Gem #125: Обнаружение бесконечной рекурсии с помощью API Python в GDB.

Автор: Jerome Guitton - AdaCore

Краткое содержание: В данном Gem#125 описывается способ обнаружение бесконечной рекурсии с помощью API Python в GDB.

Давайте начнём…

Предположим, что вы печатаете таблицу факториалов от 9 до 0, как показано ниже:

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Try is
   function Factorial (I : Integer) return Integer is
   begin
      if I = 0 then
         return 1;
      else
         return I * Factorial (I - 1);
      end if;
   end Factorial;
   N : Integer := 9;
   F : Integer;
begin
   loop
      F := Factorial (N);
      Put (Integer'Image (N));
      Put ("! = ");
      Put (Integer'Image (F));
      New_Line;
      exit when N < 0;
      N := N - 1;
   end loop;
end Try;
```

Теперь построим пример:

```
$ ./try
 9! =  362880
 8! =  40320
 7! =  5040
 6! =  720
 5! =  120
 4! =  24
 3! =  6
 2! =  2
 1! =  1
 0! =  1
raised STORAGE_ERROR : EXCEPTION_STACK_OVERFLOW
```

Если вы запускаете программу в GDB, с помощью `'catch exception'` вы можете обнаружить бесконечную рекурсию в процедуре `Factorial`:

```
(gdb) catch exception
Catchpoint 1: all Ada exceptions
(gdb) run
```

```
Starting program: C:\home\guitton\GIT\GDB\builds\gems\1    ry.exe
[New Thread 12736.0x33a4]
 9! =  362880
 8! =  40320
 7! =  5040
 6! =  720
 5! =  120
 4! =  24
 3! =  6
 2! =  2
 1! =  1
 0! =  1
Program received signal SIGSEGV, Segmentation fault.
0x00401797 in try.factorial (i=-698787) at try.adb:9
9                return I * factorial (I - 1);
(gdb) backtrace 10
#0  0x00401797 in try.factorial (i=-698787) at try.adb:9
#1  0x0040179c in try.factorial (i=-698786) at try.adb:9
#2  0x0040179c in try.factorial (i=-698785) at try.adb:9
#3  0x0040179c in try.factorial (i=-698784) at try.adb:9
#4  0x0040179c in try.factorial (i=-698783) at try.adb:9
#5  0x0040179c in try.factorial (i=-698782) at try.adb:9
#6  0x0040179c in try.factorial (i=-698781) at try.adb:9
#7  0x0040179c in try.factorial (i=-698780) at try.adb:9
#8  0x0040179c in try.factorial (i=-698779) at try.adb:9
#9  0x0040179c in try.factorial (i=-698778) at try.adb:9
(More stack frames follow...)
```

Но теперь вы хотели бы знать контекст вызова факториала, который вызывает рекурсию. Это не может быть легко сделано в точке исключения, поскольку отладчик имеет слишком много кадров (698787, предположительно), чтобы размотать, чтобы добраться до исходного вызова.

Если добавить точку останова в `Factorial`, можно выполнить несколько команд `continue`, не обнаружив ничего подозрительного:

```
(gdb) b factorial
Breakpoint 1 at 0x40177f: file try.adb, line 6.
(gdb) run
Starting program: C:\home\guitton\GIT\GDB\builds\gems\1    ry.exe
[New Thread 11856.0x23b4]
Breakpoint 1, try.factorial (i=9) at try.adb:6
6              if I = 0 then
(gdb) cont
Continuing.
Breakpoint 1, try.factorial (i=8) at try.adb:6
6              if I = 0 then
(gdb) cont
Continuing.
Breakpoint 1, try.factorial (i=7) at try.adb:6
6              if I = 0 then
```

Есть, к сожалению, большое количество действительных вызовов факториала перед фиктивным. То, что мы хотели бы сделать, это установить верхнюю границу возможных вызовов факториала и остановиться только тогда, когда эта граница будет достигнута.

К счастью, Python API GDB может помочь: с помощью этого API вы можете пройти через структуры отладчика и получить больше информации о контексте в точке выполнения. Например, вот функция Python, которая подсчитывает количество кадров в трассировке:

```
def frame_count():
    """Count the number of frames in the backtrace"""
    c = 0
    f = gdb.newest_frame()
    while f is not None:
        c = c + 1
        f = f.older()
    return c
```

Функция использует фрейм класса из Python API GDB и его метод `Old()/newer()` для просмотра backtrace.

После создания файла `frame_count.py` которая содержит реализацию этой команды, затем ее можно добавить в команды GDB и использовать следующим образом:

```
(gdb) source frame_count.py
(gdb) python help(frame_count)
Help on function frame_count in module __main__:
frame_count()
    Count the number of frames in the backtrace
(gdb) python print frame_count()
4
```

Теперь вы можете установить точку останова в `Factorial`, которая остановится только тогда, когда `frame_count` достигнет максимальной границы. Отредактируйте `frame_count.py`, чтобы добавить следующий хук точки останова:

