+C++11并发与多线程视频课程—学习笔记

视频课程地址：<https://www.bilibili.com/video/av49288732>

<https://study.163.com/course/introduction/1006067356.htm>

目录

[1. 并发基本概念及实现，进程、线程基本概念 5](#_Toc16335757)

[1.1. 并发、进程、线程的基本概念和综述 5](#_Toc16335758)

[1.1.1. 并发 5](#_Toc16335759)

[1.1.2. 可执行程序 6](#_Toc16335760)

[1.1.3. 进程 6](#_Toc16335761)

[1.1.4. 线程 6](#_Toc16335762)

[1.1.5. 学习心得 8](#_Toc16335763)

[1.2. 并发的实现方法 8](#_Toc16335764)

[1.2.1. 多进程并发 9](#_Toc16335765)

[1.2.2. 多线程并发 9](#_Toc16335766)

[1.2.3. 总结 10](#_Toc16335767)

[1.3. C++11新标准线程库 10](#_Toc16335768)

[2. 线程启动、结束，创建线程多法、join，detach 11](#_Toc16335769)

[2.1. 范例演示线程运行的开始和结束 11](#_Toc16335770)

[1.1.1. thread 13](#_Toc16335771)

[1.1.2. join() 13](#_Toc16335772)

[1.1.3. detach() 14](#_Toc16335773)

[1.1.4. joinable() 16](#_Toc16335774)

[2.2. 其他创建线程的方法 18](#_Toc16335775)

[2.2.1. 用类对象（可调用对象），以及一个问题范例 18](#_Toc16335776)

[2.2.2. 用lambda表达式 21](#_Toc16335777)

[3. 线程传参详解，detach()大坑，成员函数做线程函数 22](#_Toc16335778)

[3.1. 传递临时对象作为线程参数 22](#_Toc16335779)

[3.1.1. 要避免的陷阱（解释1） 22](#_Toc16335780)

[3.1.2. 要避免的陷阱（解释2） 24](#_Toc16335781)

[3.1.3. 总结 26](#_Toc16335782)

[3.2. 临时对象作为线程参数继续讲 27](#_Toc16335783)

[3.2.1. 线程id概念 27](#_Toc16335784)

[3.2.2. 临时对象构造时机捕获 27](#_Toc16335785)

[3.3. 传递类对象、智能指针作为线程参数 29](#_Toc16335786)

[3.4. 用成员函数指针做线程函数 31](#_Toc16335787)

[4. 创建多个线程、数据共享问题分析、案例代码 33](#_Toc16335788)

[4.1. 创建和等待多个线程 33](#_Toc16335789)

[4.2. 数据共享问题分析 34](#_Toc16335790)

[4.2.1. 只读的数据 34](#_Toc16335791)

[4.2.2. 有读有写 35](#_Toc16335792)

[4.2.3. 其他案例 36](#_Toc16335793)

[4.3. 共享数据的保护案例代码 36](#_Toc16335794)

[5. 互斥量概念、用法、死锁演示及解决详解 39](#_Toc16335795)

[5.1. 互斥量（mutex）的基本概念 39](#_Toc16335796)

[5.2. 互斥量的用法 39](#_Toc16335797)

[5.3. 死锁 39](#_Toc16335798)

[5.3.1. 死锁演示 39](#_Toc16335799)

[5.3.2. 死锁的一般解决方案 40](#_Toc16335800)

[5.3.3. std::lock()函数模板 40](#_Toc16335801)

[5.3.4. std::lock\_guard()的std::adopt\_lock参数 42](#_Toc16335802)

[6. unique\_lock详解 43](#_Toc16335803)

[6.1. unique\_lock取代lock\_guard 43](#_Toc16335804)

[6.2. unique\_lock的第二个参数 43](#_Toc16335805)

[6.2.1. std::adopt\_lock 43](#_Toc16335806)

[6.2.2. std::try\_to\_lock 43](#_Toc16335807)

[6.2.3. std::defer\_lock 43](#_Toc16335808)

[6.3. unique\_lock的成员函数 44](#_Toc16335809)

[6.3.1. lock() 44](#_Toc16335810)

[6.3.2. unlock() 44](#_Toc16335811)

[6.3.3. try\_lock() 44](#_Toc16335812)

[6.3.4. release() 45](#_Toc16335813)

[6.4. unique\_lock()所有权的传递 45](#_Toc16335814)

[7. 单例设计模式共享数据分析、解决，call\_once 46](#_Toc16335815)

[7.1. 设计模式大概谈 46](#_Toc16335816)

[7.2. 单例设计模式 46](#_Toc16335817)

[7.3. 单例设计模式共享数据问题分析、解决 46](#_Toc16335818)

[7.4. std::call\_once() 46](#_Toc16335819)

[8. condition\_variable、wait、notify\_one、notify\_all 48](#_Toc16335820)

[8.1. 条件变量std::condition\_variable、wait()、notify\_one() 48](#_Toc16335821)

[8.2. 上述代码深入思考 50](#_Toc16335822)

[8.3. notify\_all() 50](#_Toc16335823)

[9. async、future、packaged\_task、promise 51](#_Toc16335824)

[9.1. std::async、std::future创建后台任务并返回值 51](#_Toc16335825)

[9.2. std::packaged\_task 53](#_Toc16335826)

[9.3. promise 53](#_Toc16335827)

[10. future其他成员函数、shared\_future、atomic 54](#_Toc16335828)

[10.1. std::future的其他成员函数 54](#_Toc16335829)

[10.2. std::shared\_future 54](#_Toc16335830)

[10.3. 原子操作std::atomic 54](#_Toc16335831)

[11. stdatomic续谈、stdasync深入谈 55](#_Toc16335832)

[11.1. 原子操作std::atomic续谈 55](#_Toc16335833)

[11.2. std::async深入谈 55](#_Toc16335834)

[12. windows临界区、其他各种mutex互斥量 55](#_Toc16335835)

# 并发基本概念及实现，进程、线程基本概念

## 并发、进程、线程的基本概念和综述

### 并发

并发指的是：两个或者更多的任务（独立的活动）同时发生（进行）。在程序开发中并发指的是：一个程序同时执行独立的任务。

以往的计算机，单核cpu：某一个时刻只能执行一个任务：由操作系统调度，每秒钟进行多次的所谓的“任务切换”（task switching）。它是一种并发的假象（不是真正的并发），这种切换也叫“上下文切换”，它是有时间开销的，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，都需要时间，一会儿切换回来的时候要复原这些信息。

随着硬件发展，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（或者是多个）cpu：双核、4核、8核、10核……，就能够实现真正的并发执行多个任务（硬件并发）



使用并发的原因：主要就是同时可以干多个事，提高性能

### 可执行程序

可执行程序就是磁盘上的一个文件。在Windows下，一个扩展名为.exe的文件；在linux下，在使用命令-ls la时，文件名带rwxrwxrwx（x就表示执行权限）

### 进程

我们已经知道了可执行程序是能够运行的。在Windows下：双击一个可执行程序来运行；在linux下: ./文件名 来运行，比如：./a。

进程，就是一个可执行程序运行起来了，就叫创建了一个进程。进程就是运行起来了的可执行程序

### 线程

1. 每个进程都有一个主线程，并且这个主线程是唯一的，也就是一个进程中只能有一个主线程
2. 当你执行一个可执行程序，产生了一个进程后，这个主线程就随着这个进程默默的启动起来了。

