Synchronized：['sɪŋkrənaɪzd]

Volatile：[ˈvɑ:lətl]

在JDK中，提供了多种途径**实现多线程间的并发控制**。

比如常用的：**内部锁**、**重入锁**、**读写锁**和**信号量**。

1、每一个线程有一块工作内存区，其中存放着被所有线程共享的主内存中的变量的值的拷贝。当线程执行时，它在自己的工作内存中操作这些变量。为了存取一个共享的变量，一个线程通常先获取锁定并且清除它的工作内存区，这保证该共享变量从所有线程的共享内存区正确地装入到线程的工作内存区，**当线程解锁时保证该工作内存区中变量的值写回到共享内存中。**

2、并发编程中的三个概念：原子性问题、可见性问题，有序性问题。

**原子性**：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。

**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值。

**有序性：**即程序执行的顺序按照代码的先后顺序执行。

3、指令重排：处理器为了提高程序运行效率，可能会对输入代码进行优化，它不保证程序中各个语句的执行先后顺序同代码中的顺序一致，但是它会保证程序最终执行结果和代码顺序执行的结果是一致的。

4、Java内存模型（Java Memory Model，JMM）来屏蔽各个硬件平台和操作系统的内存访问差异，以实现让Java程序在各种平台下都能达到一致的内存访问效果。那么Java内存模型《1》定义了程序中变量的访问规则，往大一点说是定义了程序执行的次序。注意，为了获得较好的执行性能，Java内存模型并没有限制执行引擎使用处理器的寄存器或者高速缓存来提升指令执行速度，也没有限制编译器对指令进行重排序。也就是说，在java内存模型中，也会存在缓存一致性问题和指令重排序的问题。

1、并发编程的三要素

1、**原子性**：原子，即一个不可再被分割的颗粒。在Java中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、**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）

3、**可见性**：当多个线程访问同一个变量时，如果其中一个线程对其作了修改，其他线程能立即获取到最新的值。

2、线程的五大状态

1、创建状态：当new操作符创建一个线程时

2、就绪状态：调用start方法，处于就绪状态的线程并不一定马上就会执行run方法，还需要等待CPU的调度

3、运行状态：CPU开始调度线程，并开始执行run方法。

4、阻塞状态：线程的执行过程由于一些原因进入到阻塞状态，比如调用sleep方法、尝试去得到一个锁等。

5、死亡状态：run方法执行完或执行的过程中遇到异常。

3、悲观锁乐观锁

悲观锁：每次操作都会加锁，会造成线程阻塞，传统的关系型数据库里面就用到了很多这种锁机制，比如行锁，表锁，读锁，写锁，都是在操作之前先上锁，因此在整个数据处理过程中，将数据处于锁定状态。

乐观锁：每次操作不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止，不会造成线程阻塞。但是如果经常产生冲突，上层应用会不断的进行retry,这样，这样反倒是降低了性能，（本质上不是一种锁，是一种冲突检测机制）

悲观锁会造成访问数据库时间较长，并发行不好，特别是长事务

乐观锁适用于并发量很大的情况，为了解决事务并发带来的问题。

Hiberate中乐观锁与悲观锁

悲观锁大多数情况下依靠数据库的锁机制实现

乐观锁多是基于数据版本记录机制实现，一般是通过为数据库表中增加一个”Version”字段来实现。

Hibernate的乐观锁一般有以下两种方式（使用乐观锁要在映射文件中配置才可生效）

1、基于version，映射文件必须指明optimistic-lock参数

2、基于时间戳，无需指明这个参数

4、线程之间的协作

**1、wait、notify、notifyAll（均是Object类的方法）**

Wait阻塞当前线程，直到notify或者notifyAll来唤醒

2、sleep、yield、join

sleep让当前线程暂停指定时间，只是让出CPU的使用权，并不释放锁

yield暂停当前线程执行，也就是让出CPU的使用权，让其他线程有机会执行，不能指定时间，会让当前线程从运行状态变为就绪状态

join等待调用join方法的线程执行结束，才执行后面的代码。使用场景：**当父线程需要等待子线程执行结束才执行后面内容，或者需要子线程的执行结果会用到join法**

**2、Condition接口**

相比使用Object的wait()、notify()，使用Condition的await()、signal()、signalAll()这种方式实现线程间协作更加安全和高效

**Condition依赖Lock接口，生成一个Condition的基本代码是lock.newCondition（）；调用Condition的await()和signal（）方法，都必须在lock保护之内（在lock和unlock之间）**

5、volatile关键字

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。Volatile是轻量级的sychronized，不会引起线程上下文切换和调度，执行开销更小。

**被volatile修饰的共享变量：**

1、保证volatile修饰的共享变量对所有线程可见。（可见性保证）

2、禁止指令重排优化。（顺序性保证）

**1、Java 中能创建 volatile 数组吗？**

能，Java 中可以创建 volatile 类型数组，不过只是一个指向数组的引用，而不是整个数组。我的意思是，如果改变引用指向的数组，将会受到 volatile 的保护，但是如果多个线程同时改变数组的元素，volatile 标示符就不能起到之前的保护作用了。

