0x01 源码研究背景

最近想自己搞一个纯自动化的平台，直接一键挖洞，最实际的方法其实就是直接拼接市面上成熟的工具，然后再根据一些自己想要的功能进行修改。

通过对很多工具的调研，发现扫描这块，kscan比较好用，理由如下：

一：调用了gonmap的指纹识别库，扫描结果可以返回端口的banner，这一点很多扫描器不具备，例如fscan。

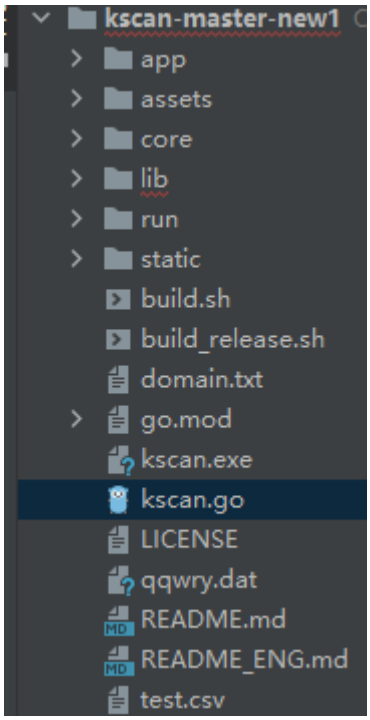
二：kscan的框架较为完善，同时整体代码的风格不错，可扩展性强，二开改东西什么的很方便。

三：里面集成了一个我比较喜欢的去除CDN功能，可以按照自己的喜好再魔改一下，增加一些定制化的东西

0x02 kscan源码分析-关于程序架构解析及端口扫描和fscan的对比

首先我直接从github拉下来最新的kscan

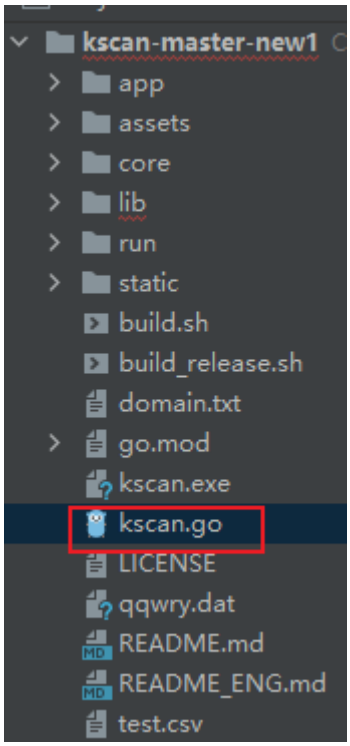
https://github.com/lcvvvv/kscan



上面是大概的框架

先从入口文件开始看起

也就是这个kscan.go



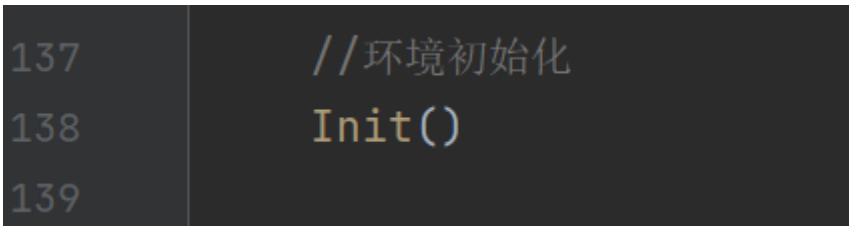
然后看到kscan.go的main函数

```go
func main() {
    startTime := time.Now()

    //环境初始化
    Init()

    //下载qqwry
    // 就是那个能够根据ip对应其地理信息的东西
    if app.Setting.DownloadQQwry == true {
        slog.Println(slog.INFO, "", "现在开始下载最新qqwry，请耐心等待！")
        err := cdn.DownloadQQWry() // 直接调用下载模块开始下载
        if err != nil {
            slog.Println(slog.WARN, "", "纯真IP库下载失败，请手动下载解压后保存到kscan同一目录")
            slog.Println(slog.WARN, "", "下载链接： https://qqwry.mirror.noc.one/qqwry.rar")
            slog.Println(slog.WARN, err)
        }
        slog.Println(slog.INFO, "", "qqwry.dat下载成功！")
        os.Exit( code: 0)
    }

    //spy模块启动
    if app.Setting.Spy != "None" { // 外部加载Spy参数
        // 刚刚拥有自了 这个keyword就是通过 spy 然后调用改参数的方法传入 然后赋值到这个keyword里面
        spy.Keyword = app.Setting.Spy // 把从设置中读取的参数加载过来 尤其是这个keyword
        spy.Scan = app.Setting.Scan // 目前就参数来看 是个布尔值的Scan 具体是什么还不清楚
        spy.Start() // 这个就很好理解了 开始spy
        if spy.Scan { // 这句话的意思就是 如果开启了Scan参数传入 就把spy.Target的值取出去
            app.Setting.Target = spy.Target // 这个target其实真不知道是啥 但是就是扫描的目标 默认的参数在spy.go里面有
        }
    }

    //fofa模块初始化
    if len(app.Setting.Fofa) > 0 {
        InitFofa()
        fofa.Run()
        if app.Setting.Check == false && app.Setting.Scan == false {
            slog.Println(slog.WARN, "", "可以使用--check参数对fofa扫描结果进行存活性及指纹探测，也可以使用--scan参数对fofa扫描结果进行端口扫描")
        }
        if app.Setting.Check == true {
            app.Setting.Target = fofa.GetUrlTarget()
            slog.Println(slog.WARN, "", "check参数已启用，现在将对fofa扫描结果进行存活性及指纹探测")
        }
        if app.Setting.Scan == true {
            app.Setting.Target = fofa.GetHostTarget()
            slog.Println(slog.WARN, "", "scan参数已启用，现在将对fofa扫描结果进行端口扫描及指纹探测")
        }
    }
    //Hydra模块初始化
    if app.Setting.Hydra == true {
        slog.Println(slog.INFO, "", "hydra模块已开启，开始进行暴力破解任务")
        slog.Println(slog.WARN, "", "当前已开启的hydra模块为：", misc.Intersection(hydra.ProtocolList, app.Setting.HydraMod))
        //加载Hydra模块自定义字典
        hydra.InitCustomAuthMap(app.Setting.HydraUser, app.Setting.HydraPass)
    }
    //kscan模块启动
    if len(app.Setting.Target) > 0 { // 这里的意思就是 有target才继续下面的扫描模块
        // 扫描模块初始化
        // 主要做了三件事 指纹 日志 CDN检测的初始化
        InitKscan()
        //开始扫描
        run.Start()
```

这里缩到最小截了一张图，下面逐行来解析代码

其实有的我在注释中已经标注了，如果看不清楚就放大看看。

首先是一个初始化



```go
func Init() {
    // 我理解就是设置各个参数
    app.Args.SetLogo(logo)
    app.Args.SetUsage(usage)
    app.Args.SetHelp(help)
    app.Args.SetSyntax(syntax)
    //参数初始化
    app.Args.Parse() // 我理解这个Parse就是把当初带参数弄进来的数值直接塞进
    //基础输出初始化
    stdio.SetEncoding(app.Args.Encoding) // 这里主要是设置编码
    //参数合法性校验
    app.Args.CheckArgs() // 这里主要是看看参数有没有问题，容错检查
    //日志初始化
    if app.Args.Debug { // 设置日志输出等级
        slog.SetLevel(slog.DEBUG)
    } else {
        slog.SetLevel(slog.INFO)
    }
    //color包初始化
    if os.Getenv( key: "KSCAN_COLOR") == "TRUE" {
        color.Enabled()
    }
    if app.Args.CloseColor == true {
        color.Disabled()
    }
    //pool包初始化
    pool.SetLogger(slog.Debug()) // 这里初始化就是输出日志而已
```

这个init做了最初始的一些设置，具体的设置在注释中标注了，我在作者原有的注释上又添加了一些。

然后再跟一个下载qqwry模块

```go
    //下载qqwry
    // 就是那个能够根据ip对应其他详细信息的东西
    if app.Setting.DownloadQQwry == true {
        slog.Println(slog.INFO, s...: "现在开始下载最新qqwry，请耐心等待！")
        err := cdn.DownloadQQWry() // 直接调用下载模块开始下载
        if err != nil {
            slog.Println(slog.WARN, s...: "纯真IP库下载失败，请手动下载解压后保存到kscan同一目录")
            slog.Println(slog.WARN, s...: "下载链接： https://qqwry.mirror.noc.one/qqwry.rar")
            slog.Println(slog.WARN, err)
        }
        slog.Println(slog.INFO, s...: "qqwry.dat下载成功！")
        os.Exit( code: 0)
    }
```

这个qqwry类似于一个大库，可以获取对应ip的一些详细信息。

然后再是spy模块，是kscan的一个功能。

```
//spy模块启动
if app.Setting.Spy != "None" { // 外部加载Spy参数
    // 刚刚搞明白了 这个keyword就是通过 spy 然后后面接参数的方法传入 然后赋值到这个keyword里面
    spy.Keyword = app.Setting.Spy // 把从设置中读取的参数加载过来 尤其是这个keyword
    spy.Scan = app.Setting.Scan // 目前就参数来看 是个布尔值的Scan 具体是什么还不清楚
    spy.Start() // 这个就很好理解了 开始spy
    if spy.Scan { // 这句话的意思就是 如果开启了Scan参数传入 就把spy.Target的值赋出去
        app.Setting.Target = spy.Target // 这个target其实真不知道是啥 感觉就是扫描的目标 默认的参数在spy
    }
}
```

这里直接运行可以看到作者的help信息

```
    _   __
   /#| /#/
   |#|/#/  _____  _____      *     _   _
   |#.#/  /Edge/ /Forum\    /#\    /#\ /#\
   |##|  |#|____ |#|      /Kv2\   |##\|#|
   |#.#\  \r0cky\|#|     /#/_\#\  |#.#.#|
   |#|\#\/\___|#||#|____/#/Rui\#\|#|\##|
   \#| \#\lcvvvv/ \aels/#/ v1.87#\#/ \#/

usage: kscan [-h,--help,--fofa-syntax] (-t,--target,-f,--fofa,--spy]) [options] [hydra options] [fofa options]


optional arguments:
  -h , --help     show this help message and exit
  -f , --fofa     从fofa获取检测对象，需提前配置环境变量:FOFA_EMAIL、FOFA_KEY
  -t , --target   指定探测对象：
                  IP地址：114.114.114.114
                  IP地址段：114.114.114.114/24,不建议子网掩码小于12
                  IP地址段：114.114.114.114-115.115.115.115
                  URL地址：https://www.baidu.com
                  文件地址：file:/tmp/target.txt
                  剪切板：paste or clipboard
  --spy           网段探测模式，此模式下将自动探测主机可达的内网网段可接收参数为：
                  (空)、192、10、172、all、指定IP地址(将探测该IP地址B段存活网关)
```

其实就是网段识别，但是我没怎么用过kscan的这个功能，因为这个功能更多用于内网，我喜欢用sanfor原来做的那个netspy，感觉更轻。

然后下面分别是其他模块

```
//fofa模块初始化
if len(app.Setting.Fofa) > 0 {
    InitFofa()
    fofa.Run()
    if app.Setting.Check == false && app.Setting.Scan == false {
        slog.Println(slog.WARN, s...: "可以使用--check参数对fofa扫描结果进行存活性及指纹探测，也可以使用--scan参数对fofa扫描结果进行端口扫描")
    }
    if app.Setting.Check == true {
        app.Setting.Target = fofa.GetUrlTarget()
        slog.Println(slog.WARN, s...: "check参数已启用，现在将对fofa扫描结果进行存活性及指纹探测")
    }
    if app.Setting.Scan == true {
        app.Setting.Target = fofa.GetHostTarget()
        slog.Println(slog.WARN, s...: "scan参数已启用，现在将对fofa扫描结果进行端口扫描及指纹探测")
    }
}
//Hydra模块初始化
if app.Setting.Hydra == true {
    slog.Println(slog.INFO, s...: "hydra模块已开启，开始监听暴力破解任务")
    slog.Println(slog.WARN, s...: "当前已开启的hydra模块为: ", misc.Intersection(hydra.ProtocolList, app.Setting.HydraMod))
    //加载Hydra模块自定义字典
    hydra.InitCustomAuthMap(app.Setting.HydraUser, app.Setting.HydraPass)
}
//kscan模块启动
if len(app.Setting.Target) > 0 { // 这里的意思就是 有target才继续下面的扫描模块
    // 扫描模块初始化
    // 主要做了三件事 指纹 日志 CDN检测的初始化
    InitKscan()
    //开始扫描
    run.Start()
}
//计算程序运行时间
elapsed := time.Since(startTime)
slog.Printf(slog.INFO, format: "程序执行总时长为: [%s]", elapsed.String())
slog.Println(slog.INFO, s...: "若有问题欢迎来我的Github提交Bug[https://github.com/lcvvvv/kscan/]")
}
```

可以从注释中看到，为fofa，hydra，kscan本体模块。

那么整体程序的基本main函数逻辑现在就可以梳理下来了。

初始化做一些基本的工作，例如传参，颜色设置之类的-->下载qqwry这个ip库-->如果有spy参数传进来了就启动spy模块进行网段探测-->初始化fofa模块-->初始化hydra爆破模块-->开始kscan的扫描模块，进行ip，port之类的扫描。

ok，基本逻辑梳理完毕了，这里我是主要研究他的扫描机制，同时想加上我自己定制化的东西，所以主要就他的扫描模块来做解析。

也就是下面这块

```
    //kscan模块启动
    if len(app.Setting.Target) > 0 { // 这里的意思就是 有target才继续下面的扫描模块
        // 扫描模块初始化
        // 主要做了三件事 指纹 日志 CDN检测的初始化
        InitKscan()
        //开始扫描
        run.Start()
    }
```

这里就三句话，很简单，那么就一句一句的看。

首先是这句

```
    //kscan模块启动
    if len(app.Setting.Target) > 0 { // 这里的意思就是 有target才继续下面的扫描模块
```

