群名称:C++ 老夏课堂
群　号:296249312

购课学员进群后联系课程客服加课程群

C++11 14 17 20 多线程从原理到线程池实战

1 多线程概述

为什么要用多线程?

任务分解

> 耗时的操作，任务分解，实时响应

数据分解

> 充分利用多核 CPU 处理数据

数据流分解

> 读写分离，解耦合设计

2 C++ 11 多线程快速入门

2.1 先动手写第一个 c++11 多线程程序

```
#include <iostream>
#include <thread>
using namespace std;
using namespace chrono;
using namespace this_thread;

//线程入口函数
void ThreadMain()
{
    cout << " 进入子线程" << get_id() << endl;
    for (int i = 0; i < 10; i++)
    {
        cout << "in thread id = " << i << endl;
        sleep_for(seconds(1)); //线程释放cpu资源1秒
    }
    cout <<get_id()<< " 线程退出" << endl;
}

//主线程入口
int main(int argc, char* argv[])
{
    cout << "主线程ID "<< get_id()  << endl;
    //1 创建并启动子线程，进入线程函数
    thread th(ThreadMain);

    cout << "等待子线程退出" << endl;
    th.join();
    cout << "等待子线程返回" << endl;
    return 0;
}
```

2.2 std::thread 对象生命周期和线程等待和分离

C++11 std::thread 源码

```
// C++11 std::thread源码
class thread
{
//.....
thread() noexcept : _Thr{} {}
explicit thread(_Fn&& _Fx, _Args&&... _Ax) {
//.....
_Thr._Hnd =
            reinterpret_cast<void*>(_CSTD _beginthreadex(nullptr, 0, _Invoker_proc,
_Decay_copied.get(), 0, &_Thr._Id));
}

~thread() noexcept {
        if (joinable()) {
            _STD terminate();
        }
    }
};
```

测试生命周期代码

```
//线程入口函数
bool is_exit = false;
void ThreadMain()
{
    while(!is_exit)
    {
        cout<<"ThreadMain"<<endl;
        this_thread::sleep_for(100ms);
    }
}

//主线程入口
int main(int argc, char* argv[])
{
    //1 创建并启动子线程，进入线程函数
    {
        thread th(ThreadMain);
    }
    //出错 线程句柄清理，线程还在运行
    {
        thread th(ThreadMain);
                    //detach是让目标线程成为守护线程（daemon threads）
        th.detach(); //主线程与子线程分离，潜在问题，主线程退出后子线程未退出

    }
```

```
    {
        thread th(ThreadMain);
        th.join(); //等待子线程，主线程阻塞
    }


}
```

2.3 C++11 线程创建的多种方式和参数传递

2.3.1 普通全局函数作为线程入口

如何传递参数

- C++11 内部如何实现参数传递
- c++11 thread 源码分析

```
class thread { // class for observing and managing threads
public:
//....
    template <class _Fn, class... _Args, enable_if_t<!is_same_v<_Remove_cvref_t<_Fn>,
thread>, int> = 0>
    explicit thread(_Fn&& _Fx, _Args&&... _Ax) {
        using _Tuple                = tuple<decay_t<_Fn>, decay_t<_Args>...>;
        auto _Decay_copied          = _STD make_unique<_Tuple>(_STD forward<_Fn>(_Fx),
_STD forward<_Args>(_Ax)...);
        constexpr auto _Invoker_proc = _Get_invoke<_Tuple>(make_index_sequence<1 +
sizeof...(_Args)>{});
        _Thr._Hnd =
            reinterpret_cast<void*>(_CSTD _beginthreadex(nullptr, 0, _Invoker_proc,
_Decay_copied.get(), 0, &_Thr._Id));
        }
};
```

传递参数示例代码

```
#include <iostream>
#include <thread>
using namespace std;

//线程入口函数
void ThreadMain(int i,string str)
{
    cout << " 进入子线程" << get_id() <<i << str<< endl;
}
//主线程入口
int main(int argc, char* argv[])
{
```

```
    //1 创建并启动子线程，进入线程函数
    //传递的参数都是一份复制
    thread th(ThreadMain,10,"test thread para");
    cout << "等待子线程退出" << endl;
    th.join();
    cout << "等待子线程返回" << endl;
    return 0;
}
```

传递引用变量

```
#include <iostream>
#include <thread>
using namespace std;

struct TPara
{
    string name;
};
//线程入口函数
void ThreadMain(TPara* str)
{
    this_thread::sleep_for(1000ms);
    cout << " 进入子线程" << get_id() << str->name << endl;
}
void ThreadMainRef(TPara& str)
{
    this_thread::sleep_for(1000ms);
    cout << " 进入子线程" << get_id() << str.name << endl;
}
//主线程入口
int main(int argc, char* argv[])
{
    //1 创建并启动子线程，进入线程函数
    thread th;
    TPara a;
    a.name = "test string in A";
    thread th1 = thread(ThreadMain,&a);
    thread th2 = thread(ThreadMain,ref(a));
    cout << "等待子线程退出" << endl;
    th.join();
    cout << "等待子线程返回" << endl;
    return 0;
}
```

