第6章　系统数据文件和信息

6.1　引言

有很多操作需要使用一些与系统有关的数据文件，例如，口令文件 /etc/passwd和组文件/etc/group就是经常由多种程序使用的两个文件。用户每次登录入 UNIX系统，以及每次执行ls -l命令时都要使用口令文件。

由于历史原因，这些数据文件都是ASCII文本文件，并且使用标准I/O库读这些文件。但是，对于较大的系统，顺序扫描口令文件变得很花费时间，我们想以非 ASCII文本格式存放这些文件，但仍向应用程序提供一个可以处理任何一种文件格式的界面。对于这些数据文件的可移植界面是本章的主题。本章也包括了系统标识函数、时间和日期函数。

6.2　口令文件

UNIX口令文件(POSIX.1则将其称为用户数据库)包含了表6-1中所示的各字段，这些字段包含在<pwd.h>中定义的passwd结构中。

> 注意，POSIX.1只指定passwd结构中7个字段中的5个。另外2个元素由SVR4和4.3+BSD支持。

表6-1　/etc/passwd文件中的字段

说　明	struct passwd成员		POSIX.1
用户名	char	*pw_name	•
加密口令	char	*pw_passwd	
数值用户ID	uid_t	pw_uid	•
数值组ID	gid_t	pw_gid	•
注释字段	char	*pw_gecos	
初始工作目录	char	*pw_dir	•
初始shell（用户程序）	char	*pw_shell	•

由于历史原因，口令文件是/etc/passwd，而且是一个文本文件。每一行包含表6-1中所示的7个字段，字段之间用冒号相分隔。例如，该文件中可能有下列三行：

```
root:jheVopR58x9Fx:0:1:The superuser:/:/bin/sh
nobody:*:65534:65534::/:
stevens:3hKVD8R58r9Fx:224:20:Richard Stevens:/home/stevens:/bin/ksh
```

关于这些登录项请注意下列各点：

- 通常有一个登录项，其用户名为root，其用户ID是0 (超级用户)。
- 加密口令字段包含了经单向密码算法处理过的用户口令副本。因为此算法是单向的，所

以不能从加密口令猜测到原来的口令。当前使用的算法(见Morris和Thompson〔1979〕)总是产生13个可打印字符(在64字符集〔a-zA-Z0-9./〕中)。因为用户名nobody的加密口令字段只包含一个字符(*)，所以加密口令决不会与此值相匹配。此用户名可用于网络服务器，这些服务器允许我们登录到一个系统，但其用户ID和组ID(65534)并不提供优先权。用此用户ID和组ID可存取的文件只是大家都可读、写的文件。(这假定用户ID65534和组ID65534并不拥有任何文件。)在本节稍后部分我们将讨论对口令文件最近所作的更改(阴影口令)。

• 口令文件中的某些字段可能是空。如果密码口令字段为空，这通常就意味着该用户没有口令(不推荐这样做)。nobody有两个空白字段：注释字段和初始shell字段。空白注释字段不产生任何影响。空白shell字段则表示取系统默认值，通常是/bin/sh。

• 支持finger(1)命令的某些UNIX系统支持注释字段中的附加信息。其中，各部分之间都用逗号分隔：用户姓名，办公室地点，办公室电话号码，家庭电话号码。另外，如果注释字段中的用户姓名是一个&，则它被代换为登录名。例如，可以有下列记录：

```
stevens:3hKVD8R58r9Fx:224:20:Richard &, B232, 555-1111, 555-2222:
                    /home/stevens:/bin/ksh
```

