C++ 信号处理 C++ Web 编程

C++ 多线程

多线程是多任务处理的一种特殊形式，多任务处理允许让电脑同时运行两个或两个以上的程序。一般情况下，两种类型的多任务处理：**基于进程和基于线程。**

- 基于进程的多任务处理是程序的并发执行。
- 基于线程的多任务处理是同一程序的片段的并发执行。

多线程程序包含可以同时运行的两个或多个部分。这样的程序中的每个部分称为一个线程，每个线程定义了一个单独的执行路径。

本教程假设您使用的是 Linux 操作系统，我们要使用 POSIX 编写多线程 C++ 程序。POSIX Threads 或 Pthreads 提供的 API 可在多种类 Unix POSIX 系统上可用，比如 FreeBSD、NetBSD、GNU/Linux、Mac OS X 和 Solaris。

创建线程

下面的程序，我们可以用它来创建一个 POSIX 线程：

```
#include <pthread.h>
pthread_create (thread, attr, start_routine, arg)
```

在这里，**pthread_create** 创建一个新的线程，并让它可执行。下面是关于参数的说明：

参数	描述
thread	指向线程标识符指针。
attr	一个不透明的属性对象，可以被用来设置线程属性。您可以指定线程属性对象，也可以使用默认值 NULL。
start_routine	线程运行函数起始地址，一旦线程被创建就会执行。
arg	运行函数的参数。它必须通过把引用作为指针强制转换为 void 类型进行传递。如果没有传递参数，则使用 NULL。

创建线程成功时，函数返回 0，若返回值不为 0 则说明创建线程失败。

终止线程

使用下面的程序，我们可以用它来终止一个 POSIX 线程：

```
#include <pthread.h>
pthread_exit (status)
```

在这里，**pthread_exit** 用于显式地退出一个线程。通常情况下，pthread_exit() 函数是在线程完成工作后无需继续存在时被调用。

如果 main() 是在它所创建的线程之前结束，并通过 pthread_exit() 退出，那么其他线程将继续执行。否则，它们将在 main() 结束时自动被终止。

实例

以下简单的实例代码使用 pthread_create() 函数创建了 5 个线程，每个线程输出"Hello Runoob！":

实例

```
#include <iostream>
// 必须的头文件
#include <pthread.h>
using namespace std;
#define NUM_THREADS 5
// 线程的运行函数
void* say_hello(void* args)
{
cout << "Hello Runoob! " << endl;
return 0;
}
int main()
{
// 定义线程的 id 变量，多个变量使用数组
pthread_t tids[NUM_THREADS];
for(int i = 0; i < NUM_THREADS; ++i)
{
//参数依次是：创建的线程id，线程参数，调用的函数，传入的函数参数
int ret = pthread_create(&tids[i], NULL, say_hello, NULL);
if (ret != 0)
{
cout << "pthread_create error: error_code=" << ret << endl;
}
}
//等各个线程退出后，进程才结束，否则进程强制结束了，线程可能还没反应过来；
pthread_exit(NULL);
}
```

使用 -lpthread 库编译下面的程序：

```
$ g++ test.cpp -lpthread -o test.o
```

现在，执行程序，将产生下列结果：

```
$ ./test.o
Hello Runoob!
Hello Runoob!
Hello Runoob!
```

```
Hello Runoob!
Hello Runoob!
```

以下简单的实例代码使用 pthread_create() 函数创建了 5 个线程，并接收传入的参数。每个线程打印一个 "Hello Runoob!" 消息，并输出接收的参数，然后调用 pthread_exit() 终止线程。

实例

```
//文件名: test.cpp
#include <iostream>
#include <cstdlib>
#include <pthread.h>
using namespace std;
#define NUM_THREADS 5
void *PrintHello(void *threadid)
{
// 对传入的参数进行强制类型转换，由无类型指针变为整形数指针，然后再读取
int tid = *((int*)threadid);
cout << "Hello Runoob! 线程 ID, " << tid << endl;
pthread_exit(NULL);
}
int main ()
{
pthread_t threads[NUM_THREADS];
int indexes[NUM_THREADS];// 用数组来保存i的值
int rc;
int i;
for( i=0; i < NUM_THREADS; i++ ){
cout << "main() : 创建线程, " << i << endl;
indexes[i] = i; //先保存i的值
// 传入的时候必须强制转换为void* 类型，即无类型指针
rc = pthread_create(&threads[i], NULL,
PrintHello, (void *)&(indexes[i]));
if (rc){
cout << "Error:无法创建线程," << rc << endl;
exit(-1);
}
}
pthread_exit(NULL);
}
```

现在编译并执行程序，将产生下列结果：

```
$ g++ test.cpp -lpthread -o test.o
$ ./test.o
main() : 创建线程, 0
main() : 创建线程, 1
Hello Runoob! 线程 ID, 0
main() : 创建线程, Hello Runoob! 线程 ID, 21

main() : 创建线程, 3
```

```
Hello Runoob! 线程 ID, 2
main() : 创建线程, 4
Hello Runoob! 线程 ID, 3
Hello Runoob! 线程 ID, 4
```

