Введение в Golang. Лекция 4

Слушаем сетевое соединение

Слушаем TCP-сокет с исполььзованием пакета net

В этой лекции мы начнем рассматривать возможности Go для построения уже каких-то реальных систем. Начнем мы с работы с сетью и будем слушать tcp socket и обрабатывать входящие команды, которые туда кто-то будет слать. Итак, с чего мы начинаем?

```
package main

import (
        "bufio"
        "fmt"
        "net"
)


func main() {
        listner, err := net.Listen("tcp", ":8080")
        if err != nil {
                panic(err)
        }
        for {
                conn, err := listner.Accept()
                if err != nil {
                        panic(err)
                }
                go handleConnection(conn)
        }
}
```

Для работы с сокетами надо подключить пакет net. Далее мы создаем объект, который будет слушать tcp socket на порту 8080. И дальше в цикле начинаем обрабатывать входящие соединения. Вызывается функция accept, и, как только кто-то подключается, мы в отдельной горутине начинаем обрабатывать это соединение. Обратите внимание, что обработка соединения в отдельной горутине нужна для того, чтобы к серверу могло подключаться сразу много пользователей. Если убрать команду go, программа не перейдет к следующему подключению, пока не отвалится текущее.

Таким образом вся логика обработки соединения описана в функциии-обработчике handleConnection, запускаемой в отдельной горутине. Так она устроена в нашем примере.

```
func handleConnection(conn net.Conn) {
        name := conn.RemoteAddr().String()
```

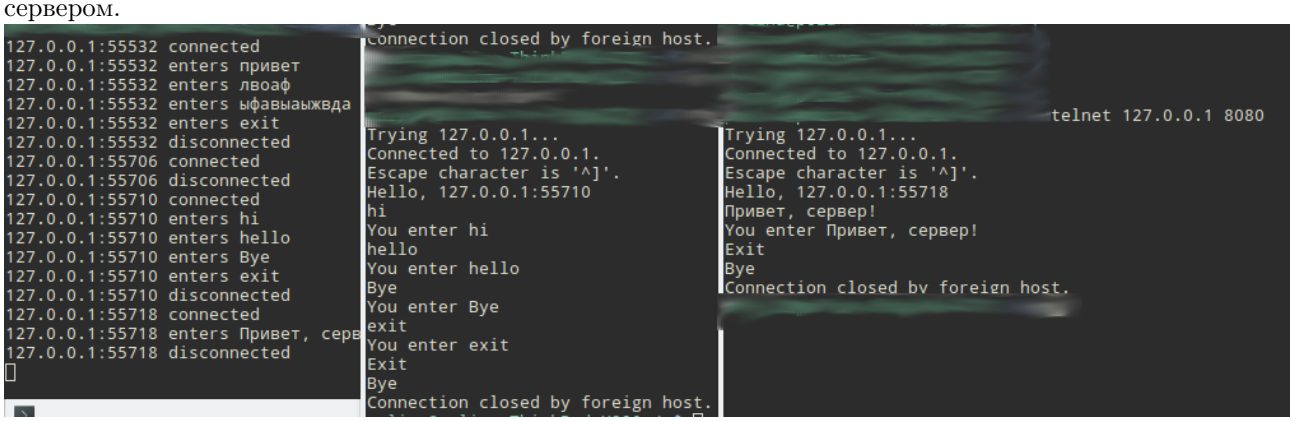
```
                fmt.Printf("%+v connected\n", name)
                conn.Write([]byte("Hello, " + name + "\n\r"))

                defer conn.Close()

                scanner := bufio.NewScanner(conn)
                for scanner.Scan() {
                        text := scanner.Text()
                        if text == "Exit" {
                                conn.Write([]byte("Bye\n\r"))
                                fmt.Println(name, "disconnected")
                                break
                        } else if text != "" {
                                fmt.Println(name, "enters", text)
                                conn.Write([]byte("You enter " + text + "\n\r"))
                        }
                }
        }
```

Для начала мы пытаемся получить имя удаленного соединения, просто его адрес для удобства. Печатаем в консоль программы, что кто-то соединился. Отправляем приветствие тому, кто соединился. Структура net.Conn, возвращаемая функцией Accept и передаваемая в обработчик соединения, реализует в частности интерфейс io.Writer, поэтому мы можем туда писать какие-то данные, используя slice byte. Дальше в defer мы указываем, что соединение нужно закрыть по завершении функции, чтобы не было никаких утечек. Дальше мы создаем scanner, который будет ожидать ввода. И пока к нам что-то приходит, мы будем смотреть, что приходит, и обрабатывать. Если нам ввели exit, то мы попрощаемся с соединением и прекратим его обработку. То есть сработает defer, соединение закроется. Если придет какой-то непустой text, мы его выведем на экран.

Теперь, чтобы проверить работу надо, во-первых, запустить программу. Во-вторых, открыть консоль и подключиться по telnet к нашему сервису.

```
telnet 127.0.0.1 8080
```

Можно подключиться к этому же серверу еще несколько раз. Примерно так может выглядеть работа с сервером.



Сейчас мы буквально десятью строчками создали tcp-сервер, с которым уже можно работать. Такого рода механизм лежит в основе практически любого tcp-сервера, и, собственно, это само сердце любого сетевого сервиса. tcp-сервер — это очень важно, это первое, с чего начинается сервер. Но обычно нас интересует более прикладной характер и интересует http-сервер. В go есть встроенный http-сервер, который мы рассмотрим в отдельной главе.

Обработка HTTP-запросов

Обслуживание HTTP-запросов

В этой главе мы познакомимся с обработкой http-запросов средствами, стандартной библиотеки Go. Начнем мы с простого веб-сервера, который будет выводить нам хотя бы что-то на экран.

```
package main

import (
        "fmt"
        "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Привет, мир!")
        w.Write([]byte("!!!"))
}

func main() {
        http.HandleFunc("/", handler)

        fmt.Println("starting server at :8080")
        http.ListenAndServe(":8080", nil)
}
```

Все необходимое для работы с http в Go лежит в пакете "net/http". Мы начнем знакомство с того, что научимся определять обработчик запросов. В самом простом варианте это делается через передачу в HandleFunc адреса и функции-обработчика, которая будет обрабатывать запрос на этом URL'е. Функция-обработчик имеет строго фиксированный тип и должна первым аргументом получать ResponseWriter, то есть то, куда мы будем записывать результат обработки, вторым аргументом — http.Request, то есть сам запрос, который мы будем обрабатывать, и все параметры этого запроса.

В w — это ResponseWriter — мы можем писать либо через функцию Fprintln, которая сама отформатирует в нужный вид, либо напрямую записывать набор байт, используя [ ]byte. В примере используем оба варианта.

То есть в самом базовом варианте http-сервер можно закодить в Go буквально на двадцати строках. А теперь давайте рассмотрим, каким образом мы можем обрабатывать несколько разных URL'ов при помощи стандартных функций. Итак, в базовом варианте мы определили просто HandleFunc как обработку просто слэша, то есть самого корневого элемента. Теперь мы объявим еще два обработчика.

```
func handler(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Main page")
}

func main() {
        http.HandleFunc("/page",
                func(w http.ResponseWriter, r *http.Request) {
                        fmt.Fprintln(w, "Single page:", r.URL.String())
                })

        http.HandleFunc("/pages/",
                func(w http.ResponseWriter, r *http.Request) {
                        fmt.Fprintln(w, "Multiple pages:", r.URL.String())
                })

        http.HandleFunc("/", handler)

        fmt.Println("starting server at :8080")
        http.ListenAndServe(":8080", nil)
}
```

Теперь добавляется обработчик для аресов "/page"и "/pages/". Обратите внимание, что page без слеша в конце, а pages со слэшом в конце, то есть это какой-то префикс. Обработчкик для "/page"будет обрабатывать только "/page"и больше ничего, обработчик "/pages/"будет обрабатывать как сам URL "/pages/ так и все адреса с таким префиксом. Корневой обработчик будет обрабатывать вообще все, что не попало в другие URL'ы.

Как можно видеть, обработчик "/page"говорит, что мы запросили одиночную страницу, и выводит её URL. URL мы достаем из объекта Request. Обработчик "/pages/"делает то же самое.

