Введение в Golang. Лекция 3

JSON

Распаковываем JSON

Эту главу посвятим разговору про json. JSON — это де факто стандарт передачи данных между client-side и server-side в современном вебе. В go имеется встроенный упаковщик и распаковщик json, и процесс упаковки-распаковки вообще каких-либо данных в go называется маршалингом и анмаршалингом.

Давайте посмотрим, как это работает.

```
package main

import (
        "encoding/json"
        "fmt"
)

type User struct {
        ID       int
        Username string
        phone    string
}

var jsonStr = `{"id": 42, "username": "rvasily", "phone": "123"}`

func main() {
        data := []byte(jsonStr)

        u := &User{}
        json.Unmarshal(data, u)
        fmt.Printf("struct:\n\t%#v\n\n", u)

        u.phone = "987654321"
        result, err := json.Marshal(u)
        if err != nil {
                panic(err)
        }
        fmt.Printf("json string:\n\t%s\n", string(result))
}
```

Мы для начала импортируем пакет для работы с json. Опишем структуру пользователь: Id, Username, телефон, в которую будем распаковывать. Теперь у нас есть запакованный json, в котором есть соответствующие поля, и мы хотим его распаковать. Для этого мы прежде всего преобразуем строку с json к

слайсу byte, Дело в том, что распаковщик работает со слайсом byte, не со строками. Дальше создаем пустого юзера и вызывем функцию json.Unmarshal, которая принимает слайс byte, полученный из json-строки, и записывает результат в пустого юзера. Затем пробуем изменить телефон у полученной структуры и обратно запаковать ее при помощи функции Marshal, которая возвращает результат и ошибку. Запустив, обнаруживаем, что распаковался json в структуру

```
User{ID:42, Username:"rvasily", phone:""}
```

с почему-то пустым телефоном. После запаковывания получилась строка

```
{"ID":42,"Username":"rvasily"},
```

в которой вообще нет поля «телефон». Почему так происходит? Посмотрим еще раз на нашу структуру. Поле «телефон» — приватное, так как начинается с маленькой буквы. То есть доступ к этому полю могут иметь только те методы, которые объявлены в том же самом пакете, в котором и эта структура. То есть в пакете main. Соответственно, пакет encoding/json не может иметь к ним доступа не может туда ни записать данные, ни прочитать оттуда данные при запаковке. Имейте это в виду.

В предыдущем примере мы упаковывали и запаковывали поля с каким-то определенным именем. А что, если вдруг надо сменить это имя, при этом нельзя менять имя структуры? Для этих целей в go у структур может быть метаинформация — теги структуры. Объявляются они в бэктиках после определения типа поля структуры.

```
type User struct {
        ID       int `json:"user_id,string"`
        Username string
        Address  string `json:",omitempty"`
        Comnpany string `json:"-"`
}
```

И строятся одинаковым образом. Сначала префикс json:, показывающий, что тэг относится к json-объекту. Дальше записывается имя поля в json, по которому надо распаковывать, в которое запаковывать поле . И дальше после запятой — тип данных, в который он должен записать этот тип. Имя можно не указывать, а сразу поставить запятую, это будет подразумевать, что в структуре и в json одинаковое имя. Кроме привычных типов может быть еще использовано, как в поле Address, omitempty, это значит, что если поле пустое, там, например, значение по умолчанию, то его не нужно писать в результат при запаковке. Минус, он же дефис, означает, что поле вообще не нужно сериализовывать и десериализовывать при json.

Итак, когда мы познакомились с тэгами структур, попробуем что-то записать.

```
func main() {
        u := &User{
                ID:       42,
                Username: "rvasily",
                Address:  "",
                Comnpany: "Mail.Ru Group",
        }
        result, _ := json.Marshal(u)
        fmt.Printf("json string: %s\n", string(result))
}
```

Создадим юзера и запакуем. Получится json string:

```
{"user_id":"42","Username":"rvasily"}
```

Почему получился именно такой результат? В структуре есть поле ID, оно является типом int, однако мы говорим, что его нужно сериализовать, используя ключ "user_id как строку. Username сериализуем, как есть. Адрес не сериализуем, если он пустой, как в нашем случае. Если написать в адресе что-то, то он должен сериализоваться. Компанию вообще не сериализуем никогда.

С тегами структуры вы будете сталкиваться очень часто. Не только json имеет свои теги, но и очень много других пакетов.

Нюансы работы с JSON

В предыдущем разделе мы рассматривали упаковку и запаковку json'а с заранее известной структурой. Но что делать, если мы хотим поработать с json'ом, у которого не определена структура, либо мы не знаем, либо она каждый раз меняется, а залезть нам в неё всё-таки надо. Для этого в Go так же можно работать, используя стандартный пакет json, но уже с помощью пустого интерфейса.

```
var jsonStr = `[
        {"id": 17, "username": "iivan", "phone": 0},
        {"id": "17", "address": "none", "company": "Mail.ru"}
]`

func main() {
        data := []byte(jsonStr)

        var user1 interface{}
        json.Unmarshal(data, &user1)
        fmt.Printf("unpacked in empty interface:\n%#v\n\n", user1)

        user2 := map[string]interface{}{
                "id":       42,
                "username": "rvasily",
        }
        var user2i interface{} = user2
        result, _ := json.Marshal(user2i)
        fmt.Printf("json string from map:\n %s\n", string(result))
}
```

Вот в коде примере определён json — массив из двух разных объектов. Распаковывать его примерно так же, как в структуру. В данном случае создаем переменную user1, которая имеет тип «пустой интерфейс», и передаем ссылку на эту переменную, чтобы в неё произошла распаковка. Если запустить программу, то увидим, что наш json будет распакован в слайс пустых интерфейсов. Каждый объект будет типа map[string]interface, Чтобы добраться до данных в удобном виде после такой распаковки, конечно придётся довольно много делать преобразований интерфейсов, но другой возможности нет.

