ZLToolKit笔记3：客户端连接

2024年12月19日　12:47

从test_tcpClient.cpp的测试程序开始分析：

1. 实现了客户端逻辑，重写了相关的虚函数，来处理连接、接收数据、发送数据和错误等事件，这些函数将在相应事件发生后进行回调。
2. main函数中：
   （1）初始化日志系统
   （2）创建客户端连接
   （3）退出（信号触发时，唤醒信号量，退出程序）

```
int main() {
    // 设置日志系统  [AUTO-TRANSLATED:45646031]
    // Set up the logging system
    Logger::Instance().add(std::make_shared<ConsoleChannel>());
    Logger::Instance().setWriter(std::make_shared<AsyncLogWriter>());

    TestClient::Ptr client(new TestClient());//必须使用智能指针
    client->startConnect("127.0.0.1",9000);//连接服务器

    TcpClientWithSSL<TestClient>::Ptr clientSSL(new TcpClientWithSSL<TestClient>());//必须使用智能指针
    clientSSL->startConnect("127.0.0.1",9001);//连接服务器

    //退出程序事件处理  [AUTO-TRANSLATED:80065cb7]
    // Exit program event handling
    static semaphore sem;
    signal(SIGINT, [](int) { sem.post(); });// 设置退出信号
    sem.wait();
    return 0;
}
```

首先分析在客户端如何进行连接服务器的。

```
TestClient::Ptr client(new TestClient());//必须使用智能指针
client->startConnect("127.0.0.1",9000);//连接服务器
```

两个步骤：

（1）使用智能指针创建了TestClient的实例；
（2）调用startConnect方法，连接到指定的服务器地址和端口

1. TestClient实例是如何构造的

（1）TestClient的构造函数中会首先调用父类TcpClient的构造函数：

```
TestClient():TcpClient() {
    DebugL;
}
```

（2）TcpClient的构造函数：

```
// 声明
TcpClient(const EventPoller::Ptr &poller = nullptr);
// 定义
TcpClient::TcpClient(const EventPoller::Ptr &poller) :
SocketHelper(nullptr) {
    setPoller(poller ? poller : EventPollerPool::Instance().getPoller());
    setOnCreateSocket([](const EventPoller::Ptr &poller) {
```

```
        //TCP客户端默认开启互斥锁  [AUTO-TRANSLATED:94fad9cd]
        //TCP client defaults to enabling mutex lock
        return Socket::createSocket(poller, true);
    });
}
```

a. 首先调用了SocketHelper的构造函数：

```
SocketHelper::SocketHelper(const Socket::Ptr &sock) {
    setSock(sock);
    setOnCreateSocket(nullptr);
}
```

传入的参数sock为nullptr，在setSock中设置了_sock为nullptr：

```
void SocketHelper::setSock(const Socket::Ptr &sock) {
    _sock = sock;
    if (_sock) {
        _poller = _sock->getPoller();
    }
}
```

之后，调用了setOnCreateSocket函数设置了创建socket的方式：

```
void SocketHelper::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); };
    }
}
```

用_on_create_socket包装了真正创建socket的方式。

```
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;
        }
    });
}
```

以上_sock、_poller和_on_create_socket都是SocketHelper类中的私有成员变量，定义如下：

```
private:
    bool _try_flush = true;
    Socket::Ptr _sock;
    EventPoller::Ptr _poller;
    Socket::onCreateSocket _on_create_socket;
```

b. 其次在TcpClient的构造函数中调用了SocketHelper类中的成员方法setPoller和setOnCreateSocket.

```
    setPoller(poller ? poller : EventPollerPool::Instance().getPoller());
```

poller就是一个事件轮询器的线程，应该从线程池中获得。
该项目中poller线程由PollerPoll（线程池）中获得：

```
EventPollerPool::Instance().getPoller(),
```

（1）通过EventPollerPoll::Instance()获取EventPollerPoll的单例。实际上返回的是对管理事件池的智能指针的一个引用。

在EventPoller.cpp中有这句代码：

```
INSTANCE_IMP(EventPollerPool)
```

这其实是个宏定义，宏定义和宏展开如下，也就是说**上面个这句代码会在预处理的时候被下面黑色的代码所替换。**

```
#define INSTANCE_IMP(class_name, ...) \
class_name &class_name::Instance() { \
    static std::shared_ptr<class_name> s_instance(new
class_name(__VA_ARGS__)); \
    static class_name &s_instance_ref = *s_instance; \
    return s_instance_ref; \
}
```

