DTrace 用户指南

文件号码 E22192
2011 年 8 月

ORACLE

120418@25097

目录

前言

《DTrace 用户指南》简要介绍了功能强大的跟踪和分析工具 DTrace。本书包含对 DTrace 工具及其功能的描述，还包含如何使用 DTrace 执行相对简单常见的任务的说明。

目标读者

DTrace 内置在 Solaris 中，是一个全面的动态跟踪工具。用户可以使用 DTrace 工具检查用户程序的行为或操作系统的行为。系统管理员或应用程序开发者可将 DTrace 用于实时的产品系统中。

使用 DTrace，Solaris 开发者和管理员可以执行以下操作：

- 执行使用 DTrace 工具的定制脚本
- 执行使用 DTrace 检索跟踪数据的分层工具

本书并非 DTrace 或 D 脚本语言的全部指南。有关详细的参考信息，请参阅《Solaris 动态跟踪指南》。

阅读本书之前

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 5 页中的 “相关书籍”中提供了一些其他文档，或许会对您有所帮助。

相关书籍

有关 DTrace 的详细参考信息，请参见《Solaris 动态跟踪指南》。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 Mauro, Jim 和 McDougall, Richard 合著的《Solaris Internals: Core Kernel Components》。Sun Microsystems Press 出版，2001。ISBN 0-13-022496-0

- 由 Vahalia, Uresh 编著的《UNIX Internals: The New Frontiers》。Prentice Hall 出版，1996。ISBN 0-13-101908-2

文档、支持和培训

Oracle Web 站点提供有关以下附加资源的信息：

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

印刷约定

下表介绍了本书中的印刷约定。

表 P–1 印刷约定

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

命令中的 shell 提示符示例

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

表 P–2 Shell 提示符

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

◆ ◆ ◆ 第 1 章

1 简介

DTrace 内置在 Solaris 中，是一个全面的动态跟踪工具。DTrace 可以由管理员和开发者使用，并且可以在实时生产系统上安全使用。使用 DTrace，可以检查用户程序的行为和操作系统的行为。DTrace 的用户可以通过 D 脚本语言创建定制程序。定制程序提供了动态检测系统的能力。定制程序为有关特定应用程序行为的具体问题提供了即时简明的回答。

DTrace 功能

DTrace 框架提供了称为**探测器**的检测点。DTrace 用户可以使用探测器来记录和显示与内核或用户进程相关的信息。每个 DTrace 探测器都是由一个特定的行为激活的。此探测器激活称为**触发**。例如，在进入任意内核函数时触发的一个探测器。此探测器示例可以显示以下信息：

- 传递给函数的任意参数
- 内核中任意的全局变量
- 一个指示函数调用时间的时间戳
- 一个指示负责调用函数的代码部分的栈跟踪
- 调用函数时正在运行的进程
- 负责执行函数调用的线程

触发探测器时，您可以指定 DTrace 要采取的具体**操作**。DTrace 操作通常记录系统行为需关注的方面，例如时间戳或函数参数。

探测器是由**提供器**来执行的。探测器提供器是使得给定探测器能够被触发的一个内核模块。例如，函数边界跟踪提供器 `fbt` 为每个内核模块中的几乎所有函数提供进入和返回探测器。

DTrace 具有重要的数据管理功能。使用这些功能，DTrace 用户能够删改探测器所报告的数据，避免与生成和过滤不需要的数据相关的开销。DTrace 还提供了用于在引导过程中进行跟踪和从内核故障转储检索数据的机制。DTrace 中的所有检测都是动态的。探测器是在其被使用时独立启用的，非活动探测器不提供已检测的代码。

DTrace **使用者** 是与 DTrace 框架交互的任意进程。虽然 dtrace(1M) 是主要的 DTrace 使用者，但也存在其他使用者。这些其他使用者主要包括现有实用程序的新版本，例如 lockstat(1M)。DTrace 框架对并发使用者的数目没有限制。

可以使用以 D 语言（结构与 C 语言类似）编写的脚本来修改 DTrace 的行为。D 语言提供对内核 C 类型与内核静态和内核全局变量的访问。D 语言支持 ANSI C 运算符。

体系结构概述

DTrace 工具包含下列各组件：

- 用户级使用者程序，例如 dtrace
- 提供器，打包为内核模块，提供探测器来收集跟踪数据
- 一个库接口，使用者程序使用该接口通过 dtrace(7D) 内核驱动程序访问 DTrace 工具

DTrace 提供器

提供器提供了用于检测系统的一种方法。提供器使得探测器可供 DTrace 框架使用。DTrace 向提供器发送有关探测器启用时间的信息。当某个已启用的探测器被触发时，提供器会将控制权移交给 DTrace。

提供器打包为一组内核模块。每个模块执行一种特定类型的检测来创建探测器。使用 DTrace 时，每个提供器都可以将其可提供的探测器发布到 DTrace 框架。可以启用跟踪操作并将其绑定到任意一个已经发布的探测器上。

某些提供器能够根据用户的跟踪请求新建探测器。

DTrace 探测器

探测器具有下列各项属性：

- 由**提供器**启用
- 可标识自己所检测的**模块**和**函数**
- 具有**名称**

上述四项属性为每个探测器定义了一个可作为其唯一标识符的 4 元组，其格式为**提供器**:**模块**:**函数**:**名称**。每个探测器还具有一个唯一的整数标识符。

DTrace 谓词

谓词是括在斜杠 / / 内的表达式。探测器被触发时，将对谓词进行评估以确定是否应执行关联的操作。谓词是用于在 D 程序中生成更为复杂的控制流的主要条件结构。对于任意一个探测器，可以完全省略探测器的谓词部分。如果省略了谓词部分，则在触发探测器时将始终执行相应的操作。

谓词表达式可以使用前面所述的任何 D 运算符。谓词表达式会引用 D 数据对象，例如变量和常量。谓词表达式的计算结果必须是一个整数或指针类型的值。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。

DTrace 操作

操作是 DTrace 虚拟机在内核中执行的可由用户编程的语句。操作具有以下特性：

- 操作在探测器触发时执行
- 操作是完全可以用 D 脚本语言进行编写的
- 大多数操作都会记录一个指定的系统状态
- 操作可以根据所描述方式精确更改系统的状态。这类操作称为**破坏性操作**。缺省情况下不允许执行破坏性操作。
- 许多操作都使用以 D 脚本语言编写的表达式

D 脚本语言

对于简单函数，您可以直接从命令行使用 `dtrace` 命令来调用 DTrace 框架。要使用 DTrace 执行较为复杂的函数，请使用 D 脚本语言编写脚本。可以使用 `-s` 选项加载指定的脚本以供 DTrace 使用。有关如何使用 D 脚本语言的信息，请参见第 3 章，使用 D 语言编写脚本。

