个人武器开发

国内项目地址：https://gitee.com/whoisDhan/LoveTools
国外项目地址：https://github.com/whoisdhan/LoveTools

这里可以开始给你第一个安全工具起名字了，_____ <- 名字留给各位师傅替自己填
注：本章节的所有代码放github仓库中，或者可以到文章最结尾复制全部源码，作者建议各位还是去github上直接下载整个项目来学习，不用自己创建了
（如果喜欢的话可以顺手点一个免费的star~后续我肯定会出一些安全工具的，各位师傅可以持续关注我的公众号！！！）

---

tips:
由于代码我在最后才放，所以这里先告诉各位师傅我存了两个全局参数在root.go中，target是通用的，proxy也是通用的，同时target目标可以用逗号分隔开作为多个目标传递，因为我使用了StringSliceVarP函数接收参数，OK交代完毕。

美观输出

先对后面的终端输出简单的美观装饰一下，装b必备，学了安全开发可以给不懂安全的朋友装一下多是一件美事。

- 设置表头：SetHeader(v[0])
- 启用边框：SetBorder(true)
- 启用行分隔线：SetRowLine(true)
- 开启自动换行：SetAutoWrapText(true)
- 自定义中心分隔符：SetCenterSeparator("+") // 即每一个列之间的分隔符
- 自定义列分隔符：SetColumnSeparator("|")
- 自定义行分隔符：SetRowSeparator("-")

当我们传入数据调用函数的时候

```
util.PrettyPrint([][]string{
    {"Name", "Age", "City"},
    {"zhangsan", "18", "shanghai"},
})
```

输出如下图所示：

```
+----------+-----+----------+
|   NAME   | AGE |   CITY   |
+----------+-----+----------+
| zhangsan |  18 | shanghai |
+----------+-----+----------+
```

函数代码如下：

```
func PrettyPrint(v [][]string) {

    table := tablewriter.NewWriter(os.Stdout)

    table.SetHeader(v[0])          // 设置表头

    table.SetBorder(true)          // 启用边框

    table.SetRowLine(true)         // 启用行分隔线

    table.SetAutoWrapText(true)    // 开启自动换行

    table.SetCenterSeparator("+") // 自定义中心分隔符，即每一个列之间的分隔符

    table.SetColumnSeparator("|") // 自定义列分隔符

    table.SetRowSeparator("-")     // 自定义行分隔符



    // 添加数据

    if len(v) == 0 {

        //如果没有数据，直接返回

        table.Render()

        return
```

```
    }
    for _, row := range v[1:] {
        // 这里v[1:]从第二行开始添加数据，第一行是表头，因为表头已经设置好了
        table.Append(row)
    }
    table.Render() // 将结果 prettily 打印到标准输出
    fmt.Println("")
}
```

Whois查询

下载

```
go get -u github.com/likexian/whois
```

简单测试一个域名：

```
t := "baidu.com"
tmp, err := whois.Whois(t)

if err != nil {

    panic(err)

}
fmt.Println(res)
```

这个是最简单的用法，然后为了接收返回的结果，我创建了一个结构体来接收

```go
// 定义结构化结果
Codeium: Refactor | Explain
type DomainInfo struct {
    Domain      string
    Registrar   string   // 注册商
    CreatedDate string   // 创建时间
    ExpiryDate  string   // 过期时间
    NameServers []string // DNS服务器
    Registrant  string   // 注册人/组织
    Status      []string // 域名状态
    UpdatedDate string   // 最后更新时间
    DNSSEC      string   // DNSSEC状态
    Emails      []string // 关联邮箱（钓鱼检测用）
}
```

但是whois返回的是整个查询结果的原生字符串，所以只能够自己写正则匹配来提取内容了
代码较多，就发核心代码截图来讲解（所有源码在结尾）

util文件的一些函数

- 加载动画

```go
// ShowLoading 显示加载动画
Codeium: Refactor | Explain | ×
func ShowLoading(stopChan chan bool) {
    frames := []string{"|", "/", "-", "\\"}
    i := 0
    for {
        select {
        case <-stopChan:
            fmt.Printf("\r\033[K") // Clear the line
            return
        default:
            fmt.Printf("\r[%s] Scanning...", frames[i])
            time.Sleep(100 * time.Millisecond)
            i = (i + 1) % len(frames)
        }
    }
}
```

- whois主要功能
  其中包含了加载动画、正则提取、打印结果

```go
Codeium: Refactor | Explain | Generate GoDoc | ×
func whoisSearch(targets []string) {
    stopChan = make(chan bool) // 初始化停止加载动画的通道
    // 启动加载动画的协程
    go util.ShowLoading(stopChan)
    var res []string
    for _, t := range targets {
        tmp, err := whois.Whois(t)
        if err != nil {
            panic(err)
        }
        res = append(res, tmp)

    }
    // 任务完成，停止加载动画
    stopChan <- true
    // 解析WHOIS数据
    infos := parseWhois_s(res)
    // 打印结果
    printDomainInfos(infos)
}
```

- 运行结果

域名	注册商	创建时间	过期时间	DNS服务器	注册人	最后更新时间	关联邮箱
BAIDU.COM	MarkMonitor Inc.	1999-10-11 11:05:17 UTC	2026-10-11 11:05:17 UTC	NS1.BAIDU.COM, NS2.BAIDU.COM, NS3.BAIDU.COM, NS4.BAIDU.COM, NS7.BAIDU.COM, ns4.baidu.com, ns7.baidu.com, ns1.baidu.com, ns2.baidu.com, ns3.baidu.com	Beijing Baidu Netcom Science Technology Co., Ltd.	nil	abusecomplaints@markmonitor.com, abusecomplaints@markmonitor.com, whoisrequest@markmonitor.com

