LO03 Notes I

Partie 2

Traduit par 22124765 UTSEUS,SHU

Table des matières

1 Linux 简介

多用户多任务操作系统：多用户操作系统是指允许多个用户同时访问和共享计算机资源的操作系统，每个用户都有独立的工作环境。多任务操作系统是指能够同时运行多个程序或任务的操作系统。

编程环境：标准的编程环境是 C 语言，近年来 C++ 也被广泛使用。

网络环境：网络环境基于 TCP/IP 协议。

1.1 Linux 的哲学

Unix 的哲学是一套确保 Unix 操作系统成功运行的规则。其中最著名的范式是“一**切皆文件(tout est fichier)**”。操作系统的所有功能都可以通过简单的文件来访问：设备、进程、内存等。

开发不再是单打独斗，项目的每一个初步构思都应该提交给社区，以便让更多的人能够以不同的方式参与开发，例如指出需要尽快修复的漏洞。

1.2 Linux 的诞生与发展

1991 年，一位名叫 Linus Torvalds 的芬兰学生决定创建一个新的内核，该内核基于 Minix（一种 UNIX 的衍生系统），以克服其局限性以及 MS-DOS 的限制。这个新的内核于 1991 年 8 月以“Linux”之名首次发布其 0.1 版本，并免费在互联网上分发。

目前，Linux 内核已经更新到 6.11.1 版本（截至 2024 年 9 月 30 日），并且可以自由下载。

1.3 流行的 Linux 发行版介绍

Ubuntu 是一种面向普通用户的商业发行版，由 Canonical 公司免费分发。Canonical 每六个月发布一个稳定版本，这些版本会得到 9 个月的维护支持。

Fedora 是一种由 Red Hat 监督的社区发行版，基于 RPM 软件包管理系统。Fedora 注重软件的新颖性，这意味着其软件会非常频繁地更新。

openSUSE 是一种既适合普通用户也适合专业用途的社区发行版。它是一个独立的发行版，以其配置工具和稳定性而闻名。

1.4 在您的计算机上使用 Linux

— 在硬盘的一个分区上安装 Linux。实现 Windows 和 Linux 的双系统启动。
— 使用可引导的 Linux USB 闪存驱动器。
— 在您的 PC 上通过 Virtualbox 等工具虚拟化 Linux。可以同时使用 Linux、Windows 和 Mac OS。

— Windows Subsystem for Linux (WSL)。
— 在 MacOS 下：可以直接使用 Zsh。

2 Shell

在 Linux 中，Shell 是指命令行解释器(interpréteur de commandes)，它充当用户与操作系统之间的接口。Shell 是命令解释器，这意味着用户输入的每条命令（或从文件中读取的命令）都会先进行语法检查，然后执行。

在 Unix 系统中存在多种 Shell 解释器，例如 Bourne Again Shell（bash）、C Shell、Korn Shell、Z Shell 等。

熟悉其中一种 Shell，就可以更容易地使用其他 Shell。

所有 Shell 都可以在同一 Unix 系统中共存。系统管理员可以在用户定义文件‘/etc/-passwd‘中为每个用户设置初始 Shell。

Shell 具有工作环境(environnement de travail) 和编程语言（langage de programmation）。

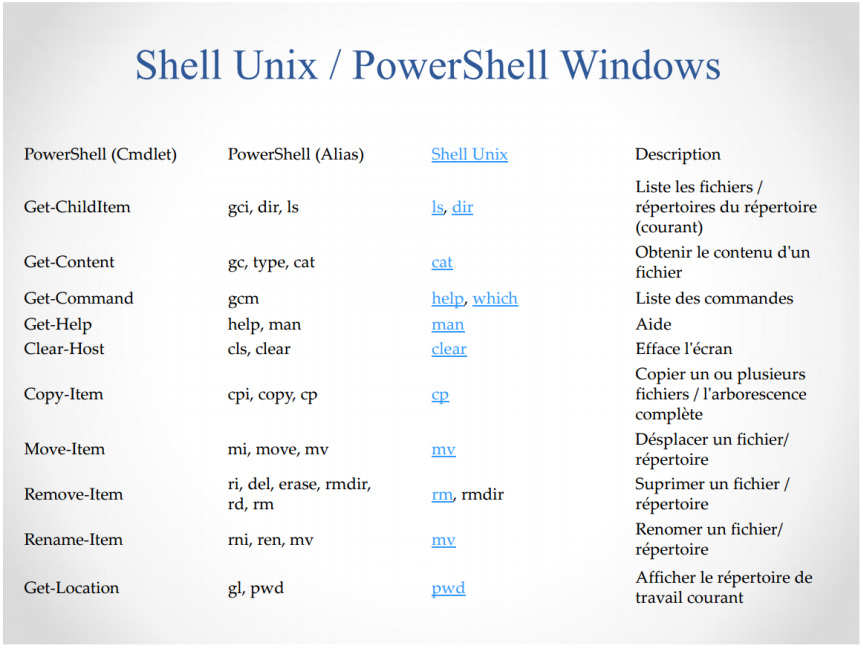


FIGURE 1 – Comparaison

2.1 Shell 的工作环境

Shell 允许以交互模式执行命令(mode interactif)，也可以通过命令文件（脚本）(par l'intermédiaire de fichiers de commandes (scripts)) 执行。在交互模式下，bash 会在屏幕上显示一个调用字符串（也称为提示符或命令提示符），默认情况下，对于系统管

理员（root 用户），该字符串以字符 # 结束，后跟一个空格；对于其他用户，该字符串以字符 $ 结束，后跟一个空格。这个调用字符串可能相对较长。

例如：`retraint@marylin:/home/retraint$`

提示符由以下几部分组成：

1. 用户名（retraint），
2. 用户工作的机器名（marylin），
3. 用户当前目录的绝对路径（/home/retraint）。

提示符表明 Shell 正在等待用户输入命令，并且用户通过按下回车键来确认输入的命令。Bash 随后执行该命令，然后重新显示提示符。

如果我们希望缩短提示符的长度，只需修改 Shell 预定义变量 PS1（Prompt Shell 1）的值即可。

例如：`retraint@marylin:/home/retraint$` 修改 PS1 变量为 `PS1='$ '`。

2.2 Shell 的优点与缺点

2.2.1 优点

1. Shell 是一种解释型语言：错误可以很容易地被定位和处理。对应用程序的功能进行修改也很容易，无需重新编译和链接整个程序。
2. Shell 主要操作字符串：这有助于避免打字错误。
3. 该语言适合快速原型开发应用程序：管道、命令替换和变量有助于通过组合 Unix 环境中的现有命令来构建应用程序。
4. Shell 允许连接用不同语言编写的组件。

2.2.2 缺点

1. 操作系统最初是由开发者为开发者编写的，Shell 使用对初学者来说难以理解的访问语法。
2. 忘记或添加一个空格很容易导致语法错误。
3. Bash 拥有多种语法来实现相同的功能。
4. 某些特殊字符，如括号，在不同上下文中有不同的含义：括号可以引入命令列表、函数定义或强制执行算术表达式的求值顺序。

2.3 命令行的语法

命令的语法如下：

```
nom_commande [options] [arguments]
```

— 这是由一个或多个空白字符分隔的单词序列。在这里，空白字符指的是制表符（水平制表符）或空格字符。
— 单词是一系列非空白字符。
— 选项通常以连字符字符开始（例如：`-a`）或在语法中以两个连续的连字符字符开始（例如：`--version`）。
— 参数指的是命令要执行的对象。

命令行的语法示例：

1. `$ ls -l /home`
2. `$ cal 3 2010`

多个命令的串联：通过“;”（分号）来分隔命令
示例：

```
$ ls -l /home ; cal 3 2010
```

2.4 系统帮助：Man 命令

系统帮助命令 Man 用于提供 Unix/Linux 命令的在线文档，其主要部分包括：

1. **Name**：简要描述文档的对象。
2. **Synopsis**：展示命令的语法。
3. **Description**：提供命令及其各个选项的精确功能描述。
4. **Files**：列出命令将要使用的配置文件或数据文件。
5. **See also**：指向文档中相关的其他部分。
6. **Exit status**：提供执行返回码，这些信息通常用于 Unix 编程。

2.5 其他文档来源

除了使用 Man 命令外，还有其他几种方式可以获取 Unix/Linux 命令的文档，包括：

1. **whatis 命令**：
   — 在「whatis」数据库中搜索完整的单词。
   — 示例：`$ whatis date`
2. **apropos 命令**：
   — 在「whatis」数据库中搜索字符串。与「whatis」命令相比，它不那么严格。
   — 示例：`$ apropos date`

