线程同步和互斥

同步就是协同步调，按预定的先后次序进行运行。如：你说完，我再说。这里的同步千万不要理解成那个同时进行，应是指协同、协助、互相配合。

线程同步是指多线程通过特定的设置（如互斥量，事件对象，临界区）来控制线程之间的执行顺序（即所谓的同步）也可以说是在线程之间通过同步建立起执行顺序的关系，如果没有同步，那线程之间是各自运行各自的！

线程互斥是指对于共享的进程系统资源，在各单个线程访问时的排它性。当有若干个线程都要使用某一共享资源时，任何时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。线程互斥可以看成是一种特殊的线程同步。

多线程解释：

https://www.runoob.com/cplusplus/cpp-multithreading.html

清华大学多线程课程

线程共享进程的堆，有自己的栈

内核线程

操作系统内核支持多线程调度，内核线程使用资源少，仅包括内核栈和上下文切换时需要的保存寄存器内容的空间

轻量级进程

由内核支持的独立调度单元，调度开销小于普通的进程，系统支持多个轻量级进程调度

用户线程

建立在用户空间的多个用户线程，映射到轻量级别进程后调度执行; 用户线程在用户空间创建，同步和销毁，开销小，每个线程具有独特的ID

线程与进程的区别

线程不需要C++11的标准库,线程空间不独立，有问题的线程会影响其它线程，进程空间独立，有问题的进程一般不会应下该其他进程，创建进程需要额外的性能开销，线程用于开发细颗粒并行性，进程用于开发粗颗粒并行，线程容易共享数据，进程共享数据必须使用进程间通讯机制

线程管理

线程创建

“pthread.h"

int pthread_create(pthread_t* thread（存储线程的ID）,const pthread_attr_t* attr（NULL), void* *(*start_routine)(void*)(线程执行函数）, (void*)*arg (附加参数，传参数列表))

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

//1
```

```
void* PrintA(void* unused)
{
while(true) std::cerr<<'a';
return NULL;
}

void* PrintZ(void* unused)
{
while(true) std::cerr<<'z';
return NULL;
}

int main()
{
        //1
    pthread_t thread_id;
    pthread_create(&thread_id,NULL,&PrintA,NULL);
    PrintZ(NULL);
    return 0;
}
```

线程创建完毕立即返回，并不等待线程结束，原线程与新线程如何执行与调度有关，程序不得依赖线程先后执行的关系，可以使用同步机制确定线程的先后执行关系

线程退出

线程函数结束执行，调用pthread_exit()函数显示结束，被其他线程撤销

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

//2-1
class InfoPrinted
{
public:
    InfoPrinted(char c,int n):_c(c),_n(n){}
    void Show() const
    { for(int i=0;i<_n;i++)
        std::cerr<<_c;

    }
private:
    char _c;
    int _n;
};

void* PrintInfo(void* info)
{
    InfoPrinted* p=reinterpret_cast<InfoPrinted*>(info);//亚星指针
    if(p)   p->Show();
    return NULL;
}
```

```
int main()
{

        //2-1
        pthread_t tid1,tid2;
    //构造InfoPrinted类的动态对象，作为线程函数参数传递给线程tid1
    //输出100个‘a'
    InfoPrinted* p=new InfoPrinted('a',100);
       pthread_create(&tid1,NULL,&PrintInfo,reinterpret_cast<void*>(p));
      //构造InfoPrinted类的动态对象，作为线程函数参数传递给线程tid2
    //输出100个‘z'

      InfoPrinted* q=new InfoPrinted('z',100);
       pthread_create(&tid2,NULL,&PrintInfo,reinterpret_cast<void*>(q));

       //2-1程序大部分情况下不会输出结果
       //主线程不结束，子线程不会结束，因此子线程可以用主线程的参数
       pthread_join(tid1,NULL);
       pthread_join(tid2,NULL);

    return 0;
}
```

程序大部分清空下不会输出结果

子线程需要使用主线程的数据，如果主线程结束，子线程如何访问这些数据呢？主线程创建完毕就结束，不会等子线程去调用！

解决办法：使用pthread_join()函数，等待子线程结束，保证主线程在子线程结束之后再结束

int pthread_join(pthread_t thread(线程ID), void** retval(接收线程返回值，不需要接收返回值时传递NULL))

线程函数返回值

thread_creat.cpp:10:50: error: cast from ‘void**’ to ‘unsigned int’ loses precision [-fpermissive]
  unsigned int p=reinterpret_cast(&n);