更多函数细节不用深究，后面我会给出所有源码，而且这些函数功能其实用ai也能写出来，不用造轮子。

反查ip

最终效果

目标	反查结果	归属地
[redacted]		中国 香港

自己在https://site.ip138.com/网站上找ip进行反查作为例子即可，很多ip都可以反查到域名

这功能在https://site.ip138.com/网站请求

- 核心函数`iprSearch`：
  负责对单个域名进行ip反查

```go
func iprSearch(target string) []string {
    client := req.C().
        SetUserAgent("Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/134.0.0.0 Safari/537.36")
    if proxy != "" {
        client.SetProxyURL(proxy) // 设置代理
    }
    //反查结果
    var (
        domain  string
        address string
    )

    data, _ := client.R().
        SetPathParam("target", target).
        Get("https://site.ip138.com/{target}")
    doc, _ := goquery.NewDocumentFromReader(data.Body)
    doc.Find("#list > li:nth-child(3) > a").Each(func(i int, s *goquery.Selection) { //#list > li:nth-child(3) > a
        domain = strings.TrimSpace(s.Text()) //域名结果赋值
    })
    doc.Find("h3").Each(func(i int, s *goquery.Selection) { //#list > li:nth-child(3) > a
        address = strings.TrimSpace(s.Text()) //域名结果赋值
    })

    return []string{target, domain, address}
}
```

- `iprsSearch`，ipr后多了一个s区分，用来对域名列表进行ip反查

```
Codeium: Refactor | Explain | Generate GoDoc | ×
func iprsSearch(targets []string) [][]string {
    res := [][]string{
        {"目标", "反查结果", "归属地"}, // 表头
    }
    for _, target := range targets {
        tmp := iprSearch(target) // 反查IP地址
        res = append(res, tmp)   // 将结果添加到结果列表中
    }
    return res
}
```

- `ipr`最终调用功能函数

```
// 反查IP地址调用函数
Codeium: Refactor | Explain | ×
func ipr() {
    stopChan = make(chan bool)    // 创建一个通道用于停止加载动画
    go util.ShowLoading(stopChan) // 启动加载动画的协程
    var res [][]string
    res = iprsSearch(targets) // 反查IP地址
    stopChan <- true          // 停止加载动画
    util.PrettyPrint(res)     // 打印结果
}
```

- 运行结果

```
+-------------+-------------+-----------+
|    目标     |  反查结果   |  归属地   |
+-------------+-------------+-----------+
|             | minkeji.net | 中国 香港 |
+-------------+-------------+-----------+
```

目录扫描

练习项目我们这里就直接用字典扫url一层目录即可

- 2xx 状态码：绿色
- 3xx 状态码：橙色
- 4xx 状态码：蓝色
- 5xx 状态码：红色
  最终运行效果如下：

（我们仅仅做于学习，并没有对服务器造成影响，扫了几个请求而已哈~）

```
http://baidu.com/docs/PCRE.LICENCE      302
http://baidu.com/docs/README    302
http://baidu.com/docs/zlib.LICENSE      302
http://baidu.com/html   302
http://baidu.com/html   302
http://baidu.com/html/50x.html  302
http://baidu.com/html/index.html        302
http://baidu.com/index.html     200
http://baidu.com/logs   302
http://baidu.com/logs   302
http://baidu.com/nginx.exe      302
http://baidu.com/temp   302
http://baidu.com/temp   302
```

细节：

- 目录扫描要设置禁止自动跟踪跳转，否则服务器有完整的302跳转的话，你扫出来的都是200
- 超时时间建议也带上，可控性强
- 这里其实还可以加一个延迟请求效果，比如每一个请求之间间隔多久，否则发送太快容易被服务端封禁（这里留着可以给感兴趣的师傅自己写）
- 线程设置，这里不做实现（×）
- 我们scanner读取每一行的时候，由于可能用户会给多个目标，所以我们的字典要指针要回到头部进行重新读取，seek需要第二层循环完成字典后记得将指针指向头部

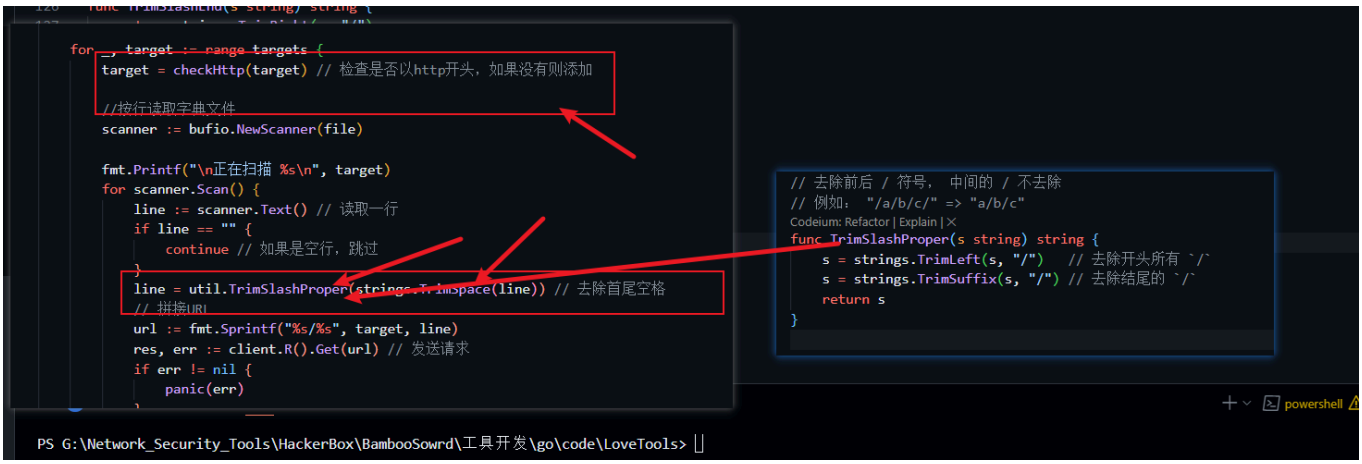
```go
func dirScan(targets []string, dict string) {
    stopChan = make(chan bool)    // 创建一个通道用于停止加载动画
    go util.ShowLoading(stopChan) // 启动加载动画的协程
    client := req.C().
        SetRedirectPolicy(func(req *http.Request, via []*http.Request) error {
            return http.ErrUseLastResponse
        }).                                                 // 禁止自动跳转
        SetTimeout(time.Duration(dirTimeOut) * time.Second) // 设置超时时间

    if proxy != "" {
        client.SetProxyURL(proxy) // 设置代理
    }

    file, err := os.Open(dict) // 打开字典文件
    defer file.Close()         // 延迟关闭文件
    if err != nil {
        fmt.Println("打开字典文件失败:", err)
        return
    }
```

