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

根据以上信息建立的总体需求模型如图4所示下：

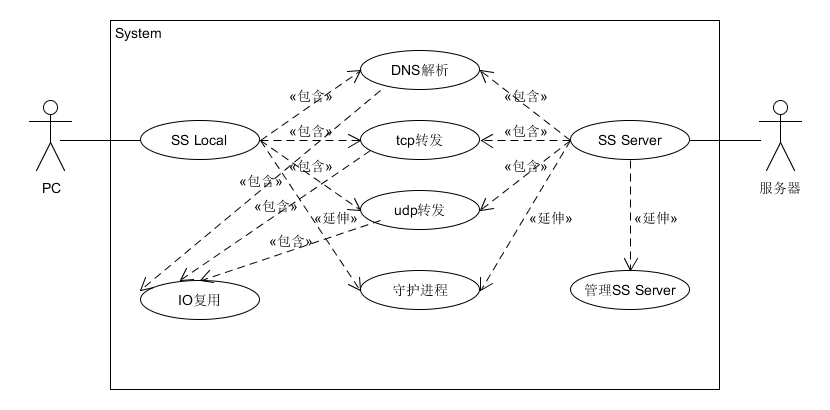


图4 总体需求模型用况图

其中每个用况的规约如下：

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

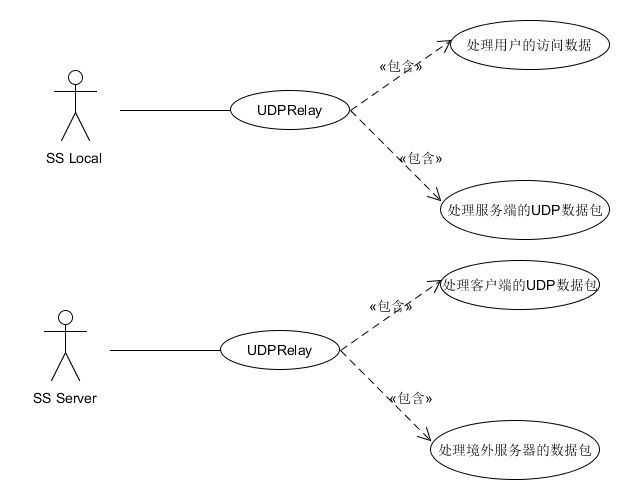
|  |  |
| --- | --- |
| 用况名称 | SS Server |
| 用况编号 | 1 |
| 行为陈述 | SS Server  读取配置 启动服务器端进程 while(1)  if DNS解析请求 then  call DNS解析  end if  if tcp转发 then  call tcp转发  endif  if udp转发 then  call udp转发  endif  if 收到退出请求 then  关闭tcp、udp连接  end if  if 收到终止请求 then  关闭客户端进程  break  end if |

|  |  |
| --- | --- |
| 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 |

|  |  |
| --- | --- |
| 用况名称 | 守护进程 |
| 用况编号 | 1 |
| 行为陈述 | 守护进程 获取配置文件 while(1)  if 命令是开启守护进程 then  开启守护进程  if 命令是终止守护进程 then  终止守护进程  if 命令是重启守护进程 then  终止守护进程  开启守护进程  break |

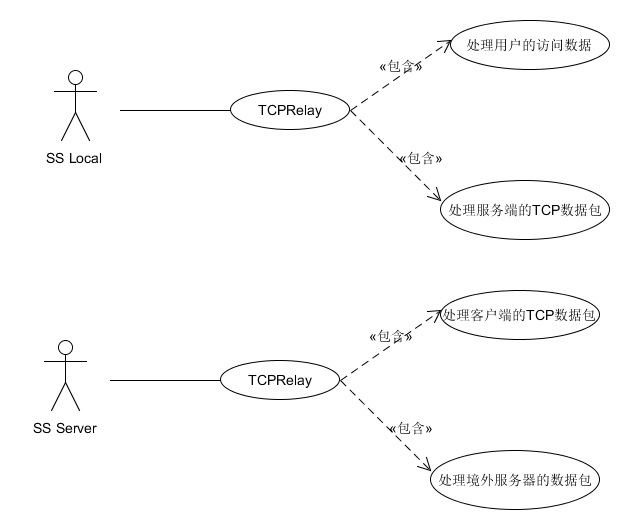
|  |  |
| --- | --- |
| 用况名称 | IO复用 |
| 用况编号 | 1 |
| 行为陈述 | IO复用 启动IO复用 while(1)  if 有新的监听对象 then  添加新的监听对象  end if  if 监听到新的活动 then  提示活动的目标对象处理该事件  end if |

