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。

## 时间相关类

### 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。

### 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);

}

### 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) {}

};



### 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);

}

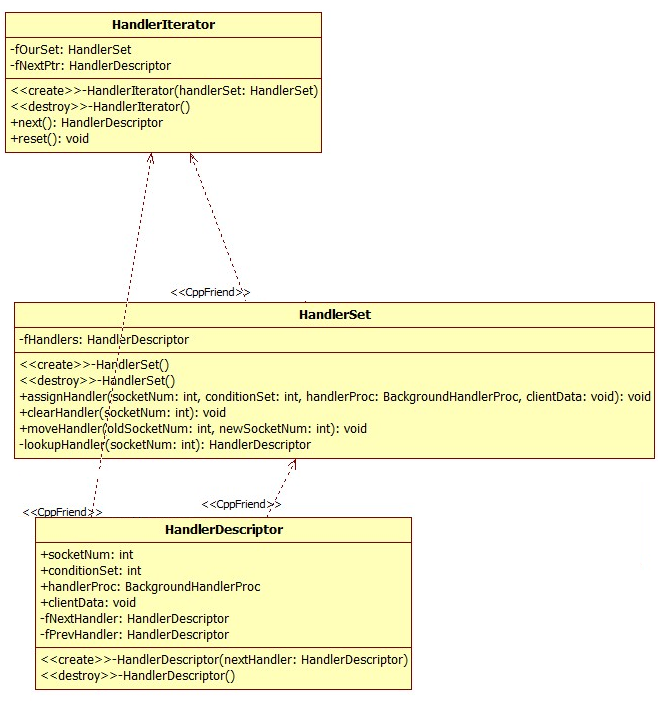
## 处理程序相关类

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。



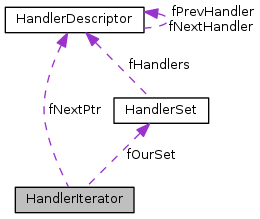
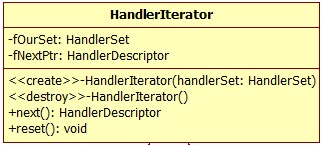
### HandlerIterator处理程序迭代器类

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;

}

### HandlerDescriptor 处理程序描述类

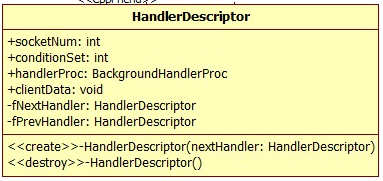
HandlerDescriptor类是一个很重要的类，其保存了处理程序的函数指针和相关的数据的地址。在构建处理任务的时候，会使用到这个类的对象，处理任务的时候也会用到。

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/unix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;

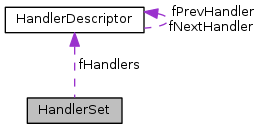
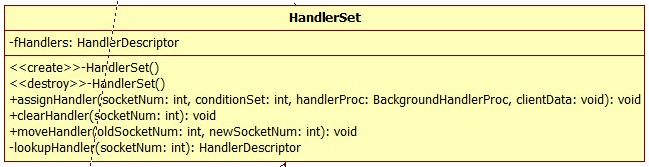
fPrevHandler->fNextHandler = fNextHandler;

}

### 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\* handlerProc, 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;

}

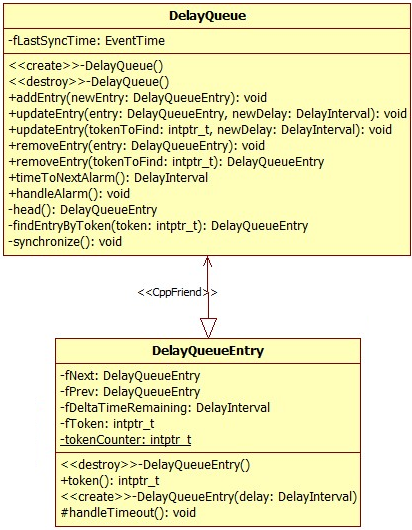
}

## 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



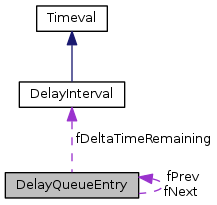
### DelayQueueEntry 延时队列节点类

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

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;