对url进行清洗

- checkHttp：检查是否是http开头，因为有的用户可能会直接给一个域名

```go
// 判断是否存在http，不存在就默认加上http
Codeium: Refactor | Explain | ×
func checkHttp(target string) string {
    if !strings.HasPrefix(target, "http") {
        return "http://" + target
    }
    return target
}
```

- 去除多余空格
- 去除多余 `/`



对不同状态码之间的请求路径上色

- 这里判断一下状态码的范围即可，注意我第一个用了 `IsSuccessState`，他就是判断200~299之间的，后面就没有了，只剩下一个 `IsErrorState`，他是大于400就true，不符合我们的要求

```
func (r *req.Response) IsErrorState() bool
```

IsErrorState method returns true if no error occurs and HTTP status `code >= 400` by default, you can also use Request.SetResultStateCheckFunc to customize the result state check logic.

(req.Response).IsErrorState on pkg.go.dev

```go
        //根据状态码输出不同颜色的结果
        var path string // 定义路径变量
        if res.IsSuccessState() {
            path = fmt.Sprintf("\r\033[K%s\t%d\n", url, res.StatusCode)
            path = color.GreenString(path) // 成功状态码，绿色输出
        }
        if res.StatusCode >= 300 && res.StatusCode < 400 {
            path = fmt.Sprintf("\r\033[K%s\t%d\n", url, res.StatusCode)
            path = color.BlueString(path) // 重定向状态码，蓝色输出

        }
        if res.StatusCode >= 400 && res.StatusCode < 500 {
            path := fmt.Sprintf("\r\033[K%s\t%d\n", url, res.StatusCode)
            path = color.YellowString(path) // 客户端错误，黄色输出
        }
        if res.StatusCode >= 500 && res.StatusCode < 600 {
            path := fmt.Sprintf("\r\033[K%s\t%d\n", url, res.StatusCode)
            path = color.RedString(path) // 服务器错误，红色输出
        }
        fmt.Print(path) // 打印路径
    }
```

一个目标完成扫描后，我们的字典到尾部了，所以我们需要将这个指针指向头部重新给下一个目标扫描目录

```go
    file.Seek(0, 0) // 重置文件指针到开头
    if err := scanner.Err(); err != nil {
        fmt.Println("Error:", err) // 处理扫描错误
    }
}
```

最后给命令Run给上运行逻辑和init上添加子命令和对应的参数即可

```go
var dirS = &cobra.Command{
    Use:   "dir",
    Short: "目录扫描",
    Long:  `目录扫描,自定义字典`,
    Run: func(cmd *cobra.Command, args []string) {
        if len(targets) == 0 {
            cmd.Help() // 显示帮助信息
            return
        }

        dirScan(targets, dict) // 执行目录扫描

    },
}

Codeium: Refactor | Explain | Generate GoDoc | ×
func init() {
    rootCmd.AddCommand(dirS) // 将 dirS 命令添加到根命令中
    dirS.Flags().StringVarP(&dict, "dict", "d", "dict.txt", "字典文件")
    dirS.Flags().IntVarP(&dirTimeOut, "timeout", "m", 5, "超时时间") // 设置超时时间
}
```

最后就是运行`go run main.go -t xxx.com,aaa.ccc`,多个目标可以用逗号隔开
(仅仅做学习，不要对服务器造成影响)
这里还有几个状态码效果颜色需要找到对应的服务器返回状态码才行，这里就忽略了，能够打印颜色就表示成功了。

```
[|] Scanning...
正在扫描 http://baidu.com
http://baidu.com/50x.html       302
http://baidu.com/conf   302
http://baidu.com/conf   302
http://baidu.com/conf/fastcgi_params    302
http://baidu.com/conf/fastcgi.conf      302
http://baidu.com/conf/koi-utf   302
http://baidu.com/conf/koi-win   302
http://baidu.com/conf/mime.types        302
http://baidu.com/html   302
http://baidu.com/html/50x.html  302
http://baidu.com/html/index.html        302
http://baidu.com/index.html     200
http://baidu.com/logs   302
http://baidu.com/logs   302
http://baidu.com/nginx.exe      302
http://baidu.com/temp   302
http://baidu.com/temp   302
```

子域名爆破

这里有一个细节要注意：

- 由于子域名爆破中主动扫描要用到dict字典，之前在目录扫描中也有一个字典，为了方便，使用同一个参数就合并起来了，放到root中作为全局参数
  `同时为了更容易区分和防止子命令撞参数就把全局的短选项参数更改为大写`

```
Codeium: Refactor | Explain | Generate GoDoc | ×
func init() {
    rootCmd.PersistentFlags().StringSliceVarP(&targets, "target", "T", nil, "目标域名或IP") // 添加目标参数
    rootCmd.PersistentFlags().StringVarP(&proxy, "proxy", "P", "", "代理地址")              // 添加代理参数
    rootCmd.PersistentFlags().StringVarP(&dict, "dict", "F", "dict.txt", "字典文件")       // 添加字典参数
}
```