**2、volatile 能使得一个非原子操作变成原子操作吗？**

能，一个典型的例子就是类中的long/double类型的成员变量，如果你知道该成员变量会被多个线程访问，你最好将其设为volatile，因为java中读取long类型变量的操作不是原子的，需要分为两步，如果一个线程正在修改long变量的值，另一个线程可能只看到该值的一半（前32位），但是对一个volatile修饰的long或double的量的读写是原子性的。

**3、volatile修饰符实践**

使用volatile修饰的long和double变量，使其能按照原子类型来读写。

Volatile的另一个作用是提供内存屏障（memory barrier），例如在分布式框架中的应用，简单来说，当你在写一个volatile变量之前，java内存模型会插入一个写屏障，读一个volatile变量之前，会插入一个读屏障，在你写一个 volatile 域时。能保证任何数值的更新对所有线程是可见的，因为内存屏障会将其他所有写的值更新到缓存。

6、ReenTrantLock可重入锁（sychronized的区别）

Java.util.concurrent.locks.lock

Lock能完成synchronized所实现的所有功能，ock有比synchronized更精确的线程语义和更好的性能。synchronized会自动释放锁，而Lock一定要求程序员手工释放，并且必须在finally从句中释放。

两个锁都是可重入锁

**实现的方式：**

Sychronized是依赖于JVM实现的，而ReenTrantLock是基于JDK实现的

**性能的区别：**

在sychronized优化以前，sychronized的性能是比ReenTrantLock差很多的，但是自从Synchronized引入了偏向锁和轻量级锁（自旋锁）后，两者的性能就差不多了，其实synchronized的优化我感觉就是借鉴了ReenTrantLock中的CAS技术，核心思想就是试图在用户态就把加锁问题解决，避免进入内核态的阻塞。

**功能上的区别：**

**便利性**：sychronized的使用比较方便，并且是由编译器去保证锁的加锁和释放，而ReenTrantLock需要手工声明来加锁和释放锁，为了避免忘记手工释放造成死锁，所以最好在finally中声明释放锁。

**灵活性和粒度:**ReenTrantLock优于Synchronized

**ReenTrantLock独有的能力**

1、ReenTrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。

2、ReenTrantLock提供了一个Condition（条件）类，用来实现分组唤醒需要唤醒的线程们，而不是像synchronized要么随机唤醒一个线程要么唤醒全部线程。

3、ReenTrantLock提供了一种能够中断等待锁的线程的机制，通过lock.lockInterruptibly()来实现这个机制。

**Sychronized的使用：**

1、修饰普通方法的同步对象是实例对象

2、修饰静态方法的同步对象是类本身

3、修饰代码块可以自己设置同步对象

**Sychronized的缺点：**

会让没有得到锁的资源进入Block状态，争夺到资源之后又转为Running状态，这个过程涉及到操作系统用户模式和内核模式的切换，代价比较高，虽然1.6对sychronized做了优化，增加了从偏向锁到轻量锁再到重量锁的过度，但是在最终转变为重量锁之后，性能仍然较低。

7、CAS

AtomicBoolean，AtomicInteger,AtomicLong（原子类）以及lock相关类的底层就是用CAS实现的，在一定程度上性能比synchronized更高

**CAS只能保证一个共享变量的原子操作，乐观锁用到的就是CAS**

CAS全称Compare And Swap，即**比较替换**，是实现并发应用用到的一种技术，操作包含三个操作数 内存位置（V）、预期原值（A）和新值(B)，如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值 。否则，处理器不做任何操作。

为什么会有CAS？

如果使用sychronized来保证同步会存在以下问题

synchronized 是一种悲观锁，在使用上会造成一定的性能问题。在多线程竞争下，加锁、释放锁会导致比较多的上下文切换和调度延时，引起性能问题。一个线程持有锁会导致其它所有需要此锁的线程挂起。

因为硬件芯片中加入了大量的并发操作原语，从硬件层面提升效率，java可以通过JNI(java本地方法)，使得java程序越过JVM直接调用本地方法。利用CPU 的CAS指令，同时借助JNI来完成java非阻塞算法。

CAS存在的问题：

1、ABA问题：

因为CAS需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。

解决方法：

ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么A－B－A 就会变成1A-2B－3A。

2、循环时间长开销大

自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销

3、只能保证一个共享变量的原子性

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。

1.5后提供了AtomicReference类来保证引用对象的原子性，可以把多个变量放在一个对象里进行CAS操作。

**Concurrent包实现**

由于java的CAS 同时具有volatile读和volatile写的内存语义。Java线程之间通信现在有了下面四种方式。

1、A线程写volatile变量，随后B线程读这个volatile变量。

2、A线程写volatile变量，随后B线程用CAS更新这个volatile变量。

3、A线程用CAS更新一个volatile变量，随后B线程用CAS更新这个volatile变量。

4、A线程用CAS更新一个volatile变量，随后B线程读这个volatile变量。

Java的CAS使用现代处理器上提供的高效机器级别的原子指令，这些原子指令以原子方式对内存执行读写改操作，这是多处理器中实现同步的关键。同时volatile变量的读写和CAS 可以实现线程间的通信，就形成了整个concurrent包的基础。

