ID	Item Name	Status	Memo
1	Architect		
	EventProcess		
	Index		
2	SrvStartStop		Start Stop Server
3	RTSP		
	RTSP_ALL		
	RTSP_Client		
	RTSP_Run(Server)		
4	RTP		
	RTP_ALL		
	RTP_Run		
5	RTCP		
6	Thread		
	Thread Schedule		
	Thread Lock		
	Thread Sync		
	Thread Sync Obj		
7	Dictionary		
8	Module		
9	RTP/RTCP/RTSP		
a	Client Session		
	ClientSession::Run		
b	Atomic		
	Ext		

Why is reentrant not enabled?

TaskThread: 0x00b97a68
IdleTaskThread: 0x00b98d48
EventThread: 0x00b68640

需要弄清楚的地方

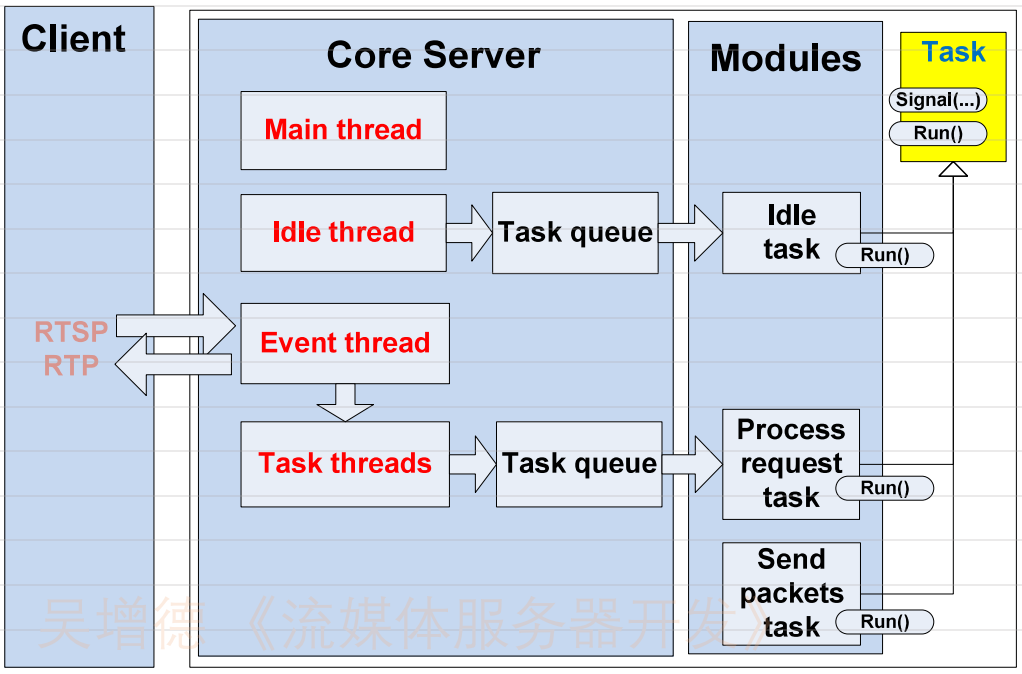
- Socket如何和事件联系到一起
- 事件如何处理
- 还没分析完的类分析完
- Server Startup→ 整条路走RTSP→RTP→RTCP
- 总共有多少线程,线程之间关系如何，_Entry(…)到底由谁来启动

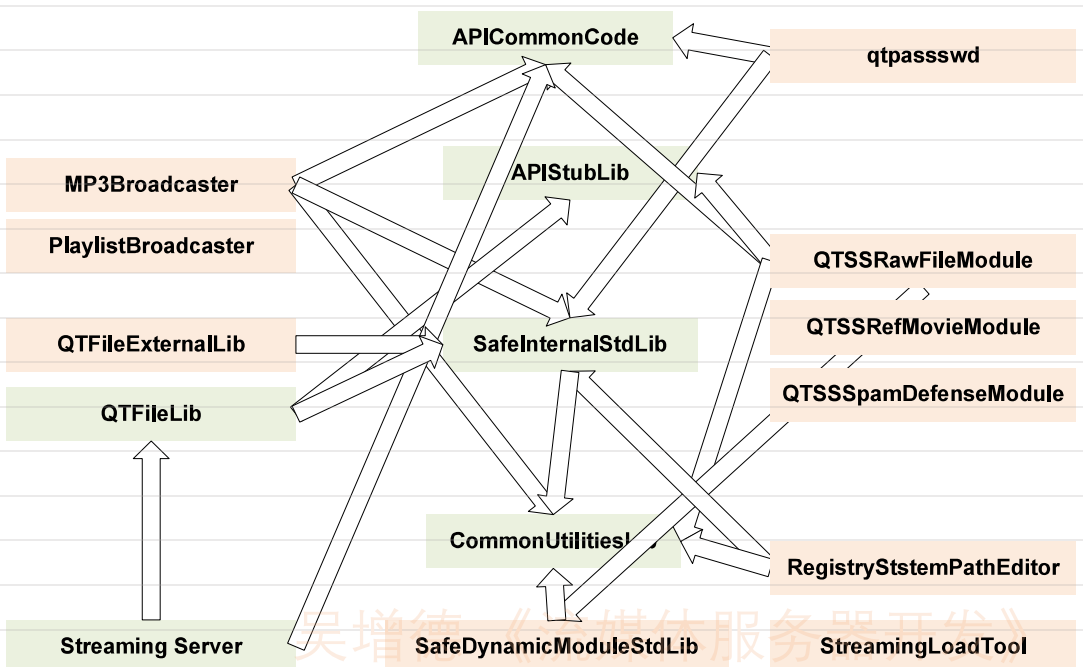
要理解Darwin体系结构, 只要理请楚四个地方:

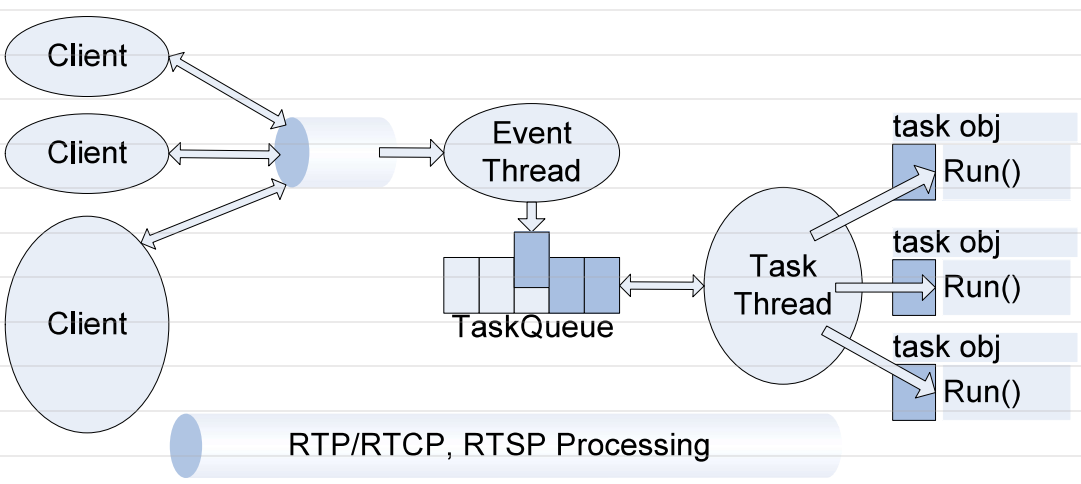
- 线程如何同步
- 任务如何调度
- 事件如何通知
- RTSP, RTP以及RTCP过程，在些基础之上, 进一步理清

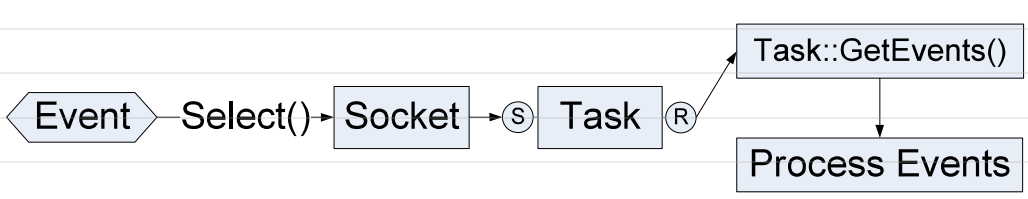
1.处理时间
2. 临界区
3. 信号量
4. 事件
5. 互斥量
6. 多线程

Architecture

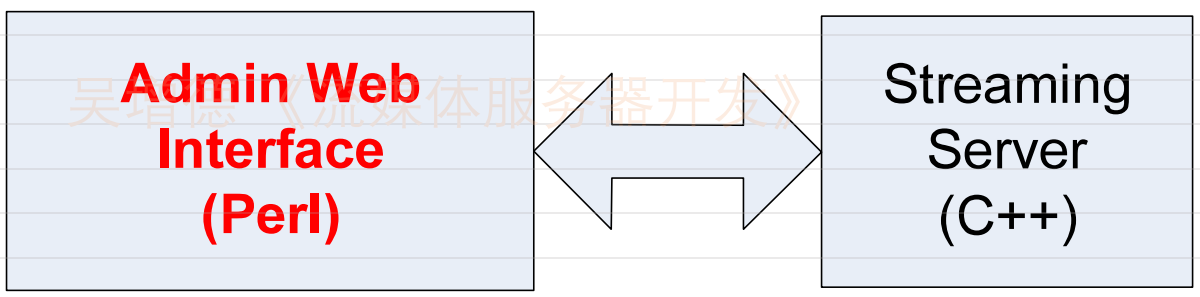








How to run & step into:
Darwin Streaming Server
source Code



streamingadminserver.conf => %config => %mime

test socket 554

startplaylist.pl
playlist-lib.pl

Open main socket

select ...
accept(SOCK, MAIN)
handle_request($acptaddr)

handle_request

$readline()
$method, $request_uri
socket => $header

parse http header => %header

Set defaults language

parse query string => %in

handle page url => $page

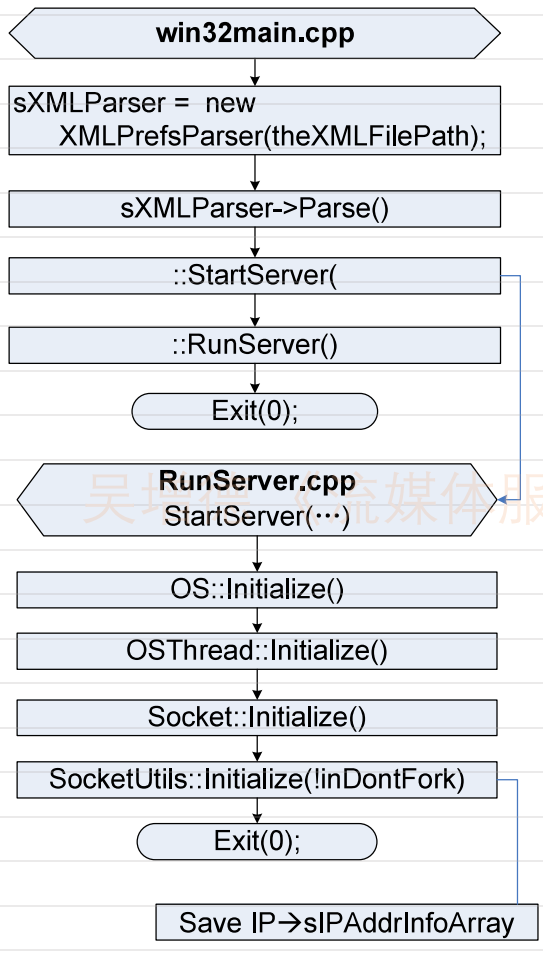
check ...
address against access list
check for the logout flag file
Check for password if needed

get $idx_full

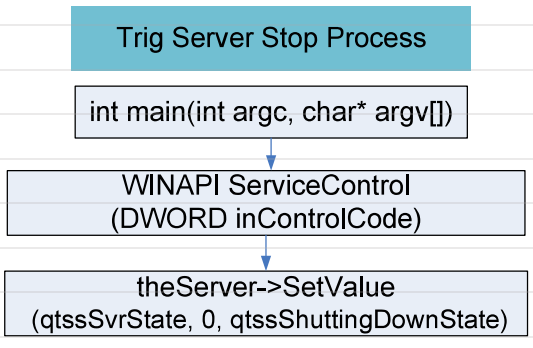
handle page url
=> $page

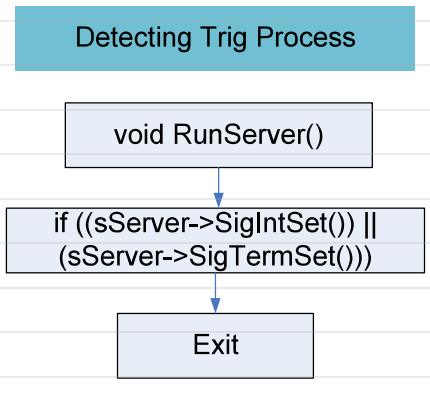
handle page url
=> $page

Server Startup Process

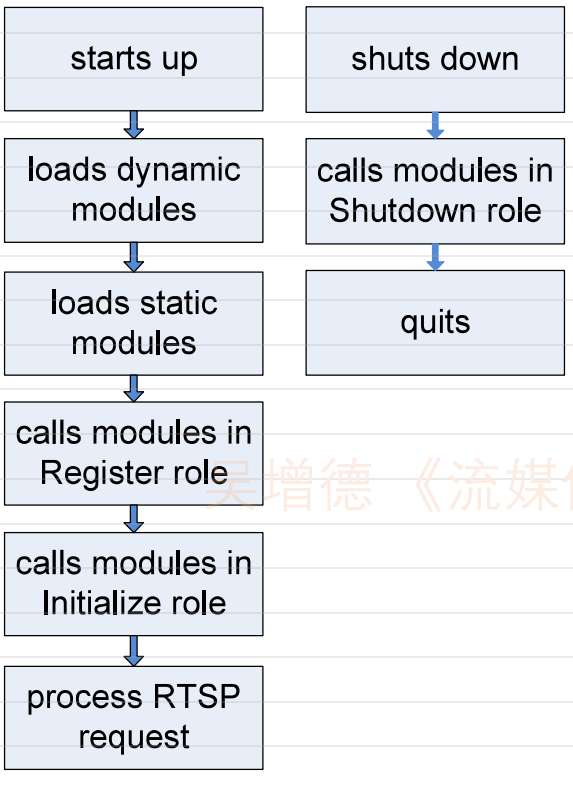


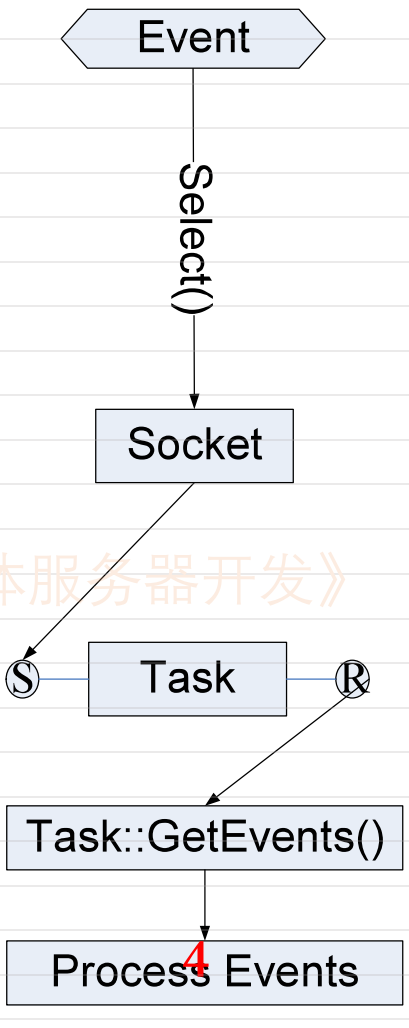
Start & Stop Server



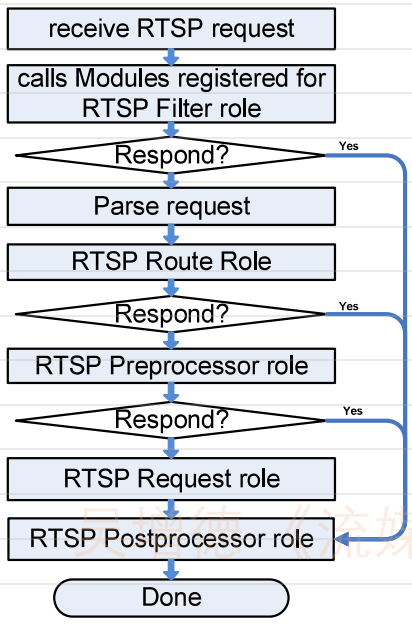


Server Startup & Shutdown

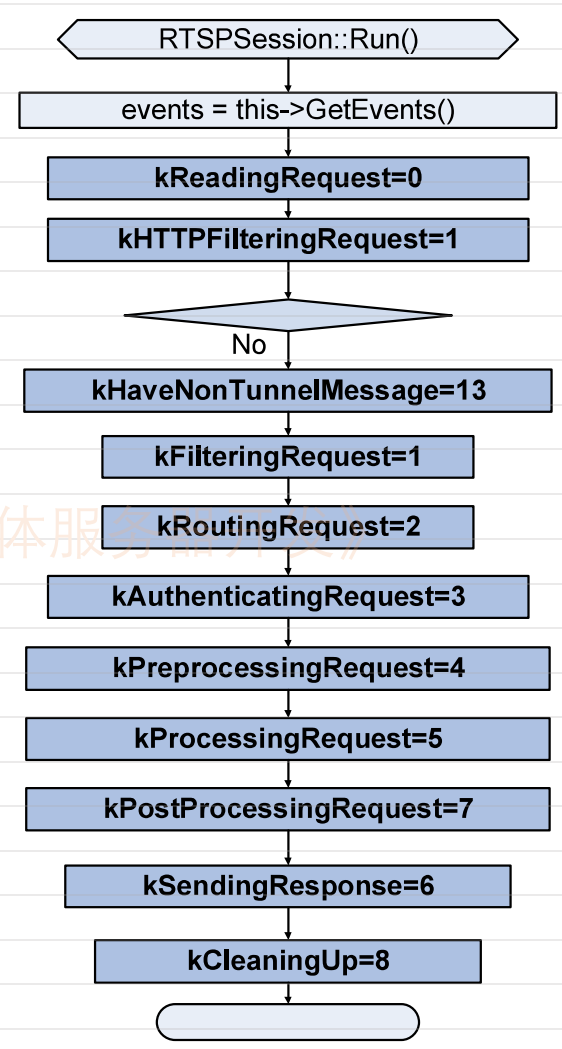




RTSP Request Processing



RTSP Process



1. Recv an RTSP request

1.创建一个RTSP请求对象，该对象是一些描述请求的属性的集合。
2.此时 qtssRTSPReqFullRequest
属性是唯一有值的属性，其值组成了RTSP请求的完整内容。

2. RTSP Filter role

1. 服务器将RTSP请求对象作为参数传递给该角色。每个RTSP Filter角色都可以改变qtssRTSPReqFullRequest属性的值。

3. Parses the request

- 对RTSP请求进行处理,包括
- 1.填充RTSP对象剩余的属性值
- 2.创建下面两个会话
  - 一个RTSP会话，当前请求相关联，客户端关闭与服务器之间的RTSP连接时，该会话随之关闭
  - 一个客户会话，和当前请求相关联，这个会话会一直保持，直到客户端的流播放结束。

4. RTSP Route role

- 1.传入一个RTSP对象。
- 2.角色使用RTSP对象中的属性值来确定是否要改变 qtssRTSPReqRootDir 属性的值，进而改变用于处理当前请求的目录。

5. RTSP Preprocessor role

- 1.角色通过qtssRTSPReqAbsoluteURL属性值判断当前请求是否和模块处理的请求类型相匹配
- 2.匹配,角色就调用QTSS_Write 或者 QTSS_WriteV 函数来向客户发送数据，对客户端进行响应
- 3.如果只需要发送标准响应，则模块可以调用 QTSS_SendStandardRTSPResponse， 或者 QTSS_AppendRTSPHeader 和 QTSS_SendRTSPHeaders 函数。

6. RTSP Request role

- RTSP Request 角色负责响应所有没有被RTSP Preprocessor角色（的模块）处理过的RTSP请求。
- 只有一个模块可注册

7. RTSP Postprocessor role

- RTSP Postprocessor 角色通常执行一些统计任务，比如记录各种统计信息。

Player

InitStreamingClient()

CreateStream(strFileName, 0, StreamingClientCallback)

CloseStreamingClient();

