[Kasa camera 4](#_Toc56429461)

[cloud 4](#_Toc56429462)

[Summary 4](#_Toc56429463)

[设置vim缩进 4](#_Toc56429464)

[samba共享设置 4](#_Toc56429465)

[git操作 5](#_Toc56429466)

[OpenWrt 6](#_Toc56429467)

[shell脚本问题 13](#_Toc56429468)

[加环境变量 13](#_Toc56429469)

[ubus相关 13](#_Toc56429470)

[dbg输出写法 14](#_Toc56429471)

[Vim操作 14](#_Toc56429472)

[查看内存CPU 15](#_Toc56429473)

[命令别名设定 15](#_Toc56429474)

[浮点型值比较 16](#_Toc56429475)

[sizeof 17](#_Toc56429476)

[goto 17](#_Toc56429477)

[malloc 17](#_Toc56429478)

[C语言内存分配方式 18](#_Toc56429479)

[scanf接收& 18](#_Toc56429480)

[union联合体、共用体 18](#_Toc56429481)

[EOF 19](#_Toc56429482)

[无缓冲io(Unbuffered I/O)函数 19](#_Toc56429483)

[select 19](#_Toc56429484)

[非阻塞读 fcntl 20](#_Toc56429485)

[硬链接 符号链接 20](#_Toc56429486)

[进程控制块PCB 21](#_Toc56429487)

[fork exec 21](#_Toc56429488)

[wait或waitpid 24](#_Toc56429489)

[进程间通信 管道 25](#_Toc56429490)

[shell 信号 SIGINT等 26](#_Toc56429491)

[信号 27](#_Toc56429492)

[可重入函数 31](#_Toc56429493)

[关于SIGCHLD信号 32](#_Toc56429494)

[终端 32](#_Toc56429495)

[线程介绍 33](#_Toc56429496)

[pthread线程库函数 34](#_Toc56429497)

[线程mutex 35](#_Toc56429498)

[条件变量（Condition Variable） 37](#_Toc56429499)

[Semaphore信号量 38](#_Toc56429500)

[静态static局部变量 静态全局变量 外部extern变量 39](#_Toc56429501)

[C 语言整数与字符串的相互转换 40](#_Toc56429502)

[makefile cmake 40](#_Toc56429503)

[C++静态库与动态库 42](#_Toc56429504)

[makefile的编写指导 52](#_Toc56429505)

[Makefile的规则 52](#_Toc56429506)

[通用makefile模板 53](#_Toc56429507)

[make的工作流程 54](#_Toc56429508)

[makefile中使用变量$(objects) 55](#_Toc56429509)

[make自动推导 55](#_Toc56429510)

[清空目标文件的规则 56](#_Toc56429511)

[makefile的组成 57](#_Toc56429512)

[引用其他的makefile 57](#_Toc56429513)

[make的工作方式 58](#_Toc56429514)

[makefile规则中使用通配符 59](#_Toc56429515)

[makefile文件搜寻 59](#_Toc56429516)

[伪目标 60](#_Toc56429517)

[多目标 62](#_Toc56429518)

[静态模式 63](#_Toc56429519)

[自动生成依赖性 64](#_Toc56429520)

[sed函数 65](#_Toc56429521)

[makefile显示命令 66](#_Toc56429522)

[makefile命令执行 66](#_Toc56429523)

[makefile命令出错 67](#_Toc56429524)

[嵌套执行make 67](#_Toc56429525)

[makefile使用变量 68](#_Toc56429526)

[makefile := 69](#_Toc56429527)

[makefile ?= 70](#_Toc56429528)

[makefile变量的替换 70](#_Toc56429529)

[makefile追加变量值 71](#_Toc56429530)

[makefile多行命令 71](#_Toc56429531)

[makefile define关键字 71](#_Toc56429532)

[makefile环境变量 72](#_Toc56429533)

[makefile目标变量 72](#_Toc56429534)

[makefile模式变量 73](#_Toc56429535)

[makefile使用条件判断 73](#_Toc56429536)

[makefile 使用函数 75](#_Toc56429537)

[makefile字符串处理函数 76](#_Toc56429538)

[makefile文件名操作函数 79](#_Toc56429539)

[makefile foreach函数 80](#_Toc56429540)

[makefile call函数 81](#_Toc56429541)

[makefile origin函数 82](#_Toc56429542)

[makefile之override 82](#_Toc56429543)

[makefile shell函数 83](#_Toc56429544)

[makefile隐含规则 86](#_Toc56429545)

[makefile隐含规则一览 87](#_Toc56429546)

[隐含规则使用的变量 89](#_Toc56429547)

[隐含规则链 90](#_Toc56429548)

[定义模式规则 90](#_Toc56429549)

[自动化变量 91](#_Toc56429550)

[模式的匹配 94](#_Toc56429551)

[makefile命令参数 94](#_Toc56429552)

[linux下的c编程 94](#_Toc56429553)

[gcc的可选参数 95](#_Toc56429554)

[cpp 98](#_Toc56429555)

[linux编译 -std=c++11 98](#_Toc56429556)

[namespace命名空间 98](#_Toc56429557)

[类型长度与选用 99](#_Toc56429558)

[引用 100](#_Toc56429559)

[指针值 100](#_Toc56429560)

[void\* 指针 101](#_Toc56429561)

[const指针 103](#_Toc56429562)

[顶层const 底层const 104](#_Toc56429563)

[auto 105](#_Toc56429564)

[decltype 105](#_Toc56429565)

[预处理 105](#_Toc56429566)

[头文件保护符（header guard） 106](#_Toc56429567)

[C++ vector 容器浅析 106](#_Toc56429568)

## Kasa camera

### cloud

Kasa camera的cloud相关命令涉及到基础云和业务云，基础云主要与账号相关，业务云则主要与具体的业务相关

firm 商号

snapshot 快照

relay 中继

siren 警报

firmware 固件

### ubus

ubus实现进程间通信的三种方式：

1 invoke的方式实现端对端通信

2 subscribe/notify的方式实现订阅

3 event的方式实现事件通知

event机制从一对一的进程之间通信来讲，和invoke机制类似。不过event机制中，发送方不需要知道谁要接收这个消息，实际上就是一个广播消息。这类似于U盘的热插拔：当插上或拔出U盘时，内核会广播一个NETLINK事件，之后内核继续做自己的事情，而不关心谁会去监听和处理这个事件。

客户端监听事件：

1. static int client\_ubus\_register\_events()
2. {
3. static struct ubus\_event\_handler listener;
4. int ret = 0;
6. */\* 注册特定event的listener。多个event可以使用同一个listener \*/*
7. memset(&listener, 0, sizeof(listener));
8. listener.cb = ubus\_probe\_device\_event;
10. ret = ubus\_register\_event\_handler(ctx, &listener, UBUS\_EVENT\_ADD\_DEVICE);
11. if (ret)
12. {
13. fprintf(stderr, "register event failed.\n");
14. return -1;
15. }
17. ret = ubus\_register\_event\_handler(ctx, &listener, UBUS\_EVENT\_REMOVE\_DEVICE);
18. if (ret)
19. {
20. ubus\_unregister\_event\_handler(ctx, &listener);
21. fprintf(stderr, "register event failed.\n");
22. return -1;
23. }
25. return 0;
26. }

服务端触发事件：

1. static int server\_ubus\_send\_event(void)
2. {
3. blob\_buf\_init(&b, 0);
5. */\* 需要传递的参数 \*/*
6. blobmsg\_add\_u32(&b, "major", 3);
7. blobmsg\_add\_u32(&b, "minor", 56);
8. blobmsg\_add\_string(&b, "name", "mmc01");
10. */\* 广播名为"add\_device"的事件 \*/*
11. return ubus\_send\_event(ctx, "add\_device", b.head);
12. }

先运行client &注册事件。我们同时启动两个client程序。   
再执行server主动触发"add\_device"事件，则凡是注册了这个事件的client都会收到该事件并执行各自的处理。

root@NVR:~*# ./server*

connected as fdecbdc1

{ "add\_device": { "major": 3, "minor": 56, "name": "mmc01" } }

{ "add\_device": { "major": 3, "minor": 56, "name": "mmc01" } }

也可以通过命令行的ubus send命令触发事件：

root@NVR:~# ubus send "rm\_device" '{ "major": 3, "minor": 56, "name": "mmc01" }'

{ "rm\_device": { "major": 3, "minor": 56, "name": "mmc01" } }

{ "rm\_device": { "major": 3, "minor": 56, "name": "mmc01" } }

在使用ubus时，可根据实际的场景来选择用哪种方式进行进程间通信。如之前所说，ubus是为发送消息而设计的，不合适传输大量数据。

转：<https://blog.csdn.net/jasonchen_gbd/article/details/46055885>

1.简介

　　openWRT提供了一个系统总线ubus，提供系统级的进程间通信（IPC）功能。

2.ubus命令使用说明

　　ubus list [-v] 该命令用于显示当前ubus中注册的接口，其中-v参数用以显示各个接口的详细信息。

　　ubus call 该命令用于调用ubus中当前注册的接口。

　　bus listen 用于监听ubus相关事件，如果不指定事件名则监听所有事件。

　　ubus send 用于发送事件。

　　ubus wait\_for 用于等待指定项的注册到ubus中。

3.实现框架

　　ubus实现的基础是unix socket，建立一个unix socket服务器和客户端一般需要4步：creat server bind socket -> creat client to connect -> send msg -> handle msg.

　　ubus实现上这个框架并对msg传输和处理进行了封装：

　　1.ubus提供了一个socket server：ubusd

　　2. ubus提供了创建socket client端的接口(提供了C、Shell、LUA三种现成的客户端)

　　3.消息必须封装位json格式

　　4.Client对消息的处理为{对象}{方法}的结构，发送请求方只需在消息中指定要调用的对象和方法的名字即可

　　使用ubus时需要引用一些动态库，主要包括：

　　　　1.libubus.so（例如创建socket，进行监听和连接，发送消息等接口函数）

　　　　2.libubox.so（ubus向外部提供的编程接口，例如等待和读取消息）

　　　　3.libblobmsg.so（提供了封装和解析json数据的接口）

4.实现原理

　　先举个例子：

1. client1向ubusd注册了两个对象：“interface”和“dotalk”，其中“interface”对象中注册了两个method：“getlanip”和“setlanip”，对应的处理函数分别为func1()和func2()。“dotalk”对象中注册了两个method：“sayhi”和“saybye”，对应的处理函数分别为func3()和func4()。

2. 接着创建一个client2用来与client1通信，注意，两个client之间不能直接通信，需要经ubusd（server）中转。

3. client2就是在前面讲到的shell/lua/C客户端。假设这里使用shell客户端，在终端输入以下命令：

ubus call interface setlanip ‘{“ip”:“10.0.0.1”, “mask”:24}’

ubus的call命令带三个参数：请求的对象名，需要调用的方法名，要传给方法的参数。

4. 消息发到server后，server根据对象名找到应该将请求转发给client1，然后将消息发送到client1，client1进而调用func2()接受参数并处理，如果处理完成后需要回复client2，则发送回复消息。

5.ubus的应用场景和局限性

　　ubus可用于两个进程之间的通信，并以类似json格式进行数据交互。ubus的常见场景为：

　　　　1.“客户端--服务器”形式的交互，即进程A注册一系列的服务，进程B去调用这些服务。

　　　　2.ubus支持以“订阅 -- 通知”的方式进行进程通信，即进程A提供订阅服务，其他进程可以选择订阅或退订该服务，进程A可以向所有订阅者发送消息。

　　由于ubus实现方式的限制，在一些场景中不适宜使用ubus：

　　　　1.ubus用于少量数据的传输，如果数据量很大或是数据交互很频繁，则不宜用ubus。经过测试，当ubus一次传输数据量超过60KB，就不能正常工作了。

　　　　2.ubus对多线程支持的不好，例如在多个线程中去请求同一个服务，就有可能出现不可预知的结果。

　　　　3.不建议递归调用ubus，例如进程A去调用进程B的服务，而B的该服务需要调用进程C的服务，之后C将结果返回给B，然后B将结果返回给A。如果不得不这样做，需要在调用过程中避免全局变量的重用问题。

### uci

1.UCI命令

一个众所周知的原因，在Linux下各种软件包有各种不同的配置脚本，每个配置脚本的语法格式和操作方式不同， 这样的设计虽然可以体现出各软件包自身的优势，同时也增加了学习曲线。在这一点上OpenWrt的UCI无疑处理的更

胜一筹。UCI是集中式配置信息管理接口(Unified Configuration Interface)的缩写，他是OpenWrt引进的一套配置 参数管理系统。UCI管理了OpenWrt下最主要的系统配置参数并且提供了简单、容易、标准化的人机交互接口。UCI中 已经包含了网络配置、无线配置、系统信息配置等作为基本路由器所需的主要配置参数。同时UCI也可以帮助开发人员快速的建立一套基于OpenWrt的智能路由产品控制界面。

2.UCI的文件和流程   
  
UCI的配置文件全部存储在/etc/config目录下。

1. root@OpenWrt:/# ls /etc/config/
2. dhcp            dropbear    firewall    network      system        wireless

日前已有大量软件包支持UCI模式管理，但不是所有的软件包，支持的软件包是这样来完成   
启动的(以samba举例):   
1.启动脚本/etc/init.d/samba   
2.启动脚本通过UCI分析库从/etc/config/samba获得启动参数   
3.启动脚本完成正常启动   
  
    由于UCI的数据文件较为简单，并且具备了很nice的直接观感，所以配置文件既可以使用UCI命令进行修改，也可以使用VI编辑器直接修改文件。但如果两种方式都是用时需要注意UCI命令修改会产生缓存，每次修改好要尽快确认保存避免出现冲突。   
  最常见的几个UCI配置作用说明

|  |  |
| --- | --- |
| 文件 | 作用 |
| /etc/config/dhcp | 面向LAN口提供的IP地址分配服务配置 |
| /etc/config/dropbear | SSH服务配置 |
| /etc/config/firewall | 路由转发，端口转发，防火墙规则 |
| /etc/config/network | 自身网络接口配置 |
| /etc/config/system | 时间服务器时区配置 |
| /etc/config/wireless | 无线网络配置 |

3.UCI的文件语法   
UCI文件语法举例

1. config 'section-type' 'section'
2. option  'key'       'value'
3. list    'list\_key'  'list\_value'
4. config 'example' 'test'
5. option  'string'        'some value'
6. option  'boolean'       '1'
7. list    'collection'    'first item'
8. list    'collection'    'second item'

config 节点 以关键字 config 开始的一行用来代表当前节点   
            section-type 节点类型   
            section 节点名称   
option 选项 表示节点中的一个元素   
            key 键   
            value 值   
list 列表选项 表示列表形式的一组参数。   
           list\_key 列表键   
           list\_value 列表值   
  
config 节点语法格式

1. config 'section-type' 'section'

config 节点(后文统一称为节点)原则   
           UCI 允许只有节点类型的匿名节点存在   
           节点类型和名字建议使用单引号包含以免引起歧义   
           节点中可以包含多个 option 选项或 list 列表选项。   
           节点遇到文件结束或遇到下一个节点代表完成。   
**option 选项语法格式**

1. option 'key' 'value'

option 选项(后文统一称为选项)原则   
           选项的键与值建议使用单引号包含   
           避免相同的选项键存在于同一个节点,否则只有一个生效   
**list 列表选项语法格式**

1. list 'list\_key' 'list\_value'

list 列表选项(后文统一称为列表)原则   
      选项的键与值建议使用单引号包含   
      列表键的名字如果相同,则相同键的值将会被当作数组传递给相应软件   
**UCI 的语法容错**

1. option example    value
2. option 'example'   value
3. option example    "value"
4. option "example"  'value'
5. option 'example'   "value"

**UCI 无法容忍的语法**

1. option 'example" "value'
2. option example some value with space

尽量使用常规字符去处理器 UCI,特殊字符有可能会破坏数据结构的完整性。

读取类语法   
取得节点类型

1. uci get <config>.<section>

取得一个值

1. uci get <config>.<section>.<option>

显示全部 UCI 配置

1. uci show

显示指定文件配置

1. uci show <config>

显示指定节点名字配置

1. uci show <config>.<section>

显示指定选项配置

1. uci show <config>.<section>.<option>

显示尚未生效的修改记录

1. uci changes <config>

匿名节点显示(如果所显示内容有匿名节点,使用-X 参数可以显示出匿名节点的 ID)

1. uci show -X <config>.<section>.<option>

**写入类语法**   
增加一个匿名节点到文件

1. uci add <config> <section-type>

增加一个节点到文件中

1. uci set <config>.<section>=<section-type>

增加一个选项和值到节点中

1. uci set <config>.<section>.<option>=<value>

增加一个值到列表中

1. uci add\_list <config>.<section>.<option>=<value>

修改一个节点的类型

1. uci set <config>.<section>=<section-type>

修改一个选项的值

1. uci set <config>.<section>.<option>=<value>

删除指定名字的节点

1. uci delete <config>.<section>

删除指定选项

1. uci delete <config>.<section>.<option>

删除列表

1. uci delete <config>.<section>.<list>

删除列表中一个值

1. uci del\_list <config>.<section>.<option>=<string>

生效修改(任何写入类的语法,最终都要执行生效修改,否则所做修改只在缓存中,切记!)

1. uci commit <config>

### IPC Algorithm

SSD，即Sum of Squared Differences，就是估算值与估算对象之差值的平方和。一般又称为 Mean Squared Error。

SAD，即Sum of Absolute Differences，就是差的绝对值的和。此专算法常用于图属像块匹配，将每个像素对应数值之差的绝对值求和，据此评估两个图像块的相似度。可以看出这个算法很快速、但并不精确，通常用于多级处理的初步筛选。

运动检测：

视频流的创建：额外视频通道用于移动侦测。

常见算法：帧差法、背景建模法、光流法等

### 音视频编码

分辨率：用以描述图像、视频的分辨能力，通常用各个方向上像素个数来表示，如640x480。

帧率：视频每秒钟的静态图像画面数量。

常见音频格式：

PCM：脉冲编码调制，无损压缩，简单取样量化，将模拟信号离散数值化。

G711：ITU-T定制的语音压缩标准，支持广泛，适用于综合业务网和大多数字电话链路，有A律（欧规）和μ律（美规）的区分, 64kbps／channel

G726：ITU-T标准，相较于G711语音质量更高，同级质量下压缩比更高

AAC：高级音频编码,mp4文件格式中应用最广泛的音频编码

音频优化算法：

AEC（Acoustic Echo Cancellation）：回声消除。在一些特定场景下，如IPC对讲时，本地MIC采集的语音有可能包含AO设备播放的声音，则需要应用此项技术。

ANR（Audio Noise Reducation）：语音降噪。用于去除外界环境噪声，保留语音数据。

其他如自动增益、高通滤波、均衡器、高动态等音频质量增强的算法。

帧率：每秒视频图像数量。

FPS （每秒传输帧数(Frames Per Second)）：FPS是图像领域中的定义，是指画面每秒传输帧数，通俗来讲就是指动画或视频的画面数。FPS是测量用于保存、显示动态视频的信息数量。每秒钟帧数越多，所显示的动作就会越流畅。通常，要避免动作不流畅的最低是30。某些计算机视频格式，每秒只能提供15帧。

用中文表达就是多少帧每秒，或每秒多少帧。电影是24fps，通常简称为24帧。

码率：即比特率，每秒视频传送的比特数。

码率计算公式：

码率=采样率 x 位深度 x 声道

所以，上面文件的码率= 44.1Khz x 16位 x 2声道 = 1411.2 Kbps

### HISI

VPSS（Video Process Sub-System）视频处理子系统，支持对一幅输入图像进行统一预处理，如去噪、去隔行等，然后再对各通道分别进行缩放、锐化等处理，最后输出多种不同分辨率的图像。

VPSS 单元支持的具体图像处理功能包括 FRC（Frame Rate Control）、 Crop、 NR（Noise Reduce）、 LDC（Lens Distortion Correction）、 Rotate、 Cover/Overlay、 Scale、Mirror/Flip、 FishEye 等。

### H.264

H.264：又叫AVC，是ITU和ISO共同提出的视频压缩格式。相较于之前的MPEG4等，在同等图像质量下码率较低，数据压缩率非常高。

H.264概念

在H.264/AVC视频编码标准中，整个系统框架被分为了两个层面：视频编码层面（VCL）和网络抽象层面（NAL）。其中，前者负责有效表示视频数据的内容，而后者则负责格式化数据并提供头信息，以保证数据适合各种信道和存储介质上的传播。VCL是H.264/AVC的规格，意思是压缩后、去冗余（Visual Redundancy）的视频信息，其技术核心包括动作估计、转换编码、预测编码、去区块效应滤波、及熵编码等。

NALU:：<https://blog.csdn.net/yanghangwww/article/details/103676530?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-3.edu_weight&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-3.edu_weight>

I帧：关键参考帧，或叫帧内编码帧，仅帧内编码，解码时仅需要本帧数据

P帧：前向预测编码帧，内含该帧与前帧的差别，解码时需要之前的I帧或I帧之后缓存的画面

B帧：双向预测内插编码帧，记录了本帧与前后帧的差别，解码时需要前面的I或P帧及后面的P帧作为参考，B帧压缩率最高，但相应需要的解码能力也越高

GOP：画面组长度，即I帧间隔，比如GOP为120，则每120帧有1个I帧，如果此时帧率为30，则每4秒有1个I帧

NALU(Network Abstraction Layer Units)网络抽象层:

每个NAL单元是一个一定语法元素的可变长字节字符串，包括一个字节的头信息(用来表示数据类型)，

以及若干整数字节的原始字节序列负荷(RBSP)。

一个NAL单元可以携带一个编码片，I帧、P帧、B帧、一个序列参数集、或一个图像参数集。

H264采用NAL单元可以适用于多种网络，而且进一步提高其抗误码能力。

序列号的设置可以发现丢失的是哪一个VLC单元，

冗余编码图像使得基本编码图像丢失仍可得到较粗糙的图像。

NALU = NAL header(1 byte) + RBSP

SPS(Sequence Paramater Set)序列参数集：

SPS中保存了一组视频编码序列(Codec Video Sequence)的全局参数，

序列中每一帧编码后的数据所依赖的参数保存于图像参数集中，

一般情况下SPS和PPS的NAL单元通常位与整个码流的起始位置，但是在某些特殊情况下，

在码流中间也可能出现这两种结构，主要的原因可能为：

\*解码器需要在码流中间开始解码；

\*编码器在编码的过程中改变了码流的参数(如图像的分辨率)；

PPS(Picture Paramater Set)图像参数集：

PPS类似于SPS，在H264的码流中单独保存在一个NAL单元中，

只是PPS NAL Unit的nal\_unit\_type值为8，

而在封装格式中，PPS通常与SPS一起，保存在视频文件的文件头中。

SEI辅助增强信息

SEI是H264标准中一个重要的技术，主要起补充和增强的作用。

SEI没有图像数据信息，只是对图像数据信息或者视频流的补充，

有些内容可能对解码有帮助.

