# 多线程

## 为什么要使用并发编程（并发编程的优点）

充分利用多核CPU的计算能力：通过并发编程的形式可以将多核CPU的计算能力发挥到极致，性能得到提升
方便进行业务拆分，提升系统并发能力和性能：在特殊的业务场景下，先天的就适合于并发编程。现在的系统动不动就要求百万级甚至千万级的并发量，而多线程并发编程正是开发高并发系统的基础，利用好多线程机制可以大大提高系统整体的并发能力以及性能。面对复杂业务模型，并行程序会比串行程序更适应业务需求，而并发编程更能吻合这种业务拆分 。

## 并发编程有什么缺点

并发编程的目的就是为了能提高程序的执行效率，提高程序运行速度，但是并发编程并不总是能提高程序运行速度的，而且并发编程可能会遇到很多问题，比如**：内存泄漏、上下文切换、线程安全、死锁**等问题。

## 并发编程三要素是什么？在 Java 程序中怎么保证多线程的运行安全？

并发编程三要素（线程的安全性问题体现在）：
1. 原子性：原子，即一个不可再被分割的颗粒。原子性指的是一个或多个操作要么全部执行成功要么全部执行失败。
2. 可见性：一个线程对共享变量的修改,另一个线程能够立刻看到。（synchronized,volatile）
3. 有序性：程序执行的顺序按照代码的先后顺序执行。（处理器可能会对指令进行重排序）

出现线程安全问题的原因：
1. 线程切换带来的原子性问题
2. 缓存导致的可见性问题
3. 编译优化带来的有序性问题

解决办法：
1. JDK Atomic开头的原子类、synchronized、LOCK，可以解决原子性问题
2. synchronized、volatile、LOCK，可以解决可见性问题
3. Happens-Before 规则可以解决有序性问题

### 35.并行和并发有什么区别?

并发：是指多个线程任务在同一个CPU上快速地轮换执行，由于切换的速度非常快，给人的感觉就是这些线程任务是在同时进行的，但其实并发只是一种逻辑上的同时进行；
并行：是指多个线程任务在不同CPU上同时进行，是真正意义上的同时执行。
并发 = 两个队列和一台咖啡机。
并行 = 两个队列和两台咖啡机。
串行 = 一个队列和一台咖啡机。

## 什么是多线程，多线程的优劣?

多线程：多线程是指程序中包含多个执行流，即在一个程序中可以同时运行多个不同的线程来执行不同的任务。

多线程的好处：
可以提高 CPU 的利用率。在多线程程序中，一个线程必须等待的时候，CPU 可以运行其它的线程而不是等待，这样就大大提高了程序的效率。也就是说允许单个程序创建多个并行执行的线程来完成各自的任务。

多线程的劣势：
- 线程也是程序，所以线程需要占用内存，线程越多占用内存也越多；
- 多线程需要协调和管理，所以需要 CPU 时间跟踪线程；
- 线程之间对共享资源的访问会相互影响，必须解决竞用共享资源的问题。

## 36.线程和进程的区别?

- 根本区别：进程是操作系统资源分配的基本单位，而线程是处理器任务调度和执行的基本单位
- 资源开销：每个进程都有独立的代码和数据空间（程序上下文），程序之间的切换会有较大的开销；线程可以看做轻量级的进程，同一类线程共享代码和数据空间，每个线程都有自己独立的运行栈和程序计数器（PC），线程之间切换的开销小。
- 包含关系：如果一个进程内有多个线程，则执行过程不是一条线的，而是多条线（线程）共同完成的；线程是进程的一部分，所以线程也被称为轻权进程或者轻量级进程。
- 内存分配：同一进程的线程共享本进程的地址空间和资源，而进程之间的地址空间和资源是相互独立的
- 影响关系：一个进程崩溃后，在保护模式下不会对其他进程产生影响，但是一个线程崩溃整个进程都死掉。所以多进程要比多线程健壮。
- 执行过程：每个独立的进程有程序运行的入口、顺序执行序列和程序出口。但是线程不能独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制，两者均可并发执行

