

ZLToolKit笔记5：服务器响应

2024年12月21日 15:34

从test_tcpEchoServer.cpp入手：

```
TcpServer::Ptr server(new TcpServer());  
server->start<EchoSession>(9000); //监听9000端口
```

- (1) 使用智能指针管理创建的TcpServer对象；
- (2) 启动监听，指定业务处理逻辑（EchoSession）。

其中开发者只需要专注于会话类（如EchoSession）的业务逻辑的实现，TcpServer会处理底层复杂的网络通信和多线程优化。

EchoSession继承于Session类，重写了响应的业务处理函数，以供在通信时进行回调。

如下所示：

```
class EchoSession: public Session {  
public:  
    EchoSession(const Socket::Ptr &sock) :  
        Session(sock) {  
        DebugL;  
    }  
    ~EchoSession() {  
        DebugL;  
    }  
    virtual void onRecv(const Buffer::Ptr &buf) override{  
        //处理客户端发送过来的数据 [AUTO-TRANSLATED:c095b82e]  
        // Handle data sent from the client  
        TraceL << buf->data() << " from port:" << get_local_port();  
        send(buf);  
    }  
    virtual void onError(const SockException &err) override{  
        //客户端断开连接或其他原因导致该对象脱离TCPServer管理 [AUTO-TRANSLATED:6b958a7b]  
        // Client disconnects or other reasons cause the object to be removed from TCPServer management  
        WarnL << err;  
    }  
    virtual void onManager() override{  
        //定时管理该对象，譬如会话超时检查 [AUTO-TRANSLATED:2caa54f6]  
        // Periodically manage the object, such as session timeout check  
        DebugL;  
    }  
private:  
    Ticker _ticker;  
};
```

1. 问题一：TcpServer对象构造的过程中会发生什么？

以智能指针来管理TcpServer对象的生命周期：

```
TcpServer::Ptr server(new TcpServer());
```

TcpServer的构造函数：

```
TcpServer::TcpServer(const EventPoller::Ptr &poller) : Server(poller) {  
    /*****  
        设置_multi_poller标志：指示是否启用了多线程负载均衡；  
        如果未显式指定poller（即poller == nullptr），表示多线程模式，将该标志设置为true。
```

```

        *****/
        _multi_poller = !poller;
        *****/
        设置_on_create_socket回调, 通过setOnCreateSocket方法, 配置默认的Socket创建逻辑
        *****/
        setOnCreateSocket(nullptr);
    }

```

TcpServer构造过程中首先会调用父类Server的构造函数, 在该函数中, 会从EventPoller池中获取一个EventPoller对象:

```

Server::Server(EventPoller::Ptr poller) {
    _poller = poller ? std::move(poller) : EventPollerPool::Instance().getPoller();
}

```

setOnCreateSocket函数用来设置_on_create_socket回调逻辑:

- 如果cb不为空, 则使用外部传入的回调逻辑; 否则使用默认的回调逻辑, 通过Socket::createSocket创建一个Socket对象;
- 更新子服务器的回调: 如果启用了多线程负载均衡, _cloned_server存储了多个子服务器, 遍历所有的子服务器, 为他们同步_on_create_socket的配置。

```

void TcpServer::setOnCreateSocket(Socket::onCreateSocket cb) {
    if (cb) {
        _on_create_socket = std::move(cb);
    } else {
        _on_create_socket = [] (const EventPoller::Ptr &poller) {
            return Socket::createSocket(poller, false);
        };
    }
    for (auto &pr : _cloned_server) {
        pr.second->setOnCreateSocket(cb);
    }
}

```

默认的Socket创建逻辑是通过如下方法实现的:

- 如果传入了Poller_in, 则使用该poller, 否则从poller池中获取一个poller线程;
- 调用new Socket(poller, enable_mutex)构造Socket对象, 指定poller和是否启用互斥锁;
- 自定义了Socket的析构逻辑, 如果poller仍然有效, 则在poller线程中异步销毁Socket对象, 否则直接销毁Socket对象。

```

Socket::Ptr Socket::createSocket(const EventPoller::Ptr &poller_in, bool enable_mutex) {
    auto poller = poller_in ? poller_in : EventPollerPool::Instance().getPoller();
    std::weak_ptr<EventPoller> weak_poller = poller;
    return Socket::Ptr(new Socket(poller, enable_mutex), [weak_poller](Socket *ptr) {
        if (auto poller = weak_poller.lock()) {
            poller->async([ptr]() { delete ptr; });
        } else {
            delete ptr;
        }
    });
}

```

这是Socket对象的构造逻辑 (构造函数): 初始化poller和互斥锁和事件回调

```

Socket::Socket(EventPoller::Ptr poller, bool enable_mutex)
    : _poller(std::move(poller))
    , _mtx_sock_fd(enable_mutex)
    , _mtx_event(enable_mutex)
    , _mtx_send_buf_waiting(enable_mutex)
    , _mtx_send_buf_sending(enable_mutex) {
    setOnRead(nullptr);
    setOnErr(nullptr);
    setOnAccept(nullptr);
    setOnFlush(nullptr);
    setOnBeforeAccept(nullptr);
    setOnSendResult(nullptr);
}

```

```
}
```

设计优点

灵活性:

支持用户自定义 Socket 的创建逻辑。

根据是否启用多线程自动调整服务器行为。

高性能:

多线程模式下, 可以均衡负载到多个 poller, 提升并发处理能力。

内存管理:

