golang调试和性能优化参考

一. 使用pprof调优

在计算机性能调试领域里，profiling 是指对应用程序的画像，画像就是应用程序使用 CPU 和内存的情况。 Go语言是一个对性能特别看重的语言，因此语言中自带了 profiling 的库，这篇文章就要讲解怎么在 golang 中做 profiling。

1.1 Go性能优化

Go语言项目中的性能优化主要有以下几个方面：

- CPU profile：报告程序的 CPU 使用情况，按照一定频率去采集应用程序在 CPU 和寄存器上面的数据
- Memory Profile（Heap Profile）：报告程序的内存使用情况
- Block Profiling：报告 goroutines 不在运行状态的情况，可以用来分析和查找死锁等性能瓶颈
- Goroutine Profiling：报告 goroutines 的使用情况，有哪些 goroutine，它们的调用关系是怎样的

Go语言内置了获取程序的运行数据的工具，包括以下两个标准库：

- `runtime/pprof`：采集工具型应用运行数据进行分析
- `net/http/pprof`：采集服务型应用运行时数据进行分析

pprof开启后，每隔一段时间（10ms）就会收集下当前的堆栈信息，获取格格函数占用的CPU以及内存资源；最后通过对这些采样数据进行分析，形成一个性能分析报告。

注意，我们只应该在性能测试的时候才在代码中引入pprof。

1.2 工具型应用

如果你的应用程序是运行一段时间就结束退出类型。那么最好的办法是在应用退出的时候把 profiling 的报告保存到文件中，进行分析。对于这种情况，可以使用`runtime/pprof`库。 首先在代码中导入`runtime/pprof`工具：

```
import "runtime/pprof"
```

开启CPU性能分析：

```
pprof.StartCPUProfile(w io.Writer)
```

停止CPU性能分析：

```
pprof.StopCPUProfile()
```

应用执行结束后，就会生成一个文件，保存了我们的 CPU profiling 数据。得到采样数据之后，使用`go tool pprof`工具进行CPU性能分析。

记录程序的堆栈信息

```
pprof.WriteHeapProfile(w io.Writer)
```

得到采样数据之后，使用`go tool pprof`工具进行内存性能分析。

`go tool pprof`默认是使用`-inuse_space`进行统计，还可以使用`-inuse-objects`查看分配对象的数量。

1.3 服务型应用

如果你的应用程序是一直运行的，比如 web 应用，那么可以使用`net/http/pprof`库，它能够在提供 HTTP 服务进行分析。

如果使用了默认的`http.DefaultServeMux`（通常是代码直接使用 http.ListenAndServe(“0.0.0.0:8000”, nil)），只需要在你的web server端代码中按如下方式导入`net/http/pprof`

