说到多线程编程，那么就不得不提**并行**和**并发**，多线程是实现并发（并行）的一种手段。**并行**是指两个或多个独立的操作**同时进行**。注意这里是*同时进行*，区别于**并发**，在一个时间段内执行多个操作。在单核时代，多个线程是**并发**的，在一个时间段内轮流执行；在多核时代，多个线程可以实现真正的**并行**，在多核上真正独立的并行执行。例如现在常见的**4核4线程**可以并行4个线程；**4核8线程**则使用了超线程技术，把一个物理核模拟为2个逻辑核心，可以并行8个线程。

**并发编程的方法**

通常，要实现并发有两种方法：多进程和多线程。

**多进程并发**

使用多进程并发是将一个应用程序划分为多个独立的进程（每个进程只有一个线程），这些独立的进程间可以互相通信，共同完成任务。由于操作系统对进程提供了大量的保护机制，以避免一个进程修改了另一个进程的数据，使用多进程比多线程更容易写出**安全**的代码。但这也造就了多进程并发的两个缺点：

* 在进程件的通信，无论是使用信号、套接字，还是文件、管道等方式，其使用要么比较复杂，要么就是速度较慢或者两者兼而有之。
* 运行多个线程的开销很大，操作系统要分配很多的资源来对这些进程进行管理。

由于多个进程并发完成同一个任务时，不可避免的是：操作同一个数据和进程间的相互通信，上述的两个缺点也就决定了多进程的并发不是一个好的选择。

**多线程并发**

多线程并发指的是在同一个进程中执行多个线程。有操作系统相关知识的应该知道，线程是轻量级的进程，每个线程可以独立的运行不同的指令序列，但是线程不独立的拥有资源，依赖于创建它的进程而存在。也就是说，**同一进程中的多个线程共享相同的地址空间，可以访问进程中的大部分数据，指针和引用可以在线程间进行传递**。这样，同一进程内的多个线程能够很方便的进行数据共享以及通信，也就比进程更适用于并发操作。由于缺少操作系统提供的保护机制，在多线程共享数据及通信时，就需要程序员做更多的工作以保证对共享数据段的操作是以预想的操作顺序进行的，并且要极力的避免**死锁(deadlock)**。

**C++ 11的多线程初体验**

C++11的标准库中提供了多线程库，使用时需要#include <thread>头文件，该头文件主要包含了对线程的管理类std::thread以及其他管理线程相关的类。下面是使用C++多线程库的一个简单示例：

#include <iostream>

#include <thread>

using namespace std;

void output(int i)

{

cout << i << endl;

}

int main()

{

for (uint8\_t i = 0; i < 4; i++)

{

thread t(output, i);

t.detach();

}

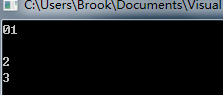
getchar();

return 0;

}

在一个for循环内，创建4个线程分别输出数字0、1、2、3，并且在每个数字的末尾输出换行符。语句thread t(output, i)创建一个线程t，该线程运行output，第二个参数i是传递给output的参数。t在创建完成后自动启动，t.detach表示该线程在后台允许，无需等待该线程完成，继续执行后面的语句。这段代码的功能是很简单的，如果是顺序执行的话，其结果很容易预测得到

0 \n 1 \n 2 \n 3 \n

但是在并行多线程下，其执行的结果就多种多样了，下图是代码一次运行的结果：  


可以看出，首先输出了01，并没有输出换行符；紧接着却连续输出了2个换行符。不是说好的并行么，同时执行，怎么还有先后的顺序？这就涉及到多线程编程最核心的问题了**资源竞争**。CPU有4核，可以同时执行4个线程这是没有问题了，但是**控制台却只有一个，同时只能有一个线程拥有这个唯一的控制台**，将数字输出。将上面代码创建的四个线程进行编号：t0,t1,t2,t3，分别输出的数字：0,1,2,3。参照上图的执行结果，控制台的拥有权的转移如下：

* t0拥有控制台，输出了数字0，但是其没有来的及输出换行符，控制的拥有权却转移到了t1；（0）
* t1完成自己的输出，t1线程完成 （1\n）
* 控制台拥有权转移给t0，输出换行符 （\n）
* t2拥有控制台，完成输出 （2\n）
* t3拥有控制台，完成输出 （3\n)