3 文件系统

文件系统是一种在存储介质（如硬盘、软盘、光盘等）上组织和存储树状结构的方式：

— 每个专有操作系统都开发了自己的组织方式。
— 可以在同一个硬盘的分区中共存多个文件系统。
— Linux 拥有自己的文件系统，称为 ext4，但也可以管理其他文件系统。这些文件系统列表可以在`/proc/filesystems` 中找到。

在 Unix 系统中，对于用户来说，一切都是文件。无论物理位置如何，所有文件系统都必须集成到 Linux 系统的唯一逻辑树状结构中。

因此，树状结构可以由位于多个磁盘上的不同分区构建而成。这实现了比 Windows 世界中更深层次的集成和抽象，在 Windows 中，分配给字母 A :、C :、D :等的分区和驱动器仍然是独立的实体。

— 挂载过程（mount 命令）是将树状结构的部分与磁盘的物理分区对应起来的方法。它还允许将任何外部系统（磁盘、CD-ROM、网络共享等）分配给树状结构中为此创建的目录。
— 之后，只需移动到该目录，即所谓的挂载点，实际上是一个“挂钩”目录，就可以访问其文件。

3.1 文件类型

文件系统包含多种类型的文件，每种文件都有其特定的用途和特性：

— **普通文件（fichiers ordinaires）**：
  — **文本文件（texte）**：包括信件、程序源代码、脚本、配置文件等。
  — **可执行文件（exécutables）**：二进制代码形式的程序。
— **目录文件（fichiers répertoires）**：这些是容器文件，包含对其他文件的引用，是树状结构的真正骨架，允许按类别组织文件。
— **特殊文件（fichiers spéciaux）**：位于`/dev` 目录中，是系统为访问设备准备的访问点。挂载过程将这些特殊文件映射到它们的“挂载点”目录。例如，文件`/dev/hda` 允许访问和加载第一个 IDE 硬盘。
— **符号链接文件（fichiers liens symboliques）**：这些文件仅包含对另一个文件的引用（指针）。这允许在不复制文件到磁盘的情况下，使用同一文件的多个名称。

3.2 一些目录

Linux 文件系统中，各个目录有其特定的用途：

— **`/bin`**：存放对用户至关重要的 **binaires**（可执行文件）。
— **`/dev`**：存放所有 **périphériques**（设备）。

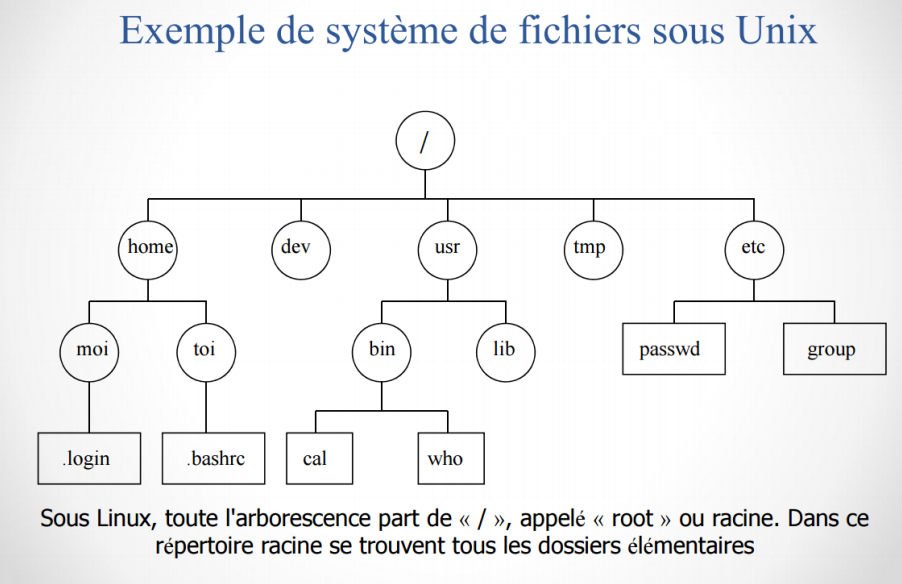


FIGURE 2 – Exemple de système de fichiers sous Unix

— **`/home`**：存放用户的 **répertoires personnels**（个人目录）。
— **`/mnt`**：用于 **points de montage temporaires**（临时挂载点）的目录。
— **`/proc`**：提供对内核参数的直接访问。
— **`/root`**：超级用户（**root**）的个人目录。
— **`/var`**：存放经常变化的文件，如日志文件、邮件和打印队列。

3.3 路径（Les chemins）

在文件系统中，有两种主要的路径描述方式：
— **绝对路径（Le chemin d'accès absolu）**：描述从树状结构的**根目录（racine）**到文件的完整路线。
— **相对路径（Le chemin d'accès relatif）**：描述从当前工作目录（répertoire de travail courant）到目标文件的路线。如果目标文件位于当前目录下，则用户需要指定剩余的路径。
— 符号 `.` 表示**当前目录（répertoire courant）**，符号 `..` 表示其**父目录（père）**。
— 注意：文件名前的符号 `.` 意味着该文件是**隐藏的（caché）**。

3.3.1 路径示例（Exemple de chemins）

— 例如，**`/usr/bin/cal`** 是文件 **cal** 的**绝对路径（chemin absolu）**。
— 如果当前工作目录是 `/home/moi`：
  — `./.login` 等同于**绝对路径（chemin absolu）** `/home/moi/.login`
  — `../toi/.cshrc` 等同于**绝对路径（chemin absolu）** `/home/toi/.cshrc`

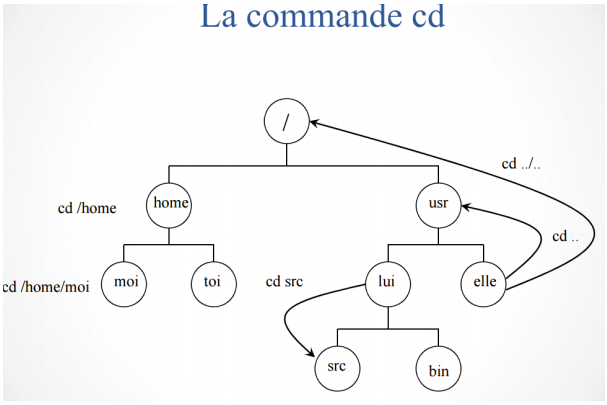


FIGURE 3 – La commande cd

— 注意：符号 ~ 表示用户的**登录目录（répertoire de connexion）**。例如，如果用户是 **moi**，那么 **`~/.login`** 对应于**相对路径（chemin relatif）** `/home/moi/.login`

3.4 文件的属性

命令：`$ ls -l fichier`

输出解释：

— 第一列（type）：表示文件类型。
  — `-`：表示普通文件（fichier）。
  — `d`：表示目录（répertoire）。
  — `l`：表示符号链接（lien symbolique）。
— 第二列（Droits，权限）：表示文件的访问权限。
— 第三列（Nombre de liens，链接数）：表示有多少个硬链接指向该文件。
— 第四列（propriétaire，所有者）：文件的所有者用户名。
— 第五列（groupe，组）：文件所属的用户组。
— 第六列（taille en octets，字节大小）：文件的大小，以字节为单位。
— 第七列（date et heure de dernière modification，最后修改日期和时间）：文件最后被修改的日期和时间。
— 第八列（nom，名称）：文件的名称。

示例输出：

```
-rwxr-x--- 1 moi etu 300 Jun 02 12:00 fichier
```

— `-rwxr-x---`：文件所有者有读、写、执行权限；所属组有读、执行权限；其他用户没有任何权限。
— `1`：有 1 个硬链接指向该文件。
— `moi`：文件所有者的用户名是 `moi`。

— `etu`：文件所属的用户组是 `etu`。
— `300`：文件大小为 300 字节。
— `Jun 02 12:00`：文件最后修改时间是 6 月 2 日 12 点。
— `fichier`：文件名为 `fichier`。

3.5 常用文件命令

在 Unix/Linux 系统中，有许多用于管理文件的命令。以下是一些最常用的命令：

1. `cp`：用于**复制文件**（**copie d'un fichier**）。
2. `rm`：用于**删除文件**（**détruit un fichier**）。
3. `mv`：用于**重命名文件**（**change le nom d'un fichier**）或**移动文件**（**déplace un fichier**）。
4. `cat`, `more`：用于**显示文件内容**（**affiche le contenu d'un fichier**）。
5. `file`：用于**显示文件类型**（**affiche le type du fichier**）。
6. `cmp`, `comm`, `diff`：用于**比较文件**（**compare des fichiers**）。

