Debugging Go

tealeg (Geoffrey J. Teale)

<2018-07-03 Di>

Debugging symbols

The Go compiler writes debugging symbols by default

- DWARF - http://dwarfstd.org/
- Contains a "Line Number Table", mapping compiled location to source location and specifies which instructions form the beginning and end of functions
- Contains a "Call Frame Information Table", which maps the location of frames on the call stack.
- Tables are in a byte-code optimised form to drive a finite state machine.

We can prove it...

Observing debug information in the binary

```
go build .
file ./presentation
ls -lath ./presentation
```

Note that it says `with debug_info`.

It also says `not stripped`, and it's 2.0Mb.

An aside: making smaller binaries

Just for fun, lets strip it:

```
strip ./presentation
file ./presentation
ls -lath ./presentation
```

We can achieve the same at compile time:

```
go build -ldflags="-s -w" .
file ./presentation
ls -lath ./presentation
```

Interactive debuggers [1]

An interactive debugger can gain control of process in order to debug it via operating system specific calls. On Linux this is the `ptrace` system call.

Typically the debugger forks off a new process. The program to be debugged is executed in the child process, having instructed the kernel to allow the parent process to trace it.

Now, any signal sent to the child process (apart from `SIGKILL`) will cause the process to pause and the parent will be notified. The interval between signals depends on what instructions we give `ptrace`.

Interactive debuggers [2]

In the child fork, call `PTRACE_ME` and execute the program to be debugged:

A trivial example in C

```
if (ptrace(PTRACE_TRACEME, 0, 0, 0) < 0) {
  perror("ptrace");
  return;
 }

execl(programname, programname, 0);
```

Interactive debuggers [3]

In the parent, step through the program, line-by-line:

Trivial steppng loop in C using ptrace

```
wait(&wait_status);

while (WIFSTOPPED(wait_status)) {
  if (ptrace(PTRACE_SINGLESTEP, child_pid, 0, 0) < 0) {
    perror("ptrace");
    return;
  }
  wait(&wait_status);
 }
```

Interactive debuggers [4]

Alternative: Attach to running process with `PTRACE_ATTACH`
Trivial example in C, attaching to pid `123`.

```
traced_process = 123;
ptrace(PTRACE_ATTACH, traced_process, NULL, NULL);
wait(NULL);
```

Note: you'll need to have the right permission to attach to a process.

Interactive debuggers [5]

Once you have a debugging session you can issue further calls to `ptrace` to inspect the memory of the running process and step through it's instructions.

To make this human readable, you'll need the DWARF tables to map the state back to the source.

This can all be done in Go too! Go provides native support in the debugging package for interacting with ELF (`debug/elf`), DWARF (`debug/dwarf`) and the line-mapping (`debug/gosym`). Go also has `ptrace` bindings in the `syscall` package.

Interactive debuggers [6]

You don't need to write a debugger.

Because DWARF is a standard, standard tools work

You can use `gdb` (Linux / Unix / Windows)

You can use `lldb` (Mac OS X)

If your IDE has a built in debugger, use that!

Otherwise use Derek Parker's `dlv` (Delve)

Installing delve

Not just a `go get` (unless you're on Linux)

Follow instructions, here:
`https://github.com/derekparker/delve/`

Simplest debug session [1]

There are two easy ways to invoke a Delve session

Debug a binary

```
dlv exec ./dumb
```

Debug from within the source directory

```
dlv debug
```

Simplest debug session [2]

We need to tell Delve where to start from. This is called a "break point".

We can specify it by name:

```
(dlv) break main.getNum
```

.. or by line number:

```
(dlv) break dumb.go:22
```

Note: you can just type `b` instead of `break`

Simplest debug session [3]

Now we need to tell the debugger to continue until it hits the next break-point (in this case, our `main` function).

```
(dlv) continue
```

Note: you can just type `c` instead of `continue` Note: if you use `next` now instead of `continue`, you'll see the Go runtimes startup code.

Simplest debug session [4]

The delve session should now be showing us some code and a pointer to the current line:

```
> main.main() ./dumb.go:22 (hits goroutine(1):1 total:1) (PC
    20:
    21: func main() {
=>  22:         r := bufio.NewReader(os.Stdin)
    23:         fmt.Print("Numerator  ")
    24:         numerator := getNum(r)
    25:         fmt.Print("Denominator  ")
    26:         denominator := getNum(r)
    27:         result := divide(numerator, denominator)
(dlv)
```

Simplest debug session [5]

We can now step the code one instruction forwards:

```
(dlv) next
```

... or simply type "n"

Simplest debug session [6]

Now that the first command completed we can inspect the variable that was set:

```
(dlv) print r
*bufio.Reader {
    buf: []uint8 len: 4096, cap: 4096, [0,0,0,0,0,0,0,0,0,0,
    rd: io.Reader(*os.File) *{
    file: *(*os.file)(0xc420094000),},
    r: 0,
    w: 0,
    err: error nil,
    lastByte: -1,
    lastRuneSize: -1,}
(dlv)
```

Simplest debug session [7]

Now lets skip forward to the 2nd breakpoint we set:

```
(dlv) c
Numerator  > main.getNum() ./dumb.go:10 (hits goroutine(1)
     9:
=>  10:     func getNum(r *bufio.Reader) int {
    11:             fmt.Print("please enter a number:")
    12:             line, _, _ := r.ReadLine()
    13:             num, _ := strconv.Atoi(string(line))
    14:             nreturn num
    15:     }
(dlv)
```

Simplest debug session [8]

And we can step through this line by line using `next` or `n`.

Eventually we'll step out of the `getNum` function.

