cyber/timer 目录下的函数

部分函数功能解释

- `Timer`对象是开放给用户的接口 主要实现了定时器的配置`TimerOption` 启动定时器和关闭定时器 3 个接口
- 定时器的配置 包括定时器周期 回调函数 一次触发还是周期触发(默认为周期触发)

```
    TimerOption(uint32_t period, std::function<void()> callback, bool oneshot) :
period(period), callback(callback), oneshot(oneshot) {}
```

---

timer

- `Timer`对象主要的实现都在`Start()`中
- `Start`中的步骤就是 -> 判断定时器是否以及启动 -> 如果定时没有启动 则初始化定时任务 -> 在时间轮中增加任务

```
void Timer::Start() {

  // 首先判断定时器是否已经启动
  if (!started_.exchange(true)) {
    // 初始化任务
    if (InitTimerTask()) {
      // 在时间轮中增加任务
      timing_wheel_->AddTask(task_);
      AINFO << "start timer [" << task_->timer_id_ << "]";
    }
  }
}
```

- 在`Timer`对象中创建`Task`任务并注册回调`task->callback` 任务回调中首先会调用用户传入的`callback()`函数 然后把新的任务放入到下一个时间轮`bucket`中 对应到代码里就是`TimingWheel::Instance()->AddTask(task)`
- `task->next_fire_duration_ms`是任务的下一次执行的间隔 这个时间间隔是以`task`执行完成之后为起始时间的 因为每次插入新任务到时间轮都是在用户`callback`函数执行之后进行的 因此这里的时间起点也是以这个时间为准
- `task->accumulater_error_ns`是累计时间误差 每次插入任务的时候都会修复这个误差 因此这个误差不会一直增大 也就是说假设你第一次执行的比较早 那么累计误差为负值 下次执行的时间间隔就会变长 如果第一次执行的时间比较晚 那么累计误差为正值 下次执行的时间间隔就会缩短 通过动态的调节 保持绝对的时间执行间隔一致

```
bool Timer::InitTimerTask() {

  // 初始化定时任务
  task_.reset(new TimerTask(timer_id_));
```

```
  task_->interval_ms = timer_opt_.period;
  task_->next_fire_duration_ms = task_->interval_ms;

  // 是否单次触发
  if (timer_opt_.oneshot) {
    std::weak_ptr<TimerTask> task_weak_ptr = task_;
    // 注册任务回调
    task_->callback = [callback = this->timer_opt_.callback, task_weak_ptr]() {
      auto task = task_weak_ptr.lock();
      if (task) {
        std::lock_guard<std::mutex> lg(task->mutex);
        callback();
      }
    };
  } else {
    std::weak_ptr<TimerTask> task_weak_ptr = task_;
    // 注册任务回调
    task_->callback = [callback = this->timer_opt_.callback, task_weak_ptr]() {

      std::lock_guard<std::mutex> lg(task->mutex);
      auto start = Time::MonoTime().ToNanosecond();
      callback();
      auto end = Time::MonoTime().ToNanosecond();
      uint64_t execute_time_ns = end - start;

      if (task->last_execute_time_ns == 0) {
        task->last_execute_time_ns = start;
      } else {
        // start - task->last_execute_time_ns 为2次执行真实间隔时间，task->interval_ms是
设定的间隔时间
        // 注意误差会修复补偿，因此这里用的是累计，2次误差会抵消，保持绝对误差为0
        task->accumulated_error_ns +=
            start - task->last_execute_time_ns - task->interval_ms * 1000000;
      }

      task->last_execute_time_ns = start;
      // 如果执行时间大于任务周期时间，则下一个tick马上执行
      if (execute_time_ms >= task->interval_ms) {
        task->next_fire_duration_ms = TIMER_RESOLUTION_MS;
      } else {
        int64_t accumulated_error_ms = ::llround(
            static_cast<double>(task->accumulated_error_ns) / 1e6);
        if (static_cast<int64_t>(task->interval_ms - execute_time_ms -
                                 TIMER_RESOLUTION_MS) >= accumulated_error_ms) {
          // 这里会补偿误差
          task->next_fire_duration_ms =
              task->interval_ms - execute_time_ms - accumulated_error_ms;
        } else {
          task->next_fire_duration_ms = TIMER_RESOLUTION_MS;
        }

      }
      TimingWheel::Instance()->AddTask(task);
    };
  }
```

```
  return true;
}
```

