

live555

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

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

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

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

http://blog.csdn.net/qz_29350001/article/details/77962082

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

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

Live555 源码主要由八个部分组成：

UsageEnvironment, BasicUsageEnvironment, groupsock, liveMedia, mediaServer, proxyServer, testProgs, WindowsAudioInputDevice。

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

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

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

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

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

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

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

UsageEnvironment 中的 ["UsageEnvironment"](#) 和 ["TaskScheduler"](#) 类用于调度延迟的事件，为异步的读事件分配处理程序，以及输出错误/警告消息。**UsageEnvironment** 中的 ["HashTable"](#) 类还为范型哈希表定义了接口，由其余的代码使用。**UsageEnvironment** 中的都是抽象类；它们必须在实现中被继承。这些子类可以利用它运行的环境的特定属性，比如它的 GUI 和/或脚本环境。

Boolean.hh: 定义跨平台布尔类型以及True和Flase的值；

strDup.hh和**strDup.cpp**: 字符串拷贝操作；

//拷贝str的数据，返回指针指向新地址，需外部释放内存，内部使用new操作符分配内存，外部需要对应使用delete释放内存。

`char* strDup(char const* str);`

//返回与str空间大小相同的内存块，需外部释放内存

`char* strDupSize(char const* str);`

//返回与str大小相同的内存块，通过参数resultBufSize待会str的内存大小，需外部释放内存

`char* strDupSize(char const* str, size_t& resultBufSize);`

补充知识点: new和malloc的区别？

new和delete是操作符，malloc和free是库函数；

new的时候，先申请内存，释然调用构造函数初始化，delete的时候先析构该对象，然后释放内存。malloc只是在堆上申请内存，然后强制转换成需要的类型，free的手直接释放内存；

补充知识点:

`char* const cp;`

常量指针cp，指向char类型，指向内存位置可读写，cp不可修改，且初始化时需赋值；

`char const *cp;`

`const char* cp;`

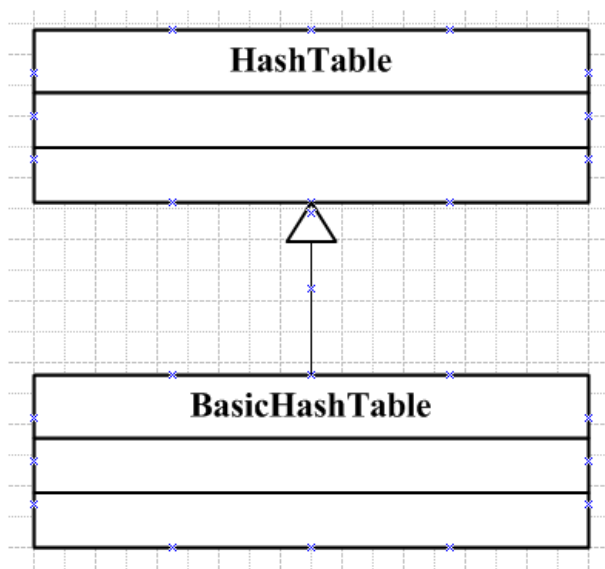
cp指针指向const char类型，cp可以修改，指向的值只读，不可改；

关键看const修饰谁，修饰哪个参数，对应值不可改；由于没有const*，若出现const*，则const修饰前面的参数。

补充知识点:

在基类中虚函数为public，在子类中public继承，但是将虚函数修改为private属性，这么做地目的：只允许通过接口调用；

HashTable类: 哈希表实现范性关联容器，基类用于规范接口，内部定义了迭代器类Iterator，用来遍历容器；为了便于分析，和**BasicHashTable**类一起分析。一个可用的容器，必须提供增、删、查和改操作，还需要能够知道容器当前容量大小和递归访问。



`virtual void* Add(char const* key, void* value) = 0;`

完成增加和修改功能；

修改：若存在相同的key值，返回key对应的原始值，用value替换原始值；

增加：若不存在key，新增一个键-值对，插入到链表头部，返回NULL；

```
virtual Boolean Remove(char const* key) = 0;
```

完成删除功能；

删除：若存在键为key的键-值对，删除该键-值对，返回True；若不存在返回False；

```
virtual void* Lookup(char const* key) const = 0;
```

完成查找功能；

查找：若存在键为key的键-值对，则返回该键-值对；否则返回NULL；

```
virtual unsigned numEntries() const = 0;
```

