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。

一、一些基本组件类(时间类，延时队列类，处理程序描述类，哈希表类)

这四个基本类主要是用在 **TaskScheduler** 和 **UsageEnvironment** 相关的类中，是这两个类的重要组成部分。

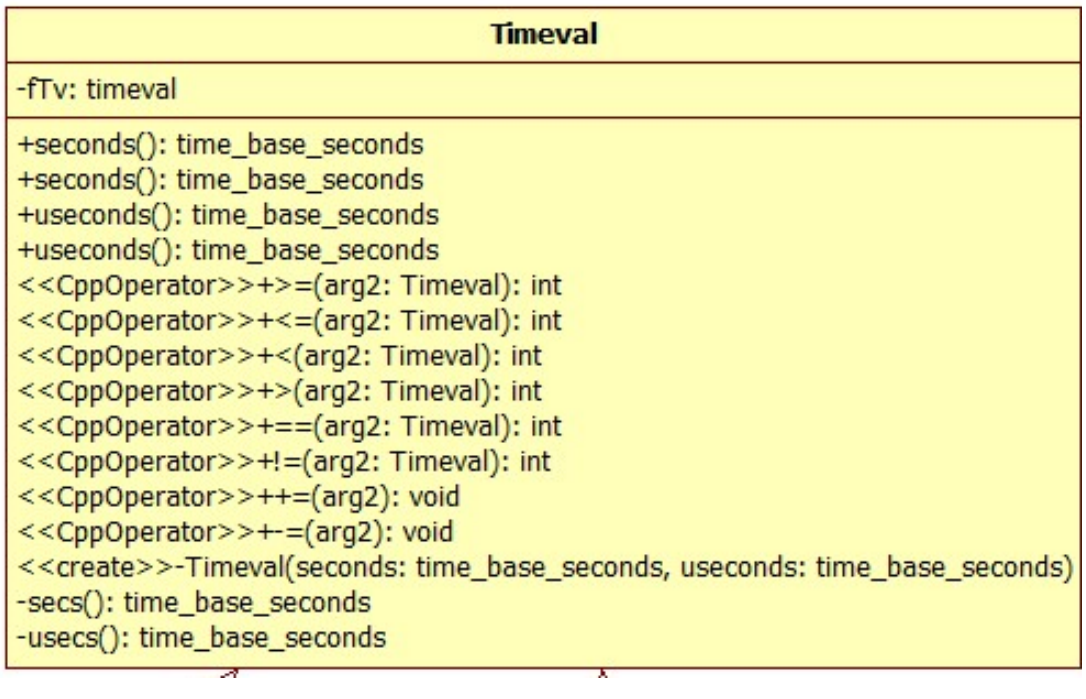
1.时间相关类

1) TimeVal 类

TimeVal 类定义在 live555sourcecontrol\BasicUsageEnvironment\include\DelayQueue.hh 文件中。其实质上是对 struct timeval 的封装。

先来看看 TimeVal 类的组成。

其只有一个数据成员，就是一个 timeval 的结构体 fTv。TimeVal 类封装的所有方法都是对其的操作。



再来看看这个 struct timeval 结构体的定义

```
struct timeval {
        long    tv_sec;         /* seconds 秒*/
        long    tv_usec;        /* and microseconds 微秒*/
};
```

下面是 TimeVal 类的定义。

```
class Timeval {
public:
  time_base_seconds seconds() const {
    return fTv.tv_sec;
  }
  time_base_seconds seconds() {
    return fTv.tv_sec;
  }
  time_base_seconds useconds() const {
    return fTv.tv_usec;
  }
```

```
  time_base_seconds useconds() {
    return fTv.tv_usec;
  }

  int operator>=(Timeval const& arg2) const;
  int operator<=(Timeval const& arg2) const {
    return arg2 >= *this;
  }
  int operator<(Timeval const& arg2) const {
    return !(*this >= arg2);
  }
  int operator>(Timeval const& arg2) const {
    return arg2 < *this;
  }
  int operator==(Timeval const& arg2) const {
    return *this >= arg2 && arg2 >= *this;
  }
  int operator!=(Timeval const& arg2) const {
    return !(*this == arg2);
  }

  void operator+=(class DelayInterval const& arg2);
  void operator-=(class DelayInterval const& arg2);
  // returns ZERO iff arg2 >= arg1

protected:
  Timeval(time_base_seconds seconds, time_base_seconds useconds) {
    fTv.tv_sec = seconds; fTv.tv_usec = useconds;
  }

private:
  time_base_seconds& secs() {
    return (time_base_seconds&)fTv.tv_sec;
  }
  time_base_seconds& usecs() {
    return (time_base_seconds&)fTv.tv_usec;
  }

  struct timeval fTv;
};
```

TimeVal 类还有两个派生类。

2）DelayInterval 延时间隔类

DelayInterval 这个类只是为了在名字上方便使用。我们可以看上面的 TimeVal 类，其带参构造函数是 protected 权限的，这里的定义就是暴露了一个构造接口，方便使用。

```
class DelayInterval: public Timeval {
public:
  DelayInterval(time_base_seconds seconds, time_base_seconds useconds)
    : Timeval(seconds, useconds) {}
};
```

除此之外 DelayInterval 类还重载了全局的 “ * ”运算符。注意，这个不是在 DelayInterval 类内部重载的，这里的第一个参数是 short 类型。其使用的时候是类似于这样的 result = arg1 * arg2;其中 result 和 arg2 是 DelayInterval 对象。

```
DelayInterval operator*(short arg1, DelayInterval const& arg2);
```

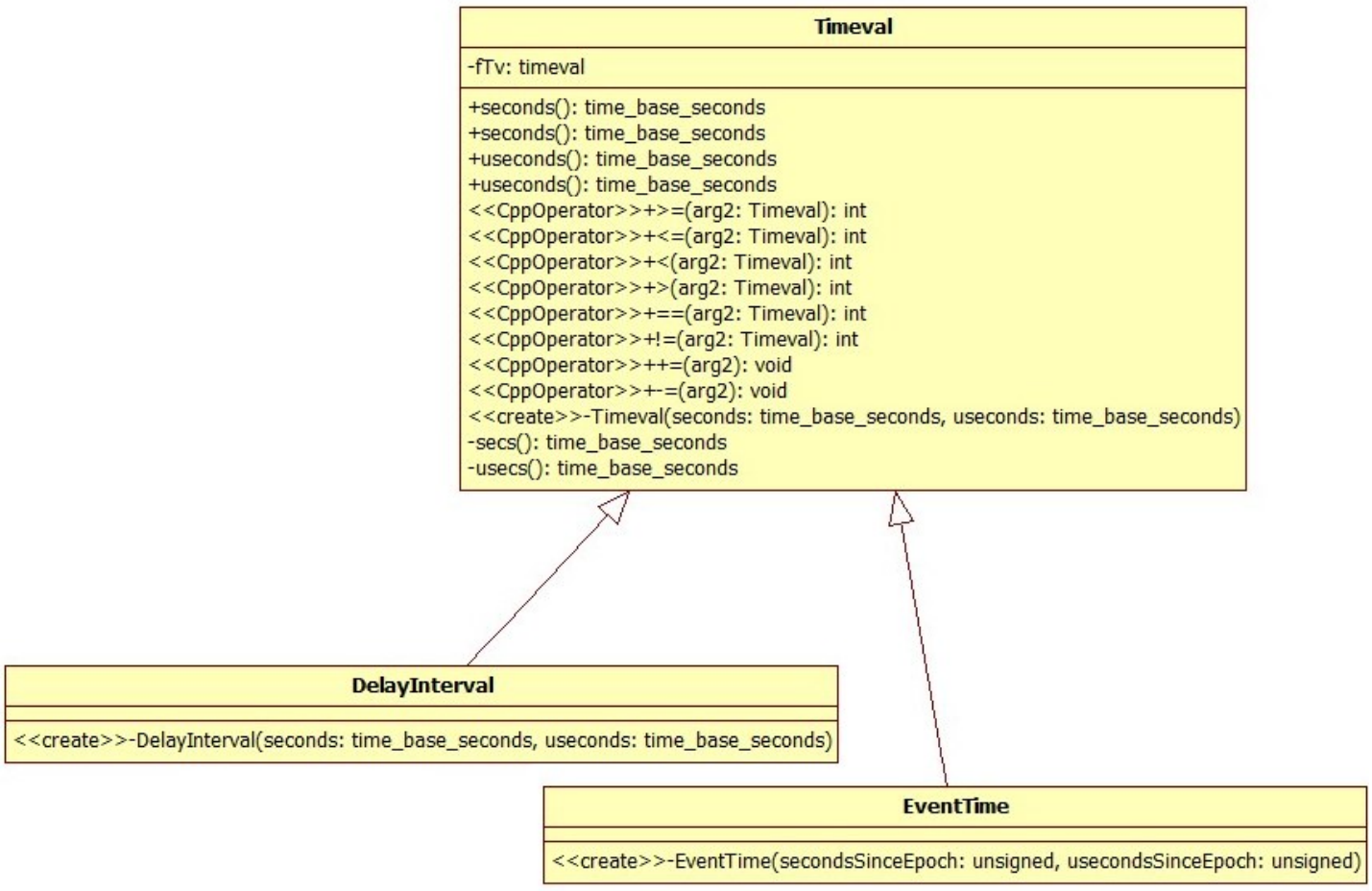
其实现如下

```
DelayInterval operator*(short arg1, const DelayInterval& arg2) {
  time_base_seconds result_seconds = arg1*arg2.seconds();
  time_base_seconds result_useconds = arg1*arg2.useconds();
  time_base_seconds carry = result_useconds/MILLION;
  result_useconds -= carry*MILLION;
  result_seconds += carry;
  return DelayInterval(result_seconds, result_useconds);
}
```

3）EventTime 事件时间类

这个类和 DelayInterval 类的是类似的，就是其构造函数默认参数是 0。

```
class EventTime: public Timeval {
public:
  EventTime(unsigned secondsSinceEpoch = 0,
      unsigned usecondsSinceEpoch = 0)
    // We use the Unix standard epoch: January 1, 1970
    : Timeval(secondsSinceEpoch, usecondsSinceEpoch) {}
};
```



4）全局函数 EventTime TimeNow();

全局函数 EventTime TimeNow()是用来获取当前时间的函数。其实现如下

```
EventTime TimeNow() {
  struct timeval tvNow;
  gettimeofday(&tvNow, NULL);
  return EventTime(tvNow.tv_sec, tvNow.tv_usec);
}
```

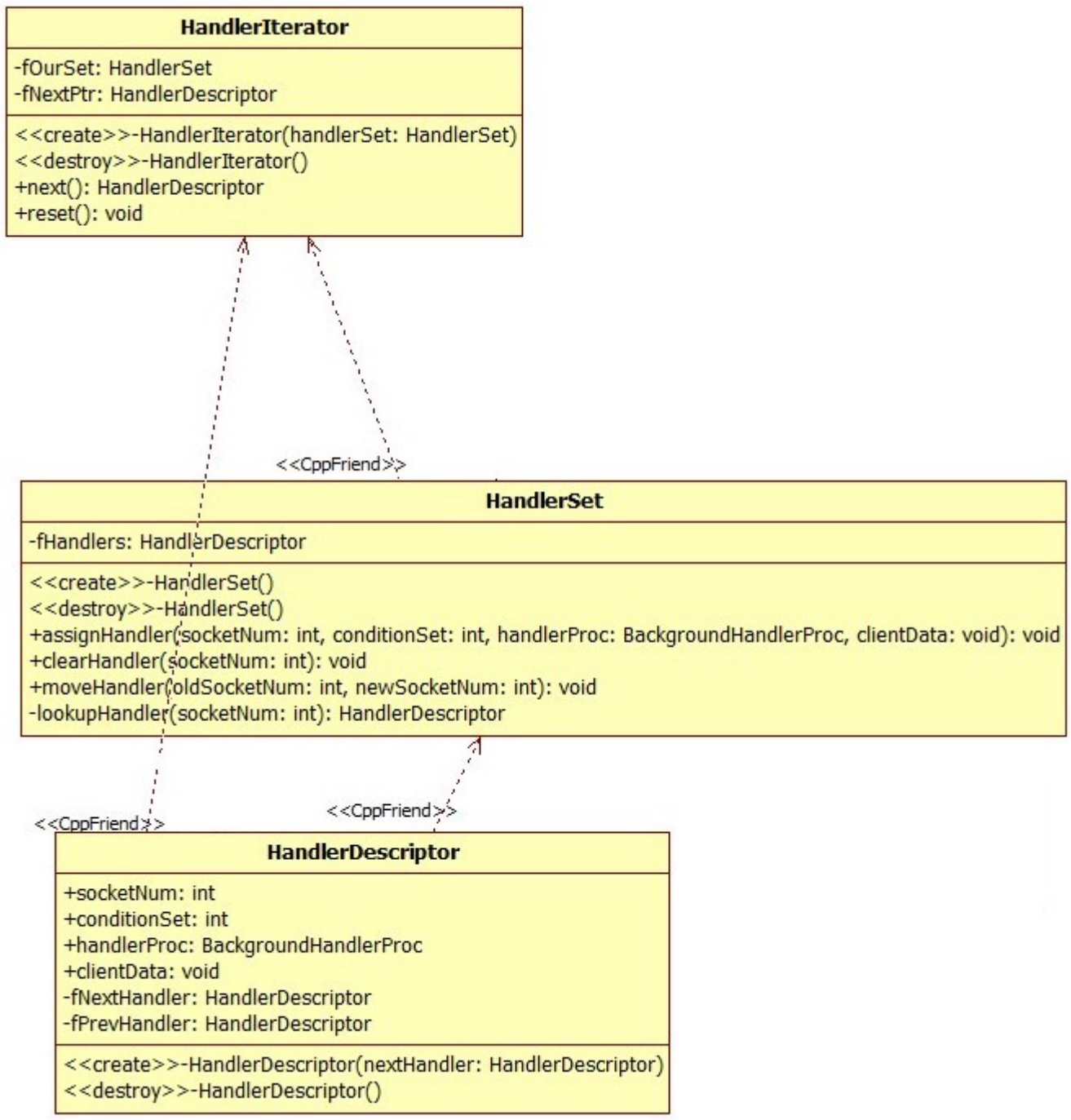
2．处理程序相关类

处理程序相关类一共有三个，其没有派生继承关系，但是其有友元关系和使用关系。处理程序相关类主要是用于对相关的处理函数的指针和数据的包装，方便在 DelayQueue 相关类中的使用等。

先来总的说以下三个类的关系。

HandlerDescriptor 是一个节点类，而 HandlerSet 是一个链表类，链表节点就是 HandlerDescriptor 对象。HandlerIterator 是一个迭代器类，其绑定一个 HandlerSet 对象。

处理程序相关的三个类都定义在 live555sourcecontrol\BasicUsageEnvironment\include\HandlerSet.hh 文件中。



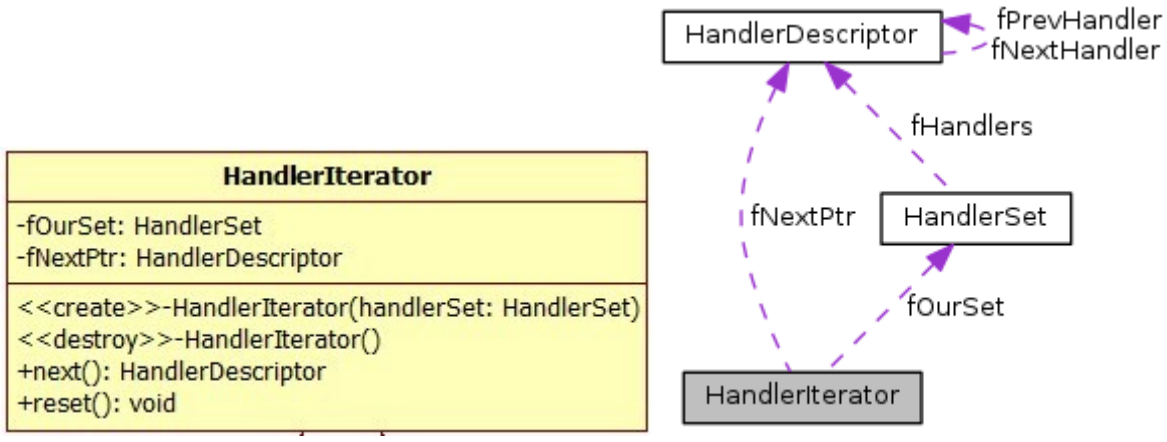
1）HandlerIterator 处理程序迭代器类

这里本应该先介绍 HandlerDescriptor 类的，因为这个类与它的关联比较大，就先介绍这个类。

HandlerIterator 是一个迭代器类，其有两个数据成员，分别是 HandlerSet 类对象的引用 fOurSet，以及一个 HandlerDescriptor 对象指针 fNextPtr。并且 HandlerIterator 同时是节点和链表的友元类。

fOurSet 是一个引用，就说明了 HandlerIterator 的初始化必须要绑定一个 HandlerSet 对象。而 HandlerSet 类的对象又是一个链表，其节点是 HandlerDescriptor 对象。**迭代器对象仅在 HandlerSet 类中使用。**

迭代器构造的时候，会将其 fNextPtr 指向链表的头节点的下一个。



下面是 HandlerIterator 类定义

```
// 处理程序描述链表迭代器类
class HandlerIterator {
public:
  // 必须绑定到一个处理程序描述链表对象，并调用 reset()将 fNextPtr 赋值为 handlerSet.fNextHandler
  HandlerIterator(HandlerSet& handlerSet);
  virtual ~HandlerIterator();

  // 返回 fNextPtr,并将 fNextPtr 指向下一个处理程序描述对象
  HandlerDescriptor* next(); // returns NULL if none
  void reset();   //将 fNextPtr 指向链表的头结点的下一个
private:
  HandlerSet& fOurSet;          //指向绑定链表的引用
  HandlerDescriptor* fNextPtr; //处理程序描述对象指针
};
```

next 方法(获取链表节点，迭代器后移)

这里返回的是当前迭代器指向的元素，但是迭代器会走向下一个。如果走到了末尾元素位置，迭代器将不会循环到第一个，而是停滞不前，并返回 NULL。

```
HandlerDescriptor* HandlerIterator::next() {
  HandlerDescriptor* result = fNextPtr;
  //要注意的是，这里是走到了最后一个，因为这是循环链表
  if (result == &fOurSet.fHandlers) { // no more
     result = NULL;
  }
  else {
     fNextPtr = fNextPtr->fNextHandler;
  }

  return result;
}
```

2) HandlerDescriptor 处理程序描述类

HandlerDescriptor 类是一个很重要的类，其保存了处理程序的函数指针和相关的数据的地址。在构建处理任务的时候，会使用到这个类的对象，处理任务的时候也会用到。

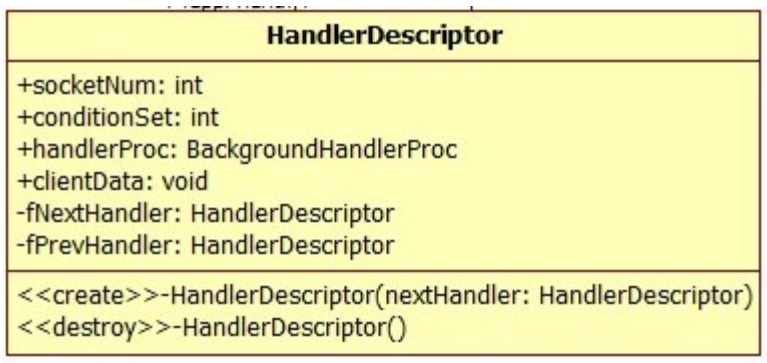
HandlerDescriptor 类同时将 HandlerIterator 类和 HandlerSet 类声明为友元类，方便了后面链表的操作。个人觉得这种封装方式不是特别好。只是个人看法而已。

这里的封装还有一个原因就是因为构造和析构都是 private 权限的，因为只能在其友元类 HandlerSet 中来调用，避免暴露接口。

这里要特别提一下数据成员 socketNum 和 conditionSet。socketNum 在链表中要来标识一个节点，那么这 socketNum 的值是如何赋值来的呢？这里先提一下，在使用到这个类对象的时候，会将一个 socket 套接口作为其值(windows 下是 SOCKET 类型 linux/uni

x 下是文件描述符，其实质都是 int 类型)，它必然是唯一的。这个变量名取为 socketNum 就是因为后面它将用于网络。而 conditionSet 是条件集合的意思，用来标识对应 socketNum 代表的套接口可以采取那写操作(读/写/异常)。

还有注意的是 handlerProc 的类型是一个**类成员函数指针**，它应该指向一个 TaskScheduler 的函数成员地址。(《C++必知必会》一个指向成员的指针并不指向一个具体的内存地址，它指向的是一个类的特定成员，而不是指向一个特定对象里的特定成员。)



下面是 HandlerDesciptor 类的定义

```
// 处理程序描述类(作为链表的节点)
class HandlerDescriptor {
//构造和析构都是 private 权限的，因为只能在其友元类 HandlerSet 中来调用
  // 如果 nextHandler 为其自身，自身就是双向链表的头结点
  // 否则将自身插入到 nextHandler 和 nextHandler->fPrevHandler 之间
  HandlerDescriptor(HandlerDescriptor* nextHandler);
  // 将自身从双向链表中移除。这个函数一般由 delete 操作来调用
  virtual ~HandlerDescriptor();
public:
  int socketNum;  //socket 在链表里面用来标识节点
  int conditionSet;  //条件集合
   //typedef void BackgroundHandlerProc(void* clientData, int mask);
  TaskScheduler::BackgroundHandlerProc* handlerProc; //后台处理程序函数指针
  void* clientData;  //客户端数据
private:
  // Descriptors are linked together in a doubly-linked list:
  friend class HandlerSet;
  friend class HandlerIterator;
  HandlerDescriptor* fNextHandler;   //下一个处理程序描述
  HandlerDescriptor* fPrevHandler;   //上一个处理程序描述
};
```

HandlerDescriptor 的构造函数

从其构造函数可以看出，其默认只被用于链表中作为节点存在。并且这个构造函数是 private 权限的，只有在本类或者友元类中可以使用其来构造对象。

```
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;
  }
}
```

HandlerDescriptor 的析构

必须说一下，这里析构不是简单的释放自身，这里将节点从链表中移除了。想一想，如果是头结点呢？也是没有问题的，只是操作之后并没有从链表移除。对头结点而言在析构结束后，就是整个链表的释放。

```
HandlerDescriptor::~HandlerDescriptor() {
  // Unlink this descriptor from a doubly-linked list:
  fNextHandler->fPrevHandler = fPrevHandler;
  fPrevHandler->fNextHandler = fNextHandler;
}
```

3) HandlerSet 处理程序链表类

这里使用的 Set 这个单词，Set 是集合的意思，这里实质上是一个双向循环链表。这个类比较重要，这里会详细的介绍。

HandlerSet 类只有一个数据成员，就是 HandlerDescriptor fHandlers;这是作为链表的头结点而存在的。

HandlerSet 的定义，代码如下

```
class HandlerSet {
public:
  //设置 fHandlers 的下一个和上一个指向 fHandler 自己
  HandlerSet();
  //逐个释放链表节点
  virtual ~HandlerSet();
  // 从链表中查找 socketNum 代表的 HandlerDescriptor,如果没有找到就创建一个并加入到链表
  void assignHandler(int socketNum, int conditionSet, TaskScheduler::BackgroundHandlerProc* handlerPro
c, void* clientData);
  //从链表中查找 socketNum 对应的 HandlerDescriptor，找到了就 delete
  void clearHandler(int socketNum);
  // 从链表中查找 oldSocketNum 代表的 HandlerDescriptor,找到了就将其 sockerNum 成员替换为 newSocketNum
  void moveHandler(int oldSocketNum, int newSocketNum);
private:
  // 从链表中查找 socketNum 代表的 HandlerDescriptor，没找到返回 NULL
  HandlerDescriptor* lookupHandler(int socketNum);
private:
  friend class HandlerIterator;
  HandlerDescriptor fHandlers; //处理程序描述对象 链表头节点
};
```

HandlerSet 的构造

在其构造函数中，默认对头结点 HandlerDescriptor fHandlers 进行了初始化操作。

```
HandlerSet::HandlerSet()
  : fHandlers(&fHandlers) {
  fHandlers.socketNum = -1; // shouldn't ever get looked at, but in case...
}
```

这里调用了 HandlerDescriptor 的构造，这个可以在之前的介绍中查看。这里可以看到，其将头结点的数据成员 socketNum 设置为了-1，之前我们说过，socketNum 在链表中被用来标识节点，这里说明了其是一个特殊的存在，**头结点不做为保存处理程序的节点**。

HandlerSet 的析构

析构函数就是释放链表，就是逐个释放除了头结点之外的节点。代码如下

```
HandlerSet::~HandlerSet() {
  // Delete each handler descriptor:
  while (fHandlers.fNextHandler != &fHandlers) {
    delete fHandlers.fNextHandler; // changes fHandlers->fNextHandler
  }
}
```

lookupHandler 方法

这里先说这个方法，因为后面的几个方法都用到了它。从方法名也可以看出来，这个方法是用来查找节点的。

这里要说以下的就是，这里面用到了迭代器。方法中创建了一个迭代器，并将自身绑定给了迭代器。前面说过迭代器构造的时候会将其 fNextPtr 指向链表的头结点的下一个。也就是说会从第二个节点开始查找。如果本身就只有头节点呢？因为在只有头结点的情况下，下一个节点就是头结点，其 socketNum 为-1，这里是没问题的。

```
HandlerDescriptor* HandlerSet::lookupHandler(int socketNum) {
  HandlerDescriptor* handler;
  HandlerIterator iter(*this);
  while ((handler = iter.next()) != NULL) {
    if (handler->socketNum == socketNum) break;
  }
  return handler;
}
```

assignHandler(分配处理程序)方法

通过前面的描述可知，HandlerSet 类都是在操作内部的一个双向链表。但是 HandlerSet 是没有一个 addNode 方法的，这个方法就由 assignHandler 来做了。

assignHandler 的参数有四个，对应了一个节点对象的四个数据成员 socketNum/conditionSet/handlerProc/clientData。前面说过 socketNum 成员在链表中用来标识节点，在这个成员方法中就可以看出来。这个方法会在 BasicTaskScheduler 的 setBackgroundHandling 方法中被用到。其 socketNum 参数应该是传一个 socket 套接口给它。

assignHandler 方法先是从链表中查找 socketNum 标识的节点是否存在，如果不存在就 new 一个，并设置新节点的 socketNum 为参数的 socketNum。这样链表中就存在了一个标识为参数 socketNum 的节点。然后把这个节点的 处理程序指针，客户端数据地址，条件集合都更新为参数中的。

```
void HandlerSet::assignHandler(int socketNum, int conditionSet, TaskScheduler::BackgroundHandlerProc* handlerProc, void* clientData) {
  // First, see if there's already a handler for this socket:
  HandlerDescriptor* handler = lookupHandler(socketNum);
  if (handler == NULL) { // No existing handler, so create a new descr:
    handler = new HandlerDescriptor(fHandlers.fNextHandler);
    handler->socketNum = socketNum;
  }

  handler->conditionSet = conditionSet;
  handler->handlerProc = handlerProc;
  handler->clientData = clientData;
}
```

clearHandler 和 moveHandler 方法

这两个方法比较类似，放在一起来说。

clearHandler 方法是从链表中找 socketNum 标识的节点，然后 delete 这个节点。有之前的描述可以知道，这里把找到的节点从链表中移除了。如果没有找到呢？ lookupHandler 会返回 NULL，delete NULL，是可以的。

```
void HandlerSet::clearHandler(int socketNum) {
  HandlerDescriptor* handler = lookupHandler(socketNum);
  delete handler;
}
```

moveHandler 则是从链表中找 oldSocketNum 标识的节点，找到了就将其标识替换为 newSocketNum。如果没有找到就声明也不做了。这里和前面说的 assignHandler 方法来对比下。assignHandler 是找到了就替换其他的三个数据成员，这里是找到了就替换标识。

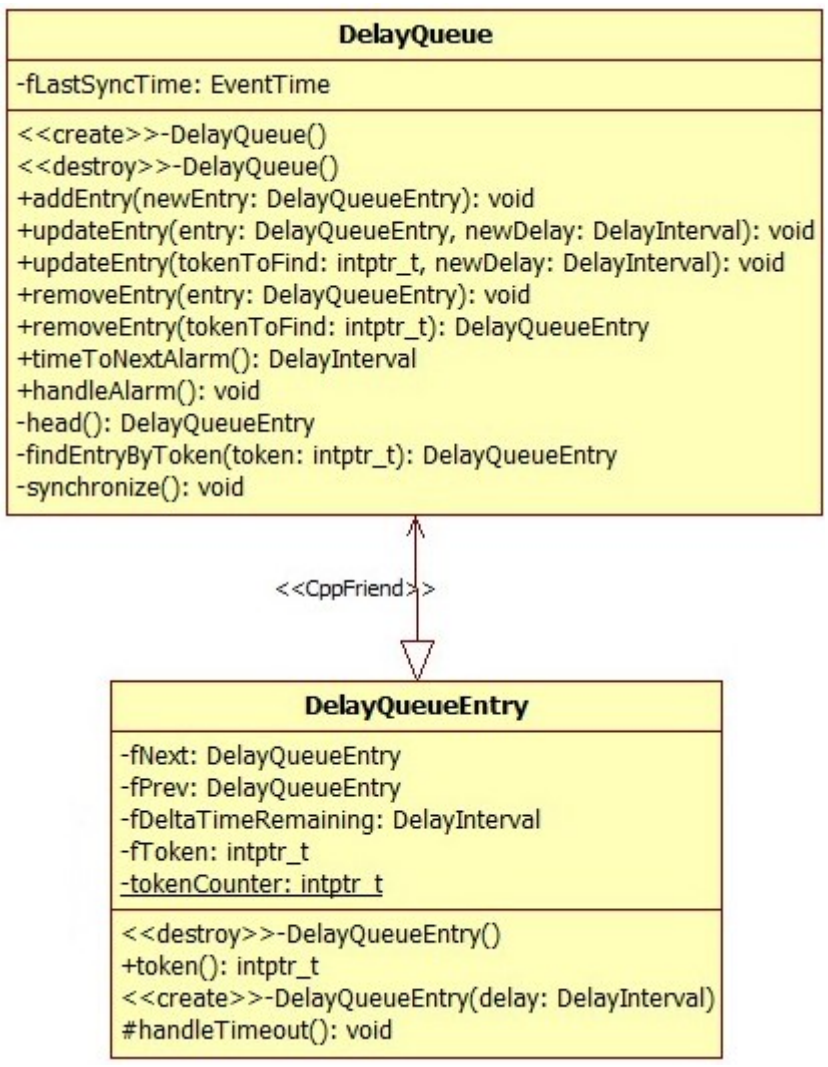
```
void HandlerSet::moveHandler(int oldSocketNum, int newSocketNum) {
  HandlerDescriptor* handler = lookupHandler(oldSocketNum);
  if (handler != NULL) {
    handler->socketNum = newSocketNum;
  }
}
```

3.延时队列相关类

延时队列相关类一共有两个，DelayQueue(延时队列)和 DelayQueueEntry(延时队列节点)。后面说到任务调度器(TaskScheduler)的时候会使用到。

DelayQueue 是 DelayQueueEntry 的派生类，同时也是它的友元类。其定义在 live555sourcecontrol\BasicUsageEnvironment\include\DelayQueue.hh 文件中。

结构关系如下图



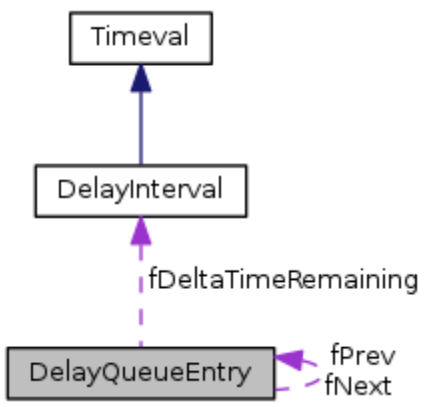
1)DelayQueueEntry 延时队列节点类

entry 的意思如下

entry n.进入，入场；入口处，门口；登记，记录；参加比赛的人；

为什么说是节点类呢？这个通过阅读代码就可以知道了。

DelayQueueEntry 类含有四个数据成员，其中 fNext 和 fPrev 说明了其是一个链表的节点。fToken 是节点的标识，DelayInterval fDeltaTimeRemaining 成员是一个代表时间间隔的量，在后面任务调度器调度任务的时候会使用到。



还有一个静态的成员 `static intptr_t tokenCounter` 用来作为 token 标识的不重复的初始化;注意，静态成员不是对象的成员，而是类的成员。(所有的对象共享这一个)

这里可以看到，其构造函数是 protected 权限的，而析构函数是 public 权限的。且没有了别的构造相关方法，也就是说这个类对象只能由其派生类来创建，但是销毁是对外开放的。其派生类有两个 AlarmHandler 和 DelayQueue。

```
///// DelayQueueEntry /////
// 延时队列记录(节点)   entry n.进入，入场; 入口处，门口; 登记，记录; 参加比赛的人;
class DelayQueueEntry {
public:
  virtual ~DelayQueueEntry();
  intptr_t token() {
    return fToken;
  }
protected: // abstract base class
  DelayQueueEntry(DelayInterval delay);
  // delete this;
  virtual void handleTimeout();
private:
  friend class DelayQueue;
  DelayQueueEntry* fNext;   //下一个节点
  DelayQueueEntry* fPrev;   //上一个节点
  DelayInterval fDeltaTimeRemaining; //延时剩余的时间

  intptr_t fToken;//标识，等指针宽度的 int 型
  static intptr_t tokenCounter;//标识计数(注意此处是 static 变量)
};
```

DelayQueueEntry 的构造

DelayQueueEntry 的构造是很简单的，其只有一个参数，就是延时间隔时间。这里的构造与前面说的 HandlerDescriptor 略有不同，因为它没有把自身加入到链表中，而是把 fNext 和 fPrev 都指向 this。

这里要说的就是 fToken 的初始化赋值，是根据静态成员 tockenCounter 自增来的。这里便保证了在一个指针表示的范围内，fToken 是不会重复的。这里说一下为什么 fToken 的类型是 intptr_t。intptr_t 是一个等指针宽度的 int 型。我们知道指针是用来寻址的，指针的宽度代表了最大的寻址空间。32 位的指针能够寻址的范围是 4G 大小。这里 DelayQueueEntry 对象的大小显然不是 1Byte，就是把内存占满的情况下，fToken 也不会重复。(不可能让它占满)

```
DelayQueueEntry::DelayQueueEntry(DelayInterval delay)
  : fDeltaTimeRemaining(delay) {
  fNext = fPrev = this;
  fToken = ++tokenCounter;
}
```

handleTimeout 方法

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

这个方法异常简单，就是销毁自身。这里要说的是它的方法名，意思很简单，处理超时。顺便说一下，DelayQueueEntry 的析构是空函数，什么也没有做。

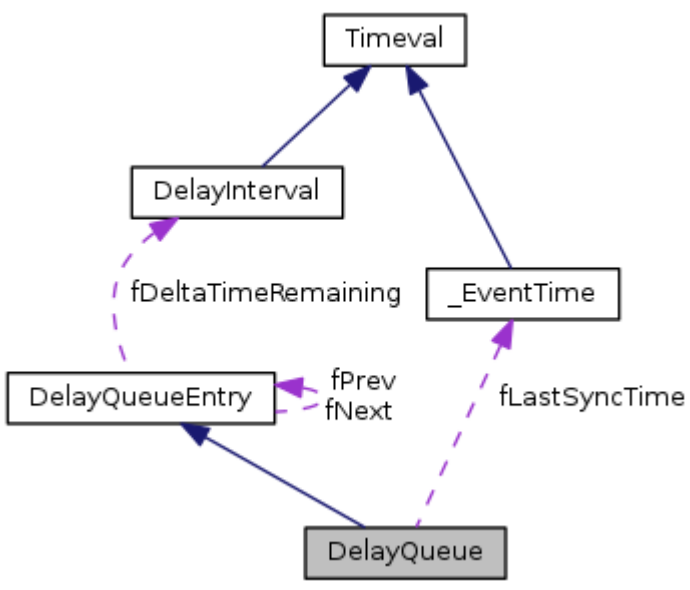
```
void DelayQueueEntry::handleTimeout() {
  delete this;
}
```

2）DelayQueue 延时队列类

这个类的设计不是很复杂，但是要清楚的知道其设计的思路。先给个图

这个链表的设计和前面不一样。其内部只有一个 EventTime fLastSyncTime 最后同步时间的数据成员。并不包含一个链表的头结点。但是其本身是 DelayQueueEntry 的派生类，所以其**本身就是一个链表头结点**。

我们前面说了，DealyQueueEntry 的构造函数是 protected 权限的，而 DelayQueue 是其友元。在后面说还会说到 AlarmHandler 类，这个类对象才是真正的链表节点（头结点除外）。




```
///// DelayQueue /////
// 延时队列(链表)
class DelayQueue: public DelayQueueEntry {
public:
  // 设置头结点的 延时剩余时间 为 永恒
  // 设置最后同步时间为当前时间
  DelayQueue();
  virtual ~DelayQueue();

  //添加记录(节点)
  void addEntry(DelayQueueEntry* newEntry); // returns a token for the entry
```

```
  void updateEntry(DelayQueueEntry* entry, DelayInterval newDelay);
  void updateEntry(intptr_t tokenToFind, DelayInterval newDelay);
  void removeEntry(DelayQueueEntry* entry); // but doesn't delete it
  DelayQueueEntry* removeEntry(intptr_t tokenToFind); // but doesn't delete it

  // 获取头结点的 延时剩余时间
  DelayInterval const& timeToNextAlarm();
  //判断头结点的 延时剩余时间 是否为 DELAY_ZERO 是的话从链表中移除
  // 并由头结点调用 handleTimeout 方法(delete this)
  void handleAlarm();
private:
  DelayQueueEntry* head() { return fNext; }
  DelayQueueEntry* findEntryByToken(intptr_t token);
  //把“剩余时间”域更新。
  //    设置最后同步时间为当前时间
  //    从链表头节点开始，遍历，看节点的延时时间是否到了,到了的设置为 DELAY_ZERO
  //  从这里可以看出来，链表中节点保存的 延时剩余时间 是与前一个节点有关系的
  //   当前节点 总的延时时间，应该是当前节点的 延时剩余时间 加上前一个节点的 总的延时时间
  void synchronize(); // bring the 'time remaining' fields up-to-date
  EventTime fLastSyncTime;  //最后同步时间
};
```

DelayQueue 的构造与析构

构造的时候，将调用了基类的构造。前面说过基类的构造就是初始化了 fDeltaTimeRemaining 成员(延时剩余时间)，并初始化了 fToken 为一个不与其他节点重复的整数，同时将节点的 fNext 和 fPrev 指针指向 this。

这里的参数 ETERNITY 是 const DelayInterval ETERNITY(INT_MAX, MILLION-1); //最大的时间(永恒) 的定义,其可能在不同平台有不同值，但肯定是一个非常大的数。也就是说头结点的延时剩余时间是一个特殊的值，正常情况下不会有比它更大的了。

这里还设置了最后同步时间为当前时间。

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

析构函数做的时间就比较多了，它负责了释放链表的操作。

```
DelayQueue::~DelayQueue() {
  while (fNext != this) {
    DelayQueueEntry* entryToRemove = fNext;
    removeEntry(entryToRemove);
    delete entryToRemove;
  }
}
```

removeEntry 方法

这个方法在析构函数中用到了，就是把节点从链表中移除。要注意的是，其只是把节点移出了链表，并没有销毁哦。

这里注意看这一句 entry->fNext->fDeltaTimeRemaining += entry->fDeltaTimeRemaining;

移除节点的下一个节点的延时间隔剩余时间增加了移除节点的延时剩余时间。这里说明了这个队列的节点的延时间隔剩余时间不是其成员 fDeltaTimeRemaining 所表示的值，而是其与其之前所有节点的 fDeltaTimeRemaining 之和才是真的延时剩余时间。这一点很重要，后面的其他方法中要知道这个设计才行。

```
void DelayQueue::removeEntry(DelayQueueEntry* entry) {
  if (entry == NULL || entry->fNext == NULL) return;

  entry->fNext->fDeltaTimeRemaining += entry->fDeltaTimeRemaining;
```

```
  entry->fPrev->fNext = entry->fNext;
  entry->fNext->fPrev = entry->fPrev;
  entry->fNext = entry->fPrev = NULL;
  // in case we should try to remove it again
}
```

其还有一个重载 `DelayQueueEntry* DelayQueue::removeEntry(intptr_t tokenToFind)` 相当于是先查找，再移除。

findEntryByToken 方法

这个方法用于查找节点，找到了返回节点的地址，没找到返回 NULL。

```
DelayQueueEntry* DelayQueue::findEntryByToken(intptr_t tokenToFind) {
  DelayQueueEntry* cur = head();
  while (cur != this) {
    if (cur->token() == tokenToFind) return cur;
    cur = cur->fNext;
  }
  return NULL;
}
```

synchronize 方法

这是 DelayQueue 中非常重要的一个方法，并且这个方法是 private 权限的，只能在类内部调用。

1、 先获取当前时间，然后比较当前时间与最后一次同步的时间。如果当前时间在最后一次同步时间之后，做下面的步骤

2、 计算出自上次同步之后，又经过了多长时间。时间差为 timeSinceLastSync，设置最后同步时间为当前时间。

3、 从头结点的下一个开始，判断其 延时剩余时间 是否比 已经过去的时间 短，如果是将其延时剩余时间设置为 0 。并将 timeSinceLastSync 减去 这个节点的延时剩余时间，因为实际的延时剩余实际是与前一个节点相关的。

4、 当找到 延时剩余时间比 timeSinceLastSync 长的节点的时候，说明当前节点的延时还得继续，操作到此，将其延时剩余时间减去 timeSinceLastSync。同步至此完成

可以看出这个方法的作用就是判断节点的延时是否到了，进行的一次更新。

```
void DelayQueue::synchronize() {
  // First, figure out how much time has elapsed since the last sync:
  // 首先，计算出自上次同步时间后又过了多少时间：
  EventTime timeNow = TimeNow();
  if (timeNow < fLastSyncTime) {
    // The system clock has apparently gone back in time; reset our sync time and return:
    //系统时钟显然已经回到了过去；重置我们的最后同步时间并返回：
    fLastSyncTime  = timeNow;
    return;
  }
  DelayInterval timeSinceLastSync = timeNow - fLastSyncTime;
  fLastSyncTime = timeNow;

  // Then, adjust the delay queue for any entries whose time is up:
  // 然后，调整延迟队列中的任何项的时间到了：(从链表头节点开始，遍历，看节点的延时时间是否到了)
  DelayQueueEntry* curEntry = head();
  while (timeSinceLastSync >= curEntry->fDeltaTimeRemaining) {
    timeSinceLastSync -= curEntry->fDeltaTimeRemaining;
    curEntry->fDeltaTimeRemaining = DELAY_ZERO;
    curEntry = curEntry->fNext;
  }
  curEntry->fDeltaTimeRemaining -= timeSinceLastSync;
}
```


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

addEntry 方法

addEntry 是添加节点的方法，这个节点必须是已经存在的。我们之前说明，节点的创建是由 AlarmHandler 来完成的。为什么这么肯定呢？因为 DelayQueue 类中没有任何方法创建了 DelayQueueEntry 对象。这里有一个问题就是，如果参数 newEntry 为 NULL 呢？

