Live555

# 1.文件目录

.

├── BasicUsageEnvironment

├── groupsock

├── liveMedia

├── mediaServer

├── proxyServer

├── testProgs

├── UsageEnvironment

└── WindowsAudioInputDevice

顶层目录结构如上，概述如下：

* UsageEnvironment和BasicUsageEnvironment两个目录中，定义了Live555运行时一定要使用的：UsageEnvironment和TaskScheduler两个功能模块；还定义了Live555中广泛应用的几个数据结构：HashTable，DelayQueue，HandlerSet；
* groupSock目录主要定义了组播模块，当然也实现了一些通用的socket实现和封装；
* liveMedia是媒体相关的主要实现部分，包含了live555支持的所有音视频媒体数据格式的解析、封装、发送等；
* mediaServer实现了RTSP server的相关功能；
* proxyServer实现了网络代理的相关功能；
* WindowsAudioInputDevice

# 2.基础数据结构

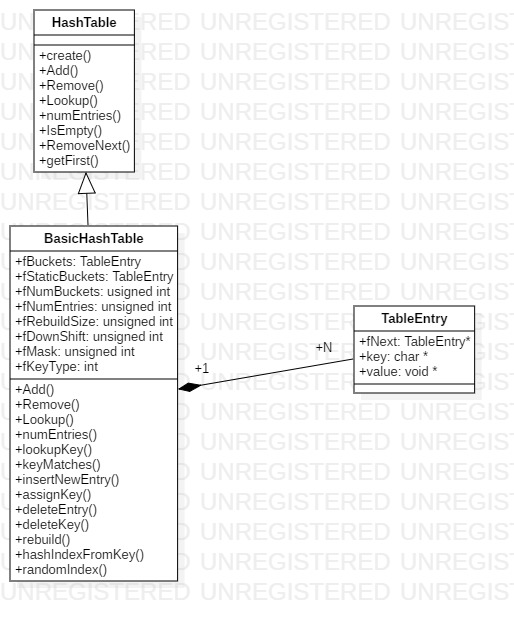
本部分描述live555中实现的几个基础数据结构。

## 2.1.HashTable

涉及到的源文件有:

* UsageEnvironment/HashTable.cpp;
* UsageEnvironment/include/HashTable.hh;
* BasicUsageEnvironment/BasicHashTable.cpp;
* BasicUsageEnvironment/include/BasicHashTable.hh;

其中HashTable是一个基类，BasicHashTable是负责实现具体方法的子类，类图如下：



虽然create创建实例的函数，存在在了HashTable中，但看一下它的实现：

HashTable\* HashTable::create(int keyType) {

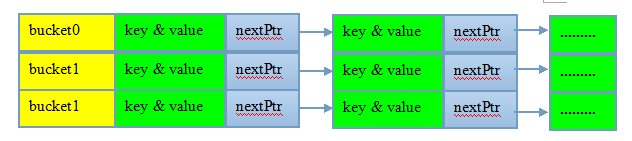
return new BasicHashTable(keyType);

}

create的时候，创建的其实就是BasciHashTable的实例，所以实现的重点还是在BasicHashTable上。

### 2.1.1.数据组织方式

其数据组织模型如下：



与一般的hashTable相同，所有的key都会被以某种特定方式，计算后确定要分配到哪个bucket中(计算方式见2.1.2.章节)，每一个bucket又都是一个单向链表，存储了这个bucket中所有的键值对(TableEntry结构)。

### 2.1.2.bucket键值计算方式

live555提供了两种键值类型：STRING\_HASH\_KEYS和ONE\_WORD\_HASH\_KEYS；相对应的，其键值(bucket)计算方式也不同。

#### STRING\_HASH\_KEYS : fKeyType=0

键值类型是一个string(char \*)，bucket计算方式是：

result = 0;

while (1) {

char c = \*key++;

if (c == 0) break;

result += (result<<3) + (unsigned)c;

}

result &= fMask;

return result;