}

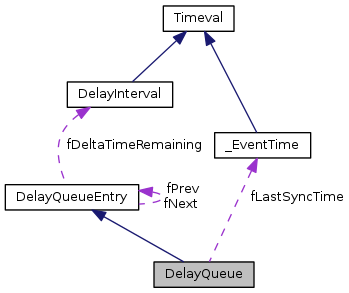
### DelayQueue 延时队列类

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;

}

#### 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();

}

}

### 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。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;

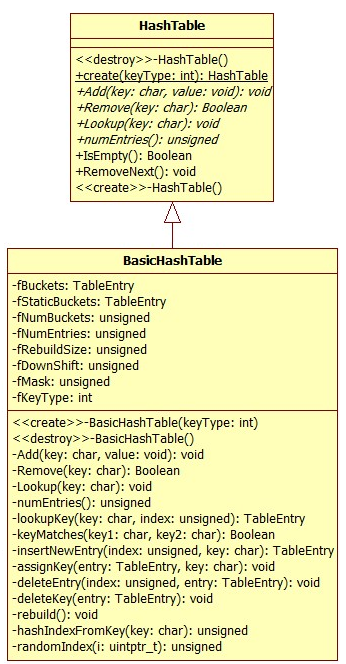
};

## 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



### 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;

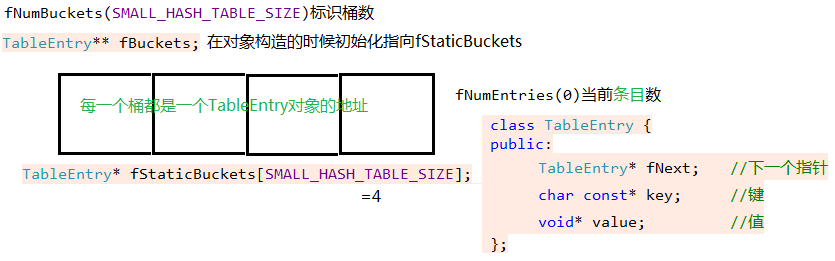
return removedValue;

}

### 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

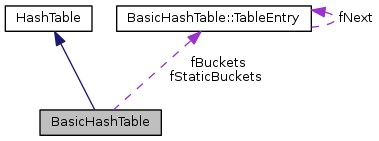


先说说这个BasicHashTable的设计吧。

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。

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;

// 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;