Simplest debug session [9]

Eventually we'll reach this line, which looks interesting:

```
(dlv) n
please enter a number:> main.main() ./dumb.go:27 (PC: 0x4a21
    22:         r := bufio.NewReader(os.Stdin)
    23:         fmt.Print("Numerator  ")
    24:         numerator := getNum(r)
    25:         fmt.Print("Denominator  ")
    26:         denominator := getNum(r)
=>  27:         result := divide(numerator, denominator)
    28:         fmt.Printf("%d/%d = %d\n", numerator, denomi
    29: }
```

Debugging goroutines [1]

```
cd goroutines
dlv debug
(dlv) break main.go:17
(dlv) continue
```

Debugging goroutines [2]

Now lets inspect the goroutines

```
(dlv) goroutines
[5 goroutines]
0 Goroutine 1 - User: ./main.go:17 main.main (0x49d00f) (thr
  Goroutine 2 - User: /usr/lib/go/src/runtime/proc.go:292 ru
  Goroutine 3 - User: /usr/lib/go/src/runtime/proc.go:292 ru
  Goroutine 4 - User: /usr/lib/go/src/runtime/proc.go:292 ru
  Goroutine 5 - User: ./main.go:7 main.f (0x49cf67)
(dlv)
```

Debugging goroutines [3]

We can also show the threads (note that this isn't necessarily a 1:1 mapping):

```
(dlv) threads
0 Thread 22099 at 0x49d00f ./main.go:17 main.main
  Thread 22107 at 0x455863 /usr/lib/go/src/runtime/sys_linux
  Thread 22108 at 0x455d93 /usr/lib/go/src/runtime/sys_linux
  Thread 22109 at 0x455d93 /usr/lib/go/src/runtime/sys_linux
  Thread 22110 at 0x455d93 /usr/lib/go/src/runtime/sys_linux
```

Debugging goroutines [4]

Now we can switch to the other goroutine and inspect it:

```
(dlv) goroutine 5
Switched from 1 to 5 (thread 22099)
(dlv) goroutine
Thread 23183 at ./main.go:5
Goroutine 5:
Runtime: ./main.go:5 main.f (0x49cf1f)
User: ./main.go:5 main.f (0x49cf1f)
Go: ./main.go:15 main.main (0x49d00f)
```

Debugging goroutines [5]

If we start stepping through now, we'll probably find ourselves deep in the go runtimes

In this case, `stepout` is your friend.

Note that this goroutine will have already run for as long as it can without blocking. If you really want to debug it from the start you'll need to set a break point. Background go routines run when unattended.

I don't (yet) know of a way to step through multiple goroutines in parallel.

Debugging tests [1]

Simple, just invoke the test command in your source directory:

```
dlv test .
```

.. set a breakpoint:

```
(dlv) break TestDivide
```

.. away you go.

Remote debugging

Start a headless debug session

```
dlv debug --headless
> API server listening at: 127.0.0.1:34607
```

Start a client

```
dlv connect localhost:34607
```

delve - command summary [1]

args	Print function arguments.
break	Sets a breakpoint.
breakpoints	Print out info for active breakpoints.
clear	Deletes breakpoint.
clearall	Deletes multiple breakpoints.
condition	Set breakpoint condition.
config	Changes configuration parameters.
continue	Run until breakpoint or program termination.
disassemble	Disassembler.
exit	Exit the debugger.
frame	Executes command on a different frame.
funcs	Print list of functions.
goroutine	Shows or changes current goroutine
goroutines	List program goroutines.

delve - command summary [2]

help	Prints the help message.
list	Show source code.
locals	Print local variables.
next	Step over to next source line.
on	Executes a command when a breakpoint is hit.
print	Evaluate an expression.
regs	Print contents of CPU registers.
restart	Restart process.
set	Changes the value of a variable.
source	Executes a file containing a list of delve commands
sources	Print list of source files.
stack	Print stack trace.
step	Single step through program.
step-instruction	Single step a single cpu instruction.

delve - command summary [3]

stepout	Step out of the current function.
thread	Switch to the specified thread.
threads	Print out info for every traced thread.
trace	Set tracepoint.
types	Print list of types
vars	Print package variables.
whatis	Prints type of an expression.

Exotica! rr

rr is project from Mozilla
`https://rr-project.org/`

Record and Replay

It's new, so it's not packaged everywhere yet

Using rr with delve [1]

On Linux, you'll need to allow `perf` to be used by non root users:

```
sudo sh -c 'echo 1 >/proc/sys/kernel/perf_event_paranoid'
```

Using rr with delve [2]

First record a programs runtime:

```
rr record ./dumb
rr: Saving execution to trace directory ‘/home/tealeg/.local
Numerator  please enter a number:1
Denominator  please enter a number:0
panic: runtime error: integer divide by zero

goroutine 1 [running]:
main.divide(...)
/home/tealeg/scratch/GoDebugPresentation/dumb/dumb.go:18
main.main()
/home/tealeg/scratch/GoDebugPresentation/dumb/dumb.go:27 +0x
```

Uing rr with delve [3]

Now we can replay the program in delve:

```
dlv replay /home/tealeg/.local/share/rr/dumb-0
```

.. we can break and step just like normal.

Sadly we can't inspect variables yet. Buggy!

Exotica! eBPF

The Enhanced Berkley Packet Filter

Available in Linux 4.x series kernels

Allows introspection of the process via the kernel.

You'll need to install the BPF Compiler Collection and its tools
`https://github.com/iovisor/bcc`

Sadly.. I haven't managed to get it to work yet!

If there's time. . .

dlv + gud