Linux基础知识

一、琐碎知识

- PATH变量：设定解释器搜索所执行的命令的路径。
  - /bin 存放 **所有用户** 皆可用的 **系统程序**，系统启动或者系统修复时可用（在没有挂载 /usr 目录时就可以使用）
  - /sbin 存放 **超级用户** 才能使用的系统程序
  - /usr/bin 存放所有用户都可用的 **应用程序**
  - /usr/sbin 存放超级用户才能使用的应用程序
  - /usr/local/bin 存放所有用户都可用的 **与本地机器无关的程序**
  - /usr/local/sbin 存放超级用户才能使用的与本地机器无关的程序

```
[root@master yujixuan]$ echo $PATH
/usr/local/sbin:/sbin:/bin:/usr/sbin:/usr/bin:/opt/java/jdk1.8.0_333/bin:/opt/mod
ule/hadoop-3.3.3/bin:/opt/module/hadoop-3.3.3/sbin
```

- usr目录是指 Unix System Resource目录，不是user目录。
- Linux 系统中，有 5 种常见的进程状态，分别为运行、中断、不可中断、僵死与停止.
  - 运行：运行中或在运行队列中等待
  - 中断：进程休眠，等待某种条件或信号，则脱离该状态
  - 不可中断：进程不响应系统异步信号，kill也杀不死
  - 僵死：进程已终止，但进程描述符依然存在，直到父进程调用wait4()后将进程释放。
  - 停止：进程收到停止信号后停止运行。

九十九、CMake

1.make

- make 通过调用 makefile 文件中用户指定的命令来进行编译和链接。
- make工具知道如何从名为“makefile”的文件来构建程序。
  - makefile列出了构建过程中涉及的每个非源文件以及如何从其他文件中计算它。
- 每当我们改变系统的一小部分时，重新编译整个程序将是低效的。因此，如果您更改了一些源文件，然后运行“Make”，它不会重新编译整个事情。它仅更新直接或间接依赖于您更改的源文件的那些非源文件。

2.cmake基础

- 允许开发者编写与平台无关的CMakeList.txt文件来制定整个编译流程。
- 然后再根据目标用户的平台进一步自动生成所需的本地化 Makefile 和工程文件。

```
# 定义工程名称、语言 隐式定义了<project>_BIANRY_DIR和<project>_SOURCE_DIR
```

```
# 但建议直接使用 PROJECT_BINARY_DIR和PROJECT_SOURCE_DIR
PROJECT(projectname [CXX] [C] [Java])
# 显示定义变量
SET(VAR [VALUE] [CACHE TYPE DOCSTRING [FORCE]])
# 向终端输出用户定义的信息
MESSAGE([SEND_ERROR | STATUS | FATAL_ERROR] "message to display" ...)
# 定义该工程会产生源文件为SRC_LIST的hello可执行文件
ADD_EXECUTABLE(hello ${SRC_LIST})

# 清理构建结果 可以增量编译
# make clean

# 外部构建时 PROJECT_BINARY_DIR是编译路径 即 工程目录/build;   PROJECT_SOURCE_DIR是工程目录;

# 向当前工程添加存放源文件的子目录，并可以指定中间二进制和目标二进制的存放位置
build/${binary_dir}
# EXCLUDE_FROM_ALL表示编译时排除该目录
ADD_SUBDIRECTORY(source_dir [binary_dir] [EXCLUDE_FROM_ALL])
# 重定向最终目标二进制的位置 哪里添加ADD_EXECUTABLE或ADD_LIBRARY 就在哪里添加这些
SET(EXECUTABLE_OUTPUT_PATH ${PROJECT_BINARY_DIR}/bin)
SET(LIBRARY_OUTPUT_PATH ${PROJECT_BINARY_DIR}/lib)

# 安装 目标文件、普通文件、非目标可执行程序、目录
# CMakelists.txt
INSTALL(TARGETS targets...
[[ARCHIVE|LIBRARY|RUNTIME]
[DESTINATION <dir>]
[PERMISSIONS permissions...]
[CONFIGURATIONS
[Debug|Release|...]]
[COMPONENT <component>]
[OPTIONAL]
] [...])
INSTALL(FILES files... DESTINATION <dir>
[PERMISSIONS permissions...]
[CONFIGURATIONS [Debug|Release|...]]
[COMPONENT <component>]
[RENAME <name>] [OPTIONAL])
INSTALL(PROGRAMS files... DESTINATION <dir>
[PERMISSIONS permissions...]
[CONFIGURATIONS [Debug|Release|...]]
[COMPONENT <component>]
[RENAME <name>] [OPTIONAL])
INSTALL(DIRECTORY dirs... DESTINATION <dir>
[FILE_PERMISSIONS permissions...]
[DIRECTORY_PERMISSIONS permissions...]
[USE_SOURCE_PERMISSIONS]
[CONFIGURATIONS [Debug|Release|...]]
[COMPONENT <component>]
[[PATTERN <pattern> | REGEX <regex>]
[EXCLUDE] [PERMISSIONS permissions...]] [...])

# 然后执行安装
# cmake -DCAKE_INSTALL_PREFIX=/usr/local ..
# make
```

```
# make install

# 添加库
ADD_LIBRARY(libname [SHARED|STATIC|MODULE] [EXCLUDE_FROM_ALL] source1 source2 ...
sourceN)
# 指定库输出的名称 如果是动态库 还可以指定版本和API版本
SET_TARGET_PROPERTIES(target1 target2 ...
PROPERTIES prop1 value1
prop2 value2 ...)

# 找库
# 向工程添加多个特定的头文件搜索路径，路径之间用空格分割，如果路径中包含了空格，可以使用双引号将它括起来
# 默认的行为是追加到当前的头文件搜索路径的后面
INCLUDE_DIRECTORIES([AFTER|BEFORE] [SYSTEM] dir1 dir2 ...)
# CMAKE_INCLUDE_DIRECTORIES_BEFORE，通过SET这个cmake变量为on，可以将添加的头文件搜索路径放在已有路径的前面

# 链接库
# 添加非标准的共享库搜索路
LINK_DIRECTORIES(directory1 directory2 ...)
# 为target添加需要链接的共享库
TARGET_LINK_LIBRARIES(target library1
<debug | optimized> library2
...)

# 在指定路径搜索文件名
FIND_PATH(myHeader NAMES hello.h PATHS /usr/include
/usr/include/hello)
```

