ZLToolKit笔记7：TCP通信

2024年12月24日 21:13

通过多线程监听内核监听队列中的监听套接字（Listen fd）+抢占式accept完成客户端连接请求的响应并建立TCP连接。

接收连接的子线程（子服务器）负责监听连接套接字上的读写/异常错误事件，事件触发后将回调相应的处理逻辑。

将监听任务加入epoll监听表的工作由attachEvent这个函数完成，何时调用该函数在笔记5最后也已经说明。

```
bool Socket::attachEvent(const SockNum::Ptr &sock) {
    weak_ptr<Socket> weak_self = shared_from_this();
    if (sock->type() == SockNum::Sock_TCP_Server) {
        // tcp服务器 [AUTO-TRANSLATED:f4b9757f]
        //TCP server
        auto result = _poller->addEvent(sock->rawFd(), EventPoller::Event_Read | EventPoller::Event_Error, [weak_self, sock](int event) {
            if (auto strong_self = weak_self.lock()) {
                strong_self->onAccept(sock, event);
            }
        });
        return -1 != result;
    }
    // tcp客户端或udp [AUTO-TRANSLATED:00c16e7f]
    //TCP client or UDP
    auto read_buffer = _poller->getSharedBuffer(sock->type() == SockNum::Sock_UDP);
    auto result = _poller->addEvent(sock->rawFd(), EventPoller::Event_Read | EventPoller::Event_Error | EventPoller::Event_Write,
[weak_self, sock, read_buffer](int event) {
        auto strong_self = weak_self.lock();
        if (!strong_self) {
            return;
        }
        if (event & EventPoller::Event_Read) {
            strong_self->onRead(sock, read_buffer);
        }
        if (event & EventPoller::Event_Write) {
            strong_self->onWriteAble(sock);
        }
        if (event & EventPoller::Event_Error) {
            if (sock->type() == SockNum::Sock_UDP) {
                // udp ignore error
            } else {
                strong_self->emitErr(getSockErr(sock->rawFd()));
            }
        }
    });
    return -1 != result;
}
```

异步监听sock->rawFd()这个socket（也就是连接套接字）上的EventPoller::Event_Read | EventPoller::Event_Error | EventPoller::Event_Write事件，如果发生将在poller线程中调用上面传入的第三个参数，也就是相应的事件处理的匿名函数。
下面分别就三个if三种事件触发回调做分析：

1. 可读

在attachEvent函数中，在将连接套接字的监听任务加入epoll表中前，通过：

```
auto read_buffer = _poller->getSharedBuffer(sock->type() == SockNum::Sock_UDP);
```

这个函数getSharedBuffer()获取当前poller线程下所有socket所共享的读缓存。

EventPoller类中的私有成员变量_shared_buffer[2]是读缓存的弱引用：

```
    //当前线程下，所有socket共享的读缓存 [AUTO-TRANSLATED:6ce70017]
    //当前线程下，所有socket共享的读缓存
// Shared read buffer for all sockets under the current thread
    std::weak_ptr<SocketRecvBuffer> _shared_buffer[2];
```

通过getSharedBuffer函数获取了管理SocketRecvBuffer对象的强引用（share_ptr）。

```
SocketRecvBuffer::Ptr EventPoller::getSharedBuffer(bool is_udp) {
#if !defined(__linux) && !defined(__linux__)
    // 非Linux平台下，tcp和udp共享recvfrom方案，使用同一个buffer [AUTO-TRANSLATED:2d2ee7bf]
    //On non-Linux platforms, tcp and udp share the recvfrom scheme, using the same buffer
```

```
    is_udp = 0;
#endif
    auto ret = _shared_buffer[is_udp].lock();
    if (!ret) {
        ret = SocketRecvBuffer::create(is_udp);
        _shared_buffer[is_udp] = ret;
    }
    return ret;
}
```

如果尚未初始化该poller线程的读缓存那么调用SocketRecvBuffer类中的静态成员函数：create() ： 定义如下：

他会返回一个管理者SocketRecvFromBuffer资源的智能指针。

```
static constexpr auto kPacketCount = 32;
static constexpr auto kBufferCapacity = 4 * 1024u;
SocketRecvBuffer::Ptr SocketRecvBuffer::create(bool is_udp) {
#if defined(__linux) || defined(__linux__)
    if (is_udp) {
        return std::make_shared<SocketRecvmmsgBuffer>(kPacketCount, kBufferCapacity);
    }
#endif
    return std::make_shared<SocketRecvFromBuffer>(kPacketCount * kBufferCapacity);
}
```