> 即使你所使用的系统并不支持finger命令，这些信息仍可存放在注释字段中，因为该字段只是一个注释，并不由系统公用程序解释。

POSIX.1只定义了两个存取口令文件中信息的函数。在给出用户登录名或数值用户ID后，这两个函数就能查看相关记录。

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

struct passwd *getpwuid(uid_t uid);

struct passwd *getpwnam(const char *name);
```

两个函数返回：若成功则为指针，若出错则为NULL

getpwuid由ls(1)程序使用，以便将包含在一个i节点中的数值用户ID映照为用户登录名。getpwnam在键入登录名时由login(1)程序使用。

这两个函数都返回一个指向passwd结构的指针，该结构已由这两个函数在执行时填入了所需的信息。该结构通常是在相关函数内的静态变量，只要调用相关函数，其内容就会被重写。

如果要查看的只是一个登录名或用户ID，那么这两个POSIX.1函数能满足要求，但是也有些程序要查看整个口令文件。下列三个函数则可用于此。

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

struct passwd *getpwent(void);
```

返回：若成功则为指针，若出错或到达文件尾端则为NULL

```
void setpwent(void);

void endpwent(void);
```

POSIX.1没有定义这三个函数，但它们受到SVR4和4.3+BSD的支持。

调用getpwent时，它返回口令文件中的下一个记录。如同上面所述的两个 POSIX.1函数一样，它返回一个由它填写好的password结构的指针。每次调用此函数时都重写该结构。在第一次调用该函数时，它打开它所使用的各个文件。在使用本函数时，对口令文件中各个记录安排的顺序并无要求。

函数setpwent反绕它所使用的文件， endpwent则关闭这些文件。在使用 getpwent查看完口令文件后，一定要调用 endpwent关闭这些文件。getpwent知道什么时间它应当打开它所使用的文件(第一次被调用时)，但是它并不能知道何时关闭这些文件。

实例

程序6-1给出了函数getpwnam的一个实现。

程序6-1　getpwnam函数

```
#include <sys/types.h>
#include <pwd.h>
#include <stddef.h>
#include <string.h>

struct passwd *
getpwnam(const char *name)
{
    struct passwd  *ptr;

    setpwent();
    while ( (ptr = getpwent()) != NULL) {
        if (strcmp(name, ptr->pw_name) == 0)
            break;       /* found a match */
    }
    endpwent();
    return(ptr);     /* ptr is NULL if no match found */
}
```

在程序开始处调用 setpwent是保护性的措施，以便在调用者在此之前已经调用过 getpwent的情况下，反绕有关文件使它们定位到文件开始处。 getpwnam和getpwuid完成后不应使有关文件仍处于打开状态，所以应调用 endpwent关闭它们。

6.3 阴影口令

在上一节我们曾提及，对UNIX口令通常使用的加密算法是单向算法。给出一个密码口令，找不到一种算法可以将其反变换到普通文本口令 (普通文本口令是在 Password:提示后键入的口令)。但是可以对口令进行猜测，将猜测的口令经单向算法变换成加密形成，然后将其与用户的加密口令相比较。如果用户口令是随机选择的，那么这种方法并不是很有用。但是用户往往以非随机方式选择口令(配偶的姓名、街名、宠物名等)。一个经常重复的试验是先得到一份口令文件，然后用试探方法猜测口令。 (Garfinkel和Spafford〔1991〕的第2章对UNIX口令及口令加密处理方案的历史情况及细节进行了说明。)

为使企图这样做的人难以获得原始资料 (加密口令)，某些系统将加密口令存放在另一个通常称为阴影口令（shadow password）的文件中。该文件至少要包含用户名和加密口令。与该口令相关的其他信息也可存放在该文件中。例如，具有阴影口令的系统经常要求用户在一定时

间间隔后选择一个新口令。这被称之为口令时效，选择新口令的时间间隔长度经常也存放在阴影口令文件中。

> 在SVR4中，阴影口令文件是 /etc/shadow。在4.3+BSD中，加密口令存放在 /etc/master. passwd 中。

阴影口令文件不应是一般用户可以读取的。仅有少数几个程序需要存取加密口令文件，例如login(1)和passwd(1)，这些程序常常设置 -用户-ID为root。有了阴影口令后，普通口令文件/etc/passwd可由各用户自由读取。

