

VScode 调试 Unix V6++ 应用程序

-by Anxi

下文介绍在 VScode 中调试 Unix V6++ 应用程序的方法。

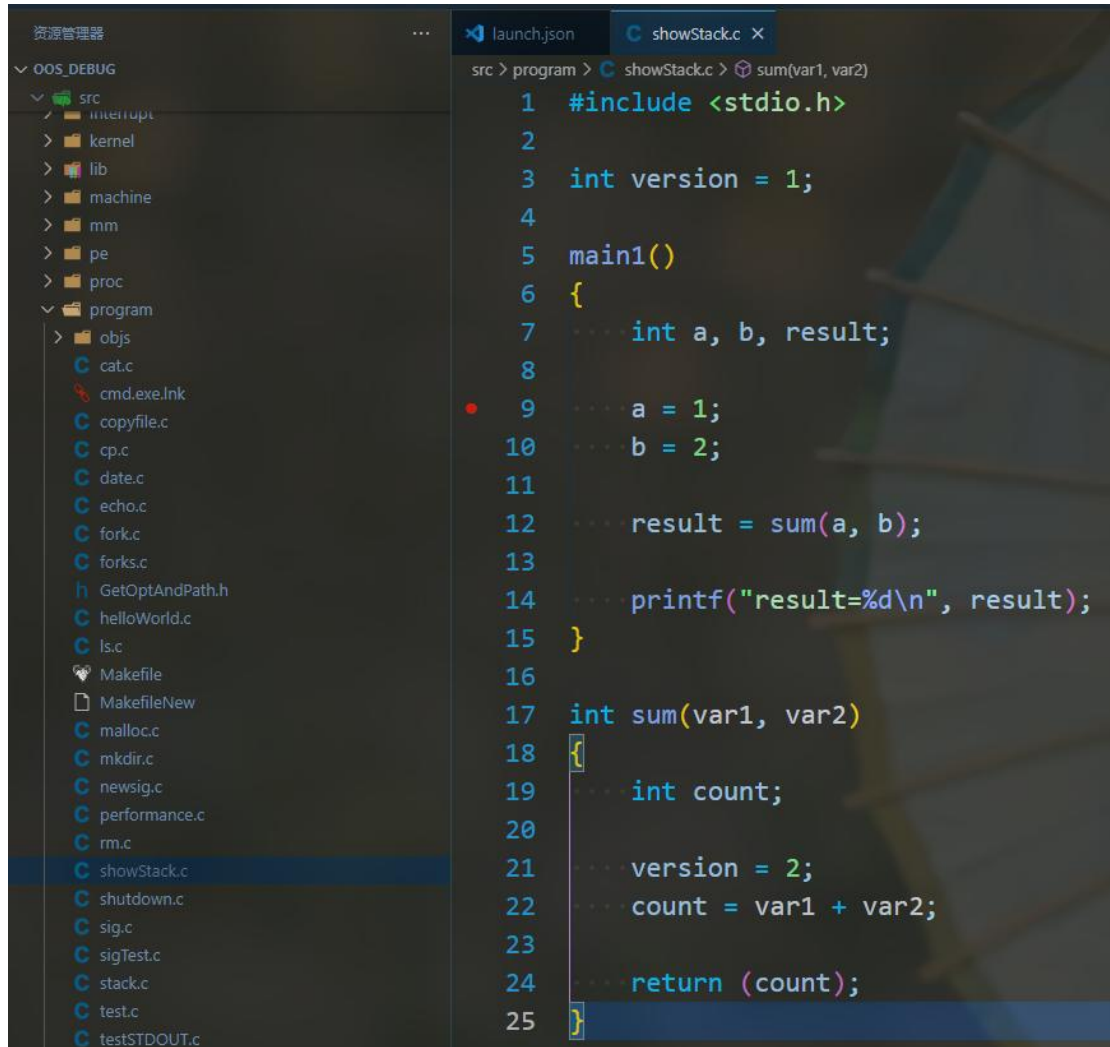
VScode 相较于 Eclipse，调试配置较为简单，但是对于反汇编的查看以及单步调试反汇编程序较为复杂。下面按步骤给出具体的使用方法：

