Solaris 10 资源管理器开发者指南

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

文件号码 819–7053–10
2006 年 11 月

061229@15821

目录

前言

《Solaris 10 资源管理器开发者指南》介绍了如何编写用于对系统资源进行分区和管理的应用程序，并讨论了要使用的 API。本书还提供了编程示例，并对编写应用程序时要考虑的编程问题进行了讨论。

目标读者

本书适用于编写用于控制和监视 Solaris 操作系统资源的应用程序的应用程序开发者和 ISV。

阅读本书之前

有关资源管理的详细概述，请参见《系统管理指南：Solaris Containers－资源管理和 Solaris Zones》。

本书的结构

《Solaris 10 资源管理器开发者指南》的结构如下：

第 1 章介绍了 Solaris 10 Resource Manager 产品。

第 2 章提供了有关项目和任务功能的信息。

第 3 章介绍了用于扩展记帐功能的 C 接口。

第 4 章介绍了用于扩展记帐功能的 Perl 接口。

第 5 章讨论了资源控制及其使用。

第 6 章介绍了动态资源池。

第 7 章介绍了为使应用程序在 Solaris 区域中工作需要采取的预防措施。

第 8 章提供了 `/etc/project` 文件的配置示例。

文档、支持和培训

Sun 提供的服务	URL	说明
文档	http://www.sun.com/documentation/	下载 PDF 及 HTML 格式的文档，购买印刷文档
支持和培训	http://www.sun.com/supportraining/	获取技术支持、下载修补程序，以及学习 Sun 提供的课程

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	执行**修补程序分析**。 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

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

表 P–2 Shell 提示符

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

◆ ◆ ◆ 第 1 章

1 Solaris 操作系统中的资源管理

本手册旨在帮助开发者编写以下两种应用程序：管理计算机资源的实用程序，以及能够检查自身使用情况并根据情况进行相应调整的自行监视应用程序。本章提供了有关 Solaris 操作系统 (Operating System, OS) 中的资源管理的说明。本章包括以下主题：

了解 Solaris OS 中的资源管理

资源管理的主要作用是平衡服务器上的工作负荷，以使系统可以高效地工作。如果没有很好地管理资源，超越工作负荷就会出现故障而使进度停止，导致优先级作业发生不必要的延迟。另一个作用是有效的资源管理使组织可以通过整合服务器有效地利用资源。为了能够进行资源管理，Solaris OS 提供了一种可以组织工作负荷和资源的结构，而且在确定某个特定的工作负荷单元可占用的资源数量时可以进行控制。有关从系统管理员角度对资源管理的深入讨论，请参见《系统管理指南：Solaris Containers－资源管理和 Solaris Zones》中的第 1 章 “Solaris 10 Resource Manager 介绍”。

工作负荷组织

工作负荷的基本单位是**进程**。进程 ID (Process ID, PID) 在整个系统中按顺序进行编号。缺省情况下，系统管理员会将每个用户都分配到**项目**（即网络范围的管理标识符）中。每次成功登录项目后，都会创建一个新的**任务**（即进程的分组机制）。任务包含登录进程以及后续的子进程。

有关项目和任务的更多信息，请参见《系统管理指南：Solaris Containers－资源管理和 Solaris Zones》中的第 2 章 “项目和任务（概述）”（针对系统管理员角度）或本书中的第 2 章（针对开发者角度）。

可以有选择性地将项目分组到**区域** (zone)中，区域是由管理员出于安全考虑设置的，目的是隔离各组用户。可以将区域看作一个盒子，其中一个或多个应用程序与系统中的

所有其他应用程序相互隔离，独立运行。《系统管理指南：Solaris Containers－资源管理和 Solaris Zones》中的第 II 部分, “Zones”详细的讨论了 Solaris Zones。有关编写在区域中运行的资源管理应用程序时需要了解的特别注意事项的更多信息，请参见第 7 章。

资源组织

系统管理员可以将工作负荷分配给系统中的特定 CPU 或定义的多组 CPU。可以将 CPU 分组到**处理器集**（也称为 *pset*）中。而 `pset` 又可以与一个或多个线程调度类（定义 CPU 优先级）联结，最终形成**资源池**。资源池为系统管理员提供了一种使用户可以使用系统资源的便捷机制。《系统管理指南：Solaris Containers－资源管理和 Solaris Zones》中的第 12 章 “动态资源池（概述）”为系统管理员介绍了资源池。第 6 章介绍了编程注意事项。

下图说明了 Solaris OS 中的工作负荷和计算机资源的组织方式。

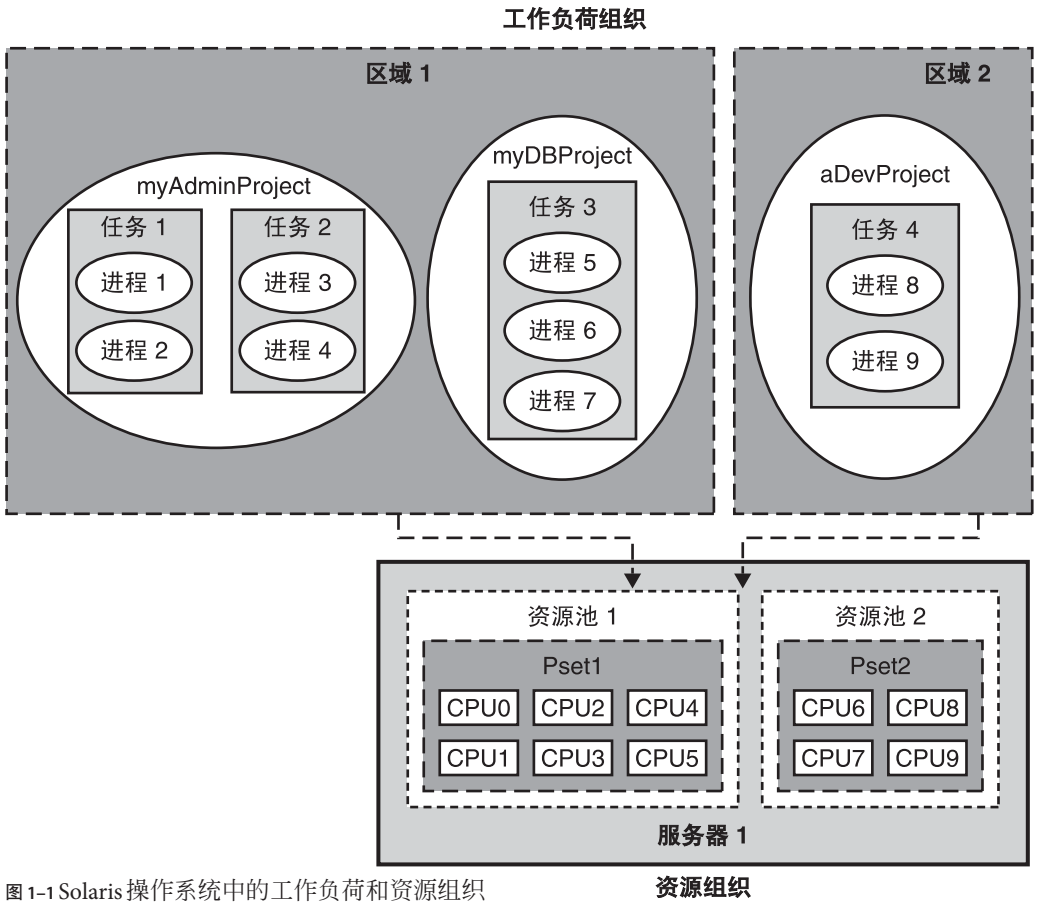


图 1–1 Solaris 操作系统中的工作负荷和资源组织

资源控制

对于管理用户所占用的资源量而言，只将工作负荷单位分配给资源单位是不够的。为了管理资源，Solaris OS 提供了一组标志、操作和信号，它们统称为**资源控制**并存储在 `/etc/project` 文件中。例如，公平共享调度程序 (Fair Share Scheduler, FSS) 可以根据工作负荷的指定重要因素在工作负荷中分配 CPU 资源的份额。使用这些资源控制，系统管理员可以设置权限级别并限制对特定项目、任务或进程的定义。要了解系统管理员如何使用资源控制，请参见《系统管理指南：Solaris Containers－资源管理和 Solaris Zones》中的第 6 章“资源控制（概述）”。有关编程注意事项，请参见第 5 章。

扩展记帐功能

除了工作负荷和资源组织以外，Solaris OS 还提供了用于监视和记录系统资源使用情况的**扩展记帐功能**。扩展记帐功能为系统管理员提供了一组有关进程和任务的详细资源占用统计信息。

《系统管理指南：Solaris Containers－资源管理和 Solaris Zones》中的第 4 章 “扩展记帐（概述）”为系统管理员深入介绍了此功能。Solaris OS 为开发者提供了用于访问扩展记帐功能的 C 接口和 PERL 接口。有关 C 接口，请参阅第 3 章，有关 PERL 接口，请参阅第 4 章。

编写资源管理应用程序

本手册着重从开发者角度介绍资源管理，并提供用于编写以下各种应用程序的信息：

- 资源管理应用程序－用于执行如分配资源、创建分区及调度作业等任务的实用程序。
- 资源监视应用程序－用于通过 `kstats` 检查系统统计信息以根据系统、工作负荷、进程和用户确定资源使用情况的应用程序。
- 资源记帐实用程序－用于为分析、计费和容量规划提供记帐信息的应用程序。
- 自行调整应用程序－可确定资源使用并且可以在必要时调整占用情况的应用程序。
- 资源建议应用程序－提供资源需求提示。

◆ ◆ ◆ 第 2 章

项目和任务

本章介绍了工作负荷分层结构，并且提供了有关项目和任务的信息。本章包含以下主题：

项目和任务概述

Solaris OS 使用工作负荷分层结构组织系统中所执行的工作。**任务**是指表示工作负荷组件的进程集合。**项目**是指表示整个工作负荷的任务集合。在任何给定的时间，进程都只能是一个任务和一个项目的组件。下图说明了工作负荷分层结构中的关系。

图 2–1 工作负荷分层结构

作为多个项目成员的用户可以同时运行多个项目中的进程。由某个进程启动的所有进程都会继承父进程的项目。在启动脚本中切换到新项目时，所有子进程都会在该新项目中运行。

正在执行的用户进程有关联的用户标识 (`uid`)、组标识 (`gid`) 和项目标识 (`projid`)。进程属性和功能是从用户、组和项目标识继承而来，从而形成任务的执行环境。

有关项目和任务的深入讨论，请参见《系统管理指南：Solaris Containers－资源管理和 Solaris Zones》中的第 2 章 “项目和任务（概述）”。有关用于管理项目和任务的管理命令，请参见《系统管理指南：Solaris Containers－资源管理和 Solaris Zones》中的第 3 章 “管理项目和任务”。

/etc/project 文件

project 文件是工作负荷分层结构的核心。project 数据库是在系统中通过 /etc/project 文件或在网络上通过名称服务（如 NIS 或 LDAP）来维护的。

/etc/project 文件包含五个标准项目。

system	此项目用于所有系统进程和守护进程。
user.root	所有的根进程都在 user.root 项目中运行。
noproject	此特殊项目用于 IPQoS。
default	缺省项目会指定给每个用户。
group.staff	此项目用于 staff 组中的所有用户。

要通过编程方式访问项目文件，请使用以下结构：

```
struct project {

  char      *pj_name;       /* name of the project */

  projid_t   pj_projid;     /* numerical project ID */

  char      *pj_comment;    /* project comment */

  char     **pj_users;      /* vector of pointers to project user names */

  char     **pj_groups;     /* vector of pointers to project group names */

  char      *pj_attr;       /* project attributes */

};
```

project 结构成员包括以下各项：

*pj_name
  项目的名称。

pj_projid
  项目 ID。

*pj_comment
  用户提供的项目说明。

**pj_users
  指向项目用户成员的指针。

**pj_groups
  指向项目组成员的指针。

*pj_attr
项目属性。使用这些属性可为资源控制和项目池设置值。

通过项目属性，可以控制资源的使用情况。可以使用四种前缀对各种类型的资源控制属性进行分组：

- project.*－此前缀表示用于控制项目的属性。例如，project.max-device-locked-memory 可指明所允许的锁定内存总量（以字节数表示）。project.pool 属性将项目绑定到资源池。请参见第 6 章。
- task.*－此前缀用于应用于任务的属性。例如，task.max-cpu-time 属性设置此任务进程可用的最长 CPU 时间（以秒数表示）。
- process.*－此前缀用于进程控制项。例如，process.max-file-size 控制项设置可由此进程写入的最大文件偏移量（以字节数表示）。
- zone.*－zone.* 前缀适用于区域中的项目、任务和进程。例如，zone.max-lwps 可以防止一个区域中有过多 LWP 影响其他区域。可以使用 project.max-lwps 的各项在区域内的各个项目之间进一步细分区域的所有 LWP。

有关资源控制的完整列表，请参见 resource_controls(5)。

项目和任务的 API 函数

以下函数用于协助开发者处理项目。这些函数使用的各项用于描述 project 数据库中的用户项目。

endprojent(3PROJECT)	处理完成后，关闭项目数据库并取消分配资源。
fgetprojent(3PROJECT)	返回一个指针，该指针指向包含项目数据库中的项的结构。fgetprojent() 从流中读取行，而不是使用 nsswitch.conf 读取行。
getdefaultproj(3PROJECT)	检查项目关键字的有效性，查找项目并返回指向项目结构（如果找到）的指针。
getprojbyid(3PROJECT)	搜索 project 数据库中带有指定项目 ID 的数字的项。
getprojbyname(3PROJECT)	搜索 project 数据库中带有指定项目名称的字符串的项。
getprojent(3PROJECT)	返回一个指针，该指针指向包含项目数据库中的项的结构。
inproj(3PROJECT)	检查是否允许指定的用户使用指定的项目。
setproject(3PROJECT)	将用户进程添加到项目中。
setprojent(3PROJECT)	反绕 project 数据库，以允许重复搜索。

用于访问 project 数据库各项的代码示例

示例 2–1 列显 project 数据库中每项的前三个字段

本示例的要点包括以下内容：

- setprojent() 开始会反绕 project 数据库来进行启动。
- getprojent() 是使用 project.h 中定义的保守最大缓冲区大小来调用的。
- endprojent() 用于关闭 project 数据库并释放资源。

```
#include <project.h>



struct project projent;

char buffer[PROJECT_BUFSZ]; /* Use safe buffer size from project.h */

    ...

struct project *pp;



setprojent();  /* Rewind the project database to start at the beginning */



while (1) {

   pp = getprojent(&projent, buffer, PROJECT_BUFSZ);

      if (pp == NULL)

          break;

    printf("%s:%d:%s\n", pp->pj_name, pp->pj_projid, pp->pj_comment);

        ...

};



endprojent();   /* Close the database and free project resources */
```

