

代码剖析

读取步骤为:

1. 全局变量
2. main()
3. loadConfig()
4. CreateBaseFolder()
5. LoadTemplate()
6. SetLogonRequest()
7. SocketLogon()
8. DataHandOut()

全局变量

config:

config 全局变量代表了设置好在 /home/sysop/ExchangeData/SH/config 目录下的 SHData_Cedar.cfg 所设置的内容

可以通过 cat 命令查看，vim 命令修改

```
[sysop@localhost config]$ cat SHData_Cedar.cfg
BASE=
{
    IP           = "10.118.8.40";
    Port         = "9929";
    ProtocolEdition = "STEP.1.0.0";
    SenderCompID  = "cedarsender";
    TragetCompID  = "cedartarget";
    EbcryptMethod = "0";
    HeartBtInt    = "10";
    Template      = "/home/sysop/ExchangeData/SH/template/template.2.20.xml";
    Path          = "/home/sysop/ExchangeData/SH/result/"
};
```

```
//config
char ip[16];
int port;
std::string p_ProtocolEdition;
std::string p_SenderCompID;
std::string p_TragetCompID;
int p_EbcryptMethod;
int p_HeartBtInt;
std::string l_template_path;
```

```
std::string l_fast_template;  
std::string export_folder;
```

定义exchCode 和 套接字的长度

```
// 上海证券交易所代码  
int exchCode = 1;  
  
// socket buffer length  
// 分配给上证15MB  
int socket_buffer_length = 15 * 1024 * 1024; // 15MB
```

backup

全局定义了一个信号量，用来进程控制

定义了一个 marketManager 对象

```
// backup  
// 信号量  
sem_t close_mutex;  
// marketmanager 对象 g_ 全局  
MarketManager g_MarketManager;
```

文件流

定义了三个文件流，分别负责 Quote, Trade以及Index

```
// 文件流，类型为FILE*  
FILE* SH_Quote_File = NULL;  
FILE* SH_Trade_File = NULL;  
FILE* SH_Index_File = NULL;
```

定义了计数器

三个计数器，用于做 实际储存条数和writeXXXCount的逻辑判断，当write次数较大时，先不写入

```
// Counter 计数器  
static int QuoteCount = 0;  
static int TradeCount = 0;  
static int IndexCount = 0;
```

```
// write counter 写计数器  
static int writeQuoteCount = 0;
```

```
static int writeTradeCount = 0;
static int writeIndexCount = 0;
```

时间变量

分别对应了 内置的时间结构体

```
//时间变量
// timeval 对象 成员对象有sec 和 微秒
struct timeval tv;
```

```
//struct timezone结构体
struct timezone tz;

// tm 对象, 可以获取日期和时间
struct tm* t;
```

三把锁

在后续 两种情况下, 需要用到**Quote_mutex** 锁, 来向全局变量 dataTick型数组 g_Quote 传递数据

1. CASE UA3202(竞价行情数据)
2. CASE UA3108(盘后固定价格交易行情数据)

在后续 两种情况下, 需要用到**Trade_mutex** 锁, 来向全局变量 dataTransaction型数组 g_Trade 传递数据

1. CASE UA3201 (竞价逐笔成交数据)
2. CASE UA3209 (盘后固定价格交易逐笔成交数据)

在后续 一种情况下, 需要用到**Index_mutex** 锁, 来向全局变量 dataIndex型数组 g_Index 传递数据

1. CASE UA3113 (指数行情数据)

```
//Mutex std::mutex 锁
mutex Quote_mutex;
mutex Trade_mutex;
mutex Index_mutex;
```

消息及存储用 数组

```
//message 消息长度
int MAX_LEN = 200000;
```

```
//自定义的data 类型数组
dataTick      g_Quote[200000];
dataTransaction g_Trade[200000];
dataIndex      g_Index[200000];
```

定义三个文件流的冲洗时间

```
//fflush times
// 冲洗流中信息的时间间隔
int quote_fflush = 2000;
int index_fflush = 2000;
int trade_fflush = 2000;
```

定义三个进程创建时的函数

```
//Thread
//进程 设置进程 函数
bool g_exit_thread = false;
void* saveQuoteIntoFiles(void* arg);
void* saveTradeIntoFiles(void* arg);
void* saveIndexIntoFiles(void* arg);
```