简单来说，就是逐字节的、3bit为一个单元的左移，左移到字符串结尾后，与fMask进行算术与操作，得到的结果就是bucket值。

fMask是与bucket的数量对应的，初始化的时候bucket数量是4(0 -- 3)，fMask设置的就是3，可以保证与其算术与后的结果，一定是一个有效的bucket；扩容(2.1.4章节)后，fMask也会随之变化，具体参见扩容实现策略。

#### ONE\_WORD\_HASH\_KEYS : fKeyType=1

键值类型是一个unsigned int，bucket计算方式是：randomIndex((unsigned int)key)，该函数的实现是：

unsigned randomIndex(uintptr\_t i) const {

return (unsigned)(((i\*1103515245) >> fDownShift) & fMask);

}

实现起来很简单，但1103515245的选择是否有什么技巧，是否为了避免一些哈希表常见的问题而采用的，我还并不明确。

fDownShift初值为28，伴随着扩容该值也会出现对应的修改，具体参见2.1.4.章节扩容的实现策略。

#### 其他类型 : fKeyType=其他值

虽然live555代码里对于除以上两种情况外，送入其他的fKeyType值也有处理策略，但并不完善，live555中本身并没有除去这两种情况的其他键值类型送入，因此这里只看一下代码实现。

unsigned\* k = (unsigned\*)key;

uintptr\_t sum = 0;

for (int i = 0; i < fKeyType; ++i) {

sum += k[i];

}

result = randomIndex(sum);

**可以看到，其他类型的情况下，live555将key认为是一个可以被unsigned int读取的数组，且有效数据长度一定要最少满足fKeyType的值。这里的隐患就是，如果送入的fKeyType是有效的，但是key的长度不足，那么就有非法内存访问的问题。**

### 2.1.3.每一个bucket对应的链表管理方式

fBuckets可以看作是包含了所有bucket的数组，通过它的index可以快速定位到每一个bucket，并借此遍历该bucket中的所有成员。

每一个bucket都指向了它对应的链表的第一个成员。在插入新成员到链表中时，采用的都是头部插入的方式，也就是说新来的成员作为链表头加入到链表中。

### 2.1.4.扩容策略

#### 扩容条件

live555的hashTable，初始化的时候采用了一个数组进行初始化，也就是：

TableEntry\* fStaticBuckets[SMALL\_HASH\_TABLE\_SIZE];// used for small tables

#define SMALL\_HASH\_TABLE\_SIZE 4

可以看到，数组的size是4。初始化的时候，fBuckets被赋值为fStaticBuckets。

什么时候需要扩容呢？

在插入一个元素到hashTable后，如果元素的总数量fNumEntries大于了fRebuildSize，就会调用rebuild函数进行扩容。

fRebuildSize初始化时赋值为：SMALL\_HASH\_TABLE\_SIZE\*REBUILD\_MULTIPLIER，也就是4\*3=12；

综上，初始化时，fBuckets只能创建0 -- 3之间的buckets组成的hashTable，一旦hashTable中的元素总数超过了12，就会触发rebuild函数进行扩容。

#### rebuild函数

每次增加4倍的buckets数量，例如初始化时fNumBuckets的值是SMALL\_HASH\_TABLE\_SIZE，第一次扩容时，就会将fNumBuckets扩容到：SMALL\_HASH\_TABLE\_SIZE \* 4 = 16；

同样的，fRebuildSize也每次增加4倍。那么第一次扩容时，fRebuildSize就会从12扩容到：12\*4 = 48；

最关键的，是fBuckets，会重新申请一次空间，申请fNumBuckets要求的空间；

还有两个变量很关键，他们关系了HashTable的bucket键值计算，一个是fDownShift，一个是fMask。

fDownShift的值，每次rebuild会自减2；而fMask的处理方式是：fMask = (fMask<<2)|0x3;我们可以做个简单推演：fMask初值是3，也就是二进制的0000 0011，那么第一次扩容时，左移2位，就是0000 1100，再与3做算术或，值就更新成0000 1111，这样就可以将扩容后的所有整数值都覆盖到，作为bucket存在了。