◆ ◆ ◆ 第 2 章

2 DTrace 基础知识

本章将提供 DTrace 工具的教程以及几个基本任务示例。

列出探测器

您可以通过将 `-l` 选项传递给 `dtrace` 命令来列出所有 DTrace 探测器：

```
# dtrace -l
ID   PROVIDER    MODULE        FUNCTION NAME
 1     dtrace                           BEGIN
 2     dtrace                           END
 3     dtrace                           ERROR
 4    syscall                     nosys entry
 5    syscall                     nosys return
 6    syscall                     rexit entry
 7    syscall                     rexit return
 8    syscall                    forkall entry
 9    syscall                    forkall return
10    syscall                       read entry
11    syscall                       read return
...
```

要计算系统上所有可用的探测器数目，可以键入以下命令：

```
# dtrace -l | wc -l
```

所报告的探测器数目会根据操作平台和所安装软件的不同而有所不同。某些探测器在 `MODULE` 或 `FUNCTION` 列下没有条目列出，例如上例中的 `BEGIN` 和 `END` 探测器。在这些字段中具有空白条目的探测器不与所检测的某个具体程序函数或位置相对应。这些探测器引用更为抽象的概念，例如某个跟踪请求的末尾。名称中包含模块和函数的探测器称为**固定探测器**。不与模块和函数关联的探测器称为**非固定探测器**。

可以使用更多选项来列出特定的探测器，如下列示例所示。

示例 2–1 按特定函数列出探测器

可以通过将 `-f` 选项与函数名称传递给 DTrace 来列出与某特定函数关联的探测器。

```
# dtrace -l -f cv_wait
ID      PROVIDER        MODULE        FUNCTION NAME
12921        fbt       genunix         cv_wait entry
12922        fbt       genunix         cv_wait return
```

示例 2–2 按特定模块列出探测器

可以通过将 `-m` 选项与模块名称传递给 DTrace 来列出与某特定模块关联的探测器。

```
# dtrace -l -m sd
ID      PROVIDER        MODULE        FUNCTION NAME
17147        fbt            sd          sdopen entry
17148        fbt            sd          sdopen return
17149        fbt            sd         sdclose entry
17150        fbt            sd         sdclose return
17151        fbt            sd      sdstrategy entry
17152        fbt            sd      sdstrategy return
...
```

示例 2–3 按特定名称列出探测器

可以通过将 `-n` 选项与名称传递给 DTrace 来列出与某特定名称关联的探测器。

```
# dtrace -l -n BEGIN
ID      PROVIDER        MODULE        FUNCTION NAME
1         dtrace                              BEGIN
```

示例 2–4 按启动点提供器列出探测器

您可以通过将 `-P` 选项与提供器名称传递给 DTrace 来列出从某特定提供器启动的探测器。

```
# dtrace -l -P lockstat
ID        PROVIDER        MODULE             FUNCTION NAME
469       lockstat       genunix          mutex_enter adaptive-acquire
470       lockstat       genunix          mutex_enter adaptive-block
471       lockstat       genunix          mutex_enter adaptive-spin
472       lockstat       genunix           mutex_exit adaptive-release
473       lockstat       genunix        mutex_destroy adaptive-release
474       lockstat       genunix       mutex_tryenter adaptive-acquire
...
```

示例 2–5 支持某个特定函数或模块的多个提供器

一个特定的函数或特定的模块可能受多个提供器支持，如下例所示。

```
# dtrace -l -f read
ID        PROVIDER        MODULE             FUNCTION NAME
  10       syscall                               read entry
  11       syscall                               read return
```

示例 2–5 支持某个特定函数或模块的多个提供器 （续）

```
4036       sysinfo       genunix                read readch
4040       sysinfo       genunix                read sysread
7885           fbt       genunix                read entry
7886           fbt       genunix                read return
```

如上例所示，针对探测器的列出命令的输出显示了以下信息：

- 为探测器分配的唯一整数探测器 ID

  注 – 探测器 ID 在给定的 Solaris 操作系统发行版或修补级别中是唯一的。

- 提供器名称
- 模块名称（如果适用）
- 函数名称（如果适用）
- 探测器名称

在 DTrace 中指定探测器

您可以通过列出唯一地标识探测器的 4 元组的每个组件来完全指定某个探测器。探测器规范的格式为**提供器**:**模块**:**函数**:**名称**。探测器规范中的空组件可以匹配任何对象。例如，`fbt::alloc:entry` 规范指定具有下列特性的探测器：

- 该探测器必须来自 `fbt` 提供器
- 该探测器可以位于任何模块中
- 该探测器必须位于 `alloc` 函数中
- 该探测器的名称必须为 `entry`

4 元组左半部分的元素是可选的。探测器规范 `::open:entry` 等效于规范 `open:entry`。这两个规范都将匹配所有提供器和内核模块中具有函数名称 `open` 且名称为 `entry` 的探测器。

```
# dtrace -l -n open:entry
  ID      PROVIDER        MODULE             FUNCTION NAME
  14       syscall                               open entry
7386           fbt       genunix                  open entry
```

您还可以使用某种模式匹配语法来描述探测器，该语法类似于 `sh(1)` 手册页的 *File Name Generation* 一节中描述的语法。该语法支持特殊字符 `*`、`?`、`[` 和 `]`。探测器描述 `syscall::open*:entry` 与 `open` 和 `open64` 这两个系统调用都匹配。`?` 字符表示名称中的任意单个字符。`[` 和 `]` 字符用于指定名称中的一组特定字符。

启用探测器

您可以使用 dtrace 命令在指定探测器但不指定 -l 选项的情况下启用探测器。如果没有进一步的指令，则在指定的探测器触发时，DTrace 执行缺省的操作。缺省的探测器操作仅指明指定的探测器已触发，不记录任何其他数据。下面的代码示例启用 sd 模块中的每个探测器。

示例 2–6 按模块启用探测器

```
# dtrace -m sd
CPU     ID                    FUNCTION:NAME
  0  17329           sd_media_watch_cb:entry
  0  17330          sd_media_watch_cb:return
  0  17167                      sdinfo:entry
  0  17168                     sdinfo:return
  0  17151                  sdstrategy:entry
  0  17152                 sdstrategy:return
  0  17661          ddi_xbuf_qstrategy:entry
  0  17662         ddi_xbuf_qstrategy:return
  0  17649                xbuf_iostart:entry
  0  17341            sd_xbuf_strategy:entry
  0  17385                sd_xbuf_init:entry
  0  17386               sd_xbuf_init:return
  0  17342           sd_xbuf_strategy:return
  0  17177     sd_mapblockaddr_iostart:entry
  0  17178    sd_mapblockaddr_iostart:return
  0  17179               sd_pm_iostart:entry
  0  17365                 sd_pm_entry:entry
  0  17366                sd_pm_entry:return
  0  17180              sd_pm_iostart:return
  0  17181             sd_core_iostart:entry
  0  17407         sd_add_buf_to_waitq:entry
...
```

