Unix系统高级编程

- 业务逻辑：根据业务需求，按照设计好的逻辑规则，处理信息，与系统无关。
- 系统访问：利用操作系统所提供的各种功能辅助业务逻辑的实现。
- 标准函数：scanf/printf - 源代码级兼容
- 系统函数：read/write - 接口级兼容
- 环境、性能、功能

一、Unix系统简介

1.1 Unix系统的背景

1961-1969：史前时代
CTSS(Compatible Time-Sharing System，兼容分时系统)，以MIT为首的开发小组，小而简单的实验室原型。
Multics(Multiplexed Information and Computing System，多路信息与计算系统)，庞大而负责，不堪重负。
Unics(Uniplexed information and Computing System，单路信息与计算系统)，返璞归真，走上正道。
1969-1971：创世纪
Ken Thompson，肯.汤普逊，Unix之父，B语言之父，内核用B语言+汇编语言开发，PDP-7，第一个Unix系统核心和简单应用。后来被移植到PDP-11平台，功能更加完善。
1971-1979：出谷纪
Dennis Ritchie，丹尼斯.里奇，C语言之父，用C语言重写了Unix系统内核，极大地提升了Unix系统的可读性、可维护性和可移植性——Unix V7，第一个真正意义上的Unix系统。
1980-1985：第一次Unix战争
AT&T贝尔实验室：SVR4
加州大学伯克利分校：BSD+TCP/IP
DARPA，ARPANET(INTERNET)
IEEE，国际电气电子工程师协会，POSIX为Unix内核和外壳制定了一系列技术标准和规范，消除系统版本之间分歧，大一统的操作系统。
1988-1990：第二次Unix战争
AT&T+Sun
IBM+DEC+HP
比尔.盖茨->Windows
1990-现在
1991，Linus Torvalds创建了Linux系统的内核
1993，Linux已达到产品级操作系统的水准
1993，AT&T将Unix系统卖给Novell
1994，Novell将Unix系统卖给X/Open组织
1995，X/Open将Unix系统捐给SCO
2000，SCO将Unix系统卖给Celdear——Linux发行商
Linux就是现代版本的Unix。

1.2 Linux系统的背景

类Unix操作系统，免费开源。
不同发行版本使用相同的内核。
支持多种硬件平台：手机、路由器、视频游戏控制器、个人电脑、大型计算机等等。
隶属于GNU工程。GNU = GNU Not Unix。
受GPL许可证限制：如果发布了可执行的二进制代码，就必须同时发布可读的源代码，并且在发布任何基于GPL许可证的软件时，不能添加任何限制性条款。

1.3 Linux系统的版本

早期版本：0.01，0.02，...，1.00
旧计划：1.0.1，...，2.6.0 (A.B.C)
A - 主版本号，内核大幅更新
B - 次版本号，内核重大修改，奇数测试版，偶数稳定版
C - 补丁序号，内核轻微修改
新计划：A.B.C-D.E
D - 构建次数，反映极微小的更新
E - 描述信息
    rc/r - 候选版本
    smp - 支持对称多处理器
    EL - Rad Hat的企业版本
    mm - 试验新技术
    ...
cat /proc/version

1.4 Linux系统的特点

遵循GNU/GPL许可证
开放性
多用户
多任务
设备无关性
丰富网络功能
可靠的系统安全
良好的可移植性

1.5 Linux的发行版本

Ubuntu - 大众化，简单易用
Linux Mint - 新潮前位
Fedora - Red Hat的桌面版本
openSUSE - 华丽
Debian - 自由开放
Slackware - 朴素简洁，简陋
Red Hat - 经典，稳定，企业应用，支持全面

二、GNU编译器(gcc)

2.1 GCC的基本特点

1)支持多种硬件架构
x86-64
Alpha
ARM
PowerPC
SPARC
VAX
...
2)支持多种操作系统
Unix
Linux
BSD
Android
Mac OS X
iOS
Windows
3)支持多种编程语言
C
C++
Objective-C
Java
Fortran
Pascal
Ada
4)GCC的版本
gcc -v

2.2 构建过程

源代码(.c)-预编译->头文件和宏扩展-编译->汇编码(.s)-汇编->目标码(.o)-链接->可执行代码(a.out)
代码：hello.c

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

vi hello.c - 编写源代码
gcc -E hello.c -o hello.i - 预编译(编译预处理)
gcc -S hello.i - 获得汇编代码(hello.s)
gcc -c hello.s - 获得目标代码(hello.o)
gcc hello.o -o hello - 获得可执行代码(hello)
./hello - 运行可执行代码

2.3 文件名后缀

可读文本

> .h - C语言源代码头文件
> .c - 预处理前的C语言源代码文件
> .s - 汇编语言文件

不可读的二进制

> .o - 目标文件
> .a - 静态库文件
> .so - 共享(动态)库文件
> .out - 可执行文件

2.4 编译选项

gcc [选项] [参数] 文件1 文件2 ...
-o: 指定输出文件
如：gcc hello.c -o hello
-E: 预编译，缺省输出到屏幕，用-o指定输出文件
如：gcc -E hello.c -o hello.i
-S: 编译，将高级语言文件编译成汇编语言文件
如：gcc -S hello.c
-c: 汇编，将汇编语言文件汇编成机器语言文件
如：gcc -c hello.s
-Wall：产生全部警告
如：gcc -Wall wall.c
-Werror：将警告作为错误处理
如：gcc -Werror werror.c
-x: 指定源代码的语言
xxx.c - C语言
xxx.cpp - C++语言
xxx.for - Fortran语言
xxx.java - Java语言
...
gcc -x c++ cpp.c -lstdc++ -o cpp
代码：cpp.c
-O0/O1/O2/O3: 指定优化等级，O0不优化，缺省O1优化

2.5 头文件

1)头文件里写什么?

