目录

代码阅读及 GDB 单步调试

1. vim 代码编辑器

Vim 是从 vi 发展出来的一个文本编辑器。代码补全、编译及错误跳转等方便编程的功能特别丰富，在程序员中被广泛使用。

简单的来说， vi 是老式的字处理器，不过功能已经很齐全了，但是还是有可以进步的地方。 vim 则可以说是程序开发者的一项很好用的工具。

1.1. Vim

我工作了近十年多年，写程序有七年年，用过各种编程工具，用错过，也用对过，虽然每种优秀的编辑器都有传奇的故事，每个程序员都有自己的脾气，但是，如果让我推荐一款编程工具，那一定是 Vim。

我们可以通过 man vim 命令来了解 vim 的相关说明。

```
VIM(1)                                  General Commands Manual                                 VIM(1)

NAME
       vim - Vi IMproved, a programmer's text editor

SYNOPSIS
       vim [options] [file ..]
       vim [options] -
       vim [options] -t tag
       vim [options] -q [errorfile]

       ex
       view
       gvim gview evim eview
       rvim rview rgvim rgview

DESCRIPTION
       Vim is a text editor that is upwards compatible to Vi.  It can be used to edit all kinds of plain text.
       It is especially useful for editing programs.

       There are a lot of enhancements above Vi: multi level undo, multi windows  and  buffers,  syntax  high-
       lighting,  command line editing, filename completion, on-line help, visual selection, etc..  See ":help
       vi_diff.txt" for a summary of the differences between Vim and Vi.

       While running Vim a lot of help can be obtained from the on-line help system, with the ":help" command.
       See the ON-LINE HELP section below.

       Most often Vim is started to edit a single file with the command

            vim file

       More generally Vim is started with:

            vim [options] [filelist]
```

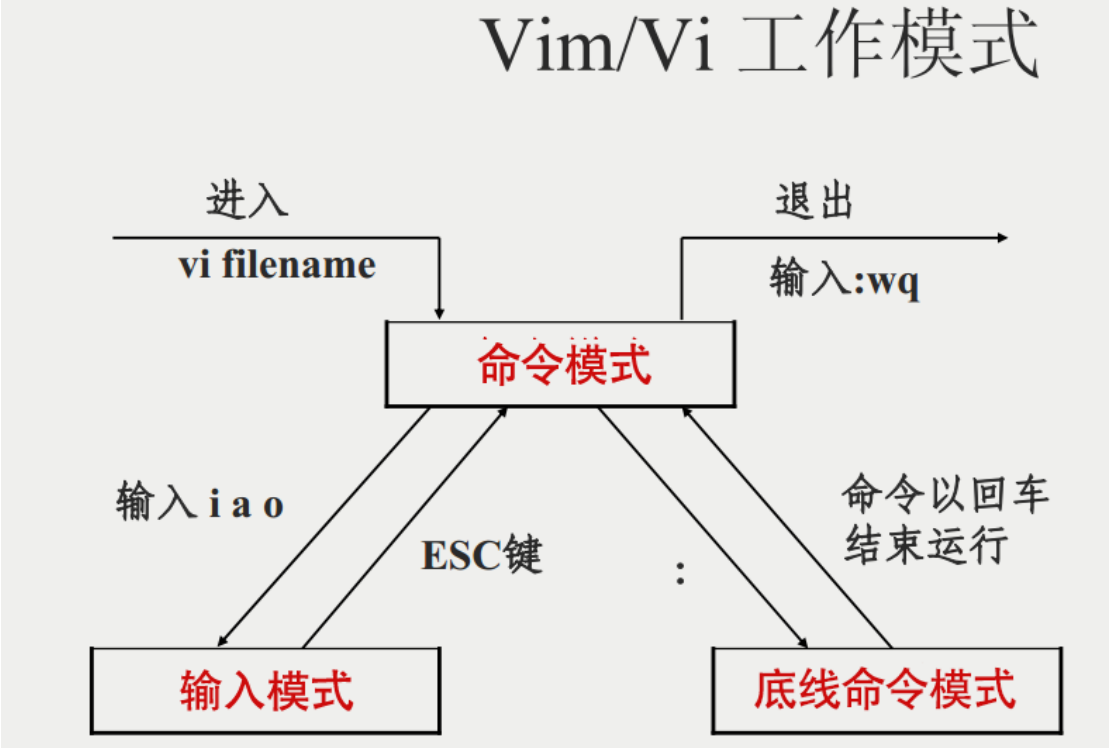
1.1.1. Vim 的模式

想用好 Vim，先要理解 Vim 的模式转换。Vim 常用的模式有四种：

- 普通模式：Vim 启动后的默认模式，用来移动光标、删除文本、覆盖输入文本、恢复操作、粘贴文本等等。
- 插入模式：输入 i 或 a 进入插入模式，在这个模式下敲击键盘会往文字缓冲区增

加文字，相当于普通编辑器的编辑模式。

- 可视模式：选择文本，可以行选、块选和依次选择，选择后可以进行复制、删除、排序等操作。
- 命令模式：执行内部和外部命令，通过“:”“/”“?”“:!”可以进入命令模式，分别对应的是：执行内部命令、向上或向下搜索、执行外部命令。



1.1.2. 命令模式

此状态下敲击键盘动作会被 Vim 识别为命令，而非输入字符，比如我们此时按下 i，并不会输入一个字符，i 被当作了一个命令。

以下是普通模式常用的几个命令：

- i -- 切换到输入模式，在光标当前位置开始输入文本。
- 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。
- a -- 进入插入模式，在光标下一个位置开始输入文本。
- o：在当前行的下方插入一个新行，并进入插入模式。
- O -- 在当前行的上方插入一个新行，并进入插入模式。
- dd -- 删除当前行。
- yy -- 复制当前行。
- p -- 粘贴剪贴板内容到光标下方。
- P -- 粘贴剪贴板内容到光标上方。
- u -- 撤销上一次操作。
- Ctrl + r -- 重做上一次撤销的操作。
- :w -- 保存文件。
- :q -- 退出 Vim 编辑器。

- :q! -- 强制退出 Vim 编辑器，不保存修改。

若想要编辑文本，只需要启动 Vim，进入了命令模式，按下 i 切换到输入模式即可。
命令模式只有一些最基本的命令，因此仍要依靠底线命令行模式输入更多命令。

1.1.3. 输入模式

在命令模式下按下 i 就进入了输入模式，使用 Esc 键可以返回到普通模式。
在输入模式中，可以使用以下按键：

- 字符按键以及 Shift 组合，输入字符
- ENTER，回车键，换行
- BACK SPACE，退格键，删除光标前一个字符
- DEL，删除键，删除光标后一个字符
- 方向键，在文本中移动光标
- HOME/END，移动光标到行首/行尾
- Page Up/Page Down，上/下翻页
- Insert，切换光标为输入/替换模式，光标将变成竖线/下划线
- ESC，退出输入模式，切换到命令模式

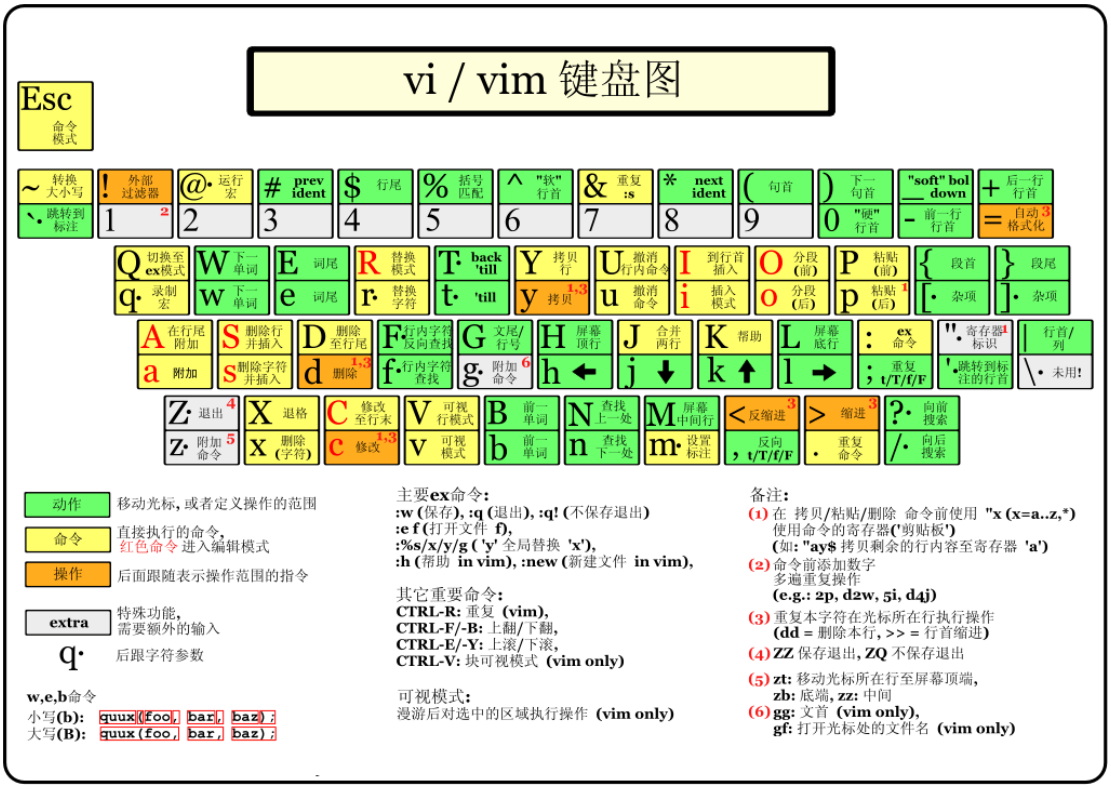
1.1.4. 命令模式

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，基本的命令有（已经省略了冒号）：

- :w：保存文件。
- :q：退出 Vim 编辑器。
- :wq：保存文件并退出 Vim 编辑器。
- :q!：强制退出 Vim 编辑器，不保存修改。

按 ESC 键可随时退出底线命令模式。

1.1.5. Vim 相关快捷键



1.2. 安装 ctags 与 cscope

1.2.1. ctags 插件的作用

ctags 是一个常用的代码索引工具，用于生成代码中各种符号（例如函数、变量、类等）的索引文件。它可以用于各种编程语言，尽管它最初是为 C 语言而设计的，但现在已扩展到支持许多其他编程语言。

ctags 通常与 vim 集成在一起，以便在编辑代码时能够更好地利用 ctags 提供的功能。插件的作用主要包括以下几个方面：

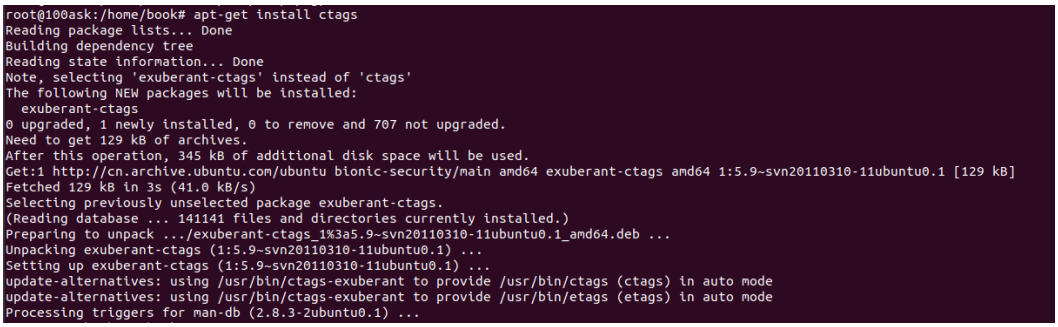
- 代码导航：ctags 生成的索引文件可以让你更轻松地浏览和导航代码。通过索引文件，你可以快速地查找函数、类、变量等定义的位置，而无需手动搜索。
- 跳转到定义：当你在编辑器中将光标悬停在一个符号上并触发快捷键或命令时，ctags 插件可以帮助你快速跳转到该符号的定义处，提高代码阅读和理解效率。
- 查找引用：使用 ctags 插件，你可以快速找到引用了特定符号的所有地方。这对于理解代码的结构和关系非常有帮助。
- 代码补全：一些 ctags 插件还可以与编辑器的自动完成功能结合使用，当你输入代码时，它会提供符号的补全建议，加快编码速度。
- 重构支持：某些高级插件还提供代码重构的支持，例如重命名一个符号，同时更新

所有引用该符号的地方。

1.2.2. 安装 ctags

可以使用以下命令直接安装，如图所示：

```
sudo apt-get install ctags
```



1.2.3. Cscope 插件的作用

虽然 tags 是一个很有用的工具，但在我们常见的代码跳转操作里它只做到了一半：可以通过符号名称（从使用的地方）跳转到定义的地方，但不能通过符号名称查找所有使用的地方。这时候，我们有两种基本的应对策略：

- 使用搜索工具：grep
- 使用专门的检查使用位置的工具：Cscope

由于 Vim 直接内置了对 cscope 的支持，因此我们采用 Cscope。

根据 Cscope 的文档，它的定位是一个代码浏览工具，最主要的功能是代码搜索，包括查找符号的定义和符号的引用，查找函数调用的函数和调用该函数的函数，等等。查找引用某符号的地方、调用某函数的地方、包含某文件的地方，就是 Cscope 的独特之处了。

