C_C++_VSCode 的简易使用教程

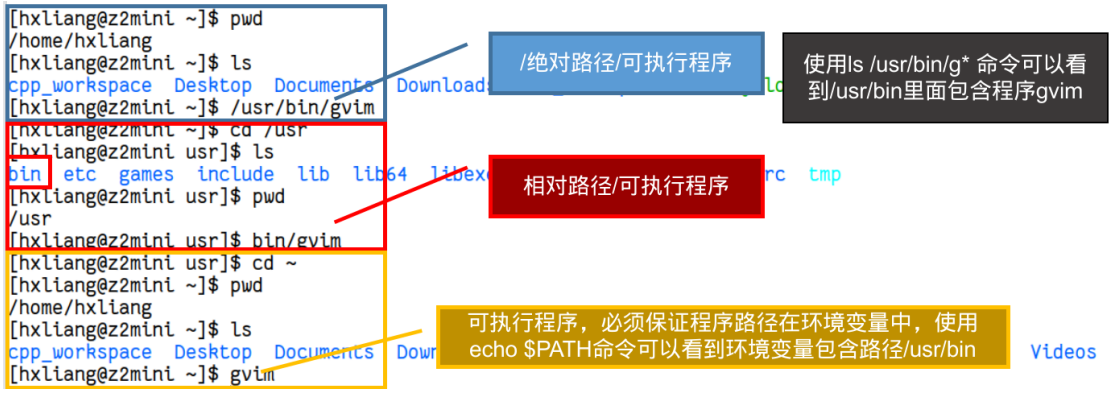
服务器最基本要求：

clangd's version >= 12.0, Git's version >= 2.3, bear, Neovim's version >= 0.8.0

本文教程仅保证在 10.34.24.11 的服务器完全有效，其它服务器可能因为 Linux 配置环境的不完整而无法完全有效，因此，本文建议如果需要进行代码编辑，代码 Debug 等操作时，请先在 10.34.24.11 的服务器完成上述操作，再去 10.34.25.195 和 10.34.24.20 等服务器进行重新编译再跑数据。同时，本文也建议，如果大家仅仅是对代码进行编辑，无需打开 VSCode，使用 Vim/Gvim 即可；只有当需要让代码进入 Debug 模式，浏览代码来了解某一功能的具体实现过程或者需要在代码编辑时频繁地浏览不同文件的代码时使用 VSCode。

本教程仅教如何使用 VSCode 进行代码浏览，编辑代码，编译代码，代码 Debug，Git 历史浏览等功能的基本使用。如何更改 VSCode 的主题和字体，请大家自行上网搜索学习，大家也务必学习这个功能，因为你们以后在组会上展示代码有可能需要。如果需要深入学习使用 VSCode，请自行去谷歌，必应，B 站和 YouTube 等网站进行搜索。例如，搜索 Linux 的 VSCode 如何配置 C/C++编程环境，VSCode 如何配置 C/C++ Debug 环境，VSCode 的 Git Graph 扩展如何使用等。除此以外，本文提供的 VSCode 配置也是最基础简单的配置，更多你们想要的 VSCode 配置，你们可以上网自行搜索（Linux 的）VSCode 如何实现 xxx 功能或 VSCode 如何配置 xxx 等问题来获取到相关配置的教程。

在阅读本文前，建议大家先了解一些使用命令调用 Linux 程序的三种方式（/绝对路径/可执行程序，相对路径/可执行程序，可执行程序），Linux 环境变量，绝对路径，相对路径，家目录符号和根目录符号，以及 Json 的语法。同时，如果在本文出现了大家不熟悉的关键字，请自行使用浏览器搜索学习或者请教他人。



使用命令调用 Linux 程序的三种方式

一，打开 VSCode 的步骤

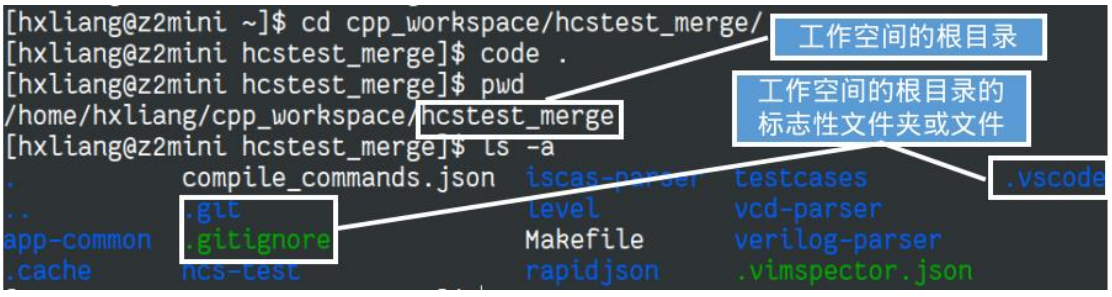


图 1-1

先进入 C/C++项目的工作空间的根目录，然后执行 code .命令，例子如上图 1-1。工作空间的根目录是指整个工作空间的根文件夹，如果当前路径是在工作空间的根目录下，那么直接执行 make/cmake 等相关命令，可以编译整个项目。在根目录下面执行 ls -a 命令，一般可以看到.git, .gitignore, .vscode 或 .svn 等标志性文件夹或文件，这些标志性文件夹或文件是一些辅助性工具识别工作空间的根目录并实现它们辅助功能的必不可少的因素，例如，使用 clangd 等相关软件和扩展实现项目代码的定义跳转和声明查找等功能都需要这些文件夹或文件识别出根目录，否则无法实现相关功能。

二，下载扩展

打开 VSCode 后，按照下图 2-1 下载图 2-1 中的七个扩展，这七个扩展是大家以后做项目和写代码时最常用的扩展，也是最需要的扩展。还有一些重要性不是很高的扩展，如果需要请自行搜索和下载，例如，与 Tcl 语言相关的扩展，与 Bash Shell 脚本相关的扩展以及与 Makefile 相关的扩展等。

图 2-1 里面的 AST 的全称是 Abstract Syntax Tree，中文名字是抽象语法树。抽象语法树的出现是为了解决编译器前后端其中一端的数据结构改变时而导致另一端的代码需要修改的问题。这里出现的 AST，编译器的前后端等关键词请自行搜索相关知识进行学习，本文不作深入介绍。

