C++多线程编程入门
2009 年 05 月 11 日 星期一 18:13
第 1 节　背景
为了更好的理解多线程的概念，先对进程，线程的概念背景做一下简单介绍。

早期的计算机系统都只允许一个程序独占系统资源，一次只能执行一个程序。在大型机年代，计算
是一种宝贵资源。对于资

源拥有方来说，最好的生财之道自然是将同一资源同时租售给尽可能多的用户。最理想的情况是全
球计算市场。所以不难理

解为何当年 IBM 预测“全球只要有 4 台计算机就够了”。

这种背景下，一个计算机能够支持多个程序并发执行的需求变得十分迫切。由此产生了进程的概念
程在多数早期多任务操作

系统中是执行工作的基本单元。进程是包含程序指令和相关资源的集合。每个进程和其他进程一起
调度，竞争 CPU，内存等

系统资源。每次进程切换，都存在进程资源的保存和恢复动作，这称为上下文切换。

进程的引入可以解决支持多用户的问题，但是多进程系统也在如下方面产生了新的问题：
进程频繁切换引起的额外开销可能会严重影响系统性能。
进程间通信要求复杂的系统级实现。

在程序功能日趋复杂的情况下，上述缺陷也就凸现出来。比如，一个简单的 GUI 程序，为了有更好
互性，通常用一个任务支

持界面交互，另一个任务支持后台运算。如果每个任务均由一个进程来实现，那会相当低效。对每
程来说，系统资源看上去

都是其独占的。比如内存空间，每个进程认为自己的内存空间是独有的。一次切换，这些独立资源
要切换。

由此就演化出了利用分配给同一个进程的资源，尽量实现多个任务的方法。这也就引入了线程的概
同一个进程内部的多个线

程，共享的是同一个进程的所有资源。

比如，与每个进程独有自己的内存空间不同，同属一个进程的多个线程共享该进程的内存空间。例
进程地址空间中有一个全

局变量 globalVar，若 A 线程将其赋值为 1，则另一线程 B 可以看到该变量值为 1。两个线程看到
局变量 globalVar 是同一个变量

。

通过线程可以支持同一个应用程序内部的并发，免去了进程频繁切换的开销，另外并发任务间通信
简单。

目前多线程应用主要用于两大领域：网络应用和嵌入式应用。为什么在这两个领域应用较多呢？因
线程应用能够解决两大问

题：
并发。网络程序具有天生的并发性。比如网络数据库可能需要同时处理数以千计的请求。而由于网
接的时延不确定性和不可

靠性，一旦等待一次网络交互，可以让当前线程进入睡眠，退出调度，处理其他线程。这样就能够
利用系统资源，充分发挥

系统处理能力。
实时。线程的切换是轻量级的，所以可以保证足够快。每当有事件发生，状态改变，都能有线程及
应，而且每次线程内部处

理的计算强度和复杂度都不大。在这种情况下，多线程实现的模型也是高效的。

在有些语言中，对多线程或者并发的支持是直接内建在语言中的，比如 Ada 和 VHDL。在 C++里面，
线程的支持由具体操作系统

提供的函数接口支持。不同的系统中具体实现方法不同。后面所有例子只给出 windows 和 Unix/L
的实现。

在后面的实现中，考虑的是尽量封装隔离底层的多线程函数接口，屏蔽操作系统底层的线程实现具
节，介绍的重点是多线程

编程中较通用的概念。同时也尽量体现 C++面向对象的一面。

最后，由于空闲时间有限，我只求示例代码能够明确表达自己的意思即可。至于代码的尽善尽美就
有劳各位尽力以为之了。

第 2 节　线程的创建
本节介绍如下内容
线程状态
线程运行环境
线程类定义
示例程序
线程类的 Windows 和 Unix 实现
线程状态
在一个线程的生存期内，可以在多种状态之间转换。不同操作系统可以实现不同的线程模型，定义

不同的线程状态，每个状

态还可以包含多个子状态。但大体说来，如下几种状态是通用的：
就绪：参与调度，等待被执行。一旦被调度选中，立即开始执行。
运行：占用 CPU，正在运行中。
休眠：暂不参与调度，等待特定事件发生。
中止：已经运行完毕，等待回收线程资源（要注意，这个很容易误解，后面解释）。
线程环境
线程存在于进程之中。进程内所有全局资源对于内部每个线程均是可见的。
进程内典型全局资源有如下几种：
代码区。这意味着当前进程空间内所有可见的函数代码，对于每个线程来说也是可见的。
静态存储区。全局变量。静态变量。
动态存储区。也就是堆空间。
线程内典型的局部资源有：
本地栈空间。存放本线程的函数调用栈，函数内部的局部变量等。
部分寄存器变量。例如本线程下一步要执行代码的指针偏移量。

