SHadowsocks设计分析文档

2017/12/27

目录

[项目背景和原理 1](#_Toc499830161)

[项目方法 2](#_Toc499830162)

[OOA模型 2](#_Toc499830163)

[1. 需求模型 2](#_Toc499830164)

[系统边界 2](#_Toc499830165)

[2. 基本模型 3](#_Toc499830166)

[3. 辅助模型 4](#_Toc499830167)

[OOD模型 4](#_Toc499830168)

[1. 问题域部分 4](#_Toc499830169)

[2. 数据接口部分 4](#_Toc499830170)

[3. 控制驱动部分 4](#_Toc499830171)

[4. 人机交互部分 4](#_Toc499830172)

[总结 4](#_Toc499830173)

SHadowsocks设计分析文档

2017/12/27

# 项目背景和原理

Shadowsocks是一款比较知名和应用比较广泛的翻墙软件。Shadowsocks的代码质量很高，并且有python编程语言编写的版本，符合面向对象编程的要求。

Shadowsocks的翻墙原理如下列示意图所示：



图1. Shadowsocks原理示意图

Shadowsocks基于Sock5协议完成了一个墙内墙外的通信过程：

1. SS Local把PC的数据包进行加密
2. SS Local转发加密后的数据包到墙外的服务器SS Server
3. SS Server把解密后的数据包转发到目标地址
4. SS Server把返回的数据包进行加密
5. SS Server把加密后的数据包转发到墙内的SS Local
6. SS Local把解密后的数据包返回到PC

上述就是Shadowsocks的基本原理。该软件实现过程涉及到网络通信、命令行交互、加密解密、IO复用等方面，是一个有一定复杂度的软件。

# 项目方法

本次项目主要针对shadoesocks的2.8.2 python版本进行建模分析。在这个项目里面我们主要采用的建模方法是面向对象建模。原因主要有以下几点：

1. 本次分析的软件是用python语言进行编写的。而python是一个面向对象的编程语言，与面向对象建模及其吻合
2. 面向对象建模是目前比较流行的一种建模方法。掌握好这一方法有利于提高以后的编程能力。

因此在采用面向对象建模方法以后，主要流程包括OOA和OOD分析两个阶段。这两个阶段建立的模型分别为OOA模型和OOD模型。

# OOA模型

OOA阶段主要发生在面向对象建模的分析阶段。OOA模型框架的示意图如下：



图2. OOA模型框架

本小节主要按照需求模型、基本模型和辅助模型的顺序一一阐述。

## 需求模型

### 系统边界

在面向对象建模的方法中，第一步需要进行的就是需求分析建模。而需求模型主要是由用况图(use case diagram)和对应的文档构成。在构建需求模型之前，我们需要确定好系统边界。

在系统边界的表示上，我们首先把整个软件系统看成一个单独的系统，把软件之外的参与者先确定出来。

首先对整个系统进行边界确定。其示意图如图3所示。首先整个系统的参与者有以下几个：

1. 客户端：即墙内的PC，利用本系统与墙外的某些服务器通信
2. 服务器端：即墙外的服务，利用本系统与墙内的PC通信
3. DNS服务器：主要是客户端再必要的时候，通过DNS服务器查询服务器的ip地址。



图3. 整体系统的边界

根据以上信息建立需求模型如图4所示下：



图4 需求模型用况图

其中每个用况的如下：

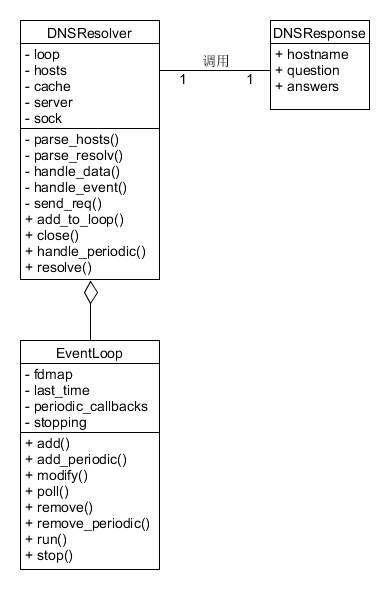
|  |  |
| --- | --- |
| 用况名称 | SS Local |
| 用况编号 | 1 |
| 行为陈述 | SS Local 获取客户端配置文件 启动客户端进程 while(1)  if DNS解析请求 then  call DNS解析  if tcp转发 then  call tcp转发  if udp转发 then  call udp转发  if 收到退出请求 then  关闭tcp、udp连接  if 收到终止请求 then  关闭客户端进程  break |

