|  |
| --- |
| ZTEsoft中兴软创 |
|  |

|  |
| --- |
|  |
|  |
| OLC详解及实现 |
|  |
|  |
| **作 者：余俊新 编写日期：2013-3-15** |
| **文档版本：V 1.0** |

# 引言

## 文档说明

本文档从OLC入手，讲解了在线流程的架构，及消息流向和实现。

虽然本文档是以OLC来讲解的，实际上其中的一些概念和实现机制也适用于OCS、PCRF系统。因为OCS和PCRF所基于的zxos，与OLC一样使用zxos协议进行消息交互。

本文档主要面向在线流程开发人员，包括OLC、OCS、PCRF等，但是文档内容不涉及具

体业务。

## 编写目的

相对于业务来说，了解在线流程实现机制的人员较少，且编程所需技能与业务开发存在

较大差异，而这方面文档几乎没有，造成了难以入门和提升。

另外，在PCRF系统中，同时面临着OLC&OCS、PHub&PCRF、SHub&SPR，在线流程变得更为复杂，难以理解。

所以，特地编写此文档用于解决上述问题。

## 必备技能

至少掌握下述3方面知识，如果不具备，需要先了解一下：

1. 基本的TCP/IP编程知识，主要是socket编程
2. 多线程编程知识
3. Unix/Linux消息队列机制

# 功能简介

## 组网图

### 计费域



OLC充当OCS与各外部在线计费上报网元的接口，处理网元发送的在线计费相关的请求，转换成OCS内部使用的的协议（DCC协议），发送给OCS；接受OCS返回的CCA，转换成各网元使用的协议，发送到网元

OLC与计费上报网元一般采用TCP通讯，也有UDP或SSL，OLC和OCS通过zxos协议通讯

### PCC域（PCRF）



在PCRF中，OLC在这里起到PHub的作用

PHub充当PCRF与外部PCC网元的接口，在外部PCC网元与PCRF系统之间分发PCC消息

PHub与外部PCC网元采用TCP通讯，PHub与PCRF通过zxos协议通讯

### PCC域（SPR）



在SPR中，OLC在这里起到SHub的作用

SHub充当SPR与PCRF的接口，在SPR与PCRF系统之间分发PCC消息

SHub与PCRF采用TCP通讯，SHub与SPR通过zxos协议通讯

## 功能概述

### 平台层

由操作系统相关的系统级功能构成，它屏蔽了不同操作系统的细节，为应用提供统一的调用系统API的接口，并提供部分通用的服务和管理程序：监控、启动、停止等。主要包括：

1. 线程管理。OLC采用基于多线程协作处理的架构，负责线程的创建、销毁、健康监

测等

1. 链路管理。对OLC与外部网元、OLC与OCS的链路进行管理，如定时检测链路是否

正常、链路是否积压、断链重连等。支持TCP、UDP、SSL。

1. 消息队列管理。线程之间使用消息队列进行通信
2. 共享内存管理。负责OLC平台中的共享内存变量，以及各service接口的共享内存

变量的创建

1. 定时器服务。为各种需要定时触发的事件提供服务，如定时输出统计数据等
2. 日志管理。提供统一的日志输出接口，控制日志文件生成的个数、大小等
3. 配置管理。负责配置文件的读取和写入
4. 系统内核资源管理。如线程锁、信号量等

### 业务层

1. 协议支持。实现与具体协议相关的功能，如接口协议编码解码、协议转换，主要侧

重于协议流程的处理，满足与网元对接的需求。常用协议有：Diameter、Radius、SMPP+、EMPP等。

1. 号码跟踪。支持针对指定的号段（或单个号码）进行跟踪，将该号段的所有消息保

存下来，以解决可能出现的单个用户异常或用户投诉问题。

1. 路由分发。在多OCS的情况下，OLC支持按配置的路由规则将不同的计费请求包发

送到不同的OCS系统。路由规则一般以计费号码所属的号段作为分发的依据。

1. Caps限制。
2. 数据库访问。支持ORACLE、QMDB。

#### service接口

业务层包含如下常用的接口，其它项目私有协议就不列出了

1、DCC接口：处理DCC协议的接口，包括计费域、PCC域

2、Radius接口：处理Radius协议的接口

3、Smpp+接口：此接口为中国移动制定的协议，用于点对点短信业务中

4、Empp接口：此协议为中国移动制定的协议，用于移动梦网短信业务

5、320接口：用于在UIP与OCS、UIP与PCRF之间分发一次性算费等消息

# 关键技术

## zxos协议

OLC和zxos使用该协议，用于同一节点中的进程或线程间通讯，也可用于不同节点中

的进程或线程间通讯。它包括zxos消息头和zxos消息体：

1、zxos消息头的作用在于在不同主机或同一主机的不同线程（或进程）之间分发消息

2、zxos消息体是业务需要关心的部分，可以为CCR、CCA或其它业务数据

zxos协议有如下几个比较重要的概念：

1. 模块号：用来区分不同的zxos节点，一个zxos节点对应一个模块号。同一模块号

之内的线程（或进程）使用消息队列通信，如位于不同模块则使用socket通信

1. 局号：对于目前的应用来说无意义，但是如果要配置对端zxos节点的局号时，还是

需要保持配置一致

3、Pno号：用来区分同一模块号内的线程（或进程），为线程（或进程）在zxos节点内

的逻辑编号

通过module+pno，即可在多个zxos节点中唯一的标识出某个线程（或进程）

这里只列出zxos消息头中的关键字段：

//zxos节点中一个线程（或进程）的PID信息，主要就是模块号和pno号

typedef struct

{

UINT16 pno //线程（或进程）在zxos节点内的逻辑编号

UINT8 module; //zxos节点的模块号

UINT8 postOffice; //局号，没多大用处，不过需要保持配置一致

} PID;

//zxos消息头结构，消息头后面就是消息体的内容

typedef struct

{

PID sender; //消息发送方的PID

PID receiver; //消息接收方的PID

UINT16 event; //发送的消息的事件ID

UINT16 len; //消息体的字节数

} MSG\_HEAD;

对于使用zxos协议来通讯的双方，接收方收到的消息中是带有zxos头的

1. 如果发送方和接收方在同一个zxos节点，那么接收方收到的消息中，zxos消息头中

的sender即为发送方的PID。例外的是下面第2点。

1. socket发送线程和接收线程在转发zxos消息时，不会改变zxos消息头中的sender

和receiver。例外的是下面第3点。

1. OLC的socket接收线程收到zxos消息时，在此连接中如果OLC作为zxos服务端，

那么会将zxos消息头中的sender改为本线程的pno（本来是不修改的，是后来320接口做的修改，看能不能把这个修改去掉）。

比如OLC的service380线程ASEND数据给OCS的OCDis，中间会经过OLC->OCS的socket发送线程和zxcomm的socket接收线程，在OCDis收到数据时：

zxos消息头中的sender为：module为OLC的，pno为380（service380线程）

zxos消息头中的receiver为：module为OCS的，pno为20（OCDis进程）

## 多线程架构

OLC采用多线程架构，其中比较重要的一点是，从socket收发数据的操作与具体的业务

处理逻辑是分离的，因此系统中主要分两类线程：

1. socket线程。一个socket连接有两个线程，socket接收线程负责从对端recv数据，

socket发送线程负责send数据到对端。

1. 业务处理线程。即通常所见的service380、service366等。

socket接收线程收到网元的消息后，需要将该消息转发给业务线程来处理；业务线程处

理完消息后，需要将消息转发给socket发送线程，从而发送到网元。

## 消息队列

OLC一共有3个消息队列：

1. 消息队列0

生产者：为socket接收线程，在收到对端（OCS或外部网元）的数据后，将数据放入到

消息队列0

消费者：为业务处理线程，如service380，从消息队列0取消息进行处理

1. 消息队列1

生产者：为业务处理线程，如service380，处理完消息后，将待转发到网元的消息放入

到消息队列1

消费者：为socket发送线程，从该队列取消息，通过socket发送到网元

1. 消息队列2

用于impmonitor进程对OLC的各线程进行健康检查，一般不需要关注

发送到消息队列的消息，实际上为一个指针，该指针指向具体的数据，包括zxos消息

头以及zxos数据体

## socket线程与业务线程

如果使用zxos连接，则视其为内部网元；如果使用非zxos连接，则视其为外部网元

下面以四种情况来说明socket线程与业务线程的关联关系，分别为：

1. 收到外部网元的数据。比如SCP->OLC。
2. 发送数据到内部网元。比如OLC->OCS、PHub->PCRF、SHub->SPR、OLC->UIP。
3. 收到内部网元的数据。比如OCS->OLC、PCRF->Phub、SPR->SHub、UIP->OLC。
4. 发送数据到外部网元。比如OLC->SCP。

