CS110 Discussion

Introduction of GCC and Makefile

Yuxuan Li

VSPLab@SIST

2026.03.13

liyx22025@shanghaitech.edu.cn

目录

1 引论

编辑器和编译器

编辑器和编译器是两个独立事物！

- 编辑器是指用于编写代码的工具，例如 VSCode，但理论上也可以用记事本！
- 编译器是指将代码转换为可执行程序的工具，例如 GCC 就是一个 C 编译器。

理解程序如何编译很重要

如果你不知道你的程序是如何运行的，那你只是在祈祷它能运行！

理解程序如何编译很重要

或许大家在初学 C 语言时，会使用某种集成环境（IDE）[1][2]。所谓集成环境，就是指将代码编辑和代码编译的功能集成在了同一个软件中。对你而言，编译的过程或许就是在某个漂亮的按钮上点一下，过了一会儿，执行程序就自己出现在那里了。

[1]请注意：VSCode 和 VS 是两个截然不同的事物！
[2]VSCode 是一个优秀的代码编辑器，VS 是一个糟糕且臃肿的集成环境。

理解程序如何编译很重要

我们坚决反对依赖这类功能！当你需要更精细的配置编译过程，或许是想添加几个编译选项，或许是想添加几个额外的链接库，事情就会变得糟糕起来：这类软件的简单性往往依赖于高度复杂且难以修改的“项目文件”，修改将变得非常困难。

因此，学会如何通过 GCC 命令直接编译自己的 C 程序，是非常重要的！

GCC 和 Makefile

在本节课中，主要会涉及两个工具

- GCC 解决如何编译 C 程序的问题。
- Makefile 解决如何将编译流程自动化的问题。

朴素的想法：记不住这么多命令！我就想用一条简单命令直接得到编译结果！

GCC 和 Makefile

GCC 和 Makefile 都可以通过下面的命令安装

```
1 sudo apt install build-essential
```

2 编译单个 C 文件

最简单的 C 程序

假设你编写了一个最简单的 main.c

```c
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

GCC 的基本用法

你可以通过下面的命令，将 main.c 编译为 MyProgram，选项 -o 指定输出文件

```
1 gcc -o MyProgram main.c
```

你可以这样运行 MyProgram

```
1 ./MyProgram
```

Makefile 引入的原因

- `MyProgram` 和 `main.c` 这两个名字可以固定下来，不用每次手动输入。
- `MyProgram` 如果已经是最新的，不要重复编译。

Makefile 的基本用法

创建一个名为 Makefile 的文件，并写入以下内容

```
1 MyProgram: main.c
2     gcc -o MyProgram main.c
```

现在这样编译

```
1 make
```

当运行 make 时，它默认会寻找当前目录下名为 Makefile 的文件作为其配置文件。

Makefile 的基本用法

Makefile 引入后，注意到以下两个问题被解决

- 若重复运行 `make`，并不会导致无意义的重复编译，而是提示“已是最新”！
- 若修改 `main.c` 后再运行 `make`，此时会重新编译以得到新的执行程序。

规则：目标、依赖、配方

Makefile 的基本结构是**规则**（Rule）

- **目标**：我最终要生成什么文件？（Target）
- **依赖**：目标的生成基于哪些文件？（Prerequisite）
- **配方**：目标是通过何种命令生成的？（Recipe）

“目标-依赖-命令”，这三者就构成了 Makefile 的一条规则

```
1 target: prerequisite
2     recipe
```

特别提醒：Makefile 中的缩进必须是 Tab 而不能是视觉上等效的四个 Space！

规则：目标、依赖、配方

Makefile 利用了文件的时间戳实现了按需构建！

- 若目标不存在，那需要进行构建。
- 若目标存在，但时间戳早于某一个依赖[3]，那也需要重新构建。

简单来说，如果目标的时间戳晚于目标的所有依赖，那目标必然就是最新的！

[3]目标上一次被构建后，依赖又被修改了！

变量

Makefile 可以引入变量，以便更高效的组织规则，例如

```
PROJECT:=MyProgram

SRCS:=main.c

PROGRAM:=${PROJECT}
```

现在你可以将规则写为

```
${PROGRAM}: ${SRCS}
    gcc -o ${PROGRAM} ${SRCS}
```

变量

Makefile 中，关于变量的定义和使用，要注意以下三点

- 变量在使用时要用 `${}` 包裹，但定义时不用！
- 变量赋值一般用 `:=`，这是最符合通常认知的赋值，不要随意使用 `=` 赋值。
- 变量命名习惯上常用全大写。

变量

Makefile 中习惯将使用的编译器和编译参数也作为变量（-Wall 显示所有警告）

```
1 CC_BIN:=gcc
2
3 CC_FLAGS:=-Wall
```

现在你可以将规则写为

```
1 ${PROJECT}: ${SRCS}
2     ${CC_BIN} ${CC_FLAGS} -o ${PROJECT} ${SRCS}
```

输出目录

编译输出和源代码位于同一目录是很糟糕的工程实践！

- 如果要清理输出，很容易错误的删除源代码。
- 如果使用了 Git 版本控制，忽略输出文件会变得特别麻烦。

推荐的做法：建立一个 `build` 目录，任何编译输出都放到 `build` 目录下。

输出目录

定义一个输出目录的变量 BUILD_DIR

