1 并发基本概念与实现，进程、线程概念

1.1并发

1、一个程序处理多个独立任务，以往的计算机在一个时间片内只能执行一个任务。有操作系统进行任务调度进行任务上下文切换，完成伪并发。这种任务切换是需要时间开销的，操作系统需要保存切换时的状态，执行进度等，复原时也需要时间消耗。
2、硬件的发展出现了处理器(多核)的处理器，可以实现真正的并行。并发的主要目的是提高性能。
3、可执行程序本质上是一个磁盘上的文件，可以直接被执行。可执行程序执行时会创建一个进程.
4、线程：每一进程均有一个主线程，主线程是唯一的。
5、多线程程序开发是实际的需要，开发过程也会有难度。
6、多进程并发：进程之间通信(同一个server上可以使用管道，文件，消息队列，共享内存)，不同的server之间的程序可以使用socket通信。
7、多线程并发：更像是轻量级的多进程并发，每个线程存在自己的独立运行路径。进程有自己的独立地址空间，进程中所有线程共享内存(全局变量、静态变量、堆内存)，多线程的开销远小于多进程的开销。
8、c++11的标准线程库：以往的windows与linux均使用不同的线程实现比如linux的"pthread.h"，编译的时候需要链接动态库。c++11开始在语言层面实现多线程，<thread>可实现跨平台。

2 线程的启动、结束，创建线程的方法

1、程序运行开始后建立一个进程，进程中的主线程随之建立并运行。主线程将mian函数执行完毕，进程也结束。这里需要注意的是进程是否执行完毕的标志是看主线程是否执行完毕。
2、一般情况下，主线程执行完毕，若其他子线程还未执行结束，会被操作系统强行终止。这意味着要保持子线程的运行必须保证主线程最后执行结束。

```
#include <iostream>
#include <thread>
using namespace std;
void doSomeWork()
{
    cout << "doing some works" << endl;
}
class backGround
{
public:
    void operator()()
    {

    }
};
int main()
{
    std::thread my_thread1(doSomeWork);
    backGround f;
    thread my_thread2(f);
    cout << std::thread::hardware_concurrency() << endl;
    my_thread1.detach();
    my_thread2.join();
    return 0;
}
```

以上使用两种方式创建线程，一种是将函数名传入thread类的有参构造，另一种先实例化一个类的对象，再将该对象传入thread的有参构造。 `detach` ，与 `join` 不同之处在于， `join` 会阻塞主线程，主线程等待my_thread2执行结束，一起退出。传统的多线程设计，主线程应该等待子线程执行结束，但c++11的新标准中，允许将主线程与子线程分离。调用detach之后子线程会与主线程失去联系，该线程会在后台执行，被c++运行时库接管，运行时库会清理该线程的资源。这里注意一但使用detach不能再使用join，反过来也一样join之后不能再detach。并且join与detach只能使用一次。
可以在使用join之前使用thread类的 `joinable` 函数，如果该线程可以join返回true，这样有效避免了错误写法。

```
class backGround
{
public:
    backGround(int &i) : m_i(i),m_a(10){};
    void operator()()
    {
        cout << "m_i = " << m_i << endl;
        cout << "m_i = " << m_i << endl;
        cout << "m_i = " << m_i << endl;
        cout << "m_i = " << m_i << endl;
    }
    int &m_i;
    const int m_a;
};
int main()
{
    std::thread my_thread1(doSomeWork); // 线程建立，也是线程的入口，这时线程开始执行
    int a = 10;
    backGround f(a);
    thread my_thread2(f);
    // cout << std::thread::hardware_concurrency() << endl;
    my_thread2.detach();
    my_thread1.detach();
    cout << "I love China" << endl;
    return 0;
}
```