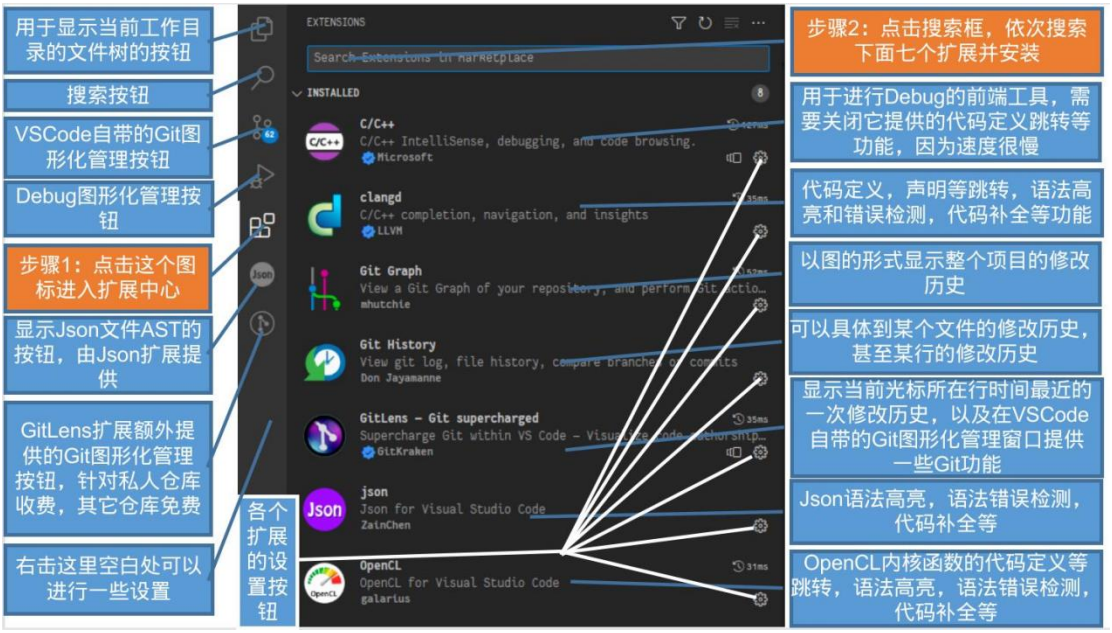


图 2-1

三， VSCode 的命令行终端

图 3-1

先点击 VSCode 的视图按钮（View 按钮）再点击终端选项（Terminal 选项），下图 3-1 的命令行终端窗口将会在 VSCode 底部弹出来，在该终端窗口可以像其它终端一样执行 Linux 支持的所有命令。除此以外，点击上图 3-1 的其它按钮，还能够显示其它窗口，以及使用相关窗口的相关功能，大家可以尝试一下。

四， make，cmake，clangd，bear/compiledb 和 compile_commands.json

4.1 C/C++代码的定义跳转

这章不会详细讲解与 GNUMake 和 CMake 相关的语法和作用等相关信息，只会讲解 make 和 cmake 对于使用 VSCode 的 clangd 扩展实现语法高亮，代码定义或声明跳转等功能的作用。同时，请注意，当只是针对单个 C/C++文件使用 VSCode 时，并不需要生成 compile_commands.json。如果是多个文件，建议使用 GNUMake 管理项目。除此以外，CMake

管理项目并不需要使用 bear/compiledb 生成 compile_commands.json 文件，CMake 自身拥有生成 compile_commands.json 文件的功能（set(CMAKE_EXPORT_COMPILE_COMMANDS 1)）。

Makefile 文件是用于管理 C/C++项目的常用文件，一般和 make 命令相关联。make 命令和 bear 命令结合可以快速生成 compile_commands.json 文件供 clangd 程序分析整个项目的代码，从而允许 clangd 扩展实现语法检测，语法高亮和代码定义跳转等功能。

clangd 扩展实现的功能都是由 clangd 程序提供的，clangd 程序实现了 C/C++的 LSP，提供 API 给 clangd 扩展调用，从而实现语法高亮，代码定义或声明跳转等功能。在这里 clangd 扩展相当于前端，clangd 程序相当于后端。LSP 的全称为 Language Sever Protocol，这是目前主流语言的辅助工具实现语法高亮，代码定义或声明跳转等功能的一个协议，大家无需了解其具体信息，只需要知道这个协议的存在即可。下载完 clangd 扩展后，打开 Linux 终端，执行 mkdir ~/.config/Code/User/ -p 命令，然后执行 gvim ~/.config/Code/User/settings.json 命令，把下面的 clangd 扩展的设置内容复制粘贴到 settings.json 文件中，并保存：

```
{
  "clangd.arguments": [
    "--all-scopes-completion",
    "--background-index",
    "--clang-tidy",
    "--compile-commands-dir=${workspaceFolder}",
    "--header-insertion=never",
    "--completion-style=detailed",
    "--enable-config",
    "--fallback-style=none",
    "--function-arg-placeholders=true",
    "--header-insertion-decorators",
    "--pch-storage=memory",
    "--pretty"
  ]
}
```

clangd 程序如果想提供整个 C/C++项目的 LSP 给 clangd 扩展使用必须先找到一个保存了整个项目的编译信息的文件，这个文件的默认名字是 compile_commands.json。该文件保存了整个项目每一个 C/C++源文件所使用的编译命令，编译参数以及源文件所在目录的绝对路径。如果项目的根目录缺失该文件可能会导致部分代码跳转失败以及语法检测失败。当添加了该文件后，如果 VSCode 依然提示语法错误，可以先关闭 VSCode 再重新打开 VSCode 或先关闭报错的文件再重新打开那个文件。下图 4-1 展示了一个 C/C++项目在 compile_commands.json 文件中所保存的信息。

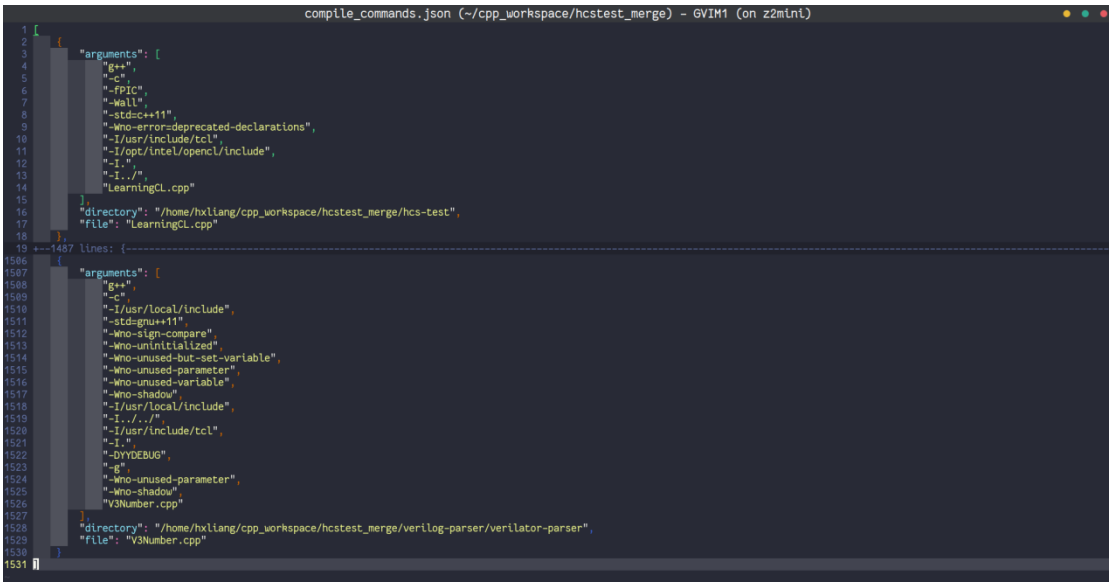


图 4-1

在使用 Makefile 管理项目的 C/C++项目中，compile_commands.json 文件的生成方法有两个，一个是开发者自己创建一个 compile_commands.json 文件并写入相关信息，另一个是使用自动

