并发

线程

1. 并行和并发有什么区别?

并发： 同一时间段，多个任务都在执行 (单位时间内不一定同时执行)，任务执行的状态。

并行： 单位时间内，多个任务同时执行。任务执行的一种策略。

2. 线程和进程的区别?

进程：就是程序的一次执行过程，系统运行的基本单位，运行一个程序就是一个进程从创建到运行再到消亡的过程。

线程：线程是一个比进程更小的程序单位，一个进程在运行过程中会产生多个线程。与进程不同的是，线程会共享进程的堆和方法区资源。 且每个线程拥有自己的程序计数器，本地方法栈，虚拟机栈。因此系统在线程间切换工作是开销小很多，线程也被称为轻量级进程。

Java中启动main函数时就是启动了一个jvm进程，而运行main函数所在的线程就是该进程中的一个进程，称为主线程。 一个 Java 程序的运行是 main 线程和多个其他线程同时运行。

进程间通信方式 [IPC](https//www.jianshu.com/p/c1015f5ffa74)

3. 守护线程是什么?

守护线程是为其他线程服务的线程; 所有非守护线程都执行完毕后jvm退出； 守护线程不能持有需要关闭的资源（如打开文件等），会导致数据丢失。

4. 创建线程有哪几种方式?

- 继承Thread类，重写run()；
- 实现一个Runnable接口，创建Thread实例的时候传入一个Runnable实例。
- 通过 Callable 和 FutureTask 创建线程

5. 实现Runnable接口和Callable接口的区别?

Runnable接口（Java 1.0）没有返回结果或抛出检查异常，Callable（Java 1.5）可以。

6. 线程的生命周期和状态?

- new：初始状态，未调用start
- Runnable：运行中，包含running和ready两个状态，由OS调度（yield）
- blocked：阻塞状态，阻塞于锁
- waiting：Object.wait()，需要notify()才能继续工作
- timed waiting：超时等待，sleep(long), wait(long)等，会自行返回
- terminated：终止状态，执行完毕

7. sleep() 和 wait() 有什么区别?

- **sleep没有释放锁，wait释放了锁**
- sleep通常用来执行暂停，时间结束线程自动苏醒。wait则需要通过别的线程notify或notifyAll才能苏醒

8. notify()和 notifyAll()有什么区别?

- 线程调用wait之后释放锁，进入等待池不参与锁竞争
- 调用notify之后，等待池中只有一个线程会进入锁池参与竞争，notifyAll则所有等待池中线程都进入锁池参与竞争

- notifyAll不会存在只唤醒同类线程的情况，一个生产者两个消费者

9. 线程的 run()和 start()有什么区别?

new一个线程后，调用start会启动该线程并进入Ready状态，当OS分配给它时间片后就可以运行了。start的源码会调用一个本地方法start0 进行一些执行线程的准备工作，然后自动调用run方法。run就是线程中一个普通方法，如果在main线程中直接调用就变成了调用了一个方法， 不在一个非主线程中去执行run则不是多线程工作。

线程池

1. 为什么要使用线程池?

池化技术可以：

- 降低资源消耗。避免重复创建销毁线程带来的资源消耗
- 提高响应速度。任务到达有可用线程直接进行处理
- 提高线程的可管理性。使用线程池可以对线程的资源分配进行统一调度，提高系统稳定性

2. 创建线程池有哪几种方式?

- Executors工具类创建
  - FixedThreadPool: 固定线程数量的线程池。处理任务是有空闲线程立即处理否任务放到任务队列等待执行；
  - SingleThreadPool：只有一个线程的线程池。用于顺序提交任务；
  - CachedThreadPool：线程数量可变的线程池。有空闲则复用，否则新建线程执行task；

实际内部都是调用ThreadPoolExecutor Constructor，允许创建的队列长度和线程数都是Integer.MAX_VALUE，因此高并发时非常容易导致OOM

- ThreadPoolExecutor：

```
ThreadPoolExecutor(
    int corePoolSize,  // 线程池保有的最小线程数
    int maximumPoolSize,  // 可创建最大线程数
    long keepAliveTime, // 空闲线程最大存活时间
    TimeUnit unit, // 存活时间单位
    BlockingQueue<Runnable> workQueue, // 任务队列
    ThreadFactory threadFactory, // 线程工厂对象，自定创建线程
    RejectedExecutionHandler handler // 自定义拒绝策略
    )
```

`keepAliveTime`：线程池线程数>corePoolSize, 且某线程空闲时间超过这个时间就要被回收

`handler`：

- AbortPolicy：默认的拒绝策略，throws RejectedExecutionException
- CallerRunsPolicy：提交任务的线程自己去执行该任务
- DiscardPolicy：直接丢弃任务，不抛出任何异常
- DiscardOldestPolicy：丢弃最老的任务，加入新的任务

3. 线程池的状态?