以上程序将background类增加两个成员变量，分别是一个引用 `m_i` ，一个常量 `m_a` 。在有参构造函数中初始化这两个成员变量。通过上面的程序可以看出对象f的成员变量 `f.m_i` 引用的是主线程的局部变量 `a` ，detach之后主线程一但执行完毕，变量 `a` 会被销毁，f的成员变量 `m_i` 引用内容被销毁，导致结果出错。这里还有一个注意点， `f` 也是主线程在栈区创建的临时对象，主线程结束后 `f` 也会被销毁，那 `my_thread2` 执行会出错吗？事实上，不会的，因为 `thread my_thread2(f)` 这里 `f` 仅仅是一个参数，子线程会调用类的拷贝构造函数，复制 `f` 的内容到子线程中。所以 `f` 中不能含有引用，指针等，否则可能会导致出错。除了使用函数与类创建线程，还可以使用lambda表达式，来创建线程。

```
int main()
{
    int a = 10;

    auto threadlam = [&]() -> void {
        cout << a << endl;
        cout << a << endl;
        cout << a << endl;
        cout << a << endl;
        cout << "线程3开始执行" << endl;
        cout << "线程3执行结束" << endl;
    };
    thread myThread3(threadlam);
    myThread3.detach();
    cout << "I love China" << endl;
    return 0;
}
```

以上代码存在问题， `threadlam = [&]() -> void` 以引用方式捕获变量，并且子线程会与主线程分离，一但主线程运行结束， `a` 会被销毁，子线程中的 `a` 值会“不可预测”。

3 线程传参，detach注意事项，成员函数做线程函数

```
void myprint(const int &i, char *pmybuf)
{
    cout << "i = " << i << endl;
    cout << "buff = " << pmybuf << endl;
}
int main()
{
    // 临时对象作为线程参数
    int var1 = 1;
    int &var2 = var1;
    char myBuff[] = "this is a test";
    thread mythread(myprint, var2, myBuff);
    mythread.detach();
    cout << "I love China !" << endl;
    return 0;
}
```

以上是线程传参的写法，子线程中线程传参后与主线程分离，子线程接收2个参数分别是 `i` 与 `pmybuf`，这里注意传参的方式。在打印 `var2` 的地址与 `i` 的地址发现，两个变量并非是同一个地址。但是第二个参数以指针的方式传入，当主线程退出的时候，该指针指向的栈内存会被释放，指针失效。

```
void myprint(int i, const string &pmybuff)
{
    cout << "i = " << i << endl;
    cout << "buff addr= " << (void *)pmybuff.c_str() << endl;
}
```

修改myprint函数的传参方式，使用string的隐式类型转换，打印出来的str的字符数组的地址与原字符数组地址不同。但是这里还有一个问题，子线程detach之后，若主线程执行完毕，buff会立刻被收回，该隐式构造若未完成，会出现问题。

```
class A
{
public:
    int m_i;
    A(int i) : m_i(i) { cout << "A的有参构造函数执行 " << this << " thread id = " << this_thread:
    A(const A &a) : m_i(a.m_i) { cout << "A的拷贝构造函数执行 " << this << " thread id = " << thi
    ~A() { cout << "A的析构函数执行 " << this << " thread id = " << this_thread::get_id() << endl
void myprint(int i, const A &a)
{
}
int main()
{
    cout << "主线程id:" << this_thread::get_id() << endl;
    int var1 = 1;
    int &var2 = var1;
    char myBuff[] = "this is a test";
    thread mythread(myprint, var2, A(var1));
    // A a(10);
    // thread mythread(myprint, var2, std::ref(a));
    cout << "子线程id:" << mythread.get_id() << endl;
    mythread.join();
    return 0;
}
};
```

使用上述测试代码可发现， `const string &pmybuff` 进行构造的时候是在主线程中进行的，而析构函数是在子线程中进行的。这里之所以要是有const关键字，是因为pmybuff引用了临时对象A(var1)，不使用

const会报错。这是虽然是引用方式传参，根据之前的介绍可以知道，引用对象的地址与被引用的地址不同，并且调用了拷贝构造函数，所以这里不是真引用。这里有一个简单的原则，如果是内置数据类型直接使用值传递，如果是复杂数据类型，使用创建临时对象的方法。这种方法运行时会发现多次调用类A的拷贝构造函数，如果使用 `std::ref(a)` 进行传参，则不会调用拷贝构造函数，此时的形参 `const A& a` 是真的引用主线程中的a但是这时不能使用detach。

```
A a(100);
thread mythread2(&A::thread_work,a,10);
mythread2.join();
```

