Live555 源码阅读

（版本："2011.12.23" 1324598400）

以下所说的类的定义，不一定是定义，可能是声明和部分定义。

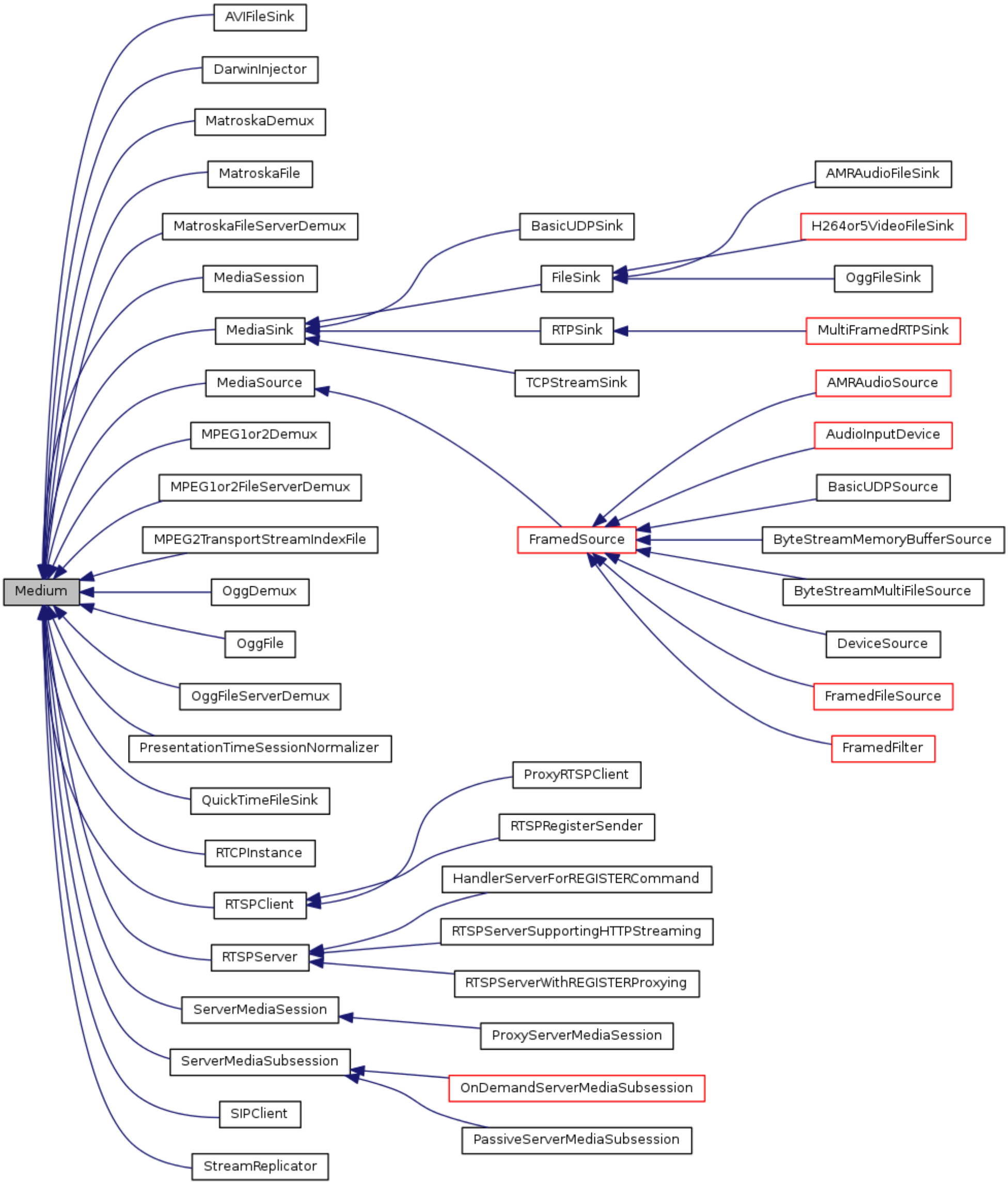
.hh 是 C++头文件，.cpp 是 C++源文件。*.h 是 C 头文件，*.c 是 C 源文件。一般 C 的头文件和源文件在一个目录中。

http://www.cnblogs.com/oloroso solym@sohu.com

一、 LiveMedia 相关类

这是 Live555 里面最最重要的部分了。

这里定义的类很多，其结构大致如下图所示。这张图还只是主要部分，还有很多是没有列出来的。图中红色的为抽象类。



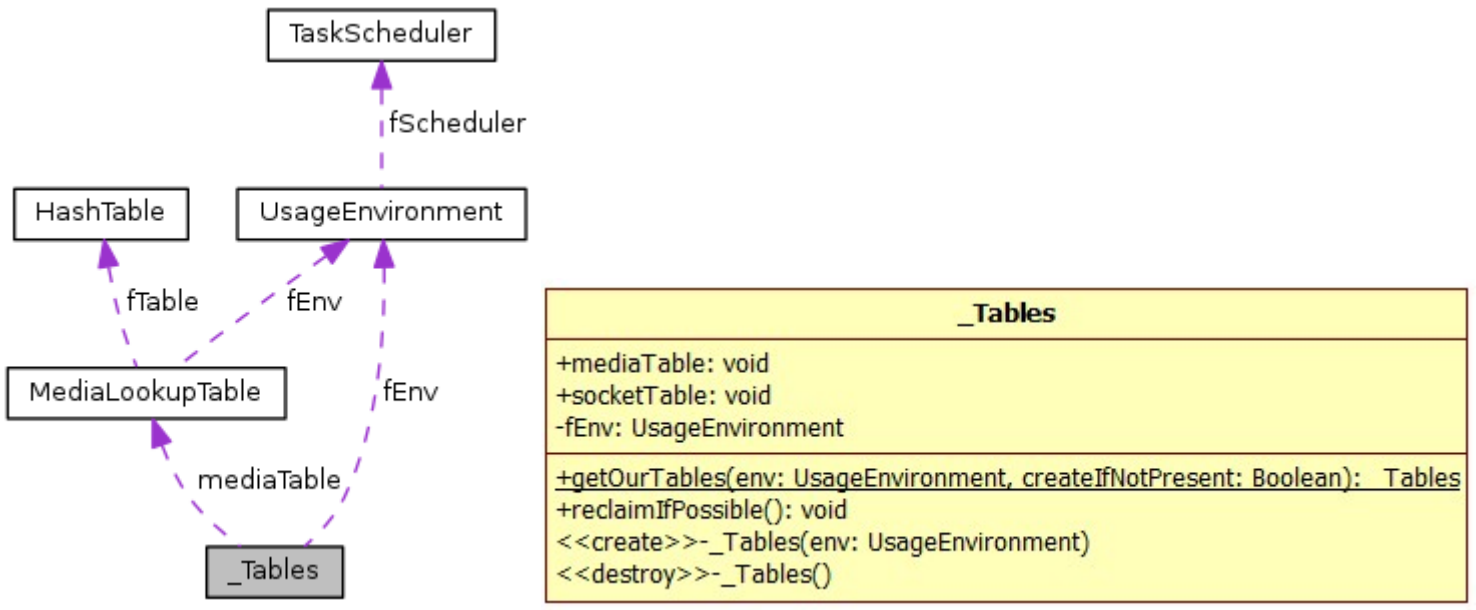
1.Medium 及相关类

这里要说的类就比较多了。

1) _Tables 类(env.liveMedia->Object)

还记得之前说的_groupsockPriv 结构体吗？这里与之类似。只是这里封装的更彻底，也略微复杂点。之前说 UsageEnvironment 的时候说过其有两个 void*的成员。groupsockPriv 已经说过了，这里要说的就是 liveMedia 了，这个成员在这里被用起来。其用于指向一个_Tables 对象，这个对象包含两个 void*指针，其在使用的时候会让其指向 BasicHashTable 对象。

_Tables 定义在 live555sourcecontrol\liveMedia\include\Media.hh 文件中。




```
// The structure pointed to by the "liveMediaPriv" UsageEnvironment field:
// UsageEnvironment 结构的 liveMediaPriv 字段指向
class _Tables {
public:
  // 返回 env.liveMediaPriv，如果其为 NULL，且 createIfNotPresent 为 true，则
  // return (_Tables*)(env.liveMediaPriv = new _Tables(env));
  static _Tables* getOurTables(UsageEnvironment& env, Boolean createIfNotPresent = True);
  // returns a pointer to an "ourTables" structure (creating it if necessary)
  // 返回一个指向“ourTables”结构（如果有必要创建它）

  // 自我销毁(在 mediaTable 和 socketTable 都为 NULL 时)
  void reclaimIfPossible();
  // used to delete ourselves when we're no longer used
  // 当我们不再使用时用于删除自己

  void* mediaTable;  //默认初始化为 NULL
  void* socketTable; //默认初始化为 NULL

protected:
  _Tables(UsageEnvironment& env);
  virtual ~_Tables();

private:
  UsageEnvironment& fEnv;
};
```

_Tables 构造与析构

这里没有说明好说的，只要指定这两个是 protected 权限即可，它们的调用在 getOurTables 和 reclasimIfPossible 中。

```
_Tables::_Tables(UsageEnvironment& env)
: mediaTable(NULL), socketTable(NULL), fEnv(env)
{
}
```


http://www.cnblogs.com/oloroso solym@sohu.com

```
_Tables::~_Tables()
{
}
```

getOurTables 方法(引用_Tables)

getOurTables 是一个 static 方法。返回 env.liveMediaPriv，如果其为 NULL，且参数 createIfNotPresent 为 true，则 return (_Tables*)(env.liveMediaPriv = new _Tables(env));

```
_Tables* _Tables::getOurTables(UsageEnvironment& env, Boolean createIfNotPresent)
{
  if (env.liveMediaPriv == NULL && createIfNotPresent) {
      env.liveMediaPriv = new _Tables(env);
  }
  return (_Tables*)(env.liveMediaPriv);
}
```

reclaimIfPossible 方法(自我回收)

reclaimIfPossible 方法用于自我回收。在成员 mediaTable 和 socketTable 都为 NULL 的时候才会去做。

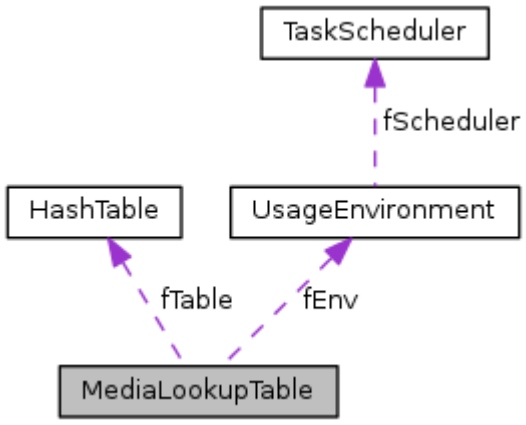
```
//自我销毁(在 mediaTable 和 socketTable 都为 NULL 时)
void _Tables::reclaimIfPossible()
{
  if (mediaTable == NULL && socketTable == NULL) {
      fEnv.liveMediaPriv = NULL;
      delete this;
  }
}
```

2) MediaLookupTable 媒体查找表类

MediaLookupTable 类与之前说过的 SocketLookupTable、GroupsockLookupTable 等类类似，都是内部有一个 HashTable，用于保存键值队来快速的访问查找。

我们这里先说这个类而非 Medium 类是因为这个类比较简单，且后面说 Medium 类的时候会涉及到这个类。

这个类声明和定义都在 live555sourcecontrol\liveMedia\Media.cpp 中。




```
// A data structure for looking up a Medium by its string name
// 一个数据结构，用于通过名称字符串查找 Medium
class MediaLookupTable {
public:
  // 获取 env.liveMedia->mediaTable，如果为 NULL，创建(new MediaLookupTable)后返回
  static MediaLookupTable* ourMedia(UsageEnvironment& env);

  // 通过 name 在哈希表中查找 Medium
  Medium* lookup(char const* name) const;
```


http://www.cnblogs.com/oloroso solym@sohu.com

```
  // Returns NULL if none already exists

  // 向哈希表中添加条目,mediumName 为 key,medium 为 value
  void addNew(Medium* medium, char* mediumName);

  // 从哈希表中移除 name 对应的条目，如果哈希表已经空了，释放它
  void remove(char const* name);

  // 产生一个"liveMedia%d"的字符串给 mediaName，%d 由 fNameGenerator 控制
  void generateNewName(char* mediumName, unsigned maxLen);

protected:
  // 绑定 env，初始化 fTable 和 fNameGenerator
  MediaLookupTable(UsageEnvironment& env);
  // 释放 fTable
  virtual ~MediaLookupTable();

private:
  UsageEnvironment& fEnv;      //使用环境
  HashTable* fTable;         //哈希表,key 类型为 char*字符串
  unsigned fNameGenerator;  //名字生成器
};
```

MediaLookupTable 构造与析构

构造的时候初始化成员，fEnv 和 fNameGenerator 就不说了。fTable 指向了一个新创建的 BasicHashTable 对象，key 类型是 char*字符串。

```
// 绑定 env，初始化 fTable 和 fNameGenerator
MediaLookupTable::MediaLookupTable(UsageEnvironment& env)
: fEnv(env), fTable(HashTable::create(STRING_HASH_KEYS)), fNameGenerator(0)
{
}
```

析构函数什么也没有做。这里要注意构造和析构的权限都是 protected 的。真正的释放自身是在 remove 中(fTable 指向哈希表中条目为空的时候)。

```
MediaLookupTable::~MediaLookupTable()
{
  delete fTable;
}
```

ourMedia 方法(引用 env.liveMedia->mediaTable)

ourMedia 是一个 static 方法。用于引用 env.liveMedia->mediaTable，如果为 NULL，创建(new MediaLookupTable)后返回。

```

MediaLookupTable* MediaLookupTable::ourMedia(UsageEnvironment& env)
{
  _Tables* ourTables = _Tables::getOurTables(env);
  if (ourTables->mediaTable == NULL) {
    // Create a new table to record the media that are to be created in
    // this environment:
    // 创建一个新的表来记录在这样的环境中所要创建的媒体
    ourTables->mediaTable = new MediaLookupTable(env);
  }
  return (MediaLookupTable*)(ourTables->mediaTable);
}
```

addNew 方法(添加条目)

http://www.cnblogs.com/oloroso solym@sohu.com

向哈希表中添加条目,mediumName 为 key,medium 为 value。

```
void MediaLookupTable::addNew(Medium* medium, char* mediumName)
{
  fTable->Add(mediumName, (void*)medium);
}
```

remove 方法(移除条目)

从哈希表中移除 name 对应的条目，如果哈希表已经空了，释放哈希表。这里释放是不会有问题的，再下一次调用 oueMedia 来引用这个 env.liveMedia 的时候，得之其为 NULL，会再为其创建。

```
void MediaLookupTable::remove(char const* name)
{
  Medium* medium = lookup(name);
  if (medium != NULL) {
     fTable->Remove(name);
     if (fTable->IsEmpty()) {
        // We can also delete ourselves (to reclaim space):
        _Tables* ourTables = _Tables::getOurTables(fEnv);
        delete this;
        ourTables->mediaTable = NULL;
        ourTables->reclaimIfPossible();
     }

     delete medium;
  }
}
```

lookup 方法(查找条目)

通过 name 在哈希表中查找 Medium。

```
Medium* MediaLookupTable::lookup(char const* name) const
{
  return (Medium*)(fTable->Lookup(name));
}
```

generateNewName 方法(为 medium 创建一个新名称)