JDK注释：

```
The runState provides the main lifecyle control, taking on values:

  RUNNING:  Accept new tasks and process queued tasks
  SHUTDOWN: Don't accept new tasks, but process queued tasks
  STOP:     Don't accept new tasks, don't process queued tasks,
            and interrupt in-progress tasks
  TIDYING:  All tasks have terminated, workerCount is zero,
            the thread transitioning to state TIDYING
            will run the terminated() hook method
  TERMINATED: terminated() has completed
```

4. 线程池中 submit()和 execute()方法有什么区别?

- execute()参数是Runnable，submit()有三种参数：Runnable，Callable， Runnable和T result；
- execute无返回值，submit返回一个Future类型对象，可以Future.get()获得执行结果或捕捉异常。

5. 在 java 程序中怎么保证多线程的运行安全?

并发编程的三大特性导致的安全问题：

- 原子性：一个或多个操作在CPU执行过程中不能被中断
- 可见性：线程对方法区共享变量的修改是互相可见的
- 有序性：程序按照代码先后顺序执行的

解决方法：

- Atomic原子类、synchronized、LOCK解决原子性问题
- synchronized、volatile、LOCK解决可见性问题
- Happens-Before规则解决有序性

锁

1. **（重要）锁升级过程？（synchronized为什么性能提高了？）**

1, 2

2. 死锁以及如何解决?

当程序中两个或以上的线程互相请求对象持有的资源时就会发生死锁，此时这些线程将一直处于等待其他线程释放资源的的状态， 没有外力干涉，将无法打破僵持状态，使程序正常运行。

死锁产生的4个条件：

- 互斥。资源只能被一个线程占用
- 占有且等待
- 不可抢占
- 循环等待

解决：破坏条件之一即可。

- 一次性占有所有资源，破坏"占有且等待"；
- 主动释放。申请不到其他资源则主动释放自己的，破坏不可抢占；
- 按序申请资源，破坏"循环等待"

3. ThreadLocal有什么作用？有哪些使用场景?

ThreadLocal类主要解决的就是让每个线程绑定自己的值，可以将ThreadLocal类形象的比喻成存放数据的盒子，盒子中可以存储每个线程的私有数据。

如果你创建了一个ThreadLocal变量，那么访问这个变量的每个线程都会有这个变量的本地副本，这也是ThreadLocal变量名的由来。他们可以使用 get（） 和 set（） 方法来获取默认值或将其值更改为当前线程所存的副本的值，从而避免了线程安全问题。

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。 spring 在处理数据库事务问题的时候，就用了 ThreadLocal 为每个线程存储了各自的数据库连接 Connection。

4. synchronized的作用？怎么用？说一下 synchronized 底层实现原理？

作用：解决多线程之间访问资源的同步性，保证其修饰的代码和方法在同一时间只能被一个线程执行

怎么用：

- 对象锁:

对象声明之后可以new多个不同的实例，且这些实例在jvm都是自己的引用地址和堆内存空间， 因此使用同一实例的线程才会受到锁影响，不同实例不会。

主要通过对非静态变量加锁、对当前对象实例加锁、对非静态方法加锁实现。

- 类锁：

类信息和静态变量都是存在jvm方法区的，所有线程共享方法区，因此在同一时刻只有一个线程能够访问类锁中的方法或代码块。

主要通过对静态变量加锁、对类属性加锁、对静态方法加锁实现。

`尽量不要给String加锁，因为String常量池具有缓存功能`

Q: 手撕个DCL?

```
public class Singleton {
    // volatile禁止jvm指令重排，使得 singletonInstance = new Singleton();
    // 分配内存空间 -> 初始化 -> 指向内存地址 能够正常完成
    private volatile static Singleton singletonInstance;

    // private只有自己能实例化自己
    private Singleton() {
    }

    public static Singleton getInstance() {
        // 未实例化才进入锁代码，已创建则直接返回提高性能
        if (singletonInstance == null) {
            // 类锁，只有一个线程获得实例权
            synchronized (Singleton.class) {
                // 阻塞线程到这里需要再次判断是否已实例化，防止重复创建
                if (singletonInstance == null) {
                    singletonInstance = new Singleton();
                }
            }
        }
        return singletonInstance;
    }
}
```

实现原理： synchronized实现属于jvm层面。

- 对代码块加锁：查看字节码可以知道，是通过monitorenter 和 monitorexit 指令获取线程的执行权。 **当执行 monitorenter 指令时，线程试图获取锁也就是获取 monitor(monitor对象存在于每个Java对象的对象头中，synchronized 锁便是通过这种方式获取锁的，也是为什么Java中任意对象可以作为锁的原因) 的持有权。** ~~当计数器为0则可以成功获取，获取后将锁计数器设为1也就是加1。相应的在执行 monitorexit 指令后，将锁计数器设为0，表明锁被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到锁被另外一个线程释放为止。~~
- 对方法加锁：jvm通过判断是否有ACC_SYNCHRONIZED flag来判断是否为同步方法

5. synchronized和volatile的区别是什么?