8、AQS和原子类

AQS:**抽象队列同步器**，是一种基于状态的链表管理方式，状态是CAS去修改的，它是java.Util.concurrent包中的基石

AQS定义了一套多线程访问共享资源的同步框架，许多同步类的实现都依赖于它。AQS定义了两种资源访问形式：独占和共享，自定义同步器在实现时只需要实现共享资源state的获取与释放即可，至于具体的线程等待队列的维护，如获取资源失败入队/唤醒出队，AQS已经实现好了。

ReentrantLock​、CountDownLatcher、Semaphore 的实现都是基于AQS

CountDownLatcher:同步辅助器，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一致等待**。一个被初始化为N的CountDownLatch可以被用来“在N个线程都完成了某种操作（或者一些操作已经被完成了N次）之后创建一个线程”。**

CountDownLatcher：它简单地阻止了任何调用await()的线程继续执行，直到所有线程都能通过。

***Semaphore***：[ˈseməˌfɔ:, -ˌfəʊr]：信号计数量，信号量控制的是线程并发的数量，内部主要是通过AQS实现线程的管理。通常用于限制可以访问某些资源的线程数目。

一个信号量有且仅有3种操作，且他们全部都是原子的：初始化、增加、减少

增加可以为一个进程解除阻塞；   
减少可以让一个进程进入阻塞。

**Semaphore面试题目思考：**

很多情况下，可能有多个线程需要访问数目很少的资源。假想在服务器上运行着若干个回答客户端请求的线程。这些线程需要连接到同一数据库，但任一时刻 只能获得一定数目的数据库连接。你要怎样才能够有效地将这些固定数目的数据库连接分配给大量的线程？

1、给方法加同步锁，保证了同一时刻只能有一个人去调用此方法，其他所有线程排队等待，但是这种情况下即使你的数据库连接有10个，也始终只有一个处于使用状态，这样将会大大浪费系统资源，而且系统的运行效率非常低下。

2、另外一种方法是使用信号量，通过信号量许可与数据库可用的连接数相同的数目，将大大提高效率和性能。

**原子类：**相比锁机制，使用原子类更精巧轻量，性能开销更小

java.util.concurrent.atomic包：原子类的小工具包，支持在单个变量上解除锁的线程安全编程

如果同一个变量要被多个线程访问，则可以使用该包中的类

AtomicBoolean

AtomicInteger

AtomicLong

AtomicReference

这个包中还提供了可以用于反射操作的类。

. 原子性整型 AtomicInteger，使用compareAndSet方法来进行原子更新操作（CAS机制）

9、Future模式

在并发编程我们一般使用Runable去执行异步任务，然而这样做我们是不能拿到异步任务的返回值的，但是使用Future 就可以

Java现在的多线程机制，核心方法run是没有返回值的；如果要保存run方法里面的计算结果，必须等待run方法计算完，无论计算过程多么耗时。

线程是属于异步计算模型，所以你不可能直接从别的线程中得到函数返回值。这时候，Future就出场了。Futrue可以监视目标线程调用call的情况，当你调用Future的get()方法以获得结果时，当前线程就开始阻塞，直接call方法结束返回结果

Java内置的Future主要使用到了Callable接口和FutureTask类，callable是类似于Runnable一样，只不过调用的是call方法，该方法有一个泛型返回值类型，你可以任意指定。

10、线程池

为了避免重复创建线程，线程池的出现可以让线程进行复用，通俗来讲，当有工作来，就会向线程池拿一个线程，当工作完成后，并不是直接关闭线程，而是将这个线程归还给线程池。

Java里面线程池的顶级接口是Executor，真正的线程池接口是ExecutorService。ThreadpoolExecutor是ExecutorService的默认实现。

配置一个线程池通常是复杂的，很可能配置的线程池不是最优的，因此Executors类里面提供了一些静态工厂类，生成一些常用的线程池。

**1、newSingleThreadExecutor创建一个单线程的线程池。**

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。

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

四类线程池的问题：

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

**1、什么是线程池？线程池的作用？**

为了避免重复创建线程，线程池的出现可以让线程进行复用，线程池作用就是限制系统中执行线程的数量。（复用+限制数量）

**2、为什么要使用线程池？**

1、减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。

2、可以根据系统的承受能力，调整线程池种中作线程的数目，防止因为内存消耗过多

**任务缓存队列**

WorkQueue

Workqueue的类型为blockingQueue<Runnable>通常有三种类型：

1、**有界任务队列**ArrayBlockingQueue：基于数组的先进先出队列，此队列创建时必须指定大小；

2、**无界任务队列**LinkedBlockingQueue：基于链表的先进先出队列，如果创建时没有指定此队列大小，则默认为Integer.MAX\_VALUE；