6.4　组文件

UNIX组文件(POSIX.1称其为组数据库)包含了表6-2中所示字段。这些字段包含在 <grp.h> 中所定义的group结构中。

表6-2　/etc/group文件中的字段

说　明	struct group成员	POSIX.1
组名	char *gr_name	•
加密口令	char *gr_passwd	
数字组ID	int gr_gid	•
指向各用户名指针的数组	char **gr_mem	•

> POSIX.1只定义了其中三个字段。另一个字段 gr_passwd则由SVR4和4.3+BSD支持。

字段gr_mem是一个指针数组，其中的指针各指向一个属于该组的用户名。该数组以 null 结尾。

可以用下列两个由POSIX.1定义的函数来查看组名或数值组ID。

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

struct group *getgrgid(gid_t gid);

struct group *getgrnam(const char *name);
```

两个函数返回：若成功则为指针，若出错则为 NULL

如同对口令文件进行操作的函数一样，这两个函数通常也返回指向一个静态变量的指针，在每次调用时都重写该静态变量。

如果需要搜索整个组文件，则须使用另外几个函数。下列三个函数类似于针对口令文件的三个函数。

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

```
struct group *getgrent(void);
```

返回：若成功则为指针，若出错或到达文件尾端则为 NULL

```
void setgrent(void);

void endgrent(void);
```

这三个函数由SVR4和4.3+BSD提供。POSIX.1并未定义这些函数。

setgrent打开组文件(如若它尚未被打开)并反绕它。getgrent从组文件中读下一个记录，如若该文件尚未打开则先打开它。endgrent关闭组文件。

6.5 添加组ID

在UNIX中，对组的使用已经作了些更改。在 V7中，每个用户任何时候都只属于一个组。当用户登录时，系统就按与口令文件相关的数字组 ID，赋给他实际组ID。可以在任何时候执行newgrp(1)以更改组ID。如果newgrp命令执行成功(关于许可权规则，请参阅手册)，则实际组ID就更改为新的组ID，它将被用于后续的文件存取许可权检查。执行不带任何参数的 newgrp，则可返回到原来的组。

这种组的成员关系一直维持到 1983年左右。此时， 4.2BSD引入了添加组ID的概念。我们不仅可以属于口令记录中组ID所对应的组，也可属于多至16个另外的组。文件存取许可权检查相应被修改为：不仅将进程的有效组 ID与文件的组ID相比较，而且也将所有添加组 ID与文件的组ID进行比较。

添加组ID是POSIX.1的可选特性。常数NGROUPS_MAX(见表2-7)规定了添加组ID的数量，其常用值是16。如果不支持添加组ID，则此常数值为0。

SVR4和4.3+BSD都支持添加组ID。

FIPS 151-1要求支持添加组ID，并要求NGROUPS_MAX至少是8。

使用添加组ID的优点是不必再显式地经常更改组。一个用户常常会参加多个项目，因此也就要同时属于多个组。

为了存取和设置添加组ID提供了下列三个函数：

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

int getgroups(int gidsetsize, gid_t grouplist[]);
```

返回：若成功则为添加的组ID数，若出错则为 - 1

```
int setgroups(int ngroups, const gid_t  grouplist[]);

int initgroups(const char *username, gid_t basegid);
```

两个函数返回：若成功则为0，若出错则为 - 1

getgroups将进程所属用户的各添加组 ID填写到数组*grouplist*中，填写入该数组的添加组

ID数最多为 *gidsetsize*个。实际填写到数组中的添加组 ID数由函数返回。如果系统常数NGROUPS_MAX为0，则返回0，这并不表示出错。

> POSIX.1只说明了getgroups。因为setgroups和initgroups是特权操作，所以POSIX.1没有说明它们。但是，SVR4和4.3+BSD支持所有这三个函数。

作为一种特殊情况，如若 *gidsetsize*为0，则函数只返回添加组ID数，而对数组*grouplist*则不作修改。(这使调用者可以确定*grouplist*数组的长度，以便进行分配。)