这里先是更新了一下同步剩余时间，然后在链表中找到合适的位置，插入节点。查找的时候实际上也更新了延时剩余时间。

```
void DelayQueue::addEntry(DelayQueueEntry* newEntry) {
  synchronize();
//这里应该判断一下 newEntry == NULL 的情况
  DelayQueueEntry* cur = head();
  while (newEntry->fDeltaTimeRemaining >= cur->fDeltaTimeRemaining) {
    newEntry->fDeltaTimeRemaining -= cur->fDeltaTimeRemaining;
    cur = cur->fNext;
  }

  cur->fDeltaTimeRemaining -= newEntry->fDeltaTimeRemaining;

  // Add "newEntry" to the queue, just before "cur":
  newEntry->fNext = cur;
  newEntry->fPrev = cur->fPrev;
  cur->fPrev = newEntry->fPrev->fNext = newEntry;
}
```

updateEntry 方法

updateEntry 实现将节点的延时剩余时间更新。先找出节点，然后从链表移出，更新延时剩余时间，再把它添加到链表。

```
void DelayQueue::updateEntry(DelayQueueEntry* entry, DelayInterval newDelay) {
  if (entry == NULL) return;

  removeEntry(entry);
  entry->fDeltaTimeRemaining = newDelay;
  addEntry(entry);
}
```

其还有重载形式 void DelayQueue::updateEntry(intptr_t tokenToFind, DelayInterval newDelay)

timeToNextAlarm 方法

timeToNextAlarm 方法返回第一个节点的延时剩余时间。注意这里说的第一个节点不是头结点哦。

这里判断一下第一个节点延时剩余时间是否为 0 很有必要，如果不为 0 要更新一次。因为当前时间可能不是最后一次同步时间。如果为 0，可以不用更新，提升效率。

```
DelayInterval const& DelayQueue::timeToNextAlarm() {
  if (head()->fDeltaTimeRemaining == DELAY_ZERO) return DELAY_ZERO; // a common case

  synchronize();
  return head()->fDeltaTimeRemaining;
}
```

handleAlarm 方法

这个方法很重要，为什么呢？我们知道每一个节点都是一个 AlarmHandler 对象，这个对象的 handleTimeout 方法做了一件事情，就是使用了一个函数指针调用了一个函数，想一想前面的 HandlerDescriptor 类，是不是处理任务了呢！

本来应先说 AlarmHandler 类的，因为它们不在一个文件中，所以放在后面说。

handleAlarm 方法中将延时等待时间已经到了的(也就是延时剩余时间已经为 0 的)对象从链表中移出，并调用其 handleTimeout 方法去处理任务。

```
void DelayQueue::handleAlarm() {
  if (head()->fDeltaTimeRemaining != DELAY_ZERO) synchronize();

  if (head()->fDeltaTimeRemaining == DELAY_ZERO) {
    // This event is due to be handled:
    // 这事件是由于要处理:
    DelayQueueEntry* toRemove = head();
    removeEntry(toRemove); // do this first, in case handler accesses queue

    toRemove->handleTimeout();
  }
}
```

3) AlarmHandler 定时处理类

这个类定义在 live555sourcecontrol\BasicUsageEnvironment\BasicTaskScheduler0.cpp 文件中。

AlarmHandler 继承自 DelayQueueEntry 其是用来作为 DelayQueued 的节点的。其和 HanlerDescriptor 有点像。其有在 DelayQueueEntry 的基础上又增加了两个数据成员，一个函数指针 TaskFunc* fProc 和一个数据地址 void* fClientData（这个在使用的时候会是调用函数的对象。即函数指针是对象的成员函数地址，数据地址就是对象的地址）。回想一下 DelayQueueEntry 是链表的节点，有前驱和后继指针，延时剩余时间，token 标识。

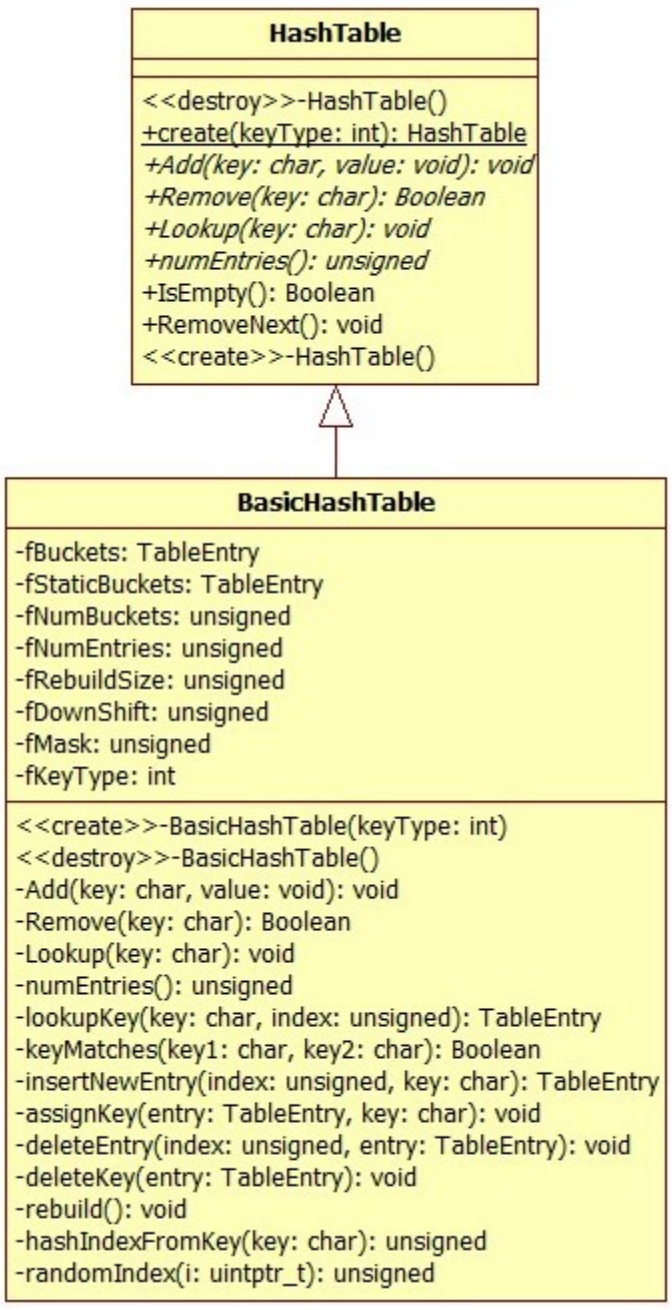
```
////////// A subclass of DelayQueueEntry,
//////////     used to implement BasicTaskScheduler0::scheduleDelayedTask()
class AlarmHandler: public DelayQueueEntry {
public:
  AlarmHandler(TaskFunc* proc, void* clientData, DelayInterval timeToDelay)
    : DelayQueueEntry(timeToDelay), fProc(proc), fClientData(clientData) {
  }

private: // redefined virtual functions
  virtual void handleTimeout() {
    (*fProc)(fClientData);
    DelayQueueEntry::handleTimeout();
  }

private:
  TaskFunc* fProc;
  void* fClientData;
};
```

4.哈希表相关类

哈希表相关类一个有两个 HashTable 和 BasicHashTable。HashTable 是 BasicHashTable 的派生类。其定义在\live555sourcecontrol\UsageEnvironment\include\HashTable.hh 和 BasicHashTable.hh 文件中。

HashTable 是一个抽象类，其没有定义数据成员，仅仅作为接口类存在。

其结构关系如图



1）HashTable 抽象哈希表类

HashTable 类内部嵌套定义了一个迭代器类 Iterator，这个迭代器类用于循环访问表的成员。这也是一个抽象类，但是其有一个静态的方法 static Iterator* create(HashTable& hashTable);这个方法用于创建一个 BasicHashTable::Iterator 对象，并返回其地址。

```
class HashTable {
    public:
  virtual ~HashTable();

  // The following must be implemented by a particular
  // implementation (subclass):
  static HashTable* create(int keyType);

  virtual void* Add(char const* key, void* value) = 0;
     // Returns the old value if different, otherwise 0
  virtual Boolean Remove(char const* key) = 0;
  virtual void* Lookup(char const* key) const = 0;
     // Returns 0 if not found
       virtual unsigned numEntries() const = 0;
       Boolean IsEmpty() const { return numEntries() == 0; }

  // Used to iterate through the members of the table:
class Iterator {
      public:
     // The following must be implemented by a particular
     // implementation (subclass):
     static Iterator* create(HashTable& hashTable);
```

```
    virtual ~Iterator();
    virtual void* next(char const*& key) = 0; // returns 0 if none
protected:
    Iterator(); // abstract base class
  };

      // A shortcut that can be used to successively remove each of
      // the entries in the table (e.g., so that their values can be
      // deleted, if they happen to be pointers to allocated memory).
      void* RemoveNext();
    protected:
  HashTable(); // abstract base class
};
```

迭代器 HashTable::Iterator::create 方法

这个就不说了，代码很简单。要注意的是，其创建的是 BasicHashTable::Iterator 对象。BasicHashTable::Iterator 是 HashTable::Iterator 的派生类。还有，这是一个 static 方法。

```
HashTable::Iterator* HashTable::Iterator::create(HashTable& hashTable) {
  // "hashTable" is assumed to be a BasicHashTable
  return new BasicHashTable::Iterator((BasicHashTable&)hashTable);
}
```

HashTable::create 方法

我们前面说了 HashTable 是一个抽象了，其定义的 create 方法是一个静态方法，实质上是调用的派生类的构造函数来创建的对象并返回。

```
HashTable* HashTable::create(int keyType) {
  return new BasicHashTable(keyType);
}
```

HashTable::RemoveNext 方法

RemoveNext 方法不是纯虚接口。其先创建了一个绑定到自身的迭代器，然后获取迭代器当前指向的节点（因为这里刚创建，所以就是第一个），然后将其从哈希表中移除(并销毁)。

```
void* HashTable::RemoveNext() {
  Iterator* iter = Iterator::create(*this);
  char const* key;
  void* removedValue = iter->next(key);
  if (removedValue != 0) Remove(key);

  delete iter;
  return removedValue;
}
```

2) BasicHashTable 基本哈希表类

这个类搞的很复杂，实质上没什么东西，就是很难看。

先画一个简图

fNumBuckets(SMALL_HASH_TABLE_SIZE)标识桶数

TableEntry** fBuckets; 在对象构造的时候初始化指向fStaticBuckets

TableEntry* fStaticBuckets[SMALL_HASH_TABLE_SIZE];
=4

fNumEntries(0)当前条目数

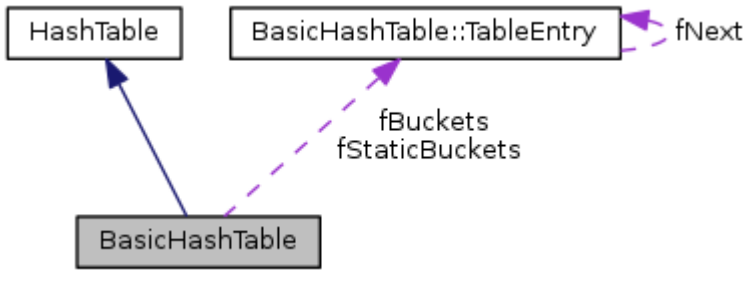
```
class TableEntry {
public:
    TableEntry* fNext;    //下一个指针
    char const* key;      //键
    void* value;          //值
};
```

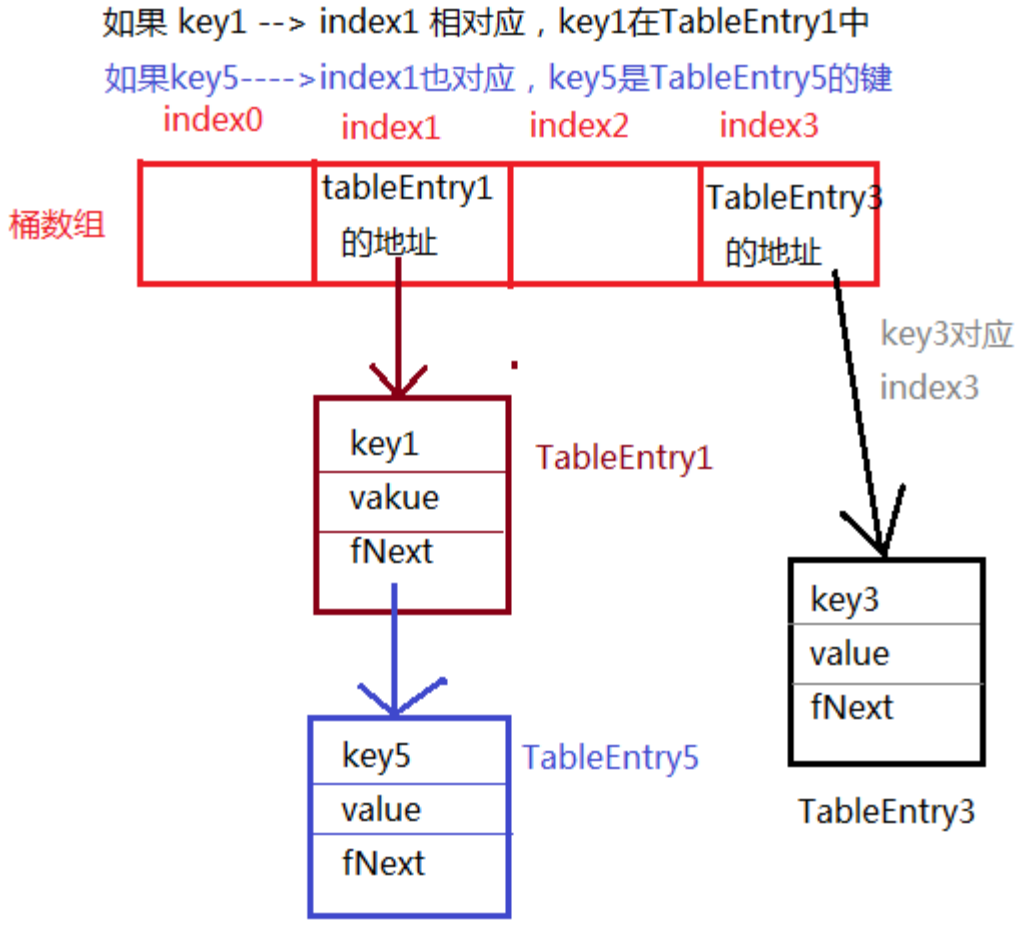
先说说这个 BasicHashTable 的设计吧。

这个表的内部嵌套定义了一个 TableEntry(表条目)的类，这个类有一个键 key，一个值 value，以及指向同一个索引下一个条目的指针(此处很有用，为什么要这么设置，后面会说到的)。这里的 key 是 char const*类型，但其并不一定是如此，可以是 unsigned 类型等，这里只是一个代表。

BasicHashTable 的内部定义了一个 TableEntry*类型的数组 fStaticBuckets，是用来保存表条目的，默认是 4 个元素。还有一个指向这个数组的指针 fBuckets，为什么还要这个数组呢？因为可能哈希表要扩展，4 个元素不够用。所以又定义了一个成员 fNumBuckets 来标识桶(Buckets)的数量，一个桶就是一个数组元素空间。除此之外还需要知道以及保存了多少个条目，于是又有了成员 fNumEntries。还有一个就是条目键的类型，这个在创建哈希表的时候就要确定，所以又增加了成员 fKeyType。

还有一个成员 fRebuildSize，这个成员是用来确定什么时候该重建的。每次在 Add 方法调用的时候就会判断当前已有条目数是否达到了 fRebuildSize，如果达到了就该重建了。它的值是现有桶数的 3 倍。前面说了，一个桶可以保存一个 TableEntry* 类型的变量，也就是一个条目的地址，而每一个 TableEntry 对象中，又含有一个 fNext 变量，指向下一个条目。因为 hash 是散列算法，那么不同的 key 可能会散列到同一个 index，如何解决这种碰撞问题呢？很好办，用链表。即把同一个散列到 index 的条目，用链表串联起来。





另外两个成员 fDownShift, fMask 是用于产生索引 index 用的。

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

下面是类 BasicHashTable 的定义

```
#define SMALL_HASH_TABLE_SIZE 4

class BasicHashTable : public HashTable {
private:
  class TableEntry; // forward

public:
  /**
  *  1、将 fBuckets 指向 fStaticBuckets,初始化其他几个数据成员
  *  2、将 FStaticBuckets 数值清零(全置为 NULL)
  */
  BasicHashTable(int keyType);
  virtual ~BasicHashTable();

//======== class iteratr ====================================
  // Used to iterate through the members of the table:
  class Iterator; friend class Iterator; // to make Sun's C++ compiler happy
  class Iterator : public HashTable::Iterator {
  public:
     //绑定到 table
     Iterator(BasicHashTable& table);

  private: // implementation of inherited pure virtual functions
     //设置 key 为下一个节点的 key,返回下一个节点的 value。如果下一个不存在，返回 NULL
     void* next(char const*& key); // returns 0 if none

  private:
     BasicHashTable& fTable;   //绑定一个哈希表
     unsigned fNextIndex; // index of next bucket to be enumerated after this
     TableEntry* fNextEntry; // next entry in the current bucket
  };
//============================================================

private: // implementation of inherited pure virtual functions
      //继承的纯虚函数的实现

  virtual void* Add(char const* key, void* value);
  // Returns the old value if different, otherwise 0
  // 如果不同的话返回旧值，否则为 0
  virtual Boolean Remove(char const* key);
  virtual void* Lookup(char const* key) const;
  // Returns 0 if not found
  //获取当前条目数
  virtual unsigned numEntries() const;

private:
//======== class TableEntry ==================================
  class TableEntry {
  public:
     TableEntry* fNext; //下一个指针
     char const* key;//键
     void* value;    //值
  };
//============================================================

  //使用 key 来确定 index 和要查找的条目
  TableEntry* lookupKey(char const* key, unsigned& index) const;
```


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

```
  // returns entry matching "key", or NULL if none
  //返回“key”匹配的条目，如果没有找到返回 null

  //比较两个 key 是否一样
  Boolean keyMatches(char const* key1, char const* key2) const;
  // used to implement "lookupKey()"
  // 用于实现 "lookupKey()"

  //创建一个条目，将其放入到桶数组的 index 位置
  TableEntry* insertNewEntry(unsigned index, char const* key);
  // creates a new entry, and inserts it in the table
  // 创建一个新条目，并插入到这个哈希表

  //给一个条目 entry 的 key 成员赋值(绑定一个 key)
  void assignKey(TableEntry* entry, char const* key);
  // used to implement "insertNewEntry()"
  // 用于实现“insertNewEntry”

  //从哈希表中找到 entry，移除后销毁
  void deleteEntry(unsigned index, TableEntry* entry);

  //将条目 entry 的 key 删除
  void deleteKey(TableEntry* entry);
  // used to implement "deleteEntry()"
  // 用于实现 "deleteEntry()"

  //重建哈希表，重建的尺寸是以前的四倍
  void rebuild(); // rebuilds the table as its size increases
  //重建表作为它的尺寸的增加而增加

  //从 key 散列索引,通过 key 来获取一个索引值
  unsigned hashIndexFromKey(char const* key) const;
  // used to implement many of the routines above
  // 用于实现许多以上的程序

  //随机索引，其实并非随机。产生一个与 i 有关的随机值，这是单向不可逆的
  unsigned randomIndex(uintptr_t i) const {
     //1103515245 这个数很有意思，rand 函数线性同余算法中用来溢出的
     //这个函数的作用就是返回一个随机值，因为默认 fMask(0x3)，也就是只保留两位
     //为什么只要保留 2 位，也就是 0 1 2 3 这四种结果咯，因为桶默认只有四个
     return (unsigned)(((i * 1103515245) >> fDownShift) & fMask);
  }

private:
  TableEntry** fBuckets; // pointer to bucket array 指向 桶数组，桶中保存 TableEntry 对象地址
  TableEntry* fStaticBuckets[SMALL_HASH_TABLE_SIZE];// used for small tables 用于小表
  unsigned fNumBuckets/*桶数*/, fNumEntries/*节点数*/, fRebuildSize/*重建尺寸大小*/,
     fDownShift/*降档变速*/, fMask/*掩码*/;
  int fKeyType;
};
```

迭代器 BasicHashTable::Iterator

这里把迭代器先分析了，这个迭代器是继承自 HashTable::Iterator 的。

通过看代码可知，其被 class BasciHashTable 声明为了友元类。

成员 fTable 是一个 BasicHashTable 对象的引用，所以其构造的时候必须绑定一个 BasicHashTable 对象。成员 fNextEntry 和 fNextIndex 是用来查找 TableEntry 条目用的，这个在 next 方法中会详细介绍。

```
  class Iterator; friend class Iterator; // to make Sun's C++ compiler happy
  class Iterator : public HashTable::Iterator {
  public:
     //绑定到 table
     Iterator(BasicHashTable& table);

  private: // implementation of inherited pure virtual functions
     //设置 key 为下一个节点的 key,返回下一个节点的 value。如果下一个不存在，返回 NULL
     void* next(char const*& key); // returns 0 if none

  private:
     BasicHashTable& fTable;   //绑定一个哈希表
     unsigned fNextIndex; // index of next bucket to be enumerated after this
     TableEntry* fNextEntry; // next entry in the current bucket
  };
```

BasicHashTable::Iterator 的构造和析构

BasicHashTable::Iterator 在构造的时候，将 fNextIndex 置为了 0，而 fNextEntry 置为 NULL。仅仅是绑定了一个 table。就是迭代器构建的时候，并没有指向一个条目。

那么它的析构呢？答案是没有。BasicHashTable::Iterator 并没有定义析构函数，其使用默认的析构。为什么呢？因为 BasicHashTable::Iterator 迭代器对象只会指向已经存在的条目，而不会自己创造条目。不存在内存的手动申请释放问题。

```
BasicHashTable::Iterator::Iterator(BasicHashTable& table)
  : fTable(table), fNextIndex(0), fNextEntry(NULL) {
}
```

BasicHashTable::Iterator::next(char const*& key)方法

这个方法就重要了，这是在迭代器里面最重要方法了。它指示了迭代器的工作原理。

前面已经说过了，fNextEntry 指针在构造的时候初始化为 NULL 了，而 fNextIndex 初始化为 0 了。所以这里必须要先找到一个可以用于索引的条目。于是先从桶数组 fBuckets 的第一个桶开始找，遍历桶，直到找到一个有指向条目的桶，如果找遍了都没有找到，那就直接返回 NULL 咯。

这里要注意的是 fNextIndex 的重要性，它保证了不会使得迭代器只用于一个桶所指向的链表，而是在走到链表尾部之后，会走向下一个可用桶去。

如果迭代器不是刚刚构造的，或还没有走到链表的尾部。已经使用过了，那么 fNextEntry 不为 NULL,就可以直接做后续的步骤了。

如果迭代器指向了一个可用的 TableEntry，那么就设置参数 key(注意参数类型，是一个指针的引用)指向这个条目的 key，同时返回这个条目的 value。同时必须将 fNextEntry 指向下一个条目。

```
void* BasicHashTable::Iterator::next(char const*& key) {
  while (fNextEntry == NULL) {
    //如果下一个索引值大于哈希表的桶数，返回 NULL
    if (fNextIndex >= fTable.fNumBuckets) return NULL;
  //fNextEntry 指向对应的桶位置，fNextEntry 后移
    fNextEntry = fTable.fBuckets[fNextIndex++];
  }

  BasicHashTable::TableEntry* entry = fNextEntry;
  fNextEntry = entry->fNext;

  key = entry->key;   //设置 key
  return entry->value;   //返回值
}
```

BasicHashTable 的构造

BasicHashTable 的构造过程很简单，但是要注意的是其参数 keyType，这说明了这个 BasicHashTable 中保存条目的 key 的类型是一致的。在 HashTable.hh 文件中定义了两个 const 量，如果不是这两个定义的，那么 key 会当作 unsigned int*类型，keyType 的值代表 key 指向的内存空间元素个数。

```
int const STRING_HASH_KEYS = 0;      //字符串型key
int const ONE_WORD_HASH_KEYS = 1;    //这个直接当作 char*变量，实质是作为整数在用
```

还有 SMALL_HASH_TABLE_SIZE 这个值，这是一个宏定义，其是 4 。另一个是 REBUILD_MULTIPLIER(重建乘数)其是 3 。这两个值确定了初始的时候桶 bucket 的个数，即桶数组 fBuckets 的大小。一个确定重新建桶数组的临界条件。在每次条目数到了桶数的 3 倍的时候就会重建桶，重建的桶数(fNumBuckets)是之前的 4 倍。

fDownShift 和 fMask 这两个值是用于将 key 散列到 index 时候用的。fDownShift 在每次重建桶的时候会减 2，所以 14 次重建桶其就为 0 了。但是基本不会有这么多次重建，7 次重建之后，桶的数目就达到了 4^8=65536 个。

初始化的时候，每个桶都被初始化为了 NULL，这时候哈希表中还没有条目。

```
BasicHashTable::BasicHashTable(int keyType)
  : fBuckets(fStaticBuckets), fNumBuckets(SMALL_HASH_TABLE_SIZE),
    fNumEntries(0), fRebuildSize(SMALL_HASH_TABLE_SIZE*REBUILD_MULTIPLIER),
    fDownShift(28), fMask(0x3), fKeyType(keyType) {
  for (unsigned i = 0; i < SMALL_HASH_TABLE_SIZE; ++i) {
    fStaticBuckets[i] = NULL;
  }
}
```

BasicHashTable 的析构

BasicHashTable 的析构中用到了 deleteEntry 方法，后面会详细的说这个方法，这里简要的说一下其作用。这个方法将第二个参数 entry 指向的条目从哈希表中移除，并将其 delete。

BasicHashTable 的析构会释放整个哈希表。逐个桶释放逐个条目链表。

```
BasicHashTable::~BasicHashTable() {
  // Free all the entries in the table:
  for (unsigned i = 0; i < fNumBuckets; ++i) {
    TableEntry* entry;
    while ((entry = fBuckets[i]) != NULL) {
      deleteEntry(i, entry);
    }
  }
```

BasicHashTable 的辅助方法

randomIndex(uintptr_t i) const 方法

randomIndex(uintptr_t i) const 方法是这个 BasicHashTable 的散列算法。它将各个 TableEntry 对象的 key 散列到不同的桶中去，就是获取 key 对应桶的索引。

下面代码中的注释已经比较清楚了，稍微再解释一下。

我们之前已经说过了，一个桶可能会有多个条目相关联。哈希表的特点就是要快速查找，那么一个 key 对应到一个桶，然后从桶中查找条目就加快了速度。这个 i * 1103515245 就是一个用来产生伪随机数的操作。这个函数在 key 类型不为字符串的时候会使用到。因为是伪随机数，所以只要 i 不变，结果就不变。在使用的时候会使用 key 来替代 i 。所以只要 key 相同，会散列到同一个桶中。

fDownShift 是用来降档移位的。如果 i=123,那么 i * 1103515245 的结果是 135732375135 转为二进制就是

1001 1010 0100 0111 1010 1110 0101 1111　　这里只看 32 位，溢出的部分不要了，然后将其左移 fDownShift =28 位

0000 0000 0000 0000 0000 0000 0000 1001　　这也一来就只剩下了最后四位有效了，其余的都被清零了。然后&fMask=0x3

0000 0000 0000 0000 0000 0000 0000 0001　　因为 0x3 的二进制形式为 0011（前面 28 位皆为 0），所以相当于就是清零了前面的 30 位，只留下最低两位。之前说过默认的桶数是 4 个，而两个二进制位能表示的是 0、1、2、3 四种可能，刚刚好。

如果重建桶，那么 fDownShift 和 fMask 的值必须有相应的改变。这个在后面会介绍。这里补充一下，上面所做的二进制表示都是在小端序的情况下的。

```
//随机化索引，其实并非随机。产生一个与 i 有关的随机值，这是单向不可逆的
  unsigned randomIndex(uintptr_t i) const {
     //1103515245 这个数很有意思，rand 函数线性同余算法中用来溢出的
     //这个函数的作用就是返回一个随机值，因为默认 fMask(0x3)，也就是只保留两位
     //为什么只要保留 2 位，也就是 0 1 2 3 这四种结果咯，因为桶默认只有四个
     // fDownShift 用来移位，其默认是 28，每次调整哈希表大小的时候会减 2
     return (unsigned)(((i * 1103515245) >> fDownShift) & fMask);
  }
```

hashIndexFromKey(char const* key) const 方法

hashIndexFromKey 方法将一个 key 值散列得到一个 index 索引值。

这里不多解释了，代码很明白。根据 key 的类型，进行不同方式的散列，得到 index 索引返回。这个方法是用于实现后面将介绍的多个方法的关键。

```
unsigned BasicHashTable::hashIndexFromKey(char const* key) const {
  unsigned result = 0;
  //如果键的类型是字符串
  if (fKeyType == STRING_HASH_KEYS) {
    while (1) {
      char c = *key++;    //从 key 指向字符串一个一个取字符
      if (c == 0) break;
    //转换为数字型的索引
      result += (result<<3) + (unsigned)c;
    }
    result &= fMask;  //掩码留位操作
  } else if (fKeyType == ONE_WORD_HASH_KEYS) {
    result = randomIndex((uintptr_t)key);
  } else {
    unsigned* k = (unsigned*)key;
    uintptr_t sum = 0;
    for (int i = 0; i < fKeyType; ++i) {
      sum += k[i];
    }
    result = randomIndex(sum);
  }
  return result;
}
```

rebuild 重新建表方法

重新建表的时候，会动态申请一个是原本桶数组大小四倍的新桶数组。然后将原哈希表中的条目重新散列到新表，因为新表的大小与原本的不同了，fDownShift 和 fMask 必须改变来适应新的桶数组，因此必须重新散列。

每一次重建表的时候，因为桶的个数是原来的四倍，所以散列的时候保留的位数必须得到相应的改变。我们可以计算得到，每次保留的位数必须是在原来基础上增加两位，因为 2 位可以表示 4 种可能，就是总的表示范围扩大了 4 倍。所以 fDownShift 每次减去 2，而 fMask 每次右移两位然后按位或 0x3 操作。

这是一个 private 权限的方法，在 Add 方法中调用。

```
void BasicHashTable::rebuild() {
  // Remember the existing table size:
  unsigned oldSize = fNumBuckets;
  TableEntry** oldBuckets = fBuckets;

  // Create the new sized table:
  fNumBuckets *= 4;
  fBuckets = new TableEntry*[fNumBuckets];
  for (unsigned i = 0; i < fNumBuckets; ++i) {
    fBuckets[i] = NULL;
  }
  fRebuildSize *= 4;
  fDownShift -= 2;
  fMask = (fMask<<2)|0x3;
  // Rehash the existing entries into the new table:
  // 重新散列，把现有的条目加入到新表
  for (TableEntry** oldChainPtr = oldBuckets; oldSize > 0;
       --oldSize, ++oldChainPtr) {
    for (TableEntry* hPtr = *oldChainPtr; hPtr != NULL;
   hPtr = *oldChainPtr) {
      *oldChainPtr = hPtr->fNext;

      unsigned index = hashIndexFromKey(hPtr->key);

      hPtr->fNext = fBuckets[index];
      fBuckets[index] = hPtr;
    }
  }
  // Free the old bucket array, if it was dynamically allocated:
  // 释放旧的桶数组，如果它是动态申请的
  if (oldBuckets != fStaticBuckets) delete[] oldBuckets;
}
```

deleteKey 和 deleteEntry 方法

deleteKey 方法将一个条目的 key 删除。这个方法用于实现 deleteEntry 方法。

```
void BasicHashTable::deleteKey(TableEntry* entry) {
  // The way we delete the key depends upon its type:
  if (fKeyType == ONE_WORD_HASH_KEYS) {
    entry->key = NULL;
  } else {
    delete[] (char*)entry->key;
    entry->key = NULL;
  }
}
```

deleteEntry 用于从哈希表中删除一个条目，并且这个条目会被回收。这里有一个参数 index，这个参数指明了这个 entry 从哪一个桶中查找。这个方法的权限是 private 的，所以这里不用担心 index 传错了的情况，这个在调用的时候会根据条目 entry 的 key 来确定的。如果没有找到，那也没有关系，本来就是要从哈希表中移除的，没有就等于是已经移除了。

```
void BasicHashTable::deleteEntry(unsigned index, TableEntry* entry) {
  TableEntry** ep = &fBuckets[index];

  Boolean foundIt = False;
  while (*ep != NULL) {
    if (*ep == entry) {
      foundIt = True;
      *ep = entry->fNext;
      break;
```

```
    }
    ep = &((*ep)->fNext);       //找到了就从 hashTable 中移除
}
  if (!foundIt) { // shouldn't happen
#ifdef DEBUG
    fprintf(stderr, "BasicHashTable[%p]::deleteEntry(%d,%p): internal error - not found (first entry %p
", this, index, entry, fBuckets[index]);
    if (fBuckets[index] != NULL) fprintf(stderr, ", next entry %p", fBuckets[index]->fNext);
    fprintf(stderr, ")\n");
#endif
}
  --fNumEntries;
  deleteKey(entry);
  delete entry;
}
```

assignKey 方法

assignKey 方法用于给条目*entry 的 key 成员分配一个与参数 key 等大小的空间，并拷贝其指向的内容。简单的说就是使用参数 key 给条目 entry 创建 key。这个方法用与实现 insertNewEntry 方法。函数 strDup 在 strDup.cpp 中实现，其作用是分配一个刚好可以保存 key 指向的字符串的空间，然后拷贝 key 指向字符串内容到新的空间，返回新空间地址。

```
void BasicHashTable::assignKey(TableEntry* entry, char const* key) {
  // The way we assign the key depends upon its type:
  if (fKeyType == STRING_HASH_KEYS) {
    entry->key = strDup(key);   //给条目的 key 成员分配空间
  } else if (fKeyType == ONE_WORD_HASH_KEYS) {
    entry->key = key;
  } else if (fKeyType > 0) {
    unsigned* keyFrom = (unsigned*)key;
    unsigned* keyTo = new unsigned[fKeyType];
    for (int i = 0; i < fKeyType; ++i) keyTo[i] = keyFrom[i];

    entry->key = (char const*)keyTo;
  }
}
```

insertNewEntry 方法(插入新条目)

insertNewEntry 方法用于使用 key 创建一个新条目，然后将这个新条目插入到 index 桶的位置。这是链表的头插法。这也是 private 权限的，在调用的时候 index 会根据 key 来生成。注意的是，目前这个条目的 value 未赋值。

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

  ++fNumEntries;
  assignKey(entry, key);

  return entry;
}
```

keyMatches 方法(键比较)

keyMatches 方法用于比较两个 key 是否一致。这个方法用于实现 lookupKey 方法。

```
Boolean BasicHashTable
::keyMatches(char const* key1, char const* key2) const {
```

```
  // The way we check the keys for a match depends upon their type:
  if (fKeyType == STRING_HASH_KEYS) {
    return (strcmp(key1, key2) == 0);
  } else if (fKeyType == ONE_WORD_HASH_KEYS) {
    return (key1 == key2);
  } else {
    unsigned* k1 = (unsigned*)key1;
    unsigned* k2 = (unsigned*)key2;

    for (int i = 0; i < fKeyType; ++i) {
      if (k1[i] != k2[i]) return False; // keys differ
    }
    return True;
  }
}
```

lookupKey 方法(查找条目)

lookupKey 使用 key 来确定 index 和要查找的条目。注意参数 index 是一个引用，是作为传出参数的。如果查找失败，会返回 NULL。

与前面几个一样，这也是 private 权限的。这几个方法主要用于实现 BasicHashTable 从类 HashTable 中继承的几个虚函数。即对哈希表的增删查改。

```
BasicHashTable::TableEntry* BasicHashTable
::lookupKey(char const* key, unsigned& index) const {
  TableEntry* entry;
  index = hashIndexFromKey(key);   //确定在那个桶里
  //因为一个桶代表一个链表，需要判断找到的条目的 key 是否对得上
  for (entry = fBuckets[index]; entry != NULL; entry = entry->fNext) {
    if (keyMatches(key, entry->key)) break;
    }
  return entry;
}
```

Add 方法(增加条目)

Add 方法先从哈希表中查找参数 key 对应的条目是否存在，存在的话替换 value,不存在就创建一个条目，并加入哈希表。如果加入后哈希表的条目过多了，就重建哈希表。其返回值为原来 key 对应条目的 value，如果不存在，返回 NULL。

```
void* BasicHashTable::Add(char const* key, void* value) {
  void* oldValue;
  unsigned index;
  TableEntry* entry = lookupKey(key, index);
  if (entry != NULL) {
    // There's already an item with this key
    oldValue = entry->value;
  } else {
    // There's no existing entry; create a new one:
    entry = insertNewEntry(index, key);
    oldValue = NULL;
  }
entry->value = value;
  // If the table has become too large, rebuild it with more buckets:
 // 如果该表已经变得太大，重建更多的桶给它:
if (fNumEntries >= fRebuildSize) rebuild();
  return oldValue;
}
```


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

Remove 方法(删除条目)

Remove 方法从哈希表中移除 key 对应的条目，并进行回收。如果 key 对应的条目在哈希表中存在，返回 true，不存在返回 false。

```
Boolean BasicHashTable::Remove(char const* key) {
  unsigned index;
  TableEntry* entry = lookupKey(key, index);
  if (entry == NULL) return False; // no such entry

  deleteEntry(index, entry);

  return True;
}
```

Lookup 方法(查找条目)

Lookup 方法用于从哈希表中查找 key 对应的条目，如果存在，返回条目的 value，如果不存在，返回 NULL。

```
void* BasicHashTable::Lookup(char const* key) const {
  unsigned index;
  TableEntry* entry = lookupKey(key, index);
  if (entry == NULL) return NULL; // no such entry

  return entry->value;
}
```

二、任务调度 TaskScheduler

任务调度是 Live555 源码中很重要的部分。前面介绍的基本组件类在这里都用到了。

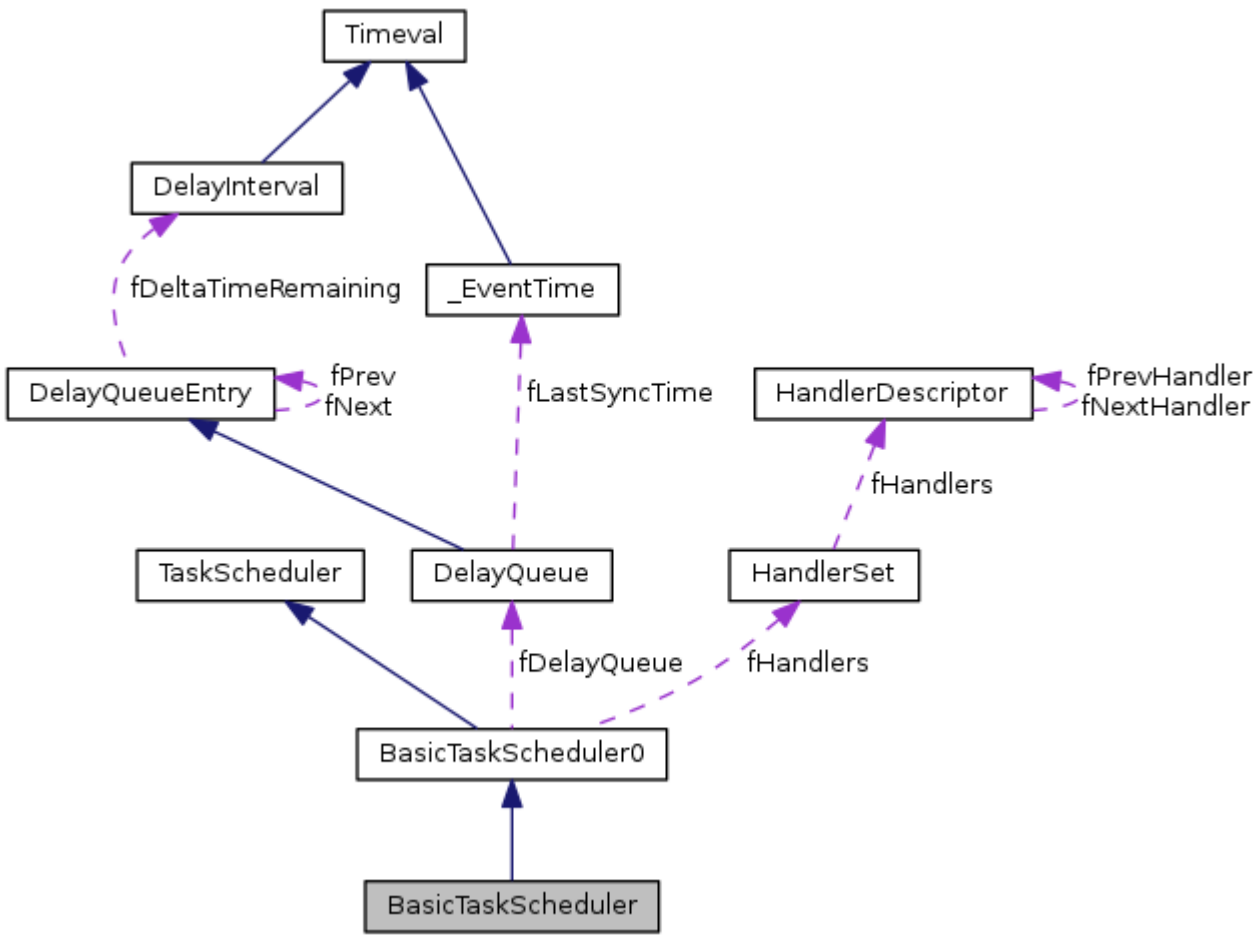
任务调度部分有三个类，其有继承关系。

抽象基类 TaskScheduler 派生出 BasicTaskScheduler0，BasicTaskScheduler0 再派生出 BasicTaskScheduler。

TaskScheduler 主要是一些接口的定义。

BasicTaskScheduler0 主要实现了触发事件的管理。触发事件其主要有三个要素，分别是触发调用函数，数据参数，和等待触发掩码。其中触发调用函数地址保存在函数指针数组中(触发函数类型是：void TaskFunc(void* clientData))，数据参数保存在数据参数指针数组里(void*),等待触发掩码是用于控制其在 doEventLoop 调用的 SingleStep 中是否被触发的标识。

BasicTaskScheduler 是任务调度器的最终成果。它包含了上述两者，并加入了延时队列 DelayQueue 和处理程序链表 HandlerSet 成员。在 SingleStep 中使用了 select 非阻塞 I/O 模型来进行处理 fHandlerSet 链表中的处理程序对象。fHandlerSet 中的链表节点成员是 HandlerDescriptor 对象类型，其有四个重要的成员(socketNum/conditionSet/handlerProc/clientData)，在前面介绍过，这里再提一下。socketNum 用来标识节点，在这里应当会赋予一个网络 socket 套接口给它；handlerProc 是调用的函数的地址，其类型是 TaskScheduler::BackgroundHandlerProc*，而 BackgroundHandlerProc 的类型是 void BackgroundHandlerProc(void* clientData, int mask)，所以其是一个类成员函数指针类型 。clientData 和 conditionSet 都是其参数，其中 conditionSet 是用于 socketNum 的读、写、异常操作的的掩码。



1.TaskScheduler 任务调度器抽象基类

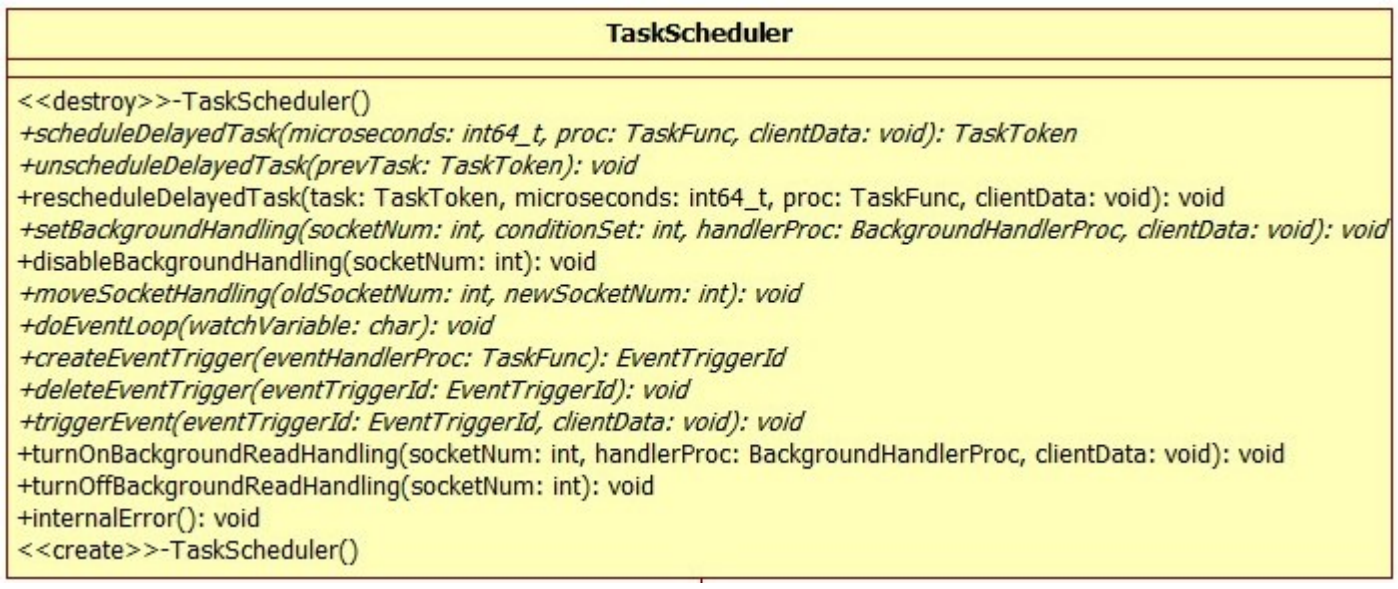
TaskScheduler 是一个抽象基类，其定义在 live555sourcecontrol\UsageEnvironment\include\UsageEnvironment.hh 文件中。

TaskScheduler 声明了很多纯虚接口，其实现一般在 class BasicTaskScheduler0 中。这里简要介绍一下。

TaskScheduler 的默认构造函数是 protected 权限的，也就是只能被其内部的或派生类的方法调用。

这里先列出三个类型定义，这个在后面就不介绍了。在前面 AlarmHandler 中提过。