扩展到：

```
EventPollerPool &EventPollerPool::Instance() {
    static std::shared_ptr<EventPollerPool> s_instance(new
EventPollerPool());
    static EventPollerPool &s_instance_ref = *s_instance;
    return s_instance_ref;
}
```

注意：在这里如果是首次调用，则会创建一个poller线程池的实例，调用了EventPollerPool的构造函数：

```
EventPollerPool::EventPollerPool() {
    auto size = addPoller("event poller", s_pool_size,
ThreadPool::PRIORITY_HIGHEST, true, s_enable_cpu_affinity);
    NOTICE_EMIT(EventPollerPoolOnStartedArgs, kOnStarted, *this, size);
    InfoL << "EventPoller created size: " << size;
}
```

更多关于这里单例模式的设计参见笔记9：ZLToolKit笔记9：设计模式运用

```
size_t TaskExecutorGetterImp::addPoller(const string &name, size_t size,
int priority, bool register_thread, bool enable_cpu_affinity) {
    auto cpus = thread::hardware_concurrency();
    size = size > 0 ? size : cpus;
    for (size_t i = 0; i < size; ++i) {
        auto full_name = name + " " + to_string(i);
        auto cpu_index = i % cpus;
        EventPoller::Ptr poller(new EventPoller(full_name));
        poller->runLoop(false, register_thread);
        poller->async([cpu_index, full_name, priority, enable_cpu_affinity]
() {
            // 设置线程优先级 [AUTO-TRANSLATED:2966f860]
            //Set thread priority
            ThreadPool::setPriority((ThreadPool::Priority)priority);
            // 设置线程名 [AUTO-TRANSLATED:f5eb4704]
            //Set thread name
            setThreadName(full_name.data());
            // 设置cpu亲和性 [AUTO-TRANSLATED:ba213aed]
            //Set CPU affinity
            if (enable_cpu_affinity) {
                setThreadAffinity(cpu_index);
            }
        });
        _threads.emplace_back(std::move(poller));
    }
    return size;
}
```