## 什么是上下文切换?

多线程编程中一般线程的个数都大于 CPU 核心的个数，而一个 CPU 核心在任意时刻只能被一个线程使用，为了让这些线程都能得到有效执行，CPU 采取的策略是为每个线程分配时间片并轮转的形式。当一个线程的时间片用完的时候就会重新处于就绪状态让给其他线程使用，这个过程就属于一次上下文切换。
概括来说就是：当前任务在执行完 CPU 时间片切换到另一个任务之前会先保存自己的状态，以便下次再切换回这个任务时，可以再加载这个任务的状态。任务从保存到再加载的过程就是一次上下文切换。
上下文切换通常是计算密集型的。也就是说，它需要相当可观的处理器时间，在每秒几十上百次的切换中，每次切换都需要纳秒量级的时间。所以，上下文切换对系统来说意味着消耗大量的 CPU 时间，事实上，可能是操作系统中时间消耗最大的操作。
Linux 相比与其他操作系统（包括其他类 Unix 系统）有很多的优点，其中有一项就是，其上下文切换和模式切换的时间消耗非常少。

## 37.守护线程是什么?

Java线程分为用户线程和守护线程。

守护线程是程序运行的时候在后台提供一种通用服务的线程。所有用户线程停止，进程会停掉所有守护线程，退出程序。
Java中把线程设置为守护线程的方法：在 start 线程之前调用线程的 setDaemon(true) 方法。

## 38.创建线程有哪几种方式?

创建线程有四种方式：

继承 Thread 类；
实现 Runnable 接口；
实现 Callable 接口；

使用 Executors 工具类创建线程池

继承 Thread 类
1. 定义一个Thread类的子类，重写run方法，将相关逻辑实现，run()方法就是线程要执行的业务逻辑方法
2. 创建自定义的线程子类对象
3. 调用子类实例的start()方法来启动线程

实现 Runnable 接口
1. 定义Runnable接口实现类MyRunnable，并重写run()方法
2. 创建MyRunnable实例myRunnable，以myRunnable作为target创建Thead对象，该Thread对象才是真正的线程对象
3. 调用线程对象的start()方法

实现 Callable 接口
1. 创建实现Callable接口的类myCallable
2. 以myCallable为参数创建FutureTask对象
3. 将FutureTask作为参数创建Thread对象
4. 调用线程对象的start()方法

使用 Executors 工具类创建线程池
Executors提供了一系列工厂方法用于创先线程池，返回的线程池都实现了ExecutorService接口。
主要有newFixedThreadPool，newCachedThreadPool，newSingleThreadExecutor，newScheduledThreadPool，后续详细介绍这四种线程池

## 39.说一下 runnable 和 callable 有什么区别?

相同点
- 都是接口
- 都可以编写多线程程序
- 都采用Thread.start()启动线程

主要区别
- Runnable 接口 run 方法无返回值；Callable 接口 call 方法有返回值，是个泛型，和Future、FutureTask配合可以用来获取异步执行的结果
- Runnable 接口 run 方法只能抛出运行时异常，且无法捕获处理；Callable 接口 call 方法允许抛出异常，可以获取异常信息

注：Callalbe接口支持返回执行结果，需要调用FutureTask.get()得到，此方法会阻塞主进程的继续往下执行，如果不调用不会阻塞。

## 什么是 FutureTask

FutureTask 表示一个异步运算的任务。FutureTask 里面可以传入一个 Callable 的具体实现类，可以对这个异步运算的任务的结果进行等待获取、判断是否已经完成、取消任务等操作。只有当运算完成的时候结果才能取回，如果运算尚未完成 get 方法将会阻塞。一个 FutureTask 对象可以对调用了 Callable 和 Runnable 的对象进行包装，由于 FutureTask 也是Runnable 接口的实现类，所以 FutureTask 也可以放入线程池中。

### 40.线程有哪些状态?

1.新建状态(New)：
当用new操作符创建一个线程时， 例如new Thread(r)，线程还没有开始运行，此时线程处在新建状态。当一个线程处于新生状态时，程序还没有开始运行线程中的代码。
2.就绪状态(Runnable)：