示例 2–2 获取与调用方的项目 ID 匹配的 `project` 数据库项

以下示例调用 `getprojbyid()` 来获取与调用方的项目 ID 匹配的项目数据库项。然后，该示例将列显项目名称和项目 ID。

```
#include <project.h>



struct project *pj;

char buffer[PROJECT_BUFSZ]; /* Use safe buffer size from project.h */



main()

{

   projid_t pjid;

   pjid = getprojid();

   pj = getprojbyid(pjid, &projent, buffer, PROJECT_BUFSZ);

   if (pj == NULL) {

       /* fail; */

   }

   printf("My project (name, id) is (%s, %d)\n", pp->pj_name, pp->pj_projid);

}
```

与项目和任务关联的编程问题

编写应用程序时，请注意以下问题：

- 不存在可以显式创建新项目的函数。
- 如果 `project` 数据库中不存在用户的缺省项目，则该用户无法登录。
- 用户登录时，系统会在该用户的缺省项目中创建新任务。
- 进程与新项目的关联会将新项目的资源控制和池成员关系应用于进程。

- setproject() 需要权限。如果您拥有进程，则 newtask 命令不需要权限。虽然二者均可用于创建任务，但仅有 newtask 可以更改运行的进程对应的项目。
- 任务之间不存在任何父/子关系。
- 可以使用 newtask -F 或setproject() 来创建最终任务，从而将调用方与新项目关联。尝试准确估计总体资源记帐时，最终任务非常有用。
- 可重复执行函数 getprojent()、getprojbyname()、getprojbyid()、getdefaultproj() 和 inproj() 会使用调用方提供的缓冲区来存储返回的结果。在单线程应用程序和多线程应用程序中可以安全地使用这些函数。
- 可重复执行函数需要以下附加参数： proj、buffer 和 bufsize。proj 参数必须是指向调用方所分配的 project 结构的指针。成功完成后，这些函数将返回此结构中的项目项。project 结构所引用的存储空间是从 buffer 参数指定的内存中分配的。bufsize 用于指定大小（以字节数表示）。
- 如果使用不正确的缓冲区大小，则 getprojent() 会返回 NULL，并将 errno 设置为 ERANGE。

◆ ◆ ◆ 第 3 章

3 使用用于扩展记帐的 C 接口

本章介绍用于扩展记帐的 C 接口，包含以下主题：

用于扩展记帐的 C 接口概述

项目和任务用于标记和分隔工作负荷。扩展记帐子系统用于监视系统中正在运行的工作负荷的资源占用情况。扩展记帐将生成工作负荷任务和进程的记帐记录。

有关扩展记帐和扩展记帐管理过程示例的概述，请参见《系统管理指南：Solaris Containers－资源管理和 Solaris Zones》中的第 4 章 “扩展记帐（概述）”和《系统管理指南：Solaris Containers－资源管理和 Solaris Zones》中的第 5 章 “管理扩展记帐（任务）”。

扩展记帐 API 函数

扩展记帐 API 包含用于执行以下操作的函数：

- exacct 系统调用
- 对 exacct 文件执行的操作
- 对 exacct 对象执行的操作
- 杂项操作

exacct 系统调用

下表列出了与扩展记帐子系统交互的系统调用。

表 3–1 扩展记帐系统调用

函数	说明
putacct(2)	使具有权限的进程能够使用特定于进程的其他数据来标记记帐记录
getacct(2)	使具有权限的进程能够从当前执行任务和进程的内核中请求扩展记帐缓冲区
wracct(2)	请求内核为指定的任务或进程写入资源使用情况的数据

对 exacct 文件执行的操作

以下函数提供对 exacct 文件的访问：

表 3–2 exacct 文件函数

函数	说明
ea_open(3EXACCT)	打开 exacct 文件。
ea_close(3EXACCT)	关闭 exacct 文件。
ea_get_object(3EXACCT)	首次对一组对象使用此函数会将数据读入 ea_object_t 结构。随后对该组使用此函数则会循环处理该组中的对象。
ea_write_object(3EXACCT)	将指定的对象附加到打开的 exacct 文件中。
ea_next_object(3EXACCT)	将基本字段（eo_catalog 和 eo_type）读入 ea_object_t 结构中，并反绕到记录头。
ea_previous_object(3EXACCT)	在 exacct 文件中往回跳一个对象，并将基本字段（eo_catalog 和 eo_type）读入 ea_object_t 中。
ea_get_hostname(3EXACCT)	获取在其上创建 exacct 文件的主机名。
ea_get_creator(3EXACCT)	确定 exacct 文件的创建者。

对 exacct 对象执行的操作

以下函数用于访问 exacct 对象：

表 3–3 exacct 对象函数

函数	说明
ea_set_item(3EXACCT)	指定 exacct 对象并设置值。
ea_set_group(3EXACCT)	设置一组 exacct 对象的值。

表 3–3 exacct 对象函数 （续）

函数	说明
ea_match_object_catalog(3EXACCT)	检查 exacct 对象的掩码，以了解该对象是否具有特定目录标记。
ea_attach_to_object(3EXACCT)	将 exacct 对象附加到指定的 exacct 对象中。
ea_attach_to_group(3EXACCT)	将 exacct 对象链作为指定组的成员项进行附加。
ea_free_item(3EXACCT)	在指定的 exacct 对象中释放 value 字段。
ea_free_object(3EXACCT)	释放指定的 exacct 对象以及任何附加的对象分层结构。

杂项操作

以下函数与杂项操作相关联：

ea_error(3EXACCT)
ea_match_object_catalog(3EXACCT)

访问 exacct 文件的 C 代码示例

本节提供访问 exacct 文件的代码示例。

示例 3–1 显示与指定的 pid 对应的 exacct 数据

本示例显示内核中特定 pid 的 exacct 数据快照。

```
...

  ea_object_t *scratch;

  int unpk_flag = EUP_ALLOC;  /* use the same allocation flag */

                              /* for unpack and free */


  /* Omit return value checking, to keep code samples short */


  bsize = getacct(P_PID, pid, NULL, 0);

  buf = malloc(bsize);
```

示例 3–1 显示与指定的 pid 对应的 exacct 数据 （续）

```
  /* Retrieve exacct object and unpack */
  getacct(P_PID, pid, buf, bsize);
  ea_unpack_object(&scratch, unpk_flag, buf, bsize);

  /* Display the exacct record */
  disp_obj(scratch);
  if (scratch->eo_type == EO_GROUP) {
         disp_group(scratch);
  }
  ea_free_object(scratch, unpk_flag);
        ...
```

示例 3–2 确定内核生成期间的各项任务

本示例评估了内核生成并显示了描述此 make 任务要生成的那部分源代码树的字符串。显示正在生成的且在按源目录的分析中有帮助的源代码部分。

本示例的要点包括以下内容：

- 要聚合 make（可以包括许多进程）的时间，可以将每个 make 作为任务来启动。make 子进程是作为不同的任务来创建的。要跨越 makefile 树进行聚合，必须确定父子任务关系。
- 将包含此信息的标记添加到任务的 exacct 文件中。添加描述此 make 任务要生成的那部分源代码树的当前工作目录字符串。

```
  ea_set_item(&cwd, EXT_STRING | EXC_LOCAL | MY_CWD,
                                  cwdbuf, strlen(cwdbuf));
  ...
/* Omit return value checking and error processing */
```

示例 3–2 确定内核生成期间的各项任务 （续）

```
/* to keep code sample short */

ptid = gettaskid();    /* Save "parent" task-id */

tid = settaskid(getprojid(), TASK_NORMAL);    /* Create new task */


/* Set data for item objects ptskid and cwd */

ea_set_item(&ptskid, EXT_UINT32 | EXC_LOCAL | MY_PTID, &ptid, 0);

ea_set_item(&cwd, EXT_STRING | EXC_LOCAL | MY_CWD, cwdbuf, strlen(cwdbuf));


/* Set grp object and attach ptskid and cwd to grp */

ea_set_group(&grp, EXT_GROUP | EXC_LOCAL |  EXD_GROUP_HEADER);

ea_attach_to_group(&grp, &ptskid);

ea_attach_to_group(&grp, &cwd);


/* Pack the object and put it back into the accounting stream */

ea_buflen = ea_pack_object(&grp, ea_buf, sizeof(ea_buf));

putacct(P_TASKID, tid, ea_buf, ea_buflen, EP_EXACCT_OBJECT);


/* Memory management: free memory allocate in ea_set_item */

ea_free_item(&cwd, EUP_ALLOC);

  ...
```

示例 3–3 读取和显示系统 exacct 文件的内容

本示例显示如何读取并显示进程或任务的系统 exacct 文件。

本示例的要点包括以下内容：

示例 3–3 读取和显示系统 exacct 文件的内容 （续）

- 调用 ea_get_object() 以获取该文件中的下一个对象。调用 ea_get_object()，循环遍历 exacct 文件，直到遇见 EOF。
- catalog_name() 使用 catalog_item 结构将 Solaris 目录的类型 ID 转换为有意义的字符串，该字符串描述了该对象数据的内容。该类型 ID 是通过屏蔽最低的 24 位（即 3 个字节）获取的。

```
switch(o->eo_catalog & EXT_TYPE_MASK) {

  case EXT_UINT8:

      printf(" 8: %u", o->eo_item.ei_uint8);

      break;

  case EXT_UINT16:

  ...

}
```

- TYPE_MASK 的前 4 位用于查找列显该对象的实际数据的数据类型。
- disp_group() 将使用指向组对象的指针及该组中的对象个数作为其参数。对于组中的每个对象，disp_group() 将调用 disp_obj() 并递归调用 disp_group()（如果该对象是组对象）。

```
/* Omit return value checking and error processing */

/* to keep code sample short */

main(int argc, char *argv)

{

  ea_file_t ef;

  ea_object_t scratch;

  char *fname;



  fname = argv[1];

  ea_open(&ef, fname, NULL,  EO_NO_VALID_HDR, O_RDONLY, 0);
```

示例 3–3 读取和显示系统 exacct 文件的内容 （续）

```
  bzero(&scratch, sizeof (ea_object_t));
  while (ea_get_object(&ef, &scratch)  != -1) {
         disp_obj(&scratch);
         if (scratch.eo_type == EO_GROUP)
             disp_group(&ef, scratch.eo_group.eg_nobjs);
         bzero(&scratch, sizeof (ea_object_t));
  }
  ea_close(&ef);
}

struct catalog_item {   /* convert Solaris catalog's type ID */
                        /* to a meaningful string */
     int    type;
     char *name;
 } catalog[] = {
     { EXD_VERSION,     "version\t" },
     ...
     { EXD_PROC_PID,    "  pid\t" },
     ...
   };

 static char *
 catalog_name(int type)
```

示例 3–3 读取和显示系统 exacct 文件的内容 （续）

```
{
    int i = 0;
    while (catalog[i].type != EXD_NONE) {
        if (catalog[i].type == type)
            return (catalog[i].name);
        else
            i++;
    }
    return ("unknown\t");
}


static void disp_obj(ea_object_t *o)
{
    printf("%s\t", catalog_name(o->eo_catalog & 0xffffff));
    switch(o->eo_catalog & EXT_TYPE_MASK) {
    case EXT_UINT8:
        printf(" 8: %u", o->eo_item.ei_uint8);
        break;
    case EXT_UINT16:
    ...
}
static void disp_group(ea_file_t *ef, uint_t nobjs)
{
```

示例 3–3 读取和显示系统 exacct 文件的内容 （续）

```
    for (i = 0; i < nobjs; i++) {

        ea_get_object(ef, &scratch));

                 disp_obj(&scratch);

        if (scratch.eo_type == EO_GROUP)

               disp_group(ef, scratch.eo_group.eg_nobjs);

       }

}
```

◆ ◆ ◆ 第 4 章

使用用于扩展记帐的 Perl 接口

Perl 接口可提供与扩展记帐任务和项目的 Perl 绑定。使用该接口，Perl 脚本可读取 exacct 框架所生成的记帐文件，还可以编写 exacct 文件。

本章包含以下主题：

扩展记帐概述

exacct 是 Solaris 操作环境的一种新记帐框架，其中除了提供传统 SVR4 记帐机制所提供的功能以外，还提供其他功能。传统的 SVR4 记帐具有以下缺点：

- 不能修改 SVR4 记帐所收集的数据。

  不能针对每个应用程序自定义 SVR4 记帐所收集的统计信息的类型或数量。SVR4 记帐所收集的数据更改不适用于使用记帐文件的所有现有应用程序。
- SVR4 记帐机制不是开放的。

  应用程序无法在系统记帐数据流中嵌入各自的数据。
- SVR4 记帐机制不具备聚合功能。

  Solaris 操作系统会为每个存在的进程写入单独的记录。未提供任何用于将记帐记录集分组为更高级别聚合的功能。

exacct 框架解除了 SVR4 记帐的限制，并且为记帐数据集合提供了可配置、开放且可扩展的框架。

- 可以使用 exacct API 配置收集到的数据。
- 应用程序既可以在系统记帐文件中嵌入各自的数据，也可以创建和处理各自的自定义记帐文件。

- 传统记帐机制中缺少数据聚合功能的问题由**任务**和**项目**加以解决。任务标识作为工作单元的一组进程。通过项目，可将一组用户执行的进程聚集为更高级别的实体。有关任务和项目的更多详细信息，请参见 project(4) 手册页。

有关扩展记帐的更详细概述，请参见《系统管理指南：Solaris Containers－资源管理和 Solaris Zones》中的第 4 章 “扩展记帐（概述）”。

用于 libexacct 的 Perl 接口

对象模型

Sun::Solaris::Exacct 模块是由 libexacct(3LIB) 库所提供的所有类的父类。libexacct(3LIB) 提供对以下各种类型的实体的操作：exacct 格式文件、catalog 标记和 exacct 对象。exacct 对象会细分为两种类型。

- 项

  单个数据值
- 组

  项列表

使用用于 libexacct 的 Perl 接口的优点

扩展记帐的 Perl 扩展可为基础 libexacct(3LIB) API 提供 Perl 接口以及以下增强功能。

- 与 C API 完全等效，可提供在功能上与基础 C API 等效的 Perl 接口。

  该接口提供了一种无需进行 C 编码即可访问的 exacct 文件的机制。所有在 C 中可以使用的功能也能借助 Perl 接口使用。
- 易于使用。

  从基础 C API 获取的数据会表示为 Perl 数据类型。Perl 数据类型使数据访问更加容易，并且无需进行缓冲区压缩和解压缩操作。
- 自动内存管理。

  C API 要求程序员在访问 exacct 文件时负责管理内存。内存管理采用的形式是将相应的标志传递给函数（如 ea_unpack_object(3EXACCT)）并显式分配传递给 API 的缓冲区。Perl API 取消了这些要求，因为所有的内存管理都由 Perl 库执行。
- 防止错误地使用 API。

  ea_object_t 结构提供了 exacct 记录在内存中的表示形式。ea_object_t 结构是用于处理组和项记录的联合类型。因此，类型不正确的结构可能会被传递给某些 API 函数。通过添加类分层结构可防止此类型的编程错误。