注意在执行poller->runLoop()时，第一个参数是false，这会在runLoop函数内部创

建线程，这个新线程会去递归的异步执行runLoop过程，此时参数为true，这个新线程会执行事件循环等待的过程。

```
void EventPoller::runLoop(bool blocked, bool ref_self) {
    if (blocked) {
        if (ref_self) {
            s_current_poller = shared_from_this();
        }
        _sem_run_started.post();
        _exit_flag = false;
        uint64_t minDelay;
#if defined(HAS_EPOLL)
        struct epoll_event events[EPOLL_SIZE];
        while (!_exit_flag) {
            minDelay = getMinDelay();
            startSleep();//用于统计当前线程负载情况
            int ret = epoll_wait(_event_fd, events, EPOLL_SIZE, minDelay ?
minDelay : -1);
            sleepWakeUp();//用于统计当前线程负载情况
            if (ret <= 0) {
                //超时或被打断  [AUTO-TRANSLATED:7005fded]
                //Timed out or interrupted
                continue;
            }
            _event_cache_expired.clear();
            for (int i = 0; i < ret; ++i) {
                struct epoll_event &ev = events[i];
                int fd = ev.data.fd;
                if (_event_cache_expired.count(fd)) {
                    //event cache refresh
                    continue;
                }
                auto it = _event_map.find(fd);
                if (it == _event_map.end()) {
                    epoll_ctl(_event_fd, EPOLL_CTL_DEL, fd, nullptr);
                    continue;
                }
                auto cb = it->second;
                try {
                    (*cb)(toPoller(ev.events));
                } catch (std::exception &ex) {
                    ErrorL << "Exception occurred when do event task: "
<< ex.what();
                }
            }
        }
#elif defined(HAS_KQUEUE)
        struct kevent kevents[KEVENT_SIZE];
        while (!_exit_flag) {
            minDelay = getMinDelay();
            struct timespec timeout = { (long)minDelay / 1000, (long)
minDelay % 1000 * 1000000 };
            startSleep();
            int ret = kevent(_event_fd, nullptr, 0, kevents, KEVENT_SIZE,
minDelay ? &timeout : nullptr);
            sleepWakeUp();
            if (ret <= 0) {
                continue;
            }
            _event_cache_expired.clear();
            for (int i = 0; i < ret; ++i) {
                auto &kev = kevents[i];
                auto fd = kev.ident;
```

```
                if (_event_cache_expired.count(fd)) {
                    // event cache refresh
                    continue;
                }
                auto it = _event_map.find(fd);
                if (it == _event_map.end()) {
                    EV_SET(&kev, fd, kev.filter, EV_DELETE, 0, 0, nullptr);
                    kevent(_event_fd, &kev, 1, nullptr, 0, nullptr);
                    continue;
                }
                auto cb = it->second;
                int event = 0;
                switch (kev.filter) {
                    case EVFILT_READ: event = Event_Read; break;
                    case EVFILT_WRITE: event = Event_Write; break;
                    default: WarnL << "unknown kevent filter: "
<< kev.filter; break;
                }
                try {
                    (*cb)(event);
                } catch (std::exception &ex) {
                    ErrorL << "Exception occurred when do event task: "
<< ex.what();
                }
            }
        }
#else
        int ret, max_fd;
        FdSet set_read, set_write, set_err;
        List<Poll_Record::Ptr> callback_list;
        struct timeval tv;
        while (!_exit_flag) {
            //定时器事件中可能操作_event_map [AUTO-TRANSLATED:f2a50ee2]
            //Possible operations on _event_map in timer events
            minDelay = getMinDelay();
            tv.tv_sec = (decltype(tv.tv_sec)) (minDelay / 1000);
            tv.tv_usec = 1000 * (minDelay % 1000);
            set_read.fdZero();
            set_write.fdZero();
            set_err.fdZero();
            max_fd = 0;
            for (auto &pr : _event_map) {
                if (pr.first > max_fd) {
                    max_fd = pr.first;
                }
                if (pr.second->event & Event_Read) {
                    set_read.fdSet(pr.first);//监听管道可读事件
                }
                if (pr.second->event & Event_Write) {
                    set_write.fdSet(pr.first);//监听管道可写事件
                }
                if (pr.second->event & Event_Error) {
                    set_err.fdSet(pr.first);//监听管道错误事件
                }
            }
            startSleep();//用于统计当前线程负载情况
            ret = zl_select(max_fd + 1, &set_read, &set_write, &set_err,
minDelay ? &tv : nullptr);
            sleepWakeUp();//用于统计当前线程负载情况
            if (ret <= 0) {
                //超时或被打断 [AUTO-TRANSLATED:7005fded]
                //Timed out or interrupted
                continue;
```

```
            }
            _event_cache_expired.clear();
            //收集select事件类型 [AUTO-TRANSLATED:9a5c41d3]
            //Collect select event types
            for (auto &pr : _event_map) {
                int event = 0;
                if (set_read.isSet(pr.first)) {
                    event |= Event_Read;
                }
                if (set_write.isSet(pr.first)) {
                    event |= Event_Write;
                }
                if (set_err.isSet(pr.first)) {
                    event |= Event_Error;
                }
                if (event != 0) {
                    pr.second->attach = event;
                    callback_list.emplace_back(pr.second);
                }
            }
            callback_list.for_each([&](Poll_Record::Ptr &record) {
                if (_event_cache_expired.count(record->fd)) {
                    //event cache refresh
                    return;
                }
                try {
                    record->call_back(record->attach);
                } catch (std::exception &ex) {
                    ErrorL << "Exception occurred when do event task: "
<< ex.what();
                }
            });
            callback_list.clear();
        }
#endif //HAS_EPOLL
    } else {
        _loop_thread = new thread(&EventPoller::runLoop, this, true,
ref_self);
        _sem_run_started.wait();
    }
}
```