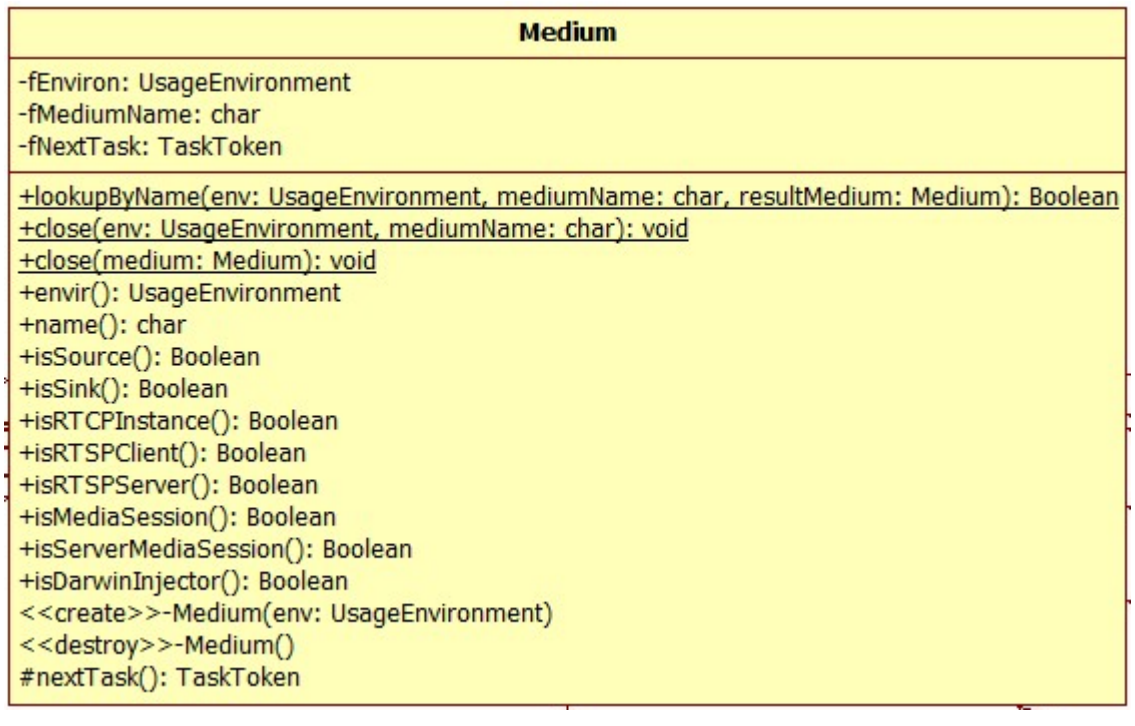
generateNewName 函数产生一个"liveMedia%d"的字符串给参数 mediaName，%d 由 fNameGenerator 控制。之前构造的时候 fNameGenerator 被初始化为了 0。这里再提一个有意思的，之前 remove 的时候，不去释放自身，那么这个值就会一直增长了，这会有问题吗？呵呵

```

void MediaLookupTable::generateNewName(char* mediumName,
  unsigned /*maxLen*/)
{
  // We should really use snprintf() here, but not all systems have it
  // 在这里我们应该真正使用 snprintf()，但不是所有的系统都有它
  // fNameGenerator 是 unsigned 类型，这里应该用%u 比较合适。%d 也是可以的
  sprintf(mediumName, "liveMedia%d", fNameGenerator++);
}
```

3) Medium 媒体基类

Medium 类是一个具有众多派生类的类，所有它定义了很多通用的方法、数据成员。

首先是绑定了一个 UsageEnvironment 对象，这个在很多类中都是有的。然后是一个媒体名称 fMediumName，这个很重要，在后面说它的派生类的时候会看到。




```cpp
#define mediumNameMaxLen 30

class Medium {
public:
  // 从哈希表 env.liveMedia->mediaTable 中查找 mediunName 对应的 Value，赋值给 resultMedium
  static Boolean lookupByName(UsageEnvironment& env,
     char const* mediumName,
     Medium*& resultMedium);

  // 从哈希表 env.liveMedia->mediaTable 中查找 mediumName 对应的条目移除
  static void close(UsageEnvironment& env, char const* mediumName);

  // // 从哈希表 medium->envir().liveMedia->mediaTable 中查找 medium->name()对应的条目移除
  static void close(Medium* medium); // alternative close() method using ptrs
  // (has no effect if medium == NULL)

  UsageEnvironment& envir() const { return fEnviron; }

  char const* name() const { return fMediumName; }

  // Test for specific types of media 对于媒体的特定类型的测试:
  // 默认都是返回 false，在各个派生类中对相关的进行重定义
  virtual Boolean isSource() const;
  virtual Boolean isSink() const;
  virtual Boolean isRTCPInstance() const;
  virtual Boolean isRTSPClient() const;
  virtual Boolean isRTSPServer() const;
  virtual Boolean isMediaSession() const;
  virtual Boolean isServerMediaSession() const;
  virtual Boolean isDarwinInjector() const;

protected:
  // 绑定 env，创建 mediunName，添加到 env.liveMedia->mediaTable
  Medium(UsageEnvironment& env); // abstract base class 抽象基类
  virtual ~Medium(); // instances are deleted using close() only 实例只能用 close()来释放

  TaskToken& nextTask()
  {
     return fNextTask;
  }
```


http://www.cnblogs.com/oloroso solym@sohu.com

```
private:
  friend class MediaLookupTable;
  UsageEnvironment& fEnviron;     //使用环境(绑定的)
  char fMediumName[mediumNameMaxLen];//名称
  TaskToken fNextTask;   //下一个任务
};
```

Medium 构造

构造函数很简单，显示生成了一个唯一的 name，然后将其添加到 env 的媒体查找表。这里说的唯一，是指在 env 的媒体表中不存在两个名字相同的媒体对象(Medium 派生类对象)。

```
Medium::Medium(UsageEnvironment& env)
: fEnviron(env), fNextTask(NULL)
{
  // First generate a name for the new medium:
  // 首先为 mediun 生成一个新 name:
  MediaLookupTable::ourMedia(env)->generateNewName(fMediumName, mediumNameMaxLen);
  env.setResultMsg(fMediumName);

  // Then add it to our table:
  MediaLookupTable::ourMedia(env)->addNew(this, fMediumName);
}
```

Medium 析构

析构的时候从任务调度器中移除所有可能被我们挂起的任务。

```
Medium::~Medium()
{
  // Remove any tasks that might be pending for us:
  // 移除任何可能被我们挂起的任务
  fEnviron.taskScheduler().unscheduleDelayedTask(fNextTask);
}
```

lookupByName 查找

在 env 的媒体表中通过媒体对象名称查找媒体对象。

```
Boolean Medium::lookupByName(UsageEnvironment& env, char const* mediumName,
  Medium*& resultMedium)
{
  resultMedium = MediaLookupTable::ourMedia(env)->lookup(mediumName);
  if (resultMedium == NULL) {
     env.setResultMsg("Medium ", mediumName, " does not exist");
     return False;
  }

  return True;
}
```

close 关闭媒体

只是简单的从 env 中把它参数代表的媒体对象移除。（ourMedia(env)->remove(name)操作会将 name 代表的对象 delete）

```
void Medium::close(UsageEnvironment& env, char const* name)
{
  MediaLookupTable::ourMedia(env)->remove(name);
}

void Medium::close(Medium* medium)
{
```


http://www.cnblogs.com/oloroso solym@sohu.com

```
  if (medium == NULL) return;

  close(medium->envir(), medium->name());
}
```

4) MediaSource 媒体源类

MediaSource 类继承自 Medium 类。没有再定义数据成员。

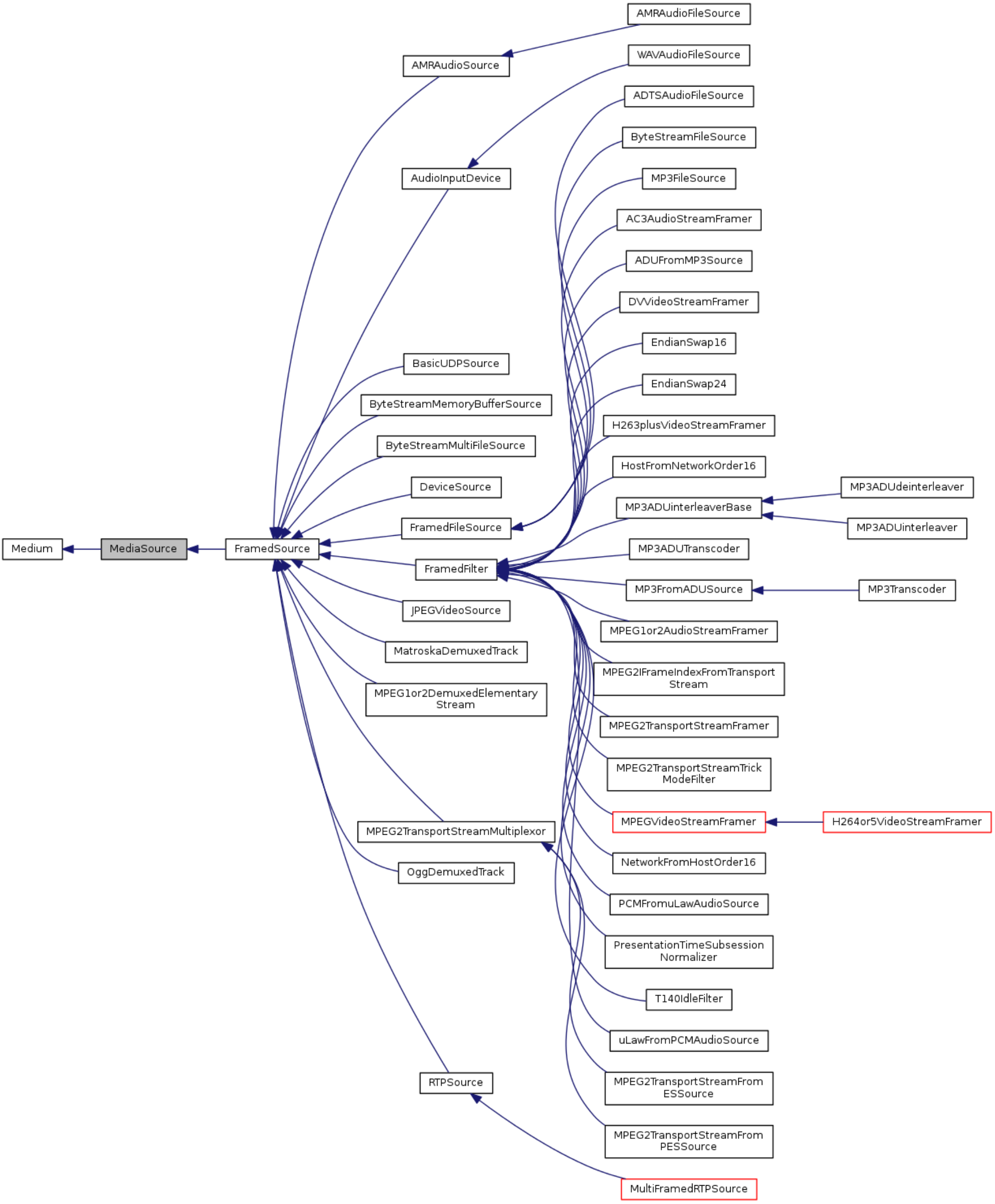
这是一个抽象基类，主要是为后面实现各种媒体源类做准备的。这里重定义了 lookupByName 方法，是因为在查找到媒体后还要判断其查找到的对象的是否是一个 MediaSource 对象。

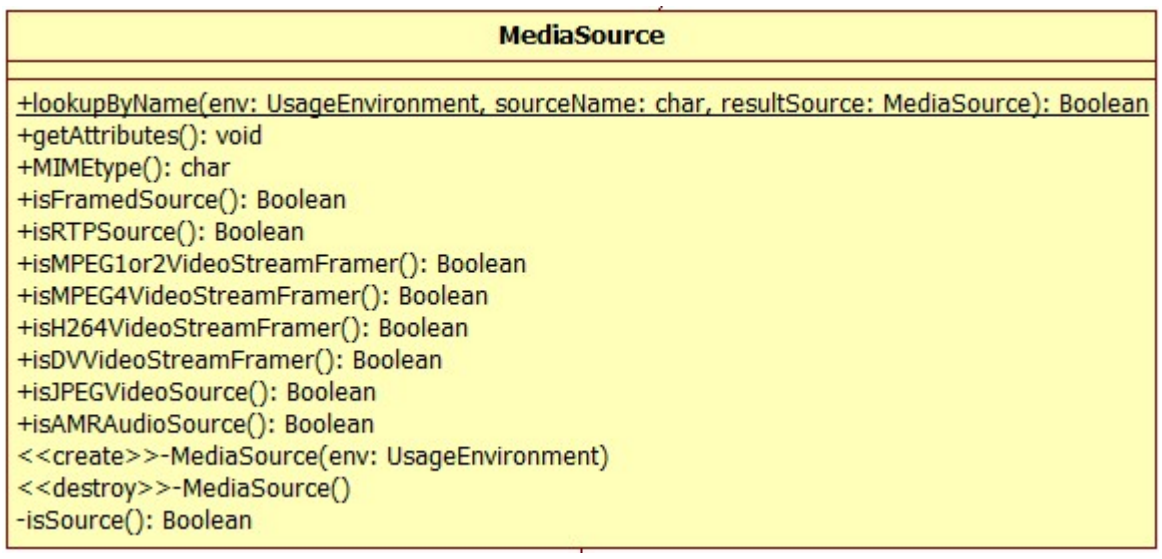
MediaSource 类的派生类都是对特定来源的媒体数据操作的封装，包括来自文件，UDP 数据报，内存缓冲区等。

这里还引入了一个知识点，MIME。这个东西在页面 http://www.w3school.com.cn/media/media_mimeref.asp 有介绍，这里默认其为字节流类型。

类中定义了一些 isXXX 的虚方法，这是为派生类准备的，用于对象类型判断。

http://www.cnblogs.com/oloroso solym@sohu.com


AMRAudioFileSource
AMRAudioSource
WAVAudioFileSource
ADTSAudioFileSource
ByteStreamFileSource
AudioInputDevice
MP3FileSource
AC3AudioStreamFramer
ADUFromMP3Source
DVVideoStreamFramer
EndianSwap16
BasicUDPSource
EndianSwap24
ByteStreamMemoryBufferSource
H263plusVideoStreamFramer
ByteStreamMultiFileSource
HostFromNetworkOrder16
DeviceSource
MP3ADUdeinterleaver
MP3ADUinterleaverBase
FramedFileSource
MP3ADUinterleaver
Medium
MediaSource
FramedSource
MP3ADUTranscoder
FramedFilter
MP3FromADUSource
MP3Transcoder
JPEGVideoSource
MPEG1or2AudioStreamFramer
MatroskaDemuxedTrack
MPEG2IFrameIndexFromTransport Stream
MPEG1or2DemuxedElementary Stream
MPEG2TransportStreamFramer
MPEG2TransportStreamTrick ModeFilter
MPEGVideoStreamFramer
H264or5VideoStreamFramer
MPEG2TransportStreamMultiplexor
NetworkFromHostOrder16
OggDemuxedTrack
PCMFromuLawAudioSource
PresentationTimeSubsession Normalizer
T140IdleFilter
uLawFromPCMAudioSource
RTPSource
MPEG2TransportStreamFrom ESSource
MPEG2TransportStreamFrom PESSource
MultiFramedRTPSource




```
class MediaSource : public Medium {
public:
  static Boolean lookupByName(UsageEnvironment& env, char const* sourceName,
     MediaSource*& resultSource);

  virtual void getAttributes() const;
  // attributes are returned in "env's" 'result message'
  // 返回的属性在 env 的 result message 中

  // The MIME type of this source:
  // 本 source 的 MIME 类型：
  virtual char const* MIMEtype() const;

  // Test for specific types of source:
  // 注意，这些是新定义的，不是从基类继承来的
  virtual Boolean isFramedSource() const;
  virtual Boolean isRTPSource() const;
  virtual Boolean isMPEG1or2VideoStreamFramer() const;
  virtual Boolean isMPEG4VideoStreamFramer() const;
  virtual Boolean isH264VideoStreamFramer() const;
  virtual Boolean isDVVideoStreamFramer() const;
  virtual Boolean isJPEGVideoSource() const;
  virtual Boolean isAMRAudioSource() const;

protected:
  MediaSource(UsageEnvironment& env); // abstract base class 抽象基类
  virtual ~MediaSource();

private:
  // redefined virtual functions:重定义虚函数
  virtual Boolean isSource() const;  //返回 true
};
```

