1. 创建线程有几种不同的方式？

Java多线程实现方式主要有四种，又分为两类：

* 线程执行完后不返回结果：a. 继承Thread类；b. 实现Runnable接口
* 返回结果：c. 实现Callable接口通过FutureTask包装器来创建Thread线程；d. 使用ExecutorService、Callable、Future实现有返回结果的线程

**继承Thread类创建线程**

通过继承Thread类的方法，来创建并启动多线程的步骤如下：

①定义Thread类的子类MyThread，并重写该类的run()方法，该方法的方法体定义的是线程需要完成的任务，run()方法也称为线程执行体；

②创建Thread子类的实例，也就是创建了线程对象；

③通过调用线程的start()方法启动线程。

**实现Runnable接口创建线程**

通过实现Runnable接口的方法，创建并启动线程步骤如下：

①定义Runnable接口的实现类，同样要重写run()方法这是线程的执行体，Thread中的run()方法相同。

②创建Runnable实现类的实例，并用这个实例作为Thread的target来创建Thread对象，这个Thread对象才是真正的线程对象。

③通过调用线程对象的start()方法来启动线程。

**Thread与Runnable的区别：**

1.Runnable避免了单继承局限，可以同时实现多了接口，Thread是Runnable接口的子类。

2.使用Runnabe实现多继承可以更好地描述出程序共享的概念。

3.Runnable采用了代理设计模式，（在静态代理模式中有两个角色一个真实角色用来实现真正的业务操作，另一个代理角色用来完成资源的调动，辅助真实业务的完成）

**实现Callable接口通过FutureTask包装器来创建Thread线程**

通过实现Callable接口和FutureTask实现的方法，创建并启动线程步骤如下(注意：我把步骤合并了详细请看代码步骤)

①创建一个类MyCallble并实现Callable接口，然后重写call方法，必须返回数据，要不然报错！call()方法比run()方法功能要强大。

②创建MyCallable实现类的对象；创建FutureTask对象，并将myCallable实现类的对象作为参数传递到FutureTask的构造器中。

③创建Thread对象并将FutureTask对象作为参数传递Thread的构造器中。

④调用Thread启动的start()方法；调用FutureTask中的get()方法获取返回值。

**使用ExecutorService、Callable、Future实现有返回结果的线程**

1.public static ExecutorService newFixedThreadPool(int nThreads){...}

ExecutorService fixedThreadPool = Executors.newFixedThreadPool(2);

fixedThreadPool.submit(**Thread**);

创建定长线程池，就是控制了线程池的大小，简单点的说就是控制了线程的数量，

2.    public static ExecutorService newCachedThreadPool(){...}

ExecutorService cachedThreadPool = Executors.newCachedThreadPool();

创建一个可缓存的线程池，调用execute将重用以前构造的可用的线程。如果现有线程没有可用的，则创建一个新线程并添加到池中，终止时并从缓存中移除那些已有60秒钟未被使用的线程，参数中不做限制处理，限制是由jvm以及计算机的处理性能决定的。

**3.**   **public static ExecutorService newSingleThreadExecutor(){...}**

ExecutorService ThreadPool=Executors.newSingleThreadExecutor();

创建一个单线程化的Executor，限制线程池大小为1，为了保证线程按照顺序进行

      newSingleThreadExecutor相比newFixedThreadPool对线程数做了限制

**4.public     static   ScheduledExecutorService newScheduledThreadPool(int corePoolSize) {...}**

它和第一个线程不同的地方在于可以定时和限制运行周期，多数情况下可用来替代Timer类；实际上就是根据两个参数实现的

**ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(3);**

//参数分别为 线程，延迟开始时间（定时），线程运行时间间隔（周期），时间单位说明前面两个参数单位为秒）

scheduledThreadPool.scheduleAtFixedRate(new ThreadForTest(), 2, 1, TimeUnit.SECONDS);

*//定时，其实就是缺省了周期参数，比较上述*

*//scheduledThreadPool.schedule(new ThreadForTest(), 1, TimeUnit.SECONDS);*

2.请介绍线程有哪些状态？

**1.NEW：**初始状态，线程被创建后还没有启动，这时候通过线程的getState方法可以获得其状态为NEW。

**2.RUNNABLE：**运行状态，该状态分为就绪状态和运行中状态，运行中状态自然就是指线程正在运行，就绪状态是指线程已经做好准备且不处于排队状态，也就是说，已经调用了start()函数，其后一些工作结束后(如等待时间片分配、等待锁或者说从sleep()状态中唤醒)。

**3.TERMINATED：**终止状态，在线程的run()方法运行结束后，线程就会处于死亡状态。

**4.BLOCKED：**阻塞状态，在进入synchronized关键字修饰的方法或代码块(获取锁)时的状态。

**5. WAITTING：**等待状态，如果该线程的操作是由其他线程进行控制的，则没有收到运行通知，该线程会一直处于等待状态，直到收到通知才会进行后续操作。

**6. TIMED\_WAITTING：**超时等待状态，与WAITTING类似，但是在规定时间内没有接收到运行通知，会自己返回，回到初始状态

3.请谈谈你对ThreadLocal的理解？

ThreadLocal是什么？ ThreadLocal并不是一个Thread，而是Thread的局部变量，很多地方叫做线程本地变量，也有些地方叫做线程本地存储，其实意思差不多。ThreadLocal为变量在每个线程中都创建了一个副本，那么每个线程可以访问自己内部的副本变量。

它采用空间来换取时间的方式，解决多线程中相同变量的访问冲突问题。

ThreadLocal不是用来解决多线程下访问共享变量问题的。

实际是在Thread类里面有一个ThreadLocalMap，用于存储每一个线程的变量的引用，这个Map中的键为ThreadLocal对象，而值对应的是ThreadLocal通过set放进去的变量引用。(这句重点！！)

ThreadLocal通过set(共享变量)然后再通过ThreadLocal方法get的是共享变量的引用！假设有多个线程都在其执行过程中将共享变量加入到自己的ThreadLocal中，那就是每个线程都持有一份共享变量的引用副本，注意是引用副本，共享变量的实例只有一个。所以，ThreadLocal不是用来解决线程间共享变量的访问的事儿的。

总结：

ThreadLocal在每个线程中对该变量会创建一个副本，即每个线程内部都会有一个该变量，且在线程内部任何地方都可以使用，线程之间互不影响，这样一来就不存在线程安全问题，也不会严重影响程序执行性能。

