Java面试题-线程/多线程

一.创建线程的方式及实现

1.继承 Thread 类创建线程类

- 定义 Thread 类的子类，并重写该类的 `run` 方法，该 `run` 方法的方法体就代表了线程要完成的任务。因此把 `run()` 方法称为执行体。
- 创建 Thread 子类的实例，即创建了线程对象。
- 调用线程对象的 `start()` 方法来启动该线程。

2.通过 Runnable 接口创建线程类

- 定义 Runnable 接口的实现类，并重写该接口的 `run()` 方法，该 `run()` 方法的方法体同样是该线程的线程执行体。
- 创建 Runnable 实现类的实例，并依此实例作为 Thread 的 target 来创建 Thread 对象，该 Thread 对象才是真正的线程对象。
- 调用线程对象的 `start()` 方法来启动该线程。

3.通过 Callable 和 Future 创建线程

- 创建 Callable 接口的实现类，并实现 `call()` 方法，该 `call()` 方法将作为线程执行体，并且有返回值。
- 创建 Callable 实现类的实例，使用 FutureTask 类来包装 Callable 对象，该 FutureTask 对象封装了该 Callable 对象的 `call()` 方法的返回值。
- 使用 FutureTask 对象作为 Thread 对象的 target 创建并启动新线程。
- 调用 FutureTask 对象的 `get()` 方法来获得子线程执行结束后的返回值

二.不同方式创建线程的优缺点:

1.采用实现 Runnable、Callable 接口的方式创见多线程时：

- 优势是：线程类只是实现了 Runnable 接口或 Callable 接口，还可以继承其他类。在这种方式下，多个线程可以共享同一个 target 对象，所以非常适合多个相同线程来处理同一份资源的情况，从而可以将 CPU、代码和数据分开，形成清晰的模型，较好地体现了面向对象的思想。
- 劣势是：编程稍微复杂，如果要访问当前线程，则必须使用 Thread.currentThread() 方法。

2.使用继承 Thread 类的方式创建多线程时：

- 优势是：编写简单，如果需要访问当前线程，则无需使用 Thread.currentThread() 方法，直接使用 this 即可获得当前线程。
- 劣势是：线程类已经继承了 Thread 类，所以不能再继承其他父类。

三.线程的生命周期

新建(New)、就绪(Runnable)、运行(Running)、阻塞(Blocked)和死亡(Dead)5种状态

四.sleep()、join（）、yield（）有什么区别?

sleep()

`sleep()` 方法需要指定等待的时间，它可以让当前正在执行的线程在指定的时间内暂停执行，进入阻塞状态，该方法既可以让其他同优先级或者高优先级的线程得到执行的机会，也可以让低优先级的线程得到执行机会。但是 `sleep()` 方法不会释放“锁标志”，也就是说如果有 `synchronized` 同步块，其他线程仍然不能访问共享数据。

wait()

`wait()` 方法需要和 `notify()` 及 `notifyAll()` 两个方法一起介绍，这三个方法用于协调多个线程对共享数据的存取，所以必须在 `synchronized` 语句块内使用，也就是说，调用 `wait()`，`notify()` 和 `notifyAll()` 的任务在调用这些方法前必须拥有对象的锁。注意，它们都是 `Object` 类的方法，而不是 `Thread` 类的方法。

`wait()` 方法与 `sleep()` 方法的不同之处在于，`wait()` 方法会释放对象的“锁

标志”。当调用某一对象的 `wait()` 方法后，会使当前线程暂停执行，并将当前线程放入对象等待池中，直到调用了 `notify()` 方法后，将从对象等待池中移出任意一个线程并放入锁标志等待池中，只有锁标志等待池中的线程可以获取锁标志，它们随时准备争夺锁的拥有权。当调用了某个对象的 `notifyAll()` 方法，会将对象等待池中的所有线程都移动到该对象的锁标志等待池。

除了使用 `notify()` 和 `notifyAll()` 方法，还可以使用带毫秒参数的 `wait(long timeout)` 方法，效果是在延迟 timeout 毫秒后，被暂停的线程将被恢复到锁标志等待池。

此外，`wait()`，`notify()` 及 `notifyAll()` 只能在 `synchronized` 语句中使用，但是如果使用的是 `ReenTrantLock` 实现同步，该如何达到这三个方法的效果呢？解决方法是使用 `ReenTrantLock.newCondition()` 获取一个 `Condition` 类对象，然后 `Condition` 的 `await()`，`signal()` 以及 `signalAll()` 分别对应上面的三个方法。

yield()

`yield()` 方法和 `sleep()` 方法类似，也不会释放“锁标志”，区别在于，它没有参数，即 `yield()` 方法只是使当前线程重新回到可执行状态，所以执行 `yield()` 的线程有可能在进入到可执行状态后马上又被执行，另外 `yield()` 方法只能使同优先级或者高优先级的线程得到执行机会，这也和 `sleep()` 方法不同。

join()

