网络 (/tags/#网络)

你也能写个 Shadowsocks

Posted by 吴浩麟 on 2017-11-03

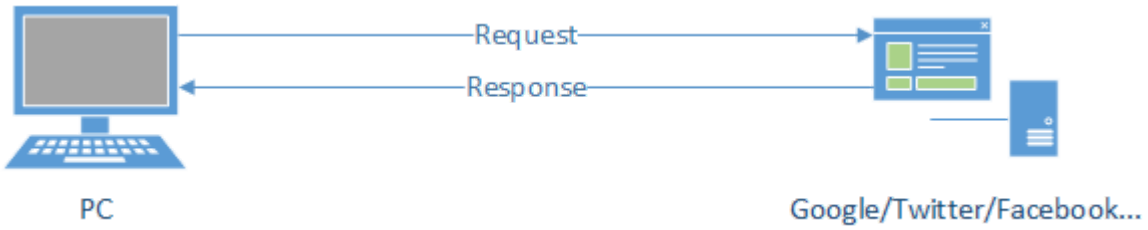
本文将教你从0写一个Shadowsocks (https://github.com/shadowsocks/shadowsocks-go)，无需任何基础，读完本文你就能完成一个轻量级、高性能的 Shadowsocks 代替品。

我们暂且把最终完成的项目叫做 Lightsocks，如果你很急切地想看到结果，可以先体验本文最终完成的项目 Lightsocks (https://github.com/gwuhaolin/lightsocks) ，也可以下载阅读源码。

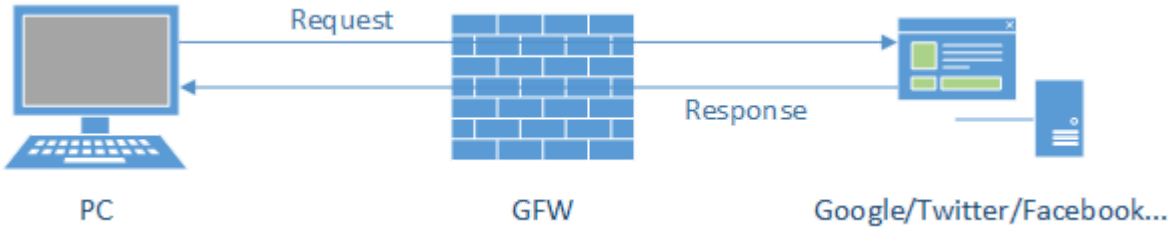
认识 Shadowsocks

Shadowsocks 是一个能骗过防火墙的网络代理工具。它把要传输的原数据经过加密后再传输，网络中的防火墙由于得不出要传输的原内容是什么而只好放行，于是就完成了防火墙穿透，也即是所谓的“翻墙”。