3.cmake命令

- file && list

```
# 宏定义 函数名是add_glob 参数是cur_list
macro(add_glob cur_list)
    # CONFIGURE_DEPENDS: CMake将在编译时给主构建系统添加逻辑来检查目标，以重新运行 GLOB 标志的命令。如果任何输出被改变，CMake都将重新生成这个构建系统。
    # RELATIVE: 将返回指定路径（CMAKE_CURRENT_SOURCE_DIR）的相对路径
    # GLOB: 当前目录下的所有;          GLOB_RECURSE: 当前目录及其子目录;
    # ARGN 指除了宏或函数指定的参数外的参数列表
    file(GLOB __tmp CONFIGURE_DEPENDS RELATIVE ${CMAKE_CURRENT_SOURCE_DIR}
${ARGN})
    list(APPEND ${cur_list} ${__tmp})
endmacro()

# 产生一个匹配 <globbing-expressions> 的文件列表并将它存储到变量 <variable> 中
file(GLOB <variable>
     [LIST_DIRECTORIES true|false] [RELATIVE <path>] [CONFIGURE_DEPENDS]
     [<globbing-expressions>...])

list (subcommand <list> [args...])
# subcommand为具体的列表操作子命令，例如读取、查找、修改、排序等。<list>为待操作的列表变量，[args...]为对列表变量操作需要使用的参数表，不同的子命令对应的参数也不一致。
```

- add_library

```
add_library(<name> [STATIC | SHARED | MODULE]
            [EXCLUDE_FROM_ALL]
            [source1] [source2 ...])
# 添加名为name的库，库的源文件可指定，也可用target_sources()后续指定。
# 库的类型是STATIC(静态库)/SHARED(动态库)/MODULE(模块库)之一。
# 设置EXCLUDE_FROM_ALL，可使这个library排除在all之外，即必须明确点击生成才会生成
```

- link_libraries

```
# 将库链接到稍后添加的所有目标。
link_libraries([item1 [item2 [...]]]
               [[debug|optimized|general] <item>] ...)
```

- target_link_libraries

```
# 指定链接给定目标和/或其依赖项时要使用的库或标志。将传播链接库目标的使用要求。目标依赖项的使用要求会影响其自身源的编译。
target_link_libraries(<target> ... <item>... ...)
target_link_libraries(<target>
                      <PRIVATE|PUBLIC|INTERFACE> <item>...
                     [<PRIVATE|PUBLIC|INTERFACE> <item>...]...)

# PUBLIC 在public后面的库会被Link到你的target中，并且里面的符号也会被导出，提供给第三方使用。
# PRIVATE 在private后面的库仅被link到你的target中，并且终结掉，第三方不能感知你调了啥库
# INTERFACE 在interface后面引入的库不会被链接到你的target中，只会导出符号。
```

- include_directories

```
# 将给定的目录添加到编译器用来搜索头文件的目录中。
include_directories([AFTER|BEFORE] [SYSTEM] dir1 [dir2 ...])
```

- target_include_directories

```
# 指定编译给定目标时要使用的include目录。名为< target >的必须由add_executable()或add_library()等命令创建的
# INTERFACE、PUBLIC和PRIVATE关键字用于（指定target_include_directories的影响范围）
target_include_directories(<target> [SYSTEM] [AFTER|BEFORE]
  <INTERFACE|PUBLIC|PRIVATE> [items1...]
  [<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])
```

