ZLToolKit笔记11：定时器

2025年1月3日　　19:52

One loop per poller pthread.

每个线程的轮询挂载点就是runLoop函数。

```
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();
                }
            }
        }

    } else {
```

```
        _loop_thread = new thread(&EventPoller::runLoop, this, true, ref_self);
        _sem_run_started.wait();
    }
```

本节重点关注考虑定时任务是如何管理、触发以及执行调用的。

一 定时器管理与设计

（1）Timer的设计

```
class Timer {
public:
    using Ptr = std::shared_ptr<Timer>;
    /**
     * 构造定时器
     * @param second 定时器重复秒数
     * @param cb 定时器任务，返回true表示重复下次任务，否则不重复，如果任务中抛异常，则默认重复下次任务
     * @param poller EventPoller对象，可以为nullptr
     * Constructs a timer
     * @param second Timer repeat interval in seconds
     * @param cb Timer task, returns true to repeat the next task, otherwise does not repeat. If an exception is thrown in the task, it defaults to repeating the next task
     * @param poller EventPoller object, can be nullptr

     * [AUTO-TRANSLATED:7dc94698]
     */
    Timer(float second, const std::function<bool()> &cb, const EventPoller::Ptr &poller);
    ~Timer();
private:
    std::weak_ptr<EventPoller::DelayTask> _tag;
    //定时器保持EventPoller的强引用 [AUTO-TRANSLATED:d171cd2f]
    //Timer keeps a strong reference to EventPoller
    EventPoller::Ptr _poller;
};
```

其中构造函数与析构函数定义如下：

```
Timer::Timer(float second, const std::function<bool()> &cb, const EventPoller::Ptr &poller) {
    _poller = poller;
    if (!_poller) {
        _poller = EventPollerPool::Instance().getPoller();
    }
    _tag = _poller->doDelayTask((uint64_t) (second * 1000), [cb, second]() {
        try {
            if (cb()) {
                //重复的任务 [AUTO-TRANSLATED:2d440b54]
                //Recurring task
                return (uint64_t) (1000 * second);
            }
            //该任务不再重复 [AUTO-TRANSLATED:4249fc53]
            //This task no longer recurs
```

```
            return (uint64_t) 0;
        } catch (std::exception &ex) {
            ErrorL << "Exception occurred when do timer task: " << ex.what();
            return (uint64_t) (1000 * second);
        }
    });
}
Timer::~Timer() {
    auto tag = _tag.lock();
    if (tag) {
        tag->cancel();
    }
}
```

a. 定时器任务的监听与执行需要绑定相应线程：

```
_poller = poller;
    if (!_poller) {
        _poller = EventPollerPool::Instance().getPoller();
    }
```

b. 然后将这个定时任务加入到_poller线程所管理的延迟执行任务队列中：

```
EventPoller::DelayTask::Ptr EventPoller::doDelayTask(uint64_t delay_ms,
function<uint64_t()> task) {
    DelayTask::Ptr ret = std::make_shared<DelayTask>(std::move(task));
    auto time_line = getCurrentMillisecond() + delay_ms;
    async_first([time_line, ret, this]() {
        //异步执行的目的是刷新select或epoll的休眠时间 [AUTO-
TRANSLATED:a6b5c8d7]
        //The purpose of asynchronous execution is to refresh the sleep time of
select or epoll
        _delay_task_map.emplace(time_line, ret);
    });
    return ret;
}
```

其中_delay_task_map用来管理定时任务：

```
    std::multimap<uint64_t, DelayTask::Ptr> _delay_task_map;
```

它是一个multimap结构的表，底层是一个红黑树。存储的表项是键值对。且是有序的，相同键可以映射不同的值。

async_first函数完成异步执行将这个（_delay_task_map.emplace(time_line, ret)）动作打入到任务队列_list_task中。并通过管道唤醒loop着的Poller线程。因为是打入到_list_task队列头部，所以它最先被执行。