### 收到外部网元的数据（jdr：根据配置找相应的业务线程）

socket接收线程收到外部网元的数据时，如何知道转发给哪个业务线程？

对于此种情况，itcom.ini中[server]或[client]配置项下的mainpno指定了socket线程与业务线程的绑定关系

### 收到内部网元的数据（jdr：根据消息头找业务线程）

socket线程收到内部网元的数据时，如何知道转发给哪个业务线程？

此时收到的数据中带有zxos消息头，zxos消息头中的receiver即标识了处理该消息的业务线程

### 发送数据到内部网元(jdr:ocl作为客户端，去servicexxx.ini中去寻找服务器端，ocl作为服务器端，则其请求的链路中的modle,就是要发送的那个model)

业务线程处理完数据转发到内部网元时，如何知道转发给哪个socket发送线程？

首先需要得到目标内部网元的module。如果OLC作为zxos客户端，对端module即为servicexxx.ini中[OCS]配置节点下的module；如果OLC作为zxos服务端，则取对应请求所来自的链路的module

取到目标内部网元的module后，根据该module与各socket链路的对端module匹配，匹配到的链路的pno即标识了socket发送线程

即：发送方根据接收方的module号查找到对应的socket链路，将消息发送到匹配到的链路的socket发送线程，然后由该socket发送线程send数据到对端zxos节点

如果OLC作为zxos客户端，可以通过配置文件静态分析出来：在servicexxx.ini下取[OCS]配置节点下的module，再根据该module号在itcom.ini中与[client]配置项下的peermodule进行匹配，匹配到的[client]下的commpno即标识了对应的socket发送线程

### 发送数据到外部网元

业务线程处理完数据转发到外部网元时，如何知道转发给哪个socket发送线程？

如果为发送响应到外部网元，比如发送CCA到SCP，那么取出请求所来自的链路的pno即可；(jdr:外部有请求，这里经过处理，回应相应的请求)

如果为主动触发请求到外部网元，比如将OCS触发的RAR发送到SCP，那么根据RAR中的OriginHost（网元名称），取出该网元的已建立的链路的pno即可。

# 在线流程

这里通过一个业务场景举例，来描述一个消息在如下环节的详细流向：

SCP -> OLC -> OCS (zxcomm -> OCDis -> OCPro -> zxcomm) -> OLC -> SCP

对于PCRF，流向也是一样的，和上面的流程可以一一找到对应关系，只是业务实体不同而已，此时流向为：

PCEF -> OLC -> PCRF (zxcomm -> PDis -> PDE -> zxcomm) -> OLC -> PCEF

如果以后引入了其它网元，比如SPR，流向也是如此，只是具体的网元不同而已

## 业务场景举例

假如某局点有使用Smpp+协议的SMSC，以及使用Diameter协议的SCP，按照协议：

1、在Smpp+协议中，OLC作为TCP客户端，SMSC作为TCP服务端

2、在Diameter协议中，OLC作为TCP服务端，SCP作为TCP客户端

3、系统内部实现时，OLC与OCS使用zxos协议通讯，OLC作为zxos客户端，OCS作为zxos服务端

在实施时，假如进行如下网络配置：

1. SMSC的IP地址为10.45.4.2，开放5555端口供OLC连接
2. OLC的IP地址为10.45.4.1，开放6000端口供SCP连接
3. OCS的IP地址为10.45.4.3，开放7000端口供OLC连接
4. OLC模块号为157，OCS模块号为136

注意：在TCP通讯中，只需要关注TCP服务端的端口号，TCP客户端的端口号一般是不需要关心的，由程序在运行时随机获取一个可用的端口号。如果有不理解，需要了解一下基本的网络编程知识。

## 配置及说明

这里并不是一个实际可运行的配置的说明，只列出核心配置，主要侧重于线程交互

注意，在这里忽略了局号postoffice的配置，实际配置时是需要与OCS保持一致的

### 配置指引

在对某个业务接口进行配置时，OLC会扮演如下角色之一，下面分别说明各种情况下

如何进行配置。

jdr:

OLC作为普通的客户端或者服务器端的时候：

在itcom的client中配置，说明这是OLC是客户端，需要配置对端的ip和端口，以供OLC连接使用。

在itcom的service项中配置，说明这是OLC是服务器，需要配置本段的的ip和端口，以供需要连接本服务器的客户端使用

#### OLC作为普通客户端（jdr：两itcomm和servicexx都要配置）

1、在itcom.ini下的[client]节点配置相关信息，主要是对端IP和服务端口等

2、在servicexxx.ini下的[ImpClient]节点下配置相关信息，主要是对端IP

#### OLC作为普通服务端

1、在itcom.ini下的[server]节点配置相关信息，主要是本端IP和服务端口等

2、在servicexxx.ini下的[ImpClient]节点下配置相关信息，主要是对端IP（jdr：说明允许哪些连接）

#### OLC作为zxos客户端

1、在itcom.ini下的[server]节点配置相关信息，主要是本端IP和服务端口等

(jdr:同一ip环境下？？)

2、在servicexxx.ini中对zxos服务端进行配置，主要是zxos服务端的module和目标进

程的Pno号

比如servicexxx转发CCR到OCS，则需要在servicexxx.ini的[OCSx]下配置目标OCS的模块号，Pno号指定为OCDis的pno号

又比如service333转发CCR到PCRF，那么需要在service333ini的[OCSx]（这个配置名不准确，应该改为[PCRFx]）下配置目标PCRF的模块号，Pno号指定为PDis的pno号

#### OLC作为zxos服务端

1、在itcom.ini下的[server]节点配置相关信息，主要是本端IP和服务端口等

2、在servicexxx.ini下的[ImpClient]节点下配置相关信息，主要是对端IP

#### 注意点

1、servicexxx.ini下的[ImpClient]节点，并不表示对端是通讯的客户端。这个配置项名称不准确。实际上，只要对端网元不为zxos服务端，都需要在[ImpClient]下对其进行配置。如果对端网元为zxos服务端，则在servicexxx.ini的[OCS]节点下进行配置，主要是配置对端的IP、module、目标进程的pno。jdr:这里对应着service362.ini 和service380.ini的配置

2、注意，servicexxx.ini的[OCS]配置要和itcom.ini中的[client]配置保持一致，可能会有疑问，itcom.ini中在[client]节点下已经配置了对端的module、IP，为什么在servicexxx.ini的[OCS]下，还要配置对端的module、IP呢？

这个过程是这样的：

a、service线程根据路由配置找到目标OCS的IP地址，然后根据该IP地址去匹配servicexxx.ini中[OCS]节点下的IP地址，由此得到目标OCS所对应的[OCS]节点配置。然后取出匹配节点下的module，以及pno，然后将消息ASEND到该[module,pno]组合。

b、ASEND的逻辑在os平台层中执行。此时service线程根据目标OCS的module与各socket链路的对端module匹配，从而找到到目标OCS的socket链路，然后将消息发送到该链路的socket发送线程，然后由该socket发送线程send数据到对端zxos节点。

由此可见，servicexxx.ini中的zxos服务端配置（即[OCS]配置节点），主要是给service线程根据目标zxos节点的IP地址，得到接收方的module及pno；

而itcom.ini中的zxos服务端配置（位于[client]配置节点），用于os平台层通过socket将zxos消息发送到前面所找到的[module,pno]所标识的接收方。

### imp.ini

[general]

Interfacecount = 2 #配置的待加载的servicexxx的个数

[interface1]

mainpno = 362 #Smpp+接口的pno号，固定为362

dllname = service362.dll # smpp+接口的名称，固定为service362.dll

[interface2]

mainpno = 380 #DCC接口的pno号，固定为380

dllname = service380.dll #DCC接口的名称，固定为service380.dll

### itcom.ini

[general]

