Chapter1.1

1. 任务切换：这种机器只能在某一时刻执行一个任务，不过它可以每秒进行多次任务切换。通过“这个任务做一会，再切换到别的任务，再做一会儿”的方式，让任务看起来是并行执行的
2. 硬件并发：基于单芯多核处理器(多核处理器)的台式机，也越来越大众化。无论拥有几个处理器，这些机器都能够真正的并行多个任务。我们称其为硬件并发(hardware concurrency)”。

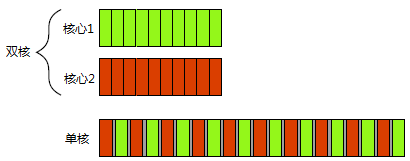


图 1.1 并发的两种方式：双核机器的真正并行 Vs. 单核机器的任务切换

不过硬件数量总会比需要质心搞的任务数量少，所以多核扔需要进行任务切换

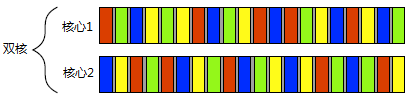
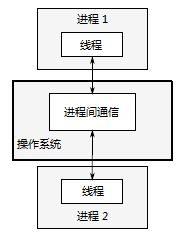
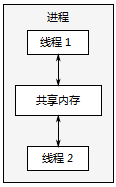


图 1.2 四个任务在两个核心之间的切换

1. 多进程并发考操作系统的进程间通信，通信代价比较大，且依赖于平台API，所以c++代码没有原生的支持，需要程序员自己编写。进程之间通常共享内存，但是这种共享通常是难以建立和管理的。因为，同一数据的内存地址在不同的进程中是不相同



而多线程并发利用共享内存进行通信。不过，共享内存的灵活性是有代价的：如果数据要被多个线程访问，那么程序员必须确保每个线程所访问到的数据是一致的(在本书第3、4、5和8章中会涉及，线程间数据共享可能会遇到的问题，以及如何使用工具来避免这些问题)。问题并非无解，只要在编写代码时适当地注意即可，这同样也意味着需要对线程通信做大量的工作。



Chapter1.2

1. 为何使用并发
2. 分离关注点（SOC）：将不相关的任务代码分离，便于代码理解和调试，比如播放DVD和接受用户的输入两个任务分为两个线程，只是线程的调用和任务之间需要人为地关联
3. 为了性能：
4. 不适用并发：收益大于成本（线程资源有限，线程切换需要操作系统的很多资源）。

Chapter1.4

1.

#include <iostream>

#include <thread> //①

void hello() //②

{

std::cout << "Hello Concurrent World\n";

}

int main()

{

std::thread t(hello); //③

t.join(); //④程序会卡在这里等待每个thread执行完成

}

管理线程的函数和类在<thread>中声明，而保护共享数据的函数和类在其他头文件中声明。

其次，打印信息的代码被移动到了一个独立的函数中②。因为每个线程都必须具有一个初始函数(initial function)，新线程的执行从这里开始。对于应用程序来说，初始线程是main()，但是对于其他线程，可以在std::thread对象的构造函数中指定——本例中，被命名为t③的std::thread对象拥有新函数hello()作为其初始函数。

下一个区别：与直接写入标准输出或是从main()调用hello()不同，该程序启动了一个全新的线程来实现，将线程数量一分为二——初始线程始于main()，而新线程始于hello()。

Chapter2.1

1. join()是简单粗暴的等待线程完成或不等待。当你需要对等待中的线程有更灵活的控制时，比如，看一下某个线程是否结束，或者只等待一段时间(超过时间就判定为超时)。想要做到这些，你需要使用其他机制来完成，比如条件变量和\_期待\_(futures)，相关的讨论将会在第4章继续。调用join()的行为，还清理了线程相关的存储部分，这样std::thread对象将不再与已经完成的线程有任何关联。这意味着，只能对一个线程使用一次join();一旦已经使用过join()，std::thread对象就不能再次加入了，当对其使用joinable()时，将返回false。
2. 因此，在调用 join() 或者 detach() 之前，我们每次都应该检查线程是否是 join-able 的。

if(threadObj.joinable())

