使用 GDB 调试 C API 扩展和 CPython 内部代码

本文档介绍了如何将 Python GDB 扩展 `python-gdb.py` 与 GDB 调试器一起使用以调试 CPython 扩展以及 CPython 解释器本身。

当调试低层级问题如崩溃或死锁时，低层级的调试器如 GDB 适合被用来诊断和修正错误。 在默认情况下，GDB（或其任一种前端）并不支持 CPython 解释器专属的高层级信息。

`python-gdb.py` 扩展可向 GDB 添加 CPython 解释器信息。 该扩展能协助对当前执行的 Python 函数栈进行内省。 当给定一个由 `PyObject*` 指针代表的 Python 对象时，该扩展将展示对象的类型和值。

开发 CPython 扩展或处理 CPython 中用 C 语言编写的部分的开发人员可以通过本文档学习如何将 `python-gdb.py` 扩展与 GDB 一起使用。

> **备注：** 本文档假定你已熟悉 GDB 和 CPython C API 的基础知识。 它对来自 devguide 和 Python wiki 的内容进行了整合。

前提条件

你需要有：

- GDB 7 或更高的版本。 （对于较低版本的 GDB，请参阅 Python 3.11 或更低版本源代码中的 `Misc/gdbinit`。）
- 针对 Python 和你正在调试的任何扩展的 GDB 兼容调试信息。
- `python-gdb.py` 扩展。

此扩展与 Python 一起构建，但可能单独发布或根本不发布。 下面，我们将以几个常见系统为例进行说明。 请注意即使这些说明与你的系统相匹配，它们也可能已经过时。

使用从源代码构建的 Python 进行设置

当你从源代码构建 CPython 时，调试信息应当是可用的，并且构建应当在你的代码库根目录中添加一个 `python-gdb.py` 文件。

要激活支持，你必须将包含 `python-gdb.py` 的目录添加到 GDB 的 "auto-load-safe-path" 中。如果你没有这样做，较新版本的 GDB 会打印一个警告来说明如何执行此操作。

> **备注：** 如果你没有看到针对你的 GDB 版本的说明，请将以下内容放到你的配置文件中 (`~/.gdbinit` 或 `~/.config/gdb/gdbinit`):
>
> ```
> add-auto-load-safe-path /path/to/cpython
> ```

你还可以添加多个路径，以 `:` 分隔。

针对 Linux 发行版的 Python 设置

大多数 Linux 系统会在名为 `python-debuginfo`、`python-dbg` 或类似的包中提供系统 Python 的调试信息。 例如：

- Fedora：

```
sudo dnf install gdb
sudo dnf debuginfo-install python3
```

- Ubuntu：

```
sudo apt install gdb python3-dbg
```

在一些最新的 Linux 系统上，GDB 可以使用 *debuginfod* 自动下载调试符号。 不过，这并不会安装 `python-gdb.py` 扩展；你通常需要单独安装调试信息包。

使用调试构建和开发模式

为了方便调试，你可能需要：

- 使用 Python 的 调试构建版。 （当从源代码构建时，使用 `configure --with-pydebug`。 在 Linux 发行版上，安装并运行 `python-debug` 或 `python-dbg` 之类的包，如果有的话。）
- 使用运行时 开发模式 (`-X dev`)。

两者都将启用额外的断言并禁用某些优化。 有时这会隐藏你想要查找的程序错误，但大多数情况下它们都能使调试过程更简单。

使用 python-gdb 扩展

当该扩展被加载时，它将提供两个主要特性：Python 值的美化打印，以及附加的命令。

美化打印

这是当此扩展被启用时 GDB 回溯信息的显示效果（截取部分）：

```
#0  0x000000000041a6b1 in PyObject_Malloc (nbytes=Cannot access memory at address
) at Objects/obmalloc.c:748
#1  0x000000000041b7c0 in _PyObject_DebugMallocApi (id=111 'o', nbytes=24) at Obje
#2  0x000000000041b717 in _PyObject_DebugMalloc (nbytes=24) at Objects/obmalloc.c:
#3  0x000000000044060a in _PyUnicode_New (length=11) at Objects/unicodeobject.c:34
#4  0x00000000004466aa in PyUnicodeUCS2_DecodeUTF8Stateful (s=0x5c2b8d "__lltrace_
    0x0) at Objects/unicodeobject.c:2531
#5  0x0000000000446647 in PyUnicodeUCS2_DecodeUTF8 (s=0x5c2b8d "__lltrace__", size
    at Objects/unicodeobject.c:2495
#6  0x000000000440d1b in PyUnicodeUCS2_FromStringAndSize (u=0x5c2b8d "__lltrace__
    at Objects/unicodeobject.c:551
#7  0x000000000440d94 in PyUnicodeUCS2_FromString (u=0x5c2b8d "__lltrace__") at O
#8  0x000000000584abd in PyDict_GetItemString (v=
```

```
    {'Yuck': <type at remote 0xad4730>, '__builtins__': <module at remote 0x7ffff7
    0x5c2b8d "__lltrace__") at Objects/dictobject.c:2171
```