Module = 157 #OLC的模块号

Myipaddress = 10.45.4.1 #OLC的IP地址

Servercnt = 1 #OLC开放的服务端口的个数

Clientcnt = 2 #OLC连接的服务端口的个数

[server1] #描述一个服务端配置，该服务端接收SCP的连接

#此服务端口所绑定的service线程的pno号，对于该服务端口所监听到的socket

连接，如果收到socket数据，将会转发给该pno号的service线程来处理

#注意，对于zxos连接该配置无效，因为对于zxos连接，已经在收到的数据的消息

头中指定了service线程的pno号。此时随便填一个300到399之间的值

Mainpno = 380 （jdr：关联业务线程）

Myipaddress = 10.45.4.1 #OLC的IP地址

Port = 6000 #OLC开放的服务端口

Socketnum = 10 #该端口所能建立的连接的个数

[client1] #描述一个客户端连接的配置，这里为OLC->OCS

#此客户端连接所绑定的service线程的pno号，对于该socket连接，如果收到socket

数据，将会转发给该pno号的service线程来处理

#注意，对于zxos连接该配置无效，原因同[server]的此配置项的描述

Mainpno = 300

Myipaddress = 10.45.4.1 #OLC的IP地址

Peeripaddress = 10.45.4.3 #服务端，即OCS的IP地址

Port = 7000 #服务端，即OCS开放端口号

#对于zxos连接，指定对端（即OCS）的模块号，需要与OCS的win\_mgt.ini中配置的OCS的模块号保持一致

Peermodule = 136

Longconn = 1 #连接类型，如TCP、UDP、SSL等，这里为TCP

#为该客户端连接预先分配的pno号，每个client连接必须有自己唯一的线程号

Commpno = 101

[client2] #描述一个客户连接的配置，这里为OLC->SMSC

Mainpno = 362 #同上 （jdr:关联业务线程）

Myipaddress = 10.45.4.1 #同上

Peeripaddress = 10.45.4.2 #服务端IP地址，即SMSC的IP地址

Port = 5555 #服务端开放的端口号，即SMSC开放的端口号

Longconn = 1 #同上

Commpno = 103 #同上

### service362.ini（OLC作为客户端时候，需要填写这个地方）

[OCS1] # OCS的配置信息

IPAddr = 10.45.4.3 #OCS的IP地址

Module = 136 #OCS的模块号

Pno = 20 #指定目标进程（这里为OCDis）的Pno号

[ImpClient1]

IPAddr = 10.45.4.1 #SMSC的IP地址

### service380.ini

[OCS1] # OCS的配置信息，配置同service362.ini

IPAddr = 10.45.4.3

Module = 136

Pno = 20

[ImpClient1]

IPAddr = 10.45.4.2 #SCP的IP地址

## 流程（jdr:这里初始化 链路建立 看不懂）

在这里列出的函数的具体逻辑可参考“5 代码实现”部分

### OLC初始化

1. main函数中，首先建立3个消息队列，然后调用IMPInit函数
2. IMPInit函数中：

InitOSConfig和InitComConfig分别读取imp.ini和itcom.ini配置信息并保存

InitDllFunc函数加载service362和service380动态库

InitPCB函数中，创建各种平台级线程（其中包括通讯总控线程），以及service业务线

程，这些线程的入口函数都为IMPOS\_Thread

注意：在IMPOS\_Thread函数中，收到消息后的处理函数如下：

如果是通讯总控线程，则其处理消息的函数为IMPOS\_CommThread

如果是service业务线程，则其处理消息的函数为servicexxx.c中定义的service380

或service362函数

1. 线程启动完毕后，IMPInit函数中会向所有线程发送初始化事件InitAllProcessEvent

Service业务线程收到该消息后，进行初始化处理，主要是执行InitMsgService函数等

通讯总控线程收到该消息后，主要是执行InitTCPIP函数，以及创建一个定时检查链路

的线程

1. InitTCPIP函数中：

对于每个服务端创建一个CreateServer监听线程，在该线程中，每监听到一个来自客户

端的连接，则分别为之创建一个socket接收线程和socket发送线程，pno号在0到99之间循环分配

对于每个客户端，创建一个socket接收线程和socket发送线程，pno号来自于itcom.ini

中[client]下的commpno配置值（位于100到255之间）

即，初始化完成后，主要创建了如下线程（其它线程未全部列出）：

通讯总控线程IMPOS\_CommThread，pno号固定为269

socket101：分别有一个socket发送线程和socket接收线程，pno号为配置值101

socket103：分别有一个socket发送线程和socket接收线程，pno号为配置值103

service362线程：pno号为362

service380线程：pno号为380

另外对于itcom.ini中每个[server]配置项，有一个CreateServer监听线程，没有pno号，运行时不受平台管理

### 链路建立

在运行时，如果监听到来自SCP的连接，即CreateServer线程中监听到SCP的连接：

1. 为该socket连接分别创建一个socket接收线程（入口函数为ReceiveData）和一

个socket发送线程（入口函数为ReceiveSendData），并为之分配一个空闲的pno号（假如为0）。

1. 在ReceiveData线程中，会向其关联的业务线程service380发送建链通知消息

IMP\_CREAT\_SOCKET\_SUCCESS\_EVENT

1. service380线程收到该消息后，执行SetSocket函数
2. SetSocket函数中，根据对端IP地址，判断其网元索引为1；另外为之分配一个空

闲的链路索引（假如为0）；同时建立网元+链路与pno的对应关系，即EntityIdx=1&&SktIdx=0的链路的Pno为0。网元索引的含义，见“5.3.4.2 相关名词”。

OLC作为客户端建立连接的过程也是一样的，只不过连接是在OLC启动时即建链，时机

不同而已。例如，对于socket103的建链通知，则为：

1. 在ReceiveData线程中，会向其关联的业务线程service362发送建链通知消息

IMP\_CREAT\_SOCKET\_SUCCESS\_EVENT

1. service362线程收到该消息后，执行SetSocket函数
2. SetSocket函数中，根据对端IP地址，判断其网元索引为1；另外为之分配一个空

闲的链路索引，假如为0；同时建立网元+链路与pno的对应关系，即EntityIdx=1&&SktIdx=0的链路的Pno为103。

### 链路断链

假如与SCP的连接socket0断链：

1. 在断链时，socket接收线程在ReceiveData函数中recv会返回错误码，此线程发

EXIT消息给socket发送线程（发送线程收到该消息时会退出），同时发断链通知IMP\_CLEAR\_SOCKET\_SUCCESS\_EVENT到关联的业务线程service380

1. service380线程收到该消息后，执行ClearSocket函数
2. ClearSocket函数中，根据对端IP地址，判断其网元索引为1；另外根据断链的

socket的pno号0来查找链路索引SktIdx，找出来的索引为0；然后清理EntityIdx=1&&SktIdx=0的socket节点的信息，主要是解除其与pno=0的绑定，及相关信息

### 消息收发（jdr:重点）（jdr: （jdr:在OLC内部的消息转发，都是用ASCEND在各个模块间使用SOCKET,比如OLC和OCS之间，OLC和SCP之间等等）

这里以SCP发送CCR到OLC，然后OLC转发CCR到OCS，OLC收到OCS响应的CCA，并

转发CCA到SCP的过程来说明。完整的过程如下图：



下面是按步骤分解之后的3张序列图：

#### OLC处理CCR

 1、socket0接收线程在ReceiveData函数中，阻塞在socket的recv调用上，等待SCP

的DCC数据

2、socket0接收线程收到SCP的数据，recv调用返回，然后将该消息ASEND到关联的service线程即service380

3、service380线程收到该消息后，调用service380函数，判断出事件号为IMP\_SENDMSG\_EVENT，则执行DealFromComm，该函数通过GetLinkInfo获取到消息所来自的网元索引Entityidx、链路索引SktIdx，如果该链路的缓冲中具备一个完整的CCR，则对该CCR进行处理，如将该请求的HashKey、Entityidx和SktIdx保存到hash节点，加OCDis分发头，非DCC接口会进行协议转换，查找目标OCS的路由（即获取EntityIdx，从而获取到目标OCS的module，以及目标进程的pno），等

4、service380线程处理完该CCR后，将该CCR ASEND到目标OCS（模块号和Pno号来源于servicexxx.ini中[OCS]配置节点的module、pno）。这里目标OCS的模块号为136，Pno为20。发送方PID设置为OLC的模块号157，pno号为380。（jdr：发送方数据，和OLC自身有关，可以直接写入到数据中发送方的模块好和pno号）

此时在ASEND函数中，由于接收方PID的模块号和OLC的模块号不同，所以根据接收方模块号136查找对端模块号为136的socket链路，然后取出该socket链路的pno号（通过itcom.ini中的配置可以看出，对端模块号为136的socket连接的pno号为101），所以最终将该CCR发送到pno为101的socket的发送线程

5、socket101发送线程收到service380发送过来的消息，通过socket的send函数，将该zxos消息发送到OCS

#### OCS处理



1. OCS的zxcomm进程的socket接收线程在ReceiveData函数中，阻塞在socket的

recv调用上，等待来自于OLC的数据

2、zxcomm接收线程收到OLC的数据，recv调用返回，由于收到的数据的zxos消息头中接收方的pno为20，所以zxcomm接收线程将该CCR 发送到OCDis进程

3、OCDis进程按照分发算法，将该CCR分发到某OCPro进程，同时将发送方PID（module为OLC的模块号，pno为service380的pno号380）也告知给OCPro进程

