Разработка веб-сервисов на Go, часть 2. Лекция 4

Конфигурирование сервиса

Локальный конфиг - flags, json, ldflags

В этой главе мы поговорим про конфигурирование сервиса. Начнем со стандартного пакета flag. Пакет flag нужен для парсинга аргументов командной строки, в нем есть довольно много функций для парсинга стандартных типов. Например, с помощью следующей строчки в переменную commentsEnabled распарсится булева переменная, причем парситься будет не только true и false, но и t, f, маленькими, большими буквами, 0 и 1.

```
commentsEnabled = flag.Bool("comments", false, "Enable comments after post")
```

Возвращает эта функция адрес на переменную. В качестве аргументов принимает имя флага, по которому нужно вытащить значение из аргументов командной строки, значение по умолчанию и комментарий, который будет выведен, как справка, в случае ошибки при вводе флагов. Так же из флагов можно получить Int, String и еще некоторые значения.

Если же вы хотите парсить что-то другое, нужно чтобы он соотвествовал требуемому интерфейсу, а именно у него булы функции String и Set. Например, тип вот так можно объявить тип AddrList, содержащий внутри всего лишь слайс стрингов.

```
type AddrList []string

func (v *AddrList) String() string {
        return fmt.Sprint(*v)
}

func (v *AddrList) Set(in string) error {
        for _, addr := range strings.Split(in, ",") {
                ipRaw, _, err := net.SplitHostPort(addr)
                if err != nil {
                        return fmt.Errorf("bad addr %v", addr)
                }
                ip := net.ParseIP(ipRaw)
                if ip.To4() == nil {
                        return fmt.Errorf("invalid ipv4 addr %v", addr)
                }
                *v = append(*v, addr)
        }
        return nil
}
```

Первая функция String для дампа значений, вторая функция — Set, которая устанавливает значения в переменную. Таким образом, вы можете устанавливать собственные типы и парсить в том виде, в котором вам это нужно. Для того чтобы спарсить произвольный тип нужно сначала объявить отдельную переменную, а потом вызвать функцию flag.Var.

```
var commentsServices = &AddrList{}

func init() {
        flag.Var(commentsServices, "servers", "Comments number per page")
}

func main() {
        flag.Parse()
...
}
```

Это нужно, поскольку функция flag.Var не умеет возвращать значение нужного типа. В данном случае я объявляем переменную commentsServices и в init указываем, что спарсить нужно туда. init выполняется до функции main. Есть небольшой нюанс, который надо учитывать: аргументы, которые парсятся через пакет flag нельзя использовать в init. Дело в том, что до команды flag.Parse в main'е во всех переменных, которые вы указали во flag, будут значения по умолчанию.

Следующим подходом к парсингу конфига является указание одной большой структуры и, например, парсинг туда значений из json, или yaml, toml, ini, xml — откуда угодно.

```
type Config struct {
        Comments bool `json:"comments"`
        Limit    int
        Servers  []string
}

var (
        config = &Config{}
)
```

Минус такого подхода — в том, что здесь больше нет значений по умолчанию. В случае, если в конфиге нет какого-то параметра, как, например, Кроме того подход имеет право на жизнь в том случае, если ваша программа умещается в один файл и нет никаких других пакетов, других файлов. Потому что если у вас есть другие пакеты, то сразу же ваш пакет начинает зависеть не только от самого себя, но и от какого-то внешнего пакета, и логика размазывается по нескольким пакетам, легко что-то забыть.

Так же при конфигурировании пакетов может возникнуть желание при сборке указать какие-то значения, которые будут для вашего бинарника постоянными. Это не совсем конфигурирование приложения, однако это часто встречающийся нюанс, который стоит знать. В Go есть параметр к сборщику ldflags, в котором можно указать значение переменной, которую вы хотите установить при компиляции. Для этого нужно указать пакет и, собственно, имя переменной. Имя переменной должно иметь тип String. Например, при компиляции следующей программы

```
package main

import (
        "fmt"
)

var (
        Version = ""
        Branch  = ""
)

func main() {
        fmt.Println("[start] starting version ", Version, Branch)
}
```

с флагом ldflag

```
go run -ldflags="-X 'main.Version=$(git rev-parse HEAD)'
-X 'main.Branch=$(git rev-parse --abbrev-ref HEAD)'" ldflags.go
```

значения, указанные, как аргументы флага, запишутся в переменные в программе. Version и Branch указаны пустые, однако при сборке указали ldfalgs main.Version и main.Branch и значения подтянутся прямо из git'a. Таким образом, всегда, когда приложение будет стартовать, вы будуту знать, из какого хеша оно было собрано. Таким образом, всегда сожно знать не только версию пакета, например, а и версию приложения — именно версию, которая лежит в системе контроля версия, она будет зашита сразу в бинарник.

Далее мы рассмотрим, каким образом можно сделать онлайн-конфигурирование сервиса, используя Consul.

Удалённый конфиг, используем Consul

В этом разделе мы поговорим про то, каким образом можно реализовать изменение конфига без перезагрузки программы. Часто это называется online config или hot reload config. Зачем нужен и почему это удобно? Дело в том, что часто хочется менять config без того, чтобы перезагружать сервис, выводить его из нагрузки, например, а используя какую-то админку. В примере в качестве админки будем использовать консул, где есть key value хранилище, которое можно организовать, как config. Создайте там папку myapi, и добавьте несколько полей: api_token и max_length и session_addr. Согласитесь, добавлять config таким образом довольно удобно. Очень опасно, безусловно, но удобно.