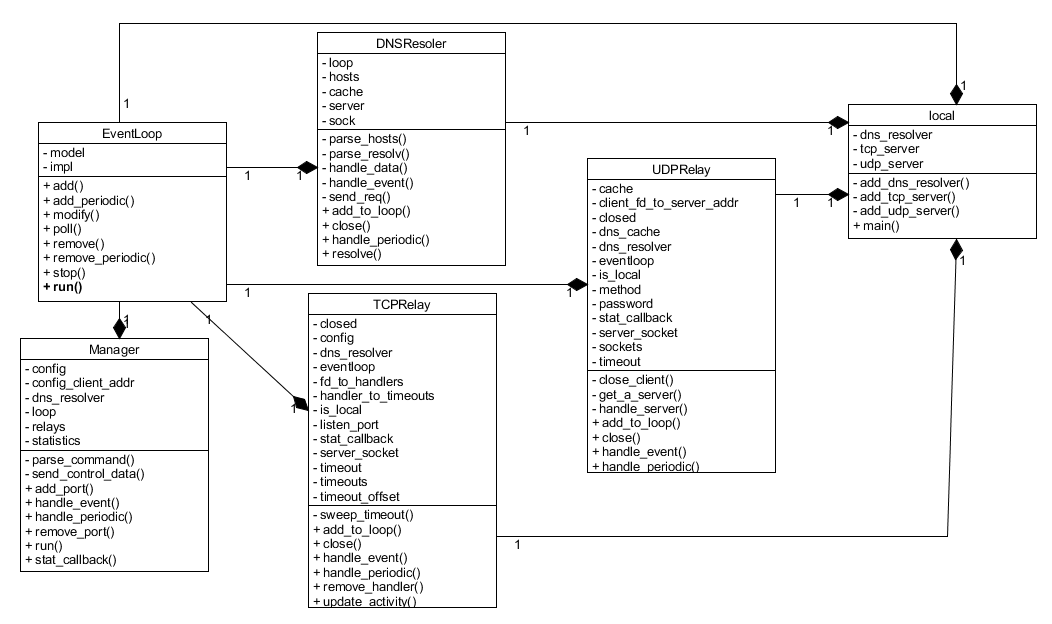
|  |  |
| --- | --- |
| 用况名称 | SS Local |
| 用况编号 | 1 |
| 行为陈述 | SS Local 获取客户端配置文件 启动客户端进程 while(1)  if DNS解析请求 then  call DNS解析  if tcp转发 then  call tcp转发  if udp转发 then  call udp转发  if 收到退出请求 then  关闭tcp、udp连接  if 收到终止请求 then  关闭客户端进程  break |

|  |  |
| --- | --- |
| 用况名称 | DNS解析 |
| 用况编号 | 1 |
| 行为陈述 | DNS 解析 if 域名为空 then  退出并提示域名为空 elif 域名已经是ip地址 then  直接返回结果 elif 域名的解析结果已经存在操作系统了 then  直接返回结果 elif 域名的解析结果已经缓存了 then  直接返回结果 else  if 域名的拼写无效 then  直接退出并提示域名的拼写错误  end if  向DNS服务器发送DNS解析请求  等待DNS解析结果返回  返回DNS解析结果  关闭DNS解析请求 end if |