Точно так же можно запаковать любое значение через пустой интерфейс, да и просто без интерфейса. В примере сериализуется user2 типа map[string]interface. Проверьте, что получается валидный json, который уже может принять любая другая программа.

Работа с динамическими данными

Пакет reflect - работаем с динамикой в рантайме

В разделе, посвященном упаковке-распаковке формата JSON, мы видели пример, как работать с полями структуры во время выполнения программы. При этом в Go нет возможности пройтись по полям структуры просто в цикле, используя оператор for. Можно ли каким-то образом реализовать что-то похожее? На самом деле есть два подхода к работе с динамическими значениями. Первый подход — это рефлексия, и как раз его мы и рассмотрим прямо сейчас.

Рефлексия позволяет оперировать со структурами и другими объектами и типами программы в runtime. Давайте посмотрим на пример.

```
package main

import (
        "fmt"
        "reflect"
)

type User struct {
        ID       int
        RealName string `unpack:"-"`
```

```
        Login    string
        Flags    int
}

func PrintReflect(u interface{}) error {
        val := reflect.ValueOf(u).Elem()

        fmt.Printf("%T have %d fields:\n", u, val.NumField())
        for i := 0; i < val.NumField(); i++ {
                valueField := val.Field(i)
                typeField := val.Type().Field(i)

                fmt.Printf("\tname=%v, type=%v, value=%v, tag=`%v`\n", typeField.Name,
                        typeField.Type.Kind(),
                        valueField,
                        typeField.Tag,
                )
        }
        return nil
}

func main() {
        u := &User{
                ID:       42,
                RealName: "rvasily",
                Flags:    32,
        }
        err := PrintReflect(u)
        if err != nil {
                panic(err)
        }
}
```

В программе объявлена структура user, для которой вызывается функция PrintReflect, которая проходится по всем полям структуры и выводит имя поля, тип, значение и теги, если они есть.

Давайте подробнее рассмотрим, как она реализована. Для начала мы получаем при помощи пакета рефлексии структуру из пустого интерфейса, После этого можно, например, получить количество полей, которые есть у этой структуры, и вывести их на экран. Дальше можно в цикле обходить эти поля, получать значения поля, тип, тег и выводить их всех на экран.

При помощи рефлексии можно делать и гораздо больше. В следующем примере рассмотрим более практическое применение с присваиванием в поля структуры каких-то значений.

```
func main() {
        /*
                perl -E '$b = pack("L L/a* L", 1_123_456, "v.romanov", 16);
                        print map { ord.", "  } split("", $b); '
        */
        data := []byte{
                128, 36, 17, 0,

                9, 0, 0, 0,
                118, 46, 114, 111, 109, 97, 110, 111, 118,

                16, 0, 0, 0,
        }
        u := new(User)
        err := UnpackReflect(u, data)
        if err != nil {
                panic(err)
        }
```

```
        fmt.Printf("%#v", u)
}
```

Пусть есть какой-то формат упаковки данных. JSON мы брать не будем, он довольно нетривиален в парсинге. Мы будем парсить бинарные данные, упакованные при помощи функции pack. Эта функция есть во многих языках, она копирует то значение, которое есть, прямо в бинарные данные. Вызываем функцию pack на perl, пакуем целое число, строчку (строка пакуется в виде длины и значения), еще целое число. Полученное байтовое представление будем использвать в нашей программе. Для того, чтобы продемонстрировать, как распаковать бинарные данные в структуру. Так выглядит основная функция UnpackReflect.

```
type User struct {
        ID       int
        RealName string `unpack:"-"`
        Login    string
        Flags    int
}

func UnpackReflect(u interface{}, data []byte) error {
        r := bytes.NewReader(data)

        val := reflect.ValueOf(u).Elem()

        for i := 0; i < val.NumField(); i++ {
                valueField := val.Field(i)
                typeField := val.Type().Field(i)

                if typeField.Tag.Get("unpack") == "-" {
                        continue
                }

                switch typeField.Type.Kind() {
                case reflect.Int:
                        var value uint32
                        binary.Read(r, binary.LittleEndian, &value)
                        valueField.Set(reflect.ValueOf(int(value)))
                case reflect.String:
                        var lenRaw uint32
                        binary.Read(r, binary.LittleEndian, &lenRaw)

                        dataRaw := make([]byte, lenRaw)
                        binary.Read(r, binary.LittleEndian, &dataRaw)

                        valueField.SetString(string(dataRaw))
                default:
                        return fmt.Errorf("bad type: %v for field %v", typeField.Type.Kind(), t
                }
        }

        return nil
}
```

Что она делает и как она это делает? Мы получаем ридер, чтобы читать из входных бинарных данных data. Дальше получаем внутреннюю структуру из рефлексии и идем по полям. Получаю значение, тип, проверяем, если есть тег unpack, то не используем поле для распаковки, иначе делаем switch по типу поля — либо int, либо строка. Внутри каждой ветки создаем внутреннюю переменную, считываем в нее нужно количество байт ридером, не забывая, что для строки нужно сначала считать длину, а потом только строку нужной длины. И после этого устанавливаю значение в поле структуры при помощи ValueField.Set. В дефолтной ветке, то есть, если встретили какой-то неизвестный тип, ничего не делаем и ругаемся. Проверьте, что все заработает.

Какой минус есть у этого подхода? Дело в том, что вы, получается, должны каждый раз в runtime, во время выполнения работы программы, создавать еще какие-то дополнительные объекты, объекты пакета рефлексии, и проходиться по ним. То есть проделывать дополнительные вычисления во время работы программы. Есть второй способ для работы с динамическими значениями. Его мы рассмотри в следующей части.