以上是使用类成员对象作为线程入口，这里注意要创建一个对象 `a` 。下面给出一个经典的例子来说明线程究竟是如何创建并传参的。

```cpp
#include <iostream>
#include <thread>
class String
{
public:
    // 0
    String(const char *cstr) { std::cout << "String(const char* cstr)" << std::endl; }

    // 1
    String(const String &v)
    {
        std::cout << "String(const String& v)" << std::endl;
    }

    // 2
    String(const String &&v) noexcept
    {
        std::cout << "String(const String&& v)" << std::endl;
    }

    // 3
    String &operator=(const String &v)
    {
        std::cout << "String& operator=(const String& v)" << std::endl;
        return *this;
    }
};
void test(int i, const String &s) {}
int main()
{
      String s("hello");
      std::cout << "----------------" << std::endl;

      // 输出 1, 2
      std::thread t1(test, 3, s);
      t1.join();
      std::cout << "----------------" << std::endl;

      // 输出 2, 2
      std::thread t2(test, 3, std::move(s));
      t2.join();
      std::cout << "----------------" << std::endl;
```

```
        // 只输出 0
        std::thread t3(test, 3, "hello");
        t3.join();
        std::cout << "----------------" << std::endl;

        // 无输出
        std::thread t4(test, 3, std::ref(s));
        std::cout << "----------------" << std::endl;
        t4.join();
    return 0;
}
```

String类里面我们写了有参构造，拷贝构造，移动构造，以及重载了赋值运算符。这里需要说明两点：1.线程传参的时候回直接将参数拷贝至子线程内存中，并且该参数在子线程中变成右值。2.并且线程传参的时候会忽略调用函数形参的类型(引用)。由于内存中的是右值，所以需要使用const关键字进行左值引用。

4 创建多个线程，数据共享问题分析

4.1 创建多个线程，示例代码如下

```
vector<thread> mythreads;
for (int i = 0; i < 10; ++i)
{
        // 线程创建开始执行
    mythreads.push_back(thread(myprint, i));
}
for (auto it = mythreads.begin(); it != mythreads.end(); ++it)
{
        // 阻塞主线程，等待所有线程执行完毕
    it->join();
}
```

以上代码使用容器创建并管理线程，将线程放到容器里。

4.2 数据共享问题分析
1.只读数据：多线程对于全局变量数据只读，不需要特殊处理，是线程安全的。
2.读写数据：加入有多个线程对全局变量进行读操作，多个线程对变量进行写操作。分析可知有3种情况，读-读，读-写，写-写。这里只有第一种是线程安全的。
4.3 互斥量mutex的基本概念
mutex是一个类对象，线程尝试使用成员函数lock()对共享变量加锁，并且只有一个线程加锁成功，如果

没有加锁成功线程会阻塞。互斥量使用要注意边界问题，包括的代码段要合理。lock()与unlock()是成对使用的。互斥量示例代码如下。

```cpp
#include <iostream>
#include <vector>
#include <thread>
#include <list>
#include <mutex>
#include <unistd.h>
using namespace std;
class A
{
public:
    // 定义成员函数作为线程入口函数
    void inMsgQueue()
    {
        for (int i = 0; i < 100000; ++i)
        {
            cout << "inMsgQueue:" << i << endl;
            this->myMutex.lock();
            this->msgRcvQueue.push_back(i);
            this->myMutex.unlock();
        }
    }
    bool outMsgLULProc(int &i)
    {
        this->myMutex.lock();
        if (!this->msgRcvQueue.empty())
        {
            i = this->msgRcvQueue.front();
            this->msgRcvQueue.pop_front();
            this->myMutex.unlock();
            return true;
        }
        this->myMutex.unlock();
        return false;
    }
    void outMsgQueue()
    {
        int command = 0;
        for (int i = 0; i < 100000; ++i)
        {
            bool result = outMsgLULProc(command);
            if (result == true)
            {
                cout << "outMsgQueue取出命令成功, command: " << command << endl;
```

```
            }
            else
            {
                cout << "msgRcvQueue队列为空" << endl;
            }
        }
        cout << "end" << endl;
    }

private:
    list<int> msgRcvQueue; // 使用stl的容器list构建一个消息队列,共享变量
    mutex myMutex;         // 声明一个互斥量
};
int main()
{
    A myobj;
    thread inMyObj(&A::inMsgQueue, &myobj);
    thread outMyObj(&A::outMsgQueue, &myobj);
    inMyObj.join();
    outMyObj.join();
    return 0;
}
```

