makefile学习

1 makefile简介

1.1 什么是makefile?

1.2 Makefile规则介绍

一个简单的Makefile描述规则如下所示：

```
TARGET... : PREREQUISITES...
    COMMAND
    ...
    ...
```

target：规则的目标。通常是最后需要生成的文件名或者为了实现这个目标而必需的中间过程文件名。可以是.o文件，也可以是最后的可执行文件等。另外，目标也可以是一个make执行的动作的名称，如目标"clean"，我们称这样的目标是"伪目标"。

prerequisites：规则的依赖。生成规则目标所需要的文件名列表。通常一个目标依赖于一个或者多个文件。

command：规则的命令行。是规则所要执行的动作（任意的shell命令或者是可在shell下执行的程序），它限定了make执行这条规则时所需要的动作。
一个规则可以有多个命令行，每一条命令占一行。**注意：每一个命令必须以[Tab]字符开始，[Tab]字符告诉make此行是一个命令行，make按照命令完成相应的动作，这也是书写Makefile时容易产生且比较隐蔽的错误。**
命令就是在任何一个目标的依赖文件发生变化后重建目标的动作描述。一个目标可以没有依赖而只有动作（指令的命令）。比如Makefile中的目标"clean"，此目标没有依赖，只有命令。它所定义的命令用来删除make过程产生的中间文件（进行清理工作）。

在Makefile中"规则"就是描述在什么情况下如何重建目标的命令，通常规则中包括了目标的依赖关系（目标的依赖文件）和重建目标的命令。make执行重建目标的命令，来创建或者重建规则的目标（此目标文件也可以是触发这个规则的上一个规则中的依赖文件）。规则包含了文件之间的依赖关系和更新此规则目标所需要的命令。

一个Makefile文件中通常还包含了除规则以外的很多东西（后续我们会一步一步的展开）。一个最简单的Makefile可能只包含规则，规则在有些Makefile中可能看起来非常复杂，但是无论规则的书写是多么的复杂，它都符合规则的基本格式。
make程序根据规则的依赖关系，决定是否执行所定义的命令的过程我们称之为**执行规则**。

1.3 简单的示例

本小节开始我们举一个简单的例子。此例子有3个头文件和8个C文件组成。我们将书写一个简单的Makefile，来描述如何创建最终的可执行文件"edit"，此可执行文件依赖于3个头文件和8个C源文件。
Makefile文件的内容如下：

```
#sample Makefile
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
               insert.o search.o files.o utils.o
main.o : main.c defs.h
        cc -c main.c
kbd.o : kdb.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
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h command.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o \
            insert.o search.o files.o utils.o
```

首先书写时，可以将一个较长行使用反斜线（\）来分解为多行，这样可以使我们的Makefile书写清晰、容易阅读。**但需要注意：反斜线之后不能有空格（这也是大家最容易犯的错误，比较隐蔽）。** 我们推荐将一个长行分解为使用反斜线连接的多行形式。
当完成了这个Makefile以后，如果需要创建可执行程序 "edit" ,只需要在包含此Makefile的目录（当然也在代码所在目录）下输入命令 "**make**" 。如果想删除之前使用 "**make**" 生成的文件（包括那些中间过程的.o文件），也只需要输入命令 "**make clean**" 即可。
在这个Makefile中，我们的目标（target）就是可执行文件 "edit" 和那些.o文件（main.o,kbd.o...）;依赖（prerequisites）就是冒号后面的那些.c文件和.h文件，所有的.o文件既是依赖（相对于可执行程序edit）又是目标（相对于.c和.h文件）。命令包括"cc -c main"、"cc -c kbd.c".....
当规则的目标是一个文件，在它的任何一个依赖文件被修改以后，执行"make"时这个目标文件将会被重新编译或者重新连接。当然，此目标的任何一个依赖文件如果有必要则首先会被重新编译。在这个例子中，"edit"的依赖为8个.o文件，而"main.o"的依赖文件为"main.c"和"defs.h"。当"main.c"或者"defs.h"被修改以后，再次执行"make"，"main.o"就会被更新（其他的.o文件不会被更新），同时"main.o"的更新将会导致"edit"被更新。