Кодогенерация - программа пишет программу

Второй способ работы с динамическими данными — кодогенерация. В отличие от рефлексии кодогенерация не выполняется в процессе работы программы. Кодогенерацией занимается вообще отдельная программа, которая, как следует из названия, генерирует код для нужных действий, который потом выполняется уже в процессе работы. Как это работает?

```
// go build gen/* && ./codegen.exe pack/packer.go  pack/marshaller.go
package main

import "fmt"

// lets generate code for this struct
// cgen: binpack
type User struct {
        ID       int
        RealName string `cgen:"-"`
        Login    string
        Flags    int
}

type Avatar struct {
        ID  int
        Url string
}

var test = 42

func main() {
        /*
                perl -E '$b = pack("L L/a* L", 1_123_456, "v.romanov", 16);
                        print map { ord.", "  } split("", $b); '
        */
        data := []byte{
                128, 36, 17, 0,

                9, 0, 0, 0,
                118, 46, 114, 111, 109, 97, 110, 111, 118,

                16, 0, 0, 0,
        }

        u := User{}
        u.Unpack(data)
        fmt.Printf("Unpacked user %#v", u)
}
```

Итак, перед нами уже знакомый вам код, который распаковывает бинарные данные. Если в прошлый раз мы вызывали функцию, куда передавали структуру, то сейчас мы вызываем метод этой структуры. Но, если мы запустим программу, компилятор будет ругаться: метода нет. У нас есть только определение самой структуры. Где этот должен быть метод?

Метод сгенерирует вот эта вот программа!

```
// go build gen/* && ./codegen.exe pack/unpack.go  pack/marshaller.go
// go run pack/*
```

```
package main

import (
        "fmt"
        "go/ast"
        "go/parser"
        "go/token"
        "log"
        "os"
        "reflect"
        "strings"
        "text/template"
)

type tpl struct {
        FieldName string
}

var (
        intTpl = template.Must(template.New("intTpl").Parse(`
        // {{.FieldName}}
        var {{.FieldName}}Raw uint32
        binary.Read(r, binary.LittleEndian, &{{.FieldName}}Raw)
        in.{{.FieldName}} = int({{.FieldName}}Raw)
`))

        strTpl = template.Must(template.New("strTpl").Parse(`
        // {{.FieldName}}
        var {{.FieldName}}LenRaw uint32
        binary.Read(r, binary.LittleEndian, &{{.FieldName}}LenRaw)
        {{.FieldName}}Raw := make([]byte, {{.FieldName}}LenRaw)
        binary.Read(r, binary.LittleEndian, &{{.FieldName}}Raw)
        in.{{.FieldName}} = string({{.FieldName}}Raw)
`))
)

func main() {
        fset := token.NewFileSet()
        node, err := parser.ParseFile(fset, os.Args[1], nil, parser.ParseComments)
        if err != nil {
                log.Fatal(err)
        }

        out, _ := os.Create(os.Args[2])

        fmt.Fprintln(out, `package `+node.Name.Name)
        fmt.Fprintln(out) // empty line
        fmt.Fprintln(out, `import "encoding/binary"`)
        fmt.Fprintln(out, `import "bytes"`)
        fmt.Fprintln(out) // empty line

        for _, f := range node.Decls {
                g, ok := f.(*ast.GenDecl)
                if !ok {
                        fmt.Printf("SKIP %#T is not *ast.GenDecl\n", f)
                        continue
                }
        SPECS_LOOP:
                for _, spec := range g.Specs {
                        currType, ok := spec.(*ast.TypeSpec)
                        if !ok {
```

```
                fmt.Printf("SKIP %#T is not ast.TypeSpec\n", spec)
                continue
        }

        currStruct, ok := currType.Type.(*ast.StructType)
        if !ok {
                fmt.Printf("SKIP %#T is not ast.StructType\n", currStruct)
                continue
        }

        if g.Doc == nil {
                fmt.Printf("SKIP struct %#v doesnt have comments\n",
                                                        currType.Name.Name)
                continue
        }

        needCodegen := false
        for _, comment := range g.Doc.List {
                needCodegen = needCodegen || strings.HasPrefix(comment.Text,
                                                        "// cgen: binpack")
        }
        if !needCodegen {
                fmt.Printf("SKIP struct %#v doesnt have cgen mark\n",
                                                        currType.Name.Name)
                continue SPECS_LOOP
        }

        fmt.Printf("process struct %s\n", currType.Name.Name)
        fmt.Printf("\tgenerating Unpack method\n")

        fmt.Fprintln(out, "func (in *"+currType.Name.Name+
                                        ") Unpack(data []byte) error {")
        fmt.Fprintln(out, "        r := bytes.NewReader(data)")

FIELDS_LOOP:
        for _, field := range currStruct.Fields.List {

                if field.Tag != nil {
                        tag := reflect.StructTag(
                                field.Tag.Value[1 : len(field.Tag.Value)-1])
                        if tag.Get("cgen") == "-" {
                                continue FIELDS_LOOP
                        }
                }

                fieldName := field.Names[0].Name
                fileType := field.Type.(*ast.Ident).Name

                fmt.Printf("\tgenerating code for field %s.%s\n",
                                                currType.Name.Name, fieldName)

                switch fileType {
                case "int":
                        intTpl.Execute(out, tpl{fieldName})
                case "string":
                        strTpl.Execute(out, tpl{fieldName})
                default:
                        log.Fatalln("unsupported", fileType)
                }
        }
```

```
                        fmt.Fprintln(out, "        return nil")
                        fmt.Fprintln(out, "}") // end of Unpack func
                        fmt.Fprintln(out)      // empty line

                }
        }
}

// go build gen/* && ./codegen.exe pack/unpack.go  pack/marshaller.go
// go run pack/*
```

В процессе работы программы разбирает исходник и пишет код метода Unpack в отдельный файл.

