第2章　UNIX标准化及实现

2.1　引言

UNIX和C程序设计语言的标准化工作已经做了很多。虽然 UNIX应用程序在不同的UNIX版本之间进行移植相当容易，但是80年代UNIX版本的剧增以及它们之间差别的扩大导致很多大用户(例如美国政府)要求对其进行标准化。

本章将介绍正在进行的各种标准化工作，然后讨论这些标准对本书所列举的实际 UNIX实现的影响。所有标准化工作的一个重要部分是对每种实现必须定义的各种限制的说明，所以我们将说明这些限制以及确定它们值的多种方法。

2.2　UNIX标准化

2.2.1　ANSI C

1989年后期，C程序设计语言的ANSI标准X3.159-1989得到批准〔ANSI 1989〕。此标准已被采用为国际标准ISO/IEC 9899:1990。ANSI是美国国家标准学会，它是由制造商和用户组成的非赢利性组织。在美国，它是全国性的无偿标准交换站，在国际标准化组织 (ISO)中是代表美国的成员。

ANSI C标准的意图是提供C程序的可移植性，使其能适合于大量不同的操作系统，而不只是UNIX。此标准不仅定义了C程序设计语言的语法和语义，也定义了其标准库〔ANSI 1989第4章；Plauger 1992;Kernighan及Ritchie 1988中的附录B〕。因为很多新的UNIX系统（例如本书介绍的几个UNIX系统）都提供C标准中说明的库函数，所以此库对我们来讲是很重要的。

按照该标准定义的各个头文件，可将该库分成 15区。表2-1中列出了C标准定义的头文件，以及下面几节中说明的另外两个标准 (POSIX.1和XPG3)定义的头文件。在其中也列举了 SVR4和4.3+BSD所支持的头文件。本章也将对这两种UNIX实现进行说明。

表2-1　由各种标准和实现定义的头文件

头　文　件	标　准			实　现		说　明
	ANSI C	POSIX.1	XPG3	SVR4	4.3+BSD	
<assert.h>	•			•	•	验证程序断言
<cpio.h>		•		•		cpio归档值
<ctype.h>	•			•	•	字符类型
<dirent.h>		•	•	•	•	目录项(4.21节)
<errno.h>	•			•	•	出错码(1.7节)
<fcntl.h>		•	•	•	•	文件控制(3.13节)
<float.h>	•			•	•	浮点常数
<ftw.h>			•	•		文件树漫游 (4.21节)

（续）

头 文 件	标 准			实 现		说 明
	ANSI C	POSIX.1	XPG3	SVR4	4.3+BSD	
<grp.h>		•	•	•	•	组文件(6.4节)
<langinfo.h>			•	•		语言信息常数
<limits.h>	•			•	•	实施常数(2.5节)
<locale.h>	•			•	•	本地类别
<math.h>	•			•	•	数学常数
<nl_types.h>			•	•		消息类别
<pwd.h>		•	•	•	•	口令文件(6.2节)
<regex.h>			•	•	•	正则表达式
<search.h>			•	•		搜索表
<setjmp.h>	•			•	•	非局部goto(7.10节)
<signal.h>	•			•	•	信号(第10章)
<stdarg.h>	•			•	•	可变参数表
<stddef.h>	•			•	•	标准定义
<stdio.h>	•			•	•	标准I/O库(第5章)
<stdlib.h>	•			•	•	公用函数
<string.h>	•			•	•	字符串操作
<tar.h>		•		•		tar归档值
<termios.h>		•	•	•	•	终端I/O(第11章)
<time.h>	•			•	•	时间和日期(6.9节)
<ulimit.h>			•	•		用户限制
<unistd.h>		•	•	•	•	符号常数
<utime.h>		•	•	•	•	文件时间(4.19节)
<sys/ipc.h>			•	•	•	IPC(14.6节)
<sys/msg.h>			•	•		消息队列(14.7节)
<sys/sem.h>			•	•		信号量(14.8节)
<sys/shm.h>			•	•	•	共享存储(14.9节)
<sys/stat.h>		•	•	•	•	文件状态(第4章)
<sys/times.h>		•	•	•	•	进程时间(8.15节)
<sys/types.h>		•	•	•	•	原系统数据类型(2.7节)
<sys/utsname.h>		•	•	•		系统名(6.8节)
<sys/wait.h>		•	•	•	•	进程控制(8.6节)

2.2.2 IEEE POSIX

POSIX是一个由IEEE(电气和电子工程师学会)制订的标准族。POSIX的意思是计算机环境的可移植操作系统界面(Portable Operating System Interface for Computer Environment)。它原来指的只是IEEE标准1003.1-1988(操作系统界面)，但是，IEEE目前正在制订POSIX族中的其他有关标准。例如，1003.2将是针对shell和公用程序的标准，1003.7将是系统管理方面的标准。在1003工作组中至少有15个子委员会。

与本书相关的是1003.1操作系统界面标准，该标准定义了“POSIX依从的”操作系统必须

提供的服务。虽然1003.1标准是以UNIX操作系统为基础的，但是它又不仅仅限于UNIX和类似于UNIX的系统。确实，有些供应专有操作系统的制造商也声称这些系统将依从 POSIX(同时还保有它们的所有专有功能)。

由于1003.1标准说明了一个界面（interface）而不是一种实现（implementation），所以并不区分系统调用和库函数。所有在标准中的例程都被称为函数。

标准是不断演变的，1003.1标准也不例外。该标准的1988版，IEEE 1003.1-1988经修改后递交给ISO，没有增加新的界面或功能，但修改了文本。最终的文档作为 IEEE Std.1003.1-1990正式出版〔IEEE 1990〕，这也就是国际标准 ISO/IEC 9945-1:1990。该标准通常被称之为POSIX.1，本书将使用此标准。

IEEE 1003.1工作组此后对其又作了更多修改，它们在1993年被批准。这些修改(现在称之为1003.1a)应由IEEE作为IEEE标准1003.1-1990的附件出版，这些修改也对本书有所影响，主要是因为伯克利风格的符号链接很可能将被加到标准中作为一种要求的功能。这些修改也很可能成为ISO/IEC 9945-1:1990的一个附录。本书将用注释的方法来说明POSIX.1的1003.1a版本，指出哪些功能很可能会加到1003.1a中。

POSIX.1没有包括超级用户这样的概念。代之以规定某些操作要求“适当的优先权”，POSIX.1将此术语的定义留由具体实现进行解释。某些符合国防部安全性指导原则要求的 Unix系统具有很多不同的安全级。本书仍使用传统的UNIX术语，并指明要求超级用户特权的操作。