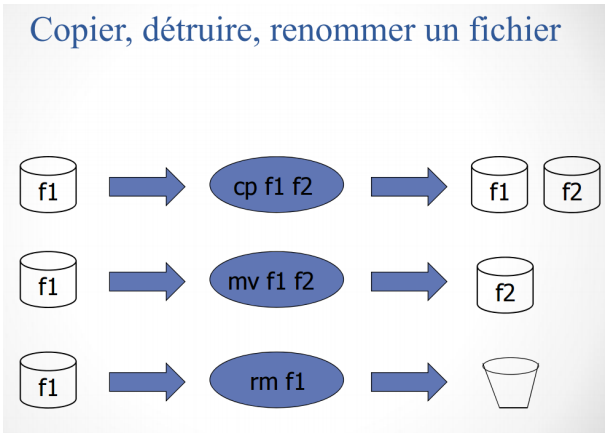


FIGURE 4 – Copier, détruire, renommer un fichier

3.6 常用目录命令

在 Unix/Linux 系统中，管理目录也需要一些基本的命令。以下是一些最常用的 **répertoires**（目录）相关命令：

1. `pwd`：显示 **répertoire courant**（**当前目录**）。
2. `mkdir`：创建一个 **répertoire**（**新目录**）。
3. `rmdir`：删除一个 **répertoire**（**空目录**）。
4. `du`：显示目录或文件的 **taille**（**磁盘使用情况**）。
5. `find`：在目录树中 **recherche**（**搜索文件**）。

3.7 了解文件权限（Connnaître les droits）

在 Unix/Linux 系统中，使用命令 `ls -l` 可以查看文件的详细信息，包括文件权限。以下是命令输出的详细解释：

— `$ ls -l fichier`：列出名为 `fichier` 的文件的详细信息。
— **Droits du propriétaire**（文件所有者的权限）：表示文件所有者（**propriétaire**）的访问权限。
— **Droits du groupe**（组的权限）：表示文件所属组（**groupe**）的访问权限。
— **Droits pour les autres**（其他用户的权限）：表示其他用户（**autres**）的访问权限。
— 第四列：文件的链接数。
— 第五列：文件所有者的用户名。
— 第六列：文件所属的用户组名。
— 第七列：文件的大小，以字节为单位。
— 第八列：文件最后修改的日期和时间。
— 第九列：文件的名称。

权限由三组字符表示，每组三个字符，分别对应文件所有者、组和其他用户的读（r）、写（w）、执行（x）权限。如果某组权限缺失，则用 '-'表示没有相应的权限。

```
-rw-r--r-- 1 moi etu 300 Jun 02 12:00 fichier
```

例如，输出 `-rw-r--r--` 表示：
— 文件所有者有读（r）和写（w）权限；
— 所属组和其他用户只有读（r）权限。

3.8 文件和目录的权限

文件和目录的权限可以通过两种主要的表示方法来设置和理解：符号表示法（Notation symbolique）和八进制表示法（Notation octale）。

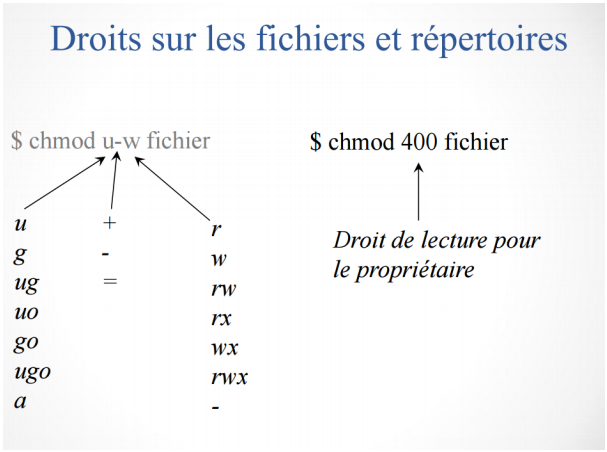
3.8.1 符号表示法

— **谁（Qui）**：
  — U = **user**（**所有者，propriétaire**）
  — G = **group**（**组，le groupe**）
  — O = **others**（**其他用户，les autres**）
— **权限（Permission）**：
  — R = **read**（**读取，consulter le contenu**）

— W = **write**（**写入，modifier et détruire**）
— X = **execute**（**执行，éxecuter un script ou un programme**）

3.8.2 八进制表示法

— 在八进制表示法中，**所有者（u）**、**组（g）**和**其他用户（o）**的权限由相应的八进制值表示，规则如下：
  — 0 = **无权限，aucun droit**
  — 1 = **执行权限，exécution (x)**
  — 2 = **写权限，écriture (w)**
  — 4 = **读权限，lecture (r)**
— 例如：`$ chmod 400 fichier` 设置文件的权限，使得所有者有读权限。



3.8.3 文件和目录的权限：示例

以下是一些关于如何设置文件和目录权限的示例：

设置文件权限

— 命令：`$ chmod 751 exemple.c`
— 输出：`-rwxr-x--x 1 retraint users 1829 Mar 18 2002 exemple.c`
— 权限解释：
  — `rwx` = 4+2+1 = 7 (**utilisateur**，用户)
  — `r-x` = 4+1 = 5 (**groupe**，组)
  — `--x` = 1 (**autres**，其他)

设置目录权限

— 命令：`$ chmod 700 ~`
— 说明：此命令将用户的 **répertoire de connexion**（登录目录）转变为 **répertoire privé**（私有目录）。

3.9 物理链接和符号链接（Liens physiques et symboliques）

3.9.1 链接的创建和识别

— 命令 `ln` 用于创建指向文件的链接（类似于 Windows 的快捷方式）。
— 运行命令 `ls -l` 时，链接将显示出来。
— 如果不使用 `-s` 选项，`ln` 将创建一个物理链接（**hard link**）而不是符号链接（**symbolic link**）。
— 符号链接（**liens symboliques**）允许不复制文件所占用的磁盘空间。

物理链接和符号链接示例

— 创建符号链接：`$ ln -s fich fich2`
— 查看链接：`$ ls -l`

```
-rw-r--r-- 1 retraint users 1829 Mar 5 14:25 fich
lrwxrwxrwx 1 retraint users 4 Mar 9 14:03 fich2 -> fich
```

— 最后一行显示了 `fich` 和 `fich2` 之间的符号链接。
— 删除原文件：`$ rm fich`
— 尝试访问符号链接：`$ cat fich2`

```
Cat: fich2: No such file or directory
```

— 删除链接：使用命令 `rm`。只有链接被删除，而不是链接指向的文件。
— 创建符号链接：`$ ln -s fichier_ou_répertoire lien`

3.10 输入输出重定向（Les redirections des entrées sorties）

在 Unix/Linux 系统中，存在一些方法可以重定向命令的输入和输出：

3.10.1 标准重定向

— 使用符号 > 将输出重定向到指定的文件（如果文件存在则替换文件）。
— 使用符号 >> 将输出追加到指定的文件末尾。
— 使用符号 < 将输入从指定的文件中读取。
— 使用 << 可以指定一个标记，从该标记开始到结束的所有内容作为命令的输入。

3.10.2 重定向示例 1

— 重定向标准输入（键盘）：

```
1 $ commande [options] [arguments] < fichier
```

输入命令的数据来自指定的文件。

— 重定向标准输出（屏幕）：

```
$ commande [options] [arguments] > fichier
```

命令产生的结果将被写入指定的文件。

3.10.3 重定向示例 2

— 将文件 fich1 的内容重定向到文件 fich2：

```
$ cat fich1 > fich2
```

— 使用 << 将多行文本重定向到文件 sortie：

```
$ cat <<FIN > sortie
> Bonjour,
> Comment ça va ?
> Au revoir.
> FIN
```

— 查看文件 sortie 的内容：

```
$ cat sortie
Bonjour,
Comment ça va ?
Au revoir
```

3.10.4 错误重定向（Les redirections des erreurs）

在 Unix/Linux 系统中，命令将具有结果价值的文本行发送到标准输出（stdout），而将表示语法或执行错误的信息发送到标准错误（stderr）。

标准错误重定向

— 要重定向命令产生的标准错误输出，需要在命令行中指定 2>fichier_de_redirection。

— 示例：$ ls -l 2>/dev/null

— 为了忽略可能干扰结果可读性的错误信息，可以将它们重定向到设备 /dev/null，这样错误信息就会被丢弃。

4 管道（Les tubes）和别名（Les alias）

4.1 管道（Les tubes）

管道，由符号 | 表示，是一种通信机制，允许在命令之间交换数据。

— 示例：commande1 | commande2 | commande3

— `$ ls -l | more`：逐页显示 `ls` 命令的结果。
— `$ ls -l | lp`：打印 `ls -l` 命令的结果。

4.2 别名（Les alias）

使用别名是一种方便的方法，可以为带有内置选项的命令定义同义词。

1. 创建别名：`$ alias dir='ls -lba'`
2. 使用别名：`$ dir`
3. 显示别名列表：`$ alias`
4. 删除别名：`$ unalias`

