C++11並發編程教程- Part 1 ： thread初探

注：文中凡遇通用的術語及行話，均不予以翻譯。譯文有不當之處還望悉心指正。

原文：C++11 Concurrency - Part 1 : Start Threads

C++11引入了一個新的線程庫，包含了用於啟動、管理線程的諸多工具，與此同時，該庫還提供了包括互斥量、鎖、原子量等在內的同步機制。在這個系列的教程中，我將嘗試向大家展示這個新庫提供的大部分特性。

為了能夠編譯本文的示例代碼，你需要有一個支持C++11 的編譯器，筆者使用的是GCC4.6.1 （你需要添加" -std=c++11 "或" -std=c++0x "編譯選項以啟動GCC對C++11的支持）[譯註：bill的編譯環境為GCC4.6.3 + codeblocks 10.05 + Ubuntu 12.04，所使用的編譯選項為" -std=gnu++0x "]。

啟動線程

啟動一個新的線程非常簡單，當你創建一個std::thread 的實例時，它便會自行啟動。創建線程實例時，必須提供該線程將要執行的函數，方法之一是傳遞一個函數指針，讓我們以經典的" Hello world"來闡釋這一方法：

```
#include <thread>
#include <iostream>
void hello(){
    std::cout << "Hello from thread " << std::endl;
}
int main(){
    std::thread t1(hello);
    t1.join();
    return 0 ;
}
```

所有的線程工具均置於頭文件<thread> 中。這個例子中值得注意的是對函數join() 的調用。該調用將導致當前線程等待被join 的線程結束（在本例中即線程main 必須等待線程t1 結束後方可繼續執行）。如果你忽略掉對join() 的調用，其結果是未定義的——程序可能打印出" Hello from thread "以及一個換行，或者只打印出" Hello from thread "卻沒有換行，甚至什麼都不做，那是因為線程main 可能在線程t1 結束之前就 返回了。

區分線程

每個線程都有唯一的ID以便我們加以區分。使用std::thread類的get_id()便可獲取標識對應線程的唯一ID 。我們可以使用std::this_thread來獲取當前線程的引用。下面的例子將創建一些線程並使

它們打印自己的ID：

```
#include <thread>
#include <iostream>
#include <vector>
void hello(){
    std::cout << "Hello from thread " << std::this_thread::get_id() << std::endl;
}
int main(){
    std::vector<std:: thread > threads;
    for ( int i = 0; i < 5; ++i){
        threads.push_back(std:: thread (hello));
    }
    for ( auto & thread : threads){
        thread .join();
    }
    return 0;
}
```

依次啟動線程並將他們存入vector是管理多個線程的常用伎倆，這樣你便可以輕鬆地改變線程的數量。回到正題，就算是上面這樣短小簡單的例子，也不能斷定其輸出結果。理論情況是：

```
Hello from thread 140276650997504
Hello from thread 140276667782912
Hello from thread 140276659390208
Hello from thread 140276642604800
Hello from thread 140276676175616
```

但實際上（至少在我這裡）上述情況並不常見，你很可能得到的是如下結果：

```
Hello from thread Hello from thread Hello from thread 139810974787328Hello from thread
139810983180032Hello from thread
139810966394624
139810991572736
139810958001920
```

或者更多其他的結果。這是因為線程之間存在interleaving。你沒辦法控制線程的執行順序，某個線程可能隨時被搶占，又因為輸出到ostream分幾個步驟（首先輸出一個string，然後是ID，最後輸出

換行），因此一個線程可能執行了第一個步驟後就被其他線程搶占了，直到其他所有線程打印完之後才能進行後面的步驟。

使用Lambda表達式啟動線程

當線程所要執行的代碼非常短小時，你沒有必要專門為之創建一個函數，取而代之的是使用Lambda表達式。我們可以很輕易地將上述例子改寫為使用Lambda表達式的形式：

```
#include <thread>
#include <iostream>
#include <vector>
int main(){
    std::vector<std:: thread > threads;
    for ( int i = 0; i < 5; ++i){
        threads.push_back(std:: thread ([](){
            std::cout << "Hello from thread " << std::this_thread::get_id() << std::endl;
        }));
    }
    for ( auto & thread : threads){
        thread .join();
    }
    return 0;
}
```

