C++11多线程

一、并发、进程、线程的基本概念和综述

1、并发

两个或多个任务（独立的活动）同时发生（进行），一个程序同时执行多个独立的任务
以往的计算机是单核cpu（中央处理器）：某一时刻只能执行一个任务：由操作系统调度，每秒钟进行多次所谓的“任务切换”
并发的假象（不是真正的并发），这种切换（上下文切换）是要有时间开销的，比如操作系统要保存你切换时的各种状态，执行进度等信息
这些都需要时间，一会切换回来的时候还要复原这些信息

硬件发展，出现了多处理器计算机：用于服务器和高性能计算领域
在一块芯片上有多核cpu
能够实现真正的并行执行多个任务（硬件并发）
使用并发的原因：主要是同时可以执行多个任务，提高性能

2、进程和线程

windows下，双击一个可执行程序运行， Linux下 ./文件名（./a）来运行
进程，就是一个可执行程序运行了，就叫创建了一个进程

a)每个进程，都有一个主线程，主线程是唯一的，也就是一个进程中只能有一个主线程
b)当你执行一个可执行程序，产生进程后，这个主线程就随着这个进程默默地启动起来了
运行程序时，实际上是进程的主线程来执行（调用）这个main函数中的代码。

多线程（并发），并不是越多越好，每个线程都需要一个独立的堆栈空间（1M），线程之间的切换要保存很多中间状态。切换会耗费本该属于程序运行的时间

二、C++11线程启动结束及创建

1、用普通函数创建线程

包含头文件thread
创建一个thread对象，myprint是可调用对象

```
//线程入口函数
void myprint() {
    cout << "我的线程开始执行了" << endl;

    cout << "执行完毕" << endl;
}
```

```
int main(){
    std::thread mytobj(myprint);//(1)创建了线程，线程执行起点myprint(),(2)mypr
int线程开始执行
    mytobj.join();//汇合，阻塞主线程，让主线程等待子线程执行完毕，然后子线程和主线
程汇合，然后主线程再往下走
                //若没有join()，则主线程可能先结束导致子线程还没结束
    mytobj.detach();//使myprint失去控制，不能再使用join()
                    //分离主线程和子线程不汇合了，主线程不必等待子线程执行完毕
                    //为什么引入？：我们创建了很多子线程，让主线程逐个等待子线程
结束，这种编程方法不友好
                    //一旦detach()之后，与主线程关联的thread对象就会失去与主线的
关联，此时这个子线程就会驻留在后台运行（主线程和该子线程失去联系）
                    //这个子线程相当于被c++运行时库接管，当这个子线程执行完毕之
后，由运行时库负责清理该线程相关的资源（守护线程）
    if (mytobj.joinable()) {//判断是否可以成功使用join()或者detach()，返回true
或者false
        mytobj.join();
    }
}
```

2、用类对象创建线程

```
class TA {
public:
    int &m_i;
    TA(int &i) :m_i(i) {
        cout << "构造函数" << endl;
    }
    TA(const TA &ta) :m_i(ta.m_i) {
        cout << "拷贝构造函数" << endl;
    }
    ~TA() {
        cout << "析构函数" << endl;
    }
    void operator()() {//不带参数
        cout << "m_i 1:" << m_i << endl; //一旦m_i回收了，会产生不可预料的后果
（引用变量）
        cout << "m_i 2:" << m_i << endl;
        cout << "m_i 3:" << m_i << endl;
        cout << "m_i 4:" << m_i << endl;
    }
};
```

```
int main(){
    int mi = 6;
    TA ta(mi);
    std::thread myobj3(ta);//ta可调用对象
    myobj3.join(); //等待子线程执行结束
    //myobj3.detach();//一旦调用detach()，主线程执行结束，则这里使用的这个ta对象不在了
                      //这个对象是被复制到了线程中去了，所以执行完主线程后ta被销毁，但是所复制的ta对象依旧存在
                      //所以，只要这个类对象里没有引用，没有指针，那么不会产生问题

}
```

3、使用lambda表达式创建线程

```
    auto mylambda = [] {
        cout << "我的线程4开始" << endl;

        cout << "我的线程4结束" << endl;
    };
    std::thread myobj4(mylambda);
    myobj4.join();
```