这里问题是：**异步执行这个动作仅仅是将pair<time_line, ret>（ret是用来管理DelayTask的智能指针）插入到_delay_task_map这个multimap中，并没有执行ret封装的定时任务。那么这个定时任务是在何处执行的呢？？？**

c. 定时任务的执行：

每个Poller线程会挂载在runLoop函数中执行轮询：

（1）首先调用getMinDelay，判断是否有定时任务需要处理，有则全部处理，且返回一个下一个定时任务的还要多久才需要执行minDelay。

```
        minDelay = getMinDelay();
```

（2）其次调用epoll_wait检测内核是否有事件触发，并将执行epoll_wait的超时时间设置为minDelay,如果没有下一个定时任务需要处理，则置-1此时epoll_wait将 一直阻塞直至某个事件发生。

```
int ret = epoll_wait(_event_fd, events, EPOLL_SIZE, minDelay ? minDelay : -1);
```

(2)定时任务管理与执行

```
uint64_t EventPoller::getMinDelay() {
    auto it = _delay_task_map.begin();
    if (it == _delay_task_map.end()) {
        //没有剩余的定时器了 [AUTO-TRANSLATED:23b1119e]
        //No remaining timers
        return 0;
    }
    auto now = getCurrentMillisecond();
    if (it->first > now) {
        //所有任务尚未到期 [AUTO-TRANSLATED:8d80eabf]
        //All tasks have not expired
        return it->first - now;
    }
    //执行已到期的任务并刷新休眠延时 [AUTO-TRANSLATED:cd6348b7]
    //Execute expired tasks and refresh sleep delay
    return flushDelayTask(now);
}
```

a. 如果定时任务不存在或者所有任务还尚未到期，则直接返回即可，但要注意返回值前者是0，后者是最近的那个任务还需多久才能执行（时间差）return it->first - now。

b. 到期任务的执行过程封装在flushDelayTask过程中：

```
uint64_t EventPoller::flushDelayTask(uint64_t now_time) {
    decltype(_delay_task_map) task_copy;
    task_copy.swap(_delay_task_map);
    for (auto it = task_copy.begin(); it != task_copy.end() && it->first <= now_time; it = task_copy.erase(it)) {
        //已到期的任务 [AUTO-TRANSLATED:849cdc29]
        //Expired tasks
        try {
            auto next_delay = (*(it->second))();// 此处便为到期任务执行点
            if (next_delay) {
                //可重复任务,更新时间截止线 [AUTO-TRANSLATED:c7746a21]
                //Repeatable tasks, update deadline
                _delay_task_map.emplace(next_delay + now_time, std::move(it->second));
            }
        } catch (std::exception &ex) {
            ErrorL << "Exception occurred when do delay task: " << ex.what();
        }
    }
    task_copy.insert(_delay_task_map.begin(), _delay_task_map.end());
    task_copy.swap(_delay_task_map);
    auto it = _delay_task_map.begin();
    if (it == _delay_task_map.end()) {
        //没有剩余的定时器了 [AUTO-TRANSLATED:23b1119e]
```

```
        //No remaining timers
        return 0;
    }
    //最近一个定时器的执行延时 [AUTO-TRANSLATED:2535621b]
    //Delay in execution of the last timer
    return it->first - now_time;
}
```

这个函数中，主要完成（1）从定时任务队列（multimap维护本质是一个红黑树）取出到期任务挨个执行；（2）执行后删除该定时任务；（3）如果是可重复任务则添加到定时任务队列末尾。

注意：这里使用task_copy的设计很巧妙，解决了以下几个问题：

1. 避免在遍历时修改原始容器

task_copy 作为 _delay_task_map 的副本，用于在遍历时避免直接修改原始容器。原始容器 _delay_task_map 是一个多重映射（std::multimap），并且可能会在遍历过程中发生修改，比如删除任务或插入新任务。如果直接修改 _delay_task_map，会导致迭代器失效，从而引发运行时错误。

