JAVA 多线程并发

1. java线程创建方式：
   1. 继承Thread类：
      1. Thread类是实现了Runnable接口的一个实例，代表着线程的实例。启动线程用其中的start方法，会跑run方法，是一个新线程
   2. 实现Runnable接口：
      1. 自己的类已经extends另外的类，就无法直接extends Thread，可以实现一个Runnable接口
   3. 有返回值任务：
      1. 必须实现callable接口，就像没返回值的一定要实现Runnable接口。执行callable任务后，获得一个Futurn对象，对该对象用get就可以获得callable任务返回的object了。再配合线程池接口就可以实现有返回结果的多线程了。
2. 线程池的方式：
3. 线程和数据库连接这种是非常宝贵的资源，采用缓存，线程池来存储，避免每次创建，销毁带来的开销。

四种线程池：

1. 线程池的顶级接口是Executor，但Executor不是一个线程池，更是一个执行线程的工具，真正的线程池接口是ExecutorService。
2. NewcachedThreadPool：
   * 1. 构建一个可根据需要创建新线程的线程池。调用execute时将重用一千构造的线程，如果没有县城可用，则创建一个新线程并添加到线程池里。

newFixedThreadPool：

创建一个可重用固定线程数的线程池，以共享的无界队列的方式来运行这些线程。在某个线程被显式地关闭前，池中的线程将一直存在。

newScheduledThreadPool：

创建一个线程池，它可以在给定延迟后运行命令或者定期地执行

newSingleThreadExecutor：

返回一个线程池，这线程池中只有一个线程，在县城死后会重新启动一个线程继续执行下去。

线程生命周期：

线程有五种状态：新建，就绪，运行，阻塞，死亡

1. 新建态：只分配了内存，并初始化了成员变量的值
2. 就绪态：对线程对象调用了start（）方法，该线程就处于就绪态，会为它创建方法调用栈和程序计数器。
3. 运行态：就绪态获得了cpu，执行run方法
4. 阻塞态：还没执行完，让出了cpu
   * 1. 等待阻塞：线程方法内执行了.wait()方法，线程会进入等待队列
     2. 同步阻塞：获取对象的同步锁的时候，发现被占用了进入锁池
     3. 其他阻塞：sleep，发出IO之类的。等他结束，再进入可运行态。
5. 死亡：
   * 1. 正常结束
     2. 异常结束
     3. 直接stop，可能会产生死锁，不推荐使用。

终止线程的方式：

1. 正常结束
2. 使用退出标志，对一些需要长时间运行的线程，只有在外部条件满足的情况下，才能关闭这些线程。

一般来说exit这个关键字会加上volatile关键字，来保证同时只有一个修改exit的值

1. Interrupt方法结束线程：
   * 1. 线程处于被阻塞状态时，调用线程的interrupt方法，会抛出InterruptException异常，通过代码捕获异常后，再用break跳出循环状态。
     2. 线程未处于阻塞状态：使用中断标志位来判断，类似于上面的退出标志，interrupt方法会把中断标志设为true。
2. stop方法终止线程：
   * 1. 不安全的地方在于thread.stop()调用后，会释放所有锁，就很可能会呈现出不一致性，导致一些应用程序错误。

Sleep和Wait区别：

1. Sleep属于Thread类，wait属于Object类
2. Sleep方法导致了程序暂停执行指定的时间，让出cpu给其他，但他的监控状态保持着，指定的时间到了才会恢复
3. 调用sleep方法的过程中，线程不会释放对象锁。
4. 调用wait方法时，线程会放弃对象锁，进入等待此对象的等待锁定池，只有对此对象再用notify方法后线程才会进入对象锁定池准备获取对象锁，再进入运行状态。

Start和run的区别：

1. start方法用来启动线程，真正实现了多线程，无需等待run方法体中代码执行完，就可以继续执行
2. 通过start，此时处于就绪态，没有运行
3. Run称为线程体，执行时就进入了运行状态。