2.2.3 X/Open XPG3

X/Open是一个国际计算机制造商组织。它提出了一个7卷本可移植性指南X/Open Portability Guide （X/Open可移植性指南）第3版〔X／Open 1989〕，我们将其称之为XPG3。XPG3的第2卷XSI System Interface and Headers（XSI系统界面和头文件）对类似UNIX的系统定义了一个界面，该界面定义是在IEEE Std.1003.1-1988界面的基础上制订的。XPG3包含了一些POSIX.1没有的功能。

例如，POSIX.1没有但XPG3却有的一个功能是X/Open的消息设施，该设施可由应用程序使用以在不同的语言中显示文本消息。

XPG3界面使用了ANSI C草案而不是最后的正式标准，所以在 XPG3界面规格说明中包含的某些功能不再使用。这些问题很可能会在将来的 XPG规格说明的新版本中解决。（有关XPG4的工作正在进行，可能在1993年完成。）

2.2.4 FIPS

FIPS的含义是联邦信息处理标准(Federal Information Processing Standard)，这些标准是由美国政府出版的，并由美国政府用于计算机系统的采购。 FIPS151-1（1989年4月）基于IEEE Std.1003.1-1988及ANSI C标准草案。FIPS 151-1要求某些在POSIX.1中可选的功能。这种FIPS有时称为POSIX.1 FIPS。2.5.5节列出了FIPS所要求的POSIX.1的选择项。

POSIX.1 FIPS的影响是：它要求任一希望向美国政府销售 POSIX.1依从的计算机系统的厂商应支持POSIX.1的某些可选功能。我们将不把POSIX.1 FIPS视作为另一个标准，因为实际上它只是一个更加严格的POSIX.1标准。

2.3 UNIX实现

上面一节说明了三个由各自独立的组织所制定的标准： ANSI C、IEEE POSIX以及X/Open

XPG3。但是，标准只是界面的规格说明。这些标准是如何与现实世界相关连的呢？这些标准由制造商采用，然后转变成具体实施。本书中我们感兴趣的是这些标准和它们的具体实施。

在Leffler 等著作〔1989〕的1.1节中给出了UNIX族树的详细历史和关系图。UNIX的各种版本和变体都起源于在PDP-11系统上运行的UNIX分时系统第6版（1976年）和第7版（1979年）（通常称为V6和V7）。这两个版本是在贝尔实验室以外首先得到广泛应用的UNIX系统。从这棵树上发展出三个分支：(a) AT&T分支，从此导出了系统III和系统V(被称之为UNIX的商用版本)，(b)加州大学伯克利分校分支，从此导出4.xBSD实现，(c) 由AT&T贝尔实验室的计算科学研究中心不断开发的UNIX研究版本，从此导出第8、第9和第10版。

2.3.1 SVR4

SVR4是AT&T UNIX系统实验室的产品，它汇集了下列系统的功能：AT&T UNIX系统V第3.2版(SVR3.2)，Sun 公司的SunOS系统，加州大学伯克利分校的4.3BSD以及微软的Xenix系统（Xenix是在V7的基础上开发的，后来又采用了很多系统V的功能）。其源代码于1989年后期分发，在1990年则开始向最终用户提供。SVR4符合POSIX 1003.1标准和X/Open XPG3标准。

AT&T也出版了系统V界面定义(SVID)〔AT&T 1989〕。SVID第3版说明了UNIX系统要达到SVR4质量要求所应提供的功能。如同 POSIX.1一样，SVID说明了一个界面，而不是一种实现。对于一个具体实现的SVR4应查看其参考手册，以了解其不同之处〔AT&T 1990e〕。

SVR4包含了BSD的兼容库〔AT&T 1990c〕，它提供了功能与4.3BSD对应的函数和命令。但是其中某些函数与POSIX的对应部分有所不同，本书中所有的SVR4实例都没有使用此兼容库。只有你有一些早期的应用程序，又不想改变它们时，才建议使用此兼容库，新的应用程序不应使用它。

2.3.2 4.3+BSD

BSD是由加州大学伯克利分校的计算机系统研究组研究开发和分发的。4.2BSD于1983年问世，4.3BSD则在1986年。这两个版本都在 VAX小型机上运行。它们的下一个版本 4.3BSD Tahoe于1988年发布，在一台称为 Tahoe的小型机上运行 (Leffler等的著作〔1989〕说明了4.3BSD Tahoe版)。其后又有1990年的4.3BSD Reno版，它支持很多POSIX.1的功能。下一个主要版本4.4BSD应在1992年发布。

早期的BSD系统包含了AT&T专有的源代码，它们需要AT&T许可证。为了获得BSD系统的源代码，首先需要持有AT&T的UNIX许可证。这种情况正在得到改变，在近几年来愈来愈多的AT&T源代码正被代换成非AT&T源代码，很多加到BSD系统上的新功能也来自于非AT&T方面。

1989年，伯克利将4.3BSD Tahoe中很多非AT&T源代码包装成BSD网络软件，1.0版，并使其成为公众可用的软件。其后则有BSD网络软件的2.0版，它是从4.3BSD Reno版导出的，其目的是使大部分(如果不是全部的话)4.4BSD系统不再受AT&T许可证的限制，于是其全部源代码都可为公众使用。

> 正如前言中所说明的，本书使用术语 4.3+BSD表示BSD系统，该系统位于BSD网络软件2.0版和即将发布的4.4BSD之间。

在伯克利所进行的UNIX开发工作是从PDP-11开始的，然后转移到VAX小型机上，接着又转移到工作站上。90年代早期，伯克利得到支持在广泛应用的 80386个人计算机上开发BSD版

本，结果产生了 386BSD。这一工作是由 Bill Jolitz 完成的。其相关文档有发表在 1991年 Dr.Dobb´s Journal上的系列文章(每月一篇)。其中很多代码出现在BSD网络软件2.0版中。

2.4 标准和实现的关系

我们已提及的标准定义了任一实际系统的子集。虽然 IEEE POSIX正致力于在其他所需方面(例如，网络界面，进程间的通信，系统管理)制订出标准，但在编著本书写作时，这些标准还并不存在。

本书集中阐述了两个实际的UNIX系统：SVR4和4.3+BSD。因为这两个系统都宣称是依从POSIX的，所以我们一方面集中阐述了 POSIX.1标准所要求的功能，同时又指出 POSIX和这两个系统具体实现之间的差别。故 SVR4或4.3+BSD特有的功能和例程都被清楚地标记出来。因为XPG3是POSIX.1的超集，所以我们还叙述了属于XPG3，但不属于POSIX.1的功能。