请介绍ThreadLocal的应用场景是什么，请举几个例子？

**1数据库连,2.Session管理,3.让SimpleDateFomat变成线程安全**

在多线程中，对于一个线程来说df.get的时候，首先尝试获得线程它自己的ThreadLocalMap，如果是第一次get，由于我们没有set值，而是重写了initialValue方法，会new SimpleDateFormat("yyyy-MM-dd")；这样以后那个线程里面就有一个ThreadLocalMap了，并且里面保存了一个SimpleDateFormat的引用。这样就实现了线程安全，每个线程创建自己的对象（SimpleDateFormat）来用，互不干扰！

那就会有人问，我直接new一个对象出来可以吗？当然也可以！用ThreadLocal会更方便，而且节省内存。

4. 请介绍synchronized是什么，怎么使用？

synchronized是Java中的关键字，是一种同步锁。例如线程A运行到这个方法时，都要检查有没有其它线程正在用这个方法(或者该类的其他同步方法)，有的话要等正在使用synchronized方法的线程运行完这个方法后再运行此线程A，没有的话，锁定调用者，然后直接运行。它包括两种用法：synchronized方法和synchronized块。用法如下

**修饰一个代码块**

被修饰的代码块称为同步语句块，其作用的范围是大括号{}括起来的代码，作用的对象是调用这个代码块的对象；一个线程访问一个对象中的synchronized(this)同步代码块时，其他试图访问该对象的线程将被阻塞。

**修饰一个方法**

被修饰的方法称为同步方法，其作用的范围是整个方法，作用的对象是调用这个方法的对象，就是在方法的前面加synchronized，public synchronized void method(){//todo}; 这时，线程获得的是成员锁，即一次只能有一个线程进入该方法，其他线程要想在此时调用该方法，只能排队等候，当前线程(就是在synchronized方法内部的线程)执行完该方法后，别的线程才能进入。

**但在用synchronized修饰方法时要注意以下几点：**

1.synchronized关键字不能继承。

虽然可以使用synchronized来定义方法，但synchronized并不属于方法定义的一部分，因此，synchronized关键字不能被继承。如果子类继承父类，必须必须显式地在子类的这个方法中加上synchronized关键字才可以让子类的方法同步，否则不同步。

2.在定义接口方法时不能使用synchronized关键字。

3.构造方法不能使用synchronized关键字，但可以使用synchronized代码块来进行同步。

**修改一个静态的方法**

其作用的范围是整个静态方法，作用的对象是这个类的所有对象；我们知道静态方法是属于类的而不属于对象的。同样的，synchronized修饰的静态方法锁定的是这个类的所有对象。

**修饰一个类**

其作用的范围是synchronized后面括号括起来的部分synchronized(SyncThread.class)，作用的对象是这个类的所有对象

**总结：**

A. 无论synchronized关键字加在方法上还是对象上，如果它作用的对象是非静态的，则它取得的锁是对象；如果synchronized作用的对象是一个静态方法或一个类，则它取得的锁是针对类，该类所有的对象同一把锁。

B.每个对象只有一个锁（lock）与之相关联，谁拿到这个锁谁就可以运行它所控制的那段代码。

C.实现同步是要很大的系统开销作为代价的，甚至可能造成死锁，所以尽量避免无谓的同步控制。

5.请介绍Synchronized的缺陷是什么？

它有以下几个缺陷：1、无法控制阻塞时长；2、阻塞不可中断；3、不能够跨越多个对象；4、当一个线程获取了对应的锁，并执行该代码块的时候，其他的线程便只能一直等待，等待获取锁的线程释放锁。

6.请简介Lock锁是什么？

Synchronized是个关键字，属于java的特性，用以修饰方法或者类等，当线程获取到锁的时候，没有办法主动释放锁，释放锁只有两种方式：

1、是线程结束；

2、就是抛出异常，常见的方式就是昨天的interrupt exception但是牢记，异常抛出后，线程就停止了。

Lock同步锁。从JDK1.5版本开始，Java提供了一种新的线程同步的机制，即通过显示定义同步锁对象来实现同步。在这种机制下，同步锁应该使用Lock对象充当。通常认为Lock提供了比Synchronized()方法和Synchronized代码块更广泛的锁定操作，Lock实现允许更灵活的结构，可以具有差别很大的属性，并且可以支持多个相关的Condition对象。

Lock是控制多个线程对共享资源进行访问的工具。通常，所提供了对共享资源的独占访问，每次只能有一个线程对Lock对象加锁，线程开始访问共享资源之前应先获得Lock对象。不过，某些锁可能允许对共享资源并发访问，如ReadWriteLock(读写锁)。在实现线程安全的控制中，通常喜欢使用ReentrantLock(可重入锁)。

**在Lock锁这个接口类中有以下几个方法：**

**lock()方法**

平常使用得最多的一个方法，就是用来获取锁。如果锁已被其他线程获取，则进行等待。如果采用Lock，必须主动去释放锁，但是在发生异常时，不会自动释放锁，因此使用Lock必须在try{}catch{}块中进行，并且将释放锁（lock.unlock();*//释放锁）*的操作放在finally块中进行，以保证锁一定被被释放，防止死锁的发生。

**tryLock()**

方法是有返回值的，它表示用来尝试获取锁，如果获取成功，则返回true，如果获取失败（即锁已被其他线程获取），则返回false，也就说这个方法无论如何都会立即返回。在拿不到锁时不会一直在那等待。

**tryLock(long time, TimeUnit unit)**

这个方法和tryLock()方法是类似的，但是线程会在参数规定的时间内尝试获取锁,，在时间期限之内如果还拿不到锁，就返回false。如果如果一开始拿到锁或者在等待期间内拿到了锁，则返回true，

**lockInterruptibly()**

这个方法比较特殊，当通过这个方法去获取锁时，如果线程正在等待获取锁，则这个线程能够响应中断，即中断线程的等待状态。也就使说，当两个线程同时通过lock.lockInterruptibly()想获取某个锁时，假若此时线程A获取到了锁，而线程B只有在等待，那么对线程B调用threadB.interrupt()方法能够中断线程B的等待过程。

由于lockInterruptibly()的声明中抛出了异常，所以lock.lockInterruptibly()必须放在try块中或者在调用lockInterruptibly()的方法外声明抛出InterruptedException。