MediaSource 的一些方法实现

这里能将的不多，直接贴代码。

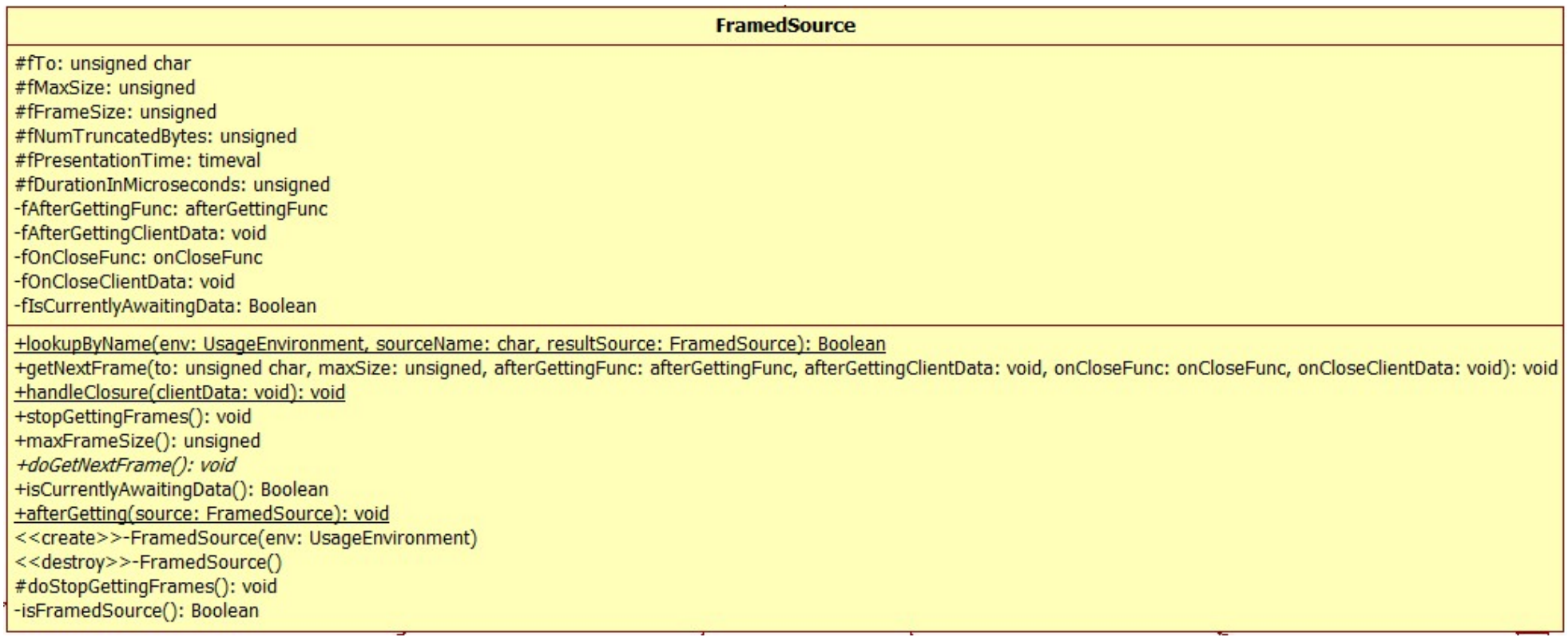
```
MediaSource::MediaSource(UsageEnvironment& env)
: Medium(env)
{}
MediaSource::~MediaSource()
{}
Boolean MediaSource::isSource() const
{ return True;}
char const* MediaSource::MIMEtype() const
{
  return "application/OCTET-STREAM"; // default type
```

```
  //OCTET 八位位组,八位字节 // 应用程序 字节    流
  // http://www.w3school.com.cn/media/media_mimeref.asp
}

Boolean MediaSource::isFramedSource() const
{ return False; // default implementation}
Boolean MediaSource::isRTPSource() const
{ return False; // default implementation}
Boolean MediaSource::isMPEG1or2VideoStreamFramer() const
{ return False; // default implementation}
Boolean MediaSource::isMPEG4VideoStreamFramer() const
{ return False; // default implementation}
Boolean MediaSource::isH264VideoStreamFramer() const
{ return False; // default implementation}
Boolean MediaSource::isDVVideoStreamFramer() const
{ return False; // default implementation}
Boolean MediaSource::isJPEGVideoSource() const
{ return False; // default implementation}
Boolean MediaSource::isAMRAudioSource() const
{ return False; // default implementation}

Boolean MediaSource::lookupByName(UsageEnvironment& env,
  char const* sourceName,
  MediaSource*& resultSource)
{
  resultSource = NULL; // unless we succeed 除非我们成功
}
  Medium* medium;
  if (!Medium::lookupByName(env, sourceName, medium)) return False;

  if (!medium->isSource()) {
     env.setResultMsg(sourceName, " is not a media source");
     return False;
  }

  resultSource = (MediaSource*)medium;
  return True;
}

void MediaSource::getAttributes() const
{
  // Default implementation
  envir().setResultMsg("");
}
```

5) FramedSource 帧源类

帧源类继承自媒体源类。这也是个抽象类，起到承上启下的作用(继承自 MediaSource，派生出一系列的媒体帧源相关类)。

http://www.cnblogs.com/oloroso solym@sohu.com



类中定义了大量的成员变量，但不是在这里使用的，是为派生类准备的。

```
// 帧源
class FramedSource : public MediaSource {
public:
  // 从 env.liveMedia->mediaTable 中查找 sourceName 代表的 FramedSource
  static Boolean lookupByName(UsageEnvironment& env, char const* sourceName,
      FramedSource*& resultSource);

  // 类型定义  (获取后...)
  typedef void (afterGettingFunc)(void* clientData, unsigned frameSize,
      unsigned numTruncatedBytes,  /*截断的字节数*/
  struct timeval presentationTime,/*显示时间*/
      unsigned durationInMicroseconds);  /*持续时长*/
  // 类型定义   (关闭时...)
  typedef void (onCloseFunc)(void* clientData);

  // 将参数赋值给对应的成员变量(目前还在等待读取时).然后调用 doGetNextFrame(纯虚函数，派生类中实现)
  void getNextFrame(unsigned char* to, unsigned maxSize,
      afterGettingFunc* afterGettingFunc,
      void* afterGettingClientData,
      onCloseFunc* onCloseFunc,
      void* onCloseClientData);

  // 关闭时处理
  static void handleClosure(void* clientData);
  // This should be called (on ourself) if the source is discovered
  // to be closed (i.e., no longer readable)
  // 这将在（在我们自己）发现源(source)被关闭时调用（即，不再可读）

  // 停止获取帧
  void stopGettingFrames();

  virtual unsigned maxFrameSize() const;
  // size of the largest possible frame that we may serve, or 0
  // if no such maximum is known (default)
  // 最大帧 size，我们可以 serve，或者 0，如果没有已知的最大值(默认)

  virtual void doGetNextFrame() = 0;
  // called by getNextFrame() 由 getNextFrame 调用

  Boolean isCurrentlyAwaitingData() const { return fIsCurrentlyAwaitingData; }
```


http://www.cnblogs.com/oloroso solym@sohu.com

```
  static void afterGetting(FramedSource* source);
  // doGetNextFrame() should arrange for this to be called after the
  // frame has been read (*iff* it is read successfully)
  // doGetNextFrame()应安排在此被调用，帧已被读取后（*当且仅当*这是成功读取）

protected:
  FramedSource(UsageEnvironment& env); // abstract base class
  virtual ~FramedSource();

  virtual void doStopGettingFrames();

protected:
  // The following variables are typically accessed/set by doGetNextFrame()
  // 通过 doGetNextFrame()去访问或设置下面的变量
  unsigned char* fTo; // in
  unsigned fMaxSize; // in  最大 size
  unsigned fFrameSize; // out  帧 size
  unsigned fNumTruncatedBytes; // out 截断的字节数
  struct timeval fPresentationTime; // out  显示时间
  unsigned fDurationInMicroseconds; // out  持续时间数(微秒)

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

private:
  afterGettingFunc* fAfterGettingFunc;  //获取之后调用
  void* fAfterGettingClientData;  //获取数据后回调对象
  onCloseFunc* fOnCloseFunc;  //关闭时调用
  void* fOnCloseClientData; //关闭时回调对象

  Boolean fIsCurrentlyAwaitingData;  //目前正在等待数据？
};
```

FramedSource 构造与析构

这个真没什么好说的，前面说了这个类中定义的很多数据成员都是为派生类准备的，这里构造的时候初始化了一下。

```
FramedSource::FramedSource(UsageEnvironment& env)
: MediaSource(env),
fAfterGettingFunc(NULL), fAfterGettingClientData(NULL),
fOnCloseFunc(NULL), fOnCloseClientData(NULL),
fIsCurrentlyAwaitingData(False)
{
  fPresentationTime.tv_sec = fPresentationTime.tv_usec = 0; // initially
}
```

```
FramedSource::~FramedSource()
{
}
```

lookupByName 查找

和它的基类很类似，不再解释。

```
Boolean FramedSource::lookupByName(UsageEnvironment& env, char const* sourceName,
  FramedSource*& resultSource)
{
  resultSource = NULL; // unless we succeed 除非我们成功
```


http://www.cnblogs.com/oloroso solym@sohu.com

```
  MediaSource* source;
  if (!MediaSource::lookupByName(env, sourceName, source)) return False;

  if (!source->isFramedSource()) {
      env.setResultMsg(sourceName, " is not a framed source");
      return False;
  }

  resultSource = (FramedSource*)source;
  return True;
}
```

getNextFrame 获取下一帧数据

这个方法并没有真正的实现获取下一帧数据的功能。这里仅仅是判断了一下是否正在获取数据，如果是，那就不能去获取了。如果不是，那么就简单的将参数赋值给对应的成员咯。最后调用 doGetNextFrame 方法，而这个方法是一个纯虚方法，在其派生类中实现。

```
void FramedSource::getNextFrame unsigned char* to/*目标位置*/, unsigned maxSize,
      afterGettingFunc* afterGettingFunc /*获取后回调*/,
      void* afterGettingClientData /*获取后回调函数的参数*/,
      onCloseFunc* onCloseFunc /* 在关闭时回调 */,
      void* onCloseClientData)
 {
  // Make sure we're not already being read:
  // 确保我们不是正在读取
  if (fIsCurrentlyAwaitingData) {
      envir() << "FramedSource[" << this << "]::getNextFrame(): attempting to read more than once at the
 same time!\n";
      envir().internalError();
  }

  fTo = to;
  fMaxSize = maxSize;
  fNumTruncatedBytes = 0; // by default; could be changed by doGetNextFrame() 默认，可以调用 doGetNextFra
med 去改变
  fDurationInMicroseconds = 0; // by default; could be changed by doGetNextFrame()
  fAfterGettingFunc = afterGettingFunc;
  fAfterGettingClientData = afterGettingClientData;
  fOnCloseFunc = onCloseFunc;
  fOnCloseClientData = onCloseClientData;
  fIsCurrentlyAwaitingData = True;    //表示当前正在读取数据

  doGetNextFrame();
}
```

handleClosure 关闭时调用处理

注意这个函数是 static 的。这里其实还是调用的成员 fOnCloseFunc。

成员 fOnCloseFunc 的类型见 typedef void (onCloseFunc)(void* clientData)

注意到这里参数 clientData 其实不能直接被 fOnColseFunc 使用，而要经过转换为 FramedSource 后再获取它的 fOnCloseClient Data 成员来使用。

```
void FramedSource::handleClosure(void* clientData)
{
  FramedSource* source = (FramedSource*)clientData;
  source->fIsCurrentlyAwaitingData = False; // because we got a close instead  因为我们有一个关闭替代品
  if (source->fOnCloseFunc != NULL) {
      // 函数指针调用函数
      (*(source->fOnCloseFunc))(source->fOnCloseClientData);
```


http://www.cnblogs.com/oloroso solym@sohu.com

```
  }
}
```

afterGetting 获取后调用

这个方法和 handleClosure 是类似的，都是 static 方法。然后这里先将 fIsCurrentlyAwaitingData(当前正在等待数据)设置为了 false，表明当前没有在读取数据，或者说当前可以调用 getNextFrame 方法咯。这里的注释也说明了这样做的原因，就是为了防止后面调用 fAfterGettingFunc 的时候，fAfterGettingFunc 中可能会有读取下一帧的操作。

```
void FramedSource::afterGetting(FramedSource* source)
{
  source->fIsCurrentlyAwaitingData = False;
  // indicates that we can be read again
  //表明我们可以再次读取
  // Note that this needs to be done here, in case the "fAfterFunc"
  // called below tries to read another frame (which it usually will)
  //注意，这需要在这里完成，以防“fAfterFunc”的调用试图读取另一帧（它通常会）

  if (source->fAfterGettingFunc != NULL) {
     (*(source->fAfterGettingFunc))(source->fAfterGettingClientData,
        source->fFrameSize, source->fNumTruncatedBytes,
        source->fPresentationTime,
        source->fDurationInMicroseconds);
  }
}
```

stopGettingFrames 停止获取帧

不说了，看代码吧。

```
void FramedSource::stopGettingFrames()
{
  fIsCurrentlyAwaitingData = False; // indicates that we can be read again 表明我们可以再次读取

  // Perform any specialized action now:现在请执行专用操作：
  // 这是个虚函数，本类是抽象类。应该调用派生类的实现
  doStopGettingFrames();
}
```

doStopGettingFrames 执行停止获取帧

这样不用说了吧，如果要说，那就是这个函数是虚函数，不是纯虚函数。

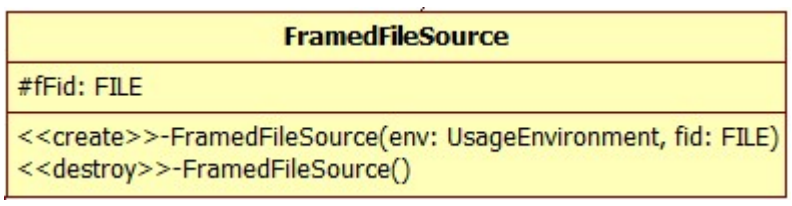
```
void FramedSource::doStopGettingFrames()
{
  // Default implementation: Do nothing  默认什么也不做
  // Subclasses may wish to specialize this so as to ensure that a
  // subsequent reader can pick up where this one left off.
  // 子类可能希望实现这个专门的操作来后续的读取者能够正确的使用
}
```

6) FramedFileSource 文件帧源

文件帧源类是帧源类的派生，这还是一个承上启下的抽象类。

其内部新定义了一个文件指针 fFid，而构造函数的作用就是为其赋值。

http://www.cnblogs.com/oloroso solym@sohu.com




```
// 文件帧源
class FramedFileSource: public FramedSource {
protected:
  FramedFileSource(UsageEnvironment& env, FILE* fid); // abstract base class
  virtual ~FramedFileSource();

protected:
  FILE* fFid;  //文件指针
};
```