1.2.4. 安装 cscope

```
sudo apt-get install cscope
```

```
root@100ask: /home/book
File  Edit  View  Search  Terminal  Help
root@100ask:/home/book#
root@100ask:/home/book#
root@100ask:/home/book# sudo apt-get install cscope
Reading package lists... Done
Building dependency tree
Reading state information... Done
Suggested packages:
  cscope-el
The following NEW packages will be installed:
  cscope
0 upgraded, 1 newly installed, 0 to remove and 707 not upgraded.
Need to get 209 kB of archives.
After this operation, 1,247 kB of additional disk space will be used.
Get:1 http://cn.archive.ubuntu.com/ubuntu bionic/universe amd64 cscope amd64 15.8b-3 [209 kB]
Fetched 209 kB in 3s (79.8 kB/s)
Selecting previously unselected package cscope.
(Reading database ... 141147 files and directories currently installed.)
Preparing to unpack .../cscope_15.8b-3_amd64.deb ...
Unpacking cscope (15.8b-3) ...
Setting up cscope (15.8b-3) ...
Processing triggers for man-db (2.8.3-2ubuntu0.1) ...
```

1.3. ctags、cscope 的使用

1.3.1. 创建 tags 数据库

Ctags 有大量的命令行参数，如果我们要对某个目录下的所有 C 代码生成 tags 文件，我们可以使用：

```
ctags --languages=c --langmap=c:.c.h --fields=+S -R .
```

这儿我用了 --languages 选项来指定只检查 C 语言的文件；同时，因为 .h 文件默认被认为是 C++ 文件，所以我使用 --langmap 选项来告诉 ctags 这也是 C 文件。而 --fields=+S 的作用，是在 tags 文件里加入函数签名信息。我们后面会看到这类信息的作用。

这样生成的 tags 文件只考虑符号的定义，而不考虑符号的声明。对于大部分项目，这应该是合适的。如果你希望 Vim 能跳转到函数的声明处，则需要加上 --c-kinds=+p，让 tags 文件包含函数的原型声明。但是这样一来，一个函数就可能有原型声明和实际定义这两个不同的跳转位置，所以通常你不应该这样做。我只对系统的头文件生成 tags 文件时使用 --c-kinds=+p。

1.3.2. 使用 tags 文件

如果当前目录下或当前文件所在目录下存在 tags 文件，Vim 会自动使用这个文件，不需要你做额外的设定。你所需要做的就是在待搜索的关键字上（也可以在可视模式下选中需要的关键字）使用正常模式命令 **Ctrl+]**。

以下是一些常用的快捷键：

- 跳转到函数或变量定义：
  将光标移动到函数或变量名上，按下 Ctrl+]，Vim 会自动跳转到该函数或变量的定义处。

- 返回跳转位置：
  在跳转之后，按下 Ctrl+T，Vim 会返回到之前的跳转位置。
- 列出所有跳转位置：
  在命令模式下，输入:tags，Vim 会显示 tags 文件中所有可跳转的位置列表。
- 在跳转位置之间切换：
  在命令模式下，输入:tnext，Vim 会跳转到 tags 列表中的下一个位置。
  在命令模式下，输入:tprevious，Vim 会跳转到 tags 列表中的上一个位置。
- 在函数调用之间切换：
  将光标移动到函数名上，按下 Ctrl+\或者<C-\>，然后按 g，Vim 会使用 cscope 数据库查找函数的调用关系，并允许您在调用之间切换。
- 查找函数/变量的引用：
  将光标移动到函数或变量名上，按下 Ctrl+\或者<C-\>，然后按 s，Vim 会使用 cscope 数据库查找所有引用该函数或变量的地方。

1.3.3. 创建 Cscop 数据库

进入您的代码项目根目录，在终端中运行以下命令，生成 cscope 数据库文件 cscope.out。cscope 将会收集有关代码中符号、函数调用和引用等信息，以便之后查询和导航。：

```
cscope -Rbq
```

例如我们需要分析 imx6 对应的 Linux-4.9.88，如图所示：

```
root@100ask:/home/book/100ask_imx6ull-sdk/Linux-4.9.88# ls cscope.out -al -h
-rw-r--r-- 1 root root 485M Aug  3 18:58 cscope.out
root@100ask:/home/book/100ask_imx6ull-sdk/Linux-4.9.88#
```

1.3.4. 使用 cscope.out 文件

Cscope 到底有哪些命令，我们可以简要列表如下：

```
cscope commands:
add  : Add a new database              (Usage: add file|dir [pre-path] [flags])
find : Query for a pattern             (Usage: find a|c|d|e|f|g|i|s|t name)
       a: Find assignments to this symbol
       c: Find functions calling this function
       d: Find functions called by this function
       e: Find this egrep pattern
       f: Find this file
       g: Find this definition
       i: Find files #including this file
       s: Find this C symbol
       t: Find this text string
help : Show this message               (Usage: help)
kill : Kill a connection               (Usage: kill #)
reset: Reinit all connections          (Usage: reset)
show : Show connections                (Usage: show)
Press ENTER or type command to continue
```

- g：查找一个符号的全局定义（global definition）
- s：查找一个符号（symbol）的引用
- d：查找被这个函数调用（called）的函数
- c：查找调用（call）这个函数的函数
- t：查找这个文本（text）字符串的所有出现位置
- e：使用 egrep 搜索模式进行查找

- f：按照文件（file）名查找（和 Vim 的 gf、f 命令相似）
- i：查找包含（include）这个文件的文件
- a：查找一个符号被赋值（assigned）的地方

2. vscode 的安装和使用

Visual Studio Code 是微软推出的跨平台编辑器，是一款开源的，可扩展性非常高的，拥有丰富的插件现代编辑器。因为其丰富的插件功能，Visual Studio Code 可以作为 C 语言开发，Web 开发，JAVA 开发，Python 等各种语言的开发工具。

官网：https://code.visualstudio.com

文档：https://code.visualstudio.com/docs

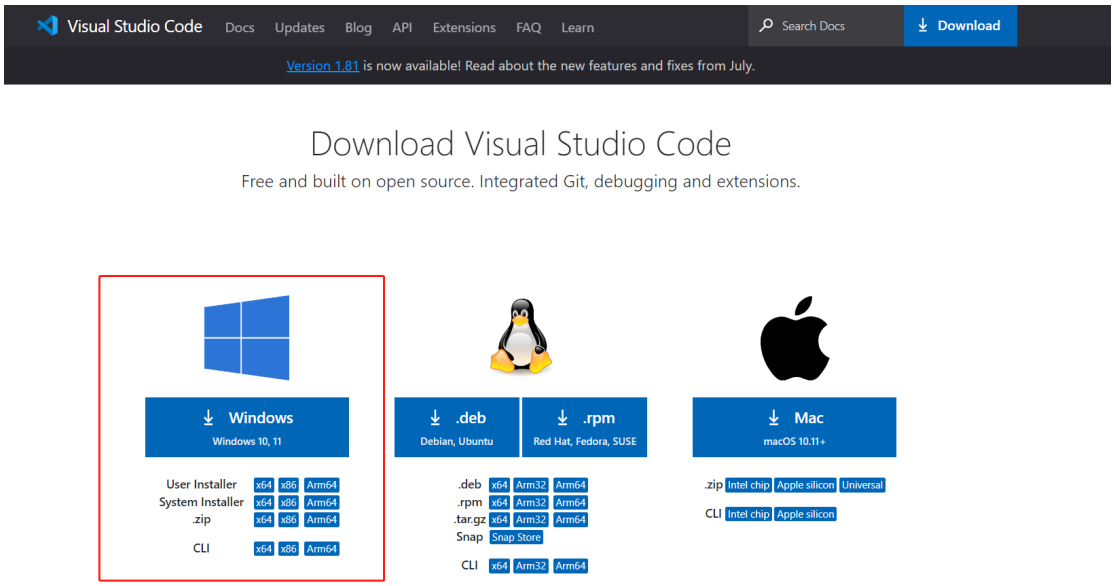
源码：https://github.com/Microsoft/Visual Studio Code

特点：

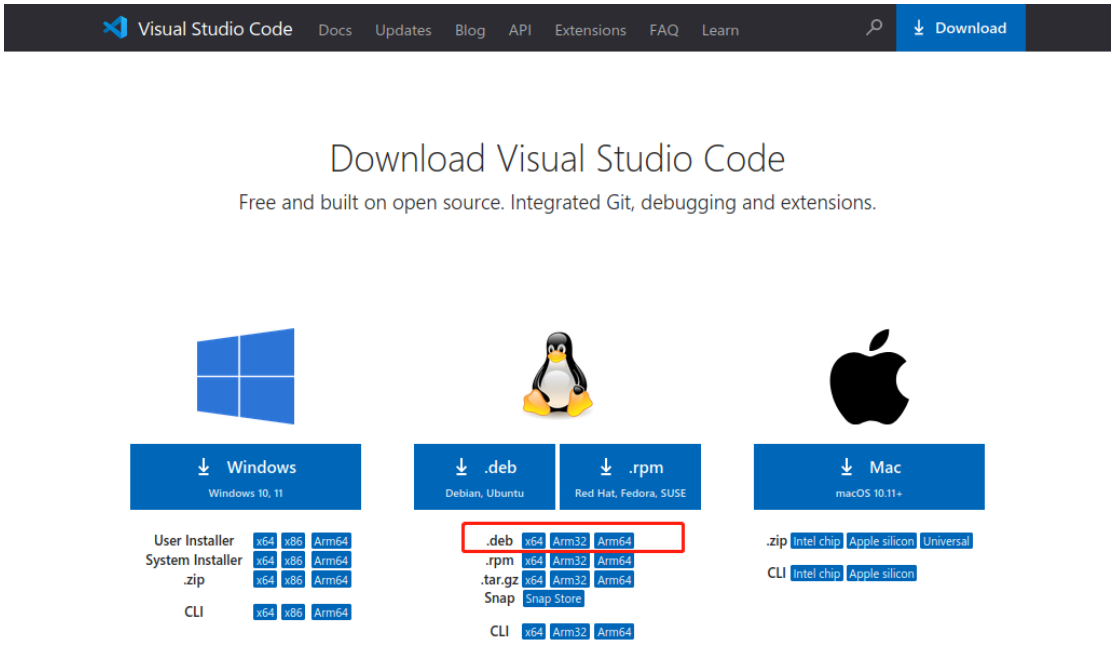
- 开源，支持多语言开发
- 轻量级
- 插件丰富，可扩展性强
- 人性化，宇宙级

2.1. Windows 环境下安装 Visual Studio Code

先进入官网下载(https://code.visualstudio.com/Download)进入如下页面，选择 window 下 System Install 版本下载，根据自己电脑的配置选择相应的版本，如下图所示。