其余buffer的内容参见笔记8。

```
ssize_t Socket::onRead(const SockNum::Ptr &sock, const SocketRecvBuffer::Ptr &buffer) noexcept {
    ssize_t ret = 0, nread = 0, count = 0;
    while (_enable_recv) { //如果允许接收数据
        nread = buffer->recvFromSocket(sock->rawFd(), count);  //从套接字接收数据
        if (nread == 0) { //接收到EOF（末尾）
            if (sock->type() == SockNum::Sock_TCP) {
                emitErr(SockException(Err_eof, "end of file"));
            } else {
                WarnL << "Recv eof on udp socket[" << sock->rawFd() << "]";
            }
            return ret;
        }
        if (nread == -1) { //如果接收发生错误
            auto err = get_uv_error(true);  //获取错误类型
            if (err != UV_EAGAIN) {
                if (sock->type() == SockNum::Sock_TCP) {
                    emitErr(toSockException(err));
                } else {
                    WarnL << "Recv err on udp socket[" << sock->rawFd() << "]: " << uv_strerror(err);
                }
            }
            return ret;
        }
        ret += nread;  //累加接收到的字节数
        if (_enable_speed) {
            // 更新接收速率 [AUTO-TRANSLATED:1e24774c]
            //Update receive rate
            _recv_speed += nread;
        }
          // 获取接收到的数据缓冲区和地址
        auto &buf = buffer->getBuffer(0);
        auto &addr = buffer->getAddress(0);
        try {
            // 此处捕获异常，目的是防止数据未读尽，epoll边沿触发失效的问题 [AUTO-TRANSLATED:2f3f813b]
            //Catch exception here, the purpose is to prevent data from not being read completely, and the epoll edge trigger fails
            LOCK_GUARD(_mtx_event);  //枷锁防止多线程问题
            _on_multi_read(&buf, &addr, count);  //调用多读回调
        } catch (std::exception &ex) { //捕获异常
            ErrorL << "Exception occurred when emit on_read: " << ex.what();
        }
    }
    return 0;
}
```

关键点：

- **循环读取：**

  通过 while (_enable_recv) 循环调用 recvFromSocket，以确保数据不会在边缘触发模式下丢失。

问题：为什么数据可能会在边缘触发模式下丢失？？？

- **错误处理：**
  当 recvFromSocket 返回 -1 时，判断错误类型。如果是非 EAGAIN 错误，会触发相应的回调。
- **多线程安全：**
  使用 LOCK_GUARD 加锁保护 _on_multi_read 的调用，确保线程安全。
- **接收速率更新：**
  如果启用了速率统计功能，接收到的字节数会累加到 _recv_speed 中。
- **回调调用：**
  调用 _on_multi_read 处理接收到的数据。

accept连接后执行的回调过程中，会调用server->**onAcceptConnection**(sock)的过程，详细可见笔记5中。
在这个函数中，通过sock->setOnRead(task)方法设置了_on_multi_read的定义：

```
sock->setOnRead([weak_session](const Buffer::Ptr &buf, struct sockaddr *, int) {
        //获取会话强应用 [AUTO-TRANSLATED:187497e6]
        //Get the strong application of the session
        auto strong_session = weak_session.lock();
        if (!strong_session) {
            return;
        }
        try {
            strong_session->onRecv(buf);
        } catch (SockException &ex) {
            strong_session->shutdown(ex);
        } catch (exception &ex) {
            strong_session->shutdown(SockException(Err_shutdown, ex.what()));
        }
    });
```

其中setOnRead的定义如下：

```
void Socket::setOnRead(onReadCB cb) {
    onMultiReadCB cb2;
    if (cb) {
        cb2 = [cb](Buffer::Ptr *buf, struct sockaddr_storage *addr, size_t count) {
            for (auto i = 0u; i < count; ++i) {
                cb(buf[i], (struct sockaddr *)(addr + i), sizeof(struct sockaddr_storage));
            }
        };
    }
    setOnMultiRead(std::move(cb2));
}
void Socket::setOnMultiRead(onMultiReadCB cb) {
    LOCK_GUARD(_mtx_event);
    if (cb) {
        _on_multi_read = std::move(cb);
    } else {
        _on_multi_read = [](Buffer::Ptr *buf, struct sockaddr_storage *addr, size_t count) {
            for (auto i = 0u; i < count; ++i) {
                WarnL << "Socket not set read callback, data ignored: " << buf[i]->size();
            }
        };
    }
}
```