一个新创建的线程并不自动开始运行，要执行线程，必须调用线程的start()方法。当线程对象调用start()方法即启动了线程，start()方法创建线程运行的系统资源，并调度线程运行run()方法。当start()方法返回后，线程就处于就绪状态。
处于就绪状态的线程并不一定立即运行run()方法，线程还必须同其他线程竞争CPU时间，只有获得CPU时间才可以运行线程。因为在单CPU的计算机系统中，不可能同时运行多个线程，一个时刻仅有一个线程处于运行状态。因此此时可能有多个线程处于就绪状态。对多个处于就绪状态的线程是由Java运行时系统的线程调度程序(thread scheduler)来调度的。
3.运行状态(Running)
当线程获得CPU时间后，它才进入运行状态，真正开始执行run()方法.
4. 阻塞状态(Blocked)
阻塞状态是指线程因为某种原因放弃了cpu 使用权，也即让出了cpu timeslice，暂时停止运行。直到线程进入可运行(runnable)状态，才有机会再次获得cpu timeslice 转到运行(running)状态。阻塞的情况分三种：
1) 等待阻塞：运行(running)的线程执行o.wait()方法，JVM会把该线程放入等待队列(waitting queue)中。
2) 同步阻塞：运行(running)的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池(lock pool)中。
3) 其他阻塞：运行(running)的线程执行Thread.sleep(long ms)或t.join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、
join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入可运行(runnable)状态。
5. 死亡状态(Dead)
有两个原因会导致线程死亡：
1) run方法正常退出而自然死亡，
2) 一个未捕获的异常终止了run方法而使线程猝死。
为了确定线程在当前是否存活着（就是要么是可运行的，要么是被阻塞了），需要使用isAlive方法。如果是可运行或被阻塞，这个方法返回true； 如果线程仍旧是new状态且不是可运行的， 或者线程死亡了，则返回false.

### 41.sleep() 和 wait() 有什么区别?

sleep()和wait()都是线程暂停执行的方法。

1、这两个方法来自不同的类分别是Thread和Object，sleep方法属于Thread类中的静态方法，wait属于Object的成员方法。
2、sleep()是线程类（Thread）的方法，不涉及线程通信，调用时会暂停此线程指定的时间，但监控依然保持，不会释放对象锁，到时间自动恢复；wait()是Object的方法，用于线程间的通信，调用时会放弃对象锁，进入等待队列，待调用notify()/notifyAll()唤醒指定的线程或者所有线程，才进入对象锁定池准备获得对象锁进入运行状态。
3、wait，notify和notifyAll只能在同步控制方法或者同步控制块里面使用，而sleep可以在任何地方使用（使用范围）。
4、sleep()方法必须捕获异常InterruptedException，而wait()\notify()以及notifyAll()不需要捕获异常。

## 42.notify()和 notifyAll()有什么区别?

如果线程调用了对象的 wait()方法，那么线程便会处于该对象的等待池中，等待池中的线程不会去竞争该对象的锁。

notifyAll() 会唤醒所有的线程，notify() 只会唤醒一个线程。

notifyAll() 调用后，会将全部线程由等待池移到锁池，然后参与锁的竞争，竞争成功则继续执行，如果不成功则留在锁池等待锁被释放后再次参与竞争。而 notify()只会唤醒一个线程，具体唤醒哪一个线程由虚拟机控制。

## 43.线程的 run()和 start()有什么区别?

调用 start() 方法是用来启动线程的，轮到该线程执行时，会自动调用 run()；直接调用 run() 方法，无法达到启动多线程的目的，相当于主线程线性执行 Thread 对象的 run() 方法。
一个线程对线的 start() 方法只能调用一次，多次调用会抛出 java.lang.IllegalThreadStateException 异常；run() 方法没有限制。

44.创建线程池有哪几种方式?