在自由的网络环境下，在本机上访问服务时是直接和远程服务建立连接传输数据，流程如图：

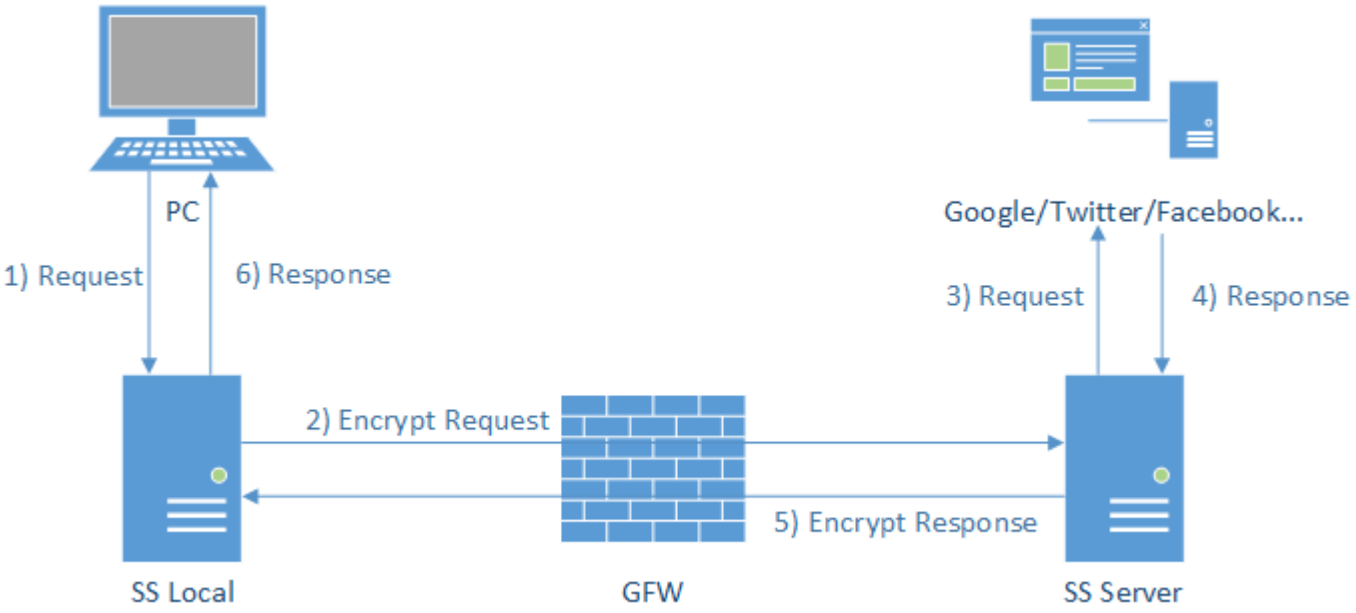


但在受限的网络环境下会有防火墙，本机电脑和远程服务之间传输的数据都必须通过防火墙的检查，流程如图：



如果防火墙发现你在传输受限的内容，就把拦截本次传输，就会导致在本机无法访问远程服务。

而 Shadowsocks 所做的就是把传输的数据加密，防火墙得到的数据是加密后的数据，防火墙不知道传输的原内容是什么，于是防火墙就放行本次请求，于是在本机就访问到了远程服务，流程如图：



也就是说使用 Shadowsocks 的前提是：

- 一台在防火墙之外的服务器；
- 在本机需要安装 Shadowsocks 本地端，用于加密传输数据；
- 服务器需要安装 Shadowsocks 服务端，用于解密加密后的传输数据，解密出原数据后发送到目标服务器。

Shadowsocks 原理

Shadowsocks 由两部分组成，运行在本地的 ss-local 和运行在防火墙之外服务器上的 ss-server，下面来分别详细介绍它们的职责（以下对 Shadowsocks 原理的解析只是我的大概估计，可能会有细微的差别）。

ss-local

ss-local 的职责是在本机启动和监听着一个服务，本地软件的网络请求都先发送到 ss-local，ss-local 收到来自本地软件的网络请求后，把要传输的原数据根据用户配置的加密方法和密码进行加密，再转发到墙外的服务器去。

ss-server

ss-server 的职责是在墙外服务器启动和监听一个服务，该服务监听来自本机的 ss-local 的请求。在收到来自 ss-local 转发过来的数据时，会先根据用户配置的加密方法和密码对数据进行对称解密，以获得加密后的数据的原内容。同时还会解 SOCKS5 协议，读出本次请求真正的目标服务地址(例如 Google 服务器地址)，再把解密后得到的原数据转发到真正的目标服务。

当真正的目标服务返回了数据时，ss-server 端会把返回的数据加密后转发给对应的 ss-local 端，ss-local 端收到数据再解密后，转发给本机的软件。这是一个对称相反的过程。

由于 ss-local 和 ss-server 端都需要用对称加密算法对数据进行加密和解密，因此这两端的加密方法和密码必须配置为一样。Shadowsocks 提供了一系列标准可靠的对称算法可供用户选择，例如 rc4、aes、des、chacha20 等等。Shadowsocks 对数据加密后再传输的目的是为了混淆原数据，让途中的防火墙无法得出传输的原数据。但其实用这些安全性高计算量大的对称加密算法去实现混淆有点“杀鸡用牛刀”。