上面说明了，lock与unlock必须要成对使用，否则程序会有死锁等问题，这里有一个很好的替代是类模板lock_guard，示例代码如下

```
    bool outMsgLULProc(int &i)
    {
        std::lock_guard<mutex> guard(mutex);
        if (!this->msgRcvQueue.empty())
        {
            i = this->msgRcvQueue.front();
            this->msgRcvQueue.pop_front();
            return true;
        }
        return false;
    }
```

其原理也很简单，局部对象guard被构造的时候相当于lock了一次，而该函数退出后，该对象被析构调用一次unlock。

4.4 死锁：考虑一种情况，存在两个线程A与线程B两个互斥量mutex1，mutex2。线程A已经锁定mutext1，尝试锁定mutex2。而线程B锁定mutex2，尝试锁定mutex1，此时会造成程序死锁问题。解决

这个问题的关键在于，两个线程加锁的循序只要一致就不会出现死锁的问题。另一个解决方法是使用 `std::lock()` 模板函数，输入参数至少有两个互斥量，它会一次锁住两个互斥量，不存在多线程中锁的顺序导致死锁问题。基本原理：如果输入的互斥量中存在一个没有锁住，它会立即释放已经锁住的互斥量，并且尝试同时锁住两个互斥量，只有同时锁住两个互斥量才会往下进行代码。这里注意依然要使用unlock进行解锁，并且解锁的顺序是无关紧要的。可以配合 `lock_guard` 的第二个参数 `std::adopt_lock` 来省略 `unlock` ，示例代码如下。

```
bool outMsgLULProc(int &i)
{
    // this->myMutex1.lock();
    // this->myMutex2.lock();
    std::lock(myMutex2,myMutex1);
    lock_guard<mutex> guard1(myMutex1,std::adopt_lock);
    lock_guard<mutex> guard2(myMutex2,std::adopt_lock);
    if (!this->msgRcvQueue.empty())
    {
        i = this->msgRcvQueue.front();
        this->msgRcvQueue.pop_front();
        // this->myMutex1.unlock();
        // this->myMutex2.unlock();
        return true;
    }
    // this->myMutex1.unlock();
    // this->myMutex2.unlock();
    return false;
}
```

这里的 `std::adopt_lock` 是一个结构体对象，起作用仅仅是说明已经lock了不需要再次lock。这里需要注意，已上所讲的std::lock()仅仅适合多个互斥量一起上锁的情况，如果两个互斥量之间存在代码，则不可以使用。

5 unique_lock详解

1、unique_lock是一个类模板，unique_lock比lock_guard效率低一些，占用内存会多一些，但是更灵活一些。

2、unique_lock的第二个参数： `std::unique_lock<mutex> guard1(this->myMutex1,std::adopt_lock)` ，第二个参数 `std::adopt_lock` 上一节已经讲过了，就是不再对相应的互斥量lock。

3、try_to_lock，使用该参数的前提是互斥量不能已经上锁，如果没有上锁成功，并不会阻塞，而是直接返回。调用unique_lock的成员函数own_lock可查看是否上锁成功。

4、defer_lock是推迟上锁，使用该参数互斥量同样不能上锁，并且可以搭配lock()成员函数对共享变量进行操作，使用unlock对非共享变量进行操作。try_lock()成员函数判断是否对互斥量上锁成功。

5、release()函数，返回它管理的mutex的指针，并释放所有权，也就是说，unique_lock与mutex没有关系了。如果原来的mutex处于加锁状态，调用release之后需要再对其unlock
6、unique_lock与mutex是一一对应的，一般是一个mutex一应一个unique_lock，这就意味着一个unique_lock1拥有mutex1的所有权。但是unique_lock可以将所有权转移，但不能复制！unique_lock是禁用拷贝构函数的。可以使用移动构造 `std::move()`，将所有权转移。
总结unique_lock：与智能指针很像，本质上是一个管理mutex的一个工具，通过成员变量可以很"智能"地使用mutex，达到线程安全的目的。

