

GDBSh - командная оболочка для GDB

(version 0.32)

Alexander Sychev (santucco@gmail.com)

1. Введение.

Отладчик GDB не имеет некоторых команд, нужных в повседневном использовании, например поиска адресов по памяти процесса. Да, есть команда `find`, позволяющая искать в регионе памяти, но процесс занимает не сплошной кусок памяти, а множество отдельных секций, сканировать в каждой из которых нужно отдельно. В принципе, GDB можно расширить с помощью скриптов на Python, но эти скриптовые возможности ограничены. Так возникла идея использовать механизм GDB/MI для запуска в GDB внешних команд. Впоследствии захотелось выстраивать команды в конвейер, так получился GDBSh

2. Реализация.

GDBSh запускает GDB в режиме интерпретатора, передает на выполнение команды от порождаемых процессов и возвращает в процессы результат выполнения команд. Каждый дочерний процесс получает, кроме стандартных дескрипторов, еще два файловых дескриптора для взаимодействия с GDB через механизм GDB/MI. GDBSh передает команды в GDB строго поочередно, после завершения выполнения предыдущей команды - это обусловлено невозможностью отличить вывод GDB для разных команд.

```
// This file is part of GDBSh toolset
// Author Alexander Sychev
//
// Copyright (c) 2015, 2016, 2018 Alexander Sychev. All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are
// met:
//
// * Redistributions of source code must retain the above copyright
// notice, this list of conditions and the following disclaimer.
// * Redistributions in binary form must reproduce the above
// copyright notice, this list of conditions and the following disclaimer
// in the documentation and/or other materials provided with the
// distribution.
// * The name of author may not be used to endorse or promote products derived from
// this software without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
// "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
// LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
// OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
// SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
// LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
// DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
// THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
// OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
package main
import(
    <Импортируемые пакеты 3>
)
type(
    <Типы 10>
)
var(
    <Глобальные переменные 4>
)
func main(){
    <Проверить аргументы командной строки, вывести информацию о программе, если необходимо 52>
    <Подготовить трассировку 41>
    <Инициализация сигнальных обработчиков 30>
    <Запустить GDB 6>
    <Читать команды из stdin, посылать их в GDB, обрабатывать результаты 12>
```

```

    < Ждать завершения процесса 7 >
    < Проверить возвращаемый результат 8 >
}

```

3.

```

< Импортируемые пакеты 3 > ≡
    "os"
    "os/exec"
    "io"

```

Также смотри секции 9, 13, 29, 39, 42, 53, и 58.

Используется в секции 2.

4.

```

< Глобальные переменные 4 > ≡
    gdbin io.WriteCloser
    gdbout io.ReadCloser
    gdberr io.ReadCloser
    cmd * exec.Cmd

```

Также смотри секции 11, 18, 40, 43, 47, и 54.

Используется в секции 2.

5. Аргументы командной строки дополняются опцией для вызова интерпретатора GDB

```

< Подготовить аргументы командной строки для запуска GDB 5 > ≡
    var args []string
    args = append(args, os.Args...)
    args[0] = "--interpreter=mi"

```

Используется в секции 6.

6.

⟨Запустить GDB 6⟩ ≡

```
{
  ⟨Подготовить аргументы командной строки для запуска GDB 5⟩
  if cmd = exec.Command("gdb", args...); cmd == nil {
    glog.Errorf("can't create command to run gdb\n")
    return
  }
  var err error
  if gdbin, err = cmd.StdinPipe(); err != nil {
    glog.Errorf("can't create pipe: %v\n", err)
    return
  }
  defer gdbin.Close()
  if gdbout, err = cmd.StdoutPipe(); err != nil {
    glog.Errorf("can't create pipe: %v\n", err)
    return
  }
  defer gdbout.Close()
  if gdberr, err = cmd.StderrPipe(); err != nil {
    glog.Errorf("can't create pipe: %v\n", err)
    return
  }
  defer gdberr.Close()
  if err = cmd.Start(); err != nil {
    glog.Errorf("can't start gdb: %v\n", err)
    return
  }
}
```

Используется в секции 2.

7.

⟨Ждать завершения процесса 7⟩ ≡

```
{
  cmd.Wait()
}
```

Используется в секциях 2 и 19.

8.

⟨Проверить возвращаемый результат 8⟩ ≡

```
{
  if !cmd.ProcessState.Success() {
    fmt.Fprintf(os.Stderr, "\n%s has finished with an error: %s\n", cmd.Path, cmd.ProcessState)
  }
}
```

Используется в секции 2.

9.

⟨Импортируемые пакеты 3⟩ +≡
"fmt"
"bufio"
"strings"

10.

⟨Типы 10⟩ ≡
request **struct**{
 pid **int**
 out *io.WriterCloser*
 cmd **string**
}

Также смотри секции 20 и 32.

Используется в секции 2.

11.

⟨Глобальные переменные 4⟩ +≡
togdbch = **make**(**chan interface{}**)
fromgdbch = **make**(**chan string**)
ackch = **make**(**chan bool**)