之后会将设置Poller线程相关属性的任务打入到任务队列中，并通过管道通知主线程执行此任务。

```
Task::Ptr EventPoller::async(TaskIn task, bool may_sync) {
    return async_l(std::move(task), may_sync, false);
}
Task::Ptr EventPoller::async_first(TaskIn task, bool may_sync) {
    return async_l(std::move(task), may_sync, true);
}
Task::Ptr EventPoller::async_l(TaskIn task, bool may_sync, bool first) {
    TimeTicker();
    if (may_sync && isCurrentThread()) {
        task();
        return nullptr;
    }
    auto ret = std::make_shared<Task>(std::move(task));
    {
        lock_guard<mutex> lck(_mtx_task);
        if (first) {
            _list_task.emplace_front(ret);
        } else {
```

```
            _list_task.emplace_back(ret);
        }
    }
    //写数据到管道,唤醒主线程 [AUTO-TRANSLATED:2ead8182]
    //Write data to the pipe and wake up the main thread
    _pipe.write("", 1);
    return ret;
}
```

那么这里就有问题了，_pipe管道是线程间通信的中介，在async_l中打入任务后，会给管道写信息，从而唤醒主线程（通过epoll机制监听），那么必然有初始化的过程，也就是需要事先将要监听的这个管道端加入到内核事件表中，这个过程是在什么地方执行的呢?

我们带着该疑问去看EventPoller的构造函数:

```
EventPoller::EventPoller(std::string name) {
#if defined(HAS_EPOLL) || defined(HAS_KQUEUE)
    _event_fd = create_event();
    if (_event_fd == -1) {
        throw runtime_error(StrPrinter << "Create event fd failed: "
<< get_uv_errmsg());
    }
    SockUtil::setCloExec(_event_fd);
#endif //HAS_EPOLL
    _name = std::move(name);
    _logger = Logger::Instance().shared_from_this();
    addEventPipe();
}

void EventPoller::addEventPipe() {
    SockUtil::setNoBlocked(_pipe.readFD());
    SockUtil::setNoBlocked(_pipe.writeFD());
    // 添加内部管道事件 [AUTO-TRANSLATED:6a72e39a]
    //Add internal pipe event
    if (addEvent(_pipe.readFD(), EventPoller::Event_Read, [this](int event)
{ onPipeEvent(); }) == -1) {
        throw std::runtime_error("Add pipe fd to poller failed");
    }
}
```

发现在Poller的构造函数中，也就是Poller实例化的过程中已经添加了内部管道事件以进行监听。

（2）getPoller方法尝试获取一个合适的poller对象

```
EventPoller::Ptr EventPollerPool::getPoller(bool prefer_current_thread) {
    auto poller = EventPoller::getCurrentPoller();
    if (prefer_current_thread && _prefer_current_thread && poller) {
        return poller;
    }
    return static_pointer_cast<EventPoller>(getExecutor());
}

static thread_local std::weak_ptr<EventPoller> s_current_poller;
// static
EventPoller::Ptr EventPoller::getCurrentPoller() {
    return s_current_poller.lock();
}
```

a. 首先调用EventPoller::getCurrentPoller方法检查当前线程是否已经有一个poller，有则直接返回；
b. 否则调用getExcutor方法，从Poller线程池中返回一个线程执行器（Poller）。

已知在前面addPoller方法中已经创建了Poller线程并将其用智能指针管理，最后将所有的这些管理Poller线程的智能指针压入到了_thread中，这是一个std::vector的对象，存储所有的Poller线程（任务执行器）。

```
TaskExecutor::Ptr TaskExecutorGetterImp::getExecutor() {
    auto thread_pos = _thread_pos;     // 上次选中的线程位置，用于从上次选择的位置开始查找，避免重复选择，提高效率
    if (thread_pos >= _threads.size()) {
        thread_pos = 0;       //超出可选范围，则从头开始选择。
    }
    TaskExecutor::Ptr executor_min_load = _threads[thread_pos];
    auto min_load = executor_min_load->load();     // load函数用于返回当前线程的cpu使用率
    //  for循环用于寻找负载最小的Poller线程
    for (size_t i = 0; i < _threads.size(); ++i) {
        ++thread_pos;
        if (thread_pos >= _threads.size()) {
            thread_pos = 0;
        }
        auto th = _threads[thread_pos];
        auto load = th->load();
        if (load < min_load) {
            min_load = load;
            executor_min_load = th;
        }
        if (min_load == 0) {
            break;    //如果某个线程的负载为0说明当前没有任务，直接选用。
        }
    }
    _thread_pos = thread_pos;
    return executor_min_load;
}
```