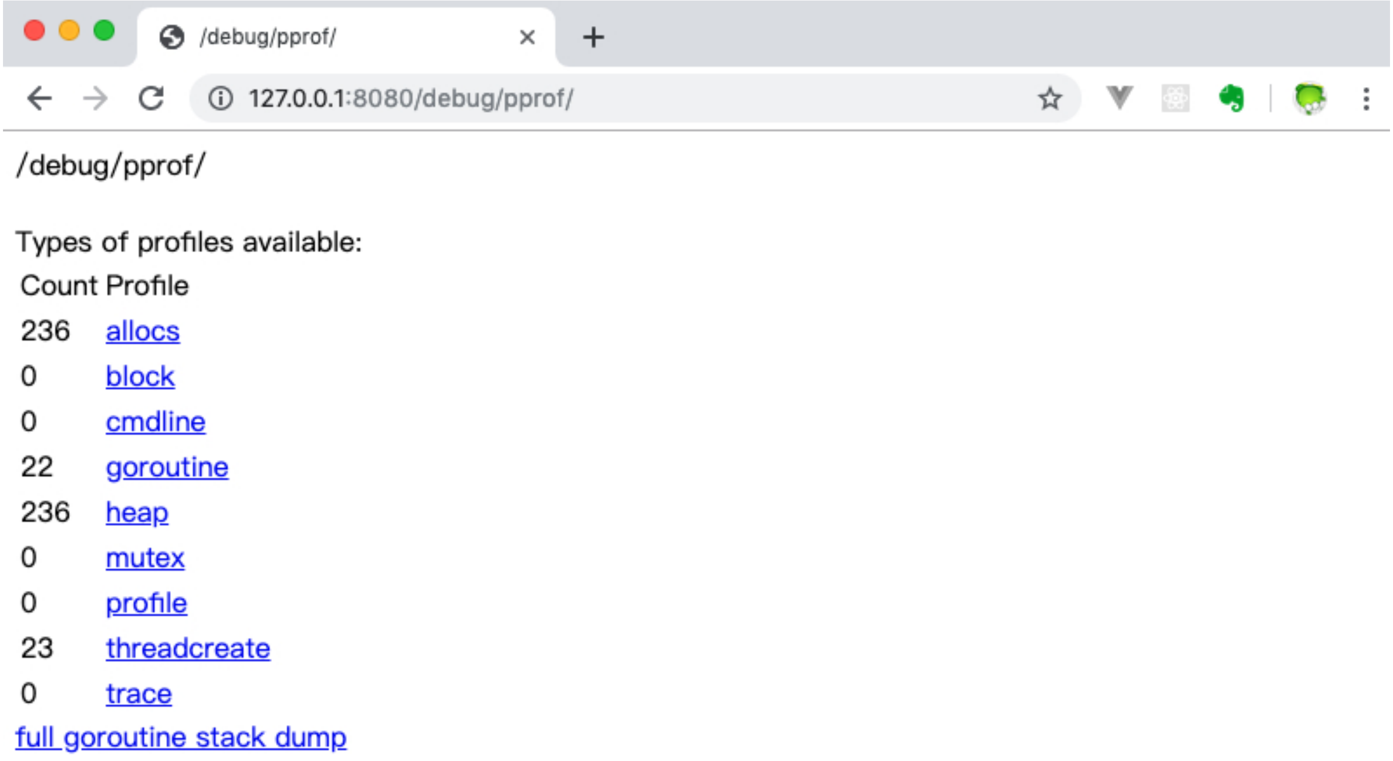
```
import _ "net/http/pprof"
```

如果你使用自定义的 Mux，则需要手动注册一些路由规则：

```
r.HandleFunc("/debug/pprof/", pprof.Index)
r.HandleFunc("/debug/pprof/cmdline", pprof.Cmdline)
r.HandleFunc("/debug/pprof/profile", pprof.Profile)
r.HandleFunc("/debug/pprof/symbol", pprof.Symbol)
r.HandleFunc("/debug/pprof/trace", pprof.Trace)
```

如果你使用的是gin框架，那么推荐使用`"github.com/DeanThompson/ginpprof"`。

不管哪种方式，你的 HTTP 服务都会多出`/debug/pprof` endpoint，访问它会得到类似下面的内容：



/debug/pprof/

Types of profiles available:

Count	Profile
236	allocs
0	block
0	cmdline
22	goroutine
236	heap
0	mutex
0	profile
23	threadcreate
0	trace

full goroutine stack dump

这个路径下还有几个子页面：

- /debug/pprof/profile：访问这个链接会自动进行 CPU profiling，持续 30s，并生成一个文件供下载
- /debug/pprof/heap： Memory Profiling 的路径，访问这个链接会得到一个内存 Profiling 结果的文件
- /debug/pprof/block：block Profiling 的路径
- /debug/pprof/goroutines：运行的 goroutines 列表，以及调用关系

