线程之间共享数据的问题

ϒ 使用互斥体保护数据

ϒ 保护共享数据的替代设施

 在单个进程中的多个线程之间轻松共享数据不仅是一个好处，也可能是一个很大的缺点。 共享数据的不正确使用是并发相关错误的最大原因之一

 无论结果如何，这是并发代码中最常见的错误原因之一的示例：竞争条件。

在并发中，竞争条件是指结果取决于两个或多个线程上操作执行的相对顺序的任何情况； 线程竞相执行各自的操作。

当竞争条件导致不变量被破坏时，就会出现问题，例如前面提到的双向链表示例。

**竞争条件的定义**

1. **什么是竞争条件？**
   * 竞争条件是指多个线程或进程同时访问和修改共享数据，导致程序的最终行为依赖于线程的执行顺序。
   * 如果程序的正确性依赖于线程的执行顺序，那么就会发生竞争条件。
2. **竞争条件的典型场景**：
   * 当完成一个操作需要修改两个或多个独立的数据时，竞争条件通常会发生。
   * 例如，在链表中插入一个节点需要修改两个指针（前驱节点和后继节点的指针），如果这两个修改操作不是原子的，可能会导致链表处于不一致的状态。

处理竞争条件的另一种方法是将数据结构的更新作为事务处理，就像在事务中完成数据库的更新一样。 所需的一系列数据修改和读取存储在事务日志中，然后一步提交。 如果由于数据结构已被另一个线程修改而无法继续提交，则事务将重新启动。 这被称为软件事务内存（STM），在撰写本文时它是一个活跃的研究领域。

**软件事务内存（STM）的概念**

1. **事务的基本思想**：
   * 事务是一系列操作的集合，这些操作要么全部成功（提交），要么全部失败（回滚）。
   * 在 STM 中，对共享数据的修改和读取操作被记录在事务日志中，然后在单个步骤中提交。
2. **STM 的工作流程**：
   * **开始事务**：记录对共享数据的所有修改和读取操作。
   * **提交事务**：如果在此期间没有其他线程修改共享数据，则提交事务，否则回滚并重试。
   * **回滚事务**：如果提交失败，撤销所有修改并重新开始事务。
3. **STM 的优点**：
   * 简化了并发编程，程序员不需要手动管理锁或原子操作。
   * 提供了更高的抽象层次，减少了竞争条件的风险。
4. **STM 的缺点**：
   * 性能开销较大，特别是在高并发场景下。
   * 目前 C++ 标准库中没有直接支持 STM（尽管有一个技术规范）

**代码解析**

cpp

复制

std::lock\_guard<std::mutex> guard(some\_mutex);

1. **std::mutex**：
   * some\_mutex 是一个 std::mutex 对象，用于保护共享数据。
2. **std::lock\_guard<std::mutex>**：
   * std::lock\_guard 是一个类模板，需要指定模板参数（这里是 std::mutex）。
   * 它接受一个 std::mutex 对象作为构造参数。
3. **guard**：
   * guard 是 std::lock\_guard<std::mutex> 的一个实例。
   * 在构造 guard 时，some\_mutex 会被自动加锁。
   * 在 guard 析构时（例如离开作用域时），some\_mutex 会被自动解锁。

尽管在某些情况下使用全局变量是合适的，但在大多数情况下，通常将互斥锁和受保护数据分组在一个类中，而不是使用全局变量。 这是面向对象设计规则的标准应用：通过将它们放入一个类中，您可以清楚地将它们标记为相关的，并且可以封装功能并强制执行保护。

使用互斥锁（std::mutex）保护共享数据时，仅仅在成员函数中使用 std::lock\_guard 是不够的。如果成员函数返回了指向受保护数据的指针或引用，或者将这些指针或引用传递给不受控制的函数，那么锁的保护就会失效，从而导致数据竞争和未定义行为。

**从好的方面来说，您可以遵循一个指南，这将在这些情况下为您提供帮助：不要将指针和引用传递到锁范围之外的受保护数据，无论是从函数返回它们，还是将它们存储在外部可见内存中 ，或将它们作为参数传递给用户提供的函数。**

**为了使线程安全地删除节点，您需要确保防止并发访问三个节点：被删除的节点和两侧的节点。 如果您单独保护对每个节点的指针的访问，那么您的情况不会比使用不使用互斥体的代码更好，因为竞争条件仍然可能发生 - 需要保护的不是单个步骤的单个节点，而是整个节点 数据结构，用于整个删除操作。**

**竞争条件的根源**

1. **接口设计问题：**
   * **std::stack 的接口设计使得组合操作（如检查栈是否为空并读取栈顶元素）无法保证线程安全。**
   * **即使每个操作是线程安全的，组合操作仍然可能导致竞争条件。**
2. **锁的范围问题：**
   * **即使使用互斥锁保护每个操作，锁的范围仅限于单个操作，无法保护组合操作。**