*setgroups*可由超级用户调用以便为调用进程设置添加组 ID表。*grouplist*是组ID数组，而*ngroups*说明了数组中的元素数。

通常，只有initgroups函数调用setgroups, initgroups读整个组文件(用函数getgrent, setgrent和endgrent)，然后对*username*确定其组的成员关系。然后，它调用setgroups,以便为该用户初始化添加组ID表。因为initgroups调用setgroups,所以只有超级用户才能调用initgroups。除了在组文件中找到*username*是成员的组，initgroups也在添加组ID表中包括了*basegid*。*basegid*是*username*在口令文件中的组ID。

initgroups只有少数几个程序调用，例如login(1)程序在用户登录时调用该函数。

6.6 其他数据文件

至此已讨论了两个系统数据文件——口令文件和组文件。在日常事务操作中, UNIX系统还使用很多其他文件。例如， BSD网络软件有一个记录各网络服务器所提供的服务的数据文件(/etc/services)，有一个记录协议信息的数据文件(/etc/protocols)，还有一个则是记录网络信息的数据文件(/etc/networks)。幸运的是，对于这些数据文件的界面都与上述对口令文件和组文件的相似。

一般情况下每个数据文件至少有三个函数：

(1) get函数：读下一个记录，如果需要还打开该文件。此种函数通常返回指向一个结构的指针。当已达到文件尾端时返回空指针。大多数 get函数返回指向一个静态存储类结构的指针，如果要保存其内容，则需复制它。

(2) set函数：打开相应数据文件 (如果尚未打开)，然后反绕该文件。如果希望在相应文件起始处开始处理，则调用此函数。

(3) end函数：关闭相应数据文件。正如前述，在结束了对相应数据文件的读、写操作后，总应调用此函数以关闭所有相关文件。

另外，如果数据文件支持某种形式的关键字搜索，则也提供搜索具有指定关键字的记录的例程。例如，对于口令文件提供了两个按关键字进行搜索的程序： getpwnam寻找具有指定用户名的记录；getpwuid寻找具有指定用户ID的记录。

表6-3中列出了一些这样的例程，这些都是 SVR4和4.3+BSD所支持的。在表中列出了针对口令文件和组文件的函数，这些已在上面说明过。表中也列出了一些与网络有关的函数。表中列出的所有数据文件都有get、set和end函数。

> 在SVR4中，表6-3中最后四个数据文件都是符号连接，连接到目录 /etc/inet下的同名文件上。
>
> SVR4和4.3+BSD都有类似于表中所列的附加函数，但是这些附加函数都处理系统管理文件，专用于各个实现。

表6-3 存取系统数据文件的一些例程

说 明	数据文件	头 文 件	结 构	附加的关键字搜索函数
口令	/etc/passwd	<pwd.h>	passwd	getpwnam,getpwuid
组	/etc/group	<grp.h>	group	getgrnam,getgrgid
主机	/etc/hosts	<netdb.h>	hostent	gethostbyname,gethostbyaddr
网络	/etc/networks	<netdb.h>	netent	getnetbyname,getnetbyaddr
协议	/etc/protocols	<netdb.h>	protoent	getprotobyname,getprotobynumber
服务	/etc/services	<netdb.h>	servent	getservbyname, getservbyport

6.7 登录会计

大多数UNIX系统都提供下列两个数据文件：utmp文件，它记录当前登录进系统的各个用户；wtmp文件，它跟踪各个登录和注销事件。

在V7中，包含下列结构的一个二进制记录写入这两个文件中：

```
struct utmp {
  char  ut_line[8];  /* tty line: "ttyh0", "ttyd0", "ttyp0", ... */
  char  ut_name[8];  /* login name */
  long  ut_time;     /* seconds since Epoch */
};
```

登录时，login程序填写这样一个结构，然后将其写入到utmp文件中，同时也将其添写到wtmp文件中。注销时，init进程将utmp文件中相应的记录擦除(每个字节都填以0)，并将一个新记录添写到wtmp文件中。读wtmp文件中的该注销记录，其ut_name字段清除为0。在系统再启动时，以及更改系统时间和日期的前后，都在wtmp文件中添写特殊的记录项。who(1)程序读utmp文件，并以可读格式打印其内容。后来的UNIX版本提供last(1)命令，它读wtmp文件并打印所选择的记录。

