

# Makefile

## Makefile

- 1.认识Makefile
- 2.增量编译
- 3.变量和函数
- 4.优化makefile
- 5.头文件依赖的自动生成

## 1.认识Makefile



# 程序的自动编译

当程序中有多个源文件时，  
先编译：

```
g++ -c main.cpp -o main.o  
g++ -c other.cpp -o other.o
```

再链接

```
g++ main.o other.o -o helloworld
```

考虑：能不能写一个脚本，自动执行所有的步骤？

对比：在VC下面，我们点一个“生成解决方案”就能  
生成整个项目了。 . . .



# Makefile

Makefile是一个描述“如何生成整个项目”的脚本文件

方法：

- ① 创建一个文件叫Makefile
- ② 输入命令，根据Makefile里面的指示，自动执行所有的步骤

**make -f Makefile** make会自动解析makefile中的内容

(或者直接 make 不带参数)



# Makefile

演示1:

helloworld:      这个是什么意思?

g++ main.cpp other.cpp -o helloworld

# Makefile的写法

Makefile就是一个普通的文本文件。。。

它由很多条“规则rule”组成，这些规则就是描述了“先干什么、后干什么”。。。

每一条规则的格式为：

target: dependencies

```
【TAB】 system command1  
【TAB】 system command2  
【TAB】 system command...
```



# Makefile的写法

```
target: dependencies
<TAB> system command1
<TAB> system command2
<TAB> system command...
```

target: 目标 ,  
dependencies: 依赖 (下节课讲)  
<TAB>: 每行命令前必须插一个TAB  
system command: 系统命令

读作：要完成目标target，必须执行命令commands...



# 演示1

helloworld: → 目标

g++ main.cpp other.cpp -o helloworld



系统命令



# 演示2

helloworld:

```
g++ -c main.cpp -o main.o  
g++ -c other.cpp -o other.o  
g++ main.o other.o -o helloworld
```



多个系统命令

作者：邵发 官网：<http://afanihao.cn>



# Makefile的写法

当存在很多规则时， 默认从第一条规则开始执行。

也可以显式地执行某条规则。



# 演示2

helloworld:

```
g++ -c main.cpp -o main.o  
g++ -c other.cpp -o other.o  
g++ main.o other.o -o helloworld
```

规则1

clean:

```
rm -rf *.o helloworld
```

可以选择删或者不删

规则2

输入make命令时，同时显式指定要执行那一条rule:

make clean

make -f Makefile clean

注: make命令默认只执行一条规则，其他规则默认不执行

# 小结

- (1) 理解Makefile的作用
- (2) 学会make命令的用法

注意：

不要忘记插一个TAB符

不能是空格键！

作者：邵发 官网: <http://afanihao.cn>



## 2. 增量编译

# 《C/C++ 学习指南》Linux篇

## 5.2 增量编译



作者：邵发 官网：<http://afanihao.cn>

# 增量编译

一个项目中有多个源文件， a.cpp b.cpp c.cpp ...  
编译：

a.cpp -> a.o

b.cpp -> b.o

c.cpp -> c.o

链接

(a.o , b.o , c.o -> helloworld )

当某个cpp更新后，只编译这个cpp文件，称为增量编译。

(增量编译，全量编译)

在VC中演示。

VC中就可以使用增量编译！

注意：生成解决方案 → 增量编译

重新生成解决方案 → 全量编译



作者：邵发 官网: <http://afanihao.cn>

# Makefile: 增量编译

Makefile里使用“依赖dependency”来实现增量编译。

```
target: dependencies  
<TAB> system command1  
<TAB> system command...
```

依赖: 是一个文件列表, 当有文件更新时, 执行这条规则。

(注: 根据文件的修改时间来判断是否更新)  
(如果某个依赖文件的时间比target的时间要新。。。)



