Debugging and Profiling

Spring 2022

Contents

1 Introduction

This lab is intended to get you more familiar with debugging and profiling in Go, which you may find helpful for the rest of the assignments this year. We will be using delve to debug in Go.

To get started, please use this github classroom link to generate stencil. You can complete this lab either individually or in pair with your Tapestry project partner. Since there are two parts to this lab, there are two directories `debugging` and `profiling` in the stencil. For each section, ***use that folder as your Go module root directory***.

We are doing things differently in this lab; there is no Gradescope assignment to submit your lab report. Instead, you have to come to the TA hours to check off. We will grade the lab based on completion; you will get full credit if you have done the check off!

2 Debugging

Use the `debugging` folder as your Go module root directory for this part of the lab.

2.1 Installation

To install `delve` run the following command:

```
go install github.com/go-delve/delve/cmd/dlv@latest
```

To check that `delve` installed successfully, check its version with `dlv version`.

```
dlv version
Delve Debugger
Version: 1.8.0
Build: Id: SOME_HASH
```

2.2 Getting Started

From the projects root directory, you can invoke `delve` on a Go file with

```
dlv debug <RELATIVE_PATH_TO_GO_FILE>
```

i.e. `dlv debug client/client.go`.

This will start a new debugging session inside `client.go`. We can then type `help` to see all available commands. This README also provides a comprehensive list of all available commands.

2.2.1 Types of Debugging Sessions

There are three ways we can invoke `delve` on Go programs.

- `dlv debug <PATH>`
  This first compiles the source into a binary then starts a debugging session (similar to 'go run')
- `dlv exec <PATH>`
  This expects a pre-compiled binary before starting the debugging session
- `dlv test <PATH>`
  This is used to start a debugging session on a Go test

TIP: If the Go program requires command line arguments, you can pass them into the debugging session as follows:

```
dlv debug <PATH> -- <ARGS>
```

i.e. `dlv debug chat.go -- -server`. Note that in our exercise, we will be using `dlv test`.

2.3 Navigating a Debugging Session

Once you're within a debugging session, here are some ways to navigate your Go program.

2.3.1 Viewing Source Code

- `list <PACKAGE> <FUNCTION>`
  Prints the source code of the given function in the given package

  i.e. `list client.UserReader`

- `list <FILE_PATH> <LINE_NUMBER>`
  Similar to the above command, but we can specify instead a filepath to a Go file and a line number we want to examine

  i.e. `list client/client.go:10`

- `funcs <PATTERN>`
  Finds all functions in the source containing the given pattern

  i.e. `funcs handle`

- `exit` Quits the debugging session

2.3.2 Adding Breakpoints

After examining the program, we can make use of breakpoints to mark locations in the code we want to pause execution to investigate.

- `break <PATH>:<LINE_NUMBER>`
  Adds a **breakpoint** in the given location (determined by filepath and line number). A breakpoint is a location in the code where you want the program to stop to inspect some state (such as the value a variable takes on).

  i.e. `break client/client.go:22`

- `breakpoints`
  Lists all currently active breakpoints in the debugging session

- `clear <NUMBER>`
  Removes the breakpoint with the given number from the debugging session

- `clearall`
  Removes all breakpoints from the debugging session

2.3.3 Running the Program

Once the breakpoints are set, we can now start execution of the program and wait until we hit certain breakpoints to view the state of different variables.

- `continue`
  Runs until a breakpoint is hit or the program terminates.

- `next`
  Moves to the next line of the source code

- `step`
  Steps *inside* to go to the next line of execution (i.e. within a function call)

- `stepout`
  Steps *outside* to go back to the location where a function may have been called
- `restart`
  Restarts the program execution from the start (allows you to debug a program multiple times without losing your breakpoints)

2.3.4 Viewing Variables

When execution is paused (i.e. a breakpoint is hit or `next` / `step` / `stepout` is used), we can examine the values of different variables.