三、线程传参以及成员函数做线程函数

1、传递临时变量作为线程参数

a)若传递int这种简单类型参数，建议都是值传递，不要引用传递

b)如果传递类对象，避免隐式类型转换，全部在创建线程时就构建出临时对象来，然后在线程函数参数里用引用来接，否则系统还会构造出一个对象来

事实一：只要用临时构造的A类对象作为参数传递给线程，那么就一定能在主线程执行完毕前把线程函数的第二个参数构建出来，从而确保即便detach()子线程也能安全运行

```
class A {
public :
    int m_i;
    //类型转换构造函数，将int转成A类型
    A(int a):m_i(a) {
        cout << "[A::A(int a)构造函数执行]" << this<< "id:"<< std::this_threa
```

```
d::get_id()<< endl;
    }
    A(const A &a) :m_i(a.m_i) {
        cout << "[A::A(const A &a)拷贝构造函数执行]" << this << "id:" <<
std::this_thread::get_id() << endl;
    }
    ~A() {
        cout << "[A::A()析构函数执行]" << this << "id:" << std::this_thread::
get_id() << endl;
    }

    void thread_work(int num) {
        cout << "[子线程thread_work执行]" << this << "id:" << std::this_threa
d::get_id() << endl;
    }

    void operator()(int num) {
        cout << "[子线程operator执行]" << this << "id:" << std::this_thread::
get_id() << endl;
    }
};
void myprint(A &mybuf) {
    mybuf.m_i = 199;//修改该值不会影响main函数
    cout << "子线程参数地址: " << &mybuf << "id:" << std::this_thread::get_id
() << endl;
}

int main(){
    int mvar = 1;
    int mysecond = 12;
    thread myobj(myprint, mvar, A(mysecond));//希望mysecond整型转成A类型对象传
递给myprint的第二个参数
                                             //在传参之前，用A类型强转（构造临时
对象），可以保证线程中的值是有对象
                                             //隐式类型转换时是在子线程中构造A类
对象
}
```

2、临时对象作为线程参数继续

线程id的概念

id是一个数字，每个线程（不管是主线程还是子线程）实际上都对应这一个数字，每一个线程对应

的数字都不相同
也就是说，不同的线程，它的线程id比不同
线程id可以用c++标准库的函数来获取，**std::this_thread::get_id()**来获取

临时对象构造时机捕获
隐式类型转换时是在子线程中构造A类对象
用了临时对象后，所有的A类对象在main()函数中就已经构建完毕了

3、传递类对象、智能指针作为线程参数

使用std::ref函数，独占式指针必须使用move()

4、用成员函数指针做线程函数

operator()

5、调用类成员函数创建线程

```
    A myA(10);
    //thread myobj(&A::thread_work, myA, 15);//此时会调用拷贝构造
    thread myobj(&A::thread_work, &myA,15);//此时不会调用拷贝构造，使用detach()会出现问题
```

6、调用类的函数对象创新线程

```
    A myA(10);
    thread myobj(myA, 15);
```

四、创建多个线程、数据共享问题

1、创建和等待多个线程

vector mythreads;
创建10个线程，统一使用myprint()
a)多个线程执行顺序是乱的，与操作系统内部对线程的运行调度机制有关。
b)主线程等待所有子线程运行结束最后主线程结束
c)使用容器，把thread对象放入容器中，看起来像thread对象数组，方便管理

```
vector<int> gv = { 1,2,3,4 };//共享数据，只读
线程入口函数
void myprint(int num) {
    //cout << "myprint线程开始: " << num << endl;
```

```
    //cout << "myprint线程结束: " << num << endl;
    cout << "id:" << this_thread::get_id() << "gv的值: " << gv[0] << gv[1] <<
 gv[2] << gv[3] << endl;
    return;
}
int main(){
    for (int i = 0; i < 10; ++i) {
        mythreads.push_back(thread(myprint,i));//创建10个线程并开始执行线程
    }
    for (auto iter = mythreads.begin(); iter != mythreads.end(); ++iter) {
        iter->join();//等待10个线程都返回
    }
}
```

