

0. 语言基础

数据类型

- array&slice

通过array的切片可以切出slice，也可以使用make创建slice，此时golang会生成一个匿名的数组。

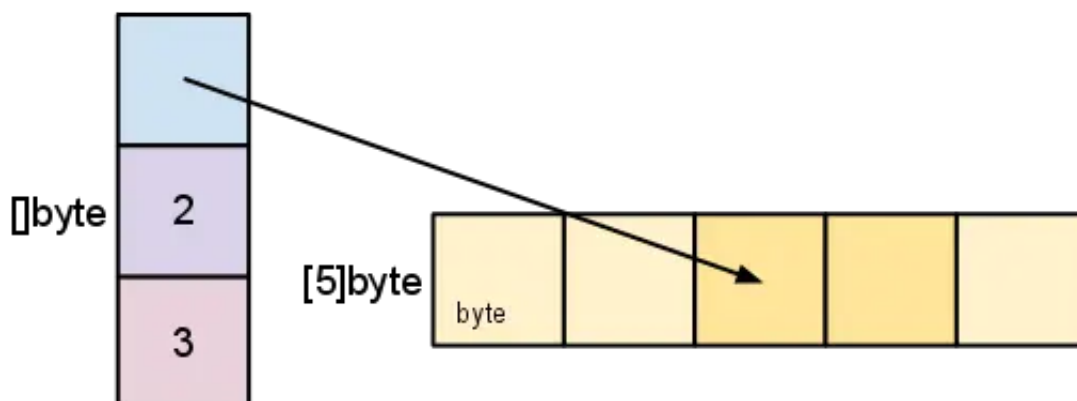
```
//创建数组
letters := [...]string{"a", "b", "c", "d"}
letters := [4]string{"a", "b", "c", "d"}

//从匿名数组创建切片
letters := []string{"a", "b", "c", "d"}

//通过make创建切片
var s []byte
s = make([]byte, 5, 5)
// s == []byte{0, 0, 0, 0, 0}

//从数组创建切片
x := [3]string{"Лайка", "Белка", "Стрелка"}
s := x[:] // a slice referencing the storage of x
```

array是值类型，slice则是复合类型，同时slice是基于array实现的。slice的第一个内容为指向数组的指针，然后是其长度和容量。



因为slice依赖其底层的array，修改slice本质是修改array，而array又是有大小限制，当超过slice的容量，即数组越界的时候，需要通过动态规划的方式创建一个新的底层数组，把原有的数据复制到新数组，然后slice的指针指向新的数组，这也是append函数的基本原理。所以array是定长的（初始化的时候指定长度，长度也是数组类型的一部分），而slice则是数据长度可变的，支持动态扩容。

因为array是值类型，作为函数参数传递时，会对整个数组进行拷贝，性能较差。

```
func main() {
    var arr = [10]int{4,5,7,11,8,9}
    fmt.Println(arr) //[4,5,7,11,8,9,0,0,0,0]
    //验证数组是值拷贝传递
    AddOne(arr)
    fmt.Println(arr) //[4,5,7,11,8,9,0,0,0,0]
}
func AddOne(arr [10]int){
    arr[9] = 999999
    fmt.Println(arr) //[4,5,7,11,8,9,0,0,0,999999]
}
```

而slice是一种引用类型，但是，在函数中作为参数传递时实际上也不是严格意义上的传递引用，因为slice本身也发生了拷贝，只不过它们都指向了同一个底层数组，函数内的操作即对切片的修改操作了底层数组。

```
...
slice := make([]int, 2, 3)
for i := 0; i < len(slice); i++ {
    slice[i] = i
}

fmt.Printf("slice %v %p \n", slice, &slice) //slice [0 1] 0xc42000a1e0

ret := changeSlice(slice)//func s [0 1] 0xc42000a260
//slice [0 1] 0xc42000a1e0, ret [0 1 3]
fmt.Printf("slice %v %p, ret %v \n", slice, &slice, ret)

ret[1] = 1111

//slice [0 1111] 0xc42000a1e0, ret [0 1111 3]
fmt.Printf("slice %v %p, ret %v \n", slice, &slice, ret)
}

func changeSlice(s []int) []int {
    fmt.Printf("func s %v %p \n", s, &s)
    s = append(s, 3)
    return s
}
```

```
}
```

不过，如果在函数内，append操作超过了原始切片的容量，将会有个新建底层数组的过程，那么此时再修改函数返回切片，则不会再影响原始切片。

```
func main() {
    slice := make([]int, 2, 2)
    for i := 0; i < len(slice); i++ {
        slice[i] = i
    }

    fmt.Printf("slice %v %p \n", slice, &slice)    //slice [0 1] 0xc42000a1a0

    ret := changeSlice(slice)//func s [0 1] 0xc42000a200
    //slice [-1 1] 0xc42000a1a0, ret [-1 1111 3]
    //函数内的修改没有影响到slice的底层数组
    fmt.Printf("slice %v %p, ret %v \n", slice, &slice, ret)

    ret[1] = -1111

    //slice [-1 1] 0xc42000a1a0, ret [-1 -1111 3]
    fmt.Printf("slice %v %p, ret %v \n", slice, &slice, ret)
}

func changeSlice(s []int) []int {
    fmt.Printf("func s %v %p \n", s, &s)
    s[0] = -1
    s = append(s, 3)
    s[1] = 1111
    return s
}
```

当然，如果为了修改原始变量，可以指定参数的类型为指针类型。传递的就是slice的内存地址。函数内的操作都是根据内存地址找到变量本身。

```
func main() {
    slice := []int{0, 1}
    fmt.Printf("slice %v %p \n", slice, &slice)//slice [0 1] 0xc42000a1e0

    changeSlice(&slice)//func s [0 1] 0xc42000a1e0
    fmt.Printf("slice %v %p \n", slice, &slice)//slice [-1 1111 3] 0xc42000a1e0

    slice[1] = -1111
}
```

```

    fmt.Printf("slice %v %p \n", slice, &slice)//slice [-1 -1111 3] 0xc42000a1e0
}

func changeSlice(s *[]int) {
    fmt.Printf("func s %v %p \n", *s, s)
    (*s)[0] = -1
    *s = append(*s, 3)
    (*s)[1] = 1111
}

```

更多细节参考：

- [Golang Slice详解](#)
- [Golang切片与函数参数“陷阱”](#)

• map

Golang的map基于哈希表实现，并采用开放寻址法中的线性探测法来处理哈希冲突。

更多细节参考：

[深入解析Golang的map设计](#)

1. 项目工程

1.1. 项目结构

golang工程典型目录结构

```

-- go_project      // go_project为GOPATH目录
-- bin
    -- myApp1      // 编译生成 go build -o ./bin/myApp1 myApp1
    -- myApp2      // 编译生成
    -- myApp3      // 编译生成
-- pkg
-- src
    -- myApp1      // project1
        -- models
        -- controllers

```

```
-- others
-- main.go
-- myApp2      // project2
-- models
-- controllers
-- others
-- main.go
-- myApp3      // project3
-- models
-- controllers
-- others
-- main.go
```

1.2. 插件管理

在VS Code里配置golang开发环境的时候，因为“墙”的存在，经常会遇到下载插件失败的问题，可以参考[VS Code配置Go语言开发环境](#)配置国内的GOPROXY，比手动安装插件方便很多。

VS Code关于golang的settings:

```
{
  "go.buildOnSave": "workspace",
  "go.lintOnSave": "package",
  "go.vetOnSave": "package",
  "go.buildTags": "",
  "go.buildFlags": [],
  "go.lintFlags": [],
  "go.vetFlags": [],
  "go.coverOnSave": false,
  "go.autocompleteUnimportedPackages": true,
  "go.inferGopath": true,
  "go.gotoSymbol.includeImports": true,
  "go.useCodeSnippetsOnFunctionSuggest": true,
  "go.formatOnSave": true,
  "go.formatTool": "goreturns",
  "go.goroot": "/usr/local/go",
  "go.gopath": "/home/workspace/Notes/golang",
  "go.gocodeAutoBuild": false,
  "files.autoSave": "onFocusChange",
  "go.docsTool": "gogetdoc",
  "go.toolsGopath": "/usr/local/go/pkg/tool/linux_amd64"
}
```

- 1.2.1. 备选：手动安装go插件

```
$ echo $GOPATH
/home/workspace/Notes/golang
$ mkdir -p $GOPATH/src/golang.org/x/
$ cd $GOPATH/src/golang.org/x/

# 直接从github上下载插件源码
$ git clone https://github.com/golang/tools.git tools
$ git clone https://github.com/golang/lint.git lint

# 手动安装插件
$ go install golang.org/x/lint/golint
```

1.3 govendor

- 安装

```
$ go get -u github.com/kardianos/govendor
```

- 基础用法

```
$ cd $GOPATH/src
#初始化vendor目录
$ govendor init

#添加依赖包
$ govendor fetch github.com/gorilla/mux
#更新依赖包
$ govendor update github.com/gorilla/mux
#移除依赖包
$ govendor remove github.com/gorilla/mux

#列出本地所有依赖包
$ govendor list
```

1.4 Modules

[关于Go Modules，看这一篇文章就够了](#)