向线程传递参数

这个实例演示了如何通过结构传递多个参数。您可以在线程回调中传递任意的数据类型，因为它指向 void，如下面的实例所示：

实例

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>
using namespace std;
#define NUM_THREADS 5
struct thread_data{
int thread_id;
char *message;
};
void *PrintHello(void *threadarg)
{
struct thread_data *my_data;
my_data = (struct thread_data *) threadarg;
cout << "Thread ID : " << my_data->thread_id ;
cout << " Message : " << my_data->message << endl;
pthread_exit(NULL);
}
int main ()
{
pthread_t threads[NUM_THREADS];
struct thread_data td[NUM_THREADS];
int rc;
int i;
for( i=0; i < NUM_THREADS; i++ ){
cout <<"main() : creating thread, " << i << endl;
td[i].thread_id = i;
td[i].message = (char*)"This is message";
rc = pthread_create(&threads[i], NULL,
PrintHello, (void *)&td[i]);
if (rc){
cout << "Error:unable to create thread," << rc << endl;
exit(-1);
}
}
pthread_exit(NULL);
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
$ g++ -Wno-write-strings test.cpp -lpthread -o test.o
$ ./test.o
main() : creating thread, 0
main() : creating thread, 1
Thread ID : 0 Message : This is message
main() : creating thread, Thread ID : 21
 Message : This is message
main() : creating thread, 3
Thread ID : 2 Message : This is message
main() : creating thread, 4
Thread ID : 3 Message : This is message
Thread ID : 4 Message : This is message
```

连接和分离线程

我们可以使用以下两个函数来连接或分离线程：

```
pthread_join (threadid, status)
pthread_detach (threadid)
```

pthread_join() 子程序阻碍调用程序，直到指定的 threadid 线程终止为止。当创建一个线程时，它的某个属性会定义它是否是可连接的（joinable）或可分离的（detached）。只有创建时定义为可连接的线程才可以被连接。如果线程创建时被定义为可分离的，则它永远也不能被连接。

这个实例演示了如何使用 pthread_join() 函数来等待线程的完成。

实例

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>
#include <unistd.h>
using namespace std;
#define NUM_THREADS 5
void *wait(void *t)
{
int i;
long tid;
tid = (long)t;
sleep(1);
cout << "Sleeping in thread " << endl;
cout << "Thread with id : " << tid << " ...exiting " << endl;
pthread_exit(NULL);
}
int main ()
{
int rc;
int i;
pthread_t threads[NUM_THREADS];
pthread_attr_t attr;
```

```
void *status;
// 初始化并设置线程为可连接的（joinable）
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
for( i=0; i < NUM_THREADS; i++ ){
cout << "main() : creating thread, " << i << endl;
rc = pthread_create(&threads[i], NULL, wait, (void *)&i );
if (rc){
cout << "Error:unable to create thread," << rc << endl;
exit(-1);
}
}
// 删除属性，并等待其他线程
pthread_attr_destroy(&attr);
for( i=0; i < NUM_THREADS; i++ ){
rc = pthread_join(threads[i], &status);
if (rc){
cout << "Error:unable to join," << rc << endl;
exit(-1);
}
cout << "Main: completed thread id :" << i ;
cout << " exiting with status :" << status << endl;
}
cout << "Main: program exiting." << endl;
pthread_exit(NULL);
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Sleeping in thread
Thread with id : 4  ...exiting
Sleeping in thread
Thread with id : 3  ...exiting
Sleeping in thread
Thread with id : 2  ...exiting
Sleeping in thread
Thread with id : 1  ...exiting
Sleeping in thread
Thread with id : 0  ...exiting
Main: completed thread id :0  exiting with status :0
Main: completed thread id :1  exiting with status :0
Main: completed thread id :2  exiting with status :0
Main: completed thread id :3  exiting with status :0
Main: completed thread id :4  exiting with status :0
Main: program exiting.
```

更多实例参考：http://www.runoob.com/w3cnote/cpp-multithread-demo.html

C++ 信号处理 C++ Web 编程

2 篇笔记

写笔记

c++ 11 之后有了标准的线程库：

```
#include <iostream>

#include <thread>

std::thread::id main_thread_id = std::this_thread::get_id();

void hello()
{
    std::cout << "Hello Concurrent World\n";
    if (main_thread_id == std::this_thread::get_id())
        std::cout << "This is the main thread.\n";
    else
        std::cout << "This is not the main thread.\n";
}

void pause_thread(int n) {
    std::this_thread::sleep_for(std::chrono::seconds(n));
    std::cout << "pause of " << n << " seconds ended\n";
}

int main() {
    std::thread t(hello);
    std::cout << t.hardware_concurrency() << std::endl;//可以并发执行多少个(不准确)
    std::cout << "native_handle " << t.native_handle() << std::endl;//可以并发执行多少个(不
准确)
    t.join();
    std::thread a(hello);
    a.detach();
    std::thread threads[5];                          // 默认构造线程

    std::cout << "Spawning 5 threads...\n";
    for (int i = 0; i < 5; ++i)
        threads[i] = std::thread(pause_thread, i + 1);   // move-assign threads
    std::cout << "Done spawning threads. Now waiting for them to join:\n";
    for (auto &thread : threads)
        thread.join();
```

```
    std::cout << "All threads joined!\n";
}
```

之前一些编译器使用 C++11 的编译参数是 `-std=c++11`

```
g++ -std=c++11 test.cpp -lpthread
```

ztftrue 1年前 (2017-11-06)

要注意内存泄露问题。

如果设置为 **PTHREAD_CREATE_JOINABLE**，就继续用 `pthread_join()` 来等待和释放资源，否则会内存泄露。

bigben 5个月前 (10-06)