```
typedef void TaskFunc(void* clientData);
typedef void* TaskToken; //token 标志
typedef u_int32_t EventTriggerId; //Trigger 触发
```



TaskScheduler 的定义如下

```
//任务调度器
class TaskScheduler {
public:
  virtual ~TaskScheduler();

  /* 这是一个纯虚接口，在 BasicTaskScheduler0 中有一个实现*/
  virtual TaskToken scheduleDelayedTask(int64_t microseconds, TaskFunc* proc,
     void* clientData) = 0;
  /* 这是一个纯虚接口，在 BasicTaskScheduler0 中有一个实现*/
  virtual void unscheduleDelayedTask(TaskToken& prevTask) = 0;
  // 没有影响，如果 prevTask == NULL
  // 完成之后将设置 prevTask 为 NULL

  // 虚接口，重新调度延时任务
  // 先调用 unscheduleDelayedTask(task);
  // 在调用 task = scheduleDelayedTask(microseconds, proc, clientData);
  virtual void rescheduleDelayedTask(TaskToken& task,
     int64_t microseconds, TaskFunc* proc,
     void* clientData);

  // For handling socket operations in the background (from the event loop):
  // 后台处理套接字操作类型（从事件循环）：注意，这是一个类型定义
  typedef void BackgroundHandlerProc(void* clientData, int mask);
  // 设置掩码位为 mask,这是特意这样定义的，为了符合 Tcl 接口的一致性
  // Tcl 是“工具控制语言（Tool Control Language）”的缩写。Tk 是 Tcl“图形工具箱”的扩展
  // 它提供各种标准的 GUI 接口项，以利于迅速进行高级应用程序开发

#define SOCKET_READABLE    (1<<1) //readable  adj.易读的;   易懂的;
#define SOCKET_WRITABLE    (1<<2) //writable  adj.可写下的，能写成文的;
#define SOCKET_EXCEPTION   (1<<3) //exception n.例外，除外; 反对，批评;[法律]异议，反对;

  //设置后台处理
  virtual void setBackgroundHandling(int socketNum, int conditionSet, BackgroundHandlerProc* handlerPro
c, void* clientData) = 0;
  //禁用后台处理
  void disableBackgroundHandling(int socketNum) { setBackgroundHandling(socketNum, 0, NULL, NULL); }
  virtual void moveSocketHandling(int oldSocketNum, int newSocketNum) = 0;
  // Changes any socket handling for "oldSocketNum" so that occurs with "newSocketNum" instead.
  // 改变任何套接字操作“oldsocketnum”，发生在“newsocketnum”代替。

  virtual void doEventLoop(char* watchVariable = NULL) = 0;

  //创建一个事件触发器
  virtual EventTriggerId createEventTrigger(TaskFunc* eventHandlerProc) = 0;
```

```
  //删除一个事件触发器
  virtual void deleteEventTrigger(EventTriggerId eventTriggerId) = 0;

  //触发事件
  virtual void triggerEvent(EventTriggerId eventTriggerId, void* clientData = NULL) = 0;
  //以下两个功能是过时的，并提供仅为了向后兼容
  void turnOnBackgroundReadHandling(int socketNum, BackgroundHandlerProc* handlerProc, void* clientDat
a) {
     setBackgroundHandling(socketNum, SOCKET_READABLE, handlerProc, clientData);
  }
  void turnOffBackgroundReadHandling(int socketNum) { disableBackgroundHandling(socketNum); }
  //内部错误
  virtual void internalError(); // used to 'handle' a 'should not occur'-type error condition within th
e library.
protected:
  TaskScheduler(); // abstract base class 抽象基类
};
```

virtual void internalError()方法

因为 TaskScheduler 只实现了两个方法，所以还是说一说了。这个方法调用了库函数 abort()。abrot 函数作用是引发不正常进程的终止。这是用于在发生了内部错误的情况下，不得作出终止当前进程的决定。

在这个函数的实现处，有一行注释，翻译为中文大概意思是：默认情况下，我们处理的不应该发生的错误的类型调用 abort()库函数。子类可以重新定义，如果需要的话。

```
// By default, we handle 'should not occur'-type library errors by calling abort().  Subclasses can rede
fine this, if desired.
void TaskScheduler::internalError() {
  abort();
}
```

rescheduleDelayedTask 重新调度延时任务

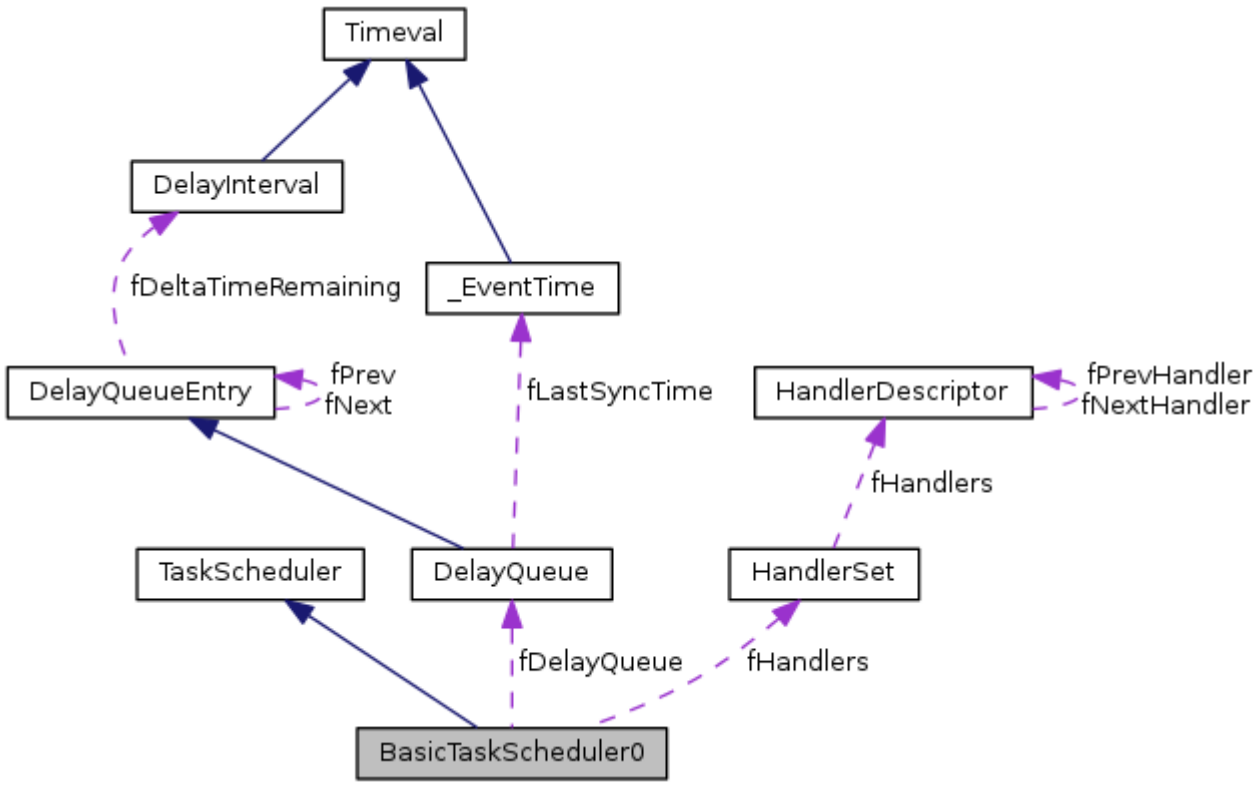
这个方法确实是在 TaskScheduler 中实现的，但是其调用的两个方法都是在其派生类中实现的。这个方法先取消一个任务的调度，然后重新调度这个任务。

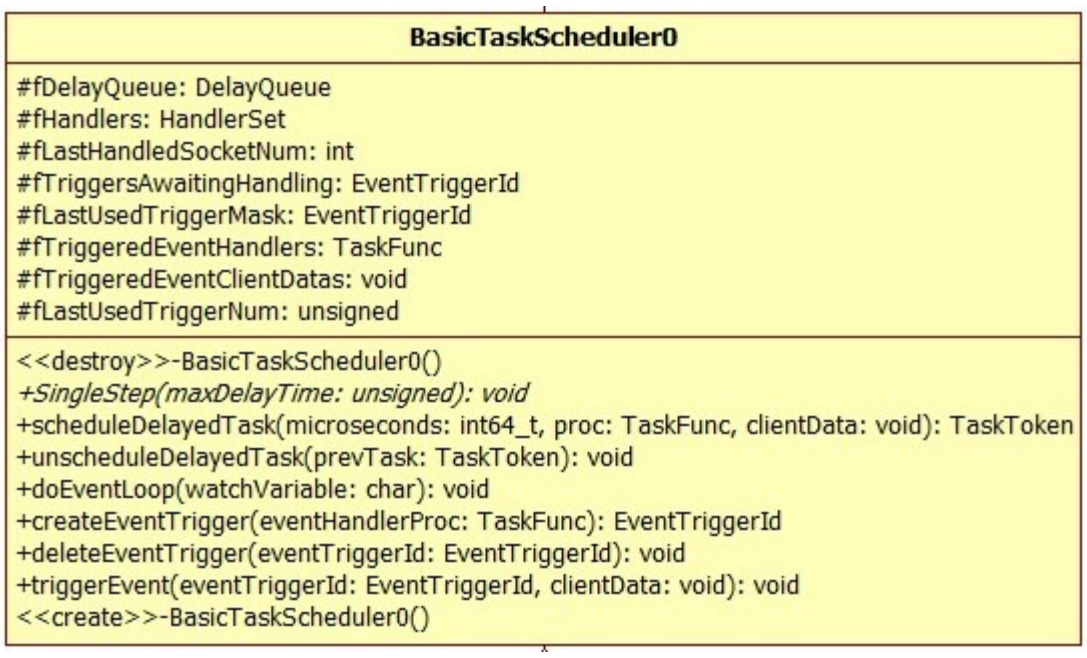
```
void TaskScheduler::rescheduleDelayedTask(TaskToken& task,
                int64_t microseconds, TaskFunc* proc,
                void* clientData) {
  unscheduleDelayedTask(task);
  task = scheduleDelayedTask(microseconds, proc, clientData);
}
```

2.BasicTaskScheduler0 基本任务调度类基类

BasicTaskScheduler0 是一个用作传递的类，它继承自 TaskScheduler，又派生出 BasicTaskScheduler。其定义在 live555 sourcecontrol\UsageEnvironment\include\BasicUsageEnvironment0.hh 文件中。

这个类实现了 TaskScheduler 中的纯虚接口，并增加了一些数据成员。其中比较重要的两个是 fDelayQueue(延时队列)和 fHandlers(处理程序集合/链表)。





下面是其定义代码，里面有一些是对注释的翻译。

```
class HandlerSet; // forward

#define MAX_NUM_EVENT_TRIGGERS 32

// An abstract base class, useful for subclassing 抽象基类，用于子类化
// (e.g., to redefine the implementation of socket event handling)
// 例如，重新定义 socket 事件处理的实现
class BasicTaskScheduler0 : public TaskScheduler {
public:
  //析构的时候 delete fHandlers
  virtual ~BasicTaskScheduler0();

  //设置 select 轮询的超时时间的最大值，如果 maxDelatTime 不大于 0，那么就设置为一百万秒
  virtual void SingleStep(unsigned maxDelayTime = 0) = 0;
  // "maxDelayTime" is in microseconds.  It allows a subclass to impose a limit
  // maxDelayTime 单位是微秒，它允许一个子类施加限制
  // on how long "select()" can delay, in case it wants to also do polling.
  // 多长时间”select()”可以延迟，如果它想做轮询。
  // 0 (the default value) means: There's no maximum; just look at the delay queue
  // 0 作为默认值，意思是：没有最大；只是看看延迟队列

public:
  // Redefined virtual functions:重新定义虚函数
```

```
/* 调度延时任务
* 1、创建一个 AlarmHandler 对象（定时处理）;(new AlarmHandler(proc, clientData, timeToDelay);)
* 2、将创建的 alarmHandler 对象添加到 fDelayQueue 中;(fDelayQueue.addEntry(alarmHandler))
* 3、返回这个 alarmHandler 的 token 标志
*/
virtual TaskToken scheduleDelayedTask(int64_t microseconds, TaskFunc* proc,
   void* clientData);

/* 取消调度延时任务
* 1、从 fDelayeQueue 中 removeEntry 这个 prevTask
* 2、设置 prevTask=NULL
* 3、delete 这个 prevTask 标识的 alarmHandler 对象
*/
virtual void unscheduleDelayedTask(TaskToken& prevTask);

/* 做事件循环
* 1、判断 watchVariable !=0 && *watchVariable != 0 是否成立，若成立，函数返回
* 2、调用函数 SingleStep();函数返回后继续做步骤 1
*/
virtual void doEventLoop(char* watchVariable);

/* 创建事件触发器 ID
*     从 fTriggeredEventHandlers 数组中寻找一个没有使用的位置 pos。如果没有空位，函数返回 0
*     将 eventHandlerProc 放置到上述数组 pos 位置
*     将 fTriggeredEventClientDatas 数组 pos 位置置为 NULL
*     设置 fLastUsedTriggerMask 的第 pos 位为 1
*     设置 fLastUsedTriggerNum 为 pos
*     返回 fLastUsedTriggerMask 的值
*/
virtual EventTriggerId createEventTrigger(TaskFunc* eventHandlerProc);

/* 删除事件触发器 eventTriggerId 可能代表多个事件触发器
*   设置 fTriggersAwaitingHandling &=~ eventTriggerId
*   即将 fTriggersAwaitingHandling 中对应于 eventTriggerId 的非零位 置零
*   从 fTriggeredEventHandlers 和 fTriggeredEventClientDatas 中将对应的位置置为 NULL
*/
virtual void deleteEventTrigger(EventTriggerId eventTriggerId);

/* 触发事件
*    从 fTriggeredEventClientDatas 找到 eventTriggerId 对应的位置，设置为 clientData
*    将 fTriggersAwaitingHandling 中对应 eventTriggerId 中的非 0 位置为 1
*/
virtual void triggerEvent(EventTriggerId eventTriggerId, void* clientData = NULL);

protected:
  BasicTaskScheduler0();

protected:
  // implement vt.实施，执行; 使生效，实现; 落实（政策）; 把…填满;n.工具，器械; 家具; 手段;[法]履行（契约等）;

  // To implement delayed operations: 实施延迟操作:
  DelayQueue fDelayQueue;

  // To implement background reads: 实施后台读
  HandlerSet* fHandlers;    //处理程序描述对象链表指针
  int fLastHandledSocketNum;   //当前最近一个调度的 HandlerDescriptor 对象的 socketNum 标识

  // To implement event triggers: 实施时间触发器
  // fTriggersAwaitingHandling 触发等待处理的 fLastUsedTriggerMask 最后使用触发器的位置置 1
```


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

```
  // implemented as 32-bit bitmaps 实现是 32 位的比特位图
  EventTriggerId fTriggersAwaitingHandling, fLastUsedTriggerMask;

  TaskFunc* fTriggeredEventHandlers[MAX_NUM_EVENT_TRIGGERS];   //保存事件触发器
  void* fTriggeredEventClientDatas[MAX_NUM_EVENT_TRIGGERS]; //保存触发事件客户端数据
  unsigned fLastUsedTriggerNum; // in the range(范围) [0,MAX_NUM_EVENT_TRIGGERS) 最后使用触发器的
};
```

BasicTaskScheduler0 的构造与析构

BasicTaskScheduler0 的构造内容不多，但是从这里开始要介绍的东西很多。还有注意，这个构造函数是 protected 权限的。

先来分析一下它的各个数据成员。

int fLastHandledSocketNum;　　从字面意思来看，这个成员的意思是保存最后一个处理 socket 的。但是这只是猜测。这里我们回到前面去看 HandlerDescriptor 类的定义。其类定义中有一个成员 socketNum，用来表示 HandlerDescriptor 在链表中的唯一存在。是不是这两者就有关联了呢？没错，确实是这样的。这个变量代表的是最后一个被加入的 HandlerDescriptor 对象。

fTriggersAwaitingHandling 和 **fLastUsedTriggerMask** 这两个一起来说。

这两者的类型是 EventTriggerId，实质上是无符号 32 未整型(u_int32_t)。这两个变量和后面的两个数组对应起来看，这两个数组都是 MAX_NUM_EVENT_TRIGGERS 个元素的，也就是 32。而这里两个变量是 32bit，它们的每一个位与数组的一个元素对应是否就刚刚好呢？这里先不说，后面会解释的。(fTriggersAwaitingHandling 等待触发集，fLastUsedTriggerMask 最近使用的触发器)

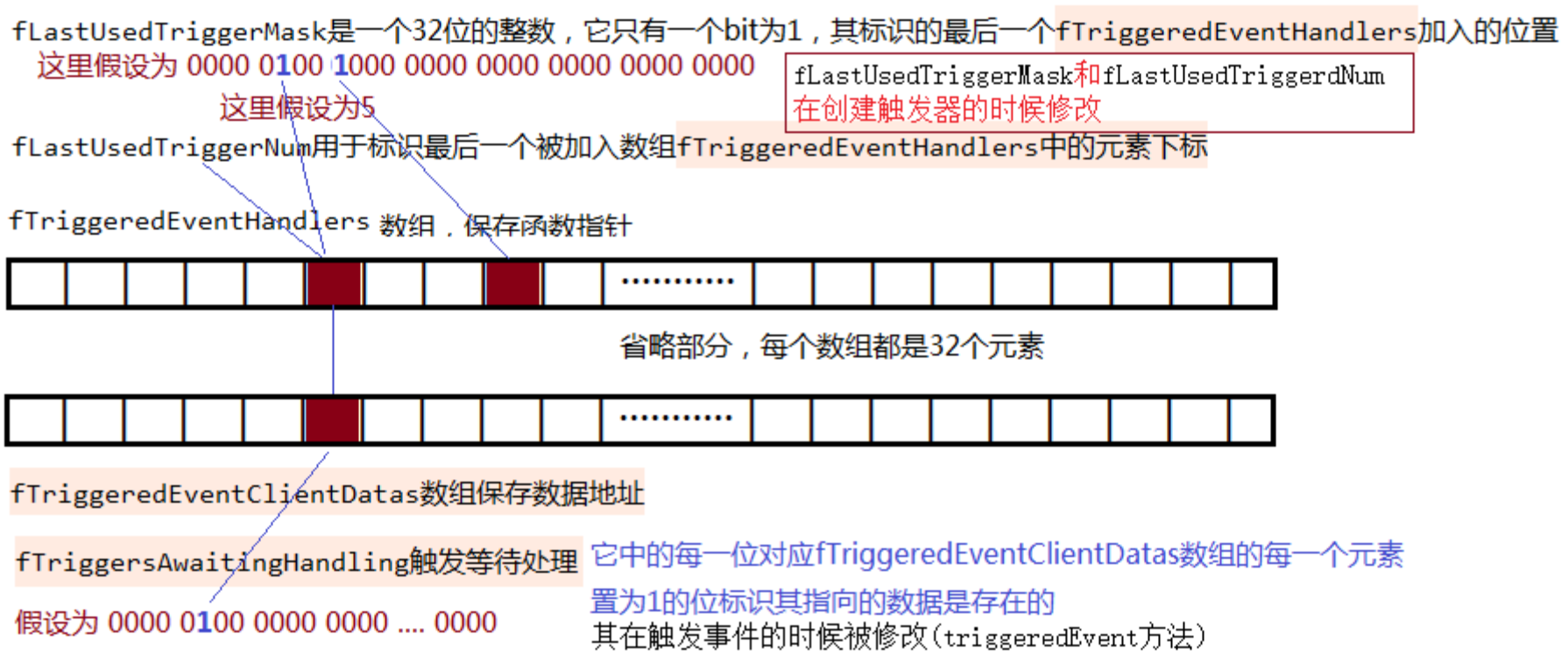
fLastUsedTriggerNum 这个参数表示的是最后一个使用触发器的位置(在 fTriggeredEventHandlers 数组中的下标)。它的范围是[0,31]。将其初始化为 31，是因为每一次创建触发器的时候会使用这个值。见 createEventTriggerd 方法。

数组 fTriggeredEventHandlers 用于保存事件处理程序函数地址，它的每一个元素是一个函数指针。

数组 fTriggeredEventClientDatas 用于保存事件处理程序函数调用时候的参数，它的元素是 void*类型。

DelayQueue fDelayQueue;是一个延时队列，前面介绍过。用于延时处理事件。注意这里这个链表的节点将是 AlarmHandler 对象。

HandlerSet* fHandlers;这一个用于保存事件处理程序和其客户数据。要注意的是这里是一个指针，而不是对象。在构造的时候创建了一个对象。




```
BasicTaskScheduler0::BasicTaskScheduler0()
  : fLastHandledSocketNum(-1), fTriggersAwaitingHandling(0), fLastUsedTriggerMask(1), fLastUsedTriggerN
um(MAX_NUM_EVENT_TRIGGERS-1) {
  fHandlers = new HandlerSet;  //创建对象
  for (unsigned i = 0; i < MAX_NUM_EVENT_TRIGGERS; ++i) {
    fTriggeredEventHandlers[i] = NULL;
    fTriggeredEventClientDatas[i] = NULL;
  }
```

```
}
```

BasicTaskScheduler0 的析构就简单很多了。前面说了，只要一个成员 fHandlers 是指针的，并且在构造的时候动态创建了一个对象给它。所以析构的时候就只是将其 delete 了。

```
BasicTaskScheduler0::~BasicTaskScheduler0() {
  delete fHandlers;
}
```

scheduleDelayedTask 方法(调度延时任务)

scheduleDelayedTask 方法有三个参数，分别是时间 microseconds，任务 proc，数据 clientData。

其使用这三个参数创建一个定时处理程序对象 AlarmHandler(proc, clientData, timeToDelay)，并将这个对象添加到延时队列链表中管理起来。返回了一个这个对象的唯一标识 token。

```
TaskToken BasicTaskScheduler0::scheduleDelayedTask(int64_t microseconds,
                  TaskFunc* proc,
                  void* clientData) {
  if (microseconds < 0) microseconds = 0;
  DelayInterval timeToDelay((long)(microseconds/1000000), (long)(microseconds%1000000));
  AlarmHandler* alarmHandler = new AlarmHandler(proc, clientData, timeToDelay);
  fDelayQueue.addEntry(alarmHandler);

  return (void*)(alarmHandler->token());
}
```

unscheduleDelayedTask 方法(取消调度延时任务)

unscheduleDelayedTask 方法将 pervTask 代表的节点从延时队列中移除，并销毁。参数类型 TaskToKen 实质是一个 void*型。其应该传入的是一个 AlarmHandler 对象的 token 标识。

```
void BasicTaskScheduler0::unscheduleDelayedTask(TaskToken& prevTask) {
  DelayQueueEntry* alarmHandler = fDelayQueue.removeEntry((intptr_t)prevTask);
  prevTask = NULL;
  delete alarmHandler;
}
```

doEventLoop 方法(事件处理循环)

这是一个死循环，在符合条件的时候，会不断调用 SingleStep。这个方法是做一次事件轮询处理。参数 watchVariable 是用来控制是否继续循环的，如果它指向的地址的内容不是'\0'，那么就会跳出死循环，不再继续。

SingleStep 在派生类 BasicTaskScheduler 中实现。

```
void BasicTaskScheduler0::doEventLoop(char* watchVariable) {
  // Repeatedly loop, handling readble sockets and timed events:
  // 反复循环，可读取套接字和定时事件的处理：
  while (1) {
     if (watchVariable != NULL && *watchVariable != 0) break;
     SingleStep();
  }
}
```

createEventTrigger 方法(创建事件触发器)

这个方法将参数 eventHandlerProc 在数组 fTriggeredEventHandlers 还有空位的时候将其添加到数组中，然后返回一个指示其被添加到数组 fTriggeredEventHandlers 中的位置的的变量。

```
EventTriggerId BasicTaskScheduler0::createEventTrigger(TaskFunc* eventHandlerProc) {
  unsigned i = fLastUsedTriggerNum;  //最后使用的触发器在数组的下标
  EventTriggerId mask = fLastUsedTriggerMask;  //bit 位置

  do {
     i = (i + 1) % MAX_NUM_EVENT_TRIGGERS;  //0-31 之间
     mask >>= 1;     //向左移位，与下标同步
     if (mask == 0) mask = 0x80000000;
     //找到一个未使用的位置，就将事件处理程序地址保存到此处
     if (fTriggeredEventHandlers[i] == NULL) {
        // This trigger number is free; use it:
        fTriggeredEventHandlers[i] = eventHandlerProc;
        fTriggeredEventClientDatas[i] = NULL; // sanity
        //更新这两个至
        fLastUsedTriggerMask = mask;
        fLastUsedTriggerNum = i;
        //这个返回值可以求得上面的数组位置值
        return mask;
     }
  } while (i != fLastUsedTriggerNum);

  // All available event triggers are allocated; return 0 instead:
  // 所有可用的事件触发都被分配；而返回 0：
  return 0;
}
```

deleteEventTrigger 方法(删除事件触发器)

deleteEventTrigger 方法将 eventTriggerId 标识的(非 0 位置)触发器从数组 fTriggeredEventHandlers 中移除。

fTriggersAwaitingHandling 在 triggeredEvent 方法中修改了。

```
void BasicTaskScheduler0::deleteEventTrigger(EventTriggerId eventTriggerId) {
  //fTriggersAwaitingHandling 是等待触发处理位标识(标识数组 fTriggeredEventHandlers 已使用)
  //那么这儿很好理解，就是 eventTriggerId 中不为 0 的位，在对应的 fTriggersAwaitingHandling 中清零
  //相当于是标识 fTriggeredEventHandlers 相应的位置已经被释放了。
  fTriggersAwaitingHandling &= ~eventTriggerId;

  if (eventTriggerId == fLastUsedTriggerMask) { // common-case optimization:
     fTriggeredEventHandlers[fLastUsedTriggerNum] = NULL;
     fTriggeredEventClientDatas[fLastUsedTriggerNum] = NULL;
  }
  else {
     // "eventTriggerId" should have just one bit set.其可能是一个集合(要删除多个触发器)
     // However, we do the reasonable thing if the user happened to 'or' together two or more "EventTri
ggerId"s:
     //从将数组对应的位置清零
     EventTriggerId mask = 0x80000000;
     for (unsigned i = 0; i < MAX_NUM_EVENT_TRIGGERS; ++i) {
        if ((eventTriggerId&mask) != 0) {
           fTriggeredEventHandlers[i] = NULL;
           fTriggeredEventClientDatas[i] = NULL;
        }
        mask >>= 1;
     }
  }
}
```

triggerEvent 方法(触发事件)

triggerEvent 并没有真正的触发事件，而是将参数 eventTriggerId 标识的位置，在 fTriggeredEventClientDatas 数组中对应的元素的值改为 clientData。

之前创建触发器的时候，将 fTriggeredEventClientDatas 数组的对应位置是置为 NULL 的。

```
void BasicTaskScheduler0::triggerEvent(EventTriggerId eventTriggerId, void* clientData) {
  // First, record the "clientData":首先，记录“客户端数据”：
  if (eventTriggerId == fLastUsedTriggerMask) { // common-case optimization:
     fTriggeredEventClientDatas[fLastUsedTriggerNum] = clientData;
  }
  else {
     EventTriggerId mask = 0x80000000;
     for (unsigned i = 0; i < MAX_NUM_EVENT_TRIGGERS; ++i) {
        if ((eventTriggerId&mask) != 0) {
           fTriggeredEventClientDatas[i] = clientData;

           fLastUsedTriggerMask = mask;
           fLastUsedTriggerNum = i;
        }
        mask >>= 1;
     }
  }

  // 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 thre
ad, we do this last, to
  //  reduce the risk of a race condition.)
  //  将 fTriggersAwaitingHandling 中对应的位置 1。
  fTriggersAwaitingHandling |= eventTriggerId;
}
```

3.BasicTaskScheduler 基本任务调度器

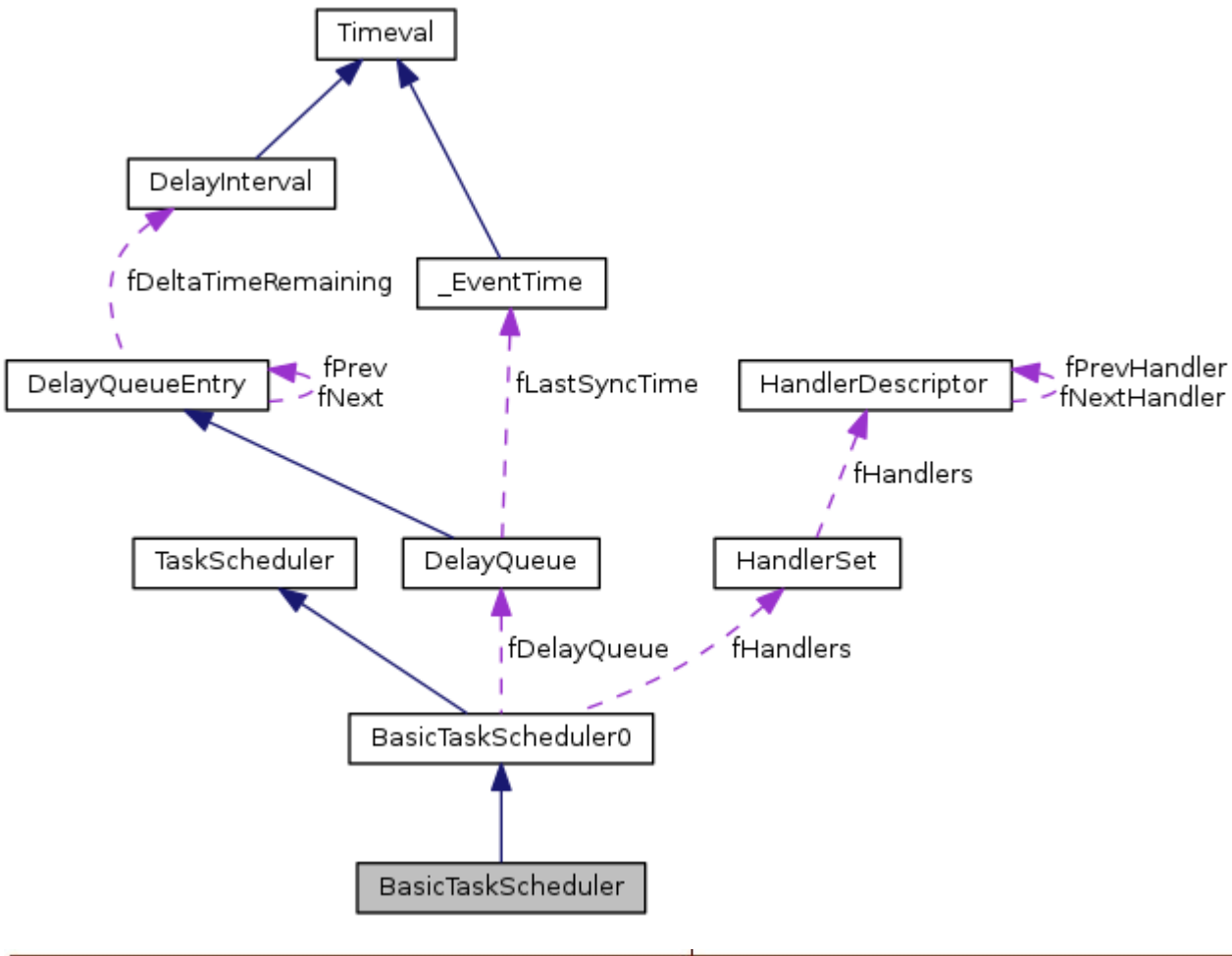
BasicTaskScheduler 很重要了，有了前面的铺垫，这个不会很难。

这个类的重点在于 BasicTaskScheduler::SingleStep 方法的实现。弄懂了这个，基于事件处理模型也就差不多弄懂了。

这里添加了四个数据成员，是用来 select 模型的。关于 select 模型，这里不解释了。在 windows 和 unix/linux 等平台都有相关的 API，实现有点差别，但是原理是一致的。

```
int fMaxNumSockets;     //最大的socket数,select调用时提高效率
fd_set fReadSet;     //监控读操作的集合
fd_set fWriteSet;    //监控写操作的集合
fd_set fExceptionSet;   //监控有异常的集合
```


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



BasicTaskScheduler
#fMaxNumSockets: int #fReadSet: fd_set #fWriteSet: fd_set #fExceptionSet: fd_set
+createNew(): BasicTaskScheduler <<destroy>>-BasicTaskScheduler() <<create>>-BasicTaskScheduler() #SingleStep(maxDelayTime: unsigned): void #setBackgroundHandling(socketNum: int, conditionSet: int, handlerProc: BackgroundHandlerProc, clientData: void): void #moveSocketHandling(oldSocketNum: int, newSocketNum: int): void

```
class BasicTaskScheduler : public BasicTaskScheduler0 {
public:
  static BasicTaskScheduler* createNew();
  virtual ~BasicTaskScheduler();

protected:
  BasicTaskScheduler();
  // called only by "createNew()"

protected:
  // Redefined virtual functions:

  /*
  *    设置 select 的超时时间为 maxDelayTime(<=0 或大于一百万秒 时 1 百万秒)
  *    调用 int selectResult = select(fMaxNumSockets, &readSet, &writeSet, &exceptionSet, &tv_timeToDela
y);
  *    如果 select 出错返回，打印出错信息，并调用 internalError 函数
  *    从处理程序描述链表中查找 fLastHandledSocketNum 代表的 处理程序描述对象指针，如果没找到，就在后面的 while
的时候从链表的头开始,否则从找到的位置开始
  *    从链表中取出处理程序描述节点对象，并调用其内部保存的处理程序
  *    设置 fTriggersAwaitingHandling
  *    调用 fDelayQueue.handleAlarm();
  */
  virtual void SingleStep(unsigned maxDelayTime);
  // 添加到后台处理
  virtual void setBackgroundHandling(int socketNum, int conditionSet, BackgroundHandlerProc* handlerPro
c, void* clientData);
  // 从后台处理移出
```


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

```
  virtual void moveSocketHandling(int oldSocketNum, int newSocketNum);

protected:
  // To implement background operations: 实施后台操作
  int fMaxNumSockets;    //最大的 socket 数,select 调用时提高效率
  fd_set fReadSet;    //监控读操作的集合
  fd_set fWriteSet;      //监控写操作的集合
  fd_set fExceptionSet; //监控有异常的集合
};
```

BasicTaskScheduler 的构造与析构

BasicTaskScheduler 的构造函数是 protected 权限的，其只在静态方法 createNew 中被调用。

创建的时候清零了四个成员，并调用了 schedulerTickTask(this)。

```
BasicTaskScheduler::BasicTaskScheduler()
: fMaxNumSockets(0) {
  FD_ZERO(&fReadSet);
  FD_ZERO(&fWriteSet);
  FD_ZERO(&fExceptionSet);

  schedulerTickTask(this); // ensures that we handle events frequently
}
```

下面来介绍一下 schedulerTickTask 函数(调度滴答任务)

这个函数的作用就是将其参数转为(BasicTaskScheduler*)类型,然后调用 scheduleDelayedTask 来调度(创建)一个延时任务。有意思的是，这个延时任务程序就是这个函数自身，延时任务程序的参数也是其参数。延时时间是 10 毫秒。

这有点像是函数递归调用了。与之不同的是，如果不去调度任务，递归就是无效的。

```
#define MAX_SCHEDULER_GRANULARITY 10000 // 10 microseconds: We will return to the event loop at least th
is often
static void schedulerTickTask(void* clientData) {
  ((BasicTaskScheduler*)clientData)->scheduleDelayedTask(MAX_SCHEDULER_GRANULARITY, schedulerTickTask,
 clientData);
}
```

析构函数是空的，就不说了。只要知道，在析构的时候会调用基类的析构函数。

SingleStep 方法

这是这里最重要的一个方法。每一次调用都是一次真正的处理数据的过程。

前面的延时队列 DelayQueue、处理程序链表 HanlerSet、触发器数组 fTriggeredEventHandlers 和 fTriggeredEventClientDatas 都是在这里被真正的调度起来的。

这一段的代码很长，过程有点多。要联系了前面讲过的内容来看才能比较好理解。这里要注意的 fLastHandledSocketNum 成员的操作。因为其在别的位置都没有修改过，只在这里轮询处理的时候，如果有处理了 fHandlers 中某个节点的时候才会去设置。再一个要思考的是，fHandlers 中的元素是从何而来的？在 BasicTaskScheduler 的两个基类中都没有对 fHandlers 成员有相关的操作。

这个函数做了三件事情。

1、 获取延时队列头结点的延时剩余时间，作为 select 操作的超时时间。调用 select 监控三个集合。

如果 select 调用成功了，那么就开始轮询 HandlerSet 对象 fHandlers 中的节点，有符合条件的就使用其内部保存的函数指针和数据指针以及条件掩码来调用函数。

2、 处理等待触发事件集里面的事件。

3、 处理延时队列中到达延时时间的节点。

```
void BasicTaskScheduler::SingleStep(unsigned maxDelayTime) {
  //拷贝三个集合去给 select 调用做参数
  fd_set readSet = fReadSet; // make a copy for this select() call
  fd_set writeSet = fWriteSet; // ditto
  fd_set exceptionSet = fExceptionSet; // ditto
  //获取延时队列头结点的延时剩余时间(作为 select 超时时间)
  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)
  // 控制在 1 百万秒以内
  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;
  }

  //调用 select 来监控集合
  int selectResult = select(fMaxNumSockets, &readSet, &writeSet, &exceptionSet, &tv_timeToDelay);
//--------------------------------------------------------------------------------
  //select 出错返回，处理错误
  if (selectResult < 0) {
#if defined(__WIN32__) || defined(_WIN32)
     int err = WSAGetLastError();
     // For some unknown reason, select() in Windoze sometimes fails with WSAEINVAL if
     // it was called with no entries set in "readSet".  If this happens, ignore it:
     if (err == WSAEINVAL && readSet.fd_count == 0) {
        err = EINTR;
        // To stop this from happening again, create a dummy socket:
        int dummySocketNum = socket(AF_INET, SOCK_DGRAM, 0);
        FD_SET((unsigned)dummySocketNum, &fReadSet);
     }
     if (err != EINTR) {
#else
     if (errno != EINTR && errno != EAGAIN) {
#endif
        // Unexpected error - treat this as fatal:
#if !defined(_WIN32_WCE)
        perror("BasicTaskScheduler::SingleStep(): select() fails");
#endif
        //内部错误，调用 abort()
        internalError();
     }
  }
//--------------------------------------------------------------------------------
  //开始处理
  // Call the handler function for one readable socket:
  HandlerIterator iter(*fHandlers);
```

```
  HandlerDescriptor* handler;
  // To ensure forward progress through the handlers, begin past the last
  // socket number that we handled:
  //注意 fLastHandledSocketNum 如果不为-1，说明已经调度过某些任务了
  if (fLastHandledSocketNum >= 0) {
     while ((handler = iter.next()) != NULL) {
        //从链表中找上一次最后调度的处理程序描述对象
        if (handler->socketNum == fLastHandledSocketNum) break;
     }
     if (handler == NULL) {
        fLastHandledSocketNum = -1;  //没有找到
        iter.reset(); // start from the beginning instead   迭代器回到起点
     }
  }
  //轮询处理
  //如果上面最后一个 Handle == NULL 成立了，那么这里不会进入，iter.next()还是会返回 NULL
  //也就是说上次最后被调度的对象被找到了，这里的循环才会进入
  //这是为了提高效率，因为找到了最后一个被调度的元素，那么其之前的元素就都已经被调度过了
  while ((handler = iter.next()) != NULL) {
     int sock = handler->socketNum; // alias 别名
     int resultConditionSet = 0;      // 结果条件(状态)集合
     if (FD_ISSET(sock, &readSet) && FD_ISSET(sock, &fReadSet)/*sanity 理智 check*/) resultConditionSet
|= SOCKET_READABLE; //添加可读属性
     if (FD_ISSET(sock, &writeSet) && FD_ISSET(sock, &fWriteSet)/*sanity check*/) resultConditionSet |=
 SOCKET_WRITABLE; //添加可写属性
     if (FD_ISSET(sock, &exceptionSet) && FD_ISSET(sock, &fExceptionSet)/*sanity check*/) resultConditi
onSet |= SOCKET_EXCEPTION;     //添加异常属性
     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.
        //调用相关处理
        (*handler->handlerProc)(handler->clientData, resultConditionSet);
        break;
     }
  }
  //如果没有找到上次最后被调度的对象，并且 fLastHandledSocketNum 标识存在
  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:
     // 我们没有给一个处理程序，但我们没有去检查所有这些，所以试着重新开始：
     iter.reset();   //回到链表头
     //从链表第头开始轮询处理
     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*/) resultConditionSe
t |= SOCKET_WRITABLE;
        if (FD_ISSET(sock, &exceptionSet) && FD_ISSET(sock, &fExceptionSet)/*sanity check*/) resultCon
ditionSet |= SOCKET_EXCEPTION;

        if ((resultConditionSet&handler->conditionSet) != 0 && handler->handlerProc != NULL) {
           //设置 fLastHandledSocketNum 为最后一个被调用的处理程序的标识
           fLastHandledSocketNum = sock;
           // Note: we set "fLastHandledSocketNum" before calling the handler,
           // in case the handler calls "doEventLoop()" reentrantly.
           (*handler->handlerProc)(handler->clientData, resultConditionSet);
           break;
```


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

```
            }
        }
        // 没有一个合适的处理程序被调用
        if (handler == NULL) fLastHandledSocketNum = -1;//because we didn't call a handler
    }
//====================================================================================