Рассмотренная возможность обрабатывать URL'ы разными функциями очень хороша, но в таком случае мы не можем прокидывать в функцию никаких других параметров, никаких общих данных, то есть никакого Dependency Injection, не можем использовать глобальные переменные. Иногда это бывает не очень удобно. Поэтому, мы можем указывать не только обработчик функции, но и вешать обработчик на структуру.

```
type Handler struct {
        Name string
}

func (h *Handler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Name:", h.Name, "URL:", r.URL.String())
}

func main() {
        testHandler := &Handler{Name: "test"}
        http.Handle("/test/", testHandler)

        rootHandler := &Handler{Name: "root"}
        http.Handle("/", rootHandler)

        fmt.Println("starting server at :8080")
        http.ListenAndServe(":8080", nil)
}
```

Например, у нас есть структура Handler, у нее есть поле Name, и есть функция ServeHTTP, которая принимает в себя ResponseWriter и Request. Такую структуру мы можем передаь в http.Handle — она подходит под нужный интерфейс, сервером будет вызываться метод ServeHTTP, который будет обрабатывать наш результат.

В примере создается два обработчика. Первый обработчик — с именем "test он обрабатывает все URL'ы, которые находятся по префиксу "test второй обработчик — с именем "root обрабатывает все остальное. В каждом обработчике мы выводим имя обработчика и URL.

Такой подход бывает удобен, когда у вас много URL'ов, которые находятся по одному префиксу, тогда внутри уже вы можете делать, например, switch.

В предыдущих примерах мы использовали HandleFunc и Handler, и ListenAndServe http прямо из пакета http. Эта функция Беркли, которая предоставляет нам очень быстрый интерфейс до внутренних объектов. Но иногда бывает необходимо, например, поднять несколько разных URL'ов и несколько разных серверов, и мы можем это сделать. Например, то, что мы раньше использовали как HandleFunc, на самом деле вызывает в себя базовый глобальный мультиплексор, который обрабатывает запросы. Мы можем создать отдельный объект с таким мультиплексором, например вот так:

```
func handler(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "URL:", r.URL.String())
}

func main() {
        mux := http.NewServeMux()
        mux.HandleFunc("/", handler)

        server := http.Server{
                Addr:         ":8080",
                Handler:      mux,
                ReadTimeout:  10 * time.Second,
                WriteTimeout: 10 * time.Second,
        }
```

```
        fmt.Println("starting server at :8080")
        server.ListenAndServe()
}
```

и зарегистрировать у него нужные обработчики. Таким образом, можно реализовать несколько мультиплексоров, которые будут обрабатывать одни и те же URL'ы, но по-разному. Затем создать отдельный сервер, указав у него разные параметры. В данном случае мы создаем http-сервер, указываем порт, мультиплексер-обработчик запросов, и Read и WriteTimeout'ы, чтобы очень медленные коннекты не висели долго.

Фактически, это тот же самый код, который мы могли бы получить shortcut'ами, но он предоставляет нам гораздо больше возможностей для настройки. После того, как параметры заданы, можно начинать слушать сервер. Мы можем также запустить любое количество http-серверов в разных гарутинах.

```
func runServer(addr string) {
        mux := http.NewServeMux()
        mux.HandleFunc("/",
                func(w http.ResponseWriter, r *http.Request) {
                        fmt.Fprintln(w, "Addr:", addr, "URL:", r.URL.String())
                })

        server := http.Server{
                Addr:    addr,
                Handler: mux,
        }

        fmt.Println("starting server at", addr)
        server.ListenAndServe()
}

func main() {
        go runServer(":8081")
        runServer(":8080")
}
```

Это бывает довольно удобно, потому что мы можем слушать разными серверами разные порты и поднять не только, например, http, но и любые другие серверы, например, для мониторинга.

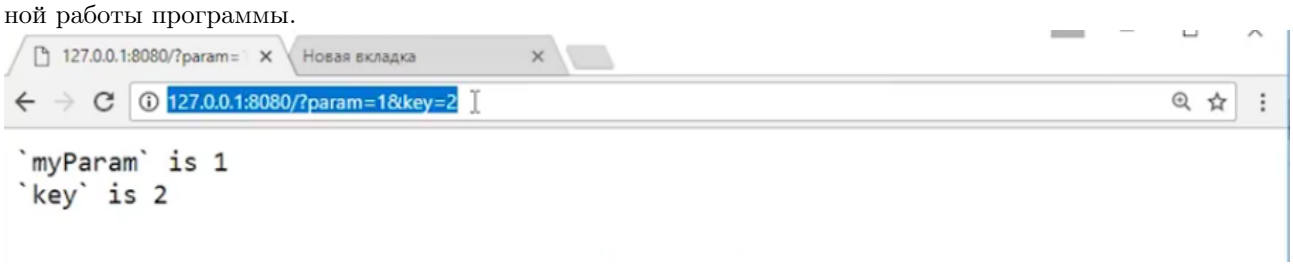
Работа с параметрами запросов

Теперь рассмотрим работу с входящими параметрами запроса, то есть с теми параметрами, которые вы можете получить от пользователя тем или иным образом. Никакая работа с динамическими данными невозможна без обработки этих параметров. Давайте рассмотрим следующую функцию.

```
func handler(w http.ResponseWriter, r *http.Request) {
        myParam := r.URL.Query().Get("param")
        if myParam != "" {
                fmt.Fprintln(w, "`myParam` is", myParam)
        }

        key := r.FormValue("key")
        if key != "" {
                fmt.Fprintln(w, "`key` is", key)
        }
}

func main() {
        http.HandleFunc("/", handler)

        fmt.Println("starting server at :8080")
        http.ListenAndServe(":8080", nil)
}
```

В ней мы просто получаем get-параметры. Получить параметры можно несколькими образами. Первым является непосредственно получение из url get-параметров, которые приходят к нам в запросе. В данном случае мы из url получаем объект query, а у него уже получаем параметр с незамысловатым именем param. Если такого параметра нет, этот метод возвращает пустоту.

Вторым способом является использование функции FormValue из объекта request и через неё мы получаем параметр key. При помощи этой функции можно также получать параметры из post-запросов. В примере мы используем оба варианта получения параметров и выводим их на экран. Вот пример типичной работы программы.



Мы рассмотрели get-параметры. А как насчёт post-параметров?

У нас есть хэндлер mainPage, который проверяет, что, если метод, с которым мы пришли, не post, то мы рисуем форму логина. Форму логина пока записываем руками, просто как html. А если мы пришли через post, то мы можем должны доступ к параметрам. Мы можем руками распарсить форму, когда нам надо, и обратиться к ней за нашими данными. Этот вариант закоментирован. А можем использовать короткое обращение, используя функцию FormValue, которая всю эту работу сделает за нас и вернет первый нужный параметр. Причем функция FormValue будет возвращать параметры вообще всего запроса, не только из get, но и из post. При этом, конечно же, параметры из post будутт иметь приоритет.

Еще одним входящим параметром являются куки. Куки — это значение, которое постоянно в рамках HTTP сессии пользователя для того, чтобы, например, запоминать уже авторизационные данные на сервер. Итак, давайте посмотрим, как же нам работать с куками.

```
func mainPage(w http.ResponseWriter, r *http.Request) {
        session, err := r.Cookie("session_id")
        loggedIn := (err != http.ErrNoCookie)

        if loggedIn {
                fmt.Fprintln(w, `<a href="/logout">logout</a>`)
                fmt.Fprintln(w, "Welcome, "+session.Value)
        } else {
                fmt.Fprintln(w, `<a href="/login">login</a>`)
                fmt.Fprintln(w, "You need to login")
        }
}

func loginPage(w http.ResponseWriter, r *http.Request) {
        expiration := time.Now().Add(10 * time.Hour)
        cookie := http.Cookie{
                Name:    "session_id",
                Value:   "rvasily",
                Expires: expiration,
        }
        http.SetCookie(w, &cookie)
        http.Redirect(w, r, "/", http.StatusFound)
}

func logoutPage(w http.ResponseWriter, r *http.Request) {
        session, err := r.Cookie("session_id")
        if err == http.ErrNoCookie {
                http.Redirect(w, r, "/", http.StatusFound)
                return
```

```
        }

        session.Expires = time.Now().AddDate(0, 0, -1)
        http.SetCookie(w, session)

        http.Redirect(w, r, "/", http.StatusFound)
}

func main() {
        http.HandleFunc("/login", loginPage)
        http.HandleFunc("/logout", logoutPage)
        http.HandleFunc("/", mainPage)

        fmt.Println("starting server at :8080")
        http.ListenAndServe(":8080", nil)
}
```

Зарегистрируем три функции: loginPage, logoutPage и mainPage. Главная страница получает куку, используя метод cookie из объекта request, и дальше проверяет, если такая кука есть, то считает, что пользователь залогинен и дает ему ссылку на logout, пишет значение куки session.Value. Session — это как раз таки наша кука. Если куки нет, то пользователь не залогинен, и ему предлагается залогиниться.