Perl 复合型标量 (double-typed scalar)

本文档中介绍的模块广泛使用 Perl 复合型标量功能。使用**复合型标量**功能可将标量值作为整数或字符串，具体取决于上下文。此行为与 `$!` Perl 变量 (`errno`) 表现的行为相同。复合型标量功能无需从整数值映射为对应的字符串即可显示值。以下示例说明了复合型标量的使用。

```
# Assume $obj is a Sun::Solaris::Item

my $type = $obj->type();



# prints out "2 EO_ITEM"

printf("%d %s\n", $type, $type);



# Behaves as an integer, $i == 2

my $i = 0 + $type;



# Behaves as a string, $s = "abc EO_ITEM xyx"

my $s = "abc $type xyz";
```

Perl 模块

各种与项目、任务和 `exacct` 相关的函数已分为多个组，并且每个组都置于单独的 Perl 模块中。每个函数都具有 Sun Microsystems 标准的 `Sun::Solaris::` Perl 软件包前缀。

表 4–1 Perl 模块

模块	说明
第 35 页中的 “Sun::Solaris::Project 模块”	提供用于访问以下项目操作函数的功能：getprojid(2)、setproject(3PROJECT)、project_walk(3PROJECT)、getprojent(3PROJECT)、getprojbyname(3PROJECT)、getprojbyid(3PROJECT)、getdefaultproj(3PROJECT)、inproj(3PROJECT)、getprojidbyname(3PROJECT)、setprojent(3PROJECT)、endprojent(3PROJECT)、fgetprojent(3PROJECT)。
第 36 页中的 “Sun::Solaris::Task 模块”	提供用于访问任务操作函数 settaskid(2) 和 gettaskid(2) 的功能。
第 37 页中的 “Sun::Solaris::Exacct 模块”	顶层 exacct 模块。此模块中的函数可访问与 exacct 有关的系统调用 getacct(2)、putacct(2) 和 wracct(2) 以及 libexacct(3LIB) 库函数 ea_error(3EXACCT)。此模块包含各种 exacct EO_*、EW_*、EXR_*、P_* 和 TASK_* 宏的所有常量。
第 38 页中的 “Sun::Solaris::Exacct::Catalog 模块”	提供面向对象的方法，用于访问 exacct 目录标记中的位字段，以及 EXC_*、EXD_* 和 EXD_* 宏。
第 40 页中的 “Sun::Solaris::Exacct::File 模块”	提供面向对象的方法，用于访问以下 libexacct(3LIB) 记帐文件函数：ea_open(3EXACCT)、ea_close(3EXACCT)、ea_get_creator(3EXACCT)、ea_get_hostname(3EXACCT)、ea_next_object(3XACCT)、ea_previous_object(3EXACCT)、ea_write_object(3EXACCT)。
第 42 页中的 “Sun::Solaris::Exacct::Object 模块”	提供面向对象的方法，用于访问单个 exacct 记帐文件对象。exacct 对象表示为指定属于相应 Sun::Solaris::Exacct::Object 子类的不透明引用。此模块可细分为两种可能的对象类型：项和组。另外，还提供了用于访问 ea_match_object_catalog(3EXACCT)、ea_attach_to_object(3EXACCT) 函数的方法。

表 4–1 Perl 模块 （续）

模块	说明
第 43 页中的 “Sun::Solaris::Exacct::Object::Item 模块”	提供面向对象的方法，用于访问单个 exacct 记帐文件项。此类型的对象从 Sun::Solaris::Exacct::Object 中继承。
第 44 页中的 “Sun::Solaris::Exacct::Object::Group 模块”	提供面向对象的方法，用于访问单个 exacct 记帐文件组。此类型的对象从 Sun::Solaris::Exacct::Object 中继承，并提供了对 ea_attach_to_group(3EXACCT) 函数的访问。组中包含的各项表示为 perl 数组。
第 44 页中的 “Sun::Solaris::Exacct::Object::_Array 模块”	专用数组类型，用作 Sun::Solaris::Exacct::Object::Group 中的数组类型。

Sun::Solaris::Project 模块

Sun::Solaris::Project 模块可以为与项目相关的系统调用和 libproject(3LIB) 库提供包装。

Sun::Solaris::Project 常量

Sun::Solaris::Project 模块使用与项目相关的头文件中的常量。

```
MAXPROJID
PROJNAME_MAX
PROJF_PATH
PROJECT_BUFSZ
SETPROJ_ERR_TASK
SETPROJ_ERR_POOL
```

Sun::Solaris::Project 函数、类方法和对象方法

libexacct(3LIB) API 的 perl 扩展为项目提供了以下函数。

```
setproject(3PROJECT)
setprojent(3PROJECT)
getdefaultproj(3PROJECT)
inproj(3PROJECT)
getprojent(3PROJECT)
fgetprojent(3PROJECT)
getprojbyname(3PROJECT)
```

getprojbyid(3PROJECT)
getprojbyname(3PROJECT)
endprojent(3PROJECT)

Sun::Solaris::Project 模块不包含类方法。

Sun::Solaris::Project 模块不包含对象方法。

Sun::Solaris::Project 导出

缺省情况下，此模块中不会导出任何内容。可以使用以下标记有选择性地导入此模块中定义的常量和函数。

标记	常量或函数
:SYSCALLS	getprojid()
:LIBCALLS	setproject()、activeprojects()、getprojent()、setprojent()、endprojent()、getprojbyname()、getprojbyid()、getdefaultproj()、fgetprojent()、inproj()、getprojidbyname()
:CONSTANTS	MAXPROJID_TASK、PROJNAME_MAX、PROJF_PATH、PROJECT_BUFSZ、SETPROJ_ERR、SETPROJ_ERR_POOL
:ALL	:SYSCALLS、:LIBCALLS、:CONSTANTS

Sun::Solaris::Task 模块

Sun::Solaris::Task 模块可以为 settaskid(2) 和 gettaskid(2) 系统调用提供包装。

Sun::Solaris::Task 常量

Sun::Solaris::Task 模块使用以下常量。

TASK_NORMAL
TASK_FINAL

Sun::Solaris::Task 函数、类方法和对象方法

libexacct(3LIB) API 的 perl 扩展为任务提供了以下函数。

settaskid(2)
gettaskid(2)

Sun::Solaris::Task 模块不包含类方法。

Sun::Solaris::Task 模块不包含对象方法。

Sun::Solaris::Task 导出

缺省情况下，此模块中不会导出任何内容。可以使用以下标记有选择性地导入此模块中定义的常量和函数。

标记	常量或函数
:SYSCALLS	settaskid()、gettaskid()
:CONSTANTS	TASK_NORMAL 和 TASK_FINAL
:ALL	:SYSCALLS 和 :CONSTANTS

Sun::Solaris::Exacct 模块

Sun::Solaris::Exacct 模块可以为 ea_error(3EXACCT) 函数和所有的 exacct 系统调用提供包装。

Sun::Solaris::Exacct 常量

Sun::Solaris::Exacct 模块可提供各种 exacct 头文件中的常量。P_PID、P_TASKID、P_PROJID 以及所有的 EW_*、EP_*、EXR_* 宏都是在模块生成过程中提取的。宏是从 /usr/include 下的 exacct 头文件中提取的，并且会被作为 Perl 常量提供。传递给 Sun::Solaris::Exacct 函数的常量可以是整数值（如 EW_FINAL）或同一变量的字符串表示形式（如 "EW_FINAL"）。

Sun::Solaris::Exacct 函数、类方法和对象方法

libexacct(3LIB) API 的 perl 扩展为 Sun::Solaris::Exacct 模块提供了以下函数。

```
getacct(2)
putacct(2)
wracct(2)
ea_error(3EXACCT)
ea_error_str
ea_register_catalog
ea_new_file
ea_new_item
ea_new_group
ea_dump_object
```

注 – ea_error_str() 作为一种便利方式提供，以便避免使用与以下类似的重复代码块：

```
if (ea_error() == EXR_SYSCALL_FAIL) {

        print("error: $!\n");

} else {

        print("error: ", ea_error(), "\n");

}
```

Sun::Solaris::Exacct 模块不包含类方法。

Sun::Solaris::Exacct 模块不包含对象方法。

Sun::Solaris::Exacct 导出

缺省情况下，此模块中不会导出任何内容。可以使用以下标记有选择性地导入此模块中定义的常量和函数。

标记	常量或函数
:SYSCALLS	getacct()、putacct()、wracct()
:LIBCALLS	ea_error()、ea_error_str()
:CONSTANTS	P_PID、P_TASKID、P_PROJID 、EW_*、EP_*、EXR_*
:SHORTAND	ea_register_catalog()、ea_new_catalog()、ea_new_file()、ea_new_item()、ea_new_group()、ea_dump_object()
:ALL	:SYSCALLS、:LIBCALLS、:CONSTANTS 和 :SHORTHAND
:EXACCT_CONSTANTS	:CONSTANTS 以及 Sun::Solaris::Catalog、Sun::Solaris::File、Sun::Solaris::Object 的 :CONSTANTS 标记
:EXACCT_ALL	:ALL 以及 Sun::Solaris::Catalog、Sun::Solaris::File、Sun::Solaris::Object 的 :ALL 标记

Sun::Solaris::Exacct::Catalog 模块

Sun::Solaris::Exacct::Catalog 模块可以为用作目录标记的 32 位整数提供包装。该目录标记表示为指定属于 Sun::Solaris::Exacct::Catalog 类的 Perl 对象。可以使用多种方法来处理目录标记中的字段。

Sun::Solaris::Exacct::Catalog 常量

所有的 EXT_*、EXC_* 和 EXD_* 宏都是在模块生成过程中从 /usr/include/sys/exact_catalog.h 文件中提取的，并且是作为常量提供的。传递给 Sun::Solaris::Exacct::Catalog 方法的常量可以是整数值（如 EXT_UINT8）或同一个变量的字符串表示形式（如 “EXT_UINT8”）。

Sun::Solaris::Exacct::Catalog 函数、类方法和对象方法

libexacct(3LIB) API 的 Perl 扩展为 Sun::Solaris::Exacct::Catalog 模块提供了以下类方法。Exacct(3PERL) 和 Exacct::Catalog(3PERL)

```
register
new
```

libexacct(3LIB) API 的 Perl 扩展为 Sun::Solaris::Exacct::Catalog 提供了以下对象类方法。

```
value
type
catalog
id
type_str
catalog_str
id_str
```

Sun::Solaris::Exacct::Catalog 导出

缺省情况下，此模块中不会导出任何内容。可以使用以下标记有选择性地导入此模块中定义的常量和函数。

标记	常量或函数
:CONSTANTS	EXT_*、EXC_* 和 EXD_*。
:ALL	:CONSTANTS

此外，还可以有选择地将使用 register() 函数定义的任何常量导出到调用方软件包中。

Sun::Solaris::Exacct::File 模块

Sun::Solaris::Exacct::File 模块可为用于处理记帐文件的 exacct 函数提供包装。该接口面向对象，并且允许创建和读取 exacct 文件。此模块所包装的 C 库调用包括以下各项：

ea_open(3EXACCT)
ea_close(3EXACCT)
ea_next_object(3EXACCT)
ea_previous_object(3EXACCT)
ea_write_object(3EXACCT)
ea_get_object(3EXACCT)
ea_get_creator(3EXACCT)
ea_get_hostname(3EXACCT)

文件的读写方法可用于对 Sun::Solaris::Exacct::Object 对象执行操作。这些方法可执行所有必需的内存管理、压缩、解压缩以及所需的结构转换操作。

Sun::Solaris::Exacct::File 常量

Sun::Solaris::Exacct::File 可提供 EO_HEAD、EO_TAIL、EO_NO_VALID_HDR、EO_POSN_MSK 和 EO_VALIDATE_MSK 常量。new() 方法所需的其他常量位于标准的 Perl Fcntl 模块中。表 4–2 介绍了针对 $oflags 和 $aflags 的各种值的 new() 操作。

Sun::Solaris::Exacct::File 函数、类方法和对象方法

Sun::Solaris::Exacct::File 模块不包含函数。

libexacct(3LIB) API 的 Perl 扩展为 Sun::Solaris::Exacct::File 提供了以下类方法。

new

下表介绍了 $oflags 和 $aflags 参数组合的 new() 操作。

表 4–2 $oflags 和 $aflags 参数

$oflags	$aflags	操作
O_RDONLY	不存在或为 EO_HEAD	打开以便在文件开头进行读取。
O_RDONLY	EO_TAIL	打开以便在文件结尾进行读取。
O_WRONLY	已忽略	文件必须存在，打开以便在文件结尾进行写入。

表 4–2 $oflags 和 $aflags 参数 （续）

$oflags	$aflags	操作
O_WRONLY | O_CREAT	已忽略	如果文件不存在，请创建文件。否则，截断并打开以便进行写入。
O_RDWR	已忽略	文件必须存在，打开以便在文件结尾进行读取或写入。
O_RDWR | O_CREAT	已忽略	如果文件不存在，请创建文件。否则，截断并打开以便进行读取或写入。

注 – $oflags 的唯一有效值是 O_RDONLY、O_WRONLY、O_RDWR 或 O_CREAT 的组合。$aflags 用于描述文件中要求对 O_RDONLY 进行的定位。允许使用 EO_HEAD 或 EO_TAIL。如果不存在，则假设使用 EO_HEAD。

libexacct(3LIB) API 的 perl 扩展为 Sun::Solaris::Exacct::File 提供了以下对象方法。

```
creator
hostname
next
previous
get
write
```

注 – 关闭 Sun::Solaris::Exacct::File。对于 Sun::Solaris::Exacct::File，不存在显式的 close() 方法。如果未定义或重新指定了文件句柄对象，则将关闭该文件。

Sun::Solaris::Exacct::File 导出

缺省情况下，此模块中不会导出任何内容。可以使用以下标记有选择性地导入此模块中定义的常量。

标记	常量或函数
:CONSTANTS	EO_HEAD、EO_TAIL、EO_NO_VALID_HDR、EO_POSN_MSK、EO_VALIDATE_MSK。
:ALL	:CONSTANTS 和 Fcntl(:DEFAULT)。

Sun::Solaris::Exacct::Object 模块

Sun::Solaris::Exacct::Object 模块可用作两种可能类型的 exacct 对象（项和组）的父对象。exacct Item 是指单个数据值、嵌入的 exacct 对象或原始数据块。进程占用的用户 CPU 时间（秒数）即是一个单个数据值的示例。exacct Group 是指 exacct 项（如特定进程或任务的所有资源使用情况的值）的有序集合。如果需要将组彼此嵌套，则可以将内部组作为封闭组内嵌入的 exacct 对象存储。