2.2. Ubuntu 环境下安装 Visual Studio Code

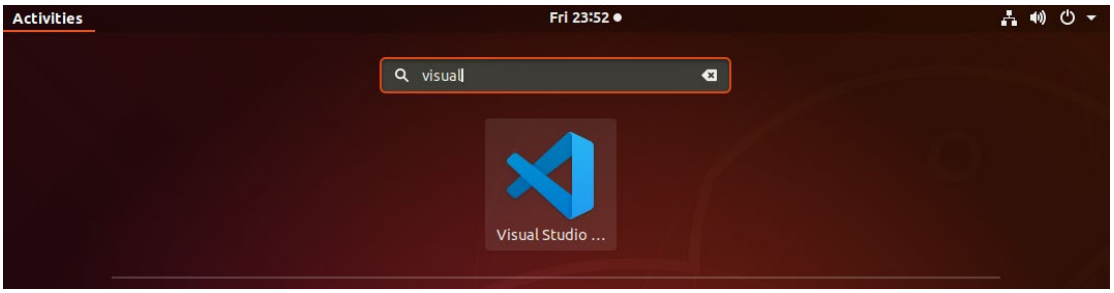


然后使用以下命令进行安装：

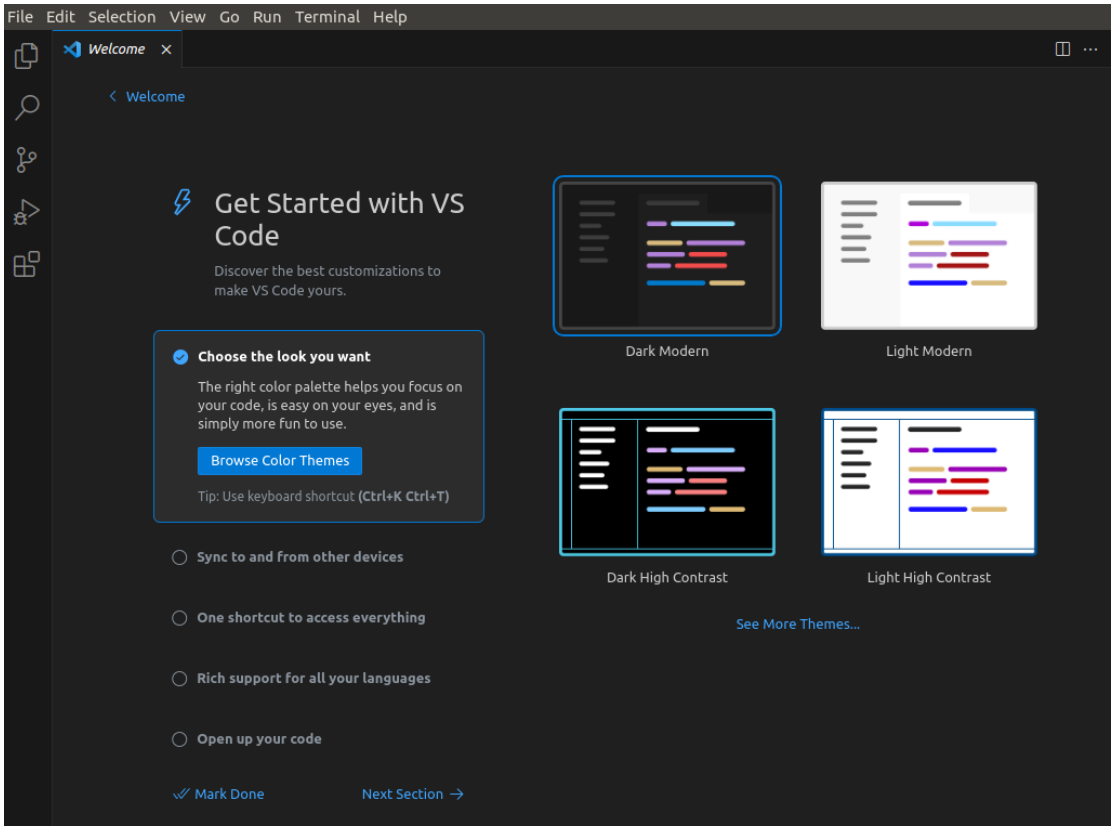
```
sudo dpkg -i code_1.59.1-1629375198_amd64.deb
```

```
book@100ask:~/Downloads$ ls
code_1.81.0-1690980880_amd64.deb
book@100ask:~/Downloads$ sudo dpkg -i code_1.81.0-1690980880_amd64.deb
[sudo] password for book:
Selecting previously unselected package code.
(Reading database ... 178125 files and directories currently installed.)
Preparing to unpack code_1.81.0-1690980880_amd64.deb ...
Unpacking code (1.81.0-1690980880) ...
Setting up code (1.81.0-1690980880) ...
gpg: WARNING: unsafe ownership on homedir '/home/book/.gnupg'
Processing triggers for gnome-menus (3.13.3-11ubuntu1.1) ...
Processing triggers for desktop-file-utils (0.23-1ubuntu3.18.04.2) ...
Processing triggers for mime-support (3.60ubuntu1) ...
Processing triggers for shared-mime-info (1.9-2) ...
```

安装完以后就可以通过搜索找到，如下图所示：



双击打开即可。打开之后如下图所示：

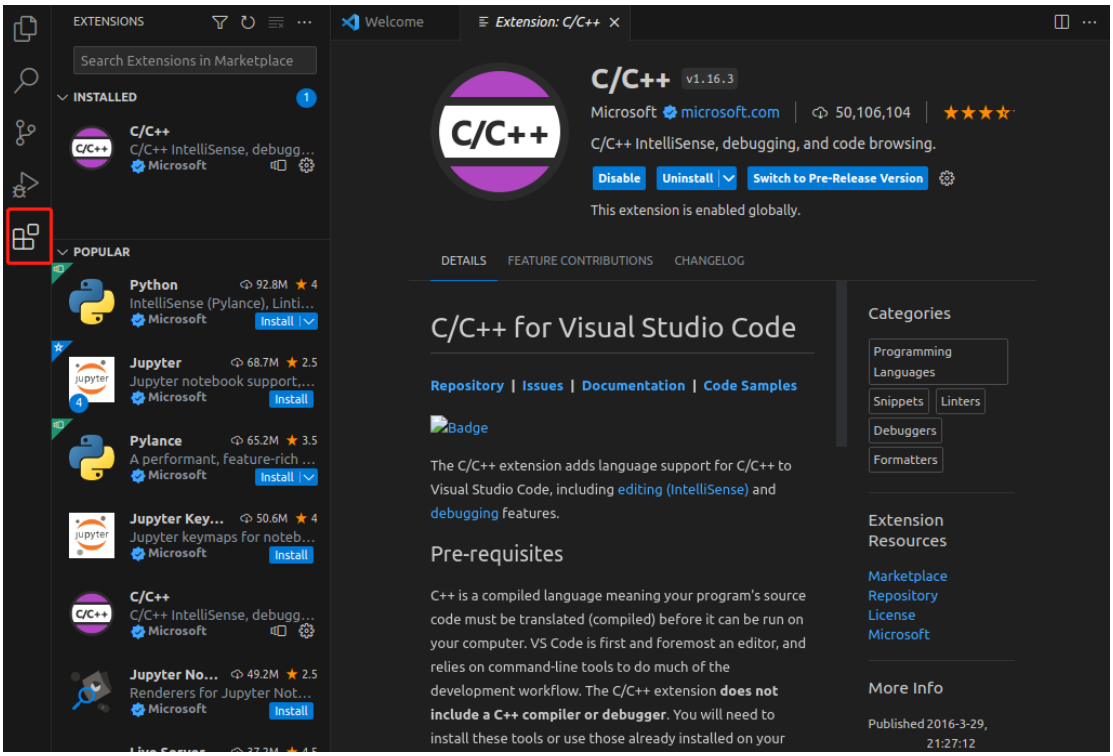


此我们在 ubuntu 环境下的 visual studio code 就安装完成了。

2.3. Visual Studio Code 插件安装

我们在此以 ubuntu 环境为例，讲解 Visual Studio Code 插件安装。

VSCode 支持多种语言，比如 C/C++、Python、C#等等，对于嵌入式开发的我们主要用 来编写 C/C++程序的，所以需要安装 C/C++的扩展包，扩展包安装很简单，点击菜单栏最下 面的 Extensions 如下图所示：



我们需要安装的插件有下面几个：

(1)、C/C++，C 和 C++的编译环境

(2)、C/C++ Snippets，即 C/C++重用代码块。

(3)、C/C++ Advanced Lint,即 C/C++静态检测 。

(4)、Code Runner，即代码运行。

(5)、Include AutoComplete，即自动头文件包含。

(6)、Rainbow Brackets，彩虹花括号，有助于阅读代码。

(7)、One Dark Pro，VSCode 的主题。

(8)、GBKtoUTF8，将 GBK 转换为 UTF8。

(9)、Arm Assembly，即支持 ARM 汇编语法高亮显示。

(10)、Chinese(Simplified)，即中文环境。

(11)、vscode-icons，VSCode 图标插件，主要是资源管理器下各个文件夹的图标。

(12)、compareit，比较插件，可以用于比较两个文件的差异。

(13)、DeviceTree，设备树语法插件。

(14)、TabNine， AI 自动补全插件

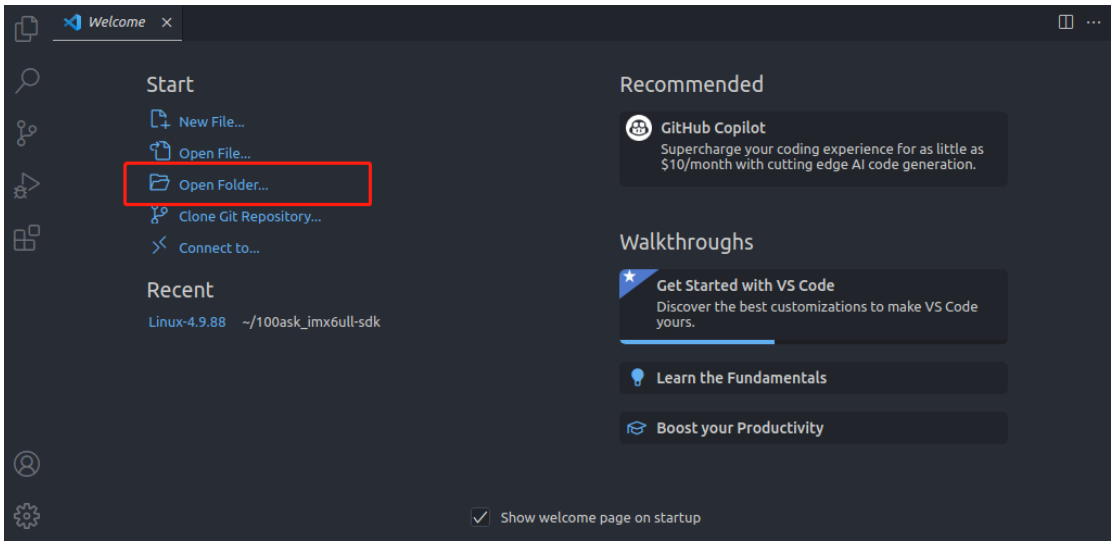
2.4. Visual Studio Code 快捷键的使用

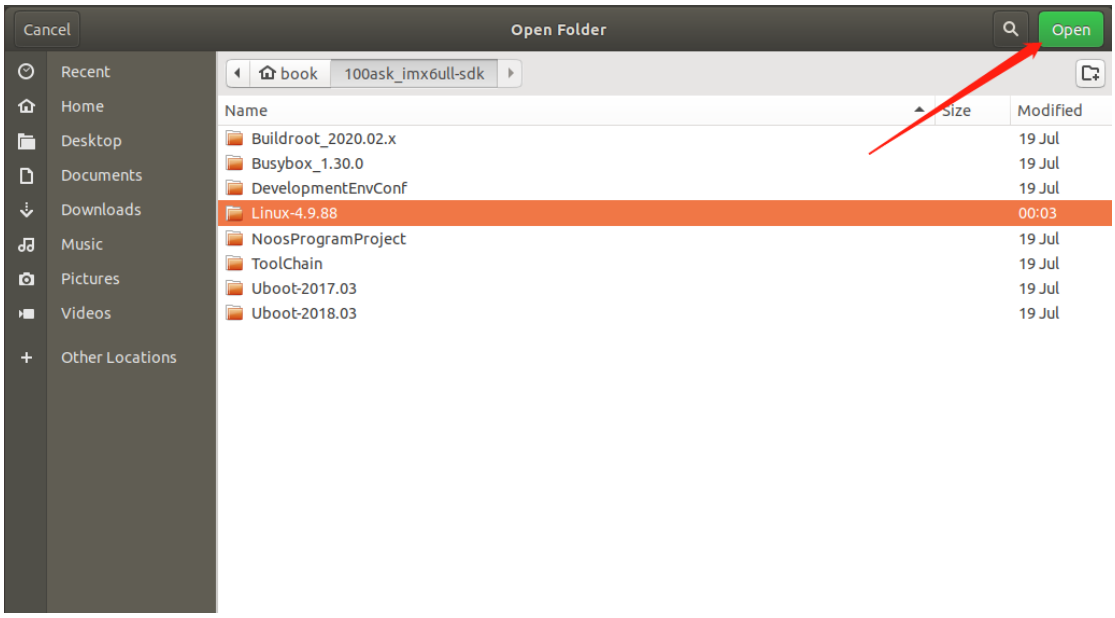
在 Visual Studio Code 软件中使用快捷键可以增加自身的效率，一些常用的快捷键如下所示：

- F1 打开可以输入命令
- F2 重命名变量，方便重构
- F5 运行和调试代码
- F12 去到定义的地方

- shift+F12 查找所有引用
- ctrl+g 会让你输入数字，快速定位到指定行
- ctrl+enter 在下方另起一行
- crtl+f 查找
- ctrl+shift+n 多开一个 vscode 编辑器
- ctrl+b 开关侧边栏
- ctrl+h 替换
- ctrl+r 打开最近文件
- ctrl+` 终端
- ctrl+tab 切换文件
- ctrl+shift+e 切到资源管理器
- ctrl+p 快速打开文件
- ctrl+[左移代码
- ctrl+] 右移代码
- ctrl+/ 行注释
- ctrl+t 匹配文本来打开文件
- ctrl+shift+t 重新打开关闭的文件
- ctrl+shift+home/end 选择光标左侧/右侧全部内容
- ctrl+backspace 删除上一个词
- ctrl+delete 删除光标右侧的词
- ctrl+左/右 跳到上/下一个词
- ctrl+shift+左/右 逐个选词 鼠标滚轮或者 shift+alt+鼠标拖拽 批量选中，方块选择
- ctrl+shift+pageup/pagedown 切换文件
- ctrl+d 选中当前词语
- ctrl+enter 下方插入一行
- alt+左/右箭头 跳回来/过去
- alt+shift+上/下箭头 向上/下复制行
- ctrl++可以放大字体
- ctrl+-可以减小字体。

2.5. Visual Studio Code 阅读 Linux 源码





3. GDB 简介

GDB 是 GNU 开发的一个 Unix/Linux 下强大的程序调试工具。GDB 主要帮忙你完成下面四个方面的功能：

1. 启动你的程序，可以按照你的自定义的要求随心所欲的运行程序；
2. 可让被调试的程序在你所指定的调置的断点处停住；
3. 当程序被停住时，可以检查此时程序中所发生的事；
4. 动态的改变你程序的执行环境。