化生成工具 bear/compiledb 自动生成 compile_commands.json 文件再根据需要手动修改。本章会介绍如何使用 bear 和 compiledb 自动生成 compile_commands.json 文件。

bear 是一个增量式的 compile_commands.json 自动化生成工具。增量式的意思是如果项目已经使用 make 命令编译过一次，如果此时使用 bear 生成 compile_commands.json 文件，bear 只会将 make 命令增量式编译时所使用到的文件的相关信息添加到 compile_commands.json 文件中，如果操作不当，生成的 compile_commands.json 文件可能会是空的，因此，建议操作不熟练的同学，每次想生成新的 compile_commands.json 文件都打开新的 compile_commands.json 文件看看是否为空的。同时，针对不同场景使用 bear 生成 compile_commands.json 文件的建议如下：

（1）第一次为项目生成 compile_commands.json 文件，先执行 make clean 删除所有由编译生成的文件，再执行 bear <make 相关的命令>，例如，想执行 make dbg 命令，那么选择执行的 bear 命令是 bear make dbg；(注意，不同系统或不同版本的 bear 命令会有些区别，建议使用 bear –help 看看具体用法。如，有些版本的命令不是 bear <make 相关的命令>，而是 bear -- <make 相关的命令>，多了两横分隔命令，两横前面是 bear 的参数，两横后面是 make 命令以及 make 命令自己的参数。这种两横分隔命令的方式，在一个命令 wrap 另一个命令时是一种挺常用的操作，大家可以记一下。)

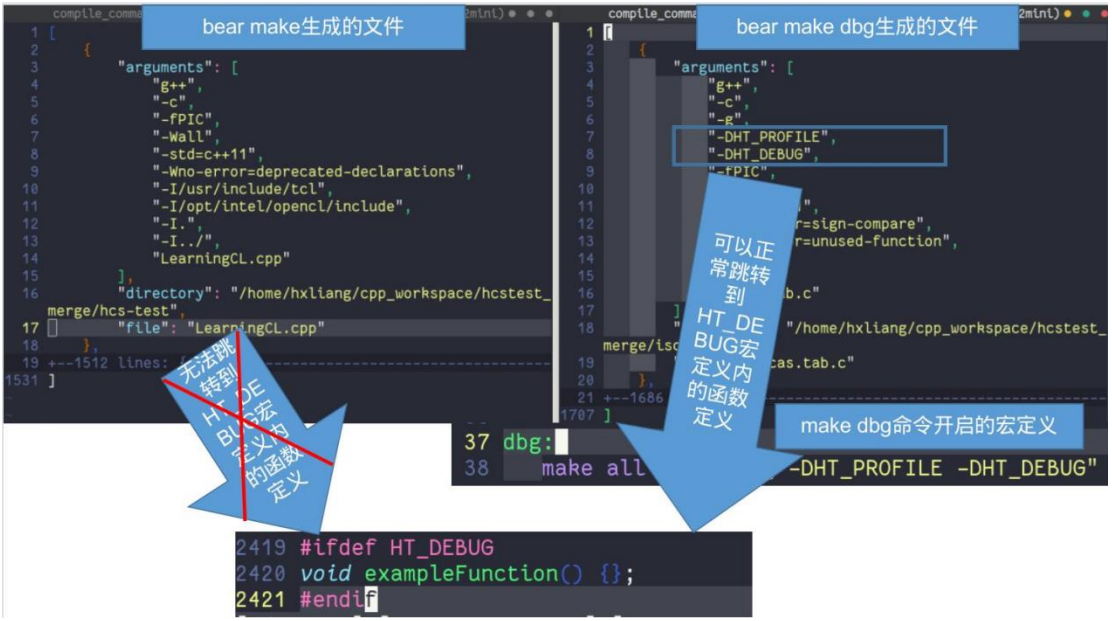


图 4-2

（2）在项目的根目录已有 compile_commands.json 文件前提下，切换 make 命令生成 compile_commands.json 文件，例如，从 make 命令切换到 make dbg 命令 或 从 make 命令切换到 make hot_flatten 命令，其步骤如下：先执行 make clean 删除所有由编译生成的文件，再执行 rm compile_commands.json .cache -rf，最后执行 bear make dbg/bear make hot_flatten。其中，.cache 文件夹为 clangd 程序根据项目信息自动生成的缓存，能够使代码跳转功能速度变快。.cache 文件夹和 compile_commands.json 文件都不应该被上传到 github 仓库中，因此，它们应该被添加进.gitignore 文件中。同时应该注意，这里切换 make 命令后之所以需要重新生成 compile_commands.json 文件，是因为不同编译命令所启动的 C/C++的不同宏定义内的代码是不同的（这里和 C/C++的条件编译有关，请大家自行搜索学习或请教他人。）。如果不重新生成 compile_commands.json 文件，那么我们是无法通过 clangd 扩展跳转进入部分宏定义内的函数，类和结构体等的定义。例子，如上图 4-2 所示。

（3）在项目的根目录已有 compile_commands.json 文件前提下，仅仅添加新的源码文件，那么只需根据之前执行的 bear 命令执行 bear --append <make 相关的命令>即可，例如，之前执行的 bear 命令是 bear make dbg，那么这次执行的 bear 命令应该是 bear --append make dbg。

（4）在项目的根目录已有 compile_commands.json 文件前提下，仅仅是修改了源码文件，无需执行 bear 命令，只需根据需求执行相关的 make 命令即可。

（5）如果觉得记住上面四个场景有点困难，那么只需要记住除了仅仅修改了源码文件无需执行 bear 命令外，其它情况都可以使用如下步骤：先执行 make clean，再执行 rm

compile_commands.json .cache -rf，最后执行 bear <make 相关的命令>。或者永远记住这两条命令即可，rm .cache -rf 和 bear --append <make 相关的命令>。

（6）如果觉得 bear 使用麻烦，可以使用 compiledb 代替，无论何种情况都只需要记住 compiledb make 命令即可。缺点是 compiledb 只支持 compiledb make -jn 和 compiledb make 命令，不支持其它 compiledb make <other arguments>命令，例如不支持 compiledb make dbg 命令，这样使用生成的 compile_commands.json 文件是空的。如果服务器没有 compiledb 程序，下载方法也简单，只需执行命令 python3 -m pip install compiledb 即可。

当完成上面的配置以后，就可以在 VSCode 正常打开源码文件，然后右击某些代码选择相关跳转功能进行跳转，也可以使用 VSCode 定义的快捷键跳转。同时，语法高亮和语法错误检测功能也会打开。