本示例中的输出表明，缺省操作显示触发探测器的 CPU、由 DTrace 分配的整数探测器 ID、触发探测器的函数，以及探测器名称。

示例 2–7 按提供器启用探测器

```
# dtrace -P syscall
dtrace: description 'syscall' matched 452 probes
CPU     ID                    FUNCTION:NAME
  0     99                     ioctl:return
  0     98                      ioctl:entry
  0     99                     ioctl:return
  0     98                      ioctl:entry
  0     99                     ioctl:return
  0    234                  sysconfig:entry
  0    235                 sysconfig:return
  0    234                  sysconfig:entry
  0    235                 sysconfig:return
  0    168                  sigaction:entry
  0    169                 sigaction:return
  0    168                  sigaction:entry
  0    169                 sigaction:return
  0     98                       ioctl:entry
```

示例 2–7 按提供器启用探测器 （续）

```
  0     99                       ioctl:return
  0    234                    sysconfig:entry
  0    235                   sysconfig:return
  0     38                          brk:entry
  0     39                         brk:return
...
```

示例 2–8 按名称启用探测器

```
# dtrace -n zfod
dtrace: description 'zfod' matched 3 probes
CPU     ID                    FUNCTION:NAME
  0   4080                   anon_zero:zfod
  0   4080                   anon_zero:zfod
^C
```

示例 2–9 按完全指定的名称启用探测器

```
# dtrace -n clock:entry
dtrace: description 'clock:entry' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0   4198                      clock:entry
^C
```

DTrace 操作基础知识

操作使得 DTrace 能够与 DTrace 框架外的系统进行交互。大多数常见操作将数据记录到 DTrace 缓冲区。其他操作可能会停止当前进程，在当前进程上发出特定信号，或者停止跟踪。更改系统状态的操作被视为**破坏性操作**。缺省情况下，数据记录操作将数据记录到**主体缓冲区**。主体缓冲区是在每个 DTrace 调用中提供的，并且始终是为每个 CPU 分别分配的。可以通过使用 `-cpu` 选项将跟踪和缓冲区分配限定到单个 CPU。有关 DTrace 缓冲的更多信息，请参见《Solaris 动态跟踪指南》中的第 11 章 “缓冲区和缓冲”。

本节中的示例使用了由内置的 D 变量组成的 D 表达式。下面列出了一些最常用的 D 变量：

`pid` 此变量包含当前进程 ID。

`execname` 此变量包含当前的可执行文件名称。

`timestamp` 此变量包含自引导以来经过的时间，以纳秒为单位。

`curthread` 此变量包含指向表示当前线程的 `kthread_t` 结构的一个指针。

`probemod` 此变量包含当前探测器的模块名称。

`probefunc` 此变量包含当前探测器的函数名称。

`probename` 此变量包含当前探测器的名称。

有关 D 脚本语言的内置变量的完整列表，请参见变量。

D 脚本语言还提供了执行特定操作的内置函数。可以在《Solaris 动态跟踪指南》中的第 10 章 “操作和子例程”。`trace()` 函数将 D 表达式的结果记录到跟踪缓冲区，如下列示例所述：

- `trace(pid)` 跟踪当前进程 ID
- `trace(execname)` 跟踪当前可执行文件的名称
- `trace(curthread->t_pri)` 跟踪当前线程的 `t_pri` 字段
- `trace(probefunc)` 跟踪探测器的函数名称

要指定您想要探测器采取的特定操作，请在 `{}` 字符中间键入操作的名称，如下例所示。

示例 2–10 指定探测器的操作

```
# dtrace -n 'readch {trace(pid)}'
dtrace: description 'readch ' matched 4 probes
CPU     ID                    FUNCTION:NAME
  0   4036                      read:readch          2040
  0   4036                      read:readch          2177
  0   4036                      read:readch          2177
  0   4036                      read:readch          2040
  0   4036                      read:readch          2181
  0   4036                      read:readch          2181
  0   4036                      read:readch             7
...
```

因为所请求的操作是 `trace(pid)`，所以进程标识号 (PID) 显示在输出的最后一列中。

示例 2–11 跟踪可执行文件名称

```
# dtrace -m 'ufs {trace(execname)}'
dtrace: description 'ufs ' matched 889 probes
CPU     ID                    FUNCTION:NAME
  0  14977                 ufs_lookup:entry            ls
  0  15748                ufs_iaccess:entry            ls
  0  15749               ufs_iaccess:return            ls
  0  14978                ufs_lookup:return            ls
...
  0  15007                   ufs_seek:entry         utmpd
  0  15008                  ufs_seek:return         utmpd
  0  14963                  ufs_close:entry         utmpd
^C
```

示例 2–12 跟踪 Entry 的系统调用时间

```
# dtrace -n 'syscall:::entry {trace(timestamp)}'
dtrace: description 'syscall:::entry ' matched 226 probes
CPU     ID                    FUNCTION:NAME
  0    312                     portfs:entry    157088479572713
  0     98                      ioctl:entry    157088479637542
  0     98                      ioctl:entry    157088479674339
  0    234                  sysconfig:entry    157088479767243
```

示例 2–12 跟踪 Entry 的系统调用时间 （续）

```
...
  0     98                        ioctl:entry    157088481033225
  0     60                        fstat:entry    157088481050686
  0     60                        fstat:entry    157088481074680
^C
```

示例 2–13 指定多个操作

要指定多个操作，请列出以 ; 字符分隔的各个操作。

```
# dtrace -n 'zfod {trace(pid);trace(execname)}'
dtrace: description 'zfod ' matched 3 probes
CPU     ID                    FUNCTION:NAME
  0   4080                    anon_zero:zfod    2195   dtrace
  0   4080                    anon_zero:zfod    2195   dtrace
  0   4080                    anon_zero:zfod    2195   dtrace
  0   4080                    anon_zero:zfod    2195   dtrace
  0   4080                    anon_zero:zfod    2195   dtrace
  0   4080                    anon_zero:zfod    2197   bash
  0   4080                    anon_zero:zfod    2207   vi
  0   4080                    anon_zero:zfod    2207   vi
...
```

数据记录操作

缺省情况下，本节中的操作将数据记录到主体缓冲区，但还可以使用各个操作将数据记录到推理缓冲区。有关推理缓冲区的更多详细信息，请参见第 49 页中的 “推理跟踪”。

trace() 函数

`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。

tracemem() 函数

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

`tracemem()` 操作将数据从内存中的某个地址复制到缓冲区中。此操作复制的字节数是在 *nbytes* 中指定的。从中复制数据的地址是在 *addr* 中以 D 表达式的形式指定的。将数据复制到的缓冲区是在 *buf* 中指定的。

printf() 函数

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

与 trace() 操作一样，printf() 操作跟踪 D 表达式。不过，printf() 操作允许您控制格式，这类似于 printf(3C) 函数。与 printf 函数一样，参数包括一个 *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 命令将根据指定的格式字符串对参数进行格式化，以便输出。

