18. 并发

18. 并发

现代化体系结构（modern system architecture）通常支持同时执行多个任务（task）和多个线程（thread）。特别是如果采用多处理器内核（multiple processor core），那么程序执行时间可在多线程情况下获得大幅改善。

并行（in parallel）处理也带来了新挑战：不再是完成一个语句后进行另一语句，而是多语句同时执行，于是可能导致并发访问（concurrently accessing）同一资源，造成创建、读取、涂写、 删除等动作不在预期次序下发生，形成不可预测的结果。事实上多线程并发访问数据很容易变成噩梦，带来诸如死锁之类的问题，而“线程之间彼此等待”只能算是最简单的一种情况。

C++11开始，语言自身和标准库都支持并发编程。

- 语言核心定义了一个内存模型，保证当你更改“被两个不同线程使用”的两个object时，它们彼此独立，并引入了一个新关键字thread_local用以定义“变量带有thread专属值”。
- 标准库提供的支持允许你启动多线程，包括得以传递实参、返回数值、跨线程边界传递异常、同步化（synchronize）等，使我们能够对“控制流程”和“数据访问”实现同步化。

标准库在不同的层面分别提供支持。它提供一个高级接口，允许你启动线程，包括传参、处理结果和异常，这是基于若干对应的低层级接口上的。标准库也提供一组低级接口，如 mutex 或 atomic，用来处理放宽的内存次序（relaxed memory order）。

这里只针对一般程序开发者介绍总体观念和典型范例，并且着重于高级接口。底层问题和特性，可以看 C++ Concurrency in Action。

本章组织如下：

- 首先介绍各种多线程启动方法。介绍了高级和低级接口后，开始介绍启动线程的细节。
- 18.4节对于同步化线程（synchronizing thread）所引发的问题提供了一份详细讨论。最主要的问题就是数据的并发访问（concurrent data access）。
- 最后探讨用以 “同步化线程” 和 “并发数据访问” 的各种特性
  - Mutex 和 lock，包括 call_once()
  - Condition variable
  - Atomic

18.1 高级接口：async() 和 Future

初学者想要多线程运行程序，可以使用C++标准库中由 std::async() 和 class std::future<> 提供的高级接口：

- async() 提供了一个接口，让 a piece of functionality，a callable object 作为一个独立线程在后台运行
- Class future<> 允许你等待线程结束并获取其返回结果：返回值或异常

本节详细介绍这些高级接口，并延伸至 class std::shared_future<>，它允许你在多个地方等待和处理线程结果。

18.1.1 async() 和 Futures 使用

假设需要计算两个操作数的总和，这两个操作数是两个函数的返回值。通常做法如下：

```
1 func1() + func2()
```

意味着对操作数的处理是顺序发生的。程序首先调用 fun1() 然后调用 fun2()，或是倒过来（根据语言规则，这次序是 undefined 的）。不论哪种情况，整体处理时间是func1() 所花时间加上func2()所花时间，再加上计算总和所花的时间。

使用多处理器（multiprocessor）的硬件几乎处处可见，因此可以将上述计算做得更好，可以尝试并行运行func1() 和func2()，使其整体运行时间只需是 “func1()和func2()运行时间中的较大者” 加上计算总和的时间。

示例程序：

```
1  // concurrency/async1.cpp
2  #include <future>
3  #include <thread>
4  #include <chrono>
5  #include <random>
6  #include <iostream>
7  #include <exception>
8  using namespace std;
9
10 int doSomething(char c)
11 {
12     //random-number genrator(use c as seed to get different sequences)
13     std::default_random_engine dre(c);
14     std::uniform_int_distribution<int> id(10,1000);
15
16     //loop to print character after a random period of time
17     for(int i=0; i<10; ++i){
```

```
18          this_thread::sleep_for(chrono::milliseconds(id(dre)));
19          cout.put(c).flush();
20      }
21
22      return c;
23  }
24
25  int fun1() {
26      return doSomething(',');
27  }
28
29  int fun2() {
30      return doSomething('+');
31  }
32
33  int main() {
34      std::cout << "starting fun1() in background"
35                << " and func2() in foreground:" << std::endl;
36      //start func1() asynchronously(now or later or never):
37      std::future<int> result(std::async(func1));
38
39      int result2 = func2();    //call fun2() synchronously(here and now)
40
41      //print result(wait for func1() to finish and add its result to result2
42      int result = result1.get() + result2;
43
44      std::cout << "\nresult of func1()+func2(): " << result
45                << std::endl;
46  }
```

首先使用 std::async() 尝试启动 func1() 于后台，并将结果赋值给某个 std::future object。

这里，async() 尝试将其所获得的函数立刻异步启动于一个分离线程内。理想下 fun1() 在这里被启动了，不会造成 main() 阻塞 block。返回 future object 是必要的，原因有以下两点：

1. 它允许你取得 “传给 async() 的那个函数” 的未来结果，也许是返回值或异常。这个 future object 是 “被启动函数” 返回类型的特化，如果被启动的是个返回 void 的后台任务，这个 future object 会是 std::future<void>。
2. 它必须存在，确保 “目标函数” 最终会被调用。注意 async() 只是尝试启动目标函数。如果没有成功，我们需要这个 future obejct 才能强迫启动它。

即使你对后台的那个函数结果不感兴趣，还是需要有这个 future object。

为了能够在启动及控制函数处和返回 future object 之间交换数据，两者都指向一个所谓的 shard state（18.3节）。

接下来启动 func2() 于前台，这是个正常的同步化调用，程序在此阻塞。

如果先前 func1() 成功被 async() 启动且尚未结束，现在 func1() 和 func2() 就是并行。

接下来处理总和。这时候需要 func1() 的结果。我们对先前返回的 future object 调用 get() 来获取结果。随着 get() 被调用，有三种情况其一会发生：

1. func1() 被 async() 启动于一个分离线程并且已结束，你会立即获得其结果
2. func1() 被启动但尚未结束，get() 会引发阻塞（block）直到 func1() 结束后获得结果
3. func1() 尚未启动，会被强迫启动如同一个同步调用；get() 会引发阻塞直到产生结果

这样的行为很重要，因为它确保了在单线程环境中，或是当 async() 无法启动新线程时，程序仍能有效运行。

注意，确保只在最必要时才获取 “被 async() 启动的函数的执行结果”。例如下面的优化就是无效的

```
1 std::future result1(std::async(func1));
2 int result = func2 + result1.get();  //might call func2() after func1() ends
```

为了获得最佳效果，一般需要将调用 async() 和 调用 get() 之间的距离最大化。Call early and return late.

传给 async() 的东西可以是任何类型的 callable object：可以是函数、成员函数、函数对象或 lambda。可采用 inline 型式将应该在专属线程中运行的函数写成一个 lambda 并传递它。

```
1 std::async([]{...})    //try to perform ... asychronously
```

1) 使用 launch 策略

你可以通过传递 launch 来强行让 async 不推迟执行函数，告诉 async 在被调用时明确异步启动目标函数。

```
1 //force func1() to start asynchronously now or throw std::system_error
2 std::future<long> result1 = std::async(std::launch::async, func1);
```

如果异步调用在此处无法实现，程序会抛出一个 std::system_error 异常（4.3.1节），以及错误码 resource_unavailable_try_again，相当于 POSIX 的 errno EAGAIN。

有了 lanch，就不必非得调用 get() 了。但是程序结束前调用 get() 会让行为更加清晰。

如果不将 std::async(std::launch::async, ...) 的结果赋值出去，调用者会在此 block 到目标函数结束，就相当于同步调用（synchronous call）。

类似，你可以强制延迟执行（deferred execution），以 std::launch::deferred 传给 async()。下面例子允许你延迟 func1() 直到你对 f 调用 get()

```
//defer func1 until get()
std::functure<...> f(std::async(std::launch::deferred, func1));
```

这保证 func1() 不会在没有 get() 或 wait() 的情况下启动。这个策略允许你写出 lazy evaluation（懒求值）。例如

```
auto f1 = std::async(std::launch::deferred, task1);
auto f2 = std::async(std::launch::deferred, task2);
...
auto val = thisOrTahtIsTheCase ? f1.get() : f2.get();
```

此外，明确使用 deferred launch 策略有助于在一个单线程环境中模拟 async() 的行为，或是简化调试——除非需要考虑 race condition。

2）处理异常

已经讨论了线程和后台任务成功执行的情况。但是出现异常怎么办呢?

