

---

# **Vimspector:** 最强 **Vim** 调试插件

孔俊

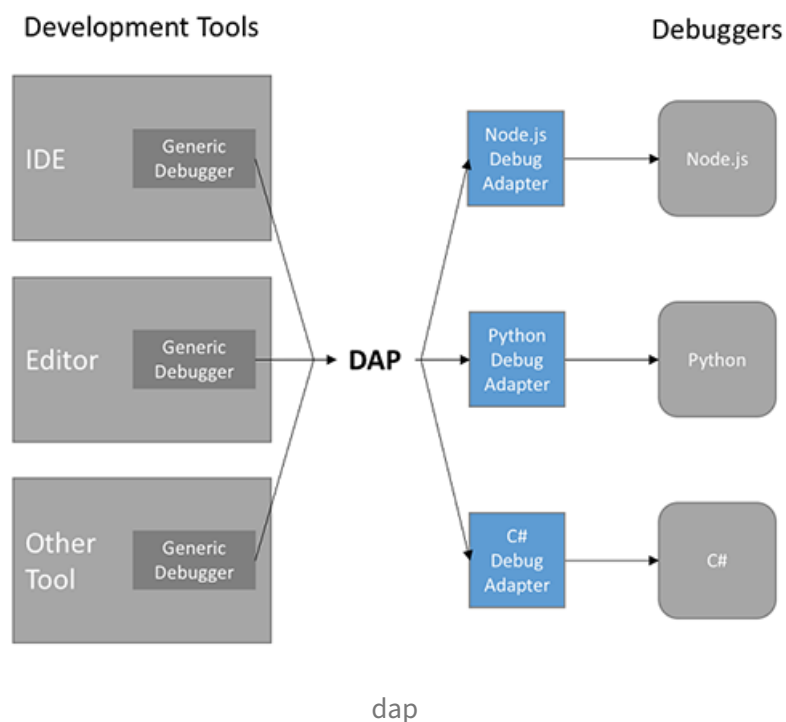
2020-05-05

## Contents

<b>1</b>	<b>依赖</b>	<b>2</b>
<b>2</b>	<b>安装</b>	<b>2</b>
2.1	vimspector . . . . .	2
2.2	调试适配器 . . . . .	3
<b>3</b>	<b>配置</b>	<b>3</b>
3.1	调试适配器配置 . . . . .	4
3.2	调试会话配置 . . . . .	5
3.3	配置选项 . . . . .	5
<b>4</b>	<b>变量</b>	<b>6</b>
4.1	预定义变量 . . . . .	6
4.2	自定义变量 . . . . .	6
4.3	默认值 . . . . .	7
4.4	类型转换 . . . . .	7
<b>5</b>	<b>多配置共存</b>	<b>8</b>
<b>6</b>	<b>断点</b>	<b>8</b>
<b>7</b>	<b>示例</b>	<b>9</b>
7.1	调试 Vim . . . . .	9
7.2	调试 qemu-riscv64 中的 OS 内核 . . . . .	10
7.3	我自己的配置 . . . . .	12
<b>8</b>	<b>使用技巧</b>	<b>14</b>
8.1	快捷键 . . . . .	14
8.2	修改 UI . . . . .	16
8.3	Ballon-Eval . . . . .	16
8.4	修改变量打印的格式 . . . . .	16
<b>9</b>	<b>题外话：GDB 前端推荐</b>	<b>16</b>

**vimspector**是一个基于 *DAP(debug adapter protocol)* 的 Vim 多语言调试插件，理论上能够支持所有支持语言（只要有对应的 DAP）。这个插件仍在实验阶段，可能会有各种 bug，但是对 C/C++、Python 等流行的语言已经进行了充分的测试。

这篇文章以调试 C/C++ 程序为例，介绍 vimspector 的配置与使用。



## 1 依赖

- 带 Python3.6+ 支持的 Vim 8.2 或更高版本
- 带 Python3.6+ 支持的 Neovim-0.4.3 或更高版本（最好是 Nightly 版本）