如上，我們使用了Lambda表達式替換掉原來的函數指針。毋庸置疑，這段代碼和之前使用函數指針的代碼實現了完全相同的功能。

下篇

在本系列的下一篇文章中，我們將看到如何使用鎖機制保護我們的並發代碼。

C++11並發編程教程- Part 2 :保護共享數據

注：文中凡遇通用的術語及行話，均不予以翻譯。譯文有不當之處還望悉心指正。

原文：C++11 Concurrency - Part 2 : Protect shared data

上一篇文章我們講到如何啟動一些線程去並發地執行某些操作，雖然那些在線程裡執行的代碼都是獨立的，但通常情況下，你都會在這些線程之間使用到共享數據。一旦你這麼做了，就面臨著一個新

的問題——同步。

下面讓我們用示例來闡釋“同步”是個什麼問題。

同步問題

我們就拿一個簡單的計數器作為示例吧。這個計數器是一個結構體，他擁有一個計數變量，以及增加或減少計數的函數，看起來像這個樣子：

[譯註：原文Counter 的value 並未初始化，其初始值隨機，讀者可自行初始化為0]

```
struct Counter {
    int value;
    void increment(){
        ++value;
    }
};
```

這並沒什麼稀奇的，下面讓我們來啟動一些線程來增加計數器的計數吧。

```
int main(){
    Counter counter;
    std::vector<std:: thread > threads;
    for ( int i = 0; i < 5; ++i){
        threads.push_back(std:: thread ([&counter](){
            for ( int i = 0; i < 100; ++i){
                counter.increment();
            }
        }));
    }
    for ( auto & thread : threads){
        thread .join();
    }
    std::cout << counter.value << std::endl;
    return 0;
}
```

[譯註：bill 的測試環境下，上述代碼始終輸出500 ，讀者可將外層for循環條件改為i < 100 ，內層for循環條件改為i < 99999以觀察實驗結果]

同樣的，也沒什麼新花樣，我們只是啟動了5 個線程，每個線程都讓計數器增加100 次而已。等這一工作結束，我們就打印計數器最後的數值。

如果運行這一程序，我們理所當然的期望運行結果是500 ，但事與願違，沒人能保證這個程序最終

輸出什麼。下面是在我的機器上得到的一些結果：

```
442
500
477
400
422
487
```

問題的根源在於計數器的increment() 並非原子操作，而是由3 個獨立的操作組 成的：

1. 讀取value 變量的當前值。
2. 將讀取的當前值加1 。
3. 將加1 後的值寫回value 變量。

當你以單線程運行上述代碼時，就不會出現任何問題，上述三個步驟會按照順序依次執行。但是一旦你身處多線程環境，情況就會變得糟糕起來，考慮如下執行順序：

1. 線程a ：讀取value 的當前值，得到值為0。加1。因此value = 1 。[譯註：此時1並沒有寫回value 內存，原文“ value = 1 ” 僅作邏輯意義，下同]
2. 線程b ：讀取value 的當前值，得到值為0。加1。因此value = 1 。
3. 線程a ：將1寫回value 內存並返回1 。
4. 線程b ：將1寫回value 內存並返回1 。

這種情況源於線程間的interleaving 。Interleaving 描述了多線程同時執行幾句代碼的各種情況。就算僅僅只有兩個線程同時執行這三個操作，也會存在很多可能的interleaving 。當你有許多線程同時執行多個操作時，要想枚舉出所有interleaving ，幾乎是不可能的。而且如果線程在執行單個操作的不同指令之間被搶占，也會導致interleaving 的發生。

目前有許多可以解決這一問題的方案：

- Semaphores
- Atomic references
- Monitors
- Condition codes
- Compare and swap
- etc.

就本文而言，我們將學習如何使用Semaphores 去解決這一問題。事實上，我們僅僅使用了Semaphores 中比較特殊的一種——互斥量。互斥量是一個特殊的對象，在同一時刻只有一個線程能夠得到該對像上的鎖。借助互斥量這種簡而有力的性質，我們便可以解決線程同步問題。

使用互斥量保證Counter 的線程安全

在C++11 的線程庫中，互斥量被放置於頭文件<mutex> ，並以std::mutex 類加以實現。互斥量有兩個重要的函數：lock() 和unlock() 。顧名思義，前者使當前線程嘗試獲取互斥量的鎖，後者則釋放已經獲取的鎖。lock() 函數是阻塞式的，線程一旦調用lock() ，就會一直阻塞直到該線程獲得對應的鎖。