对 future 调用 get() 也能处理异常。当 get() 被调用，且后台操作已经（或由于异常）终止，该异常不会在此线程内被处理，而是会传播出去。因此处理后台操作的异常，主需要同 get() “以同步方式调用该操作” 做出相同动作即可。

```
void task1()
{
    //endless insertion and memory allocation
    //will sooner or later raise an exception
}

int main()
{
    auto f1 = async(task1);
    ...
    try {
        f1.get(); //wait for task1 to finsh(raises exception if any)
    }
    catch (const exception& e) {
        cerr << "EXECEPTION: " << e.what() << endl;
    }
```

```
17 }
```

这个无限循环迟早会出现异常，该异常会终止该线程，因为它未被捕获。Future object 会保持这一状态直到 get() 被调用。使用 get() 后这个异常在 main() 内被进一步传播。

3）等待和轮询（Waiting and Polling）

一个 future<> 只能被调用 get() 一次。之后 future 就处于 valid state，可以通过对 future 调用 valid() 来检测。该情况下对它的任何调用（析构除外）会导致 undefine 行为。（18.3.2节）

但 future 还提供了一个 wait()，允许我们等待后台操作完成而不需要处理其结果。该接口不限制调用次数，可以结合一个 duration 或 timepoint 来限制等待时间。

还有两个类似函数，但它们并不强制启动线程：

1. 使用 wait_for() 并传入一个时间段，就可以让 “异步、运行中” 的操作等待一段时间

```
std::future<...> f(std::async(func));
...
f.wait_for(std::chrono::seconds(10));  //wait at most 10 seconds for func
```

2. 使用 wait_until()，就可以等待直至某特定时间点

```
std::future<...> f(std::async(func));
...
f.wait_until(std::system_clock::now()+std::chrono::minutes(1));
```

wait_for 和 wait_until 都会返回以下之一：

- std::future_status::deferred 如果 async() deferred 操作并且没有调用 wait() 或 get()，该情况下函数会立即返回
- std::future_status::timeout 如果操作已经异步启动但还没完成，但 waiting 又已经超时
- std::future_status::ready 如果操作已完成

wait_for() 或 wait_until() 可以让我们写出 speculative execution（投机）。举例，我们必须在某个时间段内获得运算可用的结果，当然有精确结果更好。

```
int quickComputation();
int accurateComputation();

```

```
 std::future<int> f;  //outside declared because lifetime of accurateComputation()
                      //might exceed lifetime of bestResultInTime()
int bestResultInTime()
{
    //define time slot to get the answer
    auto tp = std::chrono::system_clock::now() + std::chrono::minutes(1);

    //start both a quick and an accurate computaion
    f = std::async(std::launch::async, accurateComputation);
    int guess = quickComputation();

    //return the best computation result we have
    if (s == std::future_status::ready) {
        return f.get();
    }
    else {
        return guess;  //accurateComputation() continues
    }
}
```

注意 future f 不能是声明于 bestResultInTime() 内的 local 对象，那样的话如果时间太短以至于无法完成 accurateComputation()，future 析构函数会阻塞直到异步操作结束。

传递一个 zero duration 或 过去的 timepoint，你可以简单轮询查看后台任务是否启动或在运行。

```
future<...> f(async(task)); // try to call task asynchronously
...
 // do something while task has not finished (might never happen!)
 while (f.wait_for(chrono::seconds(0) != future_status::ready)) {  ...  }
```

注意，这个 loop 永远不会停止，例如在单线程环境中，这个调用会延迟到 get() 被调用。所以应该带 launch 调用 async 或者显式检查 wait_for 是否返回 deferred。

```
future<...> f(async(task)); // try to call task asynchronously
...
// check whether task was deferred:
if (f.wait_for(chrono::seconds(0)) != future_status::deferred) {
    // do something while task has not finished
    while (f.wait_for(chrono::seconds(0) != future_status::ready)) {
        ...
    }
}
```

```
10 ..
11 auto r = f.get(); //force execution of task and wait for result (or exception)
```

引发无限循环的另一可能原因是，运行次循环的线程完全占用处理器，其他线程无法获得时间来 备妥future。这会大大降低程序速度。简单修正就是在循环内调用 yield() 或是 sleep 一小段时间。

18.1.2 实例：等待两个 Task

```
1  // concurrency/async3.cpp
2  void doSomething(char c) {
3      default_random_engine dre(c);
4      uniform_int_distribution<int> id(10, 1000);
5
6      for (int i = 0; i < 10; ++i) {
7              this_thread::sleep_for(chrono::milliseconds(id(dre)));
8              cout.put(c).flush();
9      }
10 }
11
12 int main() {
13     cout << "starting 2 operatioins asynchronously" << endl;
14
15     //start two loops in the background printing characters . or +
16     auto f1 = async([] {doSomething('.');});
17     auto f2 = async([] {doSomething('+');});
18
19     //if at least one of the background tasks is running
20     if (f1.wait_for(chrono::seconds(0)) != future_status::deferred &&
21             f2.wait_for(chrono::seconds(0)) != future_status::deferred) {
22             //poll until at least one of the loops finished
23             while (f1.wait_for(chrono::seconds(0)) != future_status::ready ||
24                     f2.wait_for(chrono::seconds(0)) != future_status::ready) {
25                     this_thread::yield();       //hint to reschedule to the
   next thread
26             }
27     }
28     cout.put('\n').flush();
29
30     //wait for all loops to be finished and process any exception
31     try {
32             f1.get();
33             f2.get();
34     }
```

```
35      catch (const exception& e) {
36              cout << "\nEXCEPTION: " << e.what() << endl;
37      }
38      cout << "\ndone" << endl;
39  }
```

传递实参

使用 lambda 并让它调用后台函数：

```
auto f1 = std::async([=]{  //=: can access objects in scope by value
                        doSomething(c);  //pass copy of c to doSomething()
                    });
```

另一种办法是传递实参给 async()，因为 async() 提供了 callable object 的常用接口。举例，如果你传递 function pointer 作为第一实参给 async()，可以传递更多实参，它们将成为被调用的函数的参数：

```
char c = '@';
auto f = std::async(doSomething, c);  //call doSomething(c) asynchronously
```

也可以引用方式传递给实参，但风险是，被传递值可能在后台任务启动前就是无效的了。

如果以引用方式传递实参是为了在另一个线程中改动它们，很容易造成 undefined 行为。

```
void doSomething(const char& c);
...
char c = '@';
auto f = std::async([&]{doSomething(c);};
...
c = '_';
f.get();
```

字符的变换可能发生在输出循环前、中、后。更糟糕的是，在某一线程改动 c，在另一个线程中读取 c，这是对同一对象的异步并发处理（data race，18.4.1节），将导致 undefined 行为，除非使用 mutex 或 atomic 保护并发处理操作。

所以，如果你使用 async()，就应该以传值方式传递所有 “用来处理目标函数” 的必要 object，使 async() 只需使用局部拷贝（local copy）。若复制成本太高，则让 object 以 const reference 的形式传递，且不使用 mutable。

也可以传递 async() 一个指向成员函数的 pointer。这时，位于该成员函数之后的第一个实参必须是个引用或指针，指向某个object，后者将调用该成员函数。

18.1.3 Shared Future

有时候需要多次处理并发运算的结果，特别是多个其他线程都想处理该结果时。C++标准库提供了class std::shared_future，你可以多次调用 get()，获得相同结果或抛出同一个异常。

```
//start one therad to query a number
std::shared_future<int> f = async(queryNumber);
//or
//auto f = async(queryNumber).share();

//start 3 threads each processing this number in a loop
// f.get() called in doSomething()
auto f1 = async(launch::async, doSomething, ',', f);
auto f2 = async(launch::async, doSomething, '+', f);
auto f3 = async(launch::async, doSomething, '*', f);

//wait for all loops to be finished
f1.get();
f2.get();
f3.get();
```

18.2 低级接口：Thread 和 Promise

除了高级接口 async() 和 (shared) future，标准库还提供了一个启动及处理线程的低级接口。

18.2.1 Class std::thread

要启动一个线程，只需先声明一个 class std::thread 对象，并将目标任务（task）作为初始实参，然后要么等待它结束，要么就将它 detach。

```
void doSomething();
std::thread t(doSomething);  //start doSomething() in the background
...
t.join;  //wait for t to finish(block until doSomething() ends)
```

像 async() 一样，可以传入任何 callable object（可以是 function、member function、function object、lambda），并可夹带任何可能的实参。注意，除非你知道自己在做什么，否则处理函数的所有 object 都应该是以 by value 方式传递，使得 thread 只使用 local copy。

此外，这是个低级接口，这一接口和高级的 async() 相比下不提供哪些性质：

- Class thread 没有 launch 策略。标准库永远只是尝试将目标函数启动于一个新线程中。如果无法做到会抛出 std::system_error 并带着错误码 resource_unavailable_try_agin。
- 没有接口可处理线程结果。唯一可获得的是一个线程ID。
- 如果发生异常，但没有在线程内捕获，程序会立即中止并调用 std::terminate()。若想将异常传播到线程外的某个 context，必须使用 exceptoin_ptr（4.3.3节）。
- 必须声明是否 “想要等待线程结束”（调用 join()）或 “将它从母体 detach” 使它运行于后台而不受任何控制（调用 detach()）。如果你在 thread object 生命周期结束或 move assignment 发生前没有这样做，程序会中止（aborts），调用 std::terminate()。

