https://www.jianshu.com/p/b08729905a8c

http://blog.csdn.net/yuanchunsi/article/details/72876230

https://blog.csdn.net/liukun321/article/details/38059797

https://blog.csdn.net/ithzhang/article/details/38686477

http://blog.csdn.net/qq_29350001/article/details/77962082

http://www.cnblogs.com/oloroso/p/4599955.html

https://blog.csdn.net/huangyifei_1111/article/details/74668842

https://www.wolfcstech.com/2017/09/07/live555_src_analysis_servermediasession/

http://xingyunbaijunwei.blog.163.com/blog/#m=0&t=1&c=fks_084071082087086069082082085095083080084064092086080 0

https://www.wolfcstech.com/2017/09/08/live555_src_analysis_start_streaming/

Live555 源码主要由八个部分组成：UsageEnvironment，BasicUsageEnvironment，groupsock，liveMedia，mediaServer，proxyServer，testProgs，WindowsAudioInputDevice。

子模块	文件个数	代码量（行）
UsageEnvironment	3	162
BasicUsageEnvironment	6	1187
groupsock	8	2672
liveMedia	168	49552
mediaServer	2	332
proxyServer	1	251
WindowsAudioInputDevice	4	1037
testProgs	32	6510
总共	224	61703

UsageEnvironment： UsageEnvironment和TaskScheduler类用在调度不同事件，实现异步读取事件的句柄的设置以及错误信息的输出。还有一个HashTable 类定义了一个通用的hash 表，其它代码要用到这个表。这些都是抽象类，在应用程序中基于这些类来实现自己的子类。

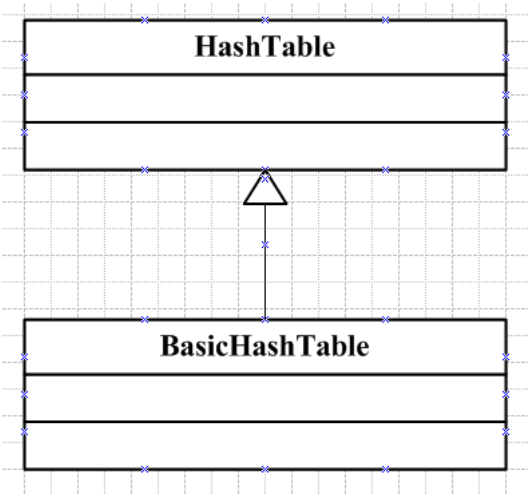
groupsock：是对网络接口的封装，用于收发数据包。正如名字本身，groupsock 主要是面向多播数据的收发的，它也同时支持单播数据的收发。

liveMedia：定义一个类栈，根类是Medium类，这些类针对不同的流媒体类型和编码。

BasicUsageEnvironment：定义一个usageEnvironment的简单实现, 这个里面除了有一个TaskScheduler以外，都是一些说明性的东西。TaskSheduler里面是一些调度相关的函数，其中 doEventLoop是主控函数，定义为纯虚函数。这个库利用Unix 或者Windows 的控制台作为输入输出，处于应用程序原形或者调试的目的，可以用这个库用户可以开发传统的运行与控制台的应用。

testProgs：目录下是一个简单的实现，使用了BasicUsageEnvironment来展示如何使用这些库，测试用例。

////////////



virtual void* Add(char const* key, void* value) = 0;
完成增加和修改功能；
修改：若存在相同的key值，返回key对应的原始值，用value替换原始值；
增加：若不存在key，新增一个键-值对，插入到链表头部，返回NULL；

virtual Boolean Remove(char const* key) = 0;
完成删除功能；
删除：若存在键为key的键-值对，删除该键-值对，返回True；若不存在返回Flase；

virtual void* Lookup(char const* key) const = 0;
完成查找功能；
查找：若存在键为key的键-值对，则返回该键-值对；否则返回NULL；

virtual unsigned numEntries() const = 0;
返回容器中，键-值对的数量；

void* RemoveNext();
删除下一个键-值对；从当前容器中的第一个键-值对开始删除；

void* getFirst();
返回当前容器中的第一个键-值对；

增：对key值进行哈希计算，获取索引，在索引处，将该键-值对新增到链表头；
改：对key值进行哈希计算，获取索引，遍历索引处的链表，当key值相同时，替换key对应的值；
删：对key值进行哈希计算，获取索引，遍历索引处的链表，当key值相同时，删除该键-值对；

若哈希表中的键-值对数目超过了阈值(初始阈值为4*3)，则需要重建哈希表，每次重建，哈希表规模都扩大到原来的4倍，并将原先的哈希表拷贝到新建哈希表中对应位置。重建哈希表后，会修改哈希函数。

```
38      fDownShift(28), fMask(0x3), fKeyType(keyType) {
93  unsigned randomIndex(uintptr_t i) const {
94      return (unsigned)(((i*1103515245) >> fDownShift) & fMask);
95  }
236     fMask = (fMask<<2)|0x3;
```

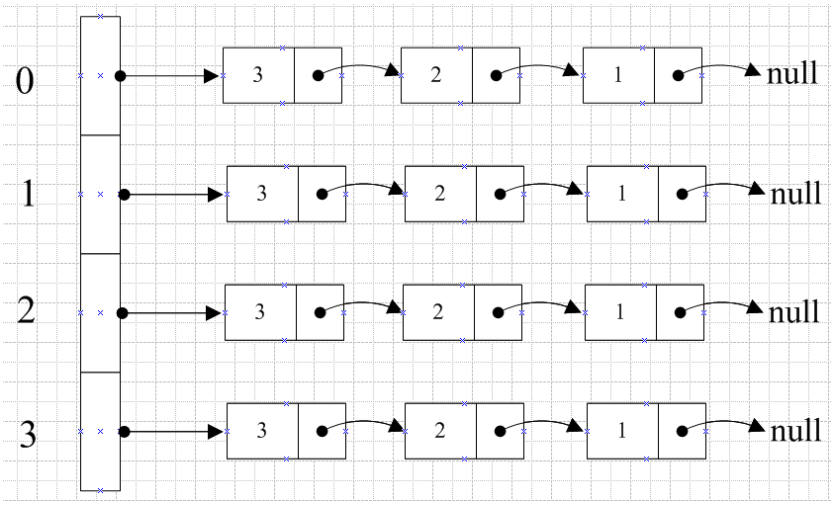
如上截图，fMask初始值为0x3，哈希函数只会产生0，1，2，3共四个索引值；当哈希表需要重建时，fMask左移2位或上原始值，此时fMask为0xf，此时，哈希函数会产生哈希值0，1，2，... ，15；如下图，哈希索引相同时，将新的键-值对插入链表头。

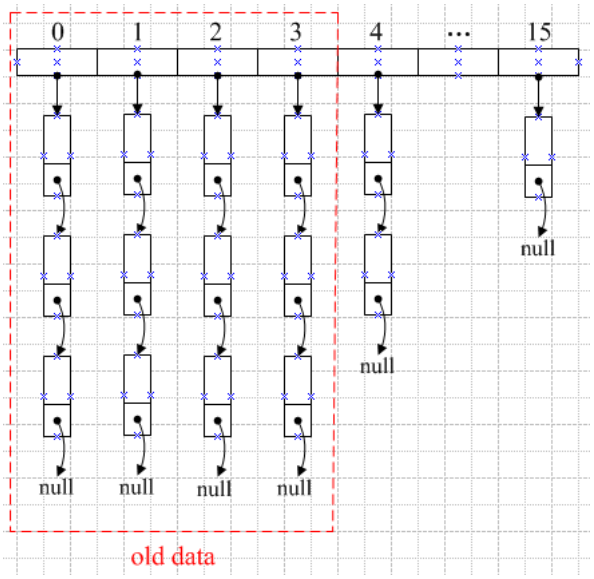
```
::insertNewEntry(unsigned index, char const* key) {
  TableEntry* entry = new TableEntry();
  entry->fNext = fBuckets[index];
  fBuckets[index] = entry;
```

当哈希冲突时，采用现行探测法解决冲突问题，将具有相同哈希值的键-值对放到单项链表中，每次新增时，都将新来的键-值对放入链表头部。如下图，0号索引处，最先来的1号键-值对在队列尾部，最后来的键-值对在队列头部。

假设在哈希算法极端差的情况下：初始时刻，所有的key均哈希到所以0处，则哈希值0处的链表长度最大为初始阈值(12)；扩大4倍后，所有的key均哈希到哈希值0处，则哈希值4处的链表长度最大为初始阈值*4；再次扩大4倍后，所有可以均哈希到哈希值0处，则哈希值16处的链表长度最大为4*4*初始阈值 ；扩大N次后，最坏哈希时，最大链表长度为：

$$4^N*12$$





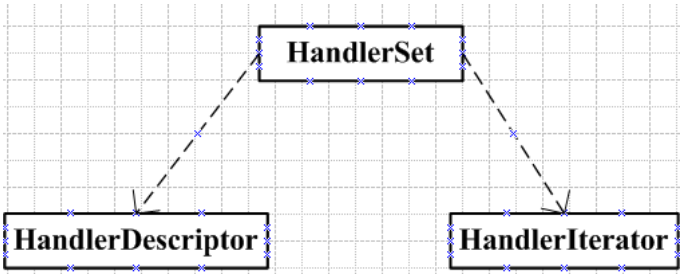
补充知识：

友元类，在类A中以public、private或者protect申明friend class xxx;这样在类xxx中就可以访问类A中的所有成员函数和成员变量了；破坏了类的封装性和隐藏性；

补充知识：

函数调用时，形参只是实参的一份拷贝，对形参的修改不会影响到实参；指针作为参数专递时，形参也只是实参的一份拷贝，但是形参和实参指向相同的内存位置，所以通过形参对该内存位置的修改会影响到实参，但是修改形参指针本身不会影响实参；指针也是一种数据类型，也有值和地址，特殊性是他的值是内存地址；指针赋值，是把一个指针指向的地址，赋值给另一个指针。

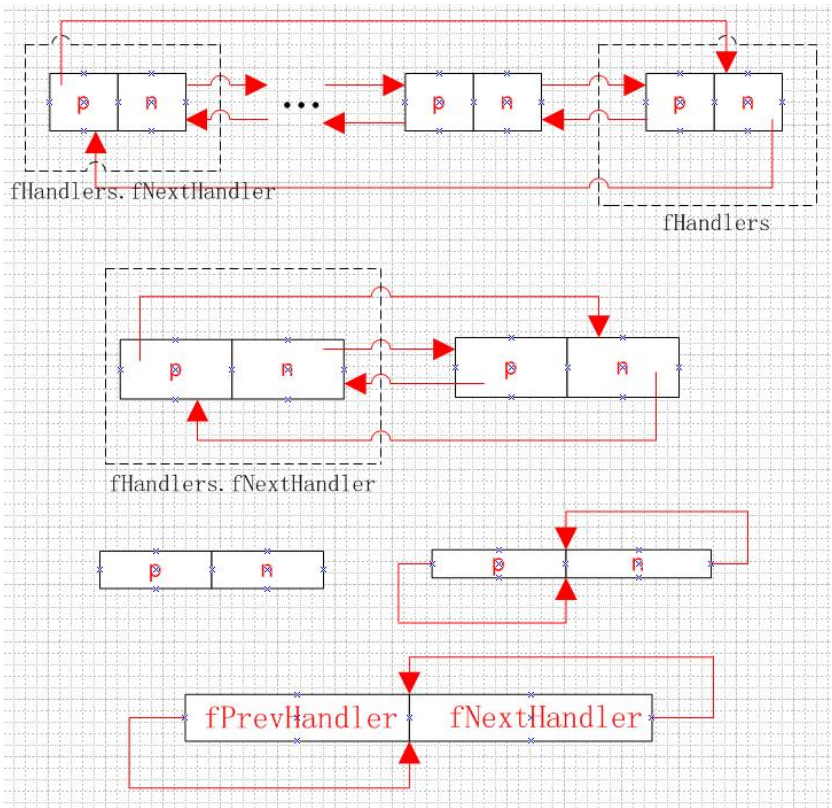
HandlerSet：双向链表，**用来处理IO事件**；新事件总是加入链表头；



HandlerSet类就是一个双向链表集合；

HandlerDescriptor类用来实现双向链表；

HandlerIterator类是为了能够递归的处理双向链表中的节点；




```
HandlerDescriptor::HandlerDescriptor(HandlerDescriptor* nextHandler)
  : conditionSet(0), handlerProc(NULL) {
  // Link this descriptor into a doubly-linked list:
  if (nextHandler == this) { // initialization
    fNextHandler = fPrevHandler = this;
  } else {
    fNextHandler = nextHandler;
    fPrevHandler = nextHandler->fPrevHandler;
    nextHandler->fPrevHandler = this;
    fPrevHandler->fNextHandler = this;
  }
}
HandlerSet::HandlerSet()
  : fHandlers(&fHandlers) {
  fHandlers.socketNum = -1; // shouldn't ever get looked at, but in case.
}
```

```
183   if (handler == NULL) { // No existing handler, so create a new descr:
184     handler = new HandlerDescriptor(fHandlers.fNextHandler);
185     handler->socketNum = socketNum;
186   }
```

从上图可以看到，在HandlerSet的构造函数中，通过fHandlers(&fHandlers)初始化了一个HandlerDescriptor对象。通过观察HandlerDescriptor类的构造函数，得出结论：fHandlers的fNextHandler和fPrevHandler均指向自己，fHandlers为双向链表的第一个节点；在HandlerDescriptor的构造函数中，对形参nextHandler的值进行操作。指针作

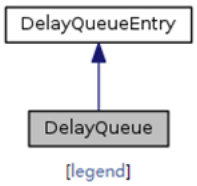
为形参时，对其值进行操作，会影响实参的对应值。所以HandlerDescriptor构造函数中修改nextHandler，对应的HandlerSet中的fHandlers也被对应修改了；修改后，fHandlers和其指针如上图所示，所以fHandlers是双向链表的尾节点。

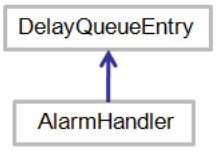
```
//向双向链表中添加IO事件，socketNum为文件描述符，conditionSet为读|写|异常状态，handlerProc为回调函数，clientData为用户数据，回调函数带回给用户
void assignHandler(int socketNum, int conditionSet,                    TaskScheduler::BackgroundHandlerProc* handlerProc, void* clientData);

//从双向链表中，删除socketNum对应的IO事件
void clearHandler(int socketNum);

//更新IO事件，将套接字oldSocketNum的事件的套接字修改为newSocketNum
void moveHandler(int oldSocketNum, int newSocketNum);
```

