# 7.1 定义和意义

使用互斥量、条件变量，以及“期望”来同步阻塞数据的算法和数据结构。应用调用库函数，将会挂起一个执行线程，直到其他线程完成某个特定的动作。库函数将调用阻塞操作来对线程进行阻塞，在阻塞移除前，线程无法继续自己的任务。通常，操作系统会完全挂起一个阻塞线程(并将其时间片交给其他线程)，直到其被其他线程“解阻塞”；“解阻塞”的方式很多，比如解锁一个互斥锁、通知条件变量达成，或让“期望”就绪。

不使用阻塞库的数据结构和算法，被称为无阻塞结构。不过，无阻塞的数据结构并非都是无锁的，那么就让我们见识一下各种各样的无阻塞数据结构吧！

## 7.1.1 非阻塞数据结构

在第5章中，我们使用 `std::atomic_flag` 实现了一个简单的自旋锁。一起回顾一下这段代码。

清单7.1 使用 `std::atomic_flag` 实现了一个简单的自旋锁

```
class spinlock_mutex
{
  std::atomic_flag flag;
public:
  spinlock_mutex():
    flag(ATOMIC_FLAG_INIT)
  {}
  void lock()
  {
    while(flag.test_and_set(std::memory_order_acquire));
  }
  void unlock()
  {
    flag.clear(std::memory_order_release);
  }
};
```

这段代码没有调用任何阻塞函数，lock()只是让循环持续调用test_and_set()，并返回false。这就是为什么取名为“自旋锁”的原因——代码“自旋”于循环当中。所以，这里没有阻塞调用，任意代码使用互斥量来保护共享数据都是非阻塞的。不过，自旋锁并不是无锁结构。这里用了一个锁，并且一次能锁住一个线程。让我们来看一下无锁结构的具体定义，这将有助于你判断哪些类型的数据结构是无锁的。

## 7.1.2 无锁数据结构

作为无锁结构，就意味着线程可以并发的访问这个数据结构。线程不能做相同的操作；一个无锁队列可能允许一个线程进行压入数据，另一个线程弹出数据，当有两个线程同时尝试添加元素时，这个数据结构将被破坏。不仅如此，当其中一个访问线程被调度器中途挂起时，其他线程必须能够继续完成自己的工作，而无需等待挂起线程。

具有“比较/交换”操作的数据结构，通常在“比较/交换”实现中都有一个循环。使用“比较/交换”操作的原因：当有其他线程同时对指定数据的修改时，代码将尝试恢复数据。当其他线程被挂起时，“比较/交换”操作执行成功，那么这样的代码就是无锁的。当执行失败时，就需要一个自旋锁了，且这个结构就是“非阻塞-有锁”的结构。

无锁算法中的循环会让一些线程处于“饥饿”状态。如有线程在“错误”时间执行，那么第一个线程将会不停得尝试自己所要完成的操作(其他程序继续执行)。“无锁-无等待”数据结构，就为了避免这种问题存在的。

## 7.1.3 无等待数据结构

无等待数据结构就是：首先，是无锁数据结构；并且，每个线程都能在有限的步数内完成操作，暂且不管其他线程是如何工作的。由于会和别的线程产生冲突，所以算法可以进行无数次尝试，因此并不是无等待的。

正确实现一个无锁的结构是十分困难的。因为，要保证每一个线程都能在有限步骤里完成操作，就需要保证每一个操作可以被一次性执行完成；当有线程执行某个操作时，不会让其他线程的操作失败。这就会让算法中所使用到的操作变的相当复杂。

考虑到获取无锁或无等待的数据结构所有权都很困难，那么就有理由来写一个数据结构了；需要保证的是，所要得获益要大于实现成本。那么，就先来找一下实现成本和所得获益的平衡点吧！

## 7.1.4 无锁数据结构的利与弊

使用无锁结构的主要原因：将并发最大化。使用基于锁的容器，会让线程阻塞或等待；互斥锁削弱了结构的并发性。在无锁数据结构中，某些线程可以逐步执行。在无等待数据结构中，无论其他线程当时在做什么，每一个线程都可以转发进度。这种理想的方式实现起来很难。结构太简单，反而不容易写，因为其就是一个自旋锁。

使用无锁数据结构的第二个原因就是鲁棒性。当一个线程在获取一个锁时被杀死，那么数据结构将被永久性的破坏。不过，当线程在无锁数据结构上执行操作，在执行到一半死亡时，数据结构上的数据没有丢失(除了线程本身的数据)，其他线程依旧可以正常执行。

另一方面，当不能限制访问数据结构的线程数量时，就需要注意不变量的状态，或选择替代品来保持不变量的状态。同时，还需要注意操作的顺序约束。为了避免未定义行为，及相关的数据竞争，就必须使用原子操作对修改操作进行限制。不过，仅使用原子操作时不够的；需要确定被其他线程看到的修改，是遵循正确的顺序。

因为，没有任何锁(有可能存在活锁)，死锁问题不会困扰无锁数据结构。活锁的产生是，两个线程同时尝试修改数据结构，但每个线程所做的修改操作都会让另一个线程重启，所以两个线程就会陷入循环，多次的尝试完成自己的操作。试想有两个人要过独木桥，当两个人从两头向中间走的时候，他们会在中间碰到，然后不得不再走回出发的地方，再次尝试过独木桥。这里，要打破僵局，除非有人先到独木桥的另一端(或是商量好了，或是走的快，或纯粹是运气)，要不这个循环将一直重复下去。不过活锁的存在时间并不久，因为其依赖于线程调度。所以其只是对性能有所消耗，而不是一个长期的问题；但这个问题仍需要关注。根据定义，无等待的代码不会被活锁所困扰，因其操作执行步骤是有上限的。换个角度，无等待的算法要比等待算法的复杂度高，且即使没有其他线程访问数据结构，也可能需要更多步骤来完成对应操作。

这就是“无锁-无等待”代码的缺点：虽然提高了并发访问的能力，减少了单个线程的等待时间，但是其可能会将整体性能拉低。首先，原子操作的无锁代码要慢于无原子操作的代码，原子操作就相当于无锁数据结构中的锁。不仅如此，硬件必须通过同一个原子变量对线程间的数据进行同步。在第8章，你将看到与“乒乓”缓存相关的原子变量(多个线程访问同时进行访问)，将会成为一个明显的性能瓶颈。在提交代码之前，无论是基于锁的数据结构，还是无锁的数据结构，对性能的检查是很重要的(最坏的等待时间，平均等待时间，整体执行时间，或者其他指标)。

让我们先来看几个例子。