    // 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.)
    // 处理等待触发的事件，这个在 fTriggersAwaitingHandling 中被标识
    if (fTriggersAwaitingHandling != 0) {
        if (fTriggersAwaitingHandling == fLastUsedTriggerMask) {
        //只有一个等待触发的事件
            // Common-case optimization for a single event trigger:
            fTriggersAwaitingHandling = 0;
            if (fTriggeredEventHandlers[fLastUsedTriggerNum] != NULL) {
                //函数调用
                (*fTriggeredEventHandlers[fLastUsedTriggerNum])(fTriggeredEventClientDatas[fLastUsedTrigger
Num]);
            }
        }
        else {
            // 有多个等待触发的事件
            // Look for an event trigger that needs handling (making sure that we make forward progress thr
ough all possible triggers):
            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);
        }
    }
//====================================================================================
    // Also handle any delayed event that may have come due.
    // 处理延时队列中已经到时间的延时任务
    fDelayQueue.handleAlarm();
}
```

setBackgroundHandling 方法(添加后台处理程序)

setBackgroundHandling 方法用于添加或更新一个处理程序到 fHandlers 链表。如果 conditionSet 为 0，就将 socketNum 标识的节点从 fHandlers 中移除。否则若 socketNum 标识的节点存在，就更新，否则就添加一个节点。

```
void BasicTaskScheduler
```

```
::setBackgroundHandling(int socketNum, int conditionSet, BackgroundHandlerProc* handlerProc, void* clie
ntData) {
  if (socketNum < 0) return;   //标识不合法
  FD_CLR((unsigned)socketNum, &fReadSet);      //不监控此套接口的可读状态
  FD_CLR((unsigned)socketNum, &fWriteSet); //写
  FD_CLR((unsigned)socketNum, &fExceptionSet);//异常
  if (conditionSet == 0) {  //不监控任何可操作状态
     fHandlers->clearHandler(socketNum);//从链表中移除
     if (socketNum + 1 == fMaxNumSockets) { //最大 socket 数减 1，效率提升
        --fMaxNumSockets;
     }
  }
  else {
     //更新链表，分配处理程序
     fHandlers->assignHandler(socketNum, conditionSet, handlerProc, clientData);
     if (socketNum + 1 > fMaxNumSockets) {
        fMaxNumSockets = socketNum + 1; //更新最大 socket 数
     }
     //设置要监控的状态
     if (conditionSet&SOCKET_READABLE) FD_SET((unsigned)socketNum, &fReadSet);
     if (conditionSet&SOCKET_WRITABLE) FD_SET((unsigned)socketNum, &fWriteSet);
     if (conditionSet&SOCKET_EXCEPTION) FD_SET((unsigned)socketNum, &fExceptionSet);
  }
}
```

moveSocketHandling 方法(转移 socket 处理)

这个方法名不怎么好翻译，有点类似 C++11 move 操作。都是转移操作。这里是将原本对 oldSocketNum 套接口操作的处理程序转移到去操作 newSocketNum 套接口。如果原本 oldSocketNum 就不再链表 fHandler 中呢？那就相当于仅仅把对 oldSocketNum 的监控给移除了。注意，这里设置了对 newSocketNum 的监控，而无论其是否被加入到 fHandler 链表。

```
void BasicTaskScheduler::moveSocketHandling(int oldSocketNum, int newSocketNum) {
  if (oldSocketNum < 0 || newSocketNum < 0) return; // sanity check 完整性检查
  //清理三个集合中对 oldSocketNum 的监控
  if (FD_ISSET(oldSocketNum, &fReadSet)) { FD_CLR((unsigned)oldSocketNum, &fReadSet); FD_SET((unsigned)
newSocketNum, &fReadSet); }
  if (FD_ISSET(oldSocketNum, &fWriteSet)) { FD_CLR((unsigned)oldSocketNum, &fWriteSet); FD_SET((unsigne
d)newSocketNum, &fWriteSet); }
  if (FD_ISSET(oldSocketNum, &fExceptionSet)) { FD_CLR((unsigned)oldSocketNum, &fExceptionSet); FD_SET
((unsigned)newSocketNum, &fExceptionSet); }
  //替换 socketNum
  fHandlers->moveHandler(oldSocketNum, newSocketNum);

  if (oldSocketNum + 1 == fMaxNumSockets) {
     --fMaxNumSockets;
  }
  if (newSocketNum + 1 > fMaxNumSockets) {
     fMaxNumSockets = newSocketNum + 1;
  }
}
```

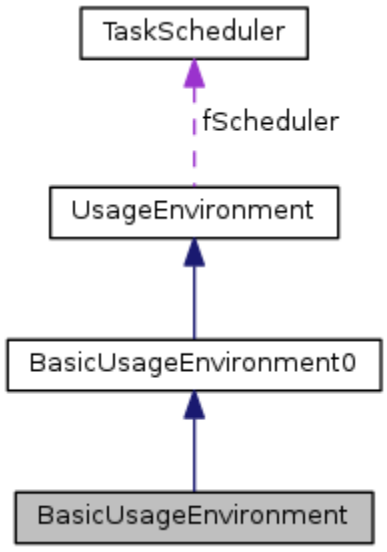
三、使用环境 UsageEnvironment

使用环境相关的类和任务调度的很类似，在 UsageEnvironment 类中有一个数据成员 fScheduler，其是一个 TaskScheduler 的引用。

使用环境相关类也是由三个类组成，其关系如下

UsageEnvrionment → 派生出 →BasicUasgeEnvironment0 → 派生出 → BasicUasgeEnvironment

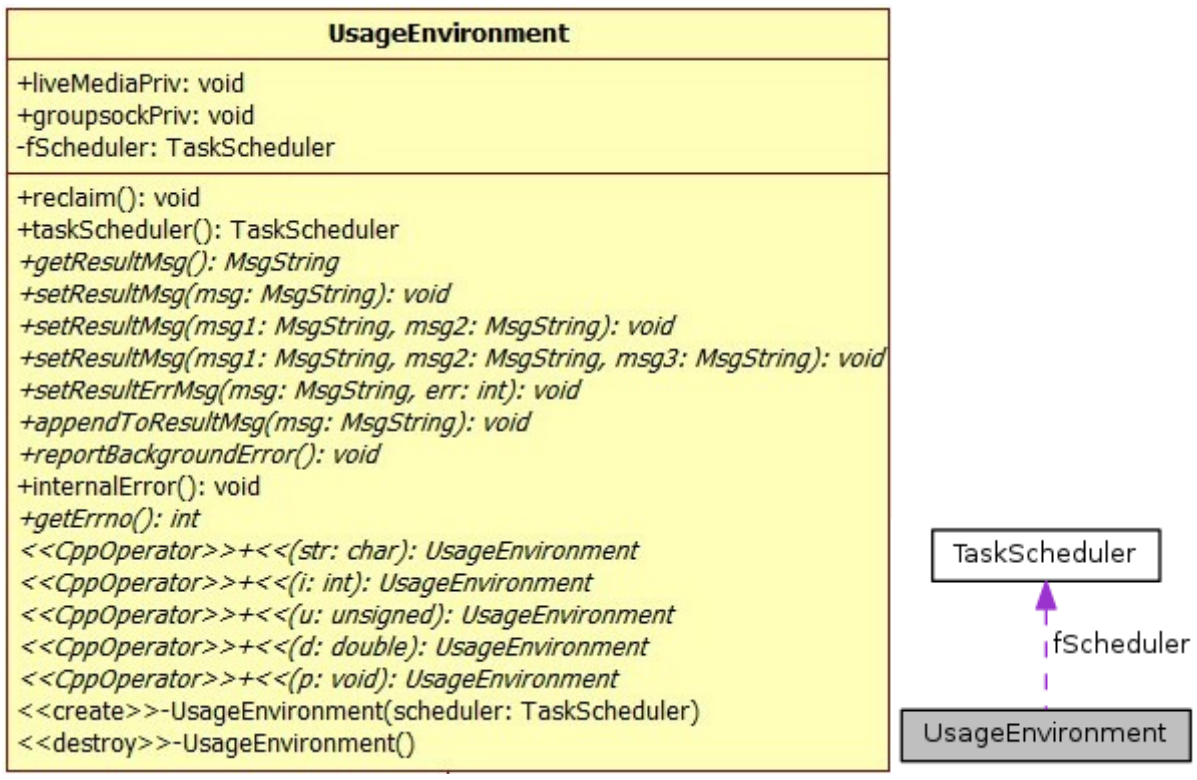
使用环境相关类的作用主要有三个方面，首先 UsageEnvrionment 包含了任务调度器，可以做任务调度相关操作；其次 BasicUasgeEnvironment0 定义了一个 buffer(fResultMsgBuffer)，与处理消息的结果相关；最后 BasicUasgeEnvironment 继承了前两者的功能并添加了向标准错误流输出数据的功能。



1.UsageEnvironment 使用环境抽象基类

UsageEnvironment 是一个抽象基类，其定义在 live555sourcecontrol\UsageEnvironment\include\UsageEnvironment.hh 文件中。

UsageEnvironment 定义了三个数据成员， void*类型的指针 liveMediaPriv 和 groupsockPriv(要注意这两者是 public 权限的，在使用环境相关类中都没有对它们进行初始化以外的操作)，这两个在后面说 Groupsock 和 LiveMedia 模块的时候就比较清楚了。还有一个很重要的是一个引用 fScheduler，它告诉了我们，每一个使用环境必须绑定一个任务调度器。



下面是其定义

```
// An abstract base class, subclassed for each use of the library
// 一个抽象类，子类为每个使用库
class UsageEnvironment {
public:

  //reclaim vt.开拓，开垦；感化；取回；沙化；n.改造，感化；教化；回收再利用；收回，取回；
```


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

```
  //自我回收，如果 liveMediaPriv 或 groupsockPriv 这两个成员变量有一个为 NULL，就 delete this;
  void reclaim();

  // task scheduler:任务调度
  //直接返回对象内部的 fScheduler 成员
  TaskScheduler& taskScheduler() const { return fScheduler; }

  // result message handling:
  //消息处理结果，注意这里是一个类型定义
  typedef char const* MsgString;
  //纯虚接口，看意思应该是获取消息处理结果
  virtual MsgString getResultMsg() const = 0;

  virtual void setResultMsg(MsgString msg) = 0;
  virtual void setResultMsg(MsgString msg1, MsgString msg2) = 0;
  virtual void setResultMsg(MsgString msg1, MsgString msg2, MsgString msg3) = 0;
  virtual void setResultErrMsg(MsgString msg, int err = 0) = 0;
  // like setResultMsg(), except that an 'errno' message is appended.  (If "err == 0", the "getErrno()"
code is used instead.)
  //类似 setResultMsg()，除了“errno”的消息被追加。（如果“err== 0”，“getErrno()”代码是用于替代。）

  virtual void appendToResultMsg(MsgString msg) = 0;

  virtual void reportBackgroundError() = 0;
  // used to report a (previously set) error message within
  //用于报告错误消息（预先设定）内的
  // a background event 事件的背景

  virtual void internalError(); // used to 'handle' a 'should not occur'-type error condition within th
e library.

  // 'errno'
  virtual int getErrno() const = 0;

  // 'console' output:
  virtual UsageEnvironment& operator<<(char const* str) = 0;
  virtual UsageEnvironment& operator<<(int i) = 0;
  virtual UsageEnvironment& operator<<(unsigned u) = 0;
  virtual UsageEnvironment& operator<<(double d) = 0;
  virtual UsageEnvironment& operator<<(void* p) = 0;

  // a pointer to additional, optional, client-specific state
  // 客户端特定的状态
  void* liveMediaPriv;
  void* groupsockPriv;

protected:
  //初始化 liveMediaPriv(NULL), groupsockPriv(NULL), fScheduler(scheduler)
  UsageEnvironment(TaskScheduler& scheduler); // abstract base class
  virtual ~UsageEnvironment(); // we are deleted only by reclaim()我们只有 reclaim()删除

private:
  TaskScheduler& fScheduler;
};
```

UsageEnvironment 的构造与析构

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

其构造的时候需要一个 TaskScheduler 对象来用于绑定，另外两个成员都被初始化为了 NULL。稍带提一下，TaskScheduler 是一个抽象基类，这里绑定的应该是 BasicTaskScheduler 对象，回忆一下 BasicTaskScheduler 的创建是通过静态方法 createNew 获得的。

UsageEnvironment 的构造和析构都受到 protected 权限的保护。

```
UsageEnvironment::UsageEnvironment(TaskScheduler& scheduler)
  : liveMediaPriv(NULL), groupsockPriv(NULL), fScheduler(scheduler) {
}

UsageEnvironment::~UsageEnvironment() {
}
```

reclaim 方法(自我回收)

自我回收是一个 public 接口，可以在外部使用。但是其必须是在(liveMediaPriv == NULL && groupsockPriv == NULL)成立的条件下才会析构自身。

```
void UsageEnvironment::reclaim() {
  // We delete ourselves only if we have no remainining state:
  //我们回收自己，仅当我们有一个删除遗留的状态：
  if (liveMediaPriv == NULL && groupsockPriv == NULL) delete this;
}
```

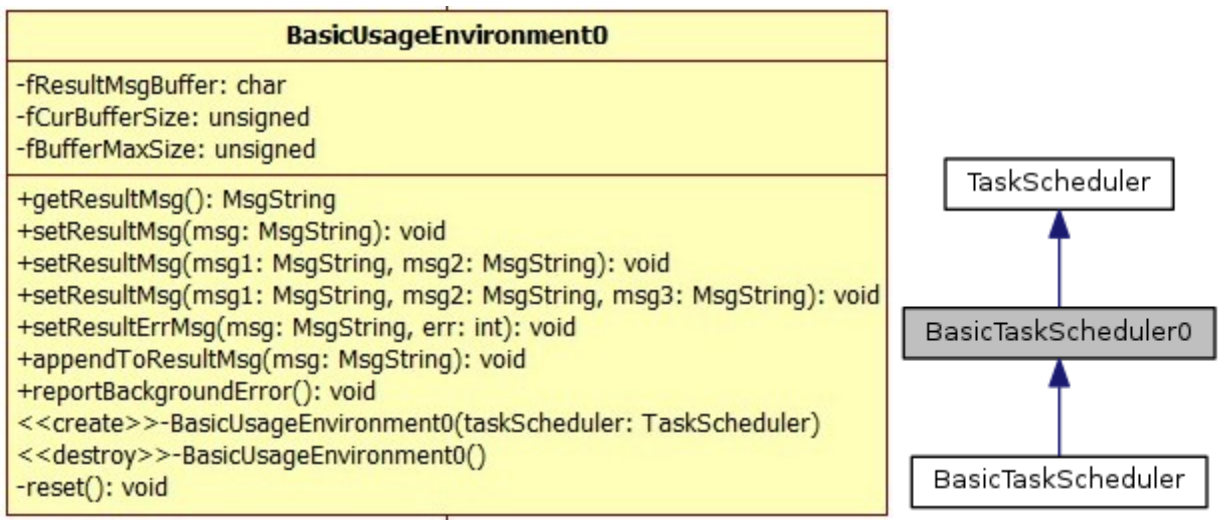
internalError 方法(内部错误)

这个就不用解释了，在 TaskScheduler 中有一个一样的。

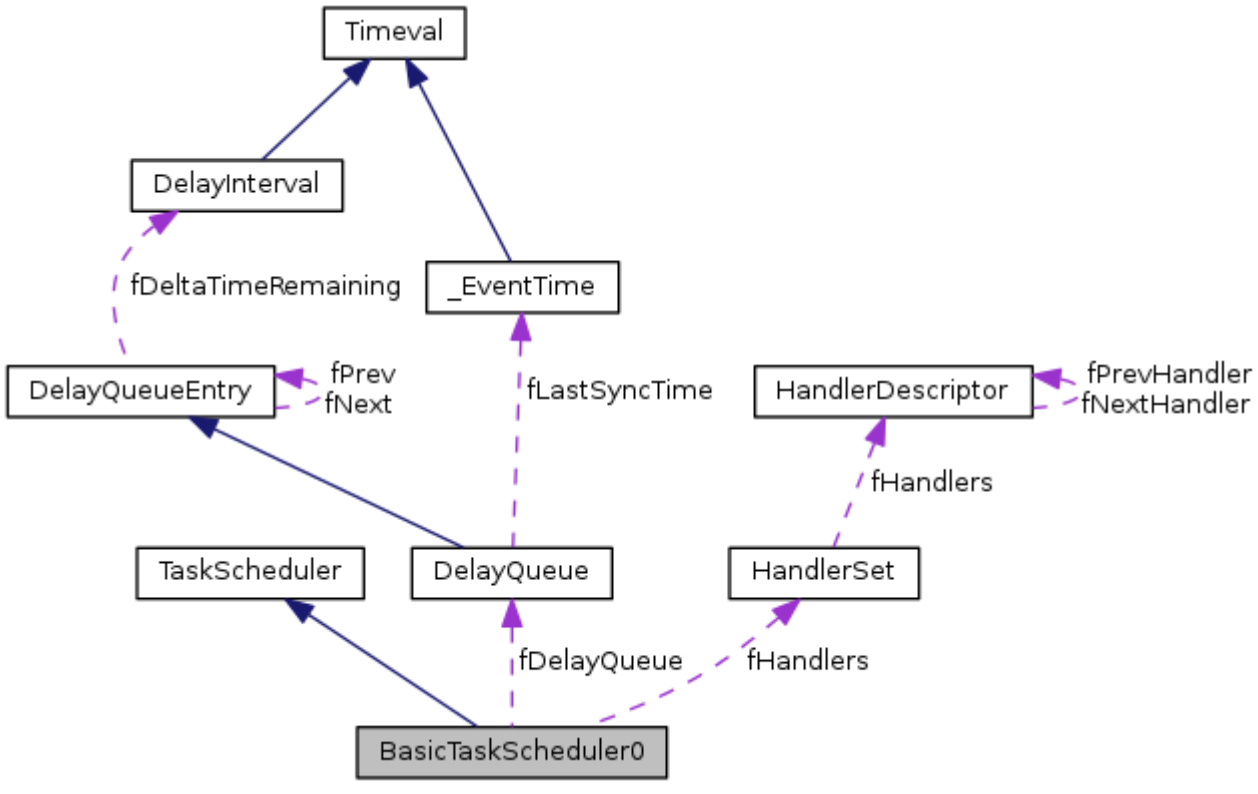
```
// By default, we handle 'should not occur'-type library errors by calling abort().  Subclasses can rede
fine this, if desired.
void UsageEnvironment::internalError() {
  abort();
}
```

2.BasicUsageEnvironment0 基本使用环境基类

BasicUsageEnvironment0 实现了其基类 UsageEnvironment 的部分纯虚接口(只有部分，其还是一个抽象类)，并添加了三个数据成员。其定义在 live555sourcecontrol\BasicUsageEnvironment\include\BasicUsageEnvironment0.hh 文件中。



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



代码定义如下

```
// An abstract base class, useful for subclassing
// (e.g., to redefine the implementation of "operator<<")
class BasicUsageEnvironment0 : public UsageEnvironment {
public:
  // redefined virtual functions:重定义虚函数

  //返回 fResultMsgBuffer
  virtual MsgString getResultMsg() const;
  // 调用 reset 将消息结果 buffer 截空，再将 msg(msg1-3)拷贝到 buffer
  virtual void setResultMsg(MsgString msg);
  virtual void setResultMsg(MsgString msg1,
     MsgString msg2);
  virtual void setResultMsg(MsgString msg1,
     MsgString msg2,
     MsgString msg3);
  //将 msg 设置到 fResultMsgBuffer，支持_WIN32_WCE 的平台会将 err 代表的错误消息也加入
  virtual void setResultErrMsg(MsgString msg, int err = 0);
  //将 msg 拷贝到 fResultMsgBuffer 可用部分，剩余空间不够时，只拷贝部分
  virtual void appendToResultMsg(MsgString msg);
  ////将 fResultMsgBuffer 中的内容写入到标准错误
  virtual void reportBackgroundError();

protected:
  BasicUsageEnvironment0(TaskScheduler& taskScheduler);
  virtual ~BasicUsageEnvironment0();

private:
  void reset();   //截空 buffer 字符串(首元素置'\0')

  //消息处理结果缓冲
  char fResultMsgBuffer[RESULT_MSG_BUFFER_MAX];
  unsigned fCurBufferSize;  //当前 buffer 已用大小
  unsigned fBufferMaxSize;  //最大 buffer 大小
};
```

BasicUsageEnvironment0 构造析构与重置

把这三个放在一起，因为其内容很少。

构造的时候调用了基类 UsageEnvironment 的构造，并把 fBufferMaxSize(buffer 最大尺寸)的值设置为 fResultMsgBuffer 数组的大小(见宏定义#define RESULT_MSG_BUFFER_MAX 1000)并调用 reset 重置 buffer。

reset 方法用于重置 buffer(这里说的 buffer 都代指 fResultMsgBuffer 字符串),其将 fResultMsgBuffer 的首元素置为'\0',也就是将其截空。

```
BasicUsageEnvironment0::BasicUsageEnvironment0(TaskScheduler& taskScheduler)
  : UsageEnvironment(taskScheduler),
    fBufferMaxSize(RESULT_MSG_BUFFER_MAX) {
  reset();
}

BasicUsageEnvironment0::~BasicUsageEnvironment0() {
}

void BasicUsageEnvironment0::reset() {
  fCurBufferSize = 0;
  fResultMsgBuffer[fCurBufferSize] = '\0';
}
```

ResultMsg 系列方法

ResultMsg 系列方法是指一系列对 fResultMsgBuffer 进行操作的方法，包括 get/set/append/report 等多个。这些方法都在基类 UsageEnvironment 中声明，这里对其进行了实现。注意，这些接口都是 public 权限的，理应对参数进行判断。后面介绍的时候会提到一些方法中没有对参数的合法性进行判断。

getResultMsg() const 方法(获取 buffer)

getResultMsg 方法是一个 const 方法，不会对对象有写操作。其返回 fResultMsgBuffer 数组的首地址。fResultMsgBuffer 数组这里再提一下，其是一个 char 类型的数组，从变量名上理解，是用于保存处理消息结果。

```
char const* BasicUsageEnvironment0::getResultMsg() const {
  return fResultMsgBuffer;
}
```

appendToResultMsg 方法(添加 msg 到 buffer)

appendToResultMsg 方法用与向 buffer 中添加内容。参数 msg 是标识一个 char*字符串。注意，这里没有判断 msg 是否为 NULL 是一个 bug。因为 strlen(NULL)以及 memmove(dest,NULL,len)的后果是未定义的。

如果 buffer 中剩余的可用空间容不下 msg 的全部内容，那么会拷贝 msg 中的部分内容，将 buffer 填满。

```
void BasicUsageEnvironment0::appendToResultMsg(MsgString msg) {
  char* curPtr = &fResultMsgBuffer[fCurBufferSize];
  unsigned spaceAvailable = fBufferMaxSize - fCurBufferSize;
  unsigned msgLength = strlen(msg);

  // Copy only enough of "msg" as will fit:
  // fResultMsgBuffer 剩余空间不够放，拷贝一部分
  if (msgLength > spaceAvailable-1) {
    msgLength = spaceAvailable-1;
  }
  /* memmove 用于从 src 拷贝 count 个字符到 dest，如果目标区域和源区域有重叠的话，memmove 能够
     保证源串在被覆盖之前将重叠区域的字节拷贝到目标区域中。但复制后 src 内容会被更改。但是当目标
    区域与源区域没有重叠则和 memcpy 函数功能相同。*/
  memmove(curPtr, (char*)msg, msgLength);
  fCurBufferSize += msgLength;
  fResultMsgBuffer[fCurBufferSize] = '\0';  //这个必须有
```

```
}
```

setResultMsg 方法(重置 buffer 内容为 msg)

setResultMsg 用于重置 buffer 内容。它将其内容重新设置为参数 msg（msg1-3）的内容。

setResultMsg 有多个重载形式，区别在于参数个数不一致。这里提一下，C++的重载就是以参数不同为依据的。在这多个重载中都使用到了 appendToResultMsg 方法，也就继承了没有判断参数合法性的 bug。

```
// 调用 reset 将消息结果 buffer 截空，再将 msg 拷贝到 buffer
void BasicUsageEnvironment0::setResultMsg(MsgString msg) {
  reset();
  appendToResultMsg(msg);
}
```

```
void BasicUsageEnvironment0::setResultMsg(MsgString msg1, MsgString msg2) {
  setResultMsg(msg1);
  appendToResultMsg(msg2);
}
```

```
void BasicUsageEnvironment0::setResultMsg(MsgString msg1, MsgString msg2,
                  MsgString msg3) {
  setResultMsg(msg1, msg2);
  appendToResultMsg(msg3);
}
```

setResultErrMsg 方法(重置 buffer 内容为 msg/err)

setResultErrMsg 方法有两个参数，msg 参数用于重置 buffer 内容。

err 参数在 windows(WIN32/WINCE)平台会使用到，如果 err 为 0，那么会调用 getError()，这个方法在派生类 BasicUsageEnvironment 中实现。在 windows 相关平台其 return WSAGetLastError()也就是该线程进行的上一次 Windows Sockets API 函数调用时的错误代码。如果是其他平台，直接返回 errno。这里说了，在非 windows 平台是不会调用的。如果 err 不为 0 ，会之间调用 strerror(err)获取错误描述字符串添加到 buffer。

```
void BasicUsageEnvironment0::setResultErrMsg(MsgString msg, int err) {
  setResultMsg(msg);

#ifndef _WIN32_WCE
  appendToResultMsg(strerror(err == 0 ? getErrno() : err));
#endif
}
```

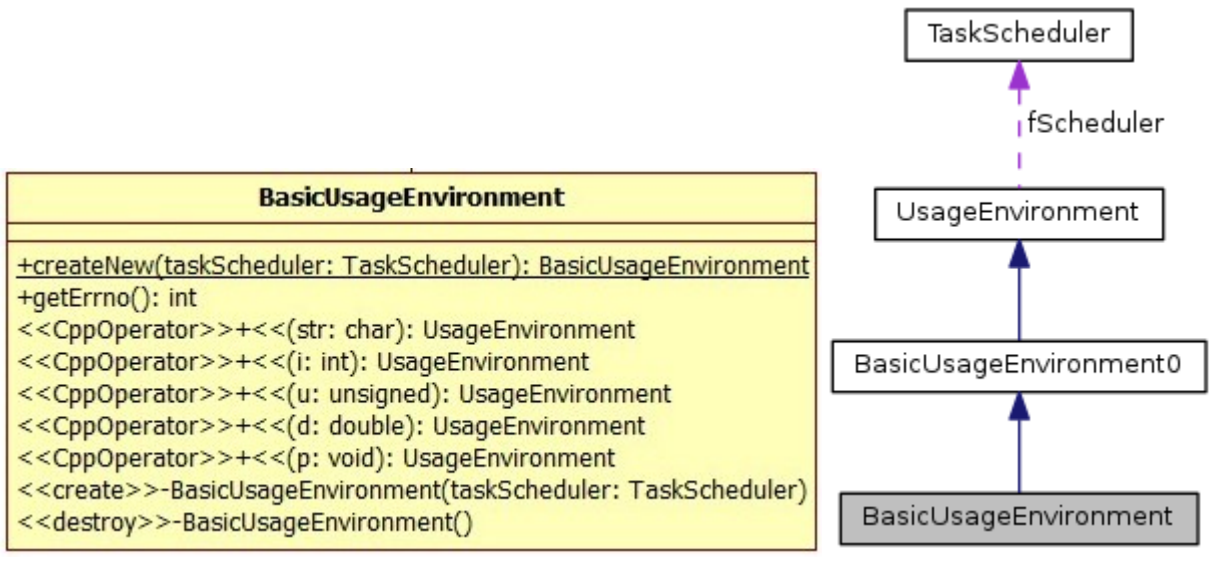
reportBackgroundError 方法(报告错误消息)

reportBackgroundError 将 buffer 中的内容输出到标准错误。这里很简单，要提的一点是，stderr 是无缓冲的的输出流，写入的数据直接送入到内核缓冲区。这是 C 语言的一点基础知识。

```
//将 fResultMsgBuffer 中的内容写入到标准错误
void BasicUsageEnvironment0::reportBackgroundError() {
  fputs(getResultMsg(), stderr);
}
```

3.BasicUsageEnvironment 基本使用环境

这个类很简单，它不想 basicTaskScheduler 那么复杂。没有增加任何成员，仅是实现了基类的几个纯虚方法。



以下是其定义

```
class BasicUsageEnvironment : public BasicUsageEnvironment0 {
public:
  static BasicUsageEnvironment* createNew(TaskScheduler& taskScheduler);

  // redefined virtual functions:
  virtual int getErrno() const;

  // 向 stderr 输出内容。stderr 是不带缓冲的
  virtual UsageEnvironment& operator<<(char const* str);
  virtual UsageEnvironment& operator<<(int i);
  virtual UsageEnvironment& operator<<(unsigned u);
  virtual UsageEnvironment& operator<<(double d);
  virtual UsageEnvironment& operator<<(void* p);

protected:
  // 避免直接构造对象，只能通过 createNew 来创建
  BasicUsageEnvironment(TaskScheduler& taskScheduler);
  // called only by "createNew()" (or subclass constructors)
  virtual ~BasicUsageEnvironment();
};
```

BasicUsageEnvironment 的构造与析构

注意构造和析构是 protected 权限的。在创建对象的时候只能使用 createNew 方法。

BasicUsageEnvironment 的构造函数还是调用了其基类 BasicUsageEnvironment0 的带参构造，要注意的是在 BasicUsageEnvironment0 的构造中又调用了 UsageEnvironment 的带参构造。如果是 win32 平台，其调用了 initializeWinsockIfNecessary 进行来初始化 WinSOCK，之后才可以正常使用 WinSOCK 相关 API。如果不是 windows 平台就不需要这么麻烦了，windows 网络编程是一件麻烦事。

initializeWinsockIfNecessary 函数定义在 live555sourcecontrol\groupsock\inet.c 文件中。注意 C++中对 C 函数不能直接调用，要先使用 extern “C”来声明。原因是 C 和 C++编译器对函数名的处理不一致。

```
#if defined(__WIN32__) || defined(_WIN32)
extern "C" int initializeWinsockIfNecessary();
#endif

BasicUsageEnvironment::BasicUsageEnvironment(TaskScheduler& taskScheduler)
: BasicUsageEnvironment0(taskScheduler) {
#if defined(__WIN32__) || defined(_WIN32)
  if (!initializeWinsockIfNecessary()) {
    setResultErrMsg("Failed to initialize 'winsock': ");
    reportBackgroundError();
```

```
    internalError();
  }
#endif
}
```

BasicUsageEnvironment 的析构还是什么也没有做，但是要注意的是，对象析构的时候会调用基类的析构函数。

顺带再多说一点 C++对象的构造析构过程。C++类定义中，构造函数不能使用 virtual 修饰，而析构函数请尽量使用 virtual 修饰。为什么呢？因为将析构函数加入虚函数表可以使得对象在析构的时候可以正确调用对应的析构函数，避免内存泄露等问题。在构建对象的时候，构造函数的调用顺序是 基类的构造---》派生类的构造，析构顺序与之相反，是 派生类的析构---》基类的析构。

```
BasicUsageEnvironment::~BasicUsageEnvironment() {
}
```

createNew 方法(创建对象)

在堆上(heap)动态创建一个 BasicUsageEnvironment 对象并返回对象地址。这是一个静态方法。

```
BasicUsageEnvironment*
BasicUsageEnvironment::createNew(TaskScheduler& taskScheduler) {
  return new BasicUsageEnvironment(taskScheduler);
}
```

getErrno 方法

这个方法在 BasicUsageEnvironment0 中实现 setResultErrMsg 的时候用到了。其返回一个错误码，在 windows 相关平台是上一次发生网络错误的错误代码，其他平台是全局的 errno。考虑一下 errno 的线程安全性。

```
int BasicUsageEnvironment::getErrno() const {
#if defined(__WIN32__) || defined(_WIN32) || defined(_WIN32_WCE)
  return WSAGetLastError();
  /* #include <winsock.h>
int PASCAL FAR WSAGetLastError ( void );
注释:该函数返回上次发生的网络错误.当一特定的 Windows Sockets API 函数指出一个错误已经发生,
   该函数就应调用来获得对应的错误代码.
返回值:返回值指出了该线程进行的上一次 Windows Sockets API 函数调用时的错误代码.
*/
#else
  return errno;
#endif
}
```

operator<<方法(输出到 strerr)

这几个方法就不说了，还是调用的 C 的库函数 fprintf 输出参数内容到 stderr。其实这里可以使用 C++面向对象的方法来解决。C++标准库中定义了 std::cerr 对象用于将数据发生到标准错误流，其用法和 std::cout 可谓是如出一辙。这里重载后使用方法也和 std::cout 及其类似，观察其返回值便知了。

```
UsageEnvironment& BasicUsageEnvironment::operator<<(char const* str) {
  if (str == NULL) str = "(NULL)"; // sanity check
  fprintf(stderr, "%s", str);
  return *this;
}
UsageEnvironment& BasicUsageEnvironment::operator<<(int i) {
  fprintf(stderr, "%d", i);
  return *this;
}
UsageEnvironment& BasicUsageEnvironment::operator<<(unsigned u) {
  fprintf(stderr, "%u", u);
  return *this;
}
```

```
UsageEnvironment& BasicUsageEnvironment::operator<<(double d) {
  fprintf(stderr, "%f", d);
  return *this;
}
UsageEnvironment& BasicUsageEnvironment::operator<<(void* p) {
  fprintf(stderr, "%p", p);
  return *this;
}
```

四、网络相关组件类

这些组件都是为 GroupSock 服务的

1.网络通用数据类型定义

因为 live555 跨平台的特点，需要定义一些在数据类型来适应各个平台环境。

这写代码在 live555sourcecontrol\groupsock\include\NetCommon.h 文件中

```
#if defined(__WIN32__) || defined(_WIN32) || defined(_WIN32_WCE)
/* Windows */
#if defined(WINNT) || defined(_WINNT) || defined(__BORLANDC__) || defined(__MINGW32__) || defined(_WIN3
2_WCE)
#define _MSWSOCK_
#include <winsock2.h>
#include <ws2tcpip.h>
#endif
#include <windows.h>
#include <string.h>

#define closeSocket   closesocket     //关闭 socket 函数
#define EWOULDBLOCK   WSAEWOULDBLOCK  //10035L 可能会被阻塞
#define EINPROGRESS   WSAEWOULDBLOCK  //10035L 操作正在进行
#define EAGAIN    WSAEWOULDBLOCK  //10035L 再试一次
#define EINTR     WSAEINTR     //10004L 中断

#if defined(_WIN32_WCE)
#define NO_STRSTREAM 1
#endif

/* Definitions of size-specific types: 定义特定大小的类型*/
typedef __int64 int64_t;
typedef unsigned __int64 u_int64_t;
typedef unsigned u_int32_t;
typedef unsigned short u_int16_t;
typedef unsigned char u_int8_t;
// For "uintptr_t" and "intptr_t", we assume that if they're not already defined, then this must be
// “uintptr_t”和“intptr_t”，我们认为如果他们不是已经定义，那么这一定是
// an old, 32-bit version of Windows: 一个老的，32 位版本的 Windows：
#if !defined(_MSC_STDINT_H_) && !defined(_UINTPTR_T_DEFINED) && !defined(_UINTPTR_T_DECLARED) && !defin
ed(_UINTPTR_T)
typedef unsigned uintptr_t;
#endif
#if !defined(_MSC_STDINT_H_) && !defined(_INTPTR_T_DEFINED) && !defined(_INTPTR_T_DECLARED) && !defined
(_INTPTR_T)
typedef int intptr_t;
#endif

#elif defined(VXWORKS)
/* VxWorks */
#include <time.h>
#include <timers.h>
#include <sys/times.h>
#include <sockLib.h>
#include <hostLib.h>
#include <resolvLib.h>
#include <ioLib.h>

typedef unsigned int u_int32_t;
```


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

```
typedef unsigned short u_int16_t;
typedef unsigned char u_int8_t;

#else
/* Unix */
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <strings.h>
#include <ctype.h>
#include <stdint.h>
#if defined(_QNX4)
#include <sys/select.h>
#include <unix.h>
#endif

#define closeSocket close

#ifdef SOLARIS
#define u_int64_t uint64_t
#define u_int32_t uint32_t
#define u_int16_t uint16_t
#define u_int8_t uint8_t
#endif
#endif

#ifndef SOCKLEN_T
#define SOCKLEN_T int
#endif
```

2.Tunnel 隧道封装

这里代码里面已经注释得很明白了。这个首先需要了解以下什么是 Tunnel(隧道)。

这里实现的 TunnelEncapsulationTrailer 类是一个很特殊的类，它不应该被用来创建对象。

对于这一部分，这里先不多说，先看后面的。

其定义在 live555sourcecontrol\groupsock\include\TunnelEncaps.hh 文件中

```
typedef u_int16_t Cookie;
/* cookie（储存在用户本地终端上的数据）
Cookie，有时也用其复数形式 Cookies，指某些网站为了辨别用户身份、进行 session 跟踪而
储存在用户本地终端上的数据（通常经过加密）。定义于 RFC2109 和 2965 都已废弃，最新规范是 RFC6265 。
*/

/*tunnel 中文译为隧道。网络隧道(Tunnelling)技术是个关键技术。网络隧道技术指的是利用一种网络协议来传输另一种网络协议，它主要利用网络隧道协议来实现这种功能。网络隧道技术涉及了三种网络协议，即网络隧道协议、隧道协议下面的承载协议和隧道协议所承载的被承载协议。
*/

// 这个类很有意思，它内部并无数据成员，其函数成员的返回都是以 this 为基准进行偏移
```

```
// 后，转换这个偏移后的地址为相应的指针类型，再取指针指向内存的内容。
// 所以这个类并不会用来创建对象，而是作为一种类型来使用。可能诸如以下代码
// unsigned long long t= 0x1239874560864216L;
//cout << ((TunnelEncapsulationTrailer*)&t)->address() << endl;
//cout << 0x12398745 << endl;

class TunnelEncapsulationTrailer {
  // The trailer is layed out as follows:
  // bytes 0-1:   source 'cookie'        源 Cookie
  // bytes 2-3:   destination 'cookie'   目的 Cookie
  // bytes 4-7:   address                地址
  // bytes 8-9:   port                端口
  // byte 10:     ttl                 TTL
  // byte 11:     command                命令

       // Optionally, there may also be a 4-byte 'auxilliary address'
     // 随意，也可能有一个 4 字节的"辅助地址"
       // (e.g., for 'source-specific multicast' preceding this)
     // （例如，“特定源组播”在此之前）
       // bytes -4 through -1: auxilliary address
     // -4 到-1 字节(this 之前 4 个字节)，辅助地址

  public:
Cookie& srcCookie()
    { return *(Cookie*)byteOffset(0); }
Cookie& dstCookie()
    { return *(Cookie*)byteOffset(2); }
u_int32_t& address()
    { return *(u_int32_t*)byteOffset(4); }
Port& port()
    { return *(Port*)byteOffset(8); }
u_int8_t& ttl()
    { return *(u_int8_t*)byteOffset(10); }
u_int8_t& command()
    { return *(u_int8_t*)byteOffset(11); }

    u_int32_t& auxAddress()
          { return *(u_int32_t*)byteOffset(-4); }

  private:
//取 this 偏移 charIndex
inline char* byteOffset(int charIndex)
     { return ((char*)this) + charIndex; }
};

const unsigned TunnelEncapsulationTrailerSize = 12; // bytes 隧道封装拖车尺寸
const unsigned TunnelEncapsulationTrailerAuxSize = 4; // bytes 辅助的尺寸
const unsigned TunnelEncapsulationTrailerMaxSize  //最大尺寸
    = TunnelEncapsulationTrailerSize + TunnelEncapsulationTrailerAuxSize;

// Command codes:命令码
// 0: unused
const u_int8_t TunnelDataCmd = 1;         //隧道的数据命令
const u_int8_t TunnelJoinGroupCmd = 2;       //隧道连接组命令
const u_int8_t TunnelLeaveGroupCmd = 3;      //隧道离开组命令
const u_int8_t TunnelTearDownCmd = 4;     //隧道拆除命令
const u_int8_t TunnelProbeCmd = 5;        //隧道探针命令
const u_int8_t TunnelProbeAckCmd = 6;     //隧道探针 ACK 命令
const u_int8_t TunnelProbeNackCmd = 7;       //隧道探针 NACK 命令
```


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

```
const u_int8_t TunnelJoinRTPGroupCmd = 8;   //隧道加入 RTP 组命令
const u_int8_t TunnelLeaveRTPGroupCmd = 9;  //隧道离开 RTP 组命令

// 0x0A through 0x10: currently unused.0x0a 到 0x10：目前未使用
// a flag, not a cmd code 一个标识，不是命令码。隧道扩展标识
const u_int8_t TunnelExtensionFlag = 0x80;  //bits:1000 0000

const u_int8_t TunnelDataAuxCmd          //隧道数据辅助命令
    = (TunnelExtensionFlag|TunnelDataCmd);
const u_int8_t TunnelJoinGroupAuxCmd  //隧道连接组辅助命令
    = (TunnelExtensionFlag|TunnelJoinGroupCmd);
const u_int8_t TunnelLeaveGroupAuxCmd //隧道离开组辅助命令
    = (TunnelExtensionFlag|TunnelLeaveGroupCmd);
// Note: the TearDown, Probe, ProbeAck, ProbeNack cmds have no Aux version
// 注意：TearDown(拆除),Probe(探针),ProbeAck(Ack 探针),ProbeNack(NACK 探针)没有辅助版命令
// 0x84 through 0x87: currently unused.
const u_int8_t TunnelJoinRTPGroupAuxCmd  //隧道加入 RTP 组辅助命令
    = (TunnelExtensionFlag|TunnelJoinRTPGroupCmd);
const u_int8_t TunnelLeaveRTPGroupAuxCmd//隧道离开 RTP 组辅助命令
    = (TunnelExtensionFlag|TunnelLeaveRTPGroupCmd);
// 0x8A through 0xFF: currently unused
  //判断参数 cmd 是否是辅助命令
inline Boolean TunnelIsAuxCmd(u_int8_t cmd) {
  return (cmd&TunnelExtensionFlag) != 0;
}
```

3.网络地址相关类

使用 Socket 进行的网络连接，网络地址一般由地址(IP)和端口(port)组成。

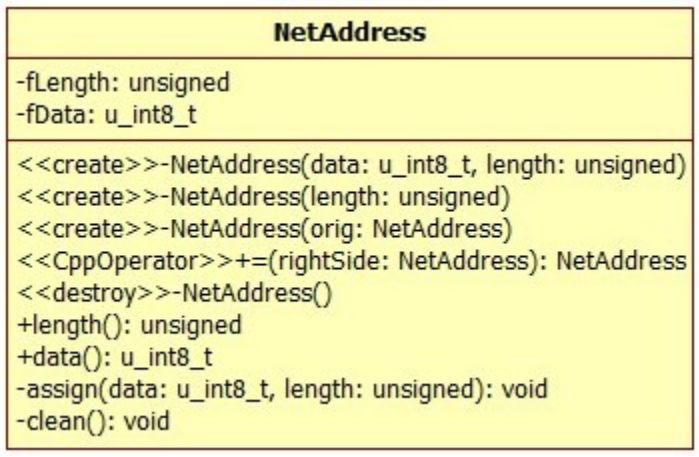
其定义了一些数据类型，表明了目前所支持的网络地址类型。

```
// Definition of a type representing a low-level network address.
// At present, this is 32-bits, for IPv4.  Later, generalize it,
// to allow for IPv6.
// 一种代表底层网络地址定义。目前，默认它 32 位，IPv4。将来，可扩展支持 IPv6。
typedef u_int32_t netAddressBits;
typedef u_int16_t portNumBits;
```

定义在文件 live555sourcecontrol\groupsock\include\NetAddress.hh

1) NetAddress 网络地址类

NetAddress 是一个用于保存网络地址的类，它不是对 struct sockaddr 的封装。其内部定义了两个数据成员，分别是用于保存地址数据的 u_int8_t* fData 和用于指示地址长度的 unsigned fLength。



下面是其定义