示例

```
1  //concurrency/thread1.cpp
2
3  void doSomething(int num, char c)
4  {
5      try
6      {
7          default_random_engine dre(42 * c);
8          uniform_int_distribution<int> id(10, 1000);
9          for (int i = 0; i < num; i++)
10         {
11                 this_thread::sleep_for(chrono::milliseconds(id(dre)));
12                 cout.put(c).flush();
13         }
14
15     }
16     catch (const exception& e)
17     {
18         cerr << "THREAD-EXCEPTION (thread "
19              << this_thread::get_id() << "):" << e.what() << endl;
20     }
21 }
22
23 int main()
24 {
25     try
26     {
27         thread t1(doSomething, 5, '1');
28         cout << "- started fg thread " << t1.get_id() << endl;
```

```
29
30          //print other characters in other background threads
31          for (int i = 0;i < 5;i++) {
32                  thread t(doSomething, 10, 'a' + i);
33                  cout << "- detach started bg thread " << t.get_id() << endl;
34                  t.detach();
35          }
36
37          cin.get();          //wait for any input(return)
38          cout << "- join fg thread " << t1.get_id() << endl;
39          t1.join();          //wait for t1 to finish
40      }
41      catch (const std::exception& e)
42      {
43          cerr << "EXCEPTOIN: " << e.what() << endl;
44      }
45  }
```

main() 启动若干线程，让它们都执行 doSomething()。main() 或 doSomething() 都有 try-catch 子句，原因如下：

- main() 中，创建线程可能抛出一个携带错误码 resouce_unavailable_try_again 的异常 std::system_error。
- doSomething() 中，由于此函数被启动为 std::thread，任何异常未被捕获都会造成程序终止。

1）小心 Detached Thread

如果使用 nonlocal 资源，detached thread 很容易产生问题。因为你失去了对 detached thread 的控制，无法轻易得知它是否运行，以及运行多久。因此，不要让一个 detached thread 访问任何生命周期已结束的 object。以 by reference 方式传递变量和 object 给线程，总是有风险的，建议 by value 方式传递。

注意，生命周期问题一样困扰 global 和 static object，因为当程序 exit，detached thread 可能还在运行，它有可能访问已被销毁或正在析构的 global 或 static object，这会导致 undefined 行为。

以下是使用 detached thread 的一般规则：

- detached thread 尽量只访问 local copy
- 如果 detached thread 用上了一个 global 或 static object，应该做到以下之一：
  - 确保这些 global/static object 在 “访问它们的” 所有 detached thread 都结束之前不被销毁。一种做法是使用 condition variable（18.6节），它让 detached thread 用来发送 signal 说明它们已结束。离开main() 或 调用exit() 之前你必须先设置好这些 condition variable，然后发信号说可进行析构了。

- 调用 quick_exit() 结束程序。它以 “不调用 global 和 static object 析构函数” 的方式结束程序。

强制 main thread 等待 detached thread 真正结束。detached thread 很少使用。

2）Thread ID

程序打印 thread ID 时，不是通过 thread object，就是在一个 thread 内使用 name-space this_thread（由<thread>提供）。

class id 有个 default 构造函数，会产生一个独一无二的 ID 用来表示 no thread。可能在被申请时才动态生成 ID。

因此，识别线程的唯一办法是，将线程启动时的ID存储下来，以此为唯一识别值。

已结束的线程 ID 可能会被系统拿去复用。

18.2.2 Promise

新问题：如何在线程间传递参数和处理异常（高级接口如 async() 如何实现这一技术）。想传递数值给线程，你可以把它们当做实参来传递。如果需要线程的运行结果，可用 by reference 方式传递，就像 async() 描述那样。

然而，另一个用来传递运行结果和异常的一般性机制是：class std::promise。promise object 是 future object 的配对，二者都能暂时持有一个 shared state（用来表现一个结果值或一个异常）。但 future object 允许你通过 get() 取回数据，promise object 却是让你通过 set_...() 提供数据。

示例

```
void doSomething(std::promise<std::string>& p)
{
    try {
        //read character and throw exceptoin if 'x'
        std::cout << "read char ('x' for exception): ";
        char c = std::cin.get();
        if (c == 'x') {
            throw std::runtime_error(std::string("char ")+c+" read");
        }
        ...
        std::string s = std::string("char ")+c+" processed";
        p.set_value(std::move(s));  //store result
    }
    catch(...){
        p.set_exceptoin(std::current_exception());  //store exception
    }
```

```cpp
}

int main()
{
    try {
        //start thread using a promise to store the outcome
        std::promise<std::string> p;  //hold string result or exception
        std::thread t(doSomething, std::ref(p));
        t.detach();
        ...
        //create a future to process the outcome
        std::future<std::string> f(p.get_future());
        //process the outcome
        std::cout << "result: " << f.get() << std::endl;
    }
    catch(const std::exception& e) {
    std::cerr << "EXCEPTION: " << e.what() << std::endl;
    }
    catch(...){
    std::cerr << "EXCEPTION" << std::endl;
    }
}
```

声明一个 promise object，令它对持有值或返回值特化（如果两者都是none，特化为void）

promise 内部会创建一个 shared state，被用来存放一个值或异常，并可被 future object 取其数据当作线程结果。

promise 随后被传给一个在分离线程中运行的任务。std::ref() 确保 promise 以 by reference 方式传递，使其状态得以被改变。

然后在线程内调用 set_value() 或 set_exception()，便可以在 promise 中存放一个值或异常。

一旦 shared state 存有某值或某个异常，其状态就变成 ready。你可以在其他地方取出其内容。取出需要借助一个共享相同 shared state 的 future object。对 promise object 调用 get_future() 创建对应的 future object。也可以在启动线程之前先创建该 future object。

现在，通过 get()，我们可以取得被存储的结果，或是令被存储的异常再次被抛出。

注意，get() 会 block 直到 shared state 变成 ready（当 promise 的 set_value() 或 set_exception() 执行后就变成 ready 状态）。但这不意味着 promise 的线程已经结束，该线程可能还执行其他语句，甚至存储其他结果放到其他promise内。

如果想让 shared state 在线程结束时变成 ready，确保线程的 local object 及其他资源在 “结果被处理之前” 清理掉。需要调用 set_value_at_thread_exit() 或 set_exceptoin_at_thread_exit()。

promise 和 future 并不局限于解决多线程问题。在单线程程序中也可以使用 promise 持有一个结果值或一个异常，并且稍后通过一个 future 来处理。

注意，我们不能既存储值又存储异常。这样做会导致 std::future_error。

18.2.3 Class packaged_task<>

async() 给予你一个 handle 句柄，使你可以处理 task 的结果。然而有时虽然你需要处理一个后台 task 的结果，你其实不需要立刻启动该 task。举例，thread pool 可控制何时运行以及多少个后台 task 同时运行，该情况下我们不再这么写：

```
double compute(int x, int y);
std::future<double> f = std::async(conpute,7,5);
...
double res = f.get();
```

而是写成这样：

```
double compute(int x, int y);
std::packaged_task<double(int,int)> task(compute);  //create a task
std::future<double> f = task.get_future();  //get its future
...
task(7,5);  //start the task (typically in a separate thread)
...
double res = f.get();  //wait for its end and process result/exception
```

