RX-M Cloud Native Consulting

Go

Debugging

Tracking down bugs in your code is a chore. This is even more true of highly parallel code. Having a good debugger at your disposal can make all the difference when it comes to tracking down a difficult, or hard to reproduce bug. In this lab we'll consider gdb and use the delve debugger.

1. GDB

GDB does not understand Go programs well. The stack management, threading, and runtime contain aspects that differ enough from the execution model GDB expects that they can confuse the debugger, even when the program is compiled with gccgo. As a consequence, although GDB can be useful in some situations, it is not a reliable debugger for Go programs, particularly heavily concurrent ones. Moreover, it is not a priority for the Go project to address these issues, which are difficult. In short, another solution is (was) needed.

2. Install Delve

Delve is a debugger focused on simplifying the debugging of Go code which contrasts with traditional tools, such as GDB, which offer only general purpose debugging features.

https://github.com/derekparker/delve

Delve is only available on 64-bit systems (32-bit support is on the way).

You can build Delve from source or simply go get it. There are instructions for Windows, OSX and Linux on the Delve site.

Install Delve with go get:

```
user@ubuntu:~$ go get github.com/derekparker/delve/cmd/dlv
user@ubuntu:~$
```

3. A simple debugging session

Ease of use is a major goal for Delve. To build and debug with delve you can simply run `dlv debug`. Run that command in the same directory you would run go build from and it will compile your program, passing along flags to make the resulting binary easier to debug, and then start your program, attach the debugger to it, and land you at a prompt to begin inspecting your program.

Let's try it. Create a workspace for your experiments:

```
user@ubuntu:~$ mkdir -p ~/go/src/lab-debug

user@ubuntu:~$ cd !$

cd ~/go/src/lab-debug

user@ubuntu:~/go/src/lab-debug$
```

Create a hello world program to test:

```
package main

import "fmt"

func main() {
        fmt.Println("Hello")
}
```

Delve (dlv is the binary) is not on the path when you go get it. Add dlv to the path and debug your program:

```
user@ubuntu:~/go/src/lab-debug$ ls -l
total 4
-rw-rw-r-- 1 user user 66 Aug 10 12:42 hello.go

user@ubuntu:~/go/src/lab-debug$ which dlv
```

```
user@ubuntu:~/go/src/lab-debug$ find ~/go -name dlv

/home/user/go/bin/dlv
/home/user/go/pkg/linux_amd64/github.com/derekparker/delve/cmd/dlv
/home/user/go/src/github.com/derekparker/delve/cmd/dlv

user@ubuntu:~/go/src/lab-debug$ export PATH=/home/user/go/bin:$PATH

user@ubuntu:~/go/src/lab-debug$ dlv debug
Type 'help' for list of commands.
(dlv)
```

Great, now we're debugging! Set a breakpoint on the main package main() function:

```
(dlv) break main.main
Breakpoint 1 set at 0x47b7a8 for main.main() ./hello.go:5
(dlv)
```

The output tells us the breakpoint ID, the address the breakpoint was set at, the function name, and the file:line.

Use the `continue` command to run to the breakpoint:

```
(dlv) continue
> main.main() ./hello.go:5 (hits goroutine(1):1 total:1) (PC: 0x47b7a8)
     1: package main
     2:
     3: import "fmt"
     4:
=>   5: func main() {
     6:         fmt.Println("Hello")
     7: }
(dlv)
```

Now use the `next` command to execute the next line of code:

```
(dlv) next
> main.main() ./hello.go:6 (PC: 0x47b7bf)
     1: package main
     2:
     3: import "fmt"
     4:
     5: func main() {
=>   6:         fmt.Println("Hello")
     7: }
(dlv)
```

Hit enter to repeat the previous command (which was next):

```
(dlv)
Hello
> main.main() ./hello.go:7 (PC: 0x47b867)
     2:
     3: import "fmt"
     4:
     5: func main() {
     6:         fmt.Println("Hello")
=>   7: }
(dlv)
```

Use the `quit` command to end the debugging session:

```
(dlv) quit
user@ubuntu:~/go/src/lab-debug$
```

