（一）线程/进程基本概念

1. 进程

进程是资源调度的基本单位，运行一个可执行程序会创建一个或多个进程，进程就是运行起来的可执行程序

2. 线程

线程是程序执行的基本单位，是轻量级的进程。每个进程中都有唯一的主线程，且只能有一个，主线程和进程是相互依存的关系，主线程结束进程也会结束。

3. 并发（多线程）

多个线程同时执行，由于处理器CPU个数有限，不同任务之间需要分时切换，这种切换是有开销的，操作系统需要保持切换时的各种状态。线程一多大量的时间用于切换，导致程序运行效率低下。

3.1 实现方法

- 多进程实现并发

  进程之间可以通过管道，文件，消息队列，共享内存进程通信

  不同电脑之间通过socket通信

- 单进程中多线程

  一个进程中的所有线程共享地址空间（共享内存），例如：全局变量，指针、引用都可以在线程之间传递，因此使用多线程的开销比较小。

  共享内存引入数据不一致问题，需要使用锁进行解决

3. 2 线程VS进程

- 线程启动速度快，轻量级
- 线程的系统开销小
- 使用有一定难度，需要处理数据一致性问题

（二）C++线程入门

1. 主线程与子线程

- 主线程执行完毕，整个进程也执行完毕，一般情况下，如果其他子线程没有执行完毕，那么这些子线程也会被操作系统强行终止。
- 子线程执行一个函数，函数运行完毕，子线程随之运行完毕。
- 一般情况下，如果想保持子线程的运行状态，需要保持主线程持续运行。

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

void myprint(){
```

```
    cout<< "子线程开始" <<endl;
    //...子线程操作
    cout << "子线程结束" << endl;
}
int main(){
    //执行两个操作，1.新建一个线程，入口为myprint 2. 执行线程
    std::thead td(myprint);
    if(td.joinable()){  //判断线程能否join或detach
       td.join();
    }
    //join()会阻塞线程，在子线程执行完毕之后，在与主线程汇合
    //如果没有join，那么主线程结束而子线程未结束，则会报错
    //td.detach(); //一旦detach则不能再join
    cout << "主线程结束" <<endl;
    return 0;
}
```

- 传统多线程，主线程需要要等待子线程结束之后才能结束。
- join()会阻塞线程，在子线程执行完毕之后，再与主线程汇合
- 新特性，使用detach()方法可以做到主线程和子线程分离，主线程可以先结束，而子线程继续运行。一旦detach()之后，主线程关联的Thread就会与主线程失去关联，子线程会驻留在后台运行，此时子线程由运行时库接管，子线程执行完后，由运行时库（守护线程）清理相关资源。
- 其他创建线程的方法

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

 //使用类对象（可调用对象）创建线程
 class TA{
 public:
     int &m_i; //不建议在线程中使用引用和指针，这样detach时，主线程结束，相关变量被释放，导致子线程中使用不可预知的值
     TA(int &i): m_i(i){
         cout<< "构造函数开始执行" <<endl;
     }
     TA(const TA &ta): m_i(ta.m_i){
         cout<< "拷贝构造函数开始执行" <<endl;
     }
     ~TA(){
         cout<< "析构函数开始执行" <<endl;
     }
     void operator()(){   //不能带参数
          cout<< "子线程开始" <<endl;
          //...子线程操作
          cout << "子线程结束" << endl;
     }
 };

 int main(){
     int myi = 6
     TA ta(myi);
     //执行两个操作，1.新建一个线程，入口为myprint 2. 执行线程
     std::thead td(ta); //自动调用operato
    if(td.joinable()){  //判断线程能否join或detach
        td.join();
    }
```

```
33        //使用lambda表达式创建线程
34        /*
35        auto mylbtd = []{
36             cout<< "子线程开始" <<endl;
37             //...子线程操作
38             cout << "子线程结束" << endl;
39        };
40        std::thead td(mylbtd);
41        td.join();
42        */
43      cout << "主线程结束" <<endl;
44        return 0;
45    }
```

ta在主线程结束之后被立即释放，为什么线程中的类对象还能正常调用类相关函数?

原因在于ta被拷贝了一份到子线程中，因此主线程退出清理ta并不影响子线程中的ta，通过拷贝构造函数可以验证这一过程。

> 源代码：《ThreadLearning01》 **https://github.com/wlonging/ThreadLearning**

（三）线程传参

1. 传递临时对象做为线程参数

```
1   #include <iostream>
2   #include <thread>
3   using namespace std;
4
5   void myprint(const int & i, char *pmybuf){
6       cout << i <<endl;
7       cout << pmybuf <<endl;
8   }
9   int main(){
10      int val = 1;
11      int &qval = val;
12      char mybuf[] = "this is a string";
13      thread td(myprint, val, mybuf);
14      td.detach();
15      cout << "finished" <<endl;
16      return 0;
17  }
```