在描述依赖关系行之下通常就是规则的命令行（存在一些规则没有命令行），命令行定义了规则的动作（如何根据依赖文件来更新目标文件）。命令行必须以 [Tab] 键开始，以和Makefile其他行区别。**就是说所有的命令行必须以 [Tab] 字符开始，但并不是所有的以 [Tab] 键出现的行都是命令行。但make程序会把出现在第一条规则之后的所有以 [Tab] 字符开始的行都作为命令行来处理。** （记住：make程序

本身并不关心命令是如何工作的，对目标文件的更新需要你在规则描述中提供正确的命令。"make"程序所做的就是当目标程序需要更新时执行规则所定义的命令）。

目标 "clean" 不是一个文件，它仅仅代表执行一个动作的标识。正常情况下，不需要执行这个规则所定义的动作，因此目标 "clean" 没有出现在其他任何规则的依赖列表中。因此在执行 "make" 时，它所指定的动作不会被执行。除非在执行make时明确的指定它。而且目标 "clean" 没有任何依赖文件，它只有一个目的，就是通过这个目标名来执行它所定义的命令。**Makefile中把那些没有任何依赖只有执行动作的目标称为“伪目标”（phony targets）** `['founi]`**。** 需要执行"clean"目标所定义的命令，可在shell下输入：*make clean* 。

1.4 make如何工作

默认的情况下，make执行的是Makefile中的第一个规则，此规则的第一个目标称为“最终目的”或者“终极目标”（就是一个Makefile最终需要更新或者创建的目标）
上例的Makefile，目标 "edit" 在Makefile中是第一个目标，因此它就是make的“终极目标”。当修改了任何C源文件或者头文件后，执行make将会重建终极目标"edit"。
当在shell提示符下输入"make"命令以后，make读取当前目录下的Makefile文件，并将Makefile文件中的第一个目标作为其执行的“终极目标”，开始处理第一个规则（终极目标所在的规则）。在我们的例子中，第一个规则就是目标"edit"所在的规则。规则描述了"edit"的依赖关系，并定义了链接.o文件生成目标"edit"的命令;make在执行这个规则所定义的命令之前，首先处理目标"edit"所在的依赖文件（例子中的那些.o文件）的更新规则（以这些.o文件为目标的规则），对这些.o文件为目标的规则处理有下列三种情况：

1. 目标.o文件不存在，使用其描述规则创建它；
2. 目标.o文件存在，目标.o文件所依赖的.c源文件、.h文件中的任何一个比目标.o文件“更新”（在上一次make之后被修改）。则根据规则重新编译生成它；
3. 目标.o文件存在，目标.o文件比它的任何一个依赖文件（.c源文件、.h文件）“更新”（它的依赖文件在上一次make之后没有被修改），则什么也不做。

这些.o文件所在的规则之所以会被执行，是因为这些.o文件出现在“终极目标”的依赖列表中。在Makefile中一个规则的目标如果不是“终极目标”所依赖的（或者“终极目标”的依赖文件所依赖的），那么这个规则将不会被执行，除非明确指定执行这个规则（可以通过make命令行指定重建目标，那么这个目标所在的规则就会被执行，例如“make clean”）。在编译或者重新编译生成一个.o文件时，make同样会去寻找它的依赖文件的重建规则（是这样一个规则：这个依赖文件在规则中作为目标出现），在这里就是.c和.h文件的重建规则。在上例的Makefile中没有那个规则的目标是.c或者.h文件，所以没有重建.c和.h文件的规则。不过C语言源文件可以使用工具Bison或者Yacc来生成（具体用法可参考相应的手册）。