6 单例设计模式，共享数据分析，call_one

1、所谓的设计模式就是代码的写法：程序灵活，维护起来方便，但是别人接管、阅读代码可能会有困难。总的来说设计模式就是写代码的经验。
2、单例设计模式：是设计模式中使用频率较高的。单例指的是在一个项目中，某些类只能实例化一个对象，比如项目的配置仅仅需要一次。以下是单例设计模式的示例代码。

```cpp
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;
std::mutex resource_myCAS;
class myCAS // 单例类
{
private:
    myCAS() {} // 构造函数私有化
private:
    static myCAS *mInstance; // 静态成员变量
public:
    static myCAS *getInstance()
    {
        if (mInstance == nullptr) // 双重锁定，可以提高概率
        {
            std::unique_lock<mutex> myCASLock(resource_myCAS);
            if (mInstance == nullptr)
            {
                mInstance = new myCAS();
                static Gar cl; // 定义
            }
        }

        return mInstance;
    }
    class Gar // 生命用于析构的类
    {
    public:
        ~Gar()
        {
            if (myCAS::mInstance != nullptr)
            {
                delete myCAS::mInstance;
                myCAS::mInstance = nullptr;
            }
        }
    };
    void fun()
    {
        cout << "mInstance addr:" << myCAS::mInstance << endl;
    }
};
```

```
myCAS *myCAS::mInstance = nullptr;
int main()
{
    myCAS *obj1 = myCAS::getInstance();
    myCAS *obj2 = myCAS::getInstance();
    obj1->fun();
    obj2->fun();
    return 0;
}
```

`myCAS` 是单例设计模式下的类，其实现依靠静态成员变量与静态成员函数，保证了该静态成员 `mInstance` 变量仅仅会生成一次。

3、call_once函数是c++11中引入的，该函数的第二个参数是一个函数指针。其功能为保证传入的函数仅仅会被调用一次。这样的话call_once具备了互斥量的能力。需要与 `std::once_flag` 配合使用，成功调用call_once后，call_once会将该标记设置为已调用状态。后续再调用call_once则传入的函数将不会被再次调用。

7 condition_variable、wait、notify_one、notify_all

1.condition_variable、wait、notify_one、notify_all：线程A等待条件满足，线程B向任务队列中添加任务。condition_variable是一个类，需要与mutex配合使用。

2.wait:是条件变量的一个成员函数，如果第二个参数的返回值是false，该函数会将互斥量解锁(主动放弃互斥量)，并开始阻塞，直到其他线程调用notify_one将该线程解除阻塞。如果wait没有第二个参数，会默认第二个参数返回false。阻塞被唤醒的wait会不断地尝试获取互斥量，如果不能立即获取互斥量会一直重复下去获取互斥量。当获取互斥量后，wait尝试判断第二参数的谓词是否为true，如果是false将会主动放弃互斥量。

3.notify_one():仅仅会唤醒一个被条件变量阻塞的线程。

4.notify_all():唤醒多个被条件变量阻塞的线程，这里将会存在多个线程抢互斥量。但是，其它线程不再被互斥量阻塞。所以这里必须要使用wait的第二参数，或者使用 `while(condtion){wait(mutex)}` ，这种方式，防止虚假唤醒。示例代码如下

```cpp
#include <iostream>
#include <vector>
#include <thread>
#include <list>
#include <mutex>
#include <condition_variable>
#include <unistd.h>
using namespace std;
class A
{
public:
    // 定义成员函数作为线程入口函数
    void inMsgQueue()
    {
        for (int i = 0; i < 100000; ++i)
        {
            std::unique_lock<mutex> guard1(this->myMutex1, std::defer_lock); // 没有加锁的mutex
            guard1.lock();
            cout << "向任务队列插入command:" << i << endl;
            msgRcvQueue.push_back(i);
            my_cond.notify_all(); // 唤醒被条件变量阻塞的线程
            guard1.unlock();      // 唤醒其他线程后及时释放互斥量
        }
    }
    bool outMsgLULProc(int &i)
    {
        // std::unique_lock<mutex> guard1(this->myMutex1);
        if (!this->msgRcvQueue.empty()) // 双重锁定，双重检查
        {
            this->myMutex1.lock();
            if (!this->msgRcvQueue.empty())
            {
                i = this->msgRcvQueue.front();
                this->msgRcvQueue.pop_front();
                this->myMutex1.unlock();
                // this->myMutex2.unlock();
                return true;
            }
            this->myMutex1.unlock();
            // this->myMutex2.unlock();
        }
        return false;
    }
```

```
    void outMsgQueue()
    {
        int command = 0;
        while (true)
        {
            std::unique_lock<mutex> guard(myMutex1);
            // wait是条件变量的一个成员函数，如果第二个参数的返回值是false，
            // 该函数会将互斥量解锁(主动放弃互斥量)，并开始阻塞，直到其他线程调用notify_one将该线程解除
            // 如果wait没有第二个参数，会默认第二个参数返回false
            // 阻塞被唤醒的wait会不断地尝试获取互斥量，如果不能立即获取互斥量会一直重复下去获取互斥量
            // 当获取互斥量后，wait尝试判断第二参数的谓词是否为true，如果是false将会主动放弃互斥量
            my_cond.wait(guard, [this]
                         {
                if(!this->msgRcvQueue.empty())
                    return true;
                else
                    return false; });
            // 当程序走到当前位置，一定获取了互斥量，并且任务队列中至少有一个命令
            command = this->msgRcvQueue.front();
            msgRcvQueue.pop_front();
            cout << "id:" << this_thread::get_id() << "  outMsgQueue()执行成功，取出command:" <<
            guard.unlock();
        }
    }

private:
    list<int> msgRcvQueue; // 使用stl的容器list构建一个消息队列
    mutex myMutex1;        // 声明一个互斥量
    mutex myMutex2;
    std::condition_variable my_cond; // 声明一个条件变量
};
int main()
{
    A myobj;
    thread inMyObj(&A::inMsgQueue, &myobj);
    thread outMyObj(&A::outMsgQueue, &myobj);
    thread outMyObj1(&A::outMsgQueue, &myobj);
    inMyObj.join();
    outMyObj.join();
    outMyObj1.join();
    return 0;
}
```