task 通常（不一定）启动于某一分离线程中。

class std::packaged_task<> 定义于 <future> 内，持有目标函数及其可能结果（也就是该函数的 shared state）。

18.3 细说启动线程

介绍完启动线程和处理返回值或异常的高级、低级接口后，让我们总结这些概念以及一些尚未提及的细节。

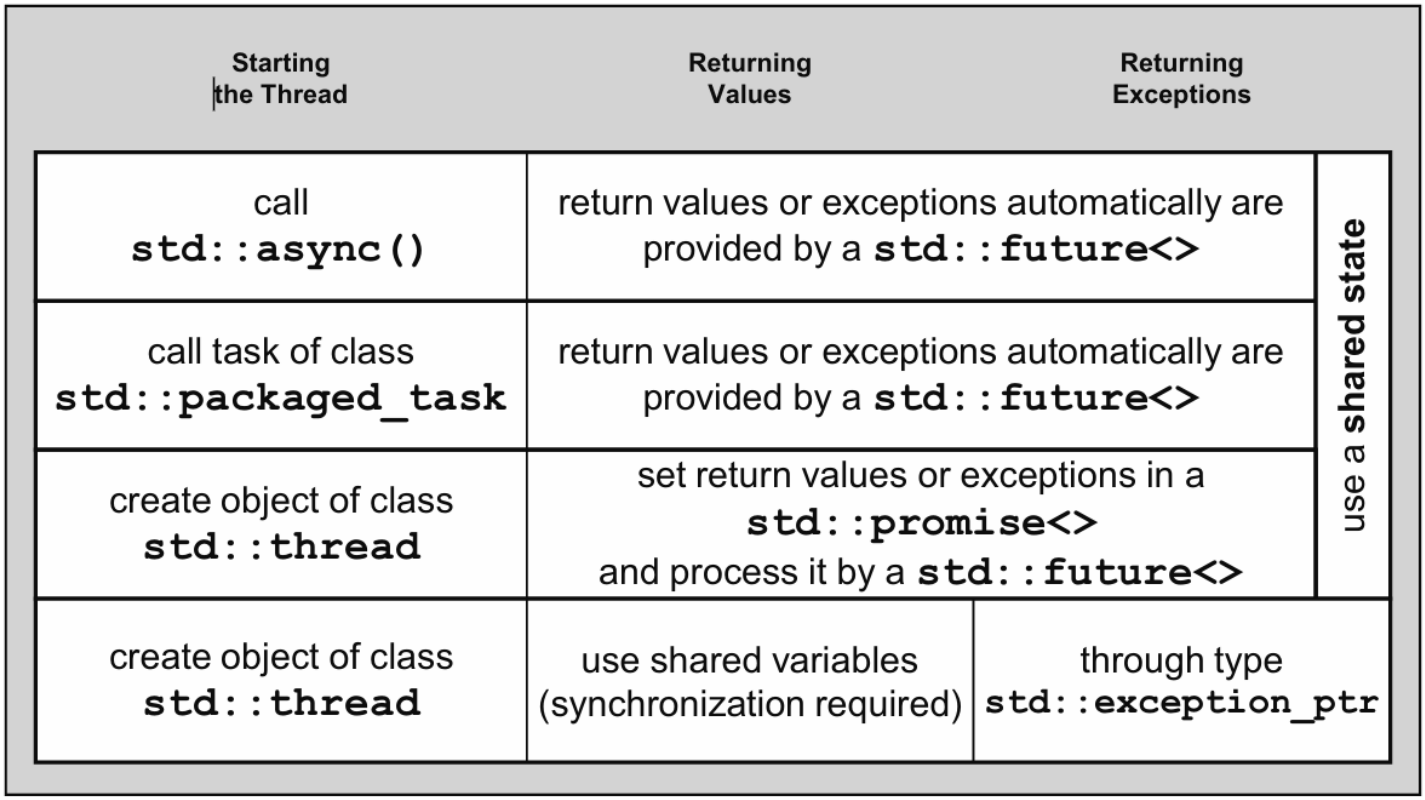

图18.1 Thread接口层级

概念上，我们有数个层面可以启动线程并处理其返回值或异常：

- 低层接口 class thread 可以启动线程。为了返回数据，我们需要可共享变量（global 或 static 变量，或是以实参传递的变量）。为了返回异常，可使用类型 std::exception_ptr（它被 std::current_exception() 反正并可被 std::rethrow_exception() 处理，4.4.3节）。
- shared state 的概念让我们能够以一种便捷的方法处理返回值或异常。搭配低层接口提供的 promise 我们可以创建一个 shared state 然后通过一个 future 来处理它。
- 高级层面中，class packaged_task 或 async() 会自动创建一个 shared state，并且由一个 return 语句或未被捕获的异常 set 设置好。
- 若是使用 std::async()，我们无须关心线程何时真正启动。当需要结果时调用 get()。

Shared State

上述所有性质都用到了这个概念：shared state。它允许启动及控制后台任务的object（promise、package task 或是 async()）能够和 “处理其结果” 的 object（future 或 shared future）相互通信。因此，shared state 必须能够持有被启动的目标函数以及某些状态和结果（一个返回值或异常）。

shared state 如果持有其函数运行结果（或异常），则它是 ready。 shared state 通常被实现为一个 reference-counter object，当它被最后一个使用者释放时就被销毁。

18.3.1 async()

18.3.2 Future

18.3.3 Shared Future

18.3.4 Class std::promise

18.3.5 Class std::packaged_task

18.3.6 Class std::thread

18.3.7 Namespace this_thread

18.4 线程同步化与 Concurrency（并发）问题

使用多线程（multiple thread）总是会伴随着数据的并发访问（concurrent data access）。线程有可能提供数据给其他线程处理，或是准备必要的先决条件（precondition）用来启动其他进程。

讨论同步化线程（synchronized thread）和并发数据处理的技术之前，我们必须先了解问题所在，然后就可以开始讨论以下的线程同步化技术：

- mutex 和 lock，包括 call_once()
- condition variable
- atomic

18.4.1 当心 Concurrency（并发）

多个线程并发处理相同数据而又没有同步化，那么唯一安全的情况就是：所有线程只读取数据。

相同数据指的是使用相同内存区。不同线程并发处理它们手上不同的变量、对象、成员，不会有问题，因为C++11开始每个变量都保证拥有自己的内存区。唯一例外是 bitfield，不同的 bitfield 可能共享同一块内存区，分享对同一块数据的访问。

当两个或更多线程并发处理相同的变量、对象、成员，而且至少其中一个线程改动了它，而你又没有同步化该处理操作，就可能埋下了严重的隐患。这就是 data race，不同线程中的两个相互冲突的操作，其中至少一个操作不是 atomic（不可分割的），且无一个操作发生在另一个操作之前。data race 会导致 undefined 行为。

问题在于，代码也许能正常运作，但却不是永远如此。也许，使用其他数据，或进入生产模式，或换到另一个平台，你的程序突然就完蛋了。所以，使用多线程，要特别小心并发数据访问。

18.4.2 Concurrent Data Access 为什么造成问题

为了解并发数据访问造成的问题，我们必须了解使用并发时C++给了什么保证。一种编程语言如 C++ 总是个抽象层，用以支持不同的平台和硬件（根据其体系结构和目的提供不同的能力和接口）。因此，C++ 标准具体描述了语句和操作的效果，但并非等同于其产生的汇编码（assembler code）。标准描述的是 what 而不是 how。

一般来说，行为不会被定义得太谨慎以至于只能有一种实现。实际上行为有可能不被明确定义。例如，函数调用的参数计算顺序没有具体说明。程序期待明确的计算顺序，会导致 undefined 行为。

因此，重要的问题是：语言给了什么保证？该范围内程序员不应该期望更多。事实上关于 as-if 规则，只要程序行为外观上相同，每个编译器都可以将代码无限优化。因此，被生成的代码是个黑盒子，是可以变化的，只要可观测行为保持稳定。

undefined 行为存在，是为了给予编译器和硬件厂商自由度和能力去生成最佳代码，不管他们的最佳标准是什么。它适用于两者：编译器有可能展开循环，重新安排语句，去除无用代码（dead code），预先获取数据，在现代化体系结构中，一个以硬件实现的 buffer 可能重排 load 或 store。

重排序可以改善程序速度，但它们也可能产生破坏行为。

18.4.3 什么情况下可能出错

C++ 中我们可能会遇到一下问题：

1）unsynchronized data access（未同步化的数据访问）

并行的两个线程读写同一笔数据，不知道哪一个语句先运行。

```
if (val>=0) {
    f(val);
}
else {
    f(-val);
}
```

多线程处理val，val值有可能在if子句和调用f()之间被改变，造成负值被传给f()。下面同理

```
std::vector<int> v;
...
if (!v.empty()) {
    std::cout<<v.front()<<std::endl;
}

//v没有足够元素就会抛出异常的保证不再成立
//当at()被调用时另一个线程有可能改动v
v.at(5);
```

除非另有说明，C++标准库提供的函数通常不支持写/读操作与另一个写操作（写到同一笔数据）并发执行。

也就是说，来自多线程对同一object的多次调用会导致 undefined 行为。

但是C++标准库对于线程安全还是提供了一些保证。例如：

- 并发处理同一容器内的不同元素是可以的（vector<bool>除外）。因此，不同线程可以并发读/写同一容器内的不同元素。例如，每个线程可以处理某些事，然后将结果存储于一个共享的 vector内专属于该线程的某元素。
- 并发处理 string stream、file stream 或 stream buffer 会导致 undefined 行为。但是格式化输入自和输出至某个标准 stream （它被 C I/O 同步化了，15.14.1节）是可以的，虽然可能导致插叙的字符。

2）half-written data（写了一半的数据）

某个线程正在读数据，另一个线程改动它，于是读取中的线程可能读到改了一般的数据，一个半新旧值。

```
1 //有一个变量
2 long long x = 0;
3
4 //某个线程对它写入数值
5 x = -1;
6
7 //另一个线程读取它
8 std::cout << x;
```

第二线程输出x时它读到哪个值?

- 0（x的旧值），若第一线程尚未赋予它-1
- -1（x的新值），若第一线程已经赋予它-1
- 任何其他值，若第二线程在第一线程对x赋值-1的过程中读取x

3）reordered statement（重排的语句）

语句和操作有可能被重排序，也许对于单线程是正确的，但对于多个线程的组合却破坏了预期的行为。

举例，有两个共享对象，一个是 long，用来将data从某个线程传递到另一个线程，另一个是 bool readyFlag，用来表示第一线程是否已提供数据：

```
long data;
bool readyFlag = false;

//生成端调用
data = 42;
readyFlag = true;

//消费端调用
while(!readyFlag) {  //loop until data is ready
    ;
}
foo(data);

```

以上做法将 “某线程中对data的设定” 和 “另一线程中对data的消费” 同步化。

在不知任何细节的情况下，几乎每个程序员都会认为第二线程一定是在data有值42之后才调用foo()，认为对foo()的调用只有在readyFlag是true的前提下才能到达，而那又只有发生在第一线程将42赋值给data之后，因为赋值之后才令readyFlag变成true。

事实上第二线程的输出有可能是data在第一线程赋值42之前的旧值（甚至任何值，因为赋值操作可能只做了一半）。

也就是说，编译器和硬件有可能重排语句，使得实际执行一下操作：

```
readFlag = true;
data = 42;
```

一般来说，这样的重排序是允许的，因为C++只要求编译所得的代码在单一线程的可观测行为正确。

同理，第二线程也可能被重排语句，前提不影响该线程的行为。

18.4.4 解决问题所需要的性质（Feature）

为解决并发数据访问的三个主要问题，需要先建立以下概念：

- atomicity（不可切割性）：这意味着读或写一个变量，或是一串语句，其行为是独占的、排他的，无任何打断，因此一个线程不可能读到 “因另一线程造成的” 中间状态。
- order（次序）：我们需要一些方法保证 “具体指定语句” 的次序。