12. Запускаем параллельные обработки ввода/вывода от GDB и организуем синхронное выполнение команд

```
< Читать команды из stdin, посылать их в GDB, обрабатывать результаты 12 > ≡  
 {  
   < Запустить параллельную обработку вывода из GDB 16 >  
   < Запустить параллельную обработку ввода из stdin 17 >  
   rp := strings.NewReplacer("\\n", "\n", "\\t", "t", "\\\" \"\", "\"")  
   devnull, _ := os.Open(os.DevNull)  
   var file io.WriteCloser = os.Stdout  
   < Подготовить синхронизацию выполнения команд 48 >  
 loop:  
 for true {  
 select{  
 case s, ok :=← fromgdbch:  
 if ¬ok {  
 break loop  
 }  
 glog.V(debug).Infof("from_gdb:%d\n", s)  
 < Обработка и отправка s в file 14 >  
 case v, ok :=← togdbch:  
 if ¬ok {  
 break loop  
 }  
 switch r := v.(type) {  
 case request:  
 glog.V(debug).Infof("to_gdb_from_%d\n", r.pid, r.cmd)  
 file = r.out  
 c := strings.TrimSpace(r.cmd)  
 if strings.HasPrefix(c, "- ") {  
 c = fmt.Sprintf("%d%s\n", r.pid, c)  
 } else {  
 c = fmt.Sprintf("%d-interpreter-exec_console\n", r.pid, c)  
 }  
 io.WriteString(gdbin, c)  
 case string:  
 glog.V(debug).Infof("to_gdb:%d\n", r)  
 io.WriteString(gdbin, r + "\n")  
 }  
 }  
 }
```

Используется в секции 2.

13.

```

⟨Импортируемые пакеты 3⟩ +=
  "unicode"
  "strconv"
  "bitbucket.org/santucco/gdbsh/common"

```

14. Сначала нужно проверить, нет ли в начале полученной строки идентификатора процесса, который сигнализирует о окончании выполнения команды. Если идентификатор есть, отправляем результат в соответствующий процесс, переключаем текущий вывод на *stdout* и разрешаем выполнение следующей команды. Если полученная строка содержит "^running", то запрещаем ввод команд до появления строки "*stopped". Если идентификатора нет, строка выводится в соответствующий процесс или в *stdout*, причем в последнем случае она проходит предварительную обработку. Приглашение "(gdb)", получаемое от GDB, не выводится, поскольку за приглашение отвечает пакет *readline*. При получении первого приглашения мы разрешаем ввод.

⟨ Обработка и отправка *s* в *file* 14 ⟩ ≡

```
{
  if len(s) ≡ 0 {
    continue
  }
  i := 0
  var r rune
  for i, r = range s {
    if ¬unicode.IsDigit(r) {
      break
    }
  }
  if p, err := strconv.Atoi(s[i]); err ≡ nil {
    s = s[i:]
    if strings.HasPrefix(s, "^running") {
      ⟨ Захватить ввод 46 ⟩
    }
    glog.V(debug).Infof("writing to process %d: %s", p, s)
    if n, err := io.WriteString(file, s); err ≠ nil ∨ n ≠ len(s) {
      glog.V(debug).Infof("can't write %s to output, %d bytes has been written: %s", s, n, err)
    }
    file = os.Stdout
    glog.Flush()
    ⟨ Разрешение выполнения следующей команды 49 ⟩
    continue
  }
  if strings.HasPrefix(s, "*stopped") {
    ⟨ Разрешить ввод 44 ⟩
  }
  ⟨ Обработать строки для вывода в os.Stdout 15 ⟩
  glog.V(debug).Infof("sending: %s", s)
  if n, err := io.WriteString(file, s); err ≠ nil ∨ n ≠ len(s) {
    glog.V(debug).Infof("can't write %s to output, %d bytes has been written: %s", s, n, err)
    file = devnull
  }
}
```

Используется в секции 12.

15.

```

⟨ Обработать строки для вывода в os.Stdout 15 ⟩ ≡
{
  if file ≡ os.Stdout {
    glog.V(debug).Infof("preprocessing_for_stdout: '%s'", s)
    switch s[0] {
      case '^', '&':
        s = s[2:len(s) - 2]
      case '^!':
        if strings.HasPrefix(s, "^error") {
          s = s[6:]
          if len(s) ≡ 0 ∨ s[0] ≠ ' ' {
            continue
          }
          v, _, ok := common.ParseResult(s[1:])
          glog.Errorf("%v\n", v)
          if ok ∧ len(v) ≠ 0 ∧ v[0].Name ≡ "msg" {
            s = fmt.Sprintf("%s\n", v[0].Val.(string))
          }
        } else if strings.HasPrefix(s, "^done") {
          continue
        } else {
          continue
        }
      case '(':
        if strings.HasPrefix(s, "(gdb)") {
          ⟨ Однократно выполнить операции инициализации при загруженном GDB 45 ⟩
        }
        continue
      case '*':
        continue
      case '=':
        continue
    }
    s = rp.Replace(s)
  }
}

```

Используется в секции 14.

16.

```

⟨ Запустить параллельную обработку вывода из GDB 16 ⟩ ≡
go func(){
  gdb := bufio.NewReader(gdbout)
  for s, err := gdb.ReadString(' \n'); err ≡ nil; s, err = gdb.ReadString(' \n') {
    glog.V(debug).Infof("%s", s)
    fromgdbch ← s
  }
  close(fromgdbch)
}()

```

Используется в секции 12.

17. Пакет *readline* осуществляет поддержку получения ввода с клавиатуры, историю и автозавершение команд. Если введенная команда пустая, используется предыдущая команда, хранящаяся в *prev*

⟨Запустить параллельную обработку ввода из *stdin* 17⟩ ≡

```

go func(){
    prev := ""
    ⟨Захватить ввод 46⟩
    ⟨Создать экземпляр readline 55⟩
    for{
        s, err := rl.Readline()
        ⟨Разрешить ввод 44⟩
        if err ≠ nil { // io.EOF
            break
        }
        glog.V(debug).Infof("entered_text:_%s", s)
        if len(s) ≡ 0 {
            s = prev
        }
        var stdout io.WriteCloser = os.Stdout
        ⟨Запуск команд с выводом в stdout 19⟩
        prev = s
        rl.SetPrompt("gdbsh$ ")
        ⟨Захватить ввод 46⟩
    }
    glog.V(debug).Infof("on_exit")
    togdbch ← "-gdb-exit"
}()

```

Используется в секции 12.

