kmk 文件输入插件手册

滇狐
edyfox@sohu.com
version: 0.06.8287

February 19, 2006

What's new?

0.07.8390: 2006 年 2 月 14 日

1. 自动包含 `default.inc` 文件。
2. 修正了处理绝对路径时的错误。

0.06.8287: 2005 年 11 月 3 日

1. 允许定义自定义编译变量。
2. 允许通过定义扩展名的编译方式，为一系列同种类型的文件定义自定义编译选项。

0.05.8217: 2005 年 8 月 25 日

1. `#include` 后如果使用尖括号，则会先在`~/.kittie/include` 中查找包含文件，然后在 `/usr/local/share/kittie/include` (Linux / Unix 环境) 或 `<安装路径>\include` (Windows 环境) 中查找包含文件。

0.04.8133: 2005 年 6 月 2 日

1. `#platform` 后支持多个参数，空格分隔。
2. 实现了 .all 块，编写 kmk 变得更加简便。
3. 修正了无 configuration 块的文件无法被包含的故障。

0.03.8091: 2005 年 4 月 21 日

1. 修正了 Windows 下相对路径调整的错误。
2. 修正了不能在工程根目录下添加文件的错误。
3. 修正了 `include` 语句相对路径调整的错误。

0.02.8048: 2005 年 3 月 9 日

1. 实现了 `include` 功能。

0.01.8042: 2005 年 3 月 4 日

1. 首次公开发布。

1 文件格式简介

kmk 格式是 kittie 工程文件的格式，它由纯文本组成，能够很方便地手工编写，而且具有较强的表现能力。Kittie.Concept 对 .kmk 格式进行了扩展，让它能够支持自定义编译选项，并且在未来不久内，还会为 .kmk 格式提供更多的功能。

一个合法的 kmk 脚本由三部分组成，第一部分是 Configuration，第二部分是 Customized Build，第三部分是 File list。

1.1 Configutation 块

下面是一个 Configuration 块的定义。方括号内的部分是 Configuration 名称，其后的部分是该 Configuration 的定义。示例中列出了所有合法的 Configuration 的选项，其中分号开头的行为注释。

```
[Debug]
; 用于存放 .o 文件的文件夹
OBJ_DIR = Debug
; 用于存放可执行文件 .exe 的文件夹
OUTPUT_DIR = Debug
; 可执行文件名。在 Windows 平台下会自动加上 .exe
TARGET = hello
; 附加头文件路径，各项之间用空格分隔，每项以 -I 开头
C_INCLUDE_DIRS = -I. -I../euc
; 预处理定义符，各项之间用空格分隔，每项以 -D 开头
C_PREPROC = -DWIN32 -D_DEBUG
; 编译器参数，各项之间用空格分隔
COMMONFLAGS = -Wall -g
; 专为 C 设置的编译器参数，各项之间用空格分隔
CFLAGS = -Wall -g
; 专为 C++ 设置的编译器参数，各项之间用空格分隔
CXXFLAGS = -Wall -g
; 资源文件附加头文件路径，各项之间用空格分隔，每项以 -I 开头
RC_INCLUDE_DIRS =
; 资源文件预处理定义符，各项之间用空格分隔，每项以 -D 开头
; 资源文件编译器参数，各项之间以空格分隔
RCFLAGS =
; 附加库文件路径，各项之间以空格分隔，每项以 -L 开头
LIB_DIRS =
; 附加库文件名，各项之间以空格分隔，每项以 -l 开头
LIBS =
; 链接器参数，各项之间以空格分隔
LDFLAGS =
```

在同一个配置中，如果一个已经出现过的配置项再次出现时，它会替换掉原有的取值。例如，如果曾经出现过：

`TARGET = hello`

之后再次出现：

`TARGET = world`

的时候，TARGET 的取值将是“world”。

对于除了“OBJ_DIR”、“OUTPUT_DIR”和“TARGET”外的其它配置项，还可以使用“+=”语法，将新的配置与原有配置合并。例如，如果曾经出现过：

C_INCLUDE_DIRS = -I.

之后再出现：

C_INCLUDE_DIRS += -I../euc

的时候，C_INCLUDE_DIRS 的取值将是“-I. -I../euc”。

一个工程可以同时存在多个不同的 Configuration，使用不同的命令来编译同一组文件。一个典型的工程一般会包含 Debug 和 Release 两个 Configuration。

有一个特殊的 Configuration 块，叫做 .all 块。.all 块自己不会成为一个单独的 Configuration，但它里面出现的选项，会被添加到其它每一个 Configuration 中。

对于“OBJ_DIR”、“OUTPUT_DIR”和“TARGET”配置项，如果在一个块中未被定义，则该块会使用 .all 块中的相应配置项的值；如果一个块中已定义这几个配置项，则忽略 .all 块中的定义。例如，在下列 kmk 文件中，由于 Debug、Release 块中均未出现 TARGET 定义，因此 Debug、Release 块均会自动采用 .all 中的定义，即“TARGET = test”。

```
[.all]
TARGET = test

[Debug]
OUTPUT_DIR = Debug
OBJ_DIR = Debug

[Release]
OUTPUT_DIR = Release
OBJ_DIR = Release
```