- 可以使用 future 和 promise，它们都保证 atomicity 和 order：一定是在生成结果（返回值或异常）之后才设定 shared state，这意味着读和写不会并发发生。
- 可以使用 mutex 和 lock 来处理 critical section 或 protected zone，借此授予独占权利，使得（例如）一个 “check 操作” 和一个 “依赖该 check 结果的操作” 之间不会发生任何事。lock 提供 atomicity，它会阻塞所有使用 second lock 的行为，直到作用于相同资源上的 first lock 被释放。更准确的说，被某个线程获得的 lock object，它被另一线程获得之前必须先被释放。但是如果两个线程使用 lock 来处理数据，每次运行的次序有可能发生变化。
- 可以使用 condition variable 令某线程等待若干 “被另一线程控制的” 判断式（predicate）变为 true。这有助于处理多线程间的次序，允许一或多个线程处理其他一或多个线程所提供的数据或状态。
- 可以使用 atomic data type 确保每次对变量或对象的访问都是不可切割的（atomic），而原子类型的操作顺序保持稳定。
- 可以使用原子数据类型的低级接口，它允许专家放宽原子语句的顺序或使用手动屏障（barrier）进行内存访问（所谓 fences 栅栏）

高级特性如 future 和 promise 或 mutex 和 lock 容易使用，风险较低。

底层特性如 atomic 和其底层接口，也许能提供较佳性能，但也增加了误用的风险。但底层特性有时候可以为某些特定的高级问题提供简单解法。

有了 atomic，我们可以进行 lock-free 编程，但那是专家偶尔也会出错的领域。

volatile 和 concurrency

注意，并没有说 volatile 是个用来解决并发数据访问问题的 feature。虽然你可能因为以下原因而有那样的期待：

- volatile 是个 C++ 关键字，用来阻止过度优化。也就是说，每次访问这个变量时都要从它的内存地址中读取，而不是使用可能已经存储在寄存器中的值。这主要用于处理可能会被硬件或者并发线程意外改变的内存位置。

  然而，C++11 标准明确指出，`volatile` 不应该用于线程同步或者实现内存模型。对于多线程编程，C++11 引入了更复杂的原子操作（`std::atomic`）和内存模型。

- Java 中，volatile 对于 atomicity 和 order 提供了某些保证。它不仅保证了变量的可见性（即一个线程对变量的修改对其他线程是立即可见的），还禁止了指令重排（reordering）的优化，从而确保了一些有序性（ordering）的保证。

  Java 的 `volatile` 关键字经常用于多线程编程中，确保共享变量的正确同步。它通常用于实现轻量级的线程间通信，特别是当不需要使用锁的时候。

差异

- **可见性**：在 Java 中，`volatile` 提供了变量的可见性保证，而 C++ 的 `volatile` 则没有这样的语义。在 C++ 中，可见性通常是通过其他机制（如互斥锁或原子操作）来保证的。

- **有序性**：Java 的 `volatile` 禁止了指令重排，而 C++ 的 `volatile` 并不提供这样的保证。在 C++ 中，指令重排可能由编译器或硬件进行优化。
- **用途**：在 C++ 中，`volatile` 主要用于与硬件或中断等直接交互的场景，而在 Java 中，它则更多地用于多线程编程中的轻量级同步。

C++ 中，volatile 只具体表示对外部资源（像共享内存）的访问不该被优化掉。如果没有 volatile，编译器也许会消除对同一块共享内存区看似多余的 load，只因它在整个程序中看不到这个区域的任何改变。但是在 C++，volatile 既不提供 atomicity 也不提供特别的 order。因此 volatile 语义在 C++ 和 Java 之间有些差异。

18.5 Mutex 和 Lock

mutex，mutual exclusion（互斥体），是个 object，用来协助采取独占排他（exclusive）方式控制“对资源的并发访问”。这里的资源可能是个 object，或多个 object 的组合。为了获得独占资源访问的能力，相应的线程必须 lock mutex，这样可以防止其他线程也 lock mutex，直到第一个线程 unlock mutex。

18.5.1 使用 Mutex 和 Lock

将并发访问同步化的一个粗劣做法是，引入 mutex，用来赋予独占控制权。这个简单方法可能会变得十分复杂。举例，你应该确保异常情况下，它会 unlock 相应的 mutex，否则资源可能被永远锁住。此外可能出现 deadlock 情景：两个线程在释放它们自己的 lock 之前彼此等待对方的 lock。

C++标准库尝试处理这些问题（但目前仍无法从概念上根本解决）。举例，面对异常你不该自己 lock/unlock mutex，应该使用 RAII 守则（Resource Acquisition Is Initialization），构造函数将获得资源，而析构函数（或异常造成生命周期结束）释放资源。为此，标准库提供了 std::lock_guard:

```
int val;
std::mutex valMutex;  //control exclusive access to val
...
{
    std::lock_guard<std::mutex> lg(valMutex);  //lock and automatically unlock
    if (val>=0) {
        f(val);
    }
    else {
        f(-val);
    }
}  //ensure that lock gets released here
```

注意，这样的 lock 应该被限制在最短周期内，因为它会阻塞其他代码并行。由于析构函数会释放这个 lock，你也许会想明确使用大括号，令lock在更进一步语句被处理前先被释放。

1）递归的（recursive）lock

有时候，递归锁定（to lock recursively）是必要的，典例是 active object 或 monitor，它们在每个 public 函数内放一个 mutex 并取得其 lock，用来防止 data race 破坏对象的内部状态。例如下面的函数接口：

```
class DatabaseAccess
{
private:
    std::mutex dbMutex;
    ...  //state of database access
public:
    void createTable(...)
    {
        std::lock_guard<std::mutex> lg(dbMutex);
        ...
    }
    void insertData(...)
    {
        std::lock_guard<std::mutex> lg(dbMutex);
        ...
    }
    ...
};
```

当我们引入一个public成员函数而它可能调用其他public成员函数，情况变得复杂：

```
void createTableAndInsertData(...)
{
    std::lock_guard<std::mutex> lg(dbMutex);
    ...
    createTable(...);  //ERROR: deadlock because dbMutex is locked again
}
```

调用 createTableAndInsertData() 会造成 deadlock，因为它锁住 dbMutex 之后调用 createTable()，造成后者尝试再次 lock dbMutex，那将会 block 直到 dbMutex 变为可用，但这绝不会发生。因为 createTableAndInsertData() 会 block 直到 createTable() 完成。

使用 recursive_mutex 可解决上述问题。这个 mutex 允许同一线程多次锁定，并在zvjn对应的 unlock() 时释放 lock：

```
class DatabaseAccess
{
private:
    std::recursive_mutex dbMutex;
    ... //state of database access
public:
    void createTable(...)
    {
        std::lock_guard<std::recursive_mutex> lg(dbMutex);
        ...
    }
    void insertData(...)
    {
        std::lock_guard<std::recursive_mutex> lg(dbMutex);
        ...
    }
    void createTableAndinsertData(...)
    {
        std::lock_guard<std::recursive_mutex> lg(dbMutex);  //1st lock
        ...
        createTable(...);  //OK: no deadlock, release
                           //2nd lock, and release the lock when function end
    }  //release the 1st lock

}
```

2）尝试性的（tried） lock 和带时间的（timed） lock

有时程序想要获得一个 lock 但若不可能成功的话它又不想永远 block。为此，mutex 提供成员函数 try_lock()。

为了仍能使用 lock_guard（使当下作用域的任何出口都会自动 unlocked mutex），可以传一个额外实参 adopt_lock 给其构造函数：

```
std::mutex m;
//try to acquire a lock and do other stuff while this isn't possible
while(m.try_lock() == false) {
    doSomeOtherStuff();
}
std::lock_guard<std::mutex> lg(m, std::adopt_lock);
...
```

注意，try_lock() 可能假性失败（spuriously），就是即使 lock 没有被他人拿走它也可能失败。

为了等待特定长的时间，可以使用 timed mutex。有两个特殊 mutex，std::timed mutex 和 std::recursive_timed_mutex 允许你调用 try_lock_for() 或 try_lock_until()，用以等待某个时间段，或直到某个时间点。对于实时需求或避免可能的deadlock有帮助。

```
std::timed_mutex m;
//try for one second to acquire a lock
if(m.try_lock_for(std::chrono::seconds(1))) {
    std::lock_guard<std::timed_mutex> lg(m, std::adopt_lock);
    ....
}
else {
    couldNotGetTheLock();
}
```

注意，处理系统时间调整（5.7.5节）时，try_lock_for() 和 try_lock_until() 往往有差异。

3）处理多个 lock

通常一个线程一次只锁定一个 mutex，但是有时必须锁定多个 mutex（例如为了传送数据，从一个受保护资源到另一个受保护资源）。

这种情况如果使用之前的 lock 机制来处理，会变得复杂且有风险：你可能取得第一个 lock 却拿不到第二个 lock，可能发生 deadlock（如果以不同次序去锁住相同的 lock）。

标准库提供了若干函数，可以锁定多个 mutex。例如：

```
std::mutex m1;
std::mutex m2;
{
    std::lock(m1,m2);  //lock both mutexes(or none if not possible)
    std::lock_guard<std::mutex> lockM1(m1, std::adopt_lock);
    std::lock_guard<std::mutex> lockM2(m2, std::adopt_lock);
}  //automatically unlock all mutexes
```