**但是，要注意，极端情况下，伴随着HashTable的size持续增长，fDownShift是有可能变为无意义的值的。当然，在live555当前的数据量级，这个并不太可能出现。**

## 2.2.DelayQueue

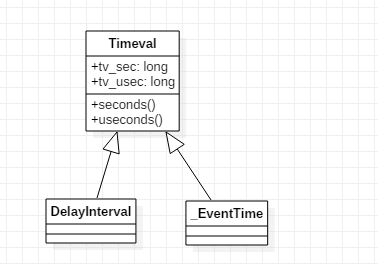
涉及到的源文件有：

* BasicUsageEnvironment/DelayQueue.cpp
* BasicUsageEnvironment/include/DelayQueue.hh

### 2.2.1.TimeVal及其子类

live555中封装了一个TimeVal类，包含的数据成员同样也是tv\_sec和tv\_usec两个，但不同的是，这个类的定位是：既可以表示绝对时间，也可以表示相对时间。这在延时队列中很重要。

该类提供了运算符重载，常见的比较、算术运算符都经过重载可以直接使用。



如上图，Timeval有2个子类，一个是DelayInteval，一个是\_EventTime.

DelayInteval定位是相对时间，或者说是相对于其他时间的时间间隔；\_EventTime定位是1970年开始的时间戳。

对于DelayInteval，我们看一下它定义的几个参数值，就很明白的知道它的定位了：

extern DelayInterval const DELAY\_ZERO; //DELAY\_ZERO(0, 0);

extern DelayInterval const DELAY\_SECOND; //DELAY\_SECOND(1, 0);

extern DelayInterval const DELAY\_MINUTE; //DELAY\_MINUTE=60\*DELAY\_SECOND;

extern DelayInterval const DELAY\_HOUR; //DELAY\_HOUR = 60\*DELAY\_MINUTE;

extern DelayInterval const DELAY\_DAY; //DELAY\_DAY = 24\*DELAY\_HOUR

这是典型的时间间隔计数方式。

对于\_EventTime，看一下它的一个定义：

\_EventTime TimeNow() {

struct timeval tvNow;

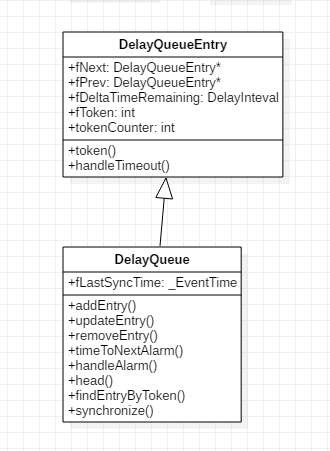
gettimeofday(&tvNow, NULL);

return \_EventTime(tvNow.tv\_sec, tvNow.tv\_usec);

}

很显然，它通过gettimeofday函数拿到的是系统时间戳。

### 2.2.2.DelayQueueEntry



DelayQueueEntry是DelayQueue的父类，其数据成员中的fNext和fPrev显然是组织双向链表时需要用到的指针。

fDeltaTimeRemaining是一个DelayInteval类型的对象，代表了距离当前时间还要多久就该调用到这个延时任务了。

tokenCounter是一个static类型的变量，初值为0，每当一个DelayQueueEntry类型的对象创建时，该值自增1，标记的是对象个数。

每个DelayQueueEntry对象中都有fToken变量，各个对象该值不同，标记每个不同的变量。

**这里tokenCounter其实没有被外部使用到，外部只是使用到了：通过fToken的值能够定位到特定的DelayQueueEntry对象这个特性。其实tokenCounter是有问题的，在每一个DelayQueuEntry对象销毁的时候，tokenCounter应该自减1。但由于没有使用到，所以没有出现任何问题。**

### 2.2.3.DeleyQueue

#### 数据组织方式

DelayQueue是一个按照相对延时时间排列的双向链表。

假设有2个延时任务，一个相对于当前时间的3秒之后执行，一个相对于当前时间的5s之后执行，那么在DelayQueue中，就会有2个节点：

* 第一个节点是相对于当前时间的延时时间，是3s，没有异议；
* 第二个节点并不是相对于当前时间的5s，而是相对于第一个节点的相对时间，或者说，第一个节点的任务执行完之后多久，第二个节点才执行，那么显然这里第二个节点的值，就是5-3 = 2s了；