Функция login просто выставляет куку с заранее заданными значениями, в примере у нас не многопользовательский сервер. Кука представляет собой структуру в пакете http. У неё есть имя, value, дата протухания, и для того, чтобы поставить куку, в пакете http есть специальная функция SetCookie, в которую надо передавать response writer и куку, которую нужно поставить. После проставления куки функция ligin делает redirect на главную страницу.

Функция logout просто удаляет куку. Если ее вообще нет, то сразу происходит redirect на главную, если она есть, то выставляется дата в прошлое, таким образом, серверу сообщается, что кука больше не валидная, ее нужно удалить. Затем тоже происходит redirect.

Откроем консоль, чтобы смотреть, что нам выставляется сервером. Итак, главная страница, которая проверила, что куки нет, и нам нужно ее получить. Жмем на логин. Итак, вот наш логин. Вот было выставление куки имя cookies *session_id*, значение rvasily, и дата протухания. Теперь мы с редиректа на главную. Если мы попробуем выйти, нажмем на кнопку logout, то нам тоже выставилась кука, но с датой уже в прошлое.

Куки в основном используются для запоминания каких-то значений, которые не очень секьюрные, потому что браузерный JavaScript может получать к ним доступ. Очень часто куки используются для того, чтобы запомнить идентификатор сессии пользователя, чтобы секретные значения уже определять на сервере.

Ещё очень важная информация, которая находится в http request, это заголовки header. Header могут быть как выставлены, так и получены из запроса. В этом примере показывается, как с этим работать.

```
func handler(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("RequestID", "d41d8cd98f00b204")

        fmt.Fprintln(w, "You browser is", r.UserAgent())
        fmt.Fprintln(w, "You accept", r.Header.Get("Accept"))
}

func main() {
        http.HandleFunc("/", handler)

        fmt.Println("starting server at :8080")
        http.ListenAndServe(":8080", nil)
}
```

Для того, чтобы выставить header, мы должны получить объект header из response writer и указать в нем функцию set, в которую мы передадим само название header и значение. В данном указываем какое-то случайное значение-идентификатор запроса. Получить значения можно двумя способами. В http.Request

есть сокращение, чтобы получить сразу UserAgent. Для того чтобы получить какие-то другие header, нужно обратиться к структуре header и вызвать у нее метод gets нужным header.

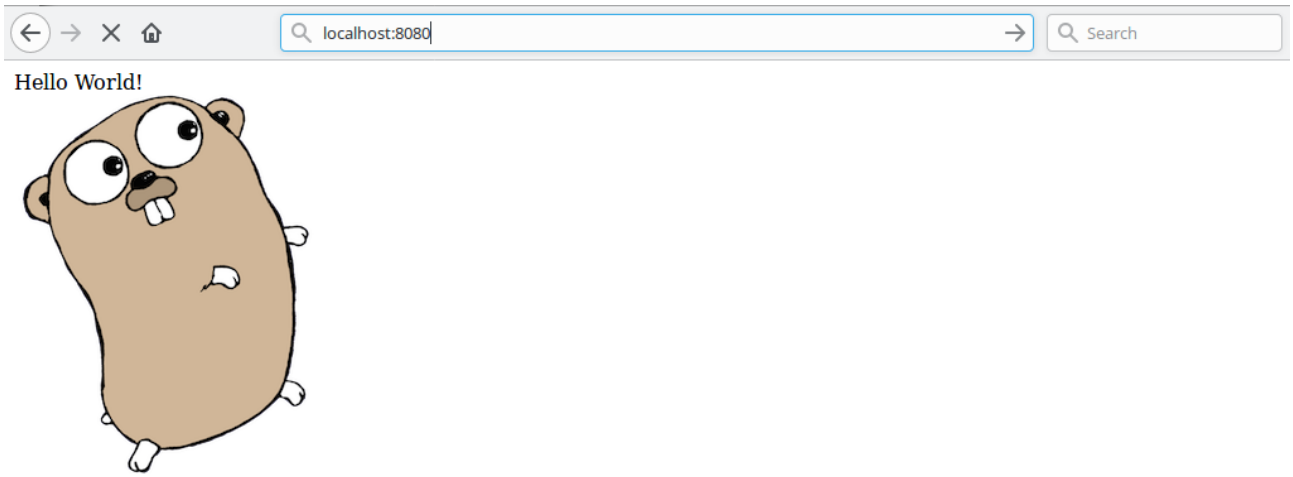
Обслуживание статичных данных

В этом разделе рассмотрим обслуживание статических файлов на Go, например, картинок либо CSS-файлов, либо JS-файлов. Зачастую ваш веб-сервер на Go будет находиться за nginx, и обслуживанием статики будет заниматься либо вообще nginx, либо она будет подгружаться откуда-то на CDN. Но иногда возникают случаи, когда вы хотите что-то поразрабатывать локально, и очень не хочется поднимать еще какой-то лишний сервер для обслуживания этих данных. И это можно сделать при помощи Go!

Итак, у нас есть картинка Gopher — это зверек языка Go, и пусть мы хотим вывести его на экран, написать Hello World! и дать на него ссылку. Это можно сделать с помощью следующей программы.

```
func handler(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Content-Type", "text/html")
        w.Write([]byte(‘
                Hello World! <br />
                <img src="/data/img/gopher.png" />
        ‘))
}

func main() {
        http.HandleFunc("/", handler)

        staticHandler := http.StripPrefix(
                "/data/",
                http.FileServer(http.Dir("./static")),
        )
        http.Handle("/data/", staticHandler)

        fmt.Println("starting server at :8080")
        http.ListenAndServe(":8080", nil)
}
```

Мы задаем handler, который будет отрбатывать почти всегда, и обработчик статических файлов. Данные будут лежать в папке static, которая находится рядом с программой. Мы создаем FileServer и обработчик для него staticHandler, который будет запускаться по префиксу data.
Теперь нужно запустить: go run static.go. И действительно появляется картинка,которая отдана гошным веб-сервером, и никакой другой софт для этого не используется. Go полностью самодостаточен для локальной разработки.

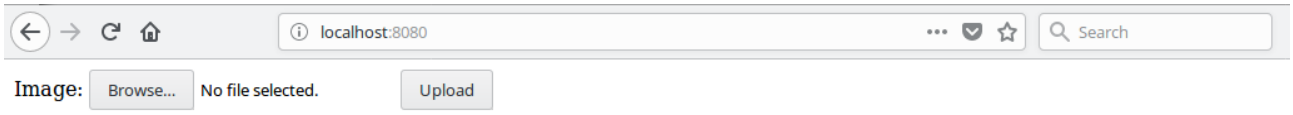


Загрузка файлов формы

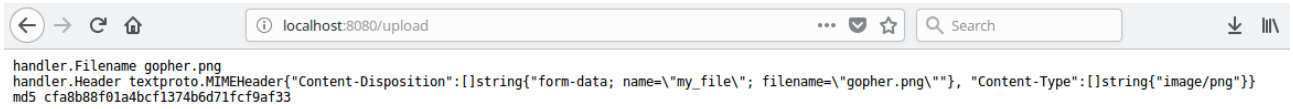
В этом разделе рассмотрим загрузку файлов из формы и то, каким образом работать с ними, используя стандартную библиотеку в go. Рассмотрим следующий пример, в котором будет очень простая форма, вывод этой формы на экран, обработчик загруженного через форму файла и обработчик переданного через post-заприс json. Вот так выглядит код формы, обработчик главной страницы, отображающий форму и функция main. Остальные два обработчика рассмотрим чуть позднее.

```
var uploadFormTmpl = []byte(‘
<html>
        <body>
        <form action="/upload" method="post" enctype="multipart/form-data">
                Image: <input type="file" name="my_file">
                <input type="submit" value="Upload">
        </form>
        </body>
</html>
‘)

func mainPage(w http.ResponseWriter, r *http.Request) {
        w.Write(uploadFormTmpl)
}

func main() {
        http.HandleFunc("/", mainPage)
        http.HandleFunc("/upload", uploadPage)
        http.HandleFunc("/raw_body", uploadRawBody)

        fmt.Println("starting server at :8080")
        http.ListenAndServe(":8080", nil)
}
```

Вот так выглядит форма.



После загрузки файла мы видим какую-то информацию.



