Luca Bonato
sviluppatore software

GDB, cos'è e come utilizzarlo per fare debugging

Techlabs

16/06/2021

develer 20

...e quando anche l'ultima speranza sembra svanire innanzi all'elusività del bug, ricorda: per quanto tu lo abbia evitato o ignorato, GDB è sempre lì accanto a te pronto a darti una mano

- anonimo su GDB

https://github.com/develersrl/webinar_gdb_in_action

per trovare codice sorgente e scripts per riprodurre
i vari esempi utilizzati in queste slides

IL MONDO DEI BUG

- i bug esistono e parte del lavoro di un programmatore è isolarli e “risolverli”
- esistono veramente tanti strumenti che aiutano a stanarli, ma conoscerli tutti è improponibile

OGNI SESSIONE DI DEBUGGING È UN MONDO A PARTE

- tipo di bug?
  - crash
  - loop infinito
  - risultato sbagliato
- riproducibilità?
  - random/flaky
  - deterministico
  - dipendente da input
- tipo di programma?
  - single-run
  - keep-alive server
- confidenza con la base di codice?

TANTE TECNICHE E STRUMENTI PER SCOPI DIFFERENTI

- `'printf'` come se piovessero
- instrumentazione durante la compilazione
- variabili d'ambiente
- debugger integrato nel proprio IDE di fiducia
- GDB
- ...e tanti altri (es: wireshark, valgrind, ipdb, strace)
- ...senza contare tutti gli script scritti ad-hoc per "il bug corrente"

...A PROPOSITO DI 'PRINTF'

+ semplice da scrivere
+ possibile post-process dei dati stampati per capire l'evoluzione del runtime del programma (senza dover rieseguire il programma)

- conoscere abbastanza bene la base di codice
- necessario ricompilare

...A PROPOSITO DI VARIABILI D'AMBIENTE

+ disponibili senza compilare o modificare il programma
+ i file di log derivanti si prestano a post-processing

- utilità discutibile: sono semplici "osservatori" built-in e non interruttori ad-hoc per lo specifico problema
- sapere che esistono e che valori possono avere può essere complesso

...A PROPOSITO DI IDE + DEBUGGER

\+ sessione di debug a portata di click
\+ unica interfaccia per codice, breakpoints ed esecuzione step-a-step

\- non tutti i progetti sono compatibili con il proprio IDE di fiducia
\- molto spesso limitato al debugging live di una sessione avviata dall'IDE stesso

...A PROPOSITO DI GDB

\+ debuggare processi remoti e/o already-running
\+ post-mortem di crash

\- interfaccia command-line può essere ostica
\- tanti comandi da mettere soggezione

+/- configurazione esplicita

GDB: the GNU Project debugger

"GDB [...] allows you to see what is going on 'inside' another program while it executes -- or what another program was doing at the moment it crashed."

https://www.gnu.org/software/gdb/

COSA CI FA VEDERE GDB

espone un'astrazione del runtime di un processo

- i threads
- lo stack
- i registri
- la memoria
- le istruzioni

I SIMBOLI DI DEBUG

i binari possono contenere informazioni aggiuntive (simboli di debug) che permettono a GDB di visualizzare informazioni “programmer-friendly” invece di sequenze binarie o codice macchina

- il binario deve essere compilato appositamente

I simboli di debug - GBD: the GNU Project debugger

per vedere se un binario contiene simboli di debug:

‘$ file eseguibile’

```
lbonato@boole ~/webinar/GDB_in_action ->
17:03 gio 03 giu$ file exe | grep stripped
exe: ELF 64-bit LSB shared object, x86-64, ve
rsion 1 (SYSV), dynamically linked, interpret
er /lib64/ld-linux-x86-64.so.2, BuildID[sha1]
=28f027e642ece48c774ba51dfdeb53025a8b0415, fo
r GNU/Linux 3.2.0, not stripped
```

per vedere i vari simboli all’interno di un binario:

‘$ nm -C eseguibile’

```
lbonato@boole ~/webinar/GDB_in_action ->
17:12 gio 03 giu$ nm -C exe | grep main
                 U __libc_start_main@@GLIBC_2.2.5
000000000000294d T main
0000000000003d46 t __gnu_cxx::__ops::_Iter_comp_val<main::{lambda(CurvePoint
 const&, CurvePoint const&)#1}>::_Iter_comp_val(__gnu_cxx::__ops::_Iter_comp
_iter<{lambda(CurvePoint const&, CurvePoint const&)#1}>&&)
0000000000003d46 t __gnu_cxx::__ops::_Iter_comp_val<main::{lambda(CurvePoint
 const&, CurvePoint const&)#1}>::_Iter_comp_val(__gnu_cxx::__ops::_Iter_comp
_iter<{lambda(CurvePoint const&, CurvePoint const&)#1}>&&)
0000000000003ed4 t bool __gnu_cxx::__ops::_Iter_comp_val<main::{lambda(Curve
Point const&, CurvePoint const&)#1}>::operator()<__gnu_cxx::__normal_iterato
```

