12.4 给关键部分加锁¶

问题¶

你需要对多线程程序中的临界区加锁以避免竞争条件。

解决方案¶

要在多线程程序中安全使用可变对象，你需要使用 threading 库中的 `Lock` 对象，就像下边这个例子这样：

```
import threading

class SharedCounter:
    '''
    A counter object that can be shared by multiple threads.
    '''
    def __init__(self, initial_value = 0):
        self._value = initial_value
        self._value_lock = threading.Lock()

    def incr(self,delta=1):
        '''
        Increment the counter with locking
        '''
        with self._value_lock:
             self._value += delta

    def decr(self,delta=1):
        '''
        Decrement the counter with locking
        '''
        with self._value_lock:
             self._value -= delta
```

`Lock` 对象和 `with` 语句块一起使用可以保证互斥执行，就是每次只有一个线程可以执行 with 语句包含的代码块。with 语句会在这个代码块执行前自动获取锁，在执行结束后自动释放锁。

讨论¶

线程调度本质上是不确定的，因此，在多线程程序中错误地使用锁机制可能会导致随机数据损坏或者其他的异常行为，我们称之为竞争条件。为了避免竞争条件，最好只在临界区（对临界资源进行操作的那部分代码）使用锁。 在一些“老的”Python 代码中，显式获取和释放锁是很常见的。下边是一个上一个例子的变种：

```
import threading

class SharedCounter:
    '''
    A counter object that can be shared by multiple threads.
    '''
    def __init__(self, initial_value = 0):
        self._value = initial_value
        self._value_lock = threading.Lock()

    def incr(self,delta=1):
        '''
        Increment the counter with locking
        '''
        self._value_lock.acquire()
        self._value += delta
        self._value_lock.release()

    def decr(self,delta=1):
        '''
        Decrement the counter with locking
```

```
        '''
        self._value_lock.acquire()
        self._value -= delta
        self._value_lock.release()
```

相比于这种显式调用的方法，with 语句更加优雅，也更不容易出错，特别是程序员可能会忘记调用 release() 方法或者程序在获得锁之后产生异常这两种情况（使用 with 语句可以保证在这两种情况下仍能正确释放锁）。 为了避免出现死锁的情况，使用锁机制的程序应该设定为每个线程一次只允许获取一个锁。如果不能这样做的话，你就需要更高级的死锁避免机制，我们将在12.5节介绍。 在 `threading` 库中还提供了其他的同步原语，比如 `RLock` 和 `Semaphore` 对象。但是根据以往经验，这些原语是用于一些特殊的情况，如果你只是需要简单地对可变对象进行锁定，那就不应该使用它们。一个 `RLock` （可重入锁）可以被同一个线程多次获取，主要用来实现基于监测对象模式的锁定和同步。在使用这种锁的情况下，当锁被持有时，只有一个线程可以使用完整的函数或者类中的方法。比如，你可以实现一个这样的 SharedCounter 类：

```
import threading

class SharedCounter:
    '''
    A counter object that can be shared by multiple threads.
    '''
    _lock = threading.RLock()
    def __init__(self, initial_value = 0):
        self._value = initial_value

    def incr(self,delta=1):
        '''
        Increment the counter with locking
        '''
        with SharedCounter._lock:
            self._value += delta

    def decr(self,delta=1):
        '''
        Decrement the counter with locking
        '''
        with SharedCounter._lock:
             self.incr(-delta)
```

在上边这个例子中，没有对每一个实例中的可变对象加锁，取而代之的是一个被所有实例共享的类级锁。这个锁用来同步类方法，具体来说就是，这个锁可以保证一次只有一个线程可以调用这个类方法。不过，与一个标准的锁不同的是，已经持有这个锁的方法在调用同样使用这个锁的方法时，无需再次获取锁。比如 decr 方法。 这种实现方式的一个特点是，无论这个类有多少个实例都只用一个锁。因此在需要大量使用计数器的情况下内存效率更高。不过这样做也有缺点，就是在程序中使用大量线程并频繁更新计数器时会有争用锁的问题。 信号量对象是一个建立在共享计数器基础上的同步原语。如果计数器不为0，with 语句将计数器减1，线程被允许执行。with 语句执行结束后，计数器加 1 。如果计数器为0，线程将被阻塞，直到其他线程结束将计数器加1。尽管你可以在程序中像标准锁一样使用信号量来做线程同步，但是这种方式并不被推荐，因为使用信号量为程序增加的复杂性会影响程序性能。相对于简单地作为锁使用，信号量更适用于那些需要在线程之间引入信号或者限制的程序。比如，你需要限制一段代码的并发访问量，你就可以像下面这样使用信号量完成：

```
from threading import Semaphore
import urllib.request

# At most, five threads allowed to run at once
_fetch_url_sema = Semaphore(5)

def fetch_url(url):
    with _fetch_url_sema:
        return urllib.request.urlopen(url)
```

如果你对线程同步原语的底层理论和实现感兴趣，可以参考操作系统相关书籍，绝大多数都有提及。