Что происходит внутри у кодогенератора и почему это возможно? Дело в том, что программа на Go собирается программой, которая написана на Go, то есть компилятор Go написан теперь уже на Go. Это значит, что нам доступны все средства компилятора, все средства обхода абстрактного синтаксического дерева и все нужные библиотеки для работы с ним. Кодегенерация — это процесс написания программы, которая генерирует другую программу.

Сразу понять, как все работает довольно сложно, поэтому мы обсудим устройство кодогенератора довольно поверхностно, чтобы в вашей картине мира появится новый подход. Итак, сначала задаем несколько шаблонов, при помощи которых генерируется код, передаем в них имя, тип и другую нужную информацию, чтобы программа создала какой-то текст на выходе. В main при помощи пакетов ast, parser и token попробуем распарсить исходный код, который передаем первым аргументов в программу. Сгенерированный код будем записывать в файл, переданный вторым аргументом. Сначала в этот файл запишем имя пакета и необходимые импорты. Потом начнем перебирать синтаксическое дерево в поиске нужной структуры, к которой будем писать распаковщик. Сначала проходим циклом по объявлениям общего вида, например, импортам, они нам не нужны. Затем проходим по объявлениям, среди которых могут быть объявления типа, констант, переменных,если попадается структура, помеченная специальным комментарием, то ставим для неё флаг, все остальное пропускаем. Если вдруг флага нет,то программа переходит к следующему объявлению. Например, структура Avatar не будет оьрабатываться, потому что к ней нет комментария, в котором говорится, что к ней надо что-то сгенерировать. Если же флаг поднят, начинается непосредственно процесс генерации кода для структуры и ее полей.

Создается функция Unpack. Дальше внутри этой функции создается ридер, чтобы было возможно итерироваться по полям. В цикле смотрим сначала теги, например, если там есть cgen на минус, то это значит, что мы хотим это поле пропустить. Затем получаем имя поля и тип поля. Смотрим, какой тип. Если это int, то генерируем код для шаблона распаковки int, если это строчка, то генерируем распаковку строчки. А если какое-то новое поле, то ругаюсь и завершаем программу. В конце пишем, что нужно вернуть пустую ошибку и завершить функцию. Пишем закрывающую скобку, пустую строку. И это все! Вот код, который сгенерировался для распаковки структуры.

```
package main

import "encoding/binary"
import "bytes"

func (in *User) Unpack(data []byte) error {
        r := bytes.NewReader(data)

        // ID
        var IDRaw uint32
        binary.Read(r, binary.LittleEndian, &IDRaw)
        in.ID = int(IDRaw)

        // Login
        var LoginLenRaw uint32
        binary.Read(r, binary.LittleEndian, &LoginLenRaw)
        LoginRaw := make([]byte, LoginLenRaw)
        binary.Read(r, binary.LittleEndian, &LoginRaw)
        in.Login = string(LoginRaw)
```

```
        // Flags
        var FlagsRaw uint32
        binary.Read(r, binary.LittleEndian, &FlagsRaw)
        in.Flags = int(FlagsRaw)
        return nil
}
```

У нас получилась функция с готовым кодом для каждого поля. Теперь мы всю обработку производим не в цикле с через магию рефлексии, а можем присваивать сразу в нужное поле.

Раньше функции Unpack не было, программа даже не компилировалась. Попробуйте запустить теперь и убедиться, что все работает и правильно распаковывается.

Использованный нами подход кодогенерации очень широко применяется компилятором, для генерации кода упаковщиков-распаковщиков, много синтаксических анализаторов используют такой подход. Напомню, что минус рефлексии в том, что в рантайме выполняется дополнительная работа за счет создания структуры рефлексии и ее обработки. В кодогенерации вы не выполняете дополнительную работу в рантайме, потому что у вас уже готовый код, однако в данном случае недостатком является то, что вам нужно писать программу отдельную и отдельно ее отлаживать, что иногда бывает довольно нетривиально.

Кодогенерацию хорошо использовать, если у вас какая-то очень большая, очень интенсивная нагрузка, и вы хотите оптимизировать какие-то горячие участки. И раз уже мы заговорили про горячие участки, то в следующем разделе мы рассмотрим вопрос о профилировании и сравним, что быстрее: функция Unpack через рефлексию или Unpack, которая получилась кодогенерацией?

Бенчмарки и производительность

Система бенчмарков в Go

В предыдущей главе мы написали две реализации распаковки бинарных данных через рефлексию и через кодогенерацию, но мы так и не выяснили, какой из этих вариантов быстрее. В этом разделе рассмотрим, каким образом можно делать бенчмарки кода и смотреть, какой из вариантов быстрее.

```
import (
        "bytes"
        "encoding/binary"
        "fmt"
        "reflect"
        "testing"
)


func BenchmarkCodegen(b *testing.B) {
        u := &User{}
        for i := 0; i < b.N; i++ {
                u = &User{}
                u.UnpackBin(data)
        }
}

func BenchmarkReflect(b *testing.B) {
        u := &User{}
        for i := 0; i < b.N; i++ {
                u = &User{}
                UnpackReflect(u, data)
        }
}


/*
        go test -bench . unpack_test.go
```

```
        go test -bench . -benchmem unpack_test.go
*/
```

Итак, бенчмарки в Go лежат рядом с тестами, то есть в пакете с суффиксом test. Функция бенчмарка начинается с суффикса Benchmark, в неё передается переменная типа testing B, в которой вы должны сделать цикл с количеством итераций, которые хочет бенчмарк. В примере мы делаем два бенчмарка на функцию распаковки бинарных данных с помощью рефлекции и полученную с помощью кодогенерации. Запускать программу надо, как тесты, но еще и с флагом -bench, который значит, что нужно делать бенчмарк. После флага нужно указать регулярное выражение, на те тесты, которые надо запускать. В нашем случае мы просто ставим точку, чтоб запустить все тесты.