IL CORE DUMP

- tutte le informazioni del processo vengono salvata in un file
- GDB offre un modo per ispezionare questo file proprio come se fosse un processo in esecuzione
- utile da avere perchè “non si sa quando si crasha, ma quando si crasha se ne vuole sapere il motivo”

I COMANDI DI GDB

GDB ha tantissimi comandi. Alcuni sono utili per degli usi molto specifici, mentre altri sono più generici. Conoscere alcuni di questi comandi più generici torna sempre comodo!

In caso di dubbio:

1. *https://sourceware.org/gdb/current/onlinedocs/gdb/*
2. (gdb) help <comando>

Lanciare GDB - i comandi di GDB

```
$ man gdb
GDB(1)                                  GNU Development Tools
        GDB(1)

NAME
        gdb - The GNU Debugger

SYNOPSIS
        gdb [-help] [-nh] [-nx] [-q] [-batch] [-cd=dir] [-f] [-b bps]
        [-tty=dev] [-s symfile] [-e prog] [-se prog] [-c core] [-p procID]
        [-x cmds] [-d dir] [prog|prog procID|prog core]


$ gdb --help
This is the GNU debugger.  Usage:

        gdb [options] [executable-file [core-file or process-id]]
        gdb [options] --args executable-file [inferior-arguments ...]
```

- '***gdb exe arg1 ... argN***' carica e inizia ad eseguire il programma
- '***gdb exe***' carica l'eseguibile ma non fa partire il programma
- '**gdb**' fa partire GDB e si demanda ad un secondo momento il caricamento (o l'ancoraggio) ad un programma

Eseguire il programma all'interno di GDB - i comandi di GDB

- '***run arg1 … argN***' inizia l'esecuzione
- '***continue***' continua l'esecuzione
- '***step [N]***' esegue le prossime N istruzioni (contando anche le istruzioni delle funzioni annidate) e si ferma
- '***next [N]***' esegue le prossime N istruzioni dello stesso frame (no funzioni annidate) e si ferma
- '***finish***' continua l'esecuzione fino al momento di uscita dalla funzione corrente
- '***CTRL-C***' interrompe l'esecuzione del programma (come se si fosse raggiunto un breakpoint). Il programma rimane in uno stato consistente ed è possibile continuare l'esecuzione

Fermarsi prima di eseguire una istruzione - i comandi di GDB

```
(gdb) break commands_stack.cpp:20
Breakpoint 1 at 0x1238: file commands_stack.cpp, line 20.
(gdb) break foo_2
Breakpoint 2 at 0x11d1: file commands_stack.cpp, line 8.
(gdb) run
Starting program: [...]/webinar/GDB_in_action/a.out

Breakpoint 2, foo_2 (b=32767) at commands_stack.cpp:8
8       int foo_2(int b) {
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x... in main(int, char**) at commands_stack.cpp:20
2       breakpoint     keep y   0x... in foo_2(int) at commands_stack.cpp:8
        breakpoint already hit 1 time
(gdb) delete 2
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x... in main(int, char**) at commands_stack.cpp:20
(gdb) continue
Continuing.

Breakpoint 1, main (argc=1, argv=0x7fffffffde68) at commands_stack.cpp:20
20      std::cout << value << std::endl;
```

- '***break <funzione>***' stop appena entrati nella funzione
- '***break <file>:<linea>***' stop al punto specifico
- '***info breakpoints***' elenca tutti i breakpoints e altre informazioni
- '***delete <N>***' per eliminare breakpoints non più utili

Fermarsi quando cambia il valore di una variabile - i comandi di GDB

```
(gdb) watch res
No symbol "res" in current context.
(gdb) # go where the symbol is in scope using break etc...
[...]
(gdb) watch res
Hardware watchpoint 2: res
(gdb) info watchpoints
Num     Type           Disp Enb Address    What
2       hw watchpoint  keep y              res
(gdb) continue
Continuing.
Hardware watchpoint 2: res
Old value = 21845
New value = 0
foo (a=12) at commands_optimizedout.cpp:8
8       while (b>0) {
(gdb) continue
Continuing.
Hardware watchpoint 2: res
Old value = 0
New value = 13
foo (a=12) at commands_optimizedout.cpp:10
10              b--;
```