}

#### 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;

}

# 任务调度 TaskSccheduler

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。

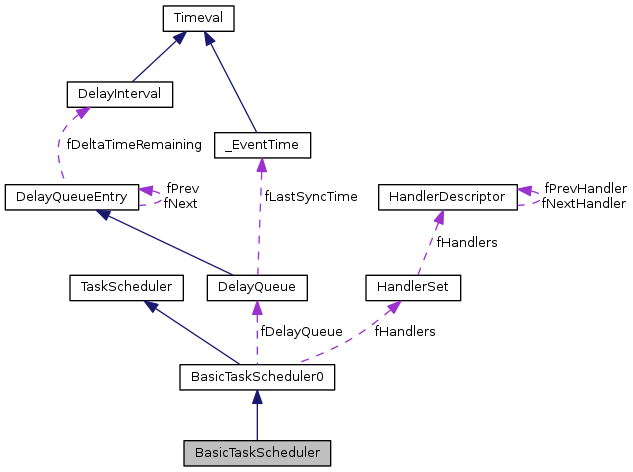
任务调度部分有三个类，其有继承关系。

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。



## 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。

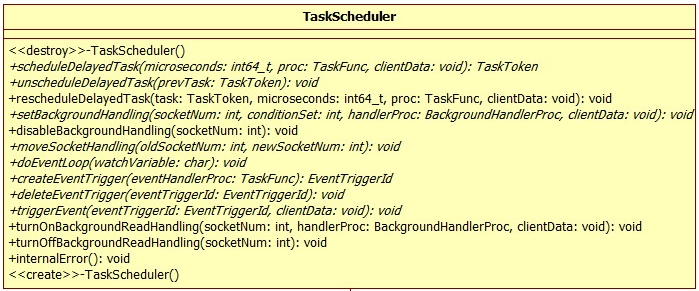
TaskScheduler的默认构造函数是protected权限的，也就是只能被其内部的或派生类的方法调用。

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\* handlerProc, 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\* clientData) {

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 the 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 redefine 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);

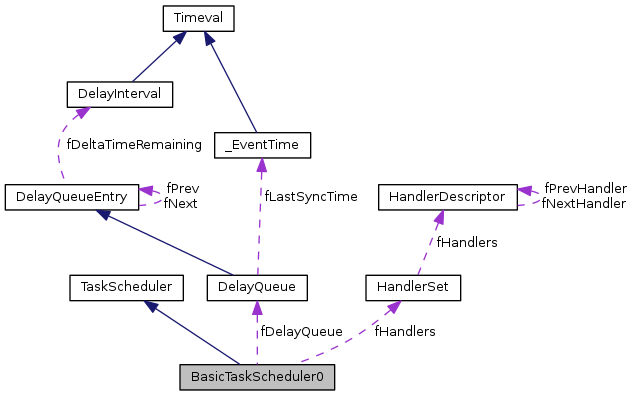
task = scheduleDelayedTask(microseconds, proc, clientData);

}

## 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live555sourcecontrol\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

// 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)

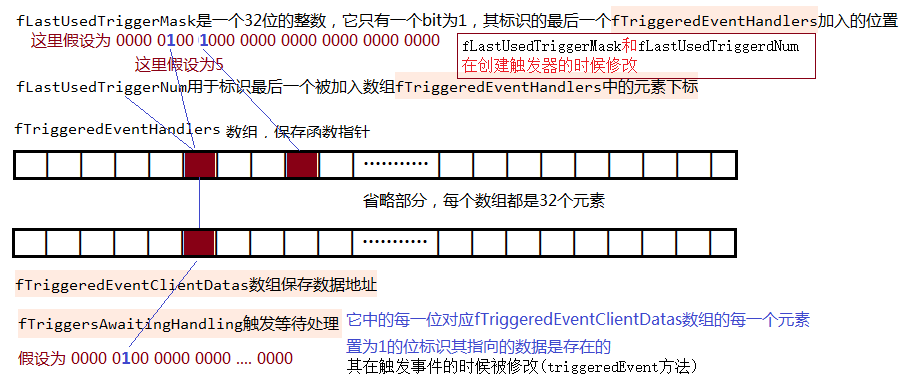
fLastUsedTriggerNum这个参数表示的是最后一个使用触发器的位置(在fTriggeredEventHandlers数组中的下标)。它的范围是[0,31]。将其初始化为31，是因为每一次创建触发器的时候会使用这个值。见createEventTriggerd方法。

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), fLastUsedTriggerNum(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 "EventTriggerId"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 thread, we do this last, to

// reduce the risk of a race condition.)

