Makefile

Cygwin

- Cygwin is a Unix-like environment and command-line interface for Microsoft Windows.
- Cygwin provides native integration of Windows-based applications, data, and other system resources with applications, software tools, and data of the Unix-like environment. Thus it is possible to launch Windows applications from the Cygwin environment, as well as to use Cygwin tools and applications within the Windows operating context.
- Cygwin consists of two parts: a Dynamic-link library (DLL) as an API compatibility layer providing a substantial part of the POSIX API functionality, and an extensive collection of software tools and applications that provide a Unix-like look and feel.
- Cygwin was originally developed by Cygnus Solutions and was acquired by Red Hat. It is free and open source software, released under the GNU General Public License version 2.

Toolchain (1/2)

- In software, a toolchain is the set of computer programs (tools) that are used to create a product (typically another computer program or system of programs).
- The tools may be used in a chain, so that the output of each tool becomes the input for the next, but the term is used widely to refer to any set of linked development tools.
- A simple software development toolchain consists of a text editor for editing source code, a compiler and linker to transform the source code into an executable program, libraries to provide interfaces to the operating system, and a debugger.
- A complex product such as a video game needs tools for preparing sound effects, music, textures, 3-dimensional models, and animations, and further tools for combining these resources into the finished product.

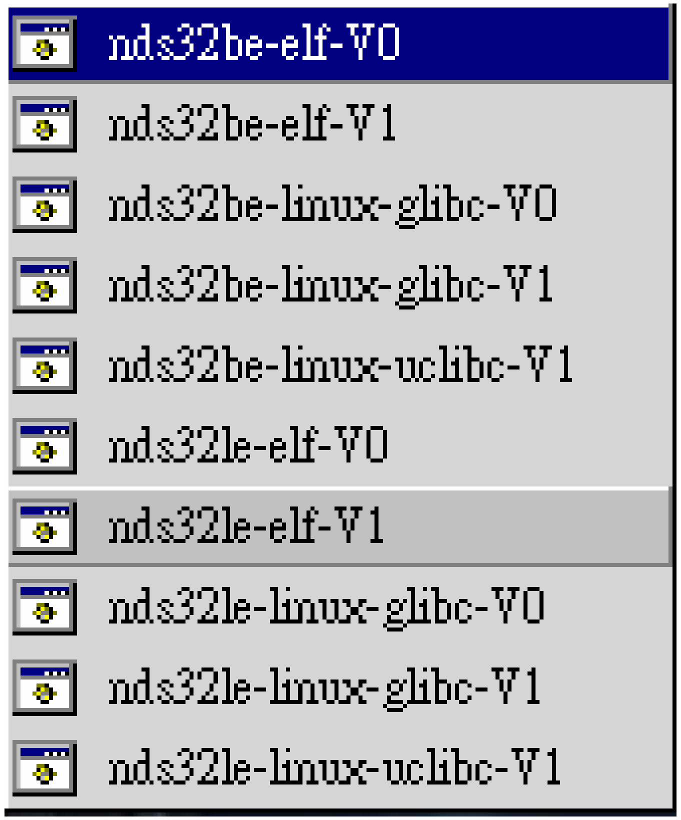
Toolchain (2/2)

GNU toolchain

- The **GNU toolchain** is a blanket term for a collection of programming tools produced by the GNU Project. These tools form a toolchain (suite of tools used in a serial manner) used for developing applications and operating systems.
- The GNU toolchain plays a vital role in development of Linux kernel, BSD, and software for embedded systems. Parts of the GNU toolchain are also directly used with or ported to other platforms such as Solaris, Mac OS X, Microsoft Windows (via Cygwin and MinGW/MSYS) and Sony PlayStation 3
- Projects included in the GNU toolchain are:
  - GNU make: Automation tool for compilation and build
  - GNU Compiler Collection (GCC): Suite of compilers for several programming languages
  - GNU Binutils: Suite of tools including linker, assembler and other tools
  - GNU Bison: Parser generator
  - GNU m4: m4 macro processor
  - GNU Debugger (GDB): Code debugging tool
  - GNU build system (autotools)
    - Autoconf
    - Autoheader
    - Automake
    - Libtool

Andes toolchain

- ❖ Project creation – tool chain selection
  - Selection based on
    - Cores or Architecture
    - CPU endian type
    - Libraries



nds32be-elf-V0
nds32be-elf-V1
nds32be-linux-glibc-V0
nds32be-linux-glibc-V1
nds32be-linux-uclibc-V1
nds32le-elf-V0
nds32le-elf-V1
nds32le-linux-glibc-V0
nds32le-linux-glibc-V1
nds32le-linux-uclibc-V1

Makefile (1/17)

- make指令格式
  - make [option] [target] option就是上面的設定項目，而target等等會講到，就是我們將要產生出來的目標，可以接很多個目標，如果目標不寫的話預設是all。
  - Example:

```
make -n all clean
make install
make
make -f makefile2 install
```

Makefile (2/17)

- 撰寫makefile檔案
- makefile是由一堆「目標」和其「相依性檔案」還有「法則」所組成的，而法則在寫的時候前面不可以使用空格，只能使用Tab鍵，而且同一法則要換行的話需要使用'\'字元，而要加入註解的話要用'#'爲開頭字元。
  - [target] 目標 - 產生出來的東西或某個項目
  - [dependency] 相依性項目 - 目標受相依檔案改變需要重新產生目標
  - [rule] 法則 - 如何讓相依性項目編譯和連結來產生目標

Makefile (3/17)