- 同时也在root添加了yaml配置文件变量名

```
var (
    targets  []string                  // 目标列表，可以是域名或 IP
    proxy    string                    // 代理地址
    stopChan chan bool                 // 停止加载动画的通道
    dict     string                    // 字典文件
    yamlPath string    = "config.yaml" // yaml配置文件
)
```

- 被动扫描用的是subfinder提供出来的sdk进行开发，下载的时候需要注意go get是否能下载到，下载不了可能你是更改成为了其他加速的地址需要更改回来官方的：
  具体要看你能不能下载，下载失败就要更改回官方的
  `go env -w GOPROXY=https://goproxy.io,direct` <- 这是官方的，之前可能你更改了阿里云或者其他国内加速地址

被动扫描

使用subfinder的核心接口sdk，有官方使用代码例子：
`https://github.com/projectdiscovery/subfinder/blob/dev/v2/examples/main.go`

我这里写了三个函数，将官方示例代码小小的拆开了几部分

```go
// 被动扫描
// subfinder 提供的 SDK 接口主要是被动枚举子域名，不是通过传统的字典爆破来发现子域名。
// 因此，这里的 subdomain 函数只是简单地调用 subfinder 的 EnumerateSingleDomainWithCtx 函数来进行子域名爆破。
Codeium: Refactor | Explain | ×
func subDomainFinder(target string) map[string]map[string]struct{} {···
}

// 被动扫描多个域名
// subDomainsFinder 函数负责调用 subDomainFinder 函数进行多个目标子域名爆破
Codeium: Refactor | Explain | ×
func subDomainsFinder(results []string) {···
}

// 打印被动扫描的结果
Codeium: Refactor | Explain | ×
func subDomainPrint(sourceMap map[string]map[string]struct{}) {···
}
```

- `subDomainFinder`：扫描单个域名
  0.写一个 `&runner.Options` 结构体
  1. `NewRunner` 创建一个runner
  2. `&bytes.Buffer{}` 缓冲区
  3.使用 `EnumerateSingleDomainWithCtx` 进行被动扫描，参数按照官方给的用就行，这些代码都是官方上拿的
  4.改动：我将结果作为函数返回值，因为我们是对多个目标进行扫描，这里单个结果返回即可

```go
// 被动扫描
// subfinder 提供的 SDK 接口主要是被动枚举子域名，不是通过传统的字典爆破来发现子域名。
// 因此，这里的 subdomain 函数只是简单地调用 subfinder 的 EnumerateSingleDomainWithCtx 函数来进行子域名爆破。
Codeium: Refactor | Explain | ×
func subDomainFinder(target string) map[string]map[string]struct{} {
    subfinderOpts := &runner.Options{
        Threads:            10, // 设置线程数
        Timeout:            30, // 设置超时时间
        MaxEnumerationTime: 10, // 设置最大枚举时间

    }
    subfinder, err := runner.NewRunner(subfinderOpts)
    if err != nil {
        log.Fatalf("failed to create subfinder runner: %v", err)
    }
    output := &bytes.Buffer{} //接收结果的缓冲区
    var sourceMap map[string]map[string]struct{}
    // 枚举单个域名
    // 这里的 io.Writer 可以是任何实现了 io.Writer 接口的对象，例如文件、缓冲区等
    if sourceMap, err = subfinder.EnumerateSingleDomainWithCtx(context.Background(), target, []io.Writer{output}); err != nil {
        log.Fatalf("failed to enumerate single domain: %v", err)
    }
    // 测试代码，打印结果
    // log.Println(output.String())

    return sourceMap

}
```

- 对多个目标进行扫描，就是结合单个域名扫描那里进行包装循环

```go
// 被动扫描多个域名
// subDomainsFinder 函数负责调用 subDomainFinder 函数进行多个目标子域名爆破“
Codeium: Refactor | Explain | ×
func subDomainsFinder(results []string) {
    for _, target := range targets {
        //每一个域名都开启爆破动画加载
        stopChan = make(chan bool) // 初始化停止加载动画的通道
        go util.ShowLoading(stopChan)
        sourceMap := subDomainFinder(target) // 枚举子域名
        subDomainPrint(sourceMap)
        //有一个域名扫描任务完成，停止加载动画
        stopChan <- true
    }
}
```

- 打印结果：没啥好说的，就看你自己需求，我这里就直接用官方给的打印方式了，唯一不同就是由于我们有加载动画，首先不能让加载动画覆盖我们的域名打印结果，所以我们在打印结果之前进行清行：`\r\033[K`，当然

```go
// 打印被动扫描的结果
Codeium: Refactor | Explain | ×
func subDomainPrint(sourceMap map[string]map[string]struct{}) {
    // 遍历 sourceMap，打印每个子域名和对应的源
    for subdomain, sources := range sourceMap {
        sourcesList := make([]string, 0, len(sources))
        for source := range sources {
            sourcesList = append(sourcesList, source)
        }
        fmt.Printf("\r\033[K%s %s (%d)\n", subdomain, sourcesList, len(sources))
    }
}

Codeium: Refactor | Explain | Generate GoDoc | ×
```

主动扫描(字典爆破)

主动扫描意思是使用字典进行爆破，所以自己就能写，可以不用subfinder的代码，使用`lookup`就能判断域名是否有效。

我这里用了一个函数一个结构体

```go
// 由于lookup返回的结果有ip，
// 需要定义一个结构体来存储结果对应的子域名，
// 以后要用的时候方便取ip
Codeium: Refactor | Explain
type LookupResult struct {
    Subdomain string
    IP        []string
}

// 主动扫描
Codeium: Refactor | Explain | X
func bruteSubdomains(targets []string) []LookupResult {…
}
```