首先需要配置 launch.json 文件，使得当前调试的路径指向需要调试的应用程序。

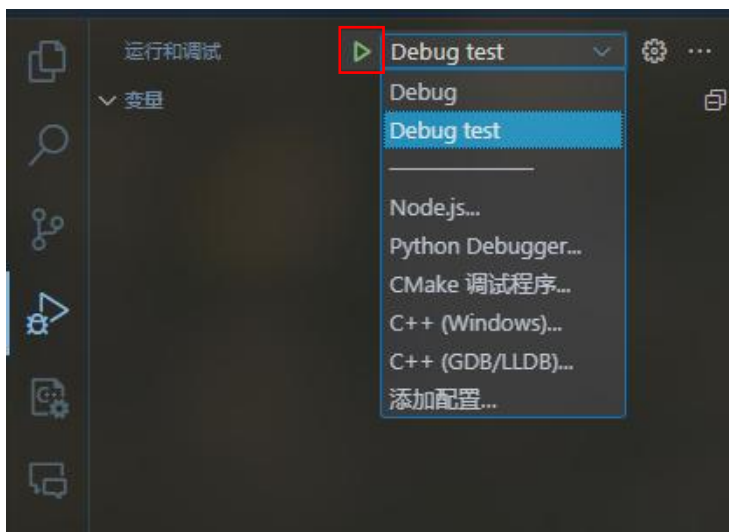
```
{
  "name": "Debug test",
  "type": "cppdbg",
  "request": "launch",
  "cwd": "${workspaceFolder}/src",
  "miDebuggerServerAddress": "localhost:1234",
  "miDebuggerPath": "E:/Unix_V6++/MinGW/bin/gdb.exe",
  "setupCommands": [
    {
      "description": "为 gdb 启用整齐打印",
      "text": "-enable-pretty-printing",
      "ignoreFailures": true
    },
    {
      "description": "将反汇编风格设置为 att",
      "text": "-gdb-set disassembly-flavor att",
      "ignoreFailures": true
    }
  ],
  "program": "${workspaceFolder}/src/program/objs/showStack.exe",
  "args": [],
  "externalConsole": false,
  "MIMode": "gdb",
  "preLaunchTask": "OOS Run" // 前置任务：启动 Bochs, 运行 OOS
}
```

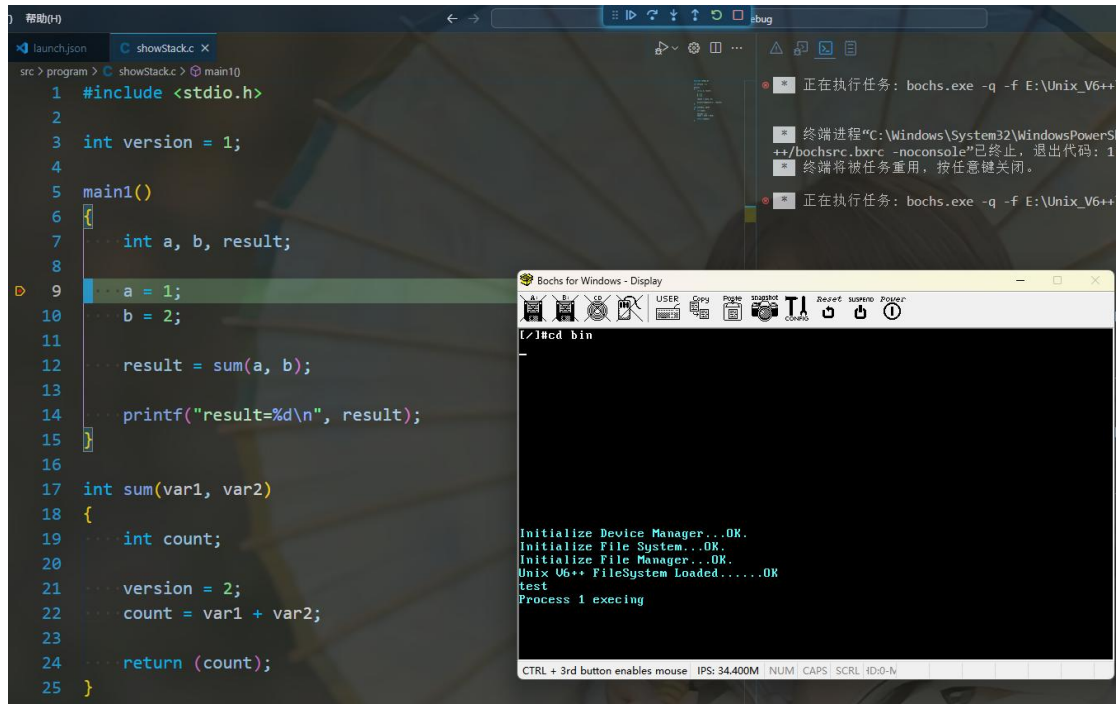
相较于给出的 launch.json 文件，这里修改了“program”指向的路径为需要调试的程序，即“showStack.exe”；其次，还添加了两条对 GDB 的设置命令，以方便查看反汇编代码。这里的反汇编风格设置为了早期 Unix 常用的 AT&T 风格，更加适合 Unix V6++ 系统。