应当了解， SVR4和4.3+BSD都提供了对它们早期版本功能的兼容性 (例如SVR3.2对4.3BSD)。例如， SVR4对POSIX规格说明中的非阻塞I/O(O_NONBLOCK)以及传统的系统V方法(O_NDELAY)都提供了支持。本书将只使用 POSIX.1的功能，但是也会提及它所替换的是哪一种非标准功能。与此相类似， SVR3.2和4.3BSD以某种方法提供了可靠信号机制，这种方法也有别于POSIX.1标准。第10章将只说明POSIX.1的信号机制。

2.5 限制

有很多由实现定义的幻数和常数，其中有很多已被编写到程序中，或由特定的技术所确定。由于大量标准化工作的努力，已有若干种可移植的方法用以确定这些幻数和实现定义的限制。这非常有助于软件的可移植性。

以下三种类型的功能是必需的：

- 编译时间选择项（该系统是否支持作业控制）。
- 编译时间限制（短整型的最大值是什么）。
- 运行时间限制（文件名的最大字符数为多少）。

前两个，编译时间选择项和限制可在头文件中定义。程序在编译时可以包含这些头文件。但是，运行时间限制则要求进程调用一个函数以获得此种限制值。

另外，某些限制在一个给定的实现中可能是固定的 (因此可以静态地在一个头文件中定义)，而在另一个实现上则可能是变动的（需要有一个运行时间函数调用）。这种类型限制的一个例子是文件名的最大字符数。系统 V由于历史原因只允许文件名有 14个字符，而伯克利的系统则将此增加为255。SVR4允许我们对每一个创建的文件系统指明是系统 V文件系统还是BSD文件系统，而每个系统有不同的限制。这就是运行时间限制的一个实例，文件名的最大长度依赖于文件所处的文件系统。例如，根文件系统中的文件名长度限制可能是 14个字符，而在某个其他文件系统中文件名长度限制可能是255个字符。

为了解决这些问题，提供了三种限制：

(1) 编辑时间选择项及限制（头文件）。

(2) 不与文件或目录相关联的运行时间限制。

(3) 与文件或目录相关联的运行时间限制。

使事情变得更加复杂的是，如果一个特定的运行时间限制在一个给定的系统上并不改变，则可将其静态地定义在一个头文件中，但是，如果没有将其定义在头文件中，则应用程序就必

须调用三个conf函数中的一个（我们很快就会对它们进行说明），以确定其运行时间值。

2.5.1 ANSI C限制

所有由ANSI C定义的限制都是编译时间限制。表2-2中列出了文件<limits.h>中定义的C标准限制。这些常数总是定义在该头文件中，而且在一个给定系统中并不会改变。第 3列列出了ANSI C标准可接受的最小值。这用于整型长度为 16位的系统，它使用1的补码表示。第4列列出了整型长度为32位的当前系统的值，用的是2的补码表示法。注意，对不带符号的数据类型都没有列出其最小值，它们都应为0。

我们将会遇到的一个区别是系统是否提供带符号（ signed）或不带符号的（unsigned）的字符值。从表 2-2中的第 4列可以看出，该特定系统使用带符号字符。从表中可以看到CHAR_MIN等于SCHAR_MIN，CHAR_MAX等于SCHAR_MAX。如果系统使用不带符号字符，则CHAR_MIN等于0，CHAR_MAX等于UCHAR_MAX。

在头文件<float.h>中，对浮点数据类型也有类似的一组定义。

我们会遇到的另一个ANSI C常数是FOPEN_MAX，这是实现保证的可同时打开的标准I/O流的最小数，该值在头文件<stdio.h>中定义，其最小值是8。POSIX.1中的STREAM_MAX（若定义的话），则应具与FOPEN_MAX相同的值。

ANSI C在<stdio.h>中也定义了常数TMP_MAX，这是由tmpnam函数产生的唯一文件名的最大数。关于此常数我们将在5.13节中进行更多说明。

表2-2 <limits.h>中的整型值大小

名 字	说 明	最小可接受值	典 型 值
CHAR_BIT	char的位	8	8
CHAR_MAX	char的最大值	(见后)	127
CHAR_MIN	char的最小值	(见后)	−128
SCHAR_MAX	带符号char的最大值	127	127
SCHAR_MIN	带符号char的最小值	−127	−128
UCHAR_MAX	不带符号char的最大值	255	255
INT_MAX	int最大值	32 767	2 147 483 647
INT_MIN	int最小值	−32 767	−2 147 483 648
UINT_MAX	不带符号的int的最大值	65 535	4 294 967 295
SHRT_MIN	short最小值	−32 767	−32 768
SHRT_MAX	short最大值	32 767	32 767
USHRT_MAX	不带符号的short的最大值	65 535	65 535
LONG_MAX	long最大值	2 147 483 647	2 147 483 647
LONG_MIN	long最小值	−2 147 483 647	−2 147 483 648
ULONG_MAX	不带符号的long的最大值	4 294 967 295	4 294 967 295
MB_LEN_MAX	一多字节字符常数中的最大字节数	1	1

2.5.2 POSIX限制

POSIX.1定义了很多涉及操作系统实现限制的常数，不幸的是，这是 POSIX.1中最令人迷惑不解的部分之一。

它包括33个限制和常数，它们被分成下列八类：

(1) 不变的最小值（表2-3中的13个常数）。

(2) 不变值：SSIZE_MAX。

(3) 运行时间不能增加的值：NGROUPS_MAX。

(4) 运行时间不变的值(可能不确定)：ARG_MAX, CHILD_MAX, OPEN_MAX, STREAM_MAX以及TZNAME_MAX。

(5) 路径名可变值(可能不确定)：LINK_MAX, MAX_CANON, MAX_INPUT, NAME_MAX, PATH_MAX以及PIPE_BUF。

(6) 编辑时间符号常数： _POSIX_SAVED_IDS, _POSIX_VERSION以及_POSIX_JOB_CONTROL。

(7) 执行时间符号常数： _POSIX_NO_TRUNC, _POSIX_VDISABLE以及_POSIX_CHOWN_RESTRICTED。

(8) 不再使用的常数：CLK_TCK。

在这33个限制和常数中，15个是必须定义的，其余的则按具体条件可定义可不定义。在2.5.4节中，在说明sysconf, pathconf和fpathconf函数时，我们描述了可定义可不定义的限制和常数（第4~8条）。在表2-7中我们总结了所有限制和常数。13个不变最小值则示于表2-3中。

