编程接口指南

文件号码 E38880
2013 年 1 月

ORACLE

130221@25097

目录

前言

《编程接口指南》介绍了应用程序开发者使用的 SunOS 5.10 网络接口和系统接口。

SunOS 5.10 是 Solaris 10 操作系统 (Solaris OS) 的核心，它符合 System V 接口说明 (System V Interface Description, SVID) 第三版以及单一 UNIX 规范版本 3 (Single UNIX Specification, version 3, SUSv3)。SunOS 5.10 与 UNIX System V 发行版 4 (System V, Release 4, SVR4) 完全兼容，并支持所有 System V 网络服务。

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、Xeon 和 Intel 64。支持的系统可以在 `http://www.sun.com/bigadmin/hcl/` 上的《Oracle Solaris OS: Hardware Compatibility Lists》（《Oracle Solaris OS：硬件兼容性列表》）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，这些与 x86 相关的术语表示以下含义：

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

若想了解本发行版支持哪些系统，请参见《Oracle Solaris OS: Hardware Compatibility Lists》（《Oracle Solaris OS：硬件兼容性列表》）。

目标读者

本书面向首次使用 SunOS 平台的程序员，或者希望进一步熟悉所提供的某些接口的程序员。《ONC+ Developer's Guide》介绍了适用于联网应用程序的其他接口和工具。

本手册假定您具备基本的编程能力、具有 C 语言编程的实际工作经验，并熟悉 UNIX 操作系统，尤其是对联网概念比较熟悉。有关 UNIX 联网基础知识的更多信息，请参见 W. Richard Stevens 编著的《UNIX Network Programming》第二版（Upper Saddle River，Prentice Hall，1998 年出版）。

本手册的结构

以下各章介绍了 Solaris OS 平台基本系统接口和基本网络接口的服务和功能。

第 1 章，内存和 CPU 管理介绍了用于创建和管理内存映射、执行高性能文件 I/O 以及控制内存管理其他方面的接口。

第 2 章，用于 Solaris Cluster 的远程共享内存 API 介绍了用于远程共享内存的应用编程接口 (Application Programming Interface, API) 框架和库函数。

第 3 章，会话描述协议 API 介绍了用于在 Solaris 中实现会话描述协议 (Session Description Protocol, SDP) 的 API 框架和库函数。

第 4 章，进程调度程序介绍了 SunOS 进程调度程序的操作、调度程序行为的修改、调度程序与进程管理接口的交互操作以及性能影响。

第 5 章，地址组 API 介绍了用于控制地址组的行为和结构以及这些组中线程的资源优先级的接口。

第 6 章，输入/输出接口介绍了基本缓冲文件 I/O 和旧式缓冲文件 I/O 以及有关 I/O 的其他内容。

第 7 章，进程间通信介绍了旧式的非联网进程间通信。

第 8 章，套接字接口介绍了如何使用套接字，套接字是联网通信的基本模式。

第 9 章，使用 XTI 和 TLI 编程介绍了如何使用 XTI 和 TLI 进行与传输无关的联网通信。

第 10 章，包过滤钩子介绍了用于在内核级别开发网络解决方案的接口，这些解决方案包括安全（包过滤和防火墙）解决方案和网络地址转换 (Network Address Translation, NAT) 解决方案等。

第 11 章，传输选择和名称到地址映射介绍了应用程序用来选择网络传输及其配置的网络选择机制。

第 12 章，实时编程和管理介绍了 SunOS 环境中的实时编程工具及其用法。

第 13 章，Solaris ABI 和 ABI 工具介绍了 Solaris 应用程序二进制接口 (Application Binary Interface, ABI) 以及用于验证应用程序与 Solaris ABI（`appcert` 和 `apptrace`）符合性的工具。

附录 A，UNIX 域套接字介绍了 UNIX 域套接字。

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/)

Oracle 欢迎您提出意见

Oracle 致力于提高其文档的质量，并十分乐意收到您的意见和建议。要分享您的意见，请访问 http://www.oracle.com/technetwork/indexes/documentation/index.html 并单击 "Feedback"。

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:`
aabbcc123	要使用实名或值替换的命令行占位符	删除文件的命令为 `rm` *filename*。
AaBbCc123	保留未译的新词或术语以及要强调的词	这些称为 *Class* 选项。 **注意**：有些强调的项目在联机时以粗体显示。
新词术语强调	新词或术语以及要强调的词	**高速缓存**是存储在本地的副本。 请**勿**保存文件。
《书名》	书名	阅读《用户指南》的第 6 章。

命令中的 shell 提示符示例

下表显示了 Oracle Solaris OS 中包含的缺省 UNIX shell 系统提示符和超级用户提示符。请注意，在命令示例中显示的缺省系统提示符可能会有所不同，具体取决于 Oracle Solaris 发行版。

表 P–2 shell 提示符

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

◆ ◆ ◆ 第 1 章

1 内存和 CPU 管理

本章从应用程序开发者的角度介绍了如何在 Solaris 操作系统中管理虚拟内存和 CPU。

- 第 15 页中的 “内存管理接口”介绍了接口和高速缓存控制。
- 第 17 页中的 “库级别动态内存”介绍了库级别动态内存分配和调试。
- 第 19 页中的 “其他内存控制接口”介绍了其他内存控制接口。
- 第 20 页中的 “CPU 性能计数器”介绍了 CPU 性能计数器 (CPU Performance Counter, CPC) 的用法。

内存管理接口

应用程序可通过多组接口使用虚拟内存功能。本节概述了这些接口，另外还提供了接口用法的示例。

创建和使用映射

mmap(2) 可用于建立指定的文件系统对象到进程地址空间的映射。指定的文件系统对象也可部分映射到进程地址空间中。此基本内存管理接口非常简单。请使用 open(2) 打开文件，接着使用 mmap(2) 创建具有适当访问权限和共享选项的映射，随后再继续执行应用程序。

mmap(2) 所建立的映射可针对指定的地址范围替换先前所有的映射。

标志 MAP_SHARED 和 MAP_PRIVATE 可指定映射的类型。必须指定一种映射类型。如果设置了 MAP_SHARED 标志，则写入操作会修改映射对象。无需对该对象进一步执行任何操作即可进行更改。如果设置了 MAP_PRIVATE 标志，则首次对映射区域进行写入操作将创建一个页面副本。所有进一步的写入操作都会引用该副本。只有修改了的页面会被复制。

映射类型可以跨 fork(2) 保留。

通过 mmap(2) 建立映射之后，将不再使用调用中所用的文件描述符。如果关闭文件，则映射在 munmap(2) 将其撤消之前会一直保留。创建新映射可以替换现有映射。

通过 truncate 调用可以截短映射文件。尝试访问不再存在的文件区域会导致产生 SIGBUS 信号。

映射 /dev/zero 可为调用程序提供零填充的虚拟内存块。块的大小在调用 mmap(2) 时指定。以下代码段说明了如何使用此技术在程序中创建零填充的存储块。块的地址由系统进行选择。

```
removed to fr.ch4/pl1.create.mapping.c
```

某些设备或文件仅当通过映射对其进行访问时才有用。用于支持位图显示的帧缓存器设备便是此现象的一个示例。显示管理算法直接对显示地址执行操作时，实现这些算法会简单得多。

删除映射

munmap(2) 可用于删除调用进程的指定地址范围内的所有页面映射。munmap(2) 对已映射的对象没有任何影响。

高速缓存控制

SunOS 中的虚拟内存系统是一个高速缓存系统，处理器内存可在其中缓冲文件系统对象中的数据。系统提供了一些接口，用于控制或询问高速缓存的状态。

使用 mincore

mincore(2) 接口可确定内存页是否驻留在指定范围内的映射所涵盖的地址空间中。由于页面状态可能会在 mincore 检查页面之后和 mincore 返回数据之前发生更改，因此返回的信息可能会过时。只能保证将锁定的页面保留在内存中。

使用 mlock 和 munlock

mlock(3C) 会导致将指定地址范围内的页面锁定在物理内存中。在此进程或其他进程中引用锁定页面不会导致缺页而需要执行 I/O 操作。由于此 I/O 操作会干扰虚拟内存的正常操作，并且会降低其他进程的速度，因此仅有超级用户才能使用 mlock。内存中可锁定页数的限制取决于系统配置。如果超过此限制，则调用 mlock 将失败。

munlock 可用于释放对物理页的锁定。如果对单个映射的地址范围进行多次 mlock 调用，则一次 munlock 调用即可释放锁定。不过，如果 mlock 锁定了对相同页面的不同映射，则在释放对所有映射的锁定之前，不会解除对这些页面的锁定。

删除映射也会释放锁定，方法是：通过 mmap(2) 操作来替换或通过 munmap(2) 来删除。

与 MAP_PRIVATE 映射关联的写复制事件会将对源页面的锁定传递到目标页面。这样，对包括 MAP_PRIVATE 映射的地址范围的锁定便会以透明方式与写复制重定向操作一起保留。有关此重定向的讨论，请参见第 15 页中的 “创建和使用映射”。

使用 mlockall 和 munlockall

mlockall(3C) 和 munlockall(3C) 与 mlock 和 munlock 类似，但是 mlockall 和 munlockall 针对整个地址空间执行操作。mlockall 用于设置对地址空间中所有页面的锁定，munlockall 用于删除对该地址空间中所有页面的锁定，无论是通过 mlock 还是 mlockall 建立操作均如此。

使用 msync

msync(3C) 会导致指定地址范围内的所有已修改的页面都刷新到这些地址所映射的对象。此命令与 fsync(3C) 类似，后者对文件执行操作。

库级别动态内存

库级别动态内存分配为动态内存分配提供了一个易于使用的接口。

动态内存分配

最常用的接口包括：

- malloc(3C)
- free(3C)
- calloc(3C)
- watchmalloc(3MALLOC)

其他动态内存分配接口包括 memalign(3C)、valloc(3C) 和 realloc(3C)。

- malloc 会返回一个指针，该指针指向的内存块至少和所请求的内存量一样大。内存块会进行对齐以存储任何类型的数据。
- free 可用于将通过 malloc、calloc、realloc、memalign 或 valloc 获取的内存返回到系统内存。尝试释放不是由动态内存分配接口所保留的块是一个错误，可能会导致进程崩溃。
- calloc 可用于返回一个指向初始化为零的内存块的指针。可以通过 watchmalloc 或 free 将 calloc 保留的内存返回到系统。内存会进行分配并对齐，以包含具有指定大小的指定元素数的数组。
- memalign 可基于指定的对齐边界分配指定的字节数。对齐边界必须是 2 的幂。
- valloc 可分配在页边界上对齐的指定字节数。

- realloc 可用于更改分配给进程的内存块的大小，使用 realloc 可增大或减小分配的内存块的大小。realloc 是缩减内存分配而不会导致问题的唯一方法。重新分配的块在内存中的位置可能会更改，但是在分配大小更改之前的内容将保持不变。

动态内存调试

Sun WorkShop 工具包有助于查找和消除在使用动态内存时出现的错误。Sun WorkShop 的运行时检查 (Run Time Checking, RTC) 工具使用本节中介绍的函数来查找使用动态内存时的错误。

RTC 不需要使用 -g 对程序进行编译，即可查找所有错误。不过，有时需要符号 (-g) 信息，以保证某些错误（尤其是从未初始化内存中读取的错误）的正确性。因此，如果没有可用的符号信息，则某些错误会受到抑制。这些错误包括 rui（针对 a.out）和 rui + aib + air（针对共享库）。可以使用 suppress 和 unsuppress 来更改此行为。

check -access

-access 选项可用于启动访问检查。RTC 会报告以下错误：

baf　可用项错误

duf　可用项重复

maf　可用项排列错误

mar　读取排列错误

maw　写入排列错误

oom　内存不足

rua　从未分配的内存读取

rui　从未初始化的内存读取

rwo　写入到只读内存

wua　写入到未分配的内存

缺省行为是在检测到每个访问错误之后停止进程。可以使用 rtc_auto_continue dbxenv 变量更改此行为。如果设置为 on，则 RTC 会将访问错误记录到一个文件中。此文件名由 rtc_error_log_file_name dbxenv 变量的值确定。缺省情况下，仅在每个唯一的访问错误首次发生时报告该错误。可使用 rtc_auto_suppress dbxenv 变量更改此行为。此变量的缺省设置为 on。

check -leaks [-frames *n*] [-match *m*]

-leaks 选项可启用泄漏检查。RTC 会报告以下错误：

aib　可能发生内存泄漏－唯一指针指向块中间位置

`air` 可能发生内存泄漏－指向块的指针仅存在于寄存器中

`mel` 内存泄漏－无指向块的指针

启用泄漏检查后，在程序退出时会获取自动生成的泄漏报告。此时会报告包括潜在泄漏在内的所有泄漏。缺省情况下，将生成非详细报告。此缺省行为由 `dbxenv rtc_mel_at_exit` 控制。不过，可以随时要求提供泄漏报告。

报告泄漏时，`-frames` *n* 变量最多可显示 *n* 个不同的栈帧。`-match` *m* 变量用于合并泄漏。如果进行分配时两个或多个泄漏的调用栈与 *m* 个帧匹配，则会在单个合并的泄漏报告中报告这些泄漏。*n* 的缺省值为 8 或 *m* 值之间的较大者。*n* 的最大值为 16。*m* 的缺省值为 2。

check -memuse [-frames *n*] [-match *m*]

`-memuse` 选项可启用内存使用 (`memuse`) 检查。使用 `check -memuse` 即表示使用 `check -leaks`。除程序退出时获取泄漏报告之外，您还会获得一个列出使用中的块 (`biu`) 的报告。缺省情况下，将生成有关使用中的块的非详细报告。此缺省行为由 `dbxenv rtc_biu_at_exit` 控制。在程序执行过程中，可随时查看程序中的内存所分配到的位置。

`-frames` *n* 和 `-match` *m* 变量的作用如下节所述。

check -all [-frames *n*] [-match *m*]

与 `check -access`；`check -memuse [-frames` *n*`] [-match` *m*`]` 等效。`rtc_biu_at_exitdbxenv` 变量的值不会随 `check -all` 进行更改。因此，缺省情况下，退出时不会生成任何内存使用报告。

check [*funcs*] [*files*] [*loadobjects*]

与 *funcs files loadobjects* 中的 `check -all`；`suppress all`；`unsuppressall` 等效。使用此选项可以将 RTC 重点用于所需的位置。

其他内存控制接口

本节讨论其他内存控制接口。

使用 sysconf

`sysconf(3C)` 可用于返回与系统相关的内存页的大小。为便于移植，应用程序不应嵌入用于指定页面大小的任何常量。请注意，即使在相同指令集的执行中，变化的页面大小也很常见。

使用 mprotect

mprotect(2) 可用于为指定地址范围内的所有页面分配指定的保护。保护不能超出底层对象所允许的权限。

使用 brk 和 sbrk

中断点 (break) 是进程映像中栈外部的最大的有效数据地址。程序开始执行时，execve(2) 通常会将中断点 (break) 值设置为程序及其数据存储所定义的最大地址。

使用 brk(2) 可将中断点 (break) 设置为更大的地址。您还可以使用 sbrk(2) 向进程的数据段中添加一个存储增量。通过调用 getrlimit(2) 可以获取数据段的最大可能大小。

```
caddr_t
brk(caddr_t addr);

caddr_t
sbrk(intptr_t incr);
```

brk 可用于将调用者未使用的最低数据段位置标识为 *addr*。此位置会向上舍入为系统页面大小的下一个倍数。

备用接口 sbrk 可用于向调用者数据空间中添加 *incr* 个字节，并返回指向新数据区域开头的指针。

CPU 性能计数器

本节讨论了使用 CPU 性能计数器 (CPU Performance Counter, CPC) 时用到的开发者接口。Solaris 应用程序可以独立于底层计数器体系结构来使用 CPC。

向 libcpc 中添加的 API

本节介绍了 libcpc(3LIB) 库的最新添加内容。有关旧接口的信息，请参见 libcpc 手册页。

初始化接口

准备使用 CPC 工具的应用程序可通过调用 cpc_open() 函数来初始化库。此函数会返回一个供其他接口使用的 cpc_t * 参数。cpc_open() 函数的语法如下：

```
cpc_t*cpc_open(intver);
```

ver 参数的值用于标识应用程序所使用的接口的版本。如果底层计数器无法访问或不可用，则 cpc_open() 函数将失败。

硬件查询接口

```
uint_t cpc_npic(cpc_t *cpc);
uint_t cpc_caps(cpc_t *cpc);
void cpc_walk_events_all(cpc_t *cpc, void *arg,
          void (*action)(void *arg, const char *event));
void cpc_walk_events_pic(cpc_t *cpc, uint_t picno, void *arg,
          void(*action)(void *arg, uint_t picno, const char *event));
void cpc_walk-attrs(cpc_t *cpc, void *arg,
          void (*action)(void *arg, const char *attr));
```

`cpc_npic()` 函数可返回底层处理器上的物理计数器数量。

`cpc_caps()` 函数可返回 *uint_t* 参数，此参数的值是基于底层处理器支持的功能执行按位或运算的结果。共有两项功能。`CPC_CAP_OVERFLOW_INTERRUPT` 功能，允许处理器在计数器溢出时产生中断；`CPC_CAP_OVERFLOW_PRECISE` 功能，允许处理器确定哪一个计数器产生溢出中断。

内核可维护底层处理器支持的事件的列表。单个芯片上的不同物理计数器不必使用相同的事件列表。`cpc_walk_events_all()` 函数可针对每个处理器支持的事件调用 `action()` 例程，而不用考虑物理计数器。`cpc_walk_events_pic()` 函数可针对特定物理计数器上每个处理器支持的事件调用 `action()` 例程。这两个函数都会将未解释的 *arg* 参数从调用者传递到每个 `action()` 函数调用。

平台可维护底层处理器支持的属性的列表。利用这些属性，可以访问特定于处理器的高级性能计数器功能。`cpc_walk_attrs()` 函数可针对每个属性名称调用操作例程。

配置接口

```
cpc_set_t *cpc_set_create(cpc_t *cpc);
int cpc_set_destroy(cpc_t *cpc, cpc_set_t *set);
int cpc_set_add_request(cpc_t *cpc, cpc_set_t *set, const char *event,
          uint64_t preset, uint_t flags, uint_t nattrs,
          const cpc_attr_t *attrs);
int cpc_set_request_preset(cpc_t *cpc, cpc_set_t *set, int index,
          uint64_t preset);
```

不透明数据类型 `cpc_set_t` 表示请求的集合。这些集合称为集。`cpc_set_create()` 函数可创建一个空集。`cpc_set_destroy()` 函数可销毁一个集，并释放由该集使用的所有内存。销毁一个集可释放由该集使用的硬件资源。

`cpc_set_add_request()` 函数可向一个集中添加请求。以下列表介绍了请求所使用的参数。

`event` 一个字符串，用于指定要进行计数的事件的名称。

`preset` 一个 64 位无符号整数，用作计数器的初始值。

`flags` 应用于一组请求标志的逻辑或运算的结果。

`nattrs` `attrs` 指向的数组中的属性个数。

attrs　　指向 cpc_attr_t 结构数组的指针。

以下列表介绍了有效的请求标志。

CPC_COUNT_USER　　利用此标志，当 CPU 在用户模式下执行时可对出现的事件进行计数。

CPC_COUNT_SYSTEM　　通过此标志，可以对 CPU 在特权模式下执行时发生的事件进行计数。

CPC_OVF_NOTIFY_EMT　　此标志可请求在硬件计数器溢出时发出通知。

CPC 接口可将属性作为 cpc_attr_t 结构数组来进行传递。

cpc_set_add_request() 函数成功返回时，它将返回一个索引。此索引会引用通过调用 cpc_set_add_request() 函数时添加的请求所生成的数据。

cpc_set_request_preset() 函数可更改请求的预设值。通过更改可将新的预设值与溢出集重新绑定在一起。

cpc_walk_requests() 函数可针对 cpc_set_t 中的每个请求调用用户提供的 action() 例程。*arg* 参数的值会在未进行解释的情况下传递给用户例程。通过 cpc_walk_requests() 函数，应用程序可显示集中每个请求的配置。cpc_walk_requests() 函数的语法如下：

```
void cpc_walk_requests(cpc_t *cpc, cpc_set_t *set, void *arg,
void (*action)(void *arg, int index, const char *event,
uint64_t preset, uint_t flags, int nattrs,
            const cpc_attr_t *attrs));
```

绑定

本节中的接口可将集中的请求绑定到物理硬件，并将计数器设置到起始位置。

```
int cpc_bind_curlwp(cpc_t *cpc, cpc_set_t *set, uint_t flags);
int cpc_bind_pctx(cpc_t *cpc, pctx_t *pctx, id_t id, cpc_set_t *set,
          uint_t flags);
int cpc_bind_cpu(cpc_t *cpc, processorid_t id, cpc_set_t *set,
          uint_t flags);
int cpc_unbind(cpc_t *cpc, cpc_set_t *set);
```

cpc_bind_curlwp() 函数可将集绑定到调用 LWP。该集的计数器将虚拟化为到此 LWP 上，并计算调用 LWP 运行时 CPU 上发生的事件的数量。cpc_bind_curlwp() 例程唯一的有效标志为 CPC_BIND_LWP_INHERIT。

cpc_bind_pctx() 函数可将集绑定到使用 libpctx(3LIB) 捕获的进程中的 LWP。此函数没有任何有效标志。

cpc_bind_cpu() 函数可将集绑定到在 *id* 参数中指定的处理器。将集绑定到 CPU 会使系统中的现有性能计数器上下文无效。此函数没有任何有效标志。

cpc_unbind() 函数可停止性能计数器，并释放与绑定集关联的硬件。如果将集绑定到 CPU，则 cpc_unbind() 函数将从该 CPU 解除绑定 LWP 并释放 CPC 伪设备。

抽样

利用本节中介绍的接口，可以将数据从计数器返回到应用程序。计数器数据驻留在名为 cpc_buf_t 的不透明数据结构中。此数据结构可用于捕获绑定集正在使用的计数器的状态快照，并且包括以下信息：

- 每个计数器的 64 位值
- 最新硬件快照的时间戳
- 累积 CPU 周期计数器，用于计算处理器针对绑定集使用的 CPU 周期的数量

```
cpc_buf_t *cpc_buf_create(cpc_t *cpc, cpc_set_t *set);
int cpc_buf_destroy(cpc_t *cpc, cpc_buf_t *buf);
int cpc_set_sample(cpc_t *cpc, cpc_set_t *set, cpc_buf_t *buf);
```

cpc_buf_create() 函数可创建一个缓冲区，用来存储 cpc_set_t 所指定的集中的数据。cpc_buf_destroy() 函数可释放与给定 cpc_buf_t 相关联的内存。cpc_buf_sample() 函数可获取正在代表指定集进行计数的计数器的快照。指定集必须已绑定，并且已创建了缓冲区，然后才能调用 cpc_buf_sample() 函数。

对缓冲区抽样不会更新与该集相关联的请求的预设值。如果使用 cpc_buf_sample() 函数对缓冲区进行抽样，然后解除绑定并重新绑定，则会从最初调用 cpc_set_add_request() 函数时该请求的预设值开始计数。

缓冲区操作

通过以下例程，可访问 cpc_buf_t 结构中的数据。

```
int cpc_buf_get(cpc_t *cpc, cpc_buf_t *buf, int index, uint64_t *val);
int cpc_buf_set(cpc_t *cpc, cpc_buf_t *buf, int index, uint64_t *val);
hrtime_t cpc_buf_hrtime(cpc_t *cpc, cpc_buf_t *buf);
uint64_t cpc_buf_tick(cpc_t *cpc, cpc_buf_t *buf);
int cpc_buf_sub(cpc_t *cpc, cpc_buf_t *result, cpc_buf_t *left
      cpc_buf_t *right);
int cpc_buf_add(cpc_t *cpc, cpc_buf_t *result, cpc_buf_t *left,
      cpc_buf_t *right);
int cpc_buf_copy(cpc_t *cpc, cpc_buf_t *dest, cpc_buf_t *src);
void cpc_buf_zero(cpc_t *cpc, cpc_buf_t *buf);
```

cpc_buf_get() 函数可检索 *index* 参数所标识的计数器的值。*index* 参数是指在绑定集之前由 cpc_set_add_request() 函数返回的值。cpc_buf_get() 函数可将计数器的值存储在 *val* 参数所指示的位置。

cpc_buf_set() 函数可设置 *index* 参数所标识的计数器的值。*index* 参数是指在绑定集之前由 cpc_set_add_request() 函数返回的值。cpc_buf_set() 函数可将计数器的值设置为 *val* 参数所指示的位置处的值。cpc_buf_get() 函数和 cpc_buf_set() 函数均不会更改相应 CPC 请求的预设值。

cpc_buf_hrtime() 函数可返回高精度时间戳，用来指示对硬件抽样的时间。cpc_buf_tick() 函数可返回 LWP 运行时所经过的 CPU 时钟周期数。

cpc_buf_sub() 函数可计算在 *left* 和 *right* 参数中指定的计数器与周期值之间的差值。cpc_buf_sub() 函数可将结果存储在 *result* 中。给定的 cpc_buf_sub() 函数调用中的所有 cpc_buf_t 值都必须源自同一个 cpc_set_t 结构。*result* 索引包含缓冲区中每个请求索引的 *left* - *right* 计算的结果。结果索引还包含 *tick* 的差。cpc_buf_sub() 函数可用于将目标缓冲区的高精度时间戳设置为 *left* 或 *right* 缓冲区的最新时间。

cpc_buf_add() 函数可计算在 *left* 和 *right* 参数中指定的计数器和周期值的总和。cpc_buf_add() 函数可将结果存储在 *result* 中。给定的 cpc_buf_add() 函数调用中的所有 cpc_buf_t 值都必须源自同一个 cpc_set_t 结构。*result* 索引包含缓冲区中每个请求索引的 *left* + *right* 计算的结果。结果索引还包含 *tick* 的总和。cpc_buf_add() 函数可用于将目标缓冲区的高精度时间戳设置为 *left* 或 *right* 缓冲区的最新时间。

使用 cpc_buf_copy() 函数时 *dest* 与 *src* 相同。

cpc_buf_zero() 函数可将 *buf* 中的所有内容都设置为零。

激活接口

本节介绍了 CPC 的激活接口。

```
int cpc_enable(cpc_t *cpc);
int cpc_disable(cpc_t *cpc);
```

这两个接口分别用于启用和禁用与正在执行的 LWP 绑定的任何集的计数器。利用这些接口，应用程序可以指定所需的代码，同时使用 libpctx 将计数器配置延迟到某个控制进程。

错误处理接口

本节介绍了 CPC 的错误处理接口。

```
typedef void (cpc_errhndlr_t)(const char *fn, int subcode, const char *fmt,
          va_list ap);
void cpc_seterrhndlr(cpc_t *cpc, cpc_errhndlr_t *errhndlr);
```

通过这两个接口，可以传递 cpc_t 句柄。除了字符串之外，cpc_errhndlr_t 句柄还采用一个整型子代码。整型 *subcode* 用于描述 *fn* 参数所引用的函数遇到的特定错误。通过整型 *subcode*，应用程序可轻易识别各种错误情况。*fmt* 参数的字符串值包含对错误子代码的国际化说明，并且适用于显示。

◆ ◆ ◆ 第 2 章

2 用于 Solaris Cluster 的远程共享内存 API

可以配置 Solaris Cluster OS 系统，以使用基于内存的互连（如 Dolphin-SCI）和分层系统软件组件。这些组件实现了一种用户级节点间通讯机制，此机制基于对驻留在远程节点上的内存进行直接访问。此机制称为远程共享内存 (Remote Shared Memory, RSM)。本章定义了 RSM 应用编程接口 (RSM Application Programming Interface, RSMAPI)。

- 第 26 页中的 “API 框架”介绍了 RSM API 框架。
- 第 27 页中的 “API 库函数”介绍了 RSM API 库函数。

共享内存模型概述

在共享内存模型中，应用程序进程会在其本地地址空间中创建一个 RSM 导出段。一个或多个远程应用程序进程创建 RSM 导入段，在导出段与导入段之间建立互连的虚拟连接。所有进程将使用其特定地址空间的本地地址，来实现对共享段的内存引用。

应用程序进程通过为 RSM 导出段分配可在本地寻址的内存来创建此导出段。使用标准 Solaris 接口（如 System V 共享内存、`mmap(2)` 或 `valloc(3C)`）之一即可实现此分配。然后，进程会调用 RSMAPI 来创建段，用于为已分配的内存提供引用句柄。RSM 段通过一个或多个互连控制器进行发布。可以远程访问已发布的段。另外，还将发布允许导入该段的节点的访问特权列表。

将为导出的段分配一个段 ID。通过该段 ID 及其创建进程的群集节点 ID，导入进程可唯一地指定一个导出段。如果成功创建了导出段，则会向进程返回一个段句柄，以便在后续段操作中使用。

应用程序进程通过使用 RSMAPI 来创建导入段，便可以对已发布的段进行访问。创建导入段之后，应用程序进程便建立了互连的虚拟连接。如果成功创建此导入段，则会向应用程序进程返回一个 RSM 导入段句柄，以便在后续段导入操作中使用。建立虚拟连接之后，如果互连支持内存映射，则应用程序可能会请求 RSMAPI 提供内存映射以进行本地访问。如果不支持内存映射，则应用程序可以使用 RSMAPI 提供的内存访问原语。

RSMAPI 提供了一种机制，该机制可以支持远程访问错误检测并能解决写入顺序内存模型问题。此机制称为**屏障**。

RSMAPI 提供一种通知机制，可以同步本地访问和远程访问。当导入进程完成数据写入操作后，导出进程便可以调用函数使自身阻塞。当导入进程完成写入后，此进程会通过调用信号函数来使导出进程解除阻塞。解除阻塞之后，导出进程便可以处理数据。

API 框架

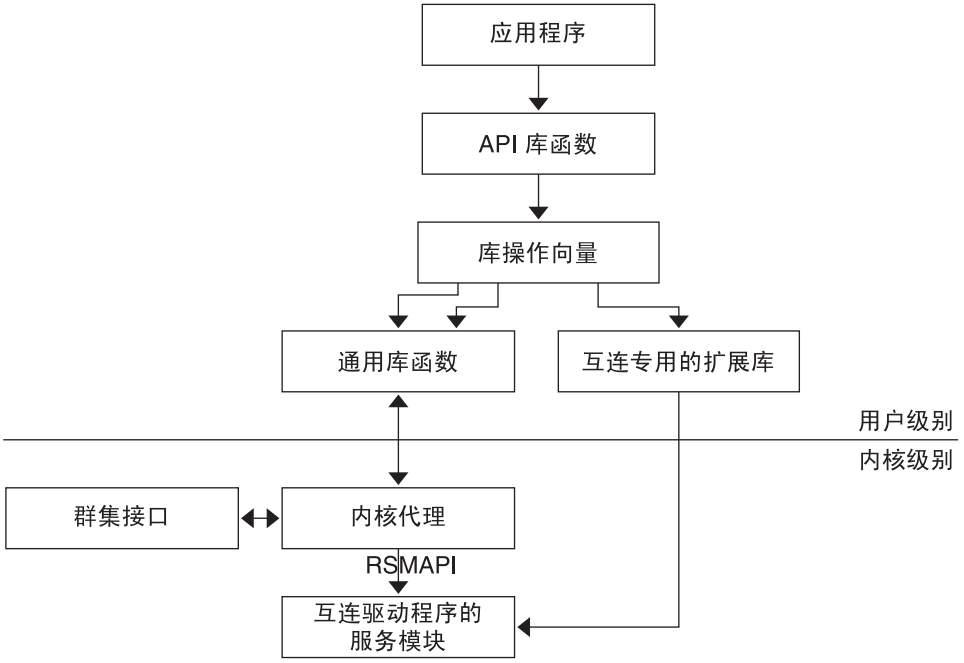
RSM 应用程序支持组件以软件包形式提供，如下所示：

- `SUNWrsm`
  - 导出 RSMAPI 函数的共享库 (`/usr/lib/librsm.so`)。
  - 内核代理 (Kernel Agent, KA) 伪设备驱动程序 (`/usr/kernel/drv/rsm`)，代表用户库通过 RSMAPI 接口与内存互连驱动程序连接。
  - 用于获取互连拓扑的群集接口模块。
- `SUNWrsmop`

  互连驱动程序服务模块 (`/kernel/misc/rsmops`)。
- `SUNWrsmdk`

  提供 API 函数和数据结构原型的头文件 (`/opt/SUNWrsmdk/include`)。
- `SUNW`*`interconnect`*

  系统中配置的可为特定互连提供 RSM 支持的可选 `librsm.so` 扩展。此扩展以库的形式 (`librsm`*`interconnect`*`.so`) 提供。



API 库函数

API 库函数支持以下操作：

- 互连控制器操作
- 群集拓扑操作
- 内存段操作，包括段管理和数据访问
- 屏障操作
- 事件操作

互连控制器操作

控制器操作提供了访问控制器的机制，还可以确定底层互连的特征。下面列出了有关控制器操作的信息：

- 获取控制器
- 获取控制器属性
- 释放控制器

rsm_get_controller

`int` **`rsm_get_controller`**(*char *name, rsmapi_controller_handle_t *controller*);

rsm_get_controller 操作可获取给定控制器实例（如 sci0 或 loopback）的控制器句柄。返回的控制器句柄用于后续 RSM 库调用。

返回值：如果成功，则返回 0。否则，返回错误值。

RSMERR_BAD_CTLR_HNDL	控制器句柄无效
RSMERR_CTLR_NOT_PRESENT	控制器不存在
RSMERR_INSUFFICIENT_MEM	内存不足
RSMERR_BAD_LIBRARY_VERSION	库版本无效
RSMERR_BAD_ADDR	地址错误

rsm_release_controller

int **rsm_release_controller**(*rsmapi_controller_handle_t chdl*);

此函数可释放与给定控制器句柄关联的控制器。每个 rsm_release_controller 调用都必须对应一个 rsm_get_controller。当与某个控制器关联的所有控制器句柄都被释放后，与此控制器关联的系统资源将被释放。尝试访问控制器句柄，或者尝试访问已释放控制器句柄上的导入段或导出段都是非法操作。执行此类尝试的结果是不确定的。

返回值：如果成功，则返回 0。否则，返回错误值。

RSMERR_BAD_CTLR_HNDL	控制器句柄无效

rsm_get_controller_attr

int **rsm_get_controller_attr**(*rsmapi_controller_handle_t chdl*, *rsmapi_controller_attr_t *attr*);

此函数可获取指定控制器句柄的属性。以下列出了此函数当前已定义的属性：

```
typedef struct {
     uint_t       attr_direct_access_sizes;
     uint_t       attr_atomic_sizes;
     size_t       attr_page_size;
     size_t       attr_max_export_segment_size;
     size_t       attr_tot_export_segment_size;
     ulong_t      attr_max_export_segments;
     size_t       attr_max_import_map_size;
     size_t       attr_tot_import_map_size;
     ulong_t      attr_max_import_segments;
 } rsmapi_controller_attr_t;
```

返回值：如果成功，则返回 0。否则，返回错误值。

RSMERR_BAD_CTLR_HNDL	控制器句柄无效
RSMERR_BAD_ADDR	地址错误

群集拓扑操作

导出操作和导入操作所需的关键互连数据包括：

- 导出群集节点 ID
- 导入群集节点 ID
- 控制器名称

作为基本约束，为段导入指定的控制器必须与用于关联段导出的控制器具有物理连接。此接口定义互连拓扑，该拓扑有助于应用程序建立有效的导出和导入策略。所提供的数据包括本地节点 ID、本地控制器实例名称以及每个本地控制器的远程连接规范。

导出内存的应用程序组件可以使用此接口提供的数据来查找现有本地控制器集。此接口提供的数据还可用于正确分配控制器，以便创建和发布段。应用程序组件可以通过与硬件互连和应用程序软件分发一致的控制器集来有效分发导出的段。

必须向要导入内存的应用程序组件通知内存导出中所用的段 ID 和控制器。通常，此信息通过预定义的段和控制器对进行传送。导入组件可以使用拓扑数据来确定适用于段导入操作的控制器。

rsm_get_interconnect_topology

`int` **`rsm_get_interconnect_topology`** (*rsm_topology_t **topology_data*);

此函数返回一个指针，该指针指向应用程序指针所指定位置的拓扑数据。下面定义了拓扑数据结构。

返回值：如果成功，则返回 0。否则，返回错误值。

`RSMERR_BAD_TOPOLOGY_PTR`	拓扑指针无效
`RSMERR_INSUFFICIENT_MEM`	内存不足
`RSMERR_BAD_ADDR`	内存不足

rsm_free_interconnect_topology

`void` **`rsm_free_interconnect_topology`** (*rsm_topology_t *topology_data*);

`rsm_free_interconnect_topology` 操作可释放通过 `rsm_get_interconnect_topology` 分配的内存。

返回值：无。

数据结构

从 rsm_get_topology_data 返回的指针会引用 rsm_topology_t structure。此结构为每个本地控制器提供本地节点 ID 以及指向 connections_t 结构的指针数组。

```
typedef struct rsm_topology {
   rsm_nodeid_t    local_nodeid;
   uint_t          local_cntrl_count;
   connections_t   *connections[1];
} rsm_topology_t;
```

管理操作

RSM 段 ID 可以由应用程序指定，或者由系统使用 rsm_memseg_export_publish() 函数生成。指定段 ID 的应用程序需要使用保留范围的段 ID。要保留一组段 ID，请使用 rsm_get_segmentid_range 并在段 ID 配置文件 /etc/rsm/rsm.segmentid 中定义保留范围的段 ID。应用程序可以使用 rsm_get_segmentid_range 函数来获取为应用程序保留的段 ID 范围。此函数会读取在 /etc/rsm/rsm.segmentid 文件中针对给定应用程序 ID 定义的段 ID 范围。

应用程序 ID 是指用于标识应用程序的以空字符结尾的字符串。应用程序可以使用任何等于或大于 baseid 并且小于 baseid+length 的值。如果修改了 baseid 或 length，则返回到应用程序的段 ID 可能会超出保留的范围。要避免此问题，请使用处于保留的段 ID 范围内的偏移来获取段 ID。

/etc/rsm/rsm.segmentid 文件中的各项形式如下：

```
#keyword      appid      baseid       length
reserve       SUNWfoo    0x600000     100
```

这些项由可以用制表符或空格分隔的字符串组成。第一个字符串是关键字 reserve，后跟应用程序标识符（不包含空格的字符串）。应用程序标识符之后是 baseid，即保留范围的起始段 ID（十六进制）。 baseid 之后是 length，即保留的段 ID 数。注释行的第一列中包含 #。此文件不应包含空行。为系统保留的段 ID 在 /usr/include/rsm/rsm_common.h 头文件中定义。应用程序不能使用为系统保留的段 ID。

rsm_get_segmentid_range 函数返回 0 表示成功。如果此函数失败，则会返回以下错误值之一：

RSMERR_BAD_ADDR	传递的地址无效
RSMERR_BAD_APPID	未在 /etc/rsm/rsm.segmentid 文件中定义应用程序 ID
RSMERR_BAD_CONF	配置文件 /etc/rsm/rsm.segmentid 不存在或无法读取。文件格式配置错误

内存段操作

通常，RSM 段表示一组映射到连续虚拟地址范围的非连续物理内存页。通过 RSM 段导出和段导入操作，可以在互连系统之间共享物理内存区域。物理页所在节点的进程称为内存的**导出者**。为远程访问发布的导出段将具有给定节点所特有的段标识符。段 ID 可以由导出者指定或由 RSMAPI 框架分配。

互连节点的进程通过创建 RSM 导入段来对导出的内存进行访问。RSM 导入段与一个导出段连接，而不是与本地物理页连接。如果互连支持内存映射，则导入者可以使用导入段的本地内存映射地址来读写导出的内存。如果互连不支持内存映射，则导入进程会使用内存访问原语。

导出端内存段操作

导出内存段时，应用程序首先通过常规操作系统接口（如 System V 共享内存接口、`mmap` 或 `valloc`）来分配其虚拟地址空间中的内存。分配内存之后，应用程序将调用 RSMAPI 库接口来创建和标记段。标记段之后，RSMAPI 库接口将物理页绑定到已分配的虚拟范围。绑定物理页之后，RSMAPI 库接口会发布段以供导入进程访问。

注 – 如果虚拟地址空间是使用 `mmap` 获取的，则映射必须为 `MAP_PRIVATE`。

导出端内存段操作包括：

- 创建和销毁内存段
- 发布和取消发布内存段
- 重新绑定内存段的后备存储

创建和销毁内存段

使用 `rsm_memseg_export_create` 建立新内存段可以在创建时将物理内存与该段进行关联。此操作将返回新内存段的导出端内存段句柄。段在创建进程的生命周期内一直存在，或者在使用 `rsm_memseg_export_destroy` 销毁该段之前一直存在。

注 – 如果在导入端断开连接之前执行销毁操作，则会强制断开连接。

创建段

`int` **`rsm_memseg_export_create`**`(`*`rsmapi_controller_handle_t controller`*`,`
`rsm_memseg_export_handle_t *memseg``,` *`void *vaddr`*`,` *`size_t size`*`,` *`uint_t flags`*`);`

此函数可用于创建段句柄。创建段句柄之后，段句柄会绑定到指定的虚拟地址范围 [vaddr..vaddr+size]。此范围必须有效并基于控制器的 alignment 属性对齐。flags 参数是位掩码，可用于执行以下操作：

- 解除绑定段
- 重新绑定段
- 将 RSM_ALLOW_REBIND 传递给 flags
- 支持锁定操作
- 将 RSM_LOCK_OPS 传递给 flags

注 – RSMAPI 的初始发行版中不包括 RSM_LOCK_OPS 标志。

返回值：如果成功，则返回 0。否则，返回错误值。

RSMERR_BAD_CTLR_HNDL	控制器句柄无效
RSMERR_CTLR_NOT_PRESENT	控制器不存在
RSMERR_BAD_SEG_HNDL	段句柄无效
RSMERR_BAD_LENGTH	长度为零或长度超出控制器限制
RSMERR_BAD_ADDR	地址无效
RSMERR_PERM_DENIED	权限被拒绝
RSMERR_INSUFFICIENT_MEM	内存不足
RSMERR_INSUFFICIENT_RESOURCES	资源不足
RSMERR_BAD_MEM_ALIGNMENT	地址未在页边界上对齐
RSMERR_INTERRUPTED	操作被信号中断

销毁段

int **rsm_memseg_export_destroy**(*rsm_memseg_export_handle_t memseg*);

此函数可用于取消分配段及其可用资源。将强制断开与所有导入进程的连接。

返回值：如果成功，则返回 0。否则，返回错误值。

RSMERR_BAD_SEG_HNDL	段句柄无效
RSMERR_POLLFD_IN_USE	pollfd 正在使用

发布、重新发布和取消发布内存段

通过发布操作，其他互连节点可以导入内存段。一个导出段可能会在多个互连适配器上发布。

段 ID 可以在授权范围内指定或指定为零，此时 RSMAPI 框架会生成有效的段 ID 并传递回该段 ID。

段访问控制列表由多对节点 ID 和访问权限组成。对于列表中指定的每个节点 ID，关联的读/写权限会通过三个八进制数字提供给所有者、组和其他用户，这与 Solaris 文件权限一样。在访问控制列表中，每个八进制数字都可以具有以下值：

2　写入访问。

4　只读访问。

6　读写访问。

访问权限值 0624 可指定以下访问类型：

- 与导出者具有相同 `uid` 的导入者具有读写访问权限。
- 与导出者具有相同 `gid` 的导入者仅有写入访问权限。
- 所有其他导入者仅有读取访问权限。

提供访问控制列表之后，未包含在此列表中的节点不能导入段。但是，如果访问列表为空，则任何节点都可导入段。所有节点的访问权限等同于导出进程的所有者/组/其他用户文件创建权限。

注 – 节点应用程序负责管理段标识符的分配，从而确保导出节点的唯一性。

发布段

`int` **`rsm_memseg_export_publish`**`(`*`rsm_memseg_export_handle_t memseg, rsm_memseg_id_t *segment_id, rsmapi_access_entry_t ACCESS_list[], uint_t access_list_length``);`

```
typedef struct {
    rsm_node_id_t       ae_node;    /* remote node id allowed to access resource */
    rsm_permission_t    ae_permissions;    /* mode of access allowed */
} rsmapi_access_entry_t;.
```

返回值：如果成功，则返回 0。否则，返回错误值。

`RSMERR_BAD_SEG_HNDL`	段句柄无效
`RSMERR_SEG_ALREADY_PUBLISHED`	段已发布
`RSMERR_BAD_ACL`	访问控制列表无效
`RSMERR_BAD_SEGID`	段标识符无效
`RSMERR_SEGID_IN_USE`	段标识符正在使用
`RSMERR_RESERVED_SEGID`	段标识符已保留

RSMERR_NOT_CREATOR	不是段的创建者
RSMERR_BAD_ADDR	地址错误
RSMERR_INSUFFICIENT_MEM	内存不足
RSMERR_INSUFFICIENT_RESOURCES	资源不足

授权的段 ID 范围：

#define RSM_DRIVER_PRIVATE_ID_BASE	0
#define RSM_DRIVER_PRIVATE_ID_END	0x0FFFFF
#define RSM_CLUSTER_TRANSPORT_ID_BASE	0x100000
#define RSM_CLUSTER_TRANSPORT_ID_END	0x1FFFFF
#define RSM_RSMLIB_ID_BASE	0x200000
#define RSM_RSMLIB_ID_END	0x2FFFFF
#define RSM_DLPI_ID_BASE	0x300000
#define RSM_DLPI_ID_END	0x3FFFFF
#define RSM_HPC_ID_BASE	0x400000
#define RSM_HPC_ID_END	0x4FFFFF

以下范围会保留，以便在发布值为零时由系统进行分配。

#define RSM_USER_APP_ID_BASE	0x80000000
#define RSM_USER_APP_ID_END	0xFFFFFFF

重新发布段

int **rsm_memseg_export_republish**(*rsm_memseg_export_handle_t memseg*, *rsmapi_access_entry_t access_list[]*, *uint_t access_list_length*);

此函数可用于建立新的节点访问列表和段访问模式。这些更改仅会影响将来的导入调用，并且不会撤消已准许的导入请求。

返回值：如果成功，则返回 0。否则，返回错误值。

RSMERR_BAD_SEG_HNDL	段句柄无效
RSMERR_SEG_NOT_PUBLISHED	段未发布
RSMERR_BAD_ACL	访问控制列表无效
RSMERR_NOT_CREATOR	不是段的创建者
RSMERR_INSUFFICIENT_MEMF	内存不足

RSMERR_INSUFFICIENT_RESOURCES	资源不足
RSMERR_INTERRUPTED	操作被信号中断

取消发布段

int **rsm_memseg_export_unpublish**(*rsm_memseg_export_handle_t memseg*);

返回值：如果成功，则返回 0。否则，返回错误值。

RSMERR_BAD_SEG_HNDL	段句柄无效
RSMERR_SEG_NOT_PUBLISHED	段未发布
RSMERR_NOT_CREATOR	不是段的创建者
RSMERR_INTERRUPTED	操作被信号中断

重新绑定内存段

重新绑定操作可释放导出段的当前后备存储。释放导出段的当前后备存储之后，重新绑定操作将分配新的后备存储。应用程序必须首先获取分配给段的新虚拟内存。此操作对于段的导入者是透明的。

注 – 应用程序负责防止在重新绑定操作完成之前对段数据进行访问。重新绑定过程中从段中检索数据不会导致系统故障，但执行此类操作的结果是不确定的。

重新绑定段

int **rsm_memseg_export_rebind**(*rsm_memseg_export_handle_t memseg*, *void *vaddr*, *offset_t off*, *size_t size*);

返回值：如果成功，则返回 0。否则，返回错误值。

RSMERR_BAD_SEG_HNDL	段句柄无效
RSMERR_BAD_LENGTH	长度无效
RSMERR_BAD_ADDR	地址无效
RSMERR_REBIND_NOT_ALLOWED	不允许重新绑定
RSMERR_NOT_CREATOR	不是段的创建者
RSMERR_PERM_DENIED	权限被拒绝
RSMERR_INSUFFICIENT_MEM	内存不足
RSMERR_INSUFFICIENT_RESOURCES	资源不足
RSMERR_INTERRUPTED	操作被信号中断

导入端内存段操作

以下列出了导入端操作：

- 连接和断开连接内存段
- 访问导入的段内存
- 屏障操作，用于强制设置数据访问操作顺序以及用于访问错误检测

连接操作用于创建 RSM 导入段并与导出的段形成逻辑连接。

对导入的段内存的访问由以下三个接口类别提供：

- 段访问。
- 数据传输。
- 段内存映射。

连接和断开连接内存段

连接到段

`int` **`rsm_memseg_import_connect`**`(`*rsmapi_controller_handle_t controller*`,` *rsm_node_id_t node_id*`,` *rsm_memseg_id_t segment_id*`,` *rsm_permission_t perm*`,` *rsm_memseg_import_handle_t *im_memseg*`);`

此函数可用于通过指定的权限 *perm* 连接到远程节点 *node_id* 上的段 *segment_id*。连接到段之后，此函数会返回一个段句柄。

参数 *perm* 用于指定导入者针对此连接请求的访问模式。要建立连接，可将导出者指定的访问权限与导入者使用的访问模式、用户 ID 和组 ID 进行比较。如果请求模式无效，则会拒绝连接请求。*perm* 参数限制为以下八进制值：

0400　读取模式

0200　写入模式

0600　读/写模式

指定的控制器必须与用于段导出的控制器具有物理连接。

返回值：如果成功，则返回 0。否则，返回错误值。

`RSMERR_BAD_CTLR_HNDL`	控制器句柄无效
`RSMERR_CTLR_NOT_PRESENT`	控制器不存在
`RSMERR_BAD_SEG_HNDL`	段句柄无效
`RSMERR_PERM_DENIED`	权限被拒绝
`RSMERR_SEG_NOT_PUBLISHED_TO_NODE`	段未发布至节点

RSMERR_SEG_NOT_PUBLISHED	未发布此类段
RSMERR_REMOTE_NODE_UNREACHABLE	无法访问远程节点
RSMERR_INTERRUPTED	连接已中断
RSMERR_INSUFFICIENT_MEM	内存不足
RSMERR_INSUFFICIENT_RESOURCES	资源不足
RSMERR_BAD_ADDR	地址错误

断开段连接

`int` **`rsm_memseg_import_disconnect`**`(`*rsm_memseg_import_handle_t im_memseg*`);`

此函数可用于断开段连接。断开段连接之后，此函数将释放段的资源。所有与断开连接的段的现有映射都将删除。句柄 `im_memseg` 将会释放。

返回值：如果成功，则返回 0。否则，返回错误值。

RSMERR_BAD_SEG_HNDL	段句柄无效
RSMERR_SEG_STILL_MAPPED	仍映射段
RSMERR_POLLFD_IN_USE	`pollfd` 正在使用

内存访问原语

以下接口提供了一种机制，用于在 8 位和 64 位数据之间进行传输。`get` 接口使用重复计数 (*rep_cnt*) 来表示进程将从连续位置读取的给定大小的数据项数。这些位置从导入的段中的字节偏移 *offset* 开始。数据写入从 *datap* 开始的连续位置。`put` 接口使用重复计数 (*rep_cnt*)。此计数表示进程将从连续位置读取的数据项数。这些位置从 *datap* 开始。然后，数据会写入已导入段中的连续位置。这些位置从 *offset* 参数所指定的字节偏移开始。

如果源与目标具有不兼容的字节存储顺序特征，则这些接口还可提供字节交换功能。

函数原型：

`int` **`rsm_memseg_import_get8`**`(`*rsm_memseg_import_handle_t im_memseg*`,` *off_t offset*`,` *uint8_t *datap*`,` *ulong_t rep_cnt*`);`

`int` **`rsm_memseg_import_get16`**`(`*rsm_memseg_import_handle_t im_memseg*`,` *off_t offset*`,` *uint16_t *datap*`,` *ulong_t rep_cnt*`);`

`int` **`rsm_memseg_import_get32`**`(`*rsm_memseg_import_handle_t im_memseg*`,` *off_t offset*`,` *uint32_t *datap*`,` *ulong_t rep_cnt*`);`

`int` **`rsm_memseg_import_get64`**`(`*rsm_memseg_import_handle_t im_memseg*`,` *off_t offset*`,` *uint64_t *datap*`,` *ulong_t rep_cnt*`);`

int **rsm_memseg_import_put8**(*rsm_memseg_import_handle_t im_memseg*, *off_t offset*, *uint8_t *datap*, *ulong_t rep_cnt*);

int **rsm_memseg_import_put16**(*rsm_memseg_import_handle_t im_memseg*, *off_t offset*, *uint16_t *datap*, *ulong_t rep_cnt*);

int **rsm_memseg_import_put32**(*rsm_memseg_import_handle_t im_memseg*, *off_t offset*, *uint32_t *datap*, *ulong_t rep_cnt*);

int **rsm_memseg_import_put64**(*rsm_memseg_import_handle_t im_memseg*, *off_t offset*, *uint64_t *datap*, *ulong_t rep_cnt*);

以下接口用于进行规模大于段访问操作所支持范围的数据传送。

放置段

int **rsm_memseg_import_put**(*rsm_memseg_import_handle_t im_memseg*, *off_t offset*, *void *src_addr*, *size_t length*);

此函数可用于将数据从 *src_addr* 和 *length* 所指定的本地内存复制到句柄和偏移所指定的对应导入的段所在位置。

获取段

int **rsm_memseg_import_get**(*rsm_memseg_import_handle_t im_memseg*, *off_t offset*, *void *dst_addr*, *size_t length*);

此函数类似于 `rsm_memseg_import_put()`，但是数据从导入的段流入 *dest_vec* 参数所定义的本地区域。

`put` 和 `get` 例程从参数 *offset* 所指定的字节偏移位置写入或读取指定的数据量。这些例程从段的基地址开始。偏移必须在相应的边界对齐。例如，`rsm_memseg_import_get64()` 要求 *offset* 和 *datap* 在双字界对齐，而 `rsm_memseg_import_put32()` 则要求偏移在单字边界对齐。

缺省情况下，段的屏障模式属性为 `implicit` 。隐式屏障模式表示调用者假设数据传输在从操作返回时已完成或失败。由于缺省屏障模式为隐式，因此应用程序必须初始化屏障。使用缺省模式时，应用程序会在调用 `put` 或 `get` 例程之前使用 `rsm_memseg_import_init_barrier ()` 函数初始化屏障。要使用显式操作模式，调用者必须使用屏障操作来强制完成传输。强制完成传输之后，调用者必须确定强制完成是否产生了任何错误。

注 – 通过在 `rsm_memseg_import_map()` 例程中传递偏移可以部分映射导入段。如果部分映射了导入段，则 `put` 或 `get` 例程中的 *offset* 参数是相对于段的基地址。用户必须确保将正确的字节偏移传递给 `put` 和 `get` 例程。

返回值：如果成功，则返回 0。否则，返回错误值。

RSMERR_BAD_SEG_HNDL	段句柄无效
RSMERR_BAD_ADDR	地址错误
RSMERR_BAD_MEM_ALIGNMENT	内存对齐无效
RSMERR_BAD_OFFSET	偏移无效
RSMERR_BAD_LENGTH	长度无效
RSMERR_PERM_DENIED	权限被拒绝
RSMERR_BARRIER_UNINITIALIZED	未初始化屏障
RSMERR_BARRIER_FAILURE	I/O 完成错误
RSMERR_CONN_ABORTED	连接已中止
RSMERR_INSUFFICIENT_RESOURCES	资源不足

分散/集中访问

rsm_memseg_import_putv() 和 rsm_memseg_import_getv () 函数允许使用 I/O 请求列表来替代单个源地址和单个目标地址。

函数原型：

int **rsm_memseg_import_putv**(*rsm_scat_gath_t *sg_io*);

int **rsm_memseg_import_getv**(*rsm_scat_gath_t *sg_io*);

使用分散/集中列表的 I/O 向量部分 (sg_io) 可以指定本地虚拟地址或 local_memory_handles 。句柄是一种重复使用本地地址范围的有效方法。在释放句柄之前，已分配的系统资源（如已锁定的本地内存）会一直保留。句柄的支持函数包括 rsm_create_localmemory_handle () 和 rsm_free_localmemory_handle()。

可以将虚拟地址或句柄收集到向量中，以便写入单个远程段。另外，还可以将从单个远程段读取的结果分散到虚拟地址或句柄的向量中。

整个向量的 I/O 会在返回之前启动。导入段的屏障模式属性可确定 I/O 是否在函数返回之前已完成。将屏障模式属性设置为 implicit 可保证数据传输按照在向量中的输入顺序完成。在每个列表项开始时会执行隐式屏障打开，在每个列表项结束时会执行隐式屏障关闭。如果检测到错误，向量的 I/O 会终止并且函数会立即返回。剩余计数表示其 I/O 尚未完成或尚未启动的项数。

可以指定在 putv 或 getv 操作成功时，向目标段发送通知事件。要指定传送通知事件，请在 rsm_scat_gath_t 结构的 flags 项中指定 RSM_IMPLICIT_SIGPOST 值。flags 项还可以包含值 RSM_SIGPOST_NO_ACCUMULATE，该值在设置了 RSM_IMPLICIT_SIGPOST 的情况下会传递给信号传递操作。

返回值：如果成功，则返回 0。否则，返回错误值。

RSMERR_BAD_SGIO	分散/集中结构指针无效
RSMERR_BAD_SEG_HNDL	段句柄无效
RSMERR_BAD_CTLR_HNDL	控制器句柄无效
RSMERR_BAD_ADDR	地址错误
RSMERR_BAD_OFFSET	偏移无效
RSMERR_BAD_LENGTH	长度无效
RSMERR_PERM_DENIED	权限被拒绝
RSMERR_BARRIER_FAILURE	I/O 完成错误
RSMERR_CONN_ABORTED	连接已中止
RSMERR_INSUFFICIENT_RESOURCES	资源不足
RSMERR_INTERRUPTED	操作被信号中断

获取本地句柄

int **rsm_create_localmemory_handle** (*rsmapi_controller_handle_t cntrl_handle* , *rsm_localmemory_handle_t *local_handle*, *caddr_t local_vaddr*, *size_t length*);

此函数可用于获取本地句柄，以便在后续调用 putv 或 getv 时用于 I/O 向量。尽快释放句柄可节省系统资源（特别是本地句柄占用的内存），这些资源可能会锁定。

返回值：如果成功，则返回 0。否则，返回错误值。

RSMERR_BAD_CTLR_HNDL	控制器句柄无效
RSMERR_BAD_LOCALMEM_HNDL	本地内存句柄无效
RSMERR_BAD_LENGTH	长度无效
RSMERR_BAD_ADDR	地址无效
RSMERR_INSUFFICIENT_MEM	内存不足

释放本地句柄

rsm_free_localmemory_handle(*rsmapi_controller_handle_t cntrl_handle*, *rsm_localmemory_handle_t handle*);

此函数可用于释放与本地句柄关联的系统资源。由于进程退出时会释放属于该进程的所有句柄，因此调用此函数可节省系统资源。

返回值：如果成功，则返回 0。否则，返回错误值。

RSMERR_BAD_CTLR_HNDL	控制器句柄无效
RSMERR_BAD_LOCALMEM_HNDL	本地内存句柄无效

以下示例说明了主数据结构的定义。

示例 2–1 主数据结构

```
typedef void *rsm_localmemory_handle_t
typedef struct {
    ulong_t    io_request_count;     /* number of rsm_iovec_t entries */
    ulong_t    io_residual_count;    /* rsm_iovec_t entries not completed */

    int        flags;
    rsm_memseg_import_handle_t    remote_handle;    /* opaque handle for import segment */
    rsm_iovec_t    *iovec;           /* pointer to array of io_vec_t */
} rsm_scat_gath_t;

typedef struct {
    int    io_type;                  /* HANDLE or VA_IMMEDIATE */
    union {
        rsm_localmemory_handle_t    handle;          /* used with HANDLE */
        caddr_t                     virtual_addr;    /* used with VA_IMMEDIATE */
    } local;
    size_t     local_offset;             /* offset from handle base vaddr */
    size_t     import_segment_offset;    /* offset from segment base vaddr */
    size_t     transfer_length;
} rsm_iovec_t;
```

段映射

映射操作只能用于本机体系结构互连，如 Dolphin-SCI 或 NewLink。映射段可授予 CPU 内存操作访问该段的权限，从而节省了调用内存访问原语的开销。

导入段映射

`int` **`rsm_memseg_import_map`**`(`*rsm_memseg_import_handle_t im_memseg*`,` *void **address*`,` *rsm_attribute_t attr*`,` *rsm_permission_t perm*`,` *off_t offset*`,` *size_t length*`);`

此函数可用于将导入的段映射成调用者地址空间。如果指定了属性 `RSM_MAP_FIXED`，则此函数会在 ***address* 中指定的值所在位置映射段。

```
typedef enum {
    RSM_MAP_NONE  = 0x0,    /* system will choose available virtual address */
    RSM_MAP_FIXED = 0x1,    /* map segment at specified virtual address */
} rsm_map_attr_t;
```

返回值：如果成功，则返回 0。否则，返回错误值。

RSMERR_BAD_SEG_HNDL	段句柄无效
RSMERR_BAD_ADDR	地址无效

RSMERR_BAD_LENGTH	长度无效
RSMERR_BAD_OFFSET	偏移无效
RSMERR_BAD_PERMS	权限无效
RSMERR_SEG_ALREADY_MAPPED	已映射段
RSMERR_SEG_NOT_CONNECTED	未连接段
RSMERR_CONN_ABORTED	连接已中止
RSMERR_MAP_FAILED	映射时出现错误
RSMERR_BAD_MEM_ALIGNMENT	地址未在页边界上对齐

取消映射段

int **rsm_memseg_import_unmap**(*rsm_memseg_import_handle_t im_memseg*);

此函数可用于从用户虚拟地址空间中取消映射导入的段。

返回值：如果成功，则返回 0。否则，返回错误值。

RSMERR_BAD_SEG_HNDL	段句柄无效

屏障操作

使用屏障操作可以解决写入访问顺序内存模型问题。屏障操作还可提供远程内存访问错误检测功能。

屏障机制由以下操作组成：

- 初始化
- 打开
- 关闭
- 排序

打开和关闭操作定义了错误检测和排序的时间间隔。通过初始化操作，可以为每个导入的段创建屏障并指定屏障类型。当前支持的唯一屏障类型针对每个段具有一个时间间隔范围。请使用类型参数值 RSM_BAR_DEFAULT。

成功执行关闭操作可保证成功完成所涉及的访问操作，这些操作在屏障打开操作和屏障关闭操作之间进行。在屏障打开操作之后直到屏障关闭操作之前，不会报告单个数据访问操作（读取和写入）故障。

要在屏障范围内强制设置特定的写入完成顺序，请使用显式屏障排序操作。在屏障排序操作之前发出的写入操作会先于在屏障排序操作之后发出的操作完成。给定屏障范围内的写入操作会根据其他屏障范围进行排序。

初始化屏障

int **rsm_memseg_import_init_barrier** (*rsm_memseg_import_handle_t im_memseg*, *rsm_barrier_type_t type*, *rsmapi_barrier_t *barrier*);

注 – 目前，RSM_BAR_DEFAULT 是唯一支持的类型。

返回值：如果成功，则返回 0。否则，返回错误值。

RSMERR_BAD_SEG_HNDL	段句柄无效
RSMERR_BAD_BARRIER_PTR	屏障指针无效
RSMERR_INSUFFICIENT_MEM	内存不足

打开屏障

int **rsm_memseg_import_open_barrier** (*rsmapi_barrier_t *barrier*);

返回值：如果成功，则返回 0。否则，返回错误值。

RSMERR_BAD_SEG_HNDL	段句柄无效
RSMERR_BAD_BARRIER_PTR	屏障指针无效

关闭屏障

int **rsm_memseg_import_close_barrier** (*rsmapi_barrier_t *barrier*);

此函数可用于关闭屏障并刷新所有存储缓冲区。此调用假设如果调用 rsm_memseg_import_close_barrier() 失败，则调用进程将重试自上次 rsm_memseg_import_open_barrier 调用以来的所有远程内存操作。

返回值：如果成功，则返回 0。否则，返回错误值。

RSMERR_BAD_SEG_HNDL	段句柄无效
RSMERR_BAD_BARRIER_PTR	屏障指针无效
RSMERR_BARRIER_UNINITIALIZED	未初始化屏障
RSMERR_BARRIER_NOT_OPENED	未打开屏障
RSMERR_BARRIER_FAILURE	内存访问错误
RSMERR_CONN_ABORTED	连接已中止