参考链接：https://zhuanlan.zhihu.com/p/398790625
https://blog.csdn.net/weixin_43862847/article/details/119274382

4.2 C/C++代码的格式化

clang-format 代码格式化程序是由 LLVM 项目提供的代码格式化程序，它能够根据.clang-format 文件，或者根据一些标准的开源项目的代码格式（LLVM, GNU, Google, Chromium, Microsoft, Mozilla, WebKit）对代码格式化。同时，它也可以通过注释标明哪部分代码可以被格式化，哪部分代码不可以被格式化，从而实现对整个进行文件格式化操作时的选择性格式化。例子如下图 4-3 所示。

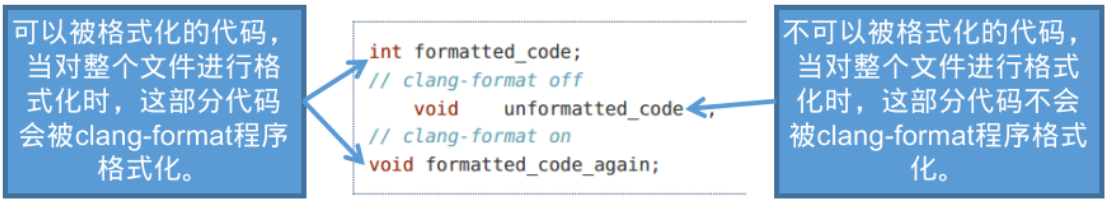


图 4-3

在我们的 C/C++项目使用 clang-format 程序对代码进行格式化时，请确保项目的工作空间的根目录拥有我们老师提供的.clang-format 文件，否则，禁止对代码进行格式化。同时，除了允许对自己写的代码进行格式化以外，禁止对项目的其它代码进行格式化。因此，本文作者建议，如果是自己新建的文件写代码，可以使用下面描述的针对整个文件的格式化操作；如果是自己在别人创建的文件写代码，只允许使用下面描述的针对代码片段的格式化操作。

clangd 扩展除了调用 clangd 程序提供代码高亮，代码定义跳转等功能外，还能够调用 clang-format 程序对代码进行格式化。它针对不同的场景提供了不同的代码格式化操作。

（1）针对整个文件的格式化操作：先打开需要被格式化的文件，接着直接在 VSCode 代码窗口右击空白处选择 Format Document 选项对整个文件进行格式化，或者使用 VSCode 提供的快捷键进行格式化。例子如下图 4-4 所示。

（2）针对代码片段的格式化操作的步骤：先打开需要被格式化的文件，接着使用鼠标左键选中要被格式化的代码片段，然后右击被选中的代码片段，最后选择 Format Selection 选项对被选中的代码片段进行格式化。当然，也可以是选中代码片段后直接使用 Format Selection 的快捷键实现代码片段的格式化操作。

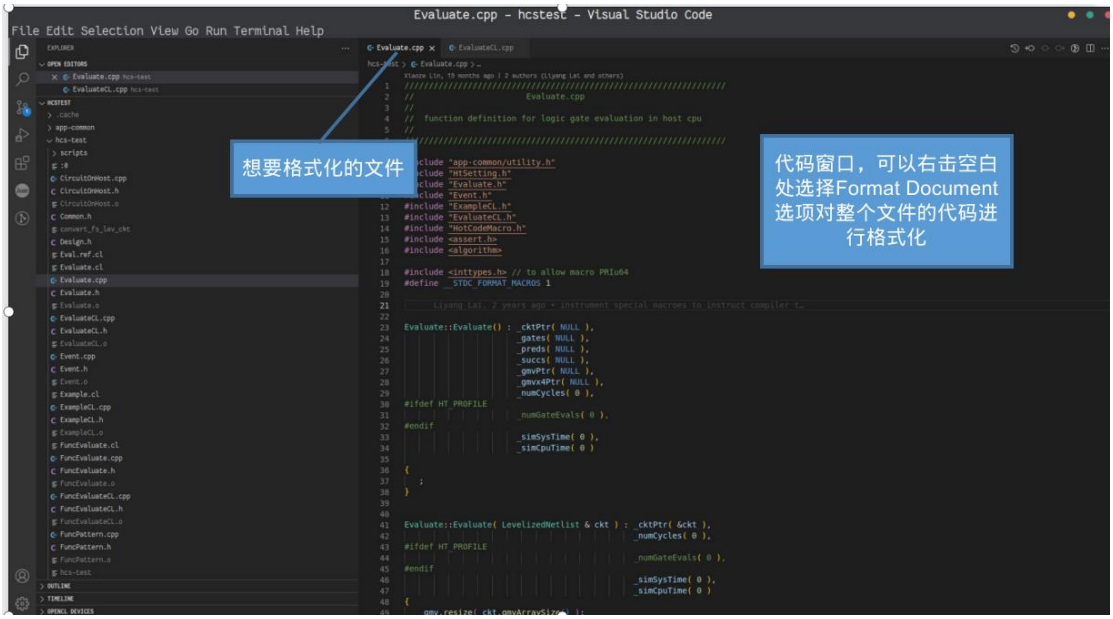


图 4-4

五，关闭 C/C++扩展的部分功能和 OpenCL 的 LSP

微软开发的 C/C++扩展提供的语法高亮，定义跳转等功能和 clangd 扩展提供的功能重复，而且速度比 clangd 扩展慢，因此，我们需要关闭其相关功能，仅保留其用于 Debug 的功能。OpenCL 扩展配置 LSP 太过麻烦和复杂，我们无需配置其 LSP，仅保留其语法高亮即可。

打开 Linux 终端，执行 gvim ~/.config/Code/User/settings.json 命令，并把如下内容添加到 settings.json 文件的{}中：

- "C_Cpp.codeFolding": "disabled",
- "C_Cpp.suggestSnippets": false,
- "C_Cpp.enhancedColorization": "disabled",
- "C_Cpp.autocomplete": "disabled",
- "C_Cpp.hover": "disabled",
- "C_Cpp.default.enableConfigurationSquiggles": false,
- "C_Cpp.configurationWarnings": "disabled",
- "C_Cpp.errorSquiggles": "disabled",
- "C_Cpp.intelliSenseEngine": "disabled",
- "C_Cpp.inlayHints.parameterNames.hideLeadingUnderscores": false,
- "C_Cpp.renameRequiresIdentifier": false,
- "C_Cpp.codeAnalysis.runAutomatically": false,
- "C_Cpp.formatting": "disabled",
- "C_Cpp.debugger.useBacktickCommandSubstitution": true,
- "C_Cpp.clang_format_sortIncludes": false,
- "C_Cpp.codeAnalysis.clangTidy.codeAction.showClear": "None",
- "OpenCL.server.enable": false,