8 async,future,packaged_task,promise

1、asyanc，future创建后台任务并返回值：考虑如果希望线程结束后返回值

`std::asyanc` 是一个模板函数，用来启动一个异步任务(创建一个线程，并执行入口函数)，返回一个 `std::future` 的对象，该对象含有线程入口函数返回的结果。通过调用future的成员函数可获得线程的返回结果。有人也称future提供了一种访问异步结果的机制。如果不调用wait或者get函数，主线程依然会等待子线程执行结束后一起退出。future是禁止拷贝构造的。get函数使用的是移动语义，一旦调用get，future的内容会移动到接收结果的对象里，所以get不能重复调用。

2、可以额外向 `std::async` 传递参数 `std::luanch` (枚举类型)， `std::luanch::deferred` :表示线程入口函数调用被延迟到调用 `std::futuer::wait` 或者 `std::futuer::get` 函数调用时才会执行，实际上线程根本不会被创建。延迟调用不会创建子线程，在主线程里调用了一个函数而已。

3、 `std::launch::async` 也可以使用该参数作为async的传入参数，async函数默认使用 `std::launch::async | std::luanch::deferred` ，这种模式会使系统自己决定。 `std::async` 与 `std::thread` 的主要区别是：如果系统资源紧张 `std::thread` 创建线程会失败，系统会报错。而 `std::async` 默认情况下不会创建线程，并且等待后续调用get，如果不调用，直接不运行。示例代码如下。

```cpp
#include <iostream>
#include <thread>
#include <mutex>
#include <future>
using namespace std;
class A
{
public:
    int mythread(int par)
    {
        cout << "thread:" << this_thread::get_id() << " start!" << endl;
        cout << "par = " << par << endl;
        chrono::milliseconds dura(5000);
        this_thread::sleep_for(dura);
        return par;
    }
};
int mythread()
{
    cout << "thread:" << this_thread::get_id() << " start!" << endl;
    chrono::milliseconds dura(5000);
    this_thread::sleep_for(dura);
    return 5;
}
int main()
{
    cout << "main thread id:" << this_thread::get_id() << " start!" << endl;

    // std::future<int> result = std::async(mythread);
    A a;
    // std::future<int> result = std::async(&A::mythread, &a, 2);
    // std::future<int> result = std::async(std::launch::deferred, &A::mythread, &a, 2);
    std::future<int> result = std::async(std::launch::async, &A::mythread, &a, 2);
    int def = 0;
    def = 5;
    cout << "continue....." << endl;
    cout << "result = " << result.get() << endl; // 阻塞等待mythread执行完毕,get函数只能调用一次
    // result.wait();
    cout << "I love China!" << endl;
    return 0;
}
```