一个进程发起之后，会首先生成一个缺省的线程，通常称这个线程为主线程。C/C++程序中主线程
通过 main 函数进入的线程

。由主线程衍生的线程称为从线程，从线程也可以有自己的入口函数，作用相当于主线程的 main

这个函数由用户指定。Pthread 和 winapi 中都是通过传入函数指针实现。在指定线程入口函数时
可以指定入口函数的参数。

就像 main 函数有固定的格式要求一样，线程的入口函数一般也有固定的格式要求，参数通常都是
*类型，返回类型在

pthread 中是 void *, winapi 中是 unsigned int，而且都需要是全局函数。

最常见的线程模型中，除主线程较为特殊之外，其他线程一旦被创建，相互之间就是对等关系（pe
peer)， 不存在隐含的

层次关系。每个进程可以创建的最大线程数由具体实现决定。

为了更好的理解上述概念，下面通过具体代码来详细说明。
线程类接口定义
一个线程类无论具体执行什么任务，其基本的共性无非就是
创建并启动线程
停止线程
另外还有就是能睡，能等，能分离执行（有点拗口，后面再解释）。
还有其他的可以继续加…
将线程的概念加以抽象，可以为其定义如下的类：
文件 thread.h

```
#ifndef __THREAD__H_
```

```
#define __THREAD__H_
class Thread
{
public:
Thread();
virtual ~Thread();
int start (void * = NULL);
void stop();
void sleep (int);
void detach();
void * wait();
protected:
virtual void * run(void *) = 0;
private:
//这部分 win 和 unix 略有不同，先不定义，后面再分别实现。
//顺便提一下，我很不习惯写中文注释，这里为了更明白一
//点还是选用中文。
…
};
#endif
```

Thread::start()函数是线程启动函数，其输入参数是无类型指针。
Thread::stop()函数中止当前线程。
Thread::sleep()函数让当前线程休眠给定时间，单位为秒。
Thread::run()函数是用于实现线程类的线程函数调用。
Thread::detach()和 thread::wait()函数涉及的概念略复杂一些。在稍后再做解释。

Thread 类是一个虚基类，派生类可以重载自己的线程函数。下面是一个例子。

示例程序

代码写的都不够精致，暴力类型转换比较多，欢迎有闲阶级美化，谢过了先。
文件 create.h

```
#ifndef __CREATOR__H_
#define __CREATOR__H_

#include <stdio.h>
#include "thread.h"

class Create: public Thread
{
protected:
void * run(void * param)
{
        char * msg = (char*) param;
```

```
        printf ("%s\n", msg);
        //sleep(100); 可以试着取消这行注释，看看结果有什么不同。
    printf("One day past.\n");
        return NULL;
}
};
#endif
```

然后，实现一个 main 函数，来看看具体效果：

文件 Genesis.cpp

```
#include <stdio.h>
#include "create.h"

int main(int argc, char** argv)
{
Create monday;
Create tuesday;

printf("At the first God made the heaven and the earth.\n");
monday.start("Naming the light, Day, and the dark, Night, the first day.");
tuesday.start("Gave the arch the name of Heaven, the second day.");
printf("These are the generations of the heaven and the earth.\n");

return 0;
}
```

编译运行，程序输出如下：

```
At the first God made the heaven and the earth.
These are the generations of the heaven and the earth.
```

令人惊奇的是，由周一和周二对象创建的子线程似乎并没有执行！这是为什么呢？别急，在最后 printf 语句之前加上如下语句

```
:
monday.wait();
tuesday.wait();
```

重新编译运行，新的输出如下：

```
At the first God made the heaven and the earth.
Naming the light, Day, and the dark, Night, the first day.
One day past.
Gave the arch the name of Heaven, the second day.
One day past.
These are the generations of the heaven and the earth.
```