将数据读取到缓冲区后，会触发相应session中的回调onRecv函数，这个就是服务器端的处理逻辑。

2. 可写

如果socket的可写事件触发，那么执行onWriteAble回调

```
void Socket::onWriteAble(const SockNum::Ptr &sock) {
    bool empty_waiting;
    bool empty_sending;
     //(1)检查一级发送缓存，当socket可写时，会把一级缓存批量送入二级缓存（加锁访问）
    {
        LOCK_GUARD(_mtx_send_buf_waiting);
        empty_waiting = _send_buf_waiting.empty();
    }
     //(2)检查二级发送缓存，当socket可写时，会把二级缓存批量写入socket中（加锁访问）
    {
        LOCK_GUARD(_mtx_send_buf_sending);
        empty_sending = _send_buf_sending.empty();
    }
```

```
    //(3)根据缓存是否为空的结果，决定是否发送数据还是停止监听可写事件
    if (empty_waiting && empty_sending) {
        // 数据已经清空了，我们停止监听可写事件 [AUTO-TRANSLATED:751f7e4e]
        //Data has been cleared, we stop listening for writable events
        stopWriteAbleEvent(sock);
    } else {
        // socket可写，我们尝试发送剩余的数据 [AUTO-TRANSLATED:d66e0207]
        //Socket is writable, we try to send the remaining data
        flushData(sock, true);
    }
}
```

我们先不管这两级的写缓存是如何设计实现的。

停止监听socket上的可写事件调用的是stopWriteAbleEvent函数：

```
void Socket::stopWriteAbleEvent(const SockNum::Ptr &sock) {
    // 停止监听socket可写事件 [AUTO-TRANSLATED:4eb5b241]
    //Stop listening for socket writable events
    _sendable = true;
    int flag = _enable_recv ? EventPoller::Event_Read : 0;
    _poller->modifyEvent(sock->rawFd(), flag | EventPoller::Event_Error, [sock](bool) {});
}
```

_sendable用来标记该socket是否可写，当缓存已满后便不可再写（为false）。

_enable_recv用来标记是否监听socket可读事件，关闭后可用于流量控制。

这两个变量的定义如下：

```
    // 控制是否接收监听socket可读事件，关闭后可用于流量控制 [AUTO-TRANSLATED:71de6ece]
    //Control whether to receive listen socket readable events, can be used for traffic control after closing
    std::atomic<bool> _enable_recv { true };
    // 标记该socket是否可写，socket写缓存满了就不可写 [AUTO-TRANSLATED:32392de2]
    //Mark whether the socket is writable, the socket write buffer is full and cannot be written
    std::atomic<bool> _sendable { true };
```

停止监听socket上的可写事件，无非就是更新对应的内核事件表，去除相应socket上的可写监听即可，这个工作是由modifyEvent函数完成的：

```
    _poller->modifyEvent(sock->rawFd(), flag | EventPoller::Event_Error, [sock](bool) {});
```

定义如下：epoll机制下，调用epoll_ctl来修改_event_fd标记的那个内核事件表。

```
int EventPoller::modifyEvent(int fd, int event, PollCompleteCB cb) {
    TimeTicker();
    if (!cb) {
        cb = [](bool success) {};
    }
    if (isCurrentThread()) {
#if defined(HAS_EPOLL)
        struct epoll_event ev = { 0 };
        ev.events = toEpoll(event);
        ev.data.fd = fd;
        auto ret = epoll_ctl(_event_fd, EPOLL_CTL_MOD, fd, &ev);
        cb(ret != -1);
        return ret;
#elif defined(HAS_KQUEUE)
        struct kevent kev[2];
        int index = 0;
        EV_SET(&kev[index++], fd, EVFILT_READ, event & Event_Read ? EV_ADD | EV_CLEAR : EV_DELETE, 0, 0, nullptr);
        EV_SET(&kev[index++], fd, EVFILT_WRITE, event & Event_Write ? EV_ADD | EV_CLEAR : EV_DELETE, 0, 0, nullptr);
        int ret = kevent(_event_fd, kev, index, nullptr, 0, nullptr);
        cb(ret != -1);
        return ret;
#else
        auto it = _event_map.find(fd);
        if (it != _event_map.end()) {
            it->second->event = event;
        }
        cb(it != _event_map.end());
        return it != _event_map.end() ? 0 : -1;
#endif // HAS_EPOLL
    }
    async([this, fd, event, cb]() mutable {
        modifyEvent(fd, event, std::move(cb));
    });
    return 0;
}
```