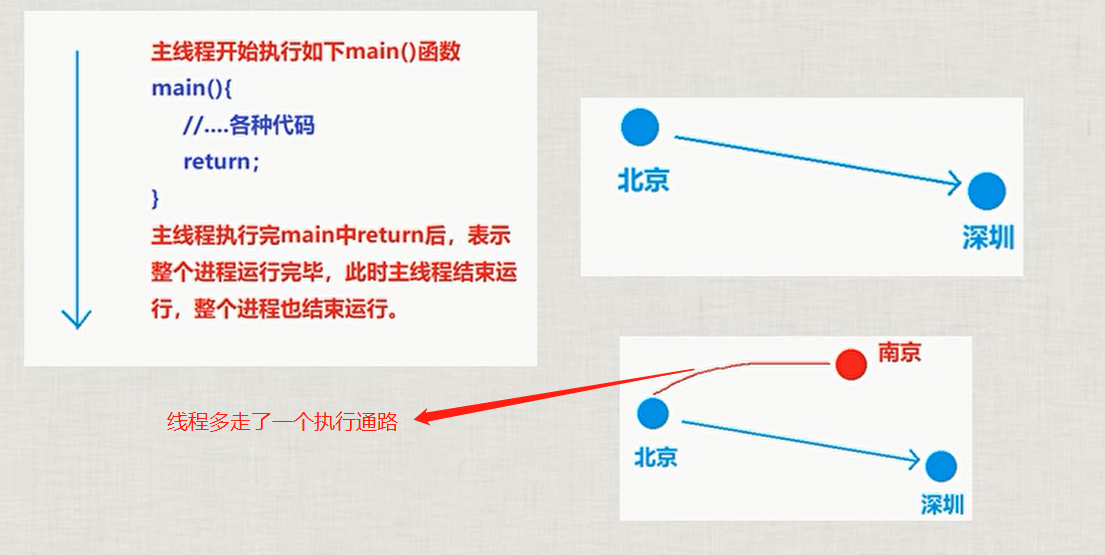
运行这个程序的时候，实际上是进程的主线程来执行（调用）这个main函数中的代码。

主线程与进程唇齿相依，有你必然有我，有我必然有你，没有我必然没有你。

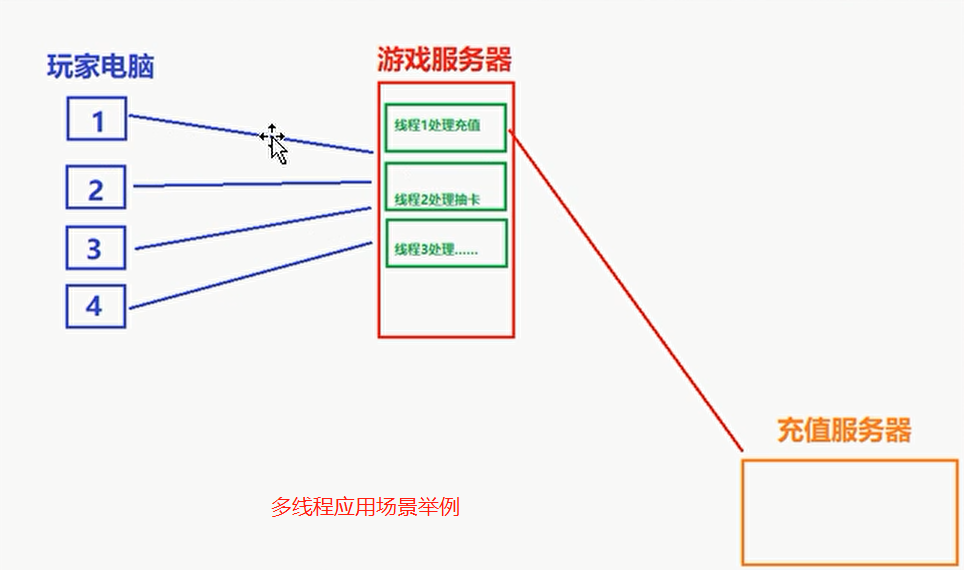
线程：用来执行代码的，理解成一条代码的执行通路

除了主线程之外，我们可以通过自己写代码来创建其他线程，其它线程走的是别的道路，甚至去不同的地方。

我每创建一个新线程，我就可以在同一时刻，多干一件不同的事（多走一条不同的代码执行路径）



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



总结一下线程：

1. 线程是用来执行代码的。
2. 把线程这个东西理解成一条代码的执行通路，一个新线程代表一个新的执行通路。
3. 一个进程自动包含一个主线程，主线程随着进程默默的启动并运行，我们可以通过编码来创建其它线程（非主线程），但是建议创建的数量最大都不超过200~300个，至于到底多少个合适，在实际的项目中可以不断的调整和优化，有的时候线程太多了效率反而会降低（因为上下文切换占用了太多没有意义的时间）。
4. 因为主线程是自动启动的，所以一个进程中最少也是有一个线程的，这个线程就是主线程
5. 说白了，多线程程序可以同时干多件事，所以运行效率高，但是到底有多高，这并不是一个很容易评估和量化的东西，所以大家仍旧需要在实际编程、实际项目中进行体会和调整优化。

### 学习心得

开发多线程程序：一个是实力的体现，一个是商用的必须需求

线程开发有一定的难度，实现代码也更复杂，理解上也更难一些，所以需要一定的学习时间。

C++线程会涉及很多新概念，对于C++道路上的成长特别关键，不要急于求成。

## 并发的实现方法

回顾并发的概念：两个或者更多的任务（独立的活动）同时发生（进行）。

实现并发的手段：

1. 通过多个进程实现并发。
2. 在单独的进程中，创建多个线程来实现并发。自己写代码来创建除了主线程之外的其他线程。

### 多进程并发

比如：Word启动起来后就是一个进程，浏览器启动起来后也是一个进程。

再比如：账号服务器 与 游戏逻辑服务器，服务器进程之间要通信。

进程间的通信：在同一个电脑上：管道、文件、消息队列、共享内存；在不同的电脑上：socket通信技术。

### 多线程并发

单个进程中，创建了多个线程。

线程：感觉像一个轻量级的进程。每个线程都有自己独立的运行路径，但是一个进程中的所有线程共享地址空间（共享内存），全局变量、指针、引用都可以在线程之间传递，所以，使用多线程开销远远小于多进程。

共享内存带来新问题：数据一致性问题。比如，两个线程在同一时刻往内存写数据，就需要采用一些技术手段，让它们有先有后；如果同时写就乱套了，你写的被他覆盖了，他写的被你覆盖了。

多进程并发和多线程并发虽然可以混合使用，但老师建议，优先考虑多线程技术手段而不是多进程。

本文中只讲多线程并发开发技术。后续谈到并发，都指的是多线程并发。

### 总结

和进程比，线程有如下优点：

1. 线程启动速度更快，更轻量级。
2. 系统资源开销更少，执行速度更快，比如共享内存这种共享方式比任何其他的通信方式都快。

缺点：

1. 使用有一定难度，要小心处理数据的一致性问题。

## C++11新标准线程库

以往：Windows下：CreateThread()、\_beginthread()、\_beginthreadex() 来创建线程。Linux下：pthread\_create() 来创建线程

以往的多线程代码不能跨平台（POSIX thread(pthread)是跨平台的，但需要做一番配置，所以用起来也不是那么方便）。

从C++11新标准开始，C++语言本身增加了对多线程的支持，意味着可移植性（跨平台），这就大大减少了开发人员的工作量了。

# 线程启动、结束，创建线程多法、join，detach

## 范例演示线程运行的开始和结束

程序运行起来，生成一个进程，该进程所属的主线程开始自动运行。

#include <iostream>

using namespace *std*;

int main()

{

*cout* << "I Love China!" << *endl*; // 实际上这个是主线程在执行，主线程从main函数返回，则整个进程执行完毕。

return 0;

}

主线程从main()开始执行，那么我们自己创建的线程，也需要从一个函数开始运行（初始函数），一旦这个函数运行完毕，就代表着我们这个线程运行结束。

提醒：整个进程是否执行完毕的标志是 主线程是否执行完，如果主线程执行完毕了，就代表整个进程执行完毕了。

此时，一般情况下：如果其它子线程还没有执行完毕，那么这些子线程也会被操作系统强行终止。所以，一般情况下，我们得到一个结论：如果想保持子线程（自己用代码创建的线程）的运行状态的话，那么就要让主线程一直保持运行，不要让主线程运行完毕。【这条规律有例外，后续会解释这种例外，大家目前先这样理解和记忆】

如何写一个多线程程序？

1. 包含一个头文件thread。
2. 初始函数要写。
3. main中开始写代码

#include <iostream>

#include <thread>

using namespace *std*;

// 自己创建的线程也要从一个函数（初始函数）开始运行：

void myPrint()

{

*cout* << "我的线程开始执行了" << *endl*;

//...

//...

