实验 1：openEuler 下开发调试工具

1. 实验目的

（1）获取 GCC for openEuler

（2）掌握熟悉命令行程序调试工具 GBD。

（3）掌握多个目录下，多个源码文件的多 Makefile 书写以及分析 linux 0.11 中 kernel 部分的 Makefile 文件 。

（4）掌握 vim 的各种模式，包括：命令模式(Command mode)，插入模式(Insert mode)和底线命令模式(Last line mode)。

（5）掌握如何提交一个好的 PR，学会使用 git。

2. 实验内容

（1）认真阅读本章节准备知识 GCC for openEuler 部分的资料，根据准备知识自行登录鲲鹏社区下载 GCC，并根据准备知识进行编译测试，完成课后任务。

（2）认真阅读本章节准备知识 GBD 部分的资料，根据准备知识完成课后任务。

（3）认真阅读本章节准备知识 makefile 部分的资料，根据准备知识完成课后任务。

（4）认真阅读本章节准备知识 Vim 部分的资料，根据准备知识完成课后任务。

（5）认真阅读本章节准备知识 Git 部分的资料，根据准备知识完成课后任务。

3. 实验一： GCC 的简单使用

3.1 获取 GCC for openEuler

获取 GCC for openEuler 有两种方式，第一种是用 yum 直接安装，另外一种可以登录鲲鹏社区，访问 GCC for openEuler 页面

（https://www.hikunpeng.com/zh/developer/devkit/download/gcc）的“软件

包下载”中获取当前版本最新的软件包，或者在命令行中使用如下操作下载（注意：下面链接的 gcc 文件非最新版，可根据最新版本来替换相应下载文件）：

步骤 1 执行下载命令

```
$wget https://mirrors.huaweicloud.com/kunpeng/archive/compiler/kunpeng_gcc/gcc-10.3.1-2024.06-aarch64-linux.tar.gz
```

步骤 2 安装包验证：GCC for openEuler 编译器提供 sha256sum 文件用于软件包的完整性校验，用户可使用以下命令生成哈希值对比确认：

```
$wget https://mirrors.huaweicloud.com/kunpeng/archive/compiler/kunpeng_gcc/gcc-10.3.1-2024.06-aarch64-linux.tar.gz.sha256
$ sha256sum gcc-10.3.1-2024.06-aarch64-linux.tar.gz
```

步骤 3 指定安装目录：以/opt/aarch64/compiler 为例：

```
$ mkdir -p /opt/aarch64/compiler
$ cp -rf gcc-10.3.1-2024.06-aarch64-linux.tar.gz /opt/aarch64/compiler
```

步骤 4 进入安装目录，进行解压缩，查看运行结果。可以通过配置环境变量的方式使用 GCC for openEuler。

```
$ cd /opt/aarch64/compiler
$ tar –xf gcc-10.3.1-2024.06-aarch64-linux.tar.gz
```

步骤 5 编辑 profile 文件（使用 vim 打开）：编辑/etc/profile 文件，在/etc/profile 文件末尾添加如下内容：

```
$export PATH=/opt/aarch64/compiler/gcc-10.3.1-2024.06-aarch64-linux/bin:$PATH
$export INCLUDE=/opt/aarch64/compiler/gcc-10.3.1-2024.06-aarch64-linux/include:$INCLUDE
$export LD_LIBRARY_PATH=/opt/aarch64/compiler/gcc-10.3.1-2024.06-aarch64-linux/lib64:$LD_LIBRARY_PATH
```

注意，如果您的安装目录不同，请注意修改为实际目录，且需遵循安装路径在环境变量前的规则。

步骤 6 使环境变量生效

```
$ source /etc/profile
```

注意，如果您的安装目录不同，请注意修改为实际目录，且需遵循安装路径在环境变量前的规则。

步骤 7 查看程序执行结果是否正确

```
$ gcc -v
```

如果返回结果包含 GCC for openEuler 的 version 信息，则 GCC for openEuler 编译器已安装成功。GCC for openEuler 编译器基于开源 GCC，其命令 gcc、g++、gfortran 等的使用方式与 GCC 相同。

3.2 课堂任务

课堂任务 1：编译多个文件。准备如下代码：

```
// hello.c
#include <stdio.h>
#include "hello.h"
void printHello()
{
```

```
    printf("hello world!\n");
}
// main.c
#include <stdio.h>
#include "hello.h"
int main()
{
 printHello();
 return 0;
}
// hello.h
// 仅包含函数声明
#ifndef _HELLO_
#define _HELLO_
void printHello();
#endif
```

要求：编译这三个文件。

4. 实验二：GDB 的简单使用

4.1 GBD 准备知识

GDB 是一个由 GNU 开源组织发布的、UNIX/LINUX 操作系统下的、基于命令行的、功能强大的程序调试工具。GDB 中的命令固然很多，但我们只需掌握其中十个左右的命令，就大致可以完成日常的基本的程序调试工作。

命令	解释	示例
file <文件名>	加载被调试的可执行程序文件。 因为一般都在被调试程序所在目录下执行GDB，因而文本名不需要带路径。	(gdb) file gdb-sample
r	Run的简写，运行被调试的程序。 如果此前没有下过断点，则执行完整个程	(gdb) r

	序；如果有断点，则程序暂停在第一个可用断点处。	
c	Continue的简写，继续执行被调试程序，直至下一个断点或程序结束。	(gdb) c
b <行号> b <函数名称> b *<函数名称> b *<代码地址> d [编号]	b: Breakpoint的简写，设置断点。两可以使用“行号”“函数名称”“执行地址”等方式指定断点位置。 其中在函数名称前面加“*”符号表示将断点设置在“由编译器生成的prolog代码处”。如果不了解汇编，可以不予理会此用法。 d: Delete breakpoint的简写，删除指定编号的某个断点，或删除所有断点。断点编号从1开始递增。	(gdb) b 8 (gdb) b main (gdb) b *main (gdb) b *0x804835c (gdb) d
s, n	s: 执行一行源程序代码，如果此行代码中有函数调用，则进入该函数； n: 执行一行源程序代码，此行代码中的函数调用也一并执行。 s 相当于其它调试器中的“Step Into (单步跟踪进入)”； n 相当于其它调试器中的“Step Over (单步跟踪)”。	(gdb) s (gdb) n

	这两个命令必须在有源代码调试信息的情况下才可以使用（GCC编译时使用“-g”参数）。	
si, ni	si命令类似于s命令，ni命令类似于n命令。 所不同的是，这两个命令（si/ni）所针对的是汇编指令，而s/n针对的是源代码。	(gdb) si (gdb) ni
p <变量名称>	Print的简写，显示指定变量（临时变量或全局变量）的值。	(gdb) p i (gdb) p nGlobalVar
display ... undisplay <编号>	display，设置程序中断后欲显示的数据及其格式。 例如，如果希望每次程序中断后可以看到即将被执行的下一条汇编指令，可以使用命令“display /i $pc” 其中 $pc 代表当前汇编指令，/i 表示以十六进行显示。当需要关心汇编代码时，此命令相当有用。 undispaly，取消先前的display设置，编号从1开始递增。	(gdb) display /i $pc (gdb) undisplay 1
i	Info的简写，用于显示各类信息，详情请查阅“help i”。	(gdb) i r
q	Quit的简写，退出GDB调试环境。	(gdb) q
help [命令名称]	GDB帮助命令，提供对GDB名种命令的解	(gdb) help display