1.4 go tool pprof命令

不管是工具型应用还是服务型应用，我们使用相应的pprof库获取数据之后，下一步的都要对这些数据进行分析，我们可以使用`go tool pprof`命令行工具。

`go tool pprof`的使用方式为:

```
 go tool pprof
usage:

Produce output in the specified format.

   pprof <format> [options] [binary] <source> ...

Omit the format to get an interactive shell whose commands can be used
to generate various views of a profile

   pprof [options] [binary] <source> ...

Omit the format and provide the "-http" flag to get an interactive web
interface at the specified host:port that can be used to navigate through
```

```
various views of a profile.

   pprof -http [host]:[port] [options] [binary] <source> ...

Details:
  Output formats (select at most one):
    -callgrind       Outputs a graph in callgrind format
    -comments        Output all profile comments
    -disasm          Output assembly listings annotated with samples
    -dot             Outputs a graph in DOT format
    -eog             Visualize graph through eog
    -evince          Visualize graph through evince
    -gif             Outputs a graph image in GIF format
    -gv              Visualize graph through gv
    -kcachegrind     Visualize report in KCachegrind
    -list            Output annotated source for functions matching regexp
    -pdf             Outputs a graph in PDF format
    -peek            Output callers/callees of functions matching regexp
    -png             Outputs a graph image in PNG format
    -proto           Outputs the profile in compressed protobuf format
    -ps              Outputs a graph in PS format
    -raw             Outputs a text representation of the raw profile
    -svg             Outputs a graph in SVG format
    -tags            Outputs all tags in the profile
    -text            Outputs top entries in text form
    -top             Outputs top entries in text form
    -topproto        Outputs top entries in compressed protobuf format
    -traces          Outputs all profile samples in text form
    -tree            Outputs a text rendering of call graph
    -web             Visualize graph through web browser
    -weblist         Display annotated source in a web browser

  Options:
    -call_tree       Create a context-sensitive call tree
    -compact_labels  Show minimal headers
    -divide_by       Ratio to divide all samples before visualization
    -drop_negative   Ignore negative differences
    -edgefraction    Hide edges below <f>*total
    -focus           Restricts to samples going through a node matching regexp
    -hide            Skips nodes matching regexp
    -ignore          Skips paths going through any nodes matching regexp
    -intel_syntax    Show assembly in Intel syntax
    -mean            Average sample value over first value (count)
    -nodecount       Max number of nodes to show
    -nodefraction    Hide nodes below <f>*total
    -noinlines       Ignore inlines.
    -normalize       Scales profile based on the base profile.
    -output          Output filename for file-based outputs
    -prune_from      Drops any functions below the matched frame.
    -relative_percentages Show percentages relative to focused subgraph
    -sample_index     Sample value to report (0-based index or name)
    -show             Only show nodes matching regexp
    -show_from        Drops functions above the highest matched frame.
    -source_path      Search path for source files
    -tagfocus         Restricts to samples with tags in range or matched by regexp
    -taghide          Skip tags matching this regexp
    -tagignore        Discard samples with tags in range or matched by regexp
    -tagshow          Only consider tags matching this regexp
    -trim             Honor nodefraction/edgefraction/nodecount defaults
    -trim_path        Path to trim from source paths before search
    -unit             Measurement units to display

  Option groups (only set one per group):
    granularity
      -functions       Aggregate at the function level.
      -filefunctions   Aggregate at the function level.
      -files           Aggregate at the file level.
      -lines           Aggregate at the source code line level.
      -addresses       Aggregate at the address level.
    sort
      -cum             Sort entries based on cumulative weight
      -flat            Sort entries based on own weight

  Source options:
    -seconds              Duration for time-based profile collection
    -timeout              Timeout in seconds for profile collection
    -buildid              Override build id for main binary
    -add_comment          Free-form annotation to add to the profile
                          Displayed on some reports or with pprof -comments
    -diff_base source     Source of base profile for comparison
    -base source          Source of base profile for profile subtraction
    profile.pb.gz         Profile in compressed protobuf format
    legacy_profile        Profile in legacy pprof format
    http://host/profile   URL for profile handler to retrieve
    -symbolize=           Controls source of symbol information
      none                  Do not attempt symbolization
      local                 Examine only local binaries
      fastlocal             Only get function names from local binaries
      remote                Do not examine local binaries
      force                 Force re-symbolization
    Binary                  Local path or build id of binary for symbolization
    -tls_cert             TLS client certificate file for fetching profile and symbols
    -tls_key              TLS private key file for fetching profile and symbols
    -tls_ca               TLS CA certs file for fetching profile and symbols

  Misc options:
   -http              Provide web interface at host:port.
                      Host is optional and 'localhost' by default.
```

```
                      Port is optional and a randomly available port by default.
   -no_browser        Skip opening a browser for the interactive web UI.
   -tools             Search path for object tools

  Legacy convenience options:
   -inuse_space           Same as -sample_index=inuse_space
   -inuse_objects         Same as -sample_index=inuse_objects
   -alloc_space           Same as -sample_index=alloc_space
   -alloc_objects         Same as -sample_index=alloc_objects
   -total_delay           Same as -sample_index=delay
   -contentions           Same as -sample_index=contentions
   -mean_delay            Same as -mean -sample_index=delay

  Environment Variables:
   PPROF_TMPDIR       Location for saved profiles (default $HOME/pprof)
   PPROF_TOOLS        Search path for object-level tools
   PPROF_BINARY_PATH  Search path for local binary files
                      default: $HOME/pprof/binaries
                      searches $name, $path, $buildid/$name, $path/$buildid
   * On Windows, %USERPROFILE% is used instead of $HOME
no profile source specified
```