参数传递的一些坑

> 传递空间已经销毁

多线程共享访问一块空间

传递的指针变量的生命周期小于线程

```
#include <iostream>
#include <thread>
using namespace std;

struct TPara
{
    string name;
};
//线程入口函数
void ThreadMain(TPara* str)
{
    this_thread::sleep_for(1000ms);
    //str已经销毁
0   cout << " 进入子线程" << get_id() << str->name << endl;
}
//主线程入口
int main(int argc, char* argv[])
{
    //1 创建并启动子线程，进入线程函数
    thread th;
    {
        TPara a;
        a.name = "test string in A";
        th = thread(ThreadMain,&a);

    }
    cout << "等待子线程退出" << endl;
    th.join();
    cout << "等待子线程返回" << endl;
    return 0;
}
```

2.3.2 类成员函数作为线程入口

接口调用和参数传递

```
class MyThread
{
public:
    void Main()
    {
        cout << "MyThread Main " <<name<< endl;
    }
    string name;
};

MyThread myth;
myth.name = "test mythread";
thread th(&MyThread::Main,&myth);
```

线程基类封装

```
class XThread
    {
    public:

        void Start()
        {
            is_exit_ = false;
            th_ = thread(&XThread::Main, this);
        }
        void Wait()
        {
            if (th_.joinable())
                th_.join();
        }
        void Stop()
        {
            is_exit_ = true;
            Wait();
        }
        bool is_exit() { return is_exit_; }

    private:

        virtual void Main() = 0;
        bool is_exit_ = false;
        thread th_;
    };
```

2.3.3 lambda 临时函数作为线程入口函数

> lambda 函数，其基本格式如下
> [捕捉列表] (参数) mutable -> 返回值类型 {函数体}

lambda 线程示例

```
thread t ( []()
    {
    cout << "Hello World from lambda thread." << endl;
    }
);
t.join();
```

lambda 线程传递参数

```
thread t([](int i) {
cout << "Hello World from lambda thread."<<i << endl;
},123);
```

线程入口为类成员 lambda 函数

```
class TestLambda
{
public:
    void Start()
    {
        thread t([this]() {
            cout << "TestLambda thread this."<<name << endl;
            });
        t.join();
    }
    string name = "test lambda";
};
```

2.3.4 call_once 多线程调用函数，但函数只进入一次

```
void SystemInit()
{
    cout << "Call SystemInit" << endl;
}
void CallOnceSystemInit()
{
    static std::once_flag flag;
    std::call_once(flag, SystemInit);
}
```