完成了对.o文件的创建（第一次编译）或者更新后，make程序将处理终极目标"edit"所在的规则，分为以下三种情况：

1. 目标文件"edit"不存在，则执行规则以创建目标"edit"。
2. 目标文件"edit"存在，其依赖文件中有一个或者多个文件比它“更新”，则根据规则重新链接生成"edit"。
3. 目标文件"edit"存在，它比它的任何一个依赖文件都“更新”，则什么也不做。

上例中，如果更改了源文件 "insert.c" 后执行make，"insert.o" 将被更新，之后终极目标 "edit" 将会被重新生成;如果我们修改了头文件"command.h"之后运行make，那么“kbd.o”、“command.o”和“files.o”将会被重新编译，之后同样终极目标"edit"也将被重新生成。

以上我们通过一个简单的例子，介绍了Makefile中目标和依赖的关系。我们简单总结一下：
对于一个Makefile文件，"make"首先解析终极目标所在的规则（上节例子中的第一个规则），根据其依赖文件（例子中第一个规则的8个.o文件）依次（按照依赖文件列表从左到右的顺序）寻找创建这些依赖文件的规则。首先为第一个依赖文件（main.o）寻找创建规则，如果第一个依赖文件依赖于其它文件（main.c、defs.h），则同样为这个依赖文件寻找创建规则（创建main.c和defs.h的规则，通常源文件和头文件已经存在，也不存在重建它们的规则）.....，直到为所有的依赖文件找到合适的创建规则。之后make按照终极目标的依赖文件的列表顺序依次完成对规则文件的创建和更新（上例的顺序是"main.o"、"kbd.o"、 "command.o".....）。
创建或者更新每一个规则依赖文件的过程都是这样的一个过程。对于任意一个规则执行的过程都是按照依赖文件列表顺序，对于规则中的每一个依赖文件，使用同样方式（按照同样的过程）去重建它，在完成对所有依赖文件的重建之后，最后一步才是重建此规则的目标。
更新（或者创建）终极目标的过程中，如果任何一个规则执行出现错误make就立即报错并退出。整个过程make只是负责执行规则，而对具体规则所描述的依赖关系的正确性、规则所定义的命令的正确性不做任何判断。就是说，一个规则的依赖关系是否正确、描述重建目标的规则命令行是否正确，make不做任何错误检查。
因此，需要正确的编译一个工程，需要在提供给make程序的Makefile中来保证其依赖关系的正确性和执行命令的正确性。

1.5 指定变量

同样是上边的例子，我们来看一下终极目标"edit"所在的规则

```
edit : main.o kbd.o command.o display.o \
       insert.o seartch.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
```

在这个规则中.o文件列表出现了两次；
第一次：作为目标"edit"的依赖文件列表出现；
第二次：规则命令行中作为"cc"的参数列表。
这样做所带来的问题是：如果我们需要为目标"edit"增加一个依赖文件，我们就需要在两个地方添加（依

赖文件列表和规则的命令行中）。添加时可能在"edit"的依赖文件中加入了，但是却忘记了给命令行中添加，或者相反。这就给后期的维护和修改带来了很多不方便，添加或修改时容易出现遗漏。
为了避免这个问题，在实际工作中大家都比较认同的方法是使用一个变量"objects"、"OBJECTS"、"objs"、"OBJS"或者"OBJ"来作为所有的.o文件的列表来替代。在使用到这些文件列表的地方，使用此变量来代替。在上例的Makefile中我们可以添加这样的一行：

```
objects = main.o kbd.o command.o display.o \
    insert.o search.o files.o utils.o
```

"objects"作为一个变量，它代表所有的.o文件的列表。在定义了此变量后，我们就可以在需要使用这些.o文件列表的地方使用"**$(objects)**"来表示它，而不需要罗列所有的.o文件列表。因此上例的规则就可以这样写：

```
objects = main.o kbd.o command.o display.o \
    insert.o search.o files.o utils.o
edit : $(objects)
        cc -o edit $(objects)
        ...
        ...
clean :
        rm edit $(objects)
```

当我们需要为终极目标"edit"增加或者去掉一个.o依赖文件时，只需要改变"objects"的定义（加入或者去掉若干个.o文件）。这样做不但减少书写的工作量，而且可以减少修改而产生错误的可能。