检查完两级缓存发现仍有数据存在其中，则调用flushData函数尝试发送剩余的数据：

```
bool Socket::flushData(const SockNum::Ptr &sock, bool poller_thread) {
    decltype(_send_buf_sending) send_buf_sending_tmp;
    {
        // 转移出二级缓存 [AUTO-TRANSLATED:a54264d2]
        //Transfer out of the secondary cache
        LOCK_GUARD(_mtx_send_buf_sending);
        if (!_send_buf_sending.empty()) {
            send_buf_sending_tmp.swap(_send_buf_sending);
        }
    }
    if (send_buf_sending_tmp.empty()) {
        _send_flush_ticker.resetTime();
        do {
            {
                // 二级发送缓存为空，那么我们接着消费一级缓存中的数据 [AUTO-TRANSLATED:8ddb2962]
                //The secondary send cache is empty, so we continue to consume data from the primary cache
                LOCK_GUARD(_mtx_send_buf_waiting);
                if (!_send_buf_waiting.empty()) {
                    // 把一级缓中数数据放置到二级缓存中并清空 [AUTO-TRANSLATED:4884aa58]
                    //Put the data from the first-level cache into the second-level cache and clear it
                    LOCK_GUARD(_mtx_event);
                    auto send_result = _enable_speed ? [this](const Buffer::Ptr &buffer, bool send_success) {
                        if (send_success) {
                            //更新发送速率 [AUTO-TRANSLATED:e35a1eba]
                            //Update the sending rate
                            _send_speed += buffer->size();
                        }
                        LOCK_GUARD(_mtx_event);
                        if (_send_result) {
                            _send_result(buffer, send_success);
                        }
                    } : _send_result;
                    send_buf_sending_tmp.emplace_back(BufferList::create(std::move(_send_buf_waiting), std::move(send_result), sock->type()
== SockNum::Sock_UDP));
                    break;
                }
            }
            // 如果一级缓存也为空,那么说明所有数据均写入socket了 [AUTO-TRANSLATED:6ae9ef8a]
            //If the first-level cache is also empty, it means that all data has been written to the socket
            if (poller_thread) {
                // poller线程触发该函数，那么该socket应该已经加入了可写事件的监听； [AUTO-TRANSLATED:5a8e123d]
                //The poller thread triggers this function, so the socket should have been added to the writable event listening
                // 那么在数据列队清空的情况下，我们需要关闭监听以免触发无意义的事件回调 [AUTO-TRANSLATED:0fb35573]
                //So, in the case of data queue clearing, we need to close the listening to avoid triggering meaningless event callbacks
                stopWriteAbleEvent(sock);
                onFlushed();
            }
            return true;
        } while (false);
    }
    while (!send_buf_sending_tmp.empty()) {
        auto &packet = send_buf_sending_tmp.front();
        auto n = packet->send(sock->rawFd(), _sock_flags);
        if (n > 0) {
            // 全部或部分发送成功 [AUTO-TRANSLATED:0721ed7c]
            //All or part of the data was sent successfully
            if (packet->empty()) {
                // 全部发送成功 [AUTO-TRANSLATED:38a7d0ac]
                //All data was sent successfully
                send_buf_sending_tmp.pop_front();
                continue;
            }
            // 部分发送成功 [AUTO-TRANSLATED:bd6609dd]
            //Part of the data was sent successfully
            if (!poller_thread) {
                // 如果该函数是poller线程触发的，那么该socket应该已经加入了可写事件的监听，所以我们不需要再次加入监听 [AUTO-
TRANSLATED:917049f0]
                //If this function is triggered by the poller thread, the socket should have been added to the writable event listening, so
we don't need to add listening again
                startWriteAbleEvent(sock);
            }
            break;
```

```
        }
        // 一个都没发送成功 [AUTO-TRANSLATED:a3b4f257]
        //None of the data was sent successfully
        int err = get_uv_error(true);
        if (err == UV_EAGAIN) {
            // 等待下一次发送 [AUTO-TRANSLATED:22980496]
            //Wait for the next send
            if (!poller_thread) {
                // 如果该函数是poller线程触发的，那么该socket应该已经加入了可写事件的监听，所以我们不需要再次加入监听 [AUTO-TRANSLATED:917049f0]
                //If this function is triggered by the poller thread, the socket should have already been added to the writable event listener, so we don't need to add it again
                startWriteAbleEvent(sock);
            }
            break;
        }
        // 其他错误代码，发生异常 [AUTO-TRANSLATED:14cca084]
        //Other error codes, an exception occurred
        if (sock->type() == SockNum::Sock_UDP) {
            // udp发送异常，把数据丢弃 [AUTO-TRANSLATED:3a7d095d]
            //UDP send exception, discard the data
            send_buf_sending_tmp.pop_front();
            WarnL << "Send udp socket[" << sock << "] failed, data ignored: " << uv_strerror(err);
            continue;
        }
        // tcp发送失败时，触发异常 [AUTO-TRANSLATED:06f06449]
        //TCP send failed, trigger an exception
        emitErr(toSockException(err));
        return false;
    }
    // 回滚未发送完毕的数据 [AUTO-TRANSLATED:9f67c1be]
    //Roll back the unsent data
    if (!send_buf_sending_tmp.empty()) {
        // 有剩余数据 [AUTO-TRANSLATED:14a89b15]
        //There is remaining data
        LOCK_GUARD(_mtx_send_buf_sending);
        send_buf_sending_tmp.swap(_send_buf_sending);
        _send_buf_sending.append(send_buf_sending_tmp);
        // 二级缓存未全部发送完毕，说明该socket不可写，直接返回 [AUTO-TRANSLATED:2d7f9f2f]
        //The secondary cache has not been sent completely, indicating that the socket is not writable, return directly
        return true;
    }
    // 二级缓存已经全部发送完毕，说明该socket还可写，我们尝试继续写 [AUTO-TRANSLATED:2c2bc316]
    //The secondary cache has been sent completely, indicating that the socket is still writable, we try to continue writing
    // 如果是poller线程，我们尝试再次写一次(因为可能其他线程调用了send函数又有新数据了) [AUTO-TRANSLATED:392684a8]
    //If it's the poller thread, we try to write again (because other threads may have called the send function and there is new data)
    return poller_thread ? flushData(sock, poller_thread) : true;
}
```