## Summary

### 设置vim缩进

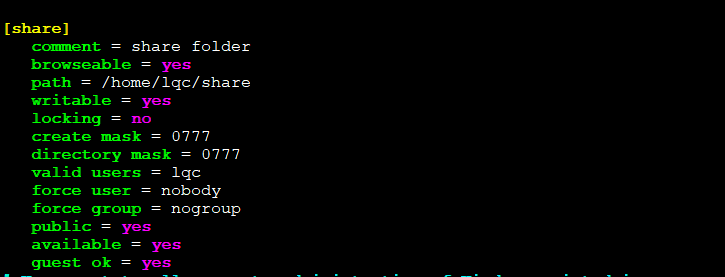
Linux vim /etc/vim/vimrc 配置文件 set ts=4 “=”两边不能有空格

### samba共享设置

samba文件共享文件保存权限设置

vim /etc/samba/

sudo vim smb.conf



service smbd restart

chmod -R 0777 /home/lqc/share/

### git操作

Git删除文件

git rm –r –cache makefile\_folder

git commit –m “del …”

git push

git删除仓库文件

git rm xxx

git commit -m “xxx”

git push origin main

makefile：

删除执行文件和所有的中间文件，执行：make clean

回车键：

Linu下：\n

Windows下：\r\n

if (0 == strcmp(buf\_update, "\r\n"))

{

continue;

}

### OpenWrt

OpenWrt helloworld

make package/symlinks

make menuconfig

make V=s

make package/ulinkied/compile V=s

make package/arp\_demo/compile V=s

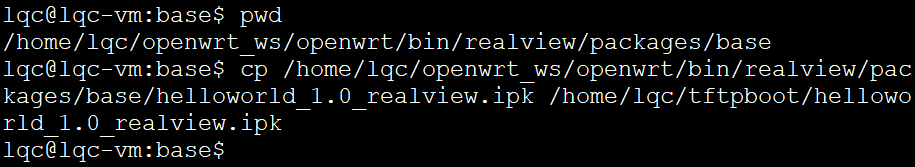
make package/ubus\_demo/compile V=s

make package/ubus\_client/compile V=s

make package/arpsd/compile V=s

make package/test\_arpsd/compile V=s

.ipk文件保存在如下路径：



命令如下：

cp /home/lqc/openwrt\_ws/openwrt/bin/realview/packages/base/helloworld\_1.0\_realview.ipk /home/lqc/tftpboot/helloworld\_1.0\_realview.ipk

cp /home/lqc/openwrt\_ws/openwrt/bin/realview/packages/private/demo\_1.0\_realview.ipk /home/lqc/tftpboot/demo\_1.0\_realview.ipk

cp /home/lqc/openwrt\_ws/openwrt/bin/realview/packages/private/ulinkied\_1.0-1\_realview.ipk /home/lqc/tftpboot/ulinkied\_1.0-1\_realview.ipk

cp /home/lqc/openwrt\_ws/openwrt/bin/realview/packages/private/ubus\_demo\_1.0-1\_realview.ipk /home/lqc/tftpboot/ubus\_demo\_1.0-1\_realview.ipk

cp /home/lqc/openwrt\_ws/openwrt/bin/realview/packages/private/ubus\_client\_1.0-1\_realview.ipk /home/lqc/tftpboot/ubus\_client\_1.0-1\_realview.ipk

cp /home/lqc/openwrt\_ws/openwrt/bin/realview/packages/private/arp\_demo\_1.0-1\_realview.ipk /home/lqc/tftpboot/arp\_demo\_1.0-1\_realview.ipk

cp /home/lqc/openwrt\_ws/openwrt/bin/realview/packages/private/arpsd\_1.0-1\_realview.ipk /home/lqc/tftpboot/arpsd\_1.0-1\_realview.ipk

cp /home/lqc/openwrt\_ws/openwrt/bin/realview/packages/private/test\_arpsd\_1.0-1\_realview.ipk /home/lqc/tftpboot/test\_arpsd\_1.0-1\_realview.ipk

tftp -gr helloworld\_1.0\_realview.ipk 192.168.10.1

tftp -gr ulinkied\_1.0-1\_realview.ipk 192.168.10.1

tftp -gr ubus\_demo\_1.0-1\_realview.ipk 192.168.10.1

tftp -gr ubus\_client\_1.0-1\_realview.ipk 192.168.10.1

tftp -gr arp\_demo\_1.0-1\_realview.ipk 192.168.10.1

tftp -gr arpsd\_1.0-1\_realview.ipk 192.168.10.1

tftp -gr test\_arpsd\_1.0-1\_realview.ipk 192.168.10.1

mount -t tmpfs none /overlay/

opkg install ulinkied\_1.0\_realview.ipk

opkg install ubus\_demo\_1.0\_realview.ipk

opkg install ubus\_client\_1.0\_realview.ipk

opkg install arpsd\_1.0\_realview.ipk

opkg install test\_arpsd\_1.0\_realview.ipk

arp table 命令：

获取扫描结果：

ubus call arp get\_scan\_result

清除扫描结果：

ubus call arp clear\_result

获取扫描状态：

ubus call arp get\_scan\_status

停止扫描：

ubus call arp stop\_scan

开始扫描：

ubus call arp start\_scan

重载参数：

ubus call arp reload\_config

带参数的命令：

ubus call arp get\_scan\_result '{"request\_type":1}'

helloworld

若出现：

root@OpenWrt:/# opkg install helloworld\_1.0\_realview.ipk

Installing helloworld (1.0) to root...

Collected errors:

\* verify\_pkg\_installable: Only have 0kb available on filesystem /overlay, pkg helloworld needs 1

\* opkg\_install\_cmd: Cannot install package helloworld.

执行如下命令：

root@OpenWrt:/# mount -t tmpfs none /overlay/

tftp命令：

root@OpenWrt:/# tftp -gr helloworld\_1.0\_realview.ipk 192.168.10.1

openwrt无tftp命令：

重新编译

cp realview.config .config

make V=s

uci：

uci show

uci get linkie.led\_status.enable

uci set linkie.led\_status.enable=0

修改network配置文件，然后调用ubus call network reload就可以使其生效了。

ubus call network reload

请按以下步骤搭建好OpenWrt开发平台。

**1、代码下载：**

git clone [git@ntrscm.rd.tp-link.net:/openwrt/openwrt.git](mailto:git@ntrscm.rd.tp-link.net:/openwrt/openwrt.git)

git checkout chaos\_calmer

git cherry-pick -n 70b104f98c0657323b28fce140b73a94bf3eb756

cd openwrt

git clone [git@ntrscm.rd.tp-link.net:/openwrt/dl.git](mailto:git@ntrscm.rd.tp-link.net:/openwrt/dl.git)

mkdir bin/realview -p

ln -sf ../../dl/OpenWrt-Toolchain-realview\_gcc-5.3.0\_musl-1.1.16\_eabi.Linux-i686.tar.bz2 bin/realview/OpenWrt-Toolchain-realview\_gcc-5.3.0\_musl-1.1.16\_eabi.Linux-i686.tar.bz2

ln -sf ../../dl/OpenWrt-Toolchain-realview\_gcc-4.8-linaro\_uClibc-0.9.33.2\_eabi.Linux-i686.tar.bz2 bin/realview/OpenWrt-Toolchain-realview\_gcc-4.8-linaro\_uClibc-0.9.33.2\_eabi.Linux-i686.tar.bz2

**2、配置软件源：**

echo 'src-git packages [git@ntrscm.rd.tp-link.net:/openwrt/packages.git^d360d159b4a1ef81e81e883c2373bb59c6b84d6c](mailto:git@ntrscm.rd.tp-link.net:/openwrt/packages.git%5ed360d159b4a1ef81e81e883c2373bb59c6b84d6c)' > feeds.conf

echo 'src-git luci [git@ntrscm.rd.tp-link.net:/openwrt/luci.git^e58dae21f265a468689d29a08f4b974d0a4cc682](mailto:git@ntrscm.rd.tp-link.net:/openwrt/luci.git%5ee58dae21f265a468689d29a08f4b974d0a4cc682)' >> feeds.conf

echo 'src-git routing [git@ntrscm.rd.tp-link.net:/openwrt/routing.git^3cc8d49900fafbe76f44e0e912ee888fafd8bbb4](mailto:git@ntrscm.rd.tp-link.net:/openwrt/routing.git%5e3cc8d49900fafbe76f44e0e912ee888fafd8bbb4)' >> feeds.conf

echo 'src-git telephony [git@ntrscm.rd.tp-link.net:/openwrt/telephony.git^5022cd6c2c9102c6c0b974701ad8052f5791eed2](mailto:git@ntrscm.rd.tp-link.net:/openwrt/telephony.git%5e5022cd6c2c9102c6c0b974701ad8052f5791eed2)' >> feeds.conf

echo 'src-git management [git@ntrscm.rd.tp-link.net:/openwrt/management.git^7ad87afe5d74b1e569872c68e6c9317952a66682](mailto:git@ntrscm.rd.tp-link.net:/openwrt/management.git%5e7ad87afe5d74b1e569872c68e6c9317952a66682)' >> feeds.conf

echo 'src-git targets [git@ntrscm.rd.tp-link.net:/openwrt/targets.git^a977f6ab0f2f1cbdc8ee6bdca08ebc86980e4350](mailto:git@ntrscm.rd.tp-link.net:/openwrt/targets.git%5ea977f6ab0f2f1cbdc8ee6bdca08ebc86980e4350)' >> feeds.conf

echo 'src-link private ../private' >> feeds.conf

make package/symlinks

**3、配置软件包并编译（邮件附件中的realview.config）：**

cp realview.config .config

make -j 4 V=w

**4、模拟器安装和使用：**

安装：sudo apt-get install qemu

运行：sudo qemu-system-arm -M realview-pbx-a9 -net nic -net tap,ifname=tap0 -nographic -kernel bin/realview/openwrt-realview-vmlinux-initramfs.elf

注意：上面命令中指定了tap接口名tap0，因此同时只能运行一个OpenWrt，如果去掉ifname参数，则可以运行多个但接口名字会变成tap1、tap2……

退出：先按Ctrl+A组合键，再按X即退出

**5、网络设置：**

设置OpenWrt的lan接口IP：ifconfig br-lan 192.168.10.2

设置主机tap0的接口IP：sudo ifconfig tap0 192.168.10.1

测试网络连通：ping 192.168.10.2

**6、TFTP服务器安装和配置：**

安装TFTP服务器：sudo apt-get install tftpd

查看TFTP配置：sudo grep tftpd /etc/inetd.conf

**7、在Openwrt上面用tftp下载tdb程序（见邮件附件）：**

tftp -gr tdb -l /usr/bin 192.168.10.1

chmod +x /usr/bin/tdb

**8、为方便Windows主机访问，可以配置Bridge模式来桥接tap0和eth0：**

 8.1、 安装bridge配置工具：sudo apt-get install bridge-utils

 8.2、禁用NetworkManager配置，从而使用/etc/network/interfaces配置

sudo vim /etc/NetworkManager/NetworkManager.conf



8.3、修改/etc/network/interfaces配置

         1) 备份旧配置sudo cp /etc/network/interfaces /etc/network/interfaces.bak

         2) 在/etc/network/interfaces中添加bridge相关配置

|  |
| --- |
| # interfaces(5) file used by ifup(8) and ifdown(8)  auto lo  iface lo inet loopback    auto br0  iface br0 inet static          address 192.168.137.164          netmask 255.255.255.0          broadcast 192.168.137.255          gateway 192.168.137.1          dns-nameservers 172.31.1.1 172.31.1.2          bridge\_ports eth0 tap0          bridge\_stp off          bridge\_fd 0          bridge\_maxwait 0    auto br0:0  iface br0:0 inet static          address 192.168.0.164          netmask 255.255.255.0    auto br0:1  iface br0:1 inet static          address 192.168.1.164          netmask 255.255.255.0 |

8.4、重启ubuntu即可生效，后续每次运行模拟机，创建tap0后自己会加到br0中。

8.5、查看bridge是否成功（eth0和tap0接口）：brctl show

bridge name    bridge id           STP enabled     interfaces

br0            8000.000xxxxxxxx             no              eth0

                                                          tap0

强制杀死进程

kill –KILL 15344

### shell脚本问题

Shell:不同平台格式问题转换

用vi

set fileformat=unix

#!/bin/bash

### 加环境变量

lqc@lqc-vm:venc$ export PATH=$PATH:/opt/hisi-linux/x86-arm/arm-himix100-linux/bin

增加环境变量：

若该变量为扩增变量内容时，则可用 "$变量名称" 或 ${变量} 累加内容，如下所示：

『PATH="$PATH":/home/bin』或『PATH=${PATH}:/home/bin』

按照惯例，环境变量字符串都是name=value这样的形式，大多数name由大写字母加下划线组成，一般把name的部分叫做环境变量，value的部分则是环境变量的值。

### ubus相关

ubus\_client 加循环

uloop\_run();

mkdir /opt

mount -t nfs -o nolock 192.168.137.128: /home/lqc/shared\_dir/mnt/

mount –t nfs –o nolock 192.168.137.128:/home/lqc/shared\_dir/mnt/opt

tftp –gr sample\_audio 192.168.2.131

### dbg输出写法

#define \_\_DEBUG\_

#ifdef \_\_DEBUG\_

#define dbg(fmt, args...) printf("[%s:%d][%s]"fmt"\n", \_\_FILE\_\_, \_\_LINE\_\_, \_\_FUNCTION\_\_, ##args)

#else

#define dbg(fmt, args...)

#endif

### Vim操作

复制一行：yy 复制多行：nyy

粘贴：p 粘贴多次：np

删除一行：dd 删除多行: ndd

搜索字符串： /word 向下搜索

?word 向上搜索

n 向下搜索下一个结果

N 向上搜索上一个结果

用vim打开Makefile文件，命令模式输入!make。运行makefile文件。

### 查看内存CPU

查看进程：

ps

查看CPU占用：

top

查看内存占用：

cat /proc/pid/status

查看总内存占用：

free

内存占用RSS

### 命令别名设定

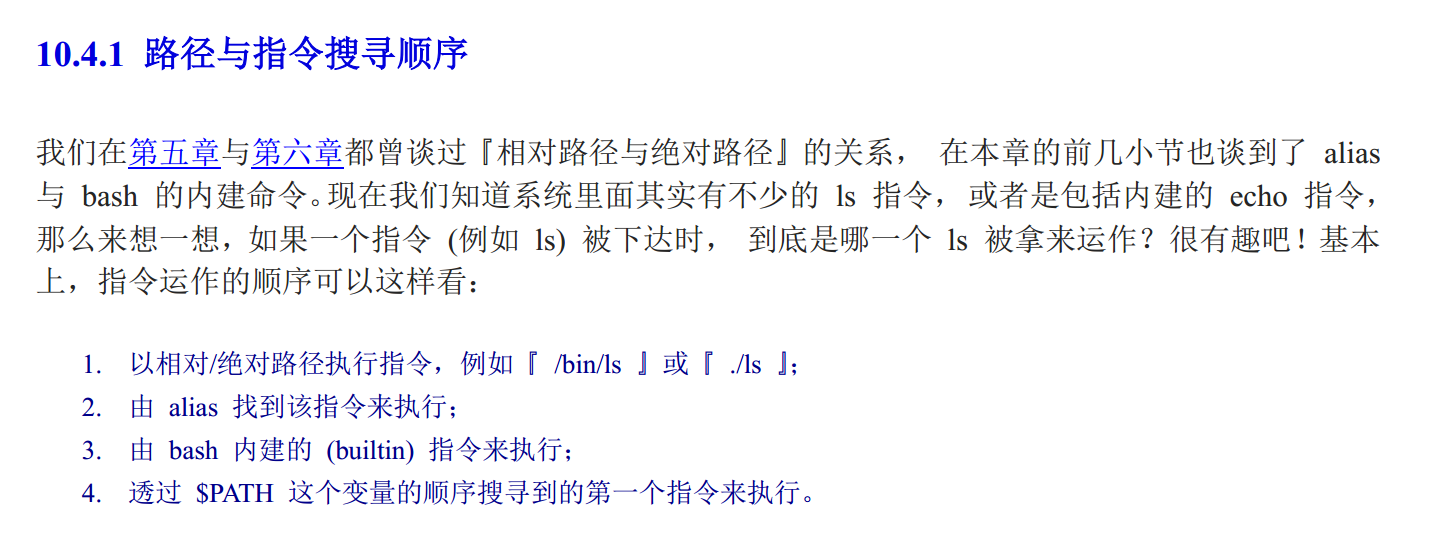
输入alias 查看目前存在的命令别名

新增命令别名：alias lm=’ls -al’

变量的取用：echo $PATH

open():

O\_TRUNC：若文件存在，则长度被截为0，属性不变。



ll / > ~/rootfile 清空建立

ll / >> ~/rootfile 数据累加

grep分析一行信息，若当中存在所需要的信息，就将该行拿出来。

last | grep ‘root’

范例：

#include<stdio.h>

main()

{

char s[80];

fputs(fgets(s,80,stdin),stdout);

}

执行：

this is a test /\*输入\*/

this is a test /\*输出\*/

给变量一个初始值几乎总是正确的，但不要求必须这样做。

### 浮点型值比较

请写出float x与零值比较的if语句：

const float EPSINON = 0.00001;

if ((x >= -EPSINON) && (x <= EPSINON))

不可将浮点变量用“==”或者“！=”与数字比较，应该设法转化成“>=”或者“<=”此类形式。

### sizeof

sizeof():长度以字节为单位。

short s[80]; sizeof(s) = 160;

char s[80]; sizeof(s) = 80;

### goto

由于break语句只影响包围它的最内层循环，要想立即从深层嵌套的循环中退出只有一个使用办法，就是使用goto语句

### malloc

malloc返回一个类型为void \*的指针，标准表示一个void \*类型的指针可以转换为其他任何类型的指针。（对于要求边界对齐的机器，malloc所返回的内存的起始位置始终满足边界对齐要求最严格的类型的要求。）

int\* pi;

pi = malloc(100);

if (pi == NULL)

{

printf(“Out of memory!\n”);

exit(1);

}

如果操作系统无法向malloc提供更多的内存，malloc就返回一个NULL指针。因此，对每一个从malloc返回的指针都进行检查，确保它并非NULL是非常重要的。

传递给free的指针必须是一个从malloc、calloc或者realloc函数返回的指针，传给free函数一个指针，让它释放一块并非动态分配的内存可能导致程序立即终止或在晚些时候终止。试图释放一块动态分配内存的一部分也有可能引起类似的问题。

### C语言内存分配方式

（1）从静态存储区域分配。内存在程序编译的时候已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量，static变量。

（2）在栈上创建。在执行函数时，函数内部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配算法内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

（3）从堆上分配，亦称动态内存分配。程序在运行的时候用malloc或new申请任意多少的内存，程序员自己负责在何时用free或者delete释放内存。动态内存的生存周期由我们决定，使用灵活方便。

### scanf接收&

(1) int a;

scanf(“%d”, &a);

(2) char a;

scanf(“%c”, &a);

(3) char str[20];

scanf(“%s”, str);

### union联合体、共用体

联合体是一种特殊的类，也是一种构造类型的数据结构。在一个“联合体”内能够定义多种不同的数据结构。一个被说明为该“联合体”类型的变量中，同意装入该“联合体”所定义的不论什么一种数据。这些数据共享同一段内存，以达到节省内存的目的。

### EOF

EOF是个宏，End Of File，文件尾标志。从数值上来看，就是整数-1.

libio.h

#define EOF (-1)

键盘产生EOF:ctrl+d

### 无缓冲io(Unbuffered I/O)函数

open,read,write,close等系统函数称为无缓冲io(Unbuffered I/O)函数，因为它们位于C标准库的缓冲区的底层。

无缓冲io函数每次读写都要进内核，调用一个系统调用比调一个用户空间的函数 要慢很多，所以在用户空间开辟io缓存区，用C标准io库函数比较方便，省去了自己管理io缓冲区的麻烦。

read/write:

读/写常规文件是不会阻塞的，但是向终端设备或者网络读/写则不一定。

轮询（poll），调用者只是查询一下，而不是一直阻塞在这里，这样可以同时监视多个设备。

非阻塞io有一个缺点，如果所有设备都一直没有数据到达，调用者需要反复查询做无用功，如果阻塞在那里，操作系统可以调度别的进程执行，就不会做无用功了。在使用非阻塞io时，通常不会在一个while循环中一直不停的查询，而是每延迟等待一会来查询一下，以免做太多无用功，在延迟等待的时候可以调度其他进程执行。

select函数可以阻塞地同时监视多个设备，还可以设定阻塞等待的超时时间，从而圆满地解决这个问题。

### select

s32Ret = select(AencFd + 1, &read\_fds, NULL, NULL, &TimeoutVal);//int select(int maxfdp, fd\_set \*readset, fd\_set \*writeset, fd\_set \*exceptset, struct timeval \*timeout);

select系统监听调用的用途是：再一段指定的时间内，监听用户感兴趣的文件描述符上可读、可写和异常等事件。

maxfdp：被监听的文件描述符的总数，它比所有文件描述符集合中的文件描述符的最大值大1，因为文件描述符是从0开始计数的；

readfds,writefds,exceptset：分指向可读，可写和异常等事件对应得描述符集合；

timeout：用于设置select函数的超时事件，即告诉内核select等待多长时间之后就放弃等待。timeout==NULL表示等待无限长的时间。返回值：超时返回0；失败返回-1；成功返回大于0的整数，这个整数表示已经准备就绪描述符的数目。

### 非阻塞读 fcntl

int flags;

flags = fcntl(STDIN\_FILENO, F\_GETFL);

flags |= O\_NONBLOCK;

if (fcntl(STDIN\_FILENO, F\_SETFL, flags) == -1)

{

perror(“fcntl”);

exit(1);

}

将输入输出设置为非阻塞时，read()函数等不到数据输入的时候就会直接返回出错信息，而不是继续等待用户输入。

read错误代码：

EAGAIN 当使用不可阻断I/O 时（O\_NONBLOCK），若无数据可读取则返回此值

### 硬链接 符号链接

link函数创建硬链接，其原理是在目录的数据块中添加一条新记录，其中的inode号字段和原文件相同。

symlink函数创建一个符号链接，这需要创建一个新的inode，其中st\_mode字段的文件类型是符号链接，原文件的路径保存在inode中或者分配一个数据块来保存。

文件真正删除的条件是与之相关的所有硬链接文件均被删除。

软链接类似于Windows的快捷方式，它其实是一个文本文件，其中包含的有另一个文件的位置信息。

软链接和硬链接的区别：

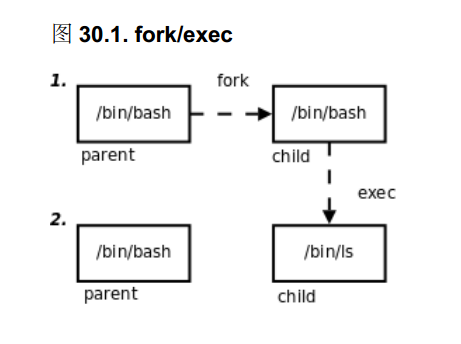
软链接和硬链接在概念上和实现上是不一样的。硬链接中，原文件和链接文件公用一个inode号，说明他们是同一个文件，而软链接中，原文件和链接文件拥有不同的inode号，表明它们是两个不同的文件；在文件属性上，软链接表明写出了是链接文件，而硬链接没有写出来，因为在本质上硬链接文件和原文件是完全平等关系；链接数目是不一样的，软链接的链接数目不会增加；文件大小是不一样的，硬链接文件显示的文件大小是跟原文件是一样的，因为是等同的，而软链接显示的大小于原文件就不同，其实是一个链接名大小。

### 进程控制块PCB

每个进程在内核中都有一个进程控制块（PCB）来维护进程相关的信息，Linux内核进程控制块是task\_struct结构体。

### fork exec

在Shell下输入命令可以运行一个程序，是因为Shell进程在读取用户输入的命令之后会调用fork复制出一个新的Shell进程，然后新的Shell进程调用exec执行新的程序。



