Linux入门教程

探索命令行

Linux命令行中的命令使用格式都是相同的:

```
命令名称 参数1 参数2 参数3 ...
```

参数之间用任意数量的空白字符分开. 关于命令行, 可以先阅读一些基本常识. 然后我们介绍最常用的一些命令:

- `ls` 用于列出当前目录(即"文件夹")下的所有文件(或目录). 目录会用蓝色显示. `ls -l` 可以显示详细信息.
- `pwd` 能够列出当前所在的目录.
- `cd DIR` 可以切换到 `DIR` 目录. 在Linux中, 每个目录中都至少包含两个目录: `.` 指向该目录自身, `..` 指向它的上级目录. 文件系统的根是 `/` .
- `touch NEWFILE` 可以创建一个内容为空的新文件 `NEWFILE` , 若 `NEWFILE` 已存在, 其内容不会丢失.
- `cp SOURCE DEST` 可以将 `SOURCE` 文件复制为 `DEST` 文件; 如果 `DEST` 是一个目录, 则将 `SOURCE` 文件复制到该目录下.
- `mv SOURCE DEST` 可以将 `SOURCE` 文件重命名为 `DEST` 文件; 如果 `DEST` 是一个目录, 则将 `SOURCE` 文件移动到该目录下.
- `mkdir DIR` 能够创建一个 `DIR` 目录.
- `rm FILE` 能够删除 `FILE` 文件; 如果使用 `-r` 选项则可以递归删除一个目录. 删除后的文件无法恢复, 使用时请谨慎!
- `man` 可以查看命令的帮助. 例如 `man ls` 可以查看 `ls` 命令的使用方法. 灵活应用 `man` 和互联网搜索, 可以快速学习新的命令.

下面给出一些常用命令使用的例子, 你可以键入每条命令之后使用 `ls` 查看命令执行的结果:

```
$ mkdir temp        # 创建一个目录temp
$ cd temp            # 切换到目录temp
$ touch newfile        # 创建一个空文件newfile
$ mkdir newdir       # 创建一个目录newdir
$ cd newdir            # 切换到目录newdir
$ cp ../newfile .    # 将上级目录中的文件newfile复制到当前目录下
$ cp newfile aaa    # 将文件newfile复制为新文件aaa
$ mv aaa bbb        # 将文件aaa重命名为bbb
$ mv bbb ..            # 将文件bbb移动到上级目录
$ cd ..                # 切换到上级目录
$ rm bbb            # 删除文件bbb
$ cd ..                # 切换到上级目录
$ rm -r temp        # 递归删除目录temp
```

消失的cd

上述各个命令除了 `cd` 之外都能找到它们的manpage. 这是为什么? 如果你思考后仍然感到困惑, 试着到互联网上寻找答案.

`man` 的功能不仅限于此. `man` 后可以跟两个参数, 可以查看不同类型的帮助(请在互联网上搜索). 例如当你不知道C标准库函数 `freopen` 如何使用时, 可以键入命令

```
man 3 freopen
```

学会使用man

如果你是第一次使用 `man` 请阅读附件**5** 这个教程除了说明如何使用 `man` 之外, 还会教你在使用一款新的命令行工具时如何获得帮助.

更多的命令行知识

仅仅了解这些最基础的命令行知识是不够的. 通常, 我们可以抱着如下的信条: 只要我们能想到的, 就一定有方便的办法能够办到. 因此当你想要完成某件事却又不知道应该做什么的时候, 请向Baidu 求助.

如果你想以Linux作为未来的事业, 那就可以去图书馆或互联网上找一些相关的书籍来阅读.

统计代码行数

第一个例子是统计一个目录中(包含子目录)中的代码行数. 如果想知道当前目录下究竟有多少行的代码, 就可以在命令行中键入如下命令:

```
find . | grep '\.c$\|\.h$' | xargs wc -l
```

如果用 `man find` 查看 `find` 操作的功能, 可以看到 `find` 是搜索目录中的文件. Linux中一个点 `.` 始终表示Shell当前所在的目录, 因此 `find .` 实际能够列出当前目录下的所有文件. 如果在文件很多的地方键入 `find .` ,将会看到过多的文件, 此时可以按 `CTRL + c` 退出.

同样, 用 `man` 查看 `grep` 的功能——"print lines matching a pattern". `grep` 实现了输入的过滤, 我们的 `grep` 有一个参数, 它能够匹配以 `.c` 或 `.h` 结束的文件. 正则表达式是处理字符串非常强大的工具之一, 每一个程序员都应该掌握其相关的知识. 有兴趣的同学可以首先阅读一个基础的教程, 然后看一个有趣的小例子: 如何用正则表达式判定素数. 正则表达式还可以用来编写一个30行的java表达式求值程序(传统方法几乎不可能), 聪明的你能想到是怎么完成的吗? 上述的 `grep` 命令能够提取所有 `.c` 和 `.h` 结尾的文件.