上述代码中，传入引用没有问题，但是不推荐，因为会创建一个新的引用地址，并不是真正意义上的引用，第二个参数为指针一定会有问题，因为该指针所指向的内容可能在主线程结束时被释放。

```
1   #include <iostream>
2   #include <thread>
3   #include <string>
4   using namespace std;
5
6   void myprint(const int & i, const string &pmybuf){
7       cout << i <<endl;
8       cout << pmybuf <<endl;
9   }
```

```
int main(){
    int val = 1;
    int &qval = val;
    char mybuf[] = "this is a string";
    thread td(myprint, val, mybuf); //这个mybuf可能会在主线程结束后再传入线程，是不安全的，解决方法是将其转为临时对象传入线程
    //thread td(myprint, val, string(mybuf)); //安全的做法
    td.detach();
    cout << "finished" <<endl;
    return 0;
}
```

总结：

1. 若传递基本类型的参数建议直接使用值传递
2. 传递类对象时，**避免隐式类型转换**，应该在创建线程时构建**临时对象**（会在主线程中构建完成），线程参数**使用引用参数来接收实参**（否则对象会创建3次：1次为构建临时对象，2次为临时对象传入触发拷贝对象函数，3次为线程中复制触发拷贝构造函数）
3. 尽量使用join()

2. 线程id

每个线程都有唯一的id。代码中使用std::this_thread::get_id()获取线程id

3. 传入类对象、智能指针

- 在线程中传入类对象时，不论函数形参是不是引用，都会得到一个新的拷贝对象，此时修改该对象并不影响实参。如何解决这个问题？

  使用std::ref()函数，同时使用join()可以使传入的参数始终为同一个对象。
- 智能指针unique_ptr在传入线程时，需要使用std::move()函数将参数传入，此时线程形参接收到的地址和原地址一致，但是主线程中的智能指针为空，需要注意的是必须使用join()，使用detach()可能导致主线程将智能指针所指对象释放而在线程中使用的情况，造成未知错误。

> 源代码：ThreadLearning02

（四）多线程和数据共享

1. 创建并等待多个线程

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

void myprint(int i){
    cout << "Thread start ..." <<endl;
    //....
    cout << "Thread end ..." << endl;
    return;
}

int main(){
```

```
    vector<thread> myThreads; //使用容器存储线程
    //创建10个线程
    for(int i = 0; i<10; i++){
        myThreads.push_back(thread(myprint, i));
    }
    //让主线程等待10个线程运行完成
    for(auto iter = myThreads.begin();iter!=myThreads.end(); ++iter){
        iter->join();
    }
    cout << "Main Thread end..." << endl;
    return 0;
}
```

2. 数据共享

2.1. 只读数据

只读数据是安全稳定的，直接读就行。

2.2 有读有写

需要特殊的处理，避免程序崩溃。写的时候不能读，任意两个线程不能同时写，其他线程不能同时读。

2.3 共享数据代码

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

class A{
public:
    //插入消息，模拟消息不断产生
    void insertMsg(){
        for(int i = 0; i < 10000; i++){
            cout<< "插入一条消息:" << i << endl;
            Msg.push_back(i);  //语句1
        }
    }
    //读取消息
    void readMsg(){
        int curMsg;
        for(int i = 0; i < 10000; i++){
            if(!Msg.empty()){
                //读取消息，读完删除
                curMsg = Msg.front();  //语句2
                Msg.pop_front();
                cout << "消息已读出" << curMsg << endl;
            }else{
                //消息暂时为空
            }
        }
    }
private:
    std::list<int> Msg;  //消息变量
};
```

```
33
34  int main(){
35      A a;
36      //创建一个插入消息线程
37      std::thread insertTd(&A::insertMsg, &a); //这里要传入引用保证是同一个对象
38      //创建一个读取消息线程
39      std::thread readTd(&A::readMsg, &a); //这里要传入引用保证是同一个对象
40      insertTd.join();
41      readTd.join();
42      return 0;
43  }
```

上述代码在执行的过程中有问题，原因在于**语句1**在执行插入操作的时候，**语句2**可能进行读和删除的操作，导致线程运行不稳定。解决方法：引入**互斥量（Mutex）**的概念。

