               3 |@@@                                      10820
               4 |@@@                                      10819
               5 |@@@                                      10817
               6 |@@@@                                     10826
               7 |@@@@                                     10847
               8 |@@@@                                     10830
               9 |@@@@                                     10830
```

在不支持任意分辨率 `profile` 探测器的体系结构中，运行此示例脚本将产生不均匀分布：

```
# dtrace -s ./restest.d
 dtrace: script ’./restest.d’ matched 2 probes
 CPU     ID                    FUNCTION:NAME
  0  28321                       :tick-1sec


           value  ------------- Distribution ------------- count
               4 |                                         0
               5 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  107864
               6 |                                         424
               7 |                                         255
```

```
8 |                                     496
9 |                                     0
```

在这些体系结构中，可在 /etc/system 中手动调整 hz，以提高有效的配置文件分辨率。

当前，UltraSPARC (sun4u) 的所有变体均支持任意分辨率 profile 探测器。x86 体系结构 (i86pc) 的许多变体也支持任意分辨率 profile 探测器，但是一些较旧的变体不支持。

探测器创建

与其他提供器不同，profile 提供器根据需要动态创建探测器。因此，所需的 profile 探测器可能并未在所有探测器列表中显示（例如，通过使用 dtrace -l -P profile），但在显式启用该探测器时将会创建它。

在支持任意分辨率 profile 探测器的体系结构中，时间间隔太短将导致计算机连续产生基于时间的中断，从而拒绝计算机上的服务。为了防止出现这种情况，profile 提供器将自动拒绝创建会导致时间间隔小于两百微秒的任何探测器。

稳定性

profile 提供器使用 DTrace 的稳定性机制说明其稳定性，如下表所示。有关稳定性机制的更多信息，请参见第 39 章。

元素	名称稳定性	数据稳定性	相关性类
提供器	发展中	发展中	公用
模块	不稳定	不稳定	未知
函数	专用	专用	未知
名称	发展中	发展中	公用
参数	发展中	发展中	公用

◆ ◆ ◆ 第 20 章

20 fbt 提供器

本章介绍函数边界跟踪 (Function Boundary Tracing, FBT) 提供器，该提供器提供了与进入和返回 Solaris 内核中的大多数函数相关的探测器。函数是程序文本的基本组成单元。 在设计完善的系统中，每个函数会对指定的对象或一系列类似对象执行独立和定义完善的操作。所以，即使在最小型的 Solaris 系统中，FBT 也将提供 20,000 个探测器。

与其他 DTrace 提供器类似，在没有显式启用 FBT 提供器时，它不会产生探测效果。启用时，FBT 仅在被探测的函数中产生探测效果。 虽然 FBT 实现高度特定于指令集体系结构，但在 SPARC 和 x86 平台上都已实现 FBT。对于每一个指令集，都有少量函数不调用其他函数，且由无法通过 FBT 检测的编译器（所谓的叶函数）高度优化。DTrace 中不存在这些函数的探测器。

要有效使用 FBT 探测器，需要对操作系统的实现有所了解。所以，建议您仅在开发内核软件或者其他提供器不满足要求时，才使用 FBT。其他 DTrace 提供器（包括 `syscall`、`sched`、`proc` 和 `io`）都可用于回答大多数系统分析问题，不需要了解操作系统实现的知识。

探测器

在内核中大多数函数的边界，FBT 都提供了探测器。 进入函数和从函数返回都会跨越函数边界。因此，FBT 为内核中的每个函数提供了两个探测器：一个用于进入函数时，一个用于从函数返回时。 这两个探测器的名称分别为 `entry` 和 `return`。函数名称和模块名称都指定为探测器的一部分。所有 FBT 探测器都指定了函数名称和模块名称。

探测器参数

entry 探测器

`entry` 探测器的参数与相应操作系统内核函数的参数相同。可以使用 `args[]` 数组以键入的方式访问这些参数。 可以使用 `arg0` .. `arg`*n* 变量，以访问 `int64_t` 参数的方式访问这些参数。

return 探测器

虽然给定的函数仅有一个进入点，但它可能有许多不同的位置可返回到其调用方。 通常，您可能只关心函数所返回的值或函数是否返回，而不关心所采用的具体返回路径。 因此，FBT 将函数的多个返回位置收集到单个 return 探测器中。 如果您对确切的返回路径感兴趣，可以检查 return 探测器的 args[0] 值，该值表示函数文本中返回指令的偏移量（以字节为单位）。

如果函数具有返回值，则该返回值存储在args[1] 中。如果函数不具有返回值，则不会定义 args[1]。

示例

可以使用 FBT 轻松地了解内核的实现。以下示例脚本记录来自任何 xclock 进程的第一个 ioctl(2)，以及一直到内核的后续代码路径：

```
/*
 * To make the output more readable, we want to indent every function entry
 * (and unindent every function return).  This is done by setting the
 * "flowindent" option.
 */
#pragma D option flowindent


syscall::ioctl:entry
/execname == "xclock" && guard++ == 0/
{
    self->traceme = 1;
    printf("fd: %d", arg0);
}


fbt:::
```

```
/self->traceme/

{}


syscall::ioctl:return

/self->traceme/

{

    self->traceme = 0;

    exit(0);

}
```

运行此脚本将会生成与以下示例类似的输出：

```
# dtrace -s ./xioctl.d

dtrace: script ’./xioctl.d’ matched 26254 probes

CPU FUNCTION

  0  => ioctl                                  fd: 3

  0    -> ioctl

  0      -> getf

  0        -> set_active_fd

  0        <- set_active_fd

  0      <- getf

  0      -> fop_ioctl

  0        -> sock_ioctl

  0          -> strioctl

  0             -> job_control_type

  0             <- job_control_type
```

```
  0            -> strcopyout

  0              -> copyout

  0              <- copyout

  0            <- strcopyout

  0          <- strioctl

  0        <- sock_ioctl

  0      <- fop_ioctl

  0      -> releasef

  0        -> clear_active_fd

  0        <- clear_active_fd

  0        -> cv_broadcast

  0        <- cv_broadcast

  0      <- releasef

  0    <- ioctl

  0  <= ioctl
```

此输出说明，在看上去与套接字关联的文件说明符中，`xclock` 进程调用了 `ioctl()`。

尝试了解内核驱动程序时，也可以使用 FBT。例如，`ssd`(7D) 驱动程序有很多代码路径，通过这些路径可以返回 `EIO`。使用 FBT 可以轻松地确定导致错误的准确代码路径，如下例所示：

```
fbt:ssd::return

/arg1 == EIO/

{

    printf("%s+%x returned EIO.", probefunc, arg0);

}
```

为了获取所返回的任何一个 EIO 的更多信息，您可能希望随机跟踪所有 fbt 探测器，然后根据特定函数的返回值执行 commit()（或 discard()）函数。有关随机跟踪的详细信息，请参见第 13 章。

或者，您可以使用 FBT 来了解指定模块中调用的函数。以下示例列出了 UFS 中调用的所有函数。

```
# dtrace -n fbt:ufs::entry'{@a[probefunc] = count()}'

dtrace: description 'fbt:ufs::entry' matched 353 probes

^C

  ufs_ioctl                                                 1

  ufs_statvfs                                               1

  ufs_readlink                                              1

  ufs_trans_touch                                           1

  wrip                                                      1

  ufs_dirlook                                               1

  bmap_write                                                1

  ufs_fsync                                                 1

  ufs_iget                                                  1

  ufs_trans_push_inode                                      1

  ufs_putpages                                              1

  ufs_putpage                                               1

  ufs_syncip                                                1

  ufs_write                                                 1

  ufs_trans_write_resv                                      1

  ufs_log_amt                                               1

  ufs_getpage_miss                                          1

  ufs_trans_syncip                                          1
```

```
getinoquota                                                    1

ufs_inode_cache_constructor                                    1

ufs_alloc_inode                                                1

ufs_iget_alloced                                               1

ufs_iget_internal                                              2

ufs_reset_vnode                                                2

ufs_notclean                                                   2

ufs_iupdat                                                     2

blkatoff                                                       3

ufs_close                                                      5

ufs_open                                                       5

ufs_access                                                     6

ufs_map                                                        8

ufs_seek                                                      11

ufs_addmap                                                    15

rdip                                                          15

ufs_read                                                      15

ufs_rwunlock                                                  16

ufs_rwlock                                                    16

ufs_delmap                                                    18

ufs_getattr                                                   19

ufs_getpage_ra                                                24

bmap_read                                                     25

findextent                                                    25
```

```
  ufs_lockfs_begin                                                 27

  ufs_lookup                                                       46

  ufs_iaccess                                                      51

  ufs_imark                                                        92

  ufs_lockfs_begin_getpage                                        102

  bmap_has_holes                                                  102

  ufs_getpage                                                     102

  ufs_itimes_nolock                                               107

  ufs_lockfs_end                                                  125

  dirmangled                                                      498

  dirbadname                                                      498
```

如果您知道内核函数的用途或参数，可以使用 FBT 来了解调用函数的方式或原因。 例如，putnext(9F) 接受指向 queue(9S) 结构的指针作为其第一个成员。 queue 结构的 q_qinfo 成员是指向 qinit(9S) 结构的指针。 qinit 结构的 qi_minfo 成员具有一个指向 module_info(9S) 结构（其 mi_idname 成员中包含模块名称）的指针。 以下示例在 putnext 中使用 FBT 探测器将这些信息收集到一起，以便根据模块名称跟踪 putnext(9F) 调用：

```
fbt::putnext:entry

{

    @calls[stringof(args[0]->q_qinfo->qi_minfo->mi_idname)] = count();

}
```

运行上面的脚本将会生成与以下示例类似的输出：

```
# dtrace -s ./putnext.d

^C


  iprb                                                             1
```

```
  rpcmod                                                    1
  pfmod                                                     1
  timod                                                     2
  vpnmod                                                    2
  pts                                                      40
  conskbd                                                  42
  kb8042                                                   42
  tl                                                       58
  arp                                                     108
  tcp                                                     126
  ptm                                                     249
  ip                                                      313
  ptem                                                    340
  vuid2ps2                                                361
  ttcompat                                                412
  ldterm                                                  413
  udp                                                     569
  strwhead                                                624
  mouse8042                                               726
```

也可以使用 FBT 确定特定函数花费的时间。 以下示例说明了如何确定 DDI 延迟例程 `drv_usecwait`(9F) 和 `delay`(9F) 的调用方。

```
fbt::delay:entry,

fbt::drv_usecwait:entry

{

    self->in = timestamp
```

```
}


fbt::delay:return,

fbt::drv_usecwait:return

/self->in/

{

    @snoozers[stack()] = quantize(timestamp - self->in);

    self->in = 0;

}
```

在引导期间运行此示例脚本特别有意义。第 36 章介绍了在系统引导期间执行匿名跟踪的过程。重新引导时，您可能会看到与以下示例类似的输出：

```
# dtrace -ae


              ata‘ata_wait+0x34

              ata‘ata_id_common+0xf5

              ata‘ata_disk_id+0x20

              ata‘ata_drive_type+0x9a

              ata‘ata_init_drive+0xa2

              ata‘ata_attach+0x50

              genunix‘devi_attach+0x75

              genunix‘attach_node+0xb2

              genunix‘i_ndi_config_node+0x97

              genunix‘i_ddi_attachchild+0x4b

              genunix‘devi_attach_node+0x3d
```

```
              genunix‘devi_config_one+0x1d0

              genunix‘ndi_devi_config_one+0xb0

              devfs‘dv_find+0x125

              devfs‘devfs_lookup+0x40

              genunix‘fop_lookup+0x21

              genunix‘lookuppnvp+0x236

              genunix‘lookuppnat+0xe7

              genunix‘lookupnameat+0x87

              genunix‘cstatat_getvp+0x134


           value  ------------- Distribution ------------- count
            2048 |                                         0
            4096 |@@@@@@@@@@@@@@@@@@@@@@                   4105
            8192 |@@@@                                     783
           16384 |@@@@@@@@@@@@@@@                          2793
           32768 |                                         16
           65536 |                                         0



              kb8042‘kb8042_wait_poweron+0x29

              kb8042‘kb8042_init+0x22

              kb8042‘kb8042_attach+0xd6

              genunix‘devi_attach+0x75

              genunix‘attach_node+0xb2
```

```
              genunix‘i_ndi_config_node+0x97

              genunix‘i_ddi_attachchild+0x4b

              genunix‘devi_attach_node+0x3d

              genunix‘devi_config_one+0x1d0

              genunix‘ndi_devi_config_one+0xb0

              genunix‘resolve_pathname+0xa5

              genunix‘ddi_pathname_to_dev_t+0x16

              consconfig_dacf‘consconfig_load_drivers+0x14

              consconfig_dacf‘dynamic_console_config+0x6c

              consconfig‘consconfig+0x8

              unix‘stubs_common_code+0x3b


   value  ------------- Distribution ------------- count

  262144 |                                         0

  524288 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@      221

 1048576 |@@@@                                     29

 2097152 |                                         0




              usba‘hubd_enable_all_port_power+0xed

              usba‘hubd_check_ports+0x8e

              usba‘usba_hubdi_attach+0x275

              usba‘usba_hubdi_bind_root_hub+0x168

              uhci‘uhci_attach+0x191
```

```
              genunix‘devi_attach+0x75

              genunix‘attach_node+0xb2

              genunix‘i_ndi_config_node+0x97

              genunix‘i_ddi_attachchild+0x4b

              genunix‘i_ddi_attach_node_hierarchy+0x49

              genunix‘attach_driver_nodes+0x49

              genunix‘ddi_hold_installed_driver+0xe3

              genunix‘attach_drivers+0x28


           value  ------------- Distribution ------------- count

        33554432 |                                         0

        67108864 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 3

       134217728 |                                         0
```

尾部调用优化

当一个函数以调用另一个函数而结束时，编译器可能会进行尾部调用优化，优化后，被调用的函数将重用调用方的栈帧。此过程通常用于 SPARC 体系结构中，在此情况下，编译器在被调用的函数中重用调用方的注册窗口，以便使注册窗口的压力降到最低。

进行此优化会使调用函数的 `return` 探测器在被调用函数的 `entry` 探测器之前触发。这种触发顺序会导致比较严重的混乱情况。例如，如果要记录从某个特定函数调用的所有函数，以及此函数调用的所有函数，则可以使用以下脚本：

```
fbt::foo:entry

{

    self->traceme = 1;

}
```

```
fbt:::entry

/self->traceme/

{

    printf("called %s", probefunc);

}



fbt::foo:return

/self->traceme/

{

    self->traceme = 0;

}
```

但是，如果 `foo()` 以优化的尾部调用结束，那么，在尾部调用的函数以及它调用的任何函数都不会被捕获。不能即时动态取消内核优化，DTrace 不希望总是考虑如何虚构代码结构。所以，应清楚什么时候可以使用尾部调用优化。

可以在类似以下示例的源代码中使用尾部调用优化：

```
    return (bar());
```

或者在类似以下示例的源代码中使用尾部调用优化：

```
    (void) bar();

    return;
```

相反地，结尾方式与以下示例类似的函数源代码不能优化对 `bar()` 的调用，因为对 `bar()` 的调用不是尾部调用：

```
    bar();

    return (rval);
```

可以使用以下方法确定是否已对某个调用进行了尾部调用优化：

- 在运行 DTrace 时，跟踪所考虑的 `return` 探测器的 `arg0`。`arg0` 包含函数中返回指令的偏移量。

- 在 DTrace 停止后，使用 mdb(1) 查看该函数。如果跟踪的偏移量包含对另一个函数的调用，而不是从函数返回的指令，则说明已对调用进行了尾部调用优化。

由于指令集体系结构，尾部调用优化在 SPARC 系统上要比在 x86 系统上更常见。 以下示例使用 mdb 发现内核的 dup() 函数中的尾部调用优化：

```
# dtrace -q -n fbt::dup:return'{printf("%s+0x%x", probefunc, arg0);}'
```

运行此命令时，将运行执行 dup(2) 的程序（如 bash 进程）。上面的命令会提供与以下示例类似的输出：

```
dup+0x10

^C
```

现在使用 mdb 检查函数：

```
# echo "dup::dis" | mdb -k

dup:                            sra       %o0, 0, %o0

dup+4:                          mov       %o7, %g1

dup+8:                          clr       %o2

dup+0xc:                        clr       %o1

dup+0x10:                       call      -0x1278       <fcntl>

dup+0x14:                       mov       %g1, %o7
```

该输出说明，dup+0x10 是对 fcntl() 函数的调用而不是 ret 指令。 所以，fcntl() 就是尾部调用优化的一个示例。

汇编函数

您可能会发现，有时似乎只进入某些函数但却不返回，或者出现相反的情况。这种函数很稀少，通常是手动编码的汇编例程，这些例程通常分支到其他手动编码的汇编函数中间。这些函数不应妨碍分析：分支到的目标函数仍然必须返回到其来源函数的调用方。 即，如果启用所有 FBT 探测器，则应看到进入某个函数，同时从相同栈深度的另一个函数返回。

指令集限制

一些函数无法通过 FBT 进行检测。 不可检测函数的准确特性特定于指令集体系结构。

x86 限制

在 x86 系统上不创建栈帧的函数无法通过 FBT 进行检测。因为 x86 的寄存器集非常小，大多数函数必须将数据放入栈中，从而创建栈帧。 但是，一些 x86 函数不创建栈帧，因此无法对这些函数进行检测。 x86 平台上无法进行检测的函数的实际数量不固定，但通常少于百分之五。

SPARC 限制

无法通过 FBT 对 SPARC 系统上以汇编语言进行手动编码的叶例程进行检测。 大多数内核用 C 语言编写，所有用 C 语言编写的函数都可以通过 FBT 进行检测。 SPARC 平台上无法进行检测的函数的实际数量不固定，但通常很少。

断点交互

FBT 通过动态修改内核文本进行工作。 由于内核断点也通过修改内核文本进行工作，所以如果在装入 DTrace 之前将内核断点放在入口或返回位置，FBT 将拒绝提供用于函数的探测器，即使随后删除了内核断点也是如此。 如果在装入 DTrace 之后放置内核断点，则内核断点和 DTrace 探测器将对应于文本中相同的位置。 在此情况下，断点将首先触发，然后在调试器恢复内核时，探测器将触发。建议不要同时使用内核断点和 DTrace。如果必须使用断点，请改为使用 DTrace `breakpoint()` 操作。

模块装入

Solaris 内核可以动态装入和卸载内核模块。当装入 FBT 并动态装入模块时，FBT 将自动提供与新模块关联的新探测器。 如果装入的模块未启用 FBT 探测器，则可以卸载模块；卸载模块时，相应的探测器将被破坏。 如果装入的模块已启用 FBT 探测器，则该模块将被视为正忙，无法卸载该模块。

稳定性

FBT 提供器使用 DTrace 的稳定性机制说明其稳定性，如下表所示。有关稳定性机制的更多信息，请参见第 39 章。

元素	名称稳定性	数据稳定性	相关性类
提供器	发展中	发展中	ISA
模块	专用	专用	未知
函数	专用	专用	未知
名称	发展中	发展中	ISA
参数	专用	专用	ISA

FBT 显示内核实现时，所有关于实现的内容都不“稳定”－模块和函数名称以及数据稳定性都明确地显示为“专用”。提供器和名称的数据稳定性为“发展中”，但是所有其他数据稳定性都为“专用”：它们是当前实现的产物。FBT 的相关类为 ISA：虽然在当前的所有指令集体系结构中都可以使用 FBT，但不能保证在将来的任意指令集体系结构中都可以使用 FBT。

◆ ◆ ◆ 第 21 章

21 syscall 提供器

syscall 提供器可在系统中每个系统调用的入口和返回位置提供探测器。因为系统调用是用户级应用程序和操作系统内核之间的主要接口，所以 syscall 提供器可以对系统相关的应用程序行为提供全面监测。

探测器

syscall 为每个系统调用都提供一对探测器：在进入系统调用之前触发的entry 探测器，和在系统调用完成后，但在将控制传送回用户级之前触发的 return 探测器。对于所有 syscall 探测器，函数名称将设置为受检测的系统调用的名称，但对模块名称未定义。

可以在 /etc/name_to_sysnum 文件中找到 syscall 提供器提供的系统调用的名称。通常，syscall 提供的系统调用名称对应于手册页的第 2 节中的名称。但是，syscall 提供器提供的一些探测器不直接与任何记录的系统调用对应。本节中对此差异的一般原因作了说明。

系统调用记时错误

在一些情况下，syscall 提供器提供的系统调用的名称实际上是旧的实现详细信息的反映。例如，因为一些追溯到 UNIX™ 的过去的原因，/etc/name_to_sysnum 中 exit(2) 的名称为 rexit。类似地，time(2) 的名称为 gtime，execle(2) 和 execve(2) 的名称为 exece。

子编码的系统调用

第 2 节中说明的一些系统调用实现为未记录的系统调用的子操作。例如，与 System V 信号（semctl(2)、semget(2)、semids(2)、semop(2) 和 semtimedop(2)）相关的系统调用实现为单个系统调用 semsys 的子操作。semsys 系统调用将特定于实现的子代码作为其第一个参数，该子代码表示所需的特定系统调用：SEMCTL、SEMGET、SEMIDS、SEMOP 或 SEMTIMEDOP。由于过载一个系统调用以实现多个系统调用，所以对于 System V 信号仅有一对 syscall 探测器：syscall::semsys:entry 和 syscall::semsys:return。

大文件系统调用

支持大小超过 4GB 的大文件的 32 位程序必须能够处理 64 位文件偏移。 因为大文件需要使用大偏移，所以通过一组并行的系统接口处理大文件，如 lf64(5) 中所述。lf64 中记录了这些接口，但这些接口没有单独的手册页。 如表 21–1 中所示，每一个大文件系统调用接口都显示为各自的 syscall 探测器。

表 21–1 syscall 大文件探测器

大文件 syscall 探测器	系统调用
creat64	creat(2)
fstat64	fstat(2)
fstatvfs64	fstatvfs(2)
getdents64	getdents(2)
getrlimit64	getrlimit(2)
lstat64	lstat(2)
mmap64	mmap(2)
open64	open(2)
pread64	pread(2)
pwrite64	pwrite(2)
setrlimit64	setrlimit(2)
stat64	stat(2)
statvfs64	statvfs(2)

专用系统调用

一些系统调用为跨越用户内核边界的 Solaris 子系统的专用实现详细信息。 同样地，这些系统调用在第 2 节中没有手册页。此类别中的系统调用示例包括 signotify 系统调用（用作实现 POSIX.4 消息队列的一部分）和 utssys 系统调用（用于实现 fuser(1M)）。

参数

对于 `entry` 探测器，参数（`arg0` .. `arg`*n*）是系统调用的参数。对于 `return` 探测器，`arg0` 和 `arg1` 包含返回值。D 变量 `errno` 中的非零值指示系统调用失败。

稳定性

`syscall` 提供器使用 DTrace 的稳定性机制说明其稳定性，如下表所示。有关稳定性机制的更多信息，请参阅第 39 章。

元素	名称稳定性	数据稳定性	相关性类
提供器	发展中	发展中	公用
模块	专用	专用	未知
函数	不稳定	不稳定	ISA
名称	发展中	发展中	公用
参数	不稳定	不稳定	ISA

◆ ◆ ◆ 第 22 章

22 sdt 提供器

静态定义的跟踪 (Statically Defined Tracing,SDT) 提供器在软件程序员正式指定的位置创建探测器。使用 SDT 机制，程序员可以有意识地选择 DTrace 用户关注的位置，并通过探测器名称告知有关每个位置的一些语义知识。Solaris 内核已定义了少量的 SDT 探测器，在以后可能会添加更多的探测器。DTrace 还为用户应用程序开发者提供了定义静态探测器的机制，如第 34 章中所述。

探测器

表 22–1 中列出了 Solaris 内核定义的 SDT 探测器。这些探测器的名称稳定性和数据稳定性均为“专用”，因为此处对其进行的说明仅反映内核的实现，不应推断为接口约定。有关 DTrace 稳定性机制的更多信息，请参见第 254 页中的 “稳定性”。

表 22–1 SDT 探测器

探测器名称	说明	arg0
callout-start	在执行 callout（请参见 <sys/callo.h>）之前的瞬间触发的探测器。Callout 由系统时钟定期执行，代表 timeout(9F) 的实现。	指向callout_t（请参见 <sys/callo.h>）的指针，它对应于要执行的 callout。
callout-end	执行 callout（请参见 <sys/callo.h>）之后将立即触发的探测器。	指向 callout_t（请参见 <sys/callo.h>）的指针，它对应于已执行的 callout。
interrupt-start	在调入设备的中断处理程序之前的瞬间触发的探测器。	指向 dev_info 结构（请参见 <sys/ddi_impldefs.h>）的指针，它对应于中断设备。
interrupt-complete	从设备的中断处理程序返回后将立即触发的探测器。	指向 dev_info 结构（请参见 <sys/ddi_impldefs.h>）的指针，它对应于中断设备。

示例

以下示例是用于观察每秒钟内 callout 行为的脚本：

```
#pragma D option quiet


sdt:::callout-start
{
    @callouts[((callout_t *)arg0)->c_func] = count();
}


tick-1sec
{
    printa("%40a %10@d\n", @callouts);
    clear(@callouts);
}
```

运行此示例可以发现系统中经常使用 `timeout`(9F) 的用户，如以下输出所示：

```
# dtrace -s ./callout.d
                                      FUNC       COUNT
                               TS‘ts_update           1
              uhci‘uhci_cmd_timeout_hdlr           3
                            genunix‘setrun           5
                     genunix‘schedpaging           5
                          ata‘ghd_timeout          10
 uhci‘uhci_handle_root_hub_status_change         309
```

```
                                    FUNC      COUNT

               ip‘tcp_time_wait_collector          1

                           TS‘ts_update          1

               uhci‘uhci_cmd_timeout_hdlr          3

                      genunix‘schedpaging          4

                           genunix‘setrun          8

                          ata‘ghd_timeout         10

 uhci‘uhci_handle_root_hub_status_change        300


                                    FUNC      COUNT

               ip‘tcp_time_wait_collector          0

                        iprb‘mii_portmon          1

                           TS‘ts_update          1

               uhci‘uhci_cmd_timeout_hdlr          3

                      genunix‘schedpaging          4

                           genunix‘setrun          7

                          ata‘ghd_timeout         10

 uhci‘uhci_handle_root_hub_status_change        300
```

timeout(9F) 接口仅生成单个计时器过期。需要时间间隔计时器功能的 timeout() 的使用者通常通过 timeout() 处理程序安装 timeout。以下示例显示了此行为：

```
#pragma D option quiet


sdt:::callout-start

{
```

```
    self->callout = ((callout_t *)arg0)->c_func;
}

fbt::timeout:entry
/self->callout && arg2 <= 100/
{
    /*
     * In this case, we are most interested in interval timeout(9F)s that
     * are short.  We therefore do a linear quantization from 0 ticks to
     * 100 ticks.  The system clock's frequency — set by the variable
     * "hz" — defaults to 100, so 100 system clock ticks is one second.
     */
    @callout[self->callout] = lquantize(arg2, 0, 100);
}

sdt:::callout-end
{
    self->callout = NULL;
}

END
{
    printa("%a\n%@d\n\n", @callout);
}
```

运行此脚本，等待几秒钟，然后按 Ctrl-C 组合键，将产生与以下示例类似的输出：

```
# dtrace -s ./interval.d
^C
genunix‘schedpaging


           value  ------------- Distribution ------------- count
              24 |                                         0
              25 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 20
              26 |                                         0



ata‘ghd_timeout


           value  ------------- Distribution ------------- count
               9 |                                         0
              10 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 51
              11 |                                         0



uhci‘uhci_handle_root_hub_status_change


           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1515
```

```
               2 |                                         0
```

该输出说明，uhci(7D) 驱动程序中的uhci_handle_root_hub_status_change() 代表系统中最短的时间间隔计时器：在每个系统时钟周期都将调用该计时器。

interrupt-start 探测器可用于了解中断活动。以下示例说明了如何按驱动程序名称量化执行中断处理程序花费的时间：

```
interrupt-start

{

    self->ts = vtimestamp;

}



interrupt-complete

/self->ts/

{

    this->devi = (struct dev_info *)arg0;

    @[stringof(‘devnamesp[this->devi->devi_major].dn_name),

        this->devi->devi_instance] = quantize(vtimestamp - self->ts);

}
```

运行此脚本将会生成与以下示例类似的输出：

```
# dtrace -s ./intr.d

dtrace: script ’./intr.d’ matched 2 probes

^C

 isp                                                     0

           value  ------------- Distribution ------------- count

            8192 |                                         0

           16384 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
```

```
          32768 |                                         0


pcf8584                                                    0
          value  ------------- Distribution ------------- count
             64 |                                         0
            128 |                                         2
            256 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  157
            512 |@@@@@@                                   31
           1024 |                                         3
           2048 |                                         0


pcf8584                                                    1
          value  ------------- Distribution ------------- count
           2048 |                                         0
           4096 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  154
           8192 |@@@@@@@@                                 37
          16384 |                                         2
          32768 |                                         0


qlc                                                        0
          value  ------------- Distribution ------------- count
          16384 |                                         0
          32768 |@@                                       9
          65536 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  126
```

```
          131072 |@                                        5
          262144 |                                         2
          524288 |                                         0


  hme                                                      0
           value  ------------- Distribution ------------- count
            1024 |                                         0
            2048 |                                         6
            4096 |                                         2
            8192 |@@@@                                     89
           16384 |@@@@@@@@@@@@@@                           262
           32768 |@                                        37
           65536 |@@@@@@@                                  139
          131072 |@@@@@@@@                                 161
          262144 |@@@                                      73
          524288 |                                         4
         1048576 |                                         0
         2097152 |                                         1
         4194304 |                                         0


  ohci                                                     0
           value  ------------- Distribution ------------- count
            8192 |                                         0
           16384 |                                         3
```

```
     32768 |                                         1
     65536 |@@@                                      143
    131072 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@    1368
    262144 |                                         0
```

创建 SDT 探测器

如果您是设备驱动程序开发者，可能对在 Solaris 驱动程序中创建自己的 SDT 探测器感兴趣。禁用 SDT 的探测效果实际上是以不执行多个计算机指令为代价。所以，建议根据需要将 SDT 探测器添加到设备驱动程序中。如果这些探测器不对性能造成负面影响，则可以将其保留在提供的代码中。

声明探测器

使用 `<sys/sdt.h>` 中的 `DTRACE_PROBE`、`DTRACE_PROBE1`、`DTRACE_PROBE2`、`DTRACE_PROBE3` 和 `DTRACE_PROBE4` 宏声明 SDT 探测器。基于 SDT 探测器的模块名称和函数名称对应于探测器的内核模块和函数。探测器的名称取决于 `DTRACE_PROBE`*n* 宏中指定的名称。如果名称中不包含两个连续下条 (__)，则探测器的名称与宏中指定的名称相同。如果名称中包含任何两个连续下条，则探测器名称会将该连续下条转换为一个破折号 (-)。例如，如果 `DTRACE_PROBE` 宏指定 `transaction__start`，则 SDT 探测器将被命名为 `transaction-start`。此替代允许 C 代码提供不是有效 C 标识符的宏名称，无需指定字符串。

DTrace 将内核模块名称和函数名称作为用于标识探测器的元组的一部分，因此无需在探测器名称中包括这些信息，以避免名称空间冲突。可以对驱动程序 *module* 使用命令 `dtrace -l -P sdt -m` *module* 来列出已安装的探测器和 DTrace 用户将可以看到的全名。

探测器参数

每个 SDT 探测器的参数都是在相应的 `DTRACE_PROBE`*n* 宏引用中指定的参数。参数的数量取决于用于创建该探测器的宏：`DTRACE_PROBE1` 指定一个参数，`DTRACE_PROBE2` 指定两个参数，依此类推。声明 SDT 探测器时，可通过不取消引用指针，并且不从探测器参数的全局变量中装入，来将已禁用的探测影响降至最低。在启用探测器的 D 操作中可以安全地取消引用指针和装入全局变量，因此 DTrace 用户可以只在需要这些操作时才请求这些操作。

稳定性

SDT 提供器使用 DTrace 的稳定性机制说明其稳定性，如下表所示。有关稳定性机制的更多信息，请参见第 39 章。

元素	名称稳定性	数据稳定性	相关性类
提供器	发展中	发展中	ISA
模块	专用	专用	未知
函数	专用	专用	未知
名称	专用	专用	ISA
参数	专用	专用	ISA

◆ ◆ ◆ 第 23 章

23 sysinfo 提供器

sysinfo 提供器提供了与按名称 sys 分类的内核统计信息对应的探测器。因为这些统计信息为系统监视实用程序（如 mpstat(1M)）提供输入，所以 sysinfo 提供器可以快速调查观察到的异常行为。

探测器

sysinfo 提供器提供了对应于 sys 命名的内核统计信息中字段的探测器。sysinfo 提供的探测器就在递增相应的 sys 值之前触发。以下示例说明了如何使用 kstat(1M) 命令显示 sys 命名的内核统计信息的名称和当前值。

