# 第三章 线程（下）

## 3.1 线程安全与线程兼容与对立

线程安全：当多个线程访问同一个对象时，如果不同考虑这些线程在运行时环境下的调度和交替执行，也不需要进行额外的同步，或者在调用方进行任何其他的协调操作，调用这个对象的行为都可以获得正确的结果，那这个对象是线程安全的。它包括：

（1）不可变。如果它访问的对象是不可修改的，那么它本身就是安全的。比如：

①final 修饰：public final a =100;

②java.lang.String: String s = “String”;string 常量

③枚举类型：public enum Color{RED,GREEN}

④java.lang.Number 的子类，比如 Long，Double

⑤BigInteger,BigDecimal（数值类型的高精度实现）

（2）绝对线程安全：满足上述定义的线程就为绝对县城安全，javaAPI 标注自己是线程安全的类绝大多数不是绝对线程安全的。

（3）相对线程安全：通常意义上的线程安全，需要保证这个对象单独操作是线程安全的，调用的时候不需要做额外的保障措施，但是对于一些特定顺序的连续调用，就需要在调用时使用同步手段保证调用的正确性。如 Vector，HashTable 等。

（4）线程的兼容和对立。线程兼容指对象本身不是线程安全的，但是可以通过在调用端正确地使用同步手段来保证对象在并发环境中可以安全使用；线程对立是无论调用端是否采用了同步措施，都无法在多线程环境中并发使用的代码。

## 3.2 线程的安全实现——互斥同步

同步的互斥实现方式：临界区、互斥量、信号量 。解决的方式就是使用 Synchronized 关键字，经过编译后，会在同步块前后行程 monitorenter 和 monitorexit 两个字节码。synchronized 同步块对自己是可重入的，不会将自己锁死；同步块在已进入的线程执行完之前，会阻塞后面其它线程的进入，以保证程序安全性。

还有一种方式是重入锁 (java.util.concurrent)。相比采用 synchronized，重入锁可实现：等待可中断、公平锁、锁可以绑定多个条件。Synchronized 表现为原生语法层面的互斥锁，而 RenentrantLock 表现为 API 层面的互斥锁。

他们的性能比较如图 1 所示。

图 1

## 3.3 线程的安全实现——非阻塞同步

1.阻塞同步：互斥同步存在的问题是进行线程阻塞和唤醒所带来的性能问题，这种同步称为阻塞同步。这是一种悲观并发策略。

2.非阻塞同步：不同于前者，它是基于冲突检测的乐观并发策略，就是先进行操作，如

果没有其他线程征用共享数据，则操作成功；否则就是产生了冲突，采取不断重试直到成功为止的策略。这种策略不需要把线程挂起，称为非阻塞同步。

非阻塞同步使用硬件处理器指令进行不断重试策略：

①测试并设置（Test-and-Set）；

②获取并增加（Fetch-and-Increment）；

③交换（Swap）；

④比较并交换（CAS）；

⑤加载链接，条件存储（LL, SC）。

例如 java 实现类 AtomicInteger，AtomicDouble 等等。

**例：非阻塞实例**

**代码：**

```
class Counter{
    private volatile int count =0;
    public synchronized void increment(){
        count++;//若要线程安全执行 count++，需要加锁
    }
    public int getCount(){
        return count;
    }
}
```

**例：上例用 AtomicInteger 来改进**

**代码：**

```
class Counter{
    private AtomicInteger int count =new AtomicInteger();
    public void increment(){
        count。incrementAndGet();
    }
    //使用 AtomicInteger 之后不需要加锁，也可以实现线程安全
    public int getCount(){
        return count.get();
    }
}
```

## 3.4 线程的安全实现——无同步方案

1.可重入代码：也叫纯代码。相对线程安全来说，可以保证线程安全。可以在代码执行过程中中断它，转而去执行另一段代码，而在控制权返回后，原来程序不会出现任何错误。

2.线程本地存储：如果一段代码所需数据必须与其它代码共享，那么看这些数据代码能否保证在同一个线程中执行，如果可以，就可以把共享数据的可见范围限定在同一个线程之内，这样无需同步也能保证线程之间不出现数据争用问题。

**例：TthreadLocal——本地存储例子**

**代码：**

```
public class SequenceNumber{
    //①通过匿名内部类覆盖 ThreadLocal 的 initialValue()
```

```
    //方法指定初始值
    private static ThreadLocal<Integer>seqNum = new ThreadLocal<Integer>(){
        public Integer initialValue(){
            return 0;
        }

    };
    public int getNextNum(){
        //②获取下一个序列值
        seqNum.set(seqNum.get()+1);
        return seqNum.get();
    }
    public static void main(String[] args){
        SequenceNumber sn = new SequenceNumber();
        //③三个线程共享 sn，各自产生序列号
        TestClient t1 = new TestClient(sn);
        TestClient t2 = new TestClient(sn);
        TestClient t3 = new TestClient(sn);
        t1.start();
        t2.start();
        t3.start();
    }
    private static class TestClient extends Thread{
        private SequenceNumber sn;
        public TestClient(SequenceNumber sn){
            this.sn =sn;
        }
        public void run(){
            for(int i=0;i<3;i++){
                //④每个线程打出 3 个序列值

    System.out.println("thread["+Thread.currentThread().getName()+"]sn["+sn.getNextNum
()+"]");
            }
        }
    }
}
```

**程序分析：**

seqNum 利用一个匿名类来覆盖 ThreadLocal 里面的方法，直接返回 0。第二个方法是通过刚刚定义完的变量 get 掉值然后加 1，设置为新的值。

第三步利用后面产生的线程类，然后共享 sn，start()方法各自产生序列号。下面的线程类表述了第三步是要干嘛。其实是把 sn 赋值给本类的 sn，run()方法就是循环三遍，每一次都打印出线程名字和 sn 的值。

# 3.5 锁优化

它主要有这样几种优化方式：

1.自旋锁。互斥同步挂起线程和恢复线程都需要转入内核态中完成，这些操作给系统的并发性能带来很大压力。自旋锁就是，如果物理机器有一个以上处理器能够让两个或以上的线程同时并行执行，那么可以让后面请求所的线程等等，但不放弃处理器的执行时间，看看有锁的线程是否很快释放。为了让线程等待，可以让线程执行一个忙循环。java 中自旋默认 10 次。

2.自适应锁：自适应自旋就是自旋时间不固定，而是由前一次在同一个锁上的自旋时间及锁拥有者的状态决定。如果在同一个锁对象上自旋等待刚刚成功获得锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也很有可能再次成功，进而它允许自旋等待相对时间更长一点。

3.锁消除：JVM 即时编译器在运行时，对一些（**代码要求同步，但是被检测到不可能存在共享数据竞争的锁**）进行消除。

判定依据：如果判断在一段代码中，堆上的所有数据都不会逃逸出去从而被其它线程访问到，那就可以把他们当做栈上数据对待，认为他们是线程私有的，同步加锁自然无需进行。

4.锁粗化：如果一系列的连续操作都对同一个对象反复加锁，甚至加锁操作出现在循环体重，那么即使没有线程争用，频繁的进行互斥同步也会导致不必要的性能损耗，此时只需要将同步块范围扩大即可，即锁粗化。

5.偏向锁：它的目的是消除数据无竞争情况下的同步原语，进一步提高程序运行的性能。偏向锁就是在无竞争的情况下把整个同步都消除掉，连 CAS（比较并交换）操作都不做。

偏向的意思是这个锁会偏向第一个获得它的线程，如果接下来的执行中，该锁没有被其它线程获取，则持有偏向锁的线程永远不需要再进行同步。