Json对象

Json 库的 Reader 和Value类

Value为实际的对象，而 Reader 是用来将字符串转换为Json::Value 对象的

```
//Json 对象 每组分别对应一个reader 一个 val
// 共计 quote, Index和trade 三组
Json::Reader QuoteReader;
Json::Value QuoteVal;
Json::Reader IndexReader;
Json::Value IndexVal;
Json::Reader TradeReader;
Json::Value TradeVal;
```

main() 注解

1. 定义了int 型 checkcode 变量
2. LoadConfig() 获取目录下 设置的cfg文件
3. CreateBaseFolder() 在相关目录下 设置目录，打开相关对应的文件，创建流
同时，如果启动时间小于91500(9:15)，会在 流中输入 设置好的行首，并强制输出缓冲区数据到文件中

4. LoadTemplate() 获取对应目录下的template.2.20.xml, 输出为string型
5. SetLogonRequest()
6. SocketLogon()
7. DataHandOut()

Null

Null

```
int main() {  
  
    int checkcode = 0;  
    LoadConfig();  
    CreateBaseFolder();  
    l_fast_template = LoadTemplate(l_template_path);  
    std::string LogonRequest_str = SetLogonRequest(p_ProtocolEdition, p_SenderCompID,  
p_TragetCompID, p_EbcryptMethod, p_HeartBtInt, checkcode);  
    int l_sockfd = SocketLogon(LogonRequest_str);  
  
    DataHandOut(l_sockfd, l_fast_template);  
    return 0;  
  
}
```

1.chekcode

```
int checkcode = 0;
```

2.LoadConfig()

```
void LoadConfig() {  
    NewCedarConfig::getInstance().loadConfigFile("/home/sysop/ExchangeData/SH/config/SHData_Cedar.cfg");  
    strcpy(ip, ConfigGetStringValue("BASE.IP").c_str());  
    port = atoi(ConfigGetStringValue("BASE.Port").c_str());  
    p_ProtocolEdition = ConfigGetStringValue("BASE.ProtocolEdition");  
    p_SenderCompID = ConfigGetStringValue("BASE.SenderCompID");  
    p_TragetCompID = ConfigGetStringValue("BASE.TragetCompID");  
    p_EbcryptMethod = atoi(ConfigGetStringValue("BASE.EbcryptMethod").c_str());  
    p_HeartBtInt = atoi(ConfigGetStringValue("BASE.HeartBtInt").c_str());  
    l_template_path = ConfigGetStringValue("BASE.Template");  
    export_folder = ConfigGetStringValue("BASE.Path").c_str();  
}
```

NewCedar related

在程序第一行会调用 /include/NewCedar 目录下的 NewCedarConfig.h

静态函数 提供了获得一个NewCedarConfig 实例的方法

```
public:
    static NewCedarConfig& getInstance(){
        static NewCedarConfig cedarCfg;
        return cedarCfg;
    }
```

在获得实例之后，会调用loadConfigFile方法

```
int loadConfigFile(std::string path) {
    try {
        cfg.readFile(path.c_str());
    } catch(const FileIOException &fioex) {
        std::cerr << "I/O error while reading config file" << path << std::endl;
        exit(EXIT_FAILURE);
    } catch(const ParseException &pex) {
        std::cerr << "Parse error at " << pex.getFile() << ":" << pex.getLine()
            << " - " << pex.getError() << std::endl;
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

在上述过程中，cfg.readFile 为头文件中<libconfig.h++> 的方法

该方法会读取指定路径下的cfg.

分别获取相关的cfg value

接下来会调用configGetStringValue 获取对应 key下的值

3.CreateBaseFolder()

该函数调用了time_tool.h 中函数来获取：

1. 当日日期
2. 当日运行时的时间

调用了folder_tool.h 中的函数来创建 目录:

1. export_folder+Trade/
2. export_folder+Quote/
3. export_folder+Index/

打开相应的文件流. 形式为 "ab",追加法

```
void CreateBaseFolder() {
    std::string day = std::to_string(GetTodayDate());
    unsigned int timeNow = GetTodayTime();
    std::string path = export_folder;

    create_folder_my(path + "Trade/");
    create_folder_my(path + "Quote/");
    create_folder_my(path + "Index/");