我们重点分析flushData这个函数逻辑：

(1) 二级缓存的预先处理：

如果二级写缓存非空，则调用swap函数将其内容交换到新定义的与二级缓存_send_buf_sending相同类型的变量send_buf_sending_tmp中。

```
if (!_send_buf_sending.empty()) {
    send_buf_sending_tmp.swap(_send_buf_sending);
}
```

目的：在不阻塞其他线程的情况下处理缓存数据。

(2) 二级缓存为空时的处理：

如果二级缓存为空，则尝试将一级缓存中的内容移动到二级缓存中，如若一级缓存也为空，说明所有数据已经写入到socket中了，在这种情况下，我们需要关闭可写事件的监听以触发无意义的事件回调。

a. 一级缓存非空

```
send_buf_sending_tmp.emplace_back(BufferList::create(std::move(_send_buf_waiting), std::move(send_result), sock->type() == SockNum::Sock_UDP));
```

注意这里调用的是list的移动构造函数，调用后原先的被移动的对象_send_buf_waiting将置空。

利用std::move和移动构造的方法，避免了冗余拷贝，提高了性能。

b. 一级缓存为空：

```
if (poller_thread) {
    // poller线程触发该函数，那么该socket应该已经加入了可写事件的监听； [AUTO-TRANSLATED:5a8e123d]
    //The poller thread triggers this function, so the socket should have been added to the writable event listening
    // 那么在数据列队清空的情况下，我们需要关闭监听以免触发无意义的事件回调 [AUTO-TRANSLATED:0fb35573]
    //So, in the case of data queue clearing, we need to close the listening to avoid triggering meaningless event
```

```
callbacks
                stopWriteAbleEvent(sock);
                onFlushed();
            }
            return true;
```

缓存如何设计实现的具体内容参见：ZLToolKit笔记8：Buffer设计

（3）发送数据的循环处理：

```
auto &packet = send_buf_sending_tmp.front();
auto n = packet->send(sock->rawFd(), _sock_flags);
```

已知send_buf_sending_tmp是由BufferList类型元素构成的vector对象。

紧接着调用了send成员函数，但BufferList是个虚基类，内部的send方法是一个纯虚函数，需在派生类中实现。BufferSendTo和BufferSendMsg均继承于BufferList，故其中实现了send方法。