> 源代码：ThreadLearning03

（五）互斥量与死锁

1. 互斥量的基本概念

多个线程同时操作一个数据的时候，需要对数据进行保护，可以使用锁，让其中一个线程进行操作，其他线程处于等待状态。

互斥量可以理解成一把锁，多个线程尝试使用lock()函数对数据进行加锁，只有一个线程能锁定成功，其他线程会不断的尝试去锁数据，直到锁定成功。

互斥量使用需要小心，保护太多影响效率，保护不够会造成错误。

```
1   #include <iostream>
2   #include <vector>
3   #include <list>
4   #include <thread>
5   #include <mutex>  //引入互斥量头文件
6
7   using namespace std;
8
9   class A{
10  public:
11      //插入消息，模拟消息不断产生
12      void insertMsg(){
13          for(int i = 0; i < 10000; i++){
14              cout<< "插入一条消息:" << i << endl;
15              my_mutex.lock();
16              Msg.push_back(i);
17              my_mutex.unlock();
18          }
19      }
20      //读取消息
21      void readMsg() {
22          int MsgCom;
23          for (int i = 0; i < 10000; i++) {
24              if (MsgLULProc(MsgCom)) {
25                  //读出消息了
26                  cout << "消息已读出" << MsgCom << endl;
```

```
            }
            else {
                //消息暂时为空
                cout << "消息为空" << endl;
            }
        }
    }
    //加解锁代码
    bool MsgLULProc(int &command) {
        my_mutex.lock();   //语句1
        if (!Msg.empty()) {
            //读取消息，读完删除
            command = Msg.front();
            Msg.pop_front();
            my_mutex.unlock();   //语句2
            return true;
        }
        my_mutex.unlock();//语句3
        return false;
    }
private:
    std::list<int> Msg;  //消息变量
    std::mutex my_mutex; //互斥量对象
};

int main(){
    A a;
    //创建一个插入消息线程
    std::thread insertTd(&A::insertMsg, &a); //这里要传入引用保证是同一个对象
    //创建一个读取消息线程
    std::thread readTd(&A::readMsg, &a); //这里要传入引用保证是同一个对象
    insertTd.join();
    readTd.join();
    return 0;
}
```

2. 互斥量的用法

互斥量是一个对象。

2.1 lock和unlock

- lock()和unlock()必须成对使用，且只能出现一次，否则代码不稳定。
- 上述代码中，语句1中的lock()和后续语句2和3的unlock()成对出现，缺失任意一个unlock()都会导致程序崩溃。

2.2 std::lock_guard类模板

lock_guard的提出是为了防止程序员在使用lock()的时候忘记unlock()的情况，可以直接取代这两个函数，使用lock_guard之后不能再使用这两个函数。使用方式如下：

```
bool MsgLULProc(int &command) {
    //my_mutex.lock();   //语句1
    std::lock_guard<std::mutex> lgmutex(my_mutex); //使用lock_guard代替lock
    if (!Msg.empty()) {
        //读取消息，读完删除
        command = Msg.front();
        Msg.pop_front();
        //my_mutex.unlock();   //语句2 *使用lock_guard之后不需要自己手动释放锁
        return true;
    }
    //my_mutex.unlock();//语句3 *使用lock_guard之后不需要自己手动释放锁
    return false;
}
```

代码中**语句1**在lock_guard的构造函数中执行，mutex::lock()，在其析构的时候执行mutex::unlock()，由此保证了互斥量的正常使用。

lock_guard缺点是没有lock和unlock使用灵活，需要手动析构。可以使用{}包裹，达到提前析构的目的。见如下代码。

```
//插入消息，模拟消息不断产生
void insertMsg(){
    for(int i = 0; i < 10000; i++){
        cout<< "插入一条消息:" << i << endl;
        //在{}包裹内，lock_guard在{}结束时会自动析构，相当于unlock
        {
            std::lock_guard<std::mutex> lgmutex(my_mutex);
            Msg.push_back(i);
        }
    }
    return;
}
```

3. 死锁

死锁是指两个（多个）线程相互等待对方数据的过程，死锁的产生会导致程序卡死，不解锁程序将永远无法进行下去。

3.1 死锁产生原因

举个例子：两个线程A和B，两个数据1和2。线程A在执行过程中，首先对资源1加锁，然后再去给资源2加锁，但是由于线程的切换，导致线程A没能给资源2加锁。线程切换到B后，线程B先对资源2加锁，然后再去给资源1加锁，由于资源1已经被线程A加锁，因此线程B无法加锁成功，当线程切换为A时，A也无法成功对资源2加锁，由此就造成了线程AB双方相互对一个已加锁资源的等待，死锁产生。

理论上认为死锁产生有以下四个必要条件，缺一不可：

1. **互斥条件**：进程对所需求的资源具有排他性，若有其他进程请求该资源，请求进程只能等待。
2. **不剥夺条件**：进程在所获得的资源未释放前，不能被其他进程强行夺走，只能自己释放。
3. **请求和保持条件**：进程当前所拥有的资源在进程请求其他新资源时，由该进程继续占有。
4. **循环等待条件**：存在一种进程资源循环等待链，链中每个进程已获得的资源同时被链中下一个进程所请求。

3.2 死锁演示

通过代码的形式进行演示，需要两个线程和两个互斥量。

```
#include <iostream>
```

```cpp
#include <vector>
#include <list>
#include <thread>
#include <mutex>  //引入互斥量头文件
using namespace std;

class A {
public:
    //插入消息，模拟消息不断产生
    void insertMsg() {
        for (int i = 0; i < 10000; i++) {
            cout << "插入一条消息:" << i << endl;
            my_mutex1.lock(); //语句1
            my_mutex2.lock(); //语句2
            Msg.push_back(i);
            my_mutex2.unlock();
            my_mutex1.unlock();
        }
    }
    //读取消息
    void readMsg() {
        int MsgCom;
        for (int i = 0; i < 10000; i++) {
            MsgCom = MsgLULProc(i);
            if (MsgLULProc(MsgCom)) {
                //读出消息了
                cout << "消息已读出" << MsgCom << endl;
            }
            else {
                //消息暂时为空
                cout << "消息为空" << endl;
            }
        }
    }
    //加解锁代码
    bool MsgLULProc(int &command) {
        int curMsg;
        my_mutex2.lock();   //语句3
        my_mutex1.lock();   //语句4
        if (!Msg.empty()) {
            //读取消息，读完删除
            command = Msg.front();
            Msg.pop_front();

            my_mutex1.unlock();
            my_mutex2.unlock();
            return true;
        }
        my_mutex1.unlock();
        my_mutex2.unlock();
        return false;
    }
private:
    std::list<int> Msg;  //消息变量
    std::mutex my_mutex1; //互斥量对象1
    std::mutex my_mutex2; //互斥量对象2
};