- 结构体`LookupResult`：
  用来存储域名和对应的ip，因为他`lookup`会返回一个lookup的ip地址列表
- `bruteSubdomains`：go协程、加锁、文件读取、yaml文件解析都用上了，这里学习了一个新的解析格式`yaml`，建议自己去看https://www.runoob.com/w3cnote/yaml-intro.html菜鸟这篇文章，短小精悍，一看就懂。

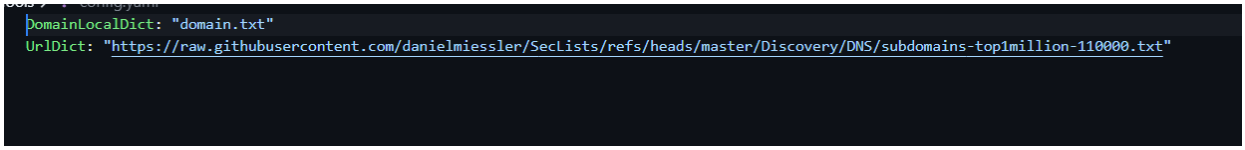
```go
func bruteSubdomains(targets []string) []LookupResult {
    var (
        wg      sync.WaitGroup //定义一个等待组
        mutex   sync.Mutex     //定义一个互斥锁
        results []LookupResult //变量写在这里，我们的scanDomain就直接进行添加操作最方便简洁
    )

    //域名nslookup
    var scanDomain = func(domain string) {
```

  - 单个域名解析：`scanDomain`函数变量
    这里还涉及到多个协程操作一个`results`列表变量的问题，所以需要用到互斥锁，否则会出现不同步的问题，可能会导致死锁。
    还有一个就是ip显示还是不显示的问题，这个也提取出来作为一个可选参数，毕竟我们

爆破域名的时候也想看看ip（默认是不开启）

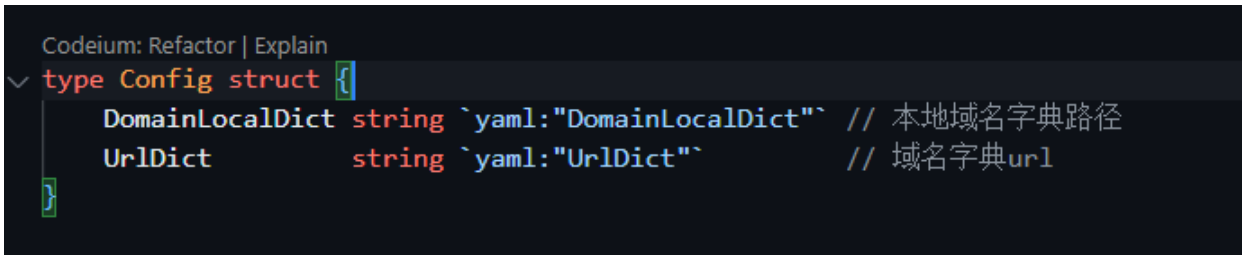
```go
//域名nslookup
var scanDomain = func(domain string) {
    defer wg.Done()
    var err error
    res := LookupResult{Subdomain: domain}
    res.IP, err = net.LookupHost(domain)

    if err != nil {
        //表示子域名不存在，直接返回
        return
    }
    //否则就打印出来

    if showIP {
        fmt.Printf("\r\033[K%s: %v\n", res.Subdomain, strings.Join(res.IP, ",")) // 打印子域名和对应的ip列表
    } else {
        fmt.Printf("\r\033[K%s\n", res.Subdomain)
    }
    //加锁，因为要对results同一个蛋糕进行操作
    //所以会出现不同步的问题，可能会导致死锁
    //所以需要加锁
    mutex.Lock()
    results = append(results, res)
    mutex.Unlock()
}
```

- 为了更加定制化，所以我将url字典提取出来了，放到了yaml文件中，方便以后url字典能够在配置文件更换



```yaml
DomainLocalDict: "domain.txt"
UrlDict: "https://raw.githubusercontent.com/danielmiessler/SecLists/refs/heads/master/Discovery/DNS/subdomains-top1million-110000.txt"
```

- 这里需要添加一个config.go文件存储结构体



```go
Codeium: Refactor | Explain
type Config struct {
    DomainLocalDict string `yaml:"DomainLocalDict"` // 本地域名字典路径
    UrlDict         string `yaml:"UrlDict"`         // 域名字典url
}
```

- 接着就是写解析`yaml`函数，其实就是之前学的导出文件中的函数一样，只不过函数类别换成了yaml来调用

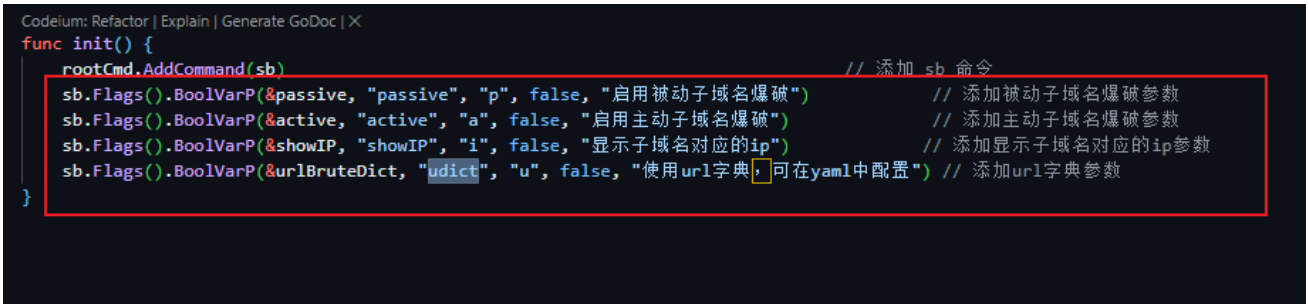
```go
Codeium: Refactor | Explain | Generate GoDoc | ×
func newConfig() *Config {
    return &Config{}
}

// 解析yaml，返回Config结构体指针
Codeium: Refactor | Explain | ×
func ParseConfig(path string) *Config {
    var err error
    var fileinfo os.FileInfo
    fileinfo, err = os.Stat(path)
    if err != nil {
        fmt.Println("找不到文件")
    }
    if fileinfo.IsDir() {
        fmt.Println("输入的路径是文件夹")
    }
    yamlFile, _ := os.ReadFile(path)

    config := newConfig()
    err = yaml.Unmarshal(yamlFile, config)
    if err != nil {
        fmt.Println("yaml文件解析失败")
    }
    return config
}
```