保证线程的选择具有负载均衡性。

2. startConnect服务器

```
client->startConnect("127.0.0.1",9000);//连接服务器
```

startConnect是TcpClient类中用于启动连接的核心逻辑。

（1）定时器设置

```
_timer = std::make_shared<Timer>(2.0f, [weak_self]() {
        auto strong_self = weak_self.lock();
        if (!strong_self) {
            return false;
        }
        strong_self->onManager();
        return true;
    }, getPoller());
```

（2）设置套接字

```
    setSock(createSocket());
```

（3）设置错误处理回调

```
    auto sock_ptr = getSock().get();
    sock_ptr->setOnErr([weak_self, sock_ptr](const SockException &ex) {
        auto strong_self = weak_self.lock();
        if (!strong_self) {
            return;
        }
        if (sock_ptr != strong_self->getSock().get()) {
            //已经重连socket，上次的socket的事件忽略掉 [AUTO-TRANSLATED:9bf35a7a]
```

```
            //Socket has been reconnected, last socket's event is ignored
            return;
        }
        strong_self->_timer.reset();
        TraceL << strong_self->getIdentifier() << " on err: " << ex;
        strong_self->onError(ex);
    });
```

- setOnErr:
  - 定义一个错误处理回调，在套接字发生错误（如连接失败或断开）时触发。
- **回调逻辑**:
  i. 锁定弱引用，检查对象是否已销毁。
  ii. 确保当前套接字是最新的（避免错误处理影响已被替换的套接字）。
  iii. 清理定时器 _timer，避免资源泄漏。
  iv. 记录错误日志。
  v. 调用 onError 方法，让子类处理具体的错误逻辑。

(4) 调用套接字连接

```
TraceL << getIdentifier() << " start connect " << url << ":" << port;
    sock_ptr->connect(url, port, [weak_self](const SockException &err) {
        auto strong_self = weak_self.lock();
        if (strong_self) {
            strong_self->onSockConnect(err);
        }
    }, timeout_sec, _net_adapter, local_port);
```

connect:

- 使用 Socket 类的 connect 方法启动连接。
- 参数解析:
  1. url & port: 目标地址和端口。
  2. **连接回调**:
     - 锁定弱引用，确保对象有效。
     - 调用 onSockConnect 方法，处理连接结果（成功或失败）。
  3. timeout_sec: 超时时间，防止连接阻塞。
  4. _net_adapter: 网络适配器（如果需要绑定特定的网卡）。
  5. local_port: 本地端口号。

3. Connect过程分析

在startConnect的最后调用了socket类中的connect方法，该方法定义如下：

connect方法：

```
void Socket::connect(const string &url, uint16_t port, const onErrCB &con_cb_in,
float timeout_sec, const string &local_ip, uint16_t local_port) {
    weak_ptr<Socket> weak_self = shared_from_this();
    // 因为涉及到异步回调，所以在poller线程中执行确保线程安全 [AUTO-
TRANSLATED:e4b29f5e]
    //Because it involves asynchronous callbacks, execute in the poller thread to
ensure thread safety
    _poller->async([=] {
        if (auto strong_self = weak_self.lock()) {
            strong_self->connect_l(url, port, con_cb_in, timeout_sec, local_ip,
local_port);
        }
    });
}
```

connect是socket连接的对外接口，负责提交连接任务到异步事件处理线程；真正的的异步网络连接的实现逻辑由connect_l函数实现。

工作流程：

1. 获取弱引用：通过 weak_ptr 获取 Socket 实例的弱引用，避免异步任务中访问到已经析构的对象。
2. 提交异步任务：调用 _poller->async，将核心逻辑（connect_l）委托到事件轮询器线程中执行。使用异步事件轮询器，避免多线程竞争问题，提高性能和线程安全性。
3. 唤醒poller线程去执行任务即调用真正的连接逻辑（在connect_l函数中实现）：真正的连接逻辑在 connect_l 中实现，connect 函数只是对外的封装接口。