```
$ kstat -n sys

module: cpu                             instance: 0

name:   sys                             class:    misc

    bawrite                         123

    bread                           2899

    bwrite                          17995

    cpu_ticks_idle                  73743866

    cpu_ticks_kernel                2096277

    cpu_ticks_user                  1010122

    cpu_ticks_wait                  46413

...
```

表 23–1 中对 sysinfo 探测器进行了说明。

表 23–1 sysinfo 探测器

bawrite	要将缓冲区异步写出到设备中时将触发的探测器。
bread	从设备中实际读取缓冲区时将触发的探测器。bread 在从设备中请求缓冲区之后，但在阻塞暂挂的请求完成之前触发。
bwrite	要将缓冲区写出（无论同步或异步）到设备中时将触发的探测器。
cpu_ticks_idle	定期系统时钟已确定 CPU 处于空闲时将触发的探测器。请注意，此探测器在系统时钟的上下文中触发，所以将在运行系统时钟的 CPU 中触发。cpu_t 参数 (arg2) 指示 CPU 已被认为处于空闲状态。有关详细信息，请参见第 258 页中的 “参数”。
cpu_ticks_kernel	定期系统时钟已确定 CPU 正在内核中执行时将触发的探测器。请注意，此探测器在系统时钟的上下文中触发，所以将在运行系统时钟的 CPU 中触发。cpu_t 参数 (arg2) 指示 CPU 已被认为正在内核中执行。有关详细信息，请参见第 258 页中的 “参数”。
cpu_ticks_user	定期系统时钟已确定 CPU 正在用户模式下执行时将触发的探测器。请注意，此探测器在系统时钟的上下文中触发，所以将在运行系统时钟的 CPU 中触发。cpu_t 参数 (arg2) 指示 CPU 已被认为正在用户模式下运行。有关详细信息，请参见第 258 页中的 “参数”。
cpu_ticks_wait	定期系统时钟已确定 CPU 处于空闲状态，但一些线程正在等待 CPU 中的 I/O 时将触发的探测器。请注意，此探测器在系统时钟的上下文中触发，所以将在运行系统时钟的 CPU 中触发。cpu_t 参数 (arg2) 指示 CPU 已被认为正在等待 I/O。有关详细信息，请参见第 258 页中的 “参数”。
idlethread	CPU 进入空闲循环时将触发的探测器。
intrblk	中断线程阻塞时将触发的探测器。
inv_swtch	强制正在运行的线程被迫放弃 CPU 时将触发的探测器。
lread	逻辑上从设备中读取缓冲区时将触发的探测器。
lwrite	逻辑上将缓冲区写入设备时将触发的探测器。
modload	装入内核模块时将触发的探测器。
modunload	卸载内核模块时将触发的探测器。
msg	进行 msgsnd(2) 或 msgrcv(2) 系统调用（但在执行消息队列操作之前）时将触发的探测器。
mutex_adenters	尝试获取已拥有的自适应锁定时将触发的探测器。如果触发此探测器，则会同时触发 lockstat 提供器的 adaptive-block 或 adaptive-spin 探测器。有关详细信息，请参见第 18 章。
namei	尝试在文件系统中进行名称查找时将触发的探测器。

表 23–1 sysinfo 探测器 （续）

nthreads	创建线程时将触发的探测器。
phread	要执行原始 I/O 读取时将触发的探测器。
phwrite	要执行原始 I/O 写入时将触发的探测器。
procovf	由于系统缺乏进程表项而无法创建新进程时将触发的探测器。
pswitch	CPU 从执行某个线程切换到执行另一个线程时将触发的探测器。
readch	每次成功读取后，但在将控制返回到执行该读取的线程之前将触发的探测器。可以通过 read(2)、readv(2) 或 pread(2) 系统调用进行读取。arg0 包含已成功读取的字节数。
rw_rdfails	如果在写入者暂挂锁定或写入者需要锁定时，尝试读取并锁定读取者/写入者，则将触发的探测器。如果触发此探测器，则 lockstat 提供器的 rw-block 探测器也将触发。有关详细信息，请参见第 18 章。
rw_wrfails	如果在一定数量的读取者或另一个写入者暂挂锁定时，尝试写入并锁定读取者/写入者锁定，则将触发的探测器。如果触发此探测器，则 lockstat 提供器的 rw-block 探测器也将触发。有关详细信息，请参见第 18 章。
sema	进行 semop(2) 系统调用（但在执行任何信号操作之前）时将触发的探测器。
sysexec	执行 exec(2) 系统调用时将触发的探测器。
sysfork	执行 fork(2) 系统调用时将触发的探测器。
sysread	执行 read(2)、readv(2) 或 pread(2) 系统调用时将触发的探测器。
sysvfork	执行 vfork(2) 系统调用时将触发的探测器。
syswrite	执行 write(2)、writev(2) 或 pwrite(2) 系统调用时将触发的探测器。
trap	发生处理器陷阱时将触发的探测器。请注意，一些处理器（特别是 UltraSPARC 变体）通过不会引起此探测器触发的机制处理一些轻权陷阱。
ufsdirblk	从 UFS 文件系统中读取目录块时将触发的探测器。有关 UFS 的详细信息，请参见 ufs(7FS)。
ufsiget	检索 inode 时将触发的探测器。有关 UFS 的详细信息，请参见 ufs(7FS)。
ufsinopage	可重用不包含任何关联数据页的内核 inode 之后将触发的探测器。有关 UFS 的详细信息，请参见 ufs(7FS)。
ufsipage	可重用包含任何关联数据页的内核 inode 之后将触发的探测器。在将关联的数据页刷新到磁盘之后将触发此探测器。有关 UFS 的详细信息，请参见 ufs(7FS)。

表 23–1 sysinfo 探测器 （续）

wait_ticks_io	定期系统时钟已确定 CPU 处于空闲状态，但一些线程正在等待 CPU 中的 I/O 时将触发的探测器。请注意，此探测器在系统时钟的上下文中触发，所以将在运行系统时钟的 CPU 中触发。cpu_t 参数 (arg2) 指示 CPU 已被描述为正在等待 I/O。有关详细信息，请参见第 258 页中的 “参数” 中的 arg2。wait_ticks_io 和 cpu_ticks_wait; wait_ticks_io 之间不存在语义差别是由于历史原因造成的。
writech	每次成功写入后，但在将控制返回到执行该写入的线程之前将触发的探测器。可以通过 write(2)、writev(2) 或 pwrite(2) 系统调用进行写入。arg0 包含已成功写入的字节数。
xcalls	要进行交叉调用时将触发的探测器。交叉调用是一个 CPU 请求另一个 CPU 的即时工作的操作系统机制。

参数

sysinfo 探测器的参数如下所示：

arg0	要对统计信息递增的值。对于大多数探测器，此参数始终为 1，但对于一些探测器，此参数可以采用其他值。
arg1	指向要对统计信息递增的当前值的指针。此值为 64 位量，将按 arg0 中的值递增。取消引用此指针可以使使用者确定对应于该探测器的统计信息的当前计数。
arg2	指向 cpu_t 结构的指针，该结构对应于要递增统计信息的 CPU。此结构在 <sys/cpuvar.h> 中定义，但它为内核实现的一部分，应视为“专用”。

对于大多数 sysinfo 探测器，arg0 的值为 1。但是，readch 和 writech 探测器将 arg0 分别设置为已读取或已写入的字节数。使用以下功能可以确定按可执行名称读取的大小，如下例所示：

```
# dtrace -n readch'{@[execname] = quantize(arg0)}'

dtrace: description 'readch' matched 4 probes

^C

  xclock

           value  ------------- Distribution ------------- count

              16 |                                         0

              32 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
```

```
            64 |                                         0


acroread

         value  ------------- Distribution ------------- count
            16 |                                         0
            32 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 3
            64 |                                         0


FvwmAuto

         value  ------------- Distribution ------------- count
             2 |                                         0
             4 |@@@@@@@@@@@@@                            13
             8 |@@@@@@@@@@@@@@@@@@@@                     21
            16 |@@@@@                                    5
            32 |                                         0


xterm

         value  ------------- Distribution ------------- count
            16 |                                         0
            32 |@@@@@@@@@@@@@@@@@@@@@@@@                 19
            64 |@@@@@@@@@                                7
           128 |@@@@@@                                   5
           256 |                                         0
```

```
fvwm2

           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@@@@@@                               186
               1 |                                         0
               2 |                                         0
               4 |@@                                       51
               8 |                                         17
              16 |                                         0
              32 |@@@@@@@@@@@@@@@@@@@@@@@@@@@              503
              64 |                                         9
             128 |                                         0


Xsun

           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@@@@@@@                              269
               1 |                                         0
               2 |                                         0
               4 |                                         2
               8 |@                                        31
              16 |@@@@@                                    128
              32 |@@@@@@@                                  171
              64 |@                                        33
```

```
           128 |@@@                                      85
           256 |@                                        24
           512 |                                         8
          1024 |                                         21
          2048 |@                                        26
          4096 |                                         21
          8192 |@@@@                                     94
         16384 |                                         0
```

sysinfo 提供器将 arg2 设置为指向 cpu_t（内核实现的一种内部结构）的指针。大多数 sysinfo 探测器在要递增统计信息的 CPU 中触发，但一些探测器不是这样。例外的探测器包括 cpu_ticks_idle、cpu_ticks_kernel、cpu_ticks_user 和 cpu_ticks_wait（始终在执行系统时钟的 CPU 中触发）。使用 cpu_t 的 cpu_id 成员可确定关注的 CPU。以下 D 脚本运行大约 10 秒后，将按统计信息快速获取相关 CPU 行为的快照：

```
cpu_ticks_*

{

    @[probename] = lquantize(((cpu_t *)arg2)->cpu_id, 0, 1024, 1);

}



tick-1sec

/x++ >= 10/

{

    exit(0);

}
```

运行上面的脚本将会生成与以下示例类似的输出：

```
# dtrace -s ./tick.d

dtrace: script ’./tick.d’ matched 5 probes
```

```
CPU     ID                    FUNCTION:NAME

 22  37588                      :tick-1sec


  cpu_ticks_user

           value  ------------- Distribution ------------- count

              11 |                                         0

              12 |@@@@@@@@@                                14

              13 |@@@@                                     7

              14 |@                                        3

              15 |@                                        2

              16 |@@                                       4

              17 |@@@@@@                                   10

              18 |                                         0

              19 |@                                        2

              20 |@@@                                      6

              21 |@@@                                      5

              22 |                                         1

              23 |@@@@@@                                   10

              24 |                                         0


  cpu_ticks_wait

           value  ------------- Distribution ------------- count

              11 |                                         0

              12 |@@@@@@@@@@@@@@                           241
```

```
           13 |@@@@@@@@@@@@@@                          236
           14 |                                        16
           15 |@@@@@@@                                 132
           16 |                                        11
           17 |                                        10
           18 |                                        7
           19 |@                                       18
           20 |                                        4
           21 |                                        16
           22 |                                        13
           23 |                                        10
           24 |                                        0


  cpu_ticks_kernel

           value  ------------- Distribution ------------- count
           11 |                                        0
           12 |@@@@@@@@                                234
           13 |@@@@@                                   159
           14 |@@@                                     104
           15 |@@@@                                    131
           16 |@@                                      66
           17 |@                                       40
           18 |@                                       51
           19 |@                                       36
```

```
           20 |@@                                       56
           21 |@                                        42
           22 |@@@                                      96
           23 |@@                                       57
           24 |                                         0


  cpu_ticks_idle

           value  ------------- Distribution ------------- count
           11 |                                         0
           12 |@@                                       534
           13 |@@                                       621
           14 |@@@                                      900
           15 |@@                                       758
           16 |@@@                                      942
           17 |@@@                                      963
           18 |@@@                                      965
           19 |@@@                                      967
           20 |@@@                                      957
           21 |@@@                                      960
           22 |@@@                                      913
           23 |@@@                                      946
           24 |                                         0
```

示例

通过 mpstat(1M) 检查以下输出：

```
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
 12   90  22 5760   422  299  435   26   71  116   11  1372    5  19  17  60
 13   46  18 4585   193  162  431   25   69  117   12  1039    3  17  14  66
 14   33  13 3186   405  381  397   21   58  105   10   770    2  17  11  70
 15   34  19 4769   109   78  417   23   57  115   13   962    3  14  14  69
 16   74  16 4421   437  406  448   29   77  111    8  1020    4  23  14  59
 17   51  15 4493   139  110  378   23   62  109    9   928    4  18  14  65
 18   41  14 4204   494  468  360   23   56  102    9   849    4  17  12  68
 19   37  14 4229   115   87  363   22   50  106   10   845    3  15  14  67
 20   78  17 5170   200  169  456   26   69  108    9  1119    5  21  25  49
 21   53  16 4817    78   51  394   22   56  106    9   978    4  17  22  57
 22   32  13 3474   486  463  347   22   48  106    9   769    3  17  17  63
 23   43  15 4572    59   34  361   21   46  102   10   947    4  15  22  59
```

从上面的输出可以总结出 xcal 字段似乎太高，特别是所指定的系统相对空闲程度。mpstat 通过检查 sys 内核统计信息的 xcalls 字段确定 xcal 字段的值。因此，通过启用 xcalls sysinfo 探测器可以轻松地研究此异常，如下例所示：

```
# dtrace -n xcalls'{@[execname] = count()}'

dtrace: description 'xcalls' matched 4 probes

^C

  dtterm                                                            1

  nsrd                                                              1

  in.mpathd                                                         2

  top                                                               3
```

```
  lockd                                                             4

  java_vm                                                          10

  ksh                                                              19

  iCald.pl6+RPATH                                                  28

  nwadmin                                                          30

  fsflush                                                          34

  nsrindexd                                                        45

  in.rlogind                                                       56

  in.routed                                                       100

  dtrace                                                          153

  rpc.rstatd                                                      246

  imapd                                                           377

  sched                                                           431

  nfsd                                                           1227

  find                                                           3767
```

该输出说明了查找交叉调用的源的位置。一些数量的 `find(1)` 处理导致大多数交叉调用。以下 D 脚本可用于进一步详细了解该问题：

```
syscall:::entry

/execname == "find"/

{

    self->syscall = probefunc;

    self->insys = 1;

}
```

```
sysinfo:::xcalls

/execname == "find"/

{

    @[self->insys ? self->syscall : "<none>"] = count();

}



syscall:::return

/self->insys/

{

    self->insys = 0;

    self->syscall = NULL;

}
```

此脚本使用 `syscall` 提供器将来自 `find` 的交叉调用归结到特定系统调用。一些交叉调用（如由于页面错误产生的系统调用）可能不是通过系统调用发出。此脚本在这些情况下将列显 "<none>"。运行此脚本将会生成与以下示例类似的输出：

```
# dtrace -s ./find.d

 dtrace: script ’./find.d’ matched 444 probes

^C

  <none>                                                        2

  lstat64                                                    2433

  getdents64                                                14873
```

此输出指示由 `find` 触发的大多数交叉调用又被 `getdents`(2) 系统调用触发。进一步的了解取决于您要了解的具体方面。如果要了解为什么 `find` 进程进行 `getdents` 调用，可以编写 D 脚本在 `find` 触发交叉调用时聚集 `ustack()`。如果要了解为什么 `getdents` 调用触发交叉调用，可以编写 D 脚本在 `find` 触发交叉调用时聚集 `stack()`。无论下一步执行什么，`xcalls` 探测器的存在都将使您快速发现异常监视输出的根本原因。

稳定性

sysinfo 提供器使用 DTrace 的稳定性机制说明其稳定性，如下表所示。有关稳定性机制的更多信息，请参见第 39 章。

元素	名称稳定性	数据稳定性	相关性类
提供器	发展中	发展中	ISA
模块	专用	专用	未知
函数	专用	专用	未知
名称	发展中	发展中	ISA
参数	专用	专用	ISA

◆ ◆ ◆ 第 24 章

24 vminfo 提供器

vminfo 提供器提供了与 vm 内核统计信息对应的探测器。由于这些统计信息为系统监视实用程序（如 vmstat(1M)）提供输入，因此 vminfo 提供器可以快速查看观察到的异常行为。

探测器

vminfo 提供器提供了与 vm 命名的内核统计信息中的字段对应的探测器：vminfo 提供的探测器就在递增相应的 vm 值之前的瞬间触发。要显示 vm 命名的内核统计信息的名称和当前值，请使用 kstat(1M) 命令，如以下示例所示：

```
$ kstat -n vm

module: cpu                             instance: 0

name:   vm                              class:    misc

        anonfree                        13

        anonpgin                        2620

        anonpgout                       13

        as_fault                        12528831

        cow_fault                       2278711

        crtime                          202.10625712

        dfree                           1328740

        execfree                        0

        execpgin                        5541
```

...

表 24–1 中对 vminfo 探测器进行了说明。

表 24–1 vminfo 探测器

anonfree	作为分页活动的一部分，释放未修改的匿名页时将触发的探测器。匿名页是指那些与文件不关联的页面。包含此类页面的内存包括堆内存、栈内存或通过显式映射 zero(7D) 获取的内存。
anonpgin	从交换设备调进匿名页时将触发的探测器。
anonpgout	将已修改的匿名页调出到交换设备时将触发的探测器。
as_fault	在页面上捕获到故障，并且该故障既不是保护故障，也不是写复制故障时将触发的探测器。
cow_fault	在页面上捕获到写复制故障时将触发的探测器。arg0 包含由于写复制而创建的页数。
dfree	由于分页活动而释放页面时将触发的探测器。触发 dfree 时，anonfree、execfree 或 fsfree 之一也必然在随后触发。
execfree	由于分页活动而释放未修改的可执行页时将触发的探测器。
execpgin	从后备存储调进可执行页时将触发的探测器。
execpgout	将已修改的可执行页调出到后备存储时将触发的探测器。可执行页的大多数分页都与 execfree 有关。只有在内存中修改了可执行页时，才能触发 execpgout，这种情况在多数系统中不常见。
fsfree	作为分页活动的一部分，释放未修改的文件系统数据页时将触发的探测器。
fspgin	从后备存储调进文件系统页时将触发的探测器。
fspgout	将已修改的文件系统页调出到后备存储时将触发的探测器。
kernel_asflt	内核在页面自身的地址空间中捕获到页面故障时将触发的探测器。在触发 kernel_asflt 之前的瞬间会触发 as_fault 探测器。
maj_fault	捕获到导致从后备存储或交换设备执行 I/O 操作的页面故障时将触发的探测器。在触发 maj_fault 之前的瞬间会触发 pgin 探测器。
pgfrec	从可用页列表回收页面时将触发的探测器。
pgin	从后备存储或交换设备调进页面时将触发的探测器。此探测器与 maj_fault 不同，因为 maj_fault 仅在页面由于故障而调进时触发，而 pgin 则无论原因是什么，每次调进页面时都会触发。
pgout	将页面调出到后备存储或交换设备时将触发的探测器。

表 24–1 vminfo 探测器 （续）

pgpgin	从后备存储或交换设备调进页面时将触发的探测器。pgpgin 和 pgin 之间的唯一区别在于，pgpgin 包含按照 arg0 调进的页数，pgin 则在 arg0 中始终包含 1。
pgpgout	将页面调出到后备存储或交换设备时将触发的探测器。pgpgout 和 pgout 之间的唯一区别在于，pgpgout 包含按照 arg0 调出的页数。（pgout 则在 arg0 中始终包含 1。）
pgrec	回收页面时将触发的探测器。
pgrrun	调度页面调度程序时将触发的探测器。
pgswapin	从换出进程换入页面时将触发的探测器。换入的页数包含在 arg0 中。
pgswapout	作为换出进程的一部分，换出页面时将触发的探测器。换出的页数包含在 arg0 中。
prot_fault	由于保护违规而捕获到页面故障时将触发的探测器。
rev	页面守护进程在所有页面中启动新循环时将触发的探测器。
scan	页面守护进程检查页面时将触发的探测器。
softlock	作为在页面上放置软件锁的一部分，页面出现故障时将触发的探测器。
swapin	将换出进程换回时将触发的探测器。
swapout	换出进程时将触发的探测器。
zfod	根据需要创建用零填充的页面时将触发的探测器。

参数

arg0	统计信息将递增的值。对于大多数探测器，此参数始终为 1，但对于一些探测器，此参数可以采用其他值；在表 24–1 中，对这些探测器进行了说明。
arg1	指向要递增的统计信息当前值的指针。此值是一个 64 位数，将按 arg0 中的值递增。取消引用此指针后，使用者可确定与探测器对应的统计信息的当前计数。

示例

检查 vmstat(1M) 的以下输出：

```
kthr      memory            page            disk          faults      cpu
 r b w   swap  free  re  mf pi po fr de sr cd s0 — —   in   sy   cs us sy id
```

```
 0 1 0 1341844 836720 26 311 1644 0 0 0  0 216 0  0  0  797  817  697  9 10 81

 0 1 0 1341344 835300 238 934 1576 0 0 0 0 194 0  0  0  750 2795  791  7 14 79

 0 1 0 1340764 833668 24 165 1149 0 0 0  0 133 0  0  0  637  813  547  5  4 91

 0 1 0 1340420 833024 24 394 1002 0 0 0  0 130 0  0  0  621 2284  653 14  7 79

 0 1 0 1340068 831520 14 202 380 0 0  0  0 59  0  0  0  482 5688 1434 25  7 68
```

在以上输出中，pi 列表示调进的页数。通过 vminfo 提供器，可以详细了解这些页调进的来源，如以下示例所示：

```
dtrace -n pgin'{@[execname] = count()}'

dtrace: description 'pgin' matched 1 probe

^C

  xterm                                                             1

  ksh                                                               1

  ls                                                                2

  lpstat                                                            7

  sh                                                               17

  soffice                                                          39

  javaldx                                                         103

  soffice.bin                                                    3065
```

以上输出表明，与 StarSuite™ 软件关联的进程 soffice.bin 负责大多数调进页。为了更全面地了解 soffice.bin 在执行时的虚拟内存行为，可启用所有 vminfo 探测器。以下示例将在启动 StarSuite 软件时运行 dtrace(1M)：

```
dtrace -P vminfo'/execname == "soffice.bin"/{@[probename] = count()}'

dtrace: description 'vminfo' matched 42 probes

^C
```

```
  kernel_asflt                                                  1
  fspgin                                                       10
  pgout                                                        16
  execfree                                                     16
  execpgout                                                    16
  fsfree                                                       16
  fspgout                                                      16
  anonfree                                                     16
  anonpgout                                                    16
  pgpgout                                                      16
  dfree                                                        16
  execpgin                                                     80
  prot_fault                                                   85
  maj_fault                                                    88
  pgin                                                         90
  pgpgin                                                       90
  cow_fault                                                   859
  zfod                                                       1619
  pgfrec                                                     8811
  pgrec                                                      8827
  as_fault                                                   9495
```

以下脚本示例提供了更多有关 StarSuite 软件在启动时的虚拟内存行为的信息：

```
vminfo:::maj_fault,

vminfo:::zfod,
```

```
vminfo:::as_fault
/execname == "soffice.bin" && start == 0/
{
    /*
     * This is the first time that a vminfo probe has been hit; record
     * our initial timestamp.
     */
    start = timestamp;
}


vminfo:::maj_fault,
vminfo:::zfod,
vminfo:::as_fault
/execname == "soffice.bin"/
{
    /*
     * Aggregate on the probename, and lquantize() the number of seconds
     * since our initial timestamp.  (There are 1,000,000,000 nanoseconds
     * in a second.)  We assume that the script will be terminated before
     * 60 seconds elapses.
     */
    @[probename] =
        lquantize((timestamp - start) / 1000000000, 0, 60);
}
```

请在重新启动 StarSuite 软件时运行以上脚本。接下来，依次创建一个新的绘图、新的表示，然后关闭所有文件并退出应用程序。在运行该 D 脚本的 shell 中按 Ctrl-C 组合键。结果显示一段时间内某虚拟内存行为的情况：

```
# dtrace -s ./soffice.d

dtrace: script ’./soffice.d’ matched 10 probes

^C


 maj_fault

           value  ------------- Distribution ------------- count

               7 |                                         0

               8 |@@@@@@@@@@                               88

               9 |@@@@@@@@@@@@@@@@@@@@@@@@                 194

              10 |@                                        18

              11 |                                         0

              12 |                                         0

              13 |                                         2

              14 |                                         0

              15 |                                         1

              16 |@@@@@@@@@                                82

              17 |                                         0

              18 |                                         0

              19 |                                         2

              20 |                                         0


  zfod
```

```
value  ------------- Distribution ------------- count
  < 0 |                                         0
    0 |@@@@@@@                                  525
    1 |@@@@@@@@                                 605
    2 |@@                                       208
    3 |@@@                                      280
    4 |                                         4
    5 |                                         0
    6 |                                         0
    7 |                                         0
    8 |                                         44
    9 |@@                                       161
   10 |                                         2
   11 |                                         0
   12 |                                         0
   13 |                                         4
   14 |                                         0
   15 |                                         29
   16 |@@@@@@@@@@@@@@@                          1048
   17 |                                         24
   18 |                                         0
   19 |                                         0
   20 |                                         1
   21 |                                         0
```

```
              22 |                                         3
              23 |                                         0


  as_fault

           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@@@@@@@@@@@@                           4139
               1 |@@@@@@@                                  2249
               2 |@@@@@@@                                  2402
               3 |@                                        594
               4 |                                         56
               5 |                                         0
               6 |                                         0
               7 |                                         0
               8 |                                         189
               9 |@@                                       929
              10 |                                         39
              11 |                                         0
              12 |                                         0
              13 |                                         6
              14 |                                         0
              15 |                                         297
              16 |@@@@                                     1349
              17 |                                         24
```

```
18 |                                       0
19 |                                       21
20 |                                       1
21 |                                       0
22 |                                       92
23 |                                       0
```

以上输出显示了与虚拟内存系统有关的某种 StarSuite 行为。例如，在启动新的应用程序实例之前，不会触发 `maj_fault` 探测器。如您所愿，StarSuite 的“热启动”并未引发新的重大故障。`as_fault` 输出显示了活动的初始突发传输、用户查找菜单以创建新绘图时的等待时间、另一段空闲时间，以及在用户单击新表示时，出现活动的最终突发传输。`zfod` 输出表明，创建新的表示对用零填充的页面产生了巨大压力，不过，该压力仅持续了一小段时间。

在此示例中，DTrace 调查的下一次重复取决于您要了解的具体方面。如果要了解用零填充的页面的请求来源，可在 `zfod` 启用项中聚集 `ustack()`。您可能希望为用零填充的页面设置阈值，并使用 `stop()` 破坏性操作，以便在阈值超出时停止违例进程。借助这种方法，您可以使用更多传统调试工具（如 `truss`(1) 或 `mdb`(1)）。通过 `vminfo` 提供器，可将常规工具（如 `vmstat`(1M)）的输出中显示的统计信息，与导致该系统行为的应用程序进行关联。

稳定性

`vminfo` 提供器使用 DTrace 的稳定性机制说明其稳定性，如下表所示。有关稳定性机制的更多信息，请参见第 39 章。

元素	名称稳定性	数据稳定性	相关类
提供器	发展中	发展中	ISA
模块	专用	专用	未知
函数	专用	专用	未知
名称	发展中	发展中	ISA
参数	专用	专用	ISA

◆ ◆ ◆ 第 25 章

25 proc 提供器

proc 提供器提供了与以下活动有关的探测器：进程创建和终止、LWP 创建和终止、新程序映像的执行以及信号的发送和处理。

探测器

表 25–1 中对 proc 探测器进行了说明。

表 25–1 proc 探测器

探测器	说明
create	使用 fork(2)、forkall(2)、fork1(2) 或 vfork(2) 创建进程时将触发的探测器。args[0] 指向对应于新的子进程的 psinfo_t。可以通过检查派生线程 lwpsinfo_t 的 pr_flag 成员中的 PR_VFORKP，将 vfork 与其他 fork 变体区分开来。可以通过检查父进程的 psinfo_t (curpsinfo) 和子进程的 psinfo_t(args[0]) 的 pr_nlwp 成员，将 fork1 与 forkall 区分开来。因为 create 探测器仅在成功创建了进程之后触发，且 LWP 创建为创建进程的一部分，所以对于任何在创建进程时（为新进程触发 create 探测器之前）创建的 LWP，都将触发 lwp-create。
exec	进程使用 exec(2) 系统调用的以下变体装入新进程映像时将触发的探测器：exec(2)、execle(2)、execlp(2)、execv(2)、execve(2)、execvp(2)。exec 探测器在装入进程映像之前触发。因此进程变量（如 execname 和 curpsinfo）将包含装入映像前的进程状态。在触发 exec 探测器之后的某个时间，exec-failure 探测器或 exec-success 探测器随后将在同一线程中触发。args[0] 指向新进程映像的路径。
exec-failure	exec(2) 变体失败时将触发的探测器。exec-failure 探测器仅在同一线程中触发了 exec 探测器之后触发。args[0] 中提供了 errno(3C) 值。

表 25–1 proc 探测器 （续）

探测器	说明
exec-success	exec(2) 变体成功时将触发的探测器。与 exec-failure 探测器一样，exec-success 探测器仅在同一线程中触发了 exec 探测器之后触发。触发 exec-success 探测器时，进程变量（如 execname 和 curpsinfo）将包含装入新的进程映像之后的进程状态。
exit	当前线程正在退出时将触发的探测器。退出的原因（使用某个 SIGCHLD siginfo(3HEAD) 代码表示）包含在 args[0] 中。
fault	线程遇到计算机故障时将触发的探测器。故障代码（如 proc(4) 中所定义）包含在 args[0] 中。args[1] 指向对应于故障的 siginfo 结构。只有引起信号的那些故障可以触发 fault 探测器。
lwp-create	创建 LWP 时将触发的探测器，通常作为 thr_create(3C) 的结果。args[0] 指向对应于新线程的 lwpsinfo_t。args[1] 指向包含该线程的进程的 psinfo_t。
lwp-start	在新创建的 LWP 的上下文中触发的探测器。lwp-start 探测器将在执行任何用户级指令之前触发。如果该 LWP 是进程中的第一个 LWP，则 start 探测器将在 lwp-start 探测器之前触发。
lwp-exit	LWP 正在退出（由于信号或对 thr_exit(3C) 的显式调用）时将触发的探测器。
signal-discard	信号被发送到单线程进程，且进程未阻塞该信号，但忽略该信号时将触发的探测器。在这些条件下，信号在生成时将被废弃。目标进程和线程的 lwpsinfo_t 和 psinfo_t 分别包含在 args[0] 和 args[1] 中。信号数字包含在 args[2] 中。
signal-send	将信号发送到线程或进程时将触发的探测器。signal-send 探测器在发送进程和线程的上下文中触发。接收进程和线程的 lwpsinfo_t 和 psinfo_t 分别包含在 args[0] 和 args[1] 中。信号数字包含在 args[2] 中。接受进程和线程时，signal-send 始终后跟 signal-handle 或 signal-clear。
signal-handle	探测器就在线程处理信号之前的瞬间触发。signal-handle 探测器在将要处理信号的线程的上下文中触发。信号数字包含在 args[0] 中。指向对应于信号的 siginfo_t 结构的指针包含在 args[1] 中。进程中的信号处理程序的地址包含在 args[2] 中。
signal-clear	因为目标线程正在等待 sigwait(2)、sigwaitinfo(3RT) 或 sigtimedwait(3RT) 中的信号而清除暂挂信号时，将触发的探测器。在这些条件下，将清除暂挂信号并将信号数字返回到调用方。信号数字包含在 args[0] 中。signal-clear 在先前的等待线程的上下文中触发。
start	在新创建的进程的上下文中触发的探测器。start 探测器将在执行进程中任何用户级指令之前触发。

参数

表 25–2 中列出了 proc 探测器的参数类型。表 25–1 中说明了这些参数。

表 25–2 proc 探测器参数

探测器	args[0]	args[1]	args[2]
create	psinfo_t *	—	—
exec	char *	—	—
exec-failure	int	—	—
exit	int	—	—
fault	int	siginfo_t *	—
lwp-create	lwpsinfo_t *	psinfo_t *	—
lwp-start	—	—	—
lwp-exit	—	—	—
signal-discard	lwpsinfo_t *	psinfo_t *	int
signal-discard	lwpsinfo_t *	psinfo_t *	int
signal-send	lwpsinfo_t *	psinfo_t *	int
signal-handle	int	siginfo_t *	void (*)(void)
signal-clear	int	—	—
start	—	—	—

lwpsinfo_t

多种 proc 探测器具有 lwpsinfo_t（proc(4) 中记录的一种结构）类型的参数。DTrace 使用者可以使用的 lwpsinfo_t 结构的定义如下所示：

```
typedef struct lwpsinfo {

    int pr_flag;              /* flags; see below */

    id_t pr_lwpid;            /* LWP id */

    uintptr_t pr_addr;        /* internal address of thread */

    uintptr_t pr_wchan;       /* wait addr for sleeping thread */

    char pr_stype;            /* synchronization event type */
```

```
    char pr_state;            /* numeric thread state */
    char pr_sname;            /* printable character for pr_state */
    char pr_nice;             /* nice for cpu usage */
    short pr_syscall;         /* system call number (if in syscall) */
    int pr_pri;               /* priority, high value = high priority */
    char pr_clname[PRCLSZ];   /* scheduling class name */
    processorid_t pr_onpro;   /* processor which last ran this thread */
    processorid_t pr_bindpro; /* processor to which thread is bound */
    psetid_t pr_bindpset;     /* processor set to which thread is bound */
} lwpsinfo_t;
```

pr_flag 字段是存储用于说明进程的标志的位掩码。表 25–3 中说明了这些标志及其含义。

表 25–3 pr_flag 值

PR_ISSYS	该进程为系统进程。
PR_VFORKP	该进程是 vfork(2) 的子进程的父进程。
PR_FORK	该进程具有从 fork 中继承的模式集。
PR_RLC	该进程具有最后一次关闭时运行的模式集。
PR_KLC	该进程具有最后一次关闭时中止的模式集。
PR_ASYNC	该进程具有异步停止模式集。
PR_MSACCT	该进程已启用微状态记帐。
PR_MSFORK	进程微状态记帐继承于 fork。
PR_BPTADJ	该进程具有断点调整模式集。
PR_PTRACE	该进程具有 ptrace(3) 兼容的模式集。
PR_STOPPED	该线程是已停止的 LWP。
PR_ISTOP	该线程是在出现关注的事件时停止的 LWP。
PR_DSTOP	该线程是具有正在生效的停止指令的 LWP。

表 25–3 pr_flag 值 （续）

PR_STEP	该线程是具有正在生效的单步指令的 LWP。
PR_ASLEEP	该线程是系统调用的可中断休眠中的 LWP。
PR_DETACH	该线程是拆离的 LWP。请参见 pthread_create(3) 和 pthread_join(3)。
PR_DAEMON	该线程是守护进程 LWP。请参见 pthread_create(3)。
PR_AGENT	该线程是进程的代理 LWP。
PR_IDLE	该线程是 CPU 的空闲线程。当 CPU 的运行队列为空时，空闲线程仅在 CPU 中运行。

pr_addr 字段是表示线程的专用、内核中的数据结构的地址。虽然该数据结构为专用，但 pr_addr 字段可用作在线程的生命周期中线程的唯一标记。

当线程在同步对象上休眠时，将设置 pr_wchan 字段。pr_wchan 字段的含义专用于内核实现，但该字段也可用作同步对象的唯一标记。

当线程在同步对象上休眠时，将设置 pr_stype 字段。表 25–4 中说明了 pr_stype 字段的可能值。

表 25–4 pr_stype 值

SOBJ_MUTEX	内核互斥同步对象。用于串行化内核中共享数据区域的访问。有关内核互斥同步对象的详细信息，请参见第 18 章和 mutex_init(9F)。
SOBJ_RWLOCK	内核读取器/写入器同步对象。用于同步内核中共享对象的访问，共享对象可以允许多个并发读取器或一个写入器。有关读取器/写入器同步对象的详细信息，请参见第 18 章和 rwlock(9F)。
SOBJ_CV	条件变量同步对象。条件变量的设计是无限等待，直到某个条件变为 true。条件变量通常用于同步（出于访问共享数据区域之外的原因），它通常是当进程执行程序指令的无限等待时使用的机制。例如，在 poll(2)、pause(2)、wait(3C) 中阻塞，以及类似情况。
SOBJ_SEMA	信号同步对象。一般用途的同步对象，与条件变量对象一样，不跟踪拥有权说明。由于在 Solaris 内核中实现优先级继承必须具有拥有权，所以信号对象中没有固有拥有权将使这些对象不能被广泛使用。有关详细信息，请参见 semaphore(9F)。
SOBJ_USER	用户级同步对象。所有对用户级同步对象的阻塞都使用 SOBJ_USER 同步对象处理。用户级同步对象包括使用 mutex_init(3)、sema_init(3C)、rwlock_init(3C)、cond_init(3C) 及其 POSIX 等价项创建的对象。
SOBJ_USER_PI	实现优先级继承的用户级同步对象。一些跟踪拥有权的用户级同步对象还允许优先级继承。例如，使用 pthread_mutex_init(3) 创建的互斥对象可以使用 pthread_mutexattr_setprotocol(3) 设置为继承优先级。

表 25–4 pr_stype 值 （续）

SOBJ_SHUTTLE	互动同步对象。互动对象用于实现门。有关更多信息，请参见 door_create(3DOOR)。

pr_state 字段设置为表 25–5 中的某个值。pr_sname 字段设置为同一个表中括号内说明的相应字符。

表 25–5 pr_state 值

SSLEEP (S)	线程处于休眠状态。sched:::sleep 探测器就在线程的状态转换为 SSLEEP 之前的瞬间触发。
SRUN (R)	该线程可运行，但当前未运行。sched:::enqueue 探测器将在线程的状态转换为 SRUN 之前的瞬间触发。
SZOMB (Z)	该线程是僵 LWP。
SSTOP (T)	由于显式 proc(4) 指令或一些其他停止机制，该线程已停止。
SIDL (I)	该线程是进程创建期间的中间状态。
SONPROC (O)	该线程正在 CPU 中运行。sched:::on-cpu 探测器将在线程的状态转换为 SONPROC 一段时间后，在 SONPROC 线程的上下文中触发。

psinfo_t

多种 proc 探测器具有 psinfo_t（proc(4) 中记录的一种结构）类型的参数。DTrace 使用者可以使用的 psinfo_t 结构的定义如下所示：

```
typedef struct psinfo {

    int     pr_nlwp;            /* number of active lwps in the process */

    pid_t   pr_pid;             /* unique process id */

    pid_t   pr_ppid;            /* process id of parent */

    pid_t   pr_pgid;            /* pid of process group leader */

    pid_t   pr_sid;             /* session id */

    uid_t   pr_uid;             /* real user id */

    uid_t   pr_euid;            /* effective user id */

    gid_t   pr_gid;             /* real group id */

    gid_t   pr_egid;            /* effective group id */
```

```
    uintptr_t pr_addr;          /* address of process */
    dev_t   pr_ttydev;          /* controlling tty device (or PRNODEV) */
    timestruc_t pr_start;       /* process start time, from the epoch */
    char    pr_fname[PRFNSZ];   /* name of execed file */
    char    pr_psargs[PRARGSZ]; /* initial characters of arg list */
    int     pr_argc;            /* initial argument count */
    uintptr_t pr_argv;          /* address of initial argument vector */
    uintptr_t pr_envp;          /* address of initial environment vector */
    char    pr_dmodel;          /* data model of the process */
    taskid_t pr_taskid;         /* task id */
    projid_t pr_projid;         /* project id */
    poolid_t pr_poolid;         /* pool id */
    zoneid_t pr_zoneid;         /* zone id */
} psinfo_t;
```

pr_dmodel 字段设置为 PR_MODEL_ILP32（表示 32 位进程）或 PR_MODEL_LP64（表示 64 位进程）。

示例

exec

您可以使用 exec 探测器轻易地确定正在执行哪些程序以及由哪个用户执行，如下例所示：

