Darwin 5.5.5 代码分析文档

DarwinStreamServer 是苹果公司开发的流媒体视频服务器。我们分析的代码基于版本 5.5.5。

一、DSS启动、初始化流程

我们从Server.tproj/main.cpp入手，分析DSS加载module及和module交互的流程。

（1）、DSS在fork子进程后调用StartServer启动服务
调用OS、OSThread、Socket、SocketUtils、QTSSDictionaryMap、
QTSServerInterface、QTSServer等类的Initialize函数进行初始化。

（2）、select_startevents函数
initialize the select() implementation of the event queue.

（3）、QTSServer::Initialize函数
继续调用QTSSModule、QTSServerPres、QTSSMessages、RTSPRequestInterface、
RTSPSessionInterface、RTPSessionInterface、RTPStream、RTSPSession、
QTSSFile、QTSSUserProfile等类的Initialize函数，进行dictionary的初始化。
加载了第一个模块QTSSErrorLogModule。
this->SetDefaultIPAddr()　　*// set default IP addr & DNS name*

// begin listening，注意这里传的是false参数。
this->CreateListeners(false,fSrvrPrefs,inPortOverride)

（4）、TaskThreadPool::AddThreads(numThreads)　　*// numThreads为 1*
到这里，第一个线程创建、运行、被添加到线程池里。
在startBroadcastRTSPSession函数里，又调用AddThreads函数在线程池里添加了一个线程。

（5）、TimeoutTask::Initialize()
Start up the server's global tasks, and start listening. The timeoutTask mechanism is task
based, we therefore must do this after adding task threads. This be done before starting the sockets and server tasks.

sThread = NEW TimeoutTaskThread();
sThread->signal(Task::kStartEvent);
创建一个TimeoutTaskThread类对象，实际上这个类的名字容易产生混淆，它并不是一个线程类，而是一个基于Task类的任务类。
因为前面已经在线程池里添加了一个任务线程，所以在这里调用signal的时候，就会找到这个线程，并把事件加入到这个线程的任务队列里，等待被处理。(这时，刚才创建的线程应该也在TaskThread::Entry函数里等待事件的发生)

（6）、IdleTask::Initialize()
// 创建并启动空闲任务线程
sIdleThread = NEW IdleTaskThread();　　sIdleThread->Start();

（7）、Socket::StartThread()
// 启动Socket类的sEventThread类所对应的线程。sEventThread类在Socket::Initialize函数里创建
// 到目前为止，这已是第三个启动的线程，分别是任务线程、空闲任务线程、事务线程。

（8）、OSThread::Sleep(1000)

这里的Sleep是调用usleep来实现，**为什么这里要睡眠 1s？？？是为了等待线程的启动？？？**

（9）、sServer->InitModules(inInitialState)

初始化并加载一些模块。共加载了　　QTSSHomeDirectoryModule、QTSSRefMovieModule、QTSSFileModule、QTSSReflectorModule、
QTSSRelayModule、QTSSAccessLogModule、QTSSFlowControlModule、QTSSPosixFileSysModule、
QTSSAdminModule、QTSSMP3StreamingModule、QTSSAccessModule这些模块。

```
fSrvrPrefs = new QTSServerPrefs(sPrefsSource, true);     ... ...
fSrvrM7essages = new QTSSMessages(sMessagesSource);
QTSSModuleUtils::Initialize(fSrvrMessages, this, QTSServerInterface::GetErrorLogStream());
... ...
Add Reread Preferences Service.

// 对于那些支持initial role的模块，通过它们的CallDispatch函数来调用具体的initial函数。
this->DoInitRole();
```

（10）、sServer->StartTasks()

创建RTCPTask、RTPStatsUpdaterTask

Start listening，因为TCPListenerSocket是EventContext的继承类，所以这里实际上调用的是EventContext::RequestEvent().

（11）、sServer->SetupUDPSockets()

udp sockets are set up after the rtcp task is instantiated。

针对系统的每一个ip地址，都创建并绑定一个socket端口对（分别用于RTP data发送和RTCP data接收），并申请对这两个socket端口的监听。

注意调用CreateUDPSocketPair函数传进去的Port参数为 0，所以在通过Darwin播放静态多媒体文件时，不论是同一个媒体文件的音频、视频流还是同时播放的多个媒体文件，都是这两个 socket端口来完成RTCP、RTP数据的处理。

（12）、CleanPid(true); WritePid(!inDontFork); doneStartingUp = true;

在/var/run下写pid文件

（13）、sServer->SwitchPersonality()

switch to run user and group ID

执行setgid、setuid函数

（14）、RunServer()

由一个大循环构成：

```
while((theServerState != qtssShuttingDownState) && (theServerState != qtssFatalErrorState))
{
        OSThread::Sleep(1000);
        if(sStatusUpdateInterval)          // 周期性更新状态，当带有-D参数时，为 3。
        {
                通过PrintHeader打印标题行
RTP-Conns  RTSP-Conns  HTTP-Conns  kBits/Sec  Pkts/Sec RTP-Playing AvgDelay CurMaxDelay MaxDelay AvgQuality NumThinned Time
RTP-Conns RTSP-Conns HTTP-Conns  kBits/Sec   Pkts/Sec    TotConn     TotBytes   TotPktsLost  Time
                通过DebugLevel_1、PrintStatus打印每字段的值。
                每个字段对应的变量分别为：
                RTP-Conns:     fNumRTPSessions;
                RTSP-Conns:    fNumRTPSessions;
                HTTP-Conns:    fNumRTSPHTTPSessions;
                kBits/Sec:     fCurrentRTPBandwidthInBits;
```

```
                Pkts/Sec:       fRTPPacketsPerSecond;
                RTP-Playing:    fNumRTPPlayingSessions;
                // sLastDebugPacket即为上次的fTotalRTPPackets。
                AvgDelay:       fTotalLate/(fTotalRTPPackets  -  sLastDebugPackets)
                CurMaxDelay:    fCurrentMaxLate;
                MaxDelay:       fMaxLate;
                // sLastDebugTotalQuality即为上次的fTotalQuality。
                AvgQuality:     (fTotalQuality-sLastDebugTotalQuality)/(fTotalRTPPackets - sLastDebugPackets)
                NumThinned:     fNumThinned;

                TotConn:        fTotalRTPSessions;
                TotBytes:       fTotalRTPBytes;
                TotPktsLost:    fTotalRTPPacketsLost

                如果接收到SigInt或者SigTerm，则终止sServer。
        }
```

二、几大部分

（一）、模块

- DSS 利用模块来响应请求和处理任务。DSS 的模块分为三种类型：

（1）、Content-Managing Modules

The content-managing modules manage RTSP requests and responses related to media sources, such as a file or a broadcast. Each module is responsible for interpreting the client's request, reading and parsing their supported files or network source, and responding with RTSP and RTP. In some cases, such as the mp3 streaming module, the module uses HTTP.

The content-managing modules are QTSSFileModule, QTSSReflectorModule, QTSSRelayModule, and QTSSMP3StreamingModule.

（2）、Server-Support Modules

The server-support modules performserver data gathering and logging functions. The server-support modules are QTSSErrorLogModule, QTSSAccessLogModule, QTSSWebStatsModule, QTSSWebDebugModule, QTSSAdminModule, and QTSSPOSIXFileSystemModule.

（3）、Access Control Modules

The access control modules provide authentication and authorization functions as well as URL path manipulation.

The access control modules are QTSSAccessModule, QTSSHomeDirectoryModule, QTSSHttpFileModule, and QTSSSpamDefenseModule.

- 几个相关类的定义

一、QTSSModule类

各个Module类均以QTSSModule类为基类。

（1）、构建函数

```
QTSSModule::QTSSModule(char* inName, char* inPath)
:
QTSSDictionary(QTSSDictionaryMap::GetMap(QTSSDictionaryMap::kModuleDictIndex)), ... ...
{
        如果模块以库文件的形式保存在disk上，利用OSCodeFragment类来处理
        另外，QTSSModule本身是QTSSDictionary的继承类，它还有一个QTSSDictionary类对象的指
        针成员
        调用SetVal、SetValue进行一些属性的设置
}
```

- 模块加载过程

一、以加载QTSSErrorLogModule为例

在创建一个QTSSModule类对象后，调用该类的SetupModule成员函数，并调用AddModule函数和调用

QTSServer的BuildModuleRoleArrays函数。

注意：DSS还提供了一个OSCodeFragment类，来处理以库文件的形式保存在disk的模块。

（1）SetupModule成员函数

传给SetupModule成员函数的两个形参分别是QTSS_CallbacksPtr（函数指针）、QTSS_MainEntryPointPtr（具体模块的进入函数指针，对于QTSSErrorLogModule类，这个函数是QTSSErrorLogModule_Main）

调用QTSSErrorLogModule_Main

```
_____stublibrary_main(inPrivateArgs, QTSSErrorLogModuleDispatch)
_____sCallbacks = theArgs->inCallbacks;
    sErrorLogStream = theArgs->inErrorLogStream;
    // Send requested information back to the server
    theArgs->outStubLibraryVersion = QTSS_API_VERSION;
    theArgs->outDispatchFunction = inDispatchFunc;
```

注：这里的 sCallbacks、sErrorLogStream 分别是在 QTSS_Private.cpp 里定义的静态全局量。theArgs->inCallbacks 为 SetupModule 的实参（即 QTSServer 类的 sCallbacks 成员），而 inDispatchFunc 为具体模块传进来的实参 QTSSErrorLogModuleDispatch。

将fDispatchFunc设置为thePrivateArgs.outDispatchFunction，fDispatchFunc为QTSSModule的私有类成员。这样DSS可以通过具体模块的基类QTSSModule的这个私有成员来让具体模块进行分发处理。

调用QTSServerInterface::LogError函数进行log功能。

（2）AddModule函数

传给AddModule函数的实参为要加载的具体模块。

```
Prepare to invoke the module's Register role. Setup the Register param block
QTSS_ModuleState theModuleState;
theModuleState.curModule = inModule;
theModuleState.curRole = QTSS_Register_Role;
theModuleState.curTask = NULL;
OSThread::SetMainThreadData(&theModuleState);

inModule->CallDispatch(QTSS_Register_Role, &theRegParams);
        (fDispatchFunc)(inRole, inParams);
```

对于 QTSSErrorLogModule 模块即为 QTSSErrorLogModuleDispatch 函数，该函数根据不同的 Role 调用模块具体的处理函数，对于 QTSS_Register_Role 调用 Register 函数。Register 函数通过调用 QTSS_AddRole 来告知 DSS 模块支持的 Role。

```
// Update the module name to reflect what was returned from the register role
inModule->SetValue(qtssModName, 0, theRegParams.regParams.outModuleName,
        ::strlen(theRegParams.regParams.outModuleName), false);

// Give the module object a prefs dictionary. Instance attributes are allowed for these
        objects.
QTSSPrefs* thePrefs = NEW QTSSPrefs( sPrefsSource, inModule->GetValue(qtssModName),
        QTSSDictionaryMap::GetMap(QTSSDictionaryMap::kModulePrefsDictIndex), true);
thePrefs->RereadPreferences();
inModule->SetPrefsDict(thePrefs);

// Add this module to the array of module (dictionaries)
UInt32 theNumModules = this->GetNumValues(qtssSvrModuleObjects);
QTSS_Error theErr = this->SetValue(qtssSvrModuleObjects, theNumModules, &inModule,
        sizeof(QTSSModule*), QTSSDictionary::kDontObeyReadOnly);

// Add this module to the module queue
sModuleQueue.EnQueue(inModule->GetQueueElem());
```

（3）BuildModuleRoleArrays函数

根据QTSSModule的fRoleArray数组的Role支持情况，在QTSServerInterface::sNumModulesInRole里做记录，并在sModuleArray数组里保存相关Role的所有QTSSModule指针。

这样，QTSServer就可以知道目前有哪些模块支持哪些Role。

到此，一个Module已经添加到QTSServer里。

二、以加载QTSSReflectorModule为例

在创建一个QTSSModule类对象后，调用该类的SetupModule成员函数，并调用AddModule函数和调用QTSServer的BuildModuleRoleArrays函数。

```
QTSSModule* theReflectorModule = new QTSSModule("QTSSReflectorModule");
(void)theReflectorModule->SetupModule(&sCallbacks, &QTSSReflectorModule_Main);
        调用 QTSSReflectorModule_Main函数，
        将fDispatchFunc设置为
        thePrivateArgs.outDispatchFunction(QTSSReflectorModuleDispatch)，fDispatchFunc为
        QTSSModule的私有类成员。这样DSS可以通过具体模块的基类QTSSModule的这个私有成员来
        让具体模块进行分发处理。
        调用QTSServerInterface::LogError函数进行log功能
           for(UInt32 x=0;x<QTSServerInterface::GetNumModulesInRole(QTSSModule::kErrorLogRole); x++)
                QTSServerInterface::GetModule(QTSSModule::kErrorLogRole,x)-
                                >CallDispatch(QTSS_ErrorLog_Role,&theParams);
                // 因为只有QTSSErrorLogModule注册kErrorLogRole，实际上会调用这个类的
                // LogError函数。
                If this is a fatal error, 调用SetValue set the proper attribute in the
                        RTSPServer dictionary。

(void)AddModule(theReflectorModule);
```

后面同上面所说的QTSSErrorLogModule加载情况。

- 一些模块的介绍

一、QTSSRTPFileModule

Content source module that uses the QTFileLib to serve Hinted QuickTime files to clients.

支持TSS_Initialize_Role、QTSS_RTSPPreProcessor_Role、QTSS_ClientSessionClosing_Role、QTSS_RereadPrefs_Role。Dispatch函数里也提供了QTSS_RTPSendPackets_Role的处理，直接通过模块来调用这个Role的处理。

在initialize函数里，通过SetupSupportedMethods登记支持的方法：DESCRIBE, SETUP, PLAY, PAUSE, and TEARDOWN。

！！这个模块并没有被加载！！

（1）、ProcessRTSPRequest

QTSS_RTSPPreProcessor_Role的处理函数。

根据请求内容，分别做出处理：

```
Describe:     DoDescribe(inParams);
Setup:        DoSetup(inParams);
Play:         DoPlay(inParams);
Teardown:     // Tell the server that this session should be killed, and send a TEARDOWN response
              QTSS_Teardown(...);  QTSS_SendStandardRTSPResponse();
Pause:        QTSS_Pause();        QTSS_SendStandardRTSPResponse();
```

（2）、DoDescribe

Check and see if this is a request we should handle. We handle all requests with URLs that end in a '.rtp'

Get the FileSession for this DESCRIBE, if any.

```
if(theFile != NULL)
```

```
{
    // There is already a file for this session. This can happen if there are multiple
    // DESCRIBES, or a DESCRIBE has been issued with a Session ID, or some such thing.
    if ( !theFullPath.Equal( *theFile->fFile.GetMoviePath() ) )
    {
        delete theFile;             theFile = NULL;
        QTSS_SetValue(inParamBlock->inClientSession, sFileSessionAttr, 0, &theFile,
        sizeof(theFile));
    }
}
if(theFile == NULL)
{   // 创建FileSession类对象，调用theFile->fFile.Initialize(theFullPath, 8);
    CreateRTPFileSession(inParamBlock, theFullPath, &theFile);
    QTSS_SetValue(inParamBlock->inClientSession, sFileSessionAttr, 0, &theFile, sizeof(theFile));
}
```

二、QTSSFileModule模块

Content source module that uses the QTFileLib to serve Hinted QuickTime files to clients.
注册了QTSS_Initialize_Role、QTSS_RTSPRequest_Role、QTSS_ClientSessionClosing_Role、QTSS_RereadPrefs_Role的处理。在Dispatch函数里还添加了QTSS_RTPSendPackets_Role的处理。
在initialize函数里，通过SetupSupportedMethods登记支持的方法：DESCRIBE, SETUP, PLAY, PAUSE, and TEARDOWN。

（1）、ProcessRTSPRequest()
QTSS_RTSPRequest_Role的处理函数。
根据请求内容，分别做出处理：

```
Describe:       DoDescribe(inParams);
Setup:          DoSetup(inParams);
Play:           DoPlay(inParams);
Teardown:       // Tell the server that this session should be killed, and send a TEARDOWN response
                QTSS_Teardown(...);   QTSS_SendStandardRTSPResponse();
Pause:          QTSS_Pause();         QTSS_SendStandardRTSPResponse();
```

（2）、DoDescribe()

```
if(isSDP(inParamBlock))             // sdp file
{       ... ...
        QTSS_GetValuePtr(inParamBlock->inRTSPRequest, qtssRTSPReqFilePath, 0,(void**)&pathStr.Ptr,
&pathStr.Len);
QTSSModuleUtils::SendErrorResponse(inParamBlock->inRTSPRequest,qtssClientNotFound,sNoSDPFileFoundErr,&pathStr);
        ... ...return err;
}
... ...
if (theFile != NULL)
{
    // There is already a file for this session. This can happen if there are multiple
    // DESCRIBES, or a DESCRIBE has been issued with a Session ID, or some such thing.
    moviePath(theFile->fFile.GetMoviePath());
    if(!requestPath.Equal(moviePath))
    {
        DeleteFileSession(theFile); theFile = NULL;
        QTSS_SetValue(inParamBlock->inClientSession, sFileSessionAttr, 0, &theFile, sizeof(theFile));
    }
}
if (theFile == NULL)
{
        // 创建FileSession类对象，调用theFile->fFile.Initialize(inPath,);
```

```
            // theFile->fFile为QTRTPFile类型的对象。
            // 通过QTRTPFile、QTFile、QTSSFile等类对象来完成媒体文件的解析
            CreateQTRTPFile(inParamBlock, thePath.GetObject(), &theFile);
            QTSS_SetValue(inParamBlock->inClientSession, sFileSessionAttr, 0, &theFile,
sizeof(theFile));
        }
        // replace the sacred character we have trodden on in order to truncate the path.
        // 文件名添加.sdp后缀
        thePath.GetObject()[thePathLen  -  sSDPSuffix.Len] = sSDPSuffix.Ptr[0];
        if (sEnableMovieFileSDP)     // 在配置文件里enable_movie_file_sdp为false，为什么也能在客
                                     // 户端的播放器里访问sdp文件？？？
        {
            // Check to see if there is an sdp file, if so, return that file instead of the
            // built-in sdp.
            QTSSModuleUtils::ReadEntireFile(thePath.GetObject(), &theSDPData);
        }
```

后续是sdp信息的处理（**这里实现什么？？？**）如果配置文件中的record_movie_file_sdp为true，则会生成一个sdp文件。

```
        // now parse the movie media sdp data. We need to do this in order to extract payload
        // information. The SDP parser object will not take responsibility of the memory (one
        // exception... see above)
        // 注意在前面已经调用了：
        // theSDPData.Ptr = theFile->fFile.GetSDPFile(&sdpLen);  theSDPData.Len = sdpLen;
        // 从媒体文件里获取sdp信息。
         theFile->fSDPSource.Parse(theSDPData.Ptr, theSDPData.Len);
```

（3）、DoSetup()

```
        if (isSDP(inParamBlock))
        {       ... ...}
        QTSS_GetValue(inParamBlock->inClientSession, sFileSessionAttr, 0, (void*)&theFile,
&theLen);
        if((theErr != QTSS_NoErr) || (theLen != sizeof(FileSession*)))
        {       ... ...}
        QTSS_GetValueAsString(inParamBlock->inRTSPRequest, qtssRTSPReqFileDigit, 0, &theDigitStr);
        // 比如客户端发送一个
        // “SETUP rtsp://192.168.2.163:554/sample_100kbit.mp4/trackID=3 RTSP/1.0\r\n”请求，则
        // theTrackID等于3。
        Uint32 theTrackID = ::strtol(theDigitStr, NULL, 10);
        // 调用QTRTPFile::AddTrack
        theFile->fFile.AddTrack(theTrackID, true);

        // Before setting up this track, check to see if there is an If-Modified-Since date. If
        // there is, and the content hasn't been modified, then just return a 304 Not Modified
        这里所起的作用是什么？？？
        // Find the payload for this track ID(if applicable)
        根据theFile->fSDPSource的streaminfo，设置thePayload、thePayloadType、bufferDelay
        // Create a new RTP stream
        // 实际上是调用RTPSession::AddStream()
        QTSS_AddRTPStream(inParamBlock->inClientSession, inParamBlock->inRTSPRequest, &newStream, 0);
        // Set the payload type, payload name & timescale of this track
        // Set the number of quality levels. Allow up to 6
        调用QTSS_SetValue进行设置。
        // Get the SSRC of this track
        // give the file some info it needs.
        根据TrackID，将对应的trackEntry->SSRC设置为上面获得的SSRC，将trackEntry->Cookie1
```

Cookie2分别设为newStream、thePayloadType。

```
theErr = QTSS_GetValuePtr(inParamBlock->inRTSPHeaders, qtssXRTPMetaInfoHeader, 0,
                          (void**)&theHeader.Ptr, &theHeader.Len);
if(theErr == QTSS_NoErr)
{      ... ...}

// Our array has now been updated to reflect the fields requested by the client. send the
// setup response
```

调用QTSS_AppendRTSPHeader

```
QTSS_SendStandardRTSPResponse(inParamBlock->inRTSPRequest, newStream, 0);
```

（4）、SetupCacheBuffers()

通过QTSS_GetValue获取playCount值。

```
// increments num buffers after initialization so do only once per session
// Allocate函数以及OSFileSource::ReadFromCache/ReadFromDisk函数的分析待续
if (sEnableSharedBuffers && playCount == 1)
        (*theFile)->fFile.AllocateSharedBuffers(sSharedBufferUnitKSize, sSharedBufferInc,
                                sSharedBufferUnitSize,sSharedBufferMaxUnits);
if (sEnablePrivateBuffers)  // reinitializes buffers to current location so do every
time
        (*theFile)->fFile.AllocatePrivateBuffers(sSharedBufferUnitKSize, sPrivateBufferUnitSize,
                                        sPrivateBufferMaxUnits);
playCount ++;
QTSS_SetValue(inParamBlock->inClientSession, sFileSessionPlayCountAttrID, 0, &playCount,
```

（5）、DoPlay()

```
if (isSDP(inParamBlock))
{      ... ...}
QTSS_GetValue(inParamBlock->inClientSession, sFileSessionAttr, 0, (void*)&theFile,
&theLen);
SetupCacheBuffers(inParamBlock, theFile);
```

调用一系列的设置工作QTSS_SetValue。。。

```
// Tell the server to start playing this movie. We do want it to send RTCP SRs, but we
// DON'T want it to write the RTP header
// 调用RTPSession::Play
QTSS_Play(inParamBlock->inClientSession, inParamBlock->inRTSPRequest, qtssPlayFlagsSendRTCP);
```

准备RTSP回复内容，并调用QTSS_SendStandardRTSPResponse

（6）、SendPackets()

QTSS_RTPSendPackets_Role的处理函数，在RTPSession::Run函数里会调用这个Role的注册函数。

```
theLastPacketTrack = (*theFile)->fFile.GetLastPacketTrack();
while (true) {
        if ((*theFile)->fPacketStruct.packetData == NULL)
        {
                // 寻找要传输的包，theTransmitTime为发送时间
                theTransmitTime = (*theFile)->fFile.GetNextPacket(
                        (char**)&(*theFile)->fPacketStruct.packetData, &(*theFile)->fNextPacketLen);
                // 刚找到的
                theLastPacketTrack = (*theFile)->fFile.GetLastPacketTrack();
                theStream = (QTSS_Object)theLastPacketTrack->Cookie1;
                // Check to see if we should stop playing now
                if (((*theFile)->fStopTime !=-1) && (theTransmitTime > (*theFile)-
>fStopTime))
                if (((*theFile)->fStopTrackID !=0) && ((*theFile)->fStopTrackID == theLastPacketTrack-
>TrackID)
                        && (theLastPacketTrack->HTCB->fCurrentPacketNumber > (*theFile)->fStopPN))
```

```
                {       // We should indeed stop playing
                        ... ... inParams->outNextPacketTime = qtssDontCallSendPacketsAgain;
                        ... ... return QTSS_NoErr;
                }
                // Find out what our play speed is. Send packets out at the specified rate,
                // and do so by altering the transmit time of the packet based on the
                // Speed rate.
                调整theTransmitTime
        }
    // 如果还没有数据
    if ((*theFile)->fPacketStruct.packetData == NULL)
    {       inParams->outNextPacketTime = qtssDontCallSendPacketsAgain; return QTSS_NoERR; }
    // 发送数据
    // If the stream is video, we need to make sure that QTRTPFile knows what quality level
    // we're at
    设置quality level。
    adjust the timestamp so it reflects paused time.
    // 调用RTPStream::Write函数。RTPStream::Write返回EAGAIN时，QTSS_Write返回QTSS_WouldBlock
    theErr = QTSS_Write(theStream, &(*theFile)->fPacketStruct, (*theFile)-
>fNextPacketLen,NULL,theFlags);
    // 设置inParams->outNextPacketTime，RTPSession::Run函数根据outNextPacketTime，决定返回值。
    if ( theErr == QTSS_WouldBlock )
    {       // reset the packet time stamp so we adjust it again when we really do send it
            if (currentTimeStamp != pauseTimeStamp)
                    SetPacketTimeStamp(currentTimeStamp, packetDataPtr);
            // In the case of a QTSS_WouldBlock error, the packetTransmitTime field of the
            // packet struct will be set to the time to wakeup, or -1 if not known.
            // If the time to wakeup is not given by the server, just give a fixed guess interval
            if ((*theFile)->fPacketStruct.suggestedWakeupTime == -1)
                    inParams->outNextPacketTime = sFlowControlProbeInterval;
            else
                    inParams->outNextPacketTime=(*theFile)->fPacketStruct.suggestedWakeupTime-
                                                    inParams->inCurrentTime;
    } else {
            QTSS_SetValue(theStream, sRTPStreamLastSentPacketSeqNumAttrID, 0, &curSeqNum,
                                    sizeof(curSeqNum));
            (*theFile)->fPacketStruct.packetData = NULL;
    }
```