18.

⟨Глобальные переменные 4⟩ +≡

```

cmds = map[string]string{⟨Зарезервированные команды GDB 60⟩⟨Дополнительные встроенные
    команды 51⟩}

```

19. Запускаем конвейер команд, начиная с последней, затем ждем окончания всех команд и отправляем запросы на удаление из списка процессоров

```

⟨Запуск команд с выводом в stdout 19⟩ ≡
{
  f := func(r rune) bool{
    return r == '|'
  }
  cl := FieldsFunc(strings.TrimSpace(s), f)
  glog.V(debug).Infof("commands:_%#v", cl)
  var cnv []Cmd
  var toclose []io.Closer
  for i := len(cl) - 1; i ≥ 0; i-- {
    first := i == 0
    c := cl[i]
    ⟨Запустить команду c на выполнение и поместить ее в cnv 21⟩
  }
  for _, cmd := range cnv {
    if v, ok := cmd.(*exec.Cmd); ok {
      glog.V(debug).Infof("waiting_for_process_%s_with_pid_%d_is_finished", v.Path, v.Process.Pid)
    }
    ⟨Ждать завершения процесса 7⟩
    if v, ok := cmd.(*exec.Cmd); ok {
      glog.V(debug).Infof("process_%s_with_pid_%d_has_finished", v.Path, v.Process.Pid)
    }
  }
}

```

Используется в секциях 17 и 59.

20. Так как команды могут быть как внутренними, так и внешними, создадим унифицированный интерфейс *Cmd*, подогнанный к функциям *exec.Cmd*. Таким образом можно будет запускать и ожидать окончания выполнения команд одинаковым образом

```

⟨Типы 10⟩ +≡
Cmd interface{
  Start() error
  Wait() error
}

```

21.

```

⟨Запустить команду c на выполнение и поместить ее в cnv 21⟩ ≡
{
  var cmd Cmd
  var toadb io.ReadCloser
  var fromgdb io.WriteCloser
  ⟨Определить запускаемую команду и создать cmd 22⟩
  ⟨Запустить cmd и добавить в список команд 27⟩
}

```

Используется в секции 19.

22. Надо разделить команду на поля, взять первое и поискать его в *cmds*. Если команда не найдется, надо поискать путь до нее в \$PATH. Если команда найдется с непустым путем или не найдется, ее нужно запустить как внешнюю команду.

```

⟨ Определить запускаемую команду и создать cmd 22 ⟩ ≡
{ n := strings.TrimSpace(c)
  if i := strings.IndexFunc(n, unicode.IsSpace); i ≠ -1 {
    n = n[:i]
  }
  if _, ok := cmds[n];
  ¬ok { ⟨ Создать внешний процесс 23 ⟩ } else {
    ⟨ Создать внутреннюю команду 35 ⟩
  }
}

```

Используется в секции 21.

23.

```

⟨ Создать внешний процесс 23 ⟩ ≡
{
  var ar []string
  ar = append(ar, "sh", "-c", c)
  glog.V(debug).Infof("command_arguments:_%#v", ar)
  c := exec.Command("/usr/bin/env", ar...)
  if c == nil {
    glog.Errorf("can't_create_command_to_run_%s\n", n)
    break
  }
  cmd = c
}

```

Используется в секции 22.

24. Определим расширенную функцию разбиения на поля с учетом экранирования символов и неделимых строковых аргументов

```
func FieldsFunc(s string, f func(rune) bool) []string{
    opened := false
    openedd := false
    escaped := false
    ff := func(r rune) bool{
        if ¬opened ∧ ¬openedd ∧ ¬escaped ∧ f(r) {
            return true
        }
        if r ≡ '\\' {
            escaped = ¬escaped
            return false
        }
        if r ≡ '\" ∧ ¬escaped {
            opened = ¬opened
        }
        if r ≡ '\" ∧ ¬escaped {
            openedd = ¬openedd
        }
        escaped = false
        return false
    }
    return strings.FieldsFunc(s, ff)
}
```

25. Инициализируем *c.Stdout* предыдущим значением *stdout*. Если это не первая команда конвейера, то создаем канал для *c.Stdin*, по которому команда будет получать данные. Второй конец нового канала сохраняется в *stdout*

⟨Заполнить *c.Stdin* и *c.Stdout* и сохранить в *stdout* второй конец канала 25⟩ ≡

```
{
    c.Stdout = stdout
    if ¬first {
        if out, in, err := os.Pipe(); err ≠ nil {
            glog.Errorf("can't create pipe: %v\n", err)
            break
        } else {
            c.Stdin = out
            stdout = in
        }
    } else {
        c.Stdin = os.Stdin
    }
}
```

Используется в секциях 26 и 36.

26. Если запускается конвейер, создаем для запускаемого процесса канал для связи с процессом - источником данных. Также создаем два канала для взаимодействия процесса с GDB через GDB/MI. Так как после запуска процесса дублирующие дескрипторы каналов должны быть закрыты, добавляем каналы в *toclose*. Канал *stdout* может быть каналом в памяти для обмена данными с внутренней командой, в этом случае его не нужно закрывать

```

⟨Заполнить у c стандартные дескрипторы и два дополнительных для взаимодействия с GDB 26⟩ ≡
{
  ⟨Заполнить c.Stdin и c.Stdout и сохранить в stdout второй конец канала 25⟩
  if _, ok := c.Stdout.(*io.PipeWriter); ¬ok ∧ c.Stdout ≠ os.Stdout {
    toclose = append(toclose, c.Stdout.(io.Closer))
  }
  if c.Stdin ≠ os.Stdin {
    toclose = append(toclose, c.Stdin.(io.Closer))
  }
  c.Stderr = os.Stderr
  var err error
  var r, w *os.File
  if r, fromgdb, err = os.Pipe(); err ≠ nil {
    glog.Errorf("can't create pipe: %v\n", err)
    break
  }
  if togdb, w, err = os.Pipe(); err ≠ nil {
    glog.Errorf("can't create pipe: %v\n", err)
    break
  }
  c.ExtraFiles = append(c.ExtraFiles, r, w)
  toclose = append(toclose, r, w)
}

```

Используется в секции 27.

