03 第三章 多线程编程与线程同步

3.1 线程的基本概念和常见问题

一个进程（process）代表计算机中实际的程序。
在现代操作系统的保护下，每个进程都拥有自己独立的进程地址空间和上下文堆栈，但就一个程序本身执行的操作而言，进程其实什么也不做（不执行任何进程代码），只提供一个大的环境容器。
进程中实际执行操作的是线程（thread）。因此在一个进程中至少得有一个线程，我们把这个称为“主线程”。

主线程退出，支线程也会退出吗?

在Windows中，如果一个进程存在多个线程，当主线程退出时，其他所有的支线程（或者说：工作线程）即使还没有执行完代码都会退出。
换言之，一旦主线程退出，整个进程就结束了。

开发时要注意如何避免主线程在工作线程的逻辑未执行前退出。解决方案：

1. 主线程一直循环等待工作线程完成业务逻辑的执行
2. 主线程等到所有工作线程退出才退出

在Linux中，如果主线程退出，支线程一般不会退出，还会继续运行，但此时这个进程就会变成僵尸进程，开发时应避免产生僵尸进程。

使用 `ps -ef` 查看系统进程
带有 `<defunct>` 的即为僵尸进程。

注意：
在某些Linux发行版本中，是和Windows一样，主线程退出，支线程都会退出。

某个线程崩溃，会导致进程退出吗?

还有另外一种问法：
进程中的某个线程崩溃，会影响其他线程吗?

线程与线程之间时相互独立的，一个线程崩溃不会影响到其他线程。
但是一般线程崩溃后，会导致操作系统让整个进程退出，这样该进程下的其他进程都会被释放

3.2 线程的基本操作

创建线程

线程的创建其本质都是调用操作系统的api来进行的，所以我们分Linux和Windows来阐述这个问题。
这里只介绍了常用的函数，更多详细内容应该去Linux man手册和Windows msdn里去找。

1、Linux下创建线程

在Linux上是使用 `pthread_create` 函数来创建线程的：

```
int pthread_create(pthread_t *restrict_thread,
    const pthread_attr_t *restrict_attr,
    void *(*start_routine)(void *),
    void *restrict_arg);
```

restrict_thread:
输出参数，如果线程创建成功，那么可以通过这个参数获取创建成功线程的id。

restrict_attr:
指定线程的属性，一般设置为 `NULL` 。

start_routine:
指定了线程函数，这个函数的调用必须是 `__cdecl` ，这是 `C Declaration` 的缩写。
`__cdecl` 是C/C++在定义全局函数是默认的调用方式。

args:
用于在创建线程的时候将某个参数传入线程函数中。
由于是 `void*` 类型，所以可以方便我们最大化地传入任意信息给线程函数。

返回值:
成功创建线程则返回0;
如果失败返回对应的错误码:

1. EAGAIN
2. EINVAL
3. EPERM

EAGAIN Insufficient resources to create another thread.

```
EAGAIN A system-imposed limit on the number of threads was
       encountered.  There are a number of limits that may
       trigger this error: the RLIMIT_NPROC soft resource limit
       (set via setrlimit(2)), which limits the number of
       processes and threads for a real user ID, was reached; the
       kernel's system-wide limit on the number of processes and
       threads, /proc/sys/kernel/threads-max, was reached (see
       proc(5)); or the maximum number of PIDs,
       /proc/sys/kernel/pid_max, was reached (see proc(5)).

EINVAL Invalid settings in attr.

EPERM  No permission to set the scheduling policy and parameters
       specified in attr.
```

参考链接:
https://man7.org/linux/man-pages/man3/pthread_create.3.html

简单的示例:

```
#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

void* threadfunc(void* args) {
    while (1) {
        sleep(1);

        printf("I am a New Thread!\n");
    }
    return NULL;
}

int main() {
    pthread_t threaddid;
    pthread_create(&threaddid, NULL, threadfunc, NULL);

    while (1) {

    }
    return 0;
}
```

输出结果：

```
gcc pthread_createDemo.cpp -o pthread_createDemo -lpthread
./pthread_createDemo

I am a New Thread!
I am a New Thread!
I am a New Thread!
I am a New Thread!
I am a New Thread!
^C
```