• Modules demo

tiedotcli这个项目依赖于本地的两个package，分别是common目录下的cmd和dbmgmt两个package，其中dbmgmt这个package又依赖于一个github上的开源项目，我们初始的代码结构如下：

```
/work/code/github $ tree gowork
gowork
├── src
│   ├── common
│   │   ├── cmd
│   │   │   └── cmd.go
│   │   └── dbmgmt
│   │       └── dbmgmt.go
│   └── tiedotcli
│       └── tiedotcli.go
```

各个源文件的包名和依赖包名如下：

```
//cmd.go
package cmd

...
```

```
//dbmgmt.go
package dbmgmt

import (
    ...
    "github.com/HouzuoGuo/tiedot/db"
)

...
```

```
//tiedotcli.go
package main

import (
    "common/cmd"
    "common/dbmgmt"
    ...
)

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

- 方法1. 分别build各个本地module

先在common/cmd下面执行 `go mod init common/cmd` 或者 `go mod init`，然后会自动生成go.mod文件。

```
/work/code/github/gowork/src/common/cmd $ go mod init
/work/code/github/gowork/src/common/cmd $ ls
cmd.go go.mod
```

然后在common/dbmgmt下面执行执行 `go mod init common/dbmgmt` 或者 `go mod init`：

```
/work/code/github/gowork/src/common/dbmgmt $ go mod init
go: creating new go.mod: module common/dbmgmt
/work/code/github/gowork/src/common/dbmgmt $ ls
dbmgmt.go go.mod
/work/code/github/gowork/src/common/dbmgmt $ cat go.mod
module common/dbmgmt

go 1.14
```

因为dbmgmt依赖了github上的tiedot这个项目，所以继续执行 `go build` 来构建这个本地module：


```
/work/code/github/gowork/src/common/dbmgmt $ go build
go: finding module for package github.com/HouzuoGuo/tiedot/db
go: found github.com/HouzuoGuo/tiedot/db in github.com/HouzuoGuo/tiedot v0.0.0-20200330175510-6fb216206052
/work/code/github/gowork/src/common/dbmgmt $ cat go.mod
module common/dbmgmt

go 1.14

require github.com/HouzuoGuo/tiedot v0.0.0-20200330175510-6fb216206052
```

我们发现build过程中依赖项目被自动找到并下载到了本地，go.mod文件也被自动更新了。

最后在我们的tiedotcli项目下面执行 `go mod init`：

```
/work/code/github/gowork/src/tiedotcli $ go mod init tiedotcli
go: creating new go.mod: module tiedotcli
/work/code/github/gowork/src/tiedotcli $ cat go.mod
module tiedotcli

go 1.14
```

OK，现在我们来build一下tiedotcli，发现如下错误：

```
/work/code/github/gowork/src/tiedotcli $ go build
tiedotcli.go:4:2: package common/cmd is not in GOROOT
(/Library/Golang/1.14.2_1/libexec/src/common/cmd)
tiedotcli.go:5:2: package common/dbmgmt is not in GOROOT
(/Library/Golang/1.14.2_1/libexec/src/common/dbmgmt)
```

因为在启用go modules后，build的时候go不再基于\$GOPATH来查找第三方package，在\$GOROOT下查找无果后就会报错。所以，需要修改tiedotcli的go.mod文件来指明这些本地package的路径：

```
module tiedotcli

go 1.14

require (
    //后面的版本格式是固定的
    common/cmd v0.0.0
    common/dbmgmt v0.0.0
```

```
)

replace (
    //后面的value也可以使用绝对路径
    common/cmd => ../common/cmd
    common/dbmgmt => ../common/dbmgmt
)
```

现在，我们就可以成功build并install tiedotcli了：

```
/work/code/github/gowork/src/tiedotcli $ go build
/work/code/github/gowork/src/tiedotcli $ ls
go.mod      go.sum      tiedotcli   tiedotcli.go
/work/code/github/gowork/src/tiedotcli $ go install
```

最后，再来看一下现在的项目代码结构，所有的go modules信息以及依赖项都被自动放在了\$GOPATH/pkg/mod目录下，执行 `go install` 后二进制也还是会被自动放置到\$GOPATH/bin下面：

```
/work/code/github $ tree gowork
gowork
├── bin
│   └── tiedotcli
├── pkg
│   └── mod
│       ├── cache
│       │   └── download
│       │       ├── github.com
│       │       │   └── !houzuo!guo
│       │       │       ├── tiedot
│       │       │       │   └── @v
│       │       │           ├── list
│       │       │           ├── list.lock
│       │       │           ├── v0.0.0-20200330175510-6fb216206052.info
│       │       │           ├── v0.0.0-20200330175510-6fb216206052.lock
│       │       │           ├── v0.0.0-20200330175510-6fb216206052.mod
│       │       │           ├── v0.0.0-20200330175510-6fb216206052.zip
│       │       │           └── v0.0.0-20200330175510-6fb216206052.ziphash
│       └── sumdb
│           ├── sum.golang.org
│           │   ├── lookup
│           │   │   ├── github.com
│           │   │   │   ├── !houzuo!guo
│           │   │   │       └── tiedot@v0.0.0-20200330175510-6fb216206052
```

```

|   |   |   |   └─ tile
...
|   |   |   └─ lock
|   |   └─ github.com
|   |       └─ !houzuo!guo
|   |           └─ tiedot@v0.0.0-20200330175510-6fb216206052
...
|   |               └─ db
|   |                   └─ col.go
|   |                   └─ db.go
|   |                   └─ db_test.go
|   |                   └─ doc.go
|   |                   └─ doc_test.go
|   |                   └─ idx_test.go
|   |                   └─ query.go
|   |                   └─ query_test.go
...
|   └─ sumdb
|       └─ sum.golang.org
|           └─ latest
└─ src
    ├── common
    │   ├── cmd
    │   │   ├── cmd.go
    │   │   └─ go.mod
    │   └─ dbmgmt
    │       ├── dbmgmt.go
    │       ├── go.mod
    │       └─ go.sum
    └─ tiedotcli
        ├── go.mod
        ├── go.sum
        ├── tiedotcli
        └─ tiedotcli.go

```

查看tiedoccli项目的依赖关系:

```
/work/code/github/gowork/src/tiedotcli $ go list -m all
tiedotcli
common/cmd v0.0.0 => ../common/cmd
common/dbmgmt v0.0.0 => ../common/dbmgmt
github.com/HouzuoGuo/tiedot v0.0.0-20200330175510-6fb216206052
```

- 方法2. 直接build tiedotcli

这一次，我们还是先在common/cmd和common/dbmgmt下面执行 `go mod init`，但是不再在common/dbmgmt下面执行 `go build`，而是直接开始build tiedotcli，也是可以build成功的，只是common/dbmgmt的依赖项目tiedot被当做间接依赖，然后被添加到tiedotcli的go.mod文件中，build后的整个目录结构和前面是一致的：

```
module tiedotcli

go 1.14

require (
    //后面的版本格式是固定的
    common/cmd v0.0.0
    common/dbmgmt v0.0.0
    github.com/HouzuoGuo/tiedot v0.0.0-20200330175510-6fb216206052 // indirect
)

replace (
    //后面的value也可以使用绝对路径
    common/cmd => ../common/cmd
    common/dbmgmt => ../common/dbmgmt
)
```

我们可以感受到，go modules并不是要完全干掉GOPATH，GOPATH仍然可以用于配置我们项目的本地路径。go modules提供了更方便的方法来管理第三方package和本地package，尤其是加入了对依赖包的版本控制，这是最为重要的一点。

1.5 Go compiler

```
$ GODEBUG=installgoroot=all go install std
$
```

性能分析和调试

pprof特征分析

pprof工具用于对程序运行时重要指标或者特性的分析（Profiling），通过分析不仅可以查找到程序中的错误（内存泄漏、race冲突、协程泄漏），也能对程序进行优化。由于Go语言运行时的指标不对外暴露，因此有标准库 `net/http/pprof` 和 `runtime/pprof` 用于与外界交互。

• pprof的使用方式

在使用pprof进行特征分析时，分为两个步骤：收集样本和分析样本。

收集样本有两种方式，最常见的是引用 `net/http/pprof` 包并在程序中开启http服务器，`net/http/pprof` 包会在初始化init函数时注册路由：

```
import (  
    "net/http"  
    _ "net/http/pprof"  
)  
  
func run_pprof_server() {  
    if err := http.ListenAndServe(":6060", nil); err != nil {  
        log.Fatal(err)  
    }  
}
```

对于这种收集方式，可以在程序运行过程中，通过调用http API去实时获取并进入交互模式分析特征数据，比如：

```
$ go tool pprof http://localhost:6060/debug/pprof/heap #堆内存数据  
$ go tool pprof http://localhost:6060/debug/pprof/profile?seconds=30 # 30秒的CPU数据
```

可以在浏览器端通过访问 `http://<ip address>:6060/debug/pprof/` 来查看所有支持的特征数据。

也可以先将收集的特征数据保存在本地文件再执行 `go tool pprof` 命令分析：

```
$ curl -o heap.out http://localhost:6060/debug/pprof/heap  
$ go tool pprof heap.out
```

另一种收集样本的方式是直接在代码中需要分析的位置嵌入分析函数，比如下面的代码，在程序调用 `StartCPUProfile` 函数结束后，收集的CPU特征数据将会被保存在cpu.prof这个文件里，执行 `go tool pprof cpu.prof` 命令即可进入交互式分析模式。