3、**直接提交队列**synchronousQueue：这个队列比较特殊，它不会保存提交的任务，而是将直接新建一个线程来执行新来的任务。

**拒绝策略：（当资源源源不断地过来，而我们的系统又处理不过来的时候，我们要采取的策略是拒绝服务）**

1、AbortPolicy:丢弃任务并抛出RejectedExecutionException(默认策略)

2、CallerRunsPolicy：只要线程池未关闭，该策略直接在调用者线程中，运行当前被丢弃的任务。这个策略显然不想放弃执行任务，但由于池中已经没有资源了，那么就直接调用该execte的线程本身来执行。

3、DiscardPolicy:不能执行的任务将被删除，这和abort策略几乎一样，也是丢弃任务，只不过不抛出异常。

4、DiscardOldestPolicy：如果执行的程序尚未关闭，则位于工作队列头部的任务将被删除，然后重试执行程序。（如果再次失败，则重复此过程 ）

**任务处理策略：**

1、如果当线程池中的线程数目小于corePoolSize，则每来一个任务，就会创建一个线程去执行这个任务；

2、如果当前线程池中的线程数目>=corePoolSize，则每来一个任务，会尝试将其添加到任务缓存队列当中，若添加成功，则该任务会等待空闲线程将其取出去执行；若添加失败（一般来说是任务缓存队列已满），则会尝试创建新的线程去执行这个任务；如果当前线程池中的线程数目达到maximumPoolSize，则会采取任务拒绝策略进行处理；

3、如果线程池中的线程数量大于 corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止。

**线程池的关闭**：

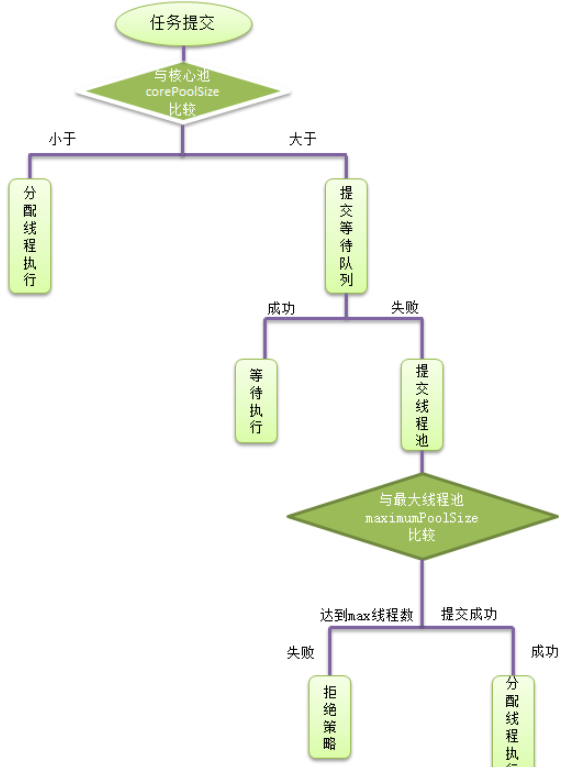
ThreadPoolExecutor提供了两个方法，用于线程池的关闭

1、shutdown():不会立即终止线程池，而是要等所有任务缓存队列中的任务都执行完后才终止，但再也不会接受新的任务

2、shutdownnow():立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任务

**源码分析：**

最核心的execute方法，这个方法在AbstractExecutorService中没有实现，从Executor接口,直到ThreadPoolExecutor才实现了该方法。Execute源码执行步骤：“



1、WorkerCountOf()方法能够取得当前线程池中的线程的总数

2、excute()方法中添加任务的方式是使用addWorker（）

**如何选择线程池数量？**

线程池的大小决定着系统的性能，过大或者过小的线程池都无法发挥最优的系统性能。一般来说确定线程池的大小要考虑CPU的数量，内存的大小，任务是计算密集型还是IO密集型等因素。

如果希望处理器达到理想的使用率，那么线程池的最优大小为

最优大小=Cpu数量\*期望cpu使用率\*（1+等待时间与计算时间的比率）

**线程池工厂：**

Executors的线程池如果不指定线程工厂会使用Executors中的DefaultThreadFactory,

默认线程池工厂创建的线程都是非守护线程。

使用自定义的线程工厂可以做很多事情，比如可以跟踪线程池在何时创建了多少线程，也可以自定义线程名称和优先级，也可以设置守护线程，如果将新建的线程设置成守护线程，当主线程退出后，并会强制销毁线程池。

**扩展线程池：**

ThreadPoolExecutor是可以扩展的，它提供了几个可以在子类中改写的方法：beforeExecute,afterExecute和terimated，在执行任务的线程中将调用beforeExecute和afterExecute