因为a.out进程是Shell进程的子进程，子进程的PCB是根据父进程复制而来的，所以其中的umask掩码也和父进程一样。

PATH：可执行文件的搜索路径。

SHELL：当前Shell，它的值通常是/bin/bash。

TERM：当前终端类型，在图形界面终端下它的值通常是xterm，终端类型决定了一些程序的输出显示方式，比如图形界面终端可以显示汉字，而字符终端一般不行。

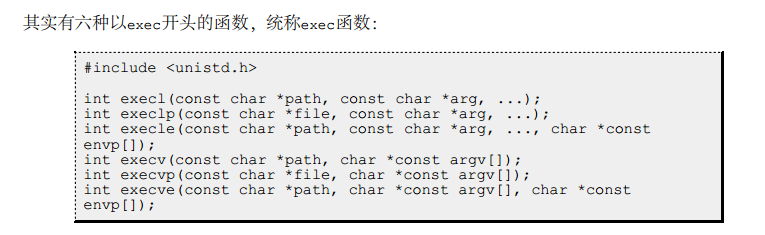
LANG：语言和locale，决定了字符编码以及时间、货币等信息的显示格式。

HOME：当前用户主目录的路径，很多程序需要在主目录下保存配置文件，使得每个用户在运行该程序时都有自己的一套配置。

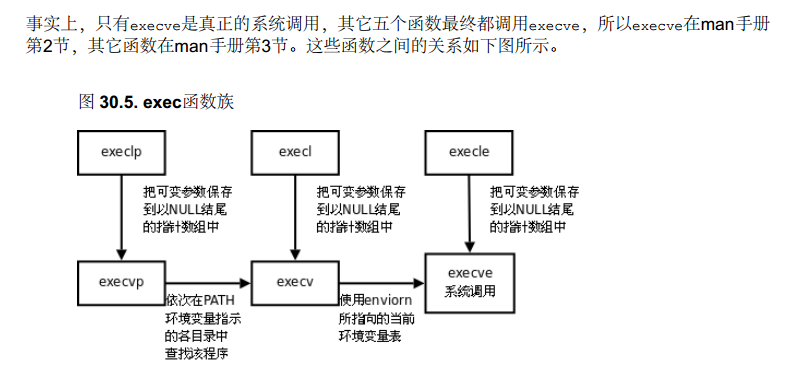
父进程在创建子进程时会复制一份环境变量给子进程，但此后二者的环境变量互不影响。

fork函数的特点概括起来就是“调用一次，返回两次”，在父进程中调用一次，在父进程和子进程中各返回一次。从上图可以看出，一开始是一个控制流程，调用fork之后发生了分叉，变成两个控制流程，这也就是“fork”（分叉）这个名字的由来了。子进程中fork的返回值是0，而父进程中fork的返回值则是子进程的id（从根本上说fork是从内核返回的，内核自有办法让父进程和子进程返回不同的值），这样当fork函数返回后，程序员可以根据返回值的不同让父进程和子进程执行不同的代码。

fork的返回值这样规定是有道理的。fork在子进程中返回0，子进程仍可以调用getpid函数得到自己的进程id，也可以调用getppid函数得到父进程的id。在父进程中用getpid可以得到自己的进程id，然而要想得到子进程的id，只有将fork的返回值记录下来，别无它法。



这些函数如果调用成功则加载新的程序从启动代码开始执行，不再返回，如果调用出错则返回-1，所以exec函数只有出错的返回值而没有成功的返回值。



dup2(fd, STDIN\_FILENO);

将标准输入重定向到这个文件

获取键盘输入直到，遇到EOF

/\* upper.c \*/

#include <ctype.h>

#include <stdio.h>

int main(void)

{

int ch;

while((ch = getchar()) != EOF) {

putchar(toupper(ch));

}

return 0;

}



### wait或waitpid

进程的父进程可以调用wait或waitpid获取这些信息，然后彻底清除掉这个进程。如果一个进程已经终止，但是它的父进程尚未调用wait或waitpid对它进行清理，这时的进程状态称为僵尸（Zombie）进程。

调用wait和waitpid不仅可以获得子进程的终止信息，还可以使父进程阻塞等待子进程终止，起到进程间同步的作用。

wait和waitpid函数的原型是：

#include <sys/types.h>

#include <sys/wait.h>

pid\_t wait(int \*status);

pid\_t waitpid(pid\_t pid, int \*status, int options);

若调用成功则返回清理掉的子进程id，若调用出错则返回-1。父进程调用wait或waitpid时可能会：

阻塞（如果它的所有子进程都还在运行）。

带子进程的终止信息立即返回（如果一个子进程已终止，正等待父进程读取其终止信息）。

出错立即返回（如果它没有任何子进程）。

这两个函数的区别是：

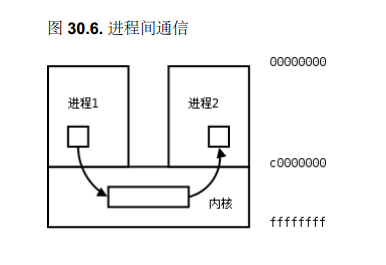
如果父进程的所有子进程都还在运行，调用wait将使父进程阻塞，而调用waitpid时如果在options参数中指定WNOHANG可以使父进程不阻塞而立即返回0。

wait等待第一个终止的子进程，而waitpid可以通过pid参数指定等待哪一个子进程。

可见，调用wait和waitpid不仅可以获得子进程的终止信息，还可以使父进程阻塞等待子进程终止，起到进程间同步的作用。如果参数status不是空指针，则子进程的终止信息通过这个参数传出，如果只是为了同步而不关心子进程的终止信息，可以将status参数指定为NULL。

### 进程间通信 管道

每个进程各自有不同的用户地址空间，任何一个进程的全局变量在另一个进程中都看不到，所以进程之间要交换数据必须通过内核，在内核中开辟一块缓冲区，进程1把数据从用户空间拷到内核缓冲区，进程2再从内核缓冲区把数据读走，内核提供的这种机制称为进程间通信（IPC，InterProcess Communication）。



管道是一种最基本的IPC机制

1. 父进程调用pipe开辟管道，得到两个文件描述符指向管道的两端。

2. 父进程调用fork创建子进程，那么子进程也有两个文件描述符指向同一管道。

3. 父进程关闭管道读端，子进程关闭管道写端。父进程可以往管道里写，子进程可以从管道里读，管道是用环形队列实现的，数据从写端流入从读端流出，这样就实现了进程间通信。

两个进程通过一个管道只能实现单向通信，比如上面的例子，父进程写子进程读，如果有时候也需要子进程写父进程读，就必须另开一个管道。

管道的读写端通过打开的文件描述符来传递，因此要通信的两个进程必须从它们的公共祖先那里继承管道文件描述符。上面的例子是父进程把文件描述符传给子进程之后父子进程之间通信，也可以父进程fork两次，把文件描述符传给两个子进程，然后两个子进程之间通信，总之需要通过fork传递文件描述符使两个进程都能访问同一管道，它们才能通信。

调用pipe函数时在内核中开辟一块缓冲区（称为管道）用于通信，它有一个读端一个写端，然后通过filedes参数传出给用户程序两个文件描述符，filedes[0]指向管道的读端，filedes[1]指向管道的写端

项目功能需要外网权限，同时，需要使用baidu，csdn等外网检索查询相关资料，提高工作效率，望批准。

### shell 信号 SIGINT等

Shell可以同时运行一个前台进程和任意多个后台进程，只有前台进程才能接到像Ctrl-C这种控制键产生的信号。前台进程在运行过程中用户随时可能按下Ctrl-C而产生一个信号，也就是说该进程的用户空间代码执行到任何地方都有可能收到SIGINT信号而终止，所以信号相对于进程的控制流程来说是异步（Asynchronous）的。

SIGHUP 1 Term

SIGINT 2 Term (Ctrl+C)

SIGQUIT 3 Core (Ctrl+\)

SIGILL 4 Core

SIGTSTP (Ctrl+Z)

如果不想按默认动作处理信号，用户程序可以调用sigaction(2)函数告诉内核如何处理某种信号（sigaction函数稍后详细介绍），可选的处理动作有以下三种：

1. 忽略此信号。

2. 执行该信号的默认处理动作。

3. 提供一个信号处理函数，要求内核在处理该信号时切换到用户态执行这个处理函数，这种方式称为捕捉（Catch）一个信号

SIGINT的默认处理动作是终止进程，SIGQUIT的默认处理动作是终止进程并且Core Dump

在开发调试阶段可以用ulimit命令改变这个限制，允许产生core文件。

### 信号

kill命令是调用kill函数实现的。kill函数可以给一个指定的进程发送指定的信号。

raise函数可以给当前进程发送指定的信号（自己给自己发信号）。

#include <signal.h>

int kill(pid\_t pid, int signo);

int raise(int signo);

abort函数使当前进程接收到SIGABRT信号而异常终止。

#include <stdlib.h>

void abort(void);

SIGPIPE是一种由软件条件产生的信号。本节主要介绍alarm函数和SIGALRM信号。

#include <unistd.h>

unsigned int alarm(unsigned int seconds);

调用alarm函数可以设定一个闹钟，也就是告诉内核在seconds秒之后给当前进程发SIGALRM信号，该信号的默认处理动作是终止当前进程。这个函数的返回值是0或者是以前设定的闹钟时间还余下的秒数。

信号集操作函数

#include <signal.h>

#include <stdio.h>

#include <unistd.h>

void printsigset(const sigset\_t \*set)

{

int i;

for (i = 1; i < 32; i++)

if (sigismember(set, i) == 1)

putchar('1');

else

putchar('0');

puts("");

}

int main(void)

{

sigset\_t s, p;

sigemptyset(&s);

sigaddset(&s, SIGINT);

sigprocmask(SIG\_BLOCK, &s, NULL);

while (1) {

sigpending(&p);

printsigset(&p);

sleep(1);

}

return 0;

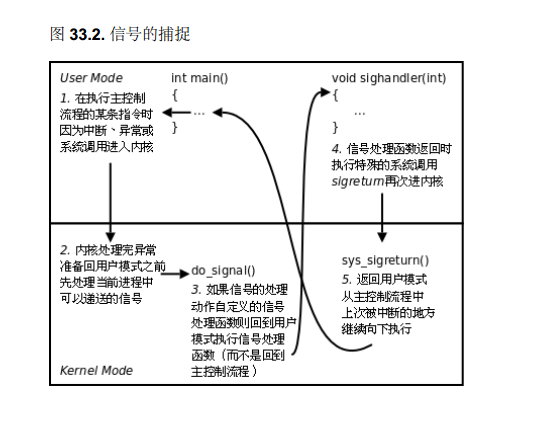
}

函数sigemptyset初始化set所指向的信号集，使其中所有信号的对应bit清零，表示该信号集不包含任何有效信号。

初始化sigset\_t变量之后就可以在调用sigaddset和sigdelset在该信号集中添加或删除某种有效信号。

调用函数sigprocmask可以读取或更改进程的信号屏蔽字。

sigpending读取当前进程的未决信号集，通过set参数传出。



#include <unistd.h>

#include <signal.h>

#include <stdio.h>

void sig\_alrm(int signo)

{

/\* nothing to do \*/

}

unsigned int mysleep(unsigned int nsecs)

{

struct sigaction newact, oldact;

unsigned int unslept;

newact.sa\_handler = sig\_alrm;

sigemptyset(&newact.sa\_mask);

newact.sa\_flags = 0;

sigaction(SIGALRM, &newact, &oldact);

alarm(nsecs);

pause();

unslept = alarm(0);

sigaction(SIGALRM, &oldact, NULL);

return unslept;

}

int main(void)

{

while(1){

mysleep(2);

printf("Two seconds passed\n");

}

return 0;

}

pause函数使调用进程挂起直到有信号递达。如果信号的处理动作是终止进程，则进程终止，pause函数没有机会返回；如果信号的处理动作是忽略，则进程继续处于挂起状态，pause不返回；如果信号的处理动作是捕捉，则调用了信号处理函数之后pause返回-1

1. main函数调用mysleep函数，后者调用sigaction注册了SIGALRM信号的处理函数sig\_alrm。

2. 调用alarm(nsecs)设定闹钟。

3. 调用pause等待，内核切换到别的进程运行。

4. nsecs秒之后，闹钟超时，内核发SIGALRM给这个进程。

5. 从内核态返回这个进程的用户态之前处理未决信号，发现有SIGALRM信号，其处理函数是sig\_alrm。

6. 切换到用户态执行sig\_alrm函数，进入sig\_alrm函数时SIGALRM信号被自动屏蔽，

从sig\_alrm函数返回时SIGALRM信号自动解除屏蔽。然后自动执行系统调用sigreturn再次进入内核，再返回用户态继续执行进程的主控制流程（main函数调用的mysleep函数）。

7. pause函数返回-1，然后调用alarm(0)取消闹钟，调用sigaction恢复SIGALRM信号以前的处理

动作。

### 可重入函数

当捕捉到信号时，不论进程的主控制流程当前执行到哪儿，都会先跳到信号处理函数中执行，从信号处理函数返回后再继续执行主控制流程。信号处理函数是一个单独的控制流程，因为它和主控制流程是异步的，二者不存在调用和被调用的关系，并且使用不同的堆栈空间。引入了信号处理函数使得一个进程具有多个控制流程，如果这些控制流程访问相同的全局资源（全局变量、硬件资源等），就有可能出现冲突

像上例这样，insert函数被不同的控制流程调用，有可能在第一次调用还没返回时就再次进入该函数，这称为重入，insert函数访问一个全局链表，有可能因为重入而造成错乱，像这样的函数称为不可重入函数，反之，如果一个函数只访问自己的局部变量或参数，则称为可重入（Reentrant）函数。

在上面的例子中，main和sighandler都调用insert函数则有可能出现链表的错乱，其根本原因在于，对全局链表的插入操作要分两步完成，不是一个原子操作，假如这两步操作必定会一起做完，中间不可能被打断，就不会出现错乱了。

如果对全局数据的访问只有一行代码，也不一定是原子操作

sig\_atomic\_t类型的变量应该总是加上volatile限定符，因为要使用sig\_atomic\_t类型的理由也正是要加volatile限定符的理由。

C语言提供了volatile限定符，如果将上述变量定义为volatile sig\_atomic\_t a=0;那么即使指定了优化选项，编译器也不会优化掉对变量a内存单元的读写。

对于程序中存在多个执行流程访问同一全局变量的情况，volatile限定符是必要的，此外，虽然程序只有单一的执行流程，但是变量属于以下情况之一的，也需要volatile限定：

变量的内存单元中的数据不需要写操作就可以自己发生变化，每次读上来的值都可能不一样

即使多次向变量的内存单元中写数据，只写不读，也并不是在做无用功，而是有特殊意义的

要是“解除信号屏蔽”和“挂起等待信号”这两步能合并成一个原子操作就好了，这正是sigsuspend函数的功能。sigsuspend包含了pause的挂起等待功能，同时解决了竞态条件的问题，在对时序要求严格的场合下都应该调用sigsuspend而不是pause。

### 关于SIGCHLD信号

其实，子进程在终止时会给父进程发SIGCHLD信号，该信号的默认处理动作是忽略，父进程可以自定义SIGCHLD信号的处理函数，这样父进程只需专心处理自己的工作，不必关心子进程了，子进程终止时会通知父进程，父进程在信号处理函数中调用wait清理子进程即可。

### 终端

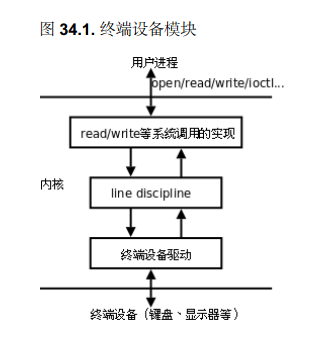
一台PC通常只有一套键盘和显示器，也就是只有一套终端设备，但是可以通过Ctrl-Alt-F1~Ctrl-AltF6切换到6个字符终端，相当于有6套虚拟的终端设备，它们共用同一套物理终端设备，对应的设备文件分别是/dev/tty1~/dev/tty6，所以称为虚拟终端（Virtual Terminal）。设备文件/dev/tty0表示当前虚拟终端，比如切换到Ctrl-Alt-F1的字符终端时/dev/tty0就表示/dev/tty1，切换到Ctrl-AltF2的字符终端时/dev/tty0就表示/dev/tty2，就像/dev/tty一样也是一个通用的接口，但它不能表示图形终端窗口所对应的终端。

串口终端：

做嵌入式开发时经常会用到串口终端，目标板的每个串口对应一个终端设备，比

如/dev/ttyS0、/dev/ttyS1等，将主机和目标板用串口线连起来，就可以在主机上通过Linux的minicom或Windows的超级终端工具登录到目标板的系统。

内核中处理终端设备的模块包括硬件驱动程序和线路规程（Line Discipline）。



硬件驱动程序负责读写实际的硬件设备，比如从键盘读入字符和把字符输出到显示器，线路规程像一个过滤器，对于某些特殊字符并不是让它直接通过，而是做特殊处理，比如在键盘上按下Ctrl-Z，对应的字符并不会被用户程序的read读到，而是被线路规程截获，解释成SIGTSTP信号发给前台进程，通常会使该进程停止。线路规程应该过滤哪些字符和做哪些特殊处理是可以配置的。

### 线程介绍

以前我们讲过，main函数和信号处理函数是同一个进程地址空间中的多个控制流程，多线程也是如此，但是比信号处理函数更加灵活，信号处理函数的控制流程只是在信号递达时产生，在处理完信号之后就结束，而多线程的控制流程可以长期并存，操作系统会在各线程之间调度和切换，就像在多个进程之间调度和切换一样。

由于同一进程的多个线程共享同一地址空间，因此Text Segment、Data Segment都是共享的，如果定义一个函数，在各线程中都可以调用，如果定义一个全局变量，在各线程中都可以访问到，除此之外，各线程还共享以下进程资源和环境：

文件描述符表

每种信号的处理方式（SIG\_IGN、SIG\_DFL或者自定义的信号处理函数）

当前工作目录

用户id和组id

但有些资源是每个线程各有一份的：

线程id

上下文，包括各种寄存器的值、程序计数器和栈指针

栈空间

errno变量

信号屏蔽字

调度优先级

### pthread线程库函数

我们将要学习的线程库函数是由POSIX标准定义的，称为POSIX thread或者pthread。在Linux上线程函数位于libpthread共享库中，因此在编译时要加上-lpthread选项。

进程id：pid\_t pid = getpid();

线程id：pthread\_t tid = pthread\_self();

在Linux上，pthread\_t类型是一个地址值，属于同一个进程的多个线程调用getpid()可以得到相同的进程号，而调用pthread\_self()得到的线程号各不相同。

如果任意一个线程调用了exit或\_exit，则整个进程的所有线程都终止，由于从main函数return也相当于调用exit，为了防止新创建的线程还没有得到执行就终止，我们在main函数return之前延时1秒，这只是一种权宜之计

exit（正常结束进程）

exit()用来正常终结目前进程的执行，并把参数status返回给父进程，

而进程所有的缓冲区数据会自动写回并关闭未关闭的文件。

\_exit(结束进程执行）

\_exit（）用来立刻结束目前进程的执行，并把参数status返回给父进程，并关闭未关闭的文件。此函数调用后不会返回，并且会传递SIGCHLD信号给父进程，父进程可以由wait数取得子进程结束状态。

\_exit（）不会处理标准I/O缓冲区，如要更新缓冲区请使用exit（）。

终止线程

如果需要只终止某个线程而不终止整个进程，可以有三种方法：

1、从线程函数return。这种方法对主线程不适用，从main函数return相当于调用exit。

2、一个线程可以调用pthread\_cancel终止同一进程中的另一个线程。

3、线程可以调用pthread\_exit终止自己。

下面介绍pthread\_exit的和pthread\_join的用法。

#include <pthread.h>

void pthread\_exit(void \*value\_ptr);

value\_ptr是void \*类型，和线程函数返回值的用法一样，其它线程可以调用pthread\_join获得这个指针。

需要注意，pthread\_exit或者return返回的指针所指向的内存单元必须是全局的或者是用malloc分配的，不能在线程函数的栈上分配，因为当其它线程得到这个返回指针时线程函数已经退出了。

#include <pthread.h>

int pthread\_join(pthread\_t thread, void \*\*value\_ptr);

返回值：成功返回0，失败返回错误号

调用该函数的线程将挂起等待，直到id为thread的线程终止。thread线程以不同的方法终止，通过pthread\_join得到的终止状态是不同的，总结如下：

1、如果thread线程通过return返回，value\_ptr所指向的单元里存放的是thread线程函数的返回值。

2、如果thread线程被别的线程调用pthread\_cancel异常终止掉，value\_ptr所指向的单元里存放的是常数PTHREAD\_CANCELED。

3、如果thread线程是自己调用pthread\_exit终止的，value\_ptr所指向的单元存放的是传给pthread\_exit的参数。

如果对thread线程的终止状态不感兴趣，可以传NULL给value\_ptr参数。

### 线程mutex

多个线程同时访问共享数据时可能会冲突，这跟前面讲信号时所说的可重入性是同样的问题。

对于多线程的程序，访问冲突的问题是很普遍的，解决的办法是引入互斥锁（Mutex，Mutual Exclusive Lock），获得锁的线程可以完成“读-修改-写”的操作，然后释放锁给其它线程，没有获得锁的线程只能等待而不能访问共享数据，这样“读-修改-写”三步操作组成一个原子操作，要么都执行，要么都不执行，不会执行到中间被打断，也不会在其它处理器上并行做这个操作。

Mutex用pthread\_mutex\_t类型的变量表示，可以这样初始化和销毁：

#include <pthread.h>

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex,

const pthread\_mutexattr\_t \*restrict attr);

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

如果Mutex变量是静态分配的（全局变量或static变量），也可以用宏定义PTHREAD\_MUTEX\_INITIALIZER来初始化，相当于用pthread\_mutex\_init初始化并且attr参数为NULL。

#include <pthread.h>

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

返回值：成功返回0，失败返回错误号。

一个线程可以调用pthread\_mutex\_lock获得Mutex，如果这时另一个线程已经调用pthread\_mutex\_lock获得了该Mutex，则当前线程需要挂起等待，直到另一个线程调用pthread\_mutex\_unlock释放Mutex，当前线程被唤醒，才能获得该Mutex并继续执行。

如果一个线程既想获得锁，又不想挂起等待，可以调用pthread\_mutex\_trylock，如果Mutex已经被另一个线程获得，这个函数会失败返回EBUSY，而不会使线程挂起等待。

细心的读者应该已经看出问题了：对Mutex变量的读取、判断和修改不是原子操作。如果两个线程同时调用lock，这时Mutex是1，两个线程都判断Mutex>0成立，然后其中一个线程置Mutex=0，而另一个线程并不知道这一情况，也置mutex=0，于是两个线程都以为自己获得了锁。

为了实现互斥锁操作，大多数体系结构都提供了swap或exchange指令，该指令的作用是把寄存器和内存单元的数据相交换，由于只有一条指令，保证了原子性，即使是多处理器平台，访问内存的总线周期也有先后，一个处理器上的交换指令执行时另一个处理器的交换指令只能等待总线周期。

一般情况下，如果同一个线程先后两次调用lock，在第二次调用时，由于锁已经被占用，该线程会挂起等待别的线程释放锁，然而锁正是被自己占用着的，该线程又被挂起而没有机会释放锁，因此就永远处于挂起等待状态了，这叫做死锁（Deadlock）。另一种典型的死锁情形是这样：线程A获得了锁1，线程B获得了锁2，这时线程A调用lock试图获得锁2，结果是需要挂起等待线程B释放