```
#pragma D option quiet


proc:::exec
```

```
{
        self->parent = execname;
}

proc:::exec-success
/self->parent != NULL/
{
    @[self->parent, execname] = count();
    self->parent = NULL;
}

proc:::exec-failure
/self->parent != NULL/
{
    self->parent = NULL;
}

END
{
    printf("%-20s %-20s %s\n", "WHO", "WHAT", "COUNT");
    printa("%-20s %-20s %@d\n", @);
}
```

在生成计算机上运行示例脚本一段时间后将产生与以下示例类似的输出：

```
# dtrace -s ./whoexec.d
```

```
^C
WHO                 WHAT                COUNT
make.bin            yacc                1
tcsh                make                1
make.bin            spec2map            1
sh                  grep                1
lint                lint2               1
sh                  lint                1
sh                  ln                  1
cc                  ld                  1
make.bin            cc                  1
lint                lint1               1
sh                  lex                 1
make.bin            mv                  2
sh                  sh                  3
sh                  make                3
sh                  sed                 4
sh                  tr                  4
make                make.bin            4
sh                  install.bin         5
sh                  rm                  6
cc                  ir2hf               33
cc                  ube                 33
sh                  date                34
```

```
sh                  mcs                 34

cc                  acomp               34

sh                  cc                  34

sh                  basename            34

basename            expr                34

make.bin            sh                  87
```

start 和 exit

如果要知道程序从创建到终止所运行的时间，可以启用 start 和 exit 探测器，如下例所示：

```
proc:::start

{

    self->start = timestamp;

}



proc:::exit

/self->start/

{

    @[execname] = quantize(timestamp - self->start);

    self->start = 0;

}
```

在生成服务器上运行示例脚本几秒钟后将产生与以下示例类似的输出：

```
# dtrace -s ./progtime.d

dtrace: script ’./progtime.d’ matched 2 probes

^C
```

```
ir2hf

           value  ------------- Distribution ------------- count
         4194304 |                                         0
         8388608 |@                                        1
        16777216 |@@@@@@@@@@@@@@@@                         14
        33554432 |@@@@@@@@@@                               9
        67108864 |@@@                                      3
       134217728 |@                                        1
       268435456 |@@@@                                     4
       536870912 |@                                        1
      1073741824 |                                         0


ube

           value  ------------- Distribution ------------- count
        16777216 |                                         0
        33554432 |@@@@@@@                                  6
        67108864 |@@@                                      3
       134217728 |@@                                       2
       268435456 |@@@@                                     4
       536870912 |@@@@@@@@@@@@                             10
      1073741824 |@@@@@@@                                  6
      2147483648 |@@                                       2
      4294967296 |                                         0
```

```
acomp

           value  ------------- Distribution ------------- count
         8388608 |                                         0
        16777216 |@@                                       2
        33554432 |                                         0
        67108864 |@                                        1
       134217728 |@@@                                      3
       268435456 |                                         0
       536870912 |@@@@@                                    5
      1073741824 |@@@@@@@@@@@@@@@@@@@@@@@@@@               22
      2147483648 |@                                        1
      4294967296 |                                         0


cc

           value  ------------- Distribution ------------- count
        33554432 |                                         0
        67108864 |@@@                                      3
       134217728 |@                                        1
       268435456 |                                         0
       536870912 |@@@@                                     4
      1073741824 |@@@@@@@@@@@@@@@                          13
      2147483648 |@@@@@@@@@@@@@                            11
      4294967296 |@@@                                      3
```

```
    8589934592 |                                         0


sh

           value  ------------- Distribution ------------- count
          262144 |                                         0
          524288 |@                                        5
         1048576 |@@@@@@@                                  29
         2097152 |                                         0
         4194304 |                                         0
         8388608 |@@@                                      12
        16777216 |@@                                       9
        33554432 |@@                                       9
        67108864 |@@                                       8
       134217728 |@                                        7
       268435456 |@@@@@                                    20
       536870912 |@@@@@@                                   26
      1073741824 |@@@                                      14
      2147483648 |@@                                       11
      4294967296 |                                         3
      8589934592 |                                         1
     17179869184 |                                         0


make.bin

           value  ------------- Distribution ------------- count
```

```
         16777216 |                                         0
         33554432 |@                                        1
         67108864 |@                                        1
        134217728 |@@                                       2
        268435456 |                                         0
        536870912 |@@                                       2
       1073741824 |@@@@@@@@@@                               9
       2147483648 |@@@@@@@@@@@@@@@@                         14
       4294967296 |@@@@@@                                   6
       8589934592 |@@                                       2
      17179869184 |                                         0
```

lwp-start 和 lwp-exit

要知道特定进程运行的时间，可能需要知道各个线程运行的时间。以下示例说明了如何将 lwp-start 和 lwp-exit 探测器用于此目的：

```
proc:::lwp-start

/tid != 1/

{

    self->start = timestamp;

}



proc:::lwp-exit

/self->start/

{

    @[execname] = quantize(timestamp - self->start);
```

```
    self->start = 0;
}
```

在 NFS（网络文件系统）和日历服务器上运行示例脚本将产生与以下示例类似的输出：

```
# dtrace -s ./lwptime.d
dtrace: script ’./lwptime.d’ matched 3 probes
^C


  nscd

           value  ------------- Distribution ------------- count
          131072 |                                         0
          262144 |@                                        18
          524288 |@@                                       24
         1048576 |@@@@@@@                                  75
         2097152 |@@@@@@@@@@@@@@@@@@@@@@@@                 245
         4194304 |@@                                       22
         8388608 |@@                                       24
        16777216 |                                         6
        33554432 |                                         3
        67108864 |                                         1
       134217728 |                                         1
       268435456 |                                         0


  mountd

           value  ------------- Distribution ------------- count
```

```
          524288 |                                         0
         1048576 |@                                        15
         2097152 |@                                        24
         4194304 |@@@                                      51
         8388608 |@                                        17
        16777216 |@                                        24
        33554432 |@                                        15
        67108864 |@@@@                                     57
       134217728 |@                                        28
       268435456 |@                                        26
       536870912 |@@                                       39
      1073741824 |@@@                                      45
      2147483648 |@@@@@                                    72
      4294967296 |@@@@@                                    77
      8589934592 |@@@                                      55
     17179869184 |                                         14
     34359738368 |                                         2
     68719476736 |                                         0


automountd

           value  ------------- Distribution ------------- count
         1048576 |                                         0
         2097152 |                                         3
         4194304 |@@@@                                     146
```

```
         8388608 |                                         6
        16777216 |                                         6
        33554432 |                                         9
        67108864 |@@@@@                                    203
       134217728 |@@                                       87
       268435456 |@@@@@@@@@@@@@@@                          534
       536870912 |@@@@@@                                   223
      1073741824 |@                                        45
      2147483648 |                                         20
      4294967296 |                                         26
      8589934592 |                                         20
     17179869184 |                                         19
     34359738368 |                                         7
     68719476736 |                                         2
    137438953472 |                                         0


  iCald
           value  ------------- Distribution ------------- count
         8388608 |                                         0
        16777216 |@@@@@@@                                  20
        33554432 |@@@                                      9
        67108864 |@@                                       8
       134217728 |@@@@@                                    16
       268435456 |@@@@                                     11
```

```
       536870912 |@@@@                                     11
      1073741824 |@                                        4
      2147483648 |                                         2
      4294967296 |                                         0
      8589934592 |@@                                       8
     17179869184 |@                                        5
     34359738368 |@                                        4
     68719476736 |@@                                       6
    137438953472 |@                                        4
    274877906944 |                                         2
    549755813888 |                                         0
```

signal-send

您可以使用 `signal-send` 探测器确定与任何信号关联的发送和接收进程，如下例所示：

```
#pragma D option quiet


proc:::signal-send
{
    @[execname, stringof(args[1]->pr_fname), args[2]] = count();
}


END
{
    printf("%20s %20s %12s %s\n",
```

```
        "SENDER", "RECIPIENT", "SIG", "COUNT");

    printa("%20s %20s %12d %@d\n", @);

}
```

运行此脚本将会生成与以下示例类似的输出：

```
# dtrace -s ./sig.d

^C

              SENDER            RECIPIENT          SIG COUNT

               xterm                dtrace            2 1

               xterm           soffice.bin            2 1

                  tr                 init           18 1

               sched                 test           18 1

               sched                fvwm2           18 1

                bash                 bash           20 1

                 sed                 init           18 2

               sched                  ksh           18 15

               sched                 Xsun           22 471
```

稳定性

proc 提供器使用 DTrace 的稳定性机制说明其稳定性，如下表所示。有关稳定性机制的更多信息，请参见第 39 章。

元素	名称稳定性	数据稳定性	相关类
提供器	发展中	发展中	ISA
模块	专用	专用	未知
函数	专用	专用	未知

元素	名称稳定性	数据稳定性	相关类
名称	发展中	发展中	ISA
参数	发展中	发展中	ISA

◆ ◆ ◆ 第 26 章

26 sched 提供器

sched 提供器提供了与 CPU 调度相关的探测器。因为 CPU 是所有线程都必须使用的一种资源，所以 sched 提供器对于了解系统行为很有帮助。例如，通过使用 sched 提供器，您可以了解线程休眠、运行、更改优先级或唤醒其他线程的时间和原因。

探测器

表 26–1 中对 sched 探测器进行了说明。

表 26–1 sched 探测器

探测器	说明
change-pri	要更改线程的优先级时将触发的探测器。args[0] 指向线程的 lwpsinfo_t。线程的当前优先级在此结构的 pr_pri 字段中。args[1] 指向包含该线程进程的 psinfo_t。线程的新优先级包含在 args[2] 中。
dequeue	探测器就在可运行线程离开运行队列之前的瞬间触发。args[0] 指向要离开队列线程的 lwpsinfo_t。args[1] 指向包含该线程进程的 psinfo_t。args[2] 指向线程将离开队列 CPU 的 cpuinfo_t。如果线程要从与特定 CPU 无关的队列中离开，则此结构的 cpu_id 成员将为 -1。
enqueue	探测器就在可运行线程进入运行队列之前的瞬间触发。args[0] 指向要进入队列线程的 lwpsinfo_t。args[1] 指向包含该线程进程的 psinfo_t。args[2] 指向线程要进入的队列所在 CPU 的 cpuinfo_t。如果线程要进入与特定 CPU 无关的运行队列中，则此结构的 cpu_id 成员将为 -1。args[3] 中的值是一个布尔值，指示线程是否排在运行队列的前面。如果线程排在运行队列的前面，则此值为非零，如果线程排在运行队列的后面，则此值为零。

表 26–1 sched 探测器 （续）

探测器	说明
off-cpu	当前 CPU 结束执行线程时将触发的探测器。curcpu 变量指示当前 CPU。curlwpsinfo 变量指示要结束执行的线程。curpsinfo 变量说明包含当前线程的进程。args[0] 指向当前 CPU 接下来要执行线程的 lwpsinfo_t 结构。args[1] 指向包含下一个线程进程的 psinfo_t。
on-cpu	CPU 开始执行线程时将触发的探测器。curcpu 变量指示当前 CPU。curlwpsinfo 变量指示要开始执行的线程。curpsinfo 变量说明包含当前线程的进程。
preempt	探测器就在当前线程被抢占之前的瞬间触发。触发此探测器后，当前线程将选择要运行的线程，并且将为当前线程触发 off-cpu 探测器。在某些情况下，一个 CPU 中的线程将被抢占，但与此同时抢占线程将在另一个 CPU 中运行。在此情况下，将触发 preempt 探测器，但分发程序将找不到要运行的更高优先级线程，因此将触发 remain-cpu 探测器而不是 off-cpu 探测器。
remain-cpu	已做出调度决策，但分发程序选择继续运行当前线程时，将触发的探测器。curcpu 变量指示当前 CPU。curlwpsinfo 变量指示要开始执行的线程。curpsinfo 变量说明包含当前线程的进程。
schedctl-nopreempt	当线程被抢占，然后重新排在运行队列的前面（由于抢占控制请求）时，将触发的探测器。有关抢占控制的详细信息，请参见 schedctl_init(3C)。与 preempt 一样，off-cpu 或 remain-cpu 将在 schedctl-nopreempt 之后触发。因为 schedctl-nopreempt 表示将当前线程排在运行队列的前面，所以 remain-cpu 更有可能在 schedctl-nopreempt（而不是 off-cpu）之后触发。args[0] 指向要被抢占线程的 lwpsinfo_t。args[1] 指向包含该线程进程的 psinfo_t。
schedctl-preempt	使用抢占控制的线程仍然被抢占，并重新排在运行队列的后面时，将触发的探测器。有关抢占控制的详细信息，请参阅 schedctl_init(3C)。与 preempt 一样，off-cpu 或 remain-cpu 将在 schedctl-preempt 之后触发。与 preempt 一样（与 schedctl-nopreempt 不同），schedctl-preempt 表示将当前线程重排在运行队列的后面。因此，off-cpu 更有可能在 schedctl-preempt（而不是 remain-cpu）之后触发。args[0] 指向要被抢占线程的 lwpsinfo_t。args[1] 指向包含该线程进程的 psinfo_t。
schedctl-yield	启用了抢占控制并且以人工方式扩展其时间片的线程，执行代码以将 CPU 提供给其他线程时，将触发的探测器。
sleep	探测器就在当前线程在同步对象上休眠之前的瞬间触发。同步对象的类型包含在 curlwpsinfo 指向的 lwpsinfo_t 的 pr_stype 成员中。同步对象的地址包含在 curlwpsinfo 指向的 lwpsinfo_t 的 pr_wchan 成员中。此地址表示专用实现详细信息，但地址值可能被视为同步对象的唯一标记。
surrender	一个 CPU 指示另一个 CPU 做出调度决策（通常是因为更高优先级线程变得可运行）时将触发的探测器。

表 26–1 sched 探测器 （续）

探测器	说明
tick	作为基于时钟周期记帐的一部分触发的探测器。在基于时钟周期的记帐中，CPU 记帐是通过检查在触发固定时间间隔中断时，哪些线程和进程正在运行来执行的。args[0] 指向对应于要对其分配 CPU 时间线程的 lwpsinfo_t。args[1] 指向对应于包含该线程进程的 psinfo_t。
wakeup	探测器就在当前线程唤醒同步对象的休眠线程之前的瞬间触发。args[0] 指向休眠线程的 lwpsinfo_t。args[1] 指向包含该休眠线程进程的 psinfo_t。同步对象的类型包含在休眠线程的 lwpsinfo_t 的 pr_stype 成员中。同步对象的地址包含在休眠线程的 lwpsinfo_t 的 pr_wchan 成员中。此地址表示专用实现详细信息，但地址值可能被视为同步对象的唯一标记。

参数

sched 探测器的参数类型列出在表 26–2 中，参数将在表 26–1 中说明。

表 26–2 sched 探测器参数

探测器	args[0]	args[1]	args[2]	args[3]
change-pri	lwpsinfo_t *	psinfo_t *	pri_t	—
dequeue	lwpsinfo_t *	psinfo_t *	cpuinfo_t *	—
enqueue	lwpsinfo_t *	psinfo_t *	cpuinfo_t *	int
off-cpu	lwpsinfo_t *	psinfo_t *	—	—
on-cpu	—	—	—	—
preempt	—	—	—	—
remain-cpu	—	—	—	—
schedctl-nopreempt	lwpsinfo_t *	psinfo_t *	—	—
schedctl-preempt	lwpsinfo_t *	psinfo_t *	—	—
schedctl-yield	lwpsinfo_t *	psinfo_t *	—	—
sleep	—	—	—	—
surrender	lwpsinfo_t *	psinfo_t *	—	—
tick	lwpsinfo_t *	psinfo_t *	—	—
wakeup	lwpsinfo_t *	psinfo_t *	—	—

如表 26–2 中所示，许多 sched 探测器的参数由指向 lwpsinfo_t 的指针和指向 psinfo_t 的指针组成，它们分别指示线程和包含该线程的进程。这些结构将分别在第 281 页中的 “lwpsinfo_t” 和第 284 页中的 “psinfo_t” 中详细介绍。

cpuinfo_t

cpuinfo_t 结构定义 CPU。如表 26–2 中所示，enqueue 和 dequeue 探测器的参数包括指向 cpuinfo_t 的指针。此外，curcpu 变量指向对应于当前 CPU 的 cpuinfo_t。cpuinfo_t 结构的定义如下所示：

```
typedef struct cpuinfo {

    processorid_t cpu_id;           /* CPU identifier */

    psetid_t cpu_pset;              /* processor set identifier */

    chipid_t cpu_chip;              /* chip identifier */

    lgrp_id_t cpu_lgrp;             /* locality group identifer */

    processor_info_t cpu_info;      /* CPU information */

} cpuinfo_t;
```

cpu_id 成员是 psrinfo(1M) 和 p_online(2) 返回的处理器标识符。

cpu_pset 成员是包含 CPU（如果存在）的处理器集。有关处理器集的更多详细信息，请参见 psrset(1M)。

cpu_chip 成员是物理芯片的标识符。物理芯片可能包含若干 CPU。有关更多信息，请参见 psrinfo(1M)。

cpu_lgrp 成员是与 CPU 关联的延迟组的标识符。有关延迟组的详细信息，请参见 liblgrp(3LIB)。

cpu_info 成员是与 CPU 相关联的 processor_info_t 结构（由 processor_info(2) 返回）。

示例

on-cpu 和 off-cpu

可能遇到的一个常见问题是哪些 CPU 在运行线程，将运行多久。此问题可以通过使用 on-cpu 和 off-cpu 探测器在系统范围内轻松回答，如下例所示：

```
sched:::on-cpu
{
    self->ts = timestamp;
}


sched:::off-cpu
/self->ts/
{
    @[cpu] = quantize(timestamp - self->ts);
    self->ts = 0;
}
```

运行上面的脚本将会生成与以下示例类似的输出：

```
# dtrace -s ./where.d
dtrace: script ’./where.d’ matched 5 probes
^C


        0
           value  ------------- Distribution ------------- count
            2048 |                                         0
            4096 |@@                                       37
            8192 |@@@@@@@@@@@@@@                           212
           16384 |@                                        30
           32768 |                                         10
           65536 |@                                        17
```

```
  131072 |                                         12
  262144 |                                         9
  524288 |                                         6
 1048576 |                                         5
 2097152 |                                         1
 4194304 |                                         3
 8388608 |@@@@                                     75
16777216 |@@@@@@@@@@@@                             201
33554432 |                                         6
67108864 |                                         0

1
   value  ------------- Distribution ------------- count
    2048 |                                         0
    4096 |@                                        6
    8192 |@@@@                                     23
   16384 |@@@                                      18
   32768 |@@@@                                     22
   65536 |@@@@                                     22
  131072 |@                                        7
  262144 |                                         5
  524288 |                                         2
 1048576 |                                         3
 2097152 |@                                        9
```

```
         4194304 |                                         4
         8388608 |@@@                                      18
        16777216 |@@@                                      19
        33554432 |@@@                                      16
        67108864 |@@@@                                     21
       134217728 |@@                                       14
       268435456 |                                         0
```

以上输出显示，在 CPU 1 中，线程运行的时间小于 100 微秒，或者约 10 毫秒。两组数据之间的显著差异显示在直方图中。或许您还想了解哪些 CPU 在运行某个特定进程。同样可以使用 `on-cpu` 和 `off-cpu` 探测器来回答此问题。以下脚本显示哪些 CPU 运行指定的应用程序超过 10 秒：

```
#pragma D option quiet


dtrace:::BEGIN
{
    start = timestamp;
}


sched:::on-cpu
/execname == $$1/
{
    self->ts = timestamp;
}


sched:::off-cpu
```

```
/self->ts/

{

    @[cpu] = sum(timestamp - self->ts);

    self->ts = 0;

}



profile:::tick-1sec

/++x == 10/

{

    exit(0);

}



dtrace:::END

{

    printf("CPU distribution over %d seconds:\n\n",

        (timestamp - start) / 1000000000);

    printf("CPU microseconds\n--- ------------\n");

    normalize(@, 1000);

    printa("%3d %@d\n", @);

}
```

在大型邮件服务器上运行以上脚本并指定 IMAP（Internet Message Access Protocol，Internet 消息访问协议）守护进程将生成与以下示例类似的输出：

```
# dtrace -s ./whererun.d imapd

CPU distribution of imapd over 10 seconds:
```

```
CPU microseconds

--- ------------

 15 10102

 12 16377

 21 25317

 19 25504

 17 35653

 13 41539

 14 46669

 20 57753

 22 70088

 16 115860

 23 127775

 18 160517
```

Solaris 在选择要运行线程的 CPU 时会考虑线程已休眠的时间： 休眠时间不足的线程不会迁移。可使用 `off-cpu` 和 `on-cpu` 探测器来观察此行为：

```
sched:::off-cpu

/curlwpsinfo->pr_state == SSLEEP/

{

    self->cpu = cpu;

    self->ts = timestamp;

}



sched:::on-cpu
```

```
/self->ts/

{

    @[self->cpu == cpu ?

        "sleep time, no CPU migration" : "sleep time, CPU migration"] =

        lquantize((timestamp - self->ts) / 1000000, 0, 500, 25);

    self->ts = 0;

    self->cpu = 0;

}
```

运行以上脚本约 30 秒将会生成与以下示例类似的输出：

```
# dtrace -s ./howlong.d

dtrace: script ’./howlong.d’ matched 5 probes

^C

 sleep time, CPU migration

           value  -------------- Distribution ------------ count

             < 0 |                                         0

               0 |@@@@@@@                                  6838

              25 |@@@@@                                    4714

              50 |@@@                                      3108

              75 |@                                        1304

             100 |@                                        1557

             125 |@                                        1425

             150 |                                         894

             175 |@                                        1526

             200 |@@                                       2010
```

```
           225 |@@                                       1933
           250 |@@                                       1982
           275 |@@                                       2051
           300 |@@                                       2021
           325 |@                                        1708
           350 |@                                        1113
           375 |                                         502
           400 |                                         220
           425 |                                         106
           450 |                                         54
           475 |                                         40
       >= 500 |@                                        1716


sleep time, no CPU migration

         value  -------------- Distribution ------------ count
           < 0 |                                         0
             0 |@@@@@@@@@@@@@@@                          58413
            25 |@@@                                      14793
            50 |@@                                       10050
            75 |                                         3858
           100 |@                                        6242
           125 |@                                        6555
           150 |                                         3980
           175 |@                                        5987
```

```
           200 |@                                        9024
           225 |@                                        9070
           250 |@@                                       10745
           275 |@@                                       11898
           300 |@@                                       11704
           325 |@@                                       10846
           350 |@                                        6962
           375 |                                         3292
           400 |                                         1713
           425 |                                         585
           450 |                                         201
           475 |                                         96
       >= 500 |                                         3946
```

示例输出显示非迁移的出现次数大大超过迁移的出现次数。而且，休眠时间越长，发生迁移的可能性越大。100 毫秒子范围中的分布明显不同，但因为休眠时间变长，所以看起来很相似。此结果指示，一但超出特定阈值，在做出调度决策时就不会考虑休眠时间。

使用 `off-cpu` 和 `on-cpu` 的最后示例说明如何使用这些探测器和 `pr_stype` 字段来确定线程休眠的原因和时间：

```
sched:::off-cpu

/curlwpsinfo->pr_state == SSLEEP/

{

    /*

     * We're sleeping.  Track our sobj type.

     */

    self->sobj = curlwpsinfo->pr_stype;

    self->bedtime = timestamp;
```

```
}


sched:::off-cpu
/curlwpsinfo->pr_state == SRUN/
{
    self->bedtime = timestamp;
}


sched:::on-cpu
/self->bedtime && !self->sobj/
{
    @["preempted"] = quantize(timestamp - self->bedtime);
    self->bedtime = 0;
}


sched:::on-cpu
/self->sobj/
{
    @[self->sobj == SOBJ_MUTEX ? "kernel-level lock" :
        self->sobj == SOBJ_RWLOCK ? "rwlock" :
        self->sobj == SOBJ_CV ? "condition variable" :
        self->sobj == SOBJ_SEMA ? "semaphore" :
        self->sobj == SOBJ_USER ? "user-level lock" :
        self->sobj == SOBJ_USER_PI ? "user-level prio-inheriting lock" :
```

```
        self->sobj == SOBJ_SHUTTLE ? "shuttle" : "unknown"] =
        quantize(timestamp - self->bedtime);


    self->sobj = 0;
    self->bedtime = 0;
}
```

运行以上脚本若干秒将会生成与以下示例类似的输出：

```
# dtrace -s ./whatfor.d
dtrace: script ’./whatfor.d’ matched 12 probes
^C

 kernel-level lock

           value  -------------- Distribution ------------ count
           16384 |                                         0
           32768 |@@@@@@@@                                 3
           65536 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  11
          131072 |@@                                       1
          262144 |                                         0


  preempted

           value  -------------- Distribution ------------ count
           16384 |                                         0
           32768 |                                         4
           65536 |@@@@@@@@                                 408
          131072 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  1031
```

```
          262144 |@@@                                      156
          524288 |@@                                       116
         1048576 |@                                        51
         2097152 |                                         42
         4194304 |                                         16
         8388608 |                                         15
        16777216 |                                         4
        33554432 |                                         8
        67108864 |                                         0


semaphore

          value  -------------- Distribution ------------ count
          32768 |                                         0
          65536 |@@                                       61
         131072 |@@@@@@@@@@@@@@@@@@@@@@@@@@               553
         262144 |@@                                       63
         524288 |@                                        36
        1048576 |                                         7
        2097152 |                                         22
        4194304 |@                                        44
        8388608 |@@@                                      84
       16777216 |@                                        36
       33554432 |                                         3
       67108864 |                                         6
```

```
      134217728 |                                         0
      268435456 |                                         0
      536870912 |                                         0
     1073741824 |                                         0
     2147483648 |                                         0
     4294967296 |                                         0
     8589934592 |                                         0
    17179869184 |                                         1
    34359738368 |                                         0


shuttle
          value  -------------- Distribution ------------ count
          32768 |                                         0
          65536 |@@@@@                                    2
         131072 |@@@@@@@@@@@@@@@@@                        6
         262144 |@@@@@                                    2
         524288 |                                         0
        1048576 |                                         0
        2097152 |                                         0
        4194304 |@@@@@                                    2
        8388608 |                                         0
       16777216 |                                         0
       33554432 |                                         0
       67108864 |                                         0
```

```
      134217728 |                                         0
      268435456 |                                         0
      536870912 |                                         0
     1073741824 |                                         0
     2147483648 |                                         0
     4294967296 |@@@@@                                    2
     8589934592 |                                         0
    17179869184 |@@                                       1
    34359738368 |                                         0


condition variable

          value  -------------- Distribution ------------ count
          32768 |                                         0
          65536 |                                         122
         131072 |@@@@@                                    1579
         262144 |@                                        340
         524288 |                                         268
        1048576 |@@@                                      1028
        2097152 |@@@                                      1007
        4194304 |@@@                                      1176
        8388608 |@@@@                                     1257
       16777216 |@@@@@@@@@@@@@@@                          4385
       33554432 |                                         295
       67108864 |                                         157
```

```
       134217728 |                                         96
       268435456 |                                         48
       536870912 |                                         144
      1073741824 |                                         10
      2147483648 |                                         22
      4294967296 |                                         18
      8589934592 |                                         5
     17179869184 |                                         6
     34359738368 |                                         4
     68719476736 |                                         0
```

enqueue 和 dequeue

CPU 空闲时，分发程序会查找其他（非空闲）CPU 中已进入队列的工作。以下示例使用 dequeue 探测器来了解传送应用程序的频率以及传送应用程序的 CPU：

```
#pragma D option quiet


sched:::dequeue
/args[2]->cpu_id != -1 && cpu != args[2]->cpu_id &&
    (curlwpsinfo->pr_flag & PR_IDLE)/
{
    @[stringof(args[1]->pr_fname), args[2]->cpu_id] =
        lquantize(cpu, 0, 100);
}


END
```

```
{
    printa("%s stolen from CPU %d by:\n%@d\n", @);
}
```

在 4 CPU 系统上运行以上脚本生成的输出后半部分与以下示例类似：

```
# dtrace -s ./whosteal.d
^C
...
 nscd stolen from CPU 1 by:


           value  -------------- Distribution ------------ count
               1 |                                         0
               2 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 28
               3 |                                         0


snmpd stolen from CPU 1 by:


           value  -------------- Distribution ------------ count
             < 0 |                                         0
               0 |@                                        1
               1 |                                         0
               2 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@     31
               3 |@@                                       2
               4 |                                         0
```

```
sched stolen from CPU 1 by:

           value  ------------- Distribution ------------ count
             < 0 |                                         0
               0 |@@                                       3
               1 |                                         0
               2 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@         36
               3 |@@@@                                     5
               4 |                                         0
```

您可能不需要了解 CPU 的工作内容，而只想了解进程和线程正在等待运行的 CPU。可以使用 enqueue 和 dequeue 探测器来回答此问题：

```
sched:::enqueue
{
    self->ts = timestamp;
}

sched:::dequeue
/self->ts/
{
    @[args[2]->cpu_id] = quantize(timestamp - self->ts);
    self->ts = 0;
}
```

运行以上脚本若干秒将会生成与以下示例类似的输出：

```
# dtrace -s ./qtime.d
dtrace: script ’./qtime.d’ matched 5 probes
^C

       -1

           value  -------------- Distribution ------------ count
            4096 |                                         0
            8192 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 2
           16384 |                                         0


        0

           value  -------------- Distribution ------------ count
            1024 |                                         0
            2048 |@@@@@@@@@@@@@@@                          262
            4096 |@@@@@@@@@@@@@                            227
            8192 |@@@@@                                    87
           16384 |@@@                                      54
           32768 |                                         7
           65536 |                                         9
          131072 |                                         1
          262144 |                                         5
          524288 |                                         4
         1048576 |                                         2
         2097152 |                                         0
         4194304 |                                         0
```

```
   8388608 |                                         0
  16777216 |                                         1
  33554432 |                                         2
  67108864 |                                         2
 134217728 |                                         0
 268435456 |                                         0
 536870912 |                                         0
1073741824 |                                         1
2147483648 |                                         1
4294967296 |                                         0


  1
      value  -------------- Distribution ------------ count
       1024 |                                         0
       2048 |@@@@                                     49
       4096 |@@@@@@@@@@@@@@@@@@@@                     241
       8192 |@@@@@@@                                  91
      16384 |@@@@                                     55
      32768 |                                         7
      65536 |                                         3
     131072 |                                         2
     262144 |                                         1
     524288 |                                         0
    1048576 |                                         0
```

```
         2097152 |                                         0
         4194304 |                                         0
         8388608 |                                         0
        16777216 |                                         0
        33554432 |                                         3
        67108864 |                                         1
       134217728 |                                         4
       268435456 |                                         2
       536870912 |                                         0
      1073741824 |                                         3
      2147483648 |                                         2
      4294967296 |                                         0
```

请注意示例输出底部的非零值。这些数据点显示两个 CPU 上的若干实例，其中线程已进入运行队列若干秒。

您可能不需要查看等待时间，而要检查一段时间内运行队列的长度。通过使用 enqueue 和 dequeue 探测器，可以建立关联数组来跟踪队列长度：

```
sched:::enqueue
{
    this->len = qlen[args[2]->cpu_id]++;
    @[args[2]->cpu_id] = lquantize(this->len, 0, 100);
}


sched:::dequeue
/qlen[args[2]->cpu_id]/
{
```

```
    qlen[args[2]->cpu_id]–;
}
```

在基本空闲的单处理器便携式系统上运行以上脚本约 30 秒将会生成与以下示例类似的输出：

```
# dtrace -s ./qlen.d
dtrace: script ’./qlen.d’ matched 5 probes
^C
        0
           value  -------------- Distribution ------------ count
             < 0 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@              110626
               1 |@@@@@@@@@@                               41142
               2 |@@                                       12655
               3 |@                                        5074
               4 |                                         1722
               5 |                                         701
               6 |                                         302
               7 |                                         63
               8 |                                         23
               9 |                                         12
              10 |                                         24
              11 |                                         58
              12 |                                         14
              13 |                                         3
              14 |                                         0
```

此输出能大体上反映出预期在空闲系统上看到的内容：可运行线程入队占用了大部分时间，运行队列非常短（三个或更少的线程）。但是，如果系统基本空闲，则表底部的异常数据点可能不会出现。例如，为什么运行队列有 13 个可运行线程？要研究此问题，可编写 D 脚本以在运行队列过长时显示运行队列的内容。因为 D 启用项不能在数据结构上迭加，因此不能在整个运行队列上简单迭加，所以此问题非常复杂。即使 D 启用项可以这样做，也应该避免对内核的内部数据结构产生依赖性。

对于此类型的脚本，应启用 enqueue 和 dequeue 探测器并使用随机和关联数组。每次线程入队时，脚本将递增队列的长度并在由线程控制的关联数组中记录时间标记。由于一个线程可能会让另一个线程入队，所以在此情况下不能使用线程局部变量。于是脚本将检查队列长度是否超出最大值。 如果超出最大值，则脚本会启动新的随机操作并记录时间标记和新的最大值。当线程离开队列时，脚本会将入队时间标记与最长长度时间标记进行比较：如果线程在最长长度的时间标记之前入队，则在记录最长长度时线程位于队列中。在此情况下，脚本将随机跟踪线程的信息。一旦内核允许在最长长度的时间标记时入队的最后一个线程离开队列，线程将提交随机数据。此脚本如下所示：

```
#pragma D option quiet

#pragma D option nspec=4

#pragma D option specsize=100k



int maxlen;

int spec[int];



sched:::enqueue

{

    this->len = ++qlen[this->cpu = args[2]->cpu_id];

    in[args[0]->pr_addr] = timestamp;

}



sched:::enqueue

/this->len > maxlen && spec[this->cpu]/

{
```

```
    /*
     * There is already a speculation for this CPU.  We just set a new
     * record, so we'll discard the old one.
     */
    discard(spec[this->cpu]);
}


sched:::enqueue
/this->len > maxlen/
{
    /*
     * We have a winner.  Set the new maximum length and set the timestamp
     * of the longest length.
     */
    maxlen = this->len;
    longtime[this->cpu] = timestamp;


    /*
     * Now start a new speculation, and speculatively trace the length.
     */
    this->spec = spec[this->cpu] = speculation();
    speculate(this->spec);
    printf("Run queue of length %d:\n", this->len);
}
```

```
sched:::dequeue
/(this->in = in[args[0]->pr_addr]) &&
    this->in <= longtime[this->cpu = args[2]->cpu_id]/
{
    speculate(spec[this->cpu]);
    printf("  %d/%d (%s)\n",
        args[1]->pr_pid, args[0]->pr_lwpid,
        stringof(args[1]->pr_fname));
}

sched:::dequeue
/qlen[args[2]->cpu_id]/
{
    in[args[0]->pr_addr] = 0;
    this->len = --qlen[args[2]->cpu_id];
}

sched:::dequeue
/this->len == 0 && spec[this->cpu]/
{
    /*
     * We just processed the last thread that was enqueued at the time
     * of longest length; commit the speculation, which by now contains
```

```
     * each thread that was enqueued when the queue was longest.
     */
    commit(spec[this->cpu]);
    spec[this->cpu] = 0;
}
```

在同一单处理器膝上型计算机上运行以上脚本将生成与以下示例类似的输出：

```
# dtrace -s ./whoqueue.d
Run queue of length 3:
 0/0 (sched)
  0/0 (sched)
  101170/1 (dtrace)
Run queue of length 4:
  0/0 (sched)
  100356/1 (Xsun)
  100420/1 (xterm)
  101170/1 (dtrace)
Run queue of length 5:
  0/0 (sched)
  0/0 (sched)
  100356/1 (Xsun)
  100420/1 (xterm)
  101170/1 (dtrace)
Run queue of length 7:
  0/0 (sched)
```

```
  100221/18 (nscd)

  100221/17 (nscd)

  100221/16 (nscd)

  100221/13 (nscd)

  100221/14 (nscd)

  100221/15 (nscd)

Run queue of length 16:

  100821/1 (xterm)

  100768/1 (xterm)

  100365/1 (fvwm2)

  101118/1 (xterm)

  100577/1 (xterm)

  101170/1 (dtrace)

  101020/1 (xterm)

  101089/1 (xterm)

  100795/1 (xterm)

  100741/1 (xterm)

  100710/1 (xterm)

  101048/1 (xterm)

  100697/1 (MozillaFirebird-)

  100420/1 (xterm)

  100394/1 (xterm)

  100368/1 (xterm)

^C
```

此输出显示出现最长的运行队列是因为有许多可运行 xterm 进程。此实验脚本与虚拟桌面中的更改有冲突，这些结果可能是因为某种 X 事件处理造成的。

sleep 和 wakeup

在第 316 页中的 “enqueue 和 dequeue”中，最后示例说明因为可运行 xterm 进程而导致运行队列长度剧增。一种猜测是这种现象是因为虚拟桌面中的更改而造成的。要验证此猜测，可使用 wakeup 探测器来确定是谁唤醒了 xterm 进程以及是何时唤醒的，如下例所示：