```
class NetAddress {
    public:
  NetAddress(u_int8_t const* data,
        unsigned length = 4 /* default: 32 bits IPv4*/);
  NetAddress(unsigned length = 4); // sets address data to all-zeros
  NetAddress(NetAddress const& orig);
  NetAddress& operator=(NetAddress const& rightSide);
  virtual ~NetAddress();

  unsigned length() const { return fLength; }
  u_int8_t const* data() const // always in network byte order
     { return fData; }

    private:
  void assign(u_int8_t const* data, unsigned length);
  void clean();

  unsigned fLength;
  u_int8_t* fData;
};
```

assign 方法(分配空间)

先说这个而不是构造函数，是因为这个方法是一个关键方法。构造函数也要用到它。

assign 为 fData 成员动态分配内存空间和拷贝数据。通过参数 length 来确定分配空间的大小，而参数 data 用于作为数据源拷贝到申请的新空间。要注意的是这个方法的权限是 private 的，所以没有检查 data==NULL 也是可以的。

这里提一个 C++的有意思的地方，就是 new 分配失败不是返回 NULL，而是抛出异常(std::bad_alloc e)。除非是重载的 new 或者使用无抛出的 new (std::nothrow)。但是早期一些 C++编译器的实现可能是 new 与 malloc 行为一致，都是返回 NULL。

```
//为 fDate 申请 length 字节内存空间，并将 data 指向内容拷贝到新空间
void NetAddress::assign(u_int8_t const* data, unsigned length) {
  fData = new u_int8_t[length];
  if (fData == NULL) {
    fLength = 0;
    return;
  }

  for (unsigned i = 0; i < length; ++i)  fData[i] = data[i];
  fLength = length;
}
```

NetAddress 的构造

NetAddress 定义了三个构造函数，两个普通的带参构造和一个拷贝构造(拷贝构造也是带参构造的一种)。

三个构造函数一致的特点就是都为 fData 成员动态申请了内存空间。代码很简单，不详述了。

```
//构造函数，为 fDate 申请 length 字节内存空间，并将 data 指向内容拷贝到新空间
NetAddress::NetAddress(u_int8_t const* data, unsigned length) {
  assign(data, length);
}
//为 fDate 申请 length 字节内存空间，并将新空间清零
NetAddress::NetAddress(unsigned length) {
  fData = new u_int8_t[length];
  if (fData == NULL) {
    fLength = 0;
    return;
  }

  for (unsigned i = 0; i < length; ++i)  fData[i] = 0;
  fLength = length;
}

//拷贝构造
NetAddress::NetAddress(NetAddress const& orig) {
  assign(orig.data(), orig.length());
}
```

clean 方法(清理)与析构

clean 方法用于将 fData 指向的内存空间进行释放。就是将 NetAddress 对象保存的数据给清理掉了，其是 private 权限。

题外话：clean 和 clear 的意思还是有一点区别的。

```
//清除地址数据
void NetAddress::clean() {
  delete[] fData; fData = NULL;
  fLength = 0;
}
```

析构就是对 clean 的调用。

```
//析构
NetAddress::~NetAddress() {
  clean();
}
```

operate= 重载赋值操作

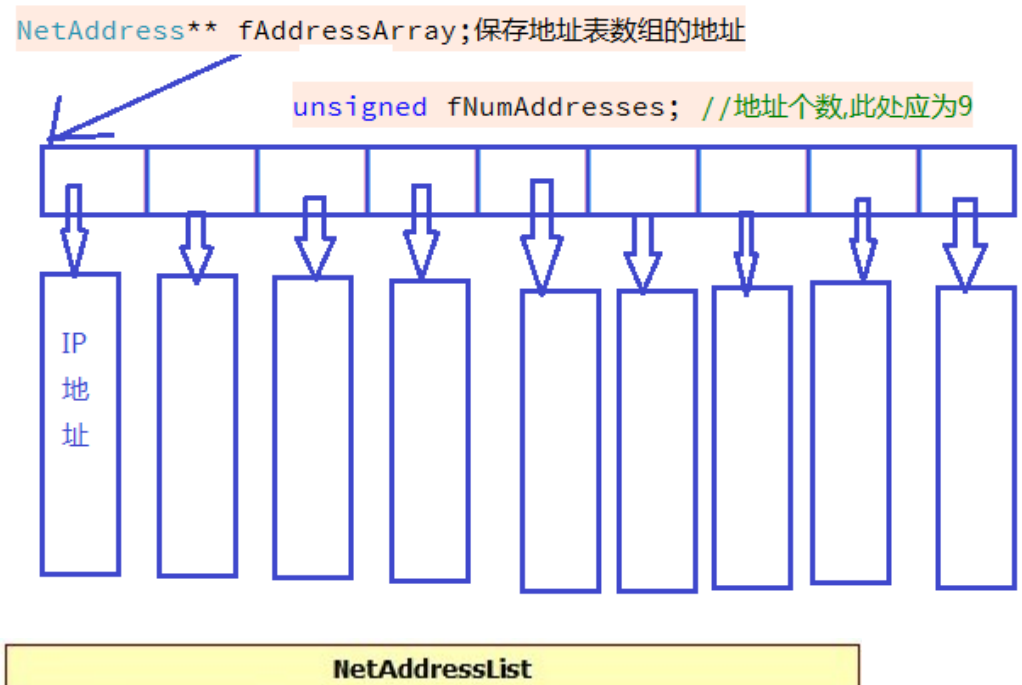
这个很简单，不详述了。

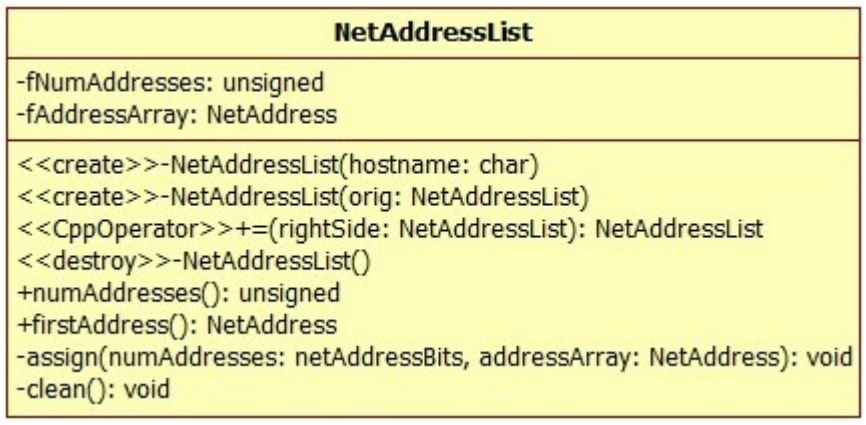
```
//重载 = 赋值
NetAddress& NetAddress::operator=(NetAddress const& rightSide) {
  if (&rightSide != this) {
    clean();
    assign(rightSide.data(), rightSide.length());
  }
  return *this;
}
```

2) NetAddressList 网络地址列表

网络地址列表是用于保存一系列网络地址的类。它与 NetAddress 无直接联系。

NetAddressList 类内部定义了一个二级指针 NetAddress** fAddressArray，在使用的时候给它动态申请一个元素个数为 unsigned fNumAddresses 的指针(NetAddress*)数组。指针数组的每一个元素又指向一个动态申请的 NetAddress 对象。






```
class NetAddressList {
public:
  // 构造函数 hostname 可以是一个点分十进制的 IP 地址，也可以是主机域名
  NetAddressList(char const* hostname);
  NetAddressList(NetAddressList const& orig);
  NetAddressList& operator=(NetAddressList const& rightSide);
  virtual ~NetAddressList();
  //获取地址表中元素个数
  unsigned numAddresses() const { return fNumAddresses; }
  //获取地址表第一个地址的内存地址
  NetAddress const* firstAddress() const;

  // Used to iterate through the addresses in a list:
  // 用于遍历列表中的地址:
  class Iterator {
  public:
     Iterator(NetAddressList const& addressList);
     NetAddress const* nextAddress(); // NULL iff none 没有跟多地址了
  private:
     NetAddressList const& fAddressList;//必须绑定一个地址表
     unsigned fNextIndex;  //下一个地址的索引
  };

private:
  //为地址表申请内存空间，并将表 addressArray 中的内容拷贝进去
  void assign(netAddressBits numAddresses, NetAddress** addressArray);
  //删除地址表和地址表中所有地址
  void clean();

  friend class Iterator;
  unsigned fNumAddresses;      //地址个数
```


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

```
  NetAddress** fAddressArray;  //地址表
};
```

assign 方法

assign 方法为地址表动态申请内存来保存地址元素。

要注意的是，这里所有的地址元素都是动态申请来的，所以释放的时候不知只释放 fAddressArray 指向的内存空间。

```
void NetAddressList::assign(unsigned numAddresses, NetAddress** addressArray) {
  //为地址表分配内存空间
  fAddressArray = new NetAddress*[numAddresses];
  if (fAddressArray == NULL) {
     fNumAddresses = 0;
     return;
  }
  //为地址表每个地址分配内存空间
  for (unsigned i = 0; i < numAddresses; ++i) {
     fAddressArray[i] = new NetAddress(*addressArray[i]);
  }
  fNumAddresses = numAddresses;
}
```

NetAddressList 的构造

NetAddressList(char const* hostname)构造函数很长，内容不多，但是涉及到一些网络编程的基础知识。

首先参数 hostname，是一个 C 风格的字符串，如果它保存的是一个点分十进制的 IP 地址(例如：”192.168.1.128”)，那么只会给这个地址表申请一个元素的空间来保存地址。注意，保存的地址在一个 NetAddress 对象中，对象里面保存的是整型数形式的地址。

这里有一句 netAddressBits addr = our_inet_addr((char*)hostname);这个函数的作用是把点分十进制的 IP 地址转换为整型数形式的地址。参数不是点分十进制的 IP 地址字符串，那么函数会返回错误码 INADDR_NONE 。our_inet_addr 实质上是调用的 inet_addr(库函数)，其定义在 live555sourcecontrol\groupsock\inet.c 文件中。

那如果参数 hostname 不是一个 IP 地址，那么它就应该是主机名(通常指域名，如 live555.com)。一个域名可能对应不止一个 IP 地址(windows 下可以使用 nslookup 命令查看，linux/unix 下可以用 dig 命令)。这里使用了 gethostbyname 函数来获取它的所有地址。然后分配空间拷贝保存了这些地址。

```
NetAddressList::NetAddressList(char const* hostname)
: fNumAddresses(0), fAddressArray(NULL) {
  // First, check whether "hostname" is an IP address string:
  // 首先，检查“hostname”是否是一个 IP 地址字符串
  netAddressBits addr = our_inet_addr((char*)hostname);
  if (addr != INADDR_NONE) {
     // Yes, it was an IP address string.  Return a 1-element list with this address:
     //它是一个 IP 地址字符串，那么这个地址表只需要 1 个元素
     fNumAddresses = 1;
     fAddressArray = new NetAddress*[fNumAddresses];
     if (fAddressArray == NULL) return;
     //申请空间，保存这个地址。注意保存的是整数地址而不是字符串
     fAddressArray[0] = new NetAddress((u_int8_t*)&addr, sizeof (netAddressBits));
     return;
  }

  // "hostname" is not an IP address string; try resolving it as a real host name instead:
  // 当它不是一个 IP 地址字符串，尝试解析 hostname 真实的地址来代替
#if defined(USE_GETHOSTBYNAME) || defined(VXWORKS)
  struct hostent* host;
#if defined(VXWORKS)
  char hostentBuf[512];
```


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

```
  host = (struct hostent*)resolvGetHostByName((char*)hostname, (char*)&hostentBuf, sizeof hostentBuf);
#else
  //gethostbyname()返回对应于给定主机名的包含主机名字和地址信息的 hostent 结构指针(不要试图 delete 这个返回的地址)
  host = gethostbyname((char*)hostname);
#endif
  if (host == NULL || host->h_length != 4 || host->h_addr_list == NULL) return; // no luck    //不幸，没有得到

  u_int8_t const** const hAddrPtr = (u_int8_t const**)host->h_addr_list;
  // First, count the number of addresses:取得地址个数
  u_int8_t const** hAddrPtr1 = hAddrPtr;
  while (*hAddrPtr1 != NULL) {
     ++fNumAddresses;
     ++hAddrPtr1;
  }

  // Next, set up the list: 给地址表分配内存
  fAddressArray = new NetAddress*[fNumAddresses];
  if (fAddressArray == NULL) return;
  //逐个拷贝地址到地址表
  for (unsigned i = 0; i < fNumAddresses; ++i) {
     fAddressArray[i] = new NetAddress(hAddrPtr[i], host->h_length);
  }
#else
  // Use "getaddrinfo()" (rather than the older, deprecated "gethostbyname()"):
  struct addrinfo addrinfoHints;
  memset(&addrinfoHints, 0, sizeof addrinfoHints);
  addrinfoHints.ai_family = AF_INET; // For now, we're interested in IPv4 addresses only
  struct addrinfo* addrinfoResultPtr = NULL;
  int result = getaddrinfo(hostname, NULL, &addrinfoHints, &addrinfoResultPtr);
  if (result != 0 || addrinfoResultPtr == NULL) return; // no luck

  // First, count the number of addresses:
  const struct addrinfo* p = addrinfoResultPtr;
  while (p != NULL) {
     if (p->ai_addrlen < 4) continue; // sanity check: skip over addresses that are too small
     ++fNumAddresses;
     p = p->ai_next;
  }

  // Next, set up the list:
  fAddressArray = new NetAddress*[fNumAddresses];
  if (fAddressArray == NULL) return;

  unsigned i = 0;
  p = addrinfoResultPtr;
  while (p != NULL) {
     if (p->ai_addrlen < 4) continue;
     fAddressArray[i++] = new NetAddress((u_int8_t const*)&(((struct sockaddr_in*)p->ai_addr)->sin_addr.s_addr), 4);
     p = p->ai_next;
  }

  // Finally, free the data that we had allocated by calling "getaddrinfo()":
  freeaddrinfo(addrinfoResultPtr);
#endif
}
```


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

clean 方法与析构

先说 clean 方法，它的作用是将地址表和表中所有的地址元素都释放了。之前 assign 分配空间，在这里对应的释放。

```
void NetAddressList::clean() {
  while (fNumAddresses-- > 0) {//逐个删除地址
     delete fAddressArray[fNumAddresses];
  }
  //释放地址表
  delete[] fAddressArray; fAddressArray = NULL;
}
```

析构函数就是简单的调用 clean。

```
NetAddressList::~NetAddressList() {
  clean();
}
```

拷贝构造与赋值运算符重载

这里就不多说了，代码很明白。(有人问赋值和拷贝构造的区别，这里简单说一下。拷贝构造的重点在于构造，是对象还没有的时候调用来创建一个一样的对象的，而赋值的重点在于赋值，是对象已经存在的时候，用来替换对象数据的。)

```
NetAddressList::NetAddressList(NetAddressList const& orig) {
  assign(orig.numAddresses(), orig.fAddressArray);
}

NetAddressList& NetAddressList::operator=(NetAddressList const& rightSide) {
  if (&rightSide != this) {
     clean();
     assign(rightSide.numAddresses(), rightSide.fAddressArray);
  }
  return *this;
}
```

NetAddressList::Iterator 迭代器

这里的迭代器与之前的 HanlerSet 类和 DelayQueue 很像。这里 NetAddressList::Iterator 在 NetAddressList 类内部嵌套定义的类，权限是 public。在构造的时候，其也需要绑定一个 NetAddressList 对象，迭代器方法 nextAddress 返回类型是 NetAddress const*，这里要注意一下。

```
NetAddressList::Iterator::Iterator(NetAddressList const& addressList)
: fAddressList(addressList), fNextIndex(0) {}

NetAddress const* NetAddressList::Iterator::nextAddress() {
  if (fNextIndex >= fAddressList.numAddresses()) return NULL; // no more
  return fAddressList.fAddressArray[fNextIndex++];
}
```

3) Port 端口类

端口类是用于保存网络端口的，计算机网络端口一般有两种含义，分别是物理意义上的网络设备接口和逻辑意义上的端口。这里指的就是逻辑意义上的端口(特指 TCP/IP 协议中端口)，端口的范围是 0 到 65535(u_int16_t 的表示范围)。

Port 类只有一个数据成员 portNumBits fPortNum，用于保存端口值。保存的是以网络字节序表示的。网络字节序是大端序。

字节序在这里稍微提一下。字节序分为大端序和小端序。拿这里来书，fPortNum 是 16bits 宽度的，其占了两个字节。假如它代表的内存区块是 0x1000 和 0x1001 这两个字节，其保存的内容是 0x5678，那么是那个字节表示 0x56 那个字节是 0x78 呢？这就涉及到字节序的问题了。通常把低地址保存低位，高地址保存高位的叫做小端序。反之为大端序。

内存地址	小端序表示 0x5678	大端序表示 0x5678
0x1001	0x56	0x78
0x1000	0x78	0x56

```
typedef u_int16_t portNumBits;
class Port {
public:
  Port(portNumBits num /* in host byte order */);

  portNumBits num() const // in network byte order
  {
     return fPortNum;
  }

private:
  portNumBits fPortNum; // stored in network byte order
#ifdef IRIX
  portNumBits filler; // hack to overcome a bug in IRIX C++ compiler
#endif
};
```

Port 的构造与全局的 << 运算符重载

构造就是对其内部的 fPortNum 进行赋值，注意赋值的时候是转换为了网络字节序。htons 函数是 host to network,return short 的意思，将参数 num 从主机序转换到网络序(可能有的主机序和网络序一致)。所有传参的时候传正常使用的就是了，别传已经转换为网络序的。

```
Port::Port(portNumBits num /* in host byte order */) {
  fPortNum = htons(num);
}
```

重载全局的 << 是为了使用 BasicUsageEnvironment 对象来输出端口值的。这里的函数 ntohs 是 htons 的反操作，其将网络字节序转为主机字节序。

```
UsageEnvironment& operator<<(UsageEnvironment& s, const Port& p) {
  return s << ntohs(p.num());
}
```

4) AddressString 字符串型地址类

AddressString类是用于以点分十进制的C风格字符串形式保存的IP地址,这是为了替换"inet_ntoa()",因为它不是线程安全的。这里就不介绍什么是线程安全了，inet_ntoa 内部存在静态变量，在不同的线程中调用可能会导致混乱。

AddressString 类定义了一个数据成员 fVal 用于动态申请内存来保存数据，注意这个类目前只能用于 IPv4 的地址保存，对于 IPv6 还未做支持。

```
// A mechanism for displaying an IPv4 address in ASCII.  This is intended to replace "inet_ntoa()", whic
h is not thread-safe.
// 一种机制，用于 ASCII 码显示 IPv4 地址。这是为了替换"inet_ntoa()",因为这是不是线程安全的。
class AddressString {
public:
  AddressString(struct sockaddr_in const& addr);
  AddressString(struct in_addr const& addr);
  AddressString(netAddressBits addr); // "addr" is assumed to be in host byte order here

  virtual ~AddressString();

  char const* val() const { return fVal; }
```


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

```
private:
  void init(netAddressBits addr); // used to implement each of the constructors

private:
  char* fVal; // The result ASCII string: allocated by the constructor; deleted by the destructor
  //其结果是 ASCII 码字符串：由构造函数分配;在析构函数中删除
};
```

AddressString::init 方法

AddressString::init 方法是一个 private 权限方法，是用于实现构造对象的时候初始化的，其只被构造函数调用。

这里在 sprintf 之前调用 htonl 将地址转为网络字节序，是为了方便 sprintf 的操作。因为一块内存空间的首地址是低地址，网络字节序是大端序，低地址保存高位。这里确保我们有一个准确的字节序。

```
void AddressString::init(netAddressBits addr) {
  //针对的是 IPv4 类型，16byte 足够,IPv6 需要 46Byte
  fVal = new char[16]; // large enough for "abc.def.ghi.jkl"
  //转为网络字节序
  netAddressBits addrNBO = htonl(addr); // make sure we have a value in a known byte order: big endian
  //转为点分十进制表示
  sprintf(fVal, "%u.%u.%u.%u", (addrNBO >> 24) & 0xFF, (addrNBO >> 16) & 0xFF, (addrNBO >> 8) & 0xFF, ad
drNBO & 0xFF);
  }
```

AddressString 构造与析构

构造没什么好说的，都是调用的 init 方法，只需要注意在 init 中为成员 fVal 申请了内存空间。

```
AddressString::AddressString(struct sockaddr_in const& addr) {
  init(addr.sin_addr.s_addr);
}

AddressString::AddressString(struct in_addr const& addr) {
  init(addr.s_addr);
}

AddressString::AddressString(netAddressBits addr) {
  init(addr);
}
```

析构的时候释放 init 方法中申请的内存空间。

```
AddressString::~AddressString() {
  delete[] fVal;
}
```

5) AddressPortLookupTable 地址端口查找表类

AddressPortLookupTable 类内部定义了一个 HashTable* fTable 用于保存哈希表的地址。在构造函数中动态创建了一个哈希表对象给它。AddressPortLookupTable 使用了两个地址和一个端口号组合作为一个 key，value 是 Add 方法的时候确定的。

AddressPortLookupTable 类只提供了增删查三种操作，没有提供修改表项的操作。

使用哈希表的优点在于可以快速的查找 key 对应的 value。

```
// A generic table for looking up objects by (address1, address2, port)
// 用于查找对象,通过一个通用表（地址 1，地址 2，端口）
class AddressPortLookupTable {
public:
```

```
// 为内部哈希表 fTable 创建对象，哈希表的 key 是 3 个元素的 unsigned int 数组
AddressPortLookupTable();
// 释放内部哈希表 fTable
virtual ~AddressPortLookupTable();

// 使用 address1、address2、port 组成 key,value 为值添加到哈希表
// 如果对应 key 的条目已经存在，返回旧的 value，否则返回 NULL
void* Add(netAddressBits address1, netAddressBits address2,
    Port port, void* value);
// Returns the old value if different, otherwise 0

//从哈希表中移除 key 对应的条目，对应条目存在返回 true
Boolean Remove(netAddressBits address1, netAddressBits address2,
    Port port);
// 从哈希表中查找 key 对应的 value，没找到返回 NULL
void* Lookup(netAddressBits address1, netAddressBits address2,
    Port port);
// Returns 0 if not found

  // Used to iterate through the entries in the table
  // 用于遍历在表中的条目
  class Iterator {
  public:
     Iterator(AddressPortLookupTable& table);
     virtual ~Iterator();

     void* next(); // NULL iff none

  private:
     HashTable::Iterator* fIter;  //哈希表迭代器
  };

private:
  friend class Iterator;
  HashTable* fTable;     //哈希表
};
```

AddressPortLookupTable 构造与析构

AddressPortLookupTable 在构造的时候创建哈希表

```
AddressPortLookupTable::AddressPortLookupTable()
: fTable(HashTable::create(3)) { // three-word keys are used 键使用 3 个元素的 unsigned int 数组
}
```

析构的时候释放哈希表

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

Add 方法（添加表项）

Add 方法使用前三个参数来组合作为一个 key，第四个参数是 value。创建一个表项添加到哈希表。

如果 key 对应的表项在哈希表中已经存在，那么返回值是已经存在表项的旧 value，这个表项的 value 替换为参数 value。如果不存在，那就返回 NULL。（表项=条目）

```
// 使用 address1、address2、port 组成 key,value 为值添加到哈希表
void* AddressPortLookupTable::Add(netAddressBits address1,
  netAddressBits address2,
```

```
  Port port, void* value) {
  int key[3];
  key[0] = (int)address1;
  key[1] = (int)address2;
  key[2] = (int)port.num();
  return fTable->Add((char*)key, value);
}
```

Remove 方法（移除表项）

Remove 方法用于从哈希表中移除表项，这三个参数依然是用于组成 key 的。如果 key 在表中存在对应的表项，那么移除后函数返回 true，否则返回 false。

```
//从哈希表中移除 key 对应的条目，对应条目存在返回 true
Boolean AddressPortLookupTable::Remove(netAddressBits address1,
  netAddressBits address2,
  Port port) {
  int key[3];
  key[0] = (int)address1;
  key[1] = (int)address2;
  key[2] = (int)port.num();
  return fTable->Remove((char*)key);
}
```

Lookup 方法（查找表项）

这里说查找表项，不是很准确，应该是查找表项的 value。如果 key 对应的表项不存在，那么就返回 NULL。存在就返回表项的 value。

```
// 从哈希表中查找 key 对应的 value，没找到返回 NULL
void* AddressPortLookupTable::Lookup(netAddressBits address1,
  netAddressBits address2,
  Port port) {
  int key[3];
  key[0] = (int)address1;
  key[1] = (int)address2;
  key[2] = (int)port.num();
  return fTable->Lookup((char*)key);
}
```

AddressPortLookupTable 迭代器方法

AddressPortLookupTable 迭代器还有三个方法，构造析构和 next。其实质是对 HashTable::Iterator 的操作。迭代器创建的时候指向哈希表的第一个条目。

构造函数，构造的时候必须绑定一个 AddressPortLookupTable 对象。

```
// 创建迭代器，绑定地址端口查找表
AddressPortLookupTable::Iterator::Iterator(AddressPortLookupTable& table)
// 创建哈希表迭代器，绑定哈希表
: fIter(HashTable::Iterator::create(*(table.fTable))) {
}
```

析构函数，删除迭代器 HashTable::Iterator fIter。

```
AddressPortLookupTable::Iterator::~Iterator() {
  delete fIter;
}
```

next 方法的返回值需要注意一下，返回的是当前迭代器指向表中条目的 value。然后迭代器会走向下一个，如果走到哈希表的尾部元素之后，那么返回 NULL。

```
// 返回当前迭代器指向条目的 value,迭代器走向下一个
```

```
void* AddressPortLookupTable::Iterator::next() {
  char const* key; // dummy
  return fIter->next(key);
}
```

4.网络相关函数

网络相关函数是一系列用于操作网络数据的函数。在多个文件中都有相关的函数的定义。还有一些函数是系统 socket API 相关函数，就不提了。

这一系列的函数大多有一个特点，需要一个 UsageEnvironmet&型的参数。

这些方法大多在 live555sourcecontrol\groupsock\include\GroupsockHelper.hh 中声明。

1) IsMulticastAddress 多播(组播)地址判断函数

IsMulticastAddress 用于判断一个地址是否为多播(组播)地址，如果是的话返回 true，否则返回 false。

声明在文件 live555sourcecontrol\groupsock\include\NetAddress.hh 中

有些应用会有这样的要求：一些分布在各处的进程需要以组的方式协同工作，组中的进程通常要给其他所有的成员发送消息。即有这样的一种方法能够给一些明确定义的组发送消息，这些组的成员数量虽然很多，但是与整个网络规模相比却很小。给这样一个组发送消息称为多点点播送，简称多播。

```
/ 判断参数释放是一个多播地址
Boolean IsMulticastAddress(netAddressBits address) {
  // Note: We return False for addresses in the range 224.0.0.0
  // through 224.0.0.255, because these are non-routable
  // 注意：我们在 224.0.0.0 到 224.0.0.255 范围地址返回 false，因为这些是不可路由的
  // Note: IPv4-specific #####
  // 注：支持 IPv4 特定#####
  netAddressBits addressInNetworkOrder = htonl(address);
  return addressInNetworkOrder > 0xE00000FF &&
     addressInNetworkOrder <= 0xEFFFFFFF;
}
```

多播简要说明

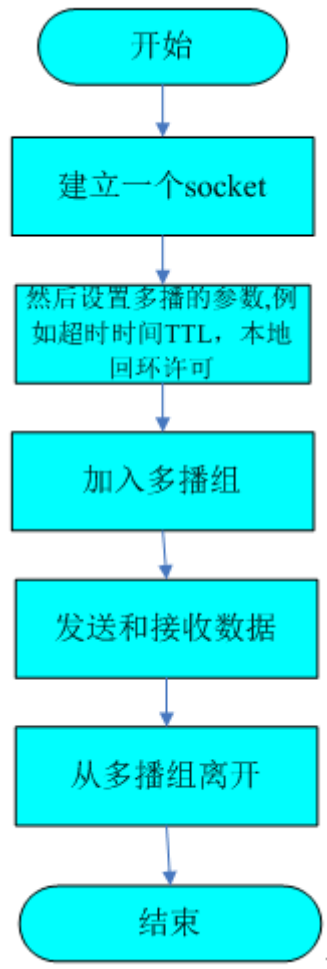
IP 多播（也称多址广播或组播）技术，是一种允许一台或多台主机（多播源）发送单一数据包到多台主机（一次的，同时的）的 TCP/IP 网络技术。多播作为一点对多点的通信，是节省网络带宽的有效方法之一。在网络音频/视频广播的应用中，当需要将一个节点的信号传送到多个节点时，无论是采用重复点对点通信方式，还是采用广播方式，都会严重浪费网络带宽，只有多播才是最好的选择。多播能使一个或多个多播源只把数据包发送给特定的多播组，而只有加入该多播组的主机才能接收到数据包。目前，IP 多播技术被广泛应用在网络音频/视频广播、AOD/VOD、网络视频会议、多媒体远程教育、“push”技术（如股票行情等）和虚拟现实游戏等方面。

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。其中，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；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；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。

使用同一个 IP 多播地址接收多播数据包的所有主机构成了一个主机组，也称为多播组。一个多播组的成员是随时变动的，一台主机可以随时加入或离开多播组，多播组成员的数目和所在的地理位置也不受限制，一台主机也可以属于几个多播组。此外，不属于某一个多播组的主机也可以向该多播组发送数据包。

多播编程

1.流程



1>建立一个 socket;

2>设置多播的参数，例如超时时间 TTL，本地回环许可 LOOP 等

3>加入多播组

4>发送和接收数据

5>从多播组离开

2.多播程序设计使用 setsockopt()函数和 getsockopt()函数来实现，组播的选项是 IP 层的。

3.setsockopt()的选项

1>IP_MULTICAST_TTL:设置多播组数据的 TTL 值（路由跳数），每跨过一个路由器，TTL 值减一.范围为 0~255 之间的任何值。

```
int  ttl;
setsockopt(sock_fd,IPPROTO_IP,IP_MULTICAST_TTL,&ttl,sizeof(ttl));
```

2>IP_MULTICAST_LOOP:默认情况下，当本机发送组播数据到某个网络接口时，在 IP 层，数据会回送公道本地的回环接口，选项 IP_MULTICAST_LOOP 用于控制数据是否回送到本地的回环接口。

```
int  loop;
setsockopt(sock_fd,IPPROTO_IP,IP_MULTICAST_LOOP,&loop,sizeof(loop));
```

参数 loop 设置为 0 表示禁止回送，设置为 1 允许回送。

3>IP_ADD_MEMBERSHIP: 该选项通过对一个结构 struct ip_mreq 类型的变量进行控制而加入一个多播组。

```
struct ip_mreq
{
       struct in_addr imr_multiaddr;/*加入的多播组 IP 地址*/
       struct  in_addr imr_interface;/*加入的网络接口 IP 地址*/
};
```

选项 IP_ADD_MEMBESHIP 选项用于加入某个多播组，之后就可以向这个多播组发送数据或者从多播组接收数据。此选项的值为 mreq 结构，成员 imr_multiaddr 是需要加入的多播组 IP 地址，成员 imr_interface 是本机需要加入多播组的网络接口 IP 地址。

```
struct ip_mreq  mreq;
setsockopt(sock_fd,IPPROTO_IP,IP_ADD_MEMBERSHIP,&mreq,sizeof(mreq));
```

使用 IP_ADD_MEMBERSHIP 选项每次只能加入一个网络接口的 IP 地址到多播组，但并不是一个多播组仅允许一个主机 IP 地址加入，可以多次调用 IP_ADD_MEMBERSHIP 选项来实现多个 IP 地址加入同一个多播组，或者同一个 IP 地址加入多个多播组。当 imr_interface 为 INADDR_ANY 时，选择的本地默认网口。

4>IP_DROP_MEMBERSHIP:该选项用于从一个多播组中退出。

```
struct ip_mreq  mreq;
setsockopt(sock_fd,IPPROTP_IP,IP_DROP_MEMBERSHIP,&merq,sizeof(sreq));
```

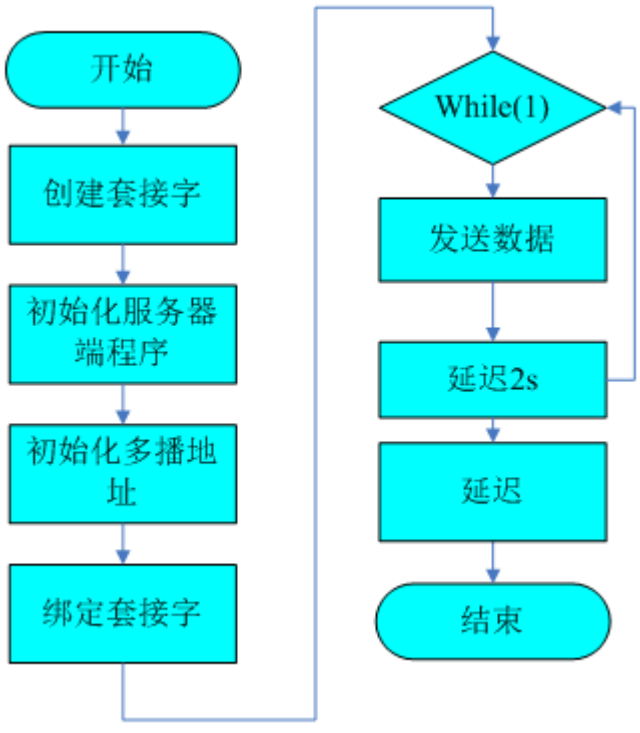
实例：

情景：

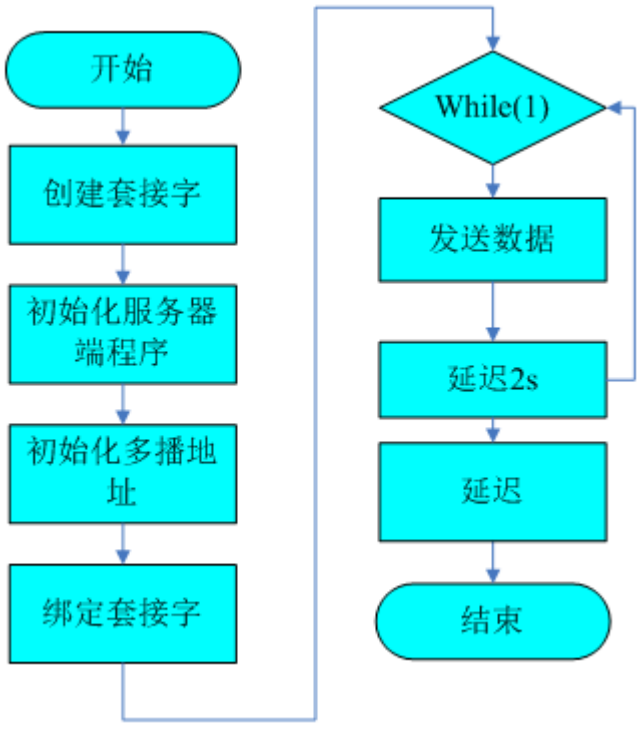
服务器端每隔 2 秒向目的端口号为 5000 和目的多播地址为 224.0.0.255 发送数据 welcome you to multicast socket programme。

客户端从多播地址为 224.0.0.255 和端口号 5000 处接收 5 次多播数据。

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



服务器端程序



客户端程序

2）socketErr 套接口错误

socketErr 是个静态方法，定义在 live555sourcecontrol\groupsock\GroupsockHelper.cpp 文件中。

实现很简单，把 errorMsg 中的内容设置到 env 中取。

```
static void socketErr(UsageEnvironment& env, char const* errorMsg) {
  env.setResultErrMsg(errorMsg);
}
```

3）groupsockPriv 函数

这个函数为其参数 env 成员 groupsockPriv 创建一个对象。

在看这个函数的时候先看一个结构体定义

```
struct _groupsockPriv { // There should be only one of these allocated
  HashTable* socketTable;      // socket 哈希表
  int reuseFlag;               //重新使用标识
};
```

我们回忆一下，在 UsageEnvironment 中有两个数据成员， void*类型的指针 liveMediaPriv 和 groupsockPriv 没有使用到，而且它们是 public 权限的。

那么在这里，groupsockPriv 成员将会进行赋值操作

groupsockPriv 函数的作用就是给参数 env 的 groupsockPriv 申请一个_groupsockPriv 对象。当然，是在其为 NULL 的情况下。

这里设置了 env.groupsockPriv 指向对象的两个成员的默认值，socketTable=NULL、reuseFlag=1。

```
_groupsockPriv* groupsockPriv(UsageEnvironment& env) {
  if (env.groupsockPriv == NULL) { // We need to create it 我们需要创建它
    _groupsockPriv* result = new _groupsockPriv;   //创建结构体
    result->socketTable = NULL;
    result->reuseFlag = 1; // default value => allow reuse of socket numbers
    env.groupsockPriv = result;//赋值
  }
  return (_groupsockPriv*)(env.groupsockPriv);
}
```

4) reclaimGroupsockPriv 函数

reclaimGroupsockPriv 函数为其参数 env 的成员 groupsockPriv 决定是否释放其对象。

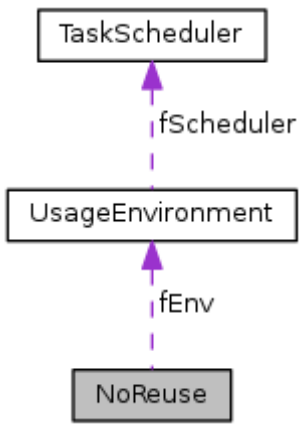
只有在其为默认值的时候，才进行释放。

```
void reclaimGroupsockPriv(UsageEnvironment& env) {
  _groupsockPriv* priv = (_groupsockPriv*)(env.groupsockPriv);
  // 两个成员是默认值的时候，进行释放
  if (priv->socketTable == NULL && priv->reuseFlag == 1/*default value*/) {
     // We can delete the structure (to save space); it will get created again, if needed:
     delete priv;
     env.groupsockPriv = NULL;
  }
}
```

5) NoReuse 不重用地址类

env.groupsockPriv->reuseFlag 成员用于指示在 setupDatagramSocket 函数中是否设置允许重用本地地址和端口。

构造的时候为构造的时候为 env.groupsockPriv 分配对象,并设置 groupsockPriv 对象的 reuseFlag=0 即不重用标识。默认情况下 reuseFlag==1，标识可重用。




```
// 构造的时候为 env.groupsockPriv 分配对象
// 并设置 groupsockPriv 对象的 reuseFlag=0
NoReuse::NoReuse(UsageEnvironment& env)
: fEnv(env) {
  groupsockPriv(fEnv)->reuseFlag = 0;
}
```

只要 env.groupsockPriv->socketTable==NULL 成立，就释放 env.groupsockPriv 指向对象。

```
// 若 groupsockPriv 对象的 socketTable==NULL
// 析构的时候为 env.groupsockPriv 释放对象
NoReuse::~NoReuse() {
  groupsockPriv(fEnv)->reuseFlag = 1;
  reclaimGroupsockPriv(fEnv);
}
```

6) initializeWinsockIfNecessary 根据需要初始化 winSock

这个函数只用于 windows 系列操作系统。

windows 网络编程是一件麻烦事，其必须要先进行一系列初始化的操作。

如果不是 windows 平台，这个函数会被宏替换为 1，就是说必然成功。

```
#if defined(__WIN32__) || defined(_WIN32)
#ifndef IMN_PIM
```

```
#define WS_VERSION_CHOICE1 0x202/*MAKEWORD(2,2)*/
#define WS_VERSION_CHOICE2 0x101/*MAKEWORD(1,1)*/
int initializeWinsockIfNecessary(void) {
  /* We need to call an initialization routine before
   * we can do anything with winsock.  (How fucking lame!):
   我们需要调用初始化例程
   之后我们可以用 Winsock 做任何事。（怎么他妈的没用的！）：
   */
  static int _haveInitializedWinsock = 0;
  WSADATA   wsadata;

  if (!_haveInitializedWinsock) {
     if ((WSAStartup(WS_VERSION_CHOICE1, &wsadata) != 0)
         && ((WSAStartup(WS_VERSION_CHOICE2, &wsadata)) != 0)) {
        return 0; /* error in initialization */
     }
        if ((wsadata.wVersion != WS_VERSION_CHOICE1)
            && (wsadata.wVersion != WS_VERSION_CHOICE2)) {
           WSACleanup();
           return 0; /* desired Winsock version was not available */
     }
     _haveInitializedWinsock = 1;
  }

  return 1;
}
#else
int initializeWinsockIfNecessary(void) { return 1; }
#endif
#else
#define initializeWinsockIfNecessary() 1
#endif
```

7) createSocket 创建 socket 方法

createSocket 使用参数 type 创建一个相关类型的 socket 套接口。如果有相关定义，将为这个套接口添加”close on exec”执行时关闭属性。

注意，这个函数是 static 类型的，只在本文件(live555sourcecontrol\groupsock\GroupsockHelper.cpp)内使用。

```
// type socket 类型，有 SOCK_STREAM、SOCK_DGRAM、SOCK_RAW、SOCK_PACKET、SOCK_SEQPACKET 等
static int createSocket(int type) {
  // Call "socket()" to create a (IPv4) socket of the specified type.
  // 调用“socket()创建一个（IPv4）指定类型的套接字。
  // But also set it to have the 'close on exec' property (if we can)
  // 还设置它具有“执行 exec 时关闭"属性（如果可以）
  int sock;

#ifdef SOCK_CLOEXEC
  sock = socket(AF_INET, type|SOCK_CLOEXEC, 0);
  if (sock != -1 || errno != EINVAL) return sock;
  // An "errno" of EINVAL likely means that the system wasn't happy with the SOCK_CLOEXEC; fall through
and try again without it:
#endif

  sock = socket(AF_INET, type, 0);
#ifdef FD_CLOEXEC
  if (sock != -1) fcntl(sock, F_SETFD, FD_CLOEXEC);
#endif
```


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

```
    return sock;
}
```

8) closeSocket 关闭套接口

closesocket 是 winsock.h 中声明的函数，用于关闭一个套接口

```
#define closeSocket    closesocket      //关闭 socket 函数
```

在 linux/unix 等系统下，被替换为 close 函数

```
#define closeSocket close//关闭 socket
```

9) setsockopt 设置 socket 套接口选项

这个函数是一个 socket API 函数，还有一个对应的 getsockopt 函数。在这里简单提一下。

函数说明：用于任意类型、任意状态套接口的设置选项值。选项可能存在于多层协议中，它们总会出现在最上面的套接字层。当操作套接字选项时，选项位于的层和选项的名称必须给出。为了操作套接字层的选项，应该 将层的值指定为 SOL_SOCKET。为了操作其它层的选项，控制选项的合适协议号必须给出。例如，为了表示一个选项由 TCP 协议解析，层应该设定为协议 号 TCP。

函数原型：