Теперь давайте посмотрим, что происходит в коде.

```
func main() {
        flag.Parse()

        var err error
        config := consulapi.DefaultConfig()
        config.Address = *consulAddr
        consul, err = consulapi.NewClient(config)

        if err != nil {
                fmt.Println("consul error", err)
                return
        }

        loadConfig()
        go runConfigUpdater()

        siteMux := http.NewServeMux()
        siteMux.HandleFunc("/", loadPostsHandle)

        siteHandler := configMiddleware(siteMux)

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

В самом начале мы парсим флаги, потому что совсем всё сделать онлайн конфигом не получится. Хотя бы адрес консула нужно откуда-то передавать. Далее мы подключаемся к консулу, загружаем config, запускаем его онлайн обновление в отдельной горутине, которая будет работать постоянно. Дальше мы регистрируем обработчик запросов и оборачиваем его в configMiddleware.

Что происходит внутри, начнем рассматривать со стороны загрузки config.

```
func loadConfig() {
        qo := &consulapi.QueryOptions{
                WaitIndex: consulLastIndex,
        }
        kvPairs, qm, err := consul.KV().List(cfgPrefix, qo)
        if err != nil {
                fmt.Println(err)
                return
        }
```

```
        fmt.Println("remote consulLastIndex", qm.LastIndex)
        if consulLastIndex == qm.LastIndex {
                fmt.Println("consulLastIndex not changed")
                return
        }

        newConfig := make(map[string]string)

        for idx, item := range kvPairs {
                if item.Key == cfgPrefix {
                        continue
                }
                fmt.Printf("item[%d] %#v\n", idx, item)
                key := prefixStripper.Replace(item.Key)
                newConfig[key] = string(item.Value)
        }

        globalCfgMu.Lock()
        globalCfg = newConfig
        consulLastIndex = qm.LastIndex
        globalCfgMu.Unlock()

        fmt.Printf("config updated to version %v\n\t%#v\n\n",
                                    consulLastIndex, newConfig)
}
```

При загрузке конфига создаем сначала QueryOptions. Когда мы рассматривали пример с консулом в grpc для service discovery, мы использовали polling для проверки, не произошли ли изменения. Polling значит, что мы периодически опрашивали внешний сервис для получения какой-то информации. В данном случае немножко оптимизируем этот механизм, и не опрашиваем постоянно, а предаем туда последний известный индекс консула. Это своего рода инкремент, который обновляется с каждым обновлением консула. Таким образом, обновление прилетает сразу же, как только оно появилось на стороне консула. Далее, получаем config, обращаясь к сервису key value в консул, чтобы получить список из префикса. Префикс это myapi/ — имя папки, в которой мы создали конфиги в консуле. Для получения списка передаем опцию "ждать последнего индекса". По умолчанию при старте программы индекс равен нулю, поэтому consul в первый раз ответит сразу. Получив список значений и какую-то информацию о самом консуле, берем LastIndex и обновляем его, то есть печатаем на той стороне номер последнего индекса. Если индекс не изменился, пишем соответственно, что он не изменился. Далее, создаем map с новым config'ом, это map типа string string. Consul возвращает данные как слайс байт, но мы будем интерпретировать их, как строки. Будем убирать префикс папки из ключа и класть его в map. После этого надо под локом обновить старый глобальный конфиг.

Сейчас конфиг сделан в виде глобалоной переменной. Это вызывает определенные проблемы. Может возникнуть такой случай, что у вас под каким-то config в вашем приложении включается или выключается какой-то функционал в начале вашего запроса и в конце вашего запроса. Однако тот функционал, который включается в конце, требует того, что должно быть проинициализировано в начале. Если где-то в середине запроса произойдет обновление config, этой глобальной map, то может возникнуть рейс. Рейс даже не по обращению к map, а именно по самой логике. Поэтому даже покрытие локом не поможет. Чтобы понять, как быть и что делать, посмотрим функцию configMiddleWare.

```
func configMiddleware(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
                ctx := r.Context()

                globalCfgMu.RLock()
                localCfg := globalCfg
                globalCfgMu.RUnlock()

                ctx = context.WithValue(ctx,
                        configKey,
                        localCfg)
                next.ServeHTTP(w, r.WithContext(ctx))
```

```
        })
}
```

В ней в контекст записываем локальный config. Из глобального конфига получаем его под ридлоком, то есть берем информацию только на чтение без изменений. Поэтому много конкурентных запросов, в этом месте тормозить не будут. В чём весь смысл получения локального конфига? А весь смысл этой операции в том, что map — это ссылочный тип. Таким образом, получаем ссылку на map, и дальше прокидываем ее вместе с контекстом везде, через контекст WithValue. И, таким образом, когда я будем получать map из контекста, будем всегда обращаться к одной и той же map на протяжении всего запроса. В случае, если глобальный конфиг обновится, у него поменяется ссылка, причем под локом. То есть оригинальная map, которая еще используется в каких-то запросах не поменяется. Как только меняем ссылку на GlobalCfg, то, что там находится внутри, будет лежать только в тех контекстах, которые использует старый config, как только все эти контексты завершатся, выйдут из памяти, то старый config просто уберется сборщиком мусора.

Получить конфиг можно через ctxValue.