27. Стартуем процесс или внутреннюю команду и параллельное считывание команд для GDB

⟨Запустить *cmd* и добавить в список команд 27⟩ ≡

```
{
  switch c := cmd.(type) {
    case *exec.Cmd:
      ⟨Заполнить у c стандартные дескрипторы и два дополнительных для взаимодействия с
        GDB 26⟩
    case *internal:
      ⟨Заполняем дескрипторы внутренней команды 36⟩
  }
  if err := cmd.Start(); err != nil {
    glog.Errorf("can't start process: %s\n", err)
    break
  }
  ⟨Закрыть переданные дескрипторы 28⟩
  go func(){
    var pid int
    if v, ok := cmd.(*exec.Cmd); ok {
      pid = v.Process.Pid
    } else {
      pid = os.Getpid()
    }
    bufr := bufio.NewReader(togdb)
    for s, err := bufr.ReadString('\n'); err == nil ∨ len(s) ≠ 0; s, err = bufr.ReadString('\n') {
      glog.V(debug).Infof("%s has been received from pid %d", s, pid)
      ⟨Ожидать разрешения выполнения следующей команды 50⟩
      togbch ← request{pid: pid, out: fromgdb, cmd: s}
    }
    glog.V(debug).Infof("end of input for pid %d", pid)
    togdb.Close()
    fromgdb.Close()
  }()
  cnv = append(cnv, cmd)
}
```

Используется в секции 21.

28.

⟨Закрыть переданные дескрипторы 28⟩ ≡

```
{
  for _, p := range toclose {
    p.Close()
  }
  toclose = nil
}
```

Используется в секции 27.

29.

⟨Импортируемые пакеты 3⟩ +≡

```
"os/signal"
"syscall"
```

30.

```

< Инициализация сигнальных обработчиков 30 > ≡
{
    sigch := make(chan os.Signal, 10)
    defer signal.Stop(sigch)
    signal.Notify(sigch)
    go func(){
        for true {
            s, ok := <- sigch
            if !ok {
                fmt.Fprintf(os.Stderr, "exit_from_handler\n")
                return
            }
            switch s {
            case syscall.SIGPIPE:
                glog.V(debug).Infof("signal_SIGPIPE(%#v)", s)
                signal.Ignore(s)
            case os.Interrupt:
                glog.V(debug).Infof("signal_SIGINT(%#v)", s)
                to gdbch <- "-exec-interrupt"
            default:
                glog.V(debug).Infof("signal_%#v", s)
            }
        }
    }()
}

```

Используется в секции 2.

31. Если в качестве внутренней команды используется *args*, то читаем их *stdin* аргументы для идущей за *args* внутренней команды и запускаем внутреннюю команду с каждым считанным из *stdin* набором аргументов

32.

```

< Типы 10 > +≡
internal struct{
    cmd string
    gdbin io.ReadCloser
    gdbout io.WriteCloser
    Stdin io.ReadCloser
    Stdout io.WriteCloser
    wait chan bool
}

```


33. Метод *Start* для внутренней команды.

```
func (this * internal) Start() error {
    go func() {
        defer this.gdbin.Close()
        defer this.gdbout.Close()
        if this.Stdout != os.Stdout {
            defer this.Stdout.Close()
        }
        defer close(this.wait)
        defer func() { glog.V(debug).Infof("command_%v_has_done", this.cmd) }()
        this.cmd = strings.TrimLeftFunc(this.cmd, unicode.IsSpace)
        var c string
        if i := strings.IndexFunc(this.cmd, unicode.IsSpace); i != -1 {
            c = this.cmd[:i]
        }
        if c == "args" {
            this.cmd = this.cmd[len(c):]
            stdr := bufio.NewReader(this.Stdin)
            for s, err := stdr.ReadString('\n'); err == nil; s, err = stdr.ReadString('\n') {
                cmd := this.cmd + " " + strings.TrimSpace(s)
                < Отправить команду cmd и обработать результат 37 >
            }
        } else {
            cmd := this.cmd
            < Отправить команду cmd и обработать результат 37 >
        }
        return
    }()
    return nil
}
```

34. Метод *Wait* для внутренней команды.

```
func (this * internal) Wait() error {
    glog.V(debug).Infof("waiting_for_internal_command_%v_is_finished", this.cmd)
    ← this.wait
    glog.V(debug).Infof("internal_command_%v_has_finished", this.cmd)
    return nil
}
```

35. Заполняю поля внутренней команды.

< Создать внутреннюю команду 35 > ≡

```
var ci internal
ci.wait = make(chan bool)
ci.cmd = c
cmd = &ci
```

Используется в секции 22.

36. Для *togdb* и *fromgdb* используем каналы в памяти. *stdin* и *stdout* могут наполняться из внешних процессов, поэтому для них используются обычные каналы

⟨Заполняем дескрипторы внутренней команды 36⟩ ≡

```
{  
    c.gdbin, fromgdb = io.Pipe()  
    togdb, c.gdbout = io.Pipe()  
    ⟨Заполнить c.Stdin и c.Stdout и сохранить в stdout второй конец канала 25⟩  
}
```

Используется в секции 27.