StreamingClientDLL.cpp

```
(void)::WSAStartup(wsVersion, &wsData);
OS::Initialize();
OSThread::Initialize();
Socket::Initialize();
SocketUtils::Initialize();
```

```
StreamingClientCallback(nDVRID,
                        char * inPacket, inLength)
Receive and play video
```

```
CreateStream(..)  //rtsp://159.99.242.150/dahua.live_15_4
{
   strServer = “159.99.242.”
   fClientSocket = NEW TCPClientSocket(...);
   fClient = NEW RTSPClient(fClientSocket, …, … );
   fClientSocket 的地址, 端口设置
   fClient 的URL设置
   //
** fClient->SendDescribe();
   if(fClient->GetStatus() == 200)
    {
      fStreamArray = NEW SourceInfo::StreamInfo[fNumStreams];
      设置fStreamArray;
      设置fDurationTimeInSecs;
      设置fLocalSDP;
    }
** //执行SendTCPSetup
   对每个流执行
       fClientSocket->SetRcvSockBufSize(2048); //实际调用fSocket..
       fClient->SendTCPSetup(…)
   fClient->SendTCPSetup(…) ;
** //执行SendPlay
   fClient->SendPlay(nStartTimeInSec);
   //申请并设置pRTPStreamEntry数据结构
   pRTPStreamEntry = NEW RTPStreamList();
      pRTPStreamEntry->rtspClient = fClient;
   gRTPStreamList = pRTPStreamEntry;
   //创建线程,以便于读流数据,
      hRTPReceiverThread = CreateThread (
               NULL, 0, ReadMediaData,
               pRTPStreamEntry, 0, NULL);
}
```

```
DWORD WINAPI ReadMediaData(LPVOID Input){
  while (pRTPStreamEntry->rtpThreadState == RTPStreamRun)  {
     pRTPStreamEntry->rtspClient->GetMediaPacket(&theTrackID, &isRTCP, &thePacket, &theLength);
     pRTPStreamEntry->callbackFunction(nDVRID, thePacket+14, theLength-14);
  }
}
```

RTSPClient.cpp

```
SendDescribe(Bool16 inAppendJunkData)
```

ClientSocket* fSocket

```
RTSPClient::DoTransaction()
{
    fSocket->Send(theTransaction.Ptr,
          theTransaction.Len);
}
```

ClientSocket.cpp

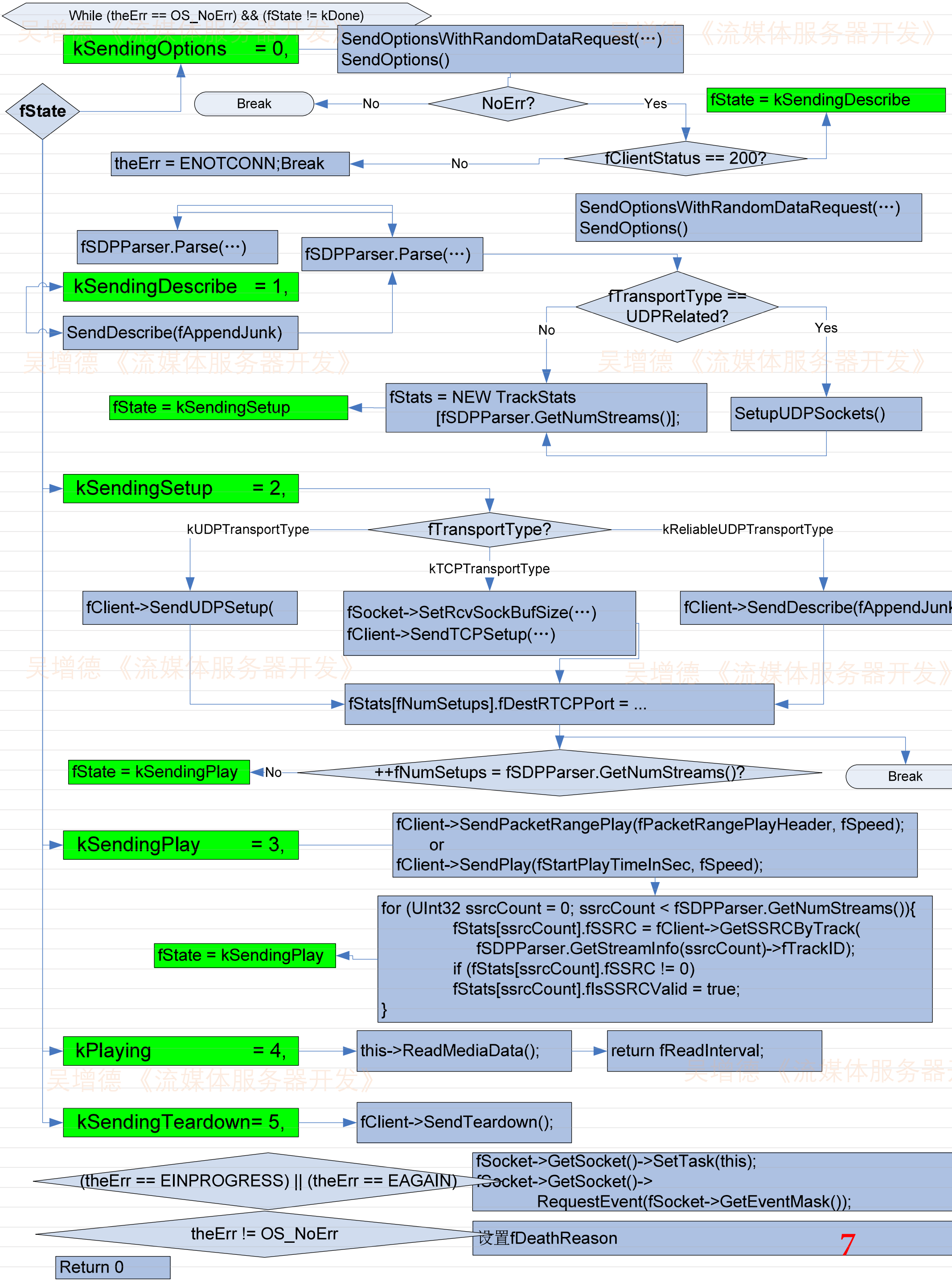
```
OS_Error ClientSocket::Send(
        char* inData, UInt32 inLength)
{
   iovec theVec[1];
   theVec[0].iov_base = (char*)inData;
   theVec[0].iov_len = inLength;

   return this->SendV(theVec, 1);
}
```

```
TCPClientSocket::SendV(iovec* inVec,
UInt32 inNumVecs)
{
   **Copy the buffer;
   this->Connect(&fSocket);
   this->SendSendBuffer(&fSocket);
}
```

```
ClientSocket::SendSendBuffer(TCPSocket* inSocket)
{
 do{
   theLengthSent = 0;
   theErr = inSocket->Send(fSendBuffer.Ptr + fSentLength,
         fSendBuffer.Len - fSentLength,  &theLengthSent);
   fSentLength += theLengthSent;
 } while (theLengthSent > 0);
}
```

Socket.cpp

```
Send(const char* inData,
        const UInt32 inLength,
        UInt32* outLengthSent)
{
    ::send(fFileDesc, inData, inLength, 0);
}
```



RTSPSession::Run()

events = this->GetEvents()

负责从RTSPRequestStream类的对象fInputStream中读取客户的RTSP请求
fInputSocketP->RequestEvent(EV_RE)

kReadingRequest=0

检查RTSP连接是否需要经过HTTP代理实现 ???

kHTTPFilteringRequest=12

No

kHaveNonTunnelMessage=14

1. 创建RTSPRequest类的对象fRequest，
fRequest = NEW RTSPRequest(this);

2. 填入fRoleParams(fRP)数据结构
fRoleParams.rtspRequestParams.inRTSPRequest = fRequest;
fRoleParams.rtspRequestParams.inRTSPHeaders = fRequest->GetHeaderDictionary();
fOutputStream = 0

3. 重置输出流 fOutputStream.ResetBytesWritten();

系统创建RTSPRequest类的对象fRequest，该对象解析客户的RTSP请求，并保存各种属性

只有当fInputStream有完整的消息时才到这里

二次开发人员可以通过编写Module对客户的请求做出特殊处理。如果客户的请求为正常的RTSP请求，系统调用SetupRequest函数建立用于管理数据传输的RTPSession类对象
1.更新超时时间 fTimeoutTask.RefreshTimeout();
2.判断是否是数据包，如果是，则处理数据包，并进入kCleaningUp状态

kFilteringRequest=1

fTimeoutTask.RefreshTimeout()

IsDataPacket() Yes

HandleIncomingDataPacket()

定义并填充以下参数块
QTSS_RoleParams theFilterParams

....rtspFilterParams.inRTSPSession = this;
....rtspFilterParams.inRTSPRequest = Request;
....rtspFilterParams.outNewRequest = &theReplacedRequest;

触发所有注册的Filter模块，循环执行

theModule->CallDispatch(QTSS_RTSPFilter_Role, &theFilterParams);

如果模块请求了一个事件，返回并等待事件过期，
下次CPU时间直接回到该状态

解析并获得RTSP会话
SetupRequest()

SetupRequest()的实现

1.首先解析请求
fRequest->Parse()，从字符串读出相关内容，填入fRequest

2.尝试找到该请求的RTP会话
FindRTPSession(theMap)

4

如果没有RTP会话，建立
fRTPSession

6

fRoleParams.rtspRequestParams.inClientSession = fRTPSession

fRTPSession(theMap) Yes

No

fRTPSession->RefreshTimeout()

fRequest->GetMethod()

1 如果是DESCRIBE请求，确保有SessionID

2 如果是SET_PARAMETER 让模块来处理

3 如果OPTIONS请求 发送一个标准的OPTION回应

5 如果Play请求，按context的不同
1. 调用SetStreamThinningParams设置参数
2. 发送回应

调用二次开发人员加入的Module，用于将该请求路由（Routing）出去。缺省情况下，系统本身对此状态不做处理。???
用于改变系统当前请求的目录

kRoutingRequest=2

触发所有注册的router模块，循环执行

改变处理请求目录
fRequest->SetupAuthLocalPath()

theModule->CallDispatch(QTSS_RTSPFilter_Role, &theFilterParams);

如果模块请求了一个事件，返回并等待事件过期，
下次CPU时间直接回到该状态

调用二次开发人员加入的安全模块，主要用于客户身份验证以及其他如规则的处理。读者如果希望开发具有商业用途的流式媒体服务器，该模块必须进行二次开发

kAuthenticatingRequest=3

1. 读取RTSP方法
method = fRequest->GetMethod()
2. 处理RTSP方法

如果请求没有设定安全模式，则采用xml的的设置
fRequest->GetAuthScheme() == qtssAuthNone

设置authentication参数块，theAuthenticationParams

调用以安全角色调用模块:
theModule=QTSServerInterface::GetModule(QTSSModule::kRTSPAthnRole, 0);
theModule->CallDispatch(QTSS_RTSPAuthenticate_Role, &theAuthenticationParams);

CheckAuthentication()

该函数对用户信息进行了认证

通过调用系统自带或二次开发人员添加的Module来处理RTSP请求，例如系统提供了QTSSReflector Module、QTSSSplitter Module以及QTSSFile Module等模块。其中比较重要的QTSSFile Module属于QTLib库的部分

Do Describe

kPreprocessingRequest=5

theModule->CallDispatch(QTSS_RTSPPreProcessor_Role, &fRoleParams)

kProcessingRequest=6

theModule->CallDispatch(QTSS_RTSPRequest_Role, &fRoleParams)

kPostProcessingRequest=8

设置RTSP状态

theModule->CallDispatch(QTSS_RTSPPostProcessor_Role, &fRoleParams);

发送对客户RTSP请求处理完成之后的响应。系统在该状态调用了fOutputStream.Flush()函数将在fOutputStream中尚未发出的请求响应通过Socket端口完全发送出去

kSendingResponse=7

拷贝 fOutputStream → fOldOutputStreamBuffer

请空 fOutputStream

发送 OPTIONS

请空 fOutputStream

恢复 fOldOutputStreamBuffer

fOutputStream.Flush()