六， Git

本章会简单介绍一些常用的 Git 命令，Git 扩展的简单操作以及.gitignore 文件。相关详细知识，请自行百度搜索。

6.1 配置 Git 全局配置

打开终端，执行如下命令，下面的命令的 YourName 请使用你的名字的拼音代替，例如，假设你的名字是李华华，那么拼音是 Huahua Li；YourEmail 则使用可以和你保持联系的邮箱代替，尽量别使用学校邮箱，因为，你毕业后，你的学校邮箱地址是会变成校友邮箱地址的，我们无法使用你原来提供的学校邮箱地址和你保持联系，你可以使用 QQ，163 或者 126 等邮箱：

- git config --global user.name "YourName"
- git config --global user.email "YourEmail"
- git config --global alias.logline "log --graph --abbrev-commit" #optional
- git config --global core.editor gvim #optional
- git config --global protocol.https.allow always #optional

- git config --global push.default "current" #optional

6.2 Git 常用命令介绍

请自行上网搜索学习。也可以通过下面的连接进去学习。https://www.runoob.com/git/git-basic-operations.html

6.3 VSCode 自带的 Git 图形化管理窗口以及 Git Graph

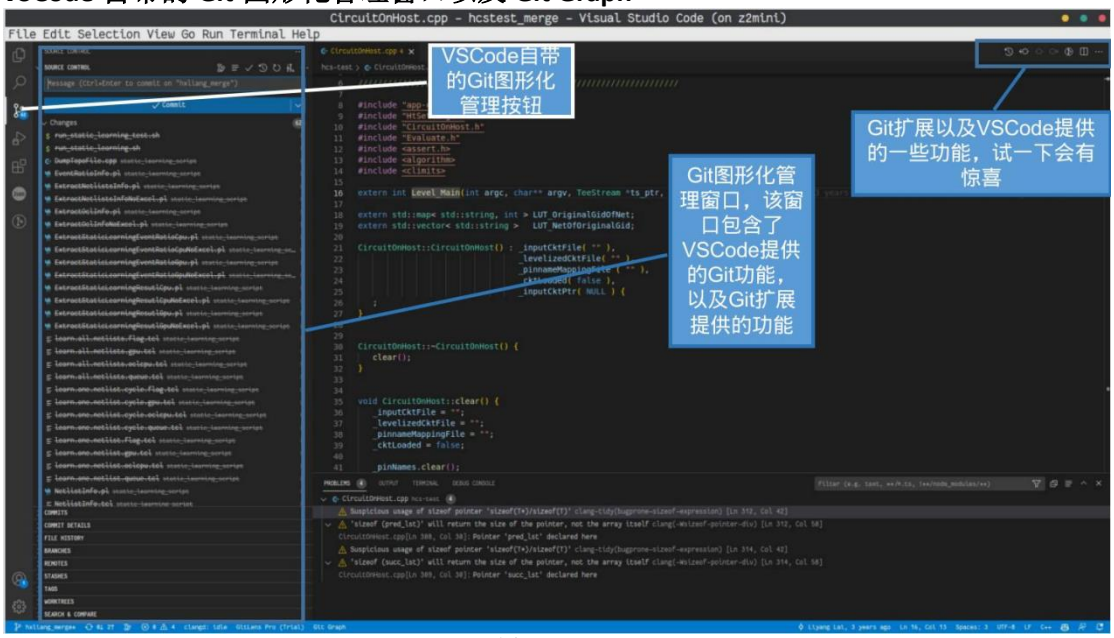


图 6-1

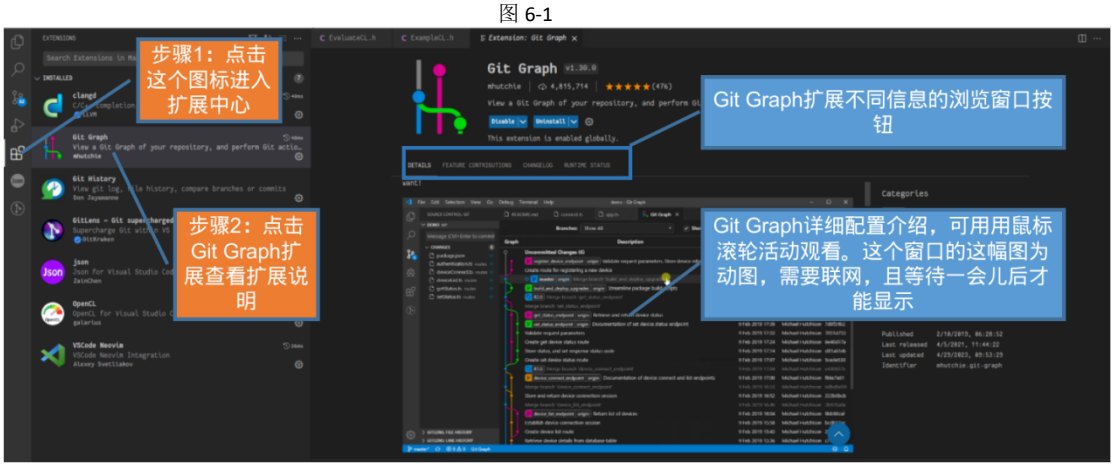


图 6-2

点击上图 6-1 所示的 VSCode 自带的 Git 图形化管理按钮，可以进入 Git 图形化管理窗口，该窗口是大部分 Git 扩展提供相关 Git 功能的窗口，上图 6-2 展示了如何查看 Git Graph 的相关操作的步骤。查看另外两个 Git 扩展的使用方法的步骤类似，这里不作介绍。

七，使用 VSCode 进行 Debug

.gdbinit 文件是 Gdb 工具启动时的初始化文件，需要我们自己进行一定的配置才能够人性化工作，因此我们需要自己配置这个文件，下面是本人的配置步骤。打开 Linux 终端，执行命令 gvim ~/.gdbinit，添加如下内容进.gdbinit 文件并保存：

- set print pretty on
- set print object on
- set print static-members on
- set print demangle on
- set print sevenbit-strings off
- set print array-indexes on
- set print vtbl on
- set confirm off
- set history filename ~/.cache/.gdbHistory

```
set history save on
set history size 333
python
import sys
sys.path.insert(0, '/opt/rh/devtoolset-7/root/usr/share/gdb/python')
from libstdcxx.v6.printers import register_libstdcxx_printers
register_libstdcxx_printers (None)
end
```

launch.json 文件是 VSCode 调用 Gdb 对 C/C++项目 Debug 的必不可少的文件，其内容基本差不多，下面是本人为我们的 hcs-test 项目配置 launch.json 文件的基本步骤。打开 Linux 终端，进入 C/C++项目的根目录，执行 code .命令。打开 VSCode 的终端窗口，确保 VSCode 的终端当前所在位置为 C/C++项目的根目录。如果项目的根目录没有.vscode 文件夹，那么在 VSCode 命令行终端窗口执行命令 mkdir .vscode，接着执行 gvim .vscode/launch.json 命令，并把下面内容复制粘贴到 launch.json 文件中并保存：