Connect_l方法：

```
void Socket::connect_l(const string &url, uint16_t port, const onErrCB &con_cb_in,
float timeout_sec, const string &local_ip, uint16_t local_port) {
    // 重置当前socket  [AUTO-TRANSLATED:b38093a6]
    //Reset the current socket
    closeSock();
    weak_ptr<Socket> weak_self = shared_from_this();
    auto con_cb = [con_cb_in, weak_self](const SockException &err) {
        auto strong_self = weak_self.lock();
        if (!strong_self) {
            return;
        }
        strong_self->_async_con_cb = nullptr;
        strong_self->_con_timer = nullptr;
        if (err) {
            strong_self->setSock(nullptr);
        }
        con_cb_in(err);
    };
    auto async_con_cb = std::make_shared<function<void(const SockNum::Ptr &)>>
([weak_self, con_cb](const SockNum::Ptr &sock) {
        auto strong_self = weak_self.lock();
        if (!sock || !strong_self) {
            con_cb(SockException(Err_dns, get_uv_errmsg(true)));
            return;
        }
        // 监听该socket是否可写，可写表明已经连接服务器成功  [AUTO-
TRANSLATED:e9809ee3]
        //Listen for whether the socket is writable, writable indicates that the
connection to the server is successful
        int result = strong_self->_poller->addEvent(sock->rawFd(),
EventPoller::Event_Write | EventPoller::Event_Error, [weak_self, sock, con_cb](int
event) {
            if (auto strong_self = weak_self.lock()) {
                strong_self->onConnected(sock, con_cb);
            }
        });
        if (result == -1) {
            con_cb(SockException(Err_other, std::string("add event to poller failed
when start connect:") + get_uv_errmsg()));
        } else {
            // 先创建SockFD对象，防止SockNum由于未执行delEvent无法析构  [AUTO-
TRANSLATED:99d4e610]
            //First create the SockFD object to prevent SockNum from being destructed
due to not executing delEvent
            strong_self->setSock(sock);
        }
    });
    // 连接超时定时器  [AUTO-TRANSLATED:1f4471b2]
    //Connection timeout timer
    _con_timer = std::make_shared<Timer>(timeout_sec,[weak_self, con_cb]() {
```

```
        con_cb(SockException(Err_timeout, uv_strerror(UV_ETIMEDOUT)));
        return false;
    }, _poller);
    if (isIP(url.data())) {
        auto fd = SockUtil::connect(url.data(), port, true, local_ip.data(),
local_port);
        (*async_con_cb)(fd == -1 ? nullptr : std::make_shared<SockNum>(fd,
SockNum::Sock_TCP));
    } else {
        auto poller = _poller;
        weak_ptr<function<void(const SockNum::Ptr &)>> weak_task = async_con_cb;
        WorkThreadPool::Instance().getExecutor()->async([url, port, local_ip,
local_port, weak_task, poller]() {
            // 阻塞式dns解析放在后台线程执行 [AUTO-TRANSLATED:e54694ea]
            //Blocking DNS resolution is executed in the background thread
            int fd = SockUtil::connect(url.data(), port, true, local_ip.data(),
local_port);
            auto sock = fd == -1 ? nullptr : std::make_shared<SockNum>(fd,
SockNum::Sock_TCP);
            poller->async([sock, weak_task]() {
                if (auto strong_task = weak_task.lock()) {
                    (*strong_task)(sock);
                }
            });
        });
        _async_con_cb = async_con_cb;
    }
}
```

创建tcp客户端套接字并连接服务器的过程由SockUtil类中的connect方法实现：

```
int SockUtil::connect(const char *host, uint16_t port, bool async, const char
*local_ip, uint16_t local_port) {
    sockaddr_storage addr;
    //优先使用ipv4地址 [AUTO-TRANSLATED:b7857afe]
    //Prefer IPv4 address
    if (!getDomainIP(host, port, addr, AF_INET, SOCK_STREAM, IPPROTO_TCP)) {
        //dns解析失败 [AUTO-TRANSLATED:1d0cd32d]
        //DNS resolution failed
        return -1;
    }
    int sockfd = (int) socket(addr.ss_family, SOCK_STREAM, IPPROTO_TCP);
    if (sockfd < 0) {
        WarnL << "Create socket failed: " << host;
        return -1;
    }
    setReuseable(sockfd);
    setNoSigpipe(sockfd);
    setNoBlocked(sockfd, async);
    setNoDelay(sockfd);
    setSendBuf(sockfd);
    setRecvBuf(sockfd);
    setCloseWait(sockfd);
    setCloExec(sockfd);
    if (bind_sock(sockfd, local_ip, local_port, addr.ss_family) == -1) {
        close(sockfd);
        return -1;
    }
    if (::connect(sockfd, (sockaddr *) &addr, get_sock_len((sockaddr *)&addr)) == 0)
{
        //同步连接成功 [AUTO-TRANSLATED:da143548]
        //Synchronous connection successful
        return sockfd;
```

```
    }
    if (async && get_uv_error(true) == UV_EAGAIN) {
        //异步连接成功 [AUTO-TRANSLATED:44ac1cad]
        //Asynchronous connection successful
        return sockfd;
    }
    WarnL << "Connect socket to " << host << " " << port << " failed: "
<< get_uv_errmsg(true);
    close(sockfd);
    return -1;
}
```