SOCKS5 协议介绍

Shadowsocks 的数据传输是建立在 SOCKS5 协议之上的，SOCKS5 是 TCP/IP 层面的网络代理协议。
ss-server 端解密出来的数据就是采用 SOCKS5 协议封装的，通过 SOCKS5 协议 ss-server 端能读出本机软件想访问的服务的真正地址以及要传输的原数据，下面来详细介绍 SOCKS5 协议的通信细节。

建立连接

客户端向服务端连接连接，客户端发送的数据包如下：

VER	NMETHODS	METHODS
1	1	1

其中各个字段的含义如下：
- `VER` ：代表 SOCKS 的版本，SOCKS5 默认为 `0x05` ，其固定长度为1个字节；
- `NMETHODS` ：表示第三个字段METHODS的长度，它的长度也是1个字节；
- `METHODS` ：表示客户端支持的验证方式，可以有多种，他的长度是1-255个字节。

目前支持的验证方式共有：

- `0x00` ：NO AUTHENTICATION REQUIRED（不需要验证）
- `0x01` ：GSSAPI
- `0x02` ：USERNAME/PASSWORD（用户名密码）
- `0x03` : to X’7F’ IANA ASSIGNED
- `0x80` : to X’FE’ RESERVED FOR PRIVATE METHODS
- `0xFF` : NO ACCEPTABLE METHODS（都不支持，没法连接了）

响应连接

服务端收到客户端的验证信息之后，就要回应客户端，服务端需要客户端提供哪种验证方式的信息。服务端回应的包格式如下：

VER	METHOD
1	1

其中各个字段的含义如下：

- `VER` ：代表 SOCKS 的版本，SOCKS5 默认为 `0x05` ，其固定长度为1个字节；

- METHOD ：代表服务端需要客户端按此验证方式提供的验证信息，其值长度为1个字节，可为上面六种验证方式之一。

举例说明，比如服务端不需要验证的话，可以这么回应客户端：

VER	METHOD
0x05	0x00

和目标服务建立连接

客户端发起的连接由服务端验证通过后，客户端下一步应该告诉真正目标服务的地址给服务器，服务器得到地址后再去请求真正的目标服务。也就是说客户端需要把 Google 服务的地址 google.com:80 告诉服务端，服务端再去请求 google.com:80 。
目标服务地址的格式为 (IP或域名)+端口，客户端需要发送的包格式如下：

VER	CMD	RSV	ATYP	DST.ADDR	DST.PORT
1	1	0x00	1	Variable	2

各个字段的含义如下：

- VER ：代表 SOCKS 协议的版本，SOCKS 默认为0x05，其值长度为1个字节；
- CMD ：代表客户端请求的类型，值长度也是1个字节，有三种类型；
  - CONNECT ： 0x01 ；
  - BIND ： 0x02 ；
  - UDP ： ASSOCIATE 0x03 ；
- RSV ：保留字，值长度为1个字节；
- ATYP ：代表请求的远程服务器地址类型，值长度1个字节，有三种类型；
  - IPV4 ： address: 0x01 ；
  - DOMAINNAME： 0x03 ；
  - IPV6 ： address: 0x04 ；
- DST.ADDR ：代表远程服务器的地址，根据 ATYP 进行解析，值长度不定；

- `DST.PORT` ：代表远程服务器的端口，要访问哪个端口的意思，值长度2个字节。

服务端在得到来自客户端告诉的目标服务地址后，便和目标服务进行连接，不管连接成功与否，服务器都应该把连接的结果告诉客户端。在连接成功的情况下，服务端返回的包格式如下：