除上述三项之外的其余 .all 中的配置项会被合并到各个配置中，相当于使用“+=”语法。例如，在下列 kmk 文件中，.all 中的“C_INCLUDE_DIRS”会被合并到 Debug 与 Release 中的 C_INCLUDE_DIRS 里，此时，Debug 下的配置为“C_INCLUDE_DIRS = -Idebug -I../test”，而 Release 的配置为“C_INCLUDE_DIRS = -Irelease -I../test”。

```
[.all]
C_INCLUDE_DIRS = -I../test

[Debug]
C_INCLUDE_DIRS = -Idebug

[Release]
C_INCLUDE_DIRS = -Irelease
```

合理使用 .all 块，可以大大简化 kmk 文件的编写工作。

除了使用内置的配置项变量外，还可以定义自己的配置项变量。定义方法是，在配置项中写上：“EXTRA_ITEMS = 自定义配置项变量名列表”，然后就可以在该配置中使用你的自定义配置项。例如：

```
[.all]
EXTRA_ITEMS = TEST_ITEM_ONE TEST_ITEM_TWO
TEST_ITEM_ONE = test
TEST_ITEM_TWO = test2

[Debug]
TEST_ITEM_ONE += test3
```

1.2 Customized Build 块

在 Customized Build 块中，我们可以指定一些扩展名对应的编译方式，这样在 File list 块中遇到这些扩展名的时候，就没有必要一一为它们指定自定义编译方式了。

一个 Customized Build 块由以下五部分组成：

1. 文件名模式，格式为“<*.扩展名>”，例如：

   <*.po>

2. 说明文字，格式如下：

   % 说明文字

   它用于说明现在正在完成什么操作。

3. 命令行，它的格式如下：

   !命令行 1

   !命令行 2

   ...

   它指定了进行自定编译时使用的命令。

4. 附加依赖项，它的格式如下：

   +附加依赖项 1

   +附加依赖项 2

   ...

   它指定了编译该文件需要的附加依赖项，该部分可以为空。

5. 输出文件，它的格式如下：

   =输出文件

   它用于说明编译该选项得到的输出文件。

在定义编译选项的时候，可以使用以下宏：

$(INPUT_DIR)：输入文件所在路径，路径名包含最右边的斜杠。

$(INPUT_NAME)：输入文件名，不包含路径和扩展名。

$(INPUT_PATH)：输入文件的全路径。

$(INPUT_FILENAME)：输入文件名，不包含路径，包含扩展名。

$(INPUT_EXT)：输入文件的扩展名。

$(PROJECT_DIR)：当前 kmk 文件所在路径，路径名包含最右边的斜杠。

`$(PROJECT_NAME)`：当前 kmk 文件的文件名，不包含路径和扩展名。

`$(PROJECT_PATH)`：当前 kmk 文件的全路径。

`$(INPUT_FILENAME)`：当前 kmk 文件的文件名，不包含路径，包含扩展名。

`$(PROJECT_EXT)`：当前 kmk 文件的扩展名。

1.3 File list 块

Configuration 块之后是 File list 块，用来描述该工程中包含的文件列表。该部分中冒号结尾的行会被看做虚拟文件夹，其余行被看作文件。如果冒号结尾的行含有一对小括号的话，小括号中的内容会被看作该文件夹的扩展名筛选项。在文件列表部分，可以任意创建多级嵌套的文件夹，使用空格或 Tab 来表示文件夹与文件之间的树形关系。注意，如果一个文件夹下既含有子文件夹，又含有文件时，要把文件写在文件夹的前面。如果需要在根文件夹下添加文件，可以在所有文件列表之前添加一个空冒号行：

```
:
readme.txt
Source files:
    Hello.cpp
```

对于文件列表中包含的文件，kittie 根据扩展名生成编译指令。对于 .cpp 和 .c，kittie 调用 gcc 将它编译为 .o 文件；而对于 .rc 文件，kittie 调用 windres 将它编译为 .res 文件[1]

如果有 kittie 不认识的扩展名，可以使用自定义编译指令来指定编译该文件的方法和命令行。自定义编译指令由五部分组成：

1. 说明文字，格式如下：

   `% 说明文字`

   它用于说明现在正在完成什么操作。

2. 命令行，它的格式如下：

   `!命令行 1`

   `!命令行 2`

   `...`

   它指定了进行自定编译时使用的命令。

3. 输入文件，它的格式如下：

   文件名

   它指定了待编译的文件名。

4. 附加依赖项，它的格式如下：

   `+附加依赖项 1`

   `+附加依赖项 2`

   `...`

   它指定了编译该文件需要的附加依赖项，该部分可以为空。

[1] 这里的 .res 文件并不是 VC 的 rc 编译器生成的 .res 文件，它实际上是 coff 格式的，但是为了防止出现重名，仍然使用 .res 扩展名。