·      **ReentrantLock（可重入锁）**

ReentrantLock是**唯一**实现了Lock接口的类，并且ReentrantLock提供了更多的方法。

学习可重入锁，在学习它之前我们先来简介一下自旋锁，什么是自旋锁呢？自旋锁是指当一个线程在获取锁的时候，如果锁已经被其它线程获取，那么该线程将循环等待，然后不断的判断锁是否能够被成功获取，直到获取到锁才会退出循环。那我们试想，当一个方法获取到锁，但是并没有释放，然后在方法体中调用另一个方法，该方法也尝试获取锁，这样会导致什么？**死锁！**可重入锁的出现，很大程度上解决了由于等待锁而出现的死锁问题。

·      **ReadWriteLock**

ReadWriteLock也是一个接口，在它里面只定义了两个方法：一个用来获取读锁，一个用来获取写锁。也就是说将文件的读写操作分开，分成2个锁来分配给线程，从而使得多个线程可以同时进行读操作。

ReentrantReadWriteLock继承了ReadWriteLock接口。

·      **ReentrantReadWriteLock**

ReentrantReadWriteLock里面提供了很多的方法，不过最主要的有两个方法：readLock()和writeLock()用来获取读锁和写锁。

7.请介绍Volatile关键字是什么？以及与Synchronized的区别？

**一、三大性质简介**

在并发编程中，我们通常会遇到以下三个问题：原子性问题，可见性问题，有序性问题。

·      **原子性**

原子是世界上的最小单位，具有不可分割性。即一个操作或者多个操作要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。例如i=0;（i非long和double类型）这个操作是不可分割的，那么我们说这个操作是原子操作。但是例如i++;这个操作实际是i=i+1是可分割的，所以它不是一个原子操作。

非原子操作都会存在线程安全问题，所以需要我们使用同步技术(synchronized)来让它变成一个原子操作。一个操作是原子操作，那么我们称它具有原子性。java的concurrent包下提供了一些原子类，我们可以通过阅读API来了解这些原子类的用法，例如AtomicInteger、AtomicLong、AtomicReference等。在Java中synchronized和在lock、unlock中操作保证原子性。

·      **可见性**

可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的结果(值)。比如用volatile修饰的变量不允许线程内部缓存和重排序，即直接修改内存，所以对其他线程是可见的。

在Java中volatile、synchronized和final实现可见性。因为我们无法确保执行读操作的线程能适时地看到其他线程写入的值，有时甚至是根本不可能的事情。为了确保多个线程之间对内存写入操作的可见性，必须使用同步机制。

·      **有序性**

有序性：即程序执行的顺序按照代码的先后顺序执行。

Java语言提供了volatile和synchronized两个关键字来保证线程之间操作的有序性，volatile是因为其本身包含“禁止指令重排序”的语义，synchronized是由“一个变量在同一个时刻只允许一条线程对其进行lock操作”这条规则来保证的，此规则决定了持有同一个对象锁的两个同步块只能串行执行。什么是指令重排序？是指CPU采用了允许将多条指令不按程序规定的顺序分开发送给各相应电路单元处理。

**总结：**要想并发程序正确地执行，必须要保证原子性、可见性以及有序性。只要前往1有一个没有被保证，就有可能会导致程序运行不正确。

**二、volatile关键字**

先来说说volatile的特点：volatile是用来修饰变量的，一旦变量被其修饰，就保证了该变量的可见性。这时候我们就会提出疑问，变量不是有自己的声明周期嘛，为什么当一个线程执行结束，它依旧是对其他线程可见的？我们先来看看volatile这个单词的意义：不稳定的。当变量被它修饰后，它就告诉了jvm：“我很不稳定哟“，所以就将它放入了主存中，这样就实现了变量共享，当然synchronized也是可以把变量刷新到主存中，实现变量共享的。

**强调下：**当一个变量定义为 volatile 之后将具有三个特点。

1、**保证此变量对所有的线程的可见性，**这里的“可见性”，如本文开头所述，当一个线程修改了这个变量的值，volatile保证了新值能立即同步到主内存，以及每次使用前立即从主内存刷新。

2、**禁止指令重排序优化。**有volatile修饰的变量，赋值后多执行了一个“load addl $0x0,(%esp)”操作，这个操作相当于一个内存屏障（指令重排序时不能把后面的指令重排序到内存屏障之前的位置），只有一个CPU访问内存时，并不需要内存屏障。

3、**不能保证其原子性。**

**三、volatile和synchronized的区别**

1、volatile是用来修饰变量的，而synchronized是用来修饰方法和类，不能修饰变量！在它修饰的方法或者代码块中的变量只有当前线程可以访问，其他线程都会被阻塞。

2、volatile可以保证变量修改的可见性，但是无法保证线程的原子性。synchronized既可以保证变量修改的可见性，也可以保证线程的原子性。

3、volatile不会造成线程阻塞，但synchronized却会，昨天那个自旋锁的例子充分说明。

4、volatile标记的变量不会被编译器优化，而synchronized标记的变量可以被编译器优化。

7.请介绍什么是线程池？为什么要使用线程池以及几种常见的线程池？

**一、什么是线程池？**

线程池是一种多线程处理形式，处理过程中将任务添加到队列，然后在创建线程后自动启动这些任务。java.util.concurrent.Executors提供了一个java.util.concurrent.Executor接口的实现用于创建线程池。线程池就是一个容纳多个线程的容器，它会提前创建若干个线程，如果有任务需要处理，线程池里的线程就会处理任务，处理完之后线程并不会被销毁，而是等待下一个任务，反复使用。由于创建和销毁线程都是消耗系统资源的，所以当你想要频繁的创建和销毁线程的时候就可以考虑使用线程池来节约资源提升系统的性能，就例如管道连接池这种、银行柜员池不断提供服务等等。

**二、为什么要使用线程池？**

为什么要使用线程池呢？因为多线程运行时间，系统不断的启动和关闭新线程，成本非常高，过渡切换线程从而可能导致系统资源的崩溃。这时线程池就是最好的选择了。相比线程池，我们直接创建多个线程对象，首先会浪费资源，每个线程运行结束，就会创建新的线程对象，很难达到对象复用的效果；其次创建的线程对象缺乏统一的管理，很可能会出现死锁等一些导致线程卡死的问题，严重会导致死机；最后是线程对象缺乏智能化的管理，如定时等等一些控制线程的功能。

