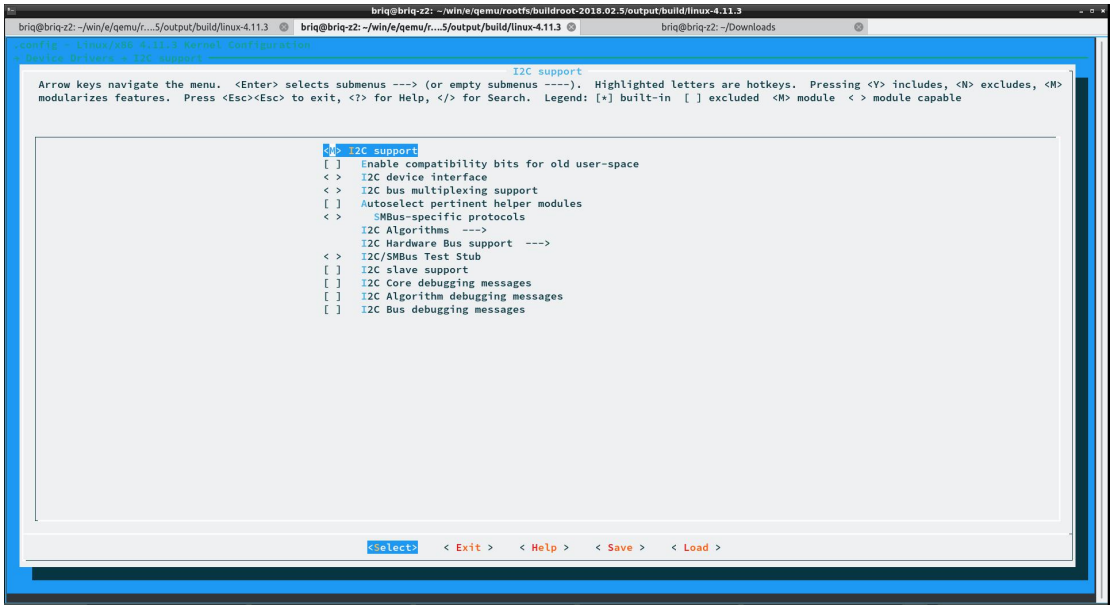
准备学习编写 linux driver 问题，但是一开始，不知道驱动是怎么编译的，所以就在 linux 源码树中，研究了下如何编译 c 文件成模块的

1、首先以最简单的 i2c 驱动为例，我使用 make menuconfig，修改 i2c 驱动编译成模块，并保存，详细如下图所示：



2、然后使用 make -w -n modules > debug_make_modules_i2c.log 捕获执行 make modules 后都干了什么， -w，表示 make 过程中如果改变了路径去调用其他 makefile 的话，会打印该路径，以便跟踪调试，-n，表示只是模拟过程，不真正的执行 makefile 中的实际编译过程。

打开 debug_make_modules_i2c.log 文件，搜索关键字 drivers/i2c，定位到 drivers/i2c 目录下的 i2c 驱动编译信息，如下所示：

```
1729 :
1730 make -f ./scripts/Makefile.modbuiltin obj=drivers/i2c
1731 make -f ./scripts/Makefile.modbuiltin obj=drivers/i2c/algos
1732 (for m in ; do echo kernel/$m; done;    \
1733 cat /dev/null ) > drivers/i2c/algos/modules.builtin
```

发现了重要信息，那就是 make -f ./scripts/Makefile.modbuiltin obj=drivers/i2c 这条

语句，原来，在源码树下执行 make modules，会调用 scripts/makefile.modbuiltin 这个 makefile，里面包含了怎么去编译 drivers/i2c 目录下的 c 文件

3、我们继续看 makefile.modbuiltin 文件，看看如何编译模块的

```
5  src := $(obj)
6
7  PHONY := __modbuiltin
8  __modbuiltin:
9
10 -include include/config/auto.conf
11 # tristate.conf sets tristate variables to uppercase 'Y' or 'M'
12 # That way, we get the list of built-in modules in obj-Y
13 -include include/config/tristate.conf
14
15 include scripts/Kbuild.include
16
17 ifneq ($(KBUILD_SRC),)
18 # Create output directory if not already present
19 _dummy := $(shell [ -d $(obj) ] || mkdir -p $(obj))
20 endif
21
22 # The filename Kbuild has precedence over Makefile
23 kbuild-dir := $(if $(filter /%,$(src)),$(src),$(srctree)/$(src))
24 kbuild-file := $(if $(wildcard $(kbuild-dir)/Kbuild),$(kbuild-dir)/Kbuild,$(kbuild-dir)/Makefile)
25 include $(kbuild-file)
```

make -f ./scripts/Makefile.modbuiltin obj=drivers/i2c，在上图中的第 5 行，src 就是 drivers/i2c 目录，再看第 24 行这条语句，执行过后，kbuild-file 就是 drivers/i2c/Makefile，