```
1 BUILD_DIR:=build
```

现在 PROGRAM 应当位于 BUILD_DIR 下

```
1 PROGRAM:=${BUILD_DIR}/${PROJECT}
```

输出目录

当然，这个 build 目录本身也不用手动创建，不妨为 BUILD_DIR 也添加一条规则

```
1 ${BUILD_DIR}:
2     mkdir -p ${BUILD_DIR}
```

现在 PROGRAM 应当依赖于 BUILD_DIR 的存在

```
1 ${PROGRAM}: ${SRCS} ${BUILD_DIR}
2     ${CC_BIN} ${CC_FLAGS} -o ${PROGRAM} ${SRCS}
```

特殊变量

Makefile 中有若干特殊变量，它们可以在配方中使用，简化配方的表达

表 1: Makefile 中的特殊变量

变量	含义
$@	目标
$<	首个依赖
$^	全部依赖
$*	由 % 匹配的内容

特殊变量

应用这些特殊变量，可以将上述两条规则重写为

```
1 ${PROGRAM}: ${SRCS} ${BUILD_DIR}
2     ${CC_BIN} ${CC_FLAGS} -o $@ $<
3
4 ${BUILD_DIR}:
5     mkdir -p $@
```

伪目标

伪目标（Phony Target）的配方运行后并不会产生目标。典型的应用是 `clean`，当运行 `make clean` 后 `build` 目录会被清空，没有一个称为 `clean` 的文件会被创建。

```
1 clean:
2     rm -r -v -I ${BUILD_DIR}
```

其中，选项 `-r` 允许递归删除，选项 `-v` 显示删除内容，选项 `-I` 触发单次确认。

伪目标很适合用来做这些特定任务，因为目标不会被产生，它总是无条件的执行！

伪目标

当运行 `make target` 时会构建名称为 `target` 的目标，而如果直接运行 `make`，则默认会构建第一个目标。因此，终极目标 `PROGRAM` 的规则必须置于最前面。在实践中，为了规则书写顺序的灵活性，常用的一个技巧是额外添加一个称为 `default` 的伪目标，它的名称没有特殊性，但必须置于最前面，而 `default` 依赖于 `PROGRAM` 即可。

```
1 default: ${PROGRAM}
```

伪目标

伪目标除了 clean 之外，另外一个典型用法是 run

```
1 run: default
2     ${PROGRAM}
```

现在运行 make run 就可以运行程序了，你不必再关心程序的名字和位置了！

伪目标

伪目标的工作前提是目标永远不会被产生，如果很不幸的，你的目录中恰好有一个文件叫 `clean` 或 `run`，这会导致伪目标无法被反复执行，以下方案可以解决

```
1 .PHONY: default run clean
```

当然，总的来说，不要创建和伪目标同名的文件。

3 编译多个 C 文件

更多文件

假设在 `main.c` 之外，你又编写了 `Vec3.h` 和 `Vec3.c`，现在怎么编译?

```c
#ifndef VEC_3H
#define VEC_3H

typedef struct
{
    double x;
    double y;
    double z;
} Vec3;

Vec3 Vec3Add(const Vec3* u, const Vec3* v);
Vec3 Vec3Mul(double k, const Vec3* u);

#endif
```

```c
#include "Vec3.h"


Vec3 Vec3Add(const Vec3* u, const Vec3* v)
{
    Vec3 result = {(u->x + v->x), (u->y + v->y), (u->z + v->z)};
    return result;
}

Vec3 Vec3Mul(double k, const Vec3* u)
{
    Vec3 result = {(u->x * k), (u->y * k), (u->z * k)};
    return result;
}
```

```c
#include <stdio.h>
#include "Vec3.h"

int main()
{
    double k = 42.0;
    Vec3 u = {1.0, 1.0, 1.0};
    Vec3 v = {2.0, 4.0, 8.0};
    Vec3 sum = Vec3Add(&u, &v);
    Vec3 mul = Vec3Mul(k, &sum);
    printf("Result = (%lf, %lf, %lf)\n", mul.x, mul.y, mul.z);
    printf("Hello World!\n");
    return 0;
}
```

编译和链接

构建执行程序分为两步：编译（Compile）和链接（Link）。

当编译 `main.c` 时，它事实上并不知道 `Vec3Add()` 和 `Vec3Mul()` 的具体实现，头文件 `Vec3.h` 的作用是提供了这些函数的接口定义。在 C 语言中，每个 C 文件是可以独立被编译的，这个中间产物称为目标文件，例如 `main.c` 和 `Vec3.c` 分别会被编译为 `main.o` 和 `Vec3.o`。目标文件本质是一种二进制片段，链接的意义是将所有目标文件拼接在一起，并将相互之间的函数调用地址正确设置，得到完整的执行程序。

编译和链接

编译 main.c 和 Vec3.c，选项 -c 用于明确仅编译不链接

```
gcc -o build/main.o -c main.c
gcc -o build/Vec3.o -c Vec3.c
```

链接 main.o 和 Vec3.o