构造与析构

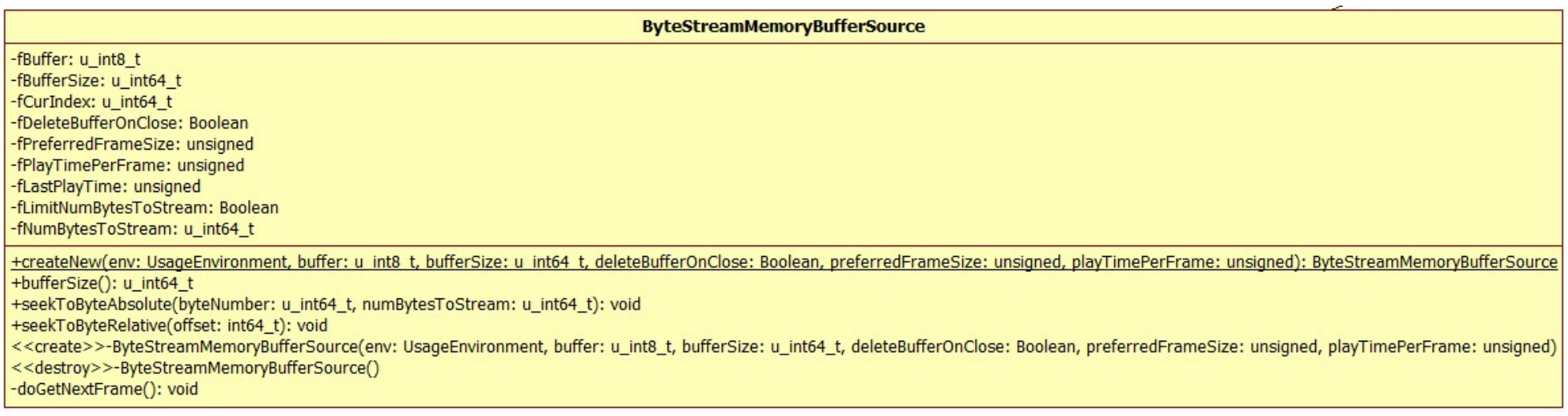
```
FramedFileSource::FramedFileSource(UsageEnvironment& env, FILE* fid)
  : FramedSource(env), fFid(fid) {
}

FramedFileSource::~FramedFileSource() {
}
```

7) ByteStreamMemoryBufferSource 内存缓冲字节流源

前面说过 FrameSource 类有多个派生类，每一个派生类都对应不同的数据来源。这一个比较简单，所以先说这一个。对于这个名称的翻译，是"内存缓冲字节流源"还是"字节流内存缓冲源"这都不是很合适。这里选择第一个，因为我觉得其首先是一个字节流源，内存缓冲用于修饰它。

这个类中定义了一些数据成员，在下面给出代码中注释了。这里还要提一下的是 fLimitNumBytesToStream 和 fNumBytesToStream 成员，因为这两个从字面意思上不好理解。

这里先要说清楚一下 ByteStreamMemoryBufferSource 这个类的作用。这个类在构造的时候需要给其指定一个字节数组 buffer，当然也要指定其大小。在 doGetNextFrame 方法获取一个帧数据的时候，就是从这个字节数组里面获取。fLimitNumBytesToStream 成员表示对输出到流的字节数是否受限制，而 fNumBytesToStream 表示能够输出到流的字节数。这里我们说的流的意思是向外部输出的数据目标。即在 FrameSource 中定义的 getNextFrame 方法的 to 参数(对应 fTO 成员)。




```
class ByteStreamMemoryBufferSource : public FramedSource {
public:
  // 在参数 buffer!=NULL 的时候，创建内存缓存字节流源对象
  static ByteStreamMemoryBufferSource* createNew(UsageEnvironment& env,
     u_int8_t* buffer, u_int64_t bufferSize,
     Boolean deleteBufferOnClose = True,
     unsigned preferredFrameSize = 0,
     unsigned playTimePerFrame = 0);
  // "preferredFrameSize" == 0 means 'no preference'(没有要求)
```


http://www.cnblogs.com/oloroso solym@sohu.com

```
  // "playTimePerFrame" is in microseconds(微秒单位)

  u_int64_t bufferSize() const { return fBufferSize; }

  // 跳寻到 byteNumber 位置(绝对位置),更新 numBytesToStream
  void seekToByteAbsolute(u_int64_t byteNumber, u_int64_t numBytesToStream = 0);
  // if "numBytesToStream" is >0, then we limit the stream to that number of bytes, before treating it a
s EOF
  // 如果“numBytesToStream”> 0，那么我们限制流的字节数，将其视为 EOF 前

  // 跳寻到字节流相对位置
  void seekToByteRelative(int64_t offset);

protected:
  // 用参数初始化成员
  ByteStreamMemoryBufferSource(UsageEnvironment& env,
     u_int8_t* buffer, u_int64_t bufferSize,
     Boolean deleteBufferOnClose,
     unsigned preferredFrameSize,
     unsigned playTimePerFrame);
  // called only by createNew()

  virtual ~ByteStreamMemoryBufferSource();

private:
  // redefined virtual functions:
  virtual void doGetNextFrame();

private:
  u_int8_t* fBuffer;     // 缓冲区
  u_int64_t fBufferSize;// 缓冲区大小
  u_int64_t fCurIndex;  // 当前索引位置
  Boolean fDeleteBufferOnClose;   //关闭时释放缓冲区
  unsigned fPreferredFrameSize;   //最佳帧大小
  unsigned fPlayTimePerFrame;     //每帧播放时间
  unsigned fLastPlayTime;      // 最后播放时间
  Boolean fLimitNumBytesToStream; //到流字节数受到限制?(表示获取帧的时候要检查)
  u_int64_t fNumBytesToStream; //可到流字节数 used iff "fLimitNumBytesToStream" is True 当且仅当用于“fLim
itNumBytesToStream”是真
};
```

构造与析构

构造函数只是简单用参数初始化了数据成员。注意，构造函数是 protected 权限的，只被 createNew 调用。

```
ByteStreamMemoryBufferSource::ByteStreamMemoryBufferSource(UsageEnvironment& env,
  u_int8_t* buffer, u_int64_t bufferSize,
  Boolean deleteBufferOnClose,
  unsigned preferredFrameSize,
  unsigned playTimePerFrame)
  : FramedSource(env), fBuffer(buffer), fBufferSize(bufferSize), fCurIndex(0), fDeleteBufferOnClose(del
eteBufferOnClose),
  fPreferredFrameSize(preferredFrameSize), fPlayTimePerFrame(playTimePerFrame), fLastPlayTime(0)
{}
```

析构函数会根据 fDeleteBufferOnClose 来判断是否释放 fBuffer 指向的内存空间。前面说过构建类对象需要一个外部传入的 buffer，这是为了方便。这里告诉我们，这个外部发 buffer 应该是 new 出来的。思考一下，如果不是 new 出来的会怎么样？(执行期间发生错误)

```
ByteStreamMemoryBufferSource::~ByteStreamMemoryBufferSource()
{
  if (fDeleteBufferOnClose) delete[] fBuffer;
```


http://www.cnblogs.com/oloroso solym@sohu.com

```
}
```

createNew 创建对象

createNew 用于创建 ByteStreamMemoryBufferSource 对象。这是一个 static 方法。

如果传入的 buffer 是 NULL，则不创建。

```
ByteStreamMemoryBufferSource*
ByteStreamMemoryBufferSource::createNew(UsageEnvironment& env,
u_int8_t* buffer, u_int64_t bufferSize,
Boolean deleteBufferOnClose,
unsigned preferredFrameSize,
unsigned playTimePerFrame)
{
  if (buffer == NULL) return NULL;

  return new ByteStreamMemoryBufferSource(env, buffer, bufferSize, deleteBufferOnClose, preferredFrameS
ize, playTimePerFrame);
}
```

seekToByteAbsolute 寻跳到绝对位置

这个方法用于跳转当前索引的位置。当前索引 fCurIndex 是用来指示 getNextFrame 的时候从 fBuffer 中获取数据的起始位置。

这里是绝对跳转，即相对于 0 位置偏移 byteNumber。这里没有检测 byteNumber 为负数的情况，为什么呢？因为参数的类型是无符号类型。这里同时还设置了可到流字节数 fNumBytesToStream。

```
void ByteStreamMemoryBufferSource::seekToByteAbsolute(u_int64_t byteNumber, u_int64_t numBytesToStream)
{
  fCurIndex = byteNumber;   // 跳到 byteNumber 位置
  // 不能跳出 fBuffer
  if (fCurIndex > fBufferSize) fCurIndex = fBufferSize;
  // 设置 可到流字节数
  fNumBytesToStream = numBytesToStream;
  // 如果字节到流数大于 0，表明是受到限制的
  fLimitNumBytesToStream = fNumBytesToStream > 0;
}
```

seekToByteRelative 寻跳到相对位置

这个方法用于相对于当前的索引位置跳寻，同样也限制了不能跳出 fBuffer。要注意的是这里并没有改变 fnumBytesTOStream。

```
void ByteStreamMemoryBufferSource::seekToByteRelative(int64_t offset)
{
  // 偏移当前位置
  int64_t newIndex = fCurIndex + offset;
  // 不能那个跳出 fBuffer
  if (newIndex < 0) {
      fCurIndex = 0;
  }
  else {
      fCurIndex = (u_int64_t)offset;
      if (fCurIndex > fBufferSize) fCurIndex = fBufferSize;
  }
}
```

doGetNextFrame 实现获取下一帧

这个方法是这里最重要的了。之前在 FramedSource::getNextFrame 中提到过这个函数被调用，但它是个虚方法。这里对它的实现是针对与字节流源的实现。

首先是检查是否还有可供获取的数据，如果没有就调用 handleClosure 来做关闭处理咯，否则见下面描述。

再设置合适的帧大小，它要求符合的条件在代码中有，这里就不说了。

最后就是将数据安全的拷贝到 fTo 咯，前面说过 fTo 是由 getNextFrame 的参数 to 设置的。

数据获取结束了，但是还有一些工作要做。首先是对 FramedSource::fPresentationTime 成员的更新。如果是第一帧，那么就是当前时间。

现在来看一个表达式 fLastPlayTime = (fPlayTimePerFrame*fFrameSize) / fPreferredFrameSize;这个表达式的意思是，计算本帧的播放时间。注意这里不是实际的实际，而是根据 fPlayTimePerFrame 每一帧的播放时间（这是指一个最佳大小帧），与本帧实际大小和最佳帧大小计算出来的应该的实际。本帧的播放时间就是下一帧的上一帧播放时间，说的有点拗口了。fLastPlayTime 在对象初始话的时候是 0。

而 fPresentationTime 呈现时间就是每次加上上一帧的播放时间咯。

一帧获取完了，还有一件很重要的事情要做，那就是调用 FramedSource::afterGetting(this);

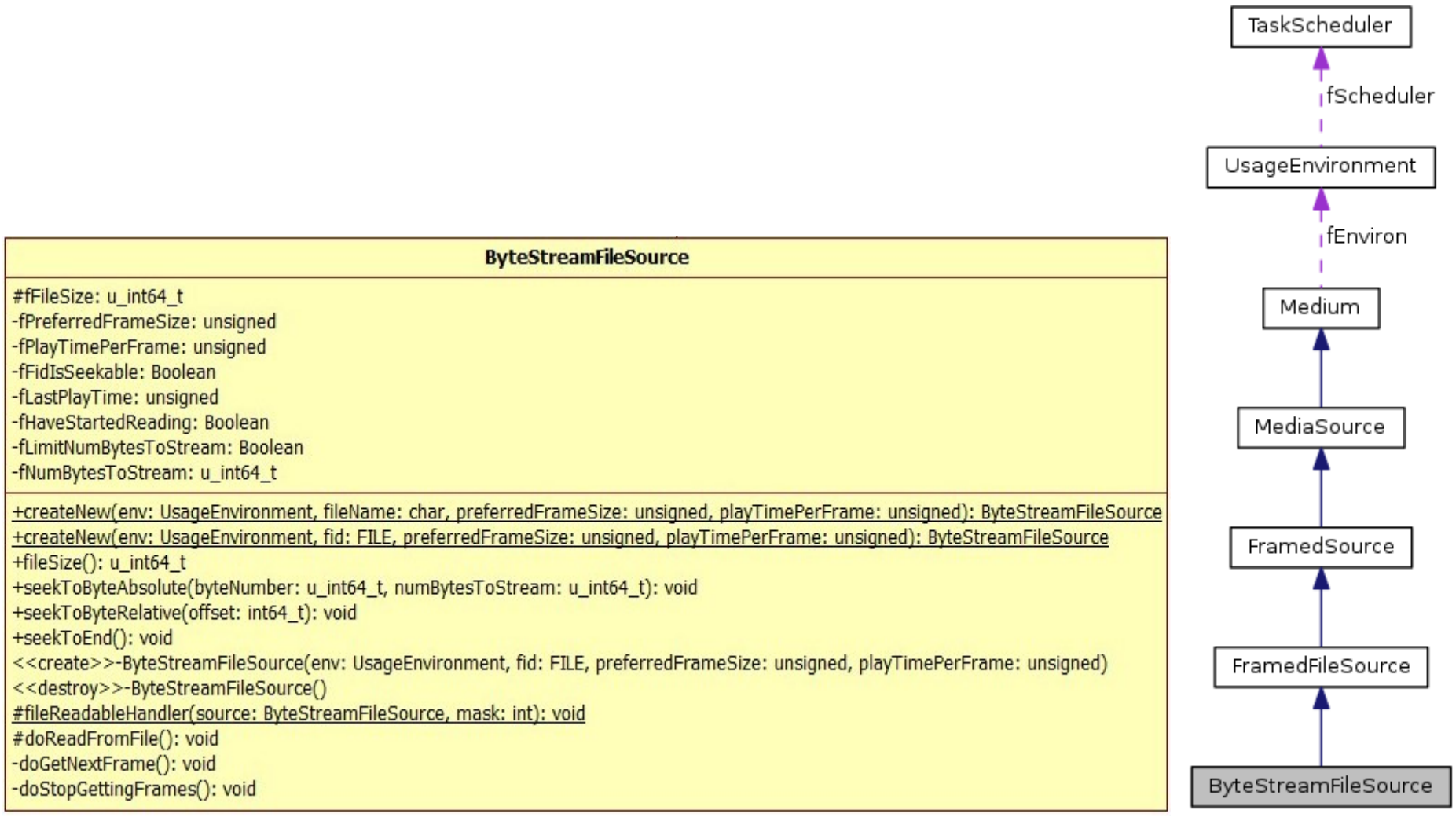
```
void ByteStreamMemoryBufferSource::doGetNextFrame()
{
  // 检查是否还有数据
  if (fCurIndex >= fBufferSize || (fLimitNumBytesToStream && fNumBytesToStream == 0)) {
     handleClosure(this);
     return;
  }

  // Try to read as many bytes as will fit in the buffer provided (or "fPreferredFrameSize" if less)
  // 尝试读取的字节数能够容纳在提供的 buffer(或它小于最佳帧尺寸)
  fFrameSize = fMaxSize; // 输出 帧尺寸=输入 最大尺寸
  // 如果字节到流数受到限制，那么一帧的大小不能超过这个值
  if (fLimitNumBytesToStream && fNumBytesToStream < (u_int64_t)fFrameSize) {
     fFrameSize = (unsigned)fNumBytesToStream;
  }
  // 也不能超过最佳帧大小
  if (fPreferredFrameSize > 0 && fPreferredFrameSize < fFrameSize) {
     fFrameSize = fPreferredFrameSize;
  }
  // 最多能获取到 fBuffer 结束的位置
  if (fCurIndex + fFrameSize > fBufferSize) {
     fFrameSize = (unsigned)(fBufferSize - fCurIndex);
  }
  // 开始获取数据了
  // fTo 是 FrameSource 的成员，由 getNextFrame 方法设置
  memmove(fTo, &fBuffer[fCurIndex], fFrameSize);
  fCurIndex += fFrameSize;
  fNumBytesToStream -= fFrameSize;

  // Set the 'presentation time':设置“呈现时间”:
  if (fPlayTimePerFrame > 0 && fPreferredFrameSize > 0) {
     if (fPresentationTime.tv_sec == 0 && fPresentationTime.tv_usec == 0) {
        // This is the first frame, so use the current time:
        // 这是第一帧，那么使用当前时间:
        gettimeofday(&fPresentationTime, NULL);
     }
     else {
        // Increment by the play time of the previous data:
        // //增量，根据前面数据的播放时间:
        unsigned uSeconds = fPresentationTime.tv_usec + fLastPlayTime;
        fPresentationTime.tv_sec += uSeconds / 1000000;
        fPresentationTime.tv_usec = uSeconds % 1000000;
     }
```


http://www.cnblogs.com/oloroso solym@sohu.com

```
      // Remember the play time of this data:记住这个数据的播放时间
      fLastPlayTime = (fPlayTimePerFrame*fFrameSize) / fPreferredFrameSize;
      fDurationInMicroseconds = fLastPlayTime;
    }
    else {
      // We don't know a specific play time duration for this data,
      // so just record the current time as being the 'presentation time':
      // 对于这个数据，我们不知道具体的播放的持续时间，所以只能记录当前的时间作为“呈现时间”:
      gettimeofday(&fPresentationTime, NULL);
    }

    // Inform the downstream object that it has data:通知下游对象，它具有数据:
    // 调用 获取后处理函数
    FramedSource::afterGetting(this);
}
```