- '***watch <variable>***' stop quando viene cambiato valore ad una variabile
- '***watch <memory>***' stop quando una locazione di memoria viene modificata
- '***info watchpoints***' elenca tutti i watchpoints
- '***delete <N>***' rimuove un watchpoint

Esplorare le variabili - i comandi di GDB

```
(gdb) print my_var
$1 = {c_1 = {a_1 = 1, a_2 = 2, a_3 = 3.1400000000000001}, c_2 = {b_1 = {a_1 = 0,
a_2 = 0, a_3 = 0}, b_2 = std::vector<bool> of length 0, capacity 0}}

(gdb) ptype my_var.c_1
type = struct A {
        int a_1;
        int a_2;
        double a_3;
}

(gdb) explore my_var
The value of 'my_var' is a struct/class of type 'C' with the following fields:

  c_1 = <Enter 0 to explore this field of type 'A'>
  c_2 = <Enter 1 to explore this field of type 'B'>

Enter the field number of choice: 0
The value of 'my_var.c_1' is a struct/class of type 'A' with the following fields:

  a_1 = 1 .. (Value of type 'int')
  a_2 = 2 .. (Value of type 'int')
  a_3 = 3.1400000000000001 .. (Value of type 'double')
```

- ***'print <var>'*** per stampare una variabile specifica
- ***'ptype <var>'*** scoprire il tipo di una variabile e osservare la composizione interna
- ***'explore <var>'*** esplorare dinamicamente una variabile e il suo contenuto

Gestire lo stack - i comandi di GDB

```
(gdb) backtrace
#0  foo_3 (c=26) at commands_stack.cpp:5
#1  0x0000555555555f2 in foo_2 (b=13) at commands_stack.cpp:10
#2  0x0000555555555216 in foo_1 (a=12) at commands_stack.cpp:15
#3  0x0000555555555235 in main (argc=1, argv=0x7ff[..]8) at [...]ck.cpp:19

(gdb) frame 2
#2  0x0000555555555216 in foo_1 (a=12) at commands_stack.cpp:15
15          return foo_2(z);
(gdb) info locals
z = 13
(gdb) info args
a = 12

(gdb) frame 1
#1  0x0000555555555f2 in foo_2 (b=13) at commands_stack.cpp:10
10          return foo_3(w);
(gdb) info locals
w = 26
(gdb) info args
b = 13
```

- '***backtrace***' per avere la lista dello stack del thread corrente
- '***frame <N>***' per navigare lo stack
- '**info locals**' per avere la lista delle variabili disponibili dallo scope del frame corrente
- '**info args**' per avere la lista delle variabili di input della funzione corrispondente al frame corrente

Gestire lo stack (part 2) - i comandi di GDB

```
Breakpoint 1, foo (a=12) at commands_optimizedout.cpp:8
8         while (b>0) {
(gdb) info locals
b = 13
res = 0
(gdb) continue
Continuing.
Breakpoint 2, foo (a=<optimized out>) at commands_optimizedout.cpp:12
12        std::cout<<res<<std::endl;
(gdb) info locals
b = 0
res = <optimized out>

(gdb) list 8
5         void __attribute__((noinline)) foo(int a) {
6           int b = a+1;
7           int res = 0;
8           while (b>0) {
9             res += b;
10            b--;
11          }
12          std::cout<<res<<std::endl;
```

- '**<optimized out>**' spesso significa che l'eseguibile è stato compilato con qualche flag di ottimizzazione, e che la variabile cercata non è più disponibile (ma era disponibile N istruzioni fa)

- '**Cannot evaluate function -- may be inlined**' spesso significa che il linker non ha aggiunto il codice della corrispettiva funzione (perché inutile o perché diventata inline)

Gestire i thread - i comandi di GDB

```
(gdb) info threads
  Id   Target Id                                  Frame
* 1      Thread 0x7ffff7a44740 "a.out" __pthread_clockjoin_ex at [...]
  2      Thread 0x7ffff7a43700 "a.out" __GI___libc_read at[...]x/read.c:26
  3      Thread 0x7ffff7242700 "a.out" [...] in foo2 () at commands.cpp:11

(gdb) thread 1
[Switching to thread 1 (Thread 0x7ffff7a44740 (LWP 28092))]
#0  __pthread_clockjoin_ex at pthread_join_common.c:145
145   in pthread_join_common.c
(gdb) bt
#0  __pthread_clockjoin_ex [...] at pthread_join_common.c:145
#1  0x00007ffff7e7bfe7 in std::thread::join() () from [...]/libstdc++.so.6
#2  0x0000555555555331 in main (argc=[...]) at commands.cpp:19

(gdb) thread 3
[Switching to thread 3 (Thread 0x7ffff7242700 (LWP 28097))]
#0  0x00005555555552da in foo2 () at commands.cpp:11
11          while(true) a++;
(gdb) bt
#0  0x00005555555552da in foo2 () at commands.cpp:11
#1  0x0000555555555d02 in std::__invoke_impl[...] at [...]/invoke.h:60
....
```