2、数据共享问题

只读的数据：是稳定安全地，不需要特别的处理，直接读取就行

有读有写的数据：2个线程写，8个线程读，如果没有特别处理程序一定崩溃
最简单的不崩溃处理：读的时候不能写，写的时候不能读，两个线程不能同时写，8个线程不能同时读

3、互斥量

场景：网络游戏服务器，有两个自己创建的线程，一个线程收集玩家命令，并把命令数据写到队列中，另一个线程从队列中取出命令，解析，然后执行玩家需要的动作。
使用成员函数作为线程函数的方法来创建线程。

```
class A {
public :
    //把收到的命令放到队列的线程
    void inMsgRecvQueue() {
        for (int i = 0; i < 10000; ++i) {
            cout << "inMsgRecvQueue() Start,insert element" << i << endl;


            {
                //my_mutex1.lock();//实际工程中这两个锁不一定连着。。需要保护不
同的数据共享块
                //my_mutex2.lock();
                std::lock(my_mutex1, my_mutex2);
                std::lock_guard<std::mutex> lguard1(my_mutex1,std::adopt_loc
```

```
k);
                std::lock_guard<std::mutex> lguard2(my_mutex2, std::adopt_lo
ck);
                msgRecvQueue.push_back(i);//假设数字i就是收到的玩家命令，放入到
消息队列
            }
            //my_mutex2.unlock();
            //my_mutex1.unlock();
        }
        return;
    }

    bool outMsg(int &command) {
        //std::lock_guard<std::mutex> lguard(my_mutex); //
        //lock_guard构造函数里执行了lock(),析构函数里执行了unlock()

        //my_mutex1.lock();
        //my_mutex2.lock();
        std::lock(my_mutex1, my_mutex2);
        if (!msgRecvQueue.empty()) {
            //消息队列不为空
            int command = msgRecvQueue.front();
            msgRecvQueue.pop_front();
            my_mutex1.unlock();
            my_mutex2.unlock();
            return true;
        }
        my_mutex1.unlock();
        my_mutex2.unlock();
        return false;
    }

    //把数据从消息队列中取出的并处理的线程
    void outMsgRecvQueue() {
        int command = 0;
        for (int i = 0; i < 10000; ++i) {

            bool result = outMsg(command);

            if (result) {
                cout << "outMsgRecvQueue() , get element" << command <<
endl;
```

```
                //处理命令
                //...
            }
            else {
                cout << "outMsgRecvQueue() start, no element" << endl;
            }
        }
        cout << "end" << endl;
    }
private:
    list<int> msgRecvQueue;//专门用于接收玩家的命令

    std::mutex my_mutex1;//创建一个互斥量
    std::mutex my_mutex2;//创建一个互斥量
};
int main(){
    A myobj;
    std::thread myOutMsgObj(&A::outMsgRecvQueue, &myobj);//第二个参数为引用，保证是同一个对象，不会调用拷贝构造构造函数
    std::thread myInMsgObj(&A::inMsgRecvQueue, &myobj);

    myOutMsgObj.join();
    myInMsgObj.join();
}
```

3.1互斥量的用法

头文件#include

lock(),unlock()：先lock(),操作共享数据，再unlock()，lock()和unlock()要成对使用

为了防止忘记使用unlock(),引入了一个叫std::lock_guard的类模板，可以自动unlock()。

3.2std::lock_guard的std::adopt_lock参数

std::adopt_lock是一个结构体对象，一个标记作用，表示这个互斥量已经lock，不需要在lock_guard构造函数里再对mutex对象lock()了。

3.3std::lock函数模板

一次锁住多个互斥量

3.4unique_lock取代lock_guard

unique_lock是个类模板，工作中，一般lock_guard（推荐使用）

unique_lock比lock_guard灵活，但是效率差一点，内存占用高一些

默认情况和lock_guard一样

3.4.1unique_lock的第二个参数

std::adopt_lock
lock_guard也可以带第二个参数,std::lock_guard lg(my_mutex1,std::adopt_lock);