1.6 自动推导规则

在使用make编译.c源文件时，编译.c源文件规则的命令可以不用明确给出。这是因为make本身存在一个默认的规则，能够自动完成对.c文件的编译并生成对应的.o文件。它执行命令"cc -c"来编译.c源文件。在Makefile中我们只需要给出需要重建的目标文件名（一个.o文件），make会自动为这个.o文件寻找合适的依赖文件（对应的.c文件，对应是指：文件名除后缀外，其余都相同的两个文件），而且使用正确的命令来重建这个目标文件。对于上边的例子，此默认规则就是使用命令"cc -c main.c -o main.o"来创建文件"main.o"。对一个目标文件是"N.o"，依赖文件是"N.c"的规则，完全可以省略其规则的命令行，而由make自身决定使用默认命令。此默认规则称为make的隐含规则。

这样，在书写Makefile时，我们就可以省略描述.c文件和.o文件依赖关系的规则，而只需要给出那些特定的规则描述（.o目标所需要的.h文件）。因此上边的例子就可以以更加简单的方式书写，我们同样使用变量"objects"。Makefile内容如下：

```
# sample Makefile
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)
```

这样格式的Makefile更接近于我们实际应用。

make的隐含规则在实际工程的make中会经常使用，它使得编译过程变得方便。几乎在所有的Makefile中都用到了make的隐含规则，make的隐含规则是非常重要的一个概念。

1.7 另类风格的makefile

上一节中我们提到过，Makefile中，所有的.o目标文件都可以使用隐含规则由make自动重建，我们可以根据这一点书写更加简洁的Makefile。而且在这个Makefile中，我们是根据依赖而不是目标对规则进行分组。形成另外一种风格的Makefile。实现如下：

```
# sample Makefile
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
```

本例中，我们以三个头文件为出发点，对依赖于每一个头文件的目标进行合并。书写出一个多目标规则。规则中多个目标同时依赖于对应的头文件，而且同一个文件可能同时存在多个规则中。例子中头文件"defs.h"作为所有.o文件的依赖文件。其它两个头文件作为规则所有目标文件（多个.o文件）的依赖文件。
这种风格的Makefile并不值得我们借鉴。问题在于：同时把多个目标的依赖文件放在同一个规则（一个规则中含有多个目标文件），这样导致规则定义不明了，比较混乱。建议大家不要在Makefile中采用这

种书写方式，否则后期维护将是一件非常痛苦的事情。
书写规则建议的方式是：**单目标，多依赖。就是说尽量要做到一个规则中只存在一个目标文件，可以有多个依赖文件。尽量避免使用多目标，单依赖的方式。** 这样书写的好处是后期维护会非常方便，而且这样会使Makefile更清晰、明了。

1.8 清除工作目录过程文件

规则除了完成源代码编译之外，也可以完成其它任务。例如：前边提到的为了实现清除当前目录中编译过程中产生的临时文件（edit和那些.o文件）的规则：

```
clean :
        rm edit $(objects)
```

在实际应用时，我们把这个规则写成如下稍微复杂一些的样子，防止出现始料未及的情况。

```
.PHONY : clean
clean :
        -rm edit $(objects)
```

这个实现有两点不同：

1. 通过 ".PHONY" 这个特殊目标将 "clean" 目标声明成伪目标。避免当磁盘上存在一个名为"clean"文件时，目标 "clean" 所在规则的命令无法执行。
2. 在命令行之前使用 "-" ，意思是忽略命令 "rm" 的执行错误。

这样的一个目标在Makefile中，不能将其作为终极目标（Makefile的第一个目标）。因为我们的初衷并不是当你在命令行输入make以后执行删除动作，而是要创建或者更新程序。在上面的例子中，就是输入make以后对目标 "edit" 进行创建或者重建。
上面中因为目标 "clean" 没有出现在终极目标 "edit" 依赖关系中（终极目标的直接依赖或者间接依赖），所以我们执行 "make" 时，目标 "clean" 所在的规则将不会被处理。当需要执行此规则时，要在make的命令行选项中明确指定这个目标（执行 **"make clean"** ）。