修改完该文件后，即可对“oos/src/program/showStack.c”打断点并调试。

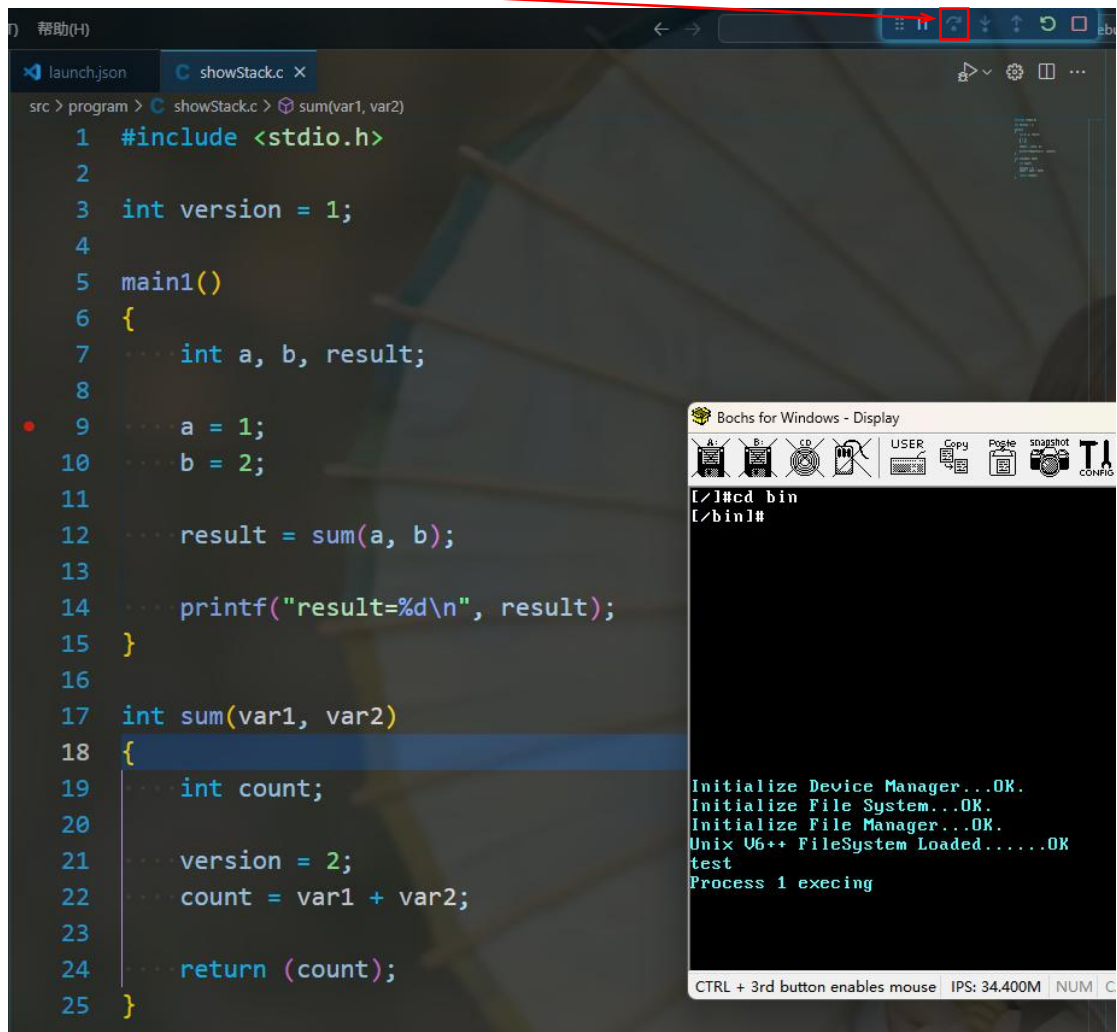


上图在 main1 函数中打上了断点，将“运行和调试”选项卡中的调试配置选为上面修改后的“Debug test”，然后按“F5”开始调试。按下调试后的行为和 debug 内核时一致。