std::adopt_lock：表示这个互斥量已经被lock了（必须提前lock()）

std::try_to_lock
我们会尝试用mutex的lock去锁住这个mutex，如果没有锁定成功，也会立即放回，并不会阻塞在那里

用这个的前提是不能自己去锁mutex

std::defer_lock
用这个的前提是不能直接先去lock，否则异常

defer_lock的意思是并没有给mutex加锁，初始化了一个没有加锁的mutex，等待后面再加锁

3.4.2unique_lock的成员函数，需要defer_lock参数

lock()函数，加锁

unlock()函数，解锁

try_lock()，尝试给互斥量加锁，如果拿不到锁，则返回false，拿到了返回true，不阻塞

release()，返回他所管理的mutex对象指针，并释放所有权，也就是说这个unique_lock和mutex不再有联系

为什么有时候需要unlock()?因为你lock住的代码段越少，执行越快，整个程序运行效率越高
有人也把锁头锁住的代码的多少 称为锁的 粒度。粒度一般用粗细来描述
a)锁住的代码少，粒度细，执行效率高
b)锁住的代码多，粒度粗，执行效率低
要学会选择合适的粒度进行保护，太细可能导致漏掉关键代码，太粗，则效率太低

3.4.3unique_lock所有权的传递，mutex

std::unique_lock lg(my_mutex1)
lg拥有my_mutex1的所有权
lg可以把自己对my_mutex1的所有权转移给其他的unique_lock对象
所以，unique_lock对象这个mutex的所有权是可以转移，但是不能复制

a)std::move
b)return std::unique_lockstd::mutex

3.5单例设计模式

```
class MyCAS {
    static void CreateInstance() {//只被调用一次的代码
        myinstance = new MyCAS();
        static GC gc;
    }

private:
    MyCAS(){}//私有化构造函数

private:
    static MyCAS *myinstance;//静态成员变量

public:
    static MyCAS *getInstance() {

        //std::unique_lock<std::mutex> mymutex(resurce_mutex);//自动加锁，此时效率低，每次都会加锁
        //if (myinstance == NULL) {
        //  myinstance = new MyCAS();
        //  static GC gc; //静态对象的生命周期到程序退出的时候，当程序退出的时候，会调用这个对象的析构函数
        //}

        //改进，提高效率。
        //a)如果myinstance == NULL不成立，则肯定表示myinstance已经被new过了
        //b)如果myinstance == NULL成立，不代表myinstance一定没有被new过，
        if (myinstance == NULL) {//双重锁定（双重检测）
            std::unique_lock<std::mutex> mymutex(resurce_mutex);//自动加锁
            if (myinstance == NULL) {
                myinstance = new MyCAS();
                static GC gc; //静态对象的生命周期到程序退出的时候，当程序退出的时候，会调用这个对象的析构函数
            }
        }

        return myinstance;
    }
```

```
    class GC {
        //类中套类，用来释放对象
    public:
        ~GC() {
            if (MyCAS::myinstance) {
                delete MyCAS::myinstance;
                MyCAS::myinstance = NULL;
            }
        }
    };


    void func() {
        cout << "test" << endl;
    }
};

//类静态变量初始化
MyCAS *MyCAS::myinstance = NULL;
```