排序屏障

int **rsm_memseg_import_order_barrier** (*rsmapi_barrier_t *barrier*);

此函数可用于刷新所有存储缓冲区。

返回值：如果成功，则返回 0。否则，返回错误值。

RSMERR_BAD_SEG_HNDL	段句柄无效
RSMERR_BAD_BARRIER_PTR	屏障指针无效
RSMERR_BARRIER_UNINITIALIZED	未初始化屏障
RSMERR_BARRIER_NOT_OPENED	未打开屏障
RSMERR_BARRIER_FAILURE	内存访问错误
RSMERR_CONN_ABORTED	连接已中止

销毁屏障

```
int rsm_memseg_import_destroy_barrier (rsmapi_barrier_t *barrier);
```

此函数可用于取消分配所有屏障资源。

返回值：如果成功，则返回 0。否则，返回错误值。

RSMERR_BAD_SEG_HNDL	段句柄无效
RSMERR_BAD_BARRIER_PTR	屏障指针无效

设置模式

```
int rsm_memseg_import_set_mode(rsm_memseg_import_handle_t im_memseg,
rsm_barrier_mode_t mode);
```

此函数支持可用于 put 例程的可选显式屏障范围。两种有效的屏障模式为 RSM_BARRIER_MODE_EXPLICIT 和 RSM_BARRIER_MODE_IMPLICIT 。屏障模式的缺省值为 RSM_BARRIER_MODE_IMPLICIT。在隐式模式下，隐式屏障打开和屏障关闭会应用于每个 put 操作。将屏障模式值设置为 RSM_BARRIER_MODE_EXPLICIT 之前，请使用 rsm_memseg_import_init_barrier 例程针对导入的段 im_memseg 初始化屏障。

返回值：如果成功，则返回 0。否则，返回错误值。

RSMERR_BAD_SEG_HNDL	段句柄无效

获取模式

```
int rsm_memseg_import_get_mode(rsm_memseg_import_handle_t im_memseg,
rsm_barrier_mode_t *mode);
```

此函数可用于获取 put 例程中屏障范围设置的当前模式值。

返回值：如果成功，则返回 0。否则，返回错误值。

RSMERR_BAD_SEG_HNDL　段句柄无效。

事件操作

通过事件操作，可以针对内存访问事件实现进程同步。如果进程无法使用 rsm_intr_signal_wait() 函数，则可以多路复用事件等待，方法是通过 rsm_memseg_get_pollfd () 获取轮询描述符并使用 poll 系统调用。

注 – 使用 rsm_intr_signal_post() 和 rsm_intr_signal_wait () 操作时需要处理对内核的 ioctl 调用。

传递信号

int **rsm_intr_signal_post**(*void *memseg, uint_t flags*);

void 指针 *memseg 可以将类型转换为导入段句柄或导出段句柄。如果 *memseg 引用导入句柄，则此函数会向导出进程发送信号。如果 *memseg 引用导出句柄，则此函数会向该段的所有导入者发送信号。如果已针对目标段暂挂事件，则将 flags 参数设置为 RSM_SIGPOST_NO_ACCUMULATE 可废弃此事件。

返回值：如果成功，则返回 0。否则，返回错误值。

RSMERR_BAD_SEG_HNDL　段句柄无效

RSMERR_REMOTE_NODE_UNREACHABLE　无法访问远程节点

等待信号

int **rsm_intr_signal_wait**(*void * memseg, int timeout*);

void 指针 *memseg 可以将类型转换为导入段句柄或导出段句柄。进程的阻塞时间最多可达到 *timeout* 毫秒，或在事件发生之前一直阻塞。如果值为 -1，则进程在事件发生之前或中断之前会一直阻塞。

返回值：如果成功，则返回 0。否则，返回错误值。

RSMERR_BAD_SEG_HNDL　段句柄无效

RSMERR_TIMEOUT　计时器已到期

RSMERR_INTERRUPTED　等待已中断

获取 pollfd

int **rsm_memseg_get_pollfd**(*void *memseg, struct pollfd *pollfd*);

此函数可用于通过指定段的描述符以及 rsm_intr_signal_post() 所生成的单个固定事件初始化指定的 pollfd 结构。将 pollfd 结构用于 poll 系统调用可等待

rsm_intr_signal_post 所通知的事件。如果当前未发布内存段，则 poll 系统调用无法返回有效的 pollfd。每次成功调用都会递增指定段的 pollfd 引用计数。

返回值：如果成功，则返回 0。否则，返回错误值。

RSMERR_BAD_SEG_HNDL　　段句柄无效

释放 pollfd

int **rsm_memseg_release_pollfd**(*oid *memseg*);

此调用可递减指定段的 pollfd 引用计数。如果引用计数为非零值，则取消发布、销毁或取消映射段的操作会失败。

返回值：如果成功，则返回 0。否则，返回错误值。

RSMERR_BAD_SEG_HNDL　　段句柄无效

RSMAPI 常规用法说明

这些用法说明介绍了共享内存操作的导出端和导入端的常规注意事项。另外，这些用法说明还包含有关段、文件描述符和 RSM 可配置参数的常规信息。

段分配和文件描述符用法

系统会为每个导出操作或导入操作分配一个文件描述符，导入或导出内存的应用程序无法访问此描述符。每个进程的文件描述符分配的缺省限制为 256。导入或导出应用程序必须相应调整此分配限制。如果应用程序将文件描述符限制增加到超过 256，则为导出段和导入段分配的文件描述符值从 256 开始。选择这些文件描述符值是为了避免干扰应用程序的正常文件描述符分配。此行为允许在仅使用小于 256 的文件描述符值的 32 位应用程序中使用特定的 libc 函数。

导出端注意事项

应用程序必须防止在重新绑定操作完成之前访问段。在重新绑定过程中，访问段数据不会导致系统故障，但是数据内容结果是不确定的。当前，虚拟地址空间必须已映射并有效。

导入端注意事项

为段导入指定的控制器必须与用于段导出的控制器具有物理连接。

RSM 可配置参数

SUNWrsm 软件包包括 rsm.conf 文件。此文件位于 /usr/kernel/drv 中。此文件是 RSM 的配置文件。rsm.conf 文件可用于指定特定可配置 RSM 属性的值。当前在 rsm.conf 中定义的可配置属性包括 max-exported-memory 和 enable-dynamic-reconfiguration。

max-exported-memory	此属性用于指定可导出内存量的上限。此上限以可用内存总量的百分比表示。如果指定此属性值为零，则表示可导出内存量没有限制。
enable-dynamic-reconfiguration	此属性的值表示是否启用了动态重新配置。值为零表示禁用动态重新配置。值为 1 将启用动态重新配置支持。此属性的缺省值为 1。

◆ ◆ ◆ 第 3 章

3 会话描述协议 API

会话描述协议 (Session Description Protocol, SDP) 用于描述多媒体会话。本章中讨论的 SDP API 包含可用于将 SDP 功能添加到应用程序的函数调用。

会话描述 API 概述

组成 SDP API 的函数调用由共享对象 `libcommputil.so.1` 提供。此共享对象中的函数可解析 SDP 描述以及检查该描述的语法。

`sdp.h` 头文件定义 `sdp_session_t` 结构，该结构包含以下成员：

```
typedef    struct sdp_session {
   int             sdp_session_version; /* SDP session verstion */
   int             s_version;           /* SDP version field */
   sdp_origin_t    *s_origin;           /* SDP origin field */
   char            *s_name;             /* SDP name field */
   char            *s_info;             /* SDP info field */
   char            *s_uri;              /* SDP uri field */
   sdp_list_t      *s_email;            /* SDP email field */
   sdp_list_t      *s_phone;            /* SDP phone field */
   sdp_conn_t      *s_conn;             /* SDP connection field */
   sdp_bandwidth_t *s_bw;               /* SDP bandwidth field */
   sdp_time_t      *s_time;             /* SDP time field */
   sdp_zone_t      *s_zone;             /* SDP zone field */
   sdp_key_t       *s_key;              /* SDP key field */
   sdp_attr_t      *s_attr;             /* SDP attribute field */
   sdp_media_t     *s_media;            /* SDP media field */
} sdp_session_t;
```

`sdp_session_version` 成员可跟踪该结构的版本。`sdp_session_version` 成员的初始值为 `SDP_SESSION_VERSION_1`。

`sdp_origin_t` 结构包含以下成员：

```
typedef struct sdp_origin {
   char        *o_username;  /* username of the originating host */
   uint64_t    o_id;         /* session id */
```

```
   uint64_t    o_version;    /* version number of this session */
                             /* description */
   char        *o_nettype;   /* type of network */
   char        *o_addrtype;  /* type of the address */
   char        *o_address;   /* address of the machine from which */
                             /* session was created */
} sdp_origin_t;
```

sdp_conn_t 结构包含以下成员：

```
typedef struct sdp_conn {
   char    *c_nettype;         /* type of network */
   char    *c_addrtype;        /* type of the address */
   char    *c_address;         /* unicast-address or multicast */
                               /* address */
   int     c_addrcount;        /* number of addresses (case of */
                               /* multicast address with layered */
                               /* encodings */
   struct  sdp_conn *c_next;   /* pointer to next connection */
                               /* structure; there could be several */
                               /* connection fields in SDP description */
   uint8_t c_ttl;              /* TTL value for IPV4 multicast address */
} sdp_conn_t;
```

sdp_bandwidth_t 结构包含以下成员：

```
typedef struct sdp_bandwidth {
   char                  *b_type; /* info needed to interpret b_value */
   uint64_t              b_value; /* bandwidth value */
   struct sdp_bandwidth  *b_next; /* pointer to next bandwidth structure*/
                                  /* (there could be several bandwidth */
                                  /* fields in SDP description */
} sdp_bandwidth_t;
```

sdp_list_t 结构是 void 指针的链接列表。该结构包含 SDP 字段。对于 email 和 phone 等 SDP 结构字段，void 指针指向字符缓冲区。如果没有预定义元素数目，则可使用该结构来保留信息，就像在重复的 offset 字段中 void 指针保留整数值的情况一样。

sdp_list_t 结构包含以下成员：

```
typedef struct sdp_list {
   void            *value; /* string values in case of email, phone and */
                           /* format (in media field) or integer values */
                           /* in case of offset (in repeat field) */
   struct sdp_list *next;  /* pointer to the next node in the list */
} sdp_list_t;
```

sdp_repeat_t 结构将始终是时间结构 sdp_time_t 的一部分。repeat 字段不会单独出现在 SDP 描述中，该字段始终与 time 字段关联。

sdp_repeat_t 结构包含以下成员：

```
typedef struct sdp_repeat {
   uint64_t          r_interval; /* repeat interval, e.g. 86400 seconds */
                                 /* (1 day) */
```

```
   uint64_t          r_duration; /* duration of session, e.g. 3600 */
                                 /* seconds (1 hour) */
   sdp_list_t        *r_offset;  /* linked list of offset values; each */
                                 /* represents offset from start-time */
                                 /* in the SDP time field */
   struct sdp_repeat *r_next;    /* pointer to next repeat structure; */
                                 /* there could be several repeat */
                                 /* fields in the SDP description */
```

sdp_time_t 结构包含以下成员：

```
typedef struct sdp_time {
   uint64_t         t_start;   /* start-time for a session */
   uint64_t         t_stop;    /* end-time for a session */
   sdp_repeat_t     *t_repeat; /* points to the SDP repeat field */
   struct sdp_time  *t_next;   /* pointer to next time field; there */
                               /* could there could be several time */
                               /* fields in SDP description */
} sdp_time_t;
```

sdp_zone_t 结构包含以下成员：

```
typedef struct sdp_zone {
   uint64_t        z_time;    /* base time */
   char            *z_offset; /* offset added to z_time to determine */
                              /* session time; mainly used for daylight */
                              /* saving time conversions */
   struct sdp_zone *z_next;   /* pointer to next zone field; there */
                              /* could be several <adjustment-time> */
                              /* <offset> pairs within a zone field */
} sdp_zone_t;
```

sdp_key_t 结构包含以下成员：

```
typedef struct sdp_key {
   char   *k_method;   /* key type */
   char   *k_enckey;   /* encryption key */
 } sdp_key_t;
```

sdp_attr_t 结构包含以下成员：

```
typedef struct sdp_attr {
   char            *a_name;  /* name of the attribute */
   char            *a_value; /* value of the attribute */
   struct sdp_attr *a_next;  /* pointer to the next attribute */
                             /* structure; there could be several */
                             /* attribute fields within SDP description */
} sdp_attr_t;
```

sdp_media_t 结构包含以下成员：

```
typedef struct sdp_media {
   char              *m_name;     /* name of the media such as "audio", */
                                  /* "video", "message" */
   uint_t            m_port;      /* transport layer port information */
```

```
    int               m_portcount; /* number of ports in case of */
                                   /* hierarchically encoded streams */
    char              *m_proto;    /* transport protocol */
    sdp_list_t        *m_format;   /* media format description */
    char              *m_info;     /* media info field */
    sdp_conn_t        *m_conn;     /* media connection field */
    sdp_bandwidth_t   *m_bw;       /* media bandwidth field */
    sdp_key_t         *m_key;      /* media key field */
    sdp_attr_t        *m_attr;     /* media attribute field */
    struct sdp_media  *m_next;     /* pointer to next media structure; */
                                   /* there could be several media */
                                   /* sections in SDP description */
    sdp_session_t     *m_session;  /* pointer to the session structure */
} sdp_media_t;
```

SDP 库函数

API 库函数支持以下操作：

- 创建 SDP 会话结构
- 在 SDP 会话结构中搜索
- 关闭 SDP 会话结构
- 实用程序函数

创建 SDP 会话结构

创建新的 SDP 会话结构的第一步是通过调用 `sdp_new_session()` 函数来为新结构分配内存。此函数可返回指向新会话结构的指针。本节中的其他函数使用该指针来构建新会话结构。完成新会话结构的构建之后，请使用 `sdp_session_to_str()` 函数将其转换为字符串表示形式。

创建新的 SDP 会话结构

`sdp_session_t` ***sdp_new_session***`();`

`sdp_new_session()` 函数可为 `session` 参数指定的新 SDP 会话结构分配内存，并向该新结构分配版本号。可以通过调用 `sdp_free_session()` 函数来释放分配给此会话结构的内存。

返回值：如果 `sdp_new_session()` 函数成功完成，该函数将返回新分配的 SDP 会话结构。如果失败，该函数将返回 NULL。

向 SDP 会话结构添加源字段

`int` ***sdp_add_origin***`(sdp_session_t` **session*`, const char` **name*`, uint64_t` *id*`, uint64_t` *ver*`, const char` **nettype*`, const char` **addrtype*`, const char` **address*`);`

`sdp_add_origin()` 函数使用 *name*、*id*、*ver*、*nettype*、*addrtype* 和 *address* 参数将 `ORIGIN (o=)` SDP 字段添加到由 `session` 参数 (`sdp_session_t`) 的值指定的会话结构。

返回值：如果 sdp_add_origin() 函数成功完成，该函数将返回 0。如果强制参数不存在，该函数将返回 EINVAL。如果内存分配失败，该函数将返回 ENOMEM。在出现错误时，errno 的值不会发生更改。

向 SDP 会话结构添加名称字段

```
int sdp_add_name(sdp_session_t *session, const char *name);
```

sdp_add_name() 函数使用 *name* 参数将 NAME (s=) SDP 字段添加到由 session 参数 (sdp_session_t) 的值指定的会话结构。

返回值：如果 sdp_add_name() 函数成功完成，该函数将返回 0。如果强制参数不存在，该函数将返回 EINVAL。如果内存分配失败，该函数将返回 ENOMEM。在出现错误时，errno 的值不会发生更改。

向 SDP 会话结构添加信息字段

```
int sdp_add_information(char **information, const char *value);
```

sdp_add_information() 函数使用 *value* 参数将 INFO (i=) SDP 字段添加到会话结构 (sdp_session_t) 或介质结构 (sdp_media_t)。此字段可以进入 SDP 描述的介质部分或会话部分。您必须将 &session->s_info 或 &media->m_info 作为第一个参数进行传递，以指定该部分。

返回值：如果 sdp_add_information() 函数成功完成，该函数将返回 0。如果强制参数不存在，该函数将返回 EINVAL。如果内存分配失败，该函数将返回 ENOMEM。在出现错误时，errno 的值不会发生更改。

向 SDP 会话结构添加 URI 字段

```
int sdp_add_uri(sdp_session_t *session, const char *uri);
```

sdp_add_uri() 函数使用 *uri* 参数将 URI (u=) SDP 字段添加到由 session 参数 (sdp_session_t) 的值指定的会话结构。

返回值：如果 sdp_add_uri() 函数成功完成，该函数将返回 0。如果强制参数不存在，该函数将返回 EINVAL。如果内存分配失败，该函数将返回 ENOMEM。在出现错误时，errno 的值不会发生更改。

向 SDP 会话结构添加电子邮件字段

```
int sdp_add_email(sdp_session_t *session, const char *email);
```

sdp_add_email() 函数使用 *email* 参数将 EMAIL (e=) SDP 字段添加到由 session 参数 (sdp_session_t) 的值指定的会话结构。

返回值：如果 sdp_add_email() 函数成功完成，该函数将返回 0。如果强制参数不存在，该函数将返回 EINVAL。如果内存分配失败，该函数将返回 ENOMEM。在出现错误时，errno 的值不会发生更改。

向 SDP 会话结构添加电话字段

```
int sdp_add_phone(sdp_session_t *session, const char *email);
```

sdp_add_phone() 函数使用 *phone* 参数将 PHONE (p=) SDP 字段添加到由 session 参数 (sdp_session_t) 的值指定的会话结构。

返回值：如果 sdp_add_phone() 函数成功完成，该函数将返回 0。如果强制参数不存在，该函数将返回 EINVAL。如果内存分配失败，该函数将返回 ENOMEM。在出现错误时，errno 的值不会发生更改。

向 SDP 会话结构添加连接字段

```
int sdp_add_connection(sdp_conn_t **conn, const char *nettype, const char
*addrtype, const char *address, uint8_t ttl, int addrcount);
```

sdp_add_connection() 函数使用 *nettype*、*addrtype*、*address*、*ttl* 和 *addrcount* 参数将 CONNECTION (c=) SDP 字段添加到会话结构 (sdp_session_t) 或介质结构 (sdp_media_t)。对于 IPv4 或 IPv6 单播地址，将 *ttl* 和 *addrcount* 参数的值设置为零。对于多播地址，将 *ttl* 参数的值设置为零到 255 之间的某个值。多播地址不能包含值为零的 addrcount 参数。

此字段可以进入 SDP 描述的介质部分或会话部分。您必须将 &session->s_info 或 &media->m_info 作为第一个参数进行传递，以指定该部分。

返回值：如果 sdp_add_connection() 函数成功完成，该函数将返回 0。如果强制参数不存在，该函数将返回 EINVAL。如果内存分配失败，该函数将返回 ENOMEM。在出现错误时，errno 的值不会发生更改。

向 SDP 会话结构添加带宽字段

```
int sdp_add_bandwidth(sdp_bandwidth_t **bw, const char *type, uint64_t value);
```

sdp_add_bandwidth() 函数使用 *type* 参数和 *value* 参数将 BANDWIDTH (b=) SDP 字段添加到会话结构 (sdp_session_t) 或介质结构 (sdp_media_t)。

此字段可以进入 SDP 描述的介质部分或会话部分。您必须将 &session->s_info 或 &media->m_info 作为第一个参数进行传递，以指定该部分。

返回值：如果 sdp_add_bandwidth() 函数成功完成，该函数将返回 0。如果强制参数不存在，该函数将返回 EINVAL。如果内存分配失败，该函数将返回 ENOMEM。在出现错误时，errno 的值不会发生更改。

向 SDP 会话结构添加时间字段

```
int sdp_add_time(sdp_session_t *session, uint64_t starttime, uint64_t stoptime,
sdp_time_t **time);
```

sdp_add_time() 函数使用 *starttime* 和 *stoptime* 参数的值将 TIME (t=) SDP 字段添加到会话结构。此函数使用 *time* 参数创建新的时间结构并返回指向该结构的指针。

返回值：如果 sdp_add_time() 函数成功完成，该函数将返回 0。如果强制参数不存在，该函数将返回 EINVAL。如果内存分配失败，该函数将返回 ENOMEM。在出现错误时，errno 的值不会发生更改。

向 SDP 会话结构添加重复字段

int **sdp_add_repeat**(sdp_time_t **time*, uint64_t *interval*, uint64_t *duration*, const char **offset*);

sdp_add_repeat() 函数使用 *interval*、*duration* 和 *offset* 参数的值将 REPEAT (r=) SDP 字段添加到会话结构。*offset* 参数的值是包含一个或多个偏移值的字符串，如 60 或 60 1d 3h。*time* 参数的值是指向 sdp_add_time() 函数创建的时间结构的指针。

返回值：如果 sdp_add_repeat() 函数成功完成，该函数将返回 0。如果强制参数不存在，该函数将返回 EINVAL。如果内存分配失败，该函数将返回 ENOMEM。在出现错误时，errno 的值不会发生更改。

向 SDP 会话结构添加区域字段

int **sdp_add_zone**(sdp_session_t **session*, uint64_t *time*, const char **offset*);

sdp_add_zone() 函数使用 *time* 参数和 *offset* 参数将 ZONE (z=) SDP 字段添加到由 session 参数 (sdp_session_t) 的值指定的会话结构。通过针对每个时间/偏移值对调用此函数，可以为单个区域字段添加多个时间和偏移值。

返回值：如果 sdp_add_zone() 函数成功完成，该函数将返回 0。如果强制参数不存在，该函数将返回 EINVAL。如果内存分配失败，该函数将返回 ENOMEM。在出现错误时，errno 的值不会发生更改。

向 SDP 会话结构添加密钥字段

int **sdp_add_key**(sdp_key_t ***key*, const char **method*, const char **enckey*);

sdp_add_key() 函数使用 *method* 参数和 *enckey* 参数将 KEY (k=) SDP 字段添加到会话结构 (sdp_session_t) 或介质结构 (sdp_media_t)。此字段可以进入 SDP 描述的介质部分或会话部分。您必须将 &session->s_info 或 &media->m_info 作为第一个参数进行传递，以指定该部分。

返回值：如果 sdp_add_key() 函数成功完成，该函数将返回 0。如果强制参数不存在，该函数将返回 EINVAL。如果内存分配失败，该函数将返回 ENOMEM。在出现错误时，errno 的值不会发生更改。

向 SDP 会话结构添加属性字段

int **sdp_add_attribute**(sdp_attr_t ***attr*, const char **name*, const char **value*);

sdp_add_attribute() 函数使用 *name* 参数和 *value* 参数将 ATTRIBUTE (a=) SDP 字段添加到会话结构 (sdp_session_t) 或介质结构 (sdp_media_t)。此字段可以进入 SDP 描述的介质部分或会话部分。您必须将 &session->s_info 或 &media->m_info 作为第一个参数进行传递，以指定该部分。

返回值：如果 sdp_add_attribute() 函数成功完成，该函数将返回 0。如果强制参数不存在，该函数将返回 EINVAL。如果内存分配失败，该函数将返回 ENOMEM。在出现错误时，errno 的值不会发生更改。

向 SDP 会话结构添加介质字段

int **sdp_add_media**(sdp_session_t **session*, const char **name*, uint_t *port*, int *portcount*, const char **protocol*, const char **format*, sdp_media_t ***media*);

sdp_add_media() 函数使用 *name*、*port*、*portcount*、*protocol* 和 *format* 参数的值将 MEDIA (m=) SDP 字段添加到由 session 参数 (sdp_session_t) 的值指定的会话结构。*format* 参数是包含一个或多个值的字符串，如字符串 0 32 97。

此函数使用 *media* 参数创建新的介质结构并返回指向该结构的指针。将 SDP 字段添加到介质结构的函数会使用该指针。

返回值：如果 sdp_add_media() 函数成功完成，该函数将返回 0。如果强制参数不存在，该函数将返回 EINVAL。如果内存分配失败，该函数将返回 ENOMEM。在出现错误时，errno 的值不会发生更改。

代码样例：构建 SDP 会话结构

本示例使用本节中述及的函数来创建新的 SDP 会话结构、将字段添加到该结构，并将已完成的结构转换为该结构的字符串表示形式。在示例的结尾，程序将调用 sdp_free_session() 函数以释放会话。

示例 3–1 构建 SDP 会话结构

```
/* SDP Message we will be building
"v=0\r\n\
o=Alice 2890844526 2890842807 IN IP4 10.47.16.5\r\n\
s=-\r\n\
i=A Seminar on the session description protocol\r\n\
u=http://www.example.com/seminars/sdp.pdf\r\n\
e=alice@example.com (Alice Smith)\r\n\
p=+1 911-345-1160\r\n\
c=IN IP4 10.47.16.5\r\n\
b=CT:1024\r\n\
t=2854678930 2854679000\r\n\
r=604800 3600 0 90000\r\n\
z=2882844526 -1h 2898848070 0h\r\n\
a=recvonly\r\n\
m=audio 49170 RTP/AVP 0\r\n\
i=audio media\r\n\
b=CT:1000\r\n\
k=prompt\r\n\
```

示例 3–1 构建 SDP 会话结构 （续）

```
m=video 51372 RTP/AVP 99 90\r\n\
i=video media\r\n\
a=rtpmap:99 h232-199/90000\r\n\
a=rtpmap:90 h263-1998/90000\r\n"
*/

#include stdio.h>
#include string.h>
#include errno.h>
#include sdp.h>

int main ()
{
   sdp_session_t  *my_sess;
   sdp_media_t    *my_media;
   sdp_time_t     *my_time;
   char *b_sdp;

   my_sess = sdp_new_session();
   if (my_sess == NULL) {
return (ENOMEM);
   }
   my_sess->version = 0;
   if (sdp_add_name(my_sess, "-") != 0)
goto err_ret;
   if (sdp_add_origin(my_sess, "Alice", 2890844526ULL, 2890842807ULL,
 "IN", "IP4", "10.47.16.5") != 0)
goto err_ret;
   if (sdp_add_information(&my_sess->s_info, "A Seminar on the session"
 "description protocol") != 0)
goto err_ret;
   if (sdp_add_uri (my_sess, "http://www.example.com/seminars/sdp.pdf")
 != 0)
goto err_ret;
   if (sdp_add_email(my_sess, "alice@example.com (Alice smith)") != 0)
goto err_ret;
   if (sdp_add_phone(my_sess, "+1 911-345-1160") != 0)
goto err_ret;
   if (sdp_add_connection(&my_sess->s_conn, "IN", "IP4", "10.47.16.5",
0, 0) != 0)
goto err_ret;
   if (sdp_add_bandwidth(&my_sess->s_bw, "CT", 1024) != 0)
goto err_ret;
   if (sdp_add_time(my_sess, 2854678930ULL, 2854679000ULL, &my_time)
!= 0)
goto err_ret;
   if (sdp_add_repeat(my_time, 604800ULL, 3600ULL, "0 90000") != 0)
goto err_ret;
   if (sdp_add_zone(my_sess, 2882844526ULL, "-1h") != 0)
goto err_ret;
   if (sdp_add_zone(my_sess, 2898848070ULL, "0h") != 0)
goto err_ret;
   if (sdp_add_attribute(&my_sess->s_attr, "sendrecv", NULL) != 0)
goto err_ret;
   if (sdp_add_media(my_sess, "audio", 49170, 1, "RTP/AVP",
"0", &my_media) != 0)
```

示例 3–1 构建 SDP 会话结构 （续）

```
goto err_ret;
   if (sdp_add_information(&my_media->m_info, "audio media") != 0)
goto err_ret;
   if (sdp_add_bandwidth(&my_media->m_bw, "CT", 1000) != 0)
goto err_ret;
   if (sdp_add_key(&my_media->m_key, "prompt", NULL) != 0)
goto err_ret;
   if (sdp_add_media(my_sess, "video", 51732, 1, "RTP/AVP",
 "99 90", &my_media) != 0)
goto err_ret;
   if (sdp_add_information(&my_media->m_info, "video media") != 0)
goto err_ret;
   if (sdp_add_attribute(&my_media->m_attr, "rtpmap",
      "99 h232-199/90000") != 0)
goto err_ret;
   if (sdp_add_attribute(&my_media->m_attr, "rtpmap",
      "90 h263-1998/90000") != 0)
goto err_ret;
   b_sdp = sdp_session_to_str(my_sess, &error);

   /*
    * b_sdp is the string representation of my_sess structure
    */

   free(b_sdp);
   sdp_free_session(my_sess);
   return (0);
err_ret:
   free(b_sdp);
   sdp_free_session(my_sess);
   return (1);
}
```

搜索 SDP 会话结构

本节中述及的函数可在 SDP 会话结构中搜索特定值，并返回指向这些值的指针。

在 SDP 会话结构中查找属性

`sdp_attr_t` ***sdp_find_attribute*** `(sdp_attr_t` **attr*, `const char` **name*`);`

`sdp_find_attribute()` 函数可在 *attr* 参数指定的属性列表中搜索 *name* 参数指定的属性名称。

返回值：如果 `sdp_find_attribute()` 函数成功完成，该函数将返回指向 *name* 参数指定的属性 (`sdp_attr_t *`) 的指针。在其他所有情况下，`sdp_find_attribute()` 函数返回 NULL 值。

示例 3–2 使用 `sdp_find_attribute()` 函数

本示例中的不完整 SDP 描述包含音频部分。

示例 3–2 使用 sdp_find_attribute() 函数 （续）

```
m=audio 49170 RTP/AVP 0 8
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000
a=sendonly
a=ptime:10000
a=maxptime:20000

/*
* Assuming that above description is parsed using sdp_parse and that
* the parsed structure is in "session" sdp_session_t structure.
*/

sdp_attr_t *ptime;
sdp_attr_t *max_ptime;
sdp_media_t *media = session->s_media;

if ((ptime = sdp_find_attribute(media->m_attr, "ptime")) == NULL)
/* ptime attribute not present */
else if((max_ptime = sdp_find_attribute(media->m_attr,
 "maxptime")) == NULL)
/* max_ptime attribute not present */
```

在 SDP 会话结构中查找介质

sdp_media_t ***sdp_find_media**(sdp_media_t **media*, const char **name*);

sdp_find_media() 函数可在 *media* 参数指定的介质列表中搜索 *name* 参数指定的介质条目。

返回值：如果 sdp_find_media() 函数成功完成，该函数将返回指向 *name* 参数指定的介质列表条目 (sdp_media_t *) 的指针。在其他所有情况下，sdp_find_media() 函数返回 NULL 值。

示例 3–3 使用 sdp_find_media() 函数

本示例中的不完整 SDP 描述包含两个部分，分别是音频部分和视频部分。

```
m=audio 49170 RTP/AVP 0 8
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000
m=video 51372 RTP/AVP 31 32
a=rtpmap:31 H261/90000
a=rtpmap:32 MPV/90000

/*
* Assuming that above description is parsed using sdp_parse() and that
* the parsed structure is in "session" sdp_session_t structure.
*/

sdp_media_t     *my_media;
my_media = sdp_find_media(session->s_media, "video");

/*
```

示例 3–3 使用 sdp_find_media() 函数 （续）

```
* my_media now points to the structure containg video media section
* information
*/
```

在 SDP 会话结构中查找介质格式

```
sdp_attr_t *sdp_find_media_rtpmap (sdp_media_t *media, const char *format);
```

sdp_find_media_rtpmap() 函数可在 *media* 参数指定的介质结构的属性列表中搜索 *format* 参数指定的格式条目。

返回值：如果 sdp_find_media_rtpmap() 函数成功完成，该函数将返回指向 *name* 参数指定的格式条目 (sdp_attr_t *) 的指针。在其他所有情况下，sdp_find_media() 函数返回 NULL 值。

示例 3–4 使用 sdp_find_media_rtpmap() 函数

本示例中的不完整 SDP 描述包含两个部分，分别是音频部分和视频部分。

```
m=audio 49170 RTP/AVP 0 8
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000
m=video 51372 RTP/AVP 31 32
a=rtpmap:31 H261/90000
a=rtpmap:32 MPV/90000

/*
* Assuming that above description is parsed using sdp_parse() and that
* the parsed structure is in "session" sdp_session_t structure.
*/

sdp_media_t     *video;
sdp_attr_t      *mpv;

video = sdp_find_media(session->s_media, "video);
mpv = sdp_find_media_rtpmap(video, "32");

/*
* Now the attribute structure sdp_attr_t, mpv will be having
* values from the attribute field "a=rtpmap:32 MPV/90000"
*/
```

关闭 SDP 会话结构

本节中述及的函数可实现以下功能：

- 从 SDP 会话结构中删除字段
- 释放 SDP 会话结构

从 SDP 会话结构中删除字段

```
int sdp_delete_all_field(sdp_session_t *session, const char field);
```

sdp_delete_all_field() 函数可从 SDP 结构中删除 *field* 参数指定的所出现的所有 SDP 字段。例如，如果 SDP 结构具有三个 BANDWIDTH (b=) 字段，使用 field 参数中的值 SDP_BANDWIDTH_FIELD 调用此函数可从会话结构中删除所有三个 BANDWIDTH 字段。

返回值：如果 sdp_delete_all_field() 函数成功完成，该函数将返回 0。如果会话参数为 NULL 或字段类型未知，该函数将返回 EINVAL。在出现错误时，errno 的值不会发生更改。

从 SDP 介质结构中删除字段

```
int sdp_delete_all_media_field(sdp_media_t *media, const char field);
```

sdp_delete_all_media_field() 函数可从 SDP 介质结构中删除 *field* 参数指定的所出现的所有 SDP 字段。

返回值：如果 sdp_delete_all_media_field() 函数成功完成，该函数将返回 0。如果会话参数为 NULL 或字段类型未知，该函数将返回 EINVAL。在出现错误时，errno 的值不会发生更改。

从 SDP 介质结构中删除介质

```
int sdp_delete_media(sdp_media_t **l_media, sdp_media_t *media);
```

sdp_delete_media() 函数可从介质列表中删除 *media* 参数指定的介质条目。此函数可通过调用 sdp_find_media() 函数来查找指定的介质条目。此函数将在删除介质条目之后释放分配给介质结构的内存。

返回值：如果 sdp_delete_media() 函数成功完成，该函数将返回 0。如果会话参数为 NULL 或强制参数不存在，该函数将返回 EINVAL。在出现错误时，errno 的值不会发生更改。

从 SDP 介质结构中删除属性

```
int sdp_delete_attribute(sdp_attr_t **l_attr, sdp_attr_t *attr );
```

sdp_delete_attribute() 函数可从介质列表中删除 *attr* 参数指定的属性。此函数可通过调用 sdp_find_media_rtpmap() 函数或 sdp_find_attribute() 函数来查找指定的属性。此函数将在删除该属性之后释放分配给属性结构的内存。

返回值：如果 sdp_delete_attribute() 函数成功完成，该函数将返回 0。如果会话参数为 NULL 或强制参数不存在，该函数将返回 EINVAL。在出现错误时，errno 的值不会发生更改。

从 SDP 介质结构中删除属性

```
void sdp_free_session(sdp_session_t *session);
```

sdp_free_session() 函数可销毁 *session* 参数指定的会话，并释放与该结构关联的资源。

SDP API 实用程序函数

本节中述及的函数可解析并填充 SDP 会话结构、克隆现有会话，以及将现有会话转换为字符串表示形式。

解析 SDP 会话结构

int **sdp_parse**(const char **sdp_info*, int *len*, int *flags*, sdp_session_t ***session*, uint_t **p_error*);

sdp_parse() 函数可解析 *sdp_info* 参数中的 SDP 描述并填充 *sdp_session_t* 结构。*len* 参数可指定字符缓冲区 *sdp_info* 的长度。该函数可分配 *sdp_session_t* 结构所需的内存。要释放该内存，请调用 sdp_free_session() 函数。

flags 参数的值必须设置为零。如果 *flags* 参数具有非零值，sdp_parse() 函数将失败并返回值 EINVAL，并且会将 *session 的值设置为 NULL。

p_error 参数可采用具有解析错误的任何字段的值。此参数不能采用 NULL 值。下面列出了 *p_error* 参数的可能值：

```
SDP_VERSION_ERROR             0x00000001
SDP_ORIGIN_ERROR              0x00000002
SDP_NAME_ERROR                0x00000004
SDP_INFO_ERROR                0x00000008
SDP_URI_ERROR                 0x00000010
SDP_EMAIL_ERROR               0x00000020
SDP_PHONE_ERROR               0x00000040
SDP_CONNECTION_ERROR          0x00000080
SDP_BANDWIDTH_ERROR           0x00000100
SDP_TIME_ERROR                0x00000200
SDP_REPEAT_TIME_ERROR         0x00000400
SDP_ZONE_ERROR                0x00000800
SDP_KEY_ERROR                 0x00001000
SDP_ATTRIBUTE_ERROR           0x00002000
SDP_MEDIA_ERROR               0x00004000
SDP_FIELDS_ORDER_ERROR        0x00008000
SDP_MISSING_FIELDS            0x00010000
```

如果 SDP 结构中的字段次序颠倒，与 RFC 4566 有冲突，sdp_parse() 函数会将 *p_error* 参数的值设置为 SDP_FIELDS_ORDER_ERROR。如果 SDP 结构中缺少必填字段，与 RFC 4566 有冲突，sdp_parse() 函数会将 *p_error* 参数的值设置为 SDP_MISSING_FIELDS。

sdp_parse() 函数在处理具有解析错误的字段之后将继续进行解析，但是具有解析错误的字段将不会显示在生成的 sdp_session_t 结构中。

返回值：如果 sdp_parse() 函数成功完成，该函数将返回 0。如果会话参数无效，sdp_parse() 函数将返回 EINVAL。如果内存分配在 sdp_parse() 函数正在解析 sdp_info 时失败，该函数将返回 ENOMEM。在出现错误时，errno 的值不会发生更改。

示例 3–5 示例：解析 SDP 会话结构

在本示例中，SDP 会话结构如下所示：

```
v=0\r\n
o=jdoe 23423423 234234234 IN IP4 192.168.1.1\r\n
s=SDP seminar\r\n
i=A seminar on the session description protocol\r\n
e=test@host.com
c=IN IP4 156.78.90.1\r\n
t=2873397496 2873404696\r\n
```

在调用 sdp_parse_t() 函数之后，生成的 sdp_session_t 结构如下所示：

```
session {
        sdp_session_version = 1
        s_version = 0
        s_origin {
                o_username = "jdoe"
                o_id = 23423423ULL
                o_version = 234234234ULL
                o_nettype = "IN"
                o_addrtype = "IP4"
                o_address = "192.168.1.1"
        }
        s_name = "SDP seminar"
        s_info = "A seminar on the session description protocol"
        s_uri =  (nil)
        s_email {
                value = "test@host.com"
                next = (nil)
        }
        s_phone = (nil)
        s_conn {
                c_nettype = "IN"
                c_addrtype = "IP4"
                c_address = "156.78.90.1"
                c_addrcount = 0
                c_ttl = 0
                c_next = (nil)
        }
        s_bw = (nil)
        s_time {
                t_start = 2873397496ULL
                t_stop = 2873404696ULL
                t_repeat = (nil)
                t_next = (nil)
        }
        s_zone = (nil)
        s_key = (nil)
        s_attr = (nil)
```

示例 3–5 示例：解析 SDP 会话结构 （续）

```
        s_media = (nil)
}
```

克隆现有的 SDP 会话结构

```
sdp_session_t sdp_clone_session(const sdp_session_t *session );
```

sdp_clone_session() 函数可创建新的 SDP 会话结构，该会话结构与 *session* 参数中指定的 SDP 会话结构相同。sdp_clone_session() 函数将在成功完成时返回克隆的会话结构。sdp_clone_session() 函数将在失败时返回 NULL。

将 SDP 会话结构转换为字符串

```
char *sdp_session_to_str(const sdp_session_t *session, int *error);
```

sdp_session_to_str() 函数返回 *session* 参数指定的 SDP 会话结构的字符串表示形式。sdp_session_to_str() 函数先将回车/换行符附加到每个 SDP 字段的结尾，然后再将此字段附加到字符串。

返回值：sdp_session_to_str() 函数将在成功完成时返回 SDP 会话结构的字符串表示形式。在其他所有情况下，sdp_session_to_str() 函数将返回 NULL。如果输入为空，sdp_session_to_str() 函数将返回指向 EINVAL 的错误指针。如果内存分配失败，sdp_session_to_str() 函数将返回指向 ENOMEM 的错误指针。在出现错误时，errno 的值不会发生更改。

◆ ◆ ◆ 第 4 章

进程调度程序

本章介绍了进程的调度以及如何修改调度。

- 第 65 页中的 “调度程序概述”概述了调度程序和分时调度类。此外，还简要介绍了其他调度类。
- 第 68 页中的 “命令和接口”介绍了用于修改调度的命令和接口。
- 第 71 页中的 “与其他接口交互”介绍了调度更改对内核进程和特定接口的影响。
- 第 72 页中的 “调度和系统性能”介绍了使用这些命令或接口时要考虑的性能问题。

本章内容适用于相对缺省调度所提供的控制而言需要更多地控制进程执行顺序的开发者。有关多线程调度的说明，请参见《多线程编程指南》。

调度程序概述

创建进程时，系统将为其指定一个轻量级进程 (Lightweight Process, LWP)。如果此进程是多线程进程，则可能会为其指定更多 LWP。LWP 是被 UNIX 系统调度程序所调度的对象，它确定了何时运行进程。调度程序可维护基于配置参数、进程行为和用户请求的进程优先级。调度程序使用这些优先级来确定下一个要运行的进程。共有六种优先级类，分别是实时、系统、交互式 (interactive, IA)、固定优先级 (fixed-priority, FX)、公平份额 (fair-share, FSS) 和分时 (time-sharing, TS)。

缺省调度采用分时策略。此策略可动态调整进程优先级，以平衡交互式进程的响应时间，另外还可动态调整优先级，以平衡使用大量 CPU 时间的进程的吞吐量。分时类的优先级最低。

SunOS 5.10 调度程序还提供了一种实时调度策略。通过实时调度，用户可以为特定进程指定固定优先级。可运行的优先级最高的实时用户进程总是会获取 CPU。

SunOS 5.10 调度程序还提供了一种固定优先级调度策略。通过固定优先级调度，用户可以为特定进程指定固定优先级。缺省情况下，固定优先级调度使用的优先级范围与分时调度类相同。

可以在编写程序时保证实时进程从系统及时获取响应。有关详细信息，请参见第 12 章，实时编程和管理。

仅在极少数情况下才需要实时调度所提供的进程调度控制。但是，如果某个程序的要求中包括严格的时间安排约束，则可能只有实时进程才可以满足这些约束。

注意 – 不加考虑地使用实时进程可能会对分时进程的性能产生严重的负面影响。

由于调度程序管理方面的更改可能会影响调度程序的行为，因此程序员可能还需要对调度程序管理有一些了解。以下接口会影响调度程序管理：

- `dispadmin(1M)`，用于显示或更改运行的系统中的调度程序配置。
- `ts_dptbl(4)` 和 `rt_dptbl(4)`，包含用于配置调度程序的分时参数和实时参数的表。

创建进程时，进程将继承其调度参数，包括调度类以及此类中的优先级。进程仅在用户请求时才会更改类。系统会根据用户请求以及与进程的调度程序类相关联的策略来调整进程的优先级。

在缺省配置中，初始化进程属于分时类。因此，所有用户登录 shell 都以分时进程开始。

调度程序可将特定类的优先级转换为全局优先级。进程的全局优先级可确定何时运行此进程。调度程序始终运行全局优先级最高的可运行进程。优先级越高，运行时间便越早。指定给 CPU 的进程在进程休眠、用完其时间片或被优先级更高的进程抢占之前会一直运行。具有相同优先级的进程将按顺序循环运行。

所有实时进程的优先级都高于内核进程的优先级，所有内核进程的优先级都高于分时进程的优先级。

注 – 在单处理器系统中，如果存在可运行的实时进程，则不会运行任何内核进程和分时进程。

管理员可在配置表中指定缺省时间片。用户可以按进程为实时进程指定时间片。

可以使用 `ps(1)` 命令的 `-cl` 选项显示进程的全局优先级。可以使用 `priocntl(1)` 命令和 `dispadmin(1M)` 命令显示有关特定于类的优先级的配置信息。

以下各节将介绍六种调度类的调度策略。

分时类

分时策略的目标是为交互式进程提供快速响应，并为计算密集 (CPU-bound) 的进程提供大吞吐量。调度程序会按合适的频率切换 CPU 分配，既保证提供快速响应，同时又不使系统花费大量时间进行切换。时间片通常只有几百毫秒。

分时策略可动态更改优先级，并指定不同长度的时间片。对于使用 CPU 较短时间后便休眠的进程，调度程序会提升其优先级。例如，进程启动 I/O 操作（如终端读取或磁盘读取）时会进入休眠状态。频繁休眠是交互式任务（如编辑和运行简单的 shell 命令）的特点。对于长时间使用 CPU 而不休眠的进程，分时策略会降低其优先级。

分时策略缺省情况下会为优先级较低的进程提供较大的时间片。低优先级进程可能是计算密集 (CPU-bound) 的进程。其他进程将首先获取 CPU，但是当优先级低的进程最终获取 CPU 之后，此进程会获取较大的时间片。但是，如果优先级较高的进程在某个时间片内进入可运行状态，则优先级较高的进程会抢占运行的进程所获取的 CPU。

全局进程优先级和用户提供的优先级按升序排列：优先级越高，运行时间便越早。用户优先级的范围是从负的配置相关最大值到正的配置相关最大值。进程可继承其用户的优先级。缺省的初始用户优先级为零。

“用户优先级限制”是指用户优先级的配置相关最大值。可以将用户优先级设置为低于用户优先级限制的任意值。如果具有适当的权限，则可以提升用户优先级限制。缺省的用户优先级限制为零。

您可以降低进程的用户优先级，以减少进程对 CPU 的访问。或者，如果具有适当的权限，则还可以提升用户优先级以获取更快的服务。但是不能将用户优先级设置为高于用户优先级限制的值。因此，如果用户优先级限制和用户优先级的缺省值均为零，则必须先提升用户优先级限制才能提升用户优先级。

管理员配置最高用户优先级时可以不考虑全局分时优先级。例如，在缺省配置中，用户可以在 –20 到 +20 范围内设置用户优先级。但是，此时会设置 60 种分时全局优先级。

调度程序使用分时参数表 `ts_dptbl(4)` 中的可配置参数来管理分时进程。此表包含特定于分时类的信息。

系统类

系统类使用固定优先级策略运行内核进程（如服务器）和内务处理进程（如换页守护进程）。系统类保留供内核使用。用户不能向系统类中添加进程，也不能从系统类中删除进程。系统类进程的优先级在内核代码中进行设置。系统进程的优先级一经建立便不再更改。在内核模式下运行的用户进程不属于系统类。

实时类

实时类使用具有固定优先级的调度策略，以便关键进程按照预先确定的顺序运行。除非用户请求进行更改，否则实时优先级永远不会更改。特权用户可以使用 `priocntl(1)` 命令或 `priocntl(2)` 接口指定实时优先级。

调度程序使用实时参数表 `rt_dptbl(4)` 中的可配置参数来管理实时进程。此表包含特定于实时类的信息。

交互式类

IA 类与 TS 类非常相似。如果将交互式类与窗口系统一起使用，则在具有输入焦点的窗口中运行时，进程会具有较高的优先级。系统运行窗口系统时，IA 类即为缺省类。否则，IA 类会与 TS 类完全相同，并且这两种类共享同一个 `ts_dptbl` 分发参数表。

公平份额类

FSS 类由公平份额调度程序 (`FSS(7)`) 使用，通过将 CPU 资源份额显式分配给各项目来管理应用程序性能。份额用于指示项目访问可用 CPU 资源的权利。系统会跟踪一段时间内的资源使用情况。如果使用大量资源，则系统将削弱使用权。如果使用较少的资源，则系统将增强使用权。FSS 根据进程属主的使用权在进程之间调度 CPU 时间，而不会考虑每个项目拥有的进程数。FSS 类使用的优先级范围与 TS 类和 IA 类相同。有关更多详细信息，请参见 FSS 手册页。

固定优先级类

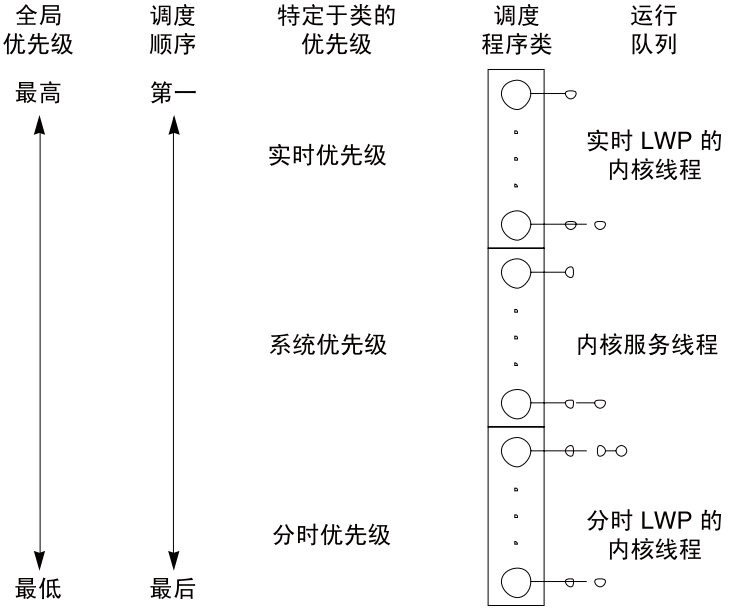
FX 类提供了固定优先级抢占调度策略。此策略由需要通过用户或应用程序控制调度优先级（而不是通过系统动态调整优先级）的进程使用。缺省情况下，FX 类的优先级范围与 TS 类、IA 类和 FSS 类相同。FX 类允许用户或应用程序通过指定给此类中的进程的用户优先级值来控制调度优先级。这些用户优先级值可确定某个固定优先级进程相对于其所属类中的其他进程的调度优先级。

调度程序使用固定优先级分发参数表 `fx_dptbl(4)` 中的可配置参数来管理固定优先级进程。此表包含特定于固定优先级类的信息。

命令和接口

下图说明了缺省的进程优先级。

图 4–1 进程优先级（从程序员的角度考虑）



进程优先级仅在调度程序类的上下文中才有意义。可以通过指定类和特定于类的优先级值来指定进程优先级。系统会将类和特定于类的值映射成其用来调度进程的全局优先级。

从系统管理员的角度考虑的优先级与从用户或程序员的角度考虑的优先级不同。配置调度程序类时，管理员直接处理全局优先级。系统会将用户提供的优先级映射成这些全局优先级。有关优先级的更多信息，请参见《Oracle Solaris 管理：基本管理》。

带有 `-cel` 选项的 `ps(1)` 命令可用于报告所有活动进程的全局优先级。`priocntl(1)` 命令可用于报告用户和程序员使用的特定于类的优先级。

`priocntl(1)` 命令以及 `priocntl(2)` 和 `priocntlset(2)` 接口用于设置或检索进程的调度程序参数。使用上述命令和两个接口设置优先级的顺序基本相同：

1. 指定目标进程。
2. 指定要用于这些进程的调度程序参数。
3. 执行命令或接口，为进程设置参数。

进程 ID 是 UNIX 进程的基本属性。有关更多信息，请参见 `Intro(2)`。类 ID 是指进程的调度程序类。`priocntl(2)` 仅适用于分时和实时类，不适用于系统类。

`priocntl` 用法

`priocntl(1)` 实用程序可在调度进程时执行四个不同的控制接口：

`priocntl -l` 显示配置信息

`priocntl -d` 显示进程的调度参数

`priocntl -s` 设置进程的调度参数

`priocntl -e` 执行带有指定调度参数的命令

以下示例说明了 `priocntl(1)` 的用法。

- 用于缺省配置的 `-l` 选项会生成以下输出：

```
$ priocntl -l
CONFIGURED CLASSES
==================

SYS (System Class)

TS (Time Sharing)
Configured TS User Priority Range -60 through 60

RT (Real Time)
Maximum Configured RT Priority: 59
```

- 要显示有关所有进程的信息，请执行以下命令：

```
$ priocntl -d -i all
```

- 要显示有关所有分时进程的信息，请执行以下命令：

```
$ priocntl -d -i class TS
```

- 要显示有关用户 ID 为 103 或 6626 的所有进程的信息，请执行以下命令：

```
$ priocntl -d -i uid 103 6626
```

- 要使 ID 为 24668 的进程成为具有缺省参数的实时进程，请执行以下命令：

```
$ priocntl -s -c RT -i pid 24668
```

- 要生成优先级为 55 的 3608 RT 和五分之一秒的时间片，请执行以下命令：

```
$ priocntl -s -c RT -p 55 -t 1 -r 5 -i pid 3608
```

- 要将所有进程更改为分时进程，请执行以下命令：

```
$ priocntl -s -c TS -i all
```

- 要将 UID 1122 的 TS 用户优先级和用户优先级限制减小到 -10，请执行以下命令：

```
$ priocntl -s -c TS -p -10 -m -10 -i uid 1122
```

- 要启动具有缺省实时优先级的实时 shell，请执行以下命令：

```
$ priocntl -e -c RT /bin/sh
```

- 要使用分时用户优先级 -10 运行 `make`，请执行以下命令：

```
$ priocntl -e -c TS -p -10 make bigprog
```

`priocntl(1)` 包括 `nice(1)` 的接口。`nice` 仅适用于分时进程，并使用较高的数值来指定较低的优先级。上一个示例相当于使用 `nice(1)` 将增量设置为 10：

```
$ nice -10 make bigprog
```

priocntl 接口

priocntl(2) 用于管理一个或一组进程的调度参数。可以针对 LWP、单个进程或一组进程调用 priocntl(2)。一组进程可以通过父进程、进程组、会话、用户、组、类或所有活动进程进行标识。有关更多详细信息，请参见 priocntl 手册页。

如果给定类 ID，则 PC_GETCLINFO 命令可以获取调度程序类名称和参数。使用此命令，在编写程序时就不用假设需要对哪些类进行配置。

PC_SETXPARMS 命令用于设置一组进程的调度程序类和参数。idtype 和 id 输入参数用于指定要更改的进程。

与其他接口交互

更改 TS 类中某个进程的优先级可能会影响该 TS 类中其他进程的行为。本节介绍可能会影响其他进程的调度更改方法。

内核进程

内核守护进程和内务处理进程是系统调度程序类的成员。用户既不能在此类中添加或删除进程，也不能更改这些进程的优先级。命令 ps -cel 可用于列出所有进程的调度程序类。运行带有 -f 选项的 ps(1) 时，可以通过 CLS 列中的 SYS 项识别系统类中的进程。

使用 fork 和 exec

执行 fork(2) 和 exec(2) 接口时将继承调度程序类、优先级和其他调度程序参数。

使用 nice

nice(1) 命令和 nice(2) 接口的使用方式与在早期版本的 UNIX 系统中的使用方式相同。使用这些命令可以更改分时进程的优先级。通过这些接口，使用较小的数字值可指定较高的分时优先级。

要更改进程的调度程序类或指定实时优先级，请使用 priocntl(2)。使用较大的数字值可指定较高的优先级。

init(1M)

init(1M) 进程是调度程序的一个特例。要更改 init(1M) 的调度属性，init 必须是 idtype 和 id 或者 procset 结构所指定的唯一进程。

调度和系统性能

调度程序可确定进程运行的时间以及持续长度。因此，调度程序的行为会对系统性能产生重要影响。

缺省情况下，所有用户进程都是分时进程。进程只能通过 priocntl(2) 调用来更改类。

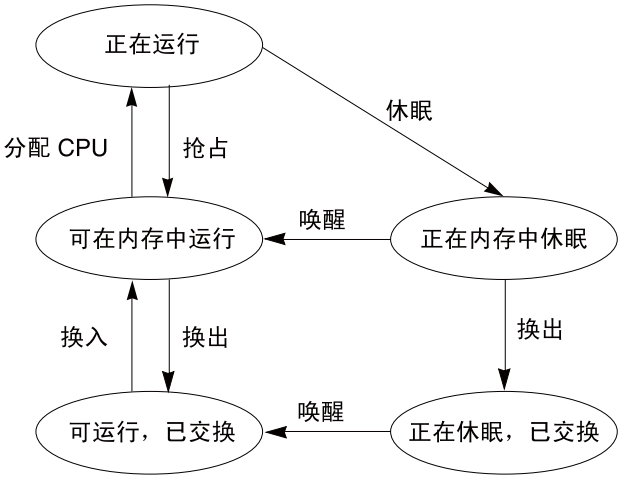
所有实时进程的优先级都高于分时进程的优先级。如果有任何实时进程可以运行，则不能运行分时进程或系统进程。有时无法放弃 CPU 控制权的实时应用程序可能会完全禁止其他用户和基本内核内务处理对其进行使用。

除了控制进程类和优先级之外，实时应用程序还必须控制影响其性能的其他因素。性能方面最重要的因素包括 CPU 处理能力、主存储器量和 I/O 吞吐量。这些因素会以很复杂的方式相互作用相互影响。sar(1) 命令可以使用一些选项来报告所有性能因素。

进程状态转换

具有严格实时约束的应用程序可能需要阻止将进程换出或换页到辅助内存。下图简要概括了 UNIX 进程状态以及状态之间的转换。

图 4–2 进程状态转换图



活动进程通常处于图中的五种状态之一。箭头指示了进程状态如何进行变化。

- 如果将进程指定给 CPU，则进程将处于运行状态。如果另一个优先级较高的进程进入可运行状态，则调度程序将使该进程脱离运行状态。当一个进程用尽其整个时间片之后，如果另一个具有相同优先级的进程进入可运行状态，则也会抢占该进程。
- 如果进程位于主存储器中并且准备运行，但是未指定给 CPU，则此进程在内存中处于可运行状态。
- 如果进程位于主存储器中，但是需要等待特定的事件完成才能继续执行，则此进程在内存中处于休眠状态。例如，进程在等待 I/O 操作完成、解除锁定已锁定的资源或者计时器到期时，将处于休眠状态。事件发生时，将向进程发送唤醒调用。如果进程进入休眠状态的原因不复存在，则进程将进入可运行状态。
- 如果某个进程的地址空间已写入到辅助存储器中，并且该进程未等待特定事件，则该进程处于可运行状态，并会进行交换。
- 如果某个进程正在等待特定事件，并且已将其整个地址空间写入到辅助存储器中，则该进程处于休眠状态，并会进行交换。

  如果计算机中没有足够的主存储器容纳其所有活动进程，则此计算机必须将部分地址空间交换或换页到辅助存储器。
- 如果系统的主存储器不足，系统会将某些进程的个别页面写入辅助存储器，从而使这些进程仍可继续运行。当正在运行的进程访问这些页面时，此进程将休眠，同时将页面读回到主存储器中。
- 当系统遇到更严重的主存储器不足时，系统会将某些进程的所有页面都写入辅助存储器，并将已写入辅助存储器的页面标记为已交换。仅当系统调度程序守护进程选择将此类进程读回到内存中时，才能调度这些进程。

当进程可以再次运行时，换页和交换都会造成延迟。对于具有严格的时间安排要求的进程而言，这种延迟可能是无法接受的。

为了避免交换延迟，实时进程永远不能进行交换，尽管某些实时进程可以进行换页。程序可以通过将其文本和数据锁入到主存储器中来阻止换页和交换。有关更多信息，请参见 `memcntl(2)` 手册页。可以锁定的内存量受限于已配置的内存量。另外，锁定过多内存可能会对未将文本和数据锁入内存的进程造成无法忍受的延迟。

实时进程与其他进程之间的性能权衡取决于局部需求。在某些系统中，可能需要进行进程锁定才能保证必需的实时响应。

注 – 有关实时应用程序延迟的信息，请参见第 244 页中的“分发延迟”。

◆ ◆ ◆ 第 5 章

5 地址组 API

本章介绍了应用程序应用程序用来与地址组进行交互的 API。

本章讨论以下主题：

- 第 75 页中的“地址组概述”介绍了地址组抽象概念。
- 第 78 页中的“验证接口版本”介绍了用于提供接口信息的函数。
- 第 78 页中的“初始化地址组接口”介绍了用于初始化和关闭接口部分（用于遍历地址组分层结构以及搜索地址组内容）的函数调用。
- 第 79 页中的“地址组分层结构”介绍了用于导航地址组分层结构的函数调用以及获取地址组分层结构特征的函数。
- 第 81 页中的“地址组内容”介绍了用于检索地址组内容信息的函数调用。
- 第 83 页中的“地址组特征”介绍了用于检索地址组特征信息的函数调用。
- 第 84 页中的“地址组及线程和内存位置”介绍了如何影响线程及其内存的地址组位置。
- 第 90 页中的“API 用法示例”包含使用本章介绍的 API 执行示例任务的代码。

地址组概述

共享内存多处理器计算机包含多个 CPU。每个 CPU 均可访问计算机中的所有内存。在某些共享内存多处理器中，内存体系结构使每个 CPU 访问某些内存区域的速度快于访问其他区域的速度。

当具有此类内存体系结构的计算机运行 Solaris 软件时，将有关给定 CPU 与给定内存区域之间最短访问时间的信息提供给内核可以提高系统性能。地址组 (lgroup) 抽象概念就是为处理这些信息而引入的。lgroup 抽象概念是内存位置优化 (Memory Placement Optimization, MPO) 功能的一部分。

lgroup 是指包含 CPU 类和内存类设备的集合，此集合中的每个 CPU 均可在限定的延迟间隔内访问此集合中的任何内存。延迟间隔的值表示此 lgroup 中所有 CPU 与所有内存

之间的最低公用延迟。定义 lgroup 的延迟界限并未限制此 lgroup 的成员之间的最大延迟。延迟界限的值是适用于此组中所有可能的 CPU-内存对的最小延迟。

lgroup 具有分层结构。lgroup 分层结构是一个有向无环图 (Directed Acyclic Graph, DAG)，与树类似，只不过一个 lgroup 可能具有多个父级。根 lgroup 包含系统中的所有资源，并且可以包含子 lgroup。此外，根 lgroup 还有一个特点，即它的延迟值是系统内所有 lgroup 中的最大值。所有子 lgroup 的延迟值均小于根 lgroup 的延迟值。越接近根的 lgroup 的延迟值越大，越接近叶的 lgroup 的延迟值越小。

如果一台计算机中的所有 CPU 访问所有内存所需的时间相同，则可以使用一个 lgroup 表示此计算机（请参见图 5–1）。如果一台计算机中的某些 CPU 访问某些内存区域所需的时间短于访问其他内存区域所需的时间，则可以使用多个 lgroup 表示此计算机（请参见图 5–2）。

图 5–1 单地址组示意图

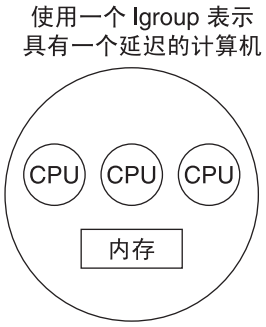
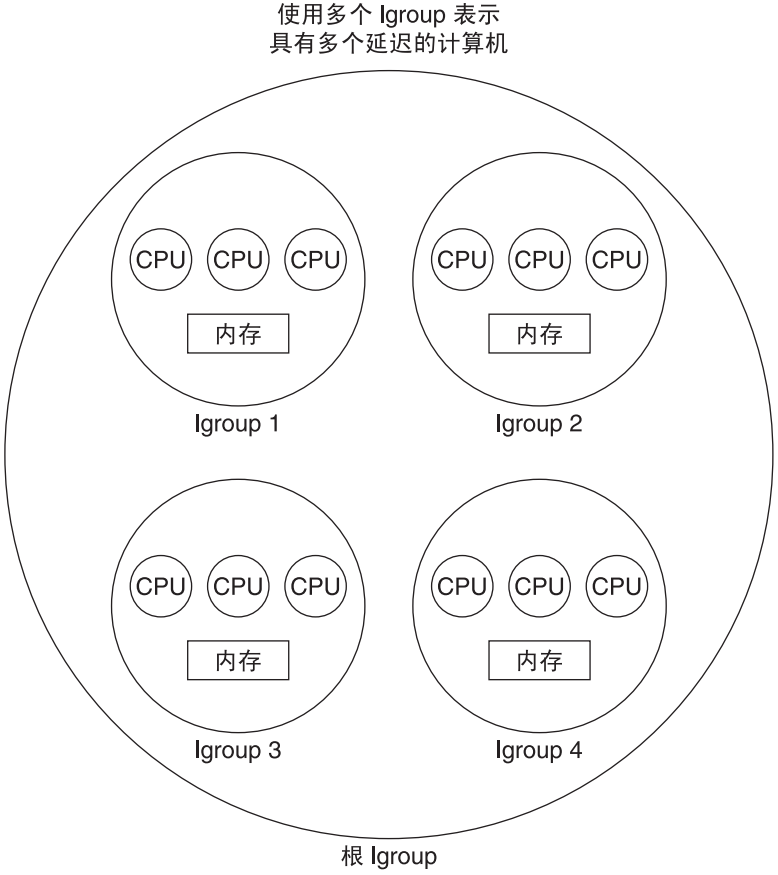