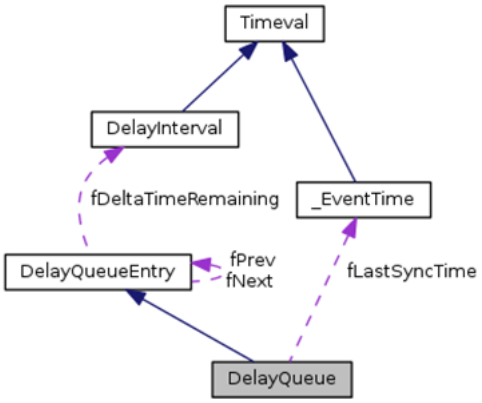
补充知识：

const修饰类的成员函数时，表示该函数不会修改类的成员；const成员函数，只能调用const成员函数

DelayQueue：一种有序队列，特点就是只有在队列中的元素到期后才能取出，head头是最先过期的元素；**在live555中主要用来实现延时事件。**







Timeval：时间处理基类；

DelayInterval：时间间隔，用来处理相对时间；

_EventTime：延时事件的基准时间；

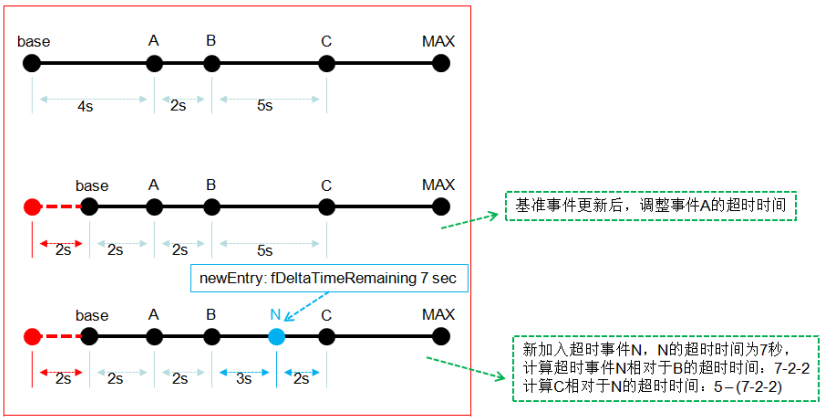
DelayQueueEntry：描述一个延时事件；

DelayQueue：延时事件组成的双向列表；

AlarmHandler：定时器时间；

```
//向双向列表中，增加延时事件，加入后才开始计算超时
void addEntry(DelayQueueEntry* newEntry); // returns a token for the entry
//更新定时事件
void updateEntry(DelayQueueEntry* entry, DelayInterval newDelay);
void updateEntry(intptr_t tokenToFind, DelayInterval newDelay);
```

```
//从双向列表中删除entry事件
void removeEntry(DelayQueueEntry* entry); // but doesn't delete it
DelayQueueEntry* removeEntry(intptr_t tokenToFind); // but doesn't delete it
//下一个超时事件的时间间隔，距离现在基准事件的时间
DelayInterval const& timeToNextAlarm();
//处理定时事件
void handleAlarm();
```



假如我们要描述一个事件发生的时间，可以有两种方法：

a、一种方法直接描述事件发生的绝对时间；

b、另一种则是可以描述和另一事件发生的相对时间。

而 LIVE555中采用的就是后者。在LIVE555中，首先将所有的事件点以发生时间的先后进行排序，然后每个事件对应的时间都是相对于前一事件发生的时间差。比如B事件中存储的时间就是A事件触发后，再去触发B事件所需要的时间。这样，我们每次去查询这个队列中是否有事件被触发的时候，就只需要查询整个队列中的第一个事件就可以了。第一个事件的发生时间，是相对于基准事件base的时间差，基准事件总是在变动的，每次变动都需要重新计算第一个超时事件的时间差。

```
DelayQueue::DelayQueue()
    : DelayQueueEntry(ETERNITY) {
    fLastSyncTime = TimeNow();
}
```

```
AlarmHandler(TaskFunc* proc, void* clientData, DelayInterval timeToDelay)
    : DelayQueueEntry(timeToDelay), fProc(proc), fClientData(clientData) {
}
```

```
DelayInterval timeToDelay((long)(microseconds/1000000), (long)(microseconds%1000000));
AlarmHandler* alarmHandler = new AlarmHandler(proc, clientData, timeToDelay);
fDelayQueue.addEntry(alarmHandler);
```

live555中的DelayQueue，初始化的时候，设置了最大超时时间：

2147483647s + (1000000 - 1)us

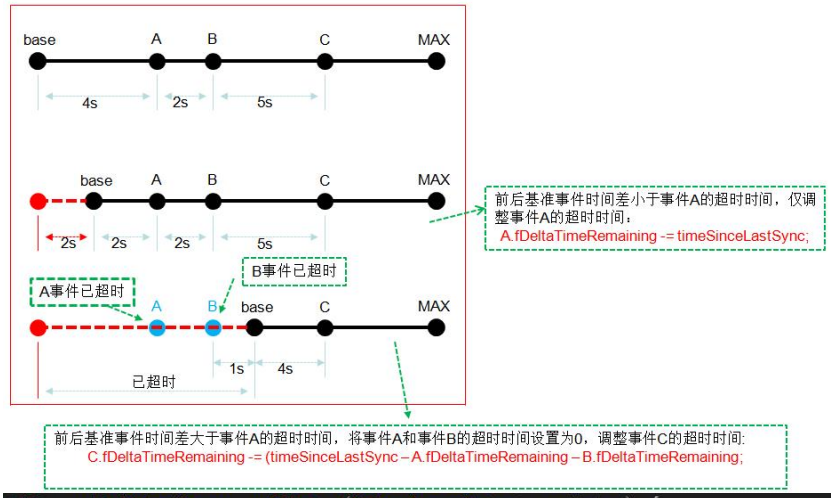
需要1635年才会超时。根据需要生成延时事件AlarmHandler，将该定时器时间加入到DelayQueue中。

```
202 void DelayQueue::synchronize() {
203   // First, figure out how much time has elapsed since the last sync
204   _EventTime timeNow = TimeNow();
205   if (timeNow < fLastSyncTime) {
206     // The system clock has apparently gone back in time; reset our
207     fLastSyncTime  = timeNow;
208     return;
209   }
210   DelayInterval timeSinceLastSync = timeNow - fLastSyncTime;
211   fLastSyncTime = timeNow;
212
213   // Then, adjust the delay queue for any entries whose time is up:
214   DelayQueueEntry* curEntry = head();
215   while (timeSinceLastSync >= curEntry->fDeltaTimeRemaining) {
216     timeSinceLastSync -= curEntry->fDeltaTimeRemaining;
217     curEntry->fDeltaTimeRemaining = DELAY_ZERO;
218     curEntry = curEntry->fNext;
219   }
220
221   curEntry->fDeltaTimeRemaining -= timeSinceLastSync;
222 }
```

首先分析DelayQueue::sychronize()函数

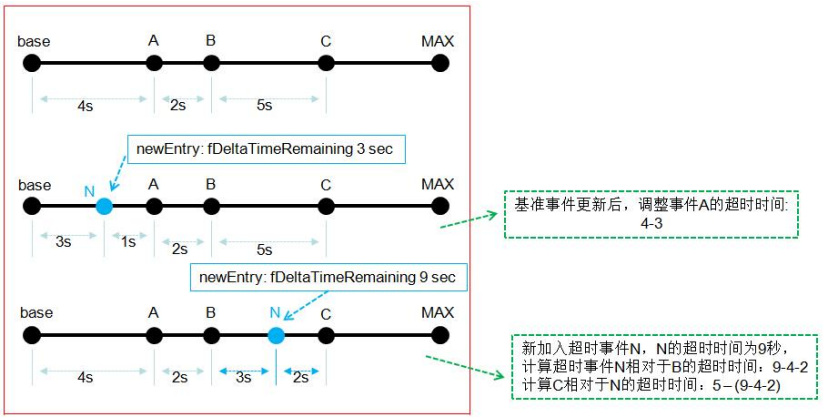
void synchronize(); // bring the 'time remaining' fields up-to-date

首先更新基准事件，计算相对于上次基准事件已经消费掉的时间timeSinceLastSync；遍历超时事件列表，若列表中超时事件的超时时间小于等于timeSinceLastSync，则该事件已经超时，将该事件的超时时间设置为0。否则将超时时间减去timeSinceLastSync。




```
127 void DelayQueue::addEntry(DelayQueueEntry* newEntry) {
128   synchronize();
129
130   DelayQueueEntry* cur = head();
131   while (newEntry->fDeltaTimeRemaining >= cur->fDeltaTimeRemaining) {
132     newEntry->fDeltaTimeRemaining -= cur->fDeltaTimeRemaining;
133     cur = cur->fNext;
134   }
135
136   cur->fDeltaTimeRemaining -= newEntry->fDeltaTimeRemaining;
137
138   // Add "newEntry" to the queue, just before "cur":
139   newEntry->fNext = cur;
140   newEntry->fPrev = cur->fPrev;
141   cur->fPrev = newEntry->fPrev->fNext = newEntry;
142 }
```

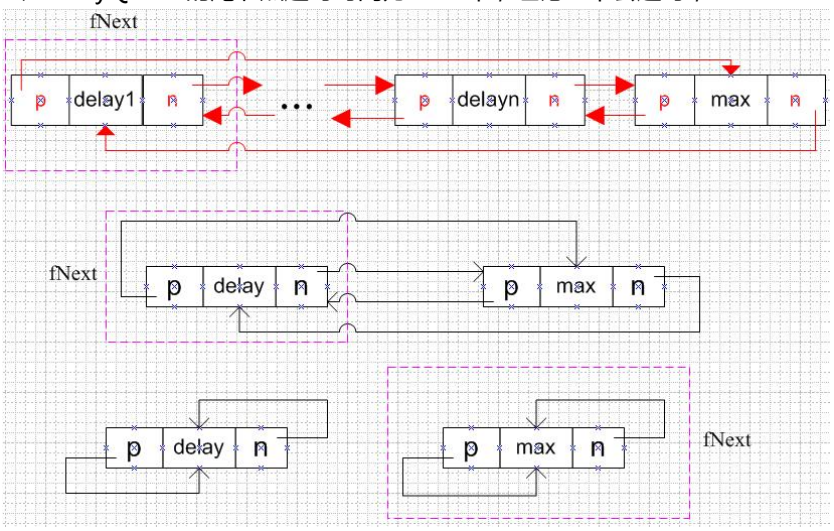
将新超时事件加入队列，根据超时时间排序，将最先会超时的事件放在队列的头部；计算列表中前后超时事件的相对时间差，所有的超时事件的超时时间都是相对于前一个事件的超时时间，第一个超时事件的超时时间是相对于基准事件计算的。

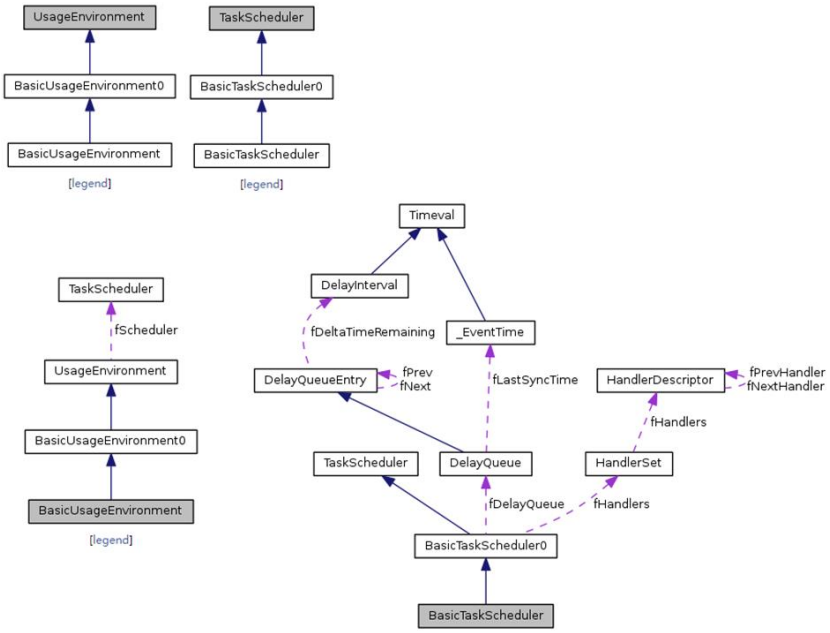


DelayQueue的双向链表结构如下图：

a、DelayQueue的fNext总是指向链表头，指向最先超时的定时事件；

b、DelayQueue的尾节点超时时间为1635年，理论上不会超时；





补充知识：

memcpy与memmove的目的都是将N个字节的源内存地址的内容拷贝到目标内存地址中。但当源内存和目标内存，存在重叠时，memcpy会出现错误，而memmove能正确地实施拷贝，但这也增加了一点点开销。

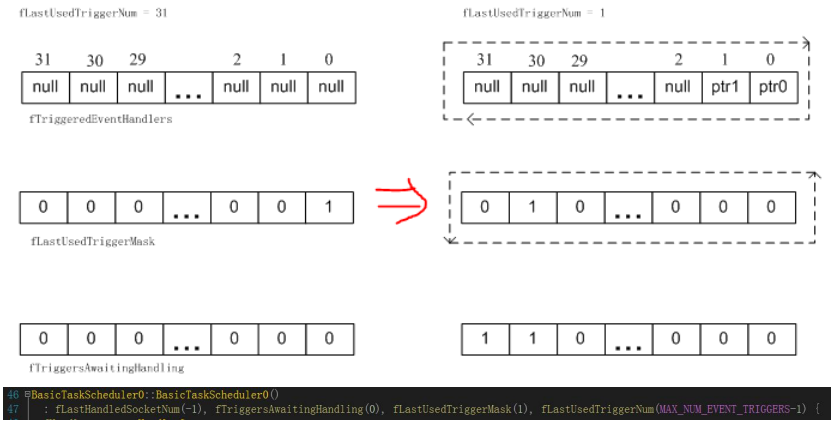
memmove的处理措施：

（1）当源内存的首地址等于目标内存的首地址时，不进行任何拷贝；
（2）当源内存的首地址大于目标内存的首地址时，实行正向拷贝；
（3）当源内存的首地址小于目标内存的首地址时，实行反向拷贝，反向拷贝避免覆盖；

补充知识：
（1）基类构造函数声明我protected：
构造函数从基类开始执行，先执行基类构造函数，再执行子类构造函数；
构造函数声明为protected，就不能直接生成基类对象了，在子类中存在静态函数，用来生成子类对象，单例大多是这种情况；
（2）基类的析构函数声明为public virtual属性：
析构函数声明为virtual时，析构时从子类开始的，先调用子类析构函数，在调用基类的析构函数；不带virtual属性时，直接调用基类的析构函数，将导致内存泄漏；
---
live555共有三种事件：
a、I/O复用事件；　　--HandlerSet　　-->重复性事件
b、触发器事件；　　--EventTriggerId　　-->一次性事件
c、延迟任务事件；　　--DelayQueue　　-->一次性事件

BasicUsageEnvironment类表示用户的使用环境，主要就三种功能：
1、获取/设置错误消息字符串；
2、重载操作符；
3、持有的TaskScheduler，用来调度live555的三种事件；