**45.线程池都有哪些状态?**

RUNNING：线程池一旦被创建，就处于 RUNNING 状态，任务数为 0，能够接收新任务，对已排队的任务进行处理。
SHUTDOWN：不接收新任务，但能处理已排队的任务。调用线程池的 shutdown() 方法，线程池由 RUNNING 转变为 SHUTDOWN 状态。
STOP：不接收新任务，不处理已排队的任务，并且会中断正在处理的任务。调用线程池的 shutdownNow() 方法，线程池由(RUNNING 或 SHUTDOWN ) 转变为 STOP 状态。
TIDYING：
    SHUTDOWN 状态下，任务数为 0， 其他所有任务已终止，线程池会变为 TIDYING 状态，会执行 terminated() 方法。线程池中的 terminated() 方法是空实现，可以重写该方法进行相应的处理。
    线程池在 SHUTDOWN 状态，任务队列为空且执行中任务为空，线程池就会由 SHUTDOWN 转变为 TIDYING 状态。
    线程池在 STOP 状态，线程池中执行中任务为空时，就会由 STOP 转变为 TIDYING 状态。
TERMINATED：线程池彻底终止。线程池在 TIDYING 状态执行完 terminated() 方法就会由 TIDYING 转变为 TERMINATED 状态。

## 46.线程池中 submit()和 execute()方法有什么区别?

接收参数：execute()只能执行 Runnable 类型的任务。submit()可以执行 Runnable 和 Callable 类型的任务。

返回值：submit()方法可以返回持有计算结果的 Future 对象，而execute()没有

异常处理：submit()方便Exception处理

## 47.在 java 程序中怎么保证多线程的运行安全?

方法一：使用安全类，比如 java.util.concurrent 下的类，使用原子类AtomicInteger
方法二：使用自动锁 synchronized。
方法三：使用手动锁 Lock。

## 48.多线程锁的升级原理是什么?

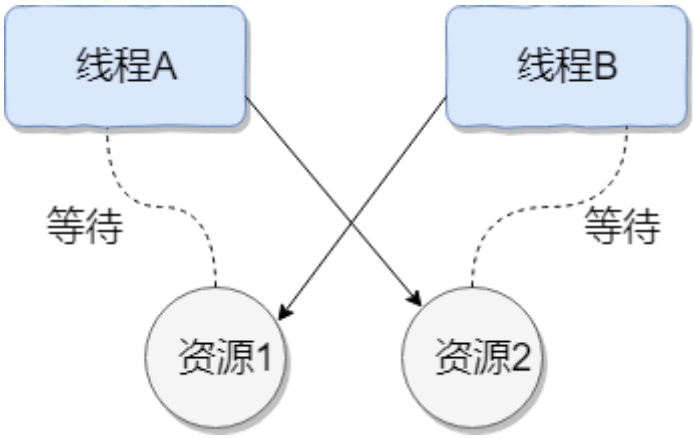
在Java中，锁共有4种状态，级别从低到高依次为：无状态锁，偏向锁，轻量级锁和重量级锁状态，这几个状态会随着竞争情况逐渐升级。锁可以升级但不能降级。

## 49.什么是死锁?

死锁是指两个或两个以上的进程（线程）在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程（线程）称为死锁进程（线程）。

多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。由于线程被无限期地阻塞，因此程序不可能正常终止。

如下图所示，线程 A 持有资源 2，线程 B 持有资源 1，他们同时都想申请对方的资源，所以这两个线程就会互相等待而进入死锁状态。




## 50.怎么防止死锁?

形成死锁的四个必要条件是什么
互斥条件：线程(进程)对于所分配到的资源具有排它性，即一个资源只能被一个线程(进程)占用，直到被该线程(进程)释放
请求与保持条件：一个线程(进程)因请求被占用资源而发生阻塞时，对已获得的资源保持不放。
不剥夺条件：线程(进程)已获得的资源在末使用完之前不能被其他线程强行剥夺，只有自己使用完毕后才释放资源。
循环等待条件：当发生死锁时，所等待的线程(进程)必定会形成一个环路（类似于死循环），造成永久阻塞
我们只要破坏产生死锁的四个条件中的其中一个就可以了。
破坏互斥条件：这个条件我们没有办法破坏，因为我们用锁本来就是想让他们互斥的（临界资源需要互斥访问）。
破坏请求与保持条件：一次性申请所有的资源。
破坏不剥夺条件：占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源。
破坏循环等待条件：靠按序申请资源来预防。按某一顺序申请资源，释放资源则反序释放。破坏循环等待条件。