3 Makefile总述

3.1 Makefile的内容

在一个完整的Makefile中，包含了5个东西：**显示规则**、**隐含规则**、**变量定义**、**指示符**和**注释**。

- 显示规则：它描述了在何种情况下如何更新一个或者多个被称为目标的文件（Makefile的目标文件）。书写Makefile时需要明确的给出目标文件、目标的依赖文件列表以及更新目标文件所需要的

命令（有些规则没有命令，这样的规则只是存粹的描述了文件之间的依赖关系）。

- 隐含规则：它是make根据一类目标文件（典型的是根据文件名的后缀）而自动推导出来的规则。make根据目标文件的名，自动产生目标的依赖文件并使用默认的命令来对目标进行更新（建立一个规则）。
- 变量定义：使用一个字符或字符串代表一段文本串，当定义了一个变量以后，Makefile后续在需要使用此文本串的地方，通过引用这个变量来实现对文本串的使用。在上面的例子中，我们就定义了一个变量 "objects" 来表示一个.o文件列表。
- Makefile指示符：指示符指明在make程序读取makefile文件过程中所要执行的一个动作。其中包括：
  - 读取一个文件，读取给定文件名的文件，将其内容作为makefile文件的一部分。
  - 决定（通常是根据一个变量的值）处理或者忽略Makefile中的某些特定部分。
  - 定义一个多行变量。
- 注释：Makefile中 "#" 字符后的内容被作为是注释内容（和shell脚本一样）处理。如果此行的第一个非空字符为 "#" ,那么此行作为注释行。注释行的结尾如果存在反斜线（\），那么下一行也被作为注释行。一般在书写Makefile时推荐将注释作为一个独立的行，而不要和Makefile的有效行放在一行中书写。当在Makefile中需要使用字符 "#" 时，可以使用反斜线加 "#" `\#` 来实现（对特殊字符 "#" 的转移），其表示将 "#" 作为一字符而不是注释的开始标志。

需要注意的地方：
Makefile中第一个规则之后的所有以 `[Tab]` 字符开始的行，make程序都会将其交给 `shell` 程序去解释执行。因此，以 `[Tab]` 字符开始的注释行也会被交给shell来处理，此命令行是否需要被执行（shell执行或者忽略）是由shell程序来判决的。
另外，在使用指示符 `define` 定义一个多行的变量或者命令时，其定义体（ `define` 和 `endef` 之间的内容）会被完整的展开到Makefile中引用此变量的地方（包含定义体中的注释行）;make在引用此变量的引用和C语言中的宏类似（但是其实质并不相同，后续将会详细讨论）。对一个变量引用的地方make所作的就是将这个变量根据定义进行基于文本的展开，展开变量的过程不涉及到任何变量的具体含义和功能分析。

3.2 makefile文件的命名

默认的情况下，make会在工作目录（执行make的目录）下按照文件名顺序寻找makefile文件读取并执行，查找的文件名顺序为： `GNUmakefie` 、 `makefile` 、 `Makefile` 。
通常应该使用 `makefile` 或者 `Makefile` 作为一个makefile的文件名（我们推荐使用 `Makefile` ，首字母大写比较显著，而且一般在一个目录中和当前目录的一些重要文件（README,Changelist等）靠近，在寻找时会比较容易发现）。而 `GNUmakefile` 是不推荐使用的文件名，因为此命名方式只有 `GNU make` 才可以识别，而其它版本的make程序只会在工作目录下寻找 `makefile` 和 `Makefile` 这两个文件。
如果make程序在工作目录下无法找到以上三个中的任意一个，它将不读取任何其它文件作为解析对象。但是根据make隐含规则的特性，我们可以通过命令行指定一个目标，如果当前目录下存在符合此目标的依赖文件，那么这个命令行所指令的目标将会被创建或者更新。