## 基本模型

基本模型是类图





Local和EventLoop的类图

|  |  |  |
| --- | --- | --- |
| 类的总体说明 |  |  |
|  | 类名：<名字> | 给出类的名称 |
|  | 解释：[<文字描述>] | 对这个类做必要的文字解释 |
|  | 一般类：[<类名>]{, 类名} | 列出被这个类继承的每个一般类 |
|  | 主动性：Yes | No | 指出这个类是不是主动类 |
|  | 持久性：Yes | No | 指出类的对象是不是持久对象 |
|  | 辅助模型：{<访问路径和名字>} | 若有针对本类的状态机图、定时图或组合  结构图等辅助模型，则指出 |
|  | 其他 |  |
| 属性说明 |  |  |
|  | { | 针对每一个属性说明如下内容 |
|  | 名称与数据类型：<属性名>：<类型> | 给出属性的名称及数据类型 |
|  | 属性解释：[<文字描述>] | 做必要的文字解释 |
|  | 多态性：[\*|x] | 如果该属性是多态的则做标记 |
|  | 关联、聚合或组合：[关联|聚合|组合][<文字描述>] | 指出用该属性实现的关系，并作必要的说明 |
|  | 其他： |  |
|  | } |  |
| 操作说明 |  |  |
|  | { | 针对每一个操作说明如下内容 |
|  | 特征标记：<操作名>([<参数>:<类型>]{,<参数>:<类型>}[:<返回类型>]) | 给出操作名称、输入参数及类型，必要  时指明返回类型 |
|  | 操作解释：[<文字描述>] | 对操作功能做必要的解释 |
|  | 主动性：主动[进程|线程]|被动 | 指出该操作是不是主动的，若是，可指  出是进程还是线程 |
|  | 多态性：[\*|x] | 如果该操作是多态的则做标记 |
|  | 消息发送：[<类名><操作名>]{,<类名>·<操作名>} | 对由该操作发送的每一种消息，指出接收者 |
|  | 操作流程：[<访问路径和名字>] | 如有描述本操作的活动图或流程图，则  指出 |
|  | 其他： |  |
|  | } |  |
| 对象实例说明 | { | 对每种需要该类对象的处理机说明如下  内容 |
|  | 处理机：<结点名>{,<结点名>} | 指出这种处理机的每一个实际的节点 |
|  | 内存对象：{<名称>[(n元数组)][<文字描述>]} | 指出用该类创建所有内存对象 |
|  | 外存对象：{<名称>}[<文字描述>] | 指出为这个类保存的外存对象 |
|  | } |  |

|  |  |  |
| --- | --- | --- |
| 类的总体说明 |  |  |
|  | 类名 | DNSResolver |
|  | 解释 | 这个类的主要功能就是处理DNS解析请求。 |
|  | 一般类 | None |
|  | 主动性 | No |
|  | 持久性 | No |
|  | 辅助模型 | 若有针对本类的状态机图、定时图或组合  结构图等辅助模型，则指出 |
|  | 其他 |  |
| 属性说明 |  |  |
|  | { |  |
|  | 名称与数据类型 | loop : EventLoop |
|  | 属性解释 | 这个是一个Eventloop的实例，主要是为了利用IO复用机制来获得DNS解析请求返回的  结果。 |
|  | 多态性 | x |
|  | 组合 | DNSResolver必须得依靠Eventloop才能利用IO复用机制。因此两者之前有一个紧密、牢  固的关系，适用于组合范畴。 |
|  | 其它 |  |
|  | } |  |
|  | { |  |
|  | 名称与数据类型 | hosts : dict |
|  | 属性解释 | hosts保存了操作系统中从域名到ip地址的映射。如果操作系统也没有缓存到从域名到ip  地址的映射，则再调用其它方法获取映射。 |
|  | 多态性 | No |
|  | 聚合 | DNSResolver可以从hosts中来获取域名到ip地址的映射，也可以从其它方法获取映射。  因此hosts和DNSResolver之间是一种比较松散的聚合关系 |
|  | 其它 |  |
|  | } |  |
| 操作说明 |  |  |
|  | { |  |
|  | 特征标记 | parse\_hosts() |
|  | 操作解释 | 该操作主要是解析本地操作系统的hosts文件配置，并将其中的映射信息提取到hosts属性  中。 |
|  | 主动性方法 | 被动 |
|  | 多态性 | x |
|  | 消息发送 | None |
|  | 操作流程 | 如有描述本操作的活动图或流程图，则  指出 |
|  | 其他 |  |
|  | } |  |
| 对象实例说明 | { | 对每种需要该类对象的处理机说明如下  内容 |
|  | 处理机 | 指出这种处理机的每一个实际的节点 |
|  | 内存对象 | 指出用该类创建所有内存对象 |
|  | 外存对象 | 指出为这个类保存的外存对象 |
|  | } |  |