表2-3 <limits.h>中的POSIX.1不变最小值

名 字	描 述	值
_POSIX_ARG_MAX	exec函数的参数长度	4096
_POSIX_CHILD_MAX	每个实际用户ID的子进程数	6
_POSIX_LINK__MAX	至一个文件的连接数	8
_POSIX_MAX_CANON	终端规范输入队列的字节数	255
_POSIX_MAX_INPUT	终端输入队列的可用空间	255
_POSIX_NAME_MAX	文件名中的字节数	14
_POSIX_NGROUPS_MAX	每个进程同时的添加组ID数	0
_POSIX_OPEN_MAX	每个进程的打开文件数	16
_POSIX_PATH_MAX	路径名中的字节数	255
_POSIX_PIPE_BUF	能原子地写到一管道的字节数	512
_POSIX_SSIZE_MAX	能存在ssize_t对象中的值	32 767
_POSIX_STREAM_MAX	一个进程能一次打开的标准I/O流数	8
_POSIX_TZNAME_MAX	时区名字节数	3

这些值是不变的——它们并不随系统而改变。它们指定了这些特征最严格的值。一个符合POSIX.1的实现应当提供至少这样大的值。这就是为什么将它们称为最小值的原因，虽然它们的名字都包含了MAX。另外，一个可移植的应用程序不应要求更大的值。我们将在本书的适当部分说明每一个常数的含意。

不幸的是，这些不变最小值中的某一些在实际应用中太小了。例如，目前UNIX系统每个进程可同时打开的文件数远远超过16，即使是1978年的V7也提供了20个。另外，_POSIX_PATH_MAX的最小值为255也太小了，路径名可能会超过这一限制。这意味着在编辑时不能使用这两个常数_POSIX_OPEN_MAX和_POSIX_PATH_MAX作为数组长度。

表2-3中的13个不变最小值的每一个都有一个相关的实现值，其名字是将表2-3中的名字删除前缀_POSIX_后构成的。(这13个实现值是本节开始部分所列出的2~5项：不变值、运行时不

能增加的值、运行时不变的值、以及路径名可变值。）问题是所有这13个实现值并不能确保在<limit.h>头文件中定义。某个特定值可能不在此头文件中定义的理由是：对于一个给定进程的实际值可能依赖于系统的存储器总量。如果没有在头文件中定义它们，则不能在编译时使用它们作为数组边界。所以，POSIX.1决定提供三个运行时间函数以供调用：sysconf, pathconf以及fpathconf，用它们可以在运行时间得到实际的实现值。但是，还有一个问题，因为其中某些值是由POSIX.1定义为“可能不确定的”(逻辑上无限的)，这就意味着该值没有实际上限。例如，SVR4的每个进程打开文件数在假想上是无限的，所以在SVR4中OPEN_MAX被认为是不确定的。2.5.7节还将讨论运行时间限制不确定的问题。

2.5.3 XPG3限制

XPG3定义了七个常数，它们总是包含在<limits.h>头文件中。POSIX.1则会把它们称之为不变最小值。它们列于表2-4中。这些值的大多数都涉及消息。

表2-4 <limits.h>中的XPG3不变最小值

名 字	说 明	最小可接受的值	典 型 值
NL_ARGMAX	调用printf和scanf的最大数字值	9	9
NL_LANGMAX	LANG环境变量中的最大字节数	14	14
NL_MSGMAX	最大消息数	32,767	32,767
NL_NMAX	在N对1映射字符中的最大字节数		1
NL_SETMAX	最大集合数	255	255
NL_TEXTMAX	消息字符串中的最大字节数	255	255
NZERO	缺省进程优先权	20	20

XPG3也定义了值PASS_MAX，作为口令中的最大有效字符数（不包括终止字符null），它可能包含在<limits.h>中。POSIX.1则把它称之为运行时间不变的值（可能不确定），其最小可接受的值是8。PASS_MAX值也可在运行时间用sysconf函数取得，该函数将在2.5.4节中说明。

2.5.4 sysconf、pathconf和fpathconf函数

我们已列出了一个实现必须支持的各种最小值，但是怎样才能找到一个特定系统实际支持的限制值呢？正如前面提到的，某些限制值在编译时是可用的，而另外一些则必须在运行时确定。我们也曾提及在一个给定的系统中某些限制值是不会更改的，而其他则与文件和目录相关联。运行时间限制是由调用下面三个函数中的一个而取得的。

```
#include <unistd.h>

long sysconf(int name);

log pathconf(const char *pathname, int name);

log fpathconf(int filedes, int name);
```

所有函数返回：若成功为相应值，若出错为-1（见后）

最后两个函数之间的差别是一个用路径名作为其参数，另一个则取文件描述符作为参数。

表2-5中列出了这三个函数所使用的*name*参数。以_SC_开始的常数用作为sysconf的参数，而以_PC_开始的常数则作为pathconf或fpathconf的参数。

对于pathconf 的参数*pathname*, fpathconf 的参数*filedes* 有很多限制。如果不满足其中任何一个限制，则结果是未定义的。

(1) _PC_MAX_CANON，_PC_MAX_INPUT以及_PC_VDISABLE所涉及的文件必须是终端文件。

(2) _PC_LINK_MAX所涉及的文件可以是文件或目录。如果是目录，则返回值用于目录本身(不用于目录内的文件名项)。

(3) _PC_NAME_MAX和_PC_NO_TRUNC所涉及的文件必须是目录，返回值用于该目录中的文件名。

(4) _PC_PATH_MAX涉及的必须是目录。当所指定的目录是工作目录时，返回值是相对路径名的最大长度。(不幸的是，这不是我们想要知道的一个绝对路径名的实际最大长度，我们将在2.5.7节中再回到这一问题上来。)

(5) _PC_PIPE_BUF所涉及的文件必须是管道， FIFO或目录。在管道或FIFO情况下，返回值是对所涉及的管道或FIFO的限制值。对于目录，返回值是对在该目录中创建的任一 FIFO的限制值。