这里要Target>0函数才能往下走，那么Target是什么。

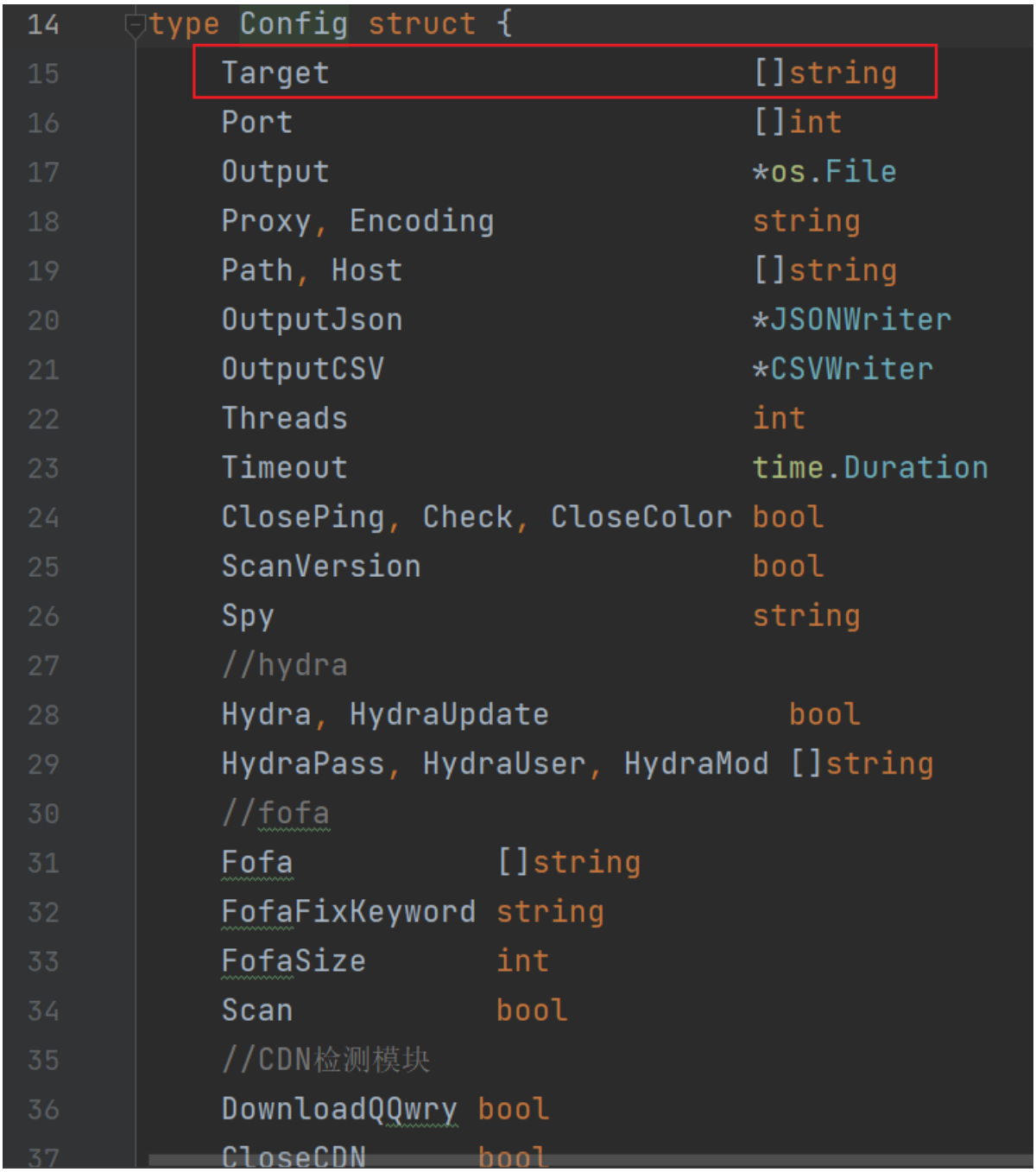
这里跟进

```
var Setting = New()
```

```
// new完之后则返回一个config类型的结构体，然后下放一些参数进来
func New() Config {
    return Config{
        Target:   []string{}, // 花括号指代了包含元素，这里为空代表没有包含任何元素
        Path:     []string{"/"}, // 这里表示包含了斜杠 / 这个元素
        Port:     []int{},
        Output:   nil,
        Proxy:    "",
        Host:     []string{},
        Threads:  800,
        Timeout:  0,
        Encoding: "utf-8",
    }
}
```

这里就找到了Target，但是没找到传参入口。

进一步跟进Config

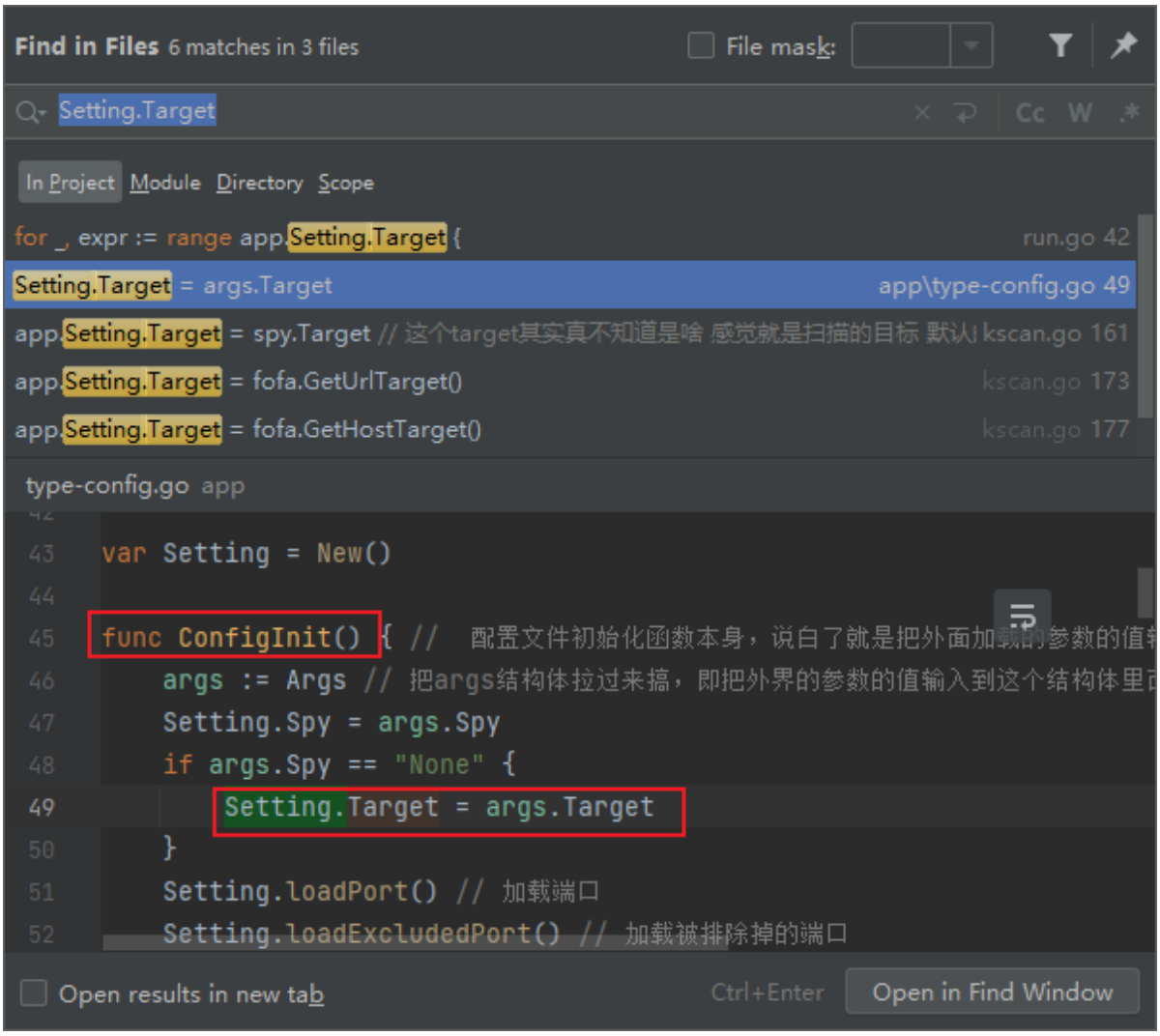


```
type Config struct {
    Target                          []string
    Port                            []int
    Output                          *os.File
    Proxy, Encoding                 string
    Path, Host                      []string
    OutputJson                      *JSONWriter
    OutputCSV                       *CSVWriter
    Threads                         int
    Timeout                         time.Duration
    ClosePing, Check, CloseColor    bool
    ScanVersion                     bool
    Spy                             string
    //hydra
    Hydra, HydraUpdate                bool
    HydraPass, HydraUser, HydraMod  []string
    //fofa
    Fofa            []string
    FofaFixKeyword  string
    FofaSize        int
    Scan            bool
    //CDN检测模块
    DownloadQQwry  bool
    CloseCDN       bool
```

发现最后落脚到了一个结构体

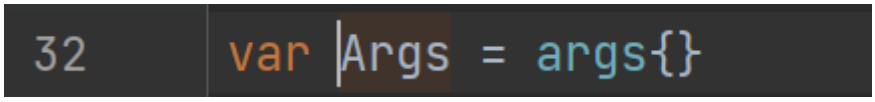
其实到这里线索就断了，需要回归到哪里调用了这个Target并且赋值给他了。

因此这里全局搜索Setting.Target



这里可以看到从args.Target赋值给了Setting.Target

进一步跟进



```
type args struct { // 定义一个大的结构体，说明用法
    USAGE, HELP, LOGO, SYNTAX string

    Help, Debug, ClosePing, Check, CloseColor, Scan bool
    ScanVersion, DownloadQQwry, CloseCDN            bool
    Output, Proxy, Encoding                         string
    Port, ExcludedPort                              []int
    Path, Host, Target                              []string
    OutputJson, OutputCSV                           string
    Spy, Touch                                      string
    Top, Threads, Timeout                           int
    //hydra模块
    Hydra, HydraUpdate             bool
    HydraUser, HydraPass, HydraMod []string
    //fofa模块
    Fofa                     []string
    FofaField, FofaFixKeyword string
    FofaSize                 int
    FofaSyntax               bool
    //输出修饰
    Match, NotMatch string
}
```

找到了args结构体的原型

然后找到传值的地方

```
// Parse 初始化参数
func (o *args) Parse() {
    //自定义Usage
    sflag.SetUsage(o.LOGO)
    //定义参数
    o.define()
    //实例化参数值
    sflag.Parse()
    //输出LOGO
    o.printBanner()
}

//定义参数
func (o *args) define() {
    sflag.BoolVar(&o.Help,  name: "h",  value: false) // 反正就是这么用的，原理不用深究
    sflag.BoolVar(&o.Help,  name: "help",  value: false)
    sflag.BoolVar(&o.Debug,  name: "debug",  value: false)
    sflag.BoolVar(&o.Debug,  name: "d",  value: false)
    //spy模块
    sflag.AutoVarString(&o.Spy,  name: "spy",  value: "None")
    //hydra模块
    sflag.BoolVar(&o.Hydra,  name: "hydra",  value: false)
    sflag.BoolVar(&o.HydraUpdate,  name: "hydra-update",  value: false)
```

```
58      sflag.StringSpliceVar(&o.HydraPass,  name: "hydra-pass")
59      sflag.StringSpliceVar(&o.HydraMod,  name: "hydra-mod")
60      //fofa模块
61      sflag.StringSpliceVar(&o.Fofa,  name: "fofa")
62      sflag.StringSpliceVar(&o.Fofa,  name: "f")
63      sflag.StringVar(&o.FofaField,  name: "fofa-field",  value: "")
64      sflag.StringVar(&o.FofaFixKeyword,  name: "fofa-fix-keyword",  value: "")
65      sflag.IntVar(&o.FofaSize,  name: "fofa-size",  value: 100)
66      sflag.BoolVar(&o.FofaSyntax,  name: "fofa-syntax",  value: false)
67      sflag.BoolVar(&o.Scan,  name: "scan",  value: false)
68      //kscan模块
69      sflag.StringSpliceVar(&o.Target,  name: "target")
70      sflag.StringSpliceVar(&o.Target,  name: "t")
71      sflag.IntSpliceVar(&o.Port,  name: "port")
72      sflag.IntSpliceVar(&o.Port,  name: "p")
73      sflag.IntSpliceVar(&o.ExcludedPort,  name: "eP")
74      sflag.IntSpliceVar(&o.ExcludedPort,  name: "excluded-port")
75      sflag.StringSpliceVar(&o.Path,  name: "path")
76      sflag.StringSpliceVar(&o.Host,  name: "host")
77      sflag.StringVar(&o.Proxy,  name: "proxy",  value: "")
```

上面很清楚的看到，就是通过调用sflag模块的Target参数key来获取值，然后最后传入到函数中，就得到了Target的Value值。

这个函数也表述的很清楚了

```
34  // Parse 初始化参数
35  func (o *args) Parse() {
36      //自定义Usage
37      sflag.SetUsage(o.LOGO)
38      //定义参数
39      o.define()
40      //实例化参数值
41      sflag.Parse()
42      //输出LOGO
43      o.printBanner()
44  }
45
```

这个Parse的调用位置往下看

```go
func main() {
    startTime := time.Now()

    //环境初始化
    Init()

    //下载qqwry
    // 就是那个能够根据ip对应其他详细信息的东西
    if app.Setting.DownloadQQwry == true {
        slog.Println(slog.INFO, s...: "现在开始下载最新qqwry，请耐心等待！")
        err := cdn.DownloadQQWry() // 直接调用下载模块开始下载
        if err != nil {
            slog.Println(slog.WARN, s...: "纯真IP库下载失败，请手动下载解压后保存到kscan同一目录")
            slog.Println(slog.WARN, s...: "下载链接： https://qqwry.mirror.noc.one/qqwry.rar")
            slog.Println(slog.WARN, err)
        }
        slog.Println(slog.INFO, s...: "qqwry.dat下载成功！")
        os.Exit( code: 0)
    }
```

```go
func Init() {
    // 我理解就是设置各个参数
    app.Args.SetLogo(logo)
    app.Args.SetUsage(usage)
    app.Args.SetHelp(help)
    app.Args.SetSyntax(syntax)
    //参数初始化
    app.Args.Parse() // 我理解这个Parse就是把当初带参数弄进来的数值直接塞进程序
    //基础输出初始化
    stdio.SetEncoding(app.Args.Encoding) // 这里主要是设置编码
    //参数合法性校验
    app.Args.CheckArgs() // 这里主要是看看参数有没有问题，容错检查
    //日志初始化
    if app.Args.Debug { // 设置日志输出等级
        slog.SetLevel(slog.DEBUG)
    } else {
        slog.SetLevel(slog.INFO)
    }
    //color包初始化
```