- 因为要用 `bufio.NewScanner`，在util.go文件中添加一个函数，所以要读取url返回一个 `io.Reader` 类型，那么就直接封装到 `util.go` 文件中使用了

```go
// 转为file一样正常给scanner读取
Codeium: Refactor | Explain | ×
func LoadUrlDict(url string) io.Reader {
    resp, err := http.Get(url)
    if err != nil {
        fmt.Println("请求失败url字典")
        os.Exit(1)
    }
    defer resp.Body.Close()
    // 读取 Body 数据
    data, err := io.ReadAll(resp.Body)
    if err != nil {
        fmt.Println("读取失败")
        os.Exit(1)
    }

    // 用 bytes.Reader 返回
    return bytes.NewReader(data)
}
```

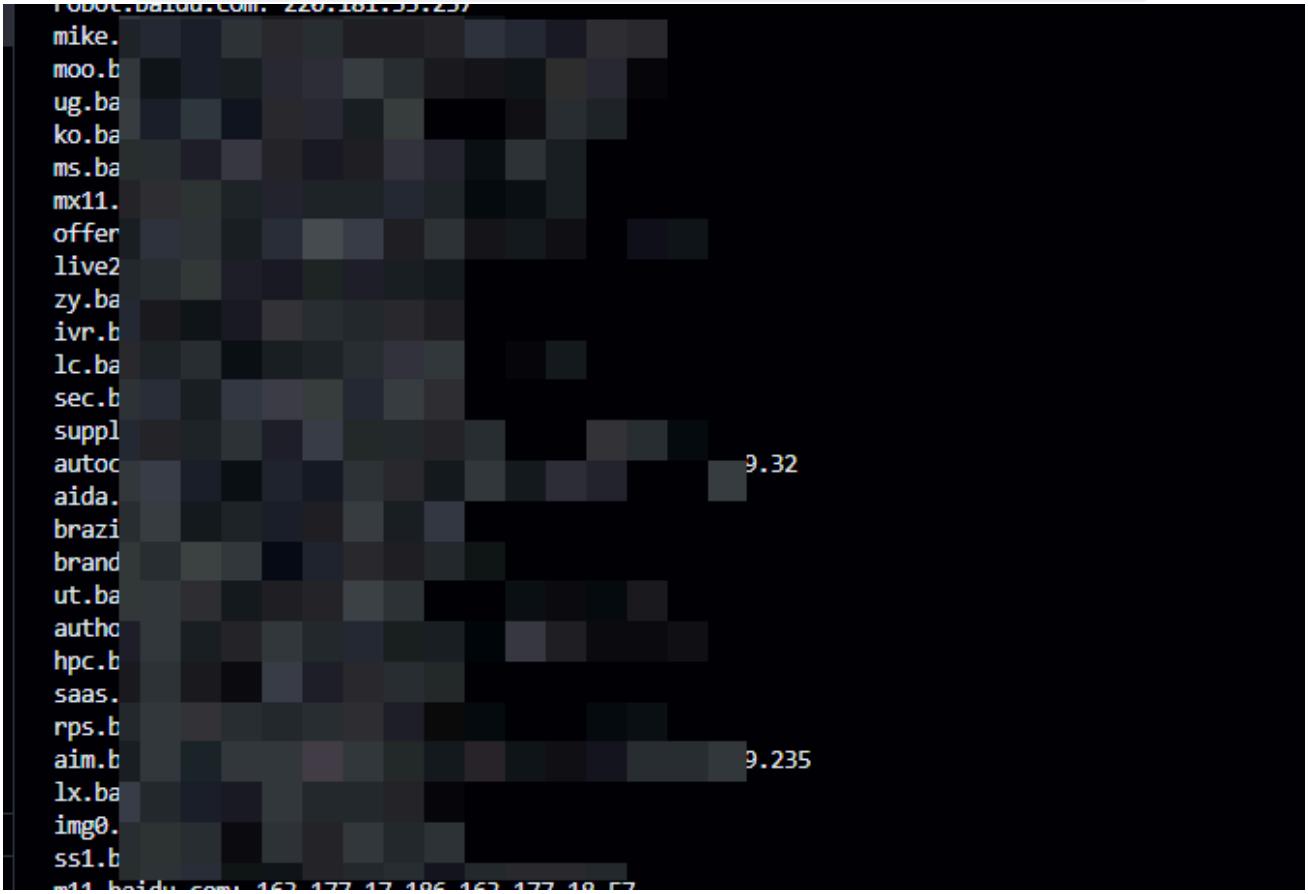
- 本地扫描就忽略了，因为就是一个读取文件一行一行拼接就行，下面我直接放剩下的代码截图

