RTSP服务运作

1. RTSP首先需建立TCP侦听socket。可见于此函数：

```
rtspServer = DynamicRTSPServer::createNew(*env, rtspServerPortNum, authDB);

DynamicRTSPServer*
DynamicRTSPServer::createNew(UsageEnvironment& env, Port ourPort,
            UserAuthenticationDatabase* authDatabase,
            unsigned reclamationTestSeconds) {
  int ourSocket = setUpOurSocket(env, ourPort);
  if (ourSocket == -1) return NULL;

  return new DynamicRTSPServer(env, ourSocket, ourPort, authDatabase, reclamationTestSec
}
```

- setUpOurSocket

```
 setUpOurSocket()
     ->setupStreamSocket() 建立套接字
     ->listen()
     ->getSourcePort() //bind()
```

2. 运行：rtspServer = DynamicRTSPServer::createNew(*env, rtspServerPortNum, authDB); 会运行DynamicRTSPServer的构造函数

```
 DynamicRTSPServer::DynamicRTSPServer(UsageEnvironment& env, int ourSocket,
             Port ourPort,
             UserAuthenticationDatabase* authDatabase, unsigned reclamationTestSeconds)
   : RTSPServer(env, ourSocket, ourPort, authDatabase, reclamationTestSeconds) {
}
```

- RTSPServer()：要帧听客户端的连接，就需要利用任务调度机制了，所以需添加一个socket handler。可见于此函数：

```
RTSPServer::RTSPServer(UsageEnvironment& env,
            int ourSocket, Port ourPort,
            UserAuthenticationDatabase* authDatabase,
            unsigned reclamationSeconds)
   : GenericMediaServer(env, ourSocket, ourPort, reclamationSeconds),
     fHTTPServerSocket(-1), fHTTPServerPort(0),
```

```
    fClientConnectionsForHTTPTunneling(NULL), // will get created if needed
    fTCPStreamingDatabase(HashTable::create(ONE_WORD_HASH_KEYS)),
    fPendingRegisterOrDeregisterRequests(HashTable::create(ONE_WORD_HASH_KEYS)),
    fRegisterOrDeregisterRequestCounter(0), fAuthDB(authDatabase), fAllowStreamingRTPOver
}
```

- GenericMediaServer:通过这个函数来添加socket handler

```
GenericMediaServer
::GenericMediaServer(UsageEnvironment& env, int ourSocket, Port ourPort,
         unsigned reclamationSeconds)
  : Medium(env),
    fServerSocket(ourSocket), fServerPort(ourPort), fReclamationSeconds(reclamationSecond
    fServerMediaSessions(HashTable::create(STRING_HASH_KEYS)),
    fClientConnections(HashTable::create(ONE_WORD_HASH_KEYS)),
    fClientSessions(HashTable::create(STRING_HASH_KEYS)),
    fPreviousClientSessionId(0)
{
  ignoreSigPipeOnSocket(fServerSocket); // so that clients on the same host that are kil

  // Arrange to handle connections from others:
  env.taskScheduler().turnOnBackgroundReadHandling(fServerSocket, incomingConnectionHandl
}
```

- **env.taskScheduler().turnOnBackgroundReadHandling(fServerSocket, incomingConnectionHandler, this);**

```
这个函数其实还是调用的
 void turnOnBackgroundReadHandling(int socketNum, BackgroundHandlerProc* handlerProc, vo
    setBackgroundHandling(socketNum, SOCKET_READABLE, handlerProc, clientData);
  }
  handlerProc ： 是select检测到函数动作之后的处理函数。
```

- **incomingConnectionHandler**

```

```

- setBackgroundHandling（）： 向任务调度对象添加socket Task任务

3. 当收到客户的连接时需保存下代表客户端的新socket，以后用这个socket与这个客户通讯。每个客户将来会对应一个rtp会话，而且各客户的RTSP请求只控制自己的rtp会话，那么最好建立一个会话

类，代表各客户的rtsp会话。于是类RTSPServer::RTSPClientSession产生，它保存的代表客户的socket。下为RTSPClientSession的创建过程