## 51.ThreadLocal 是什么？有哪些使用场景?

ThreadLocal 是一个本地线程副本变量工具类，在每个线程中都创建了一个 ThreadLocalMap 对象，简单说 ThreadLocal 就是一种以空间换时间的做法，每个线程可以访问自己内部 ThreadLocalMap 对象内的 value。通过这种方式，避免资源在多线程间共享。

原理：线程局部变量是局限于线程内部的变量，属于线程自身所有，不在多个线程间共享。Java提供ThreadLocal类来支持线程局部变量，是一种实现线程安全的方式。但是在管理环境下（如 web 服务器）使用线程局部变量的时候要特别小心，在这种情况下，工作线程的生命周期比任何应用变量的生命周期都要长。任何线程局部变量一旦在工作完成后没有释放，Java 应用就存在内存泄露的风险。

经典的使用场景是为每个线程分配一个 JDBC 连接 Connection。这样就可以保证每个线程的都在各自的 Connection 上进行数据库的操作，不会出现 A 线程关了 B线程正在使用的 Connection； 还有 Session 管理 等问题。

## 52.说一下 synchronized 底层实现原理?

synchronized是Java中的一个关键字，在使用的过程中并没有看到显示的加锁和解锁过程。因此有必要通过javap命令，查看相应的字节码文件。
可以看出在执行同步代码块之前之后都有一个monitor字样，其中前面的是monitorenter，后面的是离开monitorexit，不难想象一个线程也执行同步代码块，首先要获取锁，而获取锁的过程就是monitorenter，在执行完代码块之后，要释放锁，释放锁就是执行monitorexit指令。

## 53.synchronized 和 volatile 的区别是什么?

synchronized 表示只有一个线程可以获取作用对象的锁，执行代码，阻塞其他线程。

volatile 表示变量在 CPU 的寄存器中是不确定的，必须从主存中读取。保证多线程环境下变量的可见性；禁止指令重排序。

区别:
volatile 是变量修饰符；synchronized 可以修饰类、方法、变量。
volatile 仅能实现变量的修改可见性，不能保证原子性；而 synchronized 则可以保证变量的修改可见性和原子性。
volatile 不会造成线程的阻塞；synchronized 可能会造成线程的阻塞。
volatile标记的变量不会被编译器优化；synchronized标记的变量可以被编译器优化。
volatile关键字是线程同步的轻量级实现，所以volatile性能肯定比synchronized关键字要好。但是volatile关键字只能用于变量而synchronized关键字可以修饰方法以及代码块。synchronized关键字在JavaSE1.6之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁以及其它各种优化之后执行效率有了显著提升，实际开发中使用 synchronized 关键字的场景还是更多一些。

## 54.synchronized 和 Lock 有什么区别?

整体上来说 Lock 是 synchronized 的扩展版，Lock 提供了无条件的、可轮询的(tryLock 方法)、定时的(tryLock 带参方法)、可中断的(lockInterruptibly)、可多条件队列的(newCondition 方法)锁操作。另外 Lock 的实现类基本都支持非公平锁(默认)和公平锁，synchronized 只支持非公平锁，当然，在大部分情况下，非公平锁是高效的选择。

## 55.synchronized 和 ReentrantLock 区别是什么?

synchronized 是和 if、else、for、while 一样的关键字，ReentrantLock 是类，这是二者的本质区别。既然 ReentrantLock 是类，那么它就提供了比synchronized 更多更灵活的特性，可以被继承、可以有方法、可以有各种各样的类变量

synchronized 早期的实现比较低效，对比 ReentrantLock，大多数场景性能都相差较大，但是在 Java 6 中对 synchronized 进行了非常多的改进。

相同点：两者都是可重入锁

两者都是可重入锁。“可重入锁”概念是：自己可以再次获取自己的内部锁。比如一个线程获得了某个对象的锁，此时这个对象锁还没有释放，当其再次想要获取这个对象的锁的时候还是可以获取的，如果不可锁重入的话，就会造成死锁。同一个线程每次获取锁，锁的计数器都自增1，所以要等到锁的计数器下降为0时才能释放锁。