Запустив, молучаем примерно такой вывод (от запуска к запуску время может немного отличаться, так же оно варируется в зависимости от железа).

```
PASS
BenchmarkCodegen-4       1000000              2054 ns/op
BenchmarkReflect-4        300000              4339 ns/op
ok      command-line-arguments  4.436s
```

Итак, в варианте с кодогенерацией, одна операция заняла примерно 2000 наносекунд и успело выполниться 1 миллион итераций. В варианте с рефлектом одна операция занимает примерно 4000 наносекунд и успело выполниться 300 000 итераций. То есть, грубо можно говорить, что вариант с кодогенерацией вдвое быстрее, чем вариант с рефлексией.

Помимо того что мы можем посмотреть скорость выполнения итераций, мы можем замеры использования памяти. Делается это при помощи ключа benchmem. Поробуйте запустить код примера.

```
go test -bench . -benchmem ./unpack_test.go
```

Итак, можно увидеть, что вариант с кодогенерацией тратит меньше памяти на одну операцию и делаем меньше аллокаций памяти.

Если кодогенерация такая крутая вещь, почему бы её не использвать для json. Эта идея настолько хорошая, что в mail.ru сделали такую кодогенерацию и получили впечатляющий рост производительности. Пакет называется easyjson, вы можете скачать его на GitHub.

Вы можете посмотреть в коде к уроку еще несколько примеров, в которых с помощью бенчмарк сравнивается стандартная реализация распаковки и упаковки json с реализацией с помощью кодогенерации в пакете easyjson; поиск вхождения слова в строке с помощью регулярного выражения, компилирующегося перед каждой операцией, с помощью регулярного выращения, компилирующегося всего один раз, и с помощью функции contains из пакета strings; добавление данных в пустой слайс и в слайс с заранее заданной capacity. Попробуйте позапускать их с разными ключами и проанализировать затраты по времени и по памяти у каждой из функций. Должно получиться, что, если смотреть только по скорости, декодирование и серилизация из пакета easyjson примерно в 4 раза быстрее стандартной. По скорости и количеству итераций, быстрее всего contains из пакета strings, потом идет прикомпилированная регулярка и совсем позади них идет неприкомпилированная регулярка, которую мы каждый раз компилируем, которая значительно отстает от всех других способой по всем показателям. А слайс, в котором не происходит переаллокаций, заполнять данными примерно в 15 раз быстрее, чем пустой слайс с нулевой capacity.

Мы потренировались сравнивать бенчмарки и смотреть, какой из вариантов лучше. Однако из обычного анализа бенчмарок непонятно, что именно тормозит, что именно занимает время, что именно тратит память, что именно делает аллокация памяти. В Go есть средства для того, чтобы посмотреть более детальную информацию. И это как раз следующая тема.