线程池主要用来解决线程生命周期开销问题和资源不足问题。线程池则可以将线程对象重复利用，当一个线程生命周期结束的时候，并没有通过创建新的线程对象的方式来创建新的线程，而是使用已结束的线程对象，这样就大大减小了系统创建对象的开销，提高了多线程的执行效率；由于是在一个管理线程的池子中，我们可以控制线程的并发数，这样做可以减少阻塞情况的出现，其次可以更好的利用计算机的性能，在其能力范围内，进行多线程的工作；

8.请介绍在java程序中怎么保证多线程的线程安全？

**一、线程安全的概念**

线程安全是多线程编程时产生的一个问题，是指在拥有共享数据的多条线程并行执行的程序中，线程安全的代码会通过同步机制保证各个线程都可以正常且正确的执行，不会出现数据污染等意外情况，例如死锁和脏数据就是典型的线程安全问题。

线程安全的问题实际上只会出现在线程与线程之间有共享模块的时候，所有线程都需要对其进行修改，如果线程之间毫不相关，那肯定不会出现线程安全的问题，举个生活中的例子：银行的存取钱。

**二、线程安全的划分**

在你心里，线程安全和不安全是如何划分的呢？由于线程安全最关键在于共享的数据，从这个基础上可以将其划分为：不可变、绝对线程安全、相对线程安全、线程兼容和线程对立，接下来我们来分别聊聊它们。

·             **不可变**

不可变（Immutable）的对象一定是线程安全的，为什么呢？因为Java语言中，如果共享数据是一个基本数据类型，那么只要在定义时使用final关键字修饰它就可以保证它是不可变的。或者加入共享数据是一个对象，那就需要保证对象的行为不会对其状态产生任何影响才行，例如java.lang.String类的对象，它也是一个典型的不可变对象。

不可变的对象无论是对象的方法实现还是方法的调用者，都不需要再采取任何的线程安全保障措施。

·             **绝对线程安全**

Brian Goetz的《Java并发编程实战》对线程安全是有定义的，原文为：

“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进行，并且在主调代码中不需要任何额外的同步或协同，这个类都能表现出正确的行为，那么称这个类是线程安全的。”

简言之，就是“不管运行时环境如何，调用者都不需要任何额外的同步措施”，这样我们可以想到，很多逻辑方法需要在类中实现，明显减小了类的可扩展性，增加了它的局限性。

在Java API中标注自己是线程安全的类，大多数都不是绝对的线程安全，即使它所有的方法都被修饰成同步，也不意味着调用它的时候永远都不再需要同步手段了。

·             **相对线程安全**

相对的线程安全就是我们通常意义上所讲的线程安全，扩展性略强，它需要保证对这个对象单独的操作是线程安全的，我们在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，就可能需要在调用端使用额外的同步手段来保证调用的正确性，例如使用synchorized关键字。

在Java语言中，大部分的线程安全类都属于这种类型，例如Vector、HashTable、Collections的synchronizedCollection()方法包装的集合等。

·             **线程兼容**

线程兼容是指对象本身就是线程不安全的，我们可以使用关键字或者加锁等同步手段，使用一些方法逻辑来保证其线程安全。Java API中大部分的类都是属于线程兼容的，如与前面的Vector和HashTable相对应的集合类ArrayList和HashMap等。

·             **线程对立**

线程对立是指无论怎么样，使用何种方式都无法保证线程安全，都会有死锁等危险出现的可能性的代码，这样的程序是无法使用在多线程中的，一般都会废弃掉例如suspend()和resume()方法。

**三、如何保证线程安全**

并发的三个原则：原子性，可见性，有序性。我认为，要保证线程安全，就要在共享模块做手脚，如果共享模块是一段完整的程序，我们就要保证其原子性，例如通过锁或其他方式来保证，我们要灵活使用我们学到的关键字和各类锁，在保证共享模块原子性的同时还要避免死锁的发生，例如一个带锁的方法线程调用另外一个带锁的方法线程，方法一会等待方法二，方法二又会等待方法一释放锁，这就进入了自旋锁的冏境，造成死锁的出现，这是用使用可重入锁就可以完美的解决问题。保证线程安全就要灵活的使用我们学习到的各种用以保证线程安全的关键字和类，这样就能写出线程安全的多线程代码！

 创建线程有几种不同的方式？

Java多线程实现方式主要有四种，又分为两类：

* 线程执行完后不返回结果：a. 继承Thread类；b. 实现Runnable接口
* 返回结果：c. 实现Callable接口通过FutureTask包装器来创建Thread线程；d. 使用ExecutorService、Callable、Future实现有返回结果的线程

**继承Thread类创建线程**

通过继承Thread类的方法，来创建并启动多线程的步骤如下：

①定义Thread类的子类MyThread，并重写该类的run()方法，该方法的方法体定义的是线程需要完成的任务，run()方法也称为线程执行体；

②创建Thread子类的实例，也就是创建了线程对象；

③通过调用线程的start()方法启动线程。

**实现Runnable接口创建线程**

通过实现Runnable接口的方法，创建并启动线程步骤如下：

①定义Runnable接口的实现类，同样要重写run()方法这是线程的执行体，Thread中的run()方法相同。

②创建Runnable实现类的实例，并用这个实例作为Thread的target来创建Thread对象，这个Thread对象才是真正的线程对象。

③通过调用线程对象的start()方法来启动线程。

**Thread与Runnable的区别：**

1.Runnable避免了单继承局限，可以同时实现多了接口，Thread是Runnable接口的子类。

2.使用Runnabe实现多继承可以更好地描述出程序共享的概念。

3.Runnable采用了代理设计模式，（在静态代理模式中有两个角色一个真实角色用来实现真正的业务操作，另一个代理角色用来完成资源的调动，辅助真实业务的完成）

**实现Callable接口通过FutureTask包装器来创建Thread线程**

通过实现Callable接口和FutureTask实现的方法，创建并启动线程步骤如下(注意：我把步骤合并了详细请看代码步骤)