VER	REP	RSV	ATYP	BND.ADDR	BND.PORT
1	1	`0x00`	1	Variable	2

各个字段的含义如下：

- `VER` ：代表 SOCKS 协议的版本，SOCKS 默认为0x05，其值长度为1个字节；
- REP代表响应状态码，值长度也是1个字节，有以下几种类型
  - `0x00` succeeded
  - `0x01` general SOCKS server failure
  - `0x02` connection not allowed by ruleset
  - `0x03` Network unreachable
  - `0x04` Host unreachable
  - `0x05` Connection refused
  - `0x06` TTL expired
  - `0x07` Command not supported
  - `0x08` Address type not supported
  - `0x09` to `0xFF` unassigned
- `RSV` ：保留字，值长度为1个字节
- `ATYP` ：代表请求的远程服务器地址类型，值长度1个字节，有三种类型
  - IP V4 address： `0x01`
  - DOMAINNAME： `0x03`
  - IP V6 address： `0x04`
- `BND.ADDR` ：表示绑定地址，值长度不定。
- `BND.PORT` ： 表示绑定端口，值长度2个字节

数据转发

客户端在收到来自服务器成功的响应后，就会开始发送数据了，服务端在收到来自客户端的数据后，会转发到目标服务。

总结

SOCKS5 协议的目的其实就是为了把来自原本应该在本机直接请求目标服务的流程，放到了服务端去代理客户端访问。
其运行流程总结如下：

1. 本机和代理服务端协商和建立连接；
2. 本机告诉代理服务端目标服务的地址；
3. 代理服务端去连接目标服务，成功后告诉本机；
4. 本机开始发送原本应发送到目标服务的数据给代理服务端，由代理服务端完成数据转发。

以上内容来自 SOCKS5 协议规范 rfc1928 (http://www.ietf.org/rfc/rfc1928.txt)。

Lightsocks (https://github.com/gwuhaolin/lightsocks) 实现

要实现 Lightsocks 需要实现两部分：运行在本地的 lightsocks-local，和运行在墙外代理服务器上 lightsocks-server。
下面来分别教你如果使用 Golang 来实现它们，采用 Golang 语言的原因在于：性能好、跨平台、适合高并发、学习门槛低。对Golang感兴趣？请看Golang 中文学习资料汇总 (http://go.wuhaolin.cn/)

实现数据混淆

在 Shadowsocks 中是采用的标准的对称加密算法去实现数据混淆的，对称算法在加密和解密过程中需要大量计算。
为了简单起见，Lightsocks 将采用最简单高效的方法去实现数据混淆，具体原理如下。

这个数据混淆算法和对称加密很相似，两端都需要有同样的密钥。
这个密钥有如下要求：

- 由256个 byte 组成，也就是一个数组，在 Golang 中类型表示为 `[256]byte` ；
- 这个数组必须由 0～255 这256个数字组成，一个都不能差；
- 这个数组中第 `I` 个的值不能等于 `I` ；

例如以下为一个合法的密钥(上为索引，下为值)：

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
186	118	82	201	235	236	180	66	228	96	43	90	203	200	34	104	41

如果原数据为 `[5,0,1,2,3]` ，则采用以上密钥加密后变成 `[236,186,118,82,201]` 。
如果加密后的数据为 `[186,118,82,201,235]` ，则采用以上密钥解密得到的原数据为 `[0,1,2,3,4]`

聪明的你肯定看懂了其中的规律：把1～255 这256个数字确定一种一对一的映射关系，加密是从一个数字得到对应的一个数字，而解密则是反向的过程，而这个密钥的作用正是描述这个映射关系。这其实就是中学学的**反函数**。

为什么要这样设计数据混淆算法呢？在数据传输时，数据是以 byte 为最小单位流式传输的。一个 byte 的取值只可能是 0～255。该混淆算法可以直接对一个个 byte 进行加解密，而无需像标准的对称算法那样只能对一大块数据进行加密。
再加上本算法的加解密 N byte 数据的算法复杂度为 N（直接通过数组索引访问），非常适合流式加密。