8) ByteStreamFileSource 单文件字节流源

单文件字节流源继承自文件帧源 FramedFileSource。它使用一个打开的文件作为数据来源，文件的 IO 模式可能有同步和异步的情况，这里都对其使用了统一的接口处理。




```
// 单文件字节流源
class ByteStreamFileSource : public FramedFileSource {
public:
  static ByteStreamFileSource* createNew(UsageEnvironment& env,
      char const* fileName,
      unsigned preferredFrameSize = 0,
      unsigned playTimePerFrame = 0);
  // "preferredFrameSize" == 0 means 'no preference'
  // "playTimePerFrame" is in microseconds

  static ByteStreamFileSource* createNew(UsageEnvironment& env,
      FILE* fid,
      unsigned preferredFrameSize = 0,
      unsigned playTimePerFrame = 0);
  // an alternative version of "createNew()" that's used if you already have
```


http://www.cnblogs.com/oloroso solym@sohu.com

```
  // an open file.

  u_int64_t fileSize() const { return fFileSize; }
  // 0 means zero-length, unbounded, or unknown0 意味着零长度，无限的，或未知的

  void seekToByteAbsolute(u_int64_t byteNumber, u_int64_t numBytesToStream = 0);
  // if "numBytesToStream" is >0, then we limit the stream to that number of bytes, before treating it a
s EOF
  // 如果“numBytesToStream”> 0，那么我们限制流的字节数，将其视为 EOF 前

  // 跳寻到相对字节位置
  void seekToByteRelative(int64_t offset);
  // 跳寻到文件结尾
  void seekToEnd(); // to force EOF handling on the next read 强制 EOF 处理在下一个读

protected:
  ByteStreamFileSource(UsageEnvironment& env,
      FILE* fid,
      unsigned preferredFrameSize,
      unsigned playTimePerFrame);
  // called only by createNew()

  virtual ~ByteStreamFileSource();

  // 文件可读处理程序(注意这是 static 方法)
  static void fileReadableHandler(ByteStreamFileSource* source, int mask);
  // 执行从文件读取
  void doReadFromFile();

private:
  // redefined virtual functions:
  virtual void doGetNextFrame();
  virtual void doStopGettingFrames();

protected:
  u_int64_t fFileSize;   //文件大小

private:
  unsigned fPreferredFrameSize;   //最佳帧大小
  unsigned fPlayTimePerFrame;  //每帧播放时间
  Boolean fFidIsSeekable;      //文件指针 fFid 可 seek 定位？
  unsigned fLastPlayTime;      //最后播放时间
  Boolean fHaveStartedReading; //开始读取？
  Boolean fLimitNumBytesToStream; //到流字节数受到限制?(表示获取帧的时候要检查)
  u_int64_t fNumBytesToStream; // used iff "fLimitNumBytesToStream" is True
};
```

构造与析构

构造的时候对成员进行了初始化赋值。宏定义 READ_FROM_FILES_SYNCHRONOUSLY 是用于同步读写模式的时候用的(默认 1)。

```
ByteStreamFileSource::ByteStreamFileSource(UsageEnvironment& env, FILE* fid,
  unsigned preferredFrameSize,
  unsigned playTimePerFrame)
  : FramedFileSource(env, fid), fFileSize(0), fPreferredFrameSize(preferredFrameSize),
  fPlayTimePerFrame(playTimePerFrame), fLastPlayTime(0),
  fHaveStartedReading(False), fLimitNumBytesToStream(False), fNumBytesToStream(0)
{
#ifndef READ_FROM_FILES_SYNCHRONOUSLY//阻塞读模式
  makeSocketNonBlocking(fileno(fFid));
#endif
```

```
  // Test whether the file is seekable 测试文件是否为可 seek 定位
  if (SeekFile64(fFid, 1, SEEK_CUR) >= 0) {
     fFidIsSeekable = True;
     SeekFile64(fFid, -1, SEEK_CUR);
  }
  else {
     fFidIsSeekable = False;
  }
}
```

析构的时候采取了关闭文件的处理。要注意构造函数是有参数 Fid 的，其是外部传入的文件指针。

```
ByteStreamFileSource::~ByteStreamFileSource()
{
  if (fFid == NULL) return;

#ifndef READ_FROM_FILES_SYNCHRONOUSLY
  envir().taskScheduler().turnOffBackgroundReadHandling(fileno(fFid));  //如果是异步模式，关闭后台处理读操作
#endif

  CloseInputFile(fFid); //关闭文件(非 stdin)
}
```

makeSocketNonBlock 是将文件描述符设置为非阻塞模式的函数，之前说过。

SeekFile64 和 CloseInputFile 在文件 live555sourcecontrol\liveMedia\InputFile.cpp 中定义。

SeekFile64 文件跳寻

clearerr(stream)函数的作用是清除由 stream 指向的文件流的文件尾标识和错误标识。它没有返回值，也未定义任何错误。你可以通过使用它从文件流的错误状态中恢复。

fflush（）函数冲洗流中的信息，如果成功刷新,fflush 返回 0。指定的流没有缓冲区或者只读打开时也返回 0 值。返回 EOF 指出一个错误。注意:如果 fflush 返回 EOF,数据可能由于写错误已经丢失。当设置一个重要错误处理器时,最安全的是用 setvbuf 函数关闭缓冲或者使用低级 I/O 例程,如 open、close 和 write 来代替流 I/O 函数。

fseeko 和 fseek 的区别在于 fseeko 适用于 64 位环境。

```
int64_t SeekFile64(FILE *fid, int64_t offset, int whence)
{
  clearerr (fid);
  fflush(fid);
#if (defined(__WIN32__) || defined(_WIN32)) && !defined(_WIN32_WCE)
  return _lseeki64(_fileno(fid), offset, whence) == (int64_t)-1 ? -1 : 0;
#else
#if defined(_WIN32_WCE)
  return fseek(fid, (long)(offset), whence);
#else
  return fseeko(fid, (off_t)(offset), whence);
#endif
#endif
}
```

CloseInputFile 关闭输入文件

这只是避免了关闭标准输入文件流。

```
void CloseInputFile(FILE* fid)
{
  // Don't close 'stdin', in case we want to use it again later.
  // 不要关闭'标准输入'，万一我们以后要再次使用它。
```


http://www.cnblogs.com/oloroso solym@sohu.com

```
  if (fid != NULL && fid != stdin) fclose(fid);
}
```

createNew 方法

createNew 方法用于创建 ByteStreamFileSource 对象，这也说明了这是抽象的结束，呵呵。这是 static 方法。

createNew 方法有两个，参数有点区别。一个传入了文件名，一个是打开的文件指针。

两个都做了获取文件大小的操作，在文件类型是管道等情况下，其结果是 0。

函数 OpenInputFile 和 GetFileSize 在文件 live555sourcecontrol\liveMedia\InputFile.cpp 中定义。

```
ByteStreamFileSource*
ByteStreamFileSource::createNew(UsageEnvironment& env, char const* fileName,
unsigned preferredFrameSize,
unsigned playTimePerFrame)
{
  FILE* fid = OpenInputFile(env, fileName);
  if (fid == NULL) return NULL;

  ByteStreamFileSource* newSource
     = new ByteStreamFileSource(env, fid, preferredFrameSize, playTimePerFrame);
  newSource->fFileSize = GetFileSize(fileName, fid);

  return newSource;
}
```

传入打开文件指针模式

```
ByteStreamFileSource*
ByteStreamFileSource::createNew(UsageEnvironment& env, FILE* fid,
unsigned preferredFrameSize,
unsigned playTimePerFrame)
{
  if (fid == NULL) return NULL;

  ByteStreamFileSource* newSource = new ByteStreamFileSource(env, fid, preferredFrameSize, playTimePerF
rame);
  newSource->fFileSize = GetFileSize(NULL, fid);

  return newSource;
}
```

OpenInputFile 打开输入文件

注意这里打开文件的模式是二进制只读模式。这个函数也告诉了我们，默认情况下 ByteStreamFileSource 对象操作的文件指针是只用于读的。

```
FILE* OpenInputFile(UsageEnvironment& env, char const* fileName)
{
  FILE* fid;

  // Check for a special case file name: "stdin"检查是否有特殊的文件名：“stdin”
  if (strcmp(fileName, "stdin") == 0) {
     fid = stdin;
#if (defined(__WIN32__) || defined(_WIN32)) && !defined(_WIN32_WCE)
     // _setmode 用于设置文件的打开模式(include <io.h>)
     _setmode(_fileno(stdin), _O_BINARY); // convert to binary mode 转换为二进制模式
#endif
  }
  else {
     fid = fopen(fileName, "rb"); //二进制只读模式
```

```
      if (fid == NULL) {
         env.setResultMsg("unable to open file \"", fileName, "\"");
      }
   }

   return fid;
}
```

GetFileSize 获取文件大小

在 Win32WCE 下只能使用 seek 到文件尾来获取文件大小，其它环境还可以使用 stat。

```
u_int64_t GetFileSize(char const* fileName, FILE* fid)
{
   u_int64_t fileSize = 0; // by default

   if (fid != stdin) {
#if !defined(_WIN32_WCE)
      if (fileName == NULL) {
#endif
         if (fid != NULL && SeekFile64(fid, 0, SEEK_END) >= 0) {
            fileSize = (u_int64_t)TellFile64(fid); //获取当前位置
            if (fileSize == (u_int64_t)-1) fileSize = 0; // TellFile64() failed
            SeekFile64(fid, 0, SEEK_SET);   //回到文件头
         }
#if !defined(_WIN32_WCE)
      }
      else {
         struct stat sb;
         if (stat(fileName, &sb) == 0) {
            fileSize = sb.st_size;
         }
      }
#endif
   }

   return fileSize;
}
```

TellFile64 当前读写位置

Ftello 适用于 64 位环境。

```
// 获取打开文件当前读写指针位置
int64_t TellFile64(FILE *fid)
{
   clearerr(fid);
   fflush(fid);
#if (defined(__WIN32__) || defined(_WIN32)) && !defined(_WIN32_WCE)
   return _telli64(_fileno(fid));
#else
#if defined(_WIN32_WCE)
   return ftell(fid);
   // 函数 ftell 用于得到文件位置指针当前位置相对于文件首的偏移字节数。在随机方式存取文件时，由于文件位置频繁的前后移动，程序不容易确定文件的当前位置。
#else
   return ftello(fid);
#endif
#endif
}
```


http://www.cnblogs.com/oloroso solym@sohu.com

seekToByteAbsolute 寻跳到绝对位置

将文件指针的读写位置设置到相对文件头 byteNumber 位置。这个在前面 ByteStreamMemoryBufferSource 类中有相类似的实现。

```
void ByteStreamFileSource::seekToByteAbsolute(u_int64_t byteNumber, u_int64_t numBytesToStream)
{
  SeekFile64(fFid, (int64_t)byteNumber, SEEK_SET);
  //更新可到流字节数
  fNumBytesToStream = numBytesToStream;
  fLimitNumBytesToStream = fNumBytesToStream > 0;
}
```

seekToByteRelative 寻跳到相对位置

相对于当前读写位置偏移 offset 字节。

```
void ByteStreamFileSource::seekToByteRelative(int64_t offset)
{
  SeekFile64(fFid, offset, SEEK_CUR);
}
```

seekToEnd 跳寻到文件尾

直接跳到文件末尾。

```
void ByteStreamFileSource::seekToEnd()
{
  SeekFile64(fFid, 0, SEEK_END);
}
```

fileReadableHandler 文件读处理

文件可读处理程序(注意这是 static 方法)。这个函数是为了方便 env 作为延时任务调度用的(C++的类成员函数指针与普通指针是有区别，这里用静态方法来方便调度，source 参数将会是 this)。

如果当前不是正在获取数据，那么调用 source->doStopGettingFrames()去停止获取数据。而在非阻塞模式下，ByteStreamFileSource::doStopGettingFrames()才会有具体的操作。后面再说。

fileReadableHandler 是 ByteStreamFileSource::doGetNextFrame 方法在 fFid 为非阻塞(异步)模式下才会调用的。

```
void ByteStreamFileSource::fileReadableHandler(ByteStreamFileSource* source, int /*mask*/)
{
  if (!source->isCurrentlyAwaitingData()) {
     // source 当前没有正在读取数据，停止获取帧
     source->doStopGettingFrames(); // we're not ready for the data yet 目前我们还没有准备好数据
     return;
  }
  // 从文件读取数据
  source->doReadFromFile();
}
```

doStopGettingFrames 执行停止获取帧

fHaveStartedReading 置为 false，表示没有开始读取操作。

```
void ByteStreamFileSource::doStopGettingFrames()
{
#ifndef READ_FROM_FILES_SYNCHRONOUSLY
  // 如果不是阻塞模式，那么从任务调度器中关闭对 fileno(fFid)的后台读处理
  envir().taskScheduler().turnOffBackgroundReadHandling(fileno(fFid));
  fHaveStartedReading = False;
```


http://www.cnblogs.com/oloroso solym@sohu.com

```
#endif
}
```

doGetNextFrame 执行获取下一帧

doGetNextFrame 是由 GetNextFrame 调用的。GetNextFrame 是从基类 FrameSource 继承来的。在 GetNextFrame 中有这么一句 fIsCurrentlyAwaitingData = True，表示当前正在等待数据(获取数据)，之后调用的 doGetNextFrame。

在 GetNextFrame 中是不分阻塞与非阻塞的，但是 doGetNextFrame 是分的。在阻塞模式下，直接调用 doReadFromFile 去从文件读取数据。而在非阻塞模式下，是将读取的操作 fileReadableHandler 作为任务加入到 env 的任务调度器中了。