37. Отправляем команду на выполнение в GDB, читаем вывод до появления результата с префиксом *CF* . Печатаем обработанные результаты и возможные ошибки, остальной вывод игнорируется

⟨ Отправить команду *cmd* и обработать результат 37 ⟩ ≡

```
{
  glog.V(debug).Infof("internal_command:_%#v", cmd)
  if _, err := io.WriteString(this.gdbout, cmd + "\n"); err != nil {
    fmt.Fprintf(os.Stderr, "can't_start_gdb_command_%s'_%s\n", cmd, err)
    return
  }
  gdbr := bufio.NewReader(this.gdbin)
  rp := strings.NewReplacer("\n", "\n", "\\t", "\t", "\\\" \"", "\"")
  quit := false
  for s, err := gdbr.ReadString('\n'); err == nil; s, err = gdbr.ReadString('\n') {
    glog.V(debug).Infof("sending:_%s", s)
    if len(s) == 0 {
      continue
    }
    print := true
    switch s[0] {
      case '^':
        s = s[2:len(s)-2]
        ⟨ Если есть приглашение для вводе ">", отправить в gdbin ввод с терминала 38 ⟩
      case '!':
        quit = true
        print = false
        if strings.HasPrefix(s, "^error") {
          s = s[6:]
          if len(s) == 0 ∨ s[0] != ' ' {
            break
          }
          v, _, ok := common.ParseResult(s[1:])
          if ok ∧ len(v) != 0 ∧ v[0].Name == "msg" {
            s = fmt.Sprintf("%s\n", v[0].Val.(string))
            print = true
          }
        } else if strings.HasPrefix(s, "^done") {
          s = s[5:]
          if len(s) == 0 ∨ s[0] != ' ' {
            break
          }
          if v, _, ok := common.ParseResult(s[1:]); ok {
            s = v.String() + "\n"
            print = true
          }
        }
      default:
        continue
    }
  }
  if print {
    s = rp.Replace(s)
    if n, err := io.WriteString(this.Stdout, s); err != nil ∨ n != len(s) {
      glog.V(debug).Infof("can't_write_%s'_to_stdout,_%d_bytes_has_been_written:_%s", s, n, err)
    }
  }
}
```

```

        return
    }
}
if quit {
    break
}
}
}
}

```

Используется в секции 33.

38. Команда GDB *commands* ждет ввода команд с терминала, будем читать ввод через *readline* и отправлять напрямую в *gdbin*

```

⟨ Если есть приглашение для вводе ">", отправить в gdbin ввод с терминала 38 ⟩ ≡
{
    if s1 := strings.TrimSpace(s); s1 == ">" {
        rl.SetPrompt(s)
        s, err := rl.Readline()
        if err != nil {
            continue
        }
        glog.V(debug).Infof("entered_text_inside_of_internal_command: '%s'", s)
        io.WriteString(gdbin, s + "\n")
        continue
    }
}

```

Используется в секции 37.

39.

```

⟨ Импортируемые пакеты 3 ⟩ +=
    "github.com/golang/glog"

```

40.

```

⟨ Глобальные переменные 4 ⟩ +=
    debug glog.Level = 0

```

41.

```

⟨ Подготовить трассировку 41 ⟩ ≡
{
    glog.V(debug).Infof("main")
    defer glog.V(debug).Infof("main_is_done")
    defer glog.Flush()
}

```

Используется в секции 2.

42.

```

⟨ Импортируемые пакеты 3 ⟩ +=
    "sync"

```

43.

⟨ Глобальные переменные 4 ⟩ +≡
ready = **make**(**chan bool**, 1)
once sync.Once

44.

⟨ Разрешить ввод 44 ⟩ ≡
 {
 *glog.V(debug).Info*ln("an_attempt_to_allow_of_input")
 ready ← **true**
 *glog.V(debug).Info*ln("input_is_allowed")
 }

Используется в секциях 14, 17, и 45.

45.

⟨ Однократно выполнить операции инициализации при загруженном GDB 45 ⟩ ≡
once.Do(**func**() {
 go func() { ⟨ Заполнить автозаполнение и список зарезервированных команд 57 ⟩ ⟨ Разрешить
 ввод 44 ⟩ }()
 })

Используется в секции 15.

46.

⟨ Захватить ввод 46 ⟩ ≡
 {
 *glog.V(debug).Info*ln("an_attempt_to_lock_of_input")
 ← *ready*
 *glog.V(debug).Info*ln("input_is_locked")
 }

Используется в секциях 14 и 17.

47.

⟨ Глобальные переменные 4 ⟩ +≡
next = **make**(**chan bool**, 1)

48.

⟨ Подготовить синхронизацию выполнения команд 48 ⟩ ≡
next ← **true**

Используется в секции 12.

49.

⟨ Разрешение выполнения следующей команды 49 ⟩ ≡
*glog.V(debug).Info*f("allow_an_execution_of_a_next_command")
next ← **true**

Используется в секции 14.

50.

⟨ Ожидать разрешения выполнения следующей команды 50 ⟩ ≡
 $\leftarrow next$
 $glog.V(debug).Infof("an_execution_of_a_next_command_is_allowed")$

Используется в секции 27.

51.

⟨ Дополнительные встроенные команды 51 ⟩ ≡
 $"args": ""$,

Используется в секции 18.

52.

⟨ Проверить аргументы командной строки, вывести информацию о программе, если необходимо 52 ⟩ ≡

```
{
    if len(os.Args)>1 & strings.TrimSpace(os.Args[1]) == "-h" {
        fmt.Fprintf(os.Stdout, "GDBSh 0.31, a shell for GDB\n",
            "Copyright (C) 2015, 2016 Alexander Sychev\n", "Usage: \n", "\tgdbsh <GDB options>\n",
            "GDBSh allows to use pipelines from GDB and external programs.\n",
            "A command 'args' allows to build and execute command lines with GDB commands from\n",
            "om standart input.\n",
            "Special external programs can send GDB commands to GDB and obtain results.\n",
            "Two descriptors (3,4) are dedicated for an every external program for such purposes.\n")
        return
    }
}
```

Используется в секции 2.