①创建一个类MyCallble并实现Callable接口，然后重写call方法，必须返回数据，要不然报错！call()方法比run()方法功能要强大。

②创建MyCallable实现类的对象；创建FutureTask对象，并将myCallable实现类的对象作为参数传递到FutureTask的构造器中。

③创建Thread对象并将FutureTask对象作为参数传递Thread的构造器中。

④调用Thread启动的start()方法；调用FutureTask中的get()方法获取返回值。

**使用ExecutorService、Callable、Future实现有返回结果的线程**

1.public static ExecutorService newFixedThreadPool(int nThreads){...}

ExecutorService fixedThreadPool = Executors.newFixedThreadPool(2);

fixedThreadPool.submit(**Thread**);

创建定长线程池，就是控制了线程池的大小，简单点的说就是控制了线程的数量，

2.    public static ExecutorService newCachedThreadPool(){...}

ExecutorService cachedThreadPool = Executors.newCachedThreadPool();

创建一个可缓存的线程池，调用execute将重用以前构造的可用的线程。如果现有线程没有可用的，则创建一个新线程并添加到池中，终止时并从缓存中移除那些已有60秒钟未被使用的线程，参数中不做限制处理，限制是由jvm以及计算机的处理性能决定的。

**3.**   **public static ExecutorService newSingleThreadExecutor(){...}**

ExecutorService ThreadPool=Executors.newSingleThreadExecutor();

创建一个单线程化的Executor，限制线程池大小为1，为了保证线程按照顺序进行

      newSingleThreadExecutor相比newFixedThreadPool对线程数做了限制

**4.public     static   ScheduledExecutorService newScheduledThreadPool(int corePoolSize) {...}**

它和第一个线程不同的地方在于可以定时和限制运行周期，多数情况下可用来替代Timer类；实际上就是根据两个参数实现的

**ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(3);**

//参数分别为 线程，延迟开始时间（定时），线程运行时间间隔（周期），时间单位说明前面两个参数单位为秒）

scheduledThreadPool.scheduleAtFixedRate(new ThreadForTest(), 2, 1, TimeUnit.SECONDS);

*//定时，其实就是缺省了周期参数，比较上述*

*//scheduledThreadPool.schedule(new ThreadForTest(), 1, TimeUnit.SECONDS);*

2.请介绍线程有哪些状态？

**1.NEW：**初始状态，线程被创建后还没有启动，这时候通过线程的getState方法可以获得其状态为NEW。

**2.RUNNABLE：**运行状态，该状态分为就绪状态和运行中状态，运行中状态自然就是指线程正在运行，就绪状态是指线程已经做好准备且不处于排队状态，也就是说，已经调用了start()函数，其后一些工作结束后(如等待时间片分配、等待锁或者说从sleep()状态中唤醒)。

**3.TERMINATED：**终止状态，在线程的run()方法运行结束后，线程就会处于死亡状态。

**4.BLOCKED：**阻塞状态，在进入synchronized关键字修饰的方法或代码块(获取锁)时的状态。

**5. WAITTING：**等待状态，如果该线程的操作是由其他线程进行控制的，则没有收到运行通知，该线程会一直处于等待状态，直到收到通知才会进行后续操作。

**6. TIMED\_WAITTING：**超时等待状态，与WAITTING类似，但是在规定时间内没有接收到运行通知，会自己返回，回到初始状态

3.请谈谈你对ThreadLocal的理解？

ThreadLocal是什么？ ThreadLocal并不是一个Thread，而是Thread的局部变量，很多地方叫做线程本地变量，也有些地方叫做线程本地存储，其实意思差不多。ThreadLocal为变量在每个线程中都创建了一个副本，那么每个线程可以访问自己内部的副本变量。

它采用空间来换取时间的方式，解决多线程中相同变量的访问冲突问题。

ThreadLocal不是用来解决多线程下访问共享变量问题的。

实际是在Thread类里面有一个ThreadLocalMap，用于存储每一个线程的变量的引用，这个Map中的键为ThreadLocal对象，而值对应的是ThreadLocal通过set放进去的变量引用。(这句重点！！)

ThreadLocal通过set(共享变量)然后再通过ThreadLocal方法get的是共享变量的引用！假设有多个线程都在其执行过程中将共享变量加入到自己的ThreadLocal中，那就是每个线程都持有一份共享变量的引用副本，注意是引用副本，共享变量的实例只有一个。所以，ThreadLocal不是用来解决线程间共享变量的访问的事儿的。

总结：

ThreadLocal在每个线程中对该变量会创建一个副本，即每个线程内部都会有一个该变量，且在线程内部任何地方都可以使用，线程之间互不影响，这样一来就不存在线程安全问题，也不会严重影响程序执行性能。

请介绍ThreadLocal的应用场景是什么，请举几个例子？

**1数据库连,2.Session管理,3.让SimpleDateFomat变成线程安全**

在多线程中，对于一个线程来说df.get的时候，首先尝试获得线程它自己的ThreadLocalMap，如果是第一次get，由于我们没有set值，而是重写了initialValue方法，会new SimpleDateFormat("yyyy-MM-dd")；这样以后那个线程里面就有一个ThreadLocalMap了，并且里面保存了一个SimpleDateFormat的引用。这样就实现了线程安全，每个线程创建自己的对象（SimpleDateFormat）来用，互不干扰！

那就会有人问，我直接new一个对象出来可以吗？当然也可以！用ThreadLocal会更方便，而且节省内存。

4. 请介绍synchronized是什么，怎么使用？

synchronized是Java中的关键字，是一种同步锁。例如线程A运行到这个方法时，都要检查有没有其它线程正在用这个方法(或者该类的其他同步方法)，有的话要等正在使用synchronized方法的线程运行完这个方法后再运行此线程A，没有的话，锁定调用者，然后直接运行。它包括两种用法：synchronized方法和synchronized块。用法如下

**修饰一个代码块**

被修饰的代码块称为同步语句块，其作用的范围是大括号{}括起来的代码，作用的对象是调用这个代码块的对象；一个线程访问一个对象中的synchronized(this)同步代码块时，其他试图访问该对象的线程将被阻塞。

**修饰一个方法**

被修饰的方法称为同步方法，其作用的范围是整个方法，作用的对象是调用这个方法的对象，就是在方法的前面加synchronized，public synchronized void method(){//todo}; 这时，线程获得的是成员锁，即一次只能有一个线程进入该方法，其他线程要想在此时调用该方法，只能排队等候，当前线程(就是在synchronized方法内部的线程)执行完该方法后，别的线程才能进入。