返回容器中，键-值对的数量；

```
void* RemoveNext();
```

删除下一个键-值对；从当前容器中的第一个键-值对开始删除；

```
void* getFirst();
```

返回当前容器中的第一个键-值对；

增：对key值进行哈希计算，获取索引，在索引处，将该键-值对新增到链表头；

改：对key值进行哈希计算，获取索引，遍历索引处的链表，当key值相同时，替换key对应的值；

删：对key值进行哈希计算，获取索引，遍历索引处的链表，当key值相同时，删除该键-值对；

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

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

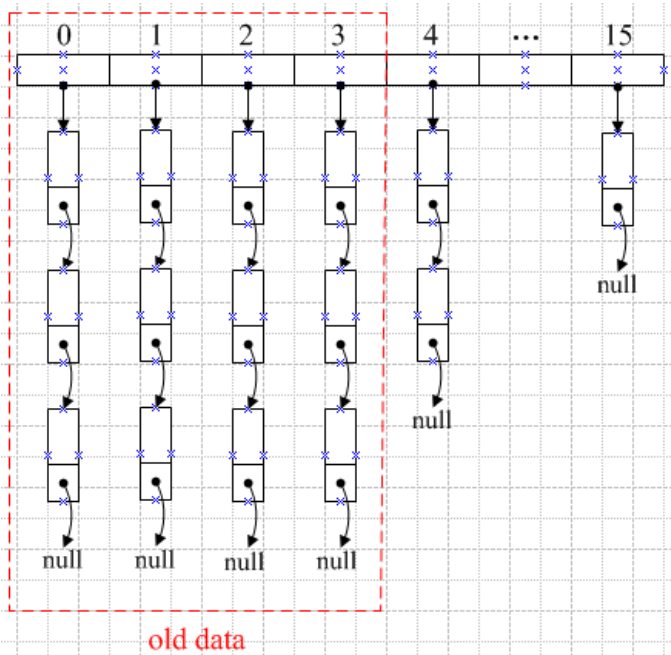
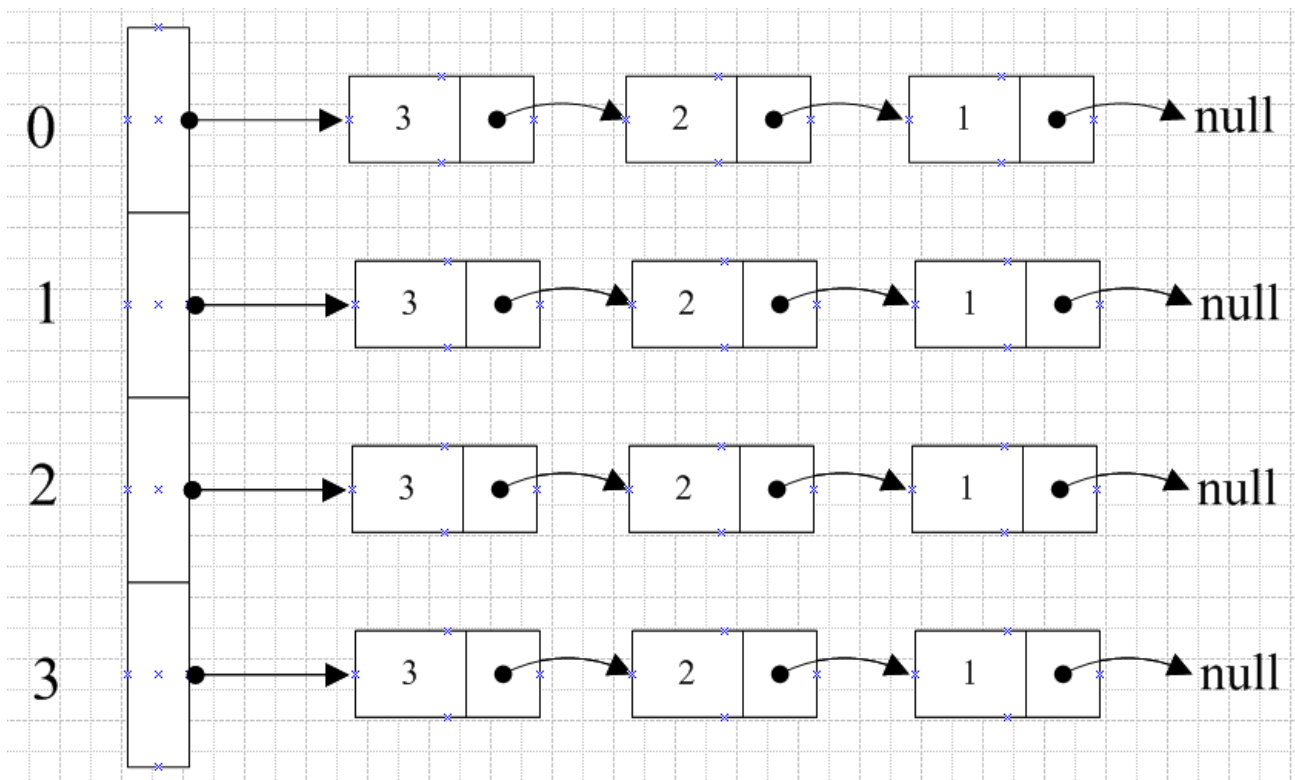
如上截图，fMask初始值为0x3，哈希函数只会产生0，1，2，3共四个索引值；当哈希表需要重建时，fMask左移2位或上原值，此时fMask为0xf，此时，哈希函数会产生哈希值0，1，2，...，15；如下图，哈希索引相同时，将新的键-值对插入链表头。

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

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

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

$$4^N * 12$$



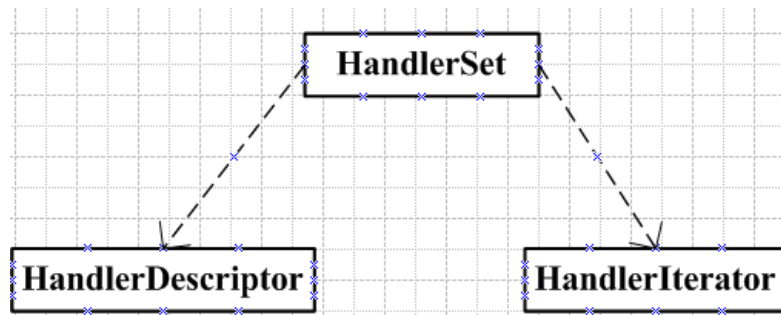
补充知识:

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

补充知识:

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

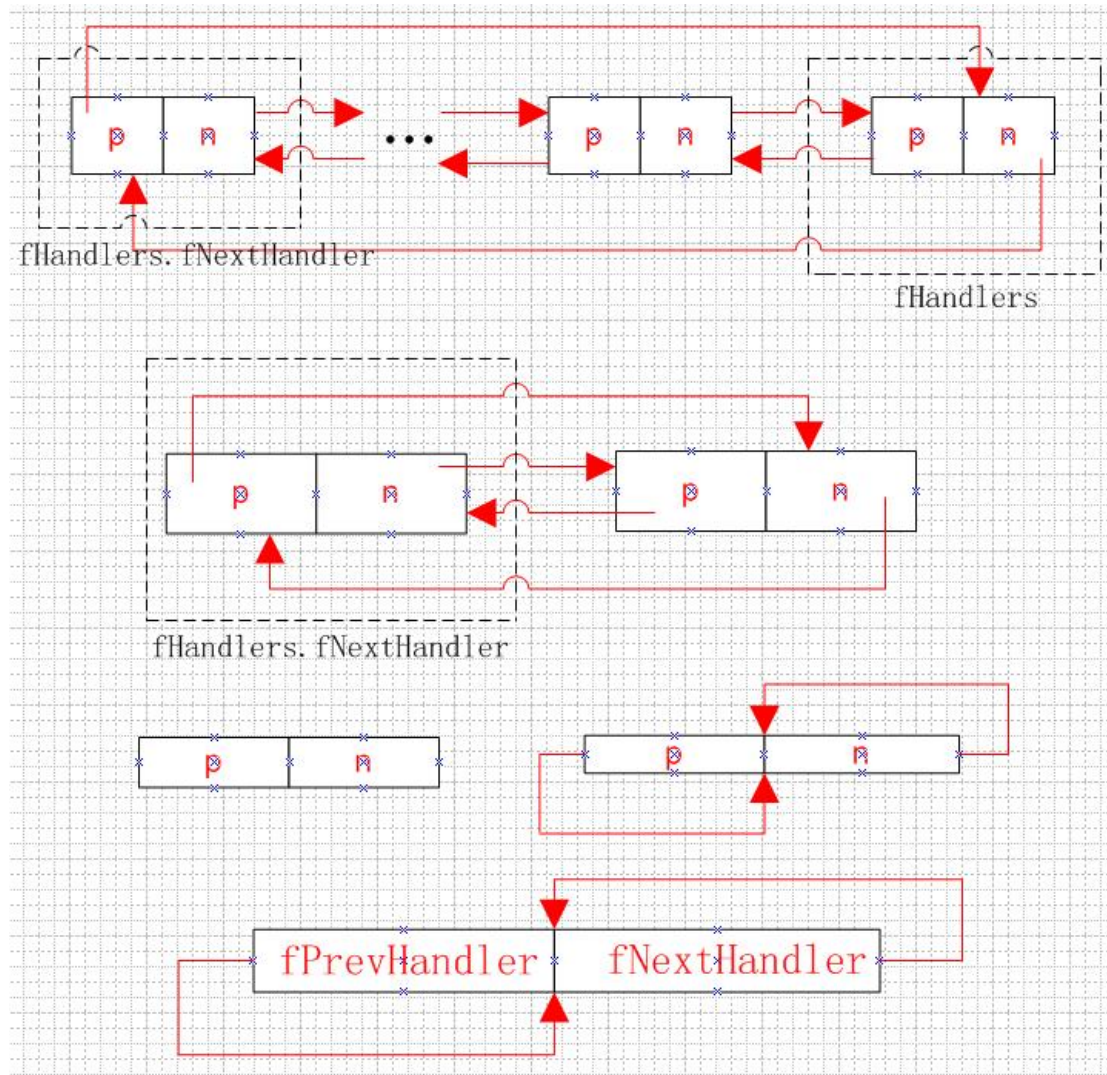
HandlerSet: 双向链表，用来处理IO事件；新事件总是加入链表头；



HandlerSet类就是一个双向链表集合；

HandlerDescriptor类用来实现双向链表；

HandlerIterator类是为了能够递归的处理双向链表中的节点；



```

HandlerDescriptor::HandlerDescriptor(HandlerDescriptor* nextHandler)
: conditionSet(0), handlerProc(NULL) {
// Link this descriptor into a doubly-linked list:
if (nextHandler == this) { // initialization
    fNextHandler = fPrevHandler = this;
} else {
    fNextHandler = nextHandler;
    fPrevHandler = nextHandler->fPrevHandler;
    nextHandler->fPrevHandler = this;
    fPrevHandler->fNextHandler = this;
}
}
  
```

```

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

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

```

从上图可以看到，在HandlerSet的构造函数中，通过fHandlers(&fHandlers)初始化了一个HandlerDescriptor对象。通过观察HandlerDescriptor类的构造函数，得出结论：fHandlers的fNextHandler和fPrevHandler均指向自己，fHandlers为双向链表的第一个节点；在HandlerDescriptor的构造函数中，对形参nextHandler的值进行操作。指针作为形参时，对其值进行操作，会影响实参的对应值。所以HandlerDescriptor构造函数中修改nextHandler，对应的HandlerSet中的fHandlers也被对应修改了；修改后，fHandlers和其指针如上图所示，所以fHandlers是双向链表的尾节点。

//向双向链表中添加IO事件，socketNum为文件描述符，conditionSet为读|写|异常状态，handlerProc为回调函数，clientData为用户数据，回调函数带回给用户

```
void assignHandler(int socketNum, int conditionSet, TaskScheduler::BackgroundHandlerProc* handlerProc, void* clientData);
```

//从双向链表中，删除socketNum对应的IO事件

```
void clearHandler(int socketNum);
```

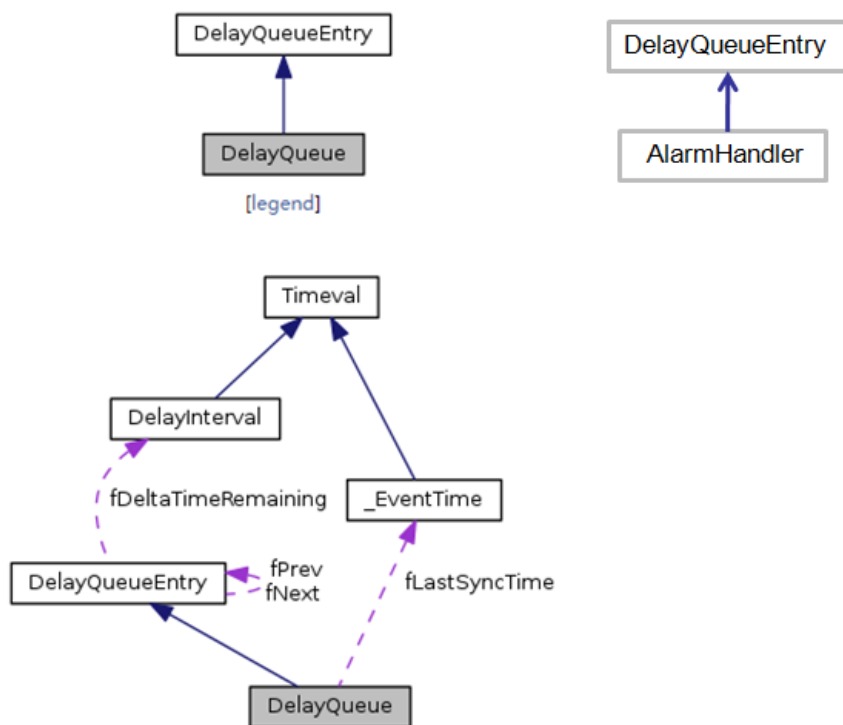
//更新IO事件，将套接字oldSocketNum的事件的套接字修改为newSocketNum

```
void moveHandler(int oldSocketNum, int newSocketNum);
```