5 文件系统（续 1）

5.1 文件内容的查看(Visualisation du contenu des fichiers)

在 Unix/Linux 系统中，可以使用多种命令来查看文件的内容：

1. `head`：用于查看文件开头的一定数量的行。
2. `tail`：用于处理文件的末尾部分。
3. `tail -f`：用于实时显示文件内容，当文件有新行添加时会立即显示。
4. `wc`：用于计算文件的字符数、单词数和行数（`-c`：字符，`-w`：单词，`-l`：行数）。如果不指定选项，`wc` 命令将显示这三种信息。
5. `file`：提供文件内容的信息。
6. `nl`：给文件的行编号。

5.1.1 文件内容查看示例

— 查看文件的前 3 行：

```
$ head -3 fich
ligne 1
ligne 2
ligne 3
```

显示文件的前 3 行。

— 查看文件的最后 2 行：

```
$ tail -2 fich
ligne 14
ligne 15
```

显示文件的最后 2 行。

— 实时显示文件的最新更新：

```
$ tail -2 fich | head -1
ligne 14
```

显示倒数第二行。

— 计算文件的字符数、单词数和行数：

```
$ wc fich
350 126 30 fich
```

— 查看文件类型：

```
$ file exemple.c
exemple.c: commands text
```

— 计算目录中的文件数量：

```
$ ls | wc -l
5
```

— 从文件的第三行开始显示到文件末尾：`$ tail +3 fich`

5.2 文件的高级命令（Commandes avancées sur les fichiers）

5.2.1 文件中元素的提取

— `cut` 命令用于从文件中提取列（colonnes）。

— 命令格式：

```
$ cut -f\textit{liste} [-\textbf{d}délimiteur] [-s] [fichie
$ cut -c\textit{liste} [fichier]
```

— 当行由长度可变的区域组成，这些区域由制表符分隔时，选择与 `-f` 选项关联的单位。

— 如果文件的行由长度固定的区域组成，则使用与 `-c` 选项关联的单位。

5.2.2 高级文件命令示例

— 创建一个示例文件 `ville`：

```
$ cat ville
10000,Troyes,Aube
51000,Reims,Marne
```

— 使用 cut 提取第一列和第三列：

```
$ cut -f1,3 -d ',' ville
10000,Aube
51000,Marne
```

— 使用 cut 提取前 5 个字符：

```
$ cut -c1-5 ville
10000
51000
```

6 脚本入门（Introduction aux scripts）

6.1 脚本概述

— 术语 **script**（脚本）指的是由 Shell 执行的文件。它包含 Unix 系统的外部命令、内部命令和注释。
— 当启动脚本执行时，Shell 会按顺序读取并执行文件中的各行命令。
— 如果 Shell 在某一行检测到错误，它将停止读取脚本并显示导致错误的命令的行号。

6.2 用户环境脚本

用户的环境（终端类型、提示符选择、字符删除键的定义等）是通过两个脚本构建的：

— 脚本 /etc/profile，对所有用户都通用。
— 脚本 ~/.bash_profile，位于用户的 **répertoire de connexion**（登录目录）中。

6.3 注释（Les commentaires）

在命令行或脚本中，注释用于提供额外的信息而不会影响命令的执行。

6.3.1 命令行中的注释

— 在一行命令中，字符 # 表示注释的开始，该注释持续到行尾。

6.3.2 脚本中的伪注释

— 伪注释 `#!` 被放置在脚本的开头，用于明确指定执行脚本的 Shell。
— 示例：`#!/bin/ksh` 表示该脚本由 Korn Shell 执行。
— 示例：`# Ce script affiche la liste des processus` 表示该脚本显示进程列表。

6.4 脚本的执行（Exécution d'un script）

6.4.1 执行脚本的方法

— 使用 `bash` 命令执行脚本：`$ bash lescript`
— 通过重定向执行脚本：`$ bash <lescript`
— 使脚本可执行并执行：`$ chmod +x lescript; ./lescript`

6.4.2 脚本中调试模式的开启和关闭

— 在脚本中包含命令 `set -x` 来激活或关闭调试模式。

6.4.3 带命令跟踪的脚本执行

— 使用 `bash -xv lescript` 执行脚本并显示命令的跟踪信息。

6.5 参数（Les paramètres）

在 Shell 术语中，参数（paramètre）指的是可以包含值的任何实体。Shell 区分三类参数：

— 变量（variables），通过名称标识，例如：`a`，`PATH`
— 位置参数（paramètres de position），通过数字标识，例如：`0`，`1`，`12`
— 特殊参数（paramètres spéciaux），通过特殊字符标识，例如：`#`，`?`，`$`

赋值的方式根据参数的类别而异。根据类别，赋值可以由用户、bash 或系统执行。要获取参数的值，无论其类别如何，总是在引用前放置字符 `$`。

示例例如：`$ echo $PATH` 显示变量 `PATH` 的值。

6.6 变量（Les variables）

用户可以通过以下方式给变量赋值：

— 使用赋值操作符 =
— 使用 Shell 的内部命令 `read`

6.6.1 直接赋值

```
variable=valeur
```

获取变量的值：

```
$variable
```

例如：

```
$ x=bon $ y=jour
$ echo $x
bon
$ x=$x$y
echo $x
bonjour
```

6.6.2 通过读取赋值

```
read [var1 var2 ….]
```

例如：

```
$ read x y
bon jour
echo $x$y
bonjour
```

当内部命令 `read` 没有参数时，读取的行将存储在 Shell 的预定义变量 `REPLY` 中。例如：

```
$ read
bonjour tout le monde
$ echo $REPLY
bonjour tout le monde
```

6.6.3 带提示的读取赋值

`read` 的 `-p` 选项在读取之前显示提示信息；使用的语法是：`read -p chaîne_d_appel [var …]` 例如：

```
$ read -p "Entrez votre prenom : " prenom
Entrez votre prenom : Eric
$ echo $prenom
Eric
```

6.6.4 变量管理

— 列出所有变量：使用命令 `set`
— 删除变量：使用命令 `unset variable`

6.6.5 变量声明（La déclaration des variables）

— 要定义一个值不可修改的变量（在其他编程语言中称为常量），我们使用语法：
`declare –r` **nom**`=valeur [` **nom**`=valeur …]`
例如：`$ declare -r mess=bonjour`
— 尝试修改常量的值将导致错误。
— 声明一个整数：`declare –i nombre=1`
— 声明一个数组：`declare –a tab`
— 声明一个函数：`declare –f fonction`

6.6.6 环境变量（Les variables d'environnement）

— 设置局部变量：`$ var_locale=alpha`
— 设置环境变量：`$ var_environ=gamma`
— 导出环境变量：`$ export var_environ`
— 命令 `export` 允许 Shell 将变量放入环境。
— 语法：`$ export variable[=valeur]`
— 命令 `env` 用于列出所有已导出的环境变量。

6.7 定义字段分隔符：IFS（Définition de séparateur de champs : IFS）

IFS（Internal Field Separator）是一个 Shell 变量，用于指定在读取输入时如何将一行文本分解为多个字段。

— 命令 `read` 读取一整行，并使用变量 **IFS** 来将该行分解为单词，然后将其赋值给传递给它的各个读取变量。
— 默认情况下，变量 **IFS** 初始化为字符 `<Espace>`（空格）、`<tabulation>`（制表符）和 `<Saut de ligne>`（换行符）。
— 修改变量 **IFS**：
  — `IFS=:` 设置字段分隔符。可以指定新的分隔符，例如 `IFS=:` 表示使用冒号作为字段分隔符。

6.8 参数(续)（Les paramètres）

— 脚本类似于一条命令。可以在命令行上提供执行所需的参数。
— 参数被赋值给变量 `0, 1, ..., 9, 10, 11, ...`

— 变量 * 表示所有参数，作为一个单独的参数。
— 变量 @ 表示所有参数，每个参数作为一个独立的参数。
— 变量 # 表示传递给脚本的参数数量。
— 命令 shift 用于移动参数：参数 1 的值被参数 2 的值替换，参数 3 的值被参数 4 的值替换，依此类推。
— shift [n]：移动 n 个参数。
— 内部命令 set 用于替换所有参数。
示例：

```
$ set un deux trois ; echo $*
un deux trois
```