触发器事件：
触发器事件，是外部控制的一种触发事件，先创建触发事件，等到外部调用触发事件时，才会将该事件触发事件集合中等待触发；触发集合中的事件，按创建的顺序触发，主要通过32位长的bitmap技术实现。




```
BasicTaskScheduler0::BasicTaskScheduler0()
    : fLastHandledSocketNum(-1), fTriggersAwaitingHandling(0), fLastUsedTriggerMask(1), fLastUsedTriggerNum(MAX_NUM_EVENT_TRIGGERS-1) {
```

```
EventTriggerId BasicTaskScheduler0::createEventTrigger(TaskFunc* eventHandlerProc) {
  unsigned i = fLastUsedTriggerNum;
  EventTriggerId mask = fLastUsedTriggerMask;

  //遍历32个槽位，找到可用槽位，返回mask
  do {
    i = (i+1)%MAX_NUM_EVENT_TRIGGERS;
    mask >>= 1;
    if (mask == 0) mask = 0x80000000;

    if (fTriggeredEventHandlers[i] == NULL) {
      // This trigger number is free; use it:
      fTriggeredEventHandlers[i] = eventHandlerProc;
      fTriggeredEventClientDatas[i] = NULL; // sanity

      fLastUsedTriggerMask = mask;
      fLastUsedTriggerNum = i;

      return mask;
    }
  } while (i != fLastUsedTriggerNum);

  // All available event triggers are allocated; return 0 instead:
  return 0;
}
void BasicTaskScheduler0::triggerEvent(EventTriggerId eventTriggerId, void* clientData) {
  // First, record the "clientData". (Note that we allow "eventTriggerId" to be a combination of bits for multiple events.)
  EventTriggerId mask = 0x80000000;
  for (unsigned i = 0; i < MAX_NUM_EVENT_TRIGGERS; ++i) {
    if ((eventTriggerId&mask) != 0) {
      fTriggeredEventClientDatas[i] = clientData;
    }
    mask >>= 1; //这里说允许eventTriggerId是多个Id的或组合，所以在满足if条件时，没有break
  }

  // Then, note this event as being ready to be handled.
  // (Note that because this function (unlike others in the library) can be called from an external thread, we do this last, to
  //  reduce the risk of a race condition.)
  fTriggersAwaitingHandling |= eventTriggerId;
}
```

fTriggeredEventHandlers：存储触发事件回调函数的数组，长度为32；

fLastUsedTriggerNum：数组的索引，记录上一次使用的位置；从0-->31，循环使用；

fLastUsedTriggerMask：32长度的bitmap，始终只有一位为1，表示正在使用；从高位开始，31->0位逐步设置为1，；

fTriggersAwaitingHandling：等待触发的事件mask集合；按创建的顺序触发；

//创建触发事件，返回值为触发事件的mask

virtual EventTriggerId createEventTrigger(TaskFunc* eventHandlerProc);

//删除触发事件

virtual void deleteEventTrigger(EventTriggerId eventTriggerId);

//触发事件，将触发事件mask，加入待触发事件集合

virtual void triggerEvent(EventTriggerId eventTriggerId, void* clientData = NULL);

从当前索引和mask的下一位开始查找触发集合中的事件，保证了，总是优先触发集合中最先创建的事件。

```
    unsigned i = fLastUsedTriggerNum;
    EventTriggerId mask = fLastUsedTriggerMask;

    do {
        i = (i + 1) % MAX_NUM_EVENT_TRIGGERS;
        mask >>= 1;
        if (mask == 0) mask = 0x80000000;

        if ((fTriggersAwaitingHandling&mask) != 0) {
            fTriggersAwaitingHandling &= ~mask;
            if (fTriggeredEventHandlers[i] != NULL) {
                (*fTriggeredEventHandlers[i])(fTriggeredEventClientDatas[i]);
            }

            fLastUsedTriggerMask = mask;
            fLastUsedTriggerNum = i;
            break;
        }
    } while (i != fLastUsedTriggerNum);
```

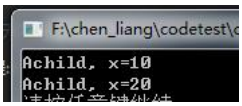
补充知识：

1、决不要重新定义继承而来的缺省参数值；

2、虚函数是动态绑定而缺省参数值是静态绑定的，也就是指针是哪种类型，就调用该类型对应的类中，该函数定义时的缺省值。

```
class Base
{
public:
	Base() {};
	virtual ~Base() {};
public:
	virtual void print(int x=10) {
		printf("base, x=%d\n",x);
	};
};

class Achlid : public Base
{
public:
	Achlid() {};
	virtual ~Achlid() {};
public:
	virtual void print(int x) {
		printf("Achild, x=%d\n", x);
	};
};

int main()
{
	Base* b = new Achlid;
	b->print();

	int y = 20;
	Achlid* c = new Achlid;
	c->print(y);

	return 0;
}
```

执行结果：



volatile关键字：

一个定义为volatile的变量是说这变量可能会被意想不到地改变，这样，编译器就不会去假设这个变量的值了。精确地说就是，优化器在用到这个变量时必须每次都小心地重新读取这个变量的值，而不是使用保存在寄存器里的备份。下面是volatile变量的几个例子：

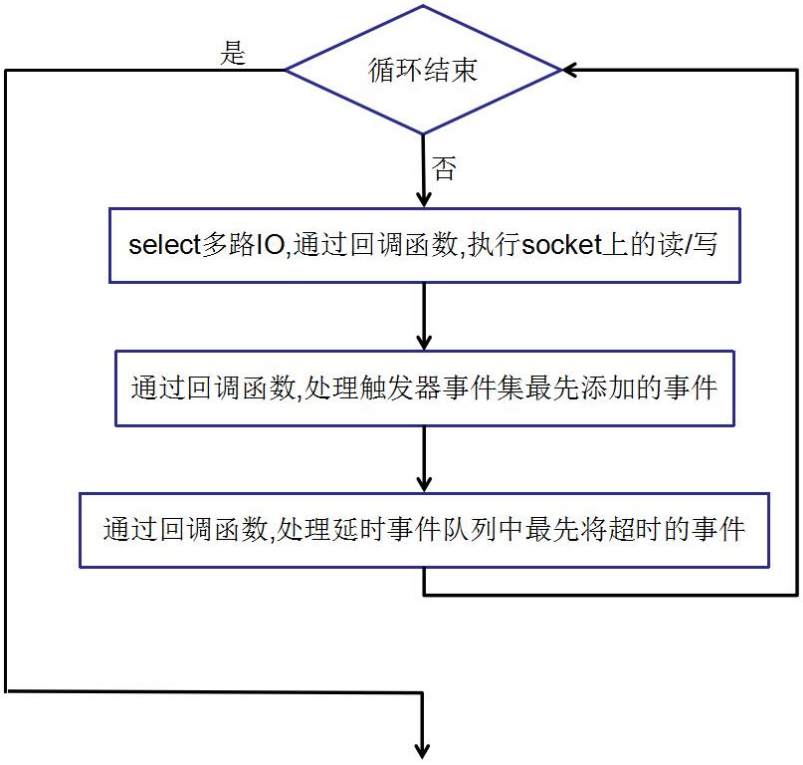
1). 并行设备的硬件寄存器（如：状态寄存器）

2). 一个中断服务子程序中会访问到的非自动变量(Non-automatic variables)

3). 多线程应用中被几个任务共享的变量

事件调度:

http://blog.chinaunix.net/uid-701988-id-3280275.html



网络IO事件：HandleSet，turnOnBackgroundReadHandling或者setBackgroundHandling将socket加入HandleSet集合，同时设置了对应的回调函数，select多路IO时u，判断socket是否可读写，可读写时，通过回调函数执行对应功能；

触发器事件：bitmap，triggerEvent触发事件；

延时事件：DelayQueue，scheduleDelayedTask将事件加入延迟队列，同时设置回调函数。超时事件通过回调函数执行相关操作；

```
void BasicTaskScheduler0::doEventLoop(char volatile* watchVariable) {
  // Repeatedly loop, handling readble sockets and timed events:
  while (1) {
    if (watchVariable != NULL && *watchVariable != 0) break;
    SingleStep();
  }
}
```

1、doEventLoop()中执行循环，在SingleStep()中执行一次具体操作，总结为以下四步：

1)、为所有需要操作的socket执行select；

2)、找出第一个应执行的socket任务(handler)并执行之；

3)、找到第一个应响应的触发事件，并执行之；

4)、找到第一个应执行的延迟任务并执行之；

可见，每一步中只执行三个任务队列中的一项。

```
void BasicTaskScheduler::SingleStep(unsigned maxDelayTime) {  // maxDelayTime default value is 10ms
  fd_set readSet = fReadSet; // make a copy for this select() call
  fd_set writeSet = fWriteSet; // ditto
  fd_set exceptionSet = fExceptionSet; // ditto

  //保证select函数每次最多睡眠10ms，精度10ms
  DelayInterval const& timeToDelay = fDelayQueue.timeToNextAlarm();
  struct timeval tv_timeToDelay;
  tv_timeToDelay.tv_sec = timeToDelay.seconds();
  tv_timeToDelay.tv_usec = timeToDelay.useconds();
  // Very large "tv_sec" values cause select() to fail.
  // Don't make it any larger than 1 million seconds (11.5 days)
  const long MAX_TV_SEC = MILLION;
  if (tv_timeToDelay.tv_sec > MAX_TV_SEC) {
    tv_timeToDelay.tv_sec = MAX_TV_SEC;
  }
  // Also check our "maxDelayTime" parameter (if it's > 0):
  if (maxDelayTime > 0 &&
      (tv_timeToDelay.tv_sec > (long)maxDelayTime/MILLION ||
       (tv_timeToDelay.tv_sec == (long)maxDelayTime/MILLION &&
        tv_timeToDelay.tv_usec > (long)maxDelayTime%MILLION))) {
    tv_timeToDelay.tv_sec = maxDelayTime/MILLION;
    tv_timeToDelay.tv_usec = maxDelayTime%MILLION;
  }
```

参数maxDelayTime默认值为10ms；

```
//获取延迟事件队列中，最先将超时的事件的剩余时间
DelayInterval const& timeToDelay = fDelayQueue.timeToNextAlarm();

//保证select的等待时间不超过10ms，精度10ms
if (maxDelayTime > 0 &&
   (tv_timeToDelay.tv_sec > (long)maxDelayTime/MILLION ||
    (tv_timeToDelay.tv_sec == (long)maxDelayTime/MILLION &&
     tv_timeToDelay.tv_usec > (long)maxDelayTime%MILLION))) {
            tv_timeToDelay.tv_sec = maxDelayTime/MILLION;
             tv_timeToDelay.tv_usec = maxDelayTime%MILLION;
}

//对三组描述符进行io复用
select(fMaxNumSockets, &readSet, &writeSet, &exceptionSet, &tv_timeToDelay);
```

```
BasicTaskScheduler0::BasicTaskScheduler0()
    : fLastHandledSocketNum(-1), fTriggersAwaitingHandling(0), f
    fHandlers = new HandlerSet;
//如果返回0代表在所有描述符状态改变前已超过timeout时间;

//执行成功则返回文件描述词状态已改变的个数；三组fd_set均将某些fd位置0，
//只有那些可读，可写以及有异常条件待处理的fd位仍然为1;

// IO事件
// Call the handler function for one readable socket:
HandlerIterator iter(*fHandlers); //fHandlers为双向链表，新节点在表头
HandlerDescriptor* handler;
// To ensure forward progress through the handlers, begin past the last
// socket number that we handled:
// IO事件存储在双向链表中，新事件总是存储在表头;
// 由于在SingleStep中，IO事件只执行一个，为了保证所有的IO事件都能执行:
// 从上次IO事件的下一个事件开始遍历
if (fLastHandledSocketNum >= 0) {
  while ((handler = iter.next()) != NULL) {
    if (handler->socketNum == fLastHandledSocketNum) break;
  }
  if (handler == NULL) {
    fLastHandledSocketNum = -1;
    iter.reset(); // start from the beginning instead
  }
}
// 遍历IO事件链表，找到第一个可读|可写|可执行的描述符
// 执行IO事件的回调函数，记录该事件的描述符
while ((handler = iter.next()) != NULL) {
  int sock = handler->socketNum; // alias
  int resultConditionSet = 0;
  if (FD_ISSET(sock, &readSet) && FD_ISSET(sock, &fReadSet)/*sanity check*/)
      resultConditionSet |= SOCKET_READABLE;
  if (FD_ISSET(sock, &writeSet) && FD_ISSET(sock, &fWriteSet)/*sanity check*/)
      resultConditionSet |= SOCKET_WRITABLE;
  if (FD_ISSET(sock, &exceptionSet) && FD_ISSET(sock, &fExceptionSet)/*sanity check*/)
      resultConditionSet |= SOCKET_EXCEPTION;
  if ((resultConditionSet&handler->conditionSet) != 0 && handler->handlerProc != NULL) {
      fLastHandledSocketNum = sock;
        // Note: we set "fLastHandledSocketNum" before calling the handler,
        // in case the handler calls "doEventLoop()" reentrantly.
      (*handler->handlerProc)(handler->clientData, resultConditionSet);
      break;
  }
}
```

```
// 未找到描述符变化的IO事件
// 由于fLastHandledSocketNum大于等于零，上文查找不是从链表头开始的
// 需要重新从链表头开始重新查找
if (handler == NULL && fLastHandledSocketNum >= 0) {
  // We didn't call a handler, but we didn't get to check all of them,
  // so try again from the beginning:
  iter.reset();
  while ((handler = iter.next()) != NULL) {
    int sock = handler->socketNum; // alias
    int resultConditionSet = 0;
    if (FD_ISSET(sock, &readSet) && FD_ISSET(sock, &fReadSet)/*sanity check*/)
        resultConditionSet |= SOCKET_READABLE;
    if (FD_ISSET(sock, &writeSet) && FD_ISSET(sock, &fWriteSet)/*sanity check*/)
        resultConditionSet |= SOCKET_WRITABLE;
    if (FD_ISSET(sock, &exceptionSet) && FD_ISSET(sock, &fExceptionSet)/*sanity check*/)
        resultConditionSet |= SOCKET_EXCEPTION;
    if ((resultConditionSet&handler->conditionSet) != 0 && handler->handlerProc != NULL) {
          fLastHandledSocketNum = sock;
          // Note: we set "fLastHandledSocketNum" before calling the handler,
             // in case the handler calls "doEventLoop()" reentrantly.
          (*handler->handlerProc)(handler->clientData, resultConditionSet);
          break;
    }
  }
  if (handler == NULL) fLastHandledSocketNum = -1;//because we didn't call a handler
}
```

IO事件是永久性事件，会重复执行；

第一次执行的时候，从IO事件链表头开始遍历链表，找到第一个文件描述符变化的IO事件，执行IO事件的回调函数，记录该事件的socket；

由于在SingerStep()中，只执行链表中第一个描述符变化的IO事件，新IO事件总是在链表头；为了所有的IO事件都能够得到执行，不能每次都从链表头开始遍历IO事件；所以从上次执行的IO事件的下一个位置开始遍历。若未找到描述符变化的事件，需要从链表头重新开始查找。

```
// 触发器事件
// Also handle any newly-triggered event (Note that we do this *after* calling a socket handler,
// in case the triggered event handler modifies The set of readable sockets.)
if (fTriggersAwaitingHandling != 0) {
  if (fTriggersAwaitingHandling == fLastUsedTriggerMask) {    // 触发器事件只有一个
    // Common-case optimization for a single event trigger:
    fTriggersAwaitingHandling &=~ fLastUsedTriggerMask;       // 触发事件只执行一次
    if (fTriggeredEventHandlers[fLastUsedTriggerNum] != NULL) {
      (*fTriggeredEventHandlers[fLastUsedTriggerNum])(fTriggeredEventClientDatas[fLastUsedTriggerNum]);
    }
  } else {
} else {
  // Look for an event trigger that needs handling
    // (making sure that we make forward progress through all possible triggers):
  unsigned i = fLastUsedTriggerNum;              //上一个创建的触发器事件的索引
  EventTriggerId mask = fLastUsedTriggerMask;    //上一个创建的触发器事件的mask

  // 从当前索引和mask的下一位开始查找触发集合中的事件，
  // 保证了，总是优先触发集合中最先创建的事件
  do {
      i = (i + 1) % MAX_NUM_EVENT_TRIGGERS;
      mask >>= 1;
      if (mask == 0) mask = 0x80000000;

      if ((fTriggersAwaitingHandling&mask) != 0) {
          fTriggersAwaitingHandling &= ~mask;   // 触发事件只执行一次
          if (fTriggeredEventHandlers[i] != NULL) {
              (*fTriggeredEventHandlers[i])(fTriggeredEventClientDatas[i]);
          }

          fLastUsedTriggerMask = mask;
          fLastUsedTriggerNum = i;
          break;    // 每次只执行一个触发器事件
      }
  } while (i != fLastUsedTriggerNum);
}
```