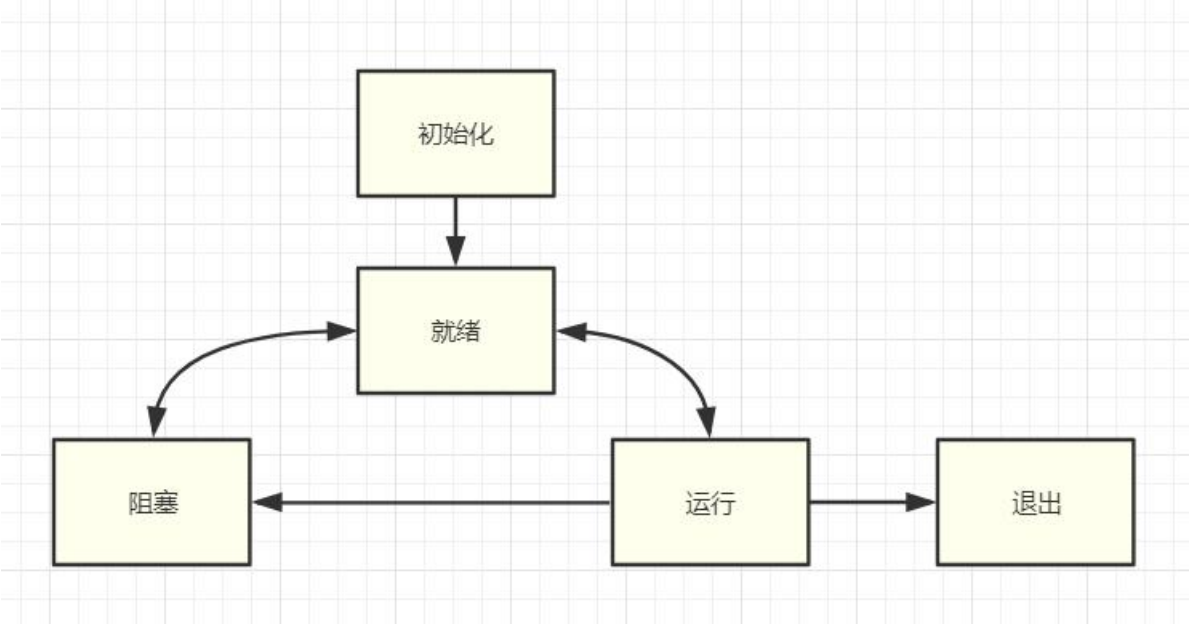
3 多线程通信和同步

3.1 多线程状态

3.1.1 线程状态说明：

- 初始化（Init）：该线程正在被创建。
- 就绪（Ready）：该线程在就绪列表中，等待 CPU 调度。
- 运行（Running）：该线程正在运行。
- 阻塞（Blocked）：该线程被阻塞挂起。Blocked 状态包括：pend(锁、 事件、信号量等阻塞)、suspend（主动 pend）、delay(延时阻塞)、 pendtime(因为锁、事件、信号量时间等超时等待)。

- 退出（Exit）：该线程运行结束，等待父线程回收其控制块资源。



3.1.2 竞争状态（Race Condition）和临界区（Critical Section）

> 竞争状态（Race Condition）

```
多线程同时读写共享数据
```

> 临界区（Critical Section）

```
读写共享数据的代码片段
```

避免竞争状态策略， 对临界区进行保护，同时只能有一个线程进入临界区

3.2 互斥体和锁 mutex

3.2.1 互斥锁 mutex

- 不用锁的情况演示
- 期望输出一整段内容
- lock 和 try_lock()
- unlock()

```
#include <thread>
#incldue <iostream>
using namespace std;
void TestThread()
```

```
{
    cout << "==============================" << endl;
    cout << "test 001" << endl;
    cout << "test 002" << endl;
    cout << "test 003" << endl;
    cout << "==============================\n" << endl;
}
int main(int argc, char* argv[])
{
    for (int i = 0; i < 10; i++)
    {
        thread th(TestThread);
        th.detach();
    }
    getchar();
    return 0;
}
```

> 输出内容

```
================================================================================
====
test 001==============================
================================================================================
==================================

test 001
test 002
==============================


test 001

test 001
test 001
test 002

test 002
test 003
test 002
test 003
test 002
test 003
```

> 加锁之后

```
#incldue <iostream>
#include <thread>
#include <mutex>
using namespace std;
static mutex mux;
void TestThread()
{
    mux.lock();
    cout << "==============================" << endl;
```

```
    cout << "test 001" << endl;
    cout << "test 002" << endl;
    cout << "test 003" << endl;
    cout << "==============================\n" << endl;
    mux.unlock();
}
int main(int argc, char* argv[])
{
    for (int i = 0; i < 10; i++)
    {
        thread th(TestThread);
        th.detach();
    }
    getchar();
    return 0;
}
```

> 程序输出

```
==============================
test 001
test 002
test 003
==============================

==============================
test 001
test 002
test 003
==============================

==============================
test 001
test 002
test 003
==============================
```

3.2.2 互斥锁的坑_线程抢占不到资源

```
mutex mux;
void ThreadMainMux(int i)
{
    for (;;)
    {
        mux.lock();
        cout << i << "[in]" << endl;
        this_thread::sleep_for(1000ms);
        mux.unlock();
    }
}

int main(int argc,char *argv[])
{
```

```
    for (int i = 0; i < 3; i++)
    {
        thread th(ThreadMainMux, i+1);
        th.detach();
    }
//....
}
```

> 输出

```
1[in]
1[in]
1[in]
1[in]
1[in]
1[in]
1[in]
1[in]
```

> 修改 unlock 增加 1 毫秒 sleep，等待内核调度其他线程抢占资源

```
mutex mux;
void ThreadMainMux(int i)
{
    for (;;)
    {
        mux.lock();
        cout << i << "[in]" << endl;
        this_thread::sleep_for(1000ms);
        mux.unlock();
        this_thread::sleep_for(1ms);
    }
}

int main(int argc,char *argv[])
{
    for (int i = 0; i < 3; i++)
    {
        thread th(ThreadMainMux, i+1);
        th.detach();
    }
//....
}
```

> 输出

```
1[in]
2[in]
3[in]
1[in]
2[in]
3[in]
```

```
1[in]
2[in]
```

3.2.3 超时锁应用 timed_mutex（避免长时间死锁）

- 可以记录锁获取情况，多次超时，可以记录日志，获取错误情况

```
timed_mutex tmux;
void ThreadMainTime(int i)
{
    for (;;)
    {

        if (!tmux.try_lock_for(milliseconds(1000)))
        {
            cout << i << "[try_lock_for] timeout" << endl;
            continue;
        }
        cout << i << "[in]" << endl;
        this_thread::sleep_for(2000ms);
        tmux.unlock();
        this_thread::sleep_for(1ms);
    }
}

int main(int argc,char *argv[])
{

    for (int i = 0; i < 3; i++)
    {
        thread th(ThreadMainTime, i);
        th.detach();
    }
// ...
}
```