6.9 消除歧义（Suppression des ambigüités）

为了避免在解释参数引用时产生歧义，我们使用语法 ${paramètre}。
— 示例：

```
$ set un deux trois quatre cinq six sept huit neuf dix
    onze douze
$ echo $11
un1
$ echo ${11} => pour obtenir la valeur du onzième
   paramètre de position
onze
```

— 当参数编号可能与 Shell 变量名冲突时，使用大括号可以明确指定引用的是位置参数。
— $11 可能会被解释为变量 11 的值，而不是位置参数 11 的值。
— 使用 $11 可以正确获取第十一个位置参数的值 onze。

6.10 命令替换（Substitution de commandes）

命令替换是指用一对圆括号 () 包围的命令 cmd，前面加上字符 $，由 Shell 执行后，字符串 $(cmd) 被替换为命令 cmd 的标准输出结果。这些结果可以赋值给变量或初始化位置参数。

6.10.1 示例

— 执行 pwd 命令并获取当前目录：

```
$ pwd
```

```
        /home/retraint
        $ repert=$(pwd) => la chaîne /home/retraint remplace
           la chaîne $(pwd)
        $ echo mon repertoire est $repert
        mon repertoire est /home/retraint
```

6.10.2 使用 set 命令时的注意事项

— 当与 set 命令一起使用命令替换时，可能会发生错误：

```
        $ set $( ls -l .bashrc)
        bash: set: -w: invalid option
        $ ls -l .bashrc
        -rw-r--r-- 1 retraint users 1342 nov 30 2009 .bashrc
```

— 第一个由替换产生的词以连字符 - 开头，内部命令 set 将其解释为选项，导致错误。为解决此问题，需要对连字符进行转义。

— 示例：$ set -- $(ls -l .bashrc)

```
        $ echo $1
        -rw-r--r—
```

6.10.3 嵌套的命令替换

命令替换可以嵌套（**imbriquées.**）使用。

— 示例：

```
        $ set $( ls $(pwd)/retraint )
        $ echo $# => nombre d'entrées du répertoire retraint
        3
```

— 命令替换还可以用来捕获 Shell 文件的输出结果。

— 示例：$ lescript

```
        38
        $ resultat=$( lescript )
```

变量 resultat 包含字符串 38。

6.11 返回码（Le code retour）

— 在 Unix 中，当一条命令或一个脚本执行结束时，它会发出一个数值形式的返回码（code de retour numérique）。

— 预定义的特殊变量 `$?` 会自动初始化为最后执行命令的返回码。

6.11.1 示例

```
$ who | grep thomas
$ echo $?
0 # 如果返回码为0，表示thomas已连接
```

在这个示例中，使用 `who` 命令列出当前登录的用户，并通过管道（|）将输出传递给 `grep` 命令以搜索名为 thomas 的用户。执行完毕后，使用 `echo $?` 打印出上一个命令的返回码。如果返回码为 `0`，则表示 thomas 已连接。

6.12 控制指令（Les instructions de contrôle）

和所有编程语言一样，Shell 语言也拥有控制指令，例如：

1. **条件选择**：`if` 指令。
2. **循环结构**：`for`、`while`、`until` 指令。
3. **多路选择**：`case` 指令。

6.13 条件语句：if（if : l'alternative）

Shell 脚本中的 **if** 语句用于基于条件执行不同的代码块。其基本语法如下：

```
if condition
then
    bloc de commandes
elif condition
then
    bloc de commandes
else
    bloc de commandes
fi
```

— `if condition`：指定要评估的条件。
— `then`：如果条件为真，则执行紧随其后的命令块。
— `elif condition`：指定另一个条件，如果初始条件不为真，则评估此条件。
— `else`：如果所有 `if` 和 `elif` 条件都不满足，则执行 `else` 块中的命令（可选）。
— `fi`：标志着 `if` 语句的结束。

6.14 测试命令（La commande test）

La commande test 用于评估表达式并返回退出状态。基本语法如下：

— `if test expression ; then ... fi`

— `if [ expression ] ; then ... fi`

test 命令不在标准输出上显示结果，如果表达式为真则返回返回码 `0`，如果表达式为假则返回 `1`。

6.14.1 文件属性测试（Test d'un attribut de fichier）

— `-f fichier`：如果文件存在且为普通文件，则为真（le fichier existe et qu'il est ordinaire）。

— `-e fichier`：如果文件存在，则为真（le fichier existe）。

— `-r fichier`：如果文件存在且可读，则为真（le fichier existe et qu'il est accessible en lecture）。

— `-w fichier`：如果文件存在且可写，则为真（vrai si le fichier existe et qu'il est accessible en écriture）。

6.14.2 字符串测试（Test sur des chaînes de caractères）

— `-z chaîne`：如果字符串长度为 `0`，则为真。

— `-n chaîne`：如果字符串长度不为 `0`，则为真。

— `chaîne1 = chaîne2`：如果字符串 `chaîne1` 与 `chaîne2` 相同（identique），则为真。

— `chaîne1 != chaîne2`：如果字符串 `chaîne1` 与 `chaîne2` 不同（différente），则为真。

6.14.3 数字测试（Test sur les nombres）

— `arg1 -eq arg2`：如果 `arg1` 等于(égal) `arg2`，则为真。

— `arg1 -ne arg2`：如果 `arg1` 不等于（différent）`arg2`，则为真。

— `arg1 -lt arg2`：如果 `arg1` 严格小于(strictement inférieur) `arg2`，则为真。

— `arg1 -gt arg2`：如果 `arg1` 严格大于(strictement supérieur) `arg2`，则为真。

— `arg1 -le arg2`：如果 `arg1` 小于或等于(inférieur ou égal) `arg2`，则为真。

— `arg1 -ge arg2`：如果 `arg1` 大于或等于(supérieur ou égal) `arg2`，则为真。

6.15 条件语句示例（If test -f $1）

以下是一个 Shell 脚本示例，用于检查传递给脚本的文件是否存在，并根据结果执行不同的操作：

```
    if test -f $1
    then
        file $1
    else
        echo 'le␣fichier␣$1␣n'existe pas'
␣␣␣␣fi
␣␣␣␣
```

— 如果传递给脚本的文件名对应的文件存在，则显示其内容类型。
— 可以将关键字 **if** 和 **then** 放在同一条线上，用分号分隔：
— `if test -f $1 ; then file $1`

6.16 逻辑运算符（L'alternative && et ||）

在 Shell 脚本中，可以使用逻辑操作符 `&&` 和 `||` 来根据前一个命令的返回码决定是否执行后续命令。

6.16.1 逻辑与（&&）

— `commande1 && commande2` 相当于：如果 `commande1` 成功（返回码为 0），则执行 `commande2`。
— 这与 `if commande1 ; then commande2 ; fi` 相同。

6.16.2 逻辑或（||）

— `commande1 ||commande2` 相当于：如果 `commande1` 失败（返回码为 1），则执行 `commande2`。
— 这与 `if commande1 ; then ; else commande2 ; fi` 相同。

6.16.3 逻辑运算符（Opérateurs logiques）

— `!expression_logique`：如果 `expression_logique` 为假（false），则为真（vrai）。
— `expression_logique1 -a expression_logique2`：如果两个表达式都为真，则为真（vrai）。
— `expression_logique1 -o expression_logique2`：如果两个表达式中至少有一个为真，则为真（vrai）。

6.17 case 结构（La structure case）

case 结构在 Shell 脚本中用于实现多路分支选择。其基本语法如下：

```
case paramètre in
    choix1.1 | choix 1.2 )
       commande1;commande2;;
   [ choix2 ) commandes ;; ]
   [ * ) commandes ;; ]
esac
```

— 每个参数值以) 结束。如果多个值需要相同处理，可以用 | 分隔开。
— 每个命令块以 ;; 结束。默认选项用 *) 标记，并且总是位于最后。

6.17.1 case 结构示例

以下是一个使用 **case** 结构的脚本示例，用于根据用户输入决定是否删除文件：

```
#!/bin/bash
echo 'voulez␣vous␣supprimer␣le␣fichier␣?'
read reponse
case $reponse in
    O | o ) rm $1 ;;
    N | n ) echo 'le␣fichier␣n'est pas supprimer ;;
    * ) echo 'réponse␣incorrecte' ;;
esac
```