由于控制台是系统资源，这里控制台拥有权的管理是操作系统完成的。但是，假如是多个线程共享进程空间的数据，这就需要自己写代码控制，每个线程何时能够拥有共享数据进行操作。**共享数据的管理**以及**线程间的通信**，是多线程编程的两大核心。

**线程管理**

每个应用程序至少有一个进程，而每个进程至少有一个主线程，除了主线程外，在一个进程中还可以创建多个线程。每个线程都需要一个入口函数，入口函数返回退出，该线程也会退出，主线程就是以main函数作为入口函数的线程。在C++ 11的线程库中，将线程的管理在了类std::thread中，使用std::thread可以创建、启动一个线程，并可以将线程挂起、结束等操作。

**启动一个线程**

C++ 11的线程库启动一个线程是非常简单的，只需要创建一个std::thread对象，就会启动一个线程，并使用该std::thread对象来管理该线程。

do\_task();

std::thread(do\_task);

这里创建std::thread传入的函数，实际上其构造函数需要的是可调用（callable）类型，只要是有函数调用类型的实例都是可以的。所有除了传递函数外，还可以使用：

* lambda表达式

使用lambda表达式启动线程输出数字

for (int i = 0; i < 4; i++)

{

thread t([i]{

cout << i << endl;

});

t.detach();

}

* 重载了()运算符的类的实例

使用重载了()运算符的类实现多线程数字输出

class Task

{

public:

void operator()(int i)

{

cout << i << endl;

}

};

int main()

{

for (uint8\_t i = 0; i < 4; i++)

{

Task task;

thread t(task, i);

t.detach();

}

}

把函数对象传入std::thread的构造函数时，要注意一个C++的语法解析错误（C++'s most vexing parse）。向std::thread的构造函数中传入的是一个临时变量，而不是命名变量就会出现语法解析错误。如下代码：

std::thread t(Task());

这里相当于声明了一个函数t，其返回类型为thread，而不是启动了一个新的线程。可以使用新的初始化语法避免这种情况

std::thread t{Task()};

当线程启动后，**一定要在和线程相关联的thread销毁前，确定以何种方式等待线程执行结束**。C++11有两种方式来等待线程结束

* detach方式，启动的线程自主在后台运行，当前的代码继续往下执行，不等待新线程结束。前面代码所使用的就是这种方式。
* join方式，等待启动的线程完成，才会继续往下执行。假如前面的代码使用这种方式，其输出就会0,1,2,3，因为每次都是前一个线程输出完成了才会进行下一个循环，启动下一个新线程。

无论在何种情形，一定要在thread销毁前，调用t.join或者t.detach，来决定线程以何种方式运行。当使用join方式时，会阻塞当前代码，等待线程完成退出后，才会继续向下执行；而使用detach方式则不会对当前代码造成影响，当前代码继续向下执行，创建的新线程同时并发执行，这时候需要特别注意：**创建的新线程对当前作用域的变量的使用**，创建新线程的作用域结束后，有可能线程仍然在执行，这时局部变量随着作用域的完成都已销毁，如果线程继续使用局部变量的**引用或者指针**，会出现意想不到的错误，并且这种错误很难排查。例如：

auto fn = [](int \*a){

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

cout << \*a << endl;

};

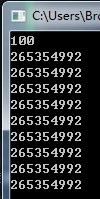
[]{

int a = 100;

thread t(fn, &a);

t.detach();

}();

在lambda表达式中，使用fn启动了一个新的线程，在装个新的线程中使用了局部变量a的指针，并且将该线程的运行方式设置为detach。这样，在lamb表达式执行结束后，变量a被销毁，但是在后台运行的线程仍然在使用已销毁变量a的指针，其输出结果如下：  


只有第一个输出是正确的值，后面输出的值是a已被销毁后输出的结果。所以在以detach的方式执行线程时，要将线程访问的局部数据复制到线程的空间（使用值传递），一定要确保线程没有使用局部变量的引用或者指针，除非你能肯定该线程会在局部作用域结束前执行结束。当然，使用join方式的话就不会出现这种问题，它会在作用域结束前完成退出。

**异常情况下等待线程完成**

当决定以detach方式让线程在后台运行时，可以在创建thread的实例后立即调用detach，这样线程就会后thread的实例分离，即使出现了异常thread的实例被销毁，仍然能保证线程在后台运行。但线程以join方式运行时，需要在主线程的合适位置调用join方法，如果调用join前出现了异常，thread被销毁，线程就会被异常所终结。为了避免异常将线程终结，或者由于某些原因，例如线程访问了局部变量，就要保证线程一定要在函数退出前完成，就要保证要在函数退出前调用join

void func() {

thread t([]{

cout << "hello C++ 11" << endl;

});

try

{

do\_something\_else();

}

catch (...)