4、条件变量std::condition_variable

线程A：等待一个条件满足

线程B：专门往消息队列中扔消息（数据），然后通知线程A

std::condition_variable实际上是一个类，是一个和条件相关的一个类，就是等待一个条件达成

这个类是需要和互斥量来配合使用，需要生成这个类的对象

```
class A {
public:
    void inMsg() {
        for (int i = 0; i < 10000; ++i) {

            std::unique_lock<std::mutex> lg(my_mutex);
            cout << "inMsg() Start,insert element" << i << endl;
            msgRecvQueue.push_back(i);
            //假如outMsg()正在处理一个事务，需要一段时间，而不是正卡在wait()那里等待唤醒，此时nofity_one()就没有效果
            my_cond.notify_one();//尝试把wait()的线程唤醒，执行完这行，那么outMsg()里面的wait()就会被唤醒
```

```
                                    //唤醒之后:
            //my_cond.notify_all();
        }
        return;
    }

    //改进
    void outMsg() {
        int command = 0;
        while (true) {
            std::unique_lock<std::mutex> lg(my_mutex);
            //wait()用来等一个东西
            //如果第二参数lambda表达式返回值是false，那么wait()将解锁互斥量，并阻塞到本行
            //阻塞到某个线程调用notify_one()成员函数为止
            //如果wait没有第二个参数，那么就跟第二个参数lambda表达式返回false效果一样
            //如果第二个参数lambda表达式返回值是true，那么wait()直接返回，继续往下走

            //当其他线程用notify_one()将这个wait()(原本阻塞)唤醒后:
            //a)wait()不断尝试重新获取互斥量,若获取不到，流程会卡在wait()这里等着获取，若获取到，这wait()往下执行
            //b)获取到锁然后加锁
                //b.1)若wait有第二个参数（lambda）,就判断这个lambda表达式，若表达式为false，那么wait()又对互斥量解锁，然后阻塞到本行
                //b.2)若lambda为true，则wait返回，流程继续往下执行，此时，互斥锁被锁着。
                //b.3)若wait没有第二个参数，则wait返回，流程往下执行
            my_cond.wait(lg, [this] { //一个lambda就是一个可调用对象（函数 ）
                if (!msgRecvQueue.empty()) {
                    return true;
                }
                return false;
            });

            //流程走到这，代表互斥锁一定被锁着，同时可以确定msgRecvQueue至少有一条数据
            command = msgRecvQueue.front();
            msgRecvQueue.pop_front();
            cout << "outMsg() , get one element:" << command <<"  id: " << this_thread::get_id()<< endl;
```

```
            lg.unlock();//unique_lock可以随时解锁，避免锁住太长时间

            //正在处理100ms
        }
    }

private:
    std::mutex my_mutex;
    list<int> msgRecvQueue;

    std::condition_variable my_cond;//生成一个条件变量对象
};

int main(){

    A a;
    std::thread myobj1(&A::outMsg, &a);//第二个参数为引用，才能保证线程里用的是同一个对象
    std::thread myobj11(&A::outMsg, &a);
    //std::thread myobj2(&A::inMsg, &a);
    myobj1.join();
    myobj11.join();
    //myobj2.join();
}
```

5、异步任务

5.1std::async、std::future创建后台任务并返回

std::async是个函数模板，用来启动一个异步任务，启动起来一个异步任务之后，他返回一个std::future对象，future是一个类模板

future对象里含有线程入口函数所返回的结果（线程返回的结果），可以通过调用future对象的成员函数get()来获取结果

future提供了一个访问异步操作结果的机制。这个结果可能没有办法马上拿到，但是在线程执行完毕之后就可以拿到结果了

future(对象)里会保存一个值，在将来的某个时刻拿到

```
class A {
public:
    int mythread(int myvar) {
```

```
        cout << myvar << "..............." << endl;
        cout << "mythread start111" << ",thread id=" << this_thread::get_id
() << endl;
        std::chrono::milliseconds dura(5000);//定义5s
        std::this_thread::sleep_for(dura);//休息一定时长
        cout << "mythread end222" << ",thread id=" << this_thread::get_id()
 << endl;
        return 5;
    }
};

线程入口函数
int mythread(int myvar) {
    cout << myvar << "..............." << endl;
    cout << "mythread start111" << ",thread id=" << this_thread::get_id() <<
 endl;
    std::chrono::milliseconds dura(5000);//定义5s
    std::this_thread::sleep_for(dura);//休息一定时长
    cout << "mythread end222" << ",thread id=" << this_thread::get_id() << e
ndl;
    return 5;
}
int main(){
    下面程序通过future对象的get()成员函数等待线程执行结束并返回结果
    get()函数不拿到将来的返回值就一直阻塞等待拿值
    通过额外向std::async传递一个参数，该参数类型是std::launch类型（枚举），来达到
特殊目的
    a)std::launch::deferred：表示线程入口函数调用被延迟到std::future的wait()或
者get()函数调用时才执行
        若wait()或者get()没有被调用，那么线程会执行吗？线程不会执行，实际上线程还
没有创建
        没有创建新线程，在主线程中调用线程入口函数

    b)std::launch::async，在调用的async函数的时候就开始创建线程

    A myobja;
    int var = 12;
    cout << "main000" << "main id=" << this_thread::get_id() << endl;
    //std::future<int> result = std::async(mythread);//创建一个线程并执行，流
程并不卡在这里，future对象和该线程绑定了
    //std::future<int> result = std::async(std::launch::deferred,&A::mythrea
d,&myobja,var);//第二个参数是对象引用，这样会保证不调用拷贝构造，线程里用的是同一
```

```
个对象
    //std::future<int> result = std::async(std::launch::async, &A::mythread,
 &myobja, var);
    std::future<int> result = std::async(std::launch::deferred, &A::mythrea
d, &myobja, var);
    cout << "contune...." << endl;
    int def;
    def = 0;
    cout << result.get() << endl;//会在这等待异步线程执行完毕，并获取结果
    //result.wait();//等待线程返回，本身并不返回结果
    cout << "I love d" << endl;
    //没有get()和wait(),主线程也会等待异步线程执行完毕才会退出
}
```

