第八章 多线程编程基础

8.1 串行与并行

8.1.1 并行计算

到目前为止，我们所编写的程序，在同一时刻都只有一条语句在执行，这样的程序称之为*串行程序*。事实上，现代处理器的一个重要特性是多任务的，即在用户看来，处理器可以同时执行多个任务。这样可以通过充分利用计算资源（包括若干台机器或者是一台机器上的若干个 CPU 核），将一个大任务划分成子任务给让各个计算资源同时计算的计算模式称为*并行计算*。

与串行相比，并行计算的好处在于通过利用计算资源，提高任务的执行效率。在日常生活中，并行的例子也是随处可见的。比如，在食堂中有多个打饭的窗口、汽车装配车间中的多条流水线、超市结账的多个收银台，它们都可以看做生活中并行的例子。

在计算机中，并行的例子也是很多的。下面，通过介绍动画的渲染，来帮助理解计算机中是如何实现并行的。

8.1.2 引例：动画渲染

计算机中的动画，通常是以“帧”为单位进行渲染的，每一帧可以理解为是一个静止的画面。在渲染过程中，包括对于图形中各个顶点的计算、几何形状的绘制、着色、加光照等等步骤。现在，将整个动画的渲染表示成以下伪代码形式：

```
Frame frames[N];                // 一共有 N 帧画面

renderFilm():
  for i←0 to N-1:
    frames[i].render()          // 对每一帧进行渲染
```

其中，`renderFilm()`方法是对一个有 N 帧画面的动画进行逐帧渲染。

动画电影的渲染

动画电影中，由于上述介绍的各个步骤中（各个顶点的计算、几何形状的绘制、着色、加光照等）每一个步骤都是计算密集的复杂浮点计算，故绘制每一帧是非常耗时的。以迪士尼的动画电影《汽车总动员 2》为例，平均渲染一帧需要 11.5 小时（最长需要 90 小时），一共约有 152 640 帧[1]。如果是串行进行渲染，那么将会需要 1 755 360 小时（约 200 年）的时间才能完成。所以，动画电影的渲染必须使用并行计算进行加速，否则，恐怕一部动画电影需要 200 年才能和观众见面喽。

```
1  class Frame {
2  public:
3      Frame() { flag = false; }
4      void render() { flag = true; }
5      bool isRendered() { return flag; }
6
7  private:
8      bool flag;
9  };
10
11 const int N = 512;
12 Frame frames[N];
13
14 bool check() {
15     for (int i = 0; i < N; i++) {
16         if (!frames[i].isRendered()) return false;
17     }
18     return true;
19 }
```

代码 8.1. Frame 类、数组及其辅助函数

```
1  #include <iostream>
2  using namespace std;
3
4  void renderFilm() {
5      for (int i = 0; i < N; i++)
6          frames[i].render();
7  }
8
9  int main() {
10     renderFilm();
11     if (check())
12         cout << "动画渲染成功\n";
13     else
14         cout << "动画渲染失败\n";
15     return 0;
16 }
```

代码 8.2. 串行的动画渲染

代码 8.1 给出了动画渲染程序所用到的一些基本类和辅助函数。其中，1-9 行定义了 `Frame` 这个类，为了简化问题，我们仅仅用一个布尔类型的成员变量 `flag` 表示该帧有没有被渲染。11-12 行创建了一个 `Frame` 的全局数组 `frames`。14-19 行的 `check()` 函数用来检测整个动画有没有被完全渲染成功。

代码 8.2 给出了根据伪代码转换而成的串行的动画渲染程序。

通常，相邻的两帧之间是有依赖关系的，它们渲染的先后关系不能颠倒，也不能够完全同时渲染。但是，我们可以把一个动画中的所有帧通过一些方式分成若干组，每一组之间都是可以同时渲染的，而组内保证依次渲染。假设每一台机器渲染其中的一组，那么将整个动画渲染过程表示成以下伪代码：（为了叙述方便，以下假设各个组是等分的）

```
Frame frames[N];                    // 一共有 N 帧画面

renderFilm():
  如果角色是 master:
    将 N 帧画面等分为 M 组
    分配 M 个 slave 机器，每个机器渲染 N/M 帧
    等待所有 slave 机器渲染完毕

  如果角色是 slave:
    获取机器的 id（id 假设从 0 开始到 M-1）
    start ← id * (N/M)
    end ← (id+1) * (N/M)          // 计算每一台机器渲染的范围[start, end)
    for i←start to end-1:
      frames[i].render()          // 对每一帧进行渲染
    通知 master 执行完成
```

上述伪代码中的“角色”，既可以指一台机器，也可以指一台机器内部的“计算资源”（如 CPU 的一个核）。该伪代码中使用的并行计算模式是常用的主从模式，master（主人）会将任务切分为小任务发布给各个 slave(从者)，之后 master 便会等待所有 slave 执行完成。当 slave 执行完成之后，也会通过一些方式通知 master。当 master 等到所有 slave 的反馈之后，任务完成。

计算机进行并行处理的方法有很多种，常见的是*线程并行*，这也是在 C++中所支持的一种编程模式。线程可以理解为在一个程序执行过程中，将一段程序分成不同子模块，同时安排一些“工作者”处理一些任务，这些“工作者”可以称之为一个线程，线程之间的运行时同时的。

并发与并行

并发与并行是相类似的概念，但是它们之间又有一些细节上的不同。[2]

从程序员的角度而言，可以认为串行是指两个执行流（一个执行流是指一组指令的执行序列）之间是一个做完，再做另一个的，它们的生命周期没有交集。而并发是指两个执行流，它们的生命周期是有交集的。

在多核计算机下，可以想象，当不同的进程或者线程在不同的核上同时执行时，是可以达到并发的效果的，这种并发也可称作是并行。但是，事实上，就算是单核上，也是能够实现并发的。这里不得不稍微阐述一下 CPU 的工作模式了。当一个 CPU 有多个任务时，不会一个任务完成后再去完成另一个，而是会把每一个任务分成若干个片段，在经过一定时间片后，切换至另一个任务片段继续完成。由于 CPU 的时间片很短（几个毫秒），所以可以宏观地认为这些任务是在同时执行的。这两种不同类型的并发如图 8.1 所示，其中图 8.1(b)中的虚线表示线程间的切换。