经过学习，在32位系统中上述转换是可以的，因为在32位系统中int为4字节，地址也为4字节，所以可以直接转换，不会损失精度。

但是在64位系统中，int依旧为4字节，但是地址已经变为64位了，所以直接转换会损失精度。

解决办法:

（1）这时就可以使用intptr_t来进行转换，因为intptr_t就是为了跨平台而存在的，总是所在平台的位数，不会损失精度。

（2）另一种方法是使用unsigned long 来进行转换，因为unsigned long 在32位平台是4字节，在64位平台占用8字节，与地址占用字节数相同也不会损失精度，所以也可以

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

#include <cmath>
```

```
//2-2
void* IsPrime(void* n)
{
    unsigned long p=reinterpret_cast<unsigned long>(n);
    unsigned long i=3u,t=(unsigned long)sqrt(p)+1u;
    if(p==2u)
        return reinterpret_cast<void*>(true);
    if(p%2u==0u)
        return reinterpret_cast<void*>(false);

    while(i<=t)
    {
        if(p%i==0u)
            return reinterpret_cast<void*>(false);
            i+=2u;
    }
    return reinterpret_cast<void*>(true);
}

int main()
{
   pthread_t tids[8];
    bool primalities[8];
    int i;
    for(i=0;i<8;i++)
        pthread_create(&tids[i],NULL,&IsPrime,reinterpret_cast<void*>(i+2));
    for(i=0;i<8;i++)
        pthread_join(tids[i],reinterpret_cast<void**>(&primalities[i]));
    for(i=0;i<8;i++)
        std::cout<<primalities[i]<<" "<<endl;


    return 0;
}
```

线程ID

pthread_equal():确定两个线程是否相等

int pthread_equal(pthread_t t1, pthread_t t2);

pthread_self():返回当前线程的ID

pthread_t pthread_self();

```
if(!pthread_equal(pthread_self(),other_tid)
   pthread_join(other_id,NULL);
```

线程属性：精细调整线程行为流程：

创建pthread_atttr_t类型的对象

调用pthread_attr_init()函数初始化线程的缺省属性，传递指向该线程属性对象的指针

对线程属性进行必要修改

调用pthread_create()函数时传递指向线程属性对象的指针

调用pthread_attr_destory()函数清除线程属性对象 ，pthread_attr_t对象本身没有被销毁，因而可以调用pthread_attr_init()函数再次初始化

说明：

单一线程属性对象可以创建多个线程

线程创建后，继续保留线程属性对象本身并没有意义

线程分类

可联线程：缺省设置，终止并不会自动清除（类似僵尸进程），主线程必须调用

pthread_join()获取返回值，此后才能清除

分离线程：结束时自动清除，不能调用pthread_join进行线程同步

可联线程可通过pthread_detach()函数分离，分离线程不能再次联接

int pthread_detach(pthread_t thread)

pthread_attr_setdetachstate()函数：设置线程分离属性

int pthread_attr_setdetachstate(pthread_attr_t* attr,int detachstate);

pthread_attr_getdetachstate()函数：获取线程分离属性

int pthread_attr_getdetachstate(pthread_attr_t* attr,int* detachstate)

```
#include <pthread.h>
#include <iostream>
using namespace std;

void* ThreadFunc(void *arg)
{
    cout<<"hello world"<<endl;
}
int main()
{
    pthread_attr_t attr;
    pthread_t thread;

    //初始化线程属性
    pthread_attr_init(&attr);
    //设置线程属性的分离状态
    pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);

    //创建线程
    pthread_create(&thread,&ThreadFunc,NULL);
    //清除线程属性对象
    pthread_attr_destory(&attr);
```

```
    //无需链接该线程
    return 0;

}
```

线程撤销

pthread_cancel()

int pthread_cancel(pthread_t thread)

已经撤销的线程可以链接，且必须链接，以释放资源，除非其为分离线程

异步可撤销：在其执行的任何时刻都可以撤销

同步可撤销：撤销操作首先进入队列排队，在线程执行到特定撤销点才可撤销

不可撤销：自动忽略

pthread_setcanceltype()设置线程撤销类型

pthread_setcancelstate()设置线程撤销状态（能不能撤销）

使用撤销状态构造临界区

```
//临界区设定,保证里面代码要么没做。要么做完了
void Transfer(double* accounts,int from,int to,double amount)
{
    int ocs;

    //将线程设置为不可撤销，进入临界
    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE,&ocs);

    accounts[to]+=amount;
    accounts[from]-=amount;