清除所有上次处理的数据，并将状态设置为kReadingRequest等待下次请求到达

kCleaningUp=9

调用请空 fOutputStream 释放资源

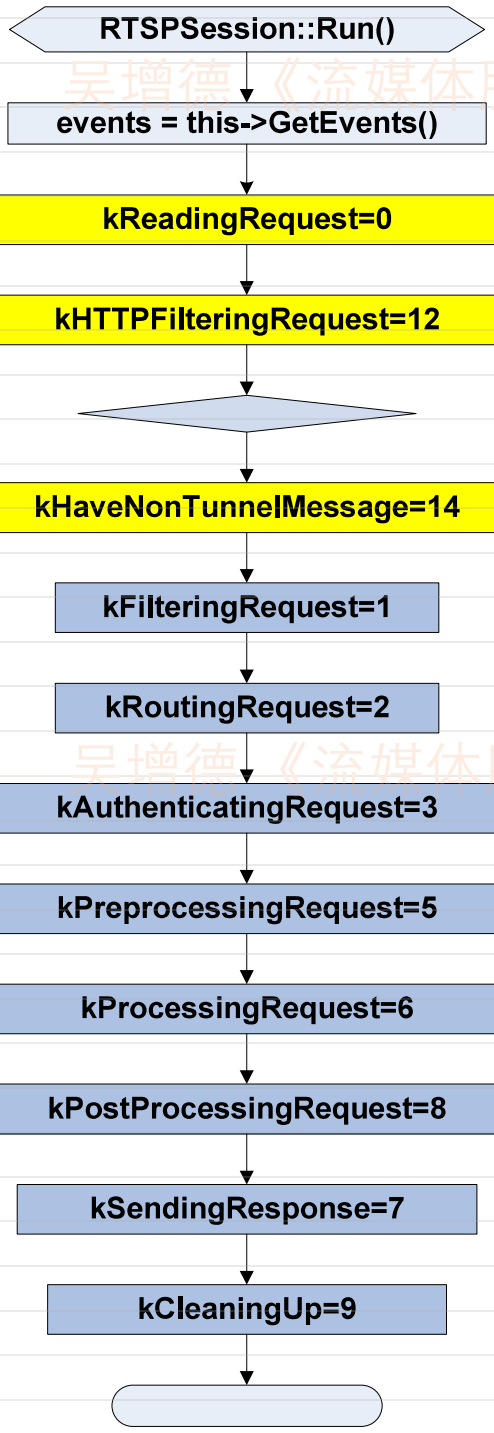
下一个状态回到 kReadingRequest

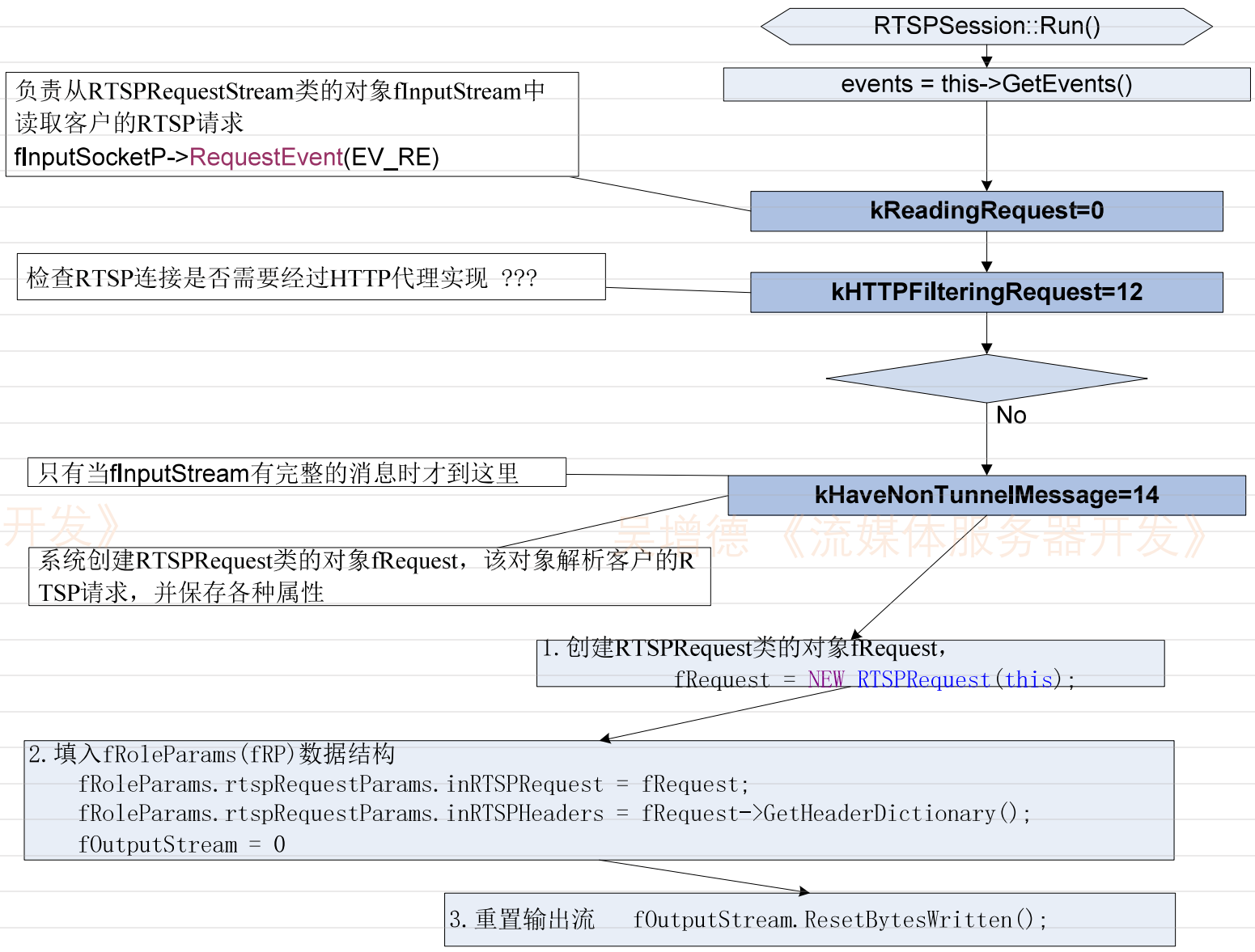
kAuthorizingRequest=4

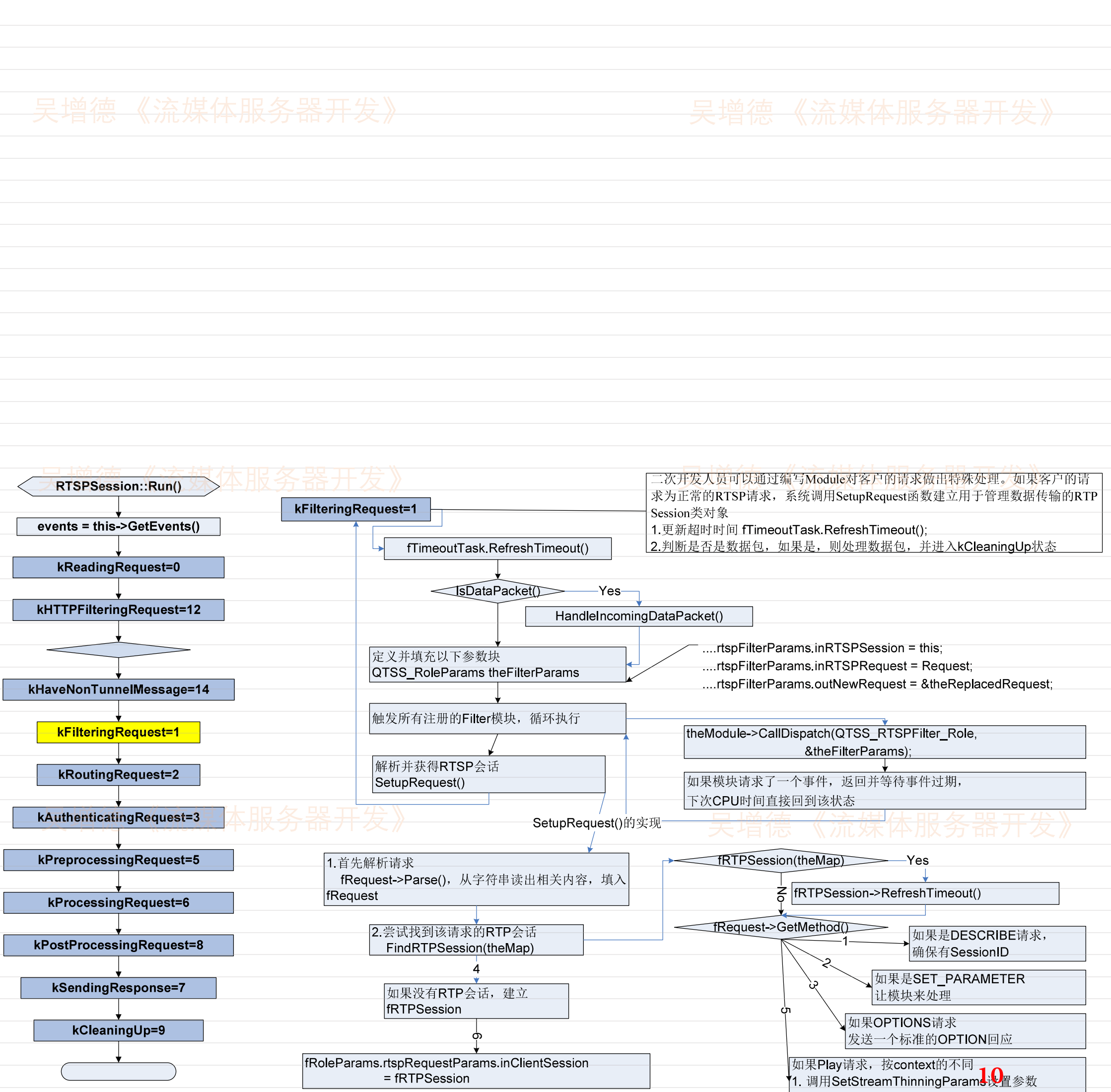
kWaitingToBindHTTPTunnel=10

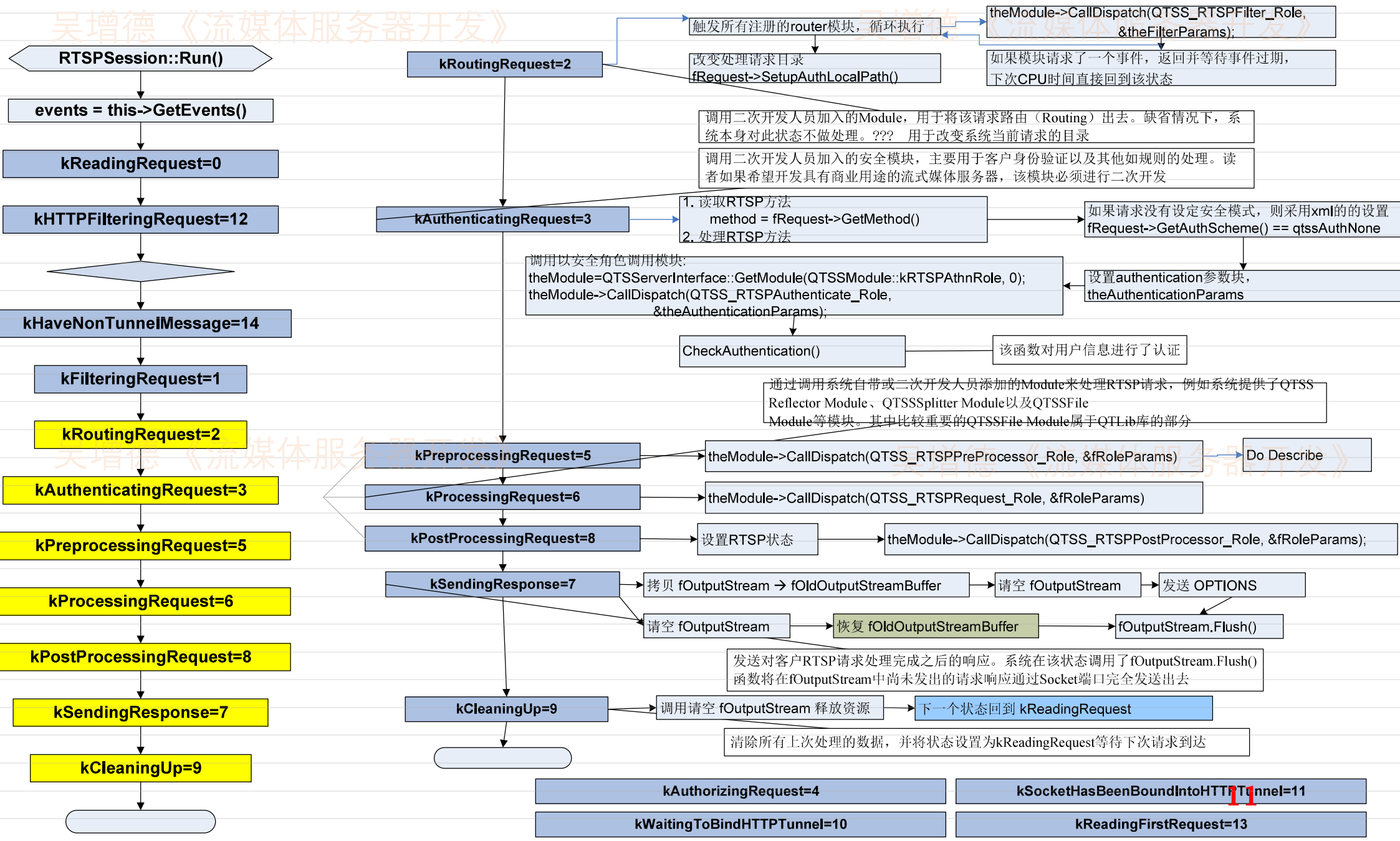
kSocketHasBeenBoundIntoHTTPTunnel=11

kReadingFirstRequest=13

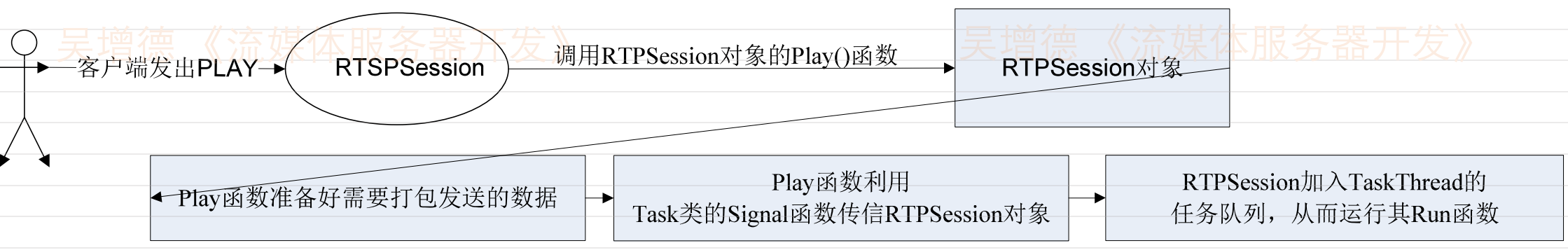


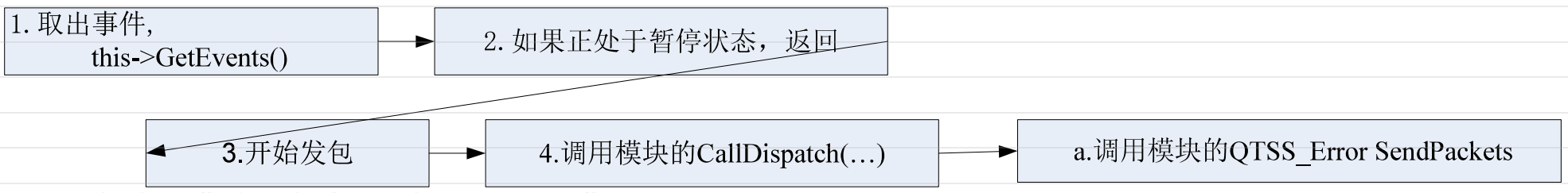


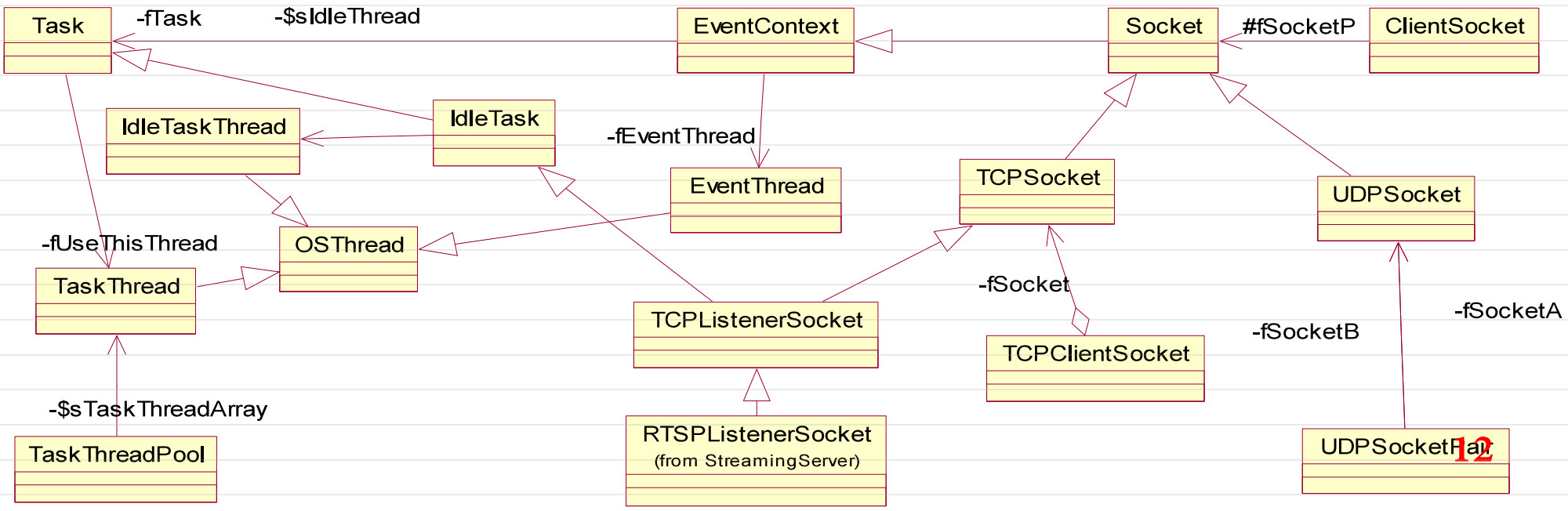




RTPSession::Play(RTSPRequestInterface* request, QTSS_PlayFlags inFlags)







RTPSession::Run()

events = this->GetEvents()

如果事件是通知RTPSession对象死亡，就准备自杀。可能导致这种情况的有两种事件：自杀kKillEvent；超时kTimeoutEvent

Yes

fModuleDoingAsyncStuff

(events & Task::kKillEvent) ||
(events & Task::kTimeoutEvent) ||
(fModuleDoingAsyncStuff)

(fState == qtssPausedState) ||
(fModule == NULL)

Y

Return 0;

是否已到发包时间

Yes

HandleIncomingDataPacket()

OSRefTable* sessionTable
QTSServerInterface::GetServer()
-> GetRTPSessionMap();

sessionTable->
TryUnRegister(&fRTPMapElem)

Yes

No

fModuleState.curTask = this;
fModuleDoingAsyncStuff = true;
fCurrentModule = 0;

this->Signal(Task::kKillEvent);
return kCantGetMutexIdleTime;
//10

Yes

对所有的客户端流会话发送RTCP SR
qtssCliSesStreamObjects
(*theStream)->SendRTCPSR(byePacketTime, true)

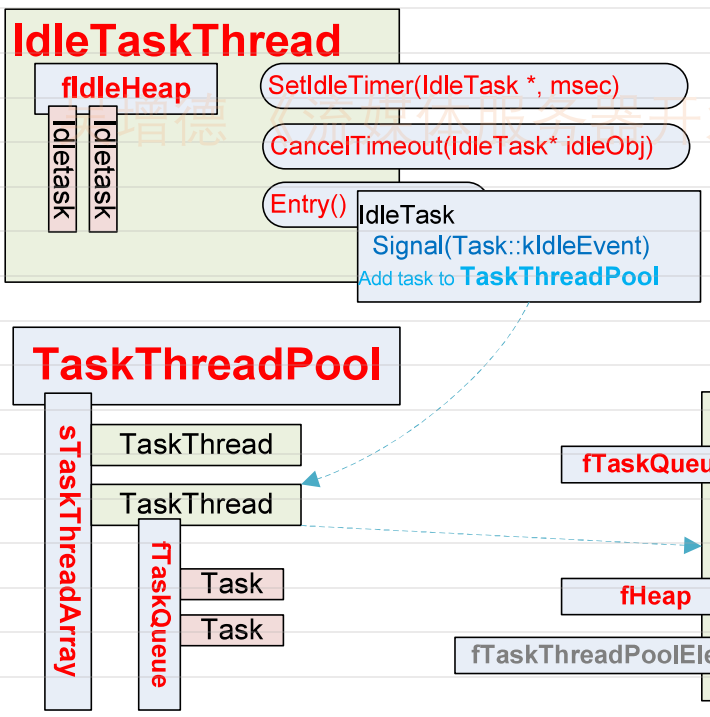
前面已做的工作:
QTSS_RoleParams theParams;
.clientSessionClosingParams.inClientSession = this;
.clientSessionClosingParams.inReason = fClosingReason;

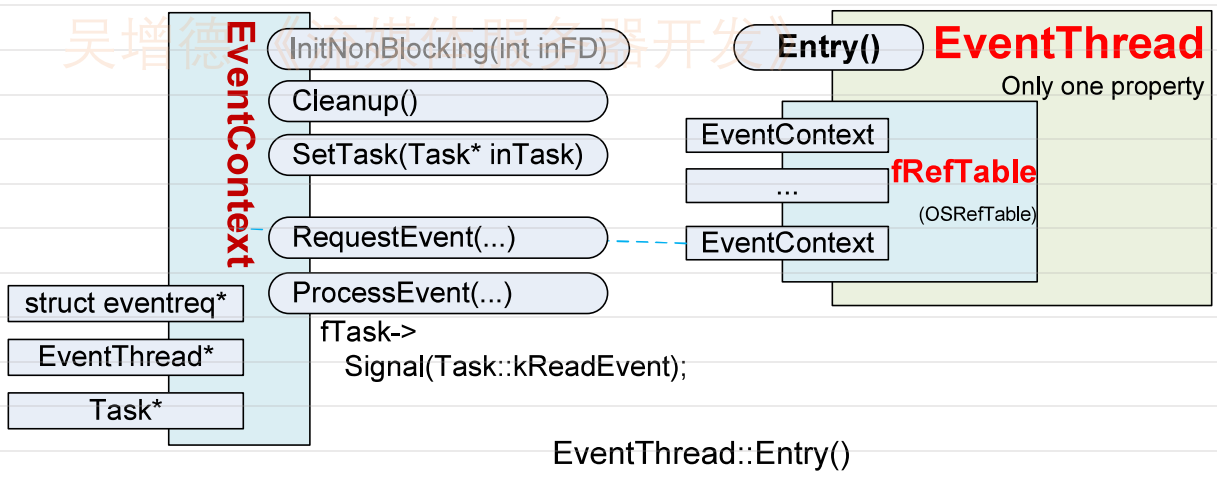
对所有注册kClientSessionClosingRole角色的模块
(void)theModule->
CallDispatch(QTSS_ClientSessionClosing_Role,
&theParams);

```
QTSSModule::CallDispatch(
        QTSS_Role inRole,
        QTSS_RoleParamPtr inParams){
    return (fDispatchFunc)(inRole, inParams);
}
```

```
void QTSSDVRModuleDispatch(QTSS_Role inRole, QTSS_RoleParamPtr
inParamBlock)
    switch (inRole)
    {
        case QTSS_Register_Role:
        case QTSS_ClientSessionClosing_Role:    ****
```

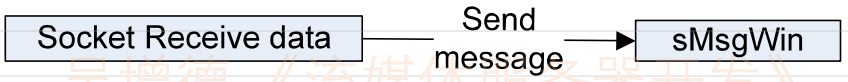
```
QTSS_Error  QTSS_GetValuePtr (QTSS_Object inDictionary, QTSS_AttributeID inID,
                              UInt32 inIndex, void** outBuffer, UInt32* outLen){
    return (sCallbacks->addr [kGetAttributePtrByIDCallback])
            (inDictionary, inID, inIndex, outBuffer, outLen);
}
```





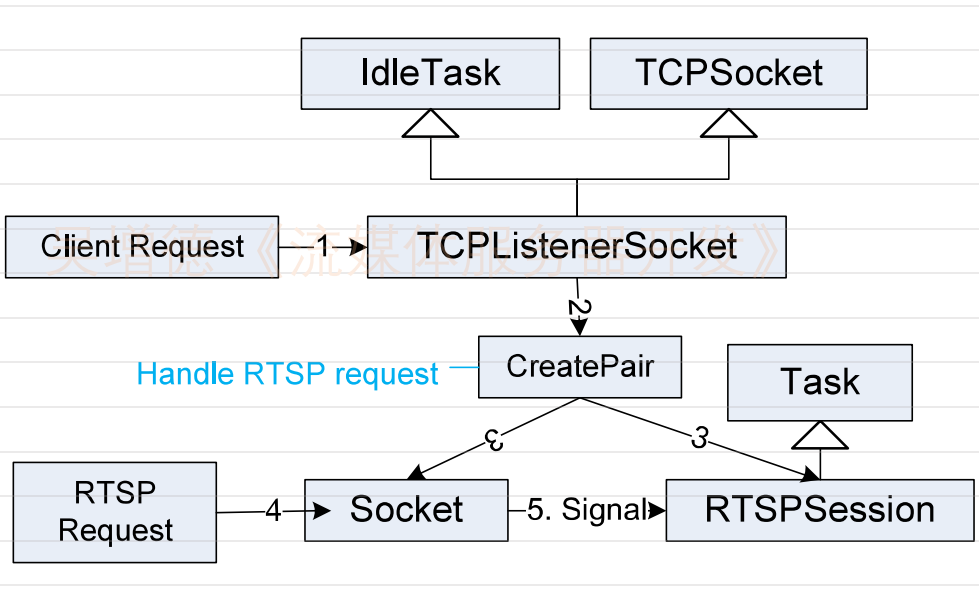

```
OSRef           fRef;
PointerSizedInt fUniqueID;  //ID
StrPtrLen       fUniqueIDStr; //ID
Bool16  fAutoCleanup; //Auto destroy
Task*   fTask;    //Task to be signaled
```

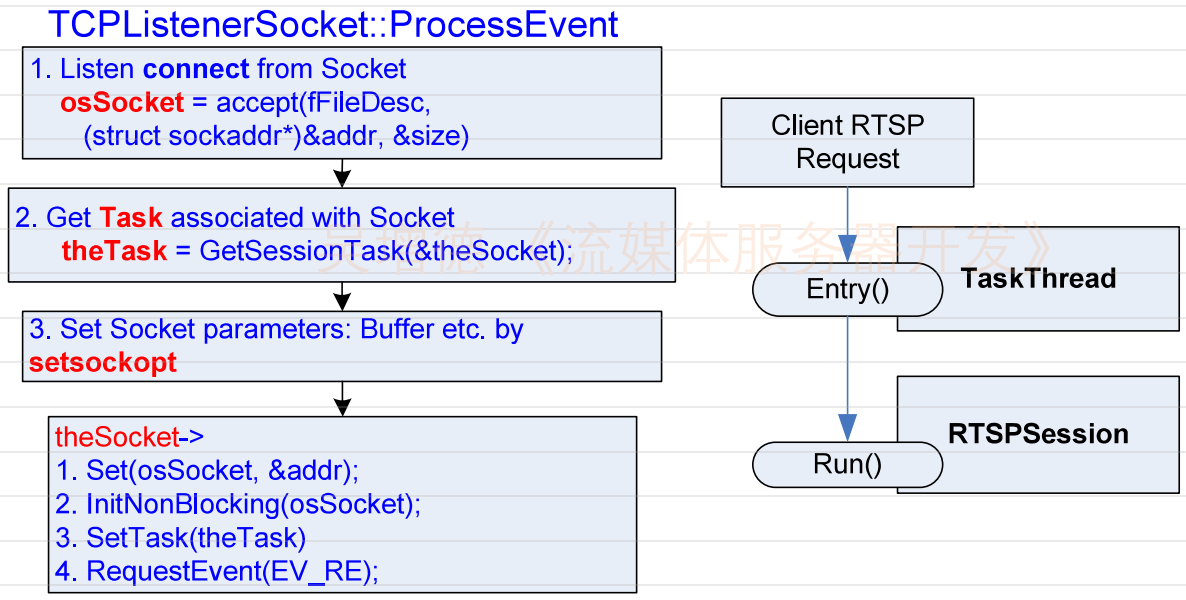
```
EventThread::Entry()
  select_waitevent(&theCurrentEvent, NULL);
  StrPtrLen idStr(theCurrentEvent);
  ref = fRefTable.Resolve(&idStr);
  theContext = (EventContext*)ref->GetObject();
  theContext->
    ProcessEvent(theCurrentEvent.er_eventbits);
```

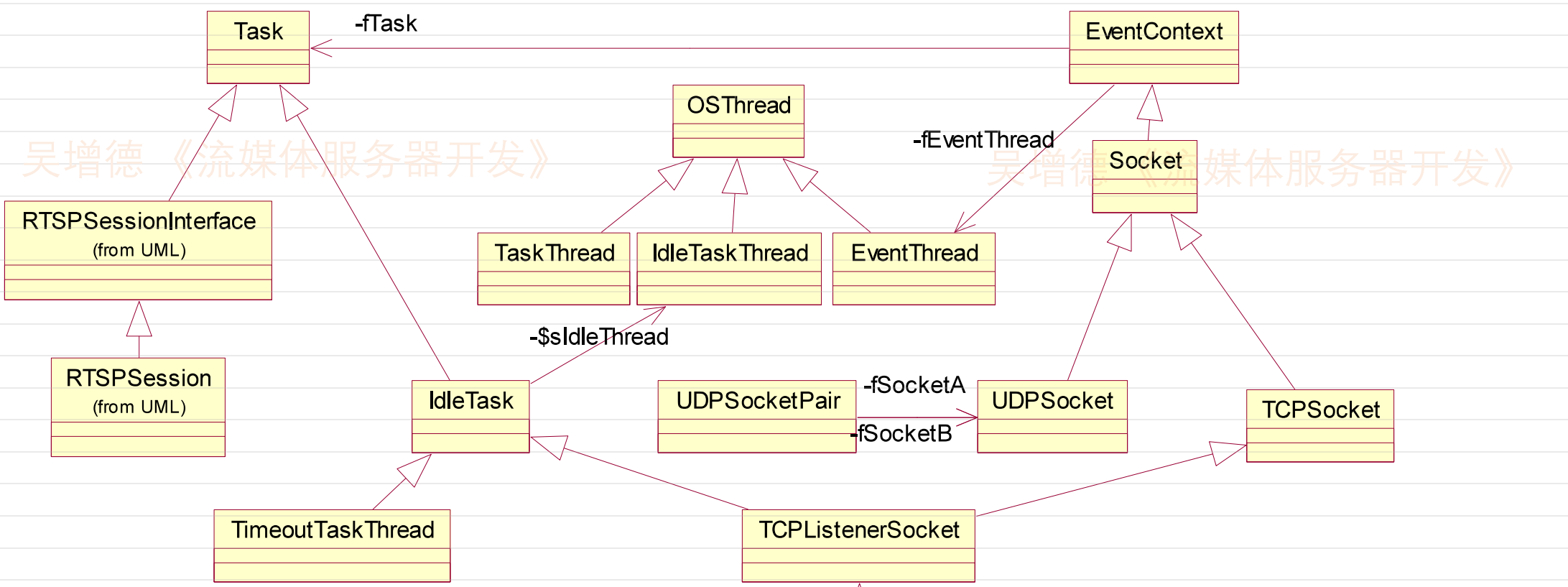



```
int select_waitevent(struct eventreq *req){
   if(!wnd) sMsgWin= ::CreateWindow(…);
   ::GetMessage(&theMessage, sMsgWindow, ...);
   req->er_handle = theMessage.wParam;  //Socket
   req->er_data = (void*)(theMessage.message);
  ::WSAAsyncSelect(req->er_handle, sMsgWin, 0, 0);
}
```

```
void EventContext::RequestEvent(int theMask) Win32ev.cpp
→int select_watchevent(struct eventreq *req, int which)
    fRef.Set(fUniqueIDStr, this);
    fEventThread->fRefTable.Register(&fRef);
    →int select_modwatch(struct eventreq *req, int which)
       unsigned int theMsg = (unsigned int)(req->er_data);
       Return ::WSAAsyncSelect(req->er_handle,
                    sMsgWindow, theMsg, theEvent);
```







```
int WSAAsyncSelect(
  _In_  SOCKET s,
  _In_  HWND hWnd,
  _In_  unsigned int wMsg,
  _In_  long lEvent
);
```

```
int select_waitevent(struct eventreq *req, void* /*onlyForMacOSX*/)

if (sMsgWindow == NULL)   sMsgWindow = ::CreateWindow
```