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "C/C++",
      "type": "cppdbg",
      "request": "launch",
      "program": "${workspaceFolder}/.exe",
      "args": [
        "",
        ""
      ],
      "stopAtEntry": true,
      "cwd": "${workspaceFolder}",
      "environment": [],
      "externalConsole": false,
      "MIMode": "gdb",
      "setupCommands": [
        {
          "description": "Enable pretty-printing for gdb",
          "text": "-enable-pretty-printing",
          "ignoreFailures": true
        }
      ],
      "breakpoints": {
        "exception": {
          "all": "Y"
        }
      }
    }
  ]
}
```

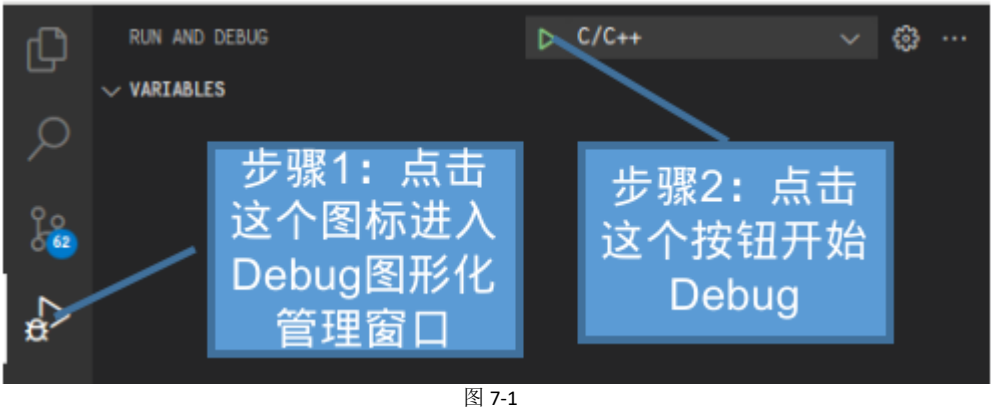


图 7-1

保存完毕后，点击 F5 键或者按上图 7-1 的步骤进入 Debug 模式。注意，在进入 Debug 模式前必须保证你已经编译生成的程序带有 Debug 信息，否则，无法对程序进行 Debug。对于 C/C++使用 gcc/g++命令进行编译必须添加-g 选项，对于我们的 hcs-test 项目则是编译时使用 make dbg 命令。在我们的项目中，建议在生成带 Debug 信息的程序前，先执行命令 make clean 删除编译生成的所有文件，再执行 make dbg 生成带 Debug 信息的程序，否则有可能无法对部分代码 Debug。不同管理工具或项目生成带 Debug 信息的程序的命令不一样，具体情况请咨询已经参与了项目一定时间的研发人员。更多详细信息可以搜索 Gdb 教程获取。

进入 Debug 模式后，VSCode 界面如下图 7-2 所示。现在，经过配置，你可以在 VSCode 里面一边使用代码跳转功能浏览代码，一边进行 Debug 了。添加断点的方式既可以在 Gdb 终端窗口处使用 Gdb 命令实现，也可以点击文件的某一行实现，具体的请大家多尝试。最后，建议大家学一下 Gdb 的相关命令和知识。

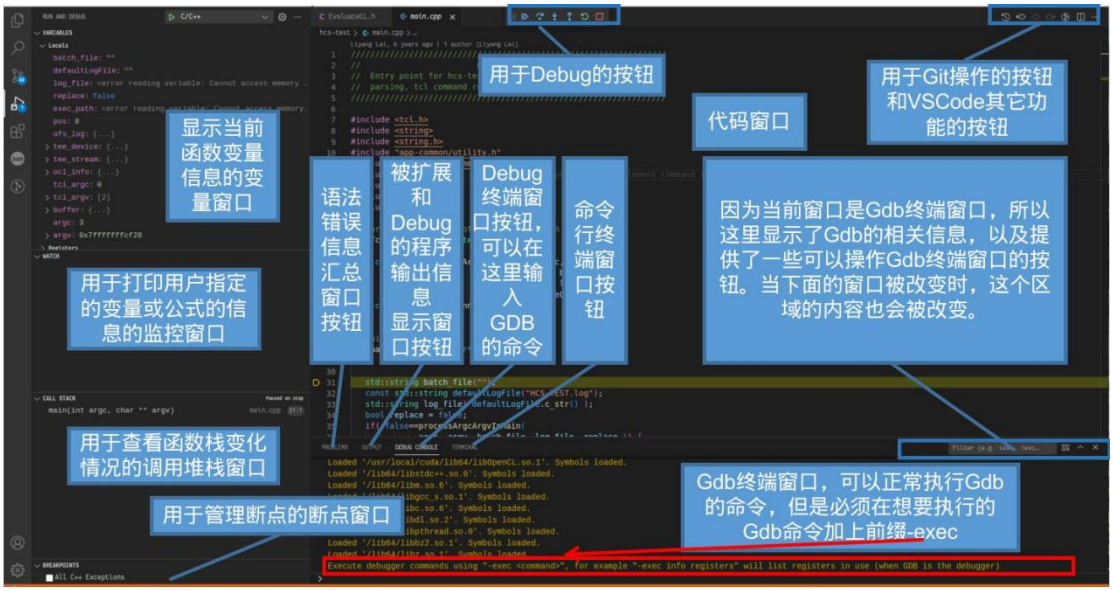


图 7-2

除了使用 Gdb 进行 Debug 外，还要学会使用 C/C++自带的 Debug 函数，宏以及 printf 函数结合 if 语句进行 Debug。特别是针对 OpenCL 语言的 Debug，必须学会使用 printf 函数结合 if 语句进行 Debug，因为目前没有针对 OpenCL 进行 Debug 的工具。

微软官方教程：https://code.visualstudio.com/docs/cpp/config-linux
https://code.visualstudio.com/docs/editor/debugging

八，用于查找文件或者捉取文件关键字的两个常用命令

虽然 VSCode 实现了代码跳转功能，但是，VSCode 并无法保证代码跳转功能一定能够跳转到目标位置。例如，当代码实现的函数存在多态时，有时 VSCode 并不一定能够跳转成功，因此要学会使用 grep 命令捉取某些文件的关键字，具体用法，请自行搜索，请教他人，以及使用 grep --help 或 man grep 命令查看。

除此以外，一个 C/C++项目的文件数量是庞大的，文件夹的深度也是难以估计的，此时，如果我们仅知道某个文件或文件夹的命字，而不知具体路径，我们难以通过人工的方式在该项目的目录中找到我们需要的目标，因此，我们应该学会使用 find 命令，具体用法，请自行搜索，

请教他人，以及使用 find --help 或 man find 命令查看。

多态是 C/C++的动态编译，也有静态编译的多态，大家可以上网搜一下。语法跳转这些功能一般针对通过 make，gcc/g++等命令进行编译的代码，通过编译命令编译的编译方式是静态编译。而对动态编译的代码，语法跳转功能支持有限。

九，使用 Neovim 作为 VSCode 编辑文件的后端（Optional）

Neovim 是 Vim 的一个分支，其操作模式和 Vim 保持一致。Neovim 抛弃了部分老旧的 Vimscript 语法，只兼容 Vim 百分之九十左右的配置。之所以在 VSCode 使用 Neovim 作为后端是因为 Alexey Svetliakov 等 Vim/Neovim 爱好者研发了支持调用 Neovim 作为后端的扩展，而没有研发支持调用 Vim 作为后端的扩展。同时，把 Neovim 作为 VSCode 的后端既能使用 VSCode 的扩展，也能使用 Neovim 的插件，同时还能使用 Neovim 的基础功能。

使用 Neovim 作为 VSCode 的后端的步骤如下：

（1）在 VSCode 扩展中心或扩展中心下载如图 9-1 所示的 VSCode Neovim 扩展。

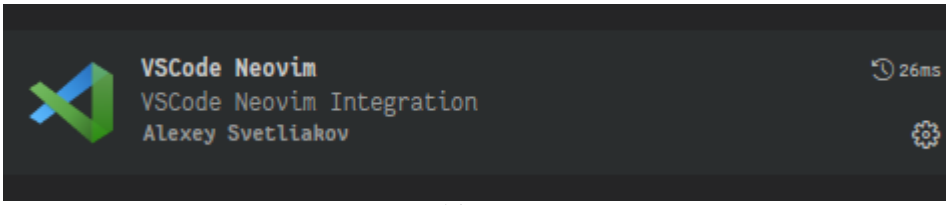


图 9-1

（2）打开 Linux 终端，执行命令 mkdir ~/.config/nvim -p 命令，接着执行 gvim ~/.config/nvim/init.vim 命令，并将如下内容添加到 init.vim 文件中：