    SH_Quote_File = fopen((path + "Quote/" + day + "_Quote_SH.csv").c_str(), "ab");
    SH_Trade_File = fopen((path + "Trade/" + day + "_Trade_SH.csv").c_str(), "ab");
    SH_Index_File = fopen((path + "Index/" + day + "_Index_SH.csv").c_str(), "ab");

    //当前时间小于9:15分时, 会
    if (timeNow < 91500)
    {
        fprintf(SH_Quote_File,
            "InstrumentID,Date,DATATIMESTAMP,TradingPhase,HIGHLIMIT,LOWLIMIT,PRECLOSE,NUMTRADES,TOTAL\n"
            "VOLUME,TURNOVER,LASTPX,OPENPX,HIGHPX,LOWPX,TOTALBIDQTY,TOTALOFFERQTY,WAVGBIDPX,WAVGOFFERP\n"
            "X,WithdrawBidNo,WithdrawBidVol,WithdrawBidAmount,WithdrawAskNo,WithdrawAskVol,WithdrawAsk\n"
            "Amount,BuyNumber,SellNumber,BuyTradeMaxDuration,SellTradeMaxDuration,NumBidOrders,NumOffe\n"
            "rOrders,B01,BV01,BC01,S01,SV01,SC01,B02,BV02,BC02,S02,SV02,SC02,B03,BV03,BC03,S03,SV03,SC\n"
            "03,B04,BV04,BC04,S04,SV04,SC04,B05,BV05,BC05,S05,SV05,SC05,B06,BV06,BC06,S06,SV06,SC06,B0\n"
            "7,BV07,BC07,S07,SV07,SC07,B08,BV08,BC08,S08,SV08,SC08,B09,BV09,BC09,S09,SV09,SC09,B10,BV1\n"
            "0,BC10,S10,SV10,SC10\n");
        fprintf(SH_Trade_File,
            "InstrumentID,Date,Time,TradeIndex,BuyIndex,SellIndex,TradeType,BSFlag,Price,Volume\n");
        fprintf(SH_Index_File,
            "InstrumentID,Date,Time,LocalTime,DelayMS,ExchCode,TradingPhase,HIGHLIMIT,LOWLIMIT,PRECLO\n"
            "SE,NUMTRADES,TOTALVOLUME,TURNOVER,LASTPX,OPENPX,HIGHPX,LOWPX\n");
        fflush(SH_Quote_File);
        fflush(SH_Trade_File);
        fflush(SH_Index_File);
    }
}
```

4. LoadTemplate()

loadTemplate的输入为l_template_path, 该变量在读取config的时候获得,函数的返回值 是一个string型

同时 该函数的返回值 需要作为参数传入 后续的 DataHandOut()中.

template.2.20.xml

template.xml 的形式如下