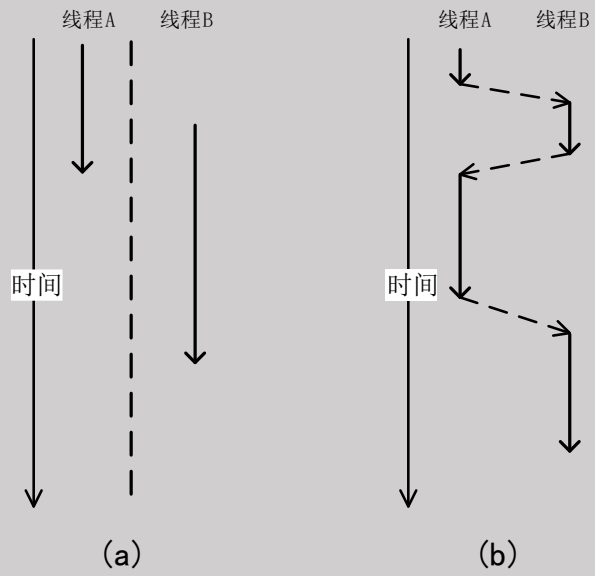

图 8.1. 线程的两种并发模式

(a) 多个核上的并发（并行）；(b) 单核上的并发

由于本书在此不涉及操作系统的细节，所以在下文中，“并发”、“并行”、“多线程”姑且认为是同一个概念，都是指一个程序同时可以由多个“工作者”执行。同样地，“串行”，则是和“单线程”是同义的。

需要注意的是，我们之前写的串行程序中，也是存在一个主线程的，所以每一个程序在运行过程中至少会有一个线程。而由主线程创建的线程，称之为对等线程。

8.2 C++的多线程编程

8.2.1 一个简单的多线程程序

在 C++11 标准中，引入了一系列多线程的标准库，增强了代码的可移植性。下文会对常用的标准库进行介绍。

C++11 之前对于多线程编程的支持

在 C++11 标准出台之前，C++在各个平台上是没有统一的多线程标准库的，所使用创建多线程的方法也是沿用 C 语言中的库。C 语言中多线程的 API 与操作系统相关，比如 Unix/Linux 下使用 pthread 的一系列函数，而在 Windows 也有自己的 WinAPI，造成接口不统一的现象。所以，C++11 标准统一了各个平台上的多线程编程的接口，从语言层面包装了底层多线程的接口，为程序员隐藏了下层操作系统的细节，这样大大增强了代码的可移植性。

代码 8.3 给出了用 C++11 标准，创建一个简单的多线程程序。

```
1  #include <iostream>
```

```
2  #include <thread>        // C++11的多线程标准库
3
4  void foo() { std::cout << "thread 1 ...\n"; }
5  void bar(int x) { std::cout << "thread 2 ... " << x << '\n'; }
6
7  int main() {
8    std::thread first(foo);     // 创建一个线程first并调用foo()
9    std::thread second(bar,0);  // 创建一个线程second并调用bar(0)
10
11   std::cout << "thread main ...\n";
12
13   // 线程回收
14   first.join();               // 等待first线程运行完
15   second.join();              // 等待second线程运行完
16
17   std::cout << "foo and bar completed.\n";
18   return 0;
19 }
```

代码 8.3. 简单的多线程程序

在代码 8.3 的 8-9 行可以看到，C++11 中线程的创建需要 include 名为<thread>的头文件，通过 std 名字空间下 thread 类进行创建的。上述程序的功能是创建两个新的线程，并且在这两个线程和主线程输出一些语句，最后进行线程的回收。读者可以在深入理解这段程序之前，运行一下这段程序看一下输出结果。一个可能的运行结果如下：

```
thread 2 ... 0
thread 1 ...
thread main ...
foo and bar completed.
```

多次运行会发现，前三行输出的顺序可能是任意的一种排列，这也是并发程序的一个重要特性：非确定性。可以看到，主线程和它创建出来的两个对等线程 first、second 是同时运行的，虽然每个线程内部是顺序执行的，但是线程之间并没有强制规定它们执行的顺序，所以这三句话的顺序是可能是任意的。至于如何规定它们的执行顺序，以及代码 8.3 的 13 行之后如何理解，在下文会有介绍。

8.2.2 C++11 中 thread 库介绍

C++11 的 thread 库中提供了 thread 类供程序员对线程进行操作。首先看一下 thread 的构造器的四种原型[3]：

```
thread() noexcept;                                  // 1. 默认构造器
template <class Fn, class... Args>
```

```
explicit thread (Fn&& fn, Args&&... args);     // 2. 初始化构造器
thread (const thread&) = delete;               // 3. 复制构造器[删除]
thread (thread&& x) noexcept;                  // 4. move构造器
```

其中复制构造器不能使用，move 构造器的使用方法读者可以参考 C++11 的文档，最常用是初始化构造器。初始化构造器的声明使用了模板可变参数，使用起来非常灵活。`fn` 传入的是一个函数指针（函数名），`args` 传入的是该函数的参数（0 个或若干个）。调用该构造器就能创建一个线程并在该线程中运行 `fn` 函数。

自己所在的线程可以用 `std` 名字空间下的 `this_thread` 去引用。

另外，再介绍一些 `thread` 类的常用方法。

```
thread::id thread::get_id()const noexcept;
```

操作系统会给每一个创建的线程安排一个线程号，该方法返回的是线程的线程号。（注意，线程号通常是一个比较大的整数，而非从 0,1,2…开始）

```
void thread::detach();
void thread::join();
bool thread::joinable() const noexcept;
```

这三个方法涉及到线程的状态。通常在一个线程在调用初始化构造器之后未调用 `detach()` 方法时的状态是 ***joinable*** 的，这样的线程在生命周期结束后是需要创建这个线程的线程调用 `join()` 方法进行线程状态的回收。而 `detach()` 方法可以使一个线程的状态变成 ***not joinable***，它在结束后不需要其他线程进行回收。使用 `joinable()` 方法可以检查一个线程的状态，***joinable*** 返回 true，***not joinable*** 返回 false。

所以，在代码 8.3 中，14-15 行就是对于 ***joinable*** 线程的状态回收。需要注意的是，`join` 方法会等到需要回收的线程生命周期结束之后才会返回，否则会一直阻塞。若需要回收的线程已经结束，那么该方法会立即返回。所以，可以发现代码 8.3 中第 17 行的输出一定是在最后打印的。