53.

⟨ Импортируемые пакеты 3 ⟩ +≡
 $"github.com/chzyer/readline"$

54.

⟨ Глобальные переменные 4 ⟩ +≡
 $pc []readline.PrefixCompleterInterface$
 $rl * readline.Instance$

55.

⟨ Создать экземпляр readline 55 ⟩ ≡

```
var err error
rl, err = readline.NewEx(&readline.Config{Prompt: "gdbsh$ ", AutoComplete:
readline.NewPrefixCompleter(pc...), InterruptPrompt: "interrupt", EOFPrompt: "quit"})
if err != nil {
    panic(err)
}
defer rl.Close()
```

Используется в секции 17.

56. С помощью рекурсивной функции *makePcItems* полученный список команд с подкомандами преобразуется в *readline.PrefixCompilerInterface*. Рекурсия нужна, чтобы сгруппировать подкоманды для одной команды в один *readline.PcItem*. Команда "help" игнорируется, поскольку она добавляется отдельно с возможностью автозаполнения всеми остальными командами.

```
func makePcItems(o []string, i int) (res []readline.PrefixCompleterInterface){
    loop:
    for len(o)>0 {
        if len(o[0]) ≤ i ∨ o[0][0] ≡ "help" {
            o = o[1:]
            continue
        }
        s := o[0][i]
        j := 1;
        for ; j<len(o); j++ {
            if len(o[j])>i ∧ o[j][i] ≠ s {
                res = append(res, readline.PcItem(s, makePcItems(o[0:j], i+1) ...))
                o = o[j:]
                continue loop
            }
        }
        res = append(res, readline.PcItem(s, makePcItems(o[0:j], i+1) ...))
        o = o[j:]
    }
    return res
}
```

57. Создаем список для автозаполнения и дополняем его командами "help" и "args" с автозаполнением всем перечнем команд.

```
< Заполнить автозаполнение и список зарезервированных команд 57 > ≡
{
    var o []string
    < Получить список команд 59 >
    for _, v := range o {
        cmds[v[0]] = ""
    }
    pc = makePcItems(o, 0)
    pc = append(pc, readline.PcItem("args", pc ...))
    pc = append(pc, readline.PcItem("help", pc ...))
}
```

Используется в секции 45.

58.

```
< Импортируемые пакеты 3 > +=
"sort"
```

59. Получим скисок всех команд с помощью команды "help_all". Для этого запустим команду, а в качестве канала для вывода будем использовать канал в памяти, из которого в отдельном потоке будет вычитываться весь вывод запущенной команды, фильтроваться и добавляться в массив строк. К сожалению, из-за ошибок в GDB, команды не всегда упорядочены, поэтому их приходится дополнительно отсортировать. Затем полученные команды разбиваются на подкоманды для дальнейшей обработки.

⟨Получить список команд 59⟩ ≡

```
{
  s := "help_all"
  var stdout io.WriteCloser
  var gdbin io.ReadCloser
  gdbin, stdout = io.Pipe()
  ready := make(chan bool)
  go func(){
    defer stdout.Close()
    defer gdbin.Close()
    defer close(ready)
    gdbr := bufio.NewReader(gdbin)
    var ss []string
    for{
      s, err := gdbr.ReadString('\n')
      if err != nil {
        break
      }
      if i := strings.Index(s, "\n-"); i != -1 {
        ss = append(ss, s[:i])
      }
    }
    sort.Strings(ss)
    for _, v := range ss {
      o = append(o, strings.Fields(v))
    }
  }()
  ⟨Запуск команд с выводом в stdout 19⟩
  ← ready
}
```

Используется в секции 57.

60. Здесь определяются зарезервированные команды, которые должны иметь возможность запуститься при инициализации и короткие команды, не описанные в GDB

⟨Зарезервированные команды GDB 60⟩ ≡

```
"help": "",  
"b": "",  
"c": "",  
"d": "",  
"f": "",  
"i": "",  
"l": "",  
"n": "",  
"p": "",  
"q": "",  
"r": "",  
"u": "",  
"x": "",
```

Используется в секции 18.

61. Индекс.