这里我在注释中表述清楚了，就不再赘述了。

ok，那么再跳回来，还是到这句话。

```go
    //kscan模块启动
    if len(app.Setting.Target) > 0 { // 这里的意思就是 有target才继续下面的扫描模块
        // 扫描模块初始化
        // 主要做了三件事 指纹 日志 CDN检测的初始化
        InitKscan()
        //开始扫描
        run.Start()
    }
```

现在理解就是，target一旦有了，也就是扫描的目标有了，就往下走。

体现在应用中如下

```
>kscan.exe -t https://baidu.com

    _   __
   /#| /#/
   |#|/#/  _____  _____      *     _   _
   |#.#/  /Edge/ /Forum\    /#\   /#\ /#\
   |##|  |#|____ |#|      /Kv2\  |##\|#|
   |#.#\  \r0cky\|#|     /#/_\#\ |#.#.#|
   |#|\#\/\___|#||#|____/#/Rui\#\|#|\##|
   \#| \#\lcvvvv/ \aels/#/ v1.87#\#/ \#/

Tips: 可以使用-Pn参数关闭存活性探测，将会将所有主机视为存活
[+]2023/10/09 15:00:45 当前环境为: windows，输出编码为: utf-8
[+]2023/10/09 15:00:45 成功加载HTTP指纹:[24758]条
[+]2023/10/09 15:00:45 成功加载NMAP探针:[150]个,指纹[11916]条
[+]2023/10/09 15:00:45 检测到qqwry.dat,将自动启动CDN检测功能，可使用-Dn参数关闭该功能
[+]2023/10/09 15:00:46 Domain、IP、Port、URL、Hydra引擎已准备就绪
[+]2023/10/09 15:00:46 所有扫描任务已下发完毕
http://baidu.com              百度一下，你就知道          Digest:全球领先的中文搜索引擎、致力于让网民更便捷地获取,Length
:400998,FingerPrint:JQuery;HTML5,Port:80,Domain:baidu.com,FoundDomain:om、1.su.bdimg.com、2.su.b
https://baidu.com            百度一下，你就知道          FingerPrint:JQuery;HTML5,FoundDomain:.com、smartapps.cn、bdtjr
c,Domain:baidu.com,Digest:全球领先的中文搜索引擎、致力于让网民更便捷地获取,Port:443,Length:401004
https://baidu.com            百度一下，你就知道          Domain:baidu.com,FingerPrint:JQuery;HTML5,Length:401044,FoundD
omain:、cdn01.baidu-img.cn、e.in,Port:443,Digest:"全球领先的中文搜索引擎、致力于让网民更便捷地获
```

现在再看第二句话

```go
//kscan模块启动
if len(app.Setting.Target) > 0 { // 这里的意思就是 有target才继续下面的扫描模块
    // 扫描模块初始化
    // 主要做了三件事 指纹 日志 CDN检测的初始化
    InitKscan()
    //开始扫描
    run.Start()
}
```

这个工具该说不说，写得比fscan要复杂很多，可以接着往后看。

跟进这个InitKscan函数

```go
250  func InitKscan() {
251      //HTTP指纹库初始化
252      fs, _ := fingerprintEmbed.Open(fingerprintPath)
253      if n, err := appfinger.InitDatabaseFS(fs); err != nil {
254          slog.Println(slog.ERROR, s...: "指纹库加载失败，请检查【fingerprint.txt】文件", err)
255      } else {
256          slog.Printf(slog.INFO, format: "成功加载HTTP指纹:[%d]条", n)
257      }
258      //超时及日志配置
259      gonmap.SetLogger(slog.Debug())
260      slog.Printf(slog.INFO, format: "成功加载NMAP探针:[%d]个,指纹[%d]条", gonmap.UsedProbesCount, gonmap.UsedMatchCount)
261      //CDN检测初始化
262      if app.Setting.CloseCDN == false {
263          if _, err := os.Lstat(qqwryPath); os.IsNotExist(err) == true {
264              slog.Printf(slog.WARN, format: "未检测到qqwry.dat,将关闭CDN检测功能，如需开启，请执行kscan --download-qqwry下载该
265              app.Setting.CloseCDN = true
266          } else {
267              slog.Printf(slog.INFO, format: "检测到qqwry.dat,将自动启动CDN检测功能，可使用-Dn参数关闭该功能")
268              scanner.CDNCheck = true
269              cdn.Init(qqwryPath)
270          }
271      }
272  }
```

根据注释，可以看到这个kscan初始化函数做了三件事，一个是http指纹初始化，一个是超时以及日志配置，一个是CDN检测初始化。

这里先留个印象，同时看到CDN检测模块这里。

```
if app.Setting.CloseCDN == false {
    if _, err := os.Lstat(qqwryPath); os.IsNotExist(err) == true {
        slog.Printf(slog.WARN, format: "未检测到qqwry.dat,将关闭CDN检测功能，如需开启，请执行kscan --download-
        app.Setting.CloseCDN = true
    } else {
        slog.Printf(slog.INFO, format: "检测到qqwry.dat,将自动启动CDN检测功能，可使用-Dn参数关闭该功能")
        scanner.CDNCheck = true
        cdn.Init(qqwryPath)
```

这里也是需要CDNChedk这个值为true，才会启动检测。

然后看回原文的第三句话

```
188         //kscan模块启动
189         if len(app.Setting.Target) > 0 { // 这里的意思就是 有target才继续下面的扫描模块
190             // 扫描模块初始化
191             // 主要做了三件事 指纹 日志 CDN检测的初始化
192             InitKscan()
193             //开始扫描
194             run.Start()
195         }
```

跟进进去看看

```
28  func Start() {
29      //启用看门狗函数定时输出负载情况
30      go watchDog()
31      //下发扫描任务
32      var wg = &sync.WaitGroup{} // 并发控制机制 等待几个协程都执行完毕，再往下执行
33      wg.Add( delta: 5) // 塞入五个协程
34      DomainScanner = generateDomainScanner(wg) // 看起来是进行域名扫描 这里之所以要把wg传进去的原因是
35      IPScanner = generateIPScanner(wg)
36      PortScanner = generatePortScanner(wg) // 对端口进行扫描，然后获取端口信息，具体调用的是gonamp
37      URLScanner = generateURLScanner(wg)
38      HydraScanner = generateHydraScanner(wg)
39      //扫描器进入监听状态
40      start()
41      //开始分发扫描任务
42      for _, expr := range app.Setting.Target {
43          pushTarget(expr)
44      }
45      slog.Println(slog.INFO, s...: "所有扫描任务已下发完毕")
46      //根据扫描情况，关闭scanner
47      go stop() // 凡是用了go什么的 都是开启一个协程 相当于新弄一个任务 这里是直接停止
48      wg.Wait() // 等待所有线程同步 就在这里wait
49  }
50
```

这里作者用了golang的协程来调任务，代码写得挺工整的。

简单解释下，就是作者弄了

```
DomainScanner = generateDomainScanner(wg)
IPScanner = generateIPScanner(wg)
PortScanner = generatePortScanner(wg) //
URLScanner = generateURLScanner(wg)
HydraScanner = generateHydraScanner(wg)
```

这么五个任务，塞到协程里面。

然后喊一声开始

```
    //扫描器进入监听状态
    start()
    //开始分发扫描任务
    for _, expr := range app.Setting.Target {
        pushTarget(expr)
    }
    slog.Println(slog.INFO, s...: "所有扫描任务已下发完毕")
    //根据扫描情况，关闭scanner
    go stop() // 凡是用了go什么的 都是开启一个协程 相当于新弄一个任务 这里是直接停止
    wg.Wait() // 等待所有线程同步 就在这里wait
}
```

各个小任务就开始运行

这里可以跟进这个start()看看

```
// 这里调用start就是用golang的协程 go一下 相当于拉一个协程出来启动 然后运行
func start() {
    go DomainScanner.Start()
    go IPScanner.Start()
    go PortScanner.Start()
    go URLScanner.Start()
    go HydraScanner.Start()
    time.Sleep(time.Second * 1)
    slog.Println(slog.INFO, s...: "Domain、IP、Port、URL、Hydra引擎已准备就绪")
}
```

主打的就是一个开冲，再跟进到这里的Start看看

```
func (c *client) Start() {
    c.pool.Run()
}
```

然后跟进Run

```go
//执行工作池当中的任务
func (p *Pool) Run() {
    p.Done = false
    //只启动有限大小的协程，协程的数量不可以超过工作池设定的数量，防止计算资源崩溃
    for i := 0; i < p.threads; i++ {
        p.wg.Add( delta: 1)
        time.Sleep(p.Interval)
        go p.work()
        if p.Done == true {
            break
        }
    }
    p.wg.Wait()
}
```

那么这里整个跟下来就很好理解了。

其实就理解为往线程池里面塞任务，不断的往里面塞就对了。

然后一个任务写成一个模块化的代码，往里塞就完事了。

之所以像fscan还有kscan这种扫描器，都用golang来写，就是利用golang的高并发特性，能开销很小的搞并发。

fscan的扫描代码我也拉出来展示下，他写的简单，但是同样也是调用的golang原生的sync模块：

```go
42
43          //接收结果
44          go func() {
45              for found := range results {
46                  AliveAddress = append(AliveAddress, found)
47                  wg.Done()
48              }
49          }()
50
51          //多线程扫描
52          for i := 0; i < workers; i++ {
53              go func() {
54                  for addr := range Addrs {
55                      PortConnect(addr, results, timeout, &wg)
56                      wg.Done()
57                  }
58              }()
59          }
60
```

这里他的端口扫描就是用PortConnect函数进行端口连接

跟进去看看

```go
func PortConnect(addr Addr, respondingHosts chan<- string, adjustedTimeout int64, wg *sync.WaitGroup
    host, port := addr.ip, addr.port
    conn, err := common.WrapperTcpWithTimeout( network: "tcp4", fmt.Sprintf("#{host}:#{port}"), time.D
    defer func() {
        if conn != nil {
            conn.Close()
        }
    }()
    if err == nil {
        address := host + ":" + strconv.Itoa(port)
        result := fmt.Sprintf("#{address} open")
        common.LogSuccess(result)
        wg.Add( delta: 1)
        respondingHosts <- address
    }
}
```

主打的就是一个连一下

这里直接往里面跟

```go
func WrapperTcpWithTimeout(network, address string, timeout time.Duration) (net.Conn, error)
    d := &net.Dialer{Timeout: timeout}
    return WrapperTCP(network, address, d)
}
```

```go
func WrapperTCP(network, address string,forward * net.Dialer) (net.Conn, error) {
    //get conn
    var conn net.Conn
    if Socks5Proxy == "" {
        var err error
        conn,err = forward.Dial(network, address)
        if err != nil : nil, err
    }else {
        dailer, err := Socks5Dailer(forward)
        if err != nil: nil, err
        conn,err = dailer.Dial(network, address)
        if err != nil : nil, err
    }
    return conn, nil

}
```

```go
func (d *Dialer) Dial(network, address string) (Conn, error) {
    return d.DialContext(context.Background(), network, address)
}
```

基本就很清晰了

Dial是golang自带的一个拨号器，主打一个连接一下。

回头看到fscan中的这段

```go
func PortConnect(addr Addr, respondingHosts chan<- string, adjustedTimeout int64, wg *sync.WaitGroup) {
    host, port := addr.ip, addr.port
    conn, err := common.WrapperTcpWithTimeout( network: "tcp4", fmt.Sprintf("#{host}:#{port}"), time.Duration(adjustedTimeout)*time.Second)
    defer func() {
        if conn != nil {
            conn.Close()
        }
    }()
```

什么意思呢，就是用tcp4协议，然后dial一下这个host:port的网络位置，超时时间是自己设置的时间。

这个就是他的port扫描的原理，很简单。

fscan的多线程体现在

```
    //多线程扫描
    for i := 0; i < workers; i++ {
        go func() {
            for addr := range Addrs {
                PortConnect(addr, results, timeout, &wg)
                wg.Done()
            }
        }()
    }
```

典型的golang协程写法。

ok，这里回归回来，到我们的kscan中。

```
func Start() {
    //启用看门狗函数定时输出负载情况
    go watchDog()
    //下发扫描任务
    var wg = &sync.WaitGroup{} // 并发控制机制 等待几个协程都执行完毕，
    wg.Add( delta: 5) // 塞入五个协程
    DomainScanner = generateDomainScanner(wg) // 看起来是进行域名扫
    IPScanner = generateIPScanner(wg)
    PortScanner = generatePortScanner(wg) // 对端口进行扫描，然后获取
    URLScanner = generateURLScanner(wg)
    HydraScanner = generateHydraScanner(wg)
    //扫描器进入监听状态
    start()
    //开始分发扫描任务
    for _, expr := range app.Setting.Target {
        pushTarget(expr)
    }
    slog.Println(slog.INFO, s...: "所有扫描任务已下发完毕")
    //根据扫描情况，关闭scanner
    go stop() // 凡是用了go什么的 都是开启一个协程 相当于新弄一个任务 这里
    wg.Wait() // 等待所有线程同步 就在这里wait
}
```