在声明一个std::thread对象之后，都可以使用detach和join函数来启动被调线程，区别在于两者是否阻塞主调线程。  
（1）当使用join()函数时，主调线程阻塞，等待被调线程终止，然后主调线程回收被调线程资源，并继续运行；  
（2）当使用detach()函数时，主调线程继续运行，被调线程驻留后台运行，主调线程无法再取得该被调线程的控制权。当主调线程结束时，由运行时库负责清理与被调线程相关的资源。  
 当thread::join()函数被调用后，调用它的线程会被block，直到线程的执行被完成。基本上，这是一种可以用来知道一个线程已结束的机制。当thread::join()返回时，OS的执行的线程已经完成，C++线程对象可以被销毁。  
当thread::detach()函数被调用后，执行的线程从线程对象中被分离，已不再被一个线程对象所表达--这是两个独立的事情。C++线程对象可以被销毁，同时OS执行的线程可以继续。如果程序想要知道执行的线程何时结束，就需要一些其它的机制。join()函数在那个thread对象上不能再被调用，因为它已经不再和一个执行的线程相关联。  
去销毁一个仍然可以“joinable”的C++线程对象会被认为是一种错误。为了销毁一个C++线程对象，要么join()函数需要被调用（并结束），要么detach()函数被调用。如果一个C++线程对象当销毁时仍然可以被join，异常会被抛出。

1. join函数的位置需要精心挑选。因为如果开辟了一个线程，还没到join的时候，主线程（比如main函数）出了错误，导致main函数结束，而新的thread并没有执行完成，就会导致一些问题，比如访问被销毁的局部变量等。

避免应用被抛出的异常所终止，就需要作出一个决定。通常，当倾向于在无异常的情况下使用join()时，需要在异常处理过程中调用join()，从而避免生命周期的问题。下面的程序清单是一个例子。

清单 2.2 等待线程完成

struct func; // 定义在清单2.1中

void f()

{

int some\_local\_state=0;

func my\_func(some\_local\_state);

std::thread t(my\_func);

try

{

do\_something\_in\_current\_thread();

}

catch(...)

{

t.join(); // 1

throw;

}

t.join(); // 2

}

开辟了新线程t，然后主线程在执行try函数里面的时候如果捕获异常，就会进行catch，调用join（），保持线程t能执行完成。如果主程序没有问题，就会调用2的join（），等待线程t的完成。

Chapter2.2

1. thread构造函数无视函数期待的参数类型，并盲目的拷贝已提供的变量
2. c++11 std::ref

<https://murphypei.github.io/blog/2019/04/cpp-std-ref>

 C++ 中本来就有引用的存在，为何 C++11 中还要引入一个 std::ref 了？主要是考虑函数式编程（如 std::bind）在使用时，是对参数直接拷贝，而不是引用，即便函数的参数列表示引用。std::cref表示const引用传递

void f(int& n1, int& n2, const int& n3);

int n1 = 1, n2 = 2, n3 = 3;  
std::function<void()> bound\_f = std::bind(f, n1, std::ref(n2), std::cref(n3));

bound\_f();

f的参数是引用，但是在使用bind的时候，传入的n1是原始的n1的拷贝，所以运行bound\_f()的时候，f操作的不是原始的n1函数，而利用ref之后，则f操作的是原始的n2

1. bind使用

bind能够在绑定时候就同时绑定一部分参数，未提供的参数则使用占位符表示，然后在运行时传入实际的参数值。绑定的参数将会以值传递的方式传递给具体函数，占位符将会以引用传递。

#include <iostream>

using namespace std;

class A

{

public:

void fun\_3(int k,int m)

{

cout<<k<<" "<<m<<endl;

}

};

void fun(int x,int y,int z)

{

cout<<x<<" "<<y<<" "<<z<<endl;

}

void fun\_2(int &a,int &b)

{

a++;

b++;

cout<<a<<" "<<b<<endl;

}

int main(int argc, const char \* argv[])