三、QTSSPosixFileSysModule模块

支持QTSS_OpenFile_Role，实际上在该模块的Dispatch函数里同时支持QTSS_Advise_Role、QTSS_ReadFile_Role、QTSS_CloseFile_Role、QTSS_RequestEventFile_Role的处理，这些Role是直接通过模块来调用的。

（1）、OpenFile

```
// 创建OSFileSource对象，该类实现底层的文件操作。
theFileSource = NEW OSFileSource(inParams->inPath);
获取文件长度、修改时间等信息。
// Add this new file source object to the file object
QTSS_SetValue(inParams->inFileObject, sOSFileSourceAttr, 0, &theFileSource, sizeof(theFileSource));
// 如果是异步I/O, at this point we should set up the EventContext
if (inParams->inFlags & qtssOpenFileAsync)
{
        theEventContext = NEW EventContext(EventContext::kInvalidFileDesc,
Socket::GetEventThread());
        theEventContext->InitNonBlocking(theFileSource->GetFD());
        QTSS_SetValue(inParams->inFileObject, sEventContextAttr, 0, &theEventContext, sizeof(theEventContext));
```

```
}
Set up the other attribute values in the file object
```

（2）、ReadFile
通过QTSS_GetValuePtr获得OSFileSource对象，调用该对象的Read函数

四、QTSSReflectorModule模块

支持QTSS_Initialize_Role、QTSS_Shutdown_Role、QTSS_RTSPPreProcessor_Role、QTSS_ClientSessionClosing_Role、QTSS_RTSPIncomingData_Role、QTSS_RTSPAuthorize_Role、QTSS_RereadPrefs_Role、QTSS_RTSPRoute_Role

（1）、Register
调用QTSS_AddRole注册支持的Role的处理函数。
调用一系列的QTSS_AddStaticAttribute、QTSS_IDForAttr函数。
调用ReflectorStream::Register函数、RTPSessionOutput::Register函数。

（2）、Initialize

```
QTSSModuleUtils::Initialize(inParams->inMessages, inParams->inServer, inParams->inErrorLogStream);
QTAccessFile::Initialize();         sSessionMap = NEW OSRefTable();
sServerPrefs = inParams->inPrefs;   sServer = inParams->inServer;
sPrefs = QTSSModuleUtils::GetModulePrefsObject(inParams->inModule);
ReflectorStream::Initialize(sPrefs);       ReflectorSession::Initialize();

Report to the server that this module handles DESCRIBE, SETUP, PLAY, PAUSE, and TEARDOWN

RereadPrefs();
```

（3）、FindOrCreateSession

```
// 注意：sSessionMap是一个静态的全局变量，这里Resolve的参数是sdp文件的路径。
// 也就是说，在同时开启多个窗口播放同一个sdp文件时，都是使用同一个ReflectorSession对象，
// FindOrCreateSession也不会调用SetupReflectorSession函数，即ReflectorStream、
// ReflectorSocket等对象也不会被再次创建。从逻辑上理解也确实如此，因为只需要一套对象和
// Mp4live这类系统打交道。
// 但是在这种情况下，会重新创建RTSPSession、RTPSession对象。这样在
// QTSSReflectorModule::DoSetup函数里，会再次创建RTPSessionOutput对象，并添加到
// ReflectorStream的fOutputArray数组里。同时也会调用QTSS_AddRTPStream函数。
OSRef* theSessionRef = sSessionMap->Resolve(inPath);
if (theSessionRef == NULL)
{
        // If this URL doesn't already have a reflector session, we must make a new one.
        // The first step is to create an SDPSourceInfo object.
        // 读取sdp文件
        if(inData == NULL)
        {
                QTSSModuleUtils::ReadEntireFile(inPath->Ptr, &theFileDeleteData);
                theFileData = theFileDeleteData;
        } else theFileData = *inData;
        OSCharArrayDeleter fileDataDeleter(theFileDeleteData.Ptr);
        // 根据读取的sdp文件信息，创建SDPSourceInfo对象，该对象是SourceInfo的继承类。
        // 在构建函数里，调用Parse函数，形成了描述sdp文件内容的StreamInfo数据结构。
        SDPSourceInfo* theInfo = NEW SDPSourceInfo(theFileData.Ptr, theFileData.Len);
        if(!theInfo->IsReflectable() || !InfoPortsOK(inParams, theInfo, inPath))
        {       delete theInfo;             return NULL;  }
        if(!AllowBroadcast(inParams->inRTSPRequest))
        {
```

```
                SendErrorResponseWithMessage(inParams->inRTSPRequest, qtssClientForbidden,
                                             &sBroadcastNotAllowed);
                return NULL;
            }
            // Setup a ReflectorSession and bind the sockets. If we are negotiating, make sure
            // to let the session know that this is a Push Session so ports may be modified.
            // 创建并配置ReflectorSession对象
            theSession = NEW ReflectorSession(inPath);
            // buffer the incoming streams for clients
            theSession->SetHasBufferedStreams(true);
            // SetupReflectorSession stores theInfo in theSession so DONT delete the Info if
            // we fail here, leave it alone. deleting the session will delete the info.
            theSession->SetupReflectorSession(theInfo, inParams,
                        theSetupFlag,sOneSSRCPerStream, sTimeoutSSRCSecs);
            sSessionMap->Register(theSession->GetRef());
        }
```

（4）、ProcessRTSPRequest

QTSS_RTSPPreProcessor_Role的处理函数。

针对RTSP请求的内容，做不同的处理：

AnnounceMethod: DoAnnounce()

DescribeMethod: DoDescribe()

SetupMethod: DoSetup()

PlayMethod: DoPlay()

TeardownMethod: QTSS_Teardown()、QTSS_SendStandardRTSPResponse()

PauseMethod: QTSS_Pause()、QTSS_SendStandardRTSPResponse()

（5）、DoDescribe

```
        // 调用FindOrCreateSession，创建ReflectorSession对象
        ReflectorSession* theSession = DoSessionSetup(inParams, qtssRTSPReqFilePath, false, NULL,
                                        &theFilepath);
```

If there already was an RTPSessionOutput attached to this Client Session, destroy it.

// Send the DESCRIBE response

Process SDP to remove connection info and add track IDs, port info, and default c= line

Clean up missing required SDP lines

Check the headers

Put SDP header lines in correct order

Write the SDP

调用SendDescribeResponse

（6）、DoSetup

```
        ... ...
RTPSessionOutput** theOutput = NULL;
theErr = QTSS_GetValuePtr(inParams->inClientSession, sOutputAttr, 0, (void**)&theOutput,
                        &theLen);
        // 注意对于一个新的播放链接来说，RTSPSession、RTPSession都是新创建的对象，所以需要重新
        // 创建RTPSessionOutput对象
if (theLen != sizeof(RTPSessionOutput)) {
        // Check to see if we have a RTPSessionOutput for this Client Session. If we don't we
        // should make one
        ... ...
        RTPSessionOutput *theNewOutput = NEW RTPSessionOutput(inParams->inClientSession,
                        theSession, sServerPrefs, sStreamCookieAttr);
        // 针对ReflectorSession的每一个ReflectorStream，都将theNewOutput添加进
```

```
// fOutputArray[bucket][y]
theSession->AddOutput(theNewOutput, true);
// If this is an incoming data session, skip everything having to do with setting up a
// new RTP Stream.
if (isPush)
{     ... ...}
// Get info about this trackID
// 音视频流有不同的trackID，不同的PayloadName、PayloadType
theStreamInfo = theSession->GetSourceInfo()->GetStreamInfoByTrackID(theTrackID);
thePayloadName = &theStreamInfo->fPayloadName;
thePayloadType = theStreamInfo->fPayloadType;
... ...
// 实际上调用的是RTPSession::AddStream，创建RTPStream对象。
QTSS_AddRTPStream(inParams->inClientSession, inParams->inRTSPRequest, &newStream, 0);
// Set up dictionary items for this stream
QTSS_SetValue(newStream,qtssRTPStrPayloadName,0,thePayloadName->Ptr,thePayloadName->Len);
... ...
// Place the stream cookie in this stream for future reference
// 即返回对应该trackID的fStreamArray数组中的ReflectorStream对象。
QTSS_SetValue(newStream, sStreamCookieAttr, 0, &theStreamCookie, sizeof(theStreamCookie));
void* theStreamCookie = theSession->GetStreamCookie(theTrackID);
... ...
QTSS_SendStandardRTSPResponse(...);
```

（7）、DoPlay

```
// it is a broadcast session so store the broadcast session.
// broadcast session？？？应用在什么场合？？？
if(inSession == NULL)
{     ... ...};
... ...
// 实际上是调用RTPSession::Play函数，在该函数里会执行“this->Signal(Task::kStartEvent)”
// 从而导致RTPSession::Run函数运行。
// 在RTPSession::Run函数里，调用
// fModule->CallDispatch(QTSS_RTPSendPackets_Role, &theParams)。
// 在我们分析的播放sdp文件这个情景里，fModule在RTSPSession::Run函数里被
// SetPacketSendingModule函数设置成为QTSSReflectorModule，而该Module并不支持
// QTSS_RTPSendPackets_Role，所以RTPSession::Run返回0，从而RTPSession::Run函数不会被
// TaskThread再次调度。
QTSS_Play(...);
QTSS_SendStandardRTSPResponse(...);
```

（二）、一些类的定义

- RTPSessionOutput

（1）、WritePacket

```
// make sure all RTP streams with this ID see this packet
for (UInt32 z = 0; QTSS_GetValuePtr(fClientSession, qtssCliSesStreamObjects, z,
             (void**)&theStreamPtr, &theLen) == QTSS_NoErr; z++)
{
       // 找到和 ReflectorStream 相关联的 RTPStream 对象
       // 在我们分析的播放 sdp 文件的这个情景里，该 RTPStream 对象在
       // QTSSReflectorModule::DoSetup 调用的 QTSS_AddRTPStream 函数里创建。
       if (this->PacketMatchesStream(inStreamCookie, theStreamPtr))
       {
              if(this->FilterPacket(theStreamPtr, inPacket))    return QTSS_NoErr;
```

```
                if (this->PacketAlreadySent(theStreamPtr,inFlags, packetIDPtr))
                        return QTSS_NoErr;
                if (!this->PacketReadyToSend(theStreamPtr,&currentTime, inFlags,
packetIDPtr,
                                                timeToSendThisPacketAgain))
                        return QTSS_WouldBlock;
                QTSS_PacketStruct thePacket;
                thePacket.packetData = inPacket->Ptr;
                // 这样设置 packetTramsmitTime 的原因？？？
                thePacket.packetTransmitTime = (currentTime - packetLatenessInMSec) +
                                (fBufferDelayMSecs - (currentTime - *arrivalTimeMSecPtr));
                // 实际上调用的是 RTPStream::Write 函数，见该函数的分析.
                // 通过 UDP Socket 发送音视频流。注意在我们分析的情景里，这个 Socket 跟我们
                // 用来和 mp4live 交互的 socket 不同。
                writeErr = QTSS_Write(*theStreamPtr, &thePacket, inPacket->Len, NULL,
                                inFlags | qtssWriteFlagsWriteBurstBegin);
                if (writeErr == QTSS_WouldBlock)
                {       *timeToSendThisPacketAgain = thePacket.suggestedWakeupTime;     }
                else {
                        fLastIntervalMilliSec = currentTime  -  fLastPacketTransmitTime;
                        if (fLastIntervalMillisec > 1000)   fLastIntervalMilliSec = 5;
                        fLastPacketTransmitTime = currentTime;
                        if (inFlags & qtssWriteFlagsIsRTP)
                                QTSS_SetValue (*theStreamPtr, sLastRTPPacketIDAttr, 0,
                                                packetIDPtr, sizeof(UInt64));
                        else if (inFlags & qtssWriteFlagsIsRTCP)
                }
        }
}
```

- ReflectorSender

（1）、SendPacketsToOutput

```
OSQueueElem* lastPacket = currentPacket;
// starts from beginning if currentPacket == NULL, else from currentPacket
// 将传进来的 Packet 作为当前 packet
OSQueueIter qIter(&fPacketQueue, currentPacket);
while (!qIter.IsDone()) {
        currentPacket = qIter.GetCurrent();         lastPacket = currentPacket;
        thePacket = (ReflectorPacket*)currentPacket->GetEnclosingObject();
        packetLateness =  bucketDelay;      timeToSendPacket = -1;
        // 实际上是调用 RTPSessionOutput::WritePacket
        err = theOutput->WritePacket(&thePacket->fPacketPtr, fStream... )
        if (err == QTSS_WouldBlock) {
                if((timeToSendPacket >0)&&((fNextTimeToRun +currentTime) >timeToSendPacket))
                        fNextTimeToRun = timeToSendPacket - currentTime;
                if (theOutput->fLastIntervalMilliSec < 5 )
                        theOutput->fLastIntervalMilliSec = 5;
                if (theOutput->fLastIntervalMilliSec >= 1000 )
                        theOutput->fLastIntervalMilliSec = 1000;
                else    theOutput->fLastIntervalMilliSec *= 2;
                if (timeToSendPacket<0)  fNextTimeToRun =theOutput->fLastIntervalMilliSec;
                if (fNextTimeToRun > 1000)   fNextTimeToRun = 1000;
                if (fNextTimeToRun < 5)      fNextTimeToRun = 5;
                break;
        }
```

```
        count++;        qIter.Next();
    }
    // 注意：如果 WritePacket 返回为 QTSS_WouldBlock，则 lastPacket 不为空。
    return lastPacket;
```

（2）、RemoveOldPackets

遍历 fPacketQueue

```
        packetDelay = theCurrentTime  -  thePacket->fTimeArrived;
        // remove packets that are too old
        // sMaxPacketAgeMSec = sOverBufferMsec，对应于配置文件中
        // reflector_buffer_size_sec，缺省为 10s，如果延迟大于这个时间，就去掉这个包
        if ( !thePacket->fNeededByOutput && packetDelay > currentMaxPacketDelay) {
                thePacket->Reset();  fPacketQueue.Remove(elem);
                inFreeQueue->EnQueue(elem);
        } else {
                thePacket->fNeededByOutput = false;
                if (pakcetDelay <= currentMaxPacketDelay)        break;
        }
```

（3）、ReflectPackets

```
    /*************************************************************************************
*******
    /   ReflectorSender::ReflectPackets
    /   There are n ReflectorSender's for n output streams per presentation.
    /   Each sender is associated with an array of ReflectorOutput's.  Each output
    /   represents a client connection.  Each output has # RTPStream's.
    /   When we write a packet to the ReflectorOutput he matches it's payload
    /   to one of his streams and sends it there.
    /   To smooth the bandwitdth (server, not user) requirements of the reflected
    /   streams, the Sender groups the ReflectorOutput's into buckets.  The input
    /   streams are reflected to each bucket progressively later in time.  So rather
    /   than send a single packet to say 1000 clients all at once, we send it to
    /   just the first 16, then then next 16 100 ms later and so on.
    /
    /   intputs     ioWakeupTime - relative time to call us again in MSec
    /               inFreeQueue - queue of free packets.
    /*************************************************************************************
*******
    // Call old routine for relays; they don't want buffering.
    // 在 reflectorSession::FindOrCreateSession 函数里，调用了 SetHasBufferedStreams(true)
    if (!fStream->BufferEnabled()) {
            this->ReflectRelayPackets(ioWakeupTime,inFreeQueue);
            return;
    }
    // make sure to reset these state variables
    fHasNewPackets = false;     fNextTimeToRun = 10000;     // init to 10 secs
    // fRTCPSender: qtssWriteFlagsIsRTCP      fRTPSender: qtssWriteFlagsIsRTP
    if (fWriteFlag == qtssWriteFlagsIsRTCP)
            fNextTimeToRun = 1000;
    // determine if we need to send a receiver report to the multicast source
    if ((fWriteFlag == qtssWriteFlagsIsRTCP) && (currentTime > (fLastRRTime + kRRInterval)))
    {       fLastRRTime = currentTime;  fStream->SendReceiverReport();      }
    // Check to see if we should update the session's bitrate average
    fStream->UpdateBitRate(currentTime);
    // where to start new clients in the q
    // sFirstPakcetOffsetMsec 缺省为 500
```

```
        // 在 sender 的 fPacketQueue 队列里寻找合适的包
        //      packetDelay = theCurrentTime - thePacket->fTimeArrived;
        //      这里的 sOverBufferInSec 对应配置文件中的“reflector_buffer_size_sec”
        //       if ( packetDelay <= (ReflectorStream::sOverBufferInMsec - offsetMsec) )
        //             oldestPacketInClientBufferTime = &thePacket->fQueueElem;
        //      ... return  oldestPacketInClientBufferTime
        fFirstPacketInQueueForNewOutput =
                this->GetClientBufferStartPacketOffset(ReflectorStream::sFirstPacketOffsetMsec);
        // 我们在 QTSSReflectorModule::DoSetup 里面看到，对于一个 ReflectorSession 的每一个
        // ReflectorStream，都调用了 AddOutput 添加了一个 RTPSessionOutput 对象。
        // 在开启 n 个窗口同时播放同一个 sdp 文件的情况下，会有 n 个 theOutput 对应 n 个 RTPStream,
        // 依序通过这 n 个 theOutput 发送 RTP 数据。
        for (UInt32 bucketIndex = 0; bucketIndex < fStream->fNumBuckets; bucketIndex++)
                for (UInt32 bucketMemberIndex = 0; bucketMemberIndex < fStream->sBucketSize; bucketMemberIndex++)
                {       ReflectorOutput* theOutput = fStream->fOutputArray[bucketIndex][bucketMemberIndex];
                        if (theOutput != NULL) {
                                // 检查这个 output 是否处于 play 状态
                                if ( flase == theOutput->IsPlaying())       continue;
                                // 返回既在 fBookmarkedPacketsElemsArray 数组同时属于 fPacketQueue
                                // 的成员
                                packetElem = theOutput->GetBookMarkedPacket(&fPacketQueue);
                                if (packetElem == NULL) {
                                        // everybody starts at the oldest packet in the buffer delay
                                        // or uses a bookmark
                                        packetElem = fFirstPacketInQueueForNewOutput;
                                        theOutput->fNewOutput = false;
                                }
                                // sBucketDelayInMsec 对应于配置文件中的
                                // reflector_bucket_offset_delay_msec，缺省值为 73.
                                bucketDelay =
ReflectorStream::sBucketDelayInMsec*(SInt64)bucketIndex;
                                packetElem = this->SendPacketsToOutput(theOutput,
                                                        packetElem,currentTime, bucketDelay);
                                if (packetElem) {
                                        ReflectorPacket *thePacket =
                                                (ReflectorPacket *)packetElem->GetEnclosingObject();
                                        thePacket->fNeededByOutput = true;
                                        // 添加到 fBookmarkedPacketsElemsArray
                                        theOutput->SetBookMarkPacket(packetElem);
                                }
                        }
                }
        }
        this->RemoveOldPackets(inFreeQueue);
        fFirstNewPacketInQueue = NULL;
        // Don't forget that the caller also wants to know when we next want to run
        // ReflectorSocket::Run 根据*ioWakeupTime 来决定 idelTimer 的值。
        if (*ioWakeupTime == 0)      *ioWakeupTime = fNextTimeToRun;
        else if((fNextTimeToRun > 0) && (*ioWakeupTime > fNextTimeToRun))
                *ioWakeupTime = fNextTimeToRun;
        // exit with fNextTimeToRun in real time, not relative time.
        fNextTimeToRun += currentTime;
```

- ReflectorSocket

  IdleTask、UDPSocket 的继承类。

  （1）、构建函数

```
: IdleTask(), UDPSocket(NULL, Socket::kNonBlockingSocketType | UDPSocket::kWantsDemuxer),
    ... ...
this->SetTaskName(“ReflectorSocket”);    this->SetTask(this);
for (UInt32 numPackets = 0; numPackets < kNumPreallocatedPackets; numPackets++)
{
        ReflectorPacket* packet = NEW ReflectorPacket();
        fFreeQueue.EnQueue(&packet->fQueueElem);   // put this packet onto the free queue
}
```

（2）、AddSender

```
... ...
this->GetDemuxer()->RegisterTask(inSender->fStream->fStreamInfo.fSrcIPAddr, 0, inSender);
fSenderQueue.EnQueue(&inSender->fSocketQueueElem);
```

（3）、ProcessPacket

由一个循环组成。

```
// 对于 mp4live 过来的音视频流来说，肯定不是 RTCP 数据。
if (GetLocalPort() & 1)      thePacket->fIsRTCP = true;
else                         thePacket->fIsRTCP = false;
// always refresh timeout even if we are filtering
if (fBroadcasterClientSession != NULL)     {      ... ...}
if (thePacket->fPacketPtr.Len == 0)
{      将 thePacket 重新插入 fFreeQueue 队列，重新调用 RequestEvent 监听端口      }
if (thePacket->IsRTCP())     {      ... ...}
// Only reflect one SSRC stream at a time. Pass the packet and whether it is an RTCP or
// RTP packet based on the port number.
if (fFilterSSRCs)     this->FilterInvalidSSRCs(thePacket, GetLocalPort() & 1);
// Find the appropriate ReflectorSender for this packet.
// 在 bindSockets 函数里，已经对 Socket A、B 调用 AddSender 注册了 fRTPSender、fRTCPSender
ReflectorSender* theSender=(ReflectorSender*)this->GetDemuxer()->GetTask(theRemoteAddr,
0);
```

对 thePacket、theSender 进行一系列设置

```
// 将 thePacket 插入 theSender 的 fPacketQueue
theSender->fPacketQueue.EnQueue(&thePacket->fQueueElem);
... ...
```

（4）、GetIncomingData

```
theRemoteAddr = theRemotePort = 0;
// get all the outstanding packets for this socket
while(true) {
        // get a packet off the free queue.
        // 如果 fFreeQueue 里已经没有 ReflectorPacket 链接，则创建一个 ReflectorPacket 对象
        // 否则从 fFreeQueue 里 Dequeue 一个 ReflectorPacket 对象
        ReflectorPacket* thePacket = this->GetPacket();
        thePacket->fPacketPtr.Len = 0;
        // 调用::recvfrom 从 socket 里读取数据
        this->RecvFrom(&theRemoteAddr, &theRemotePort, thePacket->fPacketPtr.Ptr,
                ReflectorPacket::kMaxReflectorPacketSize, &thePacket->fPacketPtr.Len);
        // 获取 Socket 对应的 Sender，对 Sender、thePacket 进行一系列设置，最终将 thePacket
        // 挂入 Sender 的 fPacketQueue
        if (this->ProcessPacket(inMilliseconds,thePacket,theRemoteAddr, theRemotePort))
                break;
}
```

（5）、Run

我们在 ReflectorStream::BindSockets 函数里看到：

```
        fSockets->GetSocketA()->RequestEvent(EV_RE); //这里的 Socket 是 ReflectorSocket 对象
        fSockets->GetSocketB()->RequestEvent(EV_RE);
```

这样一旦监听的 socket 有数据（相当于 mp4live 将编码的音视频流通过 socket 发送过来），EventContext::ProcessEvent 函数会调用 Siganl(Task::kReadEvent)，导致该 Run 函数运行。

```
    // We want to make sure we can't get idle events WHILE we are inside this function. That
    // will cause us to run the queues unnecessarily and just get all confused.
    this->CancelTimeout();
    theEvents = this->GetEvents();
    // 从 socket 里面读取数据，并和 Sender 关联起来
    if (theEvents & Task::kReadEvent)  this->GetIncomingData(theMilliseconds);
    // Now that we've gotten all available packets, have the streams reflect
    // 为什么这里不只是该 Socket 所对应的 Sender，而是所有的 Sender!!!
    // 可能是因为另外 Socket 对应的 Sender 也会有数据等待 Reflect
    for (OSQueueIter iter2(&fSenderQueue); !iter2.IsDone(); iter2.Next())
    {
        ReflectorSender* theSender2 =
                    (ReflectorSender*)iter2.GetCurrent()->GetEnclosingObject();
        // 根据 fNextTimeToRun 和当前时间判断是否马上进行 Reflect.
        // Sender 的 fNextTimeToRun 该如何确定？？？
        if (theSender2 != NULL && theSender2->ShouldReflectNow(theMilliseconds, &fSleepTime))
            theSender2->ReflectPackets(&fSleepTime, &fFreeQueue);
    }
    // For smoothing purposes, the streams can mark when they want to wakeup.
    // fSleepTime 实际上是 Sender 的 fNextTimeToRun，利用 IdleTimerThread 实现下一次的继续运行。
    if (fSleepTime > 0)  this->SetIdleTimer(fSleepTime);
    return 0;            // 返回 0,说明该 Run 函数不会继续被运行除非被 Signal 唤醒
```

- ReflectorStream

This object supports reflecting an RTP multicast stream to N RTPStreams. It spaces out the packet send times in order to maximize the randomness of the sending pattern and smooth the stream.