使用智能指针和自定义析构器, 确保资源能够被正确释放。

整个构造过程体现了 ZLToolKit 的模块化、高效和灵活的设计理念。

2. TCP服务器的启动过程分析

```
server->start<EchoSession>(9000); //监听9000端口
```

```
1. TcpServer::Ptr server(new TcpServer());          --> 创建服务器对象
2. server->start<EchoSession>(9000);                 --> 启动服务器
   └─ 设置会话分配器 (_session_alloc)
   └─ 调用 start_l 方法
       └─ 创建定时器 (onManagerSession)
       └─ 配置多线程负载均衡 (多 poller)
       └─ 开始监听 (listen on port 9000)
3. 客户端连接到来:
   └─ 调用 _session_alloc 创建 EchoSession 对象
   └─ EchoSession 管理连接, 处理数据
```

```
template <typename SessionType>
void start(uint16_t port, const std::string &host = ":", uint32_t backlog = 1024, const
std::function<void(std::shared_ptr<SessionType> &)> &cb = nullptr) {
    static std::string cls_name = toolkit::demangle(typeid(SessionType).name());
    // Session创建器, 通过它创建不同类型的服务器 [AUTO-TRANSLATED:f5585e1e]
    //Session creator, creates different types of servers through it
    _session_alloc = [cb](const TcpServer::Ptr &server, const Socket::Ptr &sock) {
        auto session = std::shared_ptr<SessionType>(new SessionType(sock), [](SessionType *ptr) {
            TraceP(static_cast<Session *>(ptr)) << "~" << cls_name;
            delete ptr;
        });
        if (cb) {
            cb(session);
        }
        TraceP(static_cast<Session *>(session.get())) << cls_name;
        session->setOnCreateSocket(server->on_create_socket);
        return std::make_shared<SessionHelper>(server, std::move(session), cls_name);
    };
    start_l(port, host, backlog);
}
```

start函数主要执行逻辑:

- a. 设置了会话session的创建逻辑_session_alloc, 以便为每个新连接创建一个EchoSession实例;
- b. 调用内部的start_l方法, 该方法是TcpServer的内部方法, 用于完成服务器的核心配置和启动。

注：这里不对session进行展开分析，其内容的具体分析参见[ZLToolKit笔记6](#)中。

接着来看start_l的实现过程：

a. 初始化事件处理

setupEvent()函数：

b. 定时管理会话对象：定时管理服务器中的TCP会话

- 作用：

- 创建一个定时器 _timer，每隔 2 秒调用一次 onManagerSession 方法。
- onManagerSession 用于清理无效的会话对象，保持服务器运行的健康状态。

- 注意：

- 使用 weak_ptr 避免定时器对 TcpServer 形成循环引用，确保服务器对象能被正确销毁。

c. 支持多线程负载均衡

如果启用了多线程负载均衡，则回为poller池中的每一个poller线程创建一个子服务器TcpServer，通过cloneFrom的方法将主服务器的配置复制到子服务器中，使他们监听同一个listen fd。

d. 启动监听

e. 克隆套接字到子服务器

- **克隆操作：**

- 将主服务器的监听套接字 _socket 克隆到所有子服务器的 _socket。
- 子服务器可以共享主服务器的监听功能，同时分担部分连接处理。

- **作用：**

- 在多线程模型下，子服务器和主服务器可以共同处理新连接，实现负载均衡。

```
void TcpServer::start_l(uint16_t port, const std::string &host, uint32_t backlog) {
    setupEvent();
    //新建一个定时器定时管理这些tcp会话 [AUTO-TRANSLATED:ef859bd7]
    //Create a new timer to manage these TCP sessions periodically
    weak_ptr<TcpServer> weak_self = std::static_pointer_cast<TcpServer>(shared_from_this());
    _timer = std::make_shared<Timer>(2.0f, [weak_self]() -> bool {
        auto strong_self = weak_self.lock();
        if (!strong_self) {
            return false;
        }
        strong_self->onManagerSession();
        return true;
    }, _poller);

    if (_multi_poller) {
        EventPollerPool::Instance().for_each([&](const TaskExecutor::Ptr &executor) {
            EventPoller::Ptr poller = static_pointer_cast<EventPoller>(executor);
            if (poller == _poller) {
                return;
            }
            auto &serverRef = _cloned_server[poller.get()];
            if (!serverRef) {
                serverRef = onCreateServer(poller);
            }
            if (serverRef) {
                serverRef->cloneFrom(*this);
            }
        });
    }

    if (!_socket->listen(port, host.c_str(), backlog)) {
        // 创建tcp监听失败，可能是由于端口占用或权限问题 [AUTO-TRANSLATED:88ebdefc]
        //TCP listener creation failed, possibly due to port occupation or permission issues
    }
}
```

```

        string err = (StrPrinter << "Listen on " << host << " " << port << " failed: "
<< get_uv_errmsg(true));
        throw std::runtime_error(err);
    }

    for (auto &pr: _cloned_server) {
        // 启动子Server [AUTO-TRANSLATED:1820131c]
        //Start the child Server
        pr.second->_socket->cloneSocket(*_socket);
    }
    InfoL << "TCP server listening on [" << host << "]: " << port;
}

```

有关定时器的话题在后续笔记中分析。

多线程负载均衡：

1. 为什么需要多线程负载均衡？

在高并发场景中，单线程通常难以应对大量的请求。引入多线程可以通过以下方式提高性能：

1. **并发处理能力**：多个线程可以同时处理不同的请求。
2. **资源利用率**：充分利用多核 CPU 的计算能力。
3. **响应时间**：减少单个任务等待处理的时间。

但多线程的引入也带来了挑战，尤其是如何合理地将任务分配到多个线程上，这就需要负载均衡。

2. 多线程负载均衡的目标

1. **均衡分配任务**：让每个线程的工作量大致相同，避免某些线程过载。
2. **最小化开销**：分配任务的过程不能过于复杂，否则会增加额外的计算开销。
3. **高效资源利用**：充分利用系统资源，包括 CPU、内存和网络。
4. **提升响应性**：尽量降低请求的延迟。