这个在新版本怎么调用的还没理清

```
1
```

4. Client会话类还没有理清楚

5. ServerMediaSession::createNew是一个简单工厂模式函数，在其中new了一个ServerMediaSession对象，看一下ServerMediaSession这个类的定义。

```
1
2  class ServerMediaSession: public Medium {
3  public:
4    static ServerMediaSession* createNew(UsageEnvironment& env,
5                 char const* streamName = NULL,
6                 char const* info = NULL,
7                 char const* description = NULL,
8                 Boolean isSSM = False,
9                 char const* miscSDPLines = NULL);
10
11   static Boolean lookupByName(UsageEnvironment& env,
12                               char const* mediumName,
13                               ServerMediaSession*& resultSession);
14
15   char* generateSDPDescription(int addressFamily); // based on the entire session /产生媒
16       // Note: The caller is responsible for freeing the returned string
17
18   char const* streamName() const { return fStreamName; }            //返回流的名称
19
20   Boolean addSubsession(ServerMediaSubsession* subsession);       //添加表示子会话的Server
21   unsigned numSubsessions() const { return fSubsessionCounter; }
22
23   void testScaleFactor(float& scale); // sets "scale" to the actual supported scale //返
24   float duration() const;
25     // a result == 0 means an unbounded session (the default)
26     // a result < 0 means: subsession durations differ; the result is -(the largest).
27     // a result > 0 means: this is the duration of a bounded session
28
29   virtual void noteLiveness();
30     // called whenever a client - accessing this media - notes liveness.
```

```
    // The default implementation does nothing, but subclasses can redefine this - e.g.,
    // want to remove long-unused "ServerMediaSession"s from the server.

  unsigned referenceCount() const { return fReferenceCount; }  //返回请求该流的RTSP客户端数
  void incrementReferenceCount() { ++fReferenceCount; }
  void decrementReferenceCount() { if (fReferenceCount > 0) --fReferenceCount; }
  Boolean& deleteWhenUnreferenced() { return fDeleteWhenUnreferenced; }//fDeleteWhenUnre

  void deleteAllSubsessions();
    // Removes and deletes all subsessions added by "addSubsession()", returning us to ar
    // Note: If you have already added this "ServerMediaSession" to a "RTSPServer" then,
    //   you must first close any client connections that use it,
    //   by calling "RTSPServer::closeAllClientSessionsForServerMediaSession()".

protected:
  ServerMediaSession(UsageEnvironment& env, char const* streamName,
         char const* info, char const* description,
         Boolean isSSM, char const* miscSDPLines);
  // called only by "createNew()"

  virtual ~ServerMediaSession();

private: // redefined virtual functions
  virtual Boolean isServerMediaSession() const;

private:
  Boolean fIsSSM;

  // Linkage fields:
  friend class ServerMediaSubsessionIterator; // ServerMediaSubsessionIterator是一个用于访
  ServerMediaSubsession* fSubsessionsHead;
  ServerMediaSubsession* fSubsessionsTail;
  unsigned fSubsessionCounter;

  char* fStreamName;
  char* fInfoSDPString;
  char* fDescriptionSDPString;
  char* fMiscSDPLines;

  struct timeval fCreationTime;
```

```
  unsigned fReferenceCount;
  Boolean fDeleteWhenUnreferenced;
};
```