為了使我們的計數器具備線程安全性，我們需要對其添加std::mutex 成員，並在成員函數中對互斥量進行lock()/unlock() 調用。

```
struct Counter {
    std::mutex mutex;
    int value;
    Counter() : value(0) {}
    void increment(){
        mutex.lock();
        ++value;
        mutex.unlock();
    }
};
```

如果我們現在再次運行之前的測試程序，我們將始終得到正確的輸出：500。

異常與鎖

現在讓我們來看看另外一種情況會發生什麼。假設現在我們的計數器擁有一個derement() 操作，當 value 被減為0 時拋出一個異常：

```
struct Counter {
    int value;

    Counter() : value(0) {}
    void increment(){
        ++value;
    }
    void decrement(){
        if (value == 0){
            throw "Value cannot be less than 0" ;
        }
        --value;
    }
};
```

```
14
```

假設你想在不更改上述代碼的前提下為其提供線程安全性，那麼你需要為其創建一個Wrapper 類：

```
struct ConcurrentCounter {
    std::mutex mutex;
    Counter counter;
    void increment(){
        mutex.lock();
        counter.increment();
        mutex.unlock();
    }
    void decrement(){
        mutex.lock();
        counter.decrement();
        mutex.unlock();
    }
};
```

這個Wrapper 將在大多數情況下正常工作，然而一旦decrement() 拋出異常，你就遇到大麻煩了，當異常被拋出時，unlock() 函數將不會被調用，這將導致本線程獲得的鎖不被釋放，你的程序也就順理成章的被永久阻塞了。為了修復這一問題，你需要使用try/catch 塊以保證在拋出任何異常之前釋放獲得的鎖。

```
void decrement(){
    mutex.lock();
    try {
        counter.decrement();
    } catch (std::string e){
        mutex.unlock();
        throw e;
    }
    mutex.unlock();
}
```

代碼並不復雜，但是看起來卻很醜陋。試想一下，你現在的函數擁有10 個返回點，那麼你就需要在每個返回點前調用unlock() 函數，而忘掉其中的某一個的可能性是非常大的。更大的風險在於你又添加了新的函數返回點，卻沒有對應地添加unlock() 。下一節將給出解決此問題的好辦法。

鎖的自動管理

當你想保護整個代碼段（就本文而言是一個函數，但也可以是某個循環體或其他控制結構[譯註：即一個作用域]）免受多線程的侵害時，有一個辦法將有助於防止忘記釋放鎖：std::lock_guard 。

這個類是一個簡單、智能的鎖管理器。當std::lock_guard 實例被創建時，它自動地調用互斥量的lock() 函數，當該實例被銷毀時，它也順帶釋放掉獲得的鎖。你可以像這樣使用它：

```
struct ConcurrentSafeCounter {
    std::mutex mutex;
    Counter counter;
    void increment(){
        std::lock_guard<std::mutex> guard(mutex);
        counter.increment();
    }
    void decrement(){
        std::lock_guard<std::mutex> guard(mutex);
        counter.decrement();
    }
};
```

代碼變得更整潔了不是嗎？

使用這種方法，你無須繃緊神經關注每一個函數返回點是否釋放了鎖，因為這個操作已經被std::lock_guard 實例的析構函數接管了。

總結

現在我們結束了短暫的Semaphores之旅。在本章中你學習瞭如何使用C++線程庫中的互斥量來保護你的共享數據。

但有一點請牢記：鎖機制會帶來效率的降低。的確，一旦使用鎖，你的部分代碼就變得有序[譯註：非並發]了。如果你想要設計一個高度並發的應用程序，你將會用到其他一些比鎖更好的機制，但他們已不屬於本文的討論範疇。

下篇

在本系列的下一篇文章中，我將談及關於互斥量的一些進階概念，並介紹如何使用條件變量去解決一些並發編程問題。

C++11並發編程教程- Part 3 :鎖的進階與條件變量

注：文中凡遇通用的術語及行話，均不予以翻譯。譯文有不當之處還望悉心指正。

原文：C++11 Concurrency Tutorial - Part 3: Advanced locking and condition variables

上一篇文章中我們學習瞭如何使用互斥量來解決一些線程同步問題。這一講我們將進一步討論互斥量的話題，並向大家介紹C++11 並發庫中的另一種同步機制——條件變量。

遞歸鎖

考慮下面這個簡單類：

```
struct Complex {
    std::mutex mutex;
    int i;
    Complex() : i(0) {}
    void mul( int x){
        std::lock_guard<std::mutex> lock(mutex);
        i *= x;
    }
    void div ( int x){
        std::lock_guard<std::mutex> lock(mutex);
        i /= x;
    }
};
```