3. 多线程负载均衡的策略

3.1 静态负载均衡

任务分配策略在开始时已经确定，不会随着运行时的负载变化而调整。

- **特点：**
 - 简单高效，分配开销低。
 - 适合任务量和处理时间较为固定的场景。
- **常见方法：**
 1. **轮询分配：**
 - 每个任务依次分配到不同的线程。
 - 适合任务量均匀的场景。
 2. **哈希分配：**
 - 根据任务的某些特征（如 IP 地址、会话 ID）进行哈希，计算出对应的线程。
 - 适合需要保持任务一致性（如同一用户的请求总是分配到同一线程）的场景。
 3. **范围分配：**
 - 任务按照一定范围（如端口号、数据块范围）分配到特定线程。

3.2 动态负载均衡

根据运行时的实际负载情况动态调整任务的分配。

- **特点：**
 - 更灵活，可以应对负载不均或突发高负载场景。
 - 增加了分配的复杂度和开销。

- **常见方法：**
 1. **最少连接数分配：**
 - 将新任务分配到当前连接数最少的线程。
 - 常用于网络服务器。
 2. **任务队列长度分配：**
 - 根据线程的任务队列长度动态分配，队列短的线程优先。
 3. **负载预测分配：**
 - 利用历史数据或监控信息预测各线程的负载情况，提前分配任务。
 4. **动态迁移：**
 - 如果某个线程过载，可以将其任务动态迁移到其他线程。

4. 网络服务器中的多线程负载均衡

在网络服务器中，多线程负载均衡通常用于处理大量客户端请求。以下是常见实现方式：

4.1 基于轮询器模型

- **单线程轮询器：**
 - 每个线程都有一个独立的事件轮询器（如 `epoll`、`select`）。
 - 主线程监听所有连接请求，并将新连接分配给子线程的轮询器。
- **负载均衡：**
 - 使用静态策略（如轮询）或动态策略（如最少连接数）选择目标线程。
- **示例：**在 `TcpServer` 中，多个 `EventPoller` 是独立的轮询器，主线程监听新连接并分发到合适的 `EventPoller`。

4.2 基于线程池模型

- **线程池的工作机制：**
 - 预先创建固定数量的线程。
 - 所有任务都进入一个共享任务队列，线程池中的线程从队列中取任务执行。
- **负载均衡：**
 - 任务调度逻辑位于线程池的管理层。
 - 动态策略可以根据线程的繁忙程度调整任务分配。
- **优点：**
 - 线程复用减少了线程创建和销毁的开销。
 - 动态分配机制更适合负载波动大的场景。

4.3 主从模型

- **主线程：**
 - 专门负责监听和接收新连接。
 - 将新连接分配给子线程。
- **子线程：**
 - 每个线程独立处理分配到的连接。
 - 连接的后续所有请求都由该线程负责，避免线程间的数据竞争。
- **负载均衡：**
 - 主线程可以基于轮询或最少连接数等策略分配新连接。

5. 负载均衡的难点

5.1 线程间通信

负载均衡策略可能需要线程之间交换状态信息（如当前连接数或任务队列长度），如何高效地实现这些通信是一个难点。

5.2 数据竞争

当多个线程同时访问共享资源时，容易产生竞争条件，需要通过加锁等方式保证数据一致性，这会引入额外开销。

5.3 性能瓶颈

负载均衡逻辑本身可能成为性能瓶颈。例如，在动态负载均衡中，需要频繁查询线程状态或重新分配任务。

6. 实际案例

以下是多线程负载均衡的典型应用场景：

6.1 Web服务器

- 如 Nginx 的多进程模型，可以将新连接按轮询策略分配到不同的工作进程中。

6.2 数据库服务器

- 如 MySQL，使用线程池模型对查询任务进行动态分配。

6.3 高性能网络库

- 如 ZLToolKit（你提供的代码部分使用了类似的多轮询器模型），通过多线程负载均衡来实现高并发网络服务。

监听过程：

7. 总结

多线程负载均衡的核心是将任务合理分配到多个线程上，平衡负载，提高系统性能。常见的策略包括静态分配（如轮询）和动态分配（如最少连接数），具体选择需要根据业务场景和负载特性进行权衡。

```
bool SockUtil::listen(uint16_t port, const string &local_ip, int backlog) {
    closeSock();
    int fd = SockUtil::listen(port, local_ip.data(), backlog);
    return false;
}

return fromSock_l(std::make_shared<SockNum>(fd, SockNum::Sock_TCP_Server));
}
```

内部调用了SockUtil::listen方法，和fromSock_l方法。

a. SockUtil::listen

实质上这个过程封装了服务器端创建socket、命名socket和监听socket的过程。

back_log参数提示内核监听队列的最大长度，这里默认是1024，也就是说如果监听队列的长度超过backlog，服务器将不再受理新的客户连接。

```
int SockUtil::listen(const uint16_t port, const char *local_ip, int back_log) {
    int fd = -1;
    int family = support_ipv6() ? (is_ipv4(local_ip) ? AF_INET : AF_INET6) : AF_INET;
    if ((fd = (int)socket(family, SOCK_STREAM, IPPROTO_TCP)) == -1) {
        WarnL << "Create socket failed: " << get_uv_errmsg(true);
        return -1;
    }
    setReuseable(fd, true, false);
    setNoBlocked(fd);
    setCloExec(fd);
    if (bind_sock(fd, local_ip, port, family) == -1) {
        close(fd);
        return -1;
    }
    //开始监听 [AUTO-TRANSLATED:4404b1a8]
    //Start listening
    if (::listen(fd, back_log) == -1) {
        WarnL << "Listen socket failed: " << get_uv_errmsg(true);
        close(fd);
        return -1;
    }
    return fd;
}
```