（1）、构建函数

```
... ...
fStreamInfo.Copy(*inInfo);
// Allocate Bucket array
// 创建 fNumBuckets 个 Bucket(即 ReflectorOutput**)对象数组，地址赋给 fOutputArray.
// 针对 fOutputArray 的每一个元素，创建指向 ReflectorOutput 的指针数组，将地址赋给
// 发 outputArray 的元素。
this->AllocateBucketArray(fNumBuckets);
// Write RTCP Packet
设置 fReceiverReportBuffer、fEyeLocation、fReceiverReportSize
if (SocketUtils::IsMulticastIPAddr(fStreamInfo.fDestIPAddr))
{ fDestRTCPAddr = fStreamInfo.fDestIPAddr;fDestRTCPPort = fStreamInfo.fPort+1;  }
```

（2）、BindSockets

```
... ...
// get a pair of sockets. The socket must be bound on INADDR_ANY because we don't know
// which interface has access to this broadcast. If there is a source IP address
specified
// by the source info, we can use that to demultiplex separate broadcasts on the same
// port. If the src IP addr is 0, we cannot do this and must dedicate 1 port per
broadcast
// 如果找不到，就创建 socket 对
// 注意这里调用的是 ReflectorSocketPool::ConstructUDPSocketPair
```

```
        //      NEW UDPSocketPair(NEW ReflectorSocket(), NEW ReflectorSocket())
        // 这里是检查 sdp 文件所指定的目的地址是否为多播地址，相当于 mp4live 是否将媒体进行多播。
        isMulticastDest = (SocketUtils::IsMulticastIPAddr(fStreamInfo.fDestIPAddr));
        // 注意 GetUDPSocketPair 搜索 socket 对的条件。对于指定了 IP、Port 的情况，
        // QTSServer::SetupUDPSockets 创建的 socket 对显然无法满足要求，所以这里会再次创建 socket
        // 对。
        if (isMulticastDest) {
                fSockets = sSocketPool.GetUDPSocketPair(INADDR_ANY, fStreamInfo.fPort,
                                fStreamInfo.fSrcIPAddr, 0);
        } else {
                fSockets = sSocketPool.GetUDPSocketPair(fStreamInfo.fDestIPAddr, fStreamInfo.fPort,
                                fStreamInfo.fSrcIPAddr, 0);
        }
        ... ...
        // also put this stream onto the socket's queue of streams
        ((ReflectorSocket*)fSockets->GetSocketA())->AddSender(&fRTPSender);
        ((ReflectorSocket*)fSockets->GetSocketB())->AddSender(&fRTCPSender);
        // A broadcaster is setting up a UDP session so let the sockets update the session
        if (fStreamInfo.fSetupToReceive &&qtssRTPTransportTypeUDP==transportType&&inParams!=NULL)
        {
                ((ReflectorSocket*)fSockets->GetSocketA())->AddBroadcasterSession(inParams-
>inClientSession);
                ((ReflectorSocket*)fSockets->GetSocketB())->AddBroadcasterSession(inParams-
>inClientSession);
        }
        ((ReflectorSocket*)fSockets->GetSocketA())->SetSSRCFilter(filterState, timeout);
        ((ReflectorSocket*)fSockets->GetSocketB())->SetSSRCFilter(filterState, timeout);
        // Always set the Rcv buf size for the sockets. This is important because the these
        // sockets is only useful for RTCP Rrs.
        // 512K
        fSockets->GetSocketA()->SetSocketRcvBufSize(512 * 1024);
        fSockets->GetSocketB()->SetSocketRcvBufSize(512 * 1024);
        if (isMulticastDest) {      ... ...}      // 针对 sdp 的多播的目的地址
        fStreamInfo.fPort = fSockets->GetSocketA()->GetLocalPort();
        // 申请监听
        fSockets->GetSocketA()->RequestEvent(EV_RE);
        fSockets->GetSocketB()->RequestEvent(EV_RE);
        // Copy the source ID and setup the ref
        StrPtrLen theSourceID(fSourceIDBuf, kStreamIDSize);
        ReflectorStream::GenerateSourceID(&fStreamInfo, fSourceIDBuf);
        fRef.Set(theSourceID, this);

        （3）、AddOutput
        // 寻找第一个 fOutputArray[num][x] == NULL,将 putInThisBucket 设为 num
        if (putInThisBucket < 0)     putInThisBucket = this->FindBucket();
        if (fNumBuckets <= (Uint32)putInThisBucket)       this-
>AllocateBucketArray(putInThisBucket * 2);
        // 保存 inOutput
        for(UInt32 y=0; y<sBucketSize; y++) {
                if (fOutputArray[putInThisBucket][y] == NULL)
                {       fOutputArray[putInThisBucket][y] = inOutput;
                        fNumElements++;       return putInThisBucket;
                }
        }
```

- ReflectorSession

This object supports reflecting an RTP multicast stream to N RTPStreams. It spaces out the packet send times in order to maximize the randomness of the sending pattern and smooth the stream.

（1）、SetupReflectorSession

```
fLocalSDP.Delete();  fLocalSDP.Ptr = inInfo->GetLocalSDP(&fLocalSDP.Len);
// Allocate all our ReflectorStreams, using the SourceInfo
delete fStreamArray;
// 音视频分为两个流，对于每个流，创建 ReflectorStream 对象。
fStreamArray = NEW ReflectorStream*[fSourceInfo->GetNumStreams()];
for (UInt32 x = 0; x < fSourceInfo->GetNumStreams(); x++)
{       ... ...
        if(theStreamRef == NULL)
        {
                fStreamArray[x] = NEW ReflectorStream(fSourceInfo->GetStreamInfo(x));
                fStreamArray[x]->BindSockets(inParams,inFlags, filterState, filterTimeout);
                fStreamArray[x]->SetEnableBuffer(this->fHasBufferedStreams);
                fSourceInfo->GetStreamInfo(x)->fPort = fStreamArray[x]->GetStreamInfo()->fPort;
                ReflectorStream::GenerateSourceID(fSourceInfo->GetStreamInfo(x), &theStreamID[0]);
                sStreamMap->Register(fStreamArray[x]->GetRef());
        }
}
```

（2）、AddOutput

```
bucket = -1;  lastBucket = -1;
while (true) {
        // 针对每一个 ReflectorStream 调用 AddOutput 添加 inOutput
        for ( ; x < fSourceInfo->GetNumStreams(); x++) {
                bucket = fStreamArray[x]->AddOutput(inOutput, bucket);
                if (bucket == -1)     break;
                else {
                        lastBucket = bucket;
                        if (isClient)         fStreamArray[x]->IncEyeCount();
                }
        }
        ... ...
}       atomic_add(&fNumOutputs, 1);
```

- QTAtom

（1）、构建函数

```
QTAtom::QTAtom(QTFile * File, QTFile::AtomTOCEntry * Atom, Bool16 Debug, Bool16 DeepDebug)
    : fDebug(Debug), fDeepDebug(DeepDebug), fFile(File)
    {  memcpy(&fTOCEntry, Atom, sizeof(QTFile::AtomTOCEntry));  }
```

- QTAtom_mvhd

基于 QTAtom 类

（1）、构建函数

```
QTAtom_mvhd::QTAtom_mvhd(QTFile* File, QTFile::AtomTOCEntry* TOCEntry, Bool16 Debug, Bool16
DeepDebug)
    : QTAtom(File, TOCEntry, Debug, DeepDebug)
    {  }
```

（2）、Initialize()

```
// Parse this atom's fields. fFile 为 QTFile 类型的对象
// 调用 fFile->Read(fTOCEntry.AtomDataPos+Offset, Buffer, Length)
ReadInt32(mvhdPos_VersionFlags, &tempInt32);
fVersion = (Uint8)((tempInt32 >> 24) & 0x000000ff);         fFlags = tempInt32 &
```

0x00ffffff;

通过 fFile->Read 读取各个参数数据保存到 fCreationTime、fModificationTime... ...

- QTTrack

  The central point of control for a track in a QTFile.

  （1）、构建函数

```
QTTrack::QTTrack(QTFile * File, QTFile::AtomTOCEntry * Atom, Bool16 Debug, Bool16 DeepDebug)
        : fDebug(Debug), fDeepDebug(DeepDebug), fFile(File),
        ... ...
    {
        QTFile::AtomTOCEntry *tempTOCEntry;
        // Make a copy of the TOC entry.
        memcpy(&fTOCEntry, Atom, sizeof(QTFile::AtomTOCEntry));
        // Load in the track header atom for this track.
        fFile->FindTOCEntry(“:tkhd”, &tempTOCEntry, &fTOCEntry)
        // 创建 QTAtom_tkhd 类对象，并执行 Initialize 函数，类似于 QTAtom_mvhd 类。
        fTrackHeaderAtom = NEW QTAtom_tkhd(fFile, tempTOCEntry, fDebug, fDeepDebug);
        fTrackHeaderAtom->Initialize();
    }
```

  （2）、Initialize()

```
    // See if this track has a name and load it in.
    // fTOCEntry 在构建函数里初始化。
    if( fFile->FindTOCEntry(":udta:name", &tempTOCEntry, &fTOCEntry) ) {
        fTrackName = NEW char[ (SInt32) (tempTOCEntry->AtomDataLength + 1) ];
        if( fTrackName != NULL )
            fFile->Read(tempTOCEntry->AtomDataPos, fTrackName, (UInt32) tempTOCEntry->AtomDataLength);
    }
    // Load in the media header atom for this track.
    fFile->FindTOCEntry(":mdia:mdhd", &tempTOCEntry, &fTOCEntry)
    fMediaHeaderAtom = NEW QTAtom_mdhd(fFile, tempTOCEntry, fDebug, fDeepDebug);
    fMediaHeaderAtom->Initialize();
    // Load in the edit list atom for this track.
    类似于上面那样创建并初始化 fEditListAtom，并初始化 fFirstEditMediaTime。
    // Load in the data reference atom for this track.
    创建并初始化 fDataReferenceAtom。
    // Load in the sample table atoms.
    创建并初始化 fTimeToSampleAtom、fCompTimeToSampleAtom、fSampleToChunkAtom、
    fSampleDescriptionAtom、fChunkOffsetAtom、fSampleSizeAtom、fSyncSampleAtom

    fIsInitialized = true;
```

- QTHintTrack

  QTTrack 的继承类，The central point of control for a hint track in a QTFile.

  （1）、构建函数

```
QTHintTrack::QTHintTrack(QTFile * File, QTFile::AtomTOCEntry * Atom, Bool16 Debug, Bool16 DeepDebug)
        : QTTrack(File, Atom, Debug, DeepDebug),
        ... ...
```

  （2）、Initialize()

```
QTTrack::Initialize()
```

```
        // Get the sample description table for this track and verify that it is an RTP track.
        // fSampleDescriptionAtom 在 QTTrack::Initialize 里面创建并初始化
        fSampleDescriptionAtom->FindSampleDescription(FOUR_CHARS_TO_INT('r', 't', 'p', ' '),
&sampleDescription, &sampleDescriptionLength)
        ::memcpy(&fMaxPacketSize, sampleDescription + 20, 4);
        fMaxPacketSize = ntohl(fMaxPacketSize);
        for( pSampleDescription = (sampleDescription + 24); pSampleDescription <
                        (sampleDescription + sampleDescriptionLength);)
        {
                // Get the entry length and data type of this entry
                ::memcpy(&entryLength, pSampleDescription+0, 4);  entryLength=ntohl(entryLength);
                ::memcpy(&dataType, pSampleDescription+4, 4);     dataType = ntohl(dataType);
                // Process this data type.
                switch(dataType) {
                        case FOUR_CHARS_TO_INT('t', 'i', 'm', 's'):       // tims RTP timescale
                                ::memcpy(&fRTPTimescale, pSampleDescription + 8, 4);
                                fRTPTimescale = ntohl(fRTPTimescale);
                                break;
                        case FOUR_CHARS_TO_INT('t', 's', 'r', 'o'):       // tsro Timestamp random
offset
                                ::memcpy(&fTimestampRandomOffset, pSampleDescription + 8, 4);
                                fTimestampRandomOffset = ntohl(fTimestampRandomOffset);
                                break;
                        case FOUR_CHARS_TO_INT('s', 'n', 'r', 'o'): // snro Sequence number random
offset
                                ::memcpy(&fSequenceNumberRandomOffset, pSampleDescription + 8, 2);
                                fSequenceNumberRandomOffset = ntohl(fSequenceNumberRandomOffset);
                                break;
                }       pSampleDescription += entryLength;
        }
        // Load in the hint info atom for this track.
        创建并初始化 fHintInfoAtom
        // Load in the hint track reference atom for this track.
        创建并初始化 fHintTrackReferenceAtom
        // Allocate space for our track reference table.
        numTrackRefs = fHintTrackReferenceAtom->GetNumReferences();
        fTrackRefs = NEW QTTrack *[numTrackRefs];
        // Locate all of the tracks that we use, but don't initialize them until we actually try
        // to access them.
        for( UInt32 CurRef = 0; CurRef < numTrackRefs; CurRef++ )
        {
                // Get the reference and make sure it's not empty.
                fHintTrackReferenceAtom->TrackReferenceToTrackID(CurRef, &trackID)
                // Store away a reference to this track.
                fFile->FindTrack(trackID, &fTrackRefs[CurRef]);
        }
        // Calculate the first RTP timestamp for this track.
        设置 fFirstRTPTimestamp
        fHintTrackInitialized = true;
```

- OSFileSource

  底层的文件操作类，simple file abstraction. This file abstraction is ONLY to be used for files intended for serving。

  （1）构建函数

  会调用 Set 函数，在该函数里，调用 open 打开文件、调用 fstat 获取文件信息。

（2）Read
调用 ReadFromPos 或者 ReadFromCache。
ReadFromPos 和 ReadFromCache 的区别待分析！！

- QTSSFile

（1）Open

```
// Because this is a role being executed from inside a callback, we need to make
// sure that QTSS_RequestEvent will not work ? ? ?
... ... curTask = theState->curTask;        ... ...
```

调用注册 QTSS_OpenFilePreProcess_Role 处理的模块的处理函数。系统自带的模块没有提供这个 Role 的处理。
如果返回 QTSS_FileNotFound，则调用注册 QTSS_OpenFile_Role 处理的模块的处理函数。QTSSPosixFileSysModule 提供了这个 Role 的处理。
将上述找到的模块赋给 fModule，后续可以通过 fModule 来调用该模块没有注册的 Role 处理函数。

```
... ...theState->curTask = curTask;
```

- QTFile

（1）、Read(UInt64 Offset, char* const Buffer, UInt32 Length, QTFile_FileControlBlock* FCB)

```
if (FCB)      FCB->Read(&fMovieFD, Offset, Buffer, Length);
else   {      QTSS_Seek(fMovieFD, Offset);       QTSS_Read(fMovieFD, Buffer, Length); }
```

（2）、GenerateAtomTOC

```
// Scan through all of the atoms in this movie, generating a TOC entry for each one.
CurPos = 0;
while ( Read(CurPos, (char *)&atomLength, 4) ) {
        // Swap the AtomLength for little-endian machines.
        CurPos += 4;  atomLength = ntohl(atomLength); BigAtomLength = (Uint64)atomLength;
        hasBigAtom = false;

        // Is AtomLength zero?  If so, and we're in a 'udta' atom, then all is
        // well (this is the end of a 'udta' atom).  Leave this level of
        // siblings as the 'udta' atom is obviously over.
        // 发现了 End-of-udta marker
        if( (BigAtomLength==0)&&CurParent&&(CurParent->AtomType==FOUR_CHARS_TO_INT('u', 'd', 't',
'a')) )        {      LastTOCEntry = CurParent;   CurParent = CurParent->Parent;
                goto lbl_SkipAtom;          // Keep moving up.
        }
        // Is the AtomLength zero?  If so, this is a "QT atom" which needs some additional
        // work before it can be processed.
        else if(BigAtomLength == 0)
        {
                // This is a QT atom; skip the (rest of the) reserved field and the lock
                // count field.
                CurPos += 22;
                // Read the size and the type of this atom.
                Read(CurPos, (char*)&atomLength, 4);
                CurPos += 4;  BigAtomLength = (Uint64)ntohl(atomLength);
                Read(CurPos, (char*)&AtomType, 4);
                CurPos += 4;  AtomType = (Uint64)ntohl(AtomType);
                // Skip over the rest of the fields.
                CurPos += 12;
                // Set the header size to that of a QT atom.
```

```
            CurAtomHeaderSize = 10+16+4+4+4+2+2+4;
        } else {            // This is a normal atom; get the atom type.
            Read(CurPos, (char*)&AtomType, 4); CurAtomHeaderSize = 4 + 4;
            CurPos += 4;  AtomType = (Uint64)ntohl(AtomType);
            if(atomLength == 1)         // large size atom
            {
                Read(CurPos, (char*)&BigAtomLength, 8);
                BigAtomLength = QTAtom::NTOH64(BigAtomLength);    CurPos += 8;
                CurAtomHeaderSize += 8;     //AtomLength+AtomType+big atom length
                hasBigAtom = true;
            }
            if(AtomType == FOUR_CHARS_TO_INT('u', 'u', 'i', 'd'))
            {     sExtendedTypeSize = 16;
                // read and just throw it away we don't need to store
                Read(CurPos, (char*)usertype, 16);
                CurPos += sExtendedTypeSize; CurAtomHeaderSize += sExtendedTypeSize;
            }
        }
        if(BigAtomLength == 0)      return false;
        if ((AtomType == FOUR_CHARS_TO_INT('m', 'o', 'o', 'v')) && (hasMoovAtom))
        {     // Skip over any additional 'moov' atoms once we find one.
            CurPos += BigAtomLength - CurAtomHeaderSize;    continue;
        } else if (AtomType == FOUR_CHARS_TO_INT('m', 'o', 'o', 'v')
            hasMoovAtom = true;
        else if (!hasMoovAtom) {
            CurPos == BigAtomLength - CurAtomHeaderSize;    continue;
        }
        // Create a TOC entry for this atom，并加入 fTOC、fTOCOrdHead、fTOCOrdTail 维护的
        // 链表。
        NewTOCEntry = NEW AtomTOCEntry();
        ... ...
        // Figure out if we have to descend into this entry and do so.
        switch (NewTOCEntry->AtomType)
        {
            case FOUR_CHARS_TO_INT('m', 'o', 'o', 'v'): //moov
            ... ... // clip trak matt edts tref mdia minf dinf stbl udta hnti hinf
            {     // All of the above atoms need to be descended into.  Set up our
                // variables to descend into this atom.
                if (NewTOCEntry->AtomDataLength > 0)
                {     CurParent = NewTOCEntry;    LastTOCEntry = NULL; continue; }
            } break;
        }
        CurPos += NewTOCEntry->AtomDataLength;    // Skip over this atom's data
lbl_SkipAtom:
        while(... ...) {
            LastTOCEntry = CurParent;   CurParent = CurParent->Parent;
        }
}
if(!this->ValidTOC())       // make sure we were able to read all the atoms.
        return false;
return true;                // The TOC has been successfully read in.
```

（4）、Open

Open a movie file and generate the atom table of contents.

```
// 创建 QTSSFile 对象，并调用 Open 函数
QTSS_OpenFileObject(fMoviePath, qtssOpenFileReadAhead, &fMovieFD);
// We have a file, generate the mod date str
fModDateBuffer.Update(this->GetModDate());
// Generate the table of contents for this movie.
GenerateAtomTOC();
// Find the Movie Header atom and read it in.
// 根据匹配 AtomType 找到链表中的 AtomTOCEntry 对象。
FindTOCEntry(“moov:mvhd”, &TOCEntry);
fMovieHeaderAtom = NEW QTAtom_mvhd(this, TOCEntry, fDebug, fDeepDebug);
// 读取这个 atom 的数据
fMovieHeaderAtom->Initialize();

// Create QTTrack objects for all of the tracks in this movie.  (Although this does incur
// some extra resource usage where tracks are not used, they can always A) be disposed of
// later, or B) be ignored as their use of system resources is exceptionally minimal.)
// NOTE that the tracks are *not* initialized here.  That is done when they are actually
// used; either directly or by a QTHintTrack.
TOCEntry = NULL;
// 生成所有的 Track/Hint Track 信息。
while (FindTOCEntry(“moov:trak”, &TOCEntry, TOCEntry)) {
        ListEntry = NEW TrackListEntry();
        // Make a hint track if that's what this is.
        // QTHintTrack 基于 QTTrack 类。在 QTTrack 类的构建函数里，创建 QTAtom_tkhd 类对象，
        // 并执行 Initialize 函数读取相关信息。
        if( FindTOCEntry(":tref:hint", NULL, TOCEntry) ) {
                ListEntry->Track = NEW QTHintTrack(this, TOCEntry, fDebug, fDeepDebug);
                ListEntry->IsHintTrack = true;
        } else {
                ListEntry->Track = NEW QTTrack(this, TOCEntry, fDebug, fDeepDebug);
                ListEntry->IsHintTrack = false;
        }
        // GetTrackID 实际上返回的是 QTAtom_thkd::fTrackID，由 QTTrack 构建函数调用的
        // fTrackHeaderAtom::Initialize(即 QTAtom_tkhd::Initialize)函数从媒体中读取并初始
化。
        // 在 fFirstTrack 所指向的链表中搜寻
        if(FindTrack(ListEntry->Track->GetTrackID(), theTrack))
        {       // A track with this track ID already exists. Ignore other tracks with
                // identical track IDs.
                delete ListEntry->Track;    delete ListEntry;    continue;
        }
        // Add this track object to our track list.
        ListEntry->TrackID = ListEntry->Track->GetTrackID();
        ListEntry->NextTrack=NULL;
        if(fFirstTrack == NULL)     fFirstTrack = fLastTrack = ListEntry;
        else { fLastTrack->NextTrack = ListEntry;  fLastTrack = ListEntry;     }

        fNumTracks++;
}
```

- QTRTPFile

(1)、new_QTFile

```
... ...
QTRTPFile::RTPFileCacheEntry    *fileCacheEntry;
```

```
        // Find and return the QTFile object out of our cache, if it exists.
        // 从 gFirstFileCacheEntry 开始搜索，一旦发现 listEntry->fFilename 和 filePath 相同，
        // 则返回 listEntry。
        if ( QTRTPFile::FindAndRefcountFileCacheEntry(filePath, &fileCacheEntry) )
        {       ... ...*theQTFile = fileCacheEntry->File; return errNoError;    }
        // 创建 QTFile 对象，并调用 Open 函数。
        *theQTFile = NEW QTFile(debugFlag, deepDebugFlag);
        (*theQTFile)->Open(filePath);
        // Add this file to our cache and release the global add mutex.
        // 和上面的搜索动作相对应
        QTRTPFile::AddFileToCache(filePath, &fileCacheEntry);
        // Finish setting up the fileCacheEntry.
        if( fileCacheEntry != NULL )
        {       fileCacheEntry->File = *theQTFile; fileCacheEntry->InitMutex->Unlock();
    }
```

（2）、Initialize

```
        // create our file object.
        this->new_QTFile(filePath, &fFile, fDebug, fDeepDebug);
        // Iterate through all of the tracks, adding hint tracks to our list.
        for(track = NULL; fFile->NextTrack(&track, track); )
        {
                在 fFile 中已生成的 track 信息中寻找 hint track 信息。
                对于 hint track：
                        创建并初始化 RTPTrackListEntry 对象。
                        将该对象添加到由 fFirstTrack、fLastTrack 维护的链表中。
                        计数
        }
        // If there aren't any hint tracks, there's no way we can stream this
        // movie, so notify the caller! ! !
        if (fNumHintTracks == 0)     return fErr = errNoHintTracks;
        return errNoError;           // The RTP file has been initialized.
```

