信号：

#pragma once

#include <mutex>

#include <atomic>

#include <condition\_variable>

#include <chrono>

#include <functional>

#include <future>

using std::mutex;

using std::unique\_lock;

using std::atomic;

using std::condition\_variable;

using namespace std::chrono;

namespace wz {

class **semaphore**

{

public:

**semaphore**() = default;

**semaphore**(int count)

:m\_count(count)

,m\_wakeup(0)

{

}

~**semaphore**()

{

}

/\*!

\* \brief wait 等待

\*/

void **wait**();

using function\_type = std::function<bool ()>;

bool **wait**(int milliseconds, function\_type &&f = nullptr);

#if 0

template<typename **Func**, typename ...**Args**>

void **wait\_for**(**Func** &&**f**, **Args** &&...**args**)

{

using **ftype** = decltype (**f**(**args**...));

auto **newFunc** = **std**::**make\_shared**<**std**::**packaged\_task**<**ftype**()>>(**std**::**bind**(**std**::**forward**<**Func**>(**f**), \

**std**::**forward**<**Args**>(**args**)...));

auto **addFunc** = [**newFunc**]()->bool

{

return (\***newFunc**)();

};

**unique\_lock**<**mutex**> **lck**(**m\_mutex**);

**m\_count**.**fetch\_sub**(1, **std**::**memory\_order\_relaxed**);

if(**m\_count**.**load**(**std**::**memory\_order\_relaxed**) < 0)

{

**m\_con**.**wait**(**lck**, **addFunc**);

}

}

#endif

/\*!

\* \brief signal\_one 唤醒一个

\*/

void **signal\_one**();

/\*!

\* \brief signal\_all 唤醒所有

\*/

void **signal\_all**();

private:

atomic<int> m\_count{0};

atomic<int> m\_wakeup{0};

mutex m\_mutex;

condition\_variable m\_con;

};

}

实现：

#include "semaphore.h"

void wz::semaphore::**wait**()

{

unique\_lock<mutex> lck(*m\_mutex*);

//阻塞

m\_count.fetch\_sub(1, std::memory\_order\_relaxed);

if(m\_count.load(std::memory\_order\_relaxed) < 0)

{

m\_con.wait(*lck*, [this](){return m\_wakeup.load(std::memory\_order\_relaxed) > 0;});

m\_wakeup.fetch\_sub(1, std::memory\_order\_relaxed);

}

}

bool wz::semaphore::**wait**(int milliseconds, wz::semaphore::function\_type &&f)

{

unique\_lock<mutex> lck(*m\_mutex*);

auto msec = std::chrono::milliseconds(milliseconds);

//阻塞

m\_count.fetch\_sub(1, std::memory\_order\_relaxed);

if(m\_count.load(std::memory\_order\_relaxed) < 0)

{

bool b = false;

if(nullptr == f)

{

b = m\_con.wait\_for(*lck*, msec, [this]()

{

return m\_wakeup.load(std::memory\_order\_relaxed) > 0;

});

}

else

{

b = m\_con.wait\_for(*lck*, msec, std::move(*f*));

}

m\_wakeup.fetch\_sub(1, std::memory\_order\_relaxed);

return b;

}

return true;

}

void wz::semaphore::**signal\_one**()

{

unique\_lock<mutex> lck(*m\_mutex*);

m\_count.fetch\_add(1, std::memory\_order\_relaxed);

if(m\_count.load(std::memory\_order\_relaxed) <= 0)

{

m\_wakeup.fetch\_add(1, std::memory\_order\_relaxed);

m\_con.notify\_one();

}

}

void wz::semaphore::**signal\_all**()

{

unique\_lock<mutex> lck(*m\_mutex*);

m\_count.fetch\_add(1, std::memory\_order\_relaxed);

if(m\_count.load(std::memory\_order\_relaxed) <= 0)

{

m\_wakeup.store(0, std::memory\_order\_relaxed);

m\_count.store(0, std::memory\_order\_relaxed);

m\_con.notify\_all();

}

}

定时器：

#pragma once

#include <map>

#include <mutex>

#include <memory>

#include <future>

#include <thread>

#include <functional>

#include "semaphore.h"

namespace **wz**{

namespace **wz\_timer\_cpp11** {

using u32 = unsigned int;

using i32 = int;

using u64 = unsigned long long;

using i64 = long long;

using task\_function = std::function<void ()>;

//定时器事件信息

class **task\_unit**

{

public:

explicit **task\_unit**(u32 id, i32 interval = 0, task\_function func = nullptr);

/\*!

\* \brief start 当前任务启动线程

\*/

void **start**();

/\*!

\* \brief stop 任务释放

\*/

void **stop**();

private:

/\*!