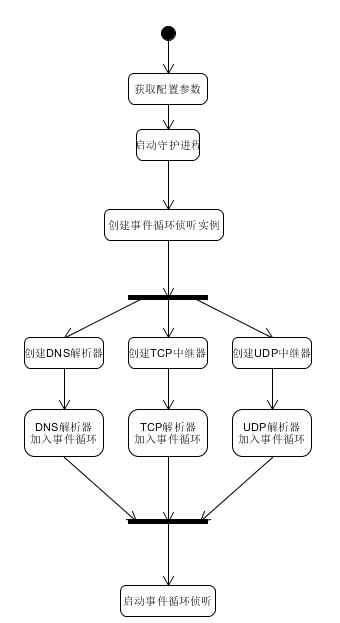
|  |  |  |
| --- | --- | --- |
| 类的总体说明 |  |  |
|  | 类名 | Local |
|  | 解释 | 这个类作为sslocal的入口类，主要是创建DNS解析器、TCP中继器、UDP中继器和事件  侦听接口，并将这三个组件注册到事件循环中，以触发实现预定的行为。 |
|  | 一般类 | None |
|  | 主动性 | Yes |
|  | 持久性 | Yes |
|  | 辅助模型 | None |
|  | 其他 | None |
| 属性说明 |  |  |
|  | { |  |
|  | 名称与数据类型 | dns\_resolver: DNSResolver |
|  | 属性解释 | 这是DNS解析的一个实例，在这里主要是将这个实例注册到事件循环中。 |
|  | 多态性 | No |
|  | 组合 | DNSResolver必须得依靠Eventloop才能利用IO复用机制。因此两者之前有一个紧密、牢  固的关系，适用于组合范畴。 |
|  | 其它 | None |
|  | } |  |
|  | { |  |
|  | 名称与数据类型 | tcp\_server: TCPRelay |
|  | 属性解释 | 这是TCP中继器的一个实例，在这里主要是将这个实例注册到事件循环中。 |
|  | 多态性 | No |
|  | 组合 | TCPRelay必须得依靠Eventloop才能利用IO复用机制。因此两者之前有一个紧密、牢  固的关系，适用于组合范畴。 |
|  | 其它 | None |
|  | } |  |
|  | { |  |
|  | 名称与数据类型 | udp\_server: UDPRelay |
|  | 属性解释 | 这是UDP中继器的一个实例，在这里主要是将这个实例注册到事件循环中。 |
|  | 多态性 | No |
|  | 组合 | UCPRelay必须得依靠Eventloop才能利用IO复用机制。因此两者之前有一个紧密、牢  固的关系，适用于组合范畴。 |
|  | 其它 | None |
|  | } |  |
| 操作说明 |  |  |
|  | { |  |
|  | 特征标记 | add\_dns\_resolver() |
|  | 操作解释 | 该操作主要是将dns\_resolver这个实例注册到事件循环eventloop中去。 |
|  | 主动性方法 | 被动 |
|  | 多态性 | x |
|  | 消息发送 | No |
|  | 操作流程 | None |
|  | 其他 |  |
|  | } |  |
|  | { |  |
|  | 特征标记 | add\_tcp\_server() |
|  | 操作解释 | 该操作主要是将tcp\_server这个实例注册到事件循环eventloop中去。 |
|  | 主动性方法 | 被动 |
|  | 多态性 | x |
|  | 消息发送 | No |
|  | 操作流程 | None |
|  | 其他 |  |
|  | } |  |
|  | { |  |
|  | 特征标记 | add\_udp\_server() |
|  | 操作解释 | 该操作主要是将udp\_server这个实例注册到事件循环eventloop中去。 |
|  | 主动性方法 | 被动 |
|  | 多态性 | x |
|  | 消息发送 | No |
|  | 操作流程 | None |
|  | 其他 |  |
|  | } |  |
|  | { |  |
|  | 特征标记 | main() |
|  | 操作解释 | sslocal的入口函数，这个函数主要是创建DNS解析器，TCP中继器、UDP中继器和事件  循环，并将三个组件注册到事件循环中去。最后启动事件循环。 |
|  | 主动性方法 | Yes |
|  | 多态性 | x |
|  | 消息发送 | No |
|  | 操作流程 | 活动图 |
|  | 其他 |  |
|  | } |  |
| 对象实例说明 | { |  |
|  | 处理机 |  |
|  | 内存对象 |  |
|  | 外存对象 |  |
|  | } |  |