以上加密算法的安全性怎么样呢？符合以上要求的密钥匙有多少种组合呢？我们来算算：
这其实就是初中学的排列组合中的排列问题，形象点其实就是，把 0～255 个不同编号的人安排到 0～255 个不同编号的坑去，并且不能有编号一样的情况，有多少种排法。
也就是 `A(255,255)=255*254*253*...*1=255!` ，但其中有一半为有重复的情况，
最终结果为 `255!/2` ，
其值大概为 `10^500` 这个数量级。

以上加密算法虽然破绽很多，但足以实现高效的数据混淆，骗过防火墙。

目前采用对称加密算法实现数据混淆的 Shadowsocks 已经能被一些防火墙通过机器学习算法通过特征分析识别出传输的原内容适合合法，而 Lightsocks 的这套混淆算法目前还不能被轻易的识别出来。

随机产生一个以上密钥匙的代码如下 (https://github.com/gwuhaolin/lightsocks/blob/master/core/password.go)：

```
package core
import (
        "math/rand"
        "time"
)
const PasswordLength = 256
type Password [PasswordLength]byte

func init() {
        // 更新随机种子，防止生成一样的随机密码
        rand.Seed(time.Now().Unix())
}

// 产生 256个byte随机组合的 密码
func RandPassword() *Password {
        // 随机生成一个由  0~255 组成的 byte 数组
        intArr := rand.Perm(PasswordLength)
        password := &Password{}
        for i, v := range intArr {
                password[i] = byte(v)
                if i == v {
                        // 确保不会出现如何一个byte位出现重复
                        return RandPassword()
                }
        }
        return password
}
```

对数据进行加密解密的代码如下 (https://github.com/gwuhaolin/lightsocks/blob/master/core/cipher.go)：

```
package core

type Cipher struct {
        // 编码用的密码
        encodePassword *Password
        // 解码用的密码
        decodePassword *Password
}

// 加密原数据
func (cipher *Cipher) encode(bs []byte) {
        for i, v := range bs {
                bs[i] = cipher.encodePassword[v]
        }
}

// 解码加密后的数据到原数据
func (cipher *Cipher) decode(bs []byte) {
        for i, v := range bs {
                bs[i] = cipher.decodePassword[v]
        }
}

// 新建一个编码解码器
func NewCipher(encodePassword *Password) *Cipher {
        decodePassword := &Password{}
        for i, v := range encodePassword {
                encodePassword[i] = v
                decodePassword[v] = byte(i)
        }
        return &Cipher{
                encodePassword: encodePassword,
                decodePassword: decodePassword,
        }
}
```

再使用以上的 Cipher 去封装一个加密传输的 SecureSocket，以方便直接加解密 TCP Socket 中的流式数据，代码如下 (https://github.com/gwuhaolin/lightsocks/blob/master/core/securesocket.go)：

```
package core

import (
        "errors"
        "fmt"
        "io"
        "net"
)