图 5–2 多地址组示意图



lgroup 分层结构的组织简化了在系统中查找最近资源的任务。创建每个线程时即为其指定了一个主 lgroup。缺省情况下，操作系统将尝试从线程的主 lgroup 为线程分配资源。例如，缺省情况下，Solaris 内核将尝试安排线程在其主 lgroup 中的 CPU 上运行，并从其主 lgroup 中分配此线程的内存。如果线程的主 lgroup 中没有所需的资源，则内核可以遍历 lgroup 分层结构，从主 lgroup 的父级中查找下一个最近的资源。如果主 lgroup 的父级中没有所需的资源，则内核将继续遍历 lgroup 分层结构，在主 lgroup 的上一级祖先 lgroup 中进行查找。根 lgroup 是计算机中所有其他 lgroup 的最后一级祖先，它包含计算机中的所有资源。

lgroup API 针对应用程序引入了 lgroup 抽象概念，以便进行观察和性能调优。新 API 包含在名为 `liblgrp` 的新库中。应用程序可以使用 API 执行以下任务：

- 遍历组分层结构
- 搜索给定 lgroup 的内容和特征
- 影响 lgroup 的线程和内存位置

验证接口版本

在使用 lgroup API 之前，必须使用 lgrp_version(3LGRP) 函数验证是否存在受支持的 lgroup 接口。lgrp_version() 函数的语法如下所示：

```
#include <sys/lgrp_user.h>
int lgrp_version(const int version);
```

lgrp_version() 函数使用 lgroup 接口的版本号作为参数，并返回系统支持的 lgroup 接口版本。如果 lgroup API 的当前实现支持 version 参数中的版本号，则 lgrp_version() 函数将返回该版本号。否则， lgrp_version() 函数将返回 LGRP_VER_NONE。

示例 5–1 lgrp_version() 用法示例

```
#include <sys/lgrp_user.h>
if (lgrp_version(LGRP_VER_CURRENT) != LGRP_VER_CURRENT) {
    fprintf(stderr, "Built with unsupported lgroup interface %d\n",
        LGRP_VER_CURRENT);
    exit (1);
}
```

初始化地址组接口

应用程序必须调用 lgrp_init(3LGRP)，才能使用 API 遍历 lgroup 分层结构并搜索 lgroup 分层结构的内容。调用 lgrp_init() 可为应用程序提供 lgroup 分层结构的一致快照。应用程序开发者可以指定快照仅包含专供调用线程使用的资源，还是包含一般情况下操作系统均可使用的资源。lgrp_init() 函数将返回一个 cookie，用于执行以下任务：

- 导航 lgroup 分层结构
- 确定 lgroup 的内容
- 确定快照是否为最新

使用 lgrp_init()

lgrp_init() 函数可初始化 lgroup 接口，并获取 lgroup 分层结构的快照。

```
#include <sys/lgrp_user.h>
lgrp_cookie_t lgrp_init(lgrp_view_t view);
```

如果在调用 lgrp_init() 函数时使用 LGRP_VIEW_CALLER 作为视图，则该函数返回的快照将仅包含可供调用线程使用的资源。如果在调用 lgrp_init() 函数时使用 LGRP_VIEW_OS 作为视图，则该函数返回的快照将仅包含可供操作系统使用的资源。在线程成功调用 lgrp_init() 函数后，该函数将返回一个 cookie，与 lgroup 分层结构进行交互的任何函数均使用此 cookie。当线程不再需要此 cookie 时，将使用此 cookie 作为参数来调用 lgrp_fini() 函数。

lgroup 分层结构包括一个根 lgroup，其中包含计算机的所有 CPU 和内存资源。根 lgroup 可以包含其他由更小延迟限定的地址组。

`lgrp_init()` 函数可能会返回两个错误。如果视图无效，则该函数返回 `EINVAL`。如果没有足够内存用于分配 lgroup 分层结构的快照，则该函数返回 `ENOMEM`。

使用 lgrp_fini()

`lgrp_fini(3LGRP)` 函数可终止使用给定的 cookie，并释放相应的 lgroup 分层结构快照。

```
#include <sys/lgrp_user.h>
int lgrp_fini(lgrp_cookie_t cookie);
```

`lgrp_fini()` 函数使用表示由先前的 `lgrp_init()` 调用创建的 lgroup 分层结构快照的 cookie。`lgrp_fini()` 函数可释放分配给该快照的内存。调用 `lgrp_fini()` 后，此 cookie 将失效。请不要再使用此 cookie。

如果传递给 `lgrp_fini()` 函数的 cookie 无效，则 `lgrp_fini()` 将返回 `EINVAL`。

地址组分层结构

使用本节介绍的 API，调用线程可以导航 lgroup 分层结构。lgroup 分层结构是一种类似于树结构的有向无环图，只不过一个节点可能具有多个父级。根 lgroup 表示整个计算机，其中包含此计算机的所有资源。根 lgroup 是系统中具有最大延迟值的 lgroup。每个子 lgroup 均包含根 lgroup 中的硬件子集，并且由一个较小的延迟值限定。越接近根的地址组具有的资源越多，延迟也越大。越接近叶的地址组具有的资源越少，延迟也越小。lgroup 可以包含直接位于其延迟界限内的资源，还可以包含具有其自己的资源集合的叶 lgroup。叶 lgroup 的资源可供封装这些叶 lgroup 的 lgroup 使用。

使用 lgrp_cookie_stale()

`lgrp_cookie_stale(3LGRP)` 函数可确定由给定 cookie 表示的 lgroup 分层结构的快照是否为最新。

```
#include <sys/lgrp_user.h>
int lgrp_cookie_stale(lgrp_cookie_t cookie);
```

`lgrp_init()` 函数返回的 cookie 可能会因多种原因而过时，具体取决于快照所表示的视图。当视图设置为 `LGRP_VIEW_OS` 时，调用 `lgrp_init()` 函数所返回的 cookie 可能会过时，原因包括 lgroup 分层结构的更改（如动态重新配置）或CPU 联机状态的更改。当视图设置为 `LGRP_VIEW_CALLER` 时，调用 `lgrp_init ()` 函数所返回的 cookie 可能会过时，原因是调用线程处理器集的更改或 lgroup 分层结构的更改。要刷新过时的 cookie，可以使用旧 cookie 调用 `lgrp_fini()` 函数，然后调用 `lgrp_init()` 函数生成一个新 cookie。

如果给定的 cookie 无效，则 lgrp_cookie_stale() 函数将返回 EINVAL。

使用 lgrp_view()

lgrp_view(3LGRP) 函数可确定用于获取给定 lgroup 分层结构快照的视图。

```
#include <sys/lgrp_user.h>
lgrp_view_t lgrp_view(lgrp_cookie_t cookie);
```

lgrp_view() 函数使用表示 lgroup 分层结构快照的 cookie，并返回该 lgroup 分层结构的快照视图。使用视图 LGRP_VIEW_CALLER 获取的快照仅包含可供调用线程使用的资源。使用视图 LGRP_VIEW_OS 获取的快照包含可供操作系统使用的所有资源。

如果给定的 cookie 无效，则 lgrp_view() 函数将返回 EINVAL。

使用 lgrp_nlgrps()

lgrp_nlgrps(3LGRP) 函数返回系统中的地址组数。如果系统中只有一个地址组，则内存位置优化没有任何效果。

```
#include <sys/lgrp_user.h>
int lgrp_nlgrps(lgrp_cookie_t cookie);
```

lgrp_nlgrps() 函数使用表示 lgroup 分层结构快照的 cookie，并返回此分层结构中的可用 lgroup 数。

如果此 cookie 无效，则 lgrp_nlgrps() 函数将返回 EINVAL。

使用 lgrp_root()

lgrp_root(3LGRP) 函数返回根 lgroup ID。

```
#include <sys/lgrp_user.h>
lgrp_id_t lgrp_root(lgrp_cookie_t cookie);
```

lgrp_root() 函数使用表示 lgroup 分层结构快照的 cookie，并返回根 lgroup ID。

使用 lgrp_parents()

lgrp_parents(3LGRP) 函数使用表示 lgroup 分层结构快照的 cookie，并返回指定 lgroup 的父 lgroup 数。

```
#include <sys/lgrp_user.h>
int lgrp_parents(lgrp_cookie_t cookie, lgrp_id_t child,
                 lgrp_id_t *lgrp_array, uint_t lgrp_array_size);
```

如果 lgrp_array 不为 NULL，并且 lgrp_array_size 的值不为零，则 lgrp_parents () 函数将使用父 lgroup ID 填充数组，直到填满数组或者所有父 lgroup ID 均在数组中为止。根 lgroup 没有父级。如果针对根 lgroup 调用 lgrp_parents() 函数，则不会填充 lgrp_array。

如果此 cookie 无效，则 lgrp_parents() 函数将返回 EINVAL。如果找不到指定的 lgroup ID，则 lgrp_parents() 函数将返回 ESRCH。

使用 lgrp_children()

lgrp_children(3LGRP) 函数使用表示调用线程的 lgroup 分层结构快照的 cookie，并返回指定 lgroup 的子 lgroup 数。

```
#include <sys/lgrp_user.h>
int lgrp_children(lgrp_cookie_t cookie, lgrp_id_t parent,
                  lgrp_id_t *lgrp_array, uint_t lgrp_array_size);
```

如果 lgrp_array 不为 NULL，并且 lgrp_array_size 的值不为零，则 lgrp_children () 函数将使用子 lgroup ID 填充数组，直到填满数组或者所有子 lgroup ID 均在数组中为止。

如果此 cookie 无效，则 lgrp_children() 函数将返回 EINVAL。如果找不到指定的 lgroup ID，则 lgrp_children() 函数将返回 ESRCH。

地址组内容

以下 API 检索有关给定 lgroup 内容的信息。

lgroup 分层结构对域资源进行了组织，以便简化查找最近资源的过程。叶 lgroup 使用具有最小延迟的资源进行定义。给定叶 lgroup 的每个上一级祖先 lgroup 均包含距离其子 lgroup 下一个最近的资源。根 lgroup 包含域中的所有资源。

给定 lgroup 的资源直接包含在此 lgroup 内，或者间接包含在此给定 lgroup 封装的叶 lgroup 内。叶 lgroup 直接包含各自的资源，而不封装其他任何 lgroup。

使用 lgrp_resources()

lgrp_resources() 函数返回指定 lgroup 中包含的资源数。

```
#include <sys/lgrp_user.h>
int lgrp_resources(lgrp_cookie_t cookie, lgrp_id_t lgrp, lgrp_id_t *lgrpids,
                   uint_t count, lgrp_rsrc_t type);
```

lgrp_resources() 函数使用表示 lgroup 分层结构快照的 cookie。此 cookie 是从 lgrp_init() 函数中获取的。lgrp_resources() 函数返回 ID 由 lgrp 参数值指定的

lgroup 中的资源数。lgrp_resources () 函数表示直接包含 CPU 或内存资源的 lgroup 集合中的资源。lgrp_rsrc_t 参数可以具有以下两个值：

LGRP_RSRC_CPU　　lgrp_resources() 函数返回 CPU 资源数。

LGRP_RSRC_MEM　　lgrp_resources() 函数返回内存资源数。

如果在 lgrpids[] 参数中传递的值不为空，并且 count 参数不为零，则 lgrp_resources() 函数将在 lgrpids[] 数组中存储 lgroup ID。此数组中存储的 lgroup ID 数最多不能超过 count 参数的值。

如果指定的 cookie、lgroup ID 或类型无效，则 lgrp_resources() 函数将返回 EINVAL。如果 lgrp_resources() 函数找不到指定的 lgroup ID，则该函数将返回 ESRCH。

使用 lgrp_cpus()

lgrp_cpus(3LGRP) 函数使用表示 lgroup 分层结构快照的 cookie，并返回给定 lgroup 中的 CPU 数。

```
#include <sys/lgrp_user.h>
int lgrp_cpus(lgrp_cookie_t cookie, lgrp_id_t lgrp, processorid_t *cpuids,
              uint_t count, int content);
```

如果 *cpuid[]* 参数不为 NULL，并且 CPU 计数不为零，则 lgrp_cpus() 函数将使用 CPU ID 填充数组，直到填满数组或者所有 CPU ID 均在数组中为止。

content 参数可以具有以下两个值：

LGRP_CONTENT_ALL　　lgrp_cpus() 函数返回此 lgroup 及其后代中的 CPU 的 ID。

LGRP_CONTENT_DIRECT　　lgrp_cpus() 函数只返回此 lgroup 中的 CPU 的 ID。

如果 cookie、lgroup ID 或其中一个标志无效，则 lgrp_cpus() 函数将返回 EINVAL。如果找不到指定的 lgroup ID，则 lgrp_cpus() 函数将返回 ESRCH。

使用 lgrp_mem_size()

lgrp_mem_size(3LGRP) 函数使用表示 lgroup 分层结构快照的 cookie，并返回给定 lgroup 中已安装内存或可用内存的大小。lgrp_mem_size() 函数报告内存大小（字节）。

```
#include <sys/lgrp_user.h>
lgrp_mem_size_t lgrp_mem_size(lgrp_cookie_t cookie, lgrp_id_t lgrp,
                              int type, int content)
```

type 参数可以具有以下两个值：

LGRP_MEM_SZ_FREE　　lgrp_mem_size() 函数返回可用内存量（字节）。

LGRP_MEM_SZ_INSTALLED　　lgrp_mem_size() 函数返回已安装内存量（字节）。

content 参数可以具有以下两个值：

LGRP_CONTENT_ALL　　lgrp_mem_size() 函数返回此 lgroup 及其后代中的内存量。

LGRP_CONTENT_DIRECT　　lgrp_mem_size() 函数只返回此 lgroup 中的内存量。

如果 cookie、lgroup ID 或其中一个标志无效，则 lgrp_mem_size() 函数将返回 EINVAL。如果找不到指定的 lgroup ID，则 lgrp_mem_size() 函数将返回 ESRCH。

地址组特征

以下 API 检索有关给定 lgroup 特征的信息。

使用 lgrp_latency_cookie()

lgrp_latency(3LGRP) 函数返回一个 lgroup 中的 CPU 与另一个 lgroup 中的内存之间的延迟。

```
#include <sys/lgrp_user.h>
int lgrp_latency_cookie(lgrp_cookie_t cookie, lgrp_id_t from, lgrp_id_t to.
                        lat_between_t between);
```

lgrp_latency_cookie() 函数使用表示 lgroup 分层结构快照的 cookie。此 cookie 由 lgrp_init() 函数创建。lgrp_latency_cookie() 函数返回一个值，此值表示 *from* 参数值所指定的 lgroup 中的硬件资源与 *to* 参数值所指定的 lgroup 中的硬件资源之间的延迟。如果这两个参数指向同一个 lgroup，则 lgrp_latency_cookie() 函数将返回此 lgroup 内的延迟值。

注 – lgrp_latency_cookie() 函数返回的延迟值由操作系统定义，并且特定于平台。此值不一定表示硬件设备之间的实际延迟，只用于在同一个域内进行比较。

如果 *between* 参数的值为 LGRP_LAT_CPU_TO_MEM，则 lgrp_latency_cookie() 函数将测量 CPU 资源与内存资源之间的延迟。

如果 lgroup ID 无效，则 lgrp_latency_cookie() 函数将返回 EINVAL。如果 lgrp_latency_cookie() 函数找不到指定的 lgroup ID，并且“源”lgroup 中不包含任何 CPU 或者“目标”lgroup 中没有任何内存，则 lgrp_latency_cookie() 函数将返回 ESRCH。

地址组及线程和内存位置

本节讨论用于搜索和影响与 lgroup 有关的线程和内存位置的 API。

- 使用 lgrp_home(3LGRP) 函数搜索线程位置。
- 使用 meminfo(2) 系统调用搜索内存位置。
- 使用 madvise(3C) 函数的 MADV_ACCESS 标志影响 lgroup 之间的内存分配。
- 通过设置线程与给定 lgroup 的关联，lgrp_affinity_set(3LGRP) 函数可以影响线程和内存位置。
- lgroup 关联可以指定从中分配资源的 lgroup 的优先级顺序。
- 内核需要有关应用程序可能的内存使用模式的信息，才能有效地分配内存资源。
- 这些信息由 madvise() 函数及其类似共享对象 madv.so.1 提供给内核。
- 正在运行的进程可通过使用 meminfo() 系统调用来收集自身的内存使用情况信息。

使用 lgrp_home()

lgrp_home() 函数返回指定进程或线程的主 lgroup。

```
#include <sys/lgrp_user.h>
lgrp_id_t lgrp_home(idtype_t idtype, id_t id);
```

如果 ID 类型无效，则 lgrp_home() 函数将返回 EINVAL。如果调用进程的有效用户不是超级用户，并且调用进程的实际或有效用户 ID 与其中一个线程的实际或有效用户 ID 不匹配，则 lgrp_home() 函数将返回 EPERM。如果找不到指定的进程或线程，则 lgrp_home() 函数将返回 ESRCH。

使用 madvise()

madvise() 函数建议内核，在从 *addr* 指定的地址开始，长度等于 *len* 参数值的范围内，该区域的用户虚拟内存应遵循特定的使用模式。内核使用这些信息优化与指定范围关联的资源的处理和维护过程。如果使用 madvise() 函数的程序明确了解其内存访问模式，则使用此函数可以提高系统性能。

```
#include <sys/types.h>
#include <sys/mman.h>
int madvise(caddr_t addr, size_t len, int advice);
```

madvise() 函数提供了以下标志，这些标志影响 lgroup 之间线程内存的分配方式：

MADV_ACCESS_DEFAULT　　此标志将指定范围的内核预期访问模式重置为缺省设置。

MADV_ACCESS_LWP	此标志通知内核，移近指定地址范围的下一个 LWP 就是将要访问此范围次数最多的 LWP。内核将相应地为此范围和 LWP 分配内存和其他资源。
MADV_ACCESS_MANY	此标志建议内核，许多进程或 LWP 将在系统内随机访问指定的地址范围。内核将相应地为此范围分配内存和其他资源。

madvise() 函数可返回以下值：

EAGAIN	指定地址范围（从 *addr* 到 *addr+len*）中的部分或所有映射均已锁定进行 I/O 操作。
EINVAL	*addr* 参数的值不是 sysconf(3C) 返回的页面大小的倍数、指定地址范围的长度小于或等于零或者建议无效。
EIO	从文件系统读取或向文件系统写入时发生 I/O 错误。
ENOMEM	指定地址范围中的地址不在进程的有效地址空间范围内，或者指定地址范围中的地址指定了一个或多个未映射的页面。
ESTALE	NFS 文件句柄过时。

使用 madv.so.1

通过 madv.so.1 共享对象，可以为已启动的进程及其后代有选择地配置虚拟内存建议。要使用此共享对象，环境中必须存在以下字符串：

```
LD_PRELOAD=$LD_PRELOAD:madv.so.1
```

madv.so.1 共享对象应用由 MADV 环境变量的值指定的内存建议。MADV 环境变量指定用于进程地址空间中的所有堆、共享内存和 mmap 区域的虚拟内存建议。此建议将应用于所有已创建进程。MADV 环境变量的以下值影响 lgroup 之间的资源分配：

access_default	此值将内核的预期访问模式重置为缺省设置。
access_lwp	此值通知内核，移近地址范围的下一个 LWP 就是将要访问此范围次数最多的 LWP。内核将相应地为此范围和 LWP 分配内存和其他资源。
access_many	此值建议内核，许多进程或 LWP 将在系统内随机访问内存。内核将相应地分配内存和其他资源。

MADVCFGFILE 环境变量的值是一个文本文件的名称，此文件中包含一个或多个格式为 *exec-name*:*advice-opts* 的内存建议配置项。

exec-name 的值是应用程序或可执行文件的名称。*exec-name* 的值可以是全路径名、基本名称或模式字符串。

advice-opts 值的格式为 *region=advice*。*advice* 的值与 MADV 环境变量的值相同。可将 *region* 替换为以下任一合法值：

madv	建议应用于进程地址空间中的所有堆、共享内存和 mmap(2) 区域。
heap	堆被定义为 brk(2) 区域。建议应用于现有堆以及将来分配的任何其他堆内存。
shm	建议应用于共享内存段。有关共享内存操作的更多信息，请参见 shmat(2)。
ism	建议应用于使用 SHM_SHARE_MMU 标志的共享内存段。ism 选项优先于 shm。
dsm	建议应用于使用 SHM_PAGEABLE 标志的共享内存段。dsm 选项优先于 shm。
mapshared	建议应用于由 mmap() 系统调用使用 MAP_SHARED 标志建立的映射。
mapprivate	建议应用于由 mmap() 系统调用使用 MAP_PRIVATE 标志建立的映射。
mapanon	建议应用于由 mmap() 系统调用使用 MAP_ANON 标志建立的映射。当多个选项都适用时，mapanon 选项优先。

MADVERRFILE 环境变量的值是在其中记录错误消息的路径的名称。如果缺少 MADVERRFILE 位置，则 madv.so.1 共享对象将使用 syslog(3C) 记录错误，而使用 LOG_ERR 作为严重级别，使用 LOG_USER 作为功能描述符。

内存建议将被继承。子进程具有与其父进程相同的建议。调用 exec(2) 之后，会将此建议重置为系统缺省建议，除非使用 madv.so.1 共享对象配置了不同级别的建议。建议只应用于由用户程序显式创建的 mmap() 区域。由运行时链接程序或执行直接系统调用的系统库建立的区域不受影响。

madv.so.1 用法示例

以下示例说明了 madv.so.1 共享对象的特定方面。

示例 5–2 为一组应用程序设置建议

此配置将建议应用于 exec 名称以 foo 开头的应用程序的所有 ISM 段。

```
$ LD_PRELOAD=$LD_PRELOAD:madv.so.1
$ MADVCFGFILE=madvcfg
$ export LD_PRELOAD MADVCFGFILE
$ cat $MADVCFGFILE
        foo*:ism=access_lwp
```

示例 5–3 从建议中排除一组应用程序

此配置为除 ls 之外的所有应用程序设置建议。

示例 5–3 从建议中排除一组应用程序 （续）

```
$ LD_PRELOAD=$LD_PRELOAD:madv.so.1
$ MADV=access_many
$ MADVCFGFILE=madvcfg
$ export LD_PRELOAD MADV MADVCFGFILE
$ cat $MADVCFGFILE
        ls:
```

示例 5–4 配置文件中的模式匹配

由于 MADVCFGFILE 中指定的配置优先于在 MADV 中设置的值，因此指定 * 作为最后一个配置项的 *exec-name* 相当于设置 MADV。本示例与上一个示例等效。

```
$ LD_PRELOAD=$LD_PRELOAD:madv.so.1
$ MADVCFGFILE=madvcfg
$ export LD_PRELOAD MADVCFGFILE
$ cat $MADVCFGFILE
        ls:
        *:madv=access_many
```

示例 5–5 针对多个区域的建议

对于 exec() 名称以 foo 开头的应用程序，此配置会为 mmap() 区域应用一种类型的建议，为堆和共享内存区域应用不同的建议。

```
$ LD_PRELOAD=$LD_PRELOAD:madv.so.1
$ MADVCFGFILE=madvcfg
$ export LD_PRELOAD MADVCFGFILE
$ cat $MADVCFGFILE
        foo*:madv=access_many,heap=sequential,shm=access_lwp
```

使用 meminfo()

meminfo() 函数为调用进程提供有关系统已分配给此进程的虚拟内存和物理内存的信息。

```
#include <sys/types.h>
#include <sys/mman.h>
int meminfo(const uint64_t inaddr[], int addr_count,
    const uint_t info_req[], int info_count, uint64_t outdata[],
    uint_t validity[]);
```

meminfo() 函数可返回以下类型的信息：

MEMINFO_VPHYSICAL	与给定虚拟地址对应的物理内存地址
MEMINFO_VLGRP	与给定虚拟地址对应的物理页所属的 lgroup
MEMINFO_VPAGESIZE	与给定虚拟地址对应的物理页的大小

MEMINFO_VREPLCNT	与给定虚拟地址对应的已复制物理页数
MEMINFO_VREPL|n	给定虚拟地址的第 *n* 个物理副本
MEMINFO_VREPL_LGRP|n	给定虚拟地址的第 *n* 个物理副本所属的 lgroup
MEMINFO_PLGRP	给定物理地址所属的 lgroup

meminfo() 函数使用以下参数：

inaddr	输入地址数组。
addr_count	传递给 meminfo() 的地址数。
info_req	列出所请求的信息类型的数组。
info_count	为 *inaddr* 数组中每个地址请求的信息数。
outdata	meminfo() 函数用于存放结果的数组。此数组的大小等于 *info_req* 参数值与 *addr_count* 参数值的乘积。
validity	大小等于 *addr_count* 参数值的数组。*validity* 数组包含按位结果代码。结果代码的第 *0* 位评估相应输入地址的有效性。结果代码中的每个后续位依次评估对 *info_req* 数组成员的响应的有效性。

如果无法向 *outdata* 或 *validity* 数组所指向的内存区域进行写入，则 meminfo() 函数将返回 EFAULT。如果无法从 *info_req* 或 *inaddr* 数组所指向的内存区域进行读取，则 meminfo() 函数将返回 EFAULT。如果 *info_count* 的值超过 31 或小于 1，则 meminfo() 函数将返回 EINVAL。如果 *addr_count* 的值小于零，则 meminfo() 函数将返回 EINVAL。

示例 5–6 使用 meminfo() 输出与一组虚拟地址对应的物理页和页面大小

```
void
print_info(void **addrvec, int how_many)
{
        static const int info[] = {
                MEMINFO_VPHYSICAL,
                MEMINFO_VPAGESIZE};
        uint64_t * inaddr = alloca(sizeof(uint64_t) * how_many);
        uint64_t * outdata = alloca(sizeof(uint64_t) * how_many * 2;
        uint_t * validity = alloca(sizeof(uint_t) * how_many);

        int i;

        for (i = 0; i < how_many; i++)
                inaddr[i] = (uint64_t *)addr[i];

        if (meminfo(inaddr, how_many,  info,
                    sizeof (info)/ sizeof(info[0]),
                    outdata, validity) < 0)
                ...

        for (i = 0; i < how_many; i++) {
                if (validity[i] & 1 == 0)
                        printf("address 0x%llx not part of address
```

示例 5–6 使用 meminfo() 输出与一组虚拟地址对应的物理页和页面大小 （续）

```
                                        space\n",
                                inaddr[i]);

                else if (validity[i] & 2 == 0)
                        printf("address 0x%llx has no physical page
                                        associated with it\n",
                                inaddr[i]);

                else {
                        char buff[80];
                        if (validity[i] & 4 == 0)
                                strcpy(buff, "<Unknown>");
                        else
                                sprintf(buff, "%lld", outdata[i * 2 +
                                        1]);
                        printf("address 0x%llx is backed by physical
                                        page 0x%llx of size %s\n",
                                        inaddr[i], outdata[i * 2], buff);
                }
        }
}
```

地址组关联

为线程创建轻量级进程 (Lightweight Process, LWP) 时，内核会将此线程指定给一个地址组。此 lgroup 称为线程的**主** *lgroup*。内核在线程的主 lgroup 中的 CPU 上运行线程，并尽量从此 lgroup 中分配内存。如果主 lgroup 中的资源不可用，则内核将从其他 lgroup 中分配资源。如果一个线程与多个 lgroup 关联，则操作系统将按关联强度选择 lgroup 并从中分配资源。lgroup 可以具有以下三种不同的关联级别之一：

1. LGRP_AFF_STRONG 指示强关联。如果此 lgroup 是线程的主 lgroup，则操作系统将尽可能避免将其他 lgroup 重新指定为此线程的主 lgroup。有些事件（如动态重新配置、处理器脱机、处理器绑定以及处理器集绑定和处理）仍然可能导致为线程重新指定主 lgroup。
2. LGRP_AFF_WEAK 指示弱关联。如果此 lgroup 是线程的主 lgroup，则操作系统将在必要时为此线程重新指定主 lgroup 以便平衡负载。
3. LGRP_AFF_NONE 指示无关联。如果线程与任何 lgroup 均无关联，则操作系统将为此线程指定一个主 lgroup。

为给定线程分配资源时，操作系统将使用 lgroup 关联作为建议。将同时考虑此建议与其他系统约束。处理器绑定和处理器集不会更改 lgroup 关联，但是可能会限制能够运行线程的 lgroup。

使用 lgrp_affinity_get()

lgrp_affinity_get(3LGRP) 函数返回 LWP 与给定 lgroup 的关联。

```
#include <sys/lgrp_user.h>
lgrp_affinity_t lgrp_affinity_get(idtype_t idtype, id_t id, lgrp_id_t lgrp);
```

idtype 和 *id* 参数用于指定 `lgrp_affinity_get()` 函数检查的 LWP。如果 *idtype* 的值为 `P_PID`，则 `lgrp_affinity_get()` 函数将获取进程 ID 与 *id* 参数值匹配的进程中某个 LWP 与 lgroup 的关联。如果 *idtype* 的值为 `P_LWPID`，则 `lgrp_affinity_get()` 函数将获取 LWP ID 与 *id* 参数值匹配的当前进程的 LWP 与 lgroup 的关联。如果 *idtype* 的值为 `P_MYID`，则 `lgrp_affinity_get()` 函数将获取当前 LWP 与 lgroup 的关联。

如果给定的 lgroup 或 ID 类型无效，则 `lgrp_affinity_get()` 函数将返回 `EINVAL`。如果调用进程的有效用户不是超级用户，并且调用进程的 ID 与其中一个 LWP 的实际或有效用户 ID 不匹配，则 `lgrp_affinity_get()` 函数将返回 `EPERM`。如果找不到给定的 lgroup 或 LWP，则 `lgrp_affinity_get()` 函数将返回 `ESRCH`。

使用 `lgrp_affinity_set()`

`lgrp_affinity_set(3LGRP)` 函数可设置一个 LWP 或一组 LWP 与给定 lgroup 的关联。

```
#include <sys/lgrp_user.h>
int lgrp_affinity_set(idtype_t idtype, id_t id, lgrp_id_t lgrp,
                      lgrp_affinity_t affinity);
```

idtype 和 *id* 参数用于指定 `lgrp_affinity_set()` 函数检查的一个或一组 LWP。如果 *idtype* 的值为 `P_PID`，则 `lgrp_affinity_set()` 函数会将进程 ID 与 *id* 参数值匹配的进程中所有 LWP 与 lgroup 的关联设置为 *affinity* 参数中指定的关联级别。如果 *idtype* 的值为 `P_LWPID`，则 `lgrp_affinity_set()` 函数会将 LWP ID 与 *id* 参数值匹配的当前进程的 LWP 与 lgroup 的关联设置为 *affinity* 参数中指定的关联级别。如果 *idtype* 的值为 `P_MYID`，则 `lgrp_affinity_set()` 函数会将当前 LWP 或进程与 lgroup 的关联设置为 *affinity* 参数中指定的关联级别。

如果给定的 lgroup、关联或 ID 类型无效，则 `lgrp_affinity_set()` 函数将返回 `EINVAL`。如果调用进程的有效用户不是超级用户，并且调用进程的 ID 与其中一个 LWP 的实际或有效用户 ID 不匹配，则 `lgrp_affinity_set()` 函数将返回 `EPERM`。如果找不到给定的 lgroup 或 LWP，则 `lgrp_affinity_set()` 函数将返回 `ESRCH`。

API 用法示例

本节包含使用本章介绍的 API 的示例任务的代码。

示例 5–7 将内存移动到线程

以下代码样例将 *addr* 到 *addr+len* 地址范围中的内存移近下一个接近此范围的线程。

```
#include <stdio.h>
#include <sys/mman.h>
#include <sys/types.h>
```

示例 5–7 将内存移动到线程 （续）

```
/*
 * Move memory to thread
 */
void
mem_to_thread(caddr_t addr, size_t len)
{
    if (madvise(addr, len, MADV_ACCESS_LWP) < 0)
        perror("madvise");
}
```

示例 5–8 将线程移动到内存

本样例代码使用 `meminfo()` 函数确定用于在给定地址备份虚拟页的物理内存的 lgroup。然后，样例代码为此 lgroup 设置一个强关联，尝试将当前线程移近该内存。

```
#include <stdio.h>
#include <sys/lgrp_user.h>
#include <sys/mman.h>
#include <sys/types.h>

/*
 * Move a thread to memory
 */
int
thread_to_memory(caddr_t va)
{
    uint64_t    addr;
    ulong_t     count;
    lgrp_id_t   home;
    uint64_t    lgrp;
    uint_t      request;
    uint_t      valid;

    addr = (uint64_t)va;
    count = 1;
    request = MEMINFO_VLGRP;
    if (meminfo(&addr, 1, &request, 1, &lgrp, &valid) != 0) {
        perror("meminfo");
        return (1);
    }

    if (lgrp_affinity_set(P_LWPID, P_MYID, lgrp, LGRP_AFF_STRONG) != 0) {
        perror("lgrp_affinity_set");
        return (2);
    }

    home = lgrp_home(P_LWPID, P_MYID);
    if (home == -1) {
        perror ("lgrp_home");
        return (3);
    }

    if (home != lgrp)
```

示例 5–8 将线程移动到内存 （续）

```
        return (-1);

    return (0);
}
```

示例 5–9 遍历 lgroup 分层结构

以下样例代码遍历并输出 lgroup 分层结构。

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/lgrp_user.h>
#include <sys/types.h>

/*
 * Walk and print lgroup hierarchy from given lgroup
 * through all its descendants
 */
int
lgrp_walk(lgrp_cookie_t cookie, lgrp_id_t lgrp, lgrp_content_t content)
{
    lgrp_affinity_t    aff;
    lgrp_id_t          *children;
    processorid_t      *cpuids;
    int                i;
    int                ncpus;
    int                nchildren;
    int                nparents;
    lgrp_id_t          *parents;
    lgrp_mem_size_t    size;

    /*
     * Print given lgroup, caller's affinity for lgroup,
     * and desired content specified
     */
    printf("LGROUP #%d:\n", lgrp);

    aff = lgrp_affinity_get(P_LWPID, P_MYID, lgrp);
    if (aff == -1)
        perror ("lgrp_affinity_get");
    printf("\tAFFINITY: %d\n", aff);

    printf("CONTENT %d:\n", content);

    /*
     * Get CPUs
     */
    ncpus = lgrp_cpus(cookie, lgrp, NULL, 0, content);
    printf("\t%d CPUS: ", ncpus);
    if (ncpus == -1) {
        perror("lgrp_cpus");
        return (-1);
    } else if (ncpus > 0) {
        cpuids = malloc(ncpus * sizeof (processorid_t));
```

示例 5–9 遍历 lgroup 分层结构 （续）

```
        ncpus = lgrp_cpus(cookie, lgrp, cpuids, ncpus, content);
                if (ncpus == -1) {
            free(cpuids);
                            perror("lgrp_cpus");
            return (-1);
        }
        for (i = 0; i < ncpus; i++)
            printf("%d ", cpuids[i]);
        free(cpuids);
    }
    printf("\n");

    /*
     * Get memory size
     */
    printf("\tMEMORY: ");
    size = lgrp_mem_size(cookie, lgrp, LGRP_MEM_SZ_INSTALLED, content);
    if (size == -1) {
        perror("lgrp_mem_size");
        return (-1);
    }
    printf("installed bytes 0x%llx, ", size);
    size = lgrp_mem_size(cookie, lgrp, LGRP_MEM_SZ_FREE, content);
        if (size == -1) {
        perror("lgrp_mem_size");
        return (-1);
    }
    printf("free bytes 0x%llx\n", size);

    /*
     * Get parents
     */
    nparents = lgrp_parents(cookie, lgrp, NULL, 0);
    printf("\t%d PARENTS: ", nparents);
    if (nparents == -1) {
        perror("lgrp_parents");
        return (-1);
    } else if (nparents > 0) {
        parents = malloc(nparents * sizeof (lgrp_id_t));
        nparents = lgrp_parents(cookie, lgrp, parents, nparents);
                   if (nparents == -1) {
            free(parents);
                        perror("lgrp_parents");
            return (-1);
                   }
        for (i = 0; i < nparents; i++)
            printf("%d ", parents[i]);
        free(parents);
    }
    printf("\n");

    /*
     * Get children
     */
    nchildren = lgrp_children(cookie, lgrp, NULL, 0);
```

示例 5–9 遍历 lgroup 分层结构 （续）

```
    printf("\t%d CHILDREN: ", nchildren);
    if (nchildren == -1) {
        perror("lgrp_children");
        return (-1);
    } else if (nchildren > 0) {
        children = malloc(nchildren * sizeof (lgrp_id_t));
        nchildren = lgrp_children(cookie, lgrp, children, nchildren);
                    if (nchildren == -1) {
            free(children);
                        perror("lgrp_children");
            return (-1);
                    }
        printf("Children: ");
        for (i = 0; i < nchildren; i++)
            printf("%d ", children[i]);
        printf("\n");

        for (i = 0; i < nchildren; i++)
            lgrp_walk(cookie, children[i], content);

        free(children);
    }
    printf("\n");

    return (0);
}
```

示例 5–10 查找位于给定 lgroup 外具有可用内存的最近 lgroup

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/lgrp_user.h>
#include <sys/types.h>

#define    INT_MAX    2147483647

/*
 * Find next closest lgroup outside given one with available memory
 */
lgrp_id_t
lgrp_next_nearest(lgrp_cookie_t cookie, lgrp_id_t from)
{
    lgrp_id_t          closest;
    int                i;
    int                latency;
    int                lowest;
    int                nparents;
    lgrp_id_t          *parents;
    lgrp_mem_size_t    size;

    /*
     * Get number of parents
     */
    nparents = lgrp_parents(cookie, from, NULL, 0);
    if (nparents == -1) {
```

示例 5–10 查找位于给定 lgroup 外具有可用内存的最近 lgroup （续）

```
        perror("lgrp_parents");
        return (LGRP_NONE);
    }

    /*
     * No parents, so current lgroup is next nearest
     */
    if (nparents == 0) {
        return (from);
    }

    /*
     * Get parents
     */
    parents = malloc(nparents * sizeof (lgrp_id_t));
    nparents = lgrp_parents(cookie, from, parents, nparents);
    if (nparents == -1) {
        perror("lgrp_parents");
        free(parents);
        return (LGRP_NONE);
        }

    /*
     * Find closest parent (ie. the one with lowest latency)
     */
    closest = LGRP_NONE;
    lowest = INT_MAX;
    for (i = 0; i < nparents; i++) {
        lgrp_id_t    lgrp;

        /*
         * See whether parent has any free memory
         */
        size = lgrp_mem_size(cookie, parents[i], LGRP_MEM_SZ_FREE,
            LGRP_CONTENT_ALL);
        if (size > 0)
            lgrp = parents[i];
        else {
            if (size == -1)
                perror("lgrp_mem_size");

            /*
             * Find nearest ancestor if parent doesn't
             * have any memory
             */
            lgrp = lgrp_next_nearest(cookie, parents[i]);
            if (lgrp == LGRP_NONE)
                continue;
        }

        /*
         * Get latency within parent lgroup
         */
        latency = lgrp_latency_cookie(lgrp, lgrp);
        if (latency == -1) {
```

示例 5–10 查找位于给定 lgroup 外具有可用内存的最近 lgroup （续）

```
            perror("lgrp_latency_cookie");
            continue;
        }

        /*
         * Remember lgroup with lowest latency
         */
        if (latency < lowest) {
            closest = lgrp;
            lowest = latency;
        }
    }

    free(parents);
    return (closest);
}

/*
 * Find lgroup with memory nearest home lgroup of current thread
 */
lgrp_id_t
lgrp_nearest(lgrp_cookie_t cookie)
{
    lgrp_id_t    home;
    longlong_t    size;

    /*
     * Get home lgroup
     */
    home = lgrp_home(P_LWPID, P_MYID);

    /*
     * See whether home lgroup has any memory available in its hierarchy
     */
    size = lgrp_mem_size(cookie, home, LGRP_MEM_SZ_FREE,
        LGRP_CONTENT_ALL);
    if (size == -1)
        perror("lgrp_mem_size");

    /*
     * It does, so return the home lgroup.
     */
    if (size > 0)
        return (home);

    /*
     * Otherwise, find next nearest lgroup outside of the home.
     */
    return (lgrp_next_nearest(cookie, home));
}
```

示例 5–11 查找具有可用内存的最近 lgroup

本示例代码查找具有可用内存且距离给定线程的主 lgroup 最近的 lgroup。

示例 5–11 查找具有可用内存的最近 lgroup （续）

```
lgrp_id_t
lgrp_nearest(lgrp_cookie_t cookie)
{
        lgrp_id_t         home;
        longlong_t        size;

        /*
         * Get home lgroup
         */

        home = lgrp_home();

        /*
         * See whether home lgroup has any memory available in its hierarchy
         */

        if (lgrp_mem_size(cookie, lgrp, LGRP_MEM_SZ_FREE,
            LGRP_CONTENT_ALL, &size) == -1)
                perror("lgrp_mem_size");

        /*
         * It does, so return the home lgroup.
         */

        if (size > 0)
                return (home);

        /*
         * Otherwise, find next nearest lgroup outside of the home.
         */

        return (lgrp_next_nearest(cookie, home));
}
```

◆ ◆ ◆ 第 6 章

6 输入/输出接口

本章介绍了在未提供虚拟内存服务的系统上所提供的文件输入/输出操作。此外，还讨论了虚拟内存功能所提供的改进的输入/输出方法。本章在第 102 页中的 “使用文件和记录锁定”中介绍了较早的锁定文件和记录的方法。

文件和 I/O 接口

组织为一系列数据的文件称为**常规**文件。常规文件可以包含 ASCII 文本、以某些其他二进制数据编码的文本、可执行代码，或者文本、数据以及代码的任意组合。

常规文件由以下部分构成：

- 称为 *inode* 的控制数据。此数据包括文件类型、访问权限、属主、文件大小以及数据块的位置。
- 文件内容：非结束符号的字节序列。

Solaris 操作系统提供了以下基本的文件输入/输出接口形式：

- 第 99 页中的 “基本文件 I/O”中介绍了传统的原始文件 I/O 样式。
- 标准 I/O 缓冲为没有虚拟内存的系统上运行的应用程序提供了更简便的接口以及更高的效率。在虚拟内存环境中（例如在 SunOS 操作系统上）运行的应用程序中，标准文件 I/O 已过时。
- 第 15 页中的 “内存管理接口”中介绍了内存映射接口。对于在 SunOS 平台运行的大多数应用程序而言，映射文件是最有效的文件 I/O 形式。

基本文件 I/O

以下接口针对文件和字符 I/O 设备执行基本操作。

表 6–1 基本文件 I/O 接口

接口名称	目的
open(2)	打开文件进行读取或写入
close(2)	关闭文件描述符
read(2)	从文件中读取
write(2)	向文件中写入
creat(2)	创建新文件或重写现有文件
unlink(2)	删除目录项
lseek(2)	移动读/写文件指针

以下代码样例说明了基本文件 I/O 接口的用法。read(2) 和 write(2) 都从文件当前偏移位置开始传送不超过指定数量的字节。返回实际传送的字节数。read(2) 返回值零时指示已到达文件结尾。

示例 6–1 基本文件 I/O 接口

```
#include            <fcntl.h>
#define            MAXSIZE            256

main()
{
    int     fd;
    ssize_t n;
    char        array[MAXSIZE];

    fd = open ("/etc/motd", O_RDONLY);
    if (fd == -1) {
        perror ("open");
        exit (1);
    }
    while ((n = read (fd, array, MAXSIZE)) > 0)
        if (write (1, array, n) != n)
            perror ("write");
    if (n == -1)
        perror ("read");
    close (fd);
}
```

完成文件读取或写入后，应始终调用 close(2)。对于不是从 open(2) 调用返回的文件描述符，请不要调用 close(2)。

可使用 read(2)、write(2) 或通过调用 lseek(2) 来更改已打开文件的文件指针偏移。以下示例说明了 lseek 的用法。

```
off_t     start, n;
struct    record    rec;
```

```
/* record current offset in start */
start = lseek (fd, 0L, SEEK_CUR);

/* go back to start */
n = lseek (fd, -start, SEEK_SET);
read (fd, &rec, sizeof (rec));

/* rewrite previous record */
n = lseek (fd, -sizeof (rec), SEEK_CUR);
write (fd, (char *&rec, sizeof (rec));
```

高级文件I/O

下表列出了高级文件I/O接口所执行的任务。

表6–2 高级文件I/O接口

接口名称	目的
link(2)	链接到文件
access(2)	确定文件的可访问性
mknod(2)	生成特殊文件或普通文件
chmod(2)	更改文件的模式
chown(2)、lchown(2)、fchown(2)	更改文件的属主和组
utime(2)	设置文件的访问时间和修改时间
stat(2)、lstat(2)、fstat(2)	获取文件状态
fcntl(2)	执行文件控制功能
ioctl(2)	控制设备
fpathconf(2)	获取可配置的路径名变量
opendir(3C)、readdir(3C)、closedir(3C)	执行目录操作
mkdir(2)	生成目录
readlink(2)	读取符号链接的值
rename(2)	更改文件的名称
rmdir(2)	删除目录
symlink(2)	生成指向某一文件的符号链接

文件系统控制

下表中列出的文件系统控制接口可用于控制文件系统的各个方面。

表 6–3 文件系统控制接口

接口名称	目的
ustat(2)	获取文件系统统计信息
sync(2)	更新超级块
mount(2)	挂载文件系统
statvfs(2)、fstatvfs(2)	获取文件系统信息
sysfs(2)	获取文件系统类型信息

使用文件和记录锁定

不需要使用传统的文件 I/O 来锁定文件元素。应针对映射文件使用《多线程编程指南》中介绍的较轻量级同步机制。

锁定文件可防止当多个用户同时尝试更新一个文件时可能会发生的错误。可以锁定文件的一部分。

锁定文件后，将阻止对整个文件进行访问。锁定记录后，将阻止对文件中指定的段进行访问。在 SunOS 中，所有文件都是一系列的数据字节：“记录”这一概念是指使用该文件的程序。

选择锁定类型

强制性锁定将暂停进程，直到所请求的文件段被释放。建议性锁定返回一个表明是否成功获得锁定的结果。进程可以忽略建议性锁定的结果。对于同一文件，不能同时使用强制性和建议性文件锁定。打开文件时所用的模式决定了对文件的锁定是强制性锁定还是建议性锁定。

在两种基本锁定调用中，fcntl(2) 比 lockf(3C) 更容易移植、功能更强大，但更不易用。fcntl(2) 是在 POSIX 1003.1 标准中指定的。提供 lockf(3C) 是为了与较早版本的应用程序兼容。

选择建议性锁定或强制性锁定

对于强制性锁定，文件必须是设置了 set-group-ID 位而未设置组执行权限的常规文件。如果不符合其中任一条件，则所有记录锁定均为建议性锁定。

可按如下方式设置强制性锁定。

```
#include <sys/types.h>
#include <sys/stat.h>

 int mode;
 struct stat buf;
     ...
     if (stat(filename, &buf) < 0) {
         perror("program");
         exit (2);
     }
     /* get currently set mode */
     mode = buf.st_mode;
     /* remove group execute permission from mode */
     mode &= ~(S_IEXEC>>3);
         /* set 'set group id bit' in mode */
     mode |= S_ISGID;
     if (chmod(filename, mode) < 0) {
         perror("program");
         exit(2);
     }
     ...
```

当系统在执行文件时，操作系统会忽略记录锁定。所有带有记录锁定的文件都不应设置执行权限。

chmod(1) 命令还可用于将文件设置为允许强制性锁定。

```
$ chmod +l file
```

此命令在文件模式中设置 O20n0 权限位，用来指示对文件进行强制性锁定。如果 *n* 是偶数，则此位将会解释为启用强制性锁定。如果 *n* 是奇数，则此位将会解释为“执行时设置组 ID”。

当使用 –l 选项请求长列表格式时，ls(1) 命令显示以下设置：

```
$ ls -l file
```

此命令显示以下信息：

```
-rw---l--- 1 user group size mod_time file
```

权限中的字母 "l" 指示已设置 set-group-ID 位。由于已设置 set-group-ID 位，因此启用强制性锁定。此外，还启用了设置组 ID (set group ID) 的一般语义。

关于强制性锁定的注意事项

请牢记锁定的以下方面：

- 强制性锁定只对本地文件有效。当通过 NFS 访问文件时，不支持强制性锁定。
- 强制性锁定只保护文件的锁定段。可以根据一般文件权限访问文件的其余部分。

- 如果一个原子事务需要多个读取或写入操作，则进程应在任何 I/O 开始之前显式锁定所有此类段。对于所有通过此方法执行的程序，建议性锁定便已足够。
- 任意程序都不应对使用记录锁定的文件具有无限制的访问权限。
- 建议性锁定更加有效，因为不必针对每个 I/O 请求执行记录锁定检查。

支持的文件系统

下表列出的文件系统既支持建议性锁定又支持强制性锁定。

表 6–4 支持的文件系统

文件系统	说明
ufs	基于磁盘的缺省文件系统
fifofs	由命名管道文件组成的伪文件系统，这些管道文件可为进程提供对数据的通用访问权限
namefs	主要由 STREAMS 用来在文件顶部动态挂载文件描述符的伪文件系统
specfs	可用于访问特殊字符设备和块设备的伪文件系统

NFS 仅支持建议性文件锁定。proc 和 fd 文件系统不支持文件锁定。

打开文件进行锁定

只能使用有效的打开描述符来请求对文件进行锁定。对于读取锁定，文件必须至少是使用读取权限打开的。对于写入锁定，文件也必须是使用写入权限打开的。在以下示例中，将打开文件以进行读取和写入访问。

```
...
     filename = argv[1];
     fd = open (filename, O_RDWR);
     if (fd < 0) {
         perror(filename);
         exit(2);
     }
     ...
```

设置文件锁定

要锁定整个文件，请将偏移量设置为零，并将大小设置为零。

可以通过多种方法设置文件锁定。选择哪种方法取决于该锁定与程序其余部分如何交互以及性能和可移植性。本示例使用与 POSIX 标准兼容的 fcntl(2) 接口。此接口尝试锁定文件，直到发生以下情况之一：

- 成功设置文件锁定。

- 出现错误。
- 超过 MAX_TRY，程序停止尝试锁定文件。

```
#include <fcntl.h>

 ...
    struct flock lck;

 ...
    lck.l_type = F_WRLCK;    /* setting a write lock */
    lck.l_whence = 0;    /* offset l_start from beginning of file */
    lck.l_start = (off_t)0;
    lck.l_len = (off_t)0;    /* until the end of the file */
    if (fcntl(fd, F_SETLK, &lck) <0) {
        if (errno == EAGAIN || errno == EACCES) {
            (void) fprintf(stderr, "File busy try again later!\n");
            return;
        }
        perror("fcntl");
        exit (2);
    }
    ...
```

使用 fcntl(2)，可通过设置结构变量来设置锁定请求的类型和起始位置。

注 – 不能使用 flock(3UCB) 锁定映射文件。但是，可以针对映射文件使用面向多线程的同步机制。可以在 POSIX 样式和 Solaris 样式下使用这些同步机制。

设置和删除记录锁定

锁定记录时，请不要将锁定段的起始点和长度设置为零。否则，此锁定过程与文件锁定相同。

使用记录锁定的原因是存在数据争用。因此，当无法获取所有所需锁定时，应采取相应的失败响应措施：

- 等待一定时间后，再次尝试
- 中止该过程，向用户发出警告
- 使进程进入休眠状态，直到向该进程发送已释放锁定的信号
- 执行上述操作的某些组合

本示例显示了一个使用 fcntl(2) 锁定的记录。

```
{
     struct flock lck;
       ...
     lck.l_type = F_WRLCK;    /* setting a write lock */
     lck.l_whence = 0;    /* offset l_start from beginning of file */
     lck.l_start = here;
     lck.l_len = sizeof(struct record);

     /* lock "this" with write lock */
```

```
     lck.l_start = this;
     if (fcntl(fd, F_SETLKW, &lck) < 0) {
         /* "this" lock failed. */
         return (-1);
 ...
}
```

下一示例显示了 lockf(3C) 接口。

```
#include <unistd.h>

{
 ...
     /* lock "this" */
     (void) lseek(fd, this, SEEK_SET);
     if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
         /* Lock on "this" failed. Clear lock on "here". */
         (void) lseek(fd, here, 0);
         (void) lockf(fd, F_ULOCK, sizeof(struct record));
         return (-1);
}
```

可使用与设置锁定相同的方法来删除锁定。只是锁定类型有所不同 (F_ULOCK)。解除锁定不受其他进程的阻止，并且只影响调用进程所设置的锁定。解除锁定只影响在先前锁定调用中指定的文件段。

获取锁定信息

可以确定哪个进程在持有锁定。可以按照前面示例所示设置锁定，并在 fcntl(2) 中使用 F_GETLK。

下一示例查找并输出文件中所有锁定段上的标识数据。

示例 6–2 输出文件的锁定段

```
struct flock lck;

     lck.l_whence = 0;
     lck.l_start = 0L;
     lck.l_len = 0L;
     do {
         lck.l_type = F_WRLCK;
         (void) fcntl(fd, F_GETLK, &lck);
         if (lck.l_type != F_UNLCK) {
             (void) printf("%d %d %c %8ld %8ld\n", lck.l_sysid, lck.l_pid,
            (lck.l_type == F_WRLCK) ? 'W' : 'R', lck.l_start, lck.l_len);
             /* If this lock goes to the end of the address space, no
              * need to look further, so break out. */
             if (lck.l_len == 0) {
             /* else, look for new lock after the one just found. */
                     lck.l_start += lck.l_len;
             }
         }
     } while (lck.l_type != F_UNLCK);
```

fcntl(2) 与 F_GETLK 命令可以在等待服务器响应时处于休眠状态。如果客户机或服务器出现资源不足的情况，则此命令可能会失败，同时返回 ENOLCK。

将 lockf(3C) 与 F_TEST 命令一起使用来测试进程是否在持有锁定。此接口并不返回有关锁定的位置或拥有权的信息。

示例 6–3 使用 lockf 测试进程

```
(void) lseek(fd, 0, 0L);
 /* set the size of the test region to zero (0). to test until the
    end of the file address space. */
if (lockf(fd, (off_t)0, SEEK_SET) < 0) {
    switch (errno) {
        case EACCES:
        case EAGAIN:
            (void) printf("file is locked by another process\n");
            break;
        case EBADF:
            /* bad argument passed to lockf */
            perror("lockf");
            break;
        default:
            (void) printf("lockf: unexpected error <%d>\n", errno);
            break;
    }
}
```

进程派生和锁定

进程派生时，子进程会收到父进程打开的文件描述符的副本。子进程不继承锁定，因为锁定由特定进程拥有。父进程和子进程共享每个文件的公用文件指针。这两个进程都可能会尝试对同一文件中的相同位置设置锁定。使用 lockf(3C) 和 fcntl(2) 都会出现此问题。如果持有记录锁定的程序进行派生，则子进程应该关闭此文件。关闭此文件之后，子进程应该重新打开此文件以设置新的独立文件指针。

死锁处理

UNIX 锁定功能可以检测和避免死锁。仅当系统准备将记录锁定接口置于休眠状态时才会发生死锁。执行搜索来确定两个进程是否处于死锁状态。如果检测到潜在的死锁，则锁定接口将失败，并设置 errno 以指示死锁。使用 F_SETLK 设置锁定的进程不会导致死锁，因为当不能被授予锁定时，这些进程并不会等待。

终端 I/O 函数

终端 I/O 接口处理用于控制异步通信端口的通用终端接口，如下表中所示。有关更多信息，请参见 termios(3C) 和 termio(7I) 手册页。

表 6–5 终端 I/O 接口

接口名称	目的
tcgetattr(3C)、tcsetattr(3C)	获取和设置终端属性
tcsendbreak(3C)、tcdrain(3C)、tcflush(3C)、tcflow(3C)	执行行控制接口
cfgetospeed(3C)、cfgetispeed(3C)、cfsetispeed(3C)、cfsetospeed(3C)	获取和设置波特率
tcsetpgrp(3C)	获取和设置终端前台进程组 ID
tcgetsid(3C)	获取终端会话 ID

以下示例说明了服务器如何在非 DEBUG 操作模式下从其调用者的控制终端分离出来。

示例 6–4 从控制终端分离

```
(void) close(0);
(void) close(1);
(void) close(2);
(void) open("/", O_RDONLY);
(void) dup2(0, 1);
(void) dup2(0, 2);
setsid();
```

此操作模式可防止服务器从控制终端的进程组接收信号。服务器分离之后，便不能将错误报告发送到终端。已分离的服务器必须使用 syslog(3C) 记录错误。

◆ ◆ ◆ 第 7 章

7 进程间通信

本章适用于开发多进程应用程序的程序员。

SunOS 5.10 及兼容的操作系统具有多种用于并发进程交换数据以及同步执行的机制。本章介绍这些机制中除映射内存之外的所有机制。

- 第 109 页中的 “进程之间的管道”介绍了管道（匿名数据队列）。
- 命名管道（包含文件名的数据队列）。第 110 页中的 “命名管道”介绍了命名管道。
- 第 113 页中的 “System V IPC”介绍了 System V 消息队列、信号量以及共享内存。
- 第 111 页中的 “POSIX 进程间通信”介绍了 POSIX 消息队列、信号量以及共享内存。
- 第 111 页中的 “套接字概述”介绍了使用套接字的进程间通信。
- 第 15 页中的 “内存管理接口”介绍了映射内存和文件。

进程之间的管道

两个进程之间的管道是指在父进程中创建的一对文件。此管道会在父进程派生时连接生成的进程。由于管道并不存在于任何文件名称空间中，因此说管道是匿名的。虽然可通过单个管道将任意数量的子进程相互连接以及连接到其相关父进程，但是一个管道通常只连接两个进程。

管道是通过调用 `pipe(2)` 在成为父进程的进程中创建的。此调用在传递给它的数组中返回两个文件描述符。派生之后，两个进程将从 `p[0]` 中读取，向 `p[1]` 中写入。实际上，进程将从为它们管理的循环缓冲区中读取以及向其中写入。

由于调用 `fork(2)` 会复制每个进程的打开文件表，因此每个进程都有两个读取器以及两个写入器。关闭额外的读取器和写入器即可使管道正常运行。例如，如果读取器的另一端保持打开状态以便同一进程进行写入，则永远不会返回文件结束指示。以下代码说明了管道创建、派生以及清除重复的管道端。

```
#include <stdio.h>
#include <unistd.h>
...
```

```
        int p[2];
...
        if (pipe(p) == -1) exit(1);
        switch( fork() )
        {
                case 0:                  /* in child */
                        close( p[0] );
                        dup2( p[1], 1);
                        close P[1] );
                        exec( ... );
                        exit(1);
                default:                 /* in parent */
                        close( p[1] );
                        dup2( P[0], 0 );
                        close( p[0] );
                        break;
        }
        ...
```

下表显示了在特定情况下，从管道中读取以及向其中写入的结果。

表 7–1 管道中的读/写结果

尝试	情况	结果
读取	管道为空，已连接写入器	阻塞读取
写入	管道已满，已连接读取器	阻塞写入
读取	管道为空，未连接写入器	返回 EOF
写入	没有读取器	SIGPIPE

可以通过为描述符调用 fcntl(2) 来设置 FNDELAY，从而避免发生阻塞。这样会导致从 I/O 调用返回错误 (-1)，并将 errno 设置为 EWOULDBLOCK。

命名管道

命名管道的运行方式与管道非常相似，但它们是在文件系统中作为命名实体创建的。这使得所有进程均可打开命名管道，而不要求进程与管道通过派生关联。命名管道是通过调用 mknod(2) 创建的。然后，任何具有相应权限的进程均可对命名管道进行读取或写入。

在 open(2) 调用中，将会阻塞打开管道的进程，直到其他进程也打开此管道为止。

要在不发生阻塞的情况下打开命名管道，open(2) 调用应将 O_NDELAY 掩码（位于 sys/fcntl.h 中）与选定的文件模式掩码联接，方法是针对 open(2) 调用使用布尔 or 运算。如果在调用 open(2) 时没有其他进程连接到管道，则会返回 -1，并将 errno 设置为 EWOULDBLOCK。

套接字概述

套接字在两个进程之间提供点对点的双向通信。套接字是进程间通信以及系统间通信的一个基本组件。套接字是可以绑定名称的通信端点。它具有一个类型以及一个或多个关联的进程。

套接字存在于通信域中。套接字域是指提供一种寻址结构以及一组协议的抽象对象。套接字仅与同一域中的套接字连接。已确定了二十三个套接字域（请参见 `sys/socket.h`），其中通常只有 UNIX 域和 Internet 域用于 Solaris 10 及兼容的操作系统。

可以使用套接字在单个系统上的进程之间进行通信，如同其他形式的 IPC。UNIX 域 (`AF_UNIX`) 在单个系统上提供一个套接字地址空间。UNIX 域套接字以 UNIX 路径命名。附录 A，UNIX 域套接字中将对 UNIX 域套接字进行进一步介绍。套接字还可用于在不同系统上的进程之间进行通信。已连接系统之间的套接字地址空间称为 Internet 域 (`AF_INET`)。Internet 域通信使用 TCP/IP Internet 协议套件。第 8 章，套接字接口中将对 Internet 域套接字进行介绍。

POSIX 进程间通信

POSIX 进程间通信 (Interprocess Communication, IPC) 是 System V 进程间通信的变体。它是在 Solaris 7 发行版中引入的。与 System V 对象类似，POSIX IPC 对象的属主、属主的组以及其他用户具有读取和写入权限，但是没有执行权限。POSIX IPC 对象的属主无法将对象分配给其他属主。POSIX IPC 包括以下功能：

- 消息允许进程将已格式化的数据流发送到任意进程。
- 信号量允许进程同步执行。
- 共享内存允许进程共享其部分虚拟地址空间。

与 System V IPC 接口不同，POSIX IPC 接口均为多线程安全接口。

POSIX 消息

下表中列出了 POSIX 消息队列接口。

表 7–2 POSIX 消息队列接口

接口名称	目的
`mq_open(3RT)`	连接到以及（可选）创建命名消息队列
`mq_close(3RT)`	结束到开放式消息队列的连接
`mq_unlink(3RT)`	结束到开放式消息队列的连接，并在最后一个进程关闭此队列时将其删除

表 7–2 POSIX 消息队列接口 （续）

接口名称	目的
mq_send(3RT)	将消息放入队列
mq_receive(3RT)	在队列中接收（删除）最早且优先级最高的消息
mq_notify(3RT)	通知进程或线程消息已存在于队列中
mq_setattr(3RT)、mq_getattr(3RT)	设置或获取消息队列属性

POSIX 信号量

POSIX 信号量比 System V 信号量轻得多。POSIX 信号量结构定义单个信号量，而不是定义最多包含 25 个信号量的数组。

POSIX 信号量接口如下所示。

sem_open(3RT)	连接到以及（可选）创建命名信号量
sem_init(3RT)	初始化信号量结构（在调用程序内部，因此不是命名信号量）
sem_close(3RT)	结束到开放式信号量的连接
sem_unlink(3RT)	结束到开放式信号量的连接，并在最后一个进程关闭此信号量时将其删除
sem_destroy(3RT)	初始化信号量结构（在调用程序内部，因此不是命名信号量）
sem_getvalue(3RT)	将信号量的值复制到指定整数中
sem_wait(3RT)、sem_trywait(3RT)	当其他进程拥有信号量时进行阻塞，或者当其他进程拥有信号量时返回错误
sem_post(3RT)	递增信号量计数

POSIX 共享内存

POSIX 共享内存实际上是映射内存的变体（请参见第 15 页中的 “创建和使用映射”）。二者的主要差异在于：

- 打开共享内存对象应使用 shm_open(3RT)，而不是通过调用 open(2)。
- 关闭和删除对象应使用 shm_unlink(3RT)，而不是通过调用 close(2)，此调用不删除对象。

shm_open(3RT) 中的选项数实际上少于 open(2) 中提供的选项数。

System V IPC

SunOS 5.10 及兼容的操作系统还提供了 System V 进程间通信 (Interprocess Communication, IPC) 软件包。POSIX IPC 已有效地替代了 System V IPC，但仍然保留了 System V IPC 以支持较早版本的应用程序。