// 将fTriggersAwaitingHandling中对应的位置1。

fTriggersAwaitingHandling |= eventTriggerId;

}

## 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。

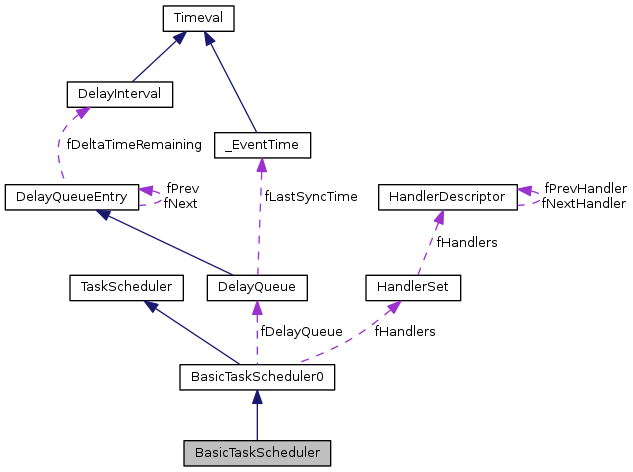
这里添加了四个数据成员，是用来select模型的。关于select模型，这里不解释了。在windows和unix/linux等平台都有相关的API，实现有点差别，但是原理是一致的。

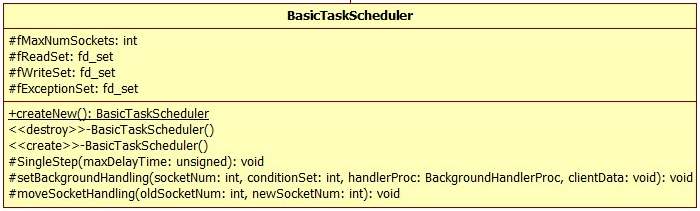
int fMaxNumSockets; //最大的socket数,select调用时提高效率

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





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\_timeToDelay);

\* 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\* handlerProc, void\* clientData);

// 从后台处理移出

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 this 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。

1. 处理等待触发事件集里面的事件。
2. 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\*/) resultConditionSet |= 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\*/) resultConditionSet |= SOCKET\_WRITABLE;

if (FD\_ISSET(sock, &exceptionSet) && FD\_ISSET(sock, &fExceptionSet)/\*sanity check\*/) resultConditionSet |= SOCKET\_EXCEPTION;

if ((resultConditionSet&handler->conditionSet) != 0 && handler->handlerProc != NULL) {

//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;

}

}

// 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[fLastUsedTriggerNum]);

}

}

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 through 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\* clientData) {

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((unsigned)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;

}

}

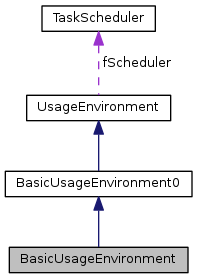
# 使用环境 UsageEnvironment

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

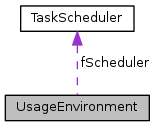
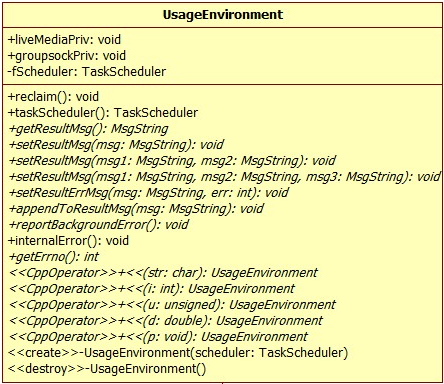
使用环境相关类的作用主要有三个方面，首先UsageEnvrionment包含了任务调度器，可以做任务调度相关操作；其次BasicUasgeEnvironment0 定义了一个buffer(fResultMsgBuffer)，与处理消息的结果相关；最后BasicUasgeEnvironment继承了前两者的功能并添加了向标准错误流输出数据的功能。



## 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;

//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 the 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

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 redefine this, if desired.