{

t.join();

throw;

}

t.join();

}

上面代码能够保证在正常或者异常的情况下，都会调用join方法，这样线程一定会在函数func退出前完成。但是使用这种方法，不但代码冗长，而且会出现一些作用域的问题，并不是一个很好的解决方法。

一种比较好的方法是资源获取即初始化（RAII,Resource Acquisition Is Initialization)，该方法提供一个类，在析构函数中调用join。

class thread\_guard

{

thread &t;

public :

explicit thread\_guard(thread& \_t) :

t(\_t){}

~thread\_guard()

{

if (t.joinable())

t.join();

}

thread\_guard(const thread\_guard&) = delete;

thread\_guard& operator=(const thread\_guard&) = delete;

};

void func(){

thread t([]{

cout << "Hello thread" <<endl ;

});

thread\_guard g(t);

}

无论是何种情况，当函数退出时，局部变量g调用其析构函数销毁，从而能够保证join一定会被调用。

**向线程传递参数**

向线程调用的函数传递参数也是很简单的，只需要在构造thread的实例时，依次传入即可。例如：

void func(int \*a,int n){}

int buffer[10];

thread t(func,buffer,10);

t.join();

需要注意的是，**默认的会将传递的参数以拷贝的方式复制到线程空间，即使参数的类型是引用。**例如：

void func(int a,const string& str);

thread t(func,3,"hello");

func的第二个参数是string &，而传入的是一个字符串字面量。该字面量以const char\*类型传入线程空间后，在**线程的空间内转换为string**。

如果在线程中使用引用来更新对象时，就需要注意了。默认的是将对象拷贝到线程空间，其引用的是拷贝的线程空间的对象，而不是初始希望改变的对象。如下：

class \_tagNode

{

public:

int a;

int b;

};

void func(\_tagNode &node)

{

node.a = 10;

node.b = 20;

}

void f()

{

\_tagNode node;

thread t(func, node);

t.join();

cout << node.a << endl ;

cout << node.b << endl ;

}

在线程内，将对象的字段a和b设置为新的值，但是在线程调用结束后，这两个字段的值并不会改变。这样由于引用的实际上是局部变量node的一个拷贝，而不是node本身。在将对象传入线程的时候，调用std::ref，将node的引用传入线程，而不是一个拷贝。thread t(func,std::ref(node));

也可以使用类的成员函数作为线程函数，示例如下

class \_tagNode{

public:

void do\_some\_work(int a);

};

\_tagNode node;

thread t(&\_tagNode::do\_some\_work, &node,20);

上面创建的线程会调用node.do\_some\_work(20)，第三个参数为成员函数的第一个参数，以此类推。

**转移线程的所有权**

thread是可移动的(movable)的，但不可复制(copyable)。可以通过move来改变线程的所有权，灵活的决定线程在什么时候join或者detach。

thread t1(f1);

thread t3(move(t1));

将线程从t1转移给t3,这时候t1就不再拥有线程的所有权，调用t1.join或t1.detach会出现异常，要使用t3来管理线程。这也就意味着thread可以作为函数的返回类型，或者作为参数传递给函数，能够更为方便的管理线程。

线程的标识类型为std::thread::id，有两种方式获得到线程的id。

* 通过thread的实例调用get\_id()直接获取
* 在当前线程上调用this\_thread::get\_id()获取

**总结**

本文主要介绍了C++11引入的标准多线程库的一些基本操作。有以下内容：

* 线程的创建
* 线程的执行方式,join或者detach
* 向线程函数传递参数，需要注意的是线程默认是以拷贝的方式传递参数的，当期望传入一个引用时，要使用std::ref进行转换
* 线程是movable的，可以在函数内部或者外部进行传递
* 每个线程都一个标识，可以调用get\_id获取。