注意，***joinable*** 的线程一定要主线程进行显示调用 `join()` 方法回收，否则会资源浪费而导致内存泄漏，这样是非常危险的。

8.2.3 综合：多线程的图形渲染程序

现在，使用 C++11 提供的多线程库，可以将 8.1.2 节中并行图形渲染的伪代码转换成 C++代码，一种可能的转换如代码 8.4 所示，使用了 8 个线程进行并行渲染。其中，`main()` 函数是与代码 8.2 一致的。

代码 3-9 是对于 slave 线程的转换，渲染自己所负责范围的帧。代码 11-18 其实是 master 线程的操作，它开创了 M 个线程，并且等待它们结束。注意，13-14 行的循环和 16-17 行的循环是不能合并的，否则各个线程之间并不是并行的，读者可以自己思考。

代码 14 行中使用了 `vector` 的 `emplace_back()` 这一方法，这个方法相当于插入了一个元素，该方法传的参数即元素的构造器传的参数。所以，代码第 14 行的操作相当于 `threads.push_back(thread(slaveRenderFilm, i))`。

```
const int M = 8;  // 线程数

void slaveRenderFilm(int id) {  // 参数是自定义的线程序号
    int start = id * (N / M);
    int end = (id + 1) * (N / M);  // 确定各个线程的负责范围
    for (int i = start; i < end; i++)
        frames[i].render();
    cout << "线程" << id << "完成\n";
}

void renderFilm() {
    vector<thread> threads;
    for (int i = 0; i < M; i++)
        threads.emplace_back(slaveRenderFilm, i); // 传入线程序号

    for (int i = 0; i < M; i++)
        threads[i].join();
}
```

代码 8.4. 并行的动画渲染

这段代码的一个可能的输出结果如下：

```
线程 3 完成
线程 0 完成
线程 2 完成
线程 9 完成
线程 7 完成
线程 6 完成
线程 1 完成
线程 4 完成
动画渲染成功
```

注意到各个线程的结束时间和创建时间并不对应，再一次体现了多线程执行的不确定性。

8.3 线程同步

8.3.1 数据竞争

前面只是介绍了一些简单多线程程序的例子，但事实上，多线程程序的设计是非常复杂的，它存在一些特殊的情况需要程序员格外小心。如果忽视这些多线程编程中特有的问题，完全按照串行程序编写的方式进行设计，那么会影响程序的正确性，带来不可预计的后果。其中，*数据竞争*是一个多线程编程中典型的问题，代码 8.5 给出了一个简单的多线程计数程

序，但是在这段代码中就隐含着数据竞争问题，会影响程序的结果。

```
1  #include <iostream>
2  #include <thread>
3  using namespace std;  // 使用了std名字空间
4
5  int countNum = 0; // 全局变量
6
7  // 处理100000次
8  void counter() {
9      for(int i = 0; i < 100000; i++)
10         countNum++;
11 }
12
13 int main() {
14     // 创建两个线程，各自调用counter进行一些处理
15     thread t1(counter), t2(counter);
16
17     // 回收
18     t1.join();
19     t2.join();
20
21     cout << "count: " << countNum << endl;
22     return 0;
23 }
```

代码 8.5. 多线程计数程序（共享全局变量）

代码 8.5 创建了两个线程(第 15 行)，每一个线程将会各自迭代 100000 次并且将每一次将计数器加一（9-10 行）。显然，这段程序的期望结果是输出：

```
count: 200000
```

然而，读者可以尝试运行多次此程序，发现结果并不是确定的，而是可能出现各种结果，比如：

```
count: 128948
count: 161194
count: 115334
count: 107798
```

这样的由于多个线程修改某类变量而导致结果出错的现象在多线程程序内是非常常见的，被称之为“*数据竞争*”。数据竞争产生的原因是对于共享变量的不正确使用，会造成程序出错。

8.3.2 共享变量

在多线程程序中，有一些类型的变量是共享的，即该变量的实例可以被一个或者多个线程引用。有三种类型的变量是共享的：**全局变量、静态变量以及共享指针**。[2]此外，线程函数内的局部非静态变量是私有的，也即各个线程都维护改变量的一个实例，不能被其他的线程所引用。

对于这些共享变量，如果最多只有一个线程修改，其他各个线程只读，那么是不会存在正确性问题的，但是如果存在多个线程进行修改，那么是会存在正确性问题的。以下将会解释，不同情况下对于共享变量的错误使用所造成的数据竞争现场。

共享变量存在数据竞争的原因

数据竞争产生的原因是由于对于共享变量操作的“非原子性”。非原子性操作是指，这样的操作在执行过程中是可能被其他线程打断的。比如，在代码 8.5 中的对共享变量的非原子操作是第 10 行。当一个线程还没有完成对于数据的操作，另一个线程拿到旧值进行了重复操作，导致结果与预期不符。

除非操作本身是原子的（即不可打断的，C++11 中也提供<atomic>库，该标准库中提供了一系列原子操作），否则对于共享变量（全局变量、静态变量、共享指针）的操作就会产生数据竞争的现象。此外函数内部的非静态局部变量则不会产生数据竞争现象，因为它们是不共享的。

8.3.2.1 全局变量

如代码 8.5 所示，使用全局变量 countNum 所谓计数器，所有的线程都是可以引用到这个变量，而且它们的操作都会反映到这个全局变量。但是，由于多个线程会同时修改这个变量（第 10 行），所以最后的结果与预期不符。

8.3.2.2 静态变量

无论是全局静态变量还是局部静态变量，对于各个线程而言都是共享的。所以，代码 8.6 中，使用一个局部静态变量（第 4 行）也能够实现计数的目的。

```
1  int counts[2];
2
3  void counter(int id) {
4      static int countNum = 0;   // 静态变量
5      for(int i = 0; i < 100000; i++)    // 处理100000次
6          countNum++;
7      counts[id] = countNum;
8  }
9
10 int main() {
11     thread t1(counter, 0), t2(counter, 1);
12     t1.join(); t2.join();
```

```
13
14      int realCount = (counts[0] > counts[1])? counts[0] : counts[1];
15      cout << "count: " << realCount << endl;
16      return 0;
17  }
```