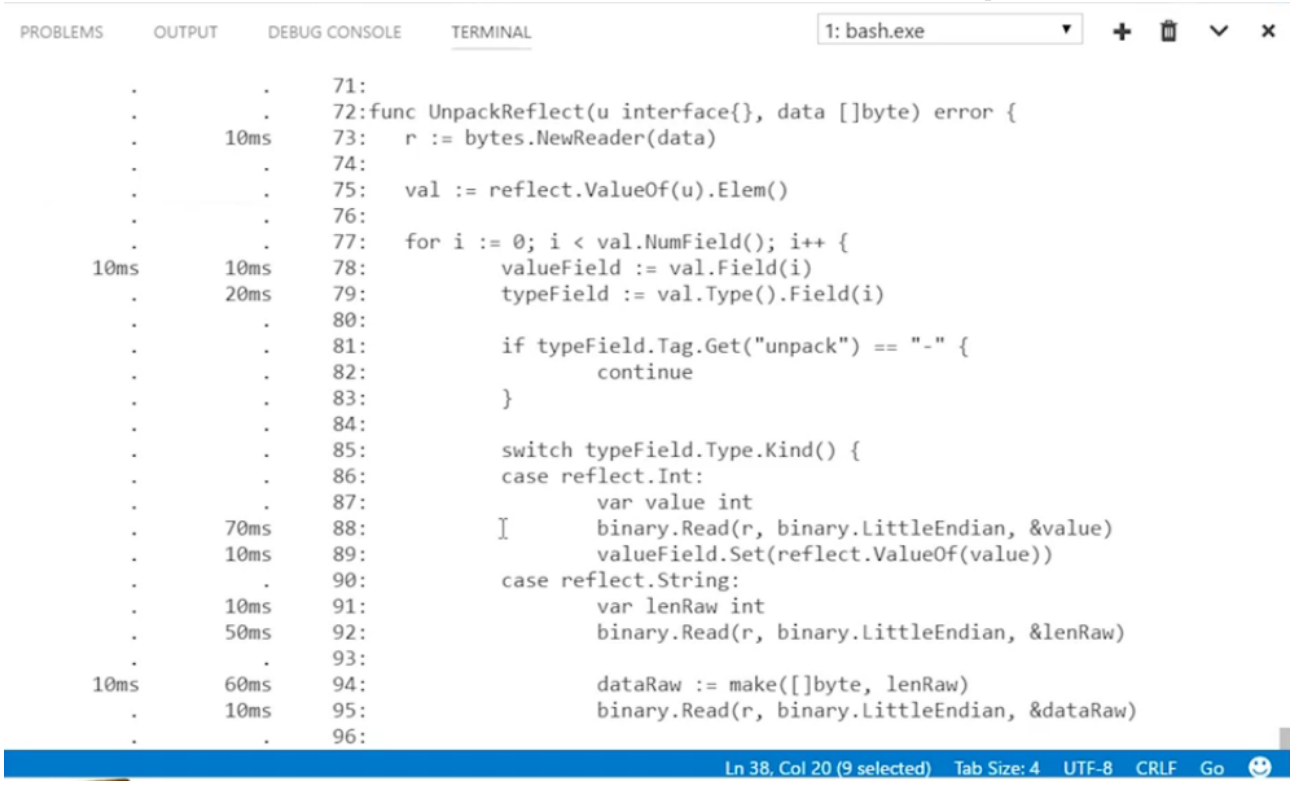
Профилирование через pprof

В предыдущем разделе мы рассмотрели, каким образом мы можем сравнить разный код на скоростьи затраты памяти. Тепер мы рассмотрим, как посмотреть не просто, какой вариант быстрее, а за счет чего какой-то вариант быстрее или медленнее. В Go вы можете снять профиль либо процессора, используя директиву cpuprofile и путь куда складировать данные, либо профиль памяти. Выполним это для уже рассмотренного файла *unpack_test.go*, выполнив команду

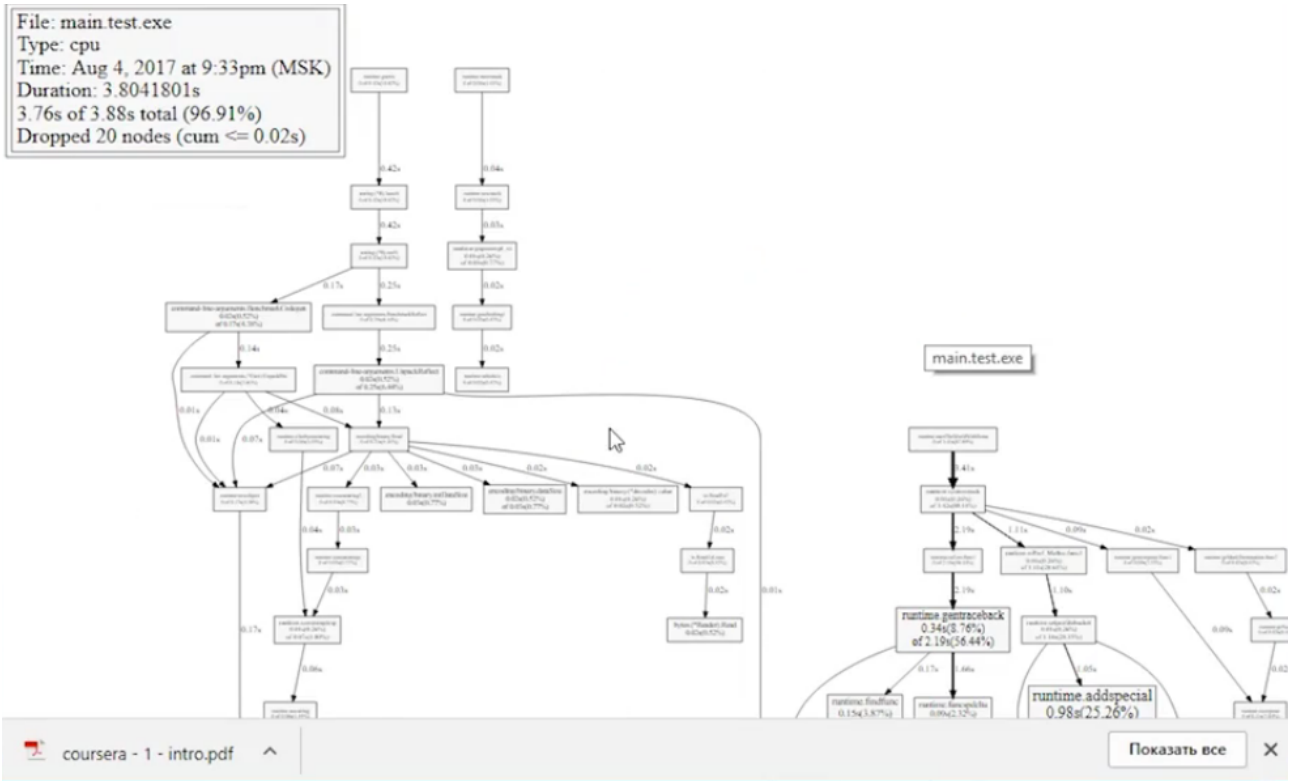
```
go test -bench . -benchmem -cpuprofile=cpu.out -memprofile=mem.out
-memprofilerate=1 unpack_test.go
```

В данном случае мы получаем профиль и цпу и памяти и они записываются в файлы cpu.out и mem.out соответственно.

Посмотрим, как проанализировать снятый профиль. В Go для этого есть инструмент, называется pprof. Запускается он таким образом: go tool pprof *имя файла*. В pprof можно посмотреть на на топ операции, которые выполнялись. Может оказаться, что в топе будет только runtime Go. Тогда можно использовать команду list и имя функции, подробности про которую интересны. Например, функцию Unpack. На экран должны вывестись те места, где было сколько-либо много задействовано CPU времени.



В функции UnpackBin в основном затрачивается время на чтение бинарных данных и преобразование слайса byte к строке. А если мы посмотрим рефлексию, тоокажется, что мы тратим много времени еще и на установку данных. Инструмент pprof позволяет так же посмотреть на код как бы с высоты птичьего полета и построить картинку с графиком в любом нужном вам формате. Делается при помощи команды web, которая строит красивый граф, в котором видны все зависимости и слабые места.



Чтобы посмотреть на профиль памяти нужно сделать то же самое для файла mem.out. В профиле памяти несколько разных режимов. Тут можно посмотреть место, которое в данный момент занимает больше врего места либо по количеству объектов, либо по самой памяти, или место, где было совершено больше всего локаций. Режим можно выбрать командам *alloc_space* и *alloc_objects* соответственно. Точно так же как с CPU, можно посмотреть всю информацию в виде web.