4. Additional Delve modes

Use the --help switch to display the dlv command line options:

```
user@ubuntu:~/go/src/lab-debug$ dlv --help

Delve is a source level debugger for Go programs.

Delve enables you to interact with your program by controlling the execution of the process,
evaluating variables, and providing information of thread / goroutine state, CPU register state and more.

The goal of this tool is to provide a simple yet powerful interface for debugging Go programs.

Pass flags to the program you are debugging using `--`, for example:

`dlv exec ./hello -- server --config conf/config.toml`

Usage:
  dlv [command]

Available Commands:
  attach      Attach to running process and begin debugging.
  connect     Connect to a headless debug server.
  core        Examine a core dump.
  debug       Compile and begin debugging main package in current directory, or the package specified.
  exec        Execute a precompiled binary, and begin a debug session.
  help        Help about any command
  run         Deprecated command. Use 'debug' instead.
  test        Compile test binary and begin debugging program.
  trace       Compile and begin tracing program.
  version     Prints version.

Flags:
      --accept-multiclient   Allows a headless server to accept multiple client connections. Note that the server API is not reentrant
and clients will have to coordinate.
      --api-version int      Selects API version when headless. (default 1)
      --backend string       Backend selection:
        default         Uses lldb on macOS, native everywhere else.
        native          Native backend.
        lldb            Uses lldb-server or debugserver.
        rr              Uses mozilla rr (https://github.com/mozilla/rr).
 (default "default")
      --build-flags string   Build flags, to be passed to the compiler.
      --headless             Run debug server only, in headless mode.
      --init string          Init file, executed by the terminal client.
  -l, --listen string        Debugging server listen address. (default "localhost:0")
      --log                  Enable debugging server logging.
      --wd string            Working directory for running the program. (default ".")

Use "dlv [command] --help" for more information about a command.

user@ubuntu:~/go/src/lab-debug$
```

As you can see, Delve has many features. To build and debug tests you can use `dlv test`. This is handy if you do not have a main function, or want to debug your program in the context of your test suite. This will build a test binary, using the correct flags for an optimal debugging experience, and land you at a prompt where you can begin issuing commands.

To attach to a running process you can use `dlv attach <pid>`. This will immediately stop the process and begin a debug session. Keep in mind, however, you may run into issues attempting to debug a binary compiled with certain optimizations.

You can also trace instead of debugging with `dlv trace [regexp]`. This will compile and start the program, setting tracepoints at any function that matches [regexp]. This will not begin a full debug session, but will print information whenever a tracepoint is hit.

> dlv trace [package] regexp Options -p, --pid int Pid to attach to. -s, --stack int Show stack trace with given depth.

Let's try a tracepoint on Println:

```
user@ubuntu:~/go/src/lab-debug$ dlv trace Println

> fmt.Println([]interface {} len: 1, cap: 1, [...], 842350813584, (unreadable invalid interface type: could not find str field))
/usr/local/go/src/fmt/print.go:256 (hits goroutine(1):1 total:1) (PC: 0x47489f)

> fmt.(*pp).doPrintln((*fmt.pp)(0xc420076000), []interface {} len: 1, cap: 1, [...]) /usr/local/go/src/fmt/print.go:1133 (hits
goroutine(1):1 total:1) (PC: 0x47b273)
Hello
Process 61731 has exited with status 0

user@ubuntu:~/go/src/lab-debug$
```

This picks up our call to fmt.Println and also the internal call it in turn makes to doPrintln().

Try a regular expression that only matches the Println() method:

```
user@ubuntu:~/go/src/lab-debug$ dlv trace '[^.]Println'
> fmt.(*pp).doPrintln((*fmt.pp)(0xc42007a000), []interface {} len: 1, cap: 1, [...]) /usr/local/go/src/fmt/print.go:1133 (hits
goroutine(1):1 total:1) (PC: 0x47b273)
Hello
Process 62571 has exited with status 0
user@ubuntu:~/go/src/lab-debug$
```

These will likely be your most used commands, however Delve has the following subcommands as well:

- `$ dlv exec ./path/to/binary` - Run and attach to an existing binary
- `$ dlv connect` - connect to a headless debug server

5. More debugging

Enter the following slightly larger program for debugging.

```
package main

import (
        "fmt"
        "sync"
)

func dostuff(wg *sync.WaitGroup, i int) {
        fmt.Printf("goroutine id %d\n", i)
        fmt.Printf("goroutine id %d\n", i)
        wg.Done()
}

func main() {
        var wg sync.WaitGroup
        workers := 10

        wg.Add(workers)
        for i := 0; i < workers; i++ {
                go dostuff(&wg, i)
        }
        wg.Wait()
}
```

Let's begin a debug session with $ dlv debug and start by setting a breakpoint at main:

```
user@ubuntu:~/go/src/lab-debug$ vim more.go

user@ubuntu:~/go/src/lab-debug$ dlv debug more.go
Type 'help' for list of commands.

(dlv) break main.main
Breakpoint 1 set at 0x47d803 for main.main() ./more.go:14

(dlv)
```

The output tells us the breakpoint ID, the address the breakpoint was set at, the function name, and the file:line.