```

```
int main() {
    A a;
    //创建一个插入消息线程
    std::thread insertTd(&A::insertMsg, &a); //这里要传入引用保证是同一个对象
    //创建一个读取消息线程
    std::thread readTd(&A::readMsg, &a); //这里要传入引用保证是同一个对象
    insertTd.join();
    readTd.join();
    return 0;
}
```

语句1和语句2表示线程A先锁资源1，再锁资源2，语句3和语句4表示线程B线索资源2再锁资源1，具备死锁产生的条件。

3.3 死锁的解决方案

保证上锁的顺序一致。

3.4 std::lock()

功能：锁住两个或两个以上的互斥量，解决因lock()顺序问题导致的死锁问题。

在时间使用过程中，只要有一个互斥量没锁住，就会进行等待，等所有互斥量都做锁住时，程序才继续进行。

要么多个互斥量都锁住，要么都没锁住，只要有一个没锁成功，会立即释放所有已经加锁的互斥量。代码如下：

```
void insertMsg(){
    for(int i = 0; i < 10000; i++){
        cout<< "插入一条消息:" << i << endl;
        std::lock(my_mutex1, my_mutex2);//顺序无所谓
        Msg.push_back(i);
        my_mutex2.unlock();
        my_mutex1.unlock();
    }
}
```

该函数一次能锁定多个互斥量，小心使用，多个互斥量的时候建议逐个lock()和unlock()。

3.5 std::lock_guard的std::adopt_lock参数

std::adopt_lock是一个结构体对象，起一个标记作用就是表示这个互斥量已经lock()，不需要在std::lock_guard<std::mutex>构造函数里对mutex对象进行lock()了。使用这个参数配合std::lock()可以做到无需手动unlock()。代码如下：

```
void insertMsg(){
    for(int i = 0; i < 10000; i++){
        cout<< "插入一条消息:" << i << endl;
        std::lock(my_mutex1, my_mutex2);//顺序无所谓
        //加上adopt_lock参数可以使互斥量不再次进行lock()
        std::lock_guard<std::mutex> lgmutex1(my_mutex1, std::adopt_lock);
        std::lock_guard<std::mutex> lgmutex2(my_mutex2, std::adopt_lock);
        Msg.push_back(i);
    }
}
```

> 源代码：ThreadLearning04

（六）unique_lock的使用

6.1 作用描述

std::unique_lock可以完全取代std::lock_guard，在使用上更加灵活。

6.2 参数说明

- std::adopt_lock：标记作用，如果互斥量已经lock，则不需要再lock;
- std::try_to_lock：尝试去lock，如果没有锁定成功，会立即返回而不会阻塞，注意其之前不能先lock;
- std::defer_lock: 初始化一个未加锁的mutex，其之前也不能先lock，否则会报异常

6.3 成员函数

- lock()：给互斥量加锁;
- unlock：互斥量解锁;
- try_lock()：尝试给互斥量加锁，如果拿不到锁，则返回false，不阻塞;
- release()：释放互斥量的所有权，返回所管理的mutex对象指针，此后unique_lock和mutex不再有关系，如果，mutex处于加锁状态，则负责接管的对象需要负责解锁

6.4 unique_lock转移所有权的方式

- 使用std::move()函数进行转移
- 创建函数返回临时unique_lock对象

> 源代码：ThreadLearning05

（七）单例设计模式与数据共享

7.1 单例设计模式

单例类：指的是在程序中该类的实例只存在一个，其实现通常需要满足以下三个条件：

- 构造函数私有化
- 唯一的私有静态类实例成员变量
- 静态方法返回类实例

实例代码7-1：

```
#include <iostream>
#include <vector>
#include <list>
#include <thread>
#include <mutex>  //引入互斥量头文件
using namespace std;

class MyCAS{
```

```
private:
    MyCAS(){}; //私有化构造函数，保证该类无法被new或者以MyCAS m方式生成实例
private:
    static MyCAS * m_instance; //类实例
public:
    static MyCAS * GetInstance(){ //返回类实例
        if(m_instance == NULL){
            m_instance = new MyCAS();
            static GCclass gc;//用于回收上一句new产生的内存，在程序结束时会调用其析构函数
        }
        return m_instance;
    }
    //类中嵌套回收类，用于回收单例类实例，防止出现内存泄露
    class GCclass{
        ~GCclass(){
            if(MyCAS::m_instance){
                delete MyCAS::m_instance;
                MyCAS::m_instance = NULL;
            }
        }
    };
};

//初始化单例类实例
MyCAS* MyCAS::m_instance = NULL;

int main(){
    MyCAS *p_a = MyCAS::GetInstance(); //获取单例类对象，最好在使用多线程之前加载单例类实例
    return 0;
}
```