**应用场景**：

这些方法中添加日志，计时，监控或统计收集的功能。

手动创建线程池的几个注意点：

1、任务独立，如果任务依赖于其他的任务，可能产生死锁。

2、**合理配置阻塞时间过长的任务。如果任务阻塞时间过长，那么即使不出现死锁，线程池的性能也会变得很糟糕**

**3、设置合理的线程池大小，用公式**

**4、选择合适的阻塞队列，**newFixedThreadPool和newSingleThreadExecutor都使用了无界的阻塞队列，无界的阻塞队列会有消耗很大的内存，如果选择了有界的阻塞队列，它会规避内存占用过大的问题，但当任务填满有界阻塞队列时，新的任务处理又比较麻烦了，这时需要选择合适的拒绝策略。队列的大小和线程池的大小要一起调节，对于非常大的或者无界的线程池，可以使用SynchronousQueue来避免任务排队，以直接将任务从生产者提交到工作者线程。

**线程池有哪些参数？**

核心线程数、最大线程数、闲置线程存活时间、任务队列容量、允许核心线程超时、

11、锁分类

**公平锁，非公平锁**

公平锁是指多个线程按照申请锁的顺序来获取锁。非公平锁是指多个线程获取锁的顺序并不是按照申请锁的顺序，有可能后申请的线程比先申请的线程优先获取锁。

**可重入锁**

可重入锁又名递归锁，是指在同一个线程在外层方法获取锁的时候，在进入内层方法会自动获取锁

**独享锁/共享锁**

独享锁是指该锁一次只能被一个线程所持有。共享锁是指该锁可被多个线程所持有。

**互斥锁/读写锁**

互斥锁在Java中的具体实现就是ReentrantLock，读写锁在Java中的具体实现就是ReadWriteLock

互斥锁：一次只能一个线程拥有互斥锁，其他线程只有等待。

读写锁：

1、多个读者可以同时进行读。

2、写着必须互斥（只允许一个写者写，也不能读者写者同时进行）

3、写着优先于读者（一旦又写者，则后续读者必须等待，唤醒时有限考虑写着）

**分段锁**

分段锁其实是一种锁的设计，并不是具体的一种锁，对于ConcurrentHashMap而言，其并发的实现就是通过分段锁的形式来实现高效的并发操作。ConcurrentHashMap中的分段锁称为Segment，它即类似于HashMap（JDK7与JDK8中HashMap的实现）的结构，即内部拥有一个Entry数组，数组中的每个元素又是一个链表；同时又是一个ReentrantLock（Segment继承了ReentrantLock)。当需要put元素的时候，并不是对整个hashmap进行加锁，而是先通过hashcode来知道他要放在那一个分段中，然后对这个分段进行加锁，所以当多线程put的时候，只要不是放在一个分段中，就实现了真正的并行的插入。

分段锁的设计目的是细化锁的粒度，当操作不需要更新整个数组的时候，就仅仅针对数组中的一项进行加锁操作。

**自旋锁：**

自旋锁是指尝试获取锁的线程不会立即阻塞，而是采用循环的方式去尝试获取锁，这样的好处是减少线程上下文切换的消耗，缺点是循环会消耗CPU。

**偏向锁/轻量级锁/重量级锁**

这三种锁是指锁的状态，并且是针对Synchronized

**偏向锁**是指一段同步代码一直被一个线程所访问，那么该线程会自动获取锁。降低获取锁的代价

**轻量级**锁是指当锁是偏向锁的时候，被另一个线程所访问，偏向锁就会升级为轻量级锁，其他线程会通过自旋的形式尝试获取锁，不会阻塞，提高性能。  
**重量级**锁是指当锁为轻量级锁的时候，另一个线程虽然是自旋，但自旋不会一直持续下去，当自旋一定次数的时候，还没有获取到锁，就会进入阻塞，该锁膨胀为重量级锁。重量级锁会让其他申请的线程进入阻塞，性能降低。

13、常见问题

**1. 进程和线程之间有什么不同？**

一个进程是一个独立(self contained)的运行环境，它可以被看作一个程序或者一个应用。而线程是在进程中执行的一个任务。Java运行环境是一个包含了不同的类和程序的单一进程。线程可以被称为轻量级进程。线程需要较少的资源来创建和驻留在进程中，并且可以共享进程中的资源。

**2. 多线程编程的好处是什么？**

在多线程程序中，多个线程被并发的执行以提高程序的效率，CPU不会因为某个线程需要等待资源而进入空闲状态。多个线程共享堆内存(heap memory)，因此创建多个线程去执行一些任务会比创建多个进程更好。举个例子，Servlets比CGI更好，是因为Servlets支持多线程而CGI不支持。