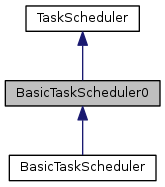
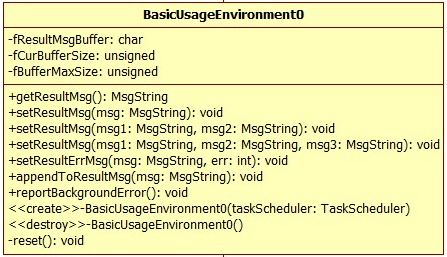
void UsageEnvironment::internalError() {

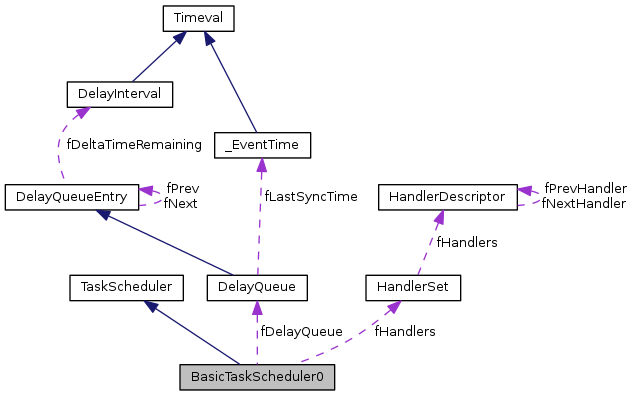
abort();

}

## 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。





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，之间返回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

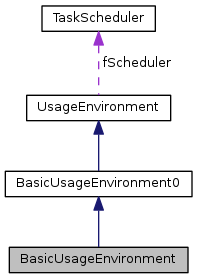
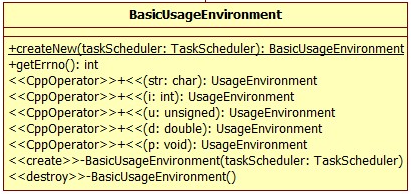
void BasicUsageEnvironment0::reportBackgroundError() {

fputs(getResultMsg(), stderr);

}

## 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;

}

# 网络相关组件类

这些组件都是为GroupSock服务的

## 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(\_WIN32\_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) && !defined(\_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;

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

## 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

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;

}

## 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。

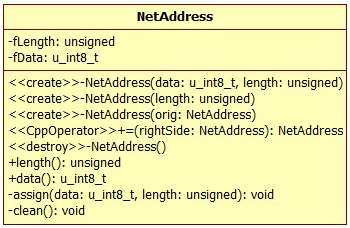
typedef u\_int32\_t netAddressBits;

typedef u\_int16\_t portNumBits;

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

### 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());

}

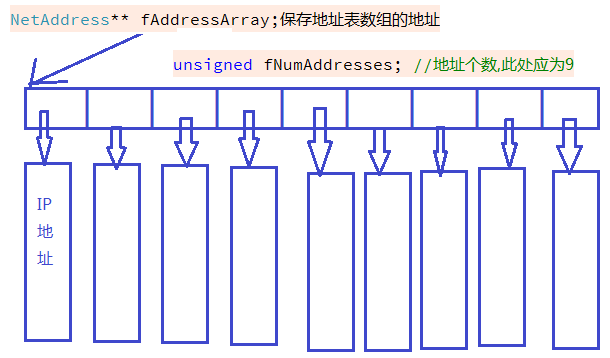
return \*this;

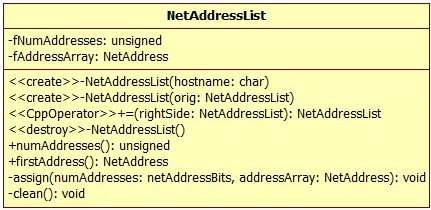
}

### 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

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];

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

}

#### 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++];

}

### 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());

}

### 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()", which 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; }

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, addrNBO & 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;