锁2，而这时线程B也调用lock试图获得锁1，结果是需要挂起等待线程A释放锁1，于是线程A和B都永远处于挂起状态了。不难想象，如果涉及到更多的线程和更多的锁，有没有可能死锁的问题将会变得复杂和难以判断。

### 条件变量（Condition Variable）

在pthread库中通过条件变量（Condition Variable）来阻塞等待一个条件，或者唤醒等待这个条件的线程。Condition Variable用pthread\_cond\_t类型的变量表示，可以这样初始化和销毁：

#include <pthread.h>

int pthread\_cond\_destroy(pthread\_cond\_t \*cond);

int pthread\_cond\_init(pthread\_cond\_t \*restrict cond,

const pthread\_condattr\_t \*restrict attr);

pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;

如果Condition Variable是静态分配的，也可以用宏定义PTHEAD\_COND\_INITIALIZER初始化，相当于用pthread\_cond\_init函数初始化并且attr参数为NULL。

Condition Variable的操作可以用下列函数：

#include <pthread.h>

int pthread\_cond\_timedwait(pthread\_cond\_t \*restrict cond,

pthread\_mutex\_t \*restrict mutex,

const struct timespec \*restrict abstime);

int pthread\_cond\_wait(pthread\_cond\_t \*restrict cond,

pthread\_mutex\_t \*restrict mutex);

int pthread\_cond\_broadcast(pthread\_cond\_t \*cond);

int pthread\_cond\_signal(pthread\_cond\_t \*cond);

返回值：成功返回0，失败返回错误号。

可见，一个Condition Variable总是和一个Mutex搭配使用的。一个线程可以调用pthread\_cond\_wait在一个Condition Variable上阻塞等待，这个函数做以下三步操作：

1. 释放Mutex

2. 阻塞等待

3. 当被唤醒时，重新获得Mutex并返回

pthread\_cond\_timedwait函数还有一个额外的参数可以设定等待超时，如果到达了abstime所指定的时刻仍然没有别的线程来唤醒当前线程，就返回ETIMEDOUT。

### Semaphore信号量

Mutex变量是非0即1的，可看作一种资源的可用数量，初始化时Mutex是1，表示有一个可用资源，加锁时获得该资源，将Mutex减到0，表示不再有可用资源，解锁时释放该资源，将Mutex重新加到1，表示又有了一个可用资源。

信号量（Semaphore）和Mutex类似，表示可用资源的数量，和Mutex不同的是这个数量可以大于1。

#include <semaphore.h>

int sem\_init(sem\_t \*sem, int pshared, unsigned int value);

int sem\_wait(sem\_t \*sem);

int sem\_trywait(sem\_t \*sem);

int sem\_post(sem\_t \* sem);

int sem\_destroy(sem\_t \* sem);

semaphore变量的类型为sem\_t，sem\_init()初始化一个semaphore变量，value参数表示可用资源的数量，pshared参数为0表示信号量用于同一进程的线程间同步，本节只介绍这种情况。在用完semaphore变量之后应该调用sem\_destroy()释放与semaphore相关的资源。

调用sem\_wait()可以获得资源，使semaphore的值减1，如果调用sem\_wait()时semaphore的值已经是0，则挂起等待。如果不希望挂起等待，可以调用sem\_trywait()。调用sem\_post()可以释放资源，使semaphore的值加1，同时唤醒挂起等待的线程。

### 静态static局部变量 静态全局变量 外部extern变量

extern 与 static

extern可以置于变量或者函数前，以标示变量或者函数的定义在别的文件中，提示编译器遇到此变量或者函数时在其他模块中寻找其定义。此外extern也可以用来进行链接指定。

静态局部变量：

静态局部变量属于静态存储方式。

1、静态局部变量在函数内定义，但不像自动变量那样，当调用时就存在，退出时就消失。静态局部变量始终存在着，也就是说它的生存周期为整个源程序； 2、静态局部变量的生存周期虽然为整个源程序，但是其作用域与自动变量相同；3、允许对构造类静态局部类赋初值。若未附以初值，则由系统自动赋以0值；4、对基本类型的静态局部变量若在说明时未赋以初值，则系统自动赋以0值。而对于自动变量不赋初值，则其值是不定的。根据静态局部变量的特点，可以看出它是生存期为整个源程序的量。虽然离开定义它的函数后不能使用，但如果再次调用定义它的函数时，它又可以继续使用，而且保存了前次被调用后留下来的值。因此，当多次调用一个函数且要求在调用之间保留某些变量的值时，可考虑采用静态局部变量。虽然全局变量也可以达到上述目的，但全局变量有时会造成意外的副作用，因此仍以采用局部静态变量为宜。

静态全局变量：

全局变量的说明之前再冠以static就构成了静态的全局变量。全局变量本身就是静态存储方式，静态全局变量当然也是静态存储方式。这两者的区别是：1、当非静态全局变量的作用域是整个源程序，当一个源程序由多个源文件组成时，非静态的全局变量在各个源文件中都是有效的。相反，静态全局变量则限制了作用域，即只在定义该变量的源文件内是有效的，在同一源程序的其他源文件中不能使用。由于静态全局变量的作用域局限在一个源文件中，只能为该源文件内的函数公用，因此可以避免在其他源文件中引起错误。

因此，可以看出，把局部变量改变为静态局部变量后是改变了它的存储方式即改变了生存周期。把全局变量改变为静态全局变量后是改变了它的作用域，限制了它的使用范围。因此，static这个说明符在不同的地方所起的作用不同。

外部extern变量：

所有未加static前缀的全局变量和函数都具有全局可见性，其他源文件也能访问。如果加了static，就会对其他源文件隐藏。

extern告诉编译器这个变量或函数在其他文档中已被定义了。

### C 语言整数与字符串的相互转换

整数转字符串

拓展函数 itoa

itoa (表示 integer to alphanumeric)是把整型数转换成字符串的一个函数。

char\* itoa(int value,char\*string,int radix);//value: 要转换的整数，string: 转换后的字符串,radix: 转换进制数，如2,8,10,16 进制等。

int number1 = 123456;

int number2 = -123456;

char string[16] = {0};

itoa(number1,string,10);

printf("数字：%d 转换后的字符串为：%s\n",number1,string);

字符串转整数

拓展函数 atoi

atoi (表示 alphanumeric to integer)是把字符串转换成整型数的一个函数。

int atoi(const char \*nptr);//字符串转整数函数，nptr: 要转换的字符串

printf("字符串\"123456\"转换为数字：%d\n",atoi("123456"));

printf("字符串\"-123456\"转换为数字：%d\n",atoi("-123456"));

### makefile cmake

Windows下常用IDE来编译，Linux下直接使用gcc来编译，编译过程是Linux嵌入式编程的基础。

实际开发很少会像上面一样自己编写Makefile文件，可借助cmake工具来生成。

CMake是一个跨平台的安装（编译）工具，可以用简单的语句来描述所有平台的安装(编译过程)。

这时候就出现了Cmake这个工具，cmake就可以更加简单的生成makefile文件给上面那个make用。当然cmake还有其他功能，就是可以跨平台生成对应平台能用的makefile，你不用再自己去修改了。

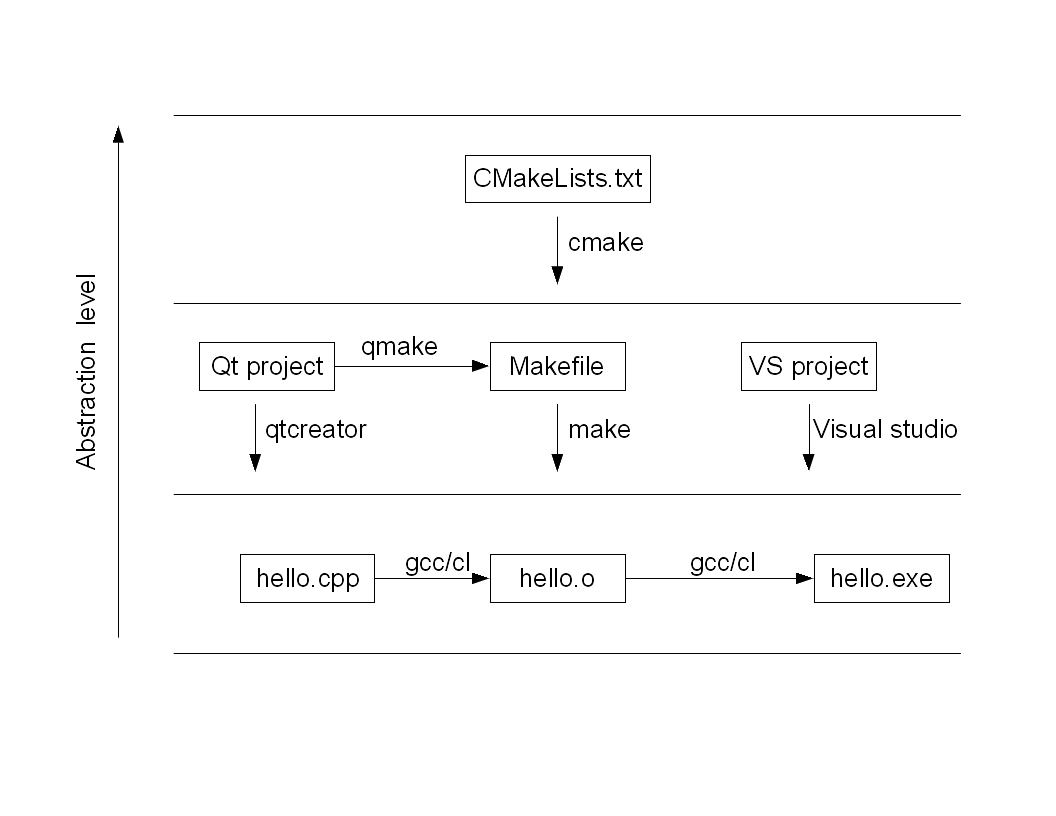
可是cmake根据什么生成makefile呢？它又要根据一个叫CMakeLists.txt文件（学名：组态档）去生成makefile。

总结一下，make用来执行Makefile，cmake用来执行CMakeLists.txt，qmake用来处理\*.pro工程文件。Makefile的抽象层次最低，cmake和qmake在Linux等环境下最后还是会生成一个Makefile。cmake和qmake支持跨平台，cmake的做法是生成指定编译器的工程文件，而qmake完全自成体系。

具体使用时，Linux下，小工程可手动写Makefile，大工程用automake来帮你生成Makefile，要想跨平台，就用cmake。如果GUI用了Qt，也可以用qmake+\*.pro来管理工程，这也是跨平台的。当然，cmake中也有针对Qt的一些规则，并代替qmake帮你将qt相关的命令整理好了。

另外，需要指出的是，make和cmake主要命令只有一条，make用于处理Makefile，cmake用来转译CMakeLists.txt，而qmake是一个体系，用于支撑一个编程环境，它还包含除qmake之外的其它多条命令(比如uic，rcc,moc)。

上个简图，其中cl表示visual studio的编译器，gcc表示linux下的编译器



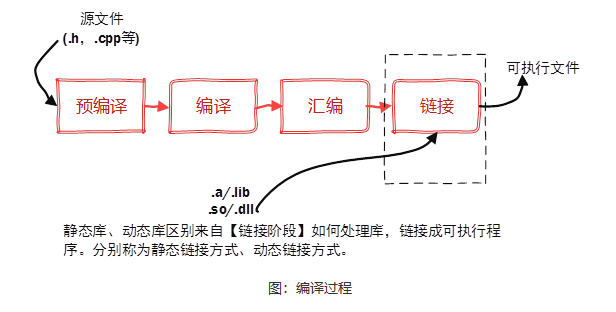
### C++静态库与动态库

1.什么是库

库是写好的现有的，成熟的，可以复用的代码。现实中每个程序都要依赖很多基础的底层库，不可能每个人的代码都从零开始，因此库的存在意义非同寻常。

本质上来说库是一种可执行代码的二进制形式，可以被操作系统载入内存执行。库有两种：静态库（.a、.lib）和动态库（.so、.dll）。

所谓静态、动态是指链接。回顾一下，将一个程序编译成可执行程序的步骤：



2.静态库

之所以成为【静态库】，是因为在链接阶段，会将汇编生成的目标文件.o与引用到的库一起链接打包到可执行文件中。因此对应的链接方式称为静态链接。

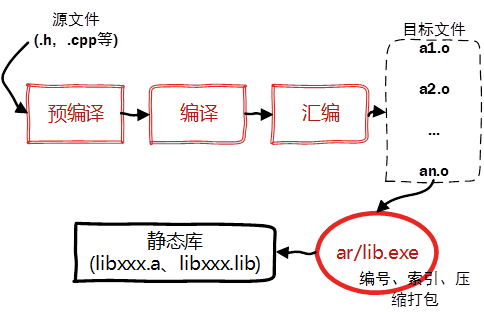
试想一下，静态库与汇编生成的目标文件一起链接为可执行文件，那么静态库必定跟.o文件格式相似。其实一个静态库可以简单看成是一组目标文件（.o/.obj文件）的集合，即很多目标文件经过压缩打包后形成的一个文件。静态库特点总结：

- 静态库对函数库的链接是放在编译时期完成的。

- 程序在运行时与函数库再无瓜葛，移植方便。

- 浪费空间和资源，因为所有相关的目标文件与牵涉到的函数库被链接合成一个可执行文件。

Linux下使用*ar*工具、Windows下vs使用*lib.exe*，将目标文件压缩到一起，并且对其进行编号和索引，以便于查找和检索。一般创建静态库的步骤如图所示：

[](http://www.runoob.com/wp-content/uploads/2015/05/16201602-e0b2900a915d4dce8b4af5153c8b2d72.png)

图：创建静态库过程

2.1.Linux下创建与使用静态库

2.1.1.Linux静态库命名规则

Linux静态库命名规范，必须是"lib[your\_library\_name].a"：lib为前缀，中间是静态库名，扩展名为.a。

2.1.2.创建静态库（.a）

通过上面的流程可以知道，Linux创建静态库过程如下：

-  首先，将代码文件编译成目标文件.o（StaticMath.o）

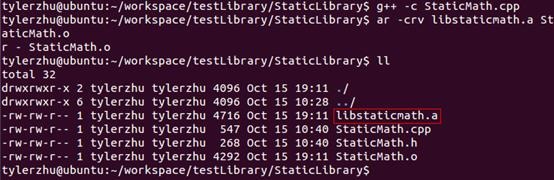
g++ -c StaticMath.cpp

注意带参数-c，否则直接编译为可执行文件

-  然后，通过ar工具将目标文件打包成.a静态库文件

ar -crv libstaticmath.a StaticMath.o

生成静态库**libstaticmath.a**。

[](http://www.runoob.com/wp-content/uploads/2015/05/16201604-713c3a4a83aa4c17868a16f13b49027b.jpg)

大一点的项目会编写makefile文件（CMake等等工程管理工具）来生成静态库，输入多个命令太麻烦了。

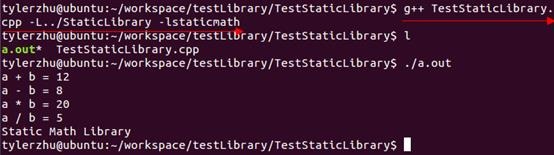
2.1.3.使用静态库

编写使用上面创建的静态库的测试代码：

|  |
| --- |
| **测试代码：** |
| #include "StaticMath.h"  #include <iostream>  using namespace std;  int main(int argc, char\* argv[])  {  double a = 10;  double b = 2;  cout << "a + b = " << StaticMath::add(a, b) << endl;  cout << "a - b = " << StaticMath::sub(a, b) << endl;  cout << "a \* b = " << StaticMath::mul(a, b) << endl;  cout << "a / b = " << StaticMath::div(a, b) << endl;  StaticMath sm;  sm.print();  system("pause");  return 0;  } |

Linux下使用静态库，只需要在编译的时候，指定静态库的搜索路径（-L选项）、指定静态库名（不需要lib前缀和.a后缀，-l选项）。

# g++ TestStaticLibrary.cpp -**L../StaticLibrary -lstaticmath**

[](http://www.runoob.com/wp-content/uploads/2015/05/16201604-ae57e89353c74265a0f1d26e89ffdbd5.jpg)

-  -L：表示要连接的库所在目录

-  -l：指定链接时需要的动态库，编译器查找动态连接库时有隐含的命名规则，即在给出的名字前面加上lib，后面加上.a或.so来确定库的名称。

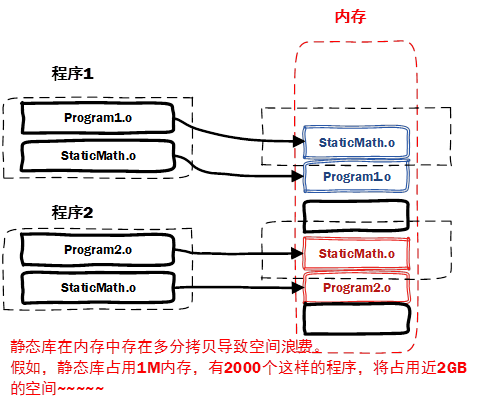
3.动态库

通过上面的介绍发现静态库，容易使用和理解，也达到了代码复用的目的，那为什么还需要动态库呢？

3.1.为什么还需要动态库？

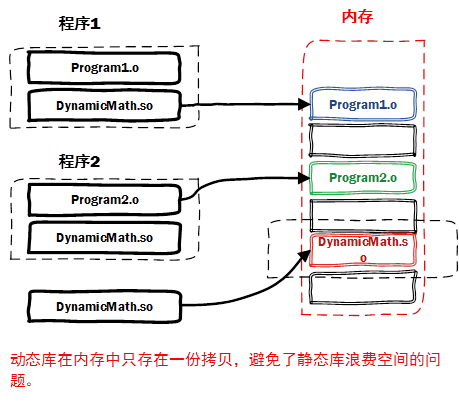
为什么需要动态库，其实也是静态库的特点导致。

- 空间浪费是静态库的一个问题。

[](http://www.runoob.com/wp-content/uploads/2015/05/16201613-1320ec57feb24583a6b1294421c323b8.png)

- 另一个问题是静态库对程序的更新、部署和发布页会带来麻烦。如果静态库liba.lib更新了，所以使用它的应用程序都需要重新编译、发布给用户（对于玩家来说，可能是一个很小的改动，却导致整个程序重新下载，全量更新）。

动态库在程序编译时并不会被连接到目标代码中，而是在程序运行是才被载入。不同的应用程序如果调用相同的库，那么在内存里只需要有一份该共享库的实例，规避了空间浪费问题。动态库在程序运行是才被载入，也解决了静态库对程序的更新、部署和发布页会带来麻烦。用户只需要更新动态库即可，增量更新。

[](http://www.runoob.com/wp-content/uploads/2015/05/16201613-110ca9e0fd684281b3ee6d9bd9bebd78.png)

动态库特点总结：

- 动态库把对一些库函数的链接载入推迟到程序运行的时期。

- 可以实现进程之间的资源共享。（因此动态库也称为共享库）

- 将一些程序升级变得简单。

- 甚至可以真正做到链接载入完全由程序员在程序代码中控制（显示调用）。

Window与Linux执行文件格式不同，在创建动态库的时候有一些差异。

- 在Windows系统下的执行文件格式是PE格式，动态库需要一个DllMain函数做出初始化的入口，通常在导出函数的声明时需要有\_declspec(dllexport)关键字。

- Linux下gcc编译的执行文件默认是ELF格式，不需要初始化入口，亦不需要函数做特别的声明，编写比较方便。

与创建静态库不同的是，不需要打包工具（ar、lib.exe），直接使用编译器即可创建动态库。

3.2.Linux下创建与使用动态库

3.2.1.linux动态库的命名规则

动态链接库的名字形式为 libxxx.so，前缀是lib，后缀名为".so"。

- 针对于实际库文件，每个共享库都有个特殊的名字"soname"。在程序启动后，程序通过这个名字来告诉动态加载器该载入哪个共享库。

- 在文件系统中，soname仅是一个链接到实际动态库的链接。对于动态库而言，每个库实际上都有另一个名字给编译器来用。它是一个指向实际库镜像文件的链接文件（lib+soname+.so）。

3.2.2.创建动态库（.so）

编写四则运算动态库代码：

DynamicMath.h头文件

#pragma once

class DynamicMath

{

public:

DynamicMath(void);

~DynamicMath(void);

static double add(double a, double b);//加法

static double sub(double a, double b);//减法

static double mul(double a, double b);//乘法

static double div(double a, double b);//除法

void print();

};

- 首先，生成目标文件，此时要加编译器选项-fpic

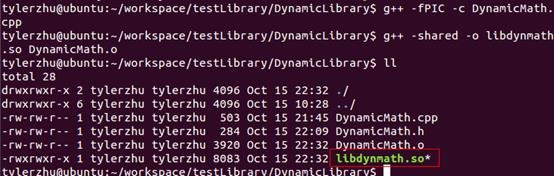
g++ -fPIC -c DynamicMath.cpp

-fPIC 创建与地址无关的编译程序（pic，position independent code），是为了能够在多个应用程序间共享。

- 然后，生成动态库，此时要加链接器选项-shared

g++ -shared -o libdynmath.so DynamicMath.o

-shared指定生成动态链接库。

[](http://www.runoob.com/wp-content/uploads/2015/05/16201614-02f4340b15664f8a83282fb9a27ca3e5.jpg)

其实上面两个步骤可以合并为一个命令：

g++ -fPIC -shared -o libdynmath.so DynamicMath.cpp

3.2.3.使用动态库

编写使用动态库的测试代码：

测试代码：

#include "../DynamicLibrary/DynamicMath.h"

#include <iostream>

using namespace std;

int main(int argc, char\* argv[])

{

double a = 10;

double b = 2;

cout << "a + b = " << DynamicMath::add(a, b) << endl;

cout << "a - b = " << DynamicMath::sub(a, b) << endl;

cout << "a \* b = " << DynamicMath::mul(a, b) << endl;

cout << "a / b = " << DynamicMath::div(a, b) << endl;

DynamicMath dyn;

dyn.print();

return 0;

}

引用动态库编译成可执行文件（跟静态库方式一样）：

g++ TestDynamicLibrary.cpp -L../DynamicLibrary -ldynmath

然后运行：./a.out，发现竟然报错了！！！

可能大家会猜测，是因为动态库跟测试程序不是一个目录，那我们验证下是否如此：

发现还是报错！！！那么，在执行的时候是如何定位共享库文件的呢？

1)当系统加载可执行代码时候，能够知道其所依赖的库的名字，但是还需要知道绝对路径。此时就需要系统动态载入器(dynamic linker/loader)。

2)对于elf格式的可执行程序，是由ld-linux.so\*来完成的，它先后搜索elf文件的 DT\_RPATH段—环境变量LD\_LIBRARY\_PATH—/etc/ld.so.cache文件列表—/lib/,/usr/lib 目录找到库文件后将其载入内存。

如何让系统能够找到它：

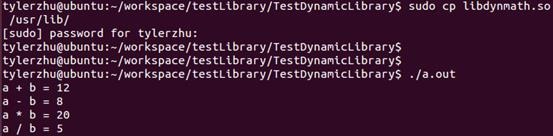
- 如果安装在/lib或者/usr/lib下，那么ld默认能够找到，无需其他操作。