代码 8.6. 多线程计数程序（共享静态变量）

如果程序是正确的话，那么 counts[0]和 counts[1]中有一个为 200000，而另一个值会比 200000 小。所以，这两者中最大的一个值可以表示真正的计数个数（第 14 行）。但是，输出结果却再一次验证了结果的不正确性。一些可能的输出结果如下：

```
count: 146112
count: 165962
count: 144445
count: 164086
.....
```

这是由于，代码 8.6 给出了共享静态变量导致的数据竞争问题，因为在第 6 行中，也会有多个线程对作为共享变量的 countNum 这个局部静态变量进行修改。

8.3.2.3 共享指针

由于指针表示的是变量存储的地址，所以在各个线程之间传递指针也容易造成数据竞争，尤其是对同一个变量的指针进行传递。

```
1   // 传入计数器count的指针
2   void counter(int *cp) {
3       for(int i = 0; i < 100000; i++)
4           (*cp)++;
5   }
6
7   int main() {
8       int countNum = 0;
9       thread t1(counter, &countNum), t2(counter, &countNum);
10      t1.join(); t2.join();
11
12      cout << "count: " << countNum << endl;
13      return 0;
14  }
```

代码 8.7. 多线程计数程序（共享同一个变量的指针）

代码 8.7 中，虽然线程函数 counter()本身接受的参数不是共享的，但是由于它们都是定义在 main()函数中 count 这个变量的地址（第 8-9 行），所以本质上也是对同一个内存空间进行共享，多个线程对这个变量进行操作，所以也会造成数据竞争现象，产生不正确

的结果。

因此，为了保证并发程序的正确性，数据竞争是一种要避免的情况。操作系统提供一系列机制保护数据，不发生数据竞争，这些机制称之为线程同步机制。

8.3.3 互斥锁

根据上文的分析，要解决代码 8.5 中的数据竞争问题，可以从共享变量的独立修改入手。也就是说，如果保证每次最多只有一个线程可以对共享变量进行修改，那么就可以解决数据竞争、保证程序的正确性了。

锁是这样一种能够有效解决数据竞争、保证线程同步的机制。这里介绍一种 C++11 提供的互斥锁。

C++11 中的<mutex>库中提供了不同种类的互斥锁，其中最基本的就是 mutex 类。mutex 类具有 lock()和 unlock()两种方法。被这两个函数所保护的代码段能够保证在同一时间，只有一个线程可以执行。

lock()和 unlock()的伪代码如下所示。

```
// lock()的操作不可被打断
void lock(){
    while(flag) ;
    flag = 1;
}
```

```
void unlock() {
    flag = 0;
}
```

这并不是这两个函数的真正实现，只是在语义上方便读者理解。这两个函数的真正实现需要保证这两个函数本身是原子的，不能被其他操作打断。可以看到，lock()的在语义上只能让一个调用者返回，另一个调用者必须阻塞，直到拿到改锁的调用者调用 unlock()之后，才能从 lock()中返回。

代码 8.8 给出了代码 8.5 的一个改进版本，使用互斥锁 mutex 保证对于全局变量 countNum 访问的互斥性。注意到，代码 8.8 和代码 8.5 的区别，仅仅是使用了 mutex 互斥锁对在 counter()函数的调用进行保护。在代码 8.8 的第 2 行，对互斥锁进行了声明（使用了默认构造器）。代码 8.8 的 counter()函数体中，在函数的开始和返回前，加入了互斥锁的上锁（第 5 行）和解锁（第 8 行）。

根据之前的分析，可以发现这时候对于 counter()函数的调用，只有一个线程能够进入循环，而另一个线程则会阻塞在第 5 行进行等待，直到另一个线程放锁才能够进入循环。这样，就解决了共享变量被两个线程同时修改的问题了。这样，在同一时刻，只有一个线程可以执行代码的第 6-7 行，保证同时最多只有一个线程可以修改共享变量。

```
1  int countNum = 0;   // 全局变量
2  mutex mtx;          // 互斥锁，需要include <mutex>
3
4  void counter() {
5      mtx.lock();
6      for(int i = 0; i < 100000; i++)
7          countNum++;
```

```
8       mtx.unlock();
9   }
```

代码 8.8. 加入 mutex 互斥锁的计数程序

8.3.4 锁的粒度

通过对代码 8.8 的分析，我们可以发现这段程序是能够正确执行出期望的结果的，说明这样加锁的方式在正确性上能够保证。

除了正确性之外，加锁对于程序的性能也有一定影响，因为它变相地把并发程序的一部分变成了串行。可以想象，如果一个并发程序中串行的部分越多，那么这个程序的性能就会越差。换言之，锁保护区域的大小会影响程序的性能，我们将锁保护区域的大小称之为锁的粒度。

代码 8.8 中，mtx 保证了第 6-7 行的循环是串行执行的。但事实上，由于我们只需要保护 countNum 变量即可。所以，可以将锁 mtx 的粒度进行适当的减小。

```
1   void counter() {
2       for(int i = 0; i < 100000; i++) {
3           mtx.lock();
4           countNum++;
5           mtx.unlock();
6       }
7   }
```

代码 8.9. 细粒度 mutex 互斥锁的计数程序

代码8.9是代码8.8的一个优化版本，与代码8.8的区别是在于counter()函数中，mtx加锁的范围并不是整个循环，而是每一次循环中对于全局变量countNum的操作。可以验证，这样的加锁方式能够保证程序的正确性，而且直观上可以看出并发程度要比代码8.8更高。所以，相对而言，我们把代码8.8的加锁方式称之为粗粒度加锁，而代码8.9的加锁方式称之为细粒度加锁。

8.3.5 加锁对于程序性能的影响

在 8.3.3 一节中，通过直观的方法，定性地了解了加锁的粒度不同会导致对程序性能的影响。本节将会通过定性的方式，来看一下加锁对于程序性能的影响。这一节将试图回答两个问题：