请注意传给 `PyDict_GetItemString` 的字典参数被显示为其 `repr()`，而非不透明的 `PyObject *` 指针。

该扩展通过为类型 `PyObject *` 的值提供自定义的打印例程来发挥作用。 如果你需要访问一个对象的低层级细节，则要将原值投射为适当类型的指针。 例如:

```
(gdb) p globals
$1 = {'__builtins__': <module at remote 0x7ffff7fb1868>, '__name__':
'__main__', 'ctypes': <module at remote 0x7ffff7f14360>, '__doc__': None,
'__package__': None}

(gdb) p *(PyDictObject*)globals
$2 = {ob_refcnt = 3, ob_type = 0x3dbdf85820, ma_fill = 5, ma_used = 5,
ma_mask = 7, ma_table = 0x63d0f8, ma_lookup = 0x3dbdc7ea70
<lookdict_string>, ma_smalltable = {{me_hash = 7065186196740147912,
me_key = '__builtins__', me_value = <module at remote 0x7ffff7fb1868>},
{me_hash = -368181376027291943, me_key = '__name__',
me_value ='__main__'}, {me_hash = 0, me_key = 0x0, me_value = 0x0},
{me_hash = 0, me_key = 0x0, me_value = 0x0},
{me_hash = -9177857982131165996, me_key = 'ctypes',
me_value = <module at remote 0x7ffff7f14360>},
{me_hash = -8518757509529533123, me_key = '__doc__', me_value = None},
{me_hash = 0, me_key = 0x0, me_value = 0x0}, {
  me_hash = 6614918939584953775, me_key = '__package__', me_value = None}}}
```

请注意美化打印并不会实际调用 `repr()`。 对于基本类型，它将尝试尽量匹配其结果。

一个可能令人困惑的地方是某些类型的自定义打印效果很像是 GDB 针对标准类型的内置打印形式。 例如，针对 Python `int` (PyLongObject*) 的美化打印表示形式与机器层级上常规的整数并无区别:

```
(gdb) p some_machine_integer
$3 = 42

(gdb) p some_python_integer
$4 = 42
```

内部结构可通过投射到 PyLongObject* 来揭示:

```
(gdb) p *(PyLongObject*)some_python_integer
$5 = {ob_base = {ob_base = {ob_refcnt = 8, ob_type = 0x3dad39f5e0}, ob_size = 1},
ob_digit = {42}}
```

类似的困惑也可能发生于 `str` 类型，这里的输出看起来很像 gdb 针对 `char *` 的内置打印形式:

```
(gdb) p ptr_to_python_str
$6 = '__builtins__'
```

针对 `str` 实例的美化打印默认使用单引号（就像 Python 字符串的 `repr` 一样）而针对 `char *` 值的标准打印形式使用双引号并且包含一个十六进制的地址:

```
(gdb) p ptr_to_char_star
$7 = 0x6d72c0 "hello world"
```

同样地，该实现细节可通过投射为 PyUnicodeObject* 来显示:

```
(gdb) p *(PyUnicodeObject*)$6
$8 = {ob_base = {ob_refcnt = 33, ob_type = 0x3dad3a95a0}, length = 12,
str = 0x7ffff2128500, hash = 7065186196740147912, state = 1, defenc = 0x0}
```

py-list

该扩展添加了一个 `py-list` 命令，它将列出选定的线程中当前帧的 Python 源代码（如果存在）。 当前行将以一个 ">" 来标记:

```
(gdb) py-list
 901        if options.profile:
 902            options.profile = False
 903            profile_me()
 904            return
 905
>906        u = UI()
 907        if not u.quit:
 908            try:
 909                gtk.main()
 910            except KeyboardInterrupt:
 911                # properly quit on a keyboard interrupt...
```