这里依旧利用golang的协程，然后把wg传入一个个工作器中，这里为了和fscan对比端口扫描的方式，就先看看kscan的端口扫描是怎么做的。

以下是跟进过程：

```
    //kscan模块启动
    if len(app.Setting.Target) > 0 { // 这里的意思就是 有target才继续下面的扫描模块
        // 扫描模块初始化
        // 主要做了三件事 指纹 日志 CDN检测的初始化
        InitKscan()
        //开始扫描
        run.Start()
    }
    //计算程序运行时间
    elapsed := time.Since(startTime)
    slog.Printf(slog.INFO, format: "程序执行总时长为：[%s]", elapsed.String())
    slog.Println(slog.INFO, s...: "若有问题欢迎来我的Github提交Bug[https://github.com/lcvvvv/kscan/]")
}
```

```
func Start() {
    //启用看门狗函数定时输出负载情况
    go watchDog()
    //下发扫描任务
    var wg = &sync.WaitGroup{} // 并发控制机制 等待几个协程都执行完毕，再往下执行
    wg.Add( delta: 5) // 塞入五个协程
    DomainScanner = generateDomainScanner(wg) // 看起来是进行域名扫描 这里之所以要把wg传进去的原因是
    IPScanner = generateIPScanner(wg)
    PortScanner = generatePortScanner(wg) // 对端口进行扫描，然后获取端口信息，具体调用的是gonamp
    URLScanner = generateURLScanner(wg)
    HydraScanner = generateHydraScanner(wg)
    //扫描器进入监听状态
    start()
    //开始分发扫描任务
    for _, expr := range app.Setting.Target {
        pushTarget(expr)
    }
    slog.Println(slog.INFO, s...: "所有扫描任务已下发完毕")
    //根据扫描情况，关闭scanner
    go stop() // 凡是用了go什么的 都是开启一个协程 相当于新弄一个任务 这里是直接停止
    wg.Wait() // 等待所有线程同步 就在这里wait
```

```
280 func generatePortScanner(wg *sync.WaitGroup) *scanner.PortClient {
281     PortConfig := scanner.DefaultConfig()
282     PortConfig.Threads = app.Setting.Threads
283     PortConfig.Timeout = getTimeout(len(app.Setting.Port))
284     if app.Setting.ScanVersion == true {
285         PortConfig.DeepInspection = true
286     }
287     client := scanner.NewPortScanner(PortConfig) // 里面的client定义了接口函数，外面来实现
288     client.HandlerClosed = func(addr net.IP, port int) {
289         //nothing
290     }
291     client.HandlerOpen = func(addr net.IP, port int) {
292         outputOpenResponse(addr, port)
293     }
294     client.HandlerNotMatched = func(addr net.IP, port int, response string) {
295         outputUnknownResponse(addr, port, response)
296     }
297     client.HandlerMatched = func(addr net.IP, port int, response *gonmap.Response) {
298         // 这里是直接获取了fingerprint的信息，是直接调用gonmap来实现的，不是作者自己实现的。
299         URLRaw := fmt.Sprintf( format: "%s://%s:%d", response.FingerPrint.Service, addr.String(), port)
300         URL, _ := url.Parse(URLRaw)
301         if appfinger.SupportCheck(URL.Scheme) == true { // 推进去进行扫描
302             pushURLTarget(URL, response)
303             return
304         }
305         outputNmapFinger(URL, response)
306         if app.Setting.Hydra == true {
```

```go
func NewPortScanner(config *Config) *PortClient {
    var client = &PortClient{
        client:            newConfig(config, config.Threads),
        HandlerClosed:     func(addr net.IP, port int) {},
        HandlerOpen:       func(addr net.IP, port int) {},
        HandlerNotMatched: func(addr net.IP, port int, response string) {},
        HandlerMatched:    func(addr net.IP, port int, response *gonmap.Response) {},
        HandlerError:      func(addr net.IP, port int, err error) {},
    }
    client.pool.Interval = config.Interval
    client.pool.Function = func(in interface{}) {
        nmap := gonmap.New()
        nmap.SetTimeout(config.Timeout)
        if config.DeepInspection == true {
            nmap.OpenDeepIdentify()
        }
        value := in.(foo1)
        status, response := nmap.ScanTimeout(value.addr.String(), value.num, 100*config.Timeout)
        switch status {
        case gonmap.Closed:
            client.HandlerClosed(value.addr, value.num)
        case gonmap.Open:
            client.HandlerOpen(value.addr, value.num)
        case gonmap.NotMatched:
            client.HandlerNotMatched(value.addr, value.num, response.Raw)
        case gonmap.Matched:
            client.HandlerMatched(value.addr, value.num, response)
        }
```

这里他调用的golang实现的nmap，继续跟进去看看

```go
func (n *Nmap) ScanTimeout(ip string, port int, timeout time.Duration) (status Stat
    ctx, cancel := context.WithTimeout(context.Background(), timeout)
    var resChan = make(chan bool)

    defer func() {
        close(resChan)
        cancel()
    }()

    go func() {
        defer func() {
            if r := recover(); r != nil {
                if fmt.Sprint(r) != "send on closed channel" { r ✷
            }
        }()
        status, response = n.Scan(ip, port)
        resChan <- true
    }()

    select {
    case <-ctx.Done():
        return Closed, response: nil
    case <-resChan:
        return status, response
    }
}
```

*Nmap.ScanTimeout(ip string, port int, timeout time.Duration) (status Status, response *Response)

```go
func (n *Nmap) Scan(ip string, port int) (status Status, response *Response) {
    var probeNames ProbeList
    if n.bypassAllProbePort.exist(port) == true {
        probeNames = append(n.portProbeMap[port], n.allProbeMap...)
    } else {
        probeNames = append(n.allProbeMap, n.portProbeMap[port]...)
    }
    probeNames = append(probeNames, n.sslProbeMap...)
    //探针去重
    probeNames = probeNames.removeDuplicate()

    firstProbe := probeNames[0]
    status, response = n.getRealResponse(ip, port, n.timeout, firstProbe)
    if status == Closed || status == Matched : status, response
    otherProbes := probeNames[1:]
    return n.getRealResponse(ip, port, 2*time.Second, otherProbes...)
}
```

```go
func (n *Nmap) getRealResponse(host string, port int, timeout time.Duration, probes ...string) (status Statu
    status, response = n.getResponseByProbes(host, port, timeout, probes...)
    if status != Matched : status, response
    if response.FingerPrint.Service == "ssl" {
        status, response := n.getResponseBySSLSecondProbes(host, port, timeout)
        if status == Matched : Matched, response
    }
    return status, response
```

```go
Reader Mode
func (n *Nmap) getResponseByProbes(host string, port int, timeout time.Duration, probes ...string) (status
    var responseNotMatch *Response
    for _, requestName := range probes {
        if n.probeUsed.exist(requestName) {
            continue
        }
        n.probeUsed = append(n.probeUsed, requestName)
        p := n.probeNameMap[requestName]

        status, response = n.getResponse(host, port, p.sslports.exist(port), timeout, p)
        //如果端口未开放，则等待10s后重新连接
        //if b.status == Closed {
        //  time.Sleep(time.Second * 10)
        //  b.Load(n.getResponse(host, port, n.probeNameMap[requestName]))
        //}

        //logger.Printf("Target:%s:%d,Probe:%s,Status:%v", host, port, requestName, status)

        if status == Closed || status == Matched {
            responseNotMatch = nil
            break
        }
        if status == NotMatched {
            responseNotMatch = response
        }
    }
    if responseNotMatch != nil {
        response = responseNotMatch
```

```go
func (n *Nmap) getResponse(host string, port int, tls bool, timeout time.Duration, p *probe) (Status, *Respo
    if port == 53 {
        if DnsScan(host, port) { Matched, &dnsResponse } else { Closed, nil }
    }
    text, tls, err := p.scan(host, port, tls, timeout, size: 10240)
    if err != nil {
        if strings.Contains(err.Error(), substr: "STEP1") { Closed, nil }
        if strings.Contains(err.Error(), substr: "STEP2") { Closed, nil }
        if p.protocol == "UDP" && strings.Contains(err.Error(), substr: "refused") { Closed, nil }
        return Open, nil
    }

    response := &Response{
        Raw:         text,
        TLS:         tls,
        FingerPrint: &FingerPrint{},
    }
```

```go
func (p *probe) scan(host string, port int, tls bool, timeout time.Duration, size int) (string, b
    uri := fmt.Sprintf("#{host}:#{port}")

    sendRaw := strings.Replace(p.sendRaw, old: "{Host}", fmt.Sprintf("#{host}:#{port}"), n: -1)

    text, err := simplenet.Send(p.protocol, tls, uri, sendRaw, timeout, size)
    if err == nil { text, tls, nil }
    if strings.Contains(err.Error(), substr: "STEP1") && tls == true {
        text, err := simplenet.Send(p.protocol, tls: false, uri, p.sendRaw, timeout, size)
        return text, false, err
    }
    return text, tls, err
}
```

```go
func Send(protocol string, tls bool, netloc string, data string, duration time.Duration, size int) (string, error) {
    if tls {
        return tlsSend(protocol, netloc, data, duration, size)
    } else {
        return tcpSend(protocol, netloc, data, duration, size)
    }
}
```

到这里就分为了两个板块进行区分，一个是带ssl的，一个不带，先看不带的tcpSend

```go
func tcpSend(protocol string, netloc string, data string, duration time.Duration, size int) (string, error) {
    protocol = strings.ToLower(protocol)
    conn, err := net.DialTimeout(protocol, netloc, duration)
    if err != nil { "", errors.New(err.Error() + " STEP1:CONNECT") }
    defer conn.Close()
    _, err = conn.Write([]byte(data))
    if err != nil { "", errors.New(err.Error() + " STEP2:WRITE") }
    //读取数据
    var buf []byte              // big buffer
    var tmp = make([]byte, 256) // using small tmo buffer for demonstrating
    var length int
```

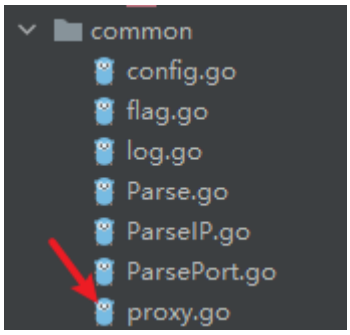
```go
// parameters.
func DialTimeout(network, address string, timeout time.Duration) (Conn, error) {
    d := Dialer{Timeout: timeout}
    return d.Dial(network, address)
}
```

```go
func (d *Dialer) Dial(network, address string) (Conn, error) {
    return d.DialContext(context.Background(), network, address)
}
```

到这里就不再继续跟了

整个跟下来，发现其实最终调用的还是golang的dial模块，和fscan没有区别。

为什么说fscan写得简单，是因为人家直接就通过调用他自己写得一个工具模块



proxy.go中的WrapperTcpWithTimeout直接调用了这个Dialer

```
11
12  func WrapperTcpWithTimeout(network, address string, timeout time.Duration) (net.Conn, error)
13      d := &net.Dialer{Timeout: timeout}
14      return WrapperTCP(network, address, d)
15  }
```

但是kscan整了半天，通过gonmap最后才调到了这个Dial，害我跟了半天，大胆！

ok，那么这么对比下来，其实协程写法一样，用的库最后也是一样的，可以理解为两者在扫描性能上不会有太大差别。

但是值得一提的是kscan作者代码写得感觉非常专业，像是一个专业开发出身来搞安全的人。

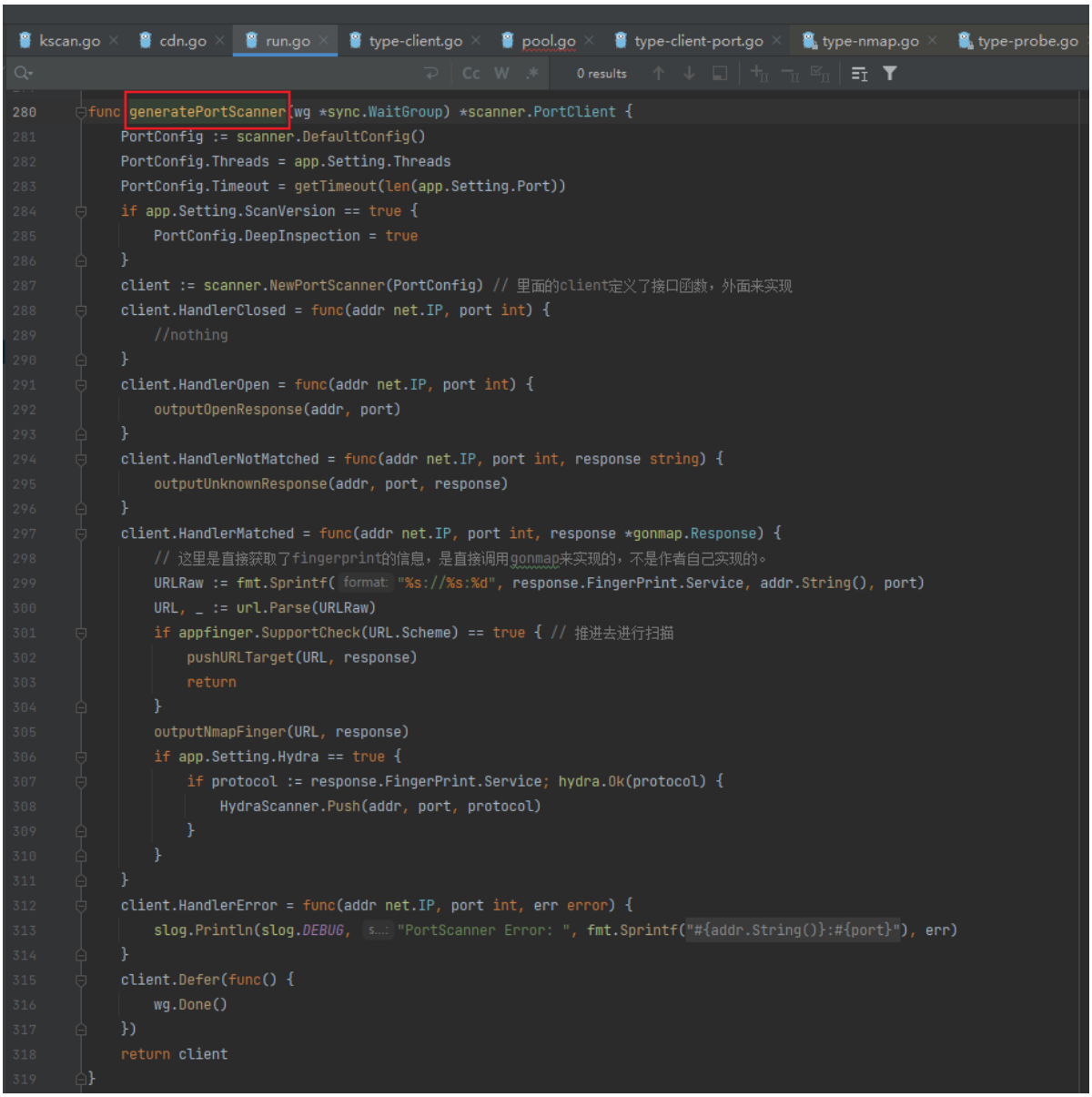
为什么这么说呢，我们再回归到他的那几个模块部分。

kscan.go | cdn.go | run.go | type-client.go | pool.go | type-client-port.go | type-nmap.go | type-probe.go | simplenet.go | dial.go

```
24      "sync"
25      "time"
26  )
27
28  func Start() {
29      //启用看门狗函数定时输出负载情况
30      go watchDog()
31      //下发扫描任务
32      var wg = &sync.WaitGroup{} // 并发控制机制 等待几个协程都执行完毕，再往下执行
33      wg.Add( delta: 5) // 塞入五个协程
34      DomainScanner = generateDomainScanner(wg) // 看起来是进行域名扫描 这里之所以要把wg传进去的原因是因为里面有个done
35      IPScanner = generateIPScanner(wg)
36      PortScanner = generatePortScanner(wg) // 对端口进行扫描，然后获取端口信息，具体调用的是gonamp
37      URLScanner = generateURLScanner(wg)
38      HydraScanner = generateHydraScanner(wg)
39      //扫描器进入监听状态
40      start()
41      //开始分发扫描任务
42      for _, expr := range app.Setting.Target {
43          pushTarget(expr)
44      }
45      slog.Println(slog.INFO, s...: "所有扫描任务已下发完毕")
46      //根据扫描情况，关闭scanner
47      go stop() // 凡是用了go什么的 都是开启一个协程 相当于新弄一个任务 这里是直接停止
48      wg.Wait() // 等待所有线程同步 就在这里wait
49  }
50
```