**原子操作 是指将多个操作组合成一个不可分割的整体，确保在执行过程中不会被其他线程打断。在这个例子中，try\_pop 函数将 top() 和 pop() 封装成一个操作，并通过互斥锁（std::mutex）确保这个组合操作是线程安全的。**

**传递引用的方式：**

* **调用者创建一个对象，并将其引用传递给 pop() 函数。**
* **pop() 函数将栈顶元素的值复制或移动到传入的引用中，然后弹出栈顶元素。**

**关键点总结：**

1. **优点：**
   * **指针可以自由复制，不会抛出异常。**
   * **使用 std::shared\_ptr 可以避免内存泄漏，并且库可以控制内存分配方案。**
2. **缺点：**
   * **需要管理对象的内存分配。**
   * **对于简单类型（如整数），内存管理的开销可能超过按值返回的成本。**

**细粒度锁（fine-grained locking） 是一种并发编程中的锁设计策略，其核心思想是将锁的粒度细化，使得每个锁只保护一小部分数据或资源，而不是用一个大锁（粗粒度锁）保护整个数据结构或所有资源。通过这种方式，可以提高并发性能，因为多个线程可以同时访问数据结构的不同部分，而不会因为一个全局锁而相互阻塞。**

**一对线程中的每一个都需要锁定一对互斥锁来执行某些操作，并且每个线程都有一个互斥锁并正在等待另一个 。 两个线程都无法继续，因为每个线程都在等待另一个线程释放其互斥体。 这种情况称为死锁，这是必须锁定两个或更多互斥体才能执行操作的最大问题。**

**std::scoped\_lock 的存在意味着在 C++17 之前使用 std::lock 的大多数情况现在都可以使用 std::scoped\_lock 编写，出错的可能性较小，这只能是一个好消息 事物！**

**死锁（Deadlock） 不仅仅是因为锁（如 std::mutex）引起的，它可以在任何线程相互等待的情况下发生。你举的例子是两个线程互相调用 join()，导致彼此等待，从而形成死锁。**

**避免死锁的准则都归结为一个想法：如果有机会另一个线程在等你，就不要等待。**

**第一个想法是最简单的：如果已经持有锁，则不要获取锁。 如果您遵循此准则，则不可能仅因锁的使用而导致死锁，因为每个线程仅持有一个锁。 您仍然可能因其他原因（例如相互等待的线程）而陷入死锁，但互斥锁可能是死锁的最常见原因。 如果您需要获取多个锁，请使用 std::lock 将其作为单个操作来执行，以便在没有死锁的情况下获取它们。**

**避免在持有锁时调用用户提供的代码**

**因为代码是用户提供的，所以你不知道它能做什么； 它可以做任何事情，包括获取锁。 如果您在持有锁的同时调用用户提供的代码，并且该代码获取了锁，则您违反了避免嵌套锁的准则，并且可能会出现死锁。**

**以固定顺序获取锁**

**按固定顺序获取锁的核心思想：**

1. **定义一致的锁获取顺序：**
   * **所有线程必须按照相同的顺序获取锁。**
   * **例如，如果线程 A 先锁定 mutex1 再锁定 mutex2，那么线程 B 也必须按照相同的顺序锁定 mutex1 和 mutex2。**
2. **避免循环等待：**
   * **如果所有线程都按照相同的顺序获取锁，就不会出现循环等待的情况，从而避免死锁。**

**交换操作中的锁顺序**：

* + 在交换两个对象的操作中，如果两个线程以相反的顺序锁定对象的互斥量，可能会导致死锁。

**使用锁层次结构**

**尽管这是定义锁顺序的特殊情况，但锁层次结构可以提供一种方法来检查运行时是否遵守约定。 这个想法是将应用程序划分为多个层，并识别可能锁定在任何给定层中的所有互斥体。 当代码尝试锁定互斥锁时，如果它已经持有来自较低层的锁，则不允许锁定该互斥锁。 您可以在运行时通过为每个互斥体分配层号并记录每个线程锁定的互斥体来检查这一点。**

**死锁也发生在锁上。 它可能发生在任何可能导致等待周期的同步结构中。 因此，值得扩展这些指南以涵盖这些情况。**

**使用 std::unique\_lock 灵活锁定**

**主要特性**

1. **自动管理锁的生命周期：与 std::lock\_guard 类似，std::unique\_lock 在析构时会自动释放锁。**
2. **延迟锁定：可以在构造时不立即锁定互斥锁，稍后再手动锁定。**
3. **手动解锁：允许在作用域结束前手动解锁。**
4. **支持条件变量：与 std::condition\_variable 配合使用时，必须使用 std::unique\_lock。**

**正如您所料，该标志必须存储在某个地方。 因此， std::unique\_lock 对象的大小通常大于 std::lock\_guard 对象的大小，并且在使用 std::unique\_lock 而不是 std::lock\_guard 时也会有轻微的性能损失，因为必须更新标志或 视情况进行检查。 如果 std::lock\_guard 足以满足您的需求，因此我建议优先使用它。 也就是说，在某些情况下 std::unique\_lock 更适合手头的任务，因为您需要利用额外的灵活性**