Thread maintains two task tables:

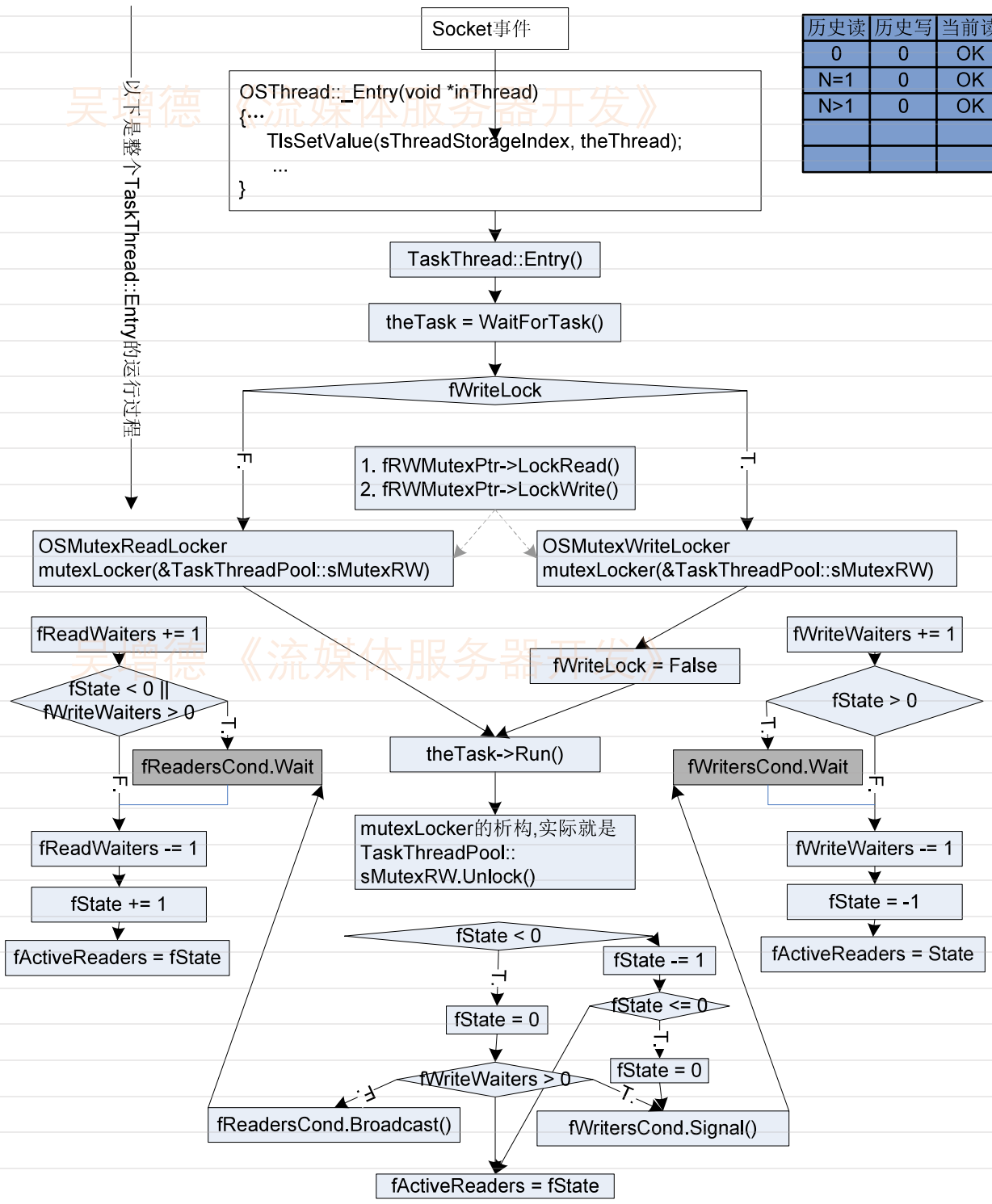
1. **TaskThread::fHeap: time critical**
2. **TaskThread::fTaskQueue: non-critical**

QTSSModule.cpp
  events & Task::kIdleEvent
    →
    (void)this->CallDispatch(QTSS_Interval_Role, NULL);

Socket class is asynchronous (non-blocking), and **Signals** Task object

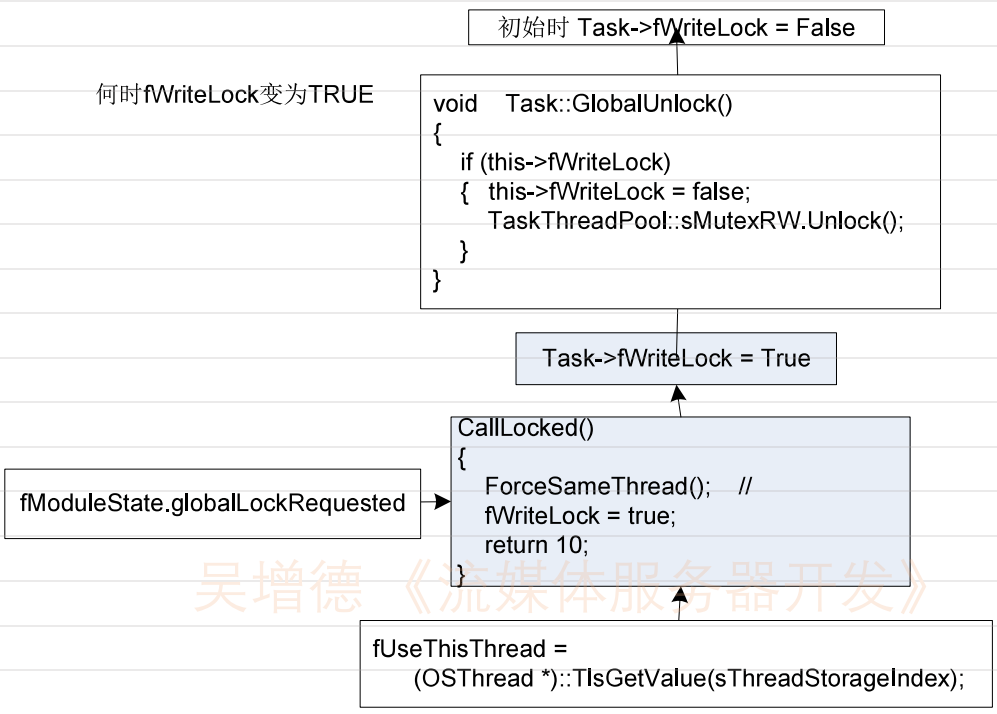
EventThread::Entry()
IdleTaskThread::Entry()
TaskThread::Entry()

以下是整个TaskThread::Entry的运行过程



历史读	历史写	当前读	当前写	Seq
0	0	OK	OK	1
N=1	0	OK	OK?	2
N>1	0	OK	OK	3
				4

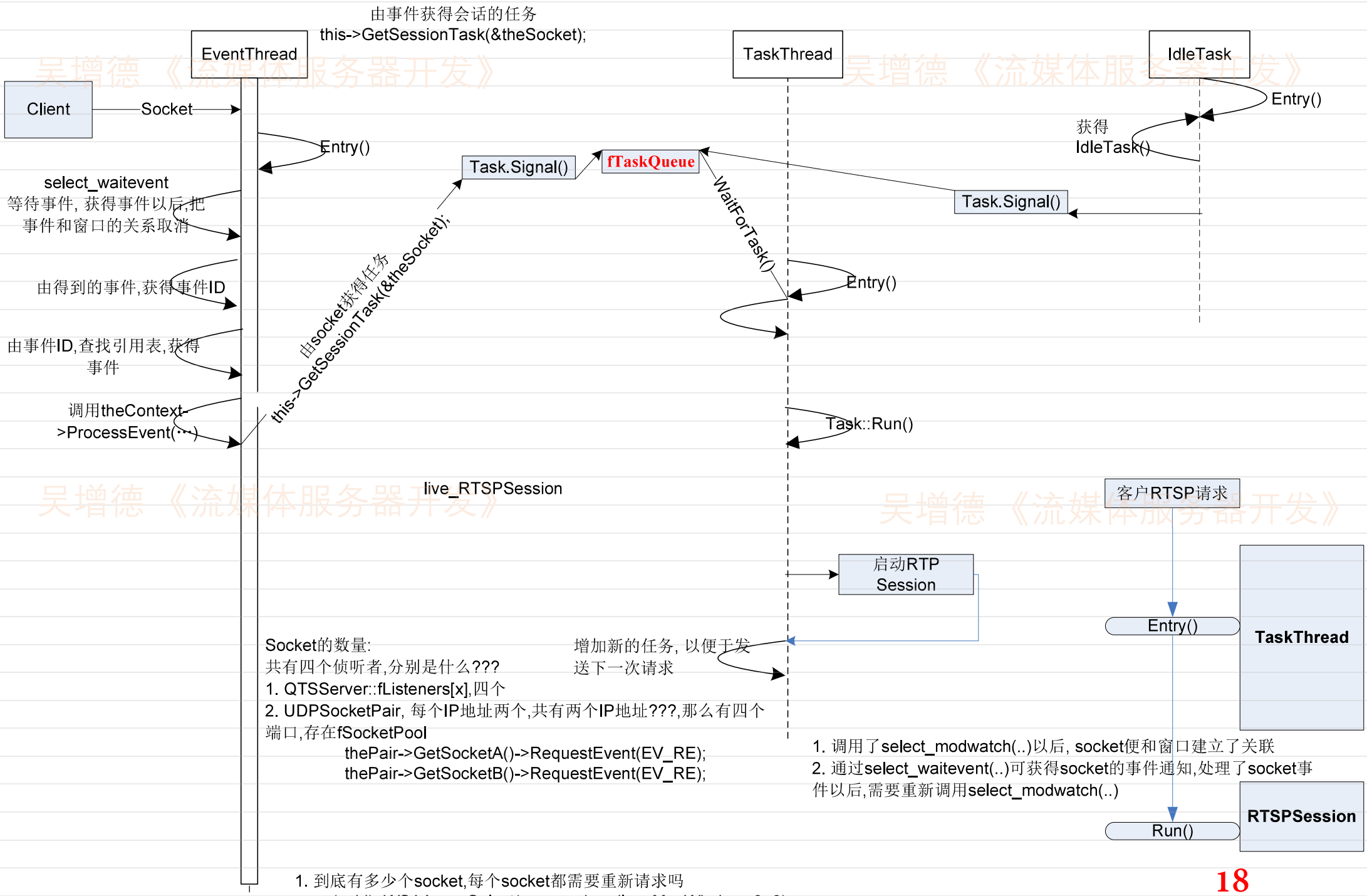
何时fWriteLock变为TRUE



对以上锁的分析可以从四种情况入手:
1. 当前有读, 进行写操作
1. 以上锁的实现目的: 任何时候可以有多个读操作
2.

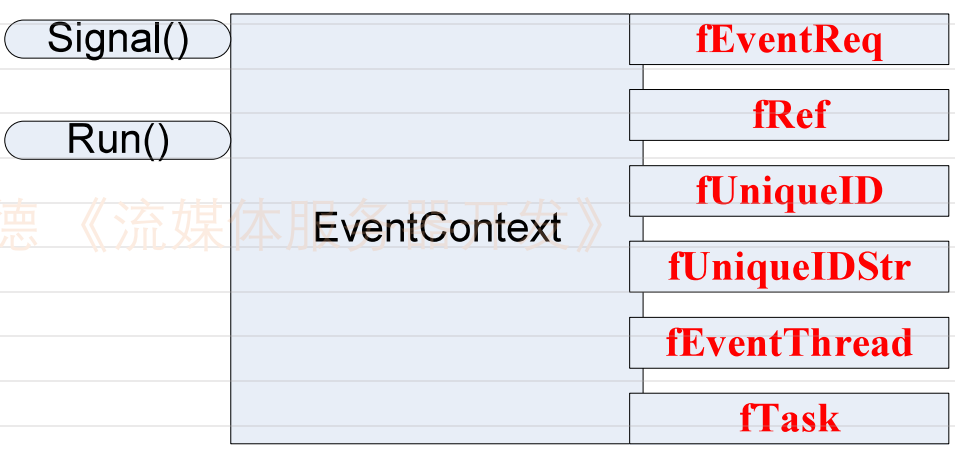
对以上锁的分析可以从左边的表格来看:
1. fWriteLock = TRUE表示当前写, 该变量用于控制线程对任务的调度,一个任务写完了必须进行读操作,除非任务在写锁后面的运行操作中调用了CallLocked()
2. 通过fWriteLock, 任何时候单一个线程对任务进行写操作
3. sMutexRW属性于TaskThreadPool类, 用于不同线程之间的同步

Seq2: 不触发fWritersCond.Signal()直接退出, 可能造成系统死锁
Seq3: 退出时触发fWritersCond.Signal()
Seq2和Seq3退出后 fState = 0
Seq4:

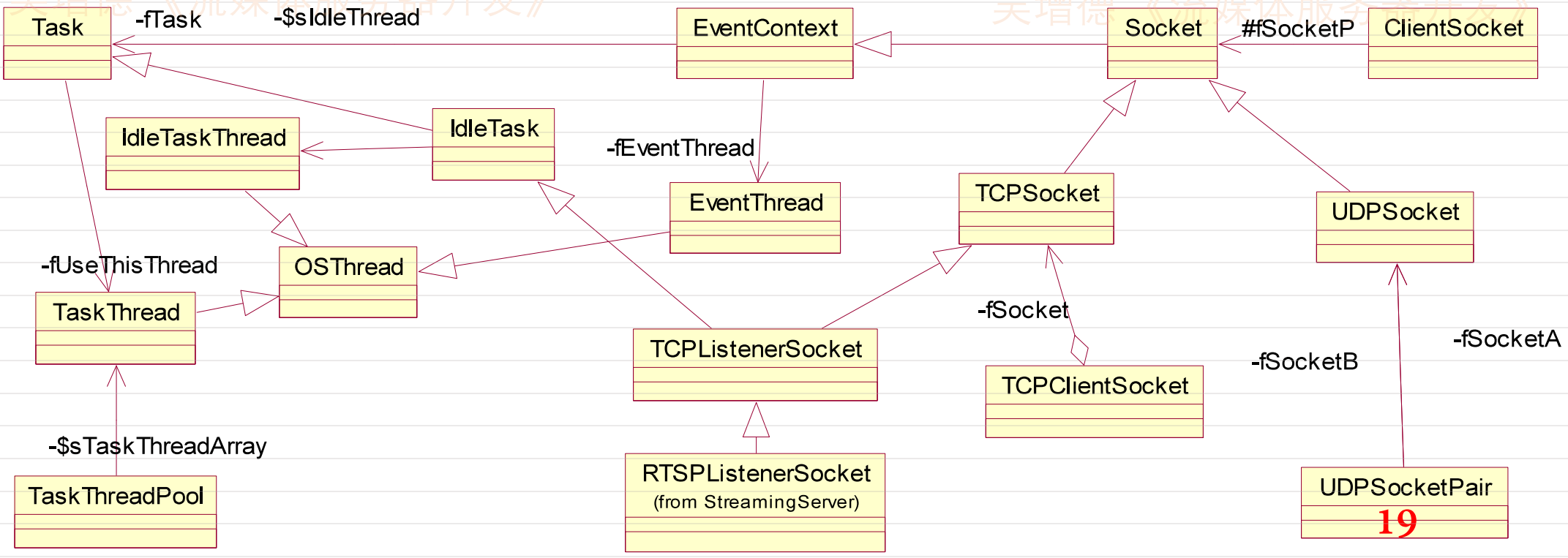


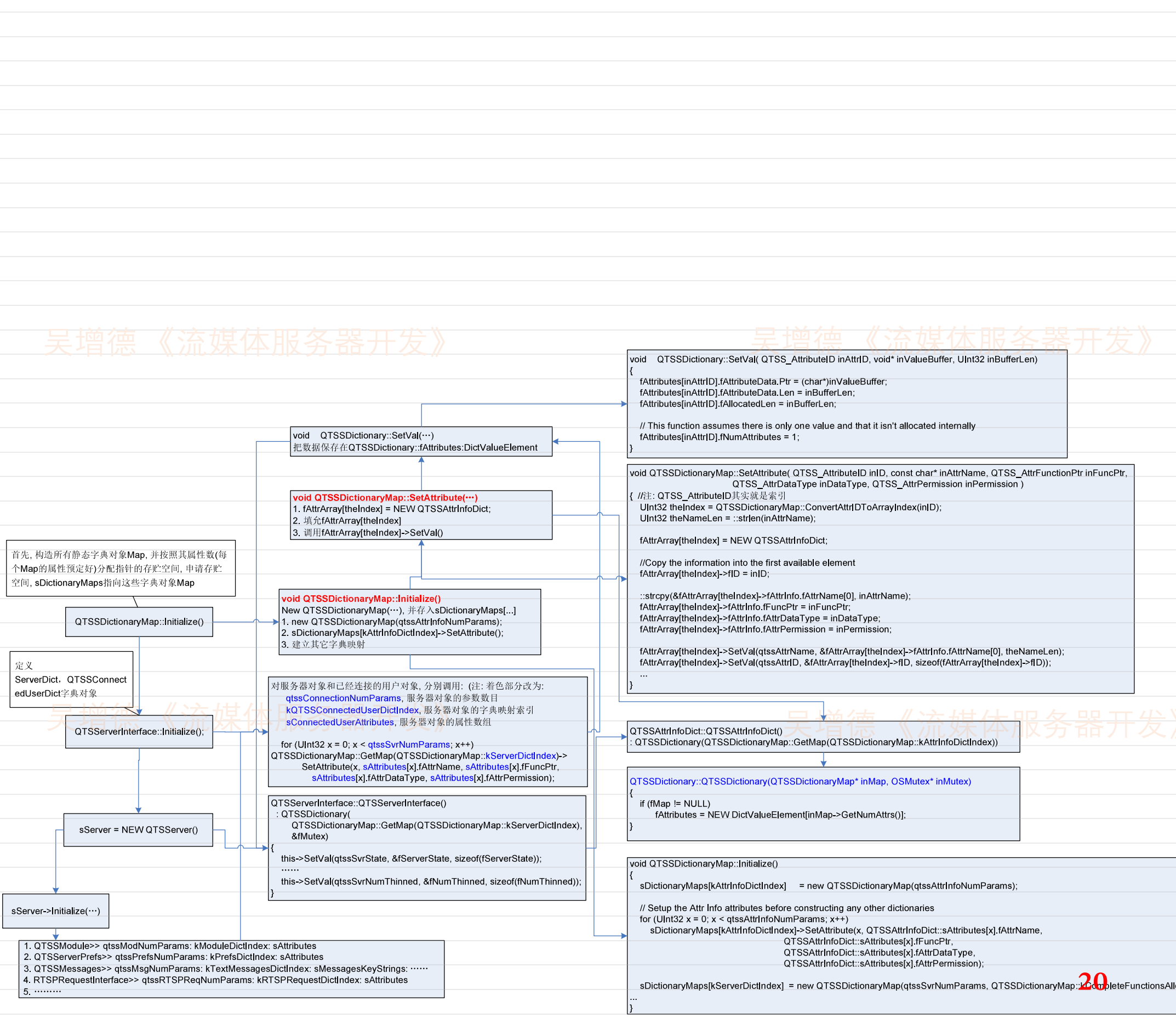
1. 到底有多少个socket,每个socket都需要重新请求吗
(void)::WSAAsyncSelect(req->er_handle, sMsgWindow, 0, 0)