才

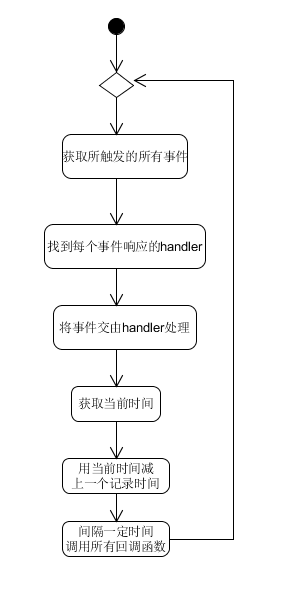
|  |  |  |
| --- | --- | --- |
|  | 类名 | Eventloop |
|  | 解释 | 这个一个侦听事件的类，用于将socket所侦听的事件注册进去，不断循环侦听，触发事件  将调用相应的handler处理。 |
|  | 一般类 | None |
|  | 主动性 | No |
|  | 持久性 | Yes |
|  | 辅助模型 | None |
|  | 其他 | None |
| 属性说明 |  |  |
|  | { |  |
|  | 名称与数据类型 | model: string |
|  | 属性解释 | 指明Eventloop目前是哪种IO复用接口。有’epoll’、’kqueue’和’select’三种模式。 |
|  | 多态性 | No |
|  | 组合 | None |
|  | 其它 | None |
|  | } |  |
|  | { |  |
|  | 名称与数据类型 | impl: Epoll/KqueueLoop/SelectLoop |
|  | 属性解释 | IO复用模式实例，可以为Epoll、KqueueLoop或者SelectLoop。 |
|  | 多态性 | No |
|  | 组合 | 作为Eventloop必不可少的成员。 |
|  | 其它 | None |
|  | } | None |
| 操作说明 |  |  |
|  | { |  |
|  | 特征标记 | add(socket, mode, handler) |
|  | 操作解释 | 将socket与对应的的处理handler加入字典中，并在相关IO复用接口中为socket注册  所指定的侦听事件。 |
|  | 主动性方法 | 被动 |
|  | 多态性 | X |
|  | 消息发送 | No |
|  | 操作流程 | None |
|  | 其他 | None |
|  | } |  |
|  | { |  |
|  | 特征标记 | add\_periodic(callback) |
|  | 操作解释 | 增加周期性回调函数，可添加的函数包括TCPRelay、UDPRelay或DNSResolver的  handle\_periodic函数处理超时或者清除缓存。 |
|  | 主动性方法 | 被动 |
|  | 多态性 | x |
|  | 消息发送 | No |
|  | 操作流程 | None |
|  | 其他 | None |
|  | } |  |
|  | { |  |
|  | 特征标记 | modify(socket, mode) |
|  | 操作解释 | 修改指定socket所侦听的事件为mode事件。 |
|  | 主动性方法 | 被动 |
|  | 多态性 | x |
|  | 消息发送 | No |
|  | 操作流程 | None |
|  | 其他 | None |
|  | } |  |
|  | { |  |
|  | 特征标记 | poll(timeout) |
|  | 操作解释 | 该操作主要是调用IO复用接口来等待事件触发，并返回触发的事件。 |
|  | 主动性方法 | 被动 |
|  | 多态性 | x |
|  | 消息发送 | No |
|  | 操作流程 | None |
|  | 其他 | None |
|  | } |  |
|  | { |  |
|  | 特征标记 | remove(socket) |
|  | 操作解释 | 将socket从字典中移除，并移除注册的侦听事件。 |
|  | 主动性方法 | 被动 |
|  | 多态性 | x |
|  | 消息发送 | No |
|  | 操作流程 | None |
|  | 其他 | None |
|  | } |  |
|  | { |  |
|  | 特征标记 | remove\_periodic(callback) |
|  | 操作解释 | 移除周期性回调函数，移除的函数可能为TCPRelay、UDPRelay或者DNSResolver的  handle\_periodic函数处理超时或者清除缓存。 |
|  | 主动性方法 | 被动 |
|  | 多态性 | x |
|  | 消息发送 | No |
|  | 操作流程 | None |
|  | 其他 | None |
|  | } |  |
|  | { |  |
|  | 特征标记 | stop() |
|  | 操作解释 | 暂停IO接口复用。 |
|  | 主动性方法 | 被动 |
|  | 多态性 | x |
|  | 消息发送 | No |
|  | 操作流程 | None |
|  | 其他 | None |
|  | } |  |
|  | { |  |
|  | 特征标记 | run() |
|  | 操作解释 | 等待注册事件发生，然后通过事件对应的文件描述符找到handler，并将事件交给handler  处理。同时每隔一定时间调用handle\_periodic函数处理超时或者清除缓存。 |
|  | 主动性方法 | 被动 |
|  | 多态性 | x |
|  | 消息发送 | No |
|  | 操作流程 | 活动图 |
|  | 其他 | None |
|  | } |  |
| 对象实例说明 | { |  |
|  | 处理机 |  |
|  | 内存对象 |  |
|  | 外存对象 |  |
|  | } |  |

## 辅助模型

本项目主要用到的辅助模型包括



Local.main()活动图



Eventloop.run()活动图

# OOD模型

OOD阶段是建立在OOA阶段的基础上，对OOA模型作必要的修改和调整或补充某些细节。其模型框架如下图所示：



本小节主要是按照问题域部分、数据接口部分、控制驱动部分和人机交互部分去阐述。

## 问题域部分

问题域部分主要是对

## 数据接口部分

数据接口部分包括

## 控制驱动部分

控制驱动部分包括

## 人机交互部分

人机交互部分包括

# 总结

总结