4、OCPro进程对CCR进行批价处理，并编码CCA

5、OCPro进程将CCA ASEND回给发送方（非OCDis进程，而是第3步所保存的OLC的module，pno为380）

在ASEND函数中，判断出接收方的module不同于本节点，因此根据接收方（OLC）的module查找socket链路，将该CCA send\_msg到OCS->OLC的socket发送线程

6、OCS->OLC的socket发送线程将该zxos消息通过socket send到OLC

#### OLC处理CCA

 1、OLC的socket101接收线程在ReceiveData函数中，阻塞在socket的recv调用上，

等待OCS的数据

2、socket101接收线程收到OCS的数据，recv调用返回，如果该链路的缓冲中已经具备一个完整的zxos消息，则将该消息ASEND到zxos消息头中指定的接收方（pno=380），即将该CCA ASEND到service380线程

3、service380线程收到该消息后，调用service380函数，判断出事件号为FROM\_OCS\_EVENT，则执行DealMsgFromOCS

该函数中，如果非DCC接口，会进行协议转换，将CCA转换为对应的协议

4、service380线程从hash中查找对应请求所来自的EntityIdx、SktIdx，从而取出该链路对应的pno（见“4.3.2 链路建立”，在SetSocket函数中会保存EntityIdx、SktIdx与pno的关系，此例中为0），然后将该CCA ASEND到该pno=0的线程

在ASEND函数中，由于接收方PID非zxos节点（模块号为0），所以该CCA会send\_msg到pno为0的socket发送线程

5、socket0发送线程收到service380发送过来的消息，通过socket的send函数，将该CCA发送到SCP

# 代码实现

下面列出的函数及说明，基本上涵盖了OLC的核心逻辑和流程，这里只列出关键代码

片段。

## 源文件说明

1、os目录。该目录主要提供平台层相关的功能，具体功能见“2.2.1 平台层”部分，关键代码文件如下：

impcomm.c：实现socket通讯方面的逻辑

impos.c：实现平台初始化逻辑，以及定义平台级线程的入口函数

impfunc.c：封装消息队列、线程管理及交互等方面的逻辑

imptimer.c：实现定时器功能

impmonitor.c：监控进程的实现逻辑

imptool.c：imptool的实现逻辑

1. servicexxx目录。主要包含业务线程的逻辑，提供对协议接口的支持，如协议解析、

协议编码等。

1. proc\_diameter目录。提供协议转换逻辑，主要是各种外部网元使用的协议到DCC

协议的互相转换。

4、route目录。提供路由功能，根据计费号码获取待分发的OCS的IP地址。

5、setimp文件。根据输入的命令行参数，建立OLC的编译环境。

5、hash目录。开源代码，主要为servicexxx提供hash功能。

6、helper目录。来源于框架代码，主要为业务层提供数据结构、算法等功能。

7、xmlparser目录。开源代码，主要为servicexxx提供XML解析功能。

## 关键全局变量

### 配置信息相关变量

保存imp.ini中一个[interface]配置项信息的结构

typedef struct

{

UINT16 mainpno; 保存imp.ini中[interface]下的mainpno配置值

UINT8 dllname[50]; 保存imp.ini中[interface]下的dllname配置值

} InterConfigStruc;

保存imp.ini中所有[interface]配置项信息

**InterConfigStruc g\_InterCfg[MAX\_INTERFACE\_NUM];**

保存itcom.ini中一个[server]或[client]配置项信息的结构

typedef struct

{

UINT16 mainpno; 保存itcom.ini中[server]或[client]下的mainpno配置值

UINT32 myipaddress; 保存itcom.ini中[server]或[client]下的myipaddress配置值

UINT32 peeripaddress; 保存itcom.ini中[client]下的peeripaddress配置值

UINT16 port; 保存itcom.ini中[server] 或[client]下的port配置值

UINT8 peermodule; 保存itcom.ini中[client]下的peermodule配置值

UINT16 commpno; 保存itcom.ini中[client]下的commpno配置值

} PeerMap;

保存itcom.ini中所有[server]配置项信息

**PeerMap pPeerMapServer[MAX\_SERVER\_SOCKET\_NUM];**

保存itcom.ini中所有[client]配置项信息

**PeerMap pPeerMapClient[MAX\_CLIENT\_SOCKET\_NUM];**

### 运行时相关变量

为业务线程消息处理函数的声明，如service362、service380函数

typedef void far (\* LPENTRY)(UINT8 far \*, UINT8 far \*, UINT8 far \*);

保存所有业务线程消息处理函数的地址，如service362、service380函数，g\_lpEntry[380]即为service380函数的地址

**LPENTRY g\_lpEntry[PCB\_NUM];**

保存系统中一个线程的相关信息

typedef struct

{

UINT16 CurEvent; 本线程当前处理的消息的事件ID

PID SelfPid; 保存本线程的PID

PID CurSender; 保存本线程当前处理的消息的发送方的PID

INT64 RunNum; 本线程已处理的消息的个数

INT64 nMsg; 待本线程处理的消息的个数

THREAD\_ID dwThread; 本线程的线程ID（如果此PCB对应socket线程，则为socket发送线程的ID）

THREAD\_ID dwThread1; 本线程的线程ID（如果此PCB对应socket线程，则为socket接收线程的ID）

void \* CurMsg; 本线程当前处理的消息

} PCBStruc;

保存系统中运行的所有线程的信息，Pno即为此数组中的索引，可见Pno即为线程在zxos系统中的逻辑编号

**PCBStruc far \* PCB[PCB\_NUM];**

保存系统中一个socket连接及对应线程的相关信息

typedef struct

{

SOCKET Socket; 该连接的socket

UINT32 myipaddress; socket连接对应的OLC的地址

UINT32 PeerIPAddr; socket连接对应的对端的地址

UINT16 Pno; socket连接对应的线程的Pno号

UINT8 PeerNode; 如果为zxos连接，则为socket连接的对端的模块号，否则为0

UINT8 longconn; socket类型，如TCP、UDP、SSL

THREAD\_ID RecePid; socket接收线程的线程ID

THREAD\_ID SendPid; socket发送线程的线程ID

与该socket连接关联的业务线程的pno，socket接收线程从socket上recv到数据后，会将数据转发给该线程来进行业务处理，其值来源于itcom.ini中[server]或[client]下的mainpno配置

UINT16 ReceiverPno;

UINT16 SenderPno; 对于服务端socket，则为监听socket，否则为0

PID SendPID; 对于zxos连接，为socket接收线程当前收到的zxos消息的消息头中的sender

UINT32 ReceLeftLen; 本socket连接的接收缓存中的数据的字节数

UINT8 ReceBuff[MAX\_NEW\_LEN]; 本socket连接从socket上recv数据的缓存

} SocketStatusNode;

保存系统中所有socket连接及对应线程的相关信息，一个socket连接在其中占用一个节点

此索引与PCB数组中的索引是一一对应的

0到MAX\_SERVER\_SOCKET\_NUM为server使用，MAX\_SERVER\_SOCKET\_NUM到MAX\_SOCKET\_NUM为client使用

**SocketStatusNode \* SocketStatus[PCB\_NUM];**

## 关键函数

### OLC初始化

#### Main函数

main(int argc, char \*argv[])

{

//保证一个用户下只能启动一个OLC实例

lockfile((UINT8 \*)IMP\_LOCKNAME);

//建立OLC所需的3个消息队列

msqid[0] = mkmsq(getkey((UINT8 \*)MSQKEY));

msqid[1] = mkmsq(getkey((UINT8 \*)MSQKEY1));

msqid[2] = mkmsq(getkey((UINT8 \*)MSQKEY2));

//建立共享内存

shmid = mkshm(getkey((UINT8 \*)SHMKEY), SHMSIZE);

//建立共享内存

shmid1 = mkshm(getkey((UINT8 \*)SHMKEY1), SHMSIZE1);

**IMPInit();**

}

#### IMPInit函数

void IMPInit(void)

{

//读取imp.ini配置文

**InitOSConfig**();

//读取itcom.ini配置文件

**InitComConfig();**

**InitDllFunc**();

**InitPCB();**

//睡眠5秒，等待线程创建完毕

Sleep(5000);

//向所有线程发送初始化消息

SendToAllThread(InitAllProcessEvent);

}

#### InitOSConfig函数

主要是读取imp.ini配置文件，并将配置信息保存到全局变量g\_InterCfg，逻辑较简单，

这里不再描述

#### InitComConfig函数

主要是读取itcom.ini配置文件，并将配置信息保存到全局变量pPeerMapServer、

pPeerMapClient，逻辑较简单，这里不再描述

#### InitDllFunc函数

主要是根据保存imp.ini配置信息的变量g\_InterCfg，从中获取协议接口线程的消息处理

函数的地址，保存到全局变量**g\_lpEntry中，**逻辑较简单，这里不再描述，只列出关键代码片段

//装载协议接口的动态库，如service362.dll、service380.dll

hDllHandle = LoadLibrary((char \*)g\_InterCfg[i].dllname);