```
import (  
    "runtime/pprof")
```

```

"log"
"os"
"runtime/pprof"
)

func busyLoop() {
    for i := 0; i < 1000000000; i++ {
    }
}

func pprof_cpu() {
    f, err := os.Create("cpu.prof")
    if err != nil {
        log.Fatal(err)
    }
    defer f.Close()

    if err := pprof.StartCPUProfile(f); err != nil {
        log.Fatal(err)
    }
    defer pprof.StopCPUProfile()
    busyLoop()
}

```

• pprof支持的特征数据类型

1. /debug/pprof/heap - 堆内存分析
2. /debug/pprof/goroutine - 协程栈分析，一是可以查看协程数量，查看协程是否泄漏；二是可以查看协程在执行哪些函数，判断当前协程是否健康。
3. /debug/pprof/profile - CPU占用分析，uri后面可以加参数指定收集多长时间的数据，比如/debug/pprof/profile?seconds=20收集20秒的CPU占用数据。
4. /debug/pprof/mutex - 用于mutex阻塞分析，可以查看锁争用导致的休眠时间。需要先在代码里调用 `runtime.SetMutexProfileFraction(1)` 开启锁追踪，默认没有开启。

• 案例-用pprof分析堆内存

比如下面的代码，我们创建几个goroutine去给全局变量申请内存：

```

var gMap = make(map[string]string)
var gSlice1 []byte
var gSlice2 []byte
var MiB = 1024 * 1024

```

```

func run_pprof_server() {
    if err := http.ListenAndServe(":6060", nil); err != nil {
        log.Fatal(err)
    }
}

func new_map_in_heap() {
    for i := 0; i < 999999; i++ {
        gMap["KeyID"+fmt.Sprint(i)] = "value" + fmt.Sprint(i)
    }
    //让函数阻塞在这里不退出
    <-make(chan int)

    //避免分配的内存提前被回收
    fmt.Println("Size of gMap:", len(gMap))
}

func new_slice_in_heap1() {
    gSlice1 = make([]byte, 12*MiB)
    new_slice_in_heap2()
    fmt.Println("Size of gSlice1:", len(gSlice1))
}

func new_slice_in_heap2() {
    gSlice2 = make([]byte, 24*MiB)
    <-make(chan int)

    fmt.Println("Size of gSlice2:", len(gSlice2))
}

func http_pprof_heap() {
    go run_pprof_server()

    go new_map_in_heap()

    go new_slice_in_heap1()

    <-make(chan int)
}

```

当执行这段代码的时候，我们可以用 `go tool pprof http://localhost:6060/debug/pprof/heap` 命令进入分析堆内存的交互模式，默认进入的是 `inuse_space` 模式，用 `top` 指令可以看到程序目前分配的正在使用的堆内存的统计，默认按flat值排序，也可以用 `top -cum` 指定按cum排序：

```

go tool pprof http://localhost:6060/debug/pprof/heap
Fetching profile over HTTP from http://localhost:8080/debug/pprof/heap
Saved profile in
/root/pprof/pprof.runtime.test.alloc_objects.alloc_space.inuse_objects.inuse_space.031.pb.
gz
File: runtime.test
Type: inuse_space
Time: Sep 21, 2023 at 8:43am (CST)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top
Showing nodes accounting for 98.13MB, 100% of 98.13MB total
      flat flat%   sum%        cum   cum%   command-line-arguments.new_map_in_heap
    61.63MB 62.80% 62.80%    62.13MB 63.31%   command-line-arguments.new_slice_in_heap2
      24MB 24.46% 87.26%     24MB 24.46%   command-line-arguments.new_slice_in_heap1
      12MB 12.23% 99.49%     36MB 36.69%   command-line-arguments.new_slice_in_heap1
      0.50MB 0.51% 100%      0.50MB 0.51%   fmt.Sprintf

```

其中，各个数据段的意义：

1. **flat** 列表示当前函数（最右边的函数）分配的堆内存，这一列相加得到的就是总的正在使用的堆内存：98.13MB。
2. **flat%** 表示flat占总字段的百分比，这一列相加之和为1。
3. **sum%** 表示本行以及上面的 **flat%** 之和，表示本行及之前的函数总共占用堆内存的百分比。
4. **cum** 列表示当前函数以及调用函数分配的堆内存。new_slice_in_heap1函数调用了new_slice_in_heap2函数，所以new_slice_in_heap1的cum就是自己的flat加上new_slice_in_heap2的cum。
5. **cum%** 表示cum字段占总字段的百分比，注意，这一列相加的和会超过1，因为计算百分比的分母仍然是所有flat之和，所有cum相加肯定大于所有flat之和。比如，对于new_map_in_heap函数，**cum%** = 62.13 MB / 98.13 MB = 63.31%。

这些统计数据段在不同的采样数据中会有不同的意义，但是都类似。

用 **list** 命令可以列出指定函数内分配堆内存的具体位置和分配的内存大小：

```

(pprof) list command-line-arguments.new_map_in_heap
Total: 98.13MB
ROUTINE ===== command-line-arguments.new_map_in_heap in
/home/dev/workspace/github/Notes/golang/src/runtime/http_pprof_heap.go
    61.63MB    62.13MB (flat, cum) 63.31% of Total
      .      .      23:func new_map_in_heap() {
      .      .      24:   fmt.Println("Entering new_map_in_heap")
      .      .      25:   for i := 0; i < 999999; i++ {
    61.63MB    62.13MB    26:       gMap["KeyID"+fmt.Sprintf(i)] = "value" +
fmt.Sprintf(i)

```



```

        .        .      27:    }
        .        .      28:    //让函数阻塞在这里不退出
        .        .      29:    <-make(chan int)
        .        .      30:
        .        .      31:    //避免分配的内存提前被回收
(pprof) list command-line-arguments.new_slice_in_heap1
Total: 98.13MB
ROUTINE ===== command-line-arguments.new_slice_in_heap1 in
/home/dev/workspace/github/Notes/golang/src/runtime/http_pprof_heap.go
    12MB      36MB (flat, cum) 36.69% of Total
        .        .      36:func new_slice_in_heap1() {
        .        .      37:    fmt.Println("Entering new_slice_in_heap1")
    12MB      12MB      38:    gSlice1 = make([]byte, 12*MiB)
        .      24MB      39:    new_slice_in_heap2()
        .        .      40:    fmt.Println("Size of gSlice1:", len(gSlice1))
        .        .      41:    fmt.Println("Exiting new_slice_in_heap1")
        .        .      42:}
        .        .      43:
        .        .      44:func new_slice_in_heap2() {

```

用 `tree` 命令可以打印函数的调用链，得到函数调用的堆栈信息：

```

(pprof) tree
Showing nodes accounting for 98.13MB, 100% of 98.13MB total
-----+-----
      flat  flat%   sum%       cum   cum%   calls calls% + context
-----+-----
    61.63MB 62.80% 62.80%    62.13MB 63.31%           | command-line-
arguments.new_map_in_heap
                                0.50MB  0.8% |   fmt.Sprint
-----+-----
                                24MB  100% |   command-line-
arguments.new_slice_in_heap1
    24MB 24.46% 87.26%    24MB 24.46%           | command-line-
arguments.new_slice_in_heap2
-----+-----
    12MB 12.23% 99.49%    36MB 36.69%           | command-line-
arguments.new_slice_in_heap1
                                24MB 66.67% |   command-line-
arguments.new_slice_in_heap2
-----+-----
                                0.50MB  100% |   command-line-
arguments.new_map_in_heap
    0.50MB  0.51% 100%    0.50MB  0.51%           |   fmt.Sprint
-----+-----

```

在heap中，可以显示下面4种不同的统计类型：

1. `alloc_objects`：表示已经被分配的对象的数量，这个类型没有考虑对象的释放情况。
2. `alloc_space`：表示已经分配的内存大小，这个类型没有考虑对象的释放情况。
3. `inuse_objects`：表示正在使用的对象的数量。
4. `inuse_space`（默认类型）：表示正在使用的内存大小。

在交互模式下输入对应指令就可以切换到对应的展示类型，切换后top指令里flat的意义也会随之变化：

```
(pprof) alloc_objects
(pprof) top
Showing nodes accounting for 2867253, 100% of 2867255 total
Dropped 2 nodes (cum <= 14336)
      flat flat% sum%      cum cum%
  1982504 69.14% 69.14%   2867253 100% command-line-arguments.new_map_in_heap
   884749 30.86% 100%    884749 30.86% fmt.Sprint
(pprof) alloc_space
(pprof) top
Showing nodes accounting for 155.94MB, 100% of 155.94MB total
      flat flat% sum%      cum cum%
  106.44MB 68.26% 68.26%   119.94MB 76.91% command-line-arguments.new_map_in_heap
    24MB 15.39% 83.65%    24MB 15.39% command-line-arguments.new_slice_in_heap2
   13.50MB 8.66% 92.30%   13.50MB 8.66% fmt.Sprint
    12MB 7.70% 100%    36MB 23.09% command-line-arguments.new_slice_in_heap1
(pprof) inuse_objects
(pprof) top
Showing nodes accounting for 1611118, 100% of 1611120 total
Dropped 2 nodes (cum <= 8055)
      flat flat% sum%      cum cum%
  1578350 97.97% 97.97%   1611118 100% command-line-arguments.new_map_in_heap
    32768 2.03% 100%    32768 2.03% fmt.Sprint
```

• 用web指令生成图片

先在运行 `go tool pprof` 的环境上安装可视化工具 [Graphviz](#)，在交互模式下输入 `web` 指令就会生成svg格式的图片并自动在浏览器里打开，如果我们在没有图形桌面的Linux系统上使用这种方式就什么也看不到了。