    //恢复线程的撤销状态，离开临界区
    pthread_setcancelstate(ocs,NULL);
}
```

线程的局部存储

每个线程的独有数据

进程的多个线程通过全局堆共享全局对象

每个线程拥有独立的栈

让线程拥有数据的独立副本，不能简单复制或读取

pthread_key_create()函数：为线程特定数据创建一个键

pthread_setspecific()函数：设置对应键值

pthread_getspecific()：读取对应键值

```

```

```
#include <pthread.h>
#include <iostream>
using namespace std;
//线程局部存储

static pthread_key_t tlk;//关联线程日志文件指针的键
 void WriteToThreadLog(const char* msg)
 {
     FILE* fp=(FILE*)pthread_getspecific(tlk);
     fprintf(fp,"%d:%s\n",(int)pthread_self(),msg);
 }

void CloseThreadLog(void* fp)
{
    fclose((FILE*)fp);
}
void* ThreadFunc(void *arg)
{
    char filename[255];
    FILE* fp;

    //生成与线程ID配套的日志文件
    sprintf(filename,"thread%d.log",(int)pthread_self());
    fp=fopen(filename,"w");

    //设置线程日志文件指针与键的局部存储关联
    pthread_setspecific(tlk,fp);
    //向日志中写入数据，不同的线程会写入不同的文件
    WriteToThreadLog("Thread starting...");
    return NULL;


}

int main()

{
    int i;
    pthread_t threads[8];
    //创建键，使用CloseThreadLog()函数作为其清除程序
    pthread_key_create(&tlk,CloseThreadLog);

    for(i=0;i<8;++i)
        pthread_create(&threads[i],NULL,ThreadFunc,NULL);
    for(i=0;i<8;++i)
        pthread_join(threads[i],NULL);

    pthread_key_delete(tlk);

    return 0;
}
```

线程清除

销毁线程退出或被撤销时未释放的资源

pthread_cleanup_push()函数：注册线程清除函数

void pthread_cleanup_push(void(*routine)(void), void* arg)

pthread_cleanup_pop()函数：取消线程清除函数注册

void pthread_cleanup_pop(int execute)

```
//线程清除
void* AllocateBuffer(size_t size)
{
    return malloc(size);
}
void DeallocateBuffer(void* buffer)
{
    free(buffer);
}
void DoSomeWork()
{
    void* temp_buffer=AllocateBuffer(1024);
    //注册清除处理函数
    pthread_cleanup_push(DeallocateBuffer,temp_buffer);

    //取消除测，传递值非0,实施清除任务
    pthread_cleanup_pop(1);

}
```

线程清除带来的问题

对象的析构函数在线程退出时可能没有机会被释放，因而线程栈上数据未清除

如何保证线程资源被正确释放?

--定义异常类，线程在准备退出时引发异常，然后在异常处理中退出线程执行

--引发异常时候，C++确保析构函数被调用

```
//线程清除异常处理
class EThreadExit
{
    public:
        EThreadExit(void* ret_val):_thread_ret_val(ret_val){}

        //实际退出线程，使用对象构造时的返回值
         void* DoThreadExit()
         {
             pthread_exit(_thread_ret_val);
         }
    private:
        void* _thread_ret_val;

};

void* ThreadFunc(void* arg)
{
    try
```

```
    {

        if(线程需要退出)
            throw EThreadExit(线程返回值);

    }
    catch(const EThreadExit& e)
    {
        e.DoThreadExit();//执行线程退出动作
    }

    return NULL;

}
```

线程同步机制

资源竞争：多个线程必须进行精确同步

互斥

死锁

信号量

条件变量

互斥：相互独占锁，与二元信号量类似

一次只有一个线程可以锁定一个数据对象，并访问只有该线程释放锁定，其他线程才能访问该数据对象

pthread_mutex_init()函数：初始化互斥

int pthread_mutex_init(pthread_mutex_t* mutex, const pthread_mutexattr_t* mutexattr);

pthread_mutex_destory()函数：互斥销毁

int pthread_mutex_destory(pthread_mutex_t* mutex)

pthread_mutex_lock()函数：互斥枷锁

int pthread_mutex_lock(pthread_mutex_t* mutex)

如果无法锁定，则调用将阻塞，至该互斥被解锁锁定状态

pthread_mutex_trylock()函数：互斥枷锁

int pthread_mutex_trylock(pthread_mutex_t* mutex)

如果无法锁定，则立即返回，不阻塞

pthread_mutex_unlock()函数：互斥解锁

使用互斥流程：

--定义pthread_mutex_t类型的额变量，将其地址作为第一个参数传给pthread_mutex_init()函数;初始化函数只需调用一次

--锁定或尝试锁定该互斥，获得访问权后，执行正常程序代码

并在执行完毕后解锁

互斥属性：

--pshared 属性：进程共享属性

进程内共享

--type属性：互斥类型

普通锁，检错锁，递归锁，默认锁