主要区别如下：

- ReentrantLock 使用起来比较灵活，但是必须有释放锁的配合动作；
- ReentrantLock 必须手动获取与释放锁，而 synchronized 不需要手动释放和开启锁；
- ReentrantLock 只适用于代码块锁，而 synchronized 可以修饰类、方法、变量等。
- 二者的锁机制其实也是不一样的。ReentrantLock 底层调用的是 Unsafe 的park 方法加锁，synchronized 操作的应该是对象头中 mark word

Java中每一个对象都可以作为锁，这是synchronized实现同步的基础：
- 普通同步方法，锁是当前实例对象
- 静态同步方法，锁是当前类的class对象
- 同步方法块，锁是括号里面的对象

## 56.说一下 atomic 的原理?

Atomic包中的类基本的特性就是在多线程环境下，当有多个线程同时对单个（包括基本类型及引用类型）变量进行操作时，具有排他性，即当多个线程同时对该变量的值进行更新时，仅有一个线程能成功，而未成功的线程可以向自旋锁一样，继续尝试，一直等到执行成功。
AtomicInteger 类主要利用 CAS (compare and swap) + volatile 和 native 方法来保证原子操作，从而避免 synchronized 的高开销，执行效率大为提升。

CAS的原理是拿期望的值和原本的一个值作比较，如果相同则更新成新的值。UnSafe 类的 objectFieldOffset() 方法是一个本地方法，这个方法是用来拿到“原来的值”的内存地址，返回值是 valueOffset。另外 value 是一个volatile变量，在内存中可见，因此 JVM 可以保证任何时刻任何线程总能拿到该变量的最新值。

## Java 线程数过多会造成什么异常?

- 线程的生命周期开销非常高
- 消耗过多的 CPU
  资源如果可运行的线程数量多于可用处理器的数量，那么有线程将会被闲置。大量空闲的线程会占用许多内存，给垃圾回收器带来压力，而且大量的线程在竞争 CPU资源时还将产生其他性能的开销。
- 降低稳定性JVM
  在可创建线程的数量上存在一个限制，这个限制值将随着平台的不同而不同，并且承受着多个因素制约，包括 JVM 的启动参数、Thread 构造函数中请求栈的大小，以及底层操作系统对线程的限制等。如果破坏了这些限制，那么可能抛出OutOfMemoryError 异常。

## 线程 B 怎么知道线程 A 修改了变量

（1）volatile 修饰变量
（2）synchronized 修饰修改变量的方法
（3）wait/notify
（4）while 轮询

## 当一个线程进入一个对象的 synchronized 方法 A 之后，其它线程是否可进入此对象的 synchronized 方法 B?

不能。其它线程只能访问该对象的非同步方法，同步方法则不能进入。因为非静态方法上的 synchronized 修饰符要求执行方法时要获得对象的锁，如果已经进入A 方法说明对象锁已经被取走，那么试图进入 B 方法的线程就只能在等锁池（注意不是等待池哦）中等待对象的锁。

## 什么是线程池？有哪几种创建方式?

线程池顾名思义就是事先创建若干个可执行的线程放入一个池（容器）中，需要的时候从池中获取线程不用自行创建，使用完毕不需要销毁线程而是放回池中，从而减少创建和销毁线程对象的开销。

（1）newSingleThreadExecutor：创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。
（2）newFixedThreadPool：创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。如果希望在服务器上使用线程池，建议使用 newFixedThreadPool方法来创建线程池，这样能获得更好的性能。
（3） newCachedThreadPool：创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60 秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说 JVM）能够创建的最大线程大小。
（4）newScheduledThreadPool：创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。

## 线程池有什么优点?

降低资源消耗：重用存在的线程，减少对象创建销毁的开销。
提高响应速度。可有效的控制最大并发线程数，提高系统资源的使用率，同时避免过多资源竞争，避免堵塞。当任务到达时，任务可以不需要的等到线程创建就能立即执行。
提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。
附加功能：提供定时执行、定期执行、单线程、并发数控制等功能。
综上所述使用线程池框架 Executor 能更好的管理线程、提供系统资源使用率。