*cout* << "我的线程执行完毕了" << *endl*;

}

int main()

{

thread **mytobj**(myPrint);

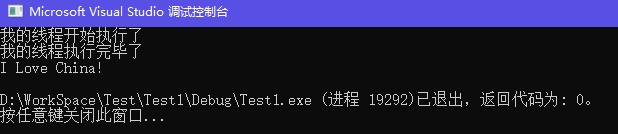
**mytobj**.*join*();

*cout* << "I Love China!" << *endl*;

return 0;

}

运行结果：



大家必须明确一点：有两个线程再跑，相当于整个程序的执行有两条线在同时走，所以，可以同时干两个事，即使一条线被堵住了，另外一条线还是可以通行的。这就是多线程。



### thread

thread是个标准库里的类

// myPrint是可调用对象

thread **mytobj**(myPrint); // （1）创建了线程，线程执行起点（入口）myPrint()；

// （2）myPrint线程开始执行

### join()

join有加入/汇合的意思，说白了就是阻塞，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。

// 阻塞主线程并使其等待myPrint子线程执行完

**mytobj**.*join*(); // 主线程阻塞到这里等待myPrint()执行完毕，这个join()就执行完毕，主线程就继续往下走。

如果不加join()：

int main()

{

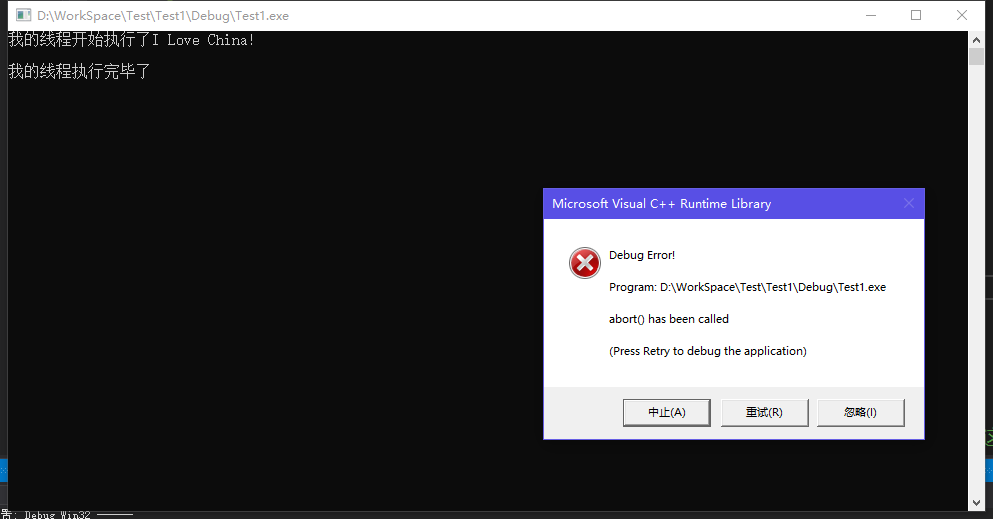
thread **mytobj**(myPrint);

//mytobj.join();

*cout* << "I Love China!" << *endl*;

return 0;

}



首先，程序报异常了，其次，控制台上的输出是混乱的。

如果主线程执行完毕了，但子线程没执行完毕，这种程序员是不合格的，写出来的程序也是不稳定的。一个书写良好的程序，应该是主线程等待子线程执行完毕后，自己才能最终退出。

### detach()

传统的多线程程序 主线程要等待子线程执行完毕，然后自己再最后退出。

detach：分离，也就是主线程不和子线程汇合了，你主线程执行你的，我子线程执行我的，你主线程也不必等我子线程执行结束了，你可以先执行结束，这并不影响我子线程的执行。

为什么引入detach()：我们创建了很多子线程，让主线程逐个等待子线程结束，这种编程方法不太好，所以引入detach。（这种说法从资料上找的，但是这种说法挺没意思的，正常情况下，还是建议主线程逐个等待子线程，但作为一门课，还是要介绍detach）

一旦detach()之后，与这个主线程关联的thread对象就会失去与这个主线程的关联，此时这个主线程就会驻留在后台运行（主线程与该子线程失去联系），这个子线程就相当于被C++运行时库接管了，当这个子线程执行完成后，由运行时库负责清理该线程相关的资源（驻留后台的线程就叫 守护线程）。

#include <iostream>

#include <thread>

using namespace *std*;

// 自己创建的线程也要从一个函数（初始函数）开始运行：

void myPrint()

{

*cout* << "我的线程开始执行了" << *endl*;

//...

//...

*cout* << "我的线程执行完毕了1" << *endl*;

*cout* << "我的线程执行完毕了2" << *endl*;

*cout* << "我的线程执行完毕了3" << *endl*;

*cout* << "我的线程执行完毕了4" << *endl*;

*cout* << "我的线程执行完毕了5" << *endl*;

*cout* << "我的线程执行完毕了6" << *endl*;

*cout* << "我的线程执行完毕了7" << *endl*;

*cout* << "我的线程执行完毕了8" << *endl*;

*cout* << "我的线程执行完毕了9" << *endl*;

*cout* << "我的线程执行完毕了10" << *endl*;

}

int main()

{

// myPrint是可调用对象

thread **mytobj**(myPrint); // （1）创建了线程，线程执行起点（入口）myPrint()；

// （2）myPrint线程开始执行

**mytobj**.*detach*();

*cout* << "I Love China 1" << *endl*;

*cout* << "I Love China 2" << *endl*;

*cout* << "I Love China 3" << *endl*;

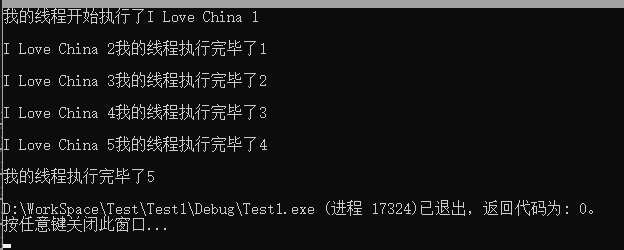
*cout* << "I Love China 4" << *endl*;

*cout* << "I Love China 5" << *endl*;

return 0;

}

运行结果：



子线程还没执行结束，主线程就退出了，随着主线程的退出，进程也就结束了，子线程在到后台执行，也就无法在控制台上看到剩下的打印信息了。

detach()使线程myPrint失去我们自己的控制。

一旦调用了detach()，就不能再用join()，否则系统会报异常：

int main()

{

thread **mytobj**(myPrint);

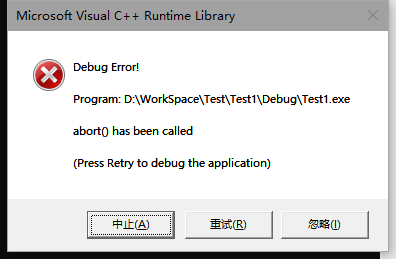
**mytobj**.*join*();

**mytobj**.*detach*();

*cout* << "I Love China 1" << *endl*;

return 0;

}



### joinable()

joinable()：判断是否可以成功使用join()或者detch()的。返回true（可以join或者detach）或者false（不能join也不能detach）

int main()

{

thread **mytobj**(myPrint);

if (**mytobj**.*joinable*()) {

*cout* << "1:joinable() == true" << *endl*;

}

else {

*cout* << "1:joinable() == false" << *endl*;

}

**mytobj**.*detach*();

if (**mytobj**.*joinable*()) {

*cout* << "2:joinable() == true " << *endl*;

}

else {

*cout* << "2:joinable() == false " << *endl*;

}

*cout* << "I Love China 1" << *endl*;

*cout* << "I Love China 2" << *endl*;

*cout* << "I Love China 3" << *endl*;

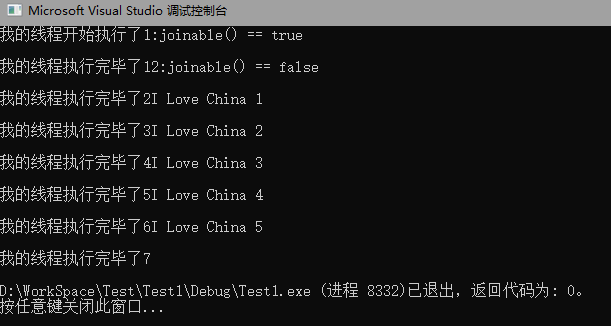
*cout* << "I Love China 4" << *endl*;

*cout* << "I Love China 5" << *endl*;

return 0;

}

运行结果：



一般用法：

int main()

{

thread **mytobj**(myPrint);

if (**mytobj**.*joinable*()) {

**mytobj**.*join*();

}

*cout* << "I Love China 1" << *endl*;

*cout* << "I Love China 2" << *endl*;

*cout* << "I Love China 3" << *endl*;

*cout* << "I Love China 4" << *endl*;

*cout* << "I Love China 5" << *endl*;

return 0;

}

## 其他创建线程的方法

### 用类对象（可调用对象），以及一个问题范例

class TA

{

public:

void operator()() { // 不带参数

*cout* << "我的线程operator()开始执行了" << *endl*;

//...