```
int setsockopt(int sockfd, int level, int optname,const void *optval, socklen_t optlen);
```

参数说明：

sockfd：标识一个套接口的描述字。

level：选项定义的层次；支持 SOL_SOCKET、IPPROTO_TCP、IPPROTO_IP 和 IPPROTO_IPV6。

optname：需设置的选项。

optval：指针，指向存放选项待设置的新值的缓冲区。

optlen：optval 缓冲区长度。

选项名称	说明	数据类型
SOL_SOCKET		
SO_BROADCAST	允许发送广播数据	int
SO_DEBUG	允许调试	int
SO_DONTROUTE	不查找路由	int
SO_ERROR	获得套接字错误	int
SO_KEEPALIVE	保持连接	int
SO_LINGER	延迟关闭连接	structlinger
SO_OOBINLINE	带外数据放入正常数据流	int
SO_RCVBUF	接收缓冲区大小	int
SO_SNDBUF	发送缓冲区大小	int
SO_RCVLOWAT	接收缓冲区下限	int
SO_SNDLOWAT	发送缓冲区下限	int
SO_RCVTIMEO	接收超时	structtimeval
SO_SNDTIMEO	发送超时	structtimeval
SO_REUSERADDR	允许重用本地地址和端口	int
SO_TYPE	获得套接字类型	int
SO_BSDCOMPAT	与 BSD 系统兼容	int
IPPROTO_IP		
IP_HDRINCL	在数据包中包含 IP 首部	int
IP_OPTINOS	IP 首部选项	int
IP_TOS	服务类型	
IP_MULTICAST_TTL	设置多播的 TTL 值	
IP_MULTICAST_IF	获取或设置多播接口	
IP_MULTICAST_LOOP	禁止多播数据回送到本地 loop 接口	
IP_ADD_MEMBERSHIP	将指定的接口加入多播	
IP_DROP_MEMBERSHIP	退出多播组	
IP_TTL	生存时间	int
IPPRO_TCP		
TCP_MAXSEG	TCP 最大数据段的大小	int

TCP_NODELAY	不使用 Nagle 算法	int

返回说明：

成功执行时，返回 0。失败返回-1，errno 被设为以下的某个值

EBADF：sock 不是有效的文件描述词

EFAULT：optval 指向的内存并非有效的进程空间

EINVAL：在调用 setsockopt()时，optlen 无效

ENOPROTOOPT：指定的协议层不能识别选项

ENOTSOCK：sock 描述的不是套接字

SO_RCVBUF 和 **SO_SNDBUF** 每个套接口都有一个发送缓冲区和一个接收缓冲区，使用这两个套接口选项可以改变缺省缓冲区大小。

```
// 接收缓冲区
int nRecvBuf=32*1024;         //设置为 32K
setsockopt(s,SOL_SOCKET,SO_RCVBUF,(const char*)&nRecvBuf,sizeof(int));
//发送缓冲区
int nSendBuf=32*1024;//设置为 32K
setsockopt(s,SOL_SOCKET,SO_SNDBUF,(const char*)&nSendBuf,sizeof(int));
```

注意：

当设置 TCP 套接口接收缓冲区的大小时，函数调用顺序是很重要的，因为 TCP 的窗口规模选项是在建立连接时用 SYN 与对方互换得到的。对于客户，SO_RCVBUF 选项必须在 connect 之前设置；对于服务器，SO_RCVBUF 选项必须在 listen 前设置。

10） MAKE_SOCKADDR_IN 构建 sockaddr_in 结构体宏

```
#ifdef HAVE_SOCKADDR_LEN
#define SET_SOCKADDR_SIN_LEN(var) var.sin_len = sizeof var
#else
#define SET_SOCKADDR_SIN_LEN(var)
#endif

// sockaddr_in var,使用 adr 和 prt 为其赋值(AF_INET)
#define MAKE_SOCKADDR_IN(var,adr,prt) /*adr,prt must be in network order*/\
    struct sockaddr_in var;\
    var.sin_family = AF_INET;\
    var.sin_addr.s_addr = (adr);\
    var.sin_port = (prt);\
    SET_SOCKADDR_SIN_LEN(var);
```

11） setupDatagramSocket 设置数据报套接口

setupDatagramSocket 函数有两个参数(UsageEnvironment& env, Port port)。根据 env 的 groupsockPriv 成员来确定重用标识。如果 groupsockPriv 为空则使用默认设置。

setupDatagramSocket 创建一个数据报形式的 socket 套接口，如果端口 port==0，且 ReceivingInterfaceAddr==INADDR_ANY 的时候，不进行绑定(bind)。否则若 port==0 的时候，绑定 ReceivingInterfaceAddr（不为 INADDR_ANY）和端口 0(内核选择端口)。不为 0 的时候绑定 INADDR_ANY(内核选择 IP)和端口 port。

绑定完成之后设置多播发生接口。如果全局的 SendingInterfaceAddr== INADDR_ANY 则不设置。

函数成功返回一个 socket 套接口句柄，失败返回-1。

```
// 设置数据报套接字
int setupDatagramSocket(UsageEnvironment& env, Port port) {
  // 初始化网络
```

```
  if (!initializeWinsockIfNecessary()) {
     socketErr(env, "Failed to initialize 'winsock': ");
     return -1;
  }
  // 创建数据报套接字
  int newSocket = createSocket(SOCK_DGRAM);
  if (newSocket < 0) {
     socketErr(env, "unable to create datagram socket: ");
     return newSocket;
  }
  // 获取 env 的 groupsockPriv 重新使用标识
  int reuseFlag = groupsockPriv(env)->reuseFlag;
  // 根据需要，为 env 释放 groupsockPriv 成员指向对象
  reclaimGroupsockPriv(env);

  // 设置允许重用本地地址和端口，reuseFlag 用来接受传出值
  if (setsockopt(newSocket, SOL_SOCKET, SO_REUSEADDR,
     (const char*)&reuseFlag, sizeof reuseFlag) < 0) {
     socketErr(env, "setsockopt(SO_REUSEADDR) error: ");
     closeSocket(newSocket);
     return -1;
  }

#if defined(__WIN32__) || defined(_WIN32)
  // Windoze doesn't properly handle SO_REUSEPORT or IP_MULTICAST_LOOP
  // win-doze 贬义,可能是由于操作系统 BUG 很多,而且运行速度慢,导致在运行的是后你会 DOZE(打瞌睡)
  // Windows 无法正确的处理 SO_REUSEPORT 或 IP_MULTICAST_LOOP
#else
#ifdef SO_REUSEPORT      //在定义了重新使用端口宏下设置
  if (setsockopt(newSocket, SOL_SOCKET, SO_REUSEPORT,
     (const char*)&reuseFlag, sizeof reuseFlag) < 0) {
     socketErr(env, "setsockopt(SO_REUSEPORT) error: ");
     closeSocket(newSocket);
     return -1;
  }
#endif

#ifdef IP_MULTICAST_LOOP //在定义了 IP 多播循环下设置
  const u_int8_t loop = 1;
  if (setsockopt(newSocket, IPPROTO_IP, IP_MULTICAST_LOOP,
     (const char*)&loop, sizeof loop) < 0) {
     socketErr(env, "setsockopt(IP_MULTICAST_LOOP) error: ");
     closeSocket(newSocket);
     return -1;
  }
#endif
#endif
  // Note: Windoze requires binding, even if the port number is 0
  // Windows 需要绑定，即使端口号是 0
  netAddressBits addr = INADDR_ANY;  // 设置绑定地址是任意网口 IP
#if defined(__WIN32__) || defined(_WIN32)
#else
  if (port.num() != 0 || ReceivingInterfaceAddr != INADDR_ANY) {
     //ReceivingInterfaceAddr 是一个全局的定义，默认是 INADDR_ANY
#endif
     if (port.num() == 0) addr = ReceivingInterfaceAddr;
     // 组建 sockaddr_in 结构体
     MAKE_SOCKADDR_IN(name, addr, port.num());
     // 绑定 socket 套接口和 sockaddr 地址
```

```
      if (bind(newSocket, (struct sockaddr*)&name, sizeof name) != 0) {
         char tmpBuffer[100];
         sprintf(tmpBuffer, "bind() error (port number: %d): ",
            ntohs(port.num()));
         socketErr(env, tmpBuffer);
         closeSocket(newSocket);
         return -1;
      }
#if defined(__WIN32__) || defined(_WIN32)
#else
   }
#endif

   // Set the sending interface for multicasts, if it's not the default:
   // 设置多播发送接口，如果它不是默认的:
   if (SendingInterfaceAddr != INADDR_ANY) {
      struct in_addr addr;
      addr.s_addr = SendingInterfaceAddr;
      // 设置多播接口
      if (setsockopt(newSocket, IPPROTO_IP, IP_MULTICAST_IF,
         (const char*)&addr, sizeof addr) < 0) {
         socketErr(env, "error setting outgoing multicast interface: ");
         closeSocket(newSocket);
         return -1;
      }
   }

   return newSocket;
}
```

12） makeSocketNonBlocking 和 makeSocketBlocking 套接口阻塞属性设置

makeSocketNonBlocking 函数用于为参数 sock 代表的套接口添加 O_NONBLOCK 非阻塞属性。

```
// 设置 sock 为非阻塞模式
Boolean makeSocketNonBlocking(int sock) {
#if defined(__WIN32__) || defined(_WIN32)
   unsigned long arg = 1;
   return ioctlsocket(sock, FIONBIO, &arg) == 0;
#elif defined(VXWORKS)
   int arg = 1;
   return ioctl(sock, FIONBIO, (int)&arg) == 0;
#else
   int curFlags = fcntl(sock, F_GETFL, 0);
   return fcntl(sock, F_SETFL, curFlags|O_NONBLOCK) >= 0;
#endif
}
```

makeSocketBlocking 函数用于为参数 sock 代表的套接口去除 O_NONBLOCK 非阻塞属性。

```
// 设置 sock 为阻塞模式
Boolean makeSocketBlocking(int sock) {
#if defined(__WIN32__) || defined(_WIN32)
   unsigned long arg = 0;
   return ioctlsocket(sock, FIONBIO, &arg) == 0;
#elif defined(VXWORKS)
   int arg = 0;
   return ioctl(sock, FIONBIO, (int)&arg) == 0;
#else
```

```
  int curFlags = fcntl(sock, F_GETFL, 0);
  return fcntl(sock, F_SETFL, curFlags&(~O_NONBLOCK)) >= 0;
#endif
}
```

13) setupStreamSocket 设置流式套接口

setupStreamSocket 和 setupDatagramSocket 的功能很像，区别在于这里返回的是一个流式套接口。

makeNonBlocking 参数用于控制创建的套接口是否是阻塞的。

```
// 设置流式套接字
int setupStreamSocket(UsageEnvironment& env,
  Port port, Boolean makeNonBlocking) {
  if (!initializeWinsockIfNecessary()) {
     socketErr(env, "Failed to initialize 'winsock': ");
     return -1;
  }

  int newSocket = createSocket(SOCK_STREAM);
  if (newSocket < 0) {
     socketErr(env, "unable to create stream socket: ");
     return newSocket;
  }

  int reuseFlag = groupsockPriv(env)->reuseFlag;
  reclaimGroupsockPriv(env);
  if (setsockopt(newSocket, SOL_SOCKET, SO_REUSEADDR,
     (const char*)&reuseFlag, sizeof reuseFlag) < 0) {
     socketErr(env, "setsockopt(SO_REUSEADDR) error: ");
     closeSocket(newSocket);
     return -1;
  }

  // SO_REUSEPORT doesn't really make sense for TCP sockets, so we
  // normally don't set them.  However, if you really want to do this
  // #define REUSE_FOR_TCP
#ifdef REUSE_FOR_TCP
#if defined(__WIN32__) || defined(_WIN32)
  // Windoze doesn't properly handle SO_REUSEPORT
#else
#ifdef SO_REUSEPORT
  if (setsockopt(newSocket, SOL_SOCKET, SO_REUSEPORT,
     (const char*)&reuseFlag, sizeof reuseFlag) < 0) {
     socketErr(env, "setsockopt(SO_REUSEPORT) error: ");
     closeSocket(newSocket);
     return -1;
  }
#endif
#endif
#endif

  // Note: Windoze requires binding, even if the port number is 0
#if defined(__WIN32__) || defined(_WIN32)
#else
  if (port.num() != 0 || ReceivingInterfaceAddr != INADDR_ANY) {
#endif
     MAKE_SOCKADDR_IN(name, ReceivingInterfaceAddr, port.num());
     if (bind(newSocket, (struct sockaddr*)&name, sizeof name) != 0) {
```

```
        char tmpBuffer[100];
        sprintf(tmpBuffer, "bind() error (port number: %d): ",
           ntohs(port.num()));
        socketErr(env, tmpBuffer);
        closeSocket(newSocket);
        return -1;
     }
#if defined(__WIN32__) || defined(_WIN32)
#else
  }
#endif
  // 根据参数设置是否为非阻塞
  if (makeNonBlocking) {
     if (!makeSocketNonBlocking(newSocket)) {
        socketErr(env, "failed to make non-blocking: ");
        closeSocket(newSocket);
        return -1;
     }
  }

  return newSocket;
}
```

14） readSocket 从套接口读取数据

readSocket 函数从套接口 socket 读取数据到 buffer，并捕获数据发送源的地址到 fromAddress。

函数返回读取到的字节数，出错时返回 0 并调用 socketErr(env, "recvfrom() error: ")来设置套接口错误消息。

```
// 从套接口读数据
int readSocket(UsageEnvironment& env,
  int socket, unsigned char* buffer, unsigned bufferSize,
struct sockaddr_in& fromAddress) {
  SOCKLEN_T addressSize = sizeof fromAddress;
  // ssize_t recvfrom(int sockfd,void *buf,int len,unsigned int flags, struct sockaddr *from,socket_t *
fromlen);
  // 读取主机经指定的 socket 传来的数据,并把数据传到由参数 buf 指向的内存空间,参数 len 为可接收数据的最大长度。fla
g 一般设置为 0。from 是来源地址，fromlen 传出来源长度
  // 如果正确接收返回接收到的字节数，失败返回-1.
  int bytesRead = recvfrom(socket, (char*)buffer, bufferSize, 0,
     (struct sockaddr*)&fromAddress,
     &addressSize);
  if (bytesRead < 0) {
     //##### HACK to work around bugs in Linux and Windows:
     int err = env.getErrno();
     if (err == 111 /*ECONNREFUSED (Linux) 连接请求被服务器拒绝*/
#if defined(__WIN32__) || defined(_WIN32)
        // What a piece of crap Windows is.  Sometimes
        // recvfrom() returns -1, but with an 'errno' of 0.
        // This appears not to be a real error; just treat
        // it as if it were a read of zero bytes, and hope
        // we don't have to do anything else to 'reset'
        // this alleged error:
        // 垃圾的 Windows。有时 recvfrom()返回- 1，但是有 errno 为 0。
        // 这似乎不是一个真正的错误；只是把它当作一个读取零字节，并希望我们不需要做什么“reset”
        // 这所谓的错误:
        || err == 0 || err == EWOULDBLOCK
#else
        || err == EAGAIN
```


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

```
#endif
        || err == 113 /*EHOSTUNREACH (Linux)*/) { // Why does Linux return this for datagram sock?
        fromAddress.sin_addr.s_addr = 0;
        return 0;
      }
      //##### END HACK
      socketErr(env, "recvfrom() error: ");
   }

   return bytesRead;
}
```

recv/recvfrom 函数

功能描述：

从套接字上接收一个消息。对于 recvfrom ，可同时应用于面向连接的和无连接的套接字。recv 一般只用在面向连接的套接字，几乎等同于 recvfrom，只要将 recvfrom 的第五个参数设置 NULL。

如果消息太大，无法完整存放在所提供的缓冲区，根据不同的套接字，多余的字节会丢弃。

假如套接字上没有消息可以读取，除了套接字已被设置为非阻塞模式，否则接收调用会等待消息的到来。

函数原型：

```
#include <sys/types.h>

#include <sys/socket.h>

ssize_t recv(int sock, void *buf, size_t len, int flags);

ssize_t recvfrom(int sock, void *buf, size_t len, int flags,

struct sockaddr *from, socklen_t *fromlen);
```

参数说明：

sock：索引将要从其接收数据的套接字。

buf：存放消息接收后的缓冲区。

len：buf 所指缓冲区的容量。

flags：是以下一个或者多个标志的组合体，可通过 or 操作连在一起

MSG_DONTWAIT：操作不会被阻塞。

MSG_ERRQUEUE：指示应该从套接字的错误队列上接收错误值，依据不同的协议，错误值以某种辅佐性消息的方式传递进来，使用者应该提供足够大的缓冲区。导致错误的原封包通过 msg_iovec 作为一般的数据来传递。导致错误的数据报原目标地址作为 msg_name 被提供。错误以 sock_extended_err 结构形态被使用，定义如下

```
#define SO_EE_ORIGIN_NONE    0
#define SO_EE_ORIGIN_LOCAL   1
#define SO_EE_ORIGIN_ICMP    2
#define SO_EE_ORIGIN_ICMP6   3
struct sock_extended_err
{
 u_int32_t ee_errno;   /* error number */
 u_int8_t ee_origin; /* where the error originated */
 u_int8_t ee_type;    /* type */
 u_int8_t ee_code;    /* code */
 u_int8_t ee_pad;
 u_int32_t ee_info;    /* additional information */
 u_int32_t ee_data;    /* other data */
```

```
  /* More data may follow */
};
```

MSG_PEEK：指示数据接收后，在接收队列中保留原数据，不将其删除，随后的读操作还可以接收相同的数据。

MSG_TRUNC：返回封包的实际长度，即使它比所提供的缓冲区更长， 只对 packet 套接字有效。

MSG_WAITALL：要求阻塞操作，直到请求得到完整的满足。然而，如果捕捉到信号，错误或者连接断开发生，或者下次被接收的数据类型不同，仍会返回少于请求量的数据。

MSG_EOR：指示记录的结束，返回的数据完成一个记录。

MSG_TRUNC：指明数据报尾部数据已被丢弃，因为它比所提供的缓冲区需要更多的空间。

MSG_CTRUNC：指明由于缓冲区空间不足，一些控制数据已被丢弃。

MSG_OOB：指示接收到 out-of-band 数据(即需要优先处理的数据)。

MSG_ERRQUEUE：指示除了来自套接字错误队列的错误外，没有接收到其它数据。

from：指向存放对端地址的区域，如果为 NULL，不储存对端地址。

fromlen：作为入口参数，指向存放表示 from 最大容量的内存单元。作为出口参数，指向存放表示 from 实际长度的内存单元。

返回说明：

成功执行时，返回接收到的字节数。另一端已关闭则返回 0。失败返回-1，errno 被设为以下的某个值

EAGAIN：套接字已标记为非阻塞，而接收操作被阻塞或者接收超时

EBADF：sock 不是有效的描述词

ECONNREFUSE：远程主机阻绝网络连接

EFAULT：内存空间访问出错

EINTR：操作被信号中断

EINVAL：参数无效

ENOMEM：内存不足

ENOTCONN：与面向连接关联的套接字尚未被连接上

ENOTSOCK：sock 索引的不是套接字

15) writeSocket 向套接口写数据

writeSocket 函数用于将 buffer 中的数据经 socket 套接口写入到目标主机(address + port)。参数 ttlArg 为 0 时被忽略，不为 0 时设置此处发送数据包的最大路由跳转次数。

```
// 往套接口写数据
Boolean writeSocket(UsageEnvironment& env,
  int socket, struct in_addr address, Port port,
  u_int8_t ttlArg,
  unsigned char* buffer, unsigned bufferSize)
{
  do {
     if (ttlArg != 0) {
        // TTL 是 Time To Live 的缩写，该字段指定 IP 包被路由器丢弃之前允许通过的最大网段数量。
        // TTL 的最大值是 255，TTL 的一个推荐值是 64。
        // Before sending, set the socket's TTL:发送前设置 socket TTL
#if defined(__WIN32__) || defined(_WIN32)
#define TTL_TYPE int
#else
```

```
#define TTL_TYPE u_int8_t
#endif
        TTL_TYPE ttl = (TTL_TYPE)ttlArg;
        // 设置多播 TTL 值
        if (setsockopt(socket, IPPROTO_IP, IP_MULTICAST_TTL,
           (const char*)&ttl, sizeof ttl) < 0) {
           socketErr(env, "setsockopt(IP_MULTICAST_TTL) error: ");
           break;
        }
     }

     MAKE_SOCKADDR_IN(dest, address.s_addr, port.num());
     // 将 buffer 中的数据经 socket 发送到 dest
     int bytesSent = sendto(socket, (char*)buffer, bufferSize, 0,
        (struct sockaddr*)&dest, sizeof dest);
     // 发送非全部发送成功
     if (bytesSent != (int)bufferSize) {
        char tmpBuf[100];
        sprintf(tmpBuf, "writeSocket(%d), sendTo() error: wrote %d bytes instead of %u: ", socket, byte
sSent, bufferSize);
        socketErr(env, tmpBuf);
        break;
     }

     return True; //发送成功返回
  } while (0);

  return False;   //失败返回
}
```

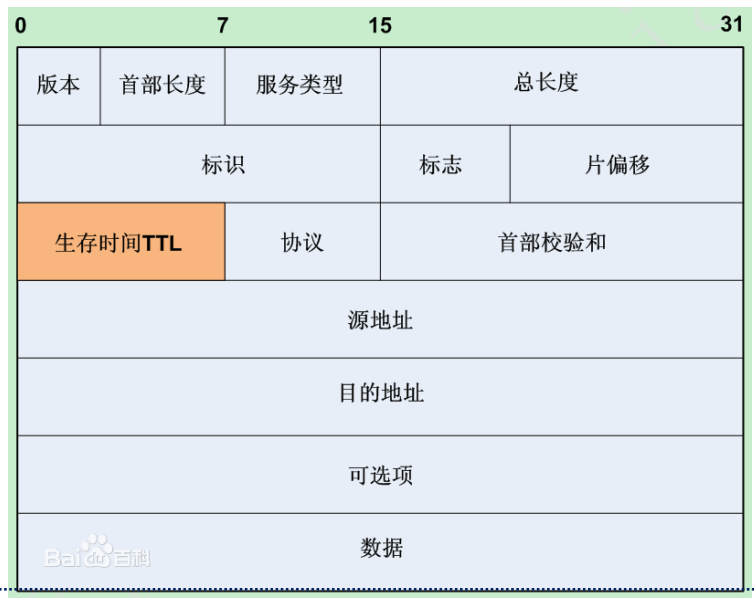
TTL 的概念

TTL 是 Time To Live 的缩写，该字段指定 IP 包被路由器丢弃之前允许通过的最大网段数量。在 IPv4 包头中 TTL 是一个 8 bit 字段，它位于 IPv4 包的第 9 个字节。

如右图所示，每一行表示 32 bit（4 字节），位从 0 开始编号，即 0~31。

TTL 的作用是限制 IP 数据包在计算机网络中的存在的时间。TTL 的最大值是 255，TTL 的一个推荐值是 64。

虽然 TTL 从字面上翻译，是可以存活的时间，但实际上 TTL 是 IP 数据包在计算机网络中可以转发的最大跳数。TTL 字段由 IP 数据包的发送者设置，在 IP 数据包从源到目的的整个转发路径上，每经过一个路由器，路由器都会修改这个 TTL 字段值，具体的做法是把该 TTL 的值减 1，然后再将 IP 包转发出去。如果在 IP 包到达目的 IP 之前，TTL 减少为 0，路由器将会丢弃收到的 TTL=0 的 IP 包并向 IP 包的发送者发送 ICMP time exceeded 消息。



TTL 的主要作用是避免 IP 包在网络中的无限循环和收发，节省了网络资源，并能使 IP 包的发送者能收到告警消息。

TTL 是由发送主机设置的，以防止数据包不断在 IP 互联网络上永不终止地循环。转发 IP 数据包时，要求路由器至少将 TTL 减小 1。

函数 sendto

```
int sendto ( socket s , const void * msg, int len, unsigned int flags, const
struct sockaddr * to , int tolen ) ;
```

函数说明

sendto() 用来将数据由指定的 socket 传给对方主机。

参数说明：

s　　为已建立连接的 socket,如果利用 UDP 协议则不需经过连线操作。

msg　　指向欲发送的数据内容

flags　　一般设 0，详细描述请参考 send()。

to　　用来指定目的套接字的地址，结构 sockaddr 请参考 bind()。

tolen　　为 sockaddr 的结构长度。

返回值

成功则返回实际传送出去的字符数，失败返回－1，错误原因存于 errno 中。

错误代码

EBADF 参数 s 非法的 socket 处理代码。

EFAULT 参数中有一指针指向无法存取的内存空间。

ENOTSOCK 参数 s 为一文件描述词，非 socket。

EINTR 被信号所中断。

EAGAIN 此动作会令进程阻断，但参数 s 的 socket 为不可阻断的。

ENOBUFS 系统的缓冲内存不足。

EINVAL 传给系统调用的参数不正确。

16） 套接字相关 buffer 操作函数

getBufferSize 获取 socket 相关缓冲区大小

getBufferSize 函数用于获取 socket 的相关缓冲区的 size 大小。参数 bufOptName 有四个可选值:SO_RCVBUF 接收缓冲区大小、SO_SNDBUF 发送缓冲区大小、SO_RCVLOWAT 接收缓冲区下限、SO_SNDLOWAT 发送缓冲区下限。

成功返回获取到的 size 值，失败返回 0，并设置 socket 错误消息到 env。

要注意这个函数是 static 的，只在后面介绍的函数中使用。

```
// 获取 bufferSize bufOptName
// SO_RCVBUF 接收缓冲区大小 SO_SNDBUF 发送缓冲区大小
// SO_RCVLOWAT 接收缓冲区下限 SO_SNDLOWAT 发送缓冲区下限
static unsigned getBufferSize(UsageEnvironment& env, int bufOptName,
  int socket) {
  unsigned curSize;
  SOCKLEN_T sizeSize = sizeof curSize;
  if (getsockopt(socket, SOL_SOCKET, bufOptName,
     (char*)&curSize, &sizeSize) < 0) {
     socketErr(env, "getBufferSize() error: ");
     return 0;
  }

  return curSize;
}
```

getSendBufferSize 与 getReceiveBufferSize

这两个函数是对前面 getBufferSize 的调用。注意，这两个没有 static 关键字修饰，适用于全局。

getSendBufferSize 用于获取 socket 的发送缓冲区 size 值。成功返回获取的 size，失败返回 0 并设置错误消息到 env。

```
// 获取发送缓冲区 size
unsigned getSendBufferSize(UsageEnvironment& env, int socket) {
  return getBufferSize(env, SO_SNDBUF, socket);
}
```

getReceiveBufferSize 用于获取 socket 的接收缓冲区 size 值。成功返回获取的 size，失败返回 0 并设置错误消息到 env。

```
// 获取接收缓冲区 size
unsigned getReceiveBufferSize(UsageEnvironment& env, int socket) {
  return getBufferSize(env, SO_RCVBUF, socket);
}
```

setBufferTo 设置 socket 缓冲区大小

setBufferTo 用于设置 socket 的相关缓冲区的 size，这个新的 size 由参数 requestedSize 指定。参数 bufOptName 与前面 getBufferSize 中的一样，用于指定操作哪一个 buffer。

函数返回设置后的 buffer 的 size 值。失败返回 0，并设置 socket 错误消息到 env。这也是一个 static 修饰的函数。

```
static unsigned setBufferTo(UsageEnvironment& env, int bufOptName,
  int socket, unsigned requestedSize) {
  SOCKLEN_T sizeSize = sizeof requestedSize;
  // 设置缓冲区大小
  setsockopt(socket, SOL_SOCKET, bufOptName, (char*)&requestedSize, sizeSize);

  // Get and return the actual, resulting buffer size:
  // 获取并返回实际的，缓冲区大小
  return getBufferSize(env, bufOptName, socket);
}
```

setSendBufferTo 和 setReceiveBufferTo

setSendBufferTo 用于设置 socket 的发送缓冲区的 size。成功返回设置后的 size，失败返回 0 并设置错误消息到 env。

```

// 设置发送缓冲区 size
unsigned setSendBufferTo(UsageEnvironment& env,
  int socket, unsigned requestedSize) {
  return setBufferTo(env, SO_SNDBUF, socket, requestedSize);
}
```

setReceiveBufferTo 用于设置 socket 的接收缓冲区的 size。成功返回设置后的 size，失败返回 0 并设置错误消息到 env。

```
//设置接收缓冲区 size
unsigned setReceiveBufferTo(UsageEnvironment& env,
  int socket, unsigned requestedSize) {
  return setBufferTo(env, SO_RCVBUF, socket, requestedSize);
}
```

increaseBufferTo 增长缓冲区 size

increaseBufferTo 用于将 bufOptName 指定的 buffer 的 size 增长到 requestedSize 大小。

如果当前的 buffer 的 size 已经大于或等于 requestedSize 了，那么就不增长了。如果增长失败，那么会将 requestedSize 设置为当前 size 到 requestedSize 的中间值，再去设置。如果又不成功，再缩小 requestedSize 去设置，迭代直至 setsockopt 函数成功。

函数成功返回操作后的 buffer 的 size，可能不会等于 requestedSize。

increase[英][ɪn'kri:s][美][ɪn'kris]

vt.& vi.增加，增大，增多；vt.增强，增进；[缝纫]放（针）；vi.增强；增进；增殖，繁殖；[缝纫]放针；

```
static unsigned increaseBufferTo(UsageEnvironment& env, int bufOptName,
  int socket, unsigned requestedSize) {
  // First, get the current buffer size.  If it's already at least
  // as big as what we're requesting, do nothing.
  // 获取当前的缓冲区大小
  unsigned curSize = getBufferSize(env, bufOptName, socket);

  // Next, try to increase the buffer to the requested size,
  // or to some smaller size, if that's not possible:
  // 当前的小于要达到的
  while (requestedSize > curSize) {
     SOCKLEN_T sizeSize = sizeof requestedSize;
     if (setsockopt(socket, SOL_SOCKET, bufOptName,
        (char*)&requestedSize, sizeSize) >= 0) {
        // success
        return requestedSize;
     }
     requestedSize = (requestedSize + curSize) / 2;
  }

  return getBufferSize(env, bufOptName, socket);
}
```

increaseSendBufferTo 和 increaseReceiveBufferTo

increaseSendBufferTo 增长 socket 发送缓冲区 size。

```
// 增长发送缓冲区 size
unsigned increaseSendBufferTo(UsageEnvironment& env,
  int socket, unsigned requestedSize) {
  return increaseBufferTo(env, SO_SNDBUF, socket, requestedSize);
}
```

increaseReceiveBufferTo 增长 socket 接收缓冲区 size。

```
// 增长接收缓冲区 size
unsigned increaseReceiveBufferTo(UsageEnvironment& env,
  int socket, unsigned requestedSize) {
  return increaseBufferTo(env, SO_RCVBUF, socket, requestedSize);
}
```

17） 套接字多播组操作函数

任意源多播(Any-Source Multicast)[ASM],即它只是关注多播组,而不关注是谁发送的,这样会有一些问题,比如说,假如一个局域网存在一个多播视频服务器,其它 Host 在上面点播视频;假如出现一个伪装者,也向这个多播组发送干扰数据,但 Host 无法辨别,这样可能引起视频的接受断断续续,影响服务的提供,显然这不是很理想;而 IGMPv3 的提出,就是为了解决这个问题,它提出了源特定多播(Source-Specific Multicast)[SSM],不仅关注多播组,也关注发送多播组的源。

SSM 的一个(S,G)对也被称为一个频道(Channel),以区分传统 PIM-SM 组播中的任意源组播组(ASM:Any Source Multicast)。由于 ASM 支持点到多点和多点到多点两种组播业务模式，因此源的发现过程是 ASM 复杂性的原因。例如在 PIM-SM 模式中，用户点击浏览器中的组播内容，接收端设备只被通知到组播组的内容，而没有被通知到组播源的信息。而在 SSM 模式中，用户端将同时接收到组播源和组播组信息。

socketJoinGroup 套接字加入一个不限源多播组

socketJoinGroup 将 socket 设置为多播组成员。

```
// 在本地 ReceivingInterfaceAddr 上加入一个多播组地址 groupAddress
Boolean socketJoinGroup(UsageEnvironment& env, int socket,
```

```
  netAddressBits groupAddress)
{ // 如果不是多播地址，直接忽略
  if (!IsMulticastAddress(groupAddress)) return True; // ignore this case
  /* struct ip_mreq {
        struct in_addr  imr_multiaddr;  // 多播组 IP 地址
     struct in_addr  imr_interface;     // 本地 IP 地址的接口
     };*/
  struct ip_mreq imr;
  imr.imr_multiaddr.s_addr = groupAddress;
  imr.imr_interface.s_addr = ReceivingInterfaceAddr;
  // 将指定的接口加入多播组
  if (setsockopt(socket, IPPROTO_IP, IP_ADD_MEMBERSHIP,
     (const char*)&imr, sizeof (struct ip_mreq)) < 0) {
#if defined(__WIN32__) || defined(_WIN32)
     if (env.getErrno() != 0) {
        // That piece-of-shit toy operating system (Windows) sometimes lies
        // about setsockopt() failing!
#endif
        socketErr(env, "setsockopt(IP_ADD_MEMBERSHIP) error: ");
        return False;
#if defined(__WIN32__) || defined(_WIN32)
     }
#endif
  }

  return True;
}
```

socketLeaveGroup 套接字离开不限源多播组

```
// 套接字离开多播组
Boolean socketLeaveGroup(UsageEnvironment&, int socket,
  netAddressBits groupAddress)
{
  if (!IsMulticastAddress(groupAddress)) return True; // ignore this case

  struct ip_mreq imr;
  imr.imr_multiaddr.s_addr = groupAddress;
  imr.imr_interface.s_addr = ReceivingInterfaceAddr;
  //离开本地接口上不限源的多播组
  if (setsockopt(socket, IPPROTO_IP, IP_DROP_MEMBERSHIP,
     (const char*)&imr, sizeof (struct ip_mreq)) < 0) {
     return False;
  }

  return True;
}
```

特定源的多播组定义

SSM（Source Specific Multicast），即“源特定多播”

```
// The source-specific join/leave operations require special setsockopt()
// commands, and a special structure (ip_mreq_source).  If the include files
// didn't define these, we do so here:
// 特定源的加入/离开操作需要特殊的 setsockopt()命令，和一个特殊的结构体（ip_mreq_source）。
// 如果文件没有定义这些，我们这样做：
#if !defined(IP_ADD_SOURCE_MEMBERSHIP)
```


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

```
struct ip_mreq_source {
  struct  in_addr imr_multiaddr;  /* IP multicast address of group */
  struct  in_addr imr_sourceaddr; /* IP address of source */
  struct  in_addr imr_interface;  /* local IP address of interface */
};
#endif

#ifndef IP_ADD_SOURCE_MEMBERSHIP

#ifdef LINUX
#define IP_ADD_SOURCE_MEMBERSHIP   39
#define IP_DROP_SOURCE_MEMBERSHIP 40
#else
#define IP_ADD_SOURCE_MEMBERSHIP   25
#define IP_DROP_SOURCE_MEMBERSHIP 26
#endif

#endif
```

socketJoinGroupSSM 套接字加入到特定源多播组

```
// 在指定的本地接口上加入一个特定源的多播组
Boolean socketJoinGroupSSM(UsageEnvironment& env, int socket,
  netAddressBits groupAddress,
  netAddressBits sourceFilterAddr)
{
  if (!IsMulticastAddress(groupAddress)) return True; // ignore this case

  struct ip_mreq_source imr;
  imr.imr_multiaddr.s_addr = groupAddress;
  imr.imr_sourceaddr.s_addr = sourceFilterAddr;
  imr.imr_interface.s_addr = ReceivingInterfaceAddr;
  if (setsockopt(socket, IPPROTO_IP, IP_ADD_SOURCE_MEMBERSHIP,
     (const char*)&imr, sizeof (struct ip_mreq_source)) < 0) {
     socketErr(env, "setsockopt(IP_ADD_SOURCE_MEMBERSHIP) error: ");
     return False;
  }

  return True;
}
```

socketLeaveGroupSSM 套接字离开到特定源多播组

```
// 离开一个特定源的多播组
Boolean socketLeaveGroupSSM(UsageEnvironment& /*env*/, int socket,
  netAddressBits groupAddress,
  netAddressBits sourceFilterAddr)
{
  if (!IsMulticastAddress(groupAddress)) return True; // ignore this case

  struct ip_mreq_source imr;
  imr.imr_multiaddr.s_addr = groupAddress;
  imr.imr_sourceaddr.s_addr = sourceFilterAddr;
  imr.imr_interface.s_addr = ReceivingInterfaceAddr;
  if (setsockopt(socket, IPPROTO_IP, IP_DROP_SOURCE_MEMBERSHIP,
     (const char*)&imr, sizeof (struct ip_mreq_source)) < 0) {
     return False;
  }
```

```
  return True;
}
```

18) getSourcePort 获取 socket 本地关联端口

getSourcePort 获取 socket 本地关联端口，如果没有本地关联端口，则为其 bind 一个内核分配的端口。

```
// 获取 socket 的本地关联端口
static Boolean getSourcePort0(int socket, portNumBits& resultPortNum/*host order*/)
{
  sockaddr_in test; test.sin_port = 0;
  SOCKLEN_T len = sizeof test;
  // getsockname 函数用于获取与某个套接字关联的本地协议地址
  if (getsockname(socket, (struct sockaddr*)&test, &len) < 0) return False;

  resultPortNum = ntohs(test.sin_port);
  return True;
}
```

```
// 获取 socket 本地关联端口，若是没有，则调用 bind 由系统内核分配一个
Boolean getSourcePort(UsageEnvironment& env, int socket, Port& port)
{
  portNumBits portNum = 0;
  if (!getSourcePort0(socket, portNum) || portNum == 0) {
      // Hack - call bind(), then try again:
      MAKE_SOCKADDR_IN(name, INADDR_ANY, 0);
      // socket 为关联本地端口，进行 bind 操作
      bind(socket, (struct sockaddr*)&name, sizeof name);

      if (!getSourcePort0(socket, portNum) || portNum == 0) {
          socketErr(env, "getsockname() error: ");
          return False;
      }
  }

  port = Port(portNum);
  return True;
}
```

19) ourIPAddress 获取本机 IP 地址

这个函数有点复杂，比较长。

首先是自身先发送一个多播包，再去接收它，来获取源地址，就是主机地址。接收的时候设置的阻塞等待超时时间是 5 秒。如果这一招失败了，那么就指定一个 IP 地址。指定的 IP 地址使用 gethostname 函数来获取主机名，然后使用 NetAddressList 类的构造来获取主机对应的地址列表。从地址列表中选择第一个没有问题的 IP 地址作为返回值。

在上面获取到 IP 之后，使用这个 IP 地址，和当前时间，初始化我们自己定义的随机数生成器(our_srandom(seed))的种子。

```
Boolean loopbackWorks = 1;  //环回工作标识

netAddressBits ourIPAddress(UsageEnvironment& env)
{
  static netAddressBits ourAddress = 0;
  int sock = -1;
  struct in_addr testAddr;
```


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

```
  if (ourAddress == 0) {
     // We need to find our source address
     // 我们需要找到我们的源地址
     struct sockaddr_in fromAddr;
     fromAddr.sin_addr.s_addr = 0;

     // Get our address by sending a (0-TTL) multicast packet,
     // receiving it, and looking at the source address used.
     // (This is kinda bogus, but it provides the best guarantee
     // that other nodes will think our address is the same as we do.)
     do {
        loopbackWorks = 0; // until we learn otherwise 直到我们获知,否则...

        testAddr.s_addr = our_inet_addr("228.67.43.91"); // arbitrary 任意
        Port testPort(15947); // ditto
        // 创建数据报 socket
        sock = setupDatagramSocket(env, testPort);
        if (sock < 0) break;
        // 加入任意源多播组"228.67.43.91"
        if (!socketJoinGroup(env, sock, testAddr.s_addr)) break;

        unsigned char testString[] = "hostIdTest";
        unsigned testStringLength = sizeof testString;
        // 向 socket 写入数据"hostIdTest"
        if (!writeSocket(env, sock, testAddr, testPort, 0,
           testString, testStringLength)) break;

        // Block until the socket is readable (with a 5-second timeout):
        // 阻塞直到套接字可读（5 秒超时）:
        fd_set rd_set;
        FD_ZERO(&rd_set);
        FD_SET((unsigned)sock, &rd_set);
        const unsigned numFds = sock + 1;
        struct timeval timeout;
        timeout.tv_sec = 5;
        timeout.tv_usec = 0;
        // 阻塞等待 sock 变为可读
        int result = select(numFds, &rd_set, NULL, NULL, &timeout);
        if (result <= 0) break;

        unsigned char readBuffer[20];
        // 从 sock 读取数据，捕获来源主机地址到 fromAddr
        int bytesRead = readSocket(env, sock,
           readBuffer, sizeof readBuffer,
           fromAddr);
        if (bytesRead != (int)testStringLength
           || strncmp((char*)readBuffer, (char*)testString, testStringLength) != 0) {
           break;
        }

        loopbackWorks = 1;
     } while (0);

     if (sock >= 0) {
        // 离开多播组
        socketLeaveGroup(env, sock, testAddr.s_addr);
        closeSocket(sock);
     }
```

```
    // 前面 do while 里面的操作没有都成功
    if (!loopbackWorks) do {
        // We couldn't find our address using multicast loopback,
        // so try instead to look it up directly - by first getting our host name, and then resolving t
his host name
        char hostname[100];
        hostname[0] = '\0';
        // 该函数把本地主机名存放入由 hostname 参数指定的缓冲区中。
        int result = gethostname(hostname, sizeof hostname);
        if (result != 0 || hostname[0] == '\0') {
            env.setResultErrMsg("initial gethostname() failed");
            break;
        }

        // Try to resolve "hostname" to an IP address:
        // 尝试解决 hostname 到 IP 地址
        NetAddressList addresses(hostname);
        NetAddressList::Iterator iter(addresses);
        NetAddress const* address;

        // Take the first address that's not bad:
        // 采取的第一个无错的地址
        netAddressBits addr = 0;
        while ((address = iter.nextAddress()) != NULL) {
            netAddressBits a = *(netAddressBits*)(address->data());
            if (!badAddressForUs(a)) {
                addr = a;
                break;
            }
        }

        // Assign the address that we found to "fromAddr" (as if the 'loopback' method had worked), to
simplify the code below:
        // 指定的地址，我们发现 fromaddr（如环回法工作了），简化代码如下：
        fromAddr.sin_addr.s_addr = addr;
    } while (0);

    // Make sure we have a good address:确保我们有一个良好的地址
    netAddressBits from = fromAddr.sin_addr.s_addr;
    if (badAddressForUs(from)) {
        char tmp[100];
        sprintf(tmp, "This computer has an invalid IP address: %s", AddressString(from).val());
        env.setResultMsg(tmp);
        from = 0;
    }

    ourAddress = from;

    // Use our newly-discovered IP address, and the current time,
    // to initialize the random number generator's seed:
    // 使用我们的新发现的 IP 地址，和当前时间，初始化随机数生成器的种子：
    struct timeval timeNow;
    gettimeofday(&timeNow, NULL);
    unsigned seed = ourAddress^timeNow.tv_sec^timeNow.tv_usec;
    our_srandom(seed);
  }
  return ourAddress; // 返回我们的 IP 地址
}
```


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

20） chooseRandomIPv4SSMAddress 特定源随机 IPv4 多播地址

chooseRandomIPv4SSMAddress 函数用来产生一个特定源的 IPv4 随机多播地址。

函数先调用了 ourIPAddress 来初始化随机数生成器的种子。

```
// 选择随机 IPv4 SSM 地址
netAddressBits chooseRandomIPv4SSMAddress(UsageEnvironment& env)
{
  // First, a hack to ensure that our random number generator is seeded:
  // 首先，一个黑客，确保我们的随机数生成器种子的初始化：
  (void)ourIPAddress(env);

  // Choose a random address in the range [232.0.1.0, 232.255.255.255)
  // i.e., [0xE8000100, 0xE8FFFFFF)
  netAddressBits const first = 0xE8000100, lastPlus1 = 0xE8FFFFFF;
  netAddressBits const range = lastPlus1 - first; //范围

  return ntohl(first + ((netAddressBits)our_random()) % range);
}
```

21） timestampString 获取当前时间戳字符串

timestampString 用于获取当前时间戳的字符串形式。函数内部定义了一个 static timeString[size]的数组，其不是可重入函数。在 WINCE 相关操作系统下，保存的时间戳字符串形式为“sec.usec”（如：1234.567），在其他操作系统下，时间戳字符串保存为”hh:mm:ss”形式。

```
// 时间戳字符串
char const* timestampString()
{
  struct timeval tvNow;
  gettimeofday(&tvNow, NULL);  //获取当前时间