全局函数 std::lock() 会锁住它收到的所有 mutex，而且 block 直到所有mutex都被锁定或直到抛出异常。如果抛出异常，已被锁定的mutex都会被unlock。同样，锁定后你应该使用 lock guard，并以 adopt_lock 作为初始化的第二实参，确保任何情况下这些 mutex 在离开作用域时会被 unlock。注意 lock() 函数提供了一个 deadlock 回避机制，也意味着多个lock的锁定次序并不明确。

全局函数 std::try_lock() 会在取得所有lock的情况下返回-1，否则返回第一个失败的lock的索引。注意，try_lock() 不提供 deadlock 回避机制，但它保证以出现在实参列的次序来试着完成锁定。

只调用 lock() 或 try_lock() 却没有把 lock adopt 给一个 lock guard 的做法是错误的。离开作用域时不会自动unlock。

4）unique_lock

除了 lock_guard<>，标准库还提供了 unique_lock<>，它处理mutex更有弹性。unique_lock<> 提供的接口和 lock_guard<> 相同，但又允许明确写出 “何时” 以及 “如何” lock 或 unlock 其 mutex。因此其 obejct 可能拥有一个被锁住的mutex（或者说拥有一个mutex）。lock_guard 不同，它的object生命周期中总是锁定一个mutex。对 unique lock 可以调用 owns_lock() 或 bool() 来查询其 mutex 当前是否被锁定。

这个 class 的优点还是，如果析构时mutex仍被锁住，其析构函数会自动调用unlock()。如果当时没有锁住mutex，则析构函数不做任何事。

和 lock_guard 比较，unique_lock 添加了以下三个构造函数：

- 可以传递 try_to_lock，表示试图锁定mutex但不希望阻塞：

```
std::unique_lock<std::mutex> lock(mutex, std::try_to_lock);
...
if(lock) {  //if lock was successful
    ...
}
```

- 可以传递一个时间段或时间点给构造函数，表示尝试在一个明确的时间周期内锁定：

```
std::unique_lock<std::timed_mutex> lock(mutex, std::chrono::seconds(1));
```

- 可以传递defer_lock，表示初始化这一lock object但尚未打算锁住mutex：

```
std::unique_lock<std::mutex> lock(mutex, std::defer_lock);
...
lock.lock();  //or(timed)try_lock()
```

上述的 defer_lock flag 可以用来建立一个或多个 lock 并于稍后才锁定它们：

```
std::mutex m1;
std::mutex m2;

```

```
std::unique_lock<std::mutex> lockM1(m1, std::defer_lock);
std::unique_lock<std::mutex> lockM2(m2, std::defer_lock);
...
std::lock(m1, m2);  //lock both mutexes(or none if not possible)
```

unique_lock 提供 release() 用来释放 mutex，或是将其 mutex 拥有权转移给另一个 lock（18.5.2 节）。

有了 lock_guard 和 unique_lock，现在可以实现一个例子，以轮询（polling）某个 ready flag 的方式，另一个线程等待另一个线程：

```
#include <mutex>
...
bool readyFlag;
std::mutex readFlagMutex;

void thread1()
{
    //do something thread2 needs as preparation
    ...
    std::lock_guard<std::mutex> lg(readyFlagMutex);
    readyFlag = true;
}

void thread2()
{
    //wait until readyFlag is true (thread1 is done)
    {
        std::unique_lock<std::mutex> ul(readyFlagMutex);
        while(!readyFlag) {
            ul.unlock();
            std::this_thread::yield();  //hint to reschedule to the next thread
            std::this_thread::sleep_for(std::chrono::milliseconds(100));
            ul.lock()
        }
    }  //release lock
    //do whatever shall happen after thread1 has prepared things
    ...
}
```

代码解释：

- 为什么使用 mutex 来控制 readyFlag 的读写，回忆本章开始介绍的准则：任何带有至少一个write 的并发处理都应该被同步化。

- 为什么声明 readyFlag 时不需要 volatile（允许thread2() 中对其多次读取可被优化掉）。注意：这些对readyFlag的读取发生在critical section内（就是在某个lock的设立和解除之间）。这样的代码不得以 “读写动作被移出critical section之外” 的形式被优化。...

这样对于某一满足条件的轮询（polling）通常不是好办法，更好的做法是使用 conditoin variable（条件变量）18.6.1节。

18.5.2 细说 Mutex 和 Lock

1）细说 class mutex

C++标准库提供了以下 mutex class（表18.6）：

表18.6 各种Mutex及其能力

操作	mutex	recursive_mutex	timed_mutex	recursive_timed_mutex
lock()	获取 mutex（若没有得到则阻塞）			
try_lock()	获取 mutex（若没有得到则返回 false）			
unlock()	unlock 被锁定的 mutex			
try_lock_for()	-	-	尝试在时间段内获取一个 lock	
try_lock_until()	-	-	尝试获取一个 lock 直到某个时间点	
多个 lock	否	是（同一线程）	否	是（同一线程）

- Class std::mutex，同一时间只可被一个线程锁定。如果它被锁住，任何其他 lock() 都会阻塞，直到这个 mutex 再次可用，且 try_lock() 会失败。
- Class std::recursive_mutex，允许在同一时间多次被同一线程获得其 lock。典例是：函数获取一个 lock 并调用另一函数而后者再次获取相同的 lock。
- Class std::timed_mutex 额外允许你传递一个时间段或时间点，用来定义多长时间内它可以尝试获取一个 lock。提供了 try_lock_for() 和 try_lock_until()。
- Class std::recursive_timed_mutex 允许同一线程多次取得其 lock，可指定期限。

表18.7 Mutex Class 的操作函数

操作	效果

mutex m	Default 构造函数，创建一个未锁定的 mutex
m.~mutex()	销毁 mutex（它必须未被锁定）
m.lock()	尝试锁住 mutex（会造成阻塞）
m.try_lock()	尝试锁定 mutex。如果锁定成功就返回 true
m.try_lock_for(dur)	尝试在时间段 dur 内锁定。锁定成功就返回 true
m.try_lock_until(tp)	尝试在时间点 tp 之前锁定
m.unlock()	解锁 mutex。如果它未被锁定则行为 undefined
m.native_handle()	返回一个因平台而异的类型 native_handle_type，这是为了不具有可移植性的扩展

2）细说 class lock_guard

std::lock_guard 提供了一个接口，用以确保一个 locked mutex 在离开作用域时总是会被释放。它的整个生命期间总是与一个 lock 相关联，也许是被显示申请，也许是构造期间 adopted。

表18.8 Class lock_guard 的操作函数

操作	效果
lock_guard lg(m)	为 mutex m 创建一个 lock guard 并锁定它
lock_guard lg(m, adopt_lock)	为已经被锁定的 mutex m 创建一个 lock guard
lg.~lock_guard()	解锁 mutex 并销毁 lock guard

3）细说 class unique_lock

Class std::unique_lock，为一个不一定锁定（或拥有）的 mutex 提供一个 lock guard。

表18.9 Class unique_lock 的操作函数

操作	效果
unique_lock l	defualt 构造函数，创建一个 lock 但不关联任何 mutex
unique_lock l(m)	为 mutex m 创建一个 lock guard 并锁定它
unique_lock l(m, adopt_lock)	为已锁定的 mutex m 创建一个 lock guard
unique_lock l(m, defer_lock)	为 mutex m 创建一个 lock guard 但是不锁定它

unique_lock l(m, try_lock)	为 mutex m 创建一个 lock guard 并试图锁定它
unique_lock l(m, dur)	为 mutex m 创建一个 lock guard 并试图在时间段 dur 内锁定它
unique_lock l(m, tp)	为 mutex m 创建一个 lock guard 并试图在时间点 tp 之前锁定它
unique_lock l(rv)	move 构造函数；将 lock state 从 rv 移动到 l（rv 不再关联任何 mutex）
l.~unique_lock()	解锁 mutex（如果它被锁定）并销毁 lock guard
unique_lock l = rv	move assignment；将 lock state 从 rv 移动到 l（rv 不再关联任何 mutex）
swap(l1, l2)	交换 lock
l1.swap(l2)	交换 lock
l.release()	返回一个 pointer 指向关联的 mutex 并释放它
l.owns_lock()	如果关联的 mutex 被锁定则返回 true
if(l)	检查关联的 mutex 是否被锁定
l.mutex()	返回一个 pointer 指向关联的 mutex
l.lock()	锁定关联的 mutex
l.try_lock()	尝试锁住关联的 mutex（如果成功就返回 true）
l.try_lock_for(dur)	尝试在时间段 dur 内锁住关联的 mutex（如果成功就返回 true）
l.try_lock_until(tp)	尝试在时间点 tp 之前锁住关联的 mutex（如果成功就返回 true）
l.unlock()	解除关联的 mutex

18.5.3 只调用一次

有时某些函数初次被某个线程调用过后，其他线程就不需要它了。典例是 lazy initialization （延迟初始化）。

单线程的做法很简单：以一个 bool flag 表示这个函数是否已被调用。

```
1 bool initialized = false;  //global flag
2 ...
3 if(!initialized) {    //initialize if not initialized yet
4     initialize();
```

```
    initialized = true;
}

//or
static std::vecotor<std::string> staticData;
void foo()
{
    if(statucData.empty()) {
        staticData = initializeStaticData();
    }
    ...
}
```