- 参数添加如下
  所以我们可以使用的命令组合有：
  `subdomain -a true -T xxx.cxm` 默认使用url字典主动扫描
  `subdomain -a true -F dict.txt -T xxx.cxm` 使用字典路径扫描
  `subdomain -b true -T xxx.cxm` 被动扫描
  `subdomain -T xxx.cxm` 两个都是false的时候，默认就是被动扫描，函数代码中有进行判断两个false
  `subdomain -i true -T xxx.cxm` 加上-i组合就更多了，自己测试即可



这里贴一个运行结果

命令是：`go run main.go subdomain -a true -u true -T baidu.com -i true`



CDN检测

检测最终结果，仅做学习用途，无非法动作。

	目标	IP数量	检测节点	IP列表	检测结果
1	b	2	中国-北京	39.1	检测成功
2	b	2	新加坡	110.	检测成功
3	b	2	欧洲-伦敦	110.	检测成功
4	b	2	欧洲-法兰克福	110.	检测成功
5	b	2	中国-成都	39.1	检测成功
6	b	2	中国-张家口	110.	检测成功
7	b	2	美国-硅谷	110.	检测成功
8	b	2	印度尼西亚-雅加达	110.	检测成功
9	b	2	印度尼西亚-雅加达	39.1	检测成功

- yaml文件添加cdn检测节点
  节点检测方式：`CDNURL/?domain=baidu.com` -> 响应返回用逗号隔开的ip列表(检测成功) 或者检测失败

```yaml
CDNList:
  "美国-硅谷":
    - "https://ips-app-vrdhcyxprn.us-west-1.fcapp.run"
  "新加坡":
    - "https://ips-app-vrdhcyxprn.ap-southeast-5.fcapp.run"
  "欧洲-伦敦":
    - "https://ips-app-vrdhcyxprn.eu-west-1.fcapp.run"
  "欧洲-法兰克福":
    - "https://ips-app-vrdhcyxprn.eu-central-1.fcapp.run"
  "印度尼西亚-雅加达":
    - "https://ips-app-vrdhcyxprn.ap-southeast-7.fcapp.run"
    - "https://ips-app-vrdhcyxprn.ap-southeast-1.fcapp.run"
    - "https://ips-app-vrdhcyxprn.ap-southeast-3.fcapp.run"
  "中国-深圳":
    - "https://ips-app-nnrrqmtriz.cn-shenzhen.fcapp.run"
  "中国-成都":
    - "https://ips-app-vrdhcyxprn.cn-chengdu.fcapp.run"
  "中国-杭州":
    - "https://ips-app-nnrrjaztiz.cn-hangzhou.fcapp.run"
  "中国-张家口":
    - "https://ips-app-vrdhcyxprn.cn-zhangjiakou.fcapp.run"
  "韩国-首尔":
    - "https://ips-app-vrdhcyxprn.ap-northeast-2.fcapp.run"
  "中国-北京":
    - "https://ips-app-nnrrjaztiz.cn-beijing.fcapp.run"
  "中国-呼和浩特":
    - "https://ips-app-vrdhcyxprn.cn-huhehaote.fcapp.run"
  "中国-香港":
    - "https://ips-app-nnrrjaztiz.cn-hongkong.fcapp.run"
```

- config.go文件添加属性

```go
type Config struct {
    DomainLocalDict string              `yaml:"DomainLocalDict"` // 本地域名字典路径
    UrlDict         string              `yaml:"UrlDict"`         // 域名字典url
    CDNList         map[string][]string `yaml:"CDNList"`
}
```

- 创建`cdn.go`文件，分别写了三个函数以及一个结构体去完成这个功能
  - `cdnInfo`就是用来存储一个域名去多个cdn节点检测结果的一个结果合集
  所以除了domain都是string列表类型，看下图抓包就能看到请求的完整路径，所以知道请求路径就知道怎么写代码了。

状态码	URL	相关插件
200	https://ips-app-vrdhcyxprn.ap-southeast-7.fcapp.run/?domain=baid...	-
200	https://ips-app-vrdhcyxprn.ap-southeast-5.fcapp.run/?domain=baid...	-
200	https://ips-app-vrdhcyxprn.ap-southeast-3.fcapp.run/?domain=baid...	-
200	https://ips-app-vrdhcyxprn.eu-west-1.fcapp.run/?domain=baidu.com	-
200	https://ips-app-vrdhcyxprn.eu-central-1.fcapp.run/?domain=baidu.co...	-
200	https://ips-app-vrdhcyxprn.us-east-1.fcapp.run/?domain=baidu.com	-
200	https://ips-app-vrdhcyxprn.us-west-1.fcapp.run/?domain=baidu.com	-
200	https://ips-app-vrdhcyxprn.ap-southeast-1.fcapp.run/?domain=baid...	-
200	https://ips-app-nnrrjaztiz.cn-hangzhou.fcapp.run/?domain=baidu.com	-
200	https://ips-app-nnrrjaztiz.cn-beijing.fcapp.run/?domain=baidu.com	-
200	https://ips-app-vrdhcyxprn.cn-zhangjiakou.fcapp.run/?domain=baid...	-
200	https://ips-app-vrdhcyxprn.cn-huhehaote.fcapp.run/?domain=baidu....	-
200	https://ips-app-vrdhcyxprn.ap-northeast-1.fcapp.run/?domain=baidu...	-
200	https://ips-app-vrdhcyxprn.ap-northeast-2.fcapp.run/?domain=baidu...	-
200	https://ips-app-vrdhcyxprn.cn-chengdu.fcapp.run/?domain=baidu.com	-
200	https://ips-app-nnrrjaztiz.cn-qingdao.fcapp.run/?domain=baidu.com	-
200	https://ips-web-fnrrjazpxz.cn-shanghai.fcapp.run/?domain=baidu.com	-
200	https://ips-app-nnrrqmtriz.cn-shenzhen.fcapp.run/?domain=baidu.co...	-
200	https://ips-app-nnrrjaztiz.cn-hongkong.fcapp.run/?domain=baidu.com	-

```
type cdnInfo struct {
    domain      string
    address     []string
    cdnURLList  []string // CDN URL 列表
    reponseList []string // 响应结果列表
    //CDN URL 和 响应结果 个数是一样的
    ok []string // 检测结果
}

var cdnInfoList []cdnInfo

var cdnCmd = &cobra.Command{ ···
}

func cdnCheck(target string, client *req.Client) cdnInfo { ···
}

func cdns(targets []string) []cdnInfo { ···
}

func printCDNInfos(cdnInfoList []cdnInfo) { ···
}
```