std::thread()如果系统资源紧张，那么创建线程可能会失败，那么执行std::thread()时整个程序可能崩溃

std::async()一般不叫创建线程，一般叫创建一个异步任务

async和thread最明显的不同是async有时并不创建新新线程，而是谁调用get()，就在那个线程调用

a)若使用std::launch::defered来调用async会怎么样?
defered延迟调用，并不会创建新的线程，会延迟到future对象调用get()或者wait()时才执行mythread，如果没有调用get或者wait，则mythread不会执行

b)若使用launch::async：强制这个异步任务在一个新线程中执行

c)launch::async|launch::defered：或运算，意味着调用async的行为可能是 创建新线程并立即开始执行 或者是 没有创建新线程并延迟调用，两种行为之一

d)不带额外参数：std::async(mythread)，只有入口函数名，默认值为：launch::async|launch::defered，和c)效果一直，此时系统会自行决定是异步还是同步运行方式

系统如何决定是异步（创建新线程）还是同步（不创建新线程）方式运行?
std::thread创建线程，如果系统资源紧张，创建线程失败则整个程序会崩溃
std::thread创建线程的方式，如果线程有返回值，要获取这个返回值不容易
std::async创建异步任务，可能创建也可能不创建线程，并且async调用方法很容易拿到线程入口函数的返回值
由于系统资源限制：
(1)如果用thread创建线程太多，则可能创建失败，系统报告异常，崩溃
(2)若用async一般就不会崩溃，因为若系统资源紧张导致无法创建新线程时，async若没有额外参数的调用就不会创建新线程，而是后续谁调用get()来请求结果，
那么就在哪个线程执行这个异步任务。

强制async一定要创建新线程，就必须使用launch::async参数，则也可能导致崩溃
(3)经验：一个程序里线程数量不宜超过100-200个，

不加额外参数的std::async调用，让系统自行决定是否创建新线程
std::future result = std::async(mythread);这个异步任务到底有没有被推迟执行?
std::future对象的wait_for()函数，返回了一个std::future_status的枚举类型，可以判断是立即（场景新线程）执行还是延迟（没创建）执行

```
std::future<int> result = std::async(mythread);
    std::future_status status = result.wait_for(std::chrono::seconds(0));//等待0s
    if (status == std::future_status::deferred) {
        //线程被延迟执行
        cout << result.get() << endl;//此时去调用
    }
    else {
        //线程没被推迟，已经开始运行了，线程被创建了
    }
```

5.2std::packaged_task

把任务包装起来，是一个类模板，模板参数是各种可调用对象，通过package_task把各种可调用对象包装起来，方便将来作为线程入口函数进行调用

package_task包装起来的可调用对象还可以直接调用，所以package_task对象也是一个可执行对象

```
    cout << "main000" << "main id=" << this_thread::get_id() << endl;
    std::packaged_task<int(int)> mypt(mythread);//我们把mythread函数通过package_task包装起来,返回int，接收int参数
    int var = 11;
    std::thread t1(std::ref(mypt),var);//创建线程，使用ref防止拷贝构造,第二个参数为线程入口函数的参数
    t1.join();//等待线程执行完毕
    std::future<int> result = mypt.get_future();//std::future对象里包含有线程入口函数的返回结果，这里result保存muthread返回的结果
    cout << result.get() << endl;




//使用lambda表达式
    cout << "main000" << "main id=" << this_thread::get_id() << endl;
    //int var = 11;
```

```
    std::packaged_task<int(int)> mypt([](int mvar) {
        cout << mvar << "..............." << endl;
        cout << "mythread start111" << ",thread id=" << this_thread::get_id
() << endl;
        std::chrono::milliseconds dura(5000);//定义5s
        std::this_thread::sleep_for(dura);//休息一定时长
        cout << "mythread end222" << ",thread id=" << this_thread::get_id()
 << endl;
        return 5;
    });
    std::thread t1(std::ref(mypt), var);
    t1.join();
    std::future<int> result = mypt.get_future();
    cout << result.get() << endl;

    cout << "I love d" << endl;

    vector<std::packaged_task<int(int)>> mytasks;
    mytasks.push_back(std::move(mypt));//入容器，移动语义，之后mypt为空
        //从容器中获取包装任务
    std::packaged_task<int(int)> mypt2;
    auto iter = mytasks.begin();
    mypt2 = move(*iter);
    mytasks.erase(iter);//删除，迭代器失效
    mypt2(2);//相当于函数调用
    std::future<int> result = mypt2.get_future();
    cout << result.get() << endl;
```