- 功能上来说，两者是互补的。volatile主要是解决变量在多个线程之间的可见性，只能修饰**变量**，强制到主存而不是寄存器读写。synchronized则是为了解决 多线程之间访问资源的同步性，可以修饰**变量、方法、代码块**。
- synchronized会阻塞线程，volatile不会。
- volatile无法保证数据的原子性，可以保证可见性，synchronized可以保证原子性和可见性。

6. synchronized 和 Lock 有什么区别?

- 实现层面。前者是关键词，jvm层面实现；后者是接口，代码层面；
- 是否自动放锁。前者在执行完成和抛出异常会自动放锁，后者需要显示finally{}代码块放锁；
- 取锁机制。前者拿不到锁会一直等待，无法获知是否取锁成功；后者可以设置等待时间，通过tryLock获得是否加锁成功；
- 复杂性。前者加锁可重入、不可中断、不公平；后者可重入、可中断、可公平可不公平、细分读写锁提高效率。

7. synchronized 和 ReentrantLock 区别是什么？应用场景?

- 都是可重入锁。同一线程可重复获得锁，锁计数器+1即可。
- 前者jvm层面实现，后者API层面（需要 lock() 和 unlock() 方法配合 try/finally 语句块来完成）实现。前者隐藏细节，后者可以看代码实现。
- ReentrantLock多了一些高级功能（灵活性很高）
  - 等待可中断
  - 可指定锁是公平还是非公平的
  - 可实现选择性通知：synchronized的wait和notify/notifyAll实现等待/通知功能，是有jvm选择对象，且前者随机从锁池选择，后者是通知所有锁池线程进入等待池竞争资源，会造成很大的效率问题。 ReentrantLock可以使用Condition接口在一个Lock对象中实现多个Condition实例（对象监视器）， 线程可以选择注册在某个Condition中，从而可选择性的进行线程通知。

8. Atomic原理？顺带讲下CAS原理应用?

Atomic类就是具有原子/原子操作（不可中断）特征的类。使用Atomic类之后不加锁也可以保证线程安全。

```
    // setup to use Unsafe.compareAndSwapInt for updates
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;

    static {
        try {
            valueOffset = unsafe.objectFieldOffset
                (AtomicInteger.class.getDeclaredField("value"));
        } catch (Exception ex) { throw new Error(ex); }
    }

    private volatile int value;
```

CAS + volatile + unsafe（对照上AtomicInteger源码）：

- CAS：（V,E,U），V等于E才更新V为U，否则失败；
- volatile：保证每次unsafe取得的值都是最新值，因为直接走主存中拿；
- unsafe：native方法，取value偏移地址。

Java利用CAS的乐观锁、原子性的特性高效解决了多线程的安全性问题，例如JDK1.8中的集合类ConcurrentHashMap、关键字volatile、ReentrantLock等。

缺点：

- 高并发的时候会有很多CAS操作失败
- CAS操作是自旋的，因此长时间失败会造成CPU开销极大
- ABA问题(加本号解决)

9. 乐观锁 悲观锁?

乐观锁

- 乐观锁总是假设最好的情况，每次去操作数据都认为不会被别的线程修改数据，所以在每次操作数据的时候都不会给数据加锁，即在线程对数据进行操作的时候，别的线程不会阻塞仍然可以对数据进行操作，**只有在需要更新数据的时候才会去判断数据是否被别的线程修改过，如果数据被修改过则会拒绝操作并且返回错误信息给用户**。 悲观锁
- 悲观锁总是假设最坏的情况，每次去操作数据时候都认为会被的线程修改数据，所以在每次操作数据的时候都会给数据加锁，让别的线程无法操作这个数据，**别的线程会一直阻塞直到获取到这个数据的锁**。这样的话就会影响效率，比如当有个线程发生一个很耗时的操作的时候，别的线程只是想获取这个数据的值而已都要等待很久。

10. AQS原理

从ReentrantLock的实现看AQS的原理及应用