3.2.4 递归锁(可重入)recursive_mutex 和 recursive_timed_mutex 用于业务组合

- 同一个线程中的同一把锁可以锁多次。避免了一些不必要的死锁
- 组合业务 用到同一个锁

```
recursive_mutex rmux;
void Task1()
{
    rmux.lock();
    // 组合业务 用到同一个锁
    cout << "task1 [in]" << endl;
    rmux.unlock();
}
```

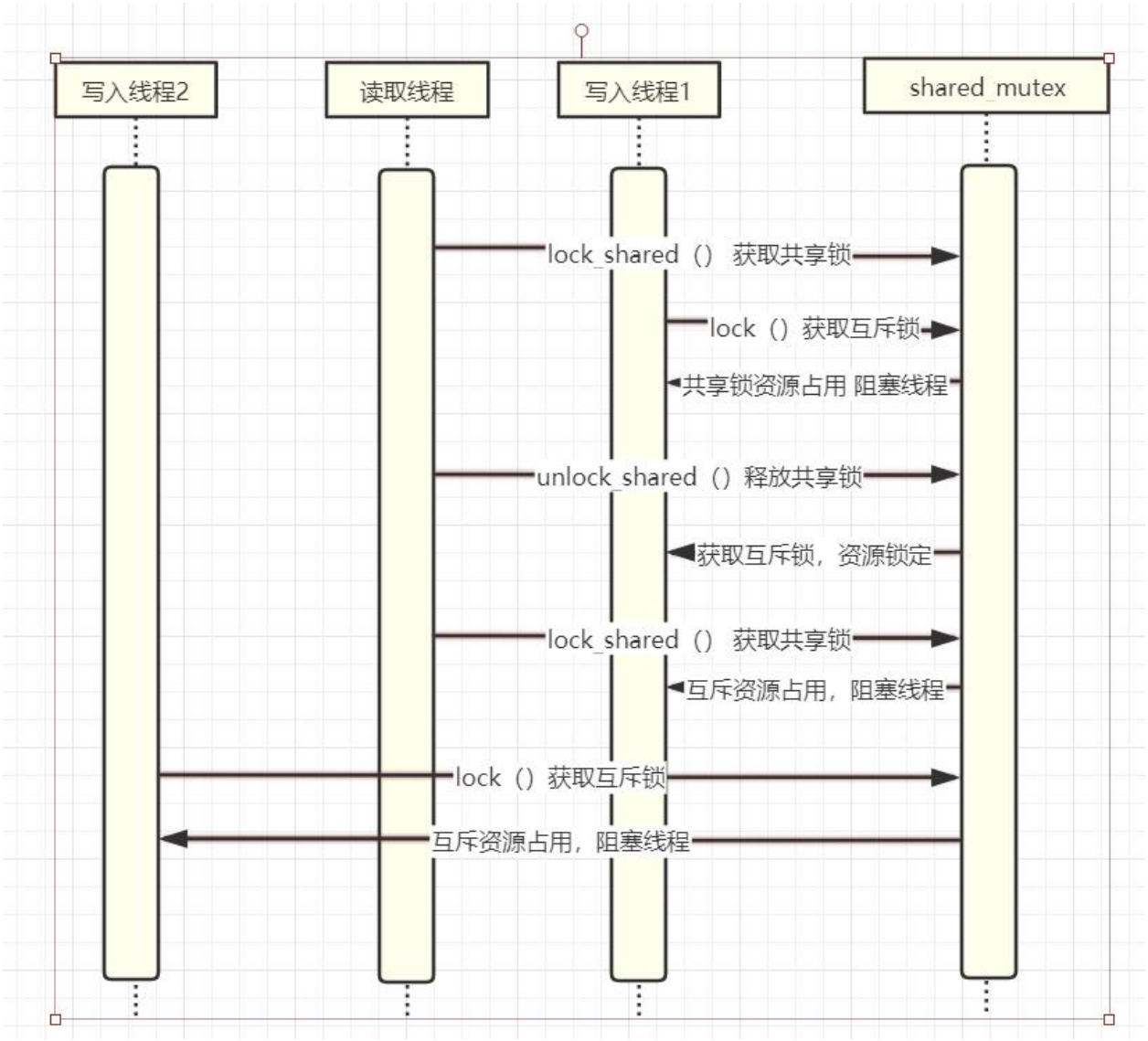
```
void Task2()
{
    rmux.lock();
    // 组合业务 用到同一个锁
    cout << "task2 [in]" << endl;
    rmux.unlock();
}
void ThreadMainRec(int i)
{
    for (;;)
    {
        rmux.lock();
        // 组合业务 用到同一个锁
        Task1();
        Task2();
        cout << i << "[in]" << endl;
        this_thread::sleep_for(2000ms);
        rmux.unlock();
        this_thread::sleep_for(1ms);
    }
}
int main(int argc,char *argv[])
{
    for (int i = 0; i < 3; i++)
    {
        thread th(ThreadMainRec, i);
        th.detach();
    }
    //....
}
```