通过将任务队列的内容复制到 task_copy 中，我们能够安全地遍历和删除任务，而不影响原始容器。task_copy 可以自由地进行 erase 和其他修改操作，而 _delay_task_map 保持不变，直到 task_copy 完成所有操作并将其内容重新插入 _delay_task_map。

2. 避免在遍历过程中产生竞态条件

在并发环境中，多个线程可能会访问 _delay_task_map，并在不合适的时机对其进行修改。如果我们在遍历过程中修改 _delay_task_map，这可能会导致竞态条件，从而引发潜在的错误。

task_copy 保证了任务的安全性。在任务被复制到 task_copy 后，其他线程仍然可以访问 _delay_task_map，但不会受到本次修改的影响，从而避免了数据竞争和并发问题。

3. 确保任务执行的原子性

在遍历和执行到期任务时，使用 task_copy 保证了所有任务的执行不会影响 _delay_task_map 的结构。即使任务执行时，可能会插入或删除新的任务，但这些修改不会干扰当前的遍历过程。

任务的执行和删除（通过 erase）是原子的。任务执行后，我们可以确定它们已经被从 task_copy 中移除，而不是直接修改 _delay_task_map。这样能保证定时任务的执行顺序和一致性。

4. 支持任务的重复调度

如果某个任务是可重复的（即在执行完后需要重新添加到定时任务队列中），flushDelayTask 会计算任务的下一次执行时间，并将其重新插入 _delay_task_map 中。使用 task_copy 可以保证在遍历已到期任务时，不会因为新任务的插入而打乱当前的遍历逻辑。

任务在执行时被添加到 task_copy，并且可以在执行后根据需要重新插入到 _delay_task_map 的适当位置，这使得可重复任务能够在下一次触发时按时执行。

5. 简化任务管理的逻辑

task_copy 使得整个任务管理过程更加简洁。通过先将 _delay_task_map 的内容复制到 task_copy 中，可以避免在遍历 _delay_task_map 时需要考虑复杂的任务删除、修改和添加逻辑。

遍历 task_copy 中的任务，而不是直接遍历 _delay_task_map，让开发者能够更专注于任

务的执行和调度，而不用担心并发修改和数据一致性问题。

6. 提高性能和可维护性

通过使用 task_copy，可以将任务处理的逻辑从对容器的直接操作中分离出来，使得代码更具可读性和可维护性。

在遍历过程中，任务的删除和添加操作被封装在 task_copy 中，这样每次任务执行和状态更新都不会影响正在执行的任务的处理。只有在整个过程完成后，task_copy 被重新插入到 _delay_task_map，确保了任务队列的一致性。

总结

task_copy 的使用是为了解决在任务调度和执行过程中可能出现的并发修改问题、迭代器失效问题和竞态条件。通过将任务队列的内容复制到 task_copy 中，我们能够安全地修改和遍历任务，确保任务的正确执行，并且不影响正在执行的任务队列的稳定性。它让任务管理变得更安全、更清晰，同时为可重复任务提供了合理的支持。

c. 定时任务的管理——multimap

底层是个红黑树。参见笔记：C++：RB-Tree

二 客户端/服务器端定时器实例

客户端：

```
_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());
```

服务器端：

```
_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);
```

注意这两定时器的回调函数都是返回true，也就是他是个重复任务：

```
    _tag = _poller->doDelayTask((uint64_t) (second * 1000), [cb, second]() {
            try {
                if (cb()) {
                    //重复的任务 [AUTO-TRANSLATED:2d440b54]
                    //Recurring task
                    return (uint64_t) (1000 * second);
                }
                //该任务不再重复 [AUTO-TRANSLATED:4249fc53]
                //This task no longer recurs
                return (uint64_t) 0;
```

```
    } catch (std::exception &ex) {
        ErrorL << "Exception occurred when do timer task: " << ex.what();
        return (uint64_t) (1000 * second);
    }
});
```