(6) _PC_CHOWN_RESTRICTED必须是文件或目录。如果是目录，则返回值指明此选择项是否适用于该目录中的文件。

表2-5 对sysconf、pathconf和fpathconf的限制及*name*参数

限 制 名	说 明	*name* 参数
ARG_MAX	exec函数的参数最大长度（字节）	_SC_ARG_MAX
CHILD_MAX	每个实际用户ID的最大进程数	_SC_CHILD_MAX
clock ticks/ second	每秒时钟滴答数	_SC_CLK_TCK
NGROUPS_MAX	每个进程的最大同时添加组ID数	_SC_NGROUPS_MAX
OPEN_MAX	每个进程最大打开文件数	_SC_OPEN_MAX
PASS_MAX	口令中的最大有效字符数	_SC_PASS_MAX
STREAM_MAX	在任一时刻每个进程的最大标准I/O流数——如若定义，则其值一定与FOPEN_MAX相同	_SC_STREAM_MAX
TZNAME_MAX	时区名中的最大字节数	_SC_TZNAME_MAX
_POSIX_JOB_CONTROL	指明实现是否支持作业控制	_SC_JOB_CONTROL
_POSIX_SAVED_IDS	指明实现是否支持保存的设置-用户-ID和保存的设置-组-ID	_SC_SAVED_IDS
_POSIX_VERSION	指明POSIX.1版本	_SC_VERSION
_XOPEN_VERSION	指明XPG版本（非POSIX.1）	_SC_XOPEN_VERSION
LINK_MAX	文件连接数的最大值	_PC_LINK_MAX
MAX_CANON	在一终端规范输入队列的最大字节数	_PC_MAX_CANON
MAX_INPUT	终端输入队列可用空间的字节数	_PC_MAX_INPUT
NAME_MAX	文件名中的最大字节数（不包括null结束符）	_PC_NAME_MAX
PATH_MAX	相对路径名中的最大字节数（不包括null结束符）	_PC_PATH_MAX
PIPE_BUF	能原子地写到一管道的最大字节数	_PC_PIPE_BUF
_POSIX_CHOWN_RESTRICTED	指明使用chown是否受到限制	_PC_CHOWN_RESTRICTED
_POSIX_NO_TRUNC	指明若路径名长于NAME_MAX是否产生一错误	_PC_NO_TRUNC
_POSIX_VDISABLE	若定义，终端专用字符可用此值禁止	_PC_VDISABLE

我们需要更详细地说明这三个函数的不同返回值。

(1) 如果*name*不是表2-5第3列中的一个合适的常数，则所有这三个函数都返回－ 1，并将

error设置为EINVAL。

(2) 包含MAX的12个*name*以及*name*_PC_PIPE_BUF可能或者返回该变量的值(返回值≥O),或者返回－1，这表示该值是不确定的，此时并不更改errno的值。

(3) 对_SC_CLK_TCK的返回值是每秒的时钟滴答数，以用于 times函数的返回值（见8.15节)。

(4) 对_SC_VERSION的返回值以4位数和2位数分别表示此标准的年和月。这可能或者是198808L或199009L，或此标准某个以后版本的值。

(5) 对_SC_XOPEN_VERSION的返回值表示此系统所遵从的XPG版本，其当前值是3。

(6) _SC_JOB_CONTROL和_SC_SAVED_IDS是两个可选功能。若sysconf返回－1（没有更改errno）则不支持相应的功能。这两个功能也可在编译时从<unistd.h>头文件中决定。

(7) 对_PC_CHOWN_RESTRICTED和_PC_NO_TRUNC的返回值若为－1(不改变errno)，则表示对所指定的*pathname*或*filedes*不支持此功能。

(8) 对_PC_VDISABLE的返回值若为－1（不改变errno)，则表示对所指定的*pathname*或*filedes*不支持此功能。若支持此功能，则返回值是被用于禁止特定终端输入字符的字符值（见表11-6)。

实例

程序2-1用于打印所有这些限制，并处理一个限制未被定义的情况。

程序2-1　打印所有可能的sysconf和pathconf值

```
#include    <errno.h>
#include    "ourhdr.h"

static void pr_sysconf(char *, int);
static void pr_pathconf(char *, char *, int);

int
main(int argc, char *argv[])
{
    if (argc != 2)
        err_quit("usage: a.out <dirname>");

    pr_sysconf("ARG_MAX            =", _SC_ARG_MAX);
    pr_sysconf("CHILD_MAX          =", _SC_CHILD_MAX);
    pr_sysconf("clock ticks/second =", _SC_CLK_TCK);
    pr_sysconf("NGROUPS_MAX        =", _SC_NGROUPS_MAX);
    pr_sysconf("OPEN_MAX           =", _SC_OPEN_MAX);
#ifdef  _SC_STREAM_MAX
    pr_sysconf("STREAM_MAX         =", _SC_STREAM_MAX);
#endif
#ifdef  _SC_TZNAME_MAX
    pr_sysconf("TZNAME_MAX         =", _SC_TZNAME_MAX);
#endif
    pr_sysconf("_POSIX_JOB_CONTROL =", _SC_JOB_CONTROL);
    pr_sysconf("_POSIX_SAVED_IDS   =", _SC_SAVED_IDS);
    pr_sysconf("_POSIX_VERSION     =", _SC_VERSION);
    pr_pathconf("MAX_CANON       =", "/dev/tty", _PC_MAX_CANON);
    pr_pathconf("MAX_INPUT       =", "/dev/tty", _PC_MAX_INPUT);
    pr_pathconf("_POSIX_VDISABLE =", "/dev/tty", _PC_VDISABLE);
    pr_pathconf("LINK_MAX        =", argv[1], _PC_LINK_MAX);
    pr_pathconf("NAME_MAX        =", argv[1], _PC_NAME_MAX);
    pr_pathconf("PATH_MAX        =", argv[1], _PC_PATH_MAX);
    pr_pathconf("PIPE_BUF        =", argv[1], _PC_PIPE_BUF);
```

```
        pr_pathconf("_POSIX_NO_TRUNC =", argv[1], _PC_NO_TRUNC);
        pr_pathconf("_POSIX_CHOWN_RESTRICTED =", argv[1], _PC_CHOWN_RESTRICTED);

        exit(0);
}

static void
pr_sysconf(char *mesg, int name)
{
        long    val;

        fputs(mesg, stdout);
        errno = 0;
        if ( (val = sysconf(name)) < 0) {
                if (errno != 0)
                        err_sys("sysconf error");
                fputs(" (not defined)\n", stdout);
        } else
                printf(" %ld\n", val);
}

static void
pr_pathconf(char *mesg, char *path, int name)
{
        long    val;

        fputs(mesg, stdout);
        errno = 0;
        if ( (val = pathconf(path, name)) < 0) {
                if (errno != 0)
                        err_sys("pathconf error, path = %s", path);
                fputs(" (no limit)\n", stdout);
        } else
                printf(" %ld\n", val);
}
```