3.2.5 共享锁 shared_mutex

- c++14 共享超时互斥锁 shared_timed_mutex
- c++17 共享互斥 shared_mutex
- 如果只有写时需要互斥，读取时不需要，用普通的锁的话如何做
- 按照如下代码，读取只能有一个线程进入，在很多业务场景中，没有充分利用 cpu 资源。

```
//读取 同时只能有一个线程读取
mux.lock();
cout<<share<<endl;
mux.unlock();

//写入
mux.lock()
share++;
mux.unlock()
```




```
//c++17
shared_mutex smux;
//c++ 14
shared_timed_mutex stmux;
void ThreadWrite(int i)
{
    for (;;)
    {
        stmux.lock_shared();
        //读数据
        stmux.unlock_shared();

        stmux.lock();
        cout << i << " Write" << endl;
        this_thread::sleep_for(100ms);
        stmux.unlock();

        this_thread::sleep_for(3000ms);
    }
}

void ThreadRead(int i)
{
```

```
    for(;;)
    {
        stmux.lock_shared();
        cout << i << " Read" << endl;
        this_thread::sleep_for(500ms);
        stmux.unlock_shared();
        this_thread::sleep_for(50ms);
    }
}


int main(int argc,char *argv[])
{
    for (int i = 0; i < 3; i++)
    {
        thread th(ThreadWrite, i + 1);
        th.detach();
    }

    this_thread::sleep_for(100ms);

    for (int i = 0; i < 3; i++)
    {
        thread th(ThreadRead, i + 1);
        th.detach();
    }
//...
}
```

3.3 利用栈特性自动释放锁 RAII

3.3.1 什么是RAII，手动代码实现

> RAII（Resource Acquisition Is Initialization）c++之父Bjarne Stroustrup提出;
> 使用局部对象来管理资源的技术称为资源获取即初始化；它的生命周期是由操作系统来管理的，无需人工介入； 资源的销毁容易忘记，造成死锁或内存泄漏。

手动实现RAII管理mutex资源

```
class XMutex
{
public:
    XMutex(mutex& mux):mux_(mux)
    {
        cout << "Lock" << endl;
        mux_.lock();
    }
    ~XMutex()
    {
        cout << "UnLock" << endl;
        mux_.unlock();
    }
```

```
private:
    mutex &mux_;
};
```

3.3.2 c++11支持的RAII管理互斥资源 lock_guard

- C++11 实现严格基于作用域的互斥体所有权包装器
- adopt_lock C++11 类型为adopt_lock_t，假设调用方已拥有互斥的所有权
- 通过{} 控制锁的临界区

```
template <class _Mutex>
class lock_guard { // class with destructor that unlocks a mutex
public:
    using mutex_type = _Mutex;
    explicit lock_guard(_Mutex& _Mtx) : _MyMutex(_Mtx) { // construct and lock
        _MyMutex.lock();
    }

    lock_guard(_Mutex& _Mtx, adopt_lock_t) : _MyMutex(_Mtx) { // construct but don't lock
    }

    ~lock_guard() noexcept {
        _MyMutex.unlock();
    }
    lock_guard(const lock_guard&) = delete;
    lock_guard& operator=(const lock_guard&) = delete;
};
```

3.3.3 unique_lock c++11

- unique_lock C++11 实现可移动的互斥体所有权包装器
- 支持临时释放锁 unlock
- 支持 adopt_lock（已经拥有锁，不加锁，出栈区会释放）
- 支持 defer_lock （延后拥有，不加锁，出栈区不释放）
- 支持 try_to_lock 尝试获得互斥的所有权而不阻塞 ，获取失败退出栈区不会释放，通过owns_lock()函数判断
- 支持超时参数，超时不拥有锁

```
unique_lock& operator=(unique_lock&& _Other) {
    if (this != _STD addressof(_Other))
        { if (_Owns) {
            _Pmtx->unlock();
        }

        _Pmtx        = _Other._Pmtx;
        _Owns        = _Other._Owns;
        _Other._Pmtx = nullptr;
        _Other._Owns = false;
```

```
    }
    return *this;
}
```