::connect是一个用于连接到远程主机（服务器）的系统调用，定义在POSIX标准中。
接收三个参数：

sockfd，套接字文件描述符，之前通过socket()函数创建，表示一个打开的网络套接字；

addr：指向struct sockaddr的指针，包含目标主机的地址和端口；实际的地址类型可能是sockaddr_in(IPV4)或sockaddr_in6(IPV6)，但通过强制转换为sockaddr类型。

Addrlen：地址结构的大小

在事先通过一系列set--函数配置了套接字的相关选项，其中就有setNoBlocked函数设置该套接字是否是非阻塞的。
如果是非阻塞的，connect会立即返回。如果连接并未立即完成会同时设置errno为EINPROGRESS.

不论是同步还是异步连接成功，都会返回相应的套接字文件描述符。

```
    auto fd = SockUtil::connect(url.data(), port, true, local_ip.data(),
    local_port);
    (*async_con_cb)(fd == -1 ? nullptr : std::make_shared<SockNum>(fd,
SockNum::Sock_TCP));
```

返回后会调用lamed函数async_con_cb，在这个函数内部会去监听该socket是否可写（使用poller线程异步监听该socket上的可写事件是否发生）。如果发生会在poller线程中执行回调函数，也就是addEvent函数中的第三个参数，此处是一个lamed函数。

```
int result = strong_self->_poller->addEvent(sock->rawFd(), EventPoller::Event_Write |
EventPoller::Event_Error, [weak_self, sock, con_cb](int event) {
            if (auto strong_self = weak_self.lock()) {
                strong_self->onConnected(sock, con_cb);
            }
        });
```

也就是说：连接成功后会执行回调调用onConnected函数（当然连接失败也会执行）：

```
void Socket::onConnected(const SockNum::Ptr &sock, const onErrCB &cb) {
    auto err = getSockErr(sock->rawFd(), false);
    if (err) {
        // 连接失败 [AUTO-TRANSLATED:50e99e6b]
        //Connection failed
        cb(err);
        return;
    }
    // 更新地址信息 [AUTO-TRANSLATED:bacb739d]
    //Update address information
    setSock(sock);
    // 先删除之前的可写事件监听 [AUTO-TRANSLATED:ca424913]
    //First delete the previous writable event listener
    _poller->delEvent(sock->rawFd(), [sock](bool) {});
    if (!attachEvent(sock)) {
        // 连接失败 [AUTO-TRANSLATED:50e99e6b]
```

```
        //Connection failed
        cb(SockException(Err_other, "add event to poller failed when connected"));
        return;
    }
    {
        LOCK_GUARD(_mtx_sock_fd);
        if (_sock_fd) {
            _sock_fd->setConnected();
        }
    }
    // 连接成功 [AUTO-TRANSLATED:7db0fbc4]
    //Connection successful
    cb(err);
}
```

连接成功时主要进行：

（1）首先删除之前的可写监听事件，异步执行连接任务时通过判断socket是否可写来确定连接是否成功，如果成功了则为了避免多余的资源占用，所以需要事先删除这个可写的事件监听任务。

（2）其次通过attachEvent函数将套接字重新加入事件监听中：事件发生时此时便可以通过监听socket实现往socket写或从socket上读的操作，由此控制与服务器端的通信。

（3）最后连接成功调用回调函数：通过回调函数通知调用者连接完成。

关键点：**事件删除与重新绑定，多路复用与回调**

attachEvent函数的定义如下：主要完成工作是将监听事件与事件发生后处理事件的回调函数加入到监听事件表中。

```
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;
}
```