Pprof очень мощный инструмент исследования. Вы можете глубоко-глубоко зарыться и при помощи него оптимизировать вашу программу так, как вам надо. Кроме того с его помощью можете снимать дампы памяти и дампы профилирования процессора прямо с работающей программы, но это мы будем рассматривать отдельно.

sync.Pool

В примерах с распаковкой бинарных данных мы видели, что пример с рефлексией медленней, в нем больше аллокаций памяти, и он в целом требует больше памяти на одну операцию. Происходит это потому, что рантайму go нужно выделить память в данном случае каждый раз на каждую операцию. Выделение памяти — на самом деле это достаточно дорогая операция по сравнению со всем остальным. Поэтому иногда для ускорения хочется не выделять новую память каждый раз, а переиспользовать какой-то пул памяти. В go есть инструмент для этого, называется он sync.Pool. Посмотрим код.

```
package main

import (
        "bytes"
        "encoding/json"
        "sync"
        "testing"
)

const iterNum = 100

type PublicPage struct {
        ID          int
        Name        string
        Url         string
        OwnerID     int
        ImageUrl    string
```

```
        Tags        []string
        Description string
        Rules       string
}

var CoolGolangPublic = PublicPage{
        ID:          1,
        Name:        "CoolGolangPublic",
        Url:         "http://example.com",
        OwnerID:     100500,
        ImageUrl:    "http://example.com/img.png",
        Tags:        []string{"programming", "go", "golang"},
        Description: "Best page about golang programming",
        Rules:       "",
}

var Pages = []PublicPage{
        CoolGolangPublic,
        CoolGolangPublic,
        CoolGolangPublic,
}

func BenchmarkAllocNew(b *testing.B) {
        b.RunParallel(func(pb *testing.PB) {
                for pb.Next() {
                        data := bytes.NewBuffer(make([]byte, 0, 64))
                        _ = json.NewEncoder(data).Encode(Pages)
                }
        })
}

var dataPool = sync.Pool{
        New: func() interface{} {
                return bytes.NewBuffer(make([]byte, 0, 64))
        },
}

func BenchmarkAllocPool(b *testing.B) {
        b.RunParallel(func(pb *testing.PB) {
                for pb.Next() {
                        data := dataPool.Get().(*bytes.Buffer)
                        _ = json.NewEncoder(data).Encode(Pages)
                        data.Reset()
                        dataPool.Put(data)
                }
        })
}

/*
        go test -bench . -benchmem pool_test.go
*/
```

Итак, sync.Pool находится в пакете sync. В коде есть какая-то структура, где довольно много полей, то есть она занимает достаточно много места в памяти. Есть экземпляр этой структуры. И список этих структур. И тесты. В первом тесте мы будем каждый раз заново аллоцировать данные. и сериализовать этот набор структур Pages просто в json. В случае, если мы будем каждый раз его аллоцировать с нуля, то мы будем вынуждены каждый раз создавать с нуля новый буфер — каждый раз выделять слайс байт на 64 байта,

Во втором тесте мы используем вариант с пулом. Мы объявляем переменную dataPool — это в общем экземпляр структуры sync.Pool, в нем есть только одна функция — New. Она выделяет память под новый объект, если вдруг его не было уже в пуле, и возвращает пустой interface, потому что все-таки это универ-

сальный инструмент. Фактически пока это то же самое, что мы делаем при аллокации в рассмотренном тесте. Теперь в тесте мы будем брать место из пула — получать элемент функцией Get, преобразовывать к нужному нам типу, сериализовать в него. Затем нам нужно будет сбрасить с помощью команды data.Reset И вернуть данные в пул обратно: Put(data).

Когда вы запустите сразу с тестированием памяти, увидите, что вариант с новыми аллокациями делает примерно пять аллокаций на операцию. Вариант с пулом делает всего две аллокации на операцию. При этом вариант с пулом тратит примерно 40 байт на операцию, а вариант с новыми аллокациями тратит почти килобайт на операцию. Тысячу байт. За счет того, что мы держим данные в пуле, то есть нам не приходится каждый раз их аллоцировать, у нас есть какое-то количество приаллоцированных данных. Если они нам нужны, мы их оттуда берем, используем и кладем обратно. Это гораздо выгоднее, чем выделять память каждый раз, чтобы она потом была убрана сборщиком мусора.

В случае, когда вам нужно либо аллоцировать очень много структур или слайс байт, куда вы что-то пишете и потом сохраняете и выкидываете, то очень хорошим шагом будет использовать sync.Pool. Он очень здорово экономит память. Конечно, sync.Pool — это не панацея. Вы не можете контролировать количество объектов, не можете контролировать, когда эти объекты действительно очистятся сборщиком мусора. Вы вообще не можете контролировать ничего, кроме функции создания нового объекта. Но в целом он очень эффективен.

Покрытие кода тестами

Говоря о тестировании какого-то кода, очень важным является вопрос покрытия кода тестами. Что это значит? Мы можем написать довольно много разных случаев тестирования, разных данных, но мы не знаем, какое количество реального кода будут покрывать эти вызовы. В разных языках это решается по-разному. Где-то есть встроенные инструменты, где-то это внешние инструменты, где-то их нет вообще. В Go подсчет покрытия тестами встроен прямо в само тестирование. Давайте посмотрим, как это работает.

```
type User struct {
        ID int
}

var data = map[string][]byte{
        "ok":   []byte(‘{"ID": 27}‘),
        "fail": []byte(‘{"ID": 27‘),
}

func GetUser(key string) (*User, error) {
        if jsonStr, ok := data[key]; ok {
                user := &User{}
                err := json.Unmarshal(jsonStr, user)
                if err != nil {
                        return nil, fmt.Errorf("Cant decode json")
                }
                return user, nil
        }
        return nil, fmt.Errorf("User doesnt exist")
}
```