//...

*cout* << "我的线程operator()执行完毕了" << *endl*;

}

};

int main()

{

TA **ta**;

thread **mytobj**(**ta**); // ta:可调用对象

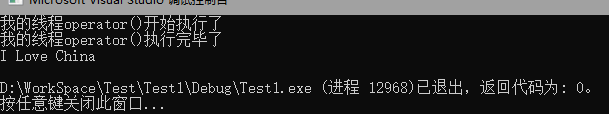
**mytobj**.*join*(); // 等待子线程执行结束

*cout* << "I Love China" << *endl*;

return 0;

}

运行结果：



用detach()当然也可以，但是detach可能会有意外的问题，现在对程序做一点调整：

class TA

{

public:

TA(int &**i**) :**m\_i**(**i**) { }

void operator()() { // 不带参数

*cout* << "m\_i1的值为：" << **m\_i** << *endl*; // 产生不可预料的后果

*cout* << "m\_i2的值为：" << **m\_i** << *endl*;

*cout* << "m\_i3的值为：" << **m\_i** << *endl*;

*cout* << "m\_i4的值为：" << **m\_i** << *endl*;

*cout* << "m\_i5的值为：" << **m\_i** << *endl*;

*cout* << "m\_i6的值为：" << **m\_i** << *endl*;

}

public:

int &**m\_i**;

};

int main()

{

int **myi** = 6;

TA **ta**(**myi**);

thread **mytobj**(**ta**); // ta:可调用对象

**mytobj**.*detach*(); // 等待子线程执行结束

*cout* << "I Love China" << *endl*;

return 0;

}

1. 当主线程退出，局部变量myi被销毁，子线程还要访问一个被销毁的变量的地址，此时会产生不可预料的后果。
2. 大家可能还有一个疑问：一旦调用了detach()，那我主线程执行结束了，那么这个ta对象还在吗？（对象不在了）

但是ta这个对象不在了也不要紧，因为这个对象实际上是被 复制 到线程中去的，执行完主线程后，ta会被销毁，但是所复制的TA对象依旧存在。所以，只要这个TA对象里没有引用，没有指针（指向主线程中的局部变量的引用或指针），那么就不会产生问题。

测试一下：

class TA

{

public:

TA(int &**i**) :**m\_i**(**i**) {

*cout* << "TA()构造函数被执行" << *endl*;

}

TA(const TA &**ta**) :**m\_i**(**ta**.**m\_i**) {

*cout* << "TA()拷贝构造函数被执行" << *endl*;

}

~TA() {

*cout*<<"TA()析构函数被执行"<<*endl*;

}

void operator()() { // 不带参数

*cout* << "m\_i1的值为：" << **m\_i** << *endl*; // 产生不可预料的后果

*cout* << "m\_i2的值为：" << **m\_i** << *endl*;

*cout* << "m\_i3的值为：" << **m\_i** << *endl*;

*cout* << "m\_i4的值为：" << **m\_i** << *endl*;

*cout* << "m\_i5的值为：" << **m\_i** << *endl*;

*cout* << "m\_i6的值为：" << **m\_i** << *endl*;

}

public:

int &**m\_i**;

};

int main()

{

int **myi** = 6;

TA **ta**(**myi**);

thread **mytobj**(**ta**); // ta:可调用对象

**mytobj**.*join*(); // 等待子线程执行结束

*cout* << "I Love China" << *endl*;

return 0;

}

运行结果：



### 用lambda表达式

int main()

{

auto **myLamThread** = [] {

*cout* << "我的线程开始执行了" << *endl*;

//...

//...

*cout* << "我的线程执行结束了" << *endl*;

};

thread **mytobj**(**myLamThread**);

**mytobj**.*join*();

return 0;

}

# 线程传参详解，detach()大坑，成员函数做线程函数

## 传递临时对象作为线程参数

### 要避免的陷阱（解释1）

#include <iostream>

#include <thread>

using namespace *std*;

void myPrint(const int &**i**,char \***pmybuf**)

{

*cout* << **i** << *endl*;

*cout* << **pmybuf** << *endl*;

return;

}

int main()

{

int **mvar** = 1;

int &**mvary** = **mvar**;

char **mybuf**[] = "this is a test!";

thread **mytobj**(myPrint, **mvary**, **mybuf**);

**mytobj**.*detach*(); // 子线程和主线程分别执行

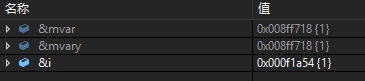
*cout* << "I Love China!" << *endl*;

return 0;

}

当主线程结束后，子线程中使用引用会不会有问题？

1. 通过调试：发现：



mvar和mvary的地址是同一个地址，但是子线程中的i的地址却是另一个地址。

分析认为：i并不是mvar的引用，实际是值传递，那么我们认为，即使主线程detach了子线程，子线程中用i值仍然是安全的（但不建议这样做）。

1. 经过分析，第一个参数是安全的，那第二个参数也是安全的吗？



这两个地址是同一个地址，所以得出结论：指针在detach子线程时绝对会有问题。

那么如何往子线程中传递字符串呢？

#include <iostream>

#include <thread>

using namespace *std*;

void myPrint(const int &**i**,const *string* &**pmybuf**)

{

*cout* << **i** << *endl*;

*cout* << **pmybuf**.*c\_str*() << *endl*;

return;

}

int main()

{

int **mvar** = 1;

int &**mvary** = **mvar**;

char **mybuf**[] = "this is a test!";

thread **mytobj**(myPrint, **mvary**, **mybuf**);

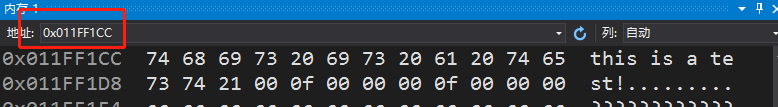
**mytobj**.*detach*(); // 子线程和主线程分别执行

*cout* << "I Love China!" << *endl*;

return 0;

}





在将mybuf传给子线程的时候做了隐式类型转换，将char\*隐式转换成了const string，自然就拷贝了一份string出来，看起来“感觉”是可以的，那么到底可不可以呢？再看3.1.2解释2。

### 要避免的陷阱（解释2）

thread **mytobj**(myPrint, **mvary**, **mybuf**); // 但是mybuf是在什么时候转换成string的？如果是在主线程return后才转换成string的，那mybuf不早就被系统回收了吗?