```
" Tab 键的显示宽度
set tabstop=3
" 按下 Tab 键时输入的宽度
set softtabstop=3
" 把 Tab 字符用空格代替，和 tabstop 相关
set expandtab
" 设置自动缩进时的缩进长度
set shiftwidth=3
" 使用 VSCode Neovim 插件时需要的配置
if exists('g:vscode')
  " VSCode extension
  " 仅针对 VSCode 的配置
else
  " ordinary Neovim
  " 仅针对 Neovim 的配置
endif
```

最后执行 gvim ~/.config/Code/User/settings.json 命令，并将如下内容添加到 settings.json 文件的{}里面：

```
"vscode-neovim.neovimExecutablePaths.darwin": "/usr/local/bin/nvim",
"vscode-neovim.neovimInitVimPaths.darwin": "~/.config/nvim/init.vim",
```

Neovim 的教程除了本文所提到的教程，大家还可以通过下图 9-2 所示步骤查看更多教程。

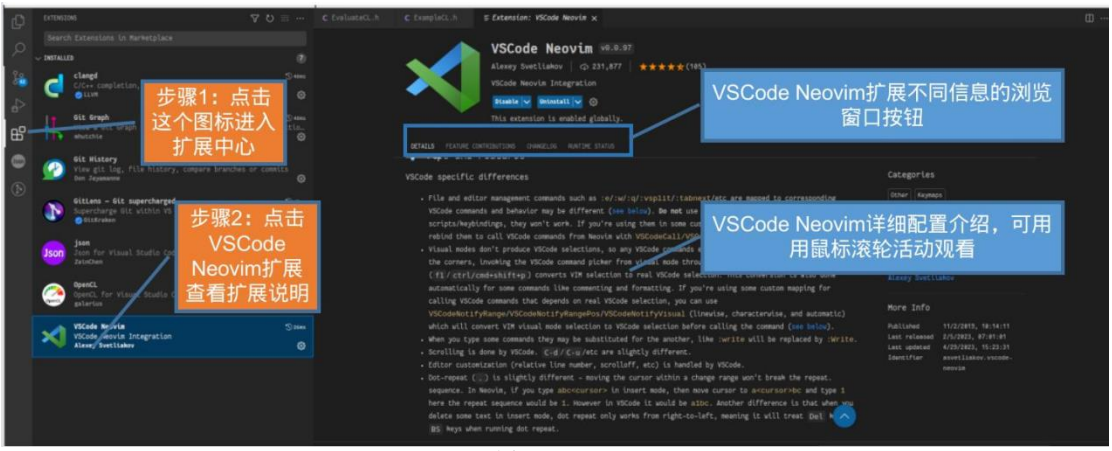


图 9-2

十，其它常用语言以及用途

Perl: 用于从大量 log 文件中提取数据。

Tcl: 绝大部分 EDA 工具都支持的语言，我们的项目程序也需要使用执行我们的程序提供的功能。

Bash Shell:用于更改个人环境变量，可以和 Tcl 语言配合调用程序批量完成任务。

十一， VSCode 扩展和 Linux 程序版本不匹配的解决办法

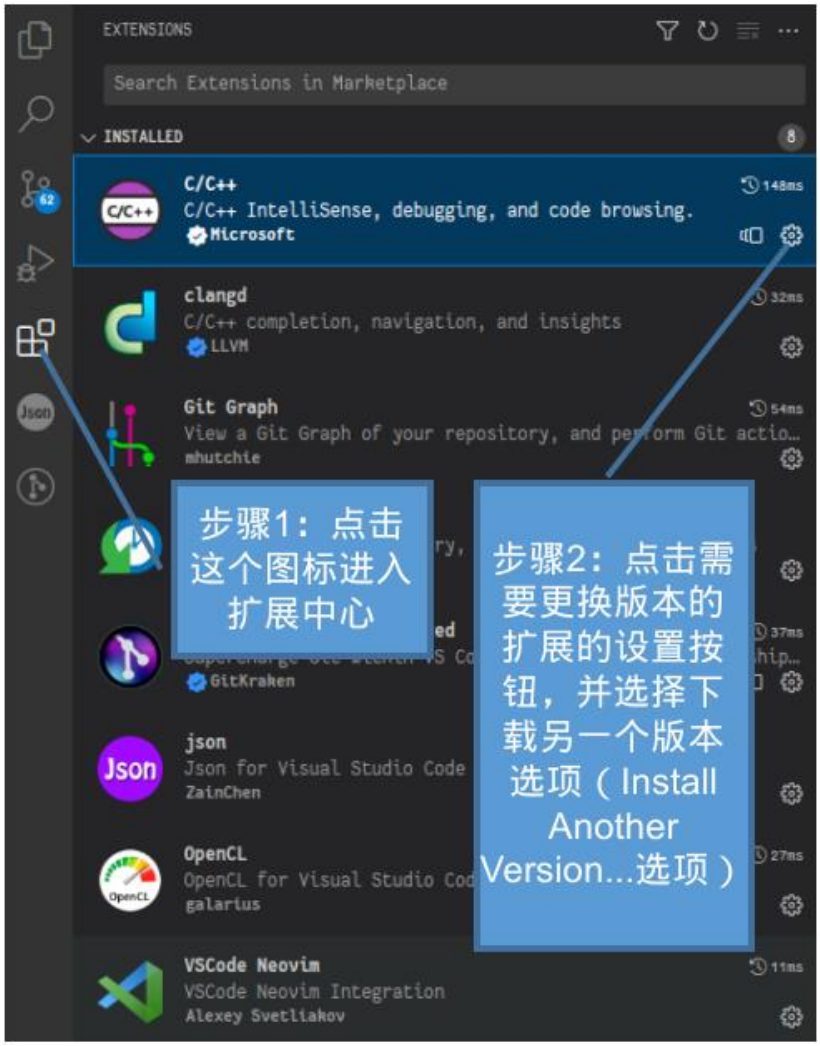


图 11-1

下载旧版本扩展。下载旧版本扩展有两个方法，一个是使用 VSCode 自带的设置自动下载如上图 11-1 所示，另一个方法是先在网上找到旧版本的扩展包（后缀为.vsix 的文件），然后手动安装，这个方法网上有，请大家有需要时自行搜索，这里不作详细介绍。