1）加锁对于程序性能的影响有多严重；

2）是否所有有数据共享的多线程程序都需要通过加锁来解决数据竞争。

下面，通过多线程的求和程序，来看一下不同实现方式下程序性能的变化。

```
1   #include <iostream>
2   #include <thread>
3   #include <mutex>
```

```
4  #include <vector>
5  using namespace std;
6
7  const int MAXTHREADS = 32;       // 最大线程数
8
9  void sum_mutex(int id);
10
11 long gsum = 0;           // 总和
12 long nelems_per_thread; // 每个线程处理的元素个数
13 mutex mtx;
14
15 int main(int argc, char *argv[]) {
16     long nelems(0);
17     int nthreads(0), log_nelems(0);
18     vector<thread> threads;
19
20     if(argc != 3) {
21         cout << "Usage: " << argv[0] << " <nthreads> <log_nelems>\n";
22         return -1;
23     }
24
25     nthreads = atoi(argv[1]);
26     log_nelems = atoi(argv[2]);
27     nelems = 1L << log_nelems;
28     nelems_per_thread = nelems / nthreads;
29
30     for(int i = 0; i < nthreads; i++)
31         threads.emplace_back(sum_mutex, i);
32
33     for(int i = 0; i < nthreads; i++)
34         threads[i].join();
35
36     if(gsum != (nelems * (nelems - 1)) / 2)
37         cerr << "Error result " << gsum << endl;
38
39     return 0;
40 }
```

代码 8.10. 多线程求和主程序

代码 8.10 是一个多线程求和的主程序，它的目的是逐个计算$0+1+2+\cdots+(N-1)$共N项的总和。从它从命令行接受 2 个参数，第一个参数 nthreads 表示使用线程个数，第二个参数 log_nelems 表示总项数$N=2^{\text{log_nelems}}$。

```
1  void sum_mutex(int id) {
```

```
2       long start = id * nelems_per_thread;
3       long end = start + nelems_per_thread;
4
5       for(long i = start; i < end; i++) {
6           mtx.lock();
7           gsum += i;
8           mtx.unlock();
9       }
10  }
```

代码 8.11. 加锁求和的 sum_mutex 函数

代码 8.11 给出了线程函数 sum_mutex 的具体实现，使用互斥锁保护一个全局的求和变量。和代码 8.4 中的并行动画渲染的程序一样，每一个线程都有自己所负责的一定范围。

我们在一台 8 核的机器上，选定$N = 2^{31}$，来测试程序的执行时间随着线程数上升的变化。测试结果如表 8.1 所示。

线程数	1	2	4	8	16	32	64
时间（s）	48.576	176.021	215.823	253.385	255.358	259.631	262.065

表 8.1. 使用 sum_mutex 计数，运行时间随线程数的变化

令人惊讶的是，测试结果表面，**在总工作情况一定的情况下，使用多线程并不会加快程序的执行性能，反而有一个大幅度的性能下降。**造成这种现象的原因是，相比于加法操作，互斥锁的加锁操作更加耗时，随着线程数上升之后，相比单线程多了锁与锁之间的冲突，所以造成了程序性能的极度下降。所以，对于并行程序而言，如果只通过加锁来解决数据竞争，虽然程序的正确性可以保证，但是程序的性能却往往还不如单线程程序。并行程序需要根据自己本身的业务特性仔细设计。

代码 8.11 中，最耗时的操作是加锁。能否通过无锁的方式解决数据竞争？答案是可以的。观察这段程序，可以发现每一个线程并不需要每一次都用 gsum 这个全局变量，而是可以各自加完自己所负责的区域之后，进行汇总，得到答案。那么，这样的设计就可以避免锁的使用。

```
1   long psum[MAXTHREADS]  = { 0 };
2   void sum_local(int id) {
3       long sum = 0;
4       long start = id * nelems_per_thread;
5       long end = start + nelems_per_thread;
6
7       for(long i = start; i < end; i++)
8           sum += i;
9
10      psum[id] = sum;
11  }
```

代码 8.12. 无锁的多线程加法

代码 8.12 给出了一个无锁版本的加法。它的方法是，每一个线程将循环内的加法加到一个局部变量，等所有加法都完成之后，将这个局部变量的值赋给一个全局数组中对应的位置，在 main()函数中，等所有线程完成之后，只要对这个数组求和即可。

可以发现，在这个过程中，各个线程使用的变量都是独立的（局部变量不共享、数组也是访问不同的位置），所以这样就通过重新设计并行程序来解决数据竞争的问题。同样，在同一台 8 核的机器上，仍旧使用$N = 2^{31}$，进行线程数上升之后运行时间的测试。测试结果如表 8.2 所示。

线程数	1	2	4	8	16	32	64
时间（s）	5.160	2.636	1.445	0.846	0.849	0.851	0.853

表 8.2. 使用 sum_local 计数，运行时间随线程数的变化

可以发现，在单线程的情况下，性能有一个数量级的提升，这是由于加锁操作已经减去了。随着线程数上升，程序性能也逐渐上升。不过，同时也看到，使用 X 个线程并不能给程序带来 X 倍的性能提升，在线程数达到 8（即核数）时候，性能最佳，之后再加线程性能变化不大，这是由于计算资源不足、各个之间线程的调度开销等更为底层的原因造成的。

8.3.6 C++11 中的 lock_guard 锁

除了上述介绍的 mutex 互斥锁之外，C++11 标准中有一个基于 mutex 锁的包装——lock_guard 锁，从方便编程和良好代码风格的角度，这个包装过的 lock_guard 类更为常用。lock_guard 类同样声明在标准库中的<mutex>头文件中，它的构造器原型如下：

```
    explicit lock_guard (mutex_type& m);
```

在构造器中，传入一个 mutex 互斥锁的引用，自构造完成之后，互斥锁的便上了锁，直到 lock_guard 类生命周期结束（退出作用域）调用析构器时，传入的互斥锁便解锁。这样，代码 8.9 中的 counter()函数可以改写如代码 8.13 给出的形式。

```
1  void counter() {
2      for(int i = 0; i < 100000; i++) {
3          // lck的生命周期在这个作用域内
4          lock_guard<mutex> lck (mtx);
5          countNum++;
6      }
7  }
```

代码 8.13. 使用 lock_guard 锁

代码 8.13 中需要注意的是第 3-5 行的作用域，保证了 lck 这把锁在这个作用域内进行上锁。准确的说，是从第 4 行构造完成之后开始上锁，直到第 6 行退出改作用域之后进行解锁。运行这段改写后的代码可以发现，其结果和正确性保持不变。