	释说明。如果指定了“命令名称”参数，则显示该命令的详细说明；如果没有指定参数，则分类显示所有GDB命令，供用户进一步浏览和查询。	

这里先给出一个 C 语言代码的示例小程序。

```
1. #include <stdio.h>
2. int nGlobalVar = 0;
3.
4. int tempFunction(int a, int b)
5. {
6.     printf("tempFunction is called, a = %d, b = %d \n", a, b);
7.     return (a + b);
8. }
9.
10. int main()
11. {
12.     int n;
13.     n = 1;
14.     n++;
15.     n--;
16.
17.     nGlobalVar += 100;
18.     nGlobalVar -= 12;
19.
20.     printf("n = %d, nGlobalVar = %d\n", n, nGlobalVar);
21.
22.     n = tempFunction(1, 2);
23.     printf("n = %d", n);
24.
25.     return 0;
26. }
```

请将此代码复制出来并保存到文件 gdb-sample.c 中，然后切换到此文件所在目录，用 GCC 编译之：

```
1. gcc gdb-sample.c -o gdb-sample -g
```

在上面的命令行中，使用 -o 参数指定了编译生成的可执行文件名为 gdb-sample，使用参数 -g 表示将源代码信息编译到可执行文件中。如果不使用参数 -g，会给后面的 GDB 调试造成不便。当然，如果我们没有程序的源代码，自然也无从使用 -g 参数，调试/跟踪时也只能是汇编代码级别的调试/跟踪。

GDB 下载：使用包管理下载：

```
$ sudo yum install gdb
```

下面“gdb”命令启动 GDB，将首先显示 GDB 说明，不管它：

```
$  gdb
```

接下来进入 GDB 交互界面，下面使用“file”命令载入被调试程序 gdb-sample（这里的 gdb-sample 即前面 GCC 编译输出的可执行文件）：

```
1. (gdb) file gdb-sample
2. Reading symbols from gdb-sample...done.
```

上面最后一行提示已经加载成功。下面使用“r”命令执行（Run）被调试文件，因为尚未设置任何断点，将直接执行到程序结束：

```
1. (gdb) r
2. n = 1, nGlobalVar = 88
3. tempFunction is called, a = 1, b = 2
```

```
4. n = 3
5. program exited normally.
```

下面使用“b”命令在 main 函数开头设置一个断点（Breakpoint）：

```
1. (gdb) b main
2. Breakpoint 1 at 0x804835c: file gdb-sample.c, line 19.
```

上面最后一行提示已经成功设置断点，并给出了该断点信息：在源文件 gdb-sample.c 第 19 行处设置断点；这是本程序的第一个断点（序号为 1）；断点处的代码地址为 0x804835c（此值可能仅在本次调试过程中有效）。回过头去看源代码，第 19 行中的代码为“n = 1”，恰好是 main 函数中的第一个可执行语句（前面的“int n;”为变量定义语句，并非可执行语句）。

再次使用“r”命令执行（Run）被调试程序：

```
1. (gdb) r
2. Breakpoint 1, main () at gdb-sample.c:19
3. 19 n = 1;
```

程序中断在 gdb-sample.c 第 19 行处，即 main 函数是第一个可执行语句处。

上面最后一行信息为：下一条将要执行的源代码为“n = 1;”，它是源代码文件 gdb-sample.c 中的第 19 行。

下面使用“s”命令（Step）执行下一行代码（即第 19 行“n = 1;”）：

```
1. (gdb) s
2. 20 n++;
```

上面的信息表示已经执行完“n = 1;”,并显示下一条要执行的代码为第 20 行的“n++;”。

既然已经执行了“n = 1;”，即给变量 n 赋值为 1，那我们用“p”命令（Print）看一

下变量 n 的值是不是 1 ：

```
(gdb) p n
$1 = 1
```

果然是 1。($1 大致是表示这是第一次使用“p”命令——再次执行“p n”将显示“$2 = 1”——此信息应该没有什么用处。）

下面我们分别在第 26 行、tempFunction 函数开头各设置一个断点 (分别使用命令“b 26”“b tempFunction”）：

```
(gdb) b 26
Breakpoint 2 at 0x804837b: file gdb-sample.c, line 26.
(gdb) b tempFunction
Breakpoint 3 at 0x804832e: file gdb-sample.c, line 12.
```

使用“c”命令继续 (Continue) 执行被调试程序，程序将中断在第二个断点 (26 行)，此时全局变量 nGlobalVar 的值应该是 88；再一次执行“c”命令，程序将中断于第三个断点（12 行，tempFunction 函数开头处），此时 tempFunction 函数的两个参数 a、b 的值应分别是 1 和 2：

```
(gdb) c
Continuing.
Breakpoint 2, main () at gdb-sample.c:26
26 printf("n = %d, nGlobalVar = %d /n", n, nGlobalVar);
(gdb) p nGlobalVar
$2 = 88
(gdb) c
```

```
Continuing.
n = 1, nGlobalVar = 88
Breakpoint 3, tempFunction (a=1, b=2) at gdb-sample.c:12
12 printf("tempFunction is called, a = %d, b = %d /n", a, b);
(gdb) p a
$3 = 1
(gdb) p b
$4 = 2
```

上面反馈的信息一切都在我们预料之中，哈哈~~~再一次执行“c”命令（Continue），因为后面再也没有其它断点，程序将一直执行到结束：

```
(gdb) c
Continuing.
tempFunction is called, a = 1, b = 2
n = 3
Program exited normally.
```

有时候需要看到编译器生成的汇编代码，以进行汇编级的调试或跟踪，又该如何操作呢？

这就要用到 display 命令“display /i $pc”了（此命令前面已有详细解释）：

```
(gdb) display /i $pc
(gdb)
```

此后程序再中断时，就可以显示出汇编代码了：

```
 (gdb) r
Breakpoint 1, main () at gdb-sample.c:19
```

```
3. 19 n = 1;
4. 1: x/i $pc 0x804835c <main+16>: movl $0x1,0xfffffffc(%ebp)
```

看到了汇编代码，“n = 1;”对应的汇编代码是“movl $0x1,0xfffffffc(%ebp)”。

并且以后程序每次中断都将显示下一条汇编指定（“si”命令用于执行一条汇编代码——区别于“s”执行一行 C 代码）：

```
1. (gdb) si
2. 20 n++;
3. 1: x/i $pc 0x8048363 <main+23>: lea 0xfffffffc(%ebp),%eax
4. (gdb) si
5. 0x08048366 20 n++;
6. 1: x/i $pc 0x8048366 <main+26>: incl (%eax)
7. ……
```