- `cdnCheck`最核心的函数：最终要的是在你`config := util.ParseConfig(yamlPath)`读取配置文件后怎么请求，其实就是很简单，用创建好的client请求，如果状态码200就请求成功，否则就请求失败，在响应结果和相应列表的append中自己修改一下即可。

```
Codeium: Refactor | Explain | Generate GoDoc | ×
func cdnCheck(target string, client *req.Client) cdnInfo {
    var cdnInfo cdnInfo
    config := util.ParseConfig(yamlPath)
    for address, cdnURLS := range config.CDNList {
        for _, cdnURL := range cdnURLS {
            req, _ := client.R().
                SetQueryParam("domain", target).
                Get(cdnURL)
            if req.StatusCode == 200 {
                contentBytes, _ := io.ReadAll(req.Body)
                resultIP := string(contentBytes)
                cdnInfo.cdnURLList = append(cdnInfo.cdnURLList, cdnURL)     //将文件解析出来的cdn也放进去结构体，后面要打印
                cdnInfo.reponseList = append(cdnInfo.reponseList, resultIP) //将一个cdn查询结果传进去
                cdnInfo.ok = append(cdnInfo.ok, "检测成功")
            } else {
                cdnInfo.cdnURLList = append(cdnInfo.cdnURLList, cdnURL)  //将文件解析出来的cdn也放进去结构体，后面要打印
                cdnInfo.reponseList = append(cdnInfo.reponseList, "---") //将一个cdn查询结果传进去
                cdnInfo.ok = append(cdnInfo.ok, "检测失败")
            }
            cdnInfo.domain = target
            cdnInfo.address = append(cdnInfo.address, address)
        }
    }
    return cdnInfo
}
```

  - 在第二层循环的时候记得把`target`和yaml配置中的cdn归属地址`address`添加进`cdnInfo`里面即可

- `cdns函数`：
  就是对多个目标进行遍历给到cdnCheck检测即可

同时记得加上加载动画过程

```go
Codeium: Refactor | Explain | Generate GoDoc | ×
func cdns(targets []string) []cdnInfo {
    stopChan = make(chan bool) // 初始化停止加载动画的通道
    // 启动加载动画的协程
    go util.ShowLoading(stopChan)

    client := req.C()
    if proxy != "" {
        client.SetProxyURL(proxy)
    }
    for _, t := range targets {
        ipList := cdnCheck(t, client)
        cdnInfoList = append(cdnInfoList, ipList)
    }
    // 任务完成，停止加载动画
    stopChan <- true
    return cdnInfoList
}
```

- 打印CDN结果
  双层遍历找到`cdnInfo`的结果列表，因为他就是最深一层，然后按照需求添加进去`res`里面给到`util.PrettyPrint`函数就行(这个函数是之前写的，一直都有用)

```go
Codeium: Refactor | Explain | Generate GoDoc | ×
func printCDNInfos(cdnInfoList []cdnInfo) {
    res := [][]string{
        {" ", "目标", "IP数量", "检测节点", "IP列表", "检测结果"},
    }

    for _, cdnInfo := range cdnInfoList {
        for index, _ := range cdnInfo.cdnURLList { //直接找最深一层，然后将要的东西都传进去即可
            res = append(res, []string{
                strconv.Itoa(index + 1), // 从 1 开始
                cdnInfo.domain,          // 目标
                strconv.Itoa(len(strings.Split(cdnInfo.reponseList[index], ","))), //响应结果通过逗号分隔，长度就是IP数量，然后将int长度转为string
                cdnInfo.address[index],     // 检测节点，即CDN URL
                cdnInfo.reponseList[index], // IP 列表
                cdnInfo.ok[index],          // 检测结果
            })
        }
    }

    util.PrettyPrint(res)
}
```

- Command结构体的Run属性写法：

```go
var cdnCmd = &cobra.Command{
    Use:   "cdn",
    Short: "CDN检测",
    Long:  `CDN检测`,
    Run: func(cmd *cobra.Command, args []string) {
        if len(targets) == 0 {
            cmd.Help() // 显示帮助信息
            return
        }
        cdnInfoList = cdns(targets)
        printCDNInfos(cdnInfoList)
        // cdns(targets) // 执行CDN检测
    },
}
```

- 最终运行结果：
  正在扫描中

结果

	目标	IP数量	检测节点	IP列表	检测结果
1	b[redacted]m	2	中国-北京	39.1[redacted]3.66	检测成功
2	b[redacted]m	2	新加坡	110.[redacted]5.10	检测成功
3	b[redacted]m	2	欧洲-伦敦	110.[redacted]5.10	检测成功
4	b[redacted]m	2	欧洲-法兰克福	110.[redacted]5.10	检测成功
5	b[redacted]m	2	中国-成都	39.1[redacted]3.66	检测成功
6	b[redacted]m	2	中国-张家口	110.[redacted]5.10	检测成功
7	b[redacted]m	2	美国-硅谷	110.[redacted]5.10	检测成功
8	b[redacted]m	2	印度尼西亚-雅加达	110.[redacted]5.10	检测成功
9	b[redacted]m	2	印度尼西亚-雅加达	39.1[redacted]3.66	检测成功