```
#include <list>
#include <iostream>
#include <pthread.h>


struct Job
{
    Job(int x=0,int y=0):x(x),y(y) {}
    int x,y;
};

//一半临界区代码越短越好，执行时间越短越好，使用C++ stl可能并不是最好选择
std::list<Job*> job_queue;
//定一个互斥对象
pthread_mutex_t job_queue_mutex=PTHREAD_MUTEX_INITIALIZER;

void ProcessJob(Job* job)
{
    std::cout<<"Thread"<<(int)pthread_self();
    std::cout<<"processing("<<job->x<<","<<job->y<<")\n";
}

//处理作业时候需要枷锁
void* DequeueJob(void *arg)
{
    while(true)
    {
        Job* job=NULL;

        //被多个线程共享资源，锁定互斥，才能访问共享资源
        pthread_mutex_lock(&job_queue_mutex);

         if(!job_queue.empty())
       {
        job=job_queue.front();
        job_queue.pop_front();
       }
       pthread_mutex_unlock(&job_queue_mutex);

       if(!job) break;
```

```
        ProcessJob(job);

        delete job;
        job=NULL;


    }


    return NULL;
}

//作业入队需要枷锁
void* EnqueueJob(void* arg)
{
    Job* job=reinterpret_cast<Job*>(arg);
    pthread_mutex_lock(&job_queue_mutex);
    job_queue.push_back(job);

    //入队时候也输出线程ID和作业内容信息
    std::cout<<"Thread"<<(int)pthread_self();
    std::cout<<"enqueueing("<<job->x<<","<<job->y<<")\n";

    pthread_mutex_unlock(&job_queue_mutex);

    return NULL;
}

int main()
{
    int i;
    pthread_t threads[8];
    for(i=0;i<5;++i)
    {
        Job* job=new Job(i+1,(i+1)*2);
        pthread_create(&threads[i],NULL,EnqueueJob,job);
    }

    for(i=5;i<8;++i)
        pthread_create(&threads[i],NULL,DequeueJob,NULL);
    for(i=0;i<8;++i)
        pthread_join(threads[i],NULL);

    return 0;
}
```

死锁：资源被竞争占用，且无法释放

处理策略：更改互斥类型

--创建互斥属性pthread_mutexattr_t型的对象

--调用pthread_mutexattr_init()函数初始化互斥属性对象，传递地址

--调用pthread_mutexattr_setkind_np()函数互斥类型

改成递归锁或检测锁避免死锁发生

--调用pthread_mutexattr_destory()函数销毁互斥属性对象

信号量

问题：如何保证任务队列中有任务可做?

如果队列中没有任务，线程可能退出，后续任务出现时候，没有线程可以执行它

头文件“semaphore.h"

--用于多个线程的同步操作

--操作方法比进程信号量简单

初始化信号量

int sem_init(sem_t* sem,int pshared,unsigned int value)

等待信号量：P操作

int sem_wait(sem_t* sem),在无法操作时阻塞

int sem_trywait(sem_t* sem),立即返回

int sem_timewait(sem_t* sem, const struct timespec* abs_timeout),有时间限制

发布信号量：V操作

int sem_post(sem_t* sem)

销毁信号量

int sem_destory(sem_t* sem)

```
#include <list>
#include <iostream>
#include <pthread.h>
#include "semaphore.h"


struct Job
{
    Job(int x=0,int y=0):x(x),y(y) {}
    int x,y;
};

//一半临界区代码越短越好，执行时间越短越好，使用C++ stl可能并不是最好选择
std::list<Job*> job_queue;
//定一个互斥对象
pthread_mutex_t job_queue_mutex=PTHREAD_MUTEX_INITIALIZER;

//控制作业数目的信号量
sem_t job_queue_count;


void ProcessJob(Job* job)
{
    std::cout<<"Thread"<<(int)pthread_self();
```

```
    std::cout<<"processing("<<job->x<<","<<job->y<<")\n";
}

//处理作业时候需要枷锁
void* DequeueJob(void *arg)
{
    while(true)
    {
        Job* job=NULL;

        sem_wait(&job_queue_count);//等待作业中有作业


        //被多个线程共享资源，锁定互斥，才能访问共享资源
        pthread_mutex_lock(&job_queue_mutex);

         if(!job_queue.empty())
       {
        job=job_queue.front();
        job_queue.pop_front();
       }
       pthread_mutex_unlock(&job_queue_mutex);

       if(!job) break;