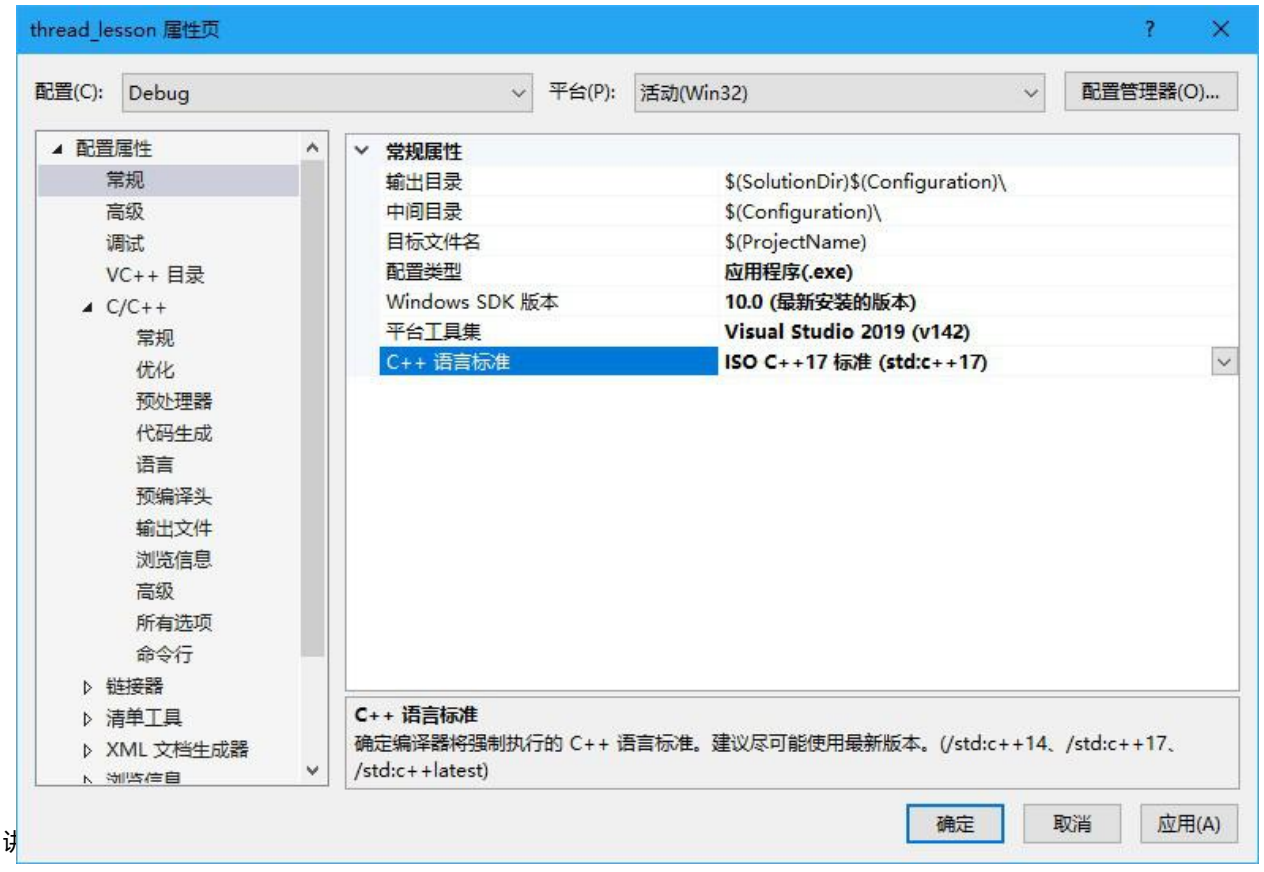
3.3.4 shared_lock C++14

```
shared_lock C++14 实现可移动的共享互斥体所有权封装器
        explicit shared_lock(mutex_type& _Mtx)
            : _Pmtx(_STD addressof(_Mtx)), _Owns(true) { // construct with mutex and lock
shared
            _Mtx.lock_shared();
        }
```

3.3.5 scoped_lock C++17

```
scoped_lock C++17 用于多个互斥体的免死锁 RAII 封装器 类似lock
    explicit scoped_lock(_Mutexes&... _Mtxes) : _MyMutexes(_Mtxes...) { // construct and
lock
        _STD lock(_Mtxes...);
    }

    lock(mux1, mux2);
    mutex mux1, mux2;
    std::scoped_lock lock(mux1, mux2);
```



3.3.5

- 封装线程基类XThread 控制线程启动和停止
- 模拟消息服务器线程 接收字符串消息，并模拟处理
- 通过unique_lock 和 mutex 互斥访问 list<string> 消息队列
- 主线程定时发送消息给子线程

3.4 条件变量

```
#include <iostream>
#include <string>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex m;
std::condition_variable cv;
std::string data;
bool ready = false;
bool processed = false;

void worker_thread()
{
    // 等待直至 main() 发送数据
    std::unique_lock<std::mutex> lk(m);
    cv.wait(lk, []{return ready;});

    // 等待后，我们占有锁。
    std::cout << "Worker thread is processing data\n";
    data += " after processing";

    // 发送数据回 main()
    processed = true;
    std::cout << "Worker thread signals data processing completed\n";

    // 通知前完成手动解锁，以避免等待线程才被唤醒就阻塞（细节见 notify_one ）
    lk.unlock();
    cv.notify_one();
}

int main()
{
    std::thread worker(worker_thread);

    data = "Example data";
    // 发送数据到 worker 线程
    {
        std::lock_guard<std::mutex> lk(m);
        ready = true;
        std::cout << "main() signals data ready for processing\n";
```

```
        }
        cv.notify_one();
```

```
    // 等候 worker
    {
        std::unique_lock<std::mutex> lk(m);
        cv.wait(lk, []{return processed;});
    }
    std::cout << "Back in main(), data = " << data << '\n';

    worker.join();
}
```