{

auto f1 = std::bind(fun,1,2,3); *//表示绑定函数 fun 的第一，二，三个参数值为： 1 2 3*

f1(); *//print:1 2 3*

auto f2 = std::bind(fun, placeholders::\_1,placeholders::\_2,3);

*//表示绑定函数 fun 的第三个参数为 3，而fun 的第一，二个参数分别有调用 f2 的第一，二个参数指定*

f2(1,2);*//print:1 2 3*

auto f3 = std::bind(fun,placeholders::\_2,placeholders::\_1,3);

*//表示绑定函数 fun 的第三个参数为 3，而fun 的第一，二个参数分别有调用 f3 的第二，一个参数指定*

*//注意： f2 和 f3 的区别。*

f3(1,2);*//print:2 1 3*

int n = 2;

int m = 3;

auto f4 = std::bind(fun\_2, n,placeholders::\_1);

f4(m); *//print:3 4*

cout<<m<<endl;*//print:4 说明：bind对于不事先绑定的参数，通过std::placeholders传递的参数是通过引用传递的*

cout<<n<<endl;*//print:2 说明：bind对于预先绑定的函数参数是通过值传递的*

A a;

auto f5 = std::bind(&A::fun\_3, a,placeholders::\_1,placeholders::\_2);

f5(10,20);*//print:10 20*

std::function<void(int,int)> fc = std::bind(&A::fun\_3, a,std::placeholders::\_1,std::placeholders::\_2);

fc(10,20);*//print:10 20*

return 0;

}

1. unique\_ptr是独占型的智能指针，它不允许其他的智能指针共享其内部的指针，不允许通过赋值将一个unique\_ptr赋值给另一个unique\_ptr，如下面错误用法：

std::unique\_ptr<T> myPtr(new T);

std::unique\_ptr<T> myOtherPtr = myPtr; // error

　　但是unique\_ptr允许通过函数返回给其他的unique\_ptr，还可以通过std::move来转移到其他的unique\_ptr，注意，这时它本身就不再拥有原来指针的所有权了。相比于auto\_ptr而言，unique\_ptr是显示的转移，而不是莫名其妙的报废，因为auto\_ptr调用拷贝构造函数后，原来的对象就失效了。

std::unique\_ptr<T> myPtr(new T);

std::unique\_ptr<T> myOtherPtr = std::move(myPtr); // ok

对于一般的程序使用std::unique\_ptr就够了，如果是多线程方面，可能存在共同使用的问题，可以使用std::shared\_ptr，注意不要引起循环引用。

Chapter2.3转移线程所有权

1. ++标准库中有很多\_资源占有\_(resource-owning)类型，比如std::ifstream,std::unique\_ptr还有std::thread都是可移动（std::move(t)），但不可拷贝。这就说明执行线程的所有权可以在std::thread实例中移动.

void some\_function();

std::thread t1(some\_function); // 1

std::thread t2=std::move(t1);

t1线程的所有权转移给了t2，这时t1与some\_function没什么关系，t2与其有关系。

Std::thread th不能直接作为函数参数，而应该用std::move(th)作为函数参数

void f(std::thread t);

void g()

{

void some\_function();

f(std::thread(some\_function));

std::thread t(some\_function);

f(std::move(t));

}

Chapte2.4 运行时决定线程的数量

1. std::thread::hardware\_concurrency()在新版C++标准库中是一个很有用的函数。这个函数将返回能同时并发在一个程序中的线程数量。例如，多核系统中，返回值可以是CPU核芯的数量。返回值也仅仅是一个提示，当系统信息无法获取时，函数也会返回0。但是，这也无法掩盖这个函数对启动线程数量的帮助。

unsigned long const min\_per\_thread=25;

unsigned long const max\_threads=

(length+min\_per\_thread-1)/min\_per\_thread; // 2

这里25表示每大约25个任务启动一个线程，不能每个任务都启动一个线程，这样比较浪费资源。

当std::thread::hardware\_concurrency()返回0，你可以选择一个合适的数作为你的选择；在本例中,我选择了"2"

将vector里的thread加入join

std::for\_each(threads.begin(),threads.end(),

std::mem\_fn(&std::thread::join));