```
#pragma D option quiet



dtrace:::BEGIN

{

    start = timestamp;

}



sched:::wakeup

/stringof(args[1]->pr_fname) == "xterm"/

{

    @[execname] = lquantize((timestamp - start) / 1000000000, 0, 10);

}



profile:::tick-1sec

/++x == 10/

{

    exit(0);

}
```

要验证此猜测，请运行以上脚本，等待大约 5 秒，然后切换一次虚拟桌面。如果 xterm 进程的剧增是因为切换虚拟桌面造成的，则输出应该在 5 秒标记处显示唤醒活动剧增。

```
# dtrace -s ./xterm.d


  Xsun


           value  -------------- Distribution ------------ count
               4 |                                         0
               5 |@                                        1
               6 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@   32
               7 |                                         0
```

此输出显示 X 服务器正在唤醒切换虚拟桌面时聚集的 xterm 进程。如果要了解 X 服务器与 xterm 进程之间的交互，可在 X 服务器触发 wakeup 探测器时聚集用户栈跟踪。

要了解像 X 窗口系统这样的客户机/服务器系统的性能，您需要了解服务器以其名义运行的客户机。常规性能分析工具很难回答这类问题。但是，如果某个模型中的客户机向服务器发送一条消息并且休眠，而且是在暂挂服务器处理时休眠，则可以使用 wakeup 探测器来确定对其执行请求的客户机，如下例所示：

```
self int last;


sched:::wakeup

/self->last && args[0]->pr_stype == SOBJ_CV/

{

    @[stringof(args[1]->pr_fname)] = sum(vtimestamp - self->last);

    self->last = 0;

}


sched:::wakeup

/execname == "Xsun" && self->last == 0/
```

```
{

    self->last = vtimestamp;

}
```

运行上面的脚本将会生成与以下示例类似的输出：

```
dtrace -s ./xwork.d

dtrace: script ’./xwork.d’ matched 14 probes

^C

  xterm                                                    9522510

  soffice.bin                                              9912594

  fvwm2                                                  100423123

  MozillaFirebird                                        312227077

  acroread                                               345901577
```

此输出显示许多 Xsun 工作都是以进程 acroread、MozillaFirebird 的名义完成的，很少以 fvwm2 的名义完成。请注意，此脚本仅检查条件变量同步对象 (SOBJ_CV) 中的唤醒次数。 如表 25–4 中所述，条件变量为通常用于同步（出于访问共享数据区域之外的原因）的同步对象类型。 如果使用 X 服务器，客户机将通过对条件变量休眠来等待管道中的数据。

而且，您还可使用 sleep 探测器和 wakeup 探测器来了解哪些应用程序在哪些应用程序上阻塞，以及阻塞多长时间，如下例所示：

```
#pragma D option quiet



sched:::sleep

/!(curlwpsinfo->pr_flag & PR_ISSYS) && curlwpsinfo->pr_stype == SOBJ_CV/

{

    bedtime[curlwpsinfo->pr_addr] = timestamp;

}
```

```
sched:::wakeup
/bedtime[args[0]->pr_addr]/
{
    @[stringof(args[1]->pr_fname), execname] =
        quantize(timestamp - bedtime[args[0]->pr_addr]);
    bedtime[args[0]->pr_addr] = 0;
}

END
{
    printa("%s sleeping on %s:\n%@d\n", @);
}
```

在桌面系统上运行以上示例若干秒生成的输出后半部分与以下示例类似：

```
# dtrace -s ./whofor.d
^C
...
 xterm sleeping on Xsun:

           value  -------------- Distribution ------------ count
          131072 |                                         0
          262144 |                                         12
          524288 |                                         2
         1048576 |                                         0
         2097152 |                                         5
```

```
           4194304 |@@@                                      45
           8388608 |                                        1
          16777216 |                                        9
          33554432 |@@@@@                                   83
          67108864 |@@@@@@@@@@@                             164
         134217728 |@@@@@@@@@@                              147
         268435456 |@@@@                                    56
         536870912 |@                                       17
        1073741824 |                                        9
        2147483648 |                                        1
        4294967296 |                                        3
        8589934592 |                                        1
       17179869184 |                                        0


fvwm2 sleeping on Xsun:


           value  -------------- Distribution ------------ count
           32768 |                                        0
           65536 |@@@@@@@@@@@@@@@@@@@@@@@                 67
          131072 |@@@@@                                   16
          262144 |@@                                      6
          524288 |@                                       3
         1048576 |@@@@@                                   15
         2097152 |                                        0
```

```
           4194304 |                                         0
           8388608 |                                         1
          16777216 |                                         0
          33554432 |                                         0
          67108864 |                                         1
         134217728 |                                         0
         268435456 |                                         0
         536870912 |                                         1
        1073741824 |                                         1
        2147483648 |                                         2
        4294967296 |                                         2
        8589934592 |                                         2
       17179869184 |                                         0
       34359738368 |                                         2
       68719476736 |                                         0


syslogd sleeping on syslogd:


           value  -------------- Distribution ------------ count
       17179869184 |                                         0
       34359738368 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 3
       68719476736 |                                         0


MozillaFirebird sleeping on MozillaFirebird:
```

```
           value  -------------- Distribution ------------ count
           65536 |                                         0
          131072 |                                         3
          262144 |@@                                       14
          524288 |                                         0
         1048576 |@@@                                      18
         2097152 |                                         0
         4194304 |                                         0
         8388608 |                                         1
        16777216 |                                         0
        33554432 |                                         1
        67108864 |                                         3
       134217728 |@                                        7
       268435456 |@@@@@@@@@@                               53
       536870912 |@@@@@@@@@@@@@@@                          78
      1073741824 |@@@@                                     25
      2147483648 |                                         0
      4294967296 |                                         0
      8589934592 |@                                        7
     17179869184 |                                         0
```

您可能需要了解 `MozillaFirebird` 如何以及为何阻塞自身。 可按以下示例中所示修改以上脚本来回答此问题 :

```
#pragma D option quiet
```

```
sched:::sleep
/execname == "MozillaFirebird" && curlwpsinfo->pr_stype == SOBJ_CV/
{
    bedtime[curlwpsinfo->pr_addr] = timestamp;
}

sched:::wakeup
/execname == "MozillaFirebird" && bedtime[args[0]->pr_addr]/
{
    @[args[1]->pr_pid, args[0]->pr_lwpid, pid, curlwpsinfo->pr_lwpid] =
        quantize(timestamp - bedtime[args[0]->pr_addr]);
    bedtime[args[0]->pr_addr] = 0;
}

sched:::wakeup
/bedtime[args[0]->pr_addr]/
{
    bedtime[args[0]->pr_addr] = 0;
}

END
{
    printa("%d/%d sleeping on %d/%d:\n%@d\n", @);
```

```
}
```

运行修改后的脚本若干秒将会生成与以下示例类似的输出：

```
# dtrace -s ./firebird.d
^C


 100459/1 sleeping on 100459/13:


           value  -------------- Distribution ------------ count
          262144 |                                         0
          524288 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
         1048576 |                                         0


100459/13 sleeping on 100459/1:


           value  -------------- Distribution ------------ count
        16777216 |                                         0
        33554432 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
        67108864 |                                         0


100459/1 sleeping on 100459/2:


           value  -------------- Distribution ------------ count
           16384 |                                         0
           32768 |@@@@                                     5
```

```
           65536 |@                                        2
          131072 |@@@@@                                    6
          262144 |                                         1
          524288 |@                                        2
         1048576 |                                         0
         2097152 |@@                                       3
         4194304 |@@@@                                     5
         8388608 |@@@@@@@@                                 9
        16777216 |@@@@@                                    6
        33554432 |@@                                       3
        67108864 |                                         0

100459/1 sleeping on 100459/5:

           value  -------------- Distribution ------------ count
           16384 |                                         0
           32768 |@@@@@                                    12
           65536 |@@                                       5
          131072 |@@@@@@                                   15
          262144 |                                         1
          524288 |                                         1
         1048576 |                                         2
         2097152 |@                                        4
         4194304 |@@@@@                                    13
```

```
          8388608 |@@@                                      8
         16777216 |@@@@@                                    13
         33554432 |@@                                       6
         67108864 |@@                                       5
        134217728 |@                                        4
        268435456 |                                         0
        536870912 |                                         1
       1073741824 |                                         0


100459/2 sleeping on 100459/1:


           value  -------------- Distribution ------------ count
           16384 |                                         0
           32768 |@@@@@@@@@@@@@@@                          11
           65536 |                                         0
          131072 |@@                                       2
          262144 |                                         0
          524288 |                                         0
         1048576 |@@@@                                     3
         2097152 |@                                        1
         4194304 |@@                                       2
         8388608 |@@                                       2
        16777216 |@                                        1
        33554432 |@@@@@@                                   5
```

```
         67108864 |                                         0
        134217728 |                                         0
        268435456 |                                         0
        536870912 |@                                        1
       1073741824 |@                                        1
       2147483648 |@                                        1
       4294967296 |                                         0


100459/5 sleeping on 100459/1:


           value  -------------- Distribution ------------ count
           16384 |                                         0
           32768 |                                         1
           65536 |                                         2
          131072 |                                         4
          262144 |                                         7
          524288 |                                         1
         1048576 |                                         5
         2097152 |                                         10
         4194304 |@@@@@@                                   77
         8388608 |@@@@@@@@@@@@@@@@@@@@@@@@                 270
        16777216 |@@@                                      43
        33554432 |@                                        20
        67108864 |@                                        14
```

```
       134217728 |                                         5
       268435456 |                                         2
       536870912 |                                         1
      1073741824 |                                         0
```

还可使用 sleep 和 wakeup 探测器来了解门服务器的性能（如名称服务高速缓存守护进程），如下例所示：

```
sched:::sleep
/curlwpsinfo->pr_stype == SOBJ_SHUTTLE/
{
    bedtime[curlwpsinfo->pr_addr] = timestamp;
}


sched:::wakeup
/execname == "nscd" && bedtime[args[0]->pr_addr]/
{
    @[stringof(curpsinfo->pr_fname), stringof(args[1]->pr_fname)] =
        quantize(timestamp - bedtime[args[0]->pr_addr]);
    bedtime[args[0]->pr_addr] = 0;
}


sched:::wakeup
/bedtime[args[0]->pr_addr]/
{
    bedtime[args[0]->pr_addr] = 0;
}
```

在大型邮件服务器上运行以上脚本生成的输出后半部分与以下示例类似：

```
imapd

           value  -------------- Distribution ------------ count
           16384 |                                         0
           32768 |                                         2
           65536 |@@@@@@@@@@@@@@@@@@@                      57
          131072 |@@@@@@@@@@@                              37
          262144 |                                         3
          524288 |@@@                                      11
         1048576 |@@@                                      10
         2097152 |@@                                       9
         4194304 |                                         1
         8388608 |                                         0


  mountd

           value  -------------- Distribution ------------ count
           65536 |                                         0
          131072 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  49
          262144 |@@@                                      6
          524288 |                                         1
         1048576 |                                         0
         2097152 |                                         0
         4194304 |@@@@                                     7
         8388608 |@                                        3
```

```
      16777216 |                                         0

sendmail

          value  -------------- Distribution ------------ count
          16384 |                                         0
          32768 |@                                        18
          65536 |@@@@@@@@@@@@@@@@@@                       205
         131072 |@@@@@@@@@@@@@                            154
         262144 |@                                        23
         524288 |                                         5
        1048576 |@@@@                                     50
        2097152 |                                         7
        4194304 |                                         5
        8388608 |                                         2
       16777216 |                                         0

automountd

          value  -------------- Distribution ------------ count
          32768 |                                         0
          65536 |@@@@@@@@@@                               22
         131072 |@@@@@@@@@@@@@@@@@@@@@@@@                 51
         262144 |@@                                       6
         524288 |                                         1
        1048576 |                                         0
```

```
         2097152 |                                       2
         4194304 |                                       2
         8388608 |                                       1
        16777216 |                                       1
        33554432 |                                       1
        67108864 |                                       0
       134217728 |                                       0
       268435456 |                                       1
       536870912 |                                       0
```

您可能会关心 automountd 的异常数据点或 sendmail 的超过 1 毫秒的持久数据点。可向以上脚本中添加其他谓词以有效查看造成任何异常或反常结果的原因。

preempt、remain-cpu

因为 Solaris 是抢占系统，所以高优先级线程将抢先于低优先级线程。抢占会导致低优先级线程出现明显延迟，所以您可能需要知道哪些线程被另一些线程抢占。以下示例说明如何使用 preempt 和 remain-cpu 探测器来显示此信息：

```
#pragma D option quiet



sched:::preempt

{

    self->preempt = 1;

}



sched:::remain-cpu

/self->preempt/

{
```

```
    self->preempt = 0;
}

sched:::off-cpu
/self->preempt/
{
    /*
     * If we were told to preempt ourselves, see who we ended up giving
     * the CPU to.
     */
    @[stringof(args[1]->pr_fname), args[0]->pr_pri, execname,
        curlwpsinfo->pr_pri] = count();
    self->preempt = 0;
}

END
{
    printf("%30s %3s %30s %3s %5s\n", "PREEMPTOR", "PRI",
        "PREEMPTED", "PRI", "#");
    printa("%30s %3d %30s %3d %5@d\n", @);
}
```

在桌面系统上运行以上脚本若干秒将会生成与以下示例类似的输出：

```
# dtrace -s ./whopreempt.d
^C
```

PREEMPTOR	PRI	PREEMPTED	PRI	#
sched	60	Xsun	53	1
xterm	59	Xsun	53	1
MozillaFirebird	57	Xsun	53	1
mpstat	100	fvwm2	59	1
sched	99	MozillaFirebird	57	1
sched	60	dtrace	30	1
mpstat	100	Xsun	59	2
sched	60	Xsun	54	2
sched	99	sched	60	2
fvwm2	59	Xsun	44	2
sched	99	Xsun	44	2
sched	60	xterm	59	2
sched	99	Xsun	53	2
sched	99	Xsun	54	3
sched	60	fvwm2	59	3
sched	60	Xsun	59	3
sched	99	Xsun	59	4
fvwm2	59	Xsun	54	8
fvwm2	59	Xsun	53	9
Xsun	59	MozillaFirebird	57	10
sched	60	MozillaFirebird	57	14
MozillaFirebird	57	Xsun	44	16
MozillaFirebird	57	Xsun	54	18

change-pri

抢占是根据优先级进行的，因此可能需要观察一段时间内优先级的变化。以下示例使用 change-pri 探测器来显示此信息：

```
sched:::change-pri

{

    @[stringof(args[0]->pr_clname)] =

        lquantize(args[2] - args[0]->pr_pri, -50, 50, 5);

}
```

示例脚本捕获优先级升高或降低的程度，并按调度类聚集。运行以上脚本将生成与以下示例类似的输出：

```
# dtrace -s ./pri.d

dtrace: script ’./pri.d’ matched 10 probes

^C

 IA

           value  -------------- Distribution ------------ count

           < -50 |                                         20

             -50 |@                                        38

             -45 |                                         4

             -40 |                                         13

             -35 |                                         12

             -30 |                                         18

             -25 |                                         18

             -20 |                                         23

             -15 |                                         6

             -10 |@@@@@@@@                                 201
```

```
          -5 |@@@@@@                                   160
           0 |@@@@@                                    138
           5 |@                                        47
          10 |@@                                       66
          15 |@                                        36
          20 |@                                        26
          25 |@                                        28
          30 |                                         18
          35 |                                         22
          40 |                                         8
          45 |                                         11
        >= 50 |@                                        34


  TS

          value  -------------- Distribution ------------ count
            -15 |                                       0
            -10 |@                                      1
             -5 |@@@@@@@@@@@@@                          7
              0 |@@@@@@@@@@@@@@@@@@@@@@                 12
              5 |                                       0
             10 |@@@@@                                  3
             15 |                                       0
```

输出显示交互 (Interactive, IA) 调度类的优先级处理。您可能不需要查看优先级处理，而只需要查看一段时间内特定进程和线程的优先级值。以下脚本使用 `change-pri` 探测器来显示此信息：

```
#pragma D option quiet


BEGIN
{
    start = timestamp;
}


sched:::change-pri
/args[1]->pr_pid == $1 && args[0]->pr_lwpid == $2/
{
    printf("%d %d\n", timestamp - start, args[2]);
}


tick-1sec
/++n == 5/
{
    exit(0);
}
```

要查看一段时间内优先级的变化，请在一个窗口中键入以下命令：

```
$ echo $$
139208
$ while true ; do let i=0 ; done
```

在另一个窗口中运行该脚本并将输出重定向到文件：

```
# dtrace -s ./pritime.d 139208 1 > /tmp/pritime.out

#
```

可将上面生成的文件 `/tmp/pritime.out` 用作绘图软件的输入，以图形方式显示一段时间内的优先级。`gnuplot` 是免费提供的绘图软件包，它包括在 Solaris 免费软件配套 CD 中。缺省情况下，`gnuplot` 安装在 `/opt/sfw/bin` 中。

tick

Solaris 使用基于计时单元的 *CPU* 记帐（其中系统时钟中断会按固定时间间隔触发）并将 CPU 的使用情况归结到计时单元时间运行的线程和进程。以下示例说明如何使用 `tick` 探测器来观察这一归结情况：

```
# dtrace -n sched:::tick'{@[stringof(args[1]->pr_fname)] = count()}'

^C

  arch                                                              1
  sh                                                                1
  sed                                                               1
  echo                                                              1
  ls                                                                1
  FvwmAuto                                                          1
  pwd                                                               1
  awk                                                               2
  basename                                                          2
  expr                                                              2
  resize                                                            2
  tput                                                              2
  uname                                                             2
  fsflush                                                           2
```

```
  dirname                                                   4

  vim                                                       9

  fvwm2                                                    10

  ksh                                                      19

  xterm                                                    21

  Xsun                                                     93

  MozillaFirebird                                         260
```

系统时钟频率随操作系统不同而变化，但通常会在 25 赫兹到 1024 赫兹之间。Solaris 系统时钟频率是可调整的，但缺省值为 100 赫兹。

仅当系统时钟检测到可运行线程时，`tick` 探测器才会触发。要使用 `tick` 探测器来观察系统时钟的频率，必须具有始终可运行的线程。在一个窗口中创建循环 shell，如下例所示：

```
$ while true ; do let i=0 ; done
```

在另一个窗口中运行以下脚本：

```
uint64_t last[int];



sched:::tick

/last[cpu]/

{

    @[cpu] = min(timestamp - last[cpu]);

}



sched:::tick

{

    last[cpu] = timestamp;

}
```

```
# dtrace -s ./ticktime.d

dtrace: script ’./ticktime.d’ matched 2 probes

^C



  0          9883789
```

最短时间间隔为 9.8 毫秒，它指示缺省时钟周期频率为 10 毫秒（100 赫兹）。因为抖动，所以观察到的最短时间间隔略小于 10 毫秒。

基于计时单元的记帐的一个缺点是，执行记帐的系统时钟通常还负责分派任何与时间有关的调度活动。因此，如果线程要在每个时钟周期（即每 10 毫秒）执行一定量的工作，则根据记帐是在与时间有关的分派调度活动之前还是之后完成，系统会对线程的工作量记帐过多或过少。在 Solaris 中，记帐是在与时间有关的分派之前完成的。因此，系统会对定期运行的线程工作量记帐过少。如果这类线程运行的时间小于时钟周期的时间间隔，它们可以有效地“隐藏”在时钟周期中。 以下示例显示系统中这类线程的情况：

```
sched:::tick,

sched:::enqueue

{

    @[probename] = lquantize((timestamp / 1000000) % 10, 0, 10);

}
```

示例脚本的输出在 10 毫秒时间间隔内分为两个毫秒偏移部分，一个对应于 `tick` 探测器，另一个对应于 `enqueue`：

```
# dtrace -s ./tick.d

dtrace: script ’./tick.d’ matched 4 probes

^C

  tick

           value  -------------- Distribution ------------ count

               6 |                                         0

               7 |@                                        3
```

```
           8 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@   79
           9 |                                          0

  enqueue
         value  ------------- Distribution ------------ count
           < 0 |                                         0
             0 |@@                                       267
             1 |@@                                       300
             2 |@@                                       259
             3 |@@                                       291
             4 |@@@                                      360
             5 |@@                                       305
             6 |@@                                       295
             7 |@@@@                                     522
             8 |@@@@@@@@@@@@                             1315
             9 |@@@                                      337
```

名为 tick 的输出直方图显示时钟周期在 8 毫秒偏移处触发。 如果调度与时钟周期完全无关，enqueue 的输出将平均分布在 10 毫秒时间间隔内。 但是，输出显示同一 8 毫秒偏移处有一个峰值，指示系统中至少有一些线程是根据时间调度的。

稳定性

sched 提供器使用 DTrace 的稳定性机制说明其稳定性，如下表中所示。有关稳定性机制的更多信息，请参见第 39 章。

元素	名称稳定性	数据稳定性	相关性类
提供器	发展中	发展中	ISA

元素	名称稳定性	数据稳定性	相关性类
模块	专用	专用	未知
函数	专用	专用	未知
名称	发展中	发展中	ISA
参数	发展中	发展中	ISA

◆ ◆ ◆ 第 27 章

27 io 提供器

io 提供器提供了与磁盘输入和输出有关的探测器。io 提供器允许快速查看通过 I/O 监视工具（如 iostat(1M)）观察到的行为。例如，通过使用 io 提供器，可以按设备、I/O 类型、I/O 大小、进程、应用程序名称、文件名或文件偏移来了解 I/O。

探测器

表 27–1 中介绍了 io 探测器。

表 27–1 io 探测器

探测器	说明
start	向外围设备或 NFS 服务器发出 I/O 请求时将触发的探测器。args[0] 指向对应于 I/O 请求的 bufinfo_t。 args[1] 指向正在对其发出 I/O 请求的设备的 devinfo_t。 args[2] 指向对应于 I/O 请求的文件的 fileinfo_t。请注意，文件信息可用性取决于发出 I/O 请求的文件系统。有关更多信息，请参见第 359 页中的 “fileinfo_t” 。
done	完成 I/O 请求后将触发的探测器。args[0] 指向对应于 I/O 请求的 bufinfo_t。done 探测器在 I/O 请求完成之后、对缓冲区执行的处理完成之前触发。因此，在触发 done 探测器时，不会在 b_flags 中设置 B_DONE。args[1] 指向已对其发出 I/O 请求的设备的 devinfo_t。args[2] 指向对应于 I/O 请求的文件的 fileinfo_t。
wait-start	探测器就在线程开始等待暂挂的指定 I/O 请求完成之前触发。args[0] 指向对应于线程将等待的 I/O 请求的 buf(9S) 结构。 args[1] 指向已对其发出 I/O 请求的设备的 devinfo_t。args[2] 指向对应于 I/O 请求的文件的 fileinfo_t。在触发 wait-start 探测器之后的某个时间，wait-done 探测器将在同一线程中触发。

表 27–1 io 探测器 （续）

探测器	说明
wait-done	线程等待指定 I/O 请求完成这一过程结束后将触发的探测器。args[0] 指向对应于线程将等待的 I/O 请求的 bufinfo_t。 args[1] 指向已对其发出 I/O 请求的设备的 devinfo_t。 args[2] 指向对应于 I/O 请求的文件的 fileinfo_t。 wait-done 探测器仅在同一线程中的 wait-done 探测器触发之后触发。

请注意，针对外围设备的所有 I/O 请求以及针对 NSF 服务器的所有文件读/写请求都会触发 io 探测器。例如，由于 readdir(3C) 请求，NFS 服务器中对元数据的请求不会触发 io 探测器。

参数

表 27–2 中列出了 io 探测器的参数类型。表 27–1 中介绍了这些参数。

表 27–2 io 探测器参数

探测器	args[0]	args[1]	args[2]
start	struct buf *	devinfo_t *	fileinfo_t *
done	struct buf *	devinfo_t *	fileinfo_t *
wait-start	struct buf *	devinfo_t *	fileinfo_t *
wait-done	struct buf *	devinfo_t *	fileinfo_t *

每个 io 探测器的参数由指向 buf(9S) 结构的指针、指向 devinfo_t 的指针和指向 fileinfo_t 的指针组成。本节将详细介绍这些结构。

bufinfo_t 结构

bufinfo_t 结构简要介绍了 I/O 请求。start、done、wait-start 和 wait-done 探测器中的 args[0] 指向对应于 I/O 请求的缓冲区。bufinfo_t 结构定义如下所示：

```
typedef struct bufinfo {

    int b_flags;                    /* flags */

    size_t b_bcount;                /* number of bytes */

    caddr_t b_addr;                 /* buffer address */
```

```
    uint64_t b_blkno;               /* expanded block # on device */

    uint64_t b_lblkno;              /* block # on device */

    size_t b_resid;                 /* # of bytes not transferred */

    size_t b_bufsize;               /* size of allocated buffer */

    caddr_t b_iodone;               /* I/O completion routine */

    dev_t b_edev;                   /* extended device */

} bufinfo_t;
```

b_flags 成员指示 I/O 缓冲区的状态，它由不同状态值的按位或的结果组成。 表 27–3 中介绍了有效的状态值。

表 27–3 b_flags 值

B_DONE	指示数据传送已完成。
B_ERROR	指示 I/O 传送错误。它与 b_error 字段一起设置。
B_PAGEIO	指示分页的 I/O 请求中正在使用该缓冲区。 有关更多信息，请参见 b_addr 字段的说明。
B_PHYS	指示正在对指向用户数据区的物理（直接）I/O 使用该缓冲区。
B_READ	指示将从外围设备中读取数据到主内存中。
B_WRITE	指示将数据从主内存传送到外围设备。
B_ASYNC	I/O 请求为异步请求，不会被等待。 wait-start 和 wait-done 探测器不会对异步 I/O 请求触发。 请注意，定向为异步的某些 I/O 可能不会设置 B_ASYNC：异步 I/O 子系统可能通过使单独的工作线程执行同步 I/O 操作来实现异步请求。

b_bcount 字段表示要作为 I/O 请求一部分进行传送的字节数。

除非设置了 B_PAGEIO，否则 b_addr 字段表示 I/O 请求的虚拟地址。除非设置了 B_PHYS（此情况下该地址为用户虚拟地址），否则该地址为内核虚拟地址。如果设置了 B_PAGEIO，则 b_addr 字段将包含内核专用数据。仅能设置 B_PHYS 和 B_PAGEIO 的其中之一，否则将不会设置任何标志。

b_lblkno 字段标识设备上要访问的逻辑块。 逻辑块到物理块（如柱面、磁轨等）的映射由设备定义。

b_resid 字段设置为由于发生错误而未传送的字节数。

b_bufsize 字段包含分配的缓冲区大小。

b_iodone 字段标识 I/O 完成时内核中调用的特定例程。

b_error 字段可能包含发生 I/O 错误时从驱动程序返回的错误代码。b_error 将与在 b_flags 成员中设置的 B_ERROR 位一起设置。

b_edev 字段包含所访问设备的主要和次要设备编号。使用者可使用 D 子例程 getmajor() 和 getminor() 从 b_edev 字段中提取主要和次要设备编号。

devinfo_t

devinfo_t 结构提供有关设备的信息。start、done、wait-start 和 wait-done 探测器中的 args[1] 指向对应于 I/O 的目标设备的 devinfo_t 结构。devinfo_t 的成员如下所示：

```
typedef struct devinfo {

    int dev_major;                  /* major number */

    int dev_minor;                  /* minor number */

    int dev_instance;               /* instance number */

    string dev_name;                /* name of device */

    string dev_statname;            /* name of device + instance/minor */

    string dev_pathname;            /* pathname of device */

} devinfo_t;
```

dev_major 字段是设备的主要编号。有关更多信息，请参见 getmajor(9F)。

dev_minor 字段是设备的次要编号。有关更多信息，请参见 getminor(9F)。

dev_instance 字段是设备的实例编号。 设备实例不同于次要编号。次要编号是由设备驱动程序管理的抽象对象。实例编号是设备节点的一个属性。 可使用 prtconf(1M) 来显示设备节点实例编号。

dev_name 字段是用于管理设备的设备驱动程序的名称。 可使用 prtconf(1M) 的 -D 选项显示设备驱动程序名称。

dev_statname 字段是 iostat(1M) 报告的设备名称。此名称同时对应于 kstat(1M) 报告的内核统计信息的名称。 提供此字段是为了便于快速地将异常的 iostat 或 kstat 输出与实际 I/O 活动相关联。

dev_pathname 字段是设备的完整路径。此路径可指定为 prtconf(1M) 的参数，以获取详细的设备信息。dev_pathname 指定的路径包括用于表示设备节点、实例编号和次要节点的组件。 但是，不必在统计信息名称中表示所有这三个元素。对于某些设备，统计信息名称由

设备名称和实例编号组成。对于其他设备，统计信息名称则由设备名称和次要节点编号组成。 因此，具有相同 dev_statname 的两个设备的 dev_pathname 可能会不同。

fileinfo_t

fileinfo_t 结构提供有关文件的信息。start、done、wait-start 和 wait-done 探测器中的 args[2] 指向 I/O 所对应的文件。 在分发 I/O 请求时，文件信息存在与否将取决于提供此信息的文件系统。某些文件系统（尤其是第三方文件系统）可能不会提供此信息。 I/O 请求也可能是从不包含任何文件信息的文件系统发出的。 例如，对文件系统元数据的任何 I/O 请求，将不会与任何一个文件关联。最后，一些高度优化的文件系统可能将不相交文件中的 I/O 聚集为单个 I/O 请求。在此情况下，文件系统可能会为表示大多数 I/O 的文件或表示部分 I/O 的文件提供文件信息。或者，在此情况下文件系统可能根本不提供任何文件信息。

fileinfo_t 结构的定义如下所示：

```
typedef struct fileinfo {

    string fi_name;                 /* name (basename of fi_pathname) */

    string fi_dirname;              /* directory (dirname of fi_pathname) */

    string fi_pathname;             /* full pathname */

    offset_t fi_offset;             /* offset within file */

    string fi_fs;                   /* filesystem */

    string fi_mount;                /* mount point of file system */

} fileinfo_t;
```

fi_name 字段包含文件的名称，但不包括任何目录部分。 如果没有任何文件信息与 I/O 关联，则 fi_name 字段将设置为字符串 <none>。在极少数情况下，与文件关联的路径名可能未知。在此情况下，fi_name 字段将设置为字符串 <unknown>。

fi_dirname 字段仅包含文件名的目录部分。与 fi_name 一样，如果不存在任何文件信息，则此字符串可能会设置为 <none>；如果与该文件关联的路径名未知，则此字符串可能会设置为 <unknown>。

fi_pathname 字段包含文件的完整路径名。与 fi_name 一样，如果不存在任何文件信息，此字符串可能会设置为 <none>；如果与该文件关联的路径名未知，此字符串可能会设置为 <unknown>。

fi_offset 字段包含文件中的偏移，如果文件信息不存在或者文件系统未指定偏移，则此字段将为 1。

示例

以下示例脚本显示了发出每个 I/O 时的相关信息：

```
#pragma D option quiet


BEGIN
{
    printf("%10s %58s %2s\n", "DEVICE", "FILE", "RW");
}


io:::start
{
    printf("%10s %58s %2s\n", args[1]->dev_statname,
        args[2]->fi_pathname, args[0]->b_flags & B_READ ? "R" : "W");
}
```

在 x86 膝上型计算机系统上冷启动 Acrobat Reader 时的示例输出与以下示例类似：

```
# dtrace -s ./iosnoop.d
    DEVICE                                                  FILE RW
     cmdk0                                /opt/Acrobat4/bin/acroread  R
     cmdk0                                /opt/Acrobat4/bin/acroread  R
     cmdk0                                                <unknown>  R
     cmdk0                          /opt/Acrobat4/Reader/AcroVersion  R
     cmdk0                                                <unknown>  R
     cmdk0                                                <unknown>  R
     cmdk0                                                   <none>  R
```

```
cmdk0                                                  <unknown>  R
cmdk0                                                     <none>  R
cmdk0                 /usr/lib/locale/iso_8859_1/iso_8859_1.so.3  R
cmdk0                 /usr/lib/locale/iso_8859_1/iso_8859_1.so.3  R
cmdk0                 /usr/lib/locale/iso_8859_1/iso_8859_1.so.3  R
cmdk0                                                     <none>  R
cmdk0                                                  <unknown>  R
cmdk0                                                  <unknown>  R
cmdk0                                                  <unknown>  R
cmdk0             /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0             /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0                                                     <none>  R
cmdk0             /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0             /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0             /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0             /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0             /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0             /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0             /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0             /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0                                                  <unknown>  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0                                                     <none>  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
```

```
    cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R

    cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R

    cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R

    cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R

    cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R

    cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R

    cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R

    cmdk0             /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R

    cmdk0             /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R

    cmdk0                                                  <unknown>  R

    cmdk0        /opt/Acrobat4/Reader/intelsolaris/lib/libAGM.so.3.0  R

    cmdk0                                                     <none>  R

    cmdk0        /opt/Acrobat4/Reader/intelsolaris/lib/libAGM.so.3.0  R

    cmdk0        /opt/Acrobat4/Reader/intelsolaris/lib/libAGM.so.3.0  R

      ...
```

输出中的 `<none>` 条目指示 I/O 与任何特定文件中的数据都不对应：这些 I/O 是由于元数据的形式不同造成的。输出中的 `<unknown>` 条目指示该文件的路径名未知。此情况相对较少。

可通过使用关联数组跟踪每个 I/O 所花费的时间来使示例脚本更加完善，如下例所示：

```
#pragma D option quiet



BEGIN

{

    printf("%10s %58s %2s %7s\n", "DEVICE", "FILE", "RW", "MS");

}
```

```
io:::start

{

    start[args[0]->b_edev, args[0]->b_blkno] = timestamp;

}



io:::done

/start[args[0]->b_edev, args[0]->b_blkno]/

{

    this->elapsed = timestamp - start[args[0]->b_edev, args[0]->b_blkno];

    printf("%10s %58s %2s %3d.%03d\n", args[1]->dev_statname,

        args[2]->fi_pathname, args[0]->b_flags & B_READ ? "R" : "W",

        this->elapsed / 10000000, (this->elapsed / 1000) % 1000);

    start[args[0]->b_edev, args[0]->b_blkno] = 0;

}
```

在将 USB 存储设备热插拔到空闲的 x86 膝上型计算机系统中时，上述示例的输出将如下例所示：

```
# dtrace -s ./iotime.d

    DEVICE                                                  FILE RW      MS

     cmdk0                                  /kernel/drv/scsa2usb  R  24.781

     cmdk0                                  /kernel/drv/scsa2usb  R  25.208

     cmdk0                                     /var/adm/messages  W  25.981

     cmdk0                                  /kernel/drv/scsa2usb  R   5.448

     cmdk0                                                <none>  W   4.172
```

```
cmdk0                                /kernel/drv/scsa2usb  R   2.620
cmdk0                                   /var/adm/messages  W   0.252
cmdk0                                           <unknown>  R   3.213
cmdk0                                              <none>  W   3.011
cmdk0                                           <unknown>  R   2.197
cmdk0                                   /var/adm/messages  W   2.680
cmdk0                                              <none>  W   0.436
cmdk0                                   /var/adm/messages  W   0.542
cmdk0                                              <none>  W   0.339
cmdk0                                   /var/adm/messages  W   0.414
cmdk0                                              <none>  W   0.344
cmdk0                                   /var/adm/messages  W   0.361
cmdk0                                              <none>  W   0.315
cmdk0                                   /var/adm/messages  W   0.421
cmdk0                                              <none>  W   0.349
cmdk0                                              <none>  R   1.524
cmdk0                                           <unknown>  R   3.648
cmdk0                               /usr/lib/librcm.so.1  R   2.553
cmdk0                               /usr/lib/librcm.so.1  R   1.332
cmdk0                               /usr/lib/librcm.so.1  R   0.222
cmdk0                               /usr/lib/librcm.so.1  R   0.228
cmdk0                               /usr/lib/librcm.so.1  R   0.927
cmdk0                                              <none>  R   1.189
  ...
```