(gdb): [14](#).
**stopped:* [14](#).
>: [37](#), [38](#).
running: [14](#).
ackch: [11](#).
ar: [23](#).
Args: [5](#), [52](#).
args: [57](#), [5](#), [6](#), [31](#).
Atoi: [14](#).
AutoComplete: [55](#).
bufio: [9](#), [16](#), [27](#), [33](#), [37](#), [59](#).
bufr: [27](#).
ci: [35](#).
cl: [19](#).
Close: [6](#), [27](#), [28](#), [33](#), [55](#), [59](#).
Closer: [19](#), [26](#).
cmd: [4](#), [6](#), [7](#), [8](#), [10](#), [12](#), [19](#), [21](#), [23](#), [27](#), [32](#), [33](#),
[34](#), [35](#), [37](#).
Cmd: [4](#), [19](#), [20](#), [21](#), [27](#).
cmds: [18](#), [22](#), [57](#).
cnv: [19](#), [27](#).
Command: [6](#), [23](#).
commands: [38](#).
common: [13](#), [15](#), [37](#).
Config: [55](#).
debug: [12](#), [14](#), [15](#), [16](#), [17](#), [19](#), [23](#), [27](#), [30](#), [33](#), [34](#),
[37](#), [38](#), [40](#), [41](#), [44](#), [46](#), [49](#), [50](#).
devnull: [12](#), [14](#).
DevNull: [12](#).
Do: [45](#).
eadline: [53](#).
EOFPrompt: [55](#).
err: [6](#), [14](#), [16](#), [17](#), [25](#), [26](#), [27](#), [33](#), [37](#), [38](#), [55](#), [59](#).
Errorf: [6](#), [15](#), [23](#), [25](#), [26](#), [27](#).
escaped: [24](#).
exec: [3](#), [4](#), [6](#), [19](#), [20](#), [23](#), [27](#).
ExtraFiles: [26](#).
ff: [24](#).
Fields: [59](#).
FieldsFunc: [19](#), [24](#).
File: [26](#).
file: [12](#), [14](#), [15](#).
first: [19](#), [25](#).
Flush: [14](#), [41](#).
fmt: [9](#), [8](#), [12](#), [15](#), [30](#), [37](#), [52](#).
Fprint: [52](#).
Fprintf: [8](#), [30](#), [37](#).
fromgdb: [21](#), [26](#), [27](#), [36](#).
fromgdbch: [11](#), [12](#), [16](#).
gdberr: [4](#), [6](#).
gdbin: [4](#), [6](#), [12](#), [32](#), [33](#), [36](#), [37](#), [38](#), [59](#).
gdbout: [4](#), [6](#), [16](#), [32](#), [33](#), [36](#), [37](#).
gdbr: [16](#), [37](#), [59](#).
Getpid: [27](#).
glog: [6](#), [12](#), [14](#), [15](#), [16](#), [17](#), [19](#), [23](#), [25](#), [26](#), [27](#), [30](#),
[33](#), [34](#), [37](#), [38](#), [40](#), [41](#), [44](#), [46](#), [49](#), [50](#).
HasPrefix: [12](#), [14](#), [15](#), [37](#).
help: [56](#), [57](#).
help all: [59](#).
Ignore: [30](#).
in: [25](#).
Index: [59](#).
IndexFunc: [22](#), [33](#).
Infof: [12](#), [14](#), [15](#), [16](#), [17](#), [19](#), [23](#), [27](#), [30](#), [33](#),
[34](#), [37](#), [38](#), [49](#), [50](#).
Infold: [41](#), [44](#), [46](#).
Instance: [54](#).
internal: [27](#), [32](#), [33](#), [34](#), [35](#).
Interrupt: [30](#).
InterruptPrompt: [55](#).
io: [3](#), [4](#), [10](#), [12](#), [14](#), [17](#), [19](#), [21](#), [26](#), [32](#), [36](#), [37](#), [38](#), [59](#).
IsDigit: [14](#).
IsSpace: [22](#), [33](#).
Level: [40](#).
log: [39](#).
loop: [12](#), [56](#).
main: [2](#).
makePcItems: [56](#), [57](#).
Name: [15](#), [37](#).
NewEx: [55](#).
NewPrefixCompleter: [55](#).
NewReader: [16](#), [27](#), [33](#), [37](#), [59](#).
NewReplacer: [12](#), [37](#).
next: [47](#), [48](#), [49](#), [50](#).
Notify: [30](#).
ok: [12](#), [15](#), [19](#), [22](#), [26](#), [27](#), [30](#), [37](#).
Once: [43](#).
once: [43](#), [45](#).
Open: [12](#).
openedd: [24](#).
openeds: [24](#).
os: [3](#), [5](#), [8](#), [12](#), [14](#), [15](#), [17](#), [25](#), [26](#), [27](#), [30](#), [33](#), [37](#), [52](#).
out: [10](#), [12](#), [25](#), [27](#).
ParseResult: [15](#), [37](#).
Path: [8](#), [19](#).
PATH: [22](#).
pc: [54](#), [55](#), [57](#).
PcItem: [56](#), [57](#).
pid: [10](#), [12](#), [27](#).
Pid: [19](#), [27](#).
Pipe: [25](#), [26](#), [36](#), [59](#).
PipeWriter: [26](#).

PrefixCompilerInterface: [56](#).
PrefixCompleterInterface: [54](#), [56](#).
prev: [17](#).
Process: [19](#), [27](#).
ProcessState: [8](#).
Prompt: [55](#).
quit: [37](#).
ReadCloser: [4](#), [21](#), [32](#), [59](#).
Readline: [17](#), [38](#).
readline: [14](#), [17](#), [38](#), [54](#), [55](#), [56](#), [57](#).
ReadString: [16](#), [27](#), [33](#), [37](#), [59](#).
ready: [43](#), [44](#), [46](#), [59](#).
Replace: [15](#), [37](#).
request: [10](#), [12](#), [27](#).
res: [56](#).
rl: [17](#), [38](#), [54](#), [55](#).
rp: [12](#), [15](#), [37](#).
SetPrompt: [17](#), [38](#).
sigch: [30](#).
Signal: [30](#).
signal: [29](#), [30](#).
SIGPIPE: [30](#).
sort: [58](#), [59](#).
Sprintf: [12](#), [15](#), [37](#).
ss: [59](#).
Start: [6](#), [20](#), [27](#), [33](#).
Stderr: [8](#), [26](#), [30](#), [37](#).
StderrPipe: [6](#).
Stdin: [25](#), [26](#), [32](#), [33](#).
stdin: [31](#), [36](#).
StdinPipe: [6](#).
Stdout: [12](#), [14](#), [15](#), [17](#), [25](#), [26](#), [32](#), [33](#), [37](#), [52](#).
stdout: [14](#), [17](#), [25](#), [26](#), [36](#), [59](#).
StdoutPipe: [6](#).
stdr: [33](#).
Stop: [30](#).
strconv: [13](#), [14](#).
String: [37](#).
strings: [9](#), [12](#), [14](#), [15](#), [19](#), [22](#), [24](#), [33](#), [37](#), [38](#), [52](#), [59](#).
Strings: [59](#).
Success: [8](#).
sync: [42](#), [43](#).
syscall: [29](#), [30](#).
s1: [38](#).
this: [33](#), [34](#), [37](#).
toclose: [19](#), [26](#), [28](#).
togdb: [21](#), [26](#), [27](#), [36](#).
togdbch: [11](#), [12](#), [17](#), [27](#), [30](#).
TrimLeftFunc: [33](#).
TrimSpace: [12](#), [19](#), [22](#), [33](#), [38](#), [52](#).
unicode: [13](#), [14](#), [22](#), [33](#).
Val: [15](#), [37](#).