但是这样的代码在多线程环境下行不通，因为如果多个线程检查初始化是否尚未发生然后启动初始化，可能发生 data race。你必须针对并发访问保护 “检查及初始化” 程序块。

同样，你可以使用 mutex，但标准库提供了一个特殊方案，只需要使用一个 std::once_flag 以及调用 std::call_once（由<mutex>提供）：

```
std::once_flag oc;  //global flag
...
std::call_once(oc, initialize);  //initialize if not initialized yet

//or
static std::vector<std::string> staticData;
void foo()
{
    static std::once_flag oc;
    std::call_once(oc,[]{staticData = initializeStaticData();});
    ...
}
```

传给 call_once() 的第一实参是相应的 once_flag。下一实参是 callable object，如 function、member function、function object 或 lambda，还可再加上其他实参给被调用的函数使用。

因此，多线程中的延迟初始化如下：

```
class X{
private:
    mutable std::once_flag initDataFlag;
    void initData() const;
public:
    data getData() const {
        std::call_once(initDataFlag, &X::initData, this);
```

```
    }
}
```

18.6 Condition Variable（条件变量）

有时，被不同线程执行的 task 必须彼此等待。所以对并发操作实现同步化除了 data race 之外还有其他原因。

future 从某线程传递数据到另一线程只能一次。它的主要目的是处理线程的返回值或异常。

本节介绍 condition variable，它用来同步化线程之间的数据流逻辑依赖关系。

18.6.1 Condition Variable（条件变量）的意图

18.5.1 4）介绍了让某线程等待另一线程的粗浅办法。但是这样针对目标条件而轮询的操作通常不是好的解决办法。

等待中的线程消耗CPU时间重复检验flag，且当它锁住mutex时，负责设置readyflag的那个线程会被阻塞。此外我们很难找出适当的sleep周期：两次检查若间隔太短则线程仍旧太浪费CPU时间在检查动作上，若太长则也许等待的task已完成而线程却还继续sleeping，导致发生延误。

较好的做法是使用 condition variable，C++ 标准库在<condition_variable>提供了它。它是个变量，通过它，一个线程可以唤醒一或多个其他等待中的线程。

原则上，condition variable 运作如下：

- 必须同时包含 <mutex> 和 <condition_variable>，并声明一个 mutex 和 condition variable

```
#include <mutex>
#include <condition_variable)

std::mutex readyMutex;
std::condition_variable readyCondVar;
```

- 激活 “条件终于满足” 的线程（或多线程之一）必须调用

```
readyCondVar.notify_one();  //notify one of the waiting threads
//or
readyCondVar.notify_all();  //notify all the waiting threads
```

- 那个 “等待条件被满足” 的线程必须调用

```
std::unique_lock<std::mutex> l(readyMutex);
readyCondVar.wait(l);
```

condition variable可能假醒（spurious wakeup），就是某个condition variable的 wait 操作可能在该condition variable尚未被 notified 时便返回。

因此，发生wakeup不一定意味着线程所需要的条件已经掌握了。在唤醒后你仍需要代码去验证条件实际已达成。 因此我们必须检查数据是否真正准备好，或是仍需要诸如 ready flag 之类的东西。为了设置和查询他端提供的数据或ready flag，可使用同一个mutex。

18.6.2 Condition Variable 的一个完整例子

```
//concurrency/condvar1.cpp
bool readyFlag;
std::mutex readyFlag
std::condition_variable readyCondVar;

void thread1()
{
    //do something thread2 needs as preparation
    ...
    //signal that thread1 has prepared a condtion
    {
        std::lock_guard<std::mutex> lg(readyMutex);
        readyFlag = true;
    }  //release one
    readyCondVar.notify_one();
}

void thread2()
{
    //wait until thread1 is ready(readyFlag is true)
    {
        std::unique_lock<std::mutex> ul(readyMutex);
        readyCondVar.wait(ul, []{ return readyFlag; });
    }  //release lock

    //do whatever shall happen after thread1 has prepared things
    std::cout << "done" << std::endl;
}

int main()
```

```
{
    auto f1 = std::async(std::launch::async, thread1);
    auto f2 = std::async(std::launch::async, thread2);
}
```

需要三个东西以便在线程间通信：

- 一个用来存放待处理数据的对象，或一个用来表示条件满足了的flag（readyFlag）
- 一个mutex （readyMutex）
- 一个condition variable （readyCondVar）

数据提供者 thread1() 锁住 readyMutex，更新条件（数据），解锁 mutex，然后通知 condition variable。注意通知操作不需要在 lock 保护区内。

等待者线程则是以一个 unique_lock 锁住mutex，等待被通知直到检查好条件，然后释放锁。

这里 condition variable 的 wait() 函数把一个 lambda 当做第二实参，用来二次检测条件是否真的满足。其效果是 wait() 内部会不断调用该第二实参，直到它返回 true，用来处理假醒。

可能会说这个例子不好，因为使用 future 会阻塞直到某些数据到达。来看看第二个例子。

18.6.3 使用 Condition Variable 实现多线程 Queue

三个线程都把数值 push 到某个queue，另两个线程则是从中读取数据：

```
//concurrency/condvar2.cpp
std::queue<int> queue;
std::mutex queueMutex;
std::condition_variable queueCondVar;

void provider(int val)
{
    for (int i = 0; i < 6; ++i) {
        {
            std::lock_guard<std::mutex> lg(queueMutex);
            queue.push(val + i);
        }  //release lock
        queueCondVar.notify_one();

        std::this_thread::sleep_for(std::chrono::milliseconds(val));
    }
}

void consumer(int num)
{
```

```
    //pop value if available (num identifies the consumer)
    while (true)
    {
        int val;
        {
            std::unique_lock<std::mutex> ul(queueMutex);
            queueCondVar.wait(ul, [] { return !queue.empty(); });
            val = queue.front();
            queue.pop();
        }  //release lock
        std::cout << "consumer" << num << ": " << val << std::endl;
    }
}

int main()
{
    //start three providers for values 100+, 300+, 500+
    auto p1 = std::async(std::launch::async, provider, 100);
    auto p2 = std::async(std::launch::async, provider, 300);
    auto p3 = std::async(std::launch::async, provider, 500);

    //start two consumers printing the values
    auto c1 = std::async(std::launch::async, consumer, 1);
    auto c2 = std::async(std::launch::async, consumer, 2);
}
```

这里有一个global queue被并发使用，它被一个mutex和一个condition variable保护着。mutex确保读写是atomic，而condition variable用来在有新元素可用时激发和唤醒另一个线程。

对 front() 的返回值处理则放在更后面，为的是将lock的持续时间最小化。

并发等待者（线程）的被通知次序是不确定的。

如果有多个 consumer 必须处理相同数据，可以调用 notify_all()。典例是事件驱动系统，其事件必须被发布给所有曾经登录的 consumer。

Condition variable 也提供一个接口允许你等待的最大时间。

18.6.4 细说 Condition Variable

头文件 <condition_variable> 针对 condition variable 提供了两个对应的 class，分别是 condition_variable 和 condition_variable_any。

1）condition_variable

std::condition_variable 用来唤醒一或多个等待某特定条件（某些必须由他人提供或执行的东西）获得满足的线程。一旦条件满足，线程就可以通知（notify）所有（或某个）等待线程。

由于可能发生假醒，当条件满足，仅仅通知是不够的，等待线程还必须在wakeup之后二次检查该条件。

表18.10 列出了标准库为 condition_variable 提供的接口。condition_variable_any 提供相同的接口但不含 native_handle() 和 notify_all_at_thread_exit()。

copy 和 assignment 都是不允许的。

表18.10 conditoin_variable 的操作函数

操作	作用
condvar cv	default 构造函数；创建一个 condition variable
cv.~condvar()	销毁 condition variable
cv.notify_one()	唤醒一个等待线程
cv.notify_all()	唤醒所有等待线程
cv.wait(ul)	使用 unique lock ul 来等待通知
cv.wait(ul, pred)	使用 unique lock ul 来等待通知，直到 pred 在一次wakeup之后结果为true
cv.wait_for(ul, duration)	使用 unique lock ul 来等待通知，等待期限是 duration
cv.wait_for(ul, dur, pred)	使用 unique lock ul 来等待通知，等待期限是 dur，或直到 pre 在一次 wakeup 后结果为 true
cv.wait_until(ul, tp)	使用 unique lock ul 来等待通知，直到时间点 tp
cv.wait_until(ul, tp, pred)	使用 unique lock ul 来等待通知，直到时间点 tp，或直到 pred 在一次 wakeup 之后结果为 true
cv.native_handle()	返回一个因平台而异的类型 native_handle_type，为了不具可移植性的扩展
notify_all_at_thread_exit(cv, ul)	在 calling trhead 结束时，使用 unique lock ul 来唤醒所有 cv 的等待线程。

所有 notification 都会被自动同步化，所以并发调用 notify_one() 和 notify_all() 不会有问题。

全局函数 notify_all_at_thread_exit(cv, l) 用来在其调用者线程 exit 时调用 notify_all()。这个调用只用做清理。

2）condition_variable_any

condition_variable_any 不要求使用 unique_lock 对象当作 lock。如果使用标准mutex类型的一个 unique_lock wrapper并搭配 condition_variable_any，那么使用者必须确保实现 condition_variable_any 实例对象关联的predicate的任何必要同步化。