```
cmdk0                              /usr/lib/devfsadm/linkmod  R   1.110
cmdk0          /usr/lib/devfsadm/linkmod/SUNW_audio_link.so  R   1.763
cmdk0          /usr/lib/devfsadm/linkmod/SUNW_audio_link.so  R   0.161
cmdk0            /usr/lib/devfsadm/linkmod/SUNW_cfg_link.so  R   0.819
cmdk0            /usr/lib/devfsadm/linkmod/SUNW_cfg_link.so  R   0.168
cmdk0           /usr/lib/devfsadm/linkmod/SUNW_disk_link.so  R   0.886
cmdk0           /usr/lib/devfsadm/linkmod/SUNW_disk_link.so  R   0.185
cmdk0         /usr/lib/devfsadm/linkmod/SUNW_fssnap_link.so  R   0.778
cmdk0         /usr/lib/devfsadm/linkmod/SUNW_fssnap_link.so  R   0.166
cmdk0           /usr/lib/devfsadm/linkmod/SUNW_lofi_link.so  R   1.634
cmdk0           /usr/lib/devfsadm/linkmod/SUNW_lofi_link.so  R   0.163
cmdk0             /usr/lib/devfsadm/linkmod/SUNW_md_link.so  R   0.477
cmdk0             /usr/lib/devfsadm/linkmod/SUNW_md_link.so  R   0.161
cmdk0           /usr/lib/devfsadm/linkmod/SUNW_misc_link.so  R   0.198
cmdk0           /usr/lib/devfsadm/linkmod/SUNW_misc_link.so  R   0.168
cmdk0           /usr/lib/devfsadm/linkmod/SUNW_misc_link.so  R   0.247
cmdk0      /usr/lib/devfsadm/linkmod/SUNW_misc_link_i386.so  R   1.735
  ...
```

可以根据此输出从多方面观察该系统的结构。首先，应注意到前几个 I/O 操作的执行时间很长，每个操作所花费的时间大概是 25 毫秒。 此执行时间可能是由于膝上型计算机上的 `cmdk0` 设备的电源管理造成的。其次，应观察到由于要处理 USB 海量存储设备而装入 `scsa2usb`(7D) 驱动程序时产生的 I/O。第三，应注意到在报告设备时将写入 `/var/adm/messages`。最后，应观察到对设备链接生成器（文件名以 `link.so` 结尾）的读取，它们可能用于处理新设备。

通过 `io` 提供器可以深入了解 `iostat`(1M) 输出。假定您观察到类似以下示例的 `iostat` 输出：

```
extended device statistics
```

```
device       r/s    w/s   kr/s   kw/s wait actv  svc_t  %w  %b
cmdk0        8.0    0.0  399.8    0.0  0.0  0.0    0.8   0   1
sd0          0.0    0.0    0.0    0.0  0.0  0.0    0.0   0   0
sd2          0.0  109.0    0.0  435.9  0.0  1.0    8.9   0  97
nfs1         0.0    0.0    0.0    0.0  0.0  0.0    0.0   0   0
nfs2         0.0    0.0    0.0    0.0  0.0  0.0    0.0   0   0
```

可使用 `iotime.d` 脚本来查看所发生的 I/O，如下例所示：

```
    DEVICE                                              FILE RW      MS
       sd2                                  /mnt/archives.tar  W   0.856
       sd2                                  /mnt/archives.tar  W   0.729
       sd2                                  /mnt/archives.tar  W   0.890
       sd2                                  /mnt/archives.tar  W   0.759
       sd2                                  /mnt/archives.tar  W   0.884
       sd2                                  /mnt/archives.tar  W   0.746
       sd2                                  /mnt/archives.tar  W   0.891
       sd2                                  /mnt/archives.tar  W   0.760
       sd2                                  /mnt/archives.tar  W   0.889
     cmdk0                     /export/archives/archives.tar  R   0.827
       sd2                                  /mnt/archives.tar  W   0.537
       sd2                                  /mnt/archives.tar  W   0.887
       sd2                                  /mnt/archives.tar  W   0.763
       sd2                                  /mnt/archives.tar  W   0.878
       sd2                                  /mnt/archives.tar  W   0.751
       sd2                                  /mnt/archives.tar  W   0.884
```

```
  sd2                                 /mnt/archives.tar  W   0.760
  sd2                                 /mnt/archives.tar  W   3.994
  sd2                                 /mnt/archives.tar  W   0.653
  sd2                                 /mnt/archives.tar  W   0.896
  sd2                                 /mnt/archives.tar  W   0.975
  sd2                                 /mnt/archives.tar  W   1.405
  sd2                                 /mnt/archives.tar  W   0.724
  sd2                                 /mnt/archives.tar  W   1.841
cmdk0                  /export/archives/archives.tar  R   0.549
  sd2                                 /mnt/archives.tar  W   0.543
  sd2                                 /mnt/archives.tar  W   0.863
  sd2                                 /mnt/archives.tar  W   0.734
  sd2                                 /mnt/archives.tar  W   0.859
  sd2                                 /mnt/archives.tar  W   0.754
  sd2                                 /mnt/archives.tar  W   0.914
  sd2                                 /mnt/archives.tar  W   0.751
  sd2                                 /mnt/archives.tar  W   0.902
  sd2                                 /mnt/archives.tar  W   0.735
  sd2                                 /mnt/archives.tar  W   0.908
  sd2                                 /mnt/archives.tar  W   0.753
```

此输出似乎说明正在从 `cmdk0`（在 `/export/archives` 中）读取文件 `archives.tar`，且正在将该文件写入到设备 `sd2`（在 `/mnt` 中）。这种并行地对名为 `archives.tar` 的两个文件进行独立处理的情况似乎不可能存在。为了进一步地深入了解，可对设备、应用程序、进程 ID 和所传送的字节数进行聚集，如下例所示：

```
#pragma D option quiet
```

```
io:::start

{

    @[args[1]->dev_statname, execname, pid] = sum(args[0]->b_bcount);

}



END

{

    printf("%10s %20s %10s %15s\n", "DEVICE", "APP", "PID", "BYTES");

    printa("%10s %20s %10d %15@d\n", @);

}
```

运行此脚本几秒钟后将生成与以下示例类似的输出：

```
# dtrace -s ./whoio.d

^C

    DEVICE                  APP        PID           BYTES

     cmdk0                   cp        790         1515520

       sd2                   cp        790         1527808
```

此输出说明此活动是将文件 `archives.tar` 从一个设备复制到另一个设备。此结论会很自然地引出另一个问题：这些设备中是否有一个设备的运行速度比另一个设备的运行速度更快？哪个设备限制了复制操作？要回答这些问题，您需要知道每个设备的有效吞吐量而不是每个设备每秒传送的字节数。可以使用以下示例脚本来确定吞吐量：

```
#pragma D option quiet



io:::start

{

    start[args[0]->b_edev, args[0]->b_blkno] = timestamp;
```

```
}


io:::done
/start[args[0]->b_edev, args[0]->b_blkno]/
{
    /*
     * We want to get an idea of our throughput to this device in KB/sec.
     * What we have, however, is nanoseconds and bytes.  That is we want
     * to calculate:
     *
     *                        bytes / 1024
     *                  ------------------------
     *                  nanoseconds / 1000000000
     *
     * But we can't calculate this using integer arithmetic without losing
     * precision (the denomenator, for one, is between 0 and 1 for nearly
     * all I/Os).  So we restate the fraction, and cancel:
     *
     *     bytes      1000000000         bytes        976562
     *   --------- * -------------  =  --------- * -------------
     *      1024      nanoseconds          1        nanoseconds
     *
     * This is easy to calculate using integer arithmetic; this is what
     * we do below.
```

```
     */
    this->elapsed = timestamp - start[args[0]->b_edev, args[0]->b_blkno];
    @[args[1]->dev_statname, args[1]->dev_pathname] =
        quantize((args[0]->b_bcount * 976562) / this->elapsed);
    start[args[0]->b_edev, args[0]->b_blkno] = 0;
}

END
{
    printa("  %s (%s)\n%@d\n", @);
}
```

运行示例脚本几秒钟后会产生以下输出：

```
  sd2 (/devices/pci@0,0/pci1179,1@1d/storage@2/disk@0,0:r)

           value  ------------- Distribution ------------- count
              32 |                                         0
              64 |                                         3
             128 |                                         1
             256 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  2257
             512 |                                         1
            1024 |                                         0

  cmdk0 (/devices/pci@0,0/pci-ide@1f,1/ide@0/cmdk@0,0:a)
```

```
           value  ------------- Distribution ------------- count
             128 |                                         0
             256 |                                         1
             512 |                                         0
            1024 |                                         2
            2048 |                                         0
            4096 |                                         2
            8192 |@@@@@@@@@@@@@@@@@@@                      172
           16384 |@@@@@                                    52
           32768 |@@@@@@@@@@@                              108
           65536 |@@@                                      34
          131072 |                                         0
```

该输出说明 sd2 很明显是限制设备。sd2 的吞吐量介于 256K/秒和 512K/秒之间，而cmdk0 在任何位置传送 I/O 的速率都是 8 MB/秒到超过 64 MB/秒。 该脚本列显了设备的名称（可在 iostat 中看到）和完整路径。要了解有关设备的更多信息，可向 prtconf 指定设备路径，如下例所示：

```
# prtconf -v /devices/pci@0,0/pci1179,1@1d/storage@2/disk@0,0

disk, instance #2 (driver name: sd)

    Driver properties:

        name='lba-access-ok' type=boolean dev=(29,128)

        name='removable-media' type=boolean dev=none

        name='pm-components' type=string items=3 dev=none

            value='NAME=spindle-motor' + '0=off' + '1=on'

        name='pm-hardware-state' type=string items=1 dev=none

            value='needs-suspend-resume'
```

```
        name='ddi-failfast-supported' type=boolean dev=none

        name='ddi-kernel-ioctl' type=boolean dev=none

    Hardware properties:

        name='inquiry-revision-id' type=string items=1

            value='1.04'

        name='inquiry-product-id' type=string items=1

            value='STORAGE DEVICE'

        name='inquiry-vendor-id' type=string items=1

            value='Generic'

        name='inquiry-device-type' type=int items=1

            value=00000000

        name='usb' type=boolean

        name='compatible' type=string items=1

            value='sd'

        name='lun' type=int items=1

            value=00000000

        name='target' type=int items=1

            value=00000000
```

如强调项中所指示的那样，此设备是可移除的 USB 存储设备。

本节中的示例已经研究了所有 I/O 请求。 但是，您可能只关注一种请求类型。 以下示例将跟踪要进行写操作的目录以及执行写操作的应用程序：

```
#pragma D option quiet


io:::start
```

```
/args[0]->b_flags & B_WRITE/

{

    @[execname, args[2]->fi_dirname] = count();

}


END

{

    printf("%20s %51s %5s\n", "WHO", "WHERE", "COUNT");

    printa("%20s %51s %5@d\n", @);

}
```

在台式机上运行此示例脚本一段时间会生成一些值得关注的结果，如以下示例输出所示：

```
# dtrace -s ./whowrite.d

^C

               WHO                                               WHERE COUNT

                su                                           /var/adm     1

           fsflush                                               /etc     1

           fsflush                                                  /     1

           fsflush                                           /var/log     1

           fsflush                                    /export/bmc/lisa     1

               esd   /export/bmc/.phoenix/default/78cxczuy.slt/Cache     1

           fsflush                                /export/bmc/.phoenix     1

               esd         /export/bmc/.phoenix/default/78cxczuy.slt     1

                vi                                           /var/tmp     2

                vi                                               /etc     2
```

```
            cat                                              <none>     2
           bash                                                   /     2
             vi                                              <none>     3
           xterm                                           /var/adm     3
         fsflush                                         /export/bmc     7
  MozillaFirebird                                          <none>     8
            vim                                         /export/bmc     9
  MozillaFirebird                                       /export/bmc    10
         fsflush                                          /var/adm    11
        devfsadm                                              /dev    14
             ksh                                              <none>    71
             ksh                                         /export/bmc    71
         fsflush         /export/bmc/.phoenix/default/78cxczuy.slt   119
  MozillaFirebird         /export/bmc/.phoenix/default/78cxczuy.slt   119
         fsflush                                              <none>   211
  MozillaFirebird   /export/bmc/.phoenix/default/78cxczuy.slt/Cache   591
         fsflush   /export/bmc/.phoenix/default/78cxczuy.slt/Cache   666
           sched                                              <none>  2385
```

如以上输出所示，所有写操作实际上都与 Mozilla Firebird 高速缓存关联。 标有 <none> 的写操作可能是由文件系统中的其他与 UFS 日志关联的写操作而引发的。 有关日志记录的详细信息，请参见 ufs(7FS)。此示例说明如何使用 io 提供器来发现更高层次的软件问题。 在此情况下，脚本显示存在配置问题： 如果 Web 浏览器的高速缓存位于 tmpfs(7FS) 文件系统的某个目录中，则它引起的 I/O 会少得多（并且很可能根本没有任何 I/O）。

前面的示例中仅使用了 start 和 done 探测器。 可使用 wait-start 和 wait-done 探测器来了解为什么应用程序会因为 I/O 阻塞以及阻塞多长时间。以下示例脚本同时使用 io 探测器和 sched 探测器（请参见第 26 章）来获取 CPU 时间（相对于 StarSuite 软件的 I/O 等待时间）：

```
#pragma D option quiet


sched:::on-cpu
/execname == "soffice.bin"/
{
    self->on = vtimestamp;
}


sched:::off-cpu
/self->on/
{
    @time["<on cpu>"] = sum(vtimestamp - self->on);
    self->on = 0;
}


io:::wait-start
/execname == "soffice.bin"/
{
    self->wait = timestamp;
}


io:::wait-done
/self->wait/
{
```

```
    @io[args[2]->fi_name] = sum(timestamp - self->wait);
    @time["<I/O wait>"] = sum(timestamp - self->wait);
    self->wait = 0;
}

END
{
    printf("Time breakdown (milliseconds):\n");
    normalize(@time, 1000000);
    printa("  %-50s %15@d\n", @time);

    printf("\nI/O wait breakdown (milliseconds):\n");
    normalize(@io, 1000000);
    printa("  %-50s %15@d\n", @io);
}
```

在 StarSuite 软件冷启动期间运行示例脚本将产生以下输出：

```
Time breakdown (milliseconds):
  <on cpu>                                                3634
  <I/O wait>                                             13114

I/O wait breakdown (milliseconds):
  soffice.tmp                                                0
  Office                                                     0
  unorc                                                      0
```

```
sbasic.cfg                                                    0
en                                                            0
smath.cfg                                                     0
toolboxlayout.xml                                             0
sdraw.cfg                                                     0
swriter.cfg                                                   0
Linguistic.dat                                                0
scalc.cfg                                                     0
Views.dat                                                     0
Store.dat                                                     0
META-INF                                                      0
Common.xml.tmp                                                0
afm                                                           0
libsimreg.so                                                  1
xiiimp.so.2                                                   3
outline                                                       4
Inet.dat                                                      6
fontmetric                                                    6
...
libucb1.so                                                   44
libj641si_g.so                                               46
libX11.so.4                                                  46
liblng641si.so                                               48
swriter.db                                                   53
```

```
libwrp641si.so                                  53
liblocaledata_ascii.so                          56
libi18npool641si.so                             65
libdbtools2.so                                  69
ofa64101.res                                    74
libxcr641si.so                                  82
libucpchelp1.so                                 83
libsot641si.so                                  86
libcppuhelper3C52.so                            98
libfwl641si.so                                 100
libsb641si.so                                  104
libcomphelp2.so                                105
libxo641si.so                                  106
libucpfile1.so                                 110
libcppu.so.3                                   111
sw64101.res                                    114
libdb-3.2.so                                   119
libtk641si.so                                  126
libdtransX11641si.so                           127
libgo641si.so                                  132
libfwe641si.so                                 150
libi18n641si.so                                152
libfwi641si.so                                 154
libso641si.so                                  173
```

```
  libpsp641si.so                                                186

  libtl641si.so                                                 189

  <unknown>                                                     189

  libucbhelper1C52.so                                           195

  libutl641si.so                                                213

  libofa641si.so                                                216

  libfwk641si.so                                                229

  libsvl641si.so                                                261

  libcfgmgr2.so                                                 368

  libsvt641si.so                                                373

  libvcl641si.so                                                741

  libsvx641si.so                                                885

  libsfx641si.so                                                993

  <none>                                                       1096

  libsw641si.so                                                1365

  applicat.rdb                                                 1580
```

如此输出所示，大部分 StarSuite 冷启动时间是由于等待 I/O 造成的。（等待 I/O 花费 13.1 秒，启动 CPU 花费 3.6 秒。）在热启动 StarSuite 软件时，运行该脚本显示出页面高速缓存已经消除了 I/O 时间，如以下示例输出所示：

```
Time breakdown (milliseconds):

  <I/O wait>                                                       0

  <on cpu>                                                      2860



I/O wait breakdown (milliseconds):

  temp                                                             0
```

```
  soffice.tmp                                                    0

  <unknown>                                                      0

  Office                                                         0
```

冷启动输出显示，与任何其他文件相比，文件 `applicat.rdb` 引起的 I/O 等待时间最长。此结果可能是由于对文件执行了许多 I/O 操作造成的。要研究对此文件执行的 I/O 操作，可使用以下 D 脚本：

```
io:::start

/execname == "soffice.bin" && args[2]->fi_name == "applicat.rdb"/

{

    @ = lquantize(args[2]->fi_offset != -1 ?

        args[2]->fi_offset / (1000 * 1024) : -1, 0, 1000);

}
```

此脚本使用 `fileinfo_t` 结构的 `fi_offset` 字段来了解要访问文件的哪些部分（以兆字节为单位）。在 StarSuite 软件冷启动期间运行此脚本将产生与以下示例类似的输出：