```
def factorial_hook():
    """Called whenever Try.Factorial is reached"""
    max_number_of_frames = 100
    fc = frame_count()
    if fc <= max_number_of_frames:
        gdb.execute("continue")
    else:
        print ""
        print ("warning: more than %d frames in backtrace."
               % max_number_of_frames)
        print "         To ease the investigation, the selected frame
will be"
        print "         the first call to factorial."
        print ""
        f = gdb.newest_frame()
        while f is not None and f.name() == "try.factorial":
            gdb.Frame.select(f)
            f = f.older()
        # Finally, show the frame that this loop selected; it is the
first
        # call to factorial
        gdb.execute("frame")
```

Затем присоедините эту команду к точке останова в Try.Factorial:

```
(gdb) source frame_count.py
(gdb) break factorial
Breakpoint 1 at 0x40177f: file try.adb, line 6.
(gdb) commands
>python factorial_hook()
>end
```

Затем можно запустить, и программа остановится при достижении максимальной границы. В этот момент Вы сможете исследовать бесконечную рекурсию гораздо проще:

```
(gdb) run
Breakpoint 1, try.factorial (i=-100) at try.adb:6
6             if I = 0 then
warning: more than 100 frames in backtrace.
         To ease the investigation, the selected frame will be
         the first call to factorial.
#99 0x0040179c in try.factorial (i=-1) at try.adb:9
9                return I * Factorial (I - 1);
```

Теперь мы можем видеть, что `Try` вызывает `Factorial` с `-1`; это и есть ошибка в условии выхода в цикле.

```
(gdb) up
#100 0x004017c1 in try () at try.adb:17
17            F := Factorial (N);
(gdb) print N
$2 = -1
```

Это всего лишь одна из расширенных возможностей отладки, которые предоставляет Python API GDB. Дополнительные сведения см. в соответствующем разделе руководства пользователя GDB.

Связанный со статьёй текст программы

Файл test.cmd

```
1. gnatmake -g try.adb
2. gdb try --command=scenario.gdb
```

Файл scenario.gdb

```
1.
2. source frame_count.py
3.
4. break factorial
5. commands
6. python factorial_hook()
7. end
8.
9. run
```

Файл frame_count.py

```
 1.
 2. def frame_count():
 3.     """Count the number of frames in the backtrace"""
 4.     c = 0
```

```
 5.     f = gdb.newest_frame()
 6.     while f is not None:
 7.         c = c + 1
 8.         f = f.older()
 9.     return c
10.
11. def factorial_hook():
12.     """Called whenever Try.Factorial is reached"""
13.     max_number_of_frames = 100
14.     fc = frame_count()
15.     if fc <= max_number_of_frames:
16.         gdb.execute("continue")
17.     else:
18.         print ""
19.         print ("warning: more than %d frames in backtrace."
20.                % max_number_of_frames)
21.         print "         To ease the investigation, the selected
frame will be"
22.         print "         the first call to factorial."
23.         print ""
24.         f = gdb.newest_frame()
25.         while f is not None and f.name() == "try.factorial":
26.             gdb.Frame.select(f)
27.             f = f.older()
28.         # Finally, show the frame that this loop selected; it is the
first
29.         # call to factorial
30.         gdb.execute("frame")
```

Файл try.adb

```
 1. with Ada.Text_IO; use Ada.Text_IO;
 2.
 3. procedure Try is
 4.    function Factorial (I : Integer) return Integer is
 5.    begin
 6.       if I = 0 then
 7.          return 1;
 8.       else
 9.          return I * Factorial (I - 1);
10.       end if;
11.    end Factorial;
12.
13.   N : Integer := 9;
14.   F : Integer;
15. begin
16.    loop
17.       F := Factorial (N);
18.
19.       Put (Integer'Image (N));
20.       Put ("! = ");
21.       Put (Integer'Image (F));
22.       New_Line;
23.
24.       exit when N < 0;
```

```
25.        N := N - 1;
26.     end loop;
27. end Try;
```

Файлы примеров Ada Gems распространяются AdaCore и могут быть использованы или изменены для любых целей без ограничений.

Об авторе

Создана инфраструктура виртуализации в парижском машинном зале. Часть команды, которая поддерживает 25 пользователей и 40 систем в различных ОС.Работа над графическими пользовательскими интерфейсами, как часть команды разработчиков, работающих над IDE.

Работа над компилятором: отвечает за перенос компилятора GNAT на Mac OS X на процессорах PowerPC и x86.Реализованы различные улучшения генерации отладочной информации.

По инфраструктуре тестирования и качества: Создано приложение для контроля качества программного обеспечения и формализм для выражения планов тестирования и обеспечения качества для всех программных продуктов компании. Этот инструмент в настоящее время находится в производстве и занимает центральное место в гибкой инфраструктуре квалификации и выпуска на AdaCore.

Автор Jerome Guitton

Jerome присоединился к AdaCore в 2002 после завершения его исследований в Ecole Nationale des Télécommunations (Париж, Франция

Национальная высшая школа связи / Telecom Paris), во время которого он уже работал с компанией на одном из ее многих исследовательских проектов, а именно, PolyORB. Его энтузиазм остался непотускневшим в течение этих шести лет, и он работал над множеством проектов, а также экспертным знанием получения в отладчиках и перекрестных технологиях. Он недавно приложил усилия к порту GNAT Pro к различным перекрестным целям (VxWorks 6, ELinOS).

Last Updated: 11/24/2017
Posted on: 5/14/2012

Обсуждение…