（3）、AddTrack(UInt32 trackID, Bool16 useRandomOffset)

```
        // Find this track.
        this->FindTrackEntry(trackID, &trackEntry);
        OSMutexLocker locker(fFile->GetMutex());
        trackEntry->HintTrack->Initialize();              // Initialize this track.
        Set up the sequence number and timestamp offsets.
        // This track is now active.
        trackEntry->IsTrackActive = true;
```

- FileSession

```
class FileSession
{
    public:
       ... ...
       QTRTPFile      fFile;
       QTSS_PacketStruct   fPacketStruct;
       SDPSourceInfo  fSDPSource;
       ... ...
};
```

- RTPFileSession

(1) Initialize(StrPtrLen& inFilePath, Float32 inBufferSeconds)

首先创建 RTPFile 类的对象，在 RTPFile 的 Initialize 函数里，创建 QTSSFile 对象，并通过 QTSSFile 对象获取文件的头部及 SDP、track 信息，同时根据这些信息分配 data buffer。

```
// check to see if this file is already open
OSMutexLocker locker(sOpenFileMap.GetMutex());
OSRef* theFileRef = sOpenFileMap.Resolve((StrPtrLen*)&inFilePath);
if(theFileRef == NULL)
{
        fFile = NEW RTPFile();  //fFile 是 RTPFileSession 的类型为 RTPFile*的成员
        fFile->Initialize(inFilePath);
        sOpenFileMap.Register(fFile->GetRef());
        // unless we do this, the refcount won't increment (and we'll delete the session prematurely
        sOpenFileMap.Resolve(&inFilePath);
}
// 创建 QTSSFile 对象 fFileSource，并执行 Open 操作
QTSS_OpenFileObject(inFilePath.Ptr, 0, &fFileSource);
// Get the file length
QTSS_GetValue(fFileSource, qtssFlObjLength, 0, &fFileLength, &theLen);
// Allocate our data buffer
// inBufferSeconds 为 8，GetBytesPerSecond 即返回 fFile->fBytesPerSecond
fDataBufferSize = this->PowerOf2Floor((UInt32)(inBufferSeconds * fFile->GetBytesPerSecond()));
限制 fDataBufferSize 在 kMaxDataBufferSize(256k)和 kBlockSize(32k)之间
fReadBuffer = fDataBuffer = NEW Uint8[fDataBufferSize];
// Allocate a buffer of TrackInfos
fTrackInfo = NEW RTPFileSessionTrackInfo[fFile->GetMaxTrackNumber() + 1];
::memset(fTrackInfo, 0, fFile->GetMaxTrackNumber() * sizeof(RTPFileSessionTrackInfo));
```

- RTPFile

(1) Initialize(const StrPtrLen& inFilePath)

```
// 创建 QTSSFile 对象，并调用 Open 函数
QTSS_OpenFileObject(inFilePath.Ptr, 0, &theFile);
根据 inFilePath 设置 fFilePath
fRef.Set(fFilePath, this);
// Read the header，调用 QTSSFile 的 Read 函数，在该函数里会调用
// fModule 的 Dispatch 函数（在 QTSSFile 的 Open 函数里，寻找注册了 Open 相关 Role 的模
// 块，并将其赋给 fModule，本系统是 QTSSPosixFileSysModule）
QTSS_Read(theFile, &fHeader, sizeof(fHeader), &theLengthRead);
// Read the SDP data
// 类似于读取 header
fSDPData.Len = fHeader.fSDPLen;     fSDPData.Ptr = NEW char[fSDPData.Len + 1];
QTSS_Read(theFile, fSDPData.Ptr, fSDPData.Len, &theLengthRead);
// Parse the SDP Information
// 待分析!!
fSourceInfo.Parse(fSDPData.Ptr, fSDPData.Len);
// Read the track info
fTrackInfo = NEW RTPFileTrackInfo[fHeader.fNumTracks];
QTSS_Read(theFile, fTrackInfo, sizeof(RTPFileTrackInfo)*fHeader.fNumTracks, &theLengthRead);
// Create and read the block map
fBlockMap = NEW Uint8[fHeader.fBlockMapSize];
QTSS_Read(theFile, fBlockMap, fHeader.fBlockMapSize, &theLengthRead);
```

```
// Calculate bit rate of all the tracks combined
for (UInt32 x = 0; x < fHeader.fNumTracks; x++)
        totalBytes += (Float64)fTrackInfo[x].fBytesInTrack;
totalBytes /= fHeader.fMovieDuration;      fBytesPerSecond = (UInt32)totalBytes;
// Get the max track number
for (UInt32 x = 0; x < fHeader.fNumTracks; x++)
        if(fTrackInfo[y].fID > fMaxTrackNumber)
                fMaxTrackNumber = fTrackInfo[y].fID;
// 调用 close 函数，销毁 QTSSFile 对象。
QTSS_CloseFileObject(theFile);
```

- QTSSCallbacks

属于 Server core 一部分。该类主要是为 QTSServer 提供支持，比如一些独立子系统/模块通过 sCallbacks 来调用 DSS 的处理函数，而这些函数由 QTSSCallbacks 类提供定义。下面是一些成员函数的分析：

（1）QTSS_AddRole

```
QTSS_ModuleState* theState = (QTSS_ModuleState*)OSThread::GetMainThreadData();
```

注意：我们在前面分析的“加载一个模块”的情景中，QTSServer::AddModule 中就调用 OSThread::SetMainThreadData(&theModuleState)，其实参为：

```
theModuleState.curModule = inModule;
theModuleState.curRole = QTSS_Register_Role;
theModuleState.curTask = NULL;
```

返回 theState->curModule->AddRole(inRole)，实际上就是调用 QTSSModule::AddRole 函数。

在 QTSSModule::AddRole 函数里，根据传入的 Role 将 fRoleArray 数组的相应位置为 true。

如果是 QTSS_RTSPRequest_Role、QTSS_OpenFile_Role、QTSS_RTSPAuthenticate_Role，特别地用一些全局量做标记。

调用 SetValue ***Add this role to the array of roles attribute***

（2）QTSS_AddService

返回 QTSSDictionaryMap::GetMap(QTSSDictionaryMap::kServiceDictIndex)->

```
AddAttribute(inServiceName, (QTSS_AttrFunctionPtr)inFunctionPtr,
                                        qtssAttrDataTypeUnknown, qtssAttrModeRead);
```

DSS 在初始化的时候调用了 QTSSDictionaryMap 的 Initialize 成员函数，针对各种类型的信息（比如 ServerDict、PrefsDict、TextMessagesDict、ServiceDict、RTPStreamDict 等等）创建了 QTSSDictionaryMap 的类对象，放在 sDictionaryMaps 数组里。

QTSS_AddService 在 kServiceDict 对应的 QTSSDictionaryMap 的 fAttrArray 里记录相关的信息。

- QTSServerInterface 类

QTSServerInterface 类属于 DSS server core，它基于 QTSSDictionary 类。

This is the internal data storage object tagged as the QTSS_ServerObject in the API.

Each of the QTSS_ServerAttributes in the API is declared and implemented in this base class.

（1）、构建函数

QTSServer 是 QTSServerInterface 的继承类，在 StartServer 函数里创建 QTSServer 类对象的时候，必然会创建 QTSServerInterface 类对象。

```
QTSServerInterface::QTSServerInterface()
:  QTSSDictionary(QTSSDictionaryMap::GetMap(QTSSDictionaryMap::kServerDictIndex),
&fMutex),
        ……
{
```

初始化 sModuleArray 数组、sNumModulesInRole 数组。

```
// 注意：在构建函数的初始化部分，传给 QTSSDictionary 构建函数的是 kServerDictIndex
// 对应的 dict map。
// 调用基类 QTSSDictionary 的 SetVal 函数记录每个属性的数据来源和长度等信息。
```

```
        // 注意和 QTSServerInterface::Initialize 函数里调用 SetAttribute 的不同。
        this->SetVal(qtssSvrState, &fServerState, sizeof(fServerState));
        this->SetVal(qtssServerAPIVersion, &sServerAPIVersion, sizeof(sServerAPIVersion));
        ……
}
```

（2）、QTSServerInterface::Initialize()

```
// 调用 kServerDictIndex、kQTSSConnectedUserDictIndex 两个 dict map 的 SetAttribute 函数，
// 根据 sAttributes、sConnectedUserAttributes 进行属性信息统计。
for(Uint32 x = 0; x < qtssSvrNumParams; x++)
        QTSSDictionaryMap::GetMap(QTSSDictionaryMap::kServerDictIndex)->
                SetAttribute(x, sAttributes[x].fAttrName, sAttributes[x].fFuncPtr,
                        sAttributes[x].fAttrDataType, sAttributes[x].fAttrPermission);
for(Uint32 y = 0; y < qtssConnectionNumParams; y++)
        QTSSDictionaryMap::GetMap(QTSSDictionaryMap::kQTSSConnectedUserDictIndex)->
    SetAttribute(y, sConnectedUserAttributes[y].fAttrName,
sConnectedUserAttributes[y].fFuncPtr,
    sConnectedUserAttributes[y].fAttrDataType, sConnectedUserAttributes[y].fAttrPermission);

//write out a premade server header
……
```

（3）、QTSServerInterface::SetValueComplete()

QTSSDictionary::SetValueComplete 函数是虚函数，对于继承类 QTSServerInterface 来说，该函数主要是处理设置 qtssSvrState 属性的情况。
对于所有注册了 kStateChangeRole 的模块，都将会调用 CallDispatch 函数，执行具体模块的处理函数。

- QTSServer

属于 Server core 一部分。处理 server 的 startup 和 shutdown。
该类继承自 QTSServerInterface 类。下面是一些成员函数的分析：

（1）、Initialize()

在 startServer 函数创建 sServer 对象后，该函数被调用。

```
this->InitCallbacks();                    // 初始化 sCallbacks

// Dictionary Initialization
QTSSModule::Initialize();   // 调用 SetAttribute Setup all the dictionary stuff，下同
QTSServerPrefs::Initialize();
QTSSMessages::Initialize();
// 初始化 headerFormatter、noServerInfoHeaderFormatter、调用 SetAttribute。
RTSPRequestInterface::Initialize();
RTSPSessionInterface::Initialize();
RTPSessionInterface::Initialize();
RTPStream::Initialize();
RTSPSession::Initialize(); //初始化 sHTTPResponseHeaderPtr、
sHTTPResponseNoserverHeaderPtr
QTSSFile::Initialize();
QTSSUserProfile::Initialize();

// STUB SERVER INITIALIZATION
// Construct stub versions of the prefs and message dictionaries。
// QTSServerPrefs 类用来管理 QTSS 的配置信息
fSrvrPrefs = new QTSServerPrefs(inPrefsSource, false);
```

```
fSrvrMessages = new QTSSMessages(inMessagesSource);
QTSSModuleUtils::Initialize(fSrvrMessages, this, QTSServerInterface::GetErrorLogStream());

// Create Global Objects
// 这两个类的构建函数基本为空
fSocketPool = new RTPSocketPool();
fRTPMap = new OSRefTable(kRTPSessionMapSize);
```

加载 Error Log Module

```
// 在 startServer 函数里已经调用了 SocketUtils::Initialize，关于网络及 socket 的一些数据
// 已经被收集。
this->SetDefaultIPAddr();

// Record Startup Time
fStartupTime_UnixMilli = OS::Milliseconds();
fGMTOffset = OS::GetGMTOffset();

// Begin Listening
// 见下面的分析。
this->CreateListeners();
```

（2）CreateListeners()

参数 inPortOverride 是由 main.cpp 的 thePort 传进来（thePort 是指 RTSP Server 的监听端口，缺省为 0，可以通过-p 参数来指定）。
根据系统的 IP 地址数目和端口数目，创建 thePortTrackers 数组，记录 IP 地址和端口号。
将 thePortTrackers 数组的每一项和 fListeners 的比较，如果发现有 IP 地址和端口号均相同的一项，将 thePortTrackers 相应项的 fNeedsCreating 设为 false，并将 fListeners 这一项的指针保存到新创建的 newListenerArray 数组。
重新遍历 thePortTrackers 数组，处理 fNeedsCreating 为 true 的项：

创建 RTSPListenerSocket 类对象，并保存到 newListenerArray 数组，并调用该对象的 Initialize 成员函数。（注意：这里调用的 Initialize 函数实际上是 TCPListenerSocket::Initialize，在这个函数里会对 socket（流套接字）执行 open、bind、listen 等操作，注意这里绑定的 IP 地址缺省为 0<在配置文件里对应 bind_ip_addr 项>，表示监听所有地址的连接。绑定的 Port 缺省为 7070、554、8000、8001<在配置文件里对应 rtsp_port 项>）

如果 CreateListeners 的参数 startListeningNow 为 true，则 start listening（这部分我们会在 QTSServer::StartTasks 成员函数里描述）

把 fListeners 里包含有但是 newListenerArray 里没有（即不再需要的）的监听撤销掉。
最后，设置属性（注意这里调用的是 SetValue 函数）：

```
fListeners = newListenerArray; fNumListeners = curPortIndex;
this->SetValue(qtssSvrRTSPPorts, portIndex, &thePort, sizeof(thePort),
                                QTSSDictionary::kDontObeyReadOnly);
this->SetNumValues(qtssSvrRTSPPorts, portIndex);
```

（3）、InitModules()

```
// 打开 module 所在目录，针对该目录下的所有文件调用 CreateModule 创建模块。
// 在 startServer 调用 InitModules 从而又调用 LoadModules 的这个情景中，加载了
// QTSSHomeDirectoryModule、QTSSRefMovieModule 这两个模块。
LoadModules（fSrvrPrefs）；

// 加载一些已编入的模块。我们会分析 QTSSReflectorModule、QTSSRelayModule 模块的加载过程。
LoadCompiledInmodules();
```

```
this->BuildModuleRoleArrays();
```

处理 DSS 系统 log、message 等方面的初始化。

```
this->DoInitRole();
```

（4）、CreateModule()

通过创建一个模块对象、调用该模块的 SetupModule 函数、调用 QTSServer::AddModule 函数这三个步骤来完成创建加载模块的过程。

（5）、DoInitRole()

```
theInitParams.initParams.inServer = this;  theInitParams.initParams.inPrefs = fSrvrPrefs;
theInitParams.initParams.inMessages = fSrvrMessages;
theInitParams.initParams.inErrorLogStream = &sErrorLogStream;
theModuleState.curRole = QTSS_Initialize_Role;    theModuleState.curTask = NULL;
OSThread::SetMainThreadData(&theModuleState);

// Add the OPTIONS method as the one method the server handles by default (it handles
// it internally). Modules that handle other RTSP methods will add
(void)this->SetValue(qtssSvrHandledMethods, 0, &theOptionsMethod,
sizeof(theOptionsMethod));

// 对于那些支持 Initialize Role 的模块，通过它们的 CallDispatch 函数来调用具体的 Initialize
// 函数。实际上这也给各个模块（包括用户自定义的模块）一个执行初始化动作的机会。
// 很多模块都支持 Init Role！！！需要一个个分析！！！
for(x=0; x<QTSServerInterface::GetNumModulesInRole(QTSSModule::kInitializeRole); x++) {
        QTSSModule* theModule = QTSServerInterface::GetModule(QTSSModule::kInitializeRole,
x);
        theInitParams.initParams.inModule = theModule; theModuleState.curModule =
theModule;
        QTSS_Error theErr = theModule->CallDispatch(QTSS_Initialize_Role, &theInitParams);
        if (theErr != QTSS_NoErr)    {... ...}
}

this->SetupPublicHeader();
OSThread::SetMainThreadData(NULL);
```

（6）、SetupPublicHeader()

```
// After the Init role, all the modules have reported the methods that they handle.
// So, we can prune this attribute for duplicates, and construct a string to use in the
// Public: header of the OPTIONS response
// 将各个模块登记的支持方法信息汇总一起。在 RTSPSession 处理 OptionMethod 时，会将这些信
// 息发送出去。
Bool16 theUniqueMethods[qtssNumMethods + 1];
::memset(theUniqueMethods, 0, sizeof(theUniqueMethods));

for (UInt32 y = 0; this->GetValuePtr(qtssSvrHandledMethods, y, (void**)&theMethod,
&theLen)                          == QTSS_NoErr; y++)
        theUniqueMethods[*theMethod] = true;
```

（7）、StartTasks()

```
fRTCPTask = new RTCPTask();
fStatsTask = new RTPStatsUpdaterTask();

// Start listening
```

```
for (Uint32 x = 0; x < fNumListeners; x++)
        fListeners[x]->RequestEvent(EV_RE);
```

（8）、SetupUDPSockets()

function finds all IP addresses on this machine, and binds 1 RTP/RTCP socket pair to a port pair on each address.
SocketUtils::GetNumIPAddrs()返回 2，包括本机的 ip、loop 地址。

```
for (UInt32 theNumPairs = 0; theNumPairs < SocketUtils::GetNumIPAddrs(); theNumPairs++)
{
        // 在 QTSServer::Initialize 函数里，fSocketPool=new RTPSocketPool();
        // RTPSocketPool 是 UDPSocketPool 的继承类，它们的构建函数均为空。
        // 基于同一个 ip，创建一个 socket 对。
        UDPSocketPair* thePair=fSocketPool->CreateUDPSocketPair(SocketUtils::GetIPAddr(theNumPairs), 0);
        // 申请监听这两个 socket 端口
        thePair->GetSocketA()->RequestEvent(EV_RE);     thePair->GetSocketB()->RequestEvent(EV_RE);
```

➢ RTPSocketPool、UDPSocketPool

UDPSocketPool: Object that creates & maintains UDP socket pairs in a pool.

（1）、UDPSocketPool::CreateUDPSocketPair

```
... ...
while((!foundPair) && (curPort < kHighestUDPPort))
{
        socketBPort = curPort +1; // make socket pairs adjacent to one another
        theElem = ConstructUDPSocketPair();         // 创建一个 udp socket pair
        // 创建数据报 socket 端口
        theElem->fSocketA->Open();          theElem->fSocketB->Open();
        // Set sokcet options on these new sockets，主要是设置 socket buf size
        this->SetUDPSocketOptions(theElem);
        // 在两个 socket 端口上执行 bind 操作，两个 port 相差 1.
        theElem->fSocketA->Bind(inAddr, curPort);  theElem->fSocketB->Bind(inAddr, socketBPort);
        // 利用 UDPSocketPair 的 fElem 成员插入 fUDPQueue 队列。
        fUDPQueue.EnQueue(&theElem->fElem);         theElem->fRefCount++;
        ... ...
}
```

（2）、RTPSocketPool::ConstructUDPSocketPair

```
Task* theTask = ((QTSServer*)QTSServerInterface::GetServer())->fRTCPTask;

// construct a pair of UDP sockets, the lower one for RTP data (outgoing only, no demuxer
// necessary), and one for RTCP data (incoming, so definitely need a demuxer).
// These are nonblocking sockets that DON'T receive events (we are going to poll for data)
// They do receive events - we don't poll from them anymore
// 在 RTCPTask 的 Run 函数里，会寻找带有 Demuxer 的 socket，将接收到的数据交由 Demuxer 相联
// 系的 Task 进行处理。
// 注意这里传入的是 fRTCPTask!
return NEW UDPSocketPair(  NEW UDPSocket(theTask, Socket::kNonBlockingSocketType),
                    NEW UDPSocket(theTask, UDPSocket::kWantsDemuxer | Socket::kNonBlockingSocketType));
```

（3）、RTPSocketPool::SetUDPSocketOptions(UDPSocketPair* inPair)

```
// Apparently the socket buffer size matters even though this is UDP and being used for
// sending... on UNIX typically the socket buffer size doesn't matter because the packet
// goes right down to the driver. On Win32 and linux, unless this is really big, we get
// packet loss.
// Socket A 用来发送 RTP data，通过 setsockopt 将发送 buf 设为 256K
```

```
inPair->GetSocketA()->SetSocketBufSize(256 * 1024);

// Socket B 用来接收 RTCP data，注意这几个 socket 端口的 buffer size：
// TCPListenerSocket 的 rcv buf size 为 96K、TCPListenerSocket accept 后返回的 socket snd
// buf size 为 256K。
// RTP data Socket snd buf size 为 256K、RTCP data Socket rcv buf size 为 768K。
// Always set the Rcv buf size for the RTCP sockets. This is important because the
// server is going to be getting many many acks.
// 获得用于 RTCP 的 rcv buf size 大小，缺省为 768K，在配置文件中对应 rtcp_rcv_buf_size 项
// In case the rcv buf size is too big for the system, retry, dividing the requested size
// by 2. Until it works, or until some minimum value is reached.
while ((theErr != OS_NoErr) && (theRcvBufSize > 32))
{
        theErr = inPair->GetSocketB()->SetSocketRcvBufSize(theRcvBufSize * 1024);
        if (theErr != OS_NoErr)      theRcvBufSize >>= 1;
}
```

- UDPSocket

  Socket 的继承类，Socket 是 EventContext 的继承类。注意在 Socket 的构建函数里，调用了 EventContext 的 SetTask 函数，将传入的 intask 参数付给 EventContext 的 fTask 成员。

- UDPSocketPair

  （1）、构建函数

```
UDPSocketPair(UDPSocket* inSocketA, UDPSocket* inSocketB)
        : fSocketA(inSocketA), fSocketB(inSocketB),fRefCount(0),fElem()
{fElem.SetEnclosingObject(this);}
```

- RTPPacketResender

  RTPPacketResender class to buffer and track re-transmits of RTP packets.
  the ctor copies the packet data, sets a timer for the packet's age limit and another timer for it's possible re-transmission.
  A duration timer is started to measure the RTT based on the client's ack.

  （1）、AddPacket()

  AddPacket adds a new packet to the resend queue. This will not send the packet.
  可以看到，在 RTPStream::ReliableRTPWrite 函数（针对 qtssRTPTransportTypeReliableUDP）里，会调用 AddPacket 登记要发送的数据以及到期 Drop 的时间。

  （2）、AckPacket()

  Acks a packet.
  可以看到，在 RTPStream::ProcessIncomingRTCPPacket 函数里，针对 qtssRTPTransportTypeReliableUDP 类型的传输，会调用 AckPacket 将对应包删除。

  （3）、ResendDueEntries()

  Resends outstanding packets in the queue。
  可以看到，在 RTPStream::ReliableRTPWrite 函数（针对 qtssRTPTransportTypeReliableUDP）里，会调用该函数，重发或者丢弃那些过期的 RTP 包。

- RTPStream

  Represents a single client stream (audio, video, etc).
  Control API is similar to overall session API.
  Contains all stream-specific data & resources, used by Session when it wants to send out or receive data for this stream.
  This is also the class that implements the RTP stream dictionary for QTSS API.