有关 printf() 操作的更多信息，请参见《Solaris 动态跟踪指南》中的第 12 章 “输出格式化”。

printa() 函数

```
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《Solaris 动态跟踪指南》中的第 9 章 “聚合”。如果没有提供 *format* 值，则 printa() 操作仅向 DTrace 使用者发送一条指令。接收该指令的使用者将处理聚合并以缺省格式显示聚合。有关 printa() 格式字符串的更详细描述，请参见《Solaris 动态跟踪指南》中的第 12 章 “输出格式化”。

stack() 函数

```
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。内核栈的深度是由 *nframes* 中指定的值指定的。如果没有为 *nframes* 指定值，则 stack 操作将记录由 stackframes 选项指定的数目的栈帧。

ustack() 函数

```
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中。用户栈的深度等于 *nframes* 中指定的值。如果没有指定 *nframes* 的值，则 ustack 操作将记录由 ustackframes 选项指定的数目的栈帧。ustack() 操作在探测器触发时确定调用帧的地址。在 DTrace 使用者在用户级别处理 ustack() 操作之前，ustack() 操作不会将栈帧转换为符号。如果为 *strsize* 指定了一个非零值，则 ustack() 操作会分配指定数量的字符串空间并使用它直接从内核执行“地址到符号”转换。

jstack() 函数

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

jstack() 操作是使用由 jstackframes 选项指定的栈帧数目的 ustack() 的别名。jstack 操作使用由 jstackstrsize 选项指定的值来确定字符串空间大小。jstacksize 操作缺省使用某个非零值。

破坏性操作

要使用破坏性操作，您必须显式启用它们。可以使用 -w 选项启用破坏性操作。如果您在没有显式启用破坏性操作的情况下试图在 dtrace 中使用它们，则 dtrace 将失败，且返回类似于下例的一条消息：

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

有关 DTrace 操作的更多信息（包括破坏性操作），请参见《Solaris 动态跟踪指南》中的第 10 章 “操作和子例程”。

处理破坏性操作

某些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《Solaris 动态跟踪指南》中的第 35 章 “安全性”。

stop() 函数

如果探测器是在启用了 stop() 操作的情况下触发的，则触发该探测器的进程将在离开内核时停止。该进程的停止方式与由proc(4) 操作停止的进程的停止方式相同。

raise() 函数

```
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。

copyout() 函数

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

copyout() 操作将数据从缓冲区复制到内存中的地址。此操作复制的字节数是在 *nbytes* 中指定的。从其中复制数据的缓冲区是在 *buf* 中指定的。将数据复制到的地址是在 *addr* 中指定的。该地址位于与当前线程相关联进程的地址空间中。

copyoutstr() 函数

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

copyoutstr() 操作将字符串复制到内存中的地址。要复制的字符串是在 *str* 中指定的。将字符串复制到的地址是在 *addr* 中指定的。该地址位于与当前线程相关联进程的地址空间中。

system() 函数

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

system() 操作导致系统执行由 *program* 指定的程序，就像该程序被作为输入传递给 shell 一样。

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。请慎用这些操作。这些操作影响系统上的每个进程，并且可能会影响其他系统，具体取决于受影响系统的网络服务。

breakpoint() 函数

```
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。

panic() 函数

```
void panic(void)
```

当包含 panic() 操作的探测器触发时，内核将崩溃。该操作可以在某个重要时间强制执行系统故障转储。您可以将该操作与环形缓冲和事后分析结合使用来对系统问题进行诊断。有关更多信息，请分别参见《Solaris 动态跟踪指南》中的第 11 章 “缓冲区和缓冲”与《Solaris 动态跟踪指南》中的第 37 章 “事后跟踪”。

chill() 函数

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

当包含 chill() 操作的探测器触发时，DTrace 将旋转指定的纳秒数。chill() 操作对于探究与计时有关的问题非常有用。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。

DTrace 聚合

对于与性能有关的问题，聚合的数据通常比离散的数据点更为有用。DTrace 提供了几个内置的聚合函数。如果某个聚合函数先应用于某个数据集合的各个子集，然后再应用于这些子集的分析结果，这样所得到的结果与该聚合函数应用于作为一个整体的集合所返回的结果相同。

DTrace 工具为聚合存储数目不断变化的数据条目。聚合函数仅存储当前的中间结果和函数所应用于的新元素。中间结果是基于每个 CPU 分别分配的。因为该分配方案不需要锁，所以其实现本质上是可伸缩的。

DTrace 聚合语法

DTrace 聚合采用下面的一般形式：

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

在此一般形式中，变量的定义如下所述：

`name` 聚合的名称，以 @ 字符作为前缀。

`keys` 以逗号分隔的 D 表达式列表。

`aggfunc` 某个 DTrace 聚合函数。

`args` 适用于聚合函数的参数的逗号分隔列表。

表 2–1 DTrace 聚合函数

函数名	参数	结果
`count`	无	`count` 函数被调用的次数。
`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。
`lquantize`	标量表达式、下界、上界、步长值	指定表达式的值的线性频数分布，其大小由指定的范围限定。此聚合函数递增**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。此聚合函数递增**最高**二次方存储桶中**小于**指定表达式的值。

示例 2–14 使用聚合函数

本示例使用 `count` 聚合函数来对每个进程的`write(2)` 系统调用进行计数。在 `dtrace` 命令终止之前，该聚合函数不输出任何数据。输出数据提供在 `dtrace` 命令处于活动状态期间收集的数据的汇总。

```
# cat writes.d
#!/usr/sbin/dtrace -s
syscall::write:entry]
{   @numWrites[execname] = count();
}

# ./writes.d
dtrace: script 'writes.d' matched 1 probe
^C
  dtrace                            1
  date                              1
  bash                              3
  grep                             20
  file                            197
  ls                              201
```

◆ ◆ ◆ 第 3 章

3 使用 D 语言编写脚本

本章讨论了开始编写您自己的 D 语言脚本所需的基本信息。

编写 D 脚本

复杂的 DTrace 探测器集合在命令行上可能难以管理。`dtrace` 命令支持脚本。您可以通过将 `-s` 选项以及脚本的文件名传递给 `dtrace` 命令来指定某个脚本。您还可以创建可执行的 DTrace 解释器文件。DTrace 解释器文件始终以 **`#!/usr/sbin/dtrace -s`**。

可执行的 D 脚本

名为 `syscall.d` 的此示例脚本跟踪可执行文件每次进入每个系统调用时的可执行文件名称：

```
syscall:::entry
{
   trace(execname);
}
```

请注意，文件名以 .d 后缀结尾。这是 D 脚本的约定结尾。您可以从 DTrace 命令行使用以下命令运行此脚本：

```
# dtrace -s syscall.d
dtrace: description 'syscall ' matched 226 probes
CPU     ID                    FUNCTION:NAME
  0    312                     pollsys:entry    java
  0     98                       ioctl:entry    dtrace
  0     98                       ioctl:entry    dtrace
  0    234                   sysconfig:entry    dtrace
  0    234                   sysconfig:entry    dtrace
  0    168                   sigaction:entry    dtrace
  0    168                   sigaction:entry    dtrace
  0     98                       ioctl:entry    dtrace
^C
```

您可以通过执行两个步骤通过在命令行上输入文件名运行此脚本。首先，确认文件的第一行调用了解释器。解释器调用行是 `#!/usr/sbin/dtrace -s`。然后，为文件设置执行权限。

示例 3–1 从命令行运行 D 脚本