有关 System V IPC 的更多信息，请参见 ipcrm(1)、ipcs(1)、Intro(2)、msgctl(2)、msgget(2)、msgrcv(2)、msgsnd(2)、semget(2)、semctl(2)、semop(2)、shmget(2)、shmctl(2)、shmop(2) 和 ftok(3C) 手册页。

消息、信号量以及共享内存的权限

消息、信号量以及共享内存的属主、组以及其他用户具有读取和写入权限，但是没有执行权限，这一点与普通文件类似。与文件类似的是，创建进程会标识缺省属主。与文件不同的是，创建进程可以将此功能的拥有权指定给其他用户或撤消拥有权指定。

IPC 接口、密钥参数以及创建标志

请求访问 IPC 功能的进程必须能够标识此功能。要标识进程请求访问的功能，初始化 IPC 功能或提供 IPC 功能访问权限的接口应使用 `key_t` *key* 参数。*key* 为任意值，或者为运行时能从通用种子派生的值。派生此类密钥的一种方法是使用 ftok(3C)，它可将文件名转换为在系统内具有唯一性的密钥值。

初始化消息、信号量或共享内存或者获取消息、信号量或共享内存访问权限的接口将返回类型为 `int` 的 ID 号。执行读取、写入以及控制操作的 IPC 接口使用此 ID。

如果将密钥参数指定为 `IPC_PRIVATE`，则此调用会初始化创建进程专有的新 IPC 功能实例。

在适用于此调用的标志参数中提供 `IPC_CREAT` 标志时，如果此功能不存在，则接口会尝试创建此功能。

当使用 `IPC_CREAT` 和 `IPC_EXCL` 标志进行调用时，如果此功能已存在，则接口会失败。当多个进程可能尝试初始化此功能时，此行为会非常有用。这样一个案例可能会涉及多个具有相同功能访问权限的服务器进程。如果它们都尝试在 `IPC_EXCL` 生效的情况下创建此功能，则只有第一个尝试会成功。

如果没有提供上述任一标志并且此功能已存在，则接口会返回此功能的 ID 以获取访问权限。如果省略了 `IPC_CREAT` 并且尚未初始化此功能，则调用会失败。

将逻辑（按位）`OR`、`IPC_CREAT` 和 `IPC_EXCL` 与八进制权限模式组合使用，以形成标志参数。例如，以下语句将在队列不存在的情况下初始化一个新消息队列：

```
msqid = msgget(ftok("/tmp", 'A'), (IPC_CREAT | IPC_EXCL | 0400));
```

第一个参数基于字符串 ("/tmp") 取值为密钥 ('A')。第二个参数取值为组合的权限和控制标志。

System V 消息

必须先通过 msgget(2) 初始化队列，然后进程才能发送或接收消息。队列的属主或创建者可以使用 msgctl(2) 更改队列的拥有权或权限。任何具有相应权限的进程均可使用 msgctl(2) 执行控制操作。

通过 IPC 消息传送，进程可以发送和接收消息以及对消息进行排队，以便按任意顺序处理。与管道的文件字节流数据流不同，每条 IPC 消息都具有显式长度。

可以为消息指定特定类型。因此通过使用客户机进程 PID 作为消息类型，服务器进程可以将客户机间的消息流量定向到其队列中。对于单消息事务，多个服务器进程可以并行处理发送到共享消息队列的事务。

发送和接收消息的操作分别通过 msgsnd(2) 和 msgrcv(2) 执行。发送消息时，会将消息的文本复制到消息队列。msgsnd(2) 和 msgrcv(2) 可以作为阻塞操作或非阻塞操作执行。被阻塞的消息操作将保持暂停状态，直到出现以下三种情况之一：

- 调用成功。
- 进程接收到信号。
- 队列被删除。

初始化消息队列

msgget(2) 可初始化新的消息队列，并且还可返回对应于密钥参数的队列的消息队列 ID (msqid)。作为 msgflg 参数传递的值必须为八进制整数，并具有该队列的权限和控制标志的设置。

MSGMNI 内核配置选项确定内核支持的单一消息队列数的最大个数。如果超过此限制，msgget(2) 会失败。

以下代码说明了 msgget(2)。

```
#include <sys/ipc.h>
#include <sys/msg.h>
...
        key_t    key;         /* key to be passed to msgget() */
        int      msgflg,      /* msgflg to be passed to msgget() */
                 msqid;       /* return value from msgget() */
        ...
        key = ...
        msgflg = ...
        if ((msqid = msgget(key, msgflg)) == -1)
```

```
        {
                perror("msgget: msgget failed");
                exit(1);
        } else
                (void) fprintf(stderr, "msgget succeeded");
...
```

控制消息队列

msgctl(2) 可更改消息队列的权限和其他特性。msqid 参数必须为现有消息队列的 ID。cmd 参数是以下各项之一：

IPC_STAT　将有关队列状态的信息放入 buf 指向的数据结构中。进程必须具有读取权限，此调用才会成功。

IPC_SET　设置消息队列的属主用户和组 ID、权限以及大小（字节数）。进程必须具有属主、创建者或超级用户的有效用户 ID，此调用才会成功。

IPC_RMID　删除 msqid 参数所指定的消息队列。

以下代码说明了 msgctl(2) 及其各种标志。

```
#include                        <sys/types.h>
#include                        <sys/ipc.h>
#include                        <sys/msg.h>
        ...
        if (msgctl(msqid, IPC_STAT, &buf) == -1)  {
                perror("msgctl: msgctl failed");
                exit(1);
        }
        ...
        if (msgctl(msqid, IPC_SET, &buf) == -1) {
                perror("msgctl: msgctl failed");
                exit(1);
        }
...
```

发送和接收消息

msgsnd(2) 和 msgrcv(2) 分别发送和接收消息。msqid 参数必须为现有消息队列的 ID。msgp 参数是指向包含消息类型及其文本的结构的指针。msgsz 参数以字节为单位指定消息的长度。msgflg 参数传递各种控制标志。

以下代码说明了 msgsnd(2) 和 msgrcv(2)。

```
#include                    <sys/types.h>
#include                    <sys/ipc.h>
#include                    <sys/msg.h>
...
        int             msgflg;        /* message flags for the operation */
        struct msgbuf   *msgp;         /* pointer to the message buffer */
        size_t          msgsz;         /* message size */
        size_t          maxmsgsize;    /* maximum message size */
```

```
        long            msgtyp;         /* desired message type */
        int             msqid           /* message queue ID to be used */
        ...
        msgp = malloc(sizeof(struct msgbuf) – sizeof (msgp–>mtext)
                        + maxmsgsz);
        if (msgp == NULL) {
              (void) fprintf(stderr, "msgop: %s %ld byte messages.\n",
                        "could not allocate message buffer for", maxmsgsz);
              exit(1);
              ...
              msgsz = ...
              msgflg = ...
              if (msgsnd(msqid, msgp, msgsz, msgflg) == –1)
                        perror("msgop: msgsnd failed");
              ...
              msgsz = ...
              msgtyp = first_on_queue;
              msgflg = ...
              if (rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg) == –1)
                        perror("msgop: msgrcv failed");
...
```

System V 信号量

使用信号量，进程可以查询或更改状态信息。通常使用信号量来监视和控制系统资源（如共享内存段）的可用性。信号量既可以作为单个单元处理，也可以作为集中的元素处理。

由于 System V IPC 信号量可以存在于大型数组中，因此它们的开销非常大，线程库提供了更轻量级的信号量。此外，POSIX 信号量是 System V 信号量的最新实现（请参见第 112 页中的 “POSIX 信号量”）。线程库信号量必须与映射内存一起使用（请参见第 15 页中的 “内存管理接口”）。

信号量集由一个控制结构以及一个包含各个信号量的数组组成。一个信号量集最多可以包含 25 个元素。必须使用 `semget(2)` 初始化信号量集。信号量创建者可以使用 `semctl(2)` 更改信号量的拥有权或权限。任何具有相应权限的进程均可使用 `semctl(2)` 执行控制操作。

信号量操作通过 `semop(2)` 执行。此接口采用指向信号量操作结构数组的指针。此数组中的每个结构都包含有关针对信号量所执行的操作的数据。任何具有读取权限的进程均可测试信号量的值是否为零。递增或递减信号量的操作需要写入权限。

当操作失败时，不会更改任何信号量。除非设置了 `IPC_NOWAIT` 标志，否则进程将阻塞，直到出现以下情况之一：

- 信号量操作可全部完成，因此调用成功。
- 进程接收到信号。
- 信号量集被删除。

一次只有一个进程能更新一个信号量。不同进程同时发出的请求可以按任意顺序执行。当通过 semop(2) 调用提供操作数组时，在针对数组的所有操作均成功完成之前，不会进行更新。

如果以独占方式使用信号量的进程异常终止，并且无法撤消操作或释放信号量，则信号量将在内存中保持锁定在进程使其处于的状态。为了防止出现这种情况，SEM_UNDO 控制标志使 semop(2) 为每个信号量操作分配撤消结构，其中包含将信号量返回到其先前状态的操作。如果进程中止，则系统将应用撤消结构中的操作。这样可防止异常中止的进程使信号量集处于不一致的状态。

如果进程共享受信号量控制的资源的访问权限，则不应在 SEM_UNDO 生效时对信号量执行操作。如果当前控制资源的进程异常终止，则假定资源不一致。其他进程必须能够识别这种情况，以将资源恢复为一致状态。

如果在 SEM_UNDO 生效时执行信号量操作，则在执行反向操作的调用时也必须使 SEM_UNDO 生效。当进程正常运行时，反向操作会使用补充值更新撤消结构。这样可确保将应用于撤消结构的值抵消为零，除非进程异常中止。当撤消结构达到零时，便会被删除。

以不一致的方式使用 SEM_UNDO 可能会导致内存泄漏，因为在重新引导系统之前可能不会释放已分配的撤消结构。

初始化信号量集

semget(2) 可初始化信号量或获取其访问权限。此调用成功时，会返回信号量 ID (semid)。密钥参数是与信号量 ID 关联的一个值。nsems 参数指定信号量数组中的元素数。如果 nsems 大于现有数组中的元素数，则此调用会失败。如果不知道正确的计数，则为此参数提供 0 可确保此调用会成功。semflg 参数指定初始访问权限和创建控制标志。

SEMMNI 系统配置选项可确定所允许的信号量数组的最大数目。SEMMNS 选项可确定所有信号量集中各个信号量的最大可能数目。由于信号量集之间存在分段，因此可能无法分配所有可用信号量。

以下代码说明了 semget(2)。

```
#include                        <sys/types.h>
#include                        <sys/ipc.h>
#include                        <sys/sem.h>
...
         key_t    key;       /* key to pass to semget() */
         int      semflg;    /* semflg to pass to semget() */
         int      nsems;     /* nsems to pass to semget() */
         int      semid;     /* return value from semget() */
         ...
         key = ...
         nsems = ...
         semflg = ...
```

```
        ...
        if ((semid = semget(key, nsems, semflg)) == -1) {
                perror("semget: semget failed");
                exit(1);
        } else
                exit(0);
...
```

控制信号量

semctl(2) 可更改信号量集的权限和其他特性。进行调用时必须使用有效的信号量 ID。semnum 值按信号量索引选择数组中的信号量。*cmd* 参数为以下控制标志之一。

GETVAL	返回单个信号量的值。
SETVAL	设置单个信号量的值。在这种情况下，arg 被视为 arg.val，其类型为 int。
GETPID	返回针对信号量或数组执行上次操作的进程的 PID。
GETNCNT	返回等待信号量值增加的进程数。
GETZCNT	返回等待特定信号量的值达到零的进程数。
GETALL	返回信号量集中所有信号量的值。在这种情况下，arg 被视为 arg.array，即指向数组的指针（值为 unsigned short）。
SETALL	设置信号量集中所有信号量的值。在这种情况下，arg 被视为 arg.array，即指向数组的指针（值为 unsigned short）。
IPC_STAT	从控制结构中返回信号量集的状态信息，并将其放入 arg.buf（指向类型为 semid_ds 的缓冲区的指针）指向的数据结构中。
IPC_SET	设置有效的用户和组标识以及权限。在这种情况下，arg 被视为 arg.buf。
IPC_RMID	删除指定的信号量集。

进程必须具有属主、创建者或超级用户的有效用户标识，才能执行 IPC_SET 或 IPC_RMID 命令。与其他控制命令相同，执行上述两个命令也需要读写权限。

以下代码说明了 semctl(2)。

```
#include                    <sys/types.h>
#include                    <sys/ipc.h>
#include                    <sys/sem.h>
...
        register int        i;
...
```

```
        i = semctl(semid, semnum, cmd, arg);
        if (i == -1) {
               perror("semctl: semctl failed");
               exit(1);
...
```

信号量操作

semop(2) 可针对信号量集执行操作。semid 参数是先前 semget(2) 调用所返回的信号量 ID。sops 参数是指向结构数组的指针，其中每种结构都包含以下有关信号量操作的信息：

- 信号量数
- 要执行的操作
- 控制标志（如果存在）

sembuf 结构指定信号量操作，如 sys/sem.h 中所定义。nsops 参数指定数组的长度，其最大大小由 *SEMOPM* 配置选项确定。此选项可确定单个 semop(2) 调用所允许的最大操作数目，并缺省设置为 10。

要执行的操作按如下方式确定：

- 正整数使信号量值递增相应的值。
- 负整数使信号量值递减相应的值。尝试将信号量设置为小于零的值会失败或阻塞，具体取决于 IPC_NOWAIT 是否生效。
- 值为零表示等待信号量值达到零。

可与 semop(2) 一起使用的两个控制标志是 IPC_NOWAIT 和 SEM_UNDO。

IPC_NOWAIT　可以针对数组中的任何操作设置。如果接口无法执行任何设置了 IPC_NOWAIT 的操作，则使接口返回且不更改任何信号量值。如果接口尝试以大于信号量当前值的数量递减信号量，或者测试非零信号量等于零，则接口会失败。

SEM_UNDO　当进程存在时，允许撤消数组中的单个操作。

以下代码说明了 semop(2)。

```
#include                                <sys/types.h>
#include                                <sys/ipc.h>
#include                                <sys/sem.h>
...
         int             i;             /* work area */
         int             nsops;         /* number of operations to do */
         int             semid;         /* semid of semaphore set */
         struct sembuf   *sops;         /* ptr to operations to perform */
         ...
         if ((i = semop(semid, sops, nsops)) == -1) {
                 perror("semop: semop failed");
         } else
```

```
                (void) fprintf(stderr, "semop: returned %d\n", i);
...
```

System V 共享内存

在 SunOS 5.10 操作系统中，实现共享内存应用程序的最有效方法是依赖 mmap(2) 以及系统的本机虚拟内存功能。有关更多信息，请参见第 1 章，内存和 CPU 管理。

SunOS 5.10 平台还支持 System V 共享内存，这是一种效率较低的将物理内存段附加到多个进程的虚拟地址空间的方法。允许针对多个进程使用写入访问权限时，可以使用外部协议或机制（如信号量）防止出现不一致和冲突。

进程使用 shmget(2) 创建共享内存段。还可以使用此调用获取现有共享段的 ID。创建进程将设置段的权限和大小（字节）。

共享内存段的初始属主可以使用 shmctl(2) 将拥有权指定给其他用户，也可以撤消此指定。其他具有适当权限的进程可以使用 shmctl(2) 在共享内存段上执行各种控制功能。

创建共享段之后，便可以使用 shmat(2) 将此段附加到进程地址空间。可以使用 shmdt(2) 拆离此共享段。附加进程必须具有相应的 shmat(2) 权限。附加此段之后，进程便可以对其执行读写操作，如附加操作中请求的权限所允许的那样。共享段可以由同一进程进行多次附加。

共享内存段由具有指向物理内存区域的唯一 ID 的控制结构说明。此段的标识符称为 shmid。可在 sys/shm.h 中找到共享内存段控制结构的结构定义。

访问共享内存段

shmget(2) 用于获取对共享内存段的访问。当此调用成功时，便会返回共享内存段 ID (*shmid*)。以下代码说明了 shmget(2)。

```
#include                    <sys/types.h>
#include                    <sys/ipc.h>
#include                    <sys/shm.h>
...
        key_t     key;      /* key to be passed to shmget() */
        int       shmflg;   /* shmflg to be passed to shmget() */
        int       shmid;    /* return value from shmget() */
        size_t    size;     /* size to be passed to shmget() */
        ...
        key = ...
        size = ...
        shmflg) = ...
        if ((shmid = shmget (key, size, shmflg)) == -1) {
                perror("shmget: shmget failed");
                exit(1);
        } else {
                (void) fprintf(stderr,
                            "shmget: shmget returned %d\n", shmid);
```

```
                exit(0);
        }
...
```

控制共享内存段

shmctl(2) 用于更改共享内存段的权限和其他特性。cmd 参数为以下控制命令之一。

SHM_LOCK	锁定内存中的指定共享内存段。进程必须具有超级用户的有效 ID，才能执行此命令。
SHM_UNLOCK	解除锁定共享内存段。进程必须具有超级用户的有效 ID，才能执行此命令。
IPC_STAT	返回控制结构中包含的状态信息，并将其放入 buf 指向的缓冲区中。进程必须具有段的读取权限，才能执行此命令。
IPC_SET	设置有效的用户和组标识以及访问权限。进程必须具有属主、创建者或超级用户的有效 ID，才能执行此命令。
IPC_RMID	删除共享内存段。进程必须具有属主、创建者或超级用户的有效 ID，才能执行此命令。

以下代码说明了 shmctl(2)。

```
#include                    <sys/types.h>
#include                    <sys/ipc.h>
#include                    <sys/shm.h>
...
int     cmd;                /* command code for shmctl() */
int     shmid;              /* segment ID */
struct  shmid_ds  shmid_ds; /* shared memory data structure to hold results */
        ...
        shmid = ...
        cmd = ...
        if ((rtrn = shmctl(shmid, cmd, shmid_ds)) == -1) {
                perror("shmctl: shmctl failed");
                exit(1);
...
```

附加和拆离共享内存段

shmat() 和 shmdt() 用于附加和拆离共享内存段（请参见 shmop(2) 手册页）。shmat(2) 返回指向共享段头的指针。shmdt(2) 可拆离位于 *shmaddr* 所指示的地址中的共享内存段。以下代码说明了对 shmat(2) 和 shmdt(2) 的调用。

```
#include            <sys/types.h>
#include            <sys/ipc.h>
#include            <sys/shm.h>

static struct state { /* Internal record of attached segments. */
```

```
        int           shmid;        /* shmid of attached segment */
        char          *shmaddr;     /* attach point */
        int           shmflg;       /* flags used on attach */
} ap[MAXnap];                       /* State of current attached segments. */
int     nap;                        /* Number of currently attached segments. */
...
char    *addr;                      /* address work variable */
register int          i;            /* work area */
register struct state *p;           /* ptr to current state entry */
...
        p = &ap[nap++];
        p->shmid = ...
        p->shmaddr = ...
        p->shmflg = ...
        p->shmaddr = shmat(p->shmid, p->shmaddr, p->shmflg);
        if(p->shmaddr == (char *)-1) {
                perror("shmat failed");
                nap--;
        } else
                (void) fprintf(stderr, "shmop: shmat returned %p\n",
                                        p->shmaddr);
        ...
        i = shmdt(addr);
        if(i == -1) {
                 perror("shmdt failed");
        } else {
                 (void) fprintf(stderr, "shmop: shmdt returned %d\n", i);
                 for (p = ap, i = nap; i--; p++) {
                         if (p->shmaddr == addr) *p = ap[--nap];
                 }
        }
...
```

◆ ◆ ◆ 第 8 章

套接字接口

本章介绍套接字接口，并且包含说明要点的样例程序。本章讨论以下主题：

- 第 123 页中的 “SunOS 4 二进制兼容性”讨论与 SunOS 4 环境的二进制兼容性。
- 第 126 页中的 “套接字基础知识”讨论套接字的创建、连接以及关闭。
- 第 145 页中的 “客户机/服务器程序”讨论客户机/服务器体系结构。
- 第 149 页中的 “高级套接字主题”讨论多播和异步套接字等高级主题。
- 第 165 页中的 “流控制传输协议”讨论用于实现流控制传输协议 (Stream Control Transmission Protocol, SCTP) 的接口。

注 – 本章介绍的接口具有多线程安全性。您可以在多线程应用程序中自由调用包含套接字接口调用的应用程序。但请注意，并未指定应用程序可获得的并发度。

SunOS 4 二进制兼容性

在 SunOS 4 环境中所做的两个主要更改对 SunOS 5.10 发行版同样有效。使用二进制兼容性软件包，基于 SunOS 4 的动态链接的套接字应用程序可以在 SunOS 5.10 上运行。

- 必须在编译行中显式指定套接字库（`-lsocket` 或 `libsocket`）。
- 还可能需要使用 `-lsocket -lnsl`（而非 `-lnsl -lsocket`）与 `libnsl` 进行链接。
- 必须使用套接字库重新编译所有基于 SunOS 4 套接字的应用程序，以便在 SunOS 5.10 环境中运行。

套接字概述

自 1981 年以来，套接字已经成为 SunOS 发行版不可缺少的组成部分。套接字是可以绑定名称的通信端点。套接字具有**类型**和关联的进程。套接字是为实现用于进程间通信的客户机/服务器模型而设计的，其中：

- 网络协议的接口需要支持多个通信协议，如 TCP/IP、Xerox Internet 协议 (XNS) 以及 UNIX 系列。
- 网络协议的接口需要支持等待连接的服务器代码以及启动连接的客户机代码。
- 根据通信是面向连接的通信还是无连接通信，操作也不相同。
- 应用程序可能需要指定要传送的数据报的目标地址，而不是使用 open(2) 调用绑定此地址。

套接字使网络协议可用，并使其行为类似于 UNIX 文件。应用程序将根据需要创建套接字。套接字适用于 close(2)、read(2)、write(2)、ioctl(2) 以及 fcntl(2) 接口。操作系统可区分文件的文件描述符和套接字的文件描述符。

套接字库

套接字接口例程位于必须与应用程序链接的库中。/usr/lib 中包含库 libsocket.so 以及其余系统服务库。libsocket.so 用于动态链接。

套接字类型

套接字类型定义对于用户可见的通信属性。Internet 系列套接字提供对 TCP/IP 传输协议的访问。对于可以通过 IPv6 和 IPv4 进行通信的套接字，由值 AF_INET6 标识 Internet 系列。此外，还支持值 AF_INET，目的是为了与旧应用程序的源代码兼容并提供对 IPv4 的原始访问。

SunOS 环境支持四种类型的套接字：

- **流**套接字。使用该套接字，进程可以使用 TCP 进行通信。流套接字提供没有记录边界的双向、可靠、有序且不重复的数据流。建立连接之后，可以将数据作为字节流在这些套接字中读取或写入。套接字类型为 SOCK_STREAM。
- **数据报**套接字。使用该套接字，进程可以使用 UDP 进行通信。数据报套接字支持双向消息流。数据报套接字上的进程接收消息的顺序可能不同于发送消息的顺序。数据报套接字上的进程可能会接收重复消息。通过数据报套接字发送的消息可能会被丢弃。但数据中的记录边界会被保留。套接字类型为 SOCK_DGRAM。
- **原始**套接字。该套接字提供对 ICMP 的访问。原始套接字还提供对联网栈不直接支持的其他基于 IP 的协议的访问。虽然这些套接字的确切特征取决于协议提供的接口，但是它们通常是面向数据报的套接字。原始套接字并不适用于大多数应用程序。提供原始套接字是为了支持开发新的通信协议，或者为了访问现有协议的更加深奥的功能。只有超级用户进程才能使用原始套接字。套接字类型为 SOCK_RAW。

- *SEQ* 套接字。该套接字支持 1 对 N 流控制传输协议 (Stream Control Transfer Protocol, SCTP) 连接。有关 SCTP 的更多详细信息，请参见第 165 页中的 “流控制传输协议”。

有关详细信息，请参见第 153 页中的 “选择特定的协议”。

接口组

SunOS 5.10 平台提供两组套接字接口。提供了 BSD 套接字接口，并且从 SunOS 版本 5.7 开始，还提供了 XNS 5 (UNIX03) 套接字接口。XNS 5 接口与 BSD 接口稍有不同。

以下手册页介绍了 XNS 5 套接字接口：

- accept(3XNET)
- bind(3XNET)
- connect(3XNET)
- endhostent(3XNET)
- endnetent(3XNET)
- endprotoent(3XNET)
- endservent(3XNET)
- gethostbyaddr(3XNET)
- gethostbyname(3XNET)
- gethostent(3XNET)
- gethostname(3XNET)
- getnetbyaddr(3XNET)
- getnetbyname(3XNET)
- getnetent(3XNET)
- getpeername(3XNET)
- getprotobyname(3XNET)
- getprotobynumber(3XNET)
- getprotoent(3XNET)
- getservbyname(3XNET)
- getservbyport(3XNET)
- getservent(3XNET)
- getsockname(3XNET)
- getsockopt(3XNET)
- htonl(3XNET)
- htons(3XNET)
- inet_addr(3XNET)
- inet_lnaof(3XNET)
- inet_makeaddr(3XNET)
- inet_netof(3XNET)
- inet_network(3XNET)
- inet_ntoa(3XNET)
- listen(3XNET)

- ntohl(3XNET)
- ntohs(3XNET)
- recv(3XNET)
- recvfrom(3XNET)
- recvmsg(3XNET)
- send(3XNET)
- sendmsg(3XNET)
- sendto(3XNET)
- sethostent(3XNET)
- setnetent(3XNET)
- setprotoent(3XNET)
- setservent(3XNET)
- setsockopt(3XNET)
- shutdown(3XNET)
- socket(3XNET)
- socketpair(3XNET)

相应的 3N 手册页介绍了传统的 BSD 套接字行为。此外，3N 节中还添加了以下新接口：

- freeaddrinfo(3SOCKET)
- freehostent(3SOCKET)
- getaddrinfo(3SOCKET)
- getipnodebyaddr(3SOCKET)
- getipnodebyname(3SOCKET)
- getnameinfo(3SOCKET)
- inet_ntop(3SOCKET)
- inet_pton(3SOCKET)

有关生成使用 XNS 5 (UNIX03) 套接字接口的应用程序的信息，请参见 standards(5) 手册页。

套接字基础知识

本节介绍基本套接字接口的用法。

创建套接字

socket(3SOCKET) 调用创建指定系列和指定类型的套接字。

```
s = socket(family, type, protocol);
```

如果未指定协议，则系统将选择支持所需套接字类型的协议。将返回套接字句柄。套接字句柄即为文件描述符。

family 由 sys/socket.h 中定义的一个常量指定。名为 AF_*suite* 的常量指定要在解释名称中使用的地址格式：

AF_APPLETALK	Apple Computer Inc. Appletalk 网络
AF_INET6	适用于 IPv6 和 IPv4 的 Internet 系列
AF_INET	仅适用于 IPv4 的 Internet 系列
AF_PUP	Xerox Corporation PUP internet
AF_UNIX	UNIX 文件系统

套接字类型在 sys/socket.h 中定义。AF_INET6、 AF_INET 和 AF_UNIX 支持 SOCK_STREAM、SOCK_DGRAM 或 SOCK_RAW 这些类型。以下示例创建 Internet 系列的流套接字：

```
s = socket(AF_INET6, SOCK_STREAM, 0);
```

此调用生成流套接字。TCP 协议提供底层通信。在大多数情况下，将 *protocol* 参数设置为缺省值 0。可以指定缺省协议之外的其他协议，如第 149 页中的 “高级套接字主题”中所述。

绑定本地名称

创建套接字时不指定名称。只有在套接字绑定到地址之后，远程进程才能引用此套接字。用于通信的进程通过地址连接。在 Internet 系列中，连接由本地和远程地址以及本地和远程端口组成。不能存在重复排序组，如 protocol、local address、 local port、 foreign address 和 foreign port。在大多数系列中，连接必须唯一。

使用 bind(3SOCKET) 接口，进程可以指定套接字的本地地址。此接口组成 local address 和 local port 组。connect(3SOCKET) 和 accept(3SOCKET) 通过添加地址元组的远程部分来完成套接字的关联。bind(3SOCKET) 调用的用法如下：

```
bind (s, name, namelen);
```

套接字句柄为 *s*。绑定名称是由支持协议解释的字节字符串。Internet 系列名称包含 Internet 地址和端口号。

本示例说明如何绑定 Internet 地址。

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in6 sin6;
...
s = socket(AF_INET6, SOCK_STREAM, 0);
bzero (&sin6, sizeof (sin6));
```

```
sin6.sin6_family = AF_INET6;
sin6.sin6_addr.s6_addr = in6addr_arg;
sin6.sin6_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin6, sizeof sin6);
```

地址 sin6 的内容在讨论 Internet 地址绑定的第 153 页中的 “地址绑定”中介绍。

建立连接

通常以非对称形式建立连接，一个进程用作客户机，而另一个进程则用作服务器。服务器将套接字绑定到与服务关联的已知地址，并阻塞在套接字上等待连接请求。然后，不相关的进程便可连接到此服务器。客户机通过启动到服务器套接字的连接，向服务器请求服务。在客户机端，connect(3SOCKET) 调用启动连接。在 Internet 系列中，此连接可能如下所示：

```
struct sockaddr_in6 server;
...
connect(s, (struct sockaddr *)&server, sizeof server);
```

如果在连接调用期间未绑定客户机的套接字，则系统会自动选择一个名称并将其绑定到套接字。有关更多信息，请参见第 153 页中的 “地址绑定”。这种自动选择是将本地地址绑定到客户机端套接字的常规方法。

要接收客户机的连接，服务器必须在绑定其套接字之后执行两个步骤。第一步是说明可以排队多少连接请求。第二步接受连接。

```
struct sockaddr_in6 from;
...
listen(s, 5);                /* Allow queue of 5 connections */
fromlen = sizeof(from);
newsock = accept(s, (struct sockaddr *) &from, &fromlen);
```

套接字句柄 *s* 是绑定到连接请求要发送到的地址的套接字。listen(3SOCKET) 的第二个参数指定可以对其进行排队的最大未完成连接数。from 结构使用客户机的地址进行填充。可能传递 NULL 指针。*fromlen* 为此结构的长度。

通常，accept(3SOCKET) 例程阻塞进程。accept(3SOCKET) 返回一个连接到请求客户机的新套接字描述符。*fromlen* 的值将更改为此地址的实际大小。

服务器无法指示其只接受来自特定地址的连接。服务器可以检查由 accept(3SOCKET) 返回的 from 地址并关闭与不可接受的客户机之间的连接。服务器可以接受多个套接字上的连接，或者避免在调用 accept(3SOCKET) 时阻塞。第 149 页中的 “高级套接字主题”中介绍了这些技术。

连接错误

如果连接失败，则会返回错误，但是由系统绑定的地址保持不变。如果连接成功，则套接字与服务器关联，并且可以开始数据传输。

下表列出在连接尝试失败时返回的一些比较常见的错误。

表 8–1 套接字连接错误

套接字错误	错误说明
ENOBUFS	支持调用的可用内存不足。
EPROTONOSUPPORT	请求未知协议。
EPROTOTYPE	请求不支持的套接字类型。
ETIMEDOUT	未在指定时间内建立连接。当目标主机关闭或由于丢失传输而导致网络问题时，会发生此错误。
ECONNREFUSED	主机拒绝服务。当服务器进程未在请求地址中显示时，会发生此错误。
ENETDOWN 或 EHOSTDOWN	这些错误是由底层通信接口传送的状态信息造成的。
ENETUNREACH 或 EHOSTUNREACH	由于不存在到网络或主机的路由，因此可能会发生这些操作错误。这些错误还可能是由中间网关或切换节点所返回的状态信息造成的。返回的状态信息并不总是足以区分网络故障和主机故障。

数据传输

本节介绍用于发送和接收数据的接口。可以使用常规 read(2) 和 write(2) 接口来发送或接收消息。

```
write(s, buf, sizeof buf);
read(s,  buf, sizeof buf);
```

还可以使用 send(3SOCKET) 和 recv(3SOCKET) :

```
send(s, buf, sizeof buf, flags);
recv(s, buf, sizeof buf, flags);
```

send(3SOCKET) 和 recv(3SOCKET) 非常类似于 read(2) 和 write(2)，但是 flags 参数至关重要。如果需要一个或多个以下项，则可以将 sys/socket.h 中定义的 flags 参数指定为非零值 :

MSG_OOB	发送和接收带外数据
MSG_PEEK	查看数据而不读取

MSG_DONTROUTE	发送数据而不路由包

带外数据特定于流套接字。使用 recv(3SOCKET) 调用指定 MSG_PEEK 之后，所有显示的数据均返回给用户，但是仍视为不可读取。套接字上的下一个 read(2) 或 recv(3SOCKET) 调用将返回相同数据。当前只有路由表管理进程使用发送数据而不路由包的选项（应用于传出包）。

关闭套接字

可以通过 close(2) 接口调用废弃 SOCK_STREAM 套接字。如果在 close(2) 接口调用之后数据排队到保证可靠传送的套接字，则协议会继续尝试传输数据。如果数据在任意时间之后还不能传送，则会将其废弃。

shutdown(3SOCKET) 可正常关闭 SOCK_STREAM 套接字。这两个进程均可确认不再发送。此调用的形式为：

```
shutdown(s, how);
```

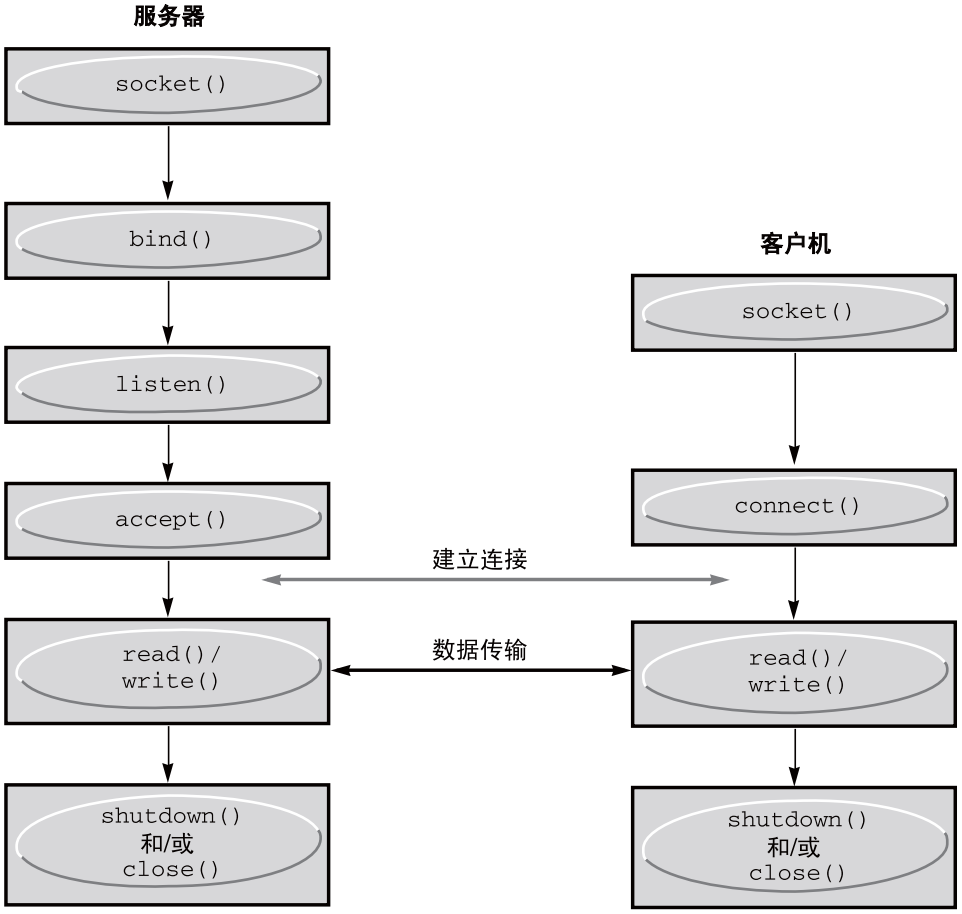
其中 how 定义为

0	禁止进一步接收数据
1	禁止进一步传输数据
2	禁止进一步传输和接收

连接流套接字

以下两个示例说明如何启动和接受 Internet 系列流连接。

图 8–1 使用流套接字的面向连接的通信



以下是针对服务器的示例程序。服务器创建套接字并将名称绑定到此套接字，然后显示端口号。此程序调用 listen(3SOCKET) 将套接字标记为可以接受连接请求并初始化请求队列。此程序的其余部分为一个死循环。每次循环都接受一个新的连接并将其从队列中删除，从而创建一个新的套接字。服务器读取并显示此套接字中的消息，然后关闭此套接字。第 153 页中的 “地址绑定”中介绍了 in6addr_any 的用法。

示例 8–1 接受 Internet 流连接（服务器）

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1
/*
```

示例 8–1 接受 Internet 流连接（服务器） （续）

```
 * This program creates a socket and then begins an infinite loop.
 * Each time through the loop it accepts a connection and prints
 * data from it. When the connection breaks, or the client closes
 * the connection, the program accepts a new connection.
*/
main() {
    int sock, length;
    struct sockaddr_in6 server;
    int msgsock;
    char buf[1024];
    int rval;
    /* Create socket. */
    sock = socket(AF_INET6, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("opening stream socket");
        exit(1);
    }
    /* Bind socket using wildcards.*/
    bzero (&server, sizeof(server));
    server.sin6_family = AF_INET6;
    server.sin6_addr = in6addr_any;
    server.sin6_port = 0;
    if (bind(sock, (struct sockaddr *) &server, sizeof server)
            == -1) {
        perror("binding stream socket");
        exit(1);
    }
    /* Find out assigned port number and print it out. */
    length = sizeof server;
    if (getsockname(sock,(struct sockaddr *) &server, &length)
            == -1) {
        perror("getting socket name");
        exit(1);
    }
    printf("Socket port #%d\n", ntohs(server.sin6_port));
    /* Start accepting connections. */
    listen(sock, 5);
    do {
        msgsock = accept(sock,(struct sockaddr *) 0,(int *) 0);
        if (msgsock == -1)
            perror("accept");
        else do {
            memset(buf, 0, sizeof buf);
            if ((rval = read(msgsock,buf, sizeof(buf))) == -1)
                perror("reading stream message");
            if (rval == 0)
                printf("Ending connection\n");
            else
                /* assumes the data is printable */
                printf("-->%s\n", buf);
        } while (rval > 0);
        close(msgsock);
    } while(TRUE);
    /*
     * Since this program has an infinite loop, the socket "sock" is
```

示例 8–1 接受 Internet 流连接（服务器） （续）

```
     * never explicitly closed. However, all sockets are closed
     * automatically when a process is killed or terminates normally.
     */
    exit(0);
}
```

要启动连接，示例 8–2 中的客户机程序会创建一个流套接字，然后调用 connect(3SOCKET)，指定用于连接的套接字地址。如果存在目标套接字，并且接受了请求，则连接会完成。现在，程序可以发送数据。数据按顺序传送，并且没有消息边界。此连接会在任何一个套接字关闭时销毁。有关此程序中的数据表示例程（如 ntohl(3SOCKET)、ntohs(3SOCKET)、htons(3SOCKET) 和 htonl(3XNET)）的更多信息，请参见 byteorder(3SOCKET) 手册页。

示例 8–2 Internet 系列流连接（客户机）

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define DATA "Half a league, half a league . . ."
/*
 * This program creates a socket and initiates a connection with
 * the socket given in the command line. Some data are sent over the
 * connection and then the socket is closed, ending the connection.
 * The form of the command line is: streamwrite hostname portnumber
 * Usage: pgm host port
 */
main(int argc, char *argv[])
{
    int sock, errnum, h_addr_index;
    struct sockaddr_in6 server;
    struct hostent *hp;
    char buf[1024];
    /* Create socket. */
    sock = socket( AF_INET6, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("opening stream socket");
        exit(1);
    }
    /* Connect socket using name specified by command line. */
    bzero (&server, sizeof (server));
    server.sin6_family = AF_INET6;
    hp = getipnodebyname(argv[1], AF_INET6, AI_DEFAULT, &errnum);
/*
 * getipnodebyname returns a structure including the network address
 * of the specified host.
 */
    if (hp == (struct hostent *) 0) {
        fprintf(stderr, "%s: unknown host\n", argv[1]);
        exit(2);
    }
```

示例 8–2 Internet 系列流连接（客户机） （续）

```
    h_addr_index = 0;
    while (hp->h_addr_list[h_addr_index] != NULL) {
        bcopy(hp->h_addr_list[h_addr_index], &server.sin6_addr,
                    hp->h_length);
        server.sin6_port = htons(atoi(argv[2]));
        if (connect(sock, (struct sockaddr *) &server,
                    sizeof (server)) == -1) {
            if (hp->h_addr_list[++h_addr_index] != NULL) {
                /* Try next address */
                continue;
            }
            perror("connecting stream socket");
            freehostent(hp);
            exit(1);
        }
        break;
    }
    freehostent(hp);
    if (write( sock, DATA, sizeof DATA) == -1)
        perror("writing on stream socket");
    close(sock);
    freehostent (hp);
    exit(0);
}
```

可以将一对一 SCTP 连接支持添加到流套接字。以下示例代码将 `-p` 添加到现有程序，使此程序可以指定要使用的协议。

示例 8–3 将 SCTP 支持添加到流套接字

```
#include <stdio.h>
#include <netdb.h>
#include <string.h>
#include <errno.h>

int
main(int argc, char *argv[])
{
    struct protoent *proto = NULL;
    int c;
    int s;
    int protocol;

    while ((c = getopt(argc, argv, "p:")) != -1) {
        switch (c) {
        case 'p':
            proto = getprotobyname(optarg);
            if (proto == NULL) {
                fprintf(stderr, "Unknown protocol: %s\n",
                        optarg);
                return (-1);
            }
            break;
```

示例 8–3 将 SCTP 支持添加到流套接字 （续）

```
        default:
            fprintf(stderr, "Unknown option: %c\n", c);
            return (-1);
        }
    }

    /* Use the default protocol, which is TCP, if not specified. */
    if (proto == NULL)
        protocol = 0;
    else
        protocol = proto->p_proto;

    /* Create a IPv6 SOCK_STREAM socket of the protocol. */
    if ((s = socket(AF_INET6, SOCK_STREAM, protocol)) == -1) {
        fprintf(stderr, "Cannot create SOCK_STREAM socket of type %s: "
                "%s\n", proto != NULL ? proto->p_name : "tcp",
                strerror(errno));
        return (-1);
    }
    printf("Success\n");
    return (0);
}
```

输入/输出多路复用

请求可以在多个套接字或多个文件之间多路复用。使用 `select(3C)` 进行多路复用：

```
#include <sys/time.h>
#include <sys/types.h>
#include <sys/select.h>
...
fd_set readmask, writemask, exceptmask;
struct timeval timeout;
...
select(nfds, &readmask, &writemask, &exceptmask, &timeout);
```

`select(3C)` 的第一个参数是列表中由接下来的三个参数指向的文件描述符数。

`select(3C)` 的第二、第三以及第四个参数指向三组文件描述符：一组描述符用于读取，一组用于写入，一组用于接受例外情况。带外数据是唯一的例外情况。可以将其中任一指针指定为已正确强制转换的空指针。每一组都是包含长整数位掩码数组的结构。可以使用 `select.h` 中定义的 `FD_SETSIZE` 来设置数组的大小。此数组的长度足以为每个 `FD_SETSIZE` 文件描述符存储一个位。

宏 `FD_SET` (*fd*, & *mask*) 和 `FD_CLR` (*fd*, & *mask*) 分别在组 `mask` 中添加和删除文件描述符 *fd*。此组应该在使用之前设置为零，并且宏 `FD_ZERO` (*&mask*) 将清除组 `mask`。

select(3C) 的第五个参数用于指定超时值。如果 timeout 指针为 NULL，则 select(3C) 将阻塞，直到可以选择描述符或收到信号为止。如果 timeout 中的字段均设置为 0，则 select(3C) 会进行轮询并立即返回。

select(3C) 例程通常返回所选的文件描述符数，或者在已超时的情况下返回零。对于错误或中断，select(3C) 例程返回 −1，并且 *errno* 中的错误号以及文件描述符掩码保持不变。对于成功的返回，三个组指示哪些文件描述符可以读取、写入或具有暂挂的例外情况。

使用 FD_ISSET (*fd*, &*mask*) 宏在选择掩码中测试文件描述符的状态。如果 *fd* 位于组 mask 中，则此宏会返回非零值。否则，此宏会返回零。针对读取组先后使用 select(3C) 和 FD_ISSET (*fd*, &*mask*) 宏，以便检查套接字上的排队连接请求。

以下示例显示如何针对可读性在侦听套接字上做出选择，以确定何时可以通过调用 accept(3SOCKET) 接受新的连接。程序将接受连接请求，读取数据，并在单个套接字上断开连接。

示例 8–4 使用 select(3C) 检查暂挂连接

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time/h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1
/*
 * This program uses select to check that someone is
 * trying to connect before calling accept.
 */
main() {
    int sock, length;
    struct sockaddr_in6 server;
    int msgsock;
    char buf[1024];
    int rval;
    fd_set ready;
    struct timeval to;
    /* Open a socket and bind it as in previous examples. */
    /* Start accepting connections. */
    listen(sock, 5);
    do {
        FD_ZERO(&ready);
        FD_SET(sock, &ready);
        to.tv_sec = 5;
        to.tv_usec = 0;
        if (select(sock + 1, &ready, (fd_set *)0,
                   (fd_set *)0, &to) == -1) {
            perror("select");
            continue;
        }
        if (FD_ISSET(sock, &ready)) {
            msgsock = accept(sock, (struct sockaddr *)0, (int *)0);
```

示例 8–4 使用 select(3C) 检查暂挂连接 （续）

```
            if (msgsock == -1)
                perror("accept");
            else do {
                memset(buf, 0, sizeof buf);
                if ((rval = read(msgsock, buf, sizeof(buf))) == -1)
                    perror("reading stream message");
                else if (rval == 0)
                    printf("Ending connection\n");
                else
                    printf("-->%s\n", buf);
            } while (rval > 0);
            close(msgsock);
        } else
            printf("Do something else\n");
        } while (TRUE);
    exit(0);
}
```

在早期版本的 select(3C) 例程中，其参数是指向整数的指针，而不是指向 fd_sets 的指针。在文件描述符数小于整数中的位数时，此风格的调用仍然有效。

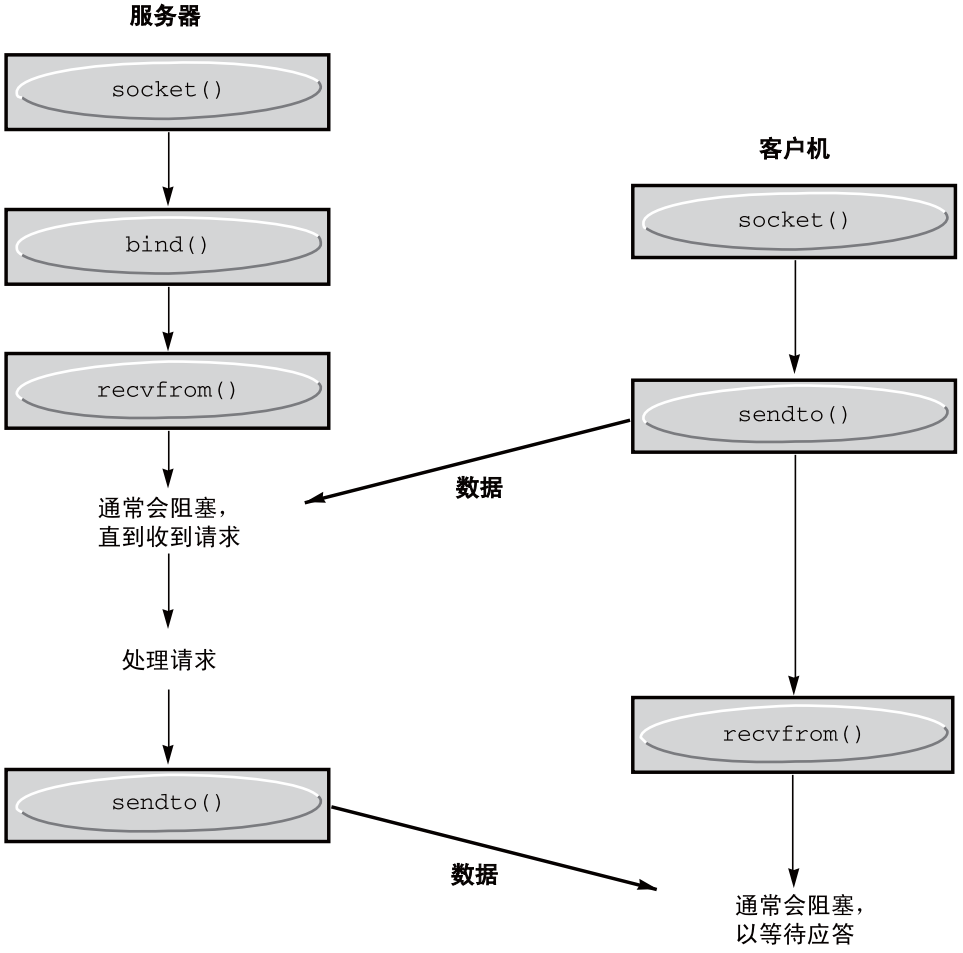
select(3C) 例程提供一种同步多路复用方案。第 149 页中的 “高级套接字主题”中介绍的 SIGIO 和 SIGURG 信号提供有关输出完成、输入可用性以及例外情况的异步通知。

数据报套接字

数据报套接字提供了对称数据交换接口，无需建立连接。每条消息都带有目标地址。下图显示了服务器与客户机之间的通信流。

服务器的 bind(3SOCKET) 步骤为可选步骤。

图 8–2 使用数据报套接字的无连接通信



按照第 126 页中的 “创建套接字”中所述创建数据报套接字。如果需要特定本地地址，必须在首次数据传输之前执行 bind(3SOCKET) 操作。否则，系统会在首次发送数据时设置本地地址或端口。使用 sendto(3SOCKET) 发送数据。

```
sendto(s, buf, buflen, flags, (struct sockaddr *) &to, tolen);
```

s、*buf*、*buflen* 和 *flags* 参数与面向连接的套接字中的相应参数相同。*to* 和 *tolen* 值指示消息预期接受者的地址。在本地检测到的错误情况（如无法访问网络）会导致返回 –1，并将 *errno* 设置为错误号。

```
recvfrom(s, buf, buflen, flags, (struct sockaddr *) &from, &fromlen);
```

要在数据报套接字上接收消息，请使用 recvfrom(3SOCKET)。在调用之前，*fromlen* 将会设置为 *from* 缓冲区的大小。返回时， fromlen 将会设置为接收数据报的地址的大小。

数据报套接字还可以使用 connect(3SOCKET) 调用将套接字与特定目标地址关联。然后，此套接字便可以使用 send(3SOCKET) 调用。如果在未显式指定目标地址的套接字上发送数据，则目标地址为已连接的对等方。只传送从该对等方接收的数据。一个套接字一次只能有一个已连接的地址。再次调用 connect(3SOCKET) 会更改目标地址。将立即返回数据报套接字上的连接请求。系统将记录对等方的地址。不能将 accept(3SOCKET) 或 listen(3SOCKET) 用于数据报套接字。

连接数据报套接字之后，便可以从先前的 send(3SOCKET) 调用中异步返回错误。此套接字可以在后续套接字操作中报告这些错误。或者，此套接字可以使用 getsockopt(3SOCKET) 的选项 SO_ERROR 来询问错误状态。

以下示例代码说明如何通过创建一个套接字、将名称绑定到该套接字以及将消息发送到该套接字来发送 Internet 调用。

示例 8–5 发送 Internet 系列数据报

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define DATA "The sea is calm, the tide is full . . ."
/*
 * Here I send a datagram to a receiver whose name I get from
 * the command line arguments. The form of the command line is:
 * dgramsend hostname portnumber
 */
main(int argc, char *argv[])
{
    int sock, errnum;
    struct sockaddr_in6 name;
    struct hostent *hp;
    /* Create socket on which to send. */
    sock = socket(AF_INET6,SOCK_DGRAM, 0);
    if (sock == -1) {
        perror("opening datagram socket");
        exit(1);
    }
    /*
     * Construct name, with no wildcards, of the socket to “send’’
     * to. getinodebyname returns a structure including the network
     * address of the specified host. The port number is taken from
     * the command line.
     */
    hp = getipnodebyname(argv[1], AF_INET6, AI_DEFAULT, &errnum);
    if (hp == (struct hostent *) 0) {
        fprintf(stderr, "%s: unknown host\n", argv[1]);
        exit(2);
    }
```

示例 8–5 发送 Internet 系列数据报 （续）

```
    bzero (&name, sizeof (name));
    memcpy((char *) &name.sin6_addr, (char *) hp->h_addr,
       hp->h_length);
    name.sin6_family = AF_INET6;
    name.sin6_port = htons(atoi(argv[2]));
    /* Send message. */
    if (sendto(sock,DATA, sizeof DATA ,0,
        (struct sockaddr *) &name,sizeof name) == -1)
        perror("sending datagram message");
    close(sock);
    exit(0);
}
```

以下样例代码说明如何通过创建套接字，将名称绑定到套接字，然后从套接字进行读取来读取 Internet 调用。

示例 8–6 读取 Internet 系列数据报

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
/*
 * This program creates a datagram socket, binds a name to it, then
 * reads from the socket.
 */
 main()
{
    int sock, length;
    struct sockaddr_in6 name;
    char buf[1024];
    /* Create socket from which to read. */
    sock = socket(AF_INET6, SOCK_DGRAM, 0);
    if (sock == -1) {
        perror("opening datagram socket");
        exit(1);
    }
    /* Create name with wildcards. */
    bzero (&name, sizeof (name));
    name.sin6_family = AF_INET6;
    name.sin6_addr = in6addr_any;
    name.sin6_port = 0;
    if (bind (sock, (struct sockaddr *)&name, sizeof (name)) == -1) {
        perror("binding datagram socket");
        exit(1);
    }
    /* Find assigned port value and print it out. */
    length = sizeof(name);
    if (getsockname(sock,(struct sockaddr *) &name, &length)
            == -1) {
        perror("getting socket name");
        exit(1);
    }
```

示例 8–6 读取 Internet 系列数据报 （续）

```
    printf("Socket port #%d\n", ntohs(name.sin6_port));
    /* Read from the socket. */
    if (read(sock, buf, 1024) == -1 )
        perror("receiving datagram packet");
    /* Assumes the data is printable */
    printf("-->%s\n", buf);
    close(sock);
    exit(0);
}
```

标准例程

本节介绍可以用来查找和构造网络地址的例程。除非另行说明，否则本节中介绍的接口只适用于 Internet 系列。

在客户机与服务器通信之前，在远程主机上查找服务需要许多级别的映射。为了便于用户使用，每个服务都具有一个名称。服务名和主机名必须转换为网络地址。最后，网络地址必须可用于查找并路由到主机。网络体系结构之间的具体映射情况可以不同。

标准例程将主机名映射到网络地址，将网络名映射到网络号，将协议名映射到协议号，将服务名映射到端口号。此外，标准例程还指明与服务器进程通信时所使用的相应协议。使用其中任一例程时，必须包括文件 `netdb.h`。

主机和服务名称

接口 `getaddrinfo(3SOCKET)`、`getnameinfo(3SOCKET)`、`gai_strerror(3SOCKET)` 和 `freeaddrinfo(3SOCKET)` 提供了一种在主机上的服务名称与地址之间进行转换的简化方法。这些接口比 `getipnodebyname(3SOCKET)`、`gethostbyname(3NSL)` 和 `getservbyname(3SOCKET)` API 更新。将透明地处理 IPv6 和 IPv4 地址。

`getaddrinfo(3SOCKET)` 例程返回指定主机名和服务名的组合地址和端口号。由于会动态分配 `getaddrinfo(3SOCKET)` 所返回的信息，因此必须通过 `freeaddrinfo(3SOCKET)` 释放信息以防止内存泄漏。`getnameinfo(3SOCKET)` 返回与指定地址和端口号关联的主机名和服务名。调用 `gai_strerror(3SOCKET)` 以便基于 `getaddrinfo(3SOCKET)` 和 `getnameinfo(3SOCKET)` 所返回的 `EAI_xxx` 代码显示错误消息。

以下是使用 `getaddrinfo(3SOCKET)` 的示例。

```
struct addrinfo        *res, *aip;
struct addrinfo        hints;
int                    error;
```

```
/* Get host address.  Any type of address will do. */
bzero(&hints, sizeof (hints));
hints.ai_flags = AI_ALL|AI_ADDRCONFIG;
hints.ai_socktype = SOCK_STREAM;

error = getaddrinfo(hostname, servicename, &hints, &res);
if (error != 0) {
    (void) fprintf(stderr, "getaddrinfo: %s for host %s service %s\n",
    gai_strerror(error), hostname, servicename);
    return (-1);
}
```

在 res 指向的结构中处理完 getaddrinfo(3SOCKET) 所返回的信息之后，应该通过 freeaddrinfo(res) 释放存储空间。

getnameinfo(3SOCKET) 例程在确定错误原因时特别有用，如以下示例所示：

```
struct sockaddr_storage faddr;
int                     sock, new_sock, sock_opt;
socklen_t               faddrlen;
int                     error;
char                    hname[NI_MAXHOST];
char                    sname[NI_MAXSERV];
...
faddrlen = sizeof (faddr);
new_sock = accept(sock, (struct sockaddr *)&faddr, &faddrlen);
if (new_sock == -1) {
    if (errno != EINTR && errno != ECONNABORTED) {
        perror("accept");
    }
    continue;
}
error = getnameinfo((struct sockaddr *)&faddr, faddrlen, hname,
    sizeof (hname), sname, sizeof (sname), 0);
if (error) {
    (void) fprintf(stderr, "getnameinfo: %s\n",
        gai_strerror(error));
} else {
    (void) printf("Connection from %s/%s\n", hname, sname);
}
```

主机名－hostent

Internet 主机名到地址映射由 hostent 结构表示，如 gethostent(3NSL) 中所定义：

```
struct hostent {
    char  *h_name;            /* official name of host */
    char  **h_aliases;        /* alias list */
    int   h_addrtype;         /* hostaddrtype(e.g.,AF_INET6) */
    int   h_length;           /* length of address */
    char  **h_addr_list;      /* list of addrs, null terminated */
};
```

```
/*1st addr, net byte order*/
#define h_addr h_addr_list[0]
```

getipnodebyname(3SOCKET)	将 Internet 主机名映射到 hostent 结构
getipnodebyaddr(3SOCKET)	将 Internet 主机地址映射到 hostent 结构
freehostent(3SOCKET)	释放 hostent 结构的内存
inet_ntop(3SOCKET)	将 Internet 主机地址映射到字符串

这些例程返回的 hostent 结构中包含主机名、主机别名、地址类型，以及以 NULL 结尾的长度可变地址的列表。此地址列表是必需的，因为主机可以具有许多地址。h_addr 定义用于向后兼容，并且是 hostent 结构的地址列表中的第一个地址。

网络名称－netent

用于将网络名映射到网络号以及将网络号映射到网络名的例程将返回 netent 结构：

```
/*
 * Assumes that a network number fits in 32 bits.
 */
struct netent {
   char     *n_name;      /* official name of net */
   char     **n_aliases;  /* alias list */
   int      n_addrtype;   /* net address type */
   int      n_net;        /* net number, host byte order */
};
```

getnetbyname(3SOCKET)、getnetbyaddr_r(3SOCKET) 和 getnetent(3SOCKET) 是前面介绍的主机例程的网络对应项。

协议名－protoent

protoent 结构定义用于 getprotobyname(3SOCKET)、getprotobynumber(3SOCKET) 和 getprotoent(3SOCKET) 且在 getprotoent(3SOCKET) 中定义的协议到名称映射：

```
struct protoent {
    char    *p_name;       /* official protocol name */
    char    **p_aliases    /* alias list */
    int     p_proto;       /* protocol number */
};
```

服务名－servent

Internet 系列服务驻留在特定的已知端口，并使用特定的协议。getprotoent(3SOCKET) 中定义的 servent 结构描述了服务名到端口号映射：

```
struct servent {
    char    *s_name;        /* official service name */
    char    **s_aliases;    /* alias list */
    int     s_port;         /* port number, network byte order */
    char    *s_proto;       /* protocol to use */
};
```

getservbyname(3SOCKET) 将服务名以及限定协议（可选）映射到 servent 结构。调用：

```
sp = getservbyname("telnet", (char *) 0);
```

将返回使用任意协议的 telnet 服务器的服务规范。调用：

```
sp = getservbyname("telnet", "tcp");
```

将返回使用 TCP 协议的 telnet 服务器。还提供了 getservbyport(3SOCKET) 和 getservent(3SOCKET)。getservbyport(3SOCKET) 具有的接口类似于 getservbyname(3SOCKET) 使用的接口。可以指定一个可选协议名来限定查找。

其他例程

可以使用其他一些例程来简化名称和地址的处理。下表概述了用于处理长度可变字节字符串以及字节交换网络地址和值的例程。

表 8–2 运行时库例程

接口	用法概要
memcmp(3C)	比较字节字符串；如果相同，则为 0，否则不为 0
memcpy(3C)	将 *n* 个字节从 *s2* 复制到 *s1*
memset(3C)	将 *n* 个字节设置为以 base 开始的 value
htonl(3SOCKET)	从主机字节顺序转换到网络字节顺序的 32 位值
htons(3SOCKET)	从主机字节顺序转换到网络字节顺序的 16 位值
ntohl(3SOCKET)	从网络字节顺序转换到主机字节顺序的 32 位值
ntohs(3SOCKET)	从网络字节顺序转换到主机字节顺序的 16 位值

因为操作系统希望以网络顺序提供地址，所以提供了字节交换例程。在某些体系结构中，主机字节顺序不同于网络字节顺序，因此有时程序必须对值进行字节交换。返回网络地址的例程以网络顺序执行此操作。仅在解释网络地址时会出现字节交换问题。例如，以下代码设置 TCP 或 UDP 端口的格式：

```
printf("port number %d\n", ntohs(sp->s_port));
```

在不需要这些例程的计算机上，将这些例程定义为空宏。

客户机/服务器程序

分布式应用程序的最常见形式为客户机/服务器模型。在此方案中，客户机进程从服务器进程请求服务。

备用方案是可以删除暂停服务器进程的服务服务器。Internet 服务守护进程 inetd(1M) 便是一个示例。inetd(1M) 可以侦听启动时通过读取配置文件而确定的不同端口。在 inetd(1M) 服务端口上请求连接时，inetd(1M) 会产生相应的服务器以便为客户机提供服务。客户机并不知道中间服务器已参与连接。第 156 页中的 “inetd 守护进程”中更详细地介绍了 inetd(1M)。

套接字和服务器

大多数服务器都是通过已知 Internet 端口号或 UNIX 系列名称进行访问的。服务 rlogin 便是已知的 UNIX 系列名称。示例 8–7 中给出了远程登录服务器的主循环。

服务器将从其调用者的控制终端分离出来，除非服务器在 DEBUG 模式下运行。

```
(void) close(0);
(void) close(1);
(void) close(2);
(void) open("/", O_RDONLY);
(void) dup2(0, 1);
(void) dup2(0, 2);
setsid();
```

进行分离可防止服务器从控制终端的进程组接收信号。服务器从控制终端分离之后，便不能将错误报告发送到终端。分离的服务器必须使用 syslog(3C) 记录错误。

服务器通过调用 getaddrinfo(3SOCKET) 获取其服务定义。

```
bzero(&hints, sizeof (hints));
hints.ai_flags = AI_ALL|AI_ADDRCONFIG;
hints.ai_socktype = SOCK_STREAM;
error = getaddrinfo(NULL, "rlogin", &hints, &api);
```

api 中返回的结果包含程序侦听服务请求所用的 Internet 端口。/usr/include/netinet/in.h 中定义了某些标准端口号。

然后，服务器创建套接字并侦听服务请求。bind(3SOCKET) 例程可以确保服务器在预期位置进行侦听。由于远程登录服务器会侦听受限的端口号，因此该服务器将以超级用户身份运行。服务器的主体是以下循环。

示例 8–7 服务器主循环

```
/* Wait for a connection request. */
for (;;) {
    faddrlen = sizeof (faddr);
    new_sock = accept(sock, (struct sockaddr *)api->ai_addr,
            api->ai_addrlen)
    if (new_sock == -1) {
        if (errno != EINTR && errno != ECONNABORTED) {
            perror("rlogind: accept");
        }
        continue;
    }
    if (fork() == 0) {
        close (sock);
        doit (new_sock, &faddr);
    }
    close (new_sock);
}
/*NOTREACHED*/
```

accept(3SOCKET) 将阻止消息，直到客户机请求服务为止。此外，如果 accept 被某信号（如 SIGCHLD）中断，则 accept(3SOCKET) 会返回故障指示。如果发生错误，则会检查来自 accept(3SOCKET) 的返回值，并使用 syslog(3C) 记录错误。