刚才的 `find` 和 `grep` 命令, 都从标准输入中读取数据, 并输出到标准输出. 关于什么是标准输入输出, 请参考这里. 连接起这两个命令的关键就是管道符号 `|` . 这一符号的左右都是Shell命令, `A | B` 的含义是创建两个进程 `A` 和 `B` , 并将 `A` 进程的标准输出连接到 `B` 进程的标准输入. 这样, 将 `find` 和 `grep` 连接起来就能够筛选出当前目录(`.`)下所有以 `.c` 或 `.h` 结尾的文件.

我们最后的任务是统计这些文件所占用的总行数, 此时可以用 `man` 查看 `wc` 命令. `wc` 命令的 `-l` 选项能够计算代码的行数. `xargs` 命令十分特殊, 它能够将标准输入转换为参数, 传送给第一个参数所指定的程序. 所以, 代码中的 `xargs wc -l` 就等价于执行 `wc -l aaa.c bbb.c include/ccc.h ...` , 最终完成代码行数统计.

统计磁盘使用情况

以下命令统计 `/usr/share` 目录下各个目录所占用的磁盘空间:

```
du -sc /usr/share/* | sort -nr
```

`du` 是磁盘空间分析工具, `du -sc` 将目录的大小顺次输出到标准输出, 继而通过管道传送给 `sort` . `sort` 是数据排序工具. 其中的选项 `-n` 表示按照数值进行排序, 而 `-r` 则表示从大到小输出. `sort` 可以将这些参数连写在一起.

然而我们发现, `/usr/share` 中的目录过多, 无法在一个屏幕内显示. 此时, 我们可以再使用一个命令: `more` 或 `less` .

```
du -sc /usr/share/* | sort -nr | more
```

此时将会看到输出的前几行结果. `more` 工具使用空格翻页, 并可以用 `q` 键在中途退出. `less` 工具则更为强大, 不仅可以向下翻页, 还可以向上翻页, 同样使用 `q` 键退出. 这里还有一个关于less的小故事.

在Linux下编写Hello World程序

Linux中用户的主目录是 `/home/用户名称` , 如果你的用户名是 `user` , 你的主目录就是 `/home/user` . 用户的 `home` 目录可以用波浪符号 `~` 替代, 例如临时文件目录 `/home/user/Templates` 可以简写为 `~/Templates` . 现在我们就可以进入主目录并编辑文件了. 如果 `Templates` 目录不存在, 可以通过 `mkdir` 命令创建它:

```
cd ~
mkdir Templates
```

创建成功后, 键入

```
cd Templates
```

可以完成目录的切换. 注意在输入目录名时, `tab` 键可以提供联想.

你感到键入困难吗

你可能会经常要在终端里输入类似于

```
cd AVeryVeryLongFileName
```

的命令, 你一定觉得非常烦躁. 回顾上面所说的原则之一: 如果你感到有什么地方不对, 就一定有什么好办法来解决. 试试 `tab` 键吧.

Shell中有很多这样的小技巧, 你也可以使用其他的Shell例如zsh, 提供更丰富好用的功能. 总之, 尝试和改变是最重要的.

进入正确的目录后就可以编辑文件了, 开源世界中主流的两大编辑器是 `vi(m)` 和 `emacs` , 你可以使用其中的任何一种. 如果你打算使用 `emacs` , 你还需要安装它

```
apt-get install emacs
```

`vi` 和 `emacs` 这两款编辑器都需要一定的时间才能上手, 它们共同的特点是需要花较多的时间才能适应基本操作方式(命令或快捷键), 但一旦熟练运用, 编辑效率就比传统的编辑器快很多.

进入了正确的目录后, 输入相应的命令就能够开始编辑文件. 例如输入

```
vi hello.c
或emacs hello.c
```

就能开启一个文件编辑. 例如可以键入如下代码(对于首次使用 `vi` 或 `emacs` 的同学, 键入代码可能会花去一些时间, 在编辑的同时要大量查看网络上的资料):

```
#include <stdio.h>
int main(void) {
    printf("Hello, Linux World!\n");
        return 0;
}
```

保存后就能够看到 `hello.c` 的内容了. 终端中可以用 `cat hello.c` 查看代码的内容. 如果要将它编译, 可以使用 `gcc` 命令:

```
gcc hello.c -o hello
```

`gcc` 的 `-o` 选项指定了输出文件的名称, 如果将 `-o hello` 改为 `-o hi` , 将会生成名为 `hi` 的可执行文件. 如果不使用 `-o` 选项, 则会默认生成名为 `a.out` 的文件, 它的含义是assembler output. 在命令行输入

```
./hello
```

就能够运行改程序. 命令中的 `./` 是不能少的, 点代表了当前目录, 而 `./hello` 则表示当前目录下的 `hello` 文件. 与Windows不同, Linux系统默认情况下并不查找当前目录, 这是因为Linux下有大量的标准工具(如 `test` 等), 很容易与用户自己编写的程序重名, 不搜索当前目录消除了命令访问的歧义.