注意用gcc运行的时候要加 `-lpthread` 参数

2、Windows下创建线程

在Windows上是使用 `CreateThread` 函数来创建线程的

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES   lpThreadAttributes,
  SIZE_T                  dwStackSize,
  LPTHREAD_START_ROUTINE  lpStartAddress,
  __drv_aliasesMem LPVOID lpParameter,
  DWORD                   dwCreationFlags,
  LPDWORD                 lpThreadId
);
```

lpThreadAttributes:
表示线程的安全属性，一般设置为NULL

dwStackSize:
指的是线程的栈大小，单位为：byte，
一般为0，表示使用默认的大小

lpStartAddress:
为线程函数，其类型为函数指针类型 `LPTHREAD_START_ROUTINE` .

`LPTHREAD_START_ROUTINE`：

```
typedef DWORD (__stdcall *LPTHREAD_START_ROUTINE) (
    [in] LPVOID lpThreadParameter
);
```

可见，此函数的调用方式是 `__stdcall` ，

```
// 如果不指定调用方式，则调用方式是 __cdecl
// 且无法作为调用方式为 __stdcall 的函数形参被 CreateThread 函数调用
DWORD threadfunc(LPVOID ipThreadParameter);

// 指定调用方式为 __stdcall 可以被 CreateThread 函数调用
DWORD __stdcall threadfunc(LPVOID ipThreadParameter);
```

另外，在Windows系统上，WINAPI和CALLBACK这两个宏的值都是 __stdcall
故，有的写法是：

```
DWORD WINAPI threadfunc(LPVOID ipThreadParameter);
DWORD CALLBACK threadfunc(LPVOID ipThreadParameter);
```

lpParameter:
是传给线程函数的形参，和Linux下args一样。
而且实际上都是 `void*` 类型， `LPVOID` 实质上用typedef封装过的 `void*`

dwCreationFlags:
32位无符号整型 `DWORD` ，一般被设置为0，表示一创建线程就启动该线程。

对于一些特殊的场景，我们不希望创建线程后立即开始执行，则可以将这个值设置为4，对应Windows定义的宏 `CREATE_SUSPENDED` .
在之后需要的时候在使用ResumeThread这个函数来运行此线程。

lpThreadId:
是一个32为无符号整型的指针（ `LPDWORD` ），表示线程创建成功时返回的线程ID。

在Windows上是使用句柄来管理线程对象的，句柄在本质上是内核句柄表中的索引值。
如果成功创建线程，则返回该线程的句柄，否则返回 `NULL`

以下代码片段演示了如何在Windows上创建线程：

```
#include <Windows.h>
#include <iostream>

DWORD WINAPI ThreadProc(LPVOID lpParameters) {
        while (true)
        {
                // 睡眠 1 sec
                Sleep(1000);

                std::cout << "I am New Thread!" << std::endl;
        }
        return 0;
}

int main() {
        DWORD dwThreadID;
        HANDLE hThread = CreateThread(NULL, 0, ThreadProc, NULL, 0, &dwThreadID);

        if (hThread == NULL) {
                std::cout << "Fail to create Thread." << std::endl;
                return -1;
        }

        // 让主线程一直保持执行 不退出。
        while (true)
        {
        }
        return 0;
}
```

3、Windows CRT提供的线程创建函数

查阅 CRT官方文档

以下代码是使用 `_beginthreadex` 创建线程的例子：

```
#include <iostream>

#include <process.h>

unsigned WINAPI threadfunc(void* args) {
        while (true)
        {
                std::cout << "I am a New Thread!" << std::endl;
        }
        return 0;
}

int main() {
        unsigned int threadid;
        _beginthreadex(0, 0, threadfunc, 0, 0, &threadid);

        while (true)
        {
        }
        return 0;
}
```

在实际开发中，许多大佬推荐使用 `_beginthreadex` 而不是 `CreatThread` 函数，不知为何

为啥推荐使用 `_beginthreadex` 而不是 `CreatThread` 函数?

ToDo

4、C++11 提供的std::thread类

C++11新标准引入了一个 `std::thread` 以解决Windows和Linux线程API的格式兼容问题。

以下代码分别创建了两个线程，线程函数签名不一致：