头文件卫士(防止重定义,重声明)

```
#ifndef __XXX_
#define __XXX_
...
#endif
```

```
     a.h
     /  \
   b.h  c.h
     \  /
      d.c
```

包含其它头文件

宏定义

```
#define PI 3.14159
```

自定义类型

```
struct Circle {
    double x, y, r;
};
```

类型别名

```
typedef struct Circle C;
```

外部变量声明

```
extern double e;
```

函数声明

```
double circleArea(C c);
```

重定义

```
   a.h
  /   \
b.c    c.c
 |      |
b.o    c.o
  \    /
    d
```

一个头文件可能会被多个源文件包含，写在头文件里的函数定义也会因此被预处理器扩展到多个包含该头文件的源文件中，并在编译阶段被编译到等多个不同的目标文件中，这将导致链接错误：multiple definition，多重定义。

2)去哪里找头文件?

gcc -I<头文件的附加搜索路径>

#include <my.h>

先找-I指定的目录，再找系统目录。

#include "my.h"
先找-I指定的目录，再找当前目录，最后找系统目录。
头文件的系统目录:
/usr/include - 标准C库
/usr/local/include - 第三方库
/usr/lib/gcc/i686-linux-gnu/5.4.0/include - 编译器库

2.6 预处理指令

#include - 将指定的文件内容插至此指令处
#define - 定义宏
#undef - 删除宏
#if - 如果
#ifdef - 如果宏已定义
#ifndef - 如果宏未定义
#else - 否则，与#if/#ifdef/#ifndef配合使用
#elif - 否则如果，与#if/#ifdef/#ifndef配合使用
#endif - 结束判定，与#if/#ifdef/#ifndef配合使用
#error - 产生错误，结束预处理
#warning - 产生警告，继续预处理
#line - 指定行号
代码：line.c

```c
#include <stdio.h>
int main(void) {
    int x = 1000;
    printf("%d\n", __LINE__);
#line 100
    printf("%d\n", __LINE__);
    return 0;
}
```

#pragma - 设定编译器的状态或者指示编译器的操作
#pragma GCC dependency 被依赖文件。
#pragma GCC poison 语法禁忌　**如#pragma GCC poison goto,意思为禁用goto**。
#pragma pack(按几字节对齐：1/2/4/8)
#pragma pack() - 按缺省字节数对齐　**默认对齐数为4**

2.7 预定义宏

无需自行定义，预处理器会根据事先设定好的规则将这些宏扩展成其对应的值。

```
__BASE_FILE__: 正在被处理的源文件名
__FILE__: 所在文件名
__LINE__: 所在行的行号
__FUNCTION__: 所在函数的函数名
__func__: 同__FUNCTION__
__DATE__: 处理日期
__TIME__: 处理时间
__INCLUDE_LEVEL__: 包含层数，从0开始
__cplusplus: C++有定义，C无定义
```

2.8 环境变量

在进程向下文中保存的一些数据：键(功能，是什么)=值(内容)。
env
C_INCLUDE_PATH
C语言头文件的附加搜索路径，相当于-I选项。
CPATH
同C_INCLUDE_PATH
CPLUS_INCLUDE_PATH
C++语言头文件的附加搜索路径，相当于-I选项。
LIBRARY_PATH
链接库路径
LD_LIBRARY_PATH
加载库路径
#include "/.../.../xxx.h"　**移植性差**
#include "xxx.h"
gcc -I/.../... ... - 推荐
C_INCLUDE_PATH/CPATH=/.../...:/...　**易冲突**

三、库

a.c -> a.out
foo()
bar()
hum()
main()
单一模型：将程序中所有功能全部实现于一个单一的源文件内部。编译时间长，不易于维护和升级，不易于协发。
分离模型：将程序中的不同功能模块划分到不同的源文件中。缩短编译时间，易于维护和升级，易于协作开发。
a.c　-> a.o \
foo()　　| -> ...
bar()　　|
b.c　-> b.o /
hum()
a.o \
b.o　| -> 库 + 其它模块 -> ...
c.o　|
...　/

3.1 静态库

静态库的本质就是将多个目标文件打包成一个文件。
链接静态库就是将库中被调用的代码复制到调用模块中。
使用静态库的程序通常会占用较大的空间，库中代码一旦修改，所有使用该库的程序必须重新链接。
使用静态库的程序在运行无需依赖库，其执行效率高。
静态库的形式：libxxx.a
构建静态库：
.c -> .o -> .a
ar -r libxxx.a x.o y.o z.o
　　^　__/

|__|

使用静态库：

gcc ... -lxxx -L<库路径>

export LIBRARY_PATH=<库路径>

gcc ... -lxxx