作者: 邵发 官网: <http://afanihao.cn>

# 示例1

演示dependency的写法

helloworld: main.cpp other.cpp other.h  
g++ main.cpp other.cpp -o helloworld

表示：当main.cpp、other.cpp、other.h中有文件  
比较helloworld更新时，则执行规则。。。  
否则，不执行规则。

查看文件时间

ls -l --full-time

作者：邵发 官网: <http://afanihao.cn>



# Makefile: 增量变量

增量编译：不要一次性把所有文件都重新编译一遍，而是只编译有变化的部分。。。

作者：邵发 官网: <http://afanihao.cn>

## 示例2

**helloworld: main.o other.o**

g++ main.o other.o -o helloworld

**main.o: main.cpp other.h** 为什么没有other.cpp?或者为什么有other.h?

g++ -c main.cpp -o main.o    查看修改other.cpp是否影响main.o?

**other.o: other.cpp other.h**

g++ -c other.cpp -o other.o

**clean:**

rm \*.o helloworld

演示：修改other.cpp。。。

作者：邵发 官网: <http://afanihao.cn>

# 特例

时间比较:

`target (T1) : dependencies (T2)`

- ① 若`target`文件不存在，则`T1`为0
- ② 若`dependencies`为空，则`T2`为0

比较`T2`与`T1`:

`if ( T1==0) 执行;`

`else if (T2 > T1) 执行;`

`else "已是最新，不执行规则 "`

(注: 当依赖文件不存在, 则提示编译错误...)



# 特例

举例：

helloworld:

```
g++ main.cpp other.cpp -o helloworld
```

- (1) 如果helloworld文件不存在，则T1为0，规则被执行，生成helloworld
- (2) 如果helloworld存在，则T1>0, T2=0，规则不被执行（已是最新）



# 小结

理解增量编译的含义，以及如何在Makefile中实现增量编译。



## 3. 变量和函数

# 《C/C++ 学习指南》Linux篇

## 5.3 注释、变量与函数



作者：刀刀生  
官网：<http://cplus.tsinghua.org/>

# Makefile中的注释

以#开头的行，为注释行

```
#this is a test ...
```



# Makefile中的变量

Makefile中支持变量:

SUBDIR = src xml

SUBDIR += osapi

test:

```
echo $(SUBDIR)
```

定义了一个变量SUBDIR

- ① 用=号定义一个变量，并且赋值 (等号两侧可以加空格)
- ② 用+=追加字符串
- ③ 用\$(SUBDIR)取得到变量的值 (要加小括号)



# Makefile中的变量

特殊的变量:

`$@` 指代target

`$^` 指代dependencies依赖项列表

`$<` 指代依赖项列表的第一项

注: 这几种特殊变量经常用到



# Makefile中的函数

Makefile中有一些预定义的函数

**\$**(**函数名** **参数列表**)

函数名： Makefile内部自带的函数

参数列表：以逗号分开

另， 函数名和参数之间以空格分开

例如，

PWD=\$(shell pwd)

CXX\_SOURCE=\$(wildcard ./\*.cpp)



## 4.优化makefile

# 《C/C++ 学习指南》Linux篇

## 5.4 优化Makefile



作者：邵发 官网：<http://afanihao.cn>

### 优化Makefile

优化的方向：针对大量CPP文件，减少手工编辑操作，争取最大程度的自动化。

原则：保持增量编译的功能。



作者：邵发 官网：<http://afanihao.cn>

# 优化Makefile

## ① 使用通配符

`%.o:%.cpp`

`g++ -c $< -o $@`

`$@` 指target

`$<` 指第一个依赖项



# 优化Makefile

## ② 自动罗列 \*.o 文件

```
CXX_SOURCES=$(wildcard *.cpp)
```

```
CXX_OBJECTS=$(patsubst %.cpp, %.o, $(CXX_SOURCES))
```

patsubst函数：

源格式

目标格式

文件名列表



作者：邵发 官网: <http://afanihao.cn>

几个例子

建立一个测试目录，在测试目录下建立一个名为sub的子目录

```
$ mkdir test  
$ cd test  
$ mkdir sub
```

建立一个简单的Makefile

```
src=$(wildcard *.c ./sub/*.c)  
dir=$(notdir $(src))  
obj=$(patsubst %.c,%.o,$(dir) )  
all:  
@echo $(src)  
@echo $(dir)  
@echo $(obj)  
@echo "end"
```

