live555 消息循环

- live555MediaServer.cpp中的main()函数

```
创建了一系列的对象之后进入env->taskScheduler().doEventLoop(); // does not return
```

- doEventLoop

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

- SingleStep()

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事件，并执行之。
4. 找到第一个应执行的延迟任务并执行之。

```

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

  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;
  }

  int selectResult = select(fMaxNumSockets, &readSet, &writeSet, &exceptionSet, &tv_timeT
  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:
      if (fDummySocketNum >= 0) closeSocket(fDummySocketNum);
      fDummySocketNum = socket(AF_INET, SOCK_DGRAM, 0);
      FD_SET((unsigned)fDummySocketNum, &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");
  // Because this failure is often "Bad file descriptor" - which is caused by an invalid
  // that had already been closed) being used in "select()" - we print out the sockets th
  // to assist in debugging:
  fprintf(stderr, "socket numbers used in the select() call:");
  for (int i = 0; i < 10000; ++i) {
    if (FD_ISSET(i, &fReadSet) || FD_ISSET(i, &fWriteSet) || FD_ISSET(i, &fExceptionSet))
      fprintf(stderr, " %d(", i);
      if (FD_ISSET(i, &fReadSet)) fprintf(stderr, "r");
      if (FD_ISSET(i, &fWriteSet)) fprintf(stderr, "w");
      if (FD_ISSET(i, &fExceptionSet)) fprintf(stderr, "e");
      fprintf(stderr, ")");
    }
  }
```

```
  fprintf(stderr, "\n");
#endif
  internalError();
      }
  }

  // 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:
  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
    }
  }
  while ((handler = iter.next()) != NULL) {
    int sock = handler->socketNum; // alias
    int resultConditionSet = 0;
    if (FD_ISSET(sock, &readSet) && FD_ISSET(sock, &fReadSet)/*sanity check*/) resultCon
    if (FD_ISSET(sock, &writeSet) && FD_ISSET(sock, &fWriteSet)/*sanity check*/) resultC
    if (FD_ISSET(sock, &exceptionSet) && FD_ISSET(sock, &fExceptionSet)/*sanity check*/)
    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 >= 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
```

```
99        int resultConditionSet = 0;
100       if (FD_ISSET(sock, &readSet) && FD_ISSET(sock, &fReadSet)/*sanity check*/) resultC
101       if (FD_ISSET(sock, &writeSet) && FD_ISSET(sock, &fWriteSet)/*sanity check*/) resul
102       if (FD_ISSET(sock, &exceptionSet) && FD_ISSET(sock, &fExceptionSet)/*sanity check*
103       if ((resultConditionSet&handler->conditionSet) != 0 && handler->handlerProc != NUL
104   fLastHandledSocketNum = sock;
105       // Note: we set "fLastHandledSocketNum" before calling the handler,
106             // in case the handler calls "doEventLoop()" reentrantly.
107   (*handler->handlerProc)(handler->clientData, resultConditionSet);
108   break;
109       }
110     }
111     if (handler == NULL) fLastHandledSocketNum = -1;//because we didn't call a handler
112   }
113
114   // Also handle any newly-triggered event (Note that we do this *after* calling a socket
115   // in case the triggered event handler modifies The set of readable sockets.)
116   if (fTriggersAwaitingHandling != 0) {
117     if (fTriggersAwaitingHandling == fLastUsedTriggerMask) {
118       // Common-case optimization for a single event trigger:
119       fTriggersAwaitingHandling &=~ fLastUsedTriggerMask;
120       if (fTriggeredEventHandlers[fLastUsedTriggerNum] != NULL) {
121   (*fTriggeredEventHandlers[fLastUsedTriggerNum])(fTriggeredEventClientDatas[fLastUsedTr
122       }
123     } else {
124       // Look for an event trigger that needs handling (making sure that we make forward
125       unsigned i = fLastUsedTriggerNum;
126       EventTriggerId mask = fLastUsedTriggerMask;
127
128       do {
129   i = (i+1)%MAX_NUM_EVENT_TRIGGERS;
130   mask >>= 1;
131   if (mask == 0) mask = 0x80000000;
132
133   if ((fTriggersAwaitingHandling&mask) != 0) {
134     fTriggersAwaitingHandling &=~ mask;
135     if (fTriggeredEventHandlers[i] != NULL) {
136       (*fTriggeredEventHandlers[i])(fTriggeredEventClientDatas[i]);
137     }
138
```

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

  // Also handle any delayed event that may have come due.
  fDelayQueue.handleAlarm();
}
```