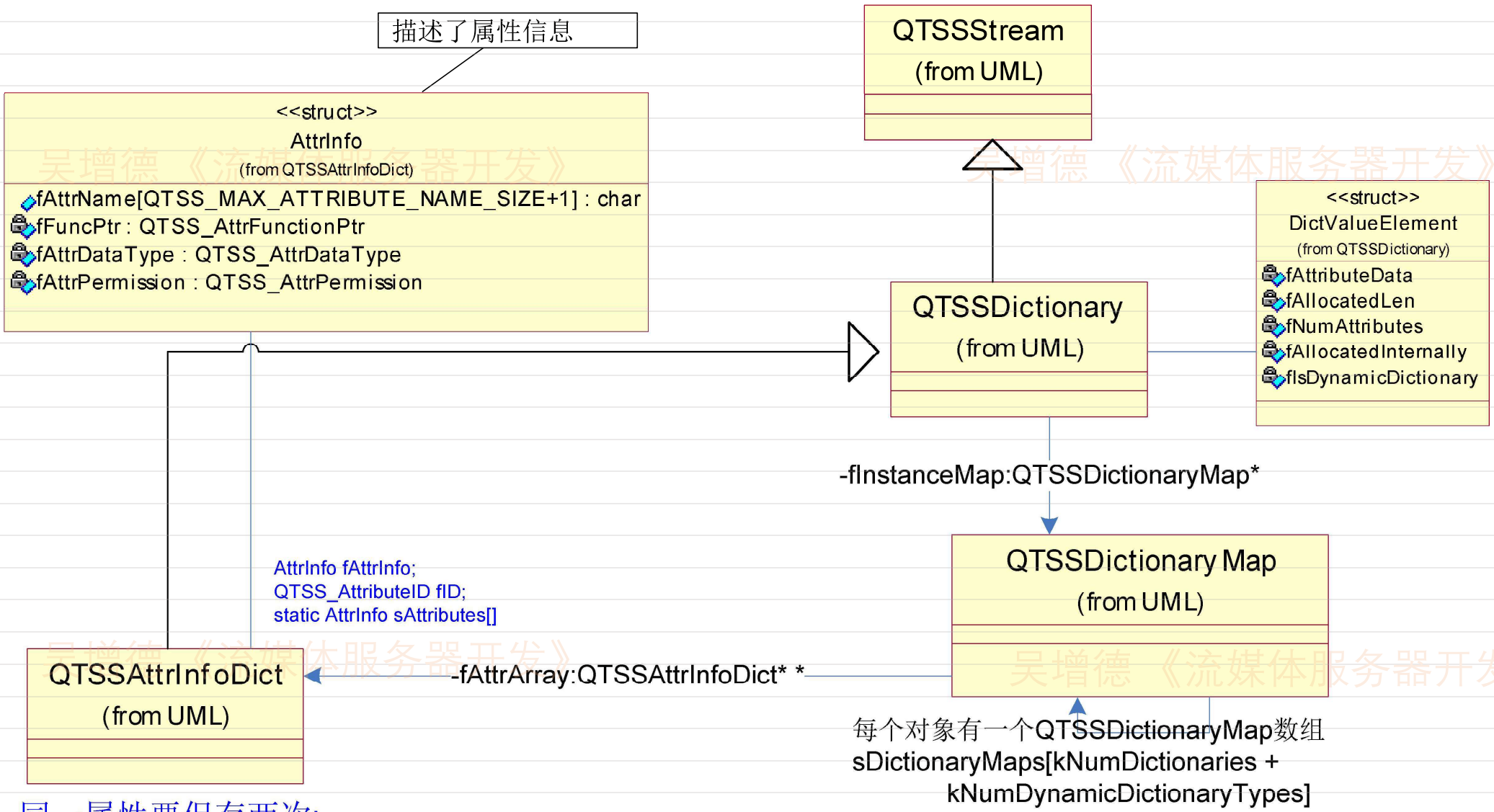
系统中有一个 void Task::Signal(EventFlags events)，函数，用于通知作用



//1. 获得一个消息
//2. 从消息中抽取socket
//3. 调用对应的EventContext(socket)对象处理消息
    a. 默认fTask->Signal(Task::kReadEvent);
    b. RTSP连接时调用TCPListenerSocket::ProcessEvent(int /*eventBits*/)
//4. 处理完以后从引用表中减少对象的引用计数







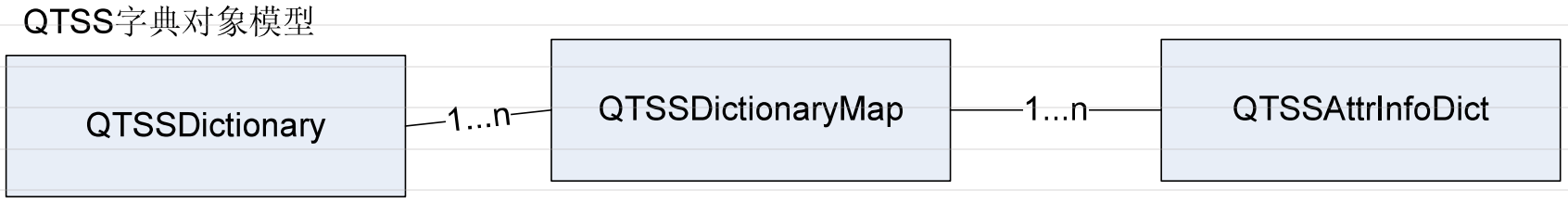
同一属性要保存两次:

1. QTSSAttrInfoDict[Idx]->fAttrInfo内, fAttrInfo是结构为**AttrInfo**的结构
2. QTSSAttrInfoDict[Idx]->fAttributes数组内，它是从QTSSDictionary继承下来的，类型为**DictValueElement**.

inAttrID = (0) qtssAttrName,
inAttrID = (1) qtssAttrID,
inAttrID = (2) qtssAttrDataType,
inAttrID = (3) qtssAttrPermissions,

```
DictValueElement*   fAttributes;
DictValueElement*   fInstanceAttrs;
UInt32              fInstanceArraySize;
QTSSDictionaryMap*  fMap;
QTSSDictionaryMap*  fInstanceMap;
```

QTSS字典对象模型



Content-Managing Modules

QTSSFileModule,
QTSSReflectorModule,
QTSSRelayModule,
QTSSMP3StreamingModule.

Access Control Modules

QTSSAccessModule,
QTSSHomeDirectoryModule,
QTSSHttpFileModule,
QTSSSpamDefenseModule.

Server-Support Modules

QTSSErrorLogModule,
QTSSAccessLogModule,
QTSSWebStatsModule,
QTSSWebDebugModule,
QTSSAdminModule,
QTSSPOSIXFileSystemModule

Modules implement HTTP

QTSSAdminModule
QTSSMP3StreamingModule
QTSSWebStatsModule
QTSSHTTPStreamingModule
QTSSRefMovieModule
QTSSWebStats
QTSSWebDebugModule

QTSServer::LoadCompiledInModules()
QTSServer.CPP

1. 创建模块 QTSSModule* theFileModule =
   new QTSSModule("QTSSFileModule");

2. 设置模块
   theFileModule->SetupModule
   (&sCallbacks, &QTSSFileModule_Main);
   函数的主要目的是设置新建立模块的fDispatchFunc
   该函数包括两个参数: 1. 2. 所加载模块的Dispatch例程

3. 增加模块 AddModule(theFileModule);
   a. 调用模块的注册角色
   b. 读取模块参数, 并氢读取的模块参数设置为模块字典
   c. 把模块加入qtssSvrModuleObjects指向的表项
   d. 把模块加入模块队列

QTSSModule.CPP

QTSSModule::QTSSModule(char* inName, char* inPath)

1. 设置fQueueElem, 设置任务名"QTSSModule"
2. 初始化并设置属性fAttributes, fRoleArray, fModuleState

QTSSModule::SetupModule(QTSS_CallbacksPtr inCallbacks,
QTSS_MainEntryPointPtr inEntrypoint)

1. 构造局部变量QTSS_PrivateArgs
   thePrivateArgs.inCallbacks = inCallbacks

2. 触发Dispatch例程
   theErr = (inEntrypoint)(&thePrivateArgs)
   传过来的sCallBacks在thePrivateArgs中

3. 设置fDispatchFunc
   比如 fDispatchFunc = QTSSFileModuleDispatch

_stublibrary_main
(inPrivateArgs, QTSSFileModuleDispatch);

QTSSFileModule

QTSSFileModule_Main(void* inPrivateArgs)

QTSS_Private.cpp

… _stublibrary_main(void* inPrivateArgs, QTSS_DispatchFuncPtr inDispatchFunc)

1. 设置回调函数, sCallbacks = theArgs->inCallbacks;
2. 设置错误流, sErrorLogStream = theArgs->inErrorLogStream;
3. 设置模块的回调函数theArgs->outDispatchFunction = inDispatchFunc;

```
//在QTSS_Private.h定义了回调数组
typedef struct {
    QTSS_CallbackProcPtr addr [kLastCallback];
} QTSS_Callbacks, *QTSS_CallbacksPtr;

//每个回调例程的索引，回具体的回调例程地址放在数组内
enum
{
    kNewCallback                = 0,
    kDeleteCallback             = 1,
…
    kLastCallback               = 61
}
所有回调函数的实现方法如下
void QTSS_Delete(void* inMemory)
{
    (sCallbacks->addr [kDeleteCallback]) (inMemory);
}
```

QTSSCallBacks实现了sCallbacks所指向的所有函数，并实现为静态函数
QTSServer::InitCallbacks()函数建立了之间的关系

QTSSCallbacks::QTSS_AddRole
QTSSCallbacks.cpp

函数定义了两个sCallbacks:
1. static QTSS_CallbacksPtr sCallbacks = NULL;
2. QTSServer.h类内定义: QTSS_Callbacks QTSServer::sCallbacks;
这两个sCallbacks是有区别的，在QTSServer.h定义的是实际的数据结构，而在QTSS_Private定义的数据结构只是指向前一个数据结构的指针

QTSS_ModuleState* theState = OSThread::GetMainThreadData()
theState = OSThread::GetCurrent()->GetThreadData();

theState->curModule->AddRole(inRole)

1. 每个回调例程的索引枚举, 数级
   QTSS_CallbackProcPtr addr [kLastCallback]
2. QTSS_PrivateArgs的定义
3. QTSS_ModuleState的定义

1.各种枚举类型、数据类型的定义、各种结构定义、
4.所有QTSS API必须实现的功能:
a.主入口点_stublibrary_main
、分发入口点、
b. 所有模块内使用的函数

C和C++对函数的处理方式是不同的.extern "C"是使C++能够调用C写作的库文件的一个手段，如果要对编译器提示使用C的方式来处理函数的话，那么就要使用extern "C"来说明。
Windows下的DLL、Linux下的库或是C++中调用C的库，均可能用到import/export

只定义了一个函数

QTSS_Private.h

QTSS.h

QTSSReflectorModule.h
QTSSRelayModule_Main(void* inPrivateArgs)

调用QTSSRelayModule_Main以后，由ABCModuleDispatch函数接管，Dispatch函数注册了相关的角色

QTSS_Private.cpp
_stublibrary_main（inPrivateArgs, QTSS_DispFunPtr.）
return (sCallbacks->addr [kAddRoleCallback]) (inRole);

QTSSRelayModule.cpp

QTSS_Error QTSSCallbacks::
QTSS_SendStandardRTSPResponse(…)
该函数应该仔细看，包含了RTSP的主要内容

QTSS字典对象模型

QTSSCallbacks.cpp

QTSS_AddRTPStream(
QTSS_ClientSessionObject inClientSession,
QTSS_RTSPRequestObject inRTSPRequest,
QTSS_RTPStreamObject* outStream,
QTSS_AddStreamFlags inFlags)

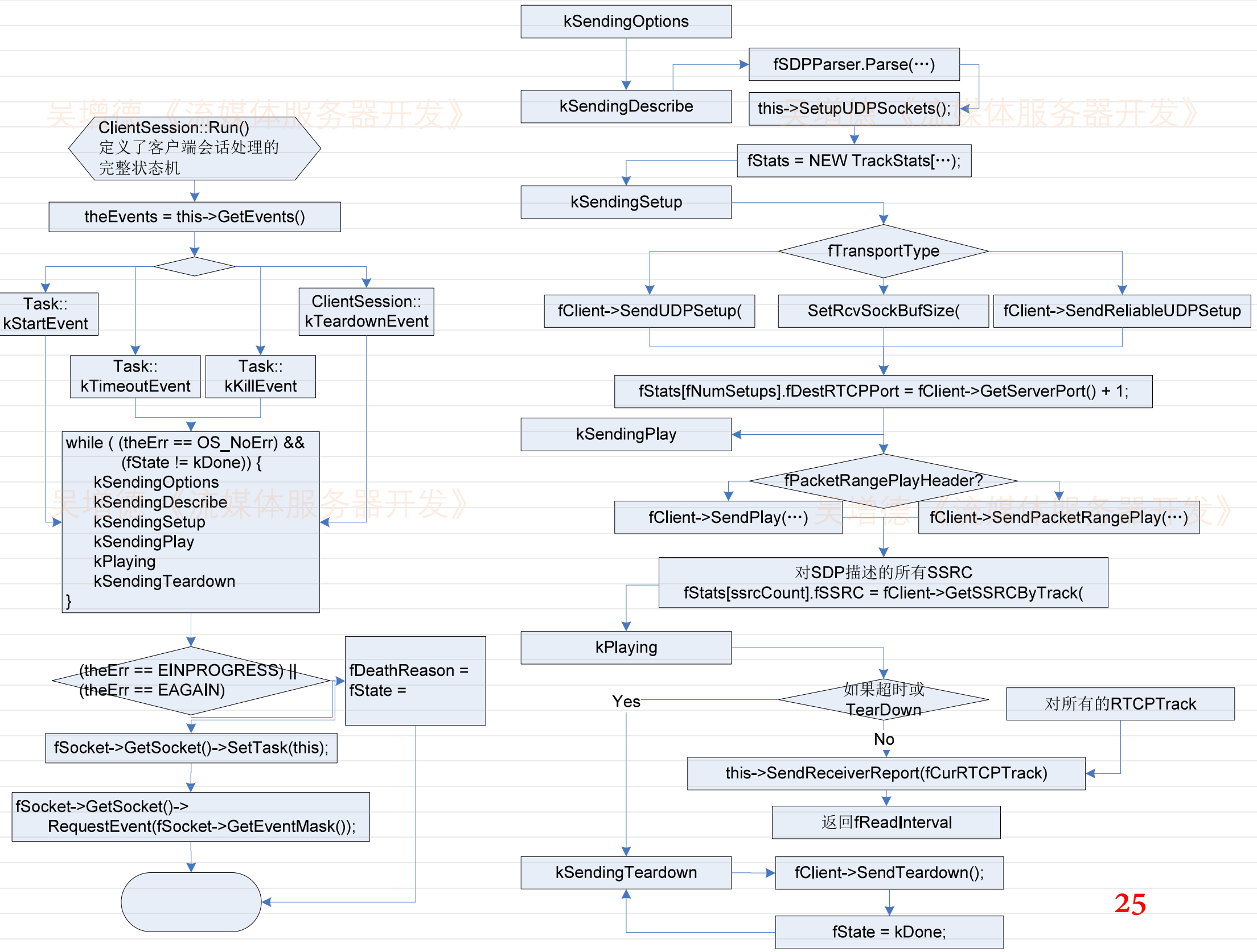
QTSS_AddRTSPStream回调例程使一个模块可以向客户端发送RTP数据包，以响应RTSP请求。多次调用QTSS_AddRTSPStream函数可以向会话中加入多个流。

((RTPSession*)inClientSession)->AddStream(
(RTSPRequestInterface*)inRTSPRequest,
(RTPStream**)outStream, inFlags)

RTSPSessionInterface

提供了模块使用的会话级资源API

tobe分析140页



Movie atom 'moov'
[P29], [P32] list of layout

Movie header atom
'mvhd' [P33]

Movie clipping atom
'clip' [P43]
Clipping region atom
'crgn' [P44]

Color table atom
'ctab' [P35]

User data atom
'udta' [P36]
Movie user data
'stsz' [P37]

Track atom 'trak'
[P39]

Track header atom
'tkhd' [P41]

Track clipping atom
'clip' [P43]
Clipping region atom
'crgn' [P43]

Track matte atom
'matt' [P44]
Compressed matte atom
'kmat' [P45]

Edit atom
'edts' [P46]
Edit list atom
'elst' [P47]

Track reference atom
'tref' [P49]

Track load setting atom
'load' [P48]

Track input map atom
'load' [P51]

User data atom
'load' [P36]

Media atom 'mdia'
[P54]

Media header atom
'mdhd' [P55]

Media handler reference atom
'hdlr' [P57]

User data atom
'load' [P36]

Video media information atom 'minf' [P58] / Sound media information atom 'minf' [P61] / Base media information atom 'minf' [P63]

Video media information header atom
'vmhd' [P59]

Data handler reference atom
'hdlr' [P57]

Data information atom 'dinf' [P65]
Data reference atom
'dref' [P66]

Sample table atom 'stsc'
[P68]

Sample description atom
'stsd' [P70]

Time-to-sample atom
'stss' [P71]

Sync sample atom
'stss' [P73]

Sample-to-chunk atom
'stsc' [P75]

Sample size atom
'stsz' [P76]

Chunk offset atom
'scto' [P78]

gDvrInterface[16]
DVRInterface

{该数据结构在所有对象间共享}

nDVRIPAddress, unsigned long
nVideoAdapterDVRID
bIsLogined, BOOL 该DVR Login?

ChannelList[16], ChannelItem

BOOL bIsPlaying

ChannelConnects * pConnection

指向特定通道的连接列表

theStream, QTSS_Object
QTSS_RTPSendPackets_Params
ChannelConnects *pNext
nDelayTime
nVideoQuality
fStreamingHeaderSented
pNext

ChannelStorage* pStorage

strFileName
fileHandle
nStartTime
nDuration
pNext

pPreRecord, nPreRecordPos, 关于预录的信息
nSeqNum, 1 ~4, for 25, 17, 9, 0.25 FPS
HIKStreamingHeader, HIKStreamingHeaderInited

FileSession
(QTSSFileModule.cpp)

QTSS_RTPStreamObject fStream
SInt64 fAdjustedPlayTime;
RealtimeDataRelay *fRealtimeDataRelay;

HDDVRSession
(QTSSHDDVRModule.cpp)

QTSS_RTPStreamObject fStream
QTSS_PacketStruct fPacketStruct
UInt32 fNextPacketLen
fStartTime, fStopTime
fSpeed, fAdjustedPlayTime
RealtimeDataRelay* fRealtimeDataRelay

RealtimeDataRelay
(RealtimeDataRelay.h)

UInt32 fSSRC
void* fCookie
int fChannelIndex
unsigned long nDVRIPAddress
int fDVRIndex
StrPtrLen strPath
InitDVR
LoginDVR

增加连接的过程

RTSPSession::Run()

QTSSModule::CallDispatch(

QTSSHDDVRModuleDispatch(

ProcessRTSPRequest(

DoDescribe(QTSS_Standar

CreateHDDVRFileSession(..)

theRealtimeData = NEW RealtimeDataRelay()
theRealtimeData->Initialize(inPath)

void RealtimeDataRelay::Initialize(

InitDVR();
LoginDVR();
PlayChannel();

0. nRTPVersion		02. nSeqNum1	
04. nSeqNum1		6. DEVICE_TYPE	7.
8	9	10	11
12. nDVRIndex		Video Data	

QTSS_Error DoSetup(

QTSS_AddRTPStream(

QTSS_Error RTPSession::AddStream(

QTSS_Error RTPStream::Setup(

QTSS_Error DoPlay(

QTSS_Error QTSS_Play(

QTSS_Error QTSSCallbacks::QTSS_Play(

QTSS_Error RTPSession::Play(
this->Signal(Task::kStartEvent)

增加连接的过程

RTPSession::Run()

QTSSModule::CallDispatch(

QTSSHDDVRModuleDispatch(

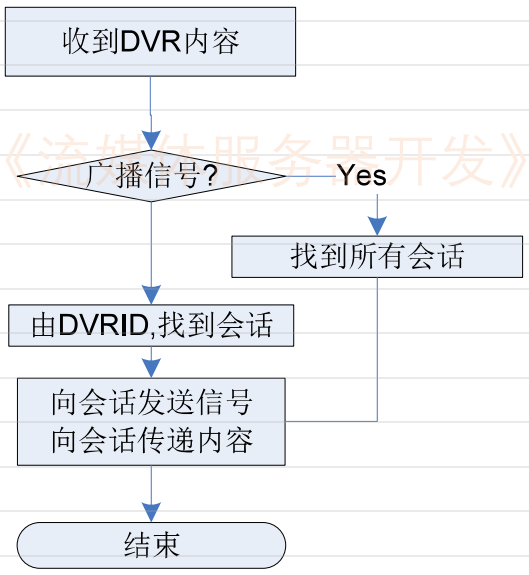
ProcessRTPData(

RealtimeDataRelay::AddStream(…)

- 申请内存
- 写入RTPVer, DVRID
- 分别针对不同的帧率进行操作
- 对数据进行分析，并把分析结果写入缓存
  {
- 对不同的连接分别写数据流

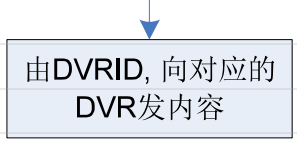
1. 从连接字符串获得nDVRID, 以及nChannelCode
2. 创建RealtimeDataRelay(nDVRID, nChannelCode);
3.