\* \brief run 线程运行函数

\*/

void **run**();

**task\_unit**(const task\_unit &r) = delete;

task\_unit &operator=(const task\_unit &r)= delete;

private:

u32 m\_id{0}; //任务id（唯一）

i32 m\_interval{0}; //定时器间隔时间(ms)

task\_function m\_func{nullptr}; //任务执行函数

std::thread m\_thread; //线程

zhb::semaphore m\_semaphore; //信号量，控制任务执行的时间

std::atomic<bool> m\_run\_flag{true}; //线程运行标识

};

/\*!

\* \brief The timer\_cpp11 class 定时器控制类

\*/

class **timer\_cpp11**

{

public:

**timer\_cpp11**() = default;

~**timer\_cpp11**();

/\*!

\* \brief start\_timer 添加计数器任务

\* \param id 任务id

\* \param interval\_ms 间隔时间，毫秒

\* \param f

\* \param args

\* \return

\*/

template<class F, class ...ARGS>

bool **start\_timer**(u32 id, i32 interval\_ms, F &&f, ARGS &&...args)

{

std::lock\_guard<std::mutex> lck(*m\_task\_mutex*);

//check if exist

if(m\_tasks.find(id) != m\_tasks.end())

{

return false;

}

//封装函数

using function\_type = decltype(f(args...));

auto task1 = std::make\_shared<std::packaged\_task<function\_type()>>(std::bind(std::forward<F>(f), \

std::forward<ARGS>(args)...));

auto task = [task1]()->void

{

(\*task1)();

(\*task1).reset(); //重置std::future的状态

};

std::shared\_ptr<task\_unit> ptask(new task\_unit(id, interval\_ms, task));

auto ret = m\_tasks.emplace(*id*, *ptask*);

if(true == ret.second)

{

//执行线程

ptask->start();

}

return ret.second;

}

/\*!

\* \brief delete\_timer 删除定时器任务

\* \param id 对应的定时器id

\*/

void **delete\_timer**(u32 id);

/\*!

\* \brief delete\_all 清空定时器

\*/

void **delete\_all**();

private:

std::mutex m\_task\_mutex;

std::map<u32, std::shared\_ptr<task\_unit>> m\_tasks;

};

/\*!线程管理\*/

class **timer\_thread**

{

private:

std::thread m\_thread;

};

}

using timer = zhb\_timer\_cpp11::timer\_cpp11;

}

实现：

#include "timer\_cpp11.h"

zhb::zhb\_timer\_cpp11::timer\_cpp11::~**timer\_cpp11**()

{

delete\_all();

}

void wz::wz\_timer\_cpp11::timer\_cpp11::**delete\_timer**(wz::wz\_timer\_cpp11::u32 id)

{

std::lock\_guard<std::mutex> lck(*m\_task\_mutex*);

auto it = m\_tasks.find(id);

if(it != m\_tasks.end())

{

std::shared\_ptr<task\_unit> ptask{it->second};

m\_tasks.erase(it);

//异步释放

std::thread([ptask]()

{

ptask->stop();

}).detach();

}

}

void wz::wz\_timer\_cpp11::timer\_cpp11::**delete\_all**()

{

std::lock\_guard<std::mutex> lck(*m\_task\_mutex*);

for(auto &m : m\_tasks)

{

std::shared\_ptr<task\_unit> ptask{m.second};

//异步释放

std::thread([ptask]()

{

ptask->stop();

}).detach();

}

m\_tasks.clear();

}

wz::wz\_timer\_cpp11::task\_unit::**task\_unit**(wz::wz\_timer\_cpp11::u32 id, wz::wz\_timer\_cpp11::i32 interval, wz::wz\_timer\_cpp11::task\_function func)

:m\_id(id)

,m\_interval(interval)

,m\_func(func)

,m\_run\_flag(true)

{

//开启线程

//start();

}

void wz::wz\_timer\_cpp11::task\_unit::**start**()

{

m\_thread = std::thread(&task\_unit::run, this);

}

void wz::wz\_timer\_cpp11::task\_unit::**stop**()

{

m\_run\_flag.store(false, std::memory\_order\_relaxed);

m\_semaphore.signal\_one();

if(m\_thread.joinable())

{

m\_thread.join();

}

printf\_s("timer job stoped.\n");

}

void wz::wz\_timer\_cpp11::task\_unit::**run**()

{

while(m\_run\_flag.load(std::memory\_order\_relaxed))

{

bool b = m\_semaphore.wait(m\_interval, [this]()

{

return !m\_run\_flag.load(std::memory\_order\_relaxed);

});

if(b)

{

return;

}

m\_func();

}

}

wz::timer time;

m\_timer.start\_timer(0,15,std::bind(&QEngineController::timerEvent,this,0));

Timer,delete...