  （1）、Setup()

设置 fStreamURLPtr、fLateToleranceInSec、fTransportType、fNetworkMode
// decide whether to overbuffer
根据 fTransportType 决定是否调用 fSession->GetOverbufferWindow()->TurnOffOverbuffering()
// Check to see if this RTP stream should be sent over TCP.
if(fTransportType == qtssRTPTransportTypeTCP)
{ ... ...}
// This track is not interleaved, so let the session know that all tracks are not
// interleaved. This affects our scheduling of packets
fSession->SetAllTracksInterleaved(false);
// 注意对于播放一个静态媒体文件的 audio 流、video 流，客户端的 RTPPort、RTCPPort 端口号不
// 同。另外 RTP 端口号应该是偶数。
设置 fRemoteAddr、fRemoteRTPPort、fRemoteRTCPPort、sourceAddr。
// If the destination address is multicast, we need to setup multicast socket options on
// the sockets. Because these options may be different for each stream, we need a
// dedicated set of sockets
if (SocketUtils::IsMulticastIPAddr(fRemoteAddr))
{
重新调用 CreateUDPSocketPair 创建 socket 对，并设置 socket 属性。
} else
// 从 fUDPQueue 队列里根据本地的地址、端口号寻找 UDP Socket 对。
// 记得我们以前曾经创建过 udp socket 对。
fSockets = QTSServerInterface::GetServer()->GetSocketPool()->
GetUDPSocketPair(sourceAddr, 0, fRemoteAddr, fRemoteRTCPPort);
if(fTransportType == qtssRTPTransportTypeReliableUDP)
{ ... ... }
fLocalRTPPort = fSockets->GetSocketA()->GetLocalPort();
// 设置 UDPSocket::fDemuxer::fRemoteAddr、UDPSocket::fDemuxer::fRemotePort。同时调用
// UDPSocket::fDemuxer::fHashTable::Add 添加 RTPStream 对象(即：this)
fSockets->GetSocketB()->GetDemuxer()->RegisterTask(fRemoteAddr, fRemoteRTCPPort, this);

（2）、Write()
RTP 包的发送函数，QTSSFileModule::SendPackets 通过调用该函数来完成包的发送。
fSession->GetSessionMutex()->TryLock();
theTime = OS::Milliseconds();
// Data passed into this version of write must be a QTSS_PacketStruct
QTSS_PacketStruct* thePacket = (QTSS_PacketStruct*)inBuffer;
thePacket->suggestedWakeupTime = -1;
theCurrentPacketDelay = theTime - thePacket->packetTransmitTime; // 延时
// Empty the overbuffer window
// Update the bit rate value
// If this is the first write in a write burst, mark beginning.
... ...
if (inFlags & qtssWriteFlagsIsRTCP)
{ ... ...}
else if (inFlags & qtssWriteFlagsIsRTP)
{ // Check to see if this packet fits in the overbuffer window
thePacket->suggestedWakeupTime = fSession->GetOverbufferWindow()->CheckTransmitTime
(thePacket->packetTransmitTime, theTime, inLen);
if(thePacket->suggestedWakeupTime > theTime)
{ fSession->GetSessionMutex()->Unlock(); return QTSS_WouldBlock; }
// Check to make sure our quality level is correct. This function also tells us
// whether this packet is just too old to send
// 在 updateQualityLevel 里，只有当 inCurrentPacketDelay>fDropAllPacketsForThisStreamDelay
// 才会返回 false，表明数据被丢弃。跟配置文件中
// 的"drop_all_packets_delay"、"drop_all_video_delay"相关。
if (this->UpdateQualityLevel(thePacket->packetTransmitTime, theCurrentPacketDelay,

```
                                        theTime, inLen))
                {       // write out in interleave format on the RTSP TCP channel
                        if (fTransportType == qtssRTPTransportTypeTCP )
                                this->InterleavedWrite( thePacket->packetData, inLen, outLenWritten, fRTPChannel );
                        else if (fTransportType == qtssRTPTransportTypeReliableUDP )
                                // 调用 fResender 等处理
                                // 调用 fSockets->GetSocketA()->SendTo
                                // SocketA 即为完成 RTP Data 发送功能的 UDP Socket 端口。
                                this->ReliableRTPWrite(thePacket->packetData, inLen, theCurrentPacketDelay);
                        else if (inLen > 0)
                                fSockets->GetSocketA()->SendTo(fRemoteAddr, fRemoteRTPPort,
                                        thePacket->packetData, inLen);
                }
                if(err == QTSS_NoErr && inLen > 0)
                {
                        // Update statistics if we were actually able to send the data (don't
                        // update if the socket is flow controlled or some such thing)
                        ... ...
                }
        }
        fSession->GetSessionMutex()->Unlock();

         (3) 、ProcessIncomingRTCPPacket
        curTime = OS::Milliseconds();
        if (!fSession->GetSessionMutex()->TryLock())            return;
        // no matter what happens (whether or not this is a valid packet) reset the timeouts
        fSession->RefreshTimeout();
        if (fSession->GetRTSPSession() != NULL)    fSession->GetRTSPSession()->RefreshTimeout();
        while(currentPtr.Len > 0)
        {
                // Due to the variable-type nature of RTCP packets, this is a bit unusual... We
                // initially treat the packet as a generic RTCPPacket in order to determine its'
                // actual packet type.  Once that is figgered out, we treat it as its' actual
                // packet type
                RTCPPacket rtcpPacket;
                if (!rtcpPacket.ParsePacket((UInt8*)currentPtr.Ptr, currentPtr.Len))
                {       fSession->GetSessionMutex()->Unlock();     return;       }
                // Increment our RTCP Packet and byte counters for the session.
                fSession->IncrTotalRTCPPacketsRecv();
                fSession->IncrTotalRTCPBytesRecv( (SInt16) currentPtr.Len);
                switch (rtcpPacket.GetPacketType())
                {
                        case RTCPPacket::kReceiverPacketType:
                        {
                                RTCPReceiverPacket receiverPacket;
                                if (!receiverPacket.ParseReceiverReport((UInt8*)currentPtr.Ptr,
currentPtr.Len))
                                {       fSession->GetSessionMutex()->Unlock();     return;}
                                this-
>PrintPacketPrefEnabled(currentPtr.Ptr,currentPtr.Len,RTPStream::rtcpRR);
                                fClientSSRC = rtcpPacket.GetPacketSSRC();
                                fFractionLostPackets=receiverPacket.GetCumulativeFractionLostPackets();
                                fJitter = receiverPacket.GetCumulativeJitter();
                                UInt32 curTotalLostPackets = receiverPacket.GetCumulativeTotalLostPackets();
                                // Workaround for client problem.  Sometimes it appears to report a
                                // bogus lost packet count. Since we can't have lost more packets
                                // than we sent, ignore the packet if that seems to be the case.
```

```
                        if(curTotalLostPackets-fTotalLostPackets<=fPacketCount - fLastPacketCount)
                        {       设置 fCurPacketLostINRTCPInterval、fTotalLostPackets、
                                fPacketCountInRTCPInterval、fLastPacketCount。     }
                } break;
                case RTCPPacket::kAPPPacketType:
                {
                        Check and see if this is an Ack packet. If it is, update the UDP
                        Resender
                        If it isn't an ACK, assume its the qtss APP packet。
                }
                ... ...
        }
        currentPtr.Ptr += (rtcpPacket.GetPacketLength() * 4) + 4;
        currentPtr.Len -= (rtcpPacket.GetPacketLength() * 4) + 4;
}
// Invoke the RTCP modules, allowing them to process this packet
// 目前系统中的 QTSSProxyModule(This module is intended to be used in a stripped down
// version of QTSS/DSS in order to handle firewall proxy behaviour for the RTP
// protocol.)、QTSSFlowControlModule(Uses information in RTCP packers to throttle back
// the server when it's pumping out too much data to a given client)
调用注册 QTSS_RTCPProcess_Role 模块的处理函数。
fSession->GetSessionMutex()->Unlock();
```

- RTPSessionInterface

  （1）、构建函数

```
... ...
fTimeoutTask.SetTask(this); fUniqueID = (UInt32)atomic_add(&sRTPSessionIDCounter, 1);
fQualityUpdate = fUniqueID;
fTimeout = QTSServerInterface::GetServer()->GetPrefs()->GetRTPTimeoutInSecs() * 1000;
```

- RTPSession

```
RTPSession represents an, well, an RTP session. The server creates one of these when a
new client connects, and it lives for the duration of an RTP presentation. It contains
all the resources associated with that presentation, such as RTPStream objects. It also
provides an API for manipulating a session (playing it, pausing it, stopping it, etc)
It is also the active element, ie. it is the object that gets scheduled to send out &
receive RTP & RTCP packets
```

  （1）、AddStream()

```
// Create a new SSRC for this stream. This should just be a random number unique to all
// the streams in the session
创建一个和当前 session 所有 stream 的 fSsrc 不同的随机数。
*outStream = NEW RTPStream(theSSRC, this);
// 调用 RTPStream::Setup 函数
(*outStream)->Setup(request, inFlags);
// If the stream init succeeded, then put it into the array of setup streams
this->SetValue(qtssCliSesStreamObjects, this->GetNumValues(qtssCliSesStreamObjects),
                outStream, sizeof(RTPStream*), QTSSDictionary::kDontObeyReadOnly);
fHasAnRTPStream = true;
```

  （2）、Play()

```
// What time is this play being issued at?
fLastBitRateUpdateTime = fNextSendPacketsTime = fPlayTime = OS::Milliseconds();
if (fIsFirstPlay)     fFirstPlayTime = fPlayTime;
```

```
        fAdjustedPlayTime = fPlayTime - ((SInt64)(request->GetStartTime() * 1000));
        // for RTCP Srs, we alse need to store the play time in NTP
        fNTPPlayTime = OS::TimeMilli_To_1900Fixed64Secs(fPlayTime);
        // we are definitely playing now, so schedule the object.
        fState = qtssPlayingState;          fIsFirstPlay = false;
        // 调整 fNumRTPPlayingSessions
        QTSServerInterface::GetServer()-> AlterRTPPlayingSessions(1);
        // 设置 theWindowSize
        ... ...
        this->GetBandwidthTracker()->SetWindowSize(theWindowSize);
        this->GetOverbufferWindow()->ResetOverBufferWindow();
        // Go through all the streams, setting their thinning params.
        调用(*theStream)->SetThinningParams();
        // Set the size of the RTSPSession's send buffer to an appropriate max size based on the

        // bitrate of the movie. This has 2 benefits:
        // 1) Each socket normally defaults to 32K. A smaller buffer prevents the system from
        // getting buffer starved if lots of clients get flow-controlled
        // 2) We may need to scale up buffer sizes for high-bandwidth movies in order to maximize
        // thruput, and we may need to scale down buffer sizes for low-bandwidth movies to
        // prevent us from buffering lots of data that the client can't use

        // If we don't know any better, assume maximum buffer size.
        // 注意：这里设置的是 RTSPSession 的 socket，即 TCP socket。
        thePrefs = QTSServerInterface::GetServer()->GetPrefs();
        theBufferSize = thePrefs->GetMaxTCPBufferSizeInBytes();
        if (this->GetMovieAvgBitrate() > 0)
        {
                // We have a bit rate... use it.
                realBufferSize = this->GetMovieAvgBitrate() * thePrefs->GetTCPSecondsToBuffer();
                theBufferSize = (Uint32)realBufferSize >> 3;
                theBufferSize = this->PowerOf2Floor(theBufferSize);
                // This is how much data we should buffer based on the scaling factor... if it is
                // lower than the min, raise to min. Same deal for max buffer size
                // 缺省值为配置文件中的 max_tcp_buffer_size(200000)、min_tcp_buffer_size(8192)
                if (theBufferSize < thePrefs->GetMinTCPBufferSizeInBytes())
                        theBufferSize = thePrefs->GetMinTCPBufferSizeInBytes();
                if (theBufferSize > thePrefs->GetMaxTCPBufferSizeInBytes())
                        theBufferSize = thePrefs->GetMaxTCPBufferSizeInBytes();
        }
        fRTPSession->GetSocket()->SetSocketBufSize(theBufferSize);
        // 发送信号，RTPSession::Run 将会被运行。
        this->Signal(Task::kStartEvent);
```

（3）、Run

在 RTPSession::Play 函数调用 Signal 后，该函数会被运行

```
        ... ...
        // if we have been instructed to go away, then let's delete ourselves
        if ((events & Task::kKillEvent) || (events & Task::kTimeoutEvent) ||
(fModuleDoingAsyncStuff))
        {       ... ...}
        // if the stream is currently paused, just return without doing anything. We'll get woken
        // up again when a play is issued
        if ((fState == qtssPausedState) || (fModule == NULL))
                return 0;               // 返回的是 0!
        // Make sure to grab the session mutex here, to protect the module against RTSP requests
```

```
// coming in while it's sending packets
{       OSMutexLocker locker(&fSessionMutex);
        // just make sure we haven't been scheduled before our scheduled play time. If so,
        // reschedule ourselves for the proper time. (if client sends a play while we are
        // already playing, this may occur)
        theParams.rtpSendPacketsParams.inCurrentTime = OS::Milliseconds();
        if (fNextSendPacketsTime > theParams.rtpSendPacketsParams.inCurrentTime)
        // Send retransmits if we need to
        for (int streamIter = 0; this->GetValuePtr(qtssCliSesStreamObjects, streamIter,
                (void**)&retransStream, &retransStreamLen) == QTSS_NoErr; streamIter++)
                if (retransStream && *retransStream)
                        (*retransStream)->SendRetransmits();
        theParams.rtpSendPacketsParams.outNextPacketTime = fNextSendPacketsTime -
                        theParams.rtpSendPacketsParams.inCurrentTime;
} else {
        if ((theParams.rtpSendPacketsParams.inCurrentTime -
                        fLastBandwidthTrackerStatsUpdate) > 1000)
                this->GetBandwidthTracker()->UpdateStats();
        theParams.rtpSendPacketsParams.outNextPacketTime = 0;
        // Async event registration is definitely allowed from this role.
        fModuleState.eventRequested = false;
        // 这里的 fModule 是在 RTSPSession::Run 函数里调用
        // fRTPSession->SetPacketSendingModule 设置为找到的注册了 Role 的模块。
        // QTSSFileModule 注册了 QTSS_RTPSendPackets_Role 的处理。
        fModule->CallDispatch(QTSS_RTPSendPackets_Role, &theParams);
        // make sure not to get deleted accidently!
        if (theParams.rtpSendPacketsParams.outNextPacketTime < 0)
                theParams.rtpSendPacketsParams.outNextPacketTime = 0;
        fNextSendPacketsTime = theParams.rtpSendPacketsParams.inCurrentTime +
                        theParams.rtpSendPacketsParams.outNextPacketTime;
}
// Make sure the duration between calls to Run() isn't greater than the max retransmit
// delay interval.
// 分别对应配置文件中的“max_retransmit_delay(500)”、“send_interval(50)”项。
// max_retransmit_delay:    maximum interval between when a retransmit is supposed to be
//                          sent and when it actually gets sent. Lower values means
//                          smoother flow but slower server performance.
// send_interval:           the minimum time the server will wait between sending packet
//                          data to a client.
theRetransDelayInMsec = QTSServerInterface::GetServer()->GetPrefs()->GetMaxRetransmitDelayInMsec();
theSendInterval = QTSServerInterface::GetServer()->GetPrefs()->GetSendIntervalInMsec();
// We want to avoid waking up to do retransmits, and then going back to sleep for like, 1
// msec. So, only adjust the time to wake up if the next packet time is greater than the
// max retransmit delay + the standard interval between wakeups.
if (theParams.rtpSendPacketsParams.outNextPacketTime > (theRetransDelayInMsec + theSendInterval))
        theParams.rtpSendPacketsParams.outNextPacketTime = theRetransDelayInMsec;
return theParams.rtpSendPacketsParams.outNextPacketTime;        // 指定时间后被再度运行
```

- RTSPRequestStream

Provides a stream abstraction for RTSP. Given a socket, this object can read in data until an entire RTSP request header is available. (do this by calling ReadRequest). It handles RTSP pipelining (request headers are produced serially even if multiple headers arrive simultaneously), & RTSP request data.

（1）、ReadRequest

```
while(true)
```

```
{       Uint32 newOffset = 0;
        // If this is the case, we already HAVE a request on this session, and we now are
        // done with the request and want to move onto the next one. The first thing we
        // should do is check whether there is any lingering data in the stream. If there
        // is, the parent session believes that is part of a new request.
        if(fRequestPtr != NULL)
        {
        }
        // We don't have any new data, so try and get some
        if(newOffset == 0)
        {
                if(fRetreatBytes > 0)
                {
                        // This will be true if we've just snarfed another input stream, in
                        // which case the encoded data is copied into our request buffer,
                        // and its length is tracked in fRetreatBytes. If this is true, just
                        // fall through and decode the data.
                        newOffset = fRetreatBytes;  fRetreatBytes = 0;
                } else {
                        // We don't have any new data, get some from the socket...
                        // 注意我们的 socket 端口是 non blocking
                        sockErr = fSocket->Read(&fRequestBuffer[fCurOffset],
                                        (kRequestBufferSizeInBytes-fCurOffset) -1,
&newOffset);

                        // assume the client is dead if we get an error back
                        if(sockErr == EAGAIN)       return QTSS_NoErr;
                        if(sockErr != QTSS_NoErr)   return sockErr;
                }
                if(fDecode)
                {
                        base64 decode;
                } else fRequest.Len += newOffset;
                fCurOffset += newOffset;
        }
        // see if this is an interleaved data packet
        if ('$' == *(fRequest.Ptr))
        {
                if(fRequest.Len < 4)        continue;
                Uint16* dataLenP = (Uint16*)fRequest.Ptr;
                Uint32 interleavedPacketLen = ntohs(dataLenP[1]) + 4;
                if(interleavedPacketLen > fRequest.Len)    continue;
                // put back any data that is not part of the header
                fRetreatBytes += fRequest.Len - interleavedPacketLen;
                fRequest.Len = interleavedPacketLen;
                fRequestPtr = &fRequest;    fIsDataPacket = true;
                return QTSS_RequestArrived;
        }
        fIsDataPacket = false;
        if(fPrintRTSP)
                打印 RTSP 包
        // use a StringParser object to search for a double EOL, which signifies the end
        // of the header.
        weAreDone = false;   headerParser(&fRequest);    lcount = 0;
        while (headerParser.GetThruEol(NULL))
        {       ... ...}
        // weAreDone means we have gotten a full request
```

```
        // 应该是指收到最后一个消息头
        if(weAreDone)
        {
                // put back any data that is not part of the header
                fRequest.Len -= headerParser.GetDataRemaining();
                fRetreatBytes += headerParser.GetDataRemaining();
                fRequestPtr = &fRequest;
                return QTSS_RequestArrived;
        }
        // check for a full buffer
        if (fCurOffset == kRequestBufferSizeInBytes - 1)
        {
                fRequestPtr = &fRequest;
                return E2BIG;
        }
}
```

- RTSPResponseStream

  Object that provides a "buffered WriteV" service. Clients can call this function to write to a socket, and buffer flow controlled data in different ways.
  It is derived from StringFormatter, which it uses as an output stream buffer. The buffer may grow infinitely.
  （1）、构建函数
  RTSPResponseStream(TCPSocket* inSocket, TimeoutTask* inTimeoutTask)
  : ResizeableStringFormatter(fOutputBuf, kOutputBufferSizeInBytes),
  fSocket(inSocket), fBytesSentInBuffer(0), fTimeoutTask(inTimeoutTask),fPrintRTSP(false) {}

- RTSPRequest

  This class encapsulates a single RTSP request. It stores the meta data associated with a request, and provides an interface (through its base class) for modules to access request information, It divides the request into a series of states.
  State 1: At first, when the object is constructed or right after its Reset function is called, it is in an uninitialized state.
  State 2: Parse() parses an RTSP header. Once this function returns, most of the request--related query functions have been setup. (unless an error occurs)
  State 3: GetHandler() uses the request information to create the proper request Handler object for this request. After that, it is the Handler object's responsibilty to process the request and send a response to the client.
  RTSPRequest是RTSPRequestInterface的继承类。

- RTSPSession

  Represents an RTSP session , which I define as a complete TCP connection lifetime, from connection to FIN or RESET termination. This object is the active element that gets scheduled and gets work done. It creates requests and processes them when data arrives. When it is time to close the connection it takes care of that.
  class RTSPSession : public RTSPSessionInterface
  class RTSPSessionInterface : public QTSSDictionary, public Task

RTSPSession的状态机：

```
enum
{
        kReadingRequest             = 0,
        kFilteringRequest           = 1,
        kRoutingRequest             = 2,
        kAuthenticatingRequest      = 3,
```

```
        kAuthorizingRequest         = 4,
        kPreprocessingRequest       = 5,
        kProcessingRequest          = 6,
        kSendingResponse            = 7,
        kPostProcessingRequest      = 8,
        kCleaningUp                 = 9,

    // states that RTSP sessions that setup RTSP
    // through HTTP tunnels pass through
        kWaitingToBindHTTPTunnel = 10,  // POST or GET side waiting to be joined with it's matching half
        kSocketHasBeenBoundIntoHTTPTunnel = 11,  // POST side after attachment by GET side (its dying)
        kHTTPFilteringRequest = 12,             // after kReadingRequest, enter this state
        kReadingFirstRequest = 13,          // initial state - the only time we look for an HTTP tunnel
        kHaveNonTunnelMessage = 14      // we've looked at the message, and its not an HTTP tunnle message
    };
```

RTSP 的消息格式:
请求消息:

方法	URI	RTSP 版本	CR LF
消息头		CR LF	CR LF
消息体		CR LF	

回应消息:

RTSP 版本		状态码	解释 CR LF
消息头		CR LF	CR LF
消息体		CR LF	

（1）、构建函数
初始化成员

（2）、ParseProxyTunnelHTTP函数
检查是否是Http连接，如果是，返回true。

```
if it's an HTTP request parse the interesing parts from the request
- check for GET or POST, set fHTTPMethod
- checck for HTTP protocol, set fWasHTTPRequest
- check for SessionID header, set fProxySessionID char array
- check for accept "application/x-rtsp-tunnelled.

... ...
if ( fFoundValidAccept && *fProxySessionID  && fWasHTTPRequest )
        isHTTPRequest = true;
return isHTTPRequest;
```

（3）、PreFilterForHTTPProxyTunnel函数
"pre" filter the request looking for the HHTP Proxy tunnel HTTP requests, merge the 2 sessions into one, let the donor Session die.

```
// returns true if it's an HTTP request that can tunnel
if(!this->ParseProxyTunnelHTTP())  return QTSS_NoErr;
```

对http连接的处理，暂不做分析！

（4）、Run函数
当一个RTSP端口有数据时，EventContext::ProcessEvent函数会调用Signal函数，TaskThread会调用RTSPSession::Run函数。见TCPListenerSocket::ProcessEvent函数的分析。

```
OSThreadDataSetter theSetter(&fModuleState, NULL);
```

```
while (this->IsLiveSession())
{
	// RTSP Session state machine. There are several well defined points in an RTSP
	// request where this session may have to return from its run function and wait
	// for a new event. Because of this, we need to track our current state and return
	// to it.
	case kReadingFirstRequest:		// 初始状态
	{
		// 返回QTSS_NoErr意味着所有数据已经从Socket中读出，但尚不能构成一个完整
		// 的请求，因此必须等待更多的数据到达
		// 我们从ReadRequest代码可以知道，只有当对socket端口recv返回EAGAIN时，
		// ReadRequest才会返回QTSS_NoErr。
		if ((err = fInputStream.ReadRequest()) == QTSS_NoErr)
		{
			fInputSocketP->RequestEvent(EV_RE);		// 重新申请监听
			return 0;
		}
		if ((err != QTSS_RequestArrived) && (err != E2BIG))
		{	break;		}	// 注意：这里没有再申请继续监听。
		if (err == QTSS_RequestArrived)	fState = kHTTPFilteringRequest;
		// If we get an E2BIG, it means our buffer was overfilled. In that case, we
		// can just jump into the following state, and the code their does a check
		// for this error and returns an error.
		if (err == E2BIG)	fState = kHaveNonTunnelMessage;
	}	continue;	// 注意：这里调用continue

	case kHTTPFilteringRequest:
	{
		// assume it's not a tunnel setup message prefilter will set correct tunnel
		// state if it is.
		fState = kHaveNonTunnelMessage;
		// 对于非Http连接的情况，会返回QTSS_NoErr。但是返回QTSS_NoErr，并不表示
		// 一定是非Http连接。而且对于Http连接（RTSP经过Http代理）情况，
		// PreFilterForHTTPProxyTunnel可能会更改fState值。
		preFilterErr = this->PreFilterForHTTPProxyTunnel();
		if(preFilterErr == QTSS_NoErr)	continue;
		else	return -1;
	}

	case kHaveNonTunnelMessage:
	{
		fRequest = NEW RTSPRequest(this);
		fRoleParams.rtspRequestParams.inRTSPRequest = fRequest;
		fRoleParams.rtspRequestParams.inRTSPHeaders = fRequest->GetHeaderDictionary();

		// We have an RTSP request and are about to begin processing. We need to
		// make sure that anyone sending interleaved data on this session won't
		// be allowed to do so until we are done sending our response
		// We also make sure that a POST session can't snarf in while we're
		// processing the request.
		// 通过锁，防止在发送请求回复过程中，有另外的数据被安插发送
		fReadMutex.Lock();	fSessionMutex.Lock();

		// The fOutputStream's fBytesWritten counter is used to count the # of
		// bytes for this RTSP response. So, at this point, reset it to 0 (we can
		// then just let it increment until the next request comes in)
```

```
                fOutputStream.ResetBytesWritten();
                如果err == E2BIG或者QTSS_BadArgument，调用SendErrorResponse, fState赋值为
                kPostProcessingRequest。否则fState赋值为kFilteringRequest。
        }
        case kFilteringRequest:
        {
                // We received something so auto refresh, The need to auto refresh is
                // because the api doesn't allow a module to refresh at this point
                // 重置fTimeoutTask的到时时间
                fTimeoutTask.RefreshTimeout();

                // Before we even do this, check to see if this is a *data* packet, in
                // which case this isn't an RTSP request, so we don't need to go through
any
                // of the remaining steps
                // 在readRequest函数里，只有碰到“$”，才会认为是interleaved packet
                if (fInputStream.IsDataPacket())
                {
                        // 在handleIncomingDataPacket里，除了调用RTPSession的
                        // ProcessIncomingInterleavedData函数外，还会调用已注册
                        // QTSS_RTSPIncomingData_Role模块的处理函数。(系统的
                        // QTSSReflectorModule模块提供该Role的处理)
                        // 这里可以进行二次开发！
                        this->HandleIncomingDataPacket();
                        fState = kCleaningUp;       break;
                }
                设置filter param block theFilterParams;
                调用已注册QTSS_RTSPFilter_Role模块的处理函数(系统的QTSSRelayModule模块、
                QTSSMP3StreamingModule模块(Handle ShoutCast/IceCast-style MP3
                streaming.)、QTSSRefMovieModule模块(A module that serves an
                RTSP text ref movie from an HTTP request.)、QTSSAdminModule模块(A module
                that uses the information available in the server to present a web page
                containing that information.)、QTSSWebStatsModule模块(A module that uses
                the stats information available in the server to present a web page
                containing that information.)、QTSSWebDebugModule模块(A module that uses
                the debugging information available in the server to present a web page
                containing that information.)、QTSSDemoSMILModule模块(?)、
                QTSSHttpFileModule模块(A module for HTTP file transfer of files and for
                on-the-fly ref movie creation.)提供该Role的处理)，可以添加二次开发模块！
                // 根据fBytesWritten来判断，在StringFormatter::Put函数里增加了
                // fBytesWritten计数，但是实际上并没有通过socket发送？？？
                if (fRequest->HasResponseBeenSent())
                {       fState = kPostProcessingRequest;    break; }
                if (fSentOptionsRequest && this->ParseOptionsResponse())
                {       ... ...fState = kSendingResponse;   break; }
                else    //this is a normal request,so parse it and get the  RTPSession.
                        // 建立用于管理数据传输的RTPSession类对象，根据请求消息中的方法
                        // 做处理。
                        this->SetupRequest();
                if (fRequest->HasResponseBeenSent())
                {       fState = kPostProcessingRequest;    break; }
                fState = kRoutingRequest;
        }
        case kRoutingRequest:
        {
                调用注册了QTSS_RTSPRoute_Role处理的模块的处理函数(系统的QTSSRelayModule模
```