触发器事件，为一次性事件，只执行一次；

创建的触发器，再有加入触发器集合中才会被触发；

触发器集合中的事件，按照创建的顺序进行触发，即优先触发最先创建的事件

```
// 延迟事件
// 每次只执行一个超时事件
// Also handle any delayed event that may have come due.
fDelayQueue.handleAlarm();
```

延迟事件：为一次性事件，只执行一次；

若延时事件队列中存在多个超时事件，一次只触发一个事件；
触发最先超时的事件，即触发表头事件；

////////////

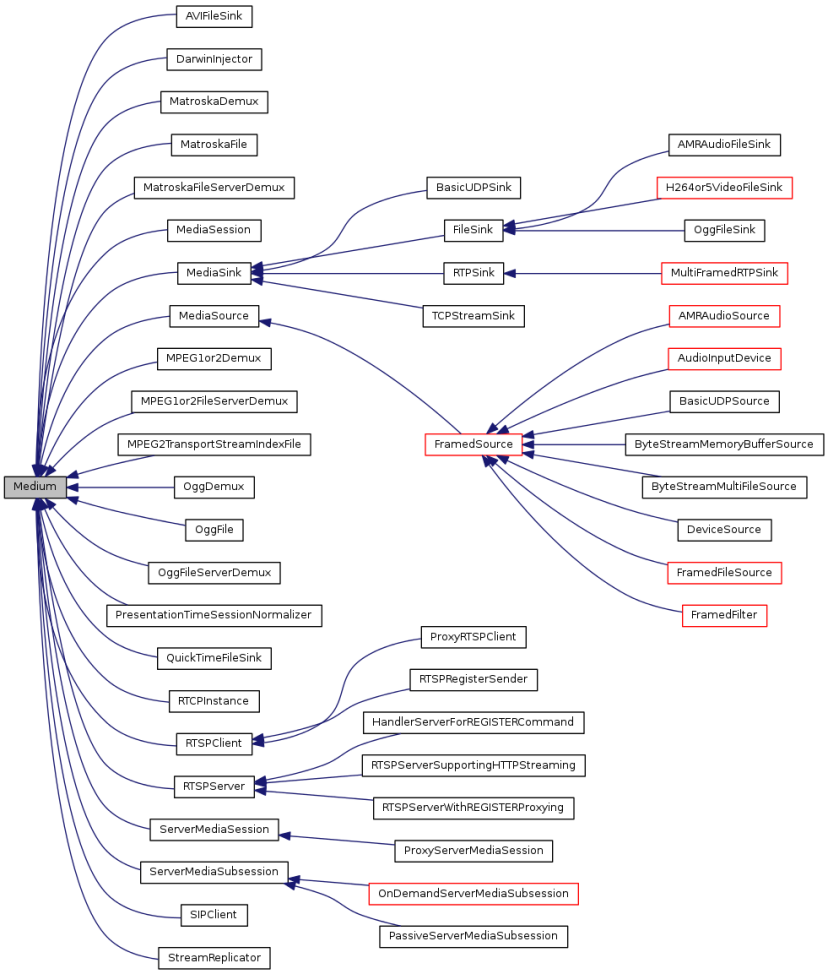
NetAddress.hh：包含一些网络地址处理类；
1、NetAddress：存储ipv4地址；
2、NetAddressList：存储ipv4地址列表，内部通过NetAddress**实现地址列表；
3、Port：存储端口数据；
4、AddressPortLookupTable：地址端口查找，通过HashTable实现；
5、AddressString：用于以点分十进制的C风格字符串形式保存的IP地址；

NetInterface.hh：网络接口，多为基类
NetInterface：基类；
DirectedNetInterface：基类；
DirectedNetInterfaceSet：DirectedNetInterface序列，通过HashTable实现；
Socket：udp套接字管理类；
SocketLookupTable：Socket列表，通过HashTable实现；
NetInterfaceTrafficStats：统计数据包的数量和流量；

IP多播(也称多址广播或组播)技术，是一种允许一台或多台主机（多播源）发送单一数据包到多台主机（一次的，同时的）的TCP/IP网络技术。多播作为一点对多点的通信，是节省网络带宽的有效方法之一。在网络音频/视频广播的应用中，当需要将一个节点的信号传送到多个节点时，无论是采用重复点对点通信方式，还是采用广播方式，都会严重浪费网络带宽，只有多播才是最好的选择。多播能使一个或多个多播源只把数据包发送给特定的多播组，而只有加入该多播组的主机才能接收到数据包。
IP多播通信必须依赖于IP多播地址，在IPv4中它是一个D类IP地址，范围从224.0.0.0到239.255.255.255，并被划分为局部链接多播地址、预留多播地址和管理权限多播地址三类。IP多播属于IP层协议。
1、局部链接多播地址范围在224.0.0.0~224.0.0.255，这是为路由协议和其它用途保留的地址，路由器并不转发属于此范围的IP包；
2、预留多播地址为224.0.1.0~238.255.255.255，可用于全球范围（如Internet）或网络协议；
3、管理权限多播地址为239.0.0.0~239.255.255.255，可供组织内部使用，类似于私有IP地址，不能用于Internet，可限制多播范围。

GroupsockHelper.hh 和 GroupsockHelper.cpp：套接字创建，属性修改，组播等功能；
Groupsock.hh 和 Groupsock.cpp：主要是用于UDP组播；

////////////



---

const修饰函数：

```
class Medium{
public:
...
  virtual Boolean isSource() const;
  virtual Boolean isSink() const;
...
}
```

1、const修饰的函数，不能修改对象的数据成员；

2、const修饰的函数，不能调用非const函数。因为非const函数，可能会修改数据成员，最终导致该const函数修改了数据成员；

3、const指针可以接受const和非const指针，而非const指针只能接受非const指针；

4、const对象只能调用const函数，而非const对象却可以访问const和非const函数；

delete this：

1、this对象是必须是用 new操作符分配的（而不是用new[]，也不是用placement new，也不是局部对象，也不是global对象）；

2、delete this后，不能访问该对象任何的成员变量及虚函数（delete this回收的是数据，这包括对象的数据成员以及vtable，不包括函数代码）；

3、delete this后，不能再访问this指针。换句话说，你不能去检查它、将它和其他指针比较、和 NULL比较、打印它、转换它，以及其它的任何事情；

c/c++函数申明和定义：

在函数声明中：

无论是C还是在C++，都可以省略形式参数名；但是，通常都不建议省略形式参数名；

在函数定义中：

1. 当需要使用形式参数的时候，显然，必须给形式参数命名；
2. 当不需要使用形式参数的时候，C与C++有微小差异：

—— C不能省略形式参数名， 即使不使用。

—— C++可以省略形式参数名，如果不使用。

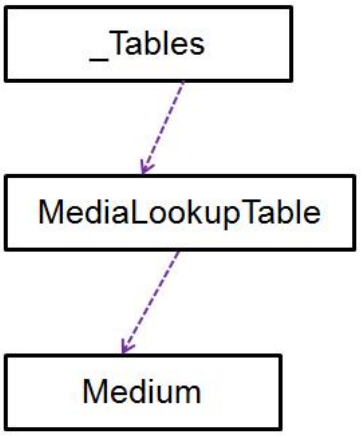
—— 并且在C++中，如果给不使用的形式参数命名，可能会得到一个警告。

Media.hh && Media.cpp：

Medium：媒体处理基类；

MediaLookupTable：通过Hash表，维护Medium和Medum对应的name，name为键值；

_Tables：指向UsageEnvironment的liveMediaPriv字段；维护一个MediaLookupTable；



GenericMediaServer.hh && GenericMediaServer.cpp：

UserAuthenticationDatabase：鉴权管理类，用于保存用户名和密码；

GenericMediaServer：通用媒体服务器类，服务基类；

//启动监听，默认非阻塞，50k发送缓冲区，关闭了SO_REUSEADDR

static int setUpOurSocket(UsageEnvironment& env, Port& ourPort);

//构造函数，将监听套接字fServerSocket加入了网络IO事件集，当套接字可读时，触发回调，回调函数为incomingConnectionHandler()；reclamationSeconds为rtspClient的保活时间，默认为65秒，超过该时间没有收到rtsp的保活消息，就删除该客户端连接；rtsp使用GET_PARAMENT或者OPTION保活；

GenericMediaServer(UsageEnvironment& env, int ourSocket, Port ourPort,
unsigned reclamationSeconds);

```
GenericMediaServer
::GenericMediaServer(UsageEnvironment& env, int ourSocket, Port ourPort,
            unsigned reclamationSeconds)
  : Medium(env),
    fServerSocket(ourSocket), fServerPort(ourPort), fReclamationSeconds(reclamationSeconds),
    fServerMediaSessions(HashTable::create(STRING_HASH_KEYS)),
    fClientConnections(HashTable::create(ONE_WORD_HASH_KEYS)),
    fClientSessions(HashTable::create(STRING_HASH_KEYS)) {
  ignoreSigPipeOnSocket(fServerSocket); // so that clients on the same host that are killed don't al

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

//析构函数，析构时，将监听套接字从网络IO事件集中移除，同时关闭监听套接字；

virtual ~GenericMediaServer()；

```
GenericMediaServer::~GenericMediaServer() {
  // Turn off background read handling:
  envir().taskScheduler().turnOffBackgroundReadHandling(fServerSocket);
  ::closeSocket(fServerSocket);
}
```

//监听套接字可读时的回调函数，用于处理接收到的网络连接请求；

static void incomingConnectionHandler(void*, int /*mask*/);

//实际处理网络连接请求，套接字clientSocket和远端地址信息clientAddr交由子类继承实现的函数createNewClientConnection，由该函数执行具体操作；

void incomingConnectionHandlerOnSocket(int serverSocket);

```
//实际执行网络请求处理的函数
void GenericMediaServer::incomingConnectionHandlerOnSocket(int serverSocket) {
  struct sockaddr_in clientAddr;
  SOCKLEN_T clientAddrLen = sizeof clientAddr;

  //clientAddr存储远端客户的ip和端口
  int clientSocket = accept(serverSocket, (struct sockaddr*)&clientAddr, &clientAddrLen);
  if (clientSocket < 0) {
    int err = envir().getErrno();
    if (err != EWOULDBLOCK) {
      envir().setResultErrMsg("accept() failed: ");
    }
    return;
  }
  ignoreSigPipeOnSocket(clientSocket); // so that clients on the same host that are kille
  makeSocketNonBlocking(clientSocket); // 非阻塞
  increaseSendBufferTo(envir(), clientSocket, 50*1024); // 50k发送缓冲区

#ifdef DEBUG
  envir() << "accept()ed connection from " << AddressString(clientAddr).val() << "\n";
#endif

  // Create a new object for handling this connection:
  (void)createNewClientConnection(clientSocket, clientAddr);
}
```

ClientConnection：处理客户端的网络连接请求，只负责数据接收，解析由子类实现；

//构造函数，每个连接准备20k的接受缓冲区，然后将该与客户端的双向套接字clientSocket加入网络IO事件集中，注册回调函数incomingRequestHandler，套接字可读|异常时，触发回调；

ClientConnection(GenericMediaServer& ourServer, int clientSocket,
struct sockaddr_in clientAddr);

```
GenericMediaServer::ClientConnection
::ClientConnection(GenericMediaServer& ourServer, int clientSocket, struct sockaddr_in clientAddr)
  : fOurServer(ourServer), fOurSocket(clientSocket), fClientAddr(clientAddr) {
  // Add ourself to our 'client connections' table:
  fOurServer.fClientConnections->Add((char const*)this, this);

  // Arrange to handle incoming requests:
  resetRequestBuffer();
  envir().taskScheduler()
    .setBackgroundHandling(fOurSocket, SOCKET_READABLE|SOCKET_EXCEPTION, incomingRequestHandler, this);
}
```

//析构函数，从网络IO事件集中删除套接字，取消注册回调，关闭套接字；

virtual ~ClientConnection();

//网络可读|异常回调函数，只负责读取网络数据，解析由子类实现；

static void incomingRequestHandler(void*, int /*mask*/);

```
void GenericMediaServer::ClientConnection::incomingRequestHandler(void* instance, int /*mask*/) {
  ClientConnection* connection = (ClientConnection*)instance;
  connection->incomingRequestHandler();
}

void GenericMediaServer::ClientConnection::incomingRequestHandler() {
  struct sockaddr_in dummy; // 'from' address, meaningless in this case

  int bytesRead = readSocket(envir(), fOurSocket, &fRequestBuffer[fRequestBytesAlreadySeen], fReque
  handleRequestBytes(bytesRead);
}
```

ClientSession：代表一个客户端会话

//构造函数，启动定时任务，进行客户端保活

ClientSession(GenericMediaServer& ourServer, u_int32_t sessionId);

