第七讲 管程和阻塞同步

本讲介绍一些更一般的同步机制。

一．管程（Monitor）

管程是 Brinch-Hansen 和 Hoare 于 70 年代提出的，用于解决线程间的同步问题。以典型的生产者-消费者问题为例，用自旋锁实现时，生产者的代码如下：

```
mutex.lock();
try {
    queue.enq(x);
}
finally {
    mutex.unlock();
}
```

当有界队列 queue 满时，enq()方法无法执行。这意味着锁 mutex 可能一直被阻塞。虽然可以通过访问队列 queue 的内部状态来解决，但这对修改和维护程序造成负担（每次访问队列前都需判断队列的状态，这对程序员来说是不小的挑战）。而且，在多线程环境下，多个生产者/消费者线程都不得不跟踪锁和队列，并保持一致的锁机制，以保证程序的正确性。这实在不是好的编程模式。

为解决这类问题，可以把锁机制和共享对象（如上述队列）打包，让对象管理自己的同步机制，这就是管程的概念。管程包含同步机制、对象和方法，支持对对象（方法）的互斥访问。管程的方法被调用时须获得管程的锁，在调用返回时释放锁。任何时刻至多有一个线程可以持有管程的锁，在该线程终止时释放锁（如果还未释放）。如果线程无法立即获得锁，要么自旋，不断检测锁是否空闲，要么阻塞，暂时放弃处理器进入休眠。当锁住某个对象（管程）的线程需要长时间等待某个事件发生时，释放锁以便于其它线程访问该对象是一个非常好的策略。管程同时提供自旋锁和阻塞锁两种机制，大大提高了管程的灵活性。

在很多并发场景下，单纯的互斥机制是不够的。线程可能需要等待某个条件成立，才能继续执行。这可以用条件（condition）对象来定义。每个条件对象 c 对应着一个断言 P_c（即条件）。形象地讲，条件对象 c 定义了一个线程（休眠）等待区，所有等待 P_c 成立的线程都在这个等待区内休眠。其它线程则进入管程执行，改变管程的状态，并唤醒等待中的线程。

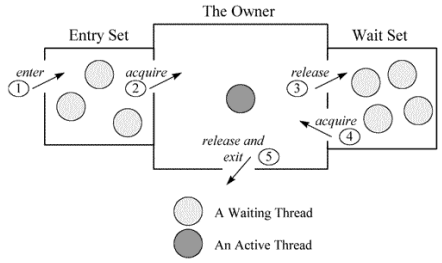
管程把阻塞锁定义为条件对象，包含以下方法：

1. void await()：线程休眠，进入条件对象定义的等待区。
2. void signal()：唤醒相应等待区内的一个线程。
3. void signalAll(): 唤醒相应等待区内的所有线程。

当线程调用 signal()时，该线程应已在管程内；而当线程被 signal()唤醒时，将尝试进入管程。但唤醒者 A 与被唤醒者 B 之间只能有一个线程占用该管程。唤醒线程有两种方式处理：

1. 非阻塞条件（Signal and Continue，Mesa style）：唤醒者 A 继续执行，直至该线程释放锁。然后，被唤醒者 B 竞争进入管程。
2. 阻塞条件（Signal and Wait，Hoare-style）：唤醒者 A 进入等待区，直到被唤醒者 B 释放锁。然后，唤醒者 A 竞争进入管程。

下图说明管程的工作流程：（1）线程申请锁；（2）线程获得锁，进入管程；（3）线程调用 await()，释放锁并进入管程；（4）线程被唤醒，获得锁，返回管程；（5）线程释放锁，退出管程。



例：有界 FIFO 队列，采用显式锁和条件实现。当队列非空时，通过 notEmpty 条件对象唤醒正在等待的出队线程。当队列未满时，通过 notFull 条件对象唤醒正在等待的入队线程。

```
Class LockedQueue<T> {
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition(); //创建条件对象
    final Condition notEmpty = lock.newCondition();
    final T[] items;
    int tail, head, count; …
}
public LockedQueue(int capacity) {
  items = (T[]) new Object[capacity];
}
public void enq(T x) {
  lock.lock();
  try {
    while (count == items.length)
        notFull.await(); //满则等待
    Enqueue x;
    ++count;
    notEmpty.signal(); //唤醒等待 notEmpty 的线程
  } finally {
    lock.unlock();
  }
}
public T deq() {
  lock.lock();
  try {
    while (count == 0)
        notEmpty.await(); //空则等待
    Dequeue the first element;
    --count;
    notFull.signal(); //唤醒等待 notFull 的线程
    Return the element dequeued;
```

```
    } finally {
        lock.unlock();
    }
}
```