```
[sysop@localhost template]$ cat template.2.20.xml
<?xml version="1.0" encoding="UTF-8"?>
<templates version="2.20" updateDate="2019-04-16" xmlns="http://www.fixprotocol.org/ns/template-definition" templateNs="http://www.fixprotocol.org/ns/templates/sample" ns="http://www.fixprotocol.org/ns/fix">
  <template name="MarketOverview" id="1115">
    <string name="MessageType" id="35"><constant value="UA1115"/></string>
    <int32 name="DataTimestamp" id="10178"><copy/></int32>
    <int32 name="DataStatus" id="10121" presence="optional"><default/></int32>
    <string name="SecurityID" id="48" presence="optional"/>
    <int32 name="AShareIndex" id="10001" presence="optional" decimalPlaces="3"><default/></int32>
    <int32 name="BShareIndex" id="10002" presence="optional" decimalPlaces="3"><default/></int32>
    <int32 name="SSEIndex" id="10005" presence="optional" decimalPlaces="3"><default/></int32>
    <int32 name="OrigTime" id="42" presence="optional"><default/></int32>
    <int32 name="OrigDate" id="10003" presence="optional"><default/></int32>
    <string name="EndOfDayMarker" id="10004" presence="optional"><default/></string>
  </template>
  <template name="AHMarketData" id="2102">
    <string name="MessageType" id="35"><constant value="UA2102"/></string>
    <int32 name="DataTimestamp" id="10178"><copy/></int32>
    <int32 name="DataStatus" id="10121" presence="optional"><default/></int32>
    <string name="SecurityID" id="48"/>
    <string name="Symbol" id="55" charset="unicode"/>
    <int32 name="PreClosePx" id="140" decimalPlaces="3"/>
    <int32 name="HighPx" id="332" decimalPlaces="3"/>
    <int32 name="LowPx" id="333" decimalPlaces="3"/>
    <int32 name="LastPx" id="31" decimalPlaces="3"/>
    <int32 name="NominalPx" id="10078" decimalPlaces="3"/>
    <int32 name="BidPx" id="132" decimalPlaces="3"/>
    <int32 name="OfferPx" id="133" decimalPlaces="3"/>
    <int64 name="TotalVolumeTrade" id="807"/>
    <int64 name="TotalValueTrade" id="8504"/>
  </template>
  <template name="AHVirtualAuctionPrice" id="2107">
    <string name="MessageType" id="35"><constant value="UA2107"/></string>
    <int32 name="DataTimestamp" id="10178"><copy/></int32>
    <int32 name="DataStatus" id="10121" presence="optional"/>
    <string name="SecurityID" id="48"/>
    <string name="Symbol" id="55" charset="unicode"/>
    <int32 name="ClosePrice" id="140" decimalPlaces="3"/>
    <int32 name="Price" id="44" decimalPlaces="3"/>
    <int64 name="VirtualAuctionQty" id="10127"/>
  </template>
  <template name="NGSTSTransaction" id="3201">
    <string name="MessageType" id="35"><constant value="UA3201"/></string>
    <int32 name="DataStatus" id="10121" presence="optional"><default/></int32>
    <int32 name="TradeIndex" id="10011"><increment/></int32>
    <int32 name="TradeChannel" id="10115"><copy/></int32>
    <string name="SecurityID" id="48" presence="optional"><copy/></string>
    <int32 name="TradeIndex" id="10012" presence="optional"><copy/></int32>
    <int32 name="TradePrice" id="10014" presence="optional" decimalPlaces="3"><default/></int32>
    <int64 name="TradeQty" id="10015" presence="optional" decimalPlaces="3"><default/></int64>
    <int64 name="TradeMoney" id="10016" presence="optional" decimalPlaces="5"><default/></int64>
    <int64 name="TradeBuyNo" id="10179" presence="optional"><default/></int64>
  </template>
</templates>
```

读取成功时

```
if (template_file)
{
    template_file.seekg(0, std::ios::end);
    size_t len = (size_t)template_file.tellg();
    if (len > 0)
    {
        char* tmp = new char[len];
        template_file.seekg(0, std::ios::beg);
        template_file.read(tmp, len);
        fast_template.assign(tmp, len);
        delete[]tmp;
    }
    else
    {
        std::cout << "template length error" << std::endl;
        template_file.close();
        return 0;
    }
    template_file.close();
}
```

读取成功时，会执行以下步骤：

1. 将输入流指针定位到末尾，即略过流内的所有数据
2. 获取收入流的长度 len
3. 如果输入流长度大于0，执行后续步骤，否则退出
4. 定义一个char* 指针 指向len长的char[]
5. 将流指针重定向到文件起始位置

6. 将文件内容传入fast_template中

SetLogonRequest()

该函数的主要功能为封装登录用的报文，返回值类型为 string
函数的输入是 config过程中获得的信息。

```
std::string SetLogonRequest(std::string p_ProtocolEdition, std::string p_SenderCompID,
    std::string p_TragetCompID, int p_EbcryptMethod, int p_HeartBtInt, int checkcode)
{
    cout << "logon" << endl;

    STEP::UserLogonRequest LogonRequest;

    //BeginInitString, tag = 8;
    LogonRequest.getHeader().set(FIX::BeginInitString(p_ProtocolEdition));
    //BodyLength, tag = 9;
    //MsgType, tag = 35;
    //SenderCompID, tag = 49;
    LogonRequest.getHeader().set(FIX::SenderCompID(p_SenderCompID)); //提供
    //TargetCompID, tag = 56;
    LogonRequest.getHeader().set(FIX::TargetCompID(p_TragetCompID));
    //MsgSeqNum, tag = 34;
    LogonRequest.getHeader().set(FIX::MsgSeqNum(checkcode));
    //SendingTime, tag = 52;
    FIX::UtcTimeStamp TimeNow;
    gettimeofday(&tv, &tz);
    t = localtime(&tv.tv_sec);
    TimeNow.setYMD(1900 + t->tm_year, 1 + t->tm_mon, t->tm_mday);
    TimeNow.setHMS(t->tm_hour, t->tm_min, t->tm_sec, tv.tv_usec);
    LogonRequest.getHeader().set(FIX::SendingTime(TimeNow));
    //EncryptMethod, tag = 98;
    LogonRequest.set(FIX::EncryptMethod(p_EbcryptMethod));
    //HeartBtInt, tag = 108;
    LogonRequest.set(FIX::HeartBtInt(p_HeartBtInt));
    //Checksum = 10;
    std::string LogonRequest_str;
    LogonRequest.toString(LogonRequest_str);