```

```
10  const (
11          BufSize = 1024
12  )
13
14  // 加密传输的 TCP Socket
15  type SecureSocket struct {
16          Cipher     *Cipher
17          ListenAddr *net.TCPAddr
18          RemoteAddr *net.TCPAddr
19  }
20
21  // 从输入流里读取加密过的数据，解密后把原数据放到bs里
22  func (secureSocket *SecureSocket) DecodeRead(conn *net.TCPConn, bs []byte) (n int, err e
23          n, err = conn.Read(bs)
24          if err != nil {
25                  return
26          }
27          secureSocket.Cipher.decode(bs[:n])
28          return
29  }
30
31  // 把放在bs里的数据加密后立即全部写入输出流
32  func (secureSocket *SecureSocket) EncodeWrite(conn *net.TCPConn, bs []byte) (int, error)
33          secureSocket.Cipher.encode(bs)
34          return conn.Write(bs)
35  }
36
37  // 从src中源源不断的读取原数据加密后写入到dst，直到src中没有数据可以再读取
38  func (secureSocket *SecureSocket) EncodeCopy(dst *net.TCPConn, src *net.TCPConn) error {
39          buf := make([]byte, BufSize)
40          for {
41                  readCount, errRead := src.Read(buf)
42                  if errRead != nil {
43                          if errRead != io.EOF {
44                                  return errRead
45                          } else {
46                                  return nil
47                          }
48                  }
49                  if readCount > 0 {
50                          writeCount, errWrite := secureSocket.EncodeWrite(dst, buf[0:reac
51                          if errWrite != nil {
52                                  return errWrite
53                          }
54                          if readCount != writeCount {
55                                  return io.ErrShortWrite
56                          }
57                  }
58          }
59  }
60
61  // 从src中源源不断的读取加密后的数据解密后写入到dst，直到src中没有数据可以再读取
62  func (secureSocket *SecureSocket) DecodeCopy(dst *net.TCPConn, src *net.TCPConn) error {
63          buf := make([]byte, BufSize)
```

```
64          for {
65                  readCount, errRead := secureSocket.DecodeRead(src, buf)
66                  if errRead != nil {
67                          if errRead != io.EOF {
68                                  return errRead
69                          } else {
70                                  return nil
71                          }
72                  }
73                  if readCount > 0 {
74                          writeCount, errWrite := dst.Write(buf[0:readCount])
75                          if errWrite != nil {
76                                  return errWrite
77                          }
78                          if readCount != writeCount {
79                                  return io.ErrShortWrite
80                          }
81                  }
82          }
83  }
84
85  // 和远程的socket建立连接，他们之间的数据传输会加密
86  func (secureSocket *SecureSocket) DialRemote() (*net.TCPConn, error) {
87          remoteConn, err := net.DialTCP("tcp", nil, secureSocket.RemoteAddr)
88          if err != nil {
89                  return nil, errors.New(fmt.Sprintf("连接到远程服务器 %s 失败:%s", secureSc
90          }
91          return remoteConn, nil
92  }
```

这个 SecureSocket 用于 local 端和 server 端之间进行 TCP 通信，并且只使用 SecureSocket 通信时中间传输的数据会被加密，防火墙无法读到原数据。

实现 local 端

运行在本机的 local 端的职责是把本机程序发送给它的数据经过加密后转发给墙外的代理服务器，总体工作流程如下：

1. 监听来自本机浏览器的代理请求；
2. 转发前加密数据；
3. 转发socket数据到墙外代理服务端；
4. 把服务端返回的数据转发给用户的浏览器。

实现以上功能的 local 端代码如下(https://github.com/gwuhaolin/lightsocks/blob/master/local/local.go)：

```
package local

import (
        "github.com/gwuhaolin/lightsocks/core"
        "log"
        "net"
)

type LsLocal struct {
        *core.SecureSocket
}

// 新建一个本地端
func New(password *core.Password, listenAddr, remoteAddr *net.TCPAddr) *LsLocal {
        return &LsLocal{
                SecureSocket: &core.SecureSocket{
                        Cipher:     core.NewCipher(password),
                        ListenAddr: listenAddr,
                        RemoteAddr: remoteAddr,
                },
        }
}

// 本地端启动监听，接收来自本机浏览器的连接
func (local *LsLocal) Listen(didListen func(listenAddr net.Addr)) error {
        listener, err := net.ListenTCP("tcp", local.ListenAddr)
        if err != nil {
                return err
        }

        defer listener.Close()

        if didListen != nil {
                didListen(listener.Addr())
        }

        for {
                userConn, err := listener.AcceptTCP()
                if err != nil {
                        log.Println(err)
                        continue
                }
                // userConn被关闭时直接清除所有数据 不管没有发送的数据
                userConn.SetLinger(0)
                go local.handleConn(userConn)
        }
        return nil
}
```

```
49
50  func (local *LsLocal) handleConn(userConn *net.TCPConn) {
51          defer userConn.Close()
52
53          proxyServer, err := local.DialRemote()
54          if err != nil {
55                  log.Println(err)
56                  return
57          }
58          defer proxyServer.Close()
59          // Conn被关闭时直接清除所有数据 不管没有发送的数据
60          proxyServer.SetLinger(0)
61
62          // 进行转发
63          // 从 proxyServer 读取数据发送到 localUser
64          go func() {
65                  err := local.DecodeCopy(userConn, proxyServer)
66                  if err != nil {
67                          // 在 copy 的过程中可能会存在网络超时等 error 被 return，只要有一个发
68                          userConn.Close()
69                          proxyServer.Close()
70                  }
71          }()
72          // 从 localUser 发送数据发送到 proxyServer，这里因为处在翻墙阶段出现网络错误的概率更大
73          local.EncodeCopy(proxyServer, userConn)
74  }
```