```
# dtrace -s ./applicat.d

dtrace: script ’./applicat.d’ matched 4 probes

^C




           value  ------------- Distribution ------------  count

             < 0 |                                         0

               0 |@@@                                      28

               1 |@@                                       17

               2 |@@@@                                     35

               3 |@@@@@@@@@@                               72
```

```
4 |@@@@@@@@@@                78
5 |@@@@@@@@                  65
6 |                          0
```

此输出指示仅访问文件的前六兆字节，这可能是因为文件的大小为六兆字节。输出还指示未访问整个文件。如果要缩短 StarSuite 的冷启动时间，则可能需要了解文件的访问模式。如果文件的各个必需部分大多数是相连的，则可能缩短 StarSuite 冷启动时间的一个方法是在运行应用程序之前先运行侦察线程，从而提前执行文件所需的 I/O 操作。（如果通过使用 `mmap(2)` 来访问文件，则此方法最简单。）此策略可使冷启动时间缩短大约 1.6 秒，但会额外增加应用程序的复杂程度和维护负担。通过任一方法，使用 `io` 提供器收集的数据有助于更好地了解此类工作最终带来的好处。

稳定性

`io` 提供器使用 DTrace 的稳定性机制来说明其稳定性，如下表所示。有关稳定性机制的更多信息，请参见第 39 章。

元素	名称稳定性	数据稳定性	相关性类
提供器	发展中	发展中	ISA
模块	专用	专用	未知
函数	专用	专用	未知
名称	发展中	发展中	ISA
参数	发展中	发展中	ISA

◆ ◆ ◆ 第 28 章

mib 提供器

mib 提供器提供了与 Solaris 管理信息库 (management information base, MIB) 中的计数器对应的探测器。MIB 计数器可供简单网络管理协议 (simple network management protocol, SNMP) 使用，以便对异构联网实体进行远程监视。此外，也可以通过 kstat(1M) 和 netstat(1M) 命令查看这些计数器。mib 提供器便于快速查看通过使用远程或本地联网监视器观察到的异常联网行为。

探测器

mib 提供器为多个 MIB 中的计数器提供了探测器。在表 28–1 中，列出了各种协议，这些协议可导出由 mib 提供器检测的 MIB。该表中列出了指定部分或全部 MIB 的文档、可用于访问正在运行的计数器（使用 kstat(1M) -n *statistic* 选项）的内核统计信息名称，以及列出探测器完整定义的表。另外，还可通过 netstat(1M) 的 -s 选项来查看所有 MIB 计数器。

表 28–1 mib 探测器

协议	MIB 描述	内核统计信息	mib 探测器表
ICMP	RFC 1213	icmp	表 28–2
IP	RFC 1213	ip	表 28–3
IPsec	—	ip	表 28–4
IPv6	RFC 2465	—	表 28–5
SCTP	“SCTP MIB”（Internet 草稿）	sctp	表 28–7
TCP	RFC 1213	tcp	表 28–8
UDP	RFC 1213	udp	表 28–9

表 28–2 ICMP mib 探测器

icmpInAddrMaskReps	接收 ICMP（Internet Control Message Protocol，Internet 控制消息协议）地址掩码回复消息时将触发的探测器。
icmpInAddrMasks	接收 ICMP 地址掩码请求消息时将触发的探测器。
icmpInBadRedirects	接收在某些方面确定存在格式错误（未知的 ICMP 代码、发送器或目标断开链接等）的 ICMP 重定向消息时将触发的探测器。
icmpInCksumErrs	接收校验和错误的 ICMP 消息时将触发的探测器。
icmpInDestUnreachs	接收 ICMP 目标不可访问的消息时将触发的探测器。
icmpInEchoReps	接收 ICMP 回显回复消息时将触发的探测器。
icmpInEchos	接收 ICMP 回显请求消息时将触发的探测器。
icmpInErrors	接收确定存在 ICMP 特定错误（错误的 ICMP 校验和、错误长度等）的 ICMP 消息时将触发的探测器。
icmpInFragNeeded	接收 ICMP 目标不可访问（需要分段）的消息，指示发送的包由于大于某一 MTU，并且设置了“不分段”标志而丢失时将触发的探测器。
icmpInMsgs	接收 ICMP 消息时将触发的探测器。触发此探测器时，如果该消息确定存在 ICMP 特定错误，则可能会同时触发 icmpInErrors 探测器。
icmpInOverflows	由于缺少缓冲区空间，导致接收的 ICMP 消息随后被删除时将触发的探测器。
icmpInParmProbs	接收 ICMP 参数问题消息时将触发的探测器。
icmpInRedirects	接收 ICMP 重定向消息时将触发的探测器。
icmpInSrcQuenchs	接收 ICMP 源抑制消息时将触发的探测器。
icmpInTimeExcds	接收 ICMP 超时的消息时将触发的探测器。
icmpInTimestampReps	接收 ICMP 时间标记回复消息时将触发的探测器。
icmpInTimestamps	接收 ICMP 时间标记请求消息时将触发的探测器。
icmpInUnknowns	接收类型未知的 ICMP 消息时将触发的探测器。
icmpOutAddrMaskReps	发送 ICMP 地址掩码回复消息时将触发的探测器。
icmpOutDestUnreachs	发送 ICMP 目标不可访问的消息时将触发的探测器。
icmpOutDrops	由于某些原因（如内存分配失败、广播/多址广播源或目标等），导致外发 ICMP 消息被删除时将触发的探测器。
icmpOutEchoReps	发送 ICMP 回显回复消息时将触发的探测器。
icmpOutErrors	由于在 ICMP 中发现问题（如缓冲区不足）而导致无法发送 ICMP 消息时将触发的探测器。如果在 ICMP 层外面发现错误（如 IP 无法路由生成的数据报），将不会触发该探测器。

表 28–2 ICMP mib 探测器 （续）

icmpOutFragNeeded	发送 ICMP 目标不可访问（需要分段）的消息时将触发的探测器。
icmpOutMsgs	发送 ICMP 消息时将触发的探测器。触发此探测器时，如果该消息确定存在 ICMP 特定错误，则可能会同时触发 icmpOutErrors 探测器。
icmpOutParmProbs	发送 ICMP 参数问题消息时将触发的探测器。
icmpOutRedirects	发送 ICMP 重定向消息时将触发的探测器。对于主机，由于它不发送重定向消息，因此不会触发该探测器。
icmpOutTimeExcds	发送 ICMP 超时的消息时将触发的探测器。
icmpOutTimestampReps	发送 ICMP 时间标记回复消息时将触发的探测器。

表 28–3 IP mib 探测器

ipForwDatagrams	接收未将此计算机设为最终 IP 目标的数据报，并尝试查找将数据报转发至该最终目标的路由时触发的探测器。在未充当 IP 网关的计算机上，此探测器将仅针对通过该计算机进行源路由，以及源路由选项处理成功的包进行触发。
ipForwProhibits	接收未将此计算机设为最终 IP 目标的数据报，但由于禁止将该计算机用作路由器，因此未尝试查找将数据报转发至该最终目标的路由时触发的探测器。
ipFragCreates	由于分段作用生成 IP 数据报分段时将触发的探测器。
ipFragFails	由于无法分段（例如需要分段，但设置了“不分段”标志），导致 IP 数据报被废弃时将触发的探测器。
ipFragOKs	IP 数据报成功分段时将触发的探测器。
ipInCksumErrs	由于 IP 数据包头校验和错误，导致输入数据报被废弃时将触发的探测器。
ipInDelivers	将输入数据报成功传送至 IP 用户协议（包括 ICMP）时触发的探测器。
ipInDiscards	由于某些与包无关的原因（例如，缺少缓冲区空间）而导致输入 IP 数据报被废弃时将触发的探测器。对于在等待重汇编时废弃的任何数据报，不会触发该探测器。
ipInHdrErrors	由于 IP 数据包头错误（如版本号不匹配、格式错误、生成时间超时、处理 IP 选项时发现错误等），导致输入数据报被废弃时将触发的探测器。
ipInIPv6	IPv6 包错误地到达 IPv4 队列时将触发的探测器。
ipInReceives	从某接口接收数据报（即使该数据报被错误接收）时将触发的探测器。
ipInUnknownProtos	由于未知或不受支持的协议导致本地处理数据报在成功接收后被废弃时将触发的探测器。

表 28–3 IP mib 探测器 （续）

ipOutDiscards	由于某些与包无关的原因（例如，缺少缓冲区空间），导致输出 IP 数据报被废弃时将触发的探测器。对于 ipForwDatagrams MIB 计数器中计数的包，如果该包符合此类（自由）废弃条件，将触发该探测器。
ipOutIPv6	通过 IPv4 连接发送 IPv6 包时将触发的探测器。
ipOutNoRoutes	由于找不到将 IP 数据报传送到目标的路由，因而将其废弃时触发的探测器。对于 ipForwDatagrams MIB 计数器中计数的包，如果该包符合此“无路由”条件，将触发该探测器。另外，对于由于所有缺省网关关闭而无法路由的任何数据报，也会触发该探测器。
ipOutRequests	将 IP 数据报提供给 IP，以便通过本地 IP 用户协议（包括 ICMP）传输时触发的探测器。请注意，对于 ipForwDatagrams MIB 计数器中计数的任何包，将不会触发该探测器。
ipOutSwitchIPv6	将连接从使用 IPv4 更改为使用 IPv6 作为其 IP 协议时触发的探测器。
ipReasmDuplicates	IP 重汇编算法确定 IP 分段仅包含以前接收的数据时触发的探测器。
ipReasmFails	IP 重汇编算法检测到任何故障时将触发的探测器。并非所有已废弃的 IP 分段都会触发该探测器，因为某些算法（特别是 RFC 815 中的算法）在接收分段时，会对这些分段进行组合，从而可能丢失分段。
ipReasmOKs	IP 数据报重新汇编成功时将触发的探测器。
ipReasmPartDups	IP 重汇编算法确定 IP 分段既包含一些以前接收的数据，又包含一些新数据时触发的探测器。
ipReasmReqds	接收的 IP 分段需要重新汇编时将触发的探测器。

表 28–4 IPsec mib 探测器

ipsecInFailed	接收的包由于无法与指定的 IPsec 策略匹配而被删除时触发的探测器。
ipsecInSucceeded	接收的包与指定的 IPsec 策略相匹配，并且允许继续处理时触发的探测器。

表 28–5 IPv6 mib 探测器

ipv6ForwProhibits	接收到未将此计算机视为最终 IPv6 目标的数据报，并尝试查找将数据报转发至其最终目标的路由时触发的探测器。
ipv6IfIcmpBadHoplimit	接收跃点限制小于定义的最大值的 ICMPv6 邻居搜索协议消息时将触发的探测器。此类消息可能并非来自邻居，因此将被废弃。
ipv6IfIcmpInAdminProhibs	接收 ICMPv6 目标不可访问（通信被管理禁止）的消息时将触发的探测器。

表 28–5 IPv6 mib 探测器 （续）

ipv6IfIcmpInBadNeighborAdvertisements	接收在某些方面存在格式错误的 ICMPv6 邻居通知消息时将触发的探测器。
ipv6IfIcmpInBadNeighborSolicitations	接收在某些方面存在格式错误的 ICMPv6 邻居请求消息时将触发的探测器。
ipv6IfIcmpInBadRedirects	接收在某些方面存在格式错误的 ICMPv6 重定向消息时将触发的探测器。
ipv6IfIcmpInDestUnreachs	接收 ICMPv6 目标不可访问的消息时将触发的探测器。
ipv6IfIcmpInEchoReplies	接收 ICMPv6 回显回复消息时将触发的探测器。
ipv6IfIcmpInEchos	接收 ICMPv6 回显请求消息时将触发的探测器。
ipv6IfIcmpInErrors	接收确定存在 ICMPv6 特定错误（如错误的 ICMPv6 校验和、错误长度等）的 ICMPv6 消息时将触发的探测器。
ipv6IfIcmpInGroupMembBadQueries	接收在某些方面存在格式错误的 ICMPv6 组成员查询消息时将触发的探测器。
ipv6IfIcmpInGroupMembBadReports	接收在某些方面存在格式错误的 ICMPv6 组成员报告消息时将触发的探测器。
ipv6IfIcmpInGroupMembOurReports	接收 ICMPv6 组成员报告消息时将触发的探测器。
ipv6IfIcmpInGroupMembQueries	接收 ICMPv6 组成员查询消息时将触发的探测器。
ipv6IfIcmpInGroupMembReductions	接收 ICMPv6 组成员减少消息时将触发的探测器。
ipv6IfIcmpInGroupMembResponses	接收 ICMPv6 组成员响应消息时将触发的探测器。
ipv6IfIcmpInGroupMembTotal	接收 ICMPv6 多址广播侦听器搜索消息时将触发的探测器。
ipv6IfIcmpInMsgs	接收 ICMPv6 消息时将触发的探测器。触发此探测器时，如果该消息存在 ICMPv6 特定错误，则可能会同时触发 ipv6IfIcmpInErrors 探测器。
ipv6IfIcmpInNeighborAdvertisements	接收 ICMPv6 邻居通知消息时将触发的探测器。
ipv6IfIcmpInNeighborSolicits	接收 ICMPv6 邻居请求消息时将触发的探测器。
ipv6IfIcmpInOverflows	由于缺少缓冲区空间，导致接收的 ICMPv6 消息随后被删除时将触发的探测器。
ipv6IfIcmpInParmProblems	接收 ICMPv6 参数问题消息时将触发的探测器。
ipv6IfIcmpInRedirects	接收 ICMPv6 重定向消息时将触发的探测器。
ipv6IfIcmpInRouterAdvertisements	接收 ICMPv6 路由器通知消息时将触发的探测器。

表 28–5 IPv6 mib 探测器 （续）

ipv6IfIcmpInRouterSolicits	接收 ICMPv6 路由器请求消息时将触发的探测器。
ipv6IfIcmpInTimeExcds	接收 ICMPv6 超时的消息时将触发的探测器。
ipv6IfIcmpOutAdminProhibs	发送 ICMPv6 目标不可访问（通信被管理禁止）的消息时将触发的探测器。
ipv6IfIcmpOutDestUnreachs	发送 ICMPv6 目标不可访问的消息时将触发的探测器。
ipv6IfIcmpOutEchoReplies	发送 ICMPv6 回显回复消息时将触发的探测器。
ipv6IfIcmpOutEchos	发送 ICMPv6 回显消息时将触发的探测器。
ipv6IfIcmpOutErrors	由于在 ICMPv6 中发现问题（如缺少缓冲区），导致无法发送 ICMPv6 消息时将触发的探测器。如果在 ICMPv6 层外面发现错误（如 IPv6 无法路由生成的数据报），将不会触发该探测器。
ipv6IfIcmpOutGroupMembQueries	发送 ICMPv6 组成员查询消息时将触发的探测器。
ipv6IfIcmpOutGroupMembReductions	发送 ICMPv6 组成员减少消息时将触发的探测器。
ipv6IfIcmpOutGroupMembResponses	发送 ICMPv6 组成员响应消息时将触发的探测器。
ipv6IfIcmpOutMsgs	发送 ICMPv6 消息时将触发的探测器。触发此探测器时，如果该消息存在 ICMPv6 特定错误，则可能会同时触发 ipv6IfIcmpOutErrors 探测器。
ipv6IfIcmpOutNeighborAdvertisements	发送 ICMPv6 邻居通知消息时将触发的探测器。
ipv6IfIcmpOutNeighborSolicits	发送 ICMPv6 邻居请求消息时将触发的探测器。
ipv6IfIcmpOutParmProblems	发送 ICMPv6 参数问题消息时将触发的探测器。
ipv6IfIcmpOutPktTooBigs	发送 ICMPv6 包太大的消息时将触发的探测器。
ipv6IfIcmpOutRedirects	发送 ICMPv6 重定向消息时将触发的探测器。对于主机，由于它不发送重定向消息，因此不会触发该探测器。
ipv6IfIcmpOutRouterAdvertisements	发送 ICMPv6 路由器通知消息时将触发的探测器。
ipv6IfIcmpOutRouterSolicits	发送 ICMPv6 路由器请求消息时将触发的探测器。
ipv6IfIcmpOutTimeExcds	发送 ICMPv6 超时的消息时将触发的探测器。
ipv6InAddrErrors	由于 IPv6 数据包头目标字段中的 IPv6 地址不是该实体可接收的有效地址而导致输入数据报被废弃时触发的探测器。对于无效的地址（如 ::0）和不受支持的地址（如带有未分配前缀的地址），将触发该探测器。对于未配置为充当 IPv6 路由器，因而不转发数据报的计算机，如果数据报由于目标地址非本地地址而被废弃，则会触发该探测器。

表 28–5 IPv6 mib 探测器 （续）

ipv6InDelivers	将输入数据报成功传送至 IPv6 用户协议（包括 ICMPv6）时触发的探测器。
ipv6InDiscards	由于某些与包无关的原因（例如，缓冲区空间不足）而导致输入 IPv6 数据报被废弃时触发的探测器。对于在等待重汇编时被废弃的任何数据报，不会触发该探测器。
ipv6InHdrErrors	由于 IPv6 数据包头错误（如版本号不匹配、格式错误、超出跃点计数、处理 IPv6 选项时发现错误等）而导致输入数据报被废弃时将触发的探测器。
ipv6InIPv4	IPv4 包错误到达 IPv6 队列时将触发的探测器。
ipv6InMcastPkts	接收多址广播 IPv6 包时将触发的探测器。
ipv6InNoRoutes	由于找不到将路由的 IPv6 数据报传送到目标的路由而废弃该数据报时触发的探测器。此探测器将仅针对来自外部的包进行触发。
ipv6InReceives	从接口接收到 IPv6 数据报（即使该数据报被错误接收）时将触发的探测器。
ipv6InTooBigErrors	接收的分段超过最大分段大小时将触发的探测器。
ipv6InTruncatedPkts	由于数据报帧包含的数据不足而导致输入数据报被废弃时将触发的探测器。
ipv6InUnknownProtos	由于协议未知或不受支持，而导致本地寻址 IPv6 数据报在成功接收后被废弃时将触发的探测器。
ipv6OutDiscards	由于某些与包无关的原因（例如，缺少缓冲区空间），导致输出 IPv6 数据报被废弃时将触发的探测器。对于 ipv6OutForwDatagrams MIB 计数器中计数的包，如果该包符合此类（自由）废弃条件，将触发该探测器。
ipv6OutForwDatagrams	接收未将此计算机设为最终 IPv6 目标的数据报，并尝试查找将数据报转发至该最终目标的路由时触发的探测器。在未充当 IPv6 路由器的计算机上，此探测器将仅针对通过该计算机进行源路由，以及源路由选项处理成功的包进行触发。
ipv6OutFragCreates	由于分段作用生成 IPv6 数据报分段时将触发的探测器。
ipv6OutFragFails	由于无法分段（例如，设置了“不分段”标志），导致 IPv6 数据报被废弃时将触发的探测器。
ipv6OutFragOKs	IPv6 数据报成功分段时将触发的探测器。
ipv6OutIPv4	通过 IPv4 连接发送 IPv6 包时将触发的探测器。

表 28–5 IPv6 mib 探测器 （续）

ipv6OutMcastPkts	发送多址广播包时将触发的探测器。
ipv6OutNoRoutes	由于找不到将 IPv6 数据报传送到目标的路由，因而将其废弃时触发的探测器。此探测器不会针对来自外部的包进行触发。
ipv6OutRequests	将 IPv6 数据报提供给 IPv6，以便通过本地 IPv6 用户协议（包括 ICMPv6）传输时触发的探测器。对于 ipv6ForwDatagrams MIB 计数器中计数的任何包，将不会触发此探测器。
ipv6OutSwitchIPv4	将连接从使用 IPv6 更改为使用 IPv4 作为其 IP 协议时触发的探测器。
ipv6ReasmDuplicates	IPv6 重汇编算法确定 IPv6 分段仅包含以前接收的数据时触发的探测器。
ipv6ReasmFails	IPv6 重汇编算法检测到故障时将触发的探测器。对于所有已废弃的 IPv6 分段，并不一定都会触发该探测器，因为某些算法在接收分段时，会对这些分段进行组合，从而可能丢失分段。
ipv6ReasmOKs	IPv6 数据报重新汇编成功时将触发的探测器。
ipv6ReasmPartDups	IPv6 重汇编算法确定 IPv6 分段既包含一些以前接收的数据，又包含一些新数据时触发的探测器。
ipv6ReasmReqds	接收的 IPv6 分段需要重新汇编时将触发的探测器。

表 28–6 原始 IP mib 探测器

rawipInCksumErrs	接收的原始 IP 包的 IP 校验和错误时将触发的探测器。
rawipInDatagrams	接收原始 IP 包时将触发的探测器。
rawipInErrors	接收在某些方面存在格式错误的原始 IP 包时将触发的探测器。
rawipInOverflows	由于缺少缓冲区空间，导致接收的原始 IP 包随后被删除时将触发的探测器。
rawipOutDatagrams	发送原始 IP 包时将触发的探测器。
rawipOutErrors	由于某一错误状态（通常是因为原始 IP 包在某些方面存在格式错误），导致无法发送原始 IP 包时触发的探测器。

表 28–7 SCTP mib 探测器

sctpAborted	SCTP 关联使用 ABORT 元语从任何状态直接转换为 CLOSED 状态，指示关联未正常终止时触发的探测器。

表 28–7 SCTP mib 探测器 （续）

sctpActiveEstab	SCTP 关联从 COOKIE-ECHOED 状态直接转换为 ESTABLISHED 状态，指示上层已开始关联尝试时触发的探测器。
sctpChecksumError	从对等方接收校验和无效的 SCTP 包时触发的探测器。
sctpCurrEstab	将 SCTP 关联作为读取 sctpCurrEstab MIB 计数器的一部分记录时触发的探测器。如果当前状态为 ESTABLISHED、SHUTDOWN-RECEIVED 或 SHUTDOWN-PENDING，则会记录 SCTP 关联。
sctpFragUsrMsgs	由于 MTU 限制，导致必须对用户消息进行分段时将触发的探测器。
sctpInClosed	在已关闭的 SCTP 关联上接收数据时将触发的探测器。
sctpInCtrlChunks	sctpInCtrlChunks MIB 计数器由于被显式查询，或由于 SCTP 连接关闭而更新时触发的探测器。MIB 计数器的递增值包含在 args[0] 中。
sctpInDupAck	接收重复的 ACK 时将触发的探测器。
sctpInInvalidCookie	接收无效的 cookie 时将触发的探测器。
sctpInOrderChunks	sctpInOrderChunks MIB 计数器由于被显式查询，或由于 SCTP 连接关闭而更新时触发的探测器。MIB 计数器的递增值包含在 args[0] 中。
sctpInSCTPPkts	sctpInSCTPPkts MIB 计数器由于被显式查询，或由于 SCTP 连接关闭而更新时触发的探测器。MIB 计数器的递增值包含在 args[0] 中。
sctpInUnorderChunks	sctpInUnorderChunks MIB 计数器由于被显式查询，或由于 SCTP 连接关闭而更新时触发的探测器。MIB 计数器的递增值包含在 args[0] 中。
sctpListenDrop	传入连接由于某种原因而删除时将触发的探测器。
sctpOutAck	发送选择性确认时将触发的探测器。
sctpOutAckDelayed	为 SCTP 关联执行延迟确认处理时将触发的探测器。作为延迟确认处理一部分发送的任何确认都会触发 sctpOutAck 探测器。
sctpOutCtrlChunks	sctpOutCtrlChunks MIB 计数器由于被显式查询，或由于 SCTP 连接关闭而更新时触发的探测器。MIB 计数器的递增值包含在 args[0] 中。
sctpOutOfBlue	接收到一个在其他方面均正确的 SCTP 包，但接收器无法确定该包所属关联时将触发的探测器。
sctpOutOrderChunks	sctpOutOrderChunks MIB 计数器由于被显式查询，或由于 SCTP 连接关闭而更新时触发的探测器。MIB 计数器的递增值包含在 args[0] 中。
sctpOutSCTPPkts	sctpOutSCTPPkts MIB 计数器由于被显式查询，或由于 SCTP 连接关闭而更新时触发的探测器。MIB 计数器的递增值包含在 args[0] 中。
sctpOutUnorderChunks	sctpOutUnorderChunks MIB 计数器由于被显式查询，或由于 SCTP 连接关闭而更新时触发的探测器。MIB 计数器的递增值包含在 args[0] 中。
sctpOutWinProbe	发送窗口探测器时将触发的探测器。
sctpOutWinUpdate	发送窗口更新时将触发的探测器。

表 28–7 SCTP mib 探测器 （续）

sctpPassiveEstab	SCTP 关联从 CLOSED 状态直接转换为 ESTABLISHED 状态时将触发的探测器。远程端点已开始关联尝试。
sctpReasmUsrMsgs	sctpReasmUsrMsgs MIB 计数器由于被显式查询，或由于 SCTP 连接关闭而更新时触发的探测器。MIB 计数器的递增值包含在 args[0] 中。
sctpRetransChunks	sctpRetransChunks MIB 计数器由于被显式查询，或由于 SCTP 连接关闭而更新时触发的探测器。MIB 计数器的递增值包含在 args[0] 中。
sctpShutdowns	SCTP 关联从 SHUTDOWN-SENT 或 SHUTDOWN-ACK-SENT 状态直接转换为 CLOSED 状态，指示关联正常终止时触发的探测器。
sctpTimHeartBeatDrop	由于无法接收心跳确认而导致 SCTP 关联异常中止时将触发的探测器。
sctpTimHeartBeatProbe	发送 SCTP 心跳确认时将触发的探测器。
sctpTimRetrans	对关联执行基于计时器的重新传输处理时将触发的探测器。
sctpTimRetransDrop	由于无法长时间执行基于计时器的重新传输，导致关联中止时将触发的探测器。

表 28–8 TCP mib 探测器

tcpActiveOpens	TCP 连接从 CLOSED 状态直接转换为 SYN_SENT 状态时将触发的探测器。
tcpAttemptFails	TCP 连接从 SYN_SENT 或 SYN_RCVD 状态直接转换为 CLOSED 状态，同时又从 SYN_RCVD 状态直接转换为 LISTEN 状态时将触发的探测器。
tcpCurrEstab	将 TCP 连接作为读取 tcpCurrEstab MIB 计数器的一部分记录时触发的探测器。如果 TCP 连接的当前状态为 ESTABLISHED 或 CLOSE_WAIT，则会对其进行记录。
tcpEstabResets	TCP 连接从 ESTABLISHED 或 CLOSE_WAIT 状态直接转换为 CLOSED 状态时将触发的探测器。
tcpHalfOpenDrop	由于处于 SYN_RCVD 状态的连接队列已满，导致连接被删除时将触发的探测器。
tcpInAckBytes	接收以前发送的数据的 ACK 时将触发的探测器。确认字节数通过 args[0] 传递。
tcpInAckSegs	接收以前发送的段的 ACK 时将触发的探测器。
tcpInAckUnsent	接收未发送的段的 ACK 时将触发的探测器。
tcpInClosed	接收处于关闭状态的连接数据时将触发的探测器。
tcpInDataDupBytes	接收某段（如果之前已收到该段中的所有数据）时将触发的探测器。重复段的字节数通过 args[0] 传递。

表 28–8 TCP mib 探测器 （续）

tcpInDataDupSegs	接收某段（如果之前已收到该段中的所有数据）时将触发的探测器。重复段的字节数通过 args[0] 传递。
tcpInDataInorderBytes	接收数据（如果之前已收到排在新数据序列号前面的所有数据）时将触发的探测器。按顺序接收的字节数通过 args[0] 传递。
tcpInDataInorderSegs	接收某段（如果之前已收到排在新段序列号前面的所有数据）时将触发的探测器。
tcpInDataPartDupBytes	接收某段（如果该段中的部分数据之前已收到，而部分数据为新数据）时将触发的探测器。重复的字节数通过 args[0] 传递。
tcpInDataPartDupSegs	接收某段（如果该段中的部分数据之前已收到，而部分数据为新数据）时将触发的探测器。重复的字节数通过 args[0] 传递。
tcpInDataPastWinBytes	接收位于当前接收窗口之后的数据时将触发的探测器。字节数包含在 args[0] 中。
tcpInDataPastWinSegs	接收位于当前接收窗口之后的段时将触发的探测器。
tcpInDataUnorderBytes	接收数据（如果排在新数据序列号前面的某些数据丢失）时将触发的探测器。无序接收的字节数通过 args[0] 传递。
tcpInDataUnorderSegs	接收某段（如果排在新数据序列号前面的某些数据丢失）时将触发的探测器。
tcpInDupAck	接收重复的 ACK 时将触发的探测器。
tcpInErrs	在已接收的段中发现 TCP 错误（如 TCP 校验和错误）时将触发的探测器。
tcpInSegs	接收某段（即使随后在该段中发现阻止进一步处理的错误）时将触发的探测器。
tcpInWinProbe	接收窗口探测器时将触发的探测器。
tcpInWinUpdate	接收窗口更新时将触发的探测器。
tcpListenDrop	由于侦听队列已满，导致传入连接被删除时将触发的探测器。
tcpListenDropQ0	由于处于 SYN_RCVD 状态的连接队列已满，导致连接被删除时将触发的探测器。
tcpOutAck	发送 ACK 时将触发的探测器。
tcpOutAckDelayed	在初始延迟后发送 ACK 时将触发的探测器。
tcpOutControl	发送 SYN、FIN 或 RST 时将触发的探测器。
tcpOutDataBytes	发送数据时将触发的探测器。发送的字节数包含在 args[0] 中。
tcpOutDataSegs	发送段时将触发的探测器。
tcpOutFastRetrans	将段作为快速重新传输算法的一部分进行重新传输时触发的探测器。

表 28–8 TCP mib 探测器 （续）

tcpOutRsts	发送设置了 RST 标志的段时将触发的探测器。
tcpOutSackRetransSegs	通过已启用选择性确认的连接重新传输段时将触发的探测器。
tcpOutSegs	发送至少包含一个未重新传输字节的段时将触发的探测器。
tcpOutUrg	发送设置了 URG 标志和有效的紧急指针的段时将触发的探测器。
tcpOutWinProbe	发送窗口探测器时将触发的探测器。
tcpOutWinUpdate	发送窗口更新时将触发的探测器。
tcpPassiveOpens	TCP 连接从 LISTEN 状态直接转换为 SYN_RCVD 状态时将触发的探测器。
tcpRetransBytes	重新传输数据时将触发的探测器。重新传输的字节数包含在 args[0] 中。
tcpRetransSegs	发送包含一个或多个已重新传输字节的段时将触发的探测器。
tcpRttNoUpdate	接收数据，但没有可用于更新 RTT 的时间标记信息时将触发的探测器。
tcpRttUpdate	接收包含 RTT 更新所需的时间标记信息的数据时将触发的探测器。
tcpTimKeepalive	对连接执行基于计时器的保持活动处理时将触发的探测器。
tcpTimKeepaliveDrop	保持活动处理导致连接终止时将触发的探测器。
tcpTimKeepaliveProbe	将保持活动探测器作为保持活动处理的一部分发送时触发的探测器。
tcpTimRetrans	对连接执行基于计时器的重新传输处理时将触发的探测器。
tcpTimRetransDrop	由于无法长时间执行基于计时器的重新传输，导致连接终止时将触发的探测器。

表 28–9 UDP mib 探测器

udpInCksumErrs	由于 UDP 校验和错误，导致数据报被废弃时将触发的探测器。
udpInDatagrams	接收 UDP 数据报时将触发的探测器。
udpInErrors	由于包头格式错误或未能分配内部缓冲区，导致接收的 UDP 数据报被废弃时将触发的探测器。
udpInOverflows	由于缺少缓冲区空间，导致接收的 UDP 数据报随后被删除时将触发的探测器。
udpNoPorts	在未绑定套接字的端口上接收 UDP 数据报时将触发的探测器。
udpOutDatagrams	发送 UDP 数据报时将触发的探测器。
udpOutErrors	由于某一错误状态（通常是因为 UDP 数据报在某些方面存在格式错误），导致无法发送该数据报时触发的探测器。

参数

每个 mib 探测器的唯一参数都具有相同语义：args[0] 包含计数器的递增值。对于大多数 mib 探测器，args[0] 始终包含值 1，但对于某些探测器，args[0] 则可接受任意的正值。对于此类探测器，在探测器说明中对 args[0] 的含义进行了说明。

稳定性

mib 提供器使用 DTrace 的稳定性机制描述其稳定性，如下表所示。有关稳定性机制的更多信息，请参见第 39 章。

元素	名称稳定性	数据稳定性	相关性类
提供器	发展中	发展中	ISA
模块	专用	专用	未知
函数	专用	专用	未知
名称	发展中	发展中	ISA
参数	发展中	发展中	ISA

◆ ◆ ◆ 第 29 章

29 fpuinfo 提供器

fpuinfo 提供器提供了与 SPARC 微处理器中的模拟浮点指令对应的探测器。 虽然大多数浮点指令是在硬件中执行的，但有些浮点操作会陷入操作系统以进行模拟。 浮点操作需要进行操作系统模拟的条件特定于微处理器实现。需要进行模拟的操作很少。 但是，如果应用程序经常使用这些操作之一，则可能会严重影响性能。 fpuinfo 提供器通过查看 kstat(1M) 和 fpu_info 内核统计信息或 trapstat(1M) 和 fp-xcp-other 陷阱可以对浮点模拟进行快速调查。

探测器

fpuinfo 提供器为可以模拟的每一种类型的浮点指令都提供了探测器。 fpuinfo 提供器包含 CPU 的“名称稳定性”；探测器的名称特定于微处理器实现，不可以在同一系列的不同微处理器中使用。 例如，所列出的一些探测器可能仅允许在 UltraSPARC-III 中使用，而不能在 UltraSPARC-III+ 中使用，反之亦然。

表 29–1 中描述了 fpuinfo 探测器。

表 29–1 fpuinfo 探测器

fpu_sim_fitoq	内核模拟 fitoq 指令时将触发的探测器。
fpu_sim_fitod	内核模拟 fitod 指令时将触发的探测器。
fpu_sim_fitos	内核模拟 fitos 指令时将触发的探测器。
fpu_sim_fxtoq	内核模拟 fxtoq 指令时将触发的探测器。
fpu_sim_fxtod	内核模拟 fxtod 指令时将触发的探测器。
fpu_sim_fxtos	内核模拟 fxtos 指令时将触发的探测器。
fpu_sim_fqtox	内核模拟 fqtox 指令时将触发的探测器。
fpu_sim_fdtox	内核模拟 fdtox 指令时将触发的探测器。

表 29–1 fpuinfo 探测器 （续）

fpu_sim_fstox	内核模拟 fstox 指令时将触发的探测器。
fpu_sim_fqtoi	内核模拟 fqtoi 指令时将触发的探测器。
fpu_sim_fdtoi	内核模拟 fdtoi 指令时将触发的探测器。
fpu_sim_fstoi	内核模拟 fstoi 指令时将触发的探测器。
fpu_sim_fsqrtq	内核模拟 fsqrtq 指令时将触发的探测器。
fpu_sim_fsqrtd	内核模拟 fsqrtd 指令时将触发的探测器。
fpu_sim_fsqrts	内核模拟 fsqrts 指令时将触发的探测器。
fpu_sim_fcmpeq	内核模拟 fcmpeq 指令时将触发的探测器。
fpu_sim_fcmped	内核模拟 fcmped 指令时将触发的探测器。
fpu_sim_fcmpes	内核模拟 fcmpes 指令时将触发的探测器。
fpu_sim_fcmpq	内核模拟 fcmpq 指令时将触发的探测器。
fpu_sim_fcmpd	内核模拟 fcmpd 指令时将触发的探测器。
fpu_sim_fcmps	内核模拟 fcmps 指令时将触发的探测器。
fpu_sim_fdivq	内核模拟 fdivq 指令时将触发的探测器。
fpu_sim_fdivd	内核模拟 fdivd 指令时将触发的探测器。
fpu_sim_fdivs	内核模拟 fdivs 指令时将触发的探测器。
fpu_sim_fdmulx	内核模拟 fdmulx 指令时将触发的探测器。
fpu_sim_fsmuld	内核模拟 fsmuld 指令时将触发的探测器。
fpu_sim_fmulq	内核模拟 fmulq 指令时将触发的探测器。
fpu_sim_fmuld	内核模拟 fmuld 指令时将触发的探测器。
fpu_sim_fmuls	内核模拟 fmuls 指令时将触发的探测器。
fpu_sim_fsubq	内核模拟 fsubq 指令时将触发的探测器。
fpu_sim_fsubd	内核模拟 fsubd 指令时将触发的探测器。
fpu_sim_fsubs	内核模拟 fsubs 指令时将触发的探测器。
fpu_sim_faddq	内核模拟 faddq 指令时将触发的探测器。
fpu_sim_faddd	内核模拟 faddd 指令时将触发的探测器。
fpu_sim_fadds	内核模拟 fadds 指令时将触发的探测器。
fpu_sim_fnegd	内核模拟 fnegd 指令时将触发的探测器。
fpu_sim_fnegq	内核模拟 fneqq 指令时将触发的探测器。

表 29–1 fpuinfo 探测器 （续）

fpu_sim_fnegs	内核模拟 fnegs 指令时将触发的探测器。
fpu_sim_fabsd	内核模拟 fabsd 指令时将触发的探测器。
fpu_sim_fabsq	内核模拟 fabsq 指令时将触发的探测器。
fpu_sim_fabss	内核模拟 fabss 指令时将触发的探测器。
fpu_sim_fmovd	内核模拟 fmovd 指令时将触发的探测器。
fpu_sim_fmovq	内核模拟 fmovq 指令时将触发的探测器。
fpu_sim_fmovs	内核模拟 fmovs 指令时将触发的探测器。
fpu_sim_fmovr	内核模拟 fmovr 指令时将触发的探测器。
fpu_sim_fmovcc	内核模拟 fmovcc 指令时将触发的探测器。

参数

fpuinfo 探测器没有参数。

稳定性

fpuinfo 提供器使用 DTrace 的稳定性机制来描述其稳定性，如下表所示。有关稳定性机制的更多信息，请参见第 39 章。

元素	名称稳定性	数据稳定性	相关性类
提供器	发展中	发展中	CPU
模块	专用	专用	未知
函数	专用	专用	未知
名称	发展中	发展中	CPU
参数	发展中	发展中	CPU

◆ ◆ ◆ 第 30 章

30 pid 提供器

使用 pid 提供器，可跟踪用户进程中任何函数的进入和返回，以及按绝对地址或函数偏移量指定的任何指令。不启用探测器时，pid 提供器不会产生探测影响。启用探测器后，探测器将仅对其跟踪的那些进程产生探测影响。

命名 pid 探测器

实际上，pid 定义了一个提供器类。每个进程都可能具有独立关联的 pid 提供器。例如，ID 为 123 的进程将使用 pid123 提供器进行跟踪。对于来自这些提供器的其中之一的探测器，探测器说明的模块部分将引用对应进程的地址空间中装入的对象。以下示例使用 mdb(1) 显示对象列表：

```
$ mdb -p 1234

Loading modules: [ ld.so.1 libc.so.1 ]

> ::objects

    BASE    LIMIT     SIZE NAME

   10000    34000    24000 /usr/bin/csh

ff3c0000 ff3e8000    28000 /lib/ld.so.1

ff350000 ff37a000    2a000 /lib/libcurses.so.1

ff200000 ff2be000    be000 /lib/libc.so.1

ff3a0000 ff3a2000     2000 /lib/libdl.so.1

ff320000 ff324000     4000 /platform/sun4u/lib/libc_psr.so.1
```

在探测器说明中，将根据文件名而不是其完整路径名来命名对象。此外，也可以省略“.1”或“so.1”后缀。以下所有示例命名的探测器都相同：

```
pid123:libc.so.1:strcpy:entry

pid123:libc.so:strcpy:entry

pid123:libc:strcpy:entry
```

第一个示例是探测器的实际名称。其他示例则是为方便而使用的别名，在内部，该别名将被替换为完整的装入对象名称。

对于可执行文件的装入对象，可以使用别名 a.out。以下两种探测器描述指定了同一探测器：

```
pid123:csh:main:return

pid123:a.out:main:return
```

与所有固定 DTrace 探测器一样，探测器描述的函数字段指定模块字段中的函数。用户应用程序二进制文件的同一个函数可能有多个名称。例如，mutex_lock 可能为 libc.so.1 中函数 pthread_mutex_lock 的备选名称。DTrace 为这样的函数选择一个标准名称，并在内部使用该名称。以下示例说明了 DTrace 在内部如何将模块和函数名称重新映射为标准形式：

```
# dtrace -q -n pid101267:libc:mutex_lock:entry'{ \

    printf("%s:%s:%s:%s\n", probeprov, probemod, probefunc, probename); }'

pid101267:libc.so.1:pthread_mutex_lock:entry

^C
```

这种自动重命名意味着，您启用的探测器名称可能与实际启用的探测器名称略有不同。在运行同一个 Solaris 发行版的系统上，每次运行 Dtrace 时，标准名称总是保持一致。

有关如何有效使用 pid 提供器的示例，请参见第 33 章。

函数边界探测器

使用pid 提供器，可以对用户程序中函数的进入和返回进行跟踪，就像 FBT 提供器为内核提供该功能一样。为适应用户进程，可对本手册中使用 FBT 提供器跟踪内核函数调用的大多数示例略作修改。

entry 探测器

调用被跟踪函数时将触发entry 探测器。entry 探测器的参数为被跟踪函数的参数的值。

return 探测器

被跟踪函数返回或对另一个函数进行尾部调用时，将触发 return 探测器。arg0 的值是函数中返回指令的偏移；arg1 存储返回值。

函数偏移探测器

使用 pid 提供器，可以跟踪函数中的任何指令。例如，要跟踪函数 main() 4 个字节处的指令，可以使用与以下示例类似的命令：

```
pid123:a.out:main:4
```

每次程序执行地址 main+4 处的指令时，将激活此探测器。未定义偏移探测器参数。uregs[] 数组可帮助您检查这些探测器位置的进程状态。有关更多信息，请参见第 418 页中的“uregs[] 数组”。

稳定性

pid 提供器使用 DTrace 的稳定性机制来说明其稳定性，如下表所示。有关稳定性机制的更多信息，请参见第 39 章。

元素	名称稳定性	数据稳定性	相关性类
提供器	发展中	发展中	ISA
模块	专用	专用	未知
函数	专用	专用	未知
名称	发展中	发展中	ISA
参数	专用	专用	未知

◆ ◆ ◆ 第 31 章

31 plockstat 提供器

plockstat 提供器提供了可用于观察用户级同步元语（包括锁定争用次数和暂挂次数）行为的探测器。plockstat (1M) 命令是使用 plockstat 提供器收集用户级锁定事件数据的 DTrace 使用者。

概述

plockstat 提供器提供了用于以下类型事件的探测器：

争用事件　这些探测器对应于用户级同步元语的争用，并在强制线程等待资源可用时触发。为了避免出现争用情况，通常会对 Solaris 进行优化，以便不发生延迟的争用；应使用这些探测器来了解出现争用位置的情况。由于设计的争用相对较少，因此启用争用事件探测器通常不会产生严重的探测影响；启用这些探测器时可以不必关注实际的性能影响。

暂挂事件　这些探测器对应于获取、释放或处理用户级同步元语。同样，这些探测器可用于回答有关处理用户级同步元语的方法的任意问题。由于应用程序获取和释放同步元语通常很频繁，因此启用暂挂事件探测器将比启用争用事件探测器产生更大的探测影响。虽然启用暂挂事件探测器产生的探测影响可能会很大，但并非不可接受；仍然可以在生产环境应用程序中放心地启用这些探测器。

错误事件　这些探测器对应于在获取或释放用户级同步元语时遇到的任何类型的异常行为。当线程在用户级同步元语上阻塞时，可以使用这些事件检测遇到的错误。错误事件会非常少见，因此启用这些探测器不会产生严重的探测影响。

互斥探测器

互斥强制临界段相互排斥。当线程尝试使用 mutex_lock(3C) 或 pthread_mutex_lock(3C) 获取另一个线程暂挂的互斥时，它将确定拥有线程是否正在其他 CPU 中运行。如果是这样，获取线程将旋转一段时间，以等待互斥可用。如果属主未在另一个 CPU 上运行，获取线程将阻塞。

表 31–1 中列出了与互斥有关的四个 plockstat 探测器。对于每一个探测器，arg0 包含指向代表互斥的 mutex_t 或 pthread_mutex_t 结构（这些结构的类型相同）的指针。

表 31–1 互斥探测器

mutex-acquire	获取互斥之后将立即触发的暂挂事件探测器。arg1 包含布尔值，用于指示获取操作对于递归互斥是否是递归的。arg2 表示获取线程在此互斥上旋转的重复次数。如果在获取此互斥时触发 mutex-spin 探测器，arg2 将为非零值。
mutex-block	线程在暂挂互斥上阻塞之前将触发的争用事件探测器。单次锁定获取可能会同时触发 mutex-block 和 mutex-spin。
mutex-spin	线程开始在暂挂互斥上旋转之前将触发的争用事件探测器。单次锁定获取可能会同时触发 mutex-block 和 mutex-spin。
mutex-release	释放互斥之后将立即触发的暂挂事件探测器。arg1 包含布尔值，用于指示事件是否对应于递归互斥上的递归释放。
mutex-error	互斥操作中遇到错误时将触发的错误事件探测器。arg1 是遇到的错误的 errno 值。

读取器/写入器锁定探测器

读取器/写入器锁定允许临界段中一次有多个读取器或一个写入器，但不允许二者同时存在。 这些锁定通常用于搜索操作比修改操作更频繁或线程在临界段中花费大量时间的结构。 用户使用 Solaris rwlock(3C) 或 POSIX pthread_rwlock_init(3C) 接口与读取器/写入器锁定交互。

表 31–2 中说明了与读取器/写入器锁定有关的探测器。对于每一个探测器，arg0 包含指向代表自适应锁定的 rwlock_t 或 pthread_rwlock_t 结构（这些结构的类型相同）的指针。arg1 包含布尔值，用于指示操作是否作为写入器。

表 31–2 读取器/写入器锁定探测器

rw-acquire	获取读取器/写入器锁定之后将立即触发的暂挂事件探测器。
rw-block	尝试获取锁定时，在线程阻塞之前将触发的争用事件探测器。如果启用该探测器，rw-acquire 探测器或 rw-error 探测器将在 rw-block 之后触发。

表 31–2 读取器/写入器锁定探测器 （续）

rw-release	释放读取器/写入器锁定之后将立即触发的暂挂事件探测器。
rw-error	读取器/写入器锁定操作期间遇到错误时将触发的错误事件探测器。arg1 是遇到的错误的 errno 值。

稳定性

plockstat 提供器使用 DTrace 的稳定性机制来说明其稳定性，如下表所示。有关稳定性机制的更多信息，请参见第 39 章。

元素	名称稳定性	数据稳定性	相关性类
提供器	发展中	发展中	ISA
模块	专用	专用	未知
函数	专用	专用	未知
名称	发展中	发展中	ISA
参数	发展中	发展中	ISA

◆ ◆ ◆ 第 32 章

32 fasttrap 提供器

fasttrap 提供器允许在特定、预编程的用户进程位置进行跟踪。与其他大多数 DTrace 提供器不一样，fasttrap 提供器的设计目的不是跟踪系统活动，而是为 DTrace 使用者提供一种向 DTrace 框架中加入信息的方法，即通过激活 fasttrap 探测器向其中加入信息。

探测器

fasttrap 提供器提供了单个探测器 fasttrap:::fasttrap，只要用户级进程在内核中进行某个 DTrace 调用，就将触发该探测器。激活该探测器的 DTrace 调用目前尚未正式发布。

稳定性

fasttrap 提供器使用 DTrace 的稳定性机制说明其稳定性，如下表所示。有关稳定性机制的更多信息，请参见第 39 章。

元素	名称稳定性	数据稳定性	相关性类
提供器	发展中	发展中	ISA
模块	专用	专用	未知
函数	专用	专用	未知
名称	发展中	发展中	ISA
参数	发展中	发展中	ISA

◆ ◆ ◆ 第 33 章

33 用户进程跟踪

DTrace 是一款功能非常强大的、用于了解用户进程行为的工具。调试、分析性能问题或简单了解复杂应用程序的行为时，DTrace 的作用不可估量。本章重点介绍与跟踪用户进程活动有关的 DTrace 工具，并提供用于说明其用途的示例。

copyin() 和 copyinstr() 子例程

与大多数传统调试器或观察工具相比，DTrace 与进程之间的交互略有不同。这类工具多数可能会在进程范围内执行，从而使用户可直接取消对指向程序变量的指针的引用。 DTrace 探测器在 Solaris 内核中执行，而不是在进程内或作为进程的一部分执行。为访问进程数据，探测器需要使用 copyin() 或 copyinstr() 子例程，以便将用户进程数据复制到该内核的地址空间中。

例如，可以参考下面的 write(2) 系统调用：

```
ssize_t write(int fd, const void *buf, size_t nbytes);
```

以下 D 程序说明试图列显传递给 write(2) 系统调用的字符串内容的操作不正确：

```
syscall::write:entry

{

    printf("%s", stringof(arg1)); /* incorrect use of arg1 */

}
```

如果尝试运行此脚本，DTrace 将生成与以下示例类似的错误消息：

```
dtrace: error on enabled probe ID 1 (ID 37: syscall::write:entry): \

    invalid address (0x10038a000) in action #1
```

arg1 变量（该变量包含 *buf* 参数的值）是一个引用执行系统调用的进程内存的地址。要读取该地址中的字符串，请使用 copyinstr() 子例程，并借助 printf() 操作记录其结果：

```
syscall::write:entry

{

    printf("%s", copyinstr(arg1)); /* correct use of arg1 */
```

此脚本输出将显示传递给 write(2) 系统调用的所有字符串。不过，有时也可能会显示类似以下的异常输出：

```
  0     37                        write:entry madaï¿½ï¿½ï¿½
```

copyinstr() 子例程对输入参数进行操作，该参数是以 Null 结尾的 ASCII 字符串的用户地址。但是，传递给 write(2) 系统调用的缓冲区可能会引用二进制数据，而非 ASCII 字符串。要仅列显调用方指定数量的字符串，请使用 copyin() 子例程，该子例程将大小作为其第二个参数：

```
syscall::write:entry

{

    printf("%s", stringof(copyin(arg1, arg2)));

}
```

请注意，为使 DTrace 将通过 copyin() 检索的用户数据正确转换为字符串，必须使用 stringof 运算符。使用 copyinstr() 时，无需使用 stringof，因为此函数返回的类型始终为 string。

避免错误

copyin() 和 copyinstr() 子例程无法从尚未访问的用户地址读取，因此，如果包含某个地址的页面没有得到访问，则即使该地址有效也可能会产生错误。以下面的示例为例：

```
# dtrace -n syscall::open:entry'{ trace(copyinstr(arg0)); }'

dtrace: description 'syscall::open:entry' matched 1 probe

CPU     ID                    FUNCTION:NAME

dtrace: error on enabled probe ID 2 (ID 50: syscall::open:entry): invalid address
```

```
(0x9af1b) in action #1 at DIF offset 52
```

在以上示例输出中，该应用程序运行正常，且 `arg0` 的地址是有效的，但其引用了尚未被对应进程访问的页面。为解决此问题，请在跟踪前等待内核或应用程序使用相应数据。例如，您可以等到系统调用返回时再应用 `copyinstr()`，如以下示例所示：

```
# dtrace -n syscall::open:entry'{ self->file = arg0; }' \
-n syscall::open:return'{ trace(copyinstr(self->file)); self->file = 0; }'
dtrace: description 'syscall::open:entry' matched 1 probe
CPU     ID                    FUNCTION:NAME
  2     51                      open:return   /dev/null
```

消除 dtrace(1M) 干扰

如果跟踪对 `write`(2) 系统调用的所有调用，将会产生层叠输出。当其显示输出时，对 `write()` 的每次调用都会导致 `dtrace`(1M) 命令调用 `write()` 等。此反馈循环是一个有关 `dtrace` 命令如何干扰所需数据的较好示例。您可以使用一个简单的谓词来防止跟踪这些不需要的数据：

```
syscall::write:entry
/pid != $pid/
{
    printf("%s", stringof(copyin(arg1, arg2)));
}
```

`$pid` 宏变量扩展为已启用探测器的进程的进程标识符。`pid` 变量包含其线程正在触发了探测器的 CPU 上运行的进程的进程标识符。因此，通过谓词 `/pid != $pid/`，可确保脚本不会跟踪与此脚本本身运行相关的任何事件。

syscall 提供器

通过 syscall 提供器，可以跟踪每个系统调用的进入和返回。系统调用是了解进程行为的良好开端，尤其是在执行进程可能会占用大量时间，或在内核阻塞了进程的情况下。您可以使用 prstat(1M) 命令查看进程在哪些地方消耗时间：

```
$ prstat -m -p 31337

   PID USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX ICX SCL SIG PROCESS/NLWP

 13499 user1     53  44 0.0 0.0 0.0 0.0 2.5 0.0  4K  24  9K   0 mystery/6
```

此示例表明该进程占用了大量系统时间。对该行为的一种可能的解释是，进程正在执行大量系统调用。可以使用在命令行中指定的简单 D 程序，查看哪些系统调用出现的频率最高：

```
# dtrace -n syscall:::entry'/pid == 31337/{ @syscalls[probefunc] = count(); }'

dtrace: description 'syscall:::entry' matched 215 probes

^C


  open                                                            1

  lwp_park                                                        2

  times                                                           4

  fcntl                                                           5

  close                                                           6

  sigaction                                                       6

  read                                                           10

  ioctl                                                          14

  sigprocmask                                                   106

  write                                                        1092
```

该报告显示了调用频率最高的系统调用，在此示例中即 write(2) 系统调用。您可以使用 syscall 提供器，进一步检查所有 write() 系统调用的起因：

```
# dtrace -n syscall::write:entry'/pid == 31337/{ @writes[arg2] = quantize(); }'
dtrace: description 'syscall::write:entry' matched 1 probe
^C


           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@   1037
               2 |@                                        3
               4 |                                         0
               8 |                                         0
              16 |                                         0
              32 |@                                        3
              64 |                                         0
             128 |                                         0
             256 |                                         0
             512 |                                         0
            1024 |@                                        5
            2048 |                                         0
```

以上输出表明，该进程正在执行多个数据量相对较少的 `write()` 系统调用。该比例可能是造成此特定进程性能问题的起因。本示例介绍了一种用于调查系统调用行为的常规方法。

ustack() 操作

通常，在激活特定探测器时对进程线程的栈进行跟踪，可以更详细地检查问题。`ustack()` 操作可跟踪用户线程的栈。例如，如果打开多个文件的进程在 open(2) 系统调用中偶尔出现故障，则可使用 `ustack()` 操作来搜索执行出现故障的 `open()` 的代码路径：

```
syscall::open:entry
```

```
/pid == $1/

{

    self->path = copyinstr(arg0);

}



syscall::open:return

/self->path != NULL && arg1 == -1/

{

    printf("open for ’%s’ failed", self->path);

    ustack();

}
```

此脚本还说明了 $1 宏变量的用法，该变量将采用在 dtrace(1M) 命令行中指定的第一个操作数的值：

```
# dtrace -s ./badopen.d 31337

dtrace: script ’./badopen.d’ matched 2 probes

CPU     ID                    FUNCTION:NAME

  0     40                      open:return open for ’/usr/lib/foo’ failed

              libc.so.1‘__open+0x4

              libc.so.1‘open+0x6c

              420b0

              tcsh‘dosource+0xe0

              tcsh‘execute+0x978

              tcsh‘execute+0xba0

              tcsh‘process+0x50c

              tcsh‘main+0x1d54
```

```
              tcsh`_start+0xdc
```

ustack() 操作将记录栈的程序计数器 (program counter, PC) 值，dtrace(1M) 则通过查找进程符号表，将这些 PC 值解析为符号名。如果 dtrace 无法将 PC 值解析为符号，便会将该值列显为十六进制整数。

如果在设置 ustack() 数据输出格式之前退出或中止进程，dtrace 可能无法将栈跟踪内的 PC 值转换为符号名，并且会强制将其显示为十六进制整数。为处理此限制，请使用 dtrace 的 -c 或 -p 选项指定所关注的进程。有关这些选项和其他选项的详细信息，请参见第 14 章。如果事先不了解进程 ID 或命令，可使用以下 D 程序示例来处理限制：