块、QTSSReflectorModule模块、QTSSHomeDirectoryModule模块提供了该Role的处理)。

```
// SetupAuthLocalPath must happen after kRoutingRequest and before
// kAuthenticatingRequest placed here so that if the state is shifted to
// kPostProcessingRequest from a response being sent then the
// AuthLocalPath will still be set.
if (fRequest->HasResponseBeenSent())
{       fState = kPostProcessingRequest;    break; }
if(fRequest->SkipAuthorization())
{       ... ...}
else    fState = kAuthenticatingRequest;
}
case kAuthenticatingRequest:
{
        QTSS_RTSPMethod method = fRequest->GetMethod();
        if (method != qtssIllegalMethod) do
        {       ... ...} while(false);
        if(fRequest->GetAuthScheme() == qtssAuthNone)
        {       ... ...}
```

调用二次开发的安全模块(QTSS_RTSPAuthenticate_Role)，主要用于客户身份验证以及其他规则的处理(系统的QTSSAccessModule模块(Module that handles authentication and authorization independent of the file system)、QTSSODAuthModule模块(This is a modified version of the QTSSAccessModule also released with QTSS 2.0. It has been modified to shrink the linespacing so that the code can fit on slides. Also, this module issues redirects to an error movie.)提供了该Role的处理)。

```
        // 有两种认证策略: basic authentication、digest authentication
        // If authenticaton failed, set qtssUserName in the qtssRTSPReqUserProfile
        // attribute to NULL and clear out the password and any groups that have
        // been set.
        this->CheckAuthentication();
        if (fRequest->HasResponseBeenSent())
        {       fState = kPostProcessingRequest;    break; }
        fState = kAuthorizingRequest;
}
case kAuthorizingRequest:
{
```

调用注册QTSS_RTSPAuthorize_Role模块的处理函数，如果失败，则发送回复并跳出循环(系统的QTSSSpamDefenseModule模块(Protects the server against denial-of-service attacks by only allowing X number of RTSP connections from a certain IP address)、QTSSFilePrivsModule模块(Module that handles and file system authorization)、QTSSReflectorModule模块、QTSSHomeDirectoryModule模块、QTSSAccessModule模块、QTSSAdminModule模块、QTSSDemoModule模块(This is a modified version of the QTSSAccessModule also released with QTSS 2.0. It has been modified to shrink the linespacing so that the code can fit on slides. Also, this module issues redirects to an error movie.)、QTSSODAuthModule模块提供了该Role的处理)。

```
        this->SaveRequestAuthorizationParams(fRequest);
        if(!allowd)   // 怎么会进入这个分支？？？
```

```
        {       ... ...}
        if (fRequest->HasResponseBeenSent())
        {       fState = kPostProcessingRequest;    break; }
        fState = kPreprocessingRequest;
        ... ...
    }
    case kPreprocessingRequest:
    {
        调用注册QTSS_RTSPPreProcessor_Role模块的处理函数，注意系统也提供了支持这
        些role的模块(系统的QTSSReflectorModule模块、QTSSSplitterModule模块、
        QTSSRTPFileModule模块(Content source module that uses the QTFileLib to
        serve Hinted QuickTime files to clients.)、QTSSRawFileModule模块(A module
        that returns the entire contents of a file to the client. Only does this if
        the suffix of the file is .raw)提供了该Role的处理)。
        注意：QTSSRTPFileModule和QTSSRawFileModule并没有加载进系统！！！
        if (fRequest->HasResponseBeenSent())
        {       fState = kPostProcessingRequest;    break; }
        fState = k```;
    }
    case kProcessingRequest:
    {
        调用注册QTSS_RTSPRequest_Role模块的处理函数，注意系统也提供了支持这
        些role的模块(系统的QTSSFileModule模块(Content source module that uses the
        QTFileLib to serve Hinted QuickTime files to clients.)、QTSSProxyModule模块
        提供了该Role的处理)。
        fState = kPostProcessingRequest;
    }
    case kPostProcessingRequest:
    {
        调用注册QTSS_RTSPPostProcessor_Role模块的处理函数，注意系统也提供了支持这
        些role的模块(系统的QTSSRelayModule模块、QTSSAccessLogModule模块提供了该
        Role的处理)。
        fState = kSendingResponse;
    }
    case kSendingResponse:
    {
        ... ...
        // 调用fSocket->Send，将在fOutputStream中尚未发出的请求响应通过Socket端
        // 口完全发送出去，如果还有数据没有发送出去，返回EAGAIN。
        err = fOutputStream.Flush();
        if (err == EAGAIN)
        {
            // If we get this error, we are currently flow-controlled and
should
            // wait for the socket to become writeable again
            fSocket.RequestEvent(EV_WR);
            // 因为前面执行了“fReadMutex.Lock(); fSessionMutex.Lock();”以禁止
在
            // 处理请求的过程中，有另外的数据在同一个session中发送。
            // 所以这里指定让后续的处理继续由同一个线程完成。
            this->ForceSameThread();
            return 0;
        } else if (err != QTSS_NoErr)           break;
        fState = kCleaningUp;
    case kCleaningUp:
    {
```

```
                // Cleaning up consists of making sure we've read all the incoming Request
                // Body data off of the socket
                if (this->GetRemainingReqBodyLen() > 0)
                        把socket里的数据读取出来并丢弃
                // If we've gotten here, we've flushed all the data. Cleanup, and wait for
                // our next request!
                // 置空fRTPSession、fRequest! 调用fSessionMutex.Unlock()、fReadMutex.Unlock();
                this->CleanupRequest();
                fState = kReadingRequest;    // 注意，这里处于一个while循环
        }
        case kReadingRequest
        {
                类似于 kReadingFirstRequest，如果socket有数据则将fState设为
                kHaveNonTunnelMessage，否则则调用RequestEvent继续申请监听。
        }
}
this->CleanupRequest();
return 0;
```

（5）、SetupRequest()

```
        首先调用fRequest->Parse分析RTSP请求
        // find the RTP session for this request
        this->FindRTPSession(theMap);

        // let's also refresh RTP session timeout so that it's kept alive in sync with the
        // RTSP session.
        if(fRTPSession != NULL)      fRTPSession->RefreshTimeout();

        // 在上面的Parse函数里已经调用RTSPProtocol::GetMethod设置fMethod
        // 如果是OPTIONS，将系统支持的方法添加到头部（在DoInitRole里已经把各个模块登记的
        // 支持方法汇总了）
        // 注意这里并没有实际的通过socket发送出去，只是和fOutputStream相关联
        if(fRequest->GetMethod() == qtssOptionsMethod)
        {       ... ...}

        // If this is a SET_PARAMETER request, don't let modules see it.
        if (fRequest->GetMethod() == qtssSetParameterMethod)
        {       ... ...}

        // If this is a DESCRIBE request, make sure there is no SessionID. This is not
        // allowed, and may screw up modules if we let them see this request.
        if (fRequest->GetMethod() == qtssDescribeMethod)
        {
                if (fRequest->GetHeaderDictionary()->GetValue(qtssSessionHeader)->Len > 0)
                {       SendErrorResponse(); return;        }
        }
        if(fRTPSession == NULL)      this->CreateNewRTPSession(theMap);
        // If it's a play request and the late tolerance is sent in the request use this
        // value
        if ((fRequest->GetMethod() == qtssPlayMethod) && (fRequest->GetLateToleranceInSec() != -1))
                // 这个函数做什么处理？？？
                fRTPSession->SetStreamThinningParams(fRequest->GetLateToleranceInSec());
        // Check to see if this is a "ping" PLAY request (a PLAY request while already
        // playing with no Range header). If so, just send back a 200 OK response and do
        // nothing. No need to go to modules to do this, because this is an RFC documented
```

```
// behavior
if ((fRequest->GetMethod() == qtssPlayMethod) && (fRTPSession->GetSessionState() == qtssPlayingState)
                && (fRequest->GetStartTime() == -1) && (fRequest->GetStopTime() == -1))
{       fRequest->SendHeader();      fRTPSession->RefreshTimeout();      return;}

// setup Authorization params;
fRequest->ParseAuthHeader();
```

（6）、CreateNewRTPSession

This is a brand spanking new session. At this point, we need to create a new RTPSession object that will represent this session until it completes. Then, we need to pass the session onto one of the modules

```
// First of all, ask the server if it's ok to add a new session
// 根据几个情况来判断是否可以添加新的RTPSession。
// 1、server的状态；2、最大连接数限制；3、最大带宽限制。
this->IsOkToAddNewRTPSession();

// create the RTPSession object
fRTPSession = NEW RTPSession();
// Lock the RTP session down so that it won't delete itself in the unusual event
// there is a timeout while we are doing this.
OSMutexLocker locker(fRTPSession->GetSessionMutex());
// Because this is a new RTP session, setup some dictionary attributes pertaining
// to RTSP that only need to be set once
this->SetupClientSessionAttrs();
// generate a session ID for this session
while (activationError == EPERM)
{
        // 利用随机数生成SessionID
        fLastRTPSessionIDPtr.Len = this->GenerateNewSessionID(fLastRTPSessionID);
        // ok, some module has bound this session, we can activate it. At this
        // point, we may find out that this new session ID is a duplicate. If
that's
        // the case, we'll simply retry until we get a unique ID
        // 该函数执行了以下操作：
        // 1. 拷贝到fRTSPSessionIDBuf,
        // 2. puts the session into the RTPSession Map:
        //      theServer->GetRTPSessionMap()->Register(&fRTPMapElem);
        // 3. 调用IncrementTotalRTPSessions()
        activationError = fRTPSession->Activate(fLastRTPSessionID);
}
```

（7）、QTSSModuleUtils::SendErrorResponse

```
... ...
if (sEnableRTSPErrorMsg)
{
        准备要输出的Error Message
        // 实际上是调用(RTSPRequestInterface*)inRequest)->AppendHeader函数
        // 在该函数里，调用RTSPRequetInterface::fOutputStream(RTSPResponseStream类
        // 对象)的Put函数关联Error Message
        QTSS_AppendRTSPHeader(inRequest, qtssContentLengthHeader, buff, ::strlen(buff));
}
// send the response header. In all situations where errors could happen, we don't
// really care, cause there's nothing we can do anyway!
// 实际上是调用(RTSPRequestInterface*)inRequest)->SendHeader函数
```

```
        // 在该函数里调用WriteStandardHeaders函数，添加标准的RTSP回复头部。
        QTSS_SendRTSPHeaders(inRequest);

        // Now that we've formatted the message into the temporary buffer, write it out to
        // the request stream and the Client Session object
        // 奇怪！！！始终只是调用RTSPRequetInterface::fOutputStream(RTSPResponseStream类
        // 对象)的Put函数关联Error Message，并没有写具体的socket端口！！！
        QTSS_Write(inRequest, theErrorMsgFormatter.GetBufPtr(), theErrorMsgFormatter.GetBytesWritten(), NULL,
0);
        ... ...
```

- RTSPSessionInterface

  Presents an API for session-wide resources for modules to use. Implements the RTSP Session dictionary for QTSS API.

  （1）、构建函数

```
:   QTSSDictionary(QTSSDictionaryMap::GetMap(QTSSDictionaryMap::kRTSPSessionDictIndex)),
    Task();
    // 对应于配置文件中的real_rtsp_timeout，缺省值为 180
    fTimeoutTask(NULL,QTSServerInterface::GetServer()->GetPrefs()->GetRealRTSPTimeoutInSecs() *
    1000),
    fInputStream(&fSocket), fOutputStream(&fSocket, &fTimeoutTask),
    fSocket(NULL, Socket::kNonBlockingSocketType), fOutputSocketP(&fSocket),
    fInputSocketP(&fSocket),
    ... ...
{
        fTimeoutTask.SetTask(this); fSocket.SetTask(this);              fStreamRef = this;
        ... ...
}
```

- RTSPListenerSocket

  属于 RTSP 子系统。基于 TCPListenerSocket 子类。

  （1）GetSessionTask

```
// when the server is behind a round robin DNS, the client needs to know the IP address
// of the server so that it can direct the "POST" half of the connection to the same
// machine when tunnelling RTSP thru HTTP
doReportHTTPConnectionAddress = QTSServerInterface::GetServer()->GetPrefs()->GetDoReportHTTPConnectionAddress();
theTask = NEW RTSPSession(doReportHTTPConnectionAddress);
// 返回 fSocket 成员
*outSocket = theTask->GetSocket(); // out socket is not attached to a unix socket yet.
// 根据配置文件中的 maximum_connections(缺省为 1000)、fNumRTSPSessions、
// fNumRTSPHTTPSessions（这两个变量值在 RTSPSession 的构建、析构函数、
PreFilterForHTTPProxyTunnel 函数里变化）来计算是否超过极限。
if(this->OverMaxConnections(0))     this->SlowDown();     // fSleepBetweenAccepts = true
else                                this->RunNormal();    // fSleepBetweenAccepts = flase
```

- TCPListenerSocket

  处理基于 TCP 协议的 socket 传输。它的继承情况为：

```
class TCPListenerSocket : public TCPSocet, public IdleTask
        class IdleTask : public Task
        class TCPSocket : public Socket
                class Socket : pulic EventContext
```

  下面是一些成员函数的分析：

  （1）构建函数

```
TCPListenerSocket() : TCPSocket(NULL, Socket::kNonBlockingSocketType),
IdleTask(),fAddr(0),
```

```
                        fPort(0), fOutOfDescriptors(false), fSleepBetweenAccepts(false)
    { this->SetTaskName("TCPListenerSocket"); }
```

（2）Initialize

创建打开流套接字（SOCK_STREAM）端口，并绑定 IP 地址、端口，执行 listen 操作。

注意在这个函数里调用了 SetSocketRcvBufSize 成员函数，以设置这个 socket 的接收缓冲区大小为 96K。而内核规定该值最大为 sysctl rmem max，可以通过/proc/sys/net/core/rmem default rmem max 来了解缺省值和最大值。（**注意 proc 下的这两个值也是可写的，可以通过调整这些值来达到优化 TCP/IP 的目的**。例如：http://www.linuxsky.org/doc/admin/200705/53.html）

（3）ProcessEvent

在 fListeners 申请监听流套接字端口后，一旦 socket 端口有数据，该函数会被调用。

这个函数的流程是这样的：

首先通过 accept 获得客户端的地址以及服务器端新创建的 socket 端口。

通过 GetSessionTask 创建一个 RTSPSession 的类对象<每个 RTSPSession 对象对应一个 RTSP 连接>，并将新创建的 socket 端口描述符、sockaddr 信息保存到 RTSPSession 的 TCPSocket 类型成员 fSocket。同时调用 fSocket::SetTask 和 RequestEvent(EV_RE)，这样 EventThread 在监听到这个

socket 端口有数据时，会调用 EventContext::processEvent 函数，在这个函数里会调用 fTask->Signal(Task::kReadEvent)，最终 TaskThread 会 调用 RTSPSession::Run 函数。

而 TCPListenerSocket 自己的 socket 端口会继续被申请监听。

```
int osSocket = accept(fFileDesc, (struct sockaddr*)&addr, &size);
if(osSocket == -1)
{
    如果 errno 为 EAGAIN，则再次调用 RequestEvent;
    ... ...                 // 其它 error 的处理，包括 Task 发送 kKillEvent 信号
}
theTask = this->GetSessionTask(&theSocket);
// set options on the socket. we are a server, always disable nagle algorithm
::setsockopt(osSocket, IPPROTO_TCP, TCP_NODELAY, (char*)&one, sizeof(int));
::setsockopt(osSocket, SOL_SOCKET, SO_KEEPALIVE, (char*)&one, sizeof(int));
sndBufSize = 96L * 1024L;
::setsockopt(osSocket, SOL_SOCKET, SO_SNDBUF, (char*)&sndBufSize, sizeof(int));

// setup the socket. When there is data on the socket, theTask will get an kReadEvent
// event
// TCPSocket::Set 函数通过 getsockname 将 osSocket 描述字对应的 sockaddr 保存到
// TCPSocket::fLocalAddr，并设置 TCPSocket:: fState |= kBound | kConnected。
// 同时将 osSocket 保存为 TCPSocket::fFileDesc
theSocket->Set(osSocket, &addr);
theSocket->InitNonBlocking(osSocket);
// 实际上是调用 EventContext::SetTask
theSocket->SetTask(theTask);
theSocket->RequestEvent(EV_RE);

// 如果 RTSPSession、HTTPSession 的连接数超过超过限制，则利用 IdleTaskThread 定时调用
// Signal 函数，在 TCPListenerSocket::Run 函数里会调用 TCPListenerSocket::ProcessEvent 函数
// 执行 accept(接收下一个连接)和 RequestEvent(继续申请监听)。
if(fSleepBetweenAccepts)    this->SetIdleTimer(kTimeBetweenAcceptsInMsec);
// sleep until there is a read event outstanding (another client wants to connect)
else                        this->RequestEvent(EV_RE);
```

- Socket

  Socket 基于 EventContext 类。

（1）、构建函数

```
Socket::Socket(Task *notifytask, UInt32 inSocketType)
:   EventContext(EventContext::kInvalidFileDesc, sEventThread), fState(inSocketType),
    fLocalAddrStrPtr(NULL),fLocalDNSStrPtr(NULL), fPortStr(fPortBuffer,
kPortBufSizeInBytes)
{
        fLocalAddr.sin_addr.s_addr = 0;     fLocalAddr.sin_port = 0;
        fDestAddr.sin_addr.s_addr = 0;      fDestAddr.sin_port = 0;
        // SetTask 是 EventContext 的成员函数，执行”fTask = notifytask”操作。
        this->SetTask(notifytask);
}
```

（2）、Initialize

```
sEventThread = new EventThread();
```

（3）、StartThread

```
sEventThread->Start();              //      启动线程
```

- TaskThreadPool

任务线程池类，该类负责生成、删除以及维护内部的任务线程列表。下面是一些成员函数的分析：

（1）AddThreads

根据 numToAdd 参数创建 TaskThread 类对象，并调用该类的 Start 成员函数。

将该类对象指针保存到 sTaskThreadArray 数组。

- TaskThread

任务线程类，基于 OSThread 类。下面是一些成员函数的分析：

（1）OSThread::Start

调用 pthread_create 创建一个线程，该线程的入口函数为 OSThread::_Entry 函数。

注意：对于有_POSIX_THREAD_PRIORITY_SCHEDULING 宏定义的 pthread 库，创建线程的时候，应该添加属性 PTHREAD_SCOPE_SYSTEM，但是这里被注释掉了！！！

（2）OSThread::_Entry

```
theThread->Entry()    //Entry 函数是 OSThread 的纯虚函数，所以这里实际上会调用派生类的 Entry
函数，也就是 TaskThread::Entry 函数。
```

注意：在这里调用了 pthread_setspecific 将 theThread 和线程的 key 相关联，而实际上 theThread 是线程创建的时候传进来的一个参数，对于每个线程来说都是不一样的数据，有必要要和 key 相关联吗??? 只有对于那些可能多个线程都会访问改写的全局量才有必要关联线程的私有数据。

（3）TaskThread::WaitForTask

该函数同样由一个大循环构成。等待任务的通知到达，或者因 stop 的请求而返回。

```
theCurrentTime = OS::Milliseconds();                //返回当前系统的时间（以 ms 表示）

//如果堆里有时间记录，并且这个时间<=系统当前时间（说明任务的运行时间已经到了），则返回//
该记录所对应的任务对象
//PeekMin 获得堆中的第一个元素（但并不取出）
if ((fHeap.PeekMin() != NULL) && (fHeap.PeekMin()->GetValue() <= theCurrentTime))
        return (Task*)fHeap.ExtractMin()->GetEnclosingObject();

//如果堆里有时间记录，则计算出还剩下的到时时间，如果小于 10ms，则设成 10ms
if (fHeap.PeekMin() != NULL)
        theTimeout = fHeap.PeekMin()->GetValue() - theCurrentTime;
```

```
//对于 theTimeout 等于 0 的情况，可以考虑将 theTimeout 设为更大的值，
//这样在没有事件的情况下，可以避免线程过于频繁的睡眠唤醒，而且也能够响应 stop 请求。
if (theTimeout < 10)        theTimeout = 10;

//等待事件发生
//TaskThread 类有一个 OSQueue_Blocking 类的私有成员 fTaskQueue。
//等待队列里有任务插入并将其取出返回。
//如果返回非空，则返回该队列项所对应的任务对象。
OSQueueElem* theElem = fTaskQueue.DeQueueBlocking(this, (SInt32) theTimeout);
if (theElem != NULL) return (Task*)theElem->GetEnclosingObject();

//如果有 stop 的请求，则退出
if (OSThread::GetCurrent()->IsStopRequested())    return NULL;
```

（4）TaskThread::Entry

这个函数是任务线程的入口函数，由一个大循环构成。

```
// 等待任务的通知到达，或者因 stop 的请求而返回（目前，WaitForTask 只有在收到 stop 请求后
// 才返回 NULL）。
theTask = this->WaitForTask();
if (theTask == NULL || false == theTask->Valid())        return;

Bool16 doneProcessingEvent = false;
// 下面也是一个循环，如果 doneProcessingEvent 为 true 则跳出循环。
// OSMutexWriteLocker、OSMutexReadLocker 均基于 OSMutexReadWriteLocker 类，在下面的使用
// 中这两个类构建函数会调用 sMutexRW->LockRead 或 sMutexRW->LockWrite 进行互斥的读写操作。
// OSMutexRW 类对象 sMutexRW 为 taskThreadPool 类的成员，所以是对所有的线程互斥。
while (!doneProcessingEvent)
{
        theTask->fUseThisThread = NULL;
        if (theTask->fWriteLock) {
                OSMutexWriteLocker mutexLocker(&TaskThreadPool::sMutexRW);
                theTimeout = theTask->Run();
                theTask->fWriteLock = false;
        } else {
                OSMutexReadLocker mutexLocker(&TaskThreadPool::sMutexRW);
                theTimeout = theTask->Run();
        }

        如果 theTimeout < 0，
                则说明任务结束，任务对象被销毁，doneProcessingEvent 设为 true，在后面调用
                ThreadYield 后(对于 linux 系统来说，ThreadYield 实际上没有做什么工作)，跳出这
                个循环。再调用 WaitForTask，等待下个处理。
        如果 theTimeout == 0，
                将 theTask->fEvents 和 Task::kAlive 比较，如果返回 1，则说明该任务已经没有事
                件处理，反之则在 ThreadYield 返回后继续在循环里处理这个任务的事件。
                doneProcessingEvent = compare_and_store(Task::kAlive, 0, &theTask->fEvents);
                // 虽然该任务目前没有事件处理，但是并不表示要销毁，所以没有 delete。
                if(doneProcessingEvent)      theTask = NULL;
        如果 theTimeout > 0，
                则说明任务希望等待 theTimeout 时间后得到处理。
                theTask->fTimerHeapElem.SetValue(OS::Milliseconds() + theTimeout);
                // 将该任务的 fTimerHeapElem 插入 fHeap。
                // 在下次的 WaitForTask 运行时，一旦发现 fHeap 有记录，就会做特别处理。
```

```
                fHeap.Insert(&theTask->fTimerHeapElem);
                // 给这个任务又添加了一个 kIdleEvent???
                atomic_or(&theTask->fEvents, Task::kIdleEvent);
                doneProcessingEvent = true;
        this->ThreadYield(); // 对于 linux 系统来说，ThreadYield 实际上没有做什么工作
}
```