Sun::Solaris::Exacct::Object 模块包含对于 exacct 项和组都通用的方法。请注意，Sun::Solaris::Exacct::Object 及其派生的所有类的属性在最初通过 new() 创建后都是只读的。设置为只读的属性可以防止无意中修改属性，进而防止产生不一致的目录标记和数据值。对于只读属性，唯一的例外是用于在组对象内存储项的数组。可以使用常规的 perl 数组运算符来修改此数组。

Sun::Solaris::Exacct::Object 常量

Sun::Solaris::Exacct::Object 可提供 EO_ERROR、EO_NONE、EO_ITEM 和 EO_GROUP 常量。

Sun::Solaris::Exacct::Object 函数、类方法和对象方法

Sun::Solaris::Exacct::Object 模块不包含函数。

libexacct(3LIB) API 的 Perl 扩展为 Sun::Solaris::Exacct::Object 提供了以下类方法。

```
dump
```

libexacct(3LIB) API 的 Perl 扩展为 Sun::Solaris::Exacct::Object 提供了以下对象方法。

```
type
catalog
match_catalog
value
```

Sun::Solaris::Exacct::Object 导出

缺省情况下，此模块中不会导出任何内容。可以使用以下标记有选择性地导入此模块中定义的常量和函数。

标记	常量或函数
:CONSTANTS	EO_ERROR、EO_NONE、EO_ITEM 和 EO_GROUP

标记	常量或函数
:ALL	:CONSTANTS

Sun::Solaris::Exacct::Object::Item 模块

Sun::Solaris::Exacct::Object::Item 模块用于 exacct 数据项。exacct 数据项表示为指定属于 Sun::Solaris::Exacct::Object::Item 类（即 Sun::Solaris::Exacct::Object 类的子类）的不透明引用。可按如下方式将基础的 exacct 数据类型映射到 Perl 类型。

表 4–3 映射到 Perl 数据类型的 exacct 数据类型

exacct 类型	Perl 内部类型
EXT_UINT8	IV（整数）
EXT_UINT16	IV（整数）
EXT_UINT32	IV（整数）
EXT_UINT64	IV（整数）
EXT_DOUBLE	NV（双精度）
EXT_STRING	PV（字符串）
EXT_EXACCT_OBJECT	Sun::Solaris::Exacct::Object 子类
EXT_RAW	PV（字符串）

Sun::Solaris::Exacct::Object::Item 常量

Sun::Solaris::Exacct::Object::Item 不包含常量。

Sun::Solaris::Exacct::Object::Item 函数、类方法和对象方法

Sun::Solaris::Exacct::Object::Item 不包含函数。

Sun::Solaris::Exacct::Object::Item 可从 Sun::Solaris::Exacct::Object 基类继承所有的类方法以及 new() 类方法。

new

Sun::Solaris::Exacct::Object::Item 可从 Sun::Solaris::Exacct::Object 基类继承所有的对象方法。

Sun::Solaris::Exacct::Object::Item 导出

Sun::Solaris::Exacct::Object::Item 不包含任何导出内容。

Sun::Solaris::Exacct::Object::Group 模块

Sun::Solaris::Exacct::Object::Group 模块用于 exacct 组对象。exacct 组对象表示为指定属于 Sun::Solaris::Exacct::Object::Group 类（即 Sun::Solaris::Exacct::Object 类的子类）的不透明引用。组内的项存储在 Perl 数组中，并且可以通过继承的 value() 方法来访问对数组的引用。这意味着可以使用常规的 Perl 数组语法和运算符来处理组内的各项。该数组的所有数据元素都必须是从 Sun::Solaris::Exacct::Object 类派生的。另外，只能通过添加现有组作为数据项来将组对象彼此嵌套。

Sun::Solaris::Exacct::Object::Group 常量

Sun::Solaris::Exacct::Object::Group 不包含常量。

Sun::Solaris::Exacct::Object::Group 函数、类方法和对象方法

Sun::Solaris::Exacct::Object::Group 不包含函数。

Sun::Solaris::Exacct::Object::Group 可从 Sun::Solaris::Exacct::Object 基类继承所有的类方法以及 new() 类方法。

new

Sun::Solaris::Exacct::Object::Group 可从 Sun::Solaris::Exacct::Object 基类继承所有的对象方法以及 new() 类方法。

as_hash
as_hashlist

Sun::Solaris::Exacct::Object::Group 导出

Sun::Solaris::Exacct::Object::Group 不包含任何导出内容。

Sun::Solaris::Exacct::Object::_Array 模块

Sun::Solaris::Exacct::Object::_Array 类在内部用于强制对置于 exacct 组中的数据项进行类型检查。Sun::Solaris::Exacct::Object::_Array 不应该由用户直接创建。

Sun::Solaris::Exacct::Object::_Array 常量

Sun::Solaris::Exacct::Object::_Array 不包含常量。

Sun::Solaris::Exacct::Object::_Array 函数、类方法和对象方法

Sun::Solaris::Exacct::Object::_Array 不包含函数。

Sun::Solaris::Exacct::Object::_Array 包含内部使用的类方法。

Sun::Solaris::Exacct::Object::_Array 使用 perl TIEARRAY 方法。

Sun::Solaris::Exacct::Object::_Array 导出

Sun::Solaris::Exacct::Object::_Array不包含任何导出内容。

Perl 代码示例

本节提供用于访问 exacct 文件的 perl 代码示例。

示例 4–1 使用伪代码原型

在典型的使用中，Perl exacct 库用于读取现有的 exacct 文件。使用伪代码可显示各种 Perl exacct 类的关系。在伪代码中说明用于打开和扫描 exacct 文件以及处理所关注对象的进程。在以下伪代码中，为清楚起见使用了“便捷”函数。

```
-- Open the exacct file ($f is a Sun::Solaris::Exacct::File)

my $f = ea_new_file(...)



-- While not EOF ($o is a Sun::Solaris::Exacct::Object)

while (my $o = $f->get())



        -- Check to see if object is of interest

        if ($o->type() == &EO_ITEM)

             ...



        -- Retrieve the catalog ($c is a Sun::Solaris::Exacct::Catalog)

        $c = $o->catalog()
```

示例 4–1 使用伪代码原型 （续）

```
        -- Retrieve the value
        $v = $o->value();


        -- $v is a reference to a Sun::Solaris::Exacct::Group for a Group
        if (ref($v))
              ....


        -- $v is perl scalar for Items
        else
```

示例 4–2 递归转储 exacct 对象

```
sub dump_object
{
    my ($obj, $indent) = @_;
    my $istr = ’  ’ x $indent;


    #
    # Retrieve the catalog tag.  Because we are doing this in an array
    # context, the catalog tag will be returned as a (type, catalog, id)
    # triplet, where each member of the triplet will behave as an integer
    # or a string, depending on context.  If instead this next line provided
    # a scalar context, e.g.
    #    my $cat  = $obj->catalog()->value();
    # then $cat would be set to the integer value of the catalog tag.
```

示例 4–2 递归转储 exacct 对象 （续）

```
#
my @cat = $obj->catalog()->value();


#
# If the object is a plain item
#
if ($obj->type() == &EO_ITEM) {
      #
      # Note:  The '%s' formats provide s string context, so the
      # components of the catalog tag will be displayed as the
      # symbolic values.  If we changed the '%s' formats to '%d',
      # the numeric value of the components would be displayed.
      #
      printf("%sITEM\n%s  Catalog = %s|%s|%s\n",
          $istr, $istr, @cat);
      $indent++;


      #
      # Retrieve the value of the item.  If the item contains in
      # turn a nested exacct object (i.e. a item or group), then
      # the value method will return a reference to the appropriate
      # sort of perl object (Exacct::Object::Item or
      # Exacct::Object::Group). We could of course figure out that
```

示例 4–2 递归转储 exacct 对象 （续）

```
    # the item contained a nested item or group by examining
    # the catalog tag in @cat and looking for a type of
    # EXT_EXACCT_OBJECT or EXT_GROUP.
    my $val = $obj->value();
    if (ref($val)) {
       # If it is a nested object, recurse to dump it.
       dump_object($val, $indent);
    } else {
       # Otherwise it is just a 'plain' value, so display it.
       printf("%s  Value = %s\n", $istr, $val);
    }


  #
  # Otherwise we know we are dealing with a group.  Groups represent
  # contents as a perl list or array (depending on context), so we
  # can process the contents of the group with a 'foreach' loop, which
  # provides a list context.  In a list context the value method
  # returns the content of the group as a perl list, which is the
  # quickest mechanism, but doesn't allow the group to be modified.
  # If we wanted to modify the contents of the group we could do so
  # like this:
  #    my $grp = $obj->value();   # Returns an array reference
  #    $grp->[0] = $newitem;
```

示例 4–2 递归转储 exacct 对象 （续）

```
        # but accessing the group elements this way is much slower.
        #
        } else {
                printf("%sGROUP\n%s  Catalog = %s|%s|%s\n",
                    $istr, $istr, @cat);
                $indent++;
                # 'foreach' provides a list context.
                foreach my $val ($obj->value()) {
                        dump_object($val, $indent);
                }
                printf("%sENDGROUP\n", $istr);
        }
}
```

示例 4–3 创建新的组记录并写入文件

```
# Prototype list of catalog tags and values.
my @items = (
   [ &EXT_STRING | &EXC_DEFAULT | &EXD_CREATOR      => "me"         ],
   [ &EXT_UINT32 | &EXC_DEFAULT | &EXD_PROC_PID     => $$           ],
   [ &EXT_UINT32 | &EXC_DEFAULT | &EXD_PROC_UID     => $<           ],
   [ &EXT_UINT32 | &EXC_DEFAULT | &EXD_PROC_GID     => $(           ],
   [ &EXT_STRING | &EXC_DEFAULT | &EXD_PROC_COMMAND => "/bin/stuff" ],
);
```

示例 4–3 创建新的组记录并写入文件 （续）

```
# Create a new group catalog object.
my $cat = new_catalog(&EXT_GROUP | &EXC_DEFAULT | &EXD_NONE);

# Create a new Group object and retrieve its data array.
my $group = new_group($cat);
my $ary = $group->value();

# Push the new Items onto the Group array.
foreach my $v (@items) {
        push(@$ary, new_item(new_catalog($v->[0]), $v->[1]));
}

# Nest the group within itself (performs a deep copy).
push(@$ary, $group);

# Dump out the group.
dump_object($group);
```

示例 4–4 转储 exacct 文件

```
#!/usr/perl5/5.6.1/bin/perl

use strict;
use warnings;
use blib;
```

示例 4–4 转储 exacct 文件 （续）

```
use Sun::Solaris::Exacct qw(:EXACCT_ALL);


die("Usage is dumpexacct


# Open the exact file and display the header information.
my $ef = ea_new_file($ARGV[0], &O_RDONLY) || die(error_str());
printf("Creator:  %s\n", $ef->creator());
printf("Hostname: %s\n\n", $ef->hostname());


# Dump the file contents
while (my $obj = $ef->get()) {
        ea_dump_object($obj);
}


# Report any errors
if (ea_error() != EXR_OK && ea_error() != EXR_EOF)  {
       printf("\nERROR: %s\n", ea_error_str());
       exit(1);
}
exit(0);
```

dump 方法的输出

本示例显示了 Sun::Solaris::Exacct::Object->dump() 方法的格式化输出。

```
GROUP
  Catalog = EXT_GROUP|EXC_DEFAULT|EXD_GROUP_PROC_PARTIAL
  ITEM
    Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_PID
    Value = 3
  ITEM
    Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_UID
    Value = 0
  ITEM
    Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_GID
    Value = 0
  ITEM
    Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_PROJID
    Value = 0
  ITEM
    Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_TASKID
    Value = 0
  ITEM
    Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_CPU_USER_SEC
    Value = 0
  ITEM
    Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_CPU_USER_NSEC
```

```
  Value = 0

ITEM

  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_CPU_SYS_SEC

  Value = 890

ITEM

  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_CPU_SYS_NSEC

  Value = 760000000

ITEM

  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_START_SEC

  Value = 1011869897

ITEM

  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_START_NSEC

  Value = 380771911

ITEM

  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_FINISH_SEC

  Value = 0

ITEM

  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_FINISH_NSEC

  Value = 0

ITEM

  Catalog = EXT_STRING|EXC_DEFAULT|EXD_PROC_COMMAND

  Value = fsflush

ITEM

  Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_TTY_MAJOR
```

```
  Value = 4294967295
ITEM
  Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_TTY_MINOR
  Value = 4294967295
ITEM
  Catalog = EXT_STRING|EXC_DEFAULT|EXD_PROC_HOSTNAME
  Value = mower
ITEM
  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_FAULTS_MAJOR
  Value = 0
ITEM
  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_FAULTS_MINOR
  Value = 0
ITEM
  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_MESSAGES_SND
  Value = 0
ITEM
  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_MESSAGES_RCV
  Value = 0
ITEM
  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_BLOCKS_IN
  Value = 19
ITEM
  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_BLOCKS_OUT
```

```
  Value = 40833
ITEM
  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_CHARS_RDWR
  Value = 0
ITEM
  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_CONTEXT_VOL
  Value = 129747
ITEM
  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_CONTEXT_INV
  Value = 79
ITEM
  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_SIGNALS
  Value = 0
  ITEM
  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_SYSCALLS
  Value = 0
ITEM
  Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_ACCT_FLAGS
  Value = 1
ITEM
  Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_ANCPID
  Value = 0
ITEM
  Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_WAIT_STATUS
```

```
        Value = 0

ENDGROUP
```

◆ ◆ ◆ 第 5 章

5 资源控制

本章介绍资源控制及其属性。

资源控制概述

可使用扩展的记帐功能确定系统上的工作负荷所占用的资源。确定占用的资源后，请使用资源控制功能为资源使用情况设置限制。 对资源设置的限制可以防止工作负荷过度占用资源。

有关资源控制以及用于管理资源控制的示例命令的概述，请参见《系统管理指南：Solaris Containers－资源管理和 Solaris Zones》中的第 6 章 “资源控制（概述）”和《系统管理指南：Solaris Containers－资源管理和 Solaris Zones》中的第 7 章 “管理资源控制（任务）”。