为什么推荐使用 lock_guard 锁，而不是直接使用 mutex 互斥锁，是由于：第一，方便

程序员编程，通过作用域明确加锁和去锁的范围；第二，通过作用域增强代码可维护性，如在 try-catch 异常处理中，不需要同一把锁解锁若干次，而是通过作用域，把解锁的任务交给 lock_guard 的析构器即可。

8.4 基于锁的并发数据结构

并发数据结构，是指一个数据结构能够有多个线程可以对它进行操作，可以同时处理多个请求。同样，需要从正确性和性能两方面考虑进行设计。下文将会介绍三个主要的基于锁并发数据结构：并发链表、并发散列表以及并发队列。[4]在每一种数据结构中，我们只会介绍该数据结构的一些核心操作，其他操作请读者自行思考。

8.4.1 并发链表

链表是常用的数据结构，核心的操作是插入和查找。代码 8.14 给出了一种并发链表的实现代码。代码 31-36 行定义了链表中每一个节点的成员，包括一个 key（该实现中使用 int 类型）和指向下一个节点的指针。整个链表有一个 head 指正表示链表的头部，每次插入是从头开始插入的。

5-20 行定义了插入操作 insert()。插入操作的并行性体现在对于 head 的修改，需要新建一个节点，然后把 head 变为这个节点。为了数据结构的效率，尽量把不存在数据竞争的代码放在锁之外。因此，7-8 行都没有使用到 head 指针，因此不存在数据竞争。注意，new 操作符是可以多线程使用的，但是在一定情况下会存在异常，故使用 try-catch 进行异常处理。10-12 行对 head 指针进行操作，因此需要用互斥锁来保护。

22-28 行定义了查找操作 lookup()。查找操作的实现直接用一把粗粒度的锁保护整个函数。这样粗粒度的锁是可以避免的。但是，如果这个数据结构支持修改和删除，那么这个粗粒度的锁是必要的。

```
1  class List {
2  public:
3      List() { head = NULL; }
4
5      bool insert(int key) {
6        try {
7          Node *newHead = new Node;
8          newHead->key = key;
9          {
10             lock_guard<mutex> lck(mtx);
11             newHead->next = head;
12             head = newHead;
13         }
14         return true;
15       }
16       catch (bad_alloc &e) {
```

```
17          cerr << "bad_alloc caught: " << e.what() << endl;
18          return false;
19        }
20      }
21
22      bool lookup(int key) {
23          lock_guard<mutex> lck(mtx);
24          for (Node *curr = head; curr; curr = curr->next) {
25              if (curr->key == key) return true;
26          }
27          return false;
28      }
29
30  private:
31      struct Node {
32          int key;
33          Node *next;
34      };
35      Node *head;
36      mutex mtx;
37  };
```

代码 8.14. 并发链表

8.4.2 并发散列表

在并发散列表中，同样实现插入和查找操作。基于 8.4.1 节中设计的并发链表，对于并发散列表的设计就非常简单了。

在代码 8.15 给出的实现中，每一个散列桶使用的是一个 `List` 数据结构，这样设计的好处是在于，每一个散列桶各自使用自己的锁，而不同桶之间本身就不存在数据竞争，所以不需要使用一把全局的大锁。

```
1   const int BUCKET = 101;
2
3   class Hash {
4   public:
5       bool insert(int key) {
6           int bucket = key % BUCKET;
7           return lists[bucket].insert(key);
8       }
9
10      bool lookup(int key) {
11          int bucket = key % BUCKET;
12          return lists[bucket].lookup(key);
```

```
13      }
14  private:
15      List lists[BUCKET];
16  };
```

代码 8.15. 并发散列表

我们使用并发链表和并发散列表分别进行测试，看一下这两种数据结构的性能比较。同样在一台 8 核的机器上，一共插入10^8个数据（0,1,2，…，10^8-1），每一个线程平均分配，它们执行时间与线程数的关系如图 8.1 所示。可见，由于并发链表使用一把全局的锁锁住插入操作，所以性能不好，这也和表 8.1 中的现象是一致的，原因也在 8.3.5 节有解释。由于并发散列表的并行程度更高，每一个散列桶有自己的锁，故随着线程数上升，能够获得更好的性能。

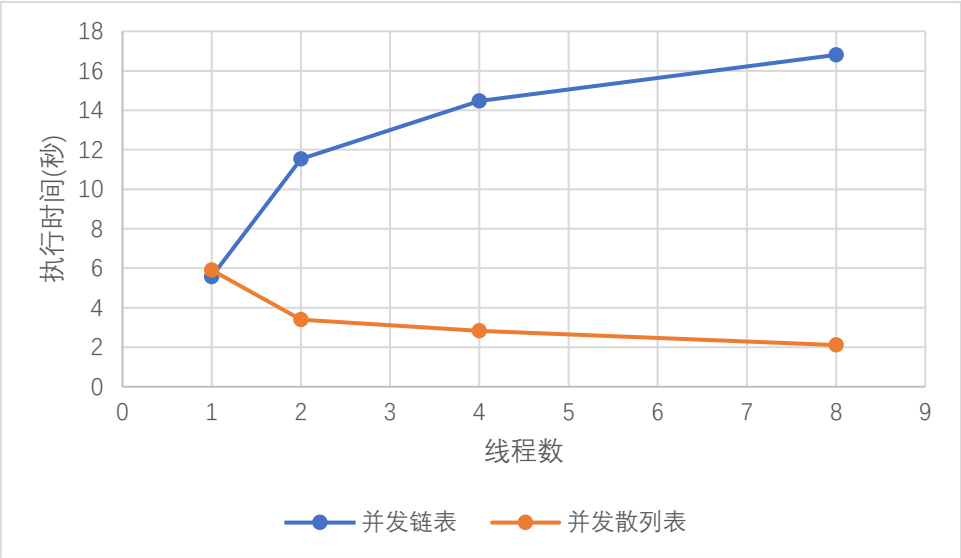

图 8.1. 并发链表与并发散列表的性能比较

8.4.3 并发队列

队列支持的两种操作是入队和出队。队列有头部 head 和尾部 tail 之分，入队会把节点插入到尾部，出队会返回头部的节点并且从队列中删除改节点。代码 8.16 给出了一种并发队列的实现，这种实现是由 Michael 和 Scott 两人在 1996 年设计出来的，其精要之处在于设计了一个 dummy 节点。目的是使对于 tail 和 head 之间的处理不会出现竞争，所以出队和入队之间不会冲突。试想一下，如果第 9 行初始化 tail 和 head 都指向 NULL，那么插入第一个节点和删除最后一个节点，需要做特殊的判断，而且同时会影响 tail 和 head 两个指针。