这个函数封装了服务器端监听套接字的几个过程：1.创建套接字；2.绑定（命名）套接字；3.监听套接字。

注意：listen系统调用用来创建一个监听队列以存放待处理的客户连接。

b. fromSock_l

```

bool Socket::fromSock_l(SocketNum::Ptr sock) {
    if (!attachEvent(sock)) {
        return false;
    }
    setSock(std::move(sock));
    return true;
}

```

主要调用了attachEvent函数：因为传入的sock类型是TCP_Server，所以会调用下面加粗的代码。

主要就一句代码：_poller->addEvent(...)将该套接字上的可读事件或错误事件异步加入poller线程的事件监听中。

如果可读或错误事件发生，会异步执行任务回调onAccept(...)。

```

bool Socket::attachEvent(const SocketNum::Ptr &sock) {
    weak_ptr<Socket> weak_self = shared_from_this();
    if (sock->type() == SocketNum::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() == SocketNum::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() == SocketNum::Sock_UDP) {
                // udp ignore error
            } else {
                strong_self->emitErr(getSockErr(sock->rawFd()));
            }
        }
    });
    return -1 != result;
}

```

poller线程会监听客户端连接 (TCP Server) 事件，事件到达会触发回调执行onAccept函数：

```

int Socket::onAccept(const SocketNum::Ptr &sock, int event) noexcept {
    int fd;
    struct sockaddr_storage peer_addr;
    socklen_t addr_len = sizeof(peer_addr);
    while (true) {
        if (event & EventPoller::Event_Read) {
            do {
                fd = (int)accept(sock->rawFd(), (struct sockaddr *)&peer_addr, &addr_len);
            } while (-1 == fd && UV_EINTR == get_uv_error(true));
            if (fd == -1) {
                // accept失败 [AUTO-TRANSLATED:496cc51e]
            }
        }
    }
}

```



```

//Accept failed
int err = get_uv_error(true);
if (err == UV_EAGAIN) {
    // 没有新连接 [AUTO-TRANSLATED:4ddd97d6]
    //No new connection
    return 0;
}
auto ex = toSockException(err);
// emitErr(ex); https://github.com/ZLMediaKit/ZLMediaKit/issues/2946
ErrorL << "Accept socket failed: " << ex.what();
// 可能打开的文件描述符太多了:UV_EMFILE/UV_ENFILE [AUTO-TRANSLATED:ecd1b4f1]
//Possibly too many open file descriptors: UV_EMFILE/UV_ENFILE
#if (defined(HAS_EPOLL) && !defined(_WIN32)) || defined(HAS_KQUEUE)
    // 边缘触发, 还需要手动再触发accept事件, [AUTO-TRANSLATED:85fa9030]
    //Edge trigger, need to manually trigger the accept event again
    // wepoll, Edge-triggered (`EPOLLET`) mode isn't supported.
    std::weak_ptr<Socket> weak_self = shared_from_this();
    _poller->doDelayTask(100, [weak_self, sock]() {
        if (auto strong_self = weak_self.lock()) {
            // 100ms后再处理accept事件, 说不定已经有空闲的fd [AUTO-TRANSLATED:532951a2]
            //Process the accept event again after 100ms, maybe there are available fds
            strong_self->onAccept(sock, EventPoller::Event_Read);
        }
        return 0;
    });
    // 暂时不处理accept事件, 等待100ms后手动触发onAccept (只有EAGAIN读空后才能通过epoll再次触发
    事件) [AUTO-TRANSLATED:32636aea]
    //Temporarily do not process the accept event, wait 100ms and manually trigger onAccept
    (can only be triggered again through epoll after EAGAIN reads empty)
    return -1;
#else
    // 水平触发; 休眠10ms, 防止无谓的accept失败 [AUTO-TRANSLATED:6f8349bb]
    //Level trigger; sleep 10ms to prevent unnecessary accept failures
    this_thread::sleep_for(std::chrono::milliseconds(10));
    // 暂时不处理accept事件, 由于是水平触发, 下次还会再次自动进入onAccept函数 [AUTO-
    TRANSLATED:9aec1432]
    //Temporarily do not process the accept event, as it is level trigger, it will
    automatically enter the onAccept function again next time
    return -1;
#endif
}
SockUtil::setNoSigpipe(fd);
SockUtil::setNoBlocked(fd);
SockUtil::setNoDelay(fd);
SockUtil::setSendBuf(fd);
SockUtil::setRecvBuf(fd);
SockUtil::setCloseWait(fd);
SockUtil::setCloExec(fd);
Socket::Ptr peer_sock;
try {
    // 此处捕获异常, 目的是防止socket未accept尽, epoll边沿触发失效的问题 [AUTO-
    TRANSLATED:523d496d]
    //Catch exceptions here to prevent the problem of epoll edge trigger failure when the
    socket is not fully accepted
    LOCK_GUARD(_mtx_event);
    // 拦截Socket对象的构造 [AUTO-TRANSLATED:b38b67b9]
    //Intercept the Socket object's constructor
    peer_sock = _on_before_accept(_poller);
} catch (std::exception &ex) {
    ErrorL << "Exception occurred when emit on_before_accept: " << ex.what();
    close(fd);
    continue;
}
if (!peer_sock) {