```
# cat syscall.d
#!/usr/sbin/dtrace -s

syscall:::entry
{
   trace(execname);
}

# chmod +x syscall.d
# ls -l syscall.d
-rwxr-xr-x   1 root     other       62 May 12 11:30 syscall.d
# ./syscall.d
dtrace: script './syscall.d' matched 226 probes
CPU     ID                    FUNCTION:NAME
  0     98                      ioctl:entry    dtrace
  0     98                      ioctl:entry    dtrace
  0    312                     pollsys:entry    java
  0    312                     pollsys:entry    java
  0    312                     pollsys:entry    java
  0     98                      ioctl:entry    dtrace
  0     98                      ioctl:entry    dtrace
  0    234                   sysconfig:entry    dtrace
  0    234                   sysconfig:entry    dtrace
^C
```

D 文本字符串

D 语言支持文本字符串。DTrace 将字符串表示为以空字节终止的字符数组。字符串可见部分的长度取决于空字节的位置。DTrace 将每个字符串存储在大小固定的一个数组中，从而确保每个探测器跟踪一致的数据量。字符串不能超出预定义的字符串界限的长度。可以在您的 D 程序中修改该界限，也可以在 `dtrace` 命令行上通过调整 `strsize` 选项来修改该界限。有关可调整的 DTrace 选项的更多信息，请参见《Solaris 动态跟踪指南》中的第 16 章 “选项和可调参数”。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。有关 D 文本字符串的更多信息，请参见《Solaris 动态跟踪指南》中的第 6 章 “字符串”。

示例 3–2 将 D 文本字符串与 trace() 函数一起使用

```
# cat string.d

#!/usr/sbin/dtrace -s

fbt::bdev_strategy:entry
{
   trace(execname);
```

示例 3–2 将 D 文本字符串与 trace() 函数一起使用 （续）

```
   trace(" is initiating a disk I/O\n");
}
```

文本字符串末尾的 \n 符号生成一个新行。要运行此脚本，请输入以下命令：

```
# dtrace -s string.d
dtrace: script 'string.d' matched 1 probes
CPU     ID                    FUNCTION:NAME
  0   9215               bdev_strategy:entry   bash is initiating a disk I/O

  0   9215               bdev_strategy:entry   vi is initiating a disk I/O

  0   9215               bdev_strategy:entry   vi is initiating a disk I/O

  0   9215               bdev_strategy:entry   sched is initiating a disk I/O

^C
```

dtrace 命令的 -q 选项仅记录在脚本中或者在命令行调用中显式指定的操作。此选项抑制了 dtrace 命令通常情况下会生成的缺省输出。

```
# dtrace -q -s string.d
ls is initiating a disk I/O
cat is initiating a disk I/O
fsflush is initiating a disk I/O
vi is initiating a disk I/O
^C
```

创建使用参数的 D 脚本

可以使用 dtrace 命令来创建可执行的解释器文件。该文件必须具有执行权限。文件的起始行必须为 #!/usr/sbin/dtrace -s。您可以在该行上指定 dtrace 命令的其他选项。在指定选项时，必须仅使用一个破折号 (-)。将 s 选项列在最后，如下例所示。

```
#!/usr/sbin/dtrace -qvs
```

可以在 D 脚本中使用 #pragma 行来指定 dtrace 命令的选项，如下面的 D 代码片段所示：

```
# cat -n mem2.d
     1  #!/usr/sbin/dtrace -s
     2
     3  #pragma D option quiet
     4  #pragma D option verbose
     5
     6  vminfo:::
     ...
```

下表列出了可以在 #pragma 行中使用的选项名称。

表 3–1 DTrace 使用者选项

选项名	值	dtrace 别名	说明
aggrate	*time*		聚合读取的速率
aggsize	*size*		聚合缓冲区大小
bufresize	auto 或 manual		缓冲区调整大小策略
bufsize	*size*	-b	主体缓冲区大小
cleanrate	*time*		清除速率
cpu	*scalar*	-c	启用跟踪的 CPU
defaultargs	—		允许引用未指定的宏参数
destructive	—	-w	允许破坏性操作
dynvarsize	*size*		动态变量空间大小
flowindent	—	- F	缩进函数输入并加前缀 ->；取消缩进函数返回并加前缀 <-
grabanon	—	-a	声明匿名状态
jstackframes	*scalar*		jstack() 的缺省栈帧数
jstackstrsize	*scalar*		jstack() 的缺省字符串空间大小
nspec	*scalar*		推理数
quiet	—	-q	仅输出显式跟踪的数据
specsize	*size*		推理缓冲区大小
strsize	*size*		字符串大小
stackframes	*scalar*		栈帧数
stackindent	*scalar*		缩进 stack() 和 ustack() 输出时要使用的空格字符数
statusrate	*time*		状态检查的速率
switchrate	*time*		缓冲区切换的速率
ustackframes	*scalar*		用户栈帧数

D 脚本可以引用一组内置的宏变量。这些宏变量是由 D 编译器定义的。

`$[0-9]+`	宏参数
`$egid`	有效的组 ID
`$euid`	有效的用户 ID
`$gid`	实际的组 ID
`$pid`	进程 ID
`$pgid`	进程组 ID
`$ppid`	父进程 ID
`$projid`	项目 ID
`$sid`	会话 ID
`$target`	目标进程 ID
`$taskid`	任务 ID
`$uid`	实际的用户 ID

示例 3–3 PID 参数示例

此示例将正在运行的 `vi` 进程的 PID 传递给 D 脚本 `syscalls2.d`。该 D 脚本在 `vi` 命令退出时终止。

```
# cat -n syscalls2.d
     1  #!/usr/sbin/dtrace -qs
     2
     3  syscall:::entry
     4  /pid == $1/
     5  {
     6    @[probefunc] = count();
     7  }
     8  syscall::rexit:entry
     9  {
    10    exit(0);
    11  }

# pgrep vi
2208
# ./syscalls2.d 2208

  rexit                                                     1
  setpgrp                                                   1
  creat                                                     1
  getpid                                                    1
  open                                                      1
  lstat64                                                   1
  stat64                                                    1
  fdsync                                                    1
  unlink                                                    1
  close                                                     1
```

示例 3–3 PID 参数示例 （续）