加入 dummy 节点，这样 tail 指向队列中最后的节点（如果队列为空，则指向 dummy 节点），而 head 指向的并不是第一个真正的节点，而是 dummy 节点，dummy 节点的下一个节点才是队列中的第一个节点。该数据结构如图 8.2 所示。

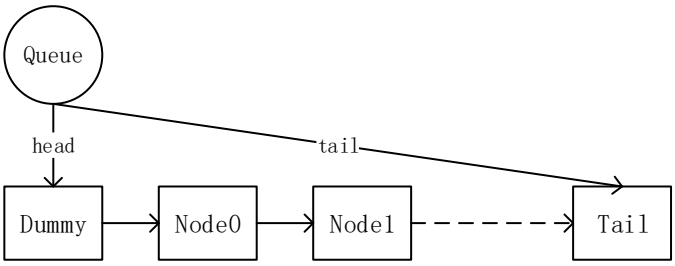

图 8.2. Michael & Scott Queue

有了 dummy 节点之后，加锁只需要在 tail 和 head 上单独加锁，而不需要一个全局的锁同时管理出队和入队，这样会在保证正确性的情况下提高数据结构的效率。

```
1  class Queue {
2  public:
3      Queue() {
4          Node *dummy = new Node{ 0, NULL };
5          head = tail = dummy;
6      }
7
8      void enqueue(int key) {
9          Node *tmp = new Node{ key, NULL };
10         lock_guard<mutex> lock(tailMtx);
11         tail->next = tmp;
12         tail = tmp;
13     }
14
15     bool dequeue(int *value) {
16         lock_guard<mutex> lock(headMtx);
17         Node *tmp = head->next;
18         if (tmp == NULL) return false;
19
20         *value = tmp->key;
21         delete head;
22         head = tmp;
23         return true;
24     }
25
26 private:
27     struct Node {
28         int key;
29         Node *next;
30     };
31
32     Node *head, *tail;
```

```
33      mutex headMtx, tailMtx;
34  };
```

代码 8.16. 并发队列

这样设计的并发队列的好处在于，入队和出队之间是不会出现竞争的，这两个操作可以完全并行地执行。我们在一台 8 核的机器上，使用并发队列分别完成10^6次入队和10^6次出队，分两种执行模式，串行和并行（一个线程负责出队，另一个线程负责入队）。为了让测试更加符合实际情况，在出队前和入队后加入了一些简单的计算。测试结果如表 8.3 所示。

执行模式	串行	并行
执行时间（秒）	3.737	2.028

表 8.3. 并发队列的测试结果

可见，使用这样的并发队列，有利于在出队和入队次数差不多的情况下，程序执行的性能。但是，如果只是大量使用单一的操作，由于出队与出队之间、入队与入队之间是需要锁串行的，所以这样的时候性能就不理想，退化成并发链表。所以，并发数据结构的设计也要考虑到应用场景。

8.5 死锁

以上介绍了通过加锁来解决数据竞争的问题，但是加锁除了会带来性能伤害之外，也并不能解决所有类似的数据同步问题，“死锁”就是加锁可能会出现的问题之一。下面，就简单介绍一下“死锁”现象，而如何解决死锁问题，则超出了本书的范围。

```
1   #include <iostream>
2   #include <mutex>
3   #include <thread>
4   using namespace std;
5
6   mutex mtx1;         // 互斥锁1
7   mutex mtx2;         // 互斥锁2
8
9   void foo() {
10      mtx1.lock();
11      mtx2.lock();
12      cout << "foo" << endl;
13      mtx1.unlock();
14      mtx2.unlock();
15  }
16
17  void bar() {
18      mtx2.lock();
```

```
19      mtx1.lock();
20      cout << "bar" << endl;
21      mtx2.unlock();
22      mtx1.unlock();
23  }
24
25  int main() {
26      // 两个线程分别调用foo和bar并回收
27      thread t1(foo), t2(bar);
28      t1.join();
29      t2.join();
30
31      cout << "Finish!" << endl;
32      return 0;
33  }
```

代码 8.17. 一个会产生“死锁”程序

代码 8.17 是一个会产生“死锁”现象的简单例子。主线程会新建两个线程 t1 和 t2，这两个线程会分别调用 foo()和 bar()两个函数。这两个函数的内部基本一致，都是先拿两个锁，在进行一些输出之后放锁。主要的不同是在于，foo()函数是先拿 mtx1 锁，再拿 mtx2 锁。而 bar()函数恰恰相反，是先拿 mtx2 锁，再拿 mtx1 锁。

读者不妨自己编译一下代码 8.17 然后执行这段程序，在重复若干次的执行中，可能的结果如下所示：

```
foo
bar
Finish!
```

```
bar
foo
Finish!
```

```
(没有显示，程序阻塞)
```

前两种结果是符合我们预期的，它们不同的顺序是是由于线程之间执行顺序的不确定性造成的。第三种结果中，运行时发现这段程序不会结束，而是一直处于一种阻塞的状态。这种由不恰当的加锁方式导致程序阻塞不能执行下去的现象，就是常见的“死锁”现象。

那么，就这段代码 8.17 的程序而言，为什么会出现“死锁”现象，值得好好分析一下。之前曾经提到过，并发编程中，若干个线程在执行过程中有一个重要的特性：非确定性，也即在没有同步机制（如加锁）下，两个线程代码的执行顺序是不确定的。所以，在代码 8.17 中，线程 t1 和 t2 的拿锁顺序是不一定的。如果在一种极端的情况下，这两个线程的调度是串行的，那么不会出现死锁问题。但是，如果考虑 t1 在执行完第 10 行拿到了 mtx1 锁，之后 t2 执行了第 18 行拿到了 mtx2 锁，那么这时候 t1 会阻塞在第 11 行，而 t2 会阻塞在第 19 行，可以发现，这样导致的一个结果是所有线程都无法进行下去，因为如果有一个线程要进