#### 延时时间更新策略

如果系统时间被改变了，或者时间发生了变化(秒级)，都会通过调用synchronize函数，更新整个DelayQueue中的所有节点的延时时间。更新算法描述如下：

1. 获取当前系统时间timeNow，注意，这里拿的是相对于1970年的绝对时间；
2. 如果fLastSyncTime大于timeNow，说明系统时间被手动向回调整过，例如从2点调整到1点这种，那么将fLastSyncTime更新为timeNow，退出函数；
3. 变量timeSinceLastSync=timeNow - fLastSyncTime，定义的是上次更新到现在经历了多少秒；
4. DelayQueue中每个成员：
   1. 如果其fDeltaTimeRemaining小于timeSinceLastSync，说明已经超过了其该执行的时间还未执行，直接将fDeltaTimeRemaining置零，后续将直接执行该任务；
5. 执行完step4之后，找到了第一个fDeltaTimeRemaining 大于timeSinceLastSync的成员，对其：fDeltaTimeRemaining -= timeSinceLastSync;更新其fDeltaTimeRemaining。**退出程序**。

要注意，step5并不是将后面的所有成员的时间都更新，而是只更新第一个fDeltaTimeRemaining 大于timeSinceLastSync的成员的就可以。

还是以上面的例子为例说明：

1. 开始的时候，2个节点，一个是3s，第二个是2s；
2. 假设1s后，更新函数被调用，那么2个节点的时间，分别更新为2s，2s；
3. 1s后，更新为：1s，2s；
4. 1s后，第一个任务执行，第二个任务成为第一个任务，2s后被执行。

**这里要注意，由于判断相对延时时间，使用的都是gettimeofday的绝对时间，因此对于设备修改系统时间的情况，可能会造成延时任务长时间不被执行，更多时候，这其实不是业务调用者的本意！！**

#### 任务执行入口 -- handleAlarm

DelayQueue的任务，如果fDeltaTimeRemaining等于DELAY\_ZERO，说明应该马上执行了，这时候会调用该节点的handleTimeout函数执行具体操作。

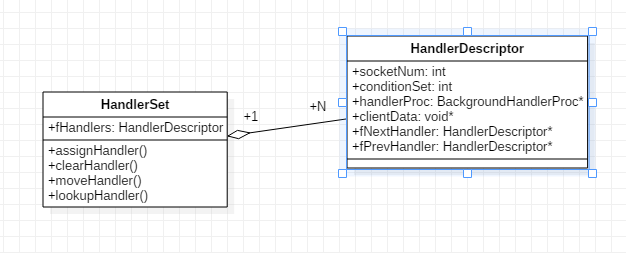
DelayQueueEntry的handleTimeout函数，默认情况下只会删除其占用的内存空间，并不会做其他业务操作；但是我们可以延伸一下，看看任务调度器中的实现。任务调度器中，实现了AlarmHandler，它是DelayQueueEntry的子类，它重写了handleTimeout函数，在其中就加入了自己的业务函数。

## 2.3.HandlerSet

涉及到的文件：

* BasicUsageEnvironment/BasicTaskScheduler0.cpp;
* BasicUsageEnvironment/include/HandlerSet.hh;

HanlderSet的数据结构，是一个set(集合)。定位是存储所有socket相关的请求。



HanlderDescriptor类，包含了一个socket连接所需要的所有参数：

* socketNum：socket的id；
* conditionSet：socket的condition，例如可读、可写、异常等；
* handlerProc：socket接收到事件后要调用的函数；
* clientData：传递给handlerProc的参数；

HandlerSet则是以一个双向链表的形式，将所有注册来的Handler组织起来，等待任务管理器在接收到对应的socket请求的时候，进行处理。

# 3.基础环境和任务调度器

# 4.组播协议实现