    return LogonRequest_str;
}
```

该函数主要功能如下:

1. 输出logon 信息告知登录
2. 调用Step 目录下的相关header 创建一个 UserLogonRequest 实例

UserLogonRequest

UserLogonRequest 由Message.h 中的Message类派生而来

在函数中，调用的是默认构造函数. 即:

```
UserLogonRequest() : Message(MsgType()) {}
```

Message.h 中的 Message 类型是由 quickfix/Message.h下的 Fix::Message 派生而来

在本例中调用的是以下 构造函数:

```
class Message : public FIX::Message
{
public:
    Message( const FIX::MsgType& msgtype )
    : FIX::Message(
        FIX::BeginString("STEP.1.0.0"), msgtype )
    {}

    .
    .
    .

}
```

由此，我们再进入 Fix::Message 中去看源码:

```
class Message : public FieldMap
{
    friend class DataDictionary;
    friend class Session;

    enum field_type { header, body, trailer };
    ....
public:
    ....
protected:
    // Constructor for derived classes
    Message( const BeginString& beginString, const MsgType& msgType );
    ...
}
```

调用相关函数

程序调用了 LogonRequest的内置函数，最终拼接了config信息.

```
//BeginString, tag = 8;
LogonRequest.getHeader().set(FIX::BeginString(p_ProtocolEdition));
```

```

//BodyLength, tag = 9;
//MsgType, tag = 35;
//SenderCompID, tag = 49;
LogonRequest.getHeader().set(FIX::SenderCompID(p_SenderCompID)); //提供
//TargetCompID, tag = 56;
LogonRequest.getHeader().set(FIX::TargetCompID(p_TragetCompID));
//MsgSeqNum, tag = 34;
LogonRequest.getHeader().set(FIX::MsgSeqNum(checkcode));
//SendingTime, tag = 52;
FIX::UtcTimeStamp TimeNow;
gettimeofday(&tv, &tz);
t = localtime(&tv.tv_sec);
TimeNow.setYMD(1900 + t->tm_year, 1 + t->tm_mon, t->tm_mday);
TimeNow.setHMS(t->tm_hour, t->tm_min, t->tm_sec, tv.tv_usec);
LogonRequest.getHeader().set(FIX::SendingTime(TimeNow));
//EncryptMethod, tag = 98;
LogonRequest.set(FIX::EncryptMethod(p_EbcryptMethod));
//HeartBtInt, tag = 108;
LogonRequest.set(FIX::HeartBtInt(p_HeartBtInt));
//Checksum = 10;
std::string LogonRequest_str;
LogonRequest.toString(LogonRequest_str);

```

为了方便理解，我添加了相关输出程序来理解 UserLogonRequest 的具体功能。

```

logon
8=STEP.1.0.09=7335=A34=049=cedarsender52=20200709-15:10:3056=cedartarget98=0108=1010=223
socket established

```

如程序所注释的，该对象调用了其重载的多个set函数，通过传入不同的参数，拼接好了最终的string

SocketLogon()

该函数功能主要用于建立 socket 连接

```

//socket连接
int SocketLogon(std::string request) {
    struct sockaddr_in servaddr;
    //socket init
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        printf("create socket error: %s(errno: %d)\n", strerror(errno), errno);
        return 0;
    }
    else {
        printf("socket established\n");
    }

    //socket设置
    int nRecvBuf = socket_buffer_length;
    int nSendBuf = socket_buffer_length;
    setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, (const char*)&nRecvBuf, sizeof(int));
    setsockopt(sockfd, SOL_SOCKET, SO_SNDBUF, (const char*)&nSendBuf, sizeof(int));
    // connection init
}

```