```
func ConfigFromContext(ctx context.Context) (map[string]string, error) {
        cfg, ok := ctx.Value(configKey).(map[string]string)
        if !ok {
                return nil, fmt.Errorf("config not found")
        }
        return cfg, nil
}
```

Обращаемся к config. В контекст Value, напомним, лежит пустой интерфейс. Получаем значение, преобразуем, проверяем, нет ли ошибки.

Стоит отметить, что рассмотренный пример — не продакшн вариант, но должен натолкнуть на правильное решение. Какие есть недостатки у этого config? Во-первых, это не структурированная map строк, то есть вы можете опечататься или забыть нужный ключ, или указать его не в том формате, поэтому, сюда стоит провести какую-то валидацию с сообщением о том, что config неправильный. Либо хотя бы писать Json, который вы будете распаковывать в нужную структуру. По-прежнему есть некоторые опасности при использовании онлайн конфига. Конечно, это создает некоторое параллельное api, но иногда всё-таки бывает нужно бизнес-логику конфигурировать быстро. И тогда онлайн конфиг — отличный вариант. Если же вы будете конфигурировать, например, адреса к каким-то базам данных или каким-то микросервисам, то тут могут возникнуть проблемы в том, что вам нужно обрабатывать код для переинициализации, вам нужно покрывать эти сервисы локом для того, чтобы не обратиться случайно туда, где уже никакого сервиса нет. Поэтому к этому нужно подходить осторожно.

Мониторинг

Зачем нужен мониторинг и что мониторить

В этой главе мы затронем очень важную для любого сервера, находящегося под нагрузкой, тему, а именно мониторинг. Если нет мониторинга, то непонятно, жив сервер или он не работает. Может быть, там постоянно идут ошибки, либо после последнего обновления очень сильно упала производительности. Для начала научимся получать базовые метрики: сколько памяти занимает сервер, какое количество горутин работают, и рассмотрим, какие инструменты есть в Go для этих целей.

Начнем с пакета, который называется expvar. Пакет expvar позволяет зарегистрировать debug-обработчик в стандартном HTTP-хендлере. Рассмотрим следующий пример.

```
package main

import (
        "fmt"
        "net/http"
        "runtime"

        "expvar"
)
```

```
var (
        hits = expvar.NewMap("hits")
)

func handler(w http.ResponseWriter, r *http.Request) {
        hits.Add(r.URL.Path, 1)
        w.Write([]byte("expvar increased"))
}

func init() {
        expvar.Publish("mystat", expvar.Func(func() interface{} {
                hits.Init()
                return map[string]int{
                        "test":          100500,
                        "value":         42,
                        "goroutine_num": runtime.NumGoroutine(),
                }
        }))
}

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

        fmt.Println("starting server at :8081")
        http.ListenAndServe(":8081", nil)
}
```

После этого по специальному URL, .../debug/vars — будет доступна некоторая статистика по серверу. Например, по умолчанию там идут параметры командной строки, из которой была запущена программа, и статистика по памяти. Основное, что у нас может интересовать — это размер хипа — HeapInuse, то есть количество байт, которые программа аллоцировала в динамической памяти. StackInuse — количество памяти, которое находится на стеке. И разного рода информация по garbage collector, например, когда последний раз был запуск сборщика мусора, сколько времени он занимал и прочая информация. Этого достаточно, чтобы сделать простой мониторинг — хотя бы дергать сервер и смотреть, живой он или неживой. Так же в этом примере показано, как добавить свою статистику, например, статистику по количеству горутин, либо по количеству обработанных запросов, либо о просто прошедшем запросе. Для этого объявлена новая мапа, специально для expvar, с именем hits. При каждом хите мы добавляем туда Path, который пришел. Автоматически, когда добавляем мапу через NewMap, она добавляется в экспорт в нашу функцию. Единственный нюанс — эта мапа по умолчанию не сбрасывается. Также можно зарегистрировать какой-то свой обработчик, который будет вызываться при каждом вызове expvar, и там будут уже отдаваться не статические данные, а выполняться какая-то функция. Например, этой функцией вы можете собрать тайминги по вашим запросам либо из глобального хранилища тайминга просто здесь настроить экспорт. В данном примере объявлена переменная mystat, которая будет вызывать функцию. В этой функции возвращается статичная мапа, в ней есть несколько статичных значений и количество активных горутин в данный момент. То есть можно делать чуть более сложный мониторинг, чем количество памяти. Стоит еще сказать, что подобным образом можно сбрасывать сбрасывать мапу, мапу со значениями, которую мы инкрементим. У нее есть функция Init, и, вызвав ее в подобном обработчике, можно сбросить все.

Буквально в 30 строчек вы можете организовать простой мониторинг вашего сервера. Однако дергать руками, если у вас, конечно, ваша система мониторинга не построена на том, что вы сами опрашиваете сервер, такой URL неудобно. Гораздо чаще у нас бывает настроен какой-то сервер статистики, куда все ваши серверы шлют метрики. Например, это может быть graphite. Давайте посмотрим, каким образом можно слать данные в graphite вообще без каких-либо внешних библиотек. Давайте рассмотрим этот код.

```
package main

import (
        "bytes"
        "flag"
        "fmt"
```

```
        "net"
        "net/http"
        "runtime"
        "time"

        _ "expvar"
)

func handler(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("hello world"))
}

var carbonAddr = flag.String("graphite", "192.168.99.100:2003",
                             "The address of carbon receiver")

func main() {
        flag.Parse()

        go sendStat()

        http.HandleFunc("/", handler)

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

func sendStat() {
        m := &runtime.MemStats{}
        conn, err := net.Dial("tcp", *carbonAddr)
        if err != nil {
                panic(err)
        }
        c := time.Tick(time.Minute)
        for tickTime := range c {
                runtime.ReadMemStats(m)

                buf := bytes.NewBuffer([]byte{})
                fmt.Fprintf(buf, "coursera.mem_heap %d %d\n",
                        m.HeapInuse, tickTime.Unix())
                fmt.Fprintf(buf, "coursera.mem_stack %d %d\n",
                        m.StackInuse, tickTime.Unix())
                fmt.Fprintf(buf, "coursera.goroutines_num %d %d\n",
                        runtime.NumGoroutine(), tickTime.Unix())