3.1. GDB 使用流程

GDB 的使用流程主要有：（gcc）编译程序、启动 gdb、运行程序、设置断点、单步执行、查看变量等，下面就从一个程序从编译到启动 gdb 进行调试到断点设置，单步执行到最后退出 GDB 调试整体流程进行介绍。

3.1.1. 编译程序

我们用一个简单的 C 语言程序来演示一下，其代码内容如下：

```
01/**********************************测试程序**********************************/
02#include <stdio.h>
03//实现 a+b
04 int add(int a, int b)
05 {
06         return a + b;
07 }
08//实现 a-b
09 int sub(int a, int b)
10 {
11         return a - b;
12 }
13
14 int main()
15 {
16         printf(" 2 + 3 = %d\n", add(2, 3));
17         printf(" 2 - 3 = %d\n", sub(2, 3));
18         return 1;
19 }
```

接下来，使用命令将其编译，-g 表示可以调试，命令如下：

```
book@100ask:~$ gcc test.c -g -o test
```

3.1.2. 启动 gdb

编译好程序之后，就可以使用 gdb 命令即可进行调试了，输入命令如下：

```
book@100ask:~$ gdb test
```

此时就进入了 gdb 调试模式，系统会阻塞等待用户的指令，如下图所示：

```
book@100ask:~$ gdb test
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from test...done.
(gdb)
```

3.1.3. 查看源码

list(简写 l)： 查看源程序代码，默认显示 10 行，按回车键可以继续看余下的，如下图所示：

```
(gdb) list
1       #include <stdio.h>
2
3       int add(int a, int b)
4       {
5           return a + b;
6       }
7
8       int sub(int a, int b)
9       {
10          return a - b;
(gdb)
```

此时，按回车键后即显出 11~18 行代码，如下图所示：

```
(gdb) list
1       #include <stdio.h>
2
3       int add(int a, int b)
4       {
5           return a + b;
6       }
7
8       int sub(int a, int b)
9       {
10          return a - b;
(gdb)
11      }
12
13      int main()
14      {
15          printf(" 2 + 3 = %d\n", add(2, 3));
16          printf(" 2 - 3 = %d\n", sub(2, 3));
17          return 1;
18      }
(gdb)
```

3.1.4. 运行程序

run(简写 r) ：运行程序直到遇到结束或者遇到断点等待下一个命令，如下图所示：

```
(gdb) r
Starting program: /home/book/test
 2 + 3 = 5
 2 - 3 = -1
[Inferior 1 (process 101606) exited with code 01]
(gdb)
```

可以看出如上图所示的运行结果正是我们程序的打印结果。

3.1.5. 设置断点

可以通过在函数名和行号等上设置断点。程序到达断点就会自动暂停运行。此时可以查看该时刻的变量值，显示栈帧、重新设置断点或重新运行等。断点命令 break 可以简写为 b，命令为 break <断点>。

断点可以通过函数名、当前文件内的行号来设置，也可以先指定文件名再指定行号，还可以指定与暂停位置的偏移量，或者用地址来设置。格式如下表：

格式	说明
break <函数名>	对当前正在执行的文件中的指定函数设置断点
break <行号>	对当前正在执行的文件中的特定行设置断点
break <文件名:行号>	对指定文件的指定行设置断点，最常用的设置断点方式
break <文件名:函数名>	对指定文件的指定函数设置断点
break <+/-偏移量>	当前指令行+/-偏移量出设置断点
break <*地址>	指定地址处设置断点

与断点相关的常用命令如下表示所示：

格式	说明
info breakpoints	显示断点信息
Num	断点编号
Disp	断点执行一次之后是否有效 kep：有效 dis：无效
Enb	当前断点是否有效 y：有效 n：无效
Address	内存地址
What	位置

通过上述命令，在前小节的基础进行实验，实验效果如下图：

```
(gdb) b 5
Breakpoint 1 at 0x654: file test.c, line 5.
(gdb) b 16
Breakpoint 2 at 0x696: file test.c, line 16.
(gdb) b 17
Breakpoint 3 at 0x6b8: file test.c, line 17.
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000000654 in add at test.c:5
2       breakpoint     keep y   0x0000000000000696 in main at test.c:16
3       breakpoint     keep y   0x00000000000006b8 in main at test.c:17
(gdb)
```

可以看出，当我们使用<b “行号”>命令时，可以精确的在目标源码行打上相应的断点，且通过<info breakpoints>能精确的看到断点的相关信息；此时，输入运行指令，即 run(简称 r)即可使程序在运行到断点处，等待下一个执行命令。

```
(gdb) b 5
Breakpoint 1 at 0x654: file test.c, line 5.
(gdb) b 16
Breakpoint 2 at 0x696: file test.c, line 16.
(gdb) b 17
Breakpoint 3 at 0x6b8: file test.c, line 17.
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000000654 in add at test.c:5
2       breakpoint     keep y   0x0000000000000696 in main at test.c:16
3       breakpoint     keep y   0x00000000000006b8 in main at test.c:17
(gdb) r
Starting program: /home/book/test

Breakpoint 1, add (a=2, b=3) at test.c:5
5           return a + b;
(gdb)
```

3.1.6. 单步执行

单步执行的意思时根据源代码一行一行地执行。如第 4 小节所示，当程序已经运行之后，如果想继续执行程序，则需输入命令：continue，让程序继续执行下去。除了 conutinue 命令之后，还有类似指令，如 next、step，它们的区别如下表示：

格式	说明
continue(简写 c)	继续执行程序，直到下一个断点或者结束
next(简写 n)	单步执行程序，但是遇到函数时会直接跳过函数，不进入函数
step(简写 s)	单步执行程序，但是遇到函数会进入函数

此时，在第 4 小节的基础上，测试 continue、next、step 指令，效果如下图所示：

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000000654 in add at test.c:5
2       breakpoint     keep y   0x0000000000000696 in main at test.c:16
3       breakpoint     keep y   0x00000000000006b8 in main at test.c:17
(gdb) r
Starting program: /home/book/test

Breakpoint 1, add (a=2, b=3) at test.c:5
5           return a + b;
(gdb) c
Continuing.
 2 + 3 = 5

Breakpoint 2, main () at test.c:16
16          printf(" 2 - 3 = %d\n", sub(2, 3));
(gdb) n
 2 - 3 = -1

Breakpoint 3, main () at test.c:17
17          return 1;
(gdb) s
18      }
(gdb)
```

3.1.7. 查看变量

使用 GDB 调试程序的好处，不仅仅是让程序按我们想法进行单步运行、停止，更重要的是在此过程中，要观察程序内各种条件或者变量的值的情况，以进行 BUG 排查；查看变更的主要命令有：print、whatis。这里以 print（简写为 p）进行演示：

```
Breakpoint 1, add (a=2, b=3) at test.c:5
5           return a + b;
(gdb) print a
$1 = 2
(gdb) print b
$2 = 3
(gdb) print a+b
$3 = 5
(gdb)
```

从上图可以看出，当我们使用<print 变量名>时可以直接打印出该变量当前的数值，甚至可以使用<print 表达式>打印出表达式的结果。

3.1.8. 显示栈帧

backtrace 命令可以在遇到断点或异常而暂停执行时显示栈帧，该命令简写为 bt。此外，backtrace 的别名还有 where 和 info stack。显示栈帧之后，就可以看出程序在何处停止，以及程序的调用路径。

在第 6 小节的基础上，使用<backtrace >指令，可以看到此时“add”函数的栈，效果如下图所示：

```
Breakpoint 1, add (a=2, b=3) at test.c:5
5           return a + b;
(gdb) print a
$1 = 2
(gdb) print b
$2 = 3
(gdb) print a+b
$3 = 5
(gdb) print a-b
$4 = -1
(gdb) bt
#0  add (a=2, b=3) at test.c:5
#1  0x0000555555554683 in main () at test.c:15
(gdb)
```

关于<backtrace >指令，还有其他用处，如下表示：

格式	说明
bt	显示所有栈帧
bt <N>	只显示开头 N 个栈帧
bt <-N>	只显示最后 N 个栈帧
bt full	不仅显示 backtrace，还要显示局部变量
bt full <N>	

3.1.9. 显示寄存器

显示寄存器指令主要是通过命令<info registers >(简写为 info reg)，进行操作，寄存器尤其在嵌入式领域用处很多，在寄存器名之前添加 $，显示寄存器的内容，例如 p $eax。 例如当我们要查看 PC 指针所指的地址时，可以通过命令<info registers $pc>或<info registers pc>实现，效果如下图所示：

```
(gdb) info registers
rax            0x555555554670   93824992233072
rbx            0x0      0
rcx            0x5555555546c0   93824992233152
rdx            0x7fffffffe518   140737488348440
rsi            0x3      3
rdi            0x2      2
rbp            0x7fffffffe410   0x7fffffffe410
rsp            0x7fffffffe410   0x7fffffffe410
r8             0x7ffff7dd0d80   140737351847296
r9             0x7ffff7dd0d80   140737351847296
r10            0x0      0
r11            0x1      1
r12            0x555555554540   93824992232768
r13            0x7fffffffe500   140737488348416
r14            0x0      0
r15            0x0      0
rip            0x555555554654   0x555555554654 <add+10>
eflags         0x246    [ PF ZF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb) info registers $pc
pc             0x555555554654   0x555555554654 <add+10>
(gdb) info registers pc
pc             0x555555554654   0x555555554654 <add+10>
```

3.1.10. 退出 GDB

使用<quit>命令退出 gdb，效果如下图所示：

```
(gdb) quit
A debugging session is active.

        Inferior 1 [process 4570] will be killed.

Quit anyway? (y or n) y
book@100ask:~$
```

3.2. ARM Linux GDB 使用

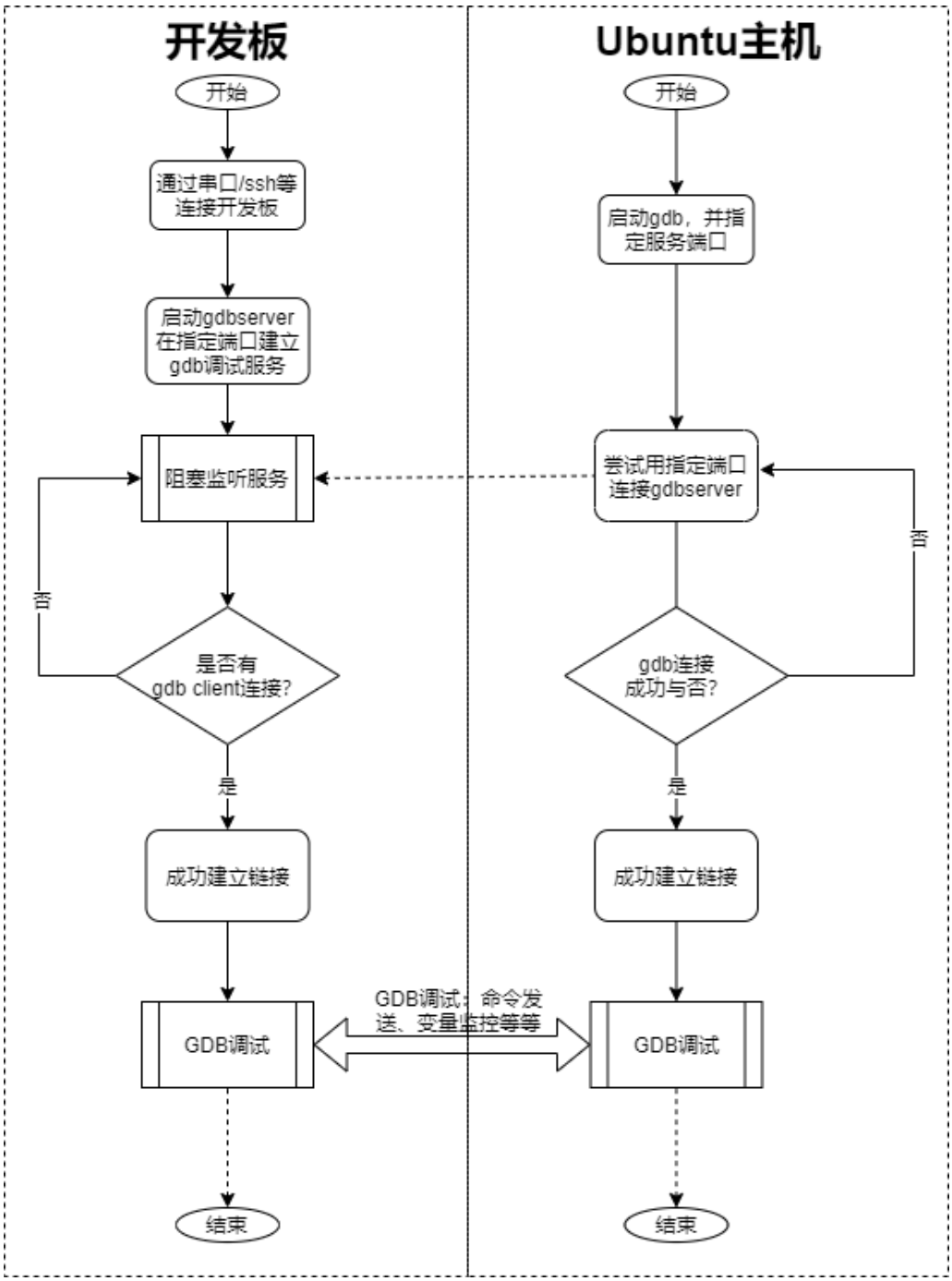
从前文看出，熟悉的 GDB 使用可以有助于我们快速定位程序问题，快速排查出 BUG；按韦老师的口头禅“程序 3 分写，7 分调”，可见建立一个有效的程序调试环境，可以有效提升我们写程序的效率。

3.2.1. 原理简介

按照前面 gdb 使用流程可以看到：程序编译、运行、调试均是在 X86 Linux 主机上运行，所以如果是交叉编译链编译出来的应用程序，那么 1.2 节所用的工具链以及运行环境要在对

应的板子上运行，即在板子上需满足有 gcc 编译器、gdb 调试器等，但一般对于嵌入式系统来说，在资源有限情况下，gcc 编译器及环境一般是在安装 X86 Linux 主机上。那么此时，编译环境与程序运行环境就分开了，如果此时要进行 GDB 调试时，就需要借助 ARM Linux 上的 GDBServer 实现了。即通过网络建立 server 与 client，进而实现 gdb 调试。

GDB Client 及 GDBServer 就构成了 ARM Linux 的调试环境，其框架如下图所示：



3.2.2. 准备程序

我们还是用 1.2 小节中的 C 语言程序来演示一下，其代码内容如下：