大多数UNIX版本仍提供utmp和wtmp文件，但其中的信息量却增加了。V7中20字节的结构在SVR2中已扩充为36字节，而在SVR4中，utmp结构已扩充为350字节。

> SVR4中这些记录的详细格式请参见手册页utmp(4)和utmpx(4)。SVR4中这两个文件都在目录/var/adm中。SVR4提供了很多函数(见getut(3)和getutx(3))读或写这两个文件。
>
> 4.3+BSD中登录记录的格式请参见手册页utmp(5)。这两个文件的路径名是/var/run/utmp和/var/log/wtmp。

6.8 系统标识

POSIX.1定义了uname函数，它返回与主机和操作系统有关的信息。

```
#include <sys/utsname.h>

int uname(struct utsname *name);
```

返回：若成功则为非负值，若出错则为-1

通过该函数的参数向其传递一个utsname结构的地址，然后该函数填写此结构。POSIX.1只定义了该结构中至少需提供的字段(它们都是字符数组)，而每个数组的长度则由实现确定。某些实现在该结构中提供了另外一些字段。在历史上，系统V为每个数组分配9个字节，其中有1个字节用于字符串结束符(null字符)。

```
struct utsname {
  char sysname[9];   /* name of the operating system */
  char nodename[9];  /* name of this node */
  char release[9];   /* current release of operating system */
  char version[9];   /* current version of this release */
  char machine[9];   /* name of hardware type */
};
```

utsname结构中的信息通常可用uname(1)命令打印。

> POSIX.1警告nodename元素可能并不适用于在通信网络上引用主机。此函数来自于系统V，早期nodename元素适用于在UUCP网络上引用主机。
>
> 也应理解在此结构中并没有给出有关 POSIX.1版本的信息。这应当使用 2.5.2节中所说明的_POSIX_VERSION以获得该信息。
>
> 最后，此函数给出了一种存取该结构中信息的方法，至于如何初始化这些信息，POSIX.1没有作任何说明。大多数系统V版本在构造内核时通过编译将这些信息存放在内核中。

伯克利类的版本提供gethostname函数，它只返回主机名，该名字通常就是TCP/IP网络上主机的名字。

```
#include <unistd.h>

int gethostname(char *name, int namelen);
```

返回：若成功则为0，若出错则为-1

通过*name*返回的字符串以null结尾(除非没有提供足够的空间)。<sys/param.h>中的常数MAXHOSTNAMELEN规定了此名字的最大长度(通常是64字节)。如果宿主机联接到TCP/IP网络中，则此主机名通常是该主机的完整域名。

hostname(1)命令可用来存取和设置主机名。(超级用户用一个类似的函数sethostname来设置主机名。)主机名通常在系统自举时设置，它由/etc/rc取自一个启动文件。

> 虽然此函数是伯克利所特有的，但是，SVR4作为BSD兼容性软件包的一部分提供gethostname和sethostname函数，以及hostname命令。SVR4也将MAXHOSTNAMELEN扩充为256字节。

6.9 时间和日期例程

由UNIX内核提供的基本时间服务是国际标准时间公元1970年1月1日00:00:00以来经过的秒数。1.10节中曾提及这种秒数是以数据类型time_t表示的。我们称它们为日历时间。日历时间

包括时间和日期。UNIX在这方面与其他操作系统的区别是：(a)以国际标准时间而非本地时间计时；(b)可自动进行转换，例如变换到夏日制；(c)将时间和日期作为一个量值保存。time函数返回当前时间和日期。

```
#include <time.h>
time_t time(time_t *calptr);
```

返回：若成功则为时间值，若出错则为-1

时间值作为函数值返回。如果参数非null，则时间值也存放在由*calptr*指向的单元内。

> 在很多伯克利类的系统中，time(3)只是一个函数，它调用gettimeofday(2)系统调用。
>
> 在SVR4中调用stime(2)函数，在伯克利类的系统中调用settimeofday(2)对内核中的当前时间设置初始值。
>
> 与time和stime函数相比，BSD的gettimeofday和settimeofday提供了更高的分辨率(微秒级)。这对某些应用很重要。