7.2 数据共享

在多线程中，如果多个类同时创建单例类对象，需要进行互斥，因此引入互斥量进行加锁。在**代码7-1**中，加入互斥量并修改函数GetInstance():

```
std::mutex my_mutex;  //引入互斥量
static MyCAS * GetInstance(){ //返回类实例
    if(m_instance == NULL){//双重锁定，保证有一次实例化之后，不会再进行资源锁定
        std::unique_lock<std::mutex> myul(my_mutex);
        if(m_instance == NULL){
            m_instance = new MyCAS();
            static GCclass gc;//用于回收上一句new产生的内存，在程序结束时会调用其析构函数
        }
    }
    return m_instance;
}
```

7.3 call_once()函数

该函数的功能是保证在多线程中，某一个函数只能被执行一次，可以解决7.2中GetInstance()函数被多次调用的问题。需要与标记**std::once_flag**配合使用。

在**代码7-1**中，进行如下修改：

```
std::once g_flag;  //引入once_flag标记
//增加函数
static void CreateInstance(){
    m_instance = new MyCAS();
    static GCclass gc;
}
//返回类实例
static MyCAS * GetInstance(){
    std::call_once(g_flag,CreateInstance);
    return m_instance;
}
```

源代码：ThreadLearning06

（八）条件变量及其成员函数

8.1 condition_variable类

条件变量可以使用通知的方式实现线程同步，其履行发送者或者接受者的角色。

实例**代码8-1**：

```
#include <condition_variable> //需要引入头文件
#include <mutex>
#include <thread>
#include <iostream>
using namespace std;

class A{
public:
    //插入消息，模拟消息不断产生
    void insertMsg(){
        for(int i = 0; i < 10000; i++){
            std::unique_lock<std::mutex> myul<my_mutex>; //加锁
            cout<< "插入一条消息:" << i << endl;
            Msg.push_back(i);
            myul.notify_one();     //语句1
        }
    }
    //读取消息
    void readMsg(){
        while(true){
            std::unique_lock<std::mutex> myul<my_mutex>; //加锁
            my_cond.wait(myul,[this]{   //语句2
               if(!Msg.empty())
                   return true;
                return false;
            });

        }

    }
    //加解锁代码
    bool MsgLULProc(int &command){
```

```
33          int curMsg;
34          my_mutex.lock();   //语句1
35          if(!Msg.empty()){
36              //读取消息，读完删除
37              curMsg = Msg.front();
38              Msg.pop_front();
39              cout << "消息已读出" << curMsg << endl;
40              my_mutex.unlock();   //语句2
41              return true;
42          }
43          my_mutex.unlock();//语句3
44          return false;
45      }
46  private:
47      std::list<int> Msg;  //消息变量
48      std::mutex my_mutex; //互斥量对象
49      std::condition_variable my_cond; //条件变量
50  }
51
52  int main(){
53      A a;
54      //创建一个插入消息线程
55      std::thread insertTd(&A::insertMsg, &a); //这里要传入引用保证是同一个对象
56      //创建一个读取消息线程
57      std::thread readTd(&A::readMsg, &a); //这里要传入引用保证是同一个对象
58      insertTd.join();
59      readTd.join();
60      return 0;
61  }
```

8.2 wait()/notify_one()/notify_all()函数

语句2中使用wait()函数进行等待，第一个参数为unique_lock()类对象，第二个参数为lambda表达式，如果是true则直接返回，程序继续往下执行，如果为false，则将互斥量解锁，并阻塞到本行，直到有线程调用**notify_one()**成员函数将其唤醒（如**语句2**）。如果没有第二个参数，则默认为false。

注意：当wait()被唤醒后，会尝试重新拿锁，拿到则程序继续往下执行，notify_one()是唤醒一个处于wait状态的线程，如果有多个线程，则不确定会唤醒哪一个，而notify_all()是唤醒所有处于wait状态的线程。另外一点是，如果在notify的过程中，没有线程处于wait状态，则这个通知会丢失。

> 源代码：ThreadLearning07

（九）异步任务

9.1 async、future创建后台任务

std::async是一个函数模板，用于启动异步任务，返回一个std::future对象，其包含入口函数返回的结果，这个结果并不一定能立即拿到，但是一定会返回，可以调用future的get()函数获取结果。

实例**代码9-1**：

```
1   #include <future>
2   #include <condition_variable> //需要引入头文件
3   #include <mutex>
```