6.18 循环结构

6.18.1 while 循环（La boucle tant que）

while 循环用于在条件为真时重复执行一系列命令。

— 语法：

```
while condition; do;
    commandes;
done
```

— 示例：

```
while [ -r $1 ]; do;
    cat $1 >> result;
    shift;
done
```

— 功能：将所有以参数形式给出的文件名对应的文件内容追加到文件 `result` 中。

6.18.2 until 循环（La boucle répéter jusqu'à）

until 循环用于在条件为假时重复执行一系列命令。

— 语法：

```
until condition; do;
    commandes;
done
```

— 示例：

```
until [ ! -r $1 ]; do;
    cat $1 >> concat;
    shift;
done
```

— 注意：测试的否定是通过！操作符实现的。

6.18.3 for 循环（La boucle for）

for 循环用于遍历列表中的每个元素并执行一系列命令。

— 语法：

```
for paramètre in liste; do;
    commandes;
done
```

— 示例：

```
for i in `ls`; do;
    cp $i /tmp/$i;
    echo '$i␣copié';
done
```

— 说明：`` `ls` `` 表示执行 `ls` 命令的结果。`` `ls` `` 等同于 `$(ls)$`。

— 功能：此脚本将当前目录中的每个文件复制到 `/tmp` 目录中。

6.19 条件跳转语句（Les sauts inconditionnels）

在 Shell 脚本中，可以使用 **continue** 和 **break** 语句来控制循环的执行流程。

6.19.1 continue 语句

`continue` 语句用于跳过当前循环迭代中剩余的命令，并继续下一次迭代。

```
while ... ; do
    ...
    if ... ; then
        continue
    fi
    ...
done
```

执行到 continue 直接回到 while

6.19.2 break 语句

`break` 语句用于立即终止循环。

```
while ... ; do
    ...
    if ... ; then
        break
    fi
done
```

执行到 break 立即跳转到 done

6.20 算术运算（L'arithmétique）

Shell 脚本中可以使用算术运算来执行整数运算。

6.20.1 使用 expr 命令

`expr` 命令提供了对变量进行算术运算的能力。

— 语法：`expr arg1 opérateur arg2`

— 示例：`expr 3 + 2 * 5`

— 结果：13（乘法优先于加法）

— 示例：`expr (3 + 2) * 5`

— 结果：25

6.20.2 expr 命令示例

```
a=1
while [ $a -le 10 ]
do
    echo $a
    a=$(expr $a + 1)
done
```

此脚本显示从 1 到 10 的整数。

6.20.3 使用（(））内部命令

（(）内部命令也用于执行算术运算。

— 语法：((expr_arith))

— 示例：

```
$ set $(date)
$ declare -i a=0
$ ((a=$2+1))
$ echo $a
```

— 为了获取算术表达式的值，使用语法：$((expr_arith))

— 示例：echo $((7 * 3))

— 命令替换和参数替换可以出现在内部命令（(））或算术表达式替换 $(( ))中，如果结果为整数。

— 例如，在表达式 $(($(ls -l | wc -l) -1))中，管道 ls -l | wc -l 的结果被解释为一个数字。

— 内部命令（(）的算术运算符包括：

— a++ 和 a--：后增量，后减量

— ++a 和 --a：前增量，前减量

— -a 和 +a：一元负号，一元正号

— a**b：指数运算

— a*b, a/b 和 a%b：乘法，整数除法，求余

6.21 算术表达式的 for 结构（Structure for pour les expressions arithmétiques）

Bash 引入了一种新的 **for** 结构，适用于算术表达式处理，源自 C 语言。

6.21.1 语法

语法：

```
for (( expr\_arith1 ; expr\_arith2 ; expr\_arith3 ))
do
    suite\_cmd
done
```

6.21.2 示例

以下是一个使用算术表达式 **for** 结构的脚本示例：

```
declare -i x y
for (( x=1, y=10 ; x<4 ; x++, y-- ))
do
    echo $(( x*y ))
done
```

在这个示例中，变量 x 和 y 被初始化，然后 **for** 循环在 x 小于 4 的条件下执行。每次迭代中，x 增加 1，而 y 减少 1，输出 x 和 y 的乘积。

6.22 一些小程序（Quelques petits scripts）

以下是一些简单的 Shell 脚本示例：

显示脚本的参数数量和最后一个参数编写一个脚本，显示其参数的数量和最后一个参数。

修改当前目录中文件的权限对于当前目录中的每个文件：

— 如果文件是普通文件，为用户添加写权限。

— 如果文件是目录，为组添加执行权限。

显示当前目录中所有普通文件的总大小编写一个脚本，在标准输出上显示当前目录中所有普通文件的累积大小。

给文件内容添加行号编写一个脚本，为传递给它的文件参数的每行添加行号。（使用 `cat -n fichier` 也可以实现相同的功能）。

```
    #!/bin/bash

# 显示脚本的参数数量和最后一个参数
echo "Nombre de paramètres : $#"
echo "Dernier argument : $1"

# 对于当前目录中的每个文件
```

```
8     for file in *
9     do
10        if [ -f "$file" ]; then  # 如果是普通文件
11            chmod u+w "$file"  # 添加写权限给用户
12        elif [ -d "$file" ]; then  # 如果是目录
13            chmod g+x "$file"  # 添加执行权限给组
14        fi
15    done
16
17    # 显示当前目录中所有普通文件的总大小
18    total_size=0
19    for file in *; do
20        if [ -f "$file" ]; then  # 检查是否是普通文件
21            total_size=$((total_size + $(stat --format="%s" "$file")
                  ))
22        fi
23    done
24    echo "La taille cumulée de tous les fichiers ordinaires : 
          $total_size octets"
25
26    # 给文件内容添加行号
27    if [ $# -eq 1 ]; then  # 检查是否传入了一个参数
28        num_lines() {
29            local line_num=1
30            while IFS= read -r line; do
31                echo "$line_num $line"
32                ((line_num++))
33            done
34        }
35        num_lines "$1"
36    else
37        echo "Veuillez passer un fichier en argument."
38    fi
```

7 函数（Les fonctions）

— **函数定义的语法**:

— function Nom {

```
    lines of code
    };
  — 或者 Nom () {
    lines of code
    };。
— 局部变量声明：
  — local -i var：声明一个整数变量 var。
  — local -a tab：定义一个数组 tab。
— 函数通过调用其名称并跟随参数来执行，参数的引用方式与脚本相同（$0, $1, …）。
— 命令 unset -f 用于删除一个函数。
— 命令 return 结束一个函数。返回码存储在 Shell 的 ? 变量中，就像脚本的返回码一样。
```

7.1 导出函数（Exporter une fonction）

为了使一个函数能在定义它的 Shell 脚本之外的另一个 Shell 程序中被执行，需要导出这个函数。可以使用内部命令 export 或 source。

7.1.1 语法

```
— export -f nomfct ...：导出函数 nomfct。
— source nomfct ...：也可以用于导出函数。
```

7.1.2 导出函数的条件

为了导出函数，执行函数的子 Shell 必须与导出函数的 Shell 脚本有继承关系。

7.2 函数示例（Exemples de fonctions）

以下为几个使用 Shell 函数的脚本示例：

1. 检查数字是奇数还是偶数

编写一个 Bash 脚本，该脚本使用函数来验证一个数字是奇数还是偶数。脚本要求用户输入一个数字，然后使用函数确定并显示结果。

```
1  #!/bin/bash
2
3  echo "Entrez un nombre :"
4  read nombre
5
```

```
estPair() {
    if (( nombre \%\ 2 == 0 )); then
        echo "$nombre est pair"
    else
        echo "$nombre est impair"
    fi
}

estPair $nombre
```

2. 递归函数：计算阶乘

编写一个脚本，该脚本使用递归函数计算参数的阶乘。

```
#!/bin/bash

echo "Entrez un nombre pour le calculer sa factorielle :"
read nombre

factorielle() {
    local result=1
    for (( i=1; i<=$1; i++ )); do
       result=$((result * i))
   done
   echo $result
}

factorielle $nombre
```

3. 计算当前目录下各子目录的大小

编写一个 Bash 脚本，该脚本使用函数计算当前目录下各子目录的大小。

```
#!/bin/bash

echo "Calcul de la taille des répertoires du répertoire courant:
   "
for repertoire in */ ; do
    if [ -d "$repertoire" ]; then  # Vérification si c'est bien
       un répertoire
        taille=$((du -sh "$repertoire" 2> /dev/null))
        echo "$repertoire : $taille octets"
    fi
done
```

7.3 特殊变量 $ 和 !