- `print <VARIABLE>` Displays the value of a variable

  i.e. `print users`
- `locals` Displays all local variables

2.4 IDE Debugging

Some IDEs (VSCode, Golang) will have built-in debugging interfaces. Feel free to use these features as well instead of delve if you prefer. These debugging options may have graphical interfaces that provide the same functionality as delve (setting breakpoints, stepping through code, viewing variable values).

2.5 Exercise

Now that you learned how to use `delve`, you can use it for debugging the `debugging` directory which contains broken code for a chat application called UMessage. **HINT:** There are a total of **3 bugs** between the client and server code.

2.5.1 UMessage

UMessage supports sending and receiving messages in a user-to-user manner (like Google Hangouts) instead of broadcast (like IRC or group chats). This means that if three users (Tom, Rodrigo, and Ugur) are connected to the same server, Tom can send a message to Rodrigo that will not be delivered to Ugur. The server is centralized (for simplicity) and has the responsibility of routing messages to and from the connected users of the system.

The directory contains a broken solution for UMessage. Your task is to find three bugs in parts containing client and server logic, `client/client.go` and `server/server.go` respectively, using `delve`. You can assume that other files, including `umessage/proto.go`, `umessage/util.go` and `umessage/listener.go` are correctly implemented as intended.

The following parts describe how UMessage is supposed to be implemented.

2.5.2 Protocol

The `UMessageMsg` struct along with the Purpose enum contained in `umessage/proto.go` define the messages that are exchanged between clients and a server. All messages between a client and server use gob encoding as implemented in `umessage/util.go`.

2.5.3 Message Semantics

The type of message and the meaning of the fields is defined by its `Purpose`, which can be any of the following: `CONNECT, MSG, LIST, ERROR, DISCONNECT`.

- `CONNECT`

  - Description: upon starting a client it sends a message of this type to initiate a connection to the server for future message transport.
  - `UMessageMsg.Username` contains the username of the client attempting to connect with the server.
  - `UMessageMsg.Body` is not defined for this type of message.
- `MSG`
  - Description: a chat msg sent from one a client to another user (may be itself).
  - `UMessageMsg.Username` is the username that the msg is destined for (e.g. Tom).
  - `UMessageMsg.Body` contains the actual msg (e.g. "hello world").
- `LIST`
  - Description: a client can ask the server what users are currently connected.
  - `UMessageMsg.Username` is not defined for this type of message.
  - `UMessageMsg.Body` is empty if this is a request and contains a list of connected users if it is a response.
- `ERROR`
  - Description: if the server cannot satisfy a request for whatever reason it should send an error message back to the client saying why it cannot.
  - `UMessageMsg.Username` is not defined for this type of message.
  - `UMessageMsg.Body` contains the error string
- `DISCONNECT`
  - Description: a client can ask to be disconnected gracefully from the server
  - `UMessageMsg.Username` is not defined for this type of message.
  - `UMessageMsg.Body` is not defined for this type of message.

2.5.4 Client

The server is responsible for conforming to the protocol detailed in the previous section. It's primary responsibility is to route messages between client users.

In order for the server to start it needs to listen for incoming connections on a defined port. This part is done in `umessage/listener.go` and you can assume that it is correctly implemented.

2.5.5 Client

Upon starting up, the UMessage client should connect to a user defined UMessage server. From there it should take user input (e.g. via a command prompt) to send and receive messages between it and the server. Likewise, you can assume that client side correctly behaves to connect to the server.

2.5.6 Getting Started

You may find it helpful to run the client and server in separate terminal windows to observer the behavior of the system. Instructions for how to run this can be found in the README in the repository. We have provided the tests `client_test.go` and `server_test.go` that are currently failing. When you have fixed the bugs in UMessage, these tests should pass. You can run `go test ./...` in the root directory of this module to run both tests.

Note that your IDE (VSCode, Goland, etc.) may have built in debugging features. Feel free to use these instead of delve if you prefer.