然后，服务器派生一个子进程，并调用远程登录协议处理的主体。父进程用于对连接请求进行排队的套接字将在子进程中关闭。accept(3SOCKET) 所创建的套接字将在父进程中关闭。将客户机的地址传递到服务器应用程序的 doit() 例程，此例程用来验证客户机。

套接字和客户机

本节介绍客户机进程所执行的步骤。与在服务器中相同，第一步是查找远程登录的服务定义。

```
bzero(&hints, sizeof (hints));
hints.ai_flags = AI_ALL|AI_ADDRCONFIG;
hints.ai_socktype = SOCK_STREAM;

error = getaddrinfo(hostname, servicename, &hints, &res);
if (error != 0) {
    (void) fprintf(stderr, "getaddrinfo: %s for host %s service %s\n",
        gai_strerror(error), hostname, servicename);
    return (-1);
}
```

getaddrinfo(3SOCKET) 返回 res 中的地址列表头。 通过创建套接字，并尝试连接列表中返回的每个地址直到一个地址有效，即可找到所需的地址。

```
for (aip = res; aip != NULL; aip = aip->ai_next) {
    /*
     * Open socket.  The address type depends on what
```

```
         * getaddrinfo() gave us.
         */
        sock = socket(aip->ai_family, aip->ai_socktype,
              aip->ai_protocol);
        if (sock == -1) {
            perror("socket");
            freeaddrinfo(res);
            return (-1);
        }

        /* Connect to the host. */
        if (connect(sock, aip->ai_addr, aip->ai_addrlen) == -1) {
            perror("connect");
            (void) close(sock);
            sock = -1;
            continue;
        }
        break;
}
```

套接字已经创建并已连接到所需的服务。connect(3SOCKET) 例程隐式绑定 sock，因为 sock 未绑定。

无连接服务器

某些服务使用数据报套接字。rwho(1) 服务提供了有关连接到局域网的主机的状态信息。应避免运行 in.rwhod(1M)，因为 in.rwho 会导致网络通信流量过大。rwho 服务将信息广播到所有连接到特定网络的主机。rwho 服务是数据报套接字用法示例。

运行 rwho(1) 服务器的主机上的用户可以使用 ruptime(1) 获取其他主机的当前状态。以下示例给出了典型输出。

示例 8–8 ruptime(1) 程序的输出

```
itchy up 9:45, 5 users, load 1.15, 1.39, 1.31
scratchy up 2+12:04, 8 users, load 4.67, 5.13, 4.59
click up 10:10, 0 users, load 0.27, 0.15, 0.14
clack up 2+06:28, 9 users, load 1.04, 1.20, 1.65
ezekiel up 25+09:48, 0 users, load 1.49, 1.43, 1.41
dandy 5+00:05, 0 users, load 1.51, 1.54, 1.56
peninsula down 0:24
wood down 17:04
carpediem down 16:09
chances up 2+15:57, 3 users, load 1.52, 1.81, 1.86
```

在每台主机上，rwho(1) 服务器进程定期广播状态信息，并接收状态信息。此外，服务器还更新数据库。将对此数据库进行解释以了解每台主机的状态。仅通过本地网络及其广播功能连接的服务器将自主运行。

使用广播时效率非常低，因为广播会生成过多网络通信流量。除非广泛且频繁地使用该服务，否则所带来的简单性相对定期广播的开销而言，得不偿失。

以下示例给出了简化的 rwho(1) 服务器版本。样例代码接收网络中其他主机广播的状态信息，并提供运行此样例代码的主机的状态。第一项任务在程序的主循环中完成：检查在 rwho(1) 端口接收到的包，以确保这些包由其他 rwho(1) 服务器进程发送并标记有到达时间。然后，这些包使用主机的状态更新文件。如果长时间没有收到来自主机的消息，则数据库例程认为此主机已关闭并记录此信息。由于主机正常运行时服务器可能会关闭，因此该应用程序容易出现错误。

示例 8–9 rwho(1) 服务器

```
main()
{
    ...
    sp = getservbyname("who", "udp");
    net = getnetbyname("localnet");
    sin.sin6_addr = inet_makeaddr(net->n_net, in6addr_any);
    sin.sin6_port = sp->s_port;
    ...
    s = socket(AF_INET6, SOCK_DGRAM, 0);
    ...
    on = 1;
    if (setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof on)
            == -1) {
        syslog(LOG_ERR, "setsockopt SO_BROADCAST: %m");
        exit(1);
    }
    bind(s, (struct sockaddr *) &sin, sizeof sin);
    ...
    signal(SIGALRM, onalrm);
    onalrm();
    while(1) {
        struct whod wd;
        int cc, whod, len = sizeof from;
        cc = recvfrom(s, (char *) &wd, sizeof(struct whod), 0,
            (struct sockaddr *) &from, &len);
        if (cc <= 0) {
            if (cc == -1 && errno != EINTR)
                syslog(LOG_ERR, "rwhod: recv: %m");
            continue;
        }
        if (from.sin6_port != sp->s_port) {
            syslog(LOG_ERR, "rwhod: %d: bad from port",
                ntohs(from.sin6_port));
            continue;
        }
        ...
        if (!verify( wd.wd_hostname)) {
            syslog(LOG_ERR, "rwhod: bad host name from %x",
                ntohl(from.sin6_addr.s6_addr));
            continue;
        }
        (void) sprintf(path, "%s/whod.%s", RWHODIR, wd.wd_hostname);
        whod = open(path, O_WRONLY|O_CREAT|O_TRUNC, 0666);
        ...
        (void) time(&wd.wd_recvtime);
        (void) write(whod, (char *) &wd, cc);
        (void) close(whod);
    }
```

示例 8–9 rwho(1) 服务器 （续）

```
    exit(0);
}
```

第二项服务器任务是提供其主机状态。这要求定期获取系统状态信息，将其打包在消息中，并在本地网络上广播，以使其他 rwho(1) 服务器收到这些信息。此任务由计时器运行，并由信号触发。

状态信息将在本地网络上广播。对于不支持广播的网络，使用多播。

高级套接字主题

对于大多数程序员而言，前面介绍的机制足以用来生成分布式应用程序。本节介绍其他功能。

带外数据

流套接字抽象概念包括带外数据。带外数据是一对连接的流套接字之间的逻辑上独立的传输通道。带外数据的传送独立于一般数据。带外数据功能必须支持每次至少可靠传送一条带外消息。此消息至少可以包含一个字节的数据。可以随时暂挂传送至少一条消息。

使用带内信号发送时，紧急数据将与一般数据一同按顺序传送，并从一般数据流中提取消息。提取的消息将单独存储。用户可以在顺序接收紧急数据与无序接收紧急数据之间进行选择，而不必缓冲中间数据。

使用 MSG_PEEK，可以查看带外数据。如果套接字具有进程组，则在通知协议其存在时会生成 SIGURG 信号。进程可以将进程组或进程 ID 设置为使用适当的 fcntl(2) 调用传送 SIGURG，如 SIGIO 的第 152 页中的 “中断驱动套接字 I/O”中所述。如果多个套接字都具有等待传送的带外数据，则用于例外情况的 select(3C) 调用可以确定哪些套接字具有此类数据暂挂。

逻辑标记位于数据流中发送带外数据的位置。远程登录和远程 shell 应用程序使用此功能在客户机进程与服务器进程之间传播信号。接收到信号之后，将废弃数据流中标记之前的所有数据。

要发送带外消息，请将 MSG_OOB 标志应用于 send(3SOCKET) 或 sendto(3SOCKET)。要接收带外数据，请将 MSG_OOB 指定给 recvfrom(3SOCKET) 或 recv(3SOCKET)。如果带外数据是内嵌数据，则不需要 MSG_OOB 标志。 SIOCATMARK ioctl(2) 指示读取指针当前是否指向数据流中的标记：

```
int yes;
ioctl(s, SIOCATMARK, &yes);
```

如果返回时 *yes* 为 1，则下次读取将返回标记之后的数据。否则，假设带外数据已经到达，下次读取将提供由客户机在发送带外信号之前发送的数据。以下示例给出了远程登录进程中接收中断或退出信号时刷新输出的例程。此代码读取标记之前的一般数据以废弃这些一般数据，然后读取带外字节。

进程还可以读取或查看带外数据，而无需首先读取标记之前的数据。当底层协议将紧急数据与一般数据一起带内传送，并仅提前发送其存在的通知时，很难访问此数据。此类型协议的示例为 TCP，此协议用于在 Internet 系列中提供套接字流。如果使用此类协议，则使用 MSG_OOB 标志调用 recv(3SOCKET) 时，带外字节可能尚未到达。在这种情况下，此调用会返回 EWOULDBLOCK 错误。此外，输入缓冲区中的带内数据量可能会导致正常流控制阻止对等方发送紧急数据，直到清除缓冲区为止。然后，在对等方可以发送紧急数据之前，此进程必须读取足够的排队数据以清除输入缓冲区。

示例 8–10 接收带外数据时刷新终端 I/O

```
#include <sys/ioctl.h>
#include <sys/file.h>
...
oob()
{
    int out = FWRITE;
    char waste[BUFSIZ];
    int mark = 0;

    /* flush local terminal output */
    ioctl(1, TIOCFLUSH, (char *) &out);
    while(1) {
        if (ioctl(rem, SIOCATMARK, &mark) == -1) {
            perror("ioctl");
            break;
        }
        if (mark)
            break;
        (void) read(rem, waste, sizeof waste);
    }
    if (recv(rem, &mark, 1, MSG_OOB) == -1) {
        perror("recv");
        ...
    }
    ...
}
```

用于保留套接字流中紧急内嵌数据位置的工具可用作套接字级别选项 SO_OOBINLINE。有关用法，请参见 getsockopt(3SOCKET)。使用此套接字级别选项，可以保留紧急数据的位置。但是，将返回一般数据流中标记之后的紧急数据，而不带 MSG_OOB 标志。接收多个紧急指示时将移动标记，但是不会丢失任何带外数据。

非阻塞套接字

某些应用程序需要不执行阻塞的套接字。例如，服务器可能返回错误代码，不执行无法立即完成的请求。此错误可能会导致进程暂停，等待完成。创建并连接套接字之后，发出 fcntl(2) 调用（如以下示例所示）使此套接字变为非阻塞套接字。

示例 8–11 设置非阻塞套接字

```
#include <fcntl.h>
#include <sys/file.h>
...
int fileflags;
int s;
...
s = socket(AF_INET6, SOCK_STREAM, 0);
...
if (fileflags = fcntl(s, F_GETFL, 0) == -1)
    perror("fcntl F_GETFL");
    exit(1);
}
if (fcntl(s, F_SETFL, fileflags | FNDELAY) == -1)
    perror("fcntl F_SETFL, FNDELAY");
    exit(1);
}
```

在非阻塞套接字上执行 I/O 时，请检查 errno.h 中是否有错误 EWOULDBLOCK，此错误通常在操作阻塞时发生。accept(3SOCKET)、connect(3SOCKET)、send(3SOCKET)、recv(3SOCKET)、read(2) 和 write(2) 都能返回 EWOULDBLOCK。如果某操作（如 send(3SOCKET)）不能全部完成，但是部分写入有效（如使用流套接字时），则会处理所有可用的数据。返回值是实际发送的数据量。

异步套接字 I/O

在同时处理多个请求的应用程序中，要求在进程之间进行异步通信。异步套接字的类型必须为 SOCK_STREAM。要使套接字异步，请发出 fcntl(2) 调用，如以下示例所示。

示例 8–12 使套接字异步

```
#include <fcntl.h>
#include <sys/file.h>
...
int fileflags;
int s;
...
s = socket(AF_INET6, SOCK_STREAM, 0);
...
if (fileflags = fcntl(s, F_GETFL ) == -1)
    perror("fcntl F_GETFL");
    exit(1);
```

示例 8–12 使套接字异步 （续）

```
}
if (fcntl(s, F_SETFL, fileflags | FNDELAY | FASYNC) == -1)
    perror("fcntl F_SETFL, FNDELAY | FASYNC");
    exit(1);
}
```

在初始化、连接套接字并使其变为非阻塞的异步套接字之后，通信类似于异步读写文件。可以使用 send(3SOCKET)、write(2)、recv(3SOCKET) 或 read(2) 来启动数据传输。信号驱动的 I/O 例程将完成数据传输，如下节中所述。

中断驱动套接字 I/O

SIGIO 信号会在套接字或任何文件描述符完成数据传输时通知进程。使用 SIGIO 的步骤如下所示：

1. 使用 signal(3C) 或 sigvec(3UCB) 调用设置 SIGIO 信号处理程序。
2. 使用 fcntl(2) 设置进程 ID 或进程组 ID，以将信号路由到其自己的进程 ID 或进程组 ID。套接字的缺省进程组为组 0 。
3. 将套接字转换为异步，如第 151 页中的 “异步套接字 I/O” 中所示。

使用以下样例代码，给定进程可以在发生套接字请求时接收有关暂挂请求的信息。添加 SIGURG 的处理程序之后，还可以使用此代码来准备接收 SIGURG 信号。

示例 8–13 异步 I/O 请求通知

```
#include <fcntl.h>
#include <sys/file.h>
 ...
signal(SIGIO, io_handler);
/* Set the process receiving SIGIO/SIGURG signals to us. */
if (fcntl(s, F_SETOWN, getpid()) < 0) {
    perror("fcntl F_SETOWN");
    exit(1);
}
```

信号和进程组 ID

对于 SIGURG 和 SIGIO，每个套接字都具有一个进程号和进程组 ID。这些值初始化为零，但是可以在稍后使用 F_SETOWN fcntl(2) 命令重新定义，如前面的示例中所示。fcntl(2) 采用正值的第三个参数可以设置套接字的进程 ID。fcntl(2) 采用负值的第三个参数可以设置套接字的进程组 ID。SIGURG 和 SIGIO 信号的唯一允许接收者是调用进程。类似的 fcntl(2)（即 F_GETOWN）将返回套接字的进程号。

还可以通过使用 ioctl(2) 将套接字分配给用户的进程组来接收 SIGURG 和 SIGIO。

```
/* oobdata is the out-of-band data handling routine */
sigset(SIGURG, oobdata);
int pid = -getpid();
if (ioctl(client, SIOCSPGRP, (char *) &pid) < 0) {
    perror("ioctl: SIOCSPGRP");
}
```

选择特定的协议

如果 socket(3SOCKET) 调用的第三个参数为 0，则socket(3SOCKET) 会选择缺省协议以用于所请求类型的返回套接字。缺省协议通常是正确的，而备用选项通常不可用。使用原始套接字与较低级别协议或较低级别硬件接口进行直接通信时，请使用协议参数设置解复用。

使用 Internet 系列中的原始套接字在 IP 上实现新协议，可以确保套接字只接收指定协议的包。要获取特定协议，请确定协议系列中定义的协议号。对于 Internet 系列，使用第 141 页中的 “标准例程”中介绍的库例程之一，如 getprotobyname(3SOCKET)。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
...
pp = getprotobyname("newtcp");
s = socket(AF_INET6, SOCK_STREAM, pp->p_proto);
```

借助基于流的连接，使用 getprotobyname 会形成套接字 s，但是协议类型为 newtcp，而不是缺省的 tcp。

地址绑定

对于寻址，TCP 和 UDP 使用 4 组件元组：

- 本地 IP 地址
- 本地端口号
- 外部 IP 地址
- 外部端口号

TCP 要求这些 4 组件元组具有唯一性。UDP 则不要求。用户程序并非总是了解用于本地地址和本地端口的正确值，因为主机可能驻留在多个网络中。用户不能直接访问已分配的端口号集。要避免这些问题，请不要指定地址部分，让系统在需要时适当分配这些部分。这些元组的不同部分可以由套接字 API 的不同部分指定：

bind(3SOCKET)	本地地址和/或本地端口
connect(3SOCKET)	外部地址和外部端口

调用 accept(3SOCKET) 可以从外部客户机检索连接信息。这样，即使 accept(3SOCKET) 的调用者并未进行任何指定，也会为系统指定本地地址和端口。将返回外部地址和外部端口。

调用 listen(3SOCKET) 可以导致选择本地端口。如果尚未完成显式 bind(3SOCKET) 以分配本地信息，则 listen(3SOCKET) 会分配暂时端口号。

驻留在特定端口的服务可以使用 bind(3SOCKET) 绑定到此端口。如果服务不需要本地地址信息，则此类服务可以不指定本地地址。将本地地址设置为 in6addr_any（<netinet/in.h> 中具有常量值的一个变量）。如果不需要固定本地端口，则调用 listen(3SOCKET) 可以选择端口。指定地址 in6addr_any 或端口号 0 的过程称为**设置通配符**。对于 AF_INET，使用 INADDR_ANY 替代 in6addr_any。

通配符地址简化了 Internet 系列中的本地地址绑定。以下样例代码将通过调用 getaddrinfo(3SOCKET) 所返回的特定端口号绑定到套接字，并且未指定本地地址：

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct addrinfo  *aip;
...
if (bind(sock, aip->ai_addr, aip->ai_addrlen) == -1) {
    perror("bind");
    (void) close(sock);
    return (-1);
}
```

主机上的每个网络接口通常都具有唯一的 IP 地址。带有通配符本地地址的套接字可以接收定向到指定端口号的消息。带有通配符本地地址的套接字还可以接收发送到分配给主机的任何可能地址的消息。要仅允许特定网络中的主机连接到服务器，服务器应绑定相应网络中接口的地址。

同样，也可以不指定本地端口号，此时系统将选择一个端口号。例如，要将特定本地地址绑定到套接字，但是不指定本地端口号，可以按如下方式使用 bind：

```
bzero (&sin, sizeof (sin));
(void) inet_pton (AF_INET6, "::ffff:127.0.0.1", sin.sin6_addr.s6_addr);
sin.sin6_family = AF_INET6;
sin.sin6_port = htons(0);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

系统使用两个条件来选择本地端口号：

- 小于 1024 (IPPORT_RESERVED) 的 Internet 端口号保留供特权用户使用。非特权用户可以使用大于 1024 的任何 Internet 端口号。最大的 Internet 端口号是 65535。
- 当前并未将此端口号绑定到其他套接字。

可以通过 accept(3SOCKET) 或 getpeername(3SOCKET) 查找客户机的端口号和 IP 地址。

在某些情况下，由于关联的创建过程分为两个步骤，因此系统用来选择端口号的算法不适用于应用程序。例如，Internet 文件传输协议会指定数据连接必须始终源于同一本地端口。但是，连接到不同的外部端口可以避免重复关联。在这种情况下，如果先前数据连接的套接字仍然存在，则系统会禁止将相同的本地地址和本地端口号绑定到套接字。

要覆盖缺省的端口选择算法，必须在绑定地址之前执行选项调用：

```
int on = 1;
...
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof on);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

通过此调用，可以绑定已处于使用状态的本地地址。此绑定并不违反唯一性要求。在连接时，系统仍会验证任何其他具有相同本地地址和本地端口的套接字不具有相同的外部地址和外部端口。如果关联已经存在，则会返回错误 `EADDRINUSE`。

套接字选项

可以通过 `setsockopt(3SOCKET)` 和 `getsockopt(3SOCKET)` 设置和获取多个套接字选项。例如，可以更改发送或接收缓冲区空间。这些调用的一般形式如下所示：

```
setsockopt(s, level, optname, optval, optlen);
```

和

```
getsockopt(s, level, optname, optval, optlen);
```

操作系统可以随时相应地调整这些值。

以下是 `setsockopt(3SOCKET)` 和 `getsockopt(3SOCKET)` 调用的参数：

s	要应用选项的套接字
level	指定协议级别，例如由 `sys/socket.h` 中的符号常量 `SOL_SOCKET` 指示的套接字级别
optname	在 `sys/socket.h` 中定义并指定选项的符号常量
optval	指向选项值
optlen	指向选项值的长度

对于 `getsockopt(3SOCKET)` 而言，*optlen* 是一个值-结果参数。此参数最初设置为 *optval* 所指向的存储区域的大小。返回时，此参数的值设置为已使用的存储的长度。

当某程序需要确定现有套接字的类型时，此程序应该使用 `SO_TYPE` 套接字选项和 `getsockopt(3SOCKET)` 调用来调用 `inetd(1M)`：

```
#include <sys/types.h>
#include <sys/socket.h>

int type, size;

size = sizeof (int);
if (getsockopt(s, SOL_SOCKET, SO_TYPE, (char *) &type, &size) < 0) {
    ...
}
```

在 getsockopt(3SOCKET) 之后，type 将设置为套接字类型的值，如 sys/socket.h 中定义的那样。对于数据报套接字，type 应该为 SOCK_DGRAM。

inetd 守护进程

inetd(1M) 守护进程在启动时调用，并从 /etc/inet/inetd.conf 文件中获取此守护进程侦听的服务。此守护进程针对 /etc/inet/inetd.conf 中列出的每个服务创建一个套接字，将相应的端口号绑定到每个套接字。有关详细信息，请参见 inetd(1M) 手册页。

inetd(1M) 守护进程轮询每个套接字，等待向对应于此套接字的服务发出连接请求。对于 SOCK_STREAM 类型套接字，inetd(1M) 在侦听套接字时接受 (accept(3SOCKET))，派生 (fork(2))，将新套接字复制 (dup(2)) 到文件描述符 0 和 1（ stdin 和 stdout），关闭其他开放式文件描述符，并执行 (exec(2)) 相应服务器。

使用 inetd(1M) 的主要优点就是未使用的服务不占用计算机资源。次要优点是 inetd(1M) 会尽力建立连接。在文件描述符 0 和 1 中，由 inetd(1M) 启动的服务器的套接字连接至其客户机。服务器可以立即进行读取、写入、发送或接收。只要服务器在适当的时候使用 fflush(3C)，便可以使用 stdio 约定所提供的缓冲 I/O。

getpeername(3SOCKET) 例程将返回连接到套接字的对等方（进程）的地址。此例程在由 inetd(1M) 启动的服务器中非常有用。例如，可以使用此例程记录诸如 fec0::56:a00:20ff:fe7d:3dd2 （通常用于表示客户机的 IPv6 地址）的 Internet 地址。inetd(1M) 服务器可以使用以下样例代码：

```
struct sockaddr_storage name;
int namelen = sizeof (name);
char abuf[INET6_ADDRSTRLEN];
struct in6_addr addr6;
struct in_addr addr;

if (getpeername(fd, (struct sockaddr *) &name, &namelen) == -1) {
    perror("getpeername");
    exit(1);
} else {
    addr = ((struct sockaddr_in *)&name)->sin_addr;
    addr6 = ((struct sockaddr_in6 *)&name)->sin6_addr;
    if (name.ss_family == AF_INET) {
            (void) inet_ntop(AF_INET, &addr, abuf, sizeof (abuf));
    } else if (name.ss_family == AF_INET6 &&
```

```
                IN6_IS_ADDR_V4MAPPED(&addr6)) {
            /* this is a IPv4-mapped IPv6 address */
            IN6_MAPPED_TO_IN(&addr6, &addr);
            (void) inet_ntop(AF_INET, &addr, abuf, sizeof (abuf));
    } else if (name.ss_family == AF_INET6) {
            (void) inet_ntop(AF_INET6, &addr6, abuf, sizeof (abuf));

    }
    syslog("Connection from %s\n", abuf);
}
```

广播及确定网络配置

IPv6 不支持广播，仅 IPv4 支持广播。

数据报套接字发送的消息可以广播到已连接网络中的所有主机。此网络必须支持广播，因为系统不在软件中提供任何广播模拟。广播消息会给网络带来很高的负载，因为广播消息会强制网络中的每台主机都为其服务。通常出于以下两个原因之一使用广播：

- 在未提供资源地址的情况下在本地网络中查找资源
- 某些功能要求将信息发送到所有可访问的相邻主机

要发送广播消息，请创建一个 Internet 数据报套接字：

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

将一个端口号绑定到此套接字：

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

通过将数据报发送到网络的广播地址，可以仅在一个网络中进行广播。通过将数据报发送到 `netinet/in.h` 中定义的特殊地址 `INADDR_BROADCAST`，可以在所有已连接的网络中进行广播。

系统会提供一种机制来确定多条有关系统上网络接口的信息。这些信息包括 IP 地址和广播地址。`SIOCGIFCONF` `ioctl(2)` 调用会以单一的 `ifconf` 结构返回主机的接口配置。此结构包含一个 `ifreq` 结构数组。主机连接的每个网络接口所支持的所有地址系列都具有自己的 `ifreq` 结构。

以下示例给出了 `net/if.h` 中定义的 `ifreq` 结构。

示例 8–14 net/if.h 头文件

```
struct ifreq {
    #define IFNAMSIZ 16
```

示例 8–14 net/if.h 头文件 （续）

```
    char ifr_name[IFNAMSIZ]; /* if name, e.g., "en0" */
    union {
        struct sockaddr ifru_addr;
        struct sockaddr ifru_dstaddr;
        char ifru_oname[IFNAMSIZ]; /* other if name */
        struct sockaddr ifru_broadaddr;
        short ifru_flags;
        int ifru_metric;
        char ifru_data[1]; /* interface dependent data */
        char ifru_enaddr[6];
    } ifr_ifru;
    #define ifr_addr ifr_ifru.ifru_addr
    #define ifr_dstaddr ifr_ifru.ifru_dstaddr
    #define ifr_oname ifr_ifru.ifru_oname
    #define ifr_broadaddr ifr_ifru.ifru_broadaddr
    #define ifr_flags ifr_ifru.ifru_flags
    #define ifr_metric ifr_ifru.ifru_metric
    #define ifr_data ifr_ifru.ifru_data
    #define ifr_enaddr ifr_ifru.ifru_enaddr
};
```

可以获取接口配置的调用为：

```
/*
 * Do SIOCGIFNUM ioctl to find the number of interfaces
 *
 * Allocate space for number of interfaces found
 *
 * Do SIOCGIFCONF with allocated buffer
 *
 */
if (ioctl(s, SIOCGIFNUM, (char *)&numifs) == -1) {
    numifs = MAXIFS;
}
bufsize = numifs * sizeof(struct ifreq);
reqbuf = (struct ifreq *)malloc(bufsize);
if (reqbuf == NULL) {
    fprintf(stderr, "out of memory\n");
    exit(1);
}
ifc.ifc_buf = (caddr_t)&reqbuf[0];
ifc.ifc_len = bufsize;
if (ioctl(s, SIOCGIFCONF, (char *)&ifc) == -1) {
    perror("ioctl(SIOCGIFCONF)");
    exit(1);
}
```

使用此调用之后，*buf* 将包含一个 `ifreq` 结构数组。主机所连接的每个网络都具有一个关联的 `ifreq` 结构。这些结构的排序顺序有两种：

- 按接口名称的字母顺序
- 按支持的地址系列的数值顺序

ifc.ifc_len 的值设置为 ifreq 结构所使用的字节数。

每个结构都有一组指示相应网络为运行或关闭、点对点或广播等等的接口标志。以下示例说明了 ioctl(2) 针对 ifreq 结构所指定的接口返回 SIOCGIFFLAGS 标志。

示例 8–15 获取接口标志

```
struct ifreq *ifr;
ifr = ifc.ifc_req;
for (n = ifc.ifc_len/sizeof (struct ifreq); --n >= 0; ifr++) {
    /*
     * Be careful not to use an interface devoted to an address
     * family other than those intended.
     */
    if (ifr->ifr_addr.sa_family != AF_INET)
        continue;
    if (ioctl(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
        ...
    }
    /* Skip boring cases */
    if ((ifr->ifr_flags & IFF_UP) == 0 ||
            (ifr->ifr_flags & IFF_LOOPBACK) ||
            (ifr->ifr_flags & (IFF_BROADCAST | IFF_POINTOPOINT)) == 0)
        continue;
}
```

以下示例使用 SIOGGIFBRDADDR ioctl(2) 命令来获取接口的广播地址。

示例 8–16 接口的广播地址

```
if (ioctl(s, SIOCGIFBRDADDR, (char *) ifr) < 0) {
    ...
}
memcpy((char *) &dst, (char *) &ifr->ifr_broadaddr,
    sizeof ifr->ifr_broadaddr);
```

还可以使用 SIOGGIFBRDADDR ioctl(2) 来获取点对点接口的目标地址。

获取接口广播地址之后，使用 sendto(3SOCKET) 传输广播数据报：

```
sendto(s, buf, buflen, 0, (struct sockaddr *)&dst, sizeof dst);
```

针对主机所连接的每个接口使用一次 sendto(3SOCKET)，前提是此接口支持广播或点对点寻址。

使用多播

只有类型为 SOCK_DGRAM 和 SOCK_RAW 的 AF_INET6 和 AF_INET 套接字支持 IP 多播。IP 多播仅在接口驱动程序支持多播的子网中受到支持。

发送 IPv4 多播数据报

要发送多播数据报，请在 224.0.0.0 到 239.255.255.255 的范围中指定一个 IP 多播地址作为 sendto(3SOCKET) 调用的目标地址。

缺省情况下，发送 IP 多播数据报时其生存时间 (time-to-live, TTL) 值为 1。此值可以阻止将数据报转发到单个子网之外。使用套接字选项 IP_MULTICAST_TTL，可以将后续多播数据报的 TTL 设置为 0 到 255 之间的任何值。此功能用于控制多播的范围。

```
u_char ttl;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, &ttl,sizeof(ttl))
```

TTL 为 0 的多播数据报不能在任何子网中传输，但是在发送主机属于目标组并且发送套接字上未禁用多播回送的情况下可以进行本地传送。如果有一个或多个多播路由器连接到第一跃点子网，则 TTL 大于 1 的多播数据报可以传送到多个子网。为了提供有意义的范围控制，多播路由器支持 TTL 阈值概念。这些阈值会阻止低于特定 TTL 值的数据报遍历某些子网。这些阈值将针对具有以下初始 TTL 值的多播数据报强制实施相应约定：

0	限定在同一主机
1	限定在同一子网
32	限定在同一站点
64	限定在同一地区
128	限定在同一洲
255	范围不受限制

站点和地区并未严格定义，站点可以根据实际情况再分为更小的管理单元。

应用程序可以选择以上列出的 TTL 值以外的初始 TTL 值。例如，应用程序可以通过发送多播查询来对网络资源执行扩展环搜索，即第一个 TTL 值为 0，然后逐渐增大 TTL 的值，直到收到回复为止。

多播路由器不转发任何目标地址在 224.0.0.0 与 224.0.0.255（包括 224.0.0.0 和 224.0.0.255）范围之间的多播数据报，而不管其 TTL 值是多少。此地址范围是为使用路由协议以及其他低级拓扑搜索或维护协议（如网关搜索和组成员关系报告）而保留的。

即使主机拥有多个具有多播功能的接口，每个多播传输也是通过单个网络接口发送的。如果主机还用作多播路由器且 TTL 值大于 1，则多播可以**转发**到源接口之外的接口。可以使用套接字选项覆盖来自给定套接字的后续传输的缺省设置：

```
struct in_addr addr;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_IF, &addr, sizeof(addr))
```

其中 `addr` 是所需传出接口的本地 IP 地址。通过指定地址 `INADDR_ANY` 恢复到缺省接口。使用 `SIOCGIFCONF ioctl` 获取接口的本地 IP 地址。要确定接口是否支持多播，请使用 `SIOCGIFFLAGS ioctl` 获取接口标志并测试是否设置了 `IFF_MULTICAST` 标志。此选项主要用于多播路由器以及其他专门针对 Internet 拓扑的系统服务。

如果将多播数据报发送到发送主机本身所属的组，则缺省情况下，本地传送的 IP 层将回送此数据报的副本。另一套接字选项可为发送主机提供针对是否回送后续数据报的显式控制：

```
u_char loop;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_LOOP, &loop, sizeof(loop))
```

其中 `loop` 为 0 即为禁用回送，为 1 即为启用回送。此选项通过消除因接收应用程序自己的传输内容而产生的开销，可提高单台主机上只有一个实例的应用程序的性能。对于可以在一台主机上具有多个实例或者其发送主机不属于目标组的应用程序，不应使用此选项。

如果发送主机属于其他接口的目标组，则发送初始 TTL 值大于 1 的多播数据报可以传送到其他接口上的发送主机。回送控制选项不会影响此类传送。

接收 IPv4 多播数据报

主机必须成为一个或多个 IP 多播组的成员，才能接收 IP 多播数据报。进程可以使用以下套接字选项请求主机加入多播组：

```
struct ip_mreq mreq;
setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq))
```

其中 `mreq` 为以下结构：

```
struct ip_mreq {
    struct in_addr imr_multiaddr;   /* multicast group to join */
    struct in_addr imr_interface;   /* interface to join on */
}
```

每个成员关系都与单个接口关联。可以在多个接口上加入同一组。将 `imr_interface` 地址指定为 `INADDR_ANY` 以选择缺省的多播接口。还可以通过指定主机的本地地址之一来选择特定的具有多播功能的接口。

要删除成员关系，请使用：

```
struct ip_mreq mreq;
setsockopt(sock, IPPROTO_IP, IP_DROP_MEMBERSHIP, &mreq, sizeof(mreq))
```

其中 mreq 包含用于添加成员关系的那些值。关闭套接字或中止保存套接字的进程将删除与此套接字关联的成员关系。可以有多个套接字请求成为特定组的成员，并且直到删除最后一个请求，主机才不再是此组的成员。

如果任一套接字请求成为数据报目标组的成员，则内核 IP 层将接受传入的多播包。给定套接字是否接收多播数据报取决于此套接字的关联目标端口和成员关系，或者取决于原始套接字的协议类型。要接收发送到特定端口的多播数据报，请将其绑定到本地端口，同时不指定本地地址，如使用 INADDR_ANY。

如果在 bind(3SOCKET) 之前存在以下内容，则可以将多个进程绑定到同一 SOCK_DGRAM UDP 端口：

```
int one = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one))
```

在这种情况下，每个目标为共享端口的传入多播或广播 UDP 数据报将传送到所有绑定到此端口的套接字。为了向后兼容，此传送不适用于传入的单播数据报。无论单播数据报的目标端口绑定有多少套接字，此类数据报永远都不会传送到多个套接字。SOCK_RAW 套接字不要求 SO_REUSEADDR 选项共享单一 IP 协议类型。

可以在 <netinet/in.h> 中找到与多播相关的新套接字选项所需的定义。所有 IP 地址均以网络字节顺序传递。

发送 IPv6 多播数据报

要发送 IPv6 多播数据报，请在 ff00::0/8 范围中指定一个 IP 多播地址作为 sendto(3SOCKET) 调用的目标地址。

缺省情况下，IP 多播数据报的发送跃点限制为 1，此值可以阻止将数据报转发到单个子网之外。使用套接字选项 IPV6_MULTICAST_HOPS，可以将后续多播数据报的跃点限制设置为 0 到 255 之间的任何值。此功能用于控制多播的范围：

```
uint_l;
setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_HOPS, &hops,sizeof(hops))
```

不能在任何子网中传输跃点限制为 0 的多播数据报，但是在以下情况下可以在本地范围内传送数据报：

- 发送主机属于目标组
- 启用了发送套接字上的多播回送

如果第一跃点子网连接到一个或多个多播路由器，则可以将跃点限制大于 1 的多播数据报传送到多个子网。与 IPv4 多播地址不同，IPv6 多播地址包含明确的范围信息，此信息在地址的第一部分进行编码。定义的范围如下，其中未指定 X：

`ffX1::0/16` 节点－本地范围，限定在同一节点

`ffX2::0/16` 链路－本地范围

`ffX5::0/16` 站点－本地范围

`ffX8::0/16` 组织－本地范围

`ffXe::0/16` 全局范围

应用程序可独立于多播地址范围，使用不同的跃点限制值。例如，应用程序可以通过发送多播查询来对网络资源执行扩展环搜索，即第一个跃点限制值为 0，然后逐渐增大跃点限制值，直到收到回复为止。

即使主机拥有多个具有多播功能的接口，每个多播传输也是通过单个网络接口发送的。如果主机还用作多播路由器且跃点限制值大于 1，则多播可以**转发**到源接口之外的接口。可以使用套接字选项覆盖来自给定套接字的后续传输的缺省设置：

```
uint_t ifindex;
ifindex = if_nametoindex ("hme3");
setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_IF, &ifindex, sizeof(ifindex))
```

其中，`ifindex` 是所需传出接口的接口索引。通过指定值 `0` 恢复到缺省接口。

如果将多播数据报发送到发送主机本身所属的组，则缺省情况下，本地传送的 IP 层将回送此数据报的副本。另一套接字选项可为发送主机提供针对是否回送后续数据报的显式控制：

```
uint_t loop;
setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_LOOP, &loop, sizeof(loop))
```

其中，`loop` 为 0 即为禁用回送，为 1 即为启用回送。此选项通过消除因接收应用程序自己的传输内容而产生的开销，可提高单台主机上只有一个实例的应用程序（如路由器或邮件守护进程）的性能。对于可以在一台主机上具有多个实例的应用程序（如会议程序）或者其发送主机不属于目标组的应用程序（如时间查询程序），不应使用此选项。

如果发送主机属于其他接口的目标组，则发送初始跃点限制值大于 1 的多播数据报可以传送到其他接口上的发送主机。回送控制选项不会影响此类传送。

接收 IPv6 多播数据报

主机必须成为一个或多个 IP 多播组的成员，才能接收 IP 多播数据报。进程可以使用以下套接字选项请求主机加入多播组：

```
struct ipv6_mreq mreq;
setsockopt(sock, IPPROTO_IPV6, IPV6_JOIN_GROUP, &mreq, sizeof(mreq))
```

其中 mreq 为以下结构：

```
struct ipv6_mreq {
    struct in6_addr    ipv6mr_multiaddr;    /* IPv6 multicast addr */
    unsigned int       ipv6mr_interface;    /* interface index */
}
```

每个成员关系都与单个接口关联。可以在多个接口上加入同一组。将 ipv6_interface 指定为 0 以选择缺省的多播接口。为主机的其中一个接口指定接口索引以选择此具有多播功能的接口。

要离开组，请使用：

```
struct ipv6_mreq mreq;
setsockopt(sock, IPPROTO_IPV6, IP_LEAVE_GROUP, &mreq, sizeof(mreq))
```

其中 mreq 包含用于添加成员关系的那些值。当关闭套接字或中止保存套接字的进程时，此套接字将删除关联的成员关系。可以有多个套接字请求成为特定组的成员。直到删除最后一个请求，主机才不再是此组的成员。

如果任一套接字已请求成为数据报目标组的成员，则内核 IP 层将接受传入的多播包。多播数据报是否传送到特定的套接字取决于与此套接字关联的目标端口和成员关系，或者取决于原始套接字的协议类型。要接收发送到特定端口的多播数据报，请将其绑定到本地端口，同时不指定本地地址，如使用 INADDR_ANY。

如果在 bind(3SOCKET) 之前存在以下内容，则可以将多个进程绑定到同一 SOCK_DGRAM UDP 端口：

```
int one = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one))
```

在这种情况下，所有绑定到此端口的套接字将接收每个目标为共享端口的传入多播 UDP 数据报。为了向后兼容，此传送**不**适用于传入的单播数据报。无论单播数据报的目标端口绑定有多少套接字，此类数据报永远都不会传送到多个套接字。 SOCK_RAW 套接字不要求 SO_REUSEADDR 选项共享单一 IP 协议类型。

可以在 <netinet/in.h> 中找到与多播相关的新套接字选项所需的定义。所有 IP 地址均以网络字节顺序传递。

流控制传输协议

流控制传输协议 (Stream Control Transmission Protocol, SCTP) 是一种可靠的传输协议，提供的服务与 TCP 提供的服务类似。此外，SCTP 还提供网络级别的容错功能。SCTP 支持关联两端具有多宿主。SCTP 套接字 API 支持模仿 TCP 的一对一的套接字风格。SCTP 套接字 API 还支持旨在用于发送信号的一对多套接字风格。一对多套接字风格可以减少在进程中使用的文件描述符数。必须链接 `libsctp` 库才能使用 SCTP 函数调用。

将在两个端点之间设置 SCTP 关联。这些端点使用的四次握手机制借助 Cookie 来防止遭到某些类型的拒绝服务 (denial-of-service, DoS) 攻击。端点可由多个 IP 地址表示。

SCTP 栈实现

本节列出了 IETF 流控制传输协议标准 (RFC 2960) 和流控制传输协议校验和更改标准 (RFC 3309) 的 Solaris 实现的详细信息。本节中的表列出了 RFC 2960 的所有例外情况。Solaris 操作系统中的 SCTP 协议完全实现了 RFC 3309，还实现了 RFC 2960 中未在此表中明确提及的所有部分。

表 8–3 RFC 2960 的 Solaris SCTP 实现例外情况

RFC 2960 部分	Solaris 实现中的例外情况
3. SCTP 包格式	3.2 块字段说明： Solaris SCTP 未实现可选 ECNE 和 CWR。
	3.3.2： Solaris SCTP 未实现启动 (INIT)、可选 ECN、主机名称地址和 Cookie 保留参数。
	3.3.3： Solaris SCTP 未实现启动确认、可选 ECN 和主机名称地址参数。
5. 关联初始化	5.1.2 处理地址参数的 (B) 部分： 未实现可选主机名称参数。
6. 用户数据传输	6.8 Adler-32 校验和计算： 本部分已经被 RFC 3309 废弃。
10. 上层接口	Solaris SCTP 实现了 IETF TSVWG SCTP 套接字 API 草案。

注 – TSVWG SCTP 套接字 API 的 Solaris 10 实现基于 Solaris 10 首次发布时发布的 API 草案版本。

SCTP 套接字接口

当 socket() 调用为 IPPROTO_SCTP 创建套接字时，它会调用特定于 SCTP 的套接字创建例程。针对 SCTP 套接字执行的套接字调用会自动调用相应的 SCTP 套接字例程。在一对一套接字中，每个套接字都对应一个 SCTP 关联。可以通过调用以下函数来创建一对一套接字：

```
socket(AF_INET[6], SOCK_STREAM, IPPROTO_STCP);
```

在一对多风格套接字中，每个套接字都处理多个 SCTP 关联。每个关联都具有一个名为 *sctp_assoc_t* 的关联标识符。可以通过调用以下函数来创建一对多套接字：

```
socket(AF_INET[6], SOCK_SEQPACKET, IPPROTO_STCP);
```

sctp_bindx()

```
int sctp_bindx(int sock, void *addrs, int addrcnt, int flags);
```

sctp_bindx() 函数管理 SCTP 套接字上的地址。如果 *sock* 参数为 IPv4 套接字，则传递给 sctp_bindx() 函数的地址必须为 IPv4 地址。如果 *sock* 参数为 IPv6 套接字，则传递给 sctp_bindx() 函数的地址可以为 IPv4 或 IPv6 地址。当传递给 sctp_bindx() 函数的地址为 INADDR_ANY 或 IN6ADDR_ANY 时，此套接字将绑定到所有可用地址。可以使用 bind(3SOCKET) 绑定 SCTP 端点。

**addrs* 参数的值是指向包含一个或多个套接字地址的数组的指针。每个地址都包含在其相应的结构中。如果地址为 IPv4 地址，则它们包含在 sockaddr_in 结构或 sockaddr_in6 结构中。如果地址为 IPv6 地址，则它们包含在 sockaddr_in6 结构中。可以通过地址类型系列区分地址长度。调用者使用 *addrcnt* 参数指定数组中的地址数。

如果成功，则 sctp_bindx() 函数将返回 0。如果失败，则 sctp_bindx() 函数将返回 -1，并将 errno 的值设置为相应的错误代码。

如果没有为每个套接字地址提供同一端口，则 sctp_bindx () 函数将失败，并将 errno 的值设置为 EINVAL。

通过对以下零个或多个当前定义的标志执行按位 OR 运算，即可形成 *flags* 参数：

- SCTP_BINDX_ADD_ADDR
- SCTP_BINDX_REM_ADDR

SCTP_BINDX_ADD_ADDR 指示 SCTP 将给定地址添加到关联中。SCTP_BINDX_REM_ADDR 指示 SCTP 从关联中删除给定地址。这两个标志相互排斥。如果同时提供这两个标志，则 sctp_bindx() 将失败，并将 errno 的值设置为 EINVAL。

调用者无法删除关联中的所有地址。sctp_bindx () 函数拒绝此类尝试的表现为：函数失败并将 errno 的值设置为 EINVAL。应用程序可以在调用 bind() 函数之后使用 sctp_bindx(SCTP_BINDX_ADD_ADDR)，将其他地址与端点关联。应用程序可以使用 sctp_bindx(SCTP_BINDX_REM_ADDR) 删除与侦听套接字关联的地址。使用

sctp_bindx(SCTP_BINDX_REM_ADDR) 删除地址之后，接受新关联将不会重新关联已删除的地址。如果端点支持动态地址，则可以使用 SCTP_BINDX_REM_ADDR 或 SCTP_BINDX_ADD_ADDR 向对等方发送消息来更改对等方的地址列表。在已连接的关联中添加和删除地址为可选功能。不支持此功能的实现将返回 EOPNOTSUPP。

如果地址系列不是 AF_INET 或 AF_INET6，则 sctp_bindx() 函数将失败并返回 EAFNOSUPPORT。如果在 *sock* 参数中传递给 sctp_bindx () 的文件描述符无效，则 sctp_bindx() 函数将失败并返回 EBADF。

sctp_opt_info()

int **sctp_opt_info**(int *sock*, sctp_assoc_id_t *id*, int *opt*, void **arg*, socklen_t **len*);

sctp_opt_info() 函数将返回与 *sock* 参数中所述的套接字关联的 SCTP 级别选项。如果此套接字为一对多风格 SCTP 套接字，则 *id* 参数的值是某个特定关联。对于一对一风格 SCTP 套接字，将忽略 *id* 参数。*opt* 参数的值指定要获取的 SCTP 套接字选项。*arg* 参数的值是为调用程序而分配的特定于选项的结构缓冲区。**len* 参数的值为选项长度。

opt 参数可以采用以下值：

SCTP_RTOINFO　返回用于初始化和绑定重新传输超时 (retransmission timeout, RTO) 可调参数的协议参数。这些协议参数使用以下结构：

```
struct sctp_rtoinfo {
    sctp_assoc_t    srto_assoc_id;
    uint32_t        srto_initial;
    uint32_t        srto_max;
    uint32_t        srto_min;
};
```

srto_assoc_id　调用程序提供此值，它指定所关注的关联。

srto_initial　此值为初始 RTO 值。

srto_max　此值为最大 RTO 值。

srto_min　此值为最小 RTO 值。

SCTP_ASSOCINFO　返回特定于关联的参数。这些参数使用以下结构：

```
struct sctp_assocparams {
    sctp_assoc_t    sasoc_assoc_id;
    uint16_t        sasoc_asocmaxrxt;
    uint16_t        sasoc_number_peer_destinations;
    uint32_t        sasoc_peer_rwnd;
    uint32_t        sasoc_local_rwnd;
    uint32_t        sasoc_cookie_life;
};
```

sasoc_assoc_id
　调用程序提供此值，它指定所关注的关联。

sasoc_assocmaxrxt
此值指定关联的最大重新传输计数。

sasoc_number_peer_destinations
此值指定对等方具有的地址数。

sasoc_peer_rwnd
此值指定对等方接收窗口的当前值。

sasoc_local_rwnd
此值指定对等方传输到的上一个已报告的接收窗口。

sasoc_cookie_life
此值指定关联 Cookie 的生命周期。可在发出 Cookie 时使用此值。

所有使用时间值的参数均以毫秒为单位。

SCTP_DEFAULT_SEND_PARAM
返回 sendto(3SOCKET) 函数调用在此关联中使用的缺省参数集。这些参数使用以下结构：

```
struct sctp_sndrcvinfo {
    uint16_t        sinfo_stream;
    uint16_t        sinfo_ssn;
    uint16_t        sinfo_flags;
    uint32_t        sinfo_ppid;
    uint32_t        sinfo_context;
    uint32_t        sinfo_timetolive;
    uint32_t        sinfo_tsn;
    uint32_t        sinfo_cumtsn;
    sctp_assoc_t    sinfo_assoc_id;
};
```

sinfo_stream
此值指定 sendmsg () 调用的缺省流。

sinfo_ssn
此值始终为 0。

sinfo_flags
此值包含 sendmsg() 调用的缺省标志。此标志可以采用以下值：

- MSG_UNORDERED
- MSG_ADDR_OVER
- MSG_ABORT
- MSG_EOF
- MSG_PR_SCTP

sinfo_ppid
此值为 sendmsg() 调用的缺省有效负荷协议标识符。

sinfo_context
此值为 sendmsg() 调用的缺省上下文。

sinfo_timetolive
此值指定时间段（毫秒）。在此时间段过后，如果消息传输尚未开始，则消息将过期。值为 0 指示消息尚未过

期。如果设置了 MSG_PR_SCTP 标志，当消息传输未在 sinfo_timetolive 所指定的时间段内成功完成时，消息将过期。

sinfo_tsn　此值始终为 0。

sinfo_cumtsn　此值始终为 0。

sinfo_assoc_id　此值由调用程序填充。它指定所关注的关联。

SCTP_PEER_ADDR_PARAMS　返回所指定对等地址的参数。这些参数使用以下结构：

```
struct sctp_paddrparams {
    sctp_assoc_t               spp_assoc_id;
    struct sockaddr_storage    spp_address;
    uint32_t                   spp_hbinterval;
    uint16_t                   spp_pathmaxrxt;
};
```

spp_assoc_id
调用程序提供此值，它指定所关注的关联。

spp_address
此值指定所关注的对等方地址。

spp_hbinterval
此值指定心跳间隔（毫秒）。

spp_pathmaxrxt
此值指定在认为地址不可访问之前针对此地址尝试的最大重新传输次数。

SCTP_STATUS　返回有关关联的当前状态信息。这些参数使用以下结构：

```
struct sctp_status {
    sctp_assoc_t            sstat_assoc_id;
    int32_t                 sstat_state;
    uint32_t                sstat_rwnd;
    uint16_t                sstat_unackdata;
    uint16_t                sstat_penddata;
    uint16_t                sstat_instrms;
    uint16_t                sstat_outstrms;
    uint32_t                sstat_fragmentation_point;
    struct sctp_paddrinfo   sstat_primary;
};
```

sstat_assoc_id
调用程序提供此值，它指定所关注的关联。

sstat_state
此值为关联的当前状态。关联可以采用以下状态：

SCTP_IDLE　SCTP 端点没有任何与其关联的关联。一旦

	socket() 函数调用打开一个端点或端点关闭，端点便会处于此状态。
SCTP_BOUND	SCTP 端点在调用 bind() 之后绑定到一个或多个本地地址。
SCTP_LISTEN	此端点在等待来自任何远程 SCTP 端点的关联请求。
SCTP_COOKIE_WAIT	此 SCTP 端点已发送 INIT 块并在等待 INIT-ACK 块。
SCTP_COOKIE_ECHOED	此 SCTP 端点已将从其对等方的 INIT-ACK 块接收的 Cookie 回显到对等方。
SCTP_ESTABLISHED	此 SCTP 端点可以与其对等方交换数据。
SCTP_SHUTDOWN_PENDING	此 SCTP 端点已从其上层接收了 SHUTDOWN 原语。此端点不再从其上层接受数据。
SCTP_SHUTDOWN_SEND	处于 SCTP_SHUTDOWN_PENDING 状态的 SCTP 端点已向其对等方发送了 SHUTDOWN 块。仅在确认所有从此端点到其对等方的未完成数据之后，才发送 SHUTDOWN 块。当此端点的对等方发送 SHUTDOWN ACK 块时，此端点会发送 SHUTDOWN COMPLETE 块并认为关联已关闭。
SCTP_SHUTDOWN_RECEIVED	SCTP 端点已从其对等方接收了 SHUTDOWN 块。此端点不再从其用户接受新数据。

SCTP_SHUTDOWN_ACK_SEND　处于 SCTP_SHUTDOWN_RECEIVED 状态的 SCTP 端点已向其对等方发送了 SHUTDOWN ACK 块。此端点仅在其对等方确认来自此端点的所有未完成数据之后才发送 SHUTDOWN ACK 块。当此端点的对等方发送 SHUTDOWN COMPLETE 块时，将关闭关联。

sstat_rwnd
此值为关联对等方的当前接收窗口。

sstat_unackdata
此值为未确认的 DATA 块数。

sstat_penddata
此值为等待接收的 DATA 块数。

sstat_instrms
此值为传入的流数。

sstat_outstrms
此值为外发的流数。

sstat_fragmentation_point
如果消息、SCTP 头和 IP 头的组合大小超出 sstat_fragmentation_point 的值，则消息会分段。此值等于包目标地址的路径最大传输单元 (Path Maximum Transmission Unit, P-MTU)。

sstat_primary
此值包含有关主要对等地址的信息。此信息使用以下结构：

```
struct sctp_paddrinfo {
    sctp_assoc_t               spinfo_assoc_id;
    struct sockaddr_storage    spinfo_address;
    int32_t                    spinfo_state;
    uint32_t                   spinfo_cwnd;
    uint32_t                   spinfo_srtt;
    uint32_t                   spinfo_rto;
    uint32_t                   spinfo_mtu;
};
```

spinfo_assoc_id　调用程序提供此值，它指定所关注的关联。

spinfo_address	此值为主要对等方的地址。
spinfo_state	此值可以采用 SCTP_ACTIVE 或 SCTP_INACTIVE 两个值中的任意一个。
spinfo_cwnd	此值为对等地址的拥塞窗口。
spinfo_srtt	此值为对等地址的当前平滑往返时间计算结果，以毫秒表示。
spinfo_rto	此值为对等地址的当前重新传输超时值，以毫秒表示。
spinfo_mtu	此值为对等地址的 P-MTU。

如果成功，则 sctp_opt_info() 函数将返回 0。如果失败，则 sctp_opt_info() 函数将返回 -1，并将 errno 的值设置为相应的错误代码。如果在 *sock* 参数中传递给 sctp_opt_info () 的文件描述符无效，则 sctp_opt_info() 函数将失败并返回 EBADF。如果在 *sock* 参数中传递给 sctp_opt_info() 函数的文件描述符没有描述套接字，则 sctp_opt_info() 函数将失败并返回 ENOTSOCK。如果关联 ID 对于一对多风格 SCTP 套接字而言无效，则 sctp_opt_info() 函数将失败，并将 errno 的值设置为 EINVAL。如果输入缓冲区长度对于指定的选项而言过短，则 sctp_opt_info() 函数将失败，并将 errno 的值设置为 EINVAL。如果对等方地址的地址系列不是 AF_INET 或 AF_INET6，则 sctp_opt_info() 函数将失败，并将 errno 的值设置为 EAFNOSUPPORT。

sctp_recvmsg()

ssize_t **sctp_recvmsg**(int *s*, void **msg*, size_t *len*, struct sockaddr **from*, socklen_t **fromlen*, struct sctp_sndrcvinfo **sinfo* , int **msg_flags*);

使用 sctp_recvmsg() 函数，可以从 *s* 参数所指定的 SCTP 端点接收消息。调用程序可以指定以下属性：

msg
　此参数为消息缓冲区的地址。

len
　此参数为消息缓冲区的长度。

from
　此参数为指向包含发送主机地址的地址的指针。

fromlen
　此参数为与 *from* 参数中的地址关联的缓冲区的大小。

sinfo
　此参数仅在调用程序启用了 sctp_data_io_events 时有效。要启用 sctp_data_io_events，请使用套接字选项 SCTP_EVENTS 调用 setsockopt() 函数。如

果启用了 sctp_data_io_events，则应用程序将接收每个传入消息的 sctp_sndrcvinfo 结构的内容。此参数为指向 sctp_sndrcvinfo 结构的指针。此结构将在接收消息时进行填充。

msg_flags
此参数包含所有存在的消息标志。

sctp_recvmsg() 函数将返回其接收的字节数。sctp_recvmsg() 函数将在出现错误时返回 -1。

如果在 *s* 参数中传递的文件描述符无效，则 sctp_recvmsg() 函数将失败，并将 errno 的值设置为 EBADF。如果在 *s* 参数中传递的文件描述符没有描述套接字，则 sctp_recvmsg() 函数将失败，并将 errno 的值设置为 ENOTSOCK。如果 *msg_flags* 参数包括值 MSG_OOB，则 sctp_recvmsg() 函数将失败，并将 errno 的值设置为 EOPNOTSUPP。如果没有建立关联，则 sctp_recvmsg() 函数将失败，并将 errno 的值设置为 ENOTCONN。

sctp_sendmsg()

ssize_t **sctp_sendmsg**(int *s*, const void **msg*, size_t *len*, const struct sockaddr **to*, socklen_t *tolen*, uint32_t *ppid*, uint32_t *flags*, uint16_t *stream_no*, uint32_t *timetolive*, uint32_t *context*);

sctp_sendmsg() 函数在发送来自 SCTP 端点的消息时启用高级 SCTP 功能。

s 此值指定发送消息的 SCTP 端点。

msg 此值包含 sctp_sendmsg () 函数所发送的消息。

len 此值为消息的长度，以字节为单位。

to 此值为消息的目标地址。

tolen 此值为目标地址的长度。

ppid 此值为应用程序指定的有效负荷协议标识符。

stream_no 此值为此消息的目标流。

timetolive 此值为消息未能成功发送到对等方的情况下消息过期之前可以等待的时间段，以毫秒表示。

context 如果在发送消息时出现错误，则返回此值。

flags 此值在将逻辑运算 OR 以按位形式应用于以下零个或多个标志位时形成：

MSG_UNORDERED
设置此标志之后，sctp_sendmsg() 函数将无序传送消息。

MSG_ADDR_OVER
设置此标志之后，sctp_sendmsg() 函数将使用 *to* 参数中的地址，而不使用关联的主要目标地址。此标志仅用于一对多风格 SCTP 套接字。

MSG_ABORT
设置此标志之后，将通过向对等方发送 ABORT 信号来中止指定的关联。此标志仅用于一对多风格 SCTP 套接字。

MSG_EOF
设置此标志之后，指定的关联将进入正常关闭状态。此标志仅用于一对多风格 SCTP 套接字。

MSG_PR_SCTP
设置此标志之后，如果消息传输未在 timetolive 参数所指定的时间段内成功完成，则消息将过期。

sctp_sendmsg() 函数将返回其发送的字节数。sctp_sendmsg() 函数将在出现错误时返回 -1。

如果在 *s* 参数中传递的文件描述符无效，则 sctp_sendmsg() 函数将失败，并将 errno 的值设置为 EBADF。如果在 *s* 参数中传递的文件描述符没有描述套接字，则 sctp_sendmsg() 函数将失败，并将 errno 的值设置为 ENOTSOCK。如果 *flags* 参数包括值 MSG_OOB，则 sctp_sendmsg() 函数将失败，并将 errno 的值设置为 EOPNOTSUPP。如果一对一风格套接字的 *flags* 参数包括 MSG_ABORT 或 MSG_EOF 值，则 sctp_sendmsg() 函数将失败，并将 errno 的值设置为 EOPNOTSUPP。如果没有建立关联，则 sctp_sendmsg() 函数将失败，并将 errno 的值设置为 ENOTCONN。如果套接字关闭，禁止进一步写入，则 sctp_sendmsg() 函数将失败，并将 errno 的值设置为 EPIPE。如果套接字为非阻塞套接字并且传输队列已满，则 sctp_sendmsg() 函数将失败，并将 errno 的值设置为 EAGAIN。

如果控制消息长度不正确，则 sctp_sendmsg() 函数将失败，并将 errno 的值设置为 EINVAL。如果指定的目标地址不属于关联，则 sctp_sendmsg() 函数将失败，并将 errno 的值设置为 EINVAL。如果 *stream_no* 的值在关联所支持的外发流数之内，则 sctp_sendmsg() 函数将失败，并将 errno 的值设置为 EINVAL。如果指定目标地址的地址系列不是 AF_INET 或 AF_INET6，则 sctp_sendmsg() 函数将失败，并将 errno 的值设置为 EINVAL。

sctp_send()

ssize_t **sctp_send**(int *s*, const void **msg*, size_t *len*, const struct sctp_sndrcvinfo **sinfo* , int *flags*);

sctp_send() 函数可供一对一及一对多风格套接字使用。sctp_send() 函数在发送来自 SCTP 端点的消息时启用高级 SCTP 功能。

s
此值指定 socket() 函数所创建的套接字。

msg
此值包含 sctp_send () 函数所发送的消息。

len
此值为消息的长度，以字节为单位。

sinfo
此值包含用于发送消息的参数。对于一对多风格套接字，此值可以包含消息所发送到的关联 ID。

flags
此值与 sendmsg() 函数中的标志参数相同。

sctp_send() 函数将返回其发送的字节数。sctp_send() 函数将在出现错误时返回 -1。

如果在 *s* 参数中传递的文件描述符无效，则 sctp_send() 函数将失败，并将 errno 的值设置为 EBADF。如果在 *s* 参数中传递的文件描述符没有描述套接字，则 sctp_send() 函数将失败，并将 errno 的值设置为 ENOTSOCK。如果 *sinfo* 参数的 *sinfo_flags* 字段包括值 MSG_OOB，则 sctp_send() 函数将失败，并将 errno 的值设置为 EOPNOTSUPP。如果一对一风格套接字的 *sinfo* 参数的 *sinfo_flags* 字段包括 MSG_ABORT 或 MSG_EOF 值，则 sctp_send() 函数将失败，并将 errno 的值设置为 EOPNOTSUPP。如果没有建立关联，则 sctp_send() 函数将失败，并将 errno 的值设置为 ENOTCONN。如果套接字关闭，禁止进一步写入，则 sctp_send() 函数将失败，并将 errno 的值设置为 EPIPE。如果套接字为非阻塞套接字并且传输队列已满，则 sctp_send() 函数将失败，并将 errno 的值设置为 EAGAIN。

如果控制消息长度不正确，则 sctp_send() 函数将失败，并将 errno 的值设置为 EINVAL。如果指定的目标地址不属于关联，则 sctp_send() 函数将失败，并将 errno 的值设置为 EINVAL。如果 *stream_no* 的值在关联所支持的外发流数之内，则 sctp_send() 函数将失败，并将 errno 的值设置为 EINVAL。如果指定目标地址的地址系列不是 AF_INET 或 AF_INET6，则 sctp_send() 函数将失败，并将 errno 的值设置为 EINVAL。

分叉关联

应用程序可以将一对多风格套接字上已建立的关联分叉为独立的套接字和文件描述符。对于具有多个偶发消息发送者或接收者的应用程序，如果这些发送者或接收者需要一直存在于原始一对多风格套接字之下，则独立的套接字和文件描述符非常有用。应用程序会将传输大量数据通信流量的关联分叉为独立的套接字描述符。应用程序使用 sctp_peeloff() 调用将关联分叉为独立的套接字。新套接字为一对一风格套接字。sctp_peeloff() 函数的语法如下所示：

```
int sctp_peeloff(int sock, sctp_assoc_t id);
```

sock
从 socket() 系统调用返回的原始一对多风格套接字描述符

id
要分叉为独立的文件描述符的关联的标识符

如果在 *sock* 参数中传递的套接字描述符不是一对多风格 SCTP 套接字，则 sctp_peeloff() 函数将失败并返回 EOPTNOTSUPP。如果 *id* 的值为 0 或者 *id* 的值大于在 *sock* 参数中传递的套接字描述符的最大关联数，则 sctp_peeloff() 函数将失败并返回 EINVAL。如果 sctp_peeloff() 函数无法创建新的用户文件描述符或文件结构，则此函数将失败并返回 EMFILE。

sctp_getpaddrs()

sctp_getpaddrs() 函数将返回关联中的所有对等地址。

```
int sctp_getpaddrs (int sock, sctp_assoc_t id, void **addrs);
```

当 sctp_getpaddrs() 函数成功返回时，**addrs 参数的值将指向每个地址相应类型的动态分配的压缩 sockaddr 结构数组。调用线程使用 sctp_freepaddrs() 函数释放内存。**addrs 参数的值不能为 NULL。如果 *sock* 中给定的套接字描述符用于 IPv4 套接字，则 sctp_getpaddrs() 函数将返回 IPv4 地址。如果 *sock* 中给定的套接字描述符用于 IPv6 套接字，则 sctp_getpaddrs() 函数将同时返回 IPv4 和 IPv6 地址。对于一对多风格套接字，*id* 参数指定要查询的关联。对于一对一风格套接字，sctp_getpaddrs() 函数将忽略 *id* 参数。当 sctp_getpaddrs() 函数成功返回时，它将返回关联中的对等地址数。如果此套接字上没有关联，则 sctp_getpaddrs() 函数将返回 0，并且不定义 **addrs 参数的值。如果出现错误，则 sctp_getpaddrs() 函数将返回 -1，并且不定义 **addrs 参数的值。