事实上存在，mybuf都被回收了（main函数执行完了），系统才用mybuf去转string的可能性。

修改代码：

thread **mytobj**(myPrint, **mvary**, *string*(**mybuf**)); // 我们这里直接将mybuf转换成string对象，这是一个可以保证在线程中用的 肯定有效的对象。

那么为什么这么写就没问题了呢？

求证：

#include <iostream>

#include <thread>

using namespace *std*;

class A

{

public:

// 类型转换构造函数，可以把一个int转换成一个类A对象。

A(int **i**) :**m\_i**(**i**) { *cout* << "[A::A(int i)构造函数执行]" << *endl*; }

A(const A &**a**) :**m\_i**(**a**.**m\_i**) { *cout* << "A::A(const A &a)拷贝构造函数执行" << *endl*; }

~A() { *cout* << "[A::A()析构函数执行]" << *endl*; }

private:

int **m\_i**;

};

void myPrint(const int **a**, const A &**pmybuf**)

{

*cout* << &**pmybuf** << *endl*; // 这里打印的是pmybuf对象的地址

}

int main()

{

int **mvar** = 1;

int **mysecondpar** = 12;

thread **mytobj**(myPrint, **mvar**, **mysecondpar**); // 我们是希望mysecondpar转成A类型对象转递给myPrint的第二个参数

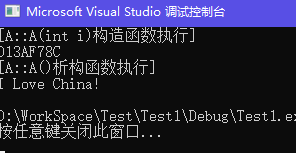
**mytobj**.*join*();

*cout* << "I Love China!" << *endl*;

return 0;

}

运行结果：



我们发现，join的时候确实是mysecondpar转成A类型对象转递给myPrint的第二个参数了。那么此时用detach再测试一下：

**mytobj**.*detach*(); // 子线程和主线程分别执行

return 0;

运行结果：



有几率出现这种情况

此时，这个构造是在主线程return后才构造的，这时就会出现问题。

改写代码，构造临时对象（并打印this）：

thread **mytobj**(myPrint, **mvar**, A(**mysecondpar**)); // 我们是希望mysecondpar转成A类型对象转递给myPrint的第二个参数

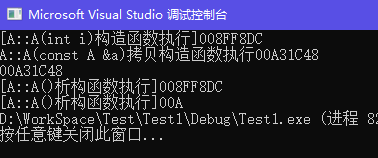
class A{

A(int **i**) :**m\_i**(**i**) { *cout* << "[A::A(int i)构造函数执行]" << this << *endl*; }

……

}

运行结果：



此时类A肯定是在main结束前构造出来的，就不怕mysecondpar被回收了。所以，在创建线程的同时构造临时对象的方法传递参数是可行的

事实1：只要用临时构造的A类对象作为参数传递给线程，那么就一定能够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()了子线程，也能安全运行

### 总结

1. 若传递int这种简单类型参数，建议都是值传递，不要用引用，防止节外生枝。
2. 如果传递类对象，避免隐式类型转换。应该全部都在创建线程这一行就构建出临时对象来，然后在函数参数里，用引用来接，否则系统还会再多构造一次对象，浪费。

终极结论：

1. 建议不使用detach()，只使用join()。这样就不存在局部变量失效导致线程对内存的非法引用问题。

## 临时对象作为线程参数继续讲

### 线程id概念

id是个数字，每个线程（不管是主线程还是子线程）实际上都对应着一个数字，而且每个线程对应的数字都不同，也就是说，不同的线程，它的线程id（数字）必然是不同的。

线程id可以用c++标准库里的函数来获取。std::this\_thread::get\_id()来获取

### 临时对象构造时机捕获

#include <iostream>

#include <thread>

using namespace *std*;

class A

{

public:

// 类型转换构造函数，可以把一个int转换成一个类A对象。

A(int **i**) :**m\_i**(**i**) { *cout* << "[A::A(int i)构造函数执行]" << this << "threadid = " << *std*::*this\_thread*::*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 << "threadid = " << *std*::*this\_thread*::*get\_id*() << *endl*; }

~A() { *cout* << "[A::A()析构函数执行]" << this << "threadid = " << *std*::*this\_thread*::*get\_id*() << *endl*; }

private:

int **m\_i**;

};

void myPrint(const A &**pmybuf**)

{

*cout* << "子线程myPrint的参数地址是" << &**pmybuf** << "threadid = " << *std*::*this\_thread*::*get\_id*() << *endl*;

}

int main()

{

*cout* << "主线程id = " << *std*::*this\_thread*::*get\_id*() << *endl*;

int **mvar** = 1;

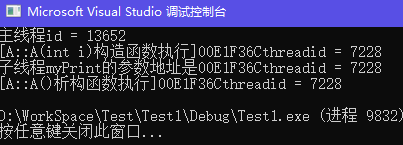
thread **mytobj**(myPrint, **mvar**);

**mytobj**.*join*();

return 0;

}

运行结果：

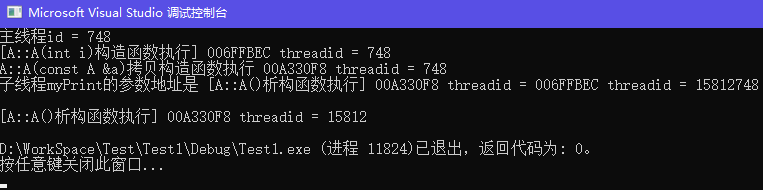


注意到：隐式类型转换居然是在子线程中进行的，这意味着如果detach的话，主线程return后，如果子线程还没开始执行隐式类型转换的话就会访问一个不存在的内存空间，从而导致程序异常。

那么如果构造临时对象呢？

thread **mytobj**(myPrint, A(**mvar**));

运行结果：



结果显示：用了临时对象后，所有的A类对象都在main()函数中就已经构建完毕了。

## 传递类对象、智能指针作为线程参数

#include <iostream>

#include <thread>

using namespace *std*;

class A

{

public:

// 类型转换构造函数，可以把一个int转换成一个类A对象。

A(int **i**) :**m\_i**(**i**) { *cout* << "[A::A(int i)构造函数执行] " << this << " threadid = " << *std*::*this\_thread*::*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 << " threadid = " << *std*::*this\_thread*::*get\_id*() << *endl*; }

~A() { *cout* << "[A::A()析构函数执行] " << this << " threadid = " << *std*::*this\_thread*::*get\_id*() << *endl*; }

public:

mutable int **m\_i**; // mutable后，就算是const的变量也可以修改了

};

void myPrint(const A &**pmybuf**)

{

**pmybuf**.**m\_i** = 199; // 我们修改该值不会影响到main函数

*cout* << "子线程myPrint的参数地址是 " << &**pmybuf** << " threadid = " << *std*::*this\_thread*::*get\_id*() << *endl*;

}

int main()

{

A **myobj**(10); // 生成一个类对象

thread **mytobj**(myPrint, **myobj**); // 将类对象作为线程参数

**mytobj**.*join*();

return 0;

}



结果显示：虽然void myPrint(const A &**pmybuf**)这里传的是引用，但是myobj的结果并没有被修改。

解决这个问题的办法就是std::ref函数：

#include <iostream>

#include <thread>

using namespace *std*;

class A

{

public:

A(int **i**) :**m\_i**(**i**) { *cout* << "[A::A(int i)构造函数执行] " << this << " threadid = " << *std*::*this\_thread*::*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 << " threadid = " << *std*::*this\_thread*::*get\_id*() << *endl*; }

~A() { *cout* << "[A::A()析构函数执行] " << this << " threadid = " << *std*::*this\_thread*::*get\_id*() << *endl*; }

public:

int **m\_i**; // 这个时候mutable也不需要了

};

void myPrint(A &**pmybuf**) // const也不需要了

{

**pmybuf**.**m\_i** = 199;

*cout* << "子线程myPrint的参数地址是 " << &**pmybuf** << " threadid = " << *std*::*this\_thread*::*get\_id*() << *endl*;

}

int main()

{

A **myobj**(10);

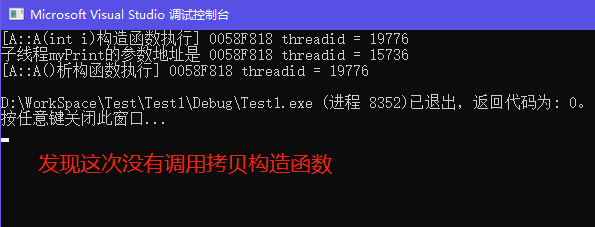
thread **mytobj**(myPrint, *std*::*ref*(**myobj**));

**mytobj**.*join*();

return 0;

}

输出结果：





此时就是真正的传引用了。

那么如果子线程的参数是智能指针，应该怎么传呢？

#include <iostream>

#include <thread>

#include <memory>

using namespace *std*;

void myPrint(*unique\_ptr*<int> **pmybuf**)

{

*cout* << "子线程myPrint的参数地址是 " << &**pmybuf** << " threadid = " << *std*::*this\_thread*::*get\_id*() << *endl*;

}

int main()