1. InitDVR(); 读DVR.ini, 初始化gDvrInterface, 初始化DLL
2. LoginDVR();
3. PlayChannel();

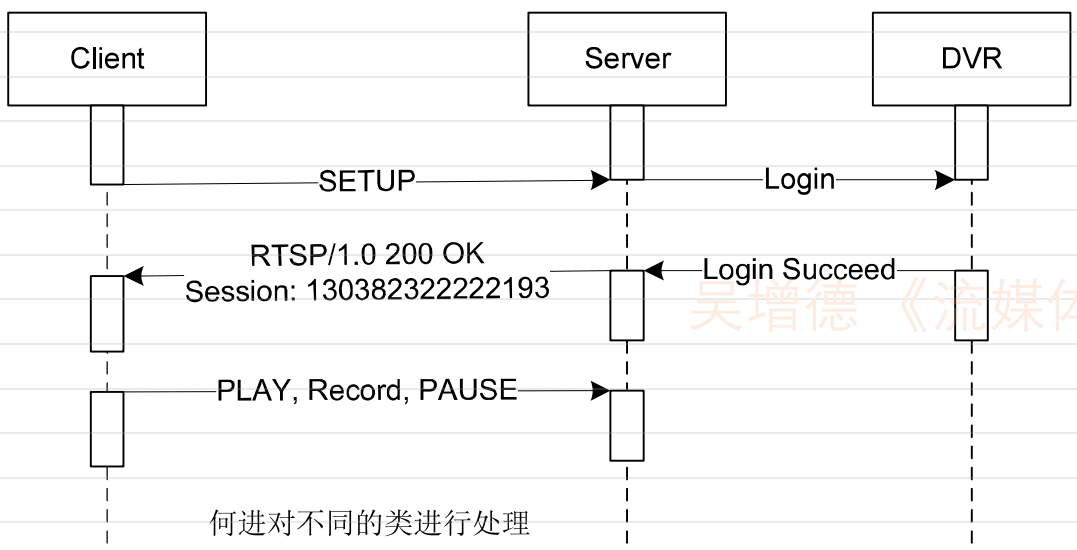


#ifndef WUZENGDE

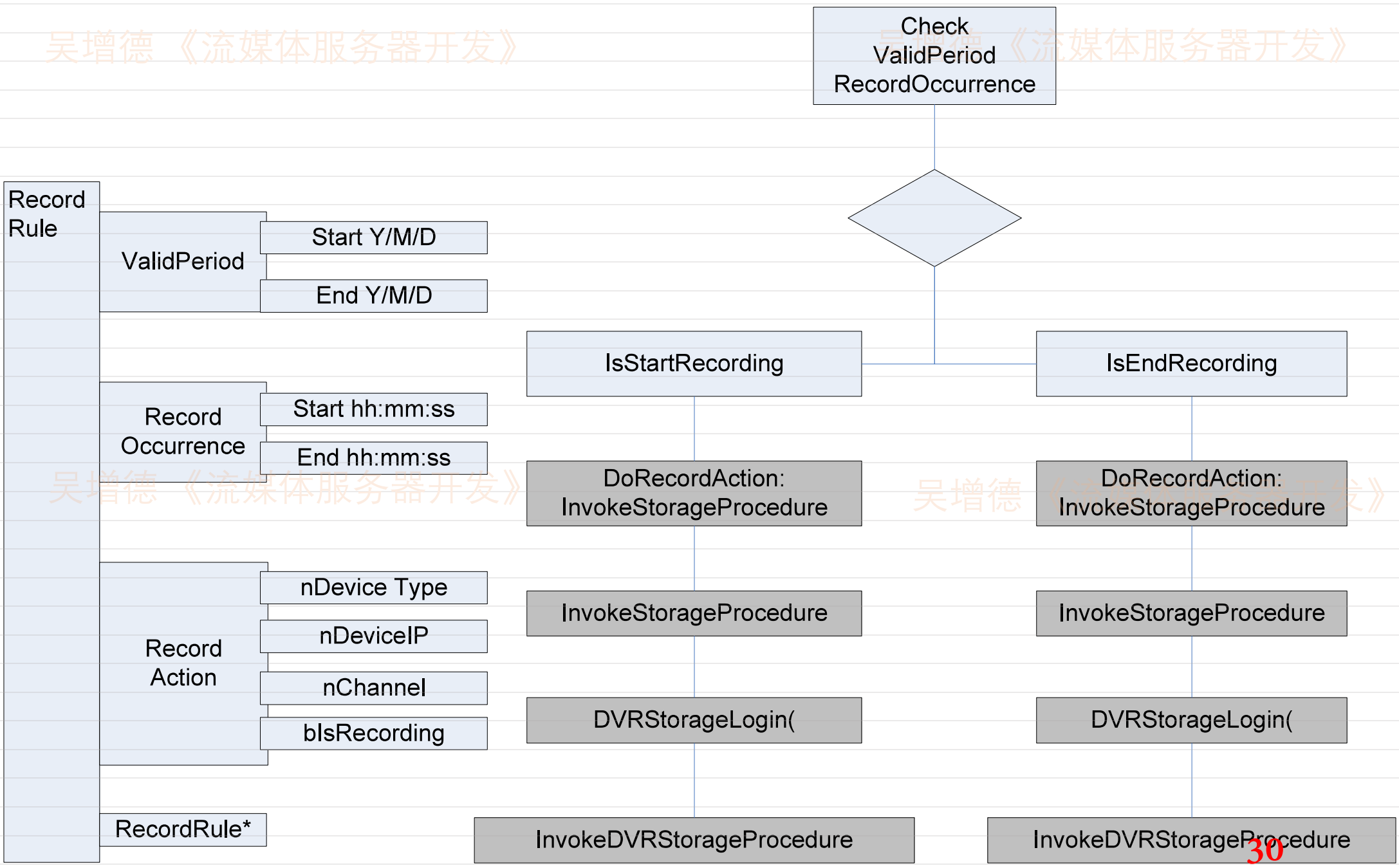
#endif



RealtimeDataRelay
. nDVRID
. nChannelCode
. theStream
. RealtimeDataRelay()
. ~RealtimeDataRelay()
. AddStream(QTSS_Object theStream,
QTSS_RTPSendPackets_Params* inParams);



QTSSFileModuleDispatch(
.ProcessRTSPRequest(
.DoDescribe(
.CreateQTRTPFile(



增加连接的过程

RTPSession::Run()

QTSSModule::CallDispatch(

QTSSHDDVRModuleDispatch(

ProcessRTSPRequest(

DoDescribe(QTSS_Standar

CreateHDDVRFileSession(..)

theRealtimeData = NEW RealtimeDataRelay()
theRealtimeData->Initialize(inPath)

void RealtimeDataRelay::Initialize(

InitDVR();
LoginDVR();

0. nRTPVersion		02. nSeqNum1	
04. nSeqNum1		6. DEVICE_TYPE	7.
8	9	10	11
12. nDVRIndex		Video Data	

//start WUZENGDE

//end WUZENGDE

#ifndef WUZENGDE

#endif

QTSS_Error DoSetup(

QTSS_AddRTPStream(

QTSS_Error RTPSession::AddStream(

QTSS_Error RTPStream::Setup(

QTSS_Error DoPlay(

QTSS_Error QTSS_Play(

QTSS_Error QTSSCallbacks::QTSS_Play(

QTSS_Error RTPSession::Play(
this->Signal(Task::kStartEvent)

增加连接的过程

RTPSession::Run()

QTSSHDDVRModuleDispatch(

ProcessRTPData(

RealtimeDataRelay::AddStream(…)

PlayChannel();

DoDescribe

Support Device?

Return

CreateHDDVRFileSession

Succeed?

TearDown

Return

CreateSDPInformation

SendSDPResponse

DoSetup

Support Device?

Return

Get The File?

Get Relative Path
theRelativePath

UseLess?

Set the File

QTSS_AddRTPStream

QTSS_Error RTPSession::AddStream(
建立RTPStream

Set the Property
bufferDelay
thePayload
thePayloadType
theTimescale
theTrackID
sNumQualityLevels

QTSS_SendStandardRTSPResponse

DoPlay

Support Device?

Return

Set Related Value

QTSS_Play(…)

中间这几步
能否省掉

QTSS_SendStandardRTSPResponse

DoDescribe
Support Device?
Return
DVRRelaySession
Succeed?
TearDown
Return
CreateSDPInformation
SendSDPResponse

DoSetup
Support Device?
Return
Get The File?
Get Relative Path
theRelativePath
Return
UseLess?
Set the File
QTSS_AddRTPStream
QTSS_Error RTPSession::AddStream(
建立RTPStream
Set the Property
bufferDelay thePayload
thePayloadType theTimescale
theTrackID sNumQualityLevels
QTSS_SendStandardRTSPResponse

DoPlay
Support Device?
Set Related Value
QTSS_Play(…)
中间这几步
能否省掉
QTSS_SendStandardRTSPResponse

Relay Client Test Code (RCTC)
RelayClientCaller (RCC)
Relay Client DLL (RCD)
Relay Server (RS)
Relay Server DVR Adaptor
(RSDA)
DVR (RS)

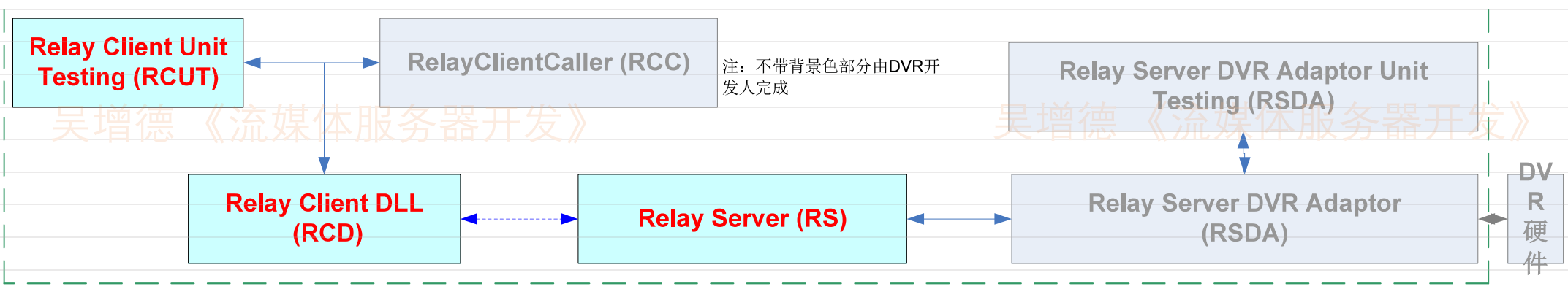
RelayClientDLL

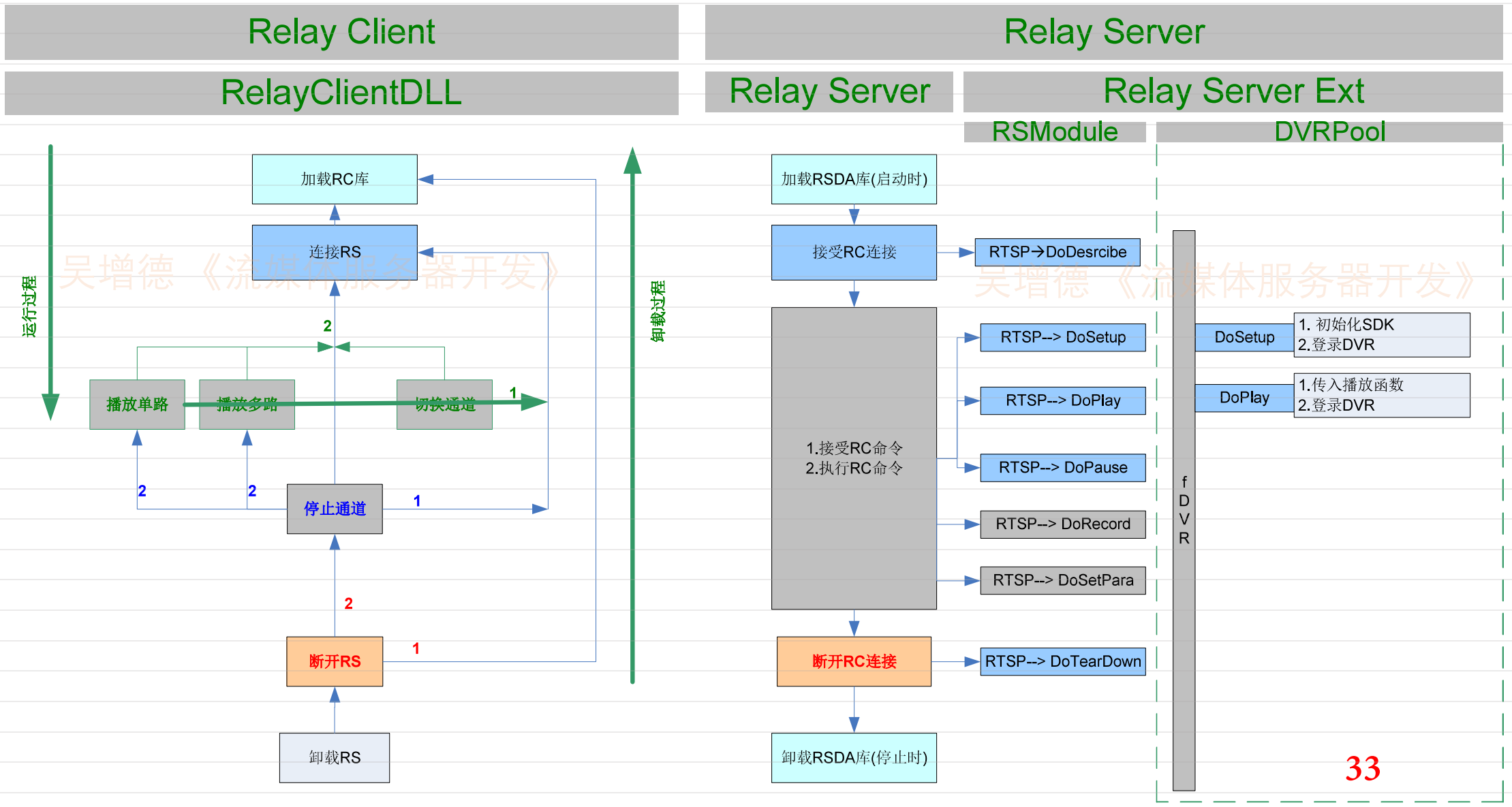
运行过程
加载RC库
连接RS
2
播放单路
播放多路
切换通道
1
2
2
停止通道
1
2
断开RS
1
卸载RS

RelayServer

卸载过程
加载RSDA库(启动时)
接受RC连接
RTSP→DoDesrcibe
RTSP--> DoSetup
1.接受RC命令
2.执行RC命令
RTSP--> DoPlay
RTSP--> DoPause
断开RC连接
RTSP--> DoTearDown
卸载RSDA库(停止时)

视频转发相关工程





RSDA

1. 初始化SDK
2.登录

CreateRelayServer

CreateDeviceInServer

fDVRIP
fDVR
fDVRHash

CreateChannelInDevice

DestroyChannel

fDVRHash

DestroyDevice

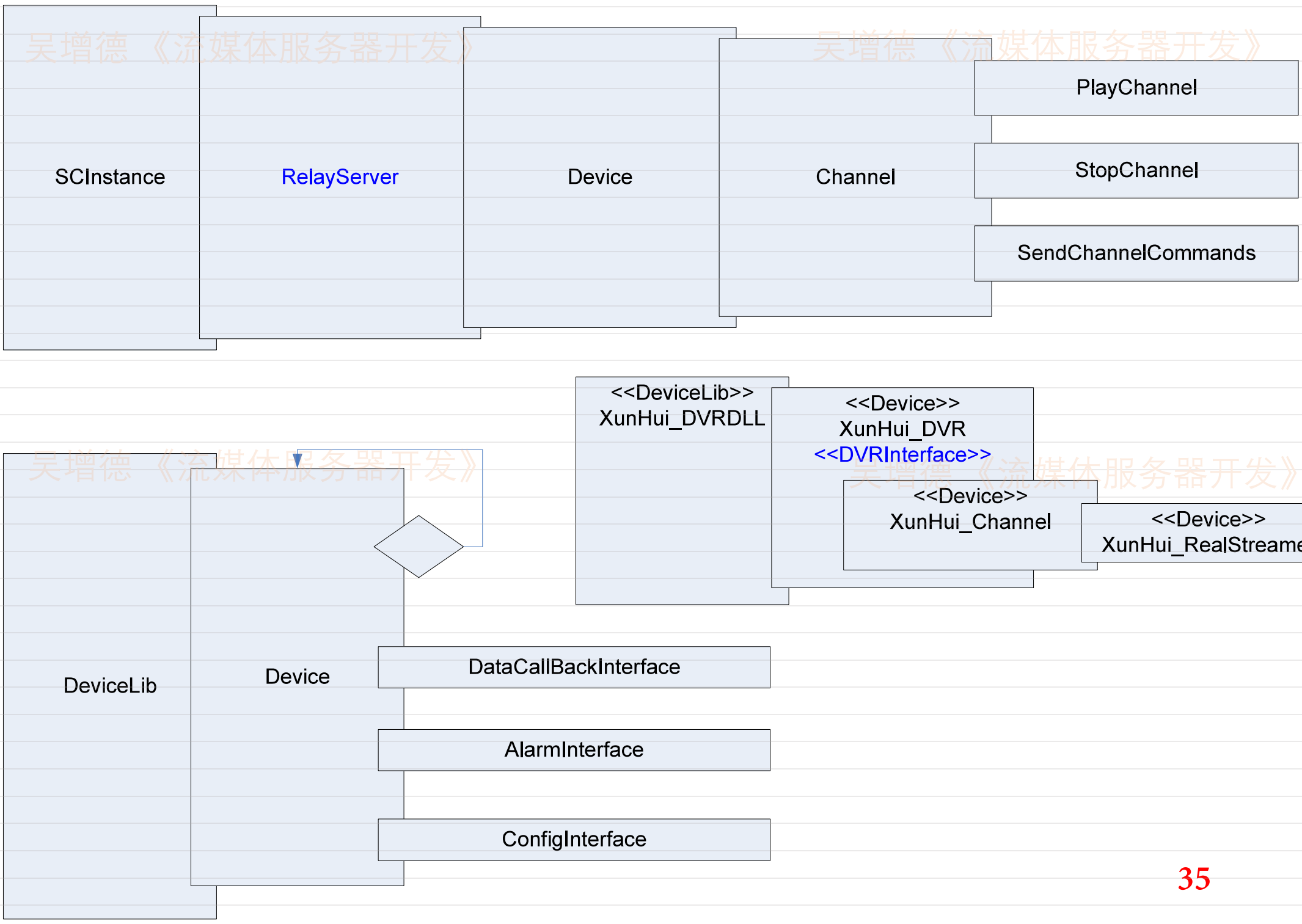
CreateChannelInDevice

Create时:
返回Handle
1. 如果Handle相同，引用数加1
2. 如果Handle不同，则为该Handle设置引用数，建立数据结构

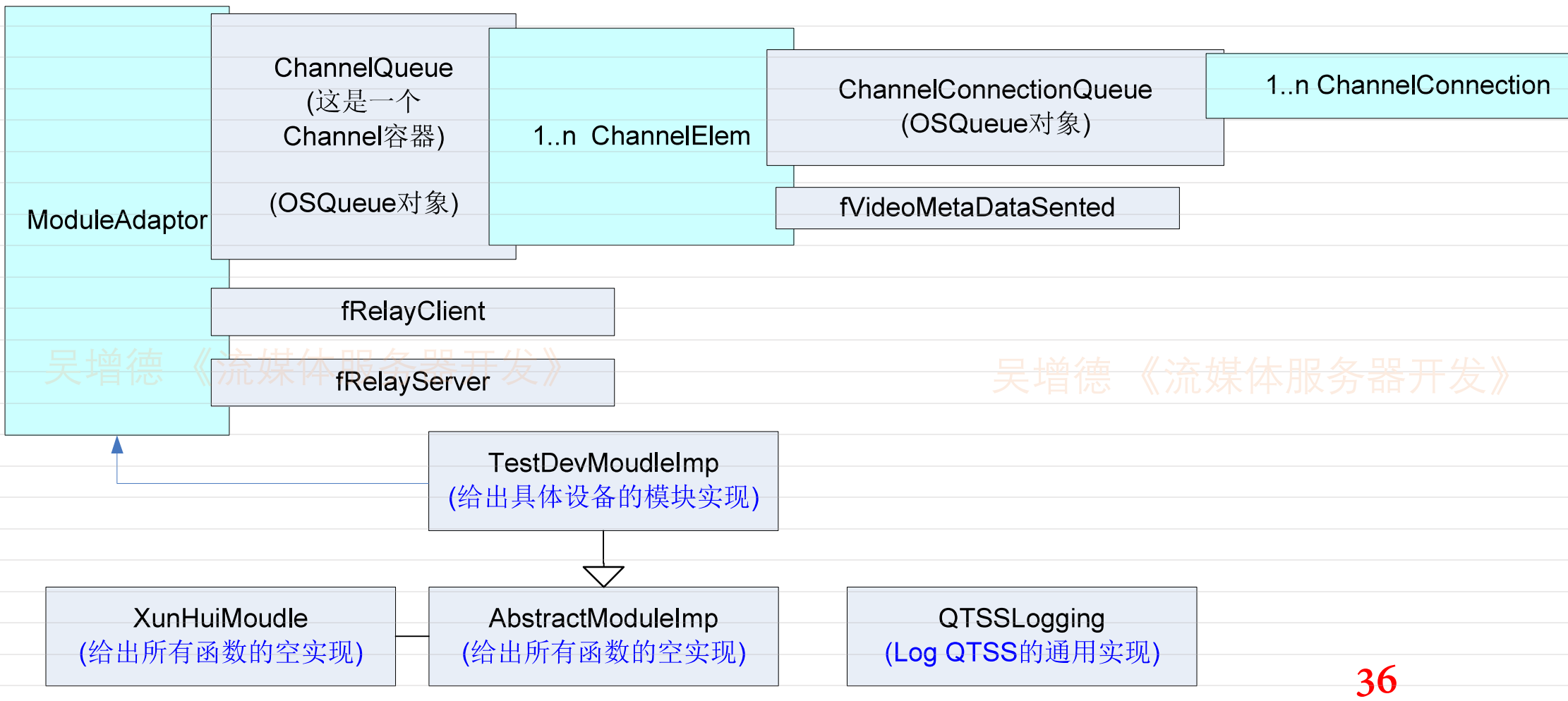
Destroy时
1. 查找该Destroy对应的Handle
2. 如果Handle存在，引用数减1，如果Handle引用为0，由删除该对象的引用结构树，如果长期未使用，则删除??