实现 server 端

运行在墙外代理服务器的 server 端职责如下：

1. 监听来自本地代理客户端的请求；
2. 解密本地代理客户端请求的数据，解析 SOCKS5 协议，连接用户浏览器真正想要连接的远程服务器；
3. 转发用户浏览器真正想要连接的远程服务器返回的数据的加密后的内容到本地代理客户端。

实现以上功能的代码如下 (https://github.com/gwuhaolin/lightsocks/blob/master/server/server.go)：

```
1  package server
2
3  import (
4          "encoding/binary"
5          "github.com/gwuhaolin/lightsocks/core"
6          "log"
```

```
 7          "net"
 8  )
 9
10  type LsServer struct {
11          *core.SecureSocket
12  }
13
14  // 新建一个服务端
15  func New(password *core.Password, listenAddr *net.TCPAddr) *LsServer {
16          return &LsServer{
17                  SecureSocket: &core.SecureSocket{
18                          Cipher:     core.NewCipher(password),
19                          ListenAddr: listenAddr,
20                  },
21          }
22  }
23
24  // 运行服务端并且监听来自本地代理客户端的请求
25  func (lsServer *LsServer) Listen(didListen func(listenAddr net.Addr)) error {
26          listener, err := net.ListenTCP("tcp", lsServer.ListenAddr)
27          if err != nil {
28                  return err
29          }
30
31          defer listener.Close()
32
33          if didListen != nil {
34                  didListen(listener.Addr())
35          }
36
37          for {
38                  localConn, err := listener.AcceptTCP()
39                  if err != nil {
40                          log.Println(err)
41                          continue
42                  }
43                  // localConn被关闭时直接清除所有数据 不管没有发送的数据
44                  localConn.SetLinger(0)
45                  go lsServer.handleConn(localConn)
46          }
47          return nil
48  }
49
50  // 解 SOCKS5 协议
51  // https://www.ietf.org/rfc/rfc1928.txt
52  func (lsServer *LsServer) handleConn(localConn *net.TCPConn) {
53          defer localConn.Close()
54          buf := make([]byte, 256)
55
56          /**
57             The localConn connects to the dstServer, and sends a ver
58             identifier/method selection message:
59                            +----+----------+----------+
60                            |VER | NMETHODS | METHODS  |
```

```
                        +----+----------+----------+
                        | 1  |    1     | 1 to 255 |
                        +----+----------+----------+
      The VER field is set to X'05' for this ver of the protocol.  The
      NMETHODS field contains the number of method identifier octets that
      appear in the METHODS field.
   */
   // 第一个字段VER代表Socks的版本，Socks5默认为0x05，其固定长度为1个字节
   _, err := lsServer.DecodeRead(localConn, buf)
   // 只支持版本5
   if err != nil || buf[0] != 0x05 {
          return
   }

   /**
      The dstServer selects from one of the methods given in METHODS, and
      sends a METHOD selection message:

                        +----+--------+
                        |VER | METHOD |
                        +----+--------+
                        | 1  |   1    |
                        +----+--------+
   */
   // 不需要验证，直接验证通过
   lsServer.EncodeWrite(localConn, []byte{0x05, 0x00})

   /**
                        +----+-----+-------+------+----------+----------+
                        |VER | CMD |  RSV  | ATYP | DST.ADDR | DST.PORT |
                        +----+-----+-------+------+----------+----------+