```
GenericMediaServer::ClientSession
::ClientSession(GenericMediaServer& ourServer, u_int32_t sessionId)
  : fOurServer(ourServer), fOurSessionId(sessionId), fOurServerMediaSession(NULL),
    fLivenessCheckTask(NULL) {
  noteLiveness();
}
```

//析构函数，删除定时保活任务

virtual ~ClientSession();

//启动定时保活任务,65秒

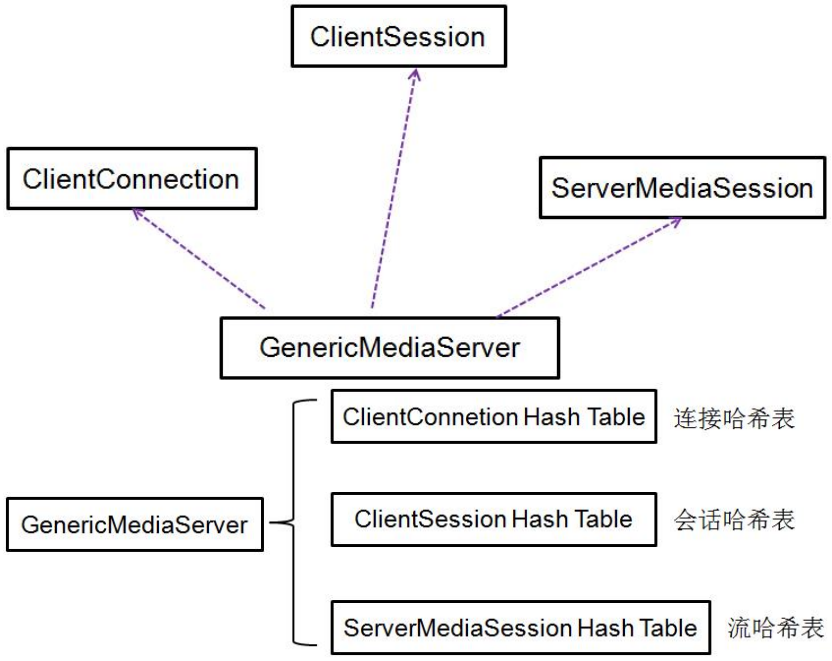
void noteLiveness();

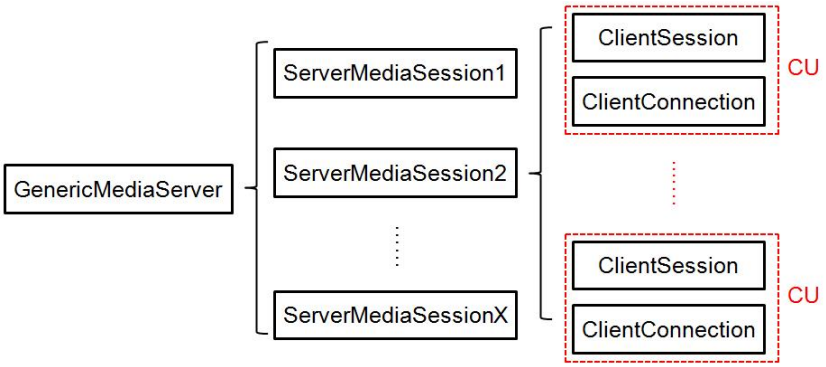
```
void GenericMediaServer::ClientSession::noteLiveness() {
#ifdef DEBUG
  char const* streamName
    = (fOurServerMediaSession == NULL) ? "???" : fOurServerMediaSession->streamName();
  fprintf(stderr, "Client session (id \"%08X\", stream name \"%s\"): Liveness indication\n",
      fOurSessionId, streamName);
#endif
  if (fOurServerMediaSession != NULL) fOurServerMediaSession->noteLiveness();

  if (fOurServer.fReclamationSeconds > 0) {
    envir().taskScheduler().rescheduleDelayedTask(fLivenessCheckTask,
                          fOurServer.fReclamationSeconds*1000000,
                          (TaskFunc*)livenessTimeoutTask, this);
  }
}
```

//定时保活事件的回调函数，触发时，说明客户端超时，删除该会话

static void livenessTimeoutTask(ClientSession* clientSession);

```
void GenericMediaServer::ClientSession::livenessTimeoutTask(ClientSession* clientSession) {
  // If this gets called, the client session is assumed to have timed out, so delete it:
#ifdef DEBUG
  char const* streamName
    = (clientSession->fOurServerMediaSession == NULL) ? "???" : clientSession->fOurServerMediaSession->streamName();
  fprintf(stderr, "Client session (id \"%08X\", stream name \"%s\") has timed out (due to inactivity)\n",
      clientSession->fOurSessionId, streamName);
#endif
  delete clientSession;
}
```





通用媒体服务类GenericMediaServer维护三个哈希表：

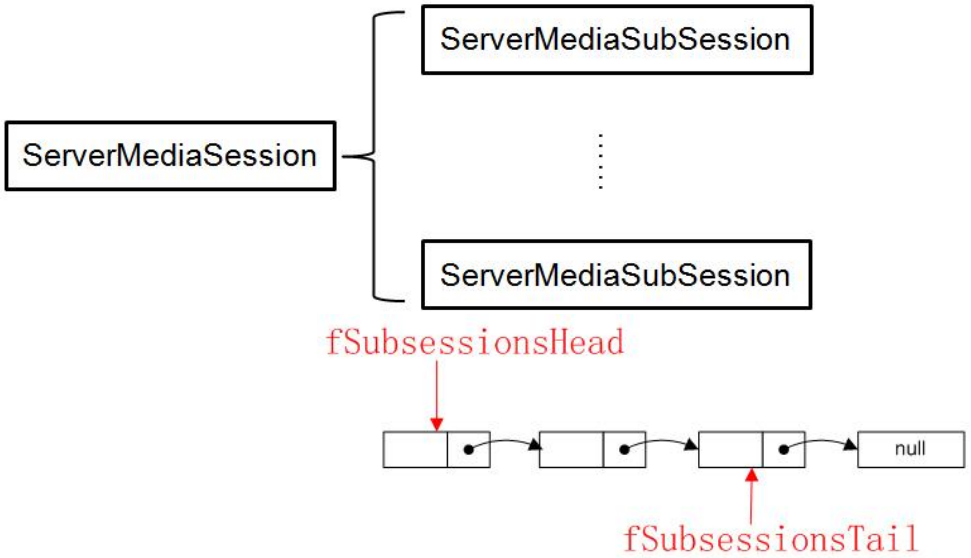
ClientConnection客户端连接哈希表；

ClientSession客户端会话哈希表；

ServerMediaSession流哈希表；

每一个ServerMediaSession都代表一路流，允许客户端连接请求。每个CU的请求都会存在一个ClientSession和一个ClientConnection；

ServerMediaSession.hh && ServerMediaSession.cpp：媒体服务器会话(流)和子会话(流中的音频或者视频)



ServerMediaSession：代表一路流

```
//创建ServerMediaSession，该类为单例，通过该接口获取单例
 static ServerMediaSession* createNew(UsageEnvironment& env,
                                      char const* streamName = NULL,
                                      char const* info = NULL,
                                      char const* description = NULL,
                                      Boolean isSSM = False,
                                      char const* miscSDPLines = NULL);

//向流中添加子会话，一般为音频或者视频
Boolean addSubsession(ServerMediaSubsession* subsession);
```

```
ServerMediaSession::addSubsession(ServerMediaSubsession* subsession) {
  if (subsession->fParentSession != NULL) return False; // it's already used

  if (fSubsessionsTail == NULL) {
    fSubsessionsHead = subsession;
  } else {
    fSubsessionsTail->fNext = subsession;
  }
  fSubsessionsTail = subsession;

  subsession->fParentSession = this;
  subsession->fTrackNumber = ++fSubsessionCounter;  //从0开始累加，区分子会话
  return True;
}
```

//设置播放速度，live555，**默认只支持1倍速**

void testScaleFactor(float& scale); // sets "scale" to the actual supported scale

//流时长，0代表不限时长，一般为实时流；**live555该字段默认为0，不限时长**

// a result == 0 means an unbounded session (the default)

// a result < 0 means: subsession durations differ; the result is -(the largest).

/ a result > 0 means: this is the duration of a bounded session

float duration() const;

//流上的客户端连接超时处理函数，**默认不处理；**流上客户端连接超时时，GenericMediaServer中会删除该客户端会话

// called whenever a client - accessing this media - notes liveness.

// The default implementation does nothing, but subclasses can redefine this - e.g., if you

// want to remove long-unused "ServerMediaSession"s from the server.

virtual void noteLiveness();

```
void ServerMediaSession::noteLiveness() {
  // default implementation: do nothing
}
```

//删除子会话

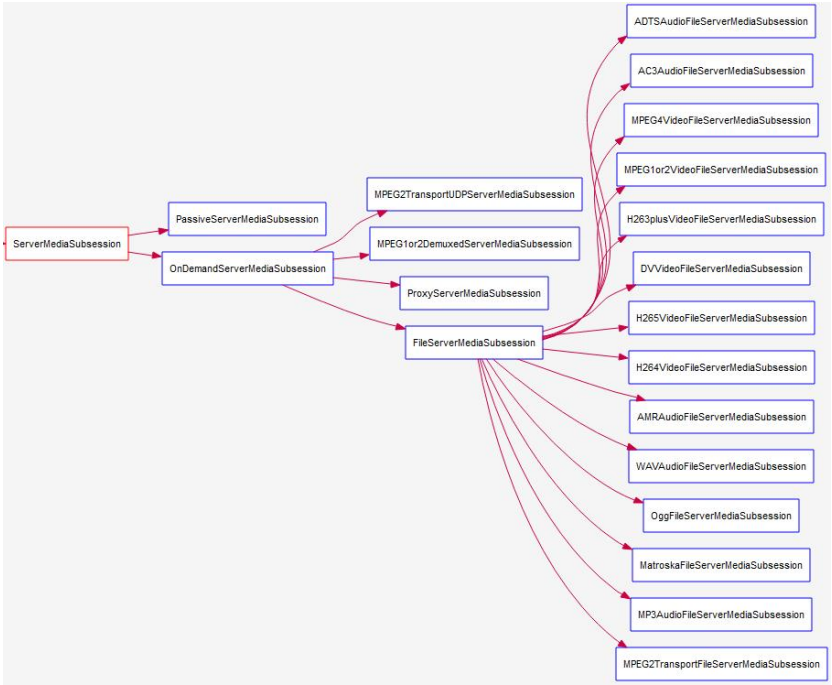
void deleteAllSubsessions();

//用于生成sdp信息，从子会话中获取sdp信息，然后生成流的sdp信息

char* generateSDPDescription();

```
  // Count the lengths of each subsession's media-level SDP lines.
  // (We do this first, because the call to "subsession->sdpLines()"
  // causes correct subsession 'duration()'s to be calculated later.)
  unsigned sdpLength = 0;
  ServerMediaSubsession* subsession;
  for (subsession = fSubsessionsHead; subsession != NULL;
   subsession = subsession->fNext) {
    char const* sdpLines = subsession->sdpLines(); //获取每个子会话的sdp信息
    if (sdpLines == NULL) continue; // the media's not available
    sdpLength += strlen(sdpLines);
  }
```

ServerMediaSubsession：代表一路会话，属于基类，继承关系如下图：



这些操作可以分为几类，一类是对播放进行控制的操作，包括：

startStream()

pauseStream()

seekStream()

nullSeekStream()

setStreamScale()

deleteStream()

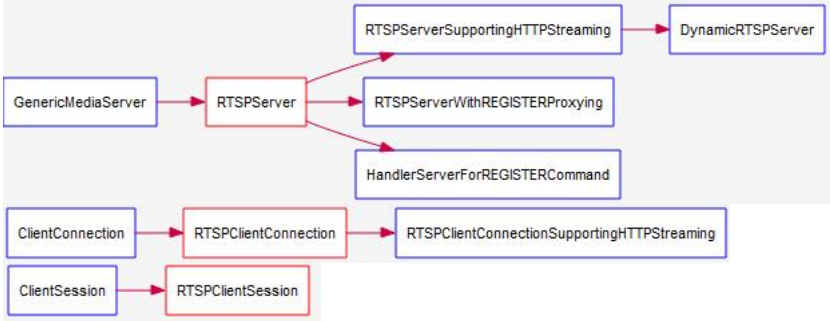
testScaleFactor()

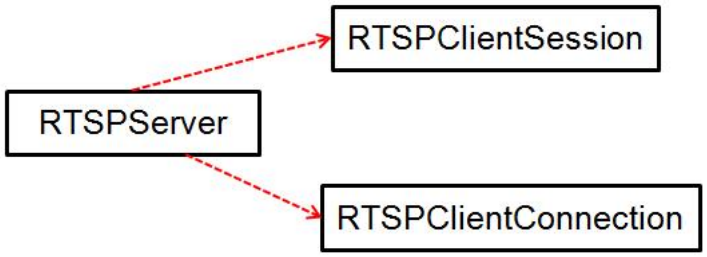
另一类是获得用于执行 I/O 操作：

getStreamSource()

getRTPSinkandRTCP()

RTSPServer.hh && RTSPServer.cpp：包含rtsp服务、连接和会话





RTSPServer：rtsp服务类

//创建rtsp服务类，继承自基类GenericMediaServer，在GenericMediaServer构造函数中，会对套接字ourSocket注册读回调，由loop线程执行io回调

static RTSPServer* createNew(UsageEnvironment& env, Port ourPort = 554,
UserAuthenticationDatabase* authDatabase = NULL,
unsigned reclamationSeconds = 65);

```
RTSPServer*
RTSPServer::createNew(UsageEnvironment& env, Port ourPort,
                      UserAuthenticationDatabase* authDatabase,
                      unsigned reclamationSeconds) {
  int ourSocket = setUpOurSocket(env, ourPort); //创建监听套接字,
  if (ourSocket == -1) return NULL; //默认非阻塞，50k发送缓冲区，关闭了SO_REUSEADDR

  return new RTSPServer(env, ourSocket, ourPort, authDatabase, reclamationSeconds);
}
```

//获取url

char* rtspURL(ServerMediaSession const* serverMediaSession, int clientSocket = -1) const;

char* rtspURLPrefix(int clientSocket = -1) const;

```
// 通过socket获取rtsp-url的前半部分
char* RTSPServer::rtspURLPrefix(int clientSocket) const {
  struct sockaddr_in ourAddress;
  if (clientSocket < 0) {
    // Use our default IP address in the URL:
    ourAddress.sin_addr.s_addr = ReceivingInterfaceAddr != 0
      ? ReceivingInterfaceAddr
      : ourIPAddress(envir()); // hack
  } else {
    SOCKLEN_T namelen = sizeof ourAddress;
    getsockname(clientSocket, (struct sockaddr*)&ourAddress, &namelen);
  }
  // rtsp协议，当cu发起连接时，server收到请求，根据socket能够获取远端ip和本地ip,
  // 多网段时，server根据cu请求的网段，从对应的网段回发响应

  // getsockname函数用于获取与某个套接字关联的本地协议地址
  // getpeername函数用于获取与某个套接字关联的外地协议地址
```

//创建新客户端连接

virtual ClientConnection* createNewClientConnection(int clientSocket, struct sockaddr_in clientAddr);

```
GenericMediaServer::ClientConnection*
RTSPServer::createNewClientConnection(int clientSocket, struct sockaddr_in clientAddr) {
  return new RTSPClientConnection(*this, clientSocket, clientAddr);
}
```

//创建新客户端会话

virtual ClientSession* createNewClientSession(u_int32_t sessionId);

```
GenericMediaServer::ClientSession*
RTSPServer::createNewClientSession(u_int32_t sessionId) {
  return new RTSPClientSession(*this, sessionId);
}
```

RTSPClientConnection：客户端连接类

//构造函数，继承自GenericMediaServer::ClientConnection，clientSocket为与客户端已建立连接的套接字，在GenericMediaServer::ClientConnection构造函数中，会对该套接字注册读回调，交由线程eventLoop统一处理

RTSPClientConnection(RTSPServer& ourServer, int clientSocket, struct sockaddr_in clientAddr);

```
RTSPServer::RTSPClientConnection
::RTSPClientConnection(RTSPServer& ourServer, int clientSocket, struct sockaddr_in clientAddr)
  : GenericMediaServer::ClientConnection(ourServer, clientSocket, clientAddr),
    //ClientConnection构造函数中，对套接字clientSocket注册读回调
    fOurRTSPServer(ourServer), fClientInputSocket(fOurSocket), fClientOutputSocket(fOurSocket),
    fIsActive(True), fRecursionCount(0), fOurSessionCookie(NULL) {
  resetRequestBuffer();
}
```

//处理接受到的客户端数据，GenericMediaServer::ClientConnection::incomingRequestHandler()中进行数据接受，基类只接受数据，不做分析处理；这个函数解析数据，处理OPTION，DESCRIBLE，SETUP，PLAY等rtsp消息，这里是rtsp消息分发的源头。

virtual void handleRequestBytes(int newBytesRead);

通过rtsp消息结束符"\r\n\r\n"，找到完整的rtsp消息，只要接收到完整的消息数据，才会解析RTSP消息。