对于tcp和udp转发以及管理服务器模块, 我们将其视为三个子模块来进行分析，其用况图如下



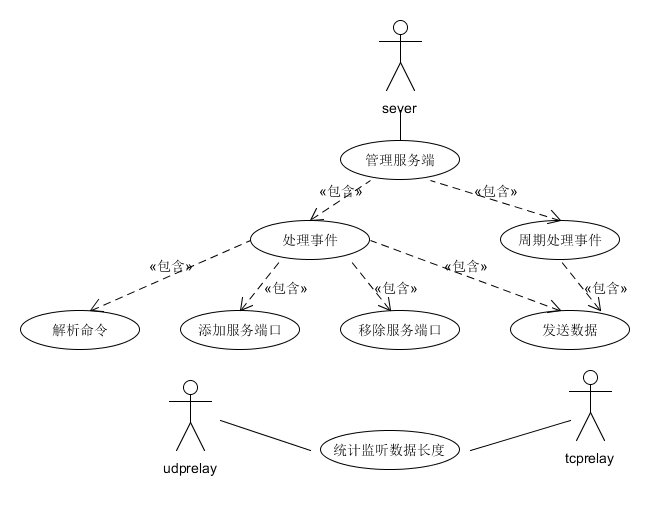
udp转发模块的用况分析图

|  |  |
| --- | --- |
| 用况名称 | UDPRelay |
| 用况编号 | 1 |
| 行为陈述 | UDPRelay  While（1）  if 运行在服务器端 then  if 收到来自客户端的UDP数据包 then  解密包  将数据发送到目标服务器  end if  if 收到来自境外服务器的数据包 then  将数据加密  将加密后的数据包使用UDP发送到目标客户端  end if  end if  if 运行在客户端 then  if 收到用户的访问数据 then  加密数据  将加密的数据使用UDP发送到服务端  end if  if 收到来自服务端的UDP数据包 then  加密数据包  将加密后的数据返回给用户  end if  end if |



tcp转发模块的用况分析图

|  |  |
| --- | --- |
| 用况名称 | TCPRelay |
| 用况编号 | 2 |
| 行为陈述 | TCPRelay  While（1）  if 运行在服务器端 then  if 收到来自客户端的TCP数据包 then  解密包  将数据发送到目标服务器  end if  if 收到来自境外服务器的数据包 then  将数据加密  将加密后的数据包使用TCP发送到目标客户端  end if  end if  if 运行在客户端 then  if 收到用户的访问数据 then  加密数据  将加密的数据使用TCP发送到服务端  end if  if 收到来自服务端的TCP数据包 then  加密数据包  将加密后的数据返回给用户  end if  end if |



管理服务器模块用况图

|  |  |
| --- | --- |
| 用况名称 | 添加服务端口 |
| 用况编号 | 1 |
| 行为陈述 | 添加服务端口 从配置中获取端口号，根据端口号获得服务端；  if 服务端已存在 then  return  end if;  实例化tcp和udp管理，添加到事件监听的IO复用； |
| 参与者 |  |

|  |  |
| --- | --- |
| 用况名称 | 移除服务端口 |
| 用况编号 | 2 |
| 行为陈述 | 移除服务端口 从配置中获取端口号，根据端口号获得服务端；  if 服务端在管理器中 then  停止监听该服务端tcp和udp数据包；  删除该端口对应的tcp和udp的管理；  else  报错；  end if； |
| 参与者 |  |

|  |  |
| --- | --- |
| 用况名称 | 解析命令 |
| 用况编号 | 3 |
| 行为陈述 | 解析命令  从数据中分出命令部分和配置部分 |
| 参与者 |  |

|  |  |
| --- | --- |
| 用况名称 | 统计监听数据的长度 |
| 用况编号 | 4 |
| 行为陈述 | 统计监听数据的长度  增加对应端口的获得数据长度 |
| 参与者 | udprelay, tcprelay |

|  |  |
| --- | --- |
| 用况名称 | 发送控制数据 |
| 用况编号 | 5 |
| 行为陈述 | 发送控制数据  if 存在客户端的地址 then  发送数据  end if； |
| 参与者 |  |