- Task

（1）Signal

如果该任务有指定处理的线程，则将该任务加入到指定线程的任务队列中。
如果没有指定的线程，则从**线程池**中随机选择一个任务线程，并将该任务加入到这个任务线程的任务队列中。
或者干脆就没有线程运行，那么只是打印信息退出。
这里随机选择线程的方法值得商榷，目前的做法是轮流选择。其实对于已经发送过信号的任务来说，可以记录选择的线程编号，这样下次可以调度同一个线程运行。需要注意线程任务均衡。
另外，是否线程池里的线程越多，效率就越高？？？
线程类的 fTaskQueue.enqueue 函数加入的是&fTaskQueueElem，而在 Task 的构建函数里，调用了 fTaskQueueElem.SetEnclosingObject(this)，这样线程类在处理这个事件的时候，就会知道对应的具体任务，并通过任务的 GetEvents 成员函数来获取待处理的事件类型。
fTaskQueueElem 是 OSQueueElem 的类对象，很多涉及到队列操作的类，都会有这样的成员。通过这个成员，可以把具体的类和队列项相关联。

有一点需要注意的是，因为是通过或操作，如果该线程的任务队列里已经有了一个事件待处理，那么该事件不会被添加，也就是说在事件被处理之前，任务只会被调度一次。

- TimeoutTask

TimeoutTask.h 文件说明:
Just like a normal task, but can be scheduled for timeouts. Unlike IdleTask, which is VERY aggressive about being on time, but high overhead for maintaining the timing information, this is a low overhead, low priority timing mechanism. Timeouts may not happen exactly when they are supposed to, but who cares?
下面是一些成员函数的分析：

（1）Initialize

```
sThread = NEW TimeoutTaskThread();
//TimeoutTaskThread 是 IdleTask 的派生类，IdleTask 是 Task 的派生类。
sThread->Signal(Task::kStartEvent);
```

（2）TimeoutTask::TimeoutTask(Task* inTask, Sint64 inTimeoutInMilSecs)

```
        : fTask(inTask), fQueueElem()       // 构建函数
    fQueueElem.SetEnclosingObject(this);
    this->SetTimeout(inTimeoutInMilSecs);
    if (NULL == inTask)          fTask = (Task *) this;
    OSMutexLocker locker(&sThread->fMutex);
    sThread->fQueue.EnQueue(&fQueueElem);
```

- TimeoutTaskThread

实际上这个类的名字容易产生混淆，它并不是一个线程类，而是一个基于Task类的任务类，它配合TimeoutTask类实现一个周期性运行的基础任务。
这部分的架构是这样的：
在startServer里，调用了TimeoutTask::Initialize()。在这个函数里创建了一个TimeoutTaskThread类对象，付给全局量sThread，同时sThread调用Signal，让自己得到运行。DSS别的模块在需要完成自己的周期性工作时，通过TimeoutTask类的成员把自己的相关信息插入sThread的队列。任务线程在周期性运行TimeoutTaskThread::Run函数时，会处理sThread队列内的所有任务，从而完成各种周期性工作。

注意：在TaskThread::Entry将TimeoutTaskThread项从线程的fTaskQueue里取出处理后，根据Run返回值，决定是否插入线程的fHeap，而不会再次插入到fTaskQueue里。如果插入fHeap，在TaskThread::WaitForTask里会被得到处理。

（1）Run

这个函数是任务类的处理函数。线程一旦捕获到任务的事件后，会调用该任务的Run函数进行处理。

```
OSMutexLocker locker(&fMutex);
intervalMilli = kIntervalSeconds * 1000;  taskInterval = intervalMilli;

在timeoutTaskThread的fQueue队列里获取TimeoutTask类对象。（在TimeoutTask类的构建函数里曾经将自己插入到TimeoutTaskThread的fQueue队列。）
for (OSQueueIter iter(&fQueue); !iter.IsDone(); iter.Next()) {
        TimeoutTask* theTimeoutTask = (TimeoutTask*)iter.GetCurrent()->GetEnclosingObject();
        // if it's time to time this task out, signal it
        // fTask是TimeoutTask类对象在创建的时候，传进来的具体任务。
        // 如果发现有某个任务到期了，通过Signal通知线程。
        if ((theTimeoutTask->fTimeoutAtThisTime > 0) && (curTime >=
                                        theTimeoutTask->fTimeoutAtThisTime))
                theTimeoutTask->fTask->Signal(Task::kTimeoutEvent);
        else {
                taskInterval = theTimeoutTask->fTimeoutAtThisTime - curTime;
                if ((taskInterval > 0) && (theTimeoutTask->fTimeoutInMilSecs > 0) &&
                                        (intervalMilli > taskInterval))
                        intervalMilli = taskInterval + 1000;
                        // 为什么多加 1s，这样任务岂不是被延迟了 1s ？？？
        }
        this->GetEvents();   // clear the event mask!
        OSThread::ThreadYield();     // 这里的yield操作岂不是和TaskThread::Entry的重复？？
    return intervalMilli;// 返回下一次的timeout值。Task::Entry根据这个返回值做后续处理。
```

- RTCPTask

基于Task类。This task handles all incoming RTCP data. It just polls, so make sure to start the polling process by signalling a start event.

（1）、构建函数

```
RTCPTask() : Task() {this->SetTaskName("RTCPTask"); this->Signal(Task::kStartEvent); }
```

（2）、Run函数

在运行StartTasks创建RTCPTask类后，该Run函数就会被调度运行。

后续当监听到用来完成RTCP Data接收的Udp socket端口有数据时，EventContext::ProcessEvent调用Signal(Task::kReadEvent)，该函数会被运行。

```
const Uint32 kMaxRTCPPacketSize = 2048;   char thePacketBuffer[kMaxRTCPPacketSize];
StrPtrLen thePacket(thePacketBuffer, 0);
QTSServerInterface* theServer = QTSServerInterface::GetServer();

// This task goes through all the UDPSockets in the RTPSocketPool, checking to see if
// they have data. If they do, it demuxes the packets and sends the packet onto the
// proper RTP session.
EventFlags events = this->GetEvents();     // get and clear events
if((events & Task::kReadEvent) || (events & Task::kIdleEvent))
{
        遍历SocketPool中的socket对中的每一个socket，如果这个socket有UDPDemuxer(在RTPSocketPool::ConstructUDPSocketPair函数里，用来接收RTCP数据的Socket B就带有Demuxer)，则将端口中接收到的数据发给UDPDemuxer对应的RTPStream，并调用RTPStream
```

```
        的ProcessIncomingRTCPPacket函数。
        在RTPStream的Setup函数里，曾经调用进行注册：
        fSockets->GetSocketB()->GetDemuxer()->RegisterTask(fRemoteAddr, fRemoteRTCPPort, this);
    }
    return 0;
```

- RTPStatsUpdaterTask

基于Task类。This class runs periodically to compute current totals & averages，同时如果发现RTP流量超过限制，会考虑是否发送kKillEvent信号给最近的会话。

（1）、构建函数

```
this->SetTaskName("RTPStatsUpdaterTask");
this->Signal(Task::kStartEvent);
```

（2）、Run函数

在运行StartTasks创建RTPStatsUpdaterTask类后，该Run函数就会被调度运行。
后续通过调用Task::Signal，该函数会被运行。

```
QTSServerInterface* theServer = QTSServerInterface::sServer;
// All of this must happen atomically wrt dictionary values we are manipulating
OSMutexLocker locker(&theServer->fMutex);

// First update total bytes. This must be done because total bytes is a 64 bit number,
// so no atomic functions can apply.
// 这种做法有什么意义呢？？？因为在一开始就获得了锁
unsigned int periodicBytes = theServer->fPeriodicRTPBytes;
(void)atomic_sub(&theServer->fPeriodicRTPBytes, periodicBytes);
theServer->fTotalRTPBytes += periodicBytes;

// Same deal for packet totals
unsigned int periodicPackets = theServer->fPeriodicRTPPackets;
(void)atomic_sub(&theServer->fPeriodicRTPPackets, periodicPackets);
theServer->fTotalRTPPackets += periodicPackets;

// ... and for lost packet totals
unsigned int periodicPacketsLost = theServer->fPeriodicRTPPacketsLost;
(void)atomic_sub(&theServer->fPeriodicRTPPacketsLost, periodicPacketsLost);
theServer->fTotalRTPPacketsLost += periodicPacketsLost;

Sint64 curTime = OS::Milliseconds();
// 调用clock函数。clock函数返回进程第一次调用clock以来所用去的cpu时间。
                                                    // 第一次调用clock返回 0.
Float32 cpuTimeInSec = GetCPUTimeInSeconds();
// also update current bandwidth statistic
if(fLastBandwidthTime != 0)
{
       Uint32 delta = (Uint32)(curTime - fLastBandwidthTime);
       if(delta < 1000) {          // 何时会出现这种情况？？？
              (void)this->GetEvents();    // we must clear the event mask
              return theServer->GetPrefs()->GetTotalBytesUpdateTimeInSecs() * 1000;
       }
       设置theServer->fRTPPacketsPerSecond.(包流量)

       设置theServer->fCurrentRTPBandwidthInBits.（字节流量）

       let's do it for MP3 bytes now，设置theServer->fCurrentMP3BandwidthInBits.

       do the computation for cpu percent，设置theServer->fCPUPercent.
```

```
}
设置fLastTotalMP3Bytes、fLastBandwidthTime、theServer->fAvgMP3BandwidthInBits、
theServer->fCPUTimeUsedInSec。

// compute average bandwidth, a much more smooth value. This is done with the
// fLastBandwidthAvg, a timestamp of the last time we did an average, and fLastBytesSent,
// the number of bytes sent when we last did an average.
// GetAvgBandwidthUpdateTimeInSecs对应于配置文件中的average_bandwidth_update，缺省为 60s.
if((fLastBandwidthAvg != 0) && (curTime > (fLastBandwidthAvg +
            (theServer->GetPrefs()->GetAvgBandwidthUpdateTimeInSecs()*1000))))
{
        设置theServer->fAvgRTPBandwidthInBits。
        // if the bandwidth is above the bandwidth setting, disconnect 1 user by sending
        // them a BYE RTCP packet.
        // 如果平均带宽大于配置中的最大带宽，则做出处理
        //  GetMaxKBitsBandwidth对应于配置文件中的maximum_bandwidth，缺省为 102400 K/s.
        SInt32 maxKBits = theServer->GetPrefs()->GetMaxKBitsBandwidth();
        // 在streamingserver.xml里对safe_play_duration的解释：
        // If the server discovers it is serving more than its allowed maximum
        // bandwidth (using the average bandwidth computation) it will attempt to
        // disconnect the most recently connected clients until the average bandwidth
        // drops to acceptable levels. However, it will not disconnect clients if they've
        // been connected for longer than this time in seconds. If this value is set to 0,
        // it will never disconnect clients.
        if ((maxKBits>-1) && (theServer->fAvgRTPBandwidthInBits > ((UInt32)maxKBits*1024)))
        {
                OSMutexLocker locker(theServer->GetRTPSessionMap()->GetMutex());
                RTPSessionInterface* theSession=this->GetNewestSession(theServer->fRTPMap);
                if(theSession != NULL)
                        if ((curTime - theSession->GetSessionCreateTime()) <
                                theServer->GetPrefs()->GetSafePlayDurationInSecs()*1000)
                                theSession->Signal(Task::kKillEvent);
        }
} else if(fLastBandwidthAvg == 0) {
        fLastBandwidthAvg = curTime;        fLastBytesSent = theServer->fTotalRTPBytes;
}
(void)this->GetEvents();     // clear the event mask!
// 对应于配置文件中的total_bytes_update，缺省为 1s。
return theServer->GetPrefs()->GetTotalBytesUpdateTimeInSecs() * 1000;
```

- QTSSErrorLogModule

（1）、Register()

```
QTSS_AddRole(QTSS_ErrorLog_Role);
QTSS_AddRole(QTSS_Shutdown_Role);
QTSS_AddRole(QTSS_StateChange_Role);
```

QTSS_AddRole 这一类函数均在 QTSS_Private.cpp 里定义，即通过 sCallbacks 调用相应的 DSS 处理函数，而这里的 sCallbacks 即是在上面分析的 SetupModule 里赋值的 QTSServer 的成员 sCallbacks。而在 QTSServer::Initialize 函数里也已调用 InitCallbacks 函数对 sCallbacks 进行了初始化。这样各个模块就可以通过它来调用 DSS 的各种处理函数。

对于 QTSS_AddRole，处理函数是 QTSSCallbacks::QTSS_AddRole(见下面的分析)

QTSS_AddRole 通过 QTSSModule::AddRole 函数在 fRoleArray 数组里记录已支持的 Role 情况

(奇怪的是，这个数组是 QTSSModule 的私有成员，感觉应该是由 QTSServer 来维护这个记录)

(原来在 QTSServer::BuildModuleRoleArrays 函数里我们发现根据 QTSSModule 的 fRoleArray 数组的 Role 支持情况，在 QTSServerInterface::sNumModulesInRole 里做记录，并在

sModuleArray 数组里保存相关 Role 的所有 QTSSModule 指针）

QTSS_AddService(“RollErrorLog”, &RollErrorLog)

类似于 QTSS_AddRole，最终会调用 QTSSCallbacks::QTSS_AddService(见下面的分析)

CheckErrorLogState ()

```
sErrorLog = NEW QTSSErrorLog();     // 决定是否启动 ErrorLog task
sErrorLog->EnableLog();
// 如果 Error log task 已启动，将启动信息通过 sErrorLog 类写到 log 文件。
WriteStartupMessage();
sErrorLogCheckTask = NEW ErrorLogCheckTask();
```

（2）、 CheckErrorLogState()

```
// check error log
// 根据streamingserver.xml配置文件来决定是否启动ErrorLog task,
// thePrefs已经记录了各个属性设置值. 应该是根据属性名称来匹配每一个属性的.
if ((NULL == sErrorLog) && (thePrefs->IsErrorLogEnabled()))
{      sErrorLog = NEW QTSSErrorLog();     sErrorLog->EnableLog();      }
```

（3）、LogError()

```
... ...
if(sErrorLog != NULL)        sErrorLog->WriteToLog(...);
... ...
```

- ErrorLogCheckTask

Task类的继承类，主要的任务是每一个小时检查一次if the log needs to roll

（1）、构建函数

```
ErrorLogCheckTask() : Task()
{this->SetTaskName("ErrorLogCheckTask"); this->Signal(Task::kStartEvent); }
```

（2）、Run函数

```
// 如果创建了ErrorLog task，则调用sErrorLog->CheckRollLog();
// 和QTSSRollingLog每一分钟一次的任务相比，这里多了log文件大小的检查。
if(sErrorLog != NULL && sErrorLog->IsLogEnabled())       success = sErrorLog-
>CheckRollLog();
return (60*60*1000);        // 1 小时后执行；
```

- QTSSErrorLog、QTSSRollingLog

QTSSRollingLog的继承类, QTSSRollingLog是Task的继承类.

（1）、QTSSRollingLog::EnableLog()

```
this->Signal(TASK::kStartEvent);    // Start this object running!
```

打开log文件，并将log header写入该文件

（2）、QTSSRollingLog::WriteToLog()

确保log file已经打开

```
if(allowLogToRoll)          this->CheckRollLog();
```

调用qtss_fprintf将数据写入log文件。

对log file执行fflush、close工作。

（3）、QTSSRollingLog::CheckRollLog()

```
// first check to see if log rolling should happen because of a date interval. This takes
// precedence over size based log rolling. this is only if a client connects just between
// 00:00:00 and 00:01:00 since the task object runs every minute.
// when an entry is written to the log file, only the file size must be checked to see if
```

// it exceeded the limits.
// the roll interval should be monitored in a task object and rolled at midnight if the
// creation time has exceeded.
如果条件满足，则调用this->RollLog();

（4）、QTSSRollingLog::RollLog()
... ...this->RenameLogFile(fLogFullPath);　　... ...

（5）、QTSSRollingLog::RenameLogFile
this function takes care of renaming a log file from "myLogFile.log" to "myLogFile.981217000.log" or if that is already taken, myLogFile.981217001.log", etc

（6）、QTSSRollingLog::Run()
根据时间判断是否需要调用RollLog函数；
返回 60 * 1000;　　*// 该任务每 1 分钟会被调度一次。*

- OSQueue_Blocking

该类用作 TaskThread 的私有成员类，实际上是利用线程的条件变量实现了一个可等待唤醒的队列操作。下面是一些成员函数的分析：

（1）Enqueue
调用 OSQueue 类成员 fQueue.EnQueue 函数
调用 OSCond 类成员 fCond.Signal 函数即调用 pthread_cond_signal(&fCondition);

（2）DeQueueBlocking
如果 fQueue.GetLength() == 0，则调用 fCond.Wait 即调用 pthread_cond_timedwait 等待条件变量有效（从上面的分析可知：任务在调用 Signal 函数时，实际上就是调用对应线程的 fCond.Signal 函数）
fCond.wait 返回或者 fQueue.GetLength() != 0，调用 fQueue.DeQueue 返回队列里的任务对象。这里要注意一点，pthread_cond_timedwait 可能只是超时退出，所以 fQueue.DeQueue 可能只是返回空指针。

需要注意的是：在使用线程的条件变量时需要配合使用一个互斥锁，上述函数都是通过一个 OSMutexLocker 类的局部变量的构建/析构函数来完成这个锁的 Lock/Unlock 操作。

- OSMutexRW

这个类是通过互斥锁、线程条件变量、读写状态、等待状态来维持一个读和写互斥的操作。它有一个 OSMutex 类的成员 fInternalLock，来完成真正的互斥锁操作。在 OSMutex 类的构建函数里，会调用 pthread_mutex_init pthread_mutexattr_init 函数来初始化它的成员 fMutex（pthread_mutex_t 对象）。
下面是一些成员函数的分析：

（1）LockRead

// 通过互斥锁保护读写状态及计数变量。
// 在 OSMutexLocker 的构建函数里，会调用 OSMutex 类的 Lock 函数。OSMutexLocker 这个类的工
// 作就是纯粹完成 lock/unlock 操作，创建时 lock、销毁时 unlock(locker 只是一个局部变量，在
// 这个函数退出的时候被销毁)。
// 在 OSMutex 的 Lock 处理里，如果该线程已经获得该锁，则退出，否则调用
// pthread_mutex_lock，虽然这里有被阻塞的可能，但是因为 LockRead、LockWrite 很快退出，//
互斥锁也会被很快释放。
// 在 OSMutex 的 Unlock 处理里，如果线程不持有该锁，则退出，否则调用 pthread_mutex_unlock
OSMutexLocker locker(&fInternalLock);

// fReadWaiters += 1;
AddReadWaiter();

```
// 根据 fState、fWriteWaiters 来判断是否有写操作在进行或者有写等待
// 如果有，通过线程条件变量来等待，注意这里的 OSMutexRW::eMaxWait 为 0xFFFFFFFF，
// 不过 Wait 调用的 pthread_cond_wait 也会先 unlock 锁，从而避免死锁。
// fWriteWaiters: number of waiting writers
// fState:    -1: writer, 0: free, >0: readers
while (ActiveWriter() || WaitingWriters())
        fReadersCond.Wait(&fInternalLock, OSMutexRW::eMaxWait);

RemoveReadWaiter();
AddActiveReader();    // add 1 to active readers
fActiveReaders = fState;
```

（2）LockWrite

类似于 LockRead。

（3）Unlock

根据当前活跃的是读操作还是写操作，来唤醒写等待或者唤醒所有的读等待。

（4）TryLockWrite、TryLockRead

非阻塞的实现。

- ## IdleTaskThread、IdleTask

IdleTaskThread 基于 OSThread、IdleTask 基于 Task 类。

```
IdleTasks are identical to normal tasks (see task.h) with one exception:
    You can schedule them for timeouts. If you call SetIdleTimer on one, after the time
    has elapsed the task object will receive an OS_IDLE event.
```

这套机制实际上是利用一个线程专门来处理一个定时事件，平时睡眠，一旦发现有任务到期，就会通过 signal 通知另外的线程处理这个事件。而别的模块可以通过 IdleTaskThread::SetIdleTimer 函数

将自己的信息插入到 IdleTaskThread 的堆栈里。

TimoutTaskThread+TaskThread 其实也实现了类似的机制。和 IdleTask 机制相比，后者要更为准确：

1、TimeoutTaskThread 机制由 TaskThread 处理，而 TaskThread 除了处理这个，还有很多的任务处理。

2、TimeoutTaskThread::Run 函数里，在发现未到期的任务时，会莫名其妙的加个 1s（用意何在？？？）

另外 TimeoutTaskThread 机制实现的是周期性的事件处理，而 IdleTask 机制的是一次性定时事务处理。

感觉奇怪的是：既然有了这个纯粹处理定时事务的 IdleTask 机制，何必去使用纠缠到 TaskThread 上的 TimeoutTaskThread 机制呢？？？

（1）、IdleTask::Initialize 函数

该函数在 StartServer 里会被调用。

```
sIdleThread = NEW IdleTaskThread();        sIdleThread->Start();
```

（2）、IdleTaskThread::Entry 函数

空闲任务线程启动后，该函数运行。主要由一个大循环构成：

```
OSMutexLocker locker(&fHeapMutex);
while (true)
{
        // if there are no events to process, block.
        if (fIdleHeap.CurrentHeapSize() == 0)              fHeapCond.Wait(&fHeapMutex);

        // pop elements out of the heap as long as their timeout time has arrived.
        while ((fIdleHeap.CurrentHeapSize() > 0) && (fIdleHeap.PeekMin()->GetValue() <=
```

```
msec))
            {
                    IdleTask* elem = (IdleTask*)fIdleHeap.ExtractMin()->GetEnclosingObject();
                    elem->Signal(Task::kIdleEvent);
            }

            // we are done sending idle events. If there is a lowest tick count, then
            // we need to sleep until that time.
            if (fIdleHeap.CurrentHeapSize() > 0)
            {
                    SInt64 timeoutTime = fIdleHeap.PeekMin()->GetValue();
                    // because sleep takes a 32 bit number
                    timeoutTime -= msec;        UInt32 smallTime = (UInt32)timeoutTime;
                    fHeapCond.Wait(&fHeapMutex, smallTime);
            }
    }
```

- EventContext、EventThread

  An event context provides the intelligence to take an event generated from a UNIX file descriptor (usually EV_RE or EV_WR) and signal a Task.
  EventThread 基于 OSThread。
  EventThread 的机制是这样的：
  在 Entry 的循环调用里，select_waitevent 要么调用 select 监听包括管道文件在内的所有文件、要么在发现已就绪的描述字后，将每一个就绪描述字的 fd、事件类型、唯一的一个标识符返回给 Entry 函数。Entry 函数根据标识符在 fRefTable 里找到登记的 EventContext 的类对象，从而调用该对象的 processEvent 函数进行处理。
  周而复始。
  而那些基于 EventContext 类的模块在需要申请监听某个文件/socket 时，先调用 EventContext::RequestEvent 函数，在这个函数里，将要登记的 fd、事件类型添加到描述字集合里，并写一个字符到管道文件里，从而引起 EventThread 的 select 退出，重新 select 新的描述字集合。
  如果这个模块是第一次申请监听，还要分配一个唯一的描述符，并将自己的信息通过 fRef 登记到 fRefTable 里，以供 EventThread 使用。
  需要注意的是：模块在调用 processEvent 处理已发生的事件后，如果需要再次监听，则仍要调用 RequestEvent 函数。

  （1）、EventContext 构建函数

```
EventContext::EventContext(int inFileDesc, EventThread* inThread)
:  fFileDesc(inFileDesc), fEventThread(inThread), fWatchEventCalled(false),
   fUniqueIDStr((char*)&fUniqueID, sizeof(fUniqueID)), ... ...
{}
```

  （2）、EventContext::RequestEvent

```
// The first time this function gets called, we're supposed to call watchevent. Each
// subsequent time, call modwatch.
if (fWatchEventCalled)
{
        fEventReq.er_eventbits = theMask;
        // 更新描述字集合，并写管道文件，以通知 EventThread
        select_modwatch(&fEventReq, theMask)
} else {
        // 如果是第一次申请，需要先分配一个唯一的标识符
        // 并将 fRef 注册到 fRefTable，以供 EventThread 在后续处理中找到对应的对象
        if(!compare_and_store(10000000, 1, &sUniqueID))
                fUniqueID=(PointerSizedInt)atomic_add(&sUniqueID, 1);    //sUniqueID 初始化为 1
```

```
        else   fUniqueID = 1;
        fRef.Set(fUniqueIDStr, this);                          // fRef 是 OSRef 的类对
象。
        fEventThread->fRefTable.Register(&fRef);          // 参见下面 OSRef、OSRefTable

        // fill out the eventreq data structure
        ::memset(&fEventReq, '\0', sizeof(fEventReq));    fEventReq.er_type = EV_FD;
        fEventReq.er_handle = fFileDesc;          fEventReq.er_eventbits = theMask;
        fEventReq.er_data = (void*) fUniqueID;
        select_watchevent(&fEventReq, theMask)     // 实际上就是调用 select_modwatch
}
```