**但在用synchronized修饰方法时要注意以下几点：**

1.synchronized关键字不能继承。

虽然可以使用synchronized来定义方法，但synchronized并不属于方法定义的一部分，因此，synchronized关键字不能被继承。如果子类继承父类，必须必须显式地在子类的这个方法中加上synchronized关键字才可以让子类的方法同步，否则不同步。

2.在定义接口方法时不能使用synchronized关键字。

3.构造方法不能使用synchronized关键字，但可以使用synchronized代码块来进行同步。

**修改一个静态的方法**

其作用的范围是整个静态方法，作用的对象是这个类的所有对象；我们知道静态方法是属于类的而不属于对象的。同样的，synchronized修饰的静态方法锁定的是这个类的所有对象。

**修饰一个类**

其作用的范围是synchronized后面括号括起来的部分synchronized(SyncThread.class)，作用的对象是这个类的所有对象

**总结：**

A. 无论synchronized关键字加在方法上还是对象上，如果它作用的对象是非静态的，则它取得的锁是对象；如果synchronized作用的对象是一个静态方法或一个类，则它取得的锁是针对类，该类所有的对象同一把锁。

B.每个对象只有一个锁（lock）与之相关联，谁拿到这个锁谁就可以运行它所控制的那段代码。

C.实现同步是要很大的系统开销作为代价的，甚至可能造成死锁，所以尽量避免无谓的同步控制。

synchronized锁升级过程：

首先，我们要知道一个知识点，就是synchronized的实现是通过monitorenter/monitorexit 指令实现的。

每个对象都有Monitor（监视器锁），当monitor被占用的时候，对象进入锁定状态。

monitorenter：线程通过monitorenter指令尝试获取monitor所有权，情况如下：

1、如果monitor的进入数为0，则该线程进入，进入数设置为1，线程占有monitor

2、如果该线程已经占有了monitor，只是重新进入，则进入数+1

3、如果已经有别的形成占有了monitor，则该线程进入阻塞状态，知道monitor的进入数为0，再重新尝试获取monitor

monitorexit：

　　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。

　　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。

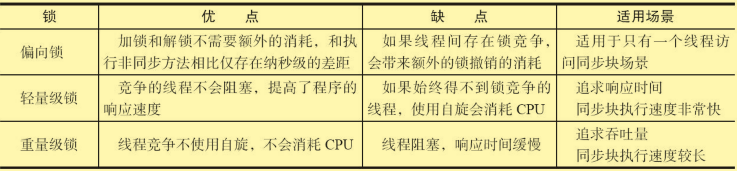
在java6以前，monitor的实现完全依靠操作系统内部的互斥锁（因为需要在user space和kernel space间切换），所以这时期的同步操作时一个无差别的重量级操作，也就没有锁的升级降级概念。

synchronized所使用的锁是存放在java的对象头中的,如果对象是数组类型,虚拟机使用3个字宽存储对象头,如果对象是非数组类型,则使用2字宽来存储对象头

首先要了解偏向锁和轻量级锁,偏向锁就是在不存在大量线程竞争的情况下,锁不存在多线程竞争,每次都有同一个线程多次获得,这个时候为了让获得锁的代价更低廉,就是用了偏向锁

当一个线程获取了锁之后,就会在对象头和栈帧中的锁记录里面存储偏向的锁id,以后线程再次进入,只需要测试对象头的Mark Word是否存储着当前线程的偏向锁即可,如果测试失败,则查看是否设置了偏向锁的标识,如果设置了,则进行锁升级使用CAS自旋锁进行锁竞争,这时锁已经升级为了轻量级锁,将对象头的偏向锁指向当前线程

当偏向锁升级为轻量级锁之后(CAS) 如果成功将对象头替换回去,则表示没有竞争发生,由轻量级锁退化为偏向锁,如果替换失败,则说明有连个以上的线程进行锁的竞争,将其变为重量级锁



5.请介绍Synchronized的缺陷是什么？

它有以下几个缺陷：1、无法控制阻塞时长；2、阻塞不可中断；3、不能够跨越多个对象；4、当一个线程获取了对应的锁，并执行该代码块的时候，其他的线程便只能一直等待，等待获取锁的线程释放锁。

6.请简介Lock锁是什么？

Synchronized是个关键字，属于java的特性，用以修饰方法或者类等，当线程获取到锁的时候，没有办法主动释放锁，释放锁只有两种方式：

1、是线程结束；

2、就是抛出异常，常见的方式就是昨天的interrupt exception但是牢记，异常抛出后，线程就停止了。

Lock同步锁。从JDK1.5版本开始，Java提供了一种新的线程同步的机制，即通过显示定义同步锁对象来实现同步。在这种机制下，同步锁应该使用Lock对象充当。通常认为Lock提供了比Synchronized()方法和Synchronized代码块更广泛的锁定操作，Lock实现允许更灵活的结构，可以具有差别很大的属性，并且可以支持多个相关的Condition对象。

Lock是控制多个线程对共享资源进行访问的工具。通常，所提供了对共享资源的独占访问，每次只能有一个线程对Lock对象加锁，线程开始访问共享资源之前应先获得Lock对象。不过，某些锁可能允许对共享资源并发访问，如ReadWriteLock(读写锁)。在实现线程安全的控制中，通常喜欢使用ReentrantLock(可重入锁)。

**在Lock锁这个接口类中有以下几个方法：**

**lock()方法**

平常使用得最多的一个方法，就是用来获取锁。如果锁已被其他线程获取，则进行等待。如果采用Lock，必须主动去释放锁，但是在发生异常时，不会自动释放锁，因此使用Lock必须在try{}catch{}块中进行，并且将释放锁（lock.unlock();*//释放锁）*的操作放在finally块中进行，以保证锁一定被被释放，防止死锁的发生。

**tryLock()**

方法是有返回值的，它表示用来尝试获取锁，如果获取成功，则返回true，如果获取失败（即锁已被其他线程获取），则返回false，也就说这个方法无论如何都会立即返回。在拿不到锁时不会一直在那等待。

**tryLock(long time, TimeUnit unit)**

这个方法和tryLock()方法是类似的，但是线程会在参数规定的时间内尝试获取锁,，在时间期限之内如果还拿不到锁，就返回false。如果如果一开始拿到锁或者在等待期间内拿到了锁，则返回true，