如果在 *sock* 参数中传递给 sctp_getpaddrs() 函数的文件描述符无效，则 sctp_getpaddrs() 函数将失败并返回 EBADF。如果在 *sock* 参数中传递给 sctp_getpaddrs() 函数的文件描述符没有描述套接字，则 sctp_getpaddrs() 函数将失败并返回 ENOTSOCK。如果在 *sock* 参数中传递给 sctp_getpaddrs() 函数的文件描述符描述了一个未连接的套接字，则 sctp_getpaddrs() 函数将失败并返回 ENOTCONN。

sctp_freepaddrs()

sctp_freepaddrs() 函数将释放所有由之前的 sctp_getpaddrs() 调用所分配的资源。sctp_freepaddrs() 函数的语法如下所示：

```
void sctp_freepaddrs(void *addrs);
```

*addrs 参数为包含 sctp_getpaddrs() 函数所返回的对等地址的数组。

sctp_getladdrs()

sctp_getladdrs() 函数将返回套接字上的所有本地绑定的地址。sctp_getladdrs() 函数的语法如下所示：

```
int sctp_getladdrs (int sock, sctp_assoc_t id, void **addrs);
```

当 sctp_getladdrs() 函数成功返回时，*addrs* 的值将指向动态分配的压缩 *sockaddr* 结构数组。*sockaddr* 结构为每个本地地址的相应类型。调用应用程序使用 sctp_freeladdrs() 函数释放内存。*addrs* 参数的值不能为 NULL。

如果 *sd* 参数引用的套接字为 IPv4 套接字，则 sctp_getladdrs() 函数将返回 IPv4 地址。如果 *sd* 参数引用的套接字为 IPv6 套接字，则 sctp_getladdrs() 函数将同时返回相应的 IPv4 或 IPv6 地址。

针对一对多风格套接字调用 sctp_getladdrs() 函数时，*id* 参数的值指定要查询的关联。sctp_getladdrs() 函数针对一对一套接字运行时将忽略 *id* 参数。

当 *id* 参数的值为 0 时，无论为何种特定关联，`sctp_getladdrs()` 函数都将返回本地绑定的地址。当 `sctp_getladdrs()` 函数成功返回时，它将报告绑定到套接字的本地地址数。如果未绑定套接字，则 `sctp_getladdrs()` 函数将返回 0，并且不定义 **addrs* 的值。如果出现错误，则 `sctp_getladdrs()` 函数将返回 -1，并且不定义 **addrs* 的值。

sctp_freeladdrs()

`sctp_freeladdrs()` 函数将释放所有由之前的 `sctp_getladdrs()` 调用所分配的资源。`sctp_freeladdrs()` 函数的语法如下所示：

```
void sctp_freeladdrs(void *addrs);
```

**addrs* 参数为包含 `sctp_getladdrs()` 函数所返回的对等地址的数组。

SCTP 用法代码示例

本节详细介绍 SCTP 套接字的两种用法。

示例 8–17 SCTP 回显客户机

```
/*
 * Copyright 2004 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/* To enable socket features used for SCTP socket. */
#define _XPG4_2
#define __EXTENSIONS__

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <netinet/sctp.h>
#include <netdb.h>

#define BUFLEN 1024

static void
usage(char *a0)
{
    fprintf(stderr, "Usage: %s <addr>\n", a0);
}

/*
 * Read from the network.
 */
static void
readit(void *vfdp)
```

示例 8–17 SCTP 回显客户机 （续）

```
{
    int   fd;
    ssize_t   n;
    char   buf[BUFLEN];
    struct msghdr  msg[1];
    struct iovec  iov[1];
    struct cmsghdr  *cmsg;
    struct sctp_sndrcvinfo *sri;
    char   cbuf[sizeof (*cmsg) + sizeof (*sri)];
    union sctp_notification *snp;

    pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);

    fd = *(int *)vfdp;

    /* Initialize the message header for receiving */
    memset(msg, 0, sizeof (*msg));
    msg->msg_control = cbuf;
    msg->msg_controllen = sizeof (*cmsg) + sizeof (*sri);
    msg->msg_flags = 0;
    cmsg = (struct cmsghdr *)cbuf;
    sri = (struct sctp_sndrcvinfo *)(cmsg + 1);
    iov->iov_base = buf;
    iov->iov_len = BUFLEN;
    msg->msg_iov = iov;
    msg->msg_iovlen = 1;

    while ((n = recvmsg(fd, msg, 0)) > 0) {
        /* Intercept notifications here */
        if (msg->msg_flags & MSG_NOTIFICATION) {
            snp = (union sctp_notification *)buf;
            printf("[ Receive notification type %u ]\n",
                snp->sn_type);
            continue;
        }
        msg->msg_control = cbuf;
        msg->msg_controllen = sizeof (*cmsg) + sizeof (*sri);
        printf("[ Receive echo (%u bytes): stream = %hu, ssn = %hu, "
            "flags = %hx, ppid = %u ]\n", n,
            sri->sinfo_stream, sri->sinfo_ssn, sri->sinfo_flags,
            sri->sinfo_ppid);
    }

    if (n < 0) {
        perror("recv");
        exit(1);
    }

    close(fd);
    exit(0);
}

#define MAX_STREAM 64

static void
echo(struct sockaddr_in *addr)
```

示例 8–17 SCTP 回显客户机 （续）

```
{
    int     fd;
    uchar_t     buf[BUFLEN];
    ssize_t    n;
    int    perr;
    pthread_t   tid;
    struct cmsghdr   *cmsg;
    struct sctp_sndrcvinfo  *sri;
    char    cbuf[sizeof (*cmsg) + sizeof (*sri)];
    struct msghdr   msg[1];
    struct iovec   iov[1];
    int    ret;
    struct sctp_initmsg   initmsg;
    struct sctp_event_subscribe events;

    /* Create a one-one SCTP socket */
    if ((fd = socket(AF_INET, SOCK_STREAM, IPPROTO_SCTP)) == -1) {
        perror("socket");
        exit(1);
    }

    /*
     * We are interested in association change events and we want
     * to get sctp_sndrcvinfo in each receive.
     */
    events.sctp_association_event = 1;
    events.sctp_data_io_event = 1;
    ret = setsockopt(fd, IPPROTO_SCTP, SCTP_EVENTS, &events,
        sizeof (events));
    if (ret < 0) {
        perror("setsockopt SCTP_EVENTS");
        exit(1);
    }

    /*
     * Set the SCTP stream parameters to tell the other side when
     * setting up the association.
     */
    memset(&initmsg, 0, sizeof(struct sctp_initmsg));
    initmsg.sinit_num_ostreams = MAX_STREAM;
    initmsg.sinit_max_instreams = MAX_STREAM;
    initmsg.sinit_max_attempts = MAX_STREAM;
    ret = setsockopt(fd, IPPROTO_SCTP, SCTP_INITMSG, &initmsg,
        sizeof(struct sctp_initmsg));
    if (ret < 0) {
        perror("setsockopt SCTP_INITMSG");
        exit(1);
    }

    if (connect(fd, (struct sockaddr *)addr, sizeof (*addr)) == -1) {
        perror("connect");
        exit(1);
    }

    /* Initialize the message header structure for sending. */
    memset(msg, 0, sizeof (*msg));
```

示例 8–17 SCTP 回显客户机 （续）

```
    iov->iov_base = buf;
    msg->msg_iov = iov;
    msg->msg_iovlen = 1;
    msg->msg_control = cbuf;
    msg->msg_controllen = sizeof (*cmsg) + sizeof (*sri);
    msg->msg_flags |= MSG_XPG4_2;

    memset(cbuf, 0, sizeof (*cmsg) + sizeof (*sri));
    cmsg = (struct cmsghdr *)cbuf;
    sri = (struct sctp_sndrcvinfo *)(cmsg + 1);

    cmsg->cmsg_len = sizeof (*cmsg) + sizeof (*sri);
    cmsg->cmsg_level = IPPROTO_SCTP;
    cmsg->cmsg_type  = SCTP_SNDRCV;

    sri->sinfo_ppid   = 1;
    /* Start sending to stream 0. */
    sri->sinfo_stream = 0;

    /* Create a thread to receive network traffic. */
    perr = pthread_create(&tid, NULL, (void *(*)(void *))readit, &fd);

    if (perr != 0) {
        fprintf(stderr, "pthread_create: %d\n", perr);
        exit(1);
    }

    /* Read from stdin and then send to the echo server. */
    while ((n = read(fileno(stdin), buf, BUFLEN)) > 0) {
        iov->iov_len = n;
        if (sendmsg(fd, msg, 0) < 0) {
            perror("sendmsg");
            exit(1);
        }
        /* Send the next message to a different stream. */
        sri->sinfo_stream = (sri->sinfo_stream + 1) % MAX_STREAM;
    }

    pthread_cancel(tid);
    close(fd);
}

int
main(int argc, char **argv)
{
    struct sockaddr_in addr[1];
    struct hostent *hp;
    int error;

    if (argc < 2) {
        usage(*argv);
        exit(1);
    }

    /* Find the host to connect to. */
    hp = getipnodebyname(argv[1], AF_INET, AI_DEFAULT, &error);
```

示例 8–17 SCTP 回显客户机 （续）

```
    if (hp == NULL) {
        fprintf(stderr, "host not found\n");
        exit(1);
    }

    addr->sin_family = AF_INET;
    addr->sin_addr.s_addr = *(ipaddr_t *)hp->h_addr_list[0];
    addr->sin_port = htons(5000);

    echo(addr);

    return (0);
}
```

示例 8–18 SCTP 回显服务器

```
/*
 * Copyright 2004 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/* To enable socket features used for SCTP socket. */
#define _XPG4_2
#define __EXTENSIONS__

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <unistd.h>
#include <netinet/sctp.h>

#define BUFLEN 1024

/*
 * Given an event notification, print out what it is.
 */
static void
handle_event(void *buf)
{
    struct sctp_assoc_change *sac;
    struct sctp_send_failed  *ssf;
    struct sctp_paddr_change *spc;
    struct sctp_remote_error *sre;
    union sctp_notification  *snp;
    char    addrbuf[INET6_ADDRSTRLEN];
    const char   *ap;
    struct sockaddr_in  *sin;
    struct sockaddr_in6  *sin6;

    snp = buf;

    switch (snp->sn_header.sn_type) {
```

示例 8–18 SCTP 回显服务器 （续）

```
    case SCTP_ASSOC_CHANGE:
        sac = &snp->sn_assoc_change;
        printf("^^^ assoc_change: state=%hu, error=%hu, instr=%hu "
            "outstr=%hu\n", sac->sac_state, sac->sac_error,
            sac->sac_inbound_streams, sac->sac_outbound_streams);
        break;
    case SCTP_SEND_FAILED:
        ssf = &snp->sn_send_failed;
        printf("^^^ sendfailed: len=%hu err=%d\n", ssf->ssf_length,
            ssf->ssf_error);
        break;
    case SCTP_PEER_ADDR_CHANGE:
        spc = &snp->sn_paddr_change;
        if (spc->spc_aaddr.ss_family == AF_INET) {
            sin = (struct sockaddr_in *)&spc->spc_aaddr;
            ap = inet_ntop(AF_INET, &sin->sin_addr, addrbuf,
                INET6_ADDRSTRLEN);
        } else {
            sin6 = (struct sockaddr_in6 *)&spc->spc_aaddr;
            ap = inet_ntop(AF_INET6, &sin6->sin6_addr, addrbuf,
                INET6_ADDRSTRLEN);
        }
        printf("^^^ intf_change: %s state=%d, error=%d\n", ap,
            spc->spc_state, spc->spc_error);
        break;
    case SCTP_REMOTE_ERROR:
        sre = &snp->sn_remote_error;
        printf("^^^ remote_error: err=%hu len=%hu\n",
            ntohs(sre->sre_error), ntohs(sre->sre_length));
        break;
    case SCTP_SHUTDOWN_EVENT:
        printf("^^^ shutdown event\n");
        break;
    default:
        printf("unknown type: %hu\n", snp->sn_header.sn_type);
        break;
    }
}

/*
 * Receive a message from the network.
 */
static void *
getmsg(int fd, struct msghdr *msg, void *buf, size_t *buflen,
    ssize_t *nrp, size_t cmsglen)
{
    ssize_t  nr = 0;
    struct iovec iov[1];

    *nrp = 0;
    iov->iov_base = buf;
    msg->msg_iov = iov;
    msg->msg_iovlen = 1;

    /* Loop until a whole message is received. */
    for (;;) {
```

示例 8–18 SCTP 回显服务器 （续）

```
        msg->msg_flags = MSG_XPG4_2;
        msg->msg_iov->iov_len = *buflen;
        msg->msg_controllen = cmsglen;

        nr += recvmsg(fd, msg, 0);
        if (nr <= 0) {
            /* EOF or error */
            *nrp = nr;
            return (NULL);
        }

        /* Whole message is received, return it. */
        if (msg->msg_flags & MSG_EOR) {
            *nrp = nr;
            return (buf);
        }

        /* Maybe we need a bigger buffer, do realloc(). */
        if (*buflen == nr) {
            buf = realloc(buf, *buflen * 2);
            if (buf == 0) {
                fprintf(stderr, "out of memory\n");
                exit(1);
            }
            *buflen *= 2;
        }

        /* Set the next read offset */
        iov->iov_base = (char *)buf + nr;
        iov->iov_len = *buflen - nr;

    }
}

/*
 * The echo server.
 */
static void
echo(int fd)
{
    ssize_t   nr;
    struct sctp_sndrcvinfo *sri;
    struct msghdr  msg[1];
    struct cmsghdr  *cmsg;
    char   cbuf[sizeof (*cmsg) + sizeof (*sri)];
    char   *buf;
    size_t   buflen;
    struct iovec  iov[1];
    size_t   cmsglen = sizeof (*cmsg) + sizeof (*sri);

    /* Allocate the initial data buffer */
    buflen = BUFLEN;
    if ((buf = malloc(BUFLEN)) == NULL) {
        fprintf(stderr, "out of memory\n");
        exit(1);
    }
```

示例 8–18 SCTP 回显服务器 （续）

```
    /* Set up the msghdr structure for receiving */
    memset(msg, 0, sizeof (*msg));
    msg->msg_control = cbuf;
    msg->msg_controllen = cmsglen;
    msg->msg_flags = 0;
    cmsg = (struct cmsghdr *)cbuf;
    sri = (struct sctp_sndrcvinfo *)(cmsg + 1);

    /* Wait for something to echo */
    while ((buf = getmsg(fd, msg, buf, &buflen, &nr, cmsglen)) != NULL) {

        /* Intercept notifications here */
        if (msg->msg_flags & MSG_NOTIFICATION) {
            handle_event(buf);
            continue;
        }

        iov->iov_base = buf;
        msg->msg_iov = iov;
        msg->msg_iovlen = 1;
        iov->iov_len = nr;
        msg->msg_control = cbuf;
        msg->msg_controllen = sizeof (*cmsg) + sizeof (*sri);

        printf("got %u bytes on stream %hu:\n", nr,
               sri->sinfo_stream);
        write(0, buf, nr);

        /* Echo it back */
        msg->msg_flags = MSG_XPG4_2;
        if (sendmsg(fd, msg, 0) < 0) {
            perror("sendmsg");
            exit(1);
        }
    }

    if (nr < 0) {
        perror("recvmsg");
    }
    close(fd);
}

int
main(void)
{
    int    lfd;
    int    cfd;
    int    onoff = 1;
    struct sockaddr_in  sin[1];
    struct sctp_event_subscribe events;
    struct sctp_initmsg  initmsg;

    if ((lfd = socket(AF_INET, SOCK_STREAM, IPPROTO_SCTP)) == -1) {
        perror("socket");
        exit(1);
```

示例 8–18 SCTP 回显服务器 （续）

```
    }

    sin->sin_family = AF_INET;
    sin->sin_port = htons(5000);
    sin->sin_addr.s_addr = INADDR_ANY;
    if (bind(lfd, (struct sockaddr *)sin, sizeof (*sin)) == -1) {
        perror("bind");
        exit(1);
    }

    if (listen(lfd, 1) == -1) {
        perror("listen");
        exit(1);
    }

    (void) memset(&initmsg, 0, sizeof(struct sctp_initmsg));
    initmsg.sinit_num_ostreams = 64;
    initmsg.sinit_max_instreams = 64;
    initmsg.sinit_max_attempts = 64;
    if (setsockopt(lfd, IPPROTO_SCTP, SCTP_INITMSG, &initmsg,
           sizeof(struct sctp_initmsg)) < 0) {
        perror("SCTP_INITMSG");
        exit (1);
    }

    /* Events to be notified for */
    (void) memset(&events, 0, sizeof (events));
    events.sctp_data_io_event = 1;
    events.sctp_association_event = 1;
    events.sctp_send_failure_event = 1;
    events.sctp_address_event = 1;
    events.sctp_peer_error_event = 1;
    events.sctp_shutdown_event = 1;

    /* Wait for new associations */
    for (;;) {
        if ((cfd = accept(lfd, NULL, 0)) == -1) {
            perror("accept");
            exit(1);
        }

        /* Enable ancillary data */
        if (setsockopt(cfd, IPPROTO_SCTP, SCTP_EVENTS, &events,
               sizeof (events)) < 0) {
            perror("setsockopt SCTP_EVENTS");
            exit(1);
        }
        /* Echo back any and all data */
        echo(cfd);
    }
}
```

◆ ◆ ◆ 第 9 章

9 使用 XTI 和 TLI 编程

本章介绍传输层接口 (Transport Layer Interface, TLI) 和 X/Open 传输接口 (X/Open Transport Interface, XTI)。诸如异步执行模式的高级主题将在第 190 页中的 “高级 XTI/TLI 主题”中进行讨论。

第 208 页中的 “XTI 接口的附加功能”中讨论了 XTI 的某些最新附加功能，如分散/集中数据传输。

OSI（开放系统互联）模型的传输层（第 4 层）是模型的最底层，为应用程序和更高的层提供端对端服务。该层会向用户隐藏底层网络的拓扑和特性。该传输层还定义一组通用于许多当前协议套件（包括 OSI 协议、传输控制协议和 TCP/IP Internet 协议套件、Xerox 网络系统 (Xerox Network Systems, XNS) 以及系统网络体系结构 (Systems Network Architecture, SNA)）的服务。

TLI 模型是根据行业标准传输服务定义 (ISO 8072) 建立的。它还可用于访问 TCP 和 UDP。XTI 和 TLI 是构成网络编程接口的一组接口。XTI 是从 SunOS 4 平台上可用的旧 TLI 接口演变而来的。虽然 XTI 代表这组接口的未来方向，但是 Solaris 操作系统仍同时支持这两种接口。Solaris 软件使用 STREAMS I/O 机制将 XTI 和 TLI 实现为用户库。

什么是 XTI 和 TLI？

注 – 本章中介绍的接口具有多线程安全性。这意味着可以在多线程应用程序中随意使用包含 XTI/TLI 接口调用的应用程序。由于这些接口调用不可重复执行，因此它们不提供线性可伸缩性。

注意 – 在异步环境中，XTI/TLI 接口行为尚未有明确的规定。请不要从信号处理程序例程使用这些接口。

TLI 是在 1986 年随 AT&T System V Release 3 引入的。TLI 当时提供了一个传输层接口 API。ISO 传输服务定义提供了 TLI 所基于的模型。现在，TLI 提供了 OSI 传输层和会话层之间的 API。TLI 接口在 UNIX 的 AT&T System V Release 4 版本中得到了进一步发展，并在 SunOS 5.6 操作系统接口中也可用。

XTI 接口由 TLI 接口演变而来，代表该系列接口的未来方向。使用 XTI 接口与使用 TLI 接口的应用程序相兼容，因此无需立即将 TLI 应用程序移植到 XTI。新应用程序可以使用 XTI 接口，而且可以在必要时将较旧的应用程序移植到 XTI。

TLI 实现为应用程序链接到的库 (`libnsl`) 中的一组接口调用。XTI 应用程序是使用 c89 前端编译的，并且必须与 `xnet` 库 (`libxnet`) 链接。有关使用 XTI 进行编译的其他信息，请参见 `standards(5)` 手册页。

注 – 使用 XTI 接口的应用程序使用 `xti.h` 头文件，而使用 TLI 接口的应用程序包含 `tiuser.h` 头文件。

与第 4 章中介绍的某些其他接口和机制一起使用时，XTI/TLI 代码可以独立于当前的传输提供器。SunOS 5 产品将某些传输提供器（例如 TCP）作为基本操作系统的一部分。传输提供器执行服务，而传输用户请求服务。传输用户向传输提供器发出服务请求。例如，通过 TCP 和 UDP 连接传输数据的请求。

利用以下两个组件，XTI/TLI 还可以用于独立于传输的编程：

- 执行传输服务（特别是传输选择和名称到地址的转换）的库例程。网络服务库包括一组为用户进程实现 XTI/TLI 的接口。请参见第 11 章，传输选择和名称到地址映射。

  使用 TLI 的程序应通过在编译时指定 `-l nsl` 选项与 `libnsl` 网络服务库链接。

  使用 XTI 的程序应通过在编译时指定 `-l xnet` 选项与 `xnet` 库链接。
- 定义传输例程调用顺序的状态转换规则。有关状态转换规则的更多信息，请参见第 200 页中的 “状态转换”。状态表基于事件的状态和处理方式来定义库调用的合法序列。这些事件包括用户生成的库调用，以及提供器生成的事件指示。XTI/TLI 程序员在使用接口之前应了解所有状态转换。

XTI/TLI 读/写接口

用户可能需要对现有程序（如 `/usr/bin/cat`）使用 `exec(2)` 来建立传输连接，以便处理通过该连接收到的数据。现有程序使用 `read(2)` 和 `write(2)`。XTI/TLI 并不直接支持传输提供器的读/写接口，但是有一个接口可用。利用此接口，可以在数据传输阶段通过传输连接发出 `read(2)` 和 `write(2)` 调用。本节介绍 XTI/TLI 连接模式服务的读/写接口。此接口不适用于无连接模式服务。

示例 9–1 读/写接口

```
#include <stropts.h>

/* Same local management and connection establishment steps. */

if (ioctl(fd, I_PUSH, "tirdwr") == –1) {
    perror(“I_PUSH of tirdwr failed”);
    exit(5);
}
close(0);
dup(fd);
execl(“/usr/bin/cat”, “/usr/bin/cat”, (char *) 0);
perror(“exec of /usr/bin/cat failed”);
exit(6);
```

客户机通过将 tirdwr 推送到与传输端点关联的流来调用读/写接口。有关 I_PUSH 的说明，请参见 streamio(7I) 手册页。tirdwr 模块将位于传输提供器之上的 XTI/TLI 转换为纯读/写接口。使用该模块后，客户机将调用 close(2) 和 dup(2) 来建立传输端点作为其标准输入文件，并使用 /usr/bin/cat 来处理输入。

将 tirdwr 推送到传输提供器会强制 XTI/TLI 使用 read(2) 和 write(2) 语义。使用 read 和 write 语义时，XTI/TLI 不保留消息边界。请从传输提供器弹出 tirdwr 以恢复 XTI/TLI 语义（有关 I_POP 的说明，请参见 streamio(7I) 手册页）。

注意 – 仅当传输端点位于数据传输阶段时，才将 tirdwr 模块推送到流。推送该模块之后，用户不能调用任何 XTI/TLI 例程。如果用户调用 XTI/TLI 例程，则 tirdwr 会在流上生成致命的协议错误 EPROTO，表明该例程不可用。如果随后将 tirdwr 模块弹出该流，则该传输连接将异常中止。有关 I_POP 的说明，请参见 streamio(7I) 手册页。

写入数据

使用 write(2) 通过传输连接发送数据之后，tirdwr 将数据传递到传输提供器。如果发送的数据包长度为零（该机制允许），则 tirdwr 会放弃该消息。如果传输连接异常中止，则会在流上生成挂起状态，后续 write(2) 调用将失败，并且 errno 被设置为 ENXIO。例如，远程用户使用 t_snddis(3NSL) 异常中止连接时可能会出现此问题。挂起之后，仍可以检索任何可用数据。

读取数据

可使用 read(2) 接收到达传输连接的数据。tirdwr 会传递来自传输提供器的数据。tirdwr 模块可处理从提供器传递至用户的任何其他事件或请求，如下所述：

- read(2) 无法识别发送给用户的加速数据。如果 read(2) 接收到加速数据请求，则 tirdwr 将在流上生成一个致命的协议错误 EPROTO。该错误将导致后续系统调用失败。请勿使用 read(2) 接收加速数据。

- tirdwr 会放弃异常断开请求并在流上生成挂起状态。后续 read(2) 调用检索所有剩余数据，然后针对所有后续调用返回零，指示文件结束。
- tirdwr 会放弃顺序释放请求并向用户发送一条长度为零的消息。如 read(2) 手册页中所述，该消息通过返回 0 来通知用户文件结束。
- 如果 read(2) 接收到任何其他 XTI/TLI 请求，则 tirdwr 将在流上生成一个致命的协议错误 EPROTO。这将导致后续系统调用失败。如果用户在连接建立之后将 tirdwr 推送到流，则 tirdwr 不会生成请求。

关闭连接

利用流上的 tirdwr，在连接有效期间内，可以通过传输连接发送和接收数据。用户可通过关闭与传输端点关联的文件描述符来终止连接，也可通过将 tirdwr 模块弹出流来终止连接。在上述任一情况下，tirdwr 都会执行以下操作：

- 如果 tirdwr 接收到顺序释放请求，则将该请求传递到传输提供器以完成该连接的顺序释放。数据传输完成后，启动顺序释放过程的远程用户将接收到预期请求。
- 如果 tirdwr 接收到断开请求，则不执行任何特殊操作。
- 如果 tirdwr 接收到的既不是顺序释放请求，也不是断开请求，则会将一个断开请求传递到传输提供器以异常中止该连接。
- 如果流上发生错误，并且 tirdwr 未接收到断开请求，则会将一个断开请求传递到传输提供器。

将 tirdwr 推送到流之后，进程便不能启动顺序释放。如果位于传输连接另一端的用户启动了顺序释放，tirdwr 将处理该释放。如果该部分中的客户机与服务器程序进行通信，则该服务器会使用顺序释放请求终止数据的传输。然后，该服务器将等待客户机的相应请求。此时，该客户机将退出并关闭传输端点。关闭文件描述符之后，tirdwr 会启动来自连接客户端的顺序释放请求。该释放会生成阻塞服务器的请求。

某些协议（如 TCP）要求此顺序释放以确保完整传送数据。

高级 XTI/TLI 主题

本节介绍其他 XTI/TLI 概念：

- 第 191 页中的 “异步执行模式”介绍某些库调用的可选非阻塞（异步）模式。
- 第 191 页中的 “高级 XTI/TLI 编程示例”提供了一个服务器的程序示例，该服务器支持多个未完成的连接请求，并以事件驱动的方式进行操作。

异步执行模式

许多 XTI/TLI 库例程会阻塞以等待传入事件。但是，无论出于何种原因，都不应阻塞某些时间关键应用程序。应用程序可以在等待某个异步 XTI/TLI 事件时进行本地处理。

应用程序可以通过组合异步功能和 XTI/TLI 库例程的非阻塞模式来访问 XTI/TLI 事件的异步处理。有关使用 `poll(2)` 系统调用和 `I_SETSIG` `ioctl(2)` 命令来异步处理事件的信息，请参见《ONC+ Developer's Guide》。

可以在特定的非阻塞模式中运行每个由于某个事件而阻塞的 XTI/TLI 例程。例如，`t_listen(3NSL)` 通常会由于连接请求而阻塞。服务器通过在非阻塞（或异步）模式中调用 `t_listen(3NSL)`，可以定期针对已排队的连接请求轮询传输端点。可通过在文件描述符中设置 `O_NDELAY` 或 `O_NONBLOCK` 来启用异步模式。在调用 XTI/TLI 例程之前，通过 `t_open(3NSL)` 或者通过调用 `fcntl(2)` 将这些模式设置为标志。使用 `fcntl(2)` 可随时启用或禁用该模式。本章中的所有程序示例都使用缺省的同步处理模式。

使用 `O_NDELAY` 或 `O_NONBLOCK` 会以不同方式影响每个 XTI/TLI 例程。需要针对特定例程确定 `O_NDELAY` 或 `O_NONBLOCK` 的确切语义。

高级 XTI/TLI 编程示例

示例 9–2 说明了两个重要概念。第一个是服务器管理多个未完成连接请求的功能。第二个是以事件驱动方式使用 XTI/TLI 和系统调用接口。

使用 XTI/TLI，一台服务器可以管理多个未完成的连接请求。接收多个同时连接请求的一个原因是设置客户机的优先级。一台服务器可以接收多个连接请求，并基于每台客户机的优先级按顺序接受这些请求。

处理多个未完成连接请求的第二个原因是克服单线程处理的限制。根据传输提供器，服务器正在处理一个连接请求时，其他客户机会将该服务器视为处于忙状态。如果同时处理多个连接请求，则该服务器仅在超过最大数量的客户机同时尝试呼叫该服务器时处于忙状态。

该服务器示例是事件驱动的：进程针对传入的 XTI/TLI 事件轮询传输端点，并针对接收到的事件执行相应操作。以下示例说明针对传入事件轮询多个传输端点的功能。

示例 9–2 端点建立（可转换为多个连接）

```
#include <tiuser.h>
#include <fcntl.h>
#include <stdio.h>
#include <poll.h>
#include <stropts.h>
#include <signal.h>

#define NUM_FDS 1
```

示例 9–2 端点建立（可转换为多个连接） （续）

```
#define MAX_CONN_IND 4
#define SRV_ADDR 1                 /* server's well known address */

int conn_fd;                       /* server connection here */
extern int t_errno;
/* holds connect requests */
struct t_call *calls[NUM_FDS][MAX_CONN_IND];

main()
{
   struct pollfd pollfds[NUM_FDS];
   struct t_bind *bind;
   int i;

   /*
    * Only opening and binding one transport endpoint, but more can
    * be supported
    */
   if ((pollfds[0].fd = t_open("/dev/tivc", O_RDWR,
         (struct t_info *) NULL)) == -1) {
      t_error("t_open failed");
      exit(1);
   }
   if ((bind = (struct t_bind *) t_alloc(pollfds[0].fd, T_BIND,
         T_ALL)) == (struct t_bind *) NULL) {
      t_error("t_alloc of t_bind structure failed");
      exit(2);
   }
   bind->qlen = MAX_CONN_IND;
   bind->addr.len = sizeof(int);
   *(int *) bind->addr.buf = SRV_ADDR;
   if (t_bind(pollfds[0].fd, bind, bind) == -1) {
      t_error("t_bind failed");
      exit(3);
   }
   /* Was the correct address bound? */
   if (bind->addr.len != sizeof(int) ||
      *(int *)bind->addr.buf != SRV_ADDR) {
      fprintf(stderr, "t_bind bound wrong address\n");
      exit(4);
   }
}
```

t_open(3NSL) 返回的文件描述符存储在针对传入数据控制传输端点轮询的 pollfd 结构中。请参见 poll(2) 手册页。本示例中只建立了一个传输端点。但是，编写示例的其余部分是为了管理多个传输端点。通过对示例 9–2 进行少量更改即可支持数个端点。

该服务器针对 t_bind(3NSL) 将 qlen 设置为大于 1 的值。该值指定服务器应对多个未完成的连接请求进行排队。该服务器先接受当前的连接请求，然后再接受其他连接请求。本示例最多可以对 MAX_CONN_IND 个连接请求进行排队。如果传输提供器不支持 MAX_CONN_IND 个未完成的连接请求，则它可以对 qlen 的值进行协商，使其更小。

在服务器绑定其地址并且可以处理连接请求之后，其行为如以下示例所示。

示例 9–3 处理连接请求

```
pollfds[0].events = POLLIN;

while (TRUE) {
    if (poll(pollfds, NUM_FDS, –1) == –1) {
        perror(“poll failed”);
        exit(5);
    }
    for (i = 0; i < NUM_FDS; i++) {
        switch (pollfds[i].revents) {
            default:
                perror(“poll returned error event”);
                exit(6);
            case 0:
                continue;
            case POLLIN:
                do_event(i, pollfds[i].fd);
                service_conn_ind(i, pollfds[i].fd);
        }
    }
}
```

`pollfd` 结构的 `events` 字段设置为 `POLLIN`，用以通知服务器任何传入的 XTI/TLI 事件。然后，服务器会进入针对事件轮询传输端点的死循环，并在事件发生时对其进行处理。

`poll(2)` 调用将针对传入事件进行无限阻塞。返回时，服务器将针对新事件检查每项（一个传输端点为一项）的 `revents` 值。如果 `revents` 为 `0`，则端点未生成任何事件，服务器将继续检查下一个端点。如果 `revents` 为 `POLLIN`，则端点上有事件。服务器会调用 `do_event` 来处理此事件。`revents` 中的任何其他值指示端点上存在错误，并且服务器将退出。由于存在多个端点，因此服务器应关闭该描述符并继续检查。

每次服务器迭代循环时，都会调用 `service_conn_ind` 来处理任何未完成的连接请求。如果还有其他暂挂的连接请求，`service_conn_ind` 将保存新的连接请求，并在稍后对其做出响应。

服务器会调用以下示例中的 `do_event` 来处理传入事件。

示例 9–4 事件处理例程

```
do_event( slot, fd)
int slot;
int fd;
{
   struct t_discon *discon;
   int i;

   switch (t_look(fd)) {
   default:
      fprintf(stderr, "t_look: unexpected event\n");
      exit(7);
   case T_ERROR:
      fprintf(stderr, "t_look returned T_ERROR event\n");
```

示例 9–4 事件处理例程 （续）

```
      exit(8);
   case −1:
      t_error("t_look failed");
      exit(9);
   case 0:
      /* since POLLIN returned, this should not happen */
      fprintf(stderr,"t_look returned no event\n");
      exit(10);
   case T_LISTEN:
      /* find free element in calls array */
      for (i = 0; i < MAX_CONN_IND; i++) {
         if (calls[slot][i] == (struct t_call *) NULL)
            break;
      }
      if ((calls[slot][i] = (struct t_call *) t_alloc( fd, T_CALL,
               T_ALL)) == (struct t_call *) NULL) {
         t_error("t_alloc of t_call structure failed");
         exit(11);
      }
      if (t_listen(fd, calls[slot][i] ) == −1) {
         t_error("t_listen failed");
         exit(12);
      }
      break;
   case T_DISCONNECT:
      discon = (struct t_discon *) t_alloc(fd, T_DIS, T_ALL);
      if (discon == (struct t_discon *) NULL) {
         t_error("t_alloc of t_discon structure failed");
         exit(13)
      }
      if(t_rcvdis( fd, discon) == −1) {
         t_error("t_rcvdis failed");
         exit(14);
      }
      /* find call ind in array and delete it */
      for (i = 0; i < MAX_CONN_IND; i++) {
         if (discon->sequence == calls[slot][i]->sequence) {
            t_free(calls[slot][i], T_CALL);
            calls[slot][i] = (struct t_call *) NULL;
         }
      }
      t_free(discon, T_DIS);
      break;
   }
}
```

示例 9–4 中的参数为一个编号 (*slot*) 和一个文件描述符 (*fd*)。*slot* 为全局数组 `calls` 的索引，它针对每个传输端点都具有一项。每一项都是用于保存端点的传入连接请求的 `t_call` 结构的数组。

`do_event` 模块会调用 `t_look(3NSL)` 以标识 *fd* 指定的端点上的 XTI/TLI 事件。如果该事件是连接请求（`T_LISTEN` 事件）或断开请求（`T_DISCONNECT` 事件），则会处理该事件。否则，服务器会显示错误消息并退出。

对于连接请求，do_event 会扫描未完成连接请求的数组，以查找第一个可用项。将为该项分配 t_call 结构，并由 t_listen(3NSL) 接收连接请求。该数组很大，足以容纳最大数量的未完成连接请求。将延迟连接请求的处理。

断开请求必须对应于先前的连接请求。do_event 模块会分配 t_discon 结构来接收请求。此结构具有以下字段：

```
struct t_discon {
    struct    netbuf    udata;
    int       reason;
    int       sequence;
}
```

udata 结构包含使用断开请求发送的所有用户数据。reason 的值包含特定于协议的断开原因代码。sequence 的值标识与断开请求匹配的连接请求。

服务器会调用 t_rcvdis(3NSL) 来接收断开请求。将会扫描连接请求的数组，以查找其序列号与断开请求中的 sequence 号匹配的连接请求。找到该连接请求时，将释放其结构并将该项设置为 NULL。

在传输端点上找到事件时，将调用 service_conn_ind 来处理该端点上的所有已排队连接请求，如以下示例所示。

示例 9–5 处理所有连接请求

```
service_conn_ind(slot, fd)
{
   int i;

    for (i = 0; i < MAX_CONN_IND; i++) {
      if (calls[slot][i] == (struct t_call *) NULL)
         continue;
      if((conn_fd = t_open( “/dev/tivc”, O_RDWR,
            (struct t_info *) NULL)) == −1) {
         t_error("open failed");
         exit(15);
      }
      if (t_bind(conn_fd, (struct t_bind *) NULL,
            (struct t_bind *) NULL) == −1) {
         t_error("t_bind failed");
         exit(16);
      }
      if (t_accept(fd, conn_fd, calls[slot][i]) == −1) {
         if (t_errno == TLOOK) {
            t_close(conn_fd);
            return;
         }
         t_error("t_accept failed");
         exit(167);
      }
      t_free(calls[slot][i], T_CALL);
      calls[slot][i] = (struct t_call *) NULL;
      run_server(fd);
   }
```

示例 9–5 处理所有连接请求 （续）

```
}
```

对于每个传输端点，将扫描未完成连接请求的数组。对于每个请求，服务器都会打开一个响应传输端点，将地址绑定到该端点，并在该端点接受连接。如果在接受当前请求之前有其他连接请求或断开请求到达，则 t_accept(3NSL) 将失败，并将 t_errno 设置为 TLOOK。如果传输端点上存在任何暂挂连接请求事件或断开请求事件，则无法接受未完成的连接请求。

如果出现该错误，则会关闭响应传输端点，service_conn_ind 会立即返回，并保存当前连接请求以便稍后进行处理。此活动将导致进入服务器的主处理循环，并通过下一个 poll(2) 调用发现新事件。这样，用户便可以对多个连接请求进行排队。

最终，会处理所有事件，而 service_conn_ind 也可以接受每个连接请求。

异步联网

本节讨论了针对实时应用程序使用 XTI/TLI 进行异步网络通信的技术。SunOS 平台使用 STREAMS 异步功能和 XTI/TLI 库例程非阻塞模式的组合来提供对 XTI/TLI 事件的异步网络处理的支持。

联网编程模型

与文件和设备 I/O 类似，网络传输可以通过进程服务请求以同步或异步方式完成。

同步联网的执行方式与同步文件和设备 I/O 类似。与 write(2) 接口类似，发送请求会在缓冲消息之后返回，但是如果缓冲区不立即可用，则可能暂停调用过程。与 read(2) 接口类似，接收请求在数据到达以满足请求之前会暂停执行调用过程。由于不存在针对传输服务的保证绑定，因此同步联网不适用于必须具有与其他设备相关的实时行为的进程。

异步联网通过非阻塞服务请求提供。此外，可能建立连接时，可能发送数据时，或可能接收数据时，应用程序可以请求异步通知。

异步无连接模式服务

通过在传输数据时针对非阻塞服务配置端点，以及针对异步通知轮询或接收异步通知来执行异步无连接模式联网。如果使用异步通知，则通常在信号处理程序中执行数据的实际接收。

使端点异步

使用 t_open(3NSL) 建立端点，并使用 t_bind(3NSL) 建立其标识之后，可以针对异步服务配置该端点。使用 fcntl(2) 接口在端点上设置 O_NONBLOCK 标志。这样，无立即可用缓冲区的 t_sndudata(3NSL) 调用将返回 -1，且 t_errno 设置为 TFLOW。同样，无可用数据的 t_rcvudata(3NSL) 调用将返回 -1，且 t_errno 设置为 TNODATA。

异步网络传输

虽然应用程序可以使用 poll(2) 来定期检查数据是否到达或等待在端点上接收数据，但是可能需要在数据到达时接收异步通知。结合使用 ioctl(2) 和 I_SETSIG 命令可请求在端点收到数据时将 SIGPOLL 信号发送到该进程。应用程序应检查多条消息导致单个信号的可能性。

在以下示例中，protocol 为应用程序选择的传输协议的名称。

```
#include <sys/types.h>
#include <tiuser.h>
#include <signal.h>
#include <stropts.h>

int              fd;
struct t_bind    *bind;
void             sigpoll(int);

    fd = t_open(protocol, O_RDWR, (struct t_info *) NULL);

    bind = (struct t_bind *) t_alloc(fd, T_BIND, T_ADDR);
    ...     /* set up binding address */
    t_bind(fd, bind, bin

    /* make endpoint non-blocking */
    fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | O_NONBLOCK);

    /* establish signal handler for SIGPOLL */
    signal(SIGPOLL, sigpoll);

    /* request SIGPOLL signal when receive data is available */
    ioctl(fd, I_SETSIG, S_INPUT | S_HIPRI);

    ...

void sigpoll(int sig)
{
    int                  flags;
    struct t_unitdata    ud;

    for (;;) {
        ... /* initialize ud */
        if (t_rcvudata(fd, &ud, &flags) < 0) {
            if (t_errno == TNODATA)
                break;  /* no more messages */
            ... /* process other error conditions */
    }
    ... /* process message in ud */
}
```

异步连接模式服务

对于连接模式服务，应用程序不仅可以安排数据传输，还可以对异步完成的连接建立本身进行安排。操作顺序取决于该进程是尝试连接到其他进程还是等待连接尝试。

异步建立连接

进程可以尝试某个连接并异步完成该连接。进程将首先创建连接端点，然后使用 fcntl(2) 针对非阻塞操作配置该端点。与无连接数据传输一样，还可以针对连接完成时的异步通知和后续数据传输来配置该端点。然后，连接进程将使用 t_connect(3NSL) 开始进行传输设置。随后将使用 t_rcvconnect(3NSL) 来确认连接的建立。

异步使用连接

要异步等待连接，进程首先要建立一个绑定到服务地址的非阻塞端点。当 poll(2) 的结果或异步通知指示连接请求到达时，该进程可以使用 t_listen(3NSL) 来获取连接请求。要接受该连接，进程可以使用 t_accept(3NSL)。必须针对异步数据传输分别配置响应端点。

以下示例说明如何异步请求连接。

```
#include <tiuser.h>
int             fd;
struct t_call   *call;

fd = /* establish a non-blocking endpoint */

call = (struct t_call *) t_alloc(fd, T_CALL, T_ADDR);
/* initialize call structure */
t_connect(fd, call, call);

/* connection request is now proceeding asynchronously */

/* receive indication that connection has been accepted */
t_rcvconnect(fd, &call);
```

以下示例说明如何异步侦听连接。

```
#include <tiuser.h>
int             fd, res_fd;
struct t_call   call;

fd = /* establish non-blocking endpoint */

/*receive indication that connection request has arrived */
call = (struct t_call *) t_alloc(fd, T_CALL, T_ALL);
t_listen(fd, &call);

/* determine whether or not to accept connection */
res_fd = /* establish non-blocking endpoint for response */
t_accept(fd, res_fd, call);
```

异步打开

有时，可能要求应用程序动态打开从远程主机挂载的文件系统中的常规文件，或者其初始化可能延迟的设备上的常规文件。但是，正在处理此类打开文件的请求的同时，应用程序不能实现对其他事件的实时响应。SunOS 软件通过以下方法解决了此问题：让另一个进程处理文件的实际打开操作，然后将文件描述符传递到实时进程。

传输文件描述符

SunOS 平台提供的 STREAMS 接口提供了一种将开放式文件描述符从一个进程传递到另一个进程的机制。带有开放式文件描述符的进程使用带有命令参数 `I_SENDFD` 的 `ioctl(2)`。另一个进程通过调用带有命令参数 `I_RECVFD` 的 `ioctl(2)` 来获取文件描述符。

在以下示例中，父进程输出有关测试文件的信息，并创建一个管道。接下来，进程会创建一个子进程，该子进程可打开该测试文件并通过管道将开放式文件描述符传递回父进程。随后父进程会显示关于新文件描述符的状态信息。

示例 9–6 文件描述符传送

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stropts.h>
#include <stdio.h>

#define TESTFILE "/dev/null"
main(int argc, char *argv[])
{
    int fd;
    int pipefd[2];
    struct stat statbuf;

    stat(TESTFILE, &statbuf);
    statout(TESTFILE, &statbuf);
    pipe(pipefd);
    if (fork() == 0) {
        close(pipefd[0]);
        sendfd(pipefd[1]);
    } else {
        close(pipefd[1])
        recvfd(pipefd[0]);
    }
}

sendfd(int p)
{
    int tfd;

    tfd = open(TESTFILE, O_RDWR);
    ioctl(p, I_SENDFD, tfd);
}

recvfd(int p)
{
```

示例 9–6 文件描述符传送 （续）

```
    struct strrecvfd rfdbuf;
    struct stat statbuf;
    char            fdbuf[32];

    ioctl(p, I_RECVFD, &rfdbuf);
    fstat(rfdbuf.fd, &statbuf);
    sprintf(fdbuf, "recvfd=%d", rfdbuf.fd);
    statout(fdbuf, &statbuf);
}

statout(char *f, struct stat *s)
{
    printf("stat: from=%s mode=0%o, ino=%ld, dev=%lx, rdev=%lx\n",
        f, s->st_mode, s->st_ino, s->st_dev, s->st_rdev);
    fflush(stdout);
}
```

状态转换

以下各节中的表介绍与 XTI/TLI 关联的所有状态转换。

XTI/TLI 状态

下表定义 XTI/TLI 状态转换中使用的状态以及服务类型。

表 9–1 XTI/TLI 状态转换和服务类型

状态	说明	服务类型
T_UNINIT	未初始化－接口的初始状态和最终状态	T_COTS、T_COTS_ORD、T_CLTS
T_UNBND	已初始化但未绑定	T_COTS、T_COTS_ORD、T_CLTS
T_IDLE	未建立连接	T_COTS、T_COTS_ORD、T_CLTS
T_OUTCON	针对客户机暂挂的传出连接	T_COTS、T_COTS_ORD
T_INCON	针对服务器暂挂的传入连接	T_COTS、T_COTS_ORD
T_DATAXFER	数据传输	T_COTS、T_COTS_ORD
T_OUTREL	传出顺序释放（等待顺序释放请求）	T_COTS_ORD
T_INREL	传入顺序释放（等待发送顺序释放请求）	T_COTS_ORD

传出事件

下表中介绍的传出事件与指定传输例程（发送请求或响应传输提供器的例程）返回的状态相对应。在该表中，某些事件（例如 "accept"）根据发生事件的上下文来区分。上下文基于以下变量的值：

- *ocnt*－未完成连接请求的计数
- *fd*－当前传输端点的文件描述符
- *resfd*－接受连接的传输端点的文件描述符

表 9–2 传出事件

事件	说明	服务类型
opened	成功返回 t_open(3NSL)	T_COTS、T_COTS_ORD、T_CLTS
bind	成功返回 t_bind(3NSL)	T_COTS、T_COTS_ORD、T_CLTS
optmgmt	成功返回 t_optmgmt(3NSL)	T_COTS、T_COTS_ORD、T_CLTS
unbind	成功返回 t_unbind(3NSL)	T_COTS、T_COTS_ORD、T_CLTS
closed	成功返回 t_close(3NSL)	T_COTS、T_COTS_ORD、T_CLT
connect1	在同步模式下成功返回 t_connect(3NSL)	T_COTS、T_COTS_ORD
connect2	异步模式下 t_connect(3NSL) 上的 TNODATA 错误，或者由于传输端点上到达断开请求而产生的 TLOOK 错误	T_COTS、T_COTS_ORD
accept1	成功返回 t_accept(3NSL)，且 ocnt == 1、fd == resfd	T_COTS、T_COTS_ORD
accept2	成功返回 t_accept(3NSL)，且 ocnt== 1、fd!= resfd	T_COTS、T_COTS_ORD
accept3	成功返回 t_accept(3NSL)，且 ocnt > 1	T_COTS、T_COTS_ORD
snd	成功返回 t_snd(3NSL)	T_COTS、T_COTS_ORD
snddis1	成功返回 t_snddis(3NSL)，且 ocnt <= 1	T_COTS、T_COTS_ORD
snddis2	成功返回 t_snddis(3NSL)，且 ocnt > 1	T_COTS、T_COTS_ORD
sndrel	成功返回 t_sndrel(3NSL)	T_COTS_ORD
sndudata	成功返回 t_sndudata(3NSL)	T_CLTS

传入事件

传入事件与指定例程的成功返回相对应。这些例程返回来自传输提供器的数据或事件信息。不与例程的返回直接关联的唯一传入事件是 pass_conn，该传入事件在连接传送到其他端点时发生。虽然未在连接传递到的端点上调用 XTI/TLI 例程，但是会在该端点上发生事件。

在下表中，根据 ocnt 值（端点上未完成的连接请求计数）来区分 rcvdis 事件。

表 9–3 传入事件

事件	说明	服务类型
listen	成功返回 t_listen(3NSL)	T_COTS、T_COTS_ORD
rcvconnect	成功返回 t_rcvconnect(3NSL)	T_COTS、T_COTS_ORD
rcv	成功返回 t_rcv(3NSL)	T_COTS、T_COTS_ORD
rcvdis1	成功返回 t_rcvdis(3NSL) rcvdis1t_rcvdis()，且 onct <= 0	T_COTS、T_COTS_ORD
rcvdis2	成功返回 t_rcvdis(3NSL)，且 ocnt == 1	T_COTS、T_COTS_ORD
rcvdis3	成功返回 t_rcvdis(3NSL)，且 ocnt > 1	T_COTS、T_COTS_ORD
rcvrel	成功返回 t_rcvrel(3NSL)	T_COTS_ORD
rcvudata	成功返回 t_rcvudata(3NSL)	T_CLTS
rcvuderr	成功返回 t_rcvuderr(3NSL)	T_CLTS
pass_conn	接收传递的连接	T_COTS、T_COTS_ORD

状态表

状态表描述 XTI/TLI 状态转换。每个框都包含下一个状态，假设已给定当前状态（列）和当前事件（行）。空框表示无效状态/事件组合。每个框还可具有一个操作列表。必须按照框中指定的顺序完成操作。

研究状态表时，应了解以下内容：

- t_close(3NSL) 针对面向连接的传输提供器终止已建立的连接。连接将按顺序终止或异常终止，具体取决于传输提供器支持的服务类型。请参见 t_getinfo(3NSL) 手册页。
- 如果传输用户发出无序接口调用，该接口将失败并将 t_errno 设置为 TOUTSTATE。状态不会更改。
- t_connect(3NSL) 之后的错误代码 TLOOK 或 TNODATA 可以导致状态更改。状态表假定正确使用 XTI/TLI。

- 任何其他传输错误都不会更改状态，除非接口的手册页另行指出。
- 可从状态表中排除支持接口 t_getinfo(3NSL)、t_getstate(3NSL)、t_alloc(3NSL)、t_free(3NSL)、t_sync(3NSL)、t_look(3NSL) 以及 t_error(3NSL)，因为它们不会影响状态。

下面各表中列出的某些状态转换提供传输用户必须执行的操作。每项操作都由通过下面列出的方法得出的数字表示：

- 将未完成连接请求的计数设置为零
- 递增未完成连接请求的计数
- 递减未完成连接请求的计数
- 将连接传递到其他传输端点，如 t_accept(3NSL) 手册页中所示

下表显示了端点建立状态。

表 9–4 连接建立状态

事件/状态	T_UNINIT	T_UNBND	T_IDLE
opened	T_UNBND		
bind		T_IDLE[1]	
optmgmt（仅适用于 TLI）			T_IDLE
unbind			T_UNBND
closed		T_UNINIT	

下表显示了连接模式下的数据传输。

表 9–5 连接模式状态：第 1 部分

事件/状态	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER
connect1	T_DATAXFER			
connect2	T_OUTCON			
rcvconnect		T_DATAXFER		
listen	T_INCON [2]		T_INCON [2]	
accept1			T_DATAXFER [3]	
accept2			T_IDLE [3] [4]	
accept3			T_INCON [3] [4]	
snd				T_DATAXFER
rcv				T_DATAXFER

表 9–5 连接模式状态：第 1 部分 （续）

事件/状态	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER
snddis1		T_IDLE	T_IDLE [3]	T_IDLE
snddis2			T_INCON [3]	
rcvdis1		T_IDLE		T_IDLE
rcvdis2			T_IDLE [3]	
rcvdis3			T_INCON [3]	
sndrel				T_OUTREL
rcvrel				T_INREL
pass_conn	T_DATAXFER			
optmgmt	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER
closed	T_UNINIT	T_UNINIT	T_UNINIT	T_UNINIT

下表显示了连接模式下的连接建立/连接释放/数据传输。

表 9–6 连接模式状态：第 2 部分

事件/状态	T_OUTREL	T_INREL	T_UNBND
connect1			
connect2			
rcvconnect			
listen			
accept1			
accept2			
accept3			
snd		T_INREL	
rcv	T_OUTREL		
snddis1	T_IDLE	T_IDLE	
snddis2			
rcvdis1	T_IDLE	T_IDLE	
rcvdis2			
rcvdis3			

表 9–6 连接模式状态：第 2 部分 （续）

事件/状态	T_OUTREL	T_INREL	T_UNBND
sndrel		T_IDLE	
rcvrel	T_IDLE		
pass_conn			T_DATAXFER
optmgmt	T_OUTREL	T_INREL	T_UNBND
closed	T_UNINIT	T_UNINIT	

下表显示了无连接模式状态。

表 9–7 无连接模式状态

事件/状态	T_IDLE
snudata	T_IDLE
rcvdata	T_IDLE
rcvuderr	T_IDLE

协议独立性准则

通用于许多传输协议的 XTI/TLI 服务集对应用程序提供协议独立性。并非所有传输协议都支持所有的 XTI/TLI 服务。如果软件必须在多种协议环境中运行，请仅使用常见服务。

下面列出了可能不通用于所有传输协议的服务。

- 在连接模式服务中，并非所有传输提供器都支持传输服务数据单元 (Transport Service Data Unit, TSDU)。请不要对保留连接的逻辑数据边界做出假设。
- t_open(3NSL) 和 t_getinfo(3NSL) 例程返回特定于协议和实现的服务限制。使用这些限制可分配缓冲区以存储特定于协议的传输地址和选项。
- 请勿将用户数据与连接请求和断开请求（如 t_connect(3NSL) 和 t_snddis(3NSL)）一起发送。并非所有传输协议都可以使用此方法。
- t_call 结构中用于 t_listen(3NSL) 的缓冲区必须足够大，以便保留客户机在建立连接期间发送的任何数据。使用 t_alloc(3NSL) 的 T_ALL 参数可设置最大缓冲区大小，用于存储当前传输提供器的地址、选项以及用户数据。
- 请勿在客户端端点上的 t_bind(3NSL) 上指定协议地址。传输提供器应该为传输端点指定适当的地址。对于 t_bind(3NSL)，服务器应该以不要求了解传输提供器名称空间的方法来检索其地址。
- 请勿对传输地址的格式做出假设。传输地址在程序中不应为常量。第 11 章，传输选择和名称到地址映射包含有关传输选择的详细信息。

- 与 t_rcvdis(3NSL) 关联的原因代码与协议相关。如果协议独立性很重要，请不要解释这些原因代码。
- t_rcvuderr(3NSL) 错误代码与协议相关。如果需要考虑协议独立性，请不要解释这些错误代码。
- 请勿将设备名称编码到程序中。设备节点标识特定的传输提供器，不独立于协议。有关传输选择的详细信息，请参见第 11 章，传输选择和名称到地址映射。
- 请勿在用于多个协议环境的程序中使用由 t_sndrel(3NSL) 和 t_rcvrel(3NSL) 提供的连接模式服务的可选顺序释放功能。并非所有基于连接的传输协议都支持此功能。使用此功能可防止程序与开放系统成功通信。

XTI/TLI 与套接字接口

XTI/TLI 和套接字是处理相同任务的不同方法。虽然它们提供功能上类似的机制和服务，但并不提供例程或低级服务的一对一兼容性。决定移植应用程序之前，请查看 XTI/TLI 接口和基于套接字的接口之间的相似之处与不同之处。

以下问题与传输独立性相关，并且可能对 RPC 应用程序产生某些影响：

- 特权端口－特权端口是 TCP/IP Internet 协议的 Berkeley 软件分发 (Berkeley Software Distribution, BSD) 实现的产物。这些端口不可移植。独立于传输的环境中不支持特权端口概念。
- 不透明地址－不能以独立于传输的方式分隔命名某主机的地址部分和命名该主机上服务的地址部分。请确保更改假设能分辨网络服务的主机地址的任何代码。
- 广播－不存在独立于传输形式的广播地址。

套接字到 XTI/TLI 的等效项

下表显示了 XTI/TLI 接口和套接字接口之间的近似等效项。注释字段介绍不同之处。如果注释列为空，则这些接口类似或不存在等效接口。

表 9–8 TLI 和套接字等效功能

TLI 接口	套接字接口	注释
t_open(3NSL)	socket(3SOCKET)	
–	socketpair(3SOCKET)	
t_bind(3NSL)	bind(3SOCKET)	t_bind(3NSL) 可设置被动套接字的队列深度，但 bind(3SOCKET) 则不会这样做。对于套接字，在 listen(3SOCKET) 的调用中指定队列长度。

表 9–8 TLI 和套接字等效功能 （续）

TLI 接口	套接字接口	注释
t_optmgmt(3NSL)	getsockopt(3SOCKET) setsockopt(3SOCKET)	t_optmgmt(3NSL) 只管理传输选项。getsockopt(3SOCKET) 和 setsockopt(3SOCKET) 可以管理传输层的选项，也可管理套接字层和任意协议层的选项。
t_unbind(3NSL)	–	
t_close(3NSL)	close(2)	
t_getinfo(3NSL)	getsockopt(3SOCKET)	t_getinfo(3NSL) 返回有关传输的信息。getsockopt(3SOCKET) 可以返回有关传输和套接字的信息。
t_getstate(3NSL)	-	
t_sync(3NSL)	-	
t_alloc(3NSL)	-	
t_free(3NSL)	-	
t_look(3NSL)	-	带有 SO_ERROR 选项的 getsockopt(3SOCKET) 返回的错误信息类型与 t_look(3NSL)t_look() 返回的相同。
t_error(3NSL)	perror(3C)	
t_connect(3NSL)	connect(3SOCKET)	在调用 connect(3SOCKET) 之前，无需绑定本地端点。在调用 t_connect(3NSL) 之前绑定端点。可以在无连接端点上使用 connect(3SOCKET) 设置数据报的缺省目标地址。可以使用 connect(3SOCKET) 发送数据。
t_rcvconnect(3NSL)	-	
t_listen(3NSL)	listen(3SOCKET)	t_listen(3NSL) 等待连接指示。listen(3SOCKET) 设置队列深度。
t_accept(3NSL)	accept(3SOCKET)	
t_snd(3NSL)	send(3SOCKET)	
	sendto(3SOCKET)	
	sendmsg(3SOCKET)	sendto(3SOCKET) 和 sendmsg(3SOCKET) 在连接模式以及数据报模式下进行操作。
t_rcv(3NSL)	recv(3SOCKET)	
	recvfrom(3SOCKET)	

表 9–8 TLI 和套接字等效功能 （续）

TLI 接口	套接字接口	注释
	recvmsg(3SOCKET)	recvfrom(3SOCKET) 和 recvmsg(3SOCKET) 在连接模式以及数据报模式下进行操作。
t_snddis(3NSL)	-	
t_rcvdis(3NSL)	-	
t_sndrel(3NSL)	shutdown(3SOCKET)	
t_rcvrel(3NSL)	-	
t_sndudata(3NSL)	sendto(3SOCKET)	
	recvmsg(3SOCKET)	
t_rcvuderr(3NSL)	-	
read(2), write(2)	read(2), write(2)	在 XTI/TLI 中，必须在调用 read(2) 或 write(2) 之前推送 tirdwr(7M) 模块。在套接字中，调用 read(2) 或 write(2) 便足够。

XTI 接口的附加功能

XNS 5 (UNIX03) 标准引入了一些新的 XTI 接口。下面对其进行简要介绍。可以在相关手册页中找到详细信息。TLI 用户不能使用这些接口。分散/集中数据传输接口为：

t_sndvudata(3NSL)	从一个或多个非连续缓冲区发送数据单元
t_rcvvudata(3NSL)	将数据单元接收到一个或多个非连续缓冲区
t_sndv(3NSL)	通过连接发送来自一个或多个非连续缓冲区的数据或加速数据
t_rcvv(3NSL)	通过连接接收数据或加速数据并将该数据放入一个或多个非连续缓冲区

XTI 实用程序接口 t_sysconf(3NSL) 可获取可配置的 XTI 变量。t_sndreldata(3NSL) 接口可启动并响应带有用户数据的顺序释放。t_rcvreldata(3NSL) 可接收包含用户数据的顺序释放指示或确认。

注 – 附加接口 t_sndreldata(3NSL) 和 t_rcvreldata(3NSL) 只与称为最小 OSI 的特定传输一起使用，该特定传输在 Solaris 平台上不可用。这些接口不能与 Internet 传输（TCP 或 UDP）一起使用。

◆ ◆ ◆ 第 10 章

10 包过滤钩子

包过滤钩子接口有助于在内核级别开发增值型网络解决方案，例如，安全（包过滤和防火墙）解决方案和网络地址转换 (Network Address Translation, NAT) 解决方案。

包过滤钩子接口提供了以下功能：

- 每当在一个钩子点上收到包时发出通知
- 每当创建新 IP 实例以支持需要专用 IP 实例的新区域引导时发出通知
- 通过内核访问其他基本网络接口信息（例如，接口名称和地址）
- 拦截回送接口上的包

当包在使用共享 IP 实例的区域之间移动时，通过拦截回送包也可以访问这些包。这是缺省模型。

包过滤钩子接口

包过滤钩子接口包括内核函数和数据类型定义。

包过滤钩子内核函数

包过滤钩子内核函数可从 `misc/neti` 和 `misc/hook` 内核模块导出以支持包过滤。要使用这些函数，请将内核模块与 `-Nmisc/neti` 和 `-Nmisc/hook` 链接在一起，以便内核可以正确装入这些函数。

`hook_alloc`(9F)	分配 `hook_t` 数据结构。
`hook_free`(9F)	释放最初由 `hook_alloc()` 分配的 `hook_t` 结构。
`net_event_notify_register`(9F)	注册在更改指定事件时调用的函数。
`net_event_notify_unregister`(9F)	指示不再需要通过调用指定回调函数来接收有关更改指定事件的通知。

net_getifname(9F)	检索为指定网络接口提供的名称。
net_getlifaddr(9F)	检索每个指定逻辑接口的网络地址信息。
net_getmtu(9F)	检索有关指定网络接口的当前 MTU 的信息。
net_getpmtuenabled(9F)	指示是否已为指定网络协议启用路径 MTU (Path MTU, PMTU) 搜索。
net_hook_register(9F)	添加一个钩子，用于向属于指定网络协议的事件注册回调。
net_hook_unregister(9F)	禁用向 net_hook_register() 注册的回调钩子。
net_inject(9F)	将网络层包传送至内核或网络。
net_inject_alloc(9F)	分配 net_inject_t 结构。
net_inject_free(9F)	释放最初由 net_inject_alloc() 分配的 net_inject_t 结构。
net_instance_alloc(9F)	分配 net_instance_t 结构。
net_instance_free(9F)	释放最初由 net_instance_alloc() 分配的 net_instance_t 结构。
net_instance_notify_register(9F)	注册在指定网络实例中添加新实例或删除实例时要调用的指定函数。
net_instance_notify_unregister(9F)	指示不再需要通过调用指定回调函数来接收有关更改指定实例的通知。
net_instance_register(9F)	记录在发生与 IP 实例维护相关的事件时要调用的一组函数。
net_instance_unregister(9F)	删除先前向 net_instance_register() 注册的一组实例。
net_ispartialchecksum(9F)	指示指定的包是否包含仅具有部分校验和值的头。
net_isvalidchecksum(9F)	验证指定的包中第 3 层校验和，在某些情况下，还可验证第 4 层校验和。
net_kstat_create(9F)	为指定的 IP 实例分配和初始化新的 kstat(9S) 结构。
net_kstat_delete(9F)	从系统中删除指定 IP 实例的 kstat。
net_lifgetnext(9F)	搜索与物理网络接口相关联的所有逻辑接口。

net_phygetnext(9F)	搜索某个网络协议所“拥有”的所有网络接口。
net_phylookup(9F)	尝试检索某个网络协议的指定接口名称。
net_protocol_lookup(9F)	查找网络层协议的实现。
net_protocol_notify_register(9F)	注册在更改指定协议时要调用的指定函数。
net_protocol_notify_unregister(9F)	从要调用的函数列表中删除指定函数。
net_protocol_release(9F)	指示不再需要对指定网络协议的引用。
net_routeto(9F)	指示要发送的网络接口包。

包过滤钩子数据类型

以下类型支持上述函数。

hook_t(9S)	要插入到联网事件中的回调。
hook_nic_event(9S)	已发生并属于某个网络接口的事件。
hook_pkt_event(9S)	传递到钩子的包事件结构。
net_inject_t(9S)	有关如何传输包的信息。
net_instance_t(9S)	在 IP 中发生相关事件时要调用的实例集合。

使用包过滤钩子接口

由于此 API 支持在同一内核中并发运行多个 IP 栈实例，因此要使用包过滤钩子，需要大量的编程工作。通过 IP 栈，对多个区域使用的多个 IP 栈实例以及多个框架实例均支持在 IP 中进行包拦截。

本节说明了使用包过滤钩子 API 接收入站 IPv4 包的设置代码。

IP 实例

在使用此 API 时，首先需要确定，是允许在内核中运行多个 IP 实例，还是仅与全局区域进行交互。

要了解 IP 实例是否存在，请注册在创建、销毁和关闭实例时要激活的回调函数。可使用 net_instance_alloc() 分配 net_instance_t 包事件结构，以便存储这三种函数指针。当不再需要这些回调和结构时，可使用 net_instance_free() 释放资源。指定

nin_name，以便为结构实例提供一个名称。请至少指定 nin_create() 和 nin_destroy() 回调。在创建新 IP 实例时，会调用 nin_create() 函数，在销毁 IP 实例时，会调用 nin_destroy() 函数。

指定 nin_shutdown() 是可选的，除非代码需要将信息导出到 kstat。要对每个实例使用 kstat，请在执行 create 回调期间使用 net_kstat_create()。必须在 shutdown 回调（而不是 destroy 回调）期间清除 kstat 信息。可使用 net_kstat_delete() 清除 kstat 信息。