```
  alarm                                             1
  lseek                                             1
  sigaction                                         1
  ioctl                                             1
  read                                              1
  write                                             1
```

DTrace 内置变量

下面的列表包括了 DTrace 框架的所有内置变量。

`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 `args[]` 数组被一个 `read(2)` 系统调用探测器引用，则 `args[0]` 是 `int` 类型的，`args[1]` 是 `void *` 类型的，`args[2]` 是 `size_t` 类型的。

`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《Solaris 动态跟踪指南》中的第 26 章 “sched 提供器”。

`processorid_t cpu`
: 当前 CPU 的 CPU 标识符。有关更多信息，请参见《Solaris 动态跟踪指南》中的第 26 章 “sched 提供器”。

`cpuinfo_t *curcpu`
: 当前 CPU 的 CPU 信息。有关更多信息，请参见《Solaris 动态跟踪指南》中的第 26 章 “sched 提供器”。

`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。`proc(4)` 手册页中更详细地介绍了此结构。

`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合启用的特定探测器。
`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`	在探测器触发时，当前 CPU 上的中断优先级别 (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 所属的延迟组的地址组 ID。有关 DTrace 中的 CPU 管理的更多信息，请参见《Solaris 动态跟踪指南》中的第 26 章 “sched 提供器”。有关地址组的更多信息，请参见《编程接口指南》中的第 4 章 “地址组 API”。
`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。
`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 的处理器集合的处理器集合 ID。有关更多信息，请参见《Solaris 动态跟踪指南》中的第 26 章 “sched 提供器”。
`string root`	与当前线程关联的进程的 root 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，该值等于对`pthread_self(3C)`的调用的结果。
`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。《Solaris 动态跟踪指南》中的第 33 章 “用户进程跟踪”中讨论了 `uregs[]` 数组的使用。
`uint64_t vtimestamp`	纳秒时间标记计数器的当前值。此计数器被虚拟化为当前进程已在某个 CPU 上运行的时间量。此计数器不包括在 DTrace 谓词和操作中花费的时间。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。

◆ ◆ ◆ 第 4 章

4 使用 DTrace

本章分析了 DTrace 在常见的基本任务中的使用，并提供了关于几种不同类型的跟踪的信息。

性能监视

多个 DTrace 提供器实现了与现有的性能监视工具对应的探测器：

- `vminfo` 提供器实现了与`vmstat(1M)` 工具对应的探测器
- `sysinfo` 提供器实现了与`mpstat(1M)` 工具对应的探测器
- `io` 提供器实现了与`iostat(1M)` 工具对应的探测器
- `syscall` 提供器实现了与`truss(1)` 工具对应的探测器

您可以使用 DTrace 工具来提取与捆绑的工具所提供的信息相同的信息，但具有更大的灵活性。DTrace 工具可以提供在探测器触发时可获得的任何内核信息。DTrace 工具使得您能够接收诸如进程标识、线程标识和栈跟踪之类的信息。

使用 sysinfo 提供器检查性能问题

`sysinfo` 提供器实现了与 `sys` 内核统计信息相对应的探测器。这些统计信息为系统监视实用程序（例如 `mpstat`）提供了输入。`sysinfo` 提供器探测器在名为 `kstat` 的 `sys` 递增前的那一刻触发。下面的列表介绍了 `sysinfo` 提供器提供的探测器。

`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) 指示已被认为处于空闲状态的 CPU。
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。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。
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。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。
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。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) 指示已被认为在等待 I/O 的 CPU。
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。如果此探测器触发，则 lockstat 提供器的 adaptive-block 和 adaptive-spin 探测器中的一个也会触发。
namei	尝试在文件系统中进行名称查找时将触发的探测器。
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，但在将控制权返回给执行该读取的线程之前将触发的探测器。读取可以通过 read(2)、readv(2) 或 pread(2) 系统调用进行。arg0 包含已成功读取的字节数。
rw_rdfails	每次尝试对读取者或写入者进行读锁定但该锁由某个写入者持有或者是某个写入者所需要的时，将触发的探测器。如果此探测器触发，则 lockstat 提供器的 rw-block 探测器也将触发。
rw_wrfails	每次尝试对读取者或写入者进行写锁定但该锁由多个读取者或另一写入者持有时，将触发的探测器。如果此探测器触发，则 lockstat 提供器的 rw-block 探测器也将触发。
sema	执行 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	每次执行 read、readv 或 pread 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量级陷阱。
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)。
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。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) 指示被描述为在等待 I/O 的 CPU。wait_ticks_io 与 cpu_ticks_wait 之间没有语义差别；wait_ticks_io 因为历史原因而单独存在。

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。写入可以通过 write、writev 或 pwrite 系统调用进行。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。

示例 4–1 将 quantize 聚合函数与 sysinfo 探测器一起使用

quantize 聚合函数显示其参数的二次方频数分布条形图。下面的示例使用 quantize 函数来确定系统上的所有进程在 10 秒的时间段内执行的 read 调用的大小。sysinfo 探测器的 arg0 参数指定统计信息的递增量。对于大多数 sysinfo 探测器，该值为 1。readch 和 writech 探测器是两个例外。对于这两个探测器，arg0 参数分别设置为读取或写入的实际字节数。

```
# cat -n read.d
   1  #!/usr/sbin/dtrace -s
   2  sysinfo:::readch
   3  {
   4     @[execname] = quantize(arg0);
   5  }
   6
   7  tick-10sec
   8  {
   9     exit(0);
  10  }

# dtrace -s read.d
dtrace: script 'read.d' matched 5 probes
CPU        ID                    FUNCTION:NAME
  0     36754                      :tick-10sec

  bash
           value  ---------- Distribution ---------- count
               0 |                                   0
               1 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 13
               2 |                                   0

  file
           value  ---------- Distribution ---------- count
              -1 |                                   0
               0 |                                   2
               1 |                                   0
               2 |                                   0
               4 |                                   6
               8 |                                   0
              16 |                                   0
              32 |                                   6
              64 |                                   6
             128 |@@                                 16
             256 |@@@@                               30
             512 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 199
            1024 |                                   0
            2048 |                                   0
```

示例 4–1 将 quantize 聚合函数与 sysinfo 探测器一起使用 （续）

```
          4096 |                                       1
          8192 |                                       1
         16384 |                                       0

  grep
         value  ---------- Distribution ---------- count
            -1 |                                       0
             0 |@@@@@@@@@@@@@@@@@@@@                   99
             1 |                                       0
             2 |                                       0
             4 |                                       0
             8 |                                       0
            16 |                                       0
            32 |                                       0
            64 |                                       0
           128 |                                       1
           256 |@@@@                                   25
           512 |@@@@                                   23
          1024 |@@@@                                   24
          2048 |@@@@                                   22
          4096 |                                       4
          8192 |                                       3
         16384 |                                       0
```

示例 4–2 查明交叉调用的源头

在本例中，请考虑 mpstat(1M) 命令的以下输出：

```
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0 2189   0 1302    14    1  215   12   54   28    0 12995   13  14   0  73
  1 3385   0 1137   218  104  195   13   58   33    0 14486   19  15   0  66
  2 1918   0 1039    12    1  226   15   49   22    0 13251   13  12   0  75
  3 2430   0 1284   220  113  201   10   50   26    0 13926   10  15   0  75
```

xcal 和 syscl 列中的值异常高，表明性能可能被浪费。系统相对空闲并且没有花费异常多的时间等待 I/O。xcal 列中的数字是按比例缩放的每秒数值，是从 sys kstat 的 xcalls 字段读取的。要查看哪些可执行文件是造成交叉调用的原因，请输入以下 dtrace 命令：