        ProcessJob(job);

        delete job;
        job=NULL;


    }


    return NULL;pthread_createpthread_create
}

//作业入队需要枷锁
void* EnqueueJob(void* arg)
{
    Job* job=reinterpret_cast<Job*>(arg);
    pthread_mutex_lock(&job_queue_mutex);
    job_queue.push_back(job);

    //新作业入队，发布信号量
     sem_post(&job_queue_count);


    //入队时候也输出线程ID和作业内容信息
    std::cout<<"Thread"<<(int)pthread_self();
    std::cout<<"enqueueing("<<job->x<<","<<job->y<<")\n";

    pthread_mutex_unlock(&job_queue_mutex);

    return NULL;
}

int main()
```

```
{
    int i;
    pthread_t threads[8];
    if(!job_queue.empty())
        job_queue.clear();

    sem_init(&job_queue_count,0,0);//初始化，非进程共享，初始值为0


    for(i=0;i<5;++i)
    {
        Job* job=new Job(i+1,(i+1)*2);
        pthread_create(&threads[i],NULL,EnqueueJob,job);
    }

    for(i=5;i<8;++i)
        pthread_create(&threads[i],NULL,DequeueJob,NULL);
    for(i=0;i<8;++i)
        pthread_join(threads[i],NULL);//等待结束


    sem_destroy(&job_queue_count);//销毁作业信号量

    return 0;
}
```

条件变量

--互斥用于同步线程对共享数据对象的访问

--条件变量用于在线程间同步共享数据对象的值

初始化条件变量

int pthread_cond_init(pthread_cond_t* cond, const pthread_condattr_t* cond_attr);

销毁条件变量

int pthread_cond_destroy(pthread_cond_t* cond);

广播条件变量

以广播方式唤醒所有等待目标条件变量的线程

int pthread_cond_broadcast(pthread_cond_t* cond)

唤醒条件变量

int pthread_cond_signal(pthread_cond_t* cond)

等待条件变量

int pthread_cond_wait(pthread_cond_t* cond,pthread_mutex_t* mutex);

因为这样的条件变量会被多个线程共享，用互斥保护

互斥以确保函数操作的原子性

C++11线程库

thread：std::thread类与std::this_thread名空间

mutex互斥类，包括std::mutex系列类，std::lock_guard类，std::unique_lock类

condition_variable:条件类，包括std::condition_variable类与std::condition_variable_any类

atomic: std::atomic类与std::atomic_flag类

库

future包含两个承诺类

线程类

支持的线程函数无参数和返回值类型，有无都行，与Linux线程类相比，C++11线程类容易使用

线程局部存储使用thread_local关键字，可派生自己的thread类

常用线程类函数

bool thread::joinable() 判断线程是否可关联

void thread::join() 判断线程结束

void thread::detach() 分离线程

```
//无参数的线程函数
void ThreadFunc()
{
    std::cout<<"Thread ID: "<<std::this_thread::get_id()<<std::endl;
}

int main()
{
    //无参数
    std::thread t(&ThreadFunc);//创建线程
    t.join();//等待线程结束
    return 0;
}
```

```
//双参数的线程函数
void ThreadFunc_1(int a,int b)
{
     std::cout<<"Thread ID: "<<std::this_thread::get_id()<<std::endl;
     std::cout<<a<<"+"<<b<<"="<<a+b<<std::endl;
}

int main()
{
      //双参数
    int m=10,n=20;
    //C++11标准库使用可变参数的模板形式参数列表，线程函数参数个数任意
    std::thread t(&ThreadFunc_1,m,n);
    t.join()
        return 0;

}
```

```
//双参数的函子对象
class Functor
{
    public:
        void operator()(int a,int b)
        {
          std::cout<<"Thread ID: "<<std::this_thread::get_id()<<std::endl;
          std::cout<<a<<"+"<<b<<"="<<a+b<<std::endl;

        }
};


  //双参数的函子对象
    int m=10,n=20;
    std::thread t(Functor(),m,n);
    t.join();
```

```
//使用std::bind()函数绑定对象及普通成员函数
class Worker
{
    public:
        Worker(int a=0,int b=0):_a(a),_b(b){}
    void ThreadFunc_2()
    {
        std::cout<<"Thread ID: "<<std::this_thread::get_id()<<std::endl;
          std::cout<<_a<<"+"<<_b<<"="<<_a+_b<<std::endl;