资源控制功能可带来以下益处。

- **动态设置**

  可以在系统运行的同时调整资源控制。

- **包含级别粒度**

  资源控制安排在项目、任务或进程的包含级别中。包含级别简化了配置，并使收集到的值更接近于特定的项目、任务或进程。

- **阈值保留**

  如果尝试将最大值设置为小于实际的资源占用额，则不会对最大值进行任何更改。

资源控制标志和操作

本节介绍了与资源控制关联的标志、操作和信号。

rlimit，资源限制

rlimit 是基于进程的。rlimit 可确立对某个进程占用各种系统资源的限制。由该进程创建的每个进程均从其原始进程继承此资源限制。资源限制由一对值来定义。这些值指定了当前（软）限制和最大（硬）限制。

进程可能会不可逆转地将其硬限制降为大于或等于软限制的任何值。只有具有超级用户 ID 的进程可以提高硬限制。请参见 setrlimit() 和 getrlimit()。

rlimit 结构包含用于定义软限制和硬限制的两个成员。

```
rlim_t     rlim_cur;       /* current (soft) limit */

rlim_t     rlim_max        /* hard limit */
```

rctl，资源控制

rctl 通过控制由项目数据库中定义的进程、任务和项目占用的资源来扩大基于进程的 rlimit 限制。

注 – 对于设置资源限制，rctl 机制优先于使用 rlimit。只有需要跨 UNIX 平台的可移植性时才应使用 rlimit 功能。

根据应用程序处理资源控制的方式，应用程序可分为以下几大类。根据执行的操作，可以进一步对资源控制进行分类。大多数资源控制会报告错误并终止操作。其他资源控制允许应用程序继续运行并适应资源占用减少的情况。可以为每种资源控制指定以递增的值表示的渐进操作链。

资源控制的属性列表由权限级别、阈值和超过阈值时执行的操作组成。

资源控制值和权限级别

有关资源控制的每个阈值都必须与以下某种权限级别关联：

RCPRIV_BASIC
: 权限级别可由调用进程的属主进行修改。RCPRIV_BASIC 与资源的软限制关联。

RCPRIV_PRIVILEGED
: 权限级别只能由具有权限的（超级用户）调用方进行修改。RCPRIV_PRIVILEGED 与资源的硬限制关联。

RCPRIV_SYSTEM
　权限级别在操作系统实例的持续时间内固定不变。

图 5–2 显示用于设置信号权限级别（由 /etc/project 文件的 process.max-cpu-time 资源控制定义）的时间线。

局部操作和局部标志

局部操作和局部标志适用于由此资源控制块表示的当前资源控制值。局部操作和局部标志特定于值。对于为资源控制设置的每个阈值，以下局部操作和局部标志是可用的：

RCTL_LOCAL_NOACTION
　超过此资源控制值时，不执行任何局部操作。

RCTL_LOCAL_SIGNAL
　指定的信号（由 rctlblk_set_local_action() 设置）将被发送到在值序列中设置此资源控制值的进程。

RCTL_LOCAL_DENY
　遇到此资源控制值时，将拒绝对资源的请求。如果为此控制设置了 RCTL_GLOBAL_DENY_ALWAYS，则会对所有值设置该项。如果为此控制设置了 RCTL_GLOBAL_DENY_NEVER，则会对所有值清除该项。

RCTL_LOCAL_MAXIMAL
　此资源控制值表示对此控制的最大资源量的请求。如果为此资源控制设置了 RCTL_GLOBAL_INFINITE，则 RCTL_LOCAL_MAXIMAL 将指示资源控制值没有限制，永远不会超出。

全局操作和全局标志

全局标志适用于由此资源控制块表示的所有当前资源控制值。全局操作和全局标志是通过 rctladm(1M) 设置的。不能使用 setrctl() 设置全局操作和全局标志。全局标志适用于所有的资源控制。对于为资源控制设置的每个阈值，以下全局操作和全局标志是可用的：

RCTL_GLOBAL_NOACTION
　超过有关此控制的资源控制值时，不执行任何全局操作。

RCTL_GLOBAL_SYSLOG
　超过序列中与此控制关联的任何资源控制值时，可通过 syslog() 功能记录标准消息。

RCTL_GLOBAL_NOBASIC
　此控制不允许具有 RCPRIV_BASIC 权限的任何值。

RCTL_GLOBAL_LOWERABLE
没有权限的调用方可以降低具有权限的资源控制值中有关此控制的值。

RCTL_GLOBAL_DENY_ALWAYS
超过有关此控制的控制值时执行的操作始终包括拒绝资源。

RCTL_GLOBAL_DENY_NEVER
超过有关此控制的控制值时执行的操作始终排除拒绝资源。尽管也可能会执行其他操作，但会始终授权使用该资源。

RCTL_GLOBAL_FILE_SIZE
局部操作的有效信号包括 SIGXFSZ 信号。

RCTL_GLOBAL_CPU_TIME
局部操作的有效信号包括 SIGXCPU 信号。

RCTL_GLOBAL_SIGNAL_NEVER
此控制不允许局部操作。始终授权使用该资源。

RCTL_GLOBAL_INFINITE
此资源控制支持没有限制的值这一概念。通常，没有限制的值只适用于逐渐累积型的资源（如 CPU 时间）。

RCTL_GLOBAL_UNOBSERVABLE
通常，与任务或项目相关的资源控制不支持观测的控制值。为任务或进程设置的具有 RCPRIV_BASIC 权限的控制值只有在被该进程超过时才生成操作。

与项目、进程和任务关联的资源控制集

下图显示了与任务、进程和项目关联的资源控制集。

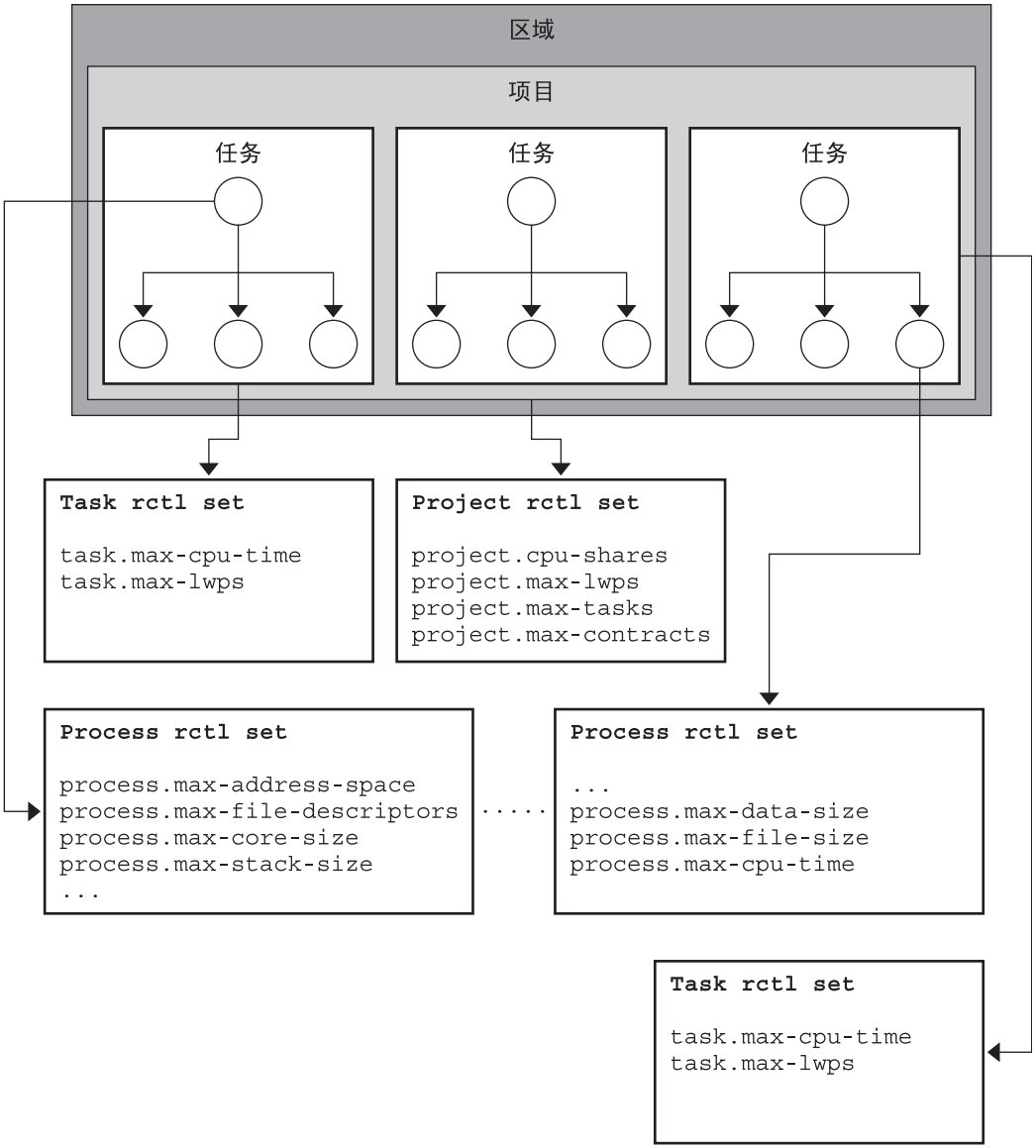


图 5–1 任务、项目和进程的资源控制集

一种资源可以存在多个资源控制，每个资源控制都处在进程模型的包含级别上。资源控制可以在进程以及集合任务或集合项目的同一种资源中处于活动状态。在这种情况

下，对进程的操作优先。例如，如果同时遇到 process.max-cpu-time 和 task.max-cpu-time 这两个控制，则首先对第一个控制执行操作。

与项目关联的资源控制

与项目关联的资源控制包括以下各项：

project.cpu-shares
: 授予此项目的 CPU 份额，用于公平共享调度程序 FSS(7)。

project.max-msg-ids
: 项目所允许的最大 System V 消息队列数。

project.max-sem-ids
: 项目所允许的最大 System V 信号数。

project.max-port-ids
: 允许的最大事件端口数。

与任务关联的资源控制

与任务关联的资源控制包括以下各项：

task.max-cpu-time
: 此任务进程可用的最长 CPU 时间（秒）。

task.max-lwps
: 此任务的进程可同时使用的最大 LWP 数。

与进程关联的资源控制

与进程关联的资源控制包括以下各项：

process.max-address-space
: 此进程可用的最大地址空间量（字节），即段大小的总和。

process.max-core-size
: 此进程创建的最大核心转储文件大小（字节）。

process.max-cpu-time
: 此进程可用的最长 CPU 时间（秒）。

process.max-file-descriptor
: 此进程可用的最大文件描述符索引。

process.max-file-size
: 此进程可写入的最大文件偏移（字节）。

process.max-msg-messages
: 消息队列中的最大消息数。该值是在 msgget() 时间从资源控制中复制的。

process.max-msg-qbytes
消息队列中的最大消息数（字节）。该值是在 msgget() 时间从资源控制中复制的。如果设置了新的 project.max-msg-qbytes 值，则仅对后续创建的值进行初始化。新的 project.max-msg-qbytes 值不影响现有值。

process.max-sem-nsems
信号集允许的最大信号数。

process.max-sem-ops
semop() 调用允许的最大信号操作数。该值是在 msgget() 时间从资源控制中复制的。新的 project.max-sem-ops 值仅影响对后续创建值的初始化，对现有值没有任何影响。

process.max-port-events
每个事件端口允许的最大事件数。

用于资源控制的信号

对于为资源控制设置的每个阈值，以下一组受限制的信号是可用的：

SIGBART
终止进程。

SIGXRES
超过资源控制限制时由资源控制功能生成的信号。

SIGHUP
当载波在断开的线路上停止时，将挂起信号发送到用于控制终端的进程组，此挂起信号即 SIGHUP。

SIGSTOP
作业控制信号。停止进程。停止信号不是来自终端。

SIGTERM
终止进程。由软件发送的终止信号。

SIGKILL
终止进程。中止程序。

SIGXFSX
终止进程。超过了文件大小限制。仅可用于具有 RCTL_GLOBAL_FILE_SIZE 属性的资源控制。

SIGXCPU
终止进程。超过了 CPU 时间限制。仅可用于具有 RCTL_GLOBAL_CPUTIME 属性的资源控制。

由于特定控制的全局属性，可能允许其他信号。

注 – 对包含非法信号的 `setrctl()` 的调用将失败。

```
                    /etc/project
cgi-bin:103:cgi-bin scripts:root,apache::\
  process.max-cpu-time=(privileged,1000,signal=SIGXCPU),\
                       (privileged,2000,signal=SIGTERM),\
                       (privileged,3000,signal=SIGKILL),\
```

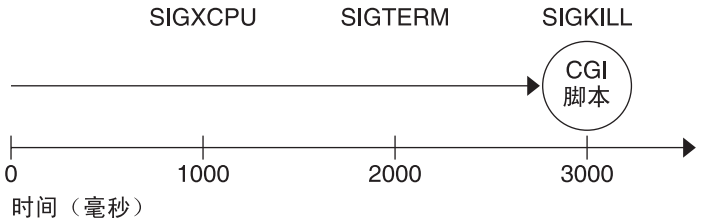


图 5–2 为信号设置权限级别

资源控制 API 函数

资源控制 API 包含的函数可以：

- 第 64 页中的 “对资源控制的操作-值对执行操作”
- 第 64 页中的 “对局部可修改值执行操作”
- 第 65 页中的 “检索局部只读值”
- 第 65 页中的 “检索全局只读操作”

对资源控制的操作-值对执行操作

以下列表包含用于设置或获取资源控制块的函数。

`setrctl(2)`
`getrctl(2)`

对局部可修改值执行操作

以下列表包含与局部可修改资源控制块关联的函数。

`rctlblk_set_privilege(3C)`
`rctlblk_get_privilege(3C)`

rctlblk_set_value(3C)
rctlblk_get_value(3C)
rctlblk_set_local_action(3C)
rctlblk_get_local_action(3C)
rctlblk_set_local_flags(3C)
rctlblk_get_local_flags(3C)

检索局部只读值

以下列表包含与局部只读资源控制块关联的函数。

rctlblk_get_recipient_pid(3C)
rctlblk_get_firing_time(3C)
rctlblk_get_enforced_value(3C)