**lockInterruptibly()**

这个方法比较特殊，当通过这个方法去获取锁时，如果线程正在等待获取锁，则这个线程能够响应中断，即中断线程的等待状态。也就使说，当两个线程同时通过lock.lockInterruptibly()想获取某个锁时，假若此时线程A获取到了锁，而线程B只有在等待，那么对线程B调用threadB.interrupt()方法能够中断线程B的等待过程。

由于lockInterruptibly()的声明中抛出了异常，所以lock.lockInterruptibly()必须放在try块中或者在调用lockInterruptibly()的方法外声明抛出InterruptedException。

·      **ReentrantLock（可重入锁）**

ReentrantLock是**唯一**实现了Lock接口的类，并且ReentrantLock提供了更多的方法。

学习可重入锁，在学习它之前我们先来简介一下自旋锁，什么是自旋锁呢？自旋锁是指当一个线程在获取锁的时候，如果锁已经被其它线程获取，那么该线程将循环等待，然后不断的判断锁是否能够被成功获取，直到获取到锁才会退出循环。那我们试想，当一个方法获取到锁，但是并没有释放，然后在方法体中调用另一个方法，该方法也尝试获取锁，这样会导致什么？**死锁！**可重入锁的出现，很大程度上解决了由于等待锁而出现的死锁问题。

·      **ReadWriteLock**

ReadWriteLock也是一个接口，在它里面只定义了两个方法：一个用来获取读锁，一个用来获取写锁。也就是说将文件的读写操作分开，分成2个锁来分配给线程，从而使得多个线程可以同时进行读操作。

ReentrantReadWriteLock继承了ReadWriteLock接口。

·      **ReentrantReadWriteLock**

ReentrantReadWriteLock里面提供了很多的方法，不过最主要的有两个方法：readLock()和writeLock()用来获取读锁和写锁。

7.请介绍Volatile关键字是什么？以及与Synchronized的区别？

**一、三大性质简介**

在并发编程中，我们通常会遇到以下三个问题：原子性问题，可见性问题，有序性问题。

·      **原子性**

原子是世界上的最小单位，具有不可分割性。即一个操作或者多个操作要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。例如i=0;（i非long和double类型）这个操作是不可分割的，那么我们说这个操作是原子操作。但是例如i++;这个操作实际是i=i+1是可分割的，所以它不是一个原子操作。

非原子操作都会存在线程安全问题，所以需要我们使用同步技术(synchronized)来让它变成一个原子操作。一个操作是原子操作，那么我们称它具有原子性。java的concurrent包下提供了一些原子类，我们可以通过阅读API来了解这些原子类的用法，例如AtomicInteger、AtomicLong、AtomicReference等。在Java中synchronized和在lock、unlock中操作保证原子性。

·      **可见性**

可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的结果(值)。比如用volatile修饰的变量不允许线程内部缓存和重排序，即直接修改内存，所以对其他线程是可见的。

在Java中volatile、synchronized和final实现可见性。因为我们无法确保执行读操作的线程能适时地看到其他线程写入的值，有时甚至是根本不可能的事情。为了确保多个线程之间对内存写入操作的可见性，必须使用同步机制。

·      **有序性**

有序性：即程序执行的顺序按照代码的先后顺序执行。

Java语言提供了volatile和synchronized两个关键字来保证线程之间操作的有序性，volatile是因为其本身包含“禁止指令重排序”的语义，synchronized是由“一个变量在同一个时刻只允许一条线程对其进行lock操作”这条规则来保证的，此规则决定了持有同一个对象锁的两个同步块只能串行执行。什么是指令重排序？是指CPU采用了允许将多条指令不按程序规定的顺序分开发送给各相应电路单元处理。

**总结：**要想并发程序正确地执行，必须要保证原子性、可见性以及有序性。只要前往1有一个没有被保证，就有可能会导致程序运行不正确。

**二、volatile关键字**

Java编程语言允许线程访问共享变量，为了确保共享变量能被准确和一致地更新，线程应该确保通过**排他锁**单独获得这个变量。Java语言提供了**volatile**，在某些情况下比锁要更加方便。如果一个字段被声明成volatile，Java线程内存模型确保所有线程看到这个变量的值是一致的。

**有volatile修饰的变量在转换为汇编语言时，前面会有一个lock指令。Lock前缀的指令在多核处理器下会做两件事情：**

1. **将当前处理器缓存行的数据写回到系统内存。**
2. **这个写回内存的操作会使在其他CPU里缓存了该内存地址的数据无效。**

**第一种解释**：为了提高处理速度，处理器不直接和内存进行，处理器先将数据从内存中读取到内部缓存中再进行操作。如果对一个已经被volatile声明的变量进行写操作，**jvm就会向处理器发送一条Lock前缀的指令，将这个变量所在缓存行的数据写回到系统内存，**但是，就算写回到内存，如果其他处理器缓存的值还是旧的，再执行计算操作就会有问题。所以，在多处理器下，为了保证各个处理器的缓存是一致的，就会实现缓存一致性协议，每个处理器通过**嗅探在总线上传播的数据**来检查自己缓存的值是不是过期了，当处理器发现自己缓存行对应的内存地址被修改，就会将当前处理器的缓存行设置成无效状态，当处理器对这个数据进行修改操作的时候，会重新从系统内存中把数据读到处理器缓存里。

**第二种解释：**每个线程都拷贝了一份count变量到自己的内存区域，在多线程情况下，如果某一个线程要对volatile 变量修改，首先运行该线程的cpu会锁定方法区变量内存区域，保证做到数据修改的原子性；那现在问题就来了，其他线程是怎么获得最新的数据？运行其他线程的cpu有会嗅探总线上该内存区域的地址是否被修改；如果该线程缓存的变量副本所指向方法区的地址发生了变化，则将自己缓存的数据设置成无效，等待线程使用到该变量的时候，会从新从方法区获得一份新的数据，同步到本地线程的缓存区域，这样就volatile变量就做到了多线程的可见以及一致性

**强调下：**当一个变量定义为 volatile 之后将具有三个特点。

1、**保证此变量对所有的线程的可见性，**这里的“可见性”，如本文开头所述，当一个线程修改了这个变量的值，volatile保证了新值能立即同步到主内存，以及每次使用前立即从主内存刷新。

2、**禁止指令重排序优化。**有volatile修饰的变量，赋值后多执行了一个“load addl $0x0,(%esp)”操作，这个操作相当于一个内存屏障（指令重排序时不能把后面的指令重排序到内存屏障之前的位置），只有一个CPU访问内存时，并不需要内存屏障。