**3. 用户线程和守护线程有什么区别？**

当我们在Java程序中创建一个线程，它就被称为用户线程。一个守护线程是在后台执行并且不会阻止JVM终止的线程。当没有用户线程在运行的时候，JVM关闭程序并且退出。一个守护线程创建的子线程依然是守护线程。

**4. 我们如何创建一个线程？**

有两种创建线程的方法：一是实现Runnable接口，然后将它传递给Thread的构造函数，创建一个Thread对象；二是直接继承Thread类。若想了解更多可以阅读这篇关于如何在[Java中创建线程](http://www.journaldev.com/1016/java-thread-example-extending-thread-class-and-implementing-runnable-interface)的文章。

**5. 有哪些不同的线程生命周期？**

当我们在Java程序中新建一个线程时，它的状态是*New。*当我们调用线程的start()方法时，状态被改变为*Runnable*。线程调度器会为*Runnable*线程池中的线程分配CPU时间并且讲它们的状态改变为*Running。*其他的线程状态还有*Waiting，Blocked* 和*Dead*。读这篇文章可以了解更多关于[线程生命周期](http://www.journaldev.com/1044/life-cycle-of-thread-understanding-thread-states-in-java)的知识。

**6. 可以直接调用Thread类的run()方法么？**

当然可以，但是如果我们调用了Thread的run()方法，它的行为就会和普通的方法一样，为了在新的线程中执行我们的代码，必须使用Thread.start()方法。

**7. 如何让正在运行的线程暂停一段时间？**

我们可以使用Thread类的Sleep()方法让线程暂停一段时间。需要注意的是，这并不会让线程终止，一旦从休眠中唤醒线程，线程的状态将会被改变为*Runnable*，并且根据线程调度，它将得到执行。

**8. 你对线程优先级的理解是什么？**

每一个线程都是有优先级的，一般来说，高优先级的线程在运行时会具有优先权，但这依赖于线程调度的实现，这个实现是和操作系统相关的(OS dependent)。我们可以定义线程的优先级，但是这并不能保证高优先级的线程会在低优先级的线程前执行。线程优先级是一个int变量(从1-10)，1代表最低优先级，10代表最高优先级。

**9. 什么是线程调度器(Thread Scheduler)和时间分片(Time Slicing)？**

线程调度器是一个操作系统服务，它负责为*Runnable*状态的线程分配CPU时间。一旦我们创建一个线程并启动它，它的执行便依赖于线程调度器的实现。时间分片是指将可用的CPU时间分配给可用的*Runnable*线程的过程。分配CPU时间可以基于线程优先级或者线程等待的时间。线程调度并不受到Java虚拟机控制，所以由应用程序来控制它是更好的选择（也就是说不要让你的程序依赖于线程的优先级）。

**10. 在多线程中，什么是上下文切换(context-switching)？**

上下文切换是存储和恢复CPU状态的过程，它使得线程执行能够从中断点恢复执行。上下文切换是多任务操作系统和多线程环境的基本特征。

**11. 你如何确保main()方法所在的线程是Java程序最后结束的线程？**

我们可以使用Thread类的joint()方法来确保所有程序创建的线程在main()方法退出前结束。这里有一篇文章关于[Thread类的joint()方法](http://www.journaldev.com/1024/java-thread-join-example-with-explanation)。

**12.线程之间是如何通信的？**

当线程间是可以共享资源时，线程间通信是协调它们的重要的手段。Object类中wait()\notify()\notifyAll()方法可以用于线程间通信关于资源的锁的状态。点击[这里](http://www.journaldev.com/1037/java-thread-wait-notify-and-notifyall-example)有更多关于线程wait, notify和notifyAll.

**13.为什么线程通信的方法wait(), notify()和notifyAll()被定义在Object类里？**

Java的每个对象中都有一个锁(monitor，也可以成为监视器) 并且wait()，notify()等方法用于等待对象的锁或者通知其他线程对象的监视器可用。在Java的线程中并没有可供任何对象使用的锁和同步器。这就是为什么这些方法是Object类的一部分，这样Java的每一个类都有用于线程间通信的基本方法

**14. 为什么wait(), notify()和notifyAll()必须在同步方法或者同步块中被调用？**

当一个线程需要调用对象的wait()方法的时候，这个线程必须拥有该对象的锁，接着它就会释放这个对象锁并进入等待状态直到其他线程调用这个对象上的notify()方法。同样的，当一个线程需要调用对象的notify()方法时，它会释放这个对象的锁，以便其他在等待的线程就可以得到这个对象锁。由于所有的这些方法都需要线程持有对象的锁，这样就只能通过同步来实现，所以他们只能在同步方法或者同步块中被调用。

**15. 为什么Thread类的sleep()和yield()方法是静态的？**

Thread类的sleep()和yield()方法将在当前正在执行的线程上运行。所以在其他处于等待状态的线程上调用这些方法是没有意义的。这就是为什么这些方法是静态的。它们可以在当前正在执行的线程中工作，并避免程序员错误的认为可以在其他非运行线程调用这些方法。