```
$ go tool pprof http://localhost:6060/debug/pprof/heap
Fetching profile over HTTP from http://localhost:8080/debug/pprof/heap
Saved profile in
/root/pprof/pprof.runtime.test.alloc_objects.alloc_space.inuse_objects.inuse_space.032.pb.
gz
File: runtime.test
Type: inuse_space
Time: Sep 22, 2023 at 8:45am (CST)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) web
```

所以，更推荐的方式是下面的操作，它会自动启动一个web服务器并在浏览器端打开网页用于查看可视化数据。当我通过VS Code 远程连接到Linux服务器执行这个操作，它也可以自动在本地浏览器打开web页面，这是非常方便的。

```
$ go tool pprof -http=:8000 http://127.0.0.1:6060/debug/pprof/heap
Fetching profile over HTTP from http://127.0.0.1:6060/debug/pprof/heap
Saved profile in
/root/pprof/pprof.runtime.test.alloc_objects.alloc_space.inuse_objects.inuse_space.035.pb.
gz
Serving web UI on http://localhost:8000

# 或者
$ go tool pprof -http=:8000 cpu.prof
Serving web UI on http://localhost:8000

# 或者
¥ go tool pprof -http=:8000 pprof.XXX.samples.cpu.001.pb.gz
```

在如下的web页面中点击VIEW菜单下的按钮就可以看到下面的连线图和火焰图等，点击SAMPLE菜单可以切换数据类型。

• Base基准分析

前面的案例分析都是每次执行一次数据采样然后开始分析，Base基准分析可以用于对比两次采样数据之间的差异。这在用于判断是否发生内存泄漏或者协程泄漏时比较有用。比如下面的函数，每2秒钟创建一个协程然后让其阻塞在读取channel：

```
func http_pprof_goroutine() {
    fmt.Println("Entering http_pprof_goroutine")
    go run_pprof_server()

    a := make(chan int)
    for {
        time.Sleep(2 * time.Second)
        go func() {
            <-a
        }()
    }
}
```

我们先按以前的方式收集两份采样数据pprof.runtime.test.goroutine.003.pb.gz和pprof.runtime.test.goroutine.004.pb.gz：

```
$ go tool pprof http://127.0.0.1:6060/debug/pprof/goroutine
Fetching profile over HTTP from http://127.0.0.1:6060/debug/pprof/goroutine
Saved profile in /root/pprof/pprof.runtime.test.goroutine.003.pb.gz
File: runtime.test
Build ID: 7b6dc6301b74ff37fab0b55888b78a6ee1430b6b
Type: goroutine
Time: Oct 8, 2023 at 4:52pm (CST)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top
Showing nodes accounting for 66, 98.51% of 67 total
Showing top 10 nodes out of 37
```

	flat	flat%	sum%	cum	cum%	
	65	97.01%	97.01%	65	97.01%	runtime.gopark
	1	1.49%	98.51%	1	1.49%	runtime.goroutineProfileWithLabels
	0	0%	98.51%	1	1.49%	command-line-arguments.TestPprofGoroutine
	0	0%	98.51%	1	1.49%	command-line-arguments.http_pprof_goroutine
	0	0%	98.51%	62	92.54%	command-line-
arguments.http_pprof_goroutine.func1						
	0	0%	98.51%	1	1.49%	command-line-arguments.run_pprof_server
	0	0%	98.51%	1	1.49%	internal/poll.(*FD).Accept
	0	0%	98.51%	1	1.49%	internal/poll.(*pollDesc).wait

```

0      0% 98.51%      1  1.49%  internal/poll.(*pollDesc).waitRead (inline)
0      0% 98.51%      1  1.49%  internal/poll.runtime_pollWait
(pprof) exit
$ go tool pprof http://127.0.0.1:6060/debug/pprof/goroutine
Fetching profile over HTTP from http://127.0.0.1:6060/debug/pprof/goroutine
Saved profile in /root/pprof/pprof.runtime.test.goroutine.004.pb.gz
File: runtime.test
Build ID: 7b6dc6301b74ff37fab0b55888b78a6ee1430b6b
Type: goroutine
Time: Oct 8, 2023 at 4:52pm (CST)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top
Showing nodes accounting for 86, 98.85% of 87 total
Showing top 10 nodes out of 37
      flat  flat%   sum%        cum   cum%
      85  97.70%  97.70%      85  97.70%  runtime.gopark
       1   1.15%  98.85%       1   1.15%  runtime.goroutineProfileWithLabels
       0      0%  98.85%       1   1.15%  command-line-arguments.TestPprofGoroutine
       0      0%  98.85%       1   1.15%  command-line-arguments.http_pprof_goroutine
       0      0%  98.85%      82  94.25%  command-line-arguments.http_pprof_goroutine.func1
       0      0%  98.85%       1   1.15%  command-line-arguments.run_pprof_server
       0      0%  98.85%       1   1.15%  internal/poll.(*FD).Accept
       0      0%  98.85%       1   1.15%  internal/poll.(*pollDesc).wait
       0      0%  98.85%       1   1.15%  internal/poll.(*pollDesc).waitRead (inline)
       0      0%  98.85%       1   1.15%  internal/poll.runtime_pollWait
(pprof) exit

```

`runtime.gopark` 是协程的休眠函数。第一次采样时，有65个协程处于休眠状态。第二次采样时，有85个协程处于休眠中。如下，用Base基准分析可以直接看到两次采样的差异：多了20个处于休眠状态的协程。

```
$ go tool pprof -base /root/pprof/pprof.runtime.test.goroutine.003.pb.gz
/root/pprof/pprof.runtime.test.goroutine.004.pb.gz
File: runtime.test
Build ID: 7b6dc6301b74ff37fab0b55888b78a6ee1430b6b
Type: goroutine
Time: Oct 8, 2023 at 4:52pm (CST)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top
Showing nodes accounting for 20, 100% of 20 total
      flat flat%   sum%   cum   cum%
      20   100%   100%    20   100% runtime.gopark
       0     0%   100%    20   100% command-line-
arguments.http_pprof_goroutine.func1
       0     0%   100%    20   100% runtime.chanrecv
       0     0%   100%    20   100% runtime.chanrecv1
```

trace 事件追踪

pprof的分析可以提供一段时间内的CPU占用、内存分配、协程堆栈信息，但是这些信息都是一段时间内数据的汇总，没有提供整个周期内发生的时间，比如指定的Goroutines何时执行、执行了多长时间、何时陷入堵塞、何时解除了堵塞、GC如何影响单个goroutine的执行、STW中断花费的时间是否太长等。Go 1.5之后推出了trace工具可以提供指定时间内程序发生的事件的完整信息：

1. 协程的创建、开始和结束。
2. 协程的堵塞 - 系统调用、通道和锁。
3. 网络I/O相关事件。
4. 系统调用事件。
5. 垃圾回收相关事件。

收集trace信息的方式和pprof类似，一共有3种方式。

第一种方式是运行 `go test` 的时候，带上 `-trace` 参数标记，`go test -trace=trace.out`

第二种方式是在程序中直接调用 `runtime/trace` 包的接口，这种比较适用于命令程序：

```
import (
    "fmt"
    "log"
    "os"
    "runtime/trace"
)
```

```
// Example demonstrates the use of the trace package to trace
// the execution of a Go program. The trace output will be
// written to the file trace.out
func main() {
    f, err := os.Create("trace.out")
    if err != nil {
        log.Fatalf("failed to create trace output file: %v", err)
    }
    defer func() {
        if err := f.Close(); err != nil {
            log.Fatalf("failed to close trace file: %v", err)
        }
    }()

    if err := trace.Start(f); err != nil {
        log.Fatalf("failed to start trace: %v", err)
    }
    defer trace.Stop()

    // your program here
    RunMyProgram()
}

func RunMyProgram() {
    fmt.Printf("this function will be traced")
}
```

第三种方式是更适用于服务程序的http服务器，`net/http/pprof` 包中集成了trace的接口，比如获取20秒内的trace事件：

```
# 收集事件并存储到文件中
$ curl -o /tmp/trace.out http://127.0.0.1:6060/debug/pprof/trace?seconds=20