`join()` 方法会使当前线程等待调用 `join()` 方法的线程结束后才能继续执行

五.说说线程安全问题

线程安全是多线程领域的问题，线程安全可以简单理解为一个方法或者一个实例可以在多线程环境中使用而不会出现问题。

在 Java 多线程编程当中，提供了多种实现 Java 线程安全的方式：

- 最简单的方式，使用 `Synchronization` 关键字
- 使用 `java.util.concurrent.atomic` 包中的原子类，例如 `AtomicInteger`

- 使用 `java.util.concurrent.locks` 包中的锁
- 使用线程安全的集合 `ConcurrentHashMap`
- 使用 `volatile` 关键字，保证变量可见性（直接从内存读，而不是从线程 `cache` 读）

六.synchronize 实现原理

同步代码块是使用 `monitorenter` 和 `monitorexit` 指令实现的，同步方法（在这看不出来需要看 JVM 底层实现）依靠的是方法修饰符上的 `ACC_SYNCHRONIZED` 实现。

七.synchronized 与 lock 的区别

- synchronized 和 lock 的用法区别:
    - synchronized(隐式锁)：在需要同步的对象中加入此控制，`synchronized` 可以加在方法上，也可以加在特定代码块中，括号中表示需要锁的对象。
    - lock(显示锁)：需要显示指定起始位置和终止位置。一般使用 `ReentrantLock` 类做为锁，多个线程中必须要使用一个 `ReentrantLock` 类做为对象才能保证锁的生效。且在加锁和解锁处需要通过 `lock()` 和 `unlock()` 显示指出。所以一般会在 `finally` 块中写 `unlock()` 以防死锁。
- synchronized 和 lock 性能区别
  `synchronized` 是托管给 JVM 执行的，而 `lock` 是 Java 写的控制锁的代码。在 **JDK 1.5** 中，`synchronize` 是性能低效的。因为这是一个重量级操作，需要调用操作接口，导致有可能加锁消耗的系统时间比加锁以外的操作还多。相比之下使用 Java 提供的 `Lock` 对象，性能更高一些。但是到了 **JDK 1.6，发生了变化。** `synchronize` 在语义上很清晰，可以进行很多优化，有适应自旋，锁消除，锁粗化，轻量级锁，偏向锁等等。导致在 **JDK 1.6** 上 `synchronize` 的性能并不比 `Lock` 差。
- synchronized 和 lock 机制区别
    - `synchronized` 原始采用的是 CPU 悲观锁机制，即线程获得的是独占锁。独占锁意味着其他线程只能依靠阻塞来等待线程释放锁。

- `Lock` 用的是乐观锁方式。所谓乐观锁就是，每次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。乐观锁实现的机制就是 CAS 操作（Compare and Swap）。

八.CAS 乐观锁

CAS 是一项乐观锁技术，当多个线程尝试使用 CAS 来同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。

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。无论哪种情况，它都会在 CAS 指令之前返回该位置的值。（在 CAS 的一些特殊情况下将仅返回 CAS 是否成功，而不提取当前值。）CAS 有效地说明了“我认为位置 V 应该包含值 A；如果包含该值，则将 B 放到这个位置；否则，不要更改该位置，只告诉我这个位置现在的值即可。”这其实和乐观锁的冲突检查 + 数据更新的原理是一样的。

九.ABA 问题

CAS 会导致“ABA问题”。

CAS 算法实现的一个重要前提是需要取出内存中某时刻的数据，而在下时刻比较并替换，那么在这个时间差类会导致数据的变化。

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，但是不代表这个过程就是没有问题的。

部分乐观锁的实现是通过版本号（version）的方式来解决 ABA 问题，乐观锁每次在执行数据的修改操作时，都会带上一个版本号，一旦版本号和数据的版本号一致就可以执行修改操作并对版本号执行 +1 操作，否则就执行失败。因为每次操作的版本号都会随之增加，所以不会出现 ABA 问题，因为版本号只会增加不会减少。

十.volatile 实现原理

- 在 JVM 底层 volatile 是采用“内存屏障”来实现的

- 缓存一致性协议（MESI协议）:它确保每个缓存中使用的共享变量的副本是一致的。其核心思想如下：当某个 CPU 在写数据时，如果发现操作的变量是共享变量，则会通知其他 CPU 告知该变量的缓存行是无效的，因此其他 CPU 在读取该变量时，发现其无效会重新从主存中加载数据

十一.讲讲线程池的实现原理

当提交一个新任务到线程池时，线程池的处理流程如下：

- 1.线程池判断核心线程池里的线程是否都在执行任务。如果不是，则创建一个新的工作线程来执行任务。如果核心线程池里的线程都在执行任务，则进入下个流程。
- 2.线程池判断工作队列是否已经满。如果工作队列没有满，则将新提交的任务存储在这个工作队列里。如果工作队列满了，则进入下个流程。
- 3.线程池判断线程池的线程是否都处于工作状态。如果没有，则创建一个新的工作线程来执行任务。如果已经满了，则交给饱和策略来处理这个任务。

十二.ThreadLocal 原理分析

ThreadLocal 提供了线程本地变量，它可以保证访问到的变量属于当前线程，每个线程都保存有一个变量副本，每个线程的变量都不同。ThreadLocal 相当于提供了一种线程隔离，将变量与线程相绑定。