由于 vimspector 的作者主要在 GNU/Linux 上使用 Vim 开发，因此 Vimspector 作者 puremourning 明确表示在 vimspector 的充分测试并稳定后才会提供对 Neovim 的完整支持（见 issue [coc.nvim: Debug Adapter Protocol Support #322](#)），因此目前对于 Neovim 和 Windows 的支持都处于实验阶段。我个人建议在 Vim 中使用本插件。

## 2 安装

### 2.1 vimspector

使用 `vim-plug` 安装：

```
1 Plug 'puremourning/vimspector'
```

使用 `dein.vim` 安装:

```
1 call dein#add('puremourning/vimspector')
```

## 2.2 调试适配器

最新版的 `vimspector` 可以在 Vim 中通过命令:`VimspectorInstall`安装调试适配器, 按Tab键可以补全。

也可以使用安装脚本安装, 进入`vimspector`的安装目录, 执行:

```
1 ./install_gadget.py <language-name>
```

`install_gadget.py`会自动下载<language-name>所需的调试适配器并进行相应配置, `--help`可以查看`vimspector`所支持的全部语言。

以在 Linux 环境上打开 C/C++ 支持为例:

```
1 ./install_gadget.py --enable-c
```

`vimspector`会自动下载微软开发的调试适配器`cpptools-linux.vsix`到`your-vimspector-path/gadgets/linux/download/vscode-cpptools/0.27.0/`中。如果是在 `mac` 上, `linux`会被改成`mac`。

如果下载速度过慢, 可以自己下载好放置在上面提到的目录中, 然后再执行以上命令。

## 3 配置

Vimspector 使用 json 作为配置文件的格式, 每个配置都是一个 json 对象。

Vimpector 有两类配置:

- 调试适配器的配置
  - 如何启动或连接到调试适配器
  - 如何 `attach` 到某进程
  - 如何设置远程调试
- 调试会话的配置
  - 使用哪个调试适配器
  - `launch` 或 `attach` 到进程

- 是否预先设置断点，在何处设置断点

这两类配置可以对应多个配置文件，**vimspector** 会将多个配置文件中的信息合并成一个配置。

### 3.1 调试适配器配置

调试适配器的这个配置在打开 **vimspector** 对某语言的支持时就已经自动设置好了，存放在 **your-path-to-vimspector/gadgets/linux/.gadgets.json** 中。

比如在我的设备上，**.gadgets.json** 内容如下：

```
1 {
2   "adapters": {
3     "vscode-cpptools": {
4       "attach": {
5         "pidProperty": "processId",
6         "pidSelect": "ask"
7       },
8       "command": [
9         "${gadgetDir}/vscode-cpptools/debugAdapters/OpenDebugAD7"
10      ],
11       "name": "cppdbg"
12     }
13   }
14 }
```

其中变量 **\${gadgetDir}** 代表着存放 **.gadgets.json** 的目录。除此之外，**vimspector** 还定义其他预定义变量，并提供了自定义和用户输入变量内容的功能，以便我们编写比较通用的配置文件。

调试适配器的配置还可以存在于其他配置文件中，**vimspector** 读取一系列配置文件，生成 **adapters** 对象。

调试适配器的配置可以存在于以下文件中：

1. **our-path-to-vimspector/gadgets/<os>/gadgets.json**：这个文件由 **install\_gadget.py** 自动生成，用户不应该修改它。
2. **your-path-to-vimspector/gadgets/<os>/gadgets.d/\*.json**：这些文件是用户自定义的。
3. 在 Vim 工作目录向父目录递归搜索到的第一个 **.gadgets.json**。
4. **.vimspector.json** 中定义的 **adapters**。

编号代表配置文件的优先级，编号越大优先级越高，高优先级的配置文件将覆盖低优先级的配置文件中的 **adapters**。

在我的机器上没有 **your-path-to-vimspector/gadgets/<os>/gadgets.d** 目录，可能是需要自己创建。

不进行远程调试的情况下不太需要修改默认的调试适配器配置。我一般没有进行远程调试的需求，没有实际使用过 `vimspector` 的远程调试功能（虽然这个功能是 `vimspector` 重点支持的），因此不介绍调试适配器的配置。

### 3.2 调试会话配置

项目的调试会话的文件位于以下两个位置：

1. `<your-path-to-vimspector>/configurations/<os>/<filetype>/*.json`
2. 项目根目录中的 `.vimspector.json`