检索全局只读操作

以下列表包含与全局只读资源控制块关联的函数。

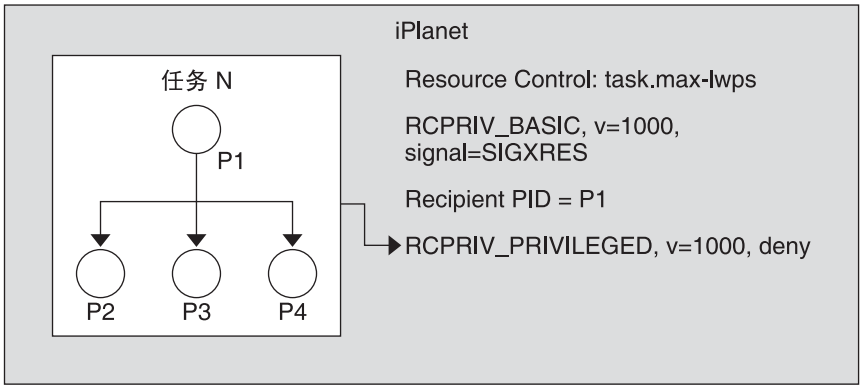
rctlblk_get_global_action(3C)
rctlblk_get_global_flags(3C)

资源控制代码示例

资源控制的主观察进程

以下示例是主观察者进程。图 5–3 显示了主观察进程的资源控制。

注 – 换行符在 `/etc/project` 文件中是无效的。此处显示的换行符仅允许示例显示在列显页或显示页上。`/etc/project` 文件中的每一项都必须占用单独的一行。



/etc/project

```
...
iPlanet:200:iPlanet Application Server:root::\
            task.max-lwps=(PRIVILEGED,1000,deny)
```

图 5–3 主观察进程

该示例的要点包括以下内容：

- 由于任务的限制具有权限，因此应用程序无法更改限制，也无法指定操作（如信号）。主进程通过为任务建立与基本资源控制相同的资源控制解决了此问题。主进程对资源使用相同的值或稍小的值，但执行不同的操作（信号 = XRES）。主进程将创建一个线程以等待此信号。
- `rctlblk` 是不透明的。需要动态分配结构。
- 请注意，sigwait(2) 要求在创建线程之前阻塞所有的信号。
- 线程将调用 `sigwait`(2) 来阻塞信号。如果 `sigwait()` 返回 SIGXRES 信号，则线程将通知主进程的子进程适当减少正在使用的 LWP 的数目。此外，还应该使用每个子进程中的线程以类似方式为每个子进程建模，以等待此信号并适当调整其进程的 LWP 使用情况。

```
rctlblk_t *mlwprcb;

sigset_t smask;



/* Omit return value checking/error processing to keep code sample short */

/* First, install a RCPRIV_BASIC, v=1000, signal=SIGXRES rctl */

mlwprcb = calloc(1, rctlblk_size());     /* rctl blocks are opaque: */
```

```
        rctlblk_set_value(mlwprcb, 1000);
        rctlblk_set_privilege(mlwprcb, RCPRIV_BASIC);
        rctlblk_set_local_action(mlwprcb, RCTL_LOCAL_SIGNAL, SIGXRES);
        if (setrctl("task.max-lwps", NULL, mlwprcb, RCTL_INSERT) == -1) {
            perror("setrctl");
            exit (1);
        }

/* Now, create the thread which waits for the signal */
         sigemptyset(&smask);
         sigaddset(&smask, SIGXRES);
         thr_sigsetmask(SIG_BLOCK, &smask, NULL);
thr_create(NULL, 0, sigthread, (void *)SIGXRES, THR_DETACHED, NULL));

/* Omit return value checking/error processing to keep code sample short */

void *sigthread(void *a)
{
         int sig = (int)a;
         int rsig;
         sigset_t sset;

         sigemptyset(&sset);
         sigaddset(&sset, sig);
```

```
        while (1) {

                rsig = sigwait(&sset);

          if (rsig == SIGXRES) {

               notify_all_children();

               /* e.g. sigsend(P_PID, child_pid, SIGXRES); */

              }

        }

}
```

列出特定资源控制的所有值-操作对

以下示例列出了特定资源控制 task.max-lwps 的所有值-操作对。该示例的要点是 getrctl(2) 使用两个资源控制块，并返回 RCTL_NEXT 标志的资源控制块。要在所有的资源控制块中迭代，请使用 rcb_tmp rctl 块反复交换此处显示的资源控制块值。

```
rctlblk_t *rcb1, *rcb2, *rcb_tmp;

    ...

/* Omit return value checking/error processing to keep code sample short */

rcb1 = calloc(1, rctlblk_size()); /* rctl blocks are opaque: */

                                  /* "rctlblk_t rcb" does not work */

rcb2 = calloc(1, rctlblk_size());

getrctl("task.max-lwps", NULL, rcb1, RCTL_FIRST);

while (1) {

     print_rctl(rcb1);

     rcb_tmp = rcb2;

     rcb2 = rcb1;

     rcb1 = rcb_tmp;        /* swap rcb1 with rcb2 */
```

```
     if (getrctl("task.max-lwps", rcb2,  rcb1, RCTL_NEXT) == -1) {
          if (errno == ENOENT) {
               break;
     } else {
          perror("getrctl");
          exit (1);
     }
     }
}
```

设置 project.cpu-shares 并添加新值

该示例的要点包括以下内容：

- 该示例类似于第 84 页中的 “设置 pool.comment 属性并添加新属性”中显示的示例。
- 请使用 bcopy()，而不要使用第 68 页中的 “列出特定资源控制的所有值-操作对”中的缓冲区交换。
- 要更改资源控制值，在调用 setrctl() 时请使用 RCTL_REPLACE 标志。新的资源控制块除了使用新的控制值以外，与原有的资源控制块相同。

```
rctlblk_set_value(blk1, nshares);

if (setrctl("project.cpu-shares", blk2, blk1, RCTL_REPLACE) != 0)
```

该示例将获取项目的 CPU 份额分配 project.cpu-shares，并将其值更改为 *nshares*。

```
/* Omit return value checking/error processing to keep code sample short */

blk1 = malloc(rctlblk_size());

getrctl("project.cpu-shares", NULL, blk1, RCTL_FIRST);

my_shares = rctlblk_get_value(blk1);

printout_my_shares(my_shares);
```

```
/* if privileged, do the following to */

/* change project.cpu-shares to "nshares" */

blk1 = malloc(rctlblk_size());

blk2 = malloc(rctlblk_size());

if (getrctl("project.cpu-shares", NULL, blk1, RCTL_FIRST) != 0) {

     perror("getrctl failed");

     exit(1);

}

bcopy(blk1, blk2, rctlblk_size());

rctlblk_set_value(blk1, nshares);

if (setrctl("project.cpu-shares", blk2, blk1, RCTL_REPLACE) != 0) {

     perror("setrctl failed");

     exit(1);

}
```

为资源控制块设置 LWP 限制

在以下示例中，应用程序设置了具有权限的限制 3000 LWP（可能没有超过）。此外，应用程序还设置了基本限制 2000 LWP。超过此限制时，系统会将 SIGXRES 发送到应用程序。收到 SIGXRES 时，应用程序可能会将通知发送给其子进程，而该进程可能会减少进程使用或需要的 LWP 数目。

```
/* Omit return value and error checking */



#include <rctl.h>



rctlblk_t *rcb1, *rcb2;
```

```
/*
         * Resource control blocks are opaque
         * and must be explicitly allocated.
         */
rcb1 = calloc(rctlblk_size());

rcb2 = calloc(rctlblk_size());



/* Install an RCPRIV_PRIVILEGED, v=3000: do not allow more than 3000 LWPs */
rctlblk_set_value(rcb1, 3000);
rctlblk_set_privilege(rcb1, RCPRIV_PRIVILEGED);
rctlblk_set_local_action(rcb1, RCTL_LOCAL_DENY);
setrctl("task.max-lwps", NULL, rcb1, RCTL_INSERT);



/* Install an RCPRIV_BASIC, v=2000 to send SIGXRES when LWPs exceeds 2000 */
rctlblk_set_value(rcb2, 2000);
rctlblk_set_privilege(rcb2, RCPRIV_BASIC);
rctlblk_set_local_action(rcb2, RCTL_LOCAL_SIGNAL, SIGXRES);
setrctl("task.max-lwps", NULL, rcb2, RCTL_INSERT);
```

与资源控制关联的编程问题

编写应用程序时，请考虑以下问题：

- 资源控制块是不透明的。需要动态分配控制块。
- 如果在任务或项目中建立了基本资源控制，则建立此资源控制的进程将成为观察者。对此资源控制块的操作将应用于该观察者。但是，不能使用此方式观察某些资源。
- 如果在任务或项目中设置具有权限的资源控制，则不存在观察者进程。但是，违反该限制的任何进程都将成为资源控制操作的适用对象。
- 只允许对每种类型执行一项操作：全局操作或局部操作。
- 只允许对每个进程、每个资源控制使用一个基本 `rctl`。

◆ ◆ ◆ 第 6 章

6 动态资源池

本章介绍资源池及其属性。

资源池概述

资源池提供了用于管理处理器集和线程调度类的框架。资源池用于对计算机资源进行分区。通过资源池可以分散工作负荷，使各工作负荷对特定资源的占用不会发生冲突。在具有混合工作负荷的系统上，这种资源预留有助于获得可预测的性能。

有关资源池以及用于管理资源池的示例命令的概述，请参见《系统管理指南：Solaris Containers－资源管理和 Solaris Zones》中的第 12 章 “动态资源池（概述）”和《系统管理指南：Solaris Containers－资源管理和 Solaris Zones》中的第 13 章 “管理动态资源池（任务）”。

处理器集将系统中的 CPU 分组到一个有界限的实体中，在该实体中进程可以采用独占方式运行。其中的进程不能扩展到该处理器集外，而其他进程也不能扩展到该处理器集内。处理器集可以将具有类似特征的任务组合在一起，并设置针对 CPU 使用的硬限制（上限）。

资源池框架用于定义具有最大和最小 CPU 计数要求的软处理器集。此外，该框架还为该处理器集提供了一个硬定义调度类。

资源池定义以下内容

- 处理器集组
- 调度类

调度类

调度类为基于算术逻辑的线程提供不同的 CPU 访问特性。调度类包括：

- 实时调度类
- 交互式调度类
- 固定优先级调度类
- 分时调度类
- 公平共享调度类

有关公平共享调度程序以及用于管理公平共享调度程序的示例命令的概述，请参见《系统管理指南：Solaris Containers－资源管理和 Solaris Zones》中的第 8 章 “公平共享调度程序（概述）”和《系统管理指南：Solaris Containers－资源管理和 Solaris Zones》中的第 9 章 “管理公平共享调度程序（任务）”。

请不要在 CPU 集中混合调度类。如果在 CPU 集中混合调度类，则系统性能可能会不稳定或不可预测。请使用处理器集按照应用程序特征分别部署应用程序。请指定使应用程序能够达到最佳性能的调度类。有关各个调度类的特征的更多信息，请参见 `priocntl`(1)。

有关资源池的概述以及对何时使用资源池的讨论，请参见第 6 章。

动态资源池约束和目标

`libpool` 库定义了可用于各种实体（使用池功能管理）的属性。每个属性都属于以下类别：

配置约束
约束定义了属性的限制。典型的约束是在 `libpool` 配置中指定的最大和最小分配量。

目标
目标可以更改当前配置的资源分配，以生成遵循已建立约束的新的候选配置。（请参见 `poold`(1M)。）目标包含以下类别：

与工作负荷有关 与工作负荷有关的目标将依据工作负荷强加的条件有所变化。与工作负荷有关的目标示例为 `utilization` 目标。

与工作负荷无关 与工作负荷无关的目标不会依据工作负荷强加的条件发生变化。与工作负荷无关的目标示例为 `cpu locality` 目标。

目标可以采用可选的前缀来指示目标的重要性。为确定目标的重要性，将在目标中增加此前缀（是 0 到 INT64_MAX 之间的整数）。

系统属性

`system.bind-default`（**可写布尔值**）
如果在 `/etc/project` 中找不到指定的池，请绑定到 `pool.default` 属性设置为 TRUE 的池。

`system.comment`（**可写字符串**）
系统的用户说明。缺省的池命令不使用 `system.comment`，但通过 `poolcfg` 实用程序启动配置时除外。此时，系统会将提示性消息置于该配置的 `system.comment` 属性中。

`system.name`（**可写字符串**）
配置的用户名称。

`system.version`（只读整数）
处理此配置所需的 `libpool` 版本。

池属性

所有的池属性都是可写的。

`pool.active`（**可写布尔值**）
如果为 TRUE，则表示该池处于活动状态。

`pool.comment`（**可写字符串**）
池的用户说明。

`pool.default`（**可写布尔值**）
如果为 TRUE，则表示该池为缺省池。请参见 `system.bind-default` 属性。

`pool.importance`（**可写整数**）
该池的相对重要性。用于可能的资源争用解决方案。

`pool.name`（**可写字符串**）
池的用户名称。`setproject`(3PROJECT) 使用 `pool.name` 作为 `project`(4) 数据库中的 `project.pool` 项目属性的值。

`pool.scheduler`（**可写字符串**）
与该池的使用者绑定的调度程序类。此属性是可选的，如果未指定，则调度程序对该池使用者的绑定将不受影响。有关各个调度类的特征的更多信息，请参见 `priocntl`(1)。调度程序类包括：

- RT（代表实时调度程序）
- TS（代表分时调度程序）
- IA（代表交互式调度程序）
- FSS（代表公平共享调度程序）
- FX（代表固定优先级调度程序）

处理器集属性

pset.comment（**可写字符串**）
资源的用户说明。

pset.default（只读布尔值）
标识缺省的处理器集。

pset.escapable（**可写布尔值**）
表示是否为此 pset 设置 PSET_NOESCAPE。请参见 pset_setattr(2) 手册页。

pset.load（只读无符号整数）
此处理器集的负载。最低值为 0。该值将随着处理器集上的负载（由系统运行队列中的作业数度量）以线性方式增加。

pset.max（**可写无符号整数**）
此处理器集中允许的最大 CPU 数。

pset.min（**可写无符号整数**）
此处理器集中允许的最小 CPU 数。

pset.name（**可写字符串**）
资源的用户名称。

pset.size（只读无符号整数）
此处理器集的当前 CPU 数。

pset.sys_id（只读整数）
系统指定的处理器集 ID。

pset.type（只读字符串）
命名资源类型。所有处理器集的值都为 pset。

pset.units（只读字符串）
标识与大小相关的属性的意义。所有处理器集的值都为 population。

cpu.comment（**可写字符串**）
CPU 的用户说明。

使用 libpool 处理池配置

libpool(3LIB) 池配置库定义了用于读取和写入池配置文件的接口。该库还定义了用于提交现有配置以成为运行的操作系统配置的接口。<pool.h> 头文件提供了所有库服务的类型和函数声明。

在资源池功能中，引入了“池”这个一般性的概念，用来指可绑定到进程的资源的集合。可以采用持久方式配置、分组并标记处理器集和其他实体。可以将工作负荷组件与系统总资源的一部分相关联。libpool(3LIB) 库提供了用于访问资源池功能的 C 语言 API。pooladm(1M)、poolbind(1M) 和 poolcfg(1M) 可通过从 shell 中调用命令来使用资源池功能。