1、 std::packaged_task :是一个模板类，模板参数是可调用对象，功能为打包任务，将任务封装。将各种可调用对象封装，方便作为线程入口函数。 std::packaged_task 模板类是禁止拷贝构造的。示例代码如下

```
int mythread(int p)
{
    cout << "thread:" << this_thread::get_id() << " start!" << endl;
    chrono::milliseconds dura(5000);
    this_thread::sleep_for(dura);
    return 5;
}
int main()
{
    cout << "main thread id:" << this_thread::get_id() << " start!" << endl;

    std::packaged_task<int(int)> mpt1(mythread);
    thread t1(std::ref(mpt1), 1);
    t1.detach();
    future<int> result1 = mpt1.get_future(); // 这使得future里面包含了线程入口函数的执行结果
    cout << result1.get() << endl;
    // t1.join();
    cout << "I love China!" << endl;
    std::packaged_task<int(int)> mpt2(mythread);
    mpt2(100); // packaged_task的对象可以直接调用，其实就是函数调用
    future<int> result2 = mpt2.get_future(); // 这使得future里面包含了线程入口函数的执行结果
    cout << result2.get() << endl;
    // 还需说明的是packaged_task的拷贝函数被禁用，是禁止拷贝的，可以使用移动语义
    return 0;
}
```

2、promise:是一个类模板，其功能为：能够给某个线程赋值，然后可以在其他线程中，把这个值取出来。promise禁用了拷贝构造函数，在进行线程传参的时候，不可以使用移动构造，必须使用左值引用传参。示例代码如下。

```
int mythread1(int p)
{
    cout << "thread:" << this_thread::get_id() << " start!" << endl;
    chrono::milliseconds dura(5000);
    this_thread::sleep_for(dura);
    return 5;
}
void mythread(std::promise<int> &pro, int a, int b)
{
    int res = a + b;
    chrono::milliseconds dura(2000);
    this_thread::sleep_for(dura);
    pro.set_value(res);
}
void mythread2(future<int> &fut)
{
    cout << "res = " << fut.get() << endl;
}
int main()
{
    cout << "main thread id:" << this_thread::get_id() << " start!" << endl;
    std::promise<int> prom;
    thread t1(mythread, std::ref(prom), 2, 3);
    t1.detach();
    // 获取结果值
    future<int> res = prom.get_future();
    // cout << "res = " << res.get() << endl;
    // future<int> res;
    thread t2(mythread2, std::ref(res));
    t2.join();
    return 0;
}
```

1、原子操作(atomic):cpp程序执行一条代码语句时可能cpu进行的操作要分解成很多指令，比如 `++a` ，可能涉及对变量的寻址，自增，保存等。这样的话如果两个线程对 a 进行前缀递增，结果是不可预料的。所以需要互斥量。而原子操作是一种无互斥量的线程安全技术，线程执行的时候原子变量的操作不会被打断，在效率上比互斥量效率较高。但是原子操作针对的仅仅是一个变量而不是整个代码段。
2、原子操作：其结果是执行完成或者未开始执行。atomic是一个模板类，改模板类重载了很多运算符，算术、赋值、逻辑，使用这些运算符是原值操作的。

```cpp
#include <iostream>
#include <thread>
#include <mutex>
#include <future>
#include<atomic>
using namespace std;
std::mutex mVar;
// int shareVar = 0;
std::atomic<int> shareVar(0);
// void wThread() // 非原子操作，通过互斥量达到线程安全
// {
//     for (int i = 0; i < 100000; ++i)
//     {
//         mVar.lock();
//         ++shareVar;
//         mVar.unlock();
//     }
//     return;
// }
void wThread()
{
    for (int i = 0; i < 100000; ++i)
    {
        ++shareVar;  // ++是atomic类中的原子操作
    }
    return;
}
int main()
{
    thread t1(wThread);
    thread t2(wThread);
    t1.join();
    t2.join();
    cout << "shareVar = " << shareVar << endl;
    return 0;
}
```

考虑将上述的示例代码中 `++shareVar` 修改成 `shareVar = shareVar+1` ，是否是线程安全的，答案是否。一般来说原子操作支持++、--、+=等操作符，其他操作可能是不支持的。

9 c++线程池