Теперь рассмотрим код, который эту информацию выводит, то есть функцию-обработчик загруженного файла.

```
func uploadPage(w http.ResponseWriter, r *http.Request) {
        r.ParseMultipartForm(5 * 1024 * 1025)
        file, handler, err := r.FormFile("my_file")
        if err != nil {
                fmt.Println(err)
                return
        }
        defer file.Close()

        fmt.Fprintf(w, "handler.Filename %v\n", handler.Filename)
        fmt.Fprintf(w, "handler.Header %#v\n", handler.Header)

        hasher := md5.New()
        io.Copy(hasher, file)
```

```
        fmt.Fprintf(w, "md5 %x\n", hasher.Sum(nil))
}
```

В go весь запрос не вычитывается, не парсится по умолчанию. Вам нужно либо явно сказать, что нужно загрузить и обработать, либо он будет загружен по умолчанию при первом обращении к каким-то определенным фолям и функциям реквеста. Поэтому в начале мы сначала парсим форму, точнее только пять мегабайт, и если что-то еще остается, оно будет положено во временные файлы. После парсинга выполняем функция FormFile, которая обращается к файлу и мы через неё получаем сами данные. После этого, конечно же, файл сразу закрываем через defer, чтобы не утекли ресурсы. И выводим нужные данные на экран. В данном случае это имя файла, MIME заголовок, который шел в форме и посчитанное MD5 от этого файла.

Когда у вас файлов много, и вы хотите их распарсить и сразу иметь все данные, можно поступать так, как мы только что рассмотрели. Еще одна достаточно частая операция, когда у вас в POST-запросе приходят данные, которые вы не хотите парсить через MIME, то есть это не multipart/form-data, а, например, application/json. В этом случае тоже можно просто считать весь реквест в свой код. А потом что-то с ним сделать. Рассмотрим код второго обработчика.

```
func uploadRawBody(w http.ResponseWriter, r *http.Request) {

        body, err := ioutil.ReadAll(r.Body)
        defer r.Body.Close()

        p := &Params{}
        err = json.Unmarshal(body, p)
        if err != nil {
                http.Error(w, err.Error(), 500)
                return
        }

        fmt.Fprintf(w, "content-type %#v\n",
                r.Header.Get("Content-Type"))
        fmt.Fprintf(w, "params %#v\n", p)
}
```

Здесь мы парсим реквест, в котором передается json, напрямую, не пользуюясь никакими вспомогательными функциями, читая все, что есть в body. Закрываем body сразу же в defer. Дальше создаем структуру и парсим json в эту структуру. Чтобы посмотреть на это в работе, выполните, например такой запрос

```
curl -v -X POST -H "Content-Type: application/json" -d  '{"id": 2, "user": "rvasily"}'
http://localhost:8080/raw_body
```

Итак, мы рассмотрели, как можно парсить формы и получать файлы, которые вы загружаете, используя стандартную библиотеку в go.

HTTP-запросы во внешние сервисы

В предыдущих главах мы рассматривали, каким образом отвечать на http-запросы и обработать входящие в него параметры. Однако часто бывает нужно отправить http-запрос на какой-то внешний сервис — данный раздел посвящен как раз тому, как это делать.

Поднимем свой небольшой сервер, к которому будем обращаться с запросами.

```
func startServer() {
        http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
                fmt.Fprintf(w, "getHandler: incoming request %#v\n", r)
                fmt.Fprintf(w, "getHandler: r.Url %#v\n", r.URL)
        })

        http.HandleFunc("/raw_body", func(w http.ResponseWriter, r *http.Request) {
                body, err := ioutil.ReadAll(r.Body)
                defer r.Body.Close() // важный пункт!
                if err != nil {
```

```
                        http.Error(w, err.Error(), 500)
                        return
                }
                fmt.Fprintf(w, "postHandler: raw body %s\n", string(body))
        })

        fmt.Println("starting server at :8080")
        http.ListenAndServe(":8080", nil)
}
```

Теперь рассмотрим, как поступать в ситуации, когда есть какой-то url и и вы просто хотите дернуть Get-запрос. В Go для этого в пакете net.http есть функция Get, куда вы можете передать url и вам вернется респонс.

```
func runGet() {
        url := "http://127.0.0.1:8080/?param=123&param2=test"
        resp, err := http.Get(url)
        if err != nil {
                fmt.Println("error happend", err)
                return
        }
        defer resp.Body.Close() // важный пункт!

        respBody, err := ioutil.ReadAll(resp.Body)

        fmt.Printf("http.Get body %#v\n\n\n", string(respBody))
}
```

Респонс — это довольно большая структура, в которой много разных данных: хедеры, статус ответа, тело. Мы в примере быдем вычитывать и выводить тело. Обратите внимание, что тело всегда надо закрывать, чтобы не было утечки памяти, поэтому закрытие стоит вызывать через defer.

Это был самый простой пример. Как поступать в ситуации, когда мы хотим отправить больше данных с запросом? Вы можете создать создать структуру типа http.Request. В нашем случае надо указывать метод Get, какие-то хедеры. После этого нужно указать url, его можно было сразу указать указать в Request'e, но но мы для разнообразия воспользуемся функцией url.Parse, потому что парсинг url тоже достаточно частая операция, если надо ,например, какие-то параметры выдернуть. Так же можно обращаться к внутренним полям url и выставлять ещё какие-то значения функцией Set. В примере выставляем user, который равен rvasily. Теперь, когда мы скомпоновали запрос, используем дефолтный клиент и обратимся к его методу Do, передав в него наш запрос. Дальше проверяем, не возникла ли ошибка, вычитываем и выводим body, не забывая закрыть его через defer. Получается такой код.

```
func runGetFullReq() {

        req := &http.Request{
                Method: http.MethodGet,
                Header: http.Header{
                        "User-Agent": {"coursera/golang"},
                },
        }

        req.URL, _ = url.Parse("http://127.0.0.1:8080/?id=42")
        req.URL.Query().Set("user", "rvasily")

        resp, err := http.DefaultClient.Do(req)
        if err != nil {
                fmt.Println("error happend", err)
                return
        }
        defer resp.Body.Close() // важный пункт!

        respBody, err := ioutil.ReadAll(resp.Body)
```

```
        fmt.Printf("testGetFullReq resp %#v\n\n\n", string(respBody))
}
```

В двух предыдущих примерах мы использовали обертку над дефолтным клиентом и сам дефолтный клиент. Но част бывает, что использовать его очень неудобно, потому что в дефолтном клиенте нет никаких таймаутов на время ожидания ответа. Из-за этого ваша программа может повиснуть, когда висит внешний сервер. Поэтому лучше никогда не использовать дефолтный клиент, а научиться сразу пользоваться структурой http.Transport, у которого есть куча всяких параметров, например, таймауты, keepalive'ы, сколько у вас коннектов до удалённого сервиса может быть установлено, и когда они будут протухать. Потом, используя этот транспорт, создать клиента. Для настройки транспорта и клиента существует очень много параметров, их лучше подробнее изучить в документации. А мы рассмотрим пример отправки пост-запроса вместе с данными на сервер.

```
func runTransportAndPost() {
        transport := &http.Transport{
                DialContext: (&net.Dialer{
                        Timeout:   30 * time.Second,
                        KeepAlive: 30 * time.Second,
                        DualStack: true,
                }).DialContext,
                MaxIdleConns:          100,
                IdleConnTimeout:       90 * time.Second,
                TLSHandshakeTimeout:   10 * time.Second,
                ExpectContinueTimeout: 1 * time.Second,
        }

        client := &http.Client{
                Timeout:   time.Second * 10,
                Transport: transport,
        }

        data := `{"id": 42, "user": "rvasily"}`
        body := bytes.NewBufferString(data)

        url := "http://127.0.0.1:8080/raw_body"
        req, _ := http.NewRequest(http.MethodPost, url, body)
        req.Header.Add("Content-Type", "application/json")
        req.Header.Add("Content-Length", strconv.Itoa(len(data)))

        resp, err := client.Do(req)
        if err != nil {
                fmt.Println("error happend", err)
                return
        }
        defer resp.Body.Close() // важный пункт!

        respBody, err := ioutil.ReadAll(resp.Body)

        fmt.Printf("runTransport %#v\n\n\n", string(respBody))
}
```

В данной функции задаем нужные параметры для транспорта, создаем своего клиента. Хотим отправить данные data. Для этого прежде всего создаем io.Reader. Задаем url, куда будем их отправлять. И создаем новый Request с методом Post, нужным url, указываем body для него, указываем в хедеры тип Content-Type, и размер данных, которые будем передавать передать. Теперь наш сформированный запрос надо передать на сервер уже при помощи нашего клиента, а не дефолтного. Делается это по проежнему методом Do. Дальше — как обычно: проверяем ошибку, закрываем Body в defer. Вычитываем Body и выводим его на экран.