```
01/*********************************测试程序*********************************/
02#include <stdio.h>
03//实现 a+b
04 int add(int a, int b)
05 {
06         return a + b;
07 }
08//实现 a-b
09 int sub(int a, int b)
10 {
11         return a - b;
12 }
13
14 int main()
15 {
16         printf(" 2 + 3 = %d\n", add(2, 3));
17         printf(" 2 - 3 = %d\n", sub(2, 3));
18         return 1;
19 }
```

接下来，使用命令将其编译，-g 表示可以调试，命令如下：

```
book@100ask:~$ arm-linux-gnueabihf-gcc test.c -g -o test
```

3.2.3. 启动 gdb

根据前面小节的准备，现需将编译生成的<test>程序拷贝到开发板上，拷贝的方法有很多种，100ASK 提供的虚拟机已经配置了 NFS、SSH 等多种服务，这里以 ssh 为例，将<test>程序拷贝开发板上，已知开发板 IP 为 192.168.1.112，使用命令：

```
book@100ask:~$ scp test root@192.168.1.112:/mnt
```

将<test>程序拷贝到开发板/mnt 目录下，效果如下所示：

```
book@100ask:~$ scp test root@192.168.1.112:/mnt
The authenticity of host '192.168.1.112 (192.168.1.112)' can't be established.
ECDSA key fingerprint is SHA256:MsPueHlgr3jWzOuHMnpE86noPQMASW3GmLt7Ra6i6UM.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.1.112' (ECDSA) to the list of known hosts.
test                                          100%   12KB 920.8KB/s   00:00
```

切换到开发板终端，并通过 gdbserver 在指定端口<12345>建立服务，同时阻塞进行监听，如下所示：

```
[root@imx6ull:/mnt]# gdbserver :12345 test
```

```
[root@imx6ull:/mnt]# gdbserver :12345 test
Process /mnt/test created; pid = 364
Listening on port 12345
```

3.2.4. 主机连接 gdbserver

根据前面第 1、2 小节的准备，并将编译生成的<test>程序拷贝到开发板/mnt 目录下，并成功在开发板的 12345 端口建立了监听服务，此时，需启动主机的 gdb 程序以进行连接，命令如下所示：

```
book@100ask:~$ arm-linux-gnueabihf-gdb    test
```

此时，当出现以下提示时：

```
Reading symbols from test...done.
```

输入命令：

```
target remote 192.168.1.112:12345
```

表示连接至远程 GDB 服务 192.168.1.112，端口 12345；回车之后，可以看到主机已经成功连接上了，如下图所示：

```
(gdb) target remote 192.168.1.112:12345
Remote debugging using 192.168.1.112:12345
Reading /lib/ld-linux-armhf.so.3 from remote target...
warning: File transfers from remote targets can be slow. Use "set sysroot" to access files locally instead.
Reading /lib/ld-linux-armhf.so.3 from remote target...
Reading symbols from target:/lib/ld-linux-armhf.so.3...(no debugging symbols found)...done.
0x76fd79c0 in _start () from target:/lib/ld-linux-armhf.so.3
(gdb)
```

切换至开发板终端，开发板的 gdbserver 会提示有客户端连接成功，如下图所示：

```
[root@imx6ull:/mnt]# gdbserver :12345 test
Process /mnt/test created; pid = 372
Listening on port 12345
Remote debugging from host 192.168.1.108
```

3.2.5. 开始调试

经过第 3 小节的操作，我们已经使得主机与开发板通过网络实现远程 GDB 连接，此时，可以在主机中输入 gdb 调试指令以实现远程调试，以 1.2 小节的几个命令进行演示，如下所示：

```
(gdb) b 5
Breakpoint 1 at 0x103de: file test.c, line 5.
(gdb) b 16
Breakpoint 2 at 0x10428: file test.c, line 16.
(gdb) b 17
Breakpoint 3 at 0x10440: file test.c, line 17.
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x000103de in add at test.c:5
2       breakpoint     keep y   0x00010428 in main at test.c:16
3       breakpoint     keep y   0x00010440 in main at test.c:17
(gdb) r
The "remote" target does not support "run".  Try "help target" or "continue".
(gdb) c
Continuing.
Reading /lib/libc.so.6 from remote target...

Breakpoint 1, add (a=2, b=3) at test.c:5
5           return a + b;
(gdb) c
Continuing.

Breakpoint 2, main () at test.c:16
16          printf(" 2 + 3 = %d\n", sub(2, 3));
(gdb)
```

切换至开发板终端，此时可以看到开发板的按断点执行，并打印出第 16 行的结果，如下图所示：

```
[root@imx6ull:/mnt]# gdbserver :12345 test
Process /mnt/test created; pid = 372
Listening on port 12345
Remote debugging from host 192.168.1.108

 2 + 3 = 5
```

到这里就已经完成了整个流程的演示了，读者可以自行尝试其他 GDB 命令。

3.3. 附录 GDB 常用命令参考

命令	简写形式	说明
backtrace	bt、where	显示 backtrace
break		设置断点
continue	c、cont	继续运行
delete	d	删除断点
finish		运行到函数结束
info breakpoints		显示断点信息
next	n	执行下一行
print	p	显示表达式
run	r	运行程序
step	s	一次执行一行，包括函数内部
x		显示内存内容
until	u	执行到指定行
directory	dir	插入目录
disable	dis	禁用断点
down	do	在当前调用的栈帧中选择要显示的栈帧

edit	e	编辑文件或函数
frame	f	选择要显示的栈帧
forward-search	fo	向前搜索
generate-core-file	gcore	生成内核转储
help	h	显示帮助一览
info	i	显示信息
list	l	显示函数或行
nexti	ni	执行下一行（以汇编代码为单位）
print-object	po	显示目标信息
sharedlibrary	share	加载共享库的符号
setpi	si	执行下一行

3.4. VSCode 搭建 ARM Linux 应用开发调试 IDE

经过前文的介绍，我们已经得到一个具备多种插件于一身的灵活编辑器，因为此时我们还没配置好插件，VSCode 还无法做为我们开发嵌入式 ARM Linux 的集成开发环境的利刃！对于 VSCode 而言，所有配置，都以.json 文件的形式进行纪录，无论是系统配置，主题颜色配置还是快捷配置都是以该形式进行纪录的，.json 文件格式编码统一，也就意味你的配置文件可以跨平台使用，甚至云同步，与他人分享你的配置！举个例子，你在 Windows 下安装的 VSCode 经过一番折腾后，得出一份自己得心应手的环境配置，如果有一天你新装了一个 Ubuntu 系统，并在该系统中安装了一个新的 VSCode，那么此时，只需要将 Windows 下的.json 配置文件复制到 Ubuntu 下，那么 Ubuntu 下的 VSCode 就能立刻变成跟 Windows 一模一样，无须再额外配置，这就是 VSCode 的强大之一：“一次配置，终身可用”。

对于，我们使用 VSCode 进行 ARM Linux 开发也是一样的原理，无非是配置 gcc 编译器的目录，头文件的目录，gdb 调试器的连接地址与端口等等，而这些都以.json 文件的形式储存在电脑中。

经过前面 1.3 节 ARM Linux GDB 的使用以及 VSCode 下载及安装，我们已经有了一个可以与开发板相连接的 Ubuntu 系统、GDB 远程命令行调试、安装好插件的 VSCode 编辑器，下面，我们将分步骤讲解如何有机组以上三个条件，形成一个可以单步调试嵌入式 ARM Linux 应用程序的集成开发环境(IDE)。

3.4.1. 环境介绍

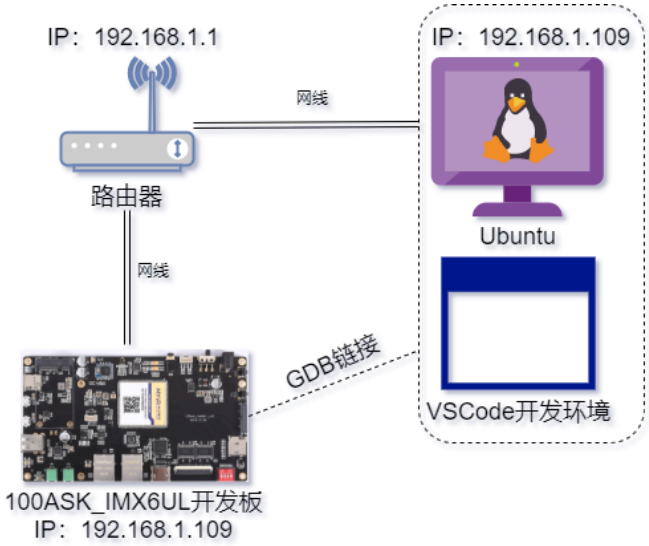
按照前面 gdb 使用的经验，需保证开发板、Ubuntu 系统处于相互 Ping 的网络环境中，然后使用 gdbserver 命令在开发板上建立一个 GDB 服务，由 Ubuntu 系统进行连接并发送相应的调试命令，下面分别介绍 Ubunt 主机、开发板的环境：

Ubuntu 版本：使用 100ASK 提供虚拟机，版本：ubuntu18.04_x64，IP：192.168.1.109

开发板环境：开发板名称：100ASK_IMX6ULL 主控：NXP IMX6ULL IP：192.168.1.110

VSCode 版本：版本: 1.46.1（安装至 Ubuntu 主机并已安装好插件）

可以用如下框架表示：



3.4.2. 调试工程介绍

这里我们以一个简单的 Makefile 流程来演示一份源码工程从 Make 编译到下载至目标开发板，并建立调试的整体流程，该工程由两份 C 语言源码，一个.h 头文件，一个 Makefile 文件构成，工程目录树如下所示：

```
book@100ask:~$ tree    test_prj/
test_prj/
├── Makefile
├── main.c
├── math.c
└── math.h
```

✧ math.c 源码如下：

```
01/*****************************示例调试工程 math.c*****************************/
02#include <stdio.h>
03//实现 a+b
04 int add(int a, int b)
05 {
06         return a + b;
07 }
08//实现 a-b
09 int sub(int a, int b)
10 {
11         return a - b;
12 }
```

✧ math.h 源码如下：

```
01/*****************************示例调试工程 math.h*****************************/
```