```
extern void *mycreate(const netid_t);

net_instance_t *n;

n = net_instance_alloc(NETINFO_VERSION);
if (n != NULL) {
    n->nin_create = mycreate;
    n->nin_destroy = mydestroy;
    n->nin_name = "my module";
    if (net_instance_register(n) != 0)
        net_instance_free(n);
}
```

如果在调用 net_instance_alloc() 时存在一个或多个 IP 实例，则会对当前处于活动状态的每个实例调用 create 回调。支持回调的框架可确保，对于给定的实例，在任意时刻只有 create、destroy 或 shutdown 函数之一处于活动状态。此框架还可确保，一旦调用了 create 回调，则只有在 create 完成之后，才会调用 shutdown 回调。同样，直到 shutdown 回调完成之后，才会启动 destroy 回调。

以下示例中的 mycreate() 函数是一个简单的 create 回调示例。mycreate() 函数可在自己的专用上下文结构中记录网络实例标识符，并注册在向此框架注册新协议（例如，IPv4 或 IPv6）时要调用的新回调。

如果目前没有运行任何区域（此时，除了全局区域之外没有其他任何实例），则调用 net_instance_register() 将对全局区域运行 create 回调。必须提供 destroy 回调，以便日后可以调用 net_instance_unregister()。如果在尝试调用 net_instance_register() 时 nin_create 或 nin_destroy 字段设置为 NULL，则该尝试将失败。

```
void *
mycreate(const netid_t id)
{
    mytype_t *ctx;

    ctx = kmem_alloc(sizeof(*ctx), KM_SLEEP);
    ctx->instance_id = id;
    net_instance_notify_register(id, mynewproto, ctx);
    return (ctx);
}
```

每当在联网实例中添加或删除网络协议时，都应调用 mynewproto() 函数。如果注册的网络协议已在给定实例中运行，则会对每个已有协议调用 create 回调。

协议注册

对于此回调，调用者只会填写 proto 参数。此时，提供的事件名或钩子名均无意义。在此示例函数中，将只查找通告 IPv4 协议注册的事件。

此函数的下一步是，通过使用 net_protocol_notify_register() 接口注册 mynewevent() 函数来搜索向 IPv4 协议添加事件的时间。

```
static int
mynewproto(hook_notify_cmd_t cmd, void *arg, const char *proto,
    const char *event, const char *hook)
{
    mytype_t *ctx = arg;

    if (strcmp(proto, NHF_INET) != 0)
        return (0);

    switch (cmd) {
        case HN_REGISTER :
            ctx->inet = net_protocol_lookup(s->id, proto);
            net_protocol_notify_register(s->inet, mynewevent, ctx);
            break;
        case HN_UNREGISTER :
        case HN_NONE :
            break;
    }
    return (0);
}
```

下表列出了应能够通过 mynewproto() 回调查看到的所有三种协议。将来可能会添加新协议，因此，必须安全地舍弃所有未知协议（即，返回值为 0）。

编程符号	协议
NHF_INET	IPv4
NHF_INET6	IPv6
NHF_ARP	ARP

事件注册

由于对实例和协议的处理是动态的，因此，对每个协议下活动事件的处理也是动态的。此 API 支持两类事件：网络接口事件和包事件。

以下函数将检查是否已对 IPv4 入站包发出事件存在通告。一旦发现此通告，就会分配 hook_t 结构，该结构说明了要对每个 IPv4 入站包调用的函数。

```
static int
mynewevent(hook_notify_cmd_t cmd, void *arg, const char *parent,
    const char *event, const char *hook)
{
    mytype_t *ctx = arg;
    char buffer[32];
    hook_t *h;

    if ((strcmp(event, NH_PHYSICAL_IN) == 0) &&
        (strcmp(parent, NHF_INET) == 0)) {
            sprintf(buffer, "mypkthook_%s_%s", parent, event);
            h = hook_alloc(HOOK_VERSION);
            h->h_hint = HH_NONE;
            h->h_arg = s;
            h->h_name = strdup(buffer);
            h->h_func = mypkthook;
            s->hook_in = h;
            net_hook_register(ctx->inet, (char *)event, h);
    } else {
            h = NULL;
    }
    return (0);
}
```

对于添加和删除的每个事件，会调用 `mynewevent()` 函数。以下事件可用。

事件名称	数据结构	注释
NH_PHYSICAL_IN	hook_pkt_event_t	对于已到达网络协议且已从网络接口驱动程序收到的每个包，将生成此事件。
NH_PHYSICAL_OUT	hook_pkt_event_t	在将每个包传送到网络接口驱动程序以便从网络协议层发送之前，将生成此事件。
NH_FORWARDING	hook_pkt_event_t	对于系统已收到并要发送到其他网络接口的所有包，将生成此事件。此事件发生在 NH_PHYSICAL_IN 之后和 NH_PHYSICAL_OUT 之前。
NH_LOOPBACK_IN	hook_pkt_event_t	对于已在回送接口上收到或已由与全局区域共享网络实例的区域收到的包，将生成此事件。
NH_LOOPBACK_OUT	hook_pkt_event_t	对于已在回送接口上发送或将由与全局区域共享网络实例的区域发送的包，将生成此事件。
NH_NIC_EVENTS	hook_nic_event_t	在网络接口状态发生特定更改时，将生成此事件。

对于包事件，将对 IP 栈中每个特定点生成一个特定事件。这样可以使您准确地选择要在包流中的哪个位置拦截包，而不会花费过多的精力去检查在内核中发生的每个包事件。而对于网络接口事件，则模型与此不同，部分原因是，事件数量比较少，而且很可能开发者会关注其中若干个事件，而不只是关注一个事件。

网络接口事件会通告以下事件之一：

- 创建接口 (NE_PLUMB) 或销毁接口 (NE_UNPLUMB)。
- 接口状态更改为打开 (NE_UP) 或关闭 (NE_DOWN)。
- 接口地址发生更改 (NE_ADDRESS_CHANGE)。

将来可能会添加新网络接口事件，因此，对于回调函数收到的任何未知或无法识别的事件，必须始终返回 0。

包钩子

收到包时，会调用包钩子函数。此时，对于从物理网络接口到达内核的每个入站包，应调用 `mypkthook()` 函数。通过共享 IP 实例模型或回送接口在区域之间传输的内部生成包将不会显示。

为了说明接受一个包以及允许函数在通常情况下返回丢弃此包的条件之间的区别，以下代码将输出每第 100 个包的源和目标地址，然后丢弃此包，从而使包的丢失率为 1%。

```
static int
mypkthook(hook_event_token_t tok, hook_data_t data, void *arg)
{
    static int counter = 0;
    mytupe_t *ctx = arg;
    hook_pkt_event_t *pkt = (hook_pkt_event_t)data;
    struct ip *ip;
    size_t bytes;

    bytes = msgdsize(pkt->hpe_mb);

    ip = (struct ip *)pkt->hpe_hdr;

    counter++;
    if (counter == 100) {
        printf("drop %d bytes received from %x to %x\n", bytes,
            ntohl(ip->ip_src.s_addr), ntohl(ip->ip_dst.s_addr));
        counter = 0;
        freemsg(*pkt->hpe_mp);
        *pkt->hpe_mp = NULL;
        pkt->hpe_mb = NULL;
        pkt->hpe_hdr = NULL;
        return (1);
    }
    return (0);
}
```

通过此函数以及作为回调从包事件调用的所有其他函数收到的包将一次接收一个。通过此接口不会将包链接在一起，因此，每个调用将仅处理一个包，并且 `b_next` 将始终为 NULL。如果不存在其他包，则一个包可能包含通过 `b_cont` 链接在一起的多个 `mblk_t` 结构。

包过滤钩子示例

下面是一个完整示例，可以编译并装入到内核中。

可以使用以下命令将此代码编译到正在 64 位系统上运行的内核模块中。

```
# gcc -D_KERNEL -m64 -c full.c
# ld -dy -Nmisc/neti -Nmisc/hook -r full.o -o full
```

示例 10–1 包过滤钩子示例程序

```
/*
 * This file is a test module written to test the netinfo APIs in OpenSolaris.
 * It is being published to demonstrate how the APIs can be used.
 */
#include <sys/param.h>
#include <sys/sunddi.h>
#include <sys/modctl.h>
#include <sys/ddi.h>
#include "neti.h"

/*
 * Module linkage information for the kernel.
 */
static struct modldrv modlmisc = {
        &mod_miscops,           /* drv_modops */
        "neti test module",     /* drv_linkinfo */
};

static struct modlinkage modlinkage = {
        MODREV_1,               /* ml_rev */
        &modlmisc,              /* ml_linkage */
        NULL
};

typedef struct scratch_s {
        int             sentinel_1;
        netid_t         id;
        int             sentinel_2;
        int             event_notify;
        int             sentinel_3;
        int             v4_event_notify;
        int             sentinel_4;
        int             v6_event_notify;
        int             sentinel_5;
        int             arp_event_notify;
        int             sentinel_6;
        int             v4_hook_notify;
        int             sentinel_7;
        int             v6_hook_notify;
        int             sentinel_8;
        int             arp_hook_notify;
        int             sentinel_9;
        hook_t          *v4_h_in;
        int             sentinel_10;
        hook_t          *v6_h_in;
        int             sentinel_11;
```

示例 10–1 包过滤钩子示例程序 （续）

```
        hook_t          *arp_h_in;
        int             sentinel_12;
        net_handle_t    v4;
        int             sentinel_13;
        net_handle_t    v6;
        int             sentinel_14;
        net_handle_t    arp;
        int             sentinel_15;
} scratch_t;

#define MAX_RECALL_DOLOG        10000
char    recall_myname[10];
net_instance_t *recall_global;
int     recall_inited = 0;
int     recall_doing[MAX_RECALL_DOLOG];
int     recall_doidx = 0;
kmutex_t        recall_lock;
int     recall_continue = 1;
timeout_id_t    recall_timeout;
int     recall_steps = 0;
int     recall_alloced = 0;
void    *recall_alloclog[MAX_RECALL_DOLOG];
int     recall_freed = 0;
void    *recall_freelog[MAX_RECALL_DOLOG];

static int recall_init(void);
static void recall_fini(void);
static void *recall_create(const netid_t id);
static void recall_shutdown(const netid_t id, void *arg);
static void recall_destroy(const netid_t id, void *arg);
static int recall_newproto(hook_notify_cmd_t cmd, void *arg,
    const char *parent, const char *event, const char *hook);
static int recall_newevent(hook_notify_cmd_t cmd, void *arg,
    const char *parent, const char *event, const char *hook);
static int recall_newhook(hook_notify_cmd_t cmd, void *arg,
    const char *parent, const char *event, const char *hook);
static void recall_expire(void *arg);

static void recall_strfree(char *);
static char *recall_strdup(char *, int);

static void
recall_add_do(int mydo)
{
        mutex_enter(&recall_lock);
        recall_doing[recall_doidx] = mydo;
        recall_doidx++;
        recall_steps++;
        if ((recall_steps % 1000000) == 0)
                printf("stamp %d %d\n", recall_steps, recall_doidx);
        if (recall_doidx == MAX_RECALL_DOLOG)
                recall_doidx = 0;
        mutex_exit(&recall_lock);
}

static void *recall_alloc(size_t len, int wait)
```

示例 10–1 包过滤钩子示例程序 （续）

```
{
        int i;

        mutex_enter(&recall_lock);
        i = recall_alloced++;
        if (recall_alloced == MAX_RECALL_DOLOG)
                recall_alloced = 0;
        mutex_exit(&recall_lock);

        recall_alloclog[i] = kmem_alloc(len, wait);
        return recall_alloclog[i];
}

static void recall_free(void *ptr, size_t len)
{
        int i;

        mutex_enter(&recall_lock);
        i = recall_freed++;
        if (recall_freed == MAX_RECALL_DOLOG)
                recall_freed = 0;
        mutex_exit(&recall_lock);

        recall_freelog[i] = ptr;
        kmem_free(ptr, len);
}

static void recall_assert(scratch_t *s)
{
        ASSERT(s->sentinel_1 == 0);
        ASSERT(s->sentinel_2 == 0);
        ASSERT(s->sentinel_3 == 0);
        ASSERT(s->sentinel_4 == 0);
        ASSERT(s->sentinel_5 == 0);
        ASSERT(s->sentinel_6 == 0);
        ASSERT(s->sentinel_7 == 0);
        ASSERT(s->sentinel_8 == 0);
        ASSERT(s->sentinel_9 == 0);
        ASSERT(s->sentinel_10 == 0);
        ASSERT(s->sentinel_11 == 0);
        ASSERT(s->sentinel_12 == 0);
        ASSERT(s->sentinel_13 == 0);
        ASSERT(s->sentinel_14 == 0);
        ASSERT(s->sentinel_15 == 0);
}

int
_init(void)
{
        int error;

        bzero(recall_doing, sizeof(recall_doing));
        mutex_init(&recall_lock, NULL, MUTEX_DRIVER, NULL);

        error = recall_init();
        if (error == DDI_SUCCESS) {
```

示例 10–1 包过滤钩子示例程序 （续）

```
                error = mod_install(&modlinkage);
                if (error != 0)
                        recall_fini();
        }

        recall_timeout = timeout(recall_expire, NULL, drv_usectohz(500000));

        return (error);
}

int
_fini(void)
{
        int error;

        recall_continue = 0;
        if (recall_timeout != NULL) {
                untimeout(recall_timeout);
                recall_timeout = NULL;
        }
        error = mod_remove(&modlinkage);
        if (error == 0) {
                recall_fini();
                delay(drv_usectohz(500000));    /* .5 seconds */

                mutex_destroy(&recall_lock);

                ASSERT(recall_inited == 0);
        }

        return (error);
}

int
_info(struct modinfo *info)
{
        return(0);
}

static int
recall_init()
{
        recall_global = net_instance_alloc(NETINFO_VERSION);

        strcpy(recall_myname, "full_");
        bcopy(((char *)&recall_global) + 4, recall_myname + 5, 4);
        recall_myname[5] = (recall_myname[5] & 0x7f) | 0x20;
        recall_myname[6] = (recall_myname[6] & 0x7f) | 0x20;
        recall_myname[7] = (recall_myname[7] & 0x7f) | 0x20;
        recall_myname[8] = (recall_myname[8] & 0x7f) | 0x20;
        recall_myname[9] = '\0';

        recall_global->nin_create = recall_create;
        recall_global->nin_shutdown = recall_shutdown;
        recall_global->nin_destroy = recall_destroy;
        recall_global->nin_name = recall_myname;
```

示例 10–1 包过滤钩子示例程序 （续）

```
        if (net_instance_register(recall_global) != 0)
                return (DDI_FAILURE);

        return (DDI_SUCCESS);
}

static void
recall_fini()
{

        if (recall_global != NULL) {
                net_instance_unregister(recall_global);
                net_instance_free(recall_global);
                recall_global = NULL;
        }
}

static void
recall_expire(void *arg)
{

        if (!recall_continue)
                return;

        recall_fini();

        if (!recall_continue)
                return;

        delay(drv_usectohz(5000));      /* .005 seconds */

        if (!recall_continue)
                return;

        if (recall_init() == DDI_SUCCESS)
                recall_timeout = timeout(recall_expire, NULL,
                    drv_usectohz(5000));        /* .005 seconds */
}

static void *
recall_create(const netid_t id)
{
        scratch_t *s = kmem_zalloc(sizeof(*s), KM_SLEEP);

        if (s == NULL)
                return (NULL);

        recall_inited++;

        s->id = id;

        net_instance_notify_register(id, recall_newproto, s);

        return s;
}
```

示例 10–1 包过滤钩子示例程序 （续）

```
static void
recall_shutdown(const netid_t id, void *arg)
{
        scratch_t *s = arg;

        ASSERT(s != NULL);
        recall_add_do(__LINE__);
        net_instance_notify_unregister(id, recall_newproto);

        if (s->v4 != NULL) {
                if (s->v4_h_in != NULL) {
                        net_hook_unregister(s->v4, NH_PHYSICAL_IN,
                            s->v4_h_in);
                        recall_strfree(s->v4_h_in->h_name);
                        hook_free(s->v4_h_in);
                        s->v4_h_in = NULL;
                }
                if (net_protocol_notify_unregister(s->v4, recall_newevent))
                        cmn_err(CE_WARN,
                            "v4:net_protocol_notify_unregister(%p) failed",
                            s->v4);
                net_protocol_release(s->v4);
                s->v4 = NULL;
        }

        if (s->v6 != NULL) {
                if (s->v6_h_in != NULL) {
                        net_hook_unregister(s->v6, NH_PHYSICAL_IN,
                            s->v6_h_in);
                        recall_strfree(s->v6_h_in->h_name);
                        hook_free(s->v6_h_in);
                        s->v6_h_in = NULL;
                }
                if (net_protocol_notify_unregister(s->v6, recall_newevent))
                        cmn_err(CE_WARN,
                            "v6:net_protocol_notify_unregister(%p) failed",
                            s->v6);
                net_protocol_release(s->v6);
                s->v6 = NULL;
        }

        if (s->arp != NULL) {
                if (s->arp_h_in != NULL) {
                        net_hook_unregister(s->arp, NH_PHYSICAL_IN,
                            s->arp_h_in);
                        recall_strfree(s->arp_h_in->h_name);
                        hook_free(s->arp_h_in);
                        s->arp_h_in = NULL;
                }
                if (net_protocol_notify_unregister(s->arp, recall_newevent))
                        cmn_err(CE_WARN,
                            "arp:net_protocol_notify_unregister(%p) failed",
                            s->arp);
                net_protocol_release(s->arp);
                s->arp = NULL;
```

示例 10–1 包过滤钩子示例程序 （续）

```
        }
}

static void
recall_destroy(const netid_t id, void *arg)
{
        scratch_t *s = arg;

        ASSERT(s != NULL);

        recall_assert(s);

        ASSERT(s->v4 == NULL);
        ASSERT(s->v6 == NULL);
        ASSERT(s->arp == NULL);
        ASSERT(s->v4_h_in == NULL);
        ASSERT(s->v6_h_in == NULL);
        ASSERT(s->arp_h_in == NULL);
        kmem_free(s, sizeof(*s));

        ASSERT(recall_inited > 0);
        recall_inited--;
}

static int
recall_newproto(hook_notify_cmd_t cmd, void *arg, const char *parent,
    const char *event, const char *hook)
{
        scratch_t *s = arg;

        s->event_notify++;

        recall_assert(s);

        switch (cmd) {
        case HN_REGISTER :
                if (strcmp(parent, NHF_INET) == 0) {
                        s->v4 = net_protocol_lookup(s->id, parent);
                        net_protocol_notify_register(s->v4, recall_newevent, s);
                } else if (strcmp(parent, NHF_INET6) == 0) {
                        s->v6 = net_protocol_lookup(s->id, parent);
                        net_protocol_notify_register(s->v6, recall_newevent, s);
                } else if (strcmp(parent, NHF_ARP) == 0) {
                        s->arp = net_protocol_lookup(s->id, parent);
                        net_protocol_notify_register(s->arp,recall_newevent, s);
                }
                break;

        case HN_UNREGISTER :
        case HN_NONE :
                break;
        }

        return 0;
}
```

示例 10–1 包过滤钩子示例程序 （续）

```
static int
recall_do_event(hook_event_token_t tok, hook_data_t data, void *ctx)
{
        scratch_t *s = ctx;

        recall_assert(s);

        return (0);
}

static int
recall_newevent(hook_notify_cmd_t cmd, void *arg, const char *parent,
    const char *event, const char *hook)
{
        scratch_t *s = arg;
        char buffer[32];
        hook_t *h;

        recall_assert(s);

        if (strcmp(event, NH_PHYSICAL_IN) == 0) {

                sprintf(buffer, "%s_%s_%s", recall_myname, parent, event);
                h = hook_alloc(HOOK_VERSION);
                h->h_hint = HH_NONE;
                h->h_arg = s;
                h->h_name = recall_strdup(buffer, KM_SLEEP);
                h->h_func = recall_do_event;
        } else {
                h = NULL;
        }

        if (strcmp(parent, NHF_INET) == 0) {
                s->v4_event_notify++;
                if (h != NULL) {
                        s->v4_h_in = h;
                        net_hook_register(s->v4, (char *)event, h);
                }
                net_event_notify_register(s->v4, (char *)event,
                    recall_newhook, s);

        } else if (strcmp(parent, NHF_INET6) == 0) {
                s->v6_event_notify++;
                if (h != NULL) {
                        s->v6_h_in = h;
                        net_hook_register(s->v6, (char *)event, h);
                }
                net_event_notify_register(s->v6, (char *)event,
                    recall_newhook, s);

        } else if (strcmp(parent, NHF_ARP) == 0) {
                s->arp_event_notify++;
                if (h != NULL) {
                        s->arp_h_in = h;
                        net_hook_register(s->arp, (char *)event, h);
                }
```

示例 10–1 包过滤钩子示例程序 （续）

```
                net_event_notify_register(s->arp, (char *)event,
                    recall_newhook, s);
        }
        recall_assert(s);

        return (0);
}

static int
recall_newhook(hook_notify_cmd_t cmd, void *arg, const char *parent,
    const char *event, const char *hook)
{
        scratch_t *s = arg;

        recall_assert(s);

        if (strcmp(parent, NHF_INET) == 0) {
                s->v4_hook_notify++;

        } else if (strcmp(parent, NHF_INET6) == 0) {
                s->v6_hook_notify++;

        } else if (strcmp(parent, NHF_ARP) == 0) {
                s->arp_hook_notify++;
        }
        recall_assert(s);

        return (0);
}

static void recall_strfree(char *str)
{
        int len;

        if (str != NULL) {
                len = strlen(str);
                recall_free(str, len + 1);
        }
}

static char* recall_strdup(char *str, int wait)
{
        char *newstr;
        int len;

        len = strlen(str);
        newstr = recall_alloc(len, wait);
        if (newstr != NULL)
                strcpy(newstr, str);

        return (newstr);
}
```

示例 10–2 net_inject 示例程序

```
* Copyright (c) 2012, Oracle and/or its affiliates.
 All rights reserved.
```

示例 10–2 net_inject 示例程序 （续）

```
 */

* PAMP driver - Ping Amplifier enables Solaris to send two ICMP echo
* responses for every ICMP request.
* This example provides a test module of the Oracle Solaris PF-hooks
* (netinfo(9f)) API.This example discovers ICMP echo
* implementation by intercepting inbound packets using
* physical-in‘ event hook.
* If the intercepted packet happens to be a ICMPv4 echo request,
* the module will generate a corresponding ICMP echo response
* which will then be sent to the network interface card using
* the net_inject(9f) function. The original ICMPv4 echo request will be
* allowed to enter the the IP stack so that the request can be
* processed by the destination IP stack.
* The destination stack in turn will send its own ICMPv4 echo response.
* Therefore there will be two ICMPv4 echo responses for a single
* ICMPv4 echo request.

*
* The following example code demonstrates two key functions of netinfo(9f) API:
*
* Packet Interception
*
* Packet Injection
*
* In order to be able to talk to netinfo(9f), the driver must allocate and
* register its own net_instance_t - ‘pamp_ninst‘. This happens in the
* pamp_attach() function, which imlements ‘ddi_attach‘ driver operation.The
* net_instance_t registers three callbacks with netinfo(9f) module:
*     _create
*    _shutdown
*    _destroy
* The netinfo(9f) command uses these functions to request the driver to
* create, shutdown, or destroy the driver context bound to a particular IP instance.
* This will enable the driver to handle packets for every IP stack found in
* the Oracle Solaris kernel. For purposes of this example, the driver is always
* implicitly bound to every IP instance.
*/

/* Use the following makefile to build the driver::
/* Begin Makefile */
ALL = pamp_drv pamp_drv.conf

pamp_drv = pamp_drv.o

pamp_drv.conf: pamp_drv
echo ’name="pamp_drv" parent="pseudo" instance=0;’ > pamp_drv.conf

pamp_drv: pamp_drv.o
ld -dy -r -Ndrv/ip -Nmisc/neti -Nmsic/hook -o pamp_drv pamp_drv.o
pamp_drv.o: pamp_drv.c
cc -m64 -xmodel=kernel -D_KERNEL -c -o $@ $<

install:
cp pamp_drv /usr/kernel/drv/‘isainfo -k‘/pamp_drv
cp pamp_drv.conf /usr/kernel/drv/pamp_drv.conf
```

示例 10–2 net_inject 示例程序 （续）

```
uninstall:
rm -rf /usr/kernel/drv/‘isainfo -k‘/pamp_drv
rm -rf /usr/kernel/drv/pamp_drv.conf

clean:
    rm -f pamp_drv.o pamp_drv pamp_drv.conf

*End Makefile */

 *
* The Makefile shown above will build a pamp_drv driver binary
* and pamp_drv.conf file for driver configuration. If you are
* building on a test machine, use ‘make install‘ to place
* driver and configuration files in the specified location.
* Otherwise copy the pamp_drv binary and the pamp_drv.conf
files to your test machine manually.
*
* Run the following command to load the driver to kernel:

    add_drv pam_drv
* Run the following command to unload the driver to kernel:

    rem_drv pamp_drv
*
* To check if your driver is working you need to use a snoop
* and ‘ping‘ which will be running
* on a remote host. Start snoop on your network interface:

    snoop -d netX icmp

 * Run a ping on a remote host:

ping -ns <test.box>
* test.box refers to the system where the driver is installed.

*
* The snoop should show there are two ICMP echo replies for every ICMP echo
 * request. The expected output should be similar to the snoop output shown below:
 * 172.16.1.2 -> 172.16.1.100 ICMP Echo request (ID: 16652 Sequence number: 0)
 * 172.16.1.100 -> 172.16.1.2   ICMP Echo reply (ID: 16652 Sequence number: 0)
 * 172.16.1.100 -> 172.16.1.2   ICMP Echo reply (ID: 16652 Sequence number: 0)
 * 172.16.1.2 -> 172.16.1.100 ICMP Echo request (ID: 16652 Sequence number: 1)
 * 172.16.1.100 -> 172.16.1.2   ICMP Echo reply (ID: 16652 Sequence number: 1)
 * 172.16.1.100 -> 172.16.1.2   ICMP Echo reply (ID: 16652 Sequence number: 1)
 * 172.16.1.2 -> 172.16.1.100 ICMP Echo request (ID: 16652 Sequence number: 2)
 * 172.16.1.100 -> 172.16.1.2   ICMP Echo reply (ID: 16652 Sequence number: 2)
 * 172.16.1.100 -> 172.16.1.2   ICMP Echo reply (ID: 16652 Sequence number: 2)
 */
#include <sys/atomic.h>
#include <sys/ksynch.h>
#include <sys/ddi.h>
#include <sys/modctl.h>
#include <sys/random.h>
#include <sys/sunddi.h>
#include <sys/stream.h>
```

示例 10–2 net_inject 示例程序 （续）

```
#include <sys/devops.h>
#include <sys/stat.h>
#include <sys/modctl.h>
#include <sys/neti.h>
#include <sys/hook.h>
#include <sys/hook_event.h>
#include <sys/synch.h>
#include <inet/ip.h>
#include <netinet/in_systm.h>
#include <netinet/in.h>
#include <netinet/ip.h
#include <netinet/ip_icmp.h>


/*
 * This is a context for the driver. The context is allocated by
 * pamp_nin_create() callback for every IP instance found in kernel.
 */
typedef struct pamp_ipstack
{
    hook_t *pamp_phyin;
    int pamp_hook_ok;
    net_handle_t    pamp_ipv4;
} pamp_ipstack_t;
static kmutex_t    pamp_stcksmx;
/*
 * The netinstance, which passes driver callbacks to netinfo module.
 */
static net_instance_t    *pamp_ninst = NULL;
/*
 * Solaris kernel driver APIs.
 */
static int pamp_getinfo(dev_info_t *, ddi_info_cmd_t, void *, void **);
static int pamp_attach(dev_info_t *, ddi_attach_cmd_t);
static int pamp_detach(dev_info_t *, ddi_detach_cmd_t);static dev_info_t    *pamp_dev_info = NULL;
/*
 * Driver does not support any device operations.
 */

extern struct cb_ops no_cb_ops;

static struct dev_ops pamp_ops = {
    DEVO_REV,
    0,
    pamp_getinfo,
    nulldev,
    nulldev,
    pamp_attach,
    pamp_detach,
    nodev,
 &no_cb_ops,
    NULL,
    NULL,
};

static struct modldrv    pamp_module = {
```

示例 10–2 net_inject 示例程序 （续）

```
&mod_driverops,
    "ECHO_1",
    &pamp_ops
};
static struct modlinkage pamp_modlink = {
    MODREV_1,
    &pamp_module,
    NULL
};

/*
 * Netinfo stack instance create/destroy/shutdown routines.
 */
static void *pamp_nin_create(const netid_t);
static void pamp_nin_destroy(const netid_t, void *);
static void pamp_nin_shutdown(const netid_t, void *);

/*
 * Callback to process intercepted packets delivered by hook event
 */
static int pamp_pkt_in(hook_event_token_t, hook_data_t, void *);

/*
 * Kernel driver getinfo operation
 */
static int
pamp_getinfo(dev_info_t *dip, ddi_info_cmd_t cmd, void * arg, void **resultp)
{
    int    e;

    switch (cmd) {
        case DDI_INFO_DEVT2DEVINFO:
            *resultp = pamp_dev_info;
            e = DDI_SUCCESS;
            break;
        case DDI_INFO_DEVT2INSTANCE:
            *resultp = NULL;
            e = DDI_SUCCESS;
            break;
        default:
            e = DDI_FAILURE;
    }

    return (e);
}
/*
 * Kernel driver attach operation. The job of the driver is to create a net
 * instance for our driver and register it with netinfo(9f)
 */
static int pamp_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    int    rc;
#define    RETURN(_x_)
    do    {
        mutex_exit(&pamp_stcksmx);
        return (_x_);
```

示例 10–2 net_inject 示例程序 （续）

```
} while (0)

/*
 * Fail for all commands except DDI_ATTACH.
 */
if (cmd != DDI_ATTACH) {
    return (DDI_FAILURE);
}
mutex_enter(&pamp_stcksmx);
/*
 * It is an error to apply attach operation on a driver which is already
 * attached.
 */
if (pamp_ninst != NULL) {
    RETURN(DDI_FAILURE);
}
/*
 * At most one driver instance is allowed (instance 0).
 */
if (ddi_get_instance(dip) != 0) {
    RETURN(DDI_FAILURE);
}

rc = ddi_create_minor_node(dip, "pamp", S_IFCHR, 0, DDI_PSEUDO, 0);
if (rc != DDI_SUCCESS) {
    ddi_remove_minor_node(dip, NULL);
    RETURN(DDI_FAILURE);
}

/*
 * Create and register pamp net instance.  Note we are assigning
 * callbacks _create, _destroy, _shutdown. These callbacks will ask
 * our driver to create/destroy/shutdown our IP driver instances.
 */
pamp_ninst = net_instance_alloc(NETINFO_VERSION);
if (pamp_ninst == NULL) {
    ddi_remove_minor_node(dip, NULL);
    RETURN(DDI_FAILURE);
}

pamp_ninst->nin_name = "pamp";
pamp_ninst->nin_create = pamp_nin_create;
pamp_ninst->nin_destroy = pamp_nin_destroy;
pamp_ninst->nin_shutdown = pamp_nin_shutdown;
pamp_dev_info = dip;
mutex_exit(&pamp_stcksmx);

/*
 * Although it is not shown in the following example, it is
 * recommended that all mutexes/exclusive locks be released before *
 * calling net_instance_register(9F) to avoid a recursive lock
 * entry.  As soon as pamp_ninst is registered, the
 * net_instance_register(9f) will call pamp_nin_create() callback.
 * The callback will run in the same context as the one in which
 * pamp_attach() is running. If pamp_nin_create() grabs the same
 * lock held already by pamp_attach(), then such a lock is being
```

示例 10–2 net_inject 示例程序 （续）

```
     * operated on recursively.
     */
    (void) net_instance_register(pamp_ninst);

    return (DDI_SUCCESS);
#undef    RETURN
}

/*
 * The detach function will unregister and destroy our driver netinstance. The same rules
 * for exclusive locks/mutexes introduced for attach operation apply to detach.
 * The netinfo will take care to call the shutdown()/destroy() callbacks for
 * every IP stack instance.
 */
static int
pamp_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    pamp_ipstack_t    *pamp_ipstack;
    net_instance_t    *ninst = NULL;

    /*
     * It is an error to apply detach operation on driver, when another
     * detach operation is running (in progress), or when detach operation
     * is complete (pamp_ninst).
     */
    mutex_enter(&pamp_stcksmx);
    if (pamp_ninst == NULL) {
        mutex_exit(&pamp_stcksmx);
        return (DDI_FAILURE);
    }

    ninst = pamp_ninst;
    pamp_ninst = NULL;
    mutex_exit(&pamp_stcksmx);

    /*
     * Calling net_instance_unregister(9f) will invoke pamp_nin_destroy()
     * for every pamp_ipstack instance created so far.  Therefore it is advisable
     * to not hold any mutexes, because it might get grabbed by pamp_nin_destroy() function.
     */
    net_instance_unregister(ninst);
    net_instance_free(ninst);

    (void) ddi_get_instance(dip);
    ddi_remove_minor_node(dip, NULL);


    return (DDI_SUCCESS);
}

/*
 * Netinfo callback, which is supposed to create an IP stack context for our
 * ICMP echo server.
 *
 * NOTE: NULL return value is not interpreted as a failure here. The
 * pamp_nin_shutdown()/pamp_nin_destroy() will receive NULL pointer for IP stack
```

示例 10–2 net_inject 示例程序 （续）

```
 * instance with given ‘netid‘ id.
 *
 */
static void *
pamp_nin_create(const netid_t netid)
{
    pamp_ipstack_t    *pamp_ipstack;

    pamp_ipstack = (pamp_ipstack_t *)kmem_zalloc(
        sizeof (pamp_ipstack_t), KM_NOSLEEP);

    if (pamp_ipstack == NULL) {
        return (NULL);
    }

    HOOK_INIT(pamp_ipstack->pamp_phyin, pamp_pkt_in, "pkt_in",
        pamp_ipstack);

    pamp_ipstack->pamp_ipv4 = net_protocol_lookup(netid, NHF_INET);
    if (pamp_ipstack->pamp_ipv4 == NULL) {
        kmem_free(pamp_ipstack, sizeof (pamp_ipstack_t));
        return (NULL);
    }

    pamp_ipstack->pamp_hook_ok = net_hook_register(
        pamp_ipstack->pamp_ipv4, NH_PHYSICAL_IN, pamp_ipstack->pamp_phyin);
    if (pamp_ipstack->pamp_hook_ok != 0) {
        net_protocol_release(pamp_ipstack->pamp_ipv4);
        hook_free(pamp_ipstack->pamp_phyin);
        kmem_free(pamp_ipstack, sizeof (pamp_ipstack_t));
        return (NULL);
    }

    return (pamp_ipstack);
}

/*
 * This event is delivered right before the particular stack instance is
 * destroyed.
 */
static void
pamp_nin_shutdown(const netid_t netid, void *stack)
{
    return;
}

/*
 * Important to note here that the netinfo(9f) module ensures that no
 * no pamp_pkt_in() is "running" when the stack it is bound to is being destroyed.
 */


static void
pamp_nin_destroy(const netid_t netid, void *stack)
{
    pamp_ipstack_t    *pamp_ipstack = (pamp_ipstack_t *)stack;
```

示例 10–2 net_inject 示例程序 （续）

```
    /*
     * Remember stack can be NULL! The pamp_nin_create() function returns
     * NULL on failure. The return value of pamp_nin_create() function will
     * be ‘kept‘ in netinfo module as a driver context for particular IP
     * instance. As soon as the instance is destroyed the NULL value
     * will appear here in pamp_nin_destroy(). Same applies to
     * pamp_nin_shutdown(). Therefore our driver must be able to handle
     * NULL here.
     */
    if (pamp_ipstack == NULL)
        return;

    /*
     * If driver has managed to initialize packet hook, then it has to be
     * unhooked here.
     */
    if (pamp_ipstack->pamp_hook_ok != -1) {
        (void) net_hook_unregister(pamp_ipstack->pamp_ipv4,
            NH_PHYSICAL_IN, pamp_ipstack->pamp_phyin);
        hook_free(pamp_ipstack->pamp_phyin);
        (void) net_protocol_release(pamp_ipstack->pamp_ipv4);
    }

    kmem_free(pamp_ipstack, sizeof (pamp_ipstack_t));
}

/*
 * Packet hook handler
 *
 * Function receives intercepted IPv4 packets coming from NIC to IP stack.  If
 * inbound packet is ICMP ehco request, then function will generate ICMP echo
 * response and use net_inject() to send it to network.  Function will also let
 * ICMP echo request in, so it will be still processed by destination IP stack,
 * which should also generate its own ICMP echo response. The snoop should show
 * you there will be two ICMP echo responses leaving the system where the pamp
 * driver is installed
 */

static int
pamp_pkt_in(hook_event_token_t ev, hook_data_t info, void *arg)
{
    hook_pkt_event_t    *hpe = (hook_pkt_event_t *)info;
    phy_if_t        phyif;
    struct ip        *ip;

    /*
     * Since our pamp_pkt_in callback is hooked to PHYSICAL_IN hook pkt.
     * event only, the physical interface index will always be passed as
     * hpe_ifp member.
     *
     * If our hook processes PHYSICAL_OUT hook pkt event, then
     * the physical interface index will be passed as hpe_ofp member.
     */
     phyif = hpe->hpe_ifp;
```

示例 10–2 net_inject 示例程序 （续）

```
ip = hpe->hpe_hdr;
if (ip->ip_p == IPPROTO_ICMP) {
    mblk_t    *mb;

    /*
     * All packets are copied/placed into a continuous buffer to make
     * parsing easier.
     */
    if ((mb = msgpullup(hpe->hpe_mb, -1)) != NULL) {
        struct icmp    *icmp;
        pamp_ipstack_t    *pamp_ipstack = (pamp_ipstack_t *)arg;

        ip = (struct ip *)mb->b_rptr;
        icmp = (struct icmp *)(mb->b_rptr + IPH_HDR_LENGTH(ip));

        if (icmp->icmp_type == ICMP_ECHO) {
            struct in_addr    addr;
            uint32_t    sum;
            mblk_t    *echo_resp = copymsg(mb);
            net_inject_t    ninj;

            /*
             * We need to make copy of packet, since we are
             * going to turn it into ICMP echo response.
             */
            if (echo_resp == NULL) {
                return (0);
            }
            ip = (struct ip *)echo_resp->b_rptr;
            addr = ip->ip_src;
            ip->ip_src = ip->ip_dst;
            ip->ip_dst = addr;
            icmp = (struct icmp *) (echo_resp->b_rptr + IPH_HDR_LENGTH(ip));
            icmp->icmp_type = ICMP_ECHO_REPLY;
            sum = ~ntohs(icmp->icmp_cksum) & 0xffff;
            sum += (ICMP_ECHO_REQUEST - ICMP_ECHO_REPLY);
            icmp->icmp_cksum =
                htons(~((sum >> 16) + (sum & 0xffff)));

            /*
             * Now we have assembled an ICMP response with
             * correct chksum.  It's time to send it out.
             * We have to initialize command for
             * net_inject(9f) --  ninj.
             */
            ninj.ni_packet = echo_resp;
            ninj.ni_physical = phyif;
            /*
             * As we are going use NI_QUEUE_OUT to send
             * our ICMP response, we don't need to set up
             * .ni_addr, which is required for NI_DIRECT_OUT
             * injection path only. In such case packet
             * bypasses IP stack routing and is pushed
             * directly to physical device queue. Therefore
             * net_inject(9f) requires as to specify
             * next-hop IP address.
```

示例 10–2 net_inject 示例程序 （续）

```
                 *
                 * Using NI_QUEUE_OUT is more convenient for us
                 * since IP stack will take care of routing
                 * process and will find out ‘ni_addr‘
                 * (next-hop) address on its own.
                 */
                (void) net_inject(pamp_ipstack->pamp_ipv4,
                    NI_QUEUE_OUT, &ninj);
            }
        }
    }

    /*
     * 0 as return value will let packet in.
     */
    return (0);
}

/*
 * Kernel module handling.
 */
int init()
{
    mutex_init(&pamp_stcksmx, "pamp_mutex", MUTEX_DRIVER, NULL);
    return (mod_install(&pamp_modlink));
}

int fini()
{
    int rv;

    rv = mod_remove(&pamp_modlink);
    return (rv);
}

int info(struct modinfo *modinfop)
{
    return (mod_info(&pamp_modlink, modinfop));
}
```

◆ ◆ ◆ 第 11 章

11 传输选择和名称到地址映射

本章介绍了如何选择传输和解析网络地址，并进一步介绍了可用于为应用程序指定可用通信协议的接口。此外，还介绍了用于将名称直接映射到网络地址的其他接口。

注 – 在本章中，术语**网络**和**传输**可互换使用。这两个术语指的是符合 OSI 参考模型传输层的编程接口。术语**网络**还用于指代通过某种电子介质连接的多台计算机的物理集合。

传输选择

注意 – 本章中介绍的接口具有多线程安全性。“多线程安全”表示可以在多线程应用程序中随意使用包含传输选择接口调用的应用程序。这些接口调用不提供线性可伸缩性，因为它们不可重复执行。

分布式应用程序必须使用标准的传输服务接口，以便可移植到其他协议。传输选择服务提供了一个接口，应用程序可通过此接口选择要使用的协议。使用此接口，应用程序可独立于协议和介质。

传输选择表示客户机应用程序可轻松尝试每种可用传输，直到该客户机与服务器建立通信为止。利用传输选择，服务器应用程序可接受多个传输请求。然后，应用程序可通过多种协议进行通信。既可按照本地缺省序列指定的顺序尝试传输，也可按照用户指定的顺序尝试传输。

应用程序负责从可用传输中进行选择。传输选择机制可统一并简化选择。

名称到地址映射

通过名称到地址映射，应用程序可以获取指定主机上某一服务的地址，而不管使用何种传输。名称到地址映射包含以下接口：

netdir_getbyname(3NSL)	将主机名和服务名映射为一组地址
netdir_getbyaddr(3NSL)	将地址映射为主机名和服务名
netdir_free(3NSL)	释放由名称到地址的转换例程所分配的结构
taddr2uaddr(3NSL)	转换一个地址，并返回该地址与传输无关的字符表示形式
uaddr2taddr(3NSL)	将通用地址转换为 netbuf 结构
netdir_options(3NSL)	特定于传输的功能（如 TCP 和 UDP 的广播地址和保留端口功能）的接口
netdir_perror(3NSL)	在 stderr 中显示一条消息，说明用来将名称映射到地址的例程之一失败的原因。
netdir_sperror(3NSL)	返回一个字符串，其中包含的错误消息说明了用来将名称映射到地址的例程之一失败的原因。

每个例程的第一个参数均指向用于描述传输的 netconfig(4) 结构。例程使用 netconfig(4) 结构中的目录查找库路径数组来调用每个路径，直到转换成功为止。

表 11–1 介绍了名称到地址库。第 237 页中的 “使用名称到地址映射例程”中介绍的例程在 netdir(3NSL) 手册页中进行了定义。

注 – Solaris 环境中不再存在以下库：tcpip.so、switch.so 和 nis.so。有关此更改的更多信息，请参见 nsswitch.conf(4) 手册页以及 gethostbyname(3NSL) 手册页中的“附注”部分。

表 11–1 名称到地址库

库	传输系列	说明
-	inet	针对协议系列 inet 的网络的名称到地址映射由名称服务转换基于文件 nsswitch.conf(4) 中对应于 *hosts* 和 *services* 的项提供。对于其他系列的网络，短划线表示无法正常使用的名称到地址映射。
straddr.so	loopback	包含在接受字符串作为地址的任何协议中执行名称到地址映射的例程，如回送传输。

straddr.so 库

straddr.so 库的名称到地址转换文件由系统管理员创建。系统管理员还将维护这些转换文件。straddr.so 文件包括 /etc/net/*transport-name*/hosts 和 /etc/net/*transport-name*/services。*transport-name* 是指接受字符串地址的传输的本地名称，该名称在 /etc/netconfig 文件的 *network ID* 字段中指定。例如，ticlts 的主机文件可以是 /etc/net/ticlts/hosts，ticlts 的服务文件可以是 /etc/net/ticlts/services。

大多数字符串地址不区分 *host* 和 *service*。不过，将字符串分隔为主机部分和服务部分这一点与其他传输是一致的。/etc/net/*transport-name*/hosts 文件包含一个文本字符串，该字符串会假定为主机地址，后跟主机名：

```
joyluckaddr      joyluck

carpediemaddr    carpediem

thehopaddr       thehop

pongoaddr        pongo
```

由于回送传输的范围不能超出包含主机，因此列出其他主机毫无意义。

/etc/net/transport-name/services 文件包含后跟标识服务地址的字符串的服务名：

```
rpcbind    rpc
listen    serve
```

例程可通过串联主机地址、句点 (.) 和服务地址来创建全字符串地址。例如，pongo 上的 listen 服务的地址为 pongoaddr.serve。

应用程序请求某个传输上使用该库的特定主机中服务的地址时，/etc/net/*transport*/hosts 中必须包含主机名。/etc/net/*transport*/services 中必须包含服务名。如果缺少任一名称，则名称到地址的转换将失败。

使用名称到地址映射例程

本节概述了可供使用的映射例程。这些例程会返回网络名称或将其转换为各自的网络地址。请注意，netdir_getbyname(3NSL)、netdir_getbyaddr(3NSL) 和 taddr2uaddr(3NSL) 可返回指向必须通过 netdir_free(3NSL) 调用来释放的数据的指针。

```
int netdir_getbyname(struct netconfig *nconf,
        struct nd_hostserv *service, struct nd_addrlist **addrs);
```

netdir_getbyname(3NSL) 可将 *service* 中指定的主机名和服务名映射为一组与 *nconf* 中标识的传输一致的地址。nd_hostserv 和 nd_addrlist 结构在 netdir(3NSL) 手册页中进行了定义。*addrs* 中会返回指向上述地址的指针。

要查找所有可用传输上的主机和服务的所有地址，请使用由 getnetpath(3NSL) 或 getnetconfig(3NSL) 返回的每个 netconfig(4) 结构来调用 netdir_getbyname(3NSL)。

```
int netdir_getbyaddr(struct netconfig *nconf,
        struct nd_hostservlist **service, struct netbuf *netaddr);
```

netdir_getbyaddr(3NSL) 可将地址映射为主机名和服务名。此接口使用 *netaddr* 中的地址进行调用，并会返回 *service* 中主机名和服务名对的列表。nd_hostservlist 结构在 netdir(3NSL) 中进行了定义。

```
void netdir_free(void *ptr, int struct_type);
```

netdir_free(3NSL) 例程会释放名称到地址转换例程所分配的结构。其中的参数可以使用下表显示的值。

表 11–2 netdir_free(3NSL) 例程

struct_type	ptr
ND_HOSTSERV	指向 nd_hostserv 结构的指针
ND_HOSTSERVLIST	指向 nd_hostservlist 结构的指针
ND_ADDR	指向 netbuf 结构的指针
ND_ADDRLIST	指向 nd_addrlist 结构的指针

```
char *taddr2uaddr(struct netconfig *nconf, struct netbuf *addr);
```

taddr2uaddr(3NSL) 可转换 *addr* 指向的地址，并返回该地址与传输无关的字符表示形式。此字符表示形式称为通用地址。*nconf* 中给定的值可指定该地址针对其有效的传输。通用地址可通过 free(3C) 进行释放。

```
struct netbuf *uaddr2taddr(struct netconfig *nconf, char *uaddr);
```

uaddr 所指向的通用地址会转换为 netbuf 结构。*nconf* 可指定该地址针对其有效的传输。

```
int netdir_options(const struct netconfig *config,
          const int option, const int fildes, char *point_to_args);
```

netdir_options(3NSL) 提供了特定于传输的功能（如 TCP 和 UDP 的广播地址和保留端口功能）的接口。*nconf* 的值指定传输，而 *option* 则指定要执行的特定于传输的操作。*option* 中的值可能会导致忽略 *fd* 中的值。第四个参数指向特定于操作的数据。

下表说明了用于 option 的值。

表 11–3 netdir_options 的值

选项	说明
ND_SET_BROADCAST	在传输支持广播时设置广播的传输
ND_SET_RESERVEDPORT	在传输允许的情况下使应用程序绑定到保留端口
ND_CHECK_RESERVEDPORT	在传输支持保留端口的情况下验证地址是否对应于保留端口
ND_MERGEADDR	将本地有意义的地址转换为客户机主机可连接到的地址

netdir_perror(3NSL) 例程会在 stderr 中显示一条消息，说明用来将名称映射到地址的例程之一失败的原因。

```
void netdir_perror(char *s);
```

netdir_sperror(3NSL) 返回一个字符串，其中包含的错误消息说明了用来将名称映射到地址的例程之一失败的原因。

```
char *netdir_sperror(void);
```

以下示例说明了如何执行网络选择和名称到地址映射。

示例 11–1 网络选择和名称到地址映射

```
#include <netconfig.h>
#include <netdir.h>
#include <sys/tiuser.h>

struct nd_hostserv nd_hostserv;   /* host and service information */
struct nd_addrlist *nd_addrlistp; /* addresses for the service */
struct netbuf *netbufp;           /* the address of the service */
struct netconfig *nconf;          /* transport information*/
int i;                            /* the number of addresses */
char *uaddr;                      /* service universal address */
void *handlep;                    /* a handle into network selection */
/*
 * Set the host structure to reference the "date"
 * service on host "gandalf"
 */
nd_hostserv.h_host = "gandalf";
nd_hostserv.h_serv = "date";
/*
 * Initialize the network selection mechanism.
 */
if ((handlep = setnetpath()) == (void *)NULL) {
   nc_perror(argv[0]);
   exit(1);
}
/*
 * Loop through the transport providers.
 */
while ((nconf = getnetpath(handlep)) != (struct netconfig *)NULL)
```

示例 11–1 网络选择和名称到地址映射 （续）

```
{
   /*
    * Print out the information associated with the
    * transport provider described in the "netconfig"
    * structure.
    */
   printf("Transport provider name: %s\n", nconf->nc_netid);
   printf("Transport protocol family: %s\n", nconf->nc_protofmly);
   printf("The transport device file: %s\n", nconf->nc_device);
   printf("Transport provider semantics: ");
       switch (nconf->nc_semantics) {
   case NC_TPI_COTS:
      printf("virtual circuit\n");
      break;
   case NC_TPI_COTS_ORD:
      printf("virtual circuit with orderly release\n");
      break;

   case NC_TPI_CLTS:
      printf("datagram\n");
      break;
   }
   /*
    * Get the address for service "date" on the host
    * named "gandalf" over the transport provider
    * specified in the netconfig structure.
    */
   if (netdir_getbyname(nconf, &nd_hostserv, &nd_addrlistp) != ND_OK) {
      printf("Cannot determine address for service\n");
      netdir_perror(argv[0]);
      continue;
   }
   printf("<%d> addresses of date service on gandalf:\n",
      nd_addrlistp->n_cnt);
   /*
    * Print out all addresses for service "date" on
    * host "gandalf" on current transport provider.
    */
   netbufp = nd_addrlistp->n_addrs;
   for (i = 0; i < nd_addrlistp->n_cnt; i++, netbufp++) {
      uaddr = taddr2uaddr(nconf,netbufp);
      printf("%s\n",uaddr);
      free(uaddr);
   }
       netdir_free( nd_addrlistp, ND_ADDRLIST );