为了说明这个问题，需要了解前面没有解释的 Thread::detach()和 Thread::wait()两个函数的含

无论在 windows 中，还是 Posix 中，主线程和子线程的默认关系是：

无论子线程执行完毕与否，一旦主线程执行完毕退出，所有子线程执行都会终止。这时整个进程处
僵死（部分线程保持一种

终止执行但还未销毁的状态，而进程必须在其所有线程销毁后销毁，这时进程处于僵死状态），在
个例子的输出中，可以看

到子线程还来不及执行完毕，主线程的 main()函数就已经执行完毕，从而所有子线程终止。

需要强调的是，线程函数执行完毕退出，或以其他非常方式终止，线程进入终止态（请回顾上面讨
程状态），但千万要记住

的是，进入终止态后，为线程分配的系统资源并不一定已经释放，而且可能在系统重启之前，一直
能释放。终止态的线程，

仍旧作为一个线程实体存在与操作系统中。（这点在 win 和 unix 中是一致的。）而什么时候销毁
取决于线程属性。

通常，这种终止方式并非我们所期望的结果，而且一个潜在的问题是未执行完就终止的子线程，除
为线程实体占用系统资源

之外，其线程函数所拥有的资源（申请的动态内存，打开的文件，打开的网络端口等）也不一定能
所以，针对这个问题，

主线程和子线程之间通常定义两种关系：
　　可会合(joinable)。这种关系下，主线程需要明确执行等待操作。在子线程结束后，主线程的
操作执行完毕，子线程

和主线程会合。这时主线程继续执行等待操作之后的下一步操作。主线程必须会合可会合的子线
Thread 类中，这个操作通过

在主线程的线程函数内部调用子线程对象的 wait()函数实现。这也就是上面加上三个 wait()调用
示正确的原因。必须强调的

是，即使子线程能够在主线程之前执行完毕，进入终止态，也必需显示执行会合操作，否则，系统
不会主动销毁线程，分配

给该线程的系统资源（线程 id 或句柄，线程管理相关的系统资源）也永远不会释放。
　　相分离(detached)。顾名思义，这表示子线程无需和主线程会合，也就是相分离的。这种情况
子线程一旦进入终止态

，系统立即销毁线程，回收资源。无需在主线程内调用 wait()实现会合。Thread 类中，调用 det
使线程进入 detached 状态。

这种方式常用在线程数较多的情况，有时让主线程逐个等待子线程结束，或者让主线程安排每个子

结束的等待顺序，是很困

难或者不可能的。所以在并发子线程较多的情况下，这种方式也会经常使用。
缺省情况下，创建的线程都是可会合的。可会合的线程可以通过调用 detach()方法变成相分离的但反向则不行。

UNIX 实现

文件 thread.h

```
#ifndef __THREAD__H_
#define __THREAD__H_
class Thread
{
public:
Thread();
virtual ~Thread();
int start (void * = NULL);
void stop();
void sleep (int);
void detach();
void * wait();
protected:
virtual void * run(void *) = 0;
private:
pthread_t handle;
bool started;
bool detached;
void * threadFuncParam;
friend void * threadFunc(void *);
};

//pthread 中线程函数必须是一个全局函数，为了解决这个问题
//将其声明为静态，以防止此文件之外的代码直接调用这个函数。
//此处实现采用了称为 Virtual friend function idiom 的方法。
Static void * threadFunc(void *);
#endif
```

文件 thread.cpp

```
#include <pthread.h>
#include <sys/time.h>
#include “thread.h”

static void * threadFunc (void * threadObject)
{
Thread * thread = (Thread *) threadObject;
```

```
return thread->run(thread->threadFuncParam);
}

Thread::Thread()
{
started = detached = false;
}

Thread::~Thread()
{
stop();
}

bool Thread::start(void * param)
{
pthread_attr_t attributes;
pthread_attr_init(&attributes);
if (detached)
{
        pthread_attr_setdetachstate(&attributes, PTHREAD_CREATE_DETACHED);
}

threadFuncParam = param;

if (pthread_create(&handle, &attributes, threadFunc, this) == 0)
{
        started = true;
}

pthread_attr_destroy(&attribute);
}


void Thread::detach()
{
if (started && !detached)
{
        pthread_detach(handle);
}
detached = true;
}

void * Thread::wait()
{
void * status = NULL;
```

```
if (started && !detached)
{
        pthread_join(handle, &status);
}
return status;
}

void Thread::stop()
{
if (started && !detached)
{
        pthread_cancel(handle);
        pthread_detach(handle);
        detached = true;
}
}

void Thread::sleep(unsigned int milliSeconds)
{
timeval timeout = { milliSeconds/1000, millisecond%1000};
select(0, NULL, NULL, NULL, &timeout);
}
```

Windows 实现

文件 thread.h

```
#ifndef _THREAD_SPECIFICAL_H__
#define _THREAD_SPECIFICAL_H__

#include <windows.h>

static unsigned int __stdcall threadFunction(void *);

class Thread {
                friend unsigned int __stdcall threadFunction(void *);
public:
                Thread();
                virtual ~Thread();
                int start(void * = NULL);
                void * wait();
                void stop();
                void detach();
                static void sleep(unsigned int);
```

```
protected:
                virtual void * run(void *) = 0;

private:
                HANDLE threadHandle;
                bool started;
                bool detached;
                void * param;
                unsigned int threadID;
};

#endif
```