```
02#ifndef _MATH_H
03#define _MATH_H
04int add(int a, int b);
05int sub(int a, int b);
06#endif
```

✧ main.c 源码如下：

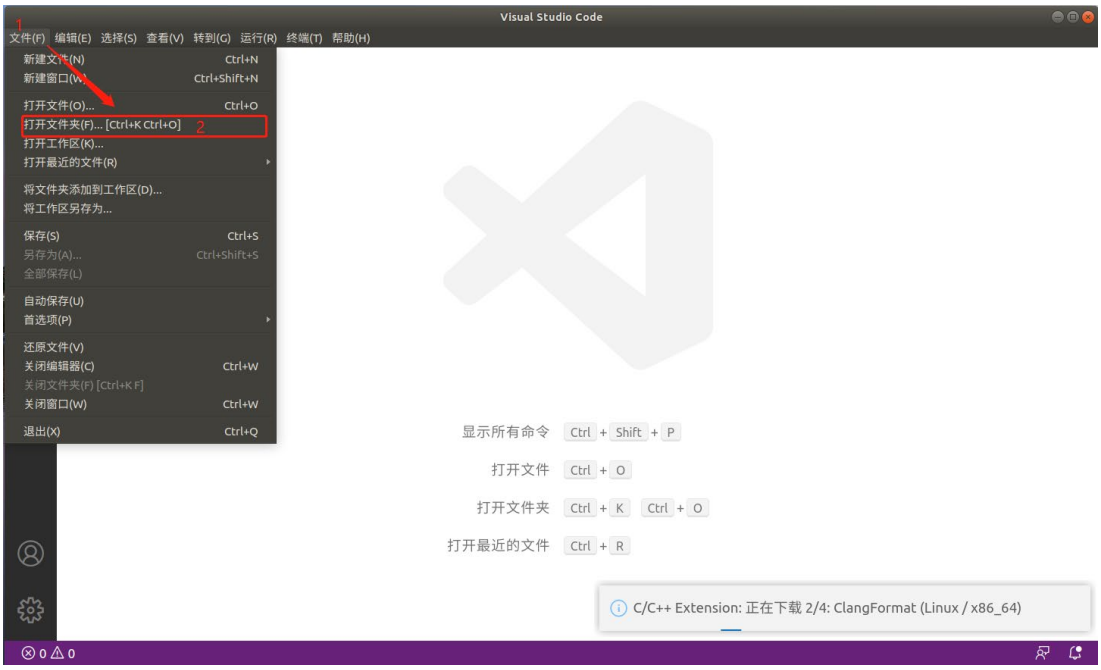
```
01/*****************************示例调试工程 main.c*****************************/
02#include <stdio.h>
03#include”math.h”
04int main( )
05 {
06        printf(" 2 + 3 = %d\n", add(2, 3));
07        printf(" 2 - 3 = %d\n", sub(2, 3));
08        return 0;
09 }
```

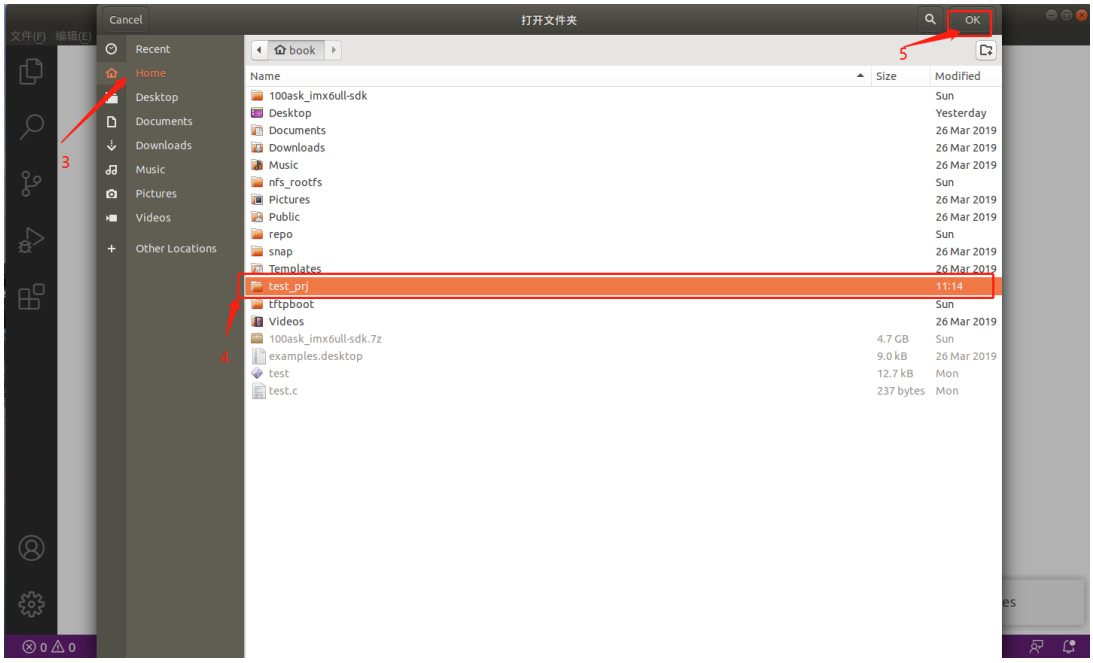
✧ Makefile 源码如下：

```
01 #*************************示例调试工程 Makefile*****************************
02 #板子对应的交叉链，可以换成其他对应的版本即可
03 CC=arm-linux-gnueabihf-gcc
04 Borad_IP=192.168.1.110
05 target=test_app
06
07 all:
08          $(CC) math.c main.c -g -o $(target)
09
10 install:
11
12 clean:
13          rm -rf $(target)
14          rm -rf *.o
```

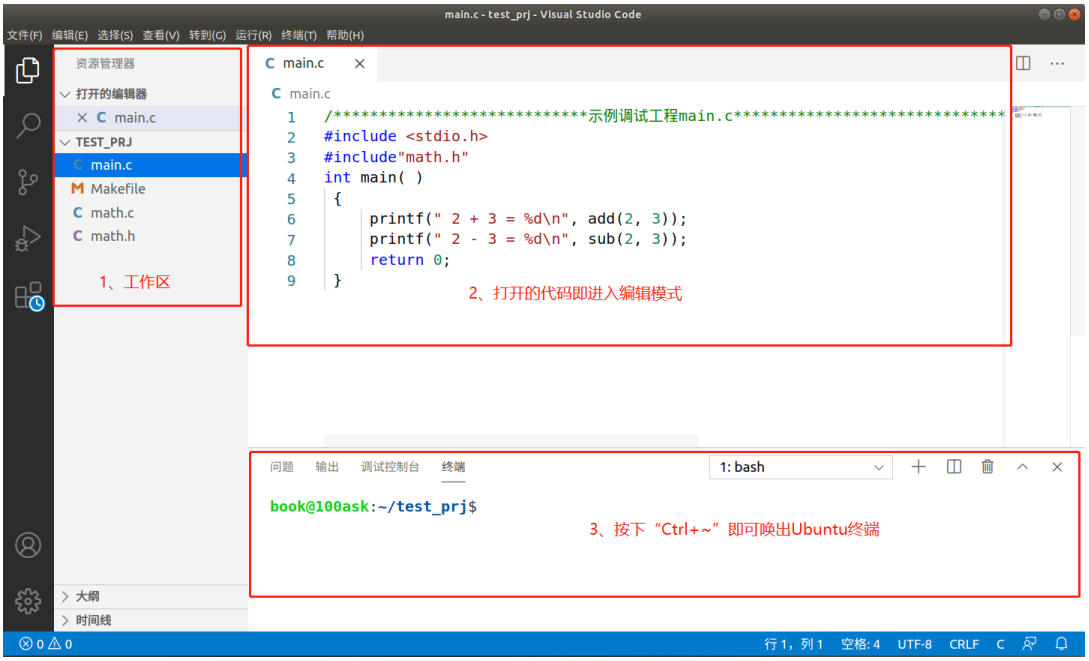
3.4.3. VSCode 配置工程

在 Ubuntu 上打开 VS Code，再选择“打开文件夹”，或者按下快捷键“Ctrl+O”打开对应工程的文件夹<test_prj>即可，如下图所示：





打开工程之后，界面出现下图所示效果，如下图所示：

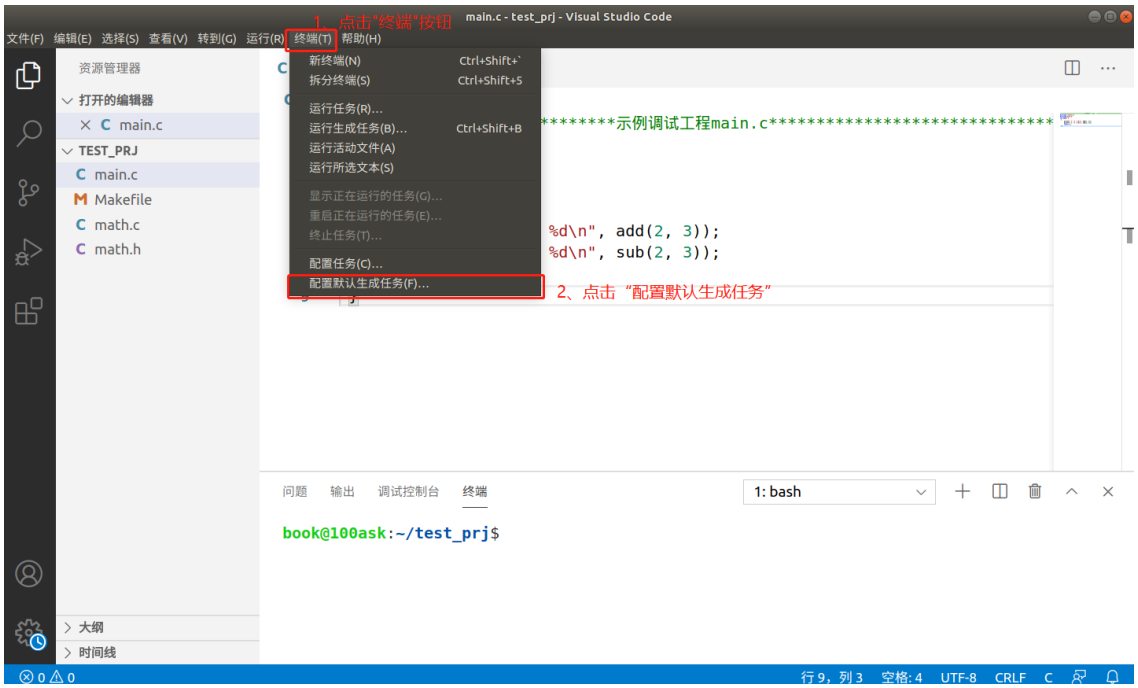


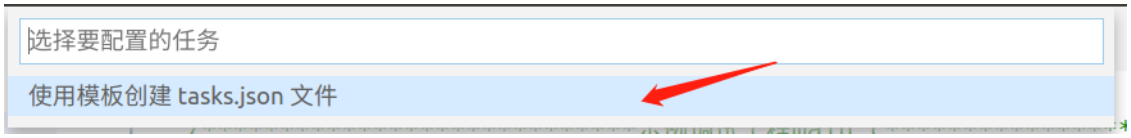
如上面提到，VSCode 的灵活性在于自定义编译任务，而 VSCode 中用于描述编译任务的定义为：tasks.json，所以自定义编译任务就是编写 tasks.json。首先，编写 tasks.json 有以下两种方法：

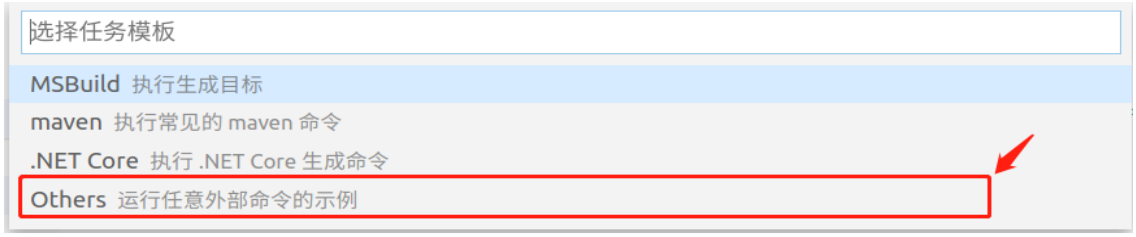
1. 在 View 下打开命令面板(Command Pallette)（快捷键 Ctrl+Shift+p），搜索 Task，选择 Tasks:Configure Task，之后选择 Others，生成 tasks.json 文件。

2. 在菜单栏上面，选择任务(T)-->配置任务(C)...-->使用模板创建 tasks.json-->Others，生成 tasks.json 文件。

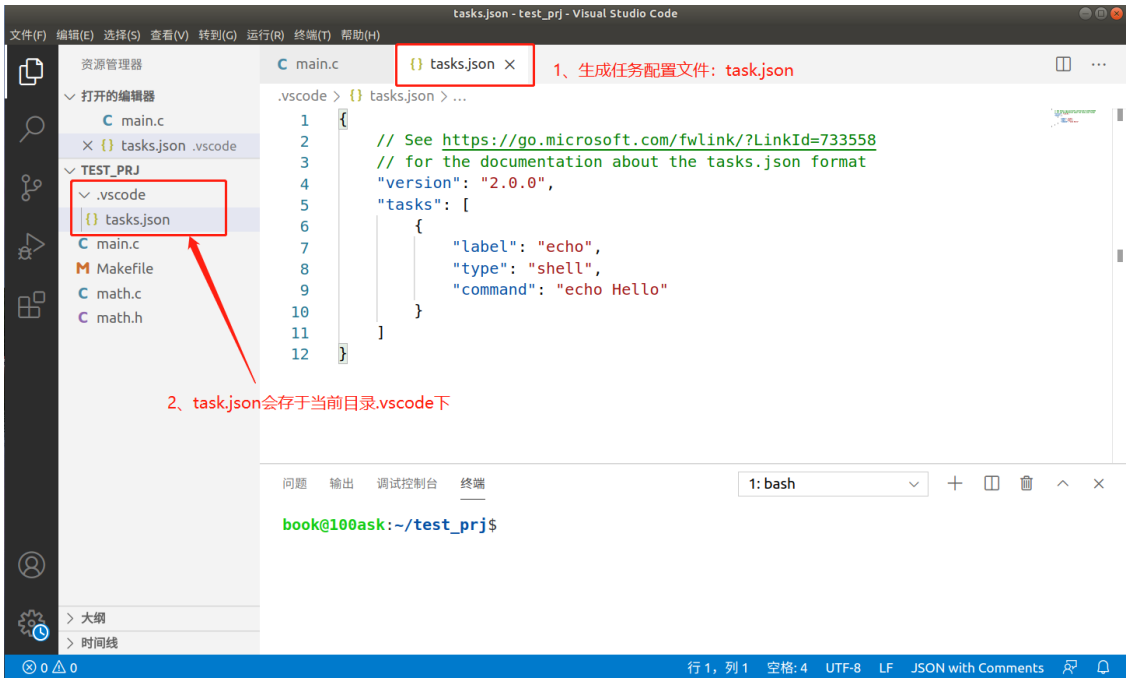
下面以方法 2 为演示，具体操作流程如下图所示：







最终生成任务配置文件：task.json，如下图所示：



task.json 文件解析，见以下//后的备注信息，其实，在 VSCode 的配置文件中，只要将鼠标悬停在 json 左边的字段上，就会自动出现提示该字体的含义。