每当打开一个新的调试会话时，`vimspector` 都会在当前目录向父目录递归搜索，如果查找到了 `.vimspector.json`，则使用其中的配置，并将其所在的目录设定为项目根目录；如果未查找到，则使用 `vimspector` 安装目录中的配置文件，将打开的文件的目录设置为项目根目录。

修改了 `.vimspector.json` 后不需要重启 Vim 就可以使用最新配置。

### 3.3 配置选项

`vimspector.json` 中只能包含一个对象，其中包含以下子对象：

- `adapters`：调试适配器配置，如果不是进行远程调试，一般不需要设置
- `configurations`：调试程序时的配置

`configurations` 主要包含以下以下字段：

- `adapter`：使用的调试配置器名称，该名称必须出现在 `adapters` 块或其他调试适配器配置中。
- `variables`：用户定义的变量
- `configuration`：配置名，如 `configuration1`
- `remote-request, remote-cmdLine`：远程调试使用

其中 `adapter` 和 `configuration` 是必须的。

`configuration` 需要包含的字段和使用的 DAP 有关，我使用 `vscode-cpptools.configuration` 必须包含以下字段：

- `request`：调试的类型，`launch` 或 `attach`
- `type`：`cpptools`(GDB/LLDB) 或 `cppvsdbg`(Visual Studio Windows debugger)

除了以上的选项，还可以设置程序路径、参数、环境变量、调试器路径等，更详细的信息可以查看 `vscode-cpptools` 文档 [launch-json-reference](#)。

上面的选项构成了 `vimspector` 配置文件的主体框架，完整的选项参考 [vimspector.schema.json](#)。

## 4 变量

Vimspector 提供了比较灵活的变量定义功能，可以方便的自定义配置。

### 4.1 预定义变量

- `${dollar}` - has the value `$`, can be used to enter a literal dollar
- `$$` - a literal dollar
- `${workspaceRoot}` - the path of the folder where `.vimspector.json` was found
- `${workspaceFolder}` - the path of the folder where `.vimspector.json` was found
- `${gadgetDir}` - path to the OS-specific gadget dir ( `<vimspector home>/gadgets/<OS>` )
- `${file}` - the current opened file
- `${relativeFile}` - the current opened file relative to `workspaceRoot`
- `${fileBasename}` - the current opened file's `basename`
- `${fileBasenameNoExtension}` - the current opened file's `basename` with no file extension
- `${fileDirname}` - the current opened file's `dirname`
- `${fileExtname}` - the current opened file's `extension`
- `${cwd}` - the current working directory of the active window on launch
- `${unusedLocalPort}` - an unused local TCP port

predefined variables

### 4.2 自定义变量

自定义变量要在起作用的配置中定义，并置于`variables`块中，如下面的`GDBServerVersion`和`SomeOtherVariable`。

```
1 {
2   "configurations": {
3     "some-configuration": {
4       "variables": {
5         "GDBServerVersion": {
6           "shell": [ "/path/to/my/scripts/get-gdbserver-version" ],
7           "env": {
8             "SOME_ENV_VAR": "Value used when running above command"
9           }
10        },
11        "SomeOtherVariable": "some value"
12      }
13    }
14  }
15 }
```

可以通过 `shell` 设置变量的值，代码块中的 `GDBServerVersion` 的值就是 `shell` 脚本 `/path/to/my/scripts/get-gdbserver-version` 的输出。

经我实验，自定义变量似乎不能够依赖自定义变量，也不可以在 `shell` 中使用预定义变量，可能之后的版本会实现这些功能。

也可以从用于输入中获取变量的值。只要 `vimspector` 发现了未定义的变量，就会在运行时提示用户输入变量值。`Vimspector` 支持 `splat` 运算符（不清楚中文叫什么），语法为 `*${Variable}`，可以将一个变量拓展开。

以上两个特性结合在一起可以实现很灵活的配置，最典型的运例子是程序参数的传递。`Vimspector` 调试的程序的参数以数组的形式传递，在配置文件中将 `args` 设置为一个在运行时用户输入的变量，就可以模拟命令行的效果。

```
1  "args": [ "${CommandLineArgs}" ]
```