{

*unique\_ptr*<int> **myp**(new int(100));

thread **mytobj**(myPrint, *std*::*move*(**myp**)); // 用std::move()

**mytobj**.*join*();

return 0;

}

## 用成员函数指针做线程函数

#include <iostream>

#include <thread>

using namespace *std*;

class A

{

public:

A(int **i**) :**m\_i**(**i**) { *cout* << "[A::A(int i)构造函数执行] " << this << " threadid = " << *std*::*this\_thread*::*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 << " threadid = " << *std*::*this\_thread*::*get\_id*() << *endl*; }

~A() { *cout* << "[A::A()析构函数执行] " << this << " threadid = " << *std*::*this\_thread*::*get\_id*() << *endl*; }

public:

void thread\_work(int **num**) {

*cout* << "子线程thread\_work执行] " << this << " threadid = " << *std*::*this\_thread*::*get\_id*() << *endl*;

}

public:

int **m\_i**;

};

int main()

{

A **myobj**(10);

thread **mytobj**(&A::thread\_work, **myobj**, 15);

**mytobj**.*join*();

return 0;

}

# 创建多个线程、数据共享问题分析、案例代码

## 创建和等待多个线程

#include <iostream>

#include <thread>

#include <vector>

using namespace *std*;

void myPrint(int **num**)

{

*cout* << "myPrint线程开始执行了，线程编号=" << **num** << *endl*;

//...干各种事情

*cout* << "myPrint线程结束执行了，线程编号=" << **num** << *endl*;

}

int main()

{

*vector*<thread> **mythreads**;

// 创建10个线程,入口函数统一使用myPrint

for (int **i** = 0; **i** < 10; ++**i**) {

**mythreads**.*push\_back*(thread(myPrint, **i**)); // 创建10个线程，同时这10个线程已经开始执行

}

for (auto **iter** = **mythreads**.*begin*(); **iter** != **mythreads**.*end*(); ++**iter**) {

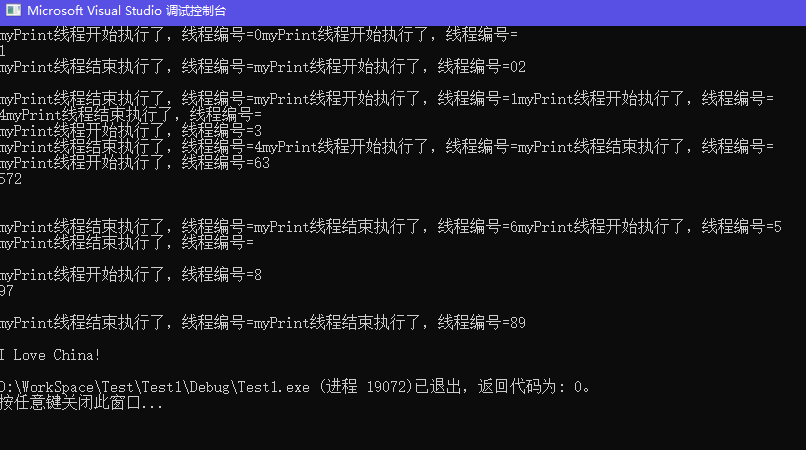
**iter**->*join*(); // 等待10个线程都返回

}

*cout* << "I Love China!" << *endl*; // 最后执行这句，整个进程退出

}

运行结果：



1. 多个线程执行顺序是乱的，跟操作系统内部对线程的运行调度机制有关。
2. 主线程等待所有子线程运行结束，最后主线程结束，老师推荐这种join的写法，更容易写出稳定的程序。
3. 咱们把thread对象放入到容器里管理，看起来像个 thread对象数组，这对我们一次创建大量的线程并对大量的线程进行管理很方便。

## 数据共享问题分析

### 只读的数据

#include <iostream>

#include <thread>

#include <vector>

using namespace *std*;

*vector*<int> **g\_v** = { 1,2,3 }; // 共享数据，只读

void myPrint(int **num**)

{

*cout* << "id为" << *std*::*this\_thread*::*get\_id*() << "的线程 打印g\_v值" << **g\_v**[0] << **g\_v**[1] << **g\_v**[2] << *endl*;

}

int main()

{

*vector*<thread> **mythreads**;

// 创建10个线程,入口函数统一使用myPrint

for (int **i** = 0; **i** < 10; ++**i**) {

**mythreads**.*push\_back*(thread(myPrint, **i**)); // 创建10个线程，同时这10个线程已经开始执行

}

for (auto **iter** = **mythreads**.*begin*(); **iter** != **mythreads**.*end*(); ++**iter**) {

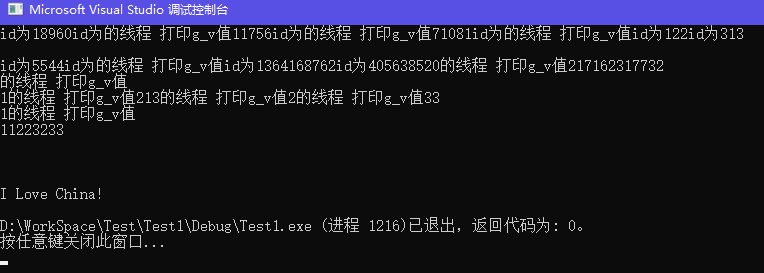
**iter**->*join*(); // 等待10个线程都返回

}

*cout* << "I Love China!" << *endl*; // 最后执行这句，整个进程退出

}

运行结果：



没有报异常。

只读的数据：是安全稳定的，不需要特别什么处理手段，直接读就可以。

### 有读有写

比如：2个线程写，8个线程读，如果代码没有特别的处理，那程序肯定崩溃。最简单的不崩溃处理：读的时候不能写，写的时候不能读，2个线程不能同时写。

比如写的步骤分10小步，由于任务切换，导致各种诡异事情发生（最可能的诡异事情还是崩溃）。

### 其他案例

现实中的数据共享：两人同时定一张火车票。

## 共享数据的保护案例代码

现在有一个网络游戏服务器。其中有两个自己创建的线程，一个线程收集玩家命令（用一个数字代表玩家发来的命令），并把命令数据写到一个队列中；另一个线程从队列中取出玩家发送来的命令，解析，然后执行玩家需要的动作。

#include <iostream>

#include <thread>

#include <vector>

#include <list>

using namespace *std*;

class A

{

public:

// 把收到的消息（玩家命令）入到一个队列的线程

void inMsgRevQueue() {

for (int **i** = 0; **i** < 100000; ++**i**) {

*cout* << "inMsgRevQueue()执行，插入一个元素" << **i** << *endl*;

**msgRevQueue**.*push\_back*(**i**); // 假设这个数字i就是我收到的玩家的命令，我直接弄到消息队列里边来

}

}

// 把数据从消息队列中取出的线程

void outMsgRevQueue() {

for (int **i** = 0; **i** < 100000; ++**i**) {

if (!**msgRevQueue**.*empty*()) {

// 消息队列不为空

int **command** = **msgRevQueue**.*front*(); // 返回第一个元素，但不检查元素是否存在

**msgRevQueue**.*pop\_front*(); // 移除第一个元素，但不返回

// 这里就考虑处理数据...

// ...