**16.如何确保线程安全？**

在Java中可以有很多方法来保证线程安全——同步，使用原子类(atomic concurrent classes)，实现并发锁，使用volatile关键字，使用不变类和线程安全类。在[线程安全教程](http://www.journaldev.com/1061/java-synchronization-and-thread-safety-tutorial-with-examples)中，你可以学到更多。

**17. volatile关键字在Java中有什么作用？**

当我们使用volatile关键字去修饰变量的时候，所以线程都会直接读取该变量并且不缓存它。这就确保了线程读取到的变量是同内存中是一致的。

**18. 同步方法和同步块，哪个是更好的选择？**

同步块是更好的选择，因为它不会锁住整个对象（当然你也可以让它锁住整个对象）。同步方法会锁住整个对象，哪怕这个类中有多个不相关联的同步块，这通常会导致他们停止执行并需要等待获得这个对象上的锁。

**19.如何创建守护线程？**

使用Thread类的setDaemon(true)方法可以将线程设置为守护线程，需要注意的是，需要在调用start()方法前调用这个方法，否则会抛出IllegalThreadStateException异常。

**20. 什么是ThreadLocal?**

ThreadLocal用于创建线程的本地变量，我们知道一个对象的所有线程会共享它的全局变量，所以这些变量不是线程安全的，我们可以使用同步技术。但是当我们不想使用同步的时候，我们可以选择ThreadLocal变量。

每个线程都会拥有他们自己的Thread变量，它们可以使用get()\set()方法去获取他们的默认值或者在线程内部改变他们的值。ThreadLocal实例通常是希望它们同线程状态关联起来是private static属性。在[ThreadLocal例子](http://www.journaldev.com/1076/java-threadlocal-example-to-create-thread-local-variables)这篇文章中你可以看到一个关于ThreadLocal的小程序。

**21. 什么是Thread Group？为什么建议使用它？**

ThreadGroup是一个类，它的目的是提供关于线程组的信息。

ThreadGroup API比较薄弱，它并没有比Thread提供了更多的功能。它有两个主要的功能：一是获取线程组中处于活跃状态线程的列表；二是设置为线程设置未捕获异常处理器(ncaught exception handler)。但在Java 1.5中Thread类也添加了*setUncaughtExceptionHandler(UncaughtExceptionHandler eh)* 方法，所以ThreadGroup是已经过时的，不建议继续使用。

|  |  |
| --- | --- |
| 1 | t1.setUncaughtExceptionHandler(new UncaughtExceptionHandler(){ |
| 2 |  |

|  |  |
| --- | --- |
| 3 | @Override |
| 4 | public void uncaughtException(Thread t, Throwable e) { |

|  |  |
| --- | --- |
| 5 | System.out.println("exception occured:"+e.getMessage()); |
| 6 | } |

|  |  |
| --- | --- |
| 7 |  |
| 8 | }); |

**22. 什么是Java线程转储(Thread Dump)，如何得到它？**

线程转储是一个JVM活动线程的列表，它对于分析系统瓶颈和死锁非常有用。有很多方法可以获取线程转储——使用Profiler，Kill -3命令，jstack工具等等。我更喜欢jstack工具，因为它容易使用并且是JDK自带的。由于它是一个基于终端的工具，所以我们可以编写一些脚本去定时的产生线程转储以待分析。读这篇文档可以了解更多关于[产生线程转储](http://www.journaldev.com/1053/how-to-generate-thread-dump-in-java)的知识。

**23. 什么是死锁(Deadlock)？如何分析和避免死锁？**

死锁是指两个以上的线程永远阻塞的情况，这种情况产生至少需要两个以上的线程和两个以上的资源。

分析死锁，我们需要查看Java应用程序的线程转储。我们需要找出那些状态为BLOCKED的线程和他们等待的资源。每个资源都有一个唯一的id，用这个id我们可以找出哪些线程已经拥有了它的对象锁。

避免嵌套锁，只在需要的地方使用锁和避免无限期等待是避免死锁的通常办法，阅读这篇文章去学习[如何分析死锁](http://www.journaldev.com/1058/java-deadlock-example-and-how-to-analyze-deadlock-situation)。

**24. 什么是Java Timer类？如何创建一个有特定时间间隔的任务？**

java.util.Timer是一个工具类，可以用于安排一个线程在未来的某个特定时间执行。Timer类可以用安排一次性任务或者周期任务。

java.util.TimerTask是一个实现了Runnable接口的抽象类，我们需要去继承这个类来创建我们自己的定时任务并使用Timer去安排它的执行。

这里有关于[java Timer的例子](http://www.journaldev.com/1050/java-timer-and-timertask-example-tutorial)。

