synchronized

1.底层实现原理：

它通过编译后加上不同的机器指令来实现。即每个对象有一个监视器锁（monitor）。当monitor被占用时就会处于锁定状态，线程执行monitorenter指令时尝试获取monitor的所有权，通过monitorexit来释放monitor的所有权，其过程如下：

- 如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。
- 如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1
- 如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。
- 执行monitorexit的线程必须是objectref所对应的monitor的所有者。指令执行时，monitor的进入数减1，如果减1后进入数为0，那线程退出monitor，不再是这个monitor的所有者。其他被这个monitor阻塞的线程可以尝试去获取这个 monitor 的所有权。
- 我们应该能很清楚的看出Synchronized的实现原理，Synchronized的语义底层是通过一个monitor的对象来完成，其实wait/notify等方法也依赖于monitor对象，这就是为什么只有在同步的块或者方法中才能调用wait/notify等方法，否则会抛出java.lang.IllegalMonitorStateException的异常的原因。

2.线程安全

在多线程编程中，有可能出现多个线程访问同一个资源的情况，由于每个线程执行的过程是不可控的，所以可能导致最终的结果与实际上的愿望相违背或者直接导致程序出错。但当多个线程访问某个类时，不管运行时环境采用何种调度方式或者这些线程将如何交替执行，并且在主调代码中不需要任何额外的同步或者协同，这个类都能表现出正确的行为，那么就称这个类是线程安全的。

解决线程安全：同步互斥访问：synchronized和lock

3.常见知识点

- java的每一个对象都有一个锁标记，多个线程访问某个对象时只有获取了该对象的锁才能访问。synchronized标记一个方法或代码块，当某个线程调用该对象的synchronized方法或者访问synchronized代码块时，便获得了该对象的锁。
- 当一个线程正在访问一个对象的的synchronized方法，其他线程不能访问该对象的其他synchronized方法，但能访问该对象的非synchronized方法。当两个线程访问不同对象的synchronized方法时，不会产生线程安全问题。
- 每个类也有一个锁，用来控制对static数据成员的并发访问，如果一个线程执行一个对象的非static synchronized方法，另一个线程执行这个对象象所属类的static synchronized方法，不会发生互斥，因为访问static synchronized占用的是类锁，而访问static synchronized方法占用的是对象锁，所以不会互斥
- 对于synchronized方法或者synchronized代码块，当出现异常时，jvm会自动释放当前线程所占用的锁，所以不会由于异常导致死锁现象
- 锁重入即当一个线程获取该对象锁后，再次请求时是可以再次得到该对象锁的并且同步不具有继承性，synchronized是可重入锁，但无法让等待线程响应中断，并且是非公平锁

4.volatile关键字和synchronized关键字区别

- volatile修饰变量，synchronized修饰方法和代码块
- 多线程访问volatile不会发生阻塞;而synchronize会发生阻塞
- volatile不能保证变量的原子性;synchronize可以保证变量原子性;
- volatile是变量在多线程之间的可见性;synchronize是多线程之间访问资源的同步性

5.synchronized与lock的区别

- synchronized为内置关键字，而lock为接口

- synchronized无法判断是否获取锁，而lock可以
- synchronized无法让等待线程响应中断，lock可以
- synchronized不需要手动释放锁，而lock必须手动释放锁，否则可能会发生死锁
- lock可以利用读写锁实现并发读来提高读操作的效率

线程与进程的区别

- **基本单位：**线程作为调度和分配的基本单位，进程作为拥有资源的基本单位
- **并发性:**不仅进程之间可以并发执行，同一个进程的多个线程之间也可并发执行
- **拥有资源:**进程是拥有资源的一个独立单位，线程不拥有系统资源，只拥有一点在运行中必不可少的资源，但它可与同属一个进程的其它线程共享进程所拥有的全部资源.
- **系统开销:**由于创建或撤销进程时，系统都要为之分配或回收资源，如内存空间、I/O 设备等，因此操作系统所付出的开销远大于创建或撤销线程时的开销。类似地，在进行进程切换时，涉及当前执行进程 CPU 环境的保存及新调度进程 CPU 环境的设置。而线程切换时只需保存和设置少量寄存器内容，开销很小。此外，由于同一进程内的多个线程共享进程的地址空间，因此，这些线程之间的同步与通信非常容易实现，甚至无需操作系统的干预。
- **通信方面：**进程间通讯有管道、信号量、信号、共享内存、消息队列、socket来维护，而线程间通过通道、共享内存、信号灯来进行通信。

并发工具类（CountDownLatch、CyclicBarrier以及Semaphore）

CountDownLatch和CyclicBarrier都能够实现线程之间的等待，只不过它们侧重点不同：

- CountDownLatch一般用于某个线程A等待若干个其他线程执行完任务之后，它才执行；而CyclicBarrier一般用于一组线程互相等待至某个状

态，然后这一组线程再同时执行；

- CountDownLatch是不能够重用的，而CyclicBarrier是可以重用的；
- 闭锁CountDownLatch用于等待事件、栅栏CyclicBarrier是等待线程.
- 闭锁CountDownLatch做减计数，而栅栏CyclicBarrier则是加计数。

Semaphore其实和锁有点类似，它一般用于控制对某组资源的访问权限即同时访问特定资源的线程数量（最多可允许几个线程同时访问）
Exchange用于线程间交换数据，它提供一个共同点，当两个线程都到达这个共同点时可以交换彼此数据
CountDownLatch应用场景：例如解析一个Excel里的多个sheet数据，使用多线程时可以每个线程解析一个sheet里的数据，主线程需要等待所有线程的sheet的解析操作
CyclicBarrier应用场景：可以应用于多线程计算数据，最后合并计算结果的场景，并且CyclicBarrier的计数器可以使用reset()方法重置，所以能处理更为复杂的业务场景
Semphore应用场景：可以用作流量控制，比如数据库连接
Exchanger应用场景：可以用于遗传算法，也可用于校对

原子操作类

java提供了java.util.concurrent.atomic包，一共提供了13个类，属于4种类型的原子更新方式，分别是原子更新基本类型、原子更新数组、原子更新引用和原子更新属性
原子更新基本类型：
AtomicBoolean、AtomicInteger、AtomicLong
以AtomicInteger为例，其中的getAndIncrement()方法已原子方式将当前值加1
实现原理：
首先取得AtomicInteger里存储的数值，然后进行加1操作，接下来调用compareAndSet方法进行原子更新操作（即CAS方式），该方法先检查当前数值是否等于current，等于意味着AutomicInteger的值没有被其他线程修改，将当前值更新成next值，如果不等于则会进入for循环重新进行compareAndSet操作

对于AutomicBoolean它是先把Boolean转成整型然后再使用compareAndSwapInt进行CAS

原子更新数组：

例如AtomicIntegerArray,它的addAndGet()方法的原理是会将当前数组复制一份，所以对元素进行修改时，不会影响传入数组

原子更新引用类型：

例如AutomicReference,首先创建对象设置进AtomicReferene中，然后调用compareAndSet方法进行原子更新操作

原子更新字段类：

例如AtomicIntegerFieldUpdater，更新需要两步：由于该类是抽象类，使用时必须使用静态方法newUpdater()创建一个更新器，并且设置想要更新的类和属性，第二步，更新类的字段必须使用public volatile修饰符