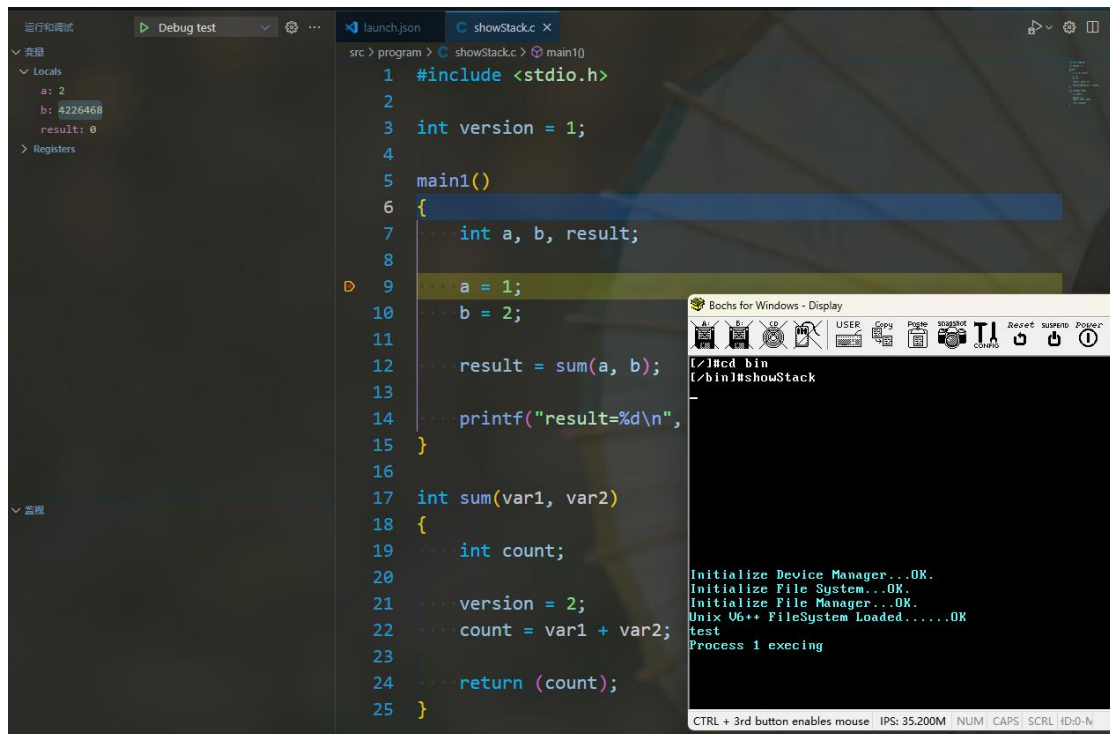




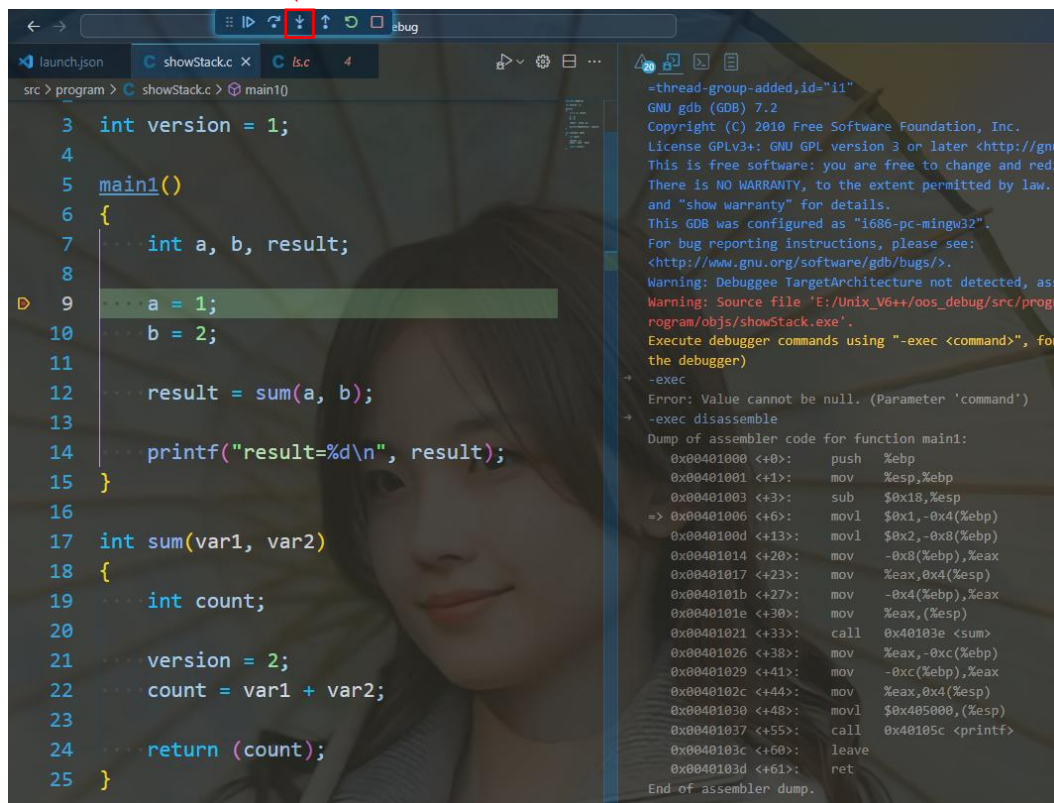
注意：在 VScode 中也会出现和 Eclipse 一样的停顿问题，采取相同的措施也可解决，即一直按“逐过程”或“F10”键直到“cd bin”运行结束：