                conn.Write(buf.Bytes())
                fmt.Println(buf.String())
        }
}
```

У нас есть функция с простым обработчиком, который просто пишет значение "Hello, world есть адрес graphite который парсится из флагов. Основное, что нас в этом примере интересует — это функция sendStat. Внутри этой функции, запускаемой в отдельной горутине, будем получать статистику по памяти и по количеству использованных горутин. Для начала соединяемся с graphite по TCP — мы ведь хотим, чтобы данные доходили всегда. После этого каждую минуту будем слать данные в графит. Для этого объявляем новый тикер, и итерируемся по нему в цикле. Напомним, что тикер возвращает канал, в котором значения появляются через указанный интервал времени. Внутри этого цикла будем читать статистику по памяти — ту же самую статистику, которую выдает expvar. У графита очень простой текстовый протокол — туда можно написать значение метрики, имя метрики, значение метрики и timestamp, в который было проведено измерение. Поэтому мы просто объявляем новый буфер и записываем туда три значения: количество памяти в хипе, количество памяти в стеке и количество горутин. Количество памяти берем из MemStats — специальной функции, которая воззвращает количество используемой в данный момент

память. А количество горутин я получим из функции NumGoroutine пакета runtime. Далее запишем эти метрики в TCP-соединение.

Обратите внимание: мы не использовали никаких библиотек, для того, чтобы отправлять статистику во внешнюю систему. Запустите программу и посмотрите, как выглядит статистика в интерфейсе graphite.

Мы рассмотрели, как нехитрым образом можно организовать статистику в базовом варианте. Вы, конечно же, будете пользоваться какими-то другими инструментами, может быть, другими библиотеками. Но самым простым вариантом вы можете воспользоваться уже сейчас. Далее мы рассмотрим систему Prometheus: каким образом можно отправлять метрики при помощи нее.

Отправка таймингов во внешнюю систему

В этом разделе мы поговорим про систему мониторинга под названием Prometheus. Это довольно молодое решение, которое было разработано в компании SoundCloud для задачи мониторинга их огромного количества серверов. По сути, это целый фреймворк для организации мониторинга, а также многомерная база данных для хранения множества метрик с разными свойствами.

Давайте рассмотрим, как базово при помощи Prometheus собирать статистику с сервера. Перед вами очень небольшой пример.

```
package main

import (
        "fmt"
        "net/http"

        "github.com/prometheus/client_golang/prometheus/promhttp"
)

func main() {
        http.Handle("/metrics", promhttp.Handler())
        fmt.Println("starting server at :8082")
        http.ListenAndServe(":8082", nil)
}
```

Мы подключаем библиотеку для работы с Prometheus, после чего я регистрирую по url'у metrics какой-то обработчик. На этом наш код заканчивается! Внутри обработчик регистрирует функцию, которая при обращении к нужному url выводит статистику. Там будет информация про время работы коллектора, количество горутин, версию Go, количество локаций и так далее. То есть фактически та же самая информация, которую мы видели в expvar, с одним отличием. Дело в том, что Prometheus работает по принципу Pull. То есть он сам ходит по вашим серверам и сам собирает статистику. Сервера, по которым он будет ходить, можно указать у него в config-файле. У него есть даже маленький дашборд, в котором можно посмотреть на все собираемые метрики. Интерфейс этот довольно простой. Сам Prometheus даже не рекомендует им пользоваться, а для организации нормальных дашбордов, панелей с графиками, он рекомендует использовать расширение Grafana. Grafana — это очень популярная система для построения графиков. Как и Prometheus, она тоже написана на Go и позволяет собирать вам информацию с разных источников, с разных баз данных, и выводить их в виде красивых графиков. Prometheus так же позволяет сделать фильтрацию отображаемой статистики.

Следующий вопрос, который встает, как собирать метрики по работе самого моего приложения? Ведь сервер может работать, а приложение — не отвечать, потому что оно зависло, либо же после обновления стало слишком медленно работать. Рассмотрим следующий пример. В нем я будем регистрировать количество вызовов API и время, которое эти вызовы заняли.

```
package main

import (
        "fmt"
        "math/rand"
        "net/http"
        "time"
```

```
        "github.com/prometheus/client_golang/prometheus"
        "github.com/prometheus/client_golang/prometheus/promhttp"
)

var (
        timings = prometheus.NewSummaryVec(
                prometheus.SummaryOpts{
                        Name: "method_timing",
                        Help: "Per method timing",
                },
                []string{"method"},
        )
        counter = prometheus.NewCounterVec(
                prometheus.CounterOpts{
                        Name: "method_counter",
                        Help: "Per method counter",
                },
                []string{"method"},
        )
)

func init() {
        prometheus.MustRegister(timings)
        prometheus.MustRegister(counter)
}
```

Для этого объявим две переменных: timings и counter. Укажем имя каждой переменной, текстовое описание и в функции init зарегистрируем эти переменные в Prometheus, чтобы он их опрашивал, когда будет выгружать статистику по url. Далее идет уже бизнес-логика. Она представлена всего тремя строчками, в которых просто спим случайное значение времени.

```
func mainPage(w http.ResponseWriter, r *http.Request) {
        rnd := time.Duration(rand.Intn(50))
        time.Sleep(time.Millisecond * rnd)
        w.Write([]byte("hello world"))
}
```

Самим сбором статистики занимается middleware timeTrackingMiddleware.