其中：

- binary 是应用的二进制文件，用来解析各种符号；
- source 表示 profile 数据的来源，可以是本地的文件，也可以是 http 地址。

注意事项： 获取的 Profiling 数据是动态的，要想获得有效的数据，请保证应用处于较大的负载（比如正在生成中运行的服务，或者通过其他工具模拟访问压力）。否则如果应用处于空闲状态，得到的结果可能没有任何意义。

具体示例

首先我们来写一段有问题的代码：

```
// runtime_pprof/main.go
package main

import (
  "flag"
  "fmt"
  "os"
  "runtime/pprof"
  "time"
)

// 一段有问题的代码
func logicCode() {
  var c chan int
  for {
    select {
```

```
    case v := <-c:
      fmt.Printf("recv from chan, value:%v\n", v)
    default:

    }
  }
}

func main() {
  var isCPUPprof bool
  var isMemPprof bool

  flag.BoolVar(&isCPUPprof, "cpu", false, "turn cpu pprof on")
  flag.BoolVar(&isMemPprof, "mem", false, "turn mem pprof on")
  flag.Parse()

  if isCPUPprof {
    file, err := os.Create("./cpu.pprof")
    if err != nil {
      fmt.Printf("create cpu pprof failed, err:%v\n", err)
      return
    }
    pprof.StartCPUProfile(file)
    defer pprof.StopCPUProfile()
  }
  for i := 0; i < 8; i++ {
    go logicCode()
  }
  time.Sleep(20 * time.Second)
  if isMemPprof {
    file, err := os.Create("./mem.pprof")
    if err != nil {
      fmt.Printf("create mem pprof failed, err:%v\n", err)
      return
    }
    pprof.WriteHeapProfile(file)
    file.Close()
  }
}
```

通过flag我们可以在命令行控制是否开启CPU和Mem的性能分析。 将上面的代码保存并编译成`runtime_pprof`可执行文件，执行时加上`-cpu`命令行参数如下：

```
./runtime_pprof -cpu
```

等待30秒后会在当前目录下生成一个`cpu.pprof`文件。

交互模式

我们使用go工具链里的`pprof`来分析一下。

```
go tool pprof cpu.pprof
```

执行上面的代码会进入交互界面如下：

```
runtime_pprof $ go tool pprof cpu.pprof
Type: cpu
Time: Jun 28, 2019 at 11:28am (CST)
Duration: 20.13s, Total samples = 1.91mins (568.60%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof)
```

我们可以在交互界面输入`top3`来查看程序中占用CPU前3位的函数：

```
(pprof) top3
Showing nodes accounting for 100.37s, 87.68% of 114.47s total
Dropped 17 nodes (cum <= 0.57s)
Showing top 3 nodes out of 4
      flat  flat%   sum%        cum   cum%
    42.52s 37.15% 37.15%     91.73s 80.13%  runtime.selectnbrecv
    35.21s 30.76% 67.90%     39.49s 34.50%  runtime.chanrecv
    22.64s 19.78% 87.68%    114.37s 99.91%  main.logicCode
```