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

//线程函数，模拟异步请求
int mythread(){
    cout << "thread start ... " << endl;
    std::chrono::milliseconds dura(5000);//线程休眠5s
    std::this_thread::sleep_for(dura);
    cout << "thread end ... " << endl;
    return 5;
}

int main(){
    //普通函数调用
    std::future<int> ful = std::async(mythread); //创建异步任务，这段代码不阻塞
    //使用成员函数调用
    /*
    A a;
    int param = 10;
    std::future<int> ful = std::async(&A::mythread, &a, params); //成员函数带参数调用
    */
    cout << ful.get() <<endl; //阻塞，等待mythread()返回
    return 0;
}
```

注意事项：如果调用future的wait()函数，则也会阻塞，但是不会返回结果，此外，get()函数只能被调用一次，多次调用会报异常，原因在于get()函数内部实现的是移动语义，调用结束后相关对象为空。

9.2 async传参

参数类型：std::launch枚举类型。

1. std::launch::deferred

表示线程入口函数调用被延迟到future对象的wait()或get()函数被调用时才执行，其实际上并没有创建新的线程，而是在主线程中调用

> std::future result = std::async(std::launch::deferred , mythread); //创建异步任务，这段代码不阻塞

2. std::launch::async

表示在调用async函数时，就开始创建新的线程

> std::future result = std::async(std::launch::async, mythread); //创建异步任务，这段代码不阻塞

注意：如果async()的第一个参数不填写时，其默认值为：**std::launch::async | std::launch::deferred**，即两种方式都有可能，系统根据实际情况选其一。

9.3 packaged_task

类模板，参数是各种可调用对象，作用是将各种可调用对象打包起来，方便作为线程入口函数的参数，与future配合使用，可以拿到线程入口函数的返回值。

实例**代码9-2**：

```cpp
//修改9-1中的main函数
int main(){
    std::packaged_task<int(int)> mypt(mythread); //语句1
    std::thread t1(std::ref(mypt),1); //语句2
    t1.join();

    /*packaged_task的lambda表达式调用
     std::packaged_task<int(int)> mypt([](int param){
         //函数内容
     });
     mypt(100);
    */
    std::future<int> ful = mypt.get_future(); //future对象里包含了函数的执行结果
    cout << ful.get() <<endl;
    return 0;
}
```

代码分析：**语句1**中定义了packaged_task类对象**mypt**，其中第一个int表示mythread()函数的返回值，第二个int表示函数的形参。**语句2**中使用引用方式传入对象**mypt**，第二个参数为传入函数mythread()的实参。关于packaged_task的lambda表达式调用，其相当于直接调用，并不会创建新的线程。

9.4 promise

std::promise为获取线程函数中的某个值提供便利，在线程函数中给外面传进来的promise赋值，当线程函数执行完成之后就可以通过promise获取该值了，值得注意的是取值是间接的通过promise内部提供的future来获取的。

实例**代码9-3**：

```cpp
int mythread(std::promise<int> &tmpp, int num){
    //此处执行一些复杂操作
    num++;
    num *= 1000;
    int res = num;
    tmpp.set_value(res);
    return 0;
}

int main(){
    std::promise<int> mypm;
    std::thread t1(mythread,std::ref(mypm),180);
    t1.join();
    std::future<int> ful = mypm.get_future(); //绑定promise和future
    cout << ful.get() <<endl;
    return 0;
}
```

> 源代码：ThreadLearning08

(十) future的其他成员函数

10.1 std::future_status

std::future_status属于枚举型变量，包含三个枚举类型，分别是：

- ready：线程执行完毕，成功返回对于的结果
- deferred：线程延迟执行，配合aysnc的第一个参数std::launch::deferred使用
- timeout：线程执行超时，未在规定时间返回结果

示例**代码10-1**：

```
#include <future>
#include <condition_variable> //需要引入头文件
#include <mutex>
#include <thread>
#include <iostream>
#include <chrono>
using namespace std;

//线程函数，模拟异步请求
int mythread(){
    cout << "thread start ... " << endl;
    std::chrono::milliseconds dura(5000);//线程休眠5s
    std::this_thread::sleep_for(dura);
    cout << "thread end ... " << endl;
    return 5;
}

int main(){
    //普通函数调用
    //std::future<int> ful = std::async(std::launch::deferred,mythread); //等待6s,deferred
    std::future<int> ful = std::async(mythread); //创建异步任务，这段代码不阻塞
    std::future_status status = ful.wait_for(std::chrono::seconds(3));//等待3s,ready
    //std::future_status status = ful.wait_for(std::chrono::seconds(1));//等待6s,timeout
    if (status == std::future_status::timeout) {
        //超时
        cout << "线程执行超时" << endl;
    }
    else if (status == std::future_status::deferred) {
        //延迟执行
        cout << "延迟执行" << endl;
    }
    else {
        //执行完毕，返回结果
        cout << "结果是：" << ful.get() << endl;
    }
    return 0;
}
```

10.2 std::share_future

由于future的get()函数是移动语义，只能调用一次，有时候多个线程都需要获取future的返回结果，share_future可以实现数据的复制，可以解决上述问题。

示例**代码10-2**：