```
gcc -o build/MyProgram build/main.o build/Vec3.o
```

编译和链接

使用 `wildcard` 函数可以实现通配

```
1 SRCS:=$(wildcard *.c)
```

使用 `addprefix` 函数可以为一个空格分隔的列表添加前缀，而 `:.c=.o` 替换后缀

```
1 OBJS:=$(addprefix ${BUILD_DIR}/,${SRCS:.c=.o})
```

若 `SRCS` 是 `main.c Vec3.c`，那 `OBJS` 就是 `build/main.o build/Vec3.o` 了。

实际上，在 Makefile 中 `${}` 和 `$()` 可以混用于函数和变量，但实践中一个比较推荐的做法是：变量统一使用 `${}`，函数统一使用 `$()`，这样就可以分清变量和函数。

编译和链接

通常来说，链接选项 LD_FLAGS 是在编译选项 CC_FLAGS 的基础上继续添加

```
1 CC_FLAGS:=-Wall
2
3 LD_FLAGS:=${CC_FLAGS}
```

请不必担心，GCC 在链接时会自动滤除那些实际上不适用于链接阶段的选项。

编译和链接

规则也要分为编译和链接两步了

```
1 ${PROGRAM}: ${OBJS}
2     ${CC_BIN} ${LD_FLAGS} -o $@ $^
3
4 ${BUILD_DIR}/%.o: %.c | ${BUILD_DIR}
5     ${CC_BIN} ${CC_FLAGS} -o $@ -c $<
```

其中，% 是专门用于目标和依赖的通配符。请注意，这里不是简单的令 OBJS 依赖于 SRCS，而是让每一个 .o 文件依赖于对应的 .c 文件，用 % 的方式批量产生规则。

编译和链接

细节：为何这里要在依赖 BUILD_DIR 前添加一个额外的 | 符号？

```
1 ${BUILD_DIR}/%.o: %.c | ${BUILD_DIR}
```

在 Makefile 中 | 后的依赖被视为顺序无关依赖，它只要求存在即可而不会校验时间戳。由于现在构建分为两步，链接产生执行程序会刷新 BUILD_DIR 目录（目录中的文件变化也会更新目录的时间戳）。若不添加 | 符号，下次运行 make 时，编译步骤又会因为 BUILD_DIR 的时间戳更新重新生成 .o 文件，导致按需构建机制被破坏！

简而言之，目标的输出目录应当作为目标的顺序无关依赖。

头文件的依赖

至此，整个构建流程已经比较完整了。但还有一组依赖关系尚未处理：试想，若头文件 `Vec3.h` 变化了，那 `Vec3.o` 和 `main.o` 理应分别从 `Vec3.c` 和 `main.c` 重新编译。

更一般的说，我们需要一套可以自动管理头文件依赖的方案！

头文件的依赖

幸运的是，GCC 提供了一个选项 `-MM` 用于生成头文件依赖

```
gcc -MM -o build/main.d -c main.c
gcc -MM -o build/Vec3.d -c Vec3.c
```

对于 `main.d`，这会得到以下结果

```
main.o: main.c Vec3.h
```

有趣的是，这个自动生成的依赖文件，其实也是一个 Makefile!

头文件的依赖

然而，这个依赖文件还有两点需要更改

- 应当改用 `build/main.o` 代替 `main.o` 以适应输出目录。
- 应当添加 `build/main.d` 本身作为目标：源文件更新时，依赖文件也要更新。

最终预期的形式是

```
1 build/main.o build/main.d: main.c Vec3.h
```

头文件的依赖

类似于 OBJS，引入一个 DEPS

```
1 DEPS:=$(addprefix ${BUILD_DIR}/,${SRCS:.c=.d})
```

生成依赖文件的规则是

```
1 ${BUILD_DIR}/%.d: %.c | ${BUILD_DIR}
2     ${CC_BIN} ${CC_FLAGS} -MM -o $@ $<
3     sed -r -e 's|$*\.o|${BUILD_DIR}/$*.o $@|g' -i $@
```

头文件的依赖

这里 sed 是一个正则文本替换工具（Stream Editor），选项 -r 代表启用扩展正则语法[4]，选项 -i 后接要替换的文件，选项 -e 后接形如 s/old/new/g 的表达式。

```
1 sed -r -e 's|$*\.o|${BUILD_DIR}/$*.o $@|g' -i $@
```

- 用 | 取带 /（避免转义），sed 允许用 / 之外的符号作为分隔符。
- $*\.o 的含义是匹配 main.o，其中 \. 是因为 . 在正则匹配中有特殊含义。
- ${BUILD_DIR}/$*.o $@| 的含义是替换为 build/main.o build/main.d

[4]启用了才是标准的正则语法，不过这个例子没有到任何正则语法

头文件的依赖

最后，将依赖文件作为子 Makefile 引入，除非执行 `make clean`

```
1 ifeq ($(filter clean,${MAKECMDGOALS}),)
2 sinclude ${DEPS}
3 endif
```

总结

至此，我们就从零编写了一个适用于 C 语言的 Makefile！

谢谢大家！