其中：

- flat：当前函数占用CPU的耗时
- flat：:当前函数占用CPU的耗时百分比
- sun%：函数占用CPU的耗时累计百分比
- cum：当前函数加上调用当前函数的函数占用CPU的总耗时
- cum%：当前函数加上调用当前函数的函数占用CPU的总耗时百分比
- 最后一列：函数名称

在大多数的情况下，我们可以通过分析这五列得出一个应用程序的运行情况，并对程序进行优化。

我们还可以使用`list 函数名`命令查看具体的函数分析，例如执行`list logicCode`查看我们编写的函数的详细分析。

```
(pprof) list logicCode
Total: 1.91mins
ROUTINE ================ main.logicCode in .../runtime_pprof/main.go
    22.64s   1.91mins (flat, cum) 99.91% of Total
         .          .     12:func logicCode() {
         .          .     13:   var c chan int
         .          .     14:   for {
         .          .     15:           select {
         .          .     16:           case v := <-c:
```

```
     22.64s    1.91mins      17:                    fmt.Printf("recv from chan,
value:%v\n", v)
          .           .      18:            default:
          .           .      19:
          .           .      20:            }
          .           .      21:    }
          .           .      22:}
```

通过分析发现大部分CPU资源被17行占用，我们分析出select语句中的default没有内容会导致上面的`case v:=<-c:`一直执行。我们在default分支添加一行`time.Sleep(time.Second)`即可。