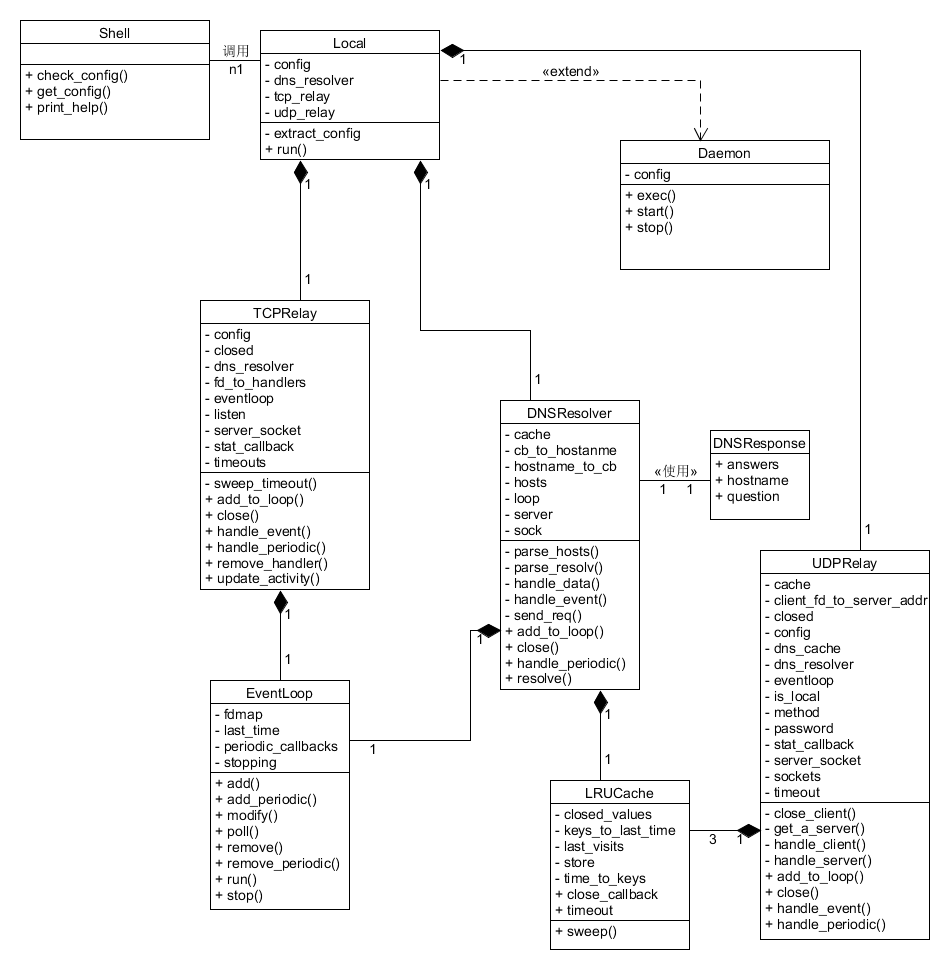
|  |  |
| --- | --- |
| 用况名称 | 周期处理事件 |
| 用况编号 | 6 |
| 行为陈述 | 周期性地向客户端发送接收到数据的总长度  for 每个要发送的数据 then  临时保存发送数据  统计数量  if 达到发送上限 then  call 发送控制数据  清除临时保存  end if；  end for；  发送剩余的数据 |