（3）、select_startevents

```
// initialize the select() implementation of the event queue

FD_ZERO(&sReadSet);          FD_ZERO(&sWriteSet);
FD_ZERO(&sReturnedReadSet); FD_ZERO(&sReturnedWriteSet);
创建 sCookieArray 并初始化
创建 sFDsToCloseArray 并初始化

// We need to wakeup select when the masks have changed. In order to do this,
// we create a pipe that gets written to from modwatch, and read when select returns.
::pipe((int*)&sPipes);

// Add the read end of the pipe to the read mask
FD_SET(sPipes[0], &sReadSet);              sMaxFDPos = sPipes[0];
```

（4）、select_modwatch

```
OSMutexLocker locker(&sMaxFDPosMutex);
// Add or remove this fd from the specified sets
if (which & EV_RE)          FD_SET(req->er_handle, &sReadSet);
else                        FD_CLR(req->er_handle, &sReadSet);
if (which & EV_WR)          FD_SET(req->er_handle, &sWriteSet);
else                        FD_CLR(req->er_handle, &sWriteSet);
if(req->er_handle > sMaxFDPos)             sMaxFDPos = req->er_handle;
sCookieArray[req->er_handle]= req->er_data;
// write to the pipe so that select wakes up and registers the new mask
// 这里写管道文件的作用见下面的分析
theErr = ::write(sPipes[1], “p”, 1);
```

（5）、select_waitevent

如果没有就绪的描述字，则调用 select 进行监听，否则通过 constructeventreq 将就绪描述字的信息传递给 EventThread::Entry 函数。

EventThread::Entry 函数根据 select_waitevent 返回值，决定是否再次调用 select_waitevent 还是调用就绪描述字对应的类的处理函数。

```
// check to see if we still have some select descriptors to process
// sNumFDsProcessed:    指的是已就绪待处理并已交由 EventThread::Entry 处理的描述字个数
// sNumFDsBackFromSelect:    指的是已就绪待处理的描述字总个数
int theFDsProcessed = (int)sNumFDsProcessed;      bool isSet = false;
if(theFDsProcessed < sNumFDsBackFromSelect)       // 还有已就绪的描述字待处理
{
       if(sInReadSet)
       {
              OSMutexLocker locker(&sMaxFDPosMutex);
              // 在 select_modwatch 函数中，sMaxFDPos 可能会被更新。
```

```
                // 也就是说 sMaxFDPos 反映现有被监听的最大文件描述符。
                // 找到 sReturnedReadSet 中第一个描述字，也就是第一个有效的描述字。

        while((!(isSet=FD_ISSET(sCurrentFDPos,&sReturnedReadSet)))&&(sCurrentFDPos<sMaxFDPos))
                        sCurrentFDPos++;
                if(isSet)
                {
                        FD_CLR(sCurrentFDPos, &sReturnedReadSet);
                        // 提出信息后交给 EventThread::Entry 处理
                        return constructeventreq(req, sCurrentFDPos, EV_RE);
                } else {
                        sInReadSet = false;          sCurrentFDPos = 0;
                }
        }
        if(!sInReadSet)
        {
                OSMutexLocker locker(&sMaxFDPosMutex);
                while((!(isSet=FD_ISSET(sCurrentFDPos,&sReturnedWriteSet)))&&(sCurrentFDPos<sMaxFDPos))
                        sCurrentFDPos++;
                if(isSet)
                {
                        FD_CLR(sCurrentFDPos, &sReturnedWriteSet);
                        return constructeventreq(req, sCurrentFDPos, EV_WR);
                } else {
                        // This can happen if another thread calls select_removeevent at
                        // just the right time, setting sMaxFDPos lower than it was when
                        // select() was last called. Becase sMaxFDPos is used as the place
to
                        // stop iterating over the read & write masks, setting it lower can
                        // cause file descriptors in the mask to get skipped. If they are
                        // skipped, that's ok, because those file descriptors were removed
by
                        // select_removeevent anyway. We need to make sure to finish
                        // iterating over the masks and call select again, which is why we
                        // set sNumFDsProcessed artificially here.
                        sNumFDsProcessed = sNumFDsBackFromSelect;
                }
        }
}
if(sNumFDsProcessed > 0)
{
        // 经过 select_waitevent 反复调用后，执行到这里说明所有已就绪的描述字都已处理。
        OSMutexLocker locker(&sMaxFDPosMutex);
        // We've just cycled through one select result. Re-init all the counting states
        sNumFDsProcessed=0; sNumFDsBackFromSelect=0; sCurrentFDPos = 0; sInReadSet = true;
}

// selecthasdata 不应该放在这里
while(!selecthasdata())
{
        {
                OSMutexLocker locker(&sMaxFDPosMutex);
                // Prepare to call select. Preserve the read and write sets by copying
                // their contents into the corresponding "returned" versions, and then pass
                // those into select
                // 更新描述字集合，从而让新申请的监听有效。
```

```
                // sReadSet、sWriteSet 会在 RequestEvent 函数里被更新。
                ::memcpy(&sReturnedReadSet, &sReadSet, sizeof(fd_set));
                ::memcpy(&sReturnedWriteSet, &sWriteSet, sizeof(fd_set));
        }

        设置 select 的 timeout 值，对于我们的系统，tv.tv_sec = 15。
        调用 OSThread::ThreadYield()        // 对于 linux 系统，这里实际上没有做什么工作。
        // select 的返回值有三种：
        // 返回-1，意味着出现错误，例如在任何描述字就绪之前就出现了信号。
        // 返回 0，意味着在任何描述字就绪之前，定时已到期。描述字集合被清 0
        // 返回正整数，它指出已就绪的描述字个数。描述字集合中只包括已就绪的描述字。
        sNumFDsBackFromSelect = ::select(sMaxFDPos+1, &sReturnedReadSet, &sReturnedWriteSet, NULL, &tv);
    }
    if(sNumFDsBackFromSelect >= 0)
        return EINTR; // either we've timed out or gotten some events. Either way, force
                      // caller to call waitevent again. select_waitevent 的调用者
                      //EventThread::Entry 在得到 EINTR 的返回值后，会再次调用 select_waitevent

    // 如果在第一次 select 时，出现错误（非 EINTR）。那么 EventThread 岂不是陷入这种情况的处理
    // 中，而不会做任何事情！！！这里应该将 sNumFDsBackFromSelect 清零。
    return sNumFDsBackFromSelect;
```

（6）、constructeventreq

```
        req->er_handle = fd;        req->er_eventbits = event;
        req->er_data = sCookieArray[fd];    sCurrentFDPos++;     sNumFDsProcessed++;
        // don't want events on this fd until modwatch is called.
        // 注意这里去掉了，如果还想继续监听的话，要重新 request。
        FD_CLR(fd, &sWriteSet);     FD_CLR(fd, &sReadSet);
```

（7）、selecthasdata

```
        // 返回 false, select_waitevent 会继续 select
        // 返回 true，select_waitevent 退出。EventThread::Entry 根据 select_waitevent 返回值
        // 决定是否处理事件还是继续调用 select_waitevent。
        if(sNumFDsBackFromSelect < 0)
        {
            int err = OSThread::GetErrno();
            if(err == EBADF || err ==EINTR)     return false;
            // if there is an error from select, we want to make sure and return
            // to the caller
            return true;
        }
        if(sNumFDsBackFromSelect == 0)
            return false; //if select returns 0, we've simply timed out, so recall
select

        // 为什么要用管道？表面上看好像就是为了通知关闭 sFDsToCloseArray 里的 fd。
        // 实际上别的模块在申请监听别的文件（或者说 socket）的时候，在添加到 sReadSet、
        // sWriteSet 后，会写管道文件（见 select_modwatch 函数）。这样 select_waitevent 中
        // 的 select 会提前返回。最终 selecthasdata 会再次返回 0，从而会将 sReadSet 拷贝到
        // sReturnedReadSet、sWriteSet 拷贝到 sReturnedWriteSet（注意 select 监听的是
        // sReturnedReadSet、sReturnedWriteSet），这样就达到了能很快监听新的文件的目的。
        // 不过这里要注意的是：select_waitevent 对于每一个有效的 fd 都会调用
        // constructeventreq 交给 EventThread::Entry 处理。而实际上管道的事件无需处理（也没
        // 有在 sCookieArray 里添加针对管道的项），所以这里会空走几个回合。不过模块写管道
        // 的次数毕竟不会很多，应该不会占用很多的 cpu。
```

```
        // 可以考虑将处理管道的这部分放在 select_waitevent 的最前面。
        if(FD_ISSET(sPipes[0], &sReturnedReadSet))
        {
                // we've gotten data on the pipe file descriptor. Clear the data.
                // increasing the select buffer fixes a hanging problem when the Darwin
                // server is under heavy load
                char theBuffer[4096];
                (void)::read(sPipes[0], &theBuffer[0], 4096);
                FD_CLR(sPipes[0], &sReturnedReadSet);           sNumFDsBackFromSelect--;
                {
                        // Check the fds to close array, and if there are any in it, close
                        // those descriptors
                        OSMutexLocker locker(&sMaxFDPosMutex);
                        for (Uint32 theIndex=0; ((sFDsToCloseArray[theIndex] != -1) &&
                                        (theIndex < sizeof(fd_set)*8)); theIndex++)
                        {
                                ::close(sFDsToCloseArray[theIndex]);
                                sFDsToCloseArray[theIndex] = -1;
                        }
                }
        }
        if(sNumFDsBackFromSelect == 0)
        return false;//if the pipe file descriptor is the ONLY data we've gotten, recall select
        else    return true;  //we've gotten a real event, return that to the caller
    }
```

（8）、EventThread::Entry

该函数是 event thread 的入口函数。类似于 Taskthread，在线程创建启动后，该函数会被调用。

主要由一个大循环构成。

```
    struct eventreq theCurrentEvent; ::memset(&theCurrentEvent, '\0',
sizeof(theCurrentEvent));
    while (true)
    {
        int theErrno = EINTR;
        while(theErrno == EINTR)
        {       int theReturnValue = select_waitevent(&theCurrentEvent, NULL);
                if(theReturnValue >= 0)      theErrno = theReturnValue;
                else                         theErrno = OSThread::GetErrno();
        }

        // there is data waiting on this socket. Send a wakeup.
        // 还有一种可能，就是前面出错（非 EINTR）。对于这种情况，其实可以根据
        // theCurrentEvent.er_data 是否为空来判断。但是在处理完 theCurrentEvent 后，却没有
        // memset 的动作！！！，这样会导致不存在的事件处理。
        if(theCurrentEvent.er_data != NULL)
        {// The cookie in this event is an objectID. Resolve that objectID into a pointer.

                StrPtrLen idStr((char*)&theCurrentEvent.er_data, sizeof(theCurrentEvent.er_data));
                OSRef* ref = fRefTable.Resolve(&idStr);
                if (ref != NULL)
                {       EventContext* theContext = (EventContext*)ref->GetObject();
                        // 在找到对应的类对象之后，调用该对象的处理函数。
                        theContext->ProcessEvent(theCurrentEvent.er_eventbits);
                        fRefTable.Release(ref);
                }
        }
```

```
        this->ThreadYield();        // 对于 linux 系统，ThreadYield 没有做任何事
    }
```

- OSRef、OSRefKey、OSRefTableUtils、OSRefTable、OSRefReleaser

  Class supports creating unique string IDs to object pointers. A grouping of an object and its string ID may be stored in an OSRefTable, and the associated object pointer can be looked up by string ID.

  Refs can only be removed from the table when no one is using the ref, therefore allowing clients to arbitrate access to objects in a preemptive, multithreaded environment.

  OSRef 对象引用表则是类似于 COM/DCOM 组件编程中 IUNKOWN 接口功能的数据结构，它首先为每个对象建立了一个字符串形式的 ID，以便于通过这个 ID 找到对象。

（三）、Attributes 相关

QTSS objects consist of attributes that are used to store data. Every attribute has a name, an attribute ID, a data type, and permissions for reading and writing the attribute's value. There are two attribute types:

- static attributes. Static attributes are present in all instances of an object type. A module can add static attributes to objects from its Register role only. All of the server's built-in attributes are static attributes.
- instance attributes. Instance attributes are added to a specific instance of any object type.A module can use any role to add an instance attribute to an object and can also remove instance attributes that it has added to an object.

相关的初始化在StartServer里开始，顺序为：
QTSSDictionaryMap::Initialize();
QTSServerInterface::Initialize();
sServer = NEW QTSServer();
而这个顺序也是不可更改的。

- QTSSDictionary类

  Dictionary is a data storage base class that implements key and value access to object data. This base class is inherited by all server objects defined by the API.
  它的继承类中的QTSSAttrInfoDict类就是被用来描述QTSS Object的具体一个属性。
  该类有指向 QTSSDictionaryMap 类对象的成员 fMap、fInstanceMap。

（1）、构建函数

```
QTSSDictionary::QTSSDictionary(QTSSDictionaryMap* inMap, OSMutex* inMutex)
: fAttributes(NULL), fInstanceAttr(NULL), fInstanceArraySize(0), fMap(inMap),
  fInstanceMap(NULL), fMutexP(inMutex), fMyMutex(false), fLocked*(false)
{
        // 根据dict map描述的属性个数，来初始化fAttributes。
        if(fMap != NULL)        fAttributes = NEW DictValueElement[inMap->GetNumAttrs()];
        if(fMutexP == NULL) {
                fMyMutex = true;        fMutexP = NEW OSMutex();
        }
}
```

（2）、QTSSDictionary::SetVal()

```
// 记录每个属性的信息、包括属性数据的来源和长度等。
fAttributes[inAttrID].fAttributeData.Ptr = (char*)inValueBuffer;
fAttributes[inAttrID].fAttributeData.Len = inBufferLen;
fAttributes[inAttrID].fAllocatedLen = inBufferLen;
fAttributes[inAttrID].fNumAttributes = 1;
```

（3）、QTSSDictionary::SetValue()

对于要设置的属性对应有多个值的情况，SetValue函数根据属性的ID、值的index找到正确的存储位置（这里可能会涉及到空间的重分配和老数据的转储），并利用inBuffer指针拷贝存储新的数据。
所以调用这个函数时传进来的inBuffer指针可以为局部变量。

```
// 根据传入的ID值，判断是否为Instance Attribute
QTSSDictionaryMap* theMap = fMap;       DictValueElement* theAttrs = fAttributes;
if(QTSSDictionaryMap::IsInstanceAttrID(inAttrID))
        { theMap = fInstanceMap;theAttrs = fInstanceAttrs; }

OSMutexLocker locker(fMutexP);
... ...                 // 一些异常情况的判断。
numValues = theAttrs[theMapIndex].fNumAttributes;
dataType = theMap->GetAttrType(the);      //fAttrArray[inIndex]->fAttrInfo.fAttrDataType
Uint32 attrLen = inLen;
if (dataType == qtssAttrDataTypeCharArray)
{
        if(inIndex > 0)    attrLen = sizeof(char *);   // value just contains a pointer
        if((numValues == 1) && (inIndex == 1))
        {
        // we're adding a second value, so we need to change the storage from directly
        // storing the string to an array of string pointers.
        // creating new memory here just to create a null terminated string instead of
        // directly using the old storage as the old storage didn't have its string null
        // terminated.
                UInt32 tempStringLen = theAttrs[theMapIndex].fAttributeData.Len;
                char* temp = NEW char[tempStringLen + 1];
                ::memcpy(temp, theAttrs[theMapIndex].fAttributeData.Ptr, tempStringLen);
                temp[tempStringLen] = '\0';
                delete [] theAttrs[theMapIndex].fAttributeData.Ptr;

                theAttrs[theMapIndex].fAllocatedLen = 16 * sizeof(char*);
                theAttrs[theMapIndex].fAttributeData.Ptr=NEW
_____________________________________char[theAttrs[theMapIndex].fAllocatedLen];
                theAttrs[theMapIndex].fAttributeData.Len = sizeof(char*);
                // store off original string as first value in array
                // 分配空间、将原先的字符串转存之后，把指针付给已重新分配的指针数组Ptr
                // 的第一个元素。
                *(char**)theAttrs[theMapIndex].fAttributeData.Ptr = temp;
        }
    } else {
        // If this attribute is iterated, this new value must be the same size as all the
        // others.
        if((inIndex > 0) || (numValues > 1)) &&
(theAttrs[theMapIndex].fAttributeData.Len != 0)
                        && (inLen != theAttrs[theMapIndex].fAttributeData.Len))
                return QTSS_BadArgument;
    }

    // Can't put empty space into the array of values
    if(inIndex > numValues)      return QTSS_BadIndex;
    if((attrLen *(inIndex + 1)) > theAttrs[theMapIndex].fAllocatedLen)
    {
        计算要重新分配的空间大小
        char * theNewBuffer = NEW char[theLen];
        if(inIndex > 0)       // 将老的数据拷贝过来
            ::memcpy(theNewBuffer, theAttrs[theMapIndex].fAttributeData.Ptr,
_____________________________theAttrs[theMapIndex].fAllocatedLen);
        if (theAttrs[theMapIndex].fAllocatedInternally)
            delete [] theAttrs[theMapIndex].fAttributeData.Ptr;
```

```
            // update this attribute structure with all the new values
            theAttrs[theMapIndex].fAttributeData.Ptr = theNewBuffer;
            theAttrs[theMapIndex].fAllocatedLen = theLen;
            theAttrs[theMapIndex].fAllocatedInternally = true;
        }

        // 先找到新数据要拷贝到的正确位置
        void *attributeBufferPtr;
        if ((dataType != qtssAttrDataTypeCharArray) || ((numValues < 2) && (inIndex == 0)))
        {
            attributeBufferPtr = theAttrs[theMapIndex].fAttributeData.Ptr + (inLen * inIndex);
            theAttrs[theMapIndex].fAttributeData.Len = inLen;
        } else {
            attributeBufferPtr = NEW char[inLen + 1]; // 为什么这里又要分配空间？？？
            char* tempBuffer = (char*)attributeBufferPtr;
            tempBuffer[inLen] = '\0';

            char** valuePtr=(char**)(theAttrs[theMapIndex].fAttributeData.Ptr+(attrLen *
inIndex));
            if(inIndex < numValues)             delete *valuePtr;
            *valuePtr = (char*)attributeBufferPtr;
        }

        // 拷贝新数据
        ::memcpy(attributeBufferPtr, inBuffer, inLen);

        if(inIndex>=theAttrs[theMapIndex].fNumAttributes)  theAttrs[theMapIndex].fNumAttributes++;

        if (((fMap == NULL) || fMap->CompleteFunctionsAllowed()) && !(inFlags &
                                    kDontCallCompletionRoutine))
            //SetValueComplete是虚函数，不同的继承类有不同的实现
            this->SetValueComplete(theMapIndex, theMap, inIndex, attributeBufferPtr, inLen);
```

- QTSSDictionaryMap类

这个类对象对应着一个具体的QTSS Object，用来描述这个Object的所有属性（或者说参数）。
它有一个QTSSAttrInfoDict**类型的成员fAttrArray，用来记录这个dict map所对应的Object的所有属性信息。
QTSSAttrInfoDict类（对一个属性的具体描述）的定义为：

```
class QTSSAttrInfoDict:     public QTSSDictionary
{
    public:
        struct AttrInfo {
            char    fAttrName[QTSS_MAX_ATTRIBUTE_NAME_SIZE + 1];
            QTSS_AttrFunctionPtr        fFuncPtr;
            QTSS_AttrDataType           fAttrDataType;
            QTSS_AttrPermission         fAttrPermission;
        };
        QTSSAttrInfoDict();
        virtual ~QTSSAttrInfoDict();
    private:
        AttrInfo            fAttrInfo;
        QTSS_AttributeID    fID;
        static AttrInfo     sAttributes[];
        friend class        QTSSDictionaryMap;
};
```

```
// 实际上kAttrInfoDictIndex所对应的dict map是专门用来描述每个属性的，也就是说这个dict
// map所对应的object是某一个属性，是属性的属性。它的属性个数是qtssAttrInfoNumParams(4):
// 属性名称、属性ID、属性数据类型、属性权限。
// 而QTSSAttrInfoDict被用来描述一个具体的属性，所以这里传入这个dict map来初始化基类
// QTSSDictionary。
QTSSAttrInfoDict::QTSSAttrInfoDict()                    // 构建函数
        :QTSSDictionary(QTSSDictionaryMap::GetMap(QTSSDictionaryMap::kAttrInfoDictIndex)),
        fID(qtssIllegalAttrID)

QTSSAttrInfoDict::AttrInfo  QTSSAttrInfoDict::sAttributes[] = {
{"qtssAttrName", NULL,qtssAttrDataTypeCharArray,qtssAttrModeRead |
qtssAttrModePreempSafe},
{"qtssAttrID", NULL, qtssAttrDataTypeUInt32, qtssAttrModeRead | qtssAttrModePreempSafe },
{"qtssAttrDataType",NULL,qtssAttrDataTypeUInt32,qtssAttrModeRead |
qtssAttrModePreempSafe},
{qtssAttrPermissions",NULL,qtssAttrDataTypeUInt32,qtssAttrModeRead|qtssAttrModePreempSafe}
}
```

（1）、构建函数

根据要描述的Object的属性个数来初始化fAttrArray。

```
QTSSDictionaryMap::QTSSDictionaryMap(UInt32 inNumReservedAttrs, UInt32 inFlags)
:   fNextAvailableID(inNumReservedAttrs), fNumValidAttrs(inNumReservedAttrs),
        fAttrArraySize(inNumReservedAttrs), fFlags(inFlags)
{
        ... ...
        fAttrArray = NEW QTSSAttrInfoDict*[fAttrArraySize];
        ::memset(fAttrArray, 0, sizeof(QTSSAttrInfoDict*) * fAttrArraySize);
}
```

（2）、QTSSDictionaryMap::Initialize()

首先创建并初始化kAttrInfoDictIndex所对应的dict map。

```
// Have to do this one first because this dict map is used by all the dict maps.
sDictionaryMaps[kAttrInfoDictIndex] = new QTSSDictionaryMap(qtssAttrInfoNumParams);

//setup the Attr Info attributes before constructing any other dictionaries.
for (Uint32 x = 0; x < qtssAttrInfoNumParams; x++)
        sDictionaryMaps[kAttrInfoDictIndex]->SetAttribute(x,
                QTSSAttrInfoDict::sAttributes[x].fAttrName,
                QTSSAttrInfoDict::sAttributes[x].fFuncPtr,
                QTSSAttrInfoDict::sAttributes[x].fAttrDataType,
                QTSSAttrInfoDict::sAttributes[x].fAttrPermission);
```

创建各种类型的dict map（比如kServerDictIndex、kPrefsDictIndex、kTextMessagesDictIndex、kServiceDictIndex、kRTPStreamDictIndex等），保存在sDictionaryMaps数组里。

（3）、QTSSDictionaryMap::SetAttribute

```
... ...
// 创建描述属性的QTSSAttrInfoDict对象，将属性包含的信息（ID、名称、函数指针、数据类型、
// 权限）保存下来。
fAttrArray[theIndex] = NEW QTSSAttrInfoDict;
fAttrArray[theIndex]->fID = inID;
::strcpy(&fAttrArray[theIndex]->fAttrInfo.fAttrName[0], inAttrName);
fAttrArray[theIndex]->fAttrInfo.fFuncPtr = inFuncPtr;
fAttrArray[theIndex]->fAttrInfo.fAttrDataType = inDataType;
fAttrArray[theIndex]->fAttrInfo.fAttrPermission = inPermission;
```

```
// 将属性四个成员(名称、ID、数据类型、权限)的相关信息通过基类QTSSDictionary保存起来。
fAttrArray[theIndex]->SetVal(qtssAttrName, &fAttrArray[theIndex]->fAttrInfo.fAttrName[0],
       theNameLen);
fAttrArray[theIndex]->SetVal(qtssAttrID, &fAttrArray[theIndex]->fID,
       sizeof(fAttrArray[theIndex]->fID));
fAttrArray[theIndex]->SetVal(qtssAttrDataType, &fAttrArray[theIndex]
       ->fAttrInfo.fAttrDataType, sizeof(fAttrArray[theIndex]->fAttrInfo.fAttrDataType));
fAttrArray[theIndex]->SetVal(qtssAttrPermissions, &fAttrArray[theIndex]
       ->fAttrInfo.fAttrPermission, sizeof(fAttrArray[theIndex]
              ->fAttrInfo.fAttrPermission));
```