```
func timeTrackingMiddleware(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
                start := time.Now()

                next.ServeHTTP(w, r)

                // r.URL.Path приходит от юзера! не делайте так в проде!
                timings.
                        WithLabelValues(r.URL.Path).
                        Observe(float64(time.Since(start).Seconds()))
                counter.
                        WithLabelValues(r.URL.Path).
                        Inc()
        })
}
```

В нем засекаем время, и после выполнения функции отправляем статистику. Отправляю статистику, обратите внимание, для конкретного пути, который ко мне пришел. Он приходит от пользователя, поэтому, если у вас нет строгих ограничений на него, не надо так делать. Регистрирую, сколько времени оно у меня заняло, и счетчик counter просто инкрементируем.

Посмотрите, как это выглядит в экспорте для самого Prometheus. Обратите внимание, тут для каждого метода указаны квантили, то есть сколько в среднем заняло, сколько, в какое время уложились

90 процентов пользователей, 99 процентов пользователей. Далее я можно на все это посмотреть в своем дашборде. Количество выводимой информации, квантили и прочее можно настраивать, как и в графите.

Итак, буквально за семьдесят строчек сделали уже довольно неплохую систему мониторинга. Мы можем отслеживать с её помощью количество запросов и время, которое занял каждый запрос. Если мы выкатим обновление и увидим, что какой-то из таймингов вырос, мы по крайней мере будем знать, где искать эту информацию. Prometheus может стать довольно хорошим решением для начала, а может быть, и не только для начала.

Низкоуровневое программирование

Пакет Unsafe

В этом разделе мы поговорим про те вещи, которыми, вам лучше никогда не пользоваться. Называется это unsafe. unsafe — это инструмент компилятора, который позволяет залезать в кишки памяти, достучаться до сырых данных. Если вы используете unsafe, сразу стоит учитывать, что вам не гарантируется обратная совместимость между версиями Go, Также unsafe запрещен в некоторых облачных платформах, например, в Google облаке.

Рассмотрим пример.

```
package main

import (
        "fmt"
        "unsafe"
)

func main() {
        a := int64(1)
        fmt.Println("memory pointer for var ‘a‘", unsafe.Pointer(&a))
        fmt.Println("memory size for var ‘a‘", unsafe.Sizeof(a))
...
```

У нас есть int64. Хотим посмотреть на указатель на этот адрес и размер. Выведется указатель и размер 8 байт. Причем указатель будет уже уже не uintptr, а прямо совсем сырой адрес. Что с этим можно сделать. Рассмотрим функцию Float64bits.

```
func Float64bits(f float64) uint64 {
        return *(*uint64)(unsafe.Pointer(&f))
}
```

Туда передадим float64, а вернется нам int. Для этого возьмем адрес &f, то есть uintptr. От него возьмем Pointer. А дальше мы интерпретируем это как память с uint64. И давайте посмотрим, как это выглядит.

```
...
f := 10.11
fmt.Printf("%d\n", Float64bits(f))
fmt.Printf("%#016x\n", Float64bits(f))
fmt.Printf("%b\n", Float64bits(f))
...
```

Передадим в функцию float, в который записано число 10.1. И выведем результат в десятеричной, шестнадцатиричной и двоичной системах. Попробуйте запустить и посмотрите на вывод. Таким способом можно посмотреть прямо в памяти, где мантисса, где экспонента у числа, интерпретировать какую-то область памяти как совершенно другое значение. Но лучше это никогда не делать.

Есть еще один пример. Рассмотрим структуру Message. Есть flag, есть строка, есть еще один flag. Именно в таком порядке. Создадим экземпляр структуры и залезем в кишки структуры, чтобы посмотреть, сколько в памяти занимает сама эта структура и сколько занимает какое из полей ее данных, какой Offset желателен, и какой Offset в памяти находится у этой структуры.

```
...
type Message struct {
```

```
        flag1 bool
        flag2 bool
        name  string
    }

    msg := Message{
        flag1: false,
        flag2: false,
        name:  "Neque porro quisquam est qui dolorem",
    }

    fmt.Println("memory size for Message struct", unsafe.Sizeof(msg))

    fmt.Println(
        "flag1 Sizeof", unsafe.Sizeof(msg.flag1),
        "Alignof", unsafe.Alignof(msg.flag1),
        "Offsetof", unsafe.Offsetof(msg.flag1),
    )

    fmt.Println(
        "flag2 Sizeof", unsafe.Sizeof(msg.flag2),
        "Alignof", unsafe.Alignof(msg.flag2),
        "Offsetof", unsafe.Offsetof(msg.flag2),
    )

    fmt.Println(
        "name Sizeof", unsafe.Sizeof(msg.name),
        "Alignof", unsafe.Alignof(msg.name),
        "Offsetof", unsafe.Offsetof(msg.name),
    )

}
```

Если вы запустите, вы увидите, что размер этой структуры 32 байта в памяти. Почему 32 байта, откуда они взялись? flag1 — у него смещение 0 в памяти относительно этих 32 байт, дальше идет name — это строчка, она занимает 16 байт, потому что внутри строчки есть ссылка на данные и длина. При этом компилятор нам советует, что выравнивание должно быть 8, но находится оно по Offset 8. Что такое выравнивание и зачем оно нужно? Дело в том, что оперирует компилятор машинными словами. В 64-битной операционной системе размер машинного слова 8. Поэтому и булевая переменная, несмотря на то, что Sizeof у нее 1, занимает 8 байт. То есть если переделать структуру, то удастся вместить в эти 8 байт еще какие-то значения. Если оменять местами флаги в Message, то структура станет занимать 24 байта, потому что мы эффективнее используем память. То есть раньше у flag1 занимал по сути 8 байт, и 7 было неиспользуемых. И потом был еще один такой flag2. А теперь в первых 8 байтах уместили flag1 и flag2, и потом тоько строковое поле name. Если у вас какие-то очень-очень большие структуры, то вот такой перекомпоновкой можно выиграть немножко места. Есть специальные инструменты, которые позволяют анализировать структуру и предлагают поменять поля местами, чтобы было эффективнее использование памяти. На самом деле постоянно заморачиваться по этому поводу не стоит. Просто это та вещь, которую полезно знать для общего развития.

Рассмотрим еще один пример. Как вы помните, мы можем конвертировать слайс байт в строку и обратно. При этом, если мы хотим их потом дальше использовать, то создается копия данных. А что делать в случае, если мы не хотим создавать копию данных? Мы хотим как-то поэкономить и оптимизировать нашу программу. В этом случае мы можем воспользоваться пакетом unsafe и, используя указатель, unsafe pointer, для того чтобы создать строку, которая будет ссылаться на те же самые данные, на которые ссылается слайс байт.