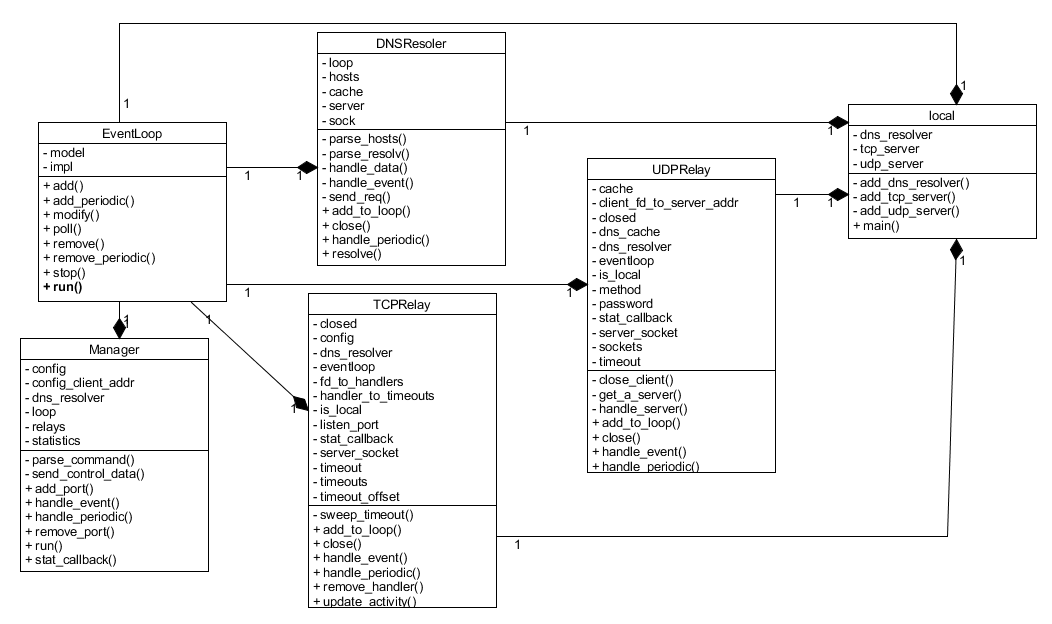
|  |  |
| --- | --- |
| 用况名称 | 管理服务端 |
| 用况编号 | 7 |
| 行为陈述 | 管理服务端  运行管理服务端程序 |

|  |  |
| --- | --- |
| 用况名称 | 处理事件 |
| 用况编号 | 8 |
| 行为陈述 | 处理事件  if 有事件触发且sock存在 then  获取数据和客户端地址  call 解析命令  if 获取到数据 then  if 获取到配置信息 then  更新配置信息  end if；  if 如果服务端口不在配置信息中 then  报错  else  if 命令为‘add’ then  call 添加服务端口  call 发送控制数据  elif 命令为‘remove’ then  call 移除服务端口  call 发送控制数据  elif 命令为‘ping’ then  call 发送控制数据  else  报错  end if；  end if；  end if；  end for； |