```
    // rtsp消息结束地址，tmpPtr + 2
    // rtsp消息已\r\n\r\n结尾，十六进制为: 0x0d, 0x0a, 0x0d, 0x0a
    unsigned char* tmpPtr = fLastCRLF + 2;
    if (fBase64RemainderCount == 0) { // no more Base-64 bytes remain to be read/decoded
      // Look for the end of the message: <CR><LF><CR><LF>, 0x0d, 0x0a, 0x0d, 0x0a
      if (tmpPtr < fRequestBuffer) tmpPtr = fRequestBuffer;
        while (tmpPtr < &ptr[newBytesRead-1]) {
            if (*tmpPtr == '\r' && *(tmpPtr+1) == '\n') {
                if (tmpPtr - fLastCRLF == 2) { // This is it:
                    endOfMsg = True;
                    break;
                }
                fLastCRLF = tmpPtr;
            }
            ++tmpPtr;
        }
    }

    fRequestBufferBytesLeft -= newBytesRead;    //可用空间剩余
    fRequestBytesAlreadySeen += newBytesRead;   //已接受字节长度

    // 未接收到完整的rtsp消息，继续读取数据
    if (!endOfMsg) break; // subsequent reads will be needed to complete the request
```

parseRTSPRequestString函数具体完成RTSP消息的解析，解析后通过判断Content-Length字段是否存在以及长度，判断是否接受到完整的RTSP消息。

```
    //解析rtsp消息
    Boolean parseSucceeded = parseRTSPRequestString((char*)fRequestBuffer, fLastCRLF+2 - fRequestBuffer,
                             cmdName, sizeof cmdName,
                             urlPreSuffix, sizeof urlPreSuffix,
                             urlSuffix, sizeof urlSuffix,
                             cseq, sizeof cseq,
                             sessionIdStr, sizeof sessionIdStr,
                             contentLength);
    // 确保收到完整的消息
    // If there was a "Content-Length:" header, then make sure we've received all of the data that it specified:
    if (ptr + newBytesRead < tmpPtr + 2 + contentLength) break; // we still need more data; subsequent reads wil
```

分发rtsp消息

```
      if (strcmp(cmdName, "OPTIONS") == 0) {
        // If the "OPTIONS" command included a "Session:" id for a session that doesn't exist,
        // then treat this as an error:
        // 带Session的option可能是用于RTSP消息保活
        if (requestIncludedSessionId && clientSession == NULL) {
            handleCmd_sessionNotFound();
        } else {
            // Normal case:
            handleCmd_OPTIONS();
        }
      } else if (urlPreSuffix[0] == '\0' && urlSuffix[0] == '*' && urlSuffix[1] == '\0') {
        // The special "*" URL means: an operation on the entire server.  This works only f
        if (strcmp(cmdName, "GET_PARAMETER") == 0) {
            handleCmd_GET_PARAMETER((char const*)fRequestBuffer);
        } else if (strcmp(cmdName, "SET_PARAMETER") == 0) {
            handleCmd_SET_PARAMETER((char const*)fRequestBuffer);
        } else {
            handleCmd_notSupported();
        }
      } else if (strcmp(cmdName, "DESCRIBE") == 0) {
        handleCmd_DESCRIBE(urlPreSuffix, urlSuffix, (char const*)fRequestBuffer);
      } else if (strcmp(cmdName, "SETUP") == 0) {
        Boolean areAuthenticated = True;

    } else if (strcmp(cmdName, "TEARDOWN") == 0
        || strcmp(cmdName, "PLAY") == 0
        || strcmp(cmdName, "PAUSE") == 0
        || strcmp(cmdName, "GET_PARAMETER") == 0
        || strcmp(cmdName, "SET_PARAMETER") == 0) {
      if (clientSession != NULL) {
        clientSession->handleCmd_withinSession(this, cmdName, urlPreSuffix, urlSuffix, (char const*)fRequestBuffer);
      } else {
        handleCmd_sessionNotFound();
      }
```

//option消息处理函数，一般用作rtsp保活消息

virtual void handleCmd_OPTIONS();

OPTION消息：

1、用于保活的OPTION消息，会携带Session字段；

2、OPTION消息，查询服务端支持的消息类型，服务端会将自己支持的消息类型返回；

```
void RTSPServer::RTSPClientConnection::handleCmd_OPTIONS() {
  snprintf((char*)fResponseBuffer, sizeof fResponseBuffer,
       "RTSP/1.0 200 OK\r\nCSeq: %s\r\n%sPublic: %s\r\n\r\n",
       fCurrentCSeq, dateHeader(), fOurRTSPServer.allowedCommandNames());
}
char const* RTSPServer::allowedCommandNames() {
  return "OPTIONS, DESCRIBE, SETUP, TEARDOWN, PLAY, PAUSE, GET_PARAMETER, SET_PARAMETER";
}
```

//get_paramenter和set_paramenter消息处理函数，可以作为保活消息使用

virtual void handleCmd_GET_PARAMETER(char const* fullRequestStr);

virtual void handleCmd_SET_PARAMETER(char const* fullRequestStr);

GET_PARAMETER && SET_PARAMETER消息：直接返回200 OK

```
void RTSPServer::RTSPClientConnection
::handleCmd_GET_PARAMETER(char const* /*fullRequestStr*/) {
  // By default, we implement "GET_PARAMETER" (on the entire server)
  // (If you want to handle this type of "GET_PARAMETER" differently,
  // and "RTSPServer::RTSPClientConnection", and then reimplement this
  setRTSPResponse("200 OK", LIVEMEDIA_LIBRARY_VERSION_STRING);
}

void RTSPServer::RTSPClientConnection
::handleCmd_SET_PARAMETER(char const* /*fullRequestStr*/) {
  // By default, we implement "SET_PARAMETER" (on the entire server)
  // (If you want to handle this type of "SET_PARAMETER" differently,
  // and "RTSPServer::RTSPClientConnection", and then reimplement this
  setRTSPResponse("200 OK");
}
```

//describle消息，用于获取流的sdp消息，告知播放器端，流的相关规格(编码、宽高、采样率。。。)

//通过url找到对应的ServerMediaSession，然后通过该ServerMediaSession获取SDP信息

virtual void handleCmd_DESCRIBE(char const* urlPreSuffix, char const* urlSuffix, char const* fullRequestStr);

DESCRIBE消息：用于获取流的sdp信息

通过请求的URL，拿到流名称，通过流名称找对应的ServerMediaSession，通过该ServerMediaSession获取流的SDP信息。

```
  // Begin by looking up the "ServerMediaSession" object for the specified "urlTotalSuffix":
  // 通过流名称找到对应的ServerMediaSession
  session = fOurServer.lookupServerMediaSession(urlTotalSuffix);
  if (session == NULL) {
    handleCmd_notFound();
    break;
  }

  // Increment the "ServerMediaSession" object's reference count, in case someone removes it
  // while we're using it:
  session->incrementReferenceCount();

  // Then, assemble a SDP description for this session:
  // 通过ServerMediaSession获取SDP信息
  sdpDescription = session->generateSDPDescription();
  if (sdpDescription == NULL) {
    // This usually means that a file name that was specified for a
    // "ServerMediaSubsession" does not exist.
    setRTSPResponse("404 File Not Found, Or In Incorrect Format");
    break;
  }
```

```
DESCRIBE rtsp://192.168.8.20:554/xiyou.mpg RTSP/1.0
Accept: application/sdp
CSeq: 2
User-Agent: Lavf57.71.100

RTSP/1.0 200 OK
CSeq: 2
Date: Sat, Aug 04 2018 08:01:20 GMT
Content-Base: rtsp://192.168.8.20/xiyou.mpg/
Content-Type: application/sdp
Content-Length: 480

v=0
o=- 1533369680103952 1 IN IP4 192.168.8.20
s=MPEG-1 or 2 Program Stream, streamed by the LIVE555 Media Server
i=xiyou.mpg
t=0 0
a=tool:LIVE555 Streaming Media v2018.02.28
a=type:broadcast
a=control:*
a=range:npt=0-292.383
a=x-qt-text-nam:MPEG-1 or 2 Program Stream, streamed by the LIVE555 Media Server
a=x-qt-text-inf:xiyou.mpg
m=video 0 RTP/AVP 32
c=IN IP4 0.0.0.0
b=AS:500
a=control:track1
m=audio 0 RTP/AVP 14
c=IN IP4 0.0.0.0
b=AS:128
a=control:track2
```

SETUP、PLAY、PAUSE和TEARDOWN消息转到RTSPClientSession类中执行。接收到SETUP消息时，live555会为该路连接生成一个sessionId，live555内部通过该sessionId区分不同的客户端连接。SETUP后续的消息交互都会携带该sessionId；

RTSPClientSession：客户端会话类

https://www.wolfcstech.com/2017/09/05/live555_src_analysis_setup/

// Make the handler functions for each command virtual, to allow subclasses to redefine them:

virtual void handleCmd_SETUP(RTSPClientConnection* ourClientConnection,

char const* urlPreSuffix, char const* urlSuffix, char const* fullRequestStr);

a、首先根据流名称找到对应的会话资源ServerMediaSession；

b、解析请求头 Transport: 中包含的关于客户端请求的传输方式的内容，如使用 UDP 传输还是用 TCP 传输，客户端为 RTP/RTCP 传输开辟的端口等；

c、解析请求头中的 Range: 和 x-playNow: 以支持 SETUP 和 PLAY 的合并(非标)；

d、为会话分配网络资源，如服务器端的 RTP/RTCP 端口等；

e、产生响应消息。

SETUP消息：用来确认传输协议(tcp/udp)和数据传输端口

1、tcp传输：rtsp消息 + rtp数据 + rtcp控制消息全部在一条链路上发送；

```
SETUP rtsp://192.168.8.20/xiyou.mpg/track1 RTSP/1.0
Transport: RTP/AVP/TCP;unicast;interleaved=0-1
CSeq: 3
User-Agent: Lavf57.71.100

RTSP/1.0 200 OK
CSeq: 3
Date: Mon, Jul 23 2018 11:36:52 GMT
Transport: RTP/AVP/
TCP;unicast;destination=192.168.8.20;source=192.168.8.20;interleaved=0-1
Session: 873C1B69;timeout=65

SETUP rtsp://192.168.8.20/xiyou.mpg/track2 RTSP/1.0
Transport: RTP/AVP/TCP;unicast;interleaved=2-3
CSeq: 4
User-Agent: Lavf57.71.100
Session: 873C1B69

RTSP/1.0 200 OK
CSeq: 4
Date: Mon, Jul 23 2018 11:36:52 GMT
Transport: RTP/AVP/
TCP;unicast;destination=192.168.8.20;source=192.168.8.20;interleaved=2-3
Session: 873C1B69;timeout=65
```

rtsp消息：ip head + tcp head + rtsp ；

rtcp消息：ip head + tcp head + rtp-over-rtsp head + rtcp；

rtp消息：ip head + tcp head + rtp-over-rtsp head + rtp；

track1一般为视频，track2为音频；

interleaved用来区分tcp模式下的数据和控制协议，偶数为数据，奇数为控制消息；

```
50888 2018-07-23 11:38:18.459334000   192.168.8.20   192.168.8.20   RTCP   196 Sender Report   Source description
51972 2018-07-23 11:38:20.241929000   192.168.8.20   192.168.8.20   RTCP   196 Sender Report   Source description

Frame 50888: 196 bytes on wire (1568 bits), 100 bytes captured (800 bits) on interface 0
Null/Loopback
Internet Protocol Version 4, Src: 192.168.8.20 (192.168.8.20), Dst: 192.168.8.20 (192.168.8.20)
Transmission Control Protocol, Src Port: rtsp (554), Dst Port: 29786 (29786), Seq: 15050325, Ack: 850, Len: 56
[2 Reassembled TCP Segments (60 bytes): #50886(4), #50888(56)]
RTSP Interleaved Frame, Channel: 0x01, 56 bytes
  Magic: 0x24
  Channel: 0x01
  Length: 56 bytes
Real-time Transport Control Protocol (Sender Report)
Real-time Transport Control Protocol (Source description)
[RTCP frame length check: OK - 56 bytes]
51972 2018-07-23 11:38:20.241929000   192.168.8.20   192.168.8.20   RTCP   196 Sender Report   Source description

Frame 51972: 196 bytes on wire (1568 bits), 100 bytes captured (800 bits) on interface 0
Null/Loopback
Internet Protocol Version 4, Src: 192.168.8.20 (192.168.8.20), Dst: 192.168.8.20 (192.168.8.20)
Transmission Control Protocol, Src Port: rtsp (554), Dst Port: 29786 (29786), Seq: 15368153, Ack: 850, Len: 56
[2 Reassembled TCP Segments (60 bytes): #51970(4), #51972(56)]
RTSP Interleaved Frame, Channel: 0x03, 56 bytes
  Magic: 0x24
  Channel: 0x03
  Length: 56 bytes
Real-time Transport Control Protocol (Sender Report)
Real-time Transport Control Protocol (Source description)
[RTCP frame length check: OK - 56 bytes]
```

01和03分别为视频和音频的rtcp数据包；

```
52148 2018-07-23 11:38:20.522308000   192.168.8.20   192.168.8.20   MPEG-1   2448 MPEG-1 message
52152 2018-07-23 11:38:20.561876000   192.168.8.20   192.168.8.20   MPEG-1   2996 MPEG-1 message

Frame 52148: 2448 bytes on wire (19584 bits), 1226 bytes captured (9808 bits) on interface 0
Null/Loopback
Internet Protocol Version 4, Src: 192.168.8.20 (192.168.8.20), Dst: 192.168.8.20 (192.168.8.20)
Transmission Control Protocol, Src Port: rtsp (554), Dst Port: 29786 (29786), Seq: 15419171, Ack: 850, Len: 1182
[2 Reassembled TCP Segments (1186 bytes): #52146(4), #52148(1182)]
RTSP Interleaved Frame, Channel: 0x00, 1182 bytes
  Magic: 0x24
  Channel: 0x00
  Length: 1182 bytes
Real-Time Transport Protocol
```

```
337 2018-07-23 11:36:52.921684000        192.168.8.20    192.168.8.20    RTP        1578 PT=MPEG-I/II Audio

⊞ Frame 337: 1578 bytes on wire (12624 bits), 791 bytes captured (6328 bits) on interface 0
⊞ Null/Loopback
⊞ Internet Protocol Version 4, Src: 192.168.8.20 (192.168.8.20), Dst: 192.168.8.20 (192.168.8.20)
⊞ Transmission Control Protocol, Src Port: rtsp (554), Dst Port: 29786 (29786), Seq: 96163, Ack: 626, Len: 747
⊞ [2 Reassembled TCP Segments (751 bytes): #335(4), #337(747)]
⊟ RTSP Interleaved Frame, Channel: 0x02, 747 bytes
    Magic: 0x24
    Channel: 0x02
    Length: 747 bytes
⊞ Real-Time Transport Protocol
```

02和04分别为视频和音频的数据包；

```
37779 2018-07-23 11:37:56.322444000        192.168.8.20    192.168.8.20    RTSP       312 Reply: RTSP/1.0 200 OK
   15 2018-07-23 11:36:52.307093000        192.168.8.20    192.168.8.20    RTSP/SDP   1370 Reply: RTSP/1.0 200 OK,