接下来我们试一下命令“b *<函数名称>”。为了更简明，有必要先删除目前所有断点（使用“d”命令——Delete breakpoint）：

```
1. (gdb) d
2. Delete all breakpoints? (y or n) y
3. (gdb)
```

当被询问是否删除所有断点时，输入“y”并按回车键即可。

下面使用命令“b *main”在 main 函数的 prolog 代码处设置断点（prolog、epilog，分别表示编译器在每个函数的开头和结尾自行插入的代码）：

```
1. (gdb) b *main
2. Breakpoint 4 at 0x804834c: file gdb-sample.c, line 17.
```

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Breakpoint 4, main () at gdb-sample.c:17
17 {
1: x/i $pc 0x804834c <main>: push %ebp
(gdb) si
0x0804834d 17 {
1: x/i $pc 0x804834d <main+1>: mov %esp,%ebp
.....
```

此时可以使用“i r”命令显示寄存器中的当前值————“i r”即“Infomation Register”：

```
(gdb) i r
eax 0xbffff6a4 -1073744220
ecx 0x42015554 1107383636
.....
```

当然也可以显示任意一个指定的寄存器值：

```
(gdb) i r eax
eax 0xbffff6a4 -1073744220
```

最后一个要介绍的命令是“q”，退出（Quit）GDB 调试环境：

```
(gdb) q
The program is running. Exit anyway? (y or n) y
```

4.2 课堂任务

课堂任务 2：下面是一段有漏洞的程序，其作用是测试函数 sort,该函数的功能是通过冒泡排序算法对一个类型为 item 的结构数组进行排序,使用 GDB 调试这个程序，修补程序漏洞，实现该函数功能。

```
typedef struct
{
 char *data;
 int key;
} item;

item array[] = {
 {“bill”, 3},
 {“neil”, 4},
 {“john”, 2},
 {“rick”, 5},
 {“neil”, 4},
 {“alex”, 1},
}
void sort(item * a, int n)
{
 int i = 0, j = 0;
 int s = 1;
 for (; i < n && s != 0; i++)
 {
  s = 0;
  for (j = 0; j < n; j++)
  {
   if (a[j].key > a[j + 1].key)
   {
    item t = a[j];
    a[j] = a[j + 1];
    a[j + 1] = t;
    s++;
   }
  }
  n--;
 }
}

int main()
```

```
37.    {
38.     sort(array, 5);
39.    }
```

要求：

① 程序中没有输出，手动添加输出，查看排序结果。

② 详细分析此程序的漏洞以及解决方案。

③ 给出实验截图，修改前，修改后以及调试过程的截图。

5. 实验三： Makefile 的编写

5.1 Makefile 的一些准备知识

① Makefile 的编写规则

(1) 基本概念

Makefile 是按照某种脚本语法编写的文本文件，而 GNU make 能够对 Makefile 中指令进行解释并执行编译操作。Makefile 文件定义了一系列的规则来指定哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至于进行更复杂的功能操作。所要完成的 Makefile 文件描述了整个工程的编译、连接等规则。其中包括：工程中的哪些源文件需要编译以及如何编译、需要创建哪些库文件以及如何创建这些库文件、如何最后产生我们想要的可执行文件。尽管看起来可能是很复杂的事情，但是为工程编写 Makefile 的好处是能够使用一行命令来完成“自动化编译”，一旦提供一个 (通常对于一个工程来说会是多个) 正确的 Makefile。编译整个工程你所要做的唯一的一件事就是在 shell 提示符下输入 make 命令。整个工程完全自动编译，极大提高了效率。

make 命令执行时，需要一个 Makefile 文件，以告诉 make 命令需要怎么样的去编译和链接程序。我们要写一个 Makefile 来告诉 make 命令如何编译和链接文件。我们的规则

是：

（1）如果这个工程没有编译过，那么我们的所有 C 文件都要编译并被链接。

（2）如果这个工程的某几个 C 文件被修改，那么我们只编译被修改的 C 文件，并链接目标程序。

（3）如果这个工程的头文件被改变了，那么我们需要编译引用了这几个头文件的 C 文件，并链接目标程序。

只要我们的 Makefile 写得够好，所有的这一切，我们只用一个 make 命令就可以完成，make 命令会自动智能地根据当前的文件修改的情况来确定哪些文件需要重编译，从而自己编译所需要的文件和链接目标程序。

GNU make 工作时的执行步骤如下：

（1）读入所有的 Makefile。

（2）读入被 include 的其它 Makefile。

（3）初始化文件中的变量。

（4）推导隐含规则（implicit rules），并分析所有规则。

（5）为所有的目标文件创建依赖关系链。

（6）根据依赖关系，决定哪些目标要重新生成。

（7）执行生成命令。

1-5 步为第一个阶段，6-7 为第二个阶段。第一个阶段中，如果定义的变量被使用了，那么，make 会把其展开在使用的位置。但 make 并不会完全马上展开，make 使用的是拖延战术，如果变量出现在依赖关系的规则中，那么仅当这条依赖被决定要使用了，变量才会在其内部展开。

(2) Makefile 的相关基本问题

下面对 Makefile 的相关问题进行简单介绍：

（1）Makefile 文件由一组依赖关系和规则构成。

（2）Makefile 的依赖关系：

依赖关系定义了最终应用程序里的每个文件与源文件之间的关系。每个依赖关系由一个目标和一组该目标所依赖的源文件组成。规则则描述了如何通过这些依赖关系创建目标。依赖关系的写法如下：先写目标的名称，后面紧跟着一个冒号，接着是空格或制表符，最后是用空格或制表符隔开的文件列表，如：

myapp：main.o my.o

main.o: main.c a.h

my.o: my.c b.h

它表示目标 myapp 依赖于 main.o 和 my.o，而 main.o 依赖于 main.c 和 a.h，等等。这组依赖关系形成一个层次结构，它显示了源文件之间的关系，如果 a.h 发生了变化，就需要重新编译 main.o。

Makefile 的规则：

target ... : prerequisites ...

command

Target 也就是一个目标文件，可以是 Object File，也可以是执行文件。prerequisites 就是，要生成那个 target 所需要的文件或是目标。 command 也就是 make 需要执行的命令。(任意的 Shell 命令) command 一定要以 Tab 键开始，否者编译器无法识别 command。

这是一个文件的依赖关系，也就是说，target 这一个或多个的目标文件依赖于 prerequisites 中的文件，其生成规则定义在 command 中。通俗一点就是说，prerequisites

中如果有一个以上的文件比 target 文件要新的话，command 所定义的命令就会被执行。这就是 Makefile 的规则。也就是 Makefile 中最核心的内容。

举一个例子，下面是一个简单的 Makefile 文件：