```
package main

import (
    "fmt"
    "reflect"
    "unsafe"
```

```
)

// bytesToStr сорздаёт строку, указывающую на слайс байт, чтобы избежать копирования.
//
// Warning: the string returned by the function should be used with care,
//as the whole input data chunk may be either blocked from being freed by
//GC because of a single string or the buffer.Data may be garbage-collected
//even when the string exists.
func bytesToStr(data []byte) string {
        h := (*reflect.SliceHeader)(unsafe.Pointer(&data))
        fmt.Printf("type: %T, value: %+v\n", h, h)
        fmt.Printf("type: %T, value: %+v\n", h.Data, h.Data)
        shdr := reflect.StringHeader{Data: h.Data, Len: h.Len}
        fmt.Printf("type: %T, value: %+v\n", shdr, shdr)
        return *(*string)(unsafe.Pointer(&shdr))
}

func main() {

        data := []byte(‘some test‘)
        str := bytesToStr(data)
        // str := string(data)
        fmt.Printf("type: %T, value: %+v\n", str, str)
}
```

Что мы для этого делаем? В нашу функцию-конвертер приходит слайс байт, на данные получаем Pointer. Слайс на самом деле тоже структура. У нее есть, длина, capacity и ссылка на данные. Теперь скажем, что надо воспринимать то, что получили с этой структурой, как указатель на заголовок слайса, то есть как саму ту базовую структуру, которая в слайсе лежит. Теперь можно использовать эти данные, потому что вернулся uintptr, то есть указатель на данные у нас уже есть. Составим из этого структуру, которая называется StringHeader, которая состоит только из указателей на данные и длины. И потом вернем уже эти данные, воспринимая их как строку. Обратимся снова к коду. Мы получили заголовок слайса, повыводили данные. Затем создаем новую структуру, указываем у нее ссылку на данные туда же, куда ввел слайс байт, и длину указываем ту же самую. Затем получаем адрес свежесозданной структуры через unsafe.Pointer и воспринимаем данные по этому адресу как строку, которую и возвращаем. Соответственно, вы можете попробовать сделать и обратную операцию, но это чревато проблемами, если вы где-то ошибетесь, потому что эти данные могут либо не собраться garbage collector’ом. Либо, наоборот, они могут собраться garbage collector’ом, когда они еще вами реально используются.

Использование unsafe имеет свою цену. В сырые данные лучше не лазть без особой необходимости. То есть вам стоит лезть в unsafe тогда, когда вы точно знаете, что вы делаете и зачем вам это надо. Если вы думаете, что вот так нужно делать всегда, просто потому что это быстрее, это не так, и в unsafe лезть не надо. Есть те вещи, которые пробовать не стоит! Однако unsafe все-таки используется, например, в cgo, про который мы поговорим дальше.

cgo - интеграция кода на си

В этом разделе мы рассмотрим возможность интеграции кода на C с кодом с Go, то есть вызовов C кода из Go. Этот механизм называется cgo, и далее мы рассмотрим, как это происходит технически, и все подводные камни, которые вам встретятся.

Итак, попробуем вызвать простую функцию из Go.

```
/*
// Всё что в комментари над import "C" является кодом на C
//он будет скомпилирован при помощи GCC.
// У вас должен быть установлен GCC

int Multiply(int a, int b) {
    return a * b;
}
*/
```

```
import "C" //это псевдо-пакет, он реализуется компилятором
import "fmt"

func main() {
        a := 2
        b := 3
        // для того чтобы вызвать СИшный крод надо добавить префикс "C."
        // там же туда надо передать СИшные переменные
        res := C.Multiply(C.int(a), C.int(b))
        fmt.Printf("Multiply in C: %d * %d = %d\n", a, b, int(res))
        fmt.Printf("c-var internals %T = %+v\n", res, res)
}
```

Реализуется это при помощи псевдо-пакета C, который надо указать в import. Псевдо-пакет — это не исходник на Go, который можно прочитать, пакет реализуется компилятором. После подключения псевдо-пакета, все, что будет написано над импортом в комментарии, будет восприниматься компилятором, как код на C, В примере приведена сишная функция multiply, которая принимает два значения и возвращает их произведение. Как её вызвать из основной программы? Чтобы вызвать любой сишный код, нужно добавить префикс C, то есть обращение к псевдо-пакету. В main'е хотим посчитать произведение переменных a и b с помощью функции multiply. Кроме вызова функции через префикс есть еще один важный нюанс — нельзя передать гошные переменные в функцию на C, потому она про них ничего не знает, потому что выполняется в другом рантайме. Поэтому гошные переменные нужно конвертировать в сишные переменные. Для простых переменных это сделать просто. Функция multiply тоже вернет мне сишную переменную res. Нужно помнить, что при желании использовать её, как гошную переменную, надо будет провести обратное преообразование.

Следующий подводный камень заключается в том, что сишный код собирается при помощи GCC компилятора. Компилятор Visual Studio и другие компиляторы тоже не поддерживаются. Поэтому, чтобы использовать cgo нужно установить GCC, а раз вы устанавливаете C, то вы теряете возможность кросс-компиляции. Вам нужно будет тогда использовать GCC для целевой платформы, если вы решите собирать отличный от того, что он стал в базовом GCC. Кроме того, если вы будете использовать какие-то внешние библиотеки, которые подключаются вам каким-либо модулем, то есть DLL, или сошкой, вы теряете всю прелесть статической компиляции, вы возвращаете ситуацию возможного dependency hell, когда вам нужно будет доустанавливать библиотеки, и они будут конфликтовать с чем-то другим. Ещё один недостаток заключается в том, что через GCC код будет пересобираться каждый раз, потому что он не знает, что изменилось. Это тоже будет замедлять кросс-компиляцию.

Теперь рассмотрим, каким образом можно кода на C, вызвать код на Go, то есть обратную ситуацию.