We can continue to that breakpoint using the continue command. Once you stop at that breakpoint, explore your program by typing next and then pressing the Enter key (Delve will repeat the last command given when it receives an empty one). The next command will step the program forward by one source code line. Now, let's try looking around: use the print command to print the value of workers:

```
> main.main() ./more.go:18 (PC: 0x47d843)
    13:
    14: func main() {
    15:         var wg sync.WaitGroup
    16:         workers := 10
    17:
=>  18:         wg.Add(workers)
    19:         for i := 0; i < workers; i++ {
    20:                 go dostuff(&wg, i)
    21:         }
    22:         wg.Wait()
    23: }
(dlv) print workers
10
(dlv)
```

Delve can also evaluate many expressions, try a Boolean expression using workers and a relational operator:

```
(dlv) print workers < 100
true
(dlv)
```

Let's set another breakpoint at our dostuff function:

```
(dlv) break dostuff
Breakpoint 2 set at 0x47d5f8 for main.dostuff() ./more.go:8
(dlv)
```

Again, let's continue which should land us at the breakpoint we just set:

```
(dlv) continue
> main.dostuff() ./more.go:8 (hits goroutine(23):1 total:2) (PC: 0x47d5f8)
> main.dostuff() ./more.go:8 (hits goroutine(28):1 total:2) (PC: 0x47d5f8)
     3: import (
     4:         "fmt"
     5:         "sync"
     6: )
     7:
=>   8: func dostuff(wg *sync.WaitGroup, i int) {
     9:         fmt.Printf("goroutine id %d\n", i)
    10:         fmt.Printf("goroutine id %d\n", i)
    11:         wg.Done()
    12: }
    13:
(dlv)
```

Let's print out the value of i:

```
(dlv) print i
9
(dlv)
```

Now, let's use the next command and then print out the value of i again. You'll notice it's the same, and this is no coincidence.

```
(dlv) next
> main.dostuff() ./more.go:9 (PC: 0x47d60f)
     4:         "fmt"
     5:         "sync"
     6: )
     7:
     8: func dostuff(wg *sync.WaitGroup, i int) {
=>   9:         fmt.Printf("goroutine id %d\n", i)
    10:         fmt.Printf("goroutine id %d\n", i)
    11:         wg.Done()
    12: }
    13:
    14: func main() {
(dlv) print i
9
(dlv)
```

We have created 10 goroutines executing this function and yet we land on the same goroutine. This is because Delve, being a Go specific debugger, has knowledge of Go specific runtime features such as Goroutines. When you execute the next command, Delve will make sure to put you on the next source line in the context of that goroutine. This prevents the frustrating "thrashing" effect from other tools, where you may end up on a completely different goroutine after using a command like next. Remember that goroutines are invisible to the host OS which only knows about the threads maintained by the Go runtime.

Use the `h` command to get help:

```
(dlv) h
The following commands are available:
    args ------------------------ Print function arguments.
    break (alias: b) ------------ Sets a breakpoint.
    breakpoints (alias: bp) ----- Print out info for active breakpoints.
    clear ----------------------- Deletes breakpoint.
    clearall -------------------- Deletes multiple breakpoints.
```

```
    condition (alias: cond) ----- Set breakpoint condition.
    continue (alias: c) --------- Run until breakpoint or program termination.
    disassemble (alias: disass) - Disassembler.
    exit (alias: quit | q) ------ 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.
    help (alias: h) ------------- Prints the help message.
    list (alias: ls) ------------ Show source code.
    locals ---------------------- Print local variables.
    next (alias: n) ------------- Step over to next source line.
    on -------------------------- Executes a command when a breakpoint is hit.
    print (alias: p) ------------ Evaluate an expression.
    regs ------------------------ Print contents of CPU registers.
    restart (alias: r) ---------- 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 (alias: bt) ----------- Print stack trace.
    step (alias: s) ------------- Single step through program.
    step-instruction (alias: si)  Single step a single cpu instruction.
    stepout --------------------- Step out of the current function.
    thread (alias: tr) ---------- Switch to the specified thread.
    threads --------------------- Print out info for every traced thread.
    trace (alias: t) ------------ Set tracepoint.
    types ----------------------- Print list of types
    vars ------------------------ Print package variables.
Type help followed by a command for full documentation.
(dlv)
```