}

### 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);

}

## 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。

### 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。

##### http://hi.csdn.net/attachment/201109/22/0_13166825985896.gif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;

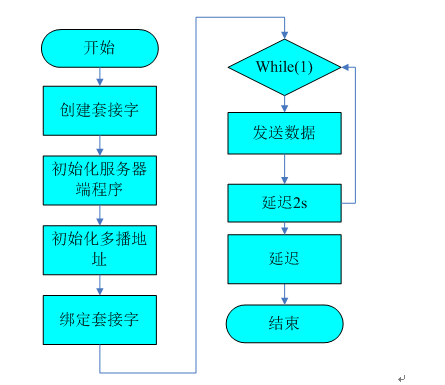
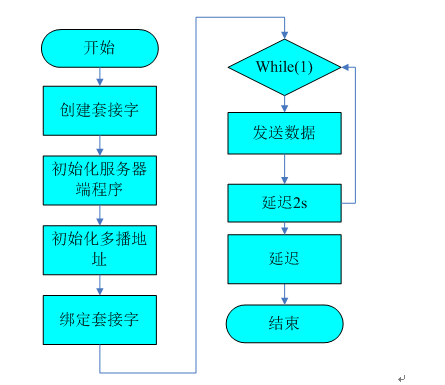
setsockopt(sock\_fd,IPPROTP\_IP,IP\_DROP\_MEMBERSHIP,&merq,sizeof(sreq));

**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。

服务器端程序 客户端程序

### 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);

}

### 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);

}

### 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;

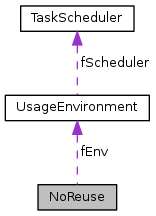
}

}

### 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);

}

### 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

### 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

return sock;

}

### 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

### 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。

### 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);

### 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;

}

### 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

}

### setupStreamSocket设置流式套接口

setupStreamSocket和setupDatagramSocket的功能和像，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;

}

### 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。flag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

#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

### 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, bytesSent, 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。

### 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][ɪnˈkri:s][美][ɪnˈkris]

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);

}

### 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)

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;

}

### 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;

}

### 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;

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 this 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

}

### 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);

}

### 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.QuadPart;

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.QuadPart);

}

#endif

return 0;

}

### 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)) {

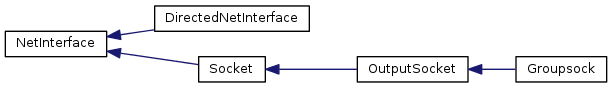
saveEnv.reportBackgroundError(); //报告错误

}

}

## 网络接口相关类

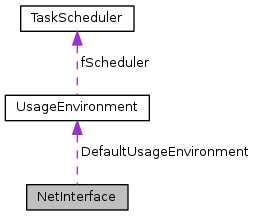
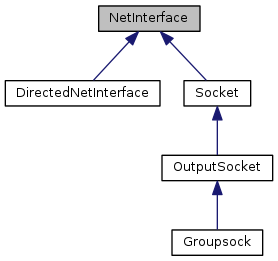
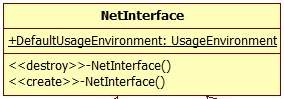
D:\work\live555sourcecontrol\groupsock\include\NetInterface.hh



### 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() {

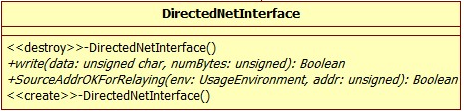
}

NetInterface::~NetInterface() {

}

### 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:

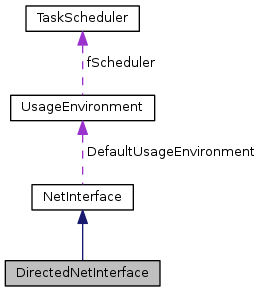
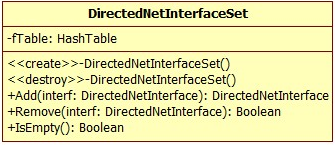
DirectedNetInterface(); // virtual base class

};

### 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);

}

### Socket套接口类

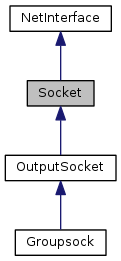
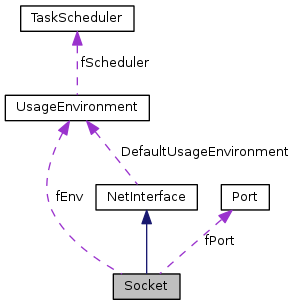
Socket类是一个抽象基类。相较于其基类NetInterFace多了三个数据成员