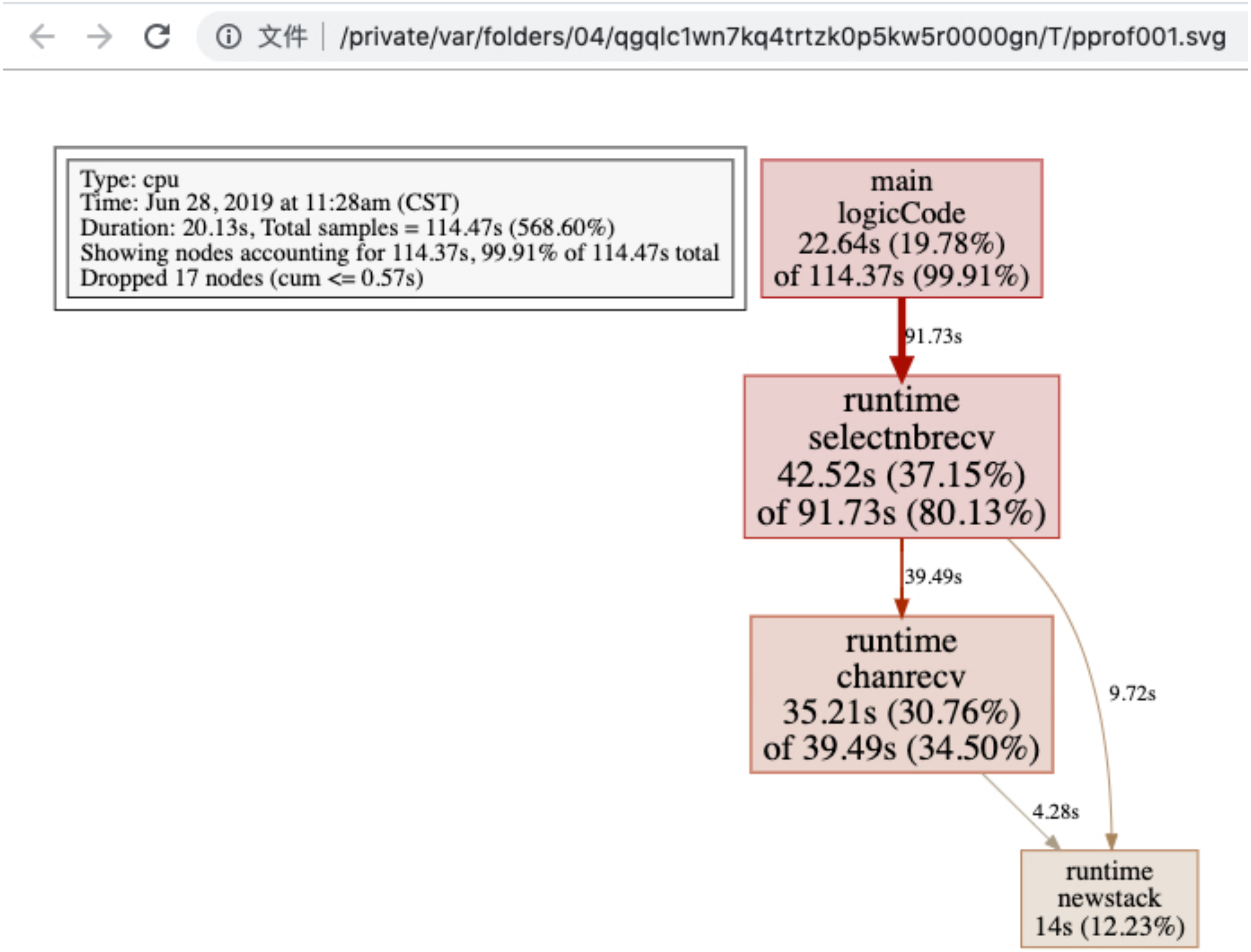
图形化

或者可以直接输入web，通过svg图的方式查看程序中详细的CPU占用情况。 想要查看图形化的界面首先需要安装graphviz图形化工具。

Mac：

```
brew install graphviz
```

Windows: 下载graphviz 将`graphviz`安装目录下的bin文件夹添加到Path环境变量中。 在终端输入`dot -version`查看是否安装成功。



文件
/private/var/folders/04/qgqlc1wn7kq4trtzk0p5kw5r0000gn/T/pprof001.svg

Type: cpu
Time: Jun 28, 2019 at 11:28am (CST)
Duration: 20.13s, Total samples = 114.47s (568.60%)
Showing nodes accounting for 114.37s, 99.91% of 114.47s total
Dropped 17 nodes (cum <= 0.57s)

关于图形

的说明： 每个框代表一个函数，理论上框的越大表示占用的CPU资源越多。 方框之间的线条代表函数之间的调用关系。 线条上的数字表示函数调用的次数。 方框中的第一行数字表示当前函数占用CPU的百分比，第二行数字表示当前函数累计占用CPU的百分比。

1.5 pprof与性能测试结合

`go test`命令有两个参数和 pprof 相关，它们分别指定生成的 CPU 和 Memory profiling 保存的文件：

- -cpuprofile：cpu profiling 数据要保存的文件地址
- -memprofile：memory profiling 数据要报文的文件地址

我们还可以选择将pprof与性能测试相结合，比如：

比如下面执行测试的同时，也会执行 CPU profiling，并把结果保存在 cpu.prof 文件中：

```
go test -bench . -cpuprofile=cpu.prof
```

比如下面执行测试的同时，也会执行 Mem profiling，并把结果保存在 cpu.prof 文件中：

```
go test -bench . -memprofile=./mem.prof
```

需要注意的是，Profiling 一般和性能测试一起使用，这个原因在前文也提到过，只有应用在负载高的情况下 Profiling 才有意义。

二、使用dlv工具

dlv是专门针对go的调试工具，项目地址是 https://github.com/go-delve/delve。

更多了解请参考：Architecture of Delve slides.

2.1 概述

我们需要先安装 Go delve，若是 Go1.16 及以后的版本，可以执行下述命令安装：

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

也可以通过 git clone 的方式安装：

```
$ git clone https://github.com/go-delve/delve
$ cd delve
$ go install github.com/go-delve/delve/cmd/dlv
```

在安装完毕后，我们执行 `dlv version` 命令，查看安装情况：

```
$ dlv version
Delve Debugger
Version: 1.7.0
Build: $Id: e353a65161e6ed74952b96bbb62ebfc56090832b $
```