**如果明显不再需要该锁，您可以选择在特定代码分支中释放它。 这对于应用程序的性能很重要； 持有锁的时间超过所需的时间可能会导致性能下降，因为等待锁的其他线程无法继续执行超过必要的时间。**

**如果可能，仅在访问共享数据时锁定互斥体； 尝试在锁之外对数据进行任何处理。 特别是，在持有锁时不要执行任何耗时的活动，例如文件 I/O。 文件 I/O 通常比从内存中读取或写入相同量的数据慢数百（如果不是数千）倍。**

**保护共享数据的替代设施**

**一种特别极端（但非常常见）的情况是，共享数据仅在初始化时需要保护免受并发访问，但之后不需要显式同步。 这可能是因为数据一旦创建就是只读的，因此不存在可能的同步问题，或者可能是因为必要的保护是作为数据操作的一部分隐式执行的**

**1. std::once\_flag 和 std::call\_once 的作用**

* **std::once\_flag：**
  + **一个标志，用于指示某个操作是否已经执行过。**
  + **每个 std::once\_flag 实例对应一个不同的初始化操作。**
* **std::call\_once：**
  + **接受一个 std::once\_flag 和一个可调用对象（如函数、lambda 表达式等）。**
  + **确保可调用对象只会被执行一次，即使在多线程环境下**
* **静态局部变量的线程安全初始化**：
  + 在 C++11 中，静态局部变量的初始化是线程安全的，适用于实现单例模式。
  + 静态局部变量的初始化只会在一个线程中执行，其他线程会阻塞，直到初始化完成。

**std::shared\_mutex 的基本概念**

1. **共享锁（Shared Lock）：**
   * **多个线程可以同时持有共享锁。**
   * **适用于只读操作，允许多个线程并发读取数据。**
   * **使用 std::shared\_lock<std::shared\_mutex> 来获取共享锁。**
2. **独占锁（Exclusive Lock）：**
   * **只有一个线程可以持有独占锁。**
   * **适用于写操作，确保写操作的独占性。**
   * **使用 std::lock\_guard<std::shared\_mutex> 或 std::unique\_lock<std::shared\_mutex> 来获取独占锁。**
3. **锁的优先级：**
   * **如果任何线程持有共享锁，尝试获取独占锁的线程会阻塞，直到所有共享锁被释放。**
   * **如果任何线程持有独占锁，其他线程无法获取共享锁或独占锁，直到独占锁被释放**

**2. std::recursive\_mutex 的作用**

* **可重入：**
  + **std::recursive\_mutex 允许同一个线程多次锁定同一个互斥锁。**
  + **每次锁定必须对应一次解锁，只有当所有锁定都被解锁后，其他线程才能锁定该互斥锁。**
* **适用场景：**
  + **当一个函数可能会递归调用自身，或者多个函数可能会嵌套调用并需要锁定同一个互斥锁时，使用 std::recursive\_mutex 可以避免死锁。**

** 性能开销：**

* **std::recursive\_mutex 的实现通常比 std::mutex 更复杂，因此可能会有额外的性能开销。**

** 设计上的考虑：**

* **使用 std::recursive\_mutex 可能意味着代码设计存在问题。如果可能，应尽量避免嵌套锁定，而是通过重构代码来减少锁的嵌套。**

**.**

**在本章中，我讨论了在线程之间共享数据时，有问题的竞争条件可能会造成灾难性的后果，以及如何使用 std::mutex 和仔细的接口设计来避免它们。 您已经看到，互斥锁并不是万能的，并且确实存在死锁形式的问题，尽管 C++ 标准库提供了一个以 std::lock() 形式帮助避免这种情况的工具。 然后，您了解了一些避免死锁的进一步技术，然后简要介绍了锁所有权的转移以及有关选择适当的锁定粒度的问题。 最后，我介绍了为特定场景提供的替代数据保护工具，例如 std::call\_once() 和 std::shared\_mutex。**

**然而，我还没有讨论的一件事是等待其他线程的输入。 如果堆栈为空，则线程安全堆栈会引发异常，因此如果一个线程想要等待另一个线程将值压入堆栈（毕竟，这是线程安全堆栈的主要用途之一） ，它必须重复尝试弹出一个值，如果抛出异常则重试。 这会在执行检查时消耗宝贵的处理时间，而不会取得任何进展； 事实上，不断的检查可能会阻止系统中其他线程的运行，从而阻碍进度。 线程需要某种方法来等待另一个线程完成任务，而不消耗进程中的 CPU 时间。 第 4 章建立在我讨论过的用于保护共享数据的设施的基础上，并介绍了在 C++ 中线程之间同步操作的各种机制； 第 6 章展示了如何使用它们来构建更大的可重用数据结构。**

**std::lock()**

**std::call\_once() 和 std::shared\_mutex。**

**std::shared\_mutex**

**std::unique\_lock 可移动：std::unique\_lock 对象可以移动**

**std::lock\_guard 不可移动或复制：std::lock\_guard 对象不能移动或复制**