}

else {

// 消息队列为空

*cout* << "outMsgRevQueue()执行，但目前消息队列中为空" << *endl*;

}

}

*cout* << "end" << *endl*;

}

private:

*list*<int> **msgRevQueue**; // 容器（消息队列），专门用于代表玩家发送的命令

};

int main()

{

A **myobj**;

thread **myOutnMsgObj**(&A::outMsgRevQueue, &**myobj**); // 第二个参数是引用，才能保证线程里用的是同一个对象

thread **myInMsgObj**(&A::inMsgRevQueue, &**myobj**);

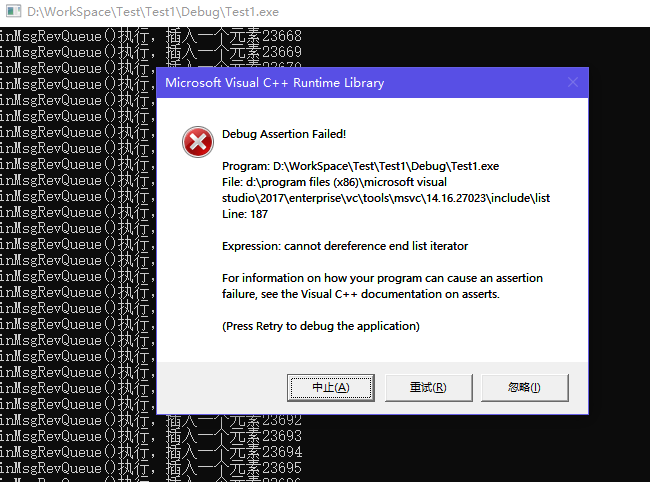
**myOutnMsgObj**.*join*();

**myInMsgObj**.*join*();

return 0;

}

运行结果：



代码化解决问题：引入一个c++解决多线程保护共享数据的第一个概念“互斥量”。

# 互斥量概念、用法、死锁演示及解决详解

## 互斥量（mutex）的基本概念

略

## 互斥量的用法

略

## 死锁

比如我有两把锁（死锁这个问题，是由至少两个锁头也就是两个互斥量才能产生）；金锁（JinLock），银锁（YinLock）；

现在有两个线程A、B；

1. 线程A执行的时候，这个线程先锁 金锁，把金锁lock()成功了，然后它去lock()银锁。。。此时出现了上下文切换。
2. 线程B执行了，这个线程先锁 银锁，因为银锁还没有被锁，所以银锁会lock()成功，线程B要去lock()金锁。。。
3. 此时此刻，死锁就产生了。

### 死锁演示

略

### 死锁的一般解决方案

只要保证两个互斥量上锁的顺序一致就不会死锁。

### std::lock()函数模板

用于处理多个互斥量时才出场。

能力：一次锁住两个或者两个以上的互斥量（至少两个，多了不限，1个不行）。它不存在这种因为在多个线程中 因为锁的顺序导致死锁的风险问题。

如果互斥量中有一个没锁柱，它就在那里等着，等所有互斥量都锁住，它才能往下走（返回）。

要么两个互斥量都锁住，要么两个互斥量都没锁住。如果只锁了一个，另一个没锁成功，则它立即把已经锁住的解锁。

#include <iostream>

#include <thread>

#include <vector>

#include <list>

#include <mutex>

using namespace *std*;

class A

{

public:

void inMsgRevQueue() {

for (int **i** = 0; **i** < 100000; ++**i**) {

*cout* << "inMsgRevQueue()执行，插入一个元素" << **i** << *endl*;

*std*::*lock*(**my\_mutex1**, **my\_mutex2**); // 相当于每个互斥量都调用了lock()

**msgRevQueue**.*push\_back*(**i**);

**my\_mutex1**.*unlock*();

**my\_mutex2**.*unlock*();

}

}

bool outMsgLULProc(int **command**) {

*std*::*lock*(**my\_mutex1**, **my\_mutex2**);

if (!**msgRevQueue**.*empty*()) {

**command** = **msgRevQueue**.*front*();

**msgRevQueue**.*pop\_front*();

// 这里就考虑处理数据...

// ...

**my\_mutex1**.*unlock*();

**my\_mutex2**.*unlock*();

return true;

}

**my\_mutex1**.*unlock*();

**my\_mutex2**.*unlock*();

return false;

}

void outMsgRevQueue() {

int **command** = 0;

for (int **i** = 0; **i** < 100000; ++**i**) {

bool **result** = outMsgLULProc(**command**);

if (**result**) {

*cout* << "outMsgLULProc() 执行，取出一个元素" << **command** << *endl*;

}

else {

*cout* << "outMsgRevQueue()执行，但目前消息队列中为空" << *endl*;

}

}

*cout* << "end" << *endl*;

}

private:

*list*<int> **msgRevQueue**;

*mutex* **my\_mutex1**;

*mutex* **my\_mutex2**;

};

int main()

{

A **myobj**;

thread **myOutnMsgObj**(&A::outMsgRevQueue, &**myobj**);

thread **myInMsgObj**(&A::inMsgRevQueue, &**myobj**);

**myOutnMsgObj**.*join*();

**myInMsgObj**.*join*();

return 0;

}

### std::lock\_guard()的std::adopt\_lock参数

std::adopt\_lock是个结构体对象，起一个标记作用：作用就是表示这个互斥量已经lock()，不需要在std::lock\_guard<mutex>构造函数里面对mutex对象进行再次lock()了。

*std*::*lock*(**my\_mutex1**, **my\_mutex2**); // 相当于每个互斥量都调用了lock()

*lock\_guard*<*mutex*> **guard1**(**my\_mutex1**, *std*::*adopt\_lock*);

*lock\_guard*<*mutex*> **guard2**(**my\_mutex2**, *std*::*adopt\_lock*);

**msgRevQueue**.*push\_back*(**i**);

//my\_mutex1.unlock();

//my\_mutex2.unlock();

总结：std::lock()：一次锁定多个互斥量；谨慎使用（建议一个一个锁）。

# unique\_lock详解

## unique\_lock取代lock\_guard

unique\_lock比lock\_guard灵活很多，但是效率上差一点，内存占用多一点。

## unique\_lock的第二个参数

### std::adopt\_lock

表示这个互斥量已经被lock了（你必须要把互斥量提前lock了，否则会报异常）

### std::try\_to\_lock

我们会尝试用mutex的lock() 去锁定这个mutex，但如果没有锁定成功，我也会立即返回，并不会阻塞在那里。

用这个try\_to\_lock的前提是你自己不能先去lock()。

### std::defer\_lock

用defer\_lock的前提是你不能自己先lock()，否则会报异常。

defer\_lock的意思就是并没有给mutex加锁：初始化了一个没有加锁的mutex。详情见下节 。

## unique\_lock的成员函数

### lock()

*unique\_lock*<*mutex*> **guard**(**my\_mutex**, *std*::*defer\_lock*); // 没有加锁的my\_mutex

**guard**.*lock*() // 咱们不用自己unlock()

### unlock()

**guard**.*lock*();

处理一些共享代码

因为有一些非共享代码要处理

**guard**.*unlock*();

处理一些非共享代码

**guard**.*lock*();

处理一些共享代码

### try\_lock()

尝试给互斥量加锁，如果拿不到锁，则返回false，如果拿到了锁，返回true。这个函数是不阻塞的。

*unique\_lock*<*mutex*> **guard**(**my\_mutex**, *std*::*defer\_lock*); // 没有加锁的my\_mutex

if (**guard**.*try\_lock*() == true) { // 返回true表示拿到锁了

// 拿到了锁

**msgRevQueue**.*push\_back*(**i**);

// 其他处理代码

// ...

}

else {

// 没拿到锁

*cout* << "inMsgRevQueue()执行，但没有拿到锁，只能干点别的事 " << **i** << *endl*;

}

### release()

返回它所管理的mutex指针，并释放所有权。也就是说，这个unique\_lock和这个mutex不再有关系。

如果原来mutex对象处于加锁状态，你有责任接管过来并负责解锁。

## unique\_lock()所有权的传递

1. 用std::move()

*unique\_lock*<*mutex*> **guard**(**my\_mutex**);

*unique\_lock*<*mutex*> **guard**(*std*::*move*(**my\_mutex**)); // 移动语义

1. return unique\_lock<mutex>

*unique\_lock*<*mutex*> rtn\_unique\_lock(){

*unique\_lock*<*mutex*> **tmpguard**(**my\_mutex**);

return **tmpguard**; // 从函数返回一个局部的unique\_lock是可以的。返回这种局部temguard会导致系统生成临时的unique\_lock对象，并调用unique\_lock的移动构造函数

}

*unique\_lock*<*mutex*> **guard** = rtn\_unique\_lock();

# 单例设计模式共享数据分析、解决，call\_once

## 设计模式大概谈

略

## 单例设计模式

单例：整个项目中，有某个或者某些特殊的类，属于该类的对象只能创建一个，不能创建多个。

A **a1**;

