C++11多线程

第一部分：进程线程的概念

第二部分：C++11子线程对象

第三部分：C++11子线程对象调用的函数

第四部分：参数传递与线程id

第五部分：数据保护机制

第一部分：进程线程的概念

在早期的unix系统中，为了更加高效的使用多核cpu，更加高效的响应程序，某些情况下简化程序设计，引入了进程的概念。程序也开始以进程为单位进行设计。这一设计理念在windows诞生后得到了进一步的优化，windows将线程这一概念发扬光大。

windows的程序设计理念总体就是的一个程序就是一个进程，一个进程中至少包含一个线程，线程是进程的执行单元，进程可以理解为一个仓库，里面包含各种资源，线程也是资源的一种。所以很少在windows上提进程间通信的说法。

unix以及linux系统也借鉴了windows的这一理念，unix以及linux系统对原有的进程进行修改，使其变得更加轻量，只有依靠进程才能存在，这就是unix下的线程。

所以说unix的多线程和windows的多线程在实现方法上有本质的区别，但是用的是C++11提供的多线程功能，所有平台通用。

简单来说进程线程的概念，一个运行的程序就是一个进程。进程可以理解为一个仓库，其中包括程序运行的各种资源，包括最小执行单元：线程。当主线程执行结束后整个进程也会结束。

操作系统调用可执行程序就会创建一个进程，进程的创建过程不需要学。要学的是在一个进程中创建多个线程的方法。

第二部分：C++11子线程对象

C++11提供了一种简单的创建线程的方法。

在子线程创建完毕后，主线程和子线程均会继续执行。值得注意的一点是，这种情况下，主线程执行完毕时，子线程会立即结束。子线程执行完毕，主线程继续执行。

严格来说这种说法并不准确，一个进程中的线程之间是没有父子之分的，都是平级关系。main执行完之后, 会调用exit()，exit() 会让整个进程终止，进程包含的所有线程自然都会退出。

当子线程被创建后，子线程只可能有两种执行方法。

1. 主线程阻塞，等待子线程的执行，这也是传统的处理多线程的方法，在C++11中，直接用子线程对象调用join函数即可。join函数调用后，主线程阻塞，子线程执行完毕后主线程继续执行。
2. 子线程和主线程分离执行，如果不采取任何措施，主线程执行完毕后会调用exit函数，进程回收全部资源，如果此时子线程执行到一半就会出现严重的错误。所以必须调用detach函数，让C++运行库来接管子线程，这样当主线程执行完毕，进程退出时，子线程由C++运行库来释放资源，就不会出现内存泄露这种严重的错误。额外说一点，detach功能有些复杂，导致它实现和使用都有非常多的坑，实现部分不需要在意，但使用部分有非常多的坑需要去踩。

基于以上两点，在创建子线程对象后就一定要调用join或detach函数。在比较严谨的项目中，调用join或detach之前要检查该线程对象是否可以被执行（也就是join或detach），这个检查方法就是joinable函数，这是一个布尔类型的函数，通常不能被joinable有以下几种情况：

1. 该线程对象是std::thread的缺省构造函数生成的（thread()没有参数）。
2. 该线程对象被move过，这一点大部分时候无需在意（包括move构造和move赋值）。
3. 该线程被join或者detach过。

第三部分：C++子线程对象调用的函数

1. 普通函数：返回值可以是任意类型，但通常写作void。可以传递参数，传递参数下一部分就会讲。
2. 类对象：就是在C++的类中重载（），让C++对象作为函数来使用
3. 类函数：就是使用C++类中的任意一个函数作为C++子线程函数
4. lambda表达式：这里给不熟悉lambda的介绍下lambda表达式。lambda表达式就是一个匿名函数，lambda被发明出来就是因为每使用一个函数就要单独写出来太麻烦了，所以发明了lambda表达式。
   1. 基本格式：[capture](params)->ret{body}，接下来对每个部分进行说明
   2. [capture]，代表捕获截取的外部变量，普通函数中，要使用局部变量需要传递参数，不是所有的外部变量都是可见的，这里也是如此。
      1. [] 不截取任何变量
      2. [&] 截取外部作用域中所有变量，并作为引用在函数体中使用
      3. [=] 截取外部作用域中所有变量，并拷贝一份在函数体中使用
      4. [=, &foo] 截取外部作用域中所有变量，并拷贝一份在函数体中使用，但是对foo变量使用引用
      5. [bar] 截取bar变量并且拷贝一份在函数体重使用，同时不截取其他变量
      6. [this] 截取当前类中的this指针。如果已经使用了&或者=就默认添加此选项。
   3. (params)，代表的函数的参数：举个例子
   4. ->ret 代表返回类型，可以不写，因为会自动推导
   5. ｛body｝函数体，函数的真正执行部分

第四部分：参数传递与线程id

对于普通变量，传值即可，对于类类型，还是传递引用；在创建线程对象时，避免隐式类型转化

获取线程id是一个非常简单的部分，只有一个函数this\_thread::get\_id();

第五部分：数据保护机制

1. 互斥体：当两个线程对同一数据进行操作时，有三种可能
   1. 都在读取数据
   2. 都在写数据
   3. 一个在读数据，一个在写数据

第一种情况不涉及数据安全问题，第二种，第三种都涉及数据安全问题了。做个假设，有线程A，线程B。线程A，线程B均对一个数据进行修改，修改一个数据有很多步骤，线程A对一个数据的操作进行到一半，数据处于一种状态，此时线程B也对这个数据进行修改，当线程A对这个数据进行下一步操作时，数据的状态已经被线程B修改了。所以会导致严重的错误。

解决方法也很简单，就是互斥锁，对一段数据进行保护，当数据被一个线程锁住时，另一个线程无法对该数据进行操作。这就保证了数据的安全。

1. 原子操作：在代码简单时，优势很大，但代码复杂时，反而会比上锁更加复杂。
2. 死锁：死锁就好比两个人A，B在互相等待对方到了再离开，A：等B来了我就去B所在的地方，B：等A来了我就去A所在的地方。

放到实际开发环境中就是有两个互斥锁A，B。线程1先lock A，再lock B。线程B先lock B，再lock A。在某一时刻，线程1打算lock A的时候，线程2打算lock B。于是两个线程均无法继续执行。这就是死锁。一个很容易出现，很典型的错误。

要达成死锁，至少要满足以下两个条件。

条件1：有至少两个互斥锁

条件2：2个线程在调用两个互斥体是顺序相反

死锁的解决方法：

方法1：两个互斥量上锁的顺序相同

方法2：std::lock函数模板，一次能锁住至少两个互斥体，可以保证不存在死锁的问题。实现原理也很简单，std::lock模板可以测试两个锁能不能全部锁住，如果不能，就可以自动放开第一个锁，从而避免死锁。

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

std::lock\_guard<std::mutex> lg1(\_mutex1, std::adopt\_lock);

std::lock\_guard<std::mutex> lg2(\_mutex2, std::adopt\_lock);