## 基本模型

基本模型是类图。根据总体需求用况图和三个子模块的用况图，我们也分成四个部分来展示类图





Local和EventLoop的类图

|  |  |  |
| --- | --- | --- |
| 类的总体说明 |  |  |
|  | 类名 | <中文>域名解析：<英文>DNSResolver |
|  | 解释 | 这个类的主要功能就是处理DNS解析请求来返回域名对应的ip地址。因为有可能SS服务器的地址是一个域名地址，所以需要相应的DNS解析请求模块来获取对应的ip地址。 |
|  | 一般类 | No |
|  | 主动性 | No |
|  | 持久性 | No |
|  | 辅助模型 |  |
|  | 其他 |  |
| 属性说明 |  |  |
|  | { |  |
|  | 名称与数据类型 | cache : LRUCache |
|  | 属性解释 | cache 缓存了最近解析过的域名，可以减少访问域名服务器的次数并提高查找域名映射的速度。 |
|  | 多态性 | None |
|  | 组合 | cache 是DNSResolver的一部分，必须得依靠cache才能利用缓存机制提高查找域名映射的速度。 |
|  | 其它 |  |
|  | } |  |
|  | { |  |
|  | 名称与数据类型 | cb\_to\_hostname : dict |
|  | 属性解释 | 该属性保留了每个回调操作应该对应的域名。 |
|  | 多态性 | None |
|  | 组合 | None |
|  | 其它 |  |
|  | } |  |
|  | { |  |
|  | 名称与数据类型 | hostname\_to\_cb : dict |
|  | 属性解释 | 该属性保留了每个域名对应的回调操作。 |
|  | 多态性 |  |
|  | 组合 |  |
|  | 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 | None |
|  | 关联、聚合或组合 | None |
|  | 其它 |  |
|  | } |  |
|  | { |  |
|  | 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 | None |
|  | 关联、聚合或组合 | 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 | server : list |
|  | 属性解释 | server里面保存了操作系统中DNS服务器的ip地址。如果操作系统没有保存有DNS服务器的ip地址，则默认为谷歌DNS服务器的ip地址。 |
|  | 多态性 | None |
|  | 关联、聚合或组合 |  |
|  | 其它 |  |
|  | } |  |
|  | { |  |
|  | 名称与数据类型 | sock ： Socket |
|  | 属性解释 | sock是Socket类的一个实例，负责和域名服务器通信来获取域名解析结果。 |
|  | 多态性 | None |
|  | 关联、聚合或组合 | None |
|  | 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 | None |
|  | 消息发送 | None |
|  | 操作流程 |  |
|  | 其他 |  |
|  | } |  |
|  | { |  |
|  | 特征标记 | parse\_resolv() |
|  | 操作解释 | 该操作主要是解析操作系统中的域名解析配置文件，并从中提取出域名服务器的ip地址。 |
|  | 主动性方法 | 被动 |
|  | 多态性 | None |
|  | 消息发送 |  |
|  | 操作流程 |  |
|  | 其他 |  |
|  | } |  |
|  | { |  |
|  | 特征标记 | handle\_data(data : bytes) |
|  | 操作解释 | 该操作主要是对域名解析请求返回的包进行解析，并从中提取出ip地址。 |
|  | 主动性方法 | 被动 |
|  | 多态性 | None |
|  | 消息发送 |  |
|  | 操作流程 |  |
|  | 其他 |  |
|  | } |  |
|  | { |  |
|  | 特征标记 | handle\_event(sock : Socket , fd : int , event : int) |
|  | 操作解释 | 该操作主要是响应IO复用的唤醒，并把域名解析请求得到数据进行处理来获得ip地址。 |
|  | 主动性方法 | 被动 |
|  | 多态性 | None |
|  | 消息发送 |  |
|  | 操作流程 |  |
|  | 其他 |  |
|  | } |  |
|  | { |  |
|  | 特征标记 | send\_req(hostname : bytes , qtype : int) |
|  | 操作解释 | 该操作主要是根据域名的类型来向域名服务器发送解析请求。 |
|  | 主动性方法 | 被动 |
|  | 多态性 |  |
|  | 消息发送 |  |
|  | 操作流程 |  |
|  | 其他 |  |
|  | } |  |
|  | { |  |
|  | 特征标记 | add\_to\_loop(loop : Eventloop) |
|  | 操作解释 | 该操作主要把该类的实例添加到Eventloop的实例中监听域名解析请求的返回与否。 |
|  | 主动性方法 | 被动 |
|  | 多态性 | None |
|  | 消息发送 |  |
|  | 操作流程 |  |
|  | 其他 |  |
|  | } |  |
|  | { |  |
|  | 特征标记 | close() |
|  | 操作解释 | 在实例结束的时候关闭域名解析请求的连结，同时注销在IO复用Eventloop中的监听。 |
|  | 主动性方法 | 被动 |
|  | 多态性 | None |
|  | 消息发送 |  |
|  | 操作流程 |  |
|  | 其他 |  |
|  | } |  |
|  | { |  |
|  | 特征标记 | handle\_periodic() |
|  | 操作解释 | 该操作主要是定时清除域名解析的缓存。毕竟缓存的大小是有限的。 |
|  | 主动性方法 | 被动 |
|  | 多态性 | None |
|  | 消息发送 |  |
|  | 操作流程 |  |
|  | 其他 |  |
|  | } |  |
|  | { |  |
|  | 特征标记 | resolve(hostname : bytes, callback : function) |
|  | 操作解释 | 该操作主要是检查域名是否合法以及是否已经可以从本地获取。若不可以从本地获取，则向域名服务器发送解析请求。同时该操作还会记录该解析请求返回以后要进行的操作。 |
|  | 主动性方法 | 被动 |
|  | 多态性 | None |
|  | 消息发送 |  |
|  | 操作流程 |  |
|  | 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 |
|  | } |  |