输出结果

执行结果分析： 第一行输出： a.c b.c ./sub/sa.c ./sub/sb.c

wildcard把 指定目录 ./ 和 ./sub/ 下的所有后缀是c的文件全部展开。

第二行输出： a.c b.c sa.c sb.c notdir把展开的文件去除掉路径信息

第三行输出: a.o b.o sa.o sb.o

在(*patsubst*(dir) )中, *patsubst*把\$(dir)中的变量符合后缀是.c的全部替换成.o, 任何输出。

## 优化Makefile

③ 设置EXE变量, 表示输出程序的名称

EXE=helloworld

作者: 邵发 官网: <http://afanihao.cn>

# 补充

把头文件作为依赖项，附加在后面

main.o: other.h

other.o: other.h

问题：如果自动生成头文件依赖项呢？

作者：邵发 官网: <http://afanihao.cn>



## 5. 头文件依赖的自动生成

# 《C/C++ 学习 指 南》Linux篇

## 5.5 头文件依赖的自动生成



作者：邵发 官网：<http://afanihao.cn>

# 头文件依赖

如果头文件被更新，则包含了它的cpp文件应该被重新编译。

```
//////// main.cpp ///////  
#include "object.h"
```

所以，增量编译必须考虑头文件的更新。。。  
**main.o: main.cpp object.h**



# 头文件依赖的生成

在上节课中：

`main.o: other.h`

`other.o: other.h`

这种头文件的依赖如何生成呢？

(手写容易遗漏)



# 头文件依赖的生成

使用编译选项 -MMD

g++/gcc在编译 xxx.cpp时，可以提取里面包含的头文件(双引号包含的头文件)

```
#include "other.h"
```

然后，生成相应的 .d 文件

```
g++ -c -MMD main.cpp -o main.o
```

则在生成main.o的同时，生成一个main.d文件



# 优化Makefile

```
DEP_FILES = $(patsubst %.o, %.d, $(CXX_OBJECTS))
```

```
% .o: %.cpp
```

```
    g++ -c -MMD $< -o $@
```

```
-include $(DEP_FILES)
```

使用`-include`指令，将所有的.d文件包含进来



作者：邵发 官网: <http://afanihao.cn>

第一个版本的Makefile

```
##### 标准Makefile Lv1.0 #####
EXE=helloworld
SUBDIR=src object
CXX_SOURCES=$(wildcard *.cpp)
CXX_SOURCES =$(foreach dir,$(SUBDIR), $(wildcard $(dir)/*.cpp))
CXX_OBJECTS=$(patsubst %.cpp, %.o, $(CXX_SOURCES))
DEP_FILES =$(patsubst %.o, %.d, $(CXX_OBJECTS))

$(EXE): $(CXX_OBJECTS)
    g++ $(CXX_OBJECTS) -o $(EXE)
%.o: %.cpp
    g++ -c -MMD $< -o $@
-include $(DEP_FILES)
clean:
    rm -rf $(CXX_OBJECTS) $(DEP_FILES) $(EXE)
test:
    echo $(CXX_OBJECTS)
```

# 注意

正常情况下， \*.d文件被g++生成在和\*.o相同的目录

例如，有一个子目录 src

```
g++ -c -MMD src/main.cpp -o src/main.o
```

则生成的main.d也在src目录下

但有些Linux上的g++/gcc版本在生成.d文件的位置有点问题

可以使用 (在Fedora Linux上存在这个问题)

```
%o:%.cpp
```

```
g++ -c -MMD -MT $@ -MF $(patsubst %.cpp, %.d, $<)  
$< -o $@
```