If using delve, we recommend debugging on the provided test files rather than chat.go. To run delve on the tests, you can run the following commands on terminal:

1. `dlv test ./client`

   Debugs tests in the `client` package

2. `dlv test ./server`

   Debugs tests in the `client` package

When you have a debugger running on the tests, here are some delve commands that might be helpful:

1. `break client.Test_SendList`

   Sets a breakpoint at the start of the `Test_SendList` test

2. `locals`

   Prints all local variables

3. `continue`

   Starts / resumes execution

4. `step`

   Steps into the function on a given line

5. `next`

   Moves to the next line of code in the given file

2.6 Checkoff Questions

After fixing the broken implementation, please come to lab hours with responses to the following questions:

1. What are the 3 bugs you found in UMessage and how did you discover them?
2. Assume we have a network of a single UMessage server and n UMessage clients. If one client node goes down, how can we identify this outage on the server side? On the client-side?
3. In the same network as above, what if the server went down?

3 Profiling

In this section, you will learn how to profile your program with pprof of a benchmark in Go. `pprof` will be used to analyze and visualize the profiling data created by benchmarks.

Use the `profiling` folder as your Go module root directory for this part of the lab.

3.1 Installation

To install `pprof`, run the following command:

```
go install github.com/google/pprof@latest
```

You also need to install Graphviz to visualize profiled data. It can be installed here.

3.2 Creating profiles

In the previous lab, you learned how to do simple benchmarking of a Go program using the following command on the test directory:

```
go test -bench=.
```

Alternatively, you can run the following command if you are not on the test directory:

```
go test <PATH_TO_THE_TEST_DIRECTORY> -bench=.
```

You can also replace `-bench=.` with `-bench=<NAME>` to benchmark using the following benchmark function:

```
func Benchmark<NAME>(b *testing.B) {
    ...
}
```

To create a memory and CPU profile, you just need to add additional flags. For instance, you can run the following command in the test directory:

```
go test -bench=. -memprofile <MEMPROFILE.out> -cpuprofile <CPUPROFILE.out>
```

It is also possible to profile mutexes and goroutines using the `mutexprofile` and `blockprofile` flags respectively:

1. `-mutexprofile <MUTEX.out>`: Write a mutex contention profile to the specified file when all tests are complete.
2. `-blockprofile <BLOCK.out>`: Write a goroutine blocking profile to the specified file when all tests are complete.

3.3 Viewing the profiles

To investigate profiling results, you can run the following command (again in the Go module root directory):

```
go tool pprof <PATH\_TO\_PROFILE\_OUTFILE>
```

For instance, to view CPU profile information, run `go tool pprof profile.out`.

This will enter pprof's interactive mode. When in interactive mode, you have many commands available to investigate profile data. Here are some that may be useful:

1. `help`: Prints all available commands
2. `top`: Displays top entries in text form
3. `png`: Saves a graph image in PNG format to your disk

3.4 Exercise

Using the above commands, generate the following:

1. CPU profile for (list of top functions + graph) `BenchmarkCalculate`
2. Comparison profile between `BenchmarkPrepend` and `BenchmarkAppend`

   To compare two profiles you will need to:

   (a) Generate *separate* profile files for both benchmarks (Hint: use different names when running the `go test` command

   (b) Use the `--diff_base` flag when invoking `go tool` to specify a profile to be subtracted from the base profile.

   Ex: `go tool pprof --diff_base=memprofileAppend.out memprofilePrepend.out`

   Note: Some percentages may have negative values in your profile output when comparing. This is expected (you can use absolute value when comparing the functions).

Note that you should have separate graphs and separate lists of functions for both benchmarks (be careful when running the `go test` command, the subsequent run will overwrite the original profile files!)

3.5 Checkoff Questions

After generating the profiles, be prepared to answer these questions to the TA during checkoff:

1. What function in `BenchmarkCalculate` takes up the most CPU usage?
2. What function takes up the most memory when comparing `BenchmarkAppend` and `BenchmarkPrepend`?
3. What do the lines between nodes in the graphs represent?
4. (Optional) Comparing the `Append` and `Prepend` function source code, why does the performance of one differ from the other?

Feedback

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS138 document by filling out the anonymous feedback form.