- 如果安装在其他目录，需要将其添加到/etc/ld.so.cache文件中，步骤如下：

-- 编辑/etc/ld.so.conf文件，加入库文件所在目录的路径

-- 运行ldconfig ，该命令会重建/etc/ld.so.cache文件

我们将创建的动态库复制到/usr/lib下面，然后运行测试程序。

[](http://www.runoob.com/wp-content/uploads/2015/05/16201618-a32a866f00c84a74893fa2181fa27b11.jpg)

5.附件：Linux下库相关命令

5.1.g++(gcc)编译选项

- -shared ：指定生成动态链接库。

- -static ：指定生成静态链接库。

- -fPIC ：表示编译为位置独立的代码，用于编译共享库。目标文件需要创建成位置无关码， 念上就是在可执行程序装载它们的时候，它们可以放在可执行程序的内存里的任何地方。

- -L. ：表示要连接的库所在的目录。

- -l：指定链接时需要的动态库。编译器查找动态连接库时有隐含的命名规则，即在给出的名字前面加上lib，后面加上.a/.so来确定库的名称。

- -Wall ：生成所有警告信息。

- -ggdb ：此选项将尽可能的生成gdb 的可以使用的调试信息。

- -g ：编译器在编译的时候产生调试信息。

- -c ：只激活预处理、编译和汇编,也就是把程序做成目标文件(.o文件) 。

- -Wl,options ：把参数(options)传递给链接器ld 。如果options 中间有逗号,就将options分成多个选项,然后传递给链接程序。

5.2.nm命令

有时候可能需要查看一个库中到底有哪些函数，nm命令可以打印出库中的涉及到的所有符号。库既可以是静态的也可以是动态的。nm列出的符号有很多，常见的有三种：

- 一种是在库中被调用，但并没有在库中定义(表明需要其他库支持)，用U表示；

- 一种是库中定义的函数，用T表示，这是最常见的；

- 一种是所谓的弱态"符号，它们虽然在库中被定义，但是可能被其他库中的同名符号覆盖，用W表示。

$nm libhello.h

5.3.ldd命令

ldd命令可以查看一个可执行程序依赖的共享库，例如我们编写的四则运算动态库依赖下面这些库：

[](http://www.runoob.com/wp-content/uploads/2015/05/16201633-aebe0e8017354a9b93644043b9d013c6.jpg)

6.总结

二者的不同点在于代码被载入的时刻不同。

- 静态库在程序编译时会被连接到目标代码中，程序运行时将不再需要该静态库，因此体积较大。

- 动态库在程序编译时并不会被连接到目标代码中，而是在程序运行是才被载入，因此在程序运行时还需要动态库存在，因此代码体积较小。

动态库的好处是，不同的应用程序如果调用相同的库，那么在内存里只需要有一份该共享库的实例。带来好处的同时，也会有问题！如经典的DLL Hell问题，关于如何规避动态库管理问题，可以自行查找相关资料。

### make -j

单核就不要加j参数了，GCC编译的时候默认使用一核，所以多核CPU如果想利用好才加j参数

用make –j带一个参数，可以把项目在进行并行编译，比如在一台双核的机器上，完全可以用make -j4，让make最多运行4个编译命令同时执行，这样可以更有效地利用CPU资源。

### sprintf sscanf

int sprintf(char \*str, const char \*format, ...)

char str[80];

sprintf(str, "Pi 的值 = %f", M\_PI);

puts(str);

int sscanf(const char \*str, const char \*format, ...)

strcpy( dtm, "Saturday March 25 1989" );

sscanf( dtm, "%s %s %d %d", weekday, month, &day, &year );

针对检索数据的 format 字符串中的每个 format 说明符，应指定一个附加参数。如果您想要把 sscanf 操作的结果存储在一个普通的变量中，您应该在标识符前放置引用运算符（&），例如：

int n;

sscanf (str,"%d",&amp;n);

### 共享内存

<https://blog.csdn.net/ruizeng88/article/details/6702346>

<https://www.cnblogs.com/tp-16b/p/8987697.html>

### 进程间的五种通信方式

进程间通信（IPC，InterProcess Communication）是指在不同进程之间传播或交换信息。

IPC的方式通常有管道（包括无名管道和命名管道）、消息队列、信号量、共享内存、Socket、Streams等。其中 Socket和Streams支持不同主机上的两个进程IPC。

转：<https://www.cnblogs.com/zgq0/p/8780893.html>

### Linux 线程同步的三种方法

线程的最大特点是资源的共享性，但资源共享中的同步问题是多线程编程的难点。linux下提供了多种方式来处理线程同步，最常用的是互斥锁、条件变量和信号量。

转：<https://blog.csdn.net/zsf8701/article/details/7844316?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.edu_weight&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.edu_weight>

## makefile的编写指导

### Makefile的规则

在讲述这个Makefile之前，还是让我们先来粗略地看一看Makefile的规则。

target ... : prerequisites ...

command

...

...

target也就是一个目标文件，可以是Object File，也可以是执行文件。还可以是一个标签（Label），对于标签这种特性，在后续的“伪目标”章节中会有叙述。

prerequisites就是，要生成那个target所需要的文件或是目标。

command也就是make需要执行的命令。（任意的Shell命令）

例：

edit : main.o kbd.o command.o display.o

cc -o edit main.o kbd.o command.o display.o

main.o : main.c defs.h

cc -c main.c

kbd.o : kbd.c defs.h command.h

cc -c kbd.c

command.o : command.c defs.h command.h

cc -c command.c

display.o : display.c defs.h buffer.h

cc -c display.c

clean:

rm edit main.o kbd.o command.o display.o

### 通用makefile模板

CC=gcc #指定编译方式

CFLAGS=-Wall #显示所有警告

LIBS = -lm

TARGET=main #目标文件

#当前文件夹所有.c文件

SRCS = $(wildcard \*.c)

#对应.o

OBJS = $(SRCS:.c=.o)

#目标：依赖

$(TARGET):$(OBJS)

#$@：目标文件 $^：所有依赖文件

$(CC) -o $@ $^ $(LIBS)

clean:

rm -rf $(TARGET) $(OBJS)

%.o:%.c

#所有目标依赖对应，$<只对应第一个依赖

$(CC) $(CFLAGS) -o $@ -c $<

在这个makefile中，目标文件（target）包含：执行文件edit和中间目标文件（\*.o），依赖文件（prerequisites）就是冒号后面的那些.c 文件和.h 文件。

每一个.o 文件都有一组依赖文件，而这些.o 文件又是执行文件edit 的依赖文件。

在定义好依赖关系后，后续的那一行定义了如何生成目标文件的操作系统命令，一定要以一个Tab键作为开头。

make会比较targets文件和prerequisites文件的修改日期，如果prerequisites文件的日期要比targets文件的日期要新，或者target不存在的话，那么，make就会执行后续定义的命令。

### make的工作流程

在默认的方式下，也就是我们只输入make命令。那么，

1、make会在当前目录下找名字叫“Makefile”或“makefile”的文件。

2、如果找到，它会找文件中的第一个目标文件（target），在上面的例子中，他会找到“edit”这个文件，并把这个文件作为最终的目标文件。

3、如果edit文件不存在，或是edit所依赖的后面的.o 文件的文件修改时间要比edit这个文件新，那么，他就会执行后面所定义的命令来生成edit这个文件。

4、如果edit所依赖的.o 文件也存在，那么make会在当前文件中找目标为.o 文件的依赖性，如果找到则再根据那一个规则生成.o文件。（这有点像一个堆栈的过程）

5、当然，你的C文件和H文件是存在的啦，于是make会生成.o 文件，然后再用.o 文件生命make的终极任务，也就是执行文件edit了。

通过上述分析，我们知道，像clean这种，没有被第一个目标文件直接或间接关联，那么它后面所定义的命令将不会被自动执行，不过，我们可以显示要make执行。即命令——“make clean”，以此来清除所有的目标文件，以便重编译。

于是在我们编程中，如果这个工程已被编译过了，当我们修改了其中一个源文件，比如file.c，那么根据我们的依赖性，我们的目标file.o会被重编译（也就是在这个依性关系后面所定义的命令），于是file.o的文件也是最新的啦，于是file.o的文件修改时间要比edit要新，所以edit也会被重新链接了（详见edit目标文件后定义的命令）。

### makefile中使用变量$(objects)

对于重复出现的[.o]文件，我们声明一个变量，叫objects,OBJECTS,objs,OBJS,obj或是OBJ，我们在makefile一开始就定义：

objects = main.o kbd.o command.o display.o

于是，在makefile中以$(objects)的方式来使用这个变量，于是makefile可以写成：

objects = main.o kbd.o command.o display.o

edit:$(objects)

cc –o edit $(objects)

…

clean:

rm edit $(objects)

于是如果有新的.o文件加入，只需要简单修改一下objects变量。

### make自动推导

GNU的make很强大，它可以自动推导文件以及文件依赖关系后面的命令。于是我们就没必要去在每一个[.o]文件后都写上类似的命令，因为，我们的make会自动识别，并自己推导命令。

只要make看到一个[.o]文件，它就会自动的把[.c]文件加在依赖关系中，如果make找到一个whatever.o，那 么whatever.c，就会是whatever.o的依赖文件。并且cc -c whatever.c 也会被推导出来，于是，我们的makefile再也不用写得这么复杂。我们的是新的makefile又出炉了。

objects = main.o kbd.o command.o display.o \

insert.o search.o files.o utils.o

edit : $(objects)

cc -o edit $(objects)

main.o : defs.h

kbd.o : defs.h command.h

command.o : defs.h command.h

display.o : defs.h buffer.h

insert.o : defs.h buffer.h

search.o : defs.h buffer.h

files.o : defs.h buffer.h command.h

utils.o : defs.h

.PHONY : clean

clean :

rm edit $(objects)

这种方法，也就是make的“隐晦规则”。上面文件内容中，“.PHONY”表示，clean是个伪目标文件。

即然我们的make可以自动推导命令，那么我看到那堆[.o]和[.h]的依赖就有点不爽，那么多的重复的[.h]，能不能把其收拢起来

objects = main.o kbd.o command.o display.o \

insert.o search.o files.o utils.o

edit : $(objects)

cc -o edit $(objects)

$(objects) : defs.h

kbd.o command.o files.o : command.h

display.o insert.o search.o files.o : buffer.h

.PHONY : clean

clean :

rm edit $(objects)

### 清空目标文件的规则

更为稳健的做法是：

.PHONY : clean

clean :

-rm edit $(objects)

前面说过，.PHONY 意思表示clean是一个“伪目标”，。而在rm命令前面加了一个小减号的意思就是，也许某些文件出现问题，但不要管，继续做后面的事。当然，clean的规则不要放在文件的开头，不然，这就会变成make的默认目标，相信谁也不愿意这样。不成文的规矩是——“clean从来都是放在文件的最后”。

### makefile的组成

Makefile里主要包含了五个东西：显式规则、隐晦规则、变量定义、文件指示和注释。

1、显式规则。显式规则说明了，如何生成一个或多的的目标文件。这是由Makefile的书写者明显指出，要生成的文件，文件的依赖文件，生成的命令。

2、隐晦规则。由于我们的make有自动推导的功能，所以隐晦的规则可以让我们比较粗糙地简略地书写Makefile，这是由make所支持的。

3、变量的定义。在Makefile中我们要定义一系列的变量，变量一般都是字符串，这个有点你C语言中的宏，当Makefile被执行时，其中的变量都会被扩展到相应的引用位置上。

4、文件指示。其包括了三个部分，一个是在一个Makefile中引用另一个Makefile，就像C语言中的include一样；另一个是指根据某些情况指定Makefile中的有效部分，就像C语言中的预编译#if一样；还有就是定义一个多行的命令。有关这一部分的内容，我会在后续的部分中讲述。

5、注释。Makefile中只有行注释，和UNIX的Shell脚本一样，其注释是用“#”字符，这个就像C/C++中的“//”一样。如果你要在你的Makefile中使用“#”字符，可以用反斜框进行转义，如：“\#”。

最后，还值得一提的是，在Makefile中的命令，必须要以[Tab]键开始。

Makefile的文件名

默认的情况下，make命令会在当前目录下按顺序找寻文件名为“GNUmakefile”、

“makefile”、“Makefile”的文件，找到了解释这个文件。在这三个文件名中，最好使用“Makefile”这个文件名，因为，这个文件名第一个字符为大写，这样有一种显目的感觉。

### 引用其他的makefile

在Makefile使用include关键字可以把别的Makefile包含进来，这很像C语言的#include，被包含的文件会原模原样的放在当前文件的包含位置，include的语法是：

include<filename>

filename可以是当前操作系统Shell的文件模式（可以包含路径和通配符）。

在include前面可以有一些空字符，但是绝对不能以[Tab]键开始。include和<filename>可以用一个或多个空格隔开。举个例子，你有这样几个Makefile：a.mk、b.mk、c.mk,还有一个文件叫foo.make，以及一个变量$(bar)，其包含了e.mk和f.mk，那么下面的语句：

include foo.make \*.mk $(bar)

等价于：

include foo.make a.mk b.mk c.mk e.mk f.mk

### make的工作方式

GNU的make工作时的执行步骤入下：（想来其它的make也是类似）

1、读入所有的Makefile。

2、读入被include的其它Makefile。

3、初始化文件中的变量。

4、推导隐晦规则，并分析所有规则。

5、为所有的目标文件创建依赖关系链。

6、根据依赖关系，决定哪些目标要重新生成。

7、执行生成命令。

1-5步为第一个阶段，6-7为第二个阶段。第一个阶段中，如果定义的变量被使用了，那么，make会把其展开在使用的位置。但make并不会完全马上展开，make使用的是拖延战术，如果变量出现在依赖关系的规则中，那么仅当这条依赖被决定要使用了，变量才会在其内部展开。

makefile书写规则

规则包含两个部分，一个是依赖关系，一个是生成目标的方法。

command是命令行，如果其不与“target:prerequisites”在一行，那么，必须以[Tab键]开头，如果和prerequisites在一行，那么可以用分号做为分隔。

如果命令太长，你可以使用反斜框（‘\’）作为换行符。

一般来说，make会以UNIX的标准Shell，也就是/bin/sh来执行命令。

### makefile规则中使用通配符

make支持三各通配符：“\*”，“?”和“[...]”。这是和Unix的B-Shell是相同的。

objects = \*.o

并不是说[\*.o]会展开，不！objects的值就是“\*.o”。Makefile中的变量其实就是C/C++中的宏。如果你要让通配符在变量中展开，也就是让objects的值是所有[.o]的文件名的集合，那么，你可以这样：

objects := $(wildcard \*.o)

### makefile文件搜寻

当make需要去找寻文件的依赖关系时，你可以在文件前加上路径，但最好的方法是把一个路径告诉make，让make在自动去找。

VPATH = src:../headers

上面的的定义指定两个目录，“src”和“../headers”，make会按照这个顺序进行搜索。目录由“冒号”分隔。（当然，当前目录永远是最高优先搜索的地方）

另一个设置文件搜索路径的方法是使用make的“vpath”关键字（注意，它是全小写的），这不是变量，这是一个make的关键字，这和上面提到的那个VPATH变量很类似，但是它更为灵活。

# Makefile VPATH用法

CC = gcc

OBJ = main.o foo.o bar.o

CFLAGS = -I include

VPATH = src:include

prog:$(OBJ)

$(CC) -o $@ $(OBJ)

%.o:%.c

$(CC) $(CFLAGS) -c $<

main.o:foo.h bar.h

.PHONY:clean

clean:

-rm -f prog $(OBJ)

# Makefile vpath用法

CC = gcc

OBJ = main.o foo.o bar.o

CPPFLAGS= -Iinclude

#VPATH = src:include

vpath %.h include

vpath %.c src

prog:$(OBJ)

$(CC) -o $@ $(OBJ)

%.o:%.c

$(CC) $(CPPFLAGS) -c $<

main.o:foo.h bar.h

.PHONY:clean

clean:

-rm -f prog $(OBJ)

注：将CFLAGS改为CPPFLAGS，因为-I选项应该是预编译中的选项，而不是编译选项。

### 伪目标

最早先的一个例子中，我们提到过一个“clean”的目标，这是一个“伪目标”

clean:

rm \*.o temp

我们并不生成“clean”这个文件。“伪目标”并不是一个文件，只是一个标签，由于“伪目标”不是文件，所以make无法生成它的依赖关系和决定它是否要执行。我们只有通过显示地指明这个“目标”才能让其生效。当然，“伪目标”的取名不能和文件名重名，不然其就失去了“伪目标”的意义了。

当然，为了避免和文件重名的这种情况，我们可以使用一个特殊的标记“.PHONY”来显示地指明一个目标是“伪目标”，向make说明，不管是否有这个文件，这个目标就是“伪目标”。

.PHONY : clean

只要有这个声明，不管是否有“clean”文件，要运行“clean”这个目标，只有“make clean”这样。于是整个过程可以这样写：

.PHONY: clean

clean:

rm \*.o temp

伪目标一般没有依赖的文件。但是，我们也可以为伪目标指定所依赖的文件。伪目标同样可以作为“默认目标”，只要将其放在第一个。

如果你的Makefile需要一口气生成若干个可执行文件，但你只想简单地敲一个make完事

所有的目标文件都写在一个Makefile中，那么你可以使用“伪目标”这个特性：

all : prog1 prog2 prog3

.PHONY : all

prog1 : prog1.o utils.o

cc -o prog1 prog1.o utils.o

prog2 : prog2.o

cc -o prog2 prog2.o

prog3 : prog3.o sort.o utils.o

cc -o prog3 prog3.o sort.o utils.o

我们知道，Makefile中的第一个目标会被作为其默认目标。我们声明了一个“all”的伪目标，其依赖于其它三个目标。

### 多目标

Makefile的规则中的目标可以不止一个，其支持多目标，有可能我们的多个目标同时依赖于一个文件，并且其生成的命令大体类似。于是我们就能把其合并起来。当然，多个目标的生成规则的执行命令是同一个，这可能会可我们带来麻烦，不过好在我们的可以使用一个自动化变量“$@”（关于自动化变量，将在后面讲述），这个变量表示着目前规则中所有的目标的集合

$@:表示目标文件。

bigoutput littleoutput : text.g

generate text.g -$(subst output,,$@) > $@

上述规则等价于：

bigoutput : text.g

generate text.g -big > bigoutput

littleoutput : text.g

generate text.g -little > littleoutput

其中，-$(subst output,,$@)中的“$”表示执行一个Makefile的函数，函数名为subst，后面的为参数。关于函数，将在后面讲述。这里的这个函数是截取字符串的意思，“$@”表示目标的集合，就像一个数组，“$@”依次取出目标，并执于命令。

subst用法

原型

$(subst ;,;,;)

名称：字符串替换函数——subst。

功能：把字串;中的;字符串替换成;。

返回：函数返回被替换过后的字符串。

例子

现在有一个变量CONF值为[MacRelease,MacDebug],现在要把Mac去掉.

代码为: ${substr Mac,,${CONF}}

结果是:Release或Debug

### 静态模式

静态模式可以更加容易地定义多目标的规则，可以让我们的规则变得更加的有弹性和灵活。我们还是先来看一下语法：

<targets ...>: <target-pattern>: <prereq-patterns ...>

<commands>

...

targets定义了一系列的目标文件，可以有通配符。是目标的一个集合。

target-parrtern是指明了targets的模式，也就是的目标集模式。

prereq-parrterns是目标的依赖模式，它对target-parrtern形成的模式再进行一次依赖目标的定义。

如果我们的<target-parrtern>定义成“%.o”，意思是我们的<target>集合中都是以“.o”结尾的，而如果我们的<prereq-parrterns>定义成“%.c”，意思是对<target-parrtern>所形成的目标集进行二次定义，其计算方法是，取<target-parrtern>模式中的“%”（也就是去掉了[.o]这个结尾），并为其加上[.c]这个结尾，形成的新集合。

看一个例子：

objects = foo.o bar.o

all: $(objects)

$(objects): %.o: %.c

$(CC) -c $(CFLAGS) $< -o $@

上面的例子中，指明了我们的目标从$object中获取，“%.o”表明要所有以“.o”结尾的目标，也就是“foo.o bar.o”，也就是变量$object集合的模式，而依赖模式“%.c”则取模式“%.o”的“%”，也就是“foo bar”，并为其加下“.c”的后缀，于是，我们的依赖目标就是“foo.c bar.c”。

而命令中的“$<”和“$@”则是自动化变量，“$<”表示所有的依赖目标集（也就是“foo.c bar.c”），“$@”表示目标集（也就是“foo.o bar.o”）。于是，上面的规则展开后等价于下面的规则：

foo.o : foo.c

$(CC) -c $(CFLAGS) foo.c -o foo.o

bar.o : bar.c

$(CC) -c $(CFLAGS) bar.c -o bar.o

试想，如果我们的“%.o”有几百个，那种我们只要用这种很简单的“静态模式规则”就可以写完一堆规则，实在是太有效率了。“静态模式规则”的用法很灵活，如果用得好，那会一个很强大的功能。

### 自动生成依赖性

大多数的C/C++编译器都支持一个“-M”的选项，即自动找寻源文件中包含的头文件，并生成一个依赖关系。

于是由编译器自动生成的依赖关系，这样一来，你就不必再手动书写若干文件的依赖关系，而由编译器自动生成了。需要提醒一句的是，如果你使用GNU的C/C++编译器，你得用“-MM”参数，不然，“-M”参数会把一些标准库的头文件也包含进来。

GNU组织建议把编译器为每一个源文件的自动生成的依赖关系放到一个文件中，为每一个“name.c”的文件都生成一个“name.d”的Makefile文件，[.d]文件中就存放对应[.c]文件的依赖关系。

我们可以写出[.c]文件和[.d]文件的依赖关系，并让make自动更新或自成[.d]文件，并把其包含在我们的主Makefile中，这样，我们就可以自动化地生成每个文件的依赖关系了。

这里，我们给出了一个模式规则来产生[.d]文件：

%.d: %.c

@set -e; rm -f $@; \

$(CC) -M $(CPPFLAGS) $< > $@.$$$$; \

sed 's,\($\*\)\.o[ :]\*,\1.o $@ : ,g' < $@.$$$$ > $@; \

rm -f $@.$$$$

所有的[.d]文件依赖于[.c]文件

“rm -f $@”的意思是删除所有的目标，也就是[.d]文件，第二行的意思是，为每个依赖文件“$<”，也就是[.c]文件生成依赖文件，“$@”表示模式“%.d”文件，如果有一个C文件是name.c，那么“%”就是“name”，“$$$$”意为一个随机编号，第二行生成的文件有可能是“name.d.12345”

第三行使用sed命令做了一个替换，关于sed命令的用法请参看相关的使用文档。

第四行就是删除临时文件。

总而言之，这个模式要做的事就是在编译器生成的依赖关系中加入[.d]文件的依赖，即把依

赖关系：

main.o : main.c defs.h

转成：

main.o main.d : main.c defs.h

于是，我们的[.d]文件也会自动更新了，并会自动生成了，当然，你还可以在这个[.d]文件中加入的不只是依赖关系，包括生成的命令也可一并加入，让每个[.d]文件都包含一个完赖的规则。

一旦我们完成这个工作，接下来，我们就要把这些自动生成的规则放进我们的主Makefile中。我们可以使用Makefile的“include”命令，来引入别的Makefile文件（前面讲过），例如：

sources = foo.c bar.c

include $(sources:.c=.d)

上述语句中的“$(sources:.c=.d)”中的“.c=.d”的意思是做一个替换，把变量$(sources)所有[.c]的字串都替换成[.d]，关于这个“替换”的内容，在后面我会有更为详细的讲述。当然，你得注意次序，因为include是按次来载入文件，最先载入的[.d]文件中的目标会成为默认目标。

### sed函数

例：

sed 's,\(main\)\.o[ :]\*,\1.o main.d : ,g' < main.d.temp > main.d

其中：

sed 's,main\.o[ :]\*,\1.o main.d : ,g'，是sed命令。

< main.d.temp，指示sed命令从临时文件main.d.temp读取输入，作为命令的来源字符串。

> main.d，把行内替换结果输出到最终文件main.d。

这条sed命令的结构是s/match/replace/g。有时为了清晰，可以把每个/写成逗号，即这里的格式s,match,replace,g。

该命令表示把源串内的match都替换成replace，s指示match可以是正则表达式。

g表示把每行内所有match都替换，如果去掉g，则只有每行的第1处match被替换(实际上不需要g，因为一个.d文件中，只会在开头有一个main.o:)。

总体来说match用来匹配'main.o :'这样的串。

这里的replace是\1.o main.d :，其中\1会被替换为前面第1个和括起的内容，即main，这样replace值为main.o main.d :

这样该sed命令就实现了把main.o :替换为main.o main.d :的目的。

这两行实现了把临时文件main.d.temp的内容main.o : main.c command.h改为main.o main.d : main.c command.h，并存入main.d文件的功能。

书写命令

每条规则中的命令和操作系统Shell的命令行是一致的。make会一按顺序一条一条的执行命令，每条命令的开头必须以[Tab]键开头，除非，命令是紧跟在依赖规则后面的分号后的。在命令行之间中的空格或是空行会被忽略，但是如果该空格或空行是以Tab键开头的，那么make会认为其是一个空命令。

### makefile显示命令

通常，make会把其要执行的命令行在命令执行前输出到屏幕上。当我们用“@”字符在命令行前，那么，这个命令将不被make显示出来，最具代表性的例子是，我们用这个功能来像屏幕显示一些信息。如：

@echo 正在编译XXX模块......