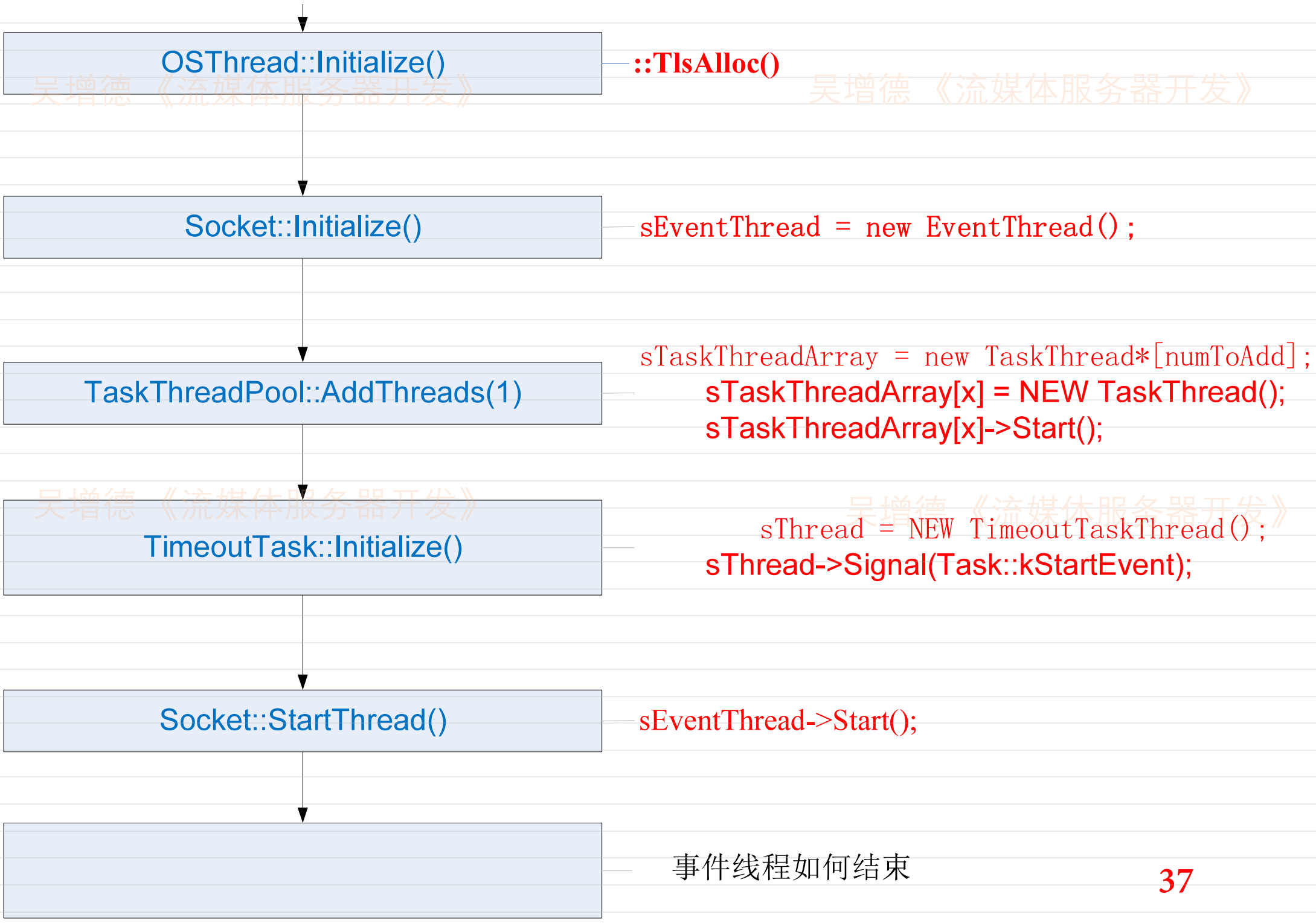
1. 用户名何进传入
a.暂时从初始化文件中读取用户名和密码
2.是否远程控制PTZ
a.由于网络时延，一般不控制
3.是否播放历史视频
a.播放历史视频采用和实时视频同样的机制