使用重定向

有时我们希望将程序的输出信息保存到文件中, 方便以后查看. 例如你编译了一个程序 `myprog` , 你可以使用以下命令对 `myprog` 进行反汇编, 并将反汇编的结果保存到 `output` 文件中:

```
objdump -d myprog > output
```

`>` 是标准输出重定向符号, 可以将前一命令的输出重定向到文件 `output` 中. 这样, 你就可以使用文本编辑工具查看 `output` 了.

但你会发现, 使用了输出重定向之后, 屏幕上就不会显示 `myprog` 输出的任何信息. 如果你希望输出到文件的同时也输出到屏幕上, 你可以使用 `tee` 命令:

```
objdump -d myprog | tee output
```

使用输出重定向还能很方便地实现一些常用的功能, 例如

```
> empty            # 创建一个名为empty的空文件
cat old_file > new_file    # 将文件old_file复制一份, 新文件名为new_file
```

如果 `myprog` 需要从键盘上读入大量数据(例如一个图的拓扑结构), 当你需要反复对 `myprog` 进行测试的时候, 你需要多次键入大量相同的数据. 为了避免这种无意义的重复键入, 你可以使用以下命令:

```
./myprog < data
```

`<` 是标准输入重定向符号, 可以将前一命令的输入重定向到文件 `data` 中. 这样, 你只需要将 `myprog` 读入的数据一次性输入到文件 `data` 中, `myprog` 就会从文件 `data` 中读入数据, 节省了大量的时间.

下面给出了一个综合使用重定向的例子:

```
time ./myprog < data | tee output
```

这个命令在运行 `myprog` 的同时, 指定其从文件 `data` 中读入数据, 并将其输出信息打印到屏幕和文件 `output` 中. `time` 工具记录了这一过程所消耗的时间, 最后你会在屏幕上看到 `myprog` 运行所需要的时间. 如果你只关心 `myprog` 的运行时间, 你可以使用以下命令将 `myprog` 的输出过滤掉:

```
time ./myprog < data > /dev/null
```

`/dev/null` 是一个特殊的文件, 任何试图输出到它的信息都会被丢弃, 你能想到这是怎么实现的吗? 总之, 上面的命令将 `myprog` 的输出过滤掉, 保留了 `time` 的计时结果, 方便又整洁.

使用Makefile管理工程

大规模的工程中通常含有几十甚至成百上千个源文件(Linux内核源码有25000+的源文件), 分别键入命令对它们进行编译是十分低效的. Linux提供了一个高效管理工程文件的工具: GNU Make. 我们首先从一个简单的例子开始, 考虑上文提到的Hello World的例子, 在 `hello.c` 所在目录下新建一个文件 `Makefile` , 输入以下内容并保存:

```
hello:hello.c
        gcc hello.c -o hello    # 注意开头的tab, 而不是空格

.PHONY: clean

clean:
        rm hello        # 注意开头的tab, 而不是空格
```

返回命令行, 键入 `make` , 你会发现 `make` 程序调用了 `gcc` 进行编译. `Makefile` 文件由若干规则组成, 规则的格式一般如下:

```
目标文件名:依赖文件列表
    用于生成目标文件的命令序列    # 注意开头的tab, 而不是空格
```

我们来解释一下上文中的 `hello` 规则. 这条规则告诉 `make` 程序, 需要生成的目标文件是 `hello` , 它依赖于文件 `hello.c` , 通过执行命令 `gcc hello.c -o hello` 来生成 `hello` 文件.

如果你连续多次执行 `make` , 你会得到"文件已经是最新版本"的提示信息, 这是 `make` 程序智能管理的功能. 如果目标文件已经存在, 并且它比所有依赖文件都要"新", 用于生成目标的命令就不会被执行. 你能想到 `make` 程序是如何进行"新"和"旧"的判断的吗?

上面例子中的 `clean` 规则比较特殊, 它并不是用来生成一个名为 `clean` 的文件, 而是用于清除编译结果, 并且它不依赖于其它任何文件. `make` 程序总是希望通过执行命令来生成目标, 但我们给出的命令 `rm hello` 并不是用来生成 `clean` 文件, 因此这样的命令总是会被执行. 你需要键入 `make clean` 命令来告诉 `make` 程序执行 `clean` 规则, 这是因为 `make` 默认执行在 `Makefile` 中文本序排在最前面的规则. 但如果很不幸地, 目录下已经存在了一个名为 `clean` 的文件, 执行 `make clean` 会得到"文件已经是最新版本"的提示. 解决这个问题的方法是在 `Makefile` 中加入一行 `PHONY: clean` , 用于指示" `clean` 是一个伪目标", 这样以后, `make` 程序就不会判断目标文件的新旧, 伪目标相应的命令序列总是会被执行.

对于一个规模稍大一点的工程, `Makefile` 文件还会使用变量, 函数, 调用Shell命令, 隐含规则等功能. 如果你希望学习如何更好地编写一个 `Makefile` , 请到互联网上搜索相关资料.