```cpp
/* ----- C++11 标准 创建线程的例子 ----- */
#include <iostream>
#include <thread>

#ifdef __linux__
#include <unistd.h>
        #define sleep0(time) (sleep(time / 1000))
#elif _WIN32
#include <windows.h>
        #define sleep0(time) (Sleep(time))
#endif

void threadproc01() {
        while (true)
        {
                //this_thread::sleep_for(chrono::seconds(0.5));
                sleep0(500);
                std::cout << "I am a New Thred01!" << std::endl;
        }
}

void threadproc02(int a, int b) {
        while (true)
        {
                //this_thread::sleep_for(chrono::seconds(0.5));
                sleep0(500);
                std::cout << "I am a New Thred02!" << std::endl;
        }
}

int main() {
        // 创建线程1
        std::thread t1(threadproc01);
        // 创建线程2
        std::thread t2(threadproc02, 1, 2);

        while (true)
        {
        }
        return 0;
}
```

注意:
在Linux上运行该代码命令如下:

```
g++ thread_std11.cpp -o thread_std11 -lpthread
```

可见还是需要 `pthread` 库啊。

这个使用方便，但是容易出错，即 std::thread 对象在线程函数运行期间必须是有效的。啥意思？看个例子：

```
/* ----- C++11 标准 创建线程 会崩溃 的例子 ----- */
#include <iostream>
#include <thread>

void threadproc() {
        while (true)
        {
                std::cout << "I am a New Thread!" << std::endl;
        }
}

void func() {
        std::thread t(threadproc);
}

int main() {
        func();

        while (true)
        {
        }
        return 0;
}
```

main函数中调用了func函数，func函数里创建了一个线程，貌似没有问题的样子。但是实际运行的时候会崩溃。
崩溃的原因是，在func函数调用结束后，func中的局部变量t（即创建的线程）会被销毁，但此时线程函数仍然在运行。
所以在使用 std::thread 类时一定要保证线程函数运行的时候线程是有效的、没有被销毁的。
那如果我们要让线程对象销毁后，线程函数能够正常运行又该怎么办呢？

只需用 detach 方法让线程函数与线程分离即可，例如：

```
void func() {
  std::thread t(threadproc);
  t.detach();
}
```

然而在实际开发中不推荐这么做，因为我们可能需要用线程对象来管理和控制线程的运行和生命周期。
所以我们的代码应该尽量保证线程对象在线程运行期间有效。

Linux和Windows上线程函数调用方式的异同

在这里我们单独谈谈Windows上创建线程的函数和Linux的区别。

在Windows上使用该函数创建线程函数时要求其调用方式为 `__stdcall` ，但是定义的函数默认调用方式却是 `__cdecl` 。
也就是说定义的全局函数可以作为Linux `pthread_create` 的线程函数，却不能作为Windows上 `CreateThread` 的线程函数。

举个例子：

```
//不显式指定函数的调用方式，其调用方式为 __cdecl
void* start_rountine(void *arge) {}

//显式指定函数的调用方式为 __cdecl，等价于上面的代码
void* __cdecl start_routine(void *arge) {}
```

获取线程ID

Linux上使用 `pthread_self()` 来获取线程ID

Windows上使用 `GetCurrentThreadID()` 来获取线程ID

pstack命令

pstack是Linux上通过线程ID用来查看线程的命令。不仅可以查看线程数量还可以查看每个线程的调用堆栈。

使用pstack命令可以方便地排查和定位一个进程CPU使用率过高的问题。

Linux系统线程ID的本质

C++11 获取当前线程ID

等待线程结束

1、Linux等待线程结束

```
pthread_join
```

2、Windows等待线程结束

有两个重要函数：

1. `WaitForSingleObject`
2. `WaitForMultipleObjects`

前者用于等待一个线程结束，后者可用于等待多个线程结束。

3、C++11提供的等待线程结束的函数

`std::thread` 的 `join()` 方法

3.3 惯用法：将C++类对象实例指针作为线程函数的参数

3.4整型变量的原子操作

这里应该参考 操作系统 的部分

3.5 Linux线程同步对象