在运行时 `vimspector` 会要求用户输入值，如果用户输入 1、2、3，`args` 就会被拓展成 `["1", "2", "3"]`。

### 4.3 默认值

可以为变量提供默认值，`${variableName:default value}`。在 `${}` 中引用变量时，`}` 要通过 `\` 转义，即将 `}` 写为 `\\}`。

```
1  {
2    "configuration": {
3      "program": "${script:${file}\\}"
4    }
5  }
```

`program` 默认设置为变量 `file` 的值。

### 4.4 类型转换

`vimspector` 介绍的的用户输入都字符串，有时会出现类型和用户期望的不同的情况。比如，用户为布尔类型的变量 `StopOnEntry` 输入 `true`，`vimspector` 接收到字符串 `"true"`，并将它赋给变量，这样出现了类型不一致的情况。

```
1  {
2    "configuration": {
3      "stopAtEntry": "${StopOnEntry}"
4    }
5  }
```



在字段后添加`#json`可以将接收到的字符串转换成 json 里的类型。如果变量以`#json`结尾，需要在字段尾部添加`#s`以告知 Vimspector 这个变量以`#json`结尾，而不是进行类型转换。

```
1  {
2      "configuration": {
3          "stopAtEntry#json": "${StopOnEntry}"
4      }
5  }
```

这样，用户输入`true`，vimspector 接收到字符串`"true"`，然后再将它解析为布尔类型的`true`。

## 5 多配置共存

可以在`.vimspector.vim` 中写多个配置，在启动 vimspector 时再选择使用的配置。这样的话，可以将所有自己需要的配置写入到一个文件，在创建项目时复制到项目中。

和配置选择有关的字段有两个：

**autoselect**: 布尔类型。在只有一个配置可用时，是否自动选择它。

**default**: 布尔类型。当启动时用户没有选择配置时，使用本配置。

## 6 断点

可以在配置中提前打好断点，比如在程序入口点暂停（通常是`main()`），在抛出异常时暂停等，暂时不支持在配置中，打函数、代码行断点

**stopAtEntry**: 布尔类型。是否在程序入口点暂停。

**cpp\_throw**: 在抛出异常时暂停

**cpp\_catch**: 在捕获异常时暂停

```
1  {
2      "example":{
3          "stopAtEntry": true,
4          "MIMode": "gdb",           // 使用 GDB
5          "breakpointers": {
6              "exception": {
7                  "cpp_throw": "Y",
8                  "cpp_catch": "N"
9              }
10         }
11     }
12 }
```

目前 Vimspector 对断点的支持还比较有限，仅支持打断点，不能够方便的编辑、禁用、使能断点，将来这些功能会实现在断点窗口中，参考[\[Feature Request\]: Breakpoints window #10](#)。

## 7 示例

Vimspector 的配置其实很简单，但是纸上谈兵有些难以理解。这里将给出几个可以实际使用的配置，如调试 Vim、调试 qemu 模拟器中的 OS 内核。

### 7.1 调试 Vim

vimspector 文档中给出的调试 vim 的配置：

```

1 {
2   "configurations": {
3     "Vim - run a test": {
4       "adapter": "vscode-cpptools",           // 配置名
5       "configuration": {                     // 使用的调试适配器
6         "type": "cppdbg",                   // 具体的配置
7         "request": "launch",                // 调试器类型:
8         "program": "${workspaceRoot}/src/vim", // 调试类型: launch
9         "args": [                           // 带有调试信息的可
10            "-f",                             // 程序的参数, 一个
11            "-u", "unix.vim",                 json 数组
12            "-U", "NONE",
13            "--noplugin",
14            "--not-a-term",
15            "-S", "runtest.vim",
16            "${Test}.vim"                     // 未定义的变量, 用
17            ],                                户输入
18        "cwd": "${workspaceRoot}/src/testdir", // 当前工作目录
19        "environment": [                     // 环境变量
20          { "name": "VIMRUNTIME", "value": "${workspaceRoot}/runtime" }
21        ],
22        "externalConsole": true,             // 是否使用外部终端
23        "stopAtEntry": true,                 // 是否在程序入口点
24        "MIMode": "lldb",                    // 暂停
25        "logging": {                         // 使用 LLDB 作为调
26          "engineLogging": false              // 试器
27        },                                   // 调试适配器的输出
28      },                                     // 是否打印调试适配
29    },
30  }
31 }