当make执行时，会输出“正在编译XXX模块......”字串，但不会输出命令，如果没有“@”，那么，make将输出：

echo 正在编译XXX模块......

正在编译XXX模块......

### makefile命令执行

Makefile文件中内容：

exec:

cd /home/lqc;pwd

执行”make exec”, cd就起作用了，pwd会打印出“/home/lqc”。

exec:

@ cd /home/lqc;pwd

不输出执行命令过程。

### makefile命令出错

为了做到这一点，忽略命令的出错，我们可以在Makefile的命令行前加一个减号“-”（在Tab键之后），标记为不管命令出不出错都认为是成功的。如：

clean:

-rm -f \*.o

还有一个全局的办法是，给make加上“-i”或是“--ignore-errors”参数，那么，Makefile中所有命令都会忽略错误。而如果一个规则是以“.IGNORE”作为目标的，那么这个规则中的所有命令将会忽略错误。这些是不同级别的防止命令出错的方法，你可以根据你的不同喜欢设置。

### 嵌套执行make

在一些大的工程中，我们会把我们不同模块或是不同功能的源文件放在不同的目录中，我们可以在每个目录中都书写一个该目录的Makefile，这有利于让我们的Makefile变得更加地简洁，而不至于把所有的东西全部写在一个Makefile中，这样会很难维护我们的Makefile，这个技术对于我们模块编译和分段编译有着非常大的好处。

例如，我们有一个子目录叫subdir，这个目录下有个Makefile文件，来指明了这个目录下文件的编译规则。那么我们总控的Makefile可以这样书写：

subsystem:

cd subdir && $(MAKE)

其等价于：

subsystem:

$(MAKE) -C subdir

定义$(MAKE)宏变量的意思是，也许我们的make需要一些参数，所以定义成一个变量比较利于维护。这两个例子的意思都是先进入“subdir”目录，然后执行make命令。

如果你要传递变量到下级Makefile中，那么你可以使用这样的声明：

export <variable ...>

如果你不想让某些变量传递到下级Makefile中，那么你可以这样声明：

unexport <variable ...>

需要注意的是，有两个变量，一个是SHELL，一个是MAKEFLAGS，这两个变量不管你是否export，其总是要传递到下层Makefile中，特别是MAKEFILES变量，其中包含了make的参数信息，如果我们执行“总控Makefile”时有make参数或是在上层Makefile中定义了这个变量，那么MAKEFILES变量将会是这些参数，并会传递到下层Makefile中，这是一个系统级的环境变量。

还有一个在“嵌套执行”中比较有用的参数，“-w”或是“--print-directory”会在make的过程中输出一些信息，让你看到目前的工作目录。比如，如果我们的下级make目录是“/home/hchen/gnu/make”，如果我们使用“make -w”来执行，那么当进入该目录时，我们

会看到：

make: Entering directory `/home/hchen/gnu/make'.

而在完成下层make后离开目录时，我们会看到：

make: Leaving directory `/home/hchen/gnu/make'

当你使用“-C”参数来指定make下层Makefile时，“-w”会被自动打开的。如果参数中有“-s”（“--slient”）或是“--no-print-directory”，那么，“-w”总是失效的。

### makefile使用变量

传统的Makefile的变量名是全大写的命名方式，但我推荐使用大小写搭配的变量名，如：MakeFlags。这样可以避免和系统的变量冲突，而发生意外的事情。

变量在声明时需要给予初值，而在使用时，需要给在变量名前加上“$”符号，但最好用小括号“（）”或是大括号“{}”把变量给包括起来。如果你要使用真实的“$”字符，那么你需要用“$$”来表示。

另外，给变量加上括号完全是为了更加安全地使用这个变量。

foo = $(bar)

bar = $(ugh)

ugh = Huh?

all:

echo $(foo)

我们执行“make all”将会打出变量$(foo)的值是“Huh?”（$(foo)的值是$(bar)，$(bar)的值是$(ugh)，$(ugh)的值是“Huh?”）可见，变量是可以使用后面的变量来定义的。

A = $(B)

B = $(A)

这会让make陷入无限的变量展开过程中去，当然，我们的make是有能力检测这样的定义，并会报错。

为了避免上面的这种方法，我们可以使用make中的另一种用变量来定义变量的方法。这种方法使用的是“:=”操作符，如：

x := foo

y := $(x) bar

x := later

其等价于：

y := foo bar

x := later

值得一提的是，这种方法，前面的变量不能使用后面的变量，只能使用前面已定义好了的变量。如果是这样：

y := $(x) bar

x := foo

那么，y的值是“bar”，而不是“foo bar”。

### makefile :=

:= 意思就是

如果这个变量是空的或者未赋初值,则对其进行赋值. 否则, 就不赋值.

### makefile ?=

FOO ?= bar

其含义是，如果FOO没有被定义过，那么变量FOO的值就是“bar”，如果FOO先前被定义过，那么这条语将什么也不做，其等价于：

ifeq ($(origin FOO), undefined)

FOO = bar

endif

### makefile变量的替换

我们可以替换变量中的共有的部分，其格式是“$(var:a=b)”或是“${var:a=b}”，其意思是，把变量“var”中所有以“a”字串“结尾”的“a”替换成“b”字串。这里的“结尾”意思是“空格”或是“结束符”。

还是看一个示例吧：

foo := a.o b.o c.o

bar := $(foo:.o=.c)

这个示例中，我们先定义了一个“$(foo)”变量，而第二行的意思是把“$(foo)”中所有以“.o”字串“结尾”全部替换成“.c”，所以我们的“$(bar)”的值就是“a.c b.c c.c”。

另外一种变量替换的技术是以“静态模式”（参见前面章节）定义的，如：

foo := a.o b.o c.o

bar := $(foo:%.o=%.c)

这依赖于被替换字串中的有相同的模式，模式中必须包含一个“%”字符，这个例子同样让$(bar)变量的值为“a.c b.c c.c”。

把变量的值再当成变量

x = y

y = z

a := $($(x))

在这个例子中，$(x)的值是“y”，所以$($(x))就是$(y)，于是$(a)的值就是“z”。（注意，是“x=y”，而不是“x=$(y)”）

first\_second = Hello

a = first

b = second

all = $($a\_$b)

这里的“$a\_$b”组成了“first\_second”，于是，$(all)的值就是“Hello”

### makefile追加变量值

我们可以使用“+=”操作符给变量追加值，如：

objects = main.o foo.o bar.o utils.o

objects += another.o

于是，我们的$(objects)值变成：“main.o foo.o bar.o utils.o another.o”（another.o被追加进去了）

### makefile多行命令

还有一种设置变量值的方法是使用define关键字。使用define关键字设置变量的值可以有换行，这有利于定义一系列的命令

define指示符后面跟的是变量的名字，而重起一行定义变量的值，定义是以endef关键字结束。其工作方式和“=”操作符一样。变量的值可以包含函数、命令、文字，或是其它变量。因为命令需要以[Tab]键开头，所以如果你用define定义的命令变量中没有以[Tab]键开头，那么make就不会把其认为是命令。

define two-lines

echo foo

echo $(bar)

endef

### makefile define关键字

make解释器提供了一系列的函数供makefile调用

在makefile中支持自定义函数实现，并调用执行

通过define关键字实现自定义函数

bar = too

define two-lines

echo foo

echo $(bar)

endef

exec:

$(call two-lines)

执行：make exec

lqc@lqc-vm:cppProj$ make exec

echo foo

foo

echo too

too

### makefile环境变量

make运行时的系统环境变量可以在make开始运行时被载入到Makefile文件中，但是如果Makefile中已定义了这个变量，或是这个变量由make命令行带入，那么系统的环境变量的值将被覆盖。

因此，如果我们在环境变量中设置了“CFLAGS”环境变量，那么我们就可以在所有的Makefile中使用这个变量了。这对于我们使用统一的编译参数有比较大的好处。如果Makefile中定义了CFLAGS，那么则会使用Makefile中的这个变量，如果没有定义则使用系统环境变量的值，一个共性和个性的统一，很像“全局变量”和“局部变量”的特性。

### makefile目标变量

我们同样可以为某个目标设置局部变量，这种变量被称为“Target-specific Variable”，它可以和“全局变量”同名，因为它的作用范围只在这条规则以及连带规则中，所以其值也只在作用范围内有效。而不会影响规则链以外的全局变量的值。

其语法是：

<target ...> : <variable-assignment>

<target ...> : overide <variable-assignment>

<variable-assignment>可以是前面讲过的各种赋值表达式，如“=”、“:=”、“+=”或是“？=”。第二个语法是针对于make命令行带入的变量，或是系统环境变量。

prog : CFLAGS = -g

prog : prog.o foo.o bar.o

$(CC) $(CFLAGS) prog.o foo.o bar.o

prog.o : prog.c

$(CC) $(CFLAGS) prog.c

foo.o : foo.c

$(CC) $(CFLAGS) foo.c

bar.o : bar.c

$(CC) $(CFLAGS) bar.c

在这个示例中，不管全局的$(CFLAGS)的值是什么，在prog目标，以及其所引发的所有规则中（prog.o foo.o bar.o的规则），$(CFLAGS)的值都是“-g”

### makefile模式变量

在GNU的make中，还支持模式变量（Pattern-specific Variable），通过上面的目标变量中，我们知道，变量可以定义在某个目标上。模式变量的好处就是，我们可以给定一种“模式”，可以把变量定义在符合这种模式的所有目标上。

我们知道，make的“模式”一般是至少含有一个“%”的，所以，我们可以以如下方式给所有以[.o]结尾的目标定义目标变量：

%.o : CFLAGS = -O

同样，模式变量的语法和“目标变量”一样：

<pattern ...> : <variable-assignment>

<pattern ...> : override <variable-assignment>

override同样是针对于系统环境传入的变量，或是make命令行指定的变量。

### makefile使用条件判断

例：

下面的例子，判断$(CC)变量是否“gcc”，如果是的话，则使用GNU函数编译目标。

libs\_for\_gcc = -lgnu

normal\_libs =

foo: $(objects)

ifeq ($(CC),gcc)

$(CC) -o foo $(objects) $(libs\_for\_gcc)

else

$(CC) -o foo $(objects) $(normal\_libs)

endif

ifeq的意思表示条件语句的开始，并指定一个条件表达式，表达式包含两个参数，以逗号分隔，表达式以圆括号括起。else表示条件表达式为假的情况。endif表示一个条件语句的结束，任何一个条件表达式都应该以endif结束。

我们还可以把上面的那个例子写得更简洁一些：

libs\_for\_gcc = -lgnu

normal\_libs =

ifeq ($(CC),gcc)

libs=$(libs\_for\_gcc)

else

libs=$(normal\_libs)

endif

foo: $(objects)

$(CC) -o foo $(objects) $(libs)

条件表达式的语法为：

<conditional-directive>

<text-if-true>

endif

以及：

<conditional-directive>

<text-if-true>

else

<text-if-false>

endif

其中<conditional-directive>表示条件关键字，如“ifeq”。这个关键字有四个。

第一个是我们前面所见过的“ifeq”

ifeq (<arg1>, <arg2>)

第二个条件关键字是“ifneq”。语法是：

ifneq (<arg1>, <arg2>)

第三个条件关键字是“ifdef”。语法是：

ifdef <variable-name>

示例一：

bar =

foo = $(bar)

ifdef foo

frobozz = yes

else

frobozz = no

endif

示例二：

foo =

ifdef foo

frobozz = yes

else

frobozz = no

endif

第一个例子中，“$(frobozz)”值是“yes”，第二个则是“no”。

第四个条件关键字是“ifndef”。其语法是：

ifndef <variable-name>

这个我就不多说了，和“ifdef”是相反的意思。

在<conditional-directive>这一行上，多余的空格是被允许的，但是不能以[Tab]键做为开始（不然就被认为是命令）。而注释符“#”同样也是安全的。“else”和“endif”也一样，只要不是以[Tab]键开始就行了。

### makefile 使用函数

函数调用，很像变量的使用，也是以“$”来标识的，其语法如下：

$(<function> <arguments>)

或是

${<function> <arguments>}

这里，<function>就是函数名，make支持的函数不多。<arguments>是函数的参数，参数间以逗号“,”分隔，而函数名和参数之间以“空格”分隔。函数调用以“$”开头，以圆括号或花括号把函数名和参数括起。感觉很像一个变量，是不是？函数中的参数可以使用变量，为了风格的统一，函数和变量的括号最好一样，如使用“$(subst a,b,$(x))”这样的形式，而不是“$(subst a,b,${x})”的形式。因为统一会更清楚，也会减少一些不必要的麻烦。

还是来看一个示例：

comma:= ,

empty:=

space:= $(empty) $(empty)

foo:= a b c

bar:= $(subst $(space),$(comma),$(foo))

在这个示例中，$(comma)的值是一个逗号。$(space)使用了$(empty)定义了一个空格，

$(foo)的值是“a b c”，$(bar)的定义用，调用了函数“subst”，这是一个替换函数，这个函数有三个参数，第一个参数是被替换字串，第二个参数是替换字串，第三个参数是替换操作作用的字串。这个函数也就是把$(foo)中的空格替换成逗号，所以$(bar)的值是“a,b,c”。

### makefile字符串处理函数

$(subst <from>,<to>,<text>)

名称：字符串替换函数——subst。

功能：把字串<text>中的<from>字符串替换成<to>。

返回：函数返回被替换过后的字符串。

$(patsubst <pattern>,<replacement>,<text>)

名称：模式字符串替换函数——patsubst。

功能：查找<text>中的单词（单词以“空格”、“Tab”或“回车”“换行”分隔）是否符合模式<pattern>，如果匹配的话，则以<replacement>替换。这里，<pattern>可以包括通配符“%”，表示任意长度的字串。如果<replacement>中也包含“%”，那么，<replacement>中的这个“%”将是<pattern>中的那个“%”所代表的字串。（可以用“\”来转义，以“\%”来表示真实含义的“%”字符）返回：函数返回被替换过后的字符串。

示例：

$(patsubst %.c,%.o,x.c.c bar.c)

把字串“x.c.c bar.c”符合模式[%.c]的单词替换成[%.o]，返回结果是“x.c.o bar.o”

$(strip <string>)

名称：去空格函数——strip。

功能：去掉<string>字串中开头和结尾的空字符。

返回：返回被去掉空格的字符串值。

示例：

$(strip a b c )

把字串“a b c ”去到开头和结尾的空格，结果是“a b c”。

$(findstring <find>,<in>)

名称：查找字符串函数——findstring。

功能：在字串<in>中查找<find>字串。

返回：如果找到，那么返回<find>，否则返回空字符串。

示例：

$(findstring a,a b c)

$(findstring a,b c)

第一个函数返回“a”字符串，第二个返回“”字符串（空字符串）

$(filter <pattern...>,<text>)

名称：过滤函数——filter。

功能：以<pattern>模式过滤<text>字符串中的单词，保留符合模式<pattern>的单词。可以有多个模式。

返回：返回符合模式<pattern>的字串。

示例：

sources := foo.c bar.c baz.s ugh.h

foo: $(sources)

cc $(filter %.c %.s,$(sources)) -o foo

$(filter %.c %.s,$(sources))返回的值是“foo.c bar.c baz.s”。

$(filter-out <pattern...>,<text>)

名称：反过滤函数——filter-out。

功能：以<pattern>模式过滤<text>字符串中的单词，去除符合模式<pattern>的单词。可以有多个模式。

返回：返回不符合模式<pattern>的字串。

示例：

objects=main1.o foo.o main2.o bar.o

mains=main1.o main2.o

$(filter-out $(mains),$(objects)) 返回值是“foo.o bar.o”。

$(sort <list>)

名称：排序函数——sort。

功能：给字符串<list>中的单词排序（升序）。

返回：返回排序后的字符串。

示例：$(sort foo bar lose)返回“bar foo lose” 。

备注：sort函数会去掉<list>中相同的单词。

$(word <n>,<text>)

名称：取单词函数——word。

功能：取字符串<text>中第<n>个单词。（从一开始）

返回：返回字符串<text>中第<n>个单词。如果<n>比<text>中的单词数要大，那么返回空字

符串。

示例：$(word 2, foo bar baz)返回值是“bar”。

$(wordlist <s>,<e>,<text>)

名称：取单词串函数——wordlist。

功能：从字符串<text>中取从<s>开始到<e>的单词串。<s>和<e>是一个数字。

返回：返回字符串<text>中从<s>到<e>的单词字串。如果<s>比<text>中的单词数要大，那么返回空字符串。如果<e>大于<text>的单词数，那么返回从<s>开始，到<text>结束的单词串。

示例： $(wordlist 2, 3, foo bar baz)返回值是“bar baz”。

$(words <text>)

名称：单词个数统计函数——words。

功能：统计<text>中字符串中的单词个数。

返回：返回<text>中的单词数。

示例：$(words, foo bar baz)返回值是“3”。

备注：如果我们要取<text>中最后的一个单词，我们可以这样：$(word $(words

<text>),<text>)。

$(firstword <text>)

名称：首单词函数——firstword。

功能：取字符串<text>中的第一个单词。

返回：返回字符串<text>的第一个单词。

示例：$(firstword foo bar)返回值是“foo”。

备注：这个函数可以用word函数来实现：$(word 1,<text>)。

### makefile文件名操作函数

$(dir <names...>)

名称：取目录函数——dir。

功能：从文件名序列<names>中取出目录部分。目录部分是指最后一个反斜杠（“/”）之前的部分。如果没有反斜杠，那么返回“./”。

返回：返回文件名序列<names>的目录部分。

示例：$(dir src/foo.c hacks)返回值是“src/ ./”。

$(notdir <names...>)

名称：取文件函数——notdir。

功能：从文件名序列<names>中取出非目录部分。非目录部分是指最后一个反斜杠（“/”）之后的部分。

返回：返回文件名序列<names>的非目录部分。

示例： $(notdir src/foo.c hacks)返回值是“foo.c hacks”。

$(suffix <names...>)

名称：取后缀函数——suffix。

功能：从文件名序列<names>中取出各个文件名的后缀。

返回：返回文件名序列<names>的后缀序列，如果文件没有后缀，则返回空字串。

示例：$(suffix src/foo.c src-1.0/bar.c hacks)返回值是“.c .c”。

$(basename <names...>)

名称：取前缀函数——basename。

功能：从文件名序列<names>中取出各个文件名的前缀部分。

返回：返回文件名序列<names>的前缀序列，如果文件没有前缀，则返回空字串。

示例：$(basename src/foo.c src-1.0/bar.c hacks)返回值是“src/foo src-1.0/bar hacks”。

$(addsuffix <suffix>,<names...>)

名称：加后缀函数——addsuffix。

功能：把后缀<suffix>加到<names>中的每个单词后面。

返回：返回加过后缀的文件名序列。

示例：$(addsuffix .c,foo bar)返回值是“foo.c bar.c”。

$(addprefix <prefix>,<names...>)

名称：加前缀函数——addprefix。

功能：把前缀<prefix>加到<names>中的每个单词后面。

返回：返回加过前缀的文件名序列。

示例：$(addprefix src/,foo bar)返回值是“src/foo src/bar”。

$(join <list1>,<list2>)

名称：连接函数——join。

功能：把<list2>中的单词对应地加到<list1>的单词后面。如果<list1>的单词个数要比<list2>的多，那么，<list1>中的多出来的单词将保持原样。如果<list2>的单词个数要比<list1>多，那么，<list2>多出来的单词将被复制到<list2>中。

返回：返回连接过后的字符串。

示例：$(join aaa bbb , 111 222 333)返回值是“aaa111 bbb222 333”。

### makefile foreach函数

$(foreach <var>,<list>,<text>)

这个函数的意思是，把参数<list>中的单词逐一取出放到参数<var>所指定的变量中，然后再执行<text>所包含的表达式。每一次<text>会返回一个字符串，循环过程中，<text>的所返回的每个字符串会以空格分隔，最后当整个循环结束时，<text>所返回的每个字符串所组成的整个字符串（以空格分隔）将会是foreach函数的返回值。

names := a b c d

files := $(foreach n,$(names),$(n).o)

上面的例子中，$(name)中的单词会被挨个取出，并存到变量“n”中，“$(n).o”每次根据“$(n)”计算出一个值，这些值以空格分隔，最后作为foreach函数的返回，所以，$(files)的值是“a.o b.o c.o d.o”。

注意，foreach中的<var>参数是一个临时的局部变量，foreach函数执行完后，参数<var>的变量将不在作用，其作用域只在foreach函数当中。

### makefile call函数

call函数是唯一一个可以用来创建新的参数化的函数。你可以写一个非常复杂的表达式，这个表达式中，你可以定义许多参数，然后你可以用call函数来向这个表达式传递参数。其语法是：

$(call <expression>,<parm1>,<parm2>,<parm3>...)