```
1.    myapp: main.o my.o
2.         gcc –o myapp main.o my.o
3.    main.o: main.c a.h
4.         gcc –c main.c
5.    my.o : my.c b.h
6.         gcc –c my.c
7.    clean:
8.         rm myapp main.o my.o
```

在这个 makefile 中，目标文件（target）包含：执行文件 myapp 和中间目标文件（*.o），依赖文件（prerequisites）就是冒号后面的那些 .c 文件和 .h 文件。每一个 .o 文件都有一组依赖文件，而这些 .o 文件又是执行文件 myapp 的依赖文件。依赖关系的实质上就是说明了目标文件是由哪些文件生成的，换言之，目标文件是哪些文件更新的。

在定义好依赖关系后，后续的那一行定义了如何生成目标文件的操作系统命令，一定要以一个 Tab 键作为开头。记住，make 并不管命令是怎么工作的，他只管执行所定义的命令。make 会比较 targets 文件和 prerequisites 文件的修改日期，如果 prerequisites 文件的日期要比 targets 文件的日期要新，或者 target 不存在的话，那么，make 就会执行后续定义的命令。

这里要说明一点的是，clean 不是一个文件，它只不过是一个动作名字，有点像 C 语言中的 label 一样，其冒号后什么也没有，那么，make 就不会自动去找文件的依赖性，也就不会自动执行其后所定义的命令。要执行其后的命令，就要在 make 命令后明显得指出这个 label 的名字。这样的方法非常有用，我们可以在一个 makefile 中定义不用的编译或是和编译无关的命令，比如程序的打包，程序的备份，等等。

(3) make 是如何工作的

在默认的方式下，也就是我们只输入 make 命令。那么，

（1）make 会在当前目录下找名字叫“Makefile”或“makefile”的文件。

（2）如果找到，它会找文件中的第一个目标文件（target），在上面的例子中，他会找到“myapp”这个文件，并把这个文件作为最终的目标文件。

（3）如果 myapp 文件不存在，或是 myapp 所依赖的后面的 .o 文件的文件修改时间要比 myapp 这个文件新，那么，他就会执行后面所定义的命令来生成 myapp 这个文件。

（4）如果 myapp 所依赖的.o 文件也存在，那么 make 会在当前文件中找目标为.o 文件的依赖性，如果找到则再根据那一个规则生成.o 文件。（这有点像一个堆栈的过程）

（5）当然，C 文件和 H 文件是需要存在的，于是 make 会生成 .o 文件，然后再用 .o 文件生成 make 的终极任务，也就是执行文件 myapp 了。

(4) GNU make 的主要预定义变量

GNU make 有许多预定义的变量，这些变量具有特殊的含义，可在规则中使用。以下给出了一些主要的预定义变量，除这些变量外，GNU make 还将所有的环境变量作为自己的预定义变量。

$@ ——表示规则中的目标文件集。在模式规则中，如果有多个目标，那么，"$@"就是匹配于目标中模式定义的集合。

$% ——仅当目标是函数库文件中，表示规则中的目标成员名。例如，如果一个目标是"foo.a(bar.o)"，那么，"$%"就是"bar.o"，"$@"就是"foo.a"。如果目标不是函数库文件（Unix 下是[.a]，Windows 下是[.lib]），那么，其值为空。

$< ——依赖目标中的第一个目标名字。如果依赖目标是以模式（即"%"）定义的，那么"$<"将是符合模式的一系列的文件集。注意，其是一个一个取出来的。

$?——所有比目标新的依赖目标的集合。以空格分隔。

$^ ——所有的依赖目标的集合。以空格分隔。如果在依赖目标中有多个重复的，那个这个变量会去除重复的依赖目标，只保留一份。

$+ ——这个变量很像"$^"，也是所有依赖目标的集合。只是它不去除重复的依赖目标。

(5) GNU make 的命令变量

命令变量	含义
AR	函数库打包程序。默认命令是“ar”。
AS	汇编语言编译程序。默认命令是“as”。
CC	C 语言编译程序。默认命令是“cc”。
CXX	C++语言编译程序。默认命令是“g++”。
CPP	C 程序的预处理器（输出是标准输出设备）。默认命令是“$(CC) –E”。
LEX	Lex 方法分析器程序（针对于 C 或 Ratfor）。默认命令是“lex”。
YACC	Yacc 文法分析器（针对于 C 程序）。默认命令是“yacc”。
YACCR	Yacc 文法分析器（针对于 Ratfor 程序）。默认命令是“yacc –r”。
RM	删除文件命令。默认命令是“rm –f”。

(6) GNU make 的命令参数变量

下面的这些变量都是相关上面的命令的参数。如果没有指明其默认值，那么其默认值都是空。

命令参数变量	含义
ARFLAGS	函数库打包程序 AR 命令的参数。默认值是“rv”。
ASFLAGS	汇编语言编译器参数。（当明显地调用“.s”或“.S”文件时）。
CFLAGS	C 语言编译器参数。
CXXFLAGS	C++语言编译器参数。
CPPFLAGS	C 预处理器参数。（C 和 Fortran 编译器也会用到）。
LDFLAGS	链接器参数。（如：“ld”）
LFLAGS	Lex 文法分析器参数。
YFLAGS	Yacc 文法分析器参数。

② Makefile 文件的优化

Makefile 里主要包含了五个东西：显式规则、隐含规则、变量定义、文件指示和注释。

1）显式规则。显式规则说明了如何生成一个或多的的目标文件。这是由 Makefile 的书写者明显指出，要生成的文件，文件的依赖文件，生成的命令。

2）隐含规则。由于我们的 make 有自动推导的功能，所以隐含的规则可以让我们比较粗糙地简略地书写 Makefile，这是由 make 所支持的。

3）变量的定义。在 Makefile 中我们要定义一系列的变量，变量一般都是字符串，这个有点类似 C 语言中的宏，当 Makefile 被执行时，其中的变量都会被扩展到相应的引用位置上。

4）文件指示。其包括了三个部分，一个是在一个 Makefile 中引用另一个 Makefile，就像 C 语言中的 include 一样；另一个是指根据某些情况指定 Makefile 中的有效部分，就像 C 语言中的预编译#if 一样；还有就是定义一个多行的命令。

5）注释。Makefile 中只有行注释，和 UNIX 的 Shell 脚本一样，其注释用“#”字符，这个就像 C/C++中的“//”一样。如果你要在你的 Makefile 中使用“#”字符，可以用反斜框进行转义，如：“\#”。

Makefile 的优化：在前面已经接触了 Makefile 的书写，但是这样写出的 Makefile 文件会很冗长，而且不易维护和修改，可以使用宏和内部规则来大大简化 Makefile 的书写。

(1) Makefile 中宏的使用

宏就是代表字符串的短名，但是它和字符串有很大的不同。如果文件或者程序中遇到字符串常量或变量时，就会将它当成一种特定类型的常量或变量。如果遇到宏时，就在宏名出现的地方用它代表的字符串来进行代替，将这个字符串作为文件有机的组成部分。这样，当定义了宏之后，如果需要修改宏，只需在宏定义的地方进行修改，而不需在程序中每次出现宏名的地方进行修改。

要在 make 中使用宏，首先要定义宏，然后在 Makefile 中引用。之所以这种顺序，是因为 make 处理 Makefile 时类似于解释执行，如果没有预编译的过程，就不能识别那些没有定义的宏名。宏定义的格式如下：