```
# dtrace -n 'xcalls {@[execname] = count()}'
dtrace: description 'xcalls ' matched 3 probes
^C
  find                                                    2
  cut                                                     2
  snmpd                                                   2
  mpstat                                                 22
  sendmail                                              101
  grep                                                  123
  bash                                                  175
  dtrace                                                435
  sched                                                 784
  xargs                                               22308
```

示例 4–2 查明交叉调用的源头 （续）

```
  file                                                  89889
#
```

该输出表明，大多数交叉调用发自 file(1) 和 xargs(1) 进程。您可以通过 pgrep(1) 和 ptree(1) 命令查明这些进程。

```
# pgrep xargs
15973
# ptree 15973
204   /usr/sbin/inetd -s
  5650  in.telnetd
    5653  -sh
      5657  bash
        15970 /bin/sh ./findtxt configuration
          15971 cut -f1 -d:
            15973 xargs file
              16686 file /usr/bin/tbl /usr/bin/troff /usr/bin/ul /usr/bin/vgrind /usr/bin/catman
```

该输出表明，xargs 和 file 命令构成了某个定制用户 shell 脚本的一部分。要定位该脚本，您可以执行以下命令：

```
# find / -name findtxt
/usrs1/james/findtxt
# cat /usrs1/james/findtxt
#!/bin/sh
find / -type f | xargs file | grep text | cut -f1 -d: > /tmp/findtxt$$
cat /tmp/findtxt$$ | xargs grep $1
rm /tmp/findtxt$$
#
```

该脚本同时运行许多进程。大量的进程间通信通过管道进行。管道的数量使得脚本成为资源密集型的。该脚本尝试查找系统上的每个文本文件，然后搜索每个文件来查找特定的文本。

跟踪用户进程

本节重点介绍了用于跟踪用户进程活动的 DTrace 工具，并提供了示例来说明其用途。

使用 copyin() 和 copyinstr() 子例程

DTrace 探测器在 Solaris 内核中执行。探测器使用 copyin() 或 copyinstr() 子例程将用户进程数据复制到内核的地址空间中。

请考虑下面的 write() 系统调用：

```
ssize_t write(int fd, const void *buf, size_t nbytes);
```

下面的 D 程序显示了一个试图输出传递给 `write` 系统调用的字符串内容的错误尝试：

```
syscall::write:entry
{
    printf("%s", stringof(arg1)); /* incorrect use of arg1 */
}
```

当您运行该脚本时，DTrace 会生成类似于下例的错误消息。

```
dtrace: error on enabled probe ID 1 (ID 37: syscall::write:entry): \
    invalid address (0x10038a000) in action #1
```

`arg1` 变量是一个地址，它引用正在执行系统调用的进程中的内存。请使用 `copyinstr()` 子例程来读取位于该地址的字符串。使用 `printf()` 操作来记录结果：

```
syscall::write:entry
{
    printf("%s", copyinstr(arg1)); /* correct use of arg1 */
```

该脚本的输出将显示传递给 `write` 系统调用的所有字符串。

避免错误

`copyin()` 和 `copyinstr()` 子例程无法从尚未触及的用户地址进行读取。如果包含某个有效地址的页面尚未由某个访问尝试引发错误，则该有效地址可能会导致错误。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
```

在上例的输出中，应用程序的运行正常且 `arg0` 中的地址有效。不过，`arg0` 中的地址引用了相应的进程尚未访问的一个页面。为解决此问题，在跟踪数据之前请等待内核或应用程序使用该数据。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 干扰

如果跟踪对 write 系统调用的每个调用，将会产生级联输出。对 write() 函数的每个调用都会导致 dtrace 命令在显示输出时调用 write() 函数。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避免此行为，如下例所示：

```
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。谓词 /pid != $pid/ 可确保脚本不跟踪与该脚本自身的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。可以使用 prstat(1M) 命令来查看 examine 进程行为。

```
$ 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
```

此报告表明，存在对 write() 函数的大量系统调用。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。

ustack() 操作

`ustack()` 操作跟踪用户线程的栈。如果在 `open()` 系统调用中，打开多个文件的某个进程偶尔失败，则可使用`ustack()` 操作来查明执行失败的 `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();
}
```

该脚本还说明了 `$1` 宏变量的用途。该宏变量接受在 `dtrace` 命令行上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() 操作记录栈的程序计数器 (PC) 值。dtrace 命令通过查看进程的符号表将那些 PC 值解析为符号名称。dtrace 命令输出无法解析为十六进制整数的 PC 值。

如果进程在 ustack() 数据针对输出进行格式化之前退出或者被终止，则 dtrace 命令可能无法将栈跟踪中的 PC 值转换为符号名称。在这种情况下，dtrace 命令将这些值显示为十六进制整数。为绕过此限制，请使用 dtrace 的 -c 或 -p 选项指定需关注的进程。如果事先不知道进程 ID 或命令，可使用以下 D 程序示例来绕过限制：示例使用了 open 系统调用探测器，但此方法可以用于使用 ustack 操作的任何脚本。

```
 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 ustack() 操作已应用于进程中的某个线程，则上面的脚本在进程退出前的那一刻停止该进程。此方法确保 dtrace 命令能够将 PC 值解析为符号名称。在清除动态变量后，stop_pids[pid] 的值被设置为 0。

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按需创建的。

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 宏变量扩展为命令行上的第一个操作数。该宏变量是要跟踪的进程的进程 ID。$2 宏变量扩展为命令行上的第二个操作数。该宏变量是从其中跟踪所有函数调用的函数的名称。

示例 4–3 userfunc.d：跟踪用户函数的进入和返回

```
pid$1::$2:entry
{
    self->trace = 1;
}
```

示例 4–3 userfunc.d：跟踪用户函数的进入和返回 （续）

```
pid$1::$2:return
/self->trace/
{
    self->trace = 0;
}

pid$1:::entry,
pid$1:::return
/self->trace/
{
}
```

此脚本生成类似于下例的输出：

```
# ./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。利用 dtrace 工具，可以使用 $target 宏变量以及 dtrace 选项 -c 和 -p 来创建和检测需关注的进程。下面的 D 脚本用于确定由某个特定的主题进程对 libc 进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。在 PID 为 123 的进程中，要启用 `bar.so` 模块的 `foo` 函数中与偏移量 `0x1c` 处的指令关联的探测器，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:
```

下面的示例说明了如何组合使用 `pid` 提供器与推理跟踪来跟踪函数中的每条指令。

示例 4–4 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);
    self->spec = 0;
}
```