第 25 行，表示去执行 drivers/i2c/Makefile，那么这个 Makefile 文件里面是什么了？

打开 drivers/i2c/Makefile 看看都有什么内容。

该文件，全是 obj-$(CONFIG_I2C)的变量赋值，那么 CONFIG_I2C 这个变量，我们知道在 make menuconfig 中我们定义的是 M，那么 CONFIG_I2C=M,扩展出来就是：

obj-M = i2c-core.o 等一系列的定义

```
4
5  obj-$(CONFIG_I2C_BOARDINFO)     += i2c-boardinfo.o
6  obj-$(CONFIG_I2C)               += i2c-core.o
7  obj-$(CONFIG_I2C_SMBUS)         += i2c-smbus.o
8  obj-$(CONFIG_I2C_CHARDEV)       += i2c-dev.o
9  obj-$(CONFIG_I2C_MUX)           += i2c-mux.o
10 obj-y                           += algos/ busses/ muxes/
11 obj-$(CONFIG_I2C_STUB)          += i2c-stub.o
12 obj-$(CONFIG_I2C_SLAVE_EEPROM)  += i2c-slave-eeprom.o
13
14 ccflags-$(CONFIG_I2C_DEBUG_CORE) := -DDEBUG
15 CFLAGS_i2c-core.o := -Wno-deprecated-declarations
~
~
```

再回到 makefile.modbuiltin 文件中 ,在该文件中 ,就会使用 obj-M 变量 ,完成 i2c-core.c 文件的编译，编译成模块。

但是我们在外部编译驱动，使用的是 makefile.modpost，而不是 makefile.modbuiltin 脚本。

4、以下摘抄网络，描述了编译驱动的 makefile 执行流程，

linux 设备驱动 makefile 入门解析

2013 年 03 月 01 日 15:40:31 18553514996 阅读数：14824 更多

个人分类： Linux

以下内容仅作参考，能力有限，如有错误还请纠正。

对于一个普通的 linux 设备驱动模块，以下是一个经典的 *makefile* 代码，使用下面这个 makefile 可以完成大部分驱动的编译，使用时只需要修改一下要编译生成的驱动名称即可。只需修改 obj-m 的值。

```
ifneq ($(KERNELRELEASE),)
obj-m:=hello.o
else
#generate the path
CURRENT_PATH:=$(shell pwd)
#the absolute path
```

```
LINUX_KERNEL_PATH:=/lib/modules/$(shell uname -r)/build
#complie object
default:
make -C $(LINUX_KERNEL_PATH) M=$(CURRENT_PATH) modules
clean:
make -C $(LINUX_KERNEL_PATH) M=$(CURRENT_PATH) clean
endif
```

说明：

当我们在模块的源代码目录下运行 make 时，make 是怎么执行的呢？

假设模块的源代码目录是/home/study/prog/mod/hello/下。

先说明以下 *makefile* 中一些变量意义：

（1）KERNELRELEASE 在 linux 内核源代码中的顶层 makefile 中有定义

（2）shell pwd 会取得当前工作路径

（3）shell uname -r 会取得当前内核的版本号

（4）LINUX_KERNEL_PATH 变量便是当前内核的源代码目录。

关于 linux 源码的目录有两个，分别为"/lib/modules/$(shell uname -r)/build"和"/usr/src/linux-header-$(shell uname -r)/"，

但如果编译过内核就会知道 ,usr 目录下那个源代码一般是我们自己下载后解压的 ,而 lib 目录下的则是在编译时自动 copy 过去的，

两者的文件结构完全一样，因此有时也将内核源码目录设置成/usr/src/linux-header-$(shell uname -r)/。关于内核源码目录可以根据自己的存放位置进行修改。

（5）make -C $(LINUX_KERNEL_PATH) M=$(CURRENT_PATH) modules

这就是编译模块了：首先改变目录到-C 选项指定的位置（即内核源代码目录），其中保存有内核的顶层 makefile；

M=选项让该 makefile 在构造 modules 目标之前返回到模块源代码目录；然后，modueles 目标指向 obj-m 变量中设定的模块；

在上面的例子中，我们将该变量设置成了 hello.o。

按照顺序分析以下 make 的执行步骤：

在模块的源代码目录下执行 make，此时，宏“KERNELRELEASE”没有定义，因此进入 else。

由于 make 后面没有目标，所以 make 会在 Makefile 中的第一个不是以.开头的目标作为默认的目标执行。

于是 default 成为 make 的目标。

make 会执行 $(MAKE) -C $(KERNELDIR) M=$(PWD) modules ,假设当前内核版本是 2.6.13-study,

所以$(shell uname -r)的结果是 2.6.13-study ，这里实际运行的是

make -C /lib/modules/2.6.13-study/build M=/home/study/prog/mod/hello/ modules

-C 表示到存放内核的目录执行其 makefile，在执行过程中会定义 KERNELRELEASE，

然后 M=$(CURDIR)表示返回到当前目录，再次执行 makefile，modules 表示编译成模块的意思。

而此时 KERNELRELEASE 已定义，则会执行 obj-m += hello.o，表示会将 hello_world.o 目标编译成.ko 模块。

若有多个源文件，则采用如下方法：

obj-m := hello.o

hello-objs := file1.o file2.o file3.o

关于 make modules 的更详细的过程可以在内核源码目录下的 scripts/Makefile.modpost 文件的注释 中找到。

如果把 hello 模块移动到内核源代码中。例如放到/usr/src/linux/driver/中， KERNELRELEASE 就有定义了。

在/usr/src/linux/Makefile 中有

KERNELRELEASE=$(VERSION).$(PATCHLEVEL).$(SUBLEVEL)$(EXTRAVERSION)$(LOCALVERSION)。

这时候，hello 模块也不再是单独用 make 编译，而是在内核中用 make modules 进行编译，此时驱动模块便和内核编译在一起。

关于内核模块编译可以参考另外一篇文章：linux 内核模块编译基础知识。