```cpp
//修改9-1中的main函数
int main(){
    std::packaged_task<int(int)> mypt(mythread); //语句1
    std::thread t1(std::ref(mypt),1);
    t1.join();
    std::share_future<int> res(std::move(mypt)); //移动语义构造
    //std::share_future<int> res(ful.share()); //share之后，ful为空
    //std::share_future<int> res(mypt.get_future());
    cout << res.get() <<endl;
    return 0;
}
```

10.3 原子操作std::atomic

场景描述：假设有两个线程，一个负责读数据，一个负责写数据，如果不做特殊处理，线程处理结果可能是一个未知值。产生上述问题的原因在于线程的上下文切换过程中，会使一些操作语句被中断，从而不能完整执行。

- 解决方案1：使用互斥量进行加锁
- 解决方案2：使用原子操作，这是一种无锁技术的多线程编程，可以保证程序片段不会被线程切换所打断，其效率高于互斥量，注意：原子操作针对的是一个变量，无法对多行代码进行操作。

std::atomic是一个类模板，可以传递多钟数据类型。

写法如下：

> std::atomic g_num = 0; //定义好之后，其使用与普通的int型变量没有区别

实例**代码10-3**：

```cpp
#include <iostream>
#include <mutex>
#include <future>
#include <list>
using namespace std;

std::atomic<int> g_num = 0;

int mythread(){
    for(int i = 0; i < 10000000; i++){
        g_num++;  //结果正常
        //g_num += 1; //结果正常
        //g_num = g_num +1;  //结果异常
    }
    return 0;
}

int main(){
    std::thread mytd1(mythread);
    std::thread mytd2(mythread);
    mytd1.join();
    mytd2.join();
    cout << "两个线程执行完毕，结果是：" << g_num <<endl;
    return 0;
}
```

> 源代码：ThreadLearning09

（十一）atomic和async深入

11.1 atomic

atomic对于整形变量的操作支持有：++、--、+=、-=，而对其他的操作比如： *g_num = g_num +1* 会得到异常结果，需要注意。

11.2 async

11.2.1 参数详解

- std::launch::deferred

该参数属于延迟调用，并且不创建新的线程，延迟到future对象调用get()或wait()函数时才执行，如果没有调用上面两个参数则不执行。

- std::launch::async

强制创建一个线程执行异步任务，意味着系统必须要创建出新线程执行入口函数。

- std::launch::async | std::launch::deferred（默认值）

这个任务可能创建新线程并立即执行，也可能不创建新线程，直到调用get()或wait()才执行入口函数，二选一，执行哪种策略由系统自行决定。

11.2.2 与thread的区别

- 线程创建过程

  thread()如果系统资源紧张，有可能创建线程失败，导致程序崩溃。

  async()一般称为异步任务，有可能不创建新线程(std::launch::deferred参数)。在系统资源紧张的情况下，async()使用默认参数，就不会创建新的线程，哪个线程调用get()或wait()函数，就在该线程中执行。如果一定要创建线程，必须使用std::launch::async参数。

- 获取返回值

  thread()方式无法直接获取返回值，通常需要放到全局变量中

  async()可以使用future对象的get()方法来获取，容易拿到线程入口函数的返回值

11.2.3 不确定性问题

async()第一个参数为默认值时，会导致线程创建具有不确定性，导致程序出现不可预知的问题。可以使用future_status来判断。

实例**代码11-1**：

```
#include <future>
#include <condition_variable> //需要引入头文件
#include <mutex>
#include <thread>
#include <iostream>
#include <chrono>
using namespace std;

```

```
//线程函数，模拟异步请求
int mythread(){
    cout << "thread start ... " << endl;
    std::chrono::milliseconds dura(3000);//线程休眠3s
    std::this_thread::sleep_for(dura);
    cout << "thread end ... " << endl;
    return 5;
}

int main(){
    //普通函数调用
    //std::future<int> ful = std::async(std::launch::deferred,mythread); //等待6s,deferred
    std::future<int> ful = std::async(mythread); //创建异步任务，这段代码不阻塞
    std::future_status status = ful.wait_for(0s);//等待0s,ready
    //std::future_status status = ful.wait_for(std::chrono::seconds(1));//等待6s,timeout
    if(status == std::future_status::deferred){
        //延迟执行
        cout << "线程执行超时" <<endl;
        cout << ful.get() << endl;
    }else if(status == std::future_status::timeout){
        //超时
        cout << "超时" <<endl;
        cout << ful.get() << endl;
    }else{
        //执行完毕，返回结果
        cout << "结果是：" + ful.get() <<endl;
        cout << ful.get() << endl;
    }
    return 0;
}
```

> 源代码：ThreadLearning10

（十二）临界区和其它互斥量

12.1 Windows临界区

windows临界区与互斥量非常类似，前者只能用于windows编程，后者是跨平台的。

在同一个线程中，对于同一个临界区变量，可以多次进入临界区，离开临界区的次数需要与进入临界区的次数对应，否则会造成临界区未离开的现象。

C++11中，如果是互斥量的话，只能对一段代码加解锁一次，如果连续加锁多次会造成异常。

12.2 自动析构技术