我们有条件地包括了两个常数，它们已被加至POSIX.1，但不是IEEE Std.1003.1-1988的一部分。表2-6显示了在几个不同的系统上，程序 2-1的样本输出。我们在4.14节中可以了解到，SVR4 S5文件系统是可以回溯到 V7的传统的系统V文件系统。UFS是伯克利快速文件系统的SVR4实现。

表2-6 配置限制的实例

限 制	SunOS 4.1.1	SVR4		4.3+BSD
		S5 filesys	UFS filesys	
ARG_MAX	1 048 576	5 120	5 120	20 480
CHILD_MAX	133	30	30	40
每秒时钟滴答	60	100	100	60
NGROUPS_MAX	16	16	16	16
OPEN_MAX	64	64	64	64
_POSIX_JOB_CONTROL	1	1	1	1
_POSIX_SAVED_IDS	1	1	1	未定义
_POSIX_VERSION	198 808	198 808	198 808	198 808
MAX_CANON	256	256	256	255
MAX_INPUT	256	512	512	255
_POSIX_VDISABLE	0	0	0	255
LINK_MAX	32 767	1 000	1 000	32 767

（续）

限　制	SunOS 4.1.1	SVR4		4.3+BSD
		S5 filesys	UFS filesys	
NAME_MAX	255	14	255	255
PATH_MAX	1 024	1 024	1 024	1 024
PIPE_BUF	4 096	5 120	5 120	512
_POSIX_NO_TRUNC	1	未定义	1	1
_POSIX_CHOWN_RESTRICTED	1	未定义	未定义	1

2.5.5 FIPS 151-1要求

FIPS 151-1标准(我们已在2.2.4节中提及)由于要求下列功能，所以它比POSIX.1标准更严：

• 要求下列POSIX.1可选功能：_POSIX_JOB_CONTROL, _POSIX_SAVED_IDS, _POSIX_NO_TRUNC, _POSIX_CHOWN_RESTRICTED和_POSIX_VDISABLE。

• NGROUPS_MAX的最小值是8。

• 新创建的文件或目录的组ID应设置为它所在目录的组ID（4.6节将说明此功能）。

• 已传输了一些数据后，若read或write被一个捕捉到的信号所中断，则这些函数应返回已被传输的字节数（10.5节将讨论被中断的系统调用）。

• 登录shell应定义环境变量HOME和LOGNAME。

因为美国政府购买了很多计算机系统，所以大多数 POSIX的制造商都将支持这些增加的FIPS要求。

2.5.6 限制总结

我们已说明了很多限制和幻常数，其中某些必须包含在头文件中，某些可选地包含在头文件中，其他则可在运行时决定。表 2-7以字母顺序总结了所有这些常数以及得到它们值的各种方法。以_SC_开始的运行时间名字是 sysconf函数的参数，以 _PC_开始的名字是 pathconf和fpathconf函数的参数，如果它有最小值，则也将其列于其中。

注意，表2-3中的13个POSIX.1不变最小值示于表2-7中的最右一列。

表2-7　编译时间和运行时间限制总结

常　数　名	编　译　时　间		运行时间名	最　小　值
	头　文　件	要　求　否		
ARG_MAX	<limits.h>	可选	_SC_ARG_MAX	_POSIX_ARG_MAX=4096
CHAR_BIT	<limits.h>	要求		8
CHAR_MAX	<limits.h>	要求		127
CHAR_MIN	<limits.h>	要求		0
CHILD_MAX	<limits.h>	可选	_SC_CHILD_MAX	_POSIX_CHILD_MAX=6
每秒时钟滴答			_SC_CLK_TCK	
FOPEN_MAX	<stdio.h>	要求		8
INT_MAX	<limits.h>	要求		32 767
INT_MIN	<limits.h>	要求		−32 767
LINK_MAX	<limits.h>	可选	_PC_LINK_MAX	_POSIX_LINK_MAX=8
LONG_MAX	<limits.h>	要求		2 147 483 647

（续）

常 数 名	编 译 时 间		运行时间名	最 小 值
	头 文 件	要 求 否		
LONG_MIN	<limits.h>	要求		−2 147 483 647
MAX_CANON	<limits.h>	可选	_PC_MAX_CANON	_POSIX_MAX_CANON=255
MAX_INPUT	<limits.h>	可选	_PC_MAX_INPUT	_POSIX_MAX_INPUT=255
MB_LEN_MAX	<limits.h>	要求		
NAME_MAX	<limits.h>	可选	_PC_NAME_MAX	_POSIX_NAME_MAX=14
NGROUPS_MAX	<limits.h>	要求	_SC_NGROUPS_MAX	_POSIX_NGROUPS_MAX=0
NL_ARGMAX	<limits.h>	要求		9
NL_LANGMAX	<limits.h>	要求		14
NL_MSGMAX	<limits.h>	要求		32 767
NL_NMAX	<limits.h>	要求		
NL_SETMAX	<limits.h>	要求		255
NL_TEXTMAX	<limits.h>	要求		255
NZERO	<limits.h>	要求		20
OPEN_MAX	<limits.h>	可选	_SC_OPEN_MAX	_POSIX_OPEN_MAX=16
PASS_MAX	<limits.h>	可选	_SC_PASS_MAX	8
PATH_MAX	<limits.h>	可选	_PC_PATH_MAX	_POSIX_PATH_MAX=255
PIPE_BUF	<limits.h>	可选	_PC_PIPE_BUF	_POSIX_PIPE_BUF=512
SCHAR_MAX	<limits.h>	要求		127
SCHAR_MIN	<limits.h>	要求		−127
SHRT_MAX	<limits.h>	要求		32 767
SHRT_MIN	<limits.h>	要求		−32 767
SSIZE_MAX	<limits.h>	要求		_POSIX_SSIZE_MAX=32,767
STREAM_MAX	<limits.h>	可选	_SC_STREAM_MAX	_POSIX_STREAM_MAX=8
TMP_MAX	<stdio.h>	要求		10,000
TZNAME_MAX	<limits.h>	可选	_SC_TZNAME_MAX	_POSIX_TZNAME_MAX=3
UCHAR_MAX	<limits.h>	要求		255
UINT_MAX	<limits.h>	要求		65 535
ULONG_MAX	<limits.h>	要求		4 294 967 295
USHRT_MAX	<limits.h>	要求		65 535
_POSIX_CHOWN_RESTRICTED	<unistd.h>	可选	_PC_CHOWN_RESTRICTED	
_POSIX_JOB_CONTROL	<unistd.h>	可选	_SC_JOB_CONTROL	
_POSIX_NO_TRUNC	<unistd.h>	可选	_PC_NO_TRUNC	
_POSIX_SAVED_IDS	<unistd.h>	可选	_SC_SAVED_IDS	
_POSIX_VDISABLE	<unistd.h>	可选	_PC_VDISABLE	
_POSIX_VERSION	<unistd.h>	要求	_SC_VERSION	
_XOPEN_VERSION	<unistd.h>	要求	_SC_XOPEN_VERSION	