    }

    private:
        int _a,_b;
};



    //使用std::bind()函数绑定对象及普通成员函数
    Worker worker(10,20);
    std::thread t(std::bind(&Worker::ThreadFunc_2,&worker));
    t.join();
```

互斥类

基本互斥类mutex类

核心成员函数lock(),try_lock()和unlock()

上述成员函数无参数，无返回值

递归互斥 recursive_mutex类

让单个线程对互斥进行多次枷锁和解锁处理

定时互斥 timed_mutex类

在某个时段或者某个时刻前获取互斥

当线程在临界区间操作的时间非常长，可以用定时锁指定时间

定时递归互斥：recursive_timed_mutex类

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

std::mutex x;

void ThreadFunc()
{
    x.lock();//锁定它，其他线程进不去临界区
    std::cout<<std::this_thread::get_id()<<"is entering..."<<std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(3));
    std::cout<<std::this_thread::get_id()<<"is leaving .."<<std::endl;
    x.unlock();
}

int main()
{
    std::vector<std::thread*> v(8);
    for(int i=0;i<8;i++)
        v[i]=new std::thread(ThreadFunc);
    for(int i=0;i<8;i++)
        v[i]->join();

    return 0;
}
```

互斥：容易导致死锁

--若某个线程在临界去内的操作导致异常，有可能无法解锁，从而导致其他线程被永久阻塞

--若临界区代码有多路分支，其中部分分支提前结束，但没有执行解锁操作，其他线程依然被永久阻塞

--当多个线程同时申请多个资源，枷锁次序不同也可能导致死锁

解决办法

资源获取初始化

--使用互斥对象管理类模板自动管理资源

基于作用阈的锁管理类模板：std::lock_guard

--构造时是否枷锁可选，不枷锁时假定当前线程已经获得锁的所有权，析构时候自动解锁，所有权不可转移，对象生存期内不要手动枷锁和解锁

独一锁管理类模板：std::unique_lock

--构造时候是否枷锁可选，对象析构时候如果持有锁会自动解锁，所有权可转移，对象生存期间内手动枷锁和解锁

共享锁管理类模板std::shared_lock

--用于管理可转移和共享所有权的互斥对象

互斥管理策略

--延迟std::defer_lock,构造互斥管理对象时候延迟枷锁操作

--尝试std::try_to_lock 构造互斥管理对象时候尝试枷锁操作，但是不阻塞线程，互斥时候不可用时候立即返回

--接收 std::adopt_lock 假定当前线程已经获得互斥所有权，不再枷锁

--缺省行为，构造互斥管理对象时候没有传递管理策略标签参数，阻塞当前线程至成功获得互斥

互斥的解锁时机

当使用C++11的互斥管理策略时候，只有析构互斥管理对象时候才自动释放互斥，因此要特别注意互斥的持有时间，若线程持有互斥时间过长，有可能极大降低程序效率

--解决方案

使用复合语句或专用辅助函数封装临界区间的操作，动态创建互斥管理对象，并尽早动态释放

多个互斥的竞争访问

--多个线程对多个互斥枷锁时候保持顺序一致性，以避免可能的死锁

--使用std::lock()和std::try_lock()

```
template<typename T>
class Worker
{
    public:
        explicit Worker(int no,T a=0,T b=0):_no(no)
        ,_a(a),_b(b){}

        void ThreadFUnc(T *r)
        {
             //使用复合语句块封装临界区操作，块结束即释放局部对象
             {
                 std::lock_guard<std::mutex> locker(x);//构造对象时候同时枷锁
                 *r=_x+_y;
             }//无需手动解锁，locker对象在析构时候自动解锁
```

```
            std::cout<<"Thread NO: "<<_no<<std::endl;
            std::cout<<_a<<"+"<<_b<<"="<<_a+_b<<std::endl;
        }
        private:
            int _no;
            T _a,_b;
};
```

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


class Account
{
    public:
        explicit Account(double balance):_balance(balance){}
        double GetBalance(){return _balance;}

        void Increase(double amount){_balance+=amount;}
        void Decrease(double amount){_balance-=amount;}

        std::mutex& GetMutex(){return _x;}

        private:
            double _balance;
            std::mutex _x;

};