# 使用trace工具分析文件，会自动开启一个http服务器并在浏览器打开网页
$ go tool trace /tmp/trace.out
2023/10/17 09:00:36 Parsing trace...
2023/10/17 09:00:36 Splitting trace...
2023/10/17 09:00:36 Opening browser. Trace viewer is listening on http://127.0.0.1:42859
```

网页中包含了很多超链接，我们可以点击这些链接查看相关的trace信息：

cmd/trace: the Go trace event viewer

This web server provides various visualizations of an event log gathered during the execution of a Go program that uses the [runtime/trace](#) package.

Event timelines for running goroutines

- [View trace](#)

This view displays a timeline for each of the GOMAXPROCS logical processors, showing which goroutine (if any) was running on that logical processor at each moment. Each goroutine has an identifying number (e.g. G123), main function, and color. A colored bar represents an uninterrupted span of execution. Execution of a goroutine may migrate from one logical processor to another, causing a single colored bar to be horizontally continuous but vertically displaced.

Clicking on a span reveals information about it, such as its duration, its causal predecessors and successors, and the stack trace at the final moment when it yielded the logical processor, for example because it made a system call or tried to acquire a mutex. Directly underneath each bar, a smaller bar or more commonly a fine vertical line indicates an event occurring during its execution. Some of these are related to garbage collection; most indicate that a goroutine yielded its logical processor but then immediately resumed execution on the same logical processor. Clicking on the event displays the stack trace at the moment it occurred.

The causal relationships between spans of goroutine execution can be displayed by clicking the Flow Events button at the top.

At the top ("STATS"), there are three additional timelines that display statistical information. "Goroutines" is a time series of the count of existing goroutines; clicking on it displays their breakdown by state at that moment: running, runnable, or waiting. "Heap" is a time series of the amount of heap memory allocated (in orange) and (in green) the allocation limit at which the next GC cycle will begin. "Threads" shows the number of kernel threads in existence: there is always one kernel thread per logical processor, and additional threads are created for calls to non-Go code such as a system call or a function written in C.

Above the event trace for the first logical processor are traces for various runtime-internal events. The "GC" bar shows when the garbage collector is running, and in which stage. Garbage collection may temporarily affect all the logical processors and the other metrics. The "Network", "Timers", and "Syscalls" traces indicate events in the runtime that cause goroutines to wake up.

The visualization allows you to navigate events at scales ranging from several seconds to a handful of nanoseconds. Consult the documentation for the Chromium [Trace Event Profiling Tool](#) for help navigating the view.

- [Goroutine analysis](#)

This view displays information about each set of goroutines that shares the same main function. Clicking on a main function shows links to the four types of blocking profile (see below) applied to that subset of goroutines. It also shows a table of specific goroutine instances, with various execution statistics and a link to the event timeline for each one. The timeline displays only the selected goroutine and any others it interacts with via block/unblock events. (The timeline is goroutine-oriented rather than logical processor-oriented.)

Profiles

Each link below displays a global profile in zoomable graph form as produced by [pprof](#)'s "web" command. In addition there is a link to download the profile for offline analysis with [pprof](#). All four profiles represent causes of delay that prevent a goroutine from running on a logical processor: because it was waiting for the network, for a synchronization operation on a mutex or channel, for a system call, or for a logical processor to become available.

- [Network blocking profile](#) (↓)
- [Synchronization blocking profile](#) (↓)
- [Syscall blocking profile](#) (↓)
- [Scheduler latency profile](#) (↓)

User-defined tasks and regions

The trace API allows a target program to annotate a [region](#) of code within a goroutine, such as a key function, so that its performance can be analyzed. [Log events](#) may be associated with a region to record progress and relevant values. The API also allows annotation of higher-level [tasks](#), which may involve work across many goroutines.

The links below display, for each region and task, a histogram of its execution times. Each histogram bucket contains a sample trace that records the sequence of events such as goroutine creations, log events, and subregion start/end times. For each task, you can click through to a logical-processor or goroutine-oriented view showing the tasks and regions on the timeline. Such information may help uncover which steps in a region are unexpectedly slow, or reveal relationships between the data values logged in a request and its running time.

- [User-defined tasks](#)
- [User-defined regions](#)

Garbage collection metrics

- [Minimum mutator utilization](#)

This chart indicates the maximum GC pause time (the largest x value for which y is zero), and more generally, the fraction of time that the processors are available to application goroutines ("mutators"), for any time window of a specified size, in the worst case.

- Goroutine analysis

Goroutine analysis 子页面主要显示每个共享相同主函数的goroutine的集合，如下，有5个goroutine在执行http_pprof_trace这个函数。

Goroutines:

[testing.tRunner](#) N=1

[net/http.\(*conn\).serve](#) N=1

[command-line-arguments.http_pprof_trace.func1](#) N=5

[runtime/trace.Start.func1](#) N=1

[net/http.\(*connReader\).backgroundRead](#) N=1

N=23

点击http_pprof_trace这个超链接，进入下面这个goroutine集合的页面，可以看到每个goroutine的运行统计数据。在这个demo程序中，大部分goroutine都因为读同一个没有数据的通道被阻塞。

Goroutine Name: command-line-arguments.http_pprof_trace.func1

Number of Goroutines: 5

Execution Time: 6.29% of total program execution time

Network Wait Time: [graph\(download\)](#)

Sync Block Time: [graph\(download\)](#)

Blocking Syscall Time: [graph\(download\)](#)

Scheduler Wait Time: [graph\(download\)](#)

Goroutine	Total	Execution	Network wait	Sync block	Blocking syscall	Scheduler wait	GC sweeping	GC pause
67	8340ms	1311ns	0ns	8340ms	0ns	29μs	0ns (0.0%)	0ns (0.0%)
22	6339ms	15μs	0ns	6339ms	0ns	28μs	0ns (0.0%)	0ns (0.0%)
68	4338ms	1534ns	0ns	4338ms	0ns	11μs	0ns (0.0%)	0ns (0.0%)
23	2337ms	2706ns	0ns	2337ms	0ns	30μs	0ns (0.0%)	0ns (0.0%)
69	336ms	1924ns	0ns	336ms	0ns	12μs	0ns (0.0%)	0ns (0.0%)

点击goroutine编号可以继续查看单个goroutine基于时间轴的统计信息。在顶部(“STATS”), 有三个额外的时间轴显示统计信息。“gooutines”是现有gooutines计数的时间序列;单击它将显示它们在当时的状态分解：

GCWaiting/Runnable/Running。“Heap”是分配的堆内存量(橙色)和下一个GC周期开始时的分配限制(绿色)的时间序列。“Threads”显示了存在的内核线程的数量：每个逻辑处理器总是有一个内核线程，并且为调用非go代码(如系统调用或用C编写的函数)创建了额外的线程。

点击时间轴上的条块，左下角就会切换到对应时间点的数据。

trace

STATS (pid 1)

Goroutines:

Heap:

Threads:

G67 command-line-arguments.http_pprof_trace.func1

Flow events | Processes | View Options

Group by: Title | Event Type | Category

▼ Goroutines

- counters
- ▼ thread_sort_index
- metadata
- ▼ thread_name
- metadata
- ▼ Heap
- counters
- ▼ process_name
- metadata
- ▼ process_sort_index
- metadata
- ▼ G67 command-line-arguments.http_pprof_trace.func1
- begin_end (compact)
- ▼ Threads
- counters

2 items selected. Counter Samples (2)

Counter	Series	Time	Value
Heap	NextGC	1660.57638	0
Heap	Allocated	1660.57638	157634016

- Profiles

Profiles部分下面由四个超链接，每个链接都是一个可缩放的图片，也可以下载进行离线分析。这四个文件都表示了一个程序在逻辑处理器上运行的延迟原因：因为它正在等待网络、等待互斥锁或通道上的同步操作、等待系统调用或等待逻辑处理器可用。

关于trace的更多案例参考[Go中trace包的使用](#)

gdb调试go程序

前段时间，在项目中需要调研我们用golang写的一些工具是否符合FIPS 140-2标准，我们使用了RedHat提供的golang编译工具集（golang-tool-set），RedHat golang-tool-set重写了一些golang原生提供的crypto的库，提供了FIPS版本，它会自动从原生crypto库调用切换到调用FIPS认证的OpenSSL 1.1.1k动态库。

即便如此，还是需要调试验证当真正执行golang相关的加解密代码时，是否真的调用到了OpenSSL的实现，因此用到了gdb来调试。

在调试过程中，我们需要打一些函数断点，但是直接用golang源代码中的函数名去打断点是无法找到相关的符号的，因为golang代码编译后生成的ELF文件里，源代码函数对应的符号名发生了变化。可以用 `readelf` 工具去看二进制的符号表，找到ELF符号表中的符号名去打断点就可以了。

```
nbapp642:/home/maintenance # readelf -a /opt/VRTSoatc/otpm_new > /tmp/otpm.elf
nbapp642:/home/maintenance # egrep ".symtab|main.main|main.OpenSSL.decrypt" /tmp/otpm.elf
[44] .symtab          SYMTAB          0000000000000000 004e21c8
Symbol table '.symtab' contains 8988 entries:
5767: 00000000006fea80 2091 FUNC      LOCAL  DEFAULT 14 main.OpenSSL.decrypt
5768: 00000000006ff2c0  77 FUNC      LOCAL  DEFAULT 14 main.OpenSSL.decrypt.func
5797: 0000000000705440 1552 FUNC      LOCAL  DEFAULT 14 main.main
5798: 0000000000705a60  77 FUNC      LOCAL  DEFAULT 14 main.main.func1
```

下面是具体调试的过程细节：

```
nbapp642:/home/maintenance # gdb /opt/VRTSoatc/otpm_new
GNU gdb (GDB) Red Hat Enterprise Linux 8.2-19.el8
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
```