在非阻塞模式下，先通过 fHaveStartedReading 成员判读是否有读取任务正在进行，有的情况下不应该再去调度一个文件读取操作。

```
void ByteStreamFileSource::doGetNextFrame()
{
  if (feof(fFid) || ferror(fFid) || (fLimitNumBytesToStream && fNumBytesToStream == 0)) {
     handleClosure(this);  //文件读取完了，处理关闭时操作
     return;
  }

#ifdef READ_FROM_FILES_SYNCHRONOUSLY //阻塞模式
  doReadFromFile();  //执行读文件
#else//非阻塞模式
  if (!fHaveStartedReading) {
     // Await readable data from the file:等待从文件读取数据:
     envir().taskScheduler().turnOnBackgroundReadHandling(fileno(fFid)/*mask*/,
        (TaskScheduler::BackgroundHandlerProc*)&fileReadableHandler, this/*source*/);
     fHaveStartedReading = True;
  }
#endif
}
```

doReadFromFile 执行从文件读取

doReadFromFile 这个方法和前面介绍过的 ByteStreamMemoryBufferSource::doGetNextFrame 方法很像。前面基本是一样的，只是在获取数据的来源有点区别。设置'presentation time'的方法也一致。但是在完成获取之后调用 afterGetting 有所不同。

doReadFromFile 在调用 afterGetting 的时候根据阻塞与非阻塞采取了不同的策略。阻塞的情况是直接调用，就不多说了。在**阻塞情况**下，其是作为任务添加到了 env 的任务调度器，这个调度的任务赋值给了 fNextTask(nextTask()返回引用)，其是从 Medium 中继承来的。在这里注释中说了，这样做是为了避免无限递归调用，为什么会出现这个情况呢？

现在来分析一下，从 getNextFrame 开始。

首先 getNextFrame 设置了获取后的回调，并设置了当前等待数据的标识 fIsCurrentlyAwaitingData。之前的时候说这个成员是用来标识当前正在读取数据，这个解释不是很准确，这个只是指示当前有去读取数据的操作，而这个操作不一定正在执行。

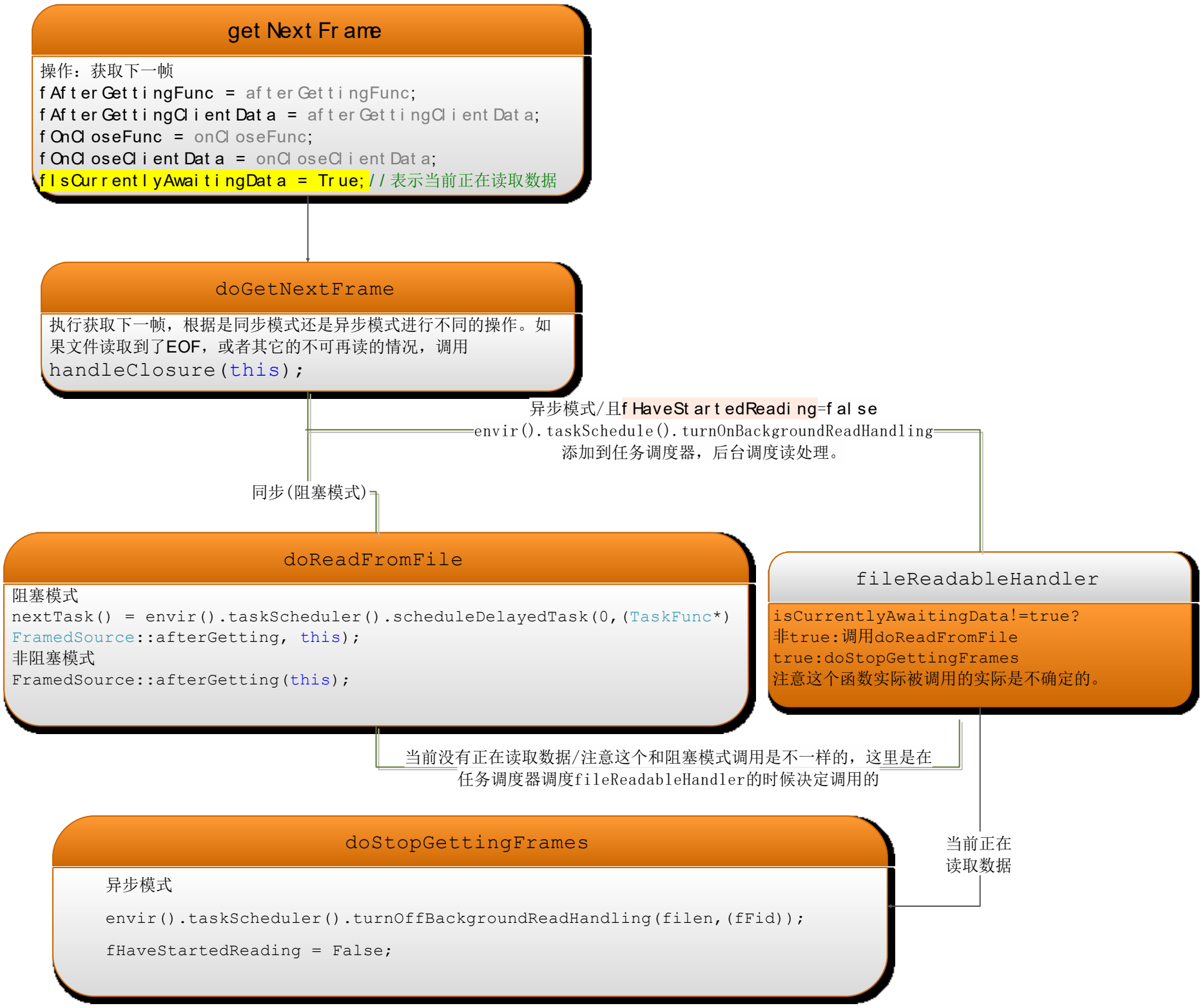
然后其调用了 doGetNextFrame 去执行获取下一帧操作，当是异步模式的时候，其把 fileReadableHandler 方法作为一个任务添加到了 env 的任务调度器。在同步(阻塞)模式下，其直接调用 doReadFromFile 来获取数据。

异步模式下，在某个时刻，任务调度器的 doEventLoop 循环的时候，调度了这个任务(有数据可读)，那么在有数据正在读取的时候，会调用 doStopGetFrame 方法将对 fFid 的可读操作从中任务调度中移除。如果没有正在读取数据，那么就调用 doReadFromFile 去真正的读取数据。

在 doReadFromFile 读取完一帧数据后，要调用获取后的操作 afterGetting 了。这里有两种情况，同步(阻塞)模式下是添加到任务调度器，异步模式是直接调用。在 afterGetting 有一个操作 fIsCurrentlyAwaitingData = False;然后再使用函数指针 fAfterGettingFunc 调用的函数。问题应该在这里了，如果 afterGetting 指向的函数调用了 getNextFrame 的话就有可能无限递归了。

如果 afterGetting 是立即调用的，而不是添加到调度器的，那么它将立即执行，如果其又调用了 getNextFrame，那么 fIsCurrentlyAwaitingData 将变为 true，然后又调用 doGetNextFrame，接着又调用 fileReadableHandler 然后又调用 doReadFromFile，接着又到了 afterGetting。这里形成了一个循环。

下面是大概的流程图。




```
void ByteStreamFileSource::doReadFromFile()
{
  // Try to read as many bytes as will fit in the buffer provided (or "fPreferredFrameSize" if less)
  // 尝试读取的字节数能够容纳在提供的 buffer(或它小于最佳帧尺寸)
  if (fLimitNumBytesToStream && fNumBytesToStream < (u_int64_t)fMaxSize) {
     fMaxSize = (unsigned)fNumBytesToStream;
  }
  if (fPreferredFrameSize > 0 && fPreferredFrameSize < fMaxSize) {
     fMaxSize = fPreferredFrameSize;
  }
#ifdef READ_FROM_FILES_SYNCHRONOUSLY //阻塞读取
  fFrameSize = fread(fTo, 1, fMaxSize, fFid);
#else
  if (fFidIsSeekable) {
     fFrameSize = fread(fTo, 1, fMaxSize, fFid);
  } else {
     // For non-seekable files (e.g., pipes), call "read()" rather than "fread()", to ensure that the r
ead doesn't block:
     // 对于非可 seek 定位的文件(例如：管道),调用"read()"而不是"fread()"以确保该读不会阻塞：
     fFrameSize = read(fileno(fFid), fTo, fMaxSize);
  }
#endif
  if (fFrameSize == 0) { //没有读取到数据(文件结尾)
     handleClosure(this);
     return;
```


http://www.cnblogs.com/oloroso solym@sohu.com

```
    }
    fNumBytesToStream -= fFrameSize;//可到流的字节数更新

    // Set the 'presentation time':设置'呈现'时间
    if (fPlayTimePerFrame > 0 && fPreferredFrameSize > 0) {
        if (fPresentationTime.tv_sec == 0 && fPresentationTime.tv_usec == 0) {
            // This is the first frame, so use the current time:
            gettimeofday(&fPresentationTime, NULL);
        }
        else {
            // Increment by the play time of the previous data:
            unsigned uSeconds = fPresentationTime.tv_usec + fLastPlayTime;
            fPresentationTime.tv_sec += uSeconds / 1000000;
            fPresentationTime.tv_usec = uSeconds % 1000000;
        }

        // Remember the play time of this data:
        fLastPlayTime = (fPlayTimePerFrame*fFrameSize) / fPreferredFrameSize;
        fDurationInMicroseconds = fLastPlayTime;
    }
    else {
        // We don't know a specific play time duration for this data,
        // so just record the current time as being the 'presentation time':
        gettimeofday(&fPresentationTime, NULL);
    }

    // Inform the reader that he has data:
#ifdef READ_FROM_FILES_SYNCHRONOUSLY
    // To avoid possible infinite recursion, we need to return to the event loop to do this:
    // 为了避免可能出现的无限递归，我们需要回到事件循环执行此操作：
    nextTask() = envir().taskScheduler().scheduleDelayedTask(0,
        (TaskFunc*)FramedSource::afterGetting, this);
#else
    // Because the file read was done from the event loop, we can call the
    // 'after getting' function directly, without risk of infinite recursion:
    //因为文件的读取是在事件循环完成后，我们可以直接调用函数 afterGetting，没有无限递归的危险：
    FramedSource::afterGetting(this);
#endif
}
```

9) ByteStreamMultiFileSource 多文件字节流源

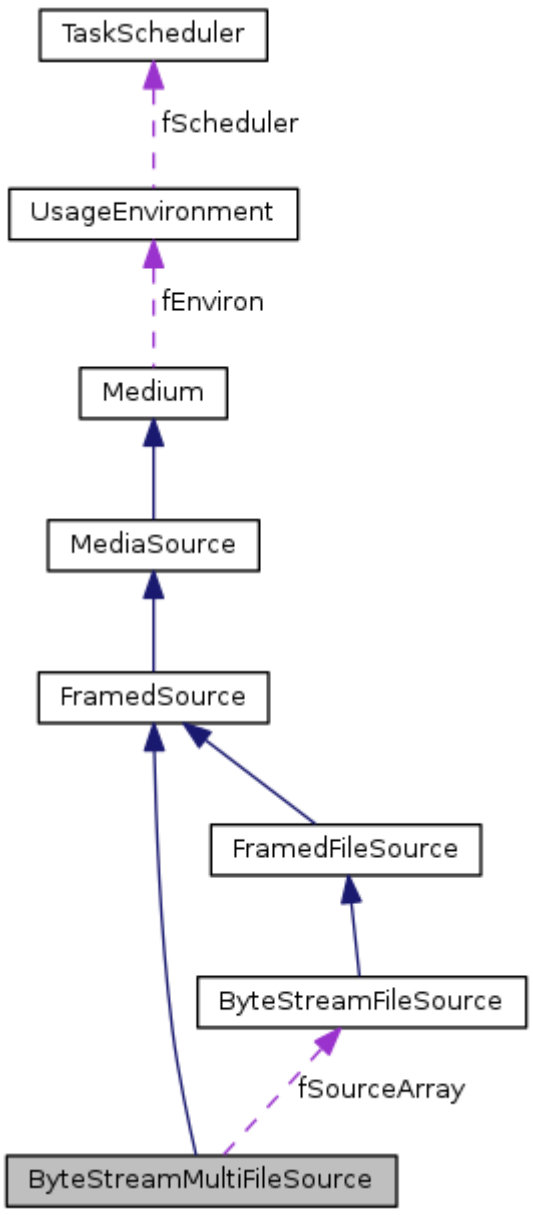
前面刚说了单文件字节流源 ByteStreamFileSource，就是为这个做准备的。这里要说明一点，单文件字节流源是继承自文件帧源，间接继承自帧源类的。而多文件字节流源是直接继承自帧源类的。定义在文件 live555sourcecontrol\liveMedia\include\ByteStreamMultiFileSource.hh 中。

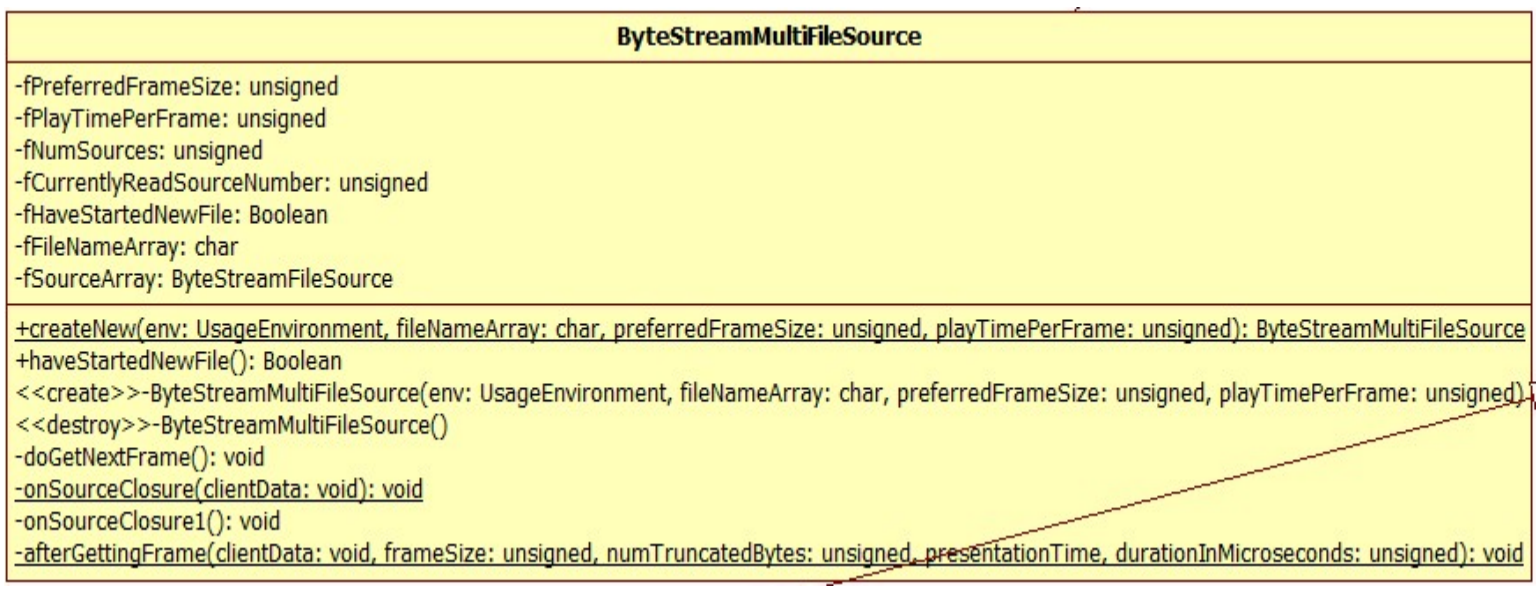
单文件字节流源只使用了一个文件作为数据的来源，而多文件字节流源使用了一组文件作为数据的来源。其实质是多个单文件字节流源的集合(这么说也不是很准确，实质上没有直接创建多个 ByteStreamFileSource 对象)。