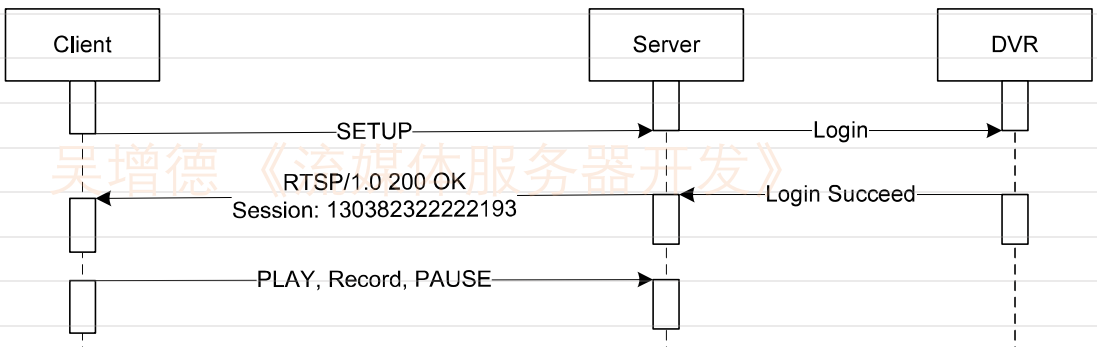
Relay Client 对象结构



Server Module







#ifndef WUZENGDE

增加锁

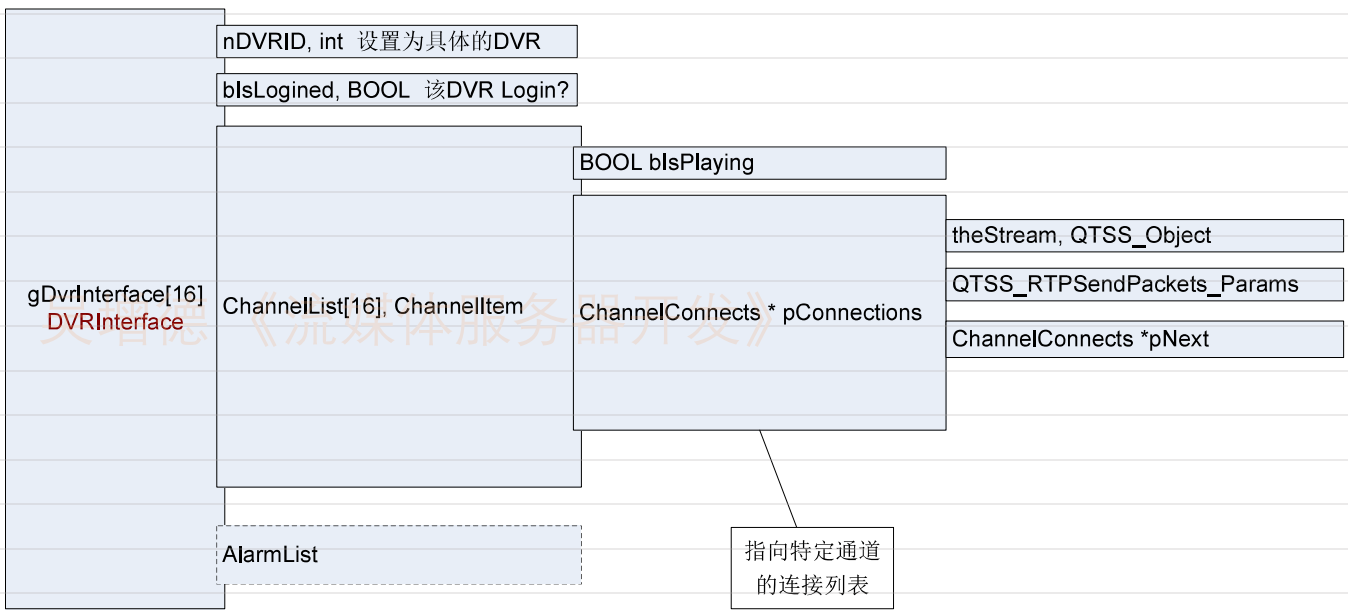
#endif

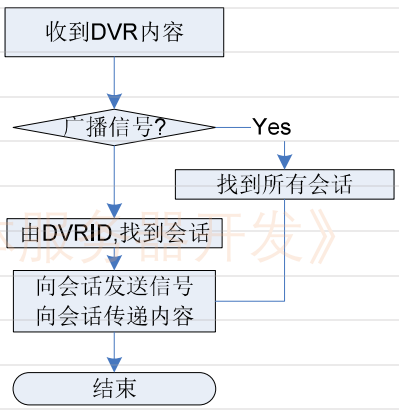
#ifdef WUZENGDE

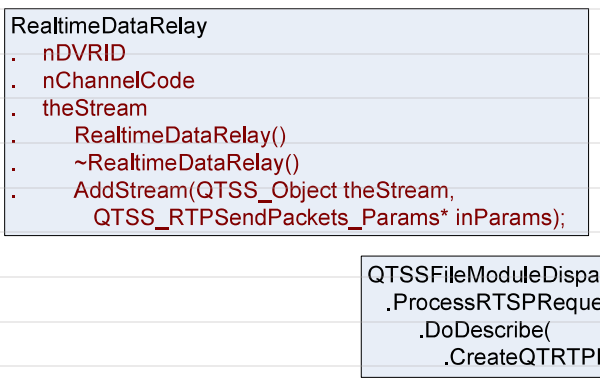
#endif

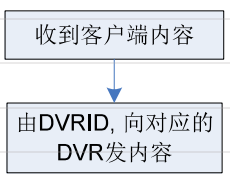
何进对不同的类进行处理

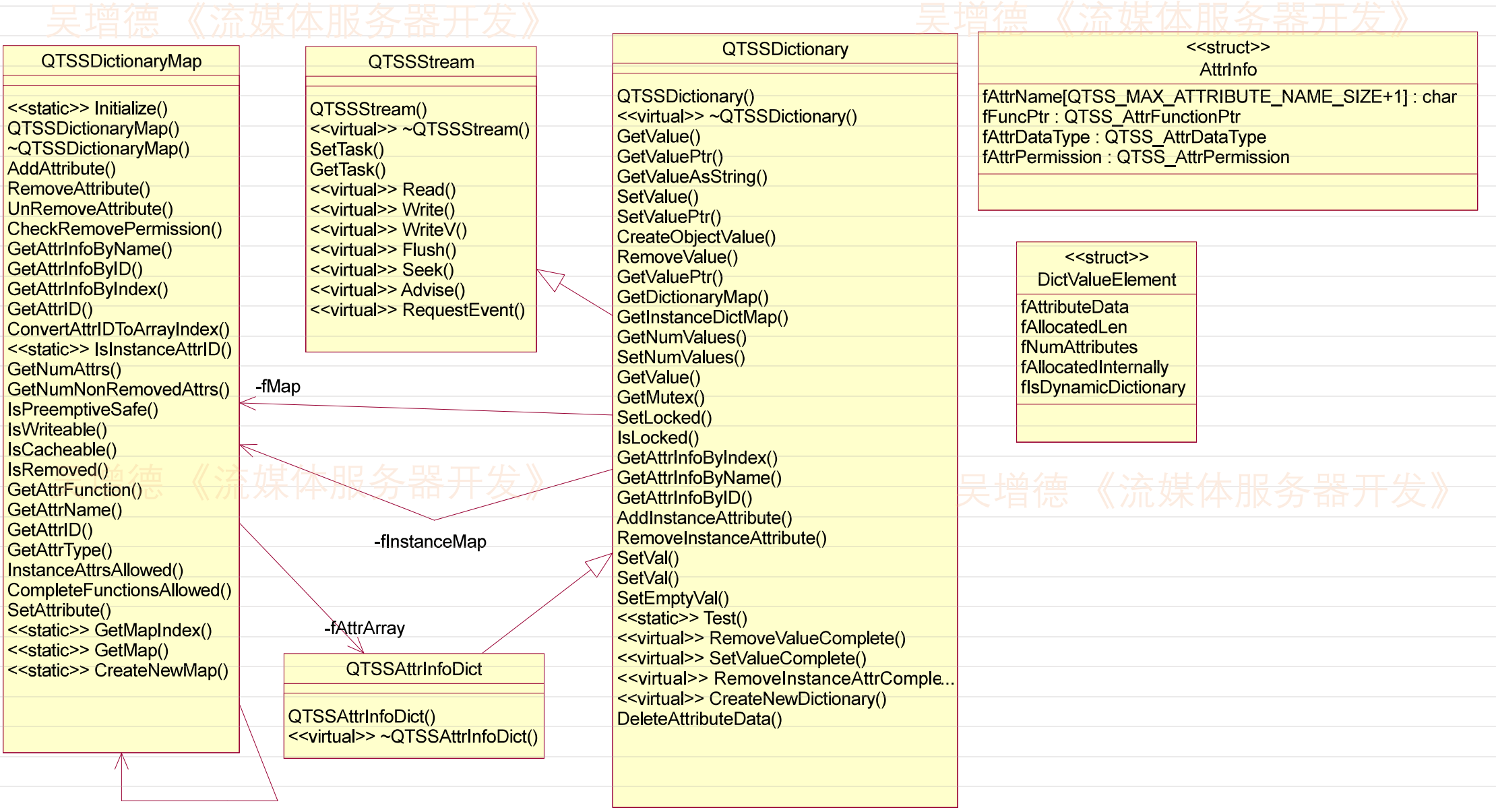
对RTSP进行扩展



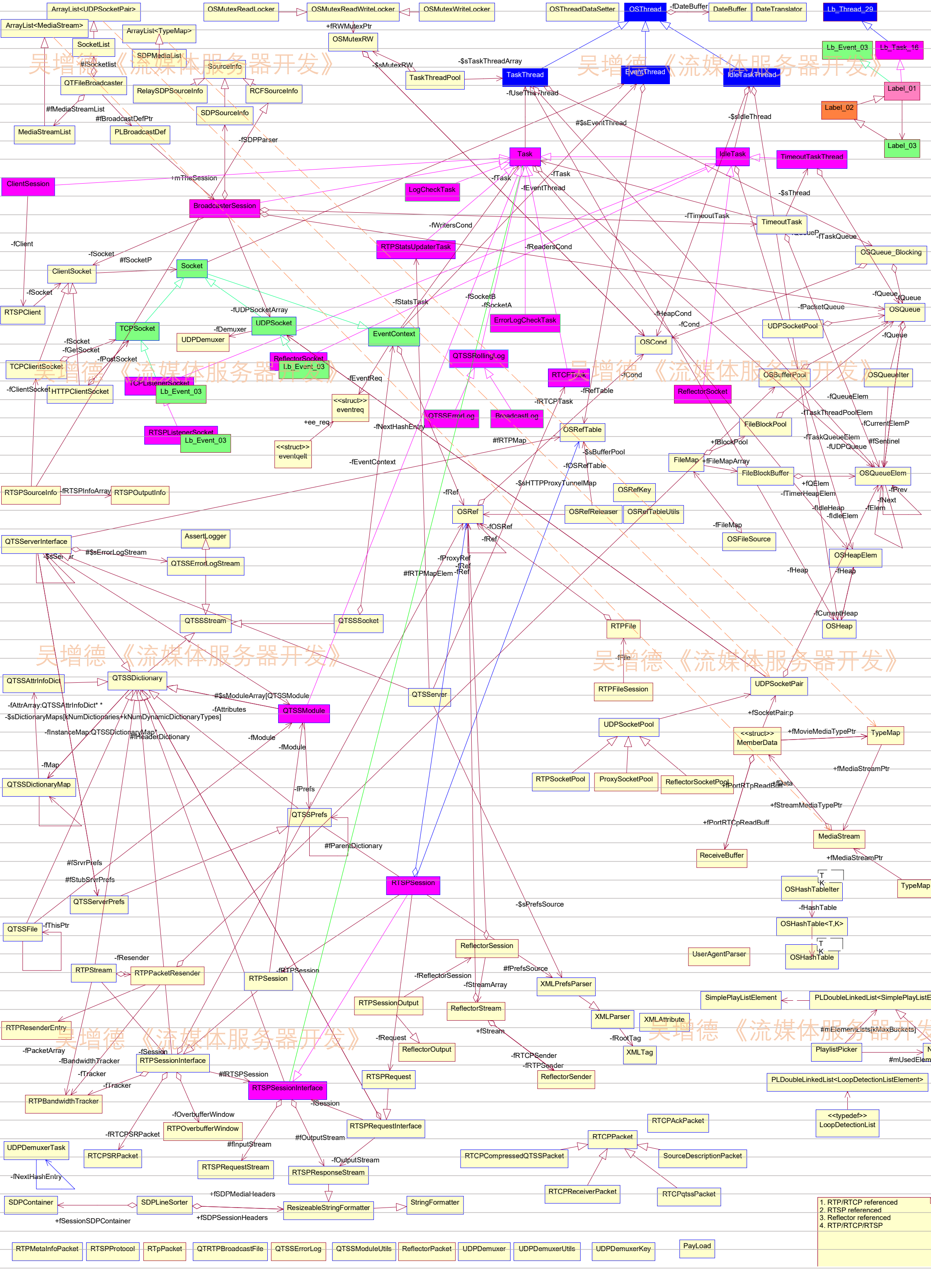


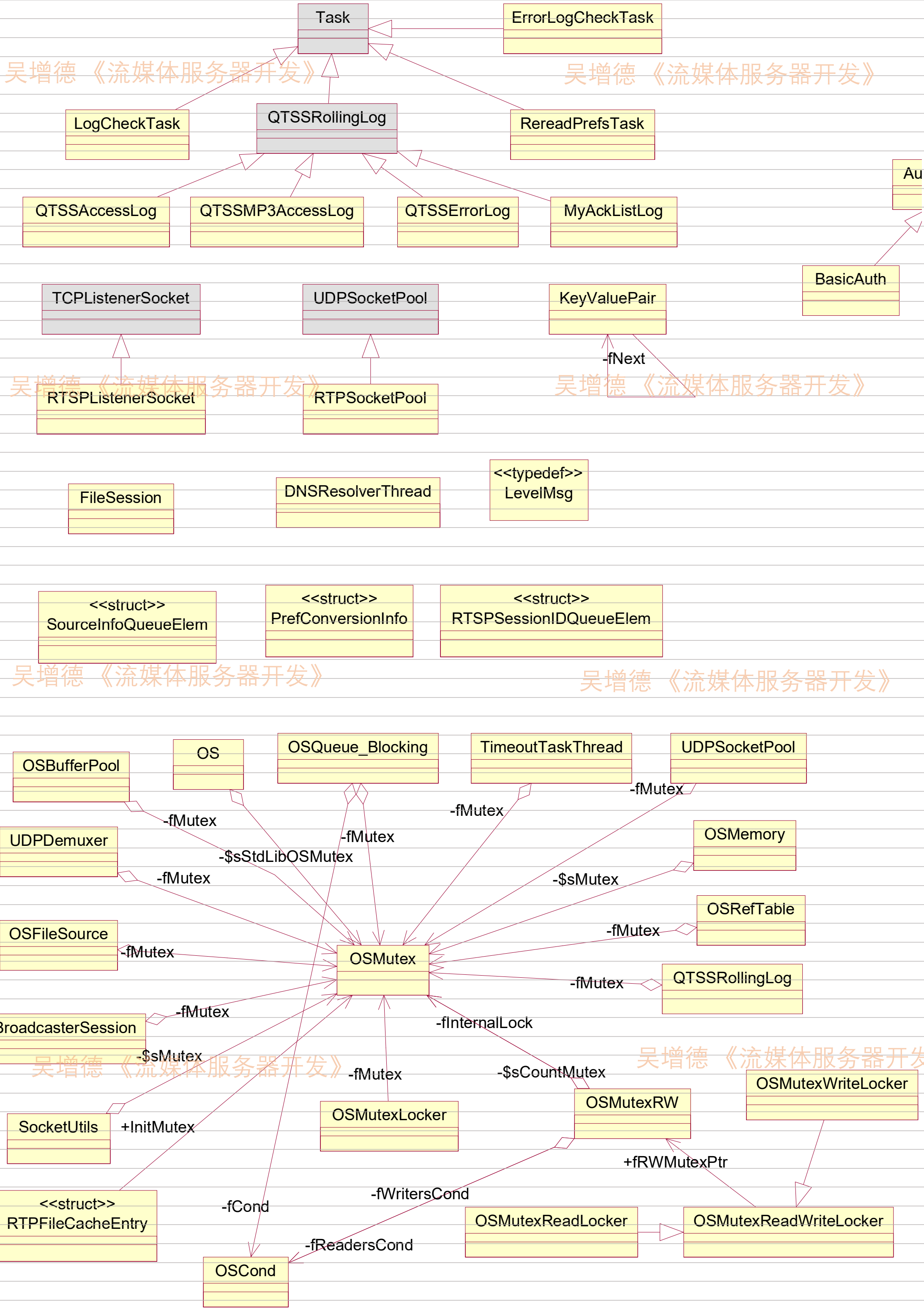


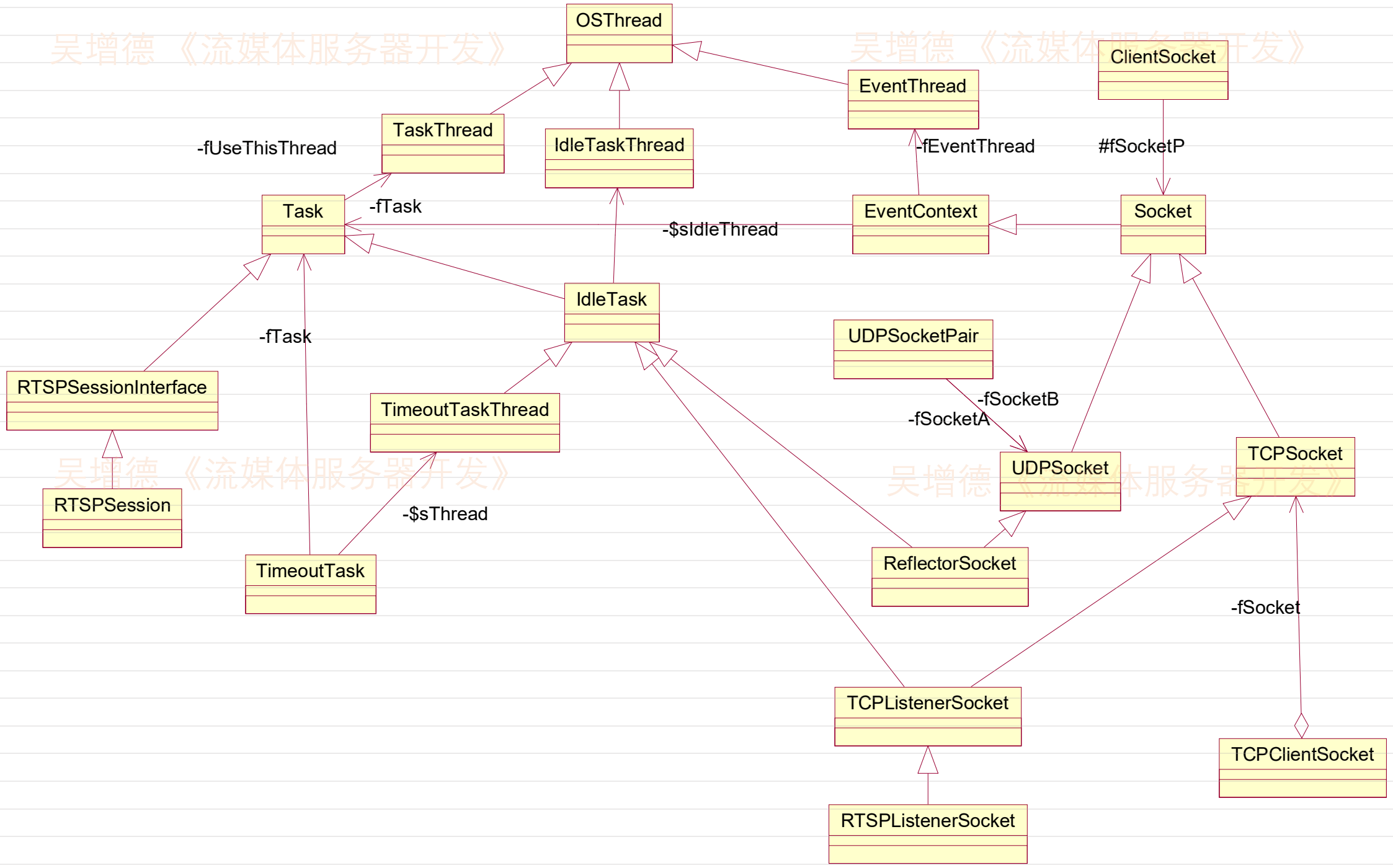




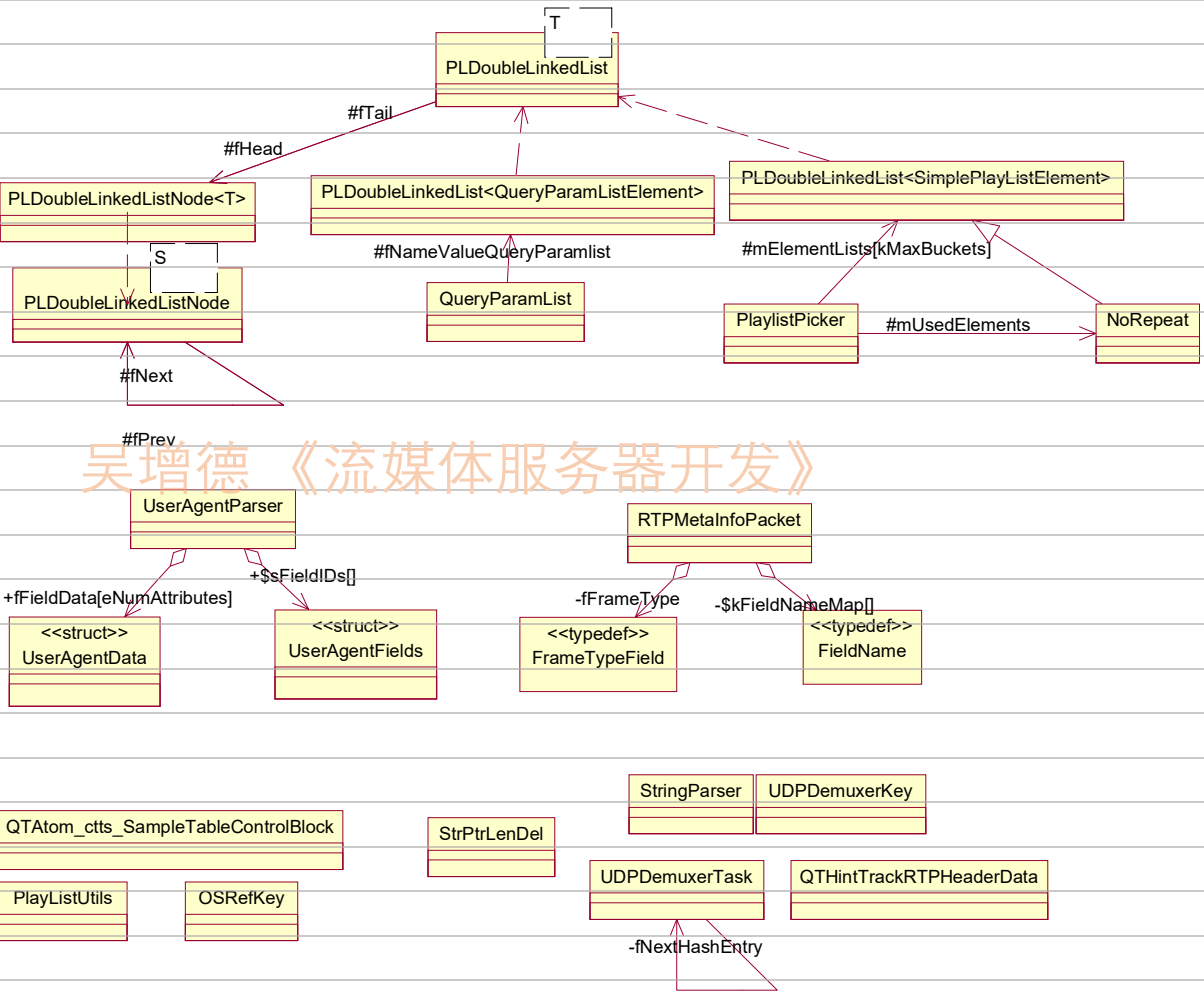
-$sDictionaryMaps[kNumDictionaries+kNumDynamicDictionaryTypes]

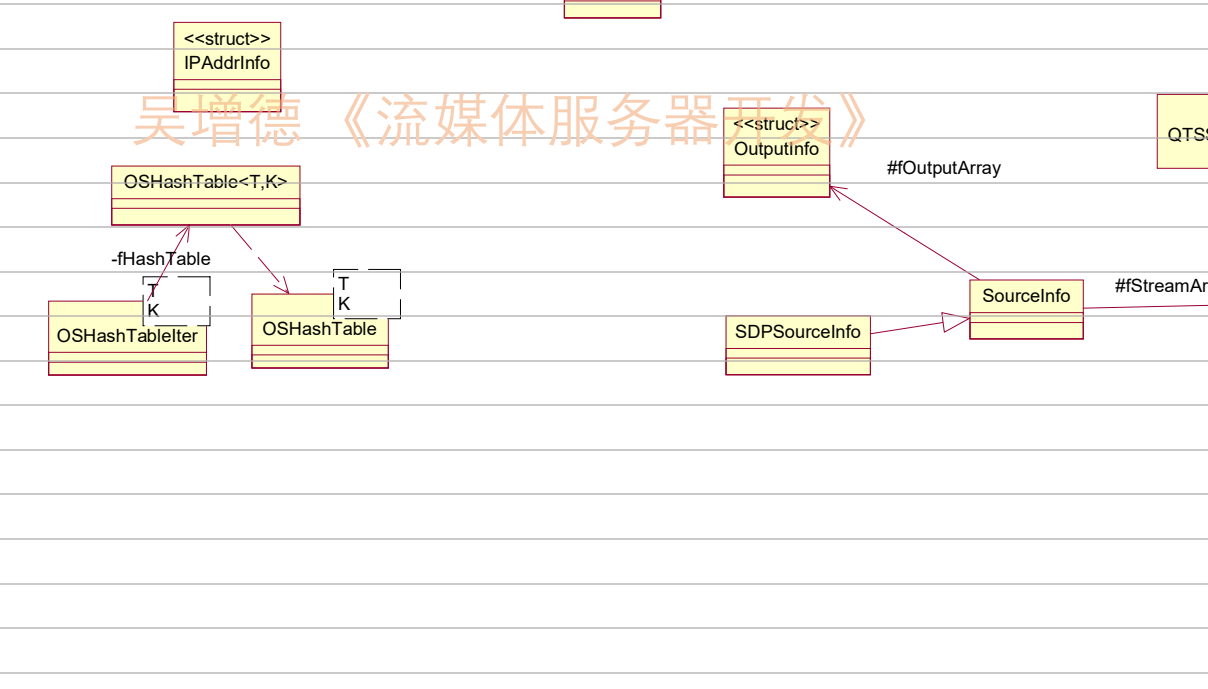


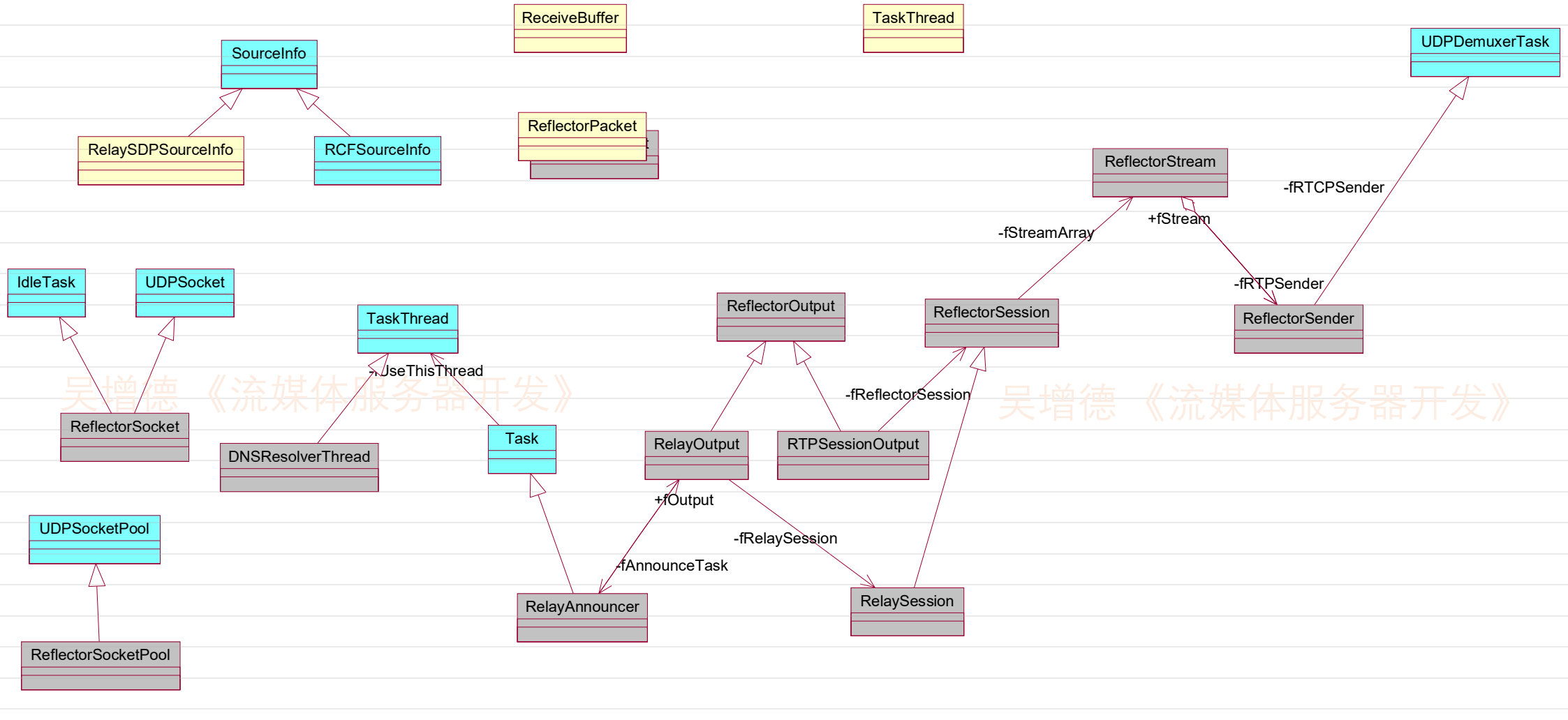




Class Diagram of Quicktime/Darwin Streaming Server -- By Zengde (Andy) Wu
http://www.linkedin.com/in/andyPlusPlus







QTSSStream

QTSSStream()
<<virtual>> ~QTSSStream()
SetTask()
GetTask()
<<virtual>> Read()
<<virtual>> Write()
<<virtual>> WriteV()
<<virtual>> Flush()
<<virtual>> Seek()
<<virtual>> Advise()
<<virtual>> RequestEvent()

QTSSDictionaryMap

<<static>> Initialize()
QTSSDictionaryMap()
~QTSSDictionaryMap()
AddAttribute()
RemoveAttribute()
UnRemoveAttribute()
CheckRemovePermission()
GetAttrInfoByName()
GetAttrInfoByID()
GetAttrInfoByIndex()
GetAttrID()
ConvertAttrIDToArrayIndex()
<<static>> IsInstanceAttrID()
GetNumAttrs()
GetNumNonRemovedAttrs()
IsPreemptiveSafe()
IsWriteable()
IsCacheable()
IsRemoved()
GetAttrFunction()
GetAttrName()
GetAttrID()
GetAttrType()
InstanceAttrsAllowed()
CompleteFunctionsAllowed()
SetAttribute()
<<static>> GetMapIndex()
<<static>> GetMap()
<<static>> CreateNewMap()

-fMap

-fInstanceMap

QTSSDictionary

QTSSDictionary()
<<virtual>> ~QTSSDictionary()
GetValue()
GetValuePtr()
GetValueAsString()
SetValue()
SetValuePtr()
CreateObjectValue()
RemoveValue()
GetValuePtr()
GetDictionaryMap()
GetInstanceDictMap()
GetNumValues()
SetNumValues()
GetValue()
GetMutex()
SetLocked()
IsLocked()
GetAttrInfoByIndex()
GetAttrInfoByName()
GetAttrInfoByID()
AddInstanceAttribute()
RemoveInstanceAttribute()
SetVal()
SetVal()
SetEmptyVal()
<<static>> Test()
<<virtual>> RemoveValueComplete()
<<virtual>> SetValueComplete()
<<virtual>> RemoveInstanceAttrComplete()
<<virtual>> CreateNewDictionary()
DeleteAttributeData()

-fAttrArray

QTSSAttrInfoDict

QTSSAttrInfoDict()
<<virtual>> ~QTSSAttrInfoDict()

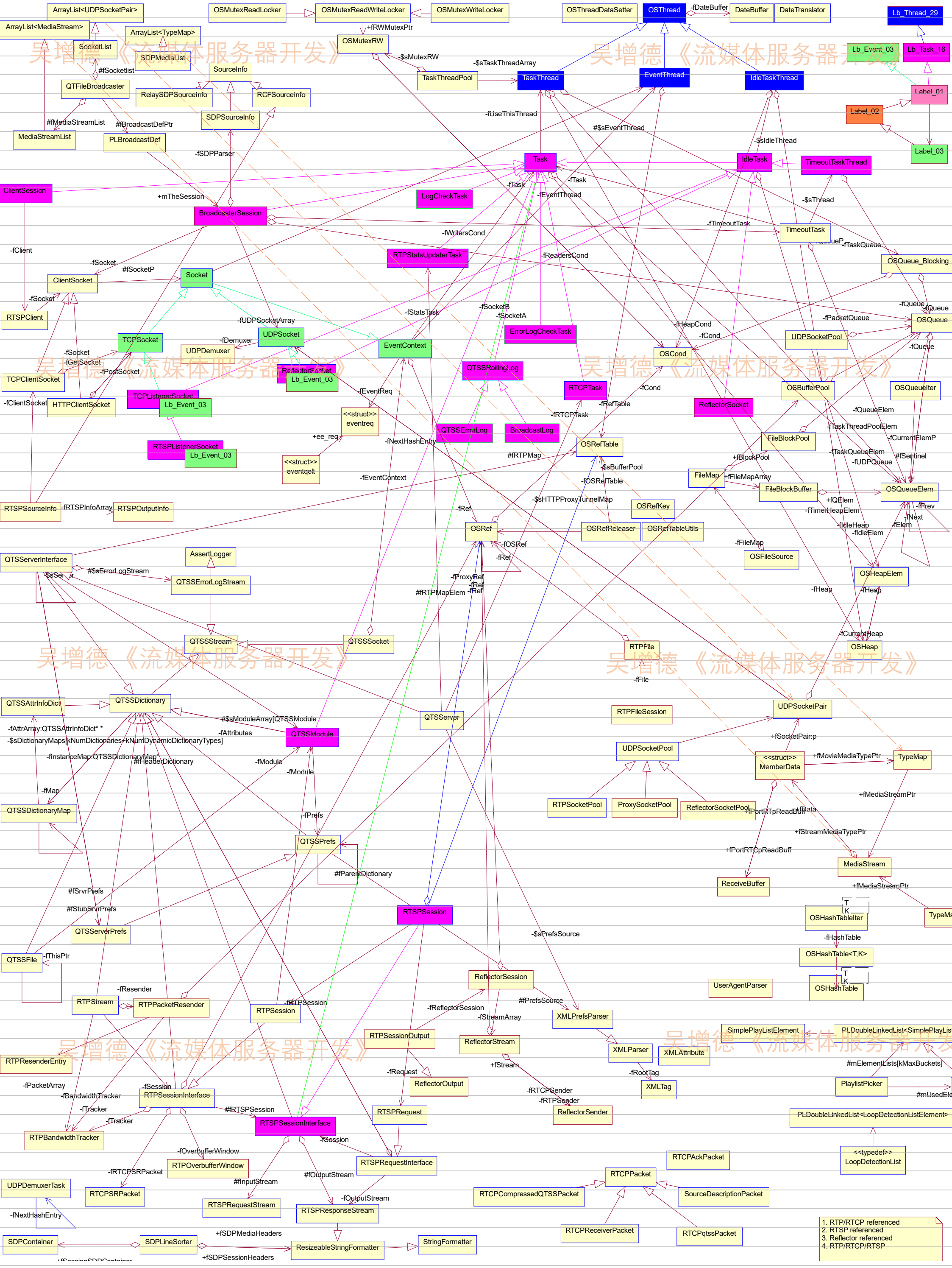
-$sDictionaryMaps[kNumDictionaries+kNumDynamicDictionaryTypes]

<<struct>>
DictValueElement

fAttributeData
fAllocatedLen
fNumAttributes
fAllocatedInternally
fIsDynamicDictionary

<<struct>>
AttrInfo

fAttrName[QTSS_MAX_ATTRIBUTE_NAME_SIZE+1] : char
fFuncPtr : QTSS_AttrFunctionPtr
fAttrDataType : QTSS_AttrDataType
fAttrPermission : QTSS_AttrPermission



Main Diagram -- By Zengde Wu http://www.linkedin.com/in/andyPlusPlus, http://www.lekuo.com