```
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from /opt/VRTSoatc/otpm_new...done.
warning: File "/usr/lib/golang/src/runtime/runtime-gdb.py" auto-loading has been declined
by your `auto-load safe-path' set to "$debugdir:$datadir/auto-load".
To enable execution of this file add
    add-auto-load-safe-path /usr/lib/golang/src/runtime/runtime-gdb.py
line to your configuration file "/root/.gdbinit".
To completely disable this security protection add
    set auto-load safe-path /
line to your configuration file "/root/.gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
    info "(gdb)Auto-loading safe path"
(gdb) set args -unlock " " # 设置二进制启动参数
(gdb) break main.OpenSSL.decrypt # 打函数断点
Breakpoint 1 at 0x6fea80: file
/home/dev/workspace/platformx/security/oatc/otpm/openssl.go, line 75.
(gdb) break EVP_aes_128_ecb
Function "EVP_aes_128_ecb" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 2 (EVP_aes_128_ecb) pending.
(gdb) break /usr/lib/golang/src/vendor/github.com/golang-fips/openssl-
fips/openssl/openssl.go:51
Breakpoint 3 at 0x6051c0: file /usr/lib/golang/src/vendor/github.com/golang-fips/openssl-
fips/openssl/openssl.go, line 51.
(gdb) break /home/dev/workspace/platformx/security/oatc/otpm/otpm.go:182
Breakpoint 4 at 0x705440: file /home/dev/workspace/platformx/security/oatc/otpm/otpm.go,
line 182.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /opt/VRTSoatc/otpm_new -unlock " "
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
[New Thread 0x7fffd02d3700 (LWP 102935)]
[New Thread 0x7fffcfad2700 (LWP 102936)]
[New Thread 0x7fffcf2d1700 (LWP 102937)]
[New Thread 0x7fffcfad0700 (LWP 102938)]
[New Thread 0x7fffcf2cf700 (LWP 102939)]
[New Thread 0x7ffcdace700 (LWP 102940)]
```

Thread 1 "otpm_new" hit Breakpoint 3, vendor/github.com/golang-fips/openssl-fips/openssl.init.0 () at /usr/lib/golang/src/vendor/github.com/golang-fips/openssl-fips/openssl/openssl.go:51

```
51 func init() {
```

```
(gdb) info breakpoints
```

Num	Type	Disp	Enb	Address	What
1	breakpoint	keep	y	0x00000000006fea80	in main.OpenSSL.decrypt at /home/dev/workspace/platformx/security/oatc/otpm/openssl.go:75
2	breakpoint	keep	y	<PENDING>	EVP_aes_128_ecb
3	breakpoint	keep	y	0x00000000006051c0	in vendor/github.com/golang-fips/openssl-fips/openssl.init.0 at /usr/lib/golang/src/vendor/github.com/golang-fips/openssl-fips/openssl/openssl.go:51

breakpoint already hit 1 time

```
4 breakpoint keep y 0x0000000000705440 in main.main at /home/dev/workspace/platformx/security/oatc/otpm/otpm.go:182
```

```
(gdb) c
```

Continuing.

Thread 1 "otpm_new" hit Breakpoint 2, 0x00007ffffccf07280 in EVP_aes_128_ecb () from /lib64/libcrypto.so.1.1

```
(gdb) where # 查看当前调用栈
```

```
#0 0x00007ffffccf07280 in EVP_aes_128_ecb () from /lib64/libcrypto.so.1.1
#1 0x00007ffffccf54835 in drbg_ctr_init () from /lib64/libcrypto.so.1.1
#2 0x00007ffffccf5498a in RAND_DRBG_set () from /lib64/libcrypto.so.1.1
#3 0x00007ffffccf55a45 in rand_drbg_new () from /lib64/libcrypto.so.1.1
#4 0x00007ffffccf563bc in rand_drbg_selftest () from /lib64/libcrypto.so.1.1
#5 0x00007ffffccf23494 in FIPS_selftest () from /lib64/libcrypto.so.1.1
#6 0x00007ffffccf1dd9f in FIPS_module_mode_set () from /lib64/libcrypto.so.1.1
#7 0x00007ffffcce20070 in OPENSSL_init_library () from /lib64/libcrypto.so.1.1
#8 0x00007ffff7dd6f1a in call_init (l=<optimized out>, argc=argc@entry=3, argv=argv@entry=0x7ffffffffffd648, env=env@entry=0x7ffffffffffd668) at dl-init.c:72
#9 0x00007ffff7dd701a in call_init (env=0x7ffffffffffd668, argv=0x7ffffffffffd648, argc=3, l=<optimized out>) at dl-init.c:30
#10 _dl_init (main_map=0xb18840, argc=3, argv=0x7ffffffffffd648, env=0x7ffffffffffd668) at dl-init.c:119
#11 0x00007ffff774dc6c in _dl_catch_exception () from /lib64/libc.so.6
#12 0x00007ffff7dde76e in dl_open_worker (a=0x7ffffffffffd200) at dl-open.c:794
#13 dl_open_worker (a=0x7ffffffffffd200) at dl-open.c:757
#14 0x00007ffff774dc14 in _dl_catch_exception () from /lib64/libc.so.6
#15 0x00007ffff7dde951 in _dl_open (file=0x7b4dd1 "libcrypto.so.1.1", mode=-2147483390, caller_dlopen=0x715921 <_cgo_0284165e5dc5_Cfunc__goboringcrypto_DLOPEN_OPENSSL+65>, nsid=<optimized out>, argc=3, argv=<optimized out>, env=0x7ffffffffffd668) at dl-open.c:876
#16 0x00007ffff79aaf8a in dlopen_doit () from /lib64/libdl.so.2
#17 0x00007ffff774dc14 in _dl_catch_exception () from /lib64/libc.so.6
#18 0x00007ffff774dcd3 in _dl_catch_error () from /lib64/libc.so.6
```

```
#19 0x00007ffff79ab52e in _dlerror_run () from /lib64/libdl.so.2
#20 0x00007ffff79ab02a in dlopen@@GLIBC_2.2.5 () from /lib64/libdl.so.2
#21 0x0000000000715921 in _goboringcrypto_DLOPEN_OPENSSL () at
/_/vendor/github.com/golang-fips/openssl-fips/openssl/goopenssl.h:71
#22 _cgo_0284165e5dc5_Cfunc__goboringcrypto_DLOPEN_OPENSSL (v=0xc0001e3a38) at cgo-gcc-
prolog:128
#23 0x0000000000472264 in runtime.asmcgocall () at
/usr/lib/golang/src/runtime/asm_amd64.s:844
#24 0x0000000000ae2360 in ?? ()
#25 0x00007ffffffffffd508 in ?? ()
#26 0x0000000000413a8e in runtime.persistentalloc.func1 () at
/usr/lib/golang/src/runtime/malloc.go:1340
#27 0x00000000004704cb in runtime.morestack () at
/usr/lib/golang/src/runtime/asm_amd64.s:570
#28 0x0000000000474ba5 in runtime.newproc (fn=0x7196d0 <__libc_csu_init>) at
<autogenerated>:1
#29 0x0000000000ae1dc0 in ?? ()
#30 0x00000000007196d0 in ?? ()
#31 0x00000000004702e5 in runtime.mstart () at /usr/lib/golang/src/runtime/asm_amd64.s:390
#32 0x000000000047026f in runtime.rt0_go () at /usr/lib/golang/src/runtime/asm_amd64.s:354
#33 0x0000000000000003 in ?? ()
#34 0x00007ffffffffffd648 in ?? ()
#35 0x00007ffffffffffd640 in ?? ()
#36 0x0000000000000003 in ?? ()
#37 0x00007ffffffffffd648 in ?? ()
#38 0x00007ffff761fd85 in __libc_start_main () from /lib64/libc.so.6
#39 0x000000000040509e in _start ()
```

(gdb) c

Continuing.

Thread 1 "otpm_new" hit Breakpoint 2, 0x00007ffffccf07280 in EVP_aes_128_ecb () from /lib64/libcrypto.so.1.1

(gdb) c

Continuing.

Thread 1 "otpm_new" hit Breakpoint 2, 0x00007ffffccf07280 in EVP_aes_128_ecb () from /lib64/libcrypto.so.1.1

(gdb) c

Continuing.

Thread 1 "otpm_new" hit Breakpoint 4, main.main () at /home/dev/workspace/platformx/security/oatc/otpm/otpm.go:182

```
182 func main() {
```

(gdb) where

```
#0 main.main () at /home/dev/workspace/platformx/security/oatc/otpm/otpm.go:182
```

(gdb) c

Continuing.