当make执行这个函数时，<expression>参数中的变量，如$(1)，$(2)，$(3)等，会被参数<parm1>，<parm2>，<parm3>依次取代。而<expression>的返回值就是call函数的返回值。

例如：

reverse = $(1) $(2)

foo = $(call reverse,a,b)

那么，foo的值就是“a b”。当然，参数的次序是可以自定义的，不一定是顺序的，如：

reverse = $(2) $(1)

foo = $(call reverse,a,b)

此时的foo的值就是“b a”。

### makefile origin函数

origin函数不像其它的函数，他并不操作变量的值，他只是告诉你你的这个变量是哪里来的？其语法是：

$(origin <variable>)

注意，<variable>是变量的名字，不应该是引用。所以你最好不要在<variable>中使用“$”字符。Origin函数会以其返回值来告诉你这个变量的“出生情况”

### makefile之override

override指示符

通常在执行 make 时,如果通过命令行定义了一个变量,那么它将替代在 Makefile中出现的同名变量的定义。  
就是说,对于一个在 Makefile 中使用常规方式(使用“=”、“:=”或者“define”)定义的变量,我们可以在执行 make 时通过命令行方式重新指定这个变量的值,命令行指定的值将替代出现在 Makefile 中此变量的值。  
如果不希望命令行指定的变量值替代在 Makefile 中的变量定义,那么我们需要在 Makefile 中使用指示符“override”来对这个变量进行声明。

override作用

1. 保护makefile中定义的变量的值;
2. 提供一种在makefile中增加或者修改命令行参数的方式;

实际情况下,我们经常会有这种需求:通过命令行指定一些附加的参数选项,对于一些通用的参数选项在makefile中指定.

1 保护makefile中定义变量值示例

1.1 没有使用override的情况

make命令行指定的变量值将会覆盖makefile中定义的同名的变量值

SRCS := A.c B.c C.c

all:

@echo "SRCS: " $(SRCS)

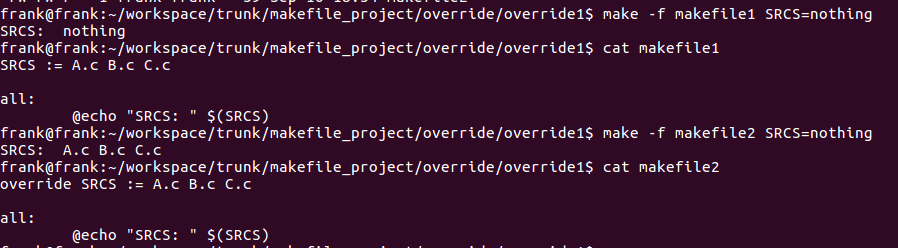
1.2 使用override的情况

make命令行指定的变量值将不会覆盖makefile中定义的同名的变量值，所以override有保护makefile中变量值不被命令行参数修改的作用。

override SRCS := A.c B.c C.c

all:

@echo "SRCS: " $(SRCS)



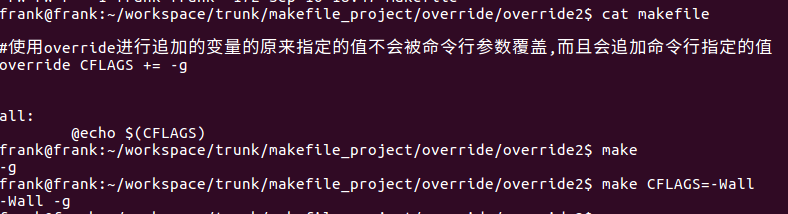
2 修改makefile中定义变量值的示例

#使用override进行追加的变量的原来指定的值不会被命令行参数覆盖,而且会追加命令行指定的值

override CFLAGS += -g

all:

@echo $(CFLAGS)



### makefile shell函数

shell函数也不像其它的函数。顾名思义，它的参数应该就是操作系统Shell的命令。它和反引号“`”是相同的功能。这就是说，shell函数把执行操作系统命令后的输出作为函数返回。于是，我们可以用操作系统命令以及字符串处理命令awk，sed等等命令来生成一个变量，如：

contents := $(shell cat foo)

files := $(shell echo \*.c)

注意，这个函数会新生成一个Shell程序来执行命令，所以你要注意其运行性能，如果你的Makefile中有一些比较复杂的规则，并大量使用了这个函数，那么对于你的系统性能是有害的。特别是Makefile的隐晦的规则可能会让你的shell函数执行的次数比你想像的多得多。

make的运行

make命令执行后有三个退出码：

0 —— 表示成功执行。

1 —— 如果make运行时出现任何错误，其返回1。

2 —— 如果你使用了make的“-q”选项，并且make使得一些目标不需要更新，那么返回2。

例如，我们有个makefile的名字是“hchen.mk”，那么，我们可以这样来让make来执行这个文件：

make –f hchen.mk

如果在make的命令行是，你不只一次地使用了“-f”参数，那么，所有指定的makefile将会被连在一起传递给make执行。

指定目标

一般来说，make的最终目标是makefile中的第一个目标，而其它目标一般是由这个目标连带出来的。这是make的默认行为。当然，一般来说，你的makefile中的第一个目标是由许多个目标组成，你可以指示make，让其完成你所指定的目标。要达到这一目的很简单，需在make命令后直接跟目标的名字就可以完成（如前面提到的“make clean”形式）

有一个make的环境变量叫“MAKECMDGOALS”，这个变量中会存放你所指定的终极目标的列表，如果在命令行上，你没有指定目标，那么，这个变量是空值。这个变量可以让你使用在一些比较特殊的情形下。比如下面的例子：

sources = foo.c bar.c

ifneq ( $(MAKECMDGOALS),clean)

include $(sources:.c=.d)

endif

基于上面的这个例子，只要我们输入的命令不是“make clean”，那么makefile会自动包含“foo.d”和“bar.d”这两个makefile。

使用指定终极目标的方法可以很方便地让我们编译我们的程序，例如下面这个例子：

.PHONY: all

all: prog1 prog2 prog3 prog4

从这个例子中，我们可以看到，这个makefile中有四个需要编译的程序——“prog1”，“prog2”，“prog3”和“prog4”，我们可以使用“make all”命令来编译所有的目标（如果把all置成第一个目标，那么只需执行“make”），我们也可以使用“make prog2”来单独编译目标“prog2”。

即然make可以指定所有makefile中的目标，那么也包括“伪目标”，于是我们可以根据这种性质来让我们的makefile根据指定的不同的目标来完成不同的事。在Unix世界中，软件发布时，特别是GNU这种开源软件的发布时，其makefile都包含了编译、安装、打包等功能。我们可以参照这种规则来书写我们的makefile中的目标。

“all”

这个伪目标是所有目标的目标，其功能一般是编译所有的目标。

“clean”

这个伪目标功能是删除所有被make创建的文件。

“install”

这个伪目标功能是安装已编译好的程序，其实就是把目标执行文件拷贝到指定的目标中去。

“print”

这个伪目标的功能是例出改变过的源文件。

“tar”

这个伪目标功能是把源程序打包备份。也就是一个tar文件。

“dist”

这个伪目标功能是创建一个压缩文件，一般是把tar文件压成Z文件。或是gz文件。

“TAGS”

这个伪目标功能是更新所有的目标，以备完整地重编译使用。

“check”和“test”

这两个伪目标一般用来测试makefile的流程。

### makefile隐含规则

“隐含规则”也就是一种惯例，make会按照这种“惯例”心照不喧地来运行，那怕我们的Makefile中没有书写这样的规则。例如，把[.c]文件编译成[.o]文件这一规则，你根本就不用写出来，make会自动推导出这种规则，并生成我们需要的[.o]文件。

“隐含规则”会使用一些我们系统变量，我们可以改变这些系统变量的值来定制隐含规则的运行时的参数。如系统变量“CFLAGS”可以控制编译时的编译器参数。

如果要使用隐含规则生成你需要的目标，你所需要做的就是不要写出这个目标的规则。那么，make会试图去自动推导产生这个目标的规则和命令，如果make可以自动推导生成这个目标的规则和命令，那么这个行为就是隐含规则的自动推导。当然，隐含规则是make事先约定好的一些东西。例如，我们有下面的一个Makefile：

foo : foo.o bar.o

cc –o foo foo.o bar.o $(CFLAGS) $(LDFLAGS)

我们可以注意到，这个Makefile中并没有写下如何生成foo.o和bar.o这两目标的规则和命令。因为make的“隐含规则”功能会自动为我们自动去推导这两个目标的依赖目标和生成命令。

make会在自己的“隐含规则”库中寻找可以用的规则，如果找到，那么就会使用。如果找不到，那么就会报错。在上面的那个例子中，make调用的隐含规则是，把[.o]的目标的依赖文件置成[.c]，并使用C的编译命令“cc –c $(CFLAGS) [.c]”来生成[.o]的目标。也就是说，

我们完全没有必要写下下面的两条规则：

foo.o : foo.c

cc –c foo.c $(CFLAGS)

bar.o : bar.c

cc –c bar.c $(CFLAGS)

因为，这已经是“约定”好了的事了，make和我们约定好了用C编译器“cc”生成[.o]文件的规则，这就是隐含规则。

还有，在make的“隐含规则库”中，每一条隐含规则都在库中有其顺序，越靠前的则是越被经常使用的，所以，这会导致我们有些时候即使我们显示地指定了目标依赖，make也不会管。

### makefile隐含规则一览

这里我们将讲述所有预先设置（也就是make内建）的隐含规则，如果我们不明确地写下规则，那么，make就会在这些规则中寻找所需要规则和命令。当然，我们也可以使用make的参数“-r”或“--no-builtin-rules”选项来取消所有的预设置的隐含规则。

1、编译C程序的隐含规则。

“<n>.o”的目标的依赖目标会自动推导为“<n>.c”，并且其生成命令是“$(CC) –c $(CPPFLAGS) $(CFLAGS)”

2、编译C++程序的隐含规则。

“<n>.o”的目标的依赖目标会自动推导为“<n>.cc”或是“<n>.C”，并且其生成命令是“$(CXX) –c $(CPPFLAGS) $(CFLAGS)”。（建议使用“.cc”作为C++源文件的后缀，而不是“.C”）

3、编译Pascal程序的隐含规则。

“<n>.o”的目标的依赖目标会自动推导为“<n>.p”，并且其生成命令是“$(PC) –c $(PFLAGS)”。

4、编译Fortran/Ratfor程序的隐含规则。

“<n>.o”的目标的依赖目标会自动推导为“<n>.r”或“<n>.F”或“<n>.f”，并且其生成命令是:

“.f” “$(FC) –c $(FFLAGS)”

“.F” “$(FC) –c $(FFLAGS) $(CPPFLAGS)”

“.f” “$(FC) –c $(FFLAGS) $(RFLAGS)”

5、预处理Fortran/Ratfor程序的隐含规则。

“<n>.f”的目标的依赖目标会自动推导为“<n>.r”或“<n>.F”。这个规则只是转换Ratfor或有预处理的Fortran程序到一个标准的Fortran程序。其使用的命令是：

“.F” “$(FC) –F $(CPPFLAGS) $(FFLAGS)”

“.r” “$(FC) –F $(FFLAGS) $(RFLAGS)”

6、编译Modula-2程序的隐含规则。

“<n>.sym”的目标的依赖目标会自动推导为“<n>.def”，并且其生成命令是：“$(M2C) $(M2FLAGS) $(DEFFLAGS)”。“<n.o>” 的目标的依赖目标会自动推导为“<n>.mod”，并且其生成命令是：“$(M2C) $(M2FLAGS) $(MODFLAGS)”。

7、汇编和汇编预处理的隐含规则。

“<n>.o” 的目标的依赖目标会自动推导为“<n>.s”，默认使用编译品“as”，并且其生成命令是：

“$(AS) $(ASFLAGS)”。“<n>.s” 的目标的依赖目标会自动推导为“<n>.S”，默认使用C预编译器“cpp”，并且其生成命令是：“$(AS) $(ASFLAGS)”。

8、链接Object文件的隐含规则。

“<n>”目标依赖于“<n>.o”，通过运行C的编译器来运行链接程序生成（一般是“ld”），其生成命令是：“$(CC) $(LDFLAGS) <n>.o $(LOADLIBES) $(LDLIBS)”。这个规则对于只有一个源文件的工程有效，同时也对多个Object文件（由不同的源文件生成）的也有效。例如如下规则：

x : y.o z.o

并且“x.c”、“y.c”和“z.c”都存在时，隐含规则将执行如下命令：

cc -c x.c -o x.o

cc -c y.c -o y.o

cc -c z.c -o z.o

cc x.o y.o z.o -o x

rm -f x.o

rm -f y.o

rm -f z.o

如果没有一个源文件（如上例中的x.c）和你的目标名字（如上例中的x）相关联，那么，你最好写出自己的生成规则，不然，隐含规则会报错的。

9、Yacc C程序时的隐含规则。

“<n>.c”的依赖文件被自动推导为“n.y”（Yacc生成的文件），其生成命令是：“$(YACC) $(YFALGS)”。（“Yacc”是一个语法分析器，关于其细节请查看相关资料）

10、Lex C程序时的隐含规则。

“<n>.c”的依赖文件被自动推导为“n.l”（Lex生成的文件），其生成命令是：“$(LEX) $(LFALGS)”。（关于“Lex”的细节请查看相关资料）

11、Lex Ratfor程序时的隐含规则。

“<n>.r”的依赖文件被自动推导为“n.l”（Lex生成的文件），其生成命令是：“$(LEX) $(LFALGS)”。

12、从C程序、Yacc文件或Lex文件创建Lint库的隐含规则。

“<n>.ln” （lint生成的文件）的依赖文件被自动推导为“n.c”，其生成命令是：“$(LINT) $(LINTFALGS) $(CPPFLAGS) -i”。对于“<n>.y”和“<n>.l”也是同样的规则。

### 隐含规则使用的变量

在隐含规则中的命令中，基本上都是使用了一些预先设置的变量。你可以在你的makefile中改变这些变量的值，或是在make的命令行中传入这些值，或是在你的环境变量中设置这些值，无论怎么样，只要设置了这些特定的变量，那么其就会对隐含规则起作用。当然，你也可以利用make的“-R”或“--no–builtin-variables”参数来取消你所定义的变量对隐含规则的作用。

例如，第一条隐含规则——编译C程序的隐含规则的命令是“$(CC) –c $(CFLAGS) $(CPPFLAGS)”。Make默认的编译命令是“cc”，如果你把变量“$(CC)”重定义成“gcc”，把变量“$(CFLAGS)”重定义成“-g”，那么，隐含规则中的命令全部会以“gcc –c -g $(CPPFLAGS)”的样子来执行了。

我们可以把隐含规则中使用的变量分成两种：一种是命令相关的，如“CC”；一种是参数相的关，如“CFLAGS”。下面是所有隐含规则中会用到的变量：

1、关于命令的变量。

CC

C语言编译程序。默认命令是“cc”。

CXX

C++语言编译程序。默认命令是“g++”。

CPP

C程序的预处理器（输出是标准输出设备）。默认命令是“$(CC) –E”。

RM

删除文件命令。默认命令是“rm –f”。

2、关于命令参数的变量

CFLAGS

C语言编译器参数。

CXXFLAGS

C++语言编译器参数。

CPPFLAGS

C预处理器参数。（C 和Fortran 编译器也会用到）。

### 隐含规则链

在默认情况下，对于中间目标，它和一般的目标有两个地方所不同：第一个不同是除非中间的目标不存在，才会引发中间规则。第二个不同的是，只要目标成功产生，那么，产生最终目标过程中，所产生的中间目标文件会被以“rm -f”删除。

在“隐含规则链”中，禁止同一个目标出现两次或两次以上，这样一来，就可防止在make自动推导时出现无限递归的情况。

Make会优化一些特殊的隐含规则，而不生成中间文件。如，从文件“foo.c”生成目标程序“foo”，按道理，make会编译生成中间文件“foo.o”，然后链接成“foo”，但在实际情况下，这一动作可以被一条“cc”的命令完成（cc –o foo foo.c），于是优化过的规则就不会生成中间文件。

### 定义模式规则

你可以使用模式规则来定义一个隐含规则。一个模式规则就好像一个一般的规则，只是在规则中，目标的定义需要有"%"字符。"%"的意思是表示一个或多个任意字符。在依赖目标中同样可以使用"%"，只是依赖目标中的"%"的取值，取决于其目标。

有一点需要注意的是，"%"的展开发生在变量和函数的展开之后，变量和函数的展开发生在make载入Makefile时，而模式规则中的"%"则发生在运行时。

模式规则中，至少在规则的目标定义中要包含"%"，否则，就是一般的规则。目标中的"%"定义表示对文件名的匹配，"%"表示长度任意的非空字符串。例如："%.c"表示以".c"结尾的文件名（文件名的长度至少为3），而"s.%.c"则表示以"s."开头，".c"结尾的文件名（文件名的长度至少为5）。

例如有一个模式规则如下：

%.o : %.c ; <command ......>

其含义是，指出了怎么从所有的[.c]文件生成相应的[.o]文件的规则。如果要生成的目标是"a.o b.o"，那么"%c"就是"a.c b.c"。

一旦依赖目标中的"%"模式被确定，那么，make会被要求去匹配当前目录下所有的文件名，一旦找到，make就会规则下的命令，所以，在模式规则中，目标可能会是多个的，如果有模式匹配出多个目标，make就会产生所有的模式目标，此时，make关心的是依赖的文件名和生成目标的命令这两件事。

下面这个例子表示了,把所有的[.c]文件都编译成[.o]文件.

%.o : %.c

$(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@

其中，"$@"表示所有的目标的挨个值，"$<"表示了所有依赖目标的挨个值。这些奇怪的变量我们叫"自动化变量"，后面会详细讲述。

### 自动化变量

在上述的模式规则中，目标和依赖文件都是一系例的文件，那么我们如何书写一个命令来完成从不同的依赖文件生成相应的目标？因为在每一次的对模式规则的解析时，都会是不同的目标和依赖文件。

自动化变量就是完成这个功能的。在前面，我们已经对自动化变量有所提涉，相信你看到这里已对它有一个感性认识了。所谓自动化变量，就是这种变量会把模式中所定义的一系列的文件自动地挨个取出，直至所有的符合模式的文件都取完了。这种自动化变量只应出现在规则的命令中。

下面是所有的自动化变量及其说明：

$@

表示规则中的目标文件集。在模式规则中，如果有多个目标，那么，"$@"就是匹配于目标中模式定义的集合。

$%

仅当目标是函数库文件中，表示规则中的目标成员名。例如，如果一个目标是"foo.a(bar.o)"，那么，"$%"就是"bar.o"，"$@"就是"foo.a"。如果目标不是函数库文件（Unix下是[.a]，Windows下是[.lib]），那么，其值为空。

$<

依赖目标中的第一个目标名字。如果依赖目标是以模式（即"%"）定义的，那么"$<"将是符合模式的一系列的文件集。注意，其是一个一个取出来的。

$?

所有比目标新的依赖目标的集合。以空格分隔。

$^

所有的依赖目标的集合。以空格分隔。如果在依赖目标中有多个重复的，那个这个变量会去除重复的依赖目标，只保留一份。

$+

这个变量很像"$^"，也是所有依赖目标的集合。只是它不去除重复的依赖目标。

$\*

这个变量表示目标模式中"%"及其之前的部分。如果目标是"dir/a.foo.b"，并且目标的模式是"a.%.b"，那么，"$\*"的值就是"dir/a.foo"。这个变量对于构造有关联的文件名是比较有较。如果目标中没有模式的定义，那么"$\*"也就不能被推导出，但是，如果目标文件的后缀是make所识别的，那么"$\*"就是除了后缀的那一部分。例如：如果目标是"foo.c"，因为".c"是make所能识别的后缀名，所以，"$\*"的值就是"foo"。这个特性是GNU make的，很有可能不兼容于其它版本的make，所以，你应该尽量避免使用"$\*"，除非是在隐含规则或是静态模式中。如果目标中的后缀是make所不能识别的，那么"$\*"就是空值。

当你希望只对更新过的依赖文件进行操作时，"$?"在显式规则中很有用，例如，假设有一个函数库文件叫"lib"，其由其它几个object文件更新。那么把object文件打包的比较有效率的Makefile规则是：

lib : foo.o bar.o lose.o win.o

ar r lib $?

在上述所列出来的自动量变量中。四个变量（$@、$<、$%、$\*）在扩展时只会有一个文件，而另三个的值是一个文件列表。这七个自动化变量还可以取得文件的目录名或是在当前目录下的符合模式的文件名，只需要搭配上"D"或"F"字样。这是GNU make中老版本的特性，在新版本中，我们使用函数"dir"或"notdir"就可以做到了。"D"的含义就是Directory，就是目录，"F"的含义就是File，就是文件。

下面是对于上面的七个变量分别加上"D"或是"F"的含义：

$(@D)

表示"$@"的目录部分（不以斜杠作为结尾），如果"$@"值是"dir/foo.o"，那么"$(@D)"就是"dir"，而如果"$@"中没有包含斜杠的话，其值就是"."（当前目录）。

$(@F)

表示"$@"的文件部分，如果"$@"值是"dir/foo.o"，那么"$(@F)"就是"foo.o"，"$(@F)"相当于函数"$(notdir $@)"。

"$(\*D)"

"$(\*F)"

和上面所述的同理，也是取文件的目录部分和文件部分。对于上面的那个例子，"$(\*D)"返回"dir"，而"$(\*F)"返回"foo"

"$(%D)"

"$(%F)"

分别表示了函数包文件成员的目录部分和文件部分。这对于形同"archive(member)"形式的目标中的"member"中包含了不同的目录很有用。

"$(<D)"

"$(<F)"

分别表示依赖文件的目录部分和文件部分。

"$(^D)"

"$(^F)"

分别表示所有依赖文件的目录部分和文件部分。（无相同的）

"$(+D)"

"$(+F)"

分别表示所有依赖文件的目录部分和文件部分。（可以有相同的）

"$(?D)"

"$(?F)"

分别表示被更新的依赖文件的目录部分和文件部分。

最后想提醒一下的是，对于"$<"，为了避免产生不必要的麻烦，我们最好给$后面的那个特定字符都加上圆括号，比如，"$(<)"就要比"$<"要好一些。

还得要注意的是，这些变量只使用在规则的命令中，而且一般都是"显式规则"和"静态模式规则"（参见前面"书写规则"一章）。其在隐含规则中并没有意义。

### 模式的匹配

一般来说，一个目标的模式有一个有前缀或是后缀的"%"，或是没有前后缀，直接就是一个"%"。因为"%"代表一个或多个字符，所以在定义好了的模式中，我们把"%"所匹配的内容叫做"茎"，例如"%.c"所匹配的文件"test.c"中"test"就是"茎"。因为在目标和依赖目标中同时有"%"时，依赖目标的"茎"会传给目标，当做目标中的"茎"。

当一个模式匹配包含有斜杠（实际也不经常包含）的文件时，那么在进行模式匹配时，目录部分会首先被移开，然后进行匹配，成功后，再把目录加回去。在进行"茎"的传递时，我们需要知道这个步骤。例如有一个模式"e%t"，文件"src/eat"匹配于该模式，于是"src/a"就是其"茎"，如果这个模式定义在依赖目标中，而被依赖于这个模式的目标中又有个模式"c%r"，那么，目标就是"src/car"。（"茎"被传递）

### makefile命令参数

make –f Makefile

如果要指定特定的Makefile，你可以使用make的“- f”和“--file”参数

如：make -f Make.Linux或make --file Make.AIX。

## linux下的c编程

gcc是Linux下的C程序编译器，具有非常强大的程序编译功能。在Linux系统下，C语言编写的程序代码一般需要通过gcc来编译成可执行程序。

gcc 在编译一个程序时，一般需要完成预处理（preprocessing）、编译（compilation）、汇编（assembly）和链接（linking）过程。使用gcc编译C程序时，这些过程是使用默认的设置自动完成的

### gcc的可选参数

在终端中显示的gcc的可选参数如下所示。进行程序编译时，可以设置下面的这些参数。

用法：gcc [选项] 文件...