条件对象存在唤醒丢失（Lost Wakeup）问题：在调用 signal()方法唤醒等待线程时，可能有等待线程会一直无法被唤醒。例如，对上述 enq()方法稍作如下修改：

```
public void enq(T x) {
    lock.lock();
    try {
        while (count == items.length)
                notFull.await(); //满则等待
        Enqueue x;
        ++count;
        if (count == 1) {
            notEmpty.signal(); //仅当队列从空变为非空时，唤醒等待中的线程
        }
    } finally {
        lock.unlock();
    }
}
```

当 count 值从 0 变为 1 时，可以唤醒一个出队线程。这在顺序执行时是正确的。但在以下并发场景中，存在唤醒丢失问题。假设线程 A、B、C 和 D 共享一个队列，且队列初始为空：

1. 线程 A 和 B 都试图从该空队列中出队元素。当它们检测到队列为空时，在 notEmpty 条件上等待。
2. 线程 C 入队第一个元素，在 notEmpty 条件上唤醒一个线程。不失一般性，假设线程 A 被唤醒。
3. 在线程 A 获得锁之前，线程 D 入队第二个元素。由于此时队列包含两个元素，线程 D 不会唤醒任何线程。
4. 线程 A 获得锁，出队第一个元素。此时，队列中仍有一个元素，但线程 B 只能等待，而失去被唤醒的机会。B 成为唤醒丢失的受害者。

解决办法：用 signalAll()唤醒等待该条件的所有线程；或者限时等待。但这两种办法都会产生额外的性能开销。

目前主流的并发语言都支持互斥和管程同步。

Features	Java	POSIX C	C++11
Threads	java.lang.thread class	pthread_t type	std::thread class
Mutual Exclusion	synchronized blocks	pthread_mutex_t type	std:mutex class
Thread Local Storage	ThreadLocal<T> class template	pthread_key_t type	thread_local variables
Atomic Operations	volatile variables	/	std::atomic<T> class template
Monitors/ Waits for a predicate	**wait() and notify() of java.lang.Object class, used inside synchronized blocks**	**pthread_cond_t type and associated API functions: pthread_cond_wait(), pthread_cond_signal()**	**std::condition_variable and std::condition_variable_any classes**

Java 以 synchronized 关键字修饰代码块或方法的形式提供对管程的内置支持，采用非阻塞条件，并内置 wait()、notify()和 notifyAll()方法。

二．读-写（Readers-Writers）锁

许多共享对象都有如下特性：大多数是读者访问，即只返回对象的状态而不修改对象；只有少数是写者访问，即会修改对象的状态。读-写锁允许多个读者并发访问。读-写锁的特性：

- 任一线程持有读锁或写锁时，其它线程不能获得写锁；
- 任一线程持有写锁时，其它线程不能获得读锁；除此之外，多个线程可获得多个读锁；

```
public class SimpleReadWriteLock implements ReadWriteLock {
    int readers; //记录有多少读者，writer == false 且 readers == 0 时可获得写锁
    boolean writer; //记录是否有写者，writer == false 时可获得读锁
    Lock lock; //同步所有的锁
    Lock readLock, writeLock; //读锁，写锁
    Condition condition; //条件对象，与 lock 关联
    public SimpleReadWriteLock() {
        writer = false; readers = 0;
        lock = new ReentrantLock();
        readLock = new ReadLock(); writeLock = new WriteLock();
        condition = lock.newCondition(); //条件对象，与 lock 关联
    }
class ReadLock implements Lock { //lock()  和  unlock()只能由内部类访问
    public void lock() {
        lock.lock();
        try {
            while (writer) //等待释放写锁
                condition.await();
            readers++; //获得读锁，readers 计数器加 1
        } finally {
            lock.unlock();
        }
    }
    public void unlock() {
        lock.lock();
        try {
            readers--; //释放读锁，readers 计数器减 1
            if (readers == 0) //唤醒等待 condition 的所有线程
                condition.signalAll();
        } finally {
            lock.unlock();
        }
    }
}
```

SimpleReadWriteLock 对象的读/写锁方法间需要同步，通过互斥锁 lock 和条件 condition 实现。尽管 SimpleReadWriteLock 实现是正确的，但读者一般比写者频繁得多，写者可能被一系列读者阻塞很长时间。