补充知识：

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

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



Timeval：时间处理基类；

DelayInterval：时间间隔，用来处理相对时间；

_EventTime：延时事件的基准时间；

DelayQueueEntry：描述一个延时事件；

DelayQueue: 延时事件组成的双向列表;

AlarmHandler: 定时器时间;

//向双向列表中, 增加延时事件, 加入后才开始计算超时

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

//更新定时事件

`void updateEntry(DelayQueueEntry* entry, DelayInterval newDelay);`

`void updateEntry(intptr_t tokenToFind, DelayInterval newDelay);`

//从双向列表中删除entry事件

`void removeEntry(DelayQueueEntry* entry); // but doesn't delete it`

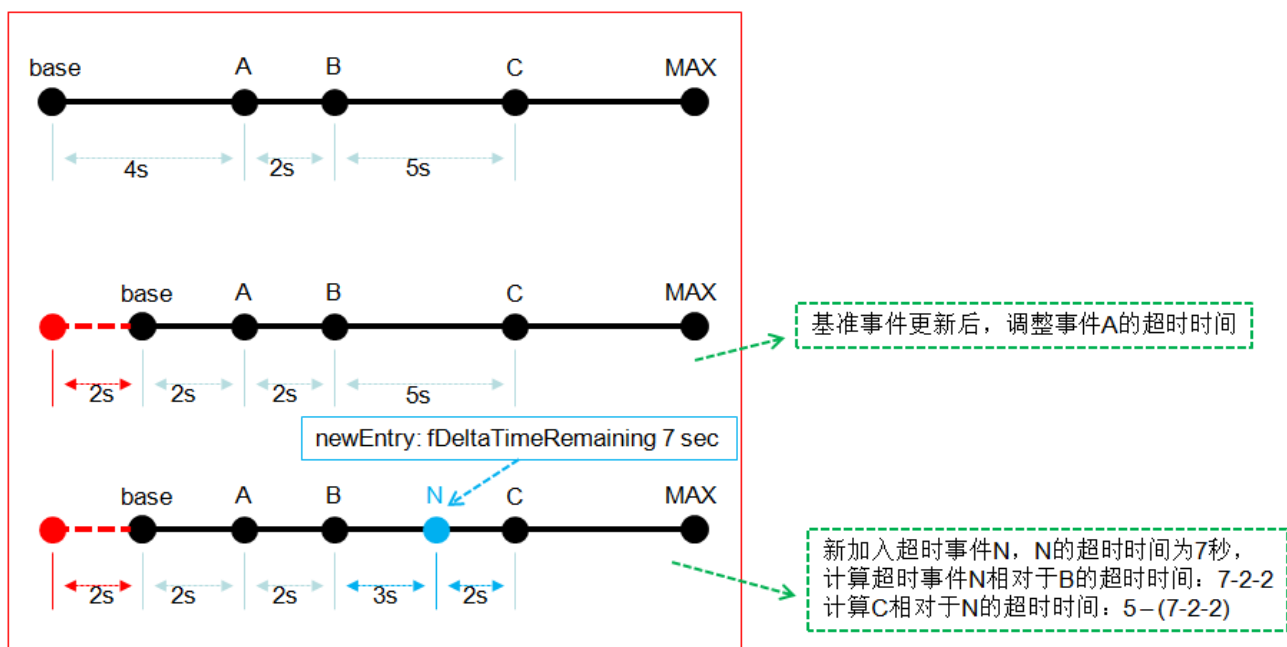
`DelayQueueEntry* removeEntry(intptr_t tokenToFind); // but doesn't delete it`

//下一个超时事件的时间间隔, 距离现在基准事件的时间

`DelayInterval const& timeToNextAlarm();`

//处理定时事件

`void handleAlarm();`



假如我们要描述一个事件发生的时间, 可以有两种方法:

- a、一种方法直接描述事件发生的绝对时间;
- b、另一种则是可以描述和另一事件发生的相对时间。

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

```
114 DelayQueue::DelayQueue()
115 : DelayQueueEntry(ETERNITY) {
116     fLastSyncTime = TimeNow();
117 }
28 AlarmHandler(TaskFunc* proc, void* clientData, DelayInterval timeToDelay)
29 : DelayQueueEntry(timeToDelay), fProc(proc), fClientData(clientData) {
30 }
63 DelayInterval timeToDelay((long)(microseconds/1000000), (long)(microseconds%1000000));
64 AlarmHandler* alarmHandler = new AlarmHandler(proc, clientData, timeToDelay);
65 fDelayQueue.addEntry(alarmHandler);
```

live555中的DelayQueue, 初始化的时候, 设置了最大超时时间:

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

```

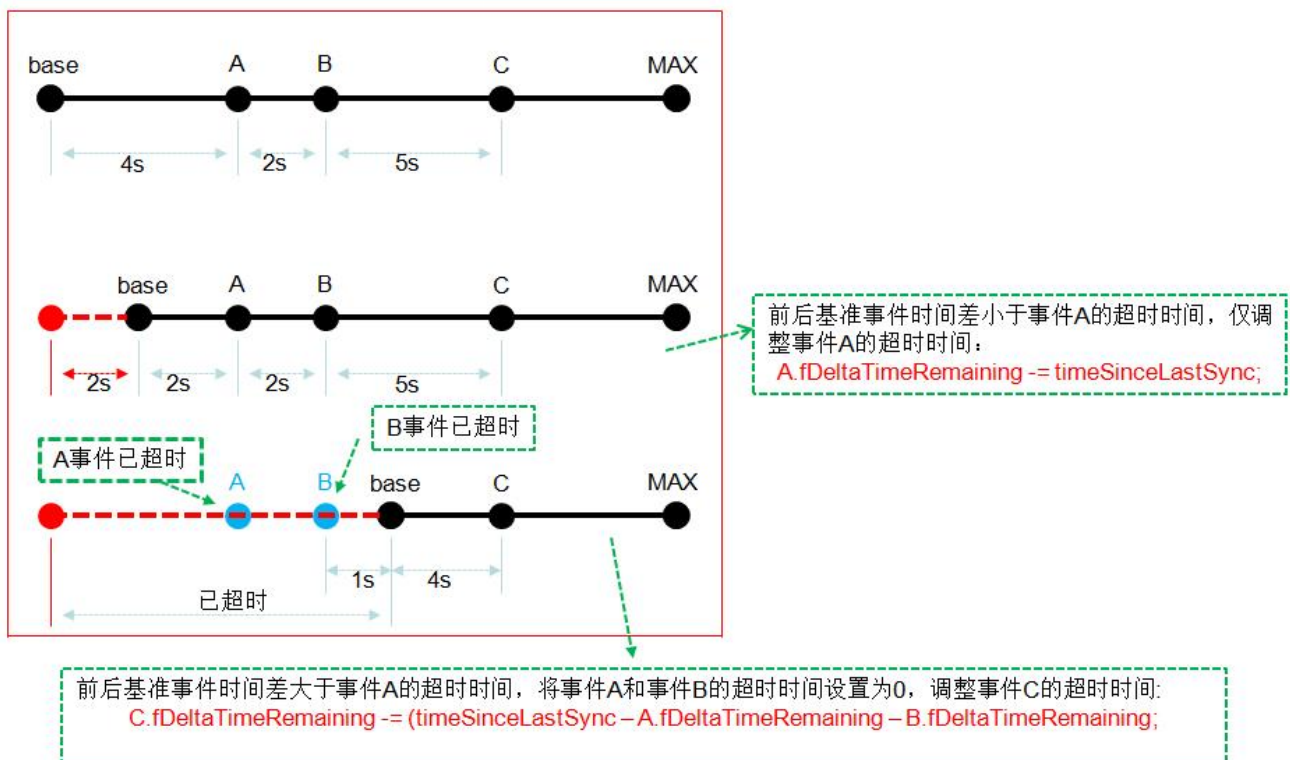
202 void DelayQueue::synchronize() {
203     // First, figure out how much time has elapsed since the last sync
204     _EventTime timeNow = TimeNow();
205     if (timeNow < fLastSyncTime) {
206         // The system clock has apparently gone back in time; reset our
207         fLastSyncTime = timeNow;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