一旦取得这种以秒计的很大的时间值后，通常要调用另一个时间函数将其变换为人们可读的时间和日期。图6-1说明了各种时间函数之间的关系。(图中以虚线表示的四个函数localtime、mktime、ctime和strftime都受到环境变量TZ的影响。我们将在本节的最后部分对其进行说明。)

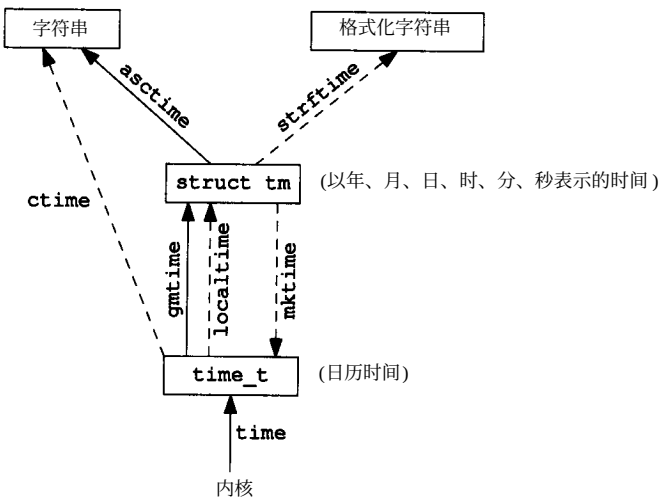

图6-1 各个时间函数之间的关系

两个函数localtime和gmtime将日历时间变换成以年、月、日、时、分、秒、周日表示的时间，并将这些存放在一个tm结构中。

```
struct tm {  /* a broken-down time */
  int  tm_sec;   /* seconds after the minute: [0, 61] */
  int  tm_min;   /* minutes after the hour: [0, 59] */
  int  tm_hour;  /* hours after midnight: [0, 23] */
  int  tm_mday;  /* day of the month: [1, 31] */
  int  tm_mon;   /* month of the year: [0, 11] */
```

```
  int   tm_year;  /* years since 1900 */
  int   tm_wday;  /* days since Sunday: [0, 6] */
  int   tm_yday;  /* days since January 1: [0, 365] */
  int   tm_isdst; /* daylight saving time flag: <0, 0, >0 */
};
```

秒可以超过59的理由是可以表示润秒。注意，除了月日字段，其他字段的值都以 0开始。如果夏时制生效，则夏时制标志值为正；如果已非夏时制时间则为 0；如果此信息不可用，则为负。

```
#include <time.h>

struct tm *gmtime(const time_t *calptr);

struct tm *localtime(const time_t *calptr);
```

两个函数返回：指向tm结构的指针

localtime和gmtime之间的区别是:localtime将日历时间变换成本地时间 (考虑到本地时区和夏时制标志)，而gmtime则将日历时间变换成国际标准时间的年、月、日、时、分、秒、周日。

函数mktime以本地时间的年、月、日等作为参数，将其变换成 time_t值。

```
#include <time.h>

time_t mktime(struct tm *tmptr);
```

返回：若成功则为日历时间，若出错则为- 1

asctime和ctime函数产生形式的26字节字符串，这与date(1)命令的系统默认输出形式类似：

```
Tue Jan 14 17:49:03 1992\n\0
```

```
#include <time.h>

char *asctime(const struct tm *tmptr);

char *ctime(const time_t *calptr);
```

两个函数返回：指向null结尾的字符串

asctime的参数是指向年、月、日等字符串的指针，而 ctime的参数则是指向日历时间的指针。

最后一个时间函数是strftime，它是非常复杂的printf类的时间值函数。

```
#include <time.h>

size_t strftime(char *buf, size_t maxsize, const char *  format,
                const struct tm *tmptr);
```

