Morden cpp notes — concurrency

Wednesday, December 9, 2020 8:26 AM

并发：一个时间段里有多个操作在同时进行，多线程是实现手段之一。

线程：在C++中，线程 thread 就是一个能够独立运行的函数。

```
auto f = [](
{
  Cout << "tid=" <<
      this_thread::get_id() << endl;
};
thread t(f); //启动一个线程，运行函数f.
```

任何程序一开始就有一个主线程，从main()开始运行，主线程可以调用接口函数，创建出子线程。子线程会立即脱离主线程的控制流程，单独运行，但共享主线程的数据。程序创建出多个子线程，执行多个不同的函数，就是多线程。

多线程的好处：
1. 任务并行
2. 避免I/O阻塞。
3. 充分利用CPU
4. 提高用户界面响应速度。

多线程的缺点：
1. 同步
2. 死锁。
3. 数据竞争
4. 系统调度开销。

读而不写就不会有数据竞争，所以在C++多线程里，用const读取变量最安全，对类调用const成员函数，对容器调用只读算法也总是线程安全的。

在C++中，有4个基本工具：
1. 仅调用一次
2. 线程局部存储。
3. 原子变量
4. 线程对象。

仅调用一次

程序要初始化数据，在多线程中可能导致初始化函数多次运行。为此，C++提供了仅调用一次的功能：声明一个 once_flag 类型的变量，最好是静态，全局的（线程可见），作为初始化的标志。

```
Static std::once_flag flag; //全局的初始化标志.
```

然后调用专门的 call_once() 函数，以函数式编程的方式，传递这个标志和初始化函数。C++会保证即使多个线程重入 call_once()，也只能有一个线程会成功运行初始化。

使用 lambda 来模拟实际的线程函数。

```
auto f = []()
{
   Std::call_once(flag,    仅一次调用，注意传flag.
       []() {       匿名lambda，初始化函数，只会执行一次.
          Cout << "only once" << endl;
       }
   );
};
thread t1(f);   启动两个线程，运行函数f.
thread t2(f);
```

call_once() 完全消除了初始化时的并发冲突，在它的调用位置根本看不到并发和线程，可解决"双重检查锁定"问题。

线程局部存储。

读写全局或局部静态变量，是另一个常见数据竞争，因为共享数据，多线程操作时，可能导致状态不一致。有些变量应该是线程独占所有权（类似switch里的独占游戏），即线程局部存储 thread local storage，由关键字 thread_local 实现，和 static，extern 同级的变量存储说明，有 thread_local 标记的变量在每个线程里都会有一个独立的副本（就像魔兽中每个玩家有一个副本），是线程独占的，不会有竞争读写的问题。

thread_local 示例：
1. 定义线程独占变量。
2. 用 lambda 捕获引用
3. 放进多线程里运行。

```
thread_local int n=0; //线程局部存储变量.
auto f = [&](int x) //在线程里运行lambda，捕获引用
{
  n+=x;     //使用线程局部变量，互不影响.
  Cout << n;   //输出，验证结果
};
thread t1(f, 10); //启动两个线程，运行f.
thread t2(f, 20);
```

两个线程分别输出10，20，互不干扰，如果改为 static int n=0; 就是静态全局变量，因为两个线程共享变量，所以n被连加了两次，输出30。

原子变量。

对于非独占，必须共享的数据，要保证多线程读写共享数据的一致性，关键要解决同步问题。不能让两个线程同时写，也就是互斥。互斥量 Mutex 成本太高，对于小数据，应采用原子化方案。

原子 atomic，指的是不可分的，操作要么完成，要么未完成，不能被任何外部操作打断，总是有一个确定的，完整的状态，也就不会存在竞争读写的问题，不需要互斥量，成本更低。

C++只允许一些最基本的类型原子化，比如 atomic_int，atomic_long：

```
using atomic_bool = std::atomic<bool>;
using atomic_int  = std::atomic<int>;
using atomic_long = std::atomic<long>;
```

原子变量禁用了拷贝构造函数，在初始化时不能用 = 的赋值形式，只能用圆/花括号：

```
atomic_int x{0};
atomic_long y{1000L};
assert(++x ==1);
y += 200;
assert(y < 2000);
```

最基本的用法，是把原子变量当作线程安全的全局计数器/标志位。或是实现高效的无锁数据结构 lock-free，可以考虑使用 boost.lock_free。

线程

call_once()，thread_local，atomic 可以消除显式使用线程，如果必须使用线程，C++提供了 std::this_thread，还有 yield()，get_id()，sleep_for()，sleep_until 等函数。

```
Static atomic_flag flag{false}; //原子化的标志量
Static atomic_int  n; //原子化的int.
auto f = [&]()  //在线程里运行的lambda，捕获引用.
{
  auto value = flag.test_and_set(); // TAS检查原子标志量.
  if (value) {
     Cout << "flag has been set." << endl;
  } else {
     Cout << "set flag by " <<
         this_thread::get_id() << endl;
  }
  n+=100;
  this_thread::sleep_for(   //线程睡眠
     n.load() * 10ms);    //使用时间字面量.
  Cout << n << endl;
};
thread t1(f);   //启动两个线程.
thread t2(f);
t1.join();   //等待线程结束.
t2.join();
```

建议不直接使用 thread，而是调用 async()，含义是异步运行一个任务，隐含的动作是启动一个线程去执行，

```
auto task =[](auto x)  //在线程里运行lambda.
{
  this_thread::sleep_for(x * 1ms); //线程睡眠.
  Cout << "sleep for" << x << endl;
  return x;
};
auto f = Std::async(task, 10); //启动一个异步线程.
f.wait(); //等待任务完成.
assert(f.valid()); //确实已经完成了任务.
Cout << f.get() << endl; // 获取任务执行结果.
```

这还是函数式编程的思路，在更高的抽象级别上看待问题，异步并发多个任务，让底层自动管理线程，

async() 会返回一个 future 变量，可以认为代表了执行结果的"期货"，如果任务有返回值，可以用成员函数 get() 获取。注意 get() 只能调用一次，否则会触发 std::future_error 异常。

即使不关心返回值，也总要用 auto 配合 async()，避免同步阻塞：

```
Std::async(task, ...); // 没有显式获取future，被同步阻塞.
auto f = Std::async(task, 10);
```

Tips:
- C++20 加入了协程，co_wait，co_yield，co_return，是用户态的线程，可以写出开销更低，性能更高的并发程序。
- 如果只是简单的在线程里启动一个异步任务，完全不关心返回值，可以调用 thread 成员函数 detach()，比 async() 方便一点。
- 多线程调试一般是打日志，里面列出线程号和使用变量，