Linux 下 C 语言编程环境

嵌入式资源免费下载
http://www.kontronn.com

Linux 的诞生与发展

- Minix 操作系统主要是用于教学科研为目的，它是完全公开源码的。并在 comp.os.minix 新闻组中进行讨论，很多 Minix 爱好者就针对 Minix 系统进行了改动，但是此举并未被 Minix 作者接收，使得很多人对 Minix 有较多的看法。
- 出于对 Minix 相对保守的做法的不满，芬兰赫尔辛基大学的 Linus Torvalds 开发了 Linux 。

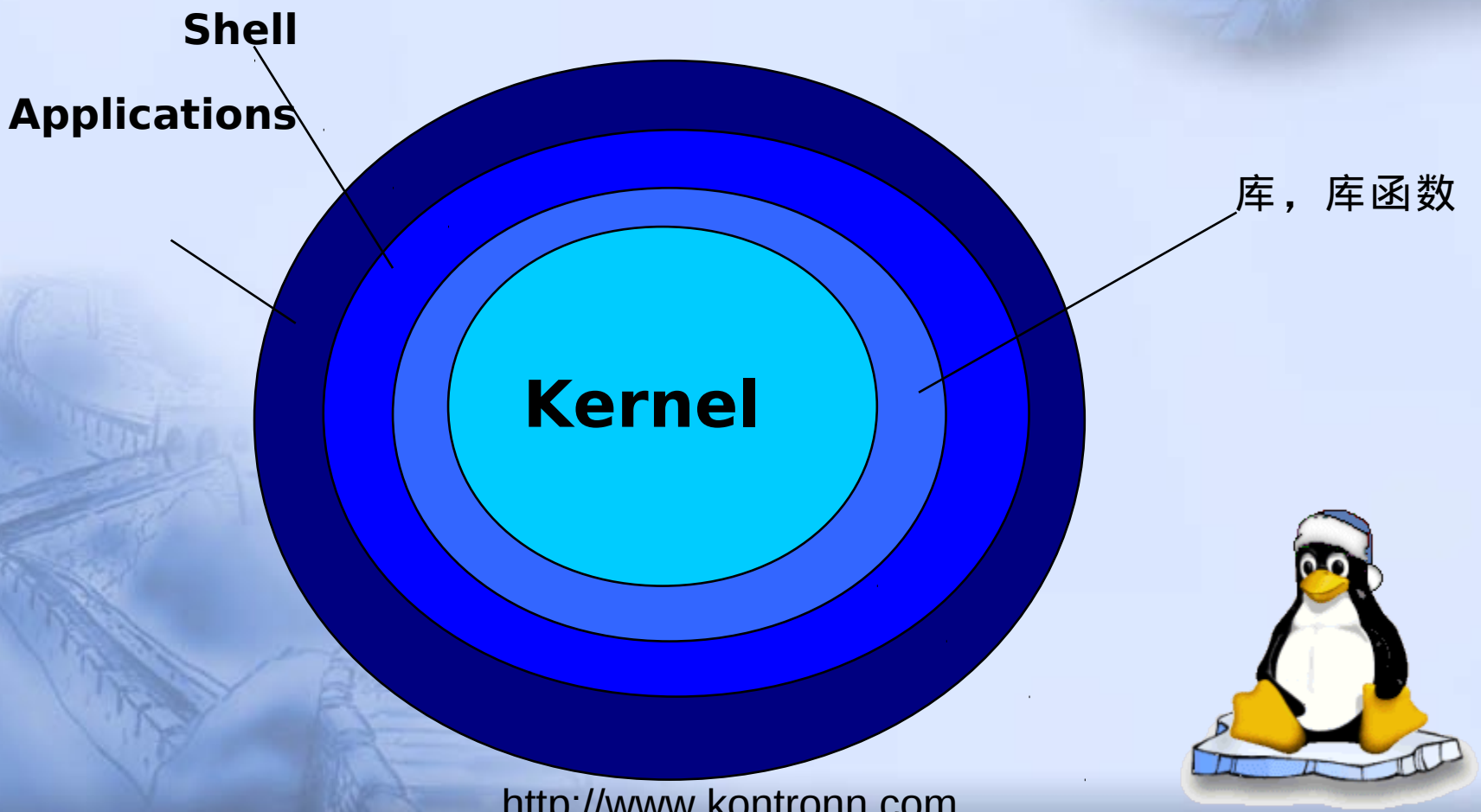
Linux 的诞生与发展

- Linux 是一个诞生于网络、成长于网络且成熟于网络的奇特的操作系统。
- Linux 一开始是要求所有的源码必须公开，并且任何人均不得从 Linux 交易中获利。然而这种纯粹的自由软件的理想对于 Linux 的普及和发展是不利的，于是 Linux 开始转向 GPL，成为 GNU 阵营中的主要一员。