List the goroutines in your program:

```
(dlv) goroutines
[14 goroutines]
  Goroutine 1 - User: /usr/local/go/src/runtime/sema.go:47 sync.runtime_Semacquire (0x435154)
  Goroutine 2 - User: /usr/local/go/src/runtime/proc.go:272 runtime.gopark (0x427f4a)
  Goroutine 17 - User: /usr/local/go/src/runtime/proc.go:272 runtime.gopark (0x427f4a)
  Goroutine 18 - User: /usr/local/go/src/runtime/proc.go:272 runtime.gopark (0x427f4a)
  Goroutine 19 - User: ./more.go:8 main.dostuff (0x47d5e0)
  Goroutine 20 - User: ./more.go:8 main.dostuff (0x47d5e0)
  Goroutine 21 - User: ./more.go:8 main.dostuff (0x47d5e0)
  Goroutine 22 - User: ./more.go:8 main.dostuff (0x47d5e0)
  Goroutine 23 - User: ./more.go:9 main.dostuff (0x47d60f) (thread 62728)
  Goroutine 24 - User: ./more.go:8 main.dostuff (0x47d5e0)
  Goroutine 25 - User: ./more.go:8 main.dostuff (0x47d5e0)
  Goroutine 26 - User: ./more.go:8 main.dostuff (0x47d5e0)
  Goroutine 27 - User: ./more.go:8 main.dostuff (0x47d5e0)
* Goroutine 28 - User: ./more.go:9 main.dostuff (0x47d60f) (thread 62720)
(dlv)
```

Note that each goroutine has a User ID, a file, a line number, a package, a function and address. Also, if actively executing the goroutine will be assigned a system thread to run on.

The main.main() function always runs on goroutine 1, which presently is in the sync package waiting for a semaphore (this is where the `wg.Wait()` call ended up blocking).

Notice that only two threads have been provisioned by the runtime even though we have 14 goroutines. Also note that the goroutine numbers are increasing but that while 19-28 are sequential (representing our dostuff() workers) there are gaps in other places indicating goroutines that have exited and been collected.

Switch to goroutine 19 and display the thread information for the program:

```
(dlv) goroutine 19
Switched from 28 to 19 (thread 62720)
(dlv) threads
* Thread 62720 at 0x47d60f ./more.go:9 main.dostuff
  Thread 62726 at 0x44e491 /usr/local/go/src/runtime/sys_linux_amd64.s:98 runtime.usleep
  Thread 62727 at 0x44e933 /usr/local/go/src/runtime/sys_linux_amd64.s:426 runtime.futex
  Thread 62728 at 0x47d60f ./more.go:9 main.dostuff
(dlv)
```

List the source code at this location:

```
(dlv) list
> main.dostuff() ./more.go:9 (PC: 0x47d60f)
     4:         "fmt"
```

```
     5:         "sync"
     6: )
     7:
     8: func dostuff(wg *sync.WaitGroup, i int) {
=>   9:         fmt.Printf("goroutine id %d\n", i)
    10:         fmt.Printf("goroutine id %d\n", i)
    11:         wg.Done()
    12: }
    13:
    14: func main() {
(dlv)
```

Ah ha, while the debugger has switched us to the new goroutine the program state is still frozen. Use the next command to advance:

```
(dlv) next
> main.dostuff() ./more.go:8 (hits goroutine(24):1 total:3) (PC: 0x47d5f8)
> main.dostuff() ./more.go:8 (PC: 0x47d5e0)
     3: import (
     4:         "fmt"
     5:         "sync"
     6: )
     7:
=>   8: func dostuff(wg *sync.WaitGroup, i int) {
     9:         fmt.Printf("goroutine id %d\n", i)
    10:         fmt.Printf("goroutine id %d\n", i)
    11:         wg.Done()
    12: }
    13:
(dlv)
```

Stepping the program allowed goroutine 24 to advanced to its breakpoint, reported in the first line of output. The second line begins the dump for our new deubgger goroutine, 19.

Try listing all of the functions loaded:

```
(dlv) funcs
_rt0_amd64_linux
callRet
errors.(*errorString).Error
errors.New
fmt.(*buffer).WriteRune
fmt.(*fmt).fmt_boolean
fmt.(*fmt).fmt_bx
fmt.(*fmt).fmt_c
fmt.(*fmt).fmt_float
fmt.(*fmt).fmt_integer
fmt.(*fmt).fmt_q
fmt.(*fmt).fmt_qc
fmt.(*fmt).fmt_s
fmt.(*fmt).fmt_sbx
fmt.(*fmt).fmt_sx
fmt.(*fmt).fmt_unicode
fmt.(*fmt).pad
fmt.(*fmt).padString

...
```

Now try listing just those in package main:

```
(dlv) funcs main
main
main.dostuff
main.init
main.main
runtime.main
runtime.main.func1
runtime.main.func2
(dlv)
```

You can use the `types` and `vars` commands to diplay package types and variables.

When you finish exploring `continue` the program to completions and `quit`.

CHALLANGE

Restart the debugger and set a break point on the dostuff() function. Force the 3rd goroutine to output 500 on its second Printf.

Congratulations you have completed the lab!!