Frame 37779: 312 bytes on wire (2496 bits), 158 bytes captured (1264 bits) on interface 0
Null/Loopback
Internet Protocol Version 4, Src: 192.168.8.20 (192.168.8.20), Dst: 192.168.8.20 (192.168.8.20)
Transmission Control Protocol, Src Port: rtsp (554), Dst Port: 29786 (29786), Seq: 11191341, Ack: 850, Len: 114
Real Time Streaming Protocol
```

rtsp消息没有4字节的rtp-over-rtsp head；

tcp模式下，通过附加4字节的rtp-over-rtsp head，用来区分数据和控制协议；

2、udp传输：rtsp消息tcp传输，rtp数据 + rtcp控制udp传输，rtp和rtcp传输端口不同；

```
SETUP rtsp://192.168.8.20/xiyou.mpg/track1 RTSP/1.0
Transport: RTP/AVP/UDP;unicast;client_port=24696-24697
CSeq: 3
User-Agent: Lavf57.71.100

RTSP/1.0 200 OK
CSeq: 3
Date: Mon, Jul 23 2018 11:26:19 GMT
Transport: RTP/
AVP;unicast;destination=192.168.8.20;source=192.168.8.20;client_port=24696-24697;server
_port=6970-6971
Session: DB1920D4;timeout=65

SETUP rtsp://192.168.8.20/xiyou.mpg/track2 RTSP/1.0
Transport: RTP/AVP/UDP;unicast;client_port=24698-24699
CSeq: 4
User-Agent: Lavf57.71.100
Session: DB1920D4

RTSP/1.0 200 OK
CSeq: 4
Date: Mon, Jul 23 2018 11:26:19 GMT
Transport: RTP/
AVP;unicast;destination=192.168.8.20;source=192.168.8.20;client_port=24698-24699;server
_port=6972-6973
Session: DB1920D4;timeout=65
```

rtsp消息通过tcp传输；

track1一般为视频，track2为音频，udp模式下，音频和视频的传输端口不同；

偶数端口传数据，奇数端口传控制消息。24696传视频数据，24697传rtcp消息；

udp模式下，通过传输端口区分音频、视频和控制协议；

//处理rtsp的PLAY消息

virtual void handleCmd_PLAY(RTSPClientConnection* ourClientConnection,
ServerMediaSubsession* subsession, char const* fullRequestStr);

设置播放速度，播放位置，然后开始播放流；

PLAY消息：用来启动播放，开始流传输

1、tcp模式下：

```
PLAY rtsp://192.168.8.20/xiyou.mpg/ RTSP/1.0
Range: npt=0.000-
CSeq: 5
User-Agent: Lavf57.71.100
Session: E2CA2FF5

$..8....`.v...D.I...0z.k............`.v...USER-20160103ZT...
$..8..........D.I.sT..V.............  ....USER-20160103ZT...RTSP/1.0 200 OK
CSeq: 5
Date: Tue, Jul 24 2018 06:24:59 GMT
Range: npt=0.000-
Session: E2CA2FF5
RTP-Info: url=rtsp://192.168.8.20/xiyou.mpg/
track1;seq=37195;rtptime=813357715,url=rtsp://192.168.8.20/xiyou.mpg/
track2;seq=41915;rtptime=359132931
```

RTCP

```
62 2018-07-24 06:24:59.289725000    192.168.8.20    192.168.8.20    MPEG-1    2
⊞ Frame 62: 2996 bytes on wire (23968 bits), 1500 bytes captured (12000 bits) on in
⊞ Null/Loopback
⊞ Internet Protocol Version 4, Src: 192.168.8.20 (192.168.8.20), Dst: 192.168.8.20
⊞ Transmission Control Protocol, Src Port: rtsp (554), Dst Port: 61856 (61856), Seq
⊞ [2 Reassembled TCP Segments (1460 bytes): #60(4), #62(1456)]
⊞ RTSP Interleaved Frame, Channel: 0x00, 1456 bytes
⊟ Real-Time Transport Protocol
    10.. .... = Version: RFC 1889 Version (2)
    ..0. .... = Padding: False
    ...0 .... = Extension: False
    .... 0000 = Contributing source identifiers count: 0
    0... .... = Marker: False
    Payload type: MPEG-I/II Video (32)
    Sequence number: 37195
    Timestamp: 813357715
66 2018-07-24 06:24:59.289787000    192.168.8.20    192.168.8.20    RTP    1578 PT=MPEG-I/II Audio,
⊞ Frame 66: 1578 bytes on wire (12624 bits), 791 bytes captured (6328 bits) on interface 0
⊞ Null/Loopback
⊞ Internet Protocol Version 4, Src: 192.168.8.20 (192.168.8.20), Dst: 192.168.8.20 (192.168.8.20)
⊞ Transmission Control Protocol, Src Port: rtsp (554), Dst Port: 61856 (61856), Seq: 3009, Ack: 626, Len:
⊞ [2 Reassembled TCP Segments (751 bytes): #64(4), #66(747)]
⊞ RTSP Interleaved Frame, Channel: 0x02, 747 bytes
⊟ Real-Time Transport Protocol
    10.. .... = Version: RFC 1889 Version (2)
    ..0. .... = Padding: False
    ...0 .... = Extension: False
    .... 0000 = Contributing source identifiers count: 0
    1... .... = Marker: True
    Payload type: MPEG-I/II Audio (14)
    Sequence number: 41915
    Timestamp: 3591329317
    Synchronization Source identifier: 0xb31980d2 (3004793042)
```

track1一般为视频，track2为音频；

seq是第一个rtp数据包的序列号，rtptime为第一个rtp数据包额时间戳；视频第一个rtp序列号为37195，时间戳为813357715；音频第一个rtp包的序列号为41915，时间戳为3591329317;

2、udp模式下：

```
PLAY rtsp://192.168.8.20/xiyou.mpg/ RTSP/1.0
Range: npt=0.000-
CSeq: 5
User-Agent: Lavf57.71.100
Session: C737A5D0

RTSP/1.0 200 OK
CSeq: 5
Date: Tue, Jul 24 2018 05:22:21 GMT
Range: npt=0.000-
Session: C737A5D0
RTP-Info: url=rtsp://192.168.8.20/xiyou.mpg/
track1;seq=23092;rtptime=2142436822,url=rtsp://192.168.8.20/xiyou.mpg/
track2;seq=43183;rtptime=1194323623
```

track1一般为视频，track2为音频；

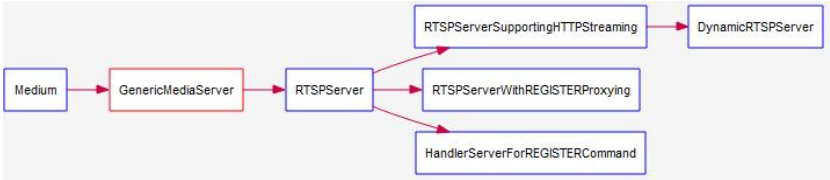
seq是第一个rtp数据包的序列号，rtptime为第一个rtp数据包额时间戳；视频第一个rtp序列号为23092，时间戳为2142436822；音频第一个rtp包的序列号为43183，时间戳为1194323623;