現在你想添加一個操作以便無誤地一併執行上述兩項操作，於是你添加了一個函數：

```
void both( int x, int y){
    std::lock_guard<std::mutex> lock(mutex);
    mul(x);
    div (y);
}
```

讓我們來測試這個函數：

```
int main(){
    Complex complex;
    complex.both(32, 23);
    return 0;
}
```

如果你運行上述測試，你會發現這個程序將永遠不會結束。原因很簡單，在both()函數中，線程將申請鎖，然後調用mul()函數，在這個函數[譯註：指mul()]中，線程將再次申請該鎖，但該鎖已經被鎖住了。這是死鎖的一種情況。默認情況下，一個線程不能重複申請同一個互斥量上的鎖。

這裡有一個簡單的解決辦法：std::recursive_mutex。這個互斥量能夠被同一個線程重複上鎖，下面

就是Complex結構體的正確實現：

```
struct Complex {
    std::recursive_mutex mutex;
    int i;
    Complex() : i(0) {}
    void mul( int x){
        std::lock_guard<std::recursive_mutex> lock(mutex);
        i *= x;
    }
    void div ( int x){
        std::lock_guard<std::recursive_mutex> lock(mutex);
        i /= x;
    }
    void both( int x, int y){
        std::lock_guard<std::recursive_mutex> lock(mutex);
        mul(x);
        div (y);
    }
};
```

這樣一來，程序就能正常的結束了。

計時鎖

有些時候，你並不想某個線程永無止境地去等待某個互斥量上的鎖。譬如說你的線程希望在等待某個鎖的時候做點其他的事情。為了達到這一目的，標準庫提供了一套解決方案：std::timed_mutex和std::recursive_timed_mutex（如果你的鎖需要具備遞歸性的話）。他們具備與std::mutex相同的函數：lock()和unlock()，同時還提供了兩個新的函數：try_lock_for()和try_lock_until()。
第一個函數，也是最有用的一個，它允許你設置一個超時參數，一旦超時，就算當前還沒有獲得鎖，函數也會自動返回。該函數在獲得鎖之後返回true，否則false。下面我們來看一個簡單示例：

```
std::timed_mutex mutex;
void work(){
    std::chrono::milliseconds timeout(100);
    while ( true ){
        if (mutex.try_lock_for(timeout)){
            std::cout << std::this_thread::get_id() << ": do work with the mutex" <<
std::endl;
            std::chrono::milliseconds sleepDuration(250);
            std::this_thread::sleep_for(sleepDuration);
            mutex.unlock();
            std::this_thread::sleep_for(sleepDuration);
        } else {
            std::cout << std::this_thread::get_id() << ": do work without mutex" << std::endl;
            std::chrono::milliseconds sleepDuration(100);
            std::this_thread::sleep_for(sleepDuration);
        }
    }
```

```
}
int main(){
    std:: thread t1(work);
    std:: thread t2(work);
    t1.join();
    t2.join();
    return 0;
}
```

（這個示例在實踐中是毫無用處的）

值得注意的是示例中時間間隔聲明：std::chrono::milliseconds。它同樣是C++11的新特性。你可以得到多種時間單位：納秒、微妙、毫秒、秒、分以及小時。我們使用上述某個時間單位以設置try_lock_for()函數的超時參數。我們同樣可以使用它們並通過std::this_thread::sleep_for()函數來設置線程的睡眠時間。示例中剩下的代碼就沒什麼令人激動的了，只是一些使得結果可見的打印語句。注意：這段示例永遠不會結束，你需要自己把他kill掉。

Call Once

有時候你希望某個函數在多線程環境中只被執行一次。譬如一個由兩部分組成的函數，第一部分只能被執行一次，而第二部分則在該函數每次被調用時都應該被執行。我們可以使用std::call_once函數輕而易舉地實現這一功能。下面是針對這一機制的示例：

```
std::once_flag flag;
void do_something(){
    std::call_once(flag, [](){std::cout << "Called once" << std::endl;});
    std::cout << "Called each time" << std::endl;
}
int main(){
    std:: thread t1(do_something);
    std:: thread t2(do_something);
    std:: thread t3(do_something);
    std:: thread t4(do_something);
    t1.join();
    t2.join();
    t3.join();
    t4.join();
    return 0;
}
```

```
12
13
14
15
16
```