Чтобы проверить работу всех примеров надо не забыть добавить нужные импорты и написать main.

```
import (
        "bytes"
```

```
        "fmt"
        "io/ioutil"
        "net"
        "net/http"
        "net/url"
        "strconv"
        "time"
)

func main() {
        go startServer()

        time.Sleep(100 * time.Millisecond)

        runGet()
        runGetFullReq()
        runTransportAndPost()
}
```

Запустите, чтоб проверить, что получится.

Тестирование HTTP-запросов и ответов

Мы успели рассмотреть, как отправлять и обрабатывать запросы. Тепер мы научимся тестировать с помощью инструментов, предоставляемых языком Go, хендлеры, то есть функции, которые обрабатывают наши запросы, и функции, которые ходят во внешние сервисы и получают оттуда ответ.
Итак, допустим, у нас есть функция GetUser — это HTTP handler. Она принимает параметр, и, в зависимости от этого параметра, отдает какой-то результат.

```
func GetUser(w http.ResponseWriter, r *http.Request) {
        key := r.FormValue("id")
        if key == "42" {
                w.WriteHeader(http.StatusOK)
                io.WriteString(w, `{"status": 200, "resp": {"user": 42}}`)
        } else {
                w.WriteHeader(http.StatusInternalServerError)
                io.WriteString(w, `{"status": 500, "err": "db_error"}`)
        }
}
```

Мы хотим ее протестировать. В го для этих целей есть пакет net/http/httptest, в котором находятся вспомогательные функции. Мы можем заготовить несколько тесткейсов и для каждого из нихподготовить составлять тестовый запрос с помощью NewRequest, в том виде, в каком он придет в функцию, готовить Recorder и вызывать тестируемую функцию как раз с этими подставными параметрами. И дальше уже проверять, все ли хорошо работает.

```
type TestCase struct {
        ID         string
        Response   string
        StatusCode int
}

func TestGetUser(t *testing.T) {
        cases := []TestCase{
                TestCase{
                        ID:         "42",
                        Response:   `{"status": 200, "resp": {"user": 42}}`,
                        StatusCode: http.StatusOK,
                },
                TestCase{
                        ID:         "500",
                        Response:   `{"status": 500, "err": "db_error"}`,
                        StatusCode: http.StatusInternalServerError,
```

```
                },
        }
        for caseNum, item := range cases {
                url := "http://example.com/api/user?id=" + item.ID
                req := httptest.NewRequest("GET", url, nil)
                w := httptest.NewRecorder()

                GetUser(w, req)

                if w.Code != item.StatusCode {
                        t.Errorf("[%d] wrong StatusCode: got %d, expected %d",
                                caseNum, w.Code, item.StatusCode)
                }

                resp := w.Result()
                body, _ := ioutil.ReadAll(resp.Body)

                bodyStr := string(body)
                if bodyStr != item.Response {
                        t.Errorf("[%d] wrong Response: got %+v, expected %+v",
                                caseNum, bodyStr, item.Response)
                }
        }
}
```

Теперь как быть, если нужно подменить какой-то внешний сервис? Это можно сделать, используя тот же пакет httptest. Рассмотрим следующим пример. Пусть у нас есть структура «корзина», у нее есть упрощенный метод Checkout (оплатить), который работает с платежным шлюзом или платежным сервисом.

```
type Cart struct {
        PaymentApiURL string
}

func (c *Cart) Checkout(id string) (*CheckoutResult, error) {
        url := c.PaymentApiURL + "?id=" + id
        resp, err := http.Get(url)
        if err != nil {
                return nil, err
        }
        defer resp.Body.Close()

        data, err := ioutil.ReadAll(resp.Body)
        if err != nil {
                return nil, err
        }

        result := &CheckoutResult{}

        err = json.Unmarshal(data, result)
        if err != nil {
                return nil, err
        }
        return result, nil
}
```

В ней мы создаем сразу URL, по которому мы идем платить, добавляем id. И дергаем его Get'ом. Проверяем ошибки, закрываем,вычитываем Body, распаковываем json, и опять возвращаем ошибку. И мы хотим проверить, функция работает корректно, то есть мы покрыли все возможные результаты работы удаленного сервиса. Если просто ходить в какой-то внешний сервис, во-первых, сервис у нас будет списывать деньги. Во-вторых, мы не можем гарантировать, что нам хватит денег дождаться какой-то экзотической ошибки, которую надо уметь обрабатывать.

Очень хорошо, что у нас есть PaymentApiURL, мы можем поднять у себя сервер и отправлять запросы уже в него, проставив в PaymentApiURL нужное нам значение локального сервера. Итак, рассмотрим.

```
func TestCartCheckout(t *testing.T) {
        cases := []TestCase{
                TestCase{
                        ID: "42",
                        Result: &CheckoutResult{
                                Status:  200,
                                Balance: 100500,
                                Err:     "",
                        },
                        IsError: false,
                },
                TestCase{
                        ID: "100500",
                        Result: &CheckoutResult{
                                Status:  400,
                                Balance: 0,
                                Err:     "bad_balance",
                        },
                        IsError: false,
                },
                TestCase{
                        ID:      "__broken_json",
                        Result:  nil,
                        IsError: true,
                },
                TestCase{
                        ID:      "__internal_error",
                        Result:  nil,
                        IsError: true,
                },
        }

        ts := httptest.NewServer(http.HandlerFunc(CheckoutDummy))

        for caseNum, item := range cases {
                c := &Cart{
                        PaymentApiURL: ts.URL,
                }
                result, err := c.Checkout(item.ID)

                if err != nil && !item.IsError {
                        t.Errorf("[%d] unexpected error: %#v", caseNum, err)
                }
                if err == nil && item.IsError {
                        t.Errorf("[%d] expected error, got nil", caseNum)
                }
                if !reflect.DeepEqual(item.Result, result) {
                        t.Errorf("[%d] wrong result, expected %#v, got %#v",
                                                caseNum, item.Result, result)
                }
        }
        ts.Close()
}
```

У нас будет четыре тест-кейса. После их задания, мы создаем сервер, указываю CheckoutDummy (её мы рассмотрим позже отдельно), то есть функцию, которая будет обрабатывать приходящие туда запросы. Этот сервер начнет слушать на каком-то случайном порту, и нам вернется его URL. Теперь на него можно отправлять все нужные запросы, не боясь разориться, и проверять результат работы на ошибки, сравнивать с ожидаемым результатом.

Давайте теперь посмотрим на функцию CheckDummy.

```
func CheckoutDummy(w http.ResponseWriter, r *http.Request) {
        key := r.FormValue("id")
        switch key {
        case "42":
                w.WriteHeader(http.StatusOK)
                io.WriteString(w, `{"status": 200, "balance": 100500}`)
        case "100500":
                w.WriteHeader(http.StatusOK)
                io.WriteString(w, `{"status": 400, "err": "bad_balance"}`)
        case "__broken_json":
                w.WriteHeader(http.StatusOK)
                io.WriteString(w, `{"status": 400`) //broken json
        case "__internal_error":
                fallthrough
        default:
                w.WriteHeader(http.StatusInternalServerError)
        }
}
```

CheckDummy — это функция, которую мы отдаем входным параметром при создании тестирующего сервера, которая должна отвечать за обработку приходящих на этот сервер запросов. Она должна уметь возвращать все возможные результаты работы, ожидаемые от внешнего сервиса, и внештатные результаты работы, от которых мы хотим защититься. В данном случае она может вернуть успешный запрос, плохой запрос, но так, будто бы это штатная ситуация, может вернуть кривой json, либо же ошибку 500, что значит сервер вообще недоступен. Запустите и убедитесь, что все тесты работают.

Шаблонизация

Inline-шаблоны и шаблоны из файлов

Теперь рассмотрим работу с шаблонами, которые предоставляет стандартная библиотека Go. Шаблонизация в виде отдельного шаблона — признак хорошего тона. Уже никто не мешает код вместе с логикой и не выводит его построчно по одному тегу на экран, как это было давным-давно в темные времена веб-разработки.