//构造协议接口线程的入口函数的名称

sprintf((char \*)sDllName, "service%d", g\_InterCfg[i].mainpno);

//从之前装载进来的动态库中，获取协议接口线程的消息处理函数（如service380函数）的地址

g\_lpEntry[iPno] = (LPENTRY)GetProcAddress(hDllHandle, (char \*)sDllName);

#### InitPCB函数

UINT8 InitPCB(void)

{

//初始化总控模块的PCB，即本线程

PCB[MAIN\_CONTROL\_PNO].SelfPid.pno = MAIN\_CONTROL\_PNO;

PCB[MAIN\_CONTROL\_PNO].dwThread = GetCurrentThreadId();

PCB[MAIN\_CONTROL\_PNO].hThread = &PCB[MAIN\_CONTROL\_PNO].dwThread;

PCB[MAIN\_CONTROL\_PNO].SelfPid.module = CUR\_MODULE();

//初始化通讯总控线程的PCB，该线程的PNO固定为MAIN\_COMM\_PNO

CREATE\_THREAD(**IMPOS\_Thread**, &PCB[MAIN\_COMM\_PNO].dwThread, &PCB[MAIN\_COMM\_PNO].hThread);

PCB[MAIN\_COMM\_PNO].SelfPid.module = CUR\_MODULE();

PCB[MAIN\_COMM\_PNO].SelfPid.pno = MAIN\_COMM\_PNO;

//初始化各协议接口线程的PCB，这些线程的Pno来源于imp.ini中[interface]下MainPno

for (i = 0; i < InterfaceCount; i++)

{

iPno = g\_InterCfg[i].mainpno;

CREATE\_THREAD(**IMPOS\_Thread**, &PCB[iPno].dwThread, &PCB[iPno].hThread);

PCB[iPno].SelfPid.module = CUR\_MODULE();

PCB[iPno].SelfPid.pno = iPno;

}

}

#### IMPOS\_Thread函数

//lpPno来源于CREATE\_THREAD所传递的参数，其值为该线程的Pno，如创建service380线程时则为380

void IMPOS\_Thread(LPVOID lpPno)

{

PTR\_ADDR pMsgHead;

//从消息队列获取的消息的内容，实际为一个指向消息的内容的指针，消息的长度在zxos消息头中指定

char sMsgBuf[100];

PCBStruc far \* PCBPtr;

MSG\_HEAD far \* MsgPtr;

UINT16 iPno = \*(UINT16 \*)lpPno;

//取消息处理函数的指针

LPENTRY lpEntry;

PCBPtr = (PCBStruc far \*)&PCB[iPno];

if (iPno == MAIN\_COMM\_PNO)

lpEntry = (LPENTRY)**IMPOS\_CommThread**; //通讯总控线程的地址

else if (iPno >= SDF\_START\_PNO && iPno < PCB\_NUM)

lpEntry = g\_lpEntry[iPno]; //即service380等函数的地址

while (1)

{

//从消息队列获取消息，注意这里的第二个参数，其作用就是各线程如service380线程只获取发送给//它的消息，而不会获取到发送给service362的消息

if (accept\_msg(msqid[0], iPno + 1, sMsgBuf, sizeof(sMsgBuf), …) <= 0)

continue;

//为监控进程发送的健康检查消息，直接响应即可

if (strncmp(sMsgBuf, "TEST", 4) == 0)

{

send\_msg(msqid[2], MONITOR\_TYPE, sMsgBuf, 5, TRANSFER\_NOWAIT, TRANSFER\_DATA);

continue;

}

memcpy(&pMsgHead,sMsgBuf,PTR\_LEN);

MsgPtr = (MSG\_HEAD \*)(\*(PTR\_ADDR \*)sMsgBuf);

PCBPtr->CurMsg = MsgPtr;

PCBPtr->CurEvent = MsgPtr->event;

PCBPtr->CurSender = MsgPtr->sender;

PCBPtr->RunNum ++;

//lpEntry，即为service380等函数的地址

//MsgPtr+1，即zxos消息头后面的数据，对于service380接口则为CCR/CCA等，对于接口service362则为SMPP+格式的数据

//在这里就进入了常见的如service380.c中的处理逻辑了

**(\* lpEntry)((UINT8 far \*)(MsgPtr + 1), …);**

}

}

### Socket链路

#### IMPOS\_CommThread函数

void far IMPOS\_CommThread(UINT8 far \* lpMsg, UINT8 far \* lpNULL0, UINT8 far \* lpNULL1)

{

switch (EVENT())

{

case InitAllProcessEvent:

{

//对于服务端则创建socket服务器监听线程

//对于客户端则连接socket，并对每个socket连接创建一个socket发送线程和一个socket接收线程

**InitTCPIP**();

//创建定时检查链路的线程，作用如断链自动重连等，入口函数为CheckIfLinkConnect

CREATE\_THREAD((HANDLE)CheckIfLinkConnect,...);

//创建定时检查链路的定时器

SET\_LOOP\_TIMER(50, TIMER1);

break;

}

case Timer1Event:

{

...//定时对zxos链路进行检查

}

case IMP\_ACK\_CLEAR\_SOCKET\_EVENT:

{

memcpy(&iPno, lpMsg, 2);

shutdown(SocketStatus[iPno].Socket,2);

closesocket(SocketStatus[iPno].Socket);

break;

}

case IMP\_CLEAR\_SOCKET\_EVENT:

{

memcpy(&iPno, lpMsg, 2);

ClearSocketConnect(iPno);

break;

}

...

}

return;

}

#### InitTCPIP函数

UINT8 InitTCPIP(void)

{

SOCKADDR\_IN remote\_sin, local\_sin;

//对itcom.ini中的每一个[server]配置，创建一个服务端监听线程

for (i=0;i<ServerCnt;i++)

CREATE\_THREAD((HANDLE)**CreateServer**, (LPVOID)&i, ...);

//对itcom.ini中的每一个[client]配置，创建一个socket连接

//对每一个socket连接，创建一个socket发送线程ReceiveSendData和一个socket接收线程ReceiveData

for (i = 0; i < ClientCnt; i ++)

{

UINT16 iPno = (pPeerMapClient + i)->commpno;

SocketStatus[iPno].myipaddress = (pPeerMapClient + i)->myipaddress;

SocketStatus[iPno].PeerIPAddr = (pPeerMapClient + i)->peeripaddress;

**SocketStatus[iPno].PeerNode = (pPeerMapClient + i)->peermodule;**

**SocketStatus[iPno].ReceiverPno = (pPeerMapClient + i)->mainpno;**

SocketStatus[iPno].longconn = (pPeerMapClient + i)->longconn;

//向对端建链

SOCKET iSocket = socket(AF\_INET, SOCK\_STREAM, 0);

local\_sin.sin\_addr.s\_addr = SocketStatus[iPno].myipaddress;

bind(iSocket, (PSOCKADDR)&local\_sin, ...);

remote\_sin.sin\_port = (pPeerMapClient + i)->port;

remote\_sin.sin\_addr.s\_addr = (pPeerMapClient + i)->peeripaddress;

connect\_nonb(iSocket, (PSOCKADDR)&remote\_sin, ...);

SocketStatus[iPno].Socket = iSocket;

//建链成功后，分别创建一个socket发送线程ReceiveSendData和一个socket接收线程ReceiveData

CREATE\_THREAD(**ReceiveData**, &iPno, 0, &PCB[iPno].dwThread1, …);

CREATE\_THREAD(**ReceiveSendData**, &iPno, 0, &PCB[iPno].dwThread, ...);

SocketStatus[iPno].RecePid = PCB[iPno].dwThread1;

SocketStatus[iPno].SendPid = PCB[iPno].dwThread;

}

}

#### CreateServer函数

// iParam为传入的此服务在itcom.ini中[server]配置项的索引

//为了便于描述此函数功能，此函数包含了实际代码中的AcceptConnect、SocketInsertQueue的逻辑

DWORD WINAPI CreateServer(LPVOID iParam)

{

UINT16 iServerIdx = \*(UINT16 \*)iParam;

//创建监听socket，并在服务端口进行监听

SOCKET ListenSocket = socket(AF\_INET, SOCK\_STREAM, 0);