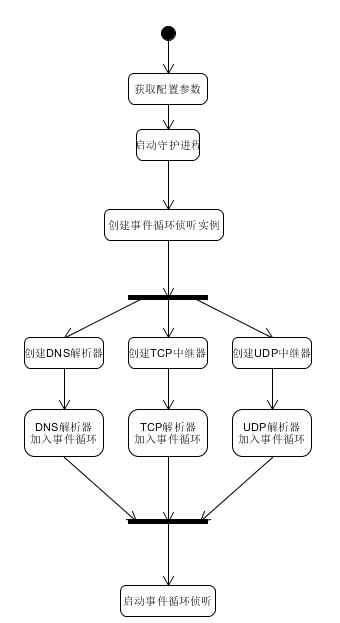
|  |  |  |
| --- | --- | --- |
| 类的总体说明 |  |  |
|  | 类名 | DNSResponse |
|  | 解释 | 这个类保存了域名解析请求返回的结果中，解析出来的域名 |
|  | 一般类 | No |
|  | 主动性 | No |
|  | 持久性 | No |
|  | 辅助模型 |  |
|  | 其他 |  |
| 属性说明 |  |  |
|  | { |  |
|  | 名称与数据类型 | answers : list |
|  | 属性解释 | 该属性保留了应答报文的地址类型、查询类型和记录类型 |
|  | 多态性 | No |
|  | 关联、聚合或组合 | No |
|  | 其它 |  |
|  | } |  |
|  | { |  |
|  | 名称与数据类型 | hostname : bytes |
|  | 属性解释 | 该属性保留域名解析请求返回的ip地址 |
|  | 多态性 | No |
|  | 关联、聚合或组合 | No |
|  | 其它 |  |
|  | } |  |
|  | { |  |
|  | 名称与数据类型 | question : list |
|  | 属性解释 | 该属性保留了请求报文的地址类型、查询类型和记录类型 |
|  | 多态性 |  |
|  | 关联、聚合或组合 |  |
|  | 其它 |  |
|  | } |  |
| 对象实例说明 | { |  |
|  | 处理机 |  |
|  | 内存对象 |  |
|  | 外存对象 |  |
|  | } |  |

|  |  |  |
| --- | --- | --- |
| 类的总体说明 |  |  |
|  | 类名 | <中文>客户端：<英文>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 | No |
|  | 主动性 | Yes |
|  | 持久性 | Yes |
|  | 辅助模型 | No |
|  | 其他 | No |
| 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 | No |
|  | } |  |
|  | { |  |
|  | 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 | No |
|  | } |  |
|  | { |  |
|  | 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 | No |
|  | } |  |
| 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 | None |
|  | 消息发送 | None |
|  | 操作流程 | None |
|  | 其他 | None |
|  | } |  |
|  | { |  |
|  | 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 | None |
|  | 消息发送 | None |
|  | 操作流程 | None |
|  | 其他 | None |
|  | } |  |
|  | { |  |
|  | 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 | None |
|  | 消息发送 | None |
|  | 操作流程 | None |
|  | 其他 | None |
|  | } |  |
|  | { |  |
|  | 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 | 主动 |
|  | 多态性 | None |
|  | 消息发送 | None |
|  | 操作流程 | 活动图 |
|  | 其他 | None |
|  | } |  |
| 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 |
|  | } |  |

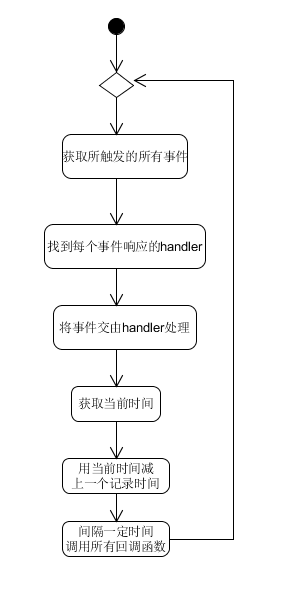
|  |  |  |
| --- | --- | --- |
| 类的总体说明 |  |  |
|  | 类名 | <中文>事件循环：<英文>EventLoop |
|  | 解释 | 这个一个侦听事件的类，用于将socket所侦听的事件注册进去，不断循环侦听，触发事件将调用相应的handler处理。 |
|  | 一般类 | No |
|  | 主动性 | No |
|  | 持久性 | Yes |
|  | 辅助模型 | No |
|  | 其他 | No |
| 属性说明 |  |  |
|  | { |  |
|  | 名称与数据类型 | model: string |
|  | 属性解释 | 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 | No |
|  | 其它 | No |
|  | } |  |
|  | { |  |
|  | 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 | No |
|  | } |  |
| 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 | None |
|  | 消息发送 | None |
|  | 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 | None |
|  | 消息发送 | None |
|  | 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 | None |
|  | 消息发送 | None |
|  | 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 | None |
|  | 消息发送 | None |
|  | 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 | None |
|  | 消息发送 | None |
|  | 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 | None |
|  | 消息发送 | None |
|  | 操作流程 | None |
|  | 其他 | None |
|  | } |  |
|  | { |  |
|  | 特征标记 | stop() |
|  | 操作解释 | 暂停IO接口复用，即EventLoop暂停了事件侦听。 |
|  | 主动性方法 | 被动 |
|  | 多态性 | None |
|  | 消息发送 | None |
|  | 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 | None |
|  | 消息发送 | None |
|  | 操作流程 | None |
|  | 其他 | None |
|  | } |  |
| 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 |
|  | } |  |