```
/*
 * This example uses the open(2) system call probe, but this technique
 * is applicable to any script using the ustack() action where the stack
 * being traced is in a process that may exit soon.
 */
 syscall::open:entry
{
    ustack();
    stop_pids[pid] = 1;
}


syscall::rexit:entry
/stop_pids[pid] != 0/
{
    printf("stopping pid %d", pid);
    stop();
    stop_pids[pid] = 0;
}
```

如果对以上脚本的进程中的线程应用 ustack() 操作，则以上脚本将在该进程退出之前停止它。此方法可确保 dtrace 命令将 PC 值解析为符号名。请注意，在使用 stop_pids[pid] 清除动态变量之后，其值将设置为 0。请记住使用 prun(1) 命令设置重新运行已停止进程，否则系统中将聚集许多已停止进程。

uregs[] 数组

通过 uregs[] 数组，可以访问各个用户寄存器。下表列出了与每个支持的 Solaris 系统体系结构对应的 uregs[] 数组索引。

表 33–1 SPARC uregs[] 常量

常量	寄存器
R_G0..R_G7	%g0..%g7 全局寄存器
R_O0..R_O7	%o0..%o7 输出寄存器
R_L0..R_L7	%l0..%l7 本地寄存器
R_I0..R_I7	%i0..%i7 输入寄存器
R_CCR	%ccr 条件代码寄存器
R_PC	%pc 程序计数器
R_NPC	%npc 下一个程序计数器
R_Y	%y 乘/除寄存器
R_ASI	%asi 地址空间标识符寄存器
R_FPRS	%fprs 浮点寄存器状态

表 33–2 x86 uregs[] 常量

常量	寄存器
R_CS	%cs
R_GS	%gs
R_ES	%es
R_DS	%ds
R_EDI	%edi
R_ESI	%esi
R_EBP	%ebp

表 33–2 x86 uregs[] 常量 （续）

常量	寄存器
R_EAX	%eax
R_ESP	%esp
R_EAX	%eax
R_EBX	%ebx
R_ECX	%ecx
R_EDX	%edx
R_TRAPNO	%trapno
R_ERR	%err
R_EIP	%eip
R_CS	%cs
R_ERR	%err
R_EFL	%efl
R_UESP	%uesp
R_SS	%ss

在 AMD64 平台上，uregs 数组的内容与其在 x86 平台上的内容相同，另外还包括下表中列出的附加元素：

表 33–3 amd64 uregs[] 常量

常量	寄存器
R_RSP	%rsp
R_RFL	%rfl
R_RIP	%rip
R_RAX	%rax
R_RCX	%rcx
R_RDX	%rdx
R_RBX	%rbx
R_RBP	%rbp
R_RSI	%rsi

表 33–3 amd64 uregs[] 常量 （续）

常量	寄存器
R_RDI	%rdi
R_R8	%r8
R_R9	%r9
R_R10	%r10
R_R11	%r11
R_R12	%r12
R_R13	%r13
R_R14	%r14
R_R15	%r15

下表中列出的别名可用于所有平台：

表 33–4 公用的 uregs[] 常量

常量	寄存器
R_PC	程序计数器寄存器
R_SP	栈指针寄存器
R_R0	第一个返回码
R_R1	第二个返回码

pid 提供器

使用 pid 提供器，可以跟踪进程中的任何指令。与大多数其他提供器不同，pid 探测器是基于 D 程序中的探测器说明根据需要创建的。因此，除非您亲自启用，否则 dtrace -l 输出中不会列出任何 pid 探测器。

用户函数边界跟踪

对于 pid 提供器，最简单的操作模式是作为与 fbt 提供器类似的用户空间。以下示例程序将跟踪通过单个函数产生的所有函数进入和返回。$1 宏变量（命令行中的第一个操作数）是要跟踪的进程的进程 ID。$2 宏变量（命令行中的第二个操作数）是要从中跟踪所有函数调用的函数的名称。

示例 33–1 userfunc.d:跟踪用户函数的进入和返回

```
pid$1::$2:entry

{

    self->trace = 1;

}



pid$1::$2:return

/self->trace/

{

    self->trace = 0;

}



pid$1:::entry,

pid$1:::return

/self->trace/

{

}
```

键入以上示例脚本，并将其保存在名为 userfunc.d 的文件中，然后通过 chmod 将其转换为可执行文件。此脚本生成的输出与以下示例类似：

```
# ./userfunc.d 15032 execute

dtrace: script './userfunc.d' matched 11594 probes

  0  -> execute

  0    -> execute

  0      -> Dfix

  0      <- Dfix
```

```
  0      -> s_strsave
  0        -> malloc
  0        <- malloc
  0      <- s_strsave
  0      -> set
  0        -> malloc
  0        <- malloc
  0      <- set
  0      -> set1
  0        -> tglob
  0        <- tglob
  0      <- set1
  0      -> setq
  0        -> s_strcmp
  0        <- s_strcmp
...
```

pid 提供器只能用于已在运行的进程。您可以使用 $target 宏变量（参见第 15 章）以及 dtrace -c 和 -p 选项，来创建和抓取所关注的进程，并使用 DTrace 对其进行检测。例如，以下 D 脚本可用于确定通过特定主题进程对 libc 执行的函数调用的分布：

```
pid$target:libc.so::entry
{
    @[probefunc] = count();
}
```

要确定由 date(1) 命令执行的此类调用的分布，请将脚本保存在名为 libc.d 的文件中并执行以下命令：

```
# dtrace -s libc.d -c date

dtrace: script 'libc.d' matched 2476 probes

Fri Jul 30 14:08:54 PDT 2004

dtrace: pid 109196 has exited


  pthread_rwlock_unlock                                             1

  _fflush_u                                                         1

  rwlock_lock                                                       1

  rw_write_held                                                     1

  strftime                                                          1

  _close                                                            1

  _read                                                             1

  __open                                                            1

  _open                                                             1

  strstr                                                            1

  load_zoneinfo                                                     1


...

  _ti_bind_guard                                                   47

  _ti_bind_clear                                                   94
```

跟踪任意指令

您可以使用 pid 提供器跟踪用户函数中的任何指令。pid 提供器将根据需要，为函数中的每条指令创建探测器。每个探测器的名称即函数中与其对应的，以十六进制整数表示的指令的偏移量。例如，要启用与 bar.so 模块的 foo 函数中偏移量为 0x1c 的指令（位于 PID 为 123 的进程中）关联的探测器，可使用以下命令：

```
# dtrace -n pid123:bar.so:foo:1c
```

要启用函数 foo 中的所有探测器（包括每条指令的探测器），可使用以下命令：

```
# dtrace -n pid123:bar.so:foo:
```

此命令展示了一种功能非常强大的、用于调试和分析用户应用程序的方法。由于很难重现不经常发生的错误，因此对其进行调试可能并不容易。通常，您可以在出现故障后确定问题，但这对于重构代码路径已太迟。以下示例演示了如何将 pid 提供器与推理跟踪（请参见第 13 章）进行结合，以便通过跟踪函数中的每条指令来解决该问题。

示例 33–2 errorpath.d:跟踪用户函数调用错误路径

```
pid$1::$2:entry
{
    self->spec = speculation();
    speculate(self->spec);
    printf("%x %x %x %x %x", arg0, arg1, arg2, arg3, arg4);
}

pid$1::$2:
/self->spec/
{
    speculate(self->spec);
}

pid$1::$2:return
/self->spec && arg1 == 0/
{
    discard(self->spec);
```

示例 33–2 errorpath.d: 跟踪用户函数调用错误路径 （续）

```
    self->spec = 0;
}


pid$1::$2:return
/self->spec && arg1 != 0/
{
    commit(self->spec);
    self->spec = 0;
}
```

执行 errorpath.d 将会生成与以下示例类似的输出：

```
# ./errorpath.d 100461 _chdir
dtrace: script ’./errorpath.d’ matched 19 probes
CPU     ID                    FUNCTION:NAME
  0  25253                     _chdir:entry 81e08 6d140 ffbfcb20 656c73 0
  0  25253                     _chdir:entry
  0  25269                         _chdir:0
  0  25270                         _chdir:4
  0  25271                         _chdir:8
  0  25272                         _chdir:c
  0  25273                        _chdir:10
  0  25274                        _chdir:14
  0  25275                        _chdir:18
  0  25276                        _chdir:1c
```

```
      0  25277                         _chdir:20
      0  25278                         _chdir:24
      0  25279                         _chdir:28
      0  25280                         _chdir:2c
      0  25268                    _chdir:return
```

◆ ◆ ◆ 第 34 章

为用户应用程序静态定义跟踪

DTrace 为用户应用程序开发者提供了用于在应用程序代码中定义自定义探测器，以扩充 pid 提供器功能的工具。这些静态探测器在禁用时几乎不会产生任何开销，且可以像任何其他 DTrace 探测器一样动态启用。您可以使用这些静态探测器向 DTrace 用户说明应用程序的语义，不需要具有或了解应用程序的实现知识。本章介绍如何在用户应用程序中定义静态探测器，以及如何使用 DTrace 在用户进程中启用此类探测器。

选择探测器位置

DTrace 允许开发者在应用程序代码（包括完整的应用程序和共享库）中嵌入静态探测器点。无论应用程序或库在什么位置（在开发环境下或在生产环境下）运行，都可以启用这些探测器。所定义探测器的语义应易于被 DTrace 用户群理解。例如，对于与提交请求的客户机对应的 Web 服务器和响应该请求的 Web 服务器，可以定义 query-receive 和 query-respond 探测器。这些示例探测器易于被大多数 DTrace 用户理解，它们对应于应用程序的最高级别的抽象，而不是较低级别的实现信息。DTrace 用户可以使用这些探测器来了解请求的时间分配。如果 query-receive 探测器将 URL 请求字符串显示为参数，则 DTrace 用户可以通过将此探测器与 io 提供器结合使用，确定哪些请求产生的磁盘 I/O 最多。

在选择探测器名称和位置时还应考虑所描述的抽象对象的稳定性。在应用程序的未来发行版中，是否仍将保留此探测器，即使实现发生变化。此探测器是在所有系统体系结构中都有意义，还是特定于特殊的指令集？本章将详细讨论这些决定如何指导您完成静态跟踪定义。

向应用程序中添加探测器

库和可执行文件的 DTrace 探测器在相应应用程序的二进制文件的 ELF 部分中定义。本节介绍如何定义探测器，如何将探测器添加到应用程序源代码中，以及如何扩充应用程序的生成过程以包含 DTrace 探测器的定义。

定义提供器和探测器

在 .d 源文件中定义 DTrace 探测器，该源文件将在编译和链接应用程序时使用。首先，选择用户应用程序提供器的正确名称。对于正在执行应用程序代码的每个进程，会在进程标识符前附加所选择的提供器名称。例如，如果为以进程 ID 1203 执行的 Web 服务器选择提供器名称 myserv，则与此进程对应的 DTrace 提供器名称应为 myserv1203。在 .d 源文件中，添加与以下示例类似的提供器定义：

```
provider myserv {

    ...

};
```

接下来，为每个探测器和相应参数添加定义。以下示例定义第 427 页中的 “选择探测器位置”中讨论的两个探测器。第一个探测器有两个参数，都为 string 类型，第二个探测器没有参数。D 编译器将任何探测器名称中的两个连续下划线 (--) 转换为一个连字符 (-)。

```
provider myserv {

    probe query__receive(string, string);

    probe query__respond();

};
```

应该向提供器定义中添加稳定性属性，以便探测器的使用者了解在应用程序的将来版本中发生变化的可能性。有关 DTrace 稳定性属性的更多信息，请参见第 39 章。稳定性属性的定义如下例所示：

示例 34–1 myserv.d: 静态定义的应用程序探测器

```
#pragma D attributes Evolving/Evolving/Common provider myserv provider

#pragma D attributes Private/Private/Unknown provider myserv module

#pragma D attributes Private/Private/Unknown provider myserv function

#pragma D attributes Evolving/Evolving/Common provider myserv name

#pragma D attributes Evolving/Evolving/Common provider myserv args



provider myserv {
```

示例 34–1 myserv.d:静态定义的应用程序探测器 （续）

```
    probe query__receive(string, string);

    probe query__respond();

};
```

向应用程序代码中添加探测器

至此，已在 .d 文件中定义了探测器，接下来需要扩充源代码以指示应触发探测器的位置。以下面的 C 应用程序源代码为例：

```
void

main_look(void)

{

    ...

    query = wait_for_new_query();

    process_query(query)

    ...

}
```

要添加探测器位置，请添加对 <sys/sdt.h> 中定义的 DTRACE_PROBE() 宏的引用，如下例所示：

```
#include <sys/sdt.h>

...



void

main_look(void)

{

    ...
```

```
    query = wait_for_new_query();

    DTRACE_PROBE2(myserv, query__receive, query->clientname, query->msg);

    process_query(query)

    ...

}
```

宏名称 DTRACE_PROBE2 中的后缀 2 表示将传递到探测器的参数个数。探测器宏的前两个参数是提供器名称和探测器名称，它们必须与 D 提供器和探测器定义对应。其余的宏参数是在触发探测器时将赋给 DTrace `arg0..9` 变量的参数。应用程序源代码可以包含对同一个提供器和探测器名称的多次引用。如果源代码中存在对同一个探测器的多次引用，则任何宏引用都将导致触发探测器。

生成包含探测器的应用程序

您必须扩充应用程序的生成过程以包含 DTrace 提供器和探测器定义。通常的生成过程会获取每个源文件，然后编译这些文件以创建相应的对象文件。然后再将编译后的对象文件链接在一起以创建最终的应用程序二进制文件，如下例所示：

```
cc -c src1.c

cc -c src2.c

...

cc -o myserv src1.o src2.o ...
```

要在应用程序中包含 DTrace 探测器定义，请向执行 `dtrace` 命令的生成过程中添加相应的 make 程序的描述文件规则，如下例所示：

```
cc -c src1.c

cc -c src2.c

...

dtrace -G -32 -s myserv.d src1.o src2.o ...

cc -o myserv myserv.o src1.o src2.o ...
```

如上所示的 dtrace 命令将对前面的编译器命令生成的对象文件进行后期处理，并生成对象文件 myserv.o（通过 myserv.d）和其他对象文件。dtrace -G 选项用于将提供器和探测器定义与用户应用程序链接起来。-32 选项用于生成 32 位应用程序二进制文件。-64 选项用于生成 64 位应用程序二进制文件。

◆ ◆ ◆ 第 35 章

35 安全性

本章介绍系统管理员可用于向特定用户或进程授予 DTrace 访问权的权限。使用 DTrace，可以查看系统的所有方面，包括用户级函数、系统调用、内核函数等等。它允许执行一些功能强大的操作，其中的有些操作可以修改程序的状态。就像不应允许一个用户访问另一个用户的专用文件一样，对于 DTrace 提供的所有工具的完全访问权限，系统管理员也不应将其授予所有用户。缺省情况下，只有超级用户可以使用 DTrace。最低权限工具可用于控制其他用户对 DTrace 的使用。

权限

使用 Solaris 最低权限工具，管理员可以为特定 Solaris 用户授予特定权限。要为用户指定登录时的权限，请在 `/etc/user_attr` 文件中插入一行代码，格式如下：

user-name`::::defaultpriv=basic,`*privilege*

要为正在运行的进程指定其他权限，请使用 `ppriv`(1) 命令：

`# ppriv -s A+`*privilege process-ID*

控制用户访问 DTrace 功能的三种权限为 `dtrace_proc`、`dtrace_user` 和 `dtrace_kernel`。每一种权限都允许使用某一组 DTrace 提供器、操作和变量，并且每一种权限都对应特定类型的 DTrace 用法。 以下各节中详细介绍了这些权限模式。系统管理员应针对不同权限模式的可见性以及对性能的影响，认真权衡每个用户的需要。用户要使用任何 DTrace 功能，需至少拥有三种 DTrace 权限之一。

使用 DTrace 的权限

拥有三种 DTrace 权限中任何一种的用户都可以启用 dtrace 提供器提供的探测器（请参见第 17 章），并且可以使用以下操作和变量：

提供器	dtrace		
操作	exit	printf	tracemem
	discard	speculate	
	printa	trace	
变量	args	probemod	this
	epid	probename	timestamp
	id	probeprov	vtimestamp
	probefunc	self	
地址空间	无		

dtrace_proc 权限

dtrace_proc 权限允许使用 pid 和 fasttrap 提供器进行进程级跟踪。它也允许使用以下操作和变量：

提供器	pid		
操作	copyin	copyout	stop
	copyinstr	raise	ustack
变量	execname	pid	uregs
地址空间	用户		

Solaris 内核数据结构或进程对不拥有此权限的用户是完全不可见的。

拥有此权限的用户可以在其自己的进程中创建和启用探测器。如果用户还拥有 proc_owner 权限，则可以在任何进程中创建和启用探测器。dtrace_proc 权限用于关注用户进程的调试或性能分析的用户。此权限适合于开发者处理新的应用程序，或者工程人员尝试在生产环境中改进应用程序的性能。

注 – 拥有 dtrace_proc 和 proc_owner 权限的用户可以启用任何进程的任何 pid 探测器，但只可以在其权限集是用户自己权限集的子集的进程中创建探测器。有关完整的详细信息，请参阅最低权限文档。

dtrace_proc 权限允许访问的 DTrace 可以仅对用户拥有权限的那些进程强加性能惩罚。受检测的进程将对系统资源强加更大的负载，同样地，它可能会对总体系统性能产生一些小的影响。除了此总体负载的增加，此权限不允许使用影响被跟踪进程之外任何进程的性能的检测过程。因为此权限不会允许用户查看其他进程或内核本身，建议将此权限授予可能需要更好地了解自己进程的内部工作情况的所有用户。

dtrace_user 权限

dtrace_user 权限允许使用包含一些警告的 profile 和 syscall 提供器，以及使用以下操作和变量：

提供器	profile	syscall	fasttrap
操作	copyin	copyout	stop
	copyinstr	raise	ustack
变量	execname	pid	uregs
地址空间	用户		

dtrace_user 权限仅提供对用户已拥有相应权限的那些进程的可视功能；它不允许内核状态或活动可视。拥有此权限，用户可以启用 syscall 提供器，但已启用的探测器将仅在用户拥有权限的进程中激活。类似地，可以启用 profile 提供器，但已启用的探测器将仅在用户拥有权限的进程中激活，而不会在 Solaris 内核中激活。

此权限允许使用可能影响总体系统性能的检测过程（但仅允许查看特定进程）。syscall 提供器会对每个进程的每个系统调用产生一些小的性能影响。profile 提供器通过执行与实时计时器类似的每个时间间隔来影响总体系统性能。这些性能降低都不会严重限制系统的进度，但系统管理员应考虑授予用户此权限所牵涉到的内容。有关 syscall 和 profile 提供器的性能影响的介绍，请参阅第 21 章和第 19 章。

dtrace_kernel 权限

dtrace_kernel 权限允许用户在不属于自己的进程中使用除 pid 和 fasttrap 提供器之外的所有提供器。此权限还允许使用除内核破坏性操作 (breakpoint(), panic(), chill()) 之外的所有操作和变量。此权限允许内核和用户状态完全可见。dtrace_user 权限启用的功能是 dtrace_kernel 启用的功能的限制子集。

提供器	除上述限制外的所有提供器
操作	除破坏性操作之外的所有操作
变量	所有
地址空间	用户 内核

超级用户权限

拥有所有权限的用户可以使用所有提供器和所有操作（包括其他各类用户不能使用的内核破坏性操作）。

提供器	所有
操作	包括破坏性操作的所有操作
变量	所有
地址空间	用户 内核

◆ ◆ ◆ 第 36 章

匿名跟踪

本章介绍匿名跟踪，匿名跟踪是不与任何 DTrace 使用者关联的跟踪。 在没有可以运行的 DTrace 进程情况下才使用匿名跟踪。匿名跟踪通常用于允许设备驱动程序开发者调试和跟踪系统引导期间发生的活动。任何可以交互执行的跟踪都可以匿名执行。 但是，仅有超级用户才可以创建匿名启用，任何时候仅可以存在一个匿名启用。

匿名启用

要创建匿名启用，请将 `-A` 选项与指定需要的探测器、谓词、操作和选项的 `dtrace`(1M) 调用结合使用。`dtrace` 会将一系列代表您请求的驱动器属性添加到 `dtrace`(7D) 驱动器的配置文件（通常为 `/kernel/drv/dtrace.conf`）中。这些属性将由 `dtrace`(7D) 驱动程序在装入时读取。该驱动程序将使用指定的操作启用指定的探测器，并创建与新的启用关联的匿名状态。 通常，`dtrace`(7D) 驱动程序根据需要进行装入，这与充当 DTrace 提供器的任何驱动程序一样。 要允许在引导期间跟踪，必须尽早装入 `dtrace`(7D) 驱动程序。`dtrace` 针对每个必需的 DTrace 提供器和 `dtrace`(7D) 本身，将所需要 `forceload` 语句添加至 `/etc/system`（请参见 `system`(4)）中。

此后，当引导系统时，`dtrace`(7D) 将发出一条消息，指示已成功处理配置文件。

所有选项都可以使用匿名启用设置，包括缓冲区大小、动态变量大小、随机大小、随机数等。

要删除匿名启用，请将 `-A` 指定给不包含任何探测器说明的 `dtrace`。

声明匿名状态

在计算机完成引导之后，可以通过指定带有 `-a` 选项的 `dtrace` 来声明匿名状态。 缺省情况下，`-a` 会声明匿名状态、处理现有数据，然后继续运行。 要使用匿名状态，然后退出，请添加 `-e` 选项。

一旦在内核中使用匿名状态，将无法替换该状态：因为包含该状态的内核内部的缓冲区被重用。 如果尝试声明不存在的匿名跟踪状态，则 `dtrace` 将生成与以下示例类似的消息：

```
dtrace: could not enable tracing: No anonymous tracing state
```

如果发生删除或错误，dtrace 将在声明匿名状态时生成相应的消息。 对于匿名和非匿名状态，删除消息和错误消息相同。

匿名跟踪示例

以下示例说明了 iprb(7D) 模块中每个探测器的匿名 DTrace 启用：

```
# dtrace -A -m iprb

dtrace: saved anonymous enabling in /kernel/drv/dtrace.conf

dtrace: added forceload directives to /etc/system

dtrace: run update_drv(1M) or reboot to enable changes

# reboot
```

重新引导后，dtrace(7D) 将在控制台上列显一条消息，指示正在启用指定的探测器：

```
  ...

  Copyright 1983-2003 Sun Microsystems, Inc.  All rights reserved.

  Use is subject to license terms.

  NOTICE: enabling probe 0 (:iprb::)

  NOTICE: enabling probe 1 (dtrace:::ERROR)

  configuring IPv4 interfaces: iprb0.

  ...
```

重新引导计算机后，可以通过指定带有 -a 选项的 dtrace 来使用匿名状态：

```
# dtrace -a

  CPU     ID                    FUNCTION:NAME

    0  22954                      _init:entry

    0  22955                     _init:return
```

```
    0  22800                  iprbprobe:entry

    0  22934           iprb_get_dev_type:entry

    0  22935          iprb_get_dev_type:return

    0  22801                  iprbprobe:return

    0  22802                  iprbattach:entry

    0  22874                iprb_getprop:entry

    0  22875               iprb_getprop:return

    0  22934           iprb_get_dev_type:entry

    0  22935          iprb_get_dev_type:return

    0  22870              iprb_self_test:entry

    0  22871             iprb_self_test:return

    0  22958             iprb_hard_reset:entry

    0  22959            iprb_hard_reset:return

    0  22862       iprb_get_eeprom_size:entry

    0  22826               iprb_shiftout:entry

    0  22828             iprb_raiseclock:entry

    0  22829            iprb_raiseclock:return

  ...
```

以下示例仅适用于通过 `iprbattach()` 调用的那些函数。在编辑器中，键入以下脚本并将其保存在名为 `iprb.d` 的文件中。

```
fbt::iprbattach:entry

{

    self->trace = 1;

}
```

```
fbt:::
/self->trace/
{}


fbt::iprbattach:return
{
    self->trace = 0;
}
```

运行以下命令清除驱动器配置文件中先前的设置，安装新的匿名跟踪请求，然后重新引导：

```
# dtrace -AFs iprb.d
dtrace: cleaned up old anonymous enabling in /kernel/drv/dtrace.conf
dtrace: cleaned up forceload directives in /etc/system
dtrace: saved anonymous enabling in /kernel/drv/dtrace.conf
dtrace: added forceload directives to /etc/system
dtrace: run update_drv(1M) or reboot to enable changes
# reboot
```

重新引导后，dtrace(7D) 将在控制台上列显一条不同的消息，指示稍微不同的启用项：

```
  ...
  Copyright 1983-2003 Sun Microsystems, Inc.  All rights reserved.
  Use is subject to license terms.
  NOTICE: enabling probe 0 (fbt::iprbattach:entry)
  NOTICE: enabling probe 1 (fbt:::)
```

```
  NOTICE: enabling probe 2 (fbt::iprbattach:return)
  NOTICE: enabling probe 3 (dtrace:::ERROR)
  configuring IPv4 interfaces: iprb0.
  ...
```

计算机完成引导后，运行带有 `-a` 选项和 `-e` 选项的 `dtrace` 以使用匿名数据，然后退出。

```
# dtrace -ae
  CPU FUNCTION
    0  -> iprbattach
    0    -> gld_mac_alloc
    0      -> kmem_zalloc
    0        -> kmem_cache_alloc
    0          -> kmem_cache_alloc_debug
    0            -> verify_and_copy_pattern
    0            <- verify_and_copy_pattern
    0            -> tsc_gethrtime
    0            <- tsc_gethrtime
    0            -> getpcstack
    0            <- getpcstack
    0            -> kmem_log_enter
    0            <- kmem_log_enter
    0          <- kmem_cache_alloc_debug
    0        <- kmem_cache_alloc
    0      <- kmem_zalloc
    0    <- gld_mac_alloc
```

```
0    -> kmem_zalloc
0      -> kmem_alloc
0        -> vmem_alloc
0          -> highbit
0          <- highbit
0          -> lowbit
0          <- lowbit
0          -> vmem_xalloc
0            -> highbit
0            <- highbit
0            -> lowbit
0            <- lowbit
0            -> segkmem_alloc
0              -> segkmem_xalloc
0                -> vmem_alloc
0                  -> highbit
0                  <- highbit
0                  -> lowbit
0                  <- lowbit
0                  -> vmem_seg_alloc
0                    -> highbit
0                    <- highbit
0                    -> highbit
0                    <- highbit
```

```
  0                    -> vmem_seg_create

...
```

◆ ◆ ◆ 第 37 章

37 事后跟踪

本章介绍用于事后提取以及处理 DTrace 使用者的内核中数据的 DTrace 工具。在发生系统崩溃时，使用 DTrace 记录的信息可能会提供导致系统故障的根本原因的重要线索。可以通过系统崩溃转储来提取和处理 DTrace 数据，以帮助您了解致命的系统故障。通过将 DTrace 的这些事后功能与其环形缓冲的缓冲区策略结合（请参见第 11 章），DTrace 可用作商用飞机上的黑匣子飞行数据记录器的操作系统模拟程序。

要从特定崩溃转储中提取 DTrace 数据，应首先在相关的崩溃转储中运行 Solaris 模块调试器 `mdb`(1)。包含 DTrace 功能的 MDB 模块将自动装入。要了解有关 MDB 的更多信息，请参阅《Solaris 模块调试器指南》。

显示 DTrace 使用者

要从 DTrace 使用者中提取 DTrace 数据，必须首先运行 `::dtrace_state` MDB dcmd 来确定相关的 DTrace 使用者：

```
> ::dtrace_state

    ADDR MINOR     PROC NAME                 FILE

ccaba400     2        - <anonymous>             -

ccab9d80     3 d1d6d7e0 intrstat         cda37078

cbfb56c0     4 d71377f0 dtrace           ceb51bd0

ccabb100     5 d713b0c0 lockstat         ceb51b60

d7ac97c0     6 d713b7e8 dtrace           ceb51ab8
```

此命令显示 DTrace 状态结构表。表的每一行由以下信息组成：

- 状态结构的地址

- 与 dtrace(7D) 设备关联的最小编号
- 对应于 DTrace 使用者的进程结构的地址
- DTrace 使用者的名称（对于匿名使用者，则为 <anonymous>）
- 对应于已打开 dtrace(7D) 设备的文件结构的名称

要获取有关特定 DTrace 使用者的详细信息，请将其进程结构的地址指定到 ::ps dcmd：

```
> d71377f0::ps

S    PID   PPID   PGID    SID    UID      FLAGS     ADDR NAME

R 100647 100642 100647 100638      0 0x00004008 d71377f0 dtrace
```

显示跟踪数据

确定相关的使用者之后，可以通过将状态结构的地址指定到 ::dtrace dcmd 来检索对应于任何未使用的缓冲区的数据。以下示例显示了在使用操作 trace(execname) 匿名启用 syscall:::entry 时，::dtrace dcmd 的输出：

```
> ::dtrace_state

    ADDR MINOR     PROC NAME                 FILE

cbfb7a40     2        - <anonymous>             -



> cbfb7a40::dtrace

CPU     ID                    FUNCTION:NAME

  0    344               resolvepath:entry   init

  0     16                     close:entry   init

  0    202                     xstat:entry   init

  0    202                     xstat:entry   init

  0     14                      open:entry   init

  0    206                    fxstat:entry   init

  0    186                      mmap:entry   init

  0    186                      mmap:entry   init
```

```
  0    186                      mmap:entry   init
  0    190                    munmap:entry   init
  0    344                resolvepath:entry   init
  0    216                    memcntl:entry   init
  0     16                     close:entry   init
  0    202                     xstat:entry   init
  0     14                      open:entry   init
  0    206                     fxstat:entry   init
  0    186                      mmap:entry   init
  0    186                      mmap:entry   init
  0    186                      mmap:entry   init
  0    190                    munmap:entry   init
...
```

::dtrace dcmd 采用与 dtrace(1M) 相同的方法处理错误：如果使用者正在执行时遇到删除、错误、随机删除或类似问题，::dtrace 将发出一条对应于 dtrace(1M) 消息的消息：

::dtrace 显示事件的顺序在给定 CPU 中始终为从最旧到最新。CPU 缓冲区本身按照数字顺序显示。如果需要对不同 CPU 中的事件进行排序，请跟踪 timestamp 变量。

可以通过将 -c 选项指定到 ::dtrace，来仅显示特定 CPU 的数据：

```
> cbfb7a40::dtrace -c 1
CPU     ID                    FUNCTION:NAME
  1     14                      open:entry   init
  1    206                     fxstat:entry   init
  1    186                      mmap:entry   init
  1    344                resolvepath:entry   init
  1     16                     close:entry   init
```

```
  1    202                      xstat:entry   init

  1    202                      xstat:entry   init

  1     14                       open:entry   init

  1    206                     fxstat:entry   init

  1    186                       mmap:entry   init

...
```

请注意，`::dtrace` 仅处理内核中的 DTrace 数据。通过内核使用并通过 `dtrace`(1M) 或其他方法处理的数据不可以使用 `::dtrace` 进行处理。要确保在出现故障时可以使用大多数的数据，请使用环形缓冲区的缓冲策略。有关缓冲区策略的更多信息，请参见第 11 章。

以下示例创建一个非常小 (16K) 的环形缓冲区，并记录所有系统调用以及使这些调用执行以下操作的进程：

```
# dtrace -P syscall'{trace(curpsinfo->pr_psargs)}' -b 16k -x bufpolicy=ring

dtrace: description 'syscall:::entry' matched 214 probes
```

查看运行上述命令时所采用的崩溃转储将生成与以下示例类似的输出：

```
> ::dtrace_state

    ADDR MINOR     PROC NAME                 FILE

cdccd400     3 d15e80a0 dtrace           ced065f0



> cdccd400::dtraceCPU     ID                    FUNCTION:NAME

  0    139                   getmsg:return   mibiisa -r -p 25216

  0    138                    getmsg:entry   mibiisa -r -p 25216

  0    139                   getmsg:return   mibiisa -r -p 25216

  0    138                    getmsg:entry   mibiisa -r -p 25216

  0    139                   getmsg:return   mibiisa -r -p 25216

  0    138                    getmsg:entry   mibiisa -r -p 25216
```

```
  0    139                   getmsg:return   mibiisa -r -p 25216
  0    138                    getmsg:entry   mibiisa -r -p 25216
  0    139                   getmsg:return   mibiisa -r -p 25216
  0    138                    getmsg:entry   mibiisa -r -p 25216
  0     17                    close:return   mibiisa -r -p 25216
...
  0     96                     ioctl:entry   mibiisa -r -p 25216
  0     97                    ioctl:return   mibiisa -r -p 25216
  0     96                     ioctl:entry   mibiisa -r -p 25216
  0     97                    ioctl:return   mibiisa -r -p 25216
  0     96                     ioctl:entry   mibiisa -r -p 25216
  0     97                    ioctl:return   mibiisa -r -p 25216
  0     96                     ioctl:entry   mibiisa -r -p 25216
  0     97                    ioctl:return   mibiisa -r -p 25216
  0     16                    close:entry    mibiisa -r -p 25216
  0     17                    close:return   mibiisa -r -p 25216
  0    124                  lwp_park:entry   mibiisa -r -p 25216
  1     68                    access:entry   mdb -kw
  1     69                   access:return   mdb -kw
  1    202                     xstat:entry   mdb -kw
  1    203                    xstat:return   mdb -kw
  1     14                      open:entry   mdb -kw
  1     15                     open:return   mdb -kw
  1    206                    fxstat:entry   mdb -kw
```

```
  1    207                   fxstat:return   mdb -kw
  1    186                      mmap:entry   mdb -kw
...
  1     13                    write:return   mdb -kw
  1     10                     read:entry    mdb -kw
  1     11                     read:return   mdb -kw
  1     12                    write:entry    mdb -kw
  1     13                    write:return   mdb -kw
  1     96                     ioctl:entry   mdb -kw
  1     97                     ioctl:return  mdb -kw
  1    364                   pread64:entry   mdb -kw
  1    365                   pread64:return  mdb -kw
  1    366                   pwrite64:entry  mdb -kw
  1    367                  pwrite64:return  mdb -kw
  1    364                    pread64:entry  mdb -kw
  1    365                   pread64:return  mdb -kw
  1     38                       brk:entry   mdb -kw
  1     39                      brk:return   mdb -kw
>
```

请注意，CPU 1 的最新记录包括由 `mdb -kw` 进程进行的一系列 `write`(2) 系统调用。此结果可能与系统故障的原因有关，因为在使用 `-k` 和 `-w` 选项运行时，用户可以使用 `mdb`(1) 修改正在运行的内核数据或文本。在此情况下，DTrace 数据至少提供了一种让人关注的调查方法（如果不是故障的根本原因）。

◆ ◆ ◆ 第 38 章

38 性能注意事项

由于 DTrace 会导致系统产生额外工作，因此启用 DTrace 始终会在某些方面影响系统性能。通常，这种影响可以忽略，但如果启用了许多会产生大量开销的探测器时，对系统的影响可能就会变得很严重。本章将介绍降低 DTrace 对性能影响的一些技术。

限制已启用的探测器

使用动态检测过程技术，DTrace 可对内核和任意用户进程提供非并行的跟踪范围。虽然此范围允许对系统行为进行彻底性的监测，但也可能会产生极大的探测影响。如果启用了大量的探测器，势必将对系统产生非常大的影响。因此，应根据解决问题时的实际需要来启用探测器。例如，在有更简明的启用项可以回答问题时，不应启用所有 FBT 探测器。例如，您可能只需要集中于所关注的特定模块或特定函数便可解决问题。

使用 `pid` 提供器时，应特别小心。因为 `pid` 提供器可以检测每条指令，而您可能会在应用程序中启用数百万个探测器，从而使目标进程速度变得非常缓慢。

在必须为要回答的问题启用大量探测器的情况下，也可以使用 DTrace。启用大量的探测器可能会大大降低系统速度，但不会导致计算机出现致命故障。因此，应放心启用所需数量的探测器。

使用聚合

如第 9 章中所讨论，DTrace 聚合允许以一种可伸缩的方式聚合数据。关联数组也可能会提供与聚合类似的功能。但是，作为全局通用变量的特性决定了它们无法提供可线性伸缩的聚合。因此，如有可能，应优先使用聚合而不是关联数组。建议您不要使用以下示例：