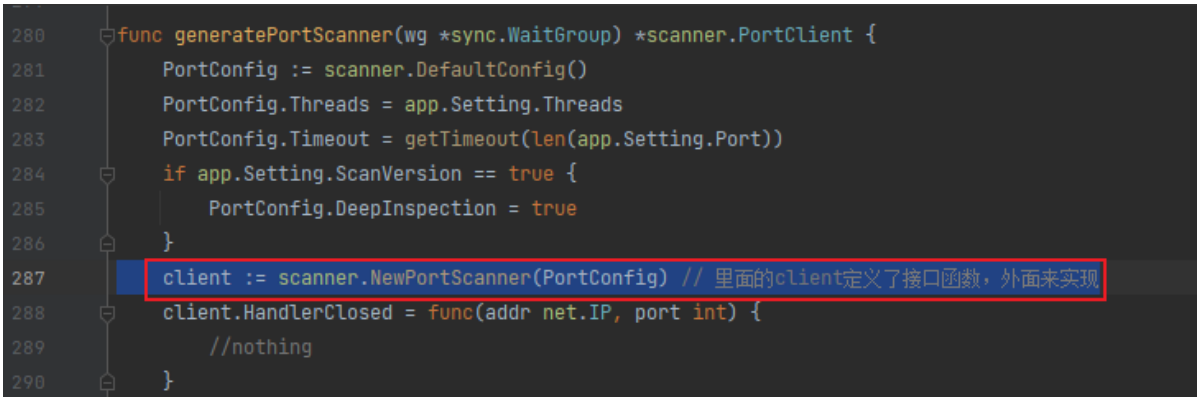
还是这段代码，我还是拿他这个PortScanner举例来解释他的写法。

从这里跟到他的这个generatePortScanner中



我挑出其中一句较为关键的话，就是位于代码287行的

```
    client := scanner.NewPortScanner(PortConfig) // 里面的client定义了接口函数，外面来实现
```



这个先放在这里标注起来，一会再看，有大用。

先从这个函数名看起。

函数传入的参数较为好理解，其实就是协程机制。

弄个wg进去， 是个指向sync.WaitGroup 类型的指针，因为golang底层是c，所以golang大量的使用指针这种语法。

这个wg传进来的目的是用于协程之间的同步，用于确保某些操作在其他操作完成后再执行。

然后再看到返回值

```
*scanner.PortClient
```

这里跟一下

```
type PortClient struct {
    *client

    HandlerClosed     func(addr net.IP, port int)
    HandlerOpen       func(addr net.IP, port int)
    HandlerNotMatched func(addr net.IP, port int, response string)
    HandlerMatched    func(addr net.IP, port int, response *gonmap.Response)
    HandlerError      func(addr net.IP, port int, err error)
}
```

```
type client struct {
    config *Config
    pool   *pool.Pool

    deferFunc func()
}
```

意思说返回值是一个指向结构体的指针

然后这个结构体又套了一个结构体，为client，里面放了配置文件和线程池参数，以及一个空函数。

先看第一层

```
type PortClient struct {
    *client

    HandlerClosed     func(addr net.IP, port int)
    HandlerOpen       func(addr net.IP, port int)
    HandlerNotMatched func(addr net.IP, port int, response string)
    HandlerMatched    func(addr net.IP, port int, response *gonmap.Response)
    HandlerError      func(addr net.IP, port int, err error)
}
```

这里定义的五个函数都为回调函数，对应了处理过程中的不同情况

具体应用场景如下

```
    }
    client := scanner.NewPortScanner(PortConfig) // 里面的client定义了接口函数，外面来实现
    client.HandlerClosed = func(addr net.IP, port int) {
        //nothing
    }
    client.HandlerOpen = func(addr net.IP, port int) {
        outputOpenResponse(addr, port)
    }
    client.HandlerNotMatched = func(addr net.IP, port int, response string) {
        outputUnknownResponse(addr, port, response)
    }
    client.HandlerMatched = func(addr net.IP, port int, response *gonmap.Response) {
        // 这里是直接获取了fingerprint的信息，是直接调用gonmap来实现的，不是作者自己实现的。
        URLRaw := fmt.Sprintf( format: "%s://%s:%d", response.FingerPrint.Service, addr.String(), port)
        URL, _ := url.Parse(URLRaw)
        if appfinger.SupportCheck(URL.Scheme) == true { // 推进去进行扫描
            pushURLTarget(URL, response)
            return
        }
        outputNmapFinger(URL, response)
        if app.Setting.Hydra == true {
            if protocol := response.FingerPrint.Service; hydra.Ok(protocol) {
                HydraScanner.Push(addr, port, protocol)
            }
        }
generatePortScanner(wg *sync.WaitGroup) *scanner.PortClient  ›  func(addr net.IP, port int)
```

```
    client.HandlerError = func(addr net.IP, port int, err error) {
        slog.Println(slog.DEBUG, s...: "PortScanner Error: ", fmt.Sprintf("#{addr.String()}:#{port}"), err)
    }
    client.Defer(func() {
        wg.Done()
    })
    return client
}
```

也就是说，比如这个时候经过探测，端口是打开的

那么就走这个回调函数

```
    client.HandlerOpen = func(addr net.IP, port int) {
        outputOpenResponse(addr, port)
    }
```

如果端口是关闭的，就走这个

```
    client.HandlerClosed = func(addr net.IP, port int) {
        //nothing
    }
```

逻辑上是这么个意思，作者在generate这一层中，只实现逻辑，并不实现具体的功能，即这个函数是端口扫描的功能函数，但是真正进行端口扫描代码却并不在这里，而在另一层中。

即逻辑和实际功能代码分离。

主打一个干湿分离。

这里跟进到实现细节看看，还是看端口扫描：

```
wg.Add( delta: 5) // 塞入五个协程
DomainScanner = generateDomainScanner(wg) // 看起来是进行域名扫描 这里之所以要把wg传进去的原因
IPScanner = generateIPScanner(wg)
PortScanner = generatePortScanner(wg) // 对端口进行扫描，然后获取端口信息，具体调用的是gonamp
URLScanner = generateURLScanner(wg)
HydraScanner = generateHydraScanner(wg)
```

```
func generatePortScanner(wg *sync.WaitGroup) *scanner.PortClient {
    PortConfig := scanner.DefaultConfig()
    PortConfig.Threads = app.Setting.Threads
    PortConfig.Timeout = getTimeout(len(app.Setting.Port))
    if app.Setting.ScanVersion == true {
        PortConfig.DeepInspection = true
    }
    client := scanner.NewPortScanner(PortConfig) // 里面的client定义了接口函数，外面来实现
    client.HandlerClosed = func(addr net.IP, port int) {
        //nothing
    }
```

```
23  func NewPortScanner(config *Config) *PortClient {
24      var client = &PortClient{
25          client:            newConfig(config, config.Threads),
26          HandlerClosed:     func(addr net.IP, port int) {},
27          HandlerOpen:       func(addr net.IP, port int) {},
28          HandlerNotMatched: func(addr net.IP, port int, response string) {},
29          HandlerMatched:    func(addr net.IP, port int, response *gonmap.Response) {},
30          HandlerError:      func(addr net.IP, port int, err error) {},
31      }
32      client.pool.Interval = config.Interval
33      client.pool.Function = func(in interface{}) {
34          nmap := gonmap.New()
35          nmap.SetTimeout(config.Timeout)
36          if config.DeepInspection == true {
37              nmap.OpenDeepIdentify()
38          }
39          value := in.(foo1)
40          status, response := nmap.ScanTimeout(value.addr.String(), value.num, 100*config.Timeout)
41          switch status {
42          case gonmap.Closed:
43              client.HandlerClosed(value.addr, value.num)
44          case gonmap.Open:
45              client.HandlerOpen(value.addr, value.num)
46          case gonmap.NotMatched:
47              client.HandlerNotMatched(value.addr, value.num, response.Raw)
48          case gonmap.Matched:
49              client.HandlerMatched(value.addr, value.num, response)
50          }
51      }
52      return client
53  }
```

通过调用nmap的nmap.ScanTimeout函数，然后获取结果status

然后根据status的分类调用对应的回调函数，如下

```
        switch status {
        case gonmap.Closed:
            client.HandlerClosed(value.addr, value.num)
        case gonmap.Open:
            client.HandlerOpen(value.addr, value.num)
        case gonmap.NotMatched:
            client.HandlerNotMatched(value.addr, value.num, response.Raw)
        case gonmap.Matched:
            client.HandlerMatched(value.addr, value.num, response)
```

然后再返回上一层，通过回调函数输出结果

```
        }
        outputNmapFinger(URL, response)
        if app.Setting.Hydra == true {
            if protocol := response.FingerPrint.Service; hydra.Ok(protocol) {
                HydraScanner.Push(addr, port, protocol)
            }
        }
    }
    client.HandlerError = func(addr net.IP, port int, err error) {
        slog.Println(slog.DEBUG, s... "PortScanner Error: ", fmt.Sprintf("#{addr.String()}:#{port}"), err)
    }
    client.Defer(func() {
        wg.Done()
    })
    return client
}
```

类似于这样的操作

然后最后再回归到协程的操作，然后返回已经经过处理的new出来的client

```
    client.HandlerError = func(addr
        slog.Println(slog.DEBUG,
    }
    client.Defer(func() {
        wg.Done()
    })
    return client
}
```

返回实例之后，就到了上一层的这里

```go
func Start() {
    //启用看门狗函数定时输出负载情况
    go watchDog()
    //下发扫描任务
    var wg = &sync.WaitGroup{} // 并发控制机制 等待几个协程都执行完毕，再往下执行
    wg.Add( delta: 5) // 塞入五个协程
    DomainScanner = generateDomainScanner(wg) // 看起来是进行域名扫描 这里之所以要
    IPScanner = generateIPScanner(wg)
    PortScanner = generatePortScanner(wg) // 对端口进行扫描，然后获取端口信息，具体
    URLScanner = generateURLScanner(wg)
    HydraScanner = generateHydraScanner(wg)
    //扫描器进入监听状态
    start()
    //开始分发扫描任务
    for _, expr := range app.Setting.Target {
        pushTarget(expr)
    }
    slog.Println(slog.INFO, s...: "所有扫描任务已下发完毕")
    //根据扫描情况，关闭scanner
    go stop() // 凡是用了go什么的 都是开启一个协程 相当于新弄一个任务 这里是直接停止
    wg.Wait() // 等待所有线程同步 就在这里wait
}
```

然后走下面的任务开始逻辑

```go
    //扫描器进入监听状态
    start()
    //开始分发扫描任务
    for _, expr := range app.Setting.Target {
        pushTarget(expr)
    }
    slog.Println(slog.INFO, s...: "所有扫描任务已下发完毕")
    //根据扫描情况，关闭scanner
    go stop() // 凡是用了go什么的 都是开启一个协程 相当于新弄一个任务 这里是直接停止
    wg.Wait() // 等待所有线程同步 就在这里wait
}
```

这里我添加的根据注释一条条看

首先是start

```go
// 这里调用start就是用golang的协程 go一下 相当于拉一个协程出来启动 然后运行
func start() {
    go DomainScanner.Start()
    go IPScanner.Start()
    go PortScanner.Start()
    go URLScanner.Start()
    go HydraScanner.Start()
    time.Sleep(time.Second * 1)
    slog.Println(slog.INFO, s...: "Domain、IP、Port、URL、Hydra引擎已准备就绪")
}
```

一个go语句后面，就是启动一个协程，然后这里start的意思就是把这些任务组都跑起来。

后面这段是塞目标进去

```go
    for _, expr := range app.Setting.Target {
        pushTarget(expr)
    }
    slog.Println(slog.INFO, s...: "所有扫描任务已下发完毕")
    //根据扫描情况，关闭scanner
    go stop() // 凡是用了go什么的 都是开启一个协程 相当于新弄一个任务 这里是直接停止
    wg.Wait() // 等待所有线程同步 就在这里wait
}
```

其实就是传参

先用for循环把外部传入的Target参数一个个取出来

然后利用pushTarget函数传进去

这里跟进一下

```go
func pushTarget(expr string) {
    if expr == "" {
        return
    }
    // 如果关键字含粘贴 就从粘贴板读取对应的数据
    if expr == "paste" || expr == "clipboard" {
        if clipboard.Unsupported == true {
            slog.Println(slog.ERROR, runtime.GOOS, "clipboard unsupported")
        }
        clipboardStr, _ := clipboard.ReadAll()
        for _, line := range strings.Split(clipboardStr, sep: "\n") {
            line = strings.ReplaceAll(line, old: "\r", new: "")
            pushTarget(line)
        }
        return
    }
    // 如果是ipv4地址 就加上http以及https协议头，然后尝试访问
    if uri.IsIPv4(expr) {
        IPScanner.Push(net.ParseIP(expr))
        if app.Setting.Check == true {
            pushURLTarget(uri.URLParse("http://"+expr), response: nil)
            pushURLTarget(uri.URLParse("https://"+expr), response: nil)
        }
```