2.5.7 未确定的运行时间限制

我们已提及表2-7中的某些值可能是未确定的，这些值在第三列中标记为可选的 (optional)，其名字中包含MAX，或PIPE_BUF。我们遇到的问题是如果这些值没有在头文件 <limits.h>中定义，那么在编译时我们也就不能使用它们。但是，如果它们的值是未确定的，那么在运行时它们可能也是未定义的。让我们观察两个特殊的例子 —— 为一个路径名分配存储器，以及决定文件描述符的数目。

1. 路径名

很多程序需要为路径名分配存储器，一般来说，在编译时就为其分配了存储器，而且使用了各种幻数（其中很少是正确的）作为数组长度： 256，512，1024或标准I/O常数BUFSIZ。4.3BSD头文件<sys/param.h>中的常数MAXPATHLEN是正确值，但是很多4.3BSD应用程序并未用它。

POSIX.1试图用PATH_MAX值来帮助我们，但是如果此值是不确定的，那么仍是毫无帮助的。程序2-2是一个全书都将使用的为路径名动态地分配存储器的函数。

如若在<limits.h>中定义了常数PATH_MAX,那么就没有任何问题，如果没有，则需调用pathconf。因为pathconf的返回值是把第一个参数视为基于工作目录的相对路径名。所以指定根为第一个参数，并将得到的返回值加1作为结果值。如果pathconf指明PATH_MAX是不确定的，那么我们就只得猜测某个值。调用 malloc时的+1是为了在尾端加null字符（PATH_MAX没有考虑它）。

处理不确定结果情况的正确方法与如何使用分配到的存储空间有关。例如，如果我们为getcwd调用分配空间（返回当前工作目录的绝对路径名，见 4.22节），而分配到的空间太小，则返回一个出错， errno设置为ERANGE。然后可调用realloc以增加分配空间(见7.8节和练习4.18)并再试。不断重复此操作，直到getcwd调用成功执行。

程序2-2　为路径名动态地分配空间

```
#include    <errno.h>
#include    <limits.h>
#include    "ourhdr.h"

#ifdef  PATH_MAX
static int  pathmax = PATH_MAX;
#else
static int  pathmax = 0;
#endif

#define PATH_MAX_GUESS  1024    /* if PATH_MAX is indeterminate */
                        /* we're not guaranteed this is adequate */
char *
path_alloc(int *size)
                /* also return allocated size, if nonnull */
{
    char    *ptr;

    if (pathmax == 0) {     /* first time through */
        errno = 0;
        if ( (pathmax = pathconf("/", _PC_PATH_MAX)) < 0) {
            if (errno == 0)
                pathmax = PATH_MAX_GUESS;   /* it's indeterminate */
            else
                err_sys("pathconf error for _PC_PATH_MAX");
        } else
            pathmax++;      /* add one since it's relative to root */
    }

    if ( (ptr = malloc(pathmax + 1)) == NULL)
        err_sys("malloc error for pathname");

    if (size != NULL)
        *size = pathmax + 1;
    return(ptr);
}
```

2. 最大打开文件数

在精灵进程(是在后台运行，不与终端相连接的一种进程)中一个常见的代码序列是关闭所有打开文件。某些程序中有下列形式的代码序列：

```
#include <sys/param.h>
for (i = 0; i < NOFILE; i++)
    close(i);
```

程序假定在<sys/param.h>头文件中定义了常数NOFILE。另外一些程序则使用某些<stdio.h>版本提供作为上限的常数_NFILE。某些程序则直接将其上限值定为20。

我们希望用POSIX.1的OPEN_MAX确定此值以提高可移植性，但是如果此值是不确定的，则仍然有问题，如果我们使用下列代码

```
#include <unistd.h>

for (i = 0; i < sysconf(_SC_OPEN_MAX); i++)
    close(i);
```

而且如果OPEN_MAX是不确定的，那么sysconf将返回−1，于是，for循环根本不会执行。在这种情况下，最好的选择就是关闭所有描述符直至某个任一的限制值 (例如256)。如同上面的路径名一样，这并不能保证在所有情况下都能正确工作，但这却是我们所能选择的最好方法。程序2-3中使用了这种技术。

程序2-3 确定文件描述符数

```
#include    <errno.h>
#include    <limits.h>
#include    "ourhdr.h"

#ifdef  OPEN_MAX
static int  openmax = OPEN_MAX;
#else
static int  openmax = 0;
#endif

#define OPEN_MAX_GUESS  256     /* if OPEN_MAX is indeterminate */
                        /* we're not guaranteed this is adequate */
int
open_max(void)
{
    if (openmax == 0) {     /* first time through */
        errno = 0;
        if ( (openmax = sysconf(_SC_OPEN_MAX)) < 0) {
            if (errno == 0)
                openmax = OPEN_MAX_GUESS;   /* it's indeterminate */
            else
                err_sys("sysconf error for _SC_OPEN_MAX");
        }
    }
    return(openmax);
}
```

我们可以耐心地调用 close，直至得到一个出错返回，但是从 close（EBADF）出错返回并不区分无效描述符和并未打开的描述符。如果试用此技术，而且描述符 9未打开，描述符10打开了，那么将停止在 9上，而不会关闭 10。dup函数（见 3.12节）在超过了 OPEN_MAX时会返回一个特定的出错值，但是用复制一个描述符一、二百次的方法来确定此值是一种极

端的方法。

> SVR4和4.3+BSD的getrlimit(2)函数（见7.11节）以及4.3+BSD的getdtablesize(2)函数返回一个进程可以打开的最大描述符数，但是这两个函数是不可移植的。
>
> OPEN_MAX被POSIX称为运行时间不变值，其意思是在一个进程的生命期间其值不应被改变，但是在SVR4和4.3+BSD之下，可以调用setrlimit (2) 函数更改一个运行进程的此值（也可用C Shell的limit或Bourne shell和KornShell的ulimit命令更改）。如果系统支持这种功能，则可以将程序 2-3更改为每次调用此程序时就调用sysconf，而不只是第一次调用此程序时。