A **a2**; // 在单例设计模式中会报错。

## 单例设计模式共享数据问题分析、解决

略

## std::call\_once()

c++11引入的函数，该函数的第二个参数是一个函数名。

call\_once的功能是能够保证函数a()只被调用一次。

call\_once具备互斥量的能力，而且效率上，比互斥量消耗的资源更少。

call\_once需要与一个标记结合使用，这个标记是std::once\_falg。

call\_once就是通过这个标记来决定对应的函数a()是否执行，调用call\_once()成功以后，call\_once()就把标记设置成一种已调用状态。

后续再次调用call\_once()，只要once\_falg被设置为了“已调用”状态，那么对应的函数a()就不会再被执行了。

# condition\_variable、wait、notify\_one、notify\_all

## 条件变量std::condition\_variable、wait()、notify\_one()

condition\_variable实际上是一个类，是一个和条件相关的类，说白了就是等待一个条件达成。

这个类是需要和互斥量来配合工作，用的时候我们要生成这个类的对象。

#include <iostream>

#include <thread>

#include <vector>

#include <list>

#include <mutex>

using namespace *std*;

class A

{

public:

void inMsgRevQueue() {

for (int **i** = 0; **i** < 100000; ++**i**) {

*cout* << "inMsgRevQueue()执行，插入一个元素" << **i** << *endl*;

*unique\_lock*<*mutex*> **guard**(**my\_mutex**);

**msgRevQueue**.*push\_back*(**i**);

// 假如outMsgRevQueue()正在处理一个事务，需要一段时间，而不是卡在wait()那里等着你唤醒，那么此时这个notify\_one()就没有效果。

**my\_cond**.*notify\_one*(); // 我们尝试把wait的线程唤醒,执行完这行，那么outMsgRevQueue()里边的wait就会被唤醒

// 其他处理代码

// ...

}

}

void outMsgRevQueue() {

int **command** = 0;

while (true) {

*unique\_lock*<*mutex*> **guard**(**my\_mutex**);

// wait用来等一个东西

// 如果第二个参数lambda表达式返回值是true，那么wait()直接返回。

// 如果第二个参数lambda表达式返回值是false，那么wait()将解锁互斥量，并堵塞到本行，

// 那堵塞到什么时候为止呢？堵塞到其他某个线程调用notify\_one()这个成员函数为止。

// 如果wait()没有第二个参数：那么就跟第二个参数返回false一样。

// 当其他线程用notify\_one()将本wait（原来是睡着/阻塞）的状态唤醒后，wait就开始恢复干活了：

// a)wait()不断的尝试重新获取互斥量锁，如果获取不到，那么流程就卡在wait这里等着获取；如果获取到了锁，那么wait就继续执行b。

// b)

// b.1)如果wait有第二个参数(lambda)，就判断这个lambda表达式，如果lambda表达式为false，那wait又对互斥量解锁，然后又休眠，等待再次被notify\_one唤醒。

// b.2)如果lambda表达式为true，则wait返回，流程走下来。

// b.3)如果wait没有第二个参数，则wait返回，流程走下来。

**my\_cond**.*wait*(**guard**, [this] {

if (!**msgRevQueue**.*empty*())

return true;

else

return false;

});

// 流程只要能走到这里来，这个互斥锁一定是锁着的，同时msgRevQueue至少有一条数据

**command** = **msgRevQueue**.*front*();

**msgRevQueue**.*pop\_front*();

**guard**.*unlock*();

*cout* << "outMsgRevQueue()执行，取出一个元素" << **command** << *endl*;

}

}

private:

*list*<int> **msgRevQueue**;

*mutex* **my\_mutex**;

*condition\_variable* **my\_cond**; // 生成一个条件变量对象

};

int main()

{

A **myobj**;

thread **myOutnMsgObj**(&A::outMsgRevQueue, &**myobj**);

thread **myInMsgObj**(&A::inMsgRevQueue, &**myobj**);

**myOutnMsgObj**.*join*();

**myInMsgObj**.*join*();

return 0;

}

## 上述代码深入思考

略

## notify\_all()

notify\_one()只能通知一个此条件变量的wait的线程，如果此条件变量有多个wait的线程呢？

notify\_all()通知此条件变量的所有wait线程，但只有一个wait的线程能够走下去（因为wait走下去后，锁是锁住的）。

# async、future、packaged\_task、promise

## std::async、std::future创建后台任务并返回值

用处：希望线程返回一个结果

std::async用来启动一个异步任务，它返回一个std::future对象。

启动一个异步任务就是指：自动创建一个线程并开始执行对应的线程入口函数，它返回一个std::future对象，我们可以通过调用future对象的成员函数get()来获取线程返回的结果。

#include <iostream>

#include <thread>

#include <future>

using namespace *std*;

int mythread()

{

*cout* << "mythread() start, threadid = " << *this\_thread*::*get\_id*() << *endl*;

*this\_thread*::*sleep\_for*(5s);

*cout* << "mythread() end, threadid = " << *this\_thread*::*get\_id*() << *endl*;

return 5;

}

int main()

{

*cout* << "main threadid = " << *this\_thread*::*get\_id*() << *endl*;

*future*<int> **result** = *std*::*async*(mythread); // 创建一个线程并开始执行，绑定关系

*cout* << "continue...!" << *endl*;

int **def** = 0;

*cout* << **result**.*get*() << *endl*; // 卡在这里等待mythread()执行完毕，拿到结果

//cout<<result.get()<<endl; // get只能调用一次，不能调用多次

*cout* << "I Love China!" << *endl*;

return 0;

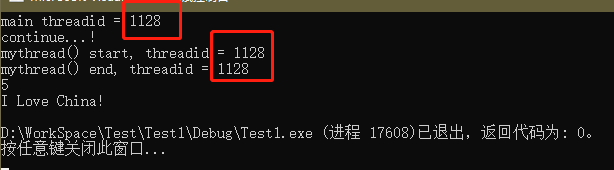
}

我们通过额外向std::async()传递一个参数，该参数类型是std::launch类型（也就是枚举类型），来达到一些特殊目的。

1. std:: launch::deferred：表示线程入口函数被延迟到std::future的wait()或者get()函数调用时才执行。

*future*<int> **result** = *std*::*async*(*launch*::*deferred*, mythread);

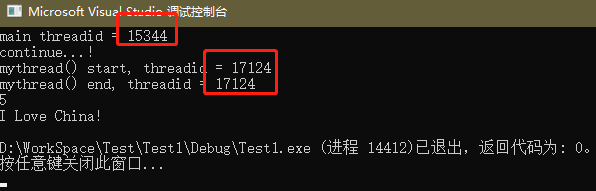
运行结果：



延迟调用，并且没有创建新线程，是在主线程中调用的 线程入口函数。

1. std::launch::async：在调用async函数的时候就开始创建线程。

*future*<int> **result** = *std*::*async*(*launch*::*async*, mythread);



1. std::launch::async | std:: launch::deferred：意味着调用async的行为可能是 创建新线程并立即执行 或者 没有创建新线程并且延迟到调用result.get()才开始执行任务入口函数，两者居其一。
2. 不带额外参数，即默认情况下：默认值是std::launch::async | std:: launch::deferred。和C效果完全一致。换句话说，系统会自动决定是异步还是同步方式运行（当系统资源不够的时候会采用同步方式，不创建新线程）。

## std::packaged\_task

打包任务，把任务包装起来，很像std::function

## promise

我们能够在某个线程中给它赋值，然后可以在其他线程中把这个值取出来用。

总结：通过promise保存一个值，在将来某个时刻我们通过把一个future绑定到这个promise上来 得到这个绑定的值。

# future其他成员函数、shared\_future、atomic

## std::future的其他成员函数

wait\_for：等待后返回一个状态std::future\_status

## std::shared\_future

std::future的get()函数是一个移动语义，转移数据；std::shared\_future的get()函数则是 复制 数据。

## 原子操作std::atomic

略

# stdatomic续谈、stdasync深入谈

## 原子操作std::atomic续谈

一般atomic原子操作，针对++,--,+=,&=,|=,^=是支持的，其他的可能不支持。

## std::async深入谈

略

# windows临界区、其他各种mutex互斥量

windows临界区类似于mutex

RAII类（Resource Acquisition initialization），翻译成中文是“资源获取即初始化“。就是具有自动析构技术的类，容器、只能指针都属于RAII类。