GDBSh - командная оболочка для GDB

(version 0.3.5)

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, 2020, 2023 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
  for i, v := range os.Args {
    if i ≡ 0 {
      continue
    }
    fmt.Fprintf(os.Stderr, "%s\n", v)
    if strings.Contains(v, "␣") {
      v = "\"" + v + "\""
    }
    args = append(args, v);
  }
  args = append(args, "--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.WriteCloser
    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:␣'%s'", 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:␣'%s'", 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␣\"%s\"\n", r.pid, c)
              }
              io.WriteString(gdbin, c)
            case string:
              glog.V(debug).Infof("to␣gdb:␣'%s'", r)
              io.WriteString(gdbin, r + "\n")
          }
      }
    }
  }
```

Используется в секции 2.

13.

```
⟨ Импортируемые пакеты 3 ⟩ +≡
  "unicode"
  "strconv"
  "github.com/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 отвечае 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(){
    gdbr := bufio.NewReader(gdbout)
    for s, err := gdbr.ReadString('\n'); err ≡ nil; s, err = gdbr.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 ожидаться окончания выполнения команд оинаковым 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 togdb 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{
  openeds := false
  openedd := false
  escaped := false
  ff := func(r rune) bool{
    if ¬openeds ∧ ¬openedd ∧ ¬escaped ∧ f(r) {
      return true
    }
    if r ≡ '\\' {
      escaped = ¬escaped
      return false
    }
    if r ≡ '\'' ∧ ¬escaped {
      openeds = ¬openeds
    }
    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 recived from pid %d", s, pid)
        ⟨ Ожидать разрешения выполнения следующей команды 50 ⟩
        togdbch ← 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)
            togdbch ← "-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. Заполням 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"
  "flag"
```

40.

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

41.

```
⟨ Подготовить трассировку 41 ⟩ ≡
  {
    flag.CommandLine = flag.NewFlagSet(os.Args[0], flag.ContinueOnError)
    flag.Parse()
    glog.V(debug).Infoln("main")
    defer glog.V(debug).Infoln("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).Infoln("an␣attempt␣to␣allow␣of␣input")
  ready ← true
  glog.V(debug).Infoln("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).Infoln("an␣attempt␣to␣lock␣of␣input")
   ← ready
  glog.V(debug).Infoln("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).Infof("allow␣an␣execution␣of␣a␣next␣command")
next ← true
```
Используется в секции 14.

50.

⟨ Ожидать разрешения выполнения следующей команды 50 ⟩ ≡
```
← 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.Fprint(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␣fr\
        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 запущеной 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, "␣--"); i ≠ -1 {
          for _, v := range strings.Split(s[:i], ",") {
            ss = append(ss, strings.Trim(v, "␣"))
          }
        }
      }
      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.

⟨ 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.
⟨ Закрыть 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.
⟨ Зарезервированные 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 0.3.5)