Linux 发展的重要里程碑

- 1990, Linus Torvalds 首次接触 MINIX
- 1991, Linus Torvalds 开始在 MINIX 上编写各种驱动程序等操作系统内核组件
- 1991 底 , Linus Torvalds 公开了 Linux 内核
- 1993, Linux 1.0 版发行， Linux 转向 GPL 版权协议
- 1994, Linux 的第一个商业发行版 Slackware 问世
- 1996，美国国家标准技术局的计算机系统实验室确认 Linux 版本 1.2.13（由 Open Linux 公司打包）符合 POSIX 标准
- 1999, Linux 的简体中文发行版相继问世，例如：红帽、 Turbo linux 。
- 值得一提的是，中标软公司在国家核高基项目的支持下，基于开源的 Linux 内核，发展出了具有自主知识产权的中标麒麟桌面操作系统，中标麒麟高级服务器，中标麒麟通用服务器

Linux 系统结构



Linux Shell

- Shell 也是一个系统程序，但她与后台工作的一般系统程序具有不同的功能，它直接面对用户，提供了用户与内核进行交互操作的界面。它接收用户输入的命令，并把它送入内核去执行。
- 实际上， Shell 是一个命令解释器，它解释由用户输入的命令，并把它们交给内核。

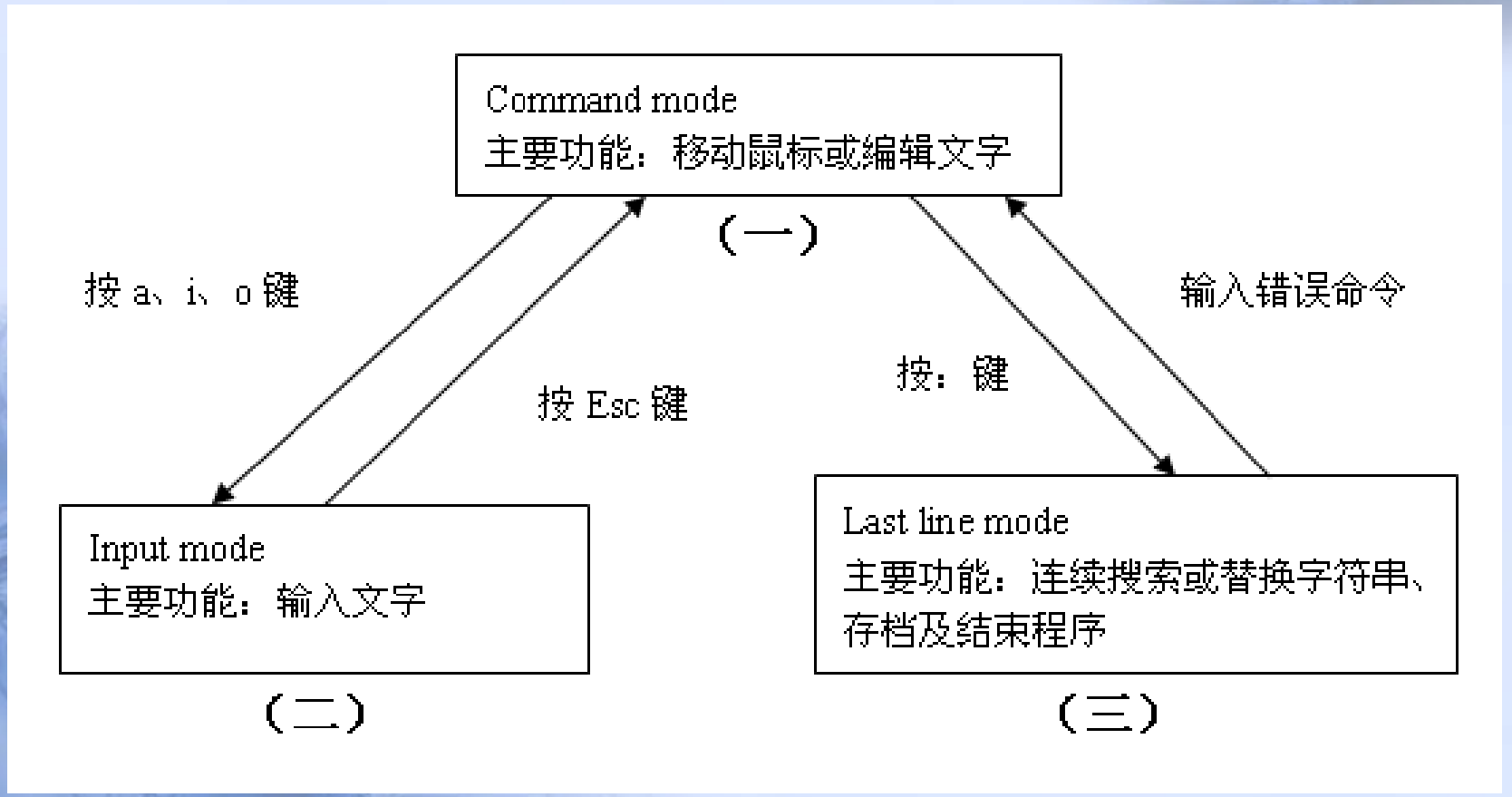
Vim 编辑器的使用

- Vim 编辑器的简介
  - Vim 是“ visual interface （虚拟界面）”的简称，它是 Unix 世界中最常用的全屏幕文本编辑器，可以执行输出、删除、查找、替换、块操作等众多文本操作
  - Vim 不是排版程序，它不像 Word 或 WPS 那样，可以对字体、格式、段落等其他属性进行编排，它只是一个文本编辑程序。
  - Vim 没有菜单，它通过命令来对文本进行编辑操作。

Vim 编辑器的使用