                        | 1  |  1  | X'00' |  1   | Variable |    2     |
                        +----+-----+-------+------+----------+----------+
   */

   // 获取真正的远程服务的地址
   n, err := lsServer.DecodeRead(localConn, buf)
   // n 最短的长度为7 情况为 ATYP=3 DST.ADDR占用1字节 值为0x0
   if err != nil || n < 7 {
          return
   }

   // CMD代表客户端请求的类型，值长度也是1个字节，有三种类型
   // CONNECT X'01'
   if buf[1] != 0x01 {
          // 目前只支持 CONNECT
          return
   }

   var dIP []byte
   // aType 代表请求的远程服务器地址类型，值长度1个字节，有三种类型
   switch buf[3] {
   case 0x01:
          //      IP V4 address: X'01'
```

```
115                 dIP = buf[4 : 4+net.IPv4len]
116         case 0x03:
117                 //      DOMAINNAME: X'03'
118                 ipAddr, err := net.ResolveIPAddr("ip", string(buf[5:n-2]))
119                 if err != nil {
120                         return
121                 }
122                 dIP = ipAddr.IP
123         case 0x04:
124                 //      IP V6 address: X'04'
125                 dIP = buf[4 : 4+net.IPv6len]
126         default:
127                 return
128         }
129         dPort := buf[n-2:]
130         dstAddr := &net.TCPAddr{
131                 IP:   dIP,
132                 Port: int(binary.BigEndian.Uint16(dPort)),
133         }
134
135         // 连接真正的远程服务
136         dstServer, err := net.DialTCP("tcp", nil, dstAddr)
137         if err != nil {
138                 return
139         } else {
140                 defer dstServer.Close()
141                 // Conn被关闭时直接清除所有数据 不管没有发送的数据
142                 dstServer.SetLinger(0)
143
144                 // 响应客户端连接成功
145                 /**
146                           +----+-----+-------+------+----------+----------+
147                           |VER | REP |  RSV  | ATYP | BND.ADDR | BND.PORT |
148                           +----+-----+-------+------+----------+----------+
149                           | 1  |  1  | X'00' |  1   | Variable |    2     |
150                           +----+-----+-------+------+----------+----------+
151                 */
152                 // 响应客户端连接成功
153                 lsServer.EncodeWrite(localConn, []byte{0x05, 0x00, 0x00, 0x01, 0x00, 0x
154         }
155
156         // 进行转发
157         // 从 localUser 读取数据发送到 dstServer
158         go func() {
159                 err := lsServer.DecodeCopy(dstServer, localConn)
160                 if err != nil {
161                         // 在 copy 的过程中可能会存在网络超时等 error 被 return，只要有一个
162                         localConn.Close()
163                         dstServer.Close()
164                 }
165         }()
166         // 从 dstServer 读取数据发送到 localUser，这里因为处在翻墙阶段出现网络错误的概率更大
167         lsServer.EncodeCopy(localConn, dstServer)
168 }
```

以上就是实现一个轻量级 Shadowsocks 的核心代码。其它一些零碎的代码，例如启动入口、配置读写等，可以去 lightsocks (https://github.com/gwuhaolin/lightsocks) 项目中阅读完整代码。

阅读原文
(http://wuhaolin.cn/2017/11/03/%E4%BD%A0%E4%B9%9F%E8%83%BD%E5%86%99%E4%B8%A

在 Github 上讨论 (https://github.com/gwuhaolin/blog/issues/12)

← PREVIOUS POST (/2017/12/27/PARCEL VS WEBPACK/)

NEXT POST → (/2017/11/03/调试利器-SSH隧道/)

FEATURED TAGS (/tags/)

网络 (/tags/#网络)

(https://www.zhihu.com/people/wu-hao-lin-67-15)

(https://github.com/gwuhaolin)