本人在写本文时使用到的扩展版本如下：

- C/C++：v1.14.5
- clangd：v0.1.24
- Git Graph：v1.30.0
- Git History：v0.6.20
- GitLens：v13.5.0
- json：v2.0.2
- OpenCL：v0.7.4
- VSCode Neovim：v0.0.97

十二， settings.json，launch.json，.gdbinit 和 init.vim 文件说明

因为 Json 文件不支持注释，因此这里只能以图添加注释的形式说明最终的 Json 文件具体配置内容。下图 12-1，12-2，12-3 和 12-4 分别解释了 settings.json，launch.json，.gdbinit 和 init.vim 文件里面主要内容的说明。如果需要了解更多关于 settings.json 和 launch.json 文件的内容请自行搜索或者到微软的官网查看。如果需要了解更多的.gdbinit 文件的内容，请自行搜索或者到 Gdb 官网查看。如果需要了解更多的 init.vim 文件的内容，请自行搜索或者到 Neovim 官网查看。

settings.json 文件是 VSCode 配置其自身设置以及所有扩展的设置的文件，在这里我们可以通过代码的方式快速设置 VSCode 以及其扩展。我们对 VSCode 以及扩展进行的修改也会被 VSCode 写进 settings.json 文件中。

launch.json 文件是 VSCode 调用 Gdb 对 C/C++项目 Debug 的必不可少的文件。

gdbinit 文件是 Gdb 工具启动时的初始化文件，我们可以在这里随意地配置 Gdb。

init.vim 文件是 Neovim 启动时的初始化文件，我们可以在这里随意地配置 Neovim。

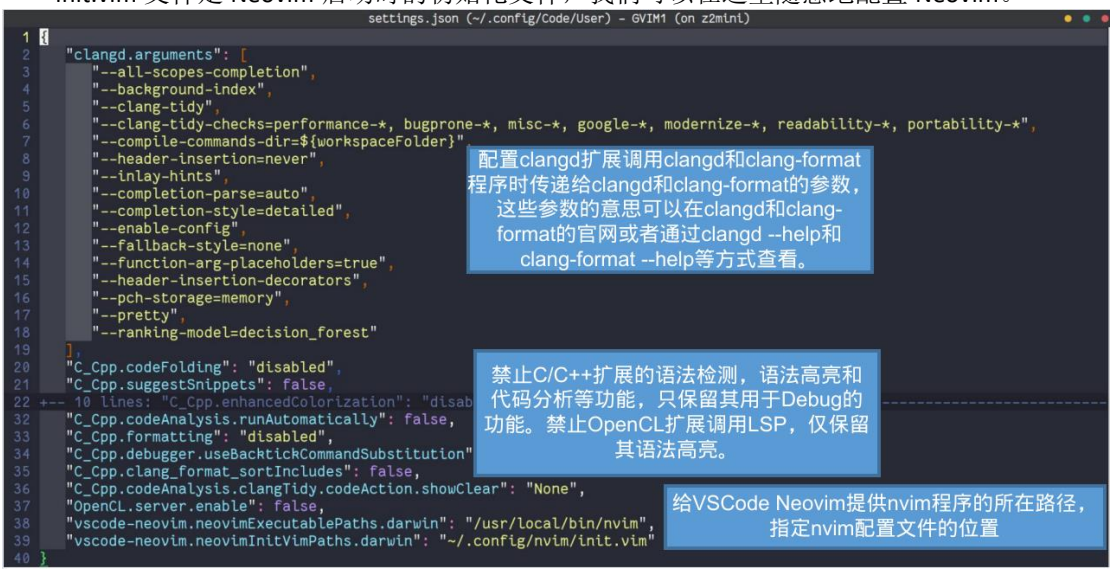


图 12-1

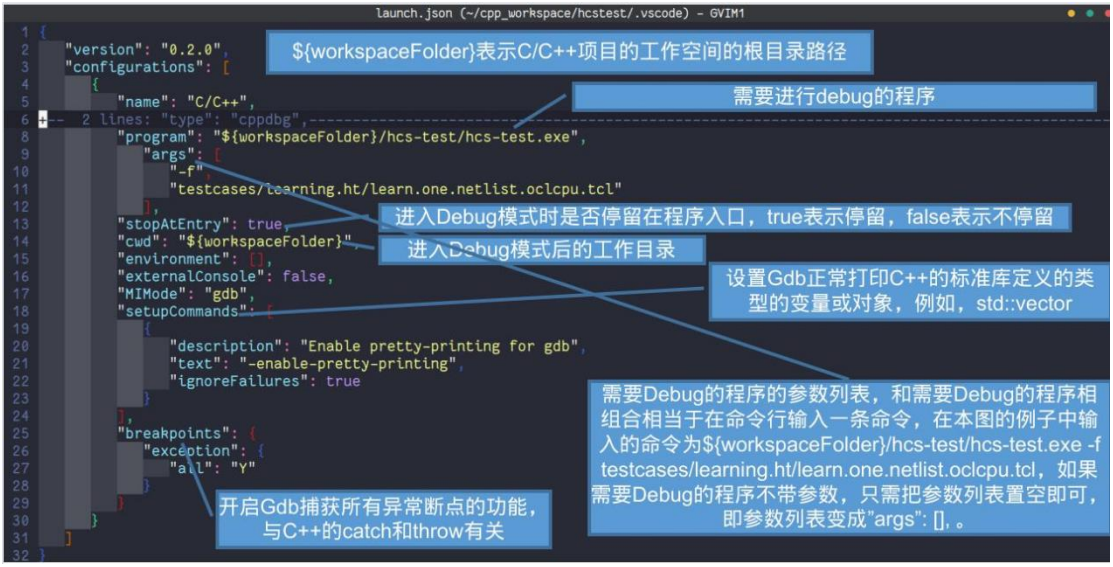


图 12-2

图 12-3

图 12-4