## 辅助模型

本项目主要用到的辅助模型包括

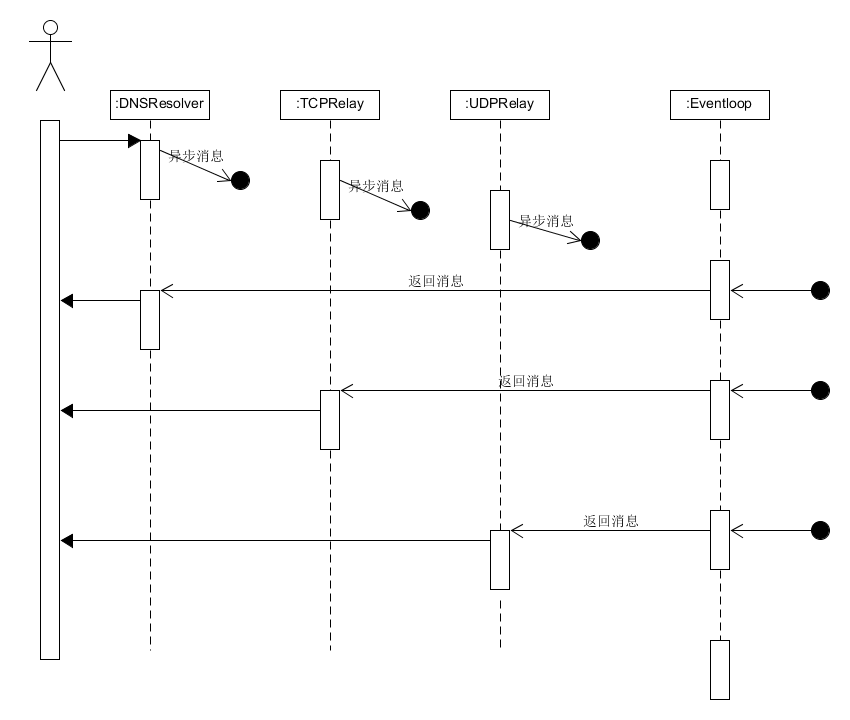


Local.main()活动图

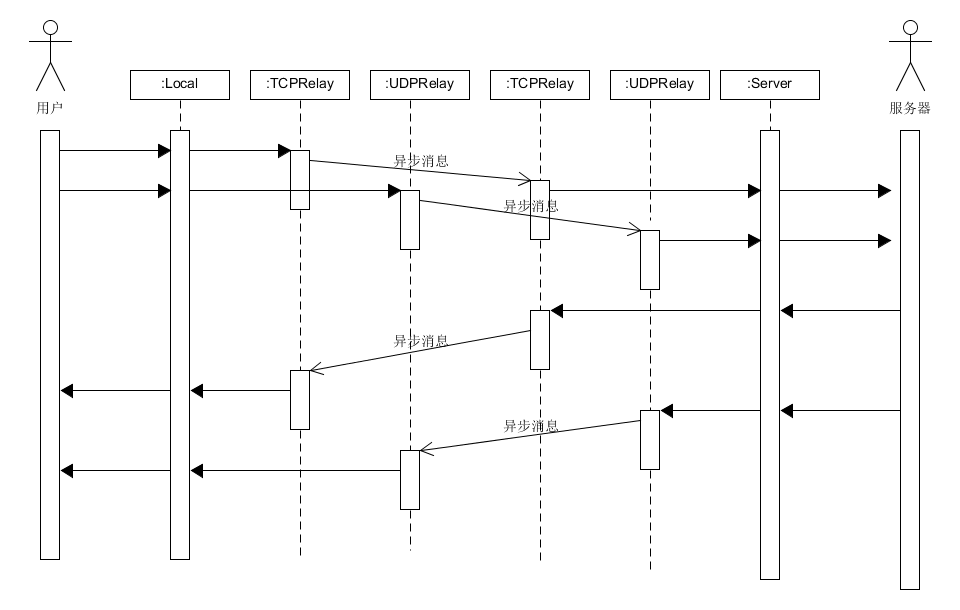


Eventloop.run()活动图

顺序图：

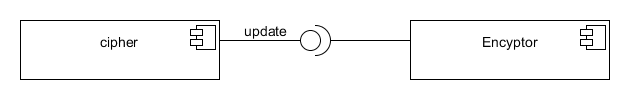


IO复用的顺序图

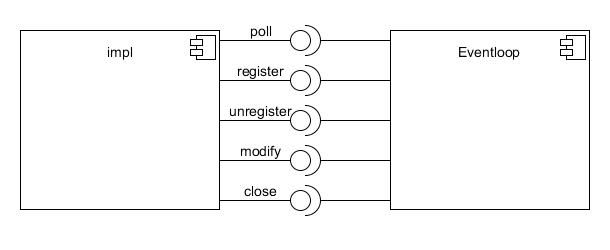


用户和服务器之间的包转发顺序图

构建图：



加密器的构建图



IO复用的构建图

# 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

本应用所采用的编程语言是Python。Python语言是一款跨平台的编程语言，可以在windows以及Linux操作系统以命令行的方式运行程序。

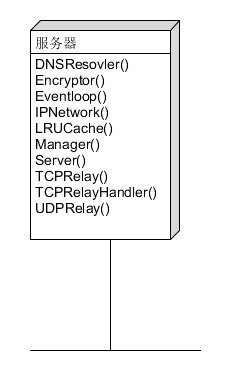
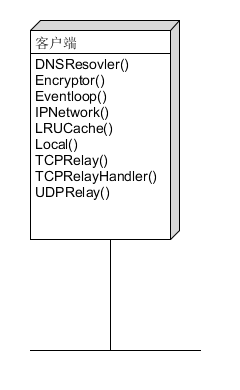
## 数据接口部分

本应用需要在计算机上存储服务器和客户端的配置信息，因此需要设计一个持久化存储的方案。纵观本应用的配置信息，完全可以用一个文本的描述来存储这些信息。因此本应用直接应用文件系统来创建一个文本文件来存储配置信息。另外采用json序列化的方法可以很好地从文本文件中提取出信息。

## 控制驱动部分

1. 计算机硬件：本应用是一款对内存、外存和CPU处理能力要求不高的应用。目前主流的PC都可以很好地运行本应用。