使用 `py-list START` 从不同的行号开始列出 Python 源代码，而 `py-list START,END` 则从列出指定行范围内的 Python 源代码。

py-up 和 py-down

`py-up` 和 `py-down` 命令类似于 GDB 的常规 `up` 和 `down` 命令，但会尝试在 CPython 帧而不是 C 帧的层级上移动。

GDB 并不总是能够读取相关的帧信息，这取决于编译 CPython 时的优化级别。 在内部，这些命令会查找正在执行默认帧求值函数（即 CPython 内的的核心字节码解释器循环）的 C 帧并查找相关 `PyFrameObject *` 的值。

它们将发出线程内的帧编号（在 C 层级上）。

例如:

```
(gdb) py-up
#37 Frame 0x9420b04, for file /usr/lib/python2.6/site-packages/
gnome_sudoku/main.py, line 906, in start_game ()
    u = UI()
(gdb) py-up
#40 Frame 0x948e82c, for file /usr/lib/python2.6/site-packages/
gnome_sudoku/gnome_sudoku.py, line 22, in start_game(main=<module at remote 0xb7
    main.start_game()
(gdb) py-up
Unable to find an older python frame
```

这样我们位于 Python 栈的顶部。

帧编号对应于 GDB 的 `backtrace` 命令所显示的内容。 该命令将跳过未在执行 Python 代码的 C 帧。

向下回退:

```
(gdb) py-down
#37 Frame 0x9420b04, for file /usr/lib/python2.6/site-packages/gnome_sudoku/mair
    u = UI()
(gdb) py-down
#34 (unable to read python frame information)
(gdb) py-down
#23 (unable to read python frame information)
(gdb) py-down
#19 (unable to read python frame information)
(gdb) py-down
#14 Frame 0x99262ac, for file /usr/lib/python2.6/site-packages/gnome_sudoku/game
            swallower.run_dialog(self.dialog)
(gdb) py-down
#11 Frame 0x9aead74, for file /usr/lib/python2.6/site-packages/gnome_sudoku/dial
            gtk.main()
(gdb) py-down
#8 (unable to read python frame information)
(gdb) py-down
Unable to find a newer python frame
```

现在我们位于 Python 栈的底部。

请注意在 Python 3.12 及更新的版本中，同一个 C 栈帧可被用于多个 Python 栈帧。 这意味着 `py-up` 和 `py-down` 可以同时移动多个 Python 帧。 例如:

```
(gdb) py-up
#6 Frame 0x7ffff7fb62b0, for file /tmp/rec.py, line 5, in recursive_function (n=
   time.sleep(5)
#6 Frame 0x7ffff7fb6240, for file /tmp/rec.py, line 7, in recursive_function (n=
   recursive_function(n-1)
#6 Frame 0x7ffff7fb61d0, for file /tmp/rec.py, line 7, in recursive_function (n=
   recursive_function(n-1)
#6 Frame 0x7ffff7fb6160, for file /tmp/rec.py, line 7, in recursive_function (n=
   recursive_function(n-1)
#6 Frame 0x7ffff7fb60f0, for file /tmp/rec.py, line 7, in recursive_function (n=
   recursive_function(n-1)
#6 Frame 0x7ffff7fb6080, for file /tmp/rec.py, line 7, in recursive_function (n=
   recursive_function(n-1)
#6 Frame 0x7ffff7fb6020, for file /tmp/rec.py, line 9, in <module> ()
   recursive_function(5)
(gdb) py-up
Unable to find an older python frame
```

py-bt

`py-bt` 命令会尝试显示当前线程的 Python 层级回溯。

例如:

```
(gdb) py-bt
#8 (unable to read python frame information)
#11 Frame 0x9aead74, for file /usr/lib/python2.6/site-packages/gnome_sudoku/dial
            gtk.main()
#14 Frame 0x99262ac, for file /usr/lib/python2.6/site-packages/gnome_sudoku/game
            swallower.run_dialog(self.dialog)
#19 (unable to read python frame information)
#23 (unable to read python frame information)
#34 (unable to read python frame information)
#37 Frame 0x9420b04, for file /usr/lib/python2.6/site-packages/gnome_sudoku/main
    u = UI()
#40 Frame 0x948e82c, for file /usr/lib/python2.6/site-packages/gnome_sudoku/gnom
    main.start_game()
```

帧编号对应于 GDB 的 `backtrace` 命令所显示的内容。

py-print

`py-print` 命令会查找一个 Python 名称并尝试打印它。 它将先在当前线程的 locals 中查找，然后是 globals，最后是 builtins:

```
(gdb) py-print self
local 'self' = <SwappableArea(running=<gtk.Dialog at remote 0x98faaa4>,
main_page=0) at remote 0x98fa6e4>
(gdb) py-print __name__
global '__name__' = 'gnome_sudoku.dialog_swallower'
(gdb) py-print len
builtin 'len' = <built-in function len>
(gdb) py-print scarlet_pimpernel
'scarlet_pimpernel' not found
```