这里我们直接看到下面的使用了PortScanner.Push的模块

```go
    if uri.IsNetlocPort(expr) {
        netloc, port := uri.SplitWithNetlocPort(expr)
        if uri.IsIPv4(netloc) {
            PortScanner.Push(net.ParseIP(netloc), port)
        }
```

```go
func (c *PortClient) Push(ip net.IP, num int) {
    c.pool.Push(foo1{ addr: ip,  num: num})
}

```

这里面含有一个foo1结构体

```go
type foo1 struct {
    addr net.IP
    num  int
}
```

参数也很好理解，地址和数量罢了

再跟进这个pool

```go
//塞一个任务进去
func (p *Pool) Push(i interface{}) {
    if p.Done {
        return
    }
    p.in <- i
}
```

看到这一句

```go
p.in <- i
```

就是把i塞进任务队列里

```go
type Pool struct {
    //母版函数
    Function func(interface{})
    //Pool输入队列
    in chan interface{}
    //size用来表明池的大小，不能超发。
    threads int
    //启动协程等待时间
    Interval time.Duration
    //正在执行的任务清单
    JobsList *sync.Map
    //正在工作的协程数量
    length int32
    //用于阻塞
    wg *sync.WaitGroup
    //提前结束标识符
    Done bool
}
```

因此这里就全部贯通起来了，关于这个kscan的端口扫描逻辑。

这里再从头梳理一次

先

```
    PortScanner = generatePortScanner(wg) // 对端口进行扫描，然后获取端口信息，具体调用的是gonamp
```

这里实例化PortScanner

然后

```
start()
```

把协程启起来

然后

```
    for _, expr := range app.Setting.Target {
        pushTarget(expr)
    }
```

把要扫描的东西给塞进去

就这么几个步骤，具体细节的话，建议自己拿代码过来看看，更加清楚些。

最后就是sync的收尾部分

```
go stop()
wg.Wait()
```

这里看我写的注释

```
go stop() // 凡是用了go什么的 都是开启一个协程 相当于新弄一个任务 这里是直接停止
wg.Wait() // 等待所有线程同步 就在这里wait
```

这里的注释就解释了这两句话的具体作用

0x03 其他模块的介绍&我的魔改

这里端口扫描是用的这种写法， 其他的扫描也是，比如从34-38的这5句话

```go
func Start() {
    //启用看门狗函数定时输出负载情况
    go watchDog()
    //下发扫描任务
    var wg = &sync.WaitGroup{} // 并发控制机制 等待几个协程都执行完毕，再往下执行
    wg.Add( delta: 5) // 塞入五个协程
    DomainScanner = generateDomainScanner(wg) // 看起来是进行域名扫描 这里之所以要把wg传
    IPScanner = generateIPScanner(wg)
    PortScanner = generatePortScanner(wg) // 对端口进行扫描，然后获取端口信息，具体调用的
    URLScanner = generateURLScanner(wg)
    HydraScanner = generateHydraScanner(wg)
    //扫描器进入监听状态
    start()
    //开始分发扫描任务
    for _, expr := range app.Setting.Target {
        pushTarget(expr)
    }
    slog.Println(slog.INFO, s...: "所有扫描任务已下发完毕")
    //根据扫描情况，关闭scanner
    go stop() // 凡是用了go什么的 都是开启一个协程 相当于新弄一个任务 这里是直接停止
    wg.Wait() // 等待所有线程同步 就在这里wait
}
```

从34行开始看起

```go
func generateDomainScanner(wg *sync.WaitGroup) *scanner.DomainClient {
    DomainConfig := scanner.DefaultConfig() // 定义一个配置文件选项
    DomainConfig.Threads = 10 // 定义10个线程
    client := scanner.NewDomainScanner(DomainConfig) // 这个client实际上是个结构体
    client.HandlerRealIP = func(domain string, ip net.IP) { // 实现接口函数的处理模块
        IPScanner.Push(ip)
    }
    client.HandlerIsCDN = func(domain, CDNInfo string) { // 判断是否为cdn模块，同样也是实现了这个接口函数
        outputCDNRecord(domain, CDNInfo)
    }
    client.HandlerError = func(domain string, err error) { // 实现错误处理接口函数
        slog.Println(slog.DEBUG, s...: "DomainScanner Error: ", domain, err)
    }
    client.Defer(func() {
        wg.Done()
    })
    return client
}
```

```go
// 这个函数主要就是把接口的功能都实现了，最关键的功能就是输入domain，解析ip
func NewDomainScanner(config *Config) *DomainClient { // 传入config结构体 返回一个结构体，DomainClient类型的
    var client = &DomainClient{
        client:        newConfig(config, config.Threads),
        HandlerIsCDN:  func(domain, CDNInfo string) {},
        HandlerRealIP: func(domain string, ip net.IP) {},
        HandlerError:  func(domain string, err error) {},
    }
    client.pool.Interval = config.Interval // 定义了一个config结构体，然后这里摘结构体里面的一个值赋出去，这里是赋时间duration
    client.pool.Function = func(in interface{}) { // 这里前置那里先定义一个接口，然后再来这里实现，传入in值
        domain := in.(string) // 这里字符串化这个in值，这个in值就是传入的domain参数，外部传入

        ip, err := cdn.Resolution(domain) // 调用cdn模块中的resolution函数来处理这个domain
        if err != nil {
            client.HandlerError(domain, err)
            return
        }
        // 这个掉毛东西好像也不是数据库 是一个map
        DomainDatabase.Store(domain, ip)
        if CDNCheck == false { // 如果不启动cdn检查，直接return对应ip
            client.HandlerRealIP(domain, net.ParseIP(ip)) // 这里是利用parseip解析ip的有效性
            return
        }

        if ok, result, _ := cdn.FindWithDomain(domain); ok {
            client.HandlerIsCDN(domain, result)
            return
        }
        if ok, result, _ := cdn.FindWithIP(ip); ok {
            client.HandlerIsCDN(domain, result)
            return
        }
        client.HandlerRealIP(domain, net.ParseIP(ip))
    }
    return client
}
```

```go
func Resolution(domain string) (string, error) { // 传入字符串，传出字符串和error
    ips, err := dns.LookupIP(domain) // 调用lookupIP函数查询domain
    if err != nil { return "", err }
    return ips[0], nil // 这里实际上只返回第一个ip，也没有增加cdn之类的判断啥的
    // （优化方案）我理解这里可以修改成，如果是cdn的就返回整个ip组，然后如果是真实ip的就返回一个ip

}
```

```go
//优化死锁
func LookupIP(domain string) ([]string, error) {
    var IPs []string
    var lastErr error
    for _, resolver := range resolvers {
        func() {
            querySemaphore <- struct{}{}            // 请求信号量
            defer func() { <-querySemaphore }()     // 使用defer确保信号量的释放

            ips, err := resolver.LookupIPAddr(context.Background(), domain)
            if err != nil {
                lastErr = err
                return
            }
            for _, v := range ips {
                IPs = append(IPs, v.IP.String())
            }
        }()
    }
    IPs = misc.RemoveDuplicateElement(IPs)
    return IPs, lastErr
}
```

这里其实不是原代码，我魔改了。

我魔改的原因是因为我想让程序多弄点dnsserver来解析，因为dnsserver的数量少了的话，解析有些url可能不一定能解析到。

解析不到，就找不到cdn，因此我这里做了修改。

因此我首先是在原作者的domainServers这里加了一堆server

```
var (
    domainServers = []string{
        // 你的服务器列表，即使是100个
        "114.114.114.114",
        //"8.8.8.8:53",
        "223.6.6.6",
        //"8.8.4.4",
        //"8.8.8.8",
        //"1.1.1.1",
        //"119.29.29.29",
        //"223.5.5.5",
        //"180.76.76.76",
        //"117.50.10.10",
        //"208.67.222.222",
        //"223.6.6.6",
        //"168.95.1.1",
        //"168.95.192.1",
        //"168.95.1.1",
        //"202.175.45.2",
        //"202.248.20.133",
        //"211.129.155.175",
        //"104.152.211.99",
        //"9.9.9.9",
        //"61.19.42.5",
```

现在我是注释掉了我加上的dnsserver，因为直接加会有问题。

直接加上，加一堆，最后代码跑出来反而不解析url的ip，从而无法找到cdn。

原来的作者代码是这样的

```go
package dns

import (
	"context"
	"github.com/miekg/dns"
	"kscan/lib/misc"
	"net"
	"time"
)

var domainServers = []string{
	"114.114.114.114:53",
	//"8.8.8.8:53",
	"223.6.6.6:53",
}

var resolvers = generateResolver()

func LookupCNAME(domain string) ([]string, error) {
	var lastErr error
	for _, domainServer := range domainServers {
		CNAMES, err := LookupCNAMEWithServer(domain, domainServer)
		if err != nil {
			lastErr = err
		}
		return CNAMES, nil
	}
	return nil, lastErr
}
```

只有两个dnsserver

然后我如果单纯在这里填充，扫描结果就会无法识别url的ip，以下是扫描结果的对比

```
Tips: 在Windows操作系统下，可以新增环境变量KSCAN_COLOR=TRUE,开启颜色显示
[+]2023/10/10 14:16:52 当前环境为: windows, 输出编码为: utf-8
[+]2023/10/10 14:16:53 成功加载HTTP指纹:[24758]条
[+]2023/10/10 14:16:53 成功加载NMAP探针:[150]个,指纹[11916]条
[+]2023/10/10 14:16:53 检测到qqwry.dat,将自动启动CDN检测功能，可使用-Dn参数关闭该功能
[+]2023/10/10 14:16:54 Domain、IP、Port、URL、Hydra引擎已准备就绪
cdn://baidu.com                 CDN资产                         CDNInfo:IP地址，且不在同一网段，该域名可能使用了CDN,Domain:ba
du.com,Length:0
cdn://taobao.com                CDN资产                         Length:0,CDNInfo:指向多个IP地址，且不在同一网段，该域名可能使
用,Domain:taobao.com
cdn://qq.com                    CDN资产                         Length:0,CDNInfo:IP地址，且不在同一网段，该域名可能使用了CDN,
omain:qq.com
http://tentcent.com             502BadGateway                   Digest:</h1></center>\r\n<hr><c,Length:339,Port:80,Domain:ten
cent.com
[+]2023/10/10 14:16:55 所有扫描任务已下发完毕
```

```
Tips: 可以使用--spy 172，将会对172.16.0.1/12进行网关存活性探测
[+]2023/10/10 14:18:52 当前环境为: windows, 输出编码为: utf-8
[+]2023/10/10 14:18:52 成功加载HTTP指纹:[24758]条
[+]2023/10/10 14:18:52 成功加载NMAP探针:[150]个,指纹[11916]条
[+]2023/10/10 14:18:52 检测到qqwry.dat,将自动启动CDN检测功能，可使用-Dn参数关闭该功能
[+]2023/10/10 14:18:53 Domain、IP、Port、URL、Hydra引擎已准备就绪
http://tentcent.com             502BadGateway                   Port:80,Domain:tentcent.com,Digest:color=\"white\">\r\n<cen,L
ngth:339
[+]2023/10/10 14:18:55 所有扫描任务已下发完毕
http://taobao.com               淘宝                            FingerPrint:蓝凌OA(EKP);Citrix-Netscaler;KISSY;Tengine;HTML5,
ength:93507,FoundIP:0.0.603,730.0.0.1,832.0.,Port:80,Domain:taobao.com,FoundDomain:.com、sf.taobao.com、ju.ta,Digest:淘
宝淘宝网亚洲较大的网上交易平台，提供各类服饰、
https://taobao.com              淘宝                            FingerPrint:蓝凌OA(EKP);Citrix-Netscaler;KISSY;Tengine;HTML5,
ength:93507,FoundIP:.0.0.603,730.0.0.1,832.0,Port:443,Domain:taobao.com,FoundDomain:hk、food.tmall.com、jia.ta,Digest:"
淘宝淘宝网亚洲较大的网上交易平台，提供各类服饰
http://baidu.com                百度一下，你就知道              Digest:全球领先的中文搜索引擎、致力于让网民更便捷地获取,Lengt
:400977,FoundDomain:idu.com、d.hiphotos.baidu,Domain:baidu.com,FingerPrint:JQuery;HTML5,Port:80
https://baidu.com               百度一下，你就知道              Port:443,Digest:全球领先的中文搜索引擎、致力于让网民更便捷地获
取,Length:401142,Domain:baidu.com,FoundDomain:、t11.baidu.com、t12.baidu,FingerPrint:JQuery;HTML5
http://163.com                  网易                            Digest:"网易网易邮箱游戏新闻体育娱乐女性亚运论坛短信数,Length
603186,Domain:163.com,FoundDomain:.com、id5.163.com、run.spo,Port:80,FingerPrint:Windows;DigiCert-Cert;360-Webmaster-Pl
tform;Sogou-Webmaster-Platform;Apache-Struts2;Frame;baidu-webmaster-platform;google-webmaster-platform
https://163.com                 网易                            Port:443,Digest:网易网易邮箱游戏新闻体育娱乐女性亚运论坛短信数
码,FoundDomain:163.com、2Fbjnewsrec-cv.w,FingerPrint:Windows;DigiCert-Cert;360-Webmaster-Platform;Sogou-Webmaster-Platf
```

可以看到后者加上过量的server之后就无法识别cdn了。

也没有ip，所以我有理由怀疑是被巨量的server冲爆了导致的。

那么怎么改呢？这里先说结论，就是这个问题到最后我也没有解决。

我读作者代码的时候想了一些办法，例如修改线程，增加信号量，或者domian判活前置等等，都没有解决这个问题，然后我就开摆了

我决定不动这个server的量

我从功能上入手，改一下他的逻辑，然后增加判断成功率。

其他的我懒得动了，累了，还是fscan好，改起来嘎嘎简单，这个kscan的作者呀，实在觉得他是开发转安全的人，一个工具写这么规范，写这么复杂干什么。

难搞

我这里就直接改他判断CDN的逻辑，找这个逻辑在哪又找半天

他原本的代码判断CDN在这

```go
func FindWithDomain(domain string) (bool, string, error) {
    IPs, err := dns.LookupIP(domain)
    if err != nil {
        slog.Println(slog.DEBUG, domain, "lookupIP err :", err)
        return false, "", err
    }
    if uri.SameSegment(IPs...) == false {
        return true, "域名指向多个IP地址，且不在同一网段，该域名可能使用了CDN技术", nil
    }