示例 4–4 errorpath.d：跟踪用户函数调用错误路径 （续）

```
pid$1::$2:return
/self->spec && arg1 != 0/
{
    commit(self->spec);
    self->spec = 0;
}
```

执行 errorpath.d 时，该脚本的输出类似于下例。

```
# ./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
```

匿名跟踪

本节描述了不与任何 DTrace 使用者关联的跟踪。在没有可以运行的 DTrace 使用者进程情况下才使用匿名跟踪。只有超级用户可以创建匿名跟踪。在任意时刻，只能存在一个匿名启用。

匿名启用

要创建匿名启用，请使用 -A 选项以及指定了所需的探测器、谓词、操作和选项的 dtrace 命令。dtrace 命令向 dtrace(7D) 驱动程序的配置文件添加一系列表示您的请求的驱动程序属性。该配置文件通常是 /kernel/drv/dtrace.conf。在加载 dtrace 驱动程序时，该驱动程序读取这些属性。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**。dtrace 驱动程序通常是随充当 dtrace 提供器的任何驱动程序按需加载的。为允许在引导期间进行跟踪，必须尽早装入 dtrace 驱动程序。dtrace 命令向 /etc/system 添加必需的 forceload 语句（请参见 system(4) 以了解每个必需的 dtrace 提供器和 dtrace 驱动程序）。

当系统引导时，dtrace 驱动程序发送一条消息，指明已成功处理配置文件。匿名启用可以设置正常使用 dtrace 命令期间可用的任何选项。

要删除匿名启用，请在不指定任何探测器描述的情况下为 dtrace 指定 -A 选项。

声明匿名状态

当计算机已完全引导后，您可以通过指定 dtrace 命令及 -a 选项来声明现有的匿名状态。缺省情况下，-a 选项会声明匿名状态并处理现有数据，然后继续运行。要使用匿名状态并退出，请添加 -e 选项。

如果已从内核使用了匿名状态，则匿名状态无法替换。如果您试图声明不存在的匿名跟踪状态，则 dtrace 命令会生成类似于下例的一条消息：

```
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 驱动程序将在控制台上输出一条消息，指出驱动程序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.
  ...
```

在重新引导计算机后，指定 dtrace 命令及 -a 选项将使用匿名状态：

```
# 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
...
```

以下示例侧重于通过 `iprbattach()` 调用的那些函数。

```
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` 驱动程序将在控制台上输出一条不同的消息，指示有些许差别的启用：

```
...
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.
  ...
```

在计算机完成引导后，采用 -a 和 -e 选项运行 dtrace 命令来使用匿名数据，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
  ...
```

推理跟踪

本节针对**推理跟踪**讨论了 DTrace 工具。推理跟踪是试探性地跟踪数据并决定是将数据**提交**到某个跟踪缓冲区还是**放弃**数据的一种能力。用于过滤掉不需关注的事件的主要机制是**谓词**机制。如果在触发探测器时，您知道该探测事件是否为需要关注的事件，则谓词很有用。如果直到探测器触发之后，您才能知道是否需要关注某个给定的探测器事件，则谓词不太适合处理这类情况。

如果系统调用偶尔失败并返回某个常见的错误代码，则您可能需要检查导致错误情况的代码路径。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，然后决定是否将这些数据提交到另一个探测位置的主体缓冲区。最终得到的跟踪数据仅包含需要关注的输出，并且不需要进行后处理。

推理接口

下表介绍了 DTrace 推理函数。

表 4–1 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
`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一个推理标识符。在对 `speculate()` 函数的后续调用中将使用该推理标识符。值为零的推理标识符始终无效，但是可以传递给 `speculate()`、`commit()` 或 `discard()`。如果对 `speculation()` 的调用失败，则 `dtrace` 命令将生成类似于下例的一条消息。

```
dtrace: 2 failed speculations (no speculative buffer space available)
```

使用推理

要使用某个推理，请在任何数据记录操作之前使用一个子句将已从 `speculation()` 返回的标识符传递给 `speculate ()` 函数。将对包含 `speculate()` 的子句中的所有数据记录操

作进行推理跟踪。如果在 D 探测子句中对 speculate() 的调用位于数据记录操作之后，D 编译器将会生成编译时错误。子句中可以包含推理跟踪请求或非推理跟踪请求，但不能同时包含两者。

不能对聚集操作、破坏性操作和 exit 操作进行推理跟踪。如果试图在包含 speculate() 的子句中采用这其中的某种操作，将导致编译时错误。speculate() 函数不能跟在前一个 speculate() 函数之后。每个子句中只允许使用一个推理。只包含一个 speculate() 函数的子句将对缺省操作进行推理跟踪，缺省操作被定义为仅跟踪已经启用的探测器 ID。

speculation() 函数的典型用途是将 speculation() 函数的结果分配给某个线程本地变量。该线程本地变量充当其他探测器的后续谓词，以及 speculate() 的参数。

示例 4–5 speculation() 函数的典型用途

```
syscall::open:entry
{
    self->spec = speculation();
}

syscall:::
/self->spec/
{
    speculate(self->spec);
    printf("this is speculative");
}
```

提交推理

使用 commit() 函数提交推理。在提交推理缓冲区时，缓冲区的数据被复制到主体缓冲区中。如果推理缓冲区中的数据超出了主体缓冲区中的可用空间，则不会复制任何数据并且缓冲区的放弃计数会递增。如果已在多个 CPU 上对缓冲区进行了推理跟踪，则会立即复制正在提交的 CPU 上的推理数据，而其他 CPU 上的推理数据将在 commit() 之后的某个时间复制。

在每个基于 CPU 的推理缓冲区完全复制到其对应的基于 CPU 的主体缓冲区之前，正被提交的推理缓冲区不可供后续的 speculation() 调用使用。试图将 speculate() 函数调用结果写入到正被提交的缓冲区的后续尝试将放弃数据，且不生成错误。对 commit() 或 discard() 的后续调用也将失败且不生成错误。包含 commit() 函数的子句不能包含数据记录操作，但一个子句可以包含多个 commit() 调用来提交没有交集的缓冲区。

放弃推理

使用 discard() 函数放弃推理。如果推理仅在正在调用 discard() 函数的 CPU 上处于活动状态，则缓冲区将立即可供对 speculation() 函数的后续调用使用。如果推理在多个 CPU 上处于活动状态，则在调用 discard() 之后，放弃的缓冲区可供对 speculation() 函数的后续调用使用。如果在调用 speculation() 函数时没有可用的推理缓冲区，则会生成类似于下例的一条 dtrace 消息：

```
dtrace: 905 failed speculations (available buffer(s) still busy)
```

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。下面的示例显示了当 `open()` 失败时，open(2) 系统调用下的完整代码路径。

示例 4–6 specopen.d：失败的 open() 的代码流

```
#!/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)));
}

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);
}

syscall::open:return,
```

示例 4–6 specopen.d：失败的 open() 的代码流 （续）

```
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;
}
```

当运行前面的脚本时，该脚本将生成类似于下例的输出。

```
# ./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
```

索引