```
1．宏名 赋值符号 宏的值
```

其中宏名由程序员自己制定，可以使字母数字以及下划线的任意组合。但不能是数字开头，一般习惯用大写，而且为了便于使用和维护，最好使名字具有一定的实际意义。赋值符号可以取以下三种：

= 将左边的变量定义为递归式扩展变量，将后面的字符串，赋值给宏， 如果右边包括

宏变量，则需要递归地处理。

:= 将左边的变量定义为，简单扩展变量，后面的字符串常量赋值给前面的宏，如果右边包含变量，则只对当前这条语句之前定义的来扩展。

+= 使宏原来的值加上一个空格，再加上后面的字符串，作为新的宏值。

?= 表示变量的取值还未定。

!= 右边是一段 shell，把 shell 的输出作为赋值。

注意：赋值符号前面除了空格之外，不能有制表符或其他分隔符，否则 make 会把它作为宏名的一部分。

宏的引用格式有两种，即使用括号，或者大括号，形如 $(宏名) 或者 ${宏名}

在前面的 Makefile 例子中，存在的问题是，它假设编译器的名字是 gcc，而在其他 Linux 系统中，编译器的名字是 cc 或者 c89，如果想将 Makefile 文件移植到另一个版本的 Linux 中，或者在现有系统中使用另一个编译器，为了使其工作，你就不得不修改 Makefile 文件中许多行的内容。为此，针对上例，利用宏对 Makefile 进行优化后结果如下：

```
2. objects= main.o my.o
3. CC=gcc
4. myapp: $(objects)
5.  $(CC) -o myapp $(objects)
6. main.o: main.c a.h
7.  $(CC) -c main.c
8. my.o : my.c b.h
9.  $(CC) -c my.c
10. clean:
11.  rm myapp $(objects)
```

可以在三种地方对宏进行定义：第一是在 Makefile 中，第二是在 Makefile 命令行中，第三是在载入环境中。类似于属性的控制，这几种宏的定义也要区分优先级。Make 在处理 Makefile 时，它将先给内定义的宏赋值，再给载入的 shell 宏赋值，然后给 Makefile 中的

宏赋值，最后才处理 make 命令行中的宏定义。

(2) Makefile 的隐含规则

GNU 的 make 很强大，它可以自动推导文件以及文件依赖关系后面的命令，于是我们就没必要去在每一个[.o]文件后都写上类似的命令，因为，我们的 make 会自动识别，并自己推导命令。只要 make 看到一个[.o]文件，它就会自动的把[.c]文件加在依赖关系中，如果 make 找到一个 whatever.o，那么 whatever.c，就会是 whatever.o 的依赖文件。并且 cc -c whatever.c 也会被推导出来，于是，Makefile 再也不用写得这么复杂，新的 Makefile 又出炉了。

```
objects= main.o my.o
CC=gcc
myapp: $(objects)
 $(CC) -o myapp $(objects)
main.o: a.h
my.o : my.c b.h
clean:
 rm myapp $(objects)
```

GNU make 支持两种不同类型的隐含规则：

1）后缀规则

是定义隐含规则的老风格方法。后缀规则定义了将一个一个具有某个后缀的文件转换具有另外一个后缀的文件的方法。每个后缀规则以两个成对出现的后缀名定义，例如，将.c 文件转换为.o 文件的后缀规则可以定义为：

```
.c.o:
    $(CC) $(CCFLAGS) $(CPPFLAGS) –c –o $@ $<
```

在上面这个例子中 $@表示目标文件 target， $<表示第一个依赖文件，也就是依赖关

系中最左边的那个，此外还有$^，表示所有的依赖文件。

2）模式规则

这种规则更加通用，因为可以利用这种模式规则定义更加复杂的依赖规则。模式规则看起来更加类似于正则规则，但在目标名称的前面多了一个%号，同时可以用来定义目标和依赖文件之间的关系，例如下面的模式规则定义了如何将任意一个 X.c 文件转换为 X.o 文件：

```
1. $(OBJS) : %.o: %.c
2.  $(CC) $(CCFLAGS) $(CPPFLAGS) –c –o $@ $<
```

3）文件引用

在 Makefile 中使用 include 关键字可以把别的 Makefile 包含进来，这很像 C 语言中的#include，被包含的文件会原模原样的放在当前文件的包含位置。

例如有这样几个 Makefile：a.mk b.mk c.mk，还有这样一个文件 foo.make，以及一个变量$(bar)，其中包含了 e.mk 和 f.mk，那么下面的语句：

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

等价于

```
include foo.mk a.mk b.mk c.mk e.mk f.mk
```

make 命令开始时会找寻 include 所指出的其他 Makefile，并把内容安置在当前位置。如果文件没有指定绝对路径或相对路径的话，make 首先会在当前目录下寻找，如果当前目录下没找到，那么，make 会在下面的几个目录寻找：

（a） 如果 make 执行时，有“-I”或“--include-dir”参数，那么 make 就会在这个参数指定的目录下寻找

（b） 如果目录<prefix>/include(一般是/usr/local/bin 或/usr/include)存在的话，make 会去找。

如果文件没有找到的话，make 会生成一条警告信息，但不会马上出现致命错误，它会继续载入其他的文件，一旦完成 Makefile 的读取，make 会重试这些没有找到的，或是不能读取的文件，如果还是不行，make 才会出现一条致命信息。

4）Makefile 中的函数

在 Makefile 中可以使用函数来处理变量，从而让命令和规则更为灵活和具有智能，函数调用很像变量的使用，也是以‘$’来标示的，函数调用后，函数的返回值可以当做变量来使用。

例如，“wildcard”函数，可以展开成一列所有符合由其参数描述的文件名。文件间以空格间隔，语法如下：