SOCKADDR\_IN local\_sin;

local\_sin.sin\_addr.s\_addr = (pPeerMapServer + iServerIdx)->myipaddress;

local\_sin.sin\_port = (pPeerMapServer + iServerIdx)->port;

bind(ListenSocket, (PSOCKADDR)&local\_sin, …);

listen(ListenSocket, MAX\_PENDING\_CONNECTS);

while (1)

{

//在该函数阻塞，直到监听到来自客户端连接

SOCKET iSocket = accept(ListenSocket, ...);

//在0到99之间，分配一个空闲的Pno

UINT8 iPno = FindUnUsedPno();

SocketStatus[iPno].Socket = iSocket;

SocketStatus[iPno].PeerIPAddr = PeerIPAddr;

SocketStatus[iPno].myipaddress = (pPeerMapServer + iServerIdx)->myipaddress;

**SocketStatus[iPno].ReceiverPno = (pPeerMapServer + iServerIdx)->mainpno;**

**SocketStatus[iPno].PeerNode = 255;** //原因见ReceiveData中的说明

SocketStatus[iPno].SenderPno = ListenSocket;

//对每个连接，都会创建一个socket发送线程ReceiveSendData和一个socket接收线程ReceiveData

CREATE\_THREAD((HANDLE)ReceiveData, &PCB[iPno].dwThread1, ...);

CREATE\_THREAD((HANDLE)ReceiveSendData, &PCB[iPno].dwThread, ...);

SocketStatus[iPno].RecePid = PCB[iPno].dwThread1;

SocketStatus[iPno].SendPid = PCB[iPno].dwThread;

}

}

#### ReceiveData函数

//iParam参数为创建线程时传入的参数，其值为本socket线程的Pno

DWORD WINAPI ReceiveData(LPVOID iParam)

{

UINT8 MsgBuff[10240];

UINT16 iPno = \*(UINT16 \*)iParam;

SocketStatus[iPno].RecePid = GetCurrentThreadId();

PCB[iPno].SelfPid.module = CUR\_MODULE();

PCB[iPno].SelfPid.postOffice = CUR\_POST();

PCB[iPno].SelfPid.pno = iPno;

//设置消息的接收方，为此socket连接关联的业务线程的pno，如380或362等

ReceiverPID.pno = SocketStatus[i].ReceiverPno;

//通知service380或service362等线程，新建立了一条socket链路，其pno值为iPno

ASENDEX(IMP\_CREAT\_SOCKET\_SUCCESS\_EVENT, &iPno, 2, ReceiverPID, iPno);

//该socket连接采用zxos协议，即对端为zxos节点，比如OCS、UIP、PCRF

if (IS\_WINNTTCP(SocketStatus[iPno].PeerNode))

{

while (1)

{

//阻塞等待socket上的数据

INT32 ReceLen = recv(SocketStatus[iPno].Socket,MsgBuff,...);

//如果链路断链，则退出此循环，从而socket接收线程将退出

if (ReceLen == SOCKET\_ERROR || ReceLen == 0)

break;

//将SocketStatus[iPno].ReceBuff（缓存的之前收到的数据）和MsgBuff（本次收到的数据）中的//内容，拼接后拷贝到MsgBuff

//ReceLen为MsgBuff中数据的字节数

……

MSG\_HEAD \* MsgHeadPtr = (MSG\_HEAD \*)MsgBuff;

while (ReceLen > 0)

{

//如果MsgBuff未包含一个完整的zxos消息，则将其缓存到SocketStatus[iPno].ReceBuff

//SocketStatus[iPno].ReceLeftLen为缓存中数据的字节数

//然后从此层循环break，继续recv等待对端的数据

if (!bContainFullMsg)

{

memcpy(SocketStatus[iPno].ReceBuff, MsgHeadPtr, ReceLen);

SocketStatus[iPno].ReceLeftLen = ReceLen;

break;

}

//如果此连接为zxos服务端，那么链路建立时是不知道对端的模块号的，程序里会暂时设置为255，//待收到对方数据时，才会根据收到的zxos消息头中的发送方的模块号设置此连接的对端模块号

if (SocketStatus[i].PeerNode == 255)

**SocketStatus[i].PeerNode = MsgHeadPtr->sender.module;**

//包含一个完整的zxos消息

//注意，因为是zxos连接，所以这里的SendPID赋值为收到的zxos消息投中的发送方，即对端zxos节点的发送方

SocketStatus[iPno].SendPID = MsgHeadPtr->sender;

//将zxos消息体发送给zxos消息头中指定的接收方，该接收方PID由对端zxos节点的发送方设置

//比如收到OCS的数据时，OCS会将接收方的pno设置为380或362等；与UIP通讯时，UIP会将接收方的pno设置为320

ASEND(MsgHeadPtr->event, (MsgHeadPtr + 1), MsgHeadPtr->len, **MsgHeadPtr->receiver**);

//剩余未处理数据的字节数中，减去该消息的长度

ReceLen = ReceLen - sizeof(MSG\_HEAD) - MsgHeadPtr->len;

//剩余未处理数据的指针，偏移该消息的长度

MsgHeadPtr = (MSG\_HEAD \*)(MsgHeadPtr + sizeof(MSG\_HEAD) + MsgHeadPtr->len);

}

}

}

//该socket连接不采用zxos协议，即对端为外部网元，比如SCP、SMSC

else

{

while (1)

{

//阻塞等待socket上的数据

INT32 ReceLen = recv(SocketStatus[iPno].Socket, MsgBuff, ...);

//如果链路断链，则退出此循环，从而socket接收线程将退出

if (ReceLen == SOCKET\_ERROR || ReceLen == 0)

break;

//设置消息的接收方，为此socket连接关联的业务线程的pno，如380或362等

PID pid = {0};

SELF\_PID(&pid);

pid.pno = SocketStatus[iPno].ReceiverPno;

//将recv到的socket数据转发给关联的业务线程，如service380或service362线程等

//注意，这里没有链路缓存逻辑，对于非zxos连接，由业务线程根据协议实现缓存逻辑

ASEND(IMP\_SENDMSG\_EVENT, MsgBuff, (UINT16)ReceLen, pid);

}

}

//链路已中断，向socket发送线程发送EXIT消息，便于发送线程也退出

ClearSocketConnect(iPno);

}

#### ReceiveSendData函数

//iParam参数为创建线程时传入的参数，其值为本socket线程的Pno

DWORD WINAPI ReceiveSendData(LPVOID iParam)

{

UINT16 iPno = \*(UINT16 \*)iParam;

UINT8 sMsgBuf[100];

MSG\_HEAD \* MsgHeadPtr;

PCB[iPno].dwThread = GetCurrentThreadId();

while (1)

{

PCB[iPno].OSState = BLOCKK;

//在消息队列1阻塞等待消息，消息发送方通常为service362、service380等业务线程

//注意这里的第二个参数，其作用是各socket发送线程只获取发送给它的消息，而不会获取到发送给其它socket发送线程的消息

if (accept\_msg(msqid[1], iPno + 1, sMsgBuf, sizeof(sMsgBuf), …) <= 0)

continue;

//收到来自于socket接收线程退出时发出的EXIT消息，则本线程也退出

if (strncmp(sMsgBuf, "EXIT", 4) == 0)

break;

PCB[iPno].OSState = RUNN;

PCB[iPno].CurMsg = MsgHeadPtr;

PCB[iPno].RunNum ++;

MsgHeadPtr = (MSG\_HEAD \*)(\*(PTR\_ADDR \*)sMsgBuf);

//下面的send函数本来是TcpSend函数（包含了socket发送缓冲满的处理逻辑），为了突出重点，这里直接写为send函数

//如果对端为zxos节点，如OCS、UIP、PCRF等，则发送到socket的数据中需要包含zxos消息头

if (IS\_VALID\_NODE(SocketStatus[iPno].PeerNode))

send(SocketStatus[iPno].Socket, MsgHeadPtr, sizeof(MSG\_HEAD)+MsgHeadPtr->len);

//如果对端非zxos节点，如SMSC、SCP等，则发送到socket的数据中不能包含zxos消息头

else

send(SocketStatus[iPno].Socket, MsgHeadPtr+1, MsgHeadPtr->len, ...)

}

}

#### ClearSocketConnect函数

//iPno为socket线程的Pno

void ClearSocketConnect(UINT16 iPno)

{

//关闭socket

shutdown(SocketStatus[iPno].Socket, 2);

closesocket(SocketStatus[iPno].Socket);

//如果对应的socket发送线程未退出，则向其发送EXIT消息，发送线程收到该消息后将会退出

if (PCB[iPno].hThread != NULL)

send\_msg(msqid[1], iPno+1, "EXIT", ...);

//设置消息的接收方，为此socket连接关联的业务线程的pno，如380或362等

PID ReceiverPID = {0};

ReceiverPID.pno = SocketStatus[iPno].ReceiverPno;

//通知service380或service362等线程，pno值为iPno的socket链路断链

ASEND(IMP\_CLEAR\_SOCKET\_SUCCESS\_EVENT, &nInfo, sizeof(NodeInfo), ReceiverPID);

...