```
//类似于lock_guard<std::mutex>
class CWinLock{ //RAII类（资源获取即初始化）
public:
    CWinLock(CRITICAL_SECTION *m_pCritical){
        m_pCritical = m_pCritical;
        EnterCriticalSection(m_pCritical);
    }
    ~CWinLock(){
        LeaveCriticalSection(m_pCritical);
    }
private:
    CRITICAL_SECTION *m_pCritical;
}
```

代码分析：实例化CWinLock对象时，将临界区变量传入，构造函数中会自动保存临界区变量并进入临界区，在析构时，自动离开临界区。

12.3 recursive_mutex

std::mutex：独占互斥量，只能被lock一次；

std::recursive_mutex：递归独占互斥量，允许同一个线程，同一个互斥量被lock多次。

12.4 超时互斥量

std::timed_mutex：带超时功能的独占互斥量，有两个成员函数

- try_lock_for()：参数是一段时间，在一段时间内拿锁，分拿到拿不到两种情况
- try_lock_until()：参数是未来的一个时间点

std::recursive_timed_mutex：带超时功能的递归独占互斥量，使用方法同std::timed_mutex

实例**代码12-1**：

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

class A {
public:
    //插入消息，模拟消息不断产生
    void insertMsg() {
        for (int i = 0; i < 10000; i++) {
            cout << "插入一条消息:" << i << endl;
            std::chrono::milliseconds limittime(100); //100ms时间
            if (my_mutex.try_lock_for(limittime)) {//如果线程在100ms之内拿到了锁，则执行的操做
            //当前时间点往后推100ms，与上面的功能相同

//if(my_mutex.try_lock_until(std::chrono::steady_clock::now()+limittime)){
                Msg.push_back(i);
                my_mutex.unlock(); //解锁不能忘记
            }
            else {
                //没有拿到锁，可以休眠一段时间
```

```
                std::chrono::milliseconds sleeptime(100); //100ms时间
                std::this_thread::sleep_for(sleeptime);
            }
        }
    }
    //读取消息
    void readMsg() {
        int curMsg;
        for (int i = 0; i < 10000; i++) {
            if (MsgLULProc(curMsg)) {
                //读出消息了
                cout << "消息已读出" << curMsg << endl;
            }
            else {
                //消息暂时为空
                cout << "消息为空" << endl;
            }
        }
    }
    //加解锁代码
    bool MsgLULProc(int &command) {
        my_mutex.lock();   //语句1
        if (!Msg.empty()) {
            //读取消息，读完删除
            command = Msg.front();
            Msg.pop_front();
            my_mutex.unlock();   //语句2
            return true;
        }
        my_mutex.unlock();//语句3
        return false;
    }
private:
    std::list<int> Msg;  //消息变量
    std::timed_mutex my_mutex; //互斥量对象
};

int main() {
    A a;
    //创建一个插入消息线程
    std::thread insertTd(&A::insertMsg, &a); //这里要传入引用保证是同一个对象
    //创建一个读取消息线程
    std::thread readTd(&A::readMsg, &a); //这里要传入引用保证是同一个对象
    insertTd.join();
    readTd.join();
    return 0;
}
```

> 源代码：ThreadLearning11

（十三）补充知识和线程池

13.1 补充知识

1. 虚假唤醒

使用notify_one/all()函数唤醒wait()，可能出现notify_one()多次，而wait()中却只有一份数据，这时只有一个notify起作用，也有可能是使用notify_all()通知多个处于wait()状态的线程去取数据，而数据只有一份，如果不做处理，将造成一些错误。比较严谨的写法如下：

```
//读取消息
void readMsg(){
    while(true){
        std::unique_lock<std::mutex> myul<my_mutex>; //加锁
            my_cond.wait(myul,[this]{   //语句2
            if(!Msg.empty())
                return true;
            return false;
        });
        //数据不为空取数据
    }
}
```

代码分析：wait()中使用第二个参数去判断数据是否为空，如果为空返回false线程继续等待，这样可以避免因为虚假唤醒而造成后续程序的异常。

2. atomic

atomic不支持拷贝复制和拷贝赋值运算符，可以使用load()读atomic的值，store()来写atomic的值，例如：

> std::atomic b = 10;
>
> std::atomic a(b.load()); //以原子方式读数据
>
> a.store(12); //以原子方式写数据

13.2 线程池

场景：人数比较多的网络游戏，不可能给每个玩家都提供一个线程

线程池：把一组线程放在一起，统一管理，循环利用

实现方式：在程序启动时，一次性创建好一定数量放的线程

线程池数量：需要进行测试，得到比较好的效率，过多会因为线程切换造成效率低下，过少则不能充分发挥计算机的资源

注意：网络游戏不建议每个玩家提供一个线程，否则一个线程挂了，整个进程也就挂了，进程中的其他线程也会受到影响

> 源代码：myThreadPool
>
> 仿照这个链接实现： **https://github.com/lzpong/threadpool**

结语

本文主要是根据网易云课堂 **课程链接** ，学习总结而来，包括了课程中的绝大部分代码，代码都经过实际运行测验，全文代码地址： **https://github.com/wlonging/ThreadLearning** ，包含线程相关知识和一个线程池的项目。

如有问题，欢迎大家留言交流。