- '***info threads***' per avere una lista di tutti i threads attivi nel processo sotto debug e un piccolo recap di cosa stanno facendo
- '***thread <N>***' per passare da un thread all'altro (per poi usare altri comandi)

Automatizzare comandi - i comandi di GDB

```
(gdb) break commands_optimizedout.cpp:8
Breakpoint 1 at 0x11c8: file commands_optimizedout.cpp, line 8.
(gdb) commands 1
Type commands for breakpoint(s) 1, one per line.
End with a line saying just "end".
>print res
>print b
>end

(gdb) run
Starting program: [...]/webinar/GDB_in_action/a.out

Breakpoint 1, foo (a=12) at commands_optimizedout.cpp:8
8         while (b>0) {
$1 = 0
$2 = 13
(gdb) continue
Continuing.

Breakpoint 1, foo (a=12) at commands_optimizedout.cpp:8
8         while (b>0) {
$3 = 13
$4 = 12
```

- ***'command <N> .... end'*** per elencare tutti i comandi che devono essere eseguiti una volta raggiunto il break/watch-point *<N>*

QUASI-LIVE DEBUGGING SESSION CON GDB

- come far interagire tra loro i comandi di gdb per ottenere qualche risultato utile
- si può fare: GDB da terminale è usabile e fa il suo lavoro

CAPITOLO 00: IL PROGRAMMA

```
    // parse input...
    std::vector<CurvePoint> interval;     // create points
    interval.reserve(max_x);
    for (int i=0; i<max_x; i++)
      interval.push_back(CurvePoint(double(i), coeff));

    double accumulator  = 0.0;  // approximate integral
using x-axis quantization of 1 unit
    for (int i=0, n=interval.size(); i<n; i++) {
        interval[i]._value_integral  = accumulator;
        accumulator  += 1.0 * interval[i].y();}

    // sort based on the just-computed integral value...
    auto soi= [](const CurvePoint &a, const CurvePoint &b)
{
        return a.integral() < b.integral();
    };
    std::sort(interval.begin()+1, interval.end(), soi);

    CurvePoint  &min = interval[1];  // find and print
result
    CurvePoint  &max = interval.back();
    print_results(min, max);   // will call CurvePoint->y()
```

```
struct CurvePoint {
    std::vector<double> _coefficients;
    double _value_x;
    double _value_integral;
    PolynomialComputation _computer;
    double y() const { return this->_computer.compute(); };
};

struct PolynomialComputation {
    CurvePoint *_parent;
    double compute() const;
};

double PolynomialComputation::compute() const {
    // using Horner's method
    double res = _parent->coeffAt(0);
    for(int i=1; i<_parent->degree(); i++) {
        res = res*_parent->_value_x + _parent->coeffAt(i);
    }
    return res;
}
```

CAPITOLO 01 – 11: ISOLIAMO E RISOLVIAMO IL BUG

il codice sorgente, i vari step da utilizzare all'interno di GDB e rr (https://rr-project.org/) e i video della sessione di debugging sono recuperabili all'indirizzo:

https://github.com/develersrl/webinar_gdb_in_action/tree/main/gdb_debugging_session

- capitolo 01: il misfatto
- capitolo 02: gli indizi
- capitolo 03: la disperazione
- capitolo 04: il raggio di speranza
- capitolo 05: capiamo chi è il colpevole
- capitolo 06: mettiamolo alle strette
- capitolo 07: ricostruiamo gli eventi passo passo
- capitolo 08: la tragedia del "rerun"
- capitolo 09: 'rr'egistriamo il crash
- capitolo 10: cosa da "giorno dopo"
- capitolo 11: veniamone a capo e 'rr'isolviamo

CAPITOLO 12: LA CONCLUSIONE

estratto da https://en.cppreference.com/w/cpp/algorithm/sort:

Type requirements

- RandomIt must meet the requirements of *ValueSwappable* and *LegacyRandomAccessIterator*.
- The type of dereferenced RandomIt must meet the requirements of ***MoveAssignable*** and ***MoveConstructible***.
- Compare must meet the requirements of *Compare*.

…ci mancava `'CurvePoint::operator='` ridefinito correttamente per gestire il puntatore interno `'CurvePoint._computer._parent'`

ogni riferimento a bug, codice e persone è puramente reale

Luca Bonato

lbonato@develer.com

Vuoi rimanere aggiornato sugli eventi Develer?
Seguici nei nostri canali social:

develer | 20

www.develer.com