3、**不能保证其原子性。**

**三、volatile和synchronized的区别**

1、volatile是用来修饰变量的，而synchronized是用来修饰方法和类，不能修饰变量！在它修饰的方法或者代码块中的变量只有当前线程可以访问，其他线程都会被阻塞。

2、volatile可以保证变量修改的可见性，但是无法保证线程的原子性。synchronized既可以保证变量修改的可见性，也可以保证线程的原子性。

3、volatile不会造成线程阻塞，但synchronized却会，昨天那个自旋锁的例子充分说明。

4、volatile标记的变量不会被编译器优化，而synchronized标记的变量可以被编译器优化。

7.请介绍什么是线程池？为什么要使用线程池以及几种常见的线程池？

**一、什么是线程池？**

线程池是一种多线程处理形式，处理过程中将任务添加到队列，然后在创建线程后自动启动这些任务。java.util.concurrent.Executors提供了一个java.util.concurrent.Executor接口的实现用于创建线程池。线程池就是一个容纳多个线程的容器，它会提前创建若干个线程，如果有任务需要处理，线程池里的线程就会处理任务，处理完之后线程并不会被销毁，而是等待下一个任务，反复使用。由于创建和销毁线程都是消耗系统资源的，所以当你想要频繁的创建和销毁线程的时候就可以考虑使用线程池来节约资源提升系统的性能，就例如管道连接池这种、银行柜员池不断提供服务等等。

**二、为什么要使用线程池？**

为什么要使用线程池呢？因为多线程运行时间，系统不断的启动和关闭新线程，成本非常高，过渡切换线程从而可能导致系统资源的崩溃。这时线程池就是最好的选择了。相比线程池，我们直接创建多个线程对象，首先会浪费资源，每个线程运行结束，就会创建新的线程对象，很难达到对象复用的效果；其次创建的线程对象缺乏统一的管理，很可能会出现死锁等一些导致线程卡死的问题，严重会导致死机；最后是线程对象缺乏智能化的管理，如定时等等一些控制线程的功能。

线程池主要用来解决线程生命周期开销问题和资源不足问题。线程池则可以将线程对象重复利用，当一个线程生命周期结束的时候，并没有通过创建新的线程对象的方式来创建新的线程，而是使用已结束的线程对象，这样就大大减小了系统创建对象的开销，提高了多线程的执行效率；由于是在一个管理线程的池子中，我们可以控制线程的并发数，这样做可以减少阻塞情况的出现，其次可以更好的利用计算机的性能，在其能力范围内，进行多线程的工作；

8.请介绍在java程序中怎么保证多线程的线程安全？

**一、线程安全的概念**

线程安全是多线程编程时产生的一个问题，是指在拥有共享数据的多条线程并行执行的程序中，线程安全的代码会通过同步机制保证各个线程都可以正常且正确的执行，不会出现数据污染等意外情况，例如死锁和脏数据就是典型的线程安全问题。

线程安全的问题实际上只会出现在线程与线程之间有共享模块的时候，所有线程都需要对其进行修改，如果线程之间毫不相关，那肯定不会出现线程安全的问题，举个生活中的例子：银行的存取钱。

**二、线程安全的划分**

在你心里，线程安全和不安全是如何划分的呢？由于线程安全最关键在于共享的数据，从这个基础上可以将其划分为：不可变、绝对线程安全、相对线程安全、线程兼容和线程对立，接下来我们来分别聊聊它们。

·             **不可变**

不可变（Immutable）的对象一定是线程安全的，为什么呢？因为Java语言中，如果共享数据是一个基本数据类型，那么只要在定义时使用final关键字修饰它就可以保证它是不可变的。或者加入共享数据是一个对象，那就需要保证对象的行为不会对其状态产生任何影响才行，例如java.lang.String类的对象，它也是一个典型的不可变对象。

不可变的对象无论是对象的方法实现还是方法的调用者，都不需要再采取任何的线程安全保障措施。

·             **绝对线程安全**

Brian Goetz的《Java并发编程实战》对线程安全是有定义的，原文为：

“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进行，并且在主调代码中不需要任何额外的同步或协同，这个类都能表现出正确的行为，那么称这个类是线程安全的。”

简言之，就是“不管运行时环境如何，调用者都不需要任何额外的同步措施”，这样我们可以想到，很多逻辑方法需要在类中实现，明显减小了类的可扩展性，增加了它的局限性。

在Java API中标注自己是线程安全的类，大多数都不是绝对的线程安全，即使它所有的方法都被修饰成同步，也不意味着调用它的时候永远都不再需要同步手段了。

·             **相对线程安全**

相对的线程安全就是我们通常意义上所讲的线程安全，扩展性略强，它需要保证对这个对象单独的操作是线程安全的，我们在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，就可能需要在调用端使用额外的同步手段来保证调用的正确性，例如使用synchorized关键字。

在Java语言中，大部分的线程安全类都属于这种类型，例如Vector、HashTable、Collections的synchronizedCollection()方法包装的集合等。

·             **线程兼容**

线程兼容是指对象本身就是线程不安全的，我们可以使用关键字或者加锁等同步手段，使用一些方法逻辑来保证其线程安全。Java API中大部分的类都是属于线程兼容的，如与前面的Vector和HashTable相对应的集合类ArrayList和HashMap等。

·             **线程对立**

线程对立是指无论怎么样，使用何种方式都无法保证线程安全，都会有死锁等危险出现的可能性的代码，这样的程序是无法使用在多线程中的，一般都会废弃掉例如suspend()和resume()方法。

**三、如何保证线程安全**

并发的三个原则：原子性，可见性，有序性。我认为，要保证线程安全，就要在共享模块做手脚，如果共享模块是一段完整的程序，我们就要保证其原子性，例如通过锁或其他方式来保证，我们要灵活使用我们学到的关键字和各类锁，在保证共享模块原子性的同时还要避免死锁的发生，例如一个带锁的方法线程调用另外一个带锁的方法线程，方法一会等待方法二，方法二又会等待方法一释放锁，这就进入了自旋锁的冏境，造成死锁的出现，这是用使用可重入锁就可以完美的解决问题。保证线程安全就要灵活的使用我们学习到的各种用以保证线程安全的关键字和类，这样就能写出线程安全的多线程代码！