选项：

-pass-exit-codes：在某一阶段退出时返回最高的错误码

--help：显示此帮助说明

--target-help：显示目标机器特定的命令行选项

-dumpspecs：显示所有内建spec 字符串

-dumpversion：显示编译器的版本号

-dumpmachine：显示编译器的目标处理器

-print-search-dirs：显示编译器的搜索路径

-print-libgcc-file-name：显示编译器伴随库的名称

-print-file-name=<库>：显示<库> 的完整路径

-print-prog-name=<程序>：显示编译器组件<程序> 的完整路径

-print-multi-directory：显示不同版本libgcc 的根目录

-print-multi-lib：显示命令行选项和多个版本库搜索路径间的映射

-print-multi-os-directory：显示操作系统库的相对路径

-Wa,<选项>： 将逗号分隔的<选项> 传递给汇编器

-Wp,<选项>：将逗号分隔的<选项> 传递给预处理器

-Wl,<选项>：将逗号分隔的<选项> 传递给链接器

-Xassembler <参数>：将<参数> 传递给汇编器

-Xpreprocessor <参数>：将<参数> 传递给预处理器

-Xlinker <参数>：将<参数> 传递给链接器

-combine：将多个源文件一次性传递给汇编器

-save-temps：不删除中间文件

-pipe：使用管道代替临时文件

-time：为每个子进程计时

-specs=<文件>： 用<文件> 的内容覆盖内建的specs 文件

-std=<标准>：指定输入源文件遵循的标准

--sysroot=<目录>：将<目录> 作为头文件和库文件的根目录

-B <目录>：将<目录> 添加到编译器的搜索路径中

-b <机器>：为gcc 指定目标机器(如果有安装)

-V <版本>：运行指定版本的gcc(如果有安装)

-v：显示编译器调用的程序

-###：与-v 类似，但选项被引号括住，并且不执行命令

-E：仅作预处理，不进行编译、汇编和链接

-S：编译到汇编语言，不进行汇编和链接

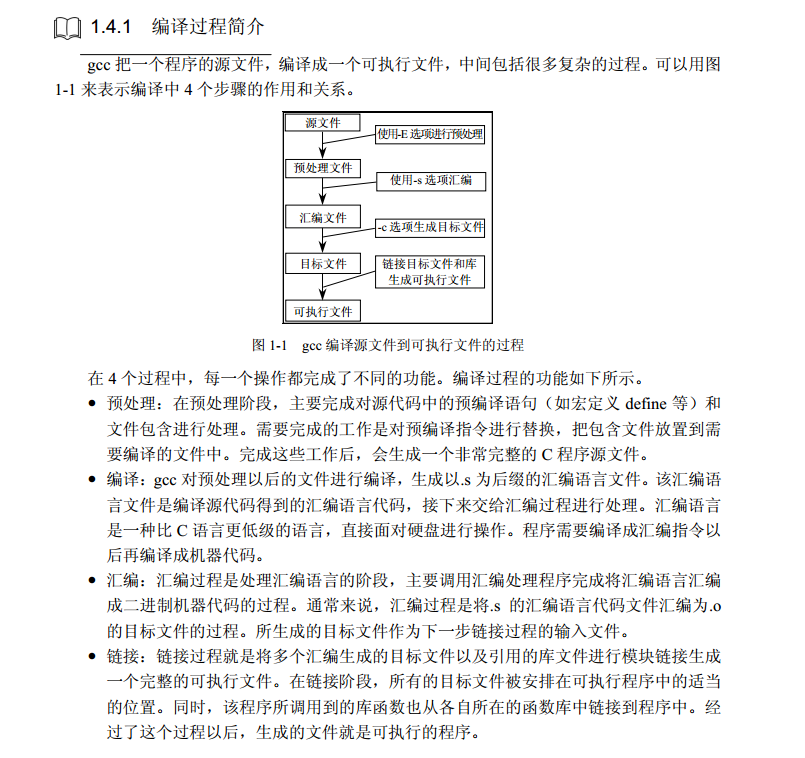
-c：编译、汇编到目标代码，不进行链接

-o <文件>：输出到<文件>

-x <语言>：指定其后输入文件的语言。允许的语言包括c、c++、assembler等。

以-g、-f、-m、-O、-W 或--param 开头的选项将由gcc 自动传递给其调用的不同子进程。若要向这些进程传递其他选项，必须使用-W<字母> 选项。

gcc可以编译C++程序。编译C程序和C++程序时，使用的是不同的命令。编译C++程序时，使用的命令是g++。该命令的使用方法与gcc是相似的。



## cpp

### linux编译 -std=c++11

g++ -o hello hello.cpp -std=c++11

查看运行状态：echo $?

g++ TemplateDemo.cpp TemplateDemo.h -o TemplateDemo -std=c++11

### base

endl：这是一个被称为操纵符（manipulator）的特殊值，效果是结束当前行，并将设备关联的缓冲区（buffer）中的内容刷到设备中。

UNIX和Mac下键盘输入文件结束符：ctrl+d，Windows下：ctrl+z

头文件：类的类型一般存储在头文件中，标准库的头文件使用<>，非标准库的头文件使用""。申明写在.h文件，定义实现写在.cpp文件。

### 使用文件重定向

./main <infile >outfile

### namespace命名空间

命名空间（namespace）将库定义的名字放在一个单一位置的机制。命名空间可以帮助避免不经意的名字冲突。C++标准库定义的名字在命名空间std中。

::运算符（== operator）作用域运算符。其用处之一是访问命名空间中的名字。例如，std:cout表示命名空间std中的名字cout。

命名空间（namespace）：使用作用域运算符::调用。

### 类型长度与选用

float 4字节 32位 7个有效位

double 8字节 64位 16个有效位

int 4字节 32位

long 4字节 32位

long long 8字节 64位

如果你的数值超过了int的表示范围，选用long long。

执行浮点运算选用double，这是因为float通常精度不够而且双精度浮点数和单精度浮点数的计算代价相差无几。

类型unsigned int可以缩写为unsigned

切勿混用带符号类型和无符号类型。

以0开头的整数代表八进制数

以0x或者0X开头的代表十六进制数

20 /\*十进制\*/

020 /\*八进制\*/

0x14 /\*十六进制\*/

变量

对于C++程序员来说，“变量(variable)”和“对象(object)”一般可以互换使用。

初始化不是赋值，初始化的含义是创建变量是赋予其一个初始值，而赋值的含义是把对象的当前值擦除，而以一个新值来替代。

绝大多数类都支持无须显式初始化而定义对象，这样的类提供了一个合适的默认值。

定义于函数体内的内置类型的对象如果没有初始化，则其值未定义。类的对象如果没有显式地初始化，则其值由类决定。

使用未初始化的变量将带来无法预计的后果。

如果想声明一个变量而非定义它，就在变量名前添加关键字extern，而且不要显示地初始化变量：

extern int i;//声明i而非定义i

int j;//声明并定义j

变量能且只能被定义一次，但是可以被多次声明。

作用域分为：全局作用域和块作用域。

### 引用

引用为对象起了另一个名字，通过将声明符写成&d的形式来定义引用类型，其中d是声明的变量名：

int ival = 1024;

int &refVal = ival;//refVal指向ival（是ival的另一个名字）

引用不是一个对象，所以不能定义引用的引用。

与引用相比：1、指针本身就是一个对象，允许对指针赋值和拷贝，而且在指针的声明周期内它可以先后指向几个不同的对象；2、指针无须在定义时赋初值。和其他内置类型一样，在块作用域内定义的指针如果没有被初始化，也将拥有一个不确定的值。

指针赋值：

int ival = 42;

int \*p = &ival;

因为引用不是对象，没有实际的地址，所以不能定义指向引用的指针。

声明语句中的指针的类型实际上被用于指定它所指向的对象类型，所以二者必须匹配。如果指针指向了一个其他类型的对象，对该对象的操作将发送错误。

引用必须初始化。

引用和它的初始值是绑定bind在一起的，而不是拷贝。

### 指针值

指针的值（即地址）应属下列4种状态之一：

1. 指向一个对象
2. 指向紧邻对象所占空间的下一个位置
3. 空指针，意味着指针没有指向任何对象
4. 无效指针，也就是上述情况之外的其他值

空指针与无效指针的区别

试图拷贝或以其他方式访问无效指针的值都将引发错误。编译器并不负责检查此类错误，这一点和试图使用未经初始化的变量是一样的。

如果指针指向了一个对象，则允许使用解引用符（操作符\*）来访问该对象：

cout << \*p;

空指针

空指针（null pointer）不指向任何对象，在试图使用一个指针之前可以首先检查它是否为空。

int \*p1 = nullptr;//c++

int \*p2 = 0;

int \*p3 = NULL;//c

//需要首先#include cstdlib

得到空指针最直接的办法就是用字面值nullptr来初始化指针。

如果实现不清楚指针应该指向何处，就把它初始化为nullptr或者0，这样程序就能检测并知道它没有指向任何具体的对象了。

对于两个类型相同的合法指针，可以用相等操作符（==）或不等操作符（！=）来比较，比较结果是布类型。

定义指针类型： int \*ip1;，从右向左读，ip1是指向int类型的指针。

指针访问对象： cout << \*p;， \*是解引用符。

两个指针相减的类型是ptrdiff\_t。

### void\* 指针

void\*指针是一种特殊的指针类型，可用于存放任意对象的地址。一个void\*指针存放着一个地址，不同的是，对该地址中到底是个什么类型的对象并不了解。

利用void\*指针能做的事比较有限：拿它和别的指针比较、作为函数的输入或输出、或者赋值给另外一个void\*指针。不能直接操作void\*指针所指的对象，因为不知道这个对象到底是什么类型，也就无法确定能在这个对象上做哪些操作。

以void\*的视角来看内存空间也就仅仅是内存空间，没办法访问内存空间中所存的对象。

复合类型的声明

int \*p;//鼓励使用

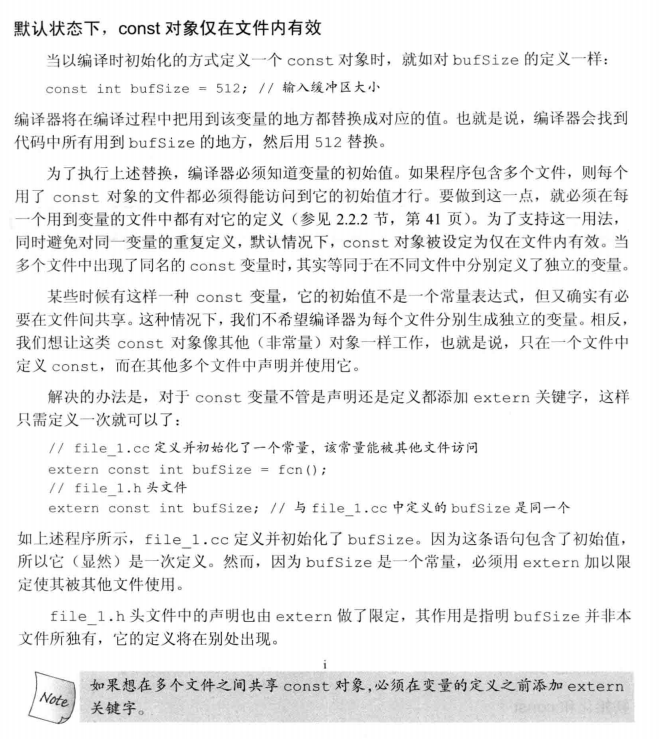
int\* p;//合法但是容易产生误导

将\*（或是&）与变量名连在一起。

const限定符

定义这样一个变量，它的值不能被改变。

因为const对象一旦创建后其值就不能在改变，所以const对象必须初始化。



### const指针

指针是对象而因引用不是，因此就像其他对象类型一样，允许把指针本身定位常量。常量指针（const pointer）必须初始化，而且一旦初始化完成，则它的值（也就是存放在指针中的那个地址）就不能改变了。把\*放在const关键字之前用以说明指针是一个常量，说明不变的是指针本身而不是指向的那个值。

int errNum = 0;

int \*const curErr = &errNum;//curErr将一直指向errNum

指针本身是一个常量并不意味着不能通过指针修改其所指对象的值，能否这样做完全依赖于所指对象的类型。

const对象必须初始化，且不能被改变。

const变量默认不能被其他文件访问，非要访问，必须在指定const前加extern。

指针和const

pointer to const（指向常量的指针）：不能用于改变其所指对象的值, 如 const double pi = 3.14; const double \*cptr = &pi;。

const pointer：指针本身是常量，如 int i = 0; int \*const ptr = &i;

### 顶层const 底层const

顶层const表示指针本身是个常量，底层const表示指针所指的对象是一个常量。

int \*const p1 = &i;//顶层const

const int \*p2 = &ci;//底层const

底层const：

const int \*p;//const在\*左边，表示\*p为常量，不可更改（经由\*p不能更改所指向的内容）

但是指针p还是变量想怎么变都可以，这就是底层指针。

int b = 22;

const int \*p;

p = &b;//正确

\*p = 200;//错误，\*p是常量，不能再对常量进行赋值

顶层const：

int \*const p = &b;//在声明的同时必须初始化，const在\*的右边，表示p为常量，p为所指向的地址

int b = 33;

int c = 22;

int \*const p = &b;

p = &c;

//p是不可更改的，所以当把c的地址赋值给它时，会报错。这就是顶层const。

const int \*const p;

//这个就相当于以上两种情况的结合体，p是常量，\*p也是常量。

类型别名

1. 使用关键字：传统别名：使用typedef来定义类型的同义词。

typedef double wages;

1. 使用别名声明（alias declaration）：新标准别名：

using SI = Sales\_item;(C++11)

### auto类型说明符

auto的原理是根据后面的值，来自己推断前面的类型，让编译器自动推断类型。

用auto声明的变量必须初始化。

函数和模板参数不能被声明为auto

定义在一个auto序列的变量必须始终推导成同一类型。

会忽略顶层const。

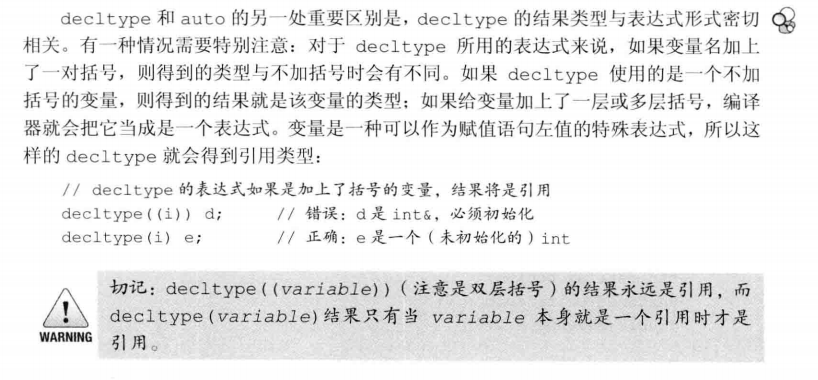
const int ci = 1; const auto f = ci;推断类型是int，需要自己加const

### decltype类型指示符

希望从表达式的类型推断出要定义的变量的类型，但是不想用该表达式的值初始化变量：使用decltype

decltype (f()) sum = x;//sum的类型就是函数f扽返回类型

不会忽略顶层const。



### 预处理

确保头文件多次包含仍能安全工作的常见技术是预处理器（prepricessor），预处理器是在编译之前执行的一段程序，可以部分地改变所写的程序。

### 头文件保护符（header guard）

避免多次包含同一头文件：

#ifndef SALES\_DATA\_H

#define SALES\_DATA\_H

#include <string>

struct Sales\_data

{

std::string bookNo;

unsigned units\_sold = 0;

double revenue = 0.0;

};

#endif

预处理变量无视c++语言中关于作用域的规则。

头文件即使（目前还）没有被包含在任何其他头文件中，也应该设置保护符。头文件保护符很简单，程序员只要习惯性地加上就可以，没必要太在乎你的程序到底需不需要。

### typedef struct

typedef是类型定义的意思。typedef struct 是为了使用这个结构体方便。

具体区别在于:

若struct node{ }这样来定义结构体的话。在定义 node 的结构体变量时，需要这样写:struct node n;

若用typedef，可以这样写：typedef struct node{}NODE; 。在申请变量时就可以这样写：NODE n;其实就相当于 NODE 是node 的别名。区别就在于使用时，是否可以省去struct这个关键字。

首先：

在C中定义一个结构体类型时如果要用typedef:

1. typedef struct Student
2. {
3. int no;
4. char name[12];
5. }Stu,student;

于是在声明变量的时候就可：Stu stu1;或者：student stu2;(Stu 和student 同时为Student的别名)

在c++中如果用typedef的话，又会造成区别：

struct Student

{

int no;

char name[12];

}stu1;//stu1是一个变量

typedef struct Student2

{

int no;

char name[12];

}stu2;//stu2是一个结构体类型，即stu2是Student2的别名

使用时可以直接访问stu1.no

但是stu2则必须先定义 stu2 s2;

然后 s2.no=10;

### C++ vector 容器浅析

vector是一个容器，也是一个类模板；

#include <vector> 然后 using std::vector;

容器：包含其他对象。

类模板：本身不是类，但可以实例化instantiation出一个类。 vector是一个模板， vector<int>是一个类型。

通过将类型放在类模板名称后面的尖括号中来指定类型，如vector<int> ivec。

定义和初始化vector对象

初始化vector对象的方法

vector<T> v1

v1是一个空vector，它潜在的元素是T类型的，执行默认初始化

vector<T> v2(v1)

v2中包含有v1所有元素的副本

vector<T> v2 = v1

等价于v2(v1)，v2中包含v1所有元素的副本

vector<T> v3(n, val)

v3包含了n个重复的元素，每个元素的值都是val

vector<T> v4(n)

v4包含了n个重复地执行了值初始化的对象

vector<T> v5{a, b, c...}

v5包含了初始值个数的元素，每个元素被赋予相应的初始值

vector<T> v5={a, b, c...}

等价于v5{a, b, c...}

列表初始化： vector<string> v{"a", "an", "the"}; （C++11）

向vector对象中添加元素

v.push\_back(e) 在尾部增加元素。

向量（Vector）是一个封装了动态大小数组的顺序容器（Sequence Container）。跟任意其它类型容器一样，它能够存放各种类型的对象。可以简单的认为，向量是一个能够存放任意类型的动态数组。

vector<int>obj;//创建一个向量存储容器 int

for(int i=0;i<10;i++) // push\_back(elem)在数组最后添加数据

{

obj.push\_back(i);

cout<<obj[i]<<",";

}

for(int i=0;i<5;i++)//去掉数组最后一个数据

{

obj.pop\_back();

}

for(int i=0;i<obj.size();i++)//size()容器中实际数据个数

{

cout<<obj[i]<<",";

}

obj.clear();//清除容器中所有数据

vector<int>obj;

obj.push\_back(1);

obj.push\_back(3);

obj.push\_back(0);

sort(obj.begin(),obj.end());//从小到大

reverse(obj.begin(),obj.end());//从大到小

vector<int>obj;

cout<<"利用迭代器：" ;

//方法二，使用迭代器将容器中数据输出

vector<int>::iterator it;//声明一个迭代器，来访问vector容器，作用：遍历或者指向vector容器的元素

for(it=obj.begin();it!=obj.end();it++)

{

cout<<\*it<<" ";

}

二维数组：

方法1、

int N=5, M=6;

vector<vector<int> > obj(N); //定义二维动态数组大小5行

for(int i =0; i< obj.size(); i++)//动态二维数组为5行6列，值全为0

{

obj[i].resize(M);

}

for(int i=0; i< obj.size(); i++)//输出二维动态数组

{

for(int j=0;j<obj[i].size();j++)

{

cout<<obj[i][j]<<" ";

}

cout<<"\n";

}

return 0;

方法2、

int N=5, M=6;

vector<vector<int> > obj(N, vector<int>(M)); //定义二维动态数组5行6列

for(int i=0; i< obj.size(); i++)//输出二维动态数组

{

for(int j=0;j<obj[i].size();j++)

{

cout<<obj[i][j]<<" ";

}

cout<<"\n";

}

return 0;

增加函数

void push\_back(const T& x):向量尾部增加一个元素X

iterator insert(iterator it,const T& x):向量中迭代器指向元素前增加一个元素x

iterator insert(iterator it,int n,const T& x):向量中迭代器指向元素前增加n个相同的元素x

iterator insert(iterator it,const\_iterator first,const\_iterator last):向量中迭代器指向元素前插入另一个相同类型向量的[first,last)间的数据

如果需要在每个元素之后插入元素，该怎么操作？

#include<iostream>

#include <vector>

using namespace std;

int main()

{

vector<int> vec{ 1,2,3,4,5};

auto iter = vec.begin();

iter++;

for (; iter != vec.end();)

{

iter = vec.insert(iter,8);

iter++;

iter++;

}

vec.insert(vec.end(),8);

for (auto num : vec)

cout << num << endl;

return 0;

}

### 迭代器iterator

所有标准库容器都可以使用迭代器。

类似于指针类型，迭代器也提供了对对象的间接访问。

#include <iostream>

#include <vector>

int main()

{

std::vector<int> ivec = {1,2,3,4,5};

std::vector<int>::iterator iter;

for (auto iter : ivec)

{

std::cout << iter << std::endl;

}

return 0;

}

std::vector<int>::iterator iter。

auto b = v.begin();返回指向第一个元素的迭代器。

auto e = v.end();返回指向最后一个元素的下一个（哨兵，尾后,one past the end）的迭代器（off the end）。

使用解引用符\*访问迭代器指向的元素。

auto b = ivec.begin();

auto e = ivec.end();

//std::cout << b << std::endl; //error

std::cout << \*(++b) << std::endl;

iter + n

迭代器加上一个整数值仍得到一个迭代器，迭代器指示的新位置和原来相比向前移动了若干个元素。结果迭代器或者指示容器内的一个元素，或者指示容器尾元素的下一位置。

iter - n

迭代器减去一个证书仍得到一个迭代器，迭代器指示的新位置比原来向后移动了若干个元素。结果迭代器或者指向容器内的一个元素，或者指示容器尾元素的下一位置。

### 数组

相当于vector的低级版，长度固定。

定义和初始化内置数组

初始化：char input\_buffer[buffer\_size];，长度必须是const表达式，或者不写，让编译器自己推断。

数组不允许直接赋值给另一个数组。

访问数组元素

数组下标的类型：size\_t 。

字符数组的特殊性：结尾处有一个空字符，如 char a[] = "hello"; 。

用数组初始化 vector： int a[] = {1,2,3,4,5}; vector<int> v(begin(a), end(a)); 。

数组和指针

使用数组时，编译器一般会把它转换成指针。

标准库类型限定使用的下标必须是无符号类型，而内置的下标可以处理负值。

指针访问数组：在表达式中使用数组名时，名字会自动转换成指向数组的第一个元素的指针。

### C++ 模板详解

<https://www.runoob.com/w3cnote/c-templates-detail.html>

### using声明

使用某个命名空间：例如 using std::cin表示使用命名空间std中的名字cin。

头文件中不应该包含using声明。这样使用了该头文件的源码也会使用这个声明，会带来风险。

### string

标准库类型string表示可变长的字符序列。

#include <string>，然后 using std::string;

string对象：注意，不同于字符串字面值。

string io：

执行读操作>>：忽略掉开头的空白（包括空格、换行符和制表符），直到遇到下一处空白为止。

getline：读取一整行，包括空白符。

字符串字面值和string是不同的类型。

ctype.h vs. cctype：C++修改了c的标准库，名称为去掉.h，前面加c。

遍历字符串：使用范围for（range for）语句： for (auto c: str)，或者 for (auto &c: str)使用引用直接改变字符串中的字符。 （C++11）