- Vim 有 3 中模式：
  - 命令模式 command mode： 供用户执行命令
  - 输入模式 input mode： 可输入内容
  - 末行模式 last line mode： 让用户做一些与输入文字无关的事，如搜索字符串、保存文件或结束编辑等。

Vim 编辑器的使用


Command mode
主要功能：移动鼠标或编辑文字
（一）
按 a、i、o 键
按 Esc 键
输入错误命令
按：键
Input mode
主要功能：输入文字
（二）
Last line mode
主要功能：连续搜索或替换字符串、存档及结束程序
（三）

Vim 编辑器的使用

- 在命令模式下：
  - 删除整行： dd 或 ndd （如 5dd 就是删除光标下面的 5 行内容）
  - 粘贴：小 p （粘贴到光标的后面）大 P （粘贴到光标的前面）
  - 复制： yy 或 nyy （如 3yy 就是复制此光标下的 3 行内容）
  - 撤消与重复： u （撤消前一条命令）；“ .” （重复最后一条命令）

Vim 编辑器的使用

- 末行模式：
  - “:q” 退出（“ !” 必要时可以强制执行）“ :w” 保存。
  - 例如：表示保存退出时我们可以使用“ :wq”
  - 查找字符串：先输入“ /”，再输入要搜索的内容（正向搜索）；“ shift+n”（反向搜索）；“ n” 继续搜索；

Vim 的高级使用

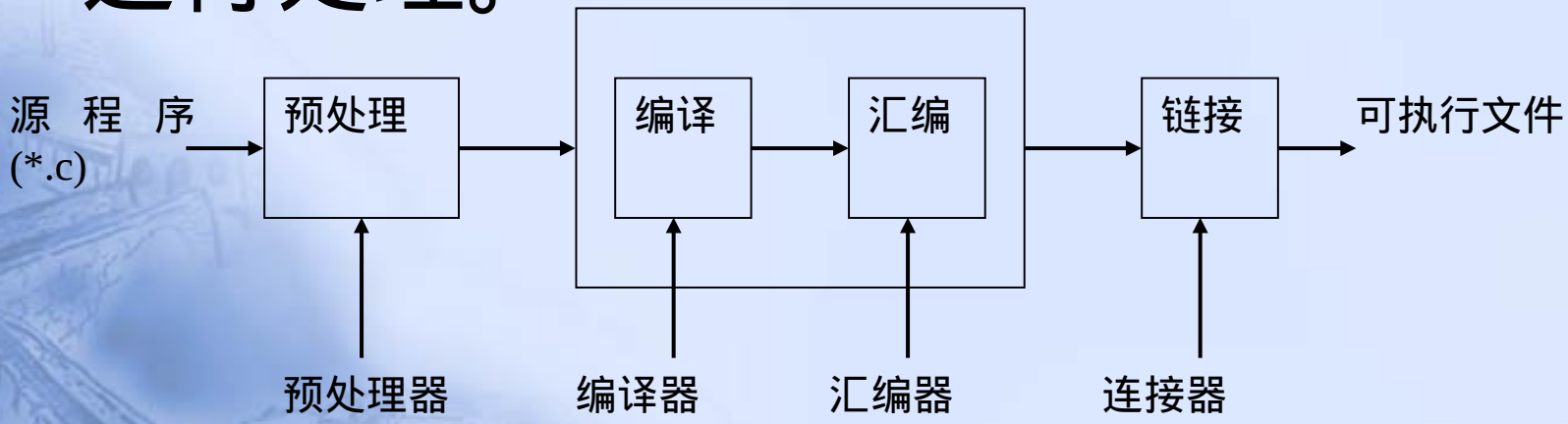
- vim 可以根据用户的不同需要来做一些设定，这些设定都是在末行模式下进行的。
- :set nu　显示行号
- :set nonu　关闭显示行号的功能
- :set nohlsearch　消除搜索的记号
- :set ic　忽略大小写，主要是为了方便搜索
- :set noic　不忽略大小写
- :syntax enable/on　打开色彩支持
- :syntax off　关闭色彩支持