当makefile文件的命名不是这三个中的任何一个时，需要通过make的 `-f` 、 `--file` 或者 `--makefile` 选项来指定make读取的makefile。格式如下所示：

> -f file
> --file=file
> --makefile=file

它指定 `file` 作为执行make时读取的makefile文件。也可通过 `-f` 、 `--file` 或者 `--makefile` 选项来指定多个需要读取的makefile文件，多个makefiel 文件将会按照指定的顺序进行链接并被make解析执行。格式如下所示：

> make -f file1 -f file2
> make --file=file1 -- file=file2
> make --makefile=file1 -- makefile=file2

当通过 `-f` 、 `--file` 或者 `--makefile` 选项指定make读取的makefile文件时，make就不会再自动查找三个标准命名的makefiel文件。

> 注释：通过命令指定目标使用make的隐含规则：
> 假设：当前目录不存在以 `GNUmakefile` 、 `makefile` 、 `Makefile` 命名的文件
>
> 1. 当前目录下存在一个源文件foo.c，我们可以使用 `make foo.o` 来使用make的隐含规则自动生成 `foo.o` 。当执行 `make foo.o` 时，我们看到其执行的命令为：
>   `cc -c foo.c`
>   之后，foo.o将会被创建或者更新。
> 2. 如果当前目录下没有foo.c文件时，就是make对.o文件目标的隐含规则中的依赖文件不存在。如果使用命令 `make foo.o` 时，将会得到如下提示：
>   `make: *** No rule to make target foo.o. Stop.`
> 3. 如果直接使用命令 `make` 时，得到的提示信息如下：
>   `make: *** No targets specified and no makefile found. Stop`

3.3 包含其它makefile文件

本节我们讨论如何在一个Makefile中包含其它的Makefile文件。Makefile中包含其它文件所需要使用的关键字是 `include` ,和C语言对头文件的包含方式一致。
`include` 指示符告诉make暂停读取当前的Makefile，而转去读取 `include` 指定的一个或者多个文件，完成以后再继续当前Makefile的读取。Makefile中指示符 `include` 书写在独立的一行，其形式如下：

> include FILENAMES...

`FILENAMES` 是shell所支持的文件名（可以使用通配符）。
指示符 `include` 所在的行可以一个或者多个空格（make程序在处理时将忽略这些空格）开始，切记不能以 `[Tab]` 字符开始（如果一行以 `[Tab]` 字符开始make程序将此行作为一个命令行来处理）。指示符 `include` 和文件名之间、多个文件之间使用空格或者 `[Tab]` 键隔开。行尾的空白字符在处理时被忽略。使用指示符包含进来的Makefile中，如果存在变量或者函数引用。它们将会在包含它们的Makefile中被展开。
来看一个例子，存在三个.mk文件a.mk、b.mk、c.mk， `$(bar)` 被扩展为 `bish bash` 。则

```
include foo *.mk $(bar)
```

等价于

```
include foo a.mk b.mk c.mk bish bash
```

之前已经提到过make程序在处理指示符 `include` 时，将暂停对当前指示符 `include` 的Makefile的读取，而转去依次读取由 `include` 指示符指定的文件列表。直到完成所有这些文件以后再回过头继续读取指示符 `include` 所在的Makefile文件。
通常指示符 `include` 用在以下场合：

1. 有多个不同的程序，由不同目录下的几个独立的Makefile来描述其重建规则。它们需要使用一组通用的变量定义或者模式规则。通用的做法是将这些共同使用的变量或者模式规则定义在一个文件中（没有具体的文件命名限制），在需要使用的Makefile中使用指示符 `include` 来包含此文件。
2. 当根据源文件自动产生依赖文件时，我们可以将自动产生的依赖关系保存在另一个文件中，主Makefile使用指示符 `include` 包含这些文件。这样的做法比直接在主Makefile中追加依赖文件的方法要明智的多，其它版本的make已经使用这种方式来处理。