Шаблоны в Go есть в нескольких вариантах. Первый, из которых мы рассмотрим — это text/template — простые шаблоны. Они все выводят, как есть. Для начала давайте определим параметры шаблонизатора URL и Browser. И шаблон, в котором будем выводить непосредственно браузер и URL, который передали.

```
type tplParams struct {
        URL     string
        Browser string
}

const EXAMPLE = `
Browser {{.Browser}}

you at {{.URL}}
`
```

Для доступа к переменным используем двойные фигурные скобки и точку. Точка — это текущий элемент, с которым мы работаем. В данном случае текущий элемент — это непосредственно tplParams, который я передам. В случае, если у нас будет какой-то цикл по значениям, то точка будет являться элементом цикла. Итак, теперь давайте воспользуемся шаблоном.

```
func handle(w http.ResponseWriter, r *http.Request) {
        tmpl := template.New(`example`)
        tmpl, _ = tmpl.Parse(EXAMPLE)
```

```
        params := tplParams{
                URL:     r.URL.String(),
                Browser: r.UserAgent(),
        }

        tmpl.Execute(w, params)
}
```

В начале мы компилируем шаблон. Мы делаем это не очень быстро, потому что компилируем его на каждый вызов http реквеста, но для примера нам хватит. Теперь создадим параметры, используя то, что нам пришло из браузера. И непосредственно установим их в шаблон, «экспанднем» шаблон. В функцию, делающую это, передаем ResponseWriter, то есть результат экспана шаблона будет передан сразу в браузер, и параметры. Если запустить, браузер выведет свое имя и URL, на котором выне ходитесь.

Простые текстовые шаблоны хороши в случаях, когда нам не надо ничего эскейпить. В них даже нет возможности передать какие-то данные от пользователя. В новом примере мы рассмотрим http/template. Он более сложен, чем text/template, потому что он добавляет необходимое экранирование к вводимым символам, причем он знает, какое экранирование должно быть. То есть в URL-ах он сделает URL escape, в HTML он заэскейпит все HTML сущности, чтобы нельзя было сделать какой-нибудь XSS.

Мы будем экспандить не из подготовленного куска внутри нашего кода, а из внешнего файла. В нем будет такой html-код.

```
<html>
<body>
        <h1>Users</h1>
        {{range .Users}}
                <b>{{.Name}}</b>
                {{if .Active}}active{{end}}
                <br />
        {{end}}
</body>
</html>
```

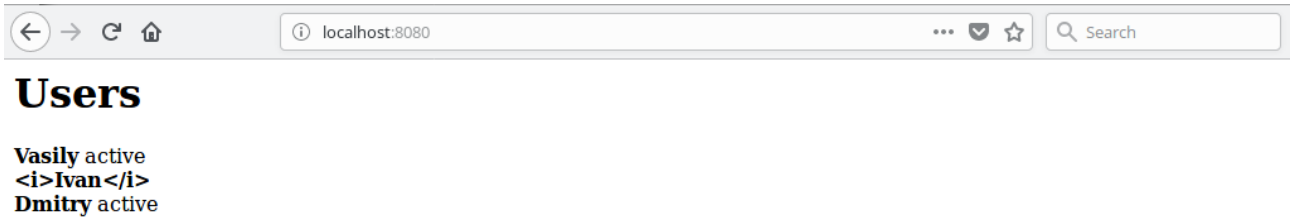
У нас простой шаблон, в котором просто идем по всем пользователям и выводим имя, если пользователь активен, пишем, что она активен. Теперь перейдем собственно к самой программе.

```
package main

import (
        "fmt"
        "html/template"
        "net/http"
)

type User struct {
        ID     int
        Name   string
        Active bool
}

func main() {
        tmpl := template.Must(template.ParseFiles("users.html"))

        users := []User{
                User{1, "Vasily", true},
                User{2, "<i>Ivan</i>", false},
                User{3, "Dmitry", true},
        }

        http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
                tmpl.Execute(w,
```

```
                        struct {
                                Users []User
                        }{
                                users,
                        })
        })

        fmt.Println("starting server at :8080")
        http.ListenAndServe(":8080", nil)
}
```

Html-код мы парсим всего один раз при старте сервера. Затем создаем пользователей, каждый юзер имеет ID, имя и флаг, он активен или нет. Обратите внимание: у Ивана имя заключено в тег какой-то. Теперь мы уже будем экспандить этот шаблон. Мы вызываем tmpl.Execute, передаем туда Writer, в качестве параметра используем не заранее подготовленную структуру юзеров, как в предыдущем примере, а определяем её прямо на месте. и сразу же создаем экземпляр этой структуры, передавая в него наших пользователей.

Теперь надо запустить. На экран выведутся наши пользователи. Василий активен, и Дмитрий активен. Иван не активен. Обратите внимание: теги в имени Ивана заменились на HTML-сущности.



Вызов методов и функций из шаблонов

В прошлом разделе мы познакомились с базовыми операциями-шаблонами, узнали, как экспандить шаблон. Это хорошо, что мы можем передать туда любые структурированные данные, целые структуры. Но можем ли мы вызывать методы у этих структур? Это было бы очень удобно.

Стандартный шаблонизатор Go позволяет такие операции. Давайте рассмотрим следующий код.

```
type User struct {
        ID     int
        Name   string
        Active bool
}

func (u *User) PrintActive() string {
        if !u.Active {
                return ""
        }
        return "method says user " + u.Name + " active"
}

func main() {
        tmpl, err := template.
                New("").
                ParseFiles("method.html")
        if err != nil {
                panic(err)
        }

        users := []User{
                User{1, "Vasily", true},
                User{2, "Ivan", false},
                User{3, "Dmitry", true},
```

```
        }

        http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
                err := tmpl.ExecuteTemplate(w, "method.html",
                        struct {
                                Users []User
                        }{
                                users,
                        })
                if err != nil {
                        panic(err)
                }
        })

        fmt.Println("starting server at :8080")
        http.ListenAndServe(":8080", nil)
}
```

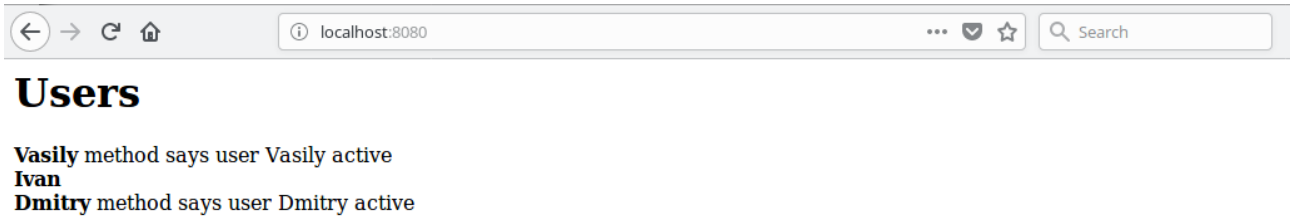
Создадим шаблон из файла method.html, добавим туда юзеров и выполним этот шаблон. Чем это отличается от того кода, что мы видели? Отличается это тем, что теперь у пользователя есть метод PrintActive(), который возвращает строку. Если пользователь неактивен, мы возвращаем пустую строку, иначе мы говорим, что пользователь с таким именем активен. Как теперь вызвать этот метод?

Рассмотрим шаблон.

```
<html>
<body>
        <h1>Users</h1>
        {{range .Users}}
                <b>{{.Name}}</b>
                {{.PrintActive}}
                <br />
        {{end}}
</body>
</html>
```

Мы итерируемся по пользователям, выводим его имя и указываем метод, который нужно вызвать. В данном случае это PrintActive().

Давайте запустим и посмотрим, что получается. Как и ожидалось, метод Василия и Дмитрия говорит, что этот пользователь активен, У Ivan'а никакой активности нет.



Вызывать функции у структур — это хорошо и полезно, но иногда хочется иметь какие-то общие методы, которые будут иметь возможность работать с любыми данными, например, с int'ами, или ещё с чем-нибудь. То есть прокинуть обособленные функции, которые уже не являются методами структуры. Покажем, как это сделать на примере функции IsUserOdd, которая не является методом структуры, а принимает структуру на вход и говорит, является ли идентификатор этого нечётным числом.

```
func IsUserOdd(u *User) bool {
        return u.ID%2 != 0
}

func main() {
        tmplFuncs := template.FuncMap{
```

```
                "OddUser": IsUserOdd,
        }

        tmpl, err := template.
                New("").
                Funcs(tmplFuncs).
                ParseFiles("func.html")
        if err != nil {
                panic(err)
        }

        users := []User{
                User{1, "Vasily", true},
                User{2, "Ivan", false},
                User{3, "Dmitry", true},
        }

        http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
                err := tmpl.ExecuteTemplate(w, "func.html",
                        struct {
                                Users []User
                        }{
                                users,
                        })
                if err != nil {
                        panic(err)
                }
        })

        fmt.Println("starting server at :8080")
        http.ListenAndServe(":8080", nil)
}
```

Отличие данного кода от предыдущего в том, что мы первым делом создаём карту функции, которую хотим прокинуть в шаблонизатор. А при создании шаблонизатора указываем, что, пожалуйста, используй вот эти вот прокинутые функции. Переобъявлять функции потом нельзя. Дальше код ничем не отличается — мы парсим сам шаблон пользователя и выполненяем его.