```

[New Thread 0x7ffffccb8b700 (LWP 104557)]
>> Please enter the security
key:U2FsdGVkX1+rrFUXVYxM4e1pruxM2yzrUnl5DzpnYpx+16r4sQ82tFBcaguvVoqKgxBWc5K8JEvgPBisQ0x0AF
GD8/JtM1DDYPKzYSqLJ5e1wPHTYLB7L/zbv8y16WCoKPDq9w8VuhhMJKm7V8PvVrvCiAE00X0YyvsUbtQXJacRFcxy
Q9cjYPULKga7/0f2QsJYLRAgz0
[Switching to Thread 0x7ffffccb8b700 (LWP 104557)]

Thread 8 "otpm_new" hit Breakpoint 1, main.OpenSSL.decrypt (o=..., key=..., iv=...,
data=..., ~r0=..., ~r1=...) at
/home/dev/workspace/platformx/security/oatc/otpm/openssl.go:75
75      func (o OpenSSL) decrypt(key, iv, data []byte) ([]byte, error) {
(gdb) n
76          logger.LoggerEntry()
(gdb) n
77          defer logger.LoggerExit()
(gdb) n
78          if len(data) == 0 || len(data)%aes.BlockSize != 0 {
(gdb) n
81          c, err := aes.NewCipher(key)
(gdb) s
crypto/aes.NewCipher (key=..., ~r0=..., ~r1=...) at
/usr/lib/golang/src/crypto/aes/cipher.go:33
33      // AES-128, AES-192, or AES-256.
(gdb) list 33,45
33      // AES-128, AES-192, or AES-256.
34      func NewCipher(key []byte) (cipher.Block, error) {
35          k := len(key)
36          switch k {
37              default:
38                  return nil, KeySizeError(k)
39              case 16, 24, 32:
40                  break
41          }
42          if boring.Enabled() {
43              return boring.NewAESCipher(key)
44          }
45          return newCipher(key)
(gdb) n
34      func NewCipher(key []byte) (cipher.Block, error) {
(gdb) n
38          return nil, KeySizeError(k)
(gdb) n
39          case 16, 24, 32:
(gdb) n
41      }
(gdb) n
42      if boring.Enabled() {

```

```

(gdb) s
vendor/github.com/golang-fips/openssl-fips/openssl.NewAESEncipher (key=..., ~r0=...,
~r1=...) at /usr/lib/golang/src/vendor/github.com/golang-fips/openssl-
fips/openssl/aes.go:48
48     func NewAESEncipher(key []byte) (cipher.Block, error) {
(gdb) s
49         c := &aesCipher{key: make([]byte, len(key))}
(gdb) list
44     }
45
46     var _ extraModes = (*aesCipher)(nil)
47
48     func NewAESEncipher(key []byte) (cipher.Block, error) {
49         c := &aesCipher{key: make([]byte, len(key))}
50         copy(c.key, key)
51
52         switch len(c.key) * 8 {
53         case 128:
(gdb) n
50         copy(c.key, key)
(gdb) n
52         switch len(c.key) * 8 {
(gdb) n
53         case 128:
(gdb) n
55         case 192:
(gdb) n
57         case 256:
(gdb) n
58             c.cipher = C._goboringcrypto_EVP_aes_256_ecb()
(gdb) list
53         case 128:
54             c.cipher = C._goboringcrypto_EVP_aes_128_ecb()
55         case 192:
56             c.cipher = C._goboringcrypto_EVP_aes_192_ecb()
57         case 256:
58             c.cipher = C._goboringcrypto_EVP_aes_256_ecb()
59         default:
60             return nil, errors.New("crypto/cipher: Invalid key size")
61         }
62
(gdb) s
vendor/github.com/golang-fips/openssl-fips/openssl._Cfunc__goboringcrypto_EVP_aes_256_ecb
(r1=0x6feb12 <main.OpenSSL.decrypt+146>) at _cgo_gotypes.go:1215
1215     _cgo_gotypes.go: No such file or directory.
(gdb) break /usr/lib/golang/src/runtime/proc.go:3681
Breakpoint 5 at 0x46de5d: file /usr/lib/golang/src/runtime/proc.go, line 3681.

```



```

(gdb) c
Continuing.
[Switching to Thread 0x7fffcead0700 (LWP 102938)]

Thread 5 "otpm_new" hit Breakpoint 5, runtime.entersyscall () at
/usr/lib/golang/src/runtime/proc.go:3681
3681          // Return to mstart, which will release the P and exit
(gdb) list
3676          if locked {
3677              // The goroutine may have locked this thread because
3678              // it put it in an unusual kernel state. Kill it
3679              // rather than returning it to the thread pool.
3680
3681              // Return to mstart, which will release the P and exit
3682              // the thread.
3683              if G00S != "plan9" { // See golang.org/issue/22227.
3684                  gogo(&_g_.m.g0.sched)
3685              } else {
(gdb) n
syscall.Syscall6 (trap=232, a1=9, a2=824636406276, a3=7, a4=18446744073709551615, a5=0,
a6=0, r1=0, r2=0, err=0) at /usr/lib/golang/src/syscall/syscall_linux.go:91
91          // Root can read and write any file.
(gdb) c
Continuing.
[Switching to Thread 0x7fffccb8b700 (LWP 104557)]

Thread 8 "otpm_new" hit Breakpoint 5, runtime.entersyscall () at
/usr/lib/golang/src/runtime/proc.go:3681
3681          // Return to mstart, which will release the P and exit
(gdb) where
#0  runtime.entersyscall () at /usr/lib/golang/src/runtime/proc.go:3681
#1  0x0000000004091dc in runtime.cgocall (fn=0x7141e0
<_cgo_0284165e5dc5_Cfunc__goboringcrypto_EVP_aes_256_ecb>, arg=0xc00003a000, ~r0=
<optimized out>) at /usr/lib/golang/src/runtime/cgocall.go:158
#2  0x0000000005fdfe9 in vendor/github.com/golang-fips/openssl-
fips/openssl._Cfunc__goboringcrypto_EVP_aes_256_ecb (r1=0x0) at _cgo_gotypes.go:1216
#3  0x0000000006000d5 in vendor/github.com/golang-fips/openssl-fips/openssl.NewAESCipher
(key=..., ~r0=..., ~r1=...) at /usr/lib/golang/src/vendor/github.com/golang-fips/openssl-
fips/openssl/aes.go:58
#4  0x0000000006436b0 in crypto/aes.NewCipher (key=..., ~r0=..., ~r1=...) at
/usr/lib/golang/src/crypto/aes/cipher.go:42
#5  0x0000000006fed77 in main.OpenSSL.decrypt (o=..., key=..., iv=..., data=..., ~r0=...,
~r1=...) at /home/dev/workspace/platformx/security/oatc/otpm/openssl.go:81
#6  0x0000000006fe792 in main.OpenSSL.DecryptBytes (o=..., passphrase=...,
encryptedBase64Data=..., ~r0=..., ~r1=...) at
/home/dev/workspace/platformx/security/oatc/otpm/openssl.go:72

```

```
#7 0x0000000006fdd98 in main.OpenSSL.DecryptString (o=..., passphrase=...,
encryptedBase64String=..., ~r0=..., ~r1=...) at
/home/dev/workspace/platformx/security/oatc/otpm/openssl.go:43
#8 0x000000000702a86 in main.(*otp).decrypt (o=0xc000229380, sk=..., dec=..., err=...)
at /home/dev/workspace/platformx/security/oatc/otpm/otp.go:91
#9 0x000000000710ab0 in main.(*support).unlock (s=0xc0001ccab0, secureKey=..., ~r0=...)
at /home/dev/workspace/platformx/security/oatc/otpm/support.go:137
#10 0x000000000704cc7 in main.handleFlag (f=0xc000192380) at
/home/dev/workspace/platformx/security/oatc/otpm/otpm.go:155
#11 0x0000000006e5a93 in flag.(*FlagSet).Visit (f=0xc000194180, fn={void (flag.Flag *)}
0xc000229e08) at /usr/lib/golang/src/flag/flag.go:461
#12 0x0000000006e5b0b in flag.Visit (fn={void (flag.Flag *)} 0xc000229e20) at
/usr/lib/golang/src/flag/flag.go:468
#13 0x000000000705930 in main.main () at
/home/dev/workspace/platformx/security/oatc/otpm/otpm.go:199
(gdb) c
Continuing.
[Switching to Thread 0x7fffcead0700 (LWP 102938)]
```