---

timing_wheel

- `TimingWheel`时间轮的配置如下

```
    512 个 bucket
    64 个 round
    tick 为 2 ms
```

- 每个`bucket`代表`tick`的时间 每一秒走一格 如果我们定义`tick`为一秒 那么`bucket[1]`就代表第 1 秒 而`bucket[8]`就代表第 8 秒
  `round`为 1 就表示需要 1 圈 如果`round`为 2 就表示需要 2 圈
- `TimingWheel`是通过`AddTask`调用执行的
- `Cyber`的时间轮单独采用一个线程调度执行`std::thread([this](){this -> TickFunc();})`
  定时任务则放入协程池中去执行 也就是说主线程单独执行时间计数 而具体的定时任务开多个协程去执行 可以并发执行多个定时任务 定时任务中最好不要引入阻塞的操作 或者执行时间过长
- `Cyber`定时器中引入了 2 级时间轮的方法(消息队列`kafka`也是类似实现) 类似时钟的小时指针和分钟指针 当一级时间轮触发完成之后 再移动到二级时间轮中执行 第二时间轮不能超过一圈 因此定时器的最大定时时间为`64 * 512 * 2 ms` 最大不超过约 65 ms

```
void TimingWheel::AddTask(const std::shared_ptr<TimerTask>& task,
                          const uint64_t current_work_wheel_index) {
  // 不是运行状态则启动时间轮
  if (!running_) {
    // 启动 Tick 线程 并且加入 scheduler 调度
    Start();
  }

  // 计算一下轮 bucket 编号
  auto work_wheel_index = current_work_wheel_index +
                          static_cast<uint64_t>(std::ceil(
                              static_cast<double>(task->next_fire_duration_ms) /
                              TIMER_RESOLUTION_MS));

  // 入果超过最大的 bucket 数
  if (work_wheel_index >= WORK_WHEEL_SIZE) {
    auto real_work_wheel_index = GetWorkWheelIndex(work_wheel_index);
    task->remainder_interval_ms = real_work_wheel_index;
    auto assistant_ticks = work_wheel_index / WORK_WHEEL_SIZE;

    // 转了一圈之后 为什么直接加入剩余的 bucket ???
    if (assistant_ticks == 1 &&
        real_work_wheel_index < current_work_wheel_index_) {
      work_wheel_[real_work_wheel_index].AddTask(task);
      ADEBUG << "add task to work wheel. index :" << real_work_wheel_index;
    } else {
      auto assistant_wheel_index = 0;
      {
```

```
        // 如果超出 则放入上一级时间轮中
        std::lock_guard<std::mutex> lock(current_assistant_wheel_index_mutex_);
        assistant_wheel_index = GetAssistantWheelIndex(
            current_assistant_wheel_index_ + assistant_ticks);
        assistant_wheel_[assistant_wheel_index].AddTask(task);
      }
      ADEBUG << "add task to assistant wheel. index : "
             << assistant_wheel_index;
    }
  } else {
    // 如果没有超过最大的 bucket 数 则增加到对应的 bucket 中
    work_wheel_[work_wheel_index].AddTask(task);
    ADEBUG << "add task [" << task->timer_id_
           << "] to work wheel. index :" << work_wheel_index;
  }
}
```

- 假设二级时间轮中有一个任务的时间周期为 512 那么在当前`bucket`回调中又会在当前`bucket`中增加一个任务 那么这么任务会执行 2 次 如何解决这个问题？
  `Cyber`中采用把这个任务放入上一级时间轮中 然后在触发一个周期之后 放到下一级时间轮中触发

```
void TimingWheel::TickFunc() {
  Rate rate(TIMER_RESOLUTION_MS * 1000000);  // ms to ns

  // 循环使用
  while (running_) {
    // 执行 bucket 中的回调 并且删除当前 bucket 中的任务(回调中会增加新的任务到 bucket)
    Tick();

    tick_count_++;

    // 休眠一个 Tick
    rate.Sleep();
    {
      std::lock_guard<std::mutex> lock(current_work_wheel_index_mutex_);

      // 获取当前 bucket id 每次加一
      current_work_wheel_index_ =
          GetWorkWheelIndex(current_work_wheel_index_ + 1);
    }

    // 下一级时间轮已经转了一圈 上一级时间轮加一
    if (current_work_wheel_index_ == 0) {
      {
        // 上一级时间轮 bucket id 加一
        std::lock_guard<std::mutex> lock(current_assistant_wheel_index_mutex_);
        current_assistant_wheel_index_ =
            GetAssistantWheelIndex(current_assistant_wheel_index_ + 1);
      }
      Cascade(current_assistant_wheel_index_);
    }
  }
```

```
}
```