❖ Example:

```
#這是makefile的格式(註解)
[target]: [dependency] [dependency]
[TAB][rule]
[TAB][rule]
[target]: [dependency]
[TAB][rule]
```

整個makefile就是利用上面的格式，目標和相依性項目還有法則，組合之後就可以編出一個執行檔了，我們下面就來舉個最容易的範例。

Makefile (4/17)

- make的巨集(macro)
  - Example:

    CC = gcc 指定
    $(CC) 叫用
    CFLAGS = -ansi -Wall -g 指定
    $(CFLAGS) 叫用
- 有幾個特別的內部巨集，讓makeifle 更加簡明，每個巨集都是在使用之前才被展開，所以巨集的意義隨makefile的處理而有所不同。
  - $? 代表需要重建的相依性項目
  - $@ 目前的目標項目名稱
  - $< 代表第一個相依性項目
  - $* 代表第一個相依性項目，不過不含副檔名

Make (5/17)

- In software development, **make** is a utility for automatically building executable programs and libraries from source code.
- Files called **makefiles** specify how to derive the target program from each of its dependencies.
- **Make** can decide where to start through topological sorting.
- Though Integrated Development Environments and language-specific compiler features can also be used to manage the build process in modern systems, *make* remains widely used, especially in Unix-based platforms.

Make (6/17)

- **Makefile structure**
  - A makefile consists of lines of text which define a file (or set of files) or a rule name as depending on a set of files. Output files are marked as depending on their source files, for example, and on files which they include internally, since they all affect the output. After each dependency is listed, a series of lines of tab-indented text may follow which define how to transform the input into the output, if the former has been modified more recently than the latter. In the case where such definitions are present, they are referred to as "build scripts" and are passed to the shell to generate the target file. The basic structure is:

Make (7/17)

- # Comments use the hash symbol

```
target: dependencies
        command 1
        command 2
        . . .
        command n
```

Makefile (8/17)

```
target…: dependencies
<tab>command
<tab>command
.
.
.
```

Makefile (9/17)

- **Example1**

```
hello:hello.o     #(target…: dependencies)
        gcc -o hello hello.o #(<tab>command)
hello.o:hello.c   #(target…: dependencies)
        gcc -c hello.c #(<tab>command)
```

Makefile (10/17)

- **Example1: hello.c**

  -
    ```
    # it is a test
    all:hello.c
        gcc hello.c -o hello
    clean:
        rm -f hello
    ```
- 執行 # make, 就會把名爲hello 的執行檔編譯出來當我們輸入 # make clean, 就會把名爲hello 的執行檔給刪除，而Makefile內的#符號指的是註解。
  ps: 注意: gcc hello.c -o hello 前的空白，請使用tab鍵，千萬別使用space鍵。

Makefile (11/17)

- Example2:建立a.h, b.h, c.h, main.c, 2.c, 3.c 六個檔案

```
mytest:main.o 2.o 3.o
	gcc -o mytest main.o 2.o 3.o
main.o:main.c a.h
	gcc -c main.c
2.o: 2.c a.h b.h
	gcc -c 2.c
3.o:3.c b.h c.h
	gcc -c 3.c
```

Makefile (12/17)

- 打上 # make -f Makefile1執行後，make就會去找第一行的必要條件: main.o 2.o 3.o，當在目錄裡卻沒這三個檔，所以make會在Makefile1往下找main.o/2.o/3.o分別把這三個檔案給編出來，然後再把mytest這個執行檔給編譯出來，-f 是什麼？第一次我們已建立一個名Makefile檔，現在我們又建立一個Makefile1檔，所以-f是告訴make去執行所指定的Makefile檔。

Makefile (13/17)

- Example4:clean.c，hello.c, hello.o及Makefile檔，其內容如下

```
all:hello.c
    gcc -c hello.c -o hello.o
    gcc hello.c -o hello
clean:
    rm -f hello.o
```

- 此時當我們執行make clean時就會發現，都沒動，hello.o沒有被刪除，因爲make檔一執行就會先去找檔案，沒有檔案時才會執行其它必要條件，所以make clean 認爲clean檔案已是存在的所以不會執行 rm -f hello.o

Makefile (14/17)

- 那怎麼辦呢?只要多加上一行命令就行了:

```
.PHONY:clean
all:hello.c
	gcc -c hello.c -o hello.o
	gcc hello.c -o hello
clean:
	rm -f hello.o
```

這樣make就不會把clean看成檔案，所以會無條件執行了

Makefile (15/17)

❖ 說內隱規則(Implicit Rules)，來個Makefile檔簡單的例子(我們現有hello.c及a.h兩個檔)

```
.PHONY:clean

hello:hello.o
    gcc $< -o $@

%.o:%.c a.h
    gcc -c $< -o $@

clean:
    rm -fr hello.o
    rm -fr hello
```

Makefile (16/17)

- %表示所相對於後面必要條件的檔名的意思，也就是所謂的樣式規格(pattern rule)，hello.o就會去找hello.c，hello.o就會去找hello.c，第一行前面說過了, 再來, gcc $< 指將指的是hello.o，而$@只的就是hello，所以一開始我們沒有hello.o，就會往下找到 %.o:%c a.h，可以把這行看成: hello.o: hello.c a.h, 而下一行，$<指的就是hello.c及a.h, $@指的就是hello.o

Makefile (17/17)

- make常用指令
    - make -k: 會讓make在遇到錯誤的時候仍然運行，而不會停在第一個問題
    - make -n: 只印出將會進行的工作，而不會真的去執行
    - make -f makefile_name: 告訴make需要用那個makefile檔案。

ANDES TECHNOLOGY

Thank You

WWW.ANDESTECH.COM