```

```

        // 此处是默认构造行为，也就是子Socket共用父Socket的poll线程并且关闭互斥锁 [AUTO-
TRANSLATED:6c057de0]
        //This is the default construction behavior, which means the child Socket shares the parent
        Socket's poll thread and closes the mutex lock
        peer_sock = Socket::createSocket(_poller, false);
    }
    auto sock = std::make_shared<SockNum>(fd, SockNum::Sock_TCP);
    // 设置好fd,以备在onAccept事件中可以正常访问该fd [AUTO-TRANSLATED:e3e3c225]
    //Set the fd properly, so that it can be accessed normally in the onAccept event
    peer_sock->setSock(sock);
    // 赋值peer ip, 防止在执行setSock时, fd已经被reset断开 [AUTO-TRANSLATED:7ca197db]
    //Assign the peer ip to prevent the fd from being reset and disconnected when executing setSock
    memcpy(&peer_sock->_peer_addr, &peer_addr, addr_len);
    shared_ptr<void> completed(nullptr, [peer_sock, sock](void *) {
        try {
            // 然后把该fd加入poll监听(确保先触发onAccept事件然后再触发onRead等事件) [AUTO-
TRANSLATED:45618926]
            //Then add the fd to the poll monitoring (ensure that the onAccept event is triggered
            first, followed by onRead and other events)
            if (!peer_sock->attachEvent(sock)) {
                // 加入poll监听失败, 触发onErr事件, 通知该Socket无效 [AUTO-TRANSLATED:e81fd478]
                //If adding to poll monitoring fails, trigger the onErr event to notify that the
                Socket is invalid
                peer_sock->emitErr(SocketException(Err_eof, "add event to poller failed when accept a
                socket"));
            }
        } catch (std::exception &ex) {
            ErrorL << "Exception occurred: " << ex.what();
        }
    });
    try {
        // 此处捕获异常, 目的是防止socket未accept尽, epoll边沿触发失效的问题 [AUTO-
TRANSLATED:523d496d]
        //Catch exceptions here to prevent the problem of socket not being accepted and epoll edge
        triggering failure
        LOCK_GUARD(_mtx_event);
        // 先触发onAccept事件, 此时应该监听该Socket的onRead等事件 [AUTO-TRANSLATED:29734871]
        //First trigger the onAccept event, at this point, you should listen for onRead and other
        events of the Socket
        _on_accept(peer_sock, completed);
    } catch (std::exception &ex) {
        ErrorL << "Exception occurred when emit on_accept: " << ex.what();
        continue;
    }
}
if (event & EventPoller::Event_Error) {
    auto ex = getSockErr(sock->rawFd());
    emitErr(ex);
    ErrorL << "TCP listener occurred a err: " << ex.what();
    return -1;
}
}
}
}

```

监听到可读或异常事件后触发onAccept回调，accept系统调用被封装在该过程中，该系统调用从listen监听队列中接收一个连接。accept成功时返回一个新的连接socket，该socket唯一的标识了被接收的这个连接，服务器可通过读写该socket来与被接受连接对应的客户端通信。accept失败时返回-1并设置errno。

这个函数主要负责处理客户端连接请求的接收（通过accept系统调用），并针对不同的接收情况（如成功接收、接收失败）进行相应的处理，包括对新连接套接字的属性设置、关联到事件轮询机制以及触发相关自定义事件等操作。代码整体框架如下：

```

int Socket::onAccept(const SockNum::Ptr &sock, int event) noexcept {
    int fd;
    struct sockaddr_storage peer_addr;
    socklen_t addr_len = sizeof(peer_addr);
    while (true) {
        if (event & EventPoller::Event_Read) { ...

        if (event & EventPoller::Event_Error) {
            auto ex = getSockErr(sock->rawFd());
            emitErr(ex);
            ErrorL << "TCP listener occurred a err: " << ex.what();
            return -1;
        }
    }
}

```

1. 处理可读事件（接收连接）

通过位与操作判断当前事件中是否包含可读事件标识，如果包含则进入该分支，表示当前有客户端尝试连接，需要调用accept来接收连接。

```

if (event & EventPoller::Event_Read) {
    do {
        fd = (int)accept(sock->rawFd(), (struct sockaddr *)&peer_addr, &addr_len);
    } while (-1 == fd && UV_EINTR == get_uv_error(true));

    if (fd == -1) { ...

        SocketUtil::setNoSigpipe(fd);
        SocketUtil::setNoBlocked(fd);
        SocketUtil::setNoDelay(fd);
        SocketUtil::setSendBuf(fd);
        SocketUtil::setRecvBuf(fd);
        SocketUtil::setCloseWait(fd);
        SocketUtil::setCloExec(fd);

        Socket::Ptr peer_sock;
        try { ...
            ErrorL << "Exception occurred when emit on_before_accept: " << ex.what();
            close(fd);
            continue;
        }

        if (!peer_sock) { ...

            auto sock = std::make_shared<SockNum>(fd, SockNum::Sock_TCP);
            // 设置好fd,以备在onAccept事件中可以正常访问该fd [AUTO-TRANSLATED:e3e3c225] ...
            peer_sock->setSock(sock);
            // 赋值peer ip,防止在执行setSock时,fd已经被reset断开 [AUTO-TRANSLATED:7ca197db] ...
            memcpy(&peer_sock->peer_addr, &peer_addr, addr_len);

            shared_ptr<void> completed(nullptr, [peer_sock, sock](void *) { ...
            });

            try { ...
                ErrorL << "Exception occurred when emit on_accept: " << ex.what();
                continue;
            }
        }
    }
}

```

关于accept的问题：我们已经知道ZLToolKit这个项目中服务器端可以设置为多线程监听的，我有个问题就是：多个线程会监听同一个listen fd，如果这个listen fd上可读了，会触发onAccept的回调，那么多个线程监听的话是不是

会多次触发这个onAccept回调，这是否合理，进一步的，如果多个线程触发了onAccept回调，都会调用accept系统调用，这会发生什么？这会不会导致错误呢？