```
Thread 5 "otpm_new" hit Breakpoint 5, runtime.entersyscall () at
/usr/lib/golang/src/runtime/proc.go:3681
3681                                // Return to mstart, which will release the P and exit
(gdb) where
#0 runtime.entersyscall () at /usr/lib/golang/src/runtime/proc.go:3681
#1 0x0000000004c8aa5 in syscall.Syscall6 (trap=232, a1=9, a2=824636406276, a3=7,
a4=18446744073709551615, a5=0, a6=0, r1=0, r2=0, err=0) at
/usr/lib/golang/src/syscall/syscall_linux.go:90
#2 0x0000000004c9039 in syscall.Syscall6 (trap=232, a1=9, a2=824636406276, a3=7,
a4=18446744073709551615, a5=0, a6=0, r1=0, r2=0, err=0) at <autogenerated>:1
#3 0x0000000006ec2d3 in golang.org/x/sys/unix.EpollWait (epfd=9, events=..., msec=-1,
n=0, err=...) at
/home/dev/workspace/platformx/security/oatc/otpm/vendor/golang.org/x/sys/unix/zsyscall_lin
ux_amd64.go:56
#4 0x0000000006eeeca5 in github.com/fsnotify/fsnotify.(*fdPoller).wait
(poller=0xc0001201c0, ~r0=false, ~r1=...)
at
/home/dev/workspace/platformx/security/oatc/otpm/vendor/github.com/fsnotify/fsnotify/inoti
fy_poller.go:86
#5 0x0000000006ed785 in github.com/fsnotify/fsnotify.(*Watcher).readEvents
(w=0xc000138000) at
/home/dev/workspace/platformx/security/oatc/otpm/vendor/github.com/fsnotify/fsnotify/inoti
fy.go:206
#6 0x0000000006ecdeb in github.com/fsnotify/fsnotify.NewWatcher.func1 () at
/home/dev/workspace/platformx/security/oatc/otpm/vendor/github.com/fsnotify/fsnotify/inoti
fy.go:60
#7 0x0000000004725a1 in runtime.goexit () at
/usr/lib/golang/src/runtime/asm_amd64.s:1594
```

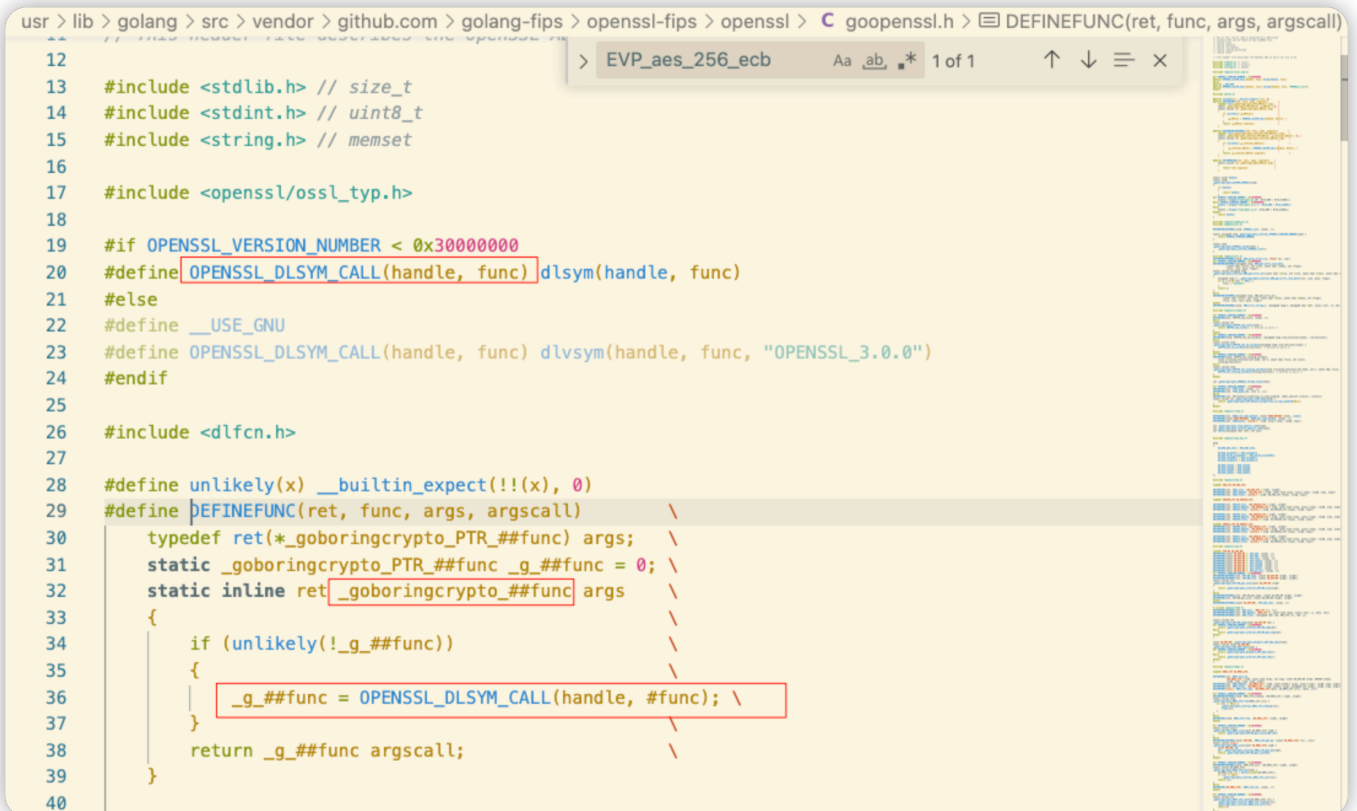
```
#8 0x0000000000000000 in ?? ()
(gdb) n
syscall.Syscall6 (trap=232, a1=9, a2=824636406276, a3=7, a4=18446744073709551615, a5=0,
a6=0, r1=0, r2=0, err=0) at /usr/lib/golang/src/syscall/syscall_linux.go:91
91                                     // Root can read and write any file.
(gdb) n

Thread 8 "otpm_new" hit Breakpoint 5, runtime.entersyscall () at
/usr/lib/golang/src/runtime/proc.go:3681
3681                                     // Return to mstart, which will release the P and exit
(gdb) quit
A debugging session is active.

        Inferior 1 [process 102934] will be killed.

Quit anyway? (y or n) y
```

调试过程中，我们无法真正跟踪进入OpenSSL的库函数，这是因为RedHat将golang调用openssl库的CGO函数写成了内联函数。比如，将OpenSSL的 `EVP_aes_256_ecb()` 函数封装成 `_goboringcrypto_EVP_aes_256_ecb()` 这么一个内联CGO函数，在调试进入 `_goboringcrypto_EVP_aes_256_ecb()` 的时候就会报错：unreadable function C._goboringcrypto_EVP_aes_256_ecb is inlined。



```
usr > lib > golang > src > vendor > github.com > golang-fips > openssl-fips > openssl > C goopenssl.h > DEFINEFUNC(ret, func, args, argscall)
12
13 #include <stdlib.h> // size_t
14 #include <stdint.h> // uint8_t
15 #include <string.h> // memset
16
17 #include <openssl/openssl_typ.h>
18
19 #if OPENSSL_VERSION_NUMBER < 0x30000000
20 #define OPENSSL_DLSYM_CALL(handle, func) dlsym(handle, func)
21 #else
22 #define __USE_GNU
23 #define OPENSSL_DLSYM_CALL(handle, func) dlvsym(handle, func, "OPENSSL_3.0.0")
24 #endif
25
26 #include <dlfcn.h>
27
28 #define unlikely(x) __builtin_expect(!(x), 0)
29 #define DEFINEFUNC(ret, func, args, argscall) \
30     typedef ret(*_goboringcrypto_PTR_##func) args; \
31     static _goboringcrypto_PTR_##func _g_##func = 0; \
32     static inline ret _goboringcrypto_##func args \
33     { \
34         if (unlikely(!_g_##func)) \
35         { \
36             _g_##func = OPENSSL_DLSYM_CALL(handle, #func); \
37         } \
38         return _g_##func argscall; \
39     }
40
```

我们用 `objdump` 命令将二进制反汇编后，可以看到相关的CGO函数的定义：

```
nbapp642:/home/maintenance # objdump -S /tmp/otpm >/tmpotpm.S # dump assembly code
```

```
0000000007141e0 <_cgo_0284165e5dc5_Cfunc__goboringcrypto_EVP_aes_256_ecb>:
7141e0: 41 54                push    %r12
7141e2: 55                  push    %rbp
7141e3: 53                  push    %rbx
7141e4: 48 89 fb            mov     %rdi,%rbx
7141e7: e8 94 e3 d5 ff      callq   472580 <_cgo_topofstack>
7141ec: 49 89 c4            mov     %rax,%r12
7141ef: 48 8b 05 6a d9 3f 00 mov     0x3fd96a(%rip),%rax    # b11b60 <_g_EVP_aes_256_ecb>
7141f6: 48 85 c0            test    %rax,%rax
7141f9: 74 1d              je      714218 <_cgo_0284165e5dc5_Cfunc__goboringcrypto_EVP_aes_256_ecb+0x38>
7141fb: ff d0              callq   *%rax
7141fd: 48 89 c5            mov     %rax,%rbp
714200: e8 7b e3 d5 ff      callq   472580 <_cgo_topofstack>
714205: 4c 29 e0            sub     %r12,%rax
714208: 48 89 2c 03         mov     %rbp,(%rbx,%rax,1)
71420c: 5b                  pop     %rbx
71420d: 5d                  pop     %rbp
71420e: 41 5c              pop     %r12
714210: c3                  retq
714211: 0f 1f 80 00 00 00 00 nopl    0x0(%rax)
714218: 48 8d 35 89 08 0a 00 lea     0xa0889(%rip),%rsi    # 7b4aa8 <_cgo_yield+0x100>
71421f: 31 ff              xor     %edi,%edi
714221: e8 ba 0d cf ff      callq   404fe0 <dlsym@plt>
714226: 48 89 05 33 d9 3f 00 mov     %rax,0x3fd933(%rip)    # b11b60 <_g_EVP_aes_256_ecb>
71422d: eb cc              jmp     7141fb <_cgo_0284165e5dc5_Cfunc__goboringcrypto_EVP_aes_256_ecb+0x1b>
71422f: 90                  nop
```

Framework

Web

- [mux](#) //http router

Plugins

- [go-plugins-helpers](#)

References

- 协程模型

- [Golang源码探索\(二\) 协程的实现原理](#)
- [Go 语言调度（一）：系统调度](#)
- [The Go scheduler](#)
- 内存管理
 - [Golang源码探索\(三\) GC的实现原理](#)
 - [Go 语言内存管理（一）：系统内存管理](#)
 - [Go 语言内存管理（二）：Go 内存管理](#)
 - [Go 语言内存管理（三）：逃逸分析](#)
 - [Go 语言内存管理（四）：垃圾回收](#)
 - [Go内存分配机制总结](#)
 - [高性能 Go 服务的内存优化\(译\)](#)
- 并发编程
 - [Handling 1 Million Requests per Minute with Go](#)
 - [Go Concurrency Patterns](#)
- Documentation & Books
 - <http://legendtkl.com/categories/golang/>
 - [Go语言设计与实现](#)
- Go最佳实践 & 编程规范
 - [Uber Go语言编程规范](#)
 - [Twelve Go Best Practices - Google](#)
 - [Effective Go](#)
 - [Golang Best Practices \(Top 20\)](#)
 - [Go Style Best Practices](#)
- 面试经验
 - [Go 程序员面试笔试宝典](#)