JAVA后台线程：

运行在JVM后台中的一种特殊线程，当剩下的所有线程都是守护线程时，JVM就可以退出。如果有其他的非守护线程就不会退出。

JAVA锁机制：

1. 乐观锁：
   * 1. 认为读多写少，遇到并发写的可能性低。取数据时不会上锁，但是更新数据的时候会判断一下在此期间有没有其他人更新，写的时候读出版本号，加锁。
     2. 是一种CAS操作，CAS是一种更新的原子操作，比较当前值和传入值是否一样，一样就更新，否则就不更新。
2. 悲观所：
   * 1. 认为写多，每次读写数据的时候就都上锁。
3. 自旋锁：上面的是对文件上锁，这里是线程自旋。认为持有锁的线程能够在很短时间内释放锁资源，等待竞争锁的线程就不需要再做内核态和用户态的切换了，只在等一等，再最大时间内如果还是没有获得锁，就停止自选，进入阻塞状态。
   * 1. 适用于锁的竞争不激烈，很快就释放的场景
     2. 不适合所得竞争激烈，或者持有锁的进程需要长时间占用锁。会造成cpu的浪费。

Jdk1.6引入了适应性自旋锁，意味着自选的时间不再是固定的了。而是由前一次的自选时间和锁的持有者的状态来决定的，一般认为一个线程上下文切换是一个最佳时间。

1. Synchronize同步锁：

独占式的悲观锁，可重入锁，非公平锁。

作用范围：

1. 作用在方法时，锁住的是对象的实例
2. 作用在静态方法时，锁住的是class实例，相当于类的全局锁
3. 作用在对象实例时，锁住所有以该对象为锁的代码块，他有多个队列，当多个线程一起访问对象监视器的时候，对象监视器会把线程存储在不同的容器中。

核心组件：

1. Wait Set：调用wait方法被阻塞的线程在这里
2. 竞争队列，请求锁的线程都放在竞争队列里
3. Entry list：有资格成为候选资源的线程被移动到这里
4. OnDeck：只有一个线程在竞争锁资源，他被称为OnDeck
5. Owner：当前获得锁资源的线程被称为Owner
6. ！Owner：当前释放锁的线程

作用过程：

JVM 每次从队列的尾部取出一个数据用于锁竞争候选者（OnDeck），但是并发情况下，

ContentionList 会被大量的并发线程进行 CAS 访问，为了降低对尾部元素的竞争，JVM 会将

一部分线程移动到 EntryList 中作为候选竞争线程。

2. Owner 线程会在 unlock 时，将 ContentionList 中的部分线程迁移到 EntryList 中，并指定

EntryList 中的某个线程为 OnDeck 线程（一般是最先进去的那个线程）。

3. Owner 线程并不直接把锁传递给 OnDeck 线程，而是把锁竞争的权利交给 OnDeck，

OnDeck 需要重新竞争锁。这样虽然牺牲了一些公平性，但是能极大的提升系统的吞吐量，在

JVM 中，也把这种选择行为称之为“竞争切换”。

4. OnDeck 线程获取到锁资源后会变为 Owner 线程，而没有得到锁资源的仍然停留在 EntryList

中。如果 Owner 线程被 wait 方法阻塞，则转移到 WaitSet 队列中，直到某个时刻通过 notify

或者 notifyAll 唤醒，会重新进去 EntryList 中。

5. 处于 ContentionList、EntryList、WaitSet 中的线程都处于阻塞状态，该阻塞是由操作系统

来完成的（Linux 内核下采用 pthread\_mutex\_lock 内核函数实现的）。

6. Synchronized 是非公平锁。 Synchronized 在线程进入 ContentionList 时，等待的线程会先

尝试自旋获取锁，如果获取不到就进入 ContentionList，这明显对于已经进入队列的线程是

不公平的，还有一个不公平的事情就是自旋获取锁的线程还可能直接抢占 OnDeck 线程的锁

资源。

参考：https://blog.csdn.net/zqz\_zqz/article/details/70233767

7. 每个对象都有个 monitor 对象，加锁就是在竞争 monitor 对象，代码块加锁是在前后分别加

上 monitorenter 和 monitorexit 指令来实现的，方法加锁是通过一个标记位来判断的

8. synchronized 是一个重量级操作，需要调用操作系统相关接口，性能是低效的，有可能给线

程加锁消耗的时间比有用操作消耗的时间更多。

9. Java1.6，synchronized 进行了很多的优化，有适应自旋、锁消除、锁粗化、轻量级锁及偏向

锁等，效率有了本质上的提高。在之后推出的 Java1.7 与 1.8 中，均对该关键字的实现机理做

了优化。引入了偏向锁和轻量级锁。都是在对象头中有标记位，不需要经过操作系统加锁。

10. 锁可以从偏向锁升级到轻量级锁，再升级到重量级锁。这种升级过程叫做锁膨胀；

11. JDK 1.6 中默认是开启偏向锁和轻量级锁，可以通过-XX:-UseBiasedLocking 来禁用偏向锁