Как выглядит сам шаблон?

```
<html>
<body>
        <h1>Users</h1>
        {{range .Users}}
                <b>{{.Name}}</b>,
                {{if OddUser .}}
                        id {{.ID}} is odd
                {{end}}
                <br />
        {{end}}
</body>
</html>
```

Мы итерируемся по пользователю, выводим его имя, и теперь мы вызываем функцию OddUser, которую прокидываем в шаблон, от точки. Точка — это текущий элемент итерации, то есть текущий объект User'а. И если условие истинно, то мы вводим какой-то текст.

Профилирование веба

Профилирование через pprof

Мы уже рассматривали, как пользоваться профилировщиком pprof, теперь мы рассмотрим применение профилировщика pprof для поиска горячих мест в уже работающей программе под боевой нагрузкой. Профилировщик можно подключить, как обработчики каких-то URL-ов, по которым мы можем снять либо профиль CPU, либо хип-дамп памяти. Выполняется это посредством импортирования пакета net/http/pprof. Поскольку мы не будем к нему обращаться непосредственно в программе, нужно дописать префикс нижнее подчеркивание. Это будет означать, что пакет только регистрирует некоторые свои обработчики в глобальном обработчике функций.
Мы хотим протестировать следующую программу.

```
package main

import (
        "fmt"
        "net/http"
        _ "net/http/pprof"
        "time"
)

type Post struct {
        ID       int
        Text     string
        Author   string
        Comments int
        Time     time.Time
}

func handle(w http.ResponseWriter, req *http.Request) {
        s := ""
        for i := 0; i < 1000; i++ {
                p := &Post{ID: i, Text: "new post"}
                s += fmt.Sprintf("%#v", p)
        }
        w.Write([]byte(s))
}

func main() {
        http.HandleFunc("/", handle)

        fmt.Println("starting server at :8080")
        fmt.Println(http.ListenAndServe(":8080", nil))
}
```

В ней определена какая-то структура, есть обработчик запроса, который просто создает в цикле структуру, приводит ее к строке, и приписывает её к общему результату, который потом выводится наверх. То есть программа обрабатывает много-много каких-то данных.

Для тестирования надо запустить программу и подать на неё нагрузку, например с помощью стандартной утилиты ab.

```
go build -o pprof_1.exe pprof_1.go && ./pprof_1.exe

ab -t 300 -n 1000000000 -c 10 http://127.0.0.1:8080/
```

Затем можно снять хип-дамп и профиль CPU, для этого нужно дернуть URL-ы debug/pprof/heap и debug/pprof/profile для снятия CPU, при этом мы можем указать в параметре seconds, за сколько времени мы хотим снять профиль CPU.

```
curl http://127.0.0.1:8080/debug/pprof/heap -o mem_out.txt
curl http://127.0.0.1:8080/debug/pprof/profile?seconds=5 -o cpu_out.txt
```

Далее, используя саму команду pprof, мы можем построить .svg файл по полученным результатам.

```
go tool pprof -svg -alloc_objects pprof_1.exe mem_out.txt > mem_ao.svg
go tool pprof -svg pprof_1.exe cpu_out.txt > cpu.svg
```

То есть это уже не обязательно делать под боевой нагрузкой. Мы можем снять хип-дамп или профиль процессора, увести их туда, где никакой нагрузки нет, где это не затронет пользователей, и там уже строить, и там уже искать узкие места. В результате получаются наглядные графы используя которые, мы можем хотя бы найти, в каких местах, где у нас есть горячие места, их даже визуально видно. То есть это очень мощный инструмент для отладки программ в продакшене без того, чтобы пытаться создать такого рода нагрузку синтетически.

Поиск утечки горутин

Еще один мощный инструмент, который предоставляет метод pprof, это возможность снять стек-трейс всех горутин, которые есть в программе. Давайте рассмотрим это на примере утечки горутин. Пусть у нас есть функция getPost, которая в цикле делает слайс некоторых структур и записывает их в канал, который к нам приходит извне, и есть handle, который запускает горутину для получения этих постов.

```
type Post struct {
        ID       int
        Text     string
        Author   string
        Comments int
        Time     time.Time
}

func getPost(out chan []Post) {
        posts := []Post{}
        for i := 1; i < 10; i++ {
                post := Post{ID: 1, Text: "text"}
                posts = append(posts, post)
        }
        out <- posts
}

func handleLeak(w http.ResponseWriter, req *http.Request) {
        res := make(chan []Post)
        go getPost(res)
}
```

Обратите внимание, что структуры из канала никто не вычитывает, то есть в горутины начнут бесконечно ожидать, пока из нее кто-то вычитает, и это может привести к утечке памяти. В случае, если у вас вдруг программа может начать расти, и вы подозреваете где-то утечку горутин, можно попробовать снять стек-трейс всех горутин и узнать, чего они ждут.

Делается это теми же средствами — программа запускается и на неё дается небольшую нагрузку. Затем выполняется снятие трейса горутин.

```
go build -o pprof_2.exe pprof_2.go && ./pprof_2.exe

ab -n 1000 -c 10 http://127.0.0.1:8080/

curl http://localhost:8080/debug/pprof/goroutine?debug=2 -o goroutines.txt
```

Теперь достаточно открыть файл goroutines.txt и посмотреть, что появилось огромное количество одинаковых строчек. В них написано, что горутина номер такой-то ожидает отправки в канал на какой-то строчке. Как раз перечислены те строчи, на которых мы вызываем горутину и пишем в канал, на котором горутина висит.

То есть используя полный стек-трейс, мы можем найти утечку опять-таки без синтетической нагрузки, прямо во время реальной боевой нагрузки.

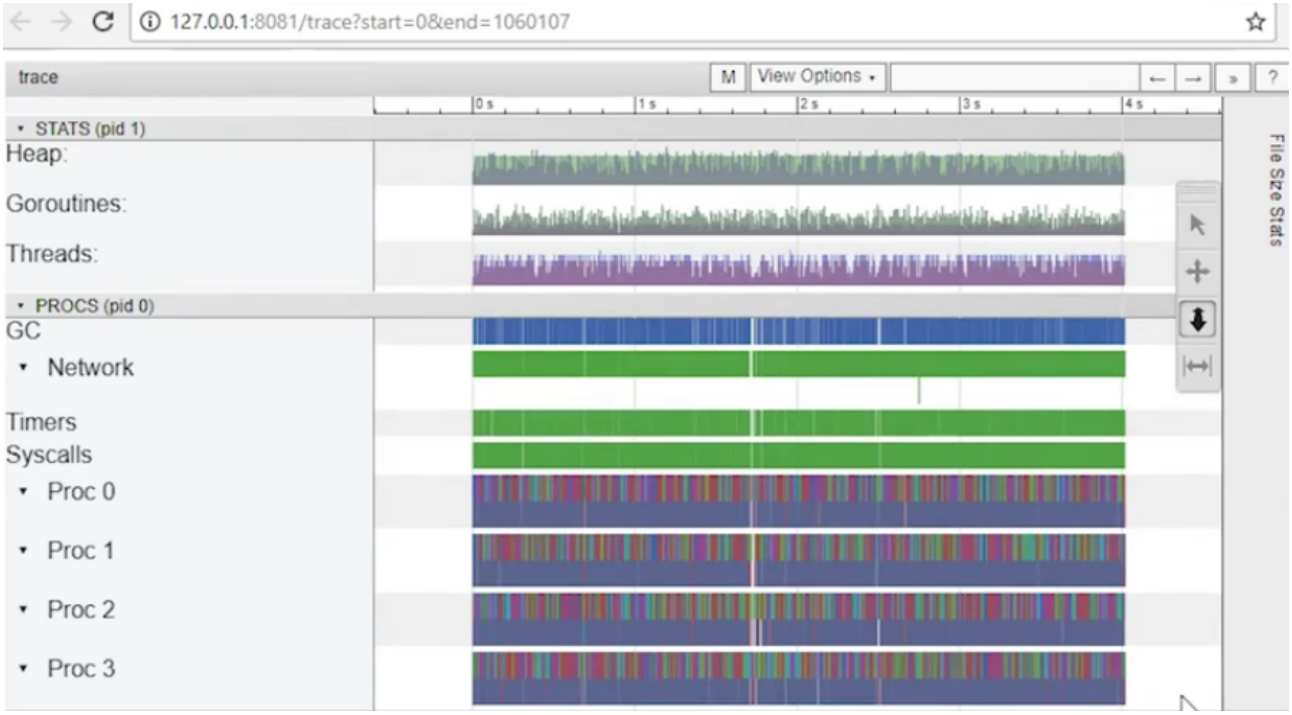
Трассировка поведения сервиса

Если рассматривать граф, который получается через pprof, например, граф аллокации памяти, либо же граф использования процессора, то мы видим, что происходило, как бы с высоты птичьего полета. Но иногда нам хочется посмотреть, а как развивались события. То есть не что получилось в итоге, в целом, а что именно шло в процессе работы программы: где были задержки, когда случались аллокации, когда были остановки на сборку мусора. В Go опять-таки есть для этого инструмент, называется он трассировка.