```
1. $(wildcard PARTERN...)
```

用 wildcard 函数找出目录中所有的.c 文件：sources=$(wildcard *.c)。实际上，GNU make 还有许多如字符串处理函数，文件名操作函数等其他函数。

5）make 的执行

一般来说，最简单的就是直接在命令行下输入 make 命令，GNU make 找寻默认的 Makefile 的规则是在当前目录下一次找到三个文件——“GNUmakefile”，“makefile”和“Makefile”。按顺序找这三个文件，一旦找到，就开始读取文件并执行，也可以给 make 命令指定一个特殊名字的 Makefile，要达到这个功能，要求使用 make 的“-f”或是“--file”参数，例如：make –f hello.makefile

（3） 清空目标文件的规则

每个 Makefile 中都应该写一个清空目标文件（.o 和执行文件）的规则，这不仅便于重编译，也很利于保持文件的清洁。一般的风格都是：

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

更为稳健的做法是：

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

.PHONY 意思表示 clean 是一个“伪目标”，而在 rm 命令前面加了一个小减号的意思就是，也许某些文件出现问题，但不要管，继续做后面的事。当然，clean 的规则不要放在文件的开头，不然，这就会变成 make 的默认目标，相信谁也不愿意这样。不成文的规矩是：“clean 从来都是放在文件的最后”。

（4） 嵌套执行 make，支持多个目录的 makefile 编译

在一些大的工程中，不同模块或是不同功能的源文件放在不同的目录下，可以在每一个目录中都写一个该目录下的 Makefile，这有利于 Makefile 变得更加地简洁，而不至于把所有的东西都写在同一个 Makefile 中，这项技术对于进行模块编译和分段编译有着非常大的好处。

例如，有一个子目录叫 subdir，这个目录下有个 Makefile 文件指明了这个目录下的变异规则。那么总控的 Makefile 可以这么写：

```
subsystem:
 cd subdir && $(MAKE)
```

5.2 课堂任务

课堂任务 3：创建一个总目录，创建一个输出信息的 C 源文件（该文件依赖于 helloworld 程序），以及一个 Makefile，再创建两个子目录，一个放置 helloworld 程序的源文件，一个放置头文件（功能自定），每个子目录都有自己的 Makefile，编译运行，正确

输出结果。

要求：

① 正确编译源文件以及 Makefile 文件

② 正确编译运行，输出结果

③ 提交相关源文件代码，Makefile 文件，以及运行结果的截图的文档，代码思路规范清晰，命名规范。

课堂任务 4：利用所学的有关 Makefile 的知识，阅读 linux0.11 部分中的 kernel 中的 Makefile 文件，并对其进行分析。

要求：

① 正确分析 kernel 部分的 Makefile 文件

② 撰写自己的分析结果

③ 提交分析文档

6. 实验四：Vim 工具使用

6.1 VIM 的基本知识

Vim（Visual Editor Improved）和 Vi 一样具有三种模式：命令模式(Command mode)，插入模式(Insert mode)和底线命令模式(Last line mode)

① 命令模式

用户刚刚启动 Vim，便进入了命令模式

以下是常用的几个命令：

- i 切换到插入模式，以输入字符
- x 删除当前光标所在处的字符
- ：切换到底线命令模式，以在最底一行输入命令

② 底线命令模式

在命令模式下按下: (英文冒号) 就进入了底线命令模式。底线命令模式可以输入单个或多个字符的命令，可用的命令非常多。

在底线命令模式中，基本的命令如下:

键	功能	键	功能
q	退出程序	x	删除一个字符
i	在光标左侧输入正文	dd	删除当前行
I	在光标所在行的行首输入正文	ndd	删除 n 行
a	在光标右侧输入正文	dw	删除一个单词（从光标处开 始）
A	在光标所在行的行尾输入正文	nx	删除 n 个字符
o	在光标所在行的下一行增添新行，光标位于新行的行首	^gndd	删除 n 行正文到缓冲区 g 中
O	在光标所在行的上一行增添新行，光标位于新行的行首	^Gndd	删除 n 行正文追加到缓冲区 g 中
yy	复制当前行	c$	从当前光标处删至行尾
nyy	复制 n 行	d^	从当前光标处删至行首
yw	复制一个词	ndw/dnw	删除 n 个词
y)	复制从光标至句末的所有正文	:w	写盘
y}	复制从光标至句首的所有正文	:wq	写盘退出
p	将缓冲区的内容粘贴到当前光标处	:q!	不存盘退出
^gp	将 g 缓冲区里的内容粘贴到当前行下	:e!	不存盘不退出
^gP	将 g 缓冲区的内容粘贴到当前行上	u	恢复前一步的改变
m	用字符 n 替换当前字符	:e 文件名	编辑指定的文件
nG	将光标定位到第 n 行	:w 文 件名	写指定的文件
^F	向前一屏	:w! 文 件名	强制重写指定的文件

^B	向后一屏	:!cmd	运行一个命令，然后返回
^D	向下半屏	^G	显示当前文件和行号
^U	向上半屏		

6.2 课堂任务

课堂任务五：通过对 Vim 编辑器的学习和使用，熟悉 Unix 类环境下的正文编辑程序及其作用。

要求：

① 学习 Linux 的 vim 编辑的相关基础知识，了解其编辑方式、插入方式和命令方式。

② 进行以下练习：

1) 进入插入模式

a 追加，I 插入，o 在当前行下插入一空行；A 在行尾追加，I 从行首插入，0 在当前行上插入一空行。

2) 复制正文

yy 复制当前行，nyy 复制 n 行，yw 复制一个词，y)复制从光标至句末的所有正文；y} 复制从光标至句首的所有正文，nyx 复制类型为 x。

3) 删除正文

x 删除一个字符，dd 删除当前行，ndd 删除 n 行，dw 删除一个单词（从光标处开始），nx 删除 n 个字符，^gndd 删除 n 行正文到缓冲区 g 中，^Gndd 删除 n 行正文追加到缓冲区 g 中，c$从当前光标处删至行尾，d^从当前光标处删至行首，ndw（或 dnw）删除 n 个词。

4) 替换正文

p 将缓冲区的内容粘贴到当前光标处，^gp 将 g 缓冲区里的内容粘贴到当前行下，^gP

将 g 缓冲区的内容粘贴到当前行上，rn 用字符 n 替换当前字符。

5) 查找定位

nG 将光标定位到第 n 行，^F 向前一屏，^B 向后一屏，^D 向下半屏，^U 向上半屏。

6) 文件操作:w 写盘，:wq（或:ZZ）写盘退出，:q!不存盘退出，:e!不存盘不退出，u 恢复前一步的改变，:e filename 编辑文件名，:w filename 写文件名，:w! filename 重写文件名，:! cmd 运行一个命令，然后返回，^G 显示当前文件和行号。

③ 将上述过程截图并上交系统。

7. 实验五：Git 工具使用

7.1 Git 工具的简单实用

Git 是一套很好的项目开发版本控制工具，它包括了 Git 仓库的创建、Git 常用的基本命令、Git 的分支管理、Git 查看提交历史、Git 标签、Git 远程仓库等功能。

① git 创建仓库

创建一个 git 仓库有如下几种方式：

- git init：初始化一个 git 仓库

```
1. git init [directory]
```

这样就出初始化了名为[directory]的仓库，在该目录下自动生成一个.git 目录，可以查看该目录下还有什么。

- git clone：clone 一个 git 仓库

```
1. git clone <url> [directory]
```

url 为 git 仓库地址，directory 为本地目录。这句的效果是把仓库代码复制到了本地目录。Git clone 时，支持的协议包括 ssh，git，https 等

② git 的基本指令使用

- git config：配置信息

可以通过 git config 来配置用户名和邮箱地址，例如