```

memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(port);
servaddr.sin_addr.s_addr = inet_addr(ip);
int connfd = connect(sockfd, (struct sockaddr*) & servaddr, sizeof(servaddr));
if (connfd < 0) {
    printf("connect error: %s(errno: %d)\n", strerror(errno), errno);
    return 0;
}
else {
    printf("connection established\n");
}
int logon_flag = send(sockfd, request.c_str(), strlen(request.c_str()), 0);
printf("send success\n");
if (logon_flag < 0)
{
    printf("logon error: %s(errno: %d)\n", strerror(errno), errno);
    return 0;
}
else
{
    printf("logon success\n");
    return sockfd;
}
}

```

函数实现了以下功能:

1. 初始化socket
2. 设置socket

DataHandOut()

DataHandOut是 整个程序最重要的部分。

其传入参数为int型变量 l_sockfd和string型的l_fast_template

```

void DataHandOut(int l_sockfd, std::string l_template_path) {
    ....
}

```

创建进程

在函数一开始建立了三个 指向函数地址的指针.

```

void* (*psaveQuoteFunc) (void* arg);
void* (*psaveTradeFunc) (void* arg);

```

```
void* (*psaveIndexFunc) (void* arg);
```

具体参考 线程创建函数

```
pthread_create(pthread_t *id, pthread_attr_t *attr, void* (*func)(void*), void *arg)
```

之后分别将对应的函数地址赋值给指针:

```
psaveQuoteFunc = saveQuoteIntoFiles;  
psaveTradeFunc = saveTradeIntoFiles;  
psaveIndexFunc = saveIndexIntoFiles;
```

后续按照例行, 先创建进程, 再回收进程

```
//创建 Quote 进程  
pthread_t saveQuoteThread;  
if ((pthread_create(&saveQuoteThread, NULL, psaveQuoteFunc, NULL)) !=0)  
{  
    printf("create error!\n");  
    return;  
}  
  
//创建 Trade 进程  
pthread_t saveTradeThread;  
if ((pthread_create(&saveTradeThread, NULL, psaveTradeFunc, NULL)) !=0)  
{  
    printf("create error!\n");  
    return;  
}  
  
//创建 Index 进程  
pthread_t saveIndexThread;  
if ((pthread_create(&saveIndexThread, NULL, psaveIndexFunc, NULL)) !=0)  
{  
    printf("create error!\n");  
    return;  
}  
  
.....  
.....  
  
//回收进程空间  
pthread_join(saveQuoteThread, NULL);  
pthread_join(saveTradeThread, NULL);  
pthread_join(saveIndexThread, NULL);
```

监听

创建一个MessageSaveHandler 对象，负责处理Message 并存储。

调用MessageSaveHandler的onMessageData 函数，传入 socket套接字，以及string 型 l_fast_template

```
//创建一个 MessageSaveHandle 对象
MessageSaveHandle l_MessageSaveHandler;

//调用对象的 onMessageData函数监听
l_MessageSaveHandler.onMessageData(l_sockfd, l_fast_template);
```

MessageSaveHandle类

MessageSaveHandle类由MarketManager派生得到，内部只有一个成员函数onMessageData(int l_sockfd, std::string l_fast_template)

负责拆解报文并写入文件

```
public:

    void onMessageData(int l_sockfd, std::string l_fast_template) {
        ....
    }
```

定义了相关内置变量和自定义变量

```
dynamic_templates_description description(l_fast_template);
const templates_description* descriptions[] = { &description };
fast_decoder decoder;
decoder.include(descriptions);

unsigned int date_today = GetTodayDate();
int length;
int header_length;
int header_length_main;
int header_length_remain;
int header_length_temp;
int body_length_main;
int body_length_remain;
int body_length_temp;
int count = 0;

std::map<std::string, std::string> l_StepMsgMap_header;
std::map<std::string, std::string> l_StepMsgMap_body;
```

循环体

在接下来的程序中跑一个无限循环的程序：

1. 每次循环 计数器count++
2. 定义了一个MsgHeader型数组
3. 初始化header_length_main
4. 当header_length_main的值小于21时，使用21减去main，如果得到的remain长度大于0，则会调用socket的recv函数，接受header_length_remain长度进缓冲

待读

关闭套接字

在程序结束时，还需要调用Linux内核中的 close socket函数

```
close(l_sockfd);
```