```

## 7.2 调试 qemu-riscv64 中的 OS 内核

用以下命令启动 qemu 模拟器, 让它监听 `localhost:1234` 等待 GDB 连接:

```

1  qemu-system-riscv64 \
2      -machine virt \
3      -s -S \
4      -nographic \
5      -bios default \
6      -device loader,file=kernel.img,addr=0x80200000

```

因此，我们要配置 vimspector 让 RISCv 架构的 GDB 连接到 `localhost:1234`。

```
1 {
2     "configurations": {
3         "qemu-riscv64-oslab": {
4             "adapter": "vscode-cpptools",
5             "variables": {
6                 "kernelBin": "kernel.bin,
// 带有调试信息的内核
// 可执行文件
7                 "riscvGDB": "/usr/local/bin/riscv64-unknown-elf-gdb"
// GDB 路径
8             },
9             "configuration": {
10                 "type": "cppdbg",
11                 "request": "launch",
12                 "program": "${kernelBin}",
13                 "cwd": "${workspaceRoot}",
14                 "environment": [],
15                 "externalConsole": true,
16                 "stopAtEntry": true,
17                 "MIMode": "gdb",
// 使用 GDB
18                 "miDebuggerPath": "${riscvGDB}",
// GDB 路径为 ${riscvGDB}
19                 "setupCommands": [
// 设置 GDB 初
// 始化命令，相当于 gdbinit
20                 {
21                     "description": "Enable pretty-printing for gdb"
, // 描述，不会被 GDB 使用
22                     "text": "set architecture riscv",
// 命令
23                     "ignoreFailures": false
// 是否忽略错误
24                 },
25                 {
26                     "description": "Connect gdbserver within qemu",
27                     "text": "target remote localhost:1234",
28                     "ignoreFailures": false
29                 }
30             ]
31         }
32     }
33 }
34 }
```