```
1. git config --global user.name '用户名'
2. git config --global user.email '邮箱'
```

- git add：添加文件到缓存命令，git add 命令可将文件添加到缓存，例如 git add *.c
- git status：查看文件的状态命令
- git diff：查看更新的详细信息命令
- git commit：提交命令，将缓存区内容添加到仓库中，可以在后面加-m 选项，以在命令行中提供提交注释
- git reset HEAD：取消缓存命令，这样我们可以取消已经提交的某个文件
- git rm：删除命令

要从 Git 中移除某个文件，就必须要从已跟踪文件清单中移除，然后提交。可以使用

```
1. git rm <file>
```

如果删除之前修改过并且已经放到暂存区域的话，则必须要用强制删除选项 -f 选项。如果要把文件从暂存区域移除，但仍然希望保留在当前工作目录中，则使用 --cached 选项。如果要删除一个目录，则需要-r 参数，表示递归删除。

- git mv：移动或重命名命令

③ Git 的分支管理

Git 中分支常用的命令包括：

- git branch：查看分支命令
- git branch (branchname)：创建分支命令

- git checkout (branchname)：切换分支命令。默认 git 的分支是 master，当需要切换到其他分支，就需要使用该命令。
- git merge：合并分支命令。可以将任意分支合并到当前分支中去。就可以把指定分支的修改合并到当前分支上。合并时候，合并冲突是非常麻烦的问题。例如在两个分支都修改了同一个文件，这时候再合并，就会出现冲突，需要人工判定。
- git branch -d (branchname)：删除分支命令

④ Git 查看提交历史

在使用 Git 提交了若干更新之后，又或者克隆了某个项目，想回顾下提交历史，我们可以使用 git log 命令查看。

看历史记录的几种选项：

- –oneline ： 查看历史记录的简洁版本
- –graph ： 查看历史中什么时候出现了分支、合并
- –reverse ： 逆向显示所有日志
- –author ： 查找指定用户的提交日志
- –since、–before、 --until、–after： 指定筛选日期
- –no-merges ： 选项以隐藏合并提交

⑤ Git 标签

使用标签可以很方便地记住专门的提交快照。具体使用 git tag 命令来实现，具体参数如下：

- 无参数，查看所有标签
- -a <tagname>，创建/追加一个带注解的标签 tagname

⑥ Git 远程仓库

当你试图通过 git 分享你的代码或者与其他人合作开发，你就需要把数据放到一台公共

服务器上。这时候，就需要使用 git 的远程仓库功能。

- git remote add：添加远程仓库

```
1. git remote add [alias] [url]
```

参数[alias]为别名， [url]为远程仓库的地址

- git remote：查看当前的远程仓库
- git fetch、git pull：提取远程仓仓库

git fetch 的作用是获取远程仓库有没有更新的数据，但没有自动合并，接着，会要求使用 git merge 将服务器上的更新合并到你当前的分支。

而 git pull 则会从远程获取最新版本，并进行自动合并。

- git push：推送新的分支到远程仓库。
- git remote rm：删除远程仓库。

7.2 如何提交一个好的 PR

“一个成功的码农是从一个合格的 PR 开始的”，下文列出了关于该主题的一些摘要。

图1-1 PR 的几个组成部分

如上图所示，所谓 PR，即 Pull Request（拉取需求），主要由以下几个部分组成：

1）题目。

2）内容。

3）评论、提交信息、文件信息等。

为什么 PR 如此重要呢？因为它是开源世界大规模软件开发的基石，也是其最重要的流程点。

- 它是质量保证体系的基石。PR 是真正合入代码和更新的入口，直接影响到最终交付件的质量。
- 它是大规模协作开发的基石。在一个互相不见面的“虚拟世界”，PR 几乎是大家交流最重要的通道和“语言”。
- 它是社区历史的记录。PR 不会被删除（除非它所在的仓库被彻底删除），每一个 PR 都记录在了社区历史中，是社区文化的传承载体之一。

下面让我们来看几个“糟糕”的 PR。

Bad PR Example 1：

这个 PR 主要的问题是无信息量，试问：

- 说要加入这位 maintainer，但是为什么要加的是他而不是其他人？
- 提交这个 PR 时该 SIG 项目不多，为什么要引入这么多 maintainer？
- 新加入的这个 maintainer 后续在项目中的分工和作用是什么？

在后面的 comment 中提交者把这些信息都补全了，所以该 PR 最终被合入了。但是单

看这个 PR 本身，这是个非常不合格的 PR。

Bad PR Example 2：

这个 PR 比第一个例子要好一点——题目写的很清楚，在描述中也做了部分解释。但审批者看到这个 PR“先天”会提出以下问题：

- 为什么要这样建仓？我们对 gcc 版本 10 是重量级的开发么？
- 需不需要像 kernel 那样的组织形式？如果类似内核，那应该只有 gcc 这样的仓通过分支来进行版本区分，而不是像这样一个版本一个仓。

所以对这些自然会遇到的问题在提交 PR 时就应有所解释。

PR Example 3:

这是一个“接近完美”的 PR。

- 题目写得非常清晰：引入一个软件包。
- 主体内容对这个包做什么做了详细的解释，同时说明了上游社区活跃、Licence 友好等情况。

但是，审批的人会问如下问题：

什么业务需要引入这个软件，不能只说这个软件好，“好”软件成千上万，为什么要单

独引入这个？它和我们这个项目有什么关系？当然，提交者后面的回答补全了这些信息：

糟糕的 PR 会对世界产生什么影响呢？下面列出了几条：

- 极大地降低工作效率，增加无谓的资源消耗。
- 延长审批时间，使提交者和审批者双向不满。
- 影响产品交付质量。

那么，如何写一个好的 PR?

1） 一个 PR 对应一个事情，不要把不同的事情放在一个 PR 中，保持 PR 的干净整洁。

2) PR 首要是说清楚 why，就是原因，为什么会有这个 PR，这个 PR 解决了什么问题。

3）PR 中的描述要清晰明了，讲清楚 commit 中的要点是什么。

4）一般来说，一个 PR 必须要有一个 issue 对应，这样才能形成需求和开发代码之间的对应关系。任何进入代码仓的内容是需要有原因的。

Good PR Example 1：

这是一个比较好的 PR 的例子，主要表现在：

- 题目清晰。
- 内容的可读性好，比如说是分了 3 个部分，每个部分做什么讲的非常清晰。
- 还注明了希望谁来检视。

需要说明一点的是，虽然以上用来演示的 PR 都是 TC 管理的 PR（如建仓、管理流程等），更多偏向于技术审批类，但代码提交、特性开发、问题修正、文档写作等的 PR，其本质要求都是一样的。

7.3 课堂任务

课堂任务 6：

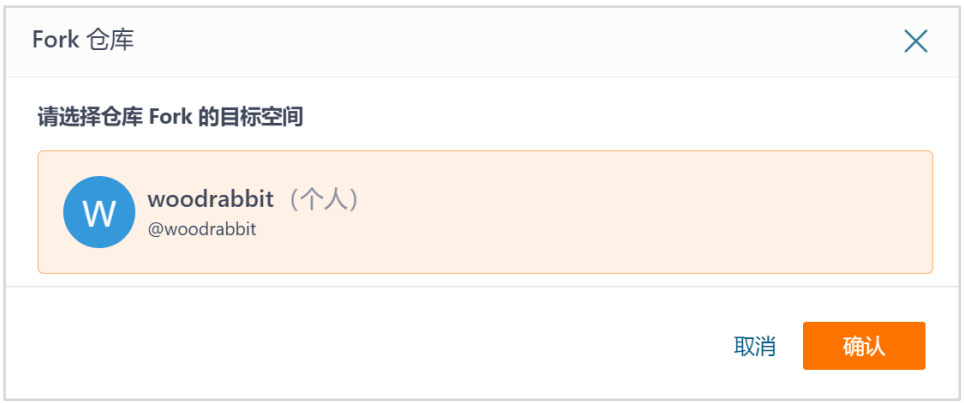
① 准备好自己远程仓库

步骤 1 ：将您要提交 PR 的上游仓库 fork 到您自己的 gitee 账号下

比如，以 openEuler/git-basics/仓库为例：

点击仓库右上角的“fork”按钮即可 fork。

本例 fork 到了一个叫 woodrabbit 的账号下（注意：应该 fork 到你自己的账号下）：



完成后该仓库会出现在自己的主页下：

注意：如果之前已经 fork 该代码仓又想要删除后重新 fork，则可以先进入自己的代码仓，然后进入“Settings（设置）”页面，点击左侧导航栏的“Delete（删除）”链接，在确认操作后，要求对用户进行密码校验确认。校验密码后即可删除仓库。

在 gitee 页面上 fork 完毕后，下面的步骤在命令行窗口运行 git 命令。

步骤 2： 将自己账号下这个 fork 而来的仓库 git clone 到本地