文件 thread.cpp

```
#include "stdafx.h"
#include <process.h>
#include "thread.h"

unsigned int __stdcall threadFunction(void * object)
{
                Thread * thread = (Thread *) object;
                return (unsigned int ) thread->run(thread->param);
}

Thread::Thread()
{
                started = false;
                detached = false;
}

Thread::~Thread()
{
                stop();
}

int Thread::start(void* pra)
{
                if (!started)
                {
                                param = pra;
                                if (threadHandle = (HANDLE)_beginthreadex(NULL, 0,
threadFunction, this, 0, &threadID))
                                {
                                                if (detached)
                                                {
```

```
                                                                CloseHandle(threadHan
                                                }
                                                started = true;
                                        }
                        }
                        return started;
}

//wait for current thread to end.
void * Thread::wait()
{
                        DWORD status = (DWORD) NULL;
                        if (started && !detached)
                        {
                                        WaitForSingleObject(threadHandle, INFINITE);
                                        GetExitCodeThread(threadHandle, &status);
                                        CloseHandle(threadHandle);
                                        detached = true;
                        }

                        return (void *)status;
}

void Thread::detach()
{
if (started && !detached)
{
       CloseHandle(threadHandle);
}
detached = true;
}

void Thread::stop()
{
                        if (started && !detached)
                        {
                                        TerminateThread(threadHandle, 0);

                                        //Closing a thread handle does not terminate
                                        //the associated thread.
                                        //To remove a thread object, you must terminate the th
                                        //then close all handles to the thread.
                                        //The thread object remains in the system until
                                        //the thread has terminated and all handles to it have
```

```
                                //closed through a call to CloseHandle
                                CloseHandle(threadHandle);
                                detached = true;
                }
}

void Thread::sleep(unsigned int delay)
{
                ::Sleep(delay);
}
```

小结

本节的主要目的是帮助入门者建立基本的线程概念，以此为基础，抽象出一个最小接口的通用线
在示例程序部分，初学者

可以体会到并行和串行程序执行的差异。有兴趣的话，大家可以在现有线程类的基础上，做进一步
展和尝试。如果觉得对线

程的概念需要进一步细化，大家可以进一步扩展和完善现有 Thread 类。

想更进一步了解的话，一个建议是，可以去看看其他语言，其他平台的线程库中，线程类抽象了哪
念。比如 Java, perl 等跨

平台语言中是如何定义的，微软从 winapi 到 dotnet 中是如何支持多线程的，其线程类是如何定
这样有助于更好的理解线程

的模型和基础概念。

另外，也鼓励大家多动手写写代码，在此基础上尝试写一些代码，也会有助于更好的理解多线程程
特点。比如，先开始的线

程不一定先结束。线程的执行可能会交替进行。把 printf 替换为 cout 可能会有新的发现，等等

每个子线程一旦被创建，就被赋予了自己的生命。管理不好的话，一只特例独行的猪是非常让人头

对于初学者而言，编写多线程程序可能会遇到很多令人手足无措的 bug。往往还没到考虑效率，避
锁等阶段就问题百出，而

且很难理解和调试。这是非常正常的，请不要气馁，后续文章会尽量解释各种常见问题的原因，引
家避免常见错误。目前能

想到入门阶段常遇到的问题是：
    内存泄漏，系统资源泄漏。

程序执行结果混乱，但是在某些点插入 sleep 语句后结果又正确了。
程序 crash，但移除或添加部分无关语句后，整个程序正常运行（假相）。
多线程程序执行结果完全不合逻辑，出于预期。

本文至此，如果自己动手改改，试一些例子，对多线程程序应该多少有一些感性认识了。刚开始只
基本概念弄懂了，后面可

以一步一步搭建出很复杂的类。不过刚开始不要贪多，否则会欲速则不达，越弄越糊涂。

最后，大家见仁见智吧，我在此起到抛砖引玉的作用就很开心了，呵呵。另外文本编辑器的原因，
如果编译不过，可能需要

把标点符号从中文换成英文。
PS：如果遇到编译错误:error C2065: '_beginthreadex' : undeclared identifier.
做以下设置：
project->setting->C/C++->Code Generation->Use run-time libray->选 Debug Multithread
程），或 Multithread