Для того чтобы с ним работать, надо подготовиться — запустить программу, как и в предыдущих примерах дать нагрузку, собрать трассировку за несколько секунд. Затем можно запустить сам трассировщик, который позволит визуализировать результат трассировки. Делается это при помощи команды go tool trace, в которой указываем флаг http, на каком порту нам разворачивать этот сервер. бинарник и дамп с данными, которые мы собрали с работающей программы.

```
go build -o tracing.exe tracing.go && ./tracing.exe

ab -t 300 -n 10000000 -c 10 http://127.0.0.1:8080/

curl http://localhost:8080/debug/pprof/trace?seconds=10 -o trace.out

go tool trace -http "0.0.0.0:8081" tracing.exe trace.out
```

Еще раз отдельно следует обратить внимание, что такого рода трассировку можно снять с работающей в продакшне программы под нагрузкой, и уже разбираться с ней на своей разработческой машине, никак не влияя на продакшн. Трассировка немножко замедляет время работы, снять несколько секунд трассировки в продакшне вполне допустимо. При этом сама трассировка займет какое-то время и результат работы получится не маленький,



Перед нами общий экран трассировки. Слева идет некая информация Heap'а, сколько у нас памяти было задействовано, количество горутин, которые у нас одновременно работали, количество системных тредов, в которых работали эти горутины. Еще информация, сеть, syscalls и процессоры, в которых крутятся наши треды, а в тредах крутятся наши горутины. Меняя масштаб и сдвигаясь, можно посмотреть что происходило в различные периоды и собрать какую-то другую статистику. Для каждой горутины можно посмотреть, что происходило во время её работы, более того, на каких процессах она выполнялась.

Однако, трассировка вовсе не панацея, но при помощи этого инструмента тоже можно получить какую-то информацию, если вдруг у вас есть нет понимания, что происходит в вашей программе. Когда случались паузы на всю программу, сколько они занимают, сколько занимают и работают отдельные горутины. К сожалению или к счастью, в Go нельзя пометить отдельную горутину именем, но тем не менее, используя этот инструмент, мы можем посмотреть, какая горутина сколько времени работала. Этот инструмент так же не поможет вам поймать некоторые утечки памяти, которые хорошо видны в pprof. Однако, если вы начинающий разработчик, то это очень полезный инструмент для изучения того, что и как работало.

Телеграм бот

Пример с telegram-ботом

В этом разделе рассмотрим создание Telegram бота на Go. Мы не будем писать обработку совсем всех событий с нуля, а воспользуемся готовой библиотекой. Есть готовая библиотека Telegram Bot API. Для начала нужно ее установить. Устанавливается она через команду

```
go get gopkg.in/telegram-bot-api.v4
```

Пакет скачается и установится в gopath. Затем нам зарегистрировать нашего бота в специализированном боте BotFather и получить токен. Последняя подготовительная часть — нам необходимо каким-то образом выложить бота в Интернет либо еще как-то сделать, чтобы Telegram мог до него достучаться. В этом примере мы воспользуемся сервисом ngrok. Этот сервис не стоит использовать в продакшене и и лучше не стоит доверять ему данные и держать постоянно запущенным, однако для целей учебной разработки он подойдет. После запуска он выдает https-адрес, при обращении к которому будет перенаправлен на мой локальный порт.

Теперь уже можно перенести полученные адрес и токен в константы. Что наш бот будет уметь делать? Наш бот будет выкидывать нам rss с Habrahabr. Итак, у нас еще есть какой-то список rss.

```
package main

import (
```

```
        "encoding/xml"
        "fmt"
        "gopkg.in/telegram-bot-api.v4"
        "io/ioutil"
        "net/http"
)

const (
        BotToken   = "310805560:AAENzjDSJPKABY9Hw1GZOdKBxxrhOHkfo_k"
        WebhookURL = "https://ea731f5c.ngrok.io"
)

var rss = map[string]string{
        "Habr": "https://habrahabr.ru/rss/best/",
}

type RSS struct {
         Items []Item `xml:"channel>item"`
         }

type Item struct {
        URL   string `xml:"guid"`
        Title string `xml:"title"`
}
```

Теперь можно перейти собственно к созданию бота. Прежде всего мы создаем объект бота, указываем ему токен, который мы создали в BotFather, он инициализируется, отправив команды в Telegram. Потом мы говорим, что мы авторизованы таким-то аккаунтом, устанавливаем WebhookURL — Наш WebhookURL — это тот URL, который как раз нам выдал ngrok — и начинаем слушать сервер. Это значит, что мы просто ставим https-handle, который будет обрабатывать по этому URL все команды, которые приходят, как сообщение от Telegram. В отдельной горутине стартуем веб-сервер, который непосредственно будет обрабатывать https-запросы. Сообщения будут представлять из себя обычный json, обработчик будет его парсить, распаковывать и возвращать удобную структуру. Сообщения будем получать из канала updates, в цикле range висеть и слушать, когда там что-то приходит. Если пришло сообщение и у нас есть такой rss url, в данном случае только Habr, мы получим его новости. Если всё плохо, то мы скажем, что извините, ошибка. Если всё хорошо, мы увидим эти новости в виде сообщения. Вывод используется при помощи команды bot.Send. Сразу используем простое сокращение для объекта нового сообщения, где указываем идентификатор, кому отправить, и просто URL и заголовок этой новости.

```
func main() {
        bot, err := tgbotapi.NewBotAPI(BotToken)
        if err != nil {
                panic(err)
        }

        // bot.Debug = true
        fmt.Printf("Authorized on account %s\n", bot.Self.UserName)

        _, err = bot.SetWebhook(tgbotapi.NewWebhook(WebhookURL))
        if err != nil {
                panic(err)
        }

        updates := bot.ListenForWebhook("/")

        go http.ListenAndServe(":8080", nil)
        fmt.Println("start listen :8080")

        // получаем все обновления из канала updates
        for update := range updates {
                if url, ok := rss[update.Message.Text]; ok {
                        rss, err := getNews(url)
```

```
                        if err != nil {
                                bot.Send(tgbotapi.NewMessage(
                                        update.Message.Chat.ID,
                                        "sorry, error happend",
                                ))
                        }
                        for _, item := range rss.Items {
                                bot.Send(tgbotapi.NewMessage(
                                        update.Message.Chat.ID,
                                        item.URL+"\n"+item.Title,
                                ))
                        }
                } else {
                        bot.Send(tgbotapi.NewMessage(
                                update.Message.Chat.ID,
                                `there is only Habr feed availible`,
                        ))
                }

        }
}
```

Каким образом будем получать новости? Новости получаем из URL get запросом на этот URL, вычитываем и распаковываем xml, что тоже очень просто, очень быстро используется, всего лишь две структуры. Конечно, не распаковываем всё, только самый-самый минимум.

```
func getNews(url string) (*RSS, error) {
        resp, err := http.Get(url)
        if err != nil {
                return nil, err
        }

        defer resp.Body.Close()
        body, _ := ioutil.ReadAll(resp.Body)

        rss := new(RSS)
        err = xml.Unmarshal(body, rss)
        if err != nil {
                return nil, err
        }

        return rss, nil
}
```

Такими простыми действиями буквально меньше 100 строк можно создать Telegram бота, который будет отвечать на ваши команды и что-то делать. Это довольно большой API, там есть много разных типов сообщений. Вы можете послать картинку, документ, и в целом на Go писать такого рода сервисы довольно просто и быстро.