5.3std::promise，类模板

能够在某个线程中给它赋值，然后在其他线程中把这个值取出来

总结：通过peomise保存一个值，在将来某个时刻通过future绑定这个promise

```
void mythread2(std::promise<int> &tmp, int calc) {//第一个参数
    //复杂操作
    calc++;
    calc *= 10;
    std::chrono::milliseconds dura(5000);//定义5s
    std::this_thread::sleep_for(dura);//休息一定时长
    //计算出结果
```

```
    int result = calc;//保存结果
    tmp.set_value(result);//结果保存在promise的对象中
    return;
}
void mythread3(std::future<int> &tmpf) {
    auto res = tmpf.get();
    cout <<"mythread2 res:"<< res << endl;
    return;
}

int main(){
    cout << "main000" << "main id=" << this_thread::get_id() << endl;
    std::promise<int> myprom;//声明一个promise对象，保存值的类型为int类型
    std::thread t1(mythread2, ref(myprom), 5);
    t1.join();
    //获取结果值
    std::future<int> result = myprom.get_future();//promise和future绑定，用于
获取线程返回值
    //auto res = result.get();
    //cout << res << endl;

    std::thread t2(mythread3, std::ref(result));
    t2.join();
}
```

5.4std::future的其他成员函数

wiat_for()返回枚举类型

```
    std::future<int> result = std::async(std::launch::deferred,mythread);
    std::future_status status = result.wait_for(std::chrono::seconds(6));//
等待
    if (status == std::future_status::timeout) {
        //线程超时，线程还没有执行完
        cout << "超时，线程还没执行完毕" << endl;
    }
    else if(status == std::future_status::ready) {
        //线程成功返回
        cout<< "线程执行完毕" << endl;
        cout << result.get() << endl;
    }
    else {
```

```
        //若async第一个参数被设置为std::launch::deferred
        //status == std::future_status::defeered，线程延迟执行
        cout << "线程延迟执行" << endl;
        cout << result.get() << endl;
        cout << "线程延迟执行完毕" << endl;
    }
```

5.5std::shared_future；也是个类模板

std::future，只能get()一次，因为get()函数的设计是一个移动语义
shared_future对象的get()函数则不是移动，而是复制数据

6、原子操作

互斥量：多线程编程中保护共享数据的，先锁，操作共享数据，开锁

有两个线程，对一个变量进程操作，一个线程读，另一个线程写

理解：原子操作不需要用到互斥量（加锁）的多线程并发编程方法

原子操作：在多线程执行过程中不会被打断的代码片段，比互斥量运行效率高

互斥量一般加锁一个代码段，原子操作一般只对一个变量进行操作

std::atomic来代表原子操作，是一个类模板，其实这个类模板是用来封装某个类型的值

```
int golbal = 0;//定义一个全局变量
std::atomic<int> golbal = 0;//封装了一个int类型的atomic对象，可以像操作int类型变量一样操作这个变量
//mutex mymutex;
void mythread() {
    for (int i = 0; i < 1000000; ++i) {
        //lock_guard<mutex> lg(mymutex);
        golbal++;
    }
    return;
}

int main(){
    thread t1(mythread);
    thread t2(mythread);
    t1.join();
    t2.join();
    cout << golbal << endl;
```

```
}
```

```
        //atomic的拷贝构造和赋值构造被删除
        atomic<int> atm1(atm.load());//以原子的方式读atomic对象的值
        //store()以原子方式写入一个对象
        atm1.store(12);
```