上面的配置依赖 `vscode-cpptools`，不支持其他的调试适配器。

### 7.3 我自己的配置

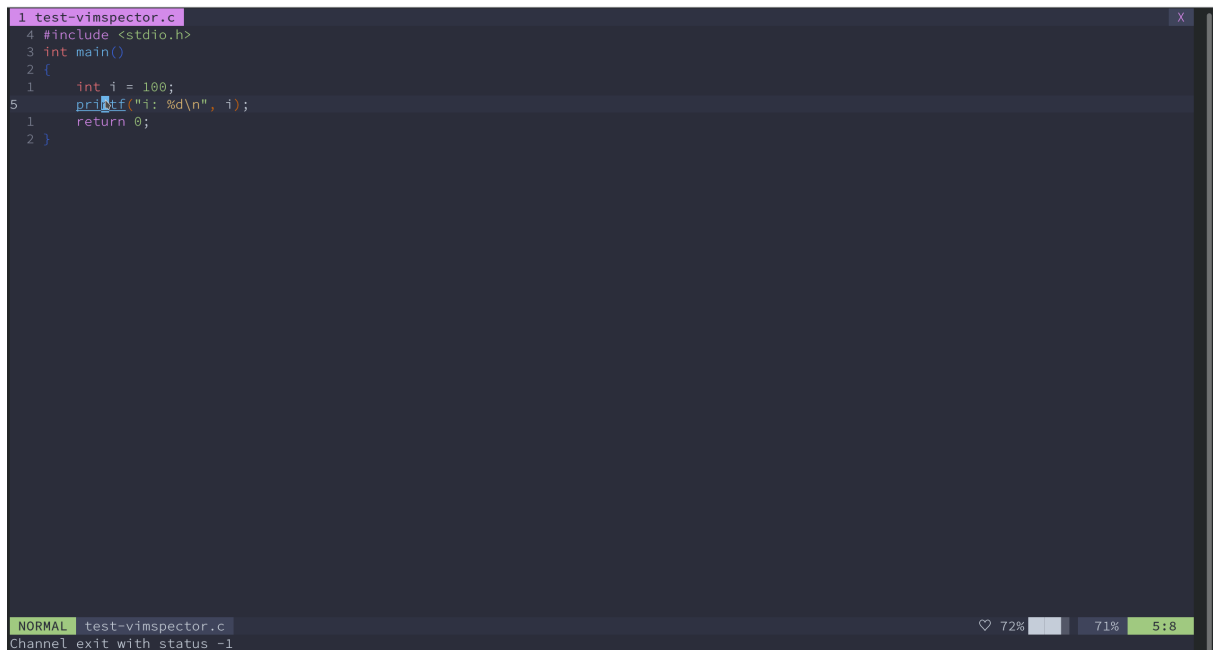
我将我可能使用的配置写入到一个文件，这样不需要重复编写，在启动 `vimspector` 选择即可。

```

1  {
2      "configurations": {
3
4          "qemu-riscv64-oslab": {
5              // ...
6          },
7          "launch-current-file": {
8              "adapter": "vscode-cpptools",
9              "configuration": {
10                 "default": true,
11                 "type": "cppdbg",
12                 "request": "launch",
13                 "program": "${fileDirname}/${fileBasenameNoExtension}",
14                 "args": ["*${ProgramArgs}"], // 用户输入
15                 "cwd": "${workspaceRoot}",
16                 "environment": [],
17                 "externalConsole": true,
18                 "stopAtEntry": true,
19                 "MIMode": "gdb",
20                 "breakpointers": {
21                     "exception": {
22                         "cpp_throw": "Y", // 抛出异常时暂停
23                         "cpp_catch": "N" // 捕获时不暂停
24                     }
25                 }
26             },
27          },
28          "launch-current-project": {
29              "adapter": "vscode-cpptools",
30              "configuration": {
31                 "variables": {
32                     "ProgramName": {
33                         "shell": ["basename ", "${workspaceRoot}"] // 无法正确执行，需要用户输入
34                     },
35                     "ProgramPath": "${workspaceRoot}/_builds/${ProgramName}"
36                 },
37                 "type": "cppdbg",
38                 "request": "launch",
39                 "program": "${workspaceRoot}/_builds/${ProgramName}", 13
40                 "args": ["*${ProgramArgs}"],
41                 "cwd": "${workspaceRoot}",
42                 "environment": [],
43                 "externalConsole": true,
44                 "stopAtEntry": true,
45                 "MIMode": "gdb",

```

上面介绍过了，不再赘述



vimspector-screenshot

## 8 使用技巧

### 8.1 快捷键

Vimspector 预设了 **vscode mode** 和 **human mode** 两套键盘映射（快捷键）。

但是对 C/C++、Python 等流行的语言已经进行了充分的测试。开启 **vscode mode**:

```
1 let g:vimspector_enable_mappings = 'VISUAL_STUDIO'
```

开启 **human mode**:

```
1 let g:vimspector_enable_mappings = 'HUMAN'
```

这两个套快捷键都要用到 **F11** 和 **F12**，往往会和终端快捷键冲突，比如 **F11** 是最大化终端，**F12** 是弹出 **guake** 之类的下拉框终端，建议终端用户重新定义快捷键。参考快捷键：

Key	Function	API
F5	When debugging, continue. Otherwise start debugging.	<code>vimspector#Continue()</code>
F3	Stop debugging.	<code>vimspector#Stop()</code>
F4	Restart debugging with the same configuration.	<code>vimspector#Restart()</code>
F6	Pause debuggee.	<code>vimspector#Pause()</code>
F9	Toggle line breakpoint on the current line.	<code>vimspector#ToggleBreakpoint()</code>
<leader>F9	Toggle conditional line breakpoint on the current line.	<code>vimspector#ToggleBreakpoint( { trigger expr, hit count expr } )</code>
F8	Add a function breakpoint for the expression under cursor	<code>vimspector#AddFunctionBreakpoint( '&lt;cexpr&gt;' )</code>
F10	Step Over	<code>vimspector#StepOver()</code>
F11	Step Into	<code>vimspector#StepInto()</code>
F12	Step out of current function scope	<code>vimspector#StepOut()</code>