```

首先分析DelayQueue::synchronize()函数

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

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

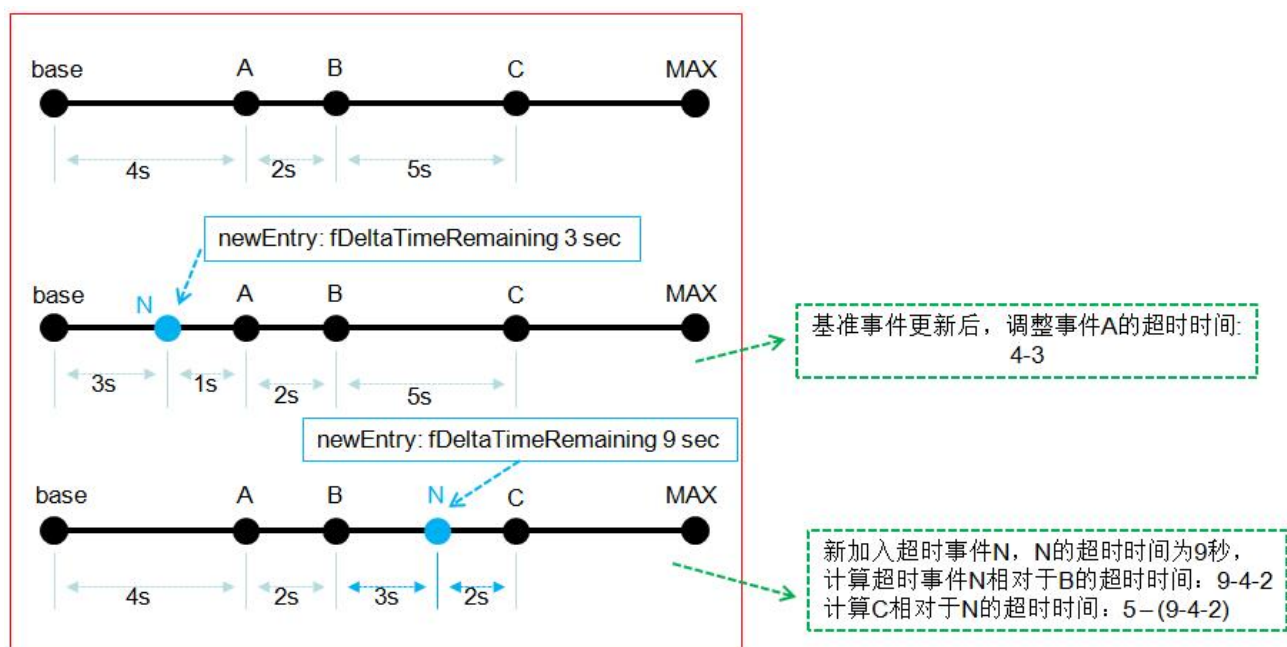



```

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

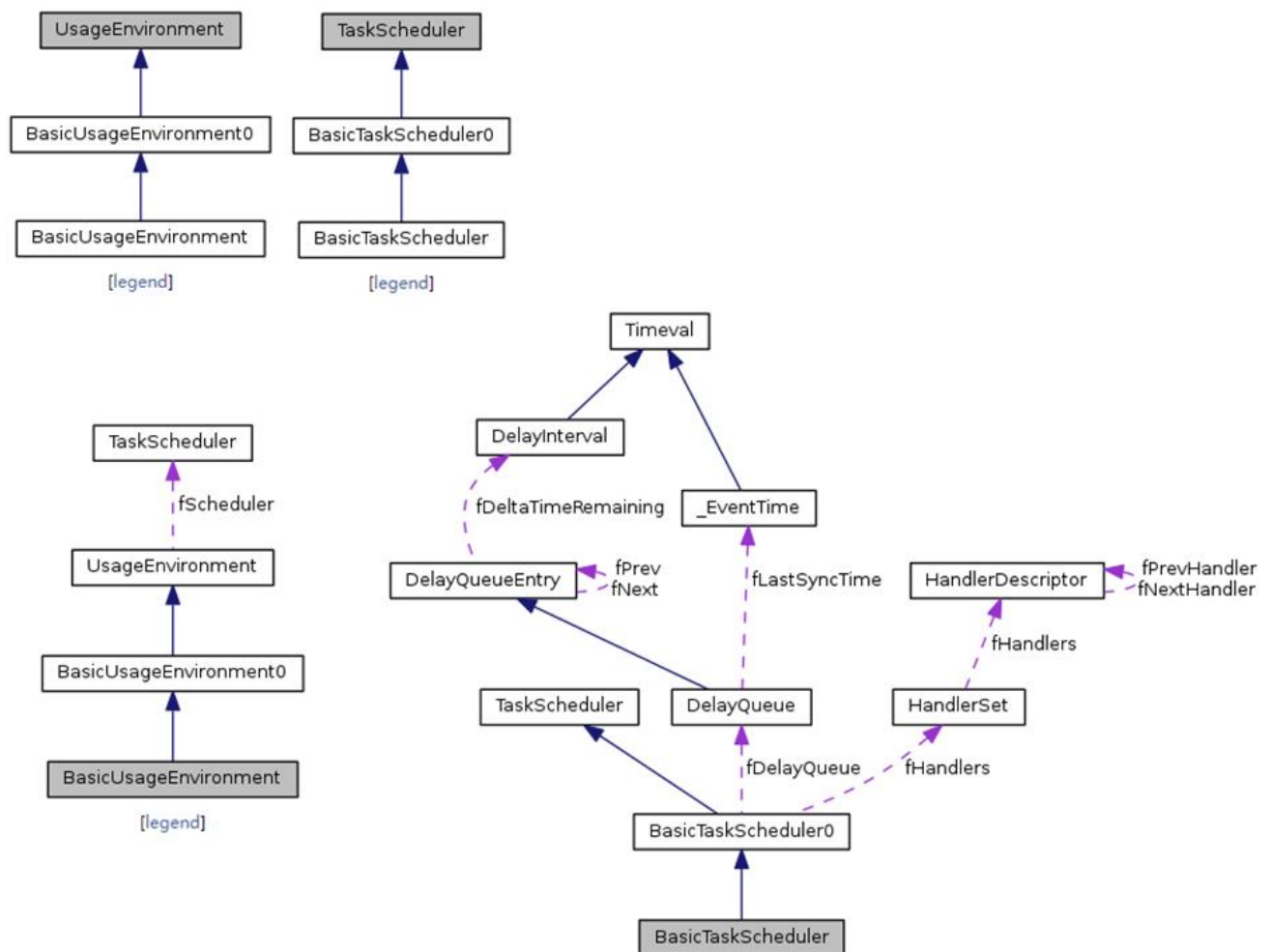
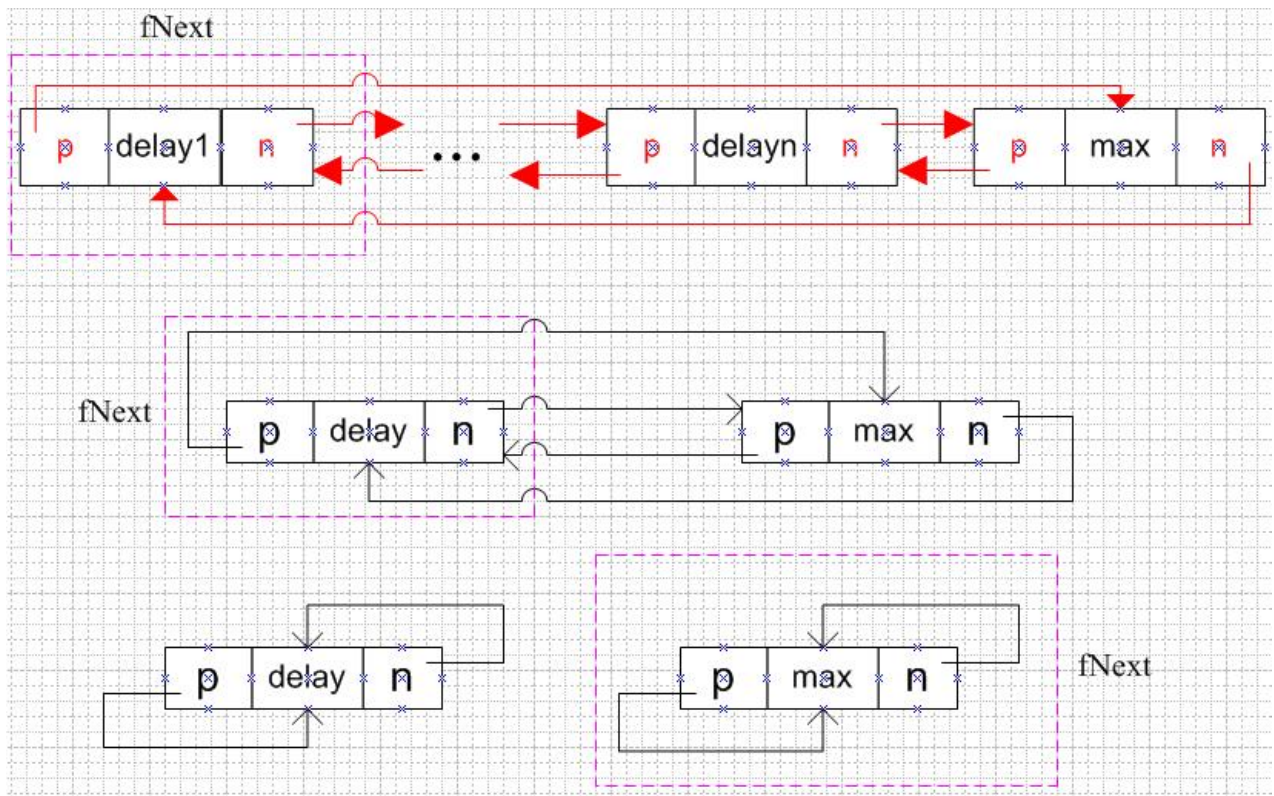
```

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



DelayQueue的双向链表结构如下图:

- DelayQueue的fNext总是指向链表头，指向最先超时的定时事件；
- DelayQueue的尾节点超时时间为1635年，理论上不会超时；



补充知识:

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

memmove的处理措施:

- (1) 当源内存的首地址等于目标内存的首地址时，不进行任何拷贝;

- (2) 当源内存的首地址大于目标内存的首地址时，实行正向拷贝；
- (3) 当源内存的首地址小于目标内存的首地址时，实行反向拷贝，反向拷贝避免覆盖；

补充知识：

- (1) 基类构造函数声明为protected:

构造函数从基类开始执行，先执行基类构造函数，再执行子类构造函数；

构造函数声明为protected，就不能直接生成基类对象了，在子类中存在静态函数，用来生成子类对象，单例大多是这种情况；

- (2) 基类的析构函数声明为public virtual属性:

析构函数声明为virtual时，析构时从子类开始的，先调用子类析构函数，在调用基类的析构函数；不带virtual属性时，直接调用基类的析构函数，将导致内存泄漏；

live555共有三种事件：

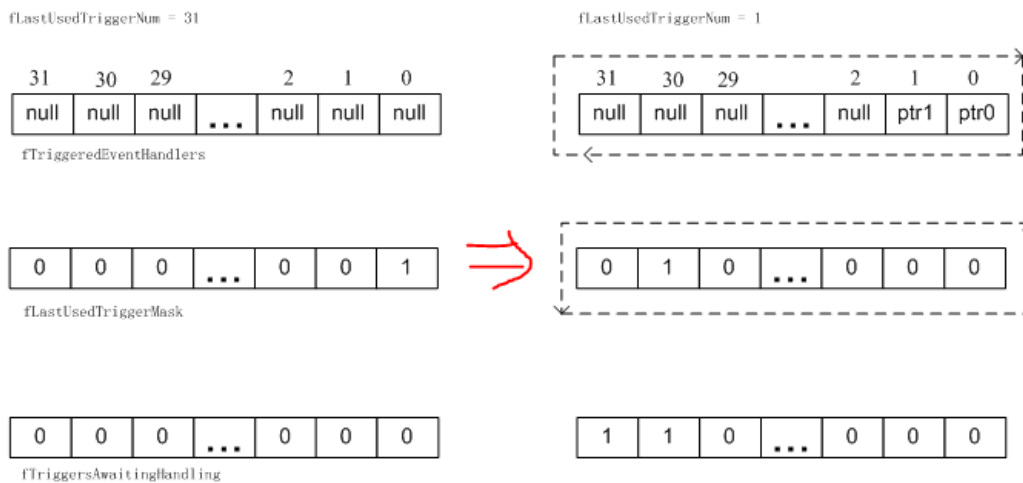
- a、I/O复用事件； --HandlerSet -->重复性事件
- b、触发器事件； --EventTriggerId -->一次性事件
- c、延迟任务事件； --DelayQueue -->一次性事件

BasicUsageEnvironment类表示用户的使用环境，主要就三种功能：

- 1、获取/设置错误消息字符串；
- 2、重载操作符；
- 3、持有的TaskScheduler，用来调度live555的三种事件；

触发器事件：

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



```
46 BasicTaskScheduler::BasicTaskScheduler()
47 : fLastHandledSocketNum(-1), fTriggersAwaitingHandling(0), fLastUsedTriggerMask(1), fLastUsedTriggerNum(MAX_NUM_EVENT_TRIGGERS-1) {
```



```

84 EventTriggerId BasicTaskScheduler0::createEventTrigger(TaskFunc* eventHandlerProc) {
85     unsigned i = fLastUsedTriggerNum;
86     EventTriggerId mask = fLastUsedTriggerMask;
87
88     //遍历32个槽位，找到可用槽位，返回mask
89     do {
90         i = (i+1)%MAX_NUM_EVENT_TRIGGERS;
91         mask >>= 1;
92         if (mask == 0) mask = 0x80000000;
93
94         if (fTriggeredEventHandlers[i] == NULL) {
95             // This trigger number is free; use it:
96             fTriggeredEventHandlers[i] = eventHandlerProc;
97             fTriggeredEventClientDatas[i] = NULL; // sanity
98
99             fLastUsedTriggerMask = mask;
100             fLastUsedTriggerNum = i;
101
102             return mask;
103         }
104     } while (i != fLastUsedTriggerNum);
105
106     // All available event triggers are allocated; return 0 instead:
107     return 0;
108 }
109
110 void BasicTaskScheduler0::triggerEvent(EventTriggerId eventTriggerId, void* clientData) {
111     // First, record the "clientData". (Note that we allow "eventTriggerId" to be a combination of bits for multiple events.)
112     EventTriggerId mask = 0x80000000;
113     for (unsigned i = 0; i < MAX_NUM_EVENT_TRIGGERS; ++i) {
114         if ((eventTriggerId & mask) != 0) {
115             fTriggeredEventClientDatas[i] = clientData;
116         }
117         mask >>= 1; //这里说允许eventTriggerId是多个Id的或组合，所以在满足if条件时，没有break
118     }
119
120     // Then, note this event as being ready to be handled.
121     // (Note that because this function (unlike others in the library) can be called from an external thread, we do this last, to
122     // reduce the risk of a race condition.)
123     fTriggersAwaitingHandling |= eventTriggerId;
124 }

```

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

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

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

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

//创建触发事件，返回值为触发事件的mask

virtual EventTriggerId createEventTrigger(TaskFunc* eventHandlerProc);

//删除触发事件

virtual void deleteEventTrigger(EventTriggerId eventTriggerId);

//触发事件，将触发事件mask，加入待触发事件集合

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

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


```

189     unsigned i = fLastUsedTriggerNum;
190     EventTriggerId mask = fLastUsedTriggerMask;
191
192     do {
193         i = (i + 1) % MAX_NUM_EVENT_TRIGGERS;
194         mask >>= 1;
195         if (mask == 0) mask = 0x80000000;
196
197         if ((fTriggersAwaitingHandling & mask) != 0) {
198             fTriggersAwaitingHandling &= ~mask;
199             if (fTriggeredEventHandlers[i] != NULL) {
200                 (*fTriggeredEventHandlers[i])(fTriggeredEventClientDatas[i]);
201             }
202
203             fLastUsedTriggerMask = mask;
204             fLastUsedTriggerNum = i;
205             break;
206         }
207     } while (i != fLastUsedTriggerNum);

```

补充知识：

- 1、决不要重新定义继承而来的缺省参数值；
- 2、虚函数是动态绑定而缺省参数值是静态绑定的，也就是指针是哪种类型，就调用该类型对应的类中，该函数定义时的缺省值。

```

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

```

```

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

```

```

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

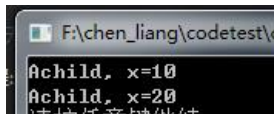
    int y = 20;
    Achlid* c = new Achlid;
    c->print(y);

    return 0;
}

```

```
}
```

执行结果:



```
F:\chen_liang\codetest\c
Achild, x=10
Achild, x=20
请按任意键继续
```

volatile关键字:

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

- 1). 并行设备的硬件寄存器 (如: 状态寄存器)
- 2). 一个中断服务子程序中会访问到的非自动变量(Non-automatic variables)
- 3). 多线程应用中被几个任务共享的变量

事件调度:

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

```
76 void BasicTaskScheduler0::doEventLoop(char volatile* watchVariable) {
77     // Repeatedly loop, handling readable sockets and timed events:
78     while (1) {
79         if (watchVariable != NULL && *watchVariable != 0) break;
80         SingleStep();
81     }
82 }
```

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

- 1)、为所有需要操作的socket执行select;
- 2)、找出第一个应执行的socket任务(handler)并执行之;
- 3)、找到第一个应响应的触发事件, 并执行之;
- 4)、找到第一个应执行的延迟任务并执行之;

可见, 每一步中只执行三个任务队列中的一项。

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

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

    // Also check our "maxDelayTime" parameter (if it's > 0):
    if (maxDelayTime > 0 &&
        (tv_timeToDelay.tv_sec > (long)maxDelayTime/MILLION ||
         (tv_timeToDelay.tv_sec == (long)maxDelayTime/MILLION &&
          tv_timeToDelay.tv_usec > (long)maxDelayTime%MILLION))) {
        tv_timeToDelay.tv_sec = maxDelayTime/MILLION;
        tv_timeToDelay.tv_usec = maxDelayTime%MILLION;
    }
}
```

参数maxDelayTime默认值为10ms;

//获取延迟事件队列中, 最先将超时的事件的剩余时间

DelayInterval const& timeToDelay = fDelayQueue.timeToNextAlarm();

//保证select的等待时间不超过10ms，精度10ms

```
if(maxDelayTime > 0 &&
    (tv_timeToDelay.tv_sec > (long)maxDelayTime/MILLION ||
    (tv_timeToDelay.tv_sec == (long)maxDelayTime/MILLION &&
    tv_timeToDelay.tv_usec > (long)maxDelayTime%MILLION))) {
    tv_timeToDelay.tv_sec = maxDelayTime/MILLION;
    tv_timeToDelay.tv_usec = maxDelayTime%MILLION;
}
```

//对三组描述符进行io复用

```
select(fMaxNumSockets, &readSet, &writeSet, &exceptionSet, &tv_timeToDelay);
```

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

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

// IO事件
// Call the handler function for one readable socket:
HandlerIterator iter(*fHandlers); //fHandlers为双向链表，新节点在表头
HandlerDescriptor* handler;
// To ensure forward progress through the handlers, begin past the last
// socket number that we handled:
// IO事件存储在双向链表中，新事件总是存储在表头；
// 由于在SingleStep中，IO事件只执行一个，为了保证所有的IO事件都能执行：
// 从上次IO事件的下一个事件开始遍历
if (fLastHandledSocketNum >= 0) {
    while ((handler = iter.next()) != NULL) {
        if (handler->socketNum == fLastHandledSocketNum) break;
    }
    if (handler == NULL) {
        fLastHandledSocketNum = -1;
        iter.reset(); // start from the beginning instead
    }
}
```



```

// 遍历IO事件链表，找到第一个可读|可写|可执行的描述符
// 执行IO事件的回调函数，记录该事件的描述符
while ((handler = iter.next()) != NULL) {
    int sock = handler->socketNum; // alias
    int resultConditionSet = 0;
    if (FD_ISSET(sock, &readSet) && FD_ISSET(sock, &fReadSet)/*sanity check*/)
        resultConditionSet |= SOCKET_READABLE;
    if (FD_ISSET(sock, &writeSet) && FD_ISSET(sock, &fWriteSet)/*sanity check*/)
        resultConditionSet |= SOCKET_WRITABLE;
    if (FD_ISSET(sock, &exceptionSet) && FD_ISSET(sock, &fExceptionSet)/*sanity check*/)
        resultConditionSet |= SOCKET_EXCEPTION;
    if ((resultConditionSet & handler->conditionSet) != 0 && handler->handlerProc != NULL) {
        fLastHandledSocketNum = sock;
        // Note: we set "fLastHandledSocketNum" before calling the handler,
        // in case the handler calls "doEventLoop()" reentrantly.
        (*handler->handlerProc)(handler->clientData, resultConditionSet);
        break;
    }
}
// 未找到描述符变化的IO事件
// 由于fLastHandledSocketNum大于等于零，上文查找不是从链表头开始的
// 需要重新从链表头开始重新查找
if (handler == NULL && fLastHandledSocketNum >= 0) {
    // We didn't call a handler, but we didn't get to check all of them,
    // so try again from the beginning:
    iter.reset();
    while ((handler = iter.next()) != NULL) {
        int sock = handler->socketNum; // alias
        int resultConditionSet = 0;
        if (FD_ISSET(sock, &readSet) && FD_ISSET(sock, &fReadSet)/*sanity check*/)
            resultConditionSet |= SOCKET_READABLE;
        if (FD_ISSET(sock, &writeSet) && FD_ISSET(sock, &fWriteSet)/*sanity check*/)
            resultConditionSet |= SOCKET_WRITABLE;
        if (FD_ISSET(sock, &exceptionSet) && FD_ISSET(sock, &fExceptionSet)/*sanity check*/)
            resultConditionSet |= SOCKET_EXCEPTION;
        if ((resultConditionSet & handler->conditionSet) != 0 && handler->handlerProc != NULL) {
            fLastHandledSocketNum = sock;
            // Note: we set "fLastHandledSocketNum" before calling the handler,
            // in case the handler calls "doEventLoop()" reentrantly.
            (*handler->handlerProc)(handler->clientData, resultConditionSet);
            break;
        }
    }
}
if (handler == NULL) fLastHandledSocketNum = -1; // because we didn't call a handler
}

```

IO事件是永久性事件，会重复执行；

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

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


```

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

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

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

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

```

触发器事件，为一次性事件，只执行一次；

创建的触发器，再有加入触发器集合中才会被触发；

触发器集合中的事件，按照创建的顺序进行触发，即优先触发最先创建的事件

```

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

```

延迟事件：为一次性事件，只执行一次；

若延时事件队列中存在多个超时事件，一次只触发一个事件；

触发最先超时的事件，即触发表头事件；

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