}

### 消息发送

#### ASEND函数

//将消息发送到接收方，为zxos系统中各线程进行消息交互的核心函数

//Event：事件ID

//In：待发送的消息

//InLen：待发送的消息的长度

//ReceiverPID：消息接收方的PID

UINT8 ASEND(UINT16 Event, UINT8 \*In, UINT16 InLen, PID ReceiverPID)

{

PCBStruc far \* SelfPCBPtr = GetCurPCB();//消息发送方的PCB，即本线程的PCB

MSG\_HEAD far \* MsgPtr;

PTR\_ADDR pMsgHead;

UINT16 iReceiverPno = ReceiverPID.pno;

//如果接收方位于zxos节点内，但不在本节点，则需要将消息发送到与其通信的socket链路，此时查找与其连接的链路的Pno

if (ReceiverPID.module!=CUR\_MODULE() && ReceiverPID.module!=0)

{

iReceiverPno = PCB\_NUM;

for (i = 0; i < MAX\_SOCKET\_NUM; i ++)

{

//如果该socket的对端节点的模块号等于接收方的模块号，则找的就是该socket连接

if (SocketStatus[i].PeerNode == ReceiverPID.module)

iReceiverPno = i, break;

}

}

//开始构造zxos消息头

MsgPtr = (MSG\_HEAD far \*)GetUB(sizeof(MSG\_HEAD)+InLen+1);

memset(MsgPtr, 0x00, sizeof(MSG\_HEAD)+InLen+1);

//如果消息发送方（本线程）为socket线程，且为zxos连接，且接收方不为service320

//即本线程对应为：OLC作为zxos客户端所建立的socket连接

if (SelfPCBPtr->SelfPid.pno < MAX\_SOCKET\_NUM

&& SocketStatus[SelfPCBPtr->SelfPid.pno].SendPID.pno != 0

&& SocketStatus[SelfPCBPtr->SelfPid.pno].ReceiverPno != 320)

{

//设置消息发送方为收到的zxos消息头中的sender，即位于对端zxos节点中的发送方

MsgPtr->sender = SocketStatus[SelfPCBPtr->SelfPid.pno].SendPID;

memset(&SocketStatus[SelfPCBPtr->SelfPid.pno].SendPID, 0, sizeof(PID));

}

//即本线程非对应为：OLC作为zxos客户端所建立的socket连接

else

{

//设置消息发送方为本线程的PID

MsgPtr->sender = SelfPCBPtr->SelfPid;

}

MsgPtr->event = Event;

MsgPtr->receiver = ReceiverPID;

MsgPtr->len = InLen;

//将待发送的消息，复制到zxos消息体

memcpy(MsgPtr+1, In, InLen);

pMsgHead = (PTR\_ADDR)MsgPtr;

//如果接收方为socket线程，则将此zxos消息放入到消息队列1

if (iPno < MAX\_SOCKET\_NUM)

status = send\_msg(msqid[1], iPno + 1, (UINT8 \*)&pMsgHead, PTR\_LEN, ...);

//如果接收方不是socket线程，比如为service380，则将此zxos消息放入到消息队列0

else

status = send\_msg(msqid[0], iPno + 1, (UINT8 \*)&pMsgHead, PTR\_LEN, ...);

//将消息接收方的待处理的消息个数+1

PCBStruc \* ReceiverPCBPtr = (PCBStruc \*)&PCB[iReceiverPno];

++ PCBPtr->nMsg;

}

### service接口

#### 相关链路结构和变量

//servicexxx.ini下[OCS]、[ImpClient]配置中的网元信息

typedef struct

{

UINT8 iType; //网元类型：0-zxos服务端，1-外部网元或zxos客户端

UINT8 iMaxSktNum; //该网元可以建链的最大连接数

UINT32 iIPAddr; //网元的IP地址

SocketInf\_T tSktInf[MAX\_SKT\_PER\_CLIENT]; //网元的链路信息

}EntityInf\_T;

//网元建立的链路信息，主要目的有2：

//保存业务请求所来自的链路的PID，以便在响应时沿原链路返回

//在未收到一个完整的业务请求时，将这部分不完整数据进行缓存

typedef struct

{

UINT8 iUseFlag;

PID PID; //该链路的PID

MsgBuf\_T RemainBuf; //该链路的数据接收缓存

}SocketInf\_T;

//保存一个service接口中所有网元，以及其中的链路的信息

**static EntityInf\_T g\_EntityInf[MAX\_ENTITY\_NUM];**

#### 相关名词

网元索引：该网元在servicexxx.ini中[OCS]、[ImpClient]配置节点下的索引。即在

g\_EntityInf数组中的索引。

链路索引：在网元的建立的所有链路中，每个链路也存在一个索引。即该链路在对应的

g\_EntityInf节点的tSktInf数组中的索引。

#### servicexxx函数

// service接口线程的业务处理函数

// lpMsg：消息的内容，如SMPP+协议数据、CCR/CCA，或建链/断链通知等

void far servicexxx(UINT8 far \* lpMsg, UINT8 far \* lpNULL0, UINT8 far \* lpNULL1)

{

UINT16 iEvent = EVENT();

UINT16 iMsgLen = LENGTH();

switch (iEvent)

{

//初始化消息，读取servicexxx.ini配置文件

case InitAllProcessEvent:

**InitMsgService**();

SET\_TIMER(50,TIMER1);

break;

//创建一个定时器，用于定时检查链路是否正常

case Timer1Event:

CheckLinkStatus();

SET\_TIMER(CHECK\_LINK\_TIMER, TIMER1);

break;

//与此service关联的socket连接的建链通知

case IMP\_CREAT\_SOCKET\_SUCCESS\_EVENT:

**SetSocket**();

break;

//与此service关联的socket连接的断链通知

case IMP\_CLEAR\_SOCKET\_SUCCESS\_EVENT:

**ClearSocket**(lpMsg);

break;

//收到来自外部网元的socket数据，如CCR、SMPP+请求等

//对于非DCC协议，该函数进行协议解析并转换成CCR，加上OCDis分发头后，发送给OCS

//对于DCC协议，加上OCDis分发头后，透传给OCS

case IMP\_SENDMSG\_EVENT:

**DealFromComm**(lpMsg, iMsgLen);

break;

//收到来自OCS等zxos节点的socket数据，如CCA等

//对于非DCC协议，该函数进行协议转换，发送给外部网元如SMSC

//对于DCC协议，则透传给SCP等

case FROM\_OCS\_EVENT:

**DealMsgFromOCS**(lpMsg,iMsgLen);

break;

}

}

#### InitMsgService函数

int InitMsgService(void)

{

UINT8 sTmpStr[20]={0},sValueBuf[100]={0};

g\_Config.iOCSNum = fGetPrivateProfileInt("general", "OCSNum", "servicexxx.ini");

g\_Config.iClientNum = fGetPrivateProfileInt("general", "ClientNum", "servicexxx.ini");

//zxos服务端的配置，只建立一条链路，module、pno都是已知且必须配置的，所以直接在socket节点中预先设置模块号、pno号

for(i=0; i<g\_Config.iOCSNum; i++)

{

sprintf(sTmpStr, "OCS%d", (i+1));

fGetPrivateProfileString(sTmpStr,"IPAddr",NULL,sValueBuf,16,"servicexxx.ini");

g\_EntityInf[i].iType = 0;

g\_EntityInf[i].iIPAddr = inet\_addr(sValueBuf);

g\_EntityInf[i].iMaxSktNum = fGetPrivateProfileInt(sTmpStr,"MaxSktNum","servicexxx.ini");

g\_EntityInf[i].tSktInf[0].PID.module = fGetPrivateInt(sTmpStr,"Module","servicexxx.ini");

g\_EntityInf[i].tSktInf[0].PID.pno = fGetProfileInt(sTmpStr,"Pno","servicexxx.ini");

}

//zxos客户端或外部网元，对端的module、pno在初始是未知的，只有在运行期才知道，在SetSocket时设定模块号、pno号

for(i=g\_Config.iOCSNum; i<g\_Config.iOCSNum+g\_Config.iClientNum; i++)