五、自动析构技术

std::lock_guard()，RAII类

```
//本类用于自动释放windows下的临界区，防止忘记LeaveCriticalSection，导致死锁的发生，类似于std::lock_gurad<std::mutex>
class CWinLock {//RAII类，资源获取即初始化，容器、智能指针、、
public:
    CWinLock(CRITICAL_SECTION *mcs) {
        my_winsec = mcs;
        EnterCriticalSection(my_winsec);
    }
    ~CWinLock() {
        LeaveCriticalSection(my_winsec);
    }
private:
    CRITICAL_SECTION *my_winsec;
};
```

六、其他互斥量

recursive_mutex递归的独占互斥量
std::mutex，独占互斥量，自己lock别人不能lock
recursive_mutex，允许同一个线程，同一个互斥量被多次lock

带超时的互斥量std::timed_mutex和std::recursive_timed_mutex
std::timed_mutex：带超时功能的独占互斥量
新接口：try_lock_for()：参数：一段时间，如100ms，等待一段时间拿到锁或者等待超时时间没有拿到锁，流程就继续往下走
try_lock_util()：参数：未来的时间点，在未来的时间点拿到了锁或者时间到了没拿到锁，流程都继续往下走

std::recursive_timed_mutex：带超时功能的递归独占互斥量

七、生产者消费者模型

```
#include <iostream>
#include <queue>
#include <thread>
#include <mutex>
#include <condition_variable>
int counter=0;
int maxSize = 30;
std::mutex mtx;
std::queue<int> dataQuene; // 被生产者和消费者共享
std::condition_variable producer, consumer;  // 条件变量是一种同步机制，要和mut
ex以及lock一起使用
void func_consumer()
{
    while (true)
    {
        std::this_thread::sleep_for(std::chrono::milliseconds(1000));// 消费
者比生产者慢
        std::unique_lock<std::mutex> lck(mtx);
        consumer.wait(lck, [] {return dataQuene.size() != 0; });     // 消费
者阻塞等待，  直到队列中元素个数大于0
        int num=dataQuene.front();
        dataQuene.pop();
        std::cout << "consumer " << std::this_thread::get_id() << ": "<< num
 <<std::endl;
        producer.notify_all();                                       // 通知
生产者当队列中元素个数小于maxSize
    }
}
void func_producer()
{
    while (true)
    {
        std::this_thread::sleep_for(std::chrono::milliseconds(900));  // 生
产者稍微比消费者快
        std::unique_lock<std::mutex> lck(mtx);
        producer.wait(lck, [] {return dataQuene.size() != maxSize; });// 生
产者阻塞等待，  直到队列中元素个数小于maxSize
        ++counter;
        dataQuene.push(counter);
        std::cout << "producer " << std::this_thread::get_id() << ": "<< cou
```

```
nter <<std::endl;
        consumer.notify_all();                                    //通知
消费者当队列中的元素个数大于0
    }
}
int main()
{
    std::thread consumers[2], producers[2];
    // 两个生产者和消费者
    for (int i = 0; i < 2; ++i)
    {
        consumers[i] = std::thread(func_consumer);
        producers[i] = std::thread(func_producer);
    }
    for (int i = 0; i < 2; ++i)
    {
        producers[i].join();
        consumers[i].join();
    }
    system("pause");
    return 0;
}
```

八、两个线程交替打印A,B

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

std::mutex data_mutex;
std::condition_variable data_var;
bool flag = true;

void printA()
{
    while(1)
    {
        std::this_thread::sleep_for(std::chrono::seconds(1));
        std::unique_lock<std::mutex> lck(data_mutex) ;
```

```
        data_var.wait(lck,[]{return flag;});
        std::cout<<"thread: "<< std::this_thread::get_id() << "   printf: "
<< "A" <<std::endl;
        flag = false;
        data_var.notify_one();
    }
}

void printB()
{
    while(1)
    {
        std::unique_lock<std::mutex> lck(data_mutex) ;
        data_var.wait(lck,[]{return !flag;});
        std::cout<<"thread: "<< std::this_thread::get_id() << "   printf: "
<< "B" <<std::endl;
        flag = true;
        data_var.notify_one();
    }
}
int main()
{
    std::thread tA(printA);
    std::thread tB(printB);
    tA.join();
    tB.join();
    return 0;
}
```