У нас есть код, есть структура User, есть список юзеров, и функция получения юзера, которая пытается получить юзера по ключу из map’ки. Если он там есть, то она распаковывает json и возвращает уже непосредственно этого юзера как структуру. Если нет, то она ругается, что такого пользователя нет. В случае, если не получилось распаковать json, то тоже возвращается ошибка. Теперь давайте посмотрим на тест.

```
package main

import (
        "testing"
        "reflect"
)
```

```
type TestCase struct{
        Key string
        User *User
        IsError bool
}

func TestGetUser(t *testing.T) {
        cases := []TestCase{
                TestCase{"ok", &User{ID: 27}, false},
                // TestCase{"fail", nil, true},
                TestCase{"not_exist", nil, true},
        }

        for caseNum, item := range cases {
                u, err := GetUser(item.Key)

                if item.IsError && err == nil {
                        t.Errorf("[%d] expected error, got nil", caseNum)
                }
                if !item.IsError && err != nil {
                        t.Errorf("[%d] unexpected error", caseNum, err)
                }
                if !reflect.DeepEqual(u, item.User) {
                        t.Errorf("[%d] wrong results: got %+v, expected %+v",
                        caseNum, u, item.User)
                }
        }

}
```

У нас есть некоторое количество тестов. Один из них закомментирован, позже станет понятно, почему. Сейчас важен не столько сам код, с ним вы сможете разобраться сами, сколько то, как мы его будем запускать. Если мы запустим просто go test, то мы увидим, что наш тест прошел, действительно, ничего не упало. Но вопрос покрытия непонятен. Для того чтобы считалось покрытие, есть опция cover. То есть запускать надо

```
go test -v -cover
```

Наш тест пройдет, при этом покрытие будет всего 85 процентов. Конечно же есть опция для того, чтобы построить более подробный отчет. Для этого надо выполнить

```
go test -coverprofile=cover.out
go tool cover -html=cover.out -o cover.html
```

Вот так выглядит отчет.

Зеленым отмечен тот код, который запускается. Красным — код, не запускающийся при выполнении тестов. То есть у нас запускается случай, когда мы не смогли распаковать json. Для того, чтобы весь код был покрыт тестами нам достаточно раскоментировать строку, на которую мы обращали внимание в начале.

Вопрос покрытия кода тестами лучше всего доводить до очень приемлемого уровня. Не всегда стоит стремиться к стопроцентному покрытию, однако, если вы делаете не бизнес-приложение, а какие-то системные утилиты, более того, системные lib'ы, или какие-то пакеты для других разработчиков, то покрытие лучше стараться делать максимально возможным. Это очень сильно уменьшит количество багов, о которых вам будут сообщать.

XML

Еще одним приемом в аспекте производительности, про которую стоит рассказать, является поточная обработка данных. Давайте рассмотрим следующий пример. У нас есть XML-код, и мы хотим его распарсить. Как это делается в лоб?

```
type User struct {
        ID      int    `xml:"id,attr"`
        Login   string `xml:"login"`
        Name    string `xml:"name"`
        Browser string `xml:"browser"`
}

type Users struct {
        Version string `xml:"version,attr"`
        List    []User `xml:"user"`
}

func CountStruct() {
        logins := make([]string, 0)
        v := new(Users)
        err := xml.Unmarshal(xmlData, &v)
        if err != nil {
                fmt.Printf("error: %v", err)
                return
        }
        for _, u := range v.List {
                logins = append(logins, u.Login)
        }
}
```

Мы создаем у себя структуру Users, куда будем все парсить. Далее мы парсим в лоб: мы указываем все наши данные, которые у нас есть, и указываем, куда их парсить. То есть мы мало того, что загружаем в память весь slice byte, в котором содержится XML, то есть тратим память на него, но еще и тратим память на все структуры, которые там есть. Соответственно, если у нас будет очень большой объем данных, мы не сможем загрузить его, скорее всего, весь в память. И, вообще, держать все в памяти — не всегда эффективно, потому что чаще всего мы проходимся по данным в цикле. Соответственно, если мы хотим получить доступ только к одной записи одновременно, то все они нам не нужны, нам стоит тогда парсить их по порядку, одну за другой, и работать с одной записью, не загружая в память сразу все. Вот следующий пример, в котором то же самое делается оптимальнее.

```
func CountDecoder() {
        input := bytes.NewReader(xmlData)
        decoder := xml.NewDecoder(input)
        logins := make([]string, 0)
        var login string
        for {
                tok, tokenErr := decoder.Token()
                if tokenErr != nil && tokenErr != io.EOF {
                        fmt.Println("error happend", tokenErr)
                        break
```

```
                } else if tokenErr == io.EOF {
                        break
                }
                if tok == nil {
                        fmt.Println("t is nil break")
                }
                switch tok := tok.(type) {
                case xml.StartElement:
                        if tok.Name.Local == "login" {
                                if err := decoder.DecodeElement(&login, &tok); err != nil {
                                        fmt.Println("error happend", err)
                                }
                                logins = append(logins, login)
                        }
                }
        }
}
```

Теперь мы создаем ридер, которые читает из слайса byte и декодер, куда передаем наш ридер. И теперь мы будем парсить записи одну за другой. Мы даже можем не парсить всю структуру вообще, мы можем вытащить только нужную нам строчку, например.

Это очень удобный прием. Подобное есть и для json — там тоже есть свой ридер, чтобы можно было организовывать поточный парсинг, не пытаясь загрузить сразу все в память. С помощью ридеров удобно читать кусками данные с диска, данные из сети, не загружая их все в память. Если сравнить два подхода банчмарком, окажется, что новый почти в два раза быстрее, кроме того существенно эффективнее по памяти и числу аллокаций.