但我们怎么知道调用的是哪一个派生类中的send呢？？？

其实，起初创建二级缓存时，是通过一级缓存的内容移动到二级缓存中（并未实现真正的移动）。二级缓存为空，且一级缓存非空时，通过（2）a的过程实现了二级缓存的构造：

```
send_buf_sending_tmp.emplace_back(BufferList::create(std::move(_send_buf_waiting), std::move(send_result), sock->type() == SockNum::Sock_UDP));
```

在create内部：根据socket的类型，决定构建的是哪一个派生类的对象，并返回管理该对象的智能指针。

```
BufferList::Ptr BufferList::create(List<std::pair<Buffer::Ptr, bool> > list, SendResult cb, bool is_udp) {
#if defined(_WIN32)
    if (is_udp) {
        // sendto/send 方案，待优化 [AUTO-TRANSLATED:e94184aa]
        //sendto/send scheme, to be optimized
        return std::make_shared<BufferSendTo>(std::move(list), std::move(cb), is_udp);
    }
    // WSASend方案 [AUTO-TRANSLATED:9ac7bb81]
    //WSASend scheme
    return std::make_shared<BufferSendMsg>(std::move(list), std::move(cb));
#elif defined(__linux__) || defined(__linux)
    if (is_udp) {
        // sendmmsg方案 [AUTO-TRANSLATED:4596c2c4]
        //sendmmsg scheme
        return std::make_shared<BufferSendMMsg>(std::move(list), std::move(cb));
    }
    // sendmsg方案 [AUTO-TRANSLATED:8846f9c4]
    //sendmsg scheme
    return std::make_shared<BufferSendMsg>(std::move(list), std::move(cb));
#else
    if (is_udp) {
        // sendto/send 方案，可优化? [AUTO-TRANSLATED:21dbae7c]
        //sendto/send scheme, can be optimized?
        return std::make_shared<BufferSendTo>(std::move(list), std::move(cb), is_udp);
    }
    // sendmsg方案 [AUTO-TRANSLATED:8846f9c4]
    //sendmsg scheme
    return std::make_shared<BufferSendMsg>(std::move(list), std::move(cb));
#endif
}
```

简单总结：

要确定 flushData 函数中调用的 send 方法是哪一个，重点在于以下条件：

1. 查看 BufferList::create 的实现，明确其返回的具体派生类。
2. 检查 sock->type() 的值是否为 SockNum::Sock_UDP。

根据 sock->type() 的值：

- 如果是 UDP，则调用 BufferSendTo::send。
- 如果是其他类型（如 TCP），则调用 BufferSendMsg::send。

有关Buffer的内容见ZLToolKit笔记8：Buffer设计

如果packet指向的是一个派生类BufferSendMsg的对象:

auto n = packet->send(sock->rawFd(), _sock_flags);

那么调用的send定义如下：

```
ssize_t BufferSendMsg::send(int fd, int flags) {
    auto remain_size = _remain_size;
    while (_remain_size && send_l(fd, flags) != -1);
    ssize_t sent = remain_size - _remain_size;
    if (sent > 0) {
```

```
            //部分或全部发送成功 [AUTO-TRANSLATED:a3f5e70e]
            //Partial or all send success
            return sent;
        }
        //一个字节都未发送成功 [AUTO-TRANSLATED:858b63e5]
        //Not a single byte sent successfully
        return -1;
    }
```

其中send_l定义如下：

```
    ssize_t BufferSendMsg::send_l(int fd, int flags) {
        ssize_t n;
    #if !defined(_WIN32)
        do {
            struct msghdr msg;
            msg.msg_name = nullptr;
            msg.msg_namelen = 0;
            msg.msg_iov = &(_iovec[_iovec_off]);
            msg.msg_iovlen = _iovec.size() - _iovec_off;
            if (msg.msg_iovlen > IOV_MAX) {
                msg.msg_iovlen = IOV_MAX;
            }
            msg.msg_control = nullptr;
            msg.msg_controllen = 0;
            msg.msg_flags = flags;
            n = sendmsg(fd, &msg, flags);
        } while (-1 == n && UV_EINTR == get_uv_error(true));
    #else
        do {
            DWORD sent = 0;
            n = WSASend(fd, const_cast<LPWSABUF>(&_iovec[0]), static_cast<DWORD>(_iovec.size()), &sent, static_cast<DWORD>(flags), 0, 0);
            if (n == SOCKET_ERROR) {return -1;}
            n = sent;
        } while (n < 0 && UV_ECANCELED == get_uv_error(true));
    #endif

        if (n >= (ssize_t)_remain_size) {
            //全部写完了 [AUTO-TRANSLATED:c990f48a]
            //All written
            _remain_size = 0;
            sendCompleted(true);
            return n;
        }
        if (n > 0) {
            //部分发送成功 [AUTO-TRANSLATED:4c240905]
            //Partial send success
            reOffset(n);
            return n;
        }
        //一个字节都未发送 [AUTO-TRANSLATED:c33c611b]
        //Not a single byte sent
        return n;
    }
```