- add_compile_definitions

```
# 将预处理器定义添加到源文件的编译中
add_compile_definitions(<definition> ...)
```

- add_compile_options

```
# 当前及以下目录编译目标时，使用这些选项
add_compile_options(<option> ...)
```

- set

```
# 一般变量
## variable：只能有一个
## value：可以有0个，1个或多个，当value值为空时，方法同unset，用于取消设置的值
## PARENT_SCOPE(父作用域)：作用域，除PARENT_SCOPE外还有function scope(方法作用域)和
directory scope(目录作用域)
set(<variable> <value>... [PARENT_SCOPE])

# 缓存变量
##  type(类型)：必须为以下中的一种：
    # BOOL：有ON/OFF，两种取值
    # FILEPATH：文件的全路径
    # PATH：目录路径
    # STRING：字符串
    # INTERNAL：字符串
##  docstring：总结性文字(字符串)
## [FORCE]：变量名相同，第二次调用set方法时，第一次的value将会被覆盖
set(<variable> <value>... CACHE <type> <docstring> [FORCE])

# 环境变量
# variable 和 value 一般都只能有一个； value多个时，取最近一个
set(ENV{<variable>} [<value>])
```

- set_property

```
# 在给定范围内设置一个对象的属性。
set_property(<GLOBAL                        |              # 全局有效，名称需要唯一
             DIRECTORY [<dir>]              |              # 指定目录内有效
             TARGET    [<target1> ...]      |              # 设置target属性
             SOURCE    [<src1> ...]                       # 源文件； 源文件同一目录下有
效
                       [DIRECTORY <dirs> ...]
                       [TARGET_DIRECTORY <targets> ...] |
             INSTALL   [<file1> ...]        |              # 属性对应已安装文件
             TEST      [<test1> ...]        |              # 属性对应现有测试
             CACHE     [<entry1> ...]       >              # 属性对应缓存现有条目
            [APPEND] [APPEND_STRING]
            PROPERTY <name> [<value1> ...])

# 其基本格式为：
set_property(<Scope> [APPEND] [APPEND_STRING] PROPERTY <name> [value...])
```

- configure_file

```
# 复制一份输入文件到输出文件，替换输入文件中被@VAR@或者${VAR}引用的变量值。也就是说，让普通文
件，也能使用CMake中的变量
configure_file(<input> <output>
               [COPYONLY] [ESCAPE_QUOTES] [@ONLY]           # 仅复制、转义符号、仅@
               [NEWLINE_STYLE [UNIX|DOS|WIN32|LF|CRLF] ])   # 换行风格
```

一百、常见命令

- mv：用于剪切文件或将文件重命名， 格式为“mv [选项] 源文件 [目标路径|目标文件名]”。剪切操作不同于复制操作，因为它会默认把源文件删除掉，只保留剪切后的文件。如果在同一个目录中对一个文件进行剪切操作，其实也就是对其进行重命名。
- ln：创建链接，软链接(-s)。

```
ln -s source target
ln source target
```

- tar：用于对文件进行打包压缩或解压
  - -c：压缩文件
  - -x：解压文件
  - -z：用Gzip压缩或解压
  - -j：用bzip2压缩或解压
  - -v：显示过程
  - -f：目标文件名
  - -C：指定输出路径
- chmod(change mode)：改变用户对文件的权限
  - 文件所有者u、用户组g、其它用户o、所有用户a。
  - r 表示可读取，w 表示可写入，x 表示可执行，X只有当文件为目录文件，或者其他类型的用户有可执行权限时，才将文件权限设置可执行。
  - -R递归，-v详情
- cp：复制文件和目录。
  - -r：递归，-l：不复制文件，只生成链接，-i：覆盖询问，-f：覆盖不询问，-p：同时复制修改时间和访问权限。

```
cp [options] source dest
```

- stat：用于显示文件或文件系统的详细信息

```
stat [OPTION]... FILE..
```

100.1 功能

> 文件中查找关键字

```
# vim
在命令行模式下输入"/关键字"
# 文件中查找
cat 文件名 | grep "关键字"
# 目录下查找
grep -r "关键字" 目录
```

100.2 强行重启

```
按住art和SysRq(printscreen)键时，输入的一切都会直接交给Linux内核来处理。
reisub中的每一个字母都是一个独立操作，他们分别表示：
Alt+SysRq+r 把键盘从X手中夺过来
Alt+SysRq+e 终结所有进程
Alt+SysRq+i 强制关闭所有进程
Alt+SysRq+s 同步所有挂载的文件系统
Alt+SysRq+u 重新挂载所有的文件系统为只读
Alt+SysRq+b 重启
```