//避免死锁，使用std::lock()函数锁定多个互斥，不同的锁定顺序不会导致死锁
//枷锁时候有可能引发异常，std::lock()函数会处理该异常
//将解锁此前已经枷锁的部分互斥，然后重新引发该异常
void Transfer(Account& from,Account& to,double amount)
{
    std::unique_lock<std::mutex> locker1(from.GetMutex(),std::adopt_lock);
    std::unique_lock<std::mutex> locker2(to.GetMutex(),std::adopt_lock);
    std::lock(from.GetMutex(),to.GetMutex());

    from.Decrease(amount);
    to.Increase(amount);

}

int main()
{
    Account a1(100.0),a2(200.0);
    //线程参数采用值传递机制，如果要传递引用，调用std::ref()函数
    std::thread t1(Transfer,std::ref(a1),std::ref(a2),10.0);
    std::thread t2(Transfer,std::ref(a2),std::ref(a1),20.0);

    t1.join();
    t2.join();

    return 0;
```

```
}
```

条件变量类

std::condition_variable 类、

--必须与std::unique_lock配合使用

std::condition_variable_any类

--更加通用的条件变量，可以与任意类型式的互斥锁配合使用，相比前者使用时候会有额外的开销

多线程通信同步

--阻塞一个或多个线程到收到来气其他线程的通知，超时或发生虚假唤醒

--两者具有同样的成员函数，且在等待条件变量前都必须要获得相应的锁

成员函数notify_one():通知一个等待线程

void motify_one() noexcept;

成员函数 notify_all():通知全部等待线程

void motify_one() noexcept;

成员函数wait() 阻塞当前线程到被唤醒

template

void wait(Lock& lock);

template<typename Lock,typename Predicate>

void wait(Lock& lock,Predicate p);

成员函数wait_for():阻塞到被唤醒或超过指定时长

成员函数wait_until()

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

std::mutex x;
std::condition_variable cond;//条件变量与互斥连用

bool ready=false;
bool IsReady(){return ready;}

void Run(int no)
{
    std::unique_lock<std::mutex> locker(x);
    while(!ready)//若标志为非true,阻塞当前进程
    {
        cond.wait(locker);//解锁并睡眠，被唤醒后重新枷锁
        std::cout<<"thread"<<no<<"\n";
```

```
        //等价cond.wait(locker,&IsReady)
    }
}

int mian()
{
    std::thread threads[8];
    for(int i=0;i<8;i++)
        threads[i]=std::thread(Run,i);
    std::cout<<"8 threads ready...\n";

    {
        std::unique_lock<std::mutex> locker(x);//互斥枷锁
        ready=true;//设置全局标志为true
        cond.notify_all();//唤醒所有线程

    }//离开作用阈，自动解锁，可将次复合语句块实现为函数

    for(auto &t:threads)
        t.join();

    return 0;
}
```

原子型

--使用atomic模板定义原子对象

使用预定义标准原子型 atomin_bool,atomic_char,atomic_int等

意义：轻量级，支持单变量上的原子操作

```
#include <atomic>
#include <iostream>
#include <thread>

int n=0;
std::atomic<int> a(0);
void AddAtomic(int m)
{
    while(m--)
        a.fetch_add(1);
}

void Add(int m)
{
    while(m--)
        ++n;
}

int main()
{
    std::thread ts1[8],ts2[8];
    for(auto &t:ts1)
        t=std::move(std::thread(AddAtomic,1000000));
```

```
    for(auto &t:ts2)
        t=std::move(std::thread(Add,1000000));
    for(auto &t:ts1) t.join();
    for(auto &t:ts2) t.join();

    //输出结果，a固定，而n多次运行结果可能不同
    std::cout<<"a= "<<a<<std::endl;
    std::cout<<"n= "<<n<<std::endl;

    return 0;
}
```