```
36 2018-07-24 05:22:21.571366000    192.168.8.20    192.168.8.20    MPEG-1    2972 MPEG-1 message
37 2018-07-24 05:22:21.571420000    192.168.8.20    192.168.8.20    RTP       1554 PT=MPEG-I/II Audio
38 2018-07-24 05:22:21.571469000    192.168.8.20    192.168.8.20    MPEG-1    2972 MPEG-1 message
⊞ Frame 36: 2972 bytes on wire (23776 bits), 1488 bytes captured (11904 bits) on interface 0
⊞ Null/Loopback
⊞ Internet Protocol Version 4, Src: 192.168.8.20 (192.168.8.20), Dst: 192.168.8.20 (192.168.8.20)
⊞ User Datagram Protocol, Src Port: 6970 (6970), Dst Port: 18018 (18018)
⊟ Real-Time Transport Protocol
  ⊞ [Stream setup by RTSP (frame 20)]
    10.. .... = Version: RFC 1889 Version (2)
    ..0. .... = Padding: False
    ...0 .... = Extension: False
    .... 0000 = Contributing source identifiers count: 0
    0... .... = Marker: False
    Payload type: MPEG-I/II Video (32)
    Sequence number: 23092
    [Extended sequence number: 88628]
    Timestamp: 2142436822
    Synchronization Source identifier: 0xfc896974 (4236863860)
37 2018-07-24 05:22:21.571420000    192.168.8.20    192.168.8.20    RTP       1554 PT=MPEG-I/II Audio
38 2018-07-24 05:22:21.571469000    192.168.8.20    192.168.8.20    MPEG-1    2972 MPEG-1 message
Null/Loopback
Internet Protocol Version 4, Src: 192.168.8.20 (192.168.8.20), Dst: 192.168.8.20 (192.168.8.20)
User Datagram Protocol, Src Port: 6972 (6972), Dst Port: 18020 (18020)
Real-Time Transport Protocol
⊞ [Stream setup by RTSP (frame 24)]
  10.. .... = Version: RFC 1889 Version (2)
  ..0. .... = Padding: False
  ...0 .... = Extension: False
  .... 0000 = Contributing source identifiers count: 0
  1... .... = Marker: True
  Payload type: MPEG-I/II Audio (14)
  Sequence number: 43183
  [Extended sequence number: 43183]
  Timestamp: 1194323623
  Synchronization Source identifier: 0xfda5c1d3 (4255498707)
  Payload: 00000000fffdb00477554455446655444444549249 2d9122...
```

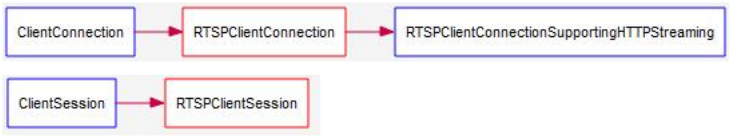
---

liveMeida类族：

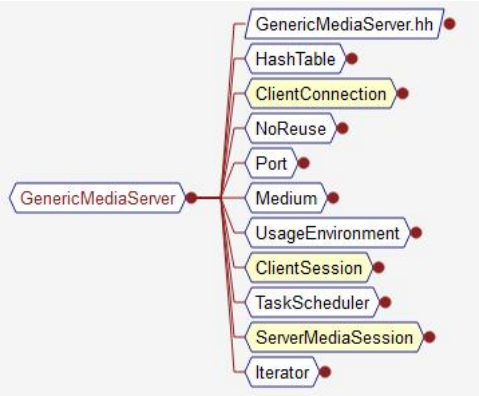
GenericMediaServer类族：服务类，监听指定IP+PORT，接受客户端连接，建立连接会话，解析协议，响应请求；



ClientConnection类族：客户端连接类，处理与客户端的socket交互，接受解析消息；

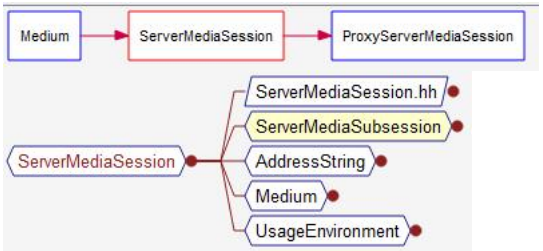


GenericMediaServer的依赖如下图，维护三个哈希表：
ClientConnection客户端连接哈希表。key：与客户端的连接套接字；
ClientSession客户端会话哈希表；
ServerMediaSession流哈希表。key：流名称；

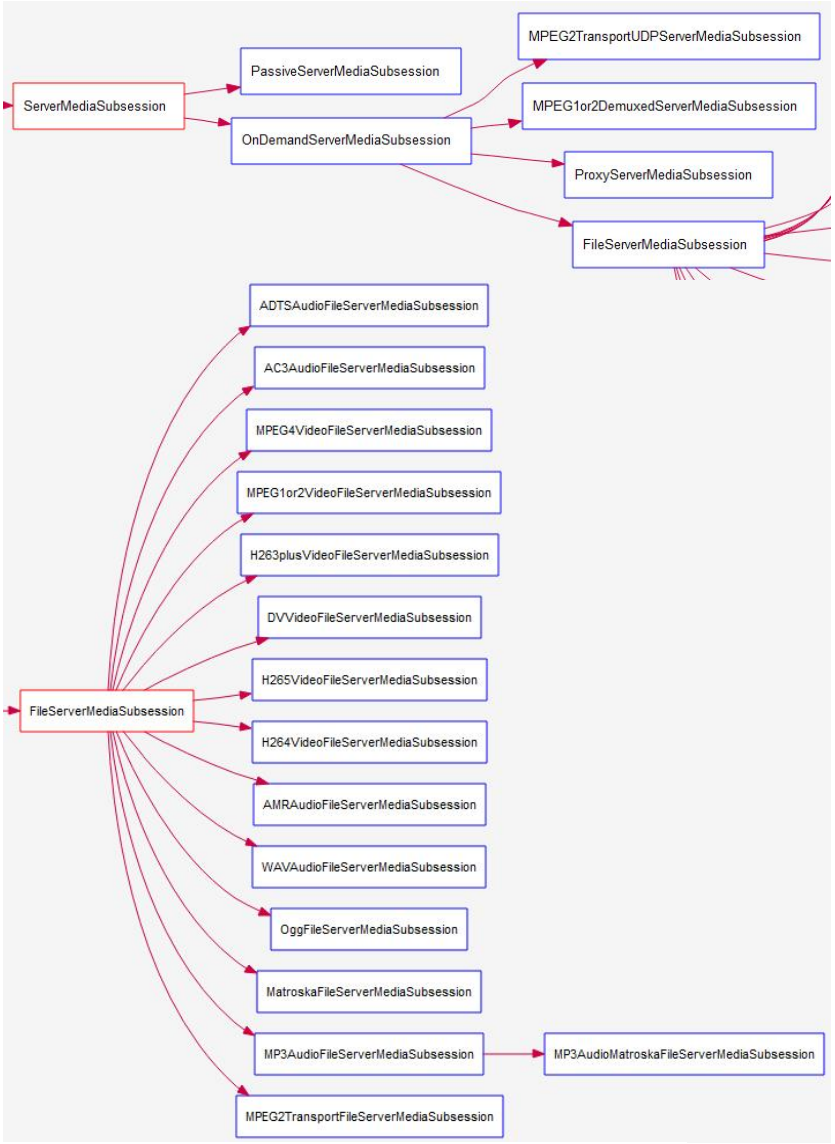


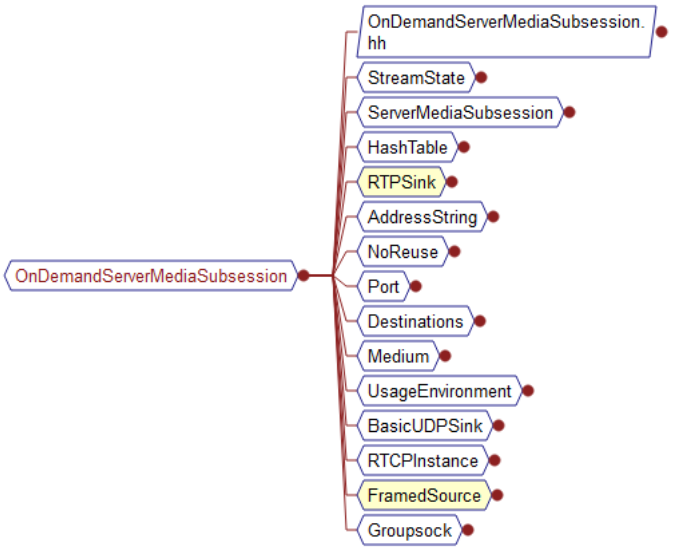
服务端流程：
1、创建GenericMediaServer监听端口，创建流，接受客户端请求；
2、接受到来的客户端请求，创建ClientConnection，用于接受客户端数据；
3、根据流名称找到对应的ServerMediaSession，通过该ServerMediaSession会话获取子会话ServerMediaSubSession，通过ServerMediaSubSession获取SDP信息；
4、创建ClientSession，处理SETUP，PLAY等后续消息；

ServerMediaSession类族：表示一路流，或者说一路会话。包含至少一种子会话(视频或者音频)。主要依赖ServerMediaSubSession(子会话)；



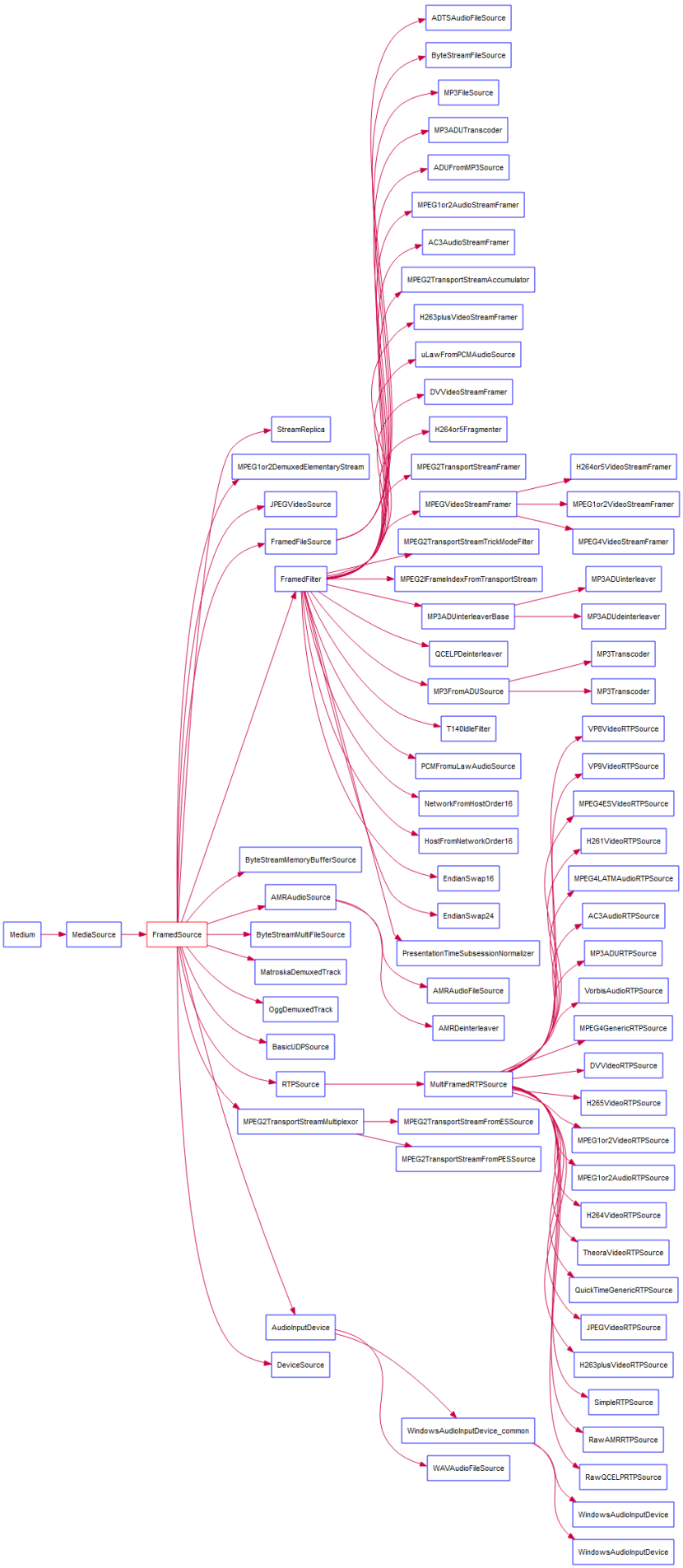
ServerMediaSession：表示一路流，与流名称一一对应；在GenericMediaServer中维护
ServerMediaSubSession：表示一个子会话，通常为视频或者音频；处理取流等操作；





每个ServerMediaSubSession都会对应一个**FramedSource**用来做流处理；有一个**RTPSink**用来做rtp包处理和发送；

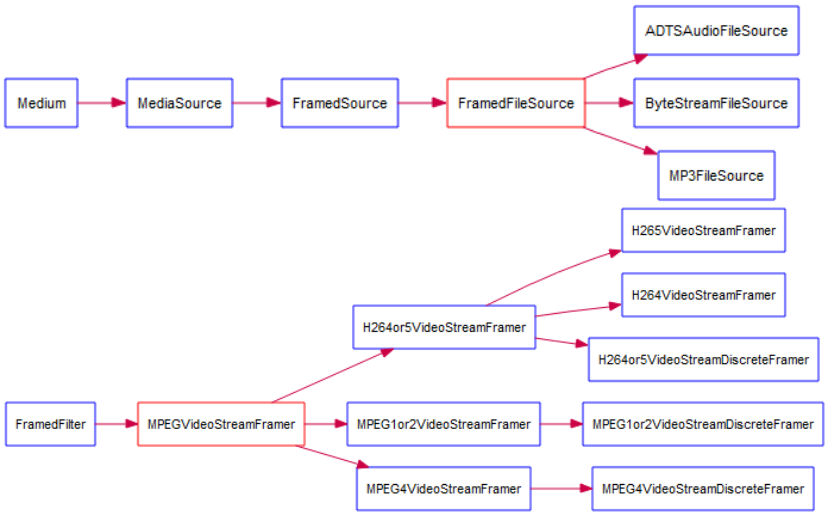
FrameSource类族：源数据类族，用来获取源数据；

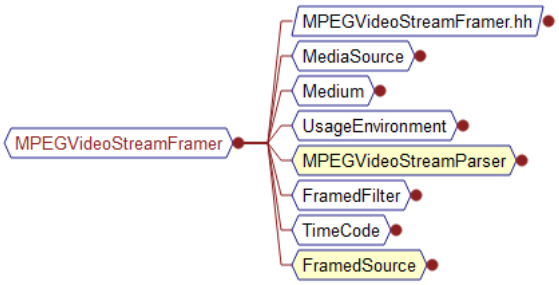

Medium
MediaSource
FramedSource
StreamReplica
MPEG1or2DemuxedElementaryStream
JPEGVideoSource
FramedFileSource
FramedFilter
ADTSAudioFileSource
ByteStreamFileSource
MP3FileSource
MP3ADUTranscoder
ADUFromMP3Source
MPEG1or2AudioStreamFramer
AC3AudioStreamFramer
MPEG2TransportStreamAccumulator
H263plusVideoStreamFramer
uLawFromPCMAudioSource
DVVideoStreamFramer
H264or5Fragmenter
MPEG2TransportStreamFramer
MPEGVideoStreamFramer
H264or5VideoStreamFramer
MPEG1or2VideoStreamFramer
MPEG4VideoStreamFramer
MPEG2TransportStreamTrickModeFilter
MPEG2IFrameIndexFromTransportStream
MP3ADUinterleaver
MP3ADUinterleaverBase
MP3ADUdeinterleaver
QCELPDeinterleaver
MP3Transcoder
MP3FromADUSource
MP3Transcoder
T140IdleFilter
PCMFromuLawAudioSource
NetworkFromHostOrder16
HostFromNetworkOrder16
EndianSwap16
EndianSwap24
PresentationTimeSubsessionNormalizer
AMRAudioFileSource
AMRDeinterleaver
ByteStreamMemoryBufferSource
AMRAudioSource
ByteStreamMultiFileSource
MatroskaDemuxedTrack
OggDemuxedTrack
BasicUDPSource
RTPSource
MultiFramedRTPSource
VP8VideoRTPSource
VP9VideoRTPSource
MPEG4ESVideoRTPSource
H261VideoRTPSource
MPEG4LATMAudioRTPSource
AC3AudioRTPSource
MP3ADURTPSource
VorbisAudioRTPSource
MPEG4GenericRTPSource
DVVideoRTPSource
H265VideoRTPSource
MPEG1or2VideoRTPSource
MPEG1or2AudioRTPSource
H264VideoRTPSource
TheoraVideoRTPSource
QuickTimeGenericRTPSource
JPEGVideoRTPSource
H263plusVideoRTPSource
SimpleRTPSource
RawAMRRTPSource
RawQCELPRTPSource
WindowsAudioInputDevice
WindowsAudioInputDevice
MPEG2TransportStreamMultiplexor
MPEG2TransportStreamFromESSource
MPEG2TransportStreamFromPESSource
AudioInputDevice
DeviceSource
WindowsAudioInputDevice_common
WAVAudioFileSource

基类FrameSource，有很多的派生类，FramedFileSource和MPEGVideoStreamFramer是两个重要的派生类，下面只大致分析这两个派生类族；

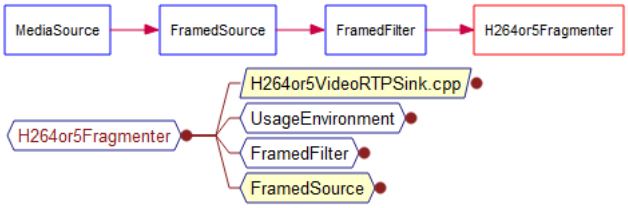
FramedFileSource派生类族，主要用来读取文件，通过读文件获取流；

MPEGVideoStreamFramer派生类族，大致看了一下，感觉是对流进行分析处理的，譬如说处理sps，pps，vps等nalu单元；



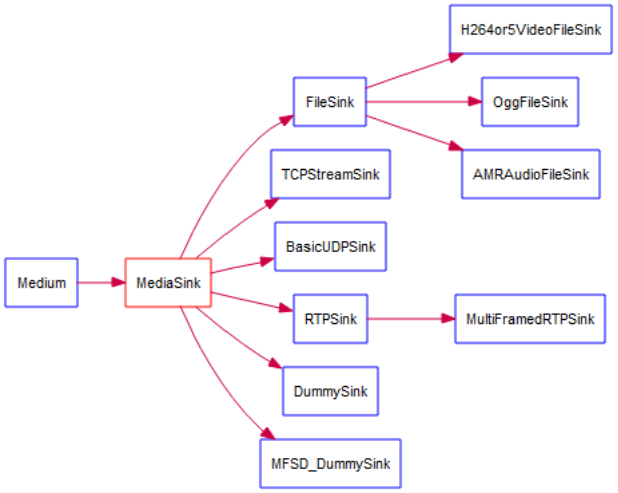


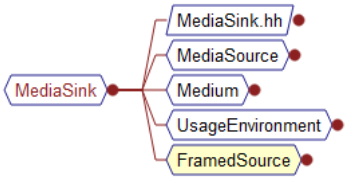
每一个MPEGVideoStreamFramer类都会有一个**MPEGVideoStreamParser**类，用来做码流分析；



H264or5Fragmenter类用来处理nalu片段，只将可以发送的片段传递给H264or5VideoRTPSink类；

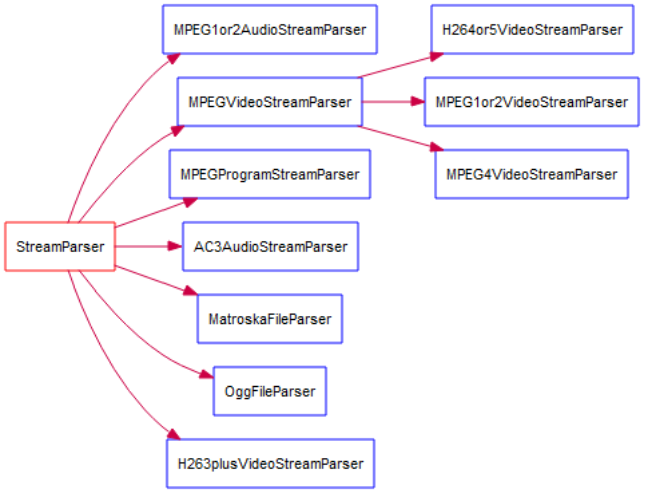
MediaSink类族：MediaSink处理的是将发往接收端的数据，所以MediaSink可以理解为接收端的管道，是接收端数据的汇聚地；通过不同的派生类，将数据通过不同途径发往接收端。

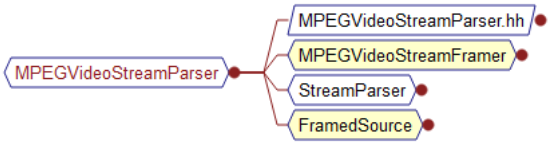




如上图，MediaSink依赖FramedSource类；FramedSource类为源端数据类，通过该类获取数据源，然后将数据层层传递给MediaSink；

StreamParser类族：用作流分析处理，通过不同的派生类来分析不同的流；依赖的输入流由FramedSource提供；





每一个MPEGVideoStreamParser类都依赖一个**MPEGVideoStreamFramer**类，该MPEGVideoStreamFramer类用来获取流；

总结：

liveMedia有5大重要类族，分别为：

GenericMediaServer类族：主要功能就是开启服务，监听指定端口，接受客户端连接请求，解析客户端请求消息，并响应客户端请求；

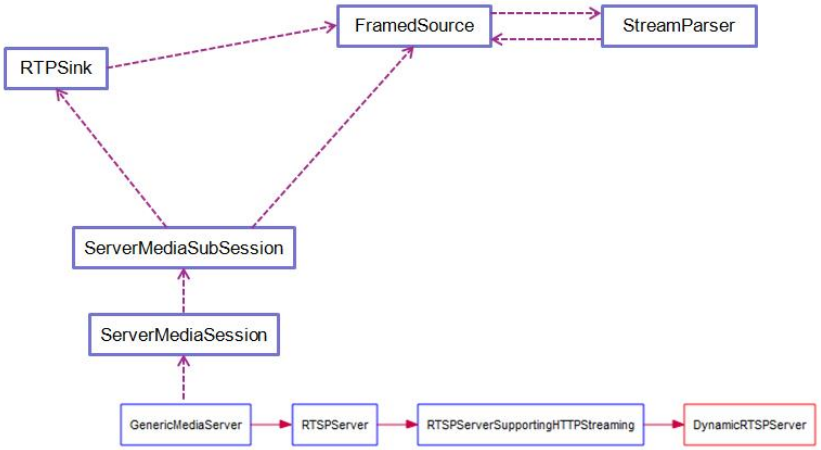
ServerMediaSession类族：该类族主要对应于流和子会话，每一路流包含若子会话；一路流对应一个ServerMediaSession，一个子会话对应一个ServerMediaSubSession；

FrameSource类族：该类族主要用来提供媒体源数据，通过不同的派生类提供不同的媒体源数据；

MediaSink类族：可以将该类族理解为接收端的媒体数据汇聚地，通过该类及其派生类通过不同途径将媒体数据发往接收端；

StreamParser类族：主要提供媒体流数据的分析，为发送做准备；

类族依赖关系：



RTSP消息流走向分析：

数据流走向分析：

////////////