特殊变量在 Shell 脚本中具有特定的用途：

— 变量 ! 包含最后后台运行的任务的进程号（numéro de processus）。

— 变量 $ 允许你杀死脚本启动的所有进程。

7.3.1 示例脚本

以下是一个示例脚本，它展示了如何使用这些特殊变量：

```
#!/bin/bash
# 启动一个后台进程
( while do
    sleep 4
   echo 'bonjour'
 done ) &
# 等待10秒
sleep 10
# 杀死后台进程
kill -KILL $!

echo 'cest␣fini'
```

7.4 命名管道的数据交换（L'échange de données par tube nommé）

命名管道允许两个进程通过一个名为管道进行数据交换：第一个进程将结果写入管道，而第二个进程从管道读取数据。

7.4.1 命名管道的数据交换方法

— 命令 `exec` 用于以读写文件的方式打开命名管道。

— 命令 `read` 用于从管道中读取数据。

— 命令 `mknod` **nom_du_tube p** : 创建名为 **nom_du_tube** 的命名管道。

命名管道允许两个进程通过一个命名的管道进行数据交换。以下是使用命名管道进行数据交换的示例：

7.4.2 示例脚本 p1

```
#!/bin/bash
exec > tube
while read var
do
    echo $var
done
```

7.4.3 示例脚本 p2

```
#!/bin/bash
exec < tube
while read ligne
do
    echo ">$ligne"
done
```

7.4.4 创建和使用命名管道

— 使用 `mknod tube p` 创建一个名为 `tube` 的命名管道。
— 启动脚本 `p1` 并将其输出重定向到管道 `tube`。
— 启动脚本 `p2` 并在后台运行，从管道 `tube` 读取输入。

```
$ mknod tube p
$ bash p1
bonjour
>bonjour
```

7.4.5 脚本执行

— 执行 `mknod tube p` 创建命名管道。
— 执行 `bash p1` 向管道写入数据 `"bonjour"`。
— 执行 `bash p2 &` 从管道读取数据并输出。

```
$ bash p2 &
```

8 文件系统（续 2）

8.1 文件的高级命令（Commandes avancées sur les fichiers）

8.1.1 文件排序

命令 `sort` 用于对文件进行排序。

— 默认情况下，排序是按照 ASCII 码的字母顺序进行的。

— 数字会在字母之前出现，大写字母会在小写字母之前出现。

8.1.2 文件分割

命令 `split` 将文件分割成包含特定行数的多个文件（默认为 1000 行），生成的文件名为 `fic_outaa`, `fic_outab` 等。

— `split [-n] [fic_in] [fic_out]`：按行数分割文件。

8.1.3 高级文件命令示例

— 使用 `ls -l | sort -n +6 -8` 按第 6 到第 8 列的数字排序文件（日期，时间）。

— 使用 `split -10 exemple.c fich` 将文件 `exemple.c` 每 10 行分割为一个新文件。

— 使用 `ls -l` 查看分割后的文件列表。

— 比较文件或目录内容的命令 `diff` 按行比较文本文件内容，并将差异输出到标准输出。

— 搜索特定字符串的命令 `grep [options] 'motif' fichier` 将文件中包含指定字符串的行输出到标准输出。

— 搜索文件的命令 `find` 可以根据指定的条件查找文件。使用条件前缀 ! 来查找不满足条件的文件。

8.2 查找命令（Commande find）

find 命令用于在文件系统中搜索文件和目录。以下是一些使用 **find** 命令的示例：

— `find / -name fichier.txt`：在根目录下查找名为 `fichier.txt` 的文件。

— `find / -name file*`：在根目录下查找所有以 `file` 开头的文件。

— `find / -f -name file*`：强制不使用任何魔术字符的查找。

— `find / -d -name file*`：仅查找目录名匹配 `file*` 的目录。

— `find / -d -iname file*`：忽略大小写查找目录名匹配 `file*` 的目录。

— `find /home /home/florent -name file*`：在指定路径下查找所有以 `file` 开头的文件。
— `find / -f -exec grep -i file` ：对找到的每个文件执行 `grep` 命令。
— `find / -f –size 2M`：查找大小恰好为 2MB 的文件。
— `find / -f –size +100k –size -200k`：查找大小在 100KB 到 200KB 之间的文件。
— `find / -mtime +8 –perm 777 –exec chmod 654` ：查找修改时间超过 8 天且权限为 777 的文件，并将其权限改为 654。
— `find / -f –size 0 –exec rm –f` ：查找并删除所有非空文件。

9 进程（Les processus）

— 我们称正在执行的程序的一个实例为**进程（processus）**。进程是唯一的，并且由系统在创建时赋予一个 **PID**（进程标识号）。
— 进程使用的内存是私有的，并且对其他进程不可见。内存在逻辑上是私有的。
— 进程所使用的**环境变量（variables d'environnement）**是其特有的（如用户身份、当前目录等）。

9.1 进程管理命令（Commandes de gestion des processus）

进程管理是系统管理中的一项重要任务，以下是一些常用的进程管理命令：

9.1.1 列出进程（Liste de processus）

命令 `ps` 用于显示有关进程的信息，特别是进程的 **PID**（进程标识号），其状态，使用的 **CPU** 时间以及执行的命令。

9.1.2 终止进程（Tuer un processus）

有时一个进程可能运行时间过长或无响应。
— 命令 `kill` 可以用来干预该进程。需要注意的是，尽管名为 `kill`，该命令实际上并不直接杀死进程，而是向进程发送一个信号。

9.1.3 示例

— 使用 `ps [options] [PID]` 列出进程及其信息。
— 使用 `kill` 发送信号以请求进程终止。

9.2 进程管理命令：示例（Commandes de gestion des processus : exemples）

以下是一些进程管理命令的示例：

9.2.1 列出进程

命令 `ps` 用于显示当前运行的进程列表。

```
$ ps
PID TTY TIME CMD
18101 ttyp9 0:00.01 sleep 4000
```

该命令显示了进程的 **PID**（进程标识号）、**TTY**（终端）、**TIME**（占用时间）和 **CMD**（命令）。

9.2.2 终止进程

如果一个进程运行时间过长或无响应，可以使用 `kill` 命令来终止它。

— 使用 `kill -9 18101` 发送 **-9**（或 **-s KILL**）信号来强制终止进程 `sleep 4000`。

— 输出显示进程 `"sleep 4000"` 已被终止。

9.2.3 发送停止信号

进程可能不会立即终止，而是可以被暂停。

```
$ sleep 200
[1] 12623
$ kill -s STOP 12623
```

此命令发送 **-s STOP** 信号给进程 `12623`，进程没有被终止，而是被暂停。

9.3 任务管理（Les jobs : manipulation）

一个任务（job）是由 Shell 创建和控制的工作。它可以由一个或多个进程组成，例如通过管道连接。

9.3.1 后台执行

— `cmd`：在前台（foreground）执行。

— `cmd &`：在后台（background），作为后台任务执行。

9.3.2 任务列表（Liste des jobs）

命令 `jobs` 列出当前 Shell 的任务（挂起或后台执行）及其 **jobID**。

9.3.3 任务管理

— 使用 CTRL-C 在前台停止当前任务。
— 使用 kill %jobID 终止编号为 jobID 的任务。
— 使用 CTRL-Z 挂起前台运行的任务。要继续任务，只需发送 CONT 信号。
— 可以将任务移至前台或后台：
  — 命令 bg 将任务移至后台。
  — 命令 fg %jobID 将任务 jobID 移至前台。

9.3.4 任务管理示例

以下是一些任务管理的示例：

```
$ (ls -l >fic.tmp ; sleep 60) &
[1] 22325
$ ps
PID TTY TIME CMD
7840 ttyp5 0:07.73 ls -l
22325 ttyp5 0:00.00 -csh (csh)
$ ps
PID TTY TIME CMD
21903 ttyp5 0:00.01 sleep 60
22325 ttyp5 0:00.00 -csh (csh)
```

在这个示例中：
— 使用 sleep 200 & 启动后台任务，任务 ID 为 20547。
— 使用 jobs 列出所有任务，显示任务 20547 正在运行。
— 使用 kill -s STOP %1 停止任务 1。
— 使用 bg %1 将任务移至后台。
— 使用 fg %1 将任务移至前台。

9.4 脚本管理的实用命令（Quelques commandes utiles pour la gestion des scripts）

以下是一些用于管理脚本的实用命令：

9.4.1 后台执行任务

— 命令 nice 允许以指定的优先级在后台执行任务：

```
nice commande &
```

— 命令 nohup 确保在当前会话结束时（例如退出）后台任务被终止。如果用户没有在命令行中明确重定向标准输出，则结果会写入文件 nohup.out：

```
nohup commande &
```