2.6 功能测试宏

正如前述，在头文件中定义了很多 POSIX.1和XPG3的符号。但是除了 POSIX.1和XPG3定义外，大多数实现在这些头文件中也加上了它们自己的定义。如果在编译一个程序时，希望它只使用POSIX定义而不使用任何实现定义的限制，那么就需定义常数 _POSIX_SOURCE，所有POSIX.1头文件中都使用此常数。当该常数定义时，就能排除任何实现专有的定义。

常数_POSIX_SOURCE及其对应的常数 _XOPEN_SOURCE被称之为功能测试宏（ feature test macro）。所有功能测试宏都以下划线开始。当要使用它们时，通常在 cc命令行中以下列方式定义：

```
cc -D_POSIX_SOURCE file.c
```

这使得在C程序包括任何头文件之前，定义了功能测试宏。如果我们仅想使用 POSIX.1定义，那么也可将源文件的第一行设置为：

```
#define _POSIX_SOURCE 1
```

另一个功能测试宏是：_ _STDC_ _，它由符合ANSI C标准的编译程序自动定义。这样就允许我们编写ANSI C编译程序和非ANSI C编译程序都能编译的程序。例如，一个头文件可能会是：

```
#ifdef __STDC__
void *myfunc(const char *, int);
#else
void *myfunc();
#endif
```

这样就能发挥ANSI C原型功能的长处，要注意在开始和结束处的两个连续的下划线常常打印成一个长下划线(如同上面一个样本源代码中一样)。

2.7 基本系统数据类型

历史上，某些UNIX变量已与某些C数据类型联系在一起，例如，历史上主、次设备号存放在一个16位的短整型中，8位表示主设备号，另外8位表示次设备号。但是，很多较大的系统需要用多于256个值来表示其设备号，于是，就需要有一种不同的技术。（确实，SVR4用32位表示设备号：14位用于主设备号，18位用于次设备号。）

头文件<sys/types.h>中定义了某些与实现有关的数据类型，它们被称之为基本系统数据类

型（primitive system data type）。有很多这种数据类型定义在其他头文件中。在头文件中这些数据类型都是用C的typedef设施来定义的。它们绝大多数都以_t 结尾。表2-8中列出了本书将使用的基本系统数据类型。

表2-8 基本系统数据类型

类 型	说 明
caddr_t	内存地址（12.9节）
clock_t	时钟滴答计数器（进程时间）（1.10节）
comp_t	压缩的时钟滴答（8.13节）
dev_t	设备号（主和次）（4.23节）
fd_set	文件描述符集（12.5.1节）
fpos_t	文件位置（5.10节）
gid_t	数值组ID
ino_t	i节点编号（4.14节）
mode_t	文件类型，文件创建方式（4.5节）
n1ink_t	目录项的连接计数（4.10节）
off_t	文件长度和位移量（带符号的）（lseek, 3.6节）
pid_t	进程ID和进程组ID（带符号的）（8.2和9.4节）
ptrdiff_t	两个指针相减的结果（带符号的）
r1im_t	资源限制（7.11节）
sig_atomic_t	能原子地存取的数据类型（10.15节）
sigset_t	信号集（10.11节）
size_t	对象（例如字符串）长度（不带符号的）（3.7节）
ssize_t	返回字节计数的函数（带符号的）（read, write, 3.7节）
time_t	日历时间的秒计数器（1.10节）
uid_t	数值用户ID
wchar_t	能表示所有不同的字符码

用这种方式定义了这些数据类型后，在编译时就不再需要考虑随系统不同而变的实施细节，在本书中涉及到这些数据类型的地方，我们会说明为什么使用它们。

2.8 标准之间的冲突

就整体而言，这些不同的标准之间配合得是相当好的。但是我们也很关注它们之间的差别，特别是ANSI C标准和POSIX.1之间的差别。(因为XPG3是一个较老的正在被修订的标准，FIPS则是一个要求更严的POSIX.1。)

ANSI C定义了函数clock，它返回进程使用的CPU时间，返回值是clock_t类型值。为了将此值变换成以秒为单位，将其除以在<time.h>头文件中定义的CLOCKS_PER_SEC。POSIX.1定义了函数times，它返回其调用者及其所有终止子进程的 CPU时间以及时钟时间，所有这些值都是clock_t类型值。IEEE Std.1003.1-1988将符号CLK_TCK定义为每秒滴答数，上述clock_t值都是以此度量的。而1990 POSIX.1标准中则说明不再使用，CLK_TCK而使用sysconf函数来获得每秒滴答数，并将其用于 times函数的返回值。术语是同一个，每秒滴答数，但 ANSI C和POSIX.1的定义却不同。这两个标准也用同一数据类型 (clock_t)来保存这些不同的值，这种差别可以在SVR4中看到，其中clock返回微秒数(CLOCK_PER_SEC是一百万)，而CLK_TCK通常是50、60或100(与CPU类型有关)。

另一个可能产生冲突的区域是：在ANSI C标准说明函数时，ANSI C所说明的函数可能会没有考虑到POSIX.1的某些要求。有些函数在POSIX环境下可能要求有一个与C环境下不同的实现，因为POSIX环境中有多个进程，而C语言环境则很少会考虑宿主操作系统。尽管如此，很多POSIX依从的系统为了兼容性的关系也实现ANSI C函数，signal函数就是一个例子。如果在不了解的情况下使用了SVR4所提供的signal函数(希望编写可在ANSI C环境和较早UNIX系统中运行的可兼容程序)，那么它提供了与POSIX.1 sigaction函数不同的语义。第10章将对signal函数作更多说明。

2.9 小结

在过去几年中，在UNIX不同版本的标准化方面已经取得了很大进展。本章对三个主要标准——ANSI C、POSIX和XPG3——进行了说明，也分析了这些标准对本书主要关注的两个实现：SVR4和4.3+BSD所产生的影响。这些标准都试图定义一些可能随实现而更改的参数，但是我们已经看到这些限制并不完善。本书将涉及所有这些限制和幻常数。

在本书最后的参考书目中，说明了如何订购这些标准的方法。

习题

2.1 2.7节中提到一些基本系统数据类型可以在多个头文件中定义。例如，size_t就在6个不同的头文件中都有定义。由于一个程序可能包含这6个不同的头文件，但是ANSI C不允许对同一个名字进行多次类型定义，如何处理这个矛盾呢？

2.2 检查系统的头文件，列出实现基本系统数据类型所用到的实际数据类型。