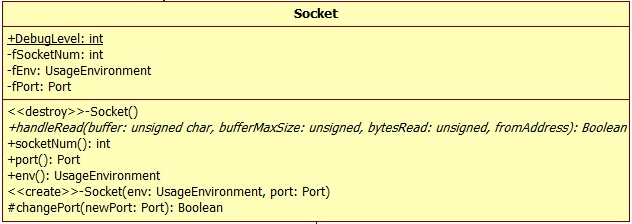
int fSocketNum; // Socket套接口

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





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) {

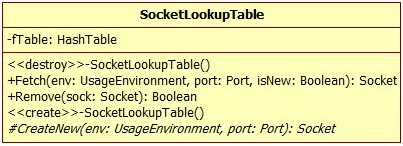
return s << timestampString() << " Socket(" << sock.socketNum() << ")";

}

### 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) {

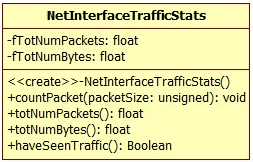
return fTable->Remove((char\*)(long)(sock->port().num()));

}

### 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;

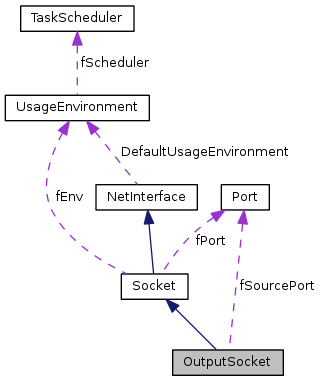
fTotNumBytes += packetSize;

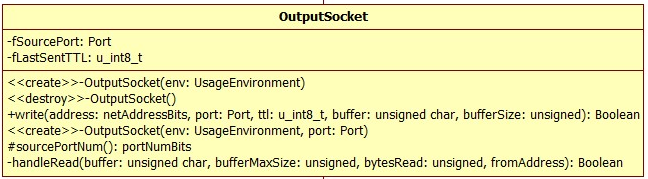
}

### 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

};

#### 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;

}

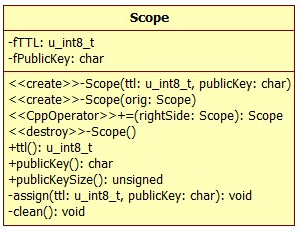
}

return True;

}

### 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

};

#### 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。

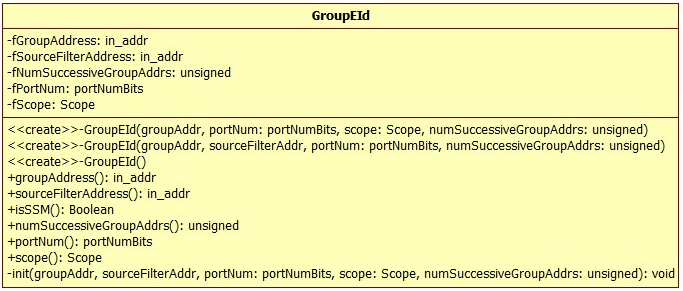
Scope::~Scope() {

clean();

}

### 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; //

};

#### 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

{

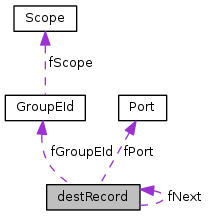
return fSourceFilterAddress.s\_addr != netAddressBits(~0);