如果当前 C 帧对应多个 Python 帧，则 `py-print` 只会考虑其中第一个。

py-locals

`py-locals` 命令会在选定的线程中查找当前 Python 帧内的所有 Python 的 locals，并打印它们的表示形式:

```
(gdb) py-locals
self = <SwappableArea(running=<gtk.Dialog at remote 0x98faaa4>,
main_page=0) at remote 0x98fa6e4>
d = <gtk.Dialog at remote 0x98faaa4>
```

如果当前 C 帧对应多个 Python 帧，同它们的所有 locals 都会被显示:

```
(gdb) py-locals
Locals for recursive_function
n = 0
Locals for recursive_function
n = 1
Locals for recursive_function
n = 2
Locals for recursive_function
n = 3
Locals for recursive_function
```

```
n = 4
Locals for recursive_function
n = 5
Locals for <module>
```

与 GDB 命令一起使用

这些扩展命令是对 GDB 的内置命令的补充。 例如，你可以使用 `py-bt` 显示的帧编号与 `frame` 命令一起使用以转到所选线程中的特定帧，如下所示:

```
(gdb) py-bt
(output snipped)
#68 Frame 0xaa4560, for file Lib/test/regrtest.py, line 1548, in <module> ()
        main()
(gdb) frame 68
#68 0x00000000004cd1e6 in PyEval_EvalFrameEx (f=Frame 0xaa4560, for file Lib/test/
2665                            x = call_function(&sp, oparg);
(gdb) py-list
1543        # Run the tests in a context manager that temporary changes the CWD to
1544        # temporary and writable directory. If it's not possible to create or
1545        # change the CWD, the original CWD will be used. The original CWD is
1546        # available from test_support.SAVEDCWD.
1547        with test_support.temp_cwd(TESTCWD, quiet=True):
>1548            main()
```

`info threads` 命令将向你提供进程内的线程列表，您还可以使用 `thread` 命令来选择不同的线程:

```
(gdb) info threads
  105 Thread 0x7fffefa18710 (LWP 10260)  sem_wait () at ../nptl/sysdeps/unix/sysv/
  104 Thread 0x7fffdf5fe710 (LWP 10259)  sem_wait () at ../nptl/sysdeps/unix/sysv/
* 1 Thread 0x7ffff7fe2700 (LWP 10145)  0x00000038e46d73e3 in select () at ../sysde
```

你可以使用 `thread apply all COMMAND` 或 (简短写法 `t a a COMMAND`) 在所有线程上运行一个命令。 配合 `py-bt`，这将让你在 Python 层级上查看看到每个线程在做什么:

```
(gdb) t a a py-bt

Thread 105 (Thread 0x7fffefa18710 (LWP 10260)):
#5 Frame 0x7fffd00019d0, for file /home/david/coding/python-svn/Lib/threading.py,
        self.__block.acquire()
#8 Frame 0x7fffac001640, for file /home/david/coding/python-svn/Lib/threading.py,
            self._acquire_restore(saved_state)
#12 Frame 0x7fffb8001a10, for file /home/david/coding/python-svn/Lib/test/lock_tes
            cond.wait()
#16 Frame 0x7fffb8001c40, for file /home/david/coding/python-svn/Lib/test/lock_tes
                f()

Thread 104 (Thread 0x7fffdf5fe710 (LWP 10259)):
#5 Frame 0x7fffe4001580, for file /home/david/coding/python-svn/Lib/threading.py,
        self.__block.acquire()
#8 Frame 0x7fffc8002090, for file /home/david/coding/python-svn/Lib/threading.py,
            self._acquire_restore(saved_state)
#12 Frame 0x7fffac001c90, for file /home/david/coding/python-svn/Lib/test/lock_tes
            cond.wait()
#16 Frame 0x7fffac0011c0, for file /home/david/coding/python-svn/Lib/test/lock_tes
                f()
```

```
Thread 1 (Thread 0x7ffff7fe2700 (LWP 10145)):
#5 Frame 0xcb5380, for file /home/david/coding/python-svn/Lib/test/lock_tests.py,
    time.sleep(0.01)
#8 Frame 0x7fffd00024a0, for file /home/david/coding/python-svn/Lib/test/lock_test
        _wait()
```