行下去，必须要等到对方放锁，尝试一个互相等待的状态。

这个例子告诉我们，如果加锁方式不当（尤其是有多个锁的情况下），即使程序的正确性有所保障，但是仍然会因为线程的调度使整个程序阻塞，进入死锁状态。

至于如何检测死锁、如何解除死锁，则有一套相对成熟的理论可以解决，但是已经超出本书所讲述的范围之外了。

8.6 本章小结

本章介绍了一种能够加快程序性能的方法：多线程编程。C++11 为多线程编程统一了标准库，使用<thread>库增强了并行程序的代码可移植性。多线程编程是困难的，它具有和串行编程所不同的编程模式，其中对于共享变量的数据竞争问题是一个严重的问题。由此，需要引入数据同步机制，保证对于数据的访问不会产生竞争，而锁是一种常见的数据同步机制。然而，锁并不是万能的，它也会有自己的问题，比如使用不当而带来的死锁问题等。最后，本章介绍了一些常见的基于锁的并发数据结构，并且对它们的实际性能进行了分析。

8.7 练习题

1. 代码 8.18 给出了一个有缺陷的程序。这个程序的目的是，创建一个线程，要求线程休 1 秒后输出一段字符串。然而，在实际在 linux 上运行的时候这段程序并不能正常输出字符串，甚至可能会出现异常（windows 上也有类似的情况，只需要把 `sleep` 函数引入相应的库 `<Windows.h>`即可，注意 windows 下该函数的参数单位是毫秒）。但是，把 13 行的注释解开，就能够正常实现功能。请尝试解释此现象。

```
1  #include <thread>
2  #include <iostream>
3  #include <unistd.h>    // sleep(seconds) under linux
4  using namespace std;
5
6  void printer() {
7    sleep(1);
8    cout << "Hello World!" << endl;
9  }
10
11 int main() {
12   thread t(printer);
13   // t.join();
14   return 0;
15 }
```

代码 8.18. 一个会产生异常的程序

2. 试分析下面这个程序的加锁序列是否会导致死锁。

线程 1	线程 2
a.lock()	c.lock()
b.lock()	c.unlock()
b.unlock()	b.lock()
c.lock()	a.lock()
c.unlock()	a.unlock()
a.unlock()	b.unlock()

3. std::for_each 是 C++标准库<algorithm>中提供的模板函数，它的一个实现形式如代码 8.19 所示。它的模板参数为一个迭代器类型和一个函数，能在[first, last]范围内所有的元素上都执行函数 fn。
   请设计一种多线程版本 parallel_for_each，要求模板参数再加一个N表示并行的线程数。（假设函数 fn 施加在元素的顺序不会对结果产生影响，即各个元素是独立的）

```
template<class Iterator, class Function>
Function for_each(Iterator first, Iterator last, Function fn) {
  while (first!=last) {
    fn (*first);
    ++first;
  }
  return fn;      // or, since C++11: return move(fn);
}
```

代码 8.19. std::for_each 的实现

4. 归并排序不仅在时间复杂度上是最优的基于排序的算法，而且该排序算法也具有良好的并行性。请实现一个多线程版本的归并排序算法，要求：
   (1) 可以接受一个参数 threadNum 表示最大可以开的线程数；
   (2) 在排序阶段和归并阶段都可以多线程执行。

5. 矩阵乘法是在数值计算、机器学习等领域中运用广泛的一种操作，对性能的影响也很大。增加并行性是一种常见的对于矩阵乘法加速的算法，请实现一种N × N矩阵乘法的多线程版本，要求：使用模板类定义矩阵，模板指定元素的类型和N的大小。

6. 使用多线程实现一个递归版本的斐波那契数列计算，空间复杂度是O(1)。注意，如果每一层递归都开一个线程去完成，线程数会呈几何级数上升，消耗大量内存资源。故，要求可以接受一个参数 threadNum 表示最大可以开的线程数。

7. Floyd–Warshall 算法是一种常用的基于动态规划的最短路径算法，可以计算一个有权图$G(V,E,w)$中任意两点之间的最短距离。其中V是点集，E是边集合，对任意一条边$e \in E$,$w(e)$表示组成这条边的两个顶点的距离。最小距离矩阵 dist 是一个$|V| \times |V|$的二位数组，记录任意一对点之间的最小距离，如果不存在路径则记为∞。(无穷大)

该算法的伪代码如下所示：

```
1   令 dist 每一个元素初始化为∞(无穷大)
2   for each vertex v
3      dist[v][v] ← 0
4   for each edge (u,v)
5      dist[u][v] ← w(u,v)   // the weight of the edge (u,v)
6   for k from 0 to |V| − 1
7      for i from 0 to |V| − 1
8         for j from 0 to |V| − 1
9            if dist[i][j] > dist[i][k] + dist[k][j]
10              dist[i][j] ← dist[i][k] + dist[k][j]
11           end if
```

尝试对 dist 矩阵进行合理划分，实现一个多线程版本的 Floyd–Warshall 算法。

8.8 文献阅读

对于 C++编程的接口，可以参看[http://www.cplusplus.com/reference/]，这是 C++标准的官方参考网站。多线程编程是实现并发编程的一种方式，[Williams A.，2012]和[Lin C ，2008]都是从并发编程的角度来描述 C++的程序设计，里面会描述一些并发编程的通用原则。此外，由于并发编程在许多高级语言中都支持，所以它们的编程模式是可以参考的。如[Eckel B, 陈昊鹏, 等，2002]、[BilLewis, DanielJ.Berg, 刘易斯,等，2000]、[葛一鸣, 郭超.，2015]中都介绍了 Java 对于并发编程的支持，都可以在设计 C++多线程程序的时候进行参考。

［Cormen T H, Leiserson C E, Rivest R L.，2006］这本算法的经典著作，从算法原理的角度描述了常用的并行算法，并且进行了算法的分析。

多线程这个概念是操作系统中为上层提供的重要抽象，［Randal E. Bryant, Davie Richard O'Hallaron.，2016］和［Arpaci-Dusseau R H, Arpaci-Dusseau A C. ，2015］从操作系统的角度，阐述了多线程、锁、数据竞争等概念。多线程编程和并行的概念应用广泛，[https://gizmodo.com/5813587/12500-cpu-cores-were-required-to-render-cars-2]给出了一个实际的例子，阐述了并行的重要性，以及如何加速性能的。