```
1. git clone https://gitee.com/woodrabbit/git-basics.git
2. cd git-basics
3. git status # Be at master branch
```

步骤 3： 创建自己的新分支

```
1. git switch -c woodrabbit # 以自己的 Gitee ID 创建新的分支
2. git branch
3. git status # Be in woodrabbit branch
```

注意命令中“woodrabbit”只是举例，同学们以自己的 Gitee-ID 为名创建分支。

步骤 4： 进入我们这个实验的工作目录

cd ./primary/pr/ # 我们的修改都在这个目录进行，并且注意不要修改这个目录下的其他文件

步骤 5： 以自己的 Gitee ID 为名建立空文件

cd . > woodrabbit # 应用自己 Gitee ID

说明：“cd .”表示改变当前目录为当前目录，等于没改变，故该操作不会有任何输出。将该输出重定向到一个文件即创建了一个空文件。

步骤 6： 将此文件加入到 git 暂存区

```
1. git add woodrabbit # 应用自己 Gitee ID
```

步骤 7： 进行 commit

```
1. git commit -s -m "Add woodrabbit" # 将引号中斜体字换成自己 Gitee ID
```

步骤 8：推送到自己的远端仓库

```
1. git push
```

但是报错，提示第一次 push 要用以下命令：

```
1. git push --set-upstream origin woodrabbit # 注意换成以自己 Gitee ID 命名的
   分支
```

注意：1）在这个过程中，您需要输入 Gitee 账号及其登录密码。2）如果后续还有 commit 要推送到远端仓库的话，直接用 git push 命令就行了。

步骤 9：查看状态

```
1. git status
2. git log
```

② 提交 PR

在前面的步骤中 git push 之后，自己仓库的代远程码上就会出现自己新建的分支（本例是 woodrabbit）：

鼠标点击这个分支就可以切换到这个分支。

步骤 10　新建 Pull Requests

点击“Pull Requests”按钮进入该页面：

点击“New Pull Request（新建 Pull Request）”按钮：

步骤 11　选择自己创建分支提交 PR

选中自己仓库的分支为刚才提交的分支（这里是从自己远端仓库的 woodrabbit 分支合并到上游仓库的 master 分支）。

并填写标题文字和注释（注释说明此次提交做了哪些改动）。比如：

标题可以这样写：woodrabbit - 提交了以自己 Gitee ID 命名的文件（注意前面要改成自己的 Gitee ID）。

注释可以这样写：提交了以自己 Gitee ID（woodrabbit）命名的文件，主程序会将我的 ID 打印出来。

然后点击右下角的“Create（创建）”按钮。

提交成功后原始远程仓（https://gitee.com/openeuler/git-basics）将会显示此次提交：

这时作为社区的一个 contributor，您的作业已经提交完毕。如果您提交的是代码，系统一般会自动进行 test。接下来要等待社区的 committer 进行 review，如果合乎要求，他们会进行/lgtm 动作。之后就需要社区的 maintainer 进行最后确认，如果一切顺利，他们会进行/approve 动作，这之后，您的作业（或代码）就会合到上游远程仓库里去。关于这些流程的具体命令及其含义可以参考以下文档进行了解：

https://clasign.osinfra.cn/sign/Z2l0ZWUlMkZvcGVuZXVsZXI=

https://gitee.com/openeuler/community/blob/master/zh/contributors/Gitee-workflow.md

https://gitee.com/openeuler/community/blob/master/en/sig-infrastructure/command.md

③ 如何运行仓库中的程序

可以运行仓库里的 main.c 程序以观看提交后的效果。该 main 程序可以在 Linux 和类

UNIX 的 macOS 上运行。运行该程序需要执行的命令如下：

```
1. cd primary/pr/src/
2. gcc main.c
3. ./a.out
```

这样凡是正确提交了的 Gitee ID 都可以在屏幕上显示出来。

要求：上交实验截图！

8. 补充知识

8.1 git 补充内容 1：注册 Gitee 账号并签署 CLA

1. 在 Gitee 官网注册 Gitee 账号。
2. 签署个人 CLA。

签署 CLA 的网址也可前往 openEuler 官方网站查看。

注意：签署 CLA 的邮箱地址应与 Gitee 账号关联的“提交邮箱”地址保持一致。

所谓 Gitee ID 即 Gitee 账号，可以在自己的“个人主页”的 URL（Uniform Resource Locator）中查看，如下例所示：

提交邮箱可以在“设置 | 邮箱管理”中查看，如下例所示：

8.2 git 补充内容 2：正确配置 git 账号

按以下命令进行配置：

```
1. git config --global user.name "your-user-name"
2. git config --global user.email "your-email-address-on-gitee"
```

注意：用户名配置成你的 Gitee 账号，邮箱地址配置成该账号对应的提交邮箱（也是签署了 CLA 的邮箱）。在本例中是这样的：

```
1. git config --global user.name   "woodrabbit" # 需要配置成你自己的 Gitee 账号
2. git config --global user.email "woodrabbit@qq.com" # 需要配置成你自己
   Gitee 账号的提交邮箱
3. git config –list # 检查
```