**25. 什么是线程池？如何创建一个Java线程池？**

一个线程池管理了一组工作线程，同时它还包括了一个用于放置等待执行的任务的队列。

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。[线程池例子](http://www.journaldev.com/1069/java-thread-pool-example-using-executors-and-threadpoolexecutor)展现了如何创建和使用线程池，或者阅读[ScheduledThreadPoolExecutor](http://www.journaldev.com/2340/java-scheduledthreadpoolexecutor-example-to-schedule-tasks-after-delay-and-execute-periodically)例子，了解如何创建一个周期任务。

**Java并发面试问题**

**1. 什么是原子操作？在Java Concurrency API中有哪些原子类(atomic classes)？**

原子操作是指一个不受其他操作影响的操作任务单元。原子操作是在多线程环境下避免数据不一致必须的手段。

int++并不是一个原子操作，所以当一个线程读取它的值并加1时，另外一个线程有可能会读到之前的值，这就会引发错误。

为了解决这个问题，必须保证增加操作是原子的，在JDK1.5之前我们可以使用同步技术来做到这一点。到JDK1.5，java.util.concurrent.atomic包提供了int和long类型的装类，它们可以自动的保证对于他们的操作是原子的并且不需要使用同步。可以阅读这篇文章来了解[Java的atomic类](http://www.journaldev.com/1095/java-atomic-operations-atomicinteger-example)。

**2. Java Concurrency API中的Lock接口(Lock interface)是什么？对比同步它有什么优势？**

Lock接口比同步方法和同步块提供了更具扩展性的锁操作。他们允许更灵活的结构，可以具有完全不同的性质，并且可以支持多个相关类的条件对象。

它的优势有：

* 可以使锁更公平
* 可以使线程在等待锁的时候响应中断
* 可以让线程尝试获取锁，并在无法获取锁的时候立即返回或者等待一段时间
* 可以在不同的范围，以不同的顺序获取和释放锁

阅读更多[关于锁的例子](http://www.journaldev.com/2377/java-lock-example-and-concurrency-lock-vs-synchronized)

**3. 什么是Executors框架？**

Executor框架同java.util.concurrent.Executor 接口在Java 5中被引入。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。

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，阅读这篇文章可以了解[如何使用Executor框架创建一个线程池](http://www.journaldev.com/1069/java-thread-pool-example-using-executors-and-threadpoolexecutor)。

**4. 什么是阻塞队列？如何使用阻塞队列来实现生产者-消费者模型？**

java.util.concurrent.BlockingQueue的特性是：当队列是空的时，从队列中获取或删除元素的操作将会被阻塞，或者当队列是满时，往队列里添加元素的操作会被阻塞。

阻塞队列不接受空值，当你尝试向队列中添加空值的时候，它会抛出NullPointerException。

阻塞队列的实现都是线程安全的，所有的查询方法都是原子的并且使用了内部锁或者其他形式的并发控制。

BlockingQueue 接口是java collections框架的一部分，它主要用于实现生产者-消费者问题。

阅读这篇文章了解[如何使用阻塞队列实现生产者-消费者问题。](http://www.journaldev.com/1034/java-blockingqueue-example-implementing-producer-consumer-problem)

**5. 什么是Callable和Future?**

Java 5在concurrency包中引入了java.util.concurrent.Callable 接口，它和Runnable接口很相似，但它可以返回一个对象或者抛出一个异常。

Callable接口使用泛型去定义它的返回类型。Executors类提供了一些有用的方法去在线程池中执行Callable内的任务。由于Callable任务是并行的，我们必须等待它返回的结果。java.util.concurrent.Future对象为我们解决了这个问题。在线程池提交Callable任务后返回了一个Future对象，使用它我们可以知道Callable任务的状态和得到Callable返回的执行结果。Future提供了get()方法让我们可以等待Callable结束并获取它的执行结果。

阅读这篇文章了解更多[关于Callable，Future的例子](http://www.journaldev.com/1090/java-callable-future-example)。

**6. 什么是FutureTask?**

FutureTask是Future的一个基础实现，我们可以将它同Executors使用处理异步任务。通常我们不需要使用FutureTask类，单当我们打算重写Future接口的一些方法并保持原来基础的实现是，它就变得非常有用。我们可以仅仅继承于它并重写我们需要的方法。阅读[Java FutureTask例子](http://www.journaldev.com/1650/java-futuretask-example-program)，学习如何使用它。

**7.什么是并发容器的实现？**

Java集合类都是快速失败的，这就意味着当集合被改变且一个线程在使用迭代器遍历集合的时候，迭代器的next()方法将抛出ConcurrentModificationException异常。

并发容器支持并发的遍历和并发的更新。

主要的类有ConcurrentHashMap, CopyOnWriteArrayList 和CopyOnWriteArraySet，阅读这篇文章了解[如何避免ConcurrentModificationException](http://www.journaldev.com/378/how-to-avoid-concurrentmodificationexception-when-using-an-iterator)。

**8. Executors类是什么？**

Executors为Executor，ExecutorService，ScheduledExecutorService，ThreadFactory和Callable类提供了一些工具方法。

Executors可以用于方便的创建线程池