在ByteStreamMultiFileSource类中有两个指针，如下，是用来保存文件名和ByteStreamFileSource对象的指针的。其中文件名数组fFileNameArray和文件字节流源对象指针数组fSourceArray是在构造函数中动态创建的，在析构函数中释放。
ByteStreamFileSource对象则是按需要进行动态创建的，见doGetNextFrame等方法。

```
char const** fFileNameArray; //文件名数组
ByteStreamFileSource** fSourceArray;  //文件字节流源数组
```


http://www.cnblogs.com/oloroso solym@sohu.com






```
// 多文件字节流源
class ByteStreamMultiFileSource : public FramedSource {
public:
  static ByteStreamMultiFileSource*
     createNew(UsageEnvironment& env, char const** fileNameArray,
     unsigned preferredFrameSize = 0, unsigned playTimePerFrame = 0);
  // A 'filename' of NULL indicates the end of the array
  // fileNameArray 数组的结束是一个为 NULL 的元素

  Boolean haveStartedNewFile() const { return fHaveStartedNewFile; }
  // True iff the most recently delivered frame was the first from a newly-opened file
  // true 如果在最近提供帧是首次从一个新打开的文件来的

protected:
  ByteStreamMultiFileSource(UsageEnvironment& env, char const** fileNameArray,
     unsigned preferredFrameSize, unsigned playTimePerFrame);
  // called only by createNew()
```


http://www.cnblogs.com/oloroso solym@sohu.com

```
  virtual ~ByteStreamMultiFileSource();

private:
  // redefined virtual functions:
  virtual void doGetNextFrame();

private:
  // 关闭时调用，注意是 static 方法(原因还是因为作为回调函数)
  static void onSourceClosure(void* clientData);
  // 关闭时的真实调用
  void onSourceClosure1();
  // 获取一帧后调用，作为回调使用
  static void afterGettingFrame(void* clientData,
     unsigned frameSize, unsigned numTruncatedBytes,
  struct timeval presentationTime,
     unsigned durationInMicroseconds);

private:
  unsigned fPreferredFrameSize;   //最佳帧大小
  unsigned fPlayTimePerFrame;     //每帧播放时间
  unsigned fNumSources; //源编号
  unsigned fCurrentlyReadSourceNumber;   //当前读取源编号
  Boolean fHaveStartedNewFile; //开始新文件?
  char const** fFileNameArray; //文件名数组
  ByteStreamFileSource** fSourceArray;   //文件字节流源数组
};
```

ByteStreamMultiFileSource 构造函数

ByteStreamMultiFileSource 的构造是一个 protected 方法，其仅被静态的 createNew 方法调用。

参数 fileNameArray 是一个指向元素类型为 char const *的数组的指针，数组的每一个元素是一个文件名字符串的指针。fileNameArray 指向的数组被规定为以 NULL 作为结尾标志。(如：fileNameArray={"fileA","fileB","fileC",NULL})

构造函数中对文件名进行了拷贝，拷贝所用的内存空间来自动态申请，这将在析构中释放。

对于 fSourceArray 指向的数组，其也是来自动态申请。并且所有的元素都置为 NULL，这是 ByteStreamFileSource 对象是按需生成的证据。

```
ByteStreamMultiFileSource
::ByteStreamMultiFileSource(UsageEnvironment& env, char const** fileNameArray,
unsigned preferredFrameSize, unsigned playTimePerFrame)
: FramedSource(env),
fPreferredFrameSize(preferredFrameSize), fPlayTimePerFrame(playTimePerFrame),
fCurrentlyReadSourceNumber(0), fHaveStartedNewFile(False)
{
  // Begin by counting the number of sources:开始通过计算源数
  for (fNumSources = 0;/**/; ++fNumSources) {
     if (fileNameArray[fNumSources] == NULL) break;
  }

  // Next, copy the source file names into our own array:
  // 接下来，复制源文件名到我们自己的数组:
  fFileNameArray = new char const*[fNumSources];
  if (fFileNameArray == NULL) return;
  unsigned i;
  for (i = 0; i < fNumSources; ++i) {
     fFileNameArray[i] = strDup(fileNameArray[i]);
  }

  // Next, set up our array of component ByteStreamFileSources
```


http://www.cnblogs.com/oloroso solym@sohu.com

```
  // Don't actually create these yet; instead, do this on demand
  // 接下来，建立我们的 ByteStreamFileSources 的数组。实际上并不创建对象;而是为此随需应变
  fSourceArray = new ByteStreamFileSource*[fNumSources];
  if (fSourceArray == NULL) return;
  for (i = 0; i < fNumSources; ++i) {
     fSourceArray[i] = NULL;
  }
}
```

ByteStreamMultiFileSource 析构函数

析构函数相对较简单，只是对构造函数中动态申请的内存空间的释放和对所有 ByteStreamFileSource 的 close 操作。

这里还要说一点，Medium::close 是一个静态的方法。如果参数为 NULL，就什么也不做。如果不是，则是从 env.livMedia->mediaTable 表中移除这个媒体对象地址，然后将其 delete。

```
ByteStreamMultiFileSource::~ByteStreamMultiFileSource()
{
  unsigned i;
  for (i = 0; i < fNumSources; ++i) {
     Medium::close(fSourceArray[i]); //关闭源
  }
  delete[] fSourceArray;//释放源数组

  for (i = 0; i < fNumSources; ++i) {
     delete[](char*)(fFileNameArray[i]);//释放源名称
  }
  delete[] fFileNameArray;  //释放源名称数组
}
```

createNew 创建对象

createNew 调用构造函数来创建对象，其是 static 方法。

```
ByteStreamMultiFileSource* ByteStreamMultiFileSource
::createNew(UsageEnvironment& env, char const** fileNameArray,
unsigned preferredFrameSize, unsigned playTimePerFrame)
{
  // 创建多文件字节流源
  ByteStreamMultiFileSource* newSource
     = new ByteStreamMultiFileSource(env, fileNameArray,
     preferredFrameSize, playTimePerFrame);

  return newSource;
}
```

afterGettingFrame 获取一帧后调用

这里先说这个函数，是为了说 doGetNextFrame 做准备的。

这里要注意的一点是，这个函数的参数 clientData，前面分析代码的时候没有说一些类成员 fXXXClientData 和对应的 fXXXFunc 的关系，是因为前面读代码的时候没有遇到这里这种容易混淆的情况，所有没有仔细去分析说明。这里说一下，它们一个代表用于回调的对象，一个表示回调的函数。一般是这样的 clientData->func。可以这么说，但这是不太准确的，因为 func 一般是一个静态的函数，而 clientData 一般作为其参数，即 func(ClinetData)。通畅 func 还有一个名称接近的非静态成员方法 func0 等，一般 func 是为了回调方便而设置的，实质还是调用的 clientData->func0。这是通常做法，不是绝对的。

在这个函数中，给相关成员赋值后调用了 FramedSource::afterGetting(source);这个静态方法，实质上是调用了

```
(*(source->fAfterGettingFunc))(source->fAfterGettingClientData,
        source->fFrameSize, source->fNumTruncatedBytes,
        source->fPresentationTime,
        source->fDurationInMicroseconds);
```


http://www.cnblogs.com/oloroso solym@sohu.com

注意上面的 source 是 afterGettingFrame(void* clientData,…中的 clientData。

```
void ByteStreamMultiFileSource
::afterGettingFrame(void* clientData,
unsigned frameSize, unsigned numTruncatedBytes,
struct timeval presentationTime,
  unsigned durationInMicroseconds)
{
  ByteStreamMultiFileSource* source
     = (ByteStreamMultiFileSource*)clientData;
  // 相关成员赋值
  source->fFrameSize = frameSize;
  source->fNumTruncatedBytes = numTruncatedBytes;
  source->fPresentationTime = presentationTime;
  source->fDurationInMicroseconds = durationInMicroseconds;
  // 真正的获取后调用(source->fAfterGettingFunc)注意两个 fAfterGettingFunc 隶属的对象不同
  FramedSource::afterGetting(source);
}
```

doGetNextFrame 执行获取下一帧

这个函数很关键，在这一系列的帧源相关类中都很关键。

我们可以从这里看出，ByteStreamFileSource 对象的创建是在进行的，而且是有序的(在从一个单文件字节流源中获取结束后(文件读完等)才创建下一个)。注意，在没有文件源可用的时候调用了 handleClosure(this)来做关闭处理，这个不是关闭某个源时处理，而是这个多文件字节流源对象关闭时的处理。

实质的数据获取工作交给了当前使用的 ByteStreamFileSource 对象来完成，其调用 FramedSource:: getNextFrame 来实现的。这里要注意的就是是哪个对象调用了哪个方法，在下面将进行分析。(关闭时调用的回调也类似，就不分析了)

我们从 this 调用 doGetNextFrame 开始。（假设一个 ByteStreamMultiFileSource 对象调用了 getNextFrame 方法）

1、 this->doGetNextFrame 方法，获取(或创建)了一个 fSourceArray 数组中一个 ByteStreamFileSource 对象的地址，保存在变量 source 中。

2、 调用 source-> getNextFrame 方法。这个方法的第三个和第四个参数要关心下。其相当于执行了以下语句：

```
source->fAfterGettingFunc = this-> afterGettingFrame

source->fAfterGettingClientData = this;
```

3、 source-> getNextFrame 中对上述成员赋值后，还调用了 source->doGetNextFrame，在这里实质上会去调用 source->ByteStreamFileSource::doReadFromFile，然后间接调用了 source->FramedSource::afterGetting(this)。

4、 在 source->afterGetting(this)注意这里的参数 this 实质上是 source。在这个调用中，又进行了这样的调用

```
source->fAfterGettingFunc))(source->fAfterGettingClientData,...
```

根据前面的赋值，我们知道其实质上是这样的调用

```
this->afterGettingFrame(this)
```

5、 上面的调用，实质上又相当于是 this->fAfterGettingFunc))(this->fAfterGettingClientData,...的调用。但是这两个成员，默认是 NULL 的。

```
void ByteStreamMultiFileSource::doGetNextFrame()
{
  do {
     // First, check whether we've run out of sources:首先，检查是否我们已经用完了来源
     if (fCurrentlyReadSourceNumber >= fNumSources) break;

     fHaveStartedNewFile = False;
     ByteStreamFileSource*& source
        = fSourceArray[fCurrentlyReadSourceNumber];
```


http://www.cnblogs.com/oloroso solym@sohu.com

```
    if (source == NULL) {
        // The current source hasn't been created yet.  Do this now:
        // 当前源尚未创建。现在这样做：(创建文件字节流源)
        source = ByteStreamFileSource::createNew(envir(),
            fFileNameArray[fCurrentlyReadSourceNumber],
            fPreferredFrameSize, fPlayTimePerFrame);
        if (source == NULL) break;
        fHaveStartedNewFile = True;  //指示这是一个新开始的文件
    }

    // (Attempt to) read from the current source.（尝试）从当前源中读取
    source->getNextFrame(fTo, fMaxSize,
        afterGettingFrame/*获取一帧后调用赋值给 source->fAfterGettingFunc*/, this,
        onSourceClosure/*关闭源时调用*/, this);
    /*
    这里要详细说一下，当 source->getNextFrame 调用的时候会
    source->fAfterGettingFunc = this->afterGettingFunc;//afterGettingFrame
    source->fAfterGettingClientData = this->afterGettingClientData;//this
    然后调用 source->doGetNexFrame 成功获取一帧数据的时候(见 doReadFromFile()实现)，会调用
    FramedSource::afterGetting(this)，进而有这样的调用 source->fAfterGettingFunc))(source->fAfterGettin
gClientData,...
    那么就也就是调用了 this->afterGettingFrame(this)
    而在 this->afterGettingFrame 中，对相关的成员进行了赋值，然后调用了 FramedSource::afterGetting(source/
*这实质是这里的 this* /)
    于是就又去调用了 this->fAfterGettingFunc))(this->fAfterGettingClientData,...
    应当是没有继续调用了的，因为在多文件字节流源类中没有对 fAfterGettingFunc 进行修改，其应为 NULL
    */

    return;
  } while (0);

  // An error occurred; consider ourselves closed://发生错误;认为自己该关闭:
  handleClosure(this);
}
```

onSourceClosure 源关闭时调用

这个方法也是静态方法，其作用还是用于回调。这个在 doGetNextFrame 中出现过了。

其实质上就是用去 clientData 调用 ByteStreamMultiFileSource 的成员方法 onSourceClosure1。

```
void ByteStreamMultiFileSource::onSourceClosure(void* clientData)
{
  ByteStreamMultiFileSource* source
      = (ByteStreamMultiFileSource*)clientData;
  source->onSourceClosure1();  //关闭当前源，指向下一个源
}
```

这个方法说一下，它的作用是关闭当前的源，然后将当前读取的源索引指向下一个，方便下一个 doGetNextFrame 调用的时候使用下一个源。

Medium::close 这个静态方法在前面讲过，它将参数关联条目从 env.liveMedia->MediaTable 表中移除，并将关联的对象进行 delete 释放。