```

```go
func SameSegment(ips ...string) bool {
        if len(ips) == 0 {
                return true
        }
        first := ips[0]
        _, network, _ := net.ParseCIDR(first + "/24")
        for _, ip := range ips[1:] {
                if Contains(network, net.ParseIP(ip)) == false {
                        return false
                }
        }
        return true
}
```

这个代码什么意思呢，就是说

如果如果发现了多个ip，那么从中间取一个ip，然后裂变成c段。

然后再从后面的ip中用for循环不断的取ip出来，然后如果后面的ip和前面的ip在一个c段中，那就直接g，然后告诉你有cdn。

好好好，这么玩是吧，那先按照原作者的逻辑跑一次。

结果识别出来了，没有问题。

但是我在实战的过程中遇到了一种情况，就是一堆CDN中，确实存在一个c段的CDN的ip。

然后我直接加了一个前置的判断量，只要解析大于两个，直接干掉。

```go
func SameSegment(ips ...string) bool {
    if len(ips) == 0 {
        return true
    }

    if len(ips) > 2{
        return false
    }

    // 意思是如果都不在一个c，那么就直接g，因为如果都不在一个c，肯定大概率是cdn
    first := ips[0]
    _, network, _ := net.ParseCIDR(first + "/24")
    for _, ip := range ips[1:] {
        if Contains(network, net.ParseIP(ip)) == false {
            return false
        }
    }
    return true
}
```

另外作者的CDN判断除了对于解析出来的ip的c段判断，还有个cname的判断。

```go
    for _, cname := range CNAMES {
        if strings.Contains(cname, substr: "cdn") {
            return true, "CNAME中含有关键字：cdn，该域名可能使用了CDN技术", nil

        }
```

意思就是cname中如果含有cdn，那就直接g，判断为cdn。

这个我觉得不错，很奈斯。

然后我在扫描的时候还遇到了一种情况，就是扫到了阿里云的防火墙。

这个防火墙吧，开一堆端口，极大的消耗了我的扫描资源，贼恶心。

```
http://47.96.193.199:81       阿里云Web应用防火墙          Port:81,FoundDomain:yundun.console.aliyun.co,FingerPrint:Tengi
ne;HTML5;Tenginehttpd;cpe:,Digest:阿里云应用防火墙简体中文阿里云应用防火墙网站暂时,Length:11000,Addr:浙江省杭州市阿里云
http://47.96.193.199:7001     阿里云Web应用防火墙          FoundDomain:yundun.console.aliyun.co,FingerPrint:Tengine;HTML5
;Tenginehttpd;cpe:,Port:7001,Addr:浙江省杭州市阿里云,Length:11000,Digest:"阿里云应用防火墙简体中文阿里云应用防火墙网站暂
pop3://47.96.193.199:110      response is empty          Port:110,Length:0
shell://47.96.193.199:514     response is empty          Length:0,Port:514
```

这里我想做一个扫描的filter，也就是说，在扫描到了这个吊玩意之后，就停止目前执行任务的这个协程。

那么还是得先看源码，kscan这里其实有个好玩的地方。

在运行的时候，明明只输入了域名，但是他会自动扫描域名解析出来的IP。

作者怎么做的呢?

直接放跟的过程的图

```
189     if len(app.Setting.Target) > 0 { // 这里的意思就是 有target才继续下面的扫描
190         // 扫描模块初始化
191         // 主要做了三件事 指纹 日志 CDN检测的初始化
192         InitKscan()
193         //开始扫描
194         run.Start()
195     }
```

```
func Start() {
    //启用看门狗函数定时输出负载情况
    go watchDog()
    //下发扫描任务
    var wg = &sync.WaitGroup{} // 并发控制机制 等待几个协程都执行
    wg.Add( delta: 5) // 塞入五个协程
    DomainScanner = generateDomainScanner(wg) // 看起来是进行域
    IPScanner = generateIPScanner(wg)
    PortScanner = generatePortScanner(wg) // 对端口进行扫描，然
    URLScanner = generateURLScanner(wg)
    HydraScanner = generateHydraScanner(wg)
```

```
223 func generateDomainScanner(wg *sync.WaitGroup) *scanner.DomainClient {
224     DomainConfig := scanner.DefaultConfig() // 定义一个配置文件选项
225     DomainConfig.Threads = 10 // 定义10个线程
226     client := scanner.NewDomainScanner(DomainConfig) // 这个client实际上是个结构体
227     client.HandlerRealIP = func(domain string, ip net.IP) { // 实现接口函数的处理模块
228         IPScanner.Push(ip)
229     }
230     client.HandlerIsCDN = func(domain, CDNInfo string) { // 判断是否为cdn模块，同样也是
231         outputCDNRecord(domain, CDNInfo)
232     }
233     client.HandlerError = func(domain string, err error) { // 实现错误处理接口函数
234         slog.Println(slog.DEBUG, s...:"DomainScanner Error: ", domain, err)
235     }
236     client.Defer(func() {
237         wg.Done()
238     })
239     return client
240 }
```

```
DomainScanner = generateDomainScanner(wg) //
IPScanner = generateIPScanner(wg)
PortScanner = generatePortScanner(wg) // 对端
URLScanner = generateURLScanner(wg)
HydraScanner = generateHydraScanner(wg)
```

```
242 func generateIPScanner(wg *sync.WaitGroup) *scanner.IPClient {
243     IPConfig := scanner.DefaultConfig()
244     IPConfig.Threads = 200
245     IPConfig.Timeout = 200 * time.Millisecond
246     IPConfig.HostDiscoverClosed = app.Setting.ClosePing
247     client := scanner.NewIPScanner(IPConfig)
248     client.HandlerDie = func(addr net.IP) {
249         slog.Println(slog.DEBUG, addr.String(), " is die")
250     }
251     client.HandlerAlive = func(addr net.IP) {
252         //启用端口存活性探测任务下发器
253         slog.Println(slog.DEBUG, addr.String(), " is alive")
254         for _, port := range app.Setting.Port { // 直接给port塞进去
255             PortScanner.Push(addr, port)
256         }
257     }
258     client.HandlerError = func(addr net.IP, err error) {
259         slog.Println(slog.DEBUG, s...:"IPScanner Error: ", addr.String(), err)
260     }
261     client.Defer(func() {
262         wg.Done()
263     })
264     return client
```

```
var wg = &sync.WaitGroup{} // 并发控制机制 等
wg.Add( delta: 5) // 塞入五个协程
DomainScanner = generateDomainScanner(wg)
IPScanner = generateIPScanner(wg)
PortScanner = generatePortScanner(wg) // 对
URLScanner = generateURLScanner(wg)
HydraScanner = generateHydraScanner(wg)
//扫描器进入监听状态
```

```go
func NewPortScanner(config *Config) *PortClient {
    var client = &PortClient{
        client:            newConfig(config, config.Threads),
        HandlerClosed:     func(addr net.IP, port int) {},
        HandlerOpen:       func(addr net.IP, port int) {},
        HandlerNotMatched: func(addr net.IP, port int, response string) {},
        HandlerMatched:    func(addr net.IP, port int, response *gonmap.Response) {},
        HandlerError:      func(addr net.IP, port int, err error) {},
    }
    client.pool.Interval = config.Interval
    client.pool.Function = func(in interface{}) {
        nmap := gonmap.New()
        nmap.SetTimeout(config.Timeout)
        if config.DeepInspection == true {
            nmap.OpenDeepIdentify()
        }
        value := in.(foo1)
        status, response := nmap.ScanTimeout(value.addr.String(), value.num, 100*config.Timeout)
        switch status {
        case gonmap.Closed:
            client.HandlerClosed(value.addr, value.num)
        case gonmap.Open:
            client.HandlerOpen(value.addr, value.num)
        case gonmap.NotMatched:
            client.HandlerNotMatched(value.addr, value.num, response.Raw)
        case gonmap.Matched:
            client.HandlerMatched(value.addr, value.num, response)
        }
    }
    return client
}
```

所以其实是一个串联的过程

意思就是，domian能解析成ip，就自动开始ip存活检测。

ip能活，就自动开始扫对应ip的端口。

然后一步步串联下去。

好好好，主打一个自动化是吧。

我现在就想给这个阿里云防火墙干了，现在得想想办法。

想来想去，我打算在传参这里做手脚。

```go
for _, expr := range app.Setting.Target {
    pushTarget(expr)
}
```

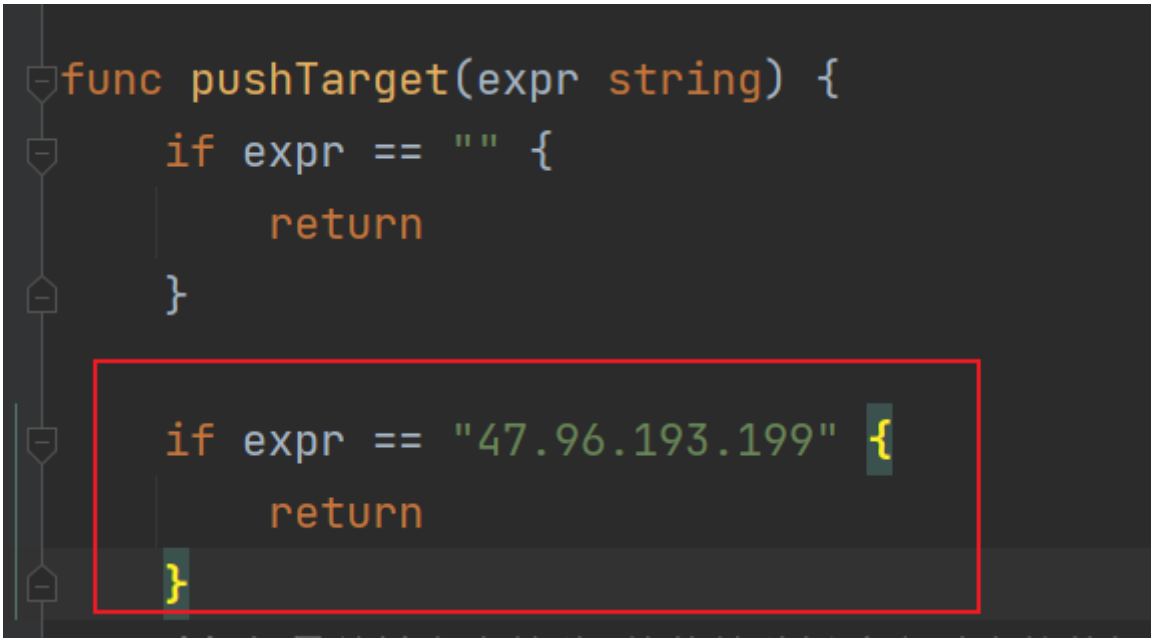
就是这个地方

先点进去

```go
func pushTarget(expr string) {
    if expr == "" {
        return
    }
    // 如果关键字含粘贴 就从粘贴板读取对应的数据
    if expr == "paste" || expr == "clipboard" {
        if clipboard.Unsupported == true {
            slog.Println(slog.ERROR, runtime.GOOS, "clipboard unsupported")
        }
        clipboardStr, _ := clipboard.ReadAll()
        for _, line := range strings.Split(clipboardStr, sep: "\n") {
            line = strings.ReplaceAll(line, old: "\r", new: "")
            pushTarget(line)
        }
        return
    }
```

实现方法1 可以做一个简单的黑名单

做出来类似是这样的



但是这也太撒比了，不可能知道所有的应用防火墙地址啊，直接pass

实现方法2 做启发式的扫描

还是简单粗暴一点，这些东西，可能不限于应用防火墙，还有蜜罐之类的，都是这个特征，啥玩意都开了，但是都没用

```
mssql://47.96.193.199:1433      response is empty       Port:1433,Length:0
pop3://47.96.193.199:110        response is empty       Port:110,Length:0
pop3://47.96.193.199:110        response is empty       Length:0,Port:110
rdp://47.96.193.199:3389        response is empty       Port:3389,Length:0
```

因此直接增加一个启发式模块，代码如下。

原理就是一个ip开了太多个端口，就判定为是有问题，直接跳过。

```go
const NUM_PORTS_TO_CHECK = 50  // 你可以设置为任意你希望检测的数量

type PortClient struct {
    *client
```

```
    HandlerClosed     func(addr net.IP, port int)
    HandlerOpen       func(addr net.IP, port int)
    HandlerNotMatched func(addr net.IP, port int, response string)
    HandlerMatched    func(addr net.IP, port int, response *gonmap.Response)
    HandlerError      func(addr net.IP, port int, err error)
    heuristicChecked map[string]bool
}

func (client *PortClient) heuristicCheck(addr net.IP, config *Config) bool {
    if _, alreadyChecked := client.heuristicChecked[addr.String()];
alreadyChecked {
        return false
    }

    openPortsCount := 0
    nmap := gonmap.New()
    nmap.SetTimeout(config.Timeout)
    for i := 1; i <= NUM_PORTS_TO_CHECK; i++ {
        status, _ := nmap.ScanTimeout(addr.String(), i, 100*config.Timeout)
        if status == gonmap.Open {
            openPortsCount++
        }
    }

    client.heuristicChecked[addr.String()] = true
    return openPortsCount >= NUM_PORTS_TO_CHECK
}

func NewPortScanner(config *Config) *PortClient {
    var client = &PortClient{
        client:            newConfig(config, config.Threads),
        HandlerClosed:     func(addr net.IP, port int) {},
        HandlerOpen:       func(addr net.IP, port int) {},
        HandlerNotMatched: func(addr net.IP, port int, response string) {},
        HandlerMatched:    func(addr net.IP, port int, response *gonmap.Response)
{},
        HandlerError:      func(addr net.IP, port int, err error) {},
        heuristicChecked: make(map[string]bool),
    }

    client.pool.Interval = config.Interval
    client.pool.Function = func(in interface{}) {
        value := in.(foo1)

        // 启发式检查
        if client.heuristicCheck(value.addr, config) { // Pass config here
            fmt.Printf("Skipping %s due to heuristic detection of open ports\n",
value.addr)
            return
        }

        nmap := gonmap.New()
        nmap.SetTimeout(config.Timeout)
        if config.DeepInspection == true {
            nmap.OpenDeepIdentify()
        }
```

```
        status, response := nmap.ScanTimeout(value.addr.String(), value.num,
100*config.Timeout)
        switch status {
        case gonmap.Closed:
            client.HandlerClosed(value.addr, value.num)
        case gonmap.Open:
            client.HandlerOpen(value.addr, value.num)
        case gonmap.NotMatched:
            client.HandlerNotMatched(value.addr, value.num, response.Raw)
        case gonmap.Matched:
            client.HandlerMatched(value.addr, value.num, response)
        }
    }
    return client
}
```