期许与承诺

线程返回值

--为支持跨平台，thread类无属性字段保存线程函数的返回值

解决方案：

--使用指针型的函数参数

--使用期许：std::future类模板

--使用承诺：std::promise类模板

```
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>
#include <tuple>

//劳工线程类模板
 std::mutex x;

 template<typename T>
 class Worker
 {
     public:
        explicit Worker(int no,T a=0,T b=0):_no(no),_a(a),_b(b){}
        void ThreadFunc(T* r)
        {
            x.lock();
            *r=_a+_b;
            x.unlock();
        }
        private:
            int _no;//线程编号，非线程ID
            T _a,_b;//保存在线程中的待处理数据
 };


int main()
{
    //定义能够存储8个三元组的向量v，元组首元素为指向劳工对象的指针
    std::vector<std::tuple<Worker<int>*, int, std::thread*>> v(8);
```

```
    //构造三元组向量，三元编号顺次为0,1,2
    for(int i=0;i<8;i++)
        v[i]=std::make_tuple(new Worker<int>(i,i+1,i+2),0,nullptr);

    //输出处理前结果，使用std::get<n>(v[i])获取向量的第i个元祖的第n个元素
    //三元编号顺次为0,1,2,因而1号元保存的将是劳工对象运算后的额结果
    for(int i=0;i<8;i++)
        std::cout<<"NO. "<<i<<":result= "<<std::get<1>(v[i])<<std::endl;



  //动态构造线程对象，并保存到向量的第i个三元组中
  //传递三元组的1号元地址，即将该地址作为线程函数的额参数
  //线程将在执行时候将结果写入地址
  //此性质由绑定函数std::bind()使用占位符号std::placeholders:_1指定
  //线程对象为2号元，即三元组的最后一个元素
    for(int i=0;i<8;i++)
    {
        auto f=std::bind(&Worker<int>::ThreadFunc,
        std::get<0>(v[i]),std::placeholders::_1);

        std::get<2>(v[i])=new std::thread(f,&std::get<1>(v[i]));

    }

    for(int i=0;i<8;i++)
    {
        //等待线程结束
        std::get<2>(v[i])->join();
        //销毁劳工对象
        delete std::get<0>(v[i]),std::get<0>(v[i])=nullptr;
        //销毁线程对象
        delete std::get<2>(v[i]),std::get<2>(v[i])=nullptr;

    }

    //输出线程计算结果
    for(int i=0;i<8;i++)
     std::cout<<"No "<<i<<":result= "<<std::get<1>(v[i])<<std::endl;

    return 0;
}
```

期许

std::future类模板

--目的：获取异步操作结果，延迟引发线程的异步操作异常’

使用方法

--定义期许模板类的期许对象

--使用std::async()函数的返回值初始化

--调用期许对象的成员函数get()获取线程的返回值

```
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>
#include <future>


unsigned long int Cal(short int n)
{
    unsigned long int r=1;
    if(n>20)
        throw std::range_error("the number is too big ");

    for(short int i=2;i<=n;i++)
      r*=i;

    return r;
}

int main()
{
    short int n=20;
    //启动异步线程，执行后台计算任务，并返回std::future对象
    std::future<unsigned long int> f=std::async(Cal,n);

    try
    {
       //获取线程返回值，若线程已经结束，立即返回，否则等待线程计算完毕
       //若线程引发异常，则延迟到std::future::get()或
       //std::future::wait()调用时候引发
       unsigned long int r=f.get();
       std::cout<<n<<"!= "<<r<<std::endl;

    }
    catch(const std::exception& e)
    {
        std::cerr << e.what() << '\n';
    }
    return 0;
}
```

承诺

std::promise类模板

--目的：承诺对象允许期许对象获取线程对象创建的线程返回值

使用方法

--创建承诺std::promise对象

--获取该承诺对象的相关期许std::future对象

--创建线程对象，并传递承诺对象

--线程函数内部通过承诺模板类的成员函数set_value().

set_value_at_thread_exit(),set_exception()或set_exception_at_thread_exit()设置值或异常

主线程期望，期许，承诺

```
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>
#include <future>


unsigned long int Cal(short int n)
{
    unsigned long int r=1;
    if(n>20)
        throw std::range_error("the number is too big ");

    for(short int i=2;i<=n;i++)
      r*=i;

    return r;
}

//Cal()函数的包装函数原型
void Dol(std::promise<unsigned long int> &&promise,
short int n)
{
    try
    {
        //设置线程返回值，共期许对象获取
        promise.set_value(Cal(n));
    }
    catch(...)
    {
        //捕获全部异常，并在期许线程返回值时重新引发
        promise.set_exception(std::current_exception());

    }
}

int main()
{
    short int n=20;
    //启动异步线程，执行后台计算任务，并返回std::future对象
    //std::future<unsigned long int> f=std::async(Cal,n);

    //创建承诺对象
    std::promise<unsigned long int> p;
    //获取相关期许对象
    std::future<unsigned long int> f=p.get_future();
    //启动线程，执行包装函数
    std::thread t(Dol,std::move(p),n);
    //分离线程，不管它了，后台跑去把
    t.detach();


        try
    {
       //获取线程返回值，若线程已经结束，立即返回，否则等待线程计算完毕
```

```
        //若线程引发异常，则延迟到std::future::get()或
        //std::future::wait()调用时候引发
        unsigned long int r=f.get();
        std::cout<<n<<"!= "<<r<<std::endl;

    }
    catch(const std::exception& e)
    {
        std::cerr << e.what() << '\n';
    }
    return 0;
}
```