可以明确看到我们所安装的版本是 v1.7.0， 参考installation

常用Options参考

```
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.
  dap         [EXPERIMENTAL] Starts a headless TCP server communicating via Debug Adaptor Protocol (DAP).
  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.
      --allow-non-terminal-interactive   Allows interactive sessions of Delve that
don't have a terminal as stdin, stdout and stderr
      --api-version int                  Selects API version when headless. New clients
should use v2. Can be reset via RPCServer.SetApiVersion. See Documentation/api/json-
rpc/README.md. (default 1)
      --backend string                   Backend selection (see 'dlv help backend').
(default "default")
      --build-flags string               Build flags, to be passed to the compiler. For
example: --build-flags="-tags=integration -mod=vendor -cover -v"
      --check-go-version                 Checks that the version of Go in use is
compatible with Delve. (default true)
      --disable-aslr                     Disables address space randomization
      --headless                         Run debug server only, in headless mode.
  -h, --help                             help for dlv
      --init string                      Init file, executed by the terminal client.
  -l, --listen string                    Debugging server listen address. (default
"127.0.0.1:0")
      --log                              Enable debugging server logging.
      --log-dest string                  Writes logs to the specified file or file
descriptor (see 'dlv help log').
      --log-output string                Comma separated list of components that should
produce debug output (see 'dlv help log')
      --only-same-user                   Only connections from the same user that
started this instance of Delve are allowed to connect. (default true)
  -r, --redirect stringArray             Specifies redirect rules for target process
(see 'dlv help redirect')
      --wd string                        Working directory for running the program.

Additional help topics:
  dlv backend  Help about the --backend flag.
  dlv log      Help about logging flags.
  dlv redirect Help about file redirection.

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

Available Commands详见：

- dlv attach - Attach to running process and begin debugging.
- dlv connect - Connect to a headless debug server.
- dlv core - Examine a core dump.
- dlv dap - [EXPERIMENTAL] Starts a headless TCP server communicating via Debug Adaptor Protocol

(DAP).

- dlv debug - Compile and begin debugging main package in current directory, or the package specified.
- dlv exec - Execute a precompiled binary, and begin a debug session.
- dlv replay - Replays a rr trace.
- dlv run - Deprecated command. Use 'debug' instead.
- dlv test - Compile test binary and begin debugging program.
- dlv trace - Compile and begin tracing program.
- dlv version - Prints version.
- dlv log - Help about logging flags
- dlv backend - Help about the `--backend` flag

2.2 使用dlv调试

2.2.1 配置和命令历史

If `$XDG_CONFIG_HOME` is set, then configuration and command history files are located in `$XDG_CONFIG_HOME/dlv`. Otherwise, they are located in `$HOME/.config/dlv` on Linux and `$HOME/.dlv` on other systems.

2.2.2 交互调试命令

Running the program

Command	Description
call	Resumes process, injecting a function call (EXPERIMENTAL!!!)
continue	Run until breakpoint or program termination.
next	Step over to next source line.
rebuild	Rebuild the target executable and restarts it. It does not work if the executable was not built by delve.
restart	Restart process.
rev	Reverses the execution of the target program for the command specified.
rewind	Run backwards until breakpoint or program termination.
step	Single step through program.
step-instruction	Single step a single cpu instruction.
stepout	Step out of the current function.

Manipulating breakpoints

Command	Description
break	Sets a breakpoint.
breakpoints	Print out info for active breakpoints.
clear	Deletes breakpoint.
clearall	Deletes multiple breakpoints.
condition	Set breakpoint condition.
on	Executes a command when a breakpoint is hit.
toggle	Toggles on or off a breakpoint.
trace	Set tracepoint.
watch	Set watchpoint.

Viewing program variables and memory

Command	Description
args	Print function arguments.
display	Print value of an expression every time the program stops.
examinemem	Examine raw memory at the given address.
locals	Print local variables.
print	Evaluate an expression.
regs	Print contents of CPU registers.
set	Changes the value of a variable.
vars	Print package variables.
whatis	Prints type of an expression.

Listing and switching between threads and goroutines

Command	Description
goroutine	Shows or changes current goroutine
goroutines	List program goroutines.
thread	Switch to the specified thread.
threads	Print out info for every traced thread.

Viewing the call stack and selecting frames

Command	Description
deferred	Executes command in the context of a deferred call.
down	Move the current frame down.
frame	Set the current frame, or execute command on a different frame.
stack	Print stack trace.
up	Move the current frame up.

Other commands

Command	Description
check	Creates a checkpoint at the current position.
checkpoints	Print out info for existing checkpoints.
clear-checkpoint	Deletes checkpoint.
config	Changes configuration parameters.
disassemble	Disassembler.
dump	Creates a core dump from the current process state
edit	Open where you are in $DELVE_EDITOR or $EDITOR
exit	Exit the debugger.
funcs	Print list of functions.
help	Prints the help message.
libraries	List loaded dynamic libraries
list	Show source code.
source	Executes a file containing a list of delve commands
sources	Print list of source files.
types	Print list of types

具体可参考cli

2.2.3 调试

delve可以调试应用，也可以调试test。

调试应用

project layout:

```
.
├── github.com/me/foo
├── cmd
│   └── foo
│       └── main.go
├── pkg
│   └── baz
│       ├── bar.go
│       └── bar_test.go
```

可以使用dlv debug github.com/me/foo/cmd/foo -- -arg1 value来调试foo

```
$ dlv debug github.com/me/foo/cmd/foo
Type 'help' for list of commands.
(dlv) break main.main
Breakpoint 1 set at 0x49ecf3 for main.main() ./test.go:5
(dlv) continue
> main.main() ./test.go:5 (hits goroutine(1):1 total:1) (PC: 0x49ecf3)
     1: package main
     2:
     3: import "fmt"
     4:
=>   5: func main() {
     6:   fmt.Println("delve test")
     7: }
(dlv)
```

调试test

可以使用dlv test调试test。

```
$ dlv test github.com/me/foo/pkg/baz
Type 'help' for list of commands.
(dlv) funcs test.Test*
/home/me/go/src/github.com/me/foo/pkg/baz/test.TestHi
(dlv) break TestHi
Breakpoint 1 set at 0x536513 for
/home/me/go/src/github.com/me/foo/pkg/baz/test.TestHi() ./test_test.go:5
(dlv) continue
> /home/me/go/src/github.com/me/foo/pkg/baz/test.TestHi() ./bar_test.go:5 (hits
goroutine(5):1 total:1) (PC: 0x536513)
     1: package baz
     2:
     3: import "testing"
     4:
=>   5: func TestHi(t *testing.T) {
     6:   t.Fatal("implement me!")
     7: }
(dlv)
```

参考：https://github.com/go-delve/delve/blob/master/Documentation/cli/getting_started.md

一个参考例子：一个 Demo 学会使用 Go Delve 调试

2.3 插件

The following editor plugins for delve are available:

Atom

- Go Debugger for Atom

Emacs

- Emacs plugin
- dap-mode

Goland

- JetBrains Goland

IntelliJ IDEA

- Golang Plugin for IntelliJ IDEA

LiteIDE

- LiteIDE

Vim

- vim-go (both Vim and Neovim)
- vim-delve (both Vim and Neovim)
- vim-godebug (only Neovim)
- vimspector

VisualStudio Code

- Go for Visual Studio Code

Sublime

- Go Debugger for Sublime

三、使用goland调试

goland下调试go程序比较方便，功能强大。在解决程序bug时，非常直观和高效帮助定位问题。

网上文章比较多，该主题不做描述。

参考官方文档：

https://blog.jetbrains.com/go/2019/02/06/debugging-with-goland-getting-started/

https://blog.jetbrains.com/go/2019/02/14/debugging-with-goland-essentials/

https://www.jetbrains.com/help/go/2021.1/debugging-code.html

四、参考

Go pprof性能调优

https://github.com/go-delve/delve

https://blog.jetbrains.com/go/2019/02/06/debugging-with-goland-getting-started/

https://blog.jetbrains.com/go/2019/02/14/debugging-with-goland-essentials/

https://www.jetbrains.com/help/go/2021.1/debugging-code.html

一个 Demo 学会使用 Go Delve 调试