5. 输出文件，它的格式如下：

   =输出文件

   它用于说明编译该选项得到的输出文件。

在自定义编译选项部分可以使用 Configuration 部分定义的宏，例如，$(OBJ_DIR) 代表 .o 文件输出到的路径。这是一个带有自定义编译指令的 File list 的例子：

```
Source files (*.cpp; *.cxx; *.cc; *.c):
    lex.yy.cpp
    entry.cpp
    % 生成词法分析文件
    !flex -+ lex.l
    !ccfix lex.yy.cc lex.yy.cpp
    !rm lex.yy.cc
    lex.l
    + parser.yy.h
    = lex.yy.cpp
Header files (*.h; *.hpp; *.hxx):
    FlexLexer.h
```

1.4 #platform 语句

为了方便跨平台开发，kmk 提供了“`#platform`”语句。当“`#platform`”之后的参数与当前平台不符时，kmk 输入插件将会跳过本行到“`#end platform`”行之间的所有内容。因此，可以将一些平台相关的定义放在“`#platform`”块中，实现同一个 .kmk 文件在多个不同平台下使用：

```
#platform win32
LIBS = -lws2_32
#end platform
#platform linux sunos
LIBS = -lpthread
#end platform
#platform freebsd
LIBS = -lc_r
#end platform
```

目前“`#platform`”语句支持 4 种参数：“`win32`”、“`linux`”、“`freebsd`”和“`sunos`”。参数大小写敏感。“`#platform`”语句之后可以接多个参数，各参数间使用空格分隔，例如“`#platform linux sunos`”表示“Linux 平台或者 SunOS 平台”。

1.5 使用 include

编写稍大的工程的时候，总会用到一些第三方的函数库。而在使用第三方库的时候，往往需要配置许多参数，如附加头文件包含路径、库文件名称与路径等等。如果有若干个工程需要使用同一套库的话，每个工程中都需要设置一遍这些参数，显得很是烦琐。

如果工程文件支持包含的话，可以把这些公共设置放在一个单独的文件中，然后每个工程 `include` 那个文件，就可以直接使用该文件中的配置了。

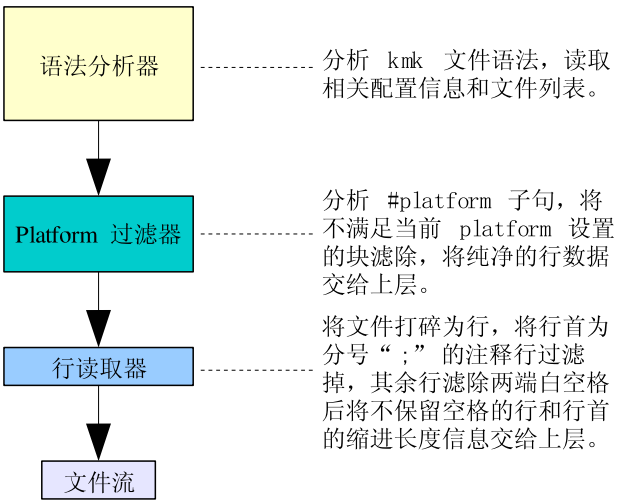

图 1: 插件逻辑层次结构

`#include` 后如果使用尖括号，则会先在`~/.kittie/include` 中查找包含文件，然后在 `/usr/local/share/kittie/include` (Linux / Unix 环境) 或 `<安装路径>\include` (Windows 环境) 中查找包含文件。

`#include` 后如果使用双引号，则使用相对路径进行查找。由于被包含的文件与原文件可能不在同一目录，因此包含文件的时候，kmk 插件会根据需要对相对路径进行自动调整。

另外，在 include 路径下如果存在 `default.inc` 文件的话，会自动被包含到 kmk 文件中，用于提供一些针对某个平台的通用编译参数，如 Linux 下的 `-fPIC` 等。

2 使用说明

该插件支持一个参数：

`-pplatform={win32|linux|freebsd|sunos}`

当“`platform`”指定的值与 .kmk 中定义的 `#platform` 块不同时，该块就会被跳过。“`platform`”的默认值与当前平台名称相同。另外，如果“`platform`”的取值不是“`win32`”的话，插件读入 .kmk 时会忽略所有资源文件（.rc）相关的配置项。

3 插件实现

由于文件格式比较简单，该插件并没有在函数或类的层面上划分层次，但是，在文件分析函数中，逻辑上还是具有一定层次结构的，如图 1 所示。

最底层从文件流中读取到原始的行信息后，进行一些最基本的处理，如记录行号，剔除注释行，去掉两端白空格，以方便上层的处理。由于 kmk 文件使用缩进来表示文件的逻辑目录结构，因此在去掉白空格的同时，还应该保留该行的缩进情况。

中间层主要处理 `#platform` 子句。它负责把不符合当前 `platform` 定义的行去掉，将符合当前定义的行交给上层继续处理。

最上层通过一个简单的有限状态自动机，实现了对 kmk 文件格式的分析和信息的记录。由于 kmk 是 kittie 的内置格式，kittie 能够支持的功能与 kmk 格式的描述能力是基本等价的，因此该插件可以作为开发 kittie 输入插件的一个模板和示例使用。