#if !defined(_WIN32_WCE)
  static char timeString[9]; // holds hh:mm:ss plus trailing '\0'
  char const* ctimeResult = ctime((time_t*)&tvNow.tv_sec);
  if (ctimeResult == NULL) {
     sprintf(timeString, "??:??:??");
  }
  else {
     char const* from = &ctimeResult[11];
     int i;
     for (i = 0; i < 8; ++i) {
        timeString[i] = from[i];
     }
     timeString[i] = '\0';
  }
#else
  // WinCE apparently doesn't have "ctime()", so instead, construct
  // a timestamp string just using the integer and fractional parts
  // of "tvNow":
  static char timeString[50];
  sprintf(timeString, "%lu.%06ld", tvNow.tv_sec, tvNow.tv_usec);
#endif

  return (char const*)&timeString;
}
```

对于 Windows，我们需要自己实现 gettimeofday()

```
#if defined(__WIN32__) || defined(_WIN32)
// For Windoze, we need to implement our own gettimeofday()
#if !defined(_WIN32_WCE)
#include <sys/timeb.h>
#endif

int gettimeofday(struct timeval* tp, int* /*tz*/) {
#if defined(_WIN32_WCE)
  /* FILETIME of Jan 1 1970 00:00:00. */
  static const unsigned __int64 epoch = 116444736000000000LL;

  FILETIME    file_time;
  SYSTEMTIME  system_time;
  ULARGE_INTEGER ularge;

  GetSystemTime(&system_time);
  SystemTimeToFileTime(&system_time, &file_time);
  ularge.LowPart = file_time.dwLowDateTime;
  ularge.HighPart = file_time.dwHighDateTime;

  tp->tv_sec = (long) ((ularge.QuadPart - epoch) / 10000000L);
  tp->tv_usec = (long) (system_time.wMilliseconds * 1000);
#else
  static LARGE_INTEGER tickFrequency, epochOffset;

  // For our first call, use "ftime()", so that we get a time with a proper epoch.
  // For subsequent calls, use "QueryPerformanceCount()", because it's more fine-grain.
  static Boolean isFirstCall = True;

  LARGE_INTEGER tickNow;
  QueryPerformanceCounter(&tickNow);

  if (isFirstCall) {
     struct timeb tb;
     ftime(&tb);
     tp->tv_sec = tb.time;
     tp->tv_usec = 1000 * tb.millitm;

     // Also get our counter frequency:
     QueryPerformanceFrequency(&tickFrequency);

     // And compute an offset to add to subsequent counter times, so we get a proper epoch:
     epochOffset.QuadPart
        = tb.time*tickFrequency.QuadPart + (tb.millitm*tickFrequency.QuadPart) / 1000 - tickNow.QuadPa
rt;

     isFirstCall = False; // for next time
  }
  else {
     // Adjust our counter time so that we get a proper epoch:
     tickNow.QuadPart += epochOffset.QuadPart;

     tp->tv_sec = (long)(tickNow.QuadPart / tickFrequency.QuadPart);
     tp->tv_usec = (long)(((tickNow.QuadPart % tickFrequency.QuadPart) * 1000000L) / tickFrequency.Quad
Part);
  }
#endif
```


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

```
  return 0;
}
```

22） socketReadHandler 套接口读处理

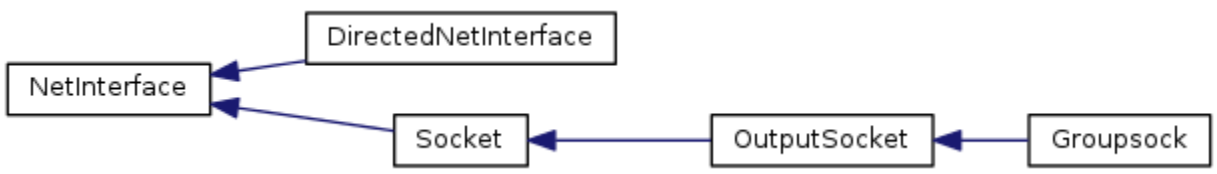
socketReadHandler 函数用于处理套接字传入的数据，其声明在文件 live555sourcecontrol\groupsock\include\IOHandlers.hh 中。

socketReadHandler 从 fromAddress 读取数据到 ioBuffer 中，注意这里 ioBuffer 是 static 数组。

```
#include "IOHandlers.hh"
#include "TunnelEncaps.hh"

//##### TEMP: Use a single buffer, sized for UDP tunnels:
//##### TEMP: 使用单个缓冲区，大小为 UDP 隧道：
//##### This assumes that the I/O handlers are non-reentrant
//#####这假定 I / O 处理程序是不可重入
static unsigned const maxPacketLength = 50 * 1024; // bytes
// This is usually overkill, because UDP packets are usually no larger
// than the typical Ethernet MTU (1500 bytes).  However, I've seen
// reports of Windows Media Servers sending UDP packets as large as
// 27 kBytes.  These will probably undego lots of IP-level
// fragmentation, but that occurs below us.  We just have to hope that
// fragments don't get lost.
// 这通常是矫枉过正，因为 UDP 数据包比典型的以太网 MTU（1500 字节）通常不会较大。不过，我见过 Windows Media 服务器发送 UDP 数据包的报告大小为 27KB 的。
// 上述很可能发生大量的 IP 级别的分片，而出现在我们下面。我们只是希望，片段不要丢失。

static unsigned const ioBufferSize/*IO 缓冲区尺寸=最大包长度+隧道最大尺寸*/
= maxPacketLength + TunnelEncapsulationTrailerMaxSize;
static unsigned char ioBuffer[ioBufferSize];   /*IO 缓冲区*/


void socketReadHandler(Socket* sock, int /*mask*/)
{
  unsigned bytesRead;
  struct sockaddr_in fromAddress;
  UsageEnvironment& saveEnv = sock->env();  //备份 env 的引用
  // because handleRead(), if it fails, may delete "sock"
  // 因为如果 handleRead()失败，可能会删除“_sock”

  // 从 sock 指定目标 fromAddress 获取数据到 iobuffer(注意 iobuffer 是 static 的，只在此处使用)
  if (!sock->handleRead(ioBuffer, ioBufferSize, bytesRead, fromAddress)) {
     saveEnv.reportBackgroundError();   //报告错误
  }
}
```

5.网络接口相关类

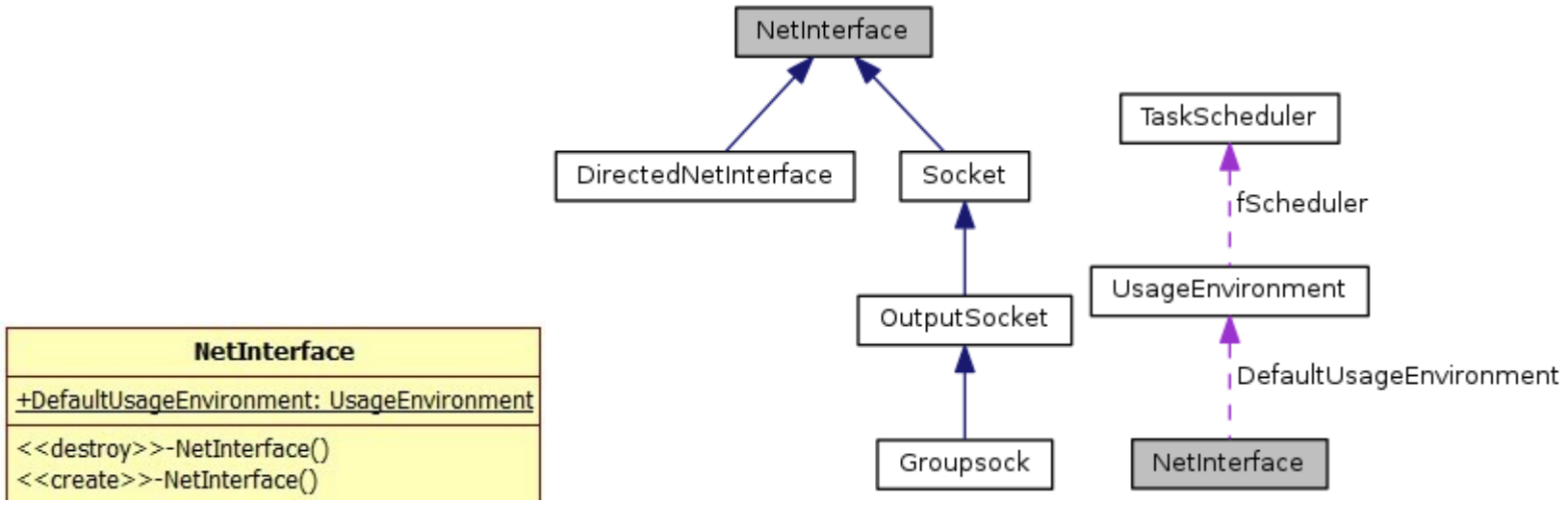
D:\work\live555sourcecontrol\groupsock\include\NetInterface.hh



1) NetInterface 网络接口类

这是一个基类，用于给派生类对象提供一个共享的 UsageEnvironment*成员。它的构造函数是 protected 权限的。

静态成员 static UsageEnvironment* DefaultUsageEnvironment 在 cpp 文件中初始化为 NULL。



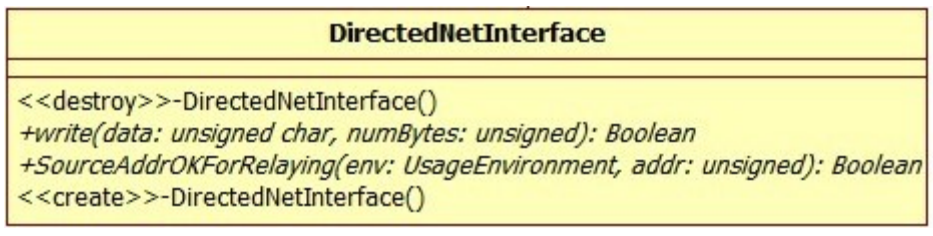

```
class NetInterface {
public:
  virtual ~NetInterface();

  static UsageEnvironment* DefaultUsageEnvironment;
  // if non-NULL, used for each new interfaces
  // 如果非 NULL，用于每一个新接口
protected:
  NetInterface(); // virtual base class
};
```

NetInterface 构造析构及 static 成员初始化

```
UsageEnvironment* NetInterface::DefaultUsageEnvironment = NULL;

NetInterface::NetInterface() {
}

NetInterface::~NetInterface() {
}
```

2) DirectedNetInterface 有向网络接口

DirectedNetInterface 类是个纯虚基类，但是这里却找不到有其子类的定义。其构造和析构都是空的，这里还不清楚其作用，先留下。



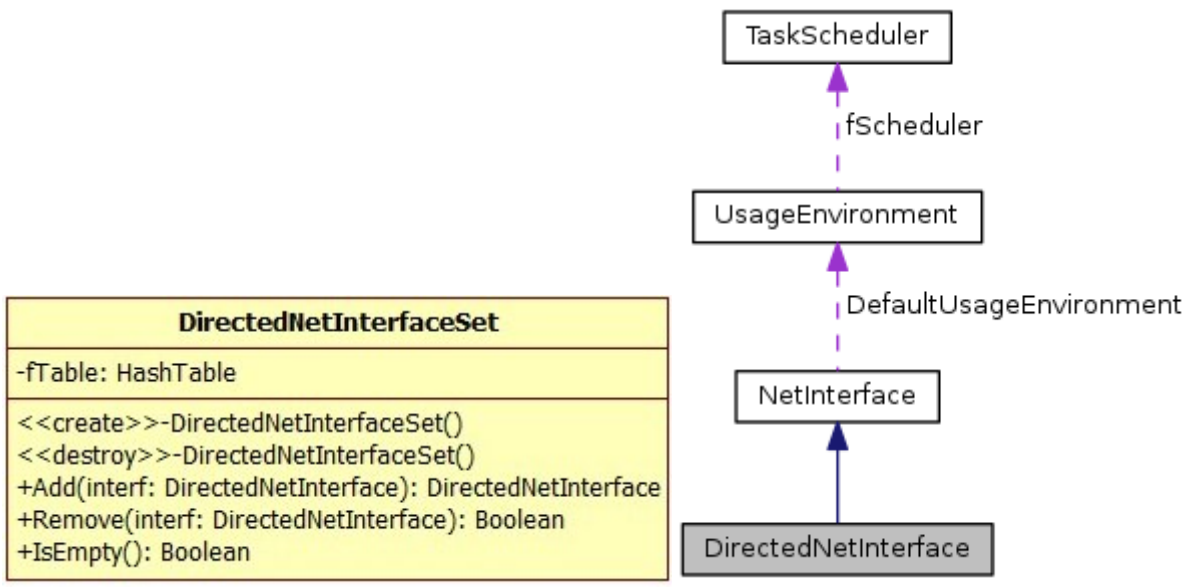

```
class DirectedNetInterface : public NetInterface {
public:
  virtual ~DirectedNetInterface();

  virtual Boolean write(unsigned char* data, unsigned numBytes) = 0;

  virtual Boolean SourceAddrOKForRelaying(UsageEnvironment& env,
```

```
      unsigned addr) = 0;

protected:
  DirectedNetInterface(); // virtual base class
};
```

3) DirectedNetInterfaceSet 有向网络接口接口集

DirectedNetInterfaceSet 有向网络接口接口集内部定义了一个成员 HashTable* fTable。哈希表在构造的时候创建，在析构的时候释放。添加条目的 key 和 value 是同一个 DirectedNetInterface const* interf。

这个类和之前的 NetAddressList 类十分像，这里就不多做解释了。




```
class DirectedNetInterfaceSet {
public:
  // 创建哈希表，键类型为 ONE_WORD_HASH_KEYS(u_intptr_t)
  DirectedNetInterfaceSet();
  // 销毁哈希表
  virtual ~DirectedNetInterfaceSet();

  // 添加条目到哈希表，参数 interf 即作为 key 又作为 value
  DirectedNetInterface* Add(DirectedNetInterface const* interf);
  // Returns the old value if different, otherwise 0

  Boolean Remove(DirectedNetInterface const* interf);

  Boolean IsEmpty() { return fTable->IsEmpty(); }

  // Used to iterate through the interfaces in the set
  class Iterator {
  public:
     Iterator(DirectedNetInterfaceSet& interfaces);
     virtual ~Iterator();

     DirectedNetInterface* next(); // NULL iff none

  private:
     HashTable::Iterator* fIter;
  };

private:
  friend class Iterator;
  HashTable* fTable;     // 保存在哈希表
```

```
};
```

DirectedNetInterfaceSet 构造与析构

构造的时候创建哈希表，析构的时候释放哈希表。

```
DirectedNetInterfaceSet::DirectedNetInterfaceSet()
  : fTable(HashTable::create(ONE_WORD_HASH_KEYS)) {
}
```

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

Add 和 Remove 方法

Add 方法用于向哈希表中添加条目，参数 interf 即作为 key，又作为 value。返回旧的 value 或者 NULL。

```
DirectedNetInterface*
DirectedNetInterfaceSet::Add(DirectedNetInterface const* interf) {
  return (DirectedNetInterface*) fTable->Add((char*)interf, (void*)interf);
}
```

Remove 方法用于从哈希表中移除条目。

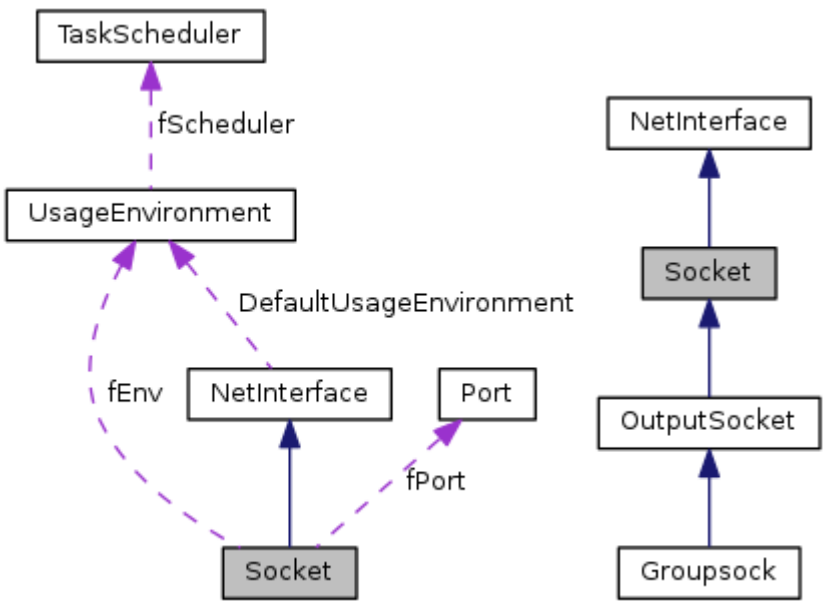
```
Boolean
DirectedNetInterfaceSet::Remove(DirectedNetInterface const* interf) {
  return fTable->Remove((char*)interf);
}
```

4）Socket 套接口类

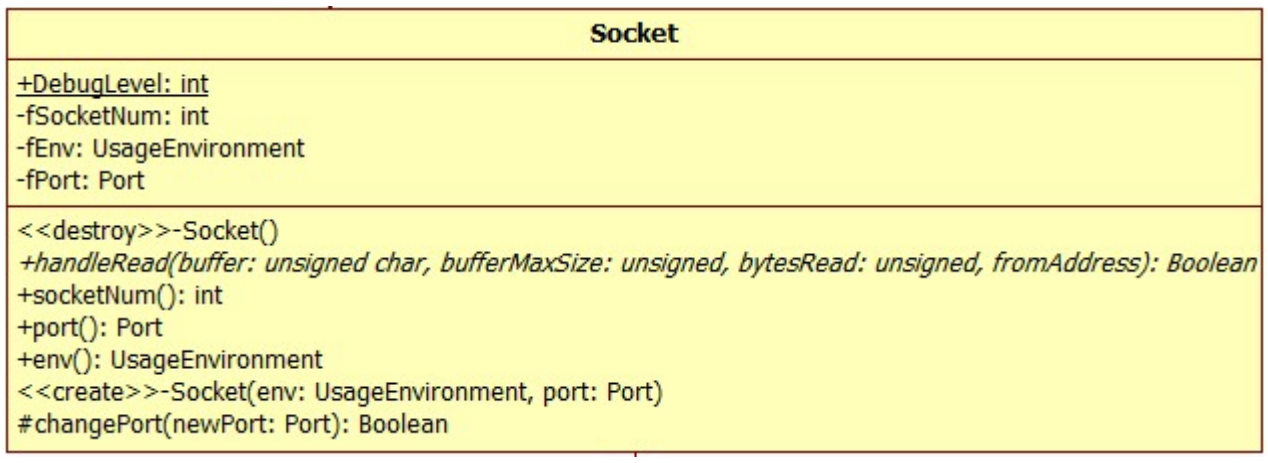
Socket类是一个抽象基类。相较于其基类NetInterFace多了三个数据成员

```
int fSocketNum;        // Socket套接口
UsageEnvironment& fEnv;  // 使用环境，这是一个引用
Port fPort;            // 端口
```

除此之外还要一个静态成员`static int DebugLevel;  // Debug等级`



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



类定义代码如下

```
class Socket : public NetInterface {
public:
  virtual ~Socket();
  // 读处理。返回 false 表示出错
  virtual Boolean handleRead(unsigned char* buffer, unsigned bufferMaxSize,
     unsigned& bytesRead, struct sockaddr_in& fromAddress) = 0;
  // Returns False on error; resultData == NULL if data ignored

  int socketNum() const { return fSocketNum; }

  Port port() const {
     return fPort;
  }

  UsageEnvironment& env() const { return fEnv; }

  static int DebugLevel; // Debug 等级

protected:
  // 设定使用环境和端口
  Socket(UsageEnvironment& env, Port port); // virtual base class

  // 改变端口号
  Boolean changePort(Port newPort); // will also cause socketNum() to change

private:
  int fSocketNum;        // Socket 套接口
  UsageEnvironment& fEnv;   // 使用环境
  Port fPort;            // 端口
};
```

Socket 的构造与析构

构造的时候使用基类中定义的 DefaultUsageEnvironment 静态成员来初始化 fEnv。如果 DefaultUsageEnvironment==NULL，则使用参数 env 来初始化 fEnv。

port 用来初始化成员 fPort，并用于创建一个数据报套接字来初始化 fSocketNum。

要注意的是，构造函数是 protected 权限的。

```
Socket::Socket(UsageEnvironment& env, Port port)
: fEnv(DefaultUsageEnvironment != NULL ? *DefaultUsageEnvironment : env), fPort(port) {
  // 创建了一个数据报套接字
  fSocketNum = setupDatagramSocket(fEnv, port);
}
```

析构的时候关闭 fSocketNum 套接字。

```
Socket::~Socket() {
  closeSocket(fSocketNum);
}
```

changePort 改变端口号

changePort 方法用于改变 Socket 对象的 fPort 成员。

首先关闭了 fSocketNum 成员代表的套接口，并使用新的 port 创建一个数据报套接字给 fSocketNum，如果创建失败了，就从任务调度器中将原来处理 fSocketNum 的程序节点移除，否则就更新它操作的套接口。

```
Boolean Socket::changePort(Port newPort) {
  int oldSocketNum = fSocketNum;
  closeSocket(fSocketNum);
  // 使用新端口创建一个数据报套接字
  fSocketNum = setupDatagramSocket(fEnv, newPort);
  // 创建失败
  if (fSocketNum < 0) {
     // 将 oldSocketNum 标识的处理程序节点从链表移除
     fEnv.taskScheduler().turnOffBackgroundReadHandling(oldSocketNum);
     return False;
  }

  if (fSocketNum != oldSocketNum) { // the socket number has changed, so move any event handling for it:
     // 将原本对 oldSocketNum 套接口操作的处理程序转移到去操作 newSocketNum 套接口
     fEnv.taskScheduler().moveSocketHandling(oldSocketNum, fSocketNum);
  }
  return True;
}
```

重载 operator<<(UsageEnvironment& s, const Socket& sock)

全局的 UsageEnvironment& operator<<(UsageEnvironment& s, const Socket& sock)重载。

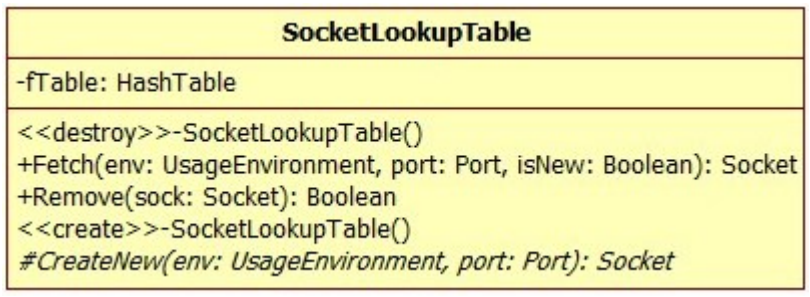
用于输出 Socket 类对象的 fSocketNum 成员值，以及当前的时间戳。

```
UsageEnvironment& operator<<(UsageEnvironment& s, const Socket& sock) {
  return s << timestampString() << " Socket(" << sock.socketNum() << ")";
}
```

5) SocketLookupTable 套接字查找表类

SocketLookupTable 类和之前说过的 AddressPortLookupTable 类非常像。要注意一点，这是一个纯虚基类。

SocketLookupTable 只定义了一个数据成员 Hash* fTable，用于管理 Socket 对象。




```
class SocketLookupTable {
public:
  virtual ~SocketLookupTable();
  // 使用 port 来查找对应的 Socket，没有找到就创建一个。isNew 是传出参数，指示返回的是否为新创建的
  Socket* Fetch(UsageEnvironment& env, Port port, Boolean& isNew);
  // Creates a new Socket if none already exists
  // 创建一个新的 Socket 如果不是已经存在
```

```
  // 从哈希表中移除 sock
  Boolean Remove(Socket const* sock);

protected:
  SocketLookupTable(); // abstract base class
  virtual Socket* CreateNew(UsageEnvironment& env, Port port) = 0;

private:
  HashTable* fTable;
};
```

SocketlookupTable 构造与析构

从构造函数中可用看出来，其构造哈希表的时候确定的键类型是一个 u_intptr_t 的整数。

```
SocketLookupTable::SocketLookupTable()
: fTable(HashTable::create(ONE_WORD_HASH_KEYS)) {
}
```

析构的时候是否哈希表

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

Fetch 方法(查找 port 对应 Socket)

Fetch 方法用于查找参数 port 来对应的 Socket 对象，没有找到就创建一个新的 Socket，如果创建失败，函数返回 NULL。isNew 是传出参数，指示返回的 socket 是否为新创建的。

这里有一个有趣的地方 sock = CreateNew(env, port)这一句中 CreateNew 方法的实现找不到。这个类没有派生类。

```
Socket* SocketLookupTable::Fetch(UsageEnvironment& env, Port port,
  Boolean& isNew) {
  isNew = False;
  Socket* sock;
  do {
     sock = (Socket*)fTable->Lookup((char*)(long)(port.num()));
     if (sock == NULL) { // we need to create one:
        sock = CreateNew(env, port);
        if (sock == NULL || sock->socketNum() < 0) break;

        fTable->Add((char*)(long)(port.num()), (void*)sock);
        isNew = True;
     }

     return sock;
  } while (0);

  delete sock;
  return NULL;
}
```

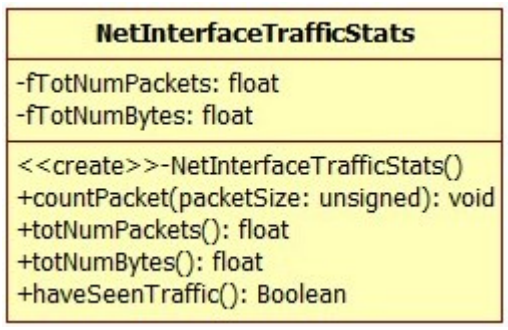
Remove 方法(从哈希表中移除 sock)

```
Boolean SocketLookupTable::Remove(Socket const* sock) {
  return fTable->Remove((char*)(long)(sock->port().num()));
}
```

6) NetInterfaceTrafficStats 网络接口流量统计类

NetInterfaceTrafficStats 类定义了两个数据成员，float fTotNumPackets(total number packets 总包数)和 fTotNumBytes(total number bytes 总字节数)。注意这两个成员都是 float 类型。

在构造的时候将这两个成员都初始化为 0。




```
class NetInterfaceTrafficStats {
public:
  //构造，初始化数据成员为 0
  NetInterfaceTrafficStats();
  // 统计 packet
  void countPacket(unsigned packetSize);

  float totNumPackets() const { return fTotNumPackets; }
  float totNumBytes() const { return fTotNumBytes; }
  // 以及统计过流量
  Boolean haveSeenTraffic() const;

private:
  float fTotNumPackets; //总包数 total Number packets
  float fTotNumBytes;   //总字节数
};
```

countPacket 方法(包计数)

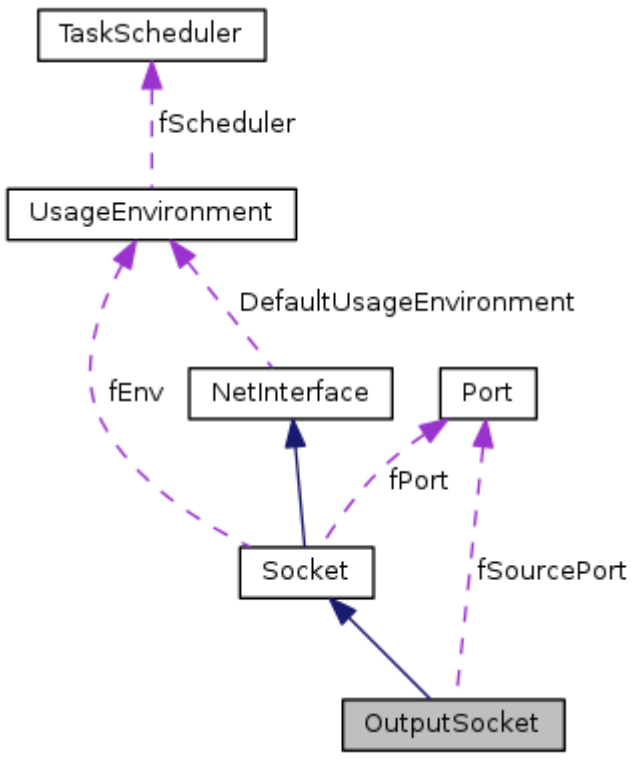
countPacket 方法计数一个新的流量包，参数 packetSize 为统计包的字节数。

```
void NetInterfaceTrafficStats:: countPacket (unsigned packetSize) {
  fTotNumPackets += 1.0;
  fTotNumBytes += packetSize;
}
```

7) OutputSocket 仅输出套接字类

OutputSocket 类是 Socket 类的派生类，其仅用于发送(输出)数据包。

OutputSocket 重写了 Socket 类中的纯虚接口 handleRead，其不是一个抽象类。OutputSocket 定义了 write 方法用于发送数据包，但是对应 handleRead 方法的实现，是什么也没做，所以其仅用于输出数据。(可以在其派生类中进行定义)



OutputSocket
-fSourcePort: Port -fLastSentTTL: u_int8_t
<<create>>-OutputSocket(env: UsageEnvironment) <<destroy>>-OutputSocket() +write(address: netAddressBits, port: Port, ttl: u_int8_t, buffer: unsigned char, bufferSize: unsigned): Boolean <<create>>-OutputSocket(env: UsageEnvironment, port: Port) #sourcePortNum(): portNumBits -handleRead(buffer: unsigned char, bufferMaxSize: unsigned, bytesRead: unsigned, fromAddress): Boolean

OutputSocket 定义如下

```
// An "OutputSocket" is (by default) used only to send packets.
// No packets are received on it (unless a subclass arranges this)
// OutputSocket 是（默认）仅用于发送数据包的类。
// 没有接收数据包方法就可以了（除非子类添加这一点）
class OutputSocket : public Socket {
public:
  //调用父类构造初始化，使用内核分配端口，设置两个成员为 0
  OutputSocket(UsageEnvironment& env);
  // 空的，还是调用基类的析构
  virtual ~OutputSocket();

  Boolean write(netAddressBits address, Port port, u_int8_t ttl,
     unsigned char* buffer, unsigned bufferSize);

protected:
  // 注意此处的权限
  OutputSocket(UsageEnvironment& env, Port port);

  portNumBits sourcePortNum() const { return fSourcePort.num(); }

private: // redefined virtual function
  virtual Boolean handleRead(unsigned char* buffer, unsigned bufferMaxSize,
     unsigned& bytesRead,
  struct sockaddr_in& fromAddress);

private:
  Port      fSourcePort; //源端口
  u_int8_t  fLastSentTTL;   //最后发送 TTL
};
```


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

OutputSocket 构造与析构

OutputSocket 的构造有两种形式，只有一个参数 env 的为 public 权限，其使用的端口是由内核选择的。带有参数 env 和 port 的是 protected 权限的，不对外暴露。

构造的时候初始化了两个成员 fSourcePort(源端口)和 fLastSentTTL(最后发送 TTL)为 0。

```
OutputSocket::OutputSocket(UsageEnvironment& env)
  : Socket(env, 0 /* let kernel choose port 让内核选择端口*/),
    fSourcePort(0), fLastSentTTL(0) {
}
```

```
OutputSocket::OutputSocket(UsageEnvironment& env, Port port)
  : Socket(env, port),
    fSourcePort(0), fLastSentTTL(0) {
}
```

析构函数是空的，调用基类的析构函数。

```
OutputSocket::~OutputSocket() {
}
```

handleRead 方法(处理读)

handleRead 方法是基类 Socket 中的一个纯虚接口，这里的实现只是为了使得 OutputSocket 类变为非抽象类。

```
// By default, we don't do reads:
Boolean OutputSocket
::handleRead(unsigned char* /*buffer*/, unsigned /*bufferMaxSize*/,
       unsigned& /*bytesRead*/, struct sockaddr_in& /*fromAddress*/) {
  return True;
}
```

write 方法(发送数据)

write 方法将 buffer 中的 bufferSize 字节数据发送到主机(address+port),发送时设置 TTL =ttl。如果 Port 为 0，那么使用内核分配的端口号，并设置 OutputSocket::fSourecPort 成员为内核分配的端口号。fSourecPort 成员从基类 Socket 中继承而来。

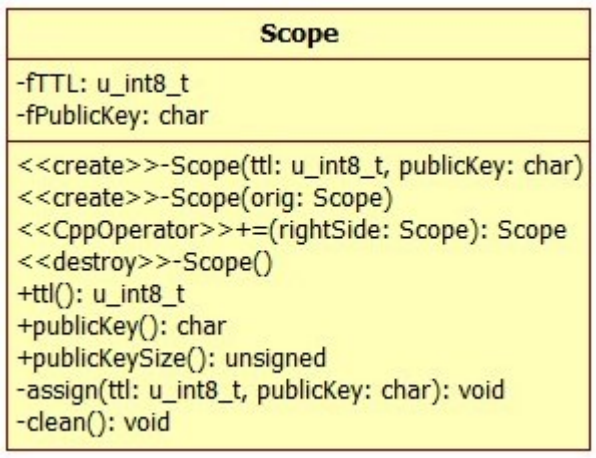
成功返回 true，失败返回 false。

```
// 将 buffer 中的 bufferSize 字节数据发送到主机(address+port),TTL =ttl
Boolean OutputSocket::write(netAddressBits address, Port port, u_int8_t ttl,
  unsigned char* buffer, unsigned bufferSize)
{
  if (ttl == fLastSentTTL) {
     // Optimization: So we don't do a 'set TTL' system call again
     // 优化：所以我们不需要再这样做了，“设置 TTL'是一个系统调用
     ttl = 0;
  }
  else {
     fLastSentTTL = ttl;
  }
  struct in_addr destAddr; destAddr.s_addr = address;

  if (!writeSocket(env(), socketNum(), destAddr, port, ttl,
     buffer, bufferSize))
     return False;   //失败时返回

  // 源端口号为 0
  if (sourcePortNum() == 0) {
```

```
    // Now that we've sent a packet, we can find out what the
    // kernel chose as our ephemeral source port number:
    // 现在我们已经发送一个包，我们可以获知
    // 内核为选择我们选择的临时源端口号：
    if (!getSourcePort(env(), socketNum(), fSourcePort)) {
        if (DebugLevel >= 1)
            env() << *this
            << ": failed to get source port: "
            << env().getResultMsg() << "\n";
        return False;
    }
  }

  return True;
}
```

8）Scope 范围类

Scope 类有两个数据成员 fTTL 和 fPublicKey。其中 fPublicKey 在默认情况先是字符串”nokey”。




```
class Scope {
public:
  Scope(u_int8_t ttl = 0, const char* publicKey = NULL);
  Scope(const Scope& orig);
  Scope& operator=(const Scope& rightSide);
  ~Scope();

  u_int8_t ttl() const
  {
     return fTTL;
  }

  const char* publicKey() const
  {
     return fPublicKey;
  }
  unsigned publicKeySize() const;

private:
  void assign(u_int8_t ttl, const char* publicKey);
  void clean();

  u_int8_t  fTTL;     //Time To Live
  char*     fPublicKey;  //公钥
};
```


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

assgin 与 clean 方法

assgin 方法用于设置成员 fTTL 和 fPublicKey 的值。

fPublicKey 是一个 char*指针，这里在参数为 NULL（声明的时候默认值为 NULL）的时候设置为”nokey”

```
void Scope::assign(u_int8_t ttl, const char* publicKey) {
  fTTL = ttl;

  fPublicKey = strDup(publicKey == NULL ? "nokey" : publicKey);
}
```

clean 释放由 assgin 分配给 fPublicKey 的内存空间。

```
void Scope::clean() {
  delete[] fPublicKey;
  fPublicKey = NULL;
}
```

Scope 构造与析构，拷贝与赋值重载

构造和拷贝构造，赋值运算符重载都是调用的 assgin 和 clean 来实现的。

```
Scope::Scope(u_int8_t ttl, const char* publicKey) {
  assign(ttl, publicKey);
}
```

```
Scope::Scope(const Scope& orig) {
  assign(orig.ttl(), orig.publicKey());
}
```

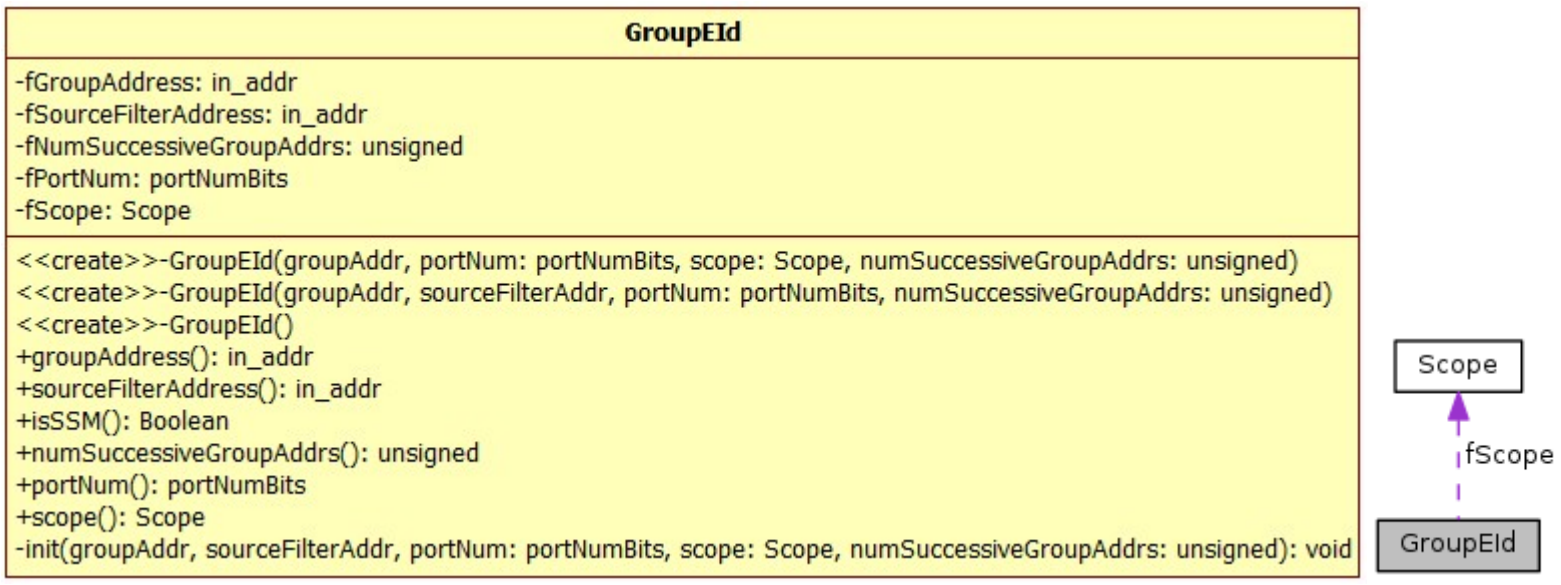
```
Scope& Scope::operator=(const Scope& rightSide) {
  if (&rightSide != this) {
    if (publicKey() == NULL
  || strcmp(publicKey(), rightSide.publicKey()) != 0) {
      clean();
      assign(rightSide.ttl(), rightSide.publicKey());
    } else { // need to assign TTL only
      fTTL = rightSide.ttl();
    }
  }

  return *this;
}
```

析构直接调用 clean 来实现。

```
Scope::~Scope() {
  clean();
}
```

9) GroupEId 组端 ID 类

GroupEId 是 Group Endpoint Id(组端点 ID)的意思。这个类用于保存组播端点的地址端口等信息。




```
class GroupEId {
public:
  GroupEId(struct in_addr const& groupAddr,
     portNumBits portNum, Scope const& scope,
     unsigned numSuccessiveGroupAddrs = 1);
  // used for a 'source-independent multicast' group
  // 用于源相互独立的多播组
  GroupEId(struct in_addr const& groupAddr,
  struct in_addr const& sourceFilterAddr,
     portNumBits portNum,
     unsigned numSuccessiveGroupAddrs = 1);
  // used for a 'source-specific multicast' group
  GroupEId(); // used only as a temp constructor prior to initialization 只是作为一个临时的构造函数在初始化之前

  struct in_addr const& groupAddress() const { return fGroupAddress; }
  struct in_addr const& sourceFilterAddress() const { return fSourceFilterAddress; }

  Boolean isSSM() const;

  unsigned numSuccessiveGroupAddrs() const {
     // could be >1 for hier encoding
     return fNumSuccessiveGroupAddrs;
  }

  portNumBits portNum() const { return fPortNum; }

  const Scope& scope() const { return fScope; }

private:
  void init(struct in_addr const& groupAddr,
  struct in_addr const& sourceFilterAddr,
     portNumBits portNum,
     Scope const& scope,
     unsigned numSuccessiveGroupAddrs);

private:
  struct in_addr fGroupAddress;   //组地址
  struct in_addr fSourceFilterAddress;  //源地址过滤
  unsigned fNumSuccessiveGroupAddrs; //连续的组地址数
  portNumBits fPortNum; //端口数
  Scope fScope;   //
};
```


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

init 方法(初始化)

init 方法是一个 private 方法，用于初始化内部数据成员，其由构造函数调用。

init 将其参数的值设置给对象的数据成员。

这里提一点 C++的知识，参数类型是引用的时候，如果不是 const 修饰形参，那么在调用的时候必须是传入一个存在的对象(左值)。如果是 const 修饰形参，那么可以是传入一个可以用于构造参数类型的匿名临时对象的常量(右值)等。

```
void GroupEId::init(struct in_addr const& groupAddr,
struct in_addr const& sourceFilterAddr,
  portNumBits portNum,
  Scope const& scope,/* TTL 如果不是 const 引用，那么传参的时候必须有一个 Scoped 对象*/
  unsigned numSuccessiveGroupAddrs) {
  fGroupAddress = groupAddr;
  fSourceFilterAddress = sourceFilterAddr;
  fNumSuccessiveGroupAddrs = numSuccessiveGroupAddrs;
  fPortNum = portNum;
  fScope = scope;
}
```

GroupEId 的构造

构造也是调用的 init 进行初始化赋值。

没有指定 sourceFilterAddr 的时候使用 (~0)表示没有源地址过滤器。

```
GroupEId::GroupEId(struct in_addr const& groupAddr,
  portNumBits portNum, Scope const& scope,
  unsigned numSuccessiveGroupAddrs)
{
  struct in_addr sourceFilterAddr;
  sourceFilterAddr.s_addr = ~0; // indicates no source filter 表示没有源过滤器
  init(groupAddr, sourceFilterAddr, portNum, scope, numSuccessiveGroupAddrs);
}
```

没有指定 scope 的时候，设置其 TTL 为 255，publicKey 为”nokey”。

```
GroupEId::GroupEId(struct in_addr const& groupAddr,
struct in_addr const& sourceFilterAddr,
  portNumBits portNum,
  unsigned numSuccessiveGroupAddrs)
{
  init(groupAddr, sourceFilterAddr, portNum, 255, numSuccessiveGroupAddrs);
}
```

这个构造函数不进行内部成员赋值操作，只是作为一个临时的构造函数在初始化之前。