```
01 {
02       // See https://go.microsoft.com/fwlink/?LinkId=733558
03       // for the documentation about the tasks.json format
04       "version": "2.0.0",
05       "tasks": [
06            {
07                 "label": "echo",     //任务的名字
08                 "type": "shell",      //任务执行模式，这里是 shell 命令行
09                 "command": "echo Hello" //根据前面所选类型表示要执行的具体 shell 命令
10            }
11       ]
12 }
```

根据以上信息，我们可以简单的将 task.json 改成我们编译程序 make 命令

```
01 {
```

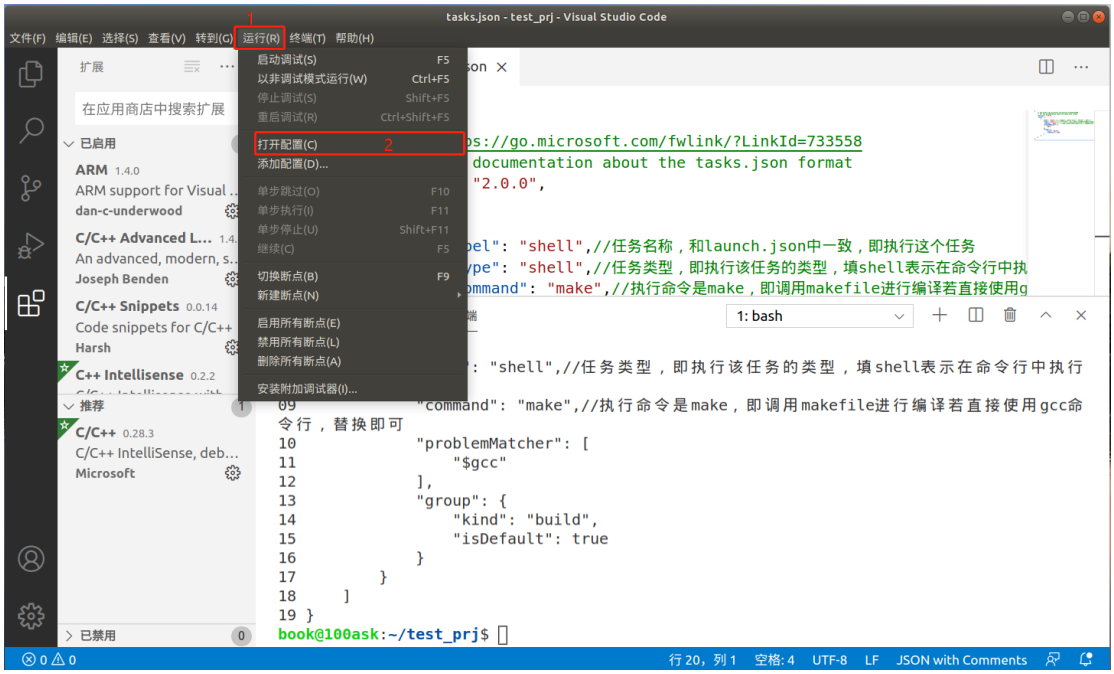
```
02      // See https://go.microsoft.com/fwlink/?LinkId=733558
03      // for the documentation about the tasks.json format
04      "version": "2.0.0",
05      "tasks": [
06          {
07              "label": "make",//任务名称，即执行这个任务
08              "type": "shell",//任务类型，即执行该任务的类型，填命令行中执行的任务
09              "command": "make",//执行命令是 make，即调用 makefile 进行编译
10              "problemMatcher": [
11                  "$gcc"
12               ],
13               "group": {
14                  "kind": "build",
15                  "isDefault": true
16               }
17          }
18      ]
19 }
```

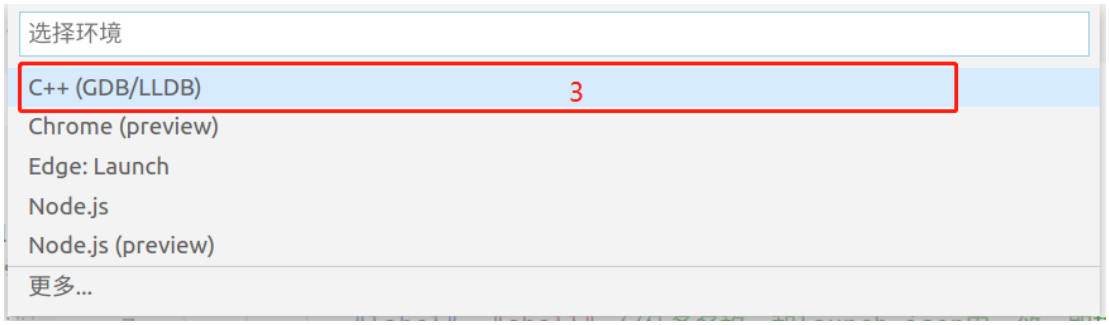
3.4.4. VSCode 配置调试信息

如上面提到，VS Code 的灵活性在于自定义调试任务，而 VS Code 中用于描述调试任务的定义为：launch.json，所以自定义编译任务就是编写 launch.json。首先，编写 launch.json 可以参考以下方法：

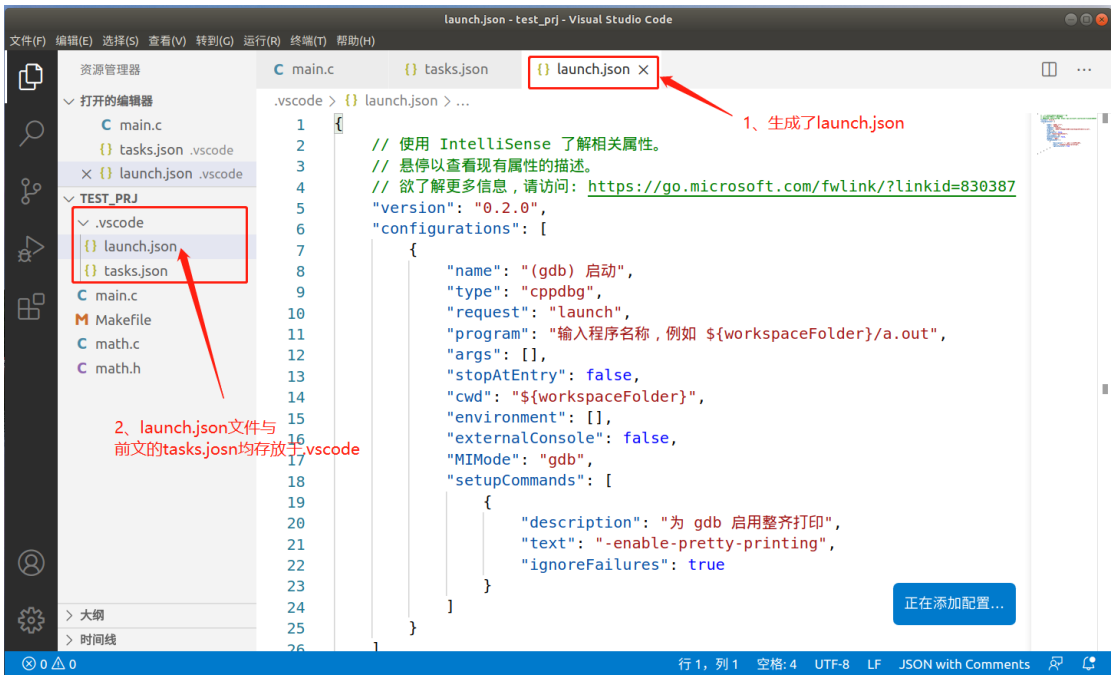
在菜单栏上面，选择调试(D)-->打开配置(C)...-->C++(GDB/LLDB)，生成 launch.json 文件。

如下图所示：





此时，系统就会生成一个默认的配置文件：launch.json，如下所示：



根据以上信息，可以看到 VS Code 帮我们生成的文件里面已经完成大部分工作，只需添加修改相应的必要项即可：

1) 修改可执行文件的位置："program": "enter program name, for example ${workspaceFolder}/a.out",修改成我们 make 编译完成的二进制文件，如本次示例中的 "program": "${workspaceFolder}/test_app"

2) 修改所用 gdb 调试器所在的位置，添加选项："miDebuggerPath", 建议最好使用绝对路径，如本例所示：/home/book/100ask_imx6ull-sdk/ToolChain/gcc-linaro-6.2.1-2016.11-x86_64_arm-linux-gnueabihf/bin/arm-linux-gnueabihf-gdb

3) 修改目标开发板的 IP 和端口，添加选项："miDebuggerServerAddress": "192.168.1.110:12345",如本例所示

4) 其他修改："preLaunchTask":"make", 与之前 tasks.json 中的任务名称要一致，表示在调试前执行的前置任务，当然，除了执行 make，还可以执行其他系统命令，如 make install 等将编译生成文件复制到目标开发板上等等。

```
//最终 launch.json 配置文件如下所示：
01 {
02         "version": "0.2.0",
03         "configurations": [
04
05             {
06                 "name": "(gdb) Launch",
07                 "type": "cppdbg",
08                 "request": "launch",
09                 "program": "${workspaceFolder}/test_app",
10                 "args": [],
11                 "stopAtEntry": true,
12                 "cwd": "${workspaceFolder}",
13                 "environment": [],
14                 "externalConsole": true,
15                 "MIMode": "gdb",
16                 "miDebuggerPath":
"/home/book/100ask_imx6ull-sdk/ToolChain/gcc-linaro-6.2.1-2016.11-x86_64_arm-linux-gnueab
ihf/bin/arm-linux-gnueabihf-gdb",
17                 "miDebuggerServerAddress": "192.168.1.110:12345",
18                 "preLaunchTask":"make",
19                 "setupCommands": [
20                     {
21                         "description": "Enable pretty-printing for gdb",
22                         "text": "-enable-pretty-printing",
23                         "ignoreFailures": true
24                     }
25                 ]
26             }
27         ]
28 }
```

3.4.5. 编译&下载

为了使开发主机上的生成的可执行文件可以复制到目标开发板上，可以通过网络的形式进行传输，毕竟 GDB 也是基于网络的，这样就更方便些，根据前面 GDB 的调试经验，可以通过 SCP 将生成的应用程序复制到开发板 mnt 目录，如下所示：

```
book@100ask:~$ scp test root@192.168.1.110:/mnt
```

因为 VS Code 中已经集成了终端，可以输入上述指令实现应用程序拷贝，又或者修改 makefile，添加 make install 之类的命令，添加后完整的 Makefile 文件如下所示：

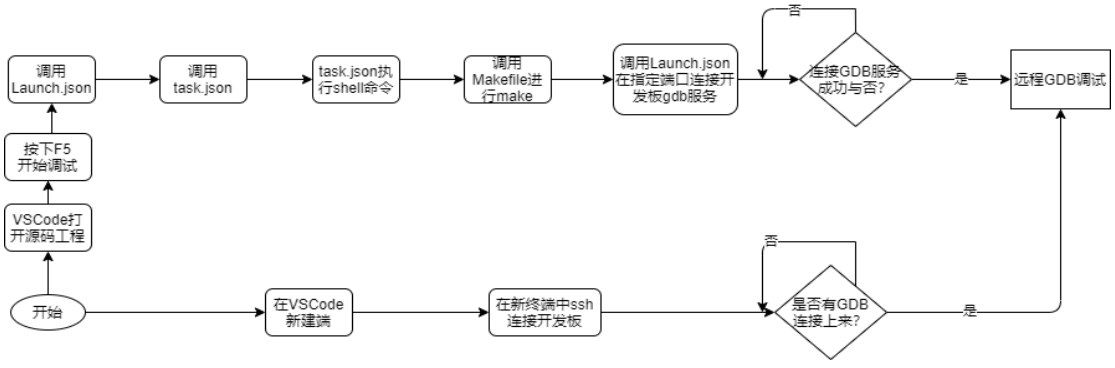
✧ Makefile 源码：