```
package main

/*
void Multiply(int a, int b);
*/
import "C" //это псевдо-пакет, он реализуется компилятором
import "fmt"

//export printResultGolang
func printResultGolang(result C.int) {
        fmt.Printf("result-var internals %T = %+v\n", result, result)
}

func main() {
        a := 2
        b := 3
        C.Multiply(C.int(a), C.int(b))
}
```

Вы можете видеть пример, в котором запускается гошная программа, из неё вызывается сишный код, из сишного кода мы теперь хотим вызвать гошный код. Мы хотим, чтобы функция multiply печатала результат при помощи гошной функции printResultGolang. Обратите внимание, что функция printResultGolang принимает сишный аргумент. И в main она объявлена, как принимающая сишный аргумент. Для того

чтобы её — гошную функцию — можно было вызвать в си, ее нужно объявить экспортируемой, чтобы компилятор все нужные биндинги подготовил. Обратим внимание на то, что, в отличие от предыдущего примере, функция на си реализована в отдельном файле с расширение .c, в основной программе указано только объявление функции. Так сделано потому, что компилятор готовит биндинги и прочее в заголовочном файле, если объявить реализацию, как раньше, компилятор может заругаться на двойное объявление. Теперь посмотрим на саму реализацию сишной функции.

```
#include "_cgo_export.h"

extern void printResultGolang(int i);

void Multiply(int a, int b) {
    printResultGolang(a*b);
}
```

Во-первых надо подсключить cgo, во-вторых внешняя функция printResultGolang будет объявляться как extern void.

Снова вернемся к недостаткам. При сборке больше не удастся воспользоваться любимым go run, придётся собирать все полностью. Кроме тогопри работе нашей программы несколько раз происходит переключение между рантаймами. Когда вы вызываете функцию main, у вас гошный runtime, потом вы вызываете функцию multiply, вы переключетесь в сишку, когда вы вызываете printResult, вы опять вызываете гошку, затем, чтобы завершить функцию multiply на си, вы преклчаетесь опять в сишный runtime, и только после этого вы вернетесь в гошный runtime. Переключение между рантаймами довольно дорогая операция. Например, если запустить бенчмарки для двух пустых функций — одной чисто на го, другой на си, вызываемой через го —

```
// go test -bench . -gcflags '-l'    # отключаем инлайнинг
package main

import (
        "testing"
)

func BenchmarkCGO(b *testing.B) {
        CallCgo(b.N) // call `C.(void f() {})` b.N times
}

func BenchmarkGo(b *testing.B) {
        CallGo(b.N) // call `func() {}` b.N times
}
```

Окажется, что чисто гошная работает в несколько раз быстрее. Дело в том, что есть "вселенная Go где есть garbage collector, треды, горутины, синхронщина, перекладывание горутин из одного системного треда в другой, а в C этого ничего нет, он про это ничего не знает. Поэтому, когда вы вызываете код через cgo, происходит lock треда на этот вызов — вы лочите полностью системный тред, для того чтобы вызвать в нем эксклюзивно C-код. В случае если бы это был чистый Go-код, и там была бы блокирующая операция, во время этого в том же треде могли бы выполняться какие-то другие горутины. В случае с cgo такого не будет.

Посколько во "вселенной C"нет никакого garbage collector, за памятью придется следить вручную Рассмотрим следующий код.

```
package main

// #include <stdio.h>
// #include <stdlib.h>
import "C"
import "unsafe"

func print(s string) {
        cs := C.CString(s) // переход в другую вселенную
        defer C.free(unsafe.Pointer(cs))
```

```
        println(cs)
        // СИ-шные не собираются через ГО-шный сборщик мусора, их надо освобождать руками
        // закомментируйте строку с defer и запустите программу - начнётся утечка памяти
}

func main() {
        for {
                print("Hello World")
        }
}
```

В примере есть функция print, которая принимает Go-строку, конвертирует её в C-строку и выводит на экран. Однако обратим внимание, что в этой функции есть еще ручное освобождение памяти. Поскольку мы получаем сишную строку, то больше не можем расссчитывать на сборщик мусора, и, если уберем ручное освобождение, получим утечку памяти. В данном примере утечку памяти более-менее можно было бы найти через, например, профилировщик pprof. Но в общем случае найти утечку было бы довольно сложно — это еще один подводный камень. Как только вы уходите в другую "вселенную pprof перестает видеть, что происходит дальше. То есть Go-функцию можно отследить, но после входа в cgo pprof не сможет определить место утечки точнее, чем "вызов cgo". И если вы захотите профилировать код, вам придется использовать valgrind, либо какие-то другие инструменты, для того чтобы найти эти утечки.

Еще один момент, который стоит учитывать, это то, что в C не будет никакой синхронщины, и тред будет лочиться эксклюзивно на ваш вызов. Рассмотрим следующий код.

```
package main

import (
        "time"
)

func main() {
        for i := 0; i < 100; i++ {
                go func() {
                        // запускаем ГОшный sleep
                        time.Sleep(time.Minute * 10)
                }()
        }
        time.Sleep(time.Minute * 11)
}
```

Мы запускаем 100 горутин, внутри которых просто sleep. Запустите и посмотрите, сколько системных тредов понадобится? Скорее всего порядко десятка. То есть тут не блокирующие операции и горутины выполняются синхронно. Что будет, если я попробовать вызвать C-sleep?