send_l的过程实现了从二级缓存向socket上发送数据的具体过程，调用了sendmsg(linux)或WSASend(Windows)实现了一次多块数据发送。

那么问题来了：由上可知二级缓存中的内容是由一级缓存移动构造而来的，那么**一级缓存的内容又是从何而来的呢？？？**

其实是由socket :: send函数实现的：

```
ssize_t Socket::send(const char *buf, size_t size, struct sockaddr *addr, socklen_t addr_len, bool try_flush) {
    if (size <= 0) {
        size = strlen(buf);
        if (!size) {
            return 0;
        }
    }
    auto ptr = BufferRaw::create();
    ptr->assign(buf, size);
    return send(std::move(ptr), addr, addr_len, try_flush);
}
ssize_t Socket::send(string buf, struct sockaddr *addr, socklen_t addr_len, bool try_flush) {
    return send(std::make_shared<BufferString>(std::move(buf)), addr, addr_len, try_flush);
}
ssize_t Socket::send(Buffer::Ptr buf, struct sockaddr *addr, socklen_t addr_len, bool try_flush) {
    if (!addr) {
        if (!_udp_send_dst) {
```

```
            return send_l(std::move(buf), false, try_flush);
        }
        // 本次发送未指定目标地址，但是目标定制已通过bindPeerAddr指定 [AUTO-TRANSLATED:afb6ce35]
        //This send did not specify a target address, but the target is customized through bindPeerAddr
        addr = (struct sockaddr *)_udp_send_dst.get();
        addr_len = SockUtil::get_sock_len(addr);
    }
    return send_l(std::make_shared<BufferSock>(std::move(buf), addr, addr_len), true, try_flush);
}
ssize_t Socket::send_l(Buffer::Ptr buf, bool is_buf_sock, bool try_flush) {
    auto size = buf ? buf->size() : 0;
    if (!size) {
        return 0;
    }
    {
        LOCK_GUARD(_mtx_send_buf_waiting);
        _send_buf_waiting.emplace_back(std::move(buf), is_buf_sock);
    }
    if (try_flush) {
        if (flushAll()) {
            return -1;
        }
    }
    return size;
}
```

最终会调用Socket::send_l方法实现往一级缓存中写入数据。

来看第一个send函数中有：

```
auto ptr = BufferRaw::create();
ptr->assign(buf, size);
```

显然这与读缓存的创建方式类似，不同处在于create之后，此处并没有调用setCapacity方法设定读缓存空间大小，而是调用assign方法进行赋值数据：即将参数buf指代的数据拷贝到刚create后的buffer中（实际上是BufferRaw类中的私有成员变量_data中）有关Buffer的内容详见ZLToolKit笔记8：Buffer设计

以下内容是由chatgpt生成的：

1. 函数逐个分析

(1) Socket::send(const char *buf, size_t size, struct sockaddr *addr, socklen_t addr_len, bool try_flush)

- **功能**：
  - 发送一个 C 风格的字符数组（const char*），可以指定数据大小（size）或通过 strlen 自动确定。
  - 数据会被封装为一个 BufferRaw 对象（裸数据缓冲区），然后通过另一个 send 函数进行处理。
- **关键点**：
  - 如果 size <= 0，会使用 strlen(buf) 来计算数据大小。
  - 数据封装为 BufferRaw，便于后续的统一处理（封装是一种设计模式，便于后续灵活操作和多态支持）。
  - 使用 std::move(ptr) 将智能指针转移到下一个 send 方法中，避免重复拷贝。

(2) Socket::send(string buf, struct sockaddr *addr, socklen_t addr_len, bool try_flush)

- **功能**：
  - 发送一个 std::string 类型的数据。
  - 数据被封装为一个 BufferString 对象（表示字符串缓冲区）。
- **关键点**：
  - 使用 std::make_shared<BufferString>(std::move(buf)) 将字符串封装为缓冲区对象。
  - 通过封装，隐藏底层数据结构细节，使得发送逻辑统一。