实现方法3 增加黑名单关键字

魔改代码如下

```
const ALIYUN_FIREWALL_KEYWORD = "阿里云web应用防火墙"


type PortClient struct {
    *client

    HandlerClosed     func(addr net.IP, port int)
    HandlerOpen       func(addr net.IP, port int)
    HandlerNotMatched func(addr net.IP, port int, response string)
    HandlerMatched    func(addr net.IP, port int, response *gonmap.Response)
    HandlerError      func(addr net.IP, port int, err error)
}

func (client *PortClient) isAliyunFirewallDetected(response string) bool {
    return strings.Contains(response, ALIYUN_FIREWALL_KEYWORD)
}

func NewPortScanner(config *Config) *PortClient {
    var client = &PortClient{
        client:            newConfig(config, config.Threads),
        HandlerClosed:     func(addr net.IP, port int) {},
        HandlerOpen:       func(addr net.IP, port int) {},
        HandlerNotMatched: func(addr net.IP, port int, response string) {},
        HandlerMatched:    func(addr net.IP, port int, response *gonmap.Response)
{},
        HandlerError:      func(addr net.IP, port int, err error) {},
    }
        client.pool.Interval = config.Interval
        client.pool.Function = func(in interface{}) {
        value := in.(foo1)

        nmap := gonmap.New()
        nmap.SetTimeout(config.Timeout)
        if config.DeepInspection == true {
            nmap.OpenDeepIdentify()
```

```
        }

        status, response := nmap.ScanTimeout(value.addr.String(), value.num, 100*config.Timeout)

            if response != nil &&  client.isAliyunFirewallDetected(response.Raw) {
            fmt.Printf("Detected Aliyun Firewall at %s, skipping the rest of the scan\n", value.addr)
            return
        }

        switch status {
        case gonmap.Closed:
            client.HandlerClosed(value.addr, value.num)
        case gonmap.Open:
            client.HandlerOpen(value.addr, value.num)
        case gonmap.NotMatched:
            client.HandlerNotMatched(value.addr, value.num, response.Raw)
        case gonmap.Matched:
            client.HandlerMatched(value.addr, value.num, response)
        }
    }
    return client
}
```

累了，码太多，累了。

后面这个就是直接加一个response黑名单，但是我发现还有问题，因为我没做协程间通信机制，因此这个扫出来到最后还是会有问题，死活都会扫一次。

```
rocrail://47.96.193.199:8051   response is empty         Port:8051,Length:0
rocrail://47.96.193.199:8051   response is empty         Port:8051,Length:0
rocrail://47.96.193.199:8051   response is empty         Port:8051,Length:0
rocrail://47.96.193.199:8051   response is empty         Length:0,Port:8051
rocrail://47.96.193.199:8051   response is empty         Port:8051,Length:0
Detected Aliyun Firewall at 47.96.193.199, skipping the rest of the scan
Detected Aliyun Firewall at 47.96.193.199, skipping the rest of the scan
Detected Aliyun Firewall at 47.96.193.199, skipping the rest of the scan
Detected Aliyun Firewall at 47.96.193.199, skipping the rest of the scan
Detected Aliyun Firewall at 47.96.193.199, skipping the rest of the scan
Detected Aliyun Firewall at 47.96.193.199, skipping the rest of the scan
Detected Aliyun Firewall at 47.96.193.199, skipping the rest of the scan
Detected Aliyun Firewall at 47.96.193.199, skipping the rest of the scan
Detected Aliyun Firewall at 47.96.193.199, skipping the rest of the scan
Detected Aliyun Firewall at 47.96.193.199, skipping the rest of the scan
Detected Aliyun Firewall at 47.96.193.199, skipping the rest of the scan
Detected Aliyun Firewall at 47.96.193.199, skipping the rest of the scan
Detected Aliyun Firewall at 47.96.193.199, skipping the rest of the scan
Detected Aliyun Firewall at 47.96.193.199, skipping the rest of the scan
Detected Aliyun Firewall at 47.96.193.199, skipping the rest of the scan
Detected Aliyun Firewall at 47.96.193.199, skipping the rest of the scan
Detected Aliyun Firewall at 47.96.193.199, skipping the rest of the scan
Detected Aliyun Firewall at 47.96.193.199, skipping the rest of the scan
Detected Aliyun Firewall at 47.96.193.199, skipping the rest of the scan
```

说是说skip了，但是节约的开销不大，就是阻止往下走了而已

```
166
167             if response != nil &&  client.isAliyunFirewallDetected(response.Raw) {
168             fmt.Printf( format: "Detected Aliyun Firewall at %s, skipping the rest of
169             return
170         }
171
172         switch status {
173         case gonmap.Closed:
174             client.HandlerClosed(value.addr, value.num)
175         case gonmap.Open:
176             client.HandlerOpen(value.addr, value.num)
177         case gonmap.NotMatched:
178             client.HandlerNotMatched(value.addr, value.num, response.Raw)
179         case gonmap.Matched:
180             client.HandlerMatched(value.addr, value.num, response)
181         }
182     }
183     return client
184 }
```

即后面switch的这段不再执行了。

难顶。

那么就来一个

实现方法4 增加进程间通信&全局变量

这里我首先保留3的方法不变。

然后增加了

```
190 func isBlockedIP(ip string) bool {
191     _, blocked := blockedIPs.Load(ip)
192     return blocked
193 }
```

```
132 var blockedIPs sync.Map
```

```
170             if response != nil &&  client.isAliyunFirewallDetected(response.Raw) {
171             fmt.Printf( format: "Detected Aliyun Firewall at %s, skipping the rest of the scan\
172             blockedIPs.Store(value.addr.String(),  value: true)  // Mark this IP as blocked.
173             return
174         }
```

这里给进来的那个ip存起来

最后再写到结构体方法里

```
195 func (c *PortClient) Push(ip net.IP, num int) {
196     if isBlockedIP(ip.String()) {
197         fmt.Printf( format: "IP %s is blocked by Aliyun Firewall, skipping...\n", ip)
198         return
199     }
200     c.pool.Push(foo1{ addr: ip,  num: num})
201 }
202
```

即实现了，一旦发现这个ip，立即同步给其他协程，然后后续push该ip前做一次检测。

然后运行实验：

```
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
IP 47.96.193.199 is blocked by Aliyun Firewall, skipping...
```

遇到的时候就大批量的skipping

然后用同样的资产进行速度测试

```
http://47.96.193.199:8026      阿里云Web应用防火墙        Port:8026,Addr:浙江省杭州市阿里云,Digest:"阿里云应用防火墙简体中文阿里云应用防火墙网站暂,FoundDomain:yundun.console.aliyun.com,FingerPrint:Tengine;HTML5;Tenginehttpd;cpe:,Length:11000
[+]2023/10/10 17:40:30 程序执行总时长为: [4m58.5708845s]
[+]2023/10/10 17:40:30 若有问题欢迎来我的Github提交Bug[https://github.com/lcvvvv/kscan/]
```

都是164个资产，原版由于碰到了这个防火墙没有检测机制，跑了将近五分钟

我的魔改版本

```
[+]2023/10/10 17:41:05 程序执行总时长为: [2m7.2434077s]
[+]2023/10/10 17:41:05 若有问题欢迎来我的Github提交Bug[https://github.com/lcvvvv/kscan/]
```

大概两分钟多一点

在猛猛遇到这种垃圾资产的时候，这个功能还是很好用的。

后续要做成一个配置文件，放一堆垃圾资产关键字，然后读取，而不是现在这样的写到代码里面。

```
const ALIYUN_FIREWALL_KEYWORD = "阿里云Web应用防火墙"
```

看起来贼傻

这里给出改过后的代码：

```
package scanner

import (
    "fmt"
    "github.com/lcvvvv/gonmap"
    "net"
    "strings"
    "sync"
)

type foo1 struct {
    addr net.IP
    num  int
}

const ALIYUN_FIREWALL_KEYWORD = "阿里云web应用防火墙"

var blockedIPs sync.Map


type PortClient struct {
    *client

    HandlerClosed     func(addr net.IP, port int)
    HandlerOpen       func(addr net.IP, port int)
    HandlerNotMatched func(addr net.IP, port int, response string)
    HandlerMatched    func(addr net.IP, port int, response *gonmap.Response)
    HandlerError      func(addr net.IP, port int, err error)
}

func (client *PortClient) isAliyunFirewallDetected(response string) bool {
    return strings.Contains(response, ALIYUN_FIREWALL_KEYWORD)
}

func NewPortScanner(config *Config) *PortClient {
    var client = &PortClient{
        client:            newConfig(config, config.Threads),
        HandlerClosed:     func(addr net.IP, port int) {},
        HandlerOpen:       func(addr net.IP, port int) {},
        HandlerNotMatched: func(addr net.IP, port int, response string) {},
        HandlerMatched:    func(addr net.IP, port int, response *gonmap.Response)
{},
        HandlerError:      func(addr net.IP, port int, err error) {},
    }
        client.pool.Interval = config.Interval
        client.pool.Function = func(in interface{}) {
        value := in.(foo1)

        nmap := gonmap.New()
        nmap.SetTimeout(config.Timeout)
        if config.DeepInspection == true {
            nmap.OpenDeepIdentify()
        }
```

```
        status, response := nmap.ScanTimeout(value.addr.String(), value.num,
100*config.Timeout)

            if response != nil &&  client.isAliyunFirewallDetected(response.Raw)
{
            fmt.Printf("Detected Aliyun Firewall at %s, skipping the rest of the
scan\n", value.addr)
            blockedIPs.Store(value.addr.String(), true)  // Mark this IP as
blocked.
            return
        }

        switch status {
        case gonmap.Closed:
            client.HandlerClosed(value.addr, value.num)
        case gonmap.Open:
            client.HandlerOpen(value.addr, value.num)
        case gonmap.NotMatched:
            client.HandlerNotMatched(value.addr, value.num, response.Raw)
        case gonmap.Matched:
            client.HandlerMatched(value.addr, value.num, response)
        }
    }
    return client
}

func isBlockedIP(ip string) bool {
    _, blocked := blockedIPs.Load(ip)
    return blocked
}

func (c *PortClient) Push(ip net.IP, num int) {
    if isBlockedIP(ip.String()) {
        fmt.Printf("IP %s is blocked by Aliyun Firewall, skipping...\n", ip)
        return
    }
    c.pool.Push(foo1{ip, num})
}
```

然后这个程序其实还有优化空间。

一样的用全局变量做进程间通信，然后加一个端口开放度的启发式扫描，扫出来有问题的ip就全局同步，然后批量干掉，可以极大的提升扫描速度。

同样程序还存在bug，就是生成csv的时候，会报错。

估计是因为存在大量被skipping的东西，导致csv写入格式报错。

这个后续再想办法解决吧。

---20231011更新---

继续就上面的问题做修改

关于启发式扫描的修改

这里我以端口的开放程度来判断，即一个ip端口开放超过50个，那么就舍弃该ip的扫描，并且同步给其他协程。

首先还是一样，添加一个sync.map

```
133     var openPortCount sync.Map // // Used to count open ports per IP
```

然后增加一个计数器函数

```
149 func (client *PortClient) incrementOpenPortCount(ip string) bool {
150     count, _ := openPortCount.LoadOrStore(ip, value: 1)
151     newCount := count.(int) + 1
152     openPortCount.Store(ip, newCount)
153 
154     if newCount > 50 {
155         blockedIPs.Store(ip, value: true)
156         fmt.Printf( format: "IP %s has over 50 open ports, considered as a honeypot or firewa
157         return true
158     }
159     return false
160 }
```

这里的意思就是，不同协程扫出来的同一个ip的端口都是共享的。

一旦对应ip的端口计数超过50，就通知所有协程ban掉这个ip。

所以后面的函数不用变

```
208 func isBlockedIP(ip string) bool {
209     _, blocked := blockedIPs.Load(ip)
210     return blocked
211 }
212 
213 func (c *PortClient) Push(ip net.IP, num int) {
214     if isBlockedIP(ip.String()) {
215         fmt.Printf( format: "IP %s is blocked, skipping...\n", ip)
216         return
217     }
218     c.pool.Push(foo1{ addr: ip,  num: num})
219 }
```

然后这样就修改完成了。

关于增加外部黑名单文件读取的修改

这里增加了一个读取函数，一个判断函数

```go
// 从外部加载黑名单的文件
func loadBlacklistedKeywords(filename string) error {
    file, err := os.Open(filename)
    if err != nil {
        return err
    }
    defer file.Close()

    decoder := json.NewDecoder(file)
    var data struct {
        Keywords []string `json:"keywords"`
    }
    if err := decoder.Decode(&data); err != nil {
        return err
    }

    blacklistedKeywords = data.Keywords
    return nil
}

func (client *PortClient) isBlacklisted(response string) bool {
    for _, keyword := range blacklistedKeywords {
        if strings.Contains(response, keyword) {
            return true
        }
    }
    return false
}
```

然后增加了一个外部加载函数容错逻辑

```go
func NewPortScanner(config *Config) *PortClient {
    // 从外部加载黑名单文件，限定文件名
    err := loadBlacklistedKeywords( filename: "blacklist_keywords.json")
    if err != nil {
        // Handle error
        fmt.Printf( format: "Error loading blacklisted keywords: %s\n", err)
    }
```

最后黑名单判断逻辑也是一样的

```go
        if response != nil && client.isBlacklisted(response.Raw) {
            fmt.Printf( format: "Detected blacklisted response at %s, skipping the rest of the scan\n", value.addr)
            blockedIPs.Store(value.addr.String(), value: true) // Mark this IP as blocked.
            return
        }
```

然后就修改完毕了

最后还有个问题，就是改完之后，生成csv会有问题。

即最后生成的东西不完整。

搞不动了，搞累了，后续再说，留个坑在这里。

为了解决这个问题，最后可以选择不输出csv，输出txt就行。