下面是chatgpt生成的回答：

在多线程中同时监听同一个 listen 文件描述符 (fd)，需要仔细设计以避免问题。我们来分析您的问题：

1. 多个线程会监听同一个 listen fd，可读时触发 onAccept

如果多个线程同时监听一个 listen fd，并且这个 fd 上有新的连接到来，那么多个线程有可能同时被唤醒。每个被唤醒的线程都会尝试调用 accept 系统调用。

这种设计在网络编程中并不罕见，但如果没有正确的同步机制，确实会带来问题。

2. 多个线程触发 onAccept，会调用 accept，是否合理？

是合理的，但需要小心设计。以下两种情况下是合理的：

1. Edge-triggered (ET) 模式：

在边缘触发模式下，事件只会在状态发生变化时触发（例如新连接到来时）。此时，唤醒的多个线程中，只有一个线程能够成功 accept。剩余线程调用 accept 时会返回 -1，错误码是 EAGAIN，表示没有更多可用连接。设计上必须捕获这种情况并正确处理。

2. Level-triggered (LT) 模式：

在水平触发模式下，事件会持续触发，直到条件消失（即所有新连接都被 accept 处理完）。多个线程同时调用 accept 可能会重复处理同一连接或触发无效调用，因此通常需要对 accept 操作进行某种程度的同步。

3. 多个线程调用 accept 会发生什么？是否会导致错误？

以下是多线程调用 accept 的行为分析：

正常情况

• 线程安全：

accept 本身是线程安全的，多个线程同时调用时，内核会确保同一连接只会被一个线程成功处理。其余线程的 accept 调用会返回 -1 并设置错误码 EAGAIN 或 EWOULDBLOCK，表示没有更多连接。

潜在问题

1. 竞态条件：

如果应用没有正确处理 EAGAIN 错误，可能会进入无效的逻辑分支（例如错误地认为连接已建立）。

2. 重复调用导致资源浪费：

如果没有同步机制，多个线程可能会反复尝试 accept，引起无谓的 CPU 占用。

3. 边缘触发模式下漏处理连接：

如果在 ET 模式下有新连接到来，但所有线程的 accept 调用都返回了 EAGAIN（例如多个线程同时调用时没有等待其他线程处理完），可能会丢失连接。

4. 如何正确设计？

以下是一些常见的设计方法，确保多线程监听同一个 listen fd 时行为合理：

方法 1：使用单线程处理 accept

将 accept 操作集中在一个线程中，处理完新连接后，将连接分发给其他工作线程处理。这种方法最简单，也最安全。

方法 2：使用多线程监听 + 锁同步

如果需要多个线程监听 listen fd，可以在调用 accept 前加锁，确保只有一个线程执行 accept 操作。例如使用 std::mutex 或其他同步机制。

方法 3：依赖内核的线程安全性（最常见）

现代操作系统的 accept 是线程安全的。多个线程可以同时调用 accept，内核会保证同一连接不会被重复处理。只需确保：

- 正确处理 EAGAIN 错误；

- 在 ET 模式下，循环读取所有连接，避免遗漏。

方法 4：使用事件驱动框架

如果使用事件驱动的框架（如 epoll、kqueue），可以将 listen fd 的事件分配到一个线程处理。其他线程只负责处理已建立的连接，避免竞争。

5. ZLToolKit 中的实现是否安全？

在 ZLToolKit 中，accept 的调用包含了对 EAGAIN 的处理，并且设计为重试机制（如延迟 100ms 再次触发 onAccept）。在这种情况下：

1. 多线程监听不会导致多个线程重复处理同一连接：

内核会确保 accept 的线程安全性。

2. EAGAIN 处理得当：

在 ZLToolKit 的代码中，accept 遇到 EAGAIN 时，会在边缘触发模式下手动触发下一次检查，确保所有连接都能被正确处理。

3. 并发竞争的消除：

尽管多个线程可能被唤醒尝试 accept，但只有一个线程会成功处理每个连接，其他线程会因返回 EAGAIN 而退出。

6. 总结

- 多线程监听同一 listen fd 是合理的，但需要正确设计和处理。
- accept 是线程安全的，多个线程同时调用时不会导致连接重复处理。
- ZLToolKit 的实现使用了事件驱动和 EAGAIN 处理机制，能安全地在多线程环境下使用。

如果需要进一步优化，可以考虑集中 accept 到一个线程，其他线程专注于处理已建立的连接，从而减少线程竞争的开销。

总而言之，在ZLToolKit设计中，将有多个线程来监听内核队列，如果有可读事件的触发（有连接socket到达监听队列等待accept），将同时触发多个线程进行事件回调（处理），也就是上面的onAccept过程，它是通过**抢占式accept**的方式获取监听队列上等待accept的连接请求，多个线程争着accept连接，但并不会重复接收，因为**accept是线程安全的**，对同一个连接请求的accept，多次调用只会保证一次接收，其余会返回-1并设置errno状态。