处理 pset

以下列表包含与创建或销毁 pset 以及处理 pset 关联的函数。

processor_bind(2)	将 LWP（lightweight process，轻量进程）或一组 LWP 绑定到指定的处理器。
pset_assign(2)	为处理器集分配处理器。
pset_bind(2)	将一个或多个 LWP（lightweight processes，轻量进程）绑定到处理器集。
pset_create(2)	创建不包含处理器的空处理器集。
pset_destroy(2)	销毁处理器集并释放关联的成员处理器和进程。
pset_setattr(2), pset_getattr(2)	设置或获取处理器集属性。

资源池 API 函数

本节列出了所有的资源池函数。每个函数都带有指向相应手册页的链接和有关该函数用途的简短说明。根据函数执行操作还是查询，函数分为两组：

- 第 77 页中的 “用于对资源池和关联元素执行操作的函数”
- 第 79 页中的 “用于查询资源池和关联元素的函数”

用于交换集的 libpool 的导入接口与本文档中定义的接口相同。

用于对资源池和关联元素执行操作的函数

本节中列出的接口用于执行与池和关联元素相关的操作。

pool_associate(3POOL)	将资源与指定的池关联起来。
pool_component_to_elem(3POOL)	将指定的组件转换为池元素类型。
pool_conf_alloc(3POOL)	创建池配置。
pool_conf_close(3POOL)	关闭指定的池配置并释放关联资源。
pool_conf_commit(3POOL)	提交对指定池配置所做的更改以进行永久存储。
pool_conf_export(3POOL)	将给定的配置保存到指定位置。
pool_conf_free(3POOL)	释放池配置。
pool_conf_open(3POOL)	在指定的位置创建池配置。
pool_conf_remove(3POOL)	删除对配置的永久存储。

pool_conf_rollback(3POOL)	将配置状态恢复到池配置的永久存储中保留的状态。
pool_conf_to_elem(3POOL)	将指定的池配置转换为池元素类型。
pool_conf_update(3POOL)	更新内核状态的库快照。
pool_create(3POOL)	使用缺省属性和每种类型的缺省资源创建新的池。
pool_destroy(3POOL)	破坏指定的池。关联的资源不会被修改。
pool_dissociate(3POOL)	删除给定资源与池之间的关联。
pool_put_property(3POOL)	将有关元素的命名属性设置为指定的值。
pool_resource_create(3POOL)	使用所提供配置的指定名称和类型创建新的资源。
pool_resource_destroy(3POOL)	从配置文件中删除指定的资源。
pool_resource_to_elem(3POOL)	将指定的池资源转换为池元素类型。
pool_resource_transfer(3POOL)	将基本单位从源资源传输到目标资源。
pool_resource_xtransfer(3POOL)	将指定组件从源资源传输到目标资源。
pool_rm_property(3POOL)	从元素中删除命名的属性。
pool_set_binding(3POOL)	将指定的进程绑定到与正在运行的系统中的池关联的资源。
pool_set_status(3POOL)	修改池功能的当前状态。
pool_to_elem(3POOL)	将指定的池转换为池元素类型。
pool_value_alloc(3POOL)	分配并返回池属性值的不透明容器。
pool_value_free(3POOL)	释放分配的属性值。
pool_value_set_bool(3POOL)	设置 boolean 类型的属性值。
pool_value_set_double(3POOL)	设置 double 类型的属性值。
pool_value_set_int64(3POOL)	设置 int64 类型的属性值。
pool_value_set_name(3POOL)	为池属性设置 name=value 对。
pool_value_set_string(3POOL)	复制已传递的字符串。
pool_value_set_uint64(3POOL)	设置 uint64 类型的属性值。

用于查询资源池和关联元素的函数

本节中列出的接口用于执行与池和关联元素相关的查询。

`pool_component_info`(3POOL)
　返回描述给定组件的字符串。

`pool_conf_info`(3POOL)
　返回描述整个配置的字符串。

`pool_conf_location`(3POOL)
　返回为指定配置的 `pool_conf_open()` 提供的位置字符串。

`pool_conf_status`(3POOL)
　返回池配置的有效性状态。

`pool_conf_validate`(3POOL)
　检查给定配置内容的有效性。

`pool_dynamic_location`(3POOL)
　返回池框架用于存储动态配置的位置。

`pool_error`(3POOL)
　返回通过调用资源池配置库函数记录的最终故障的错误值。

`pool_get_binding`(3POOL)
　返回正在运行的系统中的池名称，该池包含与指定进程绑定的资源集。

`pool_get_owning_resource`(3POOL)
　返回当前包含指定组件的资源。

`pool_get_pool`(3POOL)
　返回具有所提供配置中的指定名称的池。

`pool_get_property`(3POOL)
　检索元素中的已命名属性的值。

`pool_get_resource`(3POOL)
　返回具有所提供配置中的给定名称和类型的资源。

`pool_get_resource_binding`(3POOL)
　返回正在运行的系统中的池名称，该池包含与给定进程绑定的资源集。

`pool_get_status`(3POOL)
　检索池功能的当前状态。

`pool_info`(3POOL)
　返回指定池的说明。

`pool_query_components`(3POOL)
　检索与指定的属性列表匹配的所有资源组件。

`pool_query_pool_resources`(3POOL)
返回当前与池关联的以 NULL 结尾的资源数组。

`pool_query_pools`(3POOL)
返回与指定的属性列表匹配的池列表。

`pool_query_resource_components`(3POOL)
返回构成指定资源的以 NULL 结尾的组件数组。

`pool_query_resources`(3POOL)
返回与指定的属性列表匹配的资源列表。

`pool_resource_info`(3POOL)
返回指定资源的说明。

`pool_resource_type_list`(3POOL)
枚举此平台上的池框架支持的资源类型。

`pool_static_location`(3POOL)
返回池框架用于存储缺省池框架实例化配置的位置。

`pool_strerror`(3POOL)
返回每个有效池错误代码的说明。

`pool_value_get_bool`(3POOL)
获取 `boolean` 类型的属性值。

`pool_value_get_double`(3POOL)
获取 `double` 类型的属性值。

`pool_value_get_int64`(3POOL)
获取 `int64` 类型的属性值。

`pool_value_get_name`(3POOL)
返回为指定池属性指定的名称。

`pool_value_get_string`(3POOL)
获取 `string` 类型的属性值。

`pool_value_get_type`(3POOL)
返回指定的池值包含的数据类型。

`pool_value_get_uint64`(3POOL)
获取 `uint64` 类型的属性值。

`pool_version`(3POOL)
获取池库的版本号。

`pool_walk_components`(3POOL)
调用对资源中包含的所有组件的回调。

`pool_walk_pools`(3POOL)
调用在配置中定义的所有池的回调。

pool_walk_properties(3POOL)
调用对为给定元素定义的所有属性的回调。

pool_walk_resources(3POOL)
调用对与池关联的所有资源的回调。

资源池代码示例

本节包含资源池接口的代码示例。

确定资源池中的 CPU 数

sysconf(3C) 提供有关整个系统中的 CPU 数的信息。以下示例提供了用于确定特定应用程序的池 pset 中所定义 CPU 数的粒度。

本示例的要点包括以下内容：

- pvals[] 应该为以 NULL 结尾的数组。
- pool_query_pool_resources() 返回与应用程序的池 my_pool 中的 pvals 数组类型 pset 匹配的所有资源的列表。由于池只能有一个 pset 资源实例，因此每个实例始终在 nelem 中返回。reslist[] 仅包含一个元素，即 pset 资源。

```
pool_value_t *pvals[2] = {NULL};  /* pvals[] should be NULL terminated */



/* NOTE: Return value checking/error processing omitted */

/* in all examples for brevity */



conf_loc = pool_dynamic_location();

conf = pool_conf_alloc();

pool_conf_open(conf, conf_loc, PO_RDONLY);

my_pool_name = pool_get_binding(getpid());

my_pool = pool_get_pool(conf, my_pool_name);

pvals[0] = pool_value_alloc();

pvals2[2] = { NULL, NULL };
```

```
pool_value_set_name(pvals[0], "type");

pool_value_set_string(pvals[0], "pset");



reslist = pool_query_pool_resources(conf, my_pool, &nelem, pvals);

pool_value_free(pvals[0]);

pool_query_resource_components(conf, reslist[0], &nelem, NULL);

printf("pool %s: %u cpu", my_pool_ name, nelem);

pool_conf_close(conf);
```

列出所有的资源池

以下示例列出了在应用程序的池 `pset` 中定义的所有资源池。

该示例的要点包括以下内容：

- 采用 PO_RDONLY 以只读方式打开动态 `conf` 文件。`pool_query_pools()` 将池列表返回到 `pl` 中，并将池数目返回到 `nelem` 中。对于每个池，请调用 `pool_get_property()` 以将 `pool.name` 属性从元素放入 `pval` 值中。
- `pool_get_property()` 将调用 `pool_to_elem()` 以将 `libpool` 实体转换为不透明的值。`pool_value_get_string()` 将从不透明的池值中获取字符串。

```
conf    = pool_conf_alloc();

pool_conf_open(conf, pool_dynamic_location(), PO_RDONLY);

pl = pool_query_pools(conf, &nelem, NULL);

pval = pool_value_alloc();

for (i = 0; i < nelem; i++) {

    pool_get_property(conf, pool_to_elem(conf, pl[i]), "pool.name", pval);

    pool_value_get_string(pval, &fname);

    printf("%s\n", name);

}
```

```
pool_value_free(pval);

free(pl);

pool_conf_close(conf);
```

报告给定池的池统计信息

以下示例报告了指定池的统计信息。

该示例的要点包括以下内容：

- `pool_query_pool_resources()` 将获取 `rl` 中的所有资源的列表。由于 `pool_query_pool_resources()` 的最后一个参数为 NULL，因此将返回所有资源。对于每种资源，系统都会读取并列显 `name`、`load` 和 `size` 属性。
- 对 `strdup()` 的调用将分配本地内存并复制由 `get_string()` 返回的字符串。对 `get_string()` 的调用将返回一个指针，该指针由对 `get_property()` 的下一次调用释放。如果不包括对 `strdup()` 的调用，则对字符串的后续引用将导致应用程序出现故障，并出现段错误。

```
printf("pool %s\n:" pool_name);

pool = pool_get_pool(conf, pool_name);

rl = pool_query_pool_resources(conf, pool, &nelem, NULL);

for (i = 0; i < nelem; i++) {

  pool_get_property(conf, pool_resource_to_elem(conf, rl[i]), "type", pval);

  pool_value_get_string(pval, &type);

  type = strdup(type);

  snprintf(prop_name, 32, "%s.%s", type, "name");

  pool_get_property(conf, pool_resource_to_elem(conf, rl[i]),

        prop_name, pval);

  pool_value_get_string(val, &res_name);

  res_name = strdup(res_name);

  snprintf(prop_name, 32, "%s.%s", type, "load");
```

```
  pool_get_property(conf, pool_resource_to_elem(conf, rl[i]),
        prop_name, pval);
  pool_value_get_uint64(val, &load);
  snprintf(prop_name, 32, "%s.%s", type, "size");
  pool_get_property(conf, pool_resource_to_elem(conf, rl[i]),
        prop_name, pval);
  pool_value_get_uint64(val, &size);
  printf("resource %s: size %llu load %llu\n", res_name, size, load);
  free(type);
  free(res_name);
}
free(rl);
```

设置 pool.comment 属性并添加新属性

以下示例设置了 pset 的 pool.comment 属性。该示例还在 pool.newprop 中创建了新的属性。

该示例的要点包括以下内容：

- 在对 pool_conf_open() 的调用中，使用静态配置文件中的 PO_RDWR 要求调用方为超级用户。
- 要在运行此实用程序后提交对 pset 所做的更改，请发出 pooladm -c 命令。要使实用程序提交更改，请使用第二个非零参数调用 pool_conf_commit()。

```
pool_set_comment(const char *pool_name, const char *comment)
{
  pool_t *pool;
  pool_elem_t *pool_elem;
  pool_value_t *pval = pool_value_alloc();
```

```
  pool_conf_t  *conf = pool_conf_alloc();
  /* NOTE: need to be root to use PO_RDWR on static configuration file */
  pool_conf_open(conf, pool_static_location(), PO_RDWR);
  pool = pool_get_pool(conf,  pool_name);
  pool_value_set_string(pval, comment);
  pool_elem = pool_to_elem(conf, pool);
  pool_put_property(conf, pool_elem, "pool.comment", pval);
  printf("pool %s: pool.comment set to %s\n:" pool_name, comment);
  /* Now, create a new property, customized to installation site */
  pool_value_set_string(pval, "New String Property");
  pool_put_property(conf, pool_elem, "pool.newprop", pval);
  pool_conf_commit(conf, 0); /* NOTE: use 0 to ensure only */
                             /* static file gets updated */
  pool_value_free(pval);
  pool_conf_close(conf);
  pool_conf_free(conf);
  /* NOTE: Use "pooladm -c" later, or pool_conf_commit(conf, 1) */
  /* above for changes to the running system */
}
```

另一种修改池的注释并添加新的池属性的方法是使用 `poolcfg`(1M)。

```
poolcfg -c 'modify pool pool-name (string pool.comment = "cmt-string")'
poolcfg -c 'modify pool pool-name (string pool.newprop =
                                   "New String Property")'
```

与资源池关联的编程问题

编写应用程序时，请考虑以下问题。

- 每个站点都可以向池配置中添加其自己的属性列表。

  可以在多个配置文件中维护多个配置。系统管理员可以提交不同的文件，以反映在不同的时间段对资源占用情况的更改。根据负载条件，这些时间段可以包括不同的时间（日、周、月或季节）。
- 可以在池之间共享资源集，但是池只有一个给定类型的资源集。因此，可以在缺省应用程序数据库池与特定应用程序数据库池之间共享 `pset_default`。
- 请谨慎使用 `pool_value_*()` 接口。切记字符串池值的内存分配问题。请参见第 83 页中的 “报告给定池的池统计信息”。

◆ ◆ ◆ 第 7 章

7 有关 Solaris Zones 中资源管理应用程序的设计注意事项

本章简要概述了 Solaris Zones 技术，并介绍了编写资源管理应用程序的开发者可能遇到的潜在问题。有关区域的更多信息，请参见《系统管理指南：Solaris Containers－资源管理和 Solaris Zones》中的第 II 部分, “Zones”。