```
GroupEId::GroupEId()
{
}
```

isSSM 方法(特定源多播)

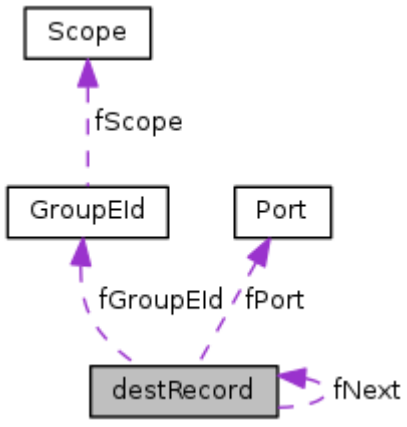
isSSM 方法用来查看其是否用于特定源多播。fSourceFilterAddress 在其为(~0)的时候代表其为任意源多播。(~0)的二进制形式为全是 1，那就不能用于过滤了。

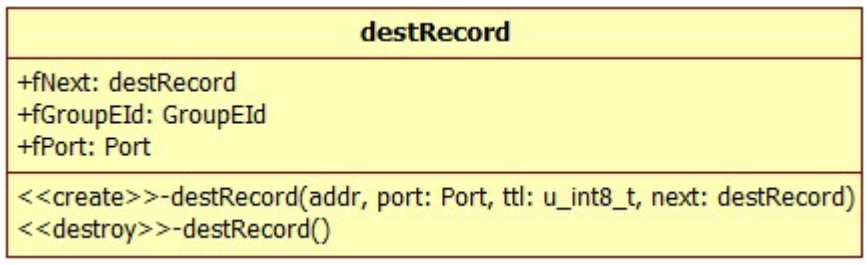
```
Boolean GroupEId::isSSM() const
{
  return fSourceFilterAddress.s_addr != netAddressBits(~0);
}
```

10） destRecord 目标记录类

destRecord 类是一个单向链表的结构，每个节点记录一个 GroupEId 对象和一个 Port 对象。

定义在 live555sourcecontrol\groupsock\include\Groupsock.hh 文件中。






```
class destRecord {
public:
  destRecord(struct in_addr const& addr, Port const& port, u_int8_t ttl,
     destRecord* next);
  virtual ~destRecord();

public:
  destRecord* fNext; //指向下一条记录
  GroupEId fGroupEId;//记录 GroupEId
  Port fPort;        //记录端口号
};
```

destRecord 的构造

构造只是简单的对内部成员的赋值操作。

```
destRecord
::destRecord(struct in_addr const& addr, Port const& port, u_int8_t ttl,
destRecord* next)
: fNext(next), fGroupEId(addr, port.num(), ttl), fPort(port) {
}
```

destRecord 的析构

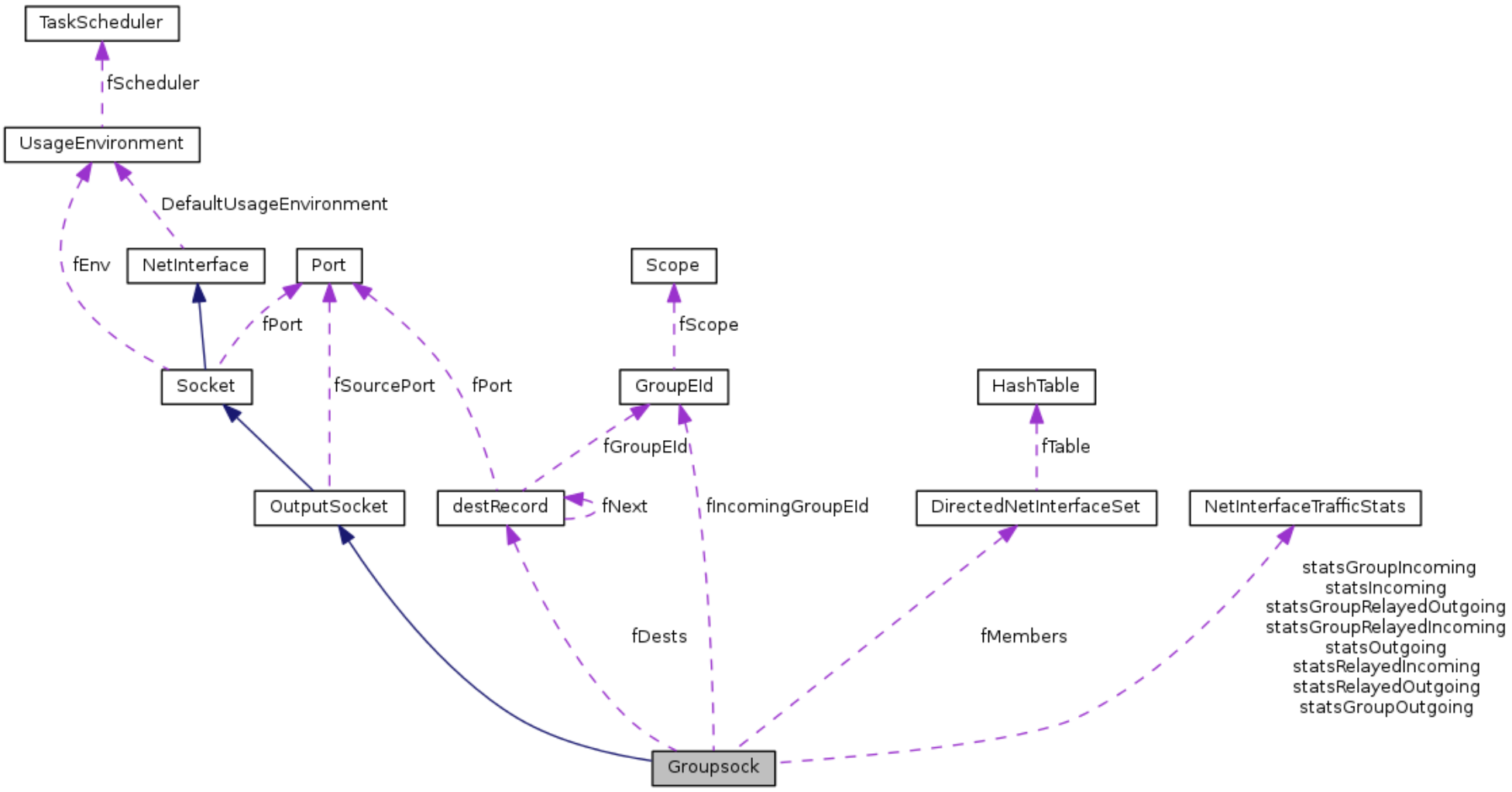
这里析构很简单，但是其体现的思想很重要。

delete fNext 表示释放其指向的节点，即当前节点的下一个节点。在释放其下一个节点的时候会调用其下一个节点的析构函数，这样层层递进，直到所有节点都被释放为止。

```
destRecord::~destRecord() {
  delete fNext;
}
```

11） Groupsock 组套接字类

这个类比较复杂。



Groupsock

+deleteIfNoMembers: Boolean
+isSlave: Boolean
+statsIncoming: NetInterfaceTrafficStats
+statsOutgoing: NetInterfaceTrafficStats
+statsRelayedIncoming: NetInterfaceTrafficStats
+statsRelayedOutgoing: NetInterfaceTrafficStats
+statsGroupIncoming: NetInterfaceTrafficStats
+statsGroupOutgoing: NetInterfaceTrafficStats
+statsGroupRelayedIncoming: NetInterfaceTrafficStats
+statsGroupRelayedOutgoing: NetInterfaceTrafficStats
-fIncomingGroupEId: GroupEId
-fDests: destRecord
-fTTL: u_int8_t
-fMembers: DirectedNetInterfaceSet

<<create>>-Groupsock(env: UsageEnvironment, groupAddr, port: Port, ttl: u_int8_t)
<<create>>-Groupsock(env: UsageEnvironment, groupAddr, sourceFilterAddr, port: Port)
<<destroy>>-Groupsock()
+changeDestinationParameters(newDestAddr, newDestPort: Port, newDestTTL: int): void
+addDestination(addr, port: Port): void
+removeDestination(addr, port: Port): void
+removeAllDestinations(): void
+groupAddress(): in_addr
+sourceFilterAddress(): in_addr
+isSSM(): Boolean
+ttl(): u_int8_t
+multicastSendOnly(): void
+output(env: UsageEnvironment, ttl: u_int8_t, buffer: unsigned char, bufferSize: unsigned, interfaceNotToFwdBackTo: DirectedNetInterface): Boolean
+members(): DirectedNetInterfaceSet
+wasLoopedBackFromUs(env: UsageEnvironment, fromAddress): Boolean
+handleRead(buffer: unsigned char, bufferMaxSize: unsigned, bytesRead: unsigned, fromAddress): Boolean
-outputToAllMembersExcept(exceptInterface: DirectedNetInterface, ttlToFwd: u_int8_t, data: unsigned char, size: unsigned, sourceAddr: netAddressBits): int

```
// A "Groupsock" is used to both send and receive packets.
// As the name suggests, it was originally designed to send/receive
// multicast, but it can send/receive unicast as well.
// Groupsock 是用来发送和接收数据包。
// 正如其名称所暗示的，它最初被设计为多播数据发送/接收，但它可以很好的单播发送/接收。
class Groupsock : public OutputSocket {
public:
```

```
  Groupsock(UsageEnvironment& env, struct in_addr const& groupAddr,
     Port port, u_int8_t ttl);
  // used for a 'source-independent multicast' group
  // 用于“源无关组播'组(任意源组播)

  Groupsock(UsageEnvironment& env, struct in_addr const& groupAddr,
  struct in_addr const& sourceFilterAddr/*源地址过滤*/,
     Port port);
  // used for a 'source-specific multicast' group
  // 用于特定源组播

  virtual ~Groupsock();

  // 改变目标参数(fDest 第一个节点)
  void changeDestinationParameters(struct in_addr const& newDestAddr,
     Port newDestPort, int newDestTTL);
  // By default, the destination address, port and ttl for
  // outgoing packets are those that were specified in
  // the constructor.  This works OK for multicast sockets,
  // but for unicast we usually want the destination port
  // number, at least, to be different from the source port.
  // 默认情况下，目的地址，端口和 TTL 传出数据包是在构造中指定的那些。
  // 这适用于组播套接字确定,但对于单播,我们通常希望的目的端口号,至少是来自源端口不同的。
  // (If a parameter is 0 (or ~0 for ttl), then no change made.)
  //（如果参数为 0（或~0 为 TTL），则不改变 made）

  // As a special case, we also allow multiple destinations (addresses & ports)
  // (This can be used to implement multi-unicast.)
  //作为一个特殊的情况下，我们也允许多个目标（地址和端口）
  //（这可以用于实现多播）。

  // 添加目标
  void addDestination(struct in_addr const& addr, Port const& port);
  // 移除目标
  void removeDestination(struct in_addr const& addr, Port const& port);
  // 移除所有目标
  void removeAllDestinations();

  struct in_addr const& groupAddress() const
  {
     return fIncomingGroupEId.groupAddress();
  }
  struct in_addr const& sourceFilterAddress() const
  {
     return fIncomingGroupEId.sourceFilterAddress();
  }

  Boolean isSSM() const
  {
     return fIncomingGroupEId.isSSM();
  }

  u_int8_t ttl() const { return fTTL; }

  // 仅组播发送
  void multicastSendOnly(); // send, but don't receive any multicast packets 发送，但不接收任何多播报文

  Boolean output(UsageEnvironment& env, u_int8_t ttl,
     unsigned char* buffer, unsigned bufferSize,
```

```
    DirectedNetInterface* interfaceNotToFwdBackTo = NULL);

  DirectedNetInterfaceSet& members() { return fMembers; }

  Boolean deleteIfNoMembers;
  Boolean isSlave; // for tunneling

  static NetInterfaceTrafficStats statsIncoming;  //统计传入数据
  static NetInterfaceTrafficStats statsOutgoing;  //统计传出数据
  static NetInterfaceTrafficStats statsRelayedIncoming; //统计中继传入数据
  static NetInterfaceTrafficStats statsRelayedOutgoing; //统计中继传出数据
  NetInterfaceTrafficStats statsGroupIncoming; // *not* static
  NetInterfaceTrafficStats statsGroupOutgoing; // *not* static
  NetInterfaceTrafficStats statsGroupRelayedIncoming; // *not* static
  NetInterfaceTrafficStats statsGroupRelayedOutgoing; // *not* static

  // 检测是否是在本地环回
  Boolean wasLoopedBackFromUs(UsageEnvironment& env,
  struct sockaddr_in& fromAddress);

public: // redefined virtual functions

  virtual Boolean handleRead(unsigned char* buffer, unsigned bufferMaxSize,
      unsigned& bytesRead,
  struct sockaddr_in& fromAddress);

private:
  // 发送到所有成员，排除 exceptInterface
  int outputToAllMembersExcept(DirectedNetInterface* exceptInterface,
      u_int8_t ttlToFwd,
      unsigned char* data, unsigned size,
      netAddressBits sourceAddr);

private:
  GroupEId fIncomingGroupEId;     //传入 GroupEId
  destRecord* fDests;    //目标记录链表头指针
  u_int8_t fTTL;     //Time To Live
  DirectedNetInterfaceSet fMembers;  //有向网络接口接口集
};
```

Groupsock 构造(任意源多播组)

参数 groupAddr 和 port 代表了要加入的组播地址

```
// Constructor for a source-independent multicast group
Groupsock::Groupsock(UsageEnvironment& env, struct in_addr const& groupAddr,
  Port port, u_int8_t ttl)
  : OutputSocket(env, port),
  deleteIfNoMembers(False), isSlave(False),
  fIncomingGroupEId(groupAddr, port.num(), ttl), fDests(NULL), fTTL(ttl)
{
  addDestination(groupAddr, port);    //添加目标到链表
  // 加入任意源多播组
  if (!socketJoinGroup(env, socketNum(), groupAddr.s_addr)) {
      if (DebugLevel >= 1) {
          env << *this << ": failed to join group: "
              << env.getResultMsg() << "\n";
      }
```


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

```
  }

  // Make sure we can get our source address:
  // 确保我们可以得到我们的源地址：
  if (ourIPAddress(env) == 0) {
      if (DebugLevel >= 0) { // this is a fatal error
          env << "Unable to determine our source address: "
              << env.getResultMsg() << "\n";
      }
  }

  if (DebugLevel >= 2) env << *this << ": created\n";
}
```

Groupsock 构造(特定源多播组)

这里比前面多了一个参数 sourceFilterAddr 源地址过滤。如果加入特定源多播组失败，再尝试加入任意源多播组。

```
// Constructor for a source-specific multicast group
// 构造函数用于一个特定源的组播组
Groupsock::Groupsock(UsageEnvironment& env, struct in_addr const& groupAddr,
struct in_addr const& sourceFilterAddr,
  Port port)
  : OutputSocket(env, port),
  deleteIfNoMembers(False), isSlave(False),
  fIncomingGroupEId(groupAddr, sourceFilterAddr, port.num()),
  fDests(NULL), fTTL(255) {
  addDestination(groupAddr, port);

  // First try a SSM join.  If that fails, try a regular join:
  // 先尝试加入一个 SSM，如果失败了，尝试常规连接
  if (!socketJoinGroupSSM(env, socketNum(), groupAddr.s_addr,
      sourceFilterAddr.s_addr)) {
      if (DebugLevel >= 3) {
          env << *this << ": SSM join failed: "
              << env.getResultMsg();
          env << " - trying regular join instead\n";
      }
      // 尝试加入一个任意源多播组
      if (!socketJoinGroup(env, socketNum(), groupAddr.s_addr)) {
          if (DebugLevel >= 1) {
              env << *this << ": failed to join group: "
                  << env.getResultMsg() << "\n";
          }
      }
  }

  if (DebugLevel >= 2) env << *this << ": created\n";
}
```

Groupsock 析构

```
Groupsock::~Groupsock() {
  if (isSSM()) {
      // 离开特定源组播
      if (!socketLeaveGroupSSM(env(), socketNum(), groupAddress().s_addr,
          sourceFilterAddress().s_addr)) {
          socketLeaveGroup(env(), socketNum(), groupAddress().s_addr);
```

```
    }
  }
  else {
    // 离开任意源组播
    socketLeaveGroup(env(), socketNum(), groupAddress().s_addr);
  }

  delete fDests;  //释放目标链表

  if (DebugLevel >= 2) env() << *this << ": deleting\n";
}
```

changeDestinationParameters 改变目标参数(addr+port)

只改变了 fDests 指向的节点，其后的节点未改变。改变地址的时候先使得旧的地址离开多播组，然后使得新的地址加入多播组。

改变端口需要先关闭原本的数据报 socket，再用新的端口值创建一个 socket，然后重新加入多播组。

```
void
Groupsock::changeDestinationParameters(struct in_addr const& newDestAddr,
Port newDestPort, int newDestTTL) {
  if (fDests == NULL) return;

  struct in_addr destAddr = fDests->fGroupEId.groupAddress();
  if (newDestAddr.s_addr != 0) {
    if (newDestAddr.s_addr != destAddr.s_addr
        && IsMulticastAddress(newDestAddr.s_addr)) {
      // If the new destination is a multicast address, then we assume that
      // we want to join it also.  (If this is not in fact the case, then
      // call "multicastSendOnly()" afterwards.)
      // 如果新的目标是多播地址，那么我们假定我们希望加入它。
      // （否则在实际的情况下，是调用“multicastSendOnly()”之后。）

      // 离开旧的多播组
      socketLeaveGroup(env(), socketNum(), destAddr.s_addr);
      // 加入新的多播组
      socketJoinGroup(env(), socketNum(), newDestAddr.s_addr);
    }
    destAddr.s_addr = newDestAddr.s_addr;
  }

  portNumBits destPortNum = fDests->fGroupEId.portNum();
  if (newDestPort.num() != 0) {
    if (newDestPort.num() != destPortNum
        && IsMulticastAddress(destAddr.s_addr)) {
      // Also bind to the new port number:
      // 也是绑定到新的端口号
      changePort(newDestPort);
      // And rejoin the multicast group:
      // 并重新加入组播组：
      socketJoinGroup(env(), socketNum(), destAddr.s_addr);
    }
    destPortNum = newDestPort.num();
    fDests->fPort = newDestPort;
  }

  u_int8_t destTTL = ttl();
  if (newDestTTL != ~0) destTTL = (u_int8_t)newDestTTL;
```


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

```
  fDests->fGroupEId = GroupEId(destAddr, destPortNum, destTTL);
}
```

addDestination 添加目标到链表

```
void Groupsock::addDestination(struct in_addr const& addr, Port const& port) {
  // Check whether this destination is already known:
  // 检查此目标是否为已知的:
  for (destRecord* dests = fDests; dests != NULL; dests = dests->fNext) {
      if (addr.s_addr == dests->fGroupEId.groupAddress().s_addr
          && port.num() == dests->fPort.num()) {
          return;
      }
  }
  // 头插入到链表
  fDests = new destRecord(addr, port, ttl(), fDests);
}
```

removeDestination 从链表移除目标

```
void Groupsock::removeDestination(struct in_addr const& addr, Port const& port) {
  for (destRecord** destsPtr = &fDests; *destsPtr != NULL;
      destsPtr = &((*destsPtr)->fNext)) {
      if (addr.s_addr == (*destsPtr)->fGroupEId.groupAddress().s_addr
          && port.num() == (*destsPtr)->fPort.num()) {
          // Remove the record pointed to by *destsPtr :
          // 删除该* destsPtr 指向的记录
          destRecord* next = (*destsPtr)->fNext;
          (*destsPtr)->fNext = NULL;
          delete (*destsPtr);
          *destsPtr = next;
          return;
      }
  }
}
```

removeAllDestinations 移除所有目标

```
void Groupsock::removeAllDestinations() {
  delete fDests; fDests = NULL;
}
```

output 发送数据

将 buffer 中的内容发送到 fDests 链表中的所有节点，并统计流量。

后面转发至成员的应该是不会用到。outputToAllMembersExcept 中调用了 DirectedNetInterface::SourceAddrOKForRelaying，而这个方法没有对应的实现。

```
Boolean Groupsock::output(UsageEnvironment& env, u_int8_t ttlToSend,
  unsigned char* buffer, unsigned bufferSize,
  DirectedNetInterface* interfaceNotToFwdBackTo) {
  do {
      // First, do the datagram send, to each destination:
      // 首先，发送数据报到每一个目标
```


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

```
    Boolean writeSuccess = True;
    for (destRecord* dests = fDests; dests != NULL; dests = dests->fNext)
    {
        if (!write(dests->fGroupEId.groupAddress().s_addr, dests->fPort, ttlToSend,
            buffer, bufferSize)) {
            writeSuccess = False;
            break; //有一个发送失败，就跳出
        }
    }
    if (!writeSuccess) break; //发送失败，直接跳出
    // 统计流量
    statsOutgoing.countPacket(bufferSize);
    statsGroupOutgoing.countPacket(bufferSize);

    // Then, forward to our members:然后，转发至我们的成员：
    int numMembers = 0;
    if (!members().IsEmpty()) {
        numMembers =
            outputToAllMembersExcept(interfaceNotToFwdBackTo,
            ttlToSend, buffer, bufferSize,
            ourIPAddress(env)/*自身的地址*/);
        if (numMembers < 0) break;
    }

    if (DebugLevel >= 3) {
        env << *this << ": wrote " << bufferSize << " bytes, ttl "
            << (unsigned)ttlToSend;
        if (numMembers > 0) {
            env << "; relayed to " << numMembers << " members";
        }
        env << "\n";
    }
    return True;
  } while (0);

  if (DebugLevel >= 0) { // this is a fatal error
    env.setResultMsg("Groupsock write failed: ", env.getResultMsg());
  }
  return False;
}
```

handleRead 处理读

```
Boolean Groupsock::handleRead(unsigned char* buffer, unsigned bufferMaxSize,
  unsigned& bytesRead,
struct sockaddr_in& fromAddress) {
  // Read data from the socket, and relay it across any attached tunnels
  // 读取来自套接字的数据，并转发给它连接的任何一个隧道
  //##### later make this code more general - independent of tunnels
  //##### 后来又使这段代码更通用 - 无关的隧道

  bytesRead = 0;

  int maxBytesToRead = bufferMaxSize - TunnelEncapsulationTrailerMaxSize;
  // 从 socket 读取数据
  int numBytes = readSocket(env(), socketNum(),
```

```
    buffer, maxBytesToRead, fromAddress);
  if (numBytes < 0) {
    if (DebugLevel >= 0) { // this is a fatal error
      env().setResultMsg("Groupsock read failed: ",
        env().getResultMsg());
    }
    return False;
  }

  // If we're a SSM group, make sure the source address matches:
  // 如果我们是一个 SSM 组，确保源地址匹配：
  if (isSSM()
    && fromAddress.sin_addr.s_addr != sourceFilterAddress().s_addr) {
    return True;
  }

  // We'll handle this data.
  // Also write it (with the encapsulation trailer) to each member,
  // unless the packet was originally sent by us to begin with.
  // 我们会处理这些数据。
  // 也是把它发送到（encapsulation trailer）每个成员
  // 除非该数据包最初是由我们发出开始。
  bytesRead = numBytes;

  int numMembers = 0;
  if (!wasLoopedBackFromUs(env(), fromAddress)) {
    //不是本地环回，统计流量
    statsIncoming.countPacket(numBytes);
    statsGroupIncoming.countPacket(numBytes);
    numMembers = //发送到每一个成员
      outputToAllMembersExcept(NULL, ttl(),
      buffer, bytesRead,
      fromAddress.sin_addr.s_addr);
    if (numMembers > 0) {
      // 统计中继流量
      statsRelayedIncoming.countPacket(numBytes);
      statsGroupRelayedIncoming.countPacket(numBytes);
    }
  }
  if (DebugLevel >= 3) {
    env() << *this << ": read " << bytesRead << " bytes from " << AddressString(fromAddress).val();
    if (numMembers > 0) {
      env() << "; relayed to " << numMembers << " members";
    }
    env() << "\n";
  }

  return True;
}
```

wasLoopedBackFromUs 判断是否是本地回环

```
Boolean Groupsock::wasLoopedBackFromUs(UsageEnvironment& env,
struct sockaddr_in& fromAddress)
{
  if (fromAddress.sin_addr.s_addr
      == ourIPAddress(env)) {
```

```
      if (fromAddress.sin_port == sourcePortNum()) {
#ifdef DEBUG_LOOPBACK_CHECKING
        if (DebugLevel >= 3) {
          env() << *this << ": got looped-back packet\n";
        }
#endif
        return True;
      }
    }

    return False;
}
```

outputToAllMembersExcept 发送数据给所有成员(例外成员除外)

```
int Groupsock::outputToAllMembersExcept(DirectedNetInterface* exceptInterface,
    u_int8_t ttlToFwd,
    unsigned char* data, unsigned size,
    netAddressBits sourceAddr) {
    // Don't forward TTL-0 packets 无需转发 TTL 为 0 包
    if (ttlToFwd == 0) return 0;

    DirectedNetInterfaceSet::Iterator iter(members());
    unsigned numMembers = 0;
    DirectedNetInterface* interf;
    while ((interf = iter.next()) != NULL) {
        // Check whether we've asked to exclude this interface:
        // 检查我们是否已经要求排除这个接口:
        if (interf == exceptInterface)
            continue;

        // Check that the packet's source address makes it OK to
        // be relayed across this interface:
        // 检查数据包的源地址，使得它确定跨过该接口被转发:
        UsageEnvironment& saveEnv = env();
        // because the following call may delete "this"
        // 因为下面的调用可能会 delete this
        if (!interf->SourceAddrOKForRelaying(saveEnv, sourceAddr)) {
            if (strcmp(saveEnv.getResultMsg(), "") != 0) {
                // Treat this as a fatal error
                return -1;
            }
            else {
                continue;
            }
        }

        if (numMembers == 0) {
            // We know that we're going to forward to at least one
            // member, so fill in the tunnel encapsulation trailer.
            // 我们知道，我们将转发给至少一个成员，所以请填满该 tunnel encapsulation trailer。
            // (Note: Allow for it not being 4-byte-aligned.)
            // (注：允许它不是 4 字节对齐的。)
            TunnelEncapsulationTrailer* trailerInPacket
                = (TunnelEncapsulationTrailer*)&data[size];
            TunnelEncapsulationTrailer* trailer;
```


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

```
        Boolean misaligned = ((uintptr_t)trailerInPacket & 3) != 0;
        unsigned trailerOffset;
        u_int8_t tunnelCmd;
        if (isSSM()) {
            // add an 'auxilliary address' before the trailer
            // 在 trailer 前加一个“若干辅助地址”
            trailerOffset = TunnelEncapsulationTrailerAuxSize;
            tunnelCmd = TunnelDataAuxCmd;
        }
        else {
            trailerOffset = 0;
            tunnelCmd = TunnelDataCmd;
        }
        unsigned trailerSize = TunnelEncapsulationTrailerSize + trailerOffset;
        unsigned tmpTr[TunnelEncapsulationTrailerMaxSize];
        if (misaligned) {
            trailer = (TunnelEncapsulationTrailer*)&tmpTr;
        }
        else {
            trailer = trailerInPacket;
        }
        trailer += trailerOffset;

        if (fDests != NULL) {
            trailer->address() = fDests->fGroupEId.groupAddress().s_addr;
            trailer->port() = fDests->fPort; // structure copy, outputs in network order 结构复制，网络字
节序输出
        }
        trailer->ttl() = ttlToFwd;
        trailer->command() = tunnelCmd;

        if (isSSM()) {
            trailer->auxAddress() = sourceFilterAddress().s_addr;
        }

        if (misaligned) {
            memmove(trailerInPacket, trailer - trailerOffset, trailerSize);
        }

        size += trailerSize;
    }

    interf->write(data, size);
    ++numMembers;
  }

  return numMembers;
}
```

全局 operator<<(UsageEnvironment& s, const Groupsock& g)

```
UsageEnvironment& operator<<(UsageEnvironment& s, const Groupsock& g) {
  UsageEnvironment& s1 = s << timestampString() << " Groupsock("
      << g.socketNum() << ": "
      << AddressString(g.groupAddress()).val()
      << ", " << g.port() << ", ";
```


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

```
  if (g.isSSM()) {
     return s1 << "SSM source: "
        << AddressString(g.sourceFilterAddress()).val() << ")";
  }
  else {
     return s1 << (unsigned)(g.ttl()) << ")";
  }
}
```

12） env.groupsockPriv->socketTable 操作相关函数

这几个函数是 getSocketTable/ unsetGroupsockBySocket/ setGroupsockBySocket/ getGroupsockBySocket，它们都定义在文件 live555sourcecontrol\groupsock\Groupsock.cpp 中。

这几个函数是给后面将要说的类 GroupsockLookupTable 准备的。

getSocketTable 获取 socket 哈希表

getSocketTable 函数用于获取 env.groupsockPriv->socketTable 的引用。如果其为 NULL，那么为其创建一个哈希表。

```
// A hash table used to index Groupsocks by socket number.
// 为 env.groupsockPriv->socketTable 引用通过套接字编号索引 Groupsocks 的哈希表。
static HashTable*& getSocketTable(UsageEnvironment& env)
{
  _groupsockPriv* priv = groupsockPriv(env);
  if (priv->socketTable == NULL) { // We need to create it
     // 创建 hashTable
     priv->socketTable = HashTable::create(ONE_WORD_HASH_KEYS);
  }
  return priv->socketTable;
}
```

unsetGroupsockBySocket 移除条目

unsetGroupsockBySocket 从 groupsock->env().groupsockPriv->socketTable 哈希表中查找 groupsock

// 如果找到了就移除，如果哈希表中已经没有了元素，就释放哈希表。

```
// 从 groupsock->env().groupsockPriv->socketTable 哈希表中查找 groupsock
// 如果找到了就移除，如果哈希表中已经没有了元素，就释放哈希表
static Boolean unsetGroupsockBySocket(Groupsock const* groupsock)
{
  do {
     if (groupsock == NULL) break;

     int sock = groupsock->socketNum();
     // Make sure "sock" is in bounds:
     // 确保 sock 是在范围内：
     if (sock < 0) break;
     //引用哈希表
     HashTable*& sockets = getSocketTable(groupsock->env());
     //从哈希表中查找 sock 对应的 Groupsock 对象
     Groupsock* gs = (Groupsock*)sockets->Lookup((char*)(long)sock);
     if (gs == NULL || gs != groupsock) break;
     sockets->Remove((char*)(long)sock);//找到了就移除

     if (sockets->IsEmpty()) {
        // We can also delete the table (to reclaim space):
        // 没有了元素，就释放哈希表
```

```
        delete sockets; sockets = NULL;
        reclaimGroupsockPriv(gs->env());
    }

    return True;
  } while (0);

  return False;
}
```

setGroupsockBySocket 添加条目

```
// 向 groupsock->env().groupsockPriv->socketTable 添加一个条目
static Boolean setGroupsockBySocket(UsageEnvironment& env, int sock,
  Groupsock* groupsock)
{
  do {
    // Make sure the "sock" parameter is in bounds:
    // 确保 sock 参数是在范围内
    if (sock < 0) {
      char buf[100];
      sprintf(buf, "trying to use bad socket (%d)", sock);
      env.setResultMsg(buf);
      break;
    }
    //引用 groupsock->env().groupsockPriv->socketTable
    HashTable* sockets = getSocketTable(env);

    // Make sure we're not replacing an existing Groupsock (although that shouldn't happen)
    // 确保我们不会替换现有的 Groupsock（尽管这不应该发生）
    Boolean alreadyExists
      = (sockets->Lookup((char*)(long)sock) != 0);
    if (alreadyExists) {   //如果哈希表中已经存在 sock 对应的条目
      char buf[100];
      sprintf(buf,
        "Attempting to replace an existing socket (%d",
        sock);
      env.setResultMsg(buf);
      break;
    }
    // 添加一个条目，sock 为 key,groupsock 为 value
    sockets->Add((char*)(long)sock, groupsock);
    return True;
  } while (0);

  return False;
}
```

getGroupsockBySocket 查找条目

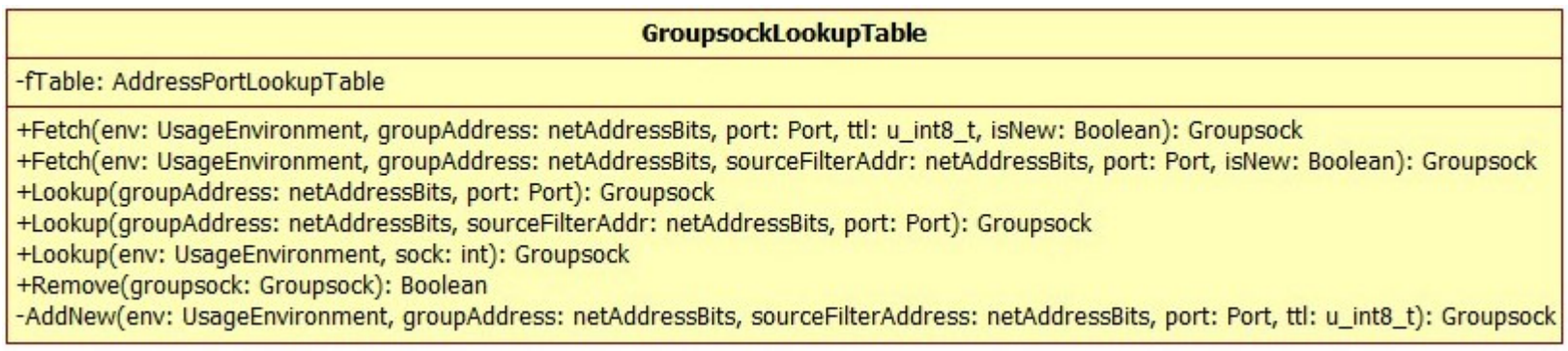
```
// 从 groupsock->env().groupsockPriv->socketTable 中查找 sock 对应的 groupsock
static Groupsock* getGroupsockBySocket(UsageEnvironment& env, int sock)
{
  do {
    // Make sure the "sock" parameter is in bounds:
    if (sock < 0) break;

    HashTable* sockets = getSocketTable(env);
    return (Groupsock*)sockets->Lookup((char*)(long)sock);
```

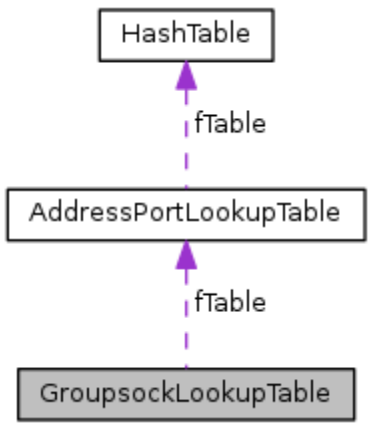
```
  } while (0);

  return NULL;
}
```

13) GroupsockLookupTable 组套接字查找表类

这个函数使用默认的构造与析构函数。

成员 AddressPortLookupTable fTable; //哈希表是一个对象，而非引用。这里无需绑定特定的哈希表，这是与前面几个不一样的。AddressPortLookupTable 类在之前说过，这个类使用三个数据(address1,address2,port)组合来作为一个 key，而 value 没有指定为特定类型(void*)。






```
// A data structure for looking up a 'groupsock'
// 数据结构，寻找一个'groupsock“
// by (multicast address, port), or by socket number
// 通过（多播地址，端口），或通过套接字数值
class GroupsockLookupTable {
public:
  // 通过 groupAddress 和 port 查找一个 Groupsock 对象，没找到就新建一个
  Groupsock* Fetch(UsageEnvironment& env, netAddressBits groupAddress,
      Port port, u_int8_t ttl, Boolean& isNew);
  // Creates a new Groupsock if none already exists

  Groupsock* Fetch(UsageEnvironment& env, netAddressBits groupAddress,
      netAddressBits sourceFilterAddr,
      Port port, Boolean& isNew);
  // Creates a new Groupsock if none already exists

  Groupsock* Lookup(netAddressBits groupAddress, Port port);
  // Returns NULL if none already exists

  Groupsock* Lookup(netAddressBits groupAddress,
      netAddressBits sourceFilterAddr,
      Port port);
  // Returns NULL if none already exists

  Groupsock* Lookup(UsageEnvironment& env, int sock);
  // Returns NULL if none already exists
```

```
  Boolean Remove(Groupsock const* groupsock);

  // Used to iterate through the groupsocks in the table
  class Iterator {
  public:
     Iterator(GroupsockLookupTable& groupsocks);

     Groupsock* next(); // NULL iff none

  private:
     AddressPortLookupTable::Iterator fIter;
  };

private:
  // 向 fTable 中添加条目，也向(env.groupsockPriv->socketTable)
  Groupsock* AddNew(UsageEnvironment& env/*使用环境*/,
     netAddressBits groupAddress/*组地址*/,
     netAddressBits sourceFilterAddress/*源地址过滤器*/,
     Port port, u_int8_t ttl);

private:
  friend class Iterator;
  AddressPortLookupTable fTable;  //哈希表
};
```

AddNew 添加新条目

AddNew 使用参数 groupAddress、port、ttl 和 socrceFilterAddress 构建一个 Groupsock 对象。然后以这个对象的地址为 Value， socketNum()为 key，添加到 env.groupsockPriv->socketTable。同时也以这个对象的地址为 Value，参数 groupAddress, sourceFilterAddress, port 组合为 key，添加到 fTable。

```
Groupsock* GroupsockLookupTable::AddNew(UsageEnvironment& env,
  netAddressBits groupAddress,
  netAddressBits sourceFilterAddress,
  Port port, u_int8_t ttl)
{
  Groupsock* groupsock;
  do {
     // 构建 Groupsock 对象 --value
     struct in_addr groupAddr; groupAddr.s_addr = groupAddress;
     if (sourceFilterAddress == netAddressBits(~0)) {
        // regular, ISM groupsock 任意源
        groupsock = new Groupsock(env, groupAddr, port, ttl);
     }
     else {
        // SSM groupsock特定源
        struct in_addr sourceFilterAddr;
        sourceFilterAddr.s_addr = sourceFilterAddress;
        groupsock = new Groupsock(env, groupAddr, sourceFilterAddr, port);
     }

     if (groupsock == NULL || groupsock->socketNum() < 0) break;
     // 添加到 env.groupsockPriv->socketTable
     if (!setGroupsockBySocket(env, groupsock->socketNum(), groupsock)) break;
     //添加到 fTable
     fTable.Add(groupAddress, sourceFilterAddress, port, (void*)groupsock);
  } while (0);
```


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

```
  return groupsock;
}
```

Fetch 任意源 Groupsock 查找

找到了就返回，没有找到就创建一个新的 Groupsock，添加到两个哈希表中后返回。

```
Groupsock*
GroupsockLookupTable::Fetch(UsageEnvironment& env,
netAddressBits groupAddress, Port port, u_int8_t ttl, Boolean& isNew)
{
  isNew = False;
  Groupsock* groupsock;
  do {
    groupsock = (Groupsock*)fTable.Lookup(groupAddress, (~0), port);
    if (groupsock == NULL) { // we need to create one:
      groupsock = AddNew(env, groupAddress, (~0), port, ttl);
      if (groupsock == NULL) break;
      isNew = True;
    }
  } while (0);

  return groupsock;
}
```

Fetch 特定源 Groupsock 查找

```
Groupsock*
GroupsockLookupTable::Fetch(UsageEnvironment& env,
netAddressBits groupAddress,
netAddressBits sourceFilterAddr, Port port,
Boolean& isNew)
{
  isNew = False;
  Groupsock* groupsock;
  do {
    groupsock
      = (Groupsock*)fTable.Lookup(groupAddress, sourceFilterAddr, port);
    if (groupsock == NULL) { // we need to create one:
      groupsock = AddNew(env, groupAddress, sourceFilterAddr, port, 0);
      if (groupsock == NULL) break;
      isNew = True;
    }
  } while (0);

  return groupsock;
}
```

Lookup 查找 Groupsock(fTable 中)

查找任意源组播的 Groupsock 对象。

```
Groupsock*
GroupsockLookupTable::Lookup(netAddressBits groupAddress, Port port)
{
  return (Groupsock*)fTable.Lookup(groupAddress, (~0), port);
}
```

查找特定源组播的 Groupsock 对象。

```
Groupsock*
GroupsockLookupTable::Lookup(netAddressBits groupAddress,
netAddressBits sourceFilterAddr, Port port)
{
  return (Groupsock*)fTable.Lookup(groupAddress, sourceFilterAddr, port);
}
```

Lookup 查找 Groupsock(env.groupsockPriv->socketTable 中)

```
Groupsock* GroupsockLookupTable::Lookup(UsageEnvironment& env, int sock)
{
  return getGroupsockBySocket(env, sock);
}
```

Remove 移除条目

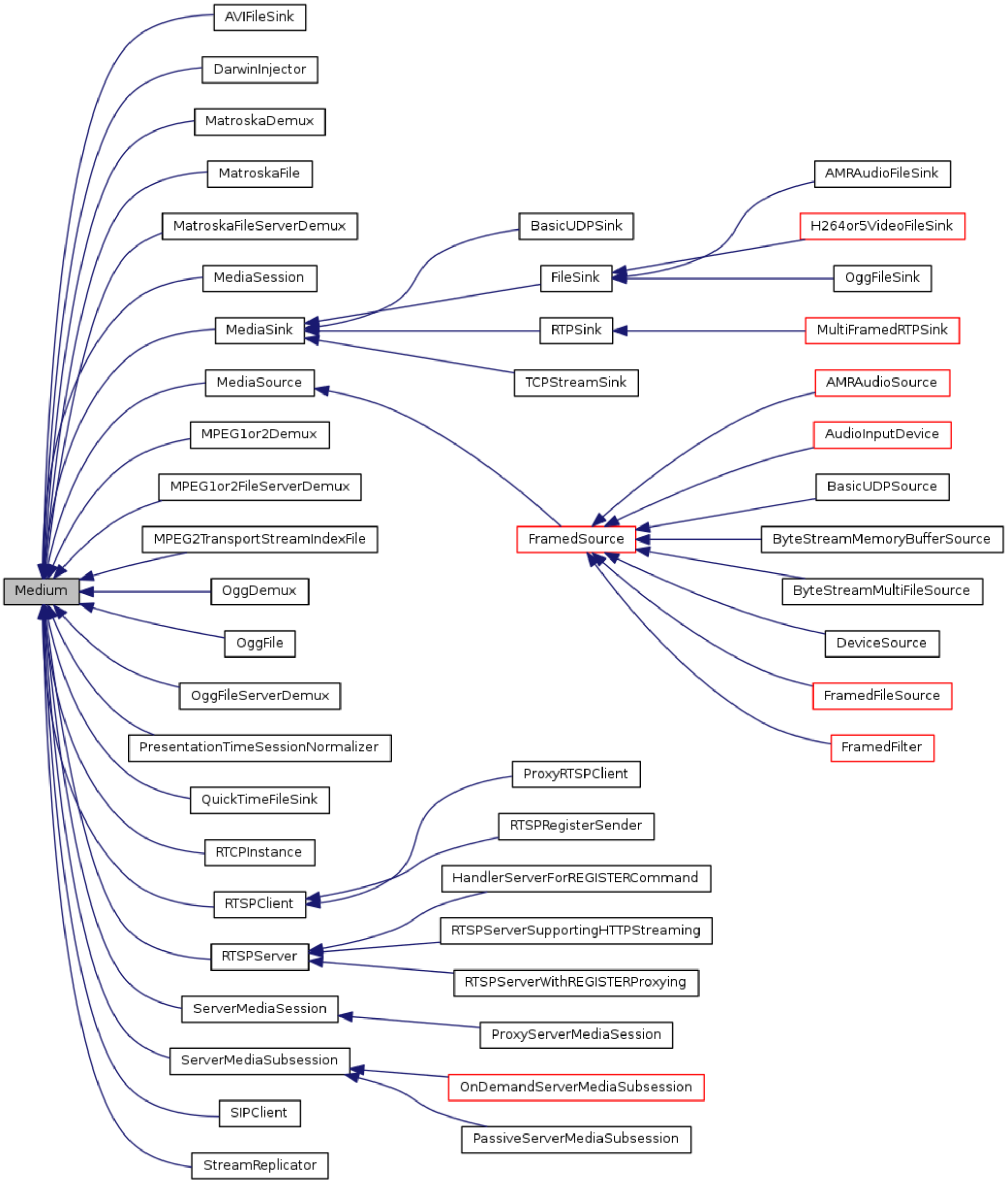
从 fTable 和 env.groupsockPriv->socketTable 中移除 groupsock 对应的条目。

```
Boolean GroupsockLookupTable::Remove(Groupsock const* groupsock)
{
  unsetGroupsockBySocket(groupsock);
  return fTable.Remove(groupsock->groupAddress().s_addr,
      groupsock->sourceFilterAddress().s_addr,
      groupsock->port());
}
```

五、 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。



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