3.4.1 condition_variable

生产者-消费者模型

- 生产者和消费者共享资源变量（list队列）
- 生产者生产一个产品，通知消费者消费
- 消费者阻塞等待信号-获取信号后消费产品（取出list队列中数据）

（一）改变共享变量的线程步骤

> 准备好信号量

```
std::condition_variable cv;
```

> 1 获得 std::mutex （常通过 std::unique_lock ）

```
unique_lock lock(mux);
```

> 2 在获取锁时进行修改

```
msgs_.push_back(data);
```

> 3 释放锁并通知读取线程

```
lock.unlock();
cv.notify_one(); //通知一个等待信号线程
cv.notify_all(); //通知所有等待信号线程
```

（二）等待信号读取共享变量的线程步骤

> 1 获得与改变共享变量线程共同的mutex

```
unique_lock lock(mux);
```

> 2 wait() 等待信号通知
>
> > 2.1 无lambada 表达式

```
//解锁lock,并阻塞等待 notify_one notify_all 通知
cv.wait(lock);

//接收到通知会再次获取锁标注. 也就是说如果此时mux资源被占用. wait函数会阻塞
msgs_.front();
//处理数据
msgs_.pop_front();
```

> > 2.2 lambada 表 达 式 cv.wait(lock, [] {return !msgs_.empty();});

只在 std::unique_lock<std::mutex> 上工作的 std::condition_variable

```
    void wait(unique_lock<mutex>& _Lck) { // wait for signal
        // Nothing to do to comply with LWG-2135 because std::mutex lock/unlock are
nothrow
        _Check_C_return(_Cnd_wait(_Mycnd(), _Lck.mutex()->_Mymtx()));
    }
    template <class _Predicate>
    void wait(unique_lock<mutex>& _Lck, _Predicate _Pred) { // wait for signal and test
predicate
        while (!_Pred())
            { wait(_Lck);
        }
    }
```

3.5 线程异步和通信

3.5.1 promise 和 future

- promise 用于异步传输变量

> std::promise 提供存储异步通信的值，再通过其对象创建的std::future异步获得结果。
> std::promise 只能使用一次。 void set_value(_Ty&& _Val) 设置传递值，只能掉用一次

- std::future 提供访问异步操作结果的机制

> get() 阻塞等待promise set_value 的值

- 代码演示

```
void TestFuture(promise<string> p)
{
    ps.set_value("TestFuture value");
}
int main()
```

```
{
    promise<string> p;
    auto fu = p.get_future(); //std::future<string>
    auto re = std::thread(TestFuture,move(p));

    cout << "fu = " << fu.get() << endl;;
    cout << "end std::async" << endl;
    re.join();
    return 0;
}
```

3.5.2 packaged_task 异步调用函数打包

- ackaged_task 包装函数为一个对象，用于异步调用。其返回值能通过std::future 对象访问
- 与bind的区别，可异步调用，函数访问和获取返回值分开调用

```
#include <iostream>
#include <thread>
#include <future>
#include <string>
using namespace std;

string TestPackagedTask(int index)
{
    cout << "begin TestPackagedTask " << index << endl;
    return "TestPackagedTask return";
}

int main()
{
    std::packaged_task<string(int)> task(TestPackagedTask);
    auto result = task.get_future();
    task(100);
    cout << "result.get() = " << result.get() << endl;
    return 0;
}
```

3.5.3 async 创建异步线程

C++11 异步运行一个函数，并返回保有其结果的std::future

- launch::deferred 延迟执行 在调用wait和get时，调用函数代码
- launch::async 创建线程（默认）
- 返回的线程函数的返回值类型的std::future<int> （std::future<线程函数的返回值类型>）
- re.get() 获取结果，会阻塞等待

```
double result = 0;
cout << "Async task with lambda triggered, thread: " << this_thread::get_id() << endl;
//auto f2 = async(launch::async, [&result]() {
```

```
auto f2 = async(launch::deferred, [&result]()
{
    cout << "Lambda task in thread: " << this_thread::get_id() << endl;
    for (int i = 0; i <= 10; i++)
    {
        result += sqrt(i);
    }
}
);
cout << "begin wait()" << endl;
f2.wait();

cout << "Async task with lambda finish, result: " << result << endl << endl;
```

异步操作结果获取 future

```
string testasy(string str)
{
    cout << "test asy " << str << endl;
    this_thread::sleep_for(3s);
    return str+" in async";
}

int main()
{
    //future<string> re =
    auto re = std::async(testasy, "test");
    //auto re = std::async(launch::deferred, testasy, "test");
    cout << "end async" << endl;
    this_thread::sleep_for(1s);
    std::cout << "the future result : " << re.get() << std::endl;
}
```