```
01 #*************************示例调试工程 Makefile*******************************
02 #板子对应的交叉链，可以换成其他对应的版本即可
03 CC=arm-linux-gnueabihf-gcc
04 Borad_IP=192.168.1.110
05 target=test_app
06
07 all:
08         $(CC) math.c main.c -g -o $(target)
09
10 install:
11         scp    $(target)    root@$(Borad_IP):/mnt
12 clean:
13         rm -rf $(target)
14         rm -rf *.o
```

然后，修改 VSCode 工程中的 tasks.josn，让它执行 make 之后再执行 make install，如下所示：

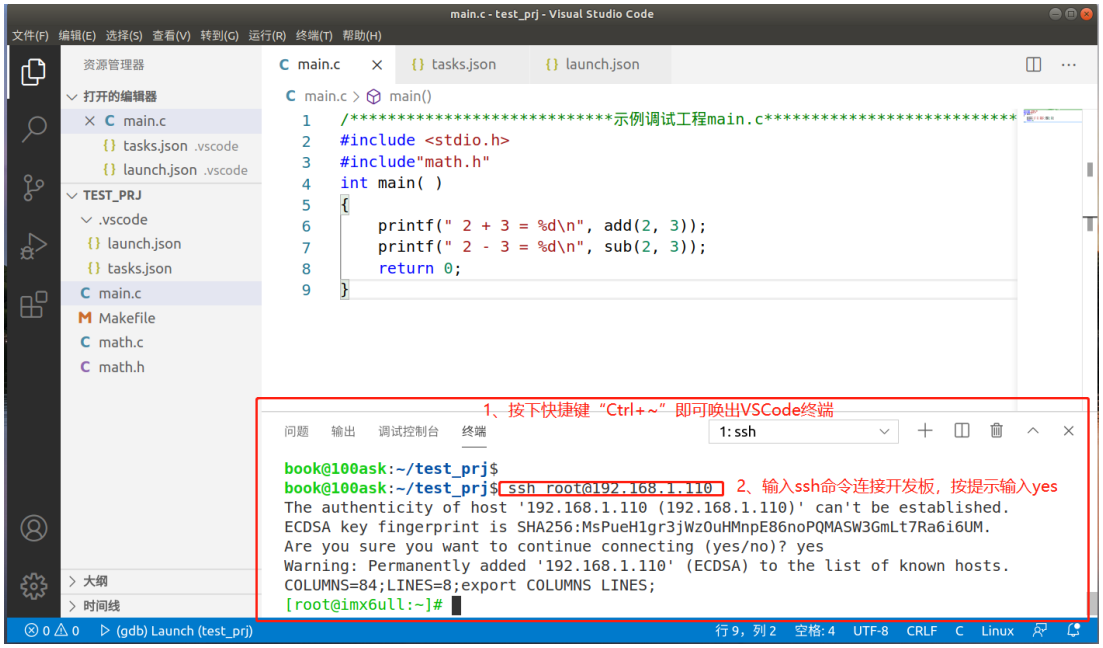
```
"command": "make;make install",
```

总结前文所言，可以将 VSCode 与开发板 GDB 环境的任务配置、编译、调试过程，用如下流程图所示：



3.4.6. 一键调试

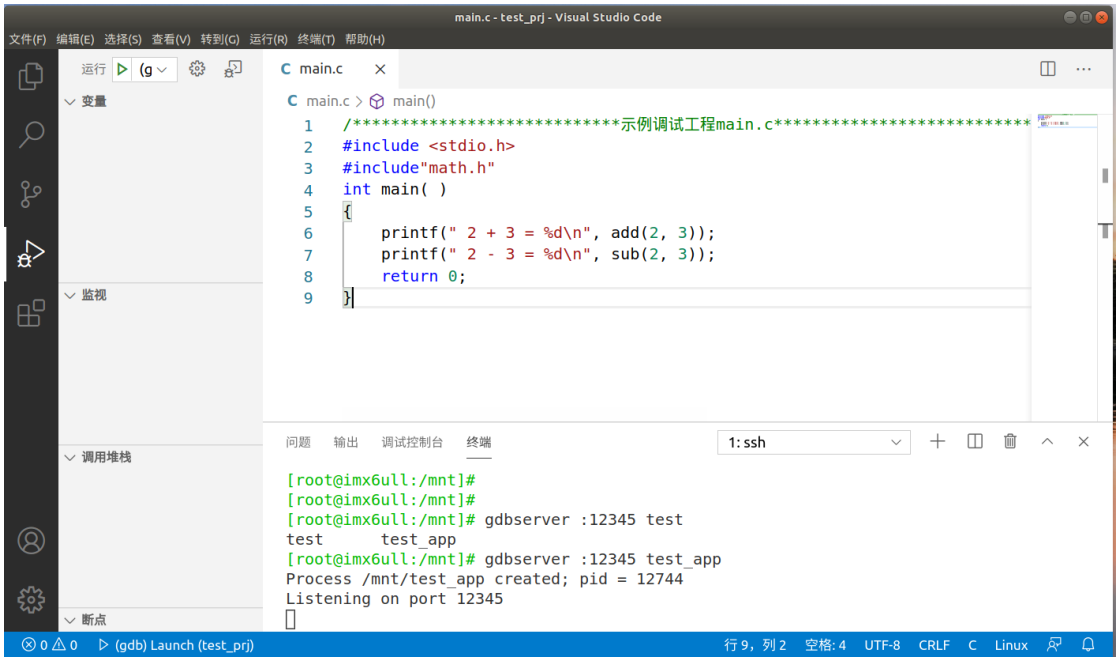
VS Code 还有一个好处就是集成了 linux 终端，使用 VSCode 终端界面，在里面连开发板的 SSH，就不再需要在调试的过程中不断的进行界面切换了，按上文所提开发板的 IP 为：192.168.1.110，如下图所示：



在 VSCode 终端中切换回开发板的终端，并输入指令：

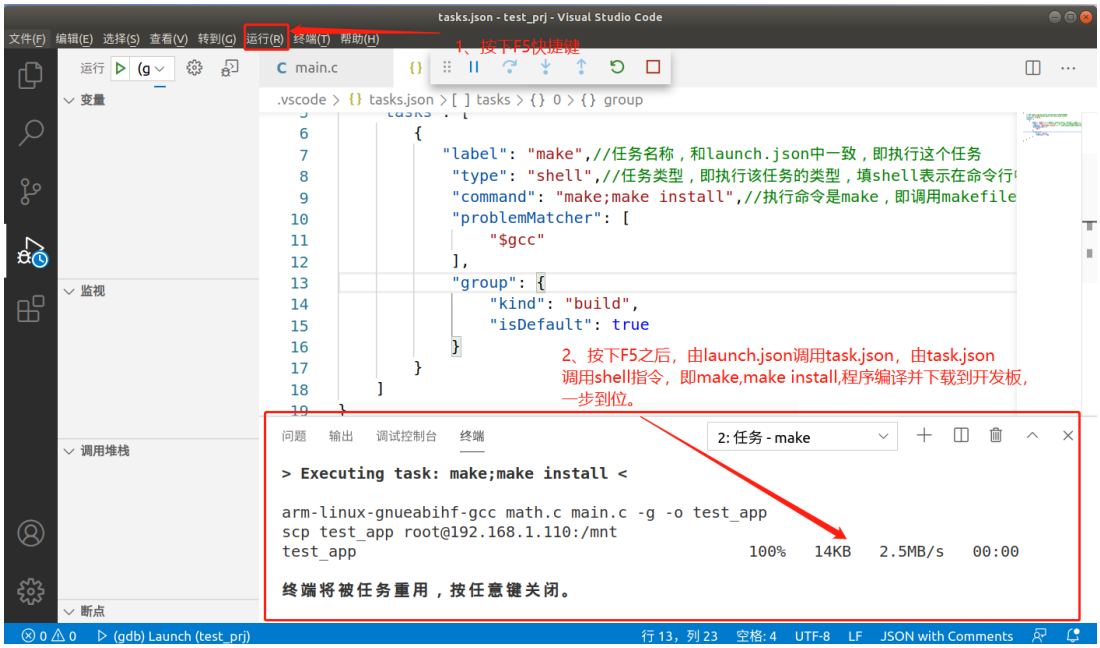
```
[root@imx6ull:/mnt]# gdbserver :12345 test_app
```

此时，VSCode 中连接开发板的终端效果如下所示：

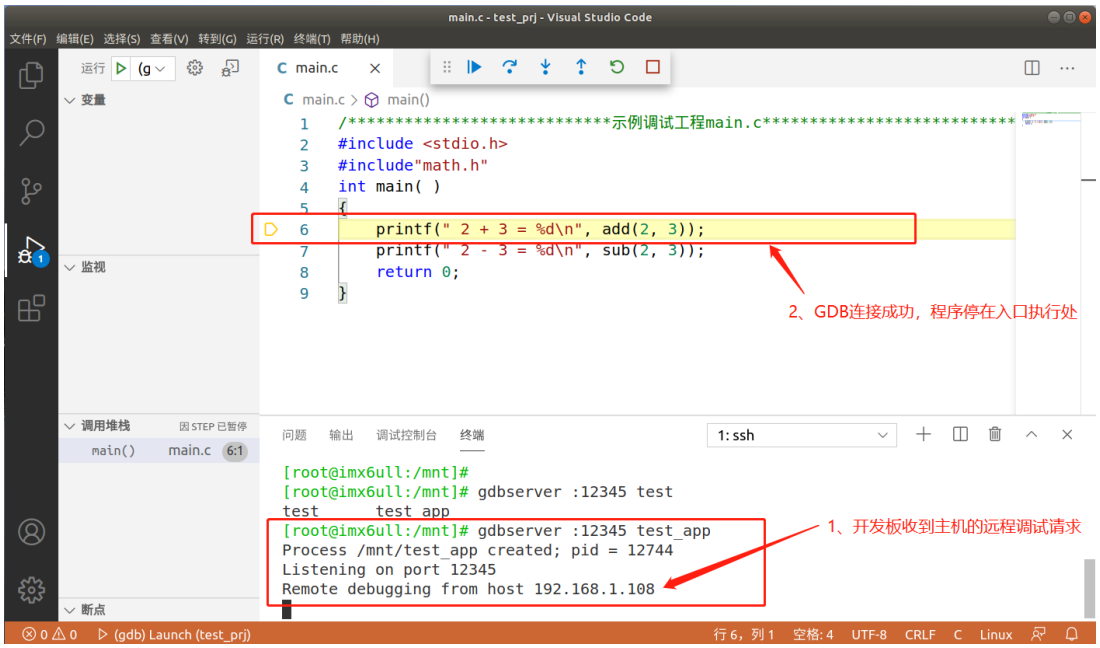


此时，在 VSCode 下面按下快捷键：F5，或者点击 **菜单栏->运行(R)->启动调试(F5)** 即可实现调用 makefile 对整个工程进行 make 并且 make install，将生成的程序复制到开发板，同

时开始启动 gdb 连接开发板指定端口，即 12345，效果如下：



GDB 连接成功，VSCode 的最下边由蓝色变为红色，调试器成功停在程序入口第一行，如下图所示：



3.4.7. VSCode 调试界面说明

经过第 5 小节，VSCode 已经成功与开发板建立远程调试通道，以下介绍在调试模式下 VSCode 的各种调试技能：

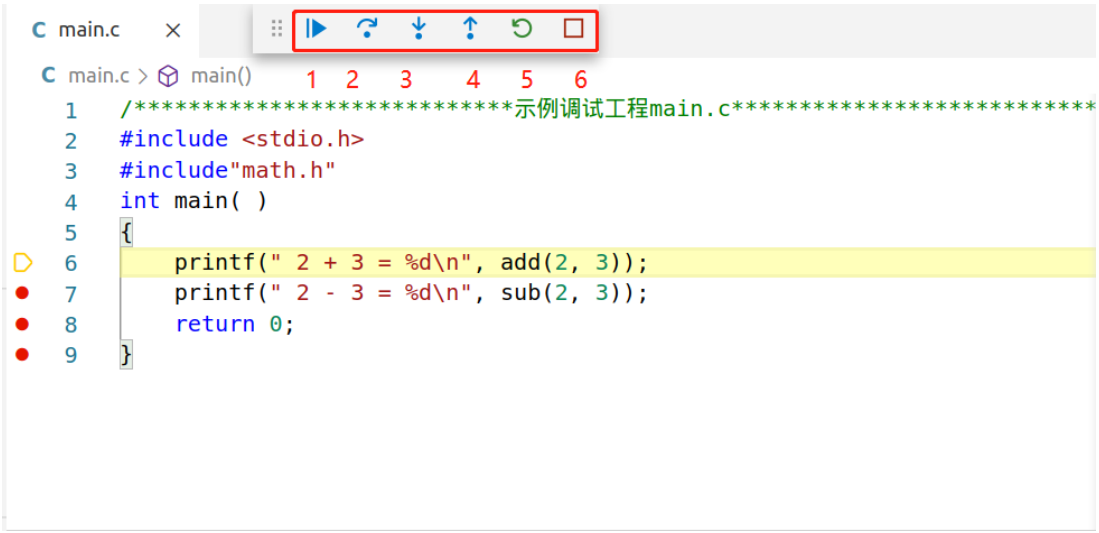
- 打断点

在源码的左边空白处，左键单击即出现一个“小红点”，即为断点：

```
4   int main( )
5   {
6       printf(" 2 + 3 = %d\n", add(2, 3));
7       printf(" 2 - 3 = %d\n", sub(2, 3));
8       return 0;
9   }
```

- 调试工具栏介绍

1：继续运行，直至下个断点：快捷键 F5；
2：单步跳过，直至下个断点：快捷键 F10；
3：单步执行，单步运行程序：快捷键 F11；
4：单步跳出，跳出单步：快捷键 Shift+F5；
5：重启调试器：快捷键 Ctrl+ShiftF5；
6：停止调试，退出当前调试模式：快捷键 Shift+F5；



- 添加变量监视