公平读写锁（Fair Readers-Writers Lock）赋予写者以优先级，保证一旦有写者申请（写）锁，则不允许更多的读者获取（读）锁，只允许读者释放（读）锁，直到写者获得（写）锁为止。

```
public class FifoReadWriteLock implements ReadWriteLock {
    int readAcquires, readReleases;
    //读锁请求总数，读锁释放总数，二者相等时，没有线程持有读锁。
    boolean writer;
    Lock lock;
    Condition condition;
    Lock readLock, writeLock;
    public FifoReadWriteLock() {
        readAcquires = readReleases = 0;
        writer = false; //writer 为 true 时，readAcquires 不再增加
        lock = new ReentrantLock();
        condition = lock.newCondition();
        readLock = new ReadLock();
        writeLock = new WriteLock();
    }
    …
}
```

FiFoReadWriteLock 算法的关键在于把读锁的 readers 计数器拆分成两个，readAcquires 负责记录申请读锁的次数，readReleases 负责记录释放读锁的次数。另一处变化是把 writer 变量看作一个互斥锁。当写者试图获得（写）锁时，把 writer 置为 true，使后续的读者无法获得（读）锁。在所有获得（读）锁的读者释放其（读）锁时，即 **readAcquires == readReleases**，最后释放（读）锁的读者将调用 signalAll()方法。此时，等待中的所有读者和写者线程都会被唤醒，但只有写者线程可以获得其申请的（写）锁。

```
private class ReadLock implements Lock {
    public void lock() {
        lock.lock();
        try {
            while (writer)
                condition.await();
            readAcquires++;
        } finally {
            lock.unlock();
        }
    }
    public void unlock() {
        lock.lock();
        try {
```

```
                readReleases++;
                if (readAcquires == readReleases)
                    condition.signalAll();
            } finally {
                lock.unlock();
            }
        }
    }
    private class WriteLock implements Lock {
        public void lock() {
            lock.lock();
            try {
                while (writer) //等待写者释放
                    condition.await();
                writer = true;
                while (readAcquires != readReleases) //等待读者释放
                    condition.await();
            } finally {
                lock.unlock();
            }
        }
        public void unlock() {
            writer = false;
            condition.signalAll();
        }
    }
```

三．可重入锁（Reentrant Lock）

可重入锁是指一个线程可以在不释放锁的情况下，重复申请并获得该锁。这样，线程在递归或重复调用时不会陷入死锁。可重入锁的实现在本质上是使用一个计数器来记录线程获得锁的次数。

```
    public class SimpleReentrantLock implements Lock {
        Lock lock;
        Condition condition;
        int owner, holdCount;
        public SimpleReentrantLock() {
            lock = new SimpleLock();
            condition = lock.newCondition();
            owner = 0;
            holdCount = 0;
        }
        public void lock() {
            int me = ThreadID.get();
            lock.lock();
            try {
```

```
            if (owner == me) {
                holdCount++;
                return;
            }
            while (holdCount != 0)
                condition.await();
            owner = me;
            holdCount = 1;
        } finally {
            lock.unlock()
        }
    }
    public void unlock() {
        lock.lock();
        try {
            if (holdCount == 0 || owner != ThreadID.get())
                throw new IllegalMonitorStateException();
            holdCount--;
            if (holdCount == 0) {
                condition.signal();
            }
        } finally {
            lock.unlock();
        }
    }
}
```

四．信号量（Semaphore）

互斥锁保证在任何时刻至多有一个线程进入临界区。信号量是互斥锁的一般形式，允许最多 capacity 个线程进入临界区。capacity 是信号量的容量，在初始化时确定。信号量本身是一个原子计数器，记录进入临界区的线程数。Hoare 已证明信号量机制和管程在理论上是等价的[1]。

- acquire()：P 操作，请求进入临界区。
- release()：V 操作，离开临界区。

用管程实现信号量：

```
public class Semaphore {
    final int capacity;
    int state;
    Lock lock;
    Condition condition;
    public Semaphore(int c) {
        capacity = c; //信号量容量
        state = 0;
        lock = new ReentrantLock();
```

```
        condition = lock.newCondition();
    }
    public void acquire() {
        lock.lock();
        try {
            while (state == capacity) //信号量满
                condition.await();
            state++;
        } finally {
            lock.unlock();
        }
    }
    public void release() {
        lock.lock();
        try {
            state--;
            condition.signalAll(); //唤醒等待线程
        } finally {
            lock.unlock();
        }
    }
}
```

[1]. C. A. R. Hoare, Monitors: an operating system structuring concept, Communications of the ACM 17, 10 (October 1974), 549-557.