4 c++17 多核并行计算

1.1 手动实现多核base16编码

1.1.1 实现base16编码

> 二进制转换为字符串
> 一个字节8位 拆分为两个4位字节（最大值16）
> 拆分后的字节映射到 0123456789abcdef

```
static const char base16[] = "0123456789abcdef";
void Base16Encode(const unsigned char *data,int size,unsigned char *out)
{
```

```
    for (int i = 0; i < size; i++)
    {
        unsigned char d = data[i];
        char a = base16[(d >> 4)];
        char b = base16[(d & 0x0F)];
        out[i*2] = a;
        out[i * 2+1] = b;
    }
}
```

1.1.2 c++11实现base16多线程编码

```
bool Base16EncodeMuti(const vector<unsigned char>&data, vector<unsigned char>&out)
{
    int size = data.size();
    int th_count = std::thread::hardware_concurrency();
    int slice_count = size / th_count;
    if (size <= th_count)
    {
        th_count = 1;
        slice_count = size;
    }

     auto d = (unsigned char*)data.data();
     auto o = (unsigned char*)out.data();

     vector<thread> ths;
     ths.resize(th_count);

    for (int i = 0; i < th_count; i++)
    {
        int offset = i * slice_count;
        int count = slice_count;
        if (th_count>1 && i == slice_count - 1)//最后一个
        {
            count = slice_count + size % th_count;
        }
        cout << offset << " " << flush;

        ths[i] = thread(Base16Encode,d + offset, count, o + ((long long)offset * 2));
    }
    for (auto& th : ths)
    {
        th.join();
    }
    return true;
}
```

1.1.3 c++17for_each实现多核并行计算base16编码

```
std::for_each(std::execution::par, //并行计算 多核
    in_data.begin(), in_data.end(),
```

```
    [&](auto& d) //多线程进入此函数
    {
        char a = base16[(d >> 4)];
        char b = base16[(d & 0x0F)];
        int index = &d - idata;
        odata[index * 2] = a;
        odata[index * 2 + 1] = b;
    }
);
```

5 C++11 14 17 线程池实现

5.1 线程池v1.0 基础版本

1 初始化线程池

确定线程数量，并做好互斥访问

2 启动所有线程

std::vector<std::thread*> threads_;

```
    unique_lock<mutex> lock(mutex_);
    for (int i = 0; i < thread_num_; i++)
    {
        auto th = new thread(&XThreadPool::Run, this);
        threads_.push_back(th);
    }
```

3 准备好任务处理基类和插入任务

```
///线程分配的任务类
class XTask
{
public:
    // 执 行 具 体 的 任 务
    virtual int Run() = 0;
};
```

> 存储任务的列表

```
std::list<XTask*> tasks_;
```

> 插入任务，通知线程池处理

```
unique_lock<mutex> lock(mutex_);
tasks_.push_back(task);
condition_.notify_one();
```

4 获取任务接口

> 通过条件变量阻塞等待任务

```
//////////////////////////////////////////////////////////////
///获取任务
XTaskType XThreadPool::GetTask()
{
    unique_lock<mutex> lock(mutex_);
    if (tasks_.empty())
    {
        condition_.wait(lock);//阻塞 等待通知
    }
    if (is_exit_)
        return nullptr;
    if (tasks_.empty())
    {
        return nullptr;
    }
    auto task = tasks_.front();
    tasks_.pop_front();
    return task;
}
```

5 执行任务线程入口函数

```
void XThreadPool::Run()
{
    while(!IsExit())
    {
        //获取任务
        auto task = GetTask();
        if (!task)
            continue;
        try
        {
           task->Run();
        }
        catch (...)
        {
            cerr << "XThreadPool::Run() exception" << endl;
```

```
        }
    }
}
```

5.2 v2.0 智能指针版本

5.3 v3.0 异步获取结果

6 基于线程池实现音视频批量转码工具

ffmpeg工具实现视频转码

ffmpeg.exe -y -i test.mp4 -s 400x300 400.mp4

7 C++20 线程特性

std::barrier 屏障

arrive

到达屏障并减少期待计数

wait

在阶段同步点阻塞，直至运行其阶段完成步骤

arrive_and_wait

到达屏障并把期待计数减少一，然后阻塞直至当前阶段完成

arrive_and_drop

将后继阶段的初始期待计数和当前阶段的期待计数均减少一

```
void TestThread(int i, barrier<>* bar) throw()
{
    this_thread::sleep_for(chrono::seconds(i));
    cout << i<< " begin arrive wait" << endl;
    bar->wait(bar->arrive());
    cout << i<< " end arrive wait" << endl;
}

int main()
{
    int count = 3;
    barrier bar(count);
    for (int i = 0; i < count; i++)
    {
        thread th(Test, i, &bar);
        th.detach();
    }
    getchar();
}
```