9.4.2 进程管理的实用命令

— 命令 at 允许在指定的时间和日期执行命令：

```
at heure [date]
```

— 命令 crontab 允许按预定的周期执行文件中的所有命令，从每分钟到每年一次。

— 命令 wait 要求 Shell 等待后台任务完成再继续执行：

```
wait [PID]
```

10 菜单管理（La gestion des menus）

10.1 菜单操作示例

以下是一个 Shell 脚本示例，它展示了如何使用 select 命令创建一个交互式菜单，用户可以通过输入数字选择不同的操作。

— 定义提示信息 PS3="votre choix ?" 询问用户的选择。

— 使用 select 命令列出可供选择的选项，包括 "archive"（压缩）、"restaure"（解压）和 "fin"（结束）。

— 使用 do...done 循环结构来处理用户的选择。

— 使用 case 语句根据用户的选择执行相应的操作：

  — 如果用户选择 1，则执行 tar -c 命令压缩文件。

  — 如果用户选择 2，则执行 tar -x arch 命令解压文件。

  — 如果用户选择 3，则使用 break 命令退出循环。

— 使用 echo "==> $choix" 显示用户的选择。

10.1.1 脚本示例

以下是脚本的代码示例：

```
1 PS3="votre␣choix␣?"
2 select choix in \\
3     "archive" "restaure" "fin" \\
```

```
4     do
5         echo "==>␣\$choix"
6         case $REPLY in
7             1) tar -c ;;
8             2) tar -x arch ;;
9             3) break ;;
10        esac
11    done
```

在这个示例中：

— 用户被提示选择 1） archive、2） restaure 或 3） fin。

— 根据用户的选择，脚本执行相应的文件压缩或解压操作。

— 用户的选择将显示在终端上。

— 变量 PS3 的内容被显示，并且从标准输入读取。如果列表中的一个数字被输入，参数 choix 将取得该值。

— 如果输入的行是空的，则列表将再次显示。

— 输入的行内容被保存在变量 REPLY 中。列表将一直执行直到遇到一个指定在列表中的中断字符。

10.2 实用命令（Quelques commandes utiles）

10.2.1 what 命令（La commande what）

命令 what 用于显示脚本的元数据信息。

— 示例：$ more lescript

```
#!/bin/ksh
# @(#) lescript du 25/03/2005
# @(#) version 1.3
date
```

— 使用 **what** 命令显示脚本信息：

```
$ what lescript
le script du 25/03/2005
version 1.3
```

10.2.2 xargs 命令（La commande xargs）

命令 **xargs** 从标准输入生成命令的参数。

— 示例：`$ cat > liste`
`fich1`
`fich2`
`$ cat liste | xargs tar -c # tar -c fich1 ; tar -c fich2`
— 等价于 `tar -c $(cat liste)`

10.2.3 tput 命令（La commande tput）

命令 **tput** 用于控制终端光标和终端控制序列，如清屏、光标位置、反转视频等。
— 示例：`$ tput clear # effacer l'écran`
— 示例：`$ tput cup 3 5 # positionner le curseur en ligne 3 et colonne 5`
— 示例：`$ tput rev # passer en mode vidéo inverse`
— 示例：`$ tput sgr0 # revenir en mode normal`

10.2.4 stat 命令（La commande stat）

命令 **stat** 用于获取文件或目录的信息。默认情况下，此命令显示 3 个日期：
— 文件内容最后被读取的日期（La date à laquelle le contenu du fichier a été lu）
— 文件内容最后被修改的日期（La date à laquelle le contenu du fichier a été modifié）
— 文件权限最后被修改的日期（La date à laquelle les permissions sur ce fichier ont été modifiées）
使用 `stat -c %G file` 命令显示文件的组（le groupe du fichier）。

10.2.5 file 命令（La commande file）

命令 **file** 用于识别文件类型。
— 示例：使用 `stat -c %s file` 命令显示文件大小（la taille du fichier）。

11 字符串

— Shell 使用不同的 **caractères particuliers**（特殊字符）来执行其自身的处理（**$** 用于访问变量的值，**>** 用于标准输出的重定向，***** 作为 **caractère générique**（通配符）等）。为了将这些特殊字符作为普通字符使用，需要防止它们被 Shell 解释。有三种机制可以使用：
  — **Protection d'un caractère à l'aide du caractère **（使用反斜杠保护单个字符）：反斜杠（\）会保护紧随其后的字符。
  示例：$ echo * => 字符 * 失去了通配符的意义。

— **Protection de caractères à l'aide d'une paire de guillemets**（使用双引号保护字符）：除了 **$**、\、`和 "之外的所有字符都会被保护，不会被 Shell 解释。这意味着，在双引号内，**$** 仍然会被解释。
示例：$ echo "$SHELL"
输出：/bin/bash
— **Protection totale**（完全保护）：在一对单引号（'）之间，除了单引号本身之外，没有任何字符会被解释。
示例：$ echo '$SHELL'
输出：$SHELL

11.1 字符串的长度

— **Syntaxe**（语法）：$#paramètre
此语法会被替换为 **paramètre**（参数）中字符串的长度。该参数可以是一个变量或位置参数。
— **Exemples**（示例）：
— $ echo $#SHELL
10 => 字符串 /bin/bash 的长度
— $ set " bonjour"
$ echo $#1
7 => $1 的值为 bonjour，其长度为 7

11.2 子字符串提取

— **Syntaxe**（语法）：$**paramètre** :ind :nb
从 **paramètre**（参数）的值中提取从索引 **ind** 开始的子字符串。参数的值不会被修改。
— **Exemples**（示例）：
— $ chaine="totomobile"
$ echo $chaine :3
omobile
— $ echo $chaine :3 :2
om
— $ set totomobile
$ echo $1 :3 :4
omob

11.3 子字符串替换

— **Syntaxe**（语法）：$**paramètre** :**mod** :**ch**
bash 在 **paramètre**（参数）的值中查找符合 **mod**（模式）的最长子字符串，并将其替换为 **ch**（目标字符串）。只有第一个匹配的子字符串会被替换。参数的值不会被修改。

— **Exemples**（示例）：

— $ chaine=”totomobile”
$ echo $chaine :toto :titi
titimobile

— **Précision de l’occurrence**（指定匹配位置）：可以指定是否希望在参数字符串的开头（使用语法：#mod）或结尾（使用语法：

— $ set totomototo
$ echo $1 :%toto :titi
totomotiti

12 数组

— 创建数组（Pour créer un tableau）：通常使用 **declare** 命令的 **-a** 选项（表示 array，数组）：

```
declare -a tab ...
```

— 数组 **tab** 被创建，但不包含任何值：数组被定义，但未被初始化。

— 查看已定义的数组（Pour connaître les tableaux définis）：使用 **declare -a**。

— 定义并初始化数组（Pour définir et initialiser un tableau）：

```
declare -a tab=(val0 val1 ...)
```

— 与 C 语言类似，数组的索引总是从 0 开始。

— 示例（Exemples）：

```
$ tableau[0]=un
$ tableau[1]=deux
$ tableau[2]=trois
$ echo ${tableau[0]}
un
$ tableau[1]=six
$ echo ${tableau[*]}
un six trois
$ echo ${#tableau[*]}
```

```
3
```

12.1 关于数组的一些简单脚本

1. 创建一个脚本，该脚本需要使用数组记录作为参数给出的整数数量，然后在同一个数组中将这些整数按升序排序，最后将排序后的数组内容输出到标准输出。

```bash
#!/bin/bash

# 检查是否有参数传入
if [ $# -eq 0 ]; then
    echo "Usage: $0 <list of integers>"
    exit 1
fi
# 将参数存储到数组中
tab=("$@")
# 对数组进行排序
IFS=$'\n' sorted_tab=($(sort -n <<<"${tab[*]}"))
unset IFS
# 输出排序后的数组
echo "Sorted array: ${sorted_tab[*]}"
```

2. 示例：关联数组（Exemple de tableau associatif）：编写一个 bash 脚本，在标准输出上显示当前目录中每个组所属的所有文件的累积大小。

```bash
#!/bin/bash

# 创建一个关联数组来存储每个组的文件大小总和
declare -A group_sizes
# 遍历当前目录中的所有文件
for file in *; do
    if [ -f "$file" ]; then
        # 获取文件所属的组
        group=$(stat -c '%G' "$file")
        # 获取文件大小并累加到对应组的总大小中
        size=$(stat -c '%s' "$file")
        group_sizes[$group]=$((group_sizes[$group] + size))
    fi
done
# 输出每个组的累积大小
for group in "${!group_sizes[@]}"; do
```

```
17      echo "Group: $group, Total Size: ${group_sizes[$group]} bytes"
18  done
```