定时器设计

- 可由用户注册自定义的回调函数
- 回调函数可被周期性触发或只触发一次
  - `setInterval(func, delay);`
  - `setTimeout(func, delay);`
- 定时器可由用户删除
  - `clearInterval(id);`
  - `clearTimeout(id);`

优先级队列 (priority queue)

- 队列中的每个元素都有一个优先级
- 从队列中弹出元素时，优先级高的元素先出队
- 优先级队列经常使用二叉堆 (binary heap) 实现
  - `std::priority_queue`

二叉堆 (binary heap)

- **二叉堆是基于数组实现的二叉树**
  - 高存储性能
  - 可使用 `std::vector` 存储元素
    - 复用被删除元素的空间
    - 可预先 `reserve()`
  - 快速查找或删除最大（或最小）的元素
  - 快速插入

二叉堆 (binary heap)

- **给定一个二叉堆**
  - `std::vector<unsigned> my_queue;`
    - 假定第一个元素永远为最大的元素
- **查找最大的元素**
  - `my_queue.front()`
    - 时间复杂度为 **O(1)**

二叉堆 (binary heap)

- **插入一个新元素**
  - `my_queue.push_back(42);`
  - `std::push_heap(my_queue.begin(), my_queue.end());`
    - 将最后一个元素插入区间 [begin, end-1) 中
    - 保持第一个元素仍然为最大
    - 时间复杂度为 O(log N)
- **删除最大的元素**
  - `std::pop_heap(my_queue.begin(), my_queue.end());`
    - 将第一个元素移到最后一个位置
    - 然后将 [begin, end-1) 中最大的元素移动到第一个位置
    - 时间复杂度为 O(log N)
  - `my_queue.pop_back();`

定义定时器队列

```
- using Callback = std::function<void (std::uint64_t)>;
- struct Element {
-     std::uint64_t next;      //  定时器下次触发的时间
-     std::uint64_t period;    //  定时器的周期，置零表示只触发一次
-     std::shared_ptr<const Callback> cb_ptr;
- };
- bool operator<(const Element & lhs, const Element & rhs){
-     return lhs.next > rhs.next;
- }
- std::vector<Element> queue;
```

创建定时器

```
- void create_timer(std::uint64_t next, std::uint64_t period, Callback callback){
-     auto cb_ptr = std::make_shared<Callback>(std::move(callback));
-     Element elem = { next, period, std::move(cb_ptr) };
-     queue.emplace_back(std::move(elem));
-     std::push_heap(queue.begin(), queue.end());
- }
```

更新定时器

```
- bool update_a_timer(std::uint64_t now){
-     if(queue.empty() || (now < queue.front().next)) //  无定时器可触发
-         return false;
-     std::pop_heap(queue.begin(), queue.end());
-     const auto cb_ptr = queue.back().cb_ptr;
-     if(queue.back().period == 0) //  非周期性触发
-         queue.pop_back();
-     else { //  周期性触发
-         queue.back().next += queue.back().period;
-         std::push_heap(queue.begin(), queue.end());
-     }
-     (*cb_ptr)(now);
-     return true;
- }
```

删除定时器

- **二叉堆只能删除最大（或最小）的元素**
  - 除非重新建堆，否则不能删除堆中间的元素
- **使用懒惰删除策略**
  - 并不实际删除定时器，仅标记为待删除
  - 更新到一个被标记为待删除的定时器时，删除之

定义定时器队列

```
using Callback = std::function<void (std::uint64_t)>;
struct Element {
    std::uint64_t next;     //  定时器下次触发的时间
    std::uint64_t period;   //  定时器的周期，置零表示只触发一次
    std::weak_ptr<const Callback> cb_wptr;
};
bool operator<(const Element & lhs, const Element & rhs){
    return lhs.next > rhs.next;
}
std::vector<Element> queue;
```

创建定时器并返回带所有权的句柄

```
- std::shared_ptr<const Callback> create_timer(
-     std::uint64_t next, std::uint64_t period, Callback callback)
- {
-     auto cb_ptr = std::make_shared<Callback>(std::move(callback));
-     Element elem = { next, period, cb_ptr };
-     queue.emplace_back(std::move(elem));
-     std::push_heap(queue.begin(), queue.end());
-     return cb_ptr;
- }
```

更新并自动删除失效的定时器

```
- bool update_a_timer(std::uint64_t now){
-     if(queue.empty() || (now < queue.front().next)) //  无定时器可触发
-         return false;
-     std::pop_heap(queue.begin(), queue.end());
-     const auto cb_ptr = queue.back().cb_wptr.lock();
-     if((queue.back().period == 0) || !cb_ptr) //  非周期性触发或已被删除
-         queue.pop_back();
-     else { //  周期性触发
-         queue.back().next += queue.back().period;
-         std::push_heap(queue.begin(), queue.end());
-     }
-     if(cb_ptr)
-         (*cb_ptr)(now);
-     return true;
- }
```

应用举例

```
- class My_class : public std::enable_shared_from_this<My_class> {
- private:
-     std::shared_ptr<const Callback> m_my_timer;
- private:
-     void timer_callback();
- public:
-     void start_timer(std::uint64_t next, std::uint64_t period);
-     void stop_timer() noexcept;
- };
```

应用举例

```
- void My_class::start_timer(std::uint64_t next, std::uint64_t period){
-     auto safety_wrapper = [](const std::weak_ptr<My_class> & wptr){
-         auto sptr = wptr.lock();
-         if(!sptr)
-             return;
-         sptr->timer_callback();
-     };
-     m_my_timer = create_timer(next, period,
-         std::bind(safety_wrapper,
-             // C++17 后可直接使用 weak_from_this()
-             std::weak_ptr<My_class>(shared_from_this())
-         ));
- }
```

应用举例

```
- void My_class::stop_timer() noexcept {
-     m_my_timer.reset();
- }
```

应用举例

```
std::shared_ptr<My_class> my_ptr;
void my_initialize(){
    my_ptr = std::make_shared<My_class>();
    my_ptr->start_timer();
}
// 不需要手动调用 my_ptr->stop_timer()
// 当 my_ptr 被销毁时，由于其成员 m_my_timer 被销毁，定时器被自动删除
```