2. 操作系统：目前主流的两大PC操作系统：windows和Linux都具有很好的多进程和多线程的支持，并且也支持进程之间的通信（IPC）以及远程过程调用（RPC）。本应用需要进行网络通信，因此无论是IPC还是RPC都是必需的一部分。不过目前这两个系统都有很好的支持了。
3. 网络方案：本应用所需的网络环境要能够支持经典的TCP、UDP/IP协议，也就是要支持OSI七层模型中的物理层、数据链路层、网络层、传输层。同时本应用并没有对网络带宽的要求，而通信的速度则取决于本地带宽和服务器带宽的较小者。
4. 软件体系结构：本应用的设计是一个很经典的客户-服务器体系结构。因为本应用的背景是是同墙外的服务器进行通信，很自然地采用了墙外墙内各自放置服务器和客户端的设置。这一种二层客户-服务器体系结构，客户端和服务器是两个界限分明的层次。
5. 根据软件体系结构锁定地系统分布方案分成两部分，一部分是对象分布；另外一部分是类分布。

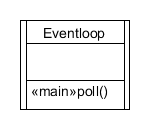
首先是对象分布图。



从中我们可以看到对于服务器端和客户端而言，大部分的对象都是需要，不同的地方仅仅在于各自扮演的角色所需的启动类。如两者相比，客户端只有Local()，而服务器只有Manager()和Server()。两者并没有交集。

其次是类分布图。

1. 从用况当中可以看出我们需要设计一个控制流来进行不同计算机之间的网络通信。本应用利用IO复用的机制设计了一个主动类：Eventloop来监听网络通信事件。因此该控制流可以被表示成下列类图：



这个是本应用实现网络通信的核心所在，也是实现进程同步的核心所在。

## 人机交互部分

本应用的人机交互部分并没有涉及到。因为本应用还是一款基于命令行界面的python应用，并没有使用类似windows的GUI界面。因此在这一部分，交互的内容无从谈起。

# 总结

总结