每一個std::call_once函數都有一個std::once_flag變量與之匹配。在上例中我使用了Lambda表達式[譯註：此處意譯]來作為只被執行一次的代碼，而使用函數指針以及std::function對像也同樣可行。

條件變量

條件變量維護著一個線程列表，列表中的線程都在等待該條件變量上的另外某個線程將其喚醒。[譯註：原文對於以下內容的闡釋有誤，故譯者參照cppreference.com ` 條件變量`一節進行翻譯]每個想要在std::condition_variable上等待的線程都必須首先獲得一個std::unique_lock鎖。[譯註：條件變量的] wait操作會自動地釋放鎖並掛起對應的線程。當條件變量被通知時，掛起的線程將被喚醒，鎖將會被再次申請。

一個非常好的例子就是有界緩衝區。它是一個環形緩衝，擁有確定的容量、起始位置以及結束位置。下面就是使用條件變量實現的一個有界緩衝區。

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
```

```
struct BoundedBuffer {
    int * buffer;
    int capacity;
    int front;
    int rear;
    int count;
    std::mutex lock;
    std::condition_variable not_full;
    std::condition_variable not_empty;
    BoundedBuffer( int capacity) : capacity(capacity), front(0), rear(0), count(0) {
        buffer = new int [capacity];
    }
    ~BoundedBuffer(){
        delete [] buffer;
    }
    void deposit( int data){
        std::unique_lock<std::mutex> l(lock);
        not_full.wait(l, [&count, &capacity](){ return count != capacity; });
        buffer[rear] = data;
        rear = (rear + 1) % capacity;
        ++count;
        not_empty.notify_one();
    }
    int fetch(){
        std::unique_lock<std::mutex> l(lock);
        not_empty.wait(l, [&count](){ return count != 0; });
        int result = buffer[front];
        front = (front + 1) % capacity;
        --count;
        not_full.notify_one();
        return result;
    }
};
```

類中互斥量由std::unique_lock接管，它是用於管理鎖的Wrapper，是使用條件變量的必要條件。我們使用notify_one()函數喚醒等待在條件變量上的某個線程。而函數wait()就有些特別了，其第一個參數是我們的std::unique_lock，而第二個參數是一個斷言。要想持續等待的話，這個斷言就必須返回false，這就有點像while(!predicate()) { cv.wait(l); }的形式。上例剩下的部分就沒什麼好說的了。

我們可以使用上例的緩衝區解決“多消費者/多生產者”問題。這是一個非常普遍的同步問題，許多線程（消費者）在等待由其他一些線程（生產者）生產的數據。下面就是一個使用這個緩衝區的例子：

```
void consumer( int id, BoundedBuffer& buffer){
    for ( int i = 0; i < 50; ++i){
        int value = buffer.fetch();
        std::cout << "Consumer " << id << " fetched " << value << std::endl;
        std::this_thread::sleep_for(std::chrono::milliseconds(250));
    }
}
void producer( int id, BoundedBuffer& buffer){
    for ( int i = 0; i < 75; ++i){
        buffer.deposit(i);
        std::cout << "Produced " << id << " produced " << i << std::endl;
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
}
int main(){
    BoundedBuffer buffer(200);
    std:: thread c1(consumer, 0, std::ref(buffer));
    std:: thread c2(consumer, 1, std::ref(buffer));
    std:: thread c3(consumer, 2, std::ref(buffer));
    std:: thread p1(producer, 0, std::ref(buffer));
    std:: thread p2(producer, 1, std::ref(buffer));
    c1.join();
    c2.join();
    c3.join();
    p1.join();
    p2.join();
    return 0;
}
```

```
20
21
22
23
24
25
26
27
28
```

三個消費者線程和兩個生產者線程被創建後就不斷地對緩衝區進行查詢。值得關注的是例子中使用std::ref來傳遞緩衝區的引用，以免造成對緩衝區的拷貝。

總結

這一節我們講到了許多東西，首先，我們看到如何使用遞歸鎖實現某個線程對同一鎖的多次加鎖。接下來知道瞭如何在加鎖時設定一個超時屬性。然後我們學習了一種調用某個函數有且只有一次的方法。最後我們使用條件變量解決了“多生產者/多消費者”同步問題。

下篇

下一節我們將講到C++11同步庫中另一個新特性——原子量。