}
endnetconfig(handlep);
```

◆ ◆ ◆ 第 12 章

12 实时编程和管理

本章介绍如何编写和移植要在 SunOS 中运行的实时应用程序。本章是为精通编写实时应用程序的程序员以及熟悉实时处理和 Solaris 系统的管理员编写的。

本章讨论以下主题：

- 实时应用程序的调度需求，相关内容在第 244 页中的 “实时调度程序”中介绍。
- 第 253 页中的 “内存锁定”。
- 第 259 页中的 “异步网络通信”。

实时应用程序的基本规则

满足特定条件时才能保证实时响应。本节介绍了这些条件以及一些比较严重的设计错误。

此处介绍的大多数潜在问题都会延长系统的响应时间。其中一个潜在问题可能会冻结工作站。其他更为隐蔽的错误包括优先级倒置和系统过载。

Solaris 实时进程具有以下特征：

- 在 RT 调度类中运行，如第 244 页中的 “实时调度程序”中所述
- 锁定其进程地址空间中的所有内存，如第 253 页中的 “内存锁定”中所述
- 来自其中所有动态绑定都在早期完成的程序，如第 242 页中的 “共享库”中所述

本章中介绍的实时操作都是针对单线程进程的，但相关说明同样适用于多线程进程。有关多线程进程的详细信息，请参见《多线程编程指南》。要保证实时调度线程，必须将线程创建为绑定线程。另外，线程的 LWP 必须在 RT 调度类中运行。内存锁定和早期动态绑定对于进程中的所有线程都有效。

如果进程是优先级最高的实时进程，则此进程会在变为可运行的保证分发延迟期间内获取处理器。有关更多信息，请参见第 244 页中的 “分发延迟”。只要此进程仍然是优先级最高的可运行进程，它便会继续运行。

实时进程可能会因为其他系统事件而失去对处理器的控制，还可能会因为其他系统事件而无法获取对处理器的控制。这些事件包括：外部事件（如中断）、资源匮乏、等待外部事件（如同步 I/O）以及进程被优先级较高的进程抢占。

实时调度通常不适用于系统初始化和终止服务，例如 open(2) 和 close(2)。

延长响应时间的因素

本节所述的问题会在不同程度上延长系统的响应时间。这种延长可能会非常严重，以至导致应用程序错过临界期限。

实时处理还可能削弱运行实时应用程序的系统上处于活动状态的其他应用程序各个方面的操作。由于实时进程具有较高的优先级，因此可能会长时间阻止分时进程运行。这种现象会导致交互式活动（如显示器和键盘响应时间）明显减慢。

同步 I/O 调用

SunOS 中的系统响应不提供 I/O 事件的时限。这意味着不应在任何执行时间性强的程序段中包含同步 I/O 调用。即使允许具有较长时限的程序段也不能执行同步 I/O。海量存储 I/O 即是这样一个示例，它会导致在执行读写操作时将系统挂起。

常见的应用程序错误是执行 I/O 以从磁盘获取错误消息文本。应该通过独立进程或独立线程以此方式执行 I/O 操作。此类独立进程或独立线程不应该实时运行。

中断服务

中断优先级与进程优先级无关。为一组进程设置的优先级不会由这些进程的操作所导致的硬件服务中断继承。结果，高优先级的实时进程控制的设备不一定具有高优先级的中断处理。

共享库

分时进程可以使用动态链接的共享库节省大量的内存。此类型的链接通过文件映射的形式实施。动态链接的库例程会导致隐式读取。

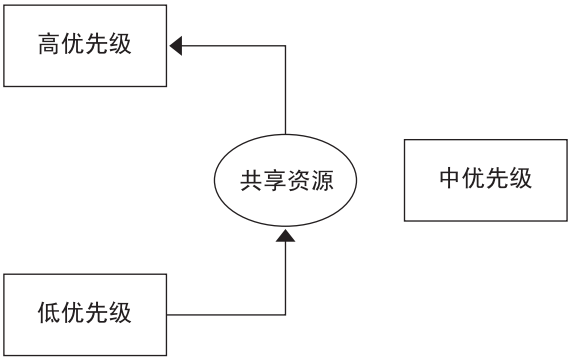
实时程序可以在调用该程序时将环境变量 LD_BIND_NOW 设置为非 NULL 值。通过设置此环境值的值，可以使用共享库，同时避免动态绑定。此过程还强制在程序开始执行之前绑定所有动态链接。有关更多信息，请参见《链接程序和库指南》。

优先级倒置

分时进程可以通过获取实时进程所需的资源来阻塞实时进程。当较高优先级的进程被较低优先级的进程阻塞时，会发生优先级倒置。术语**阻塞**描述某个进程必须等待一个或多个进程放弃对资源的控制的情况。如果该阻塞的时间延长，则实时进程可能会错过其期限。

考虑下图中描述的情况，其中高优先级进程需要共享资源。而较低优先级的进程拥有该资源并由中等优先级的进程抢占，从而阻塞了高优先级进程。其中可涉及任意数量的中间进程。必须执行完所有中间进程以及优先级较低的进程的关键部分。这一系列执行操作可能需要任意长时间。

图 12–1 无限制的优先级倒置



《多线程编程指南》中的“互斥锁属性”中介绍了此问题以及处理此问题的方法。

粘滞锁

页面在其锁定计数达到 65535 (`0xFFFF`) 时会永久锁入内存。值 `0xFFFF` 通过实施定义，并且在将来的发行版本中可能会更改。以此方式锁定的页面无法解除锁定。

失控实时进程

失控实时进程可能会导致系统停止。此类失控进程还可能会大幅减慢系统响应，以致系统看起来好像已停止。

注 – 如果在 SPARC 系统上具有失控进程，请按 Stop-A。您可能必须按 Stop-A 多次。如果按 Stop-A 不起作用，请关闭电源，等待片刻，然后重新打开电源。如果在非 SPARC 系统上具有失控进程，请关闭电源，等待片刻，然后重新打开电源。

如果某个高优先级实时进程不放弃对 CPU 的控制，您必须中断无限循环以便重新获取对系统的控制。此类失控进程对于 Ctrl-C 不响应。尝试在高于失控进程优先级的优先级使用 shell 集不起作用。

异步 I/O 行为

异步 I/O 操作不总是按照操作在内核中排列的顺序执行。异步操作不一定按照执行操作的顺序返回至调用者。

如果为 aioread(3AIO) 调用的快速序列指定了单个缓冲区，则该缓冲区的状态不确定。缓冲区状态的不确定性从首次调用开始，到将最后结果通知给调用者结束。

单个 aio_result_t 结构仅可以用于一个异步操作。操作可以是读取或写入操作。

实时文件

SunOS 不提供工具来确保按照物理连续性分配文件。

对于常规文件，始终缓冲 read(2) 和 write(2) 操作。应用程序可以使用 mmap(2) 和 msync(3C) 实现辅助存储和进程内存之间的直接 I/O 传输。

实时调度程序

实时调度约束是管理数据获取或进程控制硬件所必需的。实时环境要求进程能够在限制的时间内对外部事件做出反应。此类约束可能超出设计用于为一组分时进程公平分发处理资源的内核的功能。

本节介绍 SunOS 实时调度程序、其优先级队列，以及如何使用控制调度的系统调用和实用程序。

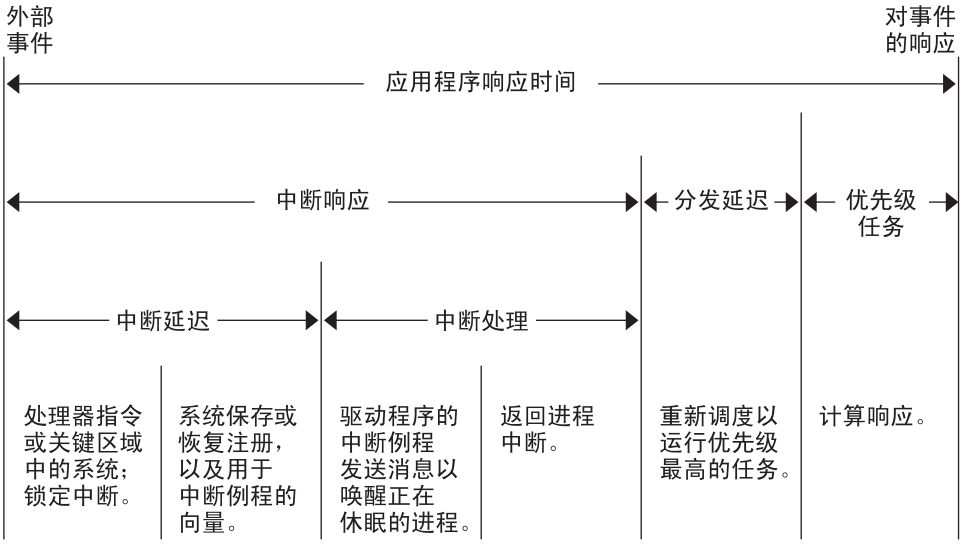
分发延迟

实时应用程序调度行为中最重要的元素是置备实时调度类。标准的分时调度类不适用于实时应用程序，因为此调度类同等对待所有进程。标准的分时调度类具有受限的优先级概念。实时应用程序需要将进程优先级作为绝对优先级的调度类。实时应用程序还需要仅通过显式应用程序操作更改进程优先级的调度类。

术语**分发延迟**描述系统响应请求以使进程开始操作的时间。通过专门为了遵守应用程序优先级而编写的调度程序，可以使用限制的分发延迟开发实时应用程序。

下图展示了应用程序响应外部事件的请求所需的时间。

图 12–2 应用程序响应时间



总体应用程序响应时间包括中断响应时间、分发延迟和应用程序的响应时间。

应用程序的中断响应时间包括系统的中断延迟和设备驱动程序本身的中断处理时间。中断延迟由系统必须在禁用中断的情况下运行的最长间隔决定。在 SunOS 中使用通常不需要提高处理器中断级别的同步原语最大程度地减少此时间。

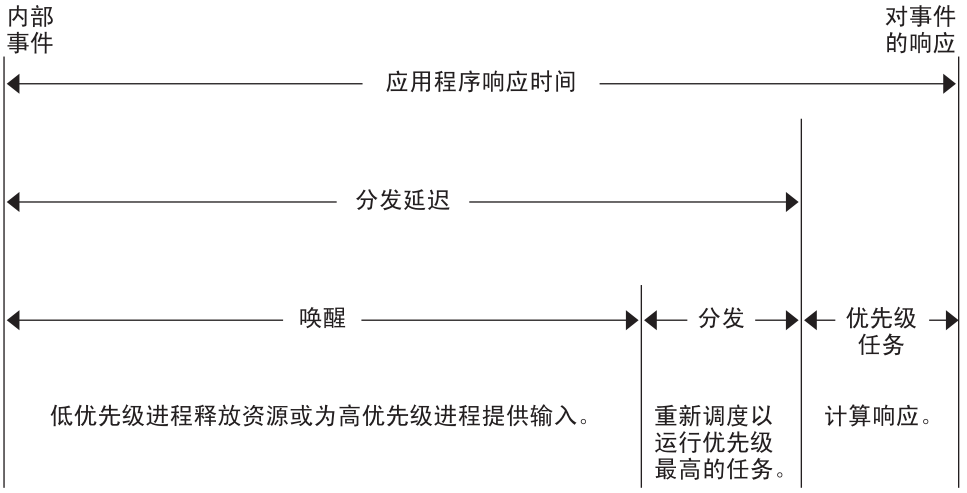
在中断处理期间，驱动程序的中断例程唤醒高优先级进程并在完成后返回。系统检测到某个优先级高于中断进程的进程现已做好分发准备，随即分发该进程。将上下文从较低优先级进程切换到较高优先级进程的时间包括在分发延迟时间中。

图 12–3 展示了系统的内部分发延迟和应用程序响应时间。响应时间定义为系统响应内部事件所需的时间。内部事件的分发延迟表示进程唤醒更高优先级进程所需的时间。分发延迟还包括系统分发更高优先级进程所需的时间。

应用程序响应时间是驱动程序执行以下操作所需的时间：唤醒更高优先级进程、从低优先级进程释放资源、重新调度更高优先级任务、计算响应和分发任务。

在分发延迟间隔期间可能会发生并处理中断。此处理会增加应用程序的响应时间，但是不算入分发延迟。因此，此处理不受分发延迟保证的约束。

图 12–3 内部分发延迟



使用实时 SunOS 随附的新调度技术，系统分发延迟时间在指定的范围之内。如您在下表中所见，分发延迟通过有限数量的进程进行改善。

表 12–1 实时系统分发延迟

工作站	有限数量的进程	任意数量的进程
SPARCstation 2	在少于 16 个活动进程的系统中 < 0.5 毫秒	1.0 毫秒
SPARCstation 5	< 0.3 毫秒	0.3 毫秒

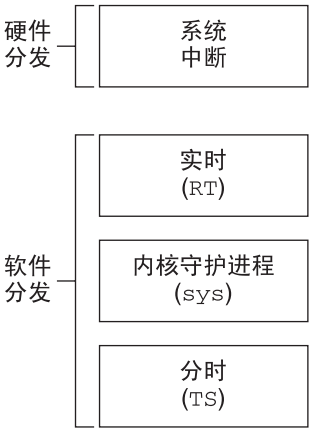
调度类

SunOS 内核按优先级分发进程。调度程序或分发程序支持调度类的概念。类定义为实时 (RT)、系统 (SYS) 和分时 (TS)。每个类都具有唯一的调度策略，用于在其类中分发进程。

内核首先分发最高优先级的进程。缺省情况下，实时进程的优先级高于 sys 和 TS 进程。管理员可以配置系统，以便 TS 进程和 RT 进程的优先级重叠。

下图从 SunOS 内核的视角展示了类的概念。

图 12–4 调度类的分发优先级



不能由软件控制的硬件中断具有最高的优先级。处理中断的例程直接从中断即时分发，而不管当前进程的优先级为何。

实时进程具有最高的缺省软件优先级。 RT 类中的进程具有优先级和**时间量程**值。RT 进程严格根据这些参数进行调度。只要 RT 进程准备运行，就不能运行 SYS 或 TS 进程。固定优先级调度允许重要进程按照预先确定的顺序运行，直到完成。这些优先级永远不会更改，除非由应用程序更改。

RT 类进程继承父项的时间量程，而不管是有限还是无限时间量程。具有有限时间量程的进程一直运行到时间量程到期。如果具有有限时间量程的进程在等待 I/O 事件时阻塞或被更高优先级的可运行实时进程抢占，则该进程也会停止运行。具有无限时间量程的进程仅在进程终止、阻塞或被抢占时才停止执行。

SYS 类用于调度特殊系统进程的执行，例如分页、STREAMS 和交换程序。您无法将进程的类更改为 SYS 类。进程的 SYS 类具有固定的优先级，这些优先级由内核在进程启动时建立。

分时 (TS) 进程的优先级最低。TS 类的进程以动态方式进行调度，每个时间片为数百毫秒。TS 调度程序以循环方式足够频繁地切换上下文，以便让每个进程具有同等的运行机会，具体取决于：

- 时间片值
- 记录进程上次进入睡眠状态的时间的进程历史记录
- CPU 使用率注意事项

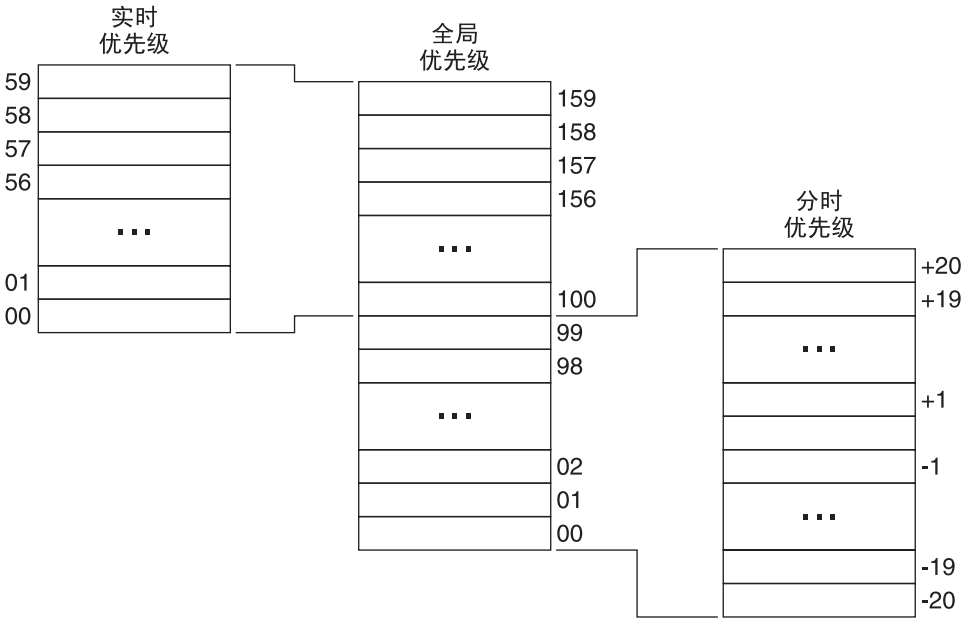
缺省的分时策略为较低优先级的进程提供较长的时间片。

子进程通过 fork(2) 继承父进程的调度类和属性。进程的调度类和属性不会被 exec(2) 更改。

不同的算法分发每个调度类。内核调用与类相关的例程以做出有关 CPU 进程调度的决定。内核与类无关，且执行其队列中最高优先级的进程。每个类负责计算该类的进程优先级值。此值会置入该进程的分发优先级变量中。

如下图所示，每个类算法都有其自己的方法来指定要置入全局运行队列的最高优先级进程。

图 12–5 内核分发队列



每个类都有一组适用于该类中进程的优先级别。特定于类的映射将这些优先级映射到一组全局优先级。一组全局调度优先级映射不需要以零开始或连续。

缺省情况下，分时 (TS) 进程的全局优先级值范围为 -20 到 +20。这些全局优先级值映射到内核中的 0-40，临时分配最高可达 99。实时 (RT) 进程的缺省优先级范围为 0-59，且映射到内核中的 100 到 159。内核的与类无关的代码运行队列中全局优先级最高的进程。

分发队列

分发队列是具有相同全局优先级的进程的线性链接列表。每个进程在调用时都具有附加到该进程的特定于类的信息。进程按照基于进程全局优先级的顺序从内核分发表中分发。

分发进程

分发进程时，进程的上下文与其内存管理信息、寄存器以及栈一起映射到内存。上下文映射完成之后，执行开始。内存管理信息采用硬件寄存器形式，其中包含为当前运行的进程执行虚拟内存转换所需的数据。

进程抢占

当较高优先级的进程变为可分发时，内核中断其计算并强制进行上下文切换，从而抢占当前运行的进程。如果内核发现目前有较高优先级的进程可分发，则可以随时抢占进程。

例如，假定进程 A 从外围设备执行读取操作。内核将进程 A 置入睡眠状态。然后，内核发现较低优先级的进程 B 可运行。进程 B 得到分发并开始执行。最后，外围设备发送中断，并进入设备的驱动程序。设备驱动程序使进程 A 可运行并返回。内核现在阻止 B 的处理，恢复已唤醒进程 A 的执行，而不是返回到中断的进程 B。

当多个进程争用内核资源时，会发生另一种有趣的情况。高优先级的实时进程可能正在等待低优先级进程持有的资源。当低优先级进程释放资源时，内核会抢占该进程以恢复较高优先级进程的执行。

内核优先级倒置

当较高优先级的进程由一个或多个较低优先级进程阻塞很长时间时，会发生优先级倒置。在 SunOS 内核中使用诸如互斥锁等同步原语可能会导致优先级倒置。

当进程必须等待一个或多个进程放弃资源时，该进程处于**阻塞**状态。延长阻塞可能会导致错过期限，即便对于低利用等级也是如此。

针对 SunOS 内核的互斥锁的优先级倒置问题已通过实施基本的优先级继承策略得到解决。策略表明当较低优先级的进程阻塞较高优先级进程的执行时，较低优先级进程继承较高优先级进程的优先级。此继承对进程可以保持阻塞状态的时间设定一个上界。策略是内核行为的属性，而不是程序员通过系统调用或接口执行制定的解决方案。但是，用户级别的进程仍然可能出现优先级倒置。

用户优先级倒置

《多线程编程指南》中的“互斥锁属性”中讨论了用户优先级倒置的问题以及处理优先级倒置的方法。

控制调度的接口调用

以下接口调用可控制进程调度。

使用 `priocntl`

通过 `priocntl(2)` 实现对活动类调度的控制。类属性通过 `fork(2)` 和 `exec(2)` 与优先级控制所需的调度参数和权限一起继承。此继承对 RT 和 TS 类发生。

priocntl(2) 是用于指定系统调用适用于的实时进程、一组进程或类的接口。priocntlset(2) 还提供用于指定系统调用适用于的完整进程组的更通用接口。

priocntl(2) 的命令参数可以为以下之一：PC_GETCID、PC_GETCLINFO、PC_GETPARMS 或 PC_SETPARMS。调用进程的实际或有效 ID 必须与受影响进程的实际或有效 ID 匹配，或必须具有超级用户特权。

PC_GETCID	此命令采用包含可识别类名称的结构的名称字段。将返回类 ID 以及类属性数据数组。
PC_GETCLINFO	此命令采用包含可识别类标识符的结构的 ID 字段。将返回类名称以及类属性数据数组。
PC_GETPARMS	此命令返回一个指定进程的调度类标识符或类特定调度参数。尽管 idtype 和 id 指定的组可能比较大，但是 PC_GETPARMS 仅返回一个进程的参数。由类选择进程。
PC_SETPARMS	此命令设置指定进程的调度类或类特定调度参数。

其他接口调用

sched_get_priority_max	返回指定策略的最大值。
sched_get_priority_min	返回指定策略的最小值。有关更多信息，请参见 sched_get_priority_max(3R) 手册页。
sched_rr_get_interval	将指定的 timespec 结构更新为当前的执行时间限制。有关更多信息，请参见 sched_get_priority_max(3RT) 手册页。
sched_setparam, sched_getparam	设置或获取指定进程的调度参数。
sched_yield	阻塞调用进程，直到调用进程返回到进程列表的开头。

控制调度的实用程序

控制进程调度的管理实用程序包括 dispadmin(1M) 和 priocntl(1)。这两个实用程序通过兼容选项和可装入模块支持 priocntl(2) 系统调用。这些实用程序提供在运行时控制实时进程调度的系统管理功能。

priocntl(1)

priocntl(1) 命令设置和检索进程的调度程序参数。

dispadmin(1M)

dispadmin(1M) 实用程序通过包括 -l 命令行选项，在运行时显示所有当前的进程调度类。还可以使用 RT 作为实时类的参数，为 -c 选项后面指定的类更改进程调度。

dispadmin(1M) 的类选项位于以下列表中：

-l　列出当前配置的调度程序类

-c　使用参数指定要显示或更改的类

-g　获取指定类的分发参数

-r　与 –g 配合使用，指定时间量程精度

-s　指定可在其中存放值的文件

包含分发参数的类特定文件也可以在运行时进行装入。使用此文件建立一组新的优先级，替代引导时建立的缺省值。此类特定文件必须采用 -g 选项所使用的格式声明参数。RT 类的参数在 rt_dptbl(4) 中，并在示例 12–1 中列出。

要将 RT 类文件添加到系统，以下模块必须存在：

- 类模块中装入 rt_dptbl(4) 的 rt_init() 例程。
- 提供分发参数和将指针返回到 config_rt_dptbl 的例程的 rt_dptbl(4) 模块。
- dispadmin(1M) 可执行文件。

以下步骤用于安装 RT 类分发表：

1. 使用以下命令装入类特定模块，其中 *module_name* 是类特定模块。

   ```
   # modload /kernel/sched/module_name
   ```

2. 调用 dispadmin 命令。

   ```
   # dispadmin -c RT -s file_name
   ```

   文件必须描述一个与要覆盖的表具有相同数量项目的表。

配置调度

两个调度类各自关联了一个参数表：rt_dptbl(4) 和 ts_dptbl(4)。这些表可以在引导时使用可装入模块或在运行时使用 dispadmin(1M) 进行配置。

分发程序参数表

实时的核心表建立 RT 调度的属性。rt_dptbl(4) 结构包括一个参数数组 struct rt_dpent_t。*n* 个优先级别中的每个级别都有一个参数。指定优先级别的属性由数组中的第 *i* 个参数结构指定，即 rt_dptbl[i]。

参数结构包含以下成员，这些成员在 /usr/include/sys/rt.h 头文件中也有介绍。

rt_globpri　与此优先级别关联的全局调度优先级。rt_globpri 值无法通过 dispadmin(1M) 进行更改。

rt_quantum　分配给此级别的进程的时间量程长度（秒）。有关更多信息，请参见第 260 页中的 “时间戳接口”。此时间量程值对于特定级别的进程只是缺省值或起始值。实时进程的时间量程可以使用 priocntl(1) 命令或 priocntl(2) 系统调用进行更改。

重新配置 config_rt_dptbl

实时管理员可以随时通过重新配置 config_rt_dptbl 来更改调度程序实时部分的行为。rt_dptbl(4) 手册页的标题为“替换 rt_dptbl 可装入模块”的一节中介绍了一种方法。

在正在运行的系统上检查或修改实时参数表的另一种方法是使用 dispadmin(1M) 命令。通过为实时类调用 dispadmin(1M)，可以从内核的核心表中检索当前 config_rt_dptbl 配置的当前 rt_quantum 值。当覆盖当前的核心表时，用于输入到 dispadmin(1M) 的配置文件必须符合 rt_dptbl(4) 手册页中描述的特定格式。

下面是具有优先权的进程 rtdpent_t 及其关联的时间量程 config_rt_dptbl[] 值的示例（如进程可能在 config_rt_dptbl[] 中显示的一样）。

示例 12–1　RT 类分发参数

```
rtdpent_t  rt_dptbl[] = {             129,    60,
     /* prilevel Time quantum */                                  130,    40,
        100,    100,                                                 131,    40,
        101,    100,                                                 132,    40,
        102,    100,                                                 133,    40,
        103,    100,                                                 134,    40,
        104,    100,                                                 135,    40,
        105,    100,                                                 136,    40,
        106,    100,                                                 137,    40,
        107,    100,                                                 138,    40
        108,    100,                                                 139,    40,
        109,    100,                                                 140,    20,
        110,    80,                                                  141,    20,
        111,    80,                                                  142,    20,
        112,    80,                                                  143,    20,
        113,    80,                                                  144,    20,
        114,    80,                                                  145,    20,
        115,    80,                                                  146,    20,
        116,    80,                                                  147,    20,
        117,    80,                                                  148,    20,
        118,    80,                                                  149,    20,
        119,    80,                                                  150,    10,
        120,    60,                                                  151,    10,
        121,    60,                                                  152,    10,
        122,    60,                                                  153,    10,
        123,    60,                                                  154,    10,
```

示例 12–1 RT 类分发参数 （续）

```
        124,    60,                                     155,    10,
        125,    60,                                     156,    10,
        126,    60,                                     157,    10,
        126,    60,                                     158,    10,
        127,    60,                                     159,    10,
        128,    60,                             }
```

内存锁定

锁定内存是实时应用程序的最重要问题之一。在实时环境中，进程必须能够保证连续的内存驻留，以减少延迟并阻止分页和交换。

本节介绍可用于 SunOS 中实时应用程序的内存锁定机制。

在 SunOS 中，进程的内存驻留由其当前状态、可用的总物理内存、活动进程的数量和进程的内存需求决定。此驻留适用于分时环境。实时进程通常不接受此驻留。在实时环境中，进程必须保证内存驻留以减少进程的内存访问和分发延迟。

SunOS 中的实时内存锁定由一组库例程提供。这些例程允许使用超级用户特权运行的进程将其虚拟地址空间的指定部分锁入物理内存。通过此方式锁定的页面会从分页中排除，直到页面解除锁定或进程退出。

操作系统对于可以在任何时间锁定的页面数量具有系统范围的限制。此限制是一个可调参数，其缺省值在引导时计算。缺省值基于页面框架的数量减去另一个百分比（当前设置为 10%）。

锁定页面

`mlock(3C)` 调用会请求将一个内存段锁入系统的物理内存。组成指定段的页面会出错。每个页面的锁定计数会递增。任何锁定计数值大于零的页面都会从分页活动中排除。

特定页面可以由多个进程通过不同的映射锁定多次。如果两个不同的进程锁定同一页面，则该页面将保持锁定状态直到两个进程都删除其锁定。但是，在一个指定的映射中，页面锁定不嵌套。相同进程对于相同地址的锁定接口的多次调用通过单个解除锁定请求即可删除。

如果删除通过其执行锁定的映射，则将隐式解除内存段锁定。通过关闭或截断文件删除页面时，也会隐式解除页面锁定。

调用 fork(2) 之后，子进程无法继承锁定。如果具有某些锁定内存的进程派生了子进程，则此子进程必须代表自己执行内存锁定操作才能锁定自己的页面。否则，此子进程会导致写复制页面错误，这些错误通常是与派生进程关联的不利结果。

解除页面锁定

为了解除内存页面的锁定，进程通过调用 munlock(3C) 请求释放锁定虚拟页面的段。munlock 会递减指定物理页面的锁定计数。将页面的锁定计数递减到 0 之后，页面便可正常交换。

锁定所有页面

超级用户进程可以通过调用 mlockall(3C) 请求锁定其地址空间内的所有映射。如果设置了 MCL_CURRENT 标志，则锁定现有的所有内存映射。如果设置了 MCL_FUTURE 标志，则将添加到现有映射或替换现有映射的所有映射锁入内存。

恢复粘滞锁

页面在其锁定计数达到 65535 (0xFFFF) 时会永久锁入内存。值 0xFFFF 通过实施定义。在将来的发行版本中，该值可能会更改。以此方式锁定的页面无法解除锁定。重新引导要恢复的系统。

高性能 I/O

本节介绍具有实时进程的 I/O。在 SunOS 中，库提供两组接口和调用，以执行快速异步 I/O 操作。POSIX 异步 I/O 接口是最新的标准。SunOS 环境还提供文件和内存中同步操作和模式，以防止信息丢失和数据不一致。

标准 UNIX I/O 与应用程序编程器同步。调用 read(2) 或 write(2) 的应用程序通常会等待系统调用完成。

实时应用程序需要异步受限的 I/O 行为。发出异步 I/O 调用的进程继续执行，而不等待 I/O 操作完成。当 I/O 操作完成后，会通知调用者。

异步 I/O 可以用于任何 SunOS 文件。文件会异步打开并且不需要进行特殊标记。异步 I/O 传输具有三个元素：调用、请求和操作。应用程序调用异步 I/O 接口，对 I/O 的请求置于队列中，然后调用立即返回。有时，系统会将请求从队列中删除并启动 I/O 操作。

异步和标准 I/O 请求可以在任意文件描述符上混合。系统不维护特定的读取和写入请求序列，而是随意重新排列所有待处理的读取和写入请求。如果应用程序需要特定的序列，则该应用程序必须确保在发出相关请求之前完成前面的操作。

POSIX 异步 I/O

POSIX 异步 I/O 通过 aiocb 结构执行。aiocb 控制块标识每个异步 I/O 请求并包含所有控制信息。控制块一次只能用于一个请求。控制块在其请求完成后可以重复使用。

典型的 POSIX 异步 I/O 操作通过调用 aio_read(3RT) 或 aio_write(3RT) 启动。可以使用轮询或信号确定操作的完成。如果将信号用于完成操作，则每个操作都可以唯一标记。然后标记在生成的信号的 si_value 组件中返回。请参见 siginfo(3HEAD) 手册页。

aio_read	通过异步 I/O 控制块调用 aio_read(3RT) 以启动读取操作。
aio_write	通过异步 I/O 控制块调用 aio_write(3RT) 以启动写入操作。
aio_return, aio_error	已知操作已完成后，分别调用 aio_return(3RT) 和 aio_error(3RT) 以获取返回值和错误值。
aio_cancel	通过异步 I/O 控制块调用 aio_cancel(3RT) 以取消待处理的操作。如果某个特定的请求由控制块指定，则 aio_cancel 可以用于取消该请求。aio_cancel 还可以取消指定文件描述符的所有待处理的请求。
aio_fsync	aio_fsync(3RT) 将指定文件上所有待处理的 I/O 操作的异步 fsync(3C) 或 fdatasync(3RT) 请求排入队列。
aio_suspend	aio_suspend(3RT) 挂起调用者，仿佛前面的一个或多个异步 I/O 请求是同步发出的一样。

Solaris 异步 I/O

本节讨论 Solaris 操作环境中的异步 I/O 操作。

通知 (SIGIO)

当异步 I/O 调用成功返回时，I/O 操作只是已排入队列并等待完成。实际操作具有一个返回值和一个潜在的错误标识符。如果调用是同步的，则将此返回值和潜在错误标识符返回给调用者。完成 I/O 时，会将返回值和错误值存储在用户在请求时指定的位置（作为 aio_result_t 的指针）。aio_result_t 的结构在 <sys/asynch.h> 中定义：

```
typedef struct aio_result_t {
     ssize_t    aio_return; /* return value of read or write */
```

```
    int         aio_errno;  /* errno generated by the IO */
} aio_result_t;
```

更新 aio_result_t 后，会将 SIGIO 信号传送给发出 I/O 请求的进程。

请注意，具有两个或更多待处理的异步 I/O 操作的进程没有特定的方法来确定 SIGIO 信号的原因。接收到 SIGIO 的进程应该检查其可能会生成 SIGIO 信号的所有条件。

使用 aioread

aioread(3AIO) 例程是 read(2) 的异步版本。除了常规的读取参数，aioread(3AIO) 还使用指定文件位置和 aio_result_t 结构地址的参数。生成的有关操作的信息存储在 aio_result_t 结构中。文件位置指定在操作之前要在文件内执行的查找。不管 aioread(3AIO) 调用是成功还是失败，都会更新文件指针。

使用 aiowrite

aiowrite(3AIO) 例程是 write(2) 的异步版本。除了常规的写入参数，aiowrite(3AIO) 还使用指定文件位置和 aio_result_t 结构地址的参数。生成的有关操作的信息存储在 aio_result_t 结构中。

文件位置指定在操作之前要在文件内执行的查找操作。如果 aiowrite(3AIO) 调用成功，则文件指针更新为导致成功查找和写入的位置。当写入失败时，文件指针也会更新，以允许后续的写入请求。

使用 aiocancel

aiocancel(3AIO) 例程尝试取消其 aio_result_t 结构指定为参数的异步请求。仅在请求仍然位于队列中时，aiocancel(3AIO) 调用才能成功。如果操作正在进行中，则 aiocancel(3AIO) 会失败。

使用 aiowait

aiowait(3AIO) 调用会阻塞调用进程，直到至少完成一个未处理的异步 I/O 操作。超时参数指向要等待 I/O 完成的最大间隔。超时值为零指定不想等待。aiowait(3AIO) 返回已完成操作的 aio_result_t 结构的指针。

使用 poll()

要同步确定异步 I/O 事件的完成，而不是依赖于 SIGIO 中断，请使用 poll(2)。您也可以轮询以确定 SIGIO 中断的源。

对非常大数量的文件使用 poll(2) 时，其会比较缓慢。此问题可通过 poll(7d) 解决。

使用 poll 驱动程序

使用 /dev/poll 可提供用于轮询大量文件描述符的高度可扩展的方式。此可扩展性通过一组新的 API 和新的驱动程序 /dev/poll 提供。/dev/poll API 是 poll(2) 的备用选项，而非替换选项。使用 poll(7d) 提供 /dev/poll API 的详细信息和示例。如果正确使用，/dev/poll API 的扩展性要比 poll(2) 好得多。此 API 特别适合满足以下条件的应用程序：

- 重复轮询大量文件描述符的应用程序
- 轮询的文件描述符相对稳定，意味着描述符不是不断关闭和重新打开
- 与要轮询的文件描述符总数量相比，实际具有待处理的已轮询事件的文件描述符组很小

使用 close

通过调用 close(2) 关闭文件。close(2) 调用会取消任何可以关闭的未处理异步 I/O 请求。close(2) 会等待无法取消的操作。有关更多信息，请参见第 256 页中的 “使用 aiocancel”。当 close(2) 返回时，针对文件描述符没有待处理的异步 I/O。关闭文件时，仅取消排入指定文件描述符队列的异步 I/O 请求。不会取消其他文件描述符的任何 I/O 待处理请求。

同步的 I/O

应用程序可能需要确保已将信息写入稳定的存储，或按照特定的顺序执行文件更新。同步的 I/O 可满足这些需求。

同步模式

在 SunOS 中，如果系统确保所有写入数据在任何后续的文件打开操作后可读取，则写入操作成功。此检查假定物理存储介质没有任何故障。当物理存储介质上的数据映像可用于请求进程时，读取操作的数据传输成功完成。当关联的数据传输成功或操作诊断为不成功时，I/O 操作完成。

在以下情况下，I/O 操作实现了同步 I/O 数据完整性完成：

- 对于读取，操作已完成或已诊断（如果不成功）。仅当数据映像成功传输到请求进程时，读取才完成。如果在待处理的写入请求影响要读取的数据时请求同步读取操作，则这些写入请求在读取数据之前成功完成。
- 对于写入，操作已完成或已诊断（如果不成功）。成功传输写入请求中指定的数据时，写入操作成功。此外，必须成功传输检索数据所需的所有文件系统信息。
- 对于不是数据检索所必需的文件属性，在返回到调用进程之前不进行传输。
- 同步 I/O 文件完整性完成要求在返回到调用进程之前成功传输所有与 I/O 操作相关的文件属性。否则，同步 I/O 文件完整性完成与同步 I/O 数据完整性完成相同。

同步文件

fsync(3C) 和 fdatasync(3RT) 将文件显式同步到辅助存储。

fsync(3C) 例程确保在 I/O 文件完整性完成级别同步接口。fdatasync(3RT) 确保在 I/O 数据完整性完成级别同步接口。

应用程序可以在操作完成前同步每个 I/O 操作。使用 open(2) 或 fcntl(2) 在文件描述上设置 O_DSYNC 标志可确保所有 I/O 写入在操作完成之前实现 I/O 数据完成。在文件描述上设置 O_SYNC 标志可确保在将操作指示为已完成之前，所有 I/O 写入都已实现完成。在文件描述上设置 O_RSYNC 标志可确保所有 I/O 读取 read(2) 和 aio_read(3RT) 实现描述符设置请求的相同级别的完成。描述符设置可以是 O_DSYNC或 O_SYNC。

进程间通信

本节介绍 SunOS 中作为与实时处理相关的接口的进程间通信 (Interprocess Communication, IPC) 接口。信号、管道、FIFO、消息队列、共享内存、文件映射和信号量都将在此处介绍。有关对进程间通信有用的库、接口和例程的更多信息，请参见第 7 章，进程间通信。

处理信号

发送者可以使用 sigqueue(3RT) 将信号与少量信息一起发送至目标进程。

要将待处理信号的后续出现排入队列，目标进程必须为指定的信号设置 SA_SIGINFO 位。请参见 sigaction(2) 手册页。

目标进程通常异步接收信号。要同步接收信号，请阻塞信号并调用 sigwaitinfo(3RT) 或 sigtimedwait(3RT)。请参见 sigprocmask(2) 手册页。此过程会导致同步接收信号。sigqueue(3RT) 的调用者发送的值存储在 siginfo_t 参数的 si_value 成员中。使信号保持解除阻塞状态会导致将信号传送至 sigaction(2) 指定的信号处理程序，并在该处理程序的 siginfo_t 参数的 si_value 中显示值。

具有关联值的指定数量的信号可以通过进程发送，也可以保持不传送。{SIGQUEUE_MAX} 信号的存储在首次调用 sigqueue(3RT) 时分配。随后，sigqueue(3RT) 的调用要么成功排入目标进程的队列，要么在限制的时间内失败。

管道、命名管道和消息队列

管道、命名管道和消息队列的行为类似于字符 I/O 设备。这些接口具有不同的连接方法。有关管道的更多信息，请参见第 109 页中的 “进程之间的管道”。有关命名管道的更

多信息，请参见第 110 页中的 “命名管道”。有关消息队列的更多信息，请参见第 114 页中的 “System V 消息”和第 111 页中的 “POSIX 消息”。

使用信号量

信号量还以 System V 和 POSIX 形式提供。有关更多信息，请参见第 116 页中的 “System V 信号量”和第 112 页中的 “POSIX 信号量”。

请注意，使用信号量可能会导致优先级倒置，除非通过本章前面所述的技术明确避免优先级倒置。

共享内存

进程通信的最快方式是直接通过共享内存段。当超过两个进程尝试同时读取和写入共享内存时，内存内容可能变得不准确。这种潜在的不准确性是使用共享内存的主要困难。

异步网络通信

本节介绍针对实时应用程序使用套接字或传输级别接口 (Transport-Level Interface, TLI) 进行的异步网络通信。使用套接字的异步联网通过将 `SOCK_STREAM` 类型的开放套接字设置为异步和非阻塞来实现。有关异步套接字的更多信息，请参见第 149 页中的 “高级套接字主题”。使用 STREAMS 异步功能和 TLI 库例程的非阻塞模式的组合，支持 TLI 事件的异步网络处理。

有关传输层接口的更多信息，请参见第 9 章，使用 XTI 和 TLI 编程。

联网模式

套接字和传输级别接口都提供两种服务模式：**连接模式**和**无连接模式**。

连接模式服务是面向线路的服务。此服务允许通过建立的连接以可靠、有序的方式传输数据。此服务还提供标识过程，可在数据传输阶段避免地址解析和传输开销。此服务适用于需要相对长时间、以数据流为中心的交互的应用程序。

无连接模式服务是面向消息的服务，支持自包含的单元中的数据传输，且多个单元之间不需要任何逻辑关系。单个服务请求传递将数据单元从发送者传送至传输提供者所需的所有信息。此服务请求包括目标地址和要传送的数据。无连接模式服务适用于涉及不需要保证按顺序传送数据的短期交互的应用程序。无连接传输通常不可靠。

计时功能

本节介绍可用于 SunOS 中的实时应用程序的计时功能。使用这些机制的实时应用程序需要本节中所列出例程的手册页中的详细信息。

SunOS 的计时接口分为两个不同的方面：**时间戳**和**间隔计时器**。时间戳接口可测量已经过的时间。时间戳接口还允许应用程序测量状态的持续时间或事件之间的相隔时间。间隔计时器允许应用程序在指定的时间唤醒以及根据时间的推移调度活动。

时间戳接口

两个接口提供时间戳。gettimeofday(3C) 以 *timeval* 结构提供当前时间，表示自格林威治标准时间 1970 年 1 月 1 日午夜以来的时间（以秒和微秒为单位）。clock_gettime（clockid 为 CLOCK_REALTIME）以 timespec 结构提供当前时间，表示 gettimeofday(3C) 所返回的相同时间间隔（以秒和纳秒为单位）。

SunOS 使用硬件定期计时器。对于某些工作站，硬件定期计时器是计时信息的唯一来源。如果硬件定期计时器是计时信息的唯一来源，则时间戳的精确度将限制为计时器的分辨率。对于其他平台，分辨率为 1 微秒的计时器寄存器表示时间戳精确到 1 微秒。

间隔计时器接口

通常，实时应用程序使用间隔计时器来调度操作。间隔计时器可以分为以下两种类型：**一次性**类型或**定期**类型。

一次性计时器是已经过设置的计时器，设置为相对于当前时间或绝对时间的到期时间。计时器到期一次后即解除设置。此类型的计时器对于将数据传输到存储后清除缓冲区，或使操作超时很有用。

定期计时器设置有初始到期时间（绝对或相对）以及重复间隔。每次间隔计时器到期时，都会按照重复间隔重新装入此计时器。然后，重新设置此计时器。此计时器可帮助进行数据日志记录或伺服控制。调用间隔计时器接口时，会将小于计时器分辨率的时间值向上舍入为下一个硬件计时器间隔的倍数。此间隔通常为 10ms。

SunOS 具有两组计时器接口。setitimer(2) 和 getitimer(2) 接口运行固定设置的计时器，此类计时器称为 BSD 计时器，使用 timeval 结构指定时间间隔。使用 timer_create(3RT) 创建的 POSIX 计时器用于运行 POSIX 时钟 CLOCK_REALTIME。POSIX 计时器操作以 timespec 结构表示。

getitimer(2) 和 setitimer(2) 函数分别用于检索和确定指定的 BSD 间隔计时器的值。可用于进程的三个 BSD 间隔计时器包括指定了 ITIMER_REAL 的实时计时器。如果 BSD 计时器已进行设置并允许到期，则系统会向设置此计时器的进程发送适当的信号。

timer_create(3RT) 例程最多可以创建 TIMER_MAX 个 POSIX 计时器。调用者可以指定计时器到期时应向进程发送的信号和关联值。timer_settime(3RT) 和 timer_gettime(3RT) 例程分别用于检索和确定指定的 POSIX 间隔计时器的值。所需的信号暂挂传送时，POSIX 计时器便会到期。此时会对计时器到期的次数进行计数，并且 timer_getoverrun(3RT) 将检索此计数。timer_delete(3RT) 用于取消分配 POSIX 计时器。

以下示例说明了如何使用 setitimer(2) 生成定期中断，以及如何控制计时器中断的到达。

示例 12–2 控制计时器中断

```
#include    <unistd.h>
#include    <signal.h>
#include    <sys/time.h>

#define TIMERCNT 8

void timerhandler();
int     timercnt;
struct     timeval alarmtimes[TIMERCNT];

main()
{
    struct itimerval times;
    sigset_t    sigset;
    int        i, ret;
    struct sigaction act;
    siginfo_t    si;

    /* block SIGALRM */
    sigemptyset (&sigset);
    sigaddset (&sigset, SIGALRM);
    sigprocmask (SIG_BLOCK, &sigset, NULL);

    /* set up handler for SIGALRM */
    act.sa_action = timerhandler;
    sigemptyset (&act.sa_mask);
    act.sa_flags = SA_SIGINFO;
    sigaction (SIGALRM, &act, NULL);
    /*
     * set up interval timer, starting in three seconds,
     *    then every 1/3 second
     */
    times.it_value.tv_sec = 3;
    times.it_value.tv_usec = 0;
    times.it_interval.tv_sec = 0;
    times.it_interval.tv_usec = 333333;
    ret = setitimer (ITIMER_REAL, &times, NULL);
    printf ("main:setitimer ret = %d\n", ret);

    /* now wait for the alarms */
    sigemptyset (&sigset);
    timerhandler (0, si, NULL);
    while (timercnt < TIMERCNT) {
```

示例 12–2 控制计时器中断 （续）

```
        ret = sigsuspend (&sigset);
    }
    printtimes();
}

void timerhandler (sig, siginfo, context)
    int        sig;
    siginfo_t    *siginfo;
    void        *context;
{
    printf ("timerhandler:start\n");
    gettimeofday (&alarmtimes[timercnt], NULL);
    timercnt++;
    printf ("timerhandler:timercnt = %d\n", timercnt);
}

printtimes ()
{
    int    i;

    for (i = 0; i < TIMERCNT; i++) {
        printf("%ld.%0l6d\n", alarmtimes[i].tv_sec,
                alarmtimes[i].tv_usec);
    }
}
```

◆ ◆ ◆ 第 13 章

13 Solaris ABI 和 ABI 工具

Solaris 应用程序二进制接口 (Application Binary Interface, ABI) 定义了可供应用程序开发者使用的接口。如果应用程序符合 ABI，则可以增强二进制稳定性。本章讨论了 Solaris ABI 以及所提供的用于验证应用程序的 ABI 符合性的工具，其中包括以下内容：

- Solaris ABI 的定义和用途，在第 264 页中的 “定义 Solaris ABI”中进行讨论。
- 两个 ABI 工具（`appcert` 和 `apptrace`）的用法，在第 266 页中的 “Solaris ABI 工具”中进行讨论。

什么是 Solaris ABI？

Solaris ABI 是一组受支持的运行时接口，可供应用程序在 Solaris 操作系统中使用。下面列出了最重要的 ABI 组件：

- Solaris 系统库提供的接口，手册页的第 3 节对其进行了介绍
- Solaris 内核系统调用提供的接口，手册页的第 2 节对其进行了介绍
- 各种系统文件和目录的位置及格式，手册页的第 4 节对其进行了介绍
- Solaris 实用程序的输入和输出语法及语义，手册页的第 1 节对其进行了介绍

Solaris ABI 的主要组件是一组系统库接口。本章中的术语 *ABI* 专指此组件。ABI 仅包含 C 语言接口，因为 Solaris 操作系统仅为 C 语言提供接口。

写入 Solaris API（Application Programming Interface，应用编程接口）的 C 源代码将通过 C 编译器转换为四个 ABI 版本之一的二进制形式。这四个版本包括：

- 32 位 SPARC
- 64 位 SPARC
- 32 位 x86
- 64 位 x86 (Opteron)

尽管 ABI 与 API 非常相似，但是源代码编译过程还是有一些明显差异：

- 编译器指令（如 `#define`）可以更改或替换源代码级别的构造。生成的二进制内容可能缺少源代码中存在的符号，或者包含源代码中不存在的符号。

- 编译器可能会生成特定于处理器的符号（如算术指令），用于扩充或替换源代码的构造。
- 编译器的二进制布局可能特定于此编译器以及此编译器接受的源代码语言的版本。这种情况下，使用不同编译器编译的相同代码可能会生成不兼容的二进制内容。

由于这些原因，源代码级别 (API) 兼容性无法在各 Solaris 发行版之间实现所要求的二进制兼容性。

Solaris ABI 由操作系统提供的支持接口构成。系统可用的接口中有一些旨在专供操作系统使用。这些专用接口不能被应用程序使用。在 SunOS 5.6 发行版之前，Solaris 库中的所有接口均可供应用程序开发者使用。利用 Solaris 链接编辑器中的库符号作用域技术，可以将不计划在库外部使用的接口的作用域缩小为完全在库的局部使用。有关详细信息，请参见《链接程序和库指南》。由于系统要求，并非所有专用接口都具有此类缩小了的作用域。这些接口会标记为**专用**，并且不包括在 Solaris ABI 中。

定义 Solaris ABI

Solaris ABI 是在 Solaris 库中定义的。这些定义是通过链接编辑器和运行时链接程序中使用的库版本控制技术和策略完成的。

Solaris 库中的符号版本控制

Solaris 链接编辑器和运行时链接程序使用两种库版本控制：文件版本控制和符号版本控制。在文件版本控制中，库名称后面会附带版本号，如 `libc.so.1`。对该库中的一个或多个公共接口进行不兼容的更改之后，版本号将递增（如递增到 `libc.so.2`）。在动态链接的应用程序中，生成时绑定的符号在运行时可能不会出现在库中。在符号版本控制中，Solaris 链接程序将符号集与名称关联。然后在运行时链接过程中，链接程序将检查库中是否存在此名称，以验证是否存在关联的符号。

库符号版本控制将符号集与符号版本名称关联，如果此名称具有编号方案，则还会通过映射文件将符号集与其编号关联。以下是假设的 Sun 库 `libfoo.so.1` 的映射文件示例。

```
SUNW_1.2 {
    global:
        symbolD;
        symbolE
} SUNW_1.1;

SUNW_1.1 {
    global:
        symbolA;
        symbolB;
        symbolC;
```

```
};

SUNWprivate {
    global:
        __fooimpl;
    local: *;
};
```

此映射文件表明 symbolA、symbolB 和 symbolC 与版本 SUNW_1.1 关联，symbolD 和 symbolE 与 SUNW_1.2 关联，并且 SUNW_1.2 继承了与 SUNW_1.1 关联的所有符号。符号 __fooimpl 与不同名称的符号集关联，该符号集没有带编号的继承链。

在生成过程中，链接编辑器会检查应用程序使用的符号，并且记录应用程序中与这些符号相关的符号集名称。如果是一系列具有继承链关系的命名集，则链接编辑器将记录包含应用程序所使用的所有符号的最小命名集。如果应用程序仅使用 symbolA 和 symbolB，则链接编辑器将记录 SUNW_1.1 的某一依赖项。如果应用程序使用 symbolA、symbolB 和 symbolD，则链接编辑器将记录 SUNW_1.2 的某一依赖项，因为 SUNW_1.2 包括 SUNW_1.1。

在运行时，链接程序会验证在所链接的库中，是否存在应用程序中作为依赖项而记录的版本名称。通过此过程可以快速验证是否存在所需的符号。有关更多详细信息，请参见《链接程序和库指南》。

注 – 映射文件中的 *local:* * 指令表示，对于库中任何不与命名的符号集显式关联的符号，其作用域均为库局部作用域。此类局部作用域符号在库外部不可见。此约定可以确保符号仅与符号版本控制名称关联时才可见。

使用符号版本控制标记 Solaris ABI

由于库中所有可见的符号都属于某个命名的符号集，因此可以使用命名方案来标记符号的 ABI 状态。这种标记操作可以通过将所有专用接口与以 *SUNWprivate* 开头的符号集名称关联来完成。公共接口以其他名称开头，具体如下：

- *SYSVABI*，适用于根据 System V ABI 定义所定义的接口
- *SISCD*，适用于根据 SPARC International SPARC Compliance Definition 定义的接口
- *SUNW*，适用于由 Sun Microsystems 定义的接口

这些已命名的公共符号集按照 *major.minor* 编号方案进行编号。如果集合中包括新符号，则集合的次版本号将增加。如果现有符号的更改致使符号与其以前的行为不兼容，则包含此符号的集合的主版本号将增加。如果现有符号更改且与以前不兼容，则库文件名中的版本号也将增加。

因此，Solaris 库 ABI 的定义会包含在多个库中，并且包括与不以 *SUNWprivate* 开头的符号版本名称关联的符号集。可以使用 pvs 命令列出库中的符号。

Solaris ABI 工具

Solaris 操作系统提供了两个工具，用于验证应用程序使用的 Solaris 接口是否符合 Solaris ABI。appcert 实用程序会静态检查 ELF 二进制代码所使用的 Solaris 库接口，以确定是否使用了专用接口。appcert 实用程序可针对其发现的任何潜在的二进制稳定性问题生成摘要和详细报告。apptrace 工具使用运行时链接程序的链接审计功能，在应用程序运行时可动态跟踪 Solaris 库例程调用。利用此功能，开发者可以检查应用程序使用 Solaris 系统接口的情况。

使用 ABI 工具，可以轻松快速地识别可能与给定的 Solaris 发行版存在二进制兼容性问题的二进制对象。要检查二进制稳定性，请执行以下步骤：

- **对于当前 Solaris 发行版使用 appcert 进行分级。**此操作可以分别识别使用有问题的接口与使用没有问题的接口的二进制对象。
- **对于目标 Solaris 发行版使用 apptrace 进行验证。**此操作通过在使用这些接口时对其进行动态观察，可以验证是否存在接口兼容性问题。

appcert 实用程序

appcert 实用程序是一个 Perl 脚本，用于静态检查 ELF 二进制对象，并将所使用的库符号与给定的 Solaris 发行版中的公共接口和专用接口模型进行比较。此实用程序在 SPARC 或 x86 平台上运行。它可以检查 SPARC 和 x86 的 32 位接口以及 SPARC 上 64 位接口的使用情况。请注意，appcert 仅检查 C 语言接口。

随着新 Solaris 发行版的问世，某些库接口的行为可能会发生变化，或者完全消失。这些变化会影响依赖这些接口的应用程序的性能。Solaris ABI 定义了可供应用程序使用的安全稳定的运行时库接口。appcert 实用程序的设计目的在于帮助开发者验证应用程序与 Solaris ABI 的符合性。

appcert 的检查内容

appcert 实用程序将针对以下内容对应用程序进行检查：

- 专用符号的使用情况
- 静态链接
- 非绑定符号

专用符号的使用情况

专用符号是 Solaris 库用来进行互相调用的函数或数据。专用符号的语义行为可能会发生变化，有时候还可能会删除符号。此类符号称为**降级符号**。专用符号可变的性质可能会导致依赖于专用符号的应用程序变得不稳定。

静态链接

Solaris 库之间的专用符号调用的语义在不同的发行版之间可能会发生变化。因此，创建指向归档文件的静态链接会降低应用程序的二进制稳定性。使用指向归档文件对应的共享目标文件的动态链接可以避免此问题。

非绑定符号

appcert 实用程序使用动态链接程序解析所检查的应用程序使用的库符号。动态链接程序无法解析的符号称为**非绑定符号**。非绑定符号可能是由于环境问题造成的，如 LD_LIBRARY_PATH 变量设置不正确。非绑定符号也可能是由于生成问题造成的，如在编译时省略了 -l*lib* 或 -z 转换参数的定义。尽管这些示例中的问题都不很重要，但是 appcert 报告的非绑定符号可能指示更严重的问题，如与不再存在的专用符号存在相关性。

appcert 不检查的内容

如果 appcert 检查的目标文件依赖于库，则必须在目标文件中记录这些依赖项。为此，编译代码时请确保使用编译器的 -l 转换参数。如果目标文件依赖于其他共享库，则运行 appcert 时必须能够通过 LD_LIBRARY_PATH 或 RPATH 访问这些库。

appcert 应用程序不能检查 64 位应用程序，除非计算机运行的是 64 位 Solaris 内核。由于 Solaris 未提供任何 64 位静态库，因此 appcert 不会对 64 位应用程序执行静态链接检查。

appcert 实用程序不能检查以下内容：

- 完全或部分静态链接的目标文件。完全静态链接的对象将被报告为不稳定。
- 没有执行权限集的可执行文件。appcert 实用程序会跳过此类可执行文件。没有执行权限集的共享对象按正常方式进行检查。
- 用户 ID 设置为 root 的目标文件。
- 非 ELF 的可执行文件，如 shell 脚本。
- 采用除 C 以外其他语言的 Solaris 接口。代码不需要采用 C 语言，但对 Solaris 库的调用必须使用 C 语言。

使用 appcert

要使用 appcert 检查应用程序，请键入以下内容：

appcert *object|directory*

将 *object|directory* 替换为以下任一项：

- 需要 appcert 检查的完整对象列表

- 包含此类对象的完整目录列表

注 – 运行 appcert 的环境可能不同于运行应用程序的环境。如果这两个环境不同，则 appcert 可能无法正确解析对 Solaris 库接口的引用。

---

appcert 实用程序使用 Solaris 运行时链接程序为每个可执行文件或共享目标文件构造一个接口依赖项的配置文件。此配置文件用于确定应用程序依赖的 Solaris 系统接口。此配置文件中概述的依赖项将与 Solaris ABI 进行比较，以验证是否与其相符。不存在任何专用接口。

appcert 实用程序以递归方式搜索目录中的目标文件，从而忽略非 ELF 目标文件。appcert 检查完应用程序之后，会在标准输出（通常是屏幕）中显示汇总报告。此报告的副本将写入工作目录（通常是 /tmp/appcert.*pid*,）中名为 Report 的文件中。在子目录名称中，*pid* 表示此特定 appcert 实例的 1 至 6 位数字的进程 ID。有关 appcert 写入输出文件的目录结构的更多信息，请参见第 269 页中的 “appcert 结果”。

appcert 选项

以下选项用于修改 appcert 实用程序的行为。可以在命令行中 appcert 命令之后，*object|directory* 操作数之前键入其中任一选项。

-B	以批处理模式运行 appcert。 在批处理模式下，appcert 生成的报告中包含的每一行都针对一个所检查的二进制对象。 以 PASS 开头的行表示该行中指定的二进制对象未触发任何 appcert 警告。 以 FAIL 开头的行表示在该二进制对象中发现问题。 以 INC 开头的行表示无法完全检查该行中指定的二进制对象。
-f *infile*	文件 *infile* 应包含要检查的文件列表，每行包含一个文件名。这些文件将添加到命令行中已经指定的任何文件中。如果使用此转换参数，则无需在命令行中指定对象或目录。
-h	显示 appcert 的使用情况信息。
-L	缺省情况下，appcert 会对应用程序中的所有共享对象进行注释，并将共享对象所驻留的目录附加到 LD_LIBRARY_PATH。使用 -L 转换参数可以禁用这一行为。
-n	缺省情况下，appcert 在搜索目录以查找要检查的二进制对象时会打开符号链接。使用 -n 转换参数可以禁用这一行为。
-S	将 Solaris 库目录 /usr/openwin/lib 和 /usr/dt/lib 附加到 LD_LIBRARY_PATH。

`-w` *working_dir* 指定要运行库组件的目录。另外，还将在此转换参数指定的目录中创建临时文件。如果未指定此转换参数，则 `appcert` 使用 `/tmp` 目录。

使用 appcert 进行应用程序分级

使用 `appcert` 实用程序，可以快速轻松地识别给定集合中存在潜在稳定性问题的应用程序。如果 `appcert` 未报告任何稳定性问题，则应用程序在后续 Solaris 发行版中可能不会遇到二进制稳定性问题。下表列出了一些常见的二进制稳定性问题。

表 13–1 常见的二进制稳定性问题

问题	采取的措施
使用确定要更改的专用符号	立即停止使用此类符号。
使用尚未更改的专用符号	应用程序目前仍然可以运行，但是应根据实际情况尽快停止使用此类符号。
静态链接包含共享对象对应项的库	改用共享对象对应项。
静态链接不包含共享对象对应项的库	如果可能，使用命令 `ld -z allextract` 将 `.a` 文件转换为 `.so` 文件。否则，继续使用静态库，直到共享对象可用为止。
使用没有可用公共对应项的专用符号	与 Oracle 联系，请求获取一个公共接口。
使用已过时的符号，或者使用计划要删除的符号	应用程序目前仍然可以运行，但是应根据实际情况尽快停止使用此类符号。
使用已更改的公共符号	重新编译。

给定的发行版中可能不会出现由于使用专用接口而导致的潜在稳定性问题。专用接口的行为在不同的发行版之间并不一定会发生变化。要验证专用接口的行为在目标发行版中是否已发生变化，请使用 `apptrace` 工具。`apptrace` 的用法在第 271 页中的 “使用 apptrace 进行应用程序验证”中进行讨论。

appcert 结果

`appcert` 实用程序对应用程序目标文件的分析结果会写入多个文件，这些文件位于 `appcert` 实用程序的工作目录中，通常为 `/tmp`。工作目录下的主要子目录是 `appcert.`*pid*，其中，*pid* 是此 `appcert` 实例的进程 ID。`appcert` 实用程序的结果会写入以下文件：

`Index` 包含所检查的二进制对象与特定于此二进制对象的 `appcert` 输出所在子目录之间的映射。

`Report` 包含运行 `appcert` 时在 `stdout` 上显示的汇总报告的副本。

`Skipped` 包含要求 `appcert` 检查但强制跳过的二进制对象的列表以及跳过每个二进制对象的原因。下面列出了这些原因：

- 文件不是二进制对象
- 用户无法读取文件
- 文件名包含元字符
- 文件未设置执行位

`objects/`*`object_name`* `appcert` 检查的每个对象的 `objects` 子目录下都有一个单独的子目录。其中每个子目录均包含以下文件：

`check.demoted.symbols`	包含 `appcert` 怀疑为降级的 Solaris 符号的列表。
`check.dynamic.private`	包含与对象直接绑定的专用 Solaris 符号的列表。
`check.dynamic.public`	包含与对象直接绑定的公共 Solaris 符号的列表。
`check.dynamic.unbound`	包含运行 `ldd -r` 时不是由动态链接程序绑定的符号的列表。另外，还包括 `ldd` 返回的包含 "`file not found`" 的行。
`summary.dynamic`	包含 `appcert` 检查的对象中的打印机格式的动态绑定摘要，其中包括每个 Solaris 库中使用的公共符号和专用符号表。

返回以下四个退出值之一。

0 `appcert` 未找到二进制不稳定性的潜在原因。

1 `appcert` 实用程序未成功运行。

2 `appcert` 检查的某些对象存在潜在的二进制稳定性问题。

3 `appcert` 实用程序未找到任何要检查的二进制对象。

更正 appcert 报告的问题

- **使用专用符号**－依赖专用符号的应用程序在与其开发发行版不同的 Solaris 发行版中可能无法运行。发生这种情况是因为，出现在给定 Solaris 发行版中的专用符号在其他发行版中可能具有不同的行为或者不存在。如果 `appcert` 报告应用程序中使用了专用符号，请重新编写应用程序，以避免使用专用符号。
- **降级符号**－降级符号是 Solaris 库中已删除的函数或数据变量，也可以是在以后的 Solaris 发行版中作用域限定为局部的函数或数据变量。如果某个发行版的库未导出此符号，则直接调用此类符号的应用程序将无法在此发行版中运行。

- **非绑定符号**－非绑定符号是指在 appcert 调用时动态链接程序无法解析的应用程序所引用的库符号。尽管非绑定符号并不始终指示二进制稳定性欠佳，但是非绑定符号可能会指示更严重的问题，如存在降级符号的依赖项。
- **过时库**－将来的发行版中可能会从 Solaris 操作系统中删除过时的库。appcert 实用程序可用于标志此类库的所有使用情况。依赖此类库的应用程序在不支持此类库的将来发行版中可能无法正常运行。要避免此问题，请勿使用过时库中的接口。
- **使用 sys_errlist 或 sys_nerr**－使用 sys_errlist 和 sys_nerr 符号可能会降低二进制稳定性。可能会在 sys_errlist 数组结尾后面进行引用。要避免此风险，请改用 strerror。
- **使用强符号和弱符号**－与弱符号关联的强符号会保留作为专用符号，因为其行为在将来的 Solaris 发行版中可能会发生变化。应用程序只应直接引用弱符号。_socket 即是一个强符号示例，它与弱符号 socket 关联。

使用 apptrace 进行应用程序验证

apptrace 实用程序是一个 C 程序，用于在应用程序运行时动态跟踪对 Solaris 库例程的调用。apptrace 可以在 SPARC 或 x86 平台上运行。apptrace 既可以跟踪 SPARC 和 x86 的 32 位接口调用，也可以跟踪 SPARC 上的 64 位接口调用。与 appcert 一样，apptrace 仅检查 C 语言接口。

应用程序验证

使用 appcert 确定应用程序面临二进制不稳定性风险之后，可以借助 apptrace 评估每种情况下的风险程度。要确定应用程序与给定发行版的二进制兼容性，请通过 apptrace 验证是否可以成功使用应用程序所用的每个接口。

apptrace 实用程序可以验证应用程序是否正确使用公共接口。例如，使用 open() 直接打开管理文件 /etc/passwd 的应用程序应改用相应的编程接口。这种检查 Solaris ABI 使用情况的能力使您能够轻松快速地识别潜在的接口问题。

运行 apptrace

apptrace 实用程序不要求对所跟踪的应用程序进行任何修改。要使用 apptrace，请键入 apptrace，后跟所需的任何选项以及用于运行有用的应用程序的命令行。apptrace 实用程序运行时可使用运行时链接程序的链接审计功能来拦截应用程序对 Solaris 库接口的调用。然后，apptrace 实用程序会通过显示调用参数的名称和值来跟踪调用并返回值。跟踪输出可以显示为一行，也可以分为多行以便于阅读。公共接口按人工可读的形式进行显示。专用接口以十六进制进行显示。

apptrace 实用程序允许有选择性跟踪调用，既可以跟踪各接口级调用，也可以跟踪库级调用。例如，apptrace 可以跟踪来自 libnsl 的 printf() 调用，也可以跟踪特定库内某个范围的调用。apptrace 实用程序还可以详细跟踪用户指定的调用。指示 apptrace

行为的规范会通过语法进行制约，此语法与 truss(1) 的用法一致。-f 选项用于指示 apptrace 遵循派生的子进程。-o 选项用于指定存储 apptrace 结果的输出文件。

apptrace 实用程序仅跟踪库级调用，并可装入运行的应用程序进程中，从而可以提高性能（比使用 truss 时的性能要高）。但使用 printf 则除外，在这种情况下 apptrace 无法跟踪对接受变量列表的函数的调用，也无法检查栈或其他调用者信息，例如 setcontext、getcontext、setjmp、longjmp 和 vfork。

解释 apptrace 输出

以下示例包含跟踪简单的单二进制应用程序 ls 所产生的 apptrace 输出样例。

示例 13–1 缺省跟踪行为

```
% apptrace ls /etc/passwd
ls       -> libc.so.1:atexit(func = 0xff3cb8f0) = 0x0
ls       -> libc.so.1:atexit(func = 0x129a4) = 0x0
ls       -> libc.so.1:getuid() = 0x32c3
ls       -> libc.so.1:time(tloc = 0x23918) = 0x3b2fe4ef
ls       -> libc.so.1:isatty(fildes = 0x1) = 0x1
ls       -> libc.so.1:ioctl(0x1, 0x540d, 0xffbff7ac)
ls       -> libc.so.1:ioctl(0x1, 0x5468, 0x23908)
ls       -> libc.so.1:setlocale(category = 0x6, locale = "") = "C"
ls       -> libc.so.1:calloc(nelem = 0x1, elsize = 0x40) = 0x23cd0
ls       -> libc.so.1:lstat64(path = "/etc/passwd", buf = 0xffbff6b0) = 0x0
ls       -> libc.so.1:acl(pathp = "/etc/passwd", cmd = 0x3, nentries = 0x0,
             aclbufp = 0x0) = 0x4
ls       -> libc.so.1:qsort(base = 0x23cd0, nel = 0x1, width = 0x40,
             compar = 0x12038)
ls       -> libc.so.1:sprintf(buf = 0x233d0, format = 0x12af8, ...) = 0
ls       -> libc.so.1:strlen(s = "") = 0x0
ls       -> libc.so.1:strlen(s = "/etc/passwd") = 0xb
ls       -> libc.so.1:sprintf(buf = 0x233d0, format = 0x12af8, ...) = 0
ls       -> libc.so.1:strlen(s = "") = 0x0
ls       -> libc.so.1:printf(format = 0x12ab8, ...) = 11
ls       -> libc.so.1:printf(/etc/passwd
format = 0x12abc, ...) = 1
ls       -> libc.so.1:exit(status = 0)
```

以上示例显示了缺省跟踪行为，即跟踪 ls /etc/passwd 命令的每个库调用。apptrace 实用程序针对每个系统调用显示一行输出，指明以下信息：

- 调用的名称
- 调用所在的库
- 调用的参数和返回值

ls 的输出与 apptrace 输出混在一起。

示例 13–2 选择性跟踪

```
% apptrace -t \*printf ls /etc/passwd
ls       -> libc.so.1:sprintf(buf = 0x233d0, format = 0x12af8, ...) = 0
```

示例 13–2 选择性跟踪 （续）

```
ls       -> libc.so.1:sprintf(buf = 0x233d0, format = 0x12af8, ...) = 0
ls       -> libc.so.1:printf(format = 0x12ab8, ...) = 11
ls       -> libc.so.1:printf(/etc/passwd
format = 0x12abc, ...) = 1
```

以上示例说明了 apptrace 如何能够有选择性地跟踪使用正则表达式语法的调用。在此示例中，在 printf（包括 sprintf）中结束的接口调用与以前一样，都是在 ls 命令中进行跟踪。因此，apptrace 仅跟踪 printf 和 sprintf 调用。

示例 13–3 详细跟踪

```
% apptrace -v sprintf ls /etc/passwd
ls       -> libc.so.1:sprintf(buf = 0x233d0, format = 0x12af8, ...) = 0
  buf =    (char *) 0x233d0 ""
  format = (char *) 0x12af8 "%s%s%s"
ls       -> libc.so.1:sprintf(buf = 0x233d0, format = 0x12af8, ...) = 0
  buf =    (char *) 0x233d0 ""
  format = (char *) 0x12af8 "%s%s%s"
/etc/passwd
```

以上示例显示了详细跟踪模式，在这种模式下，sprintf 的参数会显示在多个输出行中以便于阅读。最后，apptrace 会显示 ls 命令的输出。

◆ ◆ ◆ 附 录 A

UNIX 域套接字

UNIX 域套接字以 UNIX 路径命名。例如，可以将套接字命名为 `/tmp/foo`。UNIX 域套接字只在一台主机上的进程之间通信。UNIX 域中的套接字不会被视为网络协议的一部分，因为它们只能用于在一台主机上的进程之间通信。

套接字类型定义对于用户可见的通信属性。Internet 域套接字提供对 TCP/IP 传输协议的访问。Internet 域由值 `AF_INET` 标识。套接字仅与同一域中的套接字交换数据。

创建套接字

`socket(3SOCKET)` 调用创建指定系列和指定类型的套接字。

```
s = socket(family, type, protocol);
```

如果未指定协议（值 `0`），则系统将选择支持所需套接字类型的协议。将返回套接字句柄（文件描述符）。

系列由 `sys/socket.h` 中定义的常量之一指定。名为 `AF_`*suite* 的常量指定要在解释名称中使用的地址格式。

下面创建了在计算机内部使用的数据报套接字：

```
s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

在大多数情况下，请将 *protocol* 参数设置为 `0`（即缺省协议）。

本地名称绑定

创建套接字时不指定名称。只有在套接字绑定到地址之后，远程进程才能引用此套接字。通信进程通过地址连接。在 UNIX 系列中，连接通常包括一个或两个路径名。UNIX 系列套接字无需始终绑定到名称。如果它们始终绑定到名称，则从不会存在绑定的重复排序集合（如 `local pathname` 或 `foreign pathname`）。路径名不能引用现有文件。

通过 `bind(3SOCKET)` 调用，进程可以指定套接字的本地地址。这样便可创建 `local pathname` 排序集合，而 `connect(3SOCKET)` 和 `accept(3SOCKET)` 可通过修复地址的远程部分来完成套接字的关联。可以按如下方式使用 `bind(3SOCKET)`：

```
bind (s, name, namelen);
```

套接字句柄为 *s*。绑定名称是由支持协议解释的字节字符串。UNIX 系列名称包含一个路径名和一个系列。本示例说明如何将名称 `/tmp/foo` 绑定到 UNIX 系列套接字。

```
#include <sys/un.h>
 ...
struct sockaddr_un addr;
 ...
strcpy(addr.sun_path, "/tmp/foo");
addr.sun_family = AF_UNIX;
bind (s, (struct sockaddr *) &addr,
        strlen(addr.sun_path) + sizeof (addr.sun_family));
```

确定 `AF_UNIX` 套接字地址的大小时不计空字节，因此可以使用 `strlen(3C)`。

`addr.sun_path` 中引用的文件名在系统文件名称空间中创建为套接字。调用者必须对创建 `addr.sun_path` 的目录具有写入权限。不再需要文件时，调用者应将其删除。可使用 `unlink(1M)` 删除 `AF_UNIX` 套接字。

建立连接

通常以非对称形式建立连接。一个进程用作客户机，而另一个进程则用作服务器。服务器将套接字绑定到与服务关联的已知地址，并阻塞在套接字上等待连接请求。然后，不相关的进程便可连接到此服务器。客户机通过启动到服务器套接字的连接，向服务器请求服务。在客户机端，`connect(3SOCKET)` 调用启动连接。在 UNIX 系列中，此连接过程可能如下所示：

```
struct sockaddr_un server;
        server.sun.family = AF_UNIX;
         ...
        connect(s, (struct sockaddr *)&server, strlen(server.sun_path)
         + sizeof (server.sun_family));
```

有关连接错误的信息，请参见第 129 页中的 “连接错误”。第 129 页中的 “数据传输”介绍了如何传输数据。第 130 页中的 “关闭套接字”介绍了如何关闭套接字。

索引

S

T

U

V

X

Z

版

绑

数

锁

套

同

网