Wait: [7](#), [20](#), [34](#).
wait: [32](#), [33](#), [34](#), [35](#).
WriteCloser: [4](#), [10](#), [12](#), [17](#), [21](#), [32](#), [59](#).
WriteString: [12](#), [14](#), [37](#), [38](#).

- ⟨ Глобальные переменные [4](#), [11](#), [18](#), [40](#), [43](#), [47](#), [54](#) ⟩ Используется в секции [2](#).
- ⟨ Дополнительные встроенные команды [51](#) ⟩ Используется в секции [18](#).
- ⟨ Если есть приглашение для вводе ">", отправить в *gdbin* ввод с терминала [38](#) ⟩ Используется в секции [37](#).
- ⟨ Ждать завершения процесса [7](#) ⟩ Используется в секциях [2](#) и [19](#).
- ⟨ Закрывать переданные дескрипторы [28](#) ⟩ Используется в секции [27](#).
- ⟨ Заполнить *c.Stdin* и *c.Stdout* и сохранить в *stdout* второй конец канала [25](#) ⟩ Используется в секциях [26](#) и [36](#).
- ⟨ Заполнить автозаполнение и список зарезервированных команд [57](#) ⟩ Используется в секции [45](#).
- ⟨ Заполнить у *c* стандартные дескрипторы и два дополнительных для взаимодействия с GDB [26](#) ⟩
Используется в секции [27](#).
- ⟨ Заполняем дескрипторы внутренней команды [36](#) ⟩ Используется в секции [27](#).
- ⟨ Запуск команд с выводом в *stdout* [19](#) ⟩ Используется в секциях [17](#) и [59](#).
- ⟨ Запустить GDB [6](#) ⟩ Используется в секции [2](#).
- ⟨ Запустить *cmd* и добавить в список команд [27](#) ⟩ Используется в секции [21](#).
- ⟨ Запустить команду *c* на выполнение и поместить ее в *cnv* [21](#) ⟩ Используется в секции [19](#).
- ⟨ Запустить параллельную обработку ввода из *stdin* [17](#) ⟩ Используется в секции [12](#).
- ⟨ Запустить параллельную обработку вывода из GDB [16](#) ⟩ Используется в секции [12](#).
- ⟨ Резервированные команды GDB [60](#) ⟩ Используется в секции [18](#).
- ⟨ Захватить ввод [46](#) ⟩ Используется в секциях [14](#) и [17](#).
- ⟨ Импортируемые пакеты [3](#), [9](#), [13](#), [29](#), [39](#), [42](#), [53](#), [58](#) ⟩ Используется в секции [2](#).
- ⟨ Инициализация сигнальных обработчиков [30](#) ⟩ Используется в секции [2](#).
- ⟨ Обработать строки для вывода в *os.Stdout* [15](#) ⟩ Используется в секции [14](#).
- ⟨ Обработка и отправка *s* в *file* [14](#) ⟩ Используется в секции [12](#).
- ⟨ Однократно выполнить операции инициализации при загруженном GDB [45](#) ⟩ Используется в секции [15](#).
- ⟨ Ожидать разрешения выполнения следующей команды [50](#) ⟩ Используется в секции [27](#).
- ⟨ Определить запускаемую команду и создать *cmd* [22](#) ⟩ Используется в секции [21](#).
- ⟨ Отправить команду *cmd* и обработать результат [37](#) ⟩ Используется в секции [33](#).
- ⟨ Подготовить аргументы командной строки для запуска GDB [5](#) ⟩ Используется в секции [6](#).
- ⟨ Подготовить синхронизацию выполнения команд [48](#) ⟩ Используется в секции [12](#).
- ⟨ Подготовить трассировку [41](#) ⟩ Используется в секции [2](#).
- ⟨ Получить список команд [59](#) ⟩ Используется в секции [57](#).
- ⟨ Проверить аргументы командной строки, вывести информацию о программе, если необходимо [52](#) ⟩
Используется в секции [2](#).
- ⟨ Проверить возвращаемый результат [8](#) ⟩ Используется в секции [2](#).
- ⟨ Разрешение выполнения следующей команды [49](#) ⟩ Используется в секции [14](#).
- ⟨ Разрешить ввод [44](#) ⟩ Используется в секциях [14](#), [17](#), и [45](#).
- ⟨ Создать внешний процесс [23](#) ⟩ Используется в секции [22](#).
- ⟨ Создать внутреннюю команду [35](#) ⟩ Используется в секции [22](#).
- ⟨ Создать экземпляр *readline* [55](#) ⟩ Используется в секции [17](#).
- ⟨ Типы [10](#), [20](#), [32](#) ⟩ Используется в секции [2](#).
- ⟨ Читать команды из *stdin*, посылать их в GDB, обрабатывать результаты [12](#) ⟩ Используется в секции [2](#).

GDBSh - командная оболочка для GDB

(version 0.32)

	Section	Page
Введение	1	2
Реализация	2	3
Индекс	61	26

Copyright © 2015, 2016, 2018 Alexander Sychev. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- The name of author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.