区域概述

区域是指在 Solaris 操作系统的单个实例中创建的虚拟化的操作系统环境。区域是一种为应用程序提供安全隔离环境的分区技术。创建区域时，便创建了一个应用程序执行环境，其中的进程与系统的其余部分相互隔离。这种隔离可防止在一个区域中运行的进程监视或影响在其他区域中运行的进程。即使运行的进程具有超级用户凭证，也不能查看或影响其他区域中的活动。区域还提供了一个抽象层，用于分隔应用程序和部署该区域的计算机的物理属性。这些属性的示例包括物理设备路径和网络接口名称。

缺省情况下，所有系统都有**全局区域**。全局区域对 Solaris 环境进行全局查看的方式与超级用户模型相似。其他所有区域都称为**非全局区域**。非全局区域类似于超级用户模型的非特权用户。非全局区域中的进程只能控制该区域中的进程和文件。通常，系统管理工作主要是在全局区域中执行的。在极少数需要隔离系统管理员的情况下，可以在非全局区域中使用特权应用程序。但是，通常资源管理活动在全局区域中进行。

有关区域中资源管理应用程序的设计注意事项

与在传统的 Solaris 环境中一样，所有应用程序在全局区域中都可以完全正常地运行。只要应用程序不需要任何权限，大多数应用程序便会在非全局环境中正常运行，不会出现任何问题。如果应用程序确实需要权限，则开发者需要仔细了解所需的权限以及特定权限的使用方式。如果需要权限，则系统管理员必须为应用程序指定相应的权限。

以下是开发者需要调查的已知情况：

- 更改系统时间的系统调用需要 PRIV_SYS_TIME 权限。这些系统调用包括 adjtime(2)、ntp_adjtime(2) 和 stime(2)。
- 需要对设置了 sticky 位的文件执行的系统调用需要 PRIV_SYS_CONFIG 权限。这些系统调用包括 chmod(2)、creat(2) 和 open(2)。
- ioctl(2) 系统调用需要 PRIV_SYS_NET_CONFIG 权限才能解除对 STREAMS 模块中的锚点的锁定。
- link(2) 和 unlink(2) 系统调用需要 PRIV_SYS_LINKDIR 权限才能在非全局区域中创建目录链接或解除目录链接。用于安装或配置软件或创建临时目录的应用程序可能会受到此限制的影响。
- PRIV_SYS_PROC_LOCK_MEMORY 权限是 mlock(3C)、munlock(3C)、mlockall(3C)、munlockall(3C) 和 plock(3C) 函数以及 memcntl(2) 系统的 MC_LOCK、MC_LOCKAS、MC_UNLOCK 和 MC_UNLOCKAS 标志所必需的。此限制会影响需要锁定和解除锁定内存的应用程序，如数据库程序。如果应用程序出于性能考虑而锁定内存，则使用 shmctl(2) 系统调用的锁定共享内存 (intimate shared memory, ISM) 可以提供潜在的解决方法。
- mknod(2) 系统调用需要 PRIV_SYS_DEVICES 权限才能创建块 (S_IFBLK) 或字符 (S_IFCHAR) 特殊文件。此限制会影响需要即时创建设备节点的应用程序。
- msgctl(2) 系统调用中的 IPC_SET 标志需要 PRIV_SYS_IPC_CONFIG 权限才能增加消息队列的字节数。此限制会影响任何需要动态调整消息队列大小的应用程序。
- nice(2) 系统调用需要 PRIV_PROC_PRIOCNTL 权限才能更改进程的优先级。此限制会影响用于设置调用优先级的应用程序。如果进程需要更改其优先级，则这种更改必须由进程在全局区域中进行。另一种更改优先级的方法是将正在运行应用程序的非全局区域绑定到资源池，尽管该区域中的调用进程最终会通过公平共享调度程序来决定。
- p_online(2) 系统调用中的 P_ONLINE、P_OFFLINE、P_NOINTR、P_FAULTED、P_SPARE 和 PZ-FORCED 标志需要 PRIV_SYS_RES_CONFIG 权限才能返回或更改进程操作状态。此限制会影响需要启用或禁用 CPU 的应用程序。
- priocntl(2) 系统调用中的 PC_SETPARMS 和 PC_SETXPARMS 标志需要 PRIV_PROC_PRIOCNTL 权限才能更改轻量进程 (lightweight process, LWP) 的调度参数。
- 需要管理处理器集 (pset) 的系统调用（包括将 LWP 绑定到 pset 和设置 pset 属性）需要 PRIV_SYS_RES_CONFIG 权限。此限制会影响以下系统调用 : pset_assign(2)、pset_bind(2)、pset_create(2)、pset_destroy(2) 和 pset_setattr(2)。
- shmctl(2) 系统调用中的 SHM_LOCK 和 SHM_UNLOCK 标志需要 PRIV_PROC_LOCK_MEMORY 权限才能共享内存控制操作。如果应用程序出于性能考虑而锁定内存，则使用锁定共享内存 (intimate shared memory, ISM) 功能可以提供潜在的解决方法。
- socket(3SOCKET) 函数需要 PRIV_NET_RAWACCESS 权限才能在协议设置为 IPPROTO_RAW 或 IPPROTO_IGMP 的情况下创建原始套接字。此限制会影响使用原始套接字或需要创建或检查 TCP/IP 头的应用程序。

- swapctl(2) 系统调用需要 PRIV_SYS_CONFIG 权限才能添加或删除交换资源。此限制会影响安装和配置软件。
- uadmin(2) 系统调用需要 PRIV_SYS_CONFIG 权限才能使用 A_REMOUNT、A_FREEZE、A_DUMP 和 AD_IBOOT 命令。此限制会影响在某些情况下需要强制进行崩溃转储的应用程序。
- clock_settime(3RT) 函数需要 PRIV_SYS_TIME 权限才能设置 CLOCK_REALTIME 和 CLOCK_HIRES 时钟。
- cpc_bind_cpu(3CPC) 函数需要 PRIV_CPC_CPU 权限才能将请求集绑定到硬件计数器。作为解决办法，可以使用 cpc_bind_curlwp(3CPC) 函数监视 CPU 计数器，找到有问题的 LWP。
- pthread_attr_setschedparam(3C) 函数需要 PRIV_PROC_PRIOCNTL 权限才能更改线程的基础调度策略和参数。
- timer_create(3RT) 函数需要 PRIV_PROC_CLOCK_HIGHRES 权限才能使用高精度系统时钟创建计时器。
- t_open(3NSL) 函数需要 PRIV_NET_RAWACCESS 权限才能建立传输端点。此限制会影响使用 /dev/rawip 设备实现网络协议的应用程序以及对 TCP/IP 数据包头执行操作的应用程序。
- 以下列出的库所提供的 API 在非全局区域中不受支持。共享对象存在于区域的 /usr/lib 目录中，因此，如果代码中包括对这些库的引用，将不会出现链接时错误。对自己的 make 文件进行检查可以确定应用程序是否已显式绑定到其中的任何一个库，并可以在执行应用程序时使用 pmap(1) 来验证是否未动态装入其中的任何库。
  - libdevinfo(3LIB)
  - libcfgadm(3LIB)
  - libpool(3LIB)
  - libtnfctl(3LIB)
  - libsysevent(3LIB)
- 区域包含一组受限制的设备，这些设备主要由构成 Solaris 编程 API 一部分的伪设备组成。这些伪设备包括 /dev/null、/dev/zero、/dev/poll、/dev/random、/dev/tcp 等。除非系统管理员已配置了设备，否则不能从区域中直接访问物理设备。由于设备通常是系统中的共享资源，因此，要使设备在区域中可用，需要按照以下方式设置一些限制，以免破坏系统安全性。
  - /dev 名称空间由符号链接（即逻辑路径）组成，这些符号链接指向 /devices 中的物理路径。/devices 名称空间只能在全局区域中使用，用于反映驱动程序所创建的附加设备实例的当前状态。仅有逻辑路径 /dev 在非全局区域中是可见的。
  - 非全局区域中的进程不能创建新设备节点。例如，mknod(2) 不能在非全局区域中创建特殊文件。如果指定了 /dev 中的文件，creat(2)、link(2)、mkdir(2)、rename(2)、 symlink(2) 和 unlink(2) 系统调用将会失败并发出 EACCES 错误。您可以创建指向 /dev 中的某一项的符号链接，但是不能在 /dev 中创建该链接。

- 公开系统数据的设备只能在全局区域中使用。此类设备的示例包括 dtrace(7D)、kmem(7D)、kmdb(7d)、ksyms(7D)、lockstat(7D) 和 trapstat(1M)。
- /dev 名称空间包含组成缺省“安全”驱动程序集的设备节点以及 zonecfg(1M) 命令为该区域指定的设备节点。
- 不能在非全局区域中访问支持 DLPI 编程接口的任何 NIC 设备，例如 hme(7D) 和 ce(7D)。

- 每个非全局区域都有自己的逻辑网络和回送接口。上层流与逻辑接口之间的绑定受到限制，因此流只能与相同区域中的逻辑接口建立绑定。与之类似，来自逻辑接口的数据包只能传递给与该逻辑接口在同一区域中的上层流。到回送地址的绑定将保留在区域内，但以下情况例外：当一个区域中的流试图访问另一区域中某一接口的 IP 地址时。尽管可以将一个区域中的应用程序绑定到具有权限的网络端口，但是它们无法控制网络配置（包括 IP 地址和路由表）。

◆ ◆ ◆ 第 8 章

8 配置示例

本章提供有关 /etc/project 文件的配置示例。

/etc/project 项目文件

项目文件是项目信息的本地源。项目文件可以与其他项目源（包括 NIS 映射 project.byname 和 project.bynumber 以及 LDAP 数据库项目）结合使用。程序使用 getprojent(3PROJECT) 例程访问此信息。

定义两个项目

/etc/project 可以定义两个项目： database 和 appserver。*user* 缺省值为 user.database 和 user.appserver。*admin* 缺省值可以在 user.database 和 user.appserver 之间切换。

```
hostname# cat /etc/project

.

.

.

user.database:2001:Database backend:admin::

user.appserver:2002:Application Server frontend:admin::
```

.

.

配置资源控制

/etc/project 文件提供了应用程序的资源控制。

```
hostname# cat /etc/project
```

.

.

.

```
development:2003:Developers:::task.ax-lwps=(privileged,10,deny);
```

```
process.max-addressspace=(privileged,209715200,deny)
```

.

.

配置资源池

/etc/project 文件提供了应用程序的资源池。

```
hostname# cat /etc/project
```

.

.

.

```
batch:2001:Batch project:::project.pool=batch_pool
```

```
process:2002:Process control:::project.pool=process_pool
```

.

.

.

为项目配置 FSS project.cpu-shares

为以下两个项目设置 FSS：*database* 和 *appserver*。*database* 项目有 20 个 CPU 份额。*appserver* 项目有 10 个 CPU 份额。

```
hostname# cat /etc/project

.

.

.

user.database:2001:database backend:admin::project.cpu-shares=(privileged,

     20,deny)

user.appserver:2002:Application Server frontend:admin::project.cpu-shares=

     (privileged,10,deny)

.

.

.
```

注 – “20,deny”和“(privileged,”前面行中的换行符在 /etc/project 文件中是无效的。此处显示的换行符只是为了方便示例在打印页面或显示页面上显示。/etc/project 文件中的每一项都必须占一行。

可以将 FSS 作为缺省的用户空间调度类进行分配。但是，在没有分配份额的情况下，调度类的行为与分时类的行为相同，因为所有的线程都存在于一个线程组中。可以采用一种特定方式将份额分配给运行的进程，而且还可以将其定义为项目属性。

配置五个具备不同特征的应用程序

以下示例配置了五个具备不同特征的应用程序。

表 8–1 目标应用程序和特征

应用程序类型和名称	特征
应用程序服务器，app_server。	CPU 超过 2 个时，可伸缩性会降低。将包含两个 CPU 的处理器集分配给 app_server。使用 TS 调度类。
数据库实例，app_db。	大量多线程。使用 FSS 调度类。
测试和开发，development。	基于 Motif。执行未经测试的代码。交互式调度类可以确保用户界面的响应性。使用 process.max-address-space 强加内存限制并使对整体运行情况的影响降至最低程度。
事务处理引擎，tp_engine。	响应时间极为重要。分配一个至少由两个 CPU 组成的专用集，以确保响应时间保持在最低程度。使用分时调度类。
独立数据库实例，geo_db。	大量多线程。为多个时区提供服务。使用 FSS 调度类。

注 – 将数据库应用程序（app.db 和 geo_db）整合到一个至少由四个 CPU 组成的处理器集中。使用 FSS 调度类。应用程序 app_db 将获得 project.cpu-shares 的 25%。应用程序 geo_db 将获得 project.cpu-shares 的 75%。

编辑 /etc/project 文件。将用户映射到 app_server、app_db、development、tp_engine 和 geo_db 项目项的资源池。

```
hostname# cat /etc/project

.

.

.

user.app_server:2001:Production Application Server::

     project.pool=appserver_pool

user.app_db:2002:App Server DB:::project.pool=db_pool,

     project.cpu-shares=(privileged,1,deny)

development:2003:Test and delopment::staff:project.pool=dev.pool,

     process.max-addressspace=(privileged,536870912,deny)

user.tp_engine:Transaction Engine:::project.pool=tp_pool
```

```
user.geo_db:EDI DB:::project.pool=db_pool;

     project.cpu-shares=(privileged,3,deny)
```

注 – 以“project.pool”、“project.cpu-shares=”、“process.max-addressspace”和“project.cpu-shares=”开始的行中的换行符在项目文件中是无效的。此处显示的换行符只是为了方便示例在打印页面或显示页面上显示。每一项都必须占用一行且必须仅占用一行。

为资源池创建 `pool.host` 脚本并添加项。

```
hostname# cat pool.host

create system host

create pset dev_pset (unit pset.max = 2)

create pset tp_pset (unit pset.min = 2)

create pset db_pset (unit pset.min = 4; uint pset.max = 6)

create pset app_pset (unit pset.min = 1; uint pset.max = 2)

create pool dev_pool (string pool.scheduler="IA")

create pool appserver_pool (string pool.scheduler="TS")

create pool db_pool (string pool.scheduler="FSS")

create pool tp_pool (string pool.scheduler="TS")

associate pool pool_default (pset pset_default)

associate pool dev_pool (pset dev_pset)

associate pool pool appserver_pool (pset app_pset)

associate pool db_pool (pset db_pset)

associate pool tp_pool (pset tp_pset)
```

注 – 以“boolean”开始的行中的换行符在 `pool.host` 文件中是无效的。此处显示的换行符只是为了方便示例在打印页面或显示页面上显示。每一项都必须占用一行且必须仅占用一行。

运行 pool.host 脚本并修改 pool.host 文件中指定的配置。

```
hostname# poolcfg –f pool.host
```

读取 pool.host 资源池配置文件并初始化系统上的资源池。

```
hostname# pooladm –c
```

索引