```
syscall:::entry

{

    totals[execname]++;
```

```
}


syscall::rexit:entry
{
    printf("%40s %d\n", execname, totals[execname]);
    totals[execname] = 0;
}
```

首选使用以下示例：

```
syscall:::entry
{
    @totals[execname] = count();
}


END
{
    printa("%40s %@d\n", @totals);
}
```

使用可高速缓存的谓词

DTrace 谓词用于过滤实验脚本中不需要的数据，具体方法是仅跟踪指定的条件为 true 时的数据。启用许多探测器时，通常可使用标识特定线程或所关注线程（如 `/self->traceme/` 或 `/pid == 12345/`）形式的谓词。尽管对于多数探测器中的多数线程而言，许多谓词的计算结果为 false 值，但对成千上万个探测器进行这样的计算时，计算本身的开销也可能变得很大。为了降低此开销，如果谓词仅包含线程局部变量（例如，`/self->traceme/`）或不变变量（例如，`/pid == 12345/`），DTrace 将对谓词计算进行高速缓存。计算已高速缓存的谓词的开销要远远小于计算未高速缓存的谓词的开销，在谓词包含线程局部变量、字符串比较或其他开销相对较大的操作时，尤其如此。虽然谓词高速缓存对于用户是透明的，但它隐含了一些构造最优谓词的原则，如下表所示：

可高速缓存	不可高速缓存
self->mumble	mumble[curthread], mumble[pid, tid]
execname	curpsinfo->pr_fname, curthread->t_procp->p_user.u_comm
pid	curpsinfo->pr_pid, curthread->t_procp->p_pipd->pid_id
tid	curlwpsinfo->pr_lwpid, curthread->t_tid
curthread	curthread->*any member*, curlwpsinfo->*any member*, curpsinfo->*any member*

建议您不要使用以下示例：

```
syscall::read:entry

{

    follow[pid, tid] = 1;

}



fbt:::

/follow[pid, tid]/

{}



syscall::read:return

/follow[pid, tid]/

{

    follow[pid, tid] = 0;

}
```

首选使用线程局部变量的以下示例：

```
syscall::read:entry

{
```

```
    self->follow = 1;
}

fbt:::
/self->follow/
{}

syscall::read:return
/self->follow/
{
    self->follow = 0;
}
```

为进行高速缓存，谓词必须完全由可高速缓存的表达式组成。以下谓词均可进行高速缓存：

```
/execname == "myprogram"/
/execname == $$1/
/pid == 12345/
/pid == $1/
/self->traceme == 1/
```

以下示例使用了全局变量，因而无法高速缓存：

```
/execname == one_to_watch/
/traceme[execname]/
/pid == pid_i_care_about/
/self->traceme == my_global/
```

◆ ◆ ◆ 第 39 章

39 稳定性

Sun 通常会向开发者提前提供一些新技术及观察工具，从而使用户可以深入了解用户软件及内核软件的内部实现详细信息。但是，在软件进行升级或修补后，随着接口和实现的演变及成熟，新技术和内部实现详细信息也会随之发生变化。Sun 使用一组标签（在 `attributes(5)` 手册页中说明）来记录应用程序和接口的稳定性级别，以帮助用户设置对各种将来发行版中可能出现的更改类型的期望值。

没有一种稳定性属性可正确描述通过 D 程序访问的任意实体和服务集。因此，在 DTrace 和 D 编译器中，包含了可动态计算并描述所创建的 D 程序稳定性级别的功能。本章介绍用于确定程序稳定性的 DTrace 功能，以帮助您设计稳定的 D 程序。可以使用 DTrace 稳定性功能来通知您有关 D 程序的稳定性属性，或在程序具有不需要的接口相关性时生成编译时错误。

稳定性级别

DTrace 为内置变量、函数和探测器等实体提供了两类稳定性属性：稳定性级别和体系结构相关性类。其中，DTrace 稳定性级别通过指示在将来的发行版或修补程序中更改接口或 DTrace 实体的可能性，帮助您在基于 DTrace 开发脚本和工具时进行风险评估。DTrace 相关性类则向您指示某接口是否为所有 Solaris 平台和处理器公用，或该接口是否仅与特定的体系结构（如 SPARC 处理器）关联。用于描述接口的这两类属性可以独立变化。

下表按稳定性从低到高的顺序显示了 DTrace 使用的稳定性值。由于 Sun 将尽力确保在将来的次发行版中继续使用这些接口，因此可供所有 D 程序和分层应用程序使用的接口将更稳定。在将来的次发行版中，仅依赖稳定接口的应用程序将继续可靠地正常工作，并且不会被临时的修补程序破坏。使用稳定性较低的接口，可在当前系统上进行实验、设计原型、调节和调试，但是，使用时应认识到这些接口可能会出现不兼容的情况，甚至在将来的次发行版中可能被删除或替换。

另外，DTrace 稳定性值还有助于您了解所观察的软件实体的稳定性，以及 DTrace 接口本身的稳定性。因此，在升级或更改所观察的软件栈时，通过 DTrace 稳定性值，还可指示 D 程序和分层工具需要进行相应更改的可能性。

内部	该接口为 DTrace 专用，代表 DTrace 的实现详细信息。 在次发行版或微发行版中，内部接口可能会有所变化。
专用	该接口为 Sun 专用，代表为供其他 Sun 产品（尚未公开发布以供用户和 ISV 使用的产品）使用而开发的接口。 在次发行版或微发行版中，专用接口可能会有所变化。
过时	当前发行版支持该接口，但计划在将来的次发行版中将其删除。 停止接口支持之前，Sun 将会首先尝试提供通知，然后再停止相应接口。 如果您试图使用已过时的接口，D 编译器可能会生成警告消息。
外部	该接口受 Sun 以外的实体控制。 根据 Sun 的自行判断，Sun 可将此类接口的更新版本和可能不兼容的版本作为任何发行版的一部分提供，这主要受其在控制实体中的可用性限制。 Sun 没有就任何两个发行版之间的外部接口的源或二进制兼容性进行任何声明。 在将来的发行版中，基于这些接口的应用程序（包括包含外部接口的修补程序）可能无法运行。
不稳定	提供该接口的目的是为了使开发者能够预先体验新兴的、快速变化的技术，或预先体验对于观察或调试系统行为而言非常关键的实现产物，以便将来可采用一种更稳定的解决方案。 Sun 没有就次发行版之间的不稳定接口的源或二进制兼容性进行任何声明。
发展中	该接口最终可能会成为标准接口或稳定接口，但现在仍处于转换状态。 Sun 将会竭尽全力确保其在演变过程中与以前的发行版保持兼容。 如果必须进行非向上兼容更改，这些更改将会在次发行版和主发行版完成。 如有可能，应尽量避免在微发行版中进行这些更改。 如果必须进行此类更改，应在受影响的发行版的发行说明中予以记录，并且 Sun 将在可行的情况下，为二进制兼容性和后续的 D 程序开发提供迁移帮助。
稳定	该接口是一个受 Sun 控制的成熟接口。 Sun 将尽力避免对这类接口进行非向上兼容更改，尤其是在次发行版或微发行版中。 如果必须停止对稳定接口的支持，Sun 将会尝试提供通知并将稳定性级别更改为“过时”。
标准	该接口符合行业标准。 与该接口对应的文档将对该接口所符合的标准进行描述。 这些标准通常由标准开发组织进行控制，可以依据对标准已批准的更改对接口进行更改。 该稳定性级别还适用于已采用（但缺少行业约定的正式标准）的接口。 仅对指定版本的标准提供支持，并不保证对更高版本提供支持。 如果标准开发组织同意对 Sun 决定支持的标准接口进行非向上兼容更改，Sun 将公布一个兼容性和迁移策略。

相关性类

由于 Solaris 和 DTrace 支持各种平台和处理器，因此 DTrace 还为接口设置了相关性类标签，用于指示接口是否为所有 Solaris 平台和处理器公用，或该接口是否与特定的系统体系结构关联。 相关性类与以前描述的稳定性级别互不相关。 例如，DTrace 接口可以为“稳定”，但仅在 SPARC 微处理器中受支持，或者为“不稳定”，但为所有 Solaris 系统公用。 在以下列表中，按最不常用（即完全特定于特定的体系结构）至最常用（即为所有体系结构公用）的顺序，介绍了各种 DTrace 相关性类。

未知
: 该接口具有未知的体系结构相关性集合。 DTrace 不必了解所有实体（如操作系统实现中定义的数据类型）的体系结构相关性。 未知标签通常适用于无法计算相关性且稳定性极低的接口。 在当前使用的体系结构以外的任何体系结构上使用 DTrace 时，可能无法使用该接口。

CPU
: 该接口特定于当前系统的 CPU 型号。 您可以使用 `psrinfo`(1M) 实用程序的 `-v` 选项，显示当前的 CPU 型号和实现名称。 具有 CPU 型号相关性的接口在其他 CPU 实现中可能不可用，即使这些 CPU 导出的指令集体系结构 (instruction set architecture, ISA) 相同。 例如，尽管 UltraSPARC-III+ 微处理器和 UltraSPARC-II 微处理器都支持 SPARC 指令集，但前者的 CPU 相关接口在后者中可能不可用。

平台
: 该接口特定于当前系统的硬件平台。 通常，平台将一组系统组件及体系结构特征（如一组支持的 CPU 型号）与系统名（如 `SUNW,Ultra-Enterprise-10000`）关联。 您可以使用 `uname`(1) `-i` 选项显示当前平台名称。 该接口在其他硬件平台上可能不可用。

组
: 该接口特定于当前系统的硬件平台组。 通常，平台组使用 `sun4u` 之类的单一名称，将一组平台与相关特征关联在一起。 您可以使用 `uname`(1) `-m` 选项显示当前平台组名。 该接口在平台组中的其他平台上可用，但在不属于该组成员的硬件平台上可能不可用。

ISA
: 该接口特定于此系统中的微处理器支持的指令集体系结构 (ISA)。 ISA 介绍了可在微处理器上执行的软件的规范，包括汇编语言指令和寄存器等详细信息。 您可以使用 `isainfo`(1) 实用程序，显示系统支持的本机指令集。 在不会导出任何相同指令集的系统上，可能不支持该接口。 例如，在 Solaris x86 系统上，可能不支持 Solaris SPARC 系统中的 ISA 相关接口。

公用
: 无论底层硬件如何，该接口均为所有 Solaris 系统公用。 在具有相同 Solaris 和 DTrace 修订的其他 Solaris 系统上，可以执行和部署仅依赖公用接口的 DTrace 程序和分层应用程序。 大多数 DTrace 接口均为公用接口，因此可在使用 Solaris 的任何地方使用它们。

接口属性

DTrace 使用由两个稳定性级别和一个相关性类组成的三重属性来描述接口。 根据约定，接口属性按以下顺序列出，并用斜杠分隔：

name-stability / *data-stability* / *dependency-class*

当接口名称显示在 D 程序中或 dtrace(1M) 命令行上时，接口的 *name stability* 描述与其名称关联的稳定性级别。例如，execname D 变量是一个“稳定”名称：根据以上介绍的稳定接口规则，Sun 保证在 D 程序中继续支持该标识符。

接口的 *data stability* 不同于与接口名称关联的稳定性。 此稳定性级别说明，Sun 承诺维护接口以及任何关联的数据语义使用的数据格式。 例如，pid D 变量是一个“稳定”接口：在 Solaris 中，进程 ID 是一个“稳定”概念。Sun 保证 pid 变量的类型将为 pid_t，并具有如下语义：根据稳定接口规则，将其设置为与触发给定探测器的线程对应的进程 ID。

接口的 *dependency class* 与其名称和数据稳定性均不同，用于说明接口是否特定于当前操作平台或微处理器。

我们将很快看到，DTrace 和 D 编译器可跟踪所有 DTrace 接口实体的稳定性属性，包括提供器、探测器说明、D 变量、D 函数、类型和程序语句本身。 请注意，这三个值均能独立变化。例如，curthread D 变量具有“稳定/专用/公用”等属性：变量名称为“稳定”并且为所有 Solaris 操作平台“公用”，但该变量提供对作为 Solaris 内核实现产物的“专用”数据格式的访问。与所定义的变量相同，大多数 D 变量都提供“稳定/稳定/公用”等属性。

稳定性计算和报告

D 编译器可对 D 程序中的所有探测器说明和操作语句进行稳定性计算。 您可以使用 dtrace -v 选项显示程序的稳定性报告。 以下示例使用在命令行上编写的程序：

```
# dtrace -v -n dtrace:::BEGIN'{exit(0);}'

dtrace: description 'dtrace:::BEGIN' matched 1 probe

Stability data for description dtrace:::BEGIN:

        Minimum probe description attributes

                Identifier Names: Evolving

                Data Semantics:   Evolving

                Dependency Class: Common
```

```
        Minimum probe statement attributes

                Identifier Names: Stable

                Data Semantics:   Stable

                Dependency Class: Common

CPU     ID                    FUNCTION:NAME

  0      1                           :BEGIN
```

您可能还希望将 dtrace -v 选项与 -e 选项进行组合，以指示 dtrace 编译但不执行 D 程序，从而在不必启用任何探测器并执行程序的情况下确定程序稳定性。以下为另一个稳定性报告示例：

```
# dtrace -ev -n dtrace:::BEGIN'{trace(curthread->t_procp);}'

Stability data for description dtrace:::BEGIN:

        Minimum probe description attributes

                Identifier Names: Evolving

                Data Semantics:   Evolving

                Dependency Class: Common

        Minimum probe statement attributes

                Identifier Names: Stable

                Data Semantics:   Private

                Dependency Class: Common

#
```

请注意，在新程序中，我们引用了 D 变量 curthread，该变量具有“稳定”名称和“专用”数据语义（即如果您查看该语义，将访问内核的专用实现详细信息），并且此状态现在显示在程序的稳定性报告中。在程序报告中，通过从每个接口的三重属性的对应值中，选择最低稳定性级别和类计算稳定性属性。

对于探测器说明，可根据提供器发布的属性，通过为所有指定的探测器说明字段选择最低稳定性属性来计算稳定性属性。在与各提供器对应的章节中，显示了可用的 DTrace 提供器

的属性。 对于 DTrace 提供器发布的所有探测器，该提供器可导出四个说明字段中每一个字段的三重稳定性属性。 因此，提供器名称的稳定性可能高于所导出的各个探测器的稳定性。 例如，探测器说明：

```
fbt:::
```

指示 DTrace 应跟踪项并从所有内核函数返回，其稳定性高于以下探测器说明：

```
fbt:foo:bar:entry
```

该说明指定了内核模块 `foo` 中的特定内部函数 `bar()`。 为简单起见，大多数提供器都使用一组属性来描述所有单个 *module*:*function*:*name* 的值。 另外，提供器还会为 `args[]` 数组指定属性，因为所有探测器参数的稳定性都视提供器而定。

如果探测器说明中没有指定提供器字段，则为其指定“不稳定/不稳定/公用”的稳定性属性，因为该说明可能无法与在 Solaris 将来版本中使用时尚不存在的提供器的探测器匹配。 同样，Sun 无法保证该程序将来的稳定性和行为。 编写 D 程序子句时，应始终显式指定提供器。 另外，包含模式匹配字符（参见第 4 章），或 `$1` 等宏变量（参见第 15 章）的任何探测器说明字段，将被视为未指定，因为这些说明模式可能会进行扩展，以便与 Sun 在将来版本的 DTrace 和 Solaris OS 中发布的提供器或探测器匹配。

对于大多数 D 语言语句，通过在语句中为实体选择最低稳定性和类来计算稳定性属性。例如，下列 D 语言实体具有如下属性：

实体	属性
内置的 D 变量 `curthread`	稳定/专用/公用
用户定义的 D 变量 `x`	稳定/稳定/公用

如果编写以下 D 程序语句：

```
x += curthread->t_pri;
```

则产生的语句属性为“稳定/专用/公用”，该最低属性与操作数 `curthread` 和 `x` 关联。 表达式的稳定性通过为各操作数选择最低稳定性属性进行计算。

会自动为程序中定义的任何 D 变量指定“稳定/稳定/公用”属性。 另外，会为 D 语言语法和 D 运算符隐式指定“稳定/稳定/公用”属性。对于使用反引号 (‘) 运算符引用内核符号，将始终为其指定“专用/专用/未知”属性，因为它们反映实现产物。 在 D 程序源代码中定义的类型，尤其是与 C 和 D 类型名称空间关联的类型，将为其指定“稳定/稳定/公用”属性。 在操作系统实现中定义的类型，以及其他类型的名称空间提供的类型，将为其指定“专用/专用/未知”属性。 D 类型转换运算符生成的表达式的稳定性属性，为输入表达式属性和转换输出类型属性之间的较小者。

如果您使用 C 预处理程序来纳入 C 系统头文件，这些类型将与 C 类型名称空间关联，并会为其指定“稳定/稳定/公用”属性，因为 D 编译器除了假定您会负责这些声明外，没有其他选择。 因此，如果您使用 C 预处理程序来纳入包含实现产物的头文件，则可能会在程序稳定性方面误导您自己。为确定合适的稳定性级别，应始终参阅与要纳入的头文件对应的文档。

稳定性执行

开发 DTrace 脚本或分层工具时，您可能希望确定稳定性问题的具体来源，或确保程序具有所需的稳定性属性集。 您可以使用 `dtrace -x amin=`*attributes* 选项，强制 D 编译器在任何属性计算产生的三重属性小于在命令行上指定的最小值时生成错误。以下示例使用 D 程序源代码片段说明了 `-x amin` 的用法。 请注意，使用由 `/` 分隔的三个标签按正常顺序指定各属性。

```
# dtrace -x amin=Evolving/Evolving/Common \
    -ev -n dtrace:::BEGIN'{trace(curthread->t_procp);}'
dtrace: invalid probe specifier dtrace:::BEGIN{trace(curthread->t_procp);}: \
    in action list: attributes for scalar curthread (Stable/Private/Common) \
    are less than predefined minimum
#
```

◆ ◆ ◆ 第 40 章

40 转换器

在第 39 章中，我们学习了有关 DTrace 如何计算和报告程序稳定性属性的内容。理论上，我们想仅使用“稳定”或“发展中”的接口来构造 DTrace 程序。遗憾的是，在调试低级问题或度量系统性能时，可能需要启用与操作系统内部例程（如内核中的函数）关联的探测器，而不是与更稳定的接口（如系统调用）关联的探测器。在软件栈深层的探测位置获取的数据通常是实现产物的集合，而不是更稳定的数据结构（例如，与 Solaris 系统调用接口关联的数据结构）。为了帮助您编写稳定的 D 程序，DTrace 提供了用于将实现产物转换为可通过 D 程序语句访问的稳定数据结构的工具。

转换器声明

转换器是由接口供应商提供的 D 赋值语句的集合，可用于将输入表达式转换为结构类型的对象。要了解转换器的必要性和用法，我们将以 `stdio.h` 中定义的 ANSI-C 标准库例程为例。这些例程作用于名为 `FILE` 的数据结构，该数据结构的实现产物是从 C 程序编制器中抽象出来的。创建数据结构抽象的标准方法是：在单独的专用头文件中保留相应的结构定义的同时，在公共头文件中仅提供数据结构的转发声明。

如果要编写 C 程序并且希望知道对应于 `FILE` 结构的文件描述符，则可以使用 `fileno`(3C) 函数获取该描述符，而不是直接取消对 FILE 结构成员的引用。Solaris 头文件通过将 FILE 定义为不透明的转发声明标记来强制实施此规则，这样包括 `<stdio.h>` 的 C 程序将不能直接取消对 FILE 结构成员的引用。在 `libc.so.1` 库中，可以假设按照以下形式使用 C 语言实现 `fileno()`：

```
int

fileno(FILE *fp)

{

    struct file_impl *ip = (struct file_impl *)fp;
```

```
    return (ip->fd);

}
```

假设 `fileno()` 接受 `FILE` 指针作为参数，并将其强制转换为指向相应内部 `libc` 结构 `struct file_impl` 的指针，然后返回该实现结构的 `fd` 成员的值。为什么 Solaris 按照此形式实现接口？ 通过在客户机程序中对当前 `libc` 实现的详细信息进行抽象，Sun 可以保持对强大二进制兼容性的承诺，同时继续发展和更改 `libc` 的内部实现详细信息。 在示例中，`fd` 成员可以更改 `struct file_impl`（甚至修补程序）中的大小和位置，调用 `fileno`(3C) 的现有二进制文件不会受此更改的影响，因为它们不依赖于这些人为因素。

不过，观察软件（如 DTrace）必须能够全面监测实现的内部以提供有用的结果，但它不具有调用在 Solaris 库或内核中定义的任意 C 函数的能力。 可以在 D 程序中声明 `struct file_impl` 的副本以便检测 `stdio.h` 中声明的例程，但您的 D 程序将依赖于库的“专用”实现产物，该库可能在将来的微发行版或次发行版甚至修补程序中中断。理论上，我们要提供绑定到库的实现并进行相应更新的结构，以供 D 程序使用，并且能提供与更好的稳定性关联的其他抽象层。

使用以下形式的声明创建新的转换器：

```
translator output-type < input-type input-identifier > {

    member-name = expression ;

    member-name = expression ;

    ...

};
```

output-type 指定一个结构，这将是转换的结果类型。 *input-type* 指定输入表达式的类型，并括在尖括号 `< >` 中，后跟可在转换器表达式中用作输入表达式的别名的 *input-identifier*。转换器的主体括在花括号 `{ }` 中，以分号 (`;`) 结尾，由标识相应转换表达式的 *member-name* 列表组成。 每个成员声明必须命名一个唯一的 *output-type* 成员，且必须根据 D 赋值 (`=`) 运算符的规则指定与成员类型兼容的类型表达式。

例如，可以根据一些可用的 `libc` 接口，定义一个有关 `stdio` 文件的稳定信息结构。

```
struct file_info {

    int file_fd;   /* file descriptor from fileno(3C) */

    int file_eof;  /* eof flag from feof(3C) */

};
```

然后可以在 D 中声明从 FILE 到 file_info 的假设 D 转换器，如下所示：

```
translator struct file_info < FILE *F > {

    file_fd = ((struct file_impl *)F)->fd;

    file_eof = ((struct file_impl *)F)->eof;

};
```

在假设的转换器中，输入表达式为 FILE * 类型，且指定了 *input-identifier* F。 然后可以在转换器成员表达式中将标识符 F 用作 FILE *（仅在转换器声明的主体中可见）类型的变量。 要确定 file_fd 输出成员的值，转换器将按照上面所示的 fileno(3C) 假设实现的类似方式，执行强制类型转换和取消引用。 还将执行类似的转换以获取 EOF 指示符的值。

Sun 提供了一组可与 Solaris 接口一起使用的转换器（可在 D 程序中调用这些转换器），并承诺，当相应接口的实现发生变化时，将根据先前定义的接口稳定性规则维护这些转换器。 在本章的后面部分中，我们将在学习如何通过 D 调用转换器之后，来学习这些转换器。对于想要提供自定义的转换器以供 D 程序员观察其软件包的状态的应用程序和库开发者，转换器工具本身也可供他们使用。

转换运算符

D 运算符 xlate 用于执行从输入表达式到所定义的某种转换输出结构的转换。xlate 运算符用于以下格式的表达式：

```
xlate < output-type > ( input-expression )
```

例如，要对上面定义的 FILE 结构调用假设的转换器并访问 file_fd 成员，可以编写以下表达式：

```
xlate <struct file_info *>(f)->file_fd;
```

其中，f 是 FILE * 类型的 D 变量。为 xlate 表达式本身指定 *output-type* 定义的类型。 定义转换器之后，可以使用该转换器将输入表达式转换为转换器输出结构类型或指向该结构的指针。

如果将输入表达式转换为结构，则可以使用 "." 运算符立即取消引用输出的特定成员，或者可以将整个已转换的结构赋给另一个 D 变量，以创建一份所有成员的值的副本。 如果取消引用单个成员，则 D 编译器将仅生成对应于该成员的表达式的代码。 不可以将 & 运算符应用于已转换的结构来获取其地址，因为在复制数据对象本身或引用它的某个成员之前，数据对象本身不存在。

如果将输入表达式转换为指向某结构的指针，则可以使用 -> 运算符立即取消引用输出的特定成员，或者可以使用一元 * 运算符取消引用该指针，在此情况下，产生的结果就像已经

将表达式转换为结构一样。如果取消引用单个成员，则 D 编译器将仅生成对应于该成员的表达式的代码。不可以将已转换的指针赋给另一个 D 变量，因为在复制对象或引用它的某个成员之前，数据对象本身不存在，所以无法对其寻址。

转换器声明可以省略输出类型的一个或多个成员的表达式。如果使用 `xlate` 表达式访问未定义转换表达式的成员，则 D 编译器将产生相应的错误消息，并中止程序编译。如果通过结构赋值的方式复制整个输出类型，则将使用零填充未定义转换表达式的任何成员。

为了找到 `xlate` 操作的匹配转换器，D 编译器将按以下顺序检查可用的转换器集：

- 首先，编译器查找从确切的输入表达式类型到确切的输出类型的转换。
- 其次，编译器将解析任何 typedef 别名后面的输入和输出类型为基础类型名称，然后查找从已解析的输入类型到已解析的输出类型的转换。
- 再次，编译器将查找从兼容的输入类型到已解析的输出类型的转换。为了确定输入表达式类型是否与转换器的输入类型兼容，编译器将使用确定函数调用参数是否与函数原型兼容的相同规则。

如果根据这些规则，未找到匹配的转换器，D 编译器将产生相应的错误消息，并且程序编译失败。

进程模型转换器

DTrace 库文件 `/usr/lib/dtrace/procfs.d` 提供了一组供在 D 程序中使用的转换器，使用这些转换器可以将进程和线程的操作系统内核实现结构转换为稳定的 `proc`(4) 结构（`psinfo` 和 `lwpsinfo`）。这些结构也可以在 Solaris `/proc` 文件系统文件（`/proc/`*pid*`/psinfo` 和 `/proc/`*pid*`/lwps/`*lwpid*`/lwpsinfo`）中使用，这些结构在系统头文件 `/usr/include/sys/procfs.h` 中定义。这些结构定义有关进程和线程的有用“稳定”信息（如由 `ps`(1) 命令显示的 ID、LWP ID、初始参数和其他数据）。有关结构成员和语义的完整说明，请参阅 `proc`(4)。

表 40–1 `procfs.d` 转换器

输入类型	输入类型属性	输出类型	输出类型属性
`proc_t *`	专用/专用/公用	`psinfo_t *`	稳定/稳定/公用
`kthread_t *`	专用/专用/公用	`lwpsinfo_t *`	稳定/稳定/公用

稳定转换

虽然转换器提供了将信息转换为稳定的数据结构的功能，但并不需要解决转换数据时可能产生的所有稳定性问题。例如，如果 xlate 操作本身的输入表达式引用“不稳定”的数据，则产生的 D 程序也将“不稳定”，因为程序稳定性的计算结果始终为累积的 D 程序语句和表达式的最低稳定性。所以，要允许构造稳定的程序，有时必须为转换器定义特定的稳定输入表达式。可以使用 D 内置机制来协助进行此类稳定转换。

DTrace procfs.d 库提供先前说明的 curlwpsinfo 和 curpsinfo 变量作为稳定转换。 例如，curlwpsinfo 变量实际上是按如下方式声明的 inline：

```
inline lwpsinfo_t *curlwpsinfo = xlate <lwpsinfo_t *> (curthread);

#pragma D attributes Stable/Stable/Common curlwpsinfo
```

curlwpsinfo 变量被定义为从 curthread 变量（指向表示线程的内核“专用”数据结构的指针）到“稳定” lwpsinfo_t 类型的内置转换。 D 编译器将处理此库文件，并高速缓存 inline 声明，从而使 curlwpsinfo 显示为任何其他 D 变量。 紧跟在声明后面的#pragma 语句用于将 curlwpsinfo 标识符的属性显式重置为“稳定/稳定/公用”，从而屏蔽对内置表达式中的 curthread 的引用。D 功能的这种组合允许 D 程序员安全地使用 curthread 作为转换源，Sun 可以同时将该转换源更新为 Solaris 实现中的相应更改。

◆ ◆ ◆ 第 41 章

版本控制

在第 39 章中，我们已了解了用于确定所创建的 D 程序的稳定性属性的 DTrace 功能。创建具有合适的稳定性属性的 D 程序后，您可能还希望将此程序与 D 编程接口的特定版本进行绑定。D 接口版本是一个标签，适用于 D 编译器提供的一组特定类型、变量、函数、常量和转换器。如果您指定到 D 编程接口特定版本的绑定，则可确保在 DTrace 将来版本上重新编译程序，而不会在定义的程序标识符和 D 编程接口将来版本中定义的标识符之间遇到冲突。您应为要作为持久性脚本（请参见第 15 章）安装，或在分层工具中使用的任何 D 程序都建立版本绑定。

版本和发行版

D 编译器使用版本字符串，标记与特定软件发行版对应的类型、变量、函数、常量和转换器的集合。版本字符串是一个由句点分隔的十进制整数序列，格式为 "*x*"（主发行版本号）、"*x.y*"（次发行版本号）或 "*x.y.z*"（微发行版本号）。版本是通过从左至右比较各整数来进行比较的。如果最左边的整数不相等，则包含较大整数的字符串的版本更高（因此更新）。如果最左边的整数相等，则按从左至右的顺序对下一个整数进行比较以确定结果。在版本字符串中，所有未指定的整数在版本比较过程中都被解释为零。

DTrace 版本字符串符合 Sun 的接口版本标准命名规则，如 `attributes`(5) 中所述。D 编程接口发生的更改将产生一个新的版本字符串。下表概述了 DTrace 使用的版本字符串，以及对应的 DTrace 软件发行版的可能含义。

表 41–1 DTrace 发行版版本

发行版	版本	含义
主	*x*.0	主发行版可能包含较多新增功能；符合各种可能不兼容的标准修订版；尽管不太可能，但可以更改、删除或替换标准接口或稳定接口（请参见第 39 章）。D 编程接口的初始版本被标记为 1.0 版。

表 41–1 DTrace 发行版版本 （续）

发行版	版本	含义
次	*x.y*	与 *x*.0 或早期版本（其中 y 不等于零）相比较，新的次发行版可能包含较少的新增功能、兼容的标准接口和稳定接口、可能不兼容的发展中接口或不稳定接口。这些更改可能包括新内置的 D 类型、变量、函数、常量和转换器。另外，次发行版可能会取消对以前标记为“过时”（参阅第 39 章）的接口的支持。
微	*x.y.z*	微发行版适用于与以前的发行版（其中 z 不等于零）兼容的接口，但可能会提供错误修复、性能增强功能以及对其他硬件的支持。

通常，D 编程接口的每个新版本都会提供一个以前版本提供的功能超集，已被删除的任何过时接口除外。

版本控制选项

缺省情况下，使用 `dtrace -s` 编译或使用 `dtrace -P`、`-m`、`-f`、`-n` 或 `-i` 命令行选项指定的任何 D 程序都会绑定到 D 编译器提供的 D 编程接口的最新版本。您可以使用 `dtrace -V` 选项确定 D 编程接口的当前版本：

```
$ dtrace -V

dtrace: Sun D 1.0

$
```

如果要与 D 编程接口的特定版本建立绑定，可将 `version` 选项设置为相应的版本字符串。与其他 DTrace 选项（参阅第 16 章）类似，您可以使用 `dtrace -x` 在命令行上设置版本选项：

```
# dtrace -x version=1.0 -n 'BEGIN{trace("hello");}'
```

或者使用 `#pragma D option` 语法在 D 程序源文件中设置该选项：

```
#pragma D option version=1.0



BEGIN

{

    trace("hello");
```

```
}
```

如果您使用 `#pragma D option` 语法请求版本绑定，则必须将该指令放在 D 程序文件的顶部，位于任何其他声明和探测子句前面。 如果版本绑定参数为无效的版本字符串，或引用了 D 编译器未提供的版本，则会生成相应的错误消息，并且编译将会失败。 因此，您也可借助版本绑定功能，使在 DTrace 旧版本上执行 D 脚本失败，并生成直观的错误消息。

编译程序声明和子句之前，D 编译器会将相应接口版本的 D 类型、函数、常量和转换器集装入编译器名称空间。 因此，指定的任何版本绑定选项仅控制对程序可见的标识符、类型和转换器集，以及程序定义的变量、类型和转换器。 版本绑定可防止 D 编译器装入可能会定义与程序源代码中的声明冲突，并因此产生编译错误的标识符或转换器的新接口。 有关如何选择不会与 DTrace 将来版本提供的接口冲突的标识符名称的技巧，请参阅第 47 页中的“标识符名称和关键字”。

提供器版本控制

与 D 编译器提供的接口不同，DTrace 提供器（即探测器和探测器参数）提供的接口不受 D 编程接口或前面介绍的版本绑定选项的影响，或与其没有关联。建立可用的提供器接口是将编译后的检测过程装入操作系统内核的 DTrace 软件中的过程的一部分，此过程因指令集体系结构、操作平台、处理器、Solaris 系统中安装的软件以及当前的安全权限而异。 D 编译器和 DTrace 运行时将检查 D 程序子句中描述的探测器，并在 D 程序请求的探测器不可用时报告相应的错误消息。 这些功能与 D 编程接口版本互不相关，因为 DTrace 提供器不会导出可能与 D 程序中的定义冲突的接口；也就是说，您只能启用 D 中的探测器，而不能对其进行定义，并且探测器名称保留在一个与其他 D 程序标识符分开的名称空间中。

DTrace 提供器提供了一个特定的 Solaris 发行版，在《Solaris 动态跟踪指南》的相应版本中，对该提供器进行了介绍。在该指南中，与每个提供器对应的章节还将说明对给定提供器所做的任何相关更改，以及该提供器提供的新功能。您可以使用 `dtrace -l` 选项来了解 Solaris 系统中可用的提供器和探测器的集合。 提供器使用 DTrace 稳定性属性标记其接口，因此，您可以使用 DTrace 稳定性报告功能（参阅第 39 章），确定在将来的 Solaris 发行版中，是否可能会更改或提供 D 程序使用的提供器接口。

词汇表

action（操作） 由 DTrace 框架实现的行为，可在触发探测器时执行，用于跟踪 DTrace 外部的数据或修改系统状态。操作包括跟踪数据、停止进程、捕获堆栈跟踪以及其他内容。

aggregation（聚合） 一个对象，用于存储第 9 章中正式定义的聚合函数的结果，由可用于组织结果的表达式元组建立索引。

clause（子句） 一种 D 程序声明，其中包含探测器说明符列表、可选谓词以及由大括号 `{ }` 括起的可选操作语句列表。

consumer（使用者） 一种使用 DTrace 来启用检测过程并读出产生的跟踪数据流的程序。`dtrace` 命令为标准的 DTrace 使用者；`lockstat`(1M) 实用程序为另外一个专用的 DTrace 使用者。

DTrace 一种动态跟踪工具，用于为任意问题提供简明回答。

enabling（启用项） 一组已启用的探测器以及与其关联的谓词和操作。

predicate（谓词） 一个逻辑表达式，用于确定在触发探测器时是否应执行一组跟踪操作。每条 D 程序子句都可以包含一个与其关联的谓词，该谓词应放在斜杠 `/ /` 中。

probe（探测器） 系统中的位置或活动，DTrace 可以将包括谓词和操作的检测过程动态地绑定到这些位置或活动中。每个探测器都通过一个元组命名，指示其提供器、模块、函数和语义名称。探测器可以固定到特定模块和函数，或者，如果探测器（如 `profile` 计时器）与任何特定程序位置都没有关联，则为不固定。

provider（提供器） 一个内核模块，用于代表 DTrace 框架实现特定类型的检测过程。提供器可导出探测器的名称空间及其名称和数据语义的稳定性矩阵，如本书各章中所述。

subroutine（子例程） 由 DTrace 框架实现的行为，可在触发探测器时执行，用于修改内部 DTrace 状态，但不跟踪任何数据。与操作类似，使用 D 函数调用语法请求子例程。

translator（转换器） 一种 D 赋值语句集合，用于将特定受检测子系统的实现详细信息转换为 `struct` 类型的对象，以构成比输入表达式更稳定的接口。

索引

E

F

I

K

L

M

提

推

外

自