实例解析C _CLI线程之线程状态持久性 PDF转换可能丢失图片或格式，建议阅读原文 https://www.100test.com/kao_ti2020/264/2021_2022__E5_AE_9E_E4_BE_8B_E8_A7_A3_E6_c97_264479.htm 其他形式的同步 我们可使用类Monitor与类Thread中的某些函数，直接控制线程的同步，请看例1。 例1：using namespace System.using namespace System::Threading.int main(){ /*1*/ MessageBuffer^ m = gcnew MessageBuffer. /*2a*/ ProcessMessages^ pm = gcnew ProcessMessages(m). /*2b*/ Thread^ pmt = gcnew Thread(gcnew ThreadStart(pm,amp.CreateMessages::CreateMessagesEntryPoint)). /*3c*/ cmt->Start(). /*4*/ cmt->Join(). /*5*/ pmt->Interrupt(). /*6*/ pmt->Join(). Console::WriteLine("Primary thread terminating").}public ref class MessageBuffer { String^ messageText. public: void SetMessage(String^ s) { /*7*/ Monitor::Enter(this). messageText = s. /*8*/ Monitor::Pulse(this). Console::WriteLine("Set new message {0}", messageText). Monitor::Exit(this). } void ProcessMessages() { /*9*/ Monitor::Enter(this). while (true) { try { /*10*/ Monitor::Wait(this). } catch (ThreadInterruptedException^ e) { Console::WriteLine("ProcessMessage interrupted"). return. } Console::WriteLine("Processed new message {0}", messageText). } Monitor::Exit(this). }}.public ref class CreateMessages{ MessageBuffer^ msg. public: CreateMessages(MessageBuffer^ m) { msg = m. } void CreateMessagesEntryPoint() { for (int i = 1. i { msg->SetMessage(String::Concat("M-", i.ToString())). Thread::Sleep(2000). } Console::WriteLine("CreateMessages thread

terminating"). }}.public ref class ProcessMessages{ MessageBuffer^ msg. public: ProcessMessages(MessageBuffer^ m) { msg = m. } void ProcessMessagesEntryPoint() { msg->ProcessMessages(). Console::WriteLine("ProcessMessages thread terminating"). }}. 在标记1中，创建一个MessageBuffer类型的共享缓冲区；接着在标记2a、2b、2c中，创建了一个线程用于处理放置于缓冲区中的每条信息；标记3a、3b和3c，也创建了一个线程，并在共享缓冲区中放置了连续的5条信息以便处理。这两个线程已被同步，因此处理者线程必须等到有"东西"放入到缓冲区中，才可以进行处理，且在前一条信息被处理完之前，不能放入第二条信息。在标记4中，将一直等待，直到创建者线程完成它的工作。 当标记5执行时，处理者线程必须处理所有创建者线程放入的信息，因为使用了Thread::Interrupt让其停止工作，并继续等待标记6中调用的Thread::Join，这个函数允许调用线程阻塞它自己，直到其他线程结束。（一个线程可指定一个等待的最大时间，而不用无限等待下去。） 线程CreateMessages非常清晰明了，它向共享缓冲区中写入了5条信息，并在每条信息之间等待2秒。为把一个线程挂起一个给定的时间（以毫秒计），我们调用了Thread::Sleep，在此，一个睡眠的线程可再继续执行，原因在于运行时环境，而不是另一个线程。 线程ProcessMessages甚至更加简单，因为它利用了类MessageBuffer来做它的所有工作。类MessageBuffer中的函数是被同步的，因此在同一时间，只有一个函数能访问共享缓冲区。 主程序首先启动处理者线程，这个线程会执行ProcessMessages，其将获得父对象的同步锁；然而，它立即调用了标记10中的Wait函数，这个函数将让它一直等待，直

到再次被告之运行，期间，它也交出了同步锁，这样，允许创建者线程得到同步锁并执行SetMessage。一旦函数把新的信息放入到共享缓冲区中，就会调用标记8中的Pulse，其允许等待同步锁的任意线程被唤醒，并继续执行下去。但是，在SetMessage执行完成之前，这些都不可能发生，因为它在函数返回前都不可能交出同步锁。如果情况一旦发生，处理者线程将重新得到同步锁，并从标记10之后开始继续执行。此处要说明的是，一个线程即可无限等待，也可等到一个指定的时间到达。插1是程序的输出。 插1：Set new message M-1Processed new message M-1Set new message M-2Processed new message M-2Set new message M-3Processed new message M-3Set new message M-4Processed new message M-4Set new message M-5Processed new message M-5CreateMessages thread terminatingProcessMessage interruptedProcessMessages thread terminatingPrimary thread terminating 请仔细留意，处理者线程启动于创建者线程之前。如果以相反的顺序启动，将会在没有处理者线程等待的情况下，添加第一条信息，此时，没有可供唤醒处理者线程，当处理者线程运行到它的第一个函数调用Wait时，将会错过第一条信息，且只会在第二条信息存储时被唤醒。 