返回：若有空间，则存入数组的字符数，否则为 0

最后一个参数是要格式化的时间值，由一个指向一个年、月、日、时、分、秒、周日时间值的指针说明。格式化结果存放在一个长度为 *maxsize*个字符的*buf*数组中，如果*buf*长度足以存放格式化结果及一个null终止符，则该函数返回在 *buf*中存放的字符数 (不包括null终止符)，否则该

函数返回0。

*format*参数控制时间值的格式。如同printf函数一样，变换说明的形式是百分号之后跟一个特定字符。*format*中的其他字符则按原样输出。两个连续的百分号在输出中产生一个百分号。与printf函数的不同之处是，每个变换说明产生一个定长输出字符串，在*format*字符串中没有字段宽度修饰符。表6-4中列出了21种ANSI C规定的变换说明。

表6-4 strftime

格 式	说 明	例 子
%a	缩写的周日名	Tue
%A	全周日名	Tuesday
%b	缩写的月名	Jan
%B	月全名	January
%c	日期和时间	Tue Jan 14 19:40:30 1992
%d	月日：[01, 31]	14
%H	小时（每天24小时）：[00, 23]	19
%I	小时（上、下午各12小时)：[01, 12]	07
%j	年日：[001, 366]	014
%m	月：[01, 12]	01
%M	分：[00, 59]	40
%p	AM/PM	PM
%S	秒:[00, 61]	30
%U	星期日周数：[00, 53]	02
%w	周日：[0=星期日， 6]	2
%W	星期一周数：[00, 53]	02
%x	日期	01/14/92
%X	时间	19:40:30
%y	不带公元的年：[00, 991]	92
%Y	带公元的年	1992
%Z	时区名	MST

表中第三列的数据来自于在 SVR4，对应于下列时间和日期，执行 strftime函数所得的结果为：

```
Tue Jan 14 19:40:30 MST 1992
```

表6-4中的大多数格式说明的意义很明显。需要略作解释的是 %U和%W。%U是相应日期在该年中所属周数，包含该年中第一个星期日的周是第一周。 %W也是相应日期在该年中所属的周数，不同的是包含第一个星期一的周为第一周。

> SVR4和4.3+BSD都对strftime的格式字符串提供另一些扩充支持。

我们曾在前面提及，图6-1中以虚线表示的四个函数受到环境变量TZ的影响。这四个函数是：localtime,mktime,ctime和strftime。如果定义了TZ，则这些函数将使用其值以代替系统默认时区。如果TZ定义为空串(亦即TZ=)，则使用国际标准时间。TZ的值常常类似于：TZ=EST5EDT，但是POSIX.1允许更详细的说明。有关TZ的详细情况请参阅POSIX.1标准［IEEE 1990］的8.1.1节，

SVR4 environ(5)手册页［AT&T 1990e］, 或4.3+BSD ctime(3)手册页。

本节所述的所有时间和日期函数都是由 ANSI C标准定义的。在此基础上，POSIX.1只增加了环境变量TZ。

图6-1中七个函数中的五个可以回溯到V7 (或更早)，它们是：time、localtime、gmtime、asctime和ctime。在UNIX时间功能方面近期所作的增加大多是处理非美国时区以及夏时制的更改规则。

6.10　小结

在所有UNIX系统上都使用口令文件和组文件。我们说明了各种读这些文件的函数。也介绍了阴影口令，它可以增加系统的安全性。添加组ID正在得到更多应用，它提供了一个用户同时可以参加多个组的方法。还介绍了大多数系统所提供的存取其他与系统有关的数据文件的类似的函数。最后，说明了ANSI C和POSIX.1提供的与时间和日期有关的一些函数。

习题

6.1　如果系统使用阴影文件，如何取得加密口令？

6.2　假设你有超级用户许可权，并且系统使用了阴影口令，重新考虑上一道习题。

6.3　编程调用uname并输出utsname结构中的所有字段，并与 uname(1)命令的输出结果作比较。

6.4　编程获取当前时间并使用strftime将输出结果转换为类似于date(1)命令的缺省输出。修改环境变量TZ的值，观察输出的结果。