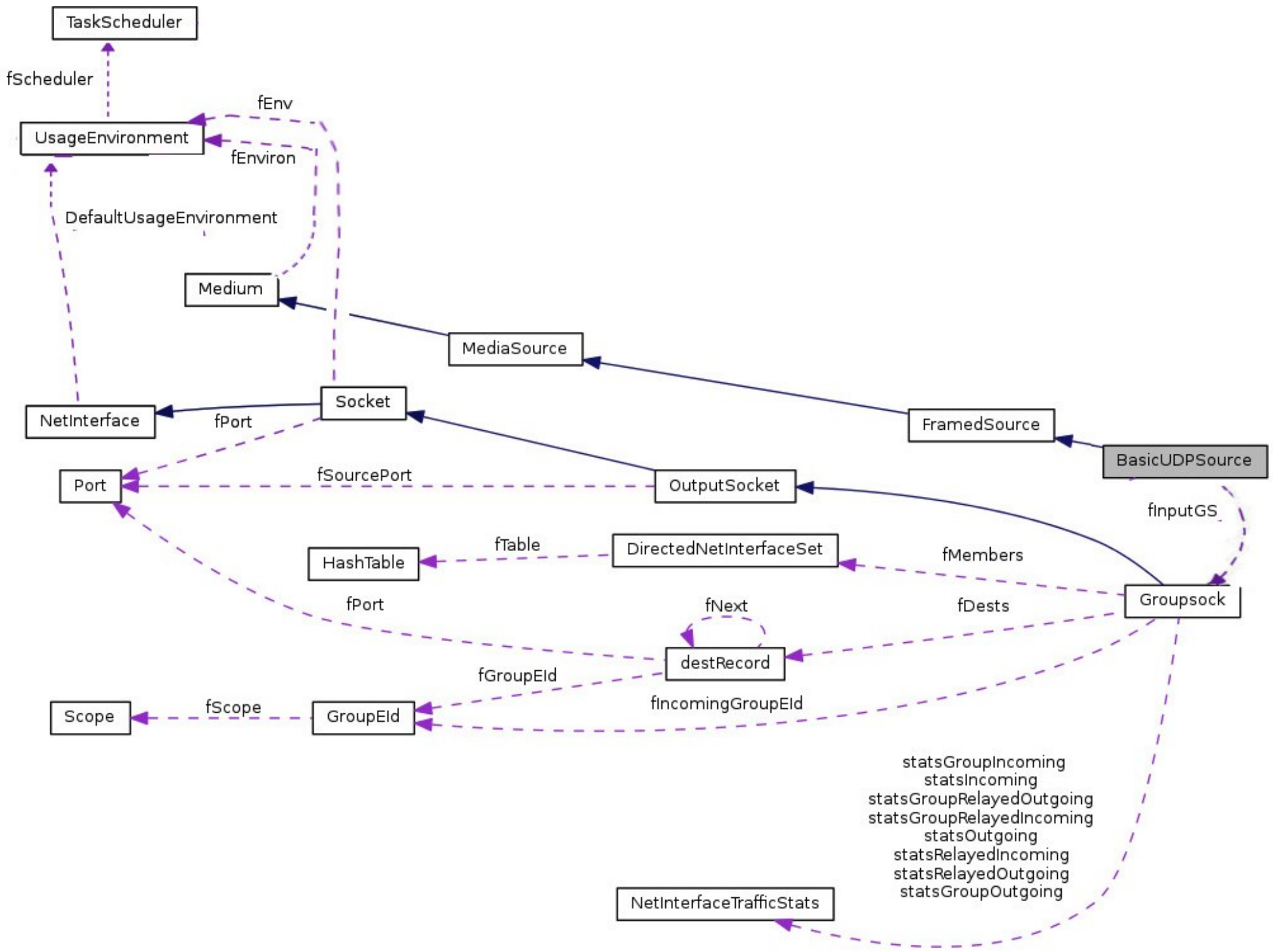
这个函数结束前调用了 doGetNextFrame 方法去获取下一帧的数据，这应该是用于去试探是否还有文件源可用的，因为没有文件源可用的时候，会调用 handleClosure(this)来进行这个多文件字节流源对象的关闭处理。

```
void ByteStreamMultiFileSource::onSourceClosure1()
{
  // This routine was called because the currently-read source was closed
  // (probably due to EOF).  Close this source down, and move to the
  // next one:
```

```
    // 这个例程被调用，因为当前读源被关闭（可能是由于 EOF）。
    // 关闭这个来源下来，并移动到下一个：
    ByteStreamFileSource*& source
        = fSourceArray[fCurrentlyReadSourceNumber++];
    Medium::close(source);
    source = NULL;

    // Try reading again:尝试再次读取
    doGetNextFrame();
}
```

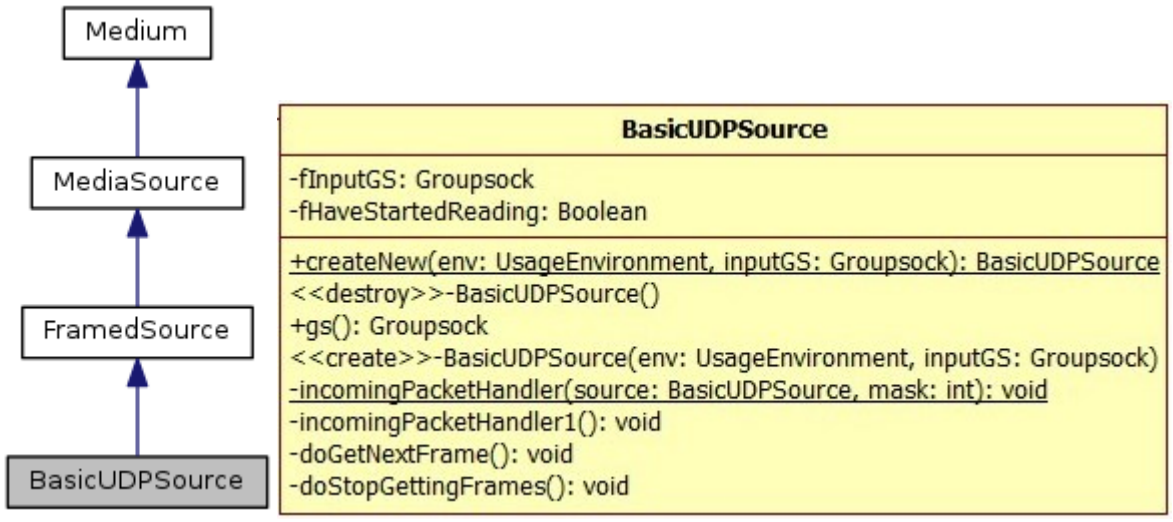
10） BasicUDPSource 基本 UDP 源

基本 UDP 源类定义在文件 live555sourcecontrol\liveMedia\include\BasicUDPSource.hh 中，它本身的定义并不复杂，但是因为它直接或间接的使用到了前面介绍的很多类，所以还是有点复杂的，看下面的图。



BasicUDPSource 类继承自 FrameSource 类，间接继承自 MediaSource 类。其内部构成比之前说的帧源相关的类要简单，但并不意味这它就简单了，因为它使用了 Groupsock 类，而 Groupsock 类是比较复杂的。BasicUDPSource 是接收 UDP 数据包来作为媒体源的。

http://www.cnblogs.com/oloroso solym@sohu.com




```
class BasicUDPSource : public FramedSource {
public:
  static BasicUDPSource* createNew(UsageEnvironment& env, Groupsock* inputGS);

  virtual ~BasicUDPSource();

  Groupsock* gs() const { return fInputGS; }

private:
  BasicUDPSource(UsageEnvironment& env, Groupsock* inputGS);
  // called only by createNew()

  static void incomingPacketHandler(BasicUDPSource* source, int mask);
  void incomingPacketHandler1();

private: // redefined virtual functions:
  virtual void doGetNextFrame();
  virtual void doStopGettingFrames();

private:
  Groupsock* fInputGS;   // 输入 Groupsock
  Boolean fHaveStartedReading; //已经开始读取数据
};
```

BasicUDPSource 构造与析构

构造的时候先调用基类的构造初始，然后调用 increaseReceiveBufferTo 来扩增了 inputGS->socketNum()的缓冲区到 50KB 大小。然后再将 inputGS->socketNum()改为非阻塞模式。

```
BasicUDPSource::BasicUDPSource(UsageEnvironment& env, Groupsock* inputGS)
: FramedSource(env), fInputGS(inputGS), fHaveStartedReading(False)
{
  // Try to use a large receive buffer (in the OS):尽量使用大接收缓冲区（在 OS 中）：
  increaseReceiveBufferTo(env, inputGS->socketNum(), 50 * 1024);

  // Make the socket non-blocking, even though it will be read from only asynchronously, when packets ar
rive.
  //使套接字非阻塞，即使它只会从异步读取，在数据包到达时。
  // The reason for this is that, in some OSs, reads on a blocking socket can (allegedly) sometimes bloc
k,
  // even if the socket was previously reported (e.g., by "select()") as having data available.
  //这样做的原因是，在一些操作系统，读取上阻塞套接字会（据说）有一些时间阻塞，即使以前套接字 report（例如，通过 select（））其有可用数据。
  // (This can supposedly happen if the UDP checksum fails, for example.)
  //（例如：如果 UDP 校验失败，可以推测发生。）
  makeSocketNonBlocking(fInputGS->socketNum());
}
```

析构的时候就比较简单了，它将对 inputGS->socketNum()的后台读处理操作从 env(fEnviron)的任务队列中移除。

http://www.cnblogs.com/oloroso solym@sohu.com

```
BasicUDPSource::~BasicUDPSource()
{
  // 取消对 fInputGS->socketNum()的后台读操作
  envir().taskScheduler().turnOffBackgroundReadHandling(fInputGS->socketNum());
}
```

createNew 创建对象

注意是静态的就可以了，这个方法在各个类中的出现次数太多了。

```
BasicUDPSource* BasicUDPSource::createNew(UsageEnvironment& env,
  Groupsock* inputGS)
{
  return new BasicUDPSource(env, inputGS);
}
```

incomingPacketHandler 传入包处理程序

static 方法，用于添加到任务队列，实质上还是调用的 incomingPacketHandler1。

```
// 传入包处理(static 修饰，作为回调)
void BasicUDPSource::incomingPacketHandler(BasicUDPSource* source, int /*mask*/)
{
  source->incomingPacketHandler1();
}
```

incomingPacketHandler1 真正的传入包处理程序

这个函数最终会加入到 fEnviron 的任务队列中进行后台可读监控(select)，当有数据到达的时候，实际上会调用这个成员函数来处理到达的数据。

fTo 是从基类 FramedSource 中继承来的，其在 FramedSource::getNextFrame(to…调用的时候被赋值 to。传入数据接收后调用了 afterGetting 方法，这是从基类继承来的 static 方法，本类中没有重写它。

```
// 实际的传入包处理
void BasicUDPSource::incomingPacketHandler1()
{
  // 当前没有数据正在被读取
  if (!isCurrentlyAwaitingData()) return; // we're not ready for the data yet

  // Read the packet into our desired destination:
  // 读取数据包到我们期望的目标处
  struct sockaddr_in fromAddress;
  if (!fInputGS->handleRead(fTo/*读取到的数据存放*/, fMaxSize, fFrameSize/*读到字节数*/, fromAddress)) re
turn;

  // Tell our client that we have new data:告诉我们的客户，我们有新的数据：
  afterGetting(this); // we're preceded by a net read; no infinite recursion 我们之前有一个网络读取;没有无限递归
}
```

doGetNextFrame 执行获取下一帧

此处调用了 envir().taskScheduler().turnOnBackgroundReadHandling 来将 fInputGS->socketNum()加入到 select 的监控列表中，对其可读进行监控。在任务队列循环处理中，有数据到达的时候就回调 incomingPacketHandler 来处理了。

```
void BasicUDPSource::doGetNextFrame()
{
  if (!fHaveStartedReading) {
     // Await incoming packets:等待传入包
     // 将对传入 UDP 数据包的处理程序 incomingPacketHandler 添加到任务队列
     envir().taskScheduler().turnOnBackgroundReadHandling(fInputGS->socketNum(),
```


http://www.cnblogs.com/oloroso solym@sohu.com

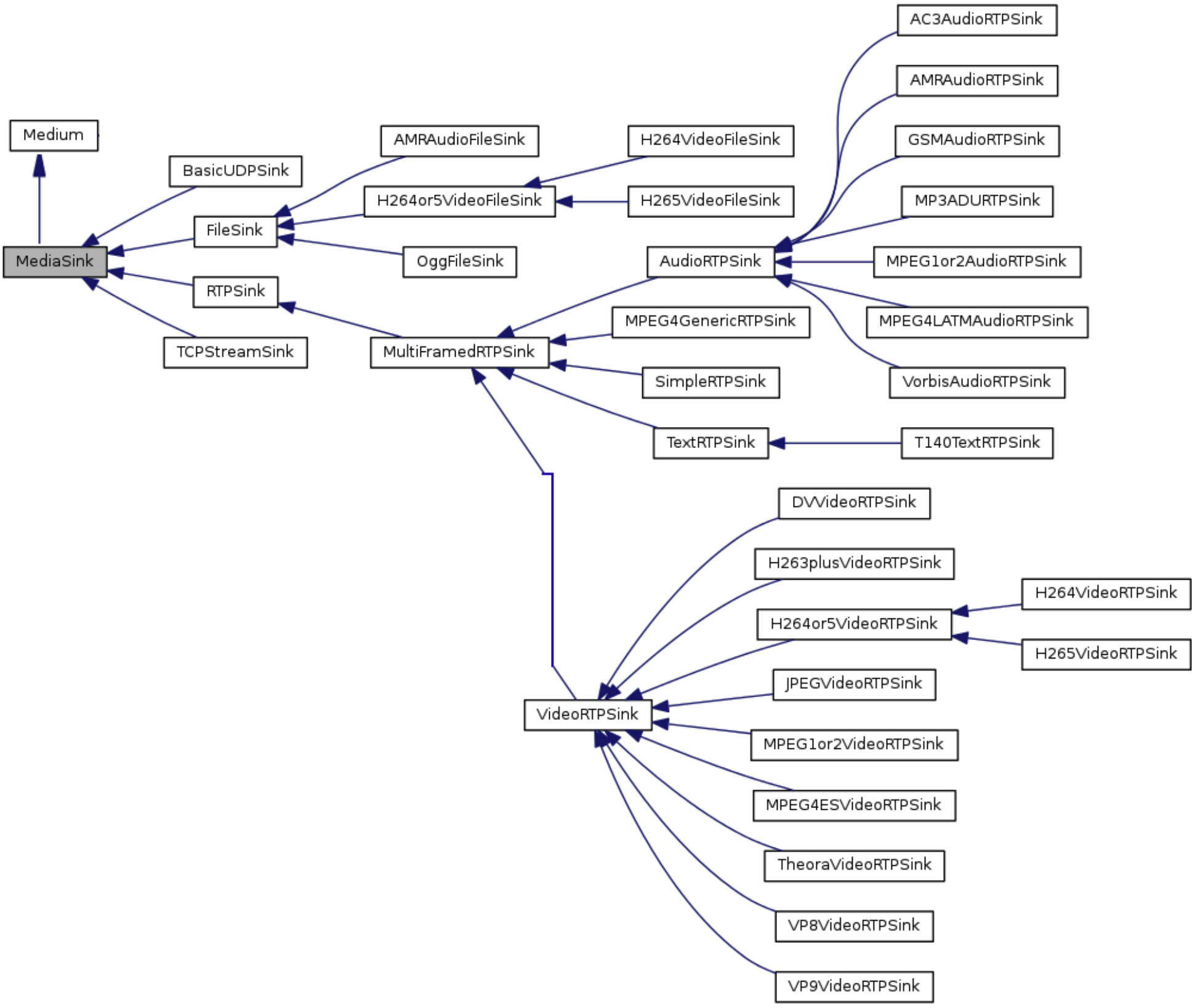
```
        (TaskScheduler::BackgroundHandlerProc*)&incomingPacketHandler, this);
    fHaveStartedReading = True;  //标识已经开始读取数据
  }
}
```

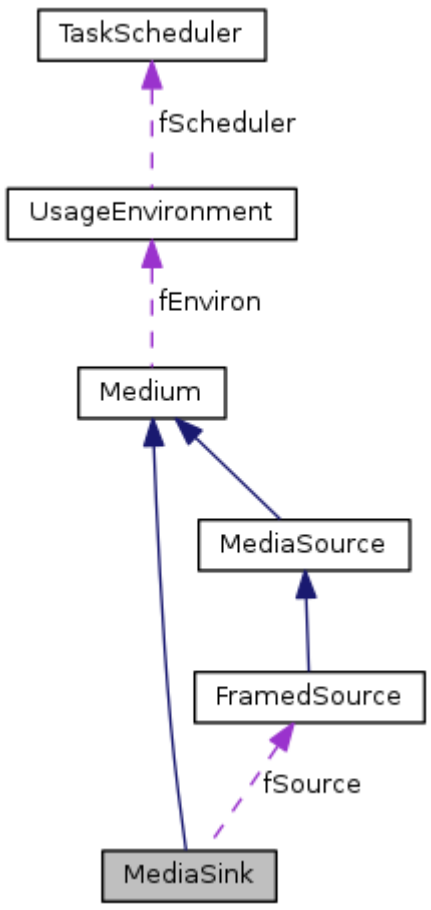
doStopGettingFrames 执行停止获取帧

将对 fInputGS->socketNum()的可读监控从 shelet 监控列表中移除。

```
void BasicUDPSource::doStopGettingFrames()
{
  // 移除对 fInputGS->socketNum()的可读监控操作
  envir().taskScheduler().turnOffBackgroundReadHandling(fInputGS->socketNum());
  fHaveStartedReading = False;
}
```

11) MediaSink





http://www.cnblogs.com/oloroso solym@sohu.com