{

sprintf(sTmpStr, "ImpClient%d", (i+1-g\_Config.iOCSNum));

fGetPrivateProfileString(sTmpStr,"IPAddr",NULL,sValueBuf,16,"servicexxx.ini");

g\_EntityInf[i].iType = 1;

g\_EntityInf[i].iIPAddr = inet\_addr(sValueBuf);

g\_EntityInf[i].iMaxSktNum = fGetPrivateProfileInt(sTmpStr,"MaxSktNum","servicexxx.ini");

for(j=0;j<g\_EntityInf[i].iMaxSktNum;j++)

memset(&g\_EntityInf[i].tSktInf[j].PID, 0, sizeof(g\_EntityInf[i].tSktInf[j].PID));

}

}

#### SetSocket函数

//处理链路建立后，socket线程所发送的建链通知

void SetSocket()

{

int iEntityIdx=0,iSktIdx=0;

NodeInfo NInfo;

PID pId;

//获取socket线程的相关信息，主要是取对端的IP地址和socket线程的pno号

GET\_NODEINFO(&NInfo);

SENDER(&pId);

//根据对端的IP地址，查找该网元的网元索引

for(iEntityIdx=0;iEntityIdx<g\_Config.iOCSNum+g\_Config.iClientNum;iEntityIdx++)

if(g\_EntityInf[iEntityIdx].iIPAddr == NInfo.PeerIPAddr) break;

//该IP未在servicexxx.ini中配置，则对其断链

if(iEntityIdx == g\_Config.iOCSNum+g\_Config.iClientNum)

CLEAR\_CONNECT(pId.pno), return;

//对端为zxos服务端，如OCS、PCRF

if (g\_EntityInf[iEntityIdx].iType == 0)

{

g\_EntityInf[iEntityIdx].tSktInf[0].iUseFlag=1;

//不需要设置PID信息，此时直接取[OCS]配置项下配置的module、pno

}

//对端为外部网元，或zxos客户端，如SMSC、SCP、UIP等

else

{

for(iSktIdx=0;iSktIdx<g\_EntityInf[iEntityIdx].iMaxSktNum;iSktIdx++)

{

//在该网元中找到一条空闲未用的socket节点

if(g\_EntityInf[iEntityIdx].tSktInf[iSktIdx].iUseFlag==0)

{

g\_EntityInf[iEntityIdx].tSktInf[iSktIdx].iUseFlag=1;

//设置该socket节点的pno号

g\_EntityInf[iEntityIdx].tSktInf[iSktIdx].PID.pno=NInfo.Pno[0];

break;

}

}

//如果该网元建链个数超过配置，则清除该链路

if(iSktIdx==g\_EntityInf[iEntityIdx].iMaxSktNum)

CLEAR\_CONNECT(pId.pno), return;

}

}

#### ClearSocket函数

//处理链路断链后，socket线程所发送的断链通知

void ClearSocket(UINT8 \* pMsg)

{

int iEntityIdx=0,iSktIdx=0;

//pMsg为socket线程的相关信息，主要是对端的IP地址和socket线程的pno号

NodeInfo \*pNInfo=(NodeInfo \*)pMsg;

//根据对端的IP地址，查找该网元的网元索引

for(iEntityIdx=0;iEntityIdx<g\_Config.iOCSNum+g\_Config.iClientNum;iEntityIdx++)

if(g\_EntityInf[iEntityIdx].iIPAddr == pNInfo->PeerIPAddr) break;

//该IP未在servicexxx.ini中配置，退出

if(iEntityIdx == g\_Config.iOCSNum+g\_Config.iClientNum)

return;

//对端为zxos服务端，如OCS、PCRF

if (g\_EntityInf[iEntityIdx].iType == 0)

{

g\_EntityInf[iEntityIdx].tSktInf[0].iUseFlag=0;

}

else

{

//根据pno号，寻找该socket连接的链路索引，然后清理该socket节点的信息

for(iSktIdx=0;iSktIdx<g\_EntityInf[iEntityIdx].iMaxSktNum;iSktIdx++)

{

if(g\_EntityInf[iEntityIdx].tSktInf[iSktIdx].PID.pno==pNInfo->Pno[0])

{

g\_EntityInf[iEntityIdx].tSktInf[iSktIdx].iUseFlag=0;

memset(&g\_EntityInf[iEntityIdx].tSktInf[iSktIdx].PID, 0, …);

break;

}

}

}

}

#### GetLinkInfo函数

//获取消息所来自的链路的网元索引及链路索引

//这两个索引会保存到hash中，这样OLC在回响应给网元时，就能沿着请求所来自的链路发回响应

int GetLinkInfo(int \*piEntityIdx, int \*piSktIdx)

{

int iEntityIdx=0,iSktIdx=0;

NodeInfo NInfo;

PID pId;

//获取socket线程的相关信息，主要是取对端的IP地址和线程的pno号

GET\_NODEINFO(&NInfo);

SENDER(&pId);

//根据对端的IP地址，查找该网元的网元索引

for(EntityIdx=g\_Config.iOCSNum;EntityIdx<g\_Config.iOCSNum+g\_Config.iClientNum;EntityIdx++)

if(g\_EntityInf[iEntityIdx].iIPAddr==NInfo.PeerIPAddr)

break;

//该IP未在servicexxx.ini中配置，退出

if(iEntityIdx == g\_Config.iOCSNum+g\_Config.iClientNum)

return -1;

//根据pno号，寻找该socket连接的链路索引

for(iSktIdx=0;iSktIdx<g\_EntityInf[iEntityIdx].iMaxSktNum;iSktIdx++)

if(g\_EntityInf[iEntityIdx].tSktInf[iSktIdx].PID.pno==NInfo.Pno[0])

break;

\*piEntityIdx=iEntityIdx;

\*piSktIdx=iSktIdx;

}

#### GetOCSIndex函数

//获取路由的函数，实际上就是查找该CCR应该分发到哪个OCS，即OCS的网元索引

//获取到目标OCS的网元索引后，即可获取到配置的对应的模块号

int GetOCSIndex(char \* pNumber)

{

int iEntityIdx;

char sAddress[20]={0},sPort[10]={0};

//非分布式，OCS只有一个，直接找iType为0的网元

if(g\_Config.iOCSNum==1)

{

for(iEntityIdx=0;iEntityIdx<g\_Config.iOCSNum;iEntityIdx++)

if(g\_EntityInf[iEntityIdx].iType==0)

return iEntityIdx;

}

//分布式，OCS有多个，根据CCR中号码来路由

else if (g\_Config.iOCSNum > 1)

{

FOCS\_GET\_ROUTE(pNumber,sOCSIpAddr,sOCSPort);

for(iEntityIdx=0;iEntityIdx<g\_Config.iOCSNum;iEntityIdx++)

{

if((g\_EntityInf[iEntityIdx].iIPAddr==inet\_addr(sAddress))

&&(g\_EntityInf[iEntityIdx].iType==0))

return iEntityIdx;

}

}

//未找到路由

return -1;

}

#### DealFromComm函数

void DealFromComm(UINT8 \* pMsg, UINT16 iMsgLen)

{

int iEntityIdx,iSktIdx;

GetLinkInfo(&iEntityIdx,&iSktIdx);

//1、如果此接口对应的连接非zxos，那么需要在service接口中的此处进行消息解析和缓存

//如果是zxos连接，则不需要进行了，因为已经在ReceiveData函数中进行了

//2、如果非DCC协议接口，这里进行协议转换，将非DCC计费请求转换为CCR

//在hash中保存了此CCR的主键，以及对应的iEntityIdx、iSktIdx，这样可以将响应沿原来链路返回

//3、将CCR前面加上OCDis分发头，假如为sCCR，字节数为sMsgLen

//4、获取此CCR待分发到的OCS的网元索引

iEntityIdx = GetOCSIndex(sSubsNumber);

//下面的函数，实际上是通过SendToOCS调用的

//将此CCR，发送到OCS，这个具体发送过程可以看下ASEND函数

ASEND(TO\_OCS\_EVENT, sCCR, sMsgLen, g\_EntityInf[iEntityIdx].tSktInf[0].PID)

}

#### DealMsgFromOCS函数

void DealMsgFromOCS(UINT8 \*IpMsg,UINT16 iMsgLen)

{

//1、如果非DCC接口，这里进行协议转换，将CCR转换为对应的协议

//2、从hash中查找对应的请求的相关信息，比如iEntityIdx、iSktIdx

//下面这行，实际上是通过SendToClient调用的

ASEND(IMP\_SENDMSG\_EVENT, IpMsg, iMsgLen, g\_EntityInf[iEntityIdx].tSktInf[iSktIdx].PID));

}

# 建议阅读资料

1. 编译、部署、运行方面的资料，请参考OLC实施文档
2. TCP/IP详解 卷1：协议
3. UNIX网络编程（第1卷）。主要是socket api编程的介绍
4. UNIX网络编程（第2卷：进程间通信）
5. 线程编程，可自行搜索“POSIX线程编程”