accept如果失败：

代码中体现了两种情况：（1）没有新的连接供accept，直接返回0即可；（2）有新的连接，但已打开的文件描述符太多了（进程可打开的文件描述符一般是有上限的）：这里分边缘触发和水平触发两种情况：前者进行了**延迟调用onAccept**，后者不管，只是休眠了一定时间，防止无谓的accept失败。

```

        if (fd == -1) {
            // accept失败 [AUTO-TRANSLATED:496cc51e] ...
            int err = get_uv_error(true);
            if (err == UV_EAGAIN) {
                // 没有新连接 [AUTO-TRANSLATED:4ddd97d6] ...
                return 0;
            }
            auto ex = toSockException(err);
            // emitErr(ex); https://github.com/ZLMediaKit/ZLMediaKit/issues/2946
            ErrorL << "Accept socket failed: " << ex.what();
            // 可能打开的文件描述符太多了:UV_EMFILE/UV_ENFILE [AUTO-TRANSLATED:ecd1b4f1] ...
#ifdef (defined(HAS_EPOLL) && !defined(_WIN32)) || defined(HAS_KQUEUE)
            // 边缘触发, 还需要手动再触发accept事件, [AUTO-TRANSLATED:85fa9030]
            // Edge trigger, need to manually trigger the accept event again
            // wepoll, Edge-triggered (`EPOLLET`) mode isn't supported.
            std::weak_ptr<Socket> weak_self = shared_from_this();
            _poller->doDelayTask(100, [weak_self, sock]() {
                if (auto strong_self = weak_self.lock()) {
                    // 100ms后再处理accept事件, 说不定已经有空闲的fd [AUTO-TRANSLATED:532951a2]
                    // Process the accept event again after 100ms, maybe there are available fds
                    strong_self->onAccept(sock, EventPoller::Event_Read);
                }
                return 0;
            });
            // 暂时不处理accept事件, 等待100ms后手动触发onAccept(只有EAGAIN读空后才能通过epoll再次触发事件) [AUTO-TRANSLATED:532951a2]
            // Temporarily do not process the accept event, wait 100ms and manually trigger onAccept (can only be triggered when EAGAIN)
            return -1;
#else
            // 水平触发; 休眠10ms, 防止无谓的accept失败 [AUTO-TRANSLATED:6f8349bb] ...
            this_thread::sleep_for(std::chrono::milliseconds(10));
            // 暂时不处理accept事件, 由于是水平触发, 下次还会再次自动进入onAccept函数 [AUTO-TRANSLATED:9aec1432] ...
            return -1;
#endif
        }
    }
}

```

accept成功:

(1) 设置新接收的套接字fd的系列相关属性:

- `SocketUtil::setNoSigpipe(fd);`: 设置套接字不产生SIGPIPE信号, 避免在向对端已关闭的套接字写数据时导致进程异常退出。
- `SocketUtil::setNoBlocked(fd);`: 将套接字设置为非阻塞模式, 便于后续在异步事件驱动框架下进行高效处理。
- `SocketUtil::setNoDelay(fd);`: 关闭Nagle算法, 使得数据能够尽快发送出去, 降低延迟。
- `SocketUtil::setSendBuf(fd);`: 设置发送缓冲区相关属性 (具体设置逻辑在SocketUtil函数内部实现)。
- `SocketUtil::setRecvBuf(fd);`: 设置接收缓冲区相关属性。
- `SocketUtil::setCloseWait(fd);`: 可能涉及到设置套接字关闭等待相关的属性 (具体依赖于SocketUtil里的实现)。
- `SocketUtil::setCloExec(fd);`: 设置文件描述符在执行exec系列系统调用时自动关闭, 避免资源泄露等问题。

(2) 创建和配置新连接对应的socket对象:

```

Socket::Ptr peer_sock;
try {
    // 此处捕获异常, 目的是防止socket未accept尽, epoll边沿触发失效的问题 [AUTO-TRANSLATED:523d496d]
    // Catch exceptions here to prevent the problem of epoll edge trigger failure when the
    // socket is not fully accepted
    LOCK_GUARD(_mtx_event);
    // 拦截Socket对象的构造 [AUTO-TRANSLATED:b38b67b9]
    // Intercept the Socket object's constructor
    peer_sock = _on_before_accept(_poller);
} catch (std::exception &ex) {
    ErrorL << "Exception occurred when emit on_before_accept: " << ex.what();
    close(fd);
    continue;
}
if (!peer_sock) {

```



```

        // 此处是默认构造行为，也就是子Socket共用父Socket的poll线程并且关闭互斥锁 [AUTO-TRANSLATED:6c057de0]
        //This is the default construction behavior, which means the child Socket shares the parent Socket's poll thread and closes the mutex lock
        peer_sock = Socket::createSocket(_poller, false);
    }
    auto sock = std::make_shared<SockNum>(fd, SockNum::Sock_TCP);
    // 设置好fd,以备在onAccept事件中可以正常访问该fd [AUTO-TRANSLATED:e3e3c225]
    //Set the fd properly, so that it can be accessed normally in the onAccept event
    peer_sock->setSock(sock);
    // 赋值peer ip, 防止在执行setSock时, fd已经被reset断开 [AUTO-TRANSLATED:7ca197db]
    //Assign the peer ip to prevent the fd from being reset and disconnected when executing setSock
    memcpy(&peer_sock->_peer_addr, &peer_addr, addr_len);

```

(3) 将新连接的套接字添加到事件轮询并触发相关事件

- a. 首先创建了一个自定义的智能指针，之所以是自定义的，因为其指定了析构函数（通过lambda表达式定义），在析构过程中会尝试将刚创建的连接socket加入epoll监听中。

```

shared_ptr<void> completed(nullptr, [peer_sock, sock](void *) {
    try {
        // 然后把该fd加入poll监听(确保先触发onAccept事件然后再触发onRead等事件) [AUTO-TRANSLATED:45618926]
        //Then add the fd to the poll monitoring (ensure that the onAccept event is triggered first, followed by onRead and other events)
        if (!peer_sock->attachEvent(sock)) {
            // 加入poll监听失败, 触发onErr事件, 通知该Socket无效 [AUTO-TRANSLATED:e81fd478]
            //If adding to poll monitoring fails, trigger the onErr event to notify that the Socket is invalid
            peer_sock->emitErr(SocketException(Err_eof, "add event to poller failed when accept a socket"));
        }
    } catch (std::exception &ex) {
        ErrorL << "Exception occurred: " << ex.what();
    }
});