然后我们就可以输入“showStack”命令，对“showStack.c”程序进行调试：



可以看到，输入“showStack”命令后正确进入了断点。在进入断点后，先按一次单步调试键（F11），这时并不会直接进入下一句的执行，而是在等待一会后重新在 `a = 1` 处断开。此时，在调试控制台中输入“-exec disassemble”命令，可以正常输出反汇编代码（注意，没有先按单步调试键的话，执行该命令输出的是错误的汇编码！）。




```

-exec disassemble
Dump of assembler code for function main1:
   0x00401000 <+0>:    push    %ebp
   0x00401001 <+1>:    mov     %esp,%ebp
   0x00401003 <+3>:    sub     $0x18,%esp
=> 0x00401006 <+6>:    movl    $0x1,-0x4(%ebp)
   0x0040100d <+13>:   movl    $0x2,-0x8(%ebp)
   0x00401014 <+20>:   mov     -0x8(%ebp),%eax
   0x00401017 <+23>:   mov     %eax,0x4(%esp)
   0x0040101b <+27>:   mov     -0x4(%ebp),%eax
   0x0040101e <+30>:   mov     %eax,(%esp)
   0x00401021 <+33>:   call    0x40103e <sum>
   0x00401026 <+38>:   mov     %eax,-0xc(%ebp)
   0x00401029 <+41>:   mov     -0xc(%ebp),%eax
   0x0040102c <+44>:   mov     %eax,0x4(%esp)
   0x00401030 <+48>:   movl    $0x405000,(%esp)
   0x00401037 <+55>:   call    0x40105c <printf>
   0x0040103c <+60>:   leave
   0x0040103d <+61>:   ret
End of assembler dump.
-exec si

```

```

-exec disassemble
Dump of assembler code for function main1:
   0x00401000 <+0>:    push    %ebp
   0x00401001 <+1>:    mov     %esp,%ebp
   0x00401003 <+3>:    sub     $0x18,%esp
   0x00401006 <+6>:    movl    $0x1,-0x4(%ebp)
=> 0x0040100d <+13>:   movl    $0x2,-0x8(%ebp)
   0x00401014 <+20>:   mov     -0x8(%ebp),%eax
   0x00401017 <+23>:   mov     %eax,0x4(%esp)
   0x0040101b <+27>:   mov     -0x4(%ebp),%eax
   0x0040101e <+30>:   mov     %eax,(%esp)
   0x00401021 <+33>:   call    0x40103e <sum>
   0x00401026 <+38>:   mov     %eax,-0xc(%ebp)
   0x00401029 <+41>:   mov     -0xc(%ebp),%eax
   0x0040102c <+44>:   mov     %eax,0x4(%esp)
   0x00401030 <+48>:   movl    $0x405000,(%esp)
   0x00401037 <+55>:   call    0x40105c <printf>
   0x0040103c <+60>:   leave
   0x0040103d <+61>:   ret
End of assembler dump.

```

图中执行“-exec disassemble”命令后输出的反汇编码中，“=>”指向的是正在执行的反汇编代码，可以看到“-exec si”命令确实是对反汇编码进行的单步运行操作。

在 GDB 中，“si”命令是单步执行（进入调用的函数内部），“ni”是逐过程（不进

入调用函数内部)。

此外，在调试控制台中也可以很方便的查看寄存器和变量的值，具体方法如下图：

```
-exec info registers
eax          0x9      9
ecx          0x407dac 4226476
edx          0x407014 4222996
ebx          0x407014 4222996
esp          0x7ffd34 0x7ffd34
ebp          0x7ffd4c 0x7ffd4c
esi          0x0      0
edi          0xffac   65452
eip          0x401011 0x401011 <main1+17>
eflags       0x246    [ PF ZF IF ]
cs           0x1b     27
ss           0x23     35
ds           0x23     35
es           0x23     35
fs           0x0      0
gs           0x0      0
-exec print a
$1 = 2
-exec print &a
$2 = (int *) 0x7ffd48
```

GDB 还有许多强大的功能可以通过该调试控制台完成，这里不再演示。