18.7 Atomic

在 condition variable 的第一个例子中（18.6.1节）我们使用 bool readyFlag 让任意线程激活，表示某件事情已为另一线程备好。为什么仍然需要 mutex。如果有了 bool 值，为什么不能并发地让某线程改变它而让另一线程检验它？

如18.4节介绍，这里需要面对两个问题：

1. 一般来说，即使是基本数据类型，读和写也不是atomic（不可切割的）。因此你可能读到一个被写一半的bool值，undefined 行为。
2. 编译器生成的代码有可能改变操作次序，生产者线程有可能在生产数据之前就先设置 readyFlag，而消费者线程有可能在检测 readyFlag 之前就处理该数据。

借助 mutex，两个问题迎刃而解，但是从必要的资源和潜在的独占访问来看，mutex 也许是个相对昂贵的操作，所以值得用 atomic 替代 mutex和lock。

本节先介绍atomic的高层接口，它提供的操作默认保证提供顺序一致性，意思是在线程中atomic操作保证一定 “像代码出现的次序” 那样发生。重排语句不会出现。最后介绍atomic的底层接口：带有宽松的次序保证的操作。

某些时候atomic低层接口被称为weak或relaxed接口，高层接口称为normal或strong接口。

18.7.1 Atomic 用例

使用 atomic 改写18.6.1节的例子：

```
#include <atomic>  //for atomic types
...
std::atomic<bool> readyFlag(false);

void thread1()
{
    //do something thread2 needs as preparation
    ...
    readyFlag.store(true);
}

void thread2()
```

```
{
    //wait until readyFlag is true
    while(!readyFlag.load()) {
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }

    //do whatever shall happen after thread1 has prepared things
    ...
}

```

需要将 atomic object 初始化，因为其默认构造函数并不完全初始化它（不是初值不明确，而是其lock未被初始化）。面对一个static-duration atomic对象，你应该使用一个常量作为初值。

处理atomic的两个最重要的语句：

- store() 赋予一个新值
- load() 取当前值

这些操作都保证是atomic，所以不需要像之前那样需要mutex的保护才能设置 read Flag。

```
{
    std::lock_guard<std::mutex> lg(readyMutex);
    readyFlag = true;
}  //release lock

//而是简单写成这样
readyFlag.store(true);
```

但是，使用condition variable时仍然需要mutex才能保护对condition variable的消费（即使它现在是个atomic object）：

```
//wait until thread1 is ready(readyFlag is true)
{
    std::unique_lock<std::mutex> l(readyMutex);
    readyCondVar.wait(l, []{ return readyFlag.load(); });
}  //release lock
```

对于atomic类型，也可使用赋值、自动转换为整型、递增、递减等操作。

但注意，为了提供atomicity，某些常用行为可能会有轻微差异，例如赋值操作返回的是 assigned value，而不是返回一个 reference 指向接受该值的 atomic。

```cpp
//concurrency/atomics1.cpp
#include <atomic>  //for atomics
#include <future>  //for async() and futures
#include <thread>  //for this_thread
#include <chrono>  //for durations
#include <iostream>

long data;
std::atomic<bool> readyFlag(false);

void provider()
{
    //after reading a character
    std::cout << "<return>" << std::endl;
    std::cin.get();

    //provide some data
    data = 42;

    //and signal readiness
    readyFlag.store(true);
}

void consumer()
{
    //wait for readiness and do something else
    while (!readyFlag.load()){
        std::cout.put('.').flush();
        std::this_thread::sleep_for(std::chrono::milliseconds(500));
    }

    //and process provided data
    std::cout << "\nvalue: " << data << std::endl;
}

int main()
{
    auto p = std::async(std::launch::async, provider);
    auto c = std::async(std::launch::async, consumer);
}
```

store() 会对影响到的内存区域执行一个所谓 release 操作，确保此前所有内存操作，不论是否为 atomic，在 store 生效之前都变成 “可被其他线程看见”。

load() 会对影响所及的内存区执行一个 acquire 操作，确保随后所有内存操作不论是否为 atomic，在 load 之后都变成 “可被其他线程看见”。

atomic 操作默认使用一个特别的内存次序（memory order），名为 memory_order_seq_cst，它代表顺序一致的内存次序。低层 atomic 操作能够放宽这一次序保证。

18.7.2 细说 Atomic 及其高级接口

类模板 std::atomic<> 提供了 atomic 数据类型的通用能力。另外还有针对 bool、所有整数类型和pointer的特化版本：

```
1 template<typename T> struct atomic;  //primary class template
2 template<> struct atomic<bool>;  //explicit speicalizations
3 template<> struct atomic<int>;
4 ...
5 template<typename T> stuct atomic<T*>;  //partial specialization for pointers
```

表18.11列出了atomic的高级操作。如果可能它们将直接映射到相关的CPU命令。triv表示针对std::atomic<bool>及其他普通类型的atomic提供的操作。int type表示针对std::atomic<>使用整数类型提供的操作。ptr type表示针对std::atomic<>使用pointer类型提供的操作。

- 一般来说，这些操作获得的是 copy 而不是 reference
- 默认构造函数不完全将object初始化。默认构造函数之后唯一合法的操作就是调用atomic_init()完成初始化。
- 接受相关类型值的构造函数不是atomic。
- 除了构造函数的所有函数，都被重载为volatile和no-volatile两个版本。

表18.11 Atomic的高层操作

操作	triv	int type	ptr type	效果
atomic a = val	yes	yes	Yes	以val为a的初值（这不是atomic操作）
atomic a; atomic_init(&a, val)	y	y	y	同上（若无后继的atomic_init()，a初始化不完整）
a.is_lock_free()	y	y	y	如果类型内部不使用lock便返回true
a.store(val)	y	y	y	赋值 val（return void）
a.load()	y	y	y	返回数值a的拷贝
a.exchange(val)	y	y	y	赋值val并返回旧值a的拷贝

a.compare_exchange_strong(exp, des)	y	y	y	CAS 操作
a.compare_exchange_weak(exp, des)	y	y	y	Weak CAS 操作
a = val	y	y	y	赋值并返回val的拷贝
a.operator atomic()	y	y	y	返回数值a的拷贝
a.fetch_add(val)		y	y	atomic 的 t+=val（返回新值的拷贝）
a.fetch_sub(val)		y	y	atomic 的 t-=val（返回新值的拷贝）
a += val		y	y	等同 t.fetch_add(val)
a -= val		y	y	等同 t.fetch_sub(val)
++a, a++		y	y	调用 t.fetch_add(1) 并返回 a 或 a+1的拷贝
a.fetch_and(val)		y		atomic 的 a&=val （返回新值的拷贝，下同）
a.fetch_or(val)		y		atomic 的 a|=val
a.fetch_xor(val)		y		atomic 的 a^=val
a &= val		y		等同 a.fetch_and(val)
a |= val		y		等同 a.fetch_or(val)
A ^= val		y		等同 a.fetch_xor(val)

举例，atomic<int>之内声明了以下赋值操作：

```
namespace std {
    //specialization of std::atomic<> for int;
    template<> struct atomic<int> {
    public:
        //ordinary assignment operators are not provided:
        atomic& operator=(const atomic&) = delete;
        atoimc& operator=(const atomic&) = delete;
        //but assignment of an int is provided, which yields the passed argument:
        int operator=(int) volatile noexcept;
        int operator=(int) noexcept;
        ...
```

```
    };
}
```

借助is_lock_free()，你可以检查atomic类型内部是否由于使用lock才成为atomic。如果不是，你的硬件就是拥有对atomic操作的固有支持（在 signal handler 内使用 atomic 的一个必要条件）。

compare_exchange_strong() 和 compare_exchange_weak() 都是所谓 compare-and-swap（CAS）操作。CPU 通常提供这个atomic操作用以比较 “某内存区内容” 和 “某给定值”，并且只在它们相同时才将该内存区内容更新为另一给定的新值。这可保证新值是根据最新信息计算出来的。效果类似以下伪代码：

```
bool compare_exchange_strong(T& expected, T desired)
{
    if(this->load() == expected) {
        this->store(desired);
        return true;
    }
    else {
        expected = this->load();
        return false;
    }
}
```

因此，如果数值就在这一段时间里被另一线程更新，它会返回 false 并以 expected 承载新值。

18.7.3 Atomic 的 C-Style 接口

C 也有 C++ 对应的 atomic，提供相同语义但是不使用诸如 template、reference 和 member function 等 C++ 特性。

例如，可以使用 atomic_bool 取代 atomic<bool>，并替换store() 和 load()，该用global函数，后者接受一个pointer指向对象：

```
std::atomic_bool ab;           //equivalent to: std::atomic<bool> ab
std::atomic_init(&ab, false);
...
std::atomic_store(&ab, true);  //equivalent to: ab.store(true)
...
if(std::atomic_load(&ab)) {    //equivalent to: if(ab.load())
    ...
}
```

C-Style 接口一般只用在需要在C和C++之间保持兼容的代码上。

18.7.4 Atomic 的低层接口

atomic 低层接口意味着使用 atomic 操作时不保证顺序一致性，因此编译器和硬件有可能重排对 atomic 的处理次序。

需要很多专业经验才能知道何时值得在内存重排上花费心力。

1）atomic 低层接口实例