(3) Socket::send(Buffer::Ptr buf, struct sockaddr *addr, socklen_t addr_len, bool try_flush)

- **功能**：
  - 接收一个 Buffer 对象（Buffer::Ptr 是 std::shared_ptr<Buffer>）。
  - 处理不同的发送场景：
    1. 如果没有指定目标地址（addr），尝试使用 _udp_send_dst（通过 bindPeerAddr 绑定的目标地址）。
    2. 如果指定了目标地址，将数据封装为 BufferSock 对象。
  - 最终调用 send_l 完成发送。
- **关键点**：
  - Buffer 是发送数据的抽象表示，可以支持多种类型的数据缓冲区（裸数据、字符串、网络缓冲区等）。
  - _udp_send_dst 是一个可选目标地址，可以通过 bindPeerAddr 设置，避免重复指定目标地址。

(4) Socket::send_l(Buffer::Ptr buf, bool is_buf_sock, bool try_flush)

- **功能**:
  - 负责将数据缓冲区加入发送队列，并尝试立即发送。
  - 如果 `try_flush` 为真，调用 `flushAll` 尝试发送数据。
- **关键点**:
  - 使用 `_mtx_send_buf_waiting` 互斥锁保护 `_send_buf_waiting` 队列（线程安全）。
  - 数据分为两种类型:
    1. 普通缓冲区（`is_buf_sock = false`）。
    2. 包含目标地址的缓冲区（`is_buf_sock = true`）。
  - 如果 `try_flush` 为真，则尝试立即刷新队列。

(5) `Socket::flushAll()`

- **功能**:
  - 尝试发送所有缓冲区中的数据。
  - 检查套接字状态和发送超时，处理不可写套接字的情况。
- **关键点**:
  - 如果 `_sock_fd` 为 `null`，说明连接已经断开或超时。
  - 如果套接字可写（`_sendable`），调用 `flushData` 实际发送数据。
  - 检测发送超时:
    - 如果发送队列中的最老数据超过 `_max_send_buffer_ms` 限制，触发超时错误并断开连接。

2. 关键概念和设计亮点

(1) 数据封装与多态设计

- 使用 `Buffer` 抽象类以及其派生类（如 `BufferRaw`、`BufferString`、`BufferSock`），统一了数据的表示和操作方式，便于扩展和维护。
- 每种数据类型（裸数据、字符串、带地址数据）都有对应的封装方式，这种分层设计提升了代码的可扩展性。

(2) 发送逻辑的分层

- `send` 方法逐层调用，最终由 `send_l` 执行核心逻辑，这种分层设计简化了代码的复杂性。
- 不同场景的数据发送逻辑（如指定地址、不指定地址）被封装到不同的层中，提高了代码复用性。

(3) 异步发送与线程安全

- `_send_buf_waiting` 是一个发送缓冲队列，使用互斥锁保护，确保线程安全。
- 支持异步发送，数据先进入队列，然后通过 `flushAll` 方法统一发送。

(4) 发送超时处理

- 使用 `_send_flush_ticker.elapsedTime()` 检测发送超时。
- 超时后通过 `emitErr` 报告错误并断开连接，避免长时间卡住。

3. 使用场景和扩展性

- **使用场景**:
  - 支持多种类型的数据发送（裸数据、字符串、带地址数据）。
  - 适用于需要异步、分段发送的网络通信场景。
- **扩展性**:
  - 新增数据类型时，只需新增 `Buffer` 的子类，不影响现有逻辑。
  - 支持更复杂的发送策略（如批量发送、优先级队列）。

4. 代码流程概述

1. 客户端调用 `send` 方法发送数据。
2. 数据被封装为 `Buffer` 对象，并加入 `_send_buf_waiting` 队列。
3. 如果 `try_flush` 为真，调用 `flushAll` 立即尝试发送。
4. `flushAll` 检查套接字状态，发送数据或处理超时。

5. 总结

这段代码展示了一种精细化的网络数据发送实现，具有以下优势:

- **灵活性**: 支持多种数据类型和发送场景。
- **安全性**: 通过锁机制确保多线程环境下的数据安全。
- **扩展性**: 基于多态设计，便于新增功能和改进逻辑。
- **可靠性**: 通过超时检测和错误处理，提高了发送过程的稳定性。