}

### 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。

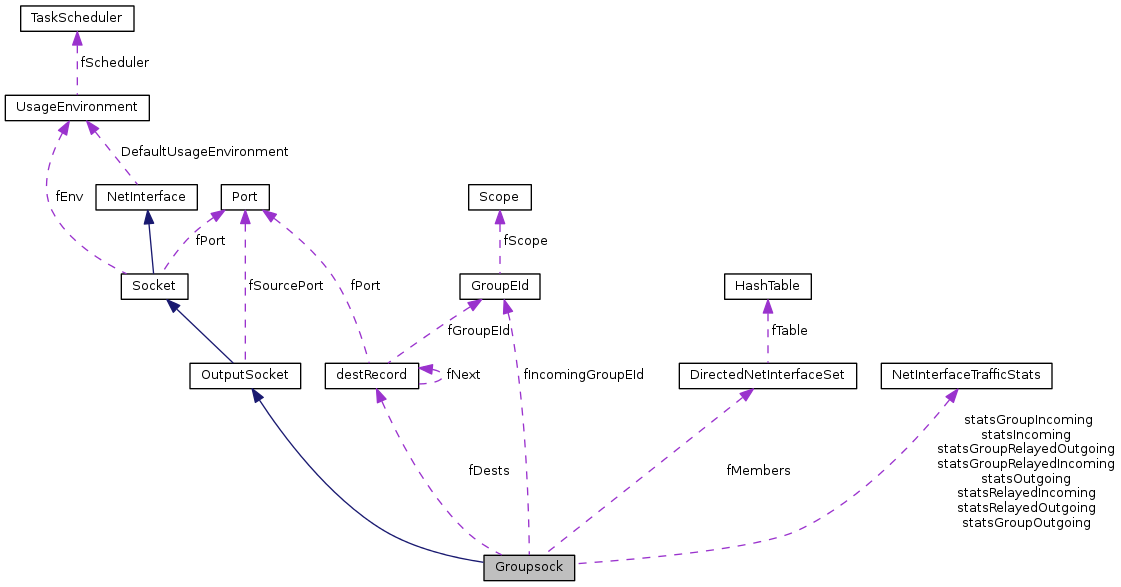
destRecord::~destRecord() {

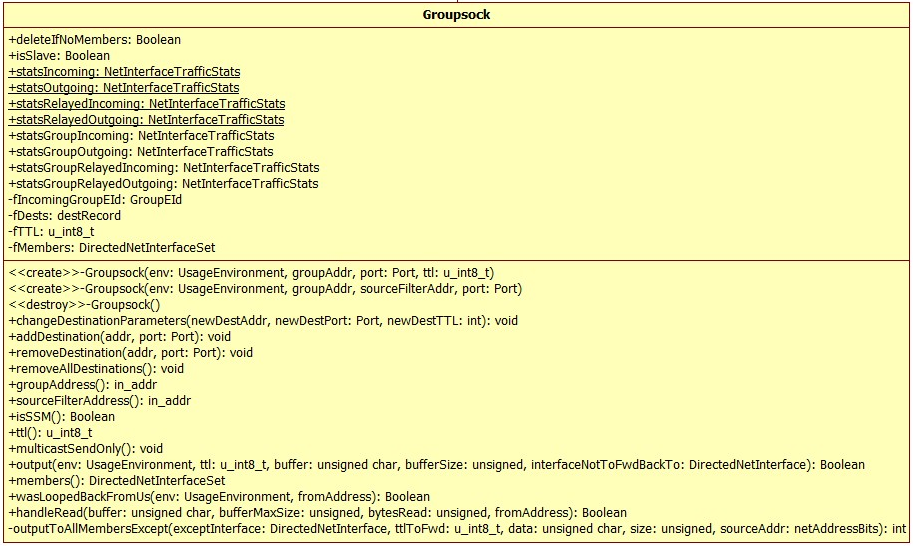
delete fNext;

}

### Groupsock组套接字类

这个类比较复杂。





// 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";

}

}

// 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;

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

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;

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() << ", ";

if (g.isSSM()) {

return s1 << "SSM source: "

<< AddressString(g.sourceFilterAddress()).val() << ")";

}

else {

return s1 << (unsigned)(g.ttl()) << ")";

}

}

### 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;

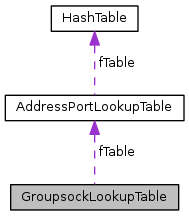
}

### 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);

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());

}

# 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。