gcc 的使用格式

- $ gcc [options][filenames]
- 其中 filenames 为所要编译的程序源文件。
- 当使用 gcc 时， gcc 会完成预处理、编译、汇编和连接。前三步分别生成目标文件，连接时，把生成的目标文件链接成可执行文件。 gcc 可以针对支持不同的源程序文件进行不同处理，文件格式以文件的后缀来识别。

程序编译过程

- gcc 可以使程序员灵活地控制编译过程。编译过程一般可以分为下面四个阶段，每个阶段分别调用不同的工具进行处理。



后缀名	所对应的语言
-c	只是编译不链接，生成目标文件“ .o”gcc –c
-S	只是编译不汇编，生成汇编代码“ .s”gcc –S
-E	只进行预编译，不做其他处理“ .i”gcc –E –o
-g	在执行程序中包含标准调试信息
-o file	把输出文件输出到 file 里 as a.s –o a.o
-v	打印编译器版本信息
-I dir	增加头文件的搜索范围
-L dir	增加库文件的搜索范围
-Wall	显示告警信息
-l	指定需要使用的库文件
-fpic/fPIC	生成位置无关的目标代码
-shared	产生共享库，在创建共享库时使用
-DM	相当于在程序中添加 #define M 1(用于调试)

使用优化选项

- 当用 gcc 编译 C 代码时，它会试着用最少的时间完成编译并且使编译后的代码易于调试．易于调试意味着编译后的代码与源代码有同样的执行次序，编译后的代码没有经过优化。有很多选项可用于告诉 gcc，在耗费更多编译时间和牺牲易调试性的基础上，产生更小更快的可执行文件。这些选项中最典型的是 -O 和 -O2 选项。
  - -O 选项告诉 gcc 对源代码进行基本优化。这些优化在大多数情况下都会使程序执行的更快。
  - -O2 选项告诉 gcc 产生尽可能小和尽可能快的代码。 -O2 选项将使编译的速度比使用 -O 时慢。但通常产生的代码执行速度会更快。

使用调试选项

- gcc 支持数种调试。在这些选项里最常用的是 -g 选项。
- -g 选项告诉 gcc 产生能被 GNU 调试器使用的调试信息以便调试程序。 gcc 提供了一个很多其他 C 编译器里没有的特性，在 gcc 里能使 -g 和 -O(产生优化代码) 连用。这一点非常有用，因为能在与最终产品尽可能相近的情况下调试代码。同时使用这两个选项时必须清楚所写的某些代码已经在优化时被 gcc 作了改动。

调试工具 gdb

gdb 调试器简介

- Linux 系统中包含了 GNU 调试程序 gdb，它是一个用来调试 C 和 C++ 程序的调试器。可以使程序开发者在程序运行时观察程序的内部结构和内存的使用情况。gdb 所提供的一些功能如下所示：
  - 运行程序，设置所有的能影响程序运行的参数和环境；
  - 控制程序在指定的条件下停止运行；
  - 当程序停止时，可以检查程序的状态；
  - 修改程序的错误，并重新运行程序；
  - 动态监视程序中变量的值；
  - 可以单步执行代码，观察程序的运行状态。