6. 看一下ServerMediaSubsession这个类。ServerMediaSession原先说代表一个流，其实是不准确的。它代表的是server端的一个媒体的名字，而说ServerMediaSubsession代表一个Track是准确的。以后流指的是那些有数据流动的组合。

```

class ServerMediaSubsession: public Medium {
public:
  unsigned trackNumber() const { return fTrackNumber; }//每个ServerMediaSubsession又叫一个
  char const* trackId();                                ////  产生关于该视频流或者音频流的描述
  virtual char const* sdpLines() = 0;
  virtual void getStreamParameters(unsigned clientSessionId, // in
           struct sockaddr_storage const& clientAddress, // in
           Port const& clientRTPPort, // in
           Port const& clientRTCPPort, // in
           int tcpSocketNum, // in (-1 means use UDP, not TCP)
           unsigned char rtpChannelId, // in (used if TCP)
           unsigned char rtcpChannelId, // in (used if TCP)
           struct sockaddr_storage& destinationAddress, // in out
           u_int8_t& destinationTTL, // in out
           Boolean& isMulticast, // out
           Port& serverRTPPort, // out
           Port& serverRTCPPort, // out
           void*& streamToken // out
           ) = 0;
  virtual void startStream(unsigned clientSessionId, void* streamToken,// // 开始流化
         TaskFunc* rtcpRRHandler,
         void* rtcpRRHandlerClientData,
         unsigned short& rtpSeqNum,
         unsigned& rtpTimestamp,
         ServerRequestAlternativeByteHandler* serverRequestAlternativeByteHandler,
         void* serverRequestAlternativeByteHandlerClientData) = 0;
  virtual void pauseStream(unsigned clientSessionId, void* streamToken);    // 暂停流化
  virtual void seekStream(unsigned clientSessionId, void* streamToken, double& seekNPT,//
        double streamDuration, u_int64_t& numBytes);
     // This routine is used to seek by relative (i.e., NPT) time.
     // "streamDuration", if >0.0, specifies how much data to stream, past "seekNPT".  (I
```

```
    // "numBytes" returns the size (in bytes) of the data to be streamed, or 0 if unknow
  virtual void seekStream(unsigned clientSessionId, void* streamToken, char*& absStart,
    // This routine is used to seek by 'absolute' time.
    // "absStart" should be a string of the form "YYYYMMDDTHHMMSSZ" or "YYYYMMDDTHHMMSS
    // "absEnd" should be either NULL (for no end time), or a string of the same form as
    // These strings may be modified in-place, or can be reassigned to a newly-allocated
  virtual void nullSeekStream(unsigned clientSessionId, void* streamToken,     //// Frame
            double streamEndTime, u_int64_t& numBytes);
    // Called whenever we're handling a "PLAY" command without a specified start time.
  virtual void setStreamScale(unsigned clientSessionId, void* streamToken, float scale);
  virtual float getCurrentNPT(void* streamToken);
  virtual FramedSource* getStreamSource(void* streamToken);
  virtual void getRTPSinkandRTCP(void* streamToken,
        RTPSink const*& rtpSink, RTCPInstance const*& rtcp) = 0;
    // Returns pointers to the "RTPSink" and "RTCPInstance" objects for "streamToken".
    // (This can be useful if you want to get the associated 'Groupsock' objects, for ex
    // You must not delete these objects, or start/stop playing them; instead, that is d
    // using the "startStream()" and "deleteStream()" functions.
  virtual void deleteStream(unsigned clientSessionId, void*& streamToken);

  virtual void testScaleFactor(float& scale); // sets "scale" to the actual supported sca
  virtual float duration() const;
    // returns 0 for an unbounded session (the default)
    // returns > 0 for a bounded session
  virtual void getAbsoluteTimeRange(char*& absStartTime, char*& absEndTime) const; //// 
    // Subclasses can reimplement this iff they support seeking by 'absolute' time.

  // The following may be called by (e.g.) SIP servers, for which the
  // address and port number fields in SDP descriptions need to be non-zero:
  void setServerAddressAndPortForSDP(struct sockaddr_storage const& address,
             portNumBits portBits);

protected: // we're a virtual base class
  ServerMediaSubsession(UsageEnvironment& env);
  virtual ~ServerMediaSubsession();

  char const* rangeSDPLine() const;
      // returns a string to be delete[]d
```

```
  ServerMediaSession* fParentSession;
  struct sockaddr_storage fServerAddressForSDP;
  portNumBits fPortNumForSDP;

private:
  friend class ServerMediaSession;
  friend class ServerMediaSubsessionIterator;
  ServerMediaSubsession* fNext;

  unsigned fTrackNumber; // within an enclosing ServerMediaSession
  char const* fTrackId;
};
```

7.
8.
9.
10.
11.
12.
13.
14.
15.
16.
17.
18.
19.
20.
21.
22.