## 线程池都有哪些状态?

RUNNING：这是最正常的状态，接受新的任务，处理等待队列中的任务。
SHUTDOWN：不接受新的任务提交，但是会继续处理等待队列中的任务。
STOP：不接受新的任务提交，不再处理等待队列中的任务，中断正在执行任务的线程。
TIDYING：所有的任务都销毁了，workCount 为 0，线程池的状态在转换为 TIDYING 状态时，会执行钩子方法 terminated()。
TERMINATED：terminated()方法结束后，线程池的状态就会变成这个。

## 什么是 Executor 框架？为什么使用 Executor 框架?

Executor 框架是一个根据一组执行策略调用，调度，执行和控制的异步任务的框架。

每次执行任务创建线程 new Thread()比较消耗性能，创建一个线程是比较耗时、耗资源的，而且无限制的创建线程会引起应用程序内存溢出。

所以创建一个线程池是个更好的的解决方案，因为可以限制线程的数量并且可以回收再利用这些线程。利用 Executors 框架可以非常方便的创建一个线程池。

## 线程池中 submit() 和 execute() 方法有什么区别?

接收参数：execute()只能执行 Runnable 类型的任务。submit()可以执行 Runnable 和 Callable 类型的任务。
返回值：submit()方法可以返回持有计算结果的 Future 对象，而execute()没有
异常处理：submit()方便Exception处理

## Executors和ThreaPoolExecutor创建线程池的区别

《阿里巴巴Java开发手册》中强制线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险
Executors 各个方法的弊端：
newFixedThreadPool 和 newSingleThreadExecutor:
主要问题是堆积的请求处理队列可能会耗费非常大的内存，甚至 OOM。
newCachedThreadPool 和 newScheduledThreadPool:
主要问题是线程数最大数是 Integer.MAX_VALUE，可能会创建数量非常多的线程，甚至 OOM。
ThreaPoolExecutor创建线程池方式只有一种，就是走它的构造函数，参数自己指定

## 乐观锁和悲观锁的理解及如何实现，有哪些实现方式?

悲观锁：总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。再比如 Java 里面的同步原语 synchronized 关键字的实现也是悲观锁。

乐观锁：顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于 write_condition 机制，其实都是提供的乐观锁。在 Java 中 java.util.concurrent.atomic 包下面的原子变量类就是使用了乐观锁的一种实现方式 CAS 实现的。

## 什么是 CAS

CAS 是 compare and swap 的缩写，即我们所说的比较交换。
cas 是一种基于锁的操作，而且是乐观锁。在 java 中锁分为乐观锁和悲观锁。悲观锁是将资源锁住，等一个之前获得锁的线程释放锁之后，下一个线程才可以访问。而乐观锁采取了一种宽泛的态度，通过某种方式不加锁来处理资源，比如通过给记录加 version 来获取数据，性能较悲观锁有很大的提高。
CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。如果内存地址里面的值和 A 的值是一样的，那么就将内存里面的值更新成 B。CAS是通过无限循环来获取数据的，若果在第一轮循环中，a 线程获取地址里面的值被b 线程修改了，那么 a 线程需要自旋，到下次循环才有可能机会执行。
java.util.concurrent.atomic 包下的类大多是使用 CAS 操作来实现的(AtomicInteger,AtomicBoolean,AtomicLong)。

## CAS 的会产生什么问题?

1、ABA 问题：
比如说一个线程 one 从内存位置 V 中取出 A，这时候另一个线程 two 也从内存中取出 A，并且 two 进行了一些操作变成了 B，然后 two 又将 V 位置的数据变成 A，这时候线程 one 进行 CAS 操作发现内存中仍然是 A，然后 one 操作成功。尽管线程 one 的 CAS 操作成功，但可能存在潜藏的问题。从 Java1.5 开始 JDK 的 atomic包里提供了一个类 AtomicStampedReference 来解决 ABA 问题。

2、循环时间长开销大
对于资源竞争严重（线程冲突严重）的情况，CAS 自旋的概率会比较大，从而浪费更多的 CPU 资源，效率低于 synchronized。

3、只能保证一个共享变量的原子操作：
当对一个共享变量执行操作时，我们可以使用循环 CAS 的方式来保证原子操作，但是对多个共享变量操作时，循环 CAS 就无法保证操作的原子性，这个时候就可以用锁。