```

- b. 接着调用_on_accept函数，传入新创建的连接套接字和complete指针指针。

```

try {
    // 此处捕获异常，目的是防止socket未accept尽，epoll边沿触发失效的问题 [AUTO-TRANSLATED:523d496d]
    //Catch exceptions here to prevent the problem of socket not being accepted and epoll edge triggering failure
    LOCK_GUARD(_mtx_event);
    // 先触发onAccept事件，此时应该监听该Socket的onRead等事件 [AUTO-TRANSLATED:29734871]
    //First trigger the onAccept event, at this point, you should listen for onRead and other events of the Socket
    _on_accept(peer_sock, completed);
} catch (std::exception &ex) {
    ErrorL << "Exception occurred when emit on_accept: " << ex.what();
    continue;
}

```

那么问题就转向这里调用的_on_accept或者前面的_on_before_accept函数的定义在哪呢？？它是干什么的呢？？这两变量在start_l函数的最开始就已经设定，通过TcpServer::setupEvent()。定义如下：

```

void TcpServer::setupEvent() {
    _socket = createSocket(_poller);
    weak_ptr<TcpServer> weak_self = std::static_pointer_cast<TcpServer>(shared_from_this());
    _socket->setOnBeforeAccept([weak_self](const EventPoller::Ptr &poller) -> Socket::Ptr {
        if (auto strong_self = weak_self.lock()) {
            return strong_self->onBeforeAcceptConnection(poller);
        }
    })
}

```

```

        return nullptr;
    });
    _socket->setOnAccept([weak_self](Socket::Ptr &sock, shared_ptr<void> &complete) {
        if (auto strong_self = weak_self.lock()) {
            auto ptr = sock->getPoller().get();
            auto server = strong_self->getServer(ptr);
            ptr->async([server, sock, complete]() {
                //该tcp客户端派发给对应线程的TcpServer服务器 [AUTO-TRANSLATED:662b882f]
                //This TCP client is dispatched to the corresponding thread of the TcpServer server
                server->onAcceptConnection(sock);
            });
        }
    });
}

```

可以看到通过setOnBeforeAccept和onAcceptConnection这两函数分别设置了：（1）设置accept时，socket构造事件回调；内部调用了onBeforeAcceptConnection函数；（2）设置tcp监听接收到连接回调，内部调用了onAcceptConnection函数。

onAcceptConnection过程

这个onAcceptConnection的过程是下发给对应线程的tcpServer服务器异步进行的。

看一下这个函数中整体框架：

```

> // 接收到客户端连接请求 [AUTO-TRANSLATED:8a67b72a] ...
Session::Ptr TcpServer::onAcceptConnection(const Socket::Ptr &sock) {
    assert(_poller->isCurrentThread());
    weak_ptr<TcpServer> weak_self = std::static_pointer_cast<TcpServer>(shared_from_this());
> //创建一个Session;这里实现创建不同的服务会话实例 [AUTO-TRANSLATED:9ed745be] ...
    auto helper = _session_alloc(std::static_pointer_cast<TcpServer>(shared_from_this()), sock);
    auto session = helper->session();
> //把本服务器的配置传递给Session [AUTO-TRANSLATED:e3711484] ...
    session->attachServer(*this);

> //_session_map::emplace肯定能成功 [AUTO-TRANSLATED:09d4aef7] ...
    auto success = _session_map.emplace(helper.get(), helper).second;
    assert(success == true);

    weak_ptr<Session> weak_session = session;
> //会话接收数据事件 [AUTO-TRANSLATED:f3f4cbbb] ...
> sock->setOnRead([weak_session](const Buffer::Ptr &buf, struct sockaddr *, int) { ...
    });

    SessionHelper *ptr = helper.get();
    auto cls = ptr->className();
> //会话接收到错误事件 [AUTO-TRANSLATED:b000e868] ...
> sock->setOnErr([weak_self, weak_session, ptr, cls](const SocketException &err) { ...
    });
    return session;
}

```

接收到客户端连接请求并将该连接下发给子服务器（TcpServer）对应的poller线程处理。

在这个函数中很明显是完成了会话的创建通过回调_session_alloc，其次设置了相应的回调函数。

_session_alloc的定义在start函数中完成：

```

_session_alloc = [cb](const TcpServer::Ptr &server, const Socket::Ptr &sock) {
    auto session = std::shared_ptr<SessionType>(new SessionType(sock), [](SessionType *ptr) {
        TraceP(static_cast<Session *>(ptr)) << "~~" << cls_name;
        delete ptr;
    });
    if (cb) {
        cb(session);
    }
    TraceP(static_cast<Session *>(session.get())) << cls_name;
}

```



```

        session->setOnCreateSocket(server->_on_create_socket);
        return std::make_shared<SessionHelper>(server, std::move(session), cls_name);
    };

```

session的分析, 参见[ZLToolKit笔记6](#)。

通过以上步骤, 再来看start的过程:

```
server->start<EchoSession>(9000); //监听9000端口
```

```

1. TcpServer::Ptr server(new TcpServer());          --> 创建服务器对象
2. server->start<EchoSession>(9000);                --> 启动服务器
   └─ 设置会话分配器 (_session_alloc)
   └─ 调用 start_l 方法
       └─ 创建定时器 (onManagerSession)
       └─ 配置多线程负载均衡 (多 poller)
       └─ 开始监听 (listen on port 9000)
3. 客户端连接到来:
   └─ 调用 _session_alloc 创建 EchoSession 对象
   └─ EchoSession 管理连接, 处理数据

```

要注意的是: 创建session对象后, 需将这个连接socket fd的监听任务加入epoll中, 这个过程实质上是由completed这个智能指针的析构函数完成的。

```

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;
}

```