调试工具 gdb

gdb 程序调试的对象是可执行文件，而不是程序的源代码文件。然而，并不是所有的可执行文件都可以用 gdb 调试。如果要让产生的可执行文件可以用来调试，需在执行 gcc 指令编译程序时，加上 -g 参数，指定程序在编译时包含调试信息。调试信息包含程序里的每个变量的类型和在可执行文件里的地址映射以及源代码的行号。 gdb 利用这些信息使源代码和机器码相关联。

gdb 最常用调试命令

- 编译命令
  - gcc –g –o debugme debugme.c
- 进入调试的命令
  - gdb debugme
- 常用调试命令
  - break 行号、 break 函数名、 break 文件名 : 行号 (函数名) 和 info break 、 disable 断点编号、 enable 断点编号
  - run
  - step
  - next
  - print i
  - continue
  - finish
  - quit
  - 回车执行上一条调试命令

Makefile 文件简介

- Makefile
- 一个工程中的源文件不计数，其按类型、功能、模块分别放在若干个目录中， makefile 定义了一系列的规则来指定，哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至于进行更复杂的功能操作，因为 makefile 就像一个 Shell 脚本一样，其中也可以执行操作系统的命令。
- makefile 带来的好处就是——“自动化编译”，一旦写好，只需要一个 make 命令，整个工程完全自动编译，极大的提高了软件开发的效率。 make 是一个命令工具，是一个解释 makefile 中指令的命令工具。
- make 工具最主要也是最基本的功能就是通过 makefile 文件来描述源程序之间的相互关系并自动维护编译工作。而 makefile 文件需要按照某种语法进行编写，文件中需要说明如何编译各个源文件并连接生成可执行文件，并要求定义源文件之间的依赖关系。

一个简单的 Makefile 的例子

- test： prog.o code.o
- 	gcc –o test prog.o code.o
- prog.o： prog.c code.h
- 	gcc –c prog.c –o prog.o
- code.o： code.c code.h
- 	gcc –c code.c –o code.o
- clean：
- 	rm –f *.o

Makefile 编写规则

- 一个 Makefile 文件主要含有一系列的规则，每条规则包含以下内容。
  - 一个目标（ target ），即 make 最终需要创建的文件，如可执行文件和目标文件；目标也可以是要执行的动作，如“ clean”。
  - 一个或多个依赖文件（ dependency ）的列表，通常是编译目标文件所需要的其他文件。
  - 一系列命令 (command)，是 make 执行的动作，通常是把指定的相关文件编译成目标文件的编译命令，每个命令占一行，且每个命令行的起始字符必须为 TAB 字符。

make 命令

- Makefile 写好之后，每次改变了某些源文件，只要执行 make 命令：
- # make
- 所有必要的重新编译将执行。 make 程序利用 makefile 中的数据和每个文件的最后修改时间来确定那个文件需要更新，对于需要更新的文件， make 程序执行 makefile 数据中定义的命令来更新。

一个带变量的 Makefile 的例子

- OBJS=prog.o code.o
- CC=gcc
- test： ${ OBJS }
- ${ CC } –o test ${ OBJS }
- prog.o： prog.c prog.h code.h
- ${ CC } –c prog.c –o prog.o
- code.o： code.c code.h
- ${ CC } –c code.c –o code.o
- clean：
- rm –f *.o

预定义变量

命令格式	含义
$*	不包含扩展名的目标文件名称
$+	所有的依赖文件，以空格分开，并以先后为顺序，可能包含重复的依赖文件
$<	第一个依赖文件的名称
$?	所有时间戳比目标文件晚的依赖文件，并以空格分开
$@	目标文件的完整名称
$^	所有不重复的依赖文件，以空格分开

Makefile 的隐含规则

在上面的例子中，几个产生目标文件的命令都是从“ .c” 的 C 语言源文件和相关文件通过编译产生“ .o” 目标文件，这也是一般的步骤。实际上， make 可以使工作更加自动化，也就是说， make 知道一些默认的动作，它有一些称作隐含规则的内置的规则，这些规则告诉 make 当用户没有完整地给出某些命令的时候，应该怎样执行。

一个隐含规则的 Makefile 的例子

- OBJS=prog.o code.o
- CC=gcc
- test： ${ OBJS }
- ${ CC } –o $@ $^
- prog.o： prog.c prog.h code.h
- code.o： code.c code.h
- clean：
- rm –f *.o