## mappings

```

1  nnoremap <silent> <F1> :call vimspector#Stop()<CR>
2  nnoremap <silent> <F2> :call vimspector#Restart()<CR>
3  nnoremap <silent> <F3> :call vimspector#Continue()<CR>
4  nnoremap <silent> <F4> :call vimspector#Pause()<CR>
5  nnoremap <silent> <F5> :call vimspector#RunToCursor()<CR>
6  nnoremap <silent> <F6> :call vimspector#ToggleBreakpoint()<CR>
7  nnoremap <silent> <Leader><F6> :call vimspector#ListBreakpoints()<CR>
8  nnoremap <silent> <F7> :call <SID>toggle_conditional_breakpoint()<CR>
9  nnoremap <silent> <F8> :call vimspector#StepOver()<CR>
10 nnoremap <silent> <F9> :call vimspector#StepInto()<CR>
11 nnoremap <silent> <F10> :call vimspector#StepOut()<CR>
12
13 function! s:toggle_conditional_breakpoint()
14     let l:condition = trim(input("Condition: "))
15     if empty(l:condition)
16         return
17     endif
18     let l:count = trim(input("Count: "))
19     if empty(l:count)
20         let l:count = 1
21     else
22         let l:count = str2nr(l:count)
23     endif
24     call vimspector#ToggleBreakpoint({'condition': l:condition, '
        hitCondition': l:count})
25 endfunction

```



## 8.2 修改 UI

Vimspector 的 UI 是针对宽屏设计的，对于笔记本屏幕可能不太友好，最主要的问题是 console 窗口挤占了源代码窗口的空间，可以在启动时关闭 console 窗口，需要时再使用 `:VimspectorShowOutput Console` 显示。

```
1 augroup MyVimspectorUICustomistaion
2   autocmd User VimspectorUICreated call <SID>vimspector_custom_ui()
3 augroup END
4
5 " Custom Vimspector UI
6 " close console window to maximise source code window
7 function s:vimspector_custom_ui()
8   if !getwinvar(g:vimspector_session_windows.output, '&hidden')
9     let l:winid = win_getid()
10    let l:cursor = getcurpos()
11    call win_gotoid(g:vimspector_session_windows.output)
12    :quit
13    call win_gotoid(l:winid)
14    call setpos('.', l:cursor)
15  endif
16 endfunction
```

调整 UI 的具体细节可以参考文档。

## 8.3 Ballon-Eval

当鼠标悬浮在变量上时，Vimspector 会自动打印变量的类型与值，这个功能依赖于 `ballon-eval` (`help`)，仅支持 GVim，但 Vim 最好也开启 `ballon-eval` 支持。如果不使用 GVim，可以使用以下映射手动查看变量的值。

```
1 nmap <Leader>di <Plug>VimspectorBalloonEval
2 xmap <Leader>di <Plug>VimspectorBalloonEval
```

## 8.4 修改变量打印的格式

在 `watch` 窗口中输入变量名时在后面加上 `,format` 可以修改打印格式，目前使用 `vscode-cpptools` 似乎只能使用 `,x`（十六进制）和 `,o`（八进制）。

## 9 题外话：GDB 前端推荐

参考 [韦易笑](#) 的文章 [终端调试哪家强](#)，尝试了几种 C/C++ 调试方案，得到以下结论：

裸 GDB ==> cgdb ==> Vimspector/VSCode ==> gdbgui

其中 Vimspector 和 VSCode 均使用 `vscode-cpptools`，个人认为在能力上没有太大区别。`gdbgui` 是一个基于浏览器的 GDB 前端，能力应该是上述几种方案中最强的。`gdbgui` 有以下几个突出特性：- 不需要配置 `gdbgui` 不需要配置，只需要像直接使用 `gdb` 一样输入命令即可，如 `gdbgui -g 'gdb program -x gdbinit'`。- 兼顾 GUI 和命令行 在 `gdbgui` 中可以直接在 `gdb` 命令行中输入命令，并且 GUI 会响应 `gdb` 命令。比如在命令行中打了断点，GUI 会立刻显示出来。- 图形化显示数据结构 GDB 可以图形化显示链表和树

Vimspector 仍在实验阶段，部分重要特性还没有实现，如果需要更加强大的调试功能，可以考虑 `gdbgui`。