```
package main

//#include <unistd.h>
import "C"
import (
        "time"
)

func main() {
        for i := 0; i < 100; i++ {
                go func() {
                        // запускаем СИшный sleep
                        C.sleep(60 * 10) // 10 минут
                }()
        }
        time.Sleep(11 * time.Minute)
}
```

Снова посмотрите, сколько тредов появится. Их будет 100. То есть каждая из 100 горутин запускается в отдельном системном треде, поскольку вызов C-кода лочит тред на себя. Получается оверхед. В случае,

если вы будете ходить по сети с какими-то сишными блокирующими вызовами, то вы как раз будете получать ту ситуацию, что ваш запрос висит, ничего не делает, ожидает ответа от внешней системы, хотя там могла бы выполняться какая-то горутина.

Еще раз пробежимся по всем подводным камням. Вызов cgo небесплатен — переключение между рантаймами довольно дорогое. Вызов C-кода из Go может повлечь за собой неприятные последствия. Например, у вас начнет течь память, либо начнет сильно расти количество системных тредов. Также вы теряете возможность кросс-компиляции.

Наконец ответим на самый главный вопрос, когда вообще стоит использовать cgo? Во-первых, у вас может не быть выбора. Например, у вас есть какая-то уже скомпилированная библиотека, для которой у вас есть только заголовочные файлики. Поэтому вам придется это подрубать и использовать. Надейтесь только, что там нет хождения по сети. Во-вторых,cgo можно использовать, если у вас какая-то библиотека, которую вы будете вызывать, например, один раз, то есть оверхед на вызов будет маленьким, но при этом внутри она очень оптимизирована, как, например, TensorFlow. В Go есть биндинги для TensorFlow, которые позволяют использовать его для расчетов модели. Еще один пример использования cgo — это библиотека SQLite. Биндинги к SQLite реализованы сейчас через cgo.

Когда cgo использовать не стоит? Когда у вас будет очень много маленьких вызовов. То есть операция небольшая и вы будете много терять на оверхеде на вызове. Не стоит использовать, если вы ходите по сети. Кроме cgo есть еще трансплиттеры C-кода в Go, однако они еще неполноценно рабочие.

Инструменты для статического анализа

go vet, gometalinter

В этой главе мы рассмотрим, какие еще инструменты Go предоставляет для того, чтобы код был красивым и поддерживаемым. Начнем с инструмента, который называется go vet. Если вы пользуетесь Visual Studio Code, то, наверно, заметили, что в нем много инструментов есть из коробки, и вы могли видеть в своем коде довольно много подчеркиваний зеленых, красных. Каким образом они получаются? Какой инструмент генерирует такого рода подсказки? Рассмотрим код.

```
package main

import (
        "fmt"
)

func test_error(is_ok bool) error {
        if !is_ok {
                fmt.Errorf("failed")
        }
        return nil
}

func main() {
        flag := true
        result := test_error(flag)
        fmt.Printf("result is\n", result)
        fmt.Printf("%v is %v\n", flag)
}
```

Этот код компилируется. Однако, давайте запустим встроенный инструмент, который называется go vet, и посмотрим, на какие ошибки он ругается. Мы получим следующий вывод.

```
vet_example.go:9: result of fmt.Errorf call not used
vet_example.go:17: no formatting directive in Printf call
vet_example.go:18: missing argument for Printf("%v"): format reads arg 2,
  have only 1 args
exit status 1
```

Оказывается, в совсем небольшой программе, которая к тому же компилируется, целых три ошибки. Первая — на 9-й строчке результат вызова функции не используется. Чтобы исправить ошибку, напишем

return. Вторая ошибка — не указана опция форматирования в Printf — это все-таки форматированный ввод, переменные мы туда передаем, а форматирование указать забыли. Третья ошибка — в другом Printf пропущен аргумент. Их ожидается два, а передается всего лишь один, нужно добавить второй. Если запустить go vet еще раз, ошибок не возникает.

go vet — это встроенный в Go инструмент статического анализа. Но он не копает очень глубоко. Что делать, если хочется больше ошибок? Рассмотрим другой код.

```
package main

import (
        "fmt"
)

type MyStruct struct {
        user_id  int
        DataJson []byte
}

func Test_error(is_ok bool) error {
        if !is_ok {
                fmt.Errorf("failed")
        }
        return nil
}

func Test() {
        flag := true
        result := Test_error(flag)
        fmt.Printf("result is\n", result)
        fmt.Printf("%v is %v", flag)
}
```

Запустим на нем второй инструмент — gometalinter. И получим целый экран ошибок! Убедитесь в этом сами. Gometalinter — это инструмент, который объединяет в себе довольно большое количество других статических анализаторов. Среди ошибок, которые вы получите, будут ошибки, вызванные тем, что в данном коде не выполнено требование к экспортируемым структурам, функциям, переменным или константам писать комментарий: зачем эта структура нужна, что она делает, что эта функция возвращает. Это требование достаточно важно, потому что документация в Go строится по этим комментариям. Следующий кластер ошибок будет связан с использованием underscore в именах переменных Go. В Go принят стиль CamelCase. Инструмент, проверяющий код на наличие ошибок этого типа даже подсказывает, как лучше изменить имя переменной. Кроме того он следит за тем, чтобы всякие общего рода сокращения были записаны в верхнем регистре. Например, Json был в верхнем регистре. Еще есть проверка на то, обрабатываете ли вы в своем коде ошибки. Также находятся ошибки, которые нашлись бы с помощью go vet. Попробуйте сами исправить все ошибки.

Если прикрутить gometalinter в ваш инструмент для непрерывной интеграции и не разрешать коммитить даже с кодом, который не проходит те проверки, которые вы выбрали, то поначалу будет тяжело, но через какое-то время ваш код будет полностью документирован, вы будете автоматически находить мелкие ошибки и поддерживаемость очень увеличится. Очень полезно начинать разработку